TENSORRT SAMPLES

SWE-SWDOCTRT-001-SAMG_vTensorRT 5.1.2 RC | March 2019

Support Guide

TABLE OF CONTENTS

Chapter 1. SAMPLES

The following samples show how to use TensorRT in numerous use cases while highlighting different capabilities of the interface.

New Sample Name	Old Sample Name	Description
"Hello World" For TensorRT	sampleMNIST	Performs the basic setup and initialization of TensorRT using the Caffe parser.
Building A Simple OCR Network	sampleMNISTAPI	Uses the TensorRT API to build an MNIST (handwritten digit recognition) layer by layer, sets up weights and inputs/outputs and then performs inference.
Import The TensorFlow Model And Run Inference	sampleUffMNIST	Imports a TensorFlow model trained on the MNIST dataset.
"Hello World" For TensorRT From ONNX	sampleOnnxMNIST	Converts a model trained on the MNIST dataset in ONNX format to a TensorRT network.
Applying FP16 To GoogleNet And Profiling The App	sampleGoogleNet	Shows how to import a model trained with Caffe into TensorRT using GoogleNet as an example.
Building An RNN Network Layer By Layer	sampleCharRNN	Uses the TensorRT API to build an RNN network layer by layer, sets up weights and inputs/outputs and then performs inference.
Performing Inference In INT8 Precision	sampleINT8	Performs INT8 calibration and inference. Calibrates a network for execution in INT8.
Performing Inference In INT8 Using Custom Calibration	sampleINT8API	Sets per tensor dynamic range and computation precision of a layer.
Adding A Custom Layer To Your Network In TensorRT	samplePlugin	Defines a custom layer that supports multiple data formats that can be serialized and deserialized. Enables a custom layer in NvCaffeParser.

New Sample Name	Old Sample Name	Description
Object Detection With FasterRCNN	sampleFasterRCNN	Uses TensorRT plugins, performs inference, and implements a fused custom layer for end-to-end inferencing of a Faster R-CNN model.
Object Detection With A TensorFlow SSD Network	sampleUffSSD	Preprocess the TensorFlow SSD network, performs inference on the SSD network in TensorRT, and uses TensorRT plugins to speed up inference.
Movie Recommendation Using Neural Collaborative Filter (NCF)	sampleMovieLens	An end-to-end sample that imports a trained TensorFlow model and predicts the highest rated movie for each user.
Movie Recommendation Using MPS (Multi-Process Service)	sampleMovieLensMPS	An end-to-end sample that imports a trained TensorFlow model and predicts the highest rated movie for each user using MPS (Multi-Process Service).
Object Detection With SSD	sampleSSD	Preprocess the input to the SSD network, performs inference on the SSD network in TensorRT, uses TensorRT plugins to speed up inference, and performs INT8 calibration on an SSD network.
“Hello World” For Multi-Layer Perceptron (MLP)	sampleMLP	Shows how to create a network that triggers the multi-layer perceptron (MLP) optimizer.
Introduction To Importing Caffe, TensorFlow And ONNX Models Into TensorRT Using Python	introductory_parser_samples	Uses TensorRT and its included suite of parsers (the UFF, Caffe and ONNX parsers), to perform inference with ResNet-50 models trained with various different frameworks.
“Hello World” For TensorRT Using TensorFlow And Python	end_to_end_tensorflow_mnist	An end-to-end sample that trains a model in TensorFlow and Keras, freezes the model and writes it to a protobuf file, converts it to UFF, and finally runs inference using TensorRT.
“Hello World” For TensorRT Using PyTorch And Python	network_api_pytorch_mnist	An end-to-end sample that trains a model in PyTorch, recreates the network in TensorRT, imports weights from the trained model, and finally runs inference with a TensorRT engine.
Adding A Custom Layer To Your Caffe Network In TensorRT In Python	fc_plugin_caffe_mnist	Implements a FullyConnected layer using cuBLAS and cuDNN, wraps the implementation in a TensorRT plugin (with a corresponding plugin factory), and generates Python bindings

