TENSORRT

DU-08540-021_v01 | July 2017

User Guide

TABLE OF CONTENTS

Chapter 1. OVERVIEW

NVIDIA® TensorRT™ is a C++ library that facilitates high performance inference on NVIDIA GPUs. TensorRT takes a network definition and optimizes it by merging tensors and layers, transforming weights, choosing efficient intermediate data formats, and selecting from a large kernel catalog based on layer parameters and measured performance.

The TensorRT API consists of import methods to help you express your trained deep learning models for TensorRT to optimize and run. It is an optimization tool that applies graph optimization and layer fusion and finds the fastest implementation of that model leveraging a diverse collection of highly optimized kernels, and a runtime that you can use to execute this network in an inference context.

TensorRT includes a full infrastructure that allows you to leverage high speed reduced precision capabilities of Pascal™ GPUs as an optional optimization.

TensorRT is built with gcc 4.8.

1.1. TensorRT Layers

TensorRT directly supports the following layer types:

Activation
The Activation layer implements per-element activation functions. Supported activation types are ReLU, TanH and Sigmoid.

Concatenation
The concatenation layer links together multiple tensors of the same height and width across the channel dimension.

Convolution
The Convolution layer computes a 3D (channel, height, width) convolution, with or without bias.

Deconvolution
The Deconvolution layer implements a deconvolution, with or without bias.

ElementWise
The ElementWise, also known as Eltwise, layer implements per-element operations. Supported operations are **`sum`**, **`product`**, and **`maximum`**.

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

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

Plugin
The Plugin Layer allows you to integrate layer implementations that TensorRT does not natively support.

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

RNN
The RNN layer implements recurrent layers. Supported types are simple **`RNN`**, **`GRU`**, and **`LSTM`**.

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

SoftMax
The SoftMax layer implements a cross-channel SoftMax.

- Batch Normalization can be implemented using the TensorRT Scale layer.
- The operation the Convolution layer performs is actually a correlation. Therefore it is a consideration if you are formatting weights to import via TensorRT's API, rather than via the Caffe parser library.

1.2. NvCaffeParser

While TensorRT is independent of any framework, the package does include a parser for Caffe models named NvCaffeParser.

NvCaffeParser provides a simple mechanism for importing network definitions. NvCaffeParser uses TensorRT's layers to implement Caffe's Convolution, ReLU, Sigmoid, TanH, Pooling, Power, BatchNorm, ElementWise (Eltwise), LRN, InnerProduct (which is what Caffe calls the FullyConnected layer), SoftMax, Scale, and Deconvolution layers.

Caffe features not currently supported by TensorRT include:

- Deconvolution groups
- Dilated convolutions
- PReLU
- Leaky ReLU
- Scale, other than per-channel scaling
- ElementWise (Eltwise) with more than two inputs

NvCaffeParser does not support legacy formats in Caffe prototxt; in particular, layer types are expected to be expressed in the prototxt as strings delimited by double quotes.

Chapter 2. TENSORRT WORKFLOW

2.1. Key Concepts

Ensure you are familiar with the following key concepts:

Network definition

A network definition consists of a sequence of layers and a set of tensors.

Layer

Each layer computes a set of output tensors from a set of input tensors. Layers have parameters, for example, **`convolution size`**, **`stride`**, and **`convolution filter weights`**.

Tensor

A tensor is either an input to the network, or an output of a layer. Tensors have a data-type specifying their precision, for example, 16- and 32-bit floats, and three dimensions, for example, channels, width, and height. The dimensions of an input tensor are defined by the application, and for output tensors they are inferred by the builder.

Each layer and tensor has a name, which is useful when profiling or reading TensorRT's build log. For more information, see Logging.

When using NvCaffeParser, tensor and layer names are taken from the Caffe prototxt file.

2.2. TensorRT Workflow Diagrams

Figure 1 shows a typical development workflow, where the user trains the model on data to produce a trained network. That trained network can then be used for inference.

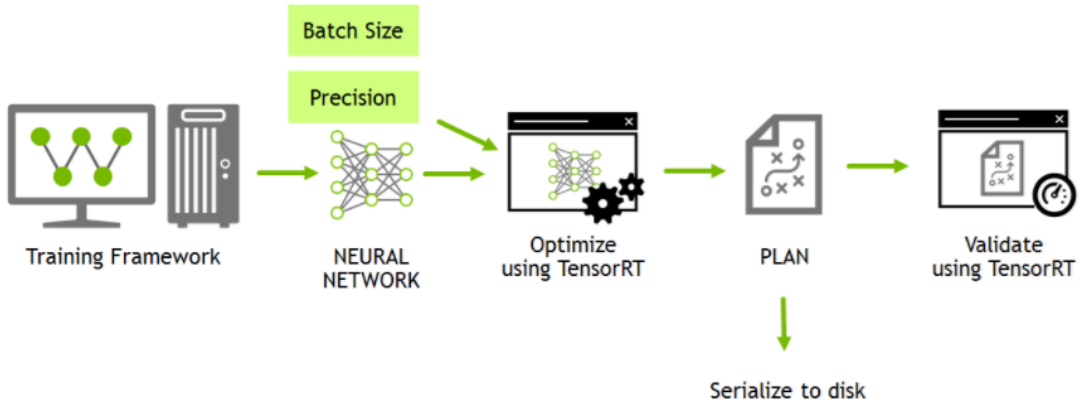


Figure 1 Typical development workflow.

