towards data science

Build, Train and Deploy A Real-World Flower Classifier of 102 Flower Types

With TensorFlow 2.3, Amazon SageMaker Python SDK 2.5.x and Custom SageMaker Training & Serving Docker Containers

Juv Chan Sep 10, 2020 · 10 min read

Lotus Flower taken at Summer Palace (颐和园), Beijing China in 2008.

Introduction

I love flowers. The lotus flower above is one of my most favorite flower photos taken during my visit to the Summer Palace Beijing in 2008. Since I am a developer and enjoy learning and working on artificial intelligence and cloud projects, I decide to write this blog post to share my project on building a real-world flower classifier with TensorFlow, Amazon SageMaker and Docker.

This post shows step-by-step guide on:

- Using ready-to-use **Flower dataset** from **TensorFlow Datasets**.
- Using **Transfer Learning** for feature extraction from a pre-trained model from **TensorFlow Hub**.
- Using tf.data API to build input pipelines for the dataset split into training, validation and test datasets.
- Using tf.keras API to build, train and evaluate the model.
- Using Callback to define **early stopping** threshold for model training.
- Preparing training script to train and export the model in SavedModel format for deploy with **TensorFlow** and **Amazon SageMaker Python SDK**.
- Preparing inference code and configuration to run the **TensorFlow Serving ModelServer** for serving the model.
- Building custom **Docker Containers** for training and serving the TensorFlow model with Amazon SageMaker Python SDK and **SageMaker TensorFlow Training Toolkit** in **Local mode**.

The project is available to the public at:

juvchan/amazon-sagemaker-tensorflow-custom-containers

This project shows step-by-step guide on how to build a real-world flower classifier of 102 flower types using...

github.com

Setup

Below is the list of system, hardware, software and Python packages that are used to develop and test the project.

- Ubuntu 18.04.5 LTS
- Docker 19.03.12
- Python 3.8.5
- Conda 4.8.4
- NVIDIA GeForce RTX 2070
- NVIDIA Container Runtime Library 1.20
- NVIDIA CUDA Toolkit 10.1
- sagemaker 2.5.3
- sagemaker-tensorflow-training 20.1.2
- tensorflow-gpu 2.3.0
- tensorflow-datasets 3.2.1
- tensorflow-hub 0.9.0
- tensorflow-model-server 2.3.0
- jupyterlab 2.2.6
- Pillow 7.2.0
- matplotlib 3.3.1

Flower Dataset

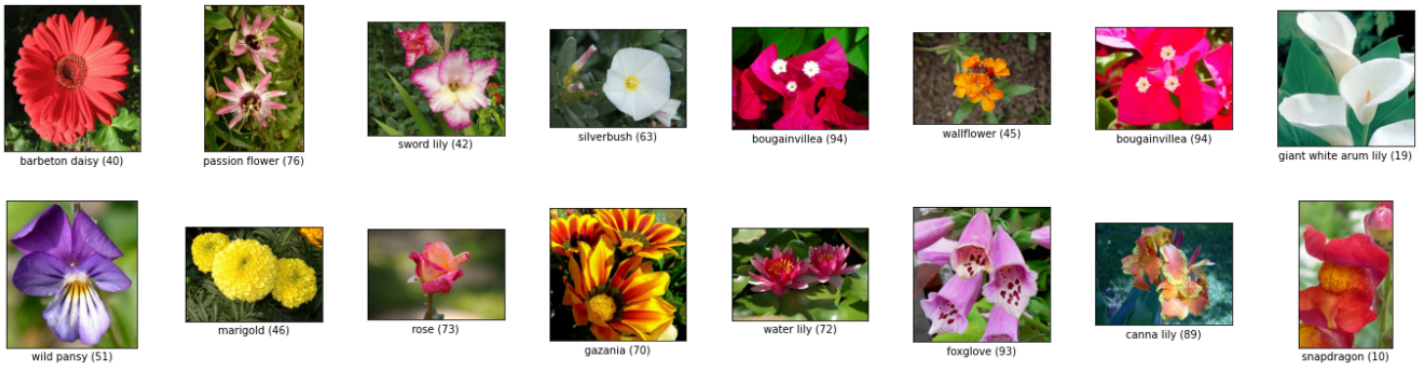
TensorFlow Datasets (TFDS) is a collection of public datasets ready to use with TensorFlow, JAX and other machine learning frameworks. All TFDS datasets are exposed as tf.data.Datasets, which are easy to use for high-performance input pipelines.