New Sample Name	Old Sample Name	Description
		for it using `pybind11`. These bindings are then used to register the plugin factory with the CaffeParser.
Adding A Custom Layer To Your TensorFlow Network In TensorRT In Python	uff_custom_plugin	Implements a clip layer (as a CUDA kernel), wraps the implementation in a TensorRT plugin (with a corresponding plugin creator), and generates a shared library module containing its code.
Object Detection With The ONNX TensorRT Backend In Python	yolov3_onnx	Implements a full ONNX-based pipeline for performing inference with the YOLOv3-608 network, including pre and post-processing.
Object Detection With SSD In Python	uff_ssd	Implements a full UFF-based pipeline for performing inference with an SSD (InceptionV2 feature extractor) network. The sample downloads a pretrained `ssd_inception_v2_coco_2017_11_17` model and uses it to perform inference. Additionally, it superimposes bounding boxes on the input image as a post-processing step.
INT8 Calibration In Python	int8_caffe_mnist	Demonstrates how to calibrate an engine to run in INT8 mode.
INT8 Calibration In Python	engine_refit_mnist	Trains an MNIST model in PyTorch, recreates the network in TensorRT with dummy weights, and finally refits the TensorRT engine with weights from the model.

1.1. C++ Samples

You can find the C++ samples in the **`/usr/src/tensorrt/samples`** directory. The following C++ samples are shipped with TensorRT:

- "Hello World" For TensorRT
- Building A Simple MNIST Network Layer By Layer
- Import The TensorFlow Model And Run Inference
- "Hello World" For TensorRT From ONNX
- Building And Running GoogleNet In TensorRT
- Building An RNN Network Layer By Layer
- Performing Inference In INT8 Using Custom Calibration

- Performing Inference In INT8 Precision
- Adding A Custom Layer To Your Network In TensorRT
- Object Detection With Faster R-CNN
- Object Detection With A TensorFlow SSD Network
- Movie Recommendation Using Neural Collaborative Filter (NCF)
- Movie Recommendation Using MPS (Multi-Process Service)
- Object Detection With SSD
- "Hello World" For Multilayer Perceptron (MLP)

Getting Started the A C++ Sample:

Every C++ sample includes a **`README.md`** file. Refer to the **`/usr/src/tensorrt/samples/<sample-name>/README.md`** file for detailed information about how the sample works, sample code, and step-by-step instructions on how to run and verify its output.

Running C++ Samples on Linux

If you installed TensorRT using the debian files, copy **`/usr/src/tensorrt`** to a new directory first before building the C++ samples. If you installed TensorRT using the tar file, then the samples are located in **`{TAR_EXTRACT_PATH}/samples`**. To build all the samples and then run one of the samples, use the following commands:

```
$ cd <samples_dir>
$ make -j4
$ cd ../bin
$ ./<sample_bin>
```

Running C++ Samples on Windows

All of the C++ samples on Windows are provided as Visual Studio Solution files. To build a sample, open its corresponding Visual Studio Solution file and build the solution. The output executable will be generated in **`(ZIP_EXTRACT_PATH)\bin`**. You can then run the executable directly or through Visual Studio.

1.2. Python Samples

You can find the Python samples in the **`/usr/src/tensorrt/samples/python`** directory. The following Python samples are shipped with TensorRT:

- Introduction To Importing Caffe, TensorFlow And ONNX Models Into TensorRT Using Python
- "Hello World" For TensorRT Using TensorFlow And Python
- "Hello World" For TensorRT Using PyTorch And Python
- Adding A Custom Layer To Your Caffe Network In TensorRT In Python

- Adding A Custom Layer To Your TensorFlow Network In TensorRT In Python
- Object Detection With The ONNX TensorRT Backend In Python
- Object Detection With SSD In Python
- INT8 Calibration In Python
- Engine Refit In Python

Getting Started the A Python Sample:

Every Python sample includes a **`README.md`** file. Refer to the **`/usr/src/tensorrt/samples/python/<sample-name>/README.md`** file for detailed information about how the sample works, sample code, and step-by-step instructions on how to run and verify its output.

