TensorFlow script mode training and serving

Script mode is a training script format for TensorFlow that lets you execute any TensorFlow training script in SageMaker with minimal modification. The SageMaker Python SDK handles transferring your script to a SageMaker training instance. On the training instance, SageMaker's native TensorFlow support sets up training-related environment variables and executes your training script. In this tutorial, we use the SageMaker Python SDK to launch a training job and deploy the trained model.

Script mode supports training with a Python script, a Python module, or a shell script. In this example, we use a Python script to train a classification model on the MNIST dataset. In this example, we will show how easily you can train a SageMaker using TensorFlow 1.x and TensorFlow 2.0 scripts with SageMaker Python SDK. In addition, this notebook demonstrates how to perform real time inference with the SageMaker TensorFlow Serving container. The TensorFlow Serving container is the default inference method for script mode. For full documentation on the TensorFlow Serving container, please visit here.

Set up the environment

Let's start by setting up the environment:

[ ]:
```
import os
import sagemaker
from sagemaker import get_execution_role

sagemaker_session = sagemaker.Session()

role = get_execution_role()
region = sagemaker_session.boto_session.region_name
```

Training Data

The MNIST dataset has been loaded to the public S3 buckets `sagemaker-sample-data-<REGION>` under the prefix `tensorflow/mnist`. There are four `.npy` file under this prefix: * `train_data.npy` * `eval_data.npy` * `train_labels.npy` * `eval_labels.npy`

[ ]:
```
training_data_uri = 's3://sagemaker-sample-data-{}/tensorflow/mnist'.format(region)
```

Construct a script for distributed training

This tutorial's training script was adapted from TensorFlow's official CNN MNIST example. We have modified it to handle the `model_dir` parameter passed in by SageMaker. This is an S3 path which can be used for data sharing during distributed training and checkpointing and/or model persistence. We have also added an argument-parsing function to handle processing training-related variables.

At the end of the training job we have added a step to export the trained model to the path stored in the environment variable `SM_MODEL_DIR`, which always points to `/opt/ml/model`. This is critical because SageMaker uploads all the model artifacts in this folder to S3 at end of training.

Here is the entire script:

[ ]:
```
!pygmentize 'mnist.py'

# TensorFlow 2.1 script
!pygmentize 'mnist-2.py'
```

Create a training job using the `TensorFlow` estimator

The `sagemaker.tensorflow.TensorFlow` estimator handles locating the script mode container, uploading your script to a S3 location and creating a SageMaker training job. Let's call out a couple important parameters here:

- `py_version` is set to `'py3'` to indicate that we are using script mode since legacy mode supports only Python 2. Though Python 2 will be deprecated soon, you can use script mode with Python 2 by setting `py_version` to `'py2'` and `script_mode` to `True`.
- `distributions` is used to configure the distributed training setup. It's required only if you are doing distributed training either across a cluster of instances or across multiple GPUs. Here we are using parameter servers as the distributed training schema. SageMaker training jobs run on homogeneous clusters. To make parameter server more performant in the SageMaker setup, we run a parameter server on every instance in the cluster, so there is no need to specify the number of parameter servers to launch. Script mode also supports distributed training with Horovod. You can find the full documentation on how to configure `distributions` here.

[ ]:
```
from sagemaker.tensorflow import TensorFlow


mnist_estimator = TensorFlow(entry_point='mnist.py',
                             role=role,
                             instance_count=2,
                             instance_type='ml.p3.2xlarge',
                             framework_version='1.15.2',
                             py_version='py3',
                             distribution={'parameter_server': {'enabled': True}})
```

You can also initiate an estimator to train with TensorFlow 2.1 script. The only things that you will need to change are the script name and `framewotk_version`

[ ]:
```
mnist_estimator2 = TensorFlow(entry_point='mnist-2.py',
                                role=role,
                                instance_count=2,
                                instance_type='ml.p3.2xlarge',
                                framework_version='2.1.0',
                                py_version='py3',
                                distribution={'parameter_server': {'enabled': True}})
```

Calling `fit`

To start a training job, we call `estimator.fit(training_data_uri)`.

