Building your own algorithm container

With Amazon SageMaker, you can package your own algorithms that can than be trained and deployed in the SageMaker environment. This notebook will guide you through an example that shows you how to build a Docker container for SageMaker and use it for training and inference.

By packaging an algorithm in a container, you can bring almost any code to the Amazon SageMaker environment, regardless of programming language, environment, framework, or dependencies.

or I'm impatient, just let me see the code!

When should I build my own algorithm container?

You may not need to create a container to bring your own code to Amazon SageMaker. When you are using a framework (such as Apache MXNet or TensorFlow) that has direct support in SageMaker, you can simply supply the Python code that implements your algorithm using the SDK entry points for that framework. This set of frameworks is continually expanding, so we recommend that you check the current list if your algorithm is written in a common machine learning environment.

Even if there is direct SDK support for your environment or framework, you may find it more effective to build your own container. If the code that implements your algorithm is quite complex on its own or you need special additions to the framework, building your own container may be the right choice.

If there isn't direct SDK support for your environment, don't worry. You'll see in this walk-through that building your own container is quite straightforward.

Permissions

Running this notebook requires the instance role to have `AmazonEC2ContainerRegistryFullAccess` permissions in addition to `SageMakerFullAccess` permissions. The role you were instructed to use to launch this notebook has already had the appropriate permissions added but this is important to keep in mind if you wish to re-use this notebook for later projects.

The example

Here, we'll show how to package a simple Python example which showcases the decision tree (http://scikit-learn.org/stable/modules/tree.html) algorithm from the widely used scikit-learn (http://scikit-learn.org/stable/) machine learning package. The example is purposefully fairly trivial since the point is to show the surrounding structure that you'll want to add to your own code so you can train and host it in Amazon SageMaker.

The ideas shown here will work in any language or environment. You'll need to choose the right tools for your environment to serve HTTP requests for inference, but good HTTP environments are available in every language these days.

In this example, we use a single image to support training and hosting. This is easy because it means that we only need to manage one image and we can set it up to do everything. Sometimes you'll want separate images for training and hosting because they have different requirements. Just separate the parts discussed below into separate Dockerfiles and build two images. Choosing whether to have a single image or two images is really a matter of which is more convenient for you to develop and manage.

If you're only using Amazon SageMaker for training or hosting, but not both, there is no need to build the unused functionality into your container.

The presentation

This presentation is divided into two parts: *building* the container and *using* the container.

Part 1: Packaging and Uploading your Algorithm for use with Amazon SageMaker

An overview of Docker

If you're familiar with Docker already, you can skip ahead to the next section.

For many data scientists, Docker containers are a new concept, but they are not difficult, as you'll see here.

Docker provides a simple way to package arbitrary code into an *image* that is totally self-contained. Once you have an image, you can use Docker to run a *container* based on that image. Running a container is just like running a program on the machine except that the container creates a fully self-contained environment for the program to run. Containers are isolated from each other and from the host environment, so the way you set up your program is the way it runs, no matter where you run it.

Docker is more powerful than environment managers like conda or virtualenv because (a) it is completely language independent and (b) it comprises your whole operating environment, including startup commands, environment variable, etc.

In some ways, a Docker container is like a virtual machine, but it is much lighter weight. For example, a program running in a container can start in less than a second and many containers can run on the same physical machine or virtual machine instance.

Docker uses a simple file called a `Dockerfile` to specify how the image is assembled. We'll see an example of that below. You can build your Docker images based on Docker images built by yourself or others, which can simplify things quite a bit.

Docker has become very popular in the programming and devops communities for its flexibility and well-defined specification of the code to be run. It is the underpinning of many services built in the past few years, such as Amazon ECS (https://aws.amazon.com/ecs/).

Amazon SageMaker uses Docker to allow users to train and deploy arbitrary algorithms.

In Amazon SageMaker, Docker containers are invoked in a certain way for training and a slightly different way for hosting. The following sections outline how to build containers for the SageMaker environment.

Some helpful links:

- Docker home page (http://www.docker.com)
- Getting started with Docker (https://docs.docker.com/get-started/)
- Dockerfile reference (https://docs.docker.com/engine/reference/builder/)
- `docker run` reference (https://docs.docker.com/engine/reference/run/)

How Amazon SageMaker runs your Docker container

Because you can run the same image in training or hosting, Amazon SageMaker runs your container with the argument `train` or `serve`. How your container processes this argument depends on the container:

- In the example here, we don't define an `ENTRYPOINT` in the Dockerfile so Docker will run the command `train` at training time and `serve` at serving time. In this example, we define these as executable Python scripts, but they could be any program that we want to start in that environment.
- If you specify a program as an `ENTRYPOINT` in the Dockerfile, that program will be run at startup and its first argument will be `train` or `serve`. The program can then look at that argument and decide what to do.
- If you are building separate containers for training and hosting (or building only for one or the other), you can define a program as an `ENTRYPOINT` in the Dockerfile and ignore (or verify) the first argument passed in.

