Using PyTorch with the SageMaker Python SDK

With PyTorch Estimators and Models, you can train and host PyTorch models on Amazon SageMaker.

For information about supported versions of PyTorch, see the AWS documentation.

We recommend that you use the latest supported version because that's where we focus our development efforts.

You can visit the PyTorch repository at https://github.com/pytorch/pytorch.

Contents

Train a Model with PyTorch

To train a PyTorch model by using the SageMaker Python SDK:

1. Prepare a training script
2. Create a `sagemaker.pytorch.PyTorch` Estimator
3. Call the estimator's `fit` method

Prepare a PyTorch Training Script

Your PyTorch training script must be a Python 2.7 or 3.5 compatible source file.

Prepare your script in a separate source file than the notebook, terminal session, or source file you're using to submit the script to SageMaker via a `PyTorch` Estimator. This will be discussed in further detail below.

The training script is very similar to a training script you might run outside of SageMaker, but you can access useful properties about the training environment through various environment variables. For example:

- `SM_NUM_GPUS`: An integer representing the number of GPUs available to the host.
- `SM_MODEL_DIR`: A string representing the path to the directory to write model artifacts to. These artifacts are uploaded to S3 for model hosting.
- `SM_OUTPUT_DATA_DIR`: A string representing the filesystem path to write output artifacts to. Output artifacts may include checkpoints, graphs, and other files to save, not including model artifacts. These artifacts are compressed and uploaded to S3 to the same S3 prefix as the model artifacts.
- `SM_CHANNEL_XXXX`: A string that represents the path to the directory that contains the input data for the specified channel. For example, if you specify two input channels in the PyTorch estimator's `fit` call, named 'train' and 'test', the environment variables `SM_CHANNEL_TRAIN` and `SM_CHANNEL_TEST` are set.

A typical training script loads data from the input channels, configures training with hyperparameters, trains a model, and saves a model to `model_dir` so that it can be hosted later. Hyperparameters are passed to your script as arguments and can be retrieved with an argparse.ArgumentParser instance. For example, a training script might start with the following:

```
import argparse
import os

if __name__ =='__main__':

    parser = argparse.ArgumentParser()

    # hyperparameters sent by the client are passed as command-line arguments to the
script.
    parser.add_argument('--epochs', type=int, default=50)
    parser.add_argument('--batch-size', type=int, default=64)
    parser.add_argument('--learning-rate', type=float, default=0.05)
    parser.add_argument('--use-cuda', type=bool, default=False)

    # Data, model, and output directories
    parser.add_argument('--output-data-dir', type=str,
default=os.environ['SM_OUTPUT_DATA_DIR'])
    parser.add_argument('--model-dir', type=str, default=os.environ['SM_MODEL_DIR'])
    parser.add_argument('--train', type=str, default=os.environ['SM_CHANNEL_TRAIN'])
    parser.add_argument('--test', type=str, default=os.environ['SM_CHANNEL_TEST'])

    args, _ = parser.parse_known_args()

    # ... load from args.train and args.test, train a model, write model to
args.model_dir.
```

Because the SageMaker imports your training script, you should put your training code in a main guard (`if __name__=='__main__':`) if you are using the same script to host your model, so that SageMaker does not inadvertently run your training code at the wrong point in execution.

Note that SageMaker doesn't support argparse actions. If you want to use, for example, boolean hyperparameters, you need to specify *type* as *bool* in your script and provide an explicit *True* or *False* value for this hyperparameter when instantiating PyTorch Estimator.

For more on training environment variables, see the SageMaker Training Toolkit.

Save the Model

In order to save your trained PyTorch model for deployment on SageMaker, your training script should save your model to a certain filesystem path called `model_dir`. This value is accessible through the environment variable `SM_MODEL_DIR`. The following code demonstrates how to save a trained PyTorch model named `model` as `model.pth` at the :

```
import argparse
import os
import torch

if __name__=='__main__':
    # default to the value in environment variable `SM_MODEL_DIR`. Using args makes the
script more portable.
    parser.add_argument('--model-dir', type=str, default=os.environ['SM_MODEL_DIR'])
    args, _ = parser.parse_known_args()

    # ... train `model`, then save it to `model_dir`
    with open(os.path.join(args.model_dir, 'model.pth'), 'wb') as f:
        torch.save(model.state_dict(), f)
```

After your training job is complete, SageMaker compresses and uploads the serialized model to S3, and your model data will be available in the S3 `output_path` you specified when you created the PyTorch Estimator.

