TensorRT

Developer Guide

SWE-SWDOCTRT-001-DEVG_v7.2.1 | September 2020

Table of Contents

Chapter 1. What Is TensorRT?.....1
1.1. Benefits Of TensorRT.....3
1.1.1. Who Can Benefit From TensorRT.....4
1.2. Where Does TensorRT Fit?.....5
1.3. How Does TensorRT Work?.....7
1.4. What Capabilities Does TensorRT Provide?.....8
1.5. How Do I Get TensorRT?.....9
1.6. TensorRT Deprecation Policy.....10
Chapter 2. Using The C++ API.....11
2.1. Instantiating TensorRT Objects in C++.....11
2.2. Creating A Network Definition In C++.....12
2.2.1. Creating A Network Definition From Scratch Using The C++ API.....13
2.2.2. Importing A Model Using A Parser In C++.....14
2.2.3. Importing A Caffe Model Using The C++ Parser API.....15
2.2.4. Importing A TensorFlow Model Using The C++ UFF Parser API.....15
2.2.5. Importing An ONNX Model Using The C++ Parser API.....16
2.3. Building An Engine In C++.....17
2.3.1. Builder Layer Timing Cache.....18
2.4. Serializing A Model In C++.....18
2.5. Performing Inference In C++.....19
2.6. Memory Management In C++.....20
2.7. Refitting An Engine.....20
2.8. Algorithm Selection.....21
2.8.1. Determinism And Reproducibility In The Builder.....22
Chapter 3. Using The Python API.....24
3.1. Importing TensorRT Into Python.....24
3.2. Creating A Network Definition In Python.....25
3.2.1. Creating A Network Definition From Scratch Using The Python API.....25
3.2.2. Importing A Model Using A Parser In Python.....26
3.2.3. Importing From Caffe Using Python.....26
3.2.4. Importing From TensorFlow Using Python.....27
3.2.5. Importing From ONNX Using Python.....28
3.2.6. Importing From PyTorch And Other Frameworks.....29
3.3. Building An Engine In Python.....29
3.4. Serializing A Model In Python.....30

3.5. Performing Inference In Python....30
Chapter 4. Extending TensorRT With Custom Layers....32
4.1. Adding Custom Layers Using The C++ API....32
4.1.1. Example: Adding A Custom Layer Using C++....34
4.1.2. Example: Adding A Custom Layer That Is Not Supported In Caffe Using C++....35
4.1.3. Example: Adding A Custom Layer That Is Not Supported In UFF Using C++....35
4.1.4. Example: Adding A Custom Layer With Dynamic Shape Support Using C++....36
4.1.5. Example: Adding A Custom Layer With INT8 I/O Support Using C++....38
4.1.6. Example: Implementing A GELU Operator Using The C++ API....40
4.2. Adding Custom Layers Using The Python API....41
4.2.1. Example: Adding A Custom Layer to a TensorRT Network Using Python....41
4.2.2. Example: Adding A Custom Layer That Is Not Supported In UFF Using Python....42
4.3. Using Custom Layers When Importing A Model From A Framework....43
4.3.1. Example: Adding A Custom Layer To A TensorFlow Model....44
4.4. Plugin API Description....45
4.4.1. Migrating Plugins From TensorRT 6.x.x To TensorRT 7.x.x....45
4.4.1.1. Migrating Plugins From TensorRT 5.x.x To TensorRT 6.x.x....46
4.4.2. IPluginV2 API Description....46
4.4.3. IPluginCreator API Description....48
4.4.4. Persistent LSTM Plugin....49
4.5. Best Practices For Custom Layers Plugin....50
Chapter 5. Working With Mixed Precision....51
5.1. Mixed Precision Using The C++ API....51
5.1.1. Setting The Layer Precision Using C++....51
5.1.2. Enabling TF32 Inference Using C++....52
5.1.3. Enabling FP16 Inference Using C++....53
5.1.4. Enabling INT8 Inference Using C++....53
5.1.4.1. Setting Per-Tensor Dynamic Range Using C++....54
5.1.4.2. INT8 Calibration Using C++....54
5.1.5. Working With Explicit Precision Using C++....56
5.2. Mixed Precision Using The Python API....56
5.2.1. Setting The Layer Precision Using Python....56
5.2.2. Enabling FP16 Inference Using Python....57
5.2.3. Enabling INT8 Inference Using Python....57
5.2.3.1. Setting Per-Tensor Dynamic Range Using Python....57
5.2.3.2. INT8 Calibration Using Python....57
5.2.4. Working With Explicit Precision Using Python....58
Chapter 6. Working With Reformat-Free Network I/O Tensors....59

6.1. Building An Engine With Reformat-Free Network I/O Tensors....59
6.2. Supported Combination Of Data Type And Memory Layout of I/O Tensors....60
6.3. Calibration For A Network With INT8 I/O Tensors....61
6.4. Restrictions With DLA....61
6.5. FAQs....62
Chapter 7. Working With Dynamic Shapes....63
7.1. Specifying Runtime Dimensions....64
7.2. Optimization Profiles....64
7.2.1. Bindings For Multiple Optimization Profiles....65
7.3. Layer Extensions For Dynamic Shapes....66
7.4. Restrictions For Dynamic Shapes....67
7.5. Execution Tensors vs. Shape Tensors....67
7.5.1. Formal Inference Rules....68
7.6. Shape Tensor I/O (Advanced)....69
7.7. INT8 Calibration With Dynamic Shapes....70
Chapter 8. Working With Empty Tensors....71
8.1. IReduceLayer And Empty Tensors....71
8.2. IMatrixMultiplyLayer, IFullyConnectedLayer, And Empty Tensors....72
8.3. Plugins And Empty Tensors....72
8.4. IRNNLayer And Empty Tensors....72
8.5. IShuffleLayer And Empty Tensors....73
8.6. ISliceLayer And Empty Tensors....73
8.7. IConvolutionLayer And Empty Tensors....74
Chapter 9. Working With Loops....75
9.1. Defining A Loop....75
9.2. Formal Semantics....78
9.3. Nested Loops....79
9.4. Limitations....79
9.5. Replacing IRNNLayer And IRNNv2Layer With Loops....80
Chapter 10. Working With Quantized Networks....81
10.1. Quantization Aware Training (QAT) Using TensorFlow....81
10.2. Converting Tensorflow To ONNX Quantized Models....82
10.3. Importing Quantized ONNX Models....82
Chapter 11. Working With DLA....83
11.1. Running On DLA During TensorRT Inference....83
11.1.1. Example: sampleMNIST With DLA....84
11.1.2. Example: Enable DLA Mode For A Layer During Network Creation....85

11.2. DLA Supported Layers....86
11.3. GPU Fallback Mode.... 88
Chapter 12. Working With Multi-Instance GPU (MIG)....90
12.1. GPU Partitioning....90
12.2. Impact On TensorRT Applications....90
12.3. Configuring NVIDIA MIG.... 91
Chapter 13. Deploying A TensorRT Optimized Model....92
13.1. Deploying In The Cloud....92
13.2. Deploying To An Embedded System.... 92
Chapter 14. Working With Deep Learning Frameworks.... 94
14.1. Working With TensorFlow....94
14.1.1. Freezing A TensorFlow Graph.... 94
14.1.2. Freezing A Keras Model....95
14.1.3. Converting A Frozen Graph To UFF.... 95
14.1.4. Working With TensorFlow RNN Weights.... 95
14.1.4.1. TensorFlow RNN Cells Supported In TensorRT.... 95
14.1.4.2. Maintaining Model Consistency Between TensorFlow And TensorRT.... 96
14.1.4.3. Workflow....96
14.1.4.4. Dumping The TensorFlow Weights....96
14.1.4.5. Loading Dumped Weights.... 97
14.1.4.6. Converting The Weights To A TensorRT Format....97
14.1.4.7. BasicLSTMCell Example....98
14.1.4.8. Setting The Converted Weights And Biases.... 100
14.1.5. Preprocessing A TensorFlow Graph Using the Graph Surgeon API.... 101
14.2. Working With PyTorch And Other Frameworks.... 102
Chapter 15. Working With DALI.... 103
15.1. Benefits Of Integration....103
Chapter 16. Troubleshooting....104
16.1. FAQs.... 104
16.2. How Do I Report A Bug?.... 109
16.3. Understanding Error Messages.... 109
16.4. Support....113
Appendix A. Appendix.... 114
A.1. TensorRT Layers....114
A.2. Data Format Descriptions.... 140
A.3. Command-Line Programs....143
A.4. ACKNOWLEDGEMENTS....144

List of Figures

Figure 1. TensorRT is a high-performance neural network inference optimizer and runtime engine for production deployment 2

Figure 2. TensorRT is a programmable inference accelerator. 3

Figure 3. ONNX Workflow V1 7

Figure 4. Optimization profile 66

Figure 5. A TensorRT loop is set by loop boundary layers. Dataflow can leave the loop only via ILoopOutputLayer. The only backedges allowed are the second input to IRecurrenceLayer 76

Figure 6. TensorFlow RNN Workflow 96

Figure 7. Tensors within a TensorFlow checkpoint 99

Figure 8. Converted tensors 100

Figure 9. Bias tensor stored format 100

List of Tables

Table 1. Base classes, ordered from least expressive to most expressive 32

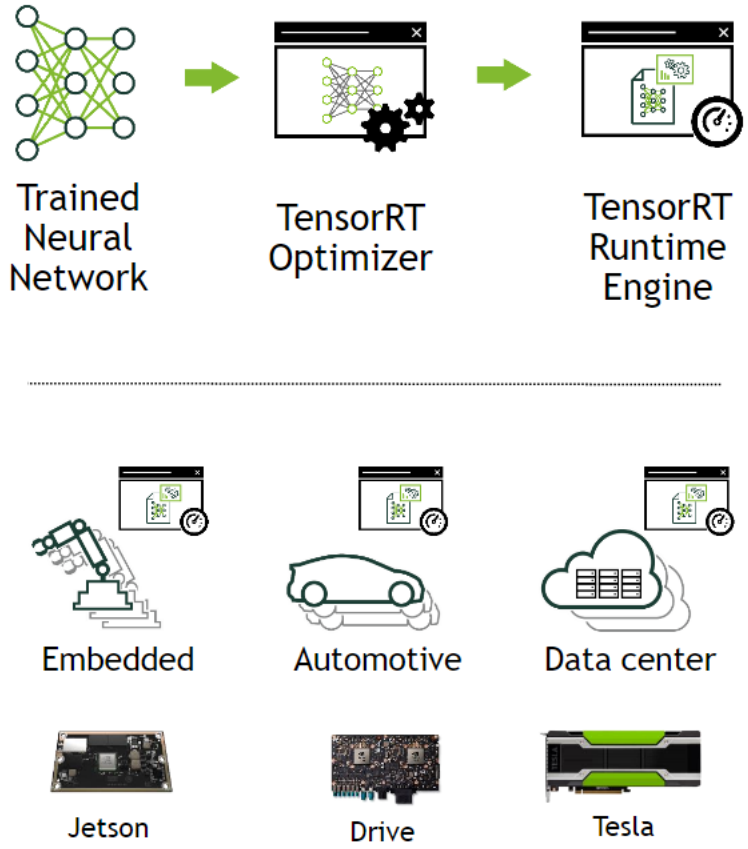
Table 2. Supported combination of data types and memory layout. 60

Chapter 1. What Is TensorRT?

The core of NVIDIA® 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.

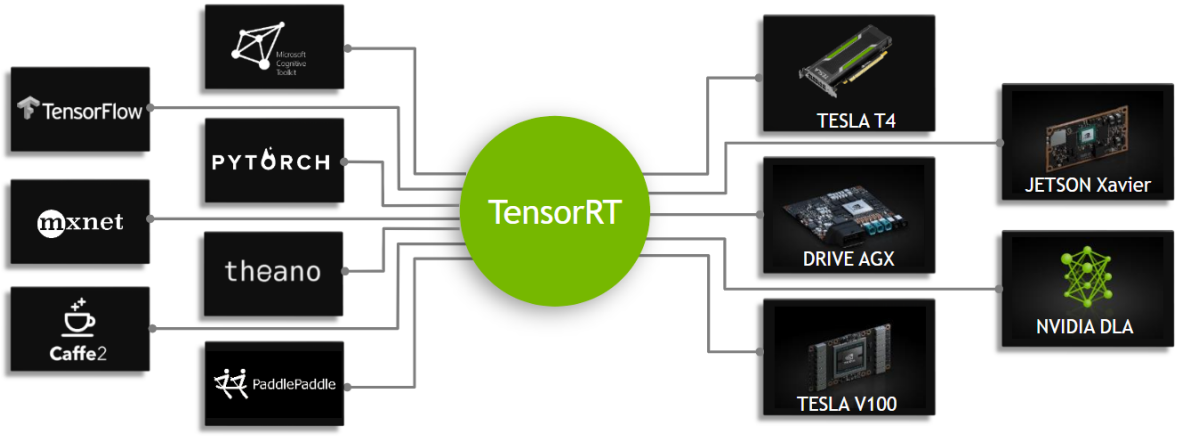
Figure 1. TensorRT is a high-performance neural network inference optimizer and runtime engine for production deployment.



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 depend 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 a high-level API that abstracts away specific hardware details and an implementation that optimizes inference for high throughput, low latency, and a low device-memory footprint.

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 data center or cloud, in an embedded device, robot or vehicle, or application software which will run on your 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 are usually done using DGX-1™ , Titan, or Tesla data center 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 is 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.

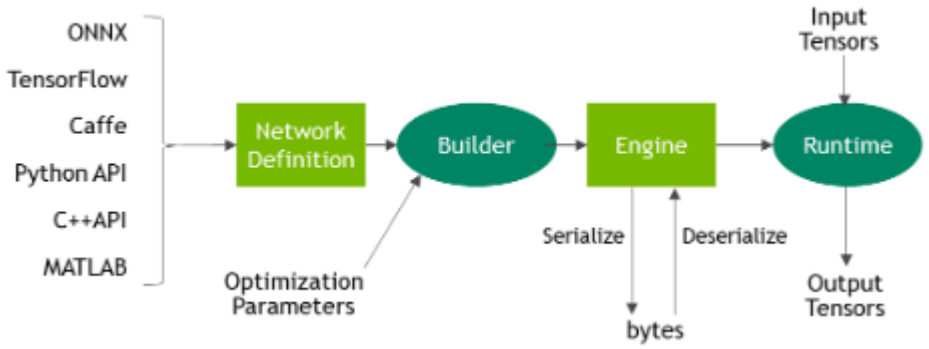
Determine what your priorities are. Given the diversity of different systems that you could implement, there are a lot of things that may need to be considered for designing and implementing the deployment architecture.

- 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), nor 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 left side of Figure 3), Caffe parser, or UFF parser.

Figure 3. ONNX Workflow V1



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

> **Note:** 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 the 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
- Elimination of operations which are equivalent to no-op
- The 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.