Figure 1 is importing the trained network into TensorRT. The user imports the trained network into TensorRT, which optimizes the network to produce a PLAN. That PLAN is then used for inference, for example, to validate that optimization has been performed correctly.

The PLAN can also be serialized to disk so that it can be later reloaded into the TensorRT runtime without having to perform the optimization step again (see Figure 2).

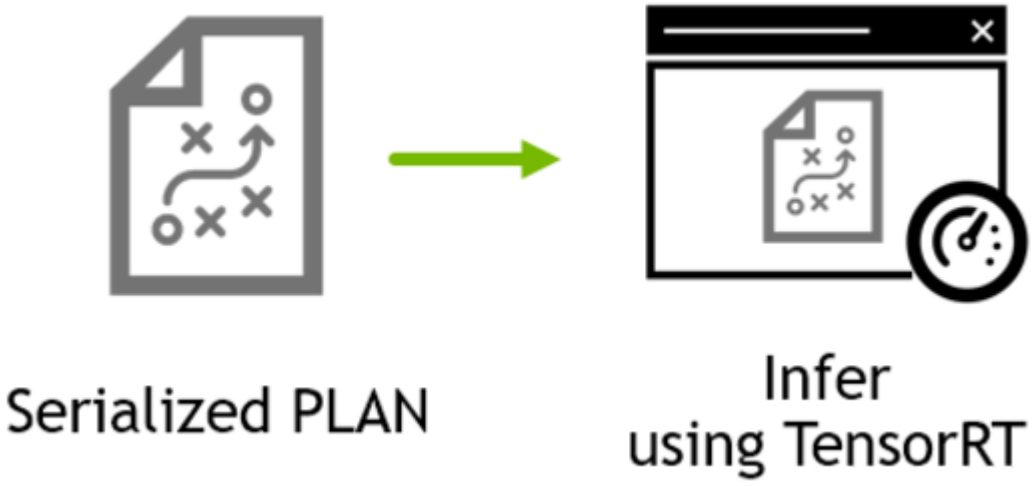


Figure 2 Typical production workflow.

2.3. Build Phase

In the *build phase*, the toolkit takes a network definition, performs optimizations, and generates the inference engine.

The build phase can take considerable time, especially when running on embedded platforms. Therefore, a typical application will build an engine once, and then serialize it for later use.

The build phase performs the following optimizations on the layer graph:

- elimination of layers whose outputs are not used
- fusion of convolution, bias and ReLU operations
- aggregation of operations with sufficiently similar parameters and the same source tensor (for example, the 1x1 convolutions in GoogleNet v5's inception module)
- elision of concatenation layers by directing layer outputs to the correct eventual destination

In addition, the build phase also runs layers on dummy data to select the fastest from its kernel catalog, and performs weight pre-formatting and memory optimization where appropriate.

2.4. Execution Phase

In the *execution phase*, the following tasks are run:

- The runtime executes the optimized engine.
- The engine runs inference tasks using input and output buffers on the GPU.

2.5. Command Line Wrapper

Included in the samples directory is a command line wrapper, called *giexec*, for TensorRT. It is useful for benchmarking networks on random data and for generating serialized engines from such models.

The command line arguments are as follows:

```
Mandatory params:
  --deploy=<file>      Caffe deploy file
  --output=<name>      Output blob name (can be specified
 multiple times)

Optional params:
  --model=<file>       Caffe model file (default = no model,
 random weights
       used)
  --batch=N            Set batch size (default = 1)
  --device=N           Set cuda device to N (default = 0)
  --iterations=N       Run N iterations (default = 10)
  --avgRuns=N          Set avgRuns to N - perf is measured as an
 average of
       avgRuns (default=10)
  --workspace=N        Set workspace size in megabytes (default =
 16)
  --half2              Run in paired fp16 mode (default = false)
  --int8               Run in int8 mode (default = false)
```

```
  --verbose             Use verbose logging (default = false)
  --hostTime            Measure host time rather than GPU time
 (default =
        false)
  --engine=<file>       Generate a serialized GIE engine
  --calib=<file>        Read INT8 calibration cache file
```

For example:

```
giexec --deploy=mnist.prototxt --model=mnist.caffemodel --
output=prob
```

If no model is supplied, random weights are generated.

