

Docker Compose : Étude de cas

Table des matières

I. API de Deep Learning dockerisée	3
II. Exercice : Quiz	8
III. Base de données dockerisée	9
IV. Exercice : Quiz	10
V. Docker Compose	11
VI. Exercice : Quiz	16
VII. Essentiel	17
VIII. Auto-évaluation	17
A. Exercice	17
B. Test	18
Solutions des exercices	19

I. API de Deep Learning dockerisée

Durée : 1 H 45

Environnement de travail : visual Studio / Visual code

Pré-requis : introduction à Docker Compose, Deep Learning, API, Docker, Python

Contexte

Une application peut être constituée de plusieurs containers exécutant différents services. Il peut être compliqué de démarrer et fastidieux de gérer manuellement les conteneurs, c'est pourquoi Docker a créé un outil utile qui permet d'accélérer le processus : Docker Compose. Docker Compose étend Docker Engine : il se comporte comme un panneau de contrôle pour exécuter des applications Docker multi-containers.

Docker Compose fonctionne en appliquant des règles définies dans un fichier « **docker-compose.yaml** ». Le fichier « **yaml** » est le fichier de configuration où vous décrivez les services de l'application et les règles spécifiant l'ordre et les conditions d'exécution.

Une fois Docker Compose en route, vous pouvez également vérifier l'état d'un service et lancer des commandes ponctuelles. Docker Compose s'avère très pratique : en une commande, vous pouvez démarrer, arrêter ou reconstruire tous les services.

Le modèle de classification

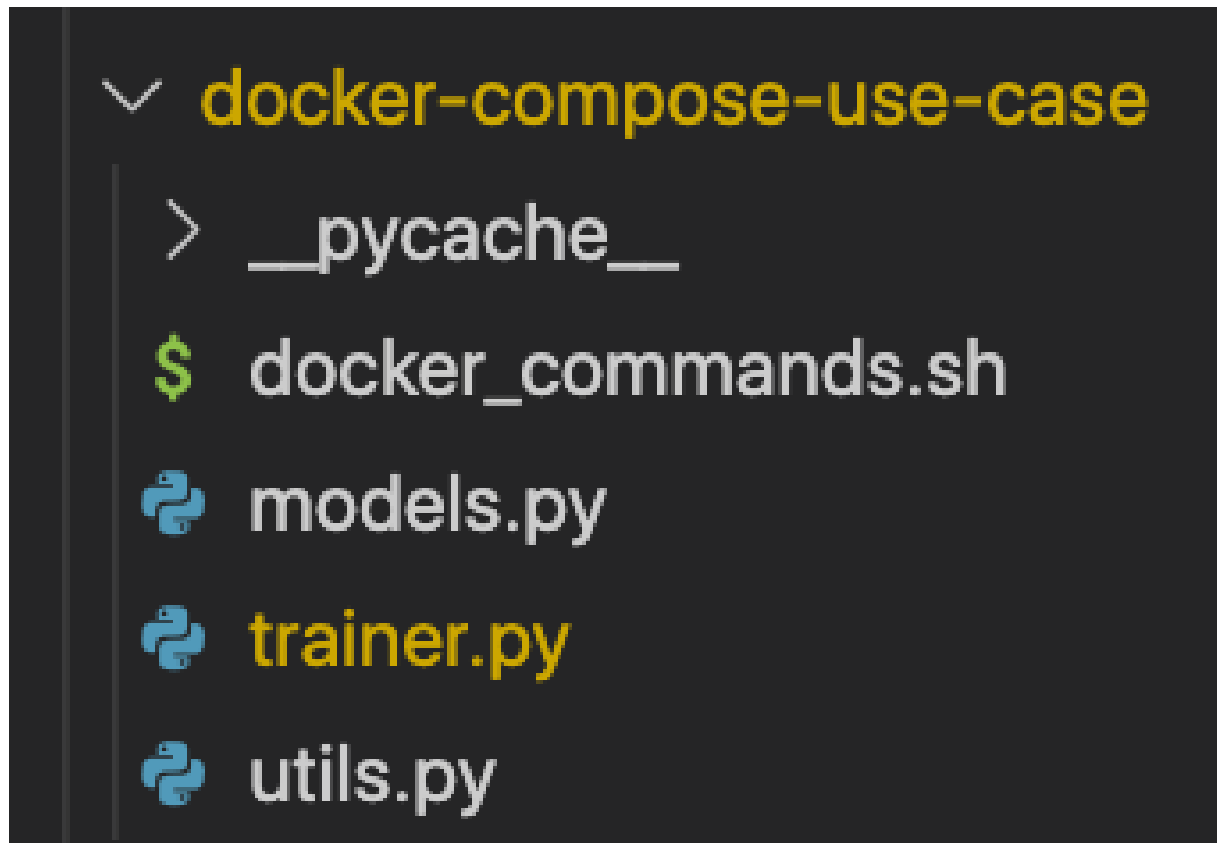
On crée un modèle de classification simple qui pourra être lancé sous forme d'application FastAPI dockerisée. On pourra alors soumettre des arguments à l'application pour lancer différents jobs d'entraînement : *epochs*, taux d'apprentissage et pourcentage d'oubli (Dropout). Ce service peut stocker ses résultats dans un autre service de stockage de données type Redis.