Optimization Profile

An optimization profile specifies constraints on dynamic dimensions. For more information, refer to the Optimization Profile API and Working With Dynamic Shapes sections.

Builder Configuration

The Builder Configuration interface specifies details for creating an engine. It allows the application to specify optimization profiles, maximum 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, refer to the Builder Config API.

Builder

The Builder interface allows the creation of an optimized engine from a network definition and a builder configuration. For more information, refer to the 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.

> **Note:** Additionally, some TensorRT Caffe and ONNX parsers and plugins can be found on GitHub.

1.5. How Do I Get TensorRT?

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

1.6. TensorRT Deprecation Policy

Deprecation is advance warning that a feature will disappear so that you have time to update your code with an alternative. Starting with TensorRT 8.0, features deprecated in release *n.x* will be removed in major release (*n*+2).0. For example, TensorRT 8.0 will omit features deprecated in TensorRT 6.0, some of which were already deprecated as far back as TensorRT 4.0. TensorRT 9.0 will omit features deprecated in TensorRT 7.x.

Chapter 2. Using The C++ API

The following sections highlight the NVIDIA® 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, in automotive.

The main benefit of the Python API is that data preprocessing and postprocessing are 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, use the interface `IExecutionContext`. In order to create an object of type `IExecutionContext`, first create an object of type `ICudaEngine` (the engine).

Create an engine 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 a serialized engine from the disk. This can save significant time compared to creating a network definition and building an engine from it.

Create a global object of type `ILogger`. It is a required argument to various methods of TensorRT API. Here is an example demonstrating the creation of the logger:

```
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;
```

Use the TensorRT API stand-alone function `createInferBuilder(gLogger)` to create an object of type `IBuilder`. For more information, see IBuilder class reference.

Use method `IBuilder::createNetworkV2` to create an object of type `INetworkDefinition`.

Create one of the available parsers (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();`

Call method `IParser::parse()` to read the model file and populate the TensorRT network.

Call method `IBuilder::buildEngineWithConfig()` to create an object of type `ICudaEngine`.

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

Create and use an execution context to perform inference.

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

Use the TensorRT API stand-alone function `createInferRuntime(gLogger)` to create an object of type `IRuntime`.

Create an engine by calling method `IRuntime::deserializeCudaEngine()`. For more information about the TensorRT runtime, see the reference `IRuntime`.

The rest of the inference is identical for whether the engine was directly built from a network, or deserialized from a file.

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 samples:

- Object Detection With A TensorFlow SSD Network (sampleMNIST), located in the GitHub repository (both BVLC and NVCaffe)
- "Hello World" For TensorRT From ONNX (sampleOnnxMNIST), located in the GitHub repository
- Import A TensorFlow Model And Run Inference (sampleUffMNIST), located in the GitHub repository (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 the 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.

About this task

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 Building A Simple MNIST Network Layer By Layer (sampleMNISTAPI) located in the `opensource/sampleMNISTAPI` directory in the GitHub repository.

Procedure

1. Create the builder and the network:

```
IBuilder* builder = createInferBuilder(gLogger);
INetworkDefinition* network = builder->createNetworkV2(1U <<
 static_cast<uint32_t>(NetworkDefinitionCreationFlag::kEXPLICIT_BATCH));
```

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

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

> **Note:** 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++

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.

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

1. Create the TensorRT builder.

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

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

2. Create the TensorRT network and parser for the specific format.

   ONNX

```
nvinfer1::INetworkDefinition* network = builder->createNetworkV2(1U <<
static_cast<uint32_t>(NetworkDefinitionCreationFlag::kEXPLICIT_BATCH));
auto parser = nvonnxparser::createParser(*network, gLogger);
```

   UFF

```
nvinfer1::INetworkDefinition* network = builder->createNetworkV2(0U);auto
parser = nvuffparser::createUffParser();
```

   Caffe

```
nvinfer1::INetworkDefinition* network = builder->createNetworkV2(0U);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.

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.

About this task

For more information, see "Hello World" For TensorRT (sampleMNIST) located in the GitHub repository.

Procedure

1. Create the builder and network:

```
IBuilder* builder = createInferBuilder(gLogger);
INetworkDefinition* network = builder->createNetworkV2(0U);
```

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 a 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

The following steps illustrate how to import a TensorFlow model using the C++ Parser API.

About this task

> **Note:** For new projects, it's recommended to use the TF-TRT integration as a method for converting your TensorFlow network to use TensorRT for inference. For integration instructions, see Accelerating Inference In TF-TRT User Guide.

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.

For more information about the UFF import, see Importing A TensorFlow Model And Running Inference (sampleUffMNIST) located in the GitHub repository.

Procedure

1. Create the builder and network:

```
IBuilder* builder = createInferBuilder(gLogger);
INetworkDefinition* network = builder->createNetworkV2(0U);
```

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");
```

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

The following steps illustrate how to import an ONNX model using the C++ Parser API.

Note:

In general, the newer version of the ONNX Parser is designed to be backward compatible up to opset 7. 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 GitHub. The instructions for building can be found here: TensorRT backend for ONNX.

About this task

For more information about the ONNX import, see "Hello World" For TensorRT From ONNX (sampleOnnxMNIST) located in the GitHub repository.

> **Note:** In TensorRT 7.0, the ONNX parser only supports full-dimensions mode, meaning that your network definition must be created with the `explicitBatch` flag set. For more information, see Working With Dynamic Shapes.

Procedure

1. Create the builder and network.

```
IBuilder* builder = createInferBuilder(gLogger);
const auto explicitBatch = 1U <<
 static_cast<uint32_t>(NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
INetworkDefinition* network = builder->createNetworkV2(explicitBatch);
```

2. Create the ONNX parser:

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

   For more information, refer to the NvOnnxParser.h file.

3. Parse the model:

```
parser->parseFromFile(onnx_filename, ILogger::Severity::kWARNING);
 for (int i = 0; i < parser.getNbErrors(); ++i)
 {
  std::cout << parser->getError(i)->desc() << std::endl;
 }
```

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 to use the same GPU for building like that on which the optimized engine will run.

About this task

The `IBuilderConfig` 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 lead 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.

One particularly important property is the maximum workspace size.

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

Procedure

1. Build the engine using the builder object:

```
IBuilderConfig* config = builder->createBuilderConfig();
config->setMaxWorkspaceSize(1 << 20);
ICudaEngine* engine = builder->buildEngineWithConfig(*network, *config);
```

   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();
config->destroy();
builder->destroy();
```

2.3.1. Builder Layer Timing Cache

Building an engine can be time-consuming since the builder needs to time candidate kernels for every layer. To reduce the builder time, TensorRT sets up a layer timing cache to keep the layer profiling information during the builder phase.

If there are other layers with the same input/output tensor configuration and layer parameters, then the TensorRT builder will skip profiling and reuse the cached result for the repeated layers. The layer timing cache is turned on by default. It can be turned off by setting the builder flag.

```
...
config->setFlag(BuilderFlag::kDISABLE_TIMING_CACHE);
```

2.4. Serializing A Model In C++

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.

About this task

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 running inference. Therefore, after the engine is built, users typically want to serialize it for later use.

> **Note:** 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 the platforms and the TensorRT version).

Procedure

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.

Procedure

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.

   For more information, see `setBindingDimension()` and `setOptimizationProfile()` for dynamic shape models.

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->enqueueV2(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 `enqueueV2()` 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 kernel (and possibly `memcpy()`) are complete, use standard CUDA synchronization mechanisms such as events, or waiting on the stream.

   For more information, refer to `enqueue()` for implicit batch networks and `enqueueV2()` for explicit batch networks. In the event that asynchronous is not wanted, see `execute()` and `executeV2()`.

2.6. Memory Management In C++

TensorRT provides two mechanisms to allow the application of 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. This is useful if your application wishes to control all GPU memory and sub allocate to TensorRT, instead of having TensorRT allocate directly from CUDA.

Once the interface is implemented, call

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

on the `IBuilder` or `IRuntime` interfaces. All device memory will then be 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.

Procedure

1. Request a refittable engine before building it:

   ```
   ...
   config->setFlag(BuilderFlag::kREFIT)
   builder->buildEngineWithConfig(network, config);
   ```

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 are 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();
```

Results

The updated engine behaves as 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.

2.8. Algorithm Selection

TensorRT provides a mechanism to control the algorithm selection for different layers in a network. The default behavior of TensorRT is to choose the algorithms that globally minimize execution time of the engine. `IAlgorithmSelector`

The application can supply a custom algorithm selector for use during engine build by implementing the `IAlgorithmSelector` interface. Once the interface is implemented, call:

```
config.setAlgorithmSelector(&selector);
```

where `config` is the `createBuilderConfig` that will be passed to `IBuilder::createBuilderConfig` to build the engine and `selector` is an instance of your class derived from `IAlgorithmSelector`.

`IAlgorithmSelector::selectAlgorithms`

This method lets the application guide algorithm selection. The method is given the algorithm context for a layer and a list of `IAlgorithm` choices, applicable to that context. You can use

your override of this method to indicate which choices TensorRT should consider, based on whatever heuristic you like, or return all the choices if TensorRT should do all the choosing.

The choices returned from `selectAlgorithm` restrict the range of algorithms allowed for some layers. The builder will do global minimization with the allowed choices. If no choice is returned, TensorRT falls back to its default behavior. You can unset `BuilderFlag::kSTRICT_TYPES`, to avoid this fallback and get an error if the override returns an empty list. If the override returns a single choice, it's guaranteed to be used.

`IAlgorithmSelector::reportAlgorithms`

The override `reportAlgorithms` can be used to record the final choices made by TensorRT for each layer. TensorRT invokes `reportAlgorithms` after all calls to `selectAlgorithms`, for a given optimization profile. To replay choices from an earlier build in a later build, make method `selectAlgorithms` return the same choices that method `reportAlgorithms` reported for the earlier build, as described in the Determinism And Reproducibility In The Builder section.

Note:

- The notion of a "layer" in Algorithm Selection is different from `ILayer` in `INetworkDefinition`. The "layer" in the former may be equivalent to a conglomeration of multiple `ILayer` due to fusion optimizations.
- Picking the fastest algorithm in `selectAlgorithms` may not get the best performance for the overall network. TensorRT optimizes for minimum timing of the whole network, possibly departing from locally greedy choices in exchange for less reformatting overhead.
- Method `reportAlgorithms` doesn't provide timing and workspace requirements of an `IAlgorithm`. Method `selectAlgorithms` can be used to query that information.
- The sequence of `IAlgorithmContext` and `IAlgorithm`, as well as the timing for each `IAlgorithm`, may not be the same for each build.

2.8.1. Determinism And Reproducibility In The Builder

The default behavior of TensorRT is to choose layer implementations that globally minimize the execution time of the engine.

Trial execution times for an algorithm are almost never identical, however, if two algorithms have similar timings, the same algorithm may not turn out to be faster every time. As a consequence, the algorithm selected by TensorRT may be different for every build, even if the network and build config is unchanged.

However, the Algorithm Selection API can be used to build TensorRT engines deterministically. For more information, see Algorithm Selection. The method `IAlgorithmSelector::selectAlgorithms` lets you select the algorithm for a layer from a

list of choices. By always returning the same choice, you can force deterministic choice for that layer.

`IAlgorithmSelector` also lets you reproduce the same implementations. `IAlgorithmSelector::reportAlgorithms` can be used to cache the algorithm choices made by TensorRT, based on the default behavior or rule set by `selectAlgorithms`. `selectAlgorithms` can then be used to choose the algorithms recorded in this cache. If you return the same algorithm choice for each combination of `layerName`, implementation, tactic and input/output formats, then you will always get the same engine.

Class `sampleAlgorithmSelector` demonstrates how to use the algorithm selector to achieve determinism and reproducibility in the builder.

Chapter 3. Using The Python API

The following sections highlight the NVIDIA® 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 are 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, in automotive. For more information about the C++ API, see Using The C++ API.

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

3.1. Importing TensorRT Into Python

Procedure

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)
- Supports ONNX releases up to ONNX 1.6, and ONNX opsets 7 to 11, 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.

Note: 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.

About this task

For more information about using the Python API to create a network and engine, see the "Hello World" For TensorRT Using PyTorch And Python (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 step-by-step instructions, 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.

About this task

For more information, see the Introduction To Importing Caffe, TensorFlow And ONNX Models Into TensorRT Using Python (introductory_parser_samples) sample.

Procedure

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.

About this task

This sample can be found in the `<site-packages>/tensorrt/samples/python/end_to_end_tensorflow_mnist` directory. For more information, see the "Hello World" For TensorRT Using TensorFlow And Python (end_to_end_tensorflow_mnist) sample.

Procedure

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.

About this task

For more information, see the Introduction To Importing Caffe, TensorFlow And ONNX Models Into TensorRT Using Python (introductory_parser_samples) sample.

Note:

In general, the newer version of the OnnxParser is designed to be backward compatible, therefore, encountering a model file produced by an earlier version of the 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 the Object Detection With The ONNX TensorRT Backend In Python (yolov3_onnx) sample.

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 GitHub. The instructions for building can be found here: TensorRT backend for ONNX.

In TensorRT 7.0, the ONNX parser only supports full-dimensions mode, meaning that your network definition must be created with the `explicitBatch` flag set. For more information, see Working With Dynamic Shapes.

Procedure

1. Import TensorRT:

```
import tensorrt as trt
```

2. Create the builder, network, and parser:

```
EXPLICIT_BATCH = 1 << (int)(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
with trt.Builder(TRT_LOGGER) as builder, builder.create_network(EXPLICIT_BATCH) as
 network, trt.OnnxParser(network, TRT_LOGGER) as parser:
with open(model_path, 'rb') as model:
if not parser.parse(model.read()):
    for error in range(parser.num_errors):
        print(parser.get_error(error))
```

3.2.6. Importing From PyTorch And Other Frameworks

About this task

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.

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 to use the same GPU for building like that on which the optimized engine will run.

About this task

The `IBuilderConfig` 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 lead 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.

One particularly important property is the maximum workspace size.

- Layer algorithms often require temporary workspace. This parameter limits the maximum size that any layer in the network can use. If an 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 Introduction To Importing Caffe, TensorFlow And ONNX Models Into TensorRT Using Python (introductory_parser_samples) sample.

Procedure

1. Build the engine using the builder object:

```
with trt.Builder(TRT_LOGGER) as builder, builder.create_builder_config() as config:
    config.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 builder.build_engine(network, config) 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

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.

About this task

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.

Note: 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 the 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)
```

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.

