TENSORRT

SWE-SWDOCTRT-001-DEVG_vTensorRT 5.1.5 | April 2019

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.

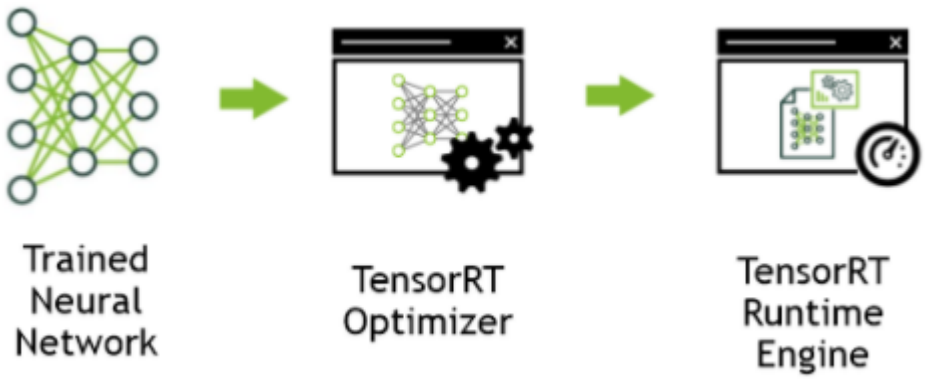


Figure 1 TensorRT is a high performance neural network inference optimizer and runtime engine for production deployment.

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 following figure shows TensorRT defined as part high-performance inference optimizer and part runtime engine. It can take in neural networks trained on these popular frameworks, optimize the neural network computation, generate a light-weight runtime engine (which is the only thing you need to deploy to your production environment), and it will then maximize the throughput, latency, and performance on these GPU platforms.

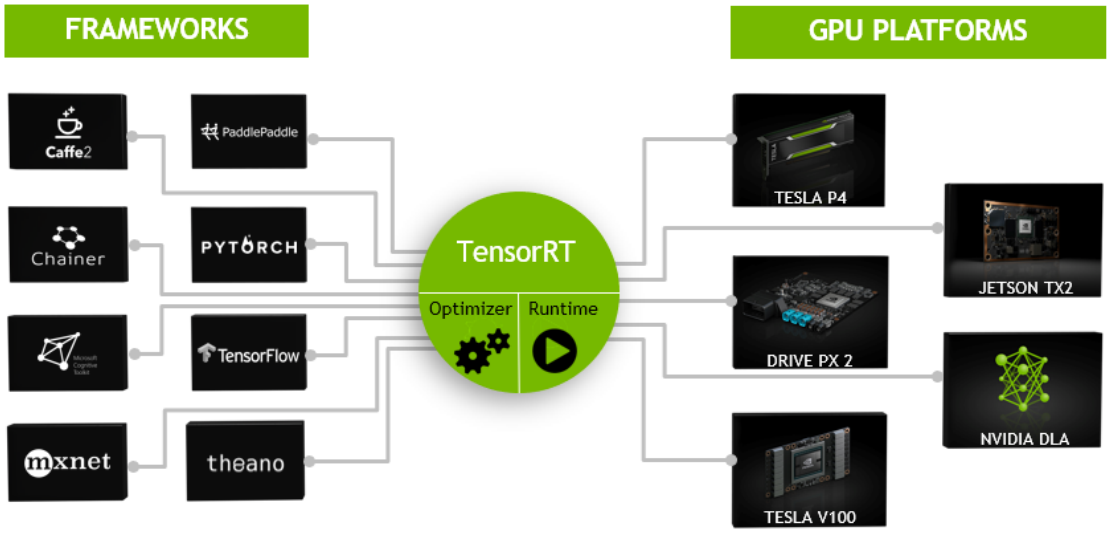


Figure 2 TensorRT is a programmable inference accelerator.

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 C++ Plugin API or Python 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.

1.1.1. Who Can Benefit From TensorRT

TensorRT is intended for use by engineers who are responsible for building features and applications based on new or existing deep learning models or deploying models into production environments. These deployments might be into servers in a datacenter or cloud, in an embedded device, robot or vehicle, or application software which will run on users workstations.

TensorRT has been used successfully across a wide range of scenarios, including:

Robots
Companies sell robots using TensorRT to run various kinds of computer vision models to autonomously guide an unmanned aerial system flying in dynamic environments.

Autonomous Vehicles
TensorRT is used to power computer vision in the NVIDIA Drive products.

Scientific and Technical Computing
A popular technical computing package embeds TensorRT to enable high throughput execution of neural network models.

Deep Learning Training and Deployment Frameworks
TensorRT is included in several popular Deep Learning Frameworks including TensorFlow and MXNet. For TensorFlow and MXNet container release notes, see TensorFlow Release Notes and MXNet Release Notes.

Video Analytics
TensorRT is used in NVIDIA's DeepStream product to power sophisticated video analytics solutions both at the edge with 1 - 16 camera feeds and in the datacenter where hundreds or even thousands of video feeds might come together.

Automatic Speech Recognition
TensorRT is used to power speech recognition on a small tabletop/desktop device. A limited vocabulary is supported on the device with a larger vocabulary speech recognition system available in the cloud.

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? For example, Are you developing a feature or system that is based on a single network (face detection), or will your system be comprised of a mixture or cascade of different models, or perhaps a more general facility that serves up a collection model that may be provided by the end user?
   - What device or compute element will you use to run the network? CPU, GPU, other, or a mixture? If the model is going to run on a GPU, is it a single type of GPU, or do you need to design an application that can run on a variety of GPUs?
   - How is data going to get to the models? What is the data pipeline? Is the data coming in from a camera or sensor, from a series of files, or being uploaded over a network connection?
   - What pre-processing will be done? What format will the data come in? If it is an image does it need to be cropped, rotated? If it is text what character set is it and are all characters allowed as inputs to the model? Are there any special tokens?
   - 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 3), Caffe parser, or TensorFlow/UFF parser.

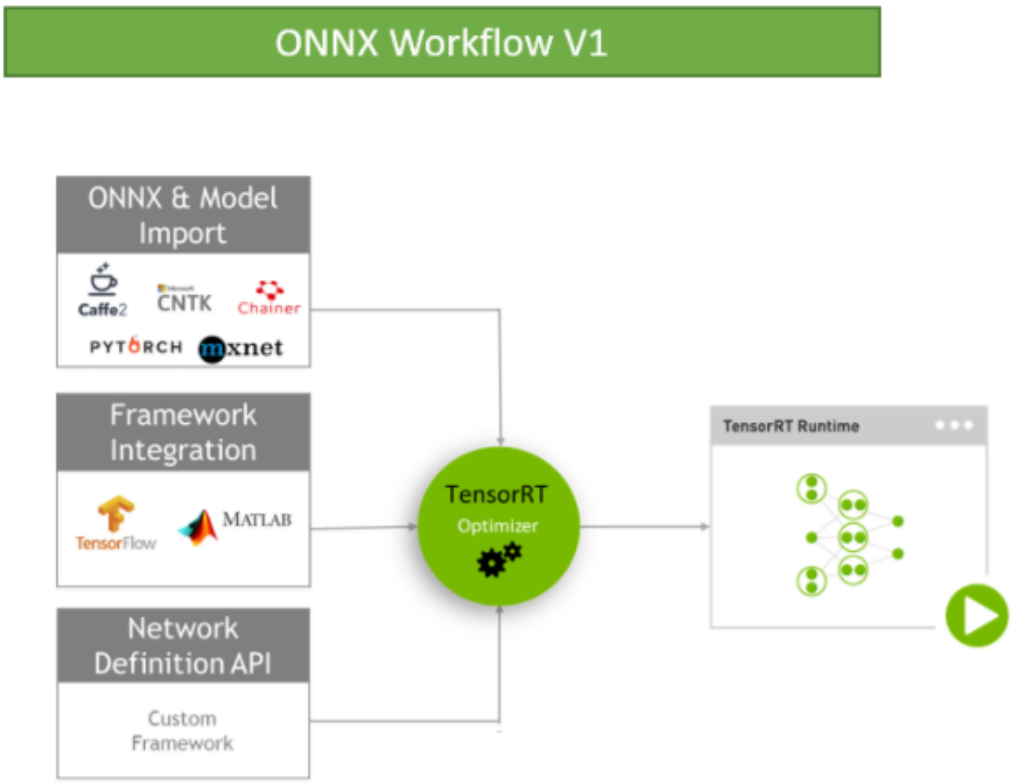


Figure 3 ONNX Workflow V1

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 4).

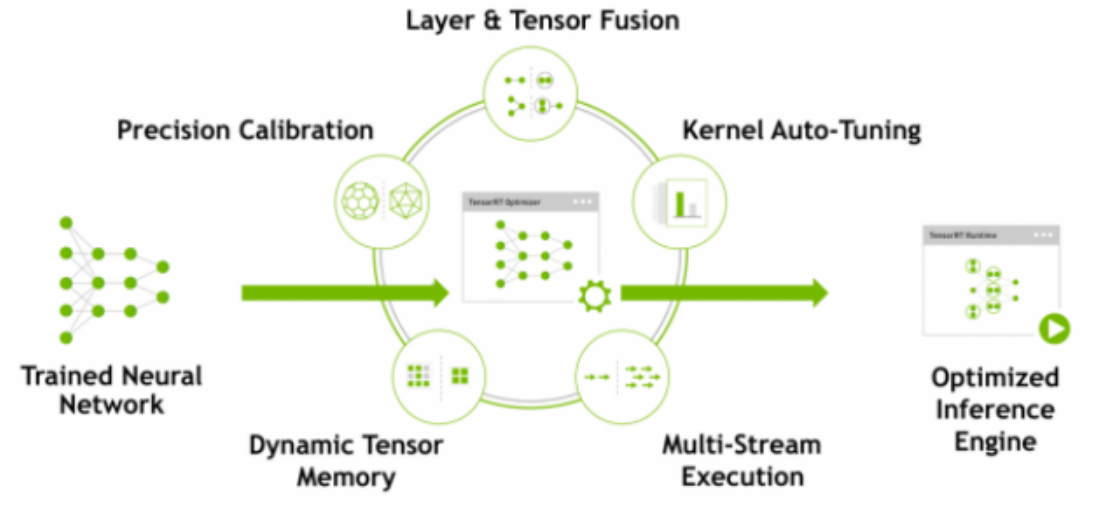


Figure 4 TensorRT optimizes trained neural network models to produce a deployment-ready runtime inference engine.

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 as a plan file for later use.

The generated plan files are not portable across platforms or TensorRT versions. Plans are specific to the exact GPU model they were built on (in addition to platforms and

the TensorRT version) and must be re-targeted to the specific GPU in case you want to run them 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. What Capabilities Does TensorRT Provide?

TensorRT 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.

Users can also run custom layers through TensorRT using the Plugin interface. The graphsurgeon utility provides the ability to map TensorFlow nodes to custom layers in TensorRT, thus enabling inference for many TensorFlow networks with TensorRT.

TensorRT provides a C++ implementation on all supported platforms, and a Python implementation on x86, aarch64, and ppc64le.

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 allows the application to execute 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 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 C++ Caffe Parser, see NvCaffeParser or the Python Caffe Parser.

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 C++ UFF Parser, see NvUffParser or the Python UFF Parser.

ONNX Parser

This parser can be used to parse an ONNX model. For more details on the C++ ONNX Parser, see NvONNXParser or the Python ONNX Parser.

Restriction Since the ONNX format is quickly developing, you may encounter a version mismatch between the model version and the parser version. The ONNX Parser shipped with TensorRT 5.1.x supports ONNX IR (Intermediate Representation) version 0.0.3, opset version 9.

1.5. How Do I Get TensorRT?

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

Chapter 2. WORKING WITH TENSORRT USING THE C++ API

The following sections highlight the TensorRT user goals and tasks that you can perform using the C++ API. Further details are provided in the Samples Support Guide 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

C++ API vs Python API

In essence, the C++ API and the Python API should be close to identical in supporting your needs. The C++ API should be used in any performance critical scenarios, as well as in situations where safety is important, for example, like in automotive.

The main benefit of the Python API is that data preprocessing and postprocessing is easy to use because you're able to use a variety of libraries like NumPy and SciPy. For more information about the Python API, see Using The Python API.

2.1. Instantiating TensorRT Objects in C++

In order to run inference, you need to use the **`IExecutionContext`** object. In order to create an object of type **`IExecutionContext`**, you first need to create an object of type **`ICudaEngine`** (the engine).

The engine can be created in one of two ways:

- via the network definition from the user model. In this case, the engine can be optionally serialized and saved for later use.
- by reading the serialized engine from the disk. In this case, the performance is better, since the steps of parsing the model and creating intermediate objects are bypassed.

An object of type **ILogger** needs to be created globally. It is used as an argument to various methods of TensorRT API. A simple example demonstrating the creation of the logger is shown here:

```
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;
```

A global TensorRT API method called **createInferBuilder(gLogger)** is used to create an object of type **IBuilder** as shown in Figure 5. For more information, see IBuilder class reference.

Figure 5 Creating **IBuilder** with **iLogger** as the input argument

A method called **createNetwork** defined for iBuilder is used to create an object of type **iNetworkDefinition** as shown in Figure 6.

Figure 6 **createNetwork()** is used to create the network

One of the available parsers is created (Caffe, ONNX, or UFF) using the **INetwork** definition as the input:

- ONNX: **auto parser = nvonnxparser::createParser(*network, gLogger);**
- Caffe: **auto parser = nvcaffeparser1::createCaffeParser();**
- UFF: **auto parser = nvuffparser::createUffParser();**

A method called **parse()** from the object of type **iParser** is called to read the model file and populate the TensorRT network Figure 7.

Figure 7 Parsing the model file

A method called **buildCudaEngine()** of **IBuilder** is called to create an object of **ICudaEngine** type as shown in Figure 8:

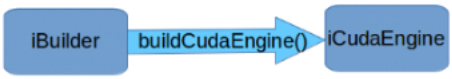


Figure 8 Creating the TensorRT engine

The engine can be optionally serialized and dumped into the file.

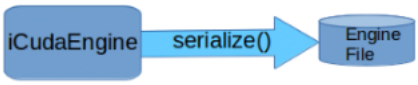


Figure 9 Creating the TensorRT engine

The execution context is used to perform inference.

Figure 10 Creating an execution context

If the serialized engine is preserved and saved to a file, you can bypass most of the steps described above.

A global TensorRT API method called **createInferRuntime(gLogger)** is used to create an object of type **IRuntime** as shown in Figure 11:

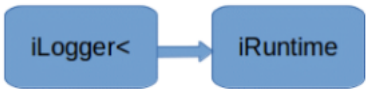


Figure 11 Creating TensorRT runtime

For more information about the TensorRT runtime, see IRuntime class reference. The engine is created by calling the runtime method **deserializeCudaEngine()**.

The rest of the inference is identical for those two usage models.

Even though it is possible to avoid creating the CUDA context, (the default context will be created for you), it is not advisable. It is recommended to create and configure the CUDA context before creating a runtime or builder object.

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:

- https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#mnist_sample (both BVLC and NVCaffe)
- https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#onnx_mnist_sample
- https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#mnist_uff_sample (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. Creating A Network Definition From Scratch 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. To see the code in totality, refer to https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#mnistapi_sample located in the **`/usr/src/tensorrt/samples/sampleMNISTAPI`** directory.

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:

```
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));
```

2.2.2. 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.

```
IBuilder* builder = createInferBuilder(gLogger);
nvinfer1::INetworkDefinition* network = builder->createNetwork();
```

For an example on how to create the logger, see Instantiating TensorRT Objects in C ++.

2. Create the TensorRT parser for the specific format.

ONNX

```
auto parser = nvonnxparser::createParser(*network, gLogger);
```

UFF

```
auto parser = nvuffparser::createUffParser();
```

Caffe

```
auto parser = nvcaffeparser1::createCaffeParser();
```

3. Use the parser to parse the imported model and populate the network.

```
parser->parse(args);
```

The specific **args** depend on what format parser is used. For more information, refer to the parsers documented in the TensorRT API.

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.

2.2.3. 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 https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#mnist_sample.

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()));
```

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

For new projects, it's recommended to use the TensorFlow-TensorRT integration as a method for converting your TensorFlow network to use TensorRT for inference. For integration instructions, see Integrating TensorFlow With TensorRT and its Release Notes.

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 https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#mnist_uff_sample.

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);
```

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

Restriction Since the ONNX format is quickly developing, you may encounter a version mismatch between the model version and the parser version. The ONNX Parser shipped with TensorRT 5.1.x supports ONNX IR (Intermediate Representation) version 0.0.3, opset version 9.

In general, the newer version of the ONNX Parser is designed to be backward compatible, therefore, encountering a model file produced by an earlier version of ONNX exporter should not cause a problem. There could be some exceptions when the changes were not backward compatible. In this case, convert the earlier ONNX model file into a later supported version. For more information on this subject, see ONNX Model Opset Version Converter.

It is also possible that the user model was generated by an exporting tool supporting later opsets than supported by the ONNX parser shipped with TensorRT. In this case, check whether the latest version of TensorRT released to GitHub, `onnx-tensorrt`, supports the required version. The supported version is defined by the `BACKEND_OPSET_VERSION` variable in `onnx_trt_backend.cpp`. Download and build

the latest version of ONNX TensorRT Parser from the GitHub. The instructions for building can be found here: TensorRT backend for ONNX.

The following steps illustrate how to import an ONNX model using the C++ Parser API. For more information about the ONNX import, see https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#onnx_mnist_sample.