Running your container during training

When Amazon SageMaker runs training, your `train` script is run just like a regular Python program. A number of files are laid out for your use, under the `/opt/ml` directory:

```
/opt/ml
├── input
│   ├── config
│   │   ├── hyperparameters.json
│   │   └── resourceConfig.json
│   └── data
│       └── <channel_name>
│           └── <input data>
├── model
│   └── <model files>
└── output
    └── failure
```

The input

- `/opt/ml/input/config` contains information to control how your program runs. `hyperparameters.json` is a JSON-formatted dictionary of hyperparameter names to values. These values will always be strings, so you may need to convert them. `resourceConfig.json` is a JSON-formatted file that describes the network layout used for distributed training. Since scikit-learn doesn't support distributed training, we'll ignore it here.
- `/opt/ml/input/data/<channel_name>/` (for File mode) contains the input data for that channel. The channels are created based on the call to CreateTrainingJob but it's generally important that channels match what the algorithm expects. The files for each channel will be copied from S3 to this directory, preserving the tree structure indicated by the S3 key structure.
- `/opt/ml/input/data/<channel_name>_<epoch_number>` (for Pipe mode) is the pipe for a given epoch. Epochs start at zero and go up by one each time you read them. There is no limit to the number of epochs that you can run, but you must close each pipe before reading the next epoch.

The output

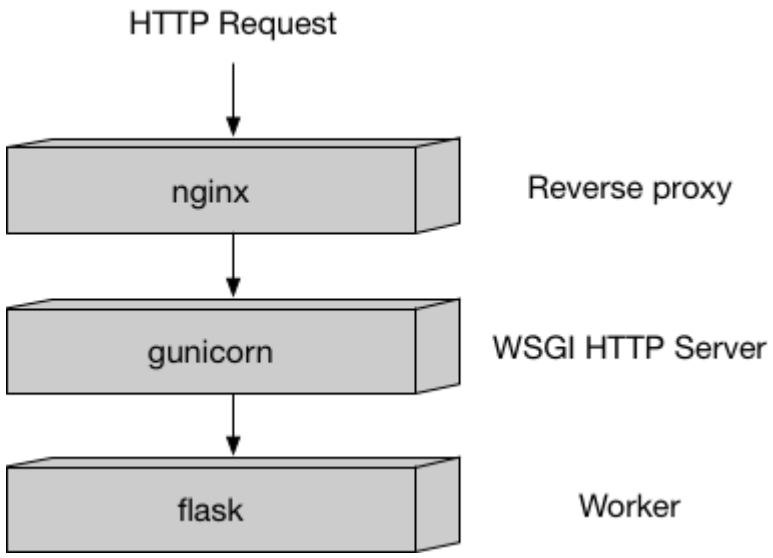
- `/opt/ml/model/` is the directory where you write the model that your algorithm generates. Your model can be in any format that you want. It can be a single file or a whole directory tree. SageMaker will

package any files in this directory into a compressed tar archive file. This file will be available at the S3 location returned in the `DescribeTrainingJob` result.

- `/opt/ml/output` is a directory where the algorithm can write a file `failure` that describes why the job failed. The contents of this file will be returned in the `FailureReason` field of the `DescribeTrainingJob` result. For jobs that succeed, there is no reason to write this file as it will be ignored.

Running your container during hosting

Hosting has a very different model than training because hosting is reponding to inference requests that come in via HTTP. In this example, we use our recommended Python serving stack to provide robust and scalable serving of inference requests:



This stack is implemented in the sample code here and you can mostly just leave it alone.

Amazon SageMaker uses two URLs in the container:

- `/ping` will receive `GET` requests from the infrastructure. Your program returns 200 if the container is up and accepting requests.
- `/invocations` is the endpoint that receives client inference `POST` requests. The format of the request and the response is up to the algorithm. If the client supplied `ContentType` and `Accept` headers, these will be passed in as well.