Procedure

1. Allocate some host and device buffers for inputs and outputs. This example assumes that `context.all_binding_dimensions == True` and that the engine has a single input at `binding_index=0` and a single output at `binding_index=1`:

```
# 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(trt.volume(context.get_binding_shape(0)),
 dtype=np.float32)
  h_output = cuda.pagelocked_empty(trt.volume(context.get_binding_shape(1)),
 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_v2(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

NVIDIA® 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.

TensorRT layers, with TopK excluded, are expected to work with zero workspace size, however, the precision requested may be ignored if there's no implementation that uses zero workspaces. In the latter case, the layer will run on FP32 even if the precision is set to something else.

4.1. Adding Custom Layers Using The C++ API

A custom layer is implemented by extending the class `IPluginCreator` and one of TensorRT's base classes for plugins.

`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 the network build phase and deserialize it during inference.

You must derive your plugin class from one of the base classes for plugins. They have varying expressive power with respect to supporting inputs/outputs with different types/formats or networks with dynamic shapes. The table below summarizes the base classes, ordered from least expressive to most expressive.

Table 1. Base classes, ordered from least expressive to most expressive

	Introduced in TensorRT version?	Mixed input/output formats/types	Dynamic shapes?
`IPluginV2Ext`	5.1	Limited	No

	Introduced in TensorRT version?	Mixed input/output formats/types	Dynamic shapes?
IPluginV2IOExt	6.0.1	General	No
IPluginV2DynamicExt	6.0.1	General	Yes

All of these base classes include versioning support and help enable custom layers that support other data formats besides NCHW and single precision.

Note: If using either IPluginV2Ext, IPluginV2IOExt, or IPluginV2DynamicExt, you should always provide an FP32 implementation of the plugin in order to allow the plugin to properly operate with any network.

Note: In versions of TensorRT prior to 6.0.1, you derived custom layers from IPluginV2 or IPluginV2Ext. While these APIs are still supported, we highly encourage you to move to IPluginV2IOExt or IPluginV2DynamicExt 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. The TensorRT library contains plugins that can be loaded into your application. For a complete list of shipped plugins, see GitHub: TensorRT plugins.

Note: 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.

Note: 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 for 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)
```

Note: `pluginData` should allocate the `PluginField` entries on the heap before passing to `createPlugin`.

Note: The `createPlugin` method above will create a new plugin object on the heap and returns a 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: Adding A Custom Layer Using C++

To add a custom layer in C++, derive it from one of the base classes described in Adding Custom Layers Using The C++ API. Because this example does not need dynamic shapes, it uses `IPluginV2IOExt`.

For Caffe based networks, if using the TensorRT Caffe Parser, you will also derive classes from `nvcaffeparser1::IPluginFactoryExt` (for plugins of type `IPluginExt`) and `nvinfer1::IPluginFactory`. 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 IPluginV2IOExt
{
 ...override all pure virtual methods of IPluginV2IOExt with definitions for your plugin.  Do
 not override the TRT_DEPRECATED methods.
};

class MyPluginFactory : public nvinfer1::IPluginFactory
{
 ...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 class `nvinfer1::IPluginFactory`.

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

If the TensorRT Caffe parser must handle your plugin, the steps are similar to Example 1, but you need to implement the `nvcaffeparser1::IPluginFactoryV2` and `IPluginCreator` classes instead and register them.

```
class FooPlugin : public IPluginV2IOExt
{
 ...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:

- Adding A Custom Layer To Your TensorRT Network (samplePlugin), located in the GitHub repository, has a user implemented the plugin.
- Object Detection With Faster R-CNN (sampleFasterRCNN), located in the GitHub repository, uses plugins registered with the TensorRT Plugin Registry.

4.1.3. Example: 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.

About this task

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

Procedure

1. Implement the `IPluginV2` and `IPluginCreator` classes as shown in Example: Adding A Custom Layer Using C++.

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.

   Object Detection With A TensorFlow SSD Network (sampleUffSSD) located in the GitHub repository, 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.

4.1.4. Example: Adding A Custom Layer With Dynamic Shape Support Using C++

To support dynamic shapes, your plugin must be derived from `IPluginV2DynamicExt`. The factory/creator and registry parts are similar to Example1, therefore the steps will not be repeated here.

About this task

`BarPlugin` is a plugin with two inputs and two outputs where:

- The first output is a copy of the second input
- The second output is the concatenation of both inputs, along the first dimension and all types/formats must be the same and be linear formats

`BarPlugin` needs to be derived as follows:

```
class BarPlugin : public IPluginV2DynamicExt
{
 ...override virtual methods inherited from IPluginV2DynamicExt.
};
```

The inherited methods are all pure virtual methods, so the compiler will remind you if you forget one.

The four methods that are affected by dynamic shapes are:

- `getOutputDimensions`
- `supportsFormatCombination`
- `configurePlugin`
- `enqueue`

The override for `getOutputDimensions` returns symbolic *expressions* for the output dimensions in terms of the input dimensions. Build the expressions from the expressions for the inputs, using the `IExprBuilder` passed into `getOutputDimensions`. In the example, the dimensions of the second output are the same as the dimensions of the first input, so no new expression has to be built for case 1.

```
DimsExprs BarPlugin::getOutputDimensions(int outputIndex,
    const DimsExprs* inputs, int nbInputs,
    IExprBuilder& exprBuilder)
{
    switch (outputIndex)
    {
    case 0:
    {
        // First dimension of output is sum of input
        // first dimensions.
        DimsExprs output(inputs[0]);
        output.d[0] =
            exprBuilder.operation(DimensionOperation::kSUM,
                inputs[0].d[0], inputs[1].d[0]);
    return output;
    }
    case 1:
        return inputs[0];
    default:
         throw std::invalid_argument("invalid output");
}
```

The override for `supportsFormatCombination` must indicate whether a format combination is allowed. The interface indexes the inputs/outputs uniformly as "connections", starting at 0 for the first input, then the rest of the inputs in order, followed by numbering the outputs. In the example, the inputs are connections 0 and 1, and the outputs are connections 2 and 3.

TensorRT uses `supportsFormatCombination` to ask whether a given combination of formats/types are okay for a connection, given formats/types for lesser indexed connections. So the override can assume that lesser indexed connections have already been vetted and focus on the connection with index `pos`.

```
bool BarPlugin::supportsFormatCombination(int pos, const PluginTensorDesc* inOut, int
 nbInputs, int nbOutputs) override
{
    assert(0 <= pos && pos < 4);
    const auto* in = inOut;
    const auto* out = inOut + nbInputs;
    switch (pos)
    {
    case 0: in[0].format == TensorFormat::kLINEAR;
    case 1: return in[1].type == in[0].type &&
                   in[0].format == TensorFormat::kLINEAR;
    case 2: return out[0].type == in[0].type &&
                   out[0].format == TensorFormat::kLINEAR;
    case 3: return out[1].type == in[0].type &&
                   out[1].format == TensorFormat::kLINEAR;
    }
    throw std::invalid_argument("invalid connection number");
```

```
}
```

The local variables in and `out` here allow inspecting `inOut` by input or output number instead of connection number.

> **Important:** The override may inspect the format/type for a connection with an index less than `pos`, but must never inspect the format/type for a connection with an index greater than `pos`. The example uses `case 3` to check connection 3 against connection 0, and not use `case 0` to check connection 0 against connection 3.

TensorRT uses `configurePlugin` to set up a plugin at runtime. Our plugin doesn't need `configurePlugin` to do anything, so it's a no-op:

```
void BarPlugin::configurePlugin(
    const DynamicPluginTensorDesc* in, int nbInputs,
    const DynamicPluginTensorDesc* out, int nbOutputs) override
{
}
```

If the plugin needed to know the minimum or maximum dimensions it might encounter, it can inspect the field `DynamicPluginTensorDesc::min` or `DynamicPluginTensorDesc::max` for any input or output. Format and build-time dimension information can be found in `DynamicPluginTensorDesc::desc`. Any runtime dimensions will appear as -1. The actual dimension is supplied to `BarPlugin::enqueue`.

Finally, the override `BarPlugin::enqueue` has to do the work. Since shapes are dynamic, enqueue is handed a `PluginTensorDesc` that describes the actual dimensions, type, and format of each input and output.

4.1.5. Example: Adding A Custom Layer With INT8 I/O Support Using C++

To support INT8 I/O, your plugin can be derived from either `IPluginV2IOExt` or `IPluginV2DynamicExt`.

The general steps are similar to Example 1: Adding A Custom Layer Using C++ For Caffe and Example 3: Adding A Custom Layer With Dynamic Shape Support Using C++, therefore the repeated parts (factory/creator and registry) will not be presented here.

`UffPoolPluginV2` is a plugin to demonstrate how to extend INT8 I/O for the custom pooling layer. The derivation is as follows:

```
class UffPoolPluginV2 : public IPluginV2IOExt
{
    ...override virtual methods inherited from IPluginV2IOExt.
};
```

Most of the pure virtual methods are common to plugins. The main methods that affect INT8 I/O are:

- `supportsFormatCombination`
- `configurePlugin`
- `enqueue`

The override for `supportsFormatCombination` must indicate which INT8 I/O combination is allowed. The usage of this interface is similar to Example 3: Adding A Custom Layer With Dynamic Shape Support Using C++. In this example, the supported I/O tensor format is linear CHW while INT32 is excluded, but the I/O tensor must have the same data type.

```
bool UffPoolPluginV2::supportsFormatCombination(int pos, const PluginTensorDesc* inOut, int
 nbInputs, int nbOutputs) const override
{
    assert(nbInputs == 1 && nbOutputs == 1 && pos < nbInputs + nbOutputs);
    bool condition = inOut[pos].format == TensorFormat::kLINEAR;
    condition &= inOut[pos].type != DataType::kINT32;
    condition &= inOut[pos].type == inOut[0].type;
    return condition;
}
```

Important:

- If INT8 auto-calibration must be used with a network with INT8 I/O plugins, the FP32 I/O variant should be supported by the plugin as it is used by FP32 calibration graph.
- If the FP32 I/O variant is not supported, or INT8 auto-calibration is not used, all required INT8 I/O tensors scales should be set explicitly.
- Auto-calibration won't generate dynamic range for plugin internal tensors. INT8 I/O plugins should calculate their own per-tensor dynamic range for internal tensors for the purpose of quantization or dequantization.

TensorRT invokes `configurePlugin` method to pass the information to the plugin through `PluginTensorDesc`, which are stored as member variables, serialized and deserialized.

```
void UffPoolPluginV2::configurePlugin(const PluginTensorDesc* in, int nbInput, const
 PluginTensorDesc* out, int nbOutput)
{
    ...
    mPoolingParams.mC = mInputDims.d[0];
    mPoolingParams.mH = mInputDims.d[1];
    mPoolingParams.mW = mInputDims.d[2];
    mPoolingParams.mP = mOutputDims.d[1];
    mPoolingParams.mQ = mOutputDims.d[2];
    mInHostScale = in[0].scale >= 0.0f ? in[0].scale : -1.0f;
    mOutHostScale = out[0].scale >= 0.0f ? out[0].scale : -1.0f;
}
```

Where INT8 I/O scales per tensor can be obtained from `PluginTensorDesc::scale`.

Finally, the override `UffPoolPluginV2::enqueue` has to do the work. It includes a collection of core algorithms to execute the custom layer at runtime by using the actual batch size, inputs, outputs, cuDNN stream, and the information configured.

```
int UffPoolPluginV2::enqueue(int batchSize, const void* const* inputs, void** outputs, void*
 workspace, cudaStream_t stream)
{
    ...
    CHECK(cudnnPoolingForward(mCudnn, mPoolingDesc, &kONE, mSrcDescriptor, input, &kZERO,
 mDstDescriptor, output));
    ...
    return 0;
}
```

4.1.6. Example: Implementing A GELU Operator Using The C++ API

To implement a GELU operator, we need to add a group of ElementWise and Unary layers in the network.

The GELU equation is:

$$GELU(x)=0.5x(1+tanh[\sqrt{2/\pi}(x+0.044715x^3)])$$

1. Prepare the constant value:

```
const float f3 = 3.0f;
const float x3Coeff = 0.044715f;
const float sqrt2OverPi = 0.7978846f;
const float f1 = 1.0f;
const float f05 = 0.5f;
```

2. Implement the GELU operator:

```
auto dim = nvinfer1::Dims3{1, 1, 1};
// y = x ^ 3
auto c3 = network->addConstant(dim, Weights{DataType::kFLOAT, &f3, 1});
auto pow1 = network->addElementWise(*x->getOutput(0), *c3->getOutput(0),
 ElementWiseOperation::kPOW);
// y = y * 0.044715f
auto cX3Coeff = network->addConstant(dim, Weights{DataType::kFLOAT, &x3Coeff, 1});
auto mul1 = network->addElementWise(
    *pow1->getOutput(0), *cX3Coeff->getOutput(0), ElementWiseOperation::kPROD);
// y = y + x
auto add1 = network->addElementWise(*mul1->getOutput(0), *x->getOutput(0),
 ElementWiseOperation::kSUM);
// y = y * 0.7978846f
auto cSqrt2OverPi = network->addConstant(dim, Weights{DataType::kFLOAT, &sqrt2OverPi,
 1});
auto mul2 = network->addElementWise(*add1->getOutput(0), *cSqrt2OverPi->getOutput(0),
 ElementWiseOperation::kPROD);
// y = tanh(y)
auto tanh1 = network->addActivation(*mul2->getOutput(0), ActivationType::kTANH);
// y = y + 1
auto c1 = network->addConstant(dim, Weights{DataType::kFLOAT, &f1, 1});
auto add2 = network->addElementWise(*tanh1->getOutput(0), *c1->getOutput(0),
 ElementWiseOperation::kSUM);
// y = y * 0.5
auto c05 = network->addConstant(dim, Weights{DataType::kFLOAT, &f05, 1});
auto mul3 = network->addElementWise(*add2->getOutput(0), *c05->getOutput(0),
 ElementWiseOperation::kPROD);
// y = y * x
auto y = network->addElementWise(*mul3->getOutput(0), *x->getOutput(0),
 ElementWiseOperation::kPROD);
```

Note: Considering that GELU is not a linear function, set the precision of every layer to FP32 when the network is set to run in INT8 mode.

For more information about layer fusion related to GELU, see the TensorRT Best Practices Guide.

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 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.

4.2.1. Example: 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: 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.

About this task

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: Adding A Custom Layer Using C++.

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

Procedure

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 Adding A Custom Layer To Your TensorFlow Network In TensorRT In Python (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 TensorFlow 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 Object Detection With A TensorFlow SSD Network (sampleUffSSD), located in the GitHub repository, 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.

> **Note:** 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 the network creation step as shown in Adding Custom Layers Using The C++ API.

The Adding A Custom Layer To Your Network In TensorRT (samplePlugin) sample, located in the GitHub repository, illustrates how to extend `nvcaffeparser1::IPluginFactoryExt` to use custom layers, while Object Detection With A TensorFlow SSD Network (sampleUffSSD) uses the UFF Parser to use custom layers.

ONNX

For ONNX models, the ONNX parser will automatically attempt to import unrecognized ops as plugins. If a plugin with the same `op_type` as the node is found in the registry, the parser will parse the plugin field parameters from the ONNX model, and use the corresponding creator to create a plugin instance. By default, it will attempt to load the plugin version `1`. This behavior can be overridden by setting a `plugin_version` string parameter in the corresponding ONNX node.

In some cases, you may want to modify an ONNX graph prior to importing it into TensorRT; for example, to add a `plugin_version` attribute as mentioned above, or to replace a set of ops with a plugin node. To accomplish this, you can use the ONNX GraphSurgeon utility.

For the Python usage of custom layers with TensorRT, refer to:

- Adding A Custom Layer To Your TensorFlow Network In TensorRT In Python (uff_custom_plugin) and Object Detection With SSD In Python (uff_ssd) samples for UFF networks
- TensorRT Inference Of ONNX Models With Custom Layers (onnx_packnet) sample for ONNX

4.3.1. Example: 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: 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 Object Detection With SSD In Python (uff_ssd) 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 derive classes from both `IPluginCreator` and one of the plugin base classes described in Adding Custom Layers Using The C++ API. 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 6.x.x To TensorRT 7.x.x

While `IPluginV2` and `IPluginV2Ext` interfaces are still supported for backward compatibility with TensorRT 5.1 and 6.0.x respectively, we recommend that you write new plugins or refactor existing ones to target the `IPluginV2DynamicExt` or `IPluginV2IOExt` interface instead, as described in section 4.1.

In order to use the most recent Plugin layer features, your custom plugin should implement the `IPluginV2DynamicExt` or `IPluginV2IOExt` interface.

The new features in `IPluginV2DynamicExt` are as follows:

```
virtual DimsExprs getOutputDimensions(int outputIndex, const DimsExprs* inputs, int nbInputs,
 IExprBuilder& exprBuilder) = 0;

virtual bool supportsFormatCombination(int pos, const PluginTensorDesc* inOut, int nbInputs,
 int nbOutputs) = 0;

virtual void configurePlugin(const DynamicPluginTensorDesc* in, int nbInputs, const
 DynamicPluginTensorDesc* out, int nbOutputs) = 0;

virtual size_t getWorkspaceSize(const PluginTensorDesc* inputs, int nbInputs, const
 PluginTensorDesc* outputs, int nbOutputs) const = 0;

virtual int enqueue(const PluginTensorDesc* inputDesc, const PluginTensorDesc* outputDesc,
 const void* const* inputs, void* const* outputs, void* workspace, cudaStream_t stream) = 0;
```

The new features in `IPluginV2IOExt` are as follows:

```
virtual void configurePlugin(const PluginTensorDesc* in, int nbInput, const PluginTensorDesc*
 out, int nbOutput) = 0;

virtual bool supportsFormatCombination(int pos, const PluginTensorDesc* inOut, int nbInputs,
 int nbOutputs) const = 0;
```

Guidelines for migration to `IPluginV2DynamicExt` or `IPluginV2IOExt`:

- `getOutputDimensions` implements the expression for output tensor dimensions given the inputs.
- `supportsFormatCombination` checks if the plugin supports the format and datatype for the specified input/output.

- `configurePlugin` mimics the behavior of equivalent `configurePlugin` in `IPluginV2Ext` but accepts tensor descriptors.
- `getWorkspaceSize` and `enqueue` mimic the behavior of equivalent APIs in `IPluginV2Ext` but accept tensor descriptors.

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

4.4.1.1. Migrating Plugins From TensorRT 5.x.x To TensorRT 6.x.x

The `IPluginV2` interface is still supported, however, we recommend that you write new plugins with the `IPluginV2Ext` interface and migrate any existing plugin implementations to the `IPluginV2Ext` interface.

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 set up 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 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 selects 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 and is broadcast across the batch.

IPluginV2IOExt

This is called by the builder prior to `initialize()`. It provides an opportunity for the layer to make algorithm choices on the basis of I/O PluginTensorDesc and the maximum batch size.

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 "").

4.4.4. Persistent LSTM Plugin

The following section describes the new `Persistent LSTM` plugin. The `Persistent LSTM` plugin supports half-precision persistent LSTM. To create a `Persistent LSTM` plugin in the network, you need to call:

```
auto creator = getPluginRegistry()->getPluginCreator("CgPersistentLSTMPlugin_TRT", "1")

IPluginV2* cgPersistentLSTMPlugin = creator->createPlugin("CgPersistentLSTMPlugin_TRT", &fc);
```

`fc` is a `PluginField` array that consists of 4 parameters:

- `hiddenSize`: This is an INT32 parameter that specifies the hidden size of LSTM.
- `numLayers`: This is an INT32 parameter that specifies the number of layers in LSTM.
- `bidirectionFactor`: This is an INT32 parameter that indicates whether LSTM is bidirectional. If LSTM is bidirectional, the value should be set to 2, otherwise, the value is set to 1.
- `setInitialStates`: This is an INT32 parameter that indicates whether LSTM has initial state and cell values as inputs. If it is set to 0, the initial state and cell values will be zero. It is recommended to use this flag instead of providing zero state and cell values as inputs for better performance.

The plugin can be added to the network by calling:

```
auto lstmLayer = network->addPluginV2(&inputs[0], 6, *cgPersistentLSTMPlugin);
```

`inputs` is a vector of `ITensor` pointers with 6 elements in the following order:

1. `input`: These are the input sequences to the LSTM.
2. `seqLenTensor`: This is the sequence length vector that stores the effective length of each sequence.
3. `weight`: This tensor consists of all weights needed for LSTM. Even though this tensor is 1D, it can be viewed with the following 3D indexing `[isW, layerNb, gateType]`. `isW` starts from `false` to `true` suggesting that the first half of weight is recurrent weight and the second half is input weight. `layerNb` starts from 0 to `numLayers*bidirectionFactor` such that the first layer is the forward direction of the actual layer and the second layer is the backward direction. The `gateType` follows this order: `input, cell, forget` and `output`.
4. `bias`: Similar to weight, this tensor consists of all biases needed for LSTM. Even though this tensor is 1D, it can be viewed with the following 3D indexing `[layerNb, isW, gateType]`. Notice the slight difference between bias and weight.
5. `initial hidden state`: The pointer should be set to null if `setInitialStates` is `0`. Otherwise, the tensor should consist of the initial hidden state values with the following coordinates `[batch index, layerNb, hidden index]`. `batch index` indicates the index within a batch and the `hidden index` is the index to vectors of `hiddenSize` length.

6. `initial cell state`: The pointer should be set to null if `setInitialStates` is `0`. Otherwise, the tensor should consist of the initial hidden state values with the following coordinates `[batch index, layerNb, hidden index]`.

4.5. Best Practices For Custom Layers Plugin

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

Debugging 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. NVIDIA® 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 the 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 config flag `BuilderFlag::kSTRICT_TYPES` to force the network or layer precision, which may not have optimal performance. The 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. Using both INT8 and FP16 mode would allow TensorRT to choose from FP32, FP16, and INT8 kernels, thus resulting in the most optimal engine from 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)
```

Note: 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 ITopKLayer.

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 match 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 with any other builder supported precision is faster, TensorRT will use it and issue a warning.

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

```
IBuilderConfig * config = builder->createBuilderConfig();config-
>setFlag(BuilderFlag::kSTRICT_TYPES)
```

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.

Note: When running INT8 precision on a GPU, the dimension of the layer tensor should be greater than or equal to 3. A layer tensor dimension less than 3 is not supported in TensorRT 6.0.1.

See Performing Inference In INT8 Precision (sampleINT8API) located in the GitHub repository for an example of running mixed-precision inference with these APIs.

5.1.2. Enabling TF32 Inference Using C++

TensorRT allows use of TF32 Tensor Cores by default. When computing inner products, such as during convolution or matrix multiplication, TF32 execution does the following:

- Rounds the FP32 multiplicands to FP16 precision but keeps FP32 dynamic range.
- Computes an exact product of the rounded multiplicands.
- Accumulates the products in an FP32 sum.

TF32 Tensor Cores can speed up networks using FP32, typically with no loss of accuracy. It is more robust than FP16 for models which require high dynamic range for weights or activations.

There is no guarantee that TF32 Tensor Cores actually used and no way to force it. TensorRT can fall back on FP32 at any time, and always falls back if the platform does not support TF32. You can check for platform support by querying an `IBuilder` like this:

```
bool hasTf32 = builder->platformHasTf32()
```

To disable TF32, clear the flag `BuilderFlag::kTF32` in in your `IBuilderConfig`, like this:

```
config->clearFlag(BuilderFlag::kTF32);
```

If TF32 works for you, we nonetheless Enabling FP16 Inference Using C++, since it may yield faster execution than TF32.

Setting the environment variable `NVIDIA_TF32_OVERRIDE=0` when building an engine disables the use of TF32, despite setting `BuilderFlag::kTF32`. This environment variable, when set to `0`, overrides any defaults or programmatic configuration of NVIDIA libraries so they never accelerate FP32 computations with TF32 Tensor Cores. This is meant to be a debugging tool only, and no code outside NVIDIA libraries should change the behavior based on this environment variable. Any other setting besides `0` is reserved for future use.

WARNING: Setting the environment variable `NVIDIA_TF32_OVERRIDE` to a different value when the engine is run may cause unpredictable precision/performance effects. It is best left unset when an engine is run.

5.1.3. Enabling FP16 Inference Using C++

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

```
config->setFlag(BuilderFlag::kFP16);
```

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:

```
config->setFlag(BuilderFlag::kSTRICT_TYPES)
```

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

See Building And Running GoogleNet In TensorRT (sampleGoogleNet) and "Hello World" For TensorRT (sampleMNIST) located in the GitHub repository for running FP16 inference.

5.1.4. Enabling INT8 Inference Using C++

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.

Setting the builder flag enables INT8 precision inference.

```
config->setFlag(BuilderFlag::kINT8);
```

TensorRT supports symmetric quantization with a 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

a scenario is possible if INT8 calibration does not generate a satisfactory dynamic range for certain tensors.

For more information, see Performing Inference In INT8 Precision (sampleINT8API) located in the GitHub repository.

5.1.4.1. 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 of 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 Performing Inference In INT8 Precision (sampleINT8API) located in the GitHub repository.

5.1.4.2. INT8 Calibration Using C++

INT8 calibration provides an alternative to generate per-tensor dynamic range. Calibrators compute a scale per tensor. This method can be categorized as a 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 (typical batches from the training set.) Experiments indicate that about 500 images are sufficient for calibrating ImageNet classification networks.

About this task

To provide calibration data to TensorRT, implement the `IInt8Calibrator` interface. TensorRT provides multiple variants of `IInt8Calibrator`:

`IEntropyCalibratorV2`

This is the recommended calibrator and is required for DLA. Calibration happens before Layer fusion by default. This is recommended for CNN based networks.

`IMinMaxCalibrator`

This is the preferred calibrator for NLP tasks for all backends. Calibration happens before Layer fusion by default. This is recommended for BERT like networks.

`IEntropyCalibrator`

This is the legacy entropy calibrator.This is less complicated than a legacy calibrator and produces better results. Calibration happens after Layer fusion by default. See `kCALIBRATION_BEFORE_FUSION` for enabling calibration before fusion.

ILegacyCalibrator

This calibrator is for compatibility with TensorRT 2.0 EA. This calibrator requires user parameterization, and is provided as a fallback option if the other calibrators yield poor results. Calibration happens after Layer fusion by default. See `kCALIBRATION_BEFORE_FUSION` for enabling calibration before fusion. Users can customize this calibrator to implement percentile max, like 99.99% percentile max is proved to have best accuracy for BERT. For more information, refer to the Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation paper.

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:

```
config->setInt8Calibrator(calibrator.get());
```

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 Performing Inference In INT8 Using Custom Calibration (sampleINT8) located in the GitHub repository.

5.1.5. Working With Explicit Precision Using C++

TensorRT supports explicit precision networks in which precision of all layers and tensors in the network can be inferred. This feature enables the import of quantized models with explicit quantizing and dequantizing scale layers (QDQ scale layers) into TensorRT. The presence of QDQ scale layers in the network determine the precision transitions, (i.e from FP32 to INT8 and vice-versa). No other precisions must be set.

To create an explicit precision network, the `INetworkDefinition` has to be created with `createNetworkV2` as follows:

```
builder->createNetworkV2(1U <<
 static_cast<uint32_t>(NetworkDefinitionCreationFlag::kEXPLICIT_PRECISION)
```

A quantizing scale layer can be specified as a scale layer with output precision type of INT8. Similarly, a dequantizing scale layer can be specified as a scale layer with output precision type of FP32. See Setting The Layer Precision Using C++ for more information on setting the precision.

In an explicit precision network, weights and bias are expected to be in FP32 precision. A layer which is expected to be quantized needs to have a QDQ scale layer inserted at the inputs (including weights if they exist).

Both activation tensors and weights are quantized/dequantized using scale parameters from a QDQ scale layer. Scale parameters are generally computed using quantization aware training or determined using experiments, for example, calibration. Users must not use calibrators or provide dynamic ranges in explicit precision mode. Instead, express quantization and dequantization scales using QDQ scale layers.

TensorRT 7.1 only supports per-tensor quantization scales for both activations and weights in explicit precision mode. No shift weights are allowed for the QDQ scale layer as only symmetric quantization is supported.

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
```

For more information, see the INT8 Calibration In Python (int8_caffe_mnist) sample.

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:

```
builder.int8_mode = True
```

Similar to the C++ API, you can choose per activation tensor the dynamic range either using `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.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.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.

Procedure

1. Import TensorRT:

   ```
   import tensorrt as trt
   ```

2. Similar to test/validation files, use a set of input files as a 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 a `batchstream` object. A `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:

```
config.set_flag(trt.BuilderFlag.INT8)
config.int8_calibrator = Int8_calibrator
```

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

5.2.4. Working With Explicit Precision Using Python

To create an explicit precision network using the Python API, pass the `EXPLICIT_PRECISION` flag to the builder.

```
network_creation_flag = 1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_PRECISION)
self.network = self.builder.create_network(network_creation_flag)
```

See Setting The Layer Precision Using Python for more information on setting the precision.

Chapter 6. Working With Reformat-Free Network I/O Tensors

Requirements from Automotive Safety Integrity Level (ASIL) for safety flows require that accessing GPU address spaces should be removed from the NvMedia DLA safety path. To achieve this objective, reformat-free network I/O tensors are introduced to let you specify I/O formats that are supported by NvMedia tensor before passing the data to NVIDIA® TensorRT™.

On the other hand, the potential overhead of tensor reformatting can cause performance issues because TensorRT less than 6.0.1 assumes that network I/O tensors are FP32. In the case of multiple TensorRT sub-networks embedded into a large network, (for example, TensorFlow), with a precision of INT8 or FP16, the unavoidable I/O reformatting from and to FP32 could waste considerable memory traffic time. The same issue may also happen on the user-defined plugins. Now you can explicitly specify network I/O tensors to INT8 or FP16 formats to eliminate those unnecessary reformatting.

6.1. Building An Engine With Reformat-Free Network I/O Tensors

You can use the following API to specify formats of network I/O tensors.

C++ API:

```
network->getInput(i)->setAllowedFormats(formats);
network->getOutput(i)->setAllowedFormats(formats);
```

Where `i` is the index of network I/O tensors.

Python API:[1]

```
network.get_input(0).allowed_formats = formats
network.get_output(0).allowed_formats = formats
```

Where `formats` is the mask form of the dense enum `TensorFormat` which sets the memory layout of the tensor. For example, `1U << TensorFormat::kLINEAR` or, in Python, `1 << int(tensorrt.TensorFormat.LINEAR)`.

`BuilderFlag::kSTRICT_TYPES` or `tensorrt.BuilderFlag.STRICT_TYPES` in Python can be explicitly set to generate an engine without reformatting rather than getting the fastest path. For example:

C++ API:

[1] The TensorRT Safety 6.0.0 Release does not support the TensorRT 6.0.1 Python API.