Chapter 3. USING TENSORRT 2.1

The following samples show how to use TensorRT in numerous contexts. The samples demonstrate the different capabilities of the interface. Each sample includes the build phase and an execution phase. The samples begin with a simple and basic usage example and continues to the more complex examples.

The samples are listed sequentially, therefore, you need to implement each sample in the following order:

1. MNISTAPI depends on MNISTGoogleNet
2. MNISTGoogleNet depends on MNISTCharRNN
3. MNISTCharRNN depends on MNISTAPIINT8
4. MNISTAPIINT8 depends on MNISTPlugin
5. MNISTPlugin depends on MNISTFasterRCNN
6. MNISTFasterRCNN depends on Plugin

3.1. Sample 1: SampleMNIST Simple Usage

The SampleMNIST example demonstrates a typical build and execution phase using a Caffe model that is trained on the MNIST dataset using the NVIDIA DIGITS tutorial.

3.1.1. Logging

TensorRT requires a logging interface to be implemented, through which it reports errors, warnings, and informational messages. The following code shows how to suppress informational messages, and report only warnings and errors.

```
class Logger : public ILogger
{
     void log(Severity severity, const char* msg) override
{
          // suppress info-level messages
          if (severity != Severity::kINFO)
              std::cout << msg << std::endl;
     }
```

```
} gLogger;
```

3.1.2. The Build Phase - caffeToGIEModel

Before creating the TensorRT builder, the application must implement a logging interface, through which TensorRT will provide information about optimization stages during the build phase, and also warnings and error information. The following code creates the builder:

```
IBuilder* builder = createInferBuilder(gLogger);
```

Next, create the network definition structure. You will populate from a Caffe model using the Caffe parser library:

```
INetworkDefinition* network = builder->createNetwork();
CaffeParser* parser = createCaffeParser();
std::unordered_map<std::string, infer1::Tensor>
 blobNameToTensor;
const IBlobNameToTensor* blobNameToTensor =
      parser->parse(locateFile(deployFile).c_str(),
                                locateFile(modelFile).c_str(),
                                *network,
                                DataType::kFLOAT);
```

In this sample, the parser is instructed to generate a network whose weights are 32-bit floats. As well as populating the network definition, the parser returns a dictionary that maps from Caffe blob names to TensorRT tensors.

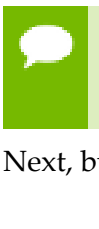
A TensorRT network definition has no notion of in-place operation, for example, the input and output tensors of a ReLU are different. When a Caffe network uses an in-place operation, the TensorRT tensor returned in the dictionary corresponds to the last write to that blob. For example, if a convolution creates a blob and is followed by an in-place ReLU, that blob's name will map to the TensorRT tensor which is the output of the ReLU.

Since the Caffe model does not tell us which tensors are the outputs of the network, we need to specify these explicitly after parsing:

```
for (auto& s : outputs)
     network->markOutput(*blobNameToTensor->find(s.c_str()));
```

There is no restriction on the number of output tensors, but marking a tensor as an output may prohibit some optimizations on that tensor.

At this point, we have parsed the Caffe model to obtain the network definition, and can now create the engine.



Do not release the parser object because the network definition holds weights by reference into the Caffe model, not by value. It is only during the build process that the weights are read from the Caffe model.

Next, build the engine from the network definition:

```
builder->setMaxBatchSize(maxBatchSize);
builder->setMaxWorkspaceSize(1 << 20);
ICudaEngine* engine = builder->buildCudaEngine(*network);
```

where:

- **maxBatchSize** is the size for which the engine will be tuned. At execution time, smaller batches may be used, but not larger.

The execution of smaller batch sizes may be slower than with a TensorRT engine optimized for that size.

- **maxWorkspaceSize** is the maximum amount of scratch space which the engine may use at runtime.

With the following code, the engine is serialized to a memory block, which you could then serialize to a file or stream:

```
gieModelStream = engine->serialize();
```

3.1.3. Deserializing the Engine

To deserialize the engine, create a TensorRT runtime object:

```
IRuntime* runtime = createInferRuntime(gLogger);
ICudaEngine* engine =
runtime->deserializeCudaEngine(gieModelStream->data(),
gieModelStream->size(), nullptr);
```

Next, create an execution context. One engine can support multiple contexts, allowing inference to be performed on multiple batches simultaneously while sharing the same weights.

```
IExecutionContext *context = engine->createExecutionContext();
```

Serialized engines are not portable across platforms or TensorRT versions.

3.1.4. The Execution Phase - doInference()

The input to the engine is an array of pointers to input and output buffers on the GPU.

All TensorRT inputs and outputs are in contiguous NCHW format.