There are a total of **195** ready-to-use datasets available in the TFDS to date. There are **2 flower datasets** in TFDS: **oxford_flowers102**, **tf_flowers**

The **oxford_flowers102** dataset is used because it has both larger dataset size and larger number of flower categories.

```
ds_name = 'oxford_flowers102'
splits = ['test', 'validation', 'train']
ds, info = tfds.load(ds_name, split = splits, with_info=True)
(train_examples, validation_examples, test_examples) = ds

print(f"Number of flower types {info.features['label'].num_classes}")
print(f"Number of training examples:
{tf.data.experimental.cardinality(train_examples)}")
print(f"Number of validation examples:
{tf.data.experimental.cardinality(validation_examples)}")
print(f"Number of test examples:
{tf.data.experimental.cardinality(test_examples)}\n")

print('Flower types full list:')
print(info.features['label'].names)

tfds.show_examples(train_examples, info, rows=2, cols=8)
```

```
Number of flower types: 102
Number of training examples: 6149
Number of validation examples: 1020
Number of test examples: 1020

Flower types full list:
['pink primrose', 'hard-leaved pocket orchid', 'canterbury bells', 'sweet pea', 'english marigold', 'tiger lily', 'moon orchid', 'bird of
paradise', 'monkshood', 'globe thistle', 'snapdragon', "colt's foot", 'king protea', 'spear thistle', 'yellow iris', 'globe-flower', 'pur
ple coneflower', 'peruvian lily', 'balloon flower', 'giant white arum lily', 'fire lily', 'pincushion flower', 'fritillary', 'red ginge
r', 'grape hyacinth', 'corn poppy', 'prince of wales feathers', 'stemless gentian', 'artichoke', 'sweet william', 'carnation', 'garden ph
lox', 'love in the mist', 'mexican aster', 'alpine sea holly', 'ruby-lipped cattleya', 'cape flower', 'great masterwort', 'siam tulip',
'lenten rose', 'barbeton daisy', 'daffodil', 'sword lily', 'poinsettia', 'bolero deep blue', 'wallflower', 'marigold', 'buttercup', 'oxey
e daisy', 'common dandelion', 'petunia', 'wild pansy', 'primula', 'sunflower', 'pelargonium', 'bishop of llandaff', 'gaura', 'geranium',
'orange dahlia', 'pink-yellow dahlia?', 'cautleya spicata', 'japanese anemone', 'black-eyed susan', 'silverbush', 'californian poppy', 'o
steospermum', 'spring crocus', 'bearded iris', 'windflower', 'tree poppy', 'gazania', 'azalea', 'water lily', 'rose', 'thorn apple', 'mor
ning glory', 'passion flower', 'lotus', 'toad lily', 'anthurium', 'frangipani', 'clematis', 'hibiscus', 'columbine', 'desert-rose', 'tree
mallow', 'magnolia', 'cyclamen', 'watercress', 'canna lily', 'hippeastrum', 'bee balm', 'ball moss', 'foxglove', 'bougainvillea', 'camell
ia', 'mallow', 'mexican petunia', 'bromelia', 'blanket flower', 'trumpet creeper', 'blackberry lily']
```



oxford_flowers dataset summary and samples

Create SageMaker TensorFlow Training Script

Amazon SageMaker allows users to use training script or inference code in the same way that would be used outside SageMaker to run custom training or inference algorithm. One of the differences is that the training script used with Amazon SageMaker could make use of the **SageMaker Containers Environment Variables**, e.g. **SM_MODEL_DIR**, **SM_NUM_GPUS**, **SM_NUM_CPUS** in the SageMaker container.