```
builderConfig->setFlag(BuilderFlag::kSTRICT_TYPES);
```

Python API:

```
builder_config.set_flag(tensorrt.BuilderFlag.STRICT_TYPES)
```

The flag can also be set via a bitmask manner as follows.

C++ API:

```
builderConfig->setFlags(flags);
```

Python API:

```
builder_config.flags = flags
```

Where `flags` is the bit mask combination of the dense enum `BuilderFlags`. For example, `1 << BuilderFlag::kFP16 | 1<< BuilderFlag::kSTRICT_TYPES` or, in Python,

```
1 << int(tensorrt.BuilderFlag.FP16) | 1 <<
        int(tensorrt.BuilderFlag.STRICT_TYPES)
```

If TensorRT doesn't find any implementation of the reformat-free path, the following warning message displays:

```
'[W] [TRT] Warning: no implementation obeys reformatting-free rules ...'
```

As a result, the fastest path will be picked instead.

If the combination of data type and memory layout of I/O tensors is well chosen, the overall performance of the reformat-free path should be very close to the fastest path for most normal cases or else it is recommended to disable the reformat-free path.

For more information, see Specifying I/O Formats Using The Reformat Free I/O APIs (sampleReformatFreeIO) for an example of setting reformat-free network I/O tensors with these APIs using C++.

6.2. Supported Combination Of Data Type And Memory Layout of I/O Tensors

The supported settings of I/O tensors are listed in the following table.

Table 2. Supported combination of data types and memory layout.

Memory Layout \ Data Type	kINT32	kFLOAT	kHALF	kINT8
kLINEAR	Supported	Supported	Supported	Supported
kCHW2	N/A	N/A	Supported	N/A
kCHW4	N/A	N/A	Supported (including DLA)	Supported
kHWC8	N/A	N/A	Supported	N/A
kCHW16	N/A	N/A	Only for DLA	N/A
kCHW32	N/A	Supported	Supported	Supported (including DLA)

6.3. Calibration For A Network With INT8 I/O Tensors

INT8 auto-calibration is supported by INT8 I/O tensors. In this case, you will need to provide FP32 data for calibration and INT8 I/O tensors for inference.

With an INT8 I/O network, TensorRT will expect calibration data to be in FP32 precision to generate calibration cache. Calibration cache data will then be internally used by the builder during inference with INT8 I/O tensors.

This limitation of INT8 I/O networks requiring FP32 calibration data will be relaxed in future releases. For now, you can create FP32 calibration data by simply casting INT8 I/O calibration data to FP32 precision. You should also ensure that FP32 cast calibration data should be in the range `[-128.0f, 127.0f]` and can be converted to INT8 data without any precision loss.

Setting up a calibrator for a network with INT8 I/O tensors remains exactly the same as a network with FP32 I/O tensors.

6.4. Restrictions With DLA

Reformat-free network I/O is expected to work in all execution modes, however, TensorRT will insert formatting layers where required for correct functionality with DLA enabled:

- If the engine capability is `EngineCapability::kDEFAULT and GPUFallBack is allowed`, TensorRT will always insert a reformat layer at the DLA boundary. Input reformats are inserted to synchronize tensors from a CUDA address space to an NvMedia address space. Output reformats are inserted after a `FinishNVMRegion` layer to synchronize tensors from an NvMedia address space back to a CUDA address space. In this case, the Input and output regions do not need to meet column stride requirements.
- With engine capability is `EngineCapability::kSAFE_DLA`, or engine capability is `EngineCapability::kDEFAULT and GPUFallBack` is not allowed, reformat layers will not be inserted for `kCHW16`, for FP16, and `kCHW32`, for INT8. However, if you only specify `kCHW4`, additional requirements must be satisfied, otherwise TensorRT fails to build the engine when engine capability is `EngineCapability::kSAFE_DLA` or additional reformat will still be inserted when engine capability is `EngineCapability::kDEFAULT`.
  - `C` must be equal or smaller than 4.
  - The first layer must be convolution.
  - The convolution parameters must meet DLA requirements, for example:
    - Dilation must be equal to 1.
    - Group convolution is not supported.

  When using `kCHW4`, the stride must meet DLA line stride requirements. Taking a 4 dimensional `kCHW4` tensor `{N, C, H, W}` as an example. The big endian stride is:

```
stride[3]= divUp(C, components_per_element);
```

```
     stride[2] = roundUp(stride[0] * W, line_stride_req/(components_per_element *
bytes_per_component);
     stride[1] = 1;
     stride[0] = stride[2]*H;
```

Where:

- `bytes_per_component` equals `1` for INT8 and `2` for FP16
- `components_per_element` equals `4` for `kCHW4`
- `line_stride_req` equals `32` for `kCHW4`
- stride is in terms of element

If engine capability is not `EngineCapability::kSAFE_DLA`, the stride can be retrieved by `IExecutionContext::getStrides()`.

- TensorRT doesn't provide native API to run `EngineCapability::kSAFE_DLA` engine. Alternatively, you can use `EngineCapability::kSAFE_DLA` to produce a DLA loadable (or named plan file) that is only consumable by NvMedia DLA API directly and requires all input formats to match exactly without reformatting layers.
- If you specify that both `kCHW4` and `kCHW16/32` are allowed, TensorRT doesn't provide native API to query which format matches input tensors of the DLA loadable. You could use NvMedia DLA API `NvMediaDlaGetInputTensorDescriptor` to get the input tensor descriptor (see DRIVE OS Linux SDK API Reference Documentation for more details).

6.5. FAQs

This section is to help troubleshoot the most asked questions when using reformat-free network I/O tensors.

Q: Why are reformatting layers observed although there is no warning message `no implementation obeys reformatting-free rules ...`?

A: Reformat-free network I/O does not mean there are no reformatting layers inserted in the entire network. Only that the input and output network tensors have a possibility to not require reformatting layers. In other words, reformatting layers can be inserted by TensorRT for internal tensors to improve performance.

Q: What is the best practice to use reformat-free network I/O tensors for DLA?

A: First, you have to check if your network can run entirely on DLA, then try to build the network with `kSAFE_DLA` enabled and specify `kLINEAR`, `kCHW4` or `kCHW16/32` format as allowed I/O formats. If multiple formats can work, you can profile them and choose the fastest I/O format combination. If your network indeed performs better with `kCHW4`, but it doesn't work, you have to check which requirement listed in the above section is unsatisfied.

Chapter 7. Working With Dynamic Shapes

Dynamic shapes are the ability to defer specifying some or all tensor dimensions until runtime. Dynamic shapes can be used via both the C++ and Python interfaces.

The following sections provide greater detail; however, here's an overview of the steps for building an engine with dynamic shapes:

1. The network definition must not have an implicit batch dimension.

   C++

   Create the `INetworkDefinition` by calling

   ```
   IBuilder::createNetworkV2(1U <<
           static_cast<int>(NetworkDefinitionCreationFlag::kEXPLICIT_BATCH))
   ```

   Python

   Create the `tensorrt.INetworkDefinition` by calling

   ```
   create_network(1 <<
           int(tensorrt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
   ```

   These calls request that the network not have an implicit batch dimension.
2. Specify each runtime dimension of an input tensor by using `-1` as a placeholder for the dimension.
3. Specify one or more *optimization profiles* at build time that specify the permitted range of dimensions for inputs with runtime dimensions, and the dimensions for which the auto-tuner should optimize. For more information, see Optimization Profiles.
4. To use the engine:

   a). Create an execution context from the engine, the same as without dynamic shapes.

   b). Specify one of the optimization profiles from step 3 that covers the input dimensions.

   c). Specify the input dimensions for the execution context. After setting input dimensions, you can get the output dimensions that TensorRT computes for the given input dimensions.

   d). Enqueue work.

To change the runtime dimensions, repeat steps 4b and 4c, which do not have to be repeated until the input dimensions change.

7.1. Specifying Runtime Dimensions

When building a network, use `-1` to denote a runtime dimension for an input tensor. For example, to create a 3D input tensor named `foo` where the last two dimensions will be specified at runtime, and the first dimension is fixed at build time, issue the following.

C++

```
networkDefinition.addInput("foo", DataType::kFLOAT, Dims3(3, -1, -1))
```

Python

```
network_definition.add_input("foo", trt.float32,(3, -1, -1))
```

At run time, you'll need to set the input dimensions after choosing an optimization profile (see Optimization Profiles). Let the `bindingIndex` of input `foo` be `0`, and the input have dimensions `[3,150,250]`. After setting an optimization profile for the example above, you would call:

C++

```
context.setBindingDimensions(0, Dims3(3, 150, 250))
```

Python

```
context.set_binding_shape(0, (3, 150, 250))
```

At runtime, asking the engine for binding dimensions returns the same dimensions used to build the network, meaning, you will get a `-1` for each runtime dimension. For example:

C++

`engine.getBindingDimensions(0)` returns a `Dims` with dimensions `{3, -1, -1}`.

Python

`engine.get_binding_shape(0)` returns `(3, -1, -1)`.

To get the actual dimensions, which are specific to each execution context, query the execution context:

C++

`context.getBindingDimensions(0)` returns a `Dims` with dimensions `{3, 150, 250}`.

Python

`context.get_binding_shape(0)` returns `(3, 150, 250)`.

After setting input dimensions, you can also use similar calls to query dimensions of the network outputs.

7.2. Optimization Profiles

An *optimization profile* describes a range of dimensions for each network input and the dimensions that the auto-tuner should use for optimization. When using runtime dimensions, you must create at least one optimization profile at build time. Two profiles can specify disjoint or overlapping ranges.

For example, one profile might specify a minimum size of `[3,100,200]`, a maximum size of `[3,200,300]`, and optimization dimensions of `[3,150,250]` while another profile might specify min, max and optimization dimensions of `[3,200,100]`, `[3,300,400]`, and `[3,250,250]`.

To create an optimization profile, first construct an `IOptimizationProfile`. Then set the min, optimization, and max dimensions, and add it to the network configuration. Here are the calls for the first profile mentioned above for an input `foo`:

C++

```
IOptimizationProfile* profile = builder.createOptimizationProfile();
profile->setDimensions("foo", OptProfileSelector::kMIN, Dims3(3,100,200);
profile->setDimensions("foo", OptProfileSelector::kOPT, Dims3(3,150,250);
profile->setDimensions("foo", OptProfileSelector::kMAX, Dims3(3,200,300);

config->addOptimizationProfile(profile)
```

Python

```
profile = builder.create_optimization_profile();
profile.set_shape("foo", (3, 100, 200), (3, 150, 250), (3, 200, 300))
config.add_optimization_profile(profile)
```

At runtime you need to set an optimization profile before setting input dimensions. Profiles are numbered in the order they were added, starting at `0`. To choose the first optimization profile in the example, use:

C++

call `context.setOptimizationProfile(0)`

Python

set `context.active_optimization_profile = 0`

If the associated CUDA engine has dynamic inputs, the optimization profile must be set at least once with a unique profile index which is not used by other execution contexts that are not destroyed. For the first execution context that is created for an engine, profile 0 is chosen implicitly.

`setOptimizationProfile` can be called to switch between profiles. It must be called after any `enqueue()` or `enqueueV2()` operations finish in the current context. When multiple execution contexts run concurrently, it is allowed to switch to a profile which was formerly used but already released by another execution context with different dynamic input dimensions.

In an engine built from multiple profiles, there are separate binding indices for each profile. The names of input/output tensors for the *K*th profile have `[profile K]` appended to them, with *K* written in decimal. For example, if the `INetworkDefinition` had the name "`foo`", and `bindingIndex` refers to that tensor in the optimization profile with index `3`, `engine.getBindingName(bindingIndex)` returns "`foo [profile 3]`".

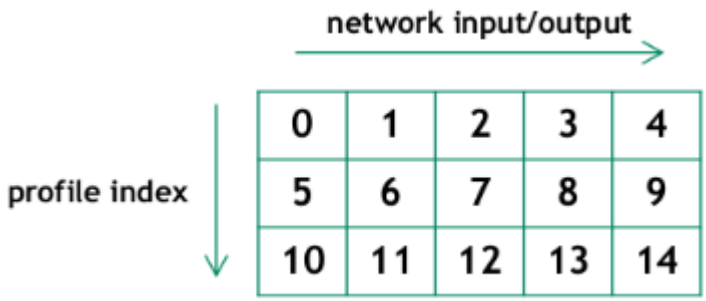
Likewise, if using `ICudaEngine::getBindingIndex(name)` to get the index for a profile *K* beyond the first profile (*K=0*), append "`[profile K]`" to the name used in the `INetworkDefinition`. For example, if the tensor was called "`foo`" in the `INetworkDefinition`, then `engine.getBindingIndex("foo [profile 3]")` returns the binding index of Tensor "`foo`" in optimization profile `3`.

Always omit the suffix for *K=0*.

7.2.1. Bindings For Multiple Optimization Profiles

TensorRT 7.1 is stricter about binding indices than its predecessors. Previously, binding indices for the wrong profile were tolerated. Consider a network with four inputs, one output, with three optimization profiles in the `IBuilderConfig`. The engine has 15 bindings, five for each optimization profile, conceptually organized as a table:

Figure 4. Optimization profile



Each row is a profile. Numbers in the table denote binding indices. The first profile has binding indices 0..4, the second has 5..9, and the third has 10..14. Prior to version 7.1, where a profile index was specified or implied, TensorRT accepted binding numbers for the wrong profile, using the binding index only to determine the column. In TensorRT 7.1, the binding index must be correct, otherwise you will see an API check failure that mentions `bindingIndexBelongsToProfile`.

However, for sake of backwards semi-compatibility, the interfaces have an "auto-correct" for the case that the binding belongs to the *first* profile but another profile was specified. In that case, TensorRT will warn about the mistake, and then choose the correct binding index from the same column.

7.3. Layer Extensions For Dynamic Shapes

Some layers have optional inputs that allow specifying dynamic shape information, and there is a new layer `IShapeLayer` for accessing the shape of a tensor at runtime. Furthermore, some layers allow calculating new shapes. The next section goes into semantic details and restrictions. Here is a summary of what you might find useful in conjunction with dynamic shapes.

`IShapeLayer` outputs a 1D tensor containing the dimensions of the input tensor. For example, if the input tensor has dimensions [2,3,5,7], the output tensor will be a four-element 1D tensor containing {2,3,5,7}. If the input tensor is a scalar, it has dimensions [], and the output tensor will be a zero-element 1D tensor containing {}.

`IResizeLayer` accepts an optional second input containing the desired dimensions of the output.

`IShuffleLayer` accepts an optional second input containing the reshape dimensions before the second transpose is applied. For example, the following network reshapes a tensor `Y` to have the same dimensions as `X`:

C++

```
auto* reshape = networkDefinition.addShuffle(Y);
reshape.setInput(1, networkDefintion.addShape(X)->getOutput(0));
```

Python

```
reshape = network_definition.add_shuffle(y)
reshape.set_input(1, network_definition.add_shape(X)->get_output(0))
```

Shuffle operations that are equivalent to identify operations on the underlying data will be omitted, if the input tensor is only used in the shuffle layer and the input and output tensors of this layer are not input and output tensors of the network. TensorRT no longer executes additional kernels or memory copies for such operations.

`ISliceLayer` accepts an optional second, third, and fourth inputs containing the start, size, and stride.