The engine can be queried for the buffer indices, using the tensor names assigned when the network was created.

```
int inputIndex = engine->getBindingIndex(INPUT_BLOB_NAME),
    outputIndex = engine->getBindingIndex(OUTPUT_BLOB_NAME);
```

In a typical production case, TensorRT will execute asynchronously. The **enqueue()** method will add kernels to a CUDA stream specified by the application, which may then

wait on that stream for completion. The fourth parameter to **`enqueue()`** is an optional cudaEvent which will be signaled when the input buffers are no longer in use and can be refilled.

The following sample code shows the input buffer being copied to the GPU, running inference, then copying the result back and waiting on the stream:

```
cudaMemcpyAsync(<...>, cudaMemcpyHostToDevice, stream);
context.enqueue(batchSize, buffers, stream, nullptr);
cudaMemcpyAsync(<...>, cudaMemcpyDeviceToHost, stream);
cudaStreamSynchronize(stream);
```

The batch size must be at most the value specified when the engine was created.

3.2. Sample 2: SampleMNISTAPI API Usage

The SampleMNISTAPI example demonstrates how to use the API in order to produce the same network as SampleMNIST but without using NvCaffeParser. This sample showcases how to target TensorRT from another framework or application other than Caffe.

3.2.1. Setting the Input

All networks must specify an input; as the input is the entry point to the network. You must provide a name for the input.

```
INetworkDefinition* network = builder->createNetwork();
//  Create input of shape { 1, 1, 28, 28 } with name referenced
 by INPUT_BLOB_NAME
auto data = network->addInput(INPUT_BLOB_NAME, dt, DimsCHW{ 1,
 INPUT_H, INPUT_W});
```

3.2.2. Creating a Layer

You can create multiple layers directly from the TensorRT API.

In the following code, both **`power`** and **`shift`** are using the default values for their weights and the scale parameter is being provided to the layer. The scaling mode is uniform scaling.

The following code is an example of the creation of a single scale layer:

```
// Create a scale layer with default power/shift and specified
 scale
parameter.
float scale_param = 0.0125f;
Weights power{DataType::kFLOAT, nullptr, 0};
Weights shift{DataType::kFLOAT, nullptr, 0};
Weights scale{DataType::kFLOAT, &scale_param, 1};
auto scale_1 = network->addScale(*data, ScaleMode::kUNIFORM,
 shift, scale, power);
```

3.2.3. Allocating Weights

In this sample, since Caffe is not generating the weights automatically, you must allocate and manage the weight memory, which is stored in the **`weightMap`** and read from the filesystem.

```
std::map<std::string, Weights> weightMap =
      loadWeights(locateFile("mnistapi.wts"));
```

3.2.4. Setting the Output

The network must know which layers set which outputs.

It is recommended to name the outputs.

If a name is not provided, TensorRT will generate a name.

```
// Add a softmax layer to determine the probability.
auto prob = network->addSoftMax(*ip2->getOutput(0));
prob->getOutput(0)->setName(OUTPUT_BLOB_NAME);
network->markOutput(*prob->getOutput(0));
```

3.2.5. Free Memory

Memory needs to be made available to the builder until after the engine is created. In this sample, the memory for weights are stored in a map after being loaded from the filesystem. After the engine has been created and the network has been destroyed, it is safe to deallocate memory.

Deallocating memory before creating the engine has undefined behavior.

3.3. Sample 3: SampleGoogleNet - Profiling and 16-bit Inference

The SampleGoogleNet example demonstrates the layer-based profiling, and TensorRT's half2 mode, which runs the network in 16-bit floating point precision.

3.3.1. Profiling

To profile a network, implement the **`IProfiler`** interface and add the profiler to the execution context:

```
context.profiler = &gProfiler;
```

Profiling is not currently supported for asynchronous execution, therefore, use TensorRT's synchronous execute() method:

```
for (int i = 0; i < TIMING_ITERATIONS;i++)
     engine->execute(context, buffers);
```

After execution has completed, the profiler callback is called once for every layer. The sample accumulates layer times over invocations, and averages the time for each layer at the end.

The layer names are modified by TensorRT's layer-combining operations.

3.3.2. `Half2Mode`

TensorRT can use 16-bit instead of 32-bit arithmetic and tensors, but this alone may not deliver significant performance benefits. **`Half2Mode`** is an execution mode where internal tensors interleave 16-bits from adjacent pairs of images, and is the fastest mode of operation for batch sizes greater than one.

To use **`Half2Mode`**, two additional steps are required:

1. Create an input network with 16-bit weights, by supplying the DataType::kHALF2 parameter to the parser. For example:

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

2. Configure the builder to use **`Half2Mode`**.

   ```
   builder->setHalf2Mode(true);
   ```

3.4. Sample 4: SampleCharRNN - RNNs and Converting Weights from TensorFlow to TensorRT

The SampleCharRNN example demonstrates how to generate a simple RNN based on the charRNN network using the PTB dataset.