If you are using Elastic Inference, you must convert your models to the TorchScript format and use `torch.jit.save` to save the model. For example:

```
import os
import torch

# ... train `model`, then save it to `model_dir`
model_dir = os.path.join(model_dir, "model.pt")
torch.jit.save(model, model_dir)
```

Using third-party libraries

When running your training script on SageMaker, it will have access to some pre-installed third-party libraries including `torch`, `torchvision`, and `numpy`. For more information on the runtime environment, including specific package versions, see SageMaker PyTorch Docker containers.

If there are other packages you want to use with your script, you can include a `requirements.txt` file in the same directory as your training script to install other dependencies at runtime. Both `requirements.txt` and your training script should be put in the same folder. You must specify this folder in `source_dir` argument when creating PyTorch estimator.

The function of installing packages using `requirements.txt` is supported for all PyTorch versions during training. When serving a PyTorch model, support for this function varies with PyTorch versions. For PyTorch 1.3.1 or newer, `requirements.txt` must be under folder `code`. The SageMaker PyTorch Estimator will automatically save `code` in `model.tar.gz` after training (assuming you set up your script and `requirements.txt` correctly as stipulated in the previous paragraph). In the case of bringing your own trained model for deployment, you must save `requirements.txt` under folder `code` in `model.tar.gz` yourself or specify it through `dependencies`. For PyTorch 1.2.0, `requirements.txt` is not supported for inference. For PyTorch 0.4.0 to 1.1.0, `requirements.txt` must be in `source_dir`.

A `requirements.txt` file is a text file that contains a list of items that are installed by using `pip install`. You can also specify the version of an item to install. For information about the format of a `requirements.txt` file, see Requirements Files in the pip documentation.

Create an Estimator

You run PyTorch training scripts on SageMaker by creating `PyTorch` Estimators. SageMaker training of your script is invoked when you call `fit` on a `PyTorch` Estimator. The following code sample shows how you train a custom PyTorch script "pytorch-train.py", passing in three hyperparameters ('epochs', 'batch-size', and 'learning-rate'), and using two input channel directories ('train' and 'test').

```
pytorch_estimator = PyTorch('pytorch-train.py',
                            instance_type='ml.p3.2xlarge',
                            instance_count=1,
                            framework_version='1.5.0',
                            py_version='py3',
                            hyperparameters = {'epochs': 20, 'batch-size': 64, 'learning-
rate': 0.1})
pytorch_estimator.fit({'train': 's3://my-data-bucket/path/to/my/training/data',
                       'test': 's3://my-data-bucket/path/to/my/test/data'})
```

Call the fit Method

You start your training script by calling `fit` on a `PyTorch` Estimator. `fit` takes both required and optional arguments.

fit Required Arguments

- `inputs`: This can take one of the following forms: A string S3 URI, for example `s3://my-bucket/my-training-data`. In this case, the S3 objects rooted at the `my-training-data` prefix will be available in the default `train` channel. A dict from string channel names to S3 URIs. In this case, the objects rooted at each S3 prefix will available as files in each channel directory.

For example:

/

```
{'train':'s3://my-bucket/my-training-data',
 'eval':'s3://my-bucket/my-evaluation-data'}
```

fit Optional Arguments

- `wait`: Defaults to True, whether to block and wait for the training script to complete before returning.
- `logs`: Defaults to True, whether to show logs produced by training job in the Python session. Only meaningful when wait is True.

Distributed PyTorch Training

You can run a multi-machine, distributed PyTorch training using the PyTorch Estimator. By default, PyTorch objects will submit single-machine training jobs to SageMaker. If you set `instance_count` to be greater than one, multi-machine training jobs will be launched when `fit` is called. When you run multi-machine training, SageMaker will import your training script and run it on each host in the cluster.