Running Python Samples

Every Python sample includes a **`README.md`** and **`requirements.txt`** file. To run one of the Python samples, the process typically involves two steps:

1. Install the sample requirements:

   ```
   python<x> -m pip install -r requirements.txt
   ```

   where **`python<x>`** is either **`python2`** or **`python3`**.
2. Run the sample code with the **`data`** directory provided if the TensorRT sample data is not in the default location. For example:

   ```
   python<x> sample.py [-d DATA_DIR]
   ```

For more information on running samples, see the **`README.md`** file included with the sample.

Chapter 2. “HELLO WORLD” FOR TENSORRT

What Does This Sample Do?

This sample, sampleMNIST, is a simple hello world example that performs the basic setup and initialization of TensorRT using the Caffe parser.

Specifically, this sample:

- Performs the basic setup and initialization of TensorRT using the Caffe parser
- Imports a trained Caffe model using Caffe parser
- Preprocesses the input and stores the result in a managed buffer
- Builds an engine
- Serializes and deserializes the engine
- Uses the engine to perform inference on an input image

To verify whether the engine is operating correctly, this sample picks a 28x28 image of a digit at random and runs inference on it using the engine it created. The output of the network is a probability distribution on the digit, showing which digit is likely that in the image.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/sampleMNIST`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/sampleMNIST/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 3. BUILDING A SIMPLE MNIST NETWORK LAYER BY LAYER

What Does This Sample Do?

This sample, sampleMNISTAPI, uses the TensorRT API to build an engine for a model trained on the MNIST dataset. It creates the network layer by layer, sets up weights and inputs/outputs, and then performs inference. This sample is similar to sampleMNIST. Both of these samples use the same model weights, handle the same input, and expect similar output.

This sample uses a Caffe model that was trained on the MNIST dataset.

In contrast to sampleMNIST, which uses the Caffe parser to import the MNIST model, this sample uses the C++ API, individually creating every layer and loading weights from a trained weights file. For a detailed description of how to create layers using the C ++ API, see Creating A Network Definition In C++.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/sampleMNISTAPI`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/sampleMNISTAPI/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 4. IMPORT THE TENSORFLOW MODEL AND RUN INFERENCE

What Does This Sample Do?

This sample, sampleUffMNIST, imports a TensorFlow model trained on the MNIST dataset.

The MNIST TensorFlow model has been converted to UFF (Universal Framework Format) using the explanation described in Working With TensorFlow.

The UFF is designed to store neural networks as a graph. The NvUffParser that we use in this sample parses the UFF file in order to create an inference engine based on that neural network.

With TensorRT, you can take a TensorFlow trained model, export it into a UFF protobuf file (`.uff`) using the UFF converter, and import it using the UFF parser.

This sample loads the `.uff` file created from the TensorFlow MNIST model, parses it to create a TensorRT engine and performs inference using the created engine.

Specifically, this sample:

- Loads a trained TensorFlow model that has been pre-converted to the UFF file format
- Creates the UFF Parser (see Importing From TensorFlow Using Python)
- Uses the UFF Parser, registers inputs and outputs, and provides the dimensions and the order of the input tensor
- Builds an engine (see Building An Engine In C++)
- Uses the engine to perform inference 10 times and reports average inference time (see Performing Inference in C++)

Where Is This Sample Located?

This sample is installed in the **/usr/src/tensorrt/samples/sampleUffMNIST** directory.

Getting Started:

Refer to the **/usr/src/tensorrt/samples/sampleUffMNIST/README.md** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 5. “HELLO WORLD” FOR TENSORRT FROM ONNX

What Does This Sample Do?

This sample, sampleOnnxMNIST, converts a model trained on the MNIST dataset in Open Neural Network Exchange (ONNX) format to a TensorRT network and runs inference on the network.

ONNX is a standard for representing deep learning models that enables models to be transferred between frameworks.

This sample creates and runs the TensorRT engine from an ONNX model of the MNIST network. It demonstrates how TensorRT can consume an ONNX model as input to create a network.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/sampleOnnxMNIST`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/sampleOnnxMNIST/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 6. BUILDING AND RUNNING GOOGLENET IN TENSORRT

What Does This Sample Do?

This sample, sampleGoogleNet, demonstrates how to import a model trained with Caffe into TensorRT using GoogleNet as an example. Specifically, this sample builds a TensorRT engine from the saved Caffe model, sets input values to the engine, and runs it.