An S3 location is used here as the input. `fit` creates a default channel named `'training'`, which points to this S3 location. In the training script we can then access the training data from the location stored in `SM_CHANNEL_TRAINING`. `fit` accepts a couple other types of input as well. See the API doc here for details.

When training starts, the TensorFlow container executes mnist.py, passing `hyperparameters` and `model_dir` from the estimator as script arguments. Because we didn't define either in this example, no hyperparameters are passed, and `model_dir` defaults to `s3://<DEFAULT_BUCKET>/<TRAINING_JOB_NAME>`, so the script execution is as follows:

```
python mnist.py --model_dir s3://<DEFAULT_BUCKET>/<TRAINING_JOB_NAME>
```

When training is complete, the training job will upload the saved model for TensorFlow serving.

[ ]:
```
mnist_estimator.fit(training_data_uri)
```

Calling fit to train a model with TensorFlow 2.1 script.

[ ]:
```
mnist_estimator2.fit(training_data_uri)
```

Deploy the trained model to an endpoint

The `deploy()` method creates a SageMaker model, which is then deployed to an endpoint to serve prediction requests in real time. We will use the TensorFlow Serving container for the endpoint, because we trained with script mode. This serving container runs an implementation of a web server that is compatible with SageMaker hosting protocol. The `Using your own inference code <>`__ document explains how SageMaker runs inference containers.

```
[ ]: predictor = mnist_estimator.deploy(initial_instance_count=1,
     instance_type='ml.p2.xlarge')
```

Deployed the trained TensorFlow 2.1 model to an endpoint.

```
[ ]: predictor2 = mnist_estimator2.deploy(initial_instance_count=1,
     instance_type='ml.p2.xlarge')
```

Invoke the endpoint

Let's download the training data and use that as input for inference.

```
[ ]: import numpy as np

     !aws --region {region} s3 cp s3://sagemaker-sample-data-
     {region}/tensorflow/mnist/train_data.npy train_data.npy
     !aws --region {region} s3 cp s3://sagemaker-sample-data-
     {region}/tensorflow/mnist/train_labels.npy train_labels.npy

     train_data = np.load('train_data.npy')
     train_labels = np.load('train_labels.npy')
```

The formats of the input and the output data correspond directly to the request and response formats of the `Predict` method in the TensorFlow Serving REST API. SageMaker's TensforFlow Serving endpoints can also accept additional input formats that are not part of the TensorFlow REST API, including the simplified JSON format, line-delimited JSON objects ("jsons" or "jsonlines"), and CSV data.

In this example we are using a `numpy` array as input, which will be serialized into the simplified JSON format. In addtion, TensorFlow serving can also process multiple items at once as you can see in the following code. You can find the complete documentation on how to make predictions against a TensorFlow serving SageMaker endpoint here.

```
[ ]: predictions = predictor.predict(train_data[:50])
     for i in range(0, 50):
         prediction = predictions['predictions'][i]['classes']
         label = train_labels[i]
         print('prediction is {}, label is {}, matched: {}'.format(prediction, label,
     prediction == label))
```

Examine the prediction result from the TensorFlow 2.1 model.

```
[ ]: predictions2 = predictor2.predict(train_data[:50])
     for i in range(0, 50):
         prediction = np.argmax(predictions2['predictions'][i])
         label = train_labels[i]
         print('prediction is {}, label is {}, matched: {}'.format(prediction, label,
     prediction == label))
```

Delete the endpoint

Let’s delete the endpoint we just created to prevent incurring any extra costs.

```
[ ]: predictor.delete_endpoint()
```

Delete the TensorFlow 2.1 endpoint as well.

```
[ ]: predictor2.delete_endpoint()
```

```
[ ]:
```