L'intérêt d'utiliser Docker Compose est de centraliser le management de la base de données dockerisée et de l'API de Deep Learning. Si ce n'était pas le cas, on devrait démarrer les containers séparément, tout en spécifiant comment ils communiquent entre eux, comme ici :

```
1 # Cette commande exécute le premier container
2 # Le paramètre (flag) '-t' permet d'avoir un pseudo-terminal
3 # (pour exécuter des commandes dans le container une fois lancé).
4 # Le paramètre '-i' permet d'activer le mode interactif,
5 # qui redirige tous les messages sur les sorties standards (permet de voir les logs).
6 # Le paramètre '-rm' permet de supprimer automatiquement le container à la fin de l'exécution
7 docker run -it my-first-container
8
9 # Cette commande exécute le second container et le connecte au premier container
10 # le flag -rm le détruira à sa sortie
11 # le flag -d exécute le container sous forme de daemon
12 # (en fond)
13 docker run -it --rm --name my-second-container \
14 --link my-first-container -d
15
```

Grâce à Docker Compose, on peut également spécifier aux containers un comportement particulier : ordre de lancement ou état du container.

Le modèle de Deep Learning est un modèle simple qui réalise un entraînement de Deep Learning. La structure du projet se présente comme ceci :



« *utils.py* »

```
1 import tensorflow as tf
2
3 def normalize_img(image, label):
4     """Normalizes images: `uint8` -> `float32`.
5     retourne un 2-tuple : image normalisée, étiquette
6     """
7     return tf.cast(image, tf.float32) / 255., label
8
```

« *Trainer.py* »

```
1 import argparse
2 import tensorflow as tf
3 import tensorflow_datasets as tfds
4 from utils import normalize_img
5 from models import model_zoo # Ce module comprend les modèles (nano,small,big)
6
7 from fastapi import FastAPI
8 import numpy as np
9 import time
10
11 app = FastAPI()
12
13 @app.get("/predict/")
14 async def read_user_item(model_name: str, lr: float, n_epoch:int):
15
16     # Vous pourrez alors exécuter l'API avec la commande ci-dessous
```

```

17 #http://127.0.0.1:8000/predict/?model=%22nano%22&lr=0.001&n_epoch=10
18
19 # Le résultat de la requête est stocké dans request item
20 request_item = {"model": model_name, "lr": lr, "n_epoch":n_epoch}
21
22 # La classe Trainer lance un entraînement avec les paramètres
23 t = Trainer(n_epoch=request_item['n_epoch'],
24 lr = request_item['lr'],
25 dl_model_name = model_name
26 )
27 # Après complétion, vous retournez le résultat d'entraînement
28 return t.train()
29
30
31 def get_config():
32     '''
33     fonction qui récupère les arguments
34     depuis la ligne de commande
35     '''
36
37     # créer un parseur d'arguments
38     opt = argparse.ArgumentParser()
39
40     opt.add_argument("--model", default='nano', help="Vous choisissez le type de modèle retenu
(nano, small,big)")
41     opt.add_argument("--lr", default='nano', help="Vous choisissez le taux d'apprentissage")
42     opt.add_argument("--n epoch", default=10, help="Vous choisissez le nombre d'époch")
43
44     args = vars(opt.parse_args())
45
46     # choix du modèle:
47     if args['model'] == 'nano':
48         model = model_nano
49     elif args['model'] == 'small':
50         model = model_small
51     else:
52         model = model_big
53
54     # choix du taux d'apprentissage:
55     lr = float(args['lr'])
56
57     # choix du nombre d'epochs:
58     n_epochs = float(args['n_epoch'])
59
60     return n_epochs,lr,model
61
62 class Trainer():
63     def __init__(self, n_epoch,lr, dl_model_name):
64         # extraction des paramètres
65         self.train_epochs, self.train_lr, self.train_model_name = n_epoch,lr, dl_model_name
66         # création du modèle (non compilé)
67         self.model = model_zoo[self.train_model_name]
68         # préparation des jeux de données
69         self.ds_val, self.ds_train = self.prepare_data()
70
71     def prepare_data(self):
72         '''
73         prépare les données (entraînement - test)

```

```

74     '''
75     # chargement du jeu de données
76     (self.ds_train, self.ds_test), self.ds_info = tfds.load(
77         'mnist',
78         split=['train', 'test'],
79         shuffle_files=True,
80         as_supervised=True,
81         with_info=True,
82     )
83
84     # ds_train (est un objet de type dataset au sens de tensorflow)
85     # le dataset va être modifié par la fonction normalize_img
86     self.ds_train = self.ds_train.map(
87         normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
88     self.ds_train = self.ds_train.cache()
89     self.ds_train = self.ds_train.shuffle(self.ds_info.splits['train'].num_examples)
90     self.ds_train = self.ds_train.batch(128)
91     self.ds_train = self.ds_train.prefetch(tf.data.AUTOTUNE)
92
93     # ds_test (est un objet de type dataset au sens de tensorflow)
94     # le dataset va être modifié par la fonction normalize_img
95     self.ds_test = self.ds_test.map(
96         normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
97     self.ds_test = self.ds_test.batch(128)
98     self.ds_test = self.ds_test.cache()
99     self.ds_test = self.ds_test.prefetch(tf.data.AUTOTUNE)
100
101     return self.ds_test, self.ds_train
102
103 def train(self, n_epoch=10, l_r=0.001):
104     """
105     fonction responsable de l'entraînement
106     """
107     # on compile le modèle avec un optimiseur de type Adam
108     # on choisit la fonction de perte :SparseCategoricalCrossentropy
109     # car les labels sont stockés sous forme d'entiers.
110
111     start_time = time.time()
112     self.model.compile(
113         optimizer=tf.keras.optimizers.Adam(l_r),
114         loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
115         metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
116     )
117     # entraînement
118     self.history = self.model.fit(
119         self.ds_train,
120         epochs=n_epoch,
121         validation_data=self.ds_val,
122     )
123     end_time = time.time()
124     delta = time.time() - start_time
125
126     # L'API retourne différentes métriques
127     return {'loss':self.history.history['loss'],
128           'min_loss':np.min(self.history.history['loss']),
129           'max_loss':np.max(self.history.history['loss']),
130           'val_loss':self.history.history['val_loss'],
131           'min_val_loss':np.min(self.history.history['val_loss']),

```

```

132         'max_val_loss': np.max(self.history.history['val_loss']),
133         'run_time': delta,
134         'model': self.train_model_name
135     }
136

```

Le code s'exécute et vous pouvez observer le déroulement dans le terminal :

```

INFO: Will watch for changes in these directories: ['/usr/local/Cellar/apache-spark/3.1.2/scripts/CLASSIF_DEEP_terra_nn/docker-compose-use-case']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [47930] using statreload
2022-01-27 22:23:02.104271: I tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations: AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
INFO: Started server process [47932]
INFO: Waiting for application startup.
INFO: Application startup complete.