This sample constructs a network based on a saved Caffe model and network description. This sample comes with a pre-trained model called **`googlenet.caffemodel`** located in the **`data/googlenet`** directory. The model used by this sample was trained using ImageNet. For more information, see the BAIR/BVLC GitHub page. The sample reads two Caffe files to build the network:

`googlenet.prototxt`
The prototxt file that contains the network design.

`googlenet.caffemodel`
The model file which contains the trained weights for the network.

For more information, see Importing A Caffe Model Using The C++ Parser API.

The sample then builds the TensorRT engine using the constructed network. See Building an Engine in C++ for more information on this. Finally, the sample runs the engine with the test input (all zeroes) and reports if the sample ran as expected.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/sampleGoogleNet`** directory.

Getting Started:

Refer to the **/usr/src/tensorrt/samples/sampleGoogleNet/README.md** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 7. BUILDING AN RNN NETWORK LAYER BY LAYER

What Does This Sample Do?

This sample, sampleCharRNN, uses the TensorRT API to build an RNN network layer by layer, sets up weights and inputs/outputs and then performs inference. Specifically, this sample creates a CharRNN network that has been trained on the Tiny Shakespeare dataset. For more information about character level modeling, see char-rnn.

TensorFlow has a useful RNN Tutorial which can be used to train a word level model. Word level models learn a probability distribution over a set of all possible word sequence. Since our goal is to train a char level model, which learns a probability distribution over a set of all possible characters, a few modifications will need to be made to get the TensorFlow sample to work. These modifications can be seen here.

This sample provides a pre-trained model called **`model-20080.data-00000-of-00001`** located in the **`/usr/src/tensorrt/data/samples/char-rnn/model`** directory, therefore, training is not required for this sample. The model used by this sample was trained using tensorflow-char-rnn. This GitHub repository includes instructions on how to train and produce checkpoint that can be used by TensorRT.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/sampleCharRNN`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/sampleCharRNN/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 8. PERFORMING INFERENCE IN INT8 USING CUSTOM CALIBRATION

What Does This Sample Do?
This sample provides the steps involved when performing inference in 8-bit integer (INT8).

INT8 inference is available only on GPUs with compute capability 6.1 or 7.x.

This sample demonstrates how to:

- Perform INT8 calibration
- Perform INT8 inference
- Calibrate a network for execution in INT8
- Cache the output of the calibration to avoid repeating the process
- Repo your own experiments with Caffe in order to validate your results on ImageNet networks

Where Is This Sample Located?
This sample is installed in the **`/usr/src/tensorrt/samples/sampleINT8`** directory.

Notes About This Sample:
INT8 engines are built from 32-bit network definitions and require significantly more investment than building a 32-bit or 16-bit engine. In particular, the TensorRT builder must perform a process called calibration to determine how best to represent the weights and activations as 8-bit integers.

This sample is accompanied by the MNIST training set, but may also be used to calibrate and score other networks. To run the sample on MNIST, use the command line:

```
./sample_int8 mnist
```

Chapter 9. PERFORMING INFERENCE IN INT8 PRECISION

What Does This Sample Do?
This sample provides steps to perform INT8 Inference without using the INT8 inference calibrator; using the user provided per activation tensor dynamic range.

INT8 inference is available only on GPUs with compute capability 6.1 or 7.x.

This sample demonstrates how to:

- Set per tensor dynamic range.
- Set computation precision of a layer.
- Perform INT8 inference using the user defined dynamic range, without using INT8 calibration.

Where Is This Sample Located?
This sample is installed in the **`/usr/src/tensorrt/samples/sampleINT8API`** directory.

Notes About This Sample:
In order to perform INT8 inference, TensorRT expects you to provide dynamic range corresponding to each network tensor including input and output tensor. Dynamic range can be obtained using various methods including quantization aware training or simply recording the min and max per tensor values during training.

To run this sample, you will need per tensor dynamic range stored in a text file along with the ImageNet label reference file. We will perform INT8 inference on a classification network, for example, ResNet50, VGG19, MobileNet v2, etc.