RNN layers are like any other TensorRT layer. Each RNN has three output tensors and up to three input tensors. For more information, see the *TensorRT API* documentation.

3.4.1. Weight Conversion

TensorFlow weights are exported with each layer concatenated into a single WTS file. The file format is defined by the **`loadWeights`** function. The weights that were previously loaded by **`loadWeights()`** are now converted into an the format required by TensorRT. The memory holding the converted weights is added to the weight map so that it can be deallocated once the engine has been built.

```
// Create an RNN layer w/ 2 layers and 512 hidden states
```

```
auto tfwts = weightMap["rnnweight"];
Weights rnnwts{convertRNNWeights(tfwts)};
auto tfbias = weightMap["rnnbias"];
Weights rnnbias{convertRNNBias(tfbias)};
...
weightMap["rnnweight2"] = rnnwts;
weightMap["rnnbias2"] = rnnbias;
```

3.4.2. Layer Generation

After the RNN weights are converted, the next step is to create the RNN layer. There are multiple different RNN types and modes that are supported. This specific RNN is a single directional LSTM layer where the input is transformed to match the same size as the hidden weight matrix.

```
auto rnn = network->addRNN(*data, LAYER_COUNT, HIDDEN_SIZE,
 SEQ_SIZE,
        RNNOperation::kLSTM, RNNInputMode::kLINEAR,
 RNNDirection::kUNIDIRECTION,
        rnnwts, rnnbias);
```

3.4.3. Optional Inputs

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

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

3.4.4. Marking the Resulting Output

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.

```
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));
}
```

3.4.5. Reshaping Data to Fit the Format of the Next Layer

The output of an RNN is optimized to feed into another RNN layer as efficiently as possible. When out-puting to another layer that has a different layer requirement, a reshaping is required.

The reshape parameter uses the plugin API and converts the layer to the format required for the FullyConnected layer. In this case we are reshaping the **`{ T, N, C }`** to **`{N * T, C, 1, 1}`** so that it can be fed properly into the FullyConnected layer.

```
Reshape reshape(SEQ_SIZE * BATCH_SIZE * HIDDEN_SIZE);
ITensor *ptr = rnn->getOutput(0);
auto plugin = network->addPlugin(&ptr, 1, reshape);
plugin->setName("reshape");

auto fc = network->addFullyConnected(*plugin->getOutput(0),
OUTPUT_SIZE, wts, bias);
```

TensorRT network inputs and outputs are 32-bit tensors in contiguous **`NCHW`** format. For weights:

- Convolution weights are in contiguous **`KCRS`** format, where **`K`** indexes over output channels, **`C`** over input channels, and **`R`** and **`S`** over the height and width of the convolution, respectively.
- FullyConnected weights are in contiguous row-major layout.
- Deconvolution weights are in contiguous **`CKRS`** format; where **`C`**, **`K`**, **`R`** and **`S`** are the same as convolution weights.

3.4.6. 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.

```
for (auto &a : input)
{
     std::copy(reinterpret_cast<const float*>(embed.values) +
char_to_id[a]*DATA_SIZE,
               reinterpret_cast<const float*>(embed.values) +
char_to_id[a]*DATA_SIZE + DATA_SIZE,
               data[0]);
     stepOnce(data, buffers, sizes, indices, 6, stream, context);
     cudaStreamSynchronize(stream);
     genstr.push_back(a); }
```

3.4.7. 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 highest probability. The final result is stored in `genstr`.

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

```
    stepOnce(data, buffers, sizes, indices, 6, stream, context);
    cudaStreamSynchronize(stream);

    float* probabilities =
 reinterpret_cast<float*>(data[indices[3]]);
    int idx = std::max_element(probabilities, probabilities +
 sizes[3]) -
probabilities;
    genstr.push_back(id_to_char[idx]);
}
```

3.5. Sample 5: SampleINT8 - Calibration and 8-bit Inference

The SampleINT8 example provides the steps involved when performing inference in 8-bit integer (INT8). The 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
```

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

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 calibrate the network to determine how best to represent the weights and activations as 8-bit integers.

The application must specify the calibration set and parameters by implementing the IInt8Calibrator interface. For ImageNet networks and MNIST, 500 images is a reasonable size for the calibration set.

3.5.1. IInt8EntropyCalibrator Interface

The IInt8EntropyCalibrator interface has methods for specifying the calibration set and calibration parameters to the builder.

In addition, because calibration is an expensive process that may need to run multiple times, it provides methods for caching intermediate values. The simplest implementation is to return immediately from the **`write()`** methods, and return nullptr from the **`read()`** methods.

3.5.1.1. Calibration Set

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

```
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. In addition, any image processing, such as, scaling, cropping or mean subtraction, that would occur prior to inference must also be performed prior to calibration.