```

Vous pouvez vérifier le bon fonctionnement dans votre navigateur. Vous obtiendrez ces informations :

```

1 {"loss":
  [0.02864069677889347,0.022059984505176544,0.01928926818072796,0.01505349762737751,0.013676436617970467,0.012500000000000001,0.011500000000000001,0.010500000000000001,0.009500000000000001,0.008500000000000001,0.007500000000000001,0.006500000000000001,0.005500000000000001,0.004500000000000001,0.003500000000000001,0.002500000000000001,0.001500000000000001,0.000500000000000001,0.0000000000000000]
}

```

Le « dockerfile »

```

1 # Dockerfile
2 # qui permet de configurer facilement un GPU
3 # lorsqu'il est de type nvidia
4 #FROM nvidia/cuda:10.2-devel-ubuntu18.04
5 FROM ubuntu:18.04
6 RUN DEBIAN_FRONTEND=noninteractive apt-get update && DEBIAN_FRONTEND=noninteractive apt-get
   install -y python3-opencv git python3 python3-pip
7
8 # copy the local requirements.txt file to the
9 # /app/requirements.txt in the container
10 # (the /app dir will be created)
11 COPY ./requirements.txt /app/requirements.txt
12 COPY ./*.py /app/
13
14 # install the packages from the requirements.txt file in the container
15 RUN pip install -r /app/requirements.txt
16 # expose the port that uvicorn will run the app
17 EXPOSE 8000
18 # copy the local app/ folder to the /app folder in the container
19 # set the working directory in the container to be the /app
20 WORKDIR /app
21 # execute the command python main.py (in the WORKDIR) to start the app
22 CMD ["python3", "trainer.py"]
23

```

L'image Docker

Vous pouvez créer l'image Docker en exécutant la commande suivante :

```
1 docker build . -t docker_dl
```

```
[+] Building 42.5s (11/11) FINISHED
=> [internal] load build definition from Dockerfile                                0.1s
=> => transferring dockerfile: 953B                                             0.0s
=> [internal] load .dockerignore                                                 0.0s
=> => transferring context: 28                                                  0.0s
=> [internal] load metadata for docker.io/library/ubuntu:18.04                 2.4s
=> [1/6] FROM docker.io/library/ubuntu:18.04@sha256:37b7471c1945a2a12e5a57488ee4e3e216a8369d0b9ee1ec2e41db9c2c1e3d22  0.0s
=> [internal] load build context                                                0.0s
=> => transferring context: 124B                                               0.0s
=> CACHED [2/6] RUN DEBIAN_FRONTEND=noninteractive apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -y python3-opencv 0.0s
=> CACHED [3/6] COPY ./requirements.txt /app/requirements.txt                  0.0s
=> CACHED [4/6] COPY ./*.py /app/                                              0.0s
=> [5/6] RUN pip3 install -r /app/requirements.txt                             21.4s
=> [6/6] WORKDIR /app                                                         0.0s
=> exporting to image                                                         18.3s
=> => exporting layers                                                         18.3s
=> writing image sha256:42b6b9c25f47bd4fd8ee617478c774600e7ae88332f2ee0f96783c0f8742fd82 0.0s
=> => naming to docker.io/library/docker_dl                                   0.0s
```

Exercice : Quiz

[solution n°1 p.21]

Question 1

On utilise Docker Compose pour organiser de manière plus flexible l'exécution des images Docker.

- ☐ Vrai
- ☐ Faux

Question 2

Le fichier Docker (« **dockerfile** ») possède la même extension que le fichier « **docker-compose** ».

- ☐ Vrai
- ☐ Faux

Question 3

Dans le cas de notre application, on doit avoir deux « **dockerfiles** » et deux fichiers « **docker-compose.yaml** »

- ☐ Vrai
- ☐ Faux

Question 4

Les fichiers Docker « **dockerfile** » peuvent être situés dans des répertoires différents que vous pourrez spécifier dans le fichier « **docker-compose** ».

- ☐ Vrai
- ☐ Faux

Question 5

On exécute d'abord la commande `docker` avec l'argument : « **build** », puis vous exécutez la commande **docker avec l'argument «compose** » et enfin la commandes docker avec l'argument « **run** ».

- ☐ Vrai
- ☐ Faux

III. Base de données dockerisée

Définition La base de données Redis

Redis est une base de données souvent utilisée dans le contexte industriel pour des solutions de streaming dont l'architecture inclut Apache Kafka et / ou Amazon Kinesis. C'est la spécialiste des bases de données pour ingérer, traiter et analyser des données en temps réel.

Redis existe également en service managé par AWS et d'autres fournisseurs Cloud. C'est une base de données qui fonctionne comme un « *cache* » et qui permet une grande vitesse d'écriture et de lecture (latence inférieure à la milliseconde).

Redis est un choix idéal pour des besoins d'analyse en temps réel tels que l'analyse des réseaux sociaux et la collecte d'informations provenant d'objets connectés.

Interaction avec l'image Redis

Pour vérifier le fonctionnement de cette base de données Redis, on peut créer une application Flask qui va interagir avec la base dockerisée Redis pour en vérifier le bon fonctionnement.