> *Amazon SageMaker always uses Docker containers when running scripts, training algorithms or deploying models. Amazon SageMaker provides containers for its built-in algorithms and pre-built Docker images for some of the most common machine learning frameworks. You can also create your own container images to manage more advanced use cases not addressed by the containers provided by Amazon SageMaker.*

The custom training script is as shown below:

```
import argparse
import numpy as np
import os
import logging
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds


EPOCHS = 5
BATCH_SIZE = 32
LEARNING_RATE = 0.001
DROPOUT_RATE = 0.3
EARLY_STOPPING_TRAIN_ACCURACY = 0.995
TF_AUTOTUNE = tf.data.experimental.AUTOTUNE
TF_HUB_MODEL_URL = 'https://tfhub.dev/google/inaturalist/inception_v3/feature_vector/4
TF_DATASET_NAME = 'oxford_flowers102'
IMAGE_SIZE = (299, 299)
SHUFFLE_BUFFER_SIZE = 473
MODEL_VERSION = '1'


class EarlyStoppingCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        if(logs.get('accuracy') > EARLY_STOPPING_TRAIN_ACCURACY):
            print(
                f"\nEarly stopping at {logs.get('accuracy'):.4f} > {EARLY_STOPPING_TRA
            self.model.stop_training = True


def parse_args():
    parser = argparse.ArgumentParser()

    # hyperparameters sent by the client are passed as command-line arguments to the s
    parser.add_argument('--epochs', type=int, default=EPOCHS)
    parser.add_argument('--batch_size', type=int, default=BATCH_SIZE)
    parser.add_argument('--learning_rate', type=float, default=LEARNING_RATE)

    # model_dir is always passed in from SageMaker. By default this is a S3 path under
    parser.add_argument('--model_dir', type=str)
    parser.add_argument('--sm_model_dir', type=str,
                        default=os.environ.get('SM_MODEL_DIR'))
    parser.add_argument('--model_version', type=str, default=MODEL_VERSION)

```

```
def set_gpu_memory_growth():
    gpus = tf.config.list_physical_devices('GPU')

    if gpus:
        print("\nGPU Available.")
        print(f"Number of GPU: {len(gpus)}")
        try:
            for gpu in gpus:
                tf.config.experimental.set_memory_growth(gpu, True)
                print(f"Enabled Memory Growth on {gpu.name}\n")
                print()
        except RuntimeError as e:
            print(e)

    print()


def get_datasets(dataset_name):
    tfds.disable_progress_bar()

    splits = ['test', 'validation', 'train']
    splits, ds_info = tfds.load(dataset_name, split=splits, with_info=True)
    (ds_train, ds_validation, ds_test) = splits

    return (ds_train, ds_validation, ds_test), ds_info


def parse_image(features):
    image = features['image']
    image = tf.image.resize(image, IMAGE_SIZE) / 255.0
    return image, features['label']


def training_pipeline(train_raw, batch_size):
    train_preprocessed = train_raw.shuffle(SHUFFLE_BUFFER_SIZE).map(
        parse_image, num_parallel_calls=TF_AUTOTUNE).cache().batch(batch_size).prefetc

    return train_preprocessed


def test_pipeline(test_raw, batch_size):
    test_preprocessed = test_raw.map(parse_image, num_parallel_calls=TF_AUTOTUNE).cach
```

```
    base_model = hub.KerasLayer(TF_HUB_MODEL_URL,
                                input_shape=IMAGE_SIZE + (3,), trainable=False)

    early_stop_callback = EarlyStoppingCallback()

    model = tf.keras.Sequential([
        base_model,
        tf.keras.layers.Dropout(DROPOUT_RATE),
        tf.keras.layers.Dense(NUM_CLASSES, activation='softmax')
    ])

    model.compile(optimizer=optimizer,
                  loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    model.summary()

    model.fit(train_batches, epochs=args.epochs,
              validation_data=val_batches,
              callbacks=[early_stop_callback])

    return model


if __name__ == "__main__":
    args, _ = parse_args()
    batch_size = args.batch_size
    epochs = args.epochs
    learning_rate = args.learning_rate
    print(
        f"\nBatch Size = {batch_size}, Epochs = {epochs}, Learning Rate = {learning_ra

    set_gpu_memory_growth()

    (ds_train, ds_validation, ds_test), ds_info = get_datasets(TF_DATASET_NAME)
    NUM_CLASSES = ds_info.features['label'].num_classes

    print(
```

```
146     model = create_model(train_batches, validation_batches, learning_rate)
147     eval_results = model.evaluate(test_batches)
148 
149     for metric, value in zip(model.metrics_names, eval_results):
150         print(metric + ': {:.4f}'.format(value))
151 
152     export_path = os.path.join(args.sm_model_dir, args.model_version)
153     print(
154         f'\nModel version: {args.model_version} exported to: {export_path}\n')
155 
156     model.save(export_path)
```