3.5.2. Configuring the Builder

For INT8 inference, the input model must be specified with 32-bit weights:

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

There are two additional methods to call on the builder:

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

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.

3.5.3. Calibration Caching

Calibration can be slow, therefore, the IInt8Calibrator interface provides methods for caching intermediate data. Using these methods effectively requires a more detailed understanding of calibration.

When building an INT8 engine, the builder performs the following steps:

1. Builds a 32-bit engine, runs it on the calibration set, and records a histogram for each tensor of the distribution of activation values.
2. Builds a calibration table from the histograms.

3. Builds the INT8 engine from the calibration table and the network definition.

The calibration table can be cached. Caching is useful when building the same network multiple times, for example, on multiple platforms. It captures data derived from the network and the calibration set. The parameters are recorded in the table. If the network or calibration set changes, it is the application's responsibility to invalidate the cache.

The cache is used as follows:

- if a calibration table is found, calibration is skipped, otherwise:
  - then the calibration table is built from the histograms and parameters
- then the INT8 network is built from the network definition and the calibration table.

Cached data is passed as a pointer and length.

3.5.4. Batch Files for Calibration

The SampleINT8 example uses batch files in order to calibrate for the INT8 data. The INT8 batch file is a binary file defined as follows:

- Four 32-bit integer values representing **{N,C, H, W}** dimensions of the data set.
- There are **N** 32-bit floating point data blobs of dimensions **{C, H, W}** that are used as inputs to the network.
- There are **N** 32-bit integer labels that correspond to the **N** input blobs.

3.5.4.1. Generating Batch Files for Caffe Users

For developers that use Caffe for their training, or can easily transfer their network to Caffe, generating the calibration data is done through a supplied patchset.

These instructions are for Caffe git commit **473f143f9422e7fc66e9590da6b2a1bb88e50b2f**. The patchfile might be slightly different for later versions of Caffe. 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
```

After the patch is applied, Caffe needs to be rebuilt and the environment variable *TENSORRT_INT8_BATCH_DIRECTORY* needs to be set 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.

First, go 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 the Caffe at the specified 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
```

After the build has finished, download the `mnist` data set from Caffe and create the link to it.

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

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 .
```

SampleINT8 can now be executed from the `bin` directory after being built with the command `./sample_int8 mnist`.

3.5.4.2. 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 data set.
2. Crop all of the input data to the same dimensions.
3. Split the data into **`N`** batch files where each batch file has **`M`** sets of input data and **`M`** sets of 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.

First, go 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 .
```

Copy the generated batch files to `int8/mnist/batches/`.

SampleINT8 can now be executed from the `bin` directory after being built with the command `./sample_int8 mnist`.

3.6. Sample 6: SamplePlugin - Implementing a Custom Layer

The SamplePlugin example demonstrates how to add a custom layer to TensorRT. It replaces the final fully connected layer of the MNIST sample with a direct call to cuBLAS.

There are two steps to adding a custom layer:

1. Create a plugin conforming to the IPlugin interface.
2. Add the plugin to the network.

The IPlugin interface methods fall into the following categories:

- Determining the Outputs
- Layer Configuration
- Workspace
- Resource Management
- Execution
- Serialization

3.6.1. Determining the Outputs

When defining the network, TensorRT needs to know which outputs the layer has.

The dimensions given in the sample are without the batch size, in a similar way to dimensions returned by `ITensor::getDimensions()`. For example, for a typical 3-dimensional convolution, the dimensions provided are given in **`{C, H, W}`** form, and the return value should also be in **`{C, H, W}`** form.

The following methods provide which outputs the layer has:

```
int getNbOutputs() const override
{
     return 1;
}

Dims getOutputDimensions(int index, const Dims* inputs, int
nbInputDims) override
{
```

```
     assert(index == 0 && nbInputDims == 1 && inputs[0].nbDims ==
 3);
     assert(mNbInputChannels == inputs[0].d[0] * inputs[0].d[1] *
inputs[0].d[2]);
     return DimsCHW(mNbOutputChannels, 1, 1);
}
```

3.6.2. Layer Configuration

The builder calls the network's `configure()` method, to give it a chance to select an algorithm based on its inputs. In this example, the inputs are checked to have the correct form. In a more complex example, you might choose a convolution algorithm based on the input dimensions.

The `configure()` method is only called at build time, therefore, anything determined here that is required at runtime should be stored as a member variable of the plugin, and serialized and/or de-serialized.

3.6.3. Workspace

TensorRT can provide workspace for temporary storage during layer execution, which is shared among layers in order to minimize memory usage. The TensorRT builder calls `getWorkspaceSize()` in order to determine the workspace requirement. In this example, no workspace is used. If workspace is requested, it will be allocated when an `IExecutionContext` is created, and passed to the `enqueue()` method at runtime.

3.6.4. Resource Management

The `initialize()` and `terminate()` methods are called by the runtime when an **`IExecutionContext`** is created and destroyed, so that the layer can allocate resources.

In the following sample, handles are created for cuDNN, cuBLAS, and some cuDNN tensor descriptors for the bias addition operation.

```
int initialize() override
{
     CHECK(cudnnCreate(&mCudnn));
     CHECK(cublasCreate(&mCublas));
     CHECK(cudnnCreateTensorDescriptor(&mSrcDescriptor));
     CHECK(cudnnCreateTensorDescriptor(&mDstDescriptor));

     return 0;
}

virtual void terminate() override
{
     CHECK(cublasDestroy(mCublas));
     CHECK(cudnnDestroy(mCudnn));
}
```

3.6.5. Execution

The `enqueue()` method is used to execute the layer's runtime implementation.

The batch size passed to `enqueue()` is at most the maximum batch size specified at build time, although it can be smaller.

Except for batch size, dimensional information is not passed to `enqueue()`. Therefore, other dimensional information required at runtime, for example, the number of input and output channels, should be serialized as part of the layer data.

```
virtual int enqueue(int batchSize, const void*const * inputs,
 void**
outputs, void* workspace, cudaStream_t stream) override
{
     int nbOutputChannels = mBiasWeights.count;
     int nbInputChannels = mKernelWeights.count /
 nbOutputChannels;
     float kONE = 1.0f, kZERO = 0.0f;
     cublasSetStream(mCublas, stream);
     cudnnSetStream(mCudnn, stream);
     CHECK(cublasSgemm(mCublas, CUBLAS_OP_T, CUBLAS_OP_N,
nbOutputChannels, batchSize, nbInputChannels, &kONE,
                reinterpret_cast<const
 float*>(mKernelWeights.values),
nbInputChannels,
                reinterpret_cast<const float*>(inputs[0]),
nbInputChannels, &kZERO,
                reinterpret_cast<float*>(outputs[0]),
 nbOutputChannels));
     CHECK(cudnnSetTensor4dDescriptor(mSrcDescriptor,
CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, 1, nbOutputChannels, 1, 1));
     CHECK(cudnnSetTensor4dDescriptor(mDstDescriptor,
CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, batchSize, nbOutputChannels,
 1, 1));
     CHECK(cudnnAddTensor(mCudnn, &kONE, mSrcDescriptor,
mBiasWeights.values, &kONE, mDstDescriptor, outputs[0]));
     return 0;
}
```

3.6.6. Serialization

Layer parameters can be serialized along with the rest of the network. The serialization system calls the following functions:

```
virtual size_t getSerializationSize() override
{
    // 3 integers (number of input channels, number of output
channels, bias size), and then the weights:
    return sizeof(int)*3 + mKernelWeights.count*sizeof(float) +
mBiasWeights.count*sizeof(float);
}

virtual void serialize(void* buffer) override
{
    char* d = reinterpret_cast<char*>(buffer), *a = d;

    write(d, mNbInputChannels);
    write(d, mNbOutputChannels);
```

```
    write(d, (int)mBiasWeights.count);
    serializeFromDevice(d, mKernelWeights);
    serializeFromDevice(d, mBiasWeights);

    assert(d == a + getSerializationSize());
}
```

Deserialization is implemented with the following constructor:

```
// create the plugin at runtime from a byte stream
FCPlugin(const void* data, size_t length)
{
    const char* d = reinterpret_cast<const char*>(data), *a = d;
    mNbInputChannels = read<int>(d);
    mNbOutputChannels = read<int>(d);
    int biasCount = read<int>(d);

    mKernelWeights = deserializeToDevice(d, mNbInputChannels *
mNbOutputChannels);
    mBiasWeights = deserializeToDevice(d, biasCount);
    assert(d == a + length);
}
```

3.6.7. Call Sequence Summary

The following lists depict the call sequence for a plugin during each phase of TensorRT's operation.

When creating the network:

These methods are called during network construction if the output size of the layer, or any subsequent layer, is requested through an `ITensor::getDimensions()` call. Otherwise, the methods are called when the builder runs.

- `getNbOutputs()`
- `getOutputDimensions()`

By the builder:

- `configure()`
- `getWorkspaceSize()`

At runtime:

- `initialize()` when an engine context is constructed
- `enqueue()` at inference time
- `terminate()` when an engine context is destroyed

For serialization:

- `getSerializationSize()`
- `serialize()`

3.6.8. Adding the Plugin Into a Network

There are three ways to add the plugin into a network:

1. Use the `INetwork::addPlugin()` method when defining the network.
2. Create the network via a parser.
3. De-serialize the network after it has been built.