Exécution du container Redis :

```
1 docker run --name my-Redis -p 6379:6379 -d Redis
```

Cette commande effectue les opérations suivantes :

- Extraction de la dernière image Redis du Docker Hub (l'endroit où toutes les images tierces publiques sont stockées).
- Création et exécution du container nommé « **my-Redis** ».
- Mapping du port 6379 de l'ordinateur portable vers le port 6379 à l'intérieur du container (6379 est le port par défaut de Redis et peut être modifié).

Micro application Flask : **Flask_test.py**

```
1 import Redis
2 from flask import Flask
3 import time
4
5 app = Flask(__name__)
6 cache = Redis.Redis(host='localhost', port=6379)
7
8 def get_hit_count():
9     retries = 5
10    while True:
11        try:
12            return cache.incr('hits')
13        except Redis.exceptions.ConnectionError as exc:
14            if retries == 0:
15                raise exc
16            retries -= 1
17            time.sleep(0.5)
18
19 @app.route('/')
20 def hello():
21     count = get_hit_count()
22     return 'Hello World! vous m\'avez cliqué {} fois.\n'.format(count)
23
24
25 @app.route('/Redis_data')
```

```

26 def hello():
27     results = {"data_heure_stockage": str(time.time())}
28     cache.mset(results)
29     return cache.get("data_heure_stockage")
30
31 if __name__ == '__main__':
32     app.run(debug=True, port=8888)
33

```

```
1 python3 ./flask_test.py
```

La base de données Redis vit dans un container dont le port 6379 a été associé à la machine hôte (votre ordinateur). L'application Flask n'est pas dockerisée. Elle vient interroger le container.

En se rendant à l'adresse <http://127.0.0.1:8888/>, on obtient le message suivant :

← → ↻ ⓘ 127.0.0.1:8888

Hello World! vous m'avez cliqué 4 fois.

Ce message prouve que l'image Redis fonctionne.

Exercice : Quiz

[solution n°2 p.22]

Question 1

Pour des opérations intensives en écriture de type *real-time analytics*, une base de données de type MySQL est recommandée.

- ☐ Vrai
- ☐ Faux

Question 2

On ne peut pas conteneuriser de bases de données autres que Redis.

- ☐ Vrai
- ☐ Faux

Question 3

La base de données Redis est supprimée à chaque fois que le container est stoppé

- ☐ Vrai
- ☐ Faux

Question 4

On peut définir un volume commun pour les deux services.

- ☐ Vrai
- ☐ Faux

Question 5

On peut imaginer un système dans lequel on a un service avec deux containers de l'image de Deep Learning et deux containers de type base de données. Il suffit de le mentionner dans le fichier « **docker-compose** ».

- ☐ Vrai
- ☐ Faux

V. Docker Compose

Définition

Dans les cas précédents, les containers Redis et l'application de Deep Learning vivent dans des containers différents. Dans le projet cible, le container de Deep Learning va communiquer directement avec le container Redis. Docker Compose crée un réseau qui permet aux deux containers de communiquer directement.

Le fichier docker-compose

Le fichier « **docker-compose** » dans sa version la plus simple requiert un « **dockerfile** » pour le service « **api** ». Pour ce service, on réalise un mapping des ports 8 000 (hôte) à 8 000 (container). L'image correspondant au service « **Redis** » est par défaut l'image « **Redis:alpine** ».

docker-compose.yml

```
1 version: "3.9"
2 services:
3   api:
4     build: .
5     ports:
6       - "8000:8000"
7   Redis:
8     image: "Redis:alpine"
9
```

main.py :

```
1 from train import Trainer
2 from fastapi import FastAPI
3 import uvicorn
4 import Redis
5
6 # On crée une application du type FastAPI
7 app = FastAPI()
8
9 # On se connecte à la base de données Redis
10 # le service Redis va être créé grâce à docker-compose
11 cache = Redis.Redis(host='Redis', port=6379)
12
13 # Route pour les prédictions
14 @app.get("/predict/")
15 async def read_user_item(model_name: str, lr: float, n_epoch:int):
16
17     # Vous pourrez alors exécuter l'API avec la commande ci-dessous
18     #http://127.0.0.1:8000/predict/?model=%22nano%22&lr=0.001&n_epoch=10
19
20     # Le résultat de la requête est stocké dans request item
21     request_item = {"model": model_name, "lr": lr, "n_epoch":n_epoch}
22
23     # La classe Trainer lance un entraînement avec les paramètres
24     t = Trainer(n_epoch=request_item['n_epoch'],
```

```

25     lr = request_item['lr'],
26     dl_model_name = model_name
27 )
28 # Après complétion, vous retournez le résultat d'entraînement
29
30 results = t.train()
31 try:
32
33     cache.mset(results)
34
35 except Redis.exceptions.ConnectionError as exc:
36     print(exc)
37     return results
38 # fonction qui récupère la configuration
39
40
41 @app.get("/get_data/")
42 async def get_user_item(model_name: str):
43     return cache.get(model_name)
44
45 if __name__ == "__main__":
46     # Uvicorn est une implémentation de serveur ASGI ultra-rapide,
47     # utilisant uvloop et httptools. ... ASGI devrait aider à activer
48     # un écosystème de frameworks Web Python qui sont très compétitifs
49     # par rapport à Node and Go en termes d'obtention d'un débit élevé dans des contextes liés
    aux E/S.
50     uvicorn.run('main:app', host='0.0.0.0', port=8000, reload=True, root_path="/predict")
51