1. Create the builder and network.

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

2. Create the ONNX parser.

```
nvonnxparser::IONNXParser* parser =
nvonnxparser::createONNXParser(*network, gLogger);
```

3. Ingest the model:

```
parser->parseFromFile(onnx_filename,
ILogger::Severity::kWARNING);
```

2.3. 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.

```
parser->destroy();
network->destroy();
builder->destroy();
```

2.4. Serializing A Model In C++

To serialize, you are transforming the engine into a format to store and use at a later time for inference. To use for inference, you would simply deserialize the engine. Serializing and deserializing are optional. Since creating an engine from the Network Definition can be time consuming, you could avoid rebuilding the engine every time the application reruns by serializing it once and deserializing it while inferencing. Therefore, after the engine is built, users typically want to serialize it for later use.

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.

2.5. Performing Inference In C++

The following steps illustrate how to perform inference in C++ now that you have an engine.

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.

2.6. 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.

2.7. Refitting An Engine

TensorRT can refit an engine with new weights, without having to rebuild it. The engine must be built as "refittable". Because of the way the engine is optimized, if you change some weights, you may have to supply some other weights too. The interface can tell you what additional weights need to be supplied.

1. Request a refittable engine before building it:

```
...
builder->setRefittable(true);
builder->buildCudaEngine(network);
```

2. Create a refitter object:

```
ICudaEngine* engine = ...;
IRefitter* refitter = createInferRefitter(*engine,gLogger)
```

3. Update the weights that you want to update. For example, to update the kernel weights for a convolution layer named "MyLayer":

```
Weights newWeights = ...;
refitter->setWeights("MyLayer",WeightsRole::kKERNEL,
                     newWeights);
```

   The new weights should have the same count as the original weights used to build the engine.

   setWeights returns false if something went wrong, such as a wrong layer name or role, or a change in the weights count.

4. Find out what other weights must be supplied. This typically requires two calls to **IRefitter::getMissing**, first to get the number of **Weights** objects that must be supplied, and second to get their layers and roles.

```
const int n = refitter->getMissing(0, nullptr, nullptr);
std::vector<const char*> layerNames(n);
std::vector<WeightsRole> weightsRoles(n);
refitter->getMissing(n, layerNames.data(),
                        weightsRoles.data());
```

5. Supply the missing weights, in any order:

```
for (int i = 0; i < n; ++i)
    refitter->setWeights(layerNames[i], weightsRoles[i],
                         Weights{...});
```

   Supplying only the missing weights will not generate a need for any more weights. Supplying any additional weights may trigger the need for yet more weights.

6. Update the engine with all the weights that are provided:

```
bool success = refitter->refitCudaEngine();
assert(success);
```

If **success** is false, check the log for a diagnostic, perhaps about weights that are still missing.

7. Destroy the refitter:

```
refitter->destroy();
```

The updated engine behaves is if it had been built from a network updated with the new weights.

To see all refittable weights in an engine, use **refitter->getAll(...)**; similarly to how **getMissing** was used in step 3.

Chapter 3. USING THE PYTHON API

The following sections highlight the TensorRT user goals and tasks that you can perform using the Python API. These sections focus on using the Python API without any frameworks. Further details are provided in the Samples Support Guide 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

Python API vs C++ API

In essence, the C++ API and the Python API should be close to identical in supporting your needs. The main benefit of the Python API is that data preprocessing and postprocessing is easy to use because you're able to use a variety of libraries like NumPy and SciPy.

The C++ API should be used in situations where safety is important, for example, like in automotive. For more information about the C++ API, see Working With TensorRT Using The C++ API.

For more information about how to optimize performance using Python, see How Do I Optimize My Python Performance? from the Best Practices guide.

3.1. Importing TensorRT Into Python

1. Import TensorRT:

```
import tensorrt as trt
```

2. 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 the TensorRT Python bindings.

```
TRT_LOGGER = trt.Logger(trt.Logger.WARNING)
```

3.2. Creating A Network Definition In Python

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, (see Importing A Model Using A Parser In Python, Importing From Caffe Using Python, Importing From TensorFlow Using Python, and Importing From ONNX Using Python), which supports serialized models in the following formats:

- Caffe (both BVLC and NVCaffe)
- ONNX 1.0 and 1.1, and
- UFF (used for TensorFlow)

An alternative is to define the model directly using the TensorRT Network API, (see Creating A Network Definition From Scratch Using The Python 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.

The TensorRT Python API is not available for all platforms. For more information, see TensorRT Support Matrix

3.2.1. Creating A Network Definition From Scratch 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 about using the Python API to create a network and engine, see the https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#network_api_pytorch_mnist sample.

The following code illustrates how to create a simple network with Input, Convolution, Pooling, FullyConnected, Activation and SoftMax layers.

```
# Create the builder and network
with trt.Builder(TRT_LOGGER) as builder, builder.create_network() as network:
 # Configure the network layers based on the weights provided. In this case, the
 weights are imported from a pytorch model.
 # Add an input layer. The name is a string, dtype is a TensorRT dtype, and the
 shape can be provided as either a list or tuple.
 input_tensor = network.add_input(name=INPUT_NAME, dtype=trt.float32,
 shape=INPUT_SHAPE)
```

```
# Add a convolution layer
conv1_w = weights['conv1.weight'].numpy()
conv1_b = weights['conv1.bias'].numpy()
conv1 = network.add_convolution(input=input_tensor, num_output_maps=20,
kernel_shape=(5, 5), kernel=conv1_w, bias=conv1_b)
conv1.stride = (1, 1)

pool1 = network.add_pooling(input=conv1.get_output(0),
type=trt.PoolingType.MAX, window_size=(2, 2))
pool1.stride = (2, 2)
conv2_w = weights['conv2.weight'].numpy()
conv2_b = weights['conv2.bias'].numpy()
conv2 = network.add_convolution(pool1.get_output(0), 50, (5, 5), conv2_w,
conv2_b)
conv2.stride = (1, 1)

pool2 = network.add_pooling(conv2.get_output(0), trt.PoolingType.MAX, (2, 2))
pool2.stride = (2, 2)

fc1_w = weights['fc1.weight'].numpy()
fc1_b = weights['fc1.bias'].numpy()
fc1 = network.add_fully_connected(input=pool2.get_output(0), num_outputs=500,
kernel=fc1_w, bias=fc1_b)

relu1 = network.add_activation(fc1.get_output(0), trt.ActivationType.RELU)

fc2_w = weights['fc2.weight'].numpy()
fc2_b = weights['fc2.bias'].numpy()
fc2 = network.add_fully_connected(relu1.get_output(0), OUTPUT_SIZE, fc2_w,
fc2_b)

fc2.get_output(0).name =OUTPUT_NAME
network.mark_output(fc2.get_output(0))
```

3.2.2. Importing A Model Using A Parser In Python

To import a model using a parser, 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.

For examples regarding each of these steps and sample code, see Importing From Caffe Using Python, Importing From TensorFlow Using Python, and Importing From ONNX Using Python.

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. For more information, see the UFF Parser API, Caffe Parser API, and ONNX Parser API.

3.2.3. Importing From Caffe Using Python

The following steps illustrate how to import a Caffe model directly using the CaffeParser and the Python API. Refer to the https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#introductory_parser_samples_resnet50 sample for more information.

1. Import TensorRT.

```
import tensorrt as trt
```

2. Define the data type. In this example, we will use float32.

```
datatype = trt.float32
```

3. Additionally, define some paths. Change the following paths to reflect where you placed the model included with the samples:

```
deploy_file = 'data/mnist/mnist.prototxt'
model_file = 'data/mnist/mnist.caffemodel'
```

4. Create the builder, network, and parser:

```
with trt.Builder(TRT_LOGGER) as builder, builder.create_network() as
 network, trt.CaffeParser() as parser:
model_tensors = parser.parse(deploy=deploy_file, model=model_file,
 network=network, dtype=datatype)
```

   The parser returns the **model_tensors**, which is a table containing the mapping from tensor names to **ITensor** objects.

3.2.4. Importing From TensorFlow Using Python

The following steps illustrate how to import a TensorFlow model directly using the UffParser and the Python API. This sample can be found in the **<site-packages>/tensorrt/samples/python/end_to_end_tensorflow_mnist** directory. For more information, see the https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#end_to_end_tensorflow_mnist Python sample.

1. Import TensorRT:

```
import tensorrt as trt
```

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. Typically, this is as simple as:

```
convert-to-uff frozen_inference_graph.pb
```

   Depending on how you installed TensorRT, the **convert-to-uff** utility might not be installed in your system path. In this case, invoke the underlying Python script directly. It should be located in the **bin** directory of the UFF module; for example, **~/.local/lib/python2.7/site-packages/uff/bin/convert_to_uff.py**.

   To find the location of the UFF module, run the **python -c "import uff; print(uff.__path__)"** command.

   Alternatively, you can use the UFF Parser API and convert the TensorFlow GraphDef directly.
4. Define some paths. Change the following paths to reflect where you placed the model that is included with the samples:

```
model_file = '/data/mnist/mnist.uff'
```

5. Create the builder, network, and parser:

```
with builder = trt.Builder(TRT_LOGGER) as builder, builder.create_network()
 as network, trt.UffParser() as parser:
    parser.register_input("Placeholder", (1, 28, 28))
    parser.register_output("fc2/Relu")
parser.parse(model_file, network)
```

3.2.5. Importing From ONNX Using Python

The following steps illustrate how to import an ONNX model directly using the OnnxParser and the Python API. For more information, see the https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#introductory_parser_samples_resnet50 Python sample.

Restriction

Since the ONNX format is quickly developing, you may encounter a version mismatch between the model version and the parser version. The ONNX Parser shipped with TensorRT 5.1.x supports ONNX IR (Intermediate Representation) version 0.0.3, opset version 9.

In general, the newer version of the ONNX Parser is designed to be backward compatible, therefore, encountering a model file produced by an earlier version of ONNX exporter should not cause a problem. There could be some exceptions when the changes were not backward compatible. In this case, convert the earlier ONNX model file into a later supported version. For more information on this subject, see ONNX Model Opset Version Converter.

It is also possible that the user model was generated by an exporting tool supporting later opsets than supported by the ONNX parser shipped with TensorRT. In this case, check whether the latest version of TensorRT released to GitHub, **onnx-tensorrt**, supports the required version. For more information, see https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#yolov3_onnx.

The supported version is defined by the **BACKEND_OPSET_VERSION**variable in **onnx_trt_backend.cpp**. Download and build the latest version of ONNX TensorRT Parser from the GitHub. The instructions for building can be found here: TensorRT backend for ONNX.

1. Import TensorRT:

```
import tensorrt as trt
```

2. Create the build, network, and parser:

```
with builder = trt.Builder(TRT_LOGGER) as builder, builder.create_network()
 as network, trt.OnnxParser(network, TRT_LOGGER) as parser:
with open(model_path, 'rb') as model:
```

```
parser.parse(model.read())
```

3.2.6. 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, (see Creating A Network Definition From Scratch Using The Python API), and then copying the weights from PyTorch. For more information, see Working With PyTorch And Other Frameworks.

On Ubuntu 14.04 and CentOS, loading the torch module and TensorRT at the same time may cause segmentation faults.

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

3.3. 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.

For more information about building an engine in Python, see the https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#introductory_parser_samples_resnet50 sample.

1. Build the engine using the builder object:

```
builder.max_batch_size = max_batch_size
builder.max_workspace_size = 1 <<  20 # This determines the amount of memory
 available to the builder when building an optimized engine and should
 generally be set as high as possible.
with trt.Builder(TRT_LOGGER) as builder:
with builder.build_cuda_engine(network) as engine:
# Do inference here.
```

When the engine is built, TensorRT makes copies of the weights.

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

3.4. Serializing A Model In Python

When you serialize, you are transforming the engine into a format to store and use at a later time for inference. To use for inference, you would simply deserialize the engine. Serializing and deserializing are optional. Since creating an engine from the Network Definition can be time consuming, you could avoid rebuilding the engine every time the application reruns by serializing it once and deserializing it while inferencing. Therefore, after the engine is built, users typically want to serialize it for later use.

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. Serialize the model to a modelstream:

```
serialized_engine = engine.serialize()
```

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

```
with trt.Runtime(TRT_LOGGER) as runtime:
engine = runtime.deserialize_cuda_engine(serialized_engine)
```

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

It is also possible to save a serialized engine to a file, and read it back from the file:

1. Serialize the engine and write to a file:

```
with open("sample.engine", "wb") as f:
  f.write(engine.serialize())
```

2. Read the engine from the file and deserialize:

```
with open("sample.engine", "rb") as f, trt.Runtime(TRT_LOGGER) as runtime:
  engine = runtime.deserialize_cuda_engine(f.read())
```

3.5. Performing Inference In Python

The following steps illustrate how to perform inference in Python, now that you have an engine.

1. Allocate some host and device buffers for inputs and outputs:

```
# Determine dimensions and create page-locked memory buffers (i.e. won't be
 swapped to disk) to hold host inputs/outputs.
  h_input = cuda.pagelocked_empty(engine.get_binding_shape(0).volume(),
 dtype=np.float32)
  h_output = cuda.pagelocked_empty(engine.get_binding_shape(1).volume(),
 dtype=np.float32)
  # Allocate device memory for inputs and outputs.
  d_input = cuda.mem_alloc(h_input.nbytes)
  d_output = cuda.mem_alloc(h_output.nbytes)
  # Create a stream in which to copy inputs/outputs and run inference.
  stream = cuda.Stream()
```

2. 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:

```
with engine.create_execution_context() as context:
  # Transfer input data to the GPU.
  cuda.memcpy_htod_async(d_input, h_input, stream)
  # Run inference.
  context.execute_async(bindings=[int(d_input), int(d_output)],
 stream_handle=stream.handle)
  # Transfer predictions back from the GPU.
  cuda.memcpy_dtoh_async(h_output, d_output, stream)
  # Synchronize the stream
  stream.synchronize()
  # Return the host output.
return h_output
```

   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.

Chapter 4. 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 **`IPluginV2Ext`** class for the C++ and Python 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.

4.1. Adding Custom Layers Using The C++ API

A custom layer is implemented by extending the **`IPluginV2Ext`** and **`IPluginCreator`** classes.

`IPluginV2Ext`
: **`IPluginV2`** is the base class you should implement for your plugins. It includes versioning support and helps enable custom layers that support other data formats besides **`NCHW`** and single precision.

`IPluginCreator`
: **`IPluginCreator`** is a creator class for custom layers using which, users can get plugin name, version and plugin field parameters. It also provides methods to create the plugin object during network build phase and deserialize it during inference.

In previous versions of TensorRT, you implemented **`IPluginV2`** for custom layers. While this API is still supported, we highly encourage you to move to **`IPluginV2Ext`** to be able to use all the new plugin functionalities.

TensorRT also provides the ability to register a plugin by calling **`REGISTER_TENSORRT_PLUGIN(pluginCreator)`** which statically registers the Plugin Creator to the Plugin Registry. During runtime, the Plugin Registry can be queried using the extern function **`getPluginRegistry()`**. The Plugin Registry stores a pointer to all the registered Plugin Creators and can be used to look up a specific Plugin Creator based on the plugin name and version. TensorRT library contains plugins that can be

loaded into your application. The version of all these plugins is set to 1. The names of these plugins are:

- **RPROI_TRT**
- **Normalize_TRT**
- **PriorBox_TRT**
- **GridAnchor_TRT**
- **NMS_TRT**
- **LReLU_TRT**
- **Reorg_TRT**
- **Region_TRT**
- **Clip_TRT**

To use TensorRT registered plugins in your application, the **libnvinfer_plugin.so** library must be loaded and all plugins must be registered. This can be done by calling **initLibNvInferPlugins(void* logger, const char* libNamespace)()** in your application code.

If you have your own plugin library, you can include a similar entry point to register all plugins in the registry under a unique namespace. This ensures there are no plugin name collisions during build time across different plugin libraries.

For more information about these plugins, see the NvInferPlugin.h File reference.

Using the Plugin Creator, the **IPluginCreator::createPlugin()** function can be called which returns a plugin object of type **IPluginV2**. This object can be added to the TensorRT network using **addPluginV2()** 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 (of type **IPluginV2Layer**), which can be used to access the layer or the plugin itself (via **getPlugin()**).

For example, to add a plugin layer to your network with plugin name set to **pluginName** and version set to **pluginVersion**, you can issue the following:

```
//Use the extern function getPluginRegistry to access the global TensorRT Plugin
 Registry
auto creator = getPluginRegistry()->getPluginCreator(pluginName, pluginVersion);
const PluginFieldCollection* pluginFC = creator->getFieldNames();
//populate the field parameters (say layerFields) for the plugin layer
PluginFieldCollection *pluginData = parseAndFillFields(pluginFC, layerFields);
//create the plugin object using the layerName and the plugin meta data
IPluginV2 *pluginObj = creator->createPlugin(layerName, pluginData);
//add the plugin to the TensorRT network using the network API
auto layer = network.addPluginV2(&inputs[0], int(inputs.size()), pluginObj);
… (build rest of the network and serialize engine)
pluginObj->destroy() // Destroy the plugin object
… (destroy network, engine, builder)
```

```
… (free allocated pluginData)
```

pluginData should allocate the **PluginField** entries on the heap before passing to **createPlugin**.

The **createPlugin** method above will create a new plugin object on the heap and return the pointer to it. Ensure you destroy the **pluginObj**, as shown above, to avoid a memory leak.

During serialization, the TensorRT engine will internally store the plugin type, plugin version and namespace (if it exists) for all **IPluginV2** type plugins. During deserialization, this information is looked up by the TensorRT engine to find the Plugin Creator from the Plugin Registry. This enables the TensorRT engine to internally call the **IPluginCreator::deserializePlugin()** method. The plugin object created during deserialization will be destroyed internally by the TensorRT engine by calling **IPluginV2::destroy()** method.

In previous versions of TensorRT, you had to implement the **nvinfer1::IPluginFactory** class to call the **createPlugin** method during deserialization. This is no longer necessary for plugins registered with TensorRT and added using **addPluginV2**.

4.1.1. Example 1: Adding A Custom Layer Using C++ For Caffe

To add a custom layer in C++, implement the **IPluginExt** class. For Caffe based networks, if using the TensorRT Caffe Parser, you will also implement the **nvcaffeparser1::IPluginFactoryExt** (for plugins of type **IPluginExt**) and **nvinfer1::IPluginFactory** classes. For more information, see Using Custom Layers When Importing A Model From A Framework.

The following sample code adds a new plugin called **FooPlugin**:

```
class FooPlugin : public IPluginExt
{
 ...implement all class methods for your plugin
};

class MyPluginFactory : public nvinfer1::IPluginFactory, public
 nvcaffeparser1::IPluginFactoryExt
{
 ...implement all factory methods for your plugin
};
```

If you are using plugins registered with the TensorRT plugin registry of type **IPluginV2**, then you do not need to implement the **nvinfer1::IPluginFactory** class. However, you do need to implement the **nvcaffeparser1::IPluginFactoryV2** and **IPluginCreator** classes instead and register them.

```
class FooPlugin : public IPluginV2
{
 ...implement all class methods for your plugin
};
```

```
class FooPluginFactory : public nvcaffeparser1::IPluginFactoryV2
{
 virtual nvinfer1::IPluginV2* createPlugin(...)
 {
  ...create and return plugin object of type FooPlugin
 }
 bool isPlugin(const char* name)
 {
  ...check if layer name corresponds to plugin
 }
}

class FooPluginCreator : public IPluginCreator
{
 ...implement all creator methods here
};
REGISTER_TENSORRT_PLUGIN(FooPluginCreator);
```

The following samples illustrate how to add a custom plugin layer using C++ for Caffe networks:

- https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#plugin_sample has a user implemented plugin
- https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#fasterrcnn_sample uses plugins registered with the TensorRT Plugin Registry

4.1.2. Example 2: Adding A Custom Layer That Is Not Supported In UFF Using C++

In order to run TensorFlow networks with TensorRT, you must first convert it to the UFF format.

The following steps add a custom plugin layer in C++ for TensorFlow networks:

1. Implement the **`IPluginV2`** and **`IPluginCreator`** classes as shown in Example 1: Adding A Custom Layer Using C++ For Caffe.
2. Map the TensorFlow operation to the plugin operation. You can use graphsurgeon for this. For example, refer to the following code snippet to map the TensorFlow **`Relu6`** operation to a plugin:

   ```
   import graphsurgeon as gs
   my_relu6 = gs.create_plugin_node(name="MyRelu6", op="Clip_TRT", clipMin=0.0,
    clipMax=6.0)
   Namespace_plugin_map = { "tf_relu6" : my_relu6 }
   def preprocess(dynamic_graph):
     dynamic_graph.collapse_namespaces(namespace_plugin_map)
   ```

   In the above code, **`tf_relu6`** is the name of the Relu6 node in the TensorFlow graph. It maps the **`tf_relu6`** node to a custom plugin node with operation "**`Clip_TRT`**" which is the name of the plugin to be used. Save the code above to a file called **`config.py`**. If the plugin layer expects parameters, they should be passed in as arguments to **`gs.create_plugin_node`**. In this case, **`clipMin`** and **`clipMax`** are the parameters expected by the clip plugin.

3. Call the UFF converter with the preprocess **-p** flag set:

```
convert-to-uff frozen_inference_graph.pb -p config.py -t
```

   This will generate a UFF file with the TensorFlow operations replaced by TensorRT plugin nodes.

4. Run the pre-processed and converted UFF file with TensorRT using the UFF parser. For details, see Using Custom Layers When Importing A Model From A Framework.

   The https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#uffssd_sample sample illustrates how to add a custom layer that is not supported in UFF using C++. See **config.py** in the sample folder for a demonstration of how to pre-process the graph.

4.2. Adding Custom Layers Using The Python API

Although the C++ API is the preferred language to implement custom layers; due to easily accessing libraries like CUDA and cuDNN, you can also work with custom layers in a Python applications.

You can use the C++ API to create a custom layer, package the layer using **pybind11** in Python, then load the plugin into a Python application. For more information, see Creating A Network Definition In Python.

The same custom layer implementation can be used for both C++ and Python. For more information, see the https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#fc_plugin_caffe_mnist Python sample located in the **/usr/src/tensorrt/samples/fc_plugin_caffe_mnist/** directory.

4.2.1. Example 1: Adding A Custom Layer to a TensorRT Network Using Python

Custom layers can be added to any TensorRT network in Python using plugin nodes. The Python API has a function called **add_plugin_v2** which enables you to add a plugin node to a network. The following example illustrates this. It creates a simple TensorRT network and adds a Leaky ReLU plugin node by looking up TensorRT Plugin Registry.

```
import tensorrt as trt
import numpy as np

TRT_LOGGER = trt.Logger()

trt.init_libnvinfer_plugins(TRT_LOGGER, '')
PLUGIN_CREATORS = trt.get_plugin_registry().plugin_creator_list

def get_trt_plugin(plugin_name):
        plugin = None
        for plugin_creator in PLUGIN_CREATORS:
            if plugin_creator.name == plugin_name:
                lrelu_slope_field = trt.PluginField("neg_slope", np.array([0.1],
 dtype=np.float32), trt.PluginFieldType.FLOAT32)
```

```
                field_collection =
trt.PluginFieldCollection([lrelu_slope_field])
                plugin = plugin_creator.create_plugin(name=plugin_name,
field_collection=field_collection)
        return plugin

def main():
    with trt.Builder(TRT_LOGGER) as builder, builder.create_network() as
network:
        builder.max_workspace_size = 2**20
        input_layer = network.add_input(name="input_layer", dtype=trt.float32,
shape=(1, 1))
        lrelu = network.add_plugin_v2(inputs=[input_layer],
plugin=get_trt_plugin("LReLU_TRT"))
        lrelu.get_output(0).name = "outputs"
        network.mark_output(lrelu.get_output(0))
```

4.2.2. Example 2: Adding A Custom Layer That Is Not Supported In UFF Using Python

TensorFlow networks can be converted to UFF format and run with TensorRT using the Python interface. In order to do this, we make use of the graphsurgeon API. If you are writing your own plugin, you need to implement it in C++ by implementing the **IPluginExt** and **IPluginCreator** classes as shown in Example 1: Adding A Custom Layer Using C++ For Caffe.

The following steps illustrate how you can use the UFF Parser to run custom layers using plugin nodes registered with the TensorRT Plugin Registry.

1. Register the TensorRT plugins by calling **trt.init_libnvinfer_plugins(TRT_LOGGER, '')** (or load the **.so** file where you have registered your own plugin).
2. Prepare the network and check the TensorFlow output:

```
tf_sess = tf.InteractiveSession()
tf_input = tf.placeholder(tf.float32, name="placeholder")
tf_lrelu = tf.nn.leaky_relu(tf_input, alpha=lrelu_alpha, name="tf_lrelu")
tf_result = tf_sess.run(tf_lrelu, feed_dict={tf_input: lrelu_args})
tf_sess.close()
```

3. Prepare the namespace mappings. The **op** name **LReLU_TRT** corresponds to the Leaky ReLU plugin shipped with TensorRT.

```
trt_lrelu = gs.create_plugin_node(name="trt_lrelu", op="LReLU_TRT",
 negSlope=lrelu_alpha)
namespace_plugin_map = {
            "tf_lrelu": trt_lrelu
 }
```

4. Transform the TensorFlow graph using graphsurgeon and save to UFF:

```
dynamic_graph = gs.DynamicGraph(tf_lrelu.graph)
dynamic_graph.collapse_namespaces(namespace_plugin_map)
```

5. Run the UFF parser and compare results with TensorFlow:

```
uff_model = uff.from_tensorflow(dynamic_graph.as_graph_def(), ["trt_lrelu"],
 output_filename=model_path, text=True)
parser = trt.UffParser()
```

```
parser.register_input("placeholder", [lrelu_args.size])
parser.register_output("trt_lrelu")
parser.parse(model_path, trt_network)
```

For more information, see the https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#uff_custom_plugin sample.

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

TensorRT parsers use the layer operation field to identify if a particular layer in the network is a TensorRT supported operation.

TensorFlow

Compared to previous releases of TensorRT, there are several changes with how custom layers in TensorFlow can be run with the TensorRT UFF parser. For TensorFlow models, use the UFF converter to convert your graph to a UFF file. In this process, if the network contains plugin layers it is also necessary to map the operation field of those layers to the corresponding registered plugin names in TensorRT. These plugins can either be plugins shipped with TensorRT or custom plugins that you have written. The plugin field names in the network should also match the fields expected by the plugin. This can be done using graphsurgeon, as explained in Preprocessing A TensorFlow Graph Using the Graph Surgeon API and as demonstrated in sampleUffSSD by using a config file with the UFF converter.

The UFF Parser will look up the Plugin Registry for every unsupported operation. If it finds a match with any of the registered plugin names, the parser will parse the plugin field parameters from the input network and create a plugin object using them. This object is then added to the network. In previous versions of TensorRT, you had to implement the **`nvuffparser::IPluginFactoryExt`** and manually pass the plugin parameters to the **`createPlugin(...)`** function. Although this flow can still be exercised, it is no longer necessary with the new additions to the Plugin API. For more information, see:

- **`IPluginV2Ext`** and **`IPluginCreator`** in the C++ API
- **`IPluginV2Ext`** and **`IPluginCreator`** in the Python API

Caffe

For Caffe models, use the **`nvcaffeparser1::IPluginFactoryV2`** class. The **`setPluginFactoryV2`** 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 **`isPluginV2`** 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 instantiate the corresponding plugin), a **`Weights`** array, and

the number of weights 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.

For the Caffe parser, if **`setPluginFactoryV2`** and **`IPluginFactoryV2`** are used, the plugin object created during deserialization will be internally destroyed by the engine by calling **`IPluginExt::destroy()`**. You are only responsible for destroying the plugin object created during network creation step as shown in Adding Custom Layers Using The C++ API.

The samplePlugin sample illustrates how to extend **`nvcaffeparser1::IPluginFactoryExt`** to use custom layers, while sampleUffSSD uses the UFF Parser to use custom layers.

For the Python usage of custom layers with TensorRT, refer to the https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#fc_plugin_caffe_mnist sample for Caffe networks, and the https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#uff_custom_plugin and https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#uff_ssd samples for UFF networks.

4.3.1. Example 1: Adding A Custom Layer To A TensorFlow Model

In order to run a TensorFlow network with TensorRT, you must first convert it to the UFF format. During the conversion process, custom layers can be marked as plugin nodes using the **`graphsurgeon`** utility.

The UFF converter then converts the processed graph to the UFF format which is then run by the UFF Parser. The plugin nodes are then added to the TensorRT network by the UFF Parser.

For details using the C++ API, see Example 2: Adding A Custom Layer That Is Not Supported In UFF Using C++.

For details using the Python API, see Example 2: Adding A Custom Layer That Is Not Supported In UFF Using Python. Additionally, the https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#uff_ssd Python sample demonstrates an end-to-end workflow in Python for running TensorFlow object detection networks using TensorRT.

4.4. Plugin API Description

All new plugins should implement both the **`IPluginV2Ext`** and **`IPluginCreator`** classes. In addition, new plugins should also call the **`REGISTER_TENSORRT_PLUGIN(...)`** macro to register the plugin with the TensorRT Plugin Registry or create an **`init`** function equivalent to **`initLibNvInferPlugins()`**.

4.4.1. Migrating Plugins From TensorRT 5.x.x To TensorRT 5.1.x

In order to use the most recent Plugin layer features, your custom plugin should implement the IPluginV2Ext interface. The new features are as follows:

```
virtual nvinfer1::DataType getOutputDataType(int index, const
 nvinfer1::DataType* inputTypes, int nbInputs) const = 0;


virtual bool isOutputBroadcastAcrossBatch(int outputIndex, const bool*
 inputIsBroadcasted, int nbInputs) const = 0;

virtual bool canBroadcastInputAcrossBatch(int inputIndex) const = 0;


virtual void configurePlugin(const Dims* inputDims, int nbInputs, const Dims*
 outputDims,
            int nbOutputs, const DataType* inputTypes, const DataType*
 outputTypes, const bool* inputIsBroadcast, const bool* outputIsBroadcast,
 PluginFormat floatFormat, int maxBatchSize) = 0;
```

For the simplest migration, follow these guidelines:

- **`getOutputDataType`** can return the type of the input (from **`inputTypes`**) or **`DataType::kFLOAT`** if the layer has no inputs.
- **`isOutputBroadcastAcrossBatch`** can return **`false`** if the plugin does not support output broadcast.
- **`canBroadcastInputAcrossBatch`** can return **`false`** if the plugin cannot handle broadcasted inputs.
- **`configurePlugin`** can mimic the behavior of **`configureWithFormat`**.

See the API description in IPluginV2 API Description for details about the API.

4.4.2. `IPluginV2` API Description

The following section describes the functions of the **`IPluginV2`** class.

To connect a plugin layer to neighboring layers and setup input and output data structures, the builder checks for the number of outputs and their dimensions by calling the following 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.

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

`getOutputDataType`
: Used to get the data type of the output at a given index. The returned data type must have a format that is supported by the plugin.

Plugin layers can support four data formats and layouts, for example:

- **NCHW** single (FP32), half precision (FP16) and integer (INT32) tensors
- **NC/2HW2** and **NHWC8** half precision (FP16) 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 times 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:

configurePlugin
: Communicates the number of inputs and outputs, dimensions and datatypes of all inputs and outputs, broadcast information for all inputs and outputs, the chosen plugin 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.

clone
: This is called every time a new builder, network or engine is created which includes this plugin layer. It should return a new plugin object with the correct parameters.

destroy
: Used to destroy the plugin object and/or other memory allocated each time a new plugin object is created. It is called whenever the builder or network or engine is destroyed.

set/getPluginNamespace
: This method is used to set the library namespace that this plugin object belongs to (default can be ""). All plugin objects from the same plugin library should have the same namespace.

IPluginV2Ext supports plugins that can handle broadcast inputs and outputs. The following methods need to be implemented for this feature:

canBroadcastInputAcrossBatch
: This method is called for each input whose tensor is semantically broadcast across a batch. If **canBroadcastInputAcrossBatch** returns **true** (meaning the plugin can

support broadcast), TensorRT will not replicate the input tensor. There will be a single copy that the plugin should share across the batch. If it returns **`false`**, TensorRT will replicate the input tensor so that it appears like a non-broadcasted tensor.

`isOutputBroadcastAcrossBatch`
This is called for each output index. The plugin should return true the output at the given index is broadcast across batch.

4.4.3. `IPluginCreator` API Description

The following methods in the **`IPluginCreator`** class are used to find and create the appropriate plugin from the Plugin Registry:

`getPluginName`
This returns the plugin name and should match the return value of **`IPluginExt::getPluginType`**.

`getPluginVersion`
Returns the plugin version. For all internal TensorRT plugins, this defaults to **`1`**.

`getFieldNames`
In order to successfully create a plugin, it is necessary to know all the field parameters of the plugin. This method returns the **`PluginFieldCollection`** struct with the **`PluginField`** entries populated to reflect the field name and **`PluginFieldType`** (the data should point to **`nullptr`**).

`createPlugin`
This method is used to create the plugin using the **`PluginFieldCollection`** argument. The data field of the **`PluginField`** entries should be populated to point to the actual data for each plugin field entry.

`deserializePlugin`
This method is called internally by the TensorRT engine based on the plugin name and version. It should return the plugin object to be used for inference.

`set/getPluginNamespace`
This method is used to set the namespace that this creator instance belongs to (default can be "").

4.5. Best Practices For Custom Layers

Converting User-Defined Layers

To create a custom layer implementation as a TensorRT plugin, you need to implement the **`IPluginV2Ext`** class and the **`IPluginCreator`** class for your plugin.

For more information about both API classes, see Plugin API Description.

For Caffe networks, see Example 1: Adding A Custom Layer Using C++ For Caffe.
For TensorFlow (UFF) networks, see Example 2: Adding A Custom Layer That Is Not Supported In UFF Using C++.

Using The UFF Plugin API

For an example of how to use plugins with UFF in both C++ and Python, see Example 1: Adding A Custom Layer Using C++ For Caffe and Example 2: Adding A Custom Layer That Is Not Supported In UFF Using Python.

Debuggin Custom Layer Issues

Memory allocated in the plugin must be freed to ensure no memory leak. If resources are acquired in the **`initialize()`** function, they need to be released in the **`terminate()`** function. All other memory allocations should be freed preferably in the plugin class destructor or in the **`destroy()`** method. Adding Custom Layers Using The C++ API outlines this in detail and also provides some notes for best practices when using plugins.

Chapter 5. 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.

You can check the supported precision on a platform using the following APIs:

```
if (builder->platformHasFastFp16()) { … };
if (builder->platformHasFastInt8()) { … };
```

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. You can set the builder flag **`setStrictTypeConstraints`** to force the network or layer precision, which may not have optimal performance. Usage of this flag is only recommended for debugging purposes.

You can also choose to set both INT8 and FP16 mode if the platform supports it. TensorRT will choose the most performance optimal kernel to perform inference.

5.1. Mixed Precision Using The C++ API

5.1.1. Setting The Layer Precision Using C++

If you want to run certain layers a specific precision, you can set the precision per layer using the following API:

```
layer->setPrecision(nvinfer1::DataType::kINT8)
```

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

```
layer->setOutputType(out_tensor_index, nvinfer1::DataType::kFLOAT)
```

This method cannot be used to set the data type of the second output tensor of the TopK layer. The data type of the second output tensor of the TopK layer is always INT32. For more information, see TopK Layer.

TensorRT has very few implementations that run in heterogeneous precision: in TensorRT 5.x.x the only ones are INT8 implementations for Convolution, Deconvolution, and FullyConnected layers that produce FP32 output.

Setting the precision, requests TensorRT to use a layer implementation whose inputs and outputs matches the preferred types, inserting reformat operations if necessary. By default, TensorRT will choose such an implementation only if it results in a higher-performance network. If an implementation at a higher precision is faster, TensorRT will use it, and issue a warning. Thus, you can detect whether using lower precision would result in unexpected performance loss.

You can override this behavior by making the type constraints *strict*.

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

If the constraints are strict, TensorRT will obey them unless there is no implementation with the preferred precision constraints, in which case it will issue a warning and use the fastest available implementation.

If the precision is not explicitly set, TensorRT will select the computational precision based on performance considerations and the flags specified to the builder.

See https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#int8_api_sample for an example of running mixed precision inference with these APIs.

5.1.2. 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. You can choose to force 16-bit precision by setting the following builder flag:

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

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

See https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#googlenet_sample and https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#mnist_sample for examples of running FP16 inference.

5.1.3. Enabling INT8 Inference Using C++

Setting the builder flag enables INT8 precision inference.

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

In order to perform INT8 inference, FP32 activation tensors and weights need to be quantized. In order to represent 32-bit floating point values and INT 8-bit quantized values, TensorRT needs to understand the dynamic range of each activation tensor. The dynamic range is used to determine the appropriate quantization scale.

TensorRT supports symmetric quantization with quantization scale calculated using absolute maximum dynamic range values.

TensorRT needs the dynamic range for each tensor in the network. There are two ways in which the dynamic range can be provided to the network:

- manually set the dynamic range for each network tensor using **`setDynamicRange`** API

Or

- use INT8 calibration to generate per tensor dynamic range using the calibration dataset.

The dynamic range API can also be used along with INT8 calibration, such that manually setting the range will take precedence over the calibration generated dynamic range. Such scenario is possible if INT8 calibration does not generate a satisfactory dynamic range for certain tensors.

For more information, see https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#int8_api_sample.

5.1.3.1. Setting Per-Tensor Dynamic Range Using C++

You can generate per tensor the dynamic range using various techniques. The basic technique includes recording per tensor the min and max values during the last epoch of training, or using quantization aware training. TensorRT expects you to set the dynamic range for each network tensor to perform INT8 inference. After you have the dynamic range information, you can set the dynamic range as follows:

```
ITensor* tensor = network->getLayer(layer_index)->getOutput(output_index);
tensor->setDynamicRange(min_float, max_float);
```

You also need to set the dynamic range for the network input:

```
ITensor* input_tensor = network->getInput(input_index);
input_tensor->setDynamicRange(min_float, max_float);
```

One way to achieve this, is to iterate through the network layers and tensors and set per tensor the dynamic range. TensorRT only supports symmetric range currently, therefore, only **`abs(min_float)`** and **`abs(max_float)`** is used for quantization. For more information, see https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#int8_api_sample.

5.1.3.2. INT8 Calibration Using C++

INT8 calibration provides an alternative to generate per activation tensor the dynamic range. This methods can be categorized as post training technique to generate the 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. TensorRT provides multiple variants of **`IInt8Calibrator`**:

`IEntropyCalibratorV2`
: This is the preferred calibrator and is required for DLA as it supports per activation tensor scaling.

`IEntropyCalibrator`
: This is the legacy entropy calibrator which supports per channel scaling. This is less complicated than legacy calibrator and produces better results.

`ILegacyCalibrator`
: This calibrator is for compatibility with 2.0EA. It is deprecated and should not be used.

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->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 https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#int8_sample.

5.2. Mixed Precision Using The Python API

5.2.1. Setting The Layer Precision Using Python

In Python, you can specify the layer precision using the **`precision`** flag:

```
layer.precision = trt.int8
```

You can set the output tensor data type to conform with the layer implementation:

```
layer.set_output_type(out_tensor_index, trt.int8)
```

Ensure that the builder understands to force the precision:

```
builder.strict_type_constraints = true
```

5.2.2. Enabling FP16 Inference Using Python

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

```
builder.fp16_mode = True
```

Force 16-bit precision by setting the builder flag:

```
builder.strict_type_constraints = True
```

5.2.3. Enabling INT8 Inference Using Python

Enable INT8 mode by setting the builder flag:

```
trt_builder.int8_mode = True
```

Similar to the C++ API, you can choose per activation tensor the dynamic range either using **`set_dynamic_range`** or using INT8 calibration.

INT8 calibration can be used along with the dynamic range APIs. Setting the dynamic range manually will override the dynamic range generated from INT8 calibration.

5.2.3.1. Setting Per-Tensor Dynamic Range Using Python

In order to perform INT8 inference, you must set the dynamic range for each network tensor. You can derive the dynamic range values using various methods including quantization aware training or simply recording per tensor the min and max values during the last training epoch. To set the dynamic range use:

```
layer = network[layer_index]
tensor = layer.get_output(output_index)
tensor.set_dynamic_range(min_float, max_float)
```

You also need to set the dynamic range for the network input:

```
input_tensor = network.get_input(input_index)
input_tensor.set_dynamic_range(min_float, max_float)
```

5.2.3.2. INT8 Calibration Using Python

INT8 calibration provides an alternative approach to generate per activation tensor the dynamic range. This method can be categorized as a post training technique to generate the appropriate quantization scale.

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

1. Import TensorRT:

   ```
   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 = EntropyCalibrator(["input_node_name"], batchstream)
   ```

4. Set INT8 mode and INT8 Calibrator:

   ```
   trt_builder.int8_calibrator = Int8_calibrator
   ```

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

Chapter 6. WORKING WITH DLA

NVIDIA DLA (Deep Learning Accelerator) is a fixed function accelerator engine targeted for deep learning operations. DLA is designed to do full hardware acceleration of convolutional neural networks. DLA supports various layers such as convolution, deconvolution, fully-connected, activation, pooling, batch normalization, etc.

For more information about DLA support in TensorRT layers, see DLA Supported Layers. The **trtexec** tool has additional arguments to run networks on DLA, see Command Line Program.

To run the AlexNet network on DLA using **trtexec** in FP16 mode, issue:

```
 ./trtexec --deploy=data/AlexNet/AlexNet_N2.prototxt --output=prob --
useDLACore=1 --fp16 --allowGPUFallback
```

To run the AlexNet network on DLA using **trtexec** in INT8 mode, issue:

```
 ./trtexec --deploy=data/AlexNet/AlexNet_N2.prototxt --output=prob --
useDLACore=1 --int8 --allowGPUFallback
```

6.1. Running On DLA During TensorRT Inference

The TensorRT builder can be configured to enable inference on DLA. DLA support is currently limited to networks running in either FP16 or INT8 mode. The **DeviceType** enumeration is used to specify the device that the network or layer will execute on. The following API functions in the **IBuilder** class can be used to configure the network to use DLA:

setDeviceType(ILayer* layer, DeviceType deviceType)
This function can be used to set the **deviceType** that the layer must execute on.

getDeviceType(const ILayer* layer)
This function can be used to return the **deviceType** that this layer will execute on. If the layer is executing on the GPU, this will return **DeviceType::kGPU**.

canRunOnDLA(const ILayer* layer)
This function can be used to check if a layer can run on DLA.

`setDefaultDeviceType(DeviceType deviceType)`
This function sets the default **`deviceType`** to be used by the builder. It ensures that all the layers that can run on DLA will run on DLA, unless **`setDeviceType`** is used to override the **`deviceType`** for a layer.

`getDefaultDeviceType()`
This function returns the default **`deviceType`** which was set by **`setDefaultDeviceType`**.

`isDeviceTypeSet(const ILayer* layer)`
This function checks whether the **`deviceType`** has been explicitly set for this layer.

`resetDeviceType(ILayer* layer)`
This function resets the **`deviceType`** for this layer. The value is reset to the **`deviceType`** that is specified by **`setDefaultDeviceType`** or **`DeviceType::kGPU`** if none specified.

`getMaxDLABatchSize(DeviceType deviceType)`
This function returns the maximum batch size DLA can support.

For any tensor, the total volume of index dimensions combined with the requested batch size should not exceed the value returned by this function.

`allowGPUFallback(bool setFallBackMode)`
This function notifies the builder to use GPU if a layer that was supposed to run on DLA cannot run on DLA. For more information, see GPU Fallback Mode.

`reset(nvinfer1::INetworkDefinition& network)`
This function can be used to reset the builder state, which sets the **`deviceType`** for all layers to be **`DeviceType::kGPU`**. After reset, the builder can be re-used to build another network with a different DLA config.

Caution In TensorRT 5.x.x, this resets the state for all networks and not the current network.

If the builder is not accessible, such as in the case where a plan file is being loaded online in an inference application, then the DLA to be utilized can be specified differently by using DLA extensions to the **`IRuntime`**. The following API functions in the **`IRuntime`** class can be used to configure the network to use DLA:

`getNbDLACores()`
This function returns the number of DLA cores that are accessible to the user.

`setDLACore(int dlaCore)`
The DLA core to execute on. Where **`dlaCore`** is a value between **`0`** and **`getNbDLACores() - 1`**. The default value is **`0`**.

6.1.1. Example 1: sampleMNIST With DLA

This section provides details on how to run a TensorRT sample with DLA enabled. The https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#mnist_sample sample demonstrates how to import a trained Caffe model, build the TensorRT engine, serialize and deserialize the engine and finally use the engine to perform inference.

The sample first creates the builder:

```
auto builder =
 SampleUniquePtr<nvinfer1::IBuilder>(nvinfer1::createInferBuilder(gLogger));
if (!builder) return false;
builder->setMaxBatchSize(batchSize);
builder->setMaxWorkspaceSize(16_MB);
```

Then, enable **GPUFallback** mode:

```
builder->allowGPUFallback(true);
builder->setFp16Mode(true); or builder->setInt8Mode(true);
```

Enable execution on DLA, where **deviceType** specifies the DLA core to execute on:

```
builder->setDefaultDeviceType(deviceType);
```

With these additional changes, sampleMNIST is ready to execute on DLA. To run sampleMNIST with DLA, use the following command:

```
 ./sample_mnist --useDLACore=1 [--int8|--fp16]
```

6.1.2. Example 2: Enable DLA Mode For A Layer During Network Creation

In this example, let's create a simple network with Input, Convolution and Output.

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.

   ```
   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.

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

4. Set the convolution layer to run on DLA:

   ```
   if(canRunOnDLA(conv1))
   {
   builder->setFp16Mode(true); or builder->setInt8Mode(true);
   builder->setDeviceType(conv1, DeviceType::kDLA);

   }
   ```

5. Mark the output:

   ```
   network->markOutput(*conv1->getOutput(0));
   ```

6. Set the DLA engine to execute on:

   ```
   engine->setDLACore(0)
   ```

6.2. DLA Supported Layers

This section lists the layers supported by DLA along with the constraints associated with each layer.

Generic restrictions while running on DLA (applicable to all layers)

- Max batch size supported is 32.

Batch size for DLA is the product of all index dimensions except the **CHW** dimensions. For example, if input dimensions are **NPQRS**, the effective batch size is **N*P**.

- Input and output tensor data format should be FP16.

Layer specific restrictions

Convolution, Deconvolution, and Fully Connected Layers

Convolution and Deconvolution Layers

- Width and height of kernel size must be in the range [1, 32]
- Width and height of padding must be in the range [0, 31]
- Width and height of stride must be in the range [1,8] for Convolution Layer and [1,32] for Deconvolution layer
- Number of output maps must be in the range [1, 8192]
- Axis must be 1
- Grouped and dilated convolution supported. Dilation values must be in the range [1,32]

Pooling Layer

- Operations supported: **kMIN**, **kMAX**, **kAVERAGE**
- Width and height of the window size must be in the range [1, 8]
- Width and height of padding must be in the range [0, 7]
- Width and height of stride must be in the range [1, 16]

Activation Layer

- Functions supported: **ReLU**, **Sigmoid**, **Hyperbolic Tangent**
  - Negative slope not supported for ReLU

ElementWise Layer

- Operations supported: **Sum**, **Product**, **Max**, and **Min**

Scale Layer

- Mode supported: **`Uniform`**, **`Per-Channel`**, and **`Elementwise`**

LRN (Local Response Normalization) Layer

- Window size is configurable to 3, 5, 7, or 9
- Normalization region supported is: **`ACROSS_CHANNELS`**

Concatenation Layer

- DLA supports concatenation only along the channel axis

6.3. GPU Fallback Mode

The **`GPUFallbackMode`** sets the builder to use GPU if a layer that was marked to run on DLA could not run on DLA. A layer may not run on DLA due to the following reasons:

1. The **`layer`** operation is not supported on DLA.
2. The parameters specified are out of supported range for DLA.
3. The given batch size exceeds the maximum permissible DLA batch size. For more information, see DLA Supported Layers.
4. A combination of layers in the network causes the internal state to exceed what the DLA is capable of supporting.
5. There are no DLA engines available on the platform.

If the **`GPUFallbackMode`** is set to **`false`**, a layer set to execute on DLA, that couldn't run on DLA will result in an error. However, with **`GPUFallbackMode`** set to **`true`**, it will continue to execute on the GPU instead, after reporting a warning.

Similarly, if **`defaultDeviceType`** is set to **`DeviceType::kDLA`** and **`GPUFallbackMode`** is set to **`false`**, it will result in an error if any of the layers can't run on DLA. With **`GPUFallbackMode`** set to **`true`**, it will report a warning and continue executing on the GPU.

Chapter 7. 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.

7.1. 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 TensorRT Inference Server Container Release Notes and TensorRT Inference Server Guide.

7.2. 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 8. 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.

8.1. Working With TensorFlow

You can work with TensorFlow in the following ways:

TF-TRT

This method accelerates a TensorFlow graph with TensorRT even if there are TensorFlow operators in the graph that are not supported by TensorRT (or TF-TRT). The subgraphs that are supported by TensorRT and TF-TRT are accelerated and the resulting graph is still a TensorFlow graph that you can execute as usual. For step-by-step instructions on how to accelerate inference in TensorFlow with TensorRT (TF-TRT), see the TF-TRT User Guide and Release Notes. For TF-TRT examples, see Examples for TensorRT in TensorFlow (TF-TRT).

UFF

This method works only if the whole graph can be converted to UFF and can be accelerated by TensorRT. For information on using TensorRT with a TensorFlow model, see the "Hello World" For TensorRT Using TensorFlow And Python Python sample.

The UFF Parser which is used to parse a network in UFF format will be deprecated in a future release. The recommended method of importing TensorFlow models to TensorRT is using TensorFlow with TensorRT (TF-TRT).

8.1.1. 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!

8.1.2. 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, "")
```

8.1.3. 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 input_file [-o output_file] [-O output_node]
```

You can list the TensorFlow layers:

```
convert-to-uff input_file -l
```

8.1.4. 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.

8.1.4.1. 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.

8.1.4.2. 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.

8.1.4.3. Workflow

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

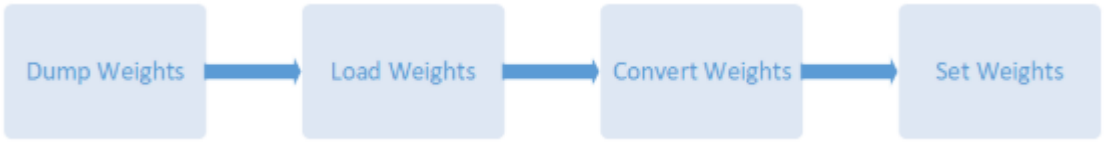


Figure 12 TensorFlow RNN Workflow

8.1.4.4. 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.

8.1.4.5. Loading Dumped Weights

Function **loadWeights()** loads from the dump of the **dumpTFWts.py** script. It has been provided as an example in https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#charRNN_sample. 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>**.

8.1.4.6. 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.

8.1.4.6.1. 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.

8.1.4.6.2. 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**)

8.1.4.6.3. 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 **`NvUtils.h`**.

8.1.4.6.4. 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.

8.1.4.6.5. 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.

8.1.4.7. **`BasicLSTMCell`** Example

8.1.4.7.1. BasicLSTMCell Kernel Tensor

To understand the format in which these tensors are being stored, let us consider an example of a **`BasicLSTMCell`**. Figure 13 illustrates what the tensor looks like within the TensorFlow checkpoint.

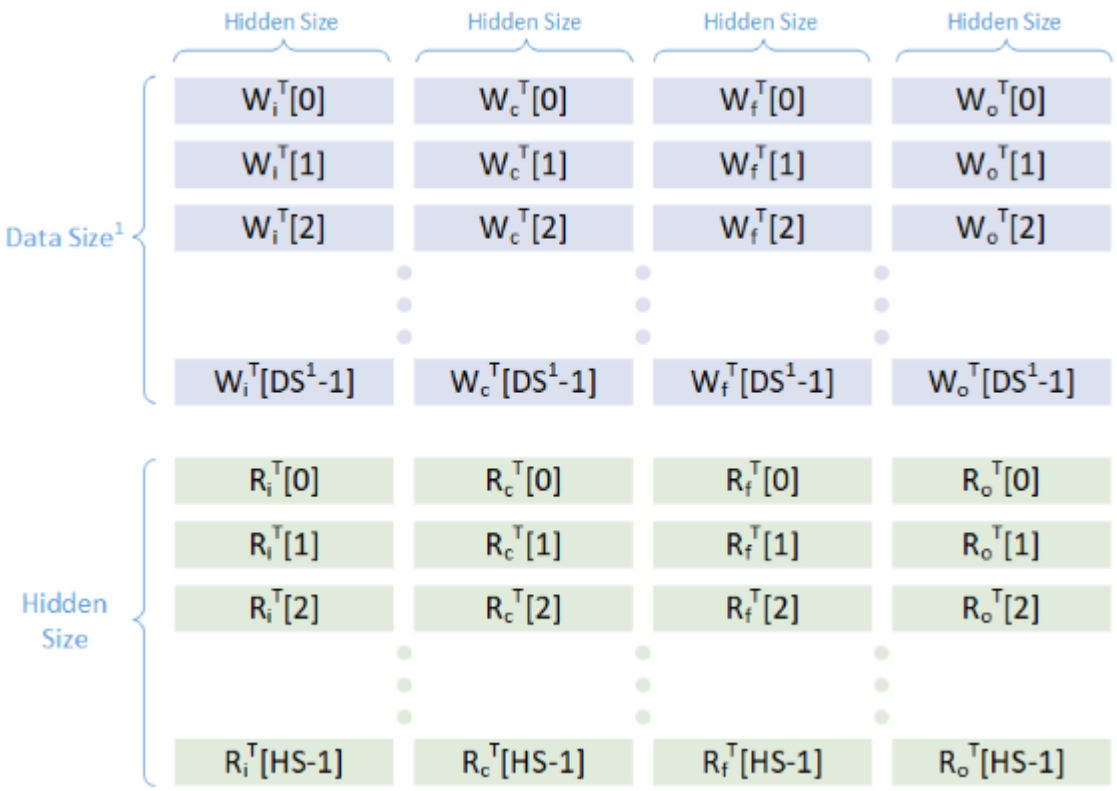


Figure 13 Tensors within a TensorFlow checkpoint

`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 13, **`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 14.

Figure 14 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`**.

8.1.4.7.2. BasicLSTMCell Bias Tensor

Figure 15 illustrates the format in which the bias tensor is stored.

Figure 15 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.

8.1.4.8. 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 https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#charRNN_sample. For more information, consult the code in this sample.

8.1.5. Preprocessing A TensorFlow Graph Using the Graph Surgeon API

The Graph Surgeon API, also known as **`graphsurgeon`**, allows you to transform TensorFlow graphs. Its capabilities are broadly divided into two categories:

Search
The search functions allow you to find nodes in a TensorFlow graph.

Manipulation
The manipulation functions allow you to modify, add, or remove nodes.

Using **`graphsurgeon`**, you can mark certain nodes (or sets of nodes) as plugin nodes in the graph. These plugins can either be plugins shipped with TensorRT or plugins written by you. For more information, see Extending TensorRT With Custom Layers.

If you are writing a plugin, also refer to see Extending TensorRT With Custom Layers for details on how to implement the **`IPluginExt`** and **`IPluignCreator`** classes in addition to registering the plugin.

The following code snippet illustrates how to use **`graphsurgeon`** to map a TensorFlow Leaky ReLU operation to a TensorRT Leaky ReLU plugin node.

```
import graphsurgeon as gs
lrelu_node = gs.create_plugin_node(name="trt_lrelu", op="LReLU_TRT",
 negSlope=0.2)
namespace_plugin_map = { "tf_lrelu" : lrelu_node }

# Transform TensorFlow graph using graphsurgeon and save to UFF
dynamic_graph = gs.DynamicGraph(tf_lrelu.graph)
dynamic_graph.collapse_namespaces(namespace_plugin_map)

# Run UFF converter using new graphdef
uff_model = uff.from_tensorflow(dynamic_graph.as_graph_def(), ["trt_lrelu"],
 output_filename="test_lrelu.uff", text=True)
```

In the above code, the **`op`** field in the **`create_plugin_node`** method should match the registered plugin name. This enables the UFF parser to look up the Plugin in the Plugin Registry using this field to insert the plugin node into the network.

For a working **`graphsurgeon`** example, see https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#uffssd_sample for C++.

For more details about the **`graphsurgeon`** API, see the Graph Surgeon API.

8.2. 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 https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html#network_api_pytorch_mnist Python sample.

Chapter 9. WORKING WITH DALI

TensorRT can be integrated with NVIDIA Data Loading Library (DALI); a collection of highly optimized building blocks and an execution engine to accelerate input data pre-processing for deep learning applications.

DALI is an open sourced library available on GitHub. It is a highly optimized open source library for data preprocessing. It uses an execution engine for fast preprocessing pipeline. DALI accelerates blocks for image loading and augmentation and also provides GPU support for JPEG decoding and image manipulation.

For more information about DALI, see the DALI data loading documentation.

9.1. Benefits Of Integration

The benefits of integrating DALI with TensorRT include the following:

- Running DNN models requires input data pre-processing
- Computational complexity of the I/O pipeline has increased. Hence, the GPU starves for data. DALI helps accelerating the preprocessing pipeline.
- DALI offsets the compute intensive data pre-processing to GPU.
- Pre-processing involves decoding, resize, crop, spatial augmentation, format conversions (NCHW and NHWC)
- Multi device DNN inference could be achieved via same I/O pipeline

DALI supports:

- The feature to accelerate pre-processing on GPUs
- Configurable graphs and custom operators
- Multiple input formats (for example JPEG, LMDB, RecordIO, TFRecord)
- Serializing a whole graph (portable graph)
- Easily integrates with framework plugins and open source bindings

DALI supports a custom operator library which can be loaded in runtime. TensorRT inference can be configured as a custom operator to be a part of DALI pipeline. A

working example of TensorRT inference integrated as a part of DALI is open sourced here.

For more information about integrating DALI with TensorRT on Xavier, see the GTC 2019 talk here.

Chapter 10. TROUBLESHOOTING

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

10.1. FAQs

Q: How do I 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: Are engines and calibration tables portable across TensorRT versions?
A: No. Internal implementations and formats are continually optimized and may change between versions. For this reason, engines and calibration tables are not guaranteed to be binary compatible with different versions of TensorRT. Applications should build new engines and INT8 calibration tables when using a new version of TensorRT.

Q: How do I 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 I 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: What can I do if my network is producing the wrong answer?

A: There are several reasons why your network may be generating incorrect answers. Here are some troubleshooting approaches which may help diagnose the problem:

- Turn on **INFO** level messages from the log stream and check what TensorRT is reporting.
- Check that your input preprocessing is generating exactly the input format required by the network.
- If you're using reduced precision, run the network in FP32. If it produces the correct result, it is possible that lower precision has insufficient dynamic range for the network.
- Try marking intermediate tensors in the network as outputs, and see if they match what you are expecting. Note: Marking tensors as outputs may inhibit optimizations, and therefore, may change the results.

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

A: TensorRT engines use device memory for two purposes: to hold the weights required by the network, and to hold the intermediate activations required by **IExecutionContext**. 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 for the engine.

IBuilder may temporarily use more device memory than what the engine requires.

- During a phase, it uses twice as much memory for the weights required by the engine. During that phase, no memory is allocated for activations.

- The auto-tuner times each layer for FP32 operation. Timing a layer in FP32 consumes twice as much memory as an FP16 operation, and four times as much for an INT8 operation, both for the weights and its input/output activations. The additional memory consumption for timing is theoretically noticeable if a single layer dominates overall memory consumption of a network.

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.

Q: If I build the engine on one GPU and run the engine on another GPU, will this work?

A: We recommend that you don't, however, if you do, you'll need to follow these guidelines:

1. The major, minor, and patch versions of TensorRT must match between systems. This ensures you are picking kernels that are still present and have not undergone certain optimizations or bug fixes that would change their behavior.
2. The CUDA compute capability major and minor versions must match between systems. This ensures that the same hardware features are present so the kernel will not fail to execute. An example would be mixing cards with different precision capabilities.
3. The following properties should match between systems:
   - Maximum GPU graphics clock speed
   - Maximum GPU memory clock speed
   - GPU memory bus width
   - Total GPU memory
   - GPU L2 cache size
   - SM processor count
   - Asynchronous engine count

   If any of the above properties do not match, you will receive the following warning: ***`Using an engine plan file across different models of devices is not recommended and is likely to affect performance or even cause errors.`***

   If you still want to proceed, then you should build the engine on the smallest SKU in the family because autotuner choices made on smaller GPUs will generalize better.

10.2. How Do I Report A Bug?

We appreciate all types of feedback. If you encounter any issues, please report them by following these steps:

1. Register for the NVIDIA Developer website.
2. Log into the developer site.
3. Click on your name in the upper right corner.
4. Click **My account** > **My Bugs** and select **Submit a New Bug**.
5. Fill out the bug reporting page. Be descriptive and if possible, provide the steps that you are following to help reproduce the problem.
6. Click **Submit a bug**.

10.3. Understanding Error Messages

If an error is encountered during execution, TensorRT will report an error message that is intended to help in debugging the problem. Some common error messages that may be encountered by developers are discussed in the following sections.

UFF Parser Error Messages

The following table captures common UFF parser error messages.

Error Message	Description
`The input to the Scale Layer is required to have a minimum of 3 dimensions.`	This error message can occur due to incorrect input dimensions. In UFF, input dimensions should always be specified in CHW format, with the implicit batch dimension *not* included in the specification.
`Invalid scale mode, nbWeights: <X>`	
`kernel weights has count <X> but <Y> was expected`	
`<NODE> Axis node has op <OP>, expected Const. The axis must be specified as a Const node.`	As indicated by the error message, the axis must be a build-time constant in order for UFF to parse the node correctly.

ONNX Parser Error Messages

The parser may issue error messages if a constant input is used with a layer that does not support constant inputs. Consider using a tensor input instead.

TensorRT Core Library Error Messages

The following table captures common TensorRT core library error messages.

	Error Message	Description
Installation Errors	`Cuda initialization failure with error <code>. Please check cuda installation: http://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html.`	This error message can occur if the CUDA or NVIDIA driver installation is corrupt. Refer to the URL for instructions on installing CUDA and the NVIDIA driver on your operating system.
Builder Errors	`Internal error: could not find any implementation for node <name>, try increasing the workspace size with IBuilder::setMaxWorkspaceSize()`	This error message occurs because there is no layer implementation for the given node in the network that can operate with the given workspace size. This usually occurs because the workspace size is insufficient, but could also indicate a bug. If increasing the workspace size as suggested doesn't help, report a bug (see How Do I Report A Bug?).
	`<layer-name>: (kernel|bias) weights has non-zero count but null values` `<layer-name>: (kernel|bias) weights has zero count but non-null values`	This error message occurs when there is a mismatch between the values and count fields in a Weights data structure passed to the builder. If the count is `0`, then the values field should contain a null pointer; otherwise the count must be non-zero and values should contain a device pointer.
	`Builder was created on device different from current device.`	This error message may show up if you: 1. Created an IBuilder targeting one GPU, then 2. Called `cudaSetDevice()` to target a different GPU, then 3. Attempted to use the IBuilder to create an engine.

	Error Message	Description
		Ensure you only use the IBuilder when targeting the GPU that was used to create the IBuilder.
	You may encounter error messages indicating that the tensor dimensions do not match the semantics of the given layer. Carefully read the documentation on NvInfer.h on the usage of each layer and the expected dimensions of the tensor inputs and outputs to the layer.	
INT8 Calibration Errors	`Tensor <X> is uniformly zero; network calibration failed.`	This warning occurs, and should be treated as an error, when data distribution for a tensor is uniformly zero. In a network, the output tensor distribution can be uniformly zero under the following scenarios: 1. Constant tensor with all zero values; not an error. 2. Activation (Relu) output with all negative inputs: not an error. 3. Data distribution is forced to all zero due to computation error in previous layer; emit a warning here.[1] 4. User does not provide any calibration images; emit a warning here.[1]
	`Could not find scales for tensor <X>.`	This error message indicates that a calibration failure occurred with no scaling factors detected. This could be due to no INT8 calibrator or insufficient custom scales for network layers. Refer to the INT8 sample to setup calibration correctly.

[1] **It is recommended to evaluate the calibration input or validate the previous layer outputs.**

	Error Message	Description
Engine Compatibility Errors	`The engine plan file is not compatible with this version of TensorRT, expecting (format|library) version <X> got <Y>, please rebuild.`	This error message can occur if you are running TensorRT using an engine PLAN file that is incompatible with the current version of TensorRT. Ensure you use the same version of TensorRT when generating the engine and running it.
	`The engine plan file is generated on an incompatible device, expecting compute <X> got compute <Y>, please rebuild.`	This error message can occur if you build an engine on a device of a different compute capability than the device that is used to run the engine. If you build an engine on a device with the same compute capability but is not identical to the device that is used to run the engine, you will see the following warning: `Using an engine plan file across different models of devices is not recommended and is likely to affect performance or even cause errors.` As indicated by the warning, it is highly recommended to use a device of the same model when generating the engine and deploying it to avoid compatibility issues.
Out Of Memory Errors	`GPU memory allocation failed during initialization of (tensor|layer): <name> GPU memory`	These error messages can occur if there is insufficient GPU memory available to instantiate a given TensorRT engine. Verify that the GPU has sufficient available memory to contain the required layer weights and activation tensors.
	`Allocation failed during deserialization of weights.`	
	`GPU does not meet the minimum memory`	

	Error Message	Description
	`requirements to run this engine …`	
FP16 Errors	`Network needs native FP16 and platform does not have native FP16`	This error message can occur if you attempted to deserialize an engine that uses FP16 arithmetic on a GPU that does not support FP16 arithmetic. You will either need to rebuild the engine without FP16 precision inference or upgrade your GPU to a model that supports FP16 precision inference.
Plugin Errors	`Custom layer <name> returned non-zero initialization`	This error message can occur if the `initialize()` method of a given plugin layer returns a non-zero value. Refer to the implementation of that layer to debug this error further. For more information, see TensorRT Layers.

10.4. Support

Support, resources, and information about TensorRT can be found online at https://developer.nvidia.com/tensorrt. This includes blogs, samples, and more.

In addition, you can access the NVIDIA DevTalk TensorRT forum at https://devtalk.nvidia.com/default/board/304/tensorrt/ for all things related to TensorRT. This forum offers the possibility of finding answers, make connections, and to get involved in discussions with customers, developers, and TensorRT engineers.

Appendix A. APPENDIX

A.1. TensorRT Layers

In TensorRT, layers represent distinct flavours of mathematical and/or programmatic operations. The following sections describe every layer that is supported by TensorRT. To view a list of the specific attributes that are supported by each layer, refer to the TensorRT API documentation.

TensorRT has the ability to optimize performance by fusing layers. For information about how to enable layer fusion optimizations, see Types Of Fusions. For information about how to optimize layer performance, see How Do I Optimize My Layer Performance? from the Best Practices guide.

A.1.1. IActivationLayer

The IActivationLayer implements element-wise activation functions.

Layer Description

Apply an activation function on a input tensor **A**, and produce an output tensor **B** with the same dimensions.

The Activation layer supports the following operations:

```
rectified Linear Unit (ReLU): B = ReLU(A)
Hyperbolic tangent: B = tanh(A)
"s" shaped curve (sigmoid): B = σ(A)
```

Conditions And Limitations

None

See the C++ IActivationLayer method or the Python IActivationLayer method for further details.

A.1.2. IConcatenationLayer

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

Layer Description

The concatenation layer is passed in an array of m input tensors $\mathbf{A}^i$ and a channel axis c.

All dimensions of all input tensors must match in every axis except axis c. Let each input tensor have dimensions $\mathbf{a}^i$. The concatenated output tensor will have dimensions $\mathbf{b}$ such that

$$\mathbf{b}_j = \{a_j \text{ if } j \neq c, \text{ and } \sum_{i=0}^{m-1} a_c^i \text{ otherwise}\}$$

Conditions And Limitations

The default channel axis is assumed to be the third from last axis, or the first non-batch axis if there are fewer than 3 non-batch axes. Concatenation cannot be done along the batch axis. All input tensors must either be non-INT32 type or all must be INT32 type.

See the C++ IConcatenationLayer method or the Python IConcatenationLayer method for further details.

A.1.3. IConstantLayer

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

Layer Description

Given dimensions $\mathbf{d}$ and weight vector $\mathbf{w}$, the constant layer will output a tensor $\mathbf{B}$ of dimensions $\mathbf{d}$ with the constant values in $\mathbf{w}$. This layer takes no input tensor. The number of elements in the weight vector $\mathbf{w}$ is equal to the volume of $\mathbf{d}$.

Conditions And Limitations

The output can be a tensor of zero to seven dimensions.

See the C++ IConstantLayer method or the Python IConstantLayer method for further details.

A.1.4. IConvolutionLayer

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

The operation the IConvolutionLayer performs is actually a correlation. Therefore, it is a consideration if you are formatting weights to import via an API, rather than via the NvCaffeParser library.

Layer Description

Compute a cross-correlation with 2D filters on a 4D tensor **A**, of dimensions **a**, to produce a 4D tensor **B**, of dimensions **b**. The dimensions of **B** depend on the dimensions of **A**, the number of output maps ***m***, kernel size **r**, symmetric padding **p**, stride **s**, dilation **d**, and dilated kernel size **t = r+d(r-1)**, such that height and width are adjusted accordingly as follows:

- $b = [a_0\ m\ b_2\ b_3]$
- $b_2 = (a_2+2p_0-t_0)/s_0+1$
- $b_3 = (a_3+2p_1-t_1)/s_1+1$

The kernel weights **w** and bias weights **x** (optional) for the number of groups ***g***, are such that:

- **w** is ordered according to shape $[m\ a_1/g\ r_0\ r_1]$
- **x** has length ***m***

Let tensor **K** with dimensions $k = [m\ a_1/g\ t_0\ t_1]$ be defined as the zero-filled tensor, such that:

- $k_{i,j,hh,ll} = w_{i,j,h,l}$
- $hh = \{0 \text{ if } h = 0,\ h + d_0(h-1) \text{ otherwise}\}$, and $ll = \{0 \text{ if } l = 0,\ l + d_1(l-1) \text{ otherwise}\}$.

and tensor **C** the zero-padded copy of **A** with dimensions $[a_0\ a_1\ a_2+p_1]$, then tensor **B** is defined as:

$$B_{i,j,k,l} = \sum (C_{i,:,k:kk,l:ll} \times K_{j,:,:,:}) + x_j$$

where $kk = k+t_0-1$, and $ll = l+t_1-1$.

Conditions And Limitations

Input and output may have more than 4 dimensions; beyond 4, all dimensions are treated as multipliers on the batch size, and input and output are treated as 4D tensors. If groups are specified and INT8 data type is used, then the size of the groups must be a multiple of 4 for both input and output.

See the C++ IConvolutionLayer method or the Python IConvolutionLayer method for further details.

A.1.5. IDeconvolutionLayer

The IDeconvolutionLayer computes a 2D (channel, height, and width) deconvolution, with or without bias.

This layer actually applies a 2D transposed convolution operator over a 2D input. It is also known as fractionally-strided convolution or transposed convolution.

Layer Description

Compute a cross-correlation with 2D filters on a 4D tensor **A**, of dimensions **a**, to produce a 4D tensor **B**, of dimensions **b**. The dimensions of **B** depend on the dimensions of **A**, the number of output maps ***m***, kernel size **r**, symmetric padding **p**, stride **s**, dilation **d**, and dilated kernel size **t = r+d(r-1)**, such that height and width are adjusted accordingly as follows:

- $\mathbf{b} = [\mathbf{a}_0\ \boldsymbol{m}\ \mathbf{b}_2\ \mathbf{b}_3]$
- $\mathbf{b}_2 = (\mathbf{a}_2-1)*\mathbf{s}_0 + \mathbf{t}_0 - 2\mathbf{p}_0$
- $\mathbf{b}_3 = (\mathbf{a}_3-1)*\mathbf{s}_1 + \mathbf{t}_1 - 2\mathbf{p}_0$

The kernel weights **w** and bias weights **x** (optional) for the number of groups ***g***, are such that:

- **w** is ordered according to shape $[\mathbf{a}_1/\boldsymbol{g}\ \boldsymbol{m}\ \mathbf{r}_0\ \mathbf{r}_1]$
- **x** has length ***m***

Let tensor **K** with dimensions $\mathbf{k} = [\boldsymbol{m}\ \mathbf{b}_1/\boldsymbol{g}\ \mathbf{t}_0\ \mathbf{t}_1]$ be defined as the zero-filled tensor, such that:

- $\mathbf{k}_{i,j,hh,ll} = \mathbf{w}_{i,j,h,l}$
- $\mathbf{hh} = \{0 \text{ if } \mathbf{h} = 0,\ \mathbf{h} + \mathbf{d}_0(\mathbf{h}-1) \text{ otherwise}\}$, and $\mathbf{ll} = \{0 \text{ if } \mathbf{l} = 0,\ \mathbf{l} + \mathbf{d}_1(\mathbf{l}-1) \text{ otherwise}\}$.

and tensor **C** the zero-padded copy of **A** with dimensions $[\mathbf{a}_0\ \mathbf{a}_1\ \mathbf{a}_2+\mathbf{p}_1]$, then tensor **B** is defined as:

$$\mathbf{B}_{i,j,k,l} = \sum_{u,v} (\mathbf{C}_{i,j,k-u,l-v}\ \ \mathbf{K}) + \mathbf{x}_j$$

where u ranges from **0** to $\mathbf{min}(\mathbf{t}_0-1,\ \mathbf{k})$, and v ranges from **0** to $\mathbf{min}(\mathbf{t}_1-1,\ \mathbf{l})$.

Conditions And Limitations

Input and output may have more than 4 dimensions; beyond 4, all dimensions are treated as multipliers on the batch size, and input and output are treated as 4D tensors. If groups are specified and INT8 data type is used, then the size of the groups must be a multiple of 4 for both input and output.

See the C++ IDeconvolutionLayer method or the Python IDeconvolutionLayer method for further details.

A.1.6. IElementWiseLayer

The IElementWiseLayer, also known as the Eltwise layer, implements per-element operations.

Layer Description

This layer computes a per-element binary operation between input tensor **A** and input tensor **B** to produce an output tensor **C**. For each dimension, their lengths must match, or one of them must be one. In the latter case, the tensor is broadcast along that axis. The output tensor has the same number of dimensions as the inputs. For each dimension, its length is the maximum of the lengths of the corresponding input dimension.

The IElementWiseLayer supports the following operations:

```
Sum: C = A+B
Product: C = A*B
Minimum: C = min(A, B)
Maximum: C = max(A, B)
Subtraction: C = A-B
Division: C = A/B
Power: C = A^B
```

Conditions And Limitations

The length of each dimension of the two input tensors **A** and **B** must be equal or equal to one.

See the C++ IElementWiseLayer method or the Python IElementWiseLayer method for further details.

A.1.6.1. ElementWise Layer Setup

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

C++ code snippet

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

Python code snippet

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

For more information, see the TensorRT API documentation.

A.1.7. IFullyConnectedLayer

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

Layer Description

The IFullyConnectedLayer expects an input tensor **A** of three or more dimensions. Given an input tensor **A** of dimensions $a=[a_0 \ldots a_{n-1}]$, it is first reshaped into a tensor **A'** of dimensions $a'=[a_0 \ldots a_{n-4} (a_{n-3}*a_{n-2}*a_{n-1})]$ by squeezing the last three dimensions into one dimension.

Then, the layer performs the operation $B'=WA'+X$ where **W** is the weight tensor of dimensions $w=[(a_{n-3}*a_{n-2}*a_{n-1}) k]$, **X** is the bias tensor of dimensions $x=[k]$ broadcasted along the other dimensions, and ***k*** is the number of output channels, configurable via `setNbOutputChannels()`. If **X** is not specified, the value of the bias is implicitly `0`. The resulting **B'** is a tensor of dimensions $b'=[a_0 \ldots a_{n-4} k]$.

Finally, **B'** is reshaped again into the output tensor **B** of dimensions $b=[a_0 \ldots a_{n-4} k 1 1]$ by inserting two lower dimensions each of size `1`.

In summary, for input tensor **A** of dimensions $a=[a_0 \ldots a_{n-1}]$, the output tensor **B** will have dimensions $b=[a_0 \ldots a_{n-4} k 1 1]$.

Conditions And Limitations

A must have three dimensions or more.

See the C++ IFullyConnectedLayer method or the Python IFullyConnectedLayer method for further details.

A.1.8. IGatherLayer

The IGatherLayer implements the **gather** operation on a given axis.

Layer Description

The IGatherLayer gathers elements of each data tensor **A** along the specified axis***x*** using indices tensor **B** of zero dimensions or more dimensions, to produce output tensor **C** of dimensions **c**.

If **B** has zero dimensions and it is a scalar ***b***, then $c_k=\{a_k \text{ if } k<x, \text{ and } a_{k+1} \text{ if } k>x\}$ and **c** has length equal to one less than the length of **a**. In this case, $C_i=A_j$ where $j_k=\{b \text{ if } k=x, i_k \text{ if } k<x, \text{ and } i_{k-1} \text{ if } k>x\}$.

If **B** is a tensor of dimensions **b** (with length ***b***), then $c_k=\{a_k \text{ if } k<x, b_{k-x} \text{ if } k\geq x \text{ and } k<x+b, \text{ and } a_{k-b+1} \text{ otherwise}\}$. In this case, $C_i=A_j$ where $j_k=\{B_{X(i)} \text{ if } k=x, i_k \text{ if } k<x, \text{ and } i_{k-b} \text{ if } k>x\}$ and $X(i)=i_{x,..,x+b-1}$.

Conditions And Limitations

Elements cannot be gathered along the batch size dimension. The data tensor **A** must contain at least one non-batch dimension. The data tensor A must contain at least *axis* + 1 non-batch dimensions. The indices tensor **B** must contain only INT32 values. The parameter axis is zero-indexed and starts at the first non-batch dimension of data tensor **A**. If there are any invalid indices elements in the indices tensor, then zeros will be stored at the appropriate locations in the output tensor.

See the C++ IGatherLayer method or the Python IGatherLayer method for further details.

A.1.9. IIdentityLayer

The IIdentityLayer implements the identity operation.

Layer Description

The output of the layer is mathematically identical to the input. This layer allows you to precisely control the precision of tensors and transform from one precision to another. If the input is at a different precision than the output, the layer will convert the input tensor into the output precision.

Conditions And Limitations

None

See the C++ IIdentityLayer method or the Python IIdentityLayer method for further details.

A.1.10. IPluginV2Layer

The IPluginV2Layer provides the ability to extend the functionalities of TensorRT by using custom implementations for unsupported layers.

Layer Description

The IPluginV2Layer is used to set-up and configure the plugin. See IPluginV2 API Description for more details on the API. TensorRT also has support for a Plugin Registry; a single registration point for all plugins in the network. In order to register plugins with

the registry, implement the **IPluginV2** class and the **IPluginCreator** class for your plugin.

Conditions And Limitations

None

See the C++ IPluginV2Layer method or the Python IPluginV2Layer method for further details.

A.1.11. ILRNLayer

The ILRNLayer implements cross-channel Local Response Normalization (LRN).

Layer Description

Given an input **A**, the LRN layer performs a cross-channel Local Response Normalization to produce output **B**of the same dimensions.. The operation of this layer depends on 4 constant values: ***w*** is the size of the cross-channel window over which the normalization will occur, ***α***, ***β***, and ***k*** are normalization parameters. The formula below describes the operation performed by the layer:

$$B_I = \frac{A_I}{(k + \alpha A_{j(I)}{}^2)^\beta}$$

Where ***I*** represents the indexes of tensor elements, and ***j(I)*** the indices where the channel dimension is replaced by ***j***. For channel index ***c*** of ***C***channels, index ***j*** ranges from ***max(0, c-w)*** and ***min(C-1, c+w)***.

Conditions And Limitations

A must have 3 or more dimensions. The following list shows the possible values for the parameters:

- ***w*** **#{1, 3, 5, 7, 9, 11, 13, 15}**
- ***α*** **#[-1 x 10^{20}, 1 x 10^{20}]**
- ***β*** **#[0.01, 1 x 10^{5}]**
- ***k*** **#[1 x 10^{-5}, 1 x 10^{10}]**

See the C++ ILRNLayer method or the Python ILRNLayer method for further details.

A.1.12. IMatrixMultiplyLayer

The IMatrixMultiplyLayer implements matrix multiplication for a collection of matrices.

Layer Description

The IMatrixMultiplyLayer computes the matrix multiplication of input tensors **A**, of dimensions **a**, and **B**, of dimensions **b**, and produces output tensor **C**, of dimensions **c**. **A**, **B**, and **C** all have the same rank $n \geq 2$. If $n > 2$, then **A**, **B**, and **C** are treated as collections of matrices; **A** and **B** may be optionally transposed (the transpose is applied to the last two dimensions). Let $\mathbf{A}^I$ and $\mathbf{B}^I$ be the input tensors after the optional transpose, then $\mathbf{C}_{i0,..,in-3,:,:} = \mathbf{A}^I_{i0,..,in-3,:,:} * \mathbf{B}^I_{i0,..,in-3,:,:}$.

Given the corresponding dimensions $\mathbf{a}^I$ and $\mathbf{b}^I$ of $\mathbf{A}^I$ and $\mathbf{B}^I$, then $\mathbf{c}_i = \{\max(a_i, b_i) \text{ if } i < n-2, a^I_i \text{ if } i = n-2, \text{ and } b^I_i \text{ if } i = n-1\}$; that is the resulting collection has the same number of matrices as the input collections, and the rows and columns correspond to the rows in $\mathbf{A}^I$ and the columns in $\mathbf{B}^I$. Notice also the use of max in the lengths, for the case of broadcast on a dimension.

Conditions And Limitations

Tensors **A** and **B** must have at least two dimensions, and agree on the number of dimensions. The length of each dimension must be the same, assuming that dimensions of length one are broadcast to match the corresponding length.

See the C++ IMatrixMultiplyLayer method or the Python IMatrixMultiplyLayer method for further details.

A.1.12.1. MatrixMultiply Layer Setup

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

C++ code snippet

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

Python code snippet

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

For more information, see the TensorRT API documentation.

A.1.13. IPaddingLayer

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

Layer Description

The IPaddingLayer pads zeros to (or trims edges from) an input tensor **A** along each of the two innermost dimensions and gives the output tensor **B**. Padding can be different on each dimension, asymmetric, and can be either positive (resulting in expansion of the tensor) or negative (resulting in trimming). Padding at the beginning and end of the two dimensions is specified by 2D vectors **x** and **y**, for pre and post padding respectively.

For input tensor **A** of ***n*** dimensions **a,** the output **B** will have ***n*** dimensions **b** such that $b_i=\{x_0+a_{n-2}+y_0$ **if** $i=n-2;$ $x_1+a_{n-1}+y_1$ **if** $i=n-1;$ **and** a_i **otherwise**$\}$. Accordingly, the values of B_w are zeros if $w_{n-2}<x_0$ **or** $x_0+a_{n-2} \leq w_{n-2}$ **or** $w_{n-1}<x_1$ **or** $x_1+a_{n-2} \leq w_{n-1}$. Otherwise, $B_w=A_z$ where $z_{n-2}=w_{n-2}+x_0$, $z_{n-1}=w_{n-1}+x_1$, and $z_i=w_i$ for all other dimensions ***i***.

Conditions And Limitations

- **A** must have three dimensions or more.
- The padding can only be applied along the two innermost dimensions.
- Only zero-padding is supported.

See the C++ IPaddingLayer method or the Python IPaddingLayer method for further details.

A.1.14. IPluginLayer

The IPluginLayer is user-defined and provides the ability to extend the functionalities of TensorRT. See Extending TensorRT With Custom Layers for more details.

See the C++ IPluginLayer method or the Python IPluginLayer method for further details.

A.1.15. IPoolingLayer

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

Layer Description

Compute a pooling with 2D filters on a tensor **A**, of dimensions **a**, to produce a tensor **B**, of dimensions **b**. The dimensions of **B** depend on the dimensions of **A**, window size **r**, symmetric padding **p** and stride **s** such that:

- $b = [a_0\ a_1 ... a_{n-3}\ b_{n-2}\ b_{n-1}]$
- $b_{n-2} = (a_{n-2}+2p_0-r_0)/s_0+1$
- $b_{n-1} = (a_{n-1}+2p_1-r_1)/s_1+1$

Let tensor **C** be the zero-padded copy of **A** with dimensions $[a_0\ a_1 ...\ a_{n-2}+2p_0\ a_{n-1}+2p_1]$ then, $B_{j.....kl} = func(C_{j....\ k:kk\ l:ll})$ where $kk = k+r_0-1$, and $ll = l+r_1-1$.

Where **func** is defined by one of the pooling types **t**:

PoolingType::kMAX
Maximum over elements in window.

PoolingType::kAVERAGE
Average over elements in the window.

PoolingType::kMAX_AVERAGE_BLEND
Hybrid of maximum and average pooling. The results of the maximum pooling and the average pooling are combined with the blending factor as **(1-blendFactor)*maximumPoolingResult + blendFactor*averagePoolingResult** to yield the result. The **blendFactor** can be set to a value between 0 and 1.

By default, average pooling is performed on the overlap between the pooling window and the padded input. If the **exclusive** parameter is set to **true**, the average pooling is performed on the overlap area between the pooling window and unpadded input.

Conditions And Limitations

Input and output tensors should have 3 or more dimensions.

See the C++ IPoolingLayer method or the Python IPoolingLayer method for further details.

A.1.16. IRaggedSoftMaxLayer

The IRaggedSoftMaxLayer applies the SoftMax function on an input tensor of sequences across the sequence lengths specified by the user.

Layer Description

This layer has two inputs: a 2D input tensor **A** of shape **zs** containing **z** sequences of data and a 1D bounds tensor **B** of shape **z** containing the lengths of each of the **z** sequences in **A**. The resulting output tensor **C** has the same dimensions as the input tensor **A**.

The SoftMax function **S** is defined on every **i** of the **z** sequences of data values $A_{i,0:Bi}$ just like in the SoftMax layer.

Conditions And Limitations

None

See the C++ IRaggedSoftMaxLayer method or the Python IRaggedSoftMaxLayer method for further details.

A.1.17. IReduceLayer

The IReduceLayer implements dimension reduction of tensors using reduce operators.

Layer Description

The IReduceLayer computes a reduction of input tensor **A**, of dimensions **a**, to produce an output tensor **B**, of dimensions **b**, over the set of reduction dimensions **r**. The reduction operator op is one of ***max***, ***min***, ***product***, ***sum***, and ***average***. The reduction can preserve the number of dimensions of **A** or not. If the dimensions are kept, then b_i**={1 if i#r, and** a_i **otherwise}**; if the dimensions are not kept, then $b_{j-m(j)}=a_j$ where **j#r** and ***m(j)*** is the number of reduction indexes in **r** less than or equal to ***j***.

With the sequence of indexes **i**, B_i**=op(**A_j**)**, where the sequence of indexes **j** is such that j_k**={: if k#r, and** i_k **otherwise}**.

Conditions And Limitations

The input must have at least one non-batch dimension. The batch size dimension cannot be reduced.

See the C++ IReduceLayer method or the Python IReduceLayer method for further details.

A.1.18. IRNNLayer

The IRNNLayer is deprecated in favor of IRNNv2Layer, however, it is still available for backwards compatibility.

Layer Description

This layer is identical to the IRNNv2Layer in functionality, but contains additional limitations as described in the Conditions and Limitations section.

Conditions And Limitations

Unlike the IRNNv2Layer, the legacy IRNNLayer does not support specifying sequence lengths via an input tensor.

The legacy IRNNLayer does not support arbitrary batch dimensions, and requires that input tensor data be specified using the dimension ordering: sequence length ***T***, batch size ***N***, embedding size ***E***. In contrast, the IRNNv2Layer requires that tensor data be

specified using the dimension ordering: batch size ***N***, sequence length ***T***, embedding size ***E***.

All limitations that apply to the IRNNv2Layer also apply to the legacy RNN layer.

See the C++ IRNNLayer method or the Python IRNNLayer method for further details.

A.1.19. IRNNv2Layer

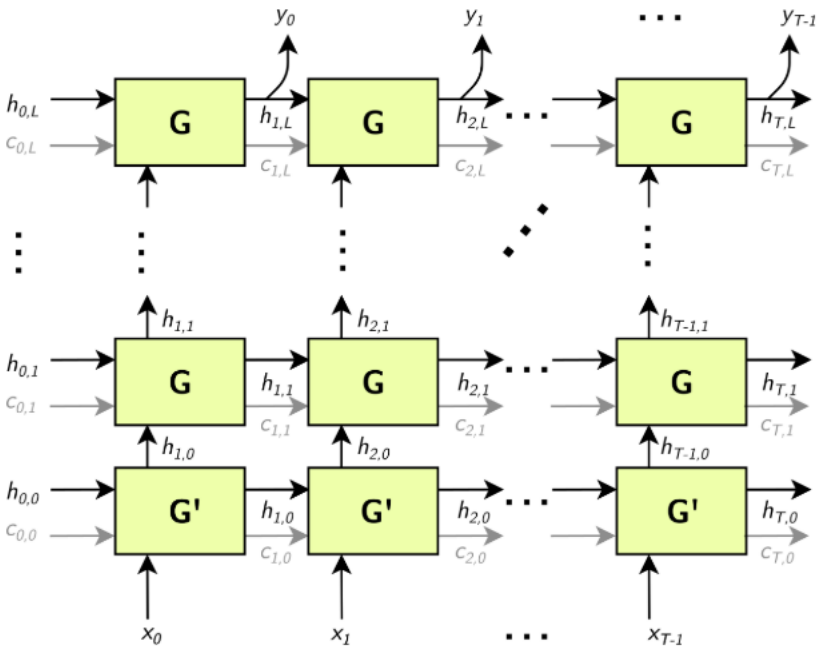
The IRNNv2Layer implements recurrent layers such as Recurrent Neural Network (RNN), Gated Recurrent Units (GRU), and Long Short-Term Memory (LSTM). Supported types are RNN, GRU, and LSTM. It performs a recurrent operation, where the operation is defined by one of several well-known recurrent neural network (RNN) "cells".

Layer Description

This layer accepts an input sequence **X**, initial hidden state $\mathbf{H_0}$, and if the cell is a long short-term memory (LSTM) cell, initial cell state $\mathbf{C_0}$, and produces an output **Y** which represents the output of the final RNN "sub-layer" computed across ***T*** timesteps (see below). Optionally, the layer can also produce an output $\mathbf{h_T}$ representing the final hidden state, and, if the cell is an LSTM cell, an output $\mathbf{c_T}$ representing the final cell state.

Let the operation of the cell be defined as the function **G(x, h, c)**. This function takes vector inputs **x**, **h**, and **c**, and produces up to two vector outputs, **h'** and **c'**, representing the hidden and cell state after the cell operation has been performed.

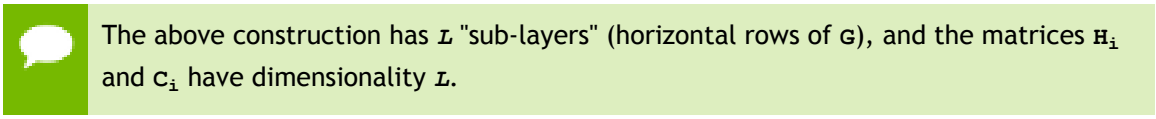
In the default (unidirectional) configuration, the RNNv2 layer applies **G** as shown in the following diagram:



G' is a variant of G,.

Arrows leading into boxes are function inputs, and arrows leading away from boxes are function outputs. $X = [x_0, x_1, \ldots, x_T]$, $Y = [y_0, y_1, \ldots, y_T]$, $H_i = [h_{i,0}, h_{i,1}, \ldots, h_{i,L}]$, and $C_i = [c_{i,0}, c_{i,1}, \ldots, c_{i,L}]$.

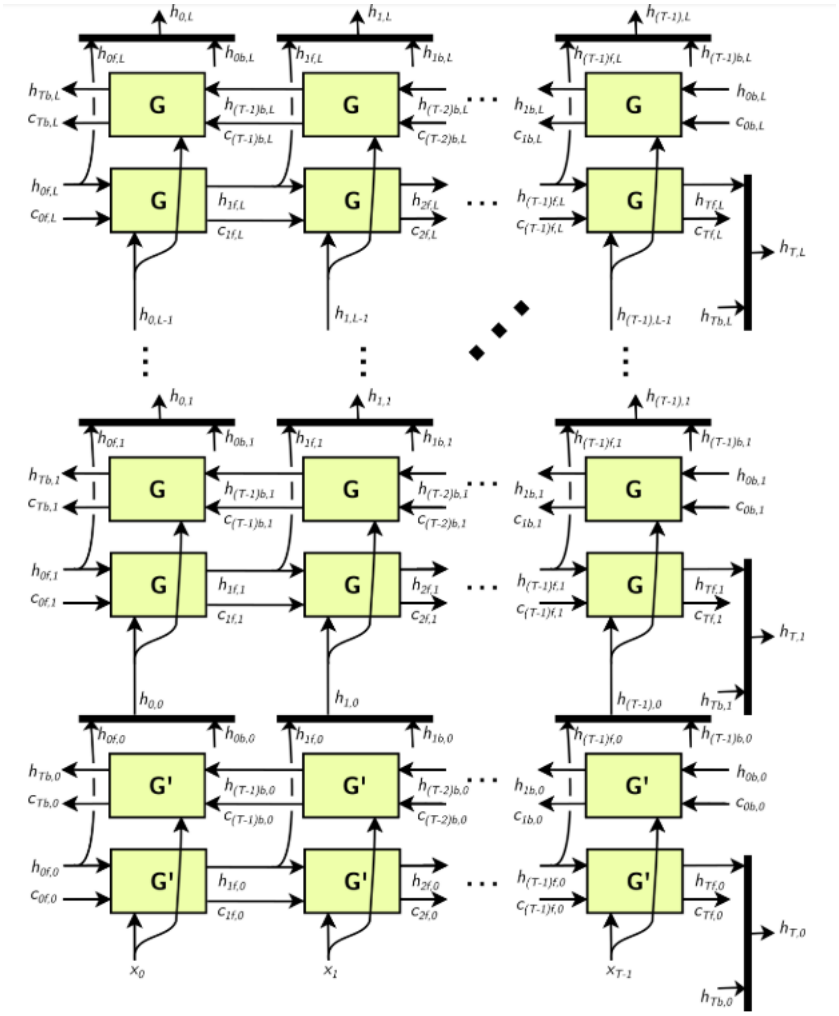
The gray c edges are only present if the RNN is using LSTM cells for G and G'.



The above construction has L "sub-layers" (horizontal rows of G), and the matrices H_i and C_i have dimensionality L.

Optionally, the sequence length T may be specified as an input to the RNNv2 layer, allowing the client to specify a batch of input sequences with different lengths.

Bidirectional RNNs (BiRNNs): The RNN can be configured to be bidirectional. In that case, each sub-layer consists of a "forward" layer and "backward" layer. The forward layer iteratively applies G using x_i from 0 to T, and the backward layer iteratively applies G using x_i from T to 0, as shown in the diagram below:



Black bars in the diagram above represent concatenation. The full hidden state h_t is defined by the concatenation of the forward hidden state h_{tf} and the backward hidden state h_{tb}:

- $h_{t,i} = [\ h_{tf,i}\ , h_{tb,i}]$
- $h_t = [\ h_{t,0},,\ \ h_{t,1},\ \ldots, h_{t,L}]$.

Similarly, for the cell state (not shown). Each $h_{t,i}$ is used as input to the next sub-layer, as shown above.

RNN operations: The RNNv2 layer supports the following cell operations:

- **ReLU:** $G(x, h, c) := \max(W_i x + R_i h + W_b + R_b, 0)$ (**c** not used)
- **tanh:** $G(x, h, c) := \tanh(W_i x + R_i h + W_b + R_b)$ (**c** not used)
- **GRU:**
  - $Z := \text{sigmoid}(W_z x + R_z h + W_{bz} + R_{bz})$
  - $M := \text{sigmoid}(W_r x + R_r h + W_{br} + R_{br})$
  - $G(x, h, c) := \tanh(W_h x + M(h + R_{bh}) + W_{bh})$ (**c** not used)
- **LSTM:**
  - $I := \text{sigmoid}(W_I x + R_I h + W_{bi} + R_{bi})$
  - $F := \text{sigmoid}(W_f x + R_f h + W_{bf} + R_{bf})$
  - $O := \text{sigmoid}(W_o x + R_o h + W_{bo} + R_{bo})$
  - $C := \tanh(W_C x + R_C h + W_{bc} + R_{bc})$
  - $C' := F \times C$
  - $H := O \times \tanh(C')$
  - $G(x, h, c) := \{ H, C' \}$

For GRU and LSTM, we refer to the intermediate computations for **Z**, **M**, **I**, **F**, etc. as "gates".

In the unidirectional case, the dimensionality of the **W** matrices is ***HxE*** for the first layer and ***HxH*** for subsequent layers (unless skip mode is set, see below). In the bidirectional case, the dimensionality of the **W** matrices is ***HxE*** for the first forward/backward layer, and ***Hx2H*** for subsequent layers.

The dimensionality of the **R** matrices is always ***HxH***. The biases W_{bx} and R_{bx} have dimensionality ***H***.

Skip mode: The default mode used by RNNv2 is "linear mode". In this mode, the first sub-layer of the RNNv2 layer uses the cell **G'(x, h, c)**, which accepts a vector **x** of size ***E*** (embedding size), and vectors **h** and **c** of size ***H*** (hidden state size), and is defined by the cell operation formula. Subsequent layers use the cell **G(x, h, c)**, where **x**, **h**, and **c** are all vectors of size ***H***, and is also defined by the cell operation formula.

Optionally, the RNN can be configured to run in "skip mode", which means the input weight matrices for the first layer are implicitly identity matrices, and **x** Is expected to be size ***H***.

Conditions And Limitations

The data (**X**) input and initial hidden/cell state (H_0 and C_0) tensors have at least 2 non-batch dimensions. Additional dimensions are considered batch dimensions.

The optional sequence length input ***T*** is **0**-dimensional (scalar) when excluding batch dimensions.

The data (**Y**) output and final hidden/cell state (H_T and C_T) tensors have at least 2 non-batch dimensions. Additional dimensions are considered batch dimensions. If the sequence length input is provided, each output in the batch is padded to the maximum sequence length T_{max}.

The IRNNv2Layer supports:

- FP32 and FP16 data type for input and output, hidden, and cell tensors.
- INT32 data type only for the sequence length tensor.

After the network is defined, you can mark the required outputs. RNNv2 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));
};
```

See the C++ IRNNv2Layer method or the Python IRNNv2Layer method for further details.

A.1.19.1. 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 example, in the network used in sampleCharRNN, we used 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.

A.1.19.2. RNNv2 Layer - Optional Inputs

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

C++ code snippet

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

Python code snippet

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

A.1.20. IScaleLayer

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

Layer Description

Given an input tensor **A**, the IScaleLayer performs a per-tensor, per-channel or per-element transformation to produce an output tensor **B** of the same dimensions. The transformations corresponding to each mode are:

ScaleMode::kUNIFORM tensor-wise transformation

$$B = (A * scale + shift)^{power}$$

ScaleMode::kCHANNEL channel-wise transformation

$$B_I = (A_I * scale_{c(I)} + shift_{c(I)})^{power_{c(I)}}$$

ScaleMode::kELEMENTWISE element-wise transformation

$$B_I = (A_I * scale_1 + shift_1)^{power_1}$$

Where I represents the indexes of tensor elements and $c(I)$ is the channel dimension in I.

Conditions And Limitations

A must have 3 or more dimensions.

If an empty weight object is provided for **scale**, **shift**, or **power**, then a default value is used. By default, **scale** has a value of **1.0**, **shift** has a value of **0.0**, and power has a value of **1.0**.

See the C++ IScaleLayer method or the Python IScaleLayer method for further details.

A.1.21. IShuffleLayer

The IShuffleLayer implements a reshape and transpose operator for tensors.

Layer Description

The IShuffleLayer implements reshuffling of tensors to permute the tensor and/or reshape it. An input tensor **A** of dimensions **a** is transformed by applying a transpose, followed by a reshape operation with reshape dimensions **r**, and then followed by another transpose operation to produce an output data tensor **B** of dimensions **b**.

To apply the transpose operation to **A**, the permutation order needs to be specified. The specified permutation *p1* is used to permute the elements of **A** in the following manner to produce output **C** of dimensions **c, such that** $c_i=a_{p1(i)}$ and $C_I=A_{p1(I)}$ for a sequence of indexes **I**. By default, the permutation is assumed to be an identity (no change to the input tensor).

The reshape operation does not alter the order of the elements, and reshapes tensor **C** into tensor **R** of shape r^I, such that $r^I_i=\{r_i \text{ if } r_i>0,\ c_i \text{ if } r_i=0,\ \text{inferred if } r_i=-1\}$. Only one dimension can be inferred, such that $\prod r^I_i=\prod a_i$.

The second transpose operation is applied after the reshape operation. It follows the same rules as the first transpose operation and requires a permutation (say ***p2***) to be specified. This permutation produces an output tensor **B** of dimensions **b**, such that $b_i=r_{p2(i)}$ and $B_{p2(I)}=R_I$ for a sequence of indexes **I**.

Conditions And Limitations

Product of dimensions r^I must be equal to the product of input dimensions **a**.

See the C++ IShuffleLayer method or the Python IShuffleLayer method for further details.

A.1.22. ISliceLayer

The ISliceLayer implements a slice operator for tensors.

Layer Description

Giving an input **n**-dimension (excluding batch dimension) tensor **A**, the Slice layer generates an output tensor **B** with elements extracted from **A**. The correspondence between element coordinates in **A** and **B** is given by: $a_i = b_i*s_i+o_i \ (0 \le i < n)$,

where **a**, **b**, **s**, **o** are element coordinates in **A**, element coordinates in **B**, stride and starting offset, respectively. The stride can be positive, negative or zero.

Conditions And Limitations

The corresponding **A** coordinates for every elements in **B** must not be out-of-bound.

See the C++ ISliceLayer method or the Python ISliceLayer method for further details.

A.1.23. ISoftMaxLayer

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

Layer Description

Given an input tensor **A** of shape **a** and an input dimension **i**, this layer applies the SoftMax function on every slice $A_{a0, \ldots, ai-1, :, ai+1, \ldots, an-1}$ along dimension **i** of **A**. The resulting output tensor **C** has the same dimensions as the input tensor **A**.

The SoftMax function ***S*** for a slice **x** is defined as:

$$S(x) = \exp(x_j) / \sum \exp(x_j)$$

The SoftMax function rescales the input such that every value in the output lies in the range **[0, 1]** and the values of every slice $C_{a0, \ldots, ai-1, :, ai+1, \ldots, an-1}$ along dimension **i** of **C** sum up to **1**.

Conditions And Limitations

For **n** being the length of **a**, the input dimension **i** should be ***i*#[0,*n*-1]**. If the user does not provide an input dimension, then ***i*=*max*(0,*n*-3)**.

See the C++ ISoftMaxLayer method or the Python ISoftMaxLayer method for further details.

A.1.24. ITopKLayer

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

Layer Description

For an input tensor **A** of dimensions **a**, given an axis ***i***, an operator that is either ***max*** or ***min***, and a value for ***k***, produces a tensor of values **V** and a tensor of indices **I** of dimensions **v** such that v_j**={*k* if i≠j, and** a_i **otherwise}**.

The output values are:

- $V_{a0, \ldots, ai-1, :, ai+1, \ldots, an} = sort(A_{a0, \ldots, ai-1, :, ai+1, \ldots, an}):K$
- $I_{a0, \ldots, ai-1, :, ai+1, \ldots, an} = argsort(A_{a0, \ldots, ai-1, :, ai+1, \ldots, an}):K$

where **sort** is in descending order for operator ***max*** and ascending order for operator ***min***.

Ties are broken during sorting with lower index considered to be larger for operator ***max***, and lower index considered to be smaller for operator ***min***.

Conditions And Limitations

The ***K*** value must be 3840 or less. Only one axis can be searched to find the top ***K*** minimum or maximum values; this axis cannot be the batch dimension.

See the C++ ITopKLayer method or the Python ITopKLayer method for further details.

A.1.24.1. TopK Layer Setup

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

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

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

C++ code snippet

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

Python code snippet

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

For more information, see the TensorRT API documentation.

A.1.25. IUnaryLayer

The IUnaryLayer supports pointwise unary operations.

Layer Description

The IUnaryLayer performs pointwise operations on input tensor **A**resulting in output tensor **B**of the same dimensions. The following functions are supported:

- **exp: B = *e*^A**

- `abs: B = |A|`
- `log: B = ln(A)`
- `sqrt: B = √A (rounded to nearest even mode)`
- `neg: B = -A`
- `recip: B = 1 / A (reciprocal) in rounded to nearest even mode`

Conditions And Limitations

Input and output can be zero to 7 dimensional tensors.

See the C++ IUnaryLayer method or the Python IUnaryLayer method for further details.

A.2. Data Format Descriptions

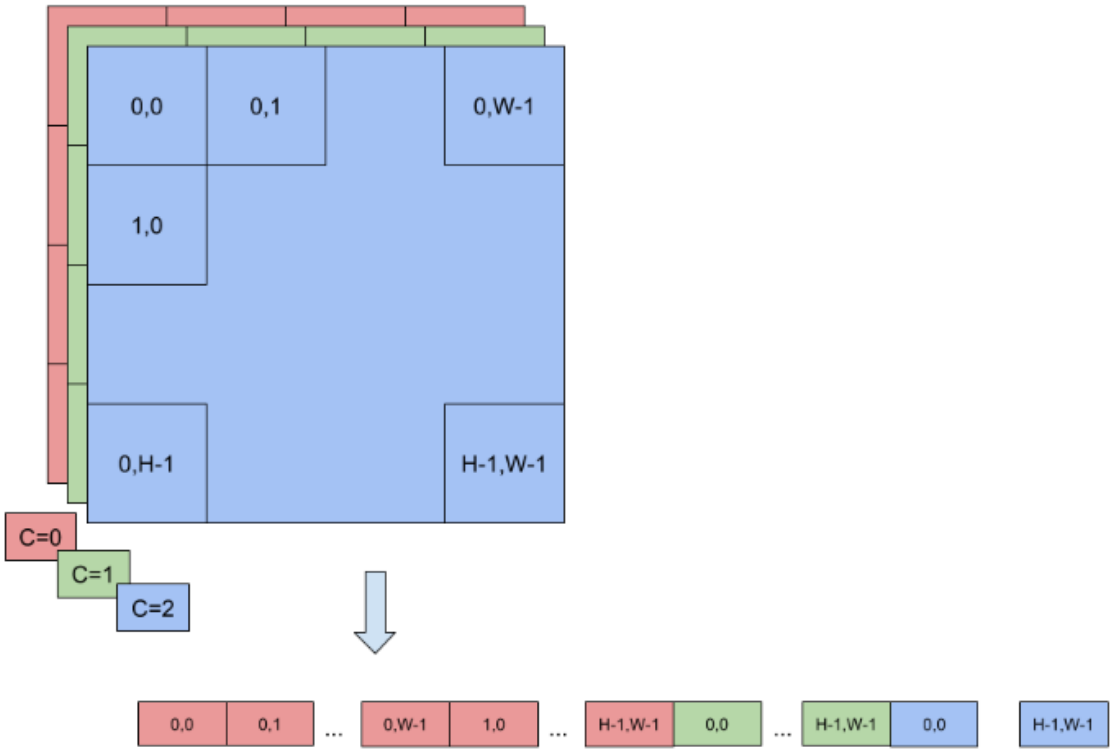
TensorRT supports different data formats. There are two aspects to consider: data type and layout.

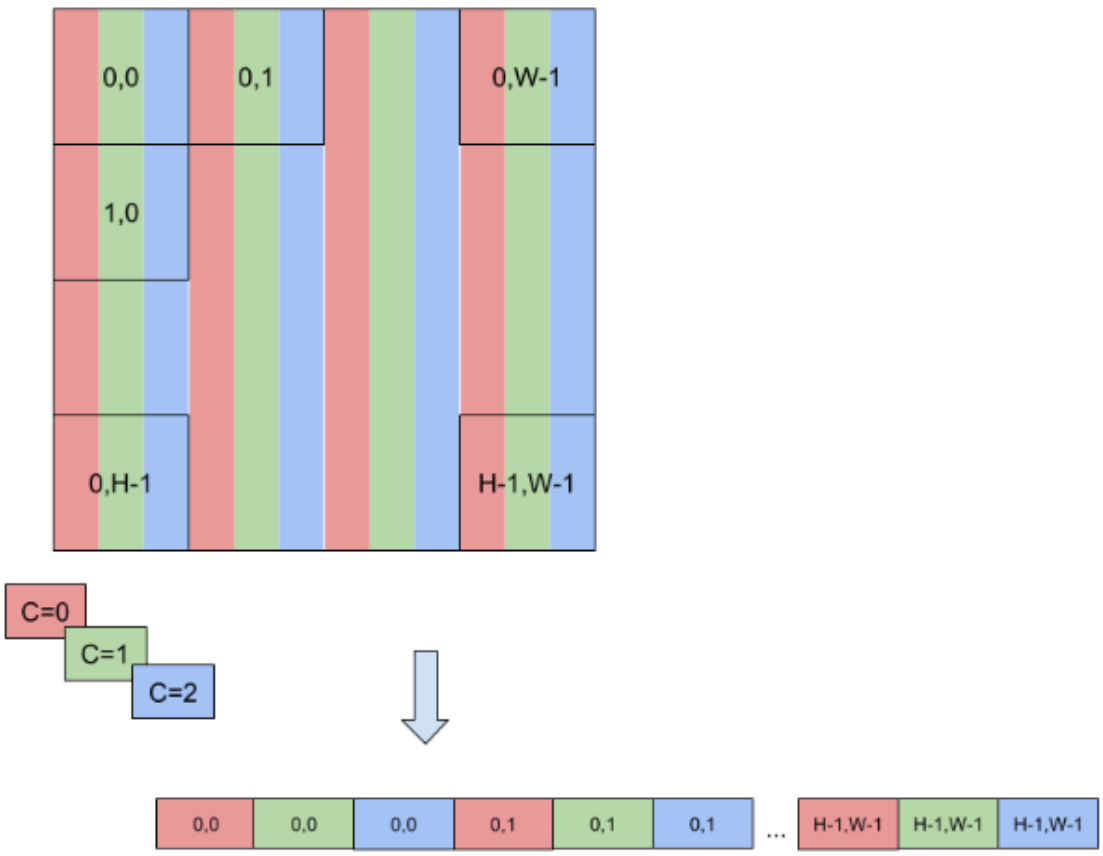
Data type format

The data type is the representation of each individual value. Its size determines the range of values and the precision of the representation; which are FP32 (32-bit floating point, or single precision), FP16 (16-bit floating point, or half precision), INT32 (32-bit integer representation) and INT8 (8-bit representation).

Layout format

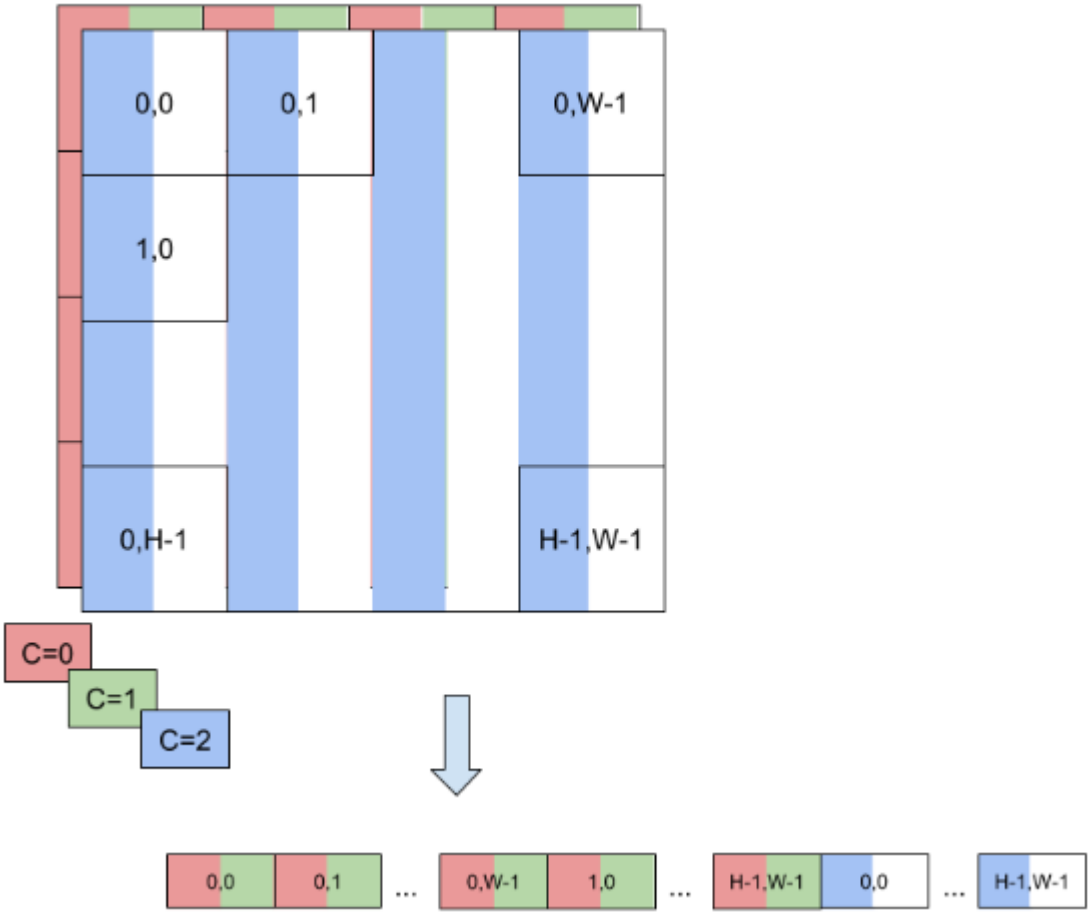
The layout format determines the ordering in which values are stored. Typically, batch dimensions are the leftmost dimensions, and the other dimensions refer to aspects of each data item such as **`C`** is channel, **`H`** is height, and **`W`** is width, in images. Ignoring batch sizes, which are always preceding these, **`C`**, **`H`**, and **`W`** are typically sorted as **`CHW`** #unique_133/unique_133_Connect_42_fig1 or **`HWC`** #unique_133/unique_133_Connect_42_fig2.


0,0
0,1
0,W-1
1,0
0,H-1
H-1,W-1
C=0
C=1
C=2
0,0
0,1
0,W-1
1,0
H-1,W-1
0,0
H-1,W-1
0,0
H-1,W-1

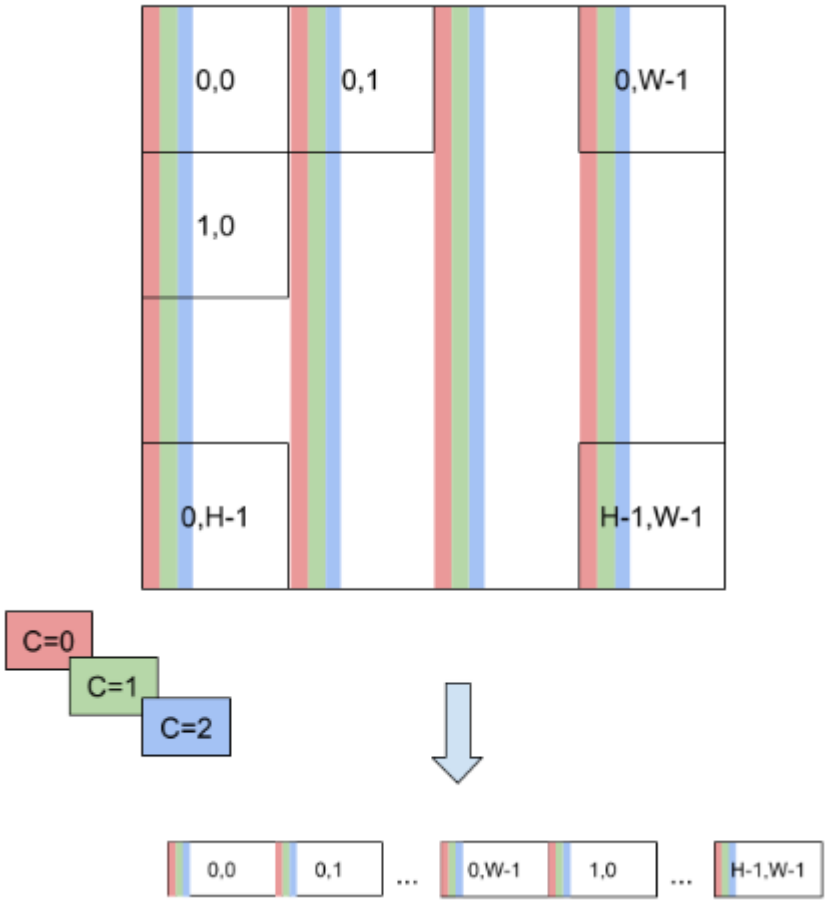

0,0
0,1
0,W-1
1,0
0,H-1
H-1,W-1
C=0
C=1
C=2
0,0
0,0
0,0
0,1
0,1
0,1
H-1,W-1
H-1,W-1
H-1,W-1

To enable faster computations, more formats are defined to pack together channel values and use reduced precision. For this reason, TensorRT also supports formats **NC/2HW2** and **NHWC8**.

In **NC/2HW2**, pairs of channel values are packed together in each **HxW** matrix (with an empty value in the case of an odd number of channels). The result is a format in which the values of **#C/2#HxW** matrices are pairs of values of two consecutive channels #unique_133/unique_133_Connect_42_fig3; notice that this ordering interleaves dimensions as values of channels that have stride **1** if they are in the same pair and stride **2xHxW** otherwise.



In **NHWC8**, the entries of an **HxW** matrix include the values of all the channels #unique_133/unique_133_Connect_42_fig4. In addition, these values are packed together in **#C/8#** 8-tuples and **C** is rounded up to the nearest multiple of 8.



A.3. Command Line Program

Included in the **samples** directory is a command line program, 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:

```
&&&& RUNNING TensorRT.trtexec # ./trtexec -h
[I] help

Mandatory params:
  --deploy=<file>          Caffe deploy file
  OR --uff=<file>          UFF file
  OR --onnx=<file>         ONNX Model file
  OR --loadEngine=<file>   Load a saved engine

Mandatory params for UFF:
  --uffInput=<name>,C,H,W Input blob name and its dimensions for
 UFF parser (can be specified multiple times)
  --output=<name>      Output blob name (can be specified
 multiple times)
```

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

Optional params:
  --model=<file>          Caffe model file (default = no model,
 random weights used)
  --batch=N               Set batch size (default = 1)
  --device=N              Set cuda device to N (default = 0)
  --iterations=N          Run N iterations (default = 10)
  --avgRuns=N             Set avgRuns to N - perf is measured as
 an average of avgRuns (default=10)
  --percentile=P          For each iteration, report the
 percentile time at P percentage (0<=P<=100, with 0 representing
 min, and 100 representing max; default = 99.0%)
  --workspace=N           Set workspace size in megabytes
 (default = 16)
  --safe                  Only test the functionality available
 in safety restricted flows.
  --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)
  --saveEngine=<file>     Save a serialized engine to file.
  --loadEngine=<file>     Load a serialized engine from file.
  --calib=<file>          Read INT8 calibration cache file.
 Currently no support for ONNX model.
  --useDLACore=N          Specify a DLA engine for layers that
 support DLA. Value can range from 0 to n-1, where n is the
 number of DLA engines on the platform.
  --allowGPUFallback      If --useDLACore flag is present and if
 a layer can't run on DLA, then run on GPU.
  --useSpinWait           Actively wait for work completion. This
 option may decrease multi-process synchronization time at the
 cost of additional CPU usage. (default = false)
  --dumpOutput            Dump outputs at end of test.
  -h, --help              Print usage
```

For example:

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

If no model is supplied, random weights are generated.

A.4. 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.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Google Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

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

Apache License Version 2.0, January 2004 http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the 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.

LICENSE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CONTRIBUTION AGREEMENT

By contributing to the BVLC/Caffe repository through pull-request, comment, or otherwise, the contributor releases their content to the license and copyright terms herein.

half.h

Copyright (c) 2012-2017 Christian Rau <rauy@users.sourceforge.net>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

jQuery.js

jQuery.js is generated automatically under doxygen.

In all cases TensorRT uses the functions under the MIT license.

CRC

policies, either expressed or implied, of the Regents of the University of California.

The copyright of UC Berkeley's Berkeley Software Distribution ("BSD") source has been updated. The copyright addendum may be found at ftp://ftp.cs.berkeley.edu/pub/4bsd/README.Impt.License.Change and is

William Hoskins

Director, Office of Technology Licensing

University of California, Berkeley

getopt.c

Copyright (c) 2002 Todd C. Miller <Todd.Miller@courtesan.com>

Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F39502-99-1-0512.

Copyright (c) 2000 The NetBSD Foundation, Inc.

All rights reserved.

This code is derived from software contributed to The NetBSD Foundation by Dieter Baron and Thomas Klausner.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE NETBSD FOUNDATION, INC. AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Notice

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

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

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

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

Trademarks

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

Copyright