For use of the `addPlugin()` method, see the *TensorRT API* documentation.

3.6.8.1. Creating Plugins from NvCaffeParser

To add custom layers via NvCaffeParser, create a factory by implementing the nvcaffeParser::IPluginFactory interface, then pass an instance to `ICaffeParser::parse()`.

The `createPlugin()` method receives the layer name, and a set of weights extracted from the Caffe model file, which are then passed to the layer constructor. The name can be used to disambiguate between multiple plugins. There is currently no way to extract parameters other than weights from the Caffe network description, therefore, these parameters must be specified in the factory.

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

virtual nvinfer1::IPlugin* createPlugin(const char* layerName,
 const
nvinfer1::Weights* weights, int nbWeights) override
{
    // there's no way to pass parameters through from the model
definition, so we have to define it here explicitly
    static const int NB_OUTPUT_CHANNELS = 10;
    assert(isPlugin(layerName) && nbWeights == 2 &&
 weights[0].type ==
DataType::kFLOAT && weights[1].type == DataType::kFLOAT);
    assert(mPlugin.get() == nullptr);
    mPlugin = std::unique_ptr<FCPlugin>(new FCPlugin(weights,
nbWeights, NB_OUTPUT_CHANNELS));
    return mPlugin.get();
}
```

3.6.8.2. Creating Plugins at Runtime

To integrate custom layers with the runtime, implement the nvinfer1::IPlugin interface and pass an instance of the factory to `IInferRuntime::deserializeCudaEngine()`.

```
// deserialization plugin implementation
IPlugin* createPlugin(const char* layerName, const void*
 serialData,
size_t serialLength) override
```

```
{
    assert(isPlugin(layerName));
    assert(mPlugin.get() == nullptr);
    mPlugin = std::make_unique<FCPlugin>(serialData,
 serialLength);
    return mPlugin.get();
}
```

When constructed using the NvCaffeParser or deserialized at runtime, the layer implementation may assume that data passed as weights (from NvCaffeParser) or a byte stream (at runtime) will exist until the call to `initialize()`, allowing the data to be copied to the GPU in that function.

Currently only FP32 precision is supported by the plugin layer.

3.7. Sample 7: SampleFasterRCNN - Using the Plugin Library

The SampleFasterRCNN is a more complex example. This sample demonstrates how to implement the FasterRCNN network in TensorRT.

The FasterRCNN Caffe model is too large to include in the distribution. To run this sample, download the model using the instructions in the README in the sample directory.

SampleFasterRCNN uses a plugin from TensorRT's plugin library to include a fused implementation of FasterRCNN's RPN and ROIPooling layers. These particular layers are from the FasterRCNN paper and are implemented together as a single plugin called the FasterRCNN plugin.

The original Caffe model has been modified to include the FasterRCNN's RPN and ROIPooling layers.

Because TensorRT does not currently support the Reshape layer, it uses plugins to implement reshaping. The Reshape plugin requires a copy operation because the current version of TensorRT does not support in-place plugin layers.

There is code within SampleFasterRCNN, along with factories, that show how you can create and deserialize multiple plugins for a network.

Chapter 4. TROUBLESHOOTING

The following sections help answer the most commonly asked questions regarding typical use cases.

4.1. Creating an Engine that is Optimized for Several Batch Sizes

While TensorRT allows an engine optimized for a given batch size to run at any smaller size, the performance for those smaller sizes may not be as well-optimized.

To optimize for multiple different batch sizes, run the builder and serialize an engine for each batch size.

A future release of TensorRT will be able to optimize a single engine for multiple batch sizes, thereby allowing for sharing of weights where layers at different batch sizes use the same weight formats.

4.2. Choosing the Optimal Workspace Size

Some TensorRT algorithms require additional workspace on the GPU. The method `IBuilder::setMaxWorkspaceSize()` controls the maximum amount of workspace that may be allocated, and will prevent algorithms that require more workspace from being considered by the builder.

At runtime, the space is allocated automatically when creating an `IExecutionContext`. The amount allocated will be no more than is required, even if the amount set in `IBuilder::setMaxWorkspaceSize()` is much higher.

Applications should therefore allow the TensorRT builder as much workspace as they can afford; at runtime TensorRT will allocate no more than this, and typically less.

4.3. Using TensorRT on Multiple GPUs

Each `ICudaEngine` object is bound to a specific GPU when it is instantiated, either by the builder or on de-serialization.

To select the GPU, use `cudaSetDevice()` before calling the builder or de-serializing the engine. Each `IExecutionContext` is bound to the same GPU as the engine from which it was created. When calling `execute()` or `enqueue()`, ensure that the thread is associated with the correct device by calling `cudaSetDevice()` if necessary.

Notice

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

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

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

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

Trademarks

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

Copyright