```

models.py

```

1 import tensorflow as tf
2
3 model_nano = tf.keras.models.Sequential([
4     # on définit la première couche avec
5     # la taille des données d'entrée
6     tf.keras.layers.Flatten(input_shape=(28, 28)),
7     # on crée une couche complètement connectée
8     # on ajoute une couche finale sans fonction d'activation
9     tf.keras.layers.Dense(64, activation='relu'),
10    tf.keras.layers.Dense(10)
11 ])
12
13 model_small = tf.keras.models.Sequential([
14     # on définit la première couche avec
15     # la taille des données d'entrée
16     tf.keras.layers.Flatten(input_shape=(28, 28)),
17     # on crée une couche complètement connectée
18     # on ajoute une couche finale sans fonction d'activation
19     tf.keras.layers.Dense(128, activation='relu'),
20     tf.keras.layers.Dense(10)
21 ])
22
23 model_big = tf.keras.models.Sequential([
24     # on définit la première couche avec
25     # la taille des données d'entrée
26     tf.keras.layers.Flatten(input_shape=(28, 28)),
27     # on crée une couche complètement connectée
28     # on ajoute une couche finale sans fonction d'activation

```

```

29 tf.keras.layers.Dense(256, activation='relu'),
30 tf.keras.layers.Dense(10)
31 ])
32
33 model_zoo = {"nano":model_nano,
34             "small":model_small,
35             "big": model_big}
36

```

train.py

```

1 import time
2 import tensorflow as tf
3 import argparse
4 import tensorflow as tf
5 import tensorflow_datasets as tfds
6 import numpy as np
7 from models import model_zoo
8 from utils import normalize_img
9 from models import model_zoo
10
11
12 def get_config():
13     '''
14     fonction qui récupère les arguments
15     depuis la ligne de commande
16     '''
17
18     # créer un parseur d'arguments
19     opt = argparse.ArgumentParser()
20
21     opt.add_argument("--model", default='nano', help="Vous choisissez le type de modèle retenu
22 (nano, small, big)")
23     opt.add_argument("--lr", default='nano', help="Vous choisissez le taux d'apprentissage")
24     opt.add_argument("--n_epoch", default=10, help="Vous choisissez le nombre d'époch")
25
26     args = vars(opt.parse_args())
27     print(args)
28
29     # choix du modèle:
30     if args['model'] == 'nano':
31         model = model_zoo.model_nano
32     elif args['model'] == 'small':
33         model = model_zoo.model_small
34     else:
35         model = model_zoo.model_big
36
37     # choix du taux d'apprentissage:
38     lr = float(args['lr'])
39
40     # choix du nombre d'epochs:
41     n_epochs = float(args['n_epoch'])
42
43     return n_epochs, lr, model
44
45 class Trainer():
46     def __init__(self, n_epoch, lr, dl_model_name, run_name):
47         # extraction des paramètres
48         self.train_epochs, self.train_lr, self.train_model_name = n_epoch, lr, dl_model_name

```

```

48     # création du modèle (non compilé)
49     self.model = model_zoo[self.train_model_name]
50     # préparation des jeux de données
51     self.ds_val, self.ds_train = self.prepare_data()
52     # le nom de l'expérience (doit être unique)
53     self.run_name = run_name
54
55     def prepare_data(self):
56         '''
57         prépare les données (entraînement - test)
58         '''
59         # chargement du jeu de données
60         (self.ds_train, self.ds_test), self.ds_info = tfds.load(
61             'mnist',
62             split=['train', 'test'],
63             shuffle_files=True,
64             as_supervised=True,
65             with_info=True,
66         )
67
68         # ds_train (est un objet de type dataset au sens de tensorflow)
69         # le dataset va être modifié par la fonction normalize_img
70         self.ds_train = self.ds_train.map(
71             normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
72         self.ds_train = self.ds_train.cache()
73         self.ds_train = self.ds_train.shuffle(self.ds_info.splits['train'].num_examples)
74         self.ds_train = self.ds_train.batch(128)
75         self.ds_train = self.ds_train.prefetch(tf.data.AUTOTUNE)
76
77         # ds_test (est un objet de type dataset au sens de tensor flow)
78         # le dataset va être modifié par la fonction normalize_img
79         self.ds_test = self.ds_test.map(
80             normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
81         self.ds_test = self.ds_test.batch(128)
82         self.ds_test = self.ds_test.cache()
83         self.ds_test = self.ds_test.prefetch(tf.data.AUTOTUNE)
84
85         return self.ds_test, self.ds_train
86
87     def train(self, n_epoch=10, l_r=0.001):
88         """
89         fonction responsable de l'entraînement
90         """
91         # on compile le modèle avec un optimiseur de type Adam
92         # on choisit la fonction de perte :SparseCategoricalCrossentropy
93         # car les labels sont stockés sous forme d'entiers.
94
95         start_time = time.time()
96         self.model.compile(
97             optimizer=tf.keras.optimizers.Adam(l_r),
98             loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
99             metrics=['accuracy'],
100         )
101         # entraînement
102         self.history = self.model.fit(
103             self.ds_train,
104             epochs=n_epoch,
105             validation_data=self.ds_val,

```

```

106     )
107     end_time = time.time()
108     delta = time.time() - start_time
109
110
111     # on stocke chaque valeur pour chaque epoch dans un dictionnaire
112     # dans deux dictionnaires :
113     # val_loss
114     # loss
115     val_loss = {key:value for (key,value) in enumerate(self.history.history['val_loss'])}
116     loss = {key:value for (key,value) in enumerate(self.history.history['loss'])}
117
118     # L'API retourne différentes métriques
119     return {str(self.run_name):
120             {'loss':str(loss),
121              'min_loss': float(np.min(self.history.history['loss'])),
122              'max_loss':float(np.max(self.history.history['loss'])),
123              'val_loss': str(val_loss),
124              'min_val_loss':float(np.min(self.history.history['val_loss'])),
125              'max_val_loss':float(np.max(self.history.history['val_loss'])),
126              'run_time': delta,
127              'model': self.train_model_name
128             }
129     }
130

