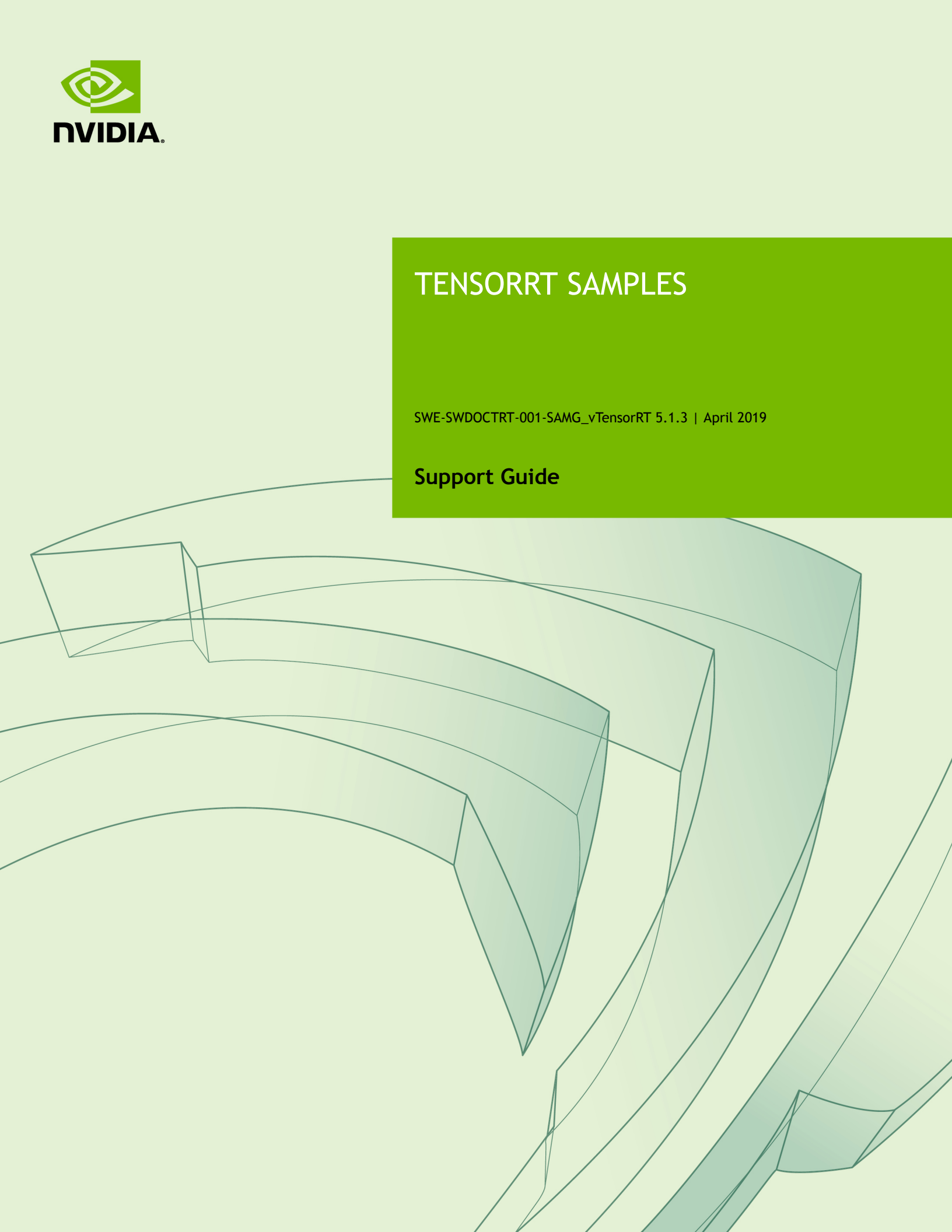

TENSORRT SAMPLES

SWE-SWDOCTRT-001-SAMG_vTensorRT 5.1.3 | April 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 With C++ Samples:

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
- Refitting An Engine In Python

Getting Started With Python Samples:

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.

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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

2.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

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

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.

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer
The Activation layer implements element-wise activation functions. Specifically, this sample uses the Activation layer with the type **`kRELU`**.

Convolution layer
The Convolution layer computes a 2D (channel, height, and width) convolution, with or without bias.

FullyConnected layer
The FullyConnected layer implements a matrix-vector product, with or without bias.

Pooling layer
The Pooling layer implements pooling within a channel. Supported pooling types are **`maximum`**, **`average`** and **`maximum-average blend`**.

Scale layer
The Scale layer implements a per-tensor, per-channel, or per-element affine transformation and/or exponentiation by constant values.

SoftMax layer
The SoftMax layer applies the SoftMax function on the input tensor along an input dimension specified by the user.

Running the sample

1. Compile this sample by running **`make`** in the **`<TensorRT root directory>/samples/sampleMNIST`** directory. The binary named **`sample_mnist`** will be created in the **`<TensorRT root directory>/bin`** directory.

   ```
   cd <TensorRT root directory>/samples/sampleMNIST
   make
   ```

   Where **`<TensorRT root directory>`** is where you installed TensorRT.
2. Run the sample to perform inference on the digit:

```
./sample_mnist [-h] [--datadir=/path/to/data/dir/] [--useDLA=N] [--fp16 or
 --int8]
```

This sample reads three Caffe files to build the network:

mnist.prototxt
The prototxt file that contains the network design.

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

mnist_mean.binaryproto
The binaryproto file which contains the means.

This sample can be run in FP16 and INT8 modes as well.

By default, the sample expects these files to be in either the **data/samples/mnist/** or **data/mnist/** directories. The list of default directories can be changed by adding one or more paths with **--datadir=/new/path/** as a command line argument.

3. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following; ASCII rendering of the input image with digit 3:

```
&&&& RUNNING TensorRT.sample_mnist # ./sample_mnist
 [I] Building and running a GPU inference engine for MNIST
 [I] Input:
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@#-:.-=@@@@@@@@@@@@@@
@@@@@%=      . *@@@@@@@@@@@@@
@@@@% .:+%%%  *@@@@@@@@@@@@@
@@@@+=#@@@@@# @@@@@@@@@@@@@@
@@@@@@@@@@%  @@@@@@@@@@@@@@
@@@@@@@@@@: *@@@@@@@@@@@@@@
@@@@@@@@@@- .@@@@@@@@@@@@@@
@@@@@@@@@:  #@@@@@@@@@@@@@@
@@@@@@@@:   +*%#@@@@@@@@@@@@
@@@@@@@%          :+*@@@@@@@@
@@@@@@@@#*+--.::     +@@@@@@
@@@@@@@@@@@@@@@@#=:.  +@@@@@
@@@@@@@@@@@@@@@@@@  .@@@@@
@@@@@@@@@@@@@@@@@@#. #@@@@
@@@@@@@@@@@@@@@@@@#  @@@@@
@@@@@@@@@%@@@@@@@@@- +@@@@@
@@@@@@@@#-@@@@@@@@*. =@@@@@@
@@@@@@@@ .+%%%%+=.  =@@@@@@@
@@@@@@@@           =@@@@@@@@
@@@@@@@@*=:   :--*@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@

[I] Output:
0:
1:
2:
```

```
3: **********
4:
5:
6:
7:
8:
9:

&&&& PASSED TensorRT.sample_mnist # ./sample_mnist
```

This output shows that the sample ran successfully; **PASSED**.

Sample `--help` options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
Usage: ./sample_mnist [-h or --help] [-d or --datadir=<path to data directory>]
 [--useDLACore=<int>]
--help Display help information
--datadir Specify path to a data directory, overriding the default. This option
 can be used multiple times to add multiple directories. If no data directories
 are given, the default is to use (data/samples/mnist/, data/mnist/)
--useDLACore=N Specify a DLA engine for layers that support DLA. Value can range
 from 0 to n-1, where n is the number of DLA engines on the platform.
--int8 Run in Int8 mode.
--fp16 Run in FP16 mode.
```

Additional resources

The following resources provide a deeper understanding about sampleMNIST:

MNIST

MNIST dataset

Documentation

- Working With TensorRT Using The C++ API
- NVIDIA’s TensorRT Documentation Library

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.

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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

3.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

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++.

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer
The Activation layer implements element-wise activation functions. Specifically, this sample uses the Activation layer with the type **`kRELU`**.

Convolution layer
The Convolution layer computes a 2D (channel, height, and width) convolution, with or without bias.

FullyConnected layer
The FullyConnected layer implements a matrix-vector product, with or without bias.

Pooling layer
The Pooling layer implements pooling within a channel. Supported pooling types are **`maximum`**, **`average`** and **`maximum-average blend`**.

Scale layer
The Scale layer implements a per-tensor, per-channel, or per-element affine transformation and/or exponentiation by constant values.

SoftMax layer
The SoftMax layer applies the SoftMax function on the input tensor along an input dimension specified by the user.

Prerequisites

When you build a network by individually creating every layer, ensure you provide the per-layer weights to TensorRT in host memory.

1. Extract the weights from their pre-trained model and deep learning framework. In this sample, the **`mnistapi.wts`** weights file stores the weights in a simple space delimited format, for example:

   ```
   <number of weight sets>
   [weights_name] [size] <data x size in hex>
   [weights_name] [size] <data x size in hex>
   [weights_name] [size] <data x size in hex>
   ```

   In the **`loadWeights`** function, the sample reads this file and creates a **`weightMap`** structure as a mapping from the **`weights_name`** to Weights.

2. Load the per-layer weights into host memory to pass to TensorRT during the network creation. For example:

```
IConvolutionLayer* conv1 = network->addConvolution(*scale_1->getOutput(0),
 20, DimsHW{5, 5}, weightMap["conv1filter"], weightMap["conv1bias"]);
```

Running the sample

1. Compile this sample by running **make** in the **<TensorRT root directory>/samples/sampleMNISTAPI** directory. The binary named **sample_mnist_api** will be created in the **<TensorRT root directory>/bin** directory.

```
cd <TensorRT root directory>/samples/sampleMNISTAPI
make
```

   Where **<TensorRT root directory>** is where you installed TensorRT.

2. Run the sample to perform inference on the digit:

```
./sample_mnist_api [-h] [--datadir=/path/to/data/dir/] [--useDLA=N]
```

3. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following; ASCII rendering of the input image with digit 9:

```
&&&& RUNNING TensorRT.sample_mnist_api # ./sample_mnist_api
 [I] Loading weights: ../../../../../../data/samples/mnist/mnistapi.wts
 [I] Input:
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@%.-@@@@@@@@@@@@
 @@@@@@@@@@@*-    %@@@@@@@@@@@
 @@@@@@@@@@= .-.  *@@@@@@@@@@@
 @@@@@@@@@= +@@@  *@@@@@@@@@@@
 @@@@@@@@* =@@@@  %@@@@@@@@@@@
 @@@@@@@@..@@@@%  @@@@@@@@@@@
 @@@@@@@# *@@@@-  @@@@@@@@@@@
 @@@@@@@: @@@@%   @@@@@@@@@@@
 @@@@@@@: @@@@-   @@@@@@@@@@@
 @@@@@@@: =+*= +: *@@@@@@@@@@
 @@@@@@@*.    +@: *@@@@@@@@@@
 @@@@@@@%#**#@@: *@@@@@@@@@@
 @@@@@@@@@@@@@@: -@@@@@@@@@@
 @@@@@@@@@@@@@@+ :@@@@@@@@@@
 @@@@@@@@@@@@@@*  @@@@@@@@@@
 @@@@@@@@@@@@@@   %@@@@@@@@@@
 @@@@@@@@@@@@@@   #@@@@@@@@@@
 @@@@@@@@@@@@@@@: +@@@@@@@@@@
 @@@@@@@@@@@@@@@- +@@@@@@@@@@
 @@@@@@@@@@@@@@@*:%@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

 [I] Output:
 0:
 1:
 2:
 3:
```

```
4:
5:
6:
7:
8:
9: **********

&&&& PASSED TensorRT.sample_mnist_api # ./sample_mnist_api
```

This output shows that the sample ran successfully; **PASSED**.

Sample `--help` options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
Usage: ./sample_mnist_api [-h or --help] [-d or --datadir=<path to data
 directory>] [--useDLACore=<int>]
--help Display help information
--datadir Specify path to a data directory, overriding the default. This option
 can be used multiple times to add multiple directories. If no data directories
 are given, the default is to use (data/samples/mnist/, data/mnist/)
--useDLACore=N Specify a DLA engine for layers that support DLA. Value can range
 from 0 to n-1, where n is the number of DLA engines on the platform.
--int8 Run in Int8 mode.
--fp16 Run in FP16 mode.
```

Additional resources

The following resources provide a deeper understanding about MNIST:

MNIST

MNIST dataset

Documentation

- Working With TensorRT Using The C++ API
- NVIDIA's TensorRT Documentation Library

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.

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.

A summary of the `README.md` file is included in this section for your reference, however, you should always refer to the `README.md` within the package for the most recent documentation updates.

4.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

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++)

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer
The Activation layer implements element-wise activation functions.

Convolution layer
The Convolution layer computes a 2D (channel, height, and width) convolution, with or without bias.

FullyConnected layer
The FullyConnected layer implements a matrix-vector product, with or without bias.

Pooling layer
The Pooling layer implements pooling within a channel. Supported pooling types are **`maximum`**, **`average`** and **`maximum-average blend`**.

Scale layer
The Scale layer implements a per-tensor, per-channel, or per-element affine transformation and/or exponentiation by constant values.

Shuffle layer
The Shuffle layer implements a reshape and transpose operator for tensors.

Running the sample

1. Compile this sample by running **make** in the **<TensorRT root directory>/samples/sampleUffMNIST** directory. The binary named **sample_uff_mnist** will be created in the **<TensorRT root directory>/bin** directory.

```
cd <TensorRT root directory>/samples/sampleUffMNIST
make
```

   Where **<TensorRT root directory>** is where you installed TensorRT.

2. Run the sample to create an MNIST engine from a UFF model and perform inference using it.

```
./sample_uff_mnist [-h or --help] [-d or --datadir=<path to data directory>]
 [--useDLACore=<int>] [--int8] [--fp16]
```

3. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following:

```
&&&& RUNNING TensorRT.sample_uff_mnist # ./sample_uff_mnist
 [I] ../../../../../../data/samples/mnist/lenet5.uff
 [I] Input:
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@+  :@@@@@@@@
@@@@@@@@@@@@@@%= :. --%@@@@@
@@@@@@@@@@@@@%. -@= - :@@@@@
@@@@@@@@@@@@@: -@@#%@@ #@@@@
@@@@@@@@@@@@: #@@@@@@@-#@@@@
@@@@@@@@@@@= #@@@@@@@@=%@@@@
@@@@@@@@@@= #@@@@@@@@@:@@@@@
@@@@@@@@@+ -@@@@@@@@@%.@@@@@
@@@@@@@@@::@@@@@@@@@@+-@@@@@
@@@@@@@@-.%@@@@@@@@@@.*@@@@@
@@@@@@@@ *@@@@@@@@@@ *@@@@@
@@@@@@@% %@@@@@@@@@%.-@@@@@@
@@@@@@@:*@@@@@@@@@+. %@@@@@@
@@@@@@# @@@@@@@@@# .*@@@@@@@
@@@@@@# @@@@@@@@=  +@@@@@@@@
@@@@@@# @@@@@@%. .+@@@@@@@@@
@@@@@@# @@@@@*. -%@@@@@@@@@@
@@@@@@# ---    =@@@@@@@@@@@@
@@@@@@#      *%@@@@@@@@@@@@@
@@@@@@@%: -=%@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

 [I] Output:
0 => 14.255573  : ***
1 => -4.830786  :
2 =>  1.091855  :
3 => -6.290083  :
4 => -0.835606  :
5 => -6.920589  :
6 =>  2.403986  :
7 => -6.011705  :
8 =>  0.730784  :
9 =>  1.500333  :
```

```
… (repeated 10 times)

[I] Average over 10 runs is 0.0643946 ms.
&&&& PASSED TensorRT.sample_uff_mnist # ./sample_uff_mnist
```

This output shows that the sample ran successfully; **PASSED**.

Sample `--help` options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
Usage: ./sample_uff_mnist [-h or --help] [-d or --datadir=<path to data
 directory>] [--useDLACore=<int>]
--help Display help information
--datadir Specify path to a data directory, overriding the default. This option
 can be used multiple times to add multiple directories. If no data directories
 are given, the default is to use (data/samples/mnist/, data/mnist/)
--useDLACore=N Specify a DLA engine for layers that support DLA. Value can range
 from 0 to n-1, where n is the number of DLA engines on the platform.
--int8 Run in Int8 mode.
--fp16 Run in FP16 mode.
```

Additional resources

The following resources provide a deeper understanding about the MNIST model from TensorFlow and using it in TensorRT:

Models

MNIST

Documentation

- Working With TensorRT Using The C++ API
- NVIDIA's TensorRT Documentation Library

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.

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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

5.1. README.md

Attention A summary of the **README.md** file is included here for your reference, however, you should always refer to the **README.md** within the package for the most recent documentation updates.

How does this sample work?

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.

Configuring the ONNX parser

The **IOnnxConfig** class is the configuration manager class for the ONNX parser. The configuration parameters can be set by creating an object of this class and set the model file.

Set the appropriate ONNX model in the **config** object where **onnx_filename** is a **c** string of the path to the filename containing that model:

```
IOnnxConfig config;
config.setModelFileName(onnx_filename);
```

The **createONNXParser** method requires a config object as an argument: **nvonnxparser::IONNXParser* parser = nvonnxparser::createONNXParser(*config);**

The ONNX model file is then passed onto the parser:

```
if (!parser->parse(onnx_filename, dataType))
{
   string msg("failed to parse onnx file");
   gLogger->log(nvinfer1::ILogger::Severity::kERROR, msg.c_str());
   exit(EXIT_FAILURE);
}
```

To view additional information about the network, including layer information and individual layer dimensions, issue the following call:

```
config.setPrintLayerInfo(true)
parser->reportParsingInfo();
```

Converting the ONNX model to a TensorRT network

The parser can convert the ONNX model to a TensorRT network which can be used for inference:

```
if (!parser->convertToTRTNetwork()) {
```

```
  string msg("ERROR, failed to convert onnx network into TRT network");
  gLogger->log(nvinfer1::ILogger::Severity::kERROR, msg.c_str());
    exit(EXIT_FAILURE);
}
```

To get the TensorRT network, issue the following call:
`nvinfer1::INetworkDefinition* network = parser->getTRTNetwork();`.

After the TensorRT network is built from the model, you can build the TensorRT engine and run inference.

Building the engine

To build the engine, create the builder and pass a logger created for TensorRT which is used for reporting errors, warnings and informational messages in the network:
`IBuilder* builder = createInferBuilder(gLogger);`

To build the engine from the generated TensorRT network, issue the following call:
`nvinfer1::ICudaEngine* engine = builder->buildCudaEngine(*network);`

After you build the engine, verify that the engine is running properly by confirming the output is what you expected. The output format of this sample should be the same as the output of sampleMNIST.

Running inference

To run inference using the created engine, see Performing Inference In C++.

It’s important to preprocess the data and convert it to the format accepted by the network. In this example, the sample input is in PGM (portable graymap) format. The model expects an input of image `1x28x28` scaled to between `[0,1]`.

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer
The Activation layer implements element-wise activation functions. Specifically, this sample uses the Activation layer with the type **`kRELU`**.

Convolution layer
The Convolution layer computes a 2D (channel, height, and width) convolution, with or without bias.

FullyConnected layer
The FullyConnected layer implements a matrix-vector product, with or without bias.

Pooling layer
The Pooling layer implements pooling within a channel. Supported pooling types are **`maximum`**, **`average`** and **`maximum-average blend`**.

Scale layer

The Scale layer implements a per-tensor, per-channel, or per-element affine transformation and/or exponentiation by constant values.

Shuffle layer

The Shuffle layer implements a reshape and transpose operator for tensors.

Running the sample

1. Compile this sample by running **make** in the **<TensorRT root directory>/samples/sampleOnnxMNIST** directory. The binary named **sample_onnx_mnist** will be created in the **<TensorRT root directory>/bin** directory.