Transfer Learning with TensorFlow Hub (TF-Hub)

TensorFlow Hub is a library of reusable pre-trained machine learning models for transfer learning in different problem domains. For this flower classification problem, we evaluate the **pre-trained image feature vectors** based on different image model architectures and datasets from TF-Hub as below for transfer learning on the **oxford_flowers102** dataset.

- ResNet50 Feature Vector
- MobileNet V2 (ImageNet) Feature Vector
- Inception V3 (ImageNet) Feature Vector
- Inception V3 (iNaturalist) Feature Vector

In the final training script, the **Inception V3 (iNaturalist) feature vector** pre-trained model is used for transfer learning for this problem because it performs the best

compared to the others above **(~95% test accuracy over 5 epochs without fine-tune)**. This model uses the Inception V3 architecture and trained on the **iNaturalist (iNat) 2017** dataset of **over 5,000** different species of plants and animals from https://www.inaturalist.org/. In contrast, the **ImageNet 2012** dataset has only 1,000 classes which has very few flower types.

Serve Flower Classifier with TensorFlow Serving

TensorFlow Serving is a flexible, high-performance machine learning models serving system, designed for production environment. It is part of **TensorFlow Extended (TFX)**, an end-to-end platform for deploying production Machine Learning (ML) pipelines. The **TensorFlow Serving ModelServer binary** is available in two variants: **tensorflow-model-server** and **tensorflow-model-server-universal**. The **TensorFlow Serving ModelServer** supports both gRPC APIs and RESTful APIs.

In the inference code, the **tensorflow-model-server** is used to serve the model via RESTful APIs from where it is exported in the SageMaker container. It is a fully optimized server that uses some platform specific compiler optimizations and should be the preferred option for users. The inference code is as shown below:

```
#!/usr/bin/env python

# This file implements the hosting solution, which just starts TensorFlow Model Serving
import subprocess
import os

TF_SERVING_DEFAULT_PORT = 8501
MODEL_NAME = 'flowers_model'
MODEL_BASE_PATH = '/opt/ml/model'


def start_server():
    print('Starting TensorFlow Serving.')

    # link the log streams to stdout/err so they will be logged to the container logs
    subprocess.check_call(
        ['ln', '-sf', '/dev/stdout', '/var/log/nginx/access.log'])
    subprocess.check_call(
        ['ln', '-sf', '/dev/stderr', '/var/log/nginx/error.log'])

    # start nginx server
    nginx = subprocess.Popen(['nginx', '-c', '/opt/ml/code/nginx.conf'])

    # start TensorFlow Serving
    # https://www.tensorflow.org/serving/api_rest#start_modelserver_with_the_rest_api_e
    tf_model_server = subprocess.call(['tensorflow_model_server',
                                       '--rest_api_port=' +
                                       str(TF_SERVING_DEFAULT_PORT),
                                       '--model_name=' + MODEL_NAME,
                                       '--model_base_path=' + MODEL_BASE_PATH])


# The main routine just invokes the start function.
if __name__ == '__main__':
    start_server()
```