```

utils.py

```

1 import tensorflow as tf
2
3 def normalize_img(image, label):
4     """Normalizes images: `uint8` -> `float32`.
5     retourne un 2-tuple : image normalisée, étiquette
6     """
7     return tf.cast(image, tf.float32) / 255., label
8

```

requirements.txt

```

1 fastapi==0.70.0
2 Flask==2.0.1
3 tensorflow==2.6.0
4 keras ==2.6.*
5 tensorflow-datasets==4.4.0
6 uvicorn==0.15.0
7 Redis==3.5.3
8

```

dockerfile

```

1 # install python in the container
2 # qui permet de configurer facilement un GPU
3 # lorsqu'il est de type nvidia
4 #FROM nvidia/cuda:10.2-devel-ubuntu18.04
5 FROM ubuntu:20.04
6 RUN DEBIAN_FRONTEND=noninteractive apt-get update && DEBIAN_FRONTEND=noninteractive apt-get
   install -y python3-opencv git python3 python3-pip
7
8 # copy the local requirements.txt file to the
9 # /app/requirements.txt in the container
10 # (the /app dir will be created)
11 COPY ./requirements.txt /app/requirements.txt

```

```

12 COPY ./*.py /app/
13
14 # install the packages from the requirements.txt file in the container
15 RUN pip3 install -r /app/requirements.txt
16 # expose the port that uvicorn will run the app
17 EXPOSE 8000
18 # copy the local app/ folder to the /app folder in the container
19 # set the working directory in the container to be the /app
20 WORKDIR /app
21 # execute the command python main.py (in the WORKDIR) to start the app
22 CMD ["python3", "./main.py"]
23

```

Exercise : Quiz

[solution n°3 p.22]

```

1 version: "2.1"
2 services:
3   studi:
4     image: linuxserver/studi
5     container_name: studi
6     volumes:
7       - /home/user/appdata/studi:/config
8     environment:
9       - PUID=1000
10      - PGID=1000
11      - TZ=Europe/London
12     ports:
13       - 80:80
14       - 443:443
15     restart: unless-stopped
16   nginx:
17     image: linuxserver/nginx
18     container_name: nginx
19     environment:
20       - PUID=1000
21       - PGID=1000
22       - TZ=Europe/London
23     volumes:
24       - /home/user/appdata/nginx:/config
25     ports:
26       - 81:80
27       - 444:443
28     restart: unless-stopped
29   mariadb:
30     image: linuxserver/mariadb
31     container_name: mariadb
32     environment:
33       - PUID=1000
34       - PGID=1000
35       - MYSQL_ROOT_PASSWORD=ROOT_ACCESS_PASSWORD
36       - TZ=Europe/London
37     volumes:
38       - /home/user/appdata/mariadb:/config
39     ports:
40       - 3306:3306
41     restart: unless-stopped
42

```


Question 1

On a deux services dans ce « **docker-compose** ».

- ☐ Vrai
- ☐ Faux

Question 2

Un des services est une base de données de type « **mariadb** ».

- ☐ Vrai
- ☐ Faux

Question 3

Les services possèdent les mêmes variables d'environnement. On aurait dû les spécifier une seule fois en haut du fichier « **docker-compose** ».

- ☐ Vrai
- ☐ Faux

Question 4

Les services vont fonctionner indéfiniment et relancer des containers lorsque ceux-ci seront stoppés.

- ☐ Vrai
- ☐ Faux

Question 5

Les services « **nginx** », « **studi** » et « **mariadb** » partagent le même volume sur l'ordinateur hôte « **/config** ».

- ☐ Vrai
- ☐ Faux

VII. Essentiel

Docker Compose est un outil permettant de définir et d'exécuter des applications Docker multi-containers. Avec Docker Compose, vous utilisez un fichier **yaml** pour configurer les services de votre application. Ensuite, avec une seule commande, vous créez et vous démarrez tous les services de votre configuration.

Vous pouvez également utiliser Docker Compose pour manager des systèmes multi-services via un unique fichier « **docker-compose.yaml** ». La flexibilité des services rend la création d'une application beaucoup plus simple et itérative, au contraire des applications sous forme de silos.

VIII. Auto-évaluation

A. Exercice

Vous désirez pousser l'expérience plus loin et voudriez exécuter plusieurs containers de prédiction mais garder un seul container pour la base de données. Ces containers pourraient par exemple permettre d'entraîner des modèles d'entraînement différents simultanément.

Question 1

[solution n°4 p.23]

Est-ce possible ? Pouvez-vous lancer plusieurs containers de prédiction ? Si oui, pouvez-vous le faire avec Docker Compose ? À quel niveau sera faite la modification ? Sur les « **dockerfiles** » ? Sur le « **docker-compose.yml** » ? Au démarrage des containers via « **docker run** » ?

```
1 version: "3.9"
2 services:
3   api:
4     build: .
5     ports:
6       - "8000-8005:8000"
7   Redis:
8     image: "Redis:alpine"
9
```

Vous devez créer une solution de machine learning devant d'abord détecter des véhicules puis les classer. L'application doit également réaliser la lecture de plaques d'immatriculation. On vous demande de concevoir une architecture qui permette de détecter les véhicules et la position de leur plaques d'immatriculation, de lire ces plaques et de stocker en base de données de type Redis.

Question 2

[solution n°5 p.24]

Docker Compose est-il une bonne solution ? Quelle architecture proposeriez-vous ?

B. Test

Exercice 1 : Quiz

[solution n°6 p.25]

```
1 version: "2.1"
2 services:
3   studi:
4     image: linuxserver/studi
5     container_name: studi
6     volumes:
7       - /home/user/appdata/studi:/config
8     environment:
9       - PUID=1000
10      - PGID=1000
11      - TZ=Europe/London
12     ports:
13       - 80:80
14       - 443:443
15     restart: unless-stopped
16
```

Question 1

Dans ce « **docker-compose** », on manage deux services.