To initialize distributed training in your script you would call `dist.init_process_group` providing desired backend and rank and setting 'WORLD_SIZE' environment variable similar to how you would do it outside of SageMaker using environment variable initialization:

```
if args.distributed:
    # Initialize the distributed environment.
    world_size = len(args.hosts)
    os.environ['WORLD_SIZE'] = str(world_size)
    host_rank = args.hosts.index(args.current_host)
    dist.init_process_group(backend=args.backend, rank=host_rank)
```

SageMaker sets 'MASTER_ADDR' and 'MASTER_PORT' environment variables for you, but you can overwrite them.

Supported backends: - *gloo* and *tcp* for cpu instances - *gloo* and *nccl* for gpu instances

Deploy PyTorch Models

After a PyTorch Estimator has been fit, you can host the newly created model in SageMaker.

After calling `fit`, you can call `deploy` on a `PyTorch` Estimator to create a SageMaker Endpoint. The Endpoint runs a SageMaker-provided PyTorch model server and hosts the model produced by your training script, which was run when you called `fit`. This was the model you saved to `model_dir`.

`deploy` returns a `Predictor` object, which you can use to do inference on the Endpoint hosting your PyTorch model. Each `Predictor` provides a `predict` method which can do inference with numpy arrays or Python lists. Inference arrays or lists are serialized and sent to the PyTorch model server by an `InvokeEndpoint` SageMaker operation.

`predict` returns the result of inference against your model. By default, the inference result a NumPy array.

```
# Train my estimator
pytorch_estimator = PyTorch(entry_point='train_and_deploy.py',
                            instance_type='ml.p3.2xlarge',
                            instance_count=1,
                            framework_version='1.5.0',
                            py_version='py3')
pytorch_estimator.fit('s3://my_bucket/my_training_data/')

# Deploy my estimator to a SageMaker Endpoint and get a Predictor
predictor = pytorch_estimator.deploy(instance_type='ml.m4.xlarge',
                                     initial_instance_count=1)

# `data` is a NumPy array or a Python list.
# `response` is a NumPy array.
response = predictor.predict(data)
```

You use the SageMaker PyTorch model server to host your PyTorch model when you call `deploy` on an `PyTorch` Estimator. The model server runs inside a SageMaker Endpoint, which your call to `deploy` creates. You can access the name of the Endpoint by the `name` property on the returned `Predictor`.

Elastic Inference

PyTorch on Amazon SageMaker has support for Elastic Inference, which allows for inference acceleration to a hosted endpoint for a fraction of the cost of using a full GPU instance. In order to attach an Elastic Inference accelerator to your endpoint provide the accelerator type to `accelerator_type` to your `deploy` call.

```
predictor = pytorch_estimator.deploy(instance_type='ml.m4.xlarge',
                                     initial_instance_count=1,
                                     accelerator_type='ml.eia2.medium')
```

Model Directory Structure

In general, if you use the same version of PyTorch for both training and inference with the SageMaker Python SDK, the SDK should take care of ensuring that the contents of your `model.tar.gz` file are organized correctly.

For versions 1.2 and higher

For PyTorch versions 1.2 and higher, the contents of `model.tar.gz` should be organized as follows:

- Model files in the top-level directory
- Inference script (and any other source files) in a directory named `code/` (for more about the inference script, see The SageMaker PyTorch Model Server)
- Optional requirements file located at `code/requirements.txt` (for more about requirements files, see Using third-party libraries)

For example:

```
model.tar.gz/
|- model.pth
|- code/
  |- inference.py
  |- requirements.txt  # only for versions 1.3.1 and higher
```

In this example, `model.pth` is the model file saved from training, `inference.py` is the inference script, and `requirements.txt` is a requirements file.

The `PyTorch` and `PyTorchModel` classes repack `model.tar.gz` to include the inference script (and related files), as long as the `framework_version` is set to 1.2 or higher.

For versions 1.1 and lower

For PyTorch versions 1.1 and lower, `model.tar.gz` should contain only the model files, while your inference script and optional requirements file are packed in a separate tarball, named `sourcedir.tar.gz` by default.

For example:

```
model.tar.gz/
|- model.pth

sourcedir.tar.gz/
|- script.py
|- requirements.txt
```

In this example, `model.pth` is the model file saved from training, `script.py` is the inference script, and `requirements.txt` is a requirements file.

The SageMaker PyTorch Model Server

The PyTorch Endpoint you create with `deploy` runs a SageMaker PyTorch model server. The model server loads the model that was saved by your training script and performs inference on the model in response to SageMaker InvokeEndpoint API calls.

You can configure two components of the SageMaker PyTorch model server: Model loading and model serving. Model loading is the process of deserializing your saved model back into a PyTorch model. Serving is the process of translating InvokeEndpoint requests to inference calls on the loaded model.

You configure the PyTorch model server by defining functions in the Python source file you passed to the PyTorch constructor.

Load a Model

Before a model can be served, it must be loaded. The SageMaker PyTorch model server loads your model by invoking a `model_fn` function that you must provide in your script when you are not using Elastic Inference. The `model_fn` should have the following signature:

```python
def model_fn(model_dir)
```

SageMaker will inject the directory where your model files and sub-directories, saved by `save`, have been mounted. Your model function should return a model object that can be used for model serving.

The following code-snippet shows an example `model_fn` implementation. It loads the model parameters from a `model.pth` file in the SageMaker model directory `model_dir`.

```python
import torch
import os

def model_fn(model_dir):
    model = Your_Model()
    with open(os.path.join(model_dir, 'model.pth'), 'rb') as f:
        model.load_state_dict(torch.load(f))
    return model
```

However, if you are using PyTorch Elastic Inference, you do not have to provide a `model_fn` since the PyTorch serving container has a default one for you. But please note that if you are utilizing the default `model_fn`, please save your ScriptModule as `model.pt`. If you are implementing your own `model_fn`, please use TorchScript and `torch.jit.save` to save your ScriptModule, then load it in your `model_fn` with `torch.jit.load(..., map_location=torch.device('cpu'))`.

The client-side Elastic Inference framework is CPU-only, even though inference still happens in a CUDA context on the server. Thus, the default `model_fn` for Elastic Inference loads the model to CPU. Tracing models may lead to tensor creation on a specific device, which may cause device-related errors when loading a model onto a different device. Providing an explicit `map_location=torch.device('cpu')` argument forces all tensors to CPU.

For more information on the default inference handler functions, please refer to: SageMaker PyTorch Default Inference Handler.

Serve a PyTorch Model

After the SageMaker model server has loaded your model by calling `model_fn`, SageMaker will serve your model. Model serving is the process of responding to inference requests, received by SageMaker InvokeEndpoint API calls. The SageMaker PyTorch model server breaks request handling into three steps:

- input processing,
- prediction, and
- output processing.

In a similar way to model loading, you configure these steps by defining functions in your Python source file.

Each step involves invoking a python function, with information about the request and the return value from the previous function in the chain. Inside the SageMaker PyTorch model server, the process looks like:

```
# Deserialize the Invoke request body into an object we can perform prediction on
input_object = input_fn(request_body, request_content_type)

# Perform prediction on the deserialized object, with the loaded model
prediction = predict_fn(input_object, model)

# Serialize the prediction result into the desired response content type
output = output_fn(prediction, response_content_type)
```

The above code sample shows the three function definitions:

- `input_fn`: Takes request data and deserializes the data into an object for prediction.
- `predict_fn`: Takes the deserialized request object and performs inference against the loaded model.
- `output_fn`: Takes the result of prediction and serializes this according to the response content type.

The SageMaker PyTorch model server provides default implementations of these functions. You can provide your own implementations for these functions in your hosting script. If you omit any definition then the SageMaker PyTorch model server will use its default implementation for that function.

The `Predictor` used by PyTorch in the SageMaker Python SDK serializes NumPy arrays to the NPY format by default, with Content-Type `application/x-npy`. The SageMaker PyTorch model server can deserialize NPY-formatted data (along with JSON and CSV data).