serve.py hosted with ❤ by **GitHub** **view raw**

Build Custom Docker Image and Container for SageMaker Training and Inference

Amazon SageMaker utilizes Docker containers to run all training jobs and inference endpoints. Amazon SageMaker provides pre-built Docker containers that support machine learning frameworks such as **SageMaker Scikit-learn Container**, **SageMaker XGBoost Container**, **SageMaker SparkML Serving Container**, **Deep Learning Containers** (TensorFlow, PyTorch, MXNet and Chainer) as well as **SageMaker RL (Reinforcement Learning) Container** for training and inference. These pre-built SageMaker containers should be sufficient for general purpose machine learning training and inference scenarios.

There are some scenarios where the pre-built SageMaker containers are unable to support, e.g.

- Using unsupported machine learning framework versions
- Using third-party packages, libraries, run-times or dependencies which are not available in the pre-built SageMaker container
- Using custom machine learning algorithms

Amazon SageMaker supports user-provided custom Docker images and containers for the advanced scenarios above. Users can use any programming language, framework or packages to build their own Docker image and container that are tailored for their machine learning scenario with Amazon SageMaker.

In this flower classification scenario, custom Docker image and containers are used for the training and inference because the pre-built SageMaker TensorFlow containers do not have the packages required for the training, i.e. **tensorflow_hub** and **tensorflow_datasets**. Below is the **Dockerfile** used to build the custom Docker image.

```
# Copyright 2020 Juv Chan. All Rights Reserved.
FROM tensorflow/tensorflow:2.3.0-gpu

LABEL maintainer="Juv Chan <juvchan@hotmail.com>"

RUN apt-get update && apt-get install -y --no-install-recommends nginx curl
RUN pip install --no-cache-dir --upgrade pip tensorflow-hub tensorflow-datasets sagemak

RUN echo "deb [arch=amd64] http://storage.googleapis.com/tensorflow-serving-apt stable
RUN curl https://storage.googleapis.com/tensorflow-serving-apt/tensorflow-serving.relea
RUN apt-get update && apt-get install tensorflow-model-server

ENV PATH="/opt/ml/code:${PATH}"

# /opt/ml and all subdirectories are utilized by SageMaker, we use the /code subdirecto
COPY /code /opt/ml/code
WORKDIR /opt/ml/code

RUN chmod 755 serve
```

Dockerfile.dockerfile hosted with ❤ by **GitHub** **view raw**

The Docker command below is used to build the custom Docker image used for both training and hosting with SageMaker for this project.

```
docker build ./container/ -t sagemaker-custom-tensorflow-container-gpu:1.0
```

After the Docker image is built successfully, use the Docker commands below to verify the new image is listed as expected.

```
docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
sagemaker-custom-tensorflow-container-gpu	1.0	05ccba2c7dbb	2 hours ago	3.62GB
tensorflow/tensorflow	2.3.0-gpu	20fd065e3887	5 weeks ago	3.15GB

SageMaker Training In Local Mode

The **SageMaker Python SDK** supports **local mode**, which allows users to create estimators, train models and deploy them to their local environments. This is very useful and cost-effective for anyone who wants to prototype, build, develop and test his or her machine learning projects in a Jupyter Notebook with the SageMaker Python SDK on the local instance before running in the cloud.

The Amazon SageMaker local mode supports **local CPU instance (single and multiple-instance)** and **local GPU instance (single instance)**. It also allows users to switch seamlessly between local and cloud instances (i.e. Amazon EC2 instance) by changing the **instance_type** argument for the SageMaker Estimator object (Note: This argument is previously known as **train_instance_type** in SageMaker Python SDK 1.x). Everything else works the same.

In this scenario, the local GPU instance is used by default if available, else fall back to local CPU instance. Note that the **output_path** is set to the local current directory (**file://.**) which will output the trained model artifacts to the local current directory instead of uploading onto Amazon S3. The **image_uri** is set to the local custom Docker image which is built locally so that SageMaker will not fetch from the pre-built Docker images based on framework and version. You can refer to the latest SageMaker TensorFlow Estimator and SageMaker Estimator Base API documentations for the full details.

In addition, **hyperparameters** can be passed to the training script by setting the **hyperparameters** of the SageMaker Estimator object. The hyperparameters that can be set depending on the hyperparameters used in the training script. In this case, they are *'epochs'*, *'batch_size'* and *'learning_rate'*.