```
cd <TensorRT root directory>/samples/sampleOnnxMNIST
make
```

   Where **<TensorRT root directory>** is where you installed TensorRT.

2. Run the sample to build and run the MNIST engine from the ONNX model.

```
./sample_onnx_mnist [-h or --help] [-d or --datadir=<path to data
 directory>] [--useDLACore=<int>] [--int8 or --fp16]
```

3. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following:

```
&&&& RUNNING TensorRT.sample_onnx_mnist # ./sample_onnx_mnist
 ----------------------------------------------------------------
 Input filename: ../../../../../../data/samples/mnist/mnist.onnx
 ONNX IR version: 0.0.3
 Opset version: 1
 Producer name: CNTK
 Producer version: 2.4
 Domain:
 Model version: 1
 Doc string:
 ----------------------------------------------------------------
 [I] Input:
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@*.   .*@@@@@@@@@@
 @@@@@@@@@*.      +@@@@@@@@@
 @@@@@@@@@. :#+    %@@@@@@@@@
 @@@@@@@@@.:@@@+   +@@@@@@@@@
 @@@@@@@@@.:@@@@:   +@@@@@@@@
 @@@@@@@@@=%@@@@:   +@@@@@@@@
 @@@@@@@@@@@@@@@@#   +@@@@@@@@
 @@@@@@@@@@@@@@@@*   +@@@@@@@@
 @@@@@@@@@@@@@@@@:   +@@@@@@@@
 @@@@@@@@@@@@@@@@:   +@@@@@@@@
 @@@@@@@@@@@@@@@*   .@@@@@@@@@
 @@@@@@@@@@%**%@.   *@@@@@@@@@
 @@@@@@@@%+.   .:   .@@@@@@@@@@
 @@@@@@@@=   ..     :@@@@@@@@@@
 @@@@@@@@:   *@@:   :@@@@@@@@@@
 @@@@@@@%    %@*     *@@@@@@@@@@
 @@@@@@@%    ++ ++   .%@@@@@@@@@
 @@@@@@@@-     +@@-   +@@@@@@@@@
 @@@@@@@@=   :*@@@#   .%@@@@@@@@
```

```
@@@@@@@@@@+*@@@@@%.   %@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

[I] Output:
Prob 0 0.0000 Class 0:
Prob 1 0.0000 Class 1:
Prob 2 1.0000 Class 2: **********
Prob 3 0.0000 Class 3:
Prob 4 0.0000 Class 4:
Prob 5 0.0000 Class 5:
Prob 6 0.0000 Class 6:
Prob 7 0.0000 Class 7:
Prob 8 0.0000 Class 8:
Prob 9 0.0000 Class 9:

&&&& PASSED TensorRT.sample_onnx_mnist # ./sample_onnx_mnist
```

This output shows that the sample ran successfully; **PASSED**.

Sample `--help` options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
Usage: ./sample_onnx_mnist [-h or --help] [-d or --datadir=<path to data
 directory>] [--useDLACore=<int>]
--help Display help information
--datadir Specify path to a data directory, overriding the default. This option
 can be used multiple times to add multiple directories. If no data directories
 are given, the default is to use (data/samples/mnist/, data/mnist/)
--useDLACore=N Specify a DLA engine for layers that support DLA. Value can range
 from 0 to n-1, where n is the number of DLA engines on the platform.
--int8 Run in Int8 mode.
--fp16 Run in FP16 mode.
```

Additional resources

The following resources provide a deeper understanding about the ONNX project and MNIST model:

ONNX

GitHub: ONNX

Models

- MNIST - Handwritten Digit Recognition
- GitHub: ONNX Models

Documentation

- Working With TensorRT Using The C++ API
- NVIDIA’s TensorRT Documentation Library

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.

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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

6.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

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.

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer
The Activation layer implements element-wise activation functions. Specifically, this sample uses the Activation layer with the type **`kRELU`**.

Concatenation layer
The Concatenation layer links together multiple tensors of the same non-channel sizes along the channel dimension.

Convolution layer
The Convolution layer computes a 2D (channel, height, and width) convolution, with or without bias.

FullyConnected layer
The FullyConnected layer implements a matrix-vector product, with or without bias.

LRN layer
The LRN layer implements cross-channel Local Response Normalization.

Pooling layer
The Pooling layer implements pooling within a channel. Supported pooling types are **`maximum`**, **`average`** and **`maximum-average blend`**.

SoftMax layer
The SoftMax layer applies the SoftMax function on the input tensor along an input dimension specified by the user.

Running the sample

1. Compile this sample by running **`make`** in the **`<TensorRT root directory>/samples/sampleGoogleNet`** directory. The binary named **`sample_googlenet`** will be created in the **`<TensorRT root directory>/bin`** directory.

```
cd <TensorRT root directory>/samples/sampleGoogleNet
make
```

Where **<TensorRT root directory>** is where you installed TensorRT.

2. Run the sample to build and run a GPU inference engine for GoogleNet.

```
./sample_googlenet -h --datadir=<path_to_data_directory> --useDLACore=N
```

Where **<path_to_data_directory>** is the path to your **data** directory.

By default, this sample assumes both **googlenet.prototxt** and **googlenet.caffemodel** files are located in either the **data/samples/googlenet/** or **data/googlenet/** directories. The default directory can be changed by supplying the **--datadir=<new_path/>** path as a command line argument.

3. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following:

```
&&&& RUNNING TensorRT.sample_googlenet # ./sample_googlenet
[I] Building and running a GPU inference engine for GoogleNet
[I] [TRT] Detected 1 input and 1 output network tensors.
[I] Ran ./sample_googlenet with:
[I] Input(s): data
[I] Output(s): prob
&&&& PASSED TensorRT.sample_googlenet # ./sample_googlenet
```

This output shows that the input to the sample is called **data**, the output tensor is called **prob** and the sample ran successfully; **PASSED**.

Sample --help options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
Usage: ./sample_googlenet [-h or --help] [-d or --
datadir=<path_to_data_directory>] [--useDLACore=<int>]

--help Display help information

--datadir Specify path to a data directory, overriding the default. This option
 can be used multiple times to add multiple directories. If no data directories
 are given, the default is to use data/samples/googlenet/ and data/googlenet/

--useDLACore=N Specify a DLA engine for layers that support DLA. Value can range
 from 0 to n-1, where n is the number of DLA engines on the platform.
```

Additional resources

The following resources provide a deeper understanding about GoogleNet:

GoogleNet

- Going Deeper with Convolutions
- BVLC/BAIR Caffe GitHub

Documentation

- Working With TensorRT Using The C++ API
- NVIDIA's TensorRT Documentation Library

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.

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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

7.1. README.md

Attention A summary of the `README.md` file contents is included here for your reference, however, you should always refer to the `README.md` within the package for the most recent documentation updates.

How does this sample work?

The CharRNN network is a fairly simple RNN network. The input into the network is a single character that is embedded into a vector of size 512. This embedded input is then supplied to a RNN layer containing two stacked LSTM cells. The output from the RNN layer is then supplied to a fully connected layer, which can be represented in TensorRT by a Matrix Multiply layer followed by an ElementWise sum layer. Constant layers are used to supply the weights and biases to the Matrix Multiply and ElementWise Layers, respectively. A TopK operation is then performed on the output of the ElementWise sum layer where `K = 1` to find the next predicted character in the sequence. For more information about these layers, see the TensorRT API documentation.

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.

If you wanted to train your own model and then perform inference with TensorRT, you will simply need to do a char to char comparison between TensorFlow and TensorRT.

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

ElementWise

The ElementWise layer, also known as the Eltwise layer, implements per-element operations. The ElementWise layer is used to execute the second step of the functionality provided by a FullyConnected layer.

MatrixMultiply

The MatrixMultiply layer implements matrix multiplication for a collection of matrices. The Matrix Multiplication layer is used to execute the first step of the functionality provided by a FullyConnected layer.

RNNv2

The RNNv2 layer implements recurrent layers such as Recurrent Neural Network (RNN), Gated Recurrent Units (GRU), and Long Short-Term Memory (LSTM). Supported types are RNN, GRU, and LSTM. It performs a recurrent operation, where the operation is defined by one of several well-known recurrent neural network (RNN) "cells". This is the first layer in the network is an RNN layer. This is added and configured in the **`addRNNv2Layer()`** function. Weights are set for each gate and layer individually. The input format for RNNv2 is BSE (Batch, Sequence, Embedding).

TopK

The TopK layer is used to identify the character that has the maximum probability of appearing next. The TopK layer finds the top K maximum (or minimum) elements along a dimension, returning a reduced tensor and a tensor of index positions.

Converting TensorFlow weights

If you want to train your own model and not use the pre-trained model included in this sample, you'll need to convert the TensorFlow weights into a format that TensorRT can use.

1. Locate TensorFlow weights dumping script: **`/usr/src/tensorrt/samples/common/dumpTFWts.py`**

   This script has been provided to extract the weights from the model checkpoint files that are created during training. Use **`dumpTFWts.py -h`** for directions on the usage of the script.
2. Convert the TensorFlow weights using the following command: **`dumpTFWts.py -m /path/to/checkpoint -o /path/to/output`**

Running the sample

1. Compile this sample by running **`make`** in the **`<TensorRT root directory>/samples/sampleCharRNN`** directory. The binary named **`sample_char_rnn`** will be created in the **`<TensorRT root directory>/bin`** directory.

   ```
   cd <TensorRT root directory>/samples/sampleCharRNN
   make
   ```

   Where **`<TensorRT root directory>`** is where you installed TensorRT.
2. Run the sample to generate characters based on the trained model:

   ```
   ./sample_char_rnn --datadir=<path/to/data>
   ```
3. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following:

   ```
   &&&& RUNNING TensorRT.sample_char_rnn # ./sample_char_rnn
    [I] [TRT] Detected 4 input and 3 output network tensors.
    [I] RNN Warmup: JACK
    [I] Expect: INGHAM:
   What shall I
    [I] Received: INGHAM:
   ```

```
What shall I
&&&& PASSED TensorRT.sample_char_rnn # ./sample_char_rnn
```

This output shows that the sample ran successfully; **PASSED**.

Sample `--help` options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
Usage: ./sample_char_rnn [-h or --help] [-d or --
datadir=<path_to_data_directory>] [--useDLACore=<int>]

--help Display help information

--datadir Specify path to a data directory, overriding the default. This option
 can be used multiple times to add multiple directories. If no data directories
 are given, the default is to use data/samples/char-rnn/ and data/char-rnn/

--useDLACore=N Specify a DLA engine for layers that support DLA. Value can range
 from 0 to n-1
```

Additional resources

The following resources provide a deeper understanding about RNN networks:

RNN networks

- GNMT
- NMT
- Transformer

Videos

Introduction to RNNs in TensorRT

Documentation

- Working With TensorRT Using The C++ API
- NVIDIA's TensorRT Documentation Library

Chapter 8. PERFORMING INFERENCE IN INT8 USING CUSTOM CALIBRATION

What Does This Sample Do?

This sample, sampleINT8, performs INT8 calibration and inference.

Specifically, this sample demonstrates how to perform inference in 8-bit integer (INT8). INT8 inference is available only on GPUs with compute capability 6.1 or 7.x. After the network is calibrated for execution in INT8, output of the calibration is cached to avoid repeating the process. You can then reproduce 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.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/sampleINT8/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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

8.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

INT8 engines are build from 32-bit network definitions, similarly to 32-bit and 16-bit engines, but with more configuration steps. In particular, the builder and network must be configured to use INT8, which requires per-tensor dynamic ranges. The INT8 calibrator can determine how best to represent weights and activations as 8-bit integers and sets the dynamic ranges accordingly. Alternatively, you can set custom dynamic ranges; this is covered in sampleINT8API.

This sample is accompanied by the MNIST training set located in the TensorRT-5.1.0.4/data/mnist/batches directory. The packaged MNIST model that is shipped with this sample is based on lenet.prototxt. For more information, see the MNIST BVLC Caffe example. This sample can also be used with other Image classification models, for example, deploy.prototxt.

The packaged data set file that is shipped with this sample is based on the MNIST data set. However, the batch file generation from the above data set is described in *Batch files for calibration*.

Defining the network

Defining a network for INT8 execution is exactly the same as for any other precision. Weights should be imported as FP32 values, and the builder will calibrate the network to find appropriate quantization factors to reduce the network to INT8 precision. This sample imports the network using the NvCaffeParser:

```
const IBlobNameToTensor* blobNameToTensor =
parser->parse(locateFile(deployFile).c_str(),
locateFile(modelFile).c_str(),
*network,
DataType::kFLOAT);
```

Setup the calibrator

Calibration is an additional step required when building networks for INT8. The application must provide TensorRT with sample input, in other words, calibration data. TensorRT will then perform inference in FP32 and gather statistics about intermediate activation layers that it will use to build the reduced precision INT8 engine.

Calibration data

Calibration must be performed using images representative of those which will be used at runtime. Since the sample is based around Caffe, any image preprocessing that caffe would perform prior to running the network (such as scaling, cropping, or mean subtraction) will be done in Caffe and captured as a set of files. The sample uses a utility class (BatchStream) to read these files and create appropriate input for calibration. Generation of these files is discussed in *Batch files for calibration*.

You can create calibration data stream (**calibrationStream**), for example:
```
BatchStream calibrationStream(CAL_BATCH_SIZE, NB_CAL_BATCHES);
```

The BatchStream class provides helper methods used to retrieve batch data. Batch stream object is used by the calibrator in order to retrieve batch data while calibrating. In general, the BatchStream class should provide implementation for **getBatch()** and **getBatchSize()** which can be invoked by **IInt8Calibrator::getBatch()** and **IInt8Calibrator::getBatchSize()**. Ideally, you can write your own custom BatchStream class to serve calibration data. For more information, see **BatchStream.h**.

The calibration data must be representative of the input provided to TensorRT at runtime; for example, for image classification networks, it should not consist of images from just a small subset of categories. For ImageNet networks, around 500 calibration images is adequate.

Calibrator interface

The application must implement the **IInt8Calibrator** interface to provide calibration data and helper methods for reading/writing the calibration table file.

We can create calibrator object (**calibrator**), for example:
```
std::unique_ptr<IInt8Calibrator> calibrator;
```

TensorRT provides 3 implementations for **IInt8Calibrator**:

1. IInt8EntropyCalibrator
2. IInt8EntropyCalibrator2
3. IInt8LegacyCalibrator

See **NvInfer.h** for more information on the **IInt8Calibrator** interface variants.

This sample uses **IInt8EntropyCalibrator2** by default. We can set the calibrator interface to use **IInt8EntropyCalibrator2** as shown:

```
calibrator.reset(new Int8EntropyCalibrator2(calibrationStream, FIRST_CAL_BATCH,
 gNetworkName, INPUT_BLOB_NAME));
```

where **calibrationStream** is a BatchStream object. The calibrator object should be configured to use the calibration batch stream.

In order to perform calibration, the interface must provide implementation for **getBatchSize()** and **getBatch()** to retrieve data from the BatchStream object.

The builder calls the **getBatchSize()** method once, at the start of calibration, to obtain the batch size for the calibration set. The method **getBatch()** is then called repeatedly to obtain batches from the application, until the method returns false. Every calibration batch must include exactly the number of images specified as the batch size.

```
bool getBatch(void* bindings[], const char* names[], int nbBindings)
override
```

```
{
 if (!mStream.next())
  return false;

 CHECK(cudaMemcpy(mDeviceInput, mStream.getBatch(), mInputCount *
sizeof(float), cudaMemcpyHostToDevice));
 assert(!strcmp(names[0], INPUT_BLOB_NAME));
 bindings[0] = mDeviceInput;
 return true;
}
```

For each input tensor, a pointer to input data in GPU memory must be written into the bindings array. The names array contains the names of the input tensors. The position for each tensor in the bindings array matches the position of its name in the names array. Both arrays have size **nbBindings**.

Since the calibration step is time consuming, you can choose to provide the implementation for **writeCalibrationCache()** to write calibration table to the appropriate location to be used for later runs. Then, implement **readCalibrationCache()** method to read calibration table file from desired location.

During calibration, the builder will check if the calibration file exists using **readCalibrationCache()**. The builder will re-calibrate only if either calibration file does not exist or is incompatible with the current TensorRT version or calibrator variant it was generated with.

For more information on implementing **IInt8Calibrator** interface, see **EntropyCalibrator.h**.

Calibration file

A calibration file stores activation scales for each network tensor. Activations scales are calculated using a dynamic range generated from a calibration algorithm, in other words, **abs(max_dynamic_range) / 127.0f**.

The calibration file is called **CalibrationTable<NetworkName>**, where **<NetworkName>** is the name of your network, for example **mnist**. The file is located in the T**ensorRT-x.x.x.x/data/mnist** directory, where **x.x.x.x** is your installed version of TensorRT.

If the **CalibrationTable** file is not found, the builder will run the calibration algorithm again to create it. The **CalibrationTable** contents include:

```
TRT-5100-EntropyCalibration2
data: 3c000889
conv1: 3c8954be
pool1: 3c8954be
conv2: 3dd33169
pool2: 3dd33169
ip1: 3daeff07
ip2: 3e7d50ec
prob: 3c010a14
```

Where:

- **<TRT-xxxx>-<xxxxxxx>** The TensorRT version followed by the calibration algorithm, for example, EntropyCalibration2.
- **<layer name> :** value corresponds to the floating point activation scales determined during calibration for each tensor in the network.

The **CalibrationTable** file is generated during the build phase while running the calibration algorithm. After the calibration file is created, it can be read for subsequent runs without running the calibration again. You can provide implementation for **readCalibrationCache()** to load calibration file from a desired location. If the read calibration file is compatible with calibrator type (which was used to generate the file) and TensorRT version, builder would skip the calibration step and use per tensor scales values from the calibration file instead.

Configuring the builder

1. Ensure that INT8 inference is supported on the platform: **if (!builder->platformHasFastInt8()) return false;**
2. Enable INT8 mode. Setting this flag ensures that builder auto-tuner will consider INT8 implementation. **builder->setInt8Mode(true);**
3. Pass the calibrator object (calibrator) to the builder. **builder->setInt8Calibrator(calibrator);**

Building the engine

After we configure the builder with INT8 mode and calibrator, we can build the engine similar to any FP32 engine. **ICudaEngine* engine = builder->buildCudaEngine(*network);**

Running the engine

After the engine has been built, it can be used just like an FP32 engine. For example, inputs and outputs remain in 32-bit floating point.

1. Retrieve the names of the input and output tensors to bind the buffers.

```
inputIndex = engine.getBindingIndex(INPUT_BLOB_NAME), outputIndex =
 engine.getBindingIndex(OUTPUT_BLOB_NAME);
```

2. Allocate memory for input and output buffers.

```
CHECK(cudaMalloc(&buffers[inputIndex], inputSize));
CHECK(cudaMalloc(&buffers[outputIndex], outputSize));
CHECK(cudaMemcpy(buffers[inputIndex], input, inputSize,
cudaMemcpyHostToDevice));
```

3. Create a CUDA stream and run inference.

```
cudaStream_t stream;
CHECK(cudaStreamCreate(&stream));
context.enqueue(batchSize, buffers, stream, nullptr);
```

4. Copy the CUDA buffer output to CPU output buffers for post processing.
   ```
   CHECK(cudaMemcpy(output, buffers[outputIndex], outputSize,
   cudaMemcpyDeviceToHost));
   ```

Verifying the output

This sample outputs Top-1 and Top-5 metrics for both FP32 and INT8 precision, as well as for FP16 if it is natively supported by the hardware. These numbers should be within 1%.

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer
The Activation layer implements element-wise activation functions.

Convolution layer
The Convolution layer computes a 2D (channel, height, and width) convolution, with or without bias.

FullyConnected layer
The FullyConnected layer implements a matrix-vector product, with or without bias.

SoftMax layer
The SoftMax layer applies the SoftMax function on the input tensor along an input dimension specified by the user.

Batch files for calibration

You can use the calibrated data that comes with this sample or you can generate the calibration data yourself. This sample uses batch files in order to calibrate for the INT8 data. The INT8 batch file is a binary file containing a set of N images, whose format is as follows:

- Four 32-bit integer values representing `{N, C, H, W}` representing the number of images N in the file, and the dimensions `{C, H, W}` of each image.
- N 32-bit floating point data blobs of dimensions `{C, H, W` that are used as inputs to the network.

If you want to generate calibration data yourself, refer to the following sections.

Generating batch files for Caffe users

Calibration requires that the images passed to the calibrator are in the same format as those that will be passed to TensorRT at runtime. For developers using Caffe for training, or who can easily transfer their network to Caffe, a supplied patchset supports capturing images after image preprocessing.

The instructions are provided so that users can easily use the sample code to test accuracy and performance on classification networks. In typical production use cases, applications will have such preprocessing already implemented, and should integrate with the calibrator directly.

These instructions are for Caffe git commit 473f143f9422e7fc66e9590da6b2a1bb88e50b2f. The patch file might be slightly different for later versions of Caffe.

1. Apply the patch. The patch can be applied by going to the root directory of the Caffe source tree and applying the patch with the command: **`patch -p1 < int8_caffe.patch`**
2. Rebuild Caffe and set the environment variable **`TENSORRT_INT8_BATCH_DIRECTORY`** to the location where the batch files are to be generated.

After training for 1000 iterations, there are 1003 batch files in the directory specified. This occurs because Caffe preprocesses three batches in advance of the current iteration.

These batch files can then be used with the **`BatchStream`** and **`Int8Calibrator`** to calibrate the data for INT8.

When running Caffe to generate the batch files, the training prototxt, and not the deployment prototxt, is required to be used.

The following example depicts the sequence of commands to run **`./sample_int8 mnist`** with Caffe generated batch files.

1. Navigate to the samples data directory and create an INT8 **`mnist`** directory:

```
cd <TensorRT>/samples/data
mkdir -p int8/mnist
cd int8/mnist
```

If Caffe is not installed anywhere, ensure you clone, checkout, patch, and build Caffe at the specific commit:

```
git clone https://github.com/BVLC/caffe.git
cd caffe
git checkout 473f143f9422e7fc66e9590da6b2a1bb88e50b2f
patch -p1 < <TensorRT>/samples/mnist/int8_caffe.patch
mkdir build
pushd build
cmake -DUSE_OPENCV=FALSE -DUSE_CUDNN=OFF ../
make -j4
popd
```

2. Download the **`mnist`** dataset from Caffe and create a link to it:

```
bash data/mnist/get_mnist.sh
bash examples/mnist/create_mnist.sh
cd ..
ln -s caffe/examples .
```

3. Set the directory to store the batch data, execute Caffe, and link the **mnist** files:

```
mkdir batches
export TENSORRT_INT8_BATCH_DIRECTORY=batches
caffe/build/tools/caffe test -gpu 0 -iterations 1000 -model examples/mnist/
lenet_train_test.prototxt -weights <TensorRT>/samples/mnist/mnist.caffemodel

ln -s <TensorRT>/samples/mnist/mnist.caffemodel .
ln -s <TensorRT>/samples/mnist/mnist.prototxt .
```

4. Execute sampleINT8 from the bin directory after being built with the following command: **./sample_int8 mnist**

Generating batch files for non-Caffe users

For developers that are not using Caffe, or cannot easily convert to Caffe, the batch files can be generated via the following sequence of steps on the input training data.

1. Subtract out the normalized mean from the dataset.
2. Crop all of the input data to the same dimensions.
3. Split the data into batch files where each batch file has **N** preprocessed images and labels.
4. Generate the batch files based on the format specified in *Batch files for calibration*.

The following example depicts the sequence of commands to run **./sample_int8 mnist** without Caffe.

1. Navigate to the samples data directory and create an INT8 **mnist** directory:

```
cd <TensorRT>/samples/data
mkdir -p int8/mnist/batches
cd int8/mnist
ln -s <TensorRT>/samples/mnist/mnist.caffemodel .
ln -s <TensorRT>/samples/mnist/mnist.prototxt .
```

2. Copy the generated batch files to the **int8/mnist/batches/** directory.
3. Execute sampleINT8 from the **bin** directory after being built with the following command: **./sample_int8 mnist**

Running the sample

1. Compile this sample by running **make** in the **<TensorRT root directory>/samples/sampleINT8** directory. The binary named **sample_int8 mnist** will be created in the **<TensorRT root directory>/bin** directory.

```
cd <TensorRT root directory>/samples/sampleINT8
make
```

   Where **<TensorRT root directory>** is where you installed TensorRT.
2. Run the sample on MNIST.

```
./sample_int8 mnist
```

3. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following:

```
&&&& RUNNING TensorRT.sample_int8 # ./sample_int8 mnist
[I] FP32 run:400 batches of size 100 starting at 100
[I] Processing next set of max 100 batches
[I] Processing next set of max 100 batches
[I] Processing next set of max 100 batches
[I] Processing next set of max 100 batches
[I] Top1: 0.9904, Top5: 1
[I] Processing 40000 images averaged 0.00170236 ms/image and 0.170236 ms/
batch.
[I] FP16 run:400 batches of size 100 starting at 100
[I] Processing next set of max 100 batches
[I] Processing next set of max 100 batches
[I] Processing next set of max 100 batches
[I] Processing next set of max 100 batches
[I] Top1: 0.9904, Top5: 1
[I] Processing 40000 images averaged 0.00128872 ms/image and 0.128872 ms/
batch.

INT8 run:400 batches of size 100 starting at 100
[I] Processing next set of max 100 batches
[I] Processing next set of max 100 batches
[I] Processing next set of max 100 batches
[I] Processing next set of max 100 batches
[I] Top1: 0.9908, Top5: 1
[I] Processing 40000 images averaged 0.000946117 ms/image and 0.0946117 ms/
batch.
&&&& PASSED TensorRT.sample_int8 # ./sample_int8 mnist
```

This output shows that the sample ran successfully; **PASSED**.

Sample `--help` options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
Usage: ./sample_int8 <network name> <optional params>

Optional params
--batch=N Set batch size (default = 100).
--start=N Set the first batch to be scored (default = 100). All batches before
 this batch will be used for calibration.
--score=N Set the number of batches to be scored (default = 400).
--search Search for best calibration. Can only be used with legacy calibration
 algorithm.
--legacy Use legacy calibration algorithm.
--useLegacyEntropy Use legacy Entropy calibration algorithm.
--useDLACore=N Enable execution on DLA for all layers that support dla. Value
 can range from 0 to n-1, where n is the number of DLA engines on the platform.
-h --help Print this help menu.
```

Additional resources

The following resources provide a deeper understanding how to perform inference in INT8 using custom calibration:

INT8

- 8-bit Inference with TensorRT
- INT8 Calibration Using C++

Models

MNIST lenet.prototxt

Blogs

- Fast INT8 Inference for Autonomous Vehicles with TensorRT 3
- Low Precision Inference with TensorRT
- 8-Bit Quantization and TensorFlow Lite: Speeding up mobile inference with low precision

Videos

- Inference and Quantization
- 8-bit Inference with TensorRT webinar

Documentation

- Working With TensorRT Using The C++ API
- NVIDIA's TensorRT Documentation Library

Chapter 9. PERFORMING INFERENCE IN INT8 PRECISION

What Does This Sample Do?

This sample, sampleINT8API, performs INT8 inference without using the INT8 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 and supports Image Classification ONNX models such as ResNet-50, VGG19, and MobileNet.

Specifically, this sample demonstrates how to:

- Use **`nvinfer1::ITensor::setDynamicRange`** to set per tensor dynamic range
- Use **`nvinfer1::ILayer::setPrecison`** to set computation precision of a layer
- Use **`nvinfer1::ILayer::setOutputType`** to set output tensor data type of a layer
- Perform INT8 inference without using INT8 calibration

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/sampleINT8API`** directory.

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/sampleINT8API/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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

9.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

In order to perform INT8 inference, you need to provide TensorRT with the dynamic range for each network tensor, including network input and output tensor. One way to choose the dynamic range is to use the TensorRT INT8 calibrator. But if you don't want to go that route (for example, let's say you used quantization-aware training or you just want to use the min and max tensor values seen during training), you can skip the INT8 calibration and set custom per-network tensor dynamic ranges. This sample implements INT8 inference for the ONNX ResNet-50 model using per-network tensor dynamic ranges specified in an input file.

This sample uses the ONNX ResNet-50 model.

Configuring the builder to use INT8 without the INT8 calibrator

1. Ensure that INT8 inference is supported on the platform: **`if (!builder->platformHasFastInt8()) return false;`**
2. Enable INT8 mode by setting the builder flag: **`builder->setInt8Mode(true);`**

   You can choose not to provide the INT8 calibrator. **`builder->setInt8Calibrator(nullptr);`**

   If you want to provide the calibrator, manual dynamic range will override calibration generate dynamic range/scale. See sampleINT8 on how to setup INT8 calibrator.
3. Optionally and for debugging purposes, the following flag configures the builder to choose type conforming layer implementation, if one exists.

   For example, in the case of **`DataType::kINT8`**, types are requested by **`setInt8Mode(true)`**. Setting this flag ensures that only the conformant layer implementation (with **`kINT8`** input and output types), are chosen even if a high performance non-conformat implementation is available. If no conformant layer exists, TensorRT will choose a non-conformant layer if available regardless of the setting for this flag.

   `builder->setStrictTypeConstraints(true);`

Configuring the network to use custom dynamic ranges and set per-layer precision

1. Iterate through the network to set the per activation tensor dynamic range.

```
readPerTensorDynamicRangeValue() // This function populates dictionary with
 keys=tensor_names, values=floating point dynamic range.
```

2. Set the dynamic range for network inputs:

```
string input_name = network->getInput(i)->getName();
network->getInput(i)->setDynamicRange(-tensorMap.at(input_name),
 tensorMap.at(input_name));
```

3. Set the dynamic range for per layer tensors:

```
string tensor_name = network->getLayer(i)->getOutput(j)->getName();
 network->getLayer(i)->getOutput(j)->setDynamicRange(-tensorMap.at(name),
 tensorMap.at(name));
```

4. Optional: This sample also showcases using layer precision APIs. Using these APIs, you can selectively choose to run the layer with user configurable precision and type constraints. It may not result in optimal inference performance, but can be helpful while debugging mixed precision inference.

   Iterate through the network to per layer precision:

```
auto layer = network->getLayer(i);
layer->setPrecision(nvinfer1::DataType::kINT8);
```

   This gives the layer's inputs and outputs a preferred type (for example, **`DataType::kINT8`**). You can choose a different preferred type for an input or output of a layer using:

```
for (int j=0; j<layer->getNbOutputs(); ++j) {
layer->setOutputType(j, nvinfer1::DataType::kFLOAT);
}
```

   Using layer precision APIs with **`builder->setStrictTypeConstraints(true)`** set, ensures that the requested layer precisions are obeyed by the builder irrespective of the performance. If no implementation is available with request precision constraints, the builder will choose the fastest implementation irrespective of precision and type constraints. For more information on using mixed precision APIs, see Setting The Layer Precision Using C++.

Building the engine

After we configure the builder with INT8 mode and calibrator, we can build the engine similar to any FP32 engine. **`ICudaEngine* engine = builder->buildCudaEngine(*network);`**

Running the engine

After the engine has been built, it can be used just like an FP32 engine. For example, inputs and outputs remain in 32-bit floating point.

1. Create an execution context and CUDA stream for the execution of this inference.

```
auto context = mEngine->createExecutionContext();
cudaStream_t stream;
cudaStreamCreate(&stream);
```

2. Copy the data from the host input buffers to the device input buffers.

```
buffers.copyInputToDeviceAsync(stream);
```

3. Enqueue the inference work and perform actual inference.

```
context->enqueue(batchSize, buffers.getDeviceBindings().data(),
 input_stream, nullptr))
```

4. Copy data from the device output buffers to the host output buffers.

```
buffers.copyOutputToHostAsync(stream);
```

5. Wait for the work in the stream to complete and release it.

```
cudaStreamSynchronize(stream);
cudaStreamDestroy(stream);
```

6. Check and print the output of the inference. `outputCorrect = verifyOutput(buffers);`

TensorRT API layers and ops

This sample demonstrates how you can enable INT8 inference using the following mixed precision APIs.

ITensor::SetDynamicRange
Set dynamic range for the tensor. Currently, only symmetric ranges are supported, therefore, the larger of the absolute values of the provided bounds is used.

ILayer::SetPrecision
Set the computational precision of this layer. Setting the precision forces TensorRT to choose the implementations which run at this precision. If precision is not set, TensorRT will select the computational precision based on performance considerations and the flags specified to the builder.

ILayer::SetOutputType
Set the output type of this layer. Setting the output type forces TensorRT to choose the implementations which generate output data with the given type. If the output type is not set, TensorRT will select the implementation based on performance considerations and the flags specified to the builder.

Prerequisites

In addition to the model file and input image, you will need per tensor dynamic range stored in a text file along with the ImageNet label reference file.

The following required files are included in the package and are located in the **data/int8_api** directory.

reference_labels.txt
The ImageNet reference label file.

resnet50_per_tensor_dynamic_range.txt
The ResNet-50 per tensor dynamic ranges file.

airliner.ppm
The image to be inferred.

1. Download the ONNX ResNet-50 model.

```
wget https://s3.amazonaws.com/download.onnx/models/opset_3/resnet50.tar.gz
```

2. Unpackage the model file. **tar -xvzf resnet50.tar.gz**
3. Copy **resnet50/model.onnx** to the **data/int8_api/resnet50.onnx** directory.

Running the sample

1. Compile this sample by running **make** in the **<TensorRT root directory>/samples/sampleINT8API** directory. The binary named **sample_int8_api** will be created in the **<TensorRT root directory>/bin** directory.

```
cd <TensorRT root directory>/samples/sampleINT8API
make
```

Where **<TensorRT root directory>** is where you installed TensorRT.

2. Run the sample to perform INT8 inference on a classification network, for example, ResNet-50.

```
./sample_int8_api [-v or --verbose]
```

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]
```

3. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following:

```
&&&& RUNNING TensorRT.sample_int8_api # ./sample_int8_api
[I] Please follow README.md to generate missing input files.
[I] Validating input parameters. Using following input files for inference.
[I] Model File: ../../../../../../../../../data/samples/int8_api/
resnet50.onnx
[I] Image File: ../../../../../../../../../data/samples/int8_api/
airliner.ppm
[I] Reference File: ../../../../../../../../../data/samples/int8_api/
reference_labels.txt
[I] Dynamic Range File: ../../../../../../../../../data/samples/int8_api/
resnet50_per_tensor_dynamic_range.txt
[I] Building and running a INT8 GPU inference engine
 for ../../../../../../../../../data/samples/int8_api/resnet50.onnx
[I] Setting Per Layer Computation Precision
[I] Setting Per Tensor Dynamic Range
[W] [TRT] Calibrator is not being used. Users must provide dynamic range for
 all tensors.
[W] [TRT] Warning: no implementation of (Unnamed Layer* 173) [Fully
 Connected] obeys the requested constraints, using a higher precision type
[W] [TRT] Warning: no implementation of (Unnamed Layer* 174) [Softmax] obeys
 the requested constraints, using a higher precision type
```

```
[I] sampleINT8API result: Detected:
[I] [1] space shuttle
[I] [2] airliner
[I] [3] warplane
[I] [4] projectile
[I] [5] wing
&&&& PASSED TensorRT.sample_int8_api # ./sample_int8_api
```

This output shows that the sample ran successfully; **PASSED**.

Sample `--help` options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
Usage: ./sample_int8_api [-h or --help] [--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]
-h or --help. Display This help information
--model=model_file.onnx or /absolute/path/to/model_file.onnx. Generate model
 file using README.md in case it does not exists. Default to resnet50.onnx
--image=image.ppm or /absolute/path/to/image.ppm. Image to infer. Defaults to
 airlines.ppm
--reference=reference.txt or /absolute/path/to/reference.txt. Reference labels
 file. Defaults to reference_labels.txt
--ranges=ranges.txt or /absolute/path/to/ranges.txt. Specify
 custom per tensor dynamic range for the network. Defaults to
 resnet50_per_tensor_dynamic_range.txt
--write_tensors. Option to generate file containing network tensors name. By
 default writes to network_tensors.txt file. To provide user defined file name
 use additional option --network_tensors_file. See --network_tensors_file option
 usage for more detail.
--network_tensors_file=network_tensors.txt or /absolute/path/to/
network_tensors.txt. This option needs to be used with --write_tensors
 option. Specify file name (will write to current execution directory) or
 absolute path to file name to write network tensor names file. Dynamic range
 corresponding to each network tensor is required to run the sample. Defaults to
 network_tensors.txt
--data=/path/to/data/dir. Specify data directory to search for above files
 in case absolute paths to files are not provided. Defaults to data/samples/
int8_api/ or data/int8_api/
--useDLACore=N. Specify a DLA engine for layers that support DLA. Value can
 range from 0 to n-1, where n is the number of DLA engines on the platform.
--verbose. Outputs per tensor dynamic range and layer precision info for the
 network
```

Models other than ResNet-50 with custom configuration

In order to use this sample with other model files with a custom configuration, perform the following steps:

1. Download the Image Classification model files from GitHub.
2. Create an input image with a PPM extension. Resize it with the dimensions of 224x224x3.

3. Create a file called **reference_labels.txt**.

Ensure each line corresponds to a single imagenet label. You can download the imagenet 1000 class human readable labels from here. The reference label file contains only a single label name per line, for example, **0:'tench, Tinca tinca'** is represented as **tench**.

4. Create a file called **<network_name>_per_tensor_dynamic_ranges.txt**.

   a. Before you can create the dynamic range file, you need to generate the tensor names by providing the dynamic range for each network tensor.

   This sample provides an option to write names of the network tensors to a file, for example **network_tensors.txt**. This file can then be used to generate the **<network_name>_per_tensor_dynamic_ranges.txt** file in step 4-b below. To generate the list of network tensors file, perform the following steps:

   1. Write network tensors to a file:

```
./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]
```

   sampleINT8API needs the following files to build the network and run inference:

   <network>.onnx
   The model file which contains the network and trained weights.

   Reference_labels.txt
   Labels reference file i.e. ground truth ImageNet 1000 class mappings.

   Per_tensor_dynamic_range.txt
   Custom per tensor dynamic range file or you can simply override them by iterating through network layers.

   Image_to_infer.ppm
   PPM Image to run inference with.

By default, the sample expects these files to be in either the **data/samples/int8_api/** or **data/int8_api/** directories. The list of default directories can be changed by adding one or more paths with **--data=/new/path** as a command line argument.

   2. To create the **<network_name>_per_tensor_dynamic_ranges.txt.txt** file, ensure each line corresponds to the tensor name and floating point dynamic range, for example **<tensor_name> : <float dynamic range>**.

   Tensor names generated in the **network_tensors.txt** file (step 4-a) can be used here to represent **<tensor_name>**. The dynamic range can either be

obtained from training (by measuring the **`min`** and **`max`** value of activation tensors in each epoch) or from using custom post processing techniques (similar to TensorRT calibration). You can also choose to use a dummy per tensor dynamic range to run the sample.

INT8 inference accuracy may reduce when dummy/random dynamic ranges are provided.

Additional resources

The following resources provide a deeper understanding how to perform inference in INT8:

INT8API

Setting Per-Tensor Dynamic Range Using C++

Generate per tensor dynamic range

1. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference
2. Quantizing deep convolutional networks for efficient inference: A whitepaper
3. 8-bit Inference with TensorRT

Models

- ONNX ResNet-50 model
- Image Classification model files

Blogs

- Why are Eight Bits Enough for Deep Neural Networks?
- What I've learned about neural network quantization

Videos

- Inference and Quantization
- 8-bit Inference with TensorRT webinar

Documentation

- Working With TensorRT Using The C++ API
- NVIDIA's TensorRT Documentation Library

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.

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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

10.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

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.

Defining the network

The **FCPlugin** redefines the InnerProduct layer, which has a single output. Accordingly, **getNbOutputs** returns **1** and **getOutputDimensions** includes validation checks and returns the dimensions of the output:

```
Dims getOutputDimensions(int index, const Dims* inputDims,
       int nbInputDims) override
{
 assert(index == 0 && nbInputDims == 1 &&
     inputDims[0].nbDims == 3);
 assert(mNbInputChannels == inputDims[0].d[0] *
          inputDims[0].d[1] *
          inputDims[0].d[2]);
 return DimsCHW(mNbOutputChannels, 1, 1);
}
```

Enabling custom layers in NvCaffeParser

The model is imported using the Caffe parser (see Importing A Caffe Model Using The C++ Parser API and Using Custom Layers When Importing A Model From a Framework). To use the **FCPlugin** implementation for the InnerProduct layer, a plugin factory is defined which recognizes the name of the InnerProduct layer (inner product **ip2** in Caffe).

```
bool isPlugin(const char* name) override
{   return !strcmp(name, "ip2"); }
```

The factory can then instantiate **FCPlugin** objects as directed by the parser. The **createPlugin** method receives the layer name, and a set of weights extracted from the Caffe model file, which are then passed to the plugin constructor. Since the lifetime of the weights and that of the newly created plugin are decoupled, the plugin makes a copy of the weights in the constructor.

```
virtual nvinfer1::IPlugin* createPlugin(const char* layerName, const
 nvinfer1::Weights* weights, int nbWeights) override
{
 …
 mPlugin =
   std::unique_ptr<FCPlugin>(new FCPlugin(weights,nbWeights));

 return mPlugin.get();
```

```
}
```

Building the engine

FCPlugin does not need any scratch space, therefore, for building the engine, the most important methods deal with the formats supported and the configuration. **FCPlugin** supports two formats: NCHW in both single and half precision as defined in the **supportsFormat** method.

```
bool supportsFormat(DataType type, PluginFormat format) const override
{
 return (type == DataType::kFLOAT || type == DataType::kHALF) &&
     format == PluginFormat::kNCHW;
}
```

Supported configurations are selected in the building phase. The builder selects a configuration with the networks **configureWithFormat()** method, to give it a chance to select an algorithm based on its inputs. In this example, the inputs are checked to ensure they are in a supported format, and the selected format is recorded in a member variable. No other information needs to be stored in this simple case; in more complex cases, you may need to do so or even choose an ad-hoc algorithm for the given configuration.

```
void configureWithFormat(..., DataType type, PluginFormat format, ...) override
{
 assert((type == DataType::kFLOAT || type == DataType::kHALF) &&
   format == PluginFormat::kNCHW);
 mDataType = type;

}
```

The configuration takes place at build time, therefore, any information or state determined here that is required at runtime should be stored as a member variable of the plugin, and serialized and deserialized.

Serializing and deserializing

Fully compliant plugins support serialization and deserialization, as described in Serializing A Model In C++. In the example, **FCPlugin** stores the number of channels and weights, the format selected, and the actual weights. The size of these variables makes up for the size of the serialized image; the size is returned by **getSerializationSize**:

```
virtual size_t getSerializationSize() override
{
 return sizeof(mNbInputChannels) + sizeof(mNbOutputChannels) +
     sizeof(mBiasWeights.count) + sizeof(mDataType) +
     (mKernelWeights.count + mBiasWeights.count) *
     type2size(mDataType);
}
```

Eventually, when the engine is serialized, these variables are serialized, the weights converted is needed, and written on a buffer:

```
virtual void serialize(void* buffer) override
{
 char* d = static_cast<char*>(buffer), *a = d;
 write(d, mNbInputChannels);
 ...
 convertAndCopyToBuffer(d, mKernelWeights);
 convertAndCopyToBuffer(d, mBiasWeights);
 assert(d == a + getSerializationSize());
}
```

Then, when the engine is deployed, it is deserialized. As the runtime scans the serialized image, when a plugin image is encountered, it create a new plugin instance via the factory. The plugin object created during deserialization (shown below using new) is destroyed when the engine is destroyed by calling **`FCPlugin::destroy()`**.

```
IPlugin* createPlugin(...) override
{
 …

 return new FCPlugin(serialData, serialLength);
}
```

In the same order as in the serialization, the variables are read and their values restored. In addition, at this point the weights have been converted to selected format and can be stored directly on the device.

```
FCPlugin(const void* data, size_t length)
{
 const char* d = static_cast<const char*>(data), *a = d;
 read(d, mNbInputChannels);
 ...
 deserializeToDevice(d, mDeviceKernel,
      mKernelWeights.count*type2size(mDataType));
 deserializeToDevice(d, mDeviceBias,
      mBiasWeights.count*type2size(mDataType));
 assert(d == a + length);
}
```

Resource management and execution

Before a custom layer is executed, the plugin is initialized. This is where resources are held for the lifetime of the plugin and can be acquired and initialized. In this example, weights are kept in CPU memory at first, so that during the build phase, for each configuration tested, weights can be converted to the desired format and then copied to the device in the initialization of the plugin. The method **`initialize`** creates the required cuBLAS and cuDNN handles, sets up tensor descriptors, allocates device memory, and copies the weights to device memory. Conversely, terminate destroys the handles and frees the memory allocated on the device.

```
int initialize() override
{
 CHECK(cudnnCreate(&mCudnn));
 CHECK(cublasCreate(&mCublas));
 …
 if (mKernelWeights.values != nullptr)
  convertAndCopyToDevice(mDeviceKernel, mKernelWeights);
 …
```

```
}
```

The core of the plugin is **enqueue**, which is used to execute the custom layer at runtime. The **call** parameters include the actual batch size, inputs, and outputs. The handles for cuBLAS and cuDNN operations are placed on the given stream; then, according to the data type and format configured, the plugin executes in single or half precision.

The two handles are part of the plugin object, therefore, the same engine cannot be executed concurrently on multiple streams. In order to enable multiple streams of execution, plugins must be re-entrant and handle stream-specific data accordingly.

```
virtual int enqueue(int batchSize, const void*const * inputs, void**
 outputs, ...) override
{
 ...
 cublasSetStream(mCublas, stream);
 cudnnSetStream(mCudnn, stream);
 if (mDataType == DataType::kFLOAT)
 {...}
 else
 {
  CHECK(cublasHgemm(mCublas, CUBLAS_OP_T, CUBLAS_OP_N,
        mNbOutputChannels, batchSize,
        mNbInputChannels, &oneh,
        mDeviceKernel), mNbInputChannels,
        inputs[0], mNbInputChannels, &zeroh,
        outputs[0], mNbOutputChannels));
 }
 if (mBiasWeights.count)
 {
  cudnnDataType_t cudnnDT = mDataType == DataType::kFLOAT ?
          CUDNN_DATA_FLOAT : CUDNN_DATA_HALF;
  ...
 }
 return 0;
}
```

The plugin object created in the sample is cloned by each of the network, builder, and engine by calling the **FCPlugin::clone()** method. The **clone()** method calls the plugin constructor and can also clone plugin parameters, if necessary.

```
IPluginExt* clone()
 {
  return new FCPlugin(&mKernelWeights, mNbWeights, mNbOutputChannels);
 }
```

The cloned plugin objects are deleted when the network, builder, or engine are destroyed. This is done by invoking the **FCPlugin::destroy()** method. **void destroy() { delete this; }**

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer
The Activation layer implements element-wise activation functions. Specifically, this sample uses the Activation layer with the type **kRELU**.

Convolution layer
The Convolution layer computes a 2D (channel, height, and width) convolution, with or without bias.

FullyConnected layer
The FullyConnected layer implements a matrix-vector product, with or without bias.

Pooling layer
The Pooling layer implements pooling within a channel. Supported pooling types are **maximum**, **average** and **maximum-average blend**.

Scale layer
The Scale layer implements a per-tensor, per-channel, or per-element affine transformation and/or exponentiation by constant values.

SoftMax layer
The SoftMax layer applies the SoftMax function on the input tensor along an input dimension specified by the user.

Running the sample

1. Compile this sample by running **make** in the **<TensorRT root directory>/samples/samplePlugin** directory. The binary named **sample_plugin** will be created in the **<TensorRT root directory>/bin** directory.

```
cd <TensorRT root directory>/samples/samplePlugin
make
```

   Where **<TensorRT root directory>** is where you installed TensorRT.
2. Run the sample to perform inference on the digit:

```
./sample_plugin
```

3. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following:

```
&&&& RUNNING TensorRT.sample_plugin # ./build/x86_64-linux/sample_plugin
 [I] [TRT] Detected 1 input and 1 output network tensors.
 [I] Input:
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@%.-@@@@@@@@@@
@@@@@@@@@@@*-    %@@@@@@@@@@
@@@@@@@@@@= .-.  *@@@@@@@@@@
@@@@@@@@@= +@@@  *@@@@@@@@@@
@@@@@@@@* =@@@@  %@@@@@@@@@@
@@@@@@@@..@@@@%  @@@@@@@@@@
@@@@@@@# *@@@@-  @@@@@@@@@@
@@@@@@@: @@@@%   @@@@@@@@@@
@@@@@@@: @@@@-   @@@@@@@@@@
```

```
@@@@@@@: =+*= +: *@@@@@@@@@@
@@@@@@@*.    +@: *@@@@@@@@@@
@@@@@@@@%#**#@@: *@@@@@@@@@@
@@@@@@@@@@@@@@@: -@@@@@@@@@@
@@@@@@@@@@@@@@@+ :@@@@@@@@@@
@@@@@@@@@@@@@@@*  @@@@@@@@@@
@@@@@@@@@@@@@@@@  %@@@@@@@@@
@@@@@@@@@@@@@@@@  #@@@@@@@@@
@@@@@@@@@@@@@@@@: +@@@@@@@@@
@@@@@@@@@@@@@@@@- +@@@@@@@@@
@@@@@@@@@@@@@@@@*:%@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@

[I] Output:
0:
1:
2:
3:
4:
5:
6:
7:
8:
9: **********

&&&& PASSED TensorRT.sample_plugin # ./build/x86_64-linux/sample_plugin
```

This output shows that the sample ran successfully; **PASSED**.

Sample `--help` options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
Usage: ./sample_plugin [-h or --help] [-d or --datadir=<path to data directory>]
 [--useDLACore=<int>]
--help Display help information
--datadir Specify path to a data directory, overriding the default. This option
 can be used multiple times to add multiple directories. If no data directories
 are given, the default is to use (data/samples/mnist/, data/mnist/)
--useDLACore=N Specify a DLA engine for layers that support DLA. Value can range
 from 0 to n-1, where n is the number of DLA engines on the platform.
--int8 Run in Int8 mode.
--fp16 Run in FP16 mode.
```

Additional resources

The following resources provide a deeper understanding about samplePlugin:
Models

- Training LeNet on MNIST with Caffe
- lenet.prototxt

Documentation

- Working With TensorRT Using The C++ API
- NVIDIA's TensorRT Documentation Library

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.

Where Is This Sample Located?

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

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

11.1. README.md

Attention A summary of the **README.md** file is included here for your reference, however, you should always refer to the **README.md** within the package for the most recent documentation updates.

How does this sample work?

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**.

Preprocessing the input

Faster R-CNN takes 3 channel 375x500 images as input. Since TensorRT does not depend on any computer vision libraries, the images are represented in binary **R**, **G**, and **B** values for each pixels. The format is Portable PixMap (PPM), which is a netpbm color image format. In this format, the **R**, **G**, and **B** values for each pixel are usually represented by a byte of integer (0-255) and they are stored together, pixel by pixel.

However, the authors of Faster R-CNN have trained the network such that the first Convolution layer sees the image data in **B**, **G**, and **R** order. Therefore, you need to reverse the order when the PPM images are being put into the network input buffer.

```
float* data = new float[N*INPUT_C*INPUT_H*INPUT_W];
// pixel mean used by the Faster R-CNN's author
float pixelMean[3]{ 102.9801f, 115.9465f, 122.7717f }; // also in BGR order
for (int i = 0, volImg = INPUT_C*INPUT_H*INPUT_W; i < N; ++i)
{
 for (int c = 0; c < INPUT_C; ++c)
 {
  // the color image to input should be in BGR order
```

```
  for (unsigned j = 0, volChl = INPUT_H*INPUT_W; j < volChl; ++j)
data[i*volImg + c*volChl + j] =  float(ppms[i].buffer[j*INPUT_C + 2 - c]) -
 pixelMean[c];
 }
}
```

There is a simple PPM reading function called **readPPMFile**.

The **readPPMFile** function will not work correctly if the header of the PPM image contains any annotations starting with **#**.

Furthermore, within the sample there is another function called **writePPMFileWithBBox**, that plots a given bounding box in the image with one-pixel width red lines.

In order to obtain PPM images, you can easily use the command-line tools such as ImageMagick to perform the resizing and conversion from JPEG images.

If you choose to use off-the-shelf image processing libraries to preprocess the inputs, ensure that the TensorRT inference engine sees the input data in the form that it is supposed to.

Defining the network

The network is defined in a prototxt file which is shipped with the sample and located in the **data/faster-rcnn** directory. The prototxt file is very similar to the one used by the inventors of Faster R-CNN except that the RPN and the ROI pooling layer is fused and replaced by a custom layer named **RPROIFused**.

This sample uses the plugin registry to add the plugin to the network. The Caffe parser adds the plugin object to the network based on the layer name as specified in the Caffe prototxt file, for example, **RPROI**.

Building the engine

To build the TensorRT engine, see Building An Engine In C++.

In the case of the Faster R-CNN sample, **maxWorkspaceSize** is set to **10 * (2^20)**, namely 10MB, because there is a need of roughly 6MB of scratch space for the plugin layer for batch size 5.

After the engine is built, the next steps are to serialize the engine, then run the inference with the deserialized engine. For more information, see Serializing A Model In C++.

Running the engine

To deserialize the engine, see Performing Inference In C++.

In **sampleFasterRCNN.cpp**, there are two inputs to the inference function:

- **`data`** is the image input
- **`imInfo`** is the image information array which stores the number of rows, columns, and the scale for each image in a batch.

and four outputs:

- **`bbox_pred`** is the predicted offsets to the heights, widths and center coordinates.
- **`cls_prob`** is the probability associated with each object class of every bounding box.
- **`rois`** is the height, width, and the center coordinates for each bounding box.
- **`count`** is deprecated and can be ignored.

The **`count`** output was used to specify the number of resulting NMS bounding boxes if the output is not aligned to **`nmsMaxOut`**. Although it is deprecated, always allocate the engine buffer of size **`batchSize * sizeof(int)`** for it until it is completely removed from the future version of TensorRT.

Verifying the output

The outputs of the Faster R-CNN network need to be post-processed in order to obtain human interpretable results.

First, because the bounding boxes are now represented by the offsets to the center, height, and width, they need to be unscaled back to the raw image space by dividing the scale defined in the **`imInfo`** (image info).

Ensure you apply the inverse transformation on the bounding boxes and clip the resulting coordinates so that they do not go beyond the image boundaries.

Lastly, overlapped predictions have to be removed by the non-maximum suppression algorithm. The post-processing codes are defined within the CPU because they are neither compute intensive nor memory intensive.

After all of the above work, the bounding boxes are available in terms of the class number, the confidence score (probability), and four coordinates. They are drawn in the output PPM images using the **`writePPMFileWithBBox`** function.

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer

The Activation layer implements element-wise activation functions. Specifically, this sample uses the Activation layer with the type **`kRELU`**.

Convolution layer

The Convolution layer computes a 2D (channel, height, and width) convolution, with or without bias.

FullyConnected layer
The FullyConnected layer implements a matrix-vector product, with or without bias.

Plugin (RPROI) layer
Plugin layers are user-defined and provide the ability to extend the functionalities of TensorRT. See Extending TensorRT With Custom Layers for more details.

Pooling layer
The Pooling layer implements pooling within a channel. Supported pooling types are **maximum**, **average** and **maximum-average blend**.

Shuffle layer
The Shuffle layer implements a reshape and transpose operator for tensors.

SoftMax layer
The SoftMax layer applies the SoftMax function on the input tensor along an input dimension specified by the user.

Prerequisities

1. Download the faster_rcnn_models.tgz dataset.
2. Extract the dataset into the **data/faster-rcnn** directory.

```
cd <TensorRT directory>
wget --no-check-certificate https://dl.dropboxusercontent.com/s/
o6ii098bu51d139/faster_rcnn_models.tgz?dl=0 -O data/faster-rcnn/faster-
rcnn.tgz
tar zxvf data/faster-rcnn/faster-rcnn.tgz -C data/faster-rcnn --strip-
components=1 --exclude=ZF_*
```

Running the sample

1. Compile this sample by running **make** in the **<TensorRT root directory>/samples/sampleFasterRCNN** directory. The binary named **sample_fasterRCNN** will be created in the **<TensorRT root directory>/bin** directory.

```
cd <TensorRT root directory>/samples/sampleFasterRCNN
make
```

   Where **<TensorRT root directory>** is where you installed TensorRT.
2. Run the sample to perform inference.

```
./sample_fasterRCNN
```
3. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following:

```
Sample output
[I] Detected car in 000456.ppm with confidence 99.0063%  (Result stored in
 car-0.990063.ppm).
[I] Detected person in 000456.ppm with confidence 97.4725%  (Result stored
 in person-0.974725.ppm).
[I] Detected cat in 000542.ppm with confidence 99.1191%  (Result stored in
 cat-0.991191.ppm).
[I] Detected dog in 001150.ppm with confidence 99.9603%  (Result stored in
 dog-0.999603.ppm).
```

```
[I] Detected dog in 001763.ppm with confidence 99.7705%  (Result stored in
 dog-0.997705.ppm).
[I] Detected horse in 004545.ppm with confidence 99.467%  (Result stored in
 horse-0.994670.ppm).
&&&& PASSED TensorRT.sample_fasterRCNN # ./build/x86_64-linux/
sample_fasterRCNN
```

This output shows that the sample ran successfully; **PASSED**.

Sample `--help` options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
./sample_fasterRCNN --help
Usage: ./build/x86_64-linux/sample_fasterRCNN
Optional Parameters:
  -h, --help        Display help information.
  --useDLACore=N    Specify the DLA engine to run on.
```

Additional resources

The following resources provide a deeper understanding about object detection with Faster R-CNN:

Faster R-CNN

Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks

Documentation

- Working With TensorRT Using The C++ API
- NVIDIA's TensorRT Documentation Library

Chapter 12. OBJECT DETECTION WITH A TENSORFLOW SSD NETWORK

What Does This Sample Do?

This sample, sampleUffSSD, preprocesses a TensorFlow SSD network, performs inference on the SSD network in TensorRT, using TensorRT plugins to speed up inference.

This sample 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.

The SSD network used in this sample is based on the TensorFlow implementation of SSD, which actually differs from the original paper, in that it has an inception_v2 backbone. For more information about the actual model, download ssd_inception_v2_coco. The TensorFlow SSD network was trained on the InceptionV2 architecture using the MSCOCO dataset which has 91 classes (including the background class). The config details of the network can be found here.

Where Is This Sample Located?

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

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/sampleUffSSD/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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

12.1. README.md

Attention A summary of the `README.md` file is included here for your reference, however, you should always refer to the `README.md` within the package for the most recent documentation updates.

How does this sample work?

The SSD network performs the task of object detection and localization in a single forward pass of the network. The TensorFlow SSD network was trained on the InceptionV2 architecture using the MSCOCO dataset.

The sample makes use of TensorRT plugins to run the SSD network. To use these plugins, the TensorFlow graph needs to be preprocessed, and we use the GraphSurgeon utility to do this.

The main components of this network are the Image Preprocessor, FeatureExtractor, BoxPredictor, GridAnchorGenerator and Postprocessor.

Image Preprocessor The image preprocessor step of the graph is responsible for resizing the image. The image is resized to a 300x300x3 size tensor. This step also performs normalization of the image so all pixel values lie between the range [-1, 1].

FeatureExtractor The FeatureExtractor portion of the graph runs the InceptionV2 network on the preprocessed image. The feature maps generated are used by the anchor generation step to generate default bounding boxes for each feature map.

In this network, the size of feature maps that are used for anchor generation are (19x19), (10x10), (5x5), (3x3), (2x2), (1x1).

BoxPredictor The BoxPredictor step takes in a high level feature map as input and produces a list of box encodings (x-y coordinates) and a list of class scores for each of these encodings per feature map. This information is passed to the postprocessor.

GridAnchorGenerator The goal of this step is to generate a set of default bounding boxes (given the scale and aspect ratios mentioned in the config) for each feature map cell. This is implemented as a plugin layer in TensorRT called the `gridAnchorGenerator` plugin. The registered plugin name is `GridAnchor_TRT`.

Postprocessor The postprocessor step performs the final steps to generate the network output. The bounding box data and confidence scores for all feature maps are fed to the step along with the pre-computed default bounding boxes (generated in the `GridAnchorGenerator` namespace). It then performs NMS (non-maximum suppression) which prunes away most of the bounding boxes based on a confidence threshold and IoU (Intersection over Union) overlap, thus storing only the top `N` boxes

per class. This is implemented as a plugin layer in TensorRT called the NMS plugin. The registered plugin name is **NMS_TRT**.

This sample also implements another plugin called **FlattenConcat** which is used to flatten each input and then concatenate the results. This is applied to the location and confidence data before it is fed to the post processor step since the NMS plugin requires the data to be in this format.

For details on how a plugin is implemented, see the implementation of **FlattenConcat** plugin and **FlattenConcatPluginCreator** in the **sampleUffSSD.cpp** file in the **tensorrt/samples/sampleUffSSD** directory.

Processing the input graph

The TensorFlow SSD graph has some operations that are currently not supported in TensorRT. Using a preprocessor on the graph, we can combine multiple operations in the graph into a single custom operation which can be implemented as a plugin layer in TensorRT. Currently, the preprocessor provides the ability to stitch all nodes within a namespace into one custom node.

To use the preprocessor, the **convert-to-uff** utility should be called with a **-p** flag and a config file. The config script should also include attributes for all custom plugins which will be embedded in the generated **.uff** file. Current sample script for SSD is located in **/usr/src/tensorrt/samples/sampleUffSSD/config.py**.

Using the preprocessor on the graph, we were able to remove the **Preprocessor** namespace from the graph, stitch the **GridAnchorGenerator** namespace together to create the **GridAnchorGenerator** plugin, stitch the **postprocessor** namespace together to get the NMS plugin and mark the concat operations in the BoxPredictor as **FlattenConcat** plugins.

The TensorFlow graph has some operations like **Assert** and **Identity** which can be removed for the inferencing. Operations like **Assert** are removed and leftover nodes (with no outputs once assert is deleted) are then recursively removed.

Identity operations are deleted and the input is forwarded to all the connected outputs. Additional documentation on the graph preprocessor can be found in the TensorRT API.

Preparing the data

The generated network has an input node called **Input**, and the output node is given the name **MarkOutput_0** by the UFF converter. These nodes are registered by the UFF Parser in the sample.

```
parser->registerInput("Input", DimsCHW(3, 300, 300),
UffInputOrder::kNCHW);
parser->registerOutput("MarkOutput_0");
```

The input to the SSD network in this sample is 3 channel 300x300 images. In the sample, we normalize the image so the pixel values lie in the range [-1,1]. This is equivalent to the image preprocessing stage of the network.

Since TensorRT does not depend on any computer vision libraries, the images are represented in binary **R**, **G**, and **B** values for each pixel. The format is Portable PixMap (PPM), which is a netpbm color image format. In this format, the **R**, **G**, and **B** values for each pixel are represented by a byte of integer (0-255) and they are stored together, pixel by pixel.

There is a simple PPM reading function called **readPPMFile**.

sampleUffSSD plugins

Details about how to create TensorRT plugins can be found in Extending TensorRT With Custom Layers.

The **config.py** defined for the **convert-to-uff** command should have the custom layers mapped to the plugin names in TensorRT by modifying the **op** field. The names of the plugin parameters should also exactly match those expected by the TensorRT plugins. For example, for the **GridAnchor** plugin, the **config.py** should have the following:

```
PriorBox = gs.create_plugin_node(name="GridAnchor",
op="GridAnchor_TRT",
 numLayers=6,
 minSize=0.2,
 maxSize=0.95,
 aspectRatios=[1.0, 2.0, 0.5, 3.0, 0.33],
 variance=[0.1,0.1,0.2,0.2],
 featureMapShapes=[19, 10, 5, 3, 2, 1])
```

Here, **GridAnchor_TRT** matches the registered plugin name and the parameters have the same name and type as expected by the plugin.

If the **config.py** is defined as above, the NvUffParser will be able to parse the network and call the appropriate plugins with the correct parameters.

Details about some of the plugin layers implemented for SSD in TensorRT are given below.

GridAnchorGeneration plugin

This plugin layer implements the grid anchor generation step in the TensorFlow SSD network. For each feature map we calculate the bounding boxes for each grid cell. In this network, there are 6 feature maps and the number of boxes per grid cell are as follows:

- [19x19] feature map: 3 boxes (19x19x3x4(co-ordinates/box))
- [10x10] feature map: 6 boxes (10x10x6x4)
- [5x5] feature map: 6 boxes (5x5x6x4)
- [3x3] feature map: 6 boxes (3x3x6x4)

- [2x2] feature map: 6 boxes (2x2x6x4)
- [1x1] feature map: 6 boxes (1x1x6x4)

`NMS` plugin

The **`NMS`** plugin generates the detection output based on location and confidence predictions generated by the BoxPredictor. This layer has three input tensors corresponding to location data (**`locData`**), confidence data (**`confData`**) and priorbox data (**`priorData`**).

The inputs to detection output plugin have to be flattened and concatenated across all the feature maps. We use the **`FlattenConcat`** plugin implemented in the sample to achieve this. The location data generated from the box predictor has the following dimensions:

```
19x19x12 -> Reshape -> 1083x4 -> Flatten -> 4332x1
10x10x24 -> Reshape -> 600x4 -> Flatten -> 2400x1
```

and so on for the remaining feature maps.

After concatenating, the input dimensions for **`locData`** input are of the order of 7668x1.

The confidence data generated from the box predictor has the following dimensions:

```
19x19x273 -> Reshape -> 1083x91 -> Flatten -> 98553x1
10x10x546 -> Reshape -> 600x91 -> Flatten -> 54600x1
```

and so on for the remaining feature maps.

After concatenating, the input dimensions for **`confData`** input are 174447x1.

The prior data generated from the grid anchor generator plugin has 6 outputs and their dimensions are as follows:

```
Output 1 corresponds to the 19x19 feature map and has dimensions 2x4332x1
Output 2 corresponds to the 10x10 feature map and has dimensions 2x2400x1
```

and so on for the other feature maps.

There are two channels in the outputs because one channel is used to store variance of each coordinate that is used in the NMS step. After concatenating, the input dimensions for **`priorData`** input are of the order of 2x7668x1.

```
struct DetectionOutputParameters
{
  bool shareLocation, varianceEncodedInTarget;
  int backgroundLabelId, numClasses, topK, keepTopK;
  float confidenceThreshold, nmsThreshold;
  CodeTypeSSD codeType;
  int inputOrder[3];
  bool confSigmoid;
  bool isNormalized;
};
```

shareLocation and **varianceEncodedInTarget** are used for the Caffe SSD network implementation, so for the TensorFlow network they should be set to **true** and **false** respectively. The **confSigmoid** and **isNormalized** parameters are necessary for the TensorFlow implementation. If **confSigmoid** is set to **true**, it calculates the sigmoid values of all the confidence scores. The **isNormalized** flag specifies if the data is normalized and is set to **true** for the TensorFlow graph.

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer
The Activation layer implements element-wise activation functions. Specifically, this sample uses the Activation layer with the type **kRELU**.

Concatenation layer
The Concatenation layer links together multiple tensors of the same non-channel sizes along the channel dimension.

Convolution layer
The Convolution layer computes a 2D (channel, height, and width) convolution, with or without bias.

Padding layer
The Padding layer implements spatial zero-padding of tensors along the two innermost dimensions.

Plugin layer
Plugin layers are user-defined and provide the ability to extend the functionalities of TensorRT. See Extending TensorRT With Custom Layers for more details.

Pooling layer
The Pooling layer implements pooling within a channel. Supported pooling types are **maximum**, **average** and **maximum-average blend**.

Scale layer
The Scale layer implements a per-tensor, per-channel, or per-element affine transformation and/or exponentiation by constant values.

Shuffle layer
The Shuffle layer implements a reshape and transpose operator for tensors.

Prerequisites

1. Install the UFF toolkit and graph surgeon; depending on your TensorRT installation method, to install the toolkit and graph surgeon, choose the method you used to install TensorRT for instructions (see TensorRT Installation Guide: Installing TensorRT).
2. Download the ssd_inception_v2_coco TensorFlow trained model.
3. Perform preprocessing on the TensorFlow model using the UFF converter.

a. Copy the TensorFlow protobuf file (**frozen_inference_graph.pb**) from the downloaded directory in the previous step to the working directory (for example **/usr/src/tensorrt/samples/sampleUffSSD/**).
b. Run the following command for the conversion. **convert-to-uff frozen_inference_graph.pb -O NMS -p config.py**

   This saves the converted **.uff** file in the same directory as the input with the name **frozen_inference_graph.pb.uff**.

   The **config.py** script specifies the preprocessing operations necessary for the SSD TensorFlow graph. The plugin nodes and plugin parameters used in the **config.py** script should match the registered plugins in TensorRT.
c. Copy the converted **.uff** file to the data directory and rename it to **sample_ssd_relu6.uff <TensorRT Install>/data/ssd/sample_ssd_relu6.uff**.

4. The sample also requires a **labels.txt** file with a list of all labels used to train the model. The labels file for this network is **<TensorRT Install>/data/ssd/ssd_coco_labels.txt**.

Running the sample

1. Compile this sample by running **make** in the **<TensorRT root directory>/samples/sampleUffSSD** directory. The binary named **sample_uff_ssd** will be created in the **<TensorRT root directory>/bin** directory.

```
cd <TensorRT root directory>/samples/sampleUffSSD
make
```

   Where **<TensorRT root directory>** is where you installed TensorRT.
2. Run the sample to perform object detection and localization.

   To run the sample in FP32 mode: **./sample_uff_ssd**

   To run the sample in INT8 mode: **./sample_uff_ssd --int8**

To run the network in INT8 mode, refer to **BatchStreamPPM.h** for details on how calibration can be performed. Currently, we require a file called **list.txt**, with a list of all PPM images for calibration in the **<TensorRT Install>/data/ssd/** folder. The PPM images to be used for calibration can also reside in the same folder.

3. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following:

```
&&&& RUNNING TensorRT.sample_uff_ssd # ./build/x86_64-linux/sample_uff_ssd
[I] ../data/samples/ssd/sample_ssd_relu6.uff
[I] Begin parsing model...
[I] End parsing model...
[I] Begin building engine...
```

```
I] Num batches 1
[I] Data Size 270000
[I] *** deserializing
[I] Time taken for inference is 4.24733 ms.
[I] KeepCount 100
[I] Detected dog in the image 0 (../../data/samples/ssd/dog.ppm) with
 confidence 89.001 and coordinates (81.7568,23.1155),(295.041,298.62).
[I] Result stored in dog-0.890010.ppm.
[I] Detected dog in the image 0 (../../data/samples/ssd/dog.ppm) with
 confidence 88.0681 and coordinates (1.39267,0),(118.431,237.262).
[I] Result stored in dog-0.880681.ppm.
&&&& PASSED TensorRT.sample_uff_ssd # ./build/x86_64-linux/sample_uff_ssd
```

This output shows that the sample ran successfully; **PASSED**.

Additional resources

The following resources provide a deeper understanding about the TensorFlow SSD network structure:

Models

TensorFlow detection model zoo

Network

ssd_inception_v2_coco_2017_11_17

Dataset

MSCOCO dataset

Documentation

- Working With TensorRT Using The C++ API
- NVIDIA's TensorRT Documentation Library

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.

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/sampleMovieLens/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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

13.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

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.

Importing a network to TensorRT

The network is converted from Tensorflow using the UFF converter (see Converting A Frozen Graph To UFF), and imported using the UFF parser. Constant layers are used to represent the trained parameters within the network, and the MLPs are implemented using MatrixMultiply layers. A TopK operation is added manually after parsing to find the highest rated movie for the given user.

Running inference

The sample fills the input buffer with **`userIDs`** and their corresponding lists of **`MovieIDs`**, which are loaded from **`movielens_ratings.txt`**. Then, it launches the inference to predict the rating probabilities for the movies using TensorRT.

Verifying the output

Finally, the sample compares the outputs predicted by TensorRT with the expected outputs which are given by **`movielens_ratings.txt`**. For each user, the **`MovieID`** with the highest probability should match the expected highest-rated **`MovieID`**. In the verbose mode, the sample also prints out the probability, which should be close to the expected probability.

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer
The Activation layer implements element-wise activation functions.

MatrixMultiply layer
The MatrixMultiply layer implements matrix multiplication for a collection of matrices.

Scale layer
The Scale layer implements a per-tensor, per-channel, or per-element affine transformation and/or exponentiation by constant values.

Shuffle layer
The Shuffle layer implements a reshape and transpose operator for tensors.

TopK layer
The TopK layer finds the top **K** maximum (or minimum) elements along a dimension, returning a reduced tensor and a tensor of index positions.

Training an NCF network

This sample comes with a pre-trained model. However, if you want to train your own model, you would need to also convert the model weights to UFF format before you can run the sample.

1. Clone the NCF repository.

```
git clone https://github.com/hexiangnan/neural_collaborative_filtering.git
cd neural_collaborative_filtering
git checkout 0cd2681598507f1cc26d110083327069963f4433
```

2. Apply the **sampleMovieLensTraining.patch** file to save the final result.

```
patch -l -p1 < <TensorRT Install>/samples/sampleMovieLens/
sampleMovieLensTraining.patch
```

3. Install Python 3.
4. Train the MLP based NCF network.

```
python3 MLP.py --dataset ml-1m --epochs 20 --batch_size 256 --layers
 [64,32,16,8] --reg_layers [0,0,0,0] --num_neg 4 --lr 0.001 --learner adam
 --verbose 1 --out 1
```

Using **0** for **reg_layers** will cause undefined behavior when training the network.

This step produces the following files in the root directory of the Git repo:

movielens_ratings.txt
A text file which contains the lists of **MovieIDs** for each user and the 10 highest-rated **MovieIDs** with their probabilities.

sampleMovieLens.pb
The frozen TensorFlow graph which contains the information of the network structure and parameters.

5. Convert the trained model weights to UFF format which sampleMovieLens understands.

a. Install UFF. The **convert_to_uff.py** utility is located in the **/usr/local/bin/convert-to-uff** directory. This utility is installed with the **UFF.whl** file that is shipped with TensorRT.
b. Convert the **frozen .pb** file to **.uff** format.

```
python3 convert_to_uff.py sampleMovieLens.pb -p preprocess.py
```

The **preprocess.py** script is a preprocessing step that needs to be applied to the TensorFlow graph before it can be used by TensorRT. The reason for this is that TensorFlow's concatenation operation accounts for the batch dimension while TensorRT's concatenation operation does not.
c. Copy:

- The **sampleMovieLens.uff** file to the **<TensorRT Install>/data/movielens** directory.
- The **movielens_ratings.txt** file to the **<TensorRT Install>/data/movielens** directory.

Running the sample

1. Compile this sample by running **make** in the **<TensorRT root directory>/samples/sampleMovieLens** directory. The binary named **sample_movielens** will be created in the **<TensorRT root directory>/bin** directory.

```
cd <TensorRT root directory>/samples/sampleMovieLens
make
```

Where **<TensorRT root directory>** is where you installed TensorRT.
2. Run the sample to predict the highest-rated movie for each user.

```
cd <TensorRT Install>/bin
./sample_movielens # Run with default batch=32 i.e. num of users
./sample_movielens -b <N> # Run with batch=N i.e. num of users
./sample_movielens --verbose # Prints out inputs, outputs, expected outputs,
 and expected vs predicted probabilities
```

3. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following:

```
&&&& RUNNING TensorRT.sample_movielens # ./sample_movielens -b 5
 [I] data/movielens/movielens_ratings.txt
 [I] Begin parsing model...
 [I] End parsing model...
 [I] End building engine...
 [I] Done execution. Duration : 514.272 microseconds.
 [I] Num of users : 5
 [I] Num of Movies : 100
 [I] | User : 0 | Expected Item : 128 | Predicted Item : 128 |
 [I] | User : 1 | Expected Item : 133 | Predicted Item : 133 |
 [I] | User : 2 | Expected Item : 515 | Predicted Item : 515 |
 [I] | User : 3 | Expected Item : 23 | Predicted Item : 23 |
 [I] | User : 4 | Expected Item : 134 | Predicted Item : 134 |
 &&&& PASSED TensorRT.sample_movielens # ./sample_movielens -b 5
```

This output shows that the sample ran successfully; **PASSED**.

Sample `--help` options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
Usage: ./sample_movielens [-h] [-b NUM_USERS] [--useDLACore=<int>] [--verbose]

-h Display help information. All single dash options enable perf mode.

-b Number of Users i.e. Batch Size (default numUsers=32).

--useDLACore=N Specify a DLA engine for layers that support DLA. Value can range
 from 0 to n-1, where n is the number of DLA engines on the platform.

--verbose Enable verbose prints.

--fp16 Run in FP16 mode.

--strict Run with strict type constraints.
```

Additional resources

The following resources provide a deeper understanding about sampleMovieLens:

MovieLens

- MovieLens dataset
- Neural Collaborative Filtering Paper

Models

Neural Collaborative Filtering GitHub Repo

Blogs

Accelerating Recommendation System Inference Performance with TensorRT

Videos

SampleMovieLens YouTube Tutorial

Documentation

- Working With TensorRT Using The C++ API
- Jupyter Notebook Tutorial for SampleMovieLens
- NVIDIA's TensorRT Documentation Library

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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

14.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

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.

Importing a network to TensorRT

The network is converted from Tensorflow using the UFF converter (see Converting A Frozen Graph To UFF), and imported using the UFF parser. Constant layers are used to represent the trained parameters within the network, and the MLPs are implemented using MatrixMultiply layers. A TopK operation is added manually after parsing to find the highest rated movie for the given user.

Running inference

The sample fills the input buffer with **`userIDs`** and their corresponding lists of **`MovieIDs`**, which are loaded from **`movielens_ratings.txt`**. Then, it launches the inference to predict the rating probabilities for the movies using TensorRT. The inference will be launched on multiple processes. When MPS is enabled, the processes will share one single CUDA context to reduce context overhead. See Multi-Process Service Introduction for more details about MPS.

Verifying the output

Finally, the sample compares the outputs predicted by TensorRT with the expected outputs which are given by **`movielens_ratings.txt`**. For each user, the **`MovieID`** with the highest probability should match the expected highest-rated **`MovieID`**. In the verbose mode, the sample also prints out the probability, which should be close to the expected probability.

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer
The Activation layer implements element-wise activation functions.

MatrixMultiply layer
The MatrixMultiply layer implements matrix multiplication for a collection of matrices.

Scale layer
The Scale layer implements a per-tensor, per-channel, or per-element affine transformation and/or exponentiation by constant values.

Shuffle layer
The Shuffle layer implements a reshape and transpose operator for tensors.

TopK layer
The TopK layer finds the top **K** maximum (or minimum) elements along a dimension, returning a reduced tensor and a tensor of index positions.

Training an NCF network

This sample comes with a pre-trained model. However, if you want to train your own model, you would need to also convert the model weights to UFF format before you can run the sample. For step-by-step instructions, refer to the **README.md** file in the **sampleMovieLens** directory.

Running the sample

1. Compile this sample by running **make** in the **<TensorRT root directory>/samples/sampleMovieLensMPS** directory. The binary named **sample_movielens_mps** will be created in the **<TensorRT root directory>/bin** directory.

   ```
   cd <TensorRT root directory>/samples/sampleMovieLensMPS
   make
   ```

   Where **<TensorRT root directory>** is where you installed TensorRT.
2. Set-up an MPS server.

   ```
   export CUDA_VISIBLE_DEVICES=<GPU_ID>
   nvidia-smi -i <GPU_ID> -c EXCLUSIVE_PROCESSexport CUDA_VISIBLE_DEVICES=0
   export CUDA_MPS_PIPE_DIRECTORY=/tmp/nvidia-mps # Select a location that's
    accessible to the given $UID
   export CUDA_MPS_LOG_DIRECTORY=/tmp/nvidia-log # Select a location that's
    accessible to the given $UID
   nvidia-cuda-mps-control -d # Start the daemon.
   ```

   The log files for MPS are located at:**$CUDA_MPS_LOG_DIRECTORY/control.log** **$CUDA_MPS_LOG_DIRECTORY/server.log**

3. Set-up an MPS client. Set the following variables in the client process environment. The **CUDA_VISIBLE_DEVICES** variable should not be set in the client's environment.

```
export CUDA_MPS_PIPE_DIRECTORY=/tmp/nvidia-mps # Set to the same location as
 the MPS control daemon
export CUDA_MPS_LOG_DIRECTORY=/tmp/nvidia-log # Set to the same location as
 the MPS control daemon
```

4. Run the sample from an MPS client to predict the highest-rated movie for each user on multiple processes.

```
cd <TensorRT Install>/bin
./sample_movielens_mps (default batch=32 i.e. num of users, Number of
 processes=1)
./sample_movielens_mps -b <bSize> -p <nbProc> (bSize=Batch size i.e. num of
 users, nbProc=Number of processes)
./sample_movielens_mps --verbose (prints inputs, groundtruth values,
 expected vs predicted probabilities)
```

5. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following:

```
&&&& RUNNING TensorRT.sample_movielens_mps # build/cuda-  10.0/7.3/x86_64/
sample_movielens_mps -b 2 -p 2
[I] data/samples/movielens/movielens_ratings.txt
[I] Begin parsing model...
[I] End parsing model...
[I] End building engine...
[I] Done execution in process: 24136 . Duration : 214.272 microseconds.
[I] Num of users : 2
[I] Num of Movies : 100
[I] | PID : 24136 | User : 0 | Expected Item : 128 | Predicted Item : 128 |
[I] | PID : 24136 | User : 1 | Expected Item : 133 | Predicted Item : 133 |
[I] Done execution in process: 24135 . Duration : 214.176 microseconds.
[I] Num of users : 2
[I] Num of Movies : 100
[I] | PID : 24135 | User : 0 | Expected Item : 128 | Predicted Item : 128 |
[I] | PID : 24135 | User : 1 | Expected Item : 133 | Predicted Item : 133 |
[I] Number of processes executed: 2. Number of processes failed: 0.
[I] Total MPS Run Duration: 1737.51 milliseconds.
&&&& PASSED TensorRT.sample_movielens_mps # build/cuda- 10.0/7.3/x86_64/
sample_movielens_mps -b 2 -p 2
```

   This output shows that the sample ran successfully; **PASSED**. The output also shows that the predicted items for each user matches the expected items and the duration of the execution. Finally, the sample prints out the PIDs of the processes, showing that the inference is launched on multiple processes.

6. To restore the system to its original state, shutdown MPS, if needed. **echo quit | nvidia-cuda-mps-control**

Sample `--help` options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
Usage:
./sample_movielens_mps [-h] [-b NUM_USERS] [-p NUM_PROCESSES] [--
useDLACore=<int>] [--verbose]
-h Display help information. All single dash options enable perf mode.
```

```
-b Number of Users i.e. Batch Size (default numUsers=32).
-p Number of child processes to launch (default nbProcesses=1. Using MPS with
 this option is strongly recommended).
--useDLACore=N Specify a DLA engine for layers that support DLA. Value can range
 from 0 to n-1, where n is the number of DLA engines on the platform.
--verbose Enable verbose prints.
--int8 Run in Int8 mode.
--fp16 Run in FP16 mode.
```

Additional resources

The following resources provide a deeper understanding about sampleMovieLensMPS:

MovieLensMPS

- MovieLens dataset
- Neural Collaborative Filtering Paper
- Multi-Process Service Introduction

Models

Neural Collaborative Filtering GitHub Repo

Documentation

- Working With TensorRT Using The C++ API
- Jupyter Notebook Tutorial for SampleMovieLens
- NVIDIA's TensorRT Documentation Library

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.

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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

15.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

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.

Processing the input

The input to the SSD network in this sample is a RGB 300x300 image. The image format is Portable PixMap (PPM), which is a netpbm color image format. In this format, the **`R`**, **`G`**, and **`B`** values for each pixel are represented by a byte of integer (0-255) and they are stored together, pixel by pixel.

The authors of SSD have trained the network such that the first Convolution layer sees the image data in **`B`**, **`G`**, and **`R`** order. Therefore, the channel order needs to be changed when the PPM image is being put into the network's input buffer.

```
float pixelMean[3]{ 104.0f, 117.0f, 123.0f }; // also in BGR order
float* data = new float[N * kINPUT_C * kINPUT_H * kINPUT_W];
  for (int i = 0, volImg = kINPUT_C * kINPUT_H * kINPUT_W; i < N; ++i)
 {
     for (int c = 0; c < kINPUT_C; ++c)
     {
      // the color image to input should be in BGR order
      for (unsigned j = 0, volChl = kINPUT_H * kINPUT_W; j < volChl; ++j){
      Data[i * volImg + c * volChl + j] = float(ppms[i].buffer[j * kINPUT_C + 2
 - c]) - pixelMean[c];
      }
     }
 }
```

The **`readPPMFile`** and **`writePPMFileWithBBox`** functions read a PPM image and produce output images with red colored bounding boxes respectively.

The **`readPPMFile`** function will not work correctly if the header of the PPM image contains any annotations starting with **`#`**.

Defining the network

The network is defined in a prototxt file which is shipped with the sample and located in the **data/ssd** directory. The original prototxt file provided by the authors is modified and included in the TensorRT in-built plugin layers in the prototxt file.

The built-in plugin layers used in sampleSSD are Normalize, PriorBox, and DetectionOutput. The corresponding registered plugins for these layers are **Normalize_TRT**, **PriorBox_TRT** and **NMS_TRT**.

To initialize and register these TensorRT plugins to the plugin registry, the **initLibNvInferPlugins** method is used. After registering the plugins and while parsing the prototxt file, the NvCaffeParser creates plugins for the layers based on the parameters that were provided in the prototxt file automatically. The details about each parameter is provided in the **README.md** and can be modified similar to the Caffe Layer parameter.

Building the engine

The sampleSSD sample builds a network based on a Caffe model and network description. For details on importing a Caffe model, see Importing A Caffe Model Using The C++ Parser API. The SSD network has few non-natively supported layers which are implemented as plugins in TensorRT. The Caffe parser can create plugins for these layers internally using the plugin registry.

This sample can run in FP16 and INT8 modes based on the user input. For more details, see INT8 Calibration Using C++ and Enabling FP16 Inference Using C++. The sample selects the entropy calibrator as a default choice. The **CalibrationMode** parameter in the sample code needs to be set to **0** to switch to the Legacy calibrator.

For details on how to build the TensorRT engine, see Building An Engine In C++. After the engine is built, the next steps are to serialize the engine and run the inference with the deserialized engine. For more information about these steps, see Serializing A Model In C++.

Verifying the output

After deserializing the engine, you can perform inference. To perform inference, see Performing Inference In C++.

In sampleSSD, there is a single input: - **data**, namely the image input

And 2 outputs:

detectionOut
The detection array, containing the image ID, label, confidence, 4 coordinates.

keepCount
The number of valid detections.

The outputs of the SSD network are directly human interpretable. The results are organized as tuples of 7. In each tuple, the 7 elements are:

- image ID
- object label
- confidence score
- (x,y) coordinates of the lower left corner of the bounding box
- (x,y) coordinates of the upper right corner of the bounding box

This information can be drawn in the output PPM image using the **`writePPMFileWithBBox`** function. The **`kVISUAL_THRESHOLD`** parameter can be used to control the visualization of objects in the image. It is currently set to 0.6, therefore, the output will display all objects with confidence score of 60% and above.

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer
The Activation layer implements element-wise activation functions. Specifically, this sample uses the Activation layer with the type **`kRELU`**.

Concatenation layer
The Concatenation layer links together multiple tensors of the same non-channel sizes along the channel dimension.

Convolution layer
The Convolution layer computes a 2D (channel, height, and width) convolution, with or without bias.

Plugin layer
Plugin layers are user-defined and provide the ability to extend the functionalities of TensorRT. See Extending TensorRT With Custom Layers for more details.

Pooling layer
The Pooling layer implements pooling within a channel. Supported pooling types are **`maximum`**, **`average`** and **`maximum-average blend`**.

Shuffle layer
The Shuffle layer implements a reshape and transpose operator for tensors.

SoftMax layer
The SoftMax layer applies the SoftMax function on the input tensor along an input dimension specified by the user.

TensorRT plugin layers in SSD

sampleSSD has three plugin layers; Normalize, PriorBox and DetectionOutput. The details about each layer and its parameters is shown below in **`caffe.proto`** format.

```
message LayerParameter {
  optional DetectionOutputParameter detection_output_param = 881;
```

```
  optional NormalizeParameter norm_param = 882;
  optional PriorBoxParameter prior_box_param ==883;
}

// Message that stores parameters used by Normalize layer
NormalizeParameter {
  optional bool across_spatial = 1 [default = true];
  // Initial value of scale. Default is 1.0
  optional FillerParameter scale_filler = 2;
  // Whether or not scale parameters are shared across channels.
  optional bool channel_shared = 3 [default = true];
  // Epsilon for not dividing by zero while normalizing variance
  optional float eps = 4 [default = 1e-10];
}

// Message that stores parameters used by PriorBoxLayer
message PriorBoxParameter {
  // Encode/decode type.
  enum CodeType {
 CORNER = 1;
 CENTER_SIZE = 2;
 CORNER_SIZE = 3;
  }
  // Minimum box size (in pixels). Required!
  repeated float min_size = 1;
  // Maximum box size (in pixels). Required!
  repeated float max_size = 2;
  // Various aspect ratios. Duplicate ratios will be ignored.
  // If none is provided, we use default ratio 1.
  repeated float aspect_ratio = 3;
  // If true, will flip each aspect ratio.
  // For example, if there is aspect ratio "r",
  // we will generate aspect ratio "1.0/r" as well.
  optional bool flip = 4 [default = true];
  // If true, will clip the prior so that it is within [0, 1]
  optional bool clip = 5 [default = false];
  // Variance for adjusting the prior bboxes.
  repeated float variance = 6;
  // By default, we calculate img_height, img_width, step_x, step_y based on
  // bottom[0] (feat) and bottom[1] (img). Unless these values are explicitly
  // provided.
  // Explicitly provide the img_size.
  optional uint32 img_size = 7;
  // Either img_size or img_h/img_w should be specified; not both.
  optional uint32 img_h = 8;
  optional uint32 img_w = 9;

  // Explicitly provide the step size.
  optional float step = 10;
  // Either step or step_h/step_w should be specified; not both.
  optional float step_h = 11;
  optional float step_w = 12;

  // Offset to the top left corner of each cell.
  optional float offset = 13 [default = 0.5];
}

message NonMaximumSuppressionParameter {
  // Threshold to be used in NMS.
  optional float nms_threshold = 1 [default = 0.3];
  // Maximum number of results to be kept.
  optional int32 top_k = 2;
  // Parameter for adaptive NMS.
  optional float eta = 3 [default = 1.0];
}
```

```
// Message that stores parameters used by DetectionOutputLayer
message DetectionOutputParameter {
  // Number of classes to be predicted. Required!
  optional uint32 num_classes = 1;
  // If true, bounding box are shared among different classes.
  optional bool share_location = 2 [default = true];
  // Background label id. If there is no background class,
  // set it as -1.
  optional int32 background_label_id = 3 [default = 0];
  // Parameters used for NMS.
  optional NonMaximumSuppressionParameter nms_param = 4;

  // Type of coding method for bbox.
  optional PriorBoxParameter.CodeType code_type = 5 [default = CORNER];
  // If true, variance is encoded in target; otherwise we need to adjust the
  // predicted offset accordingly.
  optional bool variance_encoded_in_target = 6 [default = false];
  // Number of total bboxes to be kept per image after nms step.
  // -1 means keeping all bboxes after nms step.
  optional int32 keep_top_k = 7 [default = -1];
  // Only consider detections whose confidences are larger than a threshold.
  // If not provided, consider all boxes.
  optional float confidence_threshold = 8;
  // If true, visualize the detection results.
  optional bool visualize = 9 [default = false];
  // The threshold used to visualize the detection results.
}
```

Prerequisites

1. Download models_VGGNet_VOC0712_SSD_300x300.tar.gz.
2. Extract the contents. **`tar xvf models_VGGNet_VOC0712_SSD_300x300.tar.gz`**
   a. Generate MD5 hash and compare against the reference below: **`md5sum models_VGGNet_VOC0712_SSD_300x300.tar.gz`**

      If the model is correct, you should see the following MD5 hash output: **`9a795fc161fff2e8f3aed07f4d488faf models_VGGNet_VOC0712_SSD_300x300.tar.gz`**
   b. Edit the **`deploy.prototxt`** file and change all the Flatten layers to Reshape operations with the following parameters:

      ```
      reshape_param {
       shape {
        dim: 0
        dim: -1
        dim: 1
        dim: 1
       }
      }
      ```
   c. Update the **`detection_out`** layer to add the **`keep_count`** output as expected by the TensorRT DetectionOutput Plugin. **`top: "keep_count"`**
   d. Rename the updated **`deploy.prototxt`** file to **`ssd.prototxt`** and move the file to the **`data`** directory. **`mv ssd.prototxt <TensorRT_Install_Directory>/data/ssd`**
   e. Move the **`caffemodel`** file to the **`data`** directory.

```
mv VGG_VOC0712_SSD_300x300_iter_120000.caffemodel
 <TensorRT_Install_Directory>/data/ssd
```

3. Generate the INT8 calibration batches.

   a. Install Pillow.

   - For Python 2 users, run: **`python2 -m pip install Pillow`**
   - For Python 3 users, run: **`python3 -m pip install Pillow`**

   b. Generate the INT8 batches. **`prepareINT8CalibrationBatches.sh`**

   The script selects 500 random JPEG images from the PASCAL VOC dataset and converts them to PPM images. These 500 PPM images are used to generate INT8 calibration batches.

Do not move the batch files from the **`<TensorRT_Install_Directory>/data/ssd/batches`** directory.

If you want to use a different dataset to generate INT8 batches, use the **`batchPrepare.py`** script and place the batch files in the **`<TensorRT_Install_Directory>/data/ssd/batches`** directory.

Running the sample

1. Compile this sample by running **`make`** in the **`<TensorRT root directory>/samples/sampleSSD`** directory. The binary named **`sample_ssd`** will be created in the **`<TensorRT root directory>/bin`** directory.

```
cd <TensorRT root directory>/samples/sampleSSD
make
```

   Where **`<TensorRT root directory>`** is where you installed TensorRT.

2. Run the sample to perform inference on the digit:

```
./sample_ssd [-h] [--fp16] [--int8]
```

3. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following:

```
&&&& RUNNING TensorRT.sample_ssd # ./sample_ssd
[I] Begin parsing model...
[I] FP32 mode running...
[I] End parsing model...
[I] Begin building engine...
[I] [TRT] Detected 1 input and 2 output network tensors.
[I] End building engine...
[I] *** deserializing
[I] Image name:../data/samples/ssd/bus.ppm, Label: car, confidence: 96.0587
 xmin: 4.14486 ymin: 117.443 xmax: 244.102 ymax: 241.829
&&&& PASSED TensorRT.sample_ssd # ./build/x86_64-linux/sample_ssd
```

   This output shows that the sample ran successfully; **`PASSED`**.

Sample --help options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
Usage: ./build/x86_64-linux/sample_ssd
Optional Parameters:
-h, --help Display help information.
--useDLACore=N Specify the DLA engine to run on.
--fp16 Specify to run in fp16 mode.
--int8 Specify to run in int8 mode.
```

Additional resources

The following resources provide a deeper understanding about how the SSD model works:

Models

SSD: Single Shot MultiBox Detector

Dataset

PASCAL VOC 2007+ 2012

Documentation

- Working With TensorRT Using The C++ API
- NVIDIA's TensorRT Documentation Library

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.

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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

16.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

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.

Defining the network

This sample follows the same flow as sampleMNISTAPI with one exception. The network is defined as a sequence of addMLP calls, which adds FullyConnected and Activation layers to the network.

Generally, an MLP layer is:

- a FullyConnected operation that is followed by an optional Scale and an optional Activation; or
- a MatrixMultiplication operation followed by an optional bias and an optional activation.

An MLP network is more than one MLP layer generated sequentially in the TensorRT network. The optimizer will detect this pattern and generate optimized MLP code.

More formally, the following variations of MLP layers will trigger the MLP optimizer:

```
{MatrixMultiplication [-> ElementWiseSum] [-> Activation]}+
{FullyConnected [-> Scale(with empty scale and power arguments)] [->
 Activation]}+
```

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer
The Activation layer implements element-wise activation functions. Specifically, this sample uses the Activation layer with the type **`kRELU`**.

FullyConnected layer
The FullyConnected layer implements a matrix-vector product, with or without bias.

TopK layer
The TopK layer finds the top K maximum (or minimum) elements along a dimension, returning a reduced tensor and a tensor of index positions.

Training an MLP network

This sample comes with pre-trained weights. However, if you want to train your own MLP network, you first need to generate the weights by training a TensorFlow based neural network using an MLP optimizer, and then verify that the trained weights are converted into a format that sampleMLP can read. If you want to use the weights that are shipped with this sample, see *Running the sample*.

1. Install TensorFlow 1.4 or later.
2. Download the TensorFlow tutorial.

```
git clone https://github.com/aymericdamien/TensorFlow-Examples.git
 cd TensorFlow-Examples
```

3. Install Python.
4. Apply the **update_mlp.patch** file to save the final result.

```
patch -p1 < <TensorRT Install>/samples/sampleMLP/update_mlp.patch
```

5. Train the MNIST MLP network.

```
python examples/3_NeuralNetworks/multilayer_perceptron.py
```

This step produces the following file:

```
/tmp/sampleMLP.ckpt - Trained MLP checkpoint
```

The **sampleMLP.ckpt** file contains the checkpoint for the parameters and weights.

6. Convert the trained model weights to a format sampleMLP understands.

```
python <TensorRT Install>/samples/sampleMLP/convert_weights.py -m /tmp/
sampleMLP.ckpt -o sampleMLP
 mkdir -p <TensorRT Install>/data/mlp
 cp sampleMLP.wts2 <TensorRT Install>/data/mlp/
```

Running the sample

1. Compile this sample by running **make** in the **<TensorRT root directory>/samples/sampleMLP** directory. The binary named **sample_mlp** will be created in the **<TensorRT root directory>/bin** directory.

```
cd <TensorRT root directory>/samples/sampleMLP
make
```

Where **<TensorRT root directory>** is where you installed TensorRT.

2. Run the sample to classify the MNIST digit.

```
cd <TensorRT Install>/bin
make
```

3. erify that the sample ran successfully. If the sample runs successfully you should see output similar to the following; ASCII rendering of the input image with digit 9:

```
&&&& RUNNING TensorRT.sample_mlp # build/x86_64-linux/sample_mlp
 [I] Input:
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@%.-@@@@@@@@@@@@
 @@@@@@@@@@@*-    %@@@@@@@@@@@
 @@@@@@@@@@= .-.  *@@@@@@@@@@@
 @@@@@@@@@= +@@@  *@@@@@@@@@@@
 @@@@@@@@* =@@@@  %@@@@@@@@@@@
 @@@@@@@@..@@@@%  @@@@@@@@@@@@
```

```
@@@@@@@# *@@@@-  @@@@@@@@@@
@@@@@@@: @@@@%   @@@@@@@@@@
@@@@@@@: @@@@-   @@@@@@@@@@
@@@@@@@: =+*= +: *@@@@@@@@@@
@@@@@@@*.    +@: *@@@@@@@@@@
@@@@@@@@%#**#@@: *@@@@@@@@@@
@@@@@@@@@@@@@@@: -@@@@@@@@@@
@@@@@@@@@@@@@@@+ :@@@@@@@@@@
@@@@@@@@@@@@@@@*  @@@@@@@@@@
@@@@@@@@@@@@@@@   %@@@@@@@@@@
@@@@@@@@@@@@@@@   #@@@@@@@@@@
@@@@@@@@@@@@@@@@: +@@@@@@@@@@
@@@@@@@@@@@@@@@@- +@@@@@@@@@@
@@@@@@@@@@@@@@@@*:%@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

[I] Algorithm chose 9
&&&& PASSED TensorRT.sample_mlp # build/x86_64-linux/sample_mlp
```

This output shows that the sample ran successfully; **PASSED**.

Sample `--help` options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
./sample_mlp --help

Usage: ./sample_mlp [-h or --help] [-d or --datadir=<path to data directory>]
 [--useDLACore=<int>]

--help Display help information

--datadir Specify path to a data directory, overriding the default. This option
 can be used multiple times to add multiple directories. If no data directories
 are given, the default is to use (data/samples/mnist/, data/mnist/, data/
samples/mlp/, data/mlp/)

--useDLACore=N Specify a DLA engine for layers that support DLA. Value can range
 from 0 to n-1, where n is the number of DLA engines on the platform.

--int8 Run in Int8 mode.

--fp16 Run in FP16 mode.
```

Additional resources

The following resources provide a deeper understanding about MLP:

MLP

- TensorFlow tutorial
- MLP network

Models

MNIST data set

Documentation

- Working With TensorRT Using The C++ API

- NVIDIA’s TensorRT Documentation Library

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.

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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

17.1. README.md

Attention A summary of the **README.md** file is included here for your reference, however, you should always refer to the **README.md** within the package for the most recent documentation updates.

How does this sample work?

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.

Prerequisites

1. Install the dependencies for Python.
   - For Python 2 users, from the root directory, run: **python2 -m pip install -r requirements.txt**
   - For Python 3 users, from the root directory, run: **python3 -m pip install -r requirements.txt**

Running the sample

1. Run the sample to create a TensorRT inference engine and run inference: **python <parser>_resnet50.py**

Where **<parser>** is either **caffe**, **onnx**, or **uff**.

If the TensorRT sample data is not installed in the default location, for example **/usr/src/tensorrt/data/**, the **data** directory must be specified. **python <parser>_resnet50.py [-d DATA_DIR]**

For example: **python caffe_resnet50.py -d /path/to/my/data/**

2. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following: **Correctly recognized data/samples/resnet50/reflex_camera.jpeg as reflex camera**

Sample --help options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
usage: caffe_resnet50.py [-h] [-d DATADIR]
Runs a ResNet50 network with a TensorRT inference engine.
optional arguments:
-h, --help show this help message and exit
-d DATADIR, --datadir DATADIR
Location of the TensorRT sample data directory.
(default: /usr/src/tensorrt/data)
```

Additional resources

The following resources provide a deeper understanding about importing a model into TensorRT using Python:

ResNet-50

Deep Residual Learning for Image Recognition

Parsers

- Caffe Parser
- ONNX Parser
- UFF Parser

Documentation

- Working With TensorRT Using The Python API
- Importing A Model Using A Parser In Python
- NVIDIA's TensorRT Documentation Library

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

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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

18.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

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.

Freezing a TensorFlow graph

In order to use the command-line UFF utility, TensorFlow graphs must be frozen and saved as **.pb** files.

In this sample, the converter displays information about the input and output nodes, which you can use to the register inputs and outputs with the parser. In this case, we already know the details of the input and output nodes and have included them in the sample.

Freezing a Keras model

You can use the following sample code to freeze a Keras model.

```
def save(model, filename):
 # First freeze the graph and remove training nodes.
 output_names = model.output.op.name
 sess = tf.keras.backend.get_session()
 frozen_graph = tf.graph_util.convert_variables_to_constants(sess,
 sess.graph.as_graph_def(), [output_names])
 frozen_graph = tf.graph_util.remove_training_nodes(frozen_graph)
 # Save the model
 with open(filename, "wb") as ofile:
  ofile.write(frozen_graph.SerializeToString())
```

Prerequisites

1. Install the dependencies for Python.
   - For Python 2 users, from the root directory, run: **python2 -m pip install -r requirements.txt**
   - For Python 3 users, from the root directory, run: **python3 -m pip install -r requirements.txt**
2. Install the UFF toolkit and graph surgeon; depending on your TensorRT installation method, to install the toolkit and graph surgeon, choose the method you used to install TensorRT for instructions (see TensorRT Installation Guide: Installing TensorRT).

Running the sample

1. Run the sample to train the model and write out the frozen graph:

   ```
   mkdir models
   python model.py
   ```

2. Convert the **.pb** file to **.uff** using the convert-to-uff utility: **convert-to-uff models/lenet5.pb**

   Depending on how you installed TensorRT, this utility may also be located in **/usr/lib/python2.7/dist-packages/uff/bin/convert_to_uff.py** or **/usr/lib/python<PYTHON3 VERSION>/site-packages/uff/bin/convert_to_uff.py**.
3. Create a TensorRT inference engine from the UFF file and run inference: **python sample.py [-d DATA_DIR]**

   

   If the TensorRT sample data is not installed in the default location, for example **/usr/src/tensorrt/data/**, the data directory must be specified. For example: **python sample.py -d /path/to/my/data/.**
4. Verify that the sample ran successfully. If the sample runs successfully you should see a match between the test case and the prediction.

   ```
   Test Case: 2
   Prediction: 2
   ```

Sample `--help` options

To see the full list of available options and their descriptions, use the **-h** or **--help** command line option. For example:

```
usage: sample.py [-h] [-d DATADIR]
Runs an MNIST network using a UFF model file
optional arguments:
-h, --help show this help message and exit
-d DATADIR, --datadir DATADIR
Location of the TensorRT sample data directory.
(default: /usr/src/tensorrt/data)
```

Additional resources

The following resources provide a deeper understanding about training and running inference in TensorRT using Python:

Model

TensorFlow/Keras MNIST

Dataset

MNIST database

Documentation

- Working With TensorRT Using The Python API
- NVIDIA’s TensorRT Documentation Library

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.

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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

19.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

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.

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer
The Activation layer implements element-wise activation functions. Specifically, this sample uses the Activation layer with the type **`kRELU`**.

Convolution layer
The Convolution layer computes a 2D (channel, height, and width) convolution, with or without bias.

FullyConnected layer
The FullyConnected layer implements a matrix-vector product, with or without bias.

Pooling layer
The Pooling layer implements pooling within a channel. Supported pooling types are **`maximum`**, **`average`** and **`maximum-average blend`**.

Prerequisites

1. Install the dependencies for Python.
   - For Python 2 users, from the root directory, run: **`python2 -m pip install -r requirements.txt`**
   - For Python 3 users, from the root directory, run: **`python3 -m pip install -r requirements.txt`**

Running the sample

1. Run the sample to create a TensorRT inference engine and run inference:

```
python sample.py [-d DATA_DIR]
```

If the TensorRT sample data is not installed in the default location, for example **`/usr/src/tensorrt/data/`**, the data directory must be specified. For example: **`python sample.py -d /path/to/my/data/`**.

2. Verify that the sample ran successfully. If the sample runs successfully you should see a match between the test case and the prediction.

```
Test Case: 0
Prediction: 0
```

Sample `--help` options

To see the full list of available options and their descriptions, use the **`-h`** or **`--help`** command line option. For example:

```
usage: sample.py [-h] [-d DATADIR]
Runs an MNIST network using a PyTorch model
optional arguments:
-h, --help show this help message and exit
-d DATADIR, --datadir DATADIR
Location of the TensorRT sample data directory.
(default: /usr/src/tensorrt/data)
```

Additional resources

The following resources provide a deeper understanding about getting started with TensorRT using Python:

Model

MNIST model

Dataset

MNIST database

Documentation

- Working With TensorRT Using The Python API
- NVIDIA’s TensorRT Documentation Library

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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

20.1. README.md

Attention A summary of the **README.md** file is included here for your reference, however, you should always refer to the **README.md** within the package for the most recent documentation updates.

How does this sample work?

This sample demonstrates how to use plugins written in C++ with the TensorRT Python bindings and the Caffe parser. This sample includes:

plugin/
This directory contains files for the FullyConnected layer plugin.

FullyConnected.h
This plugin implements CUDA, cuDNN, and cuBLAS.

pyFullyConnected.cpp
This plugin generates Python bindings for the **FCPlugin** and **FCPluginFactory** classes.

sample.py
This script runs an MNIST network using the provided FullyConnected layer plugin.

requirements.txt
This file specifies all the Python packages required to run this Python sample.

Prerequisites

For specific software versions, see the TensorRT Installation Guide.

1. Install cuDNN.
2. Install CMake.
3. Install cuBLAS.
4. Download **pybind11**. **git clone -b v2.2.3 https://github.com/pybind/pybind11.git**

   You can clone the repository anywhere, but the default configuration assumes that **pybind11** is located in your home directory.
5. Install the dependencies for Python.
   - For Python 2 users, from the root directory, run: **python2 -m pip install -r requirements.txt**
   - For Python 3 users, from the root directory, run: **python3 -m pip install -r requirements.txt**

Running the sample

1. Build the plugin and its corresponding Python bindings.

```
mkdir build && pushd build
cmake ..
```

If any of the dependencies are not installed in their default locations, you can manually specify them. For example:

```
cmake .. -DPYBIND11_DIR=/usr/local/pybind11/
-DCUDA_ROOT=/usr/local/cuda-9.2/
-DPYTHON3_INC_DIR=/usr/include/python3.6/
-DNVINFER_LIB=/path/to/libnvinfer.so -DTRT_INC_DIR=/path/to/
tensorrt/include/
```

`cmake ..` displays a complete list of configurable variables. If a variable is set to **`VARIABLE_NAME-NOTFOUND`**, then you'll need to specify it manually or set the variable it is derived from correctly.

The default behavior is to build bindings for both Python 2 and 3. To disable either one, for example, issue: **`cmake .. -DPYTHON3_INC_DIR=None`**

to disable Python 3.

2. Build the plugin.

```
make -j4
popd
```

3. Run the sample to perform inference using the plugin. For example, issue: **`python3 sample.py [-d DATA_DIR]`**

to run the sample with Python 3.

If the TensorRT sample data is not installed in the default location, for example **`/usr/src/tensorrt/data/`**, the data directory must be specified. For example: **`python sample.py -d /path/to/my/data/`**

A single artifact called **`mnist.engine`** is created in the source directory and contains a serialized engine.

4. Verify that the sample ran successfully. If the sample runs successfully you should see a match between the test case and the prediction.

```
Test Case: #
Prediction: #
```

Sample `--help` options

To see the full list of available options and their descriptions, use the **`-h`** or **`--help`** command line option. For example:

```
usage: sample.py [-h] [-d DATADIR]
```

```
Runs an MNIST network using a Caffe model file
optional arguments:
-h, --help show this help message and exit
-d DATADIR, --datadir DATADIR
Location of the TensorRT sample data directory.
(default: /usr/src/tensorrt/data)
```

Additional resources

The following resources provide a deeper understanding about adding a custom layer to your Caffe network using Python:

Network

MNIST network

Dataset

MNIST dataset

Documentation

- Working With TensorRT Using The Python API
- NVIDIA's TensorRT Documentation Library

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.

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.

A summary of the **README.md** file is included in this section for your reference, however, you should always refer to the **README.md** within the package for the most recent documentation updates.

21.1. README.md

Attention A summary of the **README.md** file is included here for your reference, however, you should always refer to the **README.md** within the package for the most recent documentation updates.

How does this sample work?

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.

This sample includes:

`plugin/`
This directory contains files for the Clip layer plugin.

`clipKernel.cu`
A CUDA kernel that clips input.

`clipKernel.h`
The header exposing the CUDA kernel to C++ code.

`customClipPlugin.cpp`
A custom TensorRT plugin implementation, which uses the CUDA kernel internally.

`customClipPlugin.h`
The ClipPlugin headers.

`lenet5.py`
This script trains an MNIST network that uses ReLU6 activation using the clip plugin.

`mnist_uff_relu6_plugin.py`
This script transforms the trained model into UFF (delegating ReLU6 activations to ClipPlugin instances) and runs inference in TensorRT.

`requirements.txt`
This file specifies all the Python packages required to run this Python sample.

Prerequisites

For specific software versions, see the TensorRT Installation Guide.

1. Install CMake.
2. Install the dependencies for Python.
   - For Python 2 users, from the root directory, run: **`python2 -m pip install -r requirements.txt`**
   - For Python 3 users, from the root directory, run: **`python3 -m pip install -r requirements.txt`**
3. Install the UFF toolkit and graph surgeon; depending on your TensorRT installation method, to install the toolkit and graph surgeon, choose the method you used to install TensorRT for instructions (see TensorRT Installation Guide: Installing TensorRT).

Running the sample

1. Build the plugin and its corresponding Python bindings.

```
mkdir build && pushd build
cmake ..
```

If any of the dependencies are not installed in their default locations, you can manually specify them. For example:

```
cmake .. \
  -DPYBIND11_DIR=/usr/local/pybind11/ \
  -DCUDA_ROOT=/usr/local/cuda-9.2/ \
  -DPYTHON3_INC_DIR=/usr/include/python3.6/ \
  -DNVINFER_LIB=/path/to/libnvinfer.so \
  -DTRT_INC_DIR=/path/to/tensorrt/include/
```

cmake .. displays a complete list of configurable variables. If a variable is set to **VARIABLE_NAME-NOTFOUND**, then you'll need to specify it manually or set the variable it is derived from correctly.

2. Build the plugin.

```
make -j
popd
```

3. Run the sample to train the model: **python3 lenet5.py**
4. Run inference using TensorRT with the custom clip plugin implementation: **python3 sample.py**
5. Verify that the sample ran successfully. If the sample runs successfully you should see a match between the test case and the prediction.

```
=== Testing ===
 Loading Test Case: 3
 Prediction: 3
```

Additional resources

The following resources provide a deeper understanding about getting started with TensorRT using Python:

Model

LeNet model

Dataset

MNIST dataset

Documentation

- Working With TensorRT Using The Python API
- NVIDIA's TensorRT Documentation Library

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 608x608 pixels, including pre and post-processing. This sample is based on the YOLOv3-608 paper.

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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

22.1. README.md

Attention A summary of the **README.md** file is included here for your reference, however, you should always refer to the **README.md** within the package for the most recent documentation updates.

How does this sample work?

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.

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

Prerequisites

For specific software versions, see the TensorRT Installation Guide.

1. Install ONNX-TensorRT: TensorRT backend for ONNX. ONNX-TensorRT includes layer implementations for the required ONNX operators **Upsample** and **LeakyReLU**.
2. Install the dependencies for Python.
   - For Python 2 users, from the root directory, run: **python2 -m pip install -r requirements.txt**
   - For Python 3 users, from the root directory, run: **python3 -m pip install -r requirements.txt**

Running the sample

1. Create an ONNX version of YOLOv3 with the following command. The Python script will also download all necessary files from the official mirrors (only once).

   ```
   python yolov3_to_onnx.py
   ```

   When running the above command for the first time, the output should look similar to the following:

```
Downloading from https://raw.githubusercontent.com/pjreddie/darknet/
f86901f6177dfc6116360a13cc06ab680e0c86b0/cfg/yolov3.cfg, this may take a
 while...
 100%
 [..............................................................................]
 8342 / 8342
 Downloading from https://pjreddie.com/media/files/yolov3.weights, this may
 take a while...
 100%
 [..............................................................................]
 248007048 / 248007048
 [...]
 %106_convolutional = Conv[auto_pad = u'SAME_LOWER', dilations = [1, 1],
 kernel_shape = [1, 1], strides = [1, 1]]
 (%105_convolutional_lrelu, %106_convolutional_conv_weights,
 %106_convolutional_conv_bias)
 return %082_convolutional, %094_convolutional,%106_convolutional
 }
```

2. Build a TensorRT engine from the generated ONNX file and run inference on a sample image, which will also be downloaded during the first run.

```
python onnx_to_tensorrt.py
```

When running the above command for the first time, the output should look similar to the following:

```
Downloading from https://github.com/pjreddie/darknet/raw/
f86901f6177dfc6116360a13cc06ab680e0c86b0/data/dog.jpg, this may take a
 while...
 100%
 [..............................................................................]
 163759 / 163759
 Building an engine from file yolov3.onnx, this may take a while...
 Running inference on image dog.jpg...
 Saved image with bounding boxes of detected objects to dog_bboxes.jpg.
```

3. Verify that the sample ran successfully. If the sample runs successfully you should see output similar to the following:

```
Downloading from https://github.com/pjreddie/darknet/raw/
f86901f6177dfc6116360a13cc06ab680e0c86b0/data/dog.jpg, this may take a
 while…
 100%
 [..............................................................................]
 163759 / 163759
 Loading ONNX file from path yolov3.onnx...
 Beginning ONNX file parsing
 Completed parsing of ONNX file
 Building an engine from file yolov3.onnx; this may take a while...
 Completed creating Engine
 Running inference on image dog.jpg...
 [[135.14841333 219.59879284 184.30209195 324.0265199 ]
   [ 98.30805074 135.72613533 499.71263299 299.25579652]
   [478.00605802 81.25702449 210.57787895 86.91502688]] [0.99854713
 0.99880403 0.93829258] [16 1 7]
 Saved image with bounding boxes of detected objects to dog_bboxes.png.
```

You should be able to visually confirm whether the detection was correct.

Additional resources

The following resources provide a deeper understanding about the model used in this sample, as well as the dataset it was trained on:

Model

YOLOv3: An Incremental Improvement

Dataset

COCO dataset

Documentation

- YOLOv3-608 paper
- Working With TensorRT Using The Python API
- NVIDIA's TensorRT Documentation Library

Chapter 23. OBJECT DETECTION WITH SSD IN PYTHON

What Does This Sample Do?

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

This sample is based on the SSD: Single Shot MultiBox Detector paper. The SSD network, built on the VGG-16 network, performs the task of object detection and localization in a single forward pass of the network. This approach discretizes the output space of bounding boxes into a set of default boxes over different aspect ratios and scales per feature map location. At prediction time, the network generates scores for the presence of each object category in each default box and produces adjustments to the box to better match the object shape. Additionally, the network combines predictions from multiple features with different resolutions to naturally handle objects of various sizes.

This sample is based on the TensorFlow implementation of SSD. For more information, download ssd_inception_v2_coco. Unlike the paper, the TensorFlow SSD network was trained on the InceptionV2 architecture using the MSCOCO dataset which has 91 classes (including the background class). The config details of the network can be found here.

Where Is This Sample Located?

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

Getting Started:

Refer to the **/usr/src/tensorrt/samples/python/uff_ssd/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.

A summary of the **README.md** file is included in this section for your reference, however, you should always refer to the **README.md** within the package for the most recent documentation updates.

23.1. README.md

Attention A summary of the `README.md` file is included here for your reference, however, you should always refer to the `README.md` within the package for the most recent documentation updates.

How does this sample work?

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.

The SSD network performs the task of object detection and localization in a single forward pass of the network. The TensorFlow SSD network was trained on the InceptionV2 architecture using the MSCOCO dataset.

The sample makes use of TensorRT plugins to run the SSD network. To use these plugins the TensorFlow graph needs to be preprocessed.

When picking an object detection model for our application the usual trade-off is between model accuracy and inference time. In this sample we show how inference time of pretrained network can be greatly improved, without any decrease in accuracy, using TensorRT. In order to do that, we take a pretrained Tensorflow model, and use TensorRT's UffParser to build a TensorRT inference engine.

The main components of this network are the Preprocessor, FeatureExtractor, BoxPredictor, GridAnchorGenerator and Postprocessor.

Preprocessor The preprocessor step of the graph is responsible for resizing the image. The image is resized to a 300x300x3 size tensor. The preprocessor step also performs normalization of the image so all pixel values lie between the range [-1, 1].

FeatureExtractor The FeatureExtractor portion of the graph runs the InceptionV2 network on the preprocessed image. The feature maps generated are used by the anchor generation step to generate default bounding boxes for each feature map.

In this network, the size of feature maps that are used for anchor generation are [(19x19), (10x10), (5x5), (3x3), (2x2), (1x1)].

BoxPredictor The BoxPredictor step takes in a high level feature map as input and produces a list of box encodings (x-y coordinates) and a list of class scores for each of these encodings per feature map. This information is passed to the postprocessor.

GridAnchorGenerator The goal of this step is to generate a set of default bounding boxes (given the scale and aspect ratios mentioned in the config) for

each feature map cell. This is implemented as a plugin layer in TensorRT called the **gridAnchorGenerator** plugin. The registered plugin name is **GridAnchor_TRT**.

Postprocessor The postprocessor step performs the final steps to generate the network output. The bounding box data and confidence scores for all feature maps are fed to the step along with the pre-computed default bounding boxes (generated in the **GridAnchorGenerator** namespace). It then performs NMS (non-maximum suppression) which prunes away most of the bounding boxes based on a confidence threshold and IoU (Intersection over Union) overlap, thus storing only the top N boxes per class. This is implemented as a plugin layer in TensorRT called the **NMS** plugin. The registered plugin name is **NMS_TRT**.

This sample also implements another plugin called **FlattenConcat** which is used to flatten each input and then concatenate the results. This is applied to the location and confidence data before it is fed to the post processor step since the NMS plugin requires the data to be in this format.

For details on how a plugin is implemented, see the implementation of **FlattenConcat** plugin and **FlattenConcatPluginCreator** in the **sampleUffSSD.cpp** file in the **tensorrt/samples/sampleUffSSD** directory.

Processing the input graph

The TensorFlow SSD graph has some operations that are currently not supported in TensorRT. Using GraphSurgeon, we can combine multiple operations in the graph into a single custom operation which can be implemented using a plugin layer in TensorRT. Currently, GraphSurgeon provides the ability to stitch all nodes within a namespace into one custom node.

To use GraphSurgeon, the **convert-to-uff** utility should be called with a **-p** flag and a config file. The config script should also include attributes for all custom plugins which will be embedded in the generated **.uff** file. Current sample scripts for SSD is located in **/usr/src/tensorrt/samples/sampleUffSSD/config.py**.

Using GraphSurgeon, we were able to remove the preprocessor namespace from the graph, stitch the **GridAnchorGenerator** namespace to create the **GridAnchorGenerator** plugin, stitch the postprocessor namespace to the **NMS** plugin and mark the concat operations in the BoxPredictor as **FlattenConcat** plugins.

The TensorFlow graph has some operations like **Assert** and **Identity** which can be removed for inferencing. Operations like **Assert** are removed and leftover nodes (with no outputs once assert is deleted) are then recursively removed.

Identity operations are deleted and the input is forwarded to all the connected outputs. Additional documentation on the graph preprocessor can be found in the TensorRT API.

uff_ssd plugins

Details about how to create TensorRT plugins can be found in Extending TensorRT With Custom Layers.

`GridAnchorGeneration` plugin

This plugin layer implements the grid anchor generation step in the TensorFlow SSD network. For each feature map we calculate the bounding boxes for each grid cell. In this network, there are 6 feature maps and the number of boxes per grid cell are as follows:

- [19x19] feature map: 3 boxes (19x19x3x4(co-ordinates/box))
- [10x10] feature map: 6 boxes (10x10x6x4)
- [5x5] feature map: 6 boxes (5x5x6x4)
- [3x3] feature map: 6 boxes (3x3x6x4)
- [2x2] feature map: 6 boxes (2x2x6x4)
- [1x1] feature map: 6 boxes (1x1x6x4)

`NMS` plugin

The **`NMS`** plugin generates the detection output based on location and confidence predictions generated by the BoxPredictor. This layer has three input tensors corresponding to location data (**`locData`**), confidence data (**`confData`**) and priorbox data (**`priorData`**).

The inputs to detection output plugin have to be flattened and concatenated across all the feature maps. We use the **`FlattenConcat`** plugin implemented in the sample to achieve this. The location data generated from the box predictor has the following dimensions:

```
19x19x12 -> Reshape -> 1083x4 -> Flatten -> 4332x1
10x10x24 -> Reshape -> 600x4 -> Flatten -> 2400x1
```

and so on for the remaining feature maps.

After concatenating, the input dimensions for **`locData`** input are of the order of 7668x1.

The confidence data generated from the box predictor has the following dimensions:

```
19x19x273 -> Reshape -> 1083x91 -> Flatten -> 98553x1
10x10x546 -> Reshape -> 600x91 -> Flatten -> 54600x1
```

and so on for the remaining feature maps.

After concatenating, the input dimensions for **`confData`** input are 174447x1.

The prior data generated from the grid anchor generator plugin has the following dimensions, for example 19x19 feature map has 2x4332x1 (there are two channels here because one channel is used to store variance of each coordinate that is used in the NMS step). After concatenating, the input dimensions for priorData input are of the order of 2x7668x1.

```
struct DetectionOutputParameters
{
  bool shareLocation, varianceEncodedInTarget;
  int backgroundLabelId, numClasses, topK, keepTopK;
  float confidenceThreshold, nmsThreshold;
  CodeTypeSSD codeType;
  int inputOrder[3];
  bool confSigmoid;
  bool isNormalized;
};
```

shareLocation and **varianceEncodedInTarget** are used for the Caffe implementation, so for the TensorFlow network they should be set to **true** and **false** respectively. The **confSigmoid** and **isNormalized** parameters are necessary for the TensorFlow implementation. If **confSigmoid** is set to **true**, it calculates the sigmoid values of all the confidence scores. The **isNormalized** flag specifies if the data is normalized and is set to **true** for the TensorFlow graph.

Verifying the output

After the builder is created (see Building An Engine In Python) and the engine is serialized (see Serializing A Model In Python), we can perform inference. Steps for deserialization and running inference are outlined in Performing Inference In Python.

The outputs of the SSD network are human interpretable. The post-processing work, such as the final NMS, is done in the NMS plugin. The results are organized as tuples of 7. In each tuple, the 7 elements are respectively image ID, object label, confidence score, (**x,y**) coordinates of the lower left corner of the bounding box, and (**x,y**) coordinates of the upper right corner of the bounding box. This information can be drawn in the output PPM image using the **writePPMFileWithBBox** function. The **visualizeThreshold** parameter can be used to control the visualization of objects in the image. It is currently set to 0.5 so the output will display all objects with confidence score of 50% and above.

Prerequisites

1. Install CMake.

```
**cmake >= 3.8**
```

2. Install the dependencies for Python.
   - For Python 2 users, from the root directory, run: **python2 -m pip install -r requirements.txt**
   - For Python 3 users, from the root directory, run: **python3 -m pip install -r requirements.txt**
3. Compile the **FlattenConcat** custom plugin from within the sample directory.

```
sh
mkdir -p build
cd build
cmake ..
make
cd ..
```

This should use **cmake** to build the **FlattenConcat** plugin and put it in the appropriate directory. This is needed because the frozen model that is used in this sample uses some TensorFlow operations that are not natively supported in TensorRT.

4. Optional: To evaluate the accuracy of the trained model using the VOC dataset, perform the following steps.

   a. Download the VOC dataset. Run the following command from the sample root directory.

   ```
   wget http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCtest_06-Nov-2007.tar
   tar xvf VOCtest_06-Nov-2007.tar
   ```

   The first command downloads the VOC dataset from the Oxford servers, and the second command unpacks the dataset.

   

   If you don't want to save VOC in the sample root directory, you'll need to adjust the **--voc_dir** argument to **voc_evaluation.py** script before running it. The default value of this argument is **<SAMPLE_ROOT>/VOCdevkit/VOC2007**.

Running the sample

Both the **detect_objects.py** and **voc_evaluation.py** scripts support separate advanced features, for example, lower precision inference, changing workspace directory and changing batch size.

1. Run the inference script:

   ```
   python detect_objects.py <IMAGE_PATH>
   ```

   Where **<IMAGE_PATH>** contains the image you want to run inference on using the SSD network. The script should work for all popular image formats, like PNG, JPEG, and BMP. Since the model is trained for images of size 300 x 300, the input image will be resized to this size (using bilinear interpolation), if needed.

   When the inference script is run for the first time, it will run the following things to prepare its workspace:

   - The script downloads the pretrained **ssd_inception_v2_coco_2017_11_17** model from the TensorFlow object detection API. The script converts this model to TensorRT format, and the conversion is tailored to this specific version of the model.
   - The script builds a TensorRT inference engine and saves it to a file. During this step, all TensorRT optimizations will be applied to frozen graph. This is a time consuming operation and it can take a few minutes.

After the workspace is ready, the script launches inference on the input image and saves the results to a location that will be printed on standard output. You can then open the saved image file and visually confirm that the bounding boxes are correct.

2. Run the VOC evaluation script.
   a. Run the script using TensorRT: **`python voc_evaluation.py`**
   b. Run the script using TensorFlow: **`python voc_evaluation.py tensorflow`**

Running the script using TensorFlow is much slower than the TensorRT evaluation.

   c. AP and mAP metrics are displayed at the end of the script execution. The metrics for the TensorRT engine should match those of the original TensorFlow model.

Sample `--help` options

To see the full list of available options and their descriptions, use the **`-h`** or **`--help`** command line option. For example:

```
usage: detect_objects.py [-h] [-p {32,16}] [-b MAX_BATCH_SIZE] [-w
 WORKSPACE_DIR] [-fc FLATTEN_CONCAT] [-o OUTPUT] INPUT_IMG_PATH

Run object detection inference on input image.
positional arguments:
 INPUT_IMG_PATH an image file to run inference on

optional arguments:
 -h, --help show this help message and exit
 -p {32,16}, --precision {32,16} desired TensorRT float precision to build an
 engine with
 -b MAX_BATCH_SIZE, --max_batch_size MAX_BATCH_SIZE max TensorRT engine batch
 size
 -w WORKSPACE_DIR, --workspace_dir WORKSPACE_DIR sample workspace directory
 -fc FLATTEN_CONCAT, --flatten_concat FLATTEN_CONCAT path of built FlattenConcat
 plugin
 -o OUTPUT, --output OUTPUT path of the output file
```

Additional resources

The following resources provide a deeper understanding about the SSD model and object detection:

Model

TensorFlow detection model zoo

Dataset

- ssd_inception_v2_coco
- MSCOCO dataset

Documentation

- Working With TensorRT Using The Python API

- SSD: Single Shot MultiBox Detector paper
- NVIDIA's TensorRT Documentation Library

Chapter 24. INT8 CALIBRATION IN PYTHON

What Does This Sample Do?

This sample, int8_caffe_mnist, demonstrates how to create an INT8 calibrator, build and calibrate an engine for INT8 mode, and finally run inference in INT8 mode.

Where Is This Sample Located?

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

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/python/int8_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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

24.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

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 (see **`calibrator.py`** for implementation details).

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

Prerequisites

1. Install the dependencies for Python.
   - For Python 2 users, from the root directory, run: **`python2 -m pip install -r requirements.txt`**
   - For Python 3 users, from the root directory, run: **`python3 -m pip install -r requirements.txt`**

Running the sample

1. Run the sample to create a TensorRT inference engine, perform IN8 calibration and run inference:

   ```
   python3 sample.py [-d DATA_DIR]
   ```

   to run the sample with Python 3.

   

   If the TensorRT sample data is not installed in the default location, for example **`/usr/src/tensorrt/data/`**, the **`data`** directory must be specified. For example: **`python sample.py -d /path/to/my/data/.`**

2. Verify that the sample ran successfully. If the sample runs successfully you should see a very high accuracy. For example:

   ```
   Expected Predictions:
   [1. 6. 5. 0. 2. 8. 1. 5. 6. 2. 3. 0. 2. 2. 6. 4. 3. 5. 5. 1. 7. 2. 1. 6.
   9. 1. 9. 9. 5. 5. 1. 6. 2. 2. 8. 6. 7. 1. 4. 6. 0. 4. 0. 3. 3. 2. 2. 3.
   6. 8. 9. 8. 5. 3. 8. 5. 4. 5. 2. 0. 5. 6. 3. 2. 8. 3. 9. 9. 5. 7. 9. 4.
   6. 7. 1. 3. 7. 3. 6. 6. 0. 9. 0. 1. 9. 9. 2. 8. 8. 0. 1. 6. 9. 7. 5. 3.
   4. 7. 4. 9.]
   Actual Predictions:
   [1 6 5 0 2 8 1 5 6 2 3 0 2 2 6 4 3 5 5 1 7 2 1 6 9 1 9 9 5 5 1 6 2 2 8 6 7
   1 4 6 0 4 0 3 3 2 2 3 6 8 9 8 5 3 8 5 4 5 2 0 5 6 3 2 8 3 9 9 5 7 9 4 6 7
   1 3 7 3 6 6 0 9 0 1 9 4 2 8 8 0 1 6 9 7 5 3 4 7 4 9]
   Accuracy: 99.0%
   ```

Sample `--help` options

To see the full list of available options and their descriptions, use the **`-h`** or **`--help`** command line option. For example:

```
usage: sample.py [-h]

Description for this sample

optional arguments:
```

```
-h, --help show this help message and exit
```

Additional resources

The following resources provide a deeper understanding about the model used in this sample:

Network

MNIST network

Dataset

MNIST dataset

Documentation

- Working With TensorRT Using The Python API
- Enabling INT8 Inference Using Python
- NVIDIA's TensorRT Documentation Library

Chapter 25. REFITTING AN ENGINE IN PYTHON

What Does This Sample Do?

This sample, 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. Refitting allows us to quickly modify the weights in a TensorRT engine without needing to rebuild.

Where Is This Sample Located?

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

Getting Started:

Refer to the **`/usr/src/tensorrt/samples/python/engine_refit_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.

A summary of the **`README.md`** file is included in this section for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

25.1. README.md

Attention A summary of the **`README.md`** file is included here for your reference, however, you should always refer to the **`README.md`** within the package for the most recent documentation updates.

How does this sample work?

This sample first reconstructs the model using the TensorRT network API. In the first pass, the weights for one of the conv layers (**conv_1**) are populated 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 again. With the weights now set correctly, inference should provide correct results.

TensorRT API layers and ops

In this sample, the following layers are used. For more information about these layers, see the TensorRT Developer Guide: Layers documentation.

Activation layer
The Activation layer implements element-wise activation functions. Specifically, this sample uses the Activation layer with the type **kRELU**.

Convolution layer
The Convolution layer computes a 2D (channel, height, and width) convolution, with or without bias.

FullyConnected layer
The FullyConnected layer implements a matrix-vector product, with or without bias.

Pooling layer
The Pooling layer implements pooling within a channel. Supported pooling types are **maximum**, **average** and **maximum-average blend**.

Prerequisites

1. Install the dependencies for Python.
   - For Python 2 users, from the root directory, run: **python2 -m pip install -r requirements.txt**
   - For Python 3 users, from the root directory, run: **python3 -m pip install -r requirements.txt**

Running the sample

1. Run the sample to create a TensorRT engine and run inference:

   ```
   python3 sample.py [-d DATA_DIR]
   ```

   to run the sample with Python 3.

   

   If the TensorRT sample data is not installed in the default location, for example **/usr/src/tensorrt/data/**, the **data** directory must be specified. For example: **python sample.py -d /path/to/my/data/.**

2. Verify that the sample ran successfully. If the sample runs successfully you should see a match between the test case and the prediction after refitting.

```
Output Before Engine Refit
Test Case: 8
Prediction: 1
Output After Engine Refit
Test Case: 0
Prediction: 0
```

Sample `--help` options

To see the full list of available options and their descriptions, use the **`-h`** or **`--help`** command line option. For example:

```
usage: sample.py [-h]

Description for this sample

optional arguments:
-h, --help show this help message and exit
```

Additional resources

The following resources provide a deeper understanding about the engine refitting functionality and the network used in this sample:

Network

MNIST network

Dataset

MNIST dataset

Documentation

- Working With TensorRT Using The Python API
- Refitting An Engine
- NVIDIA's TensorRT Documentation Library

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