TENSORRT 4.0.1

DU-08602-001_v4.0.1 | May 2018

Developer Guide

TABLE OF CONTENTS

Chapter 1. WHAT IS TENSORRT?

The core of TensorRT™ is a C++ library that facilitates high performance inference on NVIDIA graphics processing units (GPUs). It is designed to work in a complementary fashion with training frameworks such as TensorFlow, Caffe, PyTorch, MXNet, etc. It focuses specifically on running an already trained network quickly and efficiently on a GPU for the purpose of generating a result (a process that is referred to in various places as scoring, detecting, regression, or inference).

Some training frameworks such as TensorFlow have integrated TensorRT so that it can be used to accelerate inference within the framework. Alternatively, TensorRT can be used as a library within a user application. It includes parsers for importing existing models from Caffe, ONNX, or TensorFlow, and C++ and Python APIs for building models programmatically.

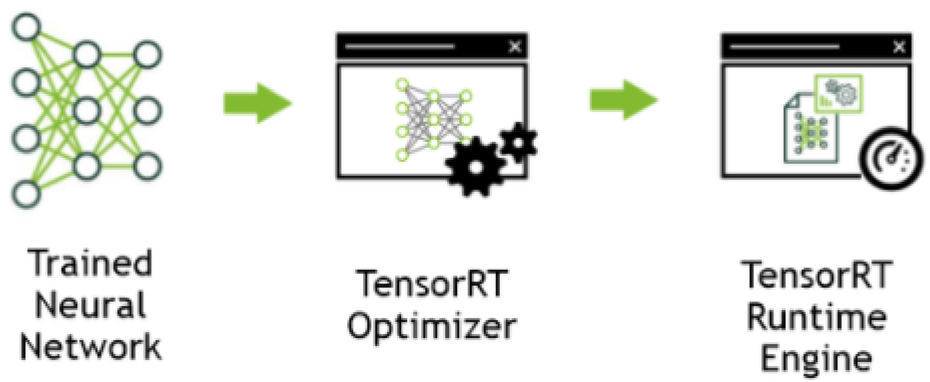

Figure 1 TensorRT is a high performance neural network inference optimizer and runtime engine for production deployment.

TensorRT optimizes the network by combining layers and optimizing kernel selection for improved latency, throughput, power efficiency and memory consumption. If the application specifies, it will additionally optimize the network to run in lower precision, further increasing performance and reducing memory requirements.

The TensorRT API includes implementations for the most common deep learning layers. For more information about the layers, see TensorRT Layers. You can also use the Plugin

API to provide implementations for infrequently used or more innovative layers that are not supported out-of-the-box by TensorRT.

1.1. Benefits Of TensorRT

After the neural network is trained, TensorRT enables the network to be compressed, optimized and deployed as a runtime without the overhead of a framework.

TensorRT combines layers, optimizes kernel selection, and also performs normalization and conversion to optimized matrix math depending on the specified precision (FP32, FP16 or INT8) for improved latency, throughput, and efficiency.

For deep learning inference, there are 5 critical factors that are used to measure software:

Throughput
: The volume of output within a given period. Often measured in inferences/second or samples/second, per-server throughput is critical to cost-effective scaling in data centers.

Efficiency
: Amount of throughput delivered per unit-power, often expressed as performance/watt. Efficiency is another key factor to cost effective data center scaling, since servers, server racks and entire data centers must operate within fixed power budgets.

Latency
: Time to execute an inference, usually measured in milliseconds. Low latency is critical to delivering rapidly growing, real-time inference-based services.

Accuracy
: A trained neural network's ability to deliver the correct answer. For image classification based usages, the critical metric is expressed as a top-5 or top-1 percentage.

Memory usage
: The host and device memory that need to be reserved to do inference on a network depends on the algorithms used. This constrains what networks and what combinations of networks can run on a given inference platform. This is particularly important for systems where multiple networks are needed and memory resources are limited - such as cascading multi-class detection networks used in intelligent video analytics and multi-camera, multi-network autonomous driving systems.

Alternatives to using TensorRT include:

- Using the training framework itself to perform inference.
- Writing a custom application that is designed specifically to execute the network using low level libraries and math operations.

Using the training framework to perform inference is easy, but tends to result in much lower performance on a given GPU than would be possible with an optimized solution like TensorRT. Training frameworks tend to implement more general purpose code which stress generality and when they are optimized the optimizations tend to focus on efficient training.

Higher efficiency can be obtained by writing a custom application just to execute a neural network, however it can be quite labor intensive and require quite a bit of specialized knowledge to reach a high level of performance on a modern GPU. Furthermore, optimizations that work on one GPU may not translate fully to other GPUs in the same family and each generation of GPU may introduce new capabilities that can only be leveraged by writing new code.

TensorRT solves these problems by combining an API with a high level of abstraction from the specific hardware details and an implementation which is developed and optimized specifically for high throughput, low latency, and low device memory footprint inference.

1.2. Where Does TensorRT Fit?

Generally the workflow for developing and deploying a deep learning model goes through three phases.

- Phase 1 is training
- Phase 2 is developing a deployment solution, and
- Phase 3 is the deployment of that solution

Phase 1: Training

During the training phase, the data scientists and developers will start with a statement of the problem they want to solve and decide on the precise inputs, outputs and loss function they will use. They will also collect, curate, augment, and probably label the training, test and validation data sets. Then they will design the structure of the network and train the model. During training, they will monitor the learning process which may provide feedback which will cause them to revise the loss function, acquire or augment the training data. At the end of this process, they will validate the model performance and save the trained model. Training and validation is usually done using DGX-1™, Titan, or Tesla datacenter GPUs.

TensorRT is generally not used during any part of the training phase.

Phase 2: Developing A Deployment Solution

During the second phase, the data scientists and developers will start with the trained model and create and validate a deployment solution using this trained model. Breaking this phase down into steps, you get:

1. Think about how the neural network functions within the larger system of which it is a part of and design and implement an appropriate solution. The range of systems that might incorporate neural networks are tremendously diverse. Examples include:
   - the autonomous driving system in a vehicle
   - a video security system on a public venue or corporate campus
   - the speech interface to a consumer device
   - an industrial production line automated quality assurance system

- an online retail system providing product recommendations, or
- a consumer web service offering entertaining filters users can apply to uploaded images.

Determine what your priorities are. Given the diversity of different systems that you could implement, there are a lot of things that may need to be considered for designing and implementing the deployment architecture.

- Do you have a single network or many networks?
- What device or compute element will you use to run the network?
- How is data going to get to the models?
- What pre-processing will be done?
- What latency and throughput requirements will you have?
- Will you be able to batch together multiple requests?
- Will you need multiple instances of a single network to achieve the required overall system throughput and latency?
- What will you do with the output of the network?
- What post processing steps are needed?

TensorRT provides a fast, modular, compact, robust, reliable inference engine that can support the inference needs within the deployment architecture.

2. After the data scientists and developers define the architecture of their inference solution, by which they determine what their priorities are, they then build an inference engine from the saved network using TensorRT. There are a number of ways to do this depending on the training framework used and the network architecture. Generally, this means you need to take the saved neural network and parse it from its saved format into TensorRT using the ONNX parser (see Figure 2), Caffe parser, or TensorFlow/UFF parser.

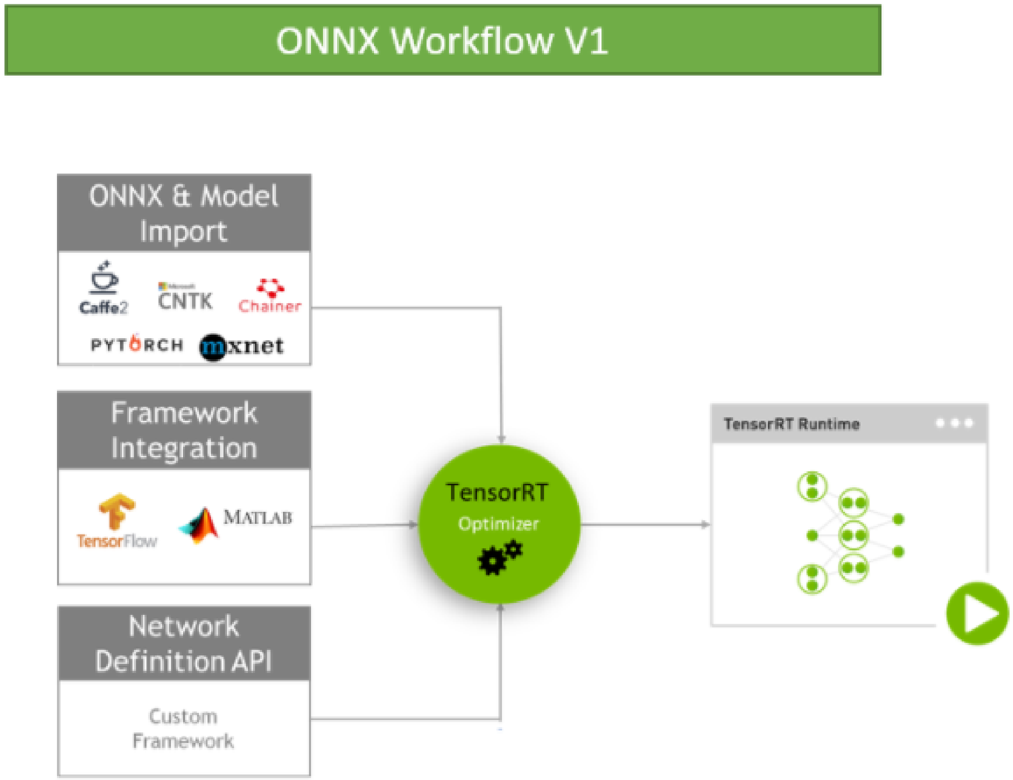

Figure 2 ONNX Workflow V1

3. After the network is being parsed, you'll need to consider optimization options -- batch size, workspace size and mixed precision. These options are chosen and specified as part of the TensorRT build step where you actually build an optimized inference engine based on your network. Subsequent sections of this guide provide detailed instructions and numerous examples on this part of the workflow, parsing your model into TensorRT and choosing the optimization parameters (see Figure 3).

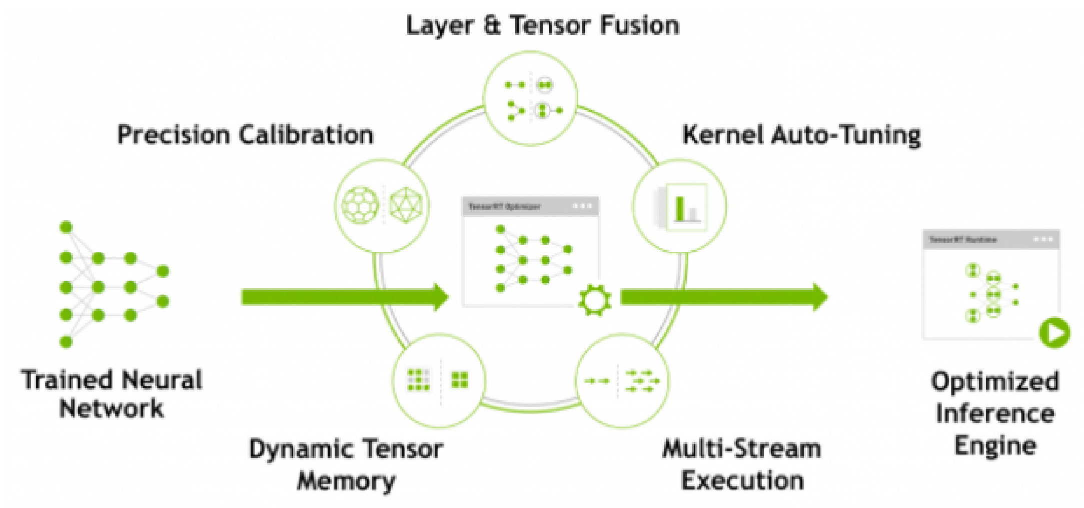

Figure 3 TensorRT optimizes trained neural network models to produce a deployment-ready runtime inference engine.