```
algo-1-nukxt_1  | 2020-09-05 04:43:55.611981: I tensorflow_serving/model_servers/server.cc:387] Exporting HTTP/REST API at:localhost:8501
...
!algo-1-nukxt_1  | 172.18.0.1 - - [05/Sep/2020:04:43:58 +0000] "GET /ping HTTP/1.1" 200 2 "-" "-"
```

```
CONTAINER ID        IMAGE                                          COMMAND             CREATED             STATUS                      P
ORTS                NAMES
e8ed5762b897        sagemaker-custom-tensorflow-container-gpu:1.0  "serve"             5 seconds ago       Up 4 seconds
0.0.0.0:8080->8080/tcp   tmpjkjv5oc7_algo-1-nukxt_1
41a631f7fa19        sagemaker-custom-tensorflow-container-gpu:1.0  "train"             6 minutes ago       Exited (0) 12 seconds ago
tmpbb00tk19_algo-1-xxmq4_1
```

```
instance_type = 'local_gpu' # For Local GPU training. For Local CPU Training, type = 'l
```

```
    instance_type = 'local'

print(f'Instance type = {instance_type}')

role = 'SageMakerRole' # Import get_execution_role from sagemaker and use get_execution

hyperparams = {'epochs': 5}

tf_local_estimator = TensorFlow(entry_point='train.py', role=role,
                                instance_count=1, instance_type='local_gpu', output_pat
                                image_uri='sagemaker-custom-tensorflow-container-gpu:1.
                                hyperparameters=hyperparams)
tf_local_estimator.fit()
```

sagemaker_local_train.py hosted with ❤ by **GitHub** **view raw**

```
algo-1-xxmq4_1  | 2020-09-05 04:43:46,555 sagemaker-training-toolkit INFO     Reporting training SUCCESS
tmpbb00tk19_algo-1-xxmq4_1 exited with code 0
Aborting on container exit...
===== Job Complete =====
```

SageMaker training job completed in custom TensorFlow container

SageMaker Local Endpoint Deployment and Model Serving

After the SageMaker training job is completed, the Docker container that run that job will be exited. When the training is completed successfully, the trained model can be deployed to a **local SageMaker endpoint** by calling the **deploy** method of the SageMaker Estimator object and setting the **instance_type** to local instance type (i.e. **local_gpu** or **local**).

A new Docker container will be started to run the custom inference code (i.e the **serve** program), which runs the TensorFlow Serving ModelServer to serve the model for real-time inference. The ModelServer will serve in RESTful APIs mode and expect both the

request and response data in JSON format. When the local SageMaker endpoint is deployed

```
def preprocess_input(image_path):
    if (os.path.exists(image_path)):
        originalImage = Image.open(image_path)
        image = originalImage.resize((299, 299))
        image = np.asarray(image) / 255.
        image = tf.expand_dims(image,0)
        input_data = {'instances': np.asarray(image).astype(float)}
        return input_data
    else:
        print(f'{image_path} does not exist!\n')
        return None

def display(image, predicted_label, confidence_score, actual_label):
    fig, ax = plt.subplots(figsize=(8, 6))
    fig.suptitle(f'Predicted: {predicted_label}     Score: {confidence_score}     Actua
                 fontsize='xx-large', fontweight='extra bold')
    ax.imshow(image, aspect='auto')
    ax.axis('off')
    plt.show()

def predict_flower_type(image_path, actual_label):
    input_data = preprocess_input(image_path)