To print usage information:

```
./sample_int8_api [-h or --help]
```

To run INT8 inference with your dynamic ranges:

```
./sample_int8_api [--model=model_file]
[--ranges=per_tensor_dynamic_range_file] [--image=image_file]
[--reference=reference_file] [--data=/path/to/data/dir]
[--useDLACore=<int>] [-v or --verbose]
```

Chapter 10. ADDING A CUSTOM LAYER TO YOUR NETWORK IN TENSORRT

What Does This Sample Do?

This sample, samplePlugin, defines a custom layer that supports multiple data formats and demonstrates how to serialize/deserialize plugin layers.. This sample also demonstrates how to use a fully connected plugin (**`FCPlugin`**) as a custom layer and the integration with NvCaffeParser.

This sample implements the MNIST model (**`data/samples/mnist/mnist.prototxt`**) with the difference that the custom layer implements the Caffe InnerProduct layer using *gemm*routines (Matrix Multiplication) in cuBLAS and tensor addition in cuDNN (bias offset). Normally, the Caffe InnerProduct layer can be implemented in TensorRT using the IFullyConnected layer. However, in this sample, we use **`FCPlugin`** for this layer as an example of how to use plugins. The sample demonstrates plugin usage through the **`IPluginExt`** interface and uses the **`nvcaffeparser1::IPluginFactoryExt`** to add the plugin object to the network.

Specifically, this sample:

- Defines the network
- Enables custom layers
- Builds the engine
- Serialize and deserialize
- Initializes the plugin and executes the custom layers

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/samplePlugin`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/samplePlugin/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 11. OBJECT DETECTION WITH FASTER R-CNN

What Does This Sample Do?

This sample, sampleFasterRCNN, uses TensorRT plugins, performs inference, and implements a fused custom layer for end-to-end inferencing of a Faster R-CNN model. Specifically, this sample demonstrates the implementation of a Faster R-CNN network in TensorRT, performs a quick performance test in TensorRT, implements a fused custom layer, and constructs the basis for further optimization, for example using INT8 calibration, user trained network, etc. The Faster R-CNN network is based on the paper Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.

Faster R-CNN is a fusion of Fast R-CNN and RPN (Region Proposal Network). The latter is a fully convolutional network that simultaneously predicts object bounds and objectness scores at each position. It can be merged with Fast R-CNN into a single network because it is trained end-to-end along with the Fast R-CNN detection network and thus shares with it the full-image convolutional features, enabling nearly cost-free region proposals. These region proposals will then be used by Fast R-CNN for detection.

Faster R-CNN is faster and more accurate than its predecessors (RCNN, Fast R-CNN) because it allows for an end-to-end inferencing and does not need standalone region proposal algorithms (like selective search in Fast R-CNN) or classification method (like SVM in RCNN).

The sampleFasterRCNN sample uses a plugin from the TensorRT plugin library to include a fused implementation of Faster R-CNN's Region Proposal Network (RPN) and ROIPooling layers. These particular layers are from the Faster R-CNN paper and are implemented together as a single plugin called **`RPNROIPlugin`**. This plugin is registered in the TensorRT Plugin Registry with the name **`RPROI_TRT`**.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/sampleFasterRNN`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/sampleFasterRCNN/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 12. OBJECT DETECTION WITH A TENSORFLOW SSD NETWORK

What Does This Sample Do?
This sample demonstrates how to:

- Preprocess the TensorFlow SSD network
- Perform inference on the SSD network in TensorRT
- Use TensorRT plugins to speed up inference

Where Is This Sample Located?
This sample is installed in the **`tensorrt/samples/sampleUffSSD`** directory.

Notes About This Sample:
The frozen graph for the SSD network is too large to include in the TensorRT package. Ensure you read the instructions in the README located at **`tensorrt/samples/sampleUffSSD`** for details on how to generate the network to run inference.

Chapter 13. MOVIE RECOMMENDATION USING NEURAL COLLABORATIVE FILTER (NCF)

What Does This Sample Do?

This sample, sampleMovieLens, is an end-to-end sample that imports a trained TensorFlow model and predicts the highest rated movie for each user. This sample demonstrates a simple movie recommender system using a multi-layer perceptron (MLP) based Neural Collaborative Filter (NCF) recommender.