The container will have the model files in the same place they were written during training:

```
/opt/ml
└── model
    └── <model files>
```

The parts of the sample container

In the `container` directory are all the components you need to package the sample algorithm for Amazon SageMager:

```
.
├── Dockerfile
├── build_and_push.sh
└── decision_trees
    ├── nginx.conf
    ├── predictor.py
    ├── serve
    ├── train
    └── wsgi.py
```

Let's discuss each of these in turn:

- **`Dockerfile`** describes how to build your Docker container image. More details below.
- **`build_and_push.sh`** is a script that uses the Dockerfile to build your container images and then pushes it to ECR. We'll invoke the commands directly later in this notebook, but you can just copy and run the script for your own algorithms.
- **`decision_trees`** is the directory which contains the files that will be installed in the container.
- **`local_test`** is a directory that shows how to test your new container on any computer that can run Docker, including an Amazon SageMaker notebook instance. Using this method, you can quickly iterate using small datasets to eliminate any structural bugs before you use the container with Amazon SageMaker. We'll walk through local testing later in this notebook.

In this simple application, we only install five files in the container. You may only need that many or, if you have many supporting routines, you may wish to install more. These five show the standard structure of our Python containers, although you are free to choose a different toolset and therefore could have a different layout. If you're writing in a different programming language, you'll certainly have a different layout depending on the frameworks and tools you choose.

The files that we'll put in the container are:

- **`nginx.conf`** is the configuration file for the nginx front-end. Generally, you should be able to take this file as-is.
- **`predictor.py`** is the program that actually implements the Flask web server and the decision tree predictions for this app. You'll want to customize the actual prediction parts to your application. Since this algorithm is simple, we do all the processing here in this file, but you may choose to have separate files for implementing your custom logic.
- **`serve`** is the program started when the container is started for hosting. It simply launches the gunicorn server which runs multiple instances of the Flask app defined in `predictor.py`. You should be able to take this file as-is.
- **`train`** is the program that is invoked when the container is run for training. You will modify this program to implement your training algorithm.
- **`wsgi.py`** is a small wrapper used to invoke the Flask app. You should be able to take this file as-is.

In summary, the two files you will probably want to change for your application are `train` and `predictor.py`.

The Dockerfile

The Dockerfile describes the image that we want to build. You can think of it as describing the complete operating system installation of the system that you want to run. A Docker container running is quite a bit lighter than a full operating system, however, because it takes advantage of Linux on the host machine for the basic operations.

For the Python science stack, we will start from a standard Ubuntu installation and run the normal tools to install the things needed by scikit-learn. Finally, we add the code that implements our specific algorithm to the container and set up the right environment to run under.

Along the way, we clean up extra space. This makes the container smaller and faster to start.

Let's look at the Dockerfile for the example:

In [ ]:
```
!cat container/Dockerfile
```

Building and registering the container

The following shell code shows how to build the container image using `docker build` and push the container image to ECR using `docker push`. This code is also available as the shell script `container/build-and-push.sh`, which you can run as `build-and-push.sh decision_trees_sample` to build the image `decision_trees_sample`.

This code looks for an ECR repository in the account you're using and the current default region (if you're using a SageMaker notebook instance, this will be the region where the notebook instance was created). If the repository doesn't exist, the script will create it.

In [ ]:
```
%%sh

your_username=<YOUR_USERNAME_HERE>

# The name of our algorithm
algorithm_name="scikit-decision-tree-${your_username}"

cd container

chmod +x decision_trees/train
chmod +x decision_trees/serve

account=$(aws sts get-caller-identity --query Account --output text)

# Get the region defined in the current configuration (default to us-west-2 if
 none defined)
region=$(aws configure get region)
region=${region:-us-west-2}

fullname="${account}.dkr.ecr.${region}.amazonaws.com/${algorithm_name}:latest"

# If the repository doesn't exist in ECR, create it.

aws ecr describe-repositories --repository-names "${algorithm_name}" > /dev/nu
ll 2>&1

if [ $? -ne 0 ]
then
    aws ecr create-repository --repository-name "${algorithm_name}" > /dev/nul
l
fi

# Get the login command from ECR and execute it directly
$(aws ecr get-login --region ${region} --no-include-email)

# Build the docker image locally with the image name and then push it to ECR
# with the full name.

docker build  -t ${algorithm_name} .
docker tag ${algorithm_name} ${fullname}

docker push ${fullname}
```

Testing your algorithm on your local machine or on an Amazon SageMaker notebook instance

While you're first packaging an algorithm use with Amazon SageMaker, you probably want to test it yourself to make sure it's working right. In the directory `container/local_test`, there is a framework for doing this. It includes three shell scripts for running and using the container and a directory structure that mimics the one outlined above.