    if (input_data):
        result = tf_local_predictor.predict(input_data)
        CLASSES = info.features['label'].names
        predicted_class_idx = np.argmax(result['predictions'][0], axis=-1)
        predicted_class_label = CLASSES[predicted_class_idx]
        predicted_score = round(result['predictions'][0][predicted_class_idx], 4)
        original_image = Image.open(image_path)
        display(original_image, predicted_class_label, predicted_score, actual_label)
    else:
        print(f'Unable to predict {image_path}!\n')
        return None
```

predict.py hosted with ❤ by **GitHub** **view raw**

```
algo-1-nukxt_1  | 172.18.0.1 - - [05/Sep/2020:04:44:00 +0000] "POST /invocations HTTP/1.1" 200 1621 "-" "-"
```

Predicted: lotus Score: 0.9867 Actual: lotus

```
algo-1-nukxt_1  | 172.18.0.1 - - [05/Sep/2020:04:44:00 +0000] "POST /invocations HTTP/1.1" 200 1652 "-" "-"
```

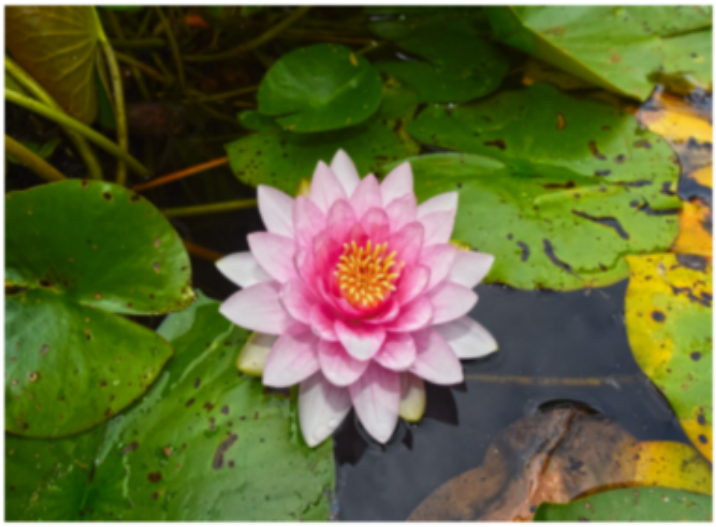
Predicted: hibiscus Score: 0.9566 Actual: hibiscus

```
algo-1-nukxt_1  | 172.18.0.1 - - [05/Sep/2020:04:44:05 +0000] "POST /invocations HTTP/1.1" 200 1634 "-" "-"
```

Predicted: sunflower Score: 0.888 Actual: sunflower

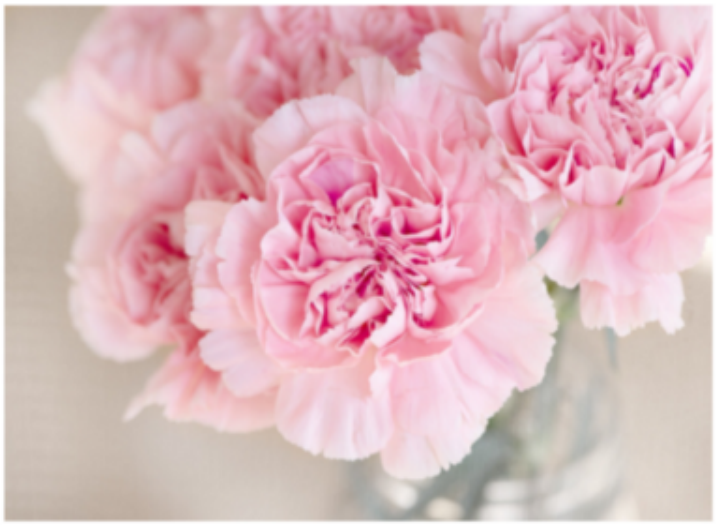
```
algo-1-nukxt_1  | 172.18.0.1 - - [05/Sep/2020:04:44:03 +0000] "POST /invocations HTTP/1.1" 200 1639 "-" "-"
```

Predicted: water lily Score: 0.9993 Actual: water lily



```
algo-1-nukxt_1  | 172.18.0.1 - - [05/Sep/2020:04:44:06 +0000] "POST /invocations HTTP/1.1" 200 1630 "-" "-"
```

Predicted: carnation Score: 0.752 Actual: carnation



```
algo-1-nukxt_1  | 172.18.0.1 - - [05/Sep/2020:04:44:07 +0000] "POST /invocations HTTP/1.1" 200 1645 "-" "-"
```

Predicted: english marigold Score: 0.9981 Actual: english marigold

```
algo-1-nukxt_1  | 172.18.0.1 - - [05/Sep/2020:04:44:08 +0000] "POST /invocations HTTP/1.1" 200 1642 "-" "-"
```

Predicted: moon orchid Score: 0.9868 Actual: moon orchid

Wrap-up

The final flower classification model is evaluated against a set of real-world flower images of different types from external sources to test how well it generalizes against unseen data. As a result, the model is able to classify all the unseen flower images correctly. The model size is approximately **80 MB**, which could be considered as reasonably compact and efficient for edge deployment in production. In summary, the model seemed to be able to perform well on a given small set of unseen data and reasonably compact for production edge or web deployment.

Proposed Enhancements

Due to time and resources constraints, the solution here may not be providing the best practices or optimal designs and implementations. Here are some of the ideas which

```
CONTAINER ID        IMAGE                                                  COMMAND             CREATED             STATUS
PORTS               NAMES
e8ed5762b897        sagemaker-custom-tensorflow-container-gpu:1.0          "serve"             28 seconds ago      Exited (137) Less than a seco
nd ago                      tmpjkjv5oc7_algo-1-nukxt_1
41a631f7fa19        sagemaker-custom-tensorflow-container-gpu:1.0          "train"             6 minutes ago       Exited (0) 34 seconds ago
tmpbb00tk19_algo-1-xxmq4_1
```

SageMaker Training and Serving Containers both in Exited Status

- Apply **Data Augmentation**, i.e. random (but realistic) transformations such as rotation, flip, crop, brightness and contrast etc. on the training dataset to increase its size and diversity

```
docker rm $(docker ps -a -q)
docker container ls -a
```

image preprocessing layers and **image Data Augmentation preprocessing**

layers which can be combined and exported as part of a Keras SavedModel. As a result, th
model can accept raw images as input.

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
```

- Convert the TensorFlow model (SavedModel format) to a **TensorFlow Lite** model (.tflite)
edge deployment and optimization on mobile and IoT devices.

mize

out

nsorF

- Use Intel Distribution of OpenVINO toolkit for the model's optimization and high performance inference on Intel hardware such as CPU, iGPU, VPU or FPGA.
- Optimize the Docker image size.

Amazon Sagemaker Tensorflow2 Tensorflow Serving Transfer Learning Docker Container

- Add unit test for the TensorFlow training script.
- Add unit test for the Dockerfile.

After the machine learning workflow has been tested working as expected in the local environment, the next step is to fully migrate this workflow to AWS Cloud with **Amazon Sa... Notebook Instance**. In the next guide, I will demonstrate how to adapt this Jupy... on ... book Instance as well as how to push the custom Docker image to the **Amazon Elastic Container Registry (ECR)** so that the whole workflow is fully hosted and managed in AWS.

Clean-up

It is always a best practice to clean up obsolete resources or sessions at the end to reclaim compute, memory and storage resources as well as to save cost if clean up on cloud or distribu environment. For this scenario, the local SageMaker inference endpoint as well as SageMaker containers are deleted as shown below.

```
tf_local_predictor.delete_endpoint()
```

```
Gracefully stopping... (press Ctrl+C again to force)
```

Delete the SageMaker inference endpoint and stop the TensorFlow Serving Container

```
docker container ls -a
```