If you rely solely on the SageMaker PyTorch model server defaults, you get the following functionality:

- Prediction on models that implement the `__call__` method
- Serialization and deserialization of torch.Tensor.

The default `input_fn` and `output_fn` are meant to make it easy to predict on torch.Tensors. If your model expects a torch.Tensor and returns a torch.Tensor, then these functions do not have to be overridden when sending NPY-formatted data.

In the following sections we describe the default implementations of input_fn, predict_fn, and output_fn. We describe the input arguments and expected return types of each, so you can define your own implementations.

Process Model Input

When an InvokeEndpoint operation is made against an Endpoint running a SageMaker PyTorch model server, the model server receives two pieces of information:

- The request Content-Type, for example “application/x-npy”
- The request data body, a byte array

The SageMaker PyTorch model server will invoke an `input_fn` function in your hosting script, passing in this information. If you define an `input_fn` function definition, it should return an object that can be passed to `predict_fn` and have the following signature:

```
def input_fn(request_body, request_content_type)
```

Where `request_body` is a byte buffer and `request_content_type` is a Python string

The SageMaker PyTorch model server provides a default implementation of `input_fn`. This function deserializes JSON, CSV, or NPY encoded data into a torch.Tensor.

Default NPY deserialization requires `request_body` to follow the NPY format. For PyTorch, the Python SDK defaults to sending prediction requests with this format.

Default JSON deserialization requires `request_body` contain a single json list. Sending multiple JSON objects within the same `request_body` is not supported. The list must have a dimensionality compatible with the model loaded in `model_fn`. The list’s shape must be identical to the model’s input shape, for all dimensions after the first (which first dimension is the batch size).

Default csv deserialization requires `request_body` contain one or more lines of CSV numerical data. The data is loaded into a two-dimensional array, where each line break defines the boundaries of the first dimension.

The example below shows a custom `input_fn` for preparing pickled torch.Tensor.

```
import numpy as np
import torch
from six import BytesIO

def input_fn(request_body, request_content_type):
    """An input_fn that loads a pickled tensor"""
    if request_content_type == 'application/python-pickle':
        return torch.load(BytesIO(request_body))
    else:
        # Handle other content-types here or raise an Exception
        # if the content type is not supported.
        pass
```

Get Predictions from a PyTorch Model

After the inference request has been deserialized by `input_fn`, the SageMaker PyTorch model server invokes `predict_fn` on the return value of `input_fn`.

As with `input_fn`, you can define your own `predict_fn` or use the SageMaker PyTorch model server default.

The `predict_fn` function has the following signature:

```
def predict_fn(input_object, model)
```

Where `input_object` is the object returned from `input_fn` and `model` is the model loaded by `model_fn`.

The default implementation of `predict_fn` invokes the loaded model's `__call__` function on `input_object`, and returns the resulting value. The return-type should be a torch.Tensor to be compatible with the default `output_fn`.

The example below shows an overridden `predict_fn`:

```
import torch
import numpy as np

def predict_fn(input_data, model):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model.to(device)
    model.eval()
    with torch.no_grad():
        return model(input_data.to(device))
```

/

If you implement your own prediction function, you should take care to ensure that:

- The first argument is expected to be the return value from input_fn. If you use the default input_fn, this will be a torch.Tensor.
- The second argument is the loaded model.
- The return value should be of the correct type to be passed as the first argument to `output_fn`. If you use the default `output_fn`, this should be a torch.Tensor.

The default Elastic Inference `predict_fn` is similar but runs the TorchScript model using `torch.jit.optimized_execution`. If you are implementing your own `predict_fn`, please also use the `torch.jit.optimized_execution` block, for example:

```
import torch
import numpy as np

def predict_fn(input_data, model):
    device = torch.device("cpu")
    model = model.to(device)
    input_data = data.to(device)
    model.eval()
    with torch.jit.optimized_execution(True, {"target_device": "eia:0"}):
        output = model(input_data)
```

Process Model Output

After invoking `predict_fn`, the model server invokes `output_fn`, passing in the return value from `predict_fn` and the content type for the response, as specified by the InvokeEndpoint request.

The `output_fn` has the following signature:

```
def output_fn(prediction, content_type)
```

Where `prediction` is the result of invoking `predict_fn` and the content type for the response, as specified by the InvokeEndpoint request. The function should return a byte array of data serialized to content_type.

The default implementation expects `prediction` to be a torch.Tensor and can serialize the result to JSON, CSV, or NPY. It accepts response content types of “application/json”, “text/csv”, and “application/x-npy”.

Bring your own model

You can deploy a PyTorch model that you trained outside of SageMaker by using the `PyTorchModel` class. Typically, you save a PyTorch model as a file with extension `.pt` or `.pth`. To do this, you need to:

- Write an inference script.
- Create the directory structure for your model files.
- Create the `PyTorchModel` object.

Write an inference script

You must create an inference script that implements (at least) the `model_fn` function that calls the loaded model to get a prediction.

Note: If you use elastic inference with PyTorch, you can use the default `model_fn` implementation provided in the serving container.

Optionally, you can also implement `input_fn` and `output_fn` to process input and output, and `predict_fn` to customize how the model server gets predictions from the loaded model. For information about how to write an inference script, see Serve a PyTorch Model. Save the inference script in the same folder where you saved your PyTorch model. Pass the filename of the inference script as the `entry_point` parameter when you create the `PyTorchModel` object.

Create the directory structure for your model files

You have to create a directory structure and place your model files in the correct location. The `PyTorchModel` constructor packs the files into a `tar.gz` file and uploads it to S3.

The directory structure where you saved your PyTorch model should look something like the following:

Note: This directory struture is for PyTorch versions 1.2 and higher. For the directory structure for versions 1.1 and lower, see For versions 1.1 and lower.

```
|   my_model
|           |--model.pth
|
|           code
|               |--inference.py
|               |--requirements.txt
```

Where `requirments.txt` is an optional file that specifies dependencies on third-party libraries.

Create a `PyTorchModel` object

Now call the **`sagemaker.pytorch.model.PyTorchModel`** constructor to create a model object, and then call its `deploy()` method to deploy your model for inference.

```
from sagemaker import get_execution_role
role = get_execution_role()

pytorch_model = PyTorchModel(model_data='s3://my-bucket/my-path/model.tar.gz', role=role,
                             entry_point='inference.py')

predictor = pytorch_model.deploy(instance_type='ml.c4.xlarge', initial_instance_count=1)
```

Now you can call the `predict()` method to get predictions from your deployed model.

Attach an estimator to an existing training job

You can attach a PyTorch Estimator to an existing training job using the `attach` method.

```
my_training_job_name = 'MyAwesomePyTorchTrainingJob'
pytorch_estimator = PyTorch.attach(my_training_job_name)
```

After attaching, if the training job has finished with job status “Completed”, it can be `deploy`ed to create a SageMaker Endpoint and return a `Predictor`. If the training job is in progress, attach will block and display log messages from the training job, until the training job completes.

The `attach` method accepts the following arguments:

- `training_job_name:` The name of the training job to attach to.
- `sagemaker_session:` The Session used to interact with SageMaker

PyTorch Training Examples

Amazon provides several example Jupyter notebooks that demonstrate end-to-end training on Amazon SageMaker using PyTorch. Please refer to:

https://github.com/awslabs/amazon-sagemaker-examples/tree/master/sagemaker-python-sdk

These are also available in SageMaker Notebook Instance hosted Jupyter notebooks under the sample notebooks folder.

SageMaker PyTorch Classes

For information about the different PyTorch-related classes in the SageMaker Python SDK, see https://sagemaker.readthedocs.io/en/stable/frameworks/pytorch/sagemaker.pytorch.html.

SageMaker PyTorch Docker Containers

For information about the SageMaker PyTorch containers, see:

- SageMaker PyTorch training toolkit
- SageMaker PyTorch serving toolkit
- Deep Learning Container (DLC) Dockerfiles for PyTorch
- Deep Learning Container (DLC) Images and release notes