Specifically, this sample demonstrates how to generate weights for a MovieLens dataset that TensorRT can then accelerate.

The network is trained in TensorFlow on the MovieLens dataset containing 6,040 users and 3,706 movies. The NCF recommender system is based off of the Neural Collaborative Filtering paper.

Each query to the network consists of a **`userID`** and list of **`MovieIDs`**. The network predicts the highest-rated movie for each user. As trained parameters, the network has embeddings for users and movies, and weights for a sequence of MLPs.

Where Is This Sample Located?

This sample in installed in the **`/usr/src/tensorrt/samples/sampleMovieLens`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/sampleMLP/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 14. MOVIE RECOMMENDATION USING MPS (MULTI-PROCESS SERVICE)

What Does This Sample Do?

This sample, sampleMovieLensMPS, is an end-to-end sample that imports a trained TensorFlow model and predicts the highest rated movie for each user using MPS (Multi-Process Service).

MPS allows multiple CUDA processes to share single GPU context. With MPS, multiple overlapping kernel execution and **`memcpy`** operations from different processes can be scheduled concurrently to achieve maximum utilization. This can be especially effective in increasing parallelism for small networks with low resource utilization such as those primarily consisting of a series of small MLPs.

This sample is identical to sampleMovieLens in terms of functionality, but is modified to support concurrent execution in multiple processes. Specifically, this sample demonstrates how to generate weights for a MovieLens dataset that TensorRT can then accelerate.

Currently, sampleMovieLensMPS supports only Linux x86-64 (includes Ubuntu and RedHat) desktop users.

Where Is This Sample Located?

This sample in installed in the **`usr/src/tensorrt/samples/sampleMovieLensMPS`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/sampleMovieLensMPS/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 15. OBJECT DETECTION WITH SSD

What Does This Sample Do?

This sample, sample SSD, is based on the SSD: Single Shot MultiBox Detector paper. The SSD network performs the task of object detection and localization in a single forward pass of the network. This network is built using the VGG network as a backbone and trained using PASCAL VOC 2007+ 2012 datasets.

Unlike Faster R-CNN, SSD completely eliminates proposal generation and subsequent pixel or feature resampling stages and encapsulates all computation in a single network. This makes SSD straightforward to integrate into systems that require a detection component.

This sample pre-processes the input to the SSD network and performs inference on the SSD network in TensorRT, using plugins to run layers that are not natively supported in TensorRT. Additionally, the sample can also be run in INT8 mode for which it first performs INT8 calibration and then does inference int INT8.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/sampleSSD`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/sampleSSD/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 16. “HELLO WORLD” FOR MULTILAYER PERCEPTRON (MLP)

What Does This Sample Do?

This sample, sampleMLP, is a simple hello world example that shows how to create a network that triggers the multilayer perceptron (MLP) optimizer. The generated MLP optimizer can then accelerate TensorRT.

This sample uses a publicly accessible TensorFlow tutorial to train a MLP network based on the MNIST data set and how to transform that data into a format that the samples use.

Specifically, this sample defines the network, triggers the MLP optimizer by creating a sequence of networks to increase performance, and creates a sequence of TensorRT layers that represent an MLP layer.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/sampleMLP`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/sampleMLP/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 17. INTRODUCTION TO IMPORTING CAFFE, TENSORFLOW AND ONNX MODELS INTO TENSORRT USING PYTHON

What Does This Sample Do?

This sample, introductory_parser_samples, is a Python sample which uses TensorRT and its included suite of parsers (the UFF, Caffe and ONNX parsers), to perform inference with ResNet-50 models trained with various different frameworks.

This sample is a collection of three smaller samples, with each focusing on a specific parser. The following sections describe how each sample works.

caffe_resnet50
: This sample demonstrates how to build an engine from a trained Caffe model using the Caffe parser and then run inference. The Caffe parser is used for Caffe2 models. After training, you can invoke the Caffe parser directly on the model file (usually `.caffemodel`) and deploy file (usually `.prototxt`).

onnx_resnet50
: This sample demonstrates how to build an engine from an ONNX model file using the open-source ONNX parser and then run inference. The ONNX parser can be used with any framework that supports the ONNX format (typically `.onnx` files).

uff_resnet50
: This sample demonstrates how to build an engine from a UFF model file (converted from a TensorFlow protobuf) and then run inference. The UFF parser is used for TensorFlow models. After freezing a TensorFlow graph and writing it to a protobuf file, you can convert it to UFF with the `convert-to-uff` utility included with TensorRT. This sample ships with a pre-generated UFF file.