4. After you've created an inference engine using TensorRT, you'll want to validate that it reproduces the results of the model as measured during the training process. If you have chosen FP32 or FP16 it should match the results quite closely. If you have chosen INT8 there may be a small gap between the accuracy achieved during training and the inference accuracy.
5. Write out the inference engine in a serialized format. This is also called a plan file.

Phase 3: Deploying A Solution

The TensorRT library will be linked into the deployment application which will call into the library when it wants an inference result. To initialize the inference engine, the application will first deserialize the model from the plan file into an inference engine.

TensorRT is usually used asynchronously, therefore, when the input data arrives, the program calls an enqueue function with the input buffer and the buffer in which TensorRT should put the result.

1.3. How Does TensorRT Work?

To optimize your model for inference, TensorRT takes your network definition, performs optimizations including platform specific optimizations, and generates the inference engine. This process is referred to as the build phase. 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 generated plan file must be retargeted to the specific GPU in case you want to run it on a different GPU.

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)
- Merging of concatenation layers by directing layer outputs to the correct eventual destination.

The builder also modifies the precision of weights if necessary. When generating networks in 8-bit integer precision, it uses a process called calibration to determine the dynamic range of intermediate activations, and hence the appropriate scaling factors for quantization.

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.

For more information, see Working With Mixed Precision.

1.4. API Overview

The TensorRT API enables developers to import, calibrate, generate, and deploy optimized networks. Networks can be imported directly from Caffe, or from other frameworks via the UFF or ONNX formats. They may also be created programmatically by instantiating individual layers and setting parameters and weights directly.

TensorRT provides a C++ implementation on all supported platforms, and a Python implementation on x86.

The key interfaces in the TensorRT core library are:

Network Definition

The Network Definition interface provides methods for the application to specify the definition of a network. Input and output tensors can be specified, layers can be added, and there is an interface for configuring each supported layer type. As well as layer types, such as convolutional and recurrent layers, and a Plugin layer type allows the application to implement functionality not natively supported by TensorRT. For more information about the Network Definition, see Network Definition API.

Builder

The Builder interface allows creation of an optimized engine from a network definition. It allows the application to specify the maximum batch and workspace size, the minimum acceptable level of precision, timing iteration counts for autotuning, and an interface for quantizing networks to run in 8-bit precision. For more information about the Builder, see Builder API.

Engine

The Engine interface provides allow the application to executing inference. It supports synchronous and asynchronous execution, profiling, and enumeration and querying of the bindings for the engine inputs and outputs. A single engine can have multiple execution contexts, allowing a single set of set of trained parameters to be used for the simultaneous execution of multiple batches. For more information about the Engine, see Execution API.

TensorRT provides parsers for importing trained networks to create network definitions:

Caffe Parser
: This parser can be used to parse a Caffe network created in BVLC Caffe or NVCaffe 0.16. It also provides the ability to register a plugin factory for custom layers. For more details on the Caffe Parser, see NvCaffeParser.

Uff Parser
: This parser can be used to parse a network in UFF format. It also provides the ability to register a plugin factory and pass field attributes for custom layers. For more details on the API, see NvUffParser.

ONNX Parser
: This parser can be used to parse an ONNX model. For more details on the API, see NvONNXParser.

The Python API implementation includes a highly abstracted interface called TensorRT Lite. TensorRT Lite handles almost everything when it comes to building an engine and executing inference, therefore, users are able to quickly create an engine and start processing data. You can find TensorRT Lite in the **`tensorrt.lite`** directory. For more information see TensorRT Lite.

1.5. How Do I Get TensorRT?

For step-by-step instructions on how to install TensorRT, see the TensorRT Installation Guide.

Chapter 2. TENSORRT TASKS

The following sections highlight the user goals and tasks that you can perform with TensorRT. Further details are provided in the Samples section and are linked to below where appropriate.

The assumption is that you are starting with a trained model. This chapter will cover the following necessary steps in using TensorRT:

- Creating a TensorRT network definition from your model
- Invoking the TensorRT builder to create an optimized runtime engine from the network
- Serializing and deserializing the engine so that it can be rapidly recreated at runtime
- Feeding the engine with data to perform inference

Some further topics may be important depending on your use case:

- Augmenting TensorRT built-in functionality with custom layers
- Using mixed precision with TensorRT

2.1. Initializing TensorRT in C++

There are two ways to initialize the TensorRT library:

- Create an `IBuilder` object to optimize a network.
- Create an `IRuntime` object to execute an optimized network.

In either case, you must implement a logging interface through which TensorRT reports errors, warnings, and informational messages. The following code shows how to implement the logging interface. In this case, we have suppressed 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;
```

It is possible to create multiple runtime and builder objects; however the logger is a singleton, so you should use the same object for each.

The builder or runtime will be created with the GPU context associated with the creating thread. Although a default context will be created if it does not already exist, it is advisable to create and configure the CUDA context before creating a runtime or builder object.

2.2. Creating A Network Definition In C++

The first step in performing inference with TensorRT is to create a TensorRT network from your model. The easiest way to achieve this is to import the model using the TensorRT parser library, which supports serialized models in the following formats:

- Caffe (both BVLC and)
- ONNX, and
- UFF (used for TensorFlow)

An alternative is to define the model directly using the TensorRT API. This requires you to make a small number of API calls to define each layer in the network graph, and to implement your own import mechanism for the model's trained parameters.

In either case, you will explicitly need to tell TensorRT which tensors are required as outputs of inference. Tensors which are not marked as outputs are considered to be transient values that may be optimized away by the builder. There is no restriction on the number of output tensors, however, marking a tensor as an output may prohibit some optimizations on that tensor. Inputs and output tensors must also be given names (using **`ITensor::setName()`**). At inference time, you will supply the engine with an array of pointers to input and output buffers. In order to determine in which order the engine expects these pointers, you can query using the tensor names.

An important aspect of a TensorRT network definition is that it contains pointers to model weights, which are copied into the optimized engine by the builder. If a network was created via a parser, the parser will own the memory occupied by the weights, and so the parser object should not be deleted until after the builder has run.

2.2.1. Importing A Model Using A Parser In C++

To import a model using the C++ Parser API, you will need to perform the following high-level steps:

1. Create the TensorRT builder and network.
2. Create the TensorRT parser for the specific format.
3. Use the parser to parse the imported model and populate the network.

The builder must be created before the network because it serves as a factory for the network. Different parsers have different mechanisms for marking network outputs.

2.2.2. Importing A Caffe Model Using The C++ Parser API

The following steps illustrate how to import a Caffe model using the C++ Parser API. For more information, see sampleMNIST.

1. Create the builder and network:

```
IBuilder* builder = createInferBuilder(gLogger);
INetworkDefinition* network = builder->createNetwork();
```

2. Create the Caffe parser:

```
ICaffeParser* parser = createCaffeParser();
```

3. Parse the imported model:

```
const IBlobNameToTensor* blobNameToTensor = parser->parse("deploy_file" ,
 "modelFile", *network, DataType::kFLOAT);
```

   This populates the TensorRT network from the Caffe model. The final argument instructs the parser to generate a network whose weights are 32-bit floats. Using **`DataType::kHALF`** would generate a model with 16-bit weights instead.

   In addition to populating the network definition, the parser returns a dictionary that maps from Caffe blob names to TensorRT tensors. Unlike Caffe, a TensorRT network definition has no notion of in-place operation. When an Caffe model 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 writes to 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.

4. Specify the outputs of the network:

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

2.2.3. Importing A TensorFlow Model Using The C++ UFF Parser API

Importing from the TensorFlow framework requires you to convert the TensorFlow model into intermediate format UFF (Universal Framework Format). For more information about the conversion, see Converting A Frozen Graph To UFF.

The following steps illustrate how to import a TensorFlow model using the C++ Parser API. For more information about the UFF import, see sampleUffMNIST.

1. Create the builder and network:

```
IBuilder* builder = createInferBuilder(gLogger);
INetworkDefinition* network = builder->createNetwork();
```

2. Create the UFF parser:

```
IUFFParser* parser = createUffParser();
```

3. Declare the network inputs and outputs to the UFF parser:

```
parser->registerInput("Input_0", DimsCHW(1, 28, 28), UffInputOrder::kNCHW);
parser->registerOutput("Binary_3");
```

> TensorRT expects the input tensor be in CHW order. When importing from TensorFlow, ensure that the input tensor is in the required order, and if not, convert it to CHW.

4. Parse the imported model to populate the network:

```
parser->parse(uffFile, *network, nvinfer1::DataType::kFLOAT);
```

2.2.4. Importing An ONNX Model Using The C++ Parser API

The following steps illustrate how to import an ONNX model using the C++ Parser API. For more information about the ONNX import, see sampleOnnxMNIST.

1. Create the ONNX parser. The parser uses an auxiliary configuration management **SampleConfig** object to pass the input arguments from the sample executable to the parser object:

```
nvonnxparser::IOnnxConfig* config = nvonnxparser::createONNXConfig();
//Create Parser
nvonnxparser::IONNXParser* parser = nvonnxparser::createONNXParser(*config);
```

2. Ingest the model:

```
parser->parse(onnx_filename, DataType::kFLOAT);
```

3. Convert the model to a TensorRT network:

```
parser->convertToTRTNetwork();
```

4. Obtain the network from the model:

```
nvinfer1::INetworkDefinition* trtNetwork = parser->getTRTNetwork();
```

2.3. Creating A Network Using The C++ API

Instead of using a parser, you can also define the network directly to TensorRT via the network definition API. This scenario assumes that the per-layer weights are ready in host memory to pass to TensorRT during the network creation.

In the following example, we will create a simple network with Input, Convolution, Pooling, FullyConnected, Activation and SoftMax layers. For more information, see sampleMNISTAPI.

1. Create the builder and the network:

```
IBuilder* builder = createInferBuilder(gLogger);
INetworkDefinition* network = builder->createNetwork();
```

2. Add the Input layer to the network, with the input dimensions. A network can have multiple inputs, although in this sample there is only one:

```
auto data = network->addInput(INPUT_BLOB_NAME, dt, Dims3{1, INPUT_H,
 INPUT_W});
```

3. Add the Convolution layer with hidden layer input nodes, strides and weights for filter and bias. In order to retrieve the tensor reference from the layer, we can use:

```
layerName->getOutput(0)
auto conv1 = network->addConvolution(*data->getOutput(0), 20, DimsHW{5, 5},
 weightMap["conv1filter"], weightMap["conv1bias"]);
conv1->setStride(DimsHW{1, 1});
```

Weights passed to TensorRT layers are in host memory.

4. Add the Pooling layer:

```
auto pool1 = network->addPooling(*conv1->getOutput(0), PoolingType::kMAX,
 DimsHW{2, 2});
pool1->setStride(DimsHW{2, 2});
```

5. Add the FullyConnected and Activation layers:

```
auto ip1 = network->addFullyConnected(*pool1->getOutput(0), 500,
 weightMap["ip1filter"], weightMap["ip1bias"]);
auto relu1 = network->addActivation(*ip1->getOutput(0),
 ActivationType::kRELU);
```

6. Add the SoftMax layer to calculate the final probabilities and set it as the output:

```
auto prob = network->addSoftMax(*relu1->getOutput(0));
prob->getOutput(0)->setName(OUTPUT_BLOB_NAME);
```

7. Mark the output:

```
network->markOutput(*prob->getOutput(0));
```

2.4. Building An Engine In C++

The next step is to invoke the TensorRT builder to create an optimized runtime. One of the functions of the builder is to search through its catalog of CUDA kernels for the fastest implementation available, and thus it is necessary use the same GPU for building as that on which the optimized engine will run.

The builder has many properties that you can set in order to control such things as the precision at which the network should run, and autotuning parameters such as how many times TensorRT should time each kernel when ascertaining which is fastest (more iterations leads to longer runtimes, but less susceptibility to noise.) You can also query the builder to find out what reduced precision types are natively supported by the hardware.

Two particularly important properties are the maximum batch size and the maximum workspace size.

- The maximum batch size specifies the batch size for which TensorRT will optimize. At runtime, a smaller batch size may be chosen.
- Layer algorithms often require temporary workspace. This parameter limits the maximum size that any layer in the network can use. If insufficient scratch is provided, it is possible that TensorRT may not be able to find an implementation for a given layer.

1. Build the engine using the builder object:

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

   When the engine is built, TensorRT makes copies of the weights.
2. Dispense with the network, builder, and parser if using one.

2.5. Serializing A Model In C++

Building can take some time, so once the engine is built, you will typically want to serialize it for later use. It is not absolutely necessary to serialize and deserialize a model before using it for inference – if desirable, the engine object can be used for inference directly.

> Serialized engines are not portable across platforms or TensorRT versions. Engines are specific to the exact GPU model they were built on (in addition to platforms and the TensorRT version).

1. Run the builder as a prior offline step and then serialize:

```
IHostMemory *serializedModel = engine->serialize();
// store model to disk
// <…>
serializedModel->destroy();
```

2. Create a runtime object to deserialize:

```
IRuntime* runtime = createInferRuntime(gLogger);
ICudaEngine* engine = runtime->deserializeCudaEngine(modelData, modelSize,
 nullptr);
```

   The final argument is a plugin layer factory for applications using custom layers. For more information, see Extending TensorRT With Custom Layers.

2.6. Performing Inference In C++

Once you have an engine, you can perform inference.

1. Create some space to store intermediate activation values. Since the engine holds the network definition and trained parameters, additional space is necessary. These are held in an execution context:

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

   An engine can have multiple execution contexts, allowing one set of weights to be used for multiple overlapping inference tasks. For example, you can process images in parallel CUDA streams using one engine and one context per stream. Each context will be created on the same GPU as the engine.
2. Use the input and output blob names to get the corresponding input and output index:

   ```
   int inputIndex = engine.getBindingIndex(INPUT_BLOB_NAME);
   int outputIndex = engine.getBindingIndex(OUTPUT_BLOB_NAME);
   ```
3. Using these indices, set up a buffer array pointing to the input and output buffers on the GPU:

   ```
   void* buffers[2];
   buffers[inputIndex] = inputbuffer;
   buffers[outputIndex] = outputBuffer;
   ```
4. TensorRT execution is typically asynchronous, so **enqueue** the kernels on a CUDA stream:

   ```
   context.enqueue(batchSize, buffers, stream, nullptr);
   ```

   It is common to **enqueue** asynchronous `memcpy()` before and after the kernels to move data from the GPU if it is not already there. The final argument to `enqueue()` is an optional CUDA event which will be signaled when the input buffers have been consumed and their memory may be safely reused.

   To determine when the kernels (and possibly `memcpy()`) are complete, use standard CUDA synchronization mechanisms such as events, or waiting on the stream.

2.7. Memory Management In C++

TensorRT provides two mechanisms to allow the application more control over device memory.

By default, when creating an **IExecutionContext**, persistent device memory is allocated to hold activation data. To avoid this allocation, call **createExecutionContextWithoutDeviceMemory**. It is then the application's responsibility to call **IExecutionContext::setDeviceMemory()** to provide the required memory to run the network. The size of the memory block is returned by **ICudaEngine::getDeviceMemorySize()**.

In addition, the application can supply a custom allocator for use during build and runtime by implementing the **IGpuAllocator** interface. Once the interface is implemented, call

```
setGpuAllocator(&allocator);
```

on the **IBuilder** or **IRuntime** interfaces. All device memory will then allocated and freed through this interface.

2.8. Initializing TensorRT in Python

There are two ways to initialize the TensorRT library:

- Create an `IBuilder` object to optimize a network.
- Create an `IRuntime` object to execute an optimized network.

In either case, you must implement a logging interface through which TensorRT reports errors, warnings, and informational messages. The following code shows how to implement the logging interface. In this case, we have suppressed informational messages, and report only warnings and errors. There is a simple logger included in **tensorrt.infer.ConsoleLogger**.

```
G_LOGGER = trt.infer.ConsoleLogger(trt.infer.LogSeverity.ERROR)
```

It is possible to create multiple runtime and builder objects; however the logger is a singleton, so you should use the same object for each.

The builder or runtime will be created with the GPU context associated with the creating thread. Although a default context will be created if it does not already exist, it is advisable to create and configure the CUDA context before creating a runtime or builder object.

2.9. Creating A Network Definition In Python

The first step in using TensorRT for inference is to create a TensorRT representation of your network, from which TensorRT can build an optimized runtime.

Whether you choose to import a model using a parser or import an existing model from a framework, both of these techniques are described below and are demonstrated in Python. Even though the steps required here are executed in Python, they are very similar to the steps outlined in Creating A Network Definition In C++.

> TensorRT Python API is available for x86_64 platform only. For more information please see Deep Learning SDK Documentation - TensorRT workflows.

2.9.1. Importing A Model Using A Parser In Python

To import a model using the Python Parser API, you will need to perform the following high-level steps:

1. Create the TensorRT builder and network.
2. Create the TensorRT parser for the specific format.
3. Use the parser to parse the imported model and populate the network.

The builder must be created before the network because it serves as a factory for the network. Different parsers have different mechanisms for marking network outputs.

2.9.2. Importing From Caffe Using Python

The following example shows how you can import a Caffe model directly using the NvCaffeParser and the Python API. Related examples can be found in the **../examples/caffe_to_trt/caffe_mnist.py** directory. For more information, see sampleMNIST.

1. Import TensorRT as you would import any other package:
```
import tensorrt as trt
```
2. Define the data type. In this example, we will use float32.
```
datatype = trt.infer.DataType.FLOAT
```
3. Additionally, define some paths. Change the following paths to reflect where you placed the model included with the samples:
```
MODEL_PROTOTXT = '/data/mnist/mnist.prototxt'
CAFFE_MODEL = '/data/mnist/mnist.caffemodel'
```
4. Create the builder:
```
builder = trt.infer.create_infer_builder(G_LOGGER)
```
5. Create the network:
```
network = builder.create_network()
```
6. Create the parser:
```
parser = parsers.caffeparser.create_caffe_parser()
```
7. Parse the Caffe network and weights, and create the TensorRT network:
```
blob_name_to_tensor = parser.parse(CAFFE_MODEL, MODEL_PROTOTXT,
                                   network, datatype)
```
   The output is the populated network (passed as the argument to the parser). In addition, the parser returns the **blob_name_to_tensor** - a table containing the mapping from tensor names to **ITensor** objects.

2.9.3. Importing From TensorFlow Using Python

The following example shows how you can import a TensorFlow model directly using the NvUffParser and the Python API. This example can be found in the **<site-packages>/tensorrt/examples/tf_to_trt** directory. For more information, see the tf_to_trt Python sample.

1. Import TensorRT and its UFF parser by running the following commands:
```
import tensorrt as trt
from tensorrt.parsers import uffparser
```

2. Create a frozen TensorFlow model for the **tensorflow** model. The instructions on freezing a TensorFlow model into a stream can be found in Freezing A TensorFlow Graph.
3. Use the UFF converter to convert a frozen **tensorflow** model to a UFF file. The instructions on freezing the TensorFlow model and saving into a file can be found in Convert a Tensorflow Model to UFF.

```
import uff
uff.from_tensorflow_frozen_model(frozen_file, ["fc2/Relu"])
```

4. Create the UFF Parser to parse the UFF file into TensorRT network. The UFF also requires the input and output nodes to be specified, along with dimensions of the input node:

```
parser = uffparser.create_uff_parser()
```

   This can be done in the following way:

```
parser.register_input("Placeholder", (1, 28, 28), 0)
parser.register_output("fc2/Relu")
```

> TensorRT expects the input tensor be in CHW order. When importing from TensorFlow, ensure that the input tensor is in the required order, and if not, convert it to CHW.

5. Create the engine:

```
engine = trt.utils.uff_to_trt_engine(G_LOGGER,
                                     uff_model,
                                     parser,
                                     MAX_BATCHSIZE,
                                     MAX_WORKSPACE)
```

2.9.4. Importing From ONNX Using Python

The following example shows how you can import an ONNX model directly using the NvOnnxParser and the Python API. For more information, see sample_onnx and sampleOnnxMNIST.

1. Import TensorRT as you would import any other package:

```
import tensorrt as trt
```

2. Import the NvOnnxParser to directly convert the ONNX model into the TensorRT network. Similar to C++ APIs, the sample_onnx Python sample uses the **config** object to pass user arguments to the parser object.

```
from tensorrt.parsers import onnxparser
apex = onnxparser.create_onnxconfig()
```

3. Parse a trained image classification model and then generate TensorRT engine for inference. Here we parse the user input arguments to generate the **config** object:

```
apex.set_model_filename("model_file_path")
apex.set_model_dtype(trt.infer.DataType.FLOAT)
apex.set_print_layer_info(True) // Optional debug option
apex.report_parsing_info()  // Optional debug option
```

In order to control the debug output, there are different ways you can control the verbosity level:

```
apex.add_verbosity()
apex.reduce_verbosity()
```

Or, you can set the specific verbosity level:

```
apex.set_verbosity_level(3)
```

4. After the **config** object is created and configured, you can create the parser. Ensure you retrieve the parameters from the created object to parse the input model file:

```
trt_parser = onnxparser.create_onnxparser(apex)
data_type = apex.get_model_dtype()
onnx_filename = apex.get_model_file_name()
```

5. Generate the TensorRT network after parsing the model file:

```
trt_parser.parse(onnx_filename, data_type)

// retrieve the network from the parser
trt_parser.convert_to_trt_network()
trt_network = trt_parsr.get_trt_network()
```

To perform inference, follow the instructions outlined in Performing Inference In Python.

2.9.5. Importing From PyTorch And Other Frameworks

Using TensorRT with PyTorch (or any other framework with NumPy compatible weights) involves replicating the network architecture using the TensorRT API, and then copying the weights from PyTorch. For more information, see Working With PyTorch And Other Frameworks.

To perform inference, follow the instructions outlined in Performing Inference In Python.

2.10. Creating A Network Using The Python API

When creating a network, you must first define the engine and create a builder object for inference. The Python API is used to create a network and engine from the Network APIs. The network definition reference is used to add various layers to the network. For more information, see sampleMNISTAPI.

In this example, we will create a simple network with Input, Convolution, Pooling, FullyConnected, Activation and SoftMax layers.

1. Create the builder and the network:

```
builder = trt.infer.create_infer_builder(G_LOGGER)
network = builder.create_network()
```

2. Add the input layer to the network. We can define the input blob name with input tensor dimensions. Any network can have multiple inputs. In this example, we have one input of given name and dimension. Dimension is defined as tuple of channel, height and width. We can also load the weights into a Weight map.

```
data = network.add_input(INPUT_LAYERS[0], dt, (1, INPUT_H, INPUT_W))
weight_map = trt.utils.load_weights(weights_file)
```

3. Add the Convolution layer with hidden layer input nodes, strides and weights for filter and bias:

```
conv1 = network.add_convolution(scale1.get_output(0), 20, (5,5),
 weight_map["conv1filter"], weight_map["conv1bias"])
conv1.set_stride((1,1))
```

> Weights passed to TensorRT layers are in host memory.

4. Add the Pooling layer with pooling type and dimension. We can also set the corresponding stride for the pooling layer:

```
pool1 = network.add_pooling(conv1.get_output(0), trt.infer.PoolingType.MAX,
 (2,2))
pool1.set_stride((2,2))
```

5. Add the FullyConnected and Activation layers:

```
ip1 = network.add_fully_connected(pool2.get_output(0), 500,
 weight_map["ip1filter"], weight_map["ip1bias"])
relu1 = network.add_activation(ip1.get_output(0),
 trt.infer.ActivationType.RELU)
```

6. Add the SoftMax layer to calculate the final probabilities and set it as the output:

```
prob = network.add_softmax(ip2.get_output(0))
prob.get_output(0).set_name(OUTPUT_LAYERS[0])
```

7. Mark the output:

```
network.mark_output(prob.get_output(0))
```

2.11. Building An Engine In Python

One of the functions of the builder is to search through its catalog of CUDA kernels for the fastest implementation available, and thus it is necessary use the same GPU for building as that on which the optimized engine will run.

The builder has many properties that you can set in order to control such things as the precision at which the network should run, and autotuning parameters such as how many times TensorRT should time each kernel when ascertaining which is fastest (more iterations leads to longer runtimes, but less susceptibility to noise.) You can also query the builder to find out what mixed precision types are natively supported by the hardware.

Two particularly important properties are the maximum batch size and the maximum workspace size.

- The maximum batch size specifies the batch size for which TensorRT will optimize. At runtime, a smaller batch size may be chosen.
- Layer algorithms often require temporary workspace. This parameter limits the maximum size that any layer in the network can use. If insufficient scratch is provided, it is possible that TensorRT may not be able to find an implementation for a given layer.

1. Build the engine using the builder object:

```
builder.set_max_batch_size(max_batch_size)
builder.set_max_workspace_size(1 << 20)
engine = builder.build_cuda_engine(network)
```

   When the engine is built, TensorRT makes copies of the weights.
2. Dispense with the network, builder, and parser if using one.

For more information about building an engine in Python, see the caffe_mnist sample as described in sampleMNIST and onnx_mnist as described in sampleOnnxMNIST.

2.12. Serializing A Model In Python

From here onwards you can either serialize the engine or you can use the engine directly for inference. Serializing and deserializing a model is an optional step before using it for inference - if desirable, the engine object can be used for inference directly.

> Serialized engines are not portable across platforms or TensorRT versions. Engines are specific to the exact GPU model they were built on (in addition to platforms and the TensorRT version).

1. Run the builder as a prior offline step and then serialize:

```
IHostMemory *serializedModel = engine->serialize();
// store model to disk
// <…>
serializedModel->destroy();
```

2. Serialize the model to a modelstream and free up some memory by deleting the engine and builder object:

```
modelstream = engine.serialize()
engine.destroy()
builder.destroy()
```

3. Deserialize modelstream to perform inference. Deserializing requires creation of runtime object:

```
runtime = trt.infer.create_infer_runtime(GLOGGER)
```

```
engine =
 runtime.deserialize_cuda_engine(modelstream.data(),modelstream.size(),
 None)
modelstream.destroy()
```

The final argument is a plugin layer factory for applications using custom layers. More details can be found in Extending TensorRT With Custom Layers.

2.13. Performing Inference In Python

Once you have an engine, you can perform inference.

1. Create some space to store intermediate activation values. Since the engine holds the network definition and trained parameters, additional space is necessary. These are held in an execution context:

    ```
    context = engine.create_execution_context()
    ```

    An engine can have multiple execution contexts, allowing one set of weights to be used for multiple overlapping inference tasks. For example, you can process images in parallel CUDA streams using one engine and one context per stream. Each context will be created on the same GPU as the engine.
2. Set up a buffer array pointing to the input and output buffers on the GPU:

    ```
    d_input = cuda.mem_alloc(insize)
    d_output = cuda.mem_alloc(outsize)
    bindings = [int(d_input), int(d_output)]
    ```
3. TensorRT execution is typically asynchronous, so **`enqueue`** the kernels on a CUDA stream:

    ```
    context.enqueue(batch_size, bindings, stream.handle, None)
    ```
4. Copy the results back from the device output buffer to the output array:

    ```
    cuda.memcpy_dtoh_async(output, d_output, stream)
    ```

    It is common to enqueue asynchronous **`memcpy()`** before and after the kernels to move data from the GPU if it is not already there. The final argument to **`enqueue()`** is an optional CUDA event which will be signaled when the input buffers have been consumed and their memory may be safely reused.
5. Determine when the kernels (and possibly **`memcpy()`**) are complete, use standard CUDA synchronization mechanisms such as events, or wait on the stream.

    ```
    stream.synchronize()
    return output
    ```

2.14. Extending TensorRT With Custom Layers

TensorRT supports many types of layers and its functionality is continually extended; however, there may be cases in which the layers supported do not cater to the specific needs of a model. In this case, users can extend TensorRT functionalities by implementing custom layers using the C++ API. Custom layers, often referred to as plugins, are implemented and instantiated by an application, and their lifetime must span their use within a TensorRT engine.

2.14.1. Adding Custom Layers Using The C++ API

A custom layer is implemented by extending the **`IPluginExt`** class. Although users extended the **`IPlugin`** class in previous versions of TensorRT, it is now recommended that users extend **`IPluginExt`**, which includes versioning (to maintain plugin portability in future versions of TensorRT) and enables custom layers that support other data formats beside NCHW and single precision. The remainder of this section refers to plugins of type **`IPluginExt`**, although everything applies also to plugins of type **`IPlugin`** except for multi-format specific support.

> Plugins of type **`IPlugin`** are assumed to support only single precision NCHW tensors.

A plugin layer is added to a network with the **`addPluginExt`** (see TensorRT APIs) method which creates and adds a layer to a network, and then binds the layer to the given plugin. The method also returns a pointer to the layer (which is of type **`IPluginLayerExt`**), which can be used to access the layer or the plugin itself (via **`getPluginExt`**).

To properly connect a plugin layer to neighboring layers, and setup input and output data structures, the builder checks what is the number of outputs and their dimensions calling plugins methods:

`getNbOutputs`
: Used to specify the number of output tensors.

`getOutputDimensions`
: Used to specify the dimensions of an output as a function of the input dimensions.

In addition, during the build phase, the network is constructed and analyzed to generate an engine, and the plugin is checked for formats supported:

`supportsFormat`
: Used to check if a plugin supports a given data format.

Plugin layers can support four data formats and layouts. These are NCHW single and half precision tensors, NC/2HW2 and NHWC8 half precision tensors. The formats are enumerated by **`PluginFormatType`**.

Plugins that do not compute all data in place and need memory space in addition to input and output tensors can specify the additional memory requirements with the **`getWorkspaceSize`** method, which is called by the builder to determine and pre-allocate scratch space.

During both build and inference time, the plugin layer is configured and executed, possibly multiple times. At build time, to discover optimal configurations, the layer is configured, initialized, executed, and terminated. Once the optimal format is selected for a plugin, the plugin is once again configured, and then it will be initialized once and executed as many time as needed for the lifetime of the inference application, and finally terminated when the engine is destroyed. These steps are controlled by the builder and the engine using the following plugin methods:

`configureWithFormat`
: Communicates input and output number, dimensions, datatype, format, and maximum batch size. At this point, the plugin sets up its internal state, and select the most appropriate algorithm and data structures for the given configuration.

`initialize`
: The configuration is known at this time and the inference engine is being created, so the plugin can set up its internal data structures and prepare for execution.

`enqueue`
: Encapsulates the actual algorithm and kernel calls of the plugin, and provides the runtime batch size, pointers to input, output, and scratch space, and the CUDA stream to be used for kernel execution.

`terminate`
: The engine context is destroyed and all the resources held by the plugin should be released.

Serializing A Model In C++ introduces the serialization and deserialization of engines, to enable storage and deployment of engines (for example, avoiding repeated build phases in deployment). To support this capability, plugins also must support serialization and deserialization; this is achieved by defining **`getSerializationSize`** returning the required size to store the state of the plugin (including relevant configuration details) and **`serialize`**, which must store the state in the given buffer. When the engine is serialized it will first check the serialization size of the plugin, and then serialize the plugin providing a buffer of the requested size.

Deserialization requires an additional **`IPluginFactory`** that recognizes plugin layers and instantiates the corresponding plugin object. When the runtime, see Serializing A Model In C++, deserializes the engine, it uses the plugin factory **`createPlugin`** method to create plugin object for a given layer name and serialized image.

The C++ API can be used also to create custom layers for use in Python. C++ is the preferred language to implement custom layers (for example, to easily access libraries like CUDA and cuDNN). The custom layer created in C++ can be packaged using the SWIG plugin in Python setuptools and then, the plugin can be loaded into a Python application (see Creating A Network Using The Python API and TensorRT Python Bindings). The same custom layer implementation can be used for both C++ and Python.

2.14.2. Using Custom Layers When Importing A Model From A Framework

Custom layers can also be integrated with model parsers and used when importing models. To extend a parser, users define a parser-specific factory. For more information, see Creating A Network Definition In C++ or Creating A Network Definition In Python.

Namespaces **`nvcaffeparser1`** and **`nvuffparser`**, for Caffe and UFF respectively, include a **`IPluginFactoryExt`** class (and **`IPluginFactory`**) that complements the **`IPluginFactoryExt`** defined in namespace **`nvinfer1`**. To be used within a parser, a plugin factory must extend both the generic and parser-specific factory classes.

The **setPluginFactoryExt** method of the parser sets the factory in the parser to enable custom layers. While parsing a model description, for each layer, the parser invokes **isPluginExt** to check with the factory if the layer name corresponds to a custom layer; if it does, the parser instantiates the plugin invoking **createPlugin** with the name of the layer (so that the factory can instantiates the corresponding plugin), a **Weights** array, and the number of weights, (and a **FieldCollection** for UFF) as arguments. There is no restriction on the number of plugins that a single factory can support if they are associated with different layer names.

samplePlugin illustrates in detail an example of a custom layer and how to extend the Caffe parser.

2.15. Working With Mixed Precision

Mixed precision is the combined use of different numerical precisions in a computational method. TensorRT can store weights and activations, and execute layers, in 32-bit floating point, 16-bit floating point, or quantized 8-bit integer.

Using precision lower than FP32 reduces memory usage, allowing deployment of larger networks. Data transfers take less time, and compute performance increases, especially on GPUs with Tensor Core support for that precision.

By default, TensorRT uses FP32 inference, but it also supports FP16 and INT8. While running FP16 inference, it automatically converts FP32 weights to FP16 weights.

> Specifying the precision for a network defines the minimum acceptable precision for the application. Higher precision kernels may be chosen if they are faster for some particular set of kernel parameters, or if no lower-precision kernel exists.

2.15.1. Enabling FP16 Inference Using C++

Setting the builder's **Fp16Mode** flag indicates that 16-bit precision is acceptable.

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

This flag allows, but does not guarantee, that 16-bit kernels will be used when building the engine.

Weights can be specified in FP16 or FP32, and they will be converted automatically to the appropriate precision for the computation.

See sampleGoogleNet for an example of running FP16 inference.

2.15.2. Enabling FP16 Inference Using Python

In Python, set the **fp16_mode** flag as follows:

```
builder.set_fp16_mode(True);
```

For more information, see sample_onnx, mnist_api as shown in sampleMNIST, and pytorch_to_trt. Both mnist_api and pytorch_to_trt samples show FP16 disabled by default.

2.15.3. Optimizing INT8 Calibration Using C++ API

When using 8-bit quantized representation, TensorRT needs to understand the dynamic range of each activation tensor so that it can choose an appropriate quantization scale. The process of determining these scale factors is called calibration, and requires the application to pass batches of representative input for the network (typically batches from the training set.) Experiments indicate that about 500 images is sufficient for calibrating ImageNet classification networks.

To provide calibration data to TensorRT, implement the **`IInt8Calibrator`** interface. The builder invokes the calibrator as follows:

- First, it calls `getBatchSize()` to determine the size of the input batch to expect
- Then, it repeatedly calls **`getBatch()`** to obtain batches of input. Batches should be exactly the batch size by **`getBatchSize()`**. When there are no more batches, **`getBatch()`** should return **`false`**.

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:
  - 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.

After you have implemented the calibrator, you can configure the builder to use it:

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

It is possible to cache the output of calibration using the `writeCalibrationCache()` and `readCalibrationCache()` methods. The builder checks the cache prior to performing calibration, and if data is found, calibration is skipped.

For more information about configuring INT8 Calibrator objects, see sampleINT8.

2.15.4. Optimizing INT8 Calibration Using Python

The following example shows you how to create an INT8 Calibrator object using the Python API. By default, TensorRT supports INT8 Calibration.

1. Import TensorRT as you would import any other package:

```
import tensorrt as trt
```

2. Similar to test/validation files, use set of input files as calibration files dataset. Make sure the calibration files are representative of the overall inference data files. For TensorRT to use the calibration files, we need to create batchstream object. Batchstream object will be used to configure the calibrator.

```
NUM_IMAGES_PER_BATCH = 5
batchstream = ImageBatchStream(NUM_IMAGES_PER_BATCH, calibration_files)
```

3. Create an **`Int8_calibrator`** object with input nodes names and batch stream:

```
Int8_calibrator = trt.infer.EntropyCalibrator(["input_node_name"],
 batchstream)
```

4. Set INT8 mode and INT8 Calibrator:

```
trt_builder = trt.infer.create_infer_builder(G_LOGGER)
trt_builder.set_int8_mode(True)
trt_builder.set_int8_calibrator(Int8_calibrator)
```

   The rest of the logic for engine creation and inference is similar to Importing From ONNX Using Python.

2.16. Deploying A TensorRT Optimized Model

After you've created a plan file containing your optimized inference model, you can deploy that file into your production environment. How you create and deploy the plan file will depend on your environment. For example, you may have a dedicated inference executable for your model that loads the plan file and then uses the TensorRT Execution API to pass inputs to the model, execute the model to perform inference, and finally read outputs from the model.

This section discusses how TensorRT can be deployed in some common deployment environments.

2.16.1. Deploying In The Cloud

One common cloud deployment strategy for inferencing is to expose a model through a server that implements an HTTP REST or gRPC endpoint for the model. A remote client can then perform inferencing by sending a properly formatted request to that endpoint. The request will select a model, provide the necessary input tensor values required by the model, and indicate which model outputs should be calculated.

To take advantage of TensorRT optimized models within this deployment strategy does not require any fundamental change. The inference server must be updated to accept models represented by TensorRT plan files and must use the TensorRT Execution APIs to load and executes those plans. An example of an inference server that provides a REST endpoint for inferencing can be found in the NVIDIA Inference Server Container Release Notes and Inference Server User Guide.

2.16.2. Deploying To An Embedded System

TensorRT can also be used to deploy trained networks to embedded systems such as NVIDIA Drive PX. In this context, deployment means taking the network and using it in a software application running on the embedded device, such as an object detection or mapping service. Deploying a trained network to an embedded system involves the following steps:

1. Export the trained network to a format such as UFF or ONNX which can be imported into TensorRT (see Working With Deep Learning Frameworks for more details).
2. Write a program that uses the TensorRT C++ API to import, optimize, and serialize the trained network to a plan file (see sections Working With Deep Learning Frameworks, Working With Mixed Precision, and Performing Inference In C++). For the purpose of discussion, let's call this program **`make_plan`**.
   a) Optionally, perform INT8 calibration and export a calibration cache (see Working With Mixed Precision).
3. Build and run **`make_plan`** on the host system to validate the trained model before deployment to the target system.
4. Copy the trained network (and INT8 calibration cache, if applicable) to the target system. Re-build and re-run the **`make_plan`** program on the target system to generate a plan file.

> The **`make_plan`** program must run on the target system in order for the TensorRT engine to be optimized correctly for that system. However, if an INT8 calibration cache was produced on the host, the cache may be re-used by the builder on the target when generating the engine (in other words, there is no need to do INT8 calibration on the target system itself).

After the plan file has been created on the embedded system, an embedded application can create an engine from the plan file and perform inferencing with the engine by using the TensorRT C++ API. For more information, see Performing Inference In C++.

To walk through a typical use case where a TensorRT engine is deployed on an embedded system, see:

- Deploying INT8 Inference For Autonomous Vehicles for DRIVE PX

- GitHub for Jetson and Jetpack

Chapter 3. WORKING WITH DEEP LEARNING FRAMEWORKS

With the Python API, an existing model built with TensorFlow, Caffe, or an ONNX compatible framework can be used to build a TensorRT engine using the provided parsers. The Python API also supports frameworks that store layer weights in a NumPy compatible format, for example PyTorch.

3.1. Supported Operations

The following lists describe the operations that are supported in a Caffe or TensorFlow framework and in the ONNX TensorRT parser:

Caffe

The following list describes the operations that are supported in a Caffe framework.

- Convolution
- Pooling
- InnerProduct
- SoftMax
- ReLU, TanH, Sigmoid
- LRN
- Power
- ElementWise
- Concatenation
- Deconvolution
- BatchNormalization
- Scale
- Crop
- Reduction

- Reshape
- Permute
- Dropout

TensorFlow

The following list describes the operations that are supported in a TensorFlow framework.

- Placeholder
- Const
- Add, Sub, Mul, Div, Minimum and Maximum
- BiasAdd
- Negative, Abs, Sqrt, Rsqrt, Pow, Exp and Log
- FusedBatchNorm
- ReLU, TanH, Sigmoid
- SoftMax
- Mean
- ConcatV2
- Reshape
- Transpose
- Conv2D
- DepthwiseConv2dNative
- ConvTranspose2D
- MaxPool
- AvgPool
- Pad is supported if followed by one of these TensorFlow layers: Conv2D, DepthwiseConv2dNative, MaxPool, and AvgPool

ONNX

Since the ONNX parser is an open source project, the most up-to-date information regarding the supported operations can be found in GitHub: ONNX TensorRT.

For a list of supported operators in an ONNX model, see the online Developer Guide.

For more information about each of the TensorRT layers, see TensorRT Layers.

3.2. Working With TensorFlow

For information on using TensorRT with a TensorFlow model, see:

- The tf_to_trt Python sample
- The lite_examples Python sample, if using the Lite API

- Generate TensorRT Engines from Tensorflow (or other UFF Compatible Frameworks)

3.2.1. Freezing A TensorFlow Graph

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

- A Tool Developer's Guide to TensorFlow Model Files: Freezing
- Exporting trained TensorFlow models to C++ the RIGHT way!

3.2.2. Freezing A Keras Model

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

```
from keras.models import load_model
import keras.backend as K
from tensorflow.python.framework import graph_io
from tensorflow.python.tools import freeze_graph
from tensorflow.core.protobuf import saver_pb2
from tensorflow.python.training import saver as saver_lib

def convert_keras_to_pb(keras_model, out_names, models_dir,
 model_filename):
 model = load_model(keras_model)
 K.set_learning_phase(0)
 sess = K.get_session()
 saver = saver_lib.Saver(write_version=saver_pb2.SaverDef.V2)
 checkpoint_path = saver.save(sess, 'saved_ckpt', global_step=0,
 latest_filename='checkpoint_state')
 graph_io.write_graph(sess.graph, '.', 'tmp.pb')
 freeze_graph.freeze_graph('./tmp.pb', '',
                           False, checkpoint_path, out_names,
                           "save/restore_all", "save/Const:0",
                           models_dir+model_filename, False, "")
```

3.2.3. Converting A Frozen Graph To UFF

You can use the following sample code to convert the **.pb** frozen graph to **.uff** format file.

```
convert-to-uff tensorflow -o name_of_output_uff_file --input-
file
name_of_input_pb_file -O  name_of_output_tensor
```

You can list the TensorFlow layers:

```
convert-to-uff tensorflow --input-file name_of_input_pb_file -l
```

to figure out the **name_of_output_tensor** value.

3.2.4. Working With TensorFlow RNN Weights

This section provides information about TensorFlow weights and their stored formats. Additionally, the following sections will guide you on how to approach and decrypt RNN weights from TensorFlow.

3.2.4.1. TensorFlow RNN Cells Supported In TensorRT

An RNN layer in TensorRT can be thought of as a **`MultiRNNCell`** from TensorFlow. One layer consists of sublayers with the same configurations, in other words, hidden and embedding size. This encapsulation is done so that the internal connections between the multiple sublayers can be abstracted away from the user. This allows for simpler code when deeper networks are involved.

TensorRT supports four different RNN layer types. These layer types are RNN **`relu`**, RNN **`tanh`**, LSTM, and GRU. The TensorFlow cells that match these types are:

TensorRT RNN Relu/Tanh Layer

1. **`BasicRNNCell`**
    a. Permitted activation functions: **`tf.tanh`** and **`tf.nn.relu`**.
    b. This is a platform independent cell.

TensorRT LSTM Layer

1. **`BasicLSTMCell`**
    a. **`forget_bias`** must be set to **`0`** when creating an instance of this cell in TensorFlow. To support a non-zero forget bias, you need to preprocess the bias by adding the parameterized forget bias to the dumped TensorFlow forget biases.
    b. This is a platform independent cell.
2. **`CudnnCompatibleLSTMCell`**
    a. Same condition for the forget bias applies to this cell as it does to the **`BasicLSTMCell`**.
    b. TensorRT does not currently support peepholes so **`use_peepholes`** must be set to **`False`**.
    c. This is a cuDNN compatible cell.

TensorRT GRU Layer

1. **`CudnnCompatibleGRUCell`**
    a. This is a cuDNN compatible cell.
    b. Differs in implementation from standard, platform independent GRU cells. Due to this, **`CudnnCompatiableGRUCell`** is the correct cell to use with TensorRT.

3.2.4.2. Maintaining Model Consistency Between TensorFlow And TensorRT

For any TensorFlow cell not listed in TensorFlow RNN Cells Supported In TensorRT, consult the TensorRT API and TensorFlow API to ensure the cell is mathematically equivalent to what TensorRT supports and the storage format is consistent with the format that you are expecting. One good way of doing this is to set up unit tests to validate the output from TensorRT by using TensorFlow as the ground truth.

3.2.4.3. Workflow

We will be using the following workflow to extract and use TensorFlow weights:

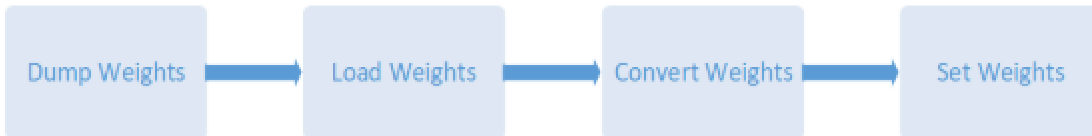

Figure 4 TensorFlow RNN Workflow

3.2.4.4. Dumping The TensorFlow Weights

Python script **`dumpTFWts.py`** can be used to dump all the variables and weights from a given TensorFlow checkpoint. The script is located in the **`/usr/src/tensorrt/samples/common/dumpTFWts.py`** directory. Issue **`dumpTFWts.py -h`** for more information on the usage of this script.

3.2.4.5. Loading Dumped Weights

Function **`loadWeights()`** loads from the dump of the **`dumpTFWts.py`** script. It has been provided as an example in sampleCharRNN. The function signature is:

```
std::map<std::string, Weights> loadWeights(const std::string file,
 std::unordered_set<std::string> names);
```

This function loads the weights specified by the names set from the specified file and returns them in a **`std::map<std::string, Weights>`**.

3.2.4.6. Converting The Weights To A TensorRT Format

At this point, we are ready to convert the weights. To do this, the following steps are required:

1. Understanding and using the TensorFlow checkpoint to get the tensor.
2. Understanding and using the tensors to extract and reformat relevant weights and set them to the corresponding layers in TensorRT.

3.2.4.6.1. TensorFlow Checkpoint Storage Format

There are two possible TensorFlow checkpoint storage formats:

1. Platform independent format - separated by layer
    a. **`Cell_i_kernel <Weights>`**
    b. **`Cell_i_bias <Weights>`**
2. cuDNN compatible format - separated by input and recurrent
    a. **`Cell_i_Candidate_Input_kernel <Weights>`**
    b. **`Cell_i_Candidate_Hidden_kernel <Weights>`**

In other words, 1.1 **`Cell_i_kernel <Weights>`** in the concatenation of 2.1 **`Cell_i_Candidate_Input_kernel <Weights>`** and 2.2 **`Cell_i_Candidate_Hidden_kernel <Weights>`**. Therefore, storage format 2 is simply a more fine-grain version of storage format 1.

3.2.4.6.2. TensorFlow Kernel Tensor Storage Format

Before storing the weights in the checkpoint, TensorFlow transposes and then interleaves the rows of transposed matrices. The order of the interleaving is described in the next section. A figure is provided in BasicLSTMCell Example to further illustrate this format.

Gate Order Based On Layer Operation Type The transposed weight matrices are interleaved in the following order:

1. RNN relu/tanh:
    a. input gate (**`i`**)
2. LSTM:
    a. input gate (**`i`**), cell gate (**`c`**) , forget gate (**`f`**), output gate (**`o`**)
3. GRU:
    a. reset (**`r`**), update (**`u`**)

3.2.4.6.3. Kernel Weights Conversion To A TensorRT Format

Converting the weights from TensorFlow format can be summarized in two steps.

1. Reshape the weights to push the interleaving down to a lower dimension.
2. Transpose the weights to get rid of the interleaving completely and have the weight matrices stored contiguously in memory.

Transformation Utilities To help perform these transformations correctly, `reorderSubBuffers()`, `transposeSubBuffers()`, and `reshapeWeights()` are functions that have been provided. For more information, see **`/usr/include/x86_64-linux-gnu/NvUtils.h`**.

3.2.4.6.4. TensorFlow Bias Weights Storage Format

The bias tensor is simply stored as contiguous vectors concatenated in the order specified in TensorFlow Kernel Tensor Storage Format. If the checkpoint storage is platform independent, then TensorFlow combines the recurrent and input biases into a single tensor by adding them together. Otherwise, the recurrent and input biases and stored in separate tensors.

3.2.4.6.5. Bias Tensor Conversion To TensorRT Format

Since the biases are stored as contiguous vectors, there aren't any transformations that need to be applied to get the bias into the TensorRT format.

3.2.4.7. `BasicLSTMCell` Example

3.2.4.7.1. BasicLSTMCell Kernel Tensor

To understand the format in which these tensors are being stored, let us consider an example of a **BasicLSTMCell**. Figure 5 illustrates what the tensor looks like within the TensorFlow checkpoint.

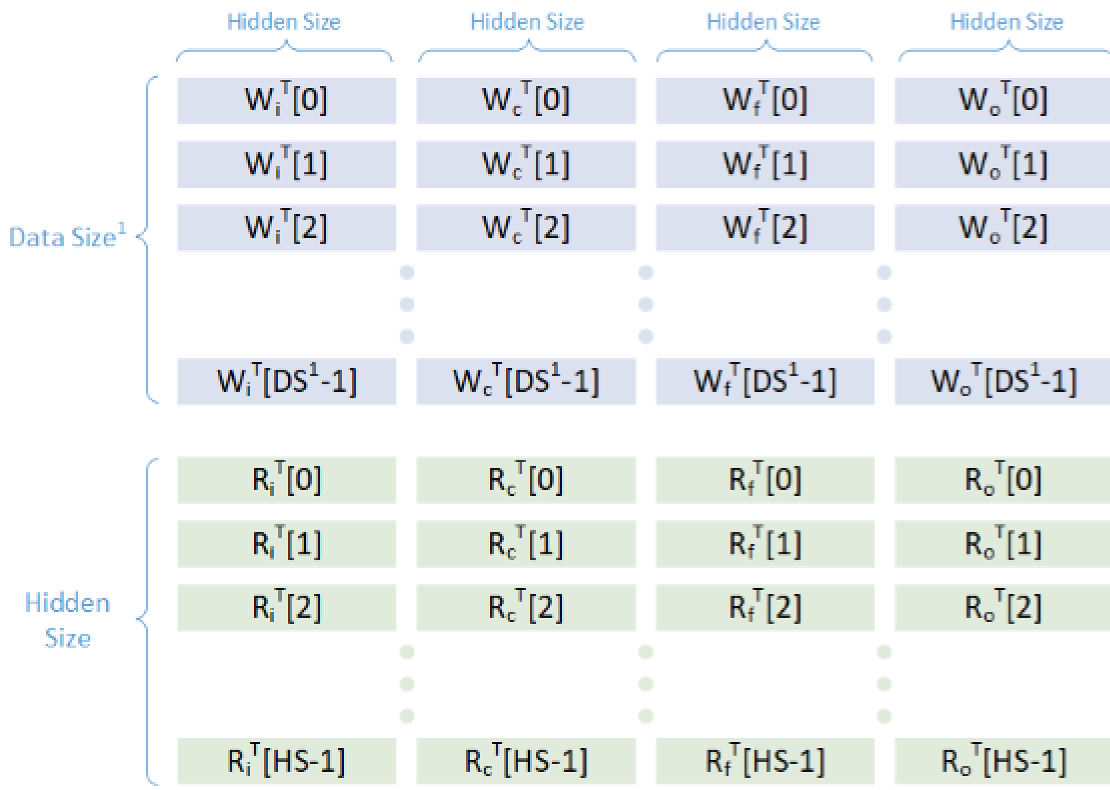

Figure 5 Tensors within a TensorFlow checkpoint

> **DS/Data Size** is distinct from **Hidden Size** for the first layer. For all the following sublayers **Data Size** is equal to **Hidden Size**.

In Figure 5, **W** represents the input weights, **R** represents the hidden weights, **DS** represents the data size, and **HS** represents hidden size.

> Since this is a platform independent cell, the input weights and hidden weights have been concatenated together. If we had used a **CudnnCompatibleLSTMCell**, then these weights would have been split into two separate tensors.

Applying the conversion process discussed earlier will result in the converted tensor shown in Figure 6.

Figure 6 Converted tensors

`Data Size` is distinct from `Hidden Size` for the first layer in the sequence of RNN sublayers. For all the following sublayers `Data Size` is equal to `Hidden Size`.

3.2.4.7.2. BasicLSTMCell Bias Tensor

Figure 7 illustrates the format in which the bias tensor is stored.

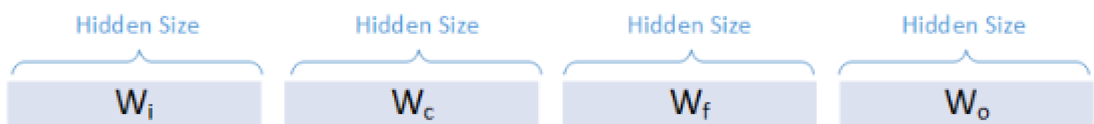

Figure 7 Bias tensor stored format

Because this is a platform independent cell, `W` in the image above represents the result of ElementWise adding the input and recurrent biases together. TensorFlow does this addition internally to save memory before it stores the tensor.

This is already in the format we require, therefore, we do not need to apply any transformations.

3.2.4.8. Setting The Converted Weights And Biases

The converted tensors for both the weights and bias are now ready to use. You need to iterate over the tensors in the order specified in TensorFlow Kernel Tensor Storage

Format and set the weights and bias using **`IRNNv2Layer::setWeightsForGate()`** and **`IRNNv2Layer::setBiasForGate()`** functions, respectively.

> If you are using a platform independent cell, you will need to set all the recurrent biases manually using zeroed out dummy weights.

A real-world example of the training, dumping, converting, and setting process is described in sampleCharRNN. For more information, consult the code in this sample.

3.3. Working With PyTorch And Other Frameworks

Using TensorRT with PyTorch and other frameworks involves replicating the network architecture using the TensorRT API, and then copying the weights from PyTorch (or any other framework with NumPy compatible weights). For more information on using TensorRT with a PyTorch model, see:

- the pytorch_to_trt Python sample
- Generate TensorRT Engines from Tensorflow (or other UFF Compatible Frameworks)

3.4. Working With The TensorRT Lite Engine

From A TensorFlow Model

The Lite Engine supports TensorFlow models directly by performing the conversion to UFF internally. The model can be provided as a serialized graph or a path to a protobuf file by using the appropriate keyword argument.

```
mnist_engine = tensorrt.lite.Engine(framework="tf",     #Source framework
                                   path="mnist/lenet5_mnist_frozen.pb",   #Model
 File
                                   max_batch_size=10,   #Max number of images to
 be processed at a time
                                   input_nodes={"in":(1,28,28)},  #Input layers
                                   output_nodes=["out"])   #Output layers
```

From A UFF Model

Similar to creating a TensorRT Lite Engine from a TensorFlow model, the TensorRT Lite Engine API can accept a UFF model stream or a path to a UFF file.

```
stream = uff.from_tensorflow_frozen_model("mnist/lenet5_mnist_frozen.pb" ,
["out"])

mnist_engine = tensorrt.lite.Engine(framework="uff",
 stream=stream,input_nodes={"in":(1,28,28)},output_nodes=["out"])
```

From A Caffe Model

The TensorRT Lite Engine API can also accept paths to a Caffe model file and deploy file.

```
mnist_engine = tensorrt.lite.Engine(framework="c1",
 #Source framework
                                deployfile="mnist/mnist.prototxt", #Deploy file
                                modelfile="mnist/mnist.caffemodel", #Model File
                                max_batch_size=10, #Max number of images to be
 processed at a time
                                input_nodes={"data":(1,28,28)}, #Input layers
                                output_nodes=["prob"], #Ouput layers
                                preprocessors={"data":sub_mean}, #Preprocessing
 functions
                                postprocessors={"prob":argmax}) #Postprocesssing
 functions
```

From A Plan File

The TensorRT Lite Engine API support loading prebuilt engines from plan files.

```
mnist_engine = tensorrt.lite.Engine(PLAN="mnist.plan",
                                 max_batch_size=10,
                                 preprocessors={"data":sub_mean},
                                 postprocessors={"prob":argmax})
```

3.4.1. Running Inference

After your engine is created, you can use the **`infer`** call to run inference on a set of data. Your data can be provided to the Lite engine in a couple of ways. Each input must be a NumPy array matching the input shape defined in the constructor. For example, if the input shape for a layer is **`1,28,28`**, then each input must be in the shape **`1,28,28`**. From the base structure, the input data can be formatted as:

- A single input, for example, one image, as a 3D NumPy array matching the input shape.
- A list or NumPy array of data, for example, a list of images as 3D NumPy arrays where each 3D array matches the input layer shape.

  > This list or array can be as long as you want. Internally, the array will be batched according to the max batch size.

- A list or an array of batched data where each batch is a list or array of data with each element being the shape of the input layer and the length of each batch is smaller than the max batch size.

If you have multiple input layers, pass the inputs for each layer as a separate argument to the engine in the order you defined the layers in the inputs dictionary of the constructor. The format of each layer must be the same, down to batch sizes, if applicable.

After inference has been run, the results are returned in the same format as the input format. Data is always returned inside a list where the index refers to the output layer in the same order as they were listed in the constructor.

For more information, see the examples in Working With The TensorRT Lite Engine.

3.4.2. Preprocessing And Postprocessing Function Tables

Typically, some preprocessing of input data before inference, and post-processing the results of inference, is required to get usable results for a larger application. To allow for cleaner code when integrating a Lite engine into a larger application, the constructor allows users to populate a function table for preprocessing and post-processing each input (3D NumPy array) and each result (also a 3D NumPy array).

For example, if a user provides a large amount of raw data to the infer function, then each image is normalized before inference. Run ArgMax on each result to receive an array of lists describing both the top class and the top-5 for each result.

In the following sample code, you will create a dictionary to contain your functions, each keyed to the name of the respective layers. If you have multiple input layers but only want to pre-process one, then you must have entries for all other layers still, but instead of the function, pass **None**.

```
# Preprocessing function
def normalize(data):
    # Each image is provided as a 3D numpy array (like how it’s provided to
 inference function)
    for i in range(len(data)): # normalize
        data[i] = 1.0 - data[i] / 255.0
    # Reshape the data to the shape expected by the network
    return data.reshape(1,28,28)

# Lambda to apply argmax to each result after inference to get prediction
# Instead of having to reshape, you can replace the 3D array provided to the
 postprocessor with the object of your choosing (e.g. the top class)
argmax = lambda res: np.argmax(res.reshape(10))

# Register pre and post processors to their layers
mnist_engine = tensorrt.lite.Engine(framework="tf",  # Source framework
                                    path=DATA_DIR + "/mnist/
lenet5_mnist_frozen.pb",       # Model File
                                    max_batch_size=10,   # Max number of images
 to be processed at a time
                                    input_nodes={"in":(1,28,28)},  # Input
 layers
                                    output_nodes=["out"],   # Ouput layers
                                    preprocessors={"in":normalize},   #
 Preprocessing functions
                                    postprocessors={"out":argmax})    #
 Postprocesssing functions


def generate_cases(num):
    '''
    Generate a list of raw data (data will be processed in the engine) and
 answers to compare to
    '''
    cases = []
    labels = []
    for c in range(num):
        rand_file = randint(0, 9)
        im = Image.open(str(rand_file) + ".pgm")
        arr = np.array(im).reshape(1,28,28) #Make the image CHANNEL x HEIGHT x
 WIDTH
```

```
        cases.append(arr) # Append the image to list of images to process
        labels.append(rand_file) # Append the correct answer to compare later
    return cases, labels

def main():
    # Generate cases
    data, target = generate_cases(10)
    # Run inference on our generated cases doing preprocessing and
 postprocessing internally
    results = mnist_engine.infer(data)[0] #Data is returned in a list by output
 layer

    # Validate results
    correct = 0
    print ("[LABEL] | [RESULT]")
    for l in range(len(target)):
        print ("   {}    |    {}   ".format(target[l], results[l]))
        if target[l] == results[l]:
            correct += 1
    print ("Inference: {:.2f}% Correct".format((correct / len(target)) * 100))
```

Chapter 4. SAMPLES

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

C++ Samples

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

- sampleMNIST
- sampleMNISTAPI
- sampleUffMNIST
- sampleOnnxMNIST
- sampleGoogleNet
- sampleCharRNN
- sampleINT8
- samplePlugin
- sampleNMT
- sampleFasterRCNN
- sampleUffSSD
- sampleMovieLens

Python Examples

You can find the Python examples in the **`{PYTHON_PACKAGE_DIR}/tensorrt/examples`** directory. The following Python examples are shipped with TensorRT:

- caffe_to_trt - functionally is identical to sampleMNIST
- custom_layers - functionally is identical to samplePlugin
- lite_examples
- resnet_as_a_service
- sample_onnx
- tf_to_trt

- googlenet - functionally is identical to sampleGoogleNet
- mnist_api - functionally is identical to sampleMNISTAPI
- onnx_mnist - functionally is identical to sampleOnnxMNIST
- uff_mnist - functionally is identical to sampleUffMNIST
- pytorch_to_trt

The TensorRT package comes with multiple example implementations, which can be found in one of the following locations:

Table 1 TensorRT Sample Application Implementations

In you installed TensorRT with:	Examples are located in:
`sudo apt-get install python(3)-libnvinfer`	`/usr/lib/python{2.7,3.5}/dist-packages/tensorrt/examples`
`sudo pip(3) install tensorrt....`	`/usr/local/lib/python{2.7,3.5}/dist-packages/tensorrt/examples`
`pip(3) install --user tensorrt....`	`$HOME/.local/lib/python{2.7,3.5}/dist-packages/tensorrt/examples`

4.1. sampleMNIST

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

- Perform the basic setup and initialization of TensorRT
- 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?
The sampleMNIST sample is installed in the **`/usr/src/tensorrt/samples/sampleMNIST`** directory and is applicable to both C++ and Python environments. This sample is functionally identical to the caffe_to_trt Python example. The caffe_to_trt example is installed in the **`{PYTHON_PACKAGE_DIR}/tensorrt/examples/caffe_to_trt`** 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, sampleMNIST 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 8 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 9 Decision output

4.2. sampleMNISTAPI

What Does This Sample Do?

The sampleMNISTAPI sample is similar to sampleMNIST sample. Both of these samples use the same model, handle the same input, and expect similar output. In contrast to sampleMNIST, the sampleMNISTAPI demonstrates how to:

- Build a network by individually creating every layer
- Load the layers with theirs weights and connecting the layers by linking their inputs and outputs

Where Is This Sample Located?

The sampleMNISTAPI sample is installed in the **`/usr/src/tensorrt/samples/sampleMNISTAPI`** directory and is applicable to both C++ and Python environments. This sample is functionally identical to the mnist_api Python example. The mnist_api example is installed in the **`{PYTHON_PACKAGE_DIR}/tensorrt/examples/mnist_api`** directory.

Notes About This Sample:

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

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.

4.3. sampleUffMNIST

What Does This Sample Do?

The sampleUffMNIST sample demonstrates how to:

- Implement 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?

The sampleUffMNIST sample is installed in the **`/usr/src/tensorrt/samples/sampleUffMNIST`** directory and is applicable to both C++ and Python environments. This sample is functionally identical to the uff_mnist Python example. The uff_mnist example is installed in the **`{PYTHON_PACKAGE_DIR}/tensorrt/examples/uff_mnist`** 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.

4.4. sampleOnnxMNIST

What Does This Sample Do?

The sampleOnnxMNIST 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

The sampleOnnxMNIST 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?

The sampleOnnxMNIST sample is installed in the **`tensorrt/samples/sampleOnnxMNIST`** directory and is applicable to both C++ and Python environments. This sample is functionally identical to the onnx_mnist Python example. The onnx_mnist example is installed in the **{PYTHON_PACKAGE_DIR}/tensorrt/examples/onnx_mnist** directory.

4.4.1. 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();
```

4.4.2. 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.

4.4.3. 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 sampleMNIST.

4.5. sampleGoogleNet

What Does This Sample Do?

The sampleGoogleNet sample demonstrates how to:

- Use FP16 mode in TensorRT
- Use TensorRT **`Half2Mode`**
- Use layer-based profiling

Where Is This Sample Located?

The sampleGoogleNet sample is installed in the **`/usr/src/tensorrt/samples/sampleGoogleNet`** directory and is applicable to both C++ and Python environments. This sample is functionally identical to the googlenet Python example. The googlenet example is installed in the **`{PYTHON_PACKAGE_DIR}/tensorrt/examples/googlenet`** directory.

4.5.1. Configuring The Builder

The sampleGoogleNet example 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);
```

4.5.2. 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 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 layer-combining operations, so the reported layer names in the profiling output may not be a one-to-one map to the original layer names. For example, the layers **inception_5a/3x3** and **inception_5a/relu_3x3** in the original network are fused into one layer named **inception_5a/3x3 + inception_5a/relu_3x3**.

4.6. sampleCharRNN

What Does This Sample Do?

The sampleCharRNN 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?

The sampleCharRNN 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

4.6.1. 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.

4.6.1.1. 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.

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

> 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.

4.6.1.2. 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.

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

4.6.1.3. 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.

```
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 Multiplicaton output");
```

For more information, see the TensorRT API documentation.

4.6.1.4. 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:

```
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");
```

For more information, see the TensorRT API documentation.

4.6.1.5. 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.

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

For more information, see the TensorRT API documentation.

4.6.1.6. 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.

```
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));
};
```

4.6.2. 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.

4.6.2.1. 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.

4.6.2.2. 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.

4.6.2.3. 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.

```
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;
```

4.6.2.4. 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_Lf, W_Li, W_Lc, W_Lo, R_Lf, R_Li, R_Lc, R_Lo]**, 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.

```
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;
}
```

4.6.3. 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.

```
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]);
```

4.6.4. 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**.

```
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]);
}
```

4.7. sampleINT8

What Does This Sample Do?

The sampleINT8 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.

The sampleINT8 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?

The sampleINT8 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.

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
```

4.7.1. 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);
```

4.7.2. 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.

4.7.2.1. 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.

4.7.2.2. 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.

4.7.3. Configuring The Builder

There are two additional methods to call on the builder:

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

4.7.4. 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.

4.7.5. 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%.

4.7.6. 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.

4.7.6.1. 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
```

4.7.6.2. 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
```

4.8. samplePlugin

What Does This Sample Do?

The samplePlugin demonstrates how to add a Custom layer to TensorRT. This example 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 sampleMNIST, sampleMNISTAPI, and sampleUffMNIST.

The samplePlugin 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?

The samplePlugin sample is installed in the **/usr/src/tensorrt/samples/samplePlugin** directory and is applicable to both C++ and Python environments.

This sample is functionally identical to the custom_layers Python example. The custom_layers example is installed in the **{PYTHON_PACKAGE_DIR}/tensorrt/examples/custom_layers** 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.

4.8.1. 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);
}
```

4.8.2. 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();
}
```

4.8.3. 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.

4.8.4. 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.

```
IPlugin* createPlugin(...) override
{
    …
    mPlugin = std::make_unique<FCPlugin>(serialData, serialLength);
    return mPlugin.get();
}
```

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);
}
```

4.8.5. 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.

```
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;
}
```

4.9. sampleNMT

What Does This Sample Do?

sampleNMT 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.

The sampleNMT 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?

The sampleNMT 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.

4.9.1. 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++.

4.9.2. 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
```

4.9.3. 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.

4.9.4. 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=<path>/wmt16_de_en/vocab.bpe.32000 \
   --train_prefix=<path>/wmt16_de_en/train.tok.clean.bpe.32000 \
   --dev_prefix=<path>/wmt16_de_en/newstest2013.tok.bpe.32000 \
   --test_prefix=<path>/wmt16_de_en/newstest2015.tok.bpe.32000
```

4.9.5. 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:

```
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.

4.10. sampleFasterRCNN

What Does This Sample Do?

The sampleFasterRCNN sample demonstrates how to:

- Use the Faster R-CNN plugin which allows 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?

The sampleFasterRCNN 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.

Notes About This Sample:

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

4.10.1. 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 the **`FasterRCNNPlugin`**.

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).

4.10.2. 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.

4.10.3. 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 samplePlugin, 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 samplePlugin, 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 **createFasterRCNNPlugin**. This function returns an instance that implements an optimized **RPROIFused** Custom layer and performs the same logic designed by the authors.

4.10.4. 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++.

4.10.5. Running The Engine

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

In sampleFasterRCNN, there are two inputs:

data
: **data** is the image input

im_info
: **im_info** is the image information array which contains 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.

4.10.6. 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.

4.11. sampleUffSSD

What Does This Sample Do?

The sampleUffSSD 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?

The sampleUffSSD 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.

4.11.1. 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.

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 **detectionOutput** plugin.

> 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 **detectionOutput** plugin requires the data to be in this format.

4.11.2. 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 example 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 **detectionOutput** 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. TensorRT does not currently support **Relu6(x)** operation, so the preprocessor also replaces this operation with a **Relu(x) - Relu(x-6)**.

Additional documentation on the graph preprocessor can be found in the TensorRT API.

4.11.3. 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`**.

4.11.4. Defining The Network And Plugins

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

The **`pluginFactory`** object created needs to be passed to an instance of **`IUffParser::parse()`** which will invoke the **`createPlugin()`** function for each Custom layer. 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)

`DetectionOutput` Plugin

The **`detectionOutput`** 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 > 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 TensorFlow bounding box data is not normalized so the **isNormalized** flag specifies if the data is normalized.

4.11.5. 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 **detectionOutput** layer. 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.

4.12. sampleMovieLens

What Does This Sample Do?

The sampleMovieLens 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?

The sampleMovieLens 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).

The sample can be built with Multi Process Service (MPS) mode enabled, and can use a configurable number of processes once MPS mode is enabled.

4.12.1. 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.

4.12.2. Running With MPS

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
```

In order to run the sample with MPS, recompile with **`USE_MPS=1`**.

4.12.3. 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`**.

4.13. lite_examples

What Does This Example Do?

The lite_examples example outlines various workflows using the TensorRT Lite API. It demonstrates how to:

- Build a TensorRT engine from a Caffe model using the Lite API
- Load a TensorRT engine from a plan file using the Lite API
- Build a TensorRT engine from a TensorFlow model using the Lite API

Where Is This Example Located?

The lite_examples example is installed in the **`{PYTHON_PACKAGE_DIR}/tensorrt/examples/lite_examples`** directory.

Notes About This Example:

Each of the included examples uses a LeNet5 model trained on the MNIST handwritten digits dataset. To verify accuracy, each example tests the engine against 10 test cases.

For more information on using the TensorRT Lite API see Working With The TensorRT Lite Engine.

4.14. pytorch_to_trt

What Does This Example Do?

In order to use a PyTorch model with TensorRT, the model architecture must be recreated using the TensorRT API. The pytorch_to_trt example demonstrates how to:

- Working With PyTorch And Other Frameworks
- Creating A Network Using The Python API
- Building An Engine In Python and Performing Inference In Python

Where Is This Example Located?

The pytorch_to_trt example is installed in the **`{PYTHON_PACKAGE_DIR}/tensorrt/examples/pytorch_to_trt`** directory.

Notes About This Example:

The example uses a PyTorch model and trains it on the MNIST handwritten digits dataset. To verify accuracy, it tests the engine against 10 test cases.

4.15. resnet_as_a_service

What Does This Example Do?

The resnet_as_a_service example demonstrates how to:

- Build an engine from a TensorFlow model using the TensorRT Lite API
- Deploy the engine as part of a RESTful service using Flask

Where Is This Example Located?

The resnet_as_a_service example is installed in the **`{PYTHON_PACKAGE_DIR}/tensorrt/examples/resnet_as_a_service`** directory.

Notes About This Example:

This example uses a ResNet-50 model.

4.16. sample_onnx

What Does This Example Do?

The sample_onnx example demonstrates how to:

- Convert a model in ONNX format to a TensorRT engine
- Importing From ONNX Using Python and run inference

Where Is This Example Located?

The sample_onnx example is installed in the **`{PYTHON_PACKAGE_DIR}/tensorrt/examples/sample_onnx`** directory.

Notes About This Example:

The example offers a variety of command-line options to modify various aspects of the engine. You can view a full list of options by passing the **`-h`** flag to the sample.

4.17. tf_to_trt

What Does This Example Do?

The tf_to_trt example demonstrates the TensorFlow to TensorRT workflow after the model has been frozen to a protobuf. Specifically, it highlights how to:

- Training a network in TensorFlow
- Converting A Frozen Graph To UFF

- Importing From TensorFlow Using Python and Performing Inference In Python

Where Is This Example Located?

The tf_to_trt example is installed in the **`{PYTHON_PACKAGE_DIR}/tensorrt/examples/tf_to_trt`** directory.

Notes About This Example:

This example uses a LeNet5 model trained on the MNIST handwritten digits dataset. To verify accuracy, it tests the engine against 10 test cases.

Chapter 5. TROUBLESHOOTING

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

5.1. FAQs

Q: How do you create an engine that is optimized for several different batch sizes?
A: 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.

Q: How do you choose the optimal workspace size?
A: 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.

Q: How do you use TensorRT on multiple GPUs?
A: Each **`ICudaEngine`** object is bound to a specific GPU when it is instantiated, either by the builder or on deserialization. To select the GPU, use **`cudaSetDevice()`** before calling the builder or deserializing 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.

Q: How do I get the version of TensorRT from the library file?

A: There is a symbol in the symbol table named **tensorrt_version_#_#_#_#** which contains the TensorRT version number. One possible way to read this symbol on Linux is to use the **nm** command like in the example below:

```
$ nm -D libnvinfer.so.4.1.0 | grep tensorrt_version
000000000c18f78c B tensorrt_version_4_0_0_7
```

Q: How do I determine how much device memory will be required by my network?

A: TensorRT uses device memory for two purposes: to hold the weights required by the network, and to hold the intermediate activations. The size of the weights can be closely approximated by the size of the serialized engine (in fact this will be a slight overestimate, as the serialized engine also includes the network definition). The size of the activation memory required can be determined by calling **ICudaEngine::getDeviceMemorySize()**. The sum of these will be the amount of device memory TensorRT allocates.

> The CUDA infrastructure and device code also consume device memory. The amount of memory will vary by platform, device, and TensorRT version. Use **cudaGetMemInfo** to determine the total amount of device memory in use.

5.2. Support

To ask questions and get involved in discussions in all things related to TensorRT, access the NVIDIA DevTalk TensorRT forum at https://devtalk.nvidia.com/default/board/304/tensorrt/.

Appendix A. APPENDIX

A.1. TensorRT Layers

TensorRT directly supports the following layer types:

Activation
: The Activation layer implements per-element activation functions. Supported activation types are rectified linear unit (ReLU) , hyperbolic tangent (tanh), and "s" shaped curve (sigmoid).

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

Constant
: The Constant layer emits a tensor with values provided as parameters to this layer, enabling the convenient use of constants in computations.

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

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

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

Flatten
: The Flatten layer flattens the input while maintaining the **`batch_size`**. Assumes that the first dimension represents the batch. The Flatten layer can only be placed in front of the FullyConnected layer.

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

Gather
: The Gather layer implements the **`gather`** operation, which takes a data tensor, an indices tensor, and a data tensor axis as input and reindexes the data tensor along the

given axis using the indices tensor. Currently, only the TensorRT C++ API supports this layer.

LRN

The LRN layer implements cross-channel Local Response Normalization.

MatrixMultiply

The MatrixMultiply layer implements matrix multiplication for a collection of matrices. A matrix can be transposed or non-transposed before matrix multiplication. Broadcasting is performed, when valid, on unmatched dimensions.

Padding

The Padding layer implements spatial zero-padding of tensors. Padding can be different on each axis, asymmetric, and either positive (resulting in expansion of the tensor) or negative (resulting in trimming).

Plugin

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

Pooling

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

Ragged SoftMax

The Ragged SoftMax layer implements cross-channel Softmax for an input tensor containing sequences of variable lengths. The sequence lengths are specified using a second tensor input to the layer.

Reduce

The Reduce layer implements dimension reduction of tensors using reduce operators. Supported reduce operators are **`prod`**, **`max`**, **`min`**, and **`avg`**. Currently, only the TensorRT C++ API supports this layer.

RNN

This layer type is deprecated in favor of RNNv2, however, it is still available for backwards compatibility.

RNNv2

The RNNv2 layer implements recurrent layers such as Recurrent Neural Network (RNN), Gated Recurrent Units (GRU), and Long Short-Term Memory (LSTM). Supported types are **`RNN`**, **`GRU`**, and **`LSTM`**.

Scale

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

Shuffle

The Shuffle layer implements reshuffling of tensors. It can be used to reshape or transpose data.

SoftMax

The SoftMax layer implements a cross-channel SoftMax.

Squeeze

The Squeeze layer removes dimensions of size 1 from the shape of a tensor. The Squeeze layer only implements the binary squeeze (removing specific size 1 dimensions). The batch dimension cannot be removed.

TopK

The TopK layer finds the top **`K`** elements along a dimension, returning a reduced tensor and a tensor of index positions.

Unary

The Unary layer supports pointwise unary operations. Supported operations are **`exp`**, **`log`**, **`sqrt`**, **`recip`**, **`abs`** and **`neg`**.

- 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 API, rather than via the NVCaffe™ parser library.

For more information about TensorRT layers, see the TensorRT API.

A.2. Command Line Wrapper

Included in the samples directory is a command line wrapper, called *trtexec*, 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
  OR --uff=<file>       UFF file
  --output=<name>       Output blob name (can be specified
 multiple times)

Mandatory params for onnx:
  --onnx=<file>         ONNX Model file

Optional params:
  --uffInput=<name>,C,H,W Input blob names along with their
 dimensions for UFF parser
  --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)
  --percentile=P        For each iteration, report the percentile
 time at P percentage (0<P<=100, default = 99.0%)
  --workspace=N         Set workspace size in megabytes (default =
 16)
  --fp16                Run in fp16 mode (default = false).
 Permits 16-bit kernels
  --int8                Run in int8 mode (default = false).
 Currently no support for ONNX model.
  --verbose             Use verbose logging (default = false)
  --hostTime            Measure host time rather than GPU time
 (default = false)
  --engine=<file>       Generate a serialized TensorRT engine
  --calib=<file>        Read INT8 calibration cache file.
 Currently no support for ONNX model.
```

For example:

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

If no model is supplied, random weights are generated.

A.3. ACKNOWLEDGEMENTS

TensorRT uses elements from the following software, whose licenses are reproduced below:

Google Protobuf

This license applies to all parts of Protocol Buffers except the following:

- Atomicops support for generic gcc, located in **`src/google/protobuf/stubs/atomicops_internals_generic_gcc.h`**. This file is copyrighted by Red Hat Inc.
- Atomicops support for AIX/POWER, located in **`src/google/protobuf/stubs/atomicops_internals_power.h`**. This file is copyrighted by Bloomberg Finance LP.

Code generated by the Protocol Buffer compiler is owned by the owner of the input file used when generating it. This code is not standalone and requires a support library to be linked with it. This support library is itself covered by the above license.

Google Flatbuffers

copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

    a. You must give any other recipients of the Work or Derivative Works a copy of this License; and
    b. You must cause any modified files to carry prominent notices stating that You changed the files; and
    c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
    d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices

that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright 2014 Google Inc.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at: http://www.apache.org/licenses/LICENSE-2.0.

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

BVLC Caffe

COPYRIGHT

All contributions by the University of California:

Copyright (c) 2014, 2015, The Regents of the University of California (Regents) All rights reserved.

All other contributions:

Copyright (c) 2014, 2015, the respective contributors All rights reserved.

Caffe uses a shared copyright model: each contributor holds copyright over their contributions to Caffe. The project versioning records all such contribution and copyright details. If a contributor wants to further mark their specific copyright on a particular contribution, they should indicate their copyright solely in the commit message of the change when it is committed.

half.h

jQuery.js

jQuery.js is generated automatically under doxygen. In all cases TensorRT uses the functions under the MIT license.

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