- ☐ Vrai
- ☐ Faux

Question 2

« **PUID** », « **PGID** » et « **TZ** » sont des variables d'environnement définies pour chaque image.

- ☐ Vrai
- ☐ Faux

Question 3

Ce « **docker-compose** » attache les ports de la machine hôte à celui du ou des containers créés (ports 80 et 443).

- ☐ Vrai
- ☐ Faux

Question 4

L'image et le container ne peuvent pas porter le même nom.

- ☐ Vrai
- ☐ Faux

Question 5

On ne va pas spécifier le nombre de containers (instances par services) au niveau du fichier « **docker-compose.yaml** », mais plutôt lors de la commande « **docker compose up** » comme indiqué ici :

```
1 docker-compose up -d --scale app=5
```

- ☐ Vrai
- ☐ Faux

Solutions des exercices


Exercice p. 8 Solution n°1

Question 1

On utilise Docker Compose pour organiser de manière plus flexible l'exécution des images Docker.

☐ Vrai

☒ Faux


 On veut organiser de manière plus flexible l'exécution des containers Docker entre eux.

Question 2

Le fichier Docker (« **dockerfile** ») possède la même extension que le fichier « **docker-compose** ».

☐ Vrai

☒ Faux

 Le fichier « **docker-compose** » est un fichier « **yaml** », le « **dockerfile** » n'a pas d'extension.

Question 3

Dans le cas de notre application, on doit avoir deux « **dockerfiles** » et deux fichiers « **docker-compose.yaml** »

☐ Vrai

☒ Faux


 Il y a un unique fichier « **docker-compose** » pour l'orchestration des containers.

Question 4

Les fichiers Docker « **dockerfile** » peuvent être situés dans des répertoires différents que vous pourrez spécifier dans le fichier « **docker-compose** ».

☒ Vrai

☐ Faux

 La localisation des fichiers « **dockerfile** » est décrite dans le fichier « **docker-compose** ». Ces fichiers peuvent être dans des répertoires différents.

Question 5

On exécute d'abord la commande `docker` avec l'argument : « **build** », puis vous exécutez la commande **docker** avec l'argument « **compose** » et enfin la commandes docker avec l'argument « **run** ».

☐ Vrai

☒ Faux

 Dans le cas de Docker Compose, vous n'exécutez plus ni « **docker build** » ni « **docker run** ».

Vous utilisez la commande « **docker-compose** ». Par exemple, on utilise « **docker-compose up** » pour exécuter l'ensemble des services.


Exercice p. 10 Solution n°2

Question 1

Pour des opérations intensives en écriture de type *real-time analytics*, une base de données de type MySQL est recommandée.

☐ Vrai

☒ Faux


 Une base de données très performante en écriture on préférera les bases de données types in-memory telles que Redis.

Question 2

On ne peut pas conteneuriser de bases de données autres que Redis.

☐ Vrai

☒ Faux


 Vous pouvez conteneuriser tout type de bases de données compatible avec Docker.

Question 3

La base de données Redis est supprimée à chaque fois que le container est stoppé

☒ Vrai

☐ Faux


 C'est une base de données qui vit dans la mémoire RAM du container. Pour persister le contenu de cette base de données, vous pouvez créer un volume.

Question 4

On peut définir un volume commun pour les deux services.

☒ Vrai

☐ Faux


 Vous pouvez très bien associer un répertoire unique sur l'ordinateur hôte à des répertoires dans des containers associés à deux services différents. Pour la lisibilité, mieux vaut séparer ces répertoires.

Question 5

On peut imaginer un système dans lequel on a un service avec deux containers de l'image de Deep Learning et deux containers de type base de données. Il suffit de le mentionner dans le fichier « **docker-compose** ».

☐ Vrai

☒ Faux

 Le nombre de containers associés à tel ou tel service doit être spécifié à l'exécution via « **docker compose up** ».

Exercice p. 16 Solution n°3

Question 1

On a deux services dans ce « **docker-compose** ».

☐ Vrai

☒ Faux


 On a trois services : « **studi** », « **nginx** » et « **mariadb** ».

Question 2

Un des services est une base de données de type « **mariadb** ».

☒ Vrai

☐ Faux


 On a trois services : « **studi** », « **nginx** » et « **mariadb** ». L'affirmation de l'énoncé est vraie.

Question 3

Les services possèdent les mêmes variables d'environnement. On aurait dû les spécifier une seule fois en haut du fichier « **docker-compose** ».

☐ Vrai

☒ Faux


 Les variables d'environnement sont spécifiques aux services. On les spécifie dans chaque zone concernée.

Question 4

Les services vont fonctionner indéfiniment et relancer des containers lorsque ceux-ci seront stoppés.

☐ Vrai

☒ Faux


 C'est le contraire : avec « **unless-stopped** », les containers ne seront pas relancés si on les arrête, à l'inverse de « **always** ».

Question 5

Les services « **nginx** », « **studi** » et « **mariadb** » partagent le même volume sur l'ordinateur hôte « **/config** ».

☐ Vrai

☒ Faux

 Trois volumes sont mappés respectivement au répertoire « **/config** » dans les containers de chacun des services. En d'autres termes, ces services vont nécessiter trois containers. Chacun des répertoires de l'ordinateur hôte sera mappé à un répertoire différent.

Le service est appelé « **api** » dans le fichier yaml, mais il pourrait s'agir de n'importe quel composant individuel de votre déploiement : une partie *frontend*, une base de données, etc. En fonction de votre service, vous devrez peut-être modifier d'autres parties du script, mais pour autoriser la création de plusieurs containers Nginx qui écoutent tous sur leur port interne 80. Cependant, l'hôte écoute sur des ports allant de 8 000 à 8 005 et redirige le trafic de chaque port unique vers l'un des instances Nginx.