Where Is This Sample Located?

This sample is installed in the `/usr/src/tensorrt/samples/python/introductory_parser_samples` directory.

Getting Started:

Refer to the **/usr/src/tensorrt/samples/python/introductory_parser_samples/README.md** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 18. “HELLO WORLD” FOR TENSORRT USING TENSORFLOW AND PYTHON

What Does This Sample Do?

This sample, end_to_end_tensorflow_mnist, trains a small, fully-connected model on the MNIST dataset and runs inference using TensorRT

This sample is an end-to-end Python sample that trains a small 3-layer model in TensorFlow and Keras, freezes the model and writes it to a protobuf file, converts it to UFF, and finally runs inference using TensorRT.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/python/end_to_end_tensorflow_mnist`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/python/end_to_end_tensorflow_mnist/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 19. “HELLO WORLD” FOR TENSORRT USING PYTORCH AND PYTHON

What Does This Sample Do?

This sample, network_api_pytorch_mnist, trains a convolutional model on the MNIST dataset and runs inference with a TensorRT engine.

This sample is an end-to-end sample that trains a model in PyTorch, recreates the network in TensorRT, imports weights from the trained model, and finally runs inference with a TensorRT engine. For more information, see Creating A Network Definition In Python.

The **`sample.py`** script imports the functions from the **`mnist.py`** script for training the PyTorch model, as well as retrieving test cases from the PyTorch Data Loader.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/python/network_api_pytorch_mnist`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/python/network_api_pytorch_mnist/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 20. ADDING A CUSTOM LAYER TO YOUR CAFFE NETWORK IN TENSORRT IN PYTHON

What Does This Sample Do?

This sample, fc_plugin_caffe_mnist, demonstrates how to implement a custom FullyConnected layer using cuBLAS and cuDNN, wraps the implementation in a TensorRT plugin (with a corresponding plugin factory), and generates Python bindings for it using **`pybind11`**. These bindings are then used to register the plugin factory with the CaffeParser.

The Caffe InnerProduct/FullyConnected layer is normally handled natively in TensorRT using the IFullyConnected layer. However, in this sample, we use a plugin implementation for instructive purposes.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/python/fc_plugin_caffe_mnist`** directory.

Getting started:

Refer to the **`/usr/src/tensorrt/samples/python/fc_plugin_caffe_mnist/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 21. ADDING A CUSTOM LAYER TO YOUR TENSORFLOW NETWORK IN TENSORRT IN PYTHON

What Does This Sample Do?

This sample, uff_custom_plugin, demonstrates how to use plugins written in C++ with the TensorRT Python bindings and UFF Parser. This sample uses the MNIST dataset.

This sample implements a clip layer (as a CUDA kernel), wraps the implementation in a TensorRT plugin (with a corresponding plugin creator) and then generates a shared library module containing its code. The user then dynamically loads this library in Python, which causes the plugin to be registered in TensorRT's PluginRegistry and makes it available to the UFF parser.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/python/uff_custom_plugin`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/python/uff_custom_plugin/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 22. OBJECT DETECTION WITH THE ONNX TENSORRT BACKEND IN PYTHON

What Does This Sample Do?

This sample, yolov3_onnx, implements a full ONNX-based pipeline for performing inference with the YOLOv3 network, with an input size of 608 x 608 pixels, including pre and post-processing. This sample is based on the YOLOv3-608 paper.

First, the original YOLOv3 specification from the paper is converted to the Open Neural Network Exchange (ONNX) format in **`yolov3_to_onnx.py`** (only has to be done once).

Second, this ONNX representation of YOLOv3 is used to build a TensorRT engine, followed by inference on a sample image in **`onnx_to_tensorrt.py`**. The predicted bounding boxes are finally drawn to the original input image and saved to disk.

After inference, post-processing including bounding-box clustering is applied. The resulting bounding boxes are eventually drawn to a new image file and stored on disk for inspection.

This sample is not supported on Ubuntu 14.04 and older. Additionally, the **`yolov3_to_onnx.py`** script does not support Python 3.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/python/yolov3_onnx`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/python/yolov3_onnx/README.md`** file for detailed information about how this sample works, sample code, and step-by-step instructions on how to run and verify its output.

Chapter 23. OBJECT DETECTION WITH SSD IN PYTHON

What Does This Sample Do?

This sample demonstrates a full UFF-based inference pipeline for performing inference with an SSD (InceptionV2 feature extractor) network.

The sample downloads a pretrained **`ssd_inception_v2_coco_2017_11_17`** model and uses it to perform inference. Additionally, it superimposes bounding boxes on the input image as a post-processing step.

It is also capable of validating the TensorRT engine using the VOC 2007 data set.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/python/uff_ssd`** directory.

For more details, see the **`README.md`** file included with this sample.

Chapter 24.
INT8 CALIBRATION IN PYTHON

What Does This Sample Do?

This sample demonstrates how to create an INT8 calibrator, build and calibrate an engine for INT8 mode, and finally run inference in INT8 mode.

During calibration, the calibrator retrieves a total of 1003 batches, with 100 images each. We have simplified the process of reading and writing a calibration cache in Python, so that it is now easily possible to cache calibration data to speed up engine builds.

During inference, the sample loads a random batch from the calibrator, then performs inference on the whole batch of 100 images.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/python/int8_caffe_mnist`** directory.

For more details, see the **`README.md`** file included with this sample.

Chapter 25. ENGINE REFIT IN PYTHON

What Does This Sample Do?

This sample demonstrates the engine refit functionality provided by TensorRT. The model first trains an MNIST model in PyTorch, then recreates the network in TensorRT. In the first pass, the weights for one of the conv layers (**`conv_1`**) is fed with dummy values resulting in an incorrect inference result. In the second pass, we refit the engine with the trained weights for the **`conv_1`** layer and run inference.

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/python/engine_refit_mnist`** directory.

For more details, see the **`README.md`** file included with this sample.

Notice

THE INFORMATION IN THIS GUIDE AND ALL OTHER INFORMATION CONTAINED IN NVIDIA DOCUMENTATION REFERENCED IN THIS GUIDE IS PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE INFORMATION FOR THE PRODUCT, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the product described in this guide shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

THE NVIDIA PRODUCT DESCRIBED IN THIS GUIDE IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED OR INTENDED FOR USE IN CONNECTION WITH THE DESIGN, CONSTRUCTION, MAINTENANCE, AND/OR OPERATION OF ANY SYSTEM WHERE THE USE OR A FAILURE OF SUCH SYSTEM COULD RESULT IN A SITUATION THAT THREATENS THE SAFETY OF HUMAN LIFE OR SEVERE PHYSICAL HARM OR PROPERTY DAMAGE (INCLUDING, FOR EXAMPLE, USE IN CONNECTION WITH ANY NUCLEAR, AVIONICS, LIFE SUPPORT OR OTHER LIFE CRITICAL APPLICATION). NVIDIA EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR SUCH HIGH RISK USES. NVIDIA SHALL NOT BE LIABLE TO CUSTOMER OR ANY THIRD PARTY, IN WHOLE OR IN PART, FOR ANY CLAIMS OR DAMAGES ARISING FROM SUCH HIGH RISK USES.

NVIDIA makes no representation or warranty that the product described in this guide will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this guide. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this guide, or (ii) customer product designs.

Other than the right for customer to use the information in this guide with the product, no other license, either expressed or implied, is hereby granted by NVIDIA under this guide. Reproduction of information in this guide is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, cuDNN, cuFFT, cuSPARSE, DALI, DIGITS, DGX, DGX-1, Jetson, Kepler, NVIDIA Maxwell, NCCL, NVLink, Pascal, Tegra, TensorRT, and Tesla are trademarks and/or registered trademarks of NVIDIA Corporation in the Unites States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright