TENSORRT SAMPLES

SWE-SWDOCTRT-001-SAMG_vTensorRT 5.1.0 RC | January 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 optical character recognition (OCR) network 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

New Sample Name	Old Sample Name	Description
		deserialized. Enables a custom layer in NvCaffeParser.
Neural Machine Translation (NMT) Using Sequence To Sequence (seq2seq) Models	sampleNMT	An end-to-end sample that takes a TensorFlow seq2seq model, builds an engine, and runs inference using the generated network.
Object Detection With FasterRCNN	sampleFasterRCNN	Uses TensorRT plugins, performs inference, and implements a fused custom layer for end-to-end inferencing of a FasterRCNN 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,

New Sample Name	Old Sample Name	Description
		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 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 OCR Network
- Import The TensorFlow Model And Run Inference
- "Hello World" For TensorRT From ONNX
- Applying FP16 To GoogleNet And Profiling The App
- 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
- Neural Machine Translation (NMT) Using Sequence To Sequence (seq2seq) Models
- Object Detection With FasterRCNN
- 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 Multi-Layer Perceptron (MLP)

Running C++ Samples

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>
```

1.2. Python Samples

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

- Introduction To Importing Caffe, TensorFlow And ONNX Models Into TensorRT Using Python
- "Hello World" For TensorRT Using TensorFlow And Python
- "Hello World" For TensorRT Using PyTorch And Python
- Adding A Custom Layer To Your Caffe Network In TensorRT In Python
- Adding A Custom Layer To Your TensorFlow Network In TensorRT In Python
- Object Detection With The ONNX TensorRT Backend In Python
- Object Detection With SSD In Python
- INT8 Calibration In Python
- Engine Refit In Python

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 demonstrates how to:

- Perform the basic setup and initialization of TensorRT using the Caffe parser
- Import a trained Caffe model using Caffe parser (see Importing A Caffe Model Using The C++ Parser API)
- Build an engine (see Building An Engine In C++)
- Serialize and deserialize the engine (see Serializing A Model In C++)
- Use the engine to perform inference on an input image (see Performing Inference in C++)

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

Notes About This Sample:
The Caffe model was trained on the MNIST dataset, where the dataset is from the NVIDIA DIGITS tutorial.

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 digits, showing which digit is most probably that in the image.

An example of ASCII rendering of the input image with digit 8:

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

Figure 1 ASCII output

An example of the output from network, classifying the digit 8 from the above image:

```
0:
1:
2:
3:
4:
5:
6:
7:
8: **********
9:
```

Figure 2 Decision output

Chapter 3. BUILDING A SIMPLE OCR NETWORK

What Does This Sample Do?

This sample is similar to "Hello World" For TensorRT sample. Both of these samples use the same model, handle the same input, and expect similar output. In contrast to the "Hello World" for TensorRT sample, this sample demonstrates how to:

- Build an optical character recognition (OCR) network by individually creating every layer
- Load the layers with their weights and connect the layers by linking their inputs and outputs
- Perform inference

Where Is This Sample Located?

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

Notes About This Sample:

For a detailed description of how to create layers using the C++ API, see Creating A Network Definition In C++. For a detailed description of how to create layers using the Python API, see Creating A Network Definition In Python.

Notes About Weights:

When you build a network by individually creating every layer, ensure you provide the per-layer weights to TensorRT in host memory. You will need to extract weights from their pre-trained model and deep learning framework and have these per-layer weights loaded in host memory to pass to TensorRT during network creation.

Chapter 4. IMPORT THE TENSORFLOW MODEL AND RUN INFERENCE

What Does This Sample Do?

This sample demonstrates how to:

- Import a TensorFlow model trained on the MNIST dataset
- Create the UFF Parser (see Importing From TensorFlow Using Python)
- Use the UFF Parser, register inputs and outputs, provide the dimensions and the order of the input tensor
- Load a trained TensorFlow model converted to UFF
- Build an engine (see Building An Engine In C++)
- Use the engine to perform inference (see Performing Inference In C++)

Where Is This Sample Located?

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

Notes About This Sample:

The TensorFlow model has been converted to UFF 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 format 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, and convert it to run in TensorRT. The TensorFlow to UFF converter creates an output file in a format called UFF which can then be read in TensorRT.

Chapter 5. “HELLO WORLD” FOR TENSORRT FROM ONNX

What Does This Sample Do?

This sample demonstrates how to:

- Configure the ONNX parser
- Convert an MNIST network in ONNX format to a TensorRT network
- Build the engine and run inference using the generated TensorRT network
- Covers Importing An ONNX Model Using The C++ Parser API and Importing From ONNX Using Python

This sample shows the conversion of an MNIST network in Open Neural Network Exchange (ONNX) format to a TensorRT network. ONNX is a standard for representing deep learning models that enable models to be transferred between frameworks. For more information about the ONNX format, see GitHub: ONNX. You can find a collection of ONNX networks at GitHub: ONNX Models. The network used in this sample can be found here.

Where Is This Sample Located?

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

5.1. 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();
```

5.2. 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.

5.3. Building The Engine And Running Inference

Before you can run inference, you must first build 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);
```

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

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]`.

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 the sampleMNIST described in "Hello World" For TensorRT.

Chapter 6. APPLYING FP16 TO GOOGLENET AND PROFILING THE APP

What Does This Sample Do?

This sample demonstrates how to import a model trained with Caffe into TensorRT using GoogleNet as an example.

Where Is This Sample Located?

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

6.1. Configuring The Builder

The sampleGoogleNet sample builds a network based on a saved Caffe model and network description. For more information, see Importing A Caffe Model Using The C++ Parser API or Importing From Caffe Using Python.

This sample uses optimized FP16 mode (see Enabling FP16 Inference Using C++ or Enabling FP16 Inference Using Python). To use **`Half2Mode`**, two additional steps are required:

1. Create an input network with 16-bit weights, by supplying the DataType::kHALF parameter to the parser.

   ```
   const IBlobNameToTensor *blobNameToTensor =
     parser->parse(locateFile(deployFile).c_str(),
                   locateFile(modelFile).c_str(),
                   *network,
                   DataType::kHALF);
   ```
2. Configure the builder to use **`Half2Mode`**.

   ```
   builder->setFp16Mode(true);
   ```

Chapter 7. BUILDING AN RNN NETWORK LAYER BY LAYER

What Does This Sample Do?
This sample demonstrates how to generate a simple RNN based on the charRNN network using the Penn Treebank (PTB) dataset. For more information about character level modeling, see char-rnn.

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

Notes About This Sample:
Use the TensorRT API documentation to familiarize yourself with the following layers:

- RNNv2 layer
  - Weights are set for each gate and layer individually.
  - The input format for RNNv2 is BSE (Batch, Sequence, Embedding).
- MatrixMultiply
- ElementWise
- TopK

7.1. Network Configuration

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.

7.1.1. RNNv2 Layer Setup

The first layer in the network is an RNN layer. This is added and configured in the `addRNNv2Layer()` function. This layer consists of the following configuration parameters:

Operation
: This defines the operation of the RNN cell. Supported operations are currently **`relu`**, **`LSTM`**, **`GRU`**, and **`tanh`**.

Direction
: This defines whether the RNN is unidirectional or bidirectional (BiRNN).

Input mode
: This defines whether the first layer of the RNN carries out a matrix multiply (linear mode), or the matrix multiply is skipped (skip mode).

For the purpose of the CharRNN network, we will be using a linear, unidirectional LSTM cell containing **`LAYER_COUNT`** number of stacked layers. The code below shows how to create this RNNv2 layer.

C++ code snippet

```
auto rnn = network->addRNNv2(*data, LAYER_COUNT, HIDDEN_SIZE, SEQ_SIZE,
 RNNOperation::kLSTM);
```

Python code snippet

```
rnn = network.add_rnn_v2(data, LAYER_COUNT, HIDDEN_SIZE, SEQ_SIZE,
 trt.RNNOperation.LSTM)
```

For the RNNv2 layer, weights and bias need to be set separately. For more information, see RNNv2 Layer - Optional Inputs.

For more information, see the TensorRT API documentation.

7.1.2. RNNv2 Layer - Optional Inputs

If there are cases where the hidden and cell states need to be pre-initialized to a non-zero value, then you can pre-initialize them via the **`setHiddenState`** and **`setCellState`** calls. These are optional inputs to the RNN.

C++ code snippet

```
rnn->setHiddenState(*hiddenIn);
if (rnn->getOperation() == RNNOperation::kLSTM)
    rnn->setCellState(*cellIn);
```

Python code snippet

```
rnn.hidden_state = hidden_in
if rnn.op == trt.RNNOperation.LSTM:
rnn.cell_state = cell_in
```

7.1.3. MatrixMultiply Layer Setup

The Matrix Multiplication layer is used to execute the first step of the functionality provided by a FullyConnected layer. As shown in the code below, a Constant layer will need to be used so that the FullyConnected weights can be stored in the engine. The output of the Constant and RNN layers are then used as inputs to the Matrix Multiplication layer. The RNN output is transposed so that the dimensions for the MatrixMultiply are valid.

C++ code snippet

```
weightMap["trt_fcw"] = transposeFCWeights(weightMap[FCW_NAME]);
auto fcwts = network->addConstant(Dims2(VOCAB_SIZE, HIDDEN_SIZE),
 weightMap["trt_fcw"]);
auto matrixMultLayer = network->addMatrixMultiply(
*fcwts->getOutput(0), false, *rnn->getOutput(0), true);
assert(matrixMultLayer != nullptr);
matrixMultLayer->getOutput(0)->setName("Matrix Multiplication output");
```

Python code snippet

```
weight_map["trt_fcw"] = transpose_fc_weights(weight_map[FCW_NAME])
fc_wts = network.add_constant((VOCAB_SIZE, HIDDEN_SIZE),
 weight_map["trt_fcw"])
matrix_mult_layer = network.add_matrix_multiply(
fc_wts.get_output(0), trt.MatrixOperation.NONE,  rnn.get_output(0),
 trt.MatrixOperation.TRANSPOSE)
assert matrix_mult_layer != None
matrix_mult_layer.get_output(0).name =
"Matrix Multiplication output"
```

For more information, see the TensorRT API documentation.

7.1.4. ElementWise Layer Setup

The ElementWise layer is used to execute the second step of the functionality provided by a FullyConnected layer. The output of the **fcbias** Constant layer and Matrix Multiplication layer are used as inputs to the ElementWise layer. The output from this layer is then supplied to the TopK layer. The code below demonstrates how to setup the layer:

C++ code snippet

```
auto fcbias = network->addConstant(Dims2(VOCAB_SIZE, 1),
 weightMap[FCB_NAME]);
auto addBiasLayer = network->addElementWise(
*matrixMultLayer->getOutput(0),
*fcbias->getOutput(0), ElementWiseOperation::kSUM);
assert(addBiasLayer != nullptr);
addBiasLayer->getOutput(0)->setName("Add Bias output");
```

Python code snippet

```
fc_bias = network.add_constant((VOCAB_SIZE, 1), weightMap[FCB_NAME])
add_bias_layer = network.add_elementwise(
matrix_mult_layer.get_output(0),
fc_bias.get_output(0), trt.ElementWiseOperation.SUM)
assert add_bias_layer != None
add_bias_layer.get_output(0).name = "Add Bias output"
```

For more information, see the TensorRT API documentation.

7.1.5. TopK Layer Setup

The TopK layer is used to identify the character that has the maximum probability of appearing next.

The layer has two outputs. The first output is an array of the top `K` values. The second, which is of more interest to us, is the index at which these maximum values appear.

The code below sets up the TopK layer and assigns the **`OUTPUT_BLOB_NAME`** to the second output of the layer.

C++ code snippet

```
auto pred =  network->addTopK(*addBiasLayer->getOutput(0),
                    nvinfer1::TopKOperation::kMAX, 1, reduceAxis);
assert(pred != nullptr);
pred->getOutput(1)->setName(OUTPUT_BLOB_NAME);
```

Python code snippet

```
pred = network.add_topk(add_bias_layer.get_output(0),
                trt.TopKOperation.MAX, 1, reduce_axis)
assert pred != None
pred.get_output(1).name = OUTPUT_BLOB_NAME
```

For more information, see the TensorRT API documentation.

7.1.6. Marking The Network Outputs

After the network is defined, mark the required outputs. RNN output tensors that are not marked as network outputs or used as inputs to another layer are dropped.

C++ code snippet

```
network->markOutput(*pred->getOutput(1));
pred->getOutput(1)->setType(DataType::kINT32);
rnn->getOutput(1)->setName(HIDDEN_OUT_BLOB_NAME);
network->markOutput(*rnn->getOutput(1));
if (rnn->getOperation() == RNNOperation::kLSTM)
{
rnn->getOutput(2)->setName(CELL_OUT_BLOB_NAME);
network->markOutput(*rnn->getOutput(2));
};
```

Python code snippet

```
network.mark_output(pred.get_output(1))
pred.get_output(1).dtype = trt.int32
rnn.get_output(1).name = HIDDEN_OUT_BLOB_NAME
network.mark_output(rnn.get_output(1))
if rnn.op == trt.RNNOperation.LSTM:
rnn.get_output(2).name = CELL_OUT_BLOB_NAME
network.mark_output(rnn.get_output(2))
```

```
network->markOutput(*pred->getOutput(1));
pred->getOutput(1)->setType(DataType::kINT32);
```

```
rnn->getOutput(1)->setName(HIDDEN_OUT_BLOB_NAME);
network->markOutput(*rnn->getOutput(1));
if (rnn->getOperation() == RNNOperation::kLSTM)
{
rnn->getOutput(2)->setName(CELL_OUT_BLOB_NAME);
network->markOutput(*rnn->getOutput(2));
};
```

7.2. RNNv2 Workflow - From TensorFlow To TensorRT

The following sections provide an end-to-end walkthrough of how to train your model in TensorFlow and convert the weights into a format that TensorRT can use.

7.2.1. Training A CharRNN Model With TensorFlow

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.

There are also multiple GitHub repositories that contain CharRNN implementations that will work out of the box. Tensorflow-char-rnn is one such implementation.

7.2.2. Exporting Weights From A TensorFlow Model Checkpoint

A python script **`/usr/src/tensorrt/samples/common/dumpTFWts.py`** 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.

7.2.3. Loading And Converting Weights Format

After the TensorFlow weights have been exported into a single **`WTS`** file, the next step is to load the weights and convert them into the TensorRT weights format. This is done by the **`loadWeights`** and then the **`convertRNNWeights`** and **`convertRNNBias`** functions. The functions contain detailed descriptions of the loading and conversion process. You can use those as guides in case you need to write your own conversion functions. After the conversion has taken place, the memory holding the converted weights is added to the weight map so that it can be deallocated once the engine has been built.
C++ code snippet

```
Weights rnnwL0 = convertRNNWeights(weightMap[RNNW_L0_NAME]);
Weights rnnbL0 = convertRNNBias(weightMap[RNNB_L0_NAME]);
Weights rnnwL1 = convertRNNWeights(weightMap[RNNW_L1_NAME]);
Weights rnnbL1 = convertRNNBias(weightMap[RNNB_L1_NAME]);
...
weightMap["rnnwL0"] = rnnwL0;
weightMap["rnnbL0"] = rnnbL0;
```

```
weightMap["rnnwL1"] = rnnwL1;
weightMap["rnnbL1"] = rnnbL1;
```

Python code snippet

```
rnnw_L0 = convert_rnn_weights(weight_map[RNNW_L0_NAME])
rnnb_L0 = convert_rnn_bias(weight_map[RNNB_L0_NAME])
rnnw_L1 = convert_rnn_weights(weight_map[RNNW_L1_NAME])
rnnb_L1 = convert_rnn_bias(weight_map[RNNB_L1_NAME])
...
weight_map["rnnw_L0"] = rnnw_L0
weight_map["rnnb_L0"] = rnnb_L0
weight_map["rnnw_L1"] = rnnw_L1
weight_map["rnnb_L1"] = rnnb_L1
```

7.2.4. RNNv2: Setting Weights And Bias

After the conversion to the TensorRT format, the RNN weights and biases are stored in their respective contiguous arrays. They are stored in the format of `[ ` W_L`f, `W_L`i, `W_L`c, `W_L`o, `R_L`f, `R_L`i, `R_L`c, `R_L`o ]`, where:

`W`
: The weights for the input.

`R`
: The weights for the recurrent input.

`f`
: Corresponds to the forget gate.

`i`
: Corresponds to the input gate.

`c`
: Corresponds to the cell gate.

`o`
: Corresponds to the output gate.

The code below takes advantage of this memory layout and iterates over the two layers and the eight gates to extract and set the correct gate weights and gate biases for the RNN layer.

C++ code snippet

```
for (int gateIndex = 0; gateIndex < NUM_GATES; gateIndex++)
{
    // extract weights and bias for a given gate and layer
    Weights gateWeightL0{.type = dataType,
.values = (void*)(wtsL0 + kernelOffset),
.count = DATA_SIZE * HIDDEN_SIZE};
    Weights gateBiasL0{.type = dataType,
.values = (void*)(biasesL0 + biasOffset),
.count = HIDDEN_SIZE};
    Weights gateWeightL1{.type = dataType,
.values = (void*)(wtsL1 + kernelOffset),
.count = DATA_SIZE * HIDDEN_SIZE};
    Weights gateBiasL1{.type = dataType,
.values = (void*)(biasesL1 + biasOffset),
.count = HIDDEN_SIZE};

    // set weights and bias for given gate
    rnn->setWeightsForGate(0, gateOrder[gateIndex % 4],
(gateIndex < 4), gateWeightL0);
    rnn->setBiasForGate(0, gateOrder[gateIndex % 4],
(gateIndex < 4), gateBiasL0);
    rnn->setWeightsForGate(1, gateOrder[gateIndex % 4],
(gateIndex < 4), gateWeightL1);
    rnn->setBiasForGate(1, gateOrder[gateIndex % 4],
(gateIndex < 4), gateBiasL1);

    // Update offsets
    kernelOffset = kernelOffset + DATA_SIZE * HIDDEN_SIZE;
    biasOffset = biasOffset + HIDDEN_SIZE;
}
```

Python code snippet

```
rnnw_L0_wts = numpy.split(rnnw_L0, 2*len(gate_order))
rnnb_L0_wts = numpy.split(rnnb_L0, 2*len(gate_order))
rnnw_L1_wts = numpy.split(rnnw_L1, 2*len(gate_order))
rnnb_L1_wts = numpy.split(rnnb_L1, 2*len(gate_order))
for i in range(2*len(gate_order)):
# set weights and bias for given gate
rnn.set_weights_for_gate(0, gate_order[i % len(gate_order)], (i <
 len(gate_order)), rnnw_L0_wts[i])
rnn.set_bias_for_gate(0, gate_order[i % len(gate_order)],  (i <
 len(gate_order)), rnnb_L0_wts[i])
rnn.set_weights_for_gate(1, gate_order[i % len(gate_order)], (i <
 len(gate_order)), rnnw_L1_wts[i])
rnn.set_bias_for_gate(1, gate_order[i % len(gate_order)],  (i <
 len(gate_order)), rnnb_L1_wts[i])
```

7.3. Seeding The Network

After the network is built, it is seeded with preset inputs so that the RNN can start generating data. Inside **stepOnce**, the output states are preserved for use as inputs on the next timestep.

C++ code snippet

```
for (auto &a : input)
{
     std::copy(static_cast<const float*>(embed.values) +
 char_to_id[a]*DATA_SIZE,
               static_cast<const float*>(embed.values) +
 char_to_id[a]*DATA_SIZE + DATA_SIZE,
               data[INPUT_IDX]);
     stepOnce(data, output, buffers, indices, stream, context);
     cudaStreamSynchronize(stream);

     // Copy Ct/Ht to the Ct-1/Ht-1 slots.
     std::memcpy(data[HIDDEN_IN_IDX], data[HIDDEN_OUT_IDX],
 gSizes[HIDDEN_IN_IDX] * sizeof(float));
     std::memcpy(data[CELL_IN_IDX], data[CELL_OUT_IDX], gSizes[CELL_IN_IDX] *
 sizeof(float));

     genstr.push_back(a);
}
// Extract first predicted character
uint32_t predIdx = *reinterpret_cast<uint32_t*>(data[OUTPUT_IDX]);
genstr.push_back(id_to_char[predIdx]);
```

Python code snippet

```
for a in input:
```

```
data[INPUT_IDX] = embed[char_to_id[a]]
stepOnce(data, output, buffers, indices, stream, context)
stream.synchronize()

# Copy Ct/Ht to the Ct-1/Ht-1 slots.
data[HIDDEN_IN_IDX] = data[HIDDEN_OUT_IDX]
data[CELL_IN_IDX] = data[CELL_OUT_IDX]

gen_str += a

# Extract first predicted character
predIdx = data[OUTPUT_IDX][0]
genstr += id_to_char[predIdx]
```

7.4. Generating Data

The following code is similar to the seeding code, however, this code generates an output character based on the output probability distribution. The following code simply selects the character with the highest probability. The final result is stored in **`genstr`**.

C++ code snippet

```
for (size_t x = 0, y = expected.size(); x < y; ++x)
{
     std::copy(static_cast<const float*>(embed.values) +
 char_to_id[*genstr.rbegin()]*DATA_SIZE,
             static_cast<const float*>(embed.values) +
 char_to_id[*genstr.rbegin()]*DATA_SIZE + DATA_SIZE,
             data[INPUT_IDX]);

     stepOnce(data, output, buffers, indices, stream, context);
     cudaStreamSynchronize(stream);

     // Copy Ct/Ht to the Ct-1/Ht-1 slots.
     std::memcpy(data[HIDDEN_IN_IDX], data[HIDDEN_OUT_IDX],
 gSizes[HIDDEN_IN_IDX] * sizeof(float));
     std::memcpy(data[CELL_IN_IDX], data[CELL_OUT_IDX], gSizes[CELL_IN_IDX] *
 sizeof(float));

uint32_t predIdx = *(output);
     genstr.push_back(id_to_char[predIdx]);
}
```

Python code snippet

```
for x in range(len(expected)):
     data[INPUT_IDX] = embed[char_to_id[gen_str[-1]]]
     stepOnce(data, output, buffers, indices, stream, context);
stream.synchronize()

     # Copy Ct/Ht to the Ct-1/Ht-1 slots.
     data[HIDDEN_IN_IDX] = data[HIDDEN_OUT_IDX]
data[CELL_IN_IDX] = data[CELL_OUT_IDX]

predIdx = output[0]
     gen_str += id_to_char[predIdx]
```

Chapter 8.
PERFORMING INFERENCE IN INT8 USING CUSTOM CALIBRATION

What Does This Sample Do?

This sample provides the steps involved when performing inference in 8-bit integer (INT8).

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

This sample demonstrates how to:

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

Where Is This Sample Located?

This sample is installed in the **`/usr/src/tensorrt/samples/sampleINT8`** directory.

Notes About This Sample:

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

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

```
./sample_int8 mnist
```

8.1. 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 TensorRT 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);
```

8.2. Building The Engine

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

8.2.1. Calibrating The Network

The application must specify the calibration set and parameters by implementing the IInt8Calibrator interface. Because calibration is an expensive process that may need to run multiple times, the interface provides methods for caching intermediate values. Follow this sample to learn more about how to configure a calibrator object.

8.2.2. Calibration Set

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.

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`**.

The calibration set 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.

8.2.3. Loading A 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 **`TensorRT-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. Specifically, to create the calibration file, you first need to provide a calibrator object and pass it to the builder. The calibrator object should be configured to use the calibration image batches. During the build phase, the builder will create the calibration file using the calibrator object.

After the calibration file is created, the file must get loaded. You cannot manually load a calibration file using an API, the builder first checks whether the file exists. If it does, it will not calibrate again and instead will load that same calibration file for every runtime. Therefore, the calibration file needs to be created only once.

8.3. Configuring The Builder

There are two additional methods to call on the builder:

```
builder->setInt8Mode(true);
builder->setInt8Calibrator(calibrator);
```

8.4. Running The Engine

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

8.5. 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%.

8.6. Batch Files For Calibration

The sampleINT8 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.

8.6.1. 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.

These 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`** from GitHub: BVLC Caffe. The patchfile 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
```

8.6.2. 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 command **`./sample_int8 mnist`**.

```
./sample_int8 mnist
```

Chapter 9. PERFORMING INFERENCE IN INT8 PRECISION

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

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

This sample demonstrates how to:

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

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

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

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

To print usage information:

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

To run INT8 inference with your dynamic ranges:

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

9.1. Configuring The Builder

Ensure that INT8 inference is supported on the platform:

```
if (!builder->platformHasFastInt8()) return false;
```

Enable INT8 mode by setting the builder flag:

```
builder->setInt8Mode(true);
builder->setInt8Calibrator(nullptr); // User can choose to not provide INT8
 calibrator. If user choose to provide the calibrator, manual dynamic range will
 override calibration generate dynamic range/scale.
```

Optionally, you can also force the layer precision using the following builder configuration:

```
builder->setStrictTypeConstraints(true);
```

This step is not required to perform INT8 inference. Enabling it will force INT8 precision for all the layers irrespective of performance. Therefore, it’s only recommended for debugging purposes.

9.2. Configuring The Network

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.
```

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));
```

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));
```

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

Iterate through the network to per layer precision:

```
auto layer = network->getLayer(i);
     layer->setPrecision(nvinfer1::DataType::kINT8);
     for (int j=0; j<layer->getNbOutputs(); ++j) {
         layer->setOutputType(j, nvinfer1::DataType::kINT8);
 }
```

Once the network is configured, build the engine and run inference as any other sample. For details regarding how to run the sample, see the README within the sample.

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

What Does This Sample Do?
This sample demonstrates how to add a Custom layer to TensorRT. This sample implements the MNIST model with the difference that the final FullyConnected layer is replaced by a Custom layer. To read more information about MNIST, see "Hello World" For TensorRT, Building A Simple OCR Network, and Import The TensorFlow Model And Run Inference.

This sample demonstrates how to:

- Define a Custom layer that supports multiple data formats
- Define a Custom layer that can be serialized and deserialized
- Enable a Custom layer in NvCaffeParser

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

Notes About This Sample:
The Custom layer implements the FullyConnected layer using *gemm* routines (Matrix Multiplication) in cuBLAS, and tensor addition in cuDNN (bias offset). This sample illustrates the definition of the **`FCPlugin`** for the Custom layer, and the integration with NvCaffeParser.

10.1. Defining The Network

The **`FCPlugin`** redefines the FullyConnected layer, which in this case 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);
}
```

10.2. Enabling Custom Layers In NvCaffeParser

The model is imported using NvCaffeParser (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 FullyConnected layer, a plugin factory is defined which recognizes the name of the FullyConnected 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();
}
```

10.3. 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.

10.4. Serializing And Deserializing

Fully complaint 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 (shows 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);
}
```

10.5. 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; }
```

Chapter 11. NEURAL MACHINE TRANSLATION (NMT) USING SEQUENCE TO SEQUENCE (SEQ2SEQ) MODELS

What Does This Sample Do?

This sample is a highly modular sample for inferencing using C++ and TensorRT API so that you can consider using it as a reference point in your projects. Neural Machine Translation (NMT) using sequence to sequence (seq2seq) models has garnered a lot of attention and is used in various NMT frameworks.

This sample demonstrates how to:

- Create an attention based seq2seq type NMT inference engine using a checkpoint from TensorFlow
- Convert trained weights using Python and import trained weights data into TensorRT
- Build relevant engines and run inference using the generated TensorRT network
- Use layers, such as:

  RNNv2

  The RNNv2 layer is used in the **`lstm_encoder.cpp`** and **`lstm_decoder.cpp`** files.

  Constant

  The Constant layer is used in the **`slp_attention.cpp`**, **`slp_embedder.cpp`** and **`slp_projection.cpp`** files.

  MatrixMultiply

  The MatrixMultiply layer is used in the **`context.cpp`**, **`multiplicative_alignment.cpp`**, **`slp_attention.cpp`**, and **`slp_projection.cpp`** files.

Shuffle
The Shuffle layer is used in the **`lstm_encoder.cpp`** and **`lstm_decoder.cpp`** files.

RaggedSoftmax
The RaggedSoftmax layer is used in the **`context.cpp`** file.

TopK
The TopK layer is used in the **`softmax_likelihood.cpp`** file.

Gather
The Gather layer is used in the **`slp_embedder.cpp`** file.

Where Is This Sample Located?

This sample is installed in the **`tensorrt/samples/sampleNMT`** directory. For more information about how to run the sample, see the **`README.txt`** file in the **`samples/sampleNMT/`** directory.

Notes About This Sample:

For more information about this sample, read the Neural Machine Translation Inference with TensorRT 4 technical blog.

11.1. Overview

At a high level, the basic architecture of the NMT model consists of two sides: an encoder and a decoder. Incoming sentences are translated into sequences of words in a fixed vocabulary. The incoming sequence goes through the encoder and is transformed by a network of Recurrent Neural Network (RNN) layers into an internal state space that represents a language-independent "meaning" of the sentence. The decoder works the opposite way, transforming from the internal state space back into a sequence of words in the output vocabulary.

Encoding And Embedding

The encoding process requires a fixed vocabulary of words from the source language. Words not appearing in the vocabulary are replaced with an **`UNKNOWN`** token. Special symbols also represent **`START-OF-SENTENCE`** and **`END-OF-SENTENCE`**. After the input is finished, a **`START-OF-SENTENCE`** is fed in to mark the switch to decoding. The decoder will then produce the **`END-OF-SENTENCE`** symbol to indicate it is finished translating.

Vocabulary words are not just represented as single numbers, they are encoded as word vectors of a fixed size. The mapping from vocabulary word to embedding vector is learned during training.

Attention

Attention mechanisms sit between the encoder and decoder and allow the network to focus on one part of the translation task at a time. It is possible to directly connect the encoding and decoding stages but this would mean the internal state representing the meaning of the sentence would have to cover sentences of all possible lengths at once.

This sample implements Luong attention. In this model, at each decoder step the target hidden state is combined with all source states using the attention weights. A scoring function weighs each contribution from the source states. The attention vector is then fed into the next decoder stage as an input.

Beam Search And Projection

There are several ways to organize the decode stage. The output of the RNN layer is not a single word. The simplest method, is to choose the most likely word at each time step, assume that is the correct output, and continue until the decoder generates the **`END-OF-SENTENCE`** symbol.

A better way to perform the decoding is to keep track of multiple candidate possibilities in parallel and keep updating the possibilities with the most likely sequences. In practice, a small fixed size of candidates works well. This method is called beam search. The beam width is the number of simultaneous candidate sequences that are in consideration at each time step.

As part of beam search we need a mechanism to convert output states into probability vectors over the vocabulary. This is accomplished with the projection layer using a fixed dense matrix.

For more information related to sampleNMT, see Creating A Network Definition In C++, Working With Deep Learning Frameworks, and Enabling FP16 Inference Using C++.

11.2. Preparing The Data

The NMT sample can be run with pre-trained weights. Link to the weights in the correct format can be found in the **`samples/sampleNMT/README.txt`** file.

Running the sample also requires text and vocabulary data. For the De-En model, the data can be fetched and processed using the script: wmt16_en_de.sh. Running this script may take some time, since it prepares 4.5M samples for training as well as inference.

Run the script **`wmt16_de_en.sh`** and collect the following files into a directory:

- **`newstest2015.tok.bpe.32000.de`**
- **`newstest2015.tok.bpe.32000.en`**
- **`vocab.bpe.32000.de`**
- **`vocab.bpe.32000.en`**

The weights **`.bin`** files from the link in the **`README.txt`** should be put in a subdirectory named **`weights`** in this directory.

In the event that the data files change, as of March 26, 2018 the MD5SUM for the data files are:

```
3c0a6e29d67b081a961febc6e9f53e4c  newstest2015.tok.bpe.32000.de

875215f2951b21a5140e4f3734b47d6c  newstest2015.tok.bpe.32000.en

c1d0ca6d4994c75574f28df7c9e8253f  vocab.bpe.32000.de

c1d0ca6d4994c75574f28df7c9e8253f  vocab.bpe.32000.en
```

11.3. Running The Sample

The sample executable is located in the **tensorrt/bin** directory. Running the sample requires pre-trained weights and the data files mentioned in Preparing The Data. After the data directory is setup, pass the location of the data directory to the sample with the following option:

```
--data_dir=<path_to_data_directory>
```

To generate example translation output, issue:

```
sample_nmt --data_dir=<path> --data_writer=text
```

The example translations can then be found in the **translation_output.txt** file.

To get the BLEU score for the first 100 sentences, issue:

```
sample_nmt --data_dir=<path> --max_inference_samples=100
```

The following options are available when running the sample:

--help
: Output help message and exit.

--data_writer=bleu/text/benchmark
: Type of the output the app generates (default = **bleu**).

--output_file=<path_to_file>
: Path to the output file when **data_writer=text**.

--batch=<N>
: Batch size (default = **128**).

--beam=<N>
: Beam width (default = **5**).

--max_input_sequence_length=<N>
: Maximum length for input sequences (default = **150**).

--max_output_sequence_length=<N>
: Maximum length for output sequences (default = **-1**), negative value indicates no limit.

--max_inference_samples=<N>
: Maximum sample count to run inference for, negative values indicates no limit is set (default = **-1**).

--verbose
: Output information level messages by TensorRT.

--max_workspace_size=<N>
Maximum workspace size (default = **268435456**).
--data_dir=<path_to_data_directory>
Path to the directory where data and weights are located (default = **../../../../data/samples/nmt/deen**).
--profile
Profile TensorRT execution layer by layer. Use benchmark **data_writer** when profiling on, disregard benchmark results.
--aggregate_profile
Merge profiles from multiple TensorRT engines.
--fp16
Switch on FP16 math.

11.4. Training The Model

Training the NMT model can be done in TensorFlow. This sample was trained following the general outline of the TensorFlow Neural Machine Translation Tutorial. The first step is to obtain training data, which is handled by the steps in Preparing The Data.

The next step is to fetch the TensorFlow NMT framework, for example:

```
git clone https://github.com/tensorflow/nmt.git
```

The model description is located in the **nmt/nmt/standard_hparams/wmt16.json** file. This file encodes values for all the hyperparameters available for NMT models. Not all variations are supported by the current NMT sample code so this file should be edited with appropriate values. For example, only unidirectional LSTMs and the Luong attention model are supported. The exact parameters used for the pre-trained weights are available in the sample **README.txt** file.

After the model description is ready and the training data is available in the **<path>/wmt16_de_en** directory, the command to train the model is:

```
python -m nmt.nmt \
--src=de --tgt=en \
--hparams_path=<path_to_json_config>/wmt16.json \
--out_dir=/tmp/deen_nmt \
--vocab_prefix=/tmp/wmt16_de_en/vocab.bpe.32000 \
--train_prefix=/tmp/wmt16_de_en/train.tok.clean.bpe.32000 \
--dev_prefix=/tmp/wmt16_de_en/newstest2013.tok.bpe.32000 \
--test_prefix=/tmp/wmt16_de_en/newstest2015.tok.bpe.32000
```

11.5. Importing Weights From A Checkpoint

Training the model generates various output files describing the state of the model. In order to use the model with TensorRT, model weights must be loaded into the TensorRT network. The weight values themselves are included in the TensorFlow checkpoint produced during training. In the sample directory, we provide a Python script that extracts the weights from a TensorFlow checkpoint into a set of binary weight files that can be directly loaded by the sample.

To use the script, run the command:

The **`chpt_to_bin.py`** script is located in the **`/usr/src/tensorrt/samples/sampleNMT`** directory.

```
python ./chpt_to_bin.py \
    --src=de --tgt=en \
    --ckpt=/tmp/deen_nmt/translate.ckpt-340000 \
    --hparams_path=<path_to_json_config>/wmt16.json \
    --out_dir=/tmp/deen \
    --vocab_prefix=<path>/wmt16_de_en/vocab.bpe.32000 \
    --inference_input_file=\
          <path>/wmt16_de_en/newstest2015.tok.bpe.32000.de \
    --inference_output_file=/tmp/deen/output_infer \
    --inference_ref_file=\
         <path>/wmt16_de_en/newstest2015.tok.bpe.32000.en
```

This generates 7 binary weight files for all the pieces of the model. The binary format is just a raw dump of the floating point values in order, followed by a metadata. The script was tested against TensorFlow 1.6.

Chapter 12. OBJECT DETECTION WITH FASTERRCNN

What Does This Sample Do?

This sample demonstrates how to:

- Uses TensorRT plugins which allow for end-to-end inferencing
- Implement the Faster R-CNN network in TensorRT
- Perform a quick performance test in TensorRT
- Implement a fused custom layer
- Construct the basis for further optimization, for example using INT8 calibration, user trained network, etc.

Where Is This Sample Located?

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

The Faster R-CNN Caffe model is too large to include in the product bundle. To run this sample, download the model using the instructions in the **`README.txt`** in the sample directory. The README is located in the **`<TensorRT directory>/samples/sampleFasterRCNN`** directory. Once the model is downloaded and extracted as per the instructions, the sample can be run by invoking **`sample_fasterRCNN`** binary.

Notes About This Sample:

The original Caffe model has been modified to include the Faster R-CNN's RPN and ROIPooling layers.

12.1. Overview

The sampleFasterRCNN is a more complex sample. The Faster R-CNN network is based on the paper Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.

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

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`**.

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).

12.2. Preprocessing The Input

The input to the Faster R-CNN network is 3 channel 375x500 images.

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 represented by a byte of integer (0-255) and they are stored together, pixel by pixel.

However, 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, we reverse the channel order when the PPM images are being put into the network 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.

12.3. 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`**.

Similar to Adding A Custom Layer To Your Network In TensorRT, in order to add Custom layers via NvCaffeParser, you need to create a factory by implementing the **`nvcaffeParser::IPluginFactory`** interface and then pass an instance to `ICaffeParser::parse()`. But unlike Adding A Custom Layer To Your Network In TensorRT, in which the **`FCPlugin`** is defined in the sample, the **`RPROIFused`** plugin layer instance can be created by the **`create`** function implemented in the TensorRT plugin library **`createRPNROIPlugin`**. This function returns an instance that implements an optimized **`RPROIFused`** Custom layer and performs the same logic designed by the authors.

12.4. Building The Engine

For details on how 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++.

12.5. Running The Engine

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

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

`data`
: **`data`** is the image input

`imInfo`
: **`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`
: **`bbox_pred`** is the predicted offsets to the heights, widths and center coordinates.

`cls_prob`
: **`cls_prob`** is the probability associated with each object class of every bounding box.

`rois`
: **`rois`** is the height, width, and the center coordinates for each bounding box.

`count`
: **`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.

12.6. 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.

Chapter 13. OBJECT DETECTION WITH A TENSORFLOW SSD NETWORK

What Does This Sample Do?

This sample demonstrates how to:

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

Where Is This Sample Located?

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

Notes About This Sample:

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

13.1. API Overview

The sampleUffSSD is based on the following paper, SSD: Single Shot MultiBox Detector. 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.

The sampleUffSSD is based on the TensorFlow implementation of SSD. For more information, see 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 configuration details of the network can be found at GitHub: TensorFlow models.

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.

13.2. 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 scripts 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 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 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.

13.3. 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 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 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 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**.

13.4. Plugins Used

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.

Alternately, the older flow of using the **`IPluginFactory`** can also be used. In this case, the **`pluginFactory`** object created needs to be passed to an instance of **`IUffParser::parse()`** which will invoke the **`createPlugin()`** function for each Custom layer which has to be implemented by the user. 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 of the order of 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.

13.5. Verifying The Output

After the builder is created (see Building An Engine In C++) and the engine is serialized (see Serializing A Model In C++), we can perform inference. Steps for deserialization and running inference are outlined in Performing Inference In C++.

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.

Chapter 14. MOVIE RECOMMENDATION USING NEURAL COLLABORATIVE FILTER (NCF)

What Does This Sample Do?
This sample demonstrates a simple movie recommender system using Neural Collaborative Filter (NCF). The network is trained in TensorFlow on the MovieLens dataset containing 6040 users and 3706 movies. For more information about the recommender system network, see Neural Collaborative Filtering.

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

Notes About This Sample:
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 Multi-Layer Perceptrons (MLPs).

14.1. Importing 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 FullyConnected layers. A TopK operation is added manually after parsing to find the highest rated movie for the given user.

14.2. Verifying The Output

The output of the MLP based NCF network is in human readable format. The final output is **`movieID`** with probability rating for give **`userID`**.

Chapter 15. MOVIE RECOMMENDATION USING MPS (MULTI-PROCESS SERVICE)

What Does This Sample Do?

This sample is identical to the Movie Recommendation Using Neural Collaborative Filter (NCF) sample in terms of functionality, but is modified to support concurrent execution in multiple processes.

Where Is This Sample Located?

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

Notes About This Sample:

MPS (Multi-Process Service) 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. For more information about MPS, see Multi-Process Service documentation or in the **`README.txt`** file for the sample.

MPS requires a server process. To start the process:

```
export CUDA_VISIBLE_DEVICES=<GPU_ID>
nvidia-smi -i <GPU_ID> -c EXCLUSIVE_PROCESS
nvidia-cuda-mps-control -d
```

Chapter 16. OBJECT DETECTION WITH SSD

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

- Preprocess the input to the SSD network
- Perform inference on the SSD network in TensorRT
- Use TensorRT plugins to speed up inference
- Perform INT8 calibration on an SSD network

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

Notes About This Sample:
The SSD Caffe model is too large to include in the product bundle. To run this sample, download the model using the instructions in the README.md in the sample **`<TensorRT directory>/samples/sampleSSD`** directory. The original Caffe model (prototxt) has been modified to include the SSD's customized Plugin layers.

16.1. Overview

The SSD network is based on the following paper SSD: Single Shot MultiBox Detector. This network is based on the VGG-16 network. It can perform object detection and localization in a single forward pass.

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.

16.2. Preprocessing 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 **#**.

16.3. 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.

16.4. 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 which avoids creating additional code for plugin factory like in the sampleFasterRCNN sample.

This sample can run in FP16 and INT8 modes based on the user input. For more details, seeINT8 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, seeBuilding 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, seeSerializing A Model In C++.

16.5. 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, and 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.

Chapter 17. "HELLO WORLD" FOR MULTI-LAYER PERCEPTRON (MLP)

What Does This Sample Do?
This sample is a simple hello world example that shows how to create a network that triggers the multi-layer perceptron (MLP) optimizer. The 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.

This sample demonstrates how to:

- Trigger the MLP optimizer by creating a sequence of networks to increase performance
- Create a sequence of TensorRT layers that represent an MLP layer

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

17.1. Defining The Network

This sample follows the same flow as Building A Simple OCR Network with one exception. The network is defined as a sequence of **`addMLP`** calls, which adds FullyConnected and Activation layers to the network.

Currently, an MLP layer is defined as a FullyConnected or MatrixMultiplication operation with optional bias and activations. A 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.

The current variations that trigger the MLP optimizer:

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

Chapter 18. INTRODUCTION TO IMPORTING CAFFE, TENSORFLOW AND ONNX MODELS INTO TENSORRT USING PYTHON

What Does This Sample Do?

This sample demonstrates how to use 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. TensorRT uses a suite of parsers to generate TensorRT networks from models trained in different frameworks.

This sample includes the following:

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. It can be used with **`.onnx`** files.

uff_resnet50

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

Where Is This Sample Located?

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

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

Chapter 19. "HELLO WORLD" FOR TENSORRT USING TENSORFLOW AND PYTHON

What Does This Sample Do?

This sample demonstrates how to first train a model using TensorFlow and Keras, freeze the model and write it to a protobuf file, convert it to UFF, and finally run 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.

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

Chapter 20. "HELLO WORLD" FOR TENSORRT USING PYTORCH AND PYTHON

What Does This Sample Do?

This sample demonstrates how to train a model in PyTorch, recreate the network in TensorRT and import weights from the trained model, and finally run 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.

Notes About This Sample:

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.

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

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

What Does This Sample Do?
This sample demonstrates how to use plugins written in C++ with the TensorRT Python bindings and CaffeParser. More specifically, this sample implements a FullyConnected layer using cuBLAS and cuDNN, wraps the implementation in a TensorRT plugin (with a corresponding plugin factory) and then generates Python bindings for it using **`pybind11`**. These bindings are then used to register the plugin factory with the CaffeParser.

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

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

Chapter 22. ADDING A CUSTOM LAYER TO YOUR TENSORFLOW NETWORK IN TENSORRT IN PYTHON

What Does This Sample Do?

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

Where Is This Sample Located?

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

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

Chapter 23. OBJECT DETECTION WITH THE ONNX TENSORRT BACKEND IN PYTHON

What Does This Sample Do?

This sample demonstrates a full ONNX-based pipeline for inference with the network YOLOv3-608, including pre- and post-processing.

First, the YOLOv3 configuration and the weights from the author's official mirror are read to generate an ONNX representation of the model that can be parsed by TensorRT. Thereafter, that ONNX graph is used to create a TensorRT engine with the open-sourced repository.

Next, the YOLOv3 pre-processing steps are applied on an example image and used as an input to the previously created engine.

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.

Where Is This Sample Located?

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

Notes About This Sample:

This sample requires the installation of ONNX-TensorRT: TensorRT backend for ONNX, which includes layer implementations for the required ONNX operators **`Upsample`** and **`LeakyReLU`**.

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

Chapter 24. OBJECT DETECTION WITH SSD IN PYTHON

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

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

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

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

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

24.1. Overview

The uff_ssd is based on the following paper, SSD: Single Shot MultiBox Detector. 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.

The sample is based on the TensorFlow implementation of SSD. For more information, see 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 configuration details of the network can be found at GitHub: TensorFlow models.

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.

24.2. 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.

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.

Additional documentation on GraphSurgeon can be found in the TensorRT API.

24.3. Plugins Used

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

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 of the order of 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.

Chapter 25. INT8 CALIBRATION IN PYTHON

What Does This Sample Do?

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

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

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

Where Is This Sample Located?

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

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

Chapter 26. ENGINE REFIT IN PYTHON

What Does This Sample Do?

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

Where Is This Sample Located?

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

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