Vous pouvez alors lancer la commande :

```
1 docker-compose up -d --scale api=5
```

p. 18 Solution n°5

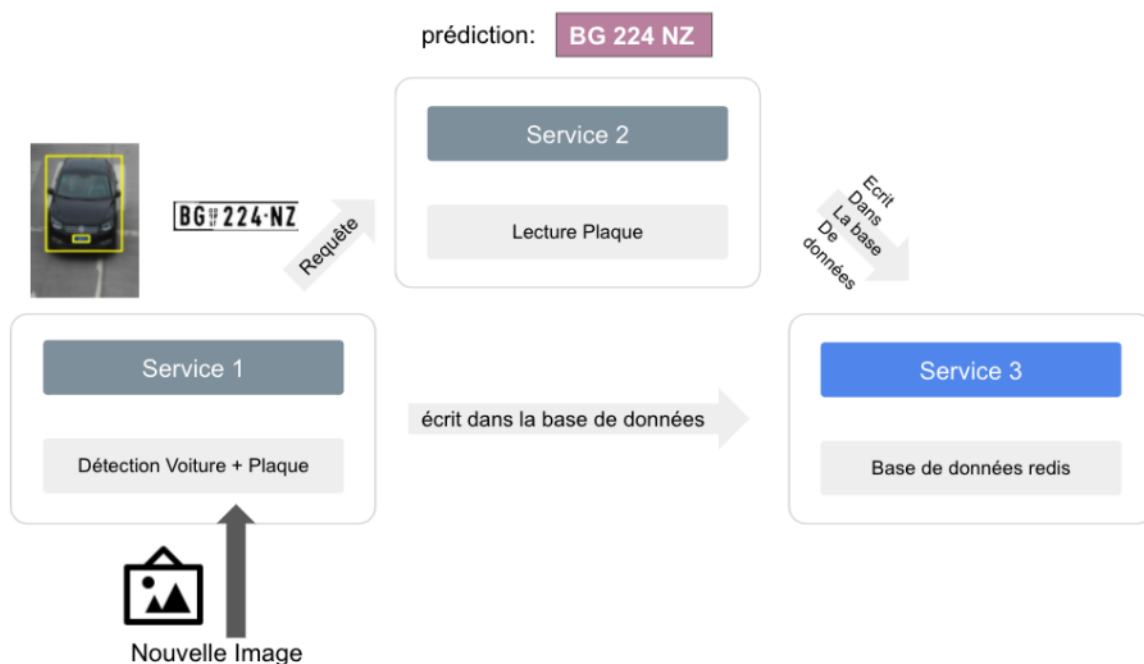
Docker Compose serait une bonne solution car on pourrait avoir :

- Un service de détection de véhicules et de localisation de plaques d'immatriculation
- Un service de lecture de plaques d'immatriculation
- Un service de stockage (base de données type Redis)

On peut si besoin créer trois « **dockerfiles** », un pour chaque service. Les trois services pourraient être interconnectés de cette façon :

- Les services 1 et 2 peuvent être implémentés comme des API (de la même manière qu'avec FastAPI).
- Vous interrogez le service 1, comme vous le feriez avec une API. Le service 1 peut sauvegarder dans la base de données Redis prévue à cet effet, car d'après le « **docker-compose** », les services seront dans le même réseau.
- Le service 1 peut interroger le service 2 avec l'image détournée pour produire la prédiction de la plaque d'immatriculation à l'aide d'un appel de type « **POST** » contre l'API « **REST** » du service 2.
- Le service 2 peut alors sauvegarder dans la base de données cette lecture de plaques.

L'intérêt de Docker Compose dans ce cas est sa capacité à augmenter le nombre de services de détection de voiture et de plaques, ou le service de la lecture de la plaque si l'un des services est plus rapide que l'autre.




```
1 version: "3.9"
2 services:
3   api_detectionvehiculeplaques:
4     build: detectionvehiculeplaques
5     ports:
6       - "8000-8005:8000"
7   api_lectureplaques:
8     build: lectureplaques
9     ports:
10      - "7000-7005:7000"
11 db:
12   image: "Redis:alpine"
13
```


Exercice p. 18 Solution n°6

Question 1

Dans ce « **docker-compose** », on manage deux services.

☐ Vrai

☒ Faux


 Dans ce cas on a un seul service appelé « **studi** ».

Question 2

« **PUID** », « **PGID** » et « **TZ** » sont des variables d'environnement définies pour chaque image.

☐ Vrai

☒ Faux


 Les variables d'environnement sont définies pour les containers, pas les images.

Question 3

Ce « **docker-compose** » attache les ports de la machine hôte à celui du ou des containers créés (ports 80 et 443).

☒ Vrai

☐ Faux


 Dans la clause « **ports:** », vous avez mappé le port 80 avec le port 80 et le port 443 avec le port 443.

Question 4

L'image et le container ne peuvent pas porter le même nom.

☐ Vrai

☒ Faux

 Vous pouvez nommer l'image et le « **docker-compose** » de la manière qui vous plaît. Il est tout de même conseillé de privilégier la lisibilité.


Question 5

On ne va pas spécifier le nombre de containers (instances par services) au niveau du fichier « **docker-compose.yml** », mais plutôt lors de la commande « **docker compose up** » comme indiqué ici :

```
1 docker-compose up -d --scale app=5
```

☒ Vrai

☐ Faux

 Vous ne spécifiez pas le nombre de services au niveau du fichier yaml, mais plutôt lors de l'exécution de vos services via la commande « **docker-compose up** ».