The scripts are:

- `train_local.sh`: Run this with the name of the image and it will run training on the local tree. You'll want to modify the directory `test_dir/input/data/...` to be set up with the correct channels and data for your algorithm. Also, you'll want to modify the file `input/config/hyperparameters.json` to have the hyperparameter settings that you want to test (as strings).
- `serve_local.sh`: Run this with the name of the image once you've trained the model and it should serve the model. It will run and wait for requests. Simply use the keyboard interrupt to stop it.
- `predict.sh`: Run this with the name of a payload file and (optionally) the HTTP content type you want. The content type will default to `text/csv`. For example, you can run `$ ./predict.sh payload.csv text/csv`.

The directories as shipped are set up to test the decision trees sample algorithm presented here.

Part 2: Training and Hosting your Algorithm in Amazon SageMaker

Once you have your container packaged, you can use it to train and serve models. Let's do that with the algorithm we made above.

Set up the environment

Here we specify a bucket to use and the role that will be used for working with SageMaker.

In [ ]:
```
# S3 prefix
your_username = "<YOUR_USERNAME_HERE>"
prefix = 'scikit-byo-iris-{}'.format(your_username)

# Define IAM role
import boto3
import re

import os
import numpy as np
import pandas as pd
from sagemaker import get_execution_role

role = get_execution_role()
```

Create the session

The session remembers our connection parameters to SageMaker. We'll use it to perform all of our SageMaker operations.

In [ ]:
```
import sagemaker as sage
from time import gmtime, strftime

sess = sage.Session()
```

Upload the data for training

When training large models with huge amounts of data, you'll typically use big data tools, like Amazon Athena, AWS Glue, or Amazon EMR, to create your data in S3. For the purposes of this example, we're using some the classic Iris dataset (https://en.wikipedia.org/wiki/Iris_flower_data_set), which we have included.

We can use use the tools provided by the SageMaker Python SDK to upload the data to a default bucket.

In [ ]:
```
WORK_DIRECTORY = 'data'

data_location = sess.upload_data(WORK_DIRECTORY, key_prefix=prefix)
```

Create an estimator and fit the model

In order to use SageMaker to fit our algorithm, we'll create an `Estimator` that defines how to use the container to train. This includes the configuration we need to invoke SageMaker training:

- The **container name**. This is constructed as in the shell commands above.
- The **role**. As defined above.
- The **instance count** which is the number of machines to use for training.
- The **instance type** which is the type of machine to use for training.
- The **output path** determines where the model artifact will be written.
- The **session** is the SageMaker session object that we defined above.

Then we use fit() on the estimator to train against the data that we uploaded above.

In [ ]:
```
algorithm_name = "scikit-decision-tree-{}".format(your_username)
account = sess.boto_session.client('sts').get_caller_identity()['Account']
region = sess.boto_session.region_name
image = '{}.dkr.ecr.{}.amazonaws.com/{}:latest'.format(account, region, algori
thm_name)

tree = sage.estimator.Estimator(image,
                        role, 1, 'ml.c4.2xlarge',
                        output_path="s3://{}/output".format(sess.default_bucket
()),
                        sagemaker_session=sess)

tree.fit(data_location)
```

Deploy the model

Deploying the model to SageMaker hosting just requires a `deploy` call on the fitted model. This call takes an instance count, instance type, and optionally serializer and deserializer functions. These are used when the resulting predictor is created on the endpoint.

In [ ]:
```
from sagemaker.predictor import csv_serializer
predictor = tree.deploy(1, 'ml.m4.xlarge', serializer=csv_serializer)
```

Choose some data and use it for a prediction

In order to do some predictions, we'll extract some of the data we used for training and do predictions against it. This is, of course, bad statistical practice, but a good way to see how the mechanism works.

In [ ]:
```
shape=pd.read_csv("data/iris.csv", header=None)

import itertools

a = [50*i for i in range(3)]
b = [40+i for i in range(10)]
indices = [i+j for i,j in itertools.product(a,b)]

test_data=shape.iloc[indices[:-1]]
test_X=test_data.iloc[:,1:]
test_y=test_data.iloc[:,0]
```

Prediction is as easy as calling predict with the predictor we got back from deploy and the data we want to do predictions with. The serializers take care of doing the data conversions for us.

In [ ]:
```
print(predictor.predict(test_X.values).decode('utf-8'))
```

Optional cleanup

When you're done with the endpoint, you'll want to clean it up

In [ ]:
```
sess.delete_endpoint(predictor.endpoint)
```