```
IConcatenationLayer, IElementWiseLayer, IGatherLayer, IIdentityLayer, and
        IReduceLayer
```

can be used to do calculations on shapes and create new shape tensors.

7.4. Restrictions For Dynamic Shapes

The following layer restrictions arise because the layer's weights have a fixed size:

- `IConvolutionLayer` and `IDeconvolutionLayer` require that the channel dimension be a build-time constant.
- `IFullyConnectedLayer` requires that the last three dimensions be build-time constants.
- `Int8` requires that the channel dimension be a build-time constant.

Values that must be build-time constants don't have to be constants at the API level. TensorRT's shape analyzer does element-by-element constant propagation through layers that do shape calculations. It's sufficient that the constant propagation discovers that a value is a build-time constant.

7.5. Execution Tensors vs. Shape Tensors

Engines using dynamic shapes employ a two-phase execution strategy.

1. Compute the shapes of all tensors
2. Stream work to the GPU.

Phase 1 is implicit and driven by demand, such as when output dimensions are requested. Phase 2 is the same as in prior versions of TensorRT. The two-phase execution puts some limits on dynamism that are important to understand.

The key limits are:

- The rank of a tensor must be determinable at build time.
- A tensor is either an *execution tensor*, *shape tensor*, or both. Tensors classified as shape tensors are subject to limits discussed below.

An *execution tensor* is a traditional TensorRT tensor. A *shape tensor* is a tensor that is related to shape calculations. It must be 0D or 1D, have type `Int32` or `Bool`, and its shape must be determinable at build time. For example, there is an `IShapeLayer` whose output is a 1D tensor containing the dimensions of the input tensor. The output is a shape tensor. `IShuffleLayer` accepts an optional second input that can specify reshaping dimensions. The second input must be a shape tensor.

Some layers are "polymorphic" with respect to the kinds of tensors they handle. For example, `IElementWiseLayer` can sum two Int32 execution tensors or sum two Int32 shape tensors. The type of tensor depends on their ultimate use. If the sum is used to reshape another tensor, then it is a "shape tensor".

7.5.1. Formal Inference Rules

The formal inference rules used by TensorRT for classifying tensors are based on a type-inference algebra. Let `E` denote an execution tensor and `S` denote a shape tensor.

`IShapeLayer` has the signature: `IShapeLayer: E → S` since it takes an execution tensor as an input and a shape tensor as an output. `IElementWiseLayer` is polymorphic in this respect, with two signatures: `IElementWiseLayer: S × S → S, E × E → E`

For brevity, let's adopt the convention that *t* is a variable denoting either class of tensor, and all *t* in a signature refers to the same class of tensor. Then the two signatures above can be written as a single polymorphic signature: `IElementWiseLayer: t × t → t`

The two-input `IShuffleLayer` has a shape tensor as the second input, and is polymorphic with respect to the first input: `IShuffleLayer (two inputs): t × S → t`

`IConstantLayer` has no inputs, but can produce a tensor of either kind, so its signature is: `IConstantLayer: → t`

Here is the complete set of rules, which also serves as a reference for which layers can be used to manipulate shape tensors:

```
IConcatenationLayer: t × t × ...→ t
IConstantLayer: → t
IElementWiseLayer: t × t → t
IGatherLayer: t × t → t
IIdentityLayer: t → t
IReduceLayer: t → t
IResizeLayer (one input): E → E
IResizeLayer (two inputs): E × S → E
ISelectLayer: t × t × t → t
IShapeLayer: E → S
IShuffleLayer (one input): t → t
IShuffleLayer (two inputs): t × S → t
ISliceLayer (one input): t → t
ISliceLayer (two inputs): t × S → t
ISliceLayer (three inputs): t × S × S → t
ISliceLayer (four inputs): t × S × S × S → t
all other layers: E × ... → E × ...
```

Because an output can be the input of more than one subsequent layer, the inferred "types" are not exclusive. For example, an `IConstantLayer` might feed into a one use that requires an execution tensor and another use that requires a shape tensor. The output of `IConstantLayer` will be classified as both, and be used in both phase 1 and phase 2 of the two-phase execution.

The requirement that the rank of a shape tensor be known at build time limits how `ISliceLayer` can be used to manipulate a shape tensor. Specifically, if the third parameter, which specifies the size of the result, is not a build-time constant, the length of the resulting shape tensor would no longer be known at build time, breaking the restriction of shape tensors to build-time shapes. Worse, it might be used to reshape another tensor, breaking the restriction that tensor ranks must be known at build time.

TensorRT's inferences can be inspected via methods `ITensor::isShapeTensor()`, which returns true for a shape tensor, and `ITensor::isExecutionTensor()`, which returns true for an execution tensor. Build the entire network first before calling these methods, because their answer can change depending on what uses of the tensor have been added.

For example, if a partially built network sums two tensors *T1* and *T2* to create tensor *T3*, and none are yet needed as shape tensors, `isShapeTensor()` returns false for all three tensors. Setting the second input of `IShuffleLayer` to *T3* would cause all three tensors to become shape tensors, because `IShuffleLayer` requires that its second optional input be a shape tensor, and if the output of `IElementWiseLayer` is a shape tensor, its inputs are too.

7.6. Shape Tensor I/O (Advanced)

Sometimes the need arises to do shape tensor I/O for a network. For example, consider a network consisting solely of an `IShuffleLayer`. TensorRT will infer that the second input is a shape tensor. `ITensor::isShapeTensor` will return true for it. Because it is an input shape tensor, TensorRT will require two things for it.

- At build time: the optimization profile *values* of the shape tensor.
- At run time: the *values* of the shape tensor.

The shape of an input shape tensor is always known at build time. It's the values that need to be described since they can be used to specify the dimensions of execution tensors.

The optimization profile values can be set using `IOptimizationProfile::setShapeValues`. Analogous to how min, max, and optimization dimensions must be supplied for execution tensors with runtime dimensions, min, max and optimization values must be provided for shape tensors at build time.

The corresponding runtime method is `IExecutionContext::setInputShapeBinding`, which sets the values of the shape tensor at runtime.

Because the inference of "execution tensor" vs "shape tensor" is based on ultimate use, TensorRT cannot infer whether a network output is a shape tensor. You must to tell it via the method `INetworkDefinition::markOutputForShapes`.

Besides letting you output shape information for debugging, this feature is useful for composing engines. For example, consider building three engines, one each for subnetworks A, B, C where a connection from A to B, or B to C might involve a shape tensor. Build the networks in reverse order: C, B, and A. After constructing network C, you can use `ITensor::isShapeTensor` to determine if an input is a shape tensor, and use `INetworkDefinition::markOutputForShapes` to mark the corresponding output tensor in network B. Then check which inputs of B are shape tensors and mark the corresponding output tensor in network A.

Shape tensors at network boundaries must have type `Int32`. They cannot have type `Bool`.

7.7. INT8 Calibration With Dynamic Shapes

To run INT8 calibration for a network with dynamic shapes, calibration optimization profile must be set. Calibration is performed using kOPT values of the profile. Calibration input data size must match this profile.

To create a calibration optimization profile, first construct an `IOptimizationProfile` the same way as it is done for a general optimization profile. Then set the profile to the configuration:

C++

```
config->setCalibrationProfile(profile)
```

Python

```
config.set_calibration_profile(profile)
```

The calibration profile must be valid or be `nullptr`. `kMIN` and `kMAX` values will be overwritten by `kOPT`. To check the current calibration profile use `IBuilderConfig::getCalibrationProfile`.

This method returns a pointer to the current calibration profile or nullptr if calibration profile is unset. `getBatchSize()` calibrator method must return `1` when running calibration for a network with dynamic shapes.

Note: If calibration optimization profile is not set, the first network optimization profile will be used as a calibration optimization profile.

Chapter 8. Working With Empty Tensors

NVIDIA® TensorRT™ supports empty tensors. A tensor is an empty tensor if it has one or more dimensions with length zero. Zero-length dimensions usually get no special treatment. If a rule works for a dimension of length `L` for an arbitrary positive value of `L`, it usually works for `L=0` too.

For example, when concatenating two tensors with dimensions `[x,y,z]` and `[x,y,w]` along the last axis, the result has dimensions `[x,y,z+w]`, regardless of whether `x`, `y`, `z`, or `w` is zero.

Implicit broadcast rules remain unchanged since only unit-length dimensions are special for broadcast. For example, given two tensors with dimensions `[1,y,z]` and `[x,1,z]`, their sum computed by `IElementWiseLayer` has dimensions `[x,y,z]`, regardless of whether `x`, `y`, or `z` is zero.

Note that if an engine binding is an empty tensor, at least one byte of memory still needs to be allocated for it.

8.1. `IReduceLayer` And Empty Tensors

If all inputs to a layer are empty, the output is usually empty, but there are exceptions. The exceptions arise from how reduction over an empty set is defined in mathematics: Reduction over an empty set yields the identity element for the operation.

The following table shows cases relevant to TensorRT:

Reduction Operation	kFLOAT & kHALF	kINT32	kINT8
kSUM	0	0	0
kPROD	1	1	1
kMAX	∞	`INT_MAX`	-128
kMIN	-∞	`INT_MIN`	127
kAVG	NaN	0	-128

The average empty set is mathematically ill-defined. The obvious definition (sum of elements)/(number of elements) yields 0/0. It's represented by "Not a Number" (NaN) for floating point. The 0 for kAVG over an empty set of kINT32 has no mathematical justification, and was chosen for compatibility with TensorFlow.

TensorRT usually performs reduction for kINT8 via kFLOAT or kHALF. The kINT8 values show the quantized *representations* of the floating-point values, not their dequantized values.

8.2. `IMatrixMultiplyLayer`, `IFullyConnectedLayer`, And Empty Tensors

Multiplying matrices with dimensions `[m,0]` and `[0,n]` results in a matrix of zeros with dimensions `[m,n]`. It's zeros because each element of the result is the sum over an empty set of products.

`IFullyConnectedLayer` is fundamentally a matrix multiplication, so similar rules apply.

8.3. Plugins And Empty Tensors

Plugins that need to handle empty tensors must be written with `IPluginV2Ext`, `IPluginV2IOExt`, or `IPluginV2DynamicExt`.

WARNING: Empty tensors can have properties not seen for non-empty tensors:

- **a volume of zero**
- **one or more strides equal to zero**

The volume of zero can break kernel launching logic, since a common approach is to set the the number of CUDA blocks proportional to the volume being processed. CUDA reports an error for launching a kernel with zero blocks. Hence plugins should be careful about avoiding such launches.

Strides should be calculated the same as for non-empty tensors. For example, given a tensor with dimensions [N,C,H,W], the stride of the memory representation corresponding to an increment along the C axis is H*W. It doesn't matter if H or W is zero. Though make sure that your code does not divide by a stride or dimension that could be zero. For example, the assertion in the fragment below risks dividing by zero in both divisions:

```
int volume = N*C*H*W;
int cStride = H*W;
...
assert(C == volume/N/cStride);
```

For some plugins, an effective strategy is to make the plugin's method `enqueue` return early if all outputs are empty, and thereby not complicate the rest of the logic with consideration of zero-length dimensions.

8.4. `IRNNLayer` And Empty Tensors

Due to the deprecation of `IRNNLayer` and `IRNNv2Layer`, use a loop to synthesize a recurrent subnetwork as discussed in the Working With Loops section.

- `IRNNv2Layer` works for empty tensors, but is deprecated in TensorRT 7.2.1.
- `IRNNLayer` has been deprecated since TensorRT 4.0 and does not work for empty tensors.

8.5. `IShuffleLayer` And Empty Tensors

By default, `IShuffleLayer` treats a `0` in the reshape dimensions as a special placeholder, and *not meaning zero.* The placeholder means "copy the corresponding input dimension". This default behavior has been kept for compatibility with earlier versions of TensorRT, but is hazardous when working with empty tensors.

If you are reshaping to dimensions that might include a zero-length dimension, disable the placeholder treatment of zero with the `IShuffleLayer::setZeroIsPlaceholder` method.

```
IShuffleLayer* s = ...;
s->setZeroIsPlaceholder(false);
```

For example, consider the following code that intends to reshape a tensor input to dimensions specified by shape tensor `reshapeDims`.

```
IShuffleLayer* s = network.addShuffle(input);
s->setInput(1, reshapeDims);
#if CORRECT
s->setZeroIsPlaceholder(false);
#endif
output = *s->getOutput(0);
```

Suppose at runtime, the input has dimensions `[3,0]` and the second input `reshapeDims` contains `[0,0]`. If the engine was built with `CORRECT==0`, the zeros in `reshapeDims` will be interpreted as placeholders for input dimensions, and the output will have dimensions `[3,0]`, not `[0,0]` as intended. Building the fragment with `CORRECT==1` ensures that the `IShuffleLayer` treats zero as zero. Unless you know that you need the placeholder feature, it is recommended that it be turned off with `setZeroIsPlaceholder(false)`.

Empty tensors also introduce the possibility of a new kind of error when using the `-1` wildcard in reshape dimensions. The wildcard denotes an unknown dimension *x* that TensorRT solves using the equation:

```
x *  (volume of other reshape dimension) = volume(input
    tensor)
```

If the volume of the other reshape dimensions is zero, one of two errors occur:

- The volume of the input tensor is zero. Then `x` is indeterminate.
- The volume of the input tensor is nonzero. Then `x` has no solution.

8.6. `ISliceLayer` And Empty Tensors

The behavior of `ISliceLayer` for empty tensors follows from a strict interpretation of its semantics. Specifically, consider slicing a dimension of length `L` with parameters `start`, `size`, and `stride`.

Constructing the output tensor requires subscripting the half-open interval `[0,L)` with generated indices of the form `start+i*stride` for all `i` such that `0 ≤ i < size`. All the generated indices must be in bounds. However, with `size=0`, no indices are generated, and

thus no bounds checking applies. So for `size=0`, the `start` and `stride` parameters do not matter and can be any values.

Conversely, if `L=0` and `size≠0`, then TensorRT reports an error since the half-open interval `[0,L)` becomes empty and the generated indices are inherently out of bounds.

8.7. `IConvolutionLayer` And Empty Tensors

Convolution with zero input channels (for example, `[n, 0, h, w]`) results in a tensor of zeros with dimensions `[n, k, p, q]`, before adding the bias, because each element of the result is a sum over an empty set of products.

Chapter 9. Working With Loops

NVIDIA® TensorRT™ supports loop-like constructs, which can be useful for recurrent networks. TensorRT loops support scanning over input tensors, recurrent definitions of tensors, and both "scan outputs" and "last value" outputs.

9.1. Defining A Loop

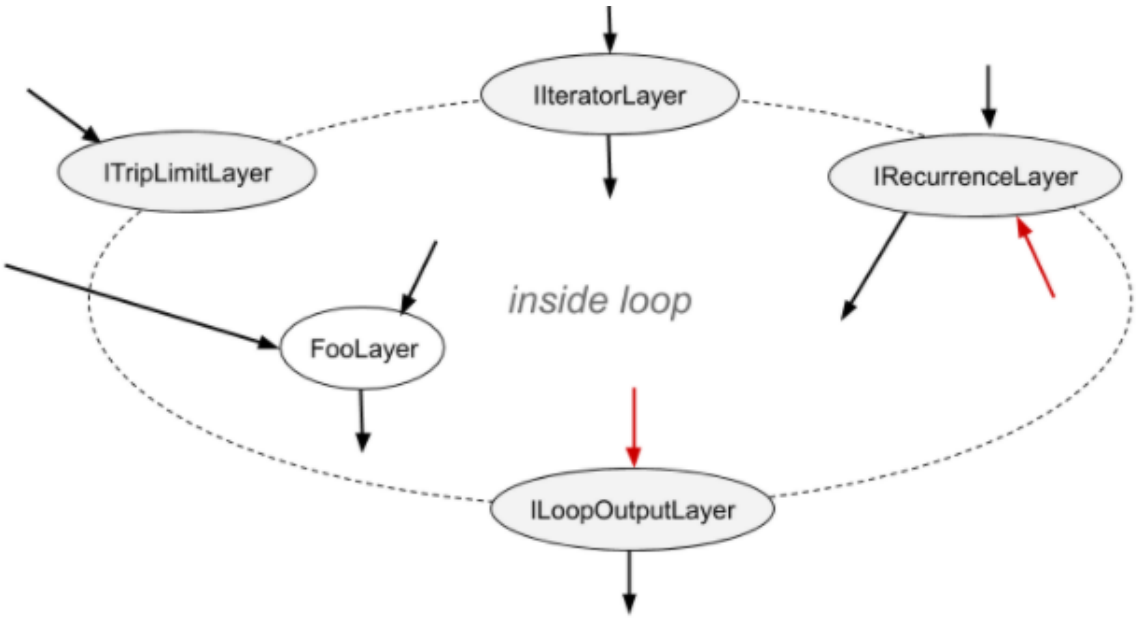
A loop is defined by *loop boundary layers*.

- `ITripLimitLayer` specifies how many times the loop iterates.
- `IIteratorLayer` enables a loop to iterate over a tensor.
- `IRecurrenceLayer` specifies a recurrent definition.
- `ILoopOutputLayer` specifies an output from the loop.

Each of the boundary layers inherit from class `ILoopBoundaryLayer`, which has a method `getLoop()` for getting its associated `ILoop`. The `ILoop` object identifies the loop. All loop boundary layers with the same `ILoop` belong to that loop.

Figure 5 depicts the structure of a loop and dataflow at the boundary. Loop-invariant tensors can be used inside the loop directly, such as shown for FooLayer.

Figure 5. A TensorRT loop is set by loop boundary layers. Dataflow can leave the loop only via `ILoopOutputLayer`. The only backedges allowed are the second input to `IRecurrenceLayer`.



A loop can have multiple `IIteratorLayer`, `IRecurrenceLayer`, and `ILoopOutputLayer`, and at most two `ITripLimitLayers` as explained later. A loop with no `ILoopOutputLayer` has no output and will be optimized away by TensorRT.

The interior of the loop can have the following kinds of layers:

- `IActivationLayer` if the operation is one of:
  - `kRELU`
  - `kSIGMOID`
  - `kTANH`
  - `kELU`
- `IConcatenationLayer`
- `IConstantLayer`
- `IIdentityLayer`
- `IFullyConnectedLayer`
- `IMatrixMultiplyLayer`
- `IElementWiseLayer`
- `IPluginV2Layer`
- `IScaleLayer`
- `ISliceLayer`
- `ISelectLayer`

- `IShuffleLayer`
- `ISoftMaxLayer`
- `IUnaryLayer` if the operation is one of:
  - `kABS`
  - `kCEIL`
  - `kEXP`
  - `kFLOOR`
  - `kLOG`
  - `kNEG`
  - `kNOT`
  - `kRECIP`
  - `kSQRT`

Interior layers are free to use tensors defined inside or outside the loop. The interior may contain other loops (see Nested Loops).

To define a loop, first create an `ILoop` object with method `INetworkDefinition::addLoop`. Then add the boundary and interior layers. The rest of this section describes the features of the boundary layers, using *`loop`* to denote the `ILoop*` returned by `INetworkDefinition::addLoop`.

`ITripLimitLayer` supports both counted loops and while-loops.

- *`loop`*`->addTripLimit(`*`t`*`,TripLimit::kCOUNT)` creates an `ITripLimitLayer` whose input *`t`* is a 0D Int32 tensor that specifies the number of loop iterations.
- *`loop`*`->addTripLimit(`*`t`*`,TripLimit::kWHILE)` creates an `ITripLimitLayer` whose input *`t`* is a 0D Bool tensor that specifies whether an iteration should occur. Typically *t* is either the output of an `IRecurrenceLayer` or a calculation based on said output.

A loop can have at most one of each kind of limit.

`IIteratorLayer` supports iterating forwards or backward over any axis.

- *`loop`*`->addIterator(`*`t`*`)` adds an `IIteratorLayer` that iterates over axis 0 of tensor *`t`*. For example, if the input is the matrix:

  ```
  2 3 5
  4 6 8
  ```

  the output will be the 1D tensor {2, 3, 5} on the first iteration and {4, 6, 8} for the second iteration. It's invalid to iterate beyond the tensor's bounds.
- *`loop`*`->addIterator(`*`t`*`,`*`axis`*`)` is similar, but the layer iterates over the given axis. For example, if axis=1 and the input is a matrix, each iteration delivers a column of the matrix.
- *`loop`*`->addIterator(`*`t`*`,`*`axis`*`,`*`reverse`*`)` is similar, but the layer produces its output in reverse order if *`reverse`*`=true`.

`ILoopOutputLayer` supports three forms of loop output:

- *`loop`*`->addLoopOutput(`*`t`*`,LoopOutput::kLAST_VALUE)` outputs the last value of *`t`*, where `t` must be the output of a `IRecurrenceLayer`.

- *loop*->addLoopOutput(*t*,LoopOutput::kCONCATENATE,*axis*) outputs the concatenation of each iteration's input to *t*. For example, if the input is a 1D tensor, with value {a,b,c} on the first iteration and {d,e,f} on the second iteration, and *axis*=0, the output is the matrix:

  ```
  a b c
  d e f
  ```

  If *axis*=1, the output is:

  ```
  a d
  b e
  c f
  ```

- *loop*->addLoopOutput(*t*,LoopOutput::kREVERSE,*axis*) is similar, but reverses the order.

Both the kCONCATENATE and kREVERSE forms of ILoopOutputLayer require a 2nd input, which is a 0D INT32 shape tensor specifying the length of the new output dimension. When the length is greater than the number of iterations, the extra elements contain arbitrary values. The second input, for example *u*, should be set using ILoopOutputLayer::setInput(1,*u*).

Finally, there is IRecurrenceLayer. Its first input specifies the initial output value, and its second input specifies the next output value. The first input must come from outside the loop; the second input usually comes from inside the loop. For example, the TensorRT analog of this C++ fragment:

```
for (int32_t i = j; ...; i += k) ...
```

could be created by these calls, where j and k are ITensor*.

```
ILoop* loop = n.addLoop();
IRecurrenceLayer* iRec = loop->addRecurrence(j);
ITensor* i = iRec->getOutput(0);
ITensor* iNext = addElementWise(*i, *k,
    ElementWiseOperation::kADD)->getOutput(0);
iRec->setInput(1, *iNext);
```

The second input to IRecurrenceLayer is the only case where TensorRT allows a backedge. If such inputs are removed, the remaining network must be acyclic.

9.2. Formal Semantics

TensorRT has applicative semantics, meaning, there are no visible side effects other than engine inputs and outputs. Because there are no side effects, intuitions about loops from imperative languages do not always work. This section defines a formal semantics for TensorRT's loop constructs.

The formal semantics is based on *lazy sequences* of tensors. Each iteration of a loop corresponds to an element in the sequence. The sequence for a tensor *X* inside the loop is denoted #*X*0, *X*1, *X*2, ...#. Elements of the sequence are evaluated lazily, meaning, as needed.

The output from IIteratorLayer(X) is #X[0], X[1], X[2], ...# where X[i] denotes subscripting on the axis specified for the IIteratorLayer.

The output from IRecurrenceLayer(X,Y)is #X, Y0, Y1, Y2, ...#.

The input and output from an ILoopOutputLayer depend on the kind of LoopOutput.

- `kLAST_VALUE`: Input is a single tensor `X`, and output is X_n for an n-trip loop.
- `kCONCATENATE`: The first input is a tensor `X` and second input is a scalar shape tensor `Y`. The result is the concatenation of `X0, X1, X2, ... Xn-1` with post padding, if necessary, to the length specified by `Y`. It is a runtime error if `Y < n`. `Y` is a build-time constant. Note the inverse relationship with `IIteratorLayer`. `IIteratorLayer` maps a tensor to a sequence of subtensors; `ILoopOutputLayer` with `kCONCATENATE` maps a sequence of subtensors to a tensor.
- `kREVERSE`: Similar to `kCONCATENATE`, but the output is in the reverse direction.

The value of `n` in the definitions for the output of `ILoopOutputLayer` is determined by the `ITripLimitLayer` for the loop:

- For counted loops, it's the iteration count, meaning, the input to the `ITripLimitLayer`.
- For while loops, it's the least n such that X_n is false, where `X` is the sequence for the `ITripLimitLayer`'s input tensor.

The output from a non-loop layer is a sequence-wise application of the layer's function. For example, for a two-input non-loop layer $F(X,Y) = \#f(X_0,Y_0),\ f(X_1,Y_1),\ f(X_2,Y_2)...\#$. If a tensor comes from outside the loop, i.e. is loop-invariant, then the sequence for it is created by replicating the tensor.

9.3. Nested Loops

TensorRT infers nesting of loops from dataflow. For instance if loop B uses values defined *inside* loop A, then B is considered to be nested inside of A.

TensorRT rejects networks where the loops are not cleanly nested, such as if loop A uses values defined in the interior of loop B and vice versa.

9.4. Limitations

A loop that refers to more than one dynamic dimension may take an unexpected amount of memory.

In a loop, memory is allocated as if all dynamic dimensions take on the maximum value of any of those dimensions. For example, if a loop refers to two tensors with dimensions [4,x,y] and [6,y], memory allocation for those tensors will be as if their dimensions were [4,max(x,y),max(x,y)] and [6,max(x,y)].

The input to a `LoopOutputLayer` with `kLAST_VALUE` must be the output from an `IRecurrenceLayer`.

The loop API supports only FP32 and FP16 precision.

9.5. Replacing `IRNNLayer` And `IRNNv2Layer` With Loops

`IRNNLayer` was deprecated in TensorRT 4.0 and will be removed in TensorRT 8.0. `IRNNv2Layer` was deprecated in TensorRT 7.2.1 and will be removed in TensorRT 9.0. Use the loop API to synthesize a recurrent subnetwork. For an example, see `sampleCharRNN`, method `SampleCharRNNLoop::addLSTMCell`. The loop API lets you express general recurrent networks instead of being limited to the prefabricated cells in `IRNNLayer` and `IRNNv2Layer`.

Chapter 10. Working With Quantized Networks

Quantized networks consist of explicit quantize and dequantize nodes in order to convert tensors from FP32 to INT8 and vice-versa.

NVIDIA® TensorRT™ supports quantized ONNX models with QuantizeLinear and DequantizeLinear nodes.

Quantize a tensor `x`

```
y = saturate((x / y_scale) + y_zero_point), where y # [-128, 127]
```

Dequantize a tensor `x`

```
y = (x - x_zero_point) * x_scale
```

TensorRT only supports INT8 activations in the range [-128, 127] and INT8 weights in the range [-127, 127]. Weights have a slightly different range than activations due to legacy reasons. Also, `zero_point` must be 0 as TensorRT only supports symmetric quantization.

Quantized ONNX models can be created using Quantization Aware Training (QAT) where FakeQuantization nodes are inserted to capture dynamic range (TensorFlow) or scale/zero-point (PyTorch).

The following sections explain how to train such a model using TensorFlow, its conversion to canonical ONNX model, and importing to TensorRT.

10.1. Quantization Aware Training (QAT) Using TensorFlow

As TensorRT only supports symmetric quantization for both activations and weights, a training graph must be created using `symmetric=True`.

Tensorflow 1.15 supports Quantization Aware Training (QAT) for creating symmetrically quantized models using tf.contrib.quantize.experimental_create_training_graph API. By default, the TensorFlow training graph would create `per-tensor` weights and activation dynamic range, meaning (min, max). If `per-channel` weights dynamic range needs to be generated we would need to update QAT scripts.

After QAT, we can create a frozen inference graph using the following commands. We are using TensorFlow models repo for training and creating an inference graph.

```
python models/research/slim/export_inference_graph.py \
```

```
    --model_name<model> \
    --output_file=quantized_symm_eval.pb \
    --quantize \
    --symmetric
```

Freeze the graph with checkpoints:

```
python tensorflow/tensorflow/python/tools/freeze_graph.py \
 --input_graph=eval.pb \
 --input_checkpoint=model.ckpt-0000 \
 --input_binary=true \
 --output_graph=quantized_symm_frozen.pb \
 --output_node_names=<OutputNode>
```

10.2. Converting Tensorflow To ONNX Quantized Models

Tensorflow quantized model with `tensorflow::ops::FakeQuantWithMinMaxVars` or `tensorflow::ops::FakeQuantWithMinMaxVarsPerChannel` nodes can be converted to sequence of `QuantizeLinear` and `DequantizeLinear` nodes (QDQ nodes).

Dynamic range with, meaning `[min, max]`, values are converted to `scale` and `zero-point`, where `scale = max(abs(min, max))/127` and `zero_point = 0`.

We use the tf2onnx converter to convert a quantized frozen model to a quantized ONNX model.

```
python -m tf2onnx.convert \
--input quantized_symm_frozen.pb \
--output quantized.onnx \
--inputs <InputNode> \
--outputs <OutputNode> \
--opset 10 \
--fold_const \
--inputs-as-nchw <InputNode>
```

10.3. Importing Quantized ONNX Models

In order to support importing quantized ONNX models, TensorRT needs to create a network in explicit precision mode.

We can create an explicit precision network using the Working With Explicit Precision Using Python or Working With Explicit Precision Using C++.

Chapter 11. 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 NVIDIA® TensorRT™ layers, see DLA Supported Layers. The `trtexec` tool has additional arguments to run networks on DLA, see trtexec.

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
```

11.1. 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 `IBuilderConfig` 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.

`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()`

This function can be used to reset the `IBuilderConfig` 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.

The following API functions in `IBuilder` class can be used to help configure the network for using the DLA:

`getMaxDLABatchSize()`

This function returns the maximum batch size DLA can support.

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

`getNbDLACores()`

This function returns the number of DLA cores available to the user.

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`.

`getDLACore()`

The DLA core the runtime execution is assigned to. The default value is 0.

11.1.1. Example: sampleMNIST With DLA

This section provides details on how to run a TensorRT sample with DLA enabled.

The "Hello World" For TensorRT (sampleMNIST) located in the GitHub repository, the 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);
config->setMaxWorkspaceSize(16_MB);
```

Then, enable `GPUFallback` mode:

```
config->setFlag(BuilderFlag::kGPU_FALLBACK);
config->setFlag(BuilderFlag::kFP16); or config->setFlag(BuilderFlag::kINT8);
```

Enable execution on DLA, where `dlaCore` specifies the DLA core to execute on:

```
config->setDefaultDeviceType(DeviceType::kDLA);
config->setDLACore(dlaCore);
```

With these additional changes, sampleMNIST is ready to execute on DLA. To run sampleMNIST with DLA Core 1, use the following command:

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

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

In this example, let's create a simple network with input, convolution and output.

About this task

Procedure

1. Create the builder and the network:

   ```
   IBuilder* builder = createInferBuilder(gLogger);
   INetworkDefinition* network = builder->createNetworkV2(0U);
   ```

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))
   {
   config->setFlag(BuilderFlag::kFP16); or config->setFlag(BuilderFlag::kINT8);
   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)
   ```

11.2. 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.
- The dimensions used for building must be used at runtime.
- The maximum size of weights supported by DLA is 512 MB.
- A DLA network can only support up to 1 GB of intermediate tensor data. Tensors that are the input and output to the DLA graph are not counted against this limit. TensorRT will reject networks that exceed this limit that are built without GPU fallback enabled.
- DLA supports wildcard dimensions on the leftmost dimension as long as the `min`, `max`, and `opt` values of the profile are equal.
- TensorRT may split a DLA network into multiple sections if any restriction is violated and `GpuFallback` is enabled. Otherwise, TensorRT may emit an error and fallback. For more information, see GPU Fallback Mode.

Note: 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`.

Layer specific restrictions

Convolution and Fully Connected layers

- Only 2 spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Each dimension of the kernel must be in the range `[1, 32]`.
- Padding must be in the range `[0, 31]`.
- Dimensions of padding must be less than the corresponding kernel dimension.
- Dimensions of stride must be in the range `[1, 8]`.
- Number of output maps must be in the range `[1, 8192]`.
- Number of groups must be in the range `[1, 8192]` for operations using the formats `TensorFormat::kLINEAR`, `TensorFormat::kCHW16`, and `TensorFormat::kCHW32`.
- Number of groups must be in the range `[1, 4]` for operations using the formats `TensorFormat::kCHW4`.
- Dilated convolution must be in the range `[1, 32]`.

Deconvolution layer

- Only 2 spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Dimensions of the kernel must be in the range `[1, 32]`, in addition to `1x[64, 96, 128]` and `[64, 96, 128]x1`.
- The stride must be the same in each dimension as the kernel dimensions.
- Padding must be `0`.
- Grouped deconvolution must be `1`.
- Dilated deconvolutions must be `1`.
- Number of input channels must be in the range `[1, 8192]`.
- Number of output channels must be in the range `[1, 8192]`.
- Best performance uses kernels of `[1, 1-32]` and `[1-32, 1]` in addition to `1x[64, 96, 128]` and `[64, 96, 128]x1`.
- Square kernels in the range of `[1, 22]` provide the next performance tier.
- Square kernels in the range of `[22, 32]` provide the lowest performance.

Pooling layer

- Only 2 spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Operations supported: `kMAX`, `kAVERAGE`.
- Dimensions of the window must be in the range `[1, 8]`.
- Dimensions of padding must be in the range `[0, 7]`.
- Dimensions of stride must be in the range `[1, 16]`.
- Exclusive pooling is not supported.
- With INT8 mode, input and output tensor scales must be the same.

Activation layer

- Only 2 spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Functions supported: `ReLU`, `Sigmoid`, `TanH`, `Clipped ReLU` and `Leaky ReLU`.
  - Negative slope is not supported for `ReLU`.
  - `Clipped ReLU` only supports values in the range `[1, 127]`.

ElementWise layer

- Only 2 spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Operations supported: `Sum`, `Sub`, `Product`, `Max`, and `Min`.
- Only `Sum` operation is supported in INT8.

- For INT8 operation, both inputs must use the same scaling factor.

> **Note:** TensorRT concatenates a DLA Scale layer and a DLA ElementWise layer with the operation `Sum` to support the `Sub` operation which is not supported by a single DLA ElementWise layer.

Scale layer

- Only 2 spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Mode supported: `Uniform`, `Per-Channel`, and `ElementWise`.
- Only `scale` and `shift` operations are supported.

LRN (Local Response Normalization) layer

- Allowed window sizes are to `3`, `5`, `7`, or `9`.
- Normalization region supported is: `ACROSS_CHANNELS`.

Concatenation layer

- DLA supports concatenation only along the channel axis.
- Concat must have at least 2 inputs.

Resize layer

- Only `Nearest Neighbor` resize mode is supported.
- The number of scales has to be exactly `4`.
- The first two elements in scales need to be exactly `1` (for unchanged batch and channel dimensions).
- The last two elements in scales, representing the scale values along height and width dimensions, respectively, need to be integer values in the range of `[1, 32]`.

> **Note:** When running INT8 networks on the DLA using TensorRT, operations are recommended to be added to the same subgraph to reduce quantization errors across the subgraph of the network running on the DLA by allowing them to fuse and retain higher precision for intermediate results. Breaking apart the subgraph in order to inspect intermediate results by setting the tensors as Network output tensors, can result in different levels of quantization errors due to these optimizations being disabled.

11.3. 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 the 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.

If a combination of layers in the network cannot run on DLA, all layers in the combination will execute on the GPU.

Chapter 12. Working With Multi-Instance GPU (MIG)

Multi-instance GPU, or MIG, is a new feature in NVIDIA Ampere GPU architecture that enables user-directed partitioning of a single GPU into multiple smaller GPUs. This improves GPU utilization by enabling the GPU to be shared effectively by parallel compute workloads on bare metal, GPU pass through, or on multiple vGPUs.

The physical partitions provide dedicated compute and memory slices with QoS and independent execution of parallel workloads on fractions of the GPU SMs. For TensorRT applications with low GPU SM or memory utilization, partitioning the GPU into smaller instances can produce higher throughput at small or no impact on latency. The optimal partitioning scheme is application specific.

MIG functionality is available in NVIDIA GPU drivers starting with the CUDA 11.0 release. For more information, refer to the NVIDIA Multi-Instance GPU User Guide.

12.1. GPU Partitioning

The NVIDIA Ampere GPU architecture supports eight independent GPU *slices*. Multiple GPU slices can be combined to create GPU *instances*. Each slice is a partition of the GPU memory and SMs resources; other engines (DMAs, NVDEC, etc.) continue to be shared resources.

GPU slices are a combination of GPU memory and SM slices. The NVIDIA Ampere GPU architecture has 7 GPU SM slices and 8 GPU memory slices (i.e. one GPU slice has no SMs).

A GPU instance is built from 1, 2, 4 or 8 naturally aligned GPU memory slices and all the available corresponding GPU SM slices. One exception is the GPU instance with 2 GPU memory slices and 1 GPU SM slice, which is not supported.

For more information, refer to the NVIDIA Multi-Instance GPU User Guide.

12.2. Impact On TensorRT Applications

The Multi-instance GPU (MIG) feature affects many internal components of the compute software stack, however, the user-visible impact is minimal.

TensorRT and CUDA applications treat a compute instance and its parent GPU instance as a single CUDA device (i.e., the MIG configuration is completely transparent to TensorRT APIs and the CUDA programming model).

12.3. Configuring NVIDIA MIG

The partitioning of GPU resources into slices is controlled by the Multi-instance GPU (MIG)-mode setting of the GPU.

Management of GPU instances is performed through the `nvidia-smi` CLI or `nvml` APIs. All the operations require MIG management privileges. For details on the MIG manager, and GPU instance profiles, refer to the GPU Deployment and Management documentation.

Chapter 13. 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 NVIDIA® 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.

13.1. Deploying In The Cloud

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

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

13.2. 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:

Procedure

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.

> **Note:** 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 14. 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 NVIDIA® TensorRT™ engine using the provided parsers. The Python API also supports frameworks that store layer weights in a NumPy compatible format, for example, PyTorch.

14.1. Working With TensorFlow

TensorRT can work with TensorFlow in the following ways.

tf2onnx

This is the recommended method and involves using the tf2onnx converter first to convert the TensorFlow graph to ONNX, and then using the ONNX parser to import the ONNX model into TensorRT. For details on tf2onnx, see the tf2onnx repository**.** For details on using the ONNX parser, see the Importing From ONNX Using Python or Importing An ONNX Model Using The C++ Parser API steps.

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 TF-TRT, see the TF-TRT User Guide. 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 (end_to_end_tensorflow_mnist) sample.

14.1.1. 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!

14.1.2. 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, "")
```

14.1.3. 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
```

14.1.4. 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.

14.1.4.1. 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.

14.1.4.2. Maintaining Model Consistency Between TensorFlow And TensorRT

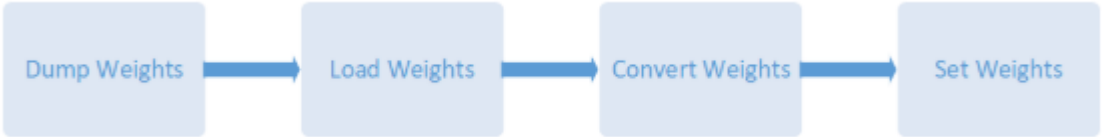
To maintain model consistency, one good way of doing this is to set up unit tests to validate the output from TensorRT by using TensorFlow as the ground truth.

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.

14.1.4.3. Workflow

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

Figure 6. TensorFlow RNN Workflow



14.1.4.4. 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.

14.1.4.5. Loading Dumped Weights

Function `loadWeights()` loads from the dump of the `dumpTFWts.py` script.

It has been provided as an example in Building An RNN Network Layer By Layer (sampleCharRNN) located in the GitHub repository. 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>`.

14.1.4.6. 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.

14.1.4.6.1. 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.

14.1.4.6.2. 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 RuLU/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`)

14.1.4.6.3. 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`.

14.1.4.6.4. TensorFlow Bias Weights 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.

The bias tensor is simply stored as contiguous vectors concatenated in the order specified in TensorFlow Kernel Tensor Storage Format.

14.1.4.6.5. 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.

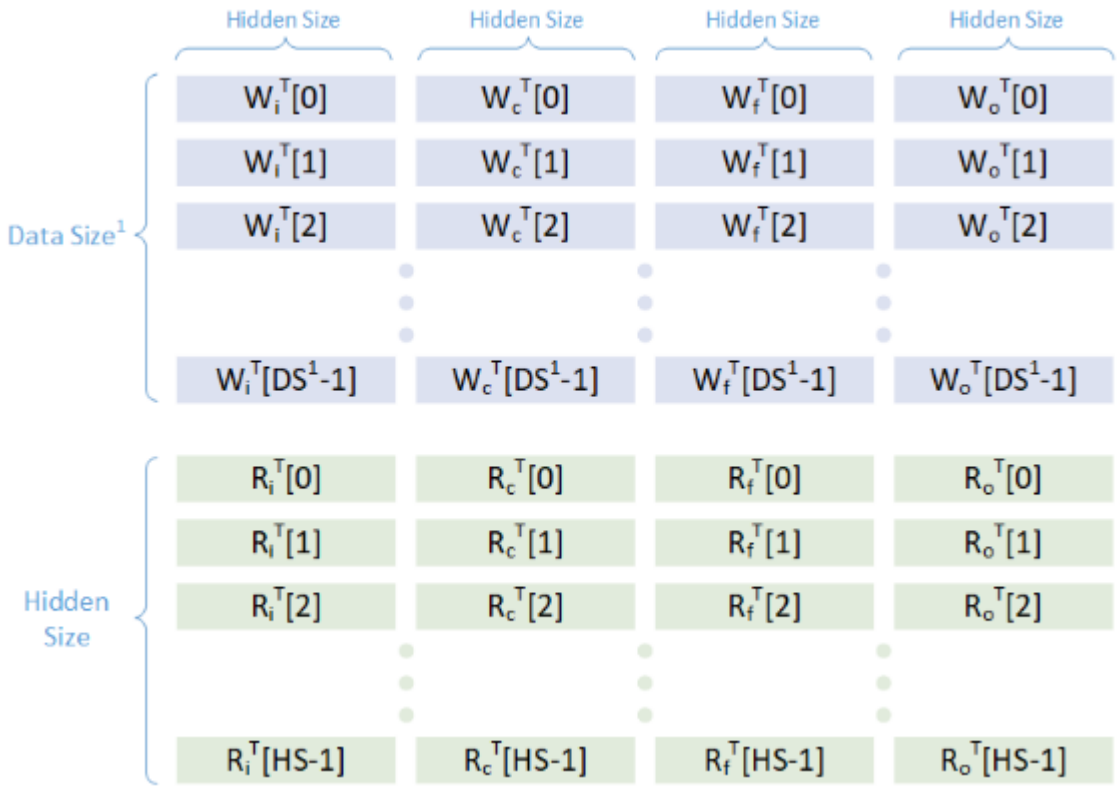
14.1.4.7. `BasicLSTMCell` Example

14.1.4.7.1. BasicLSTMCell Kernel Tensor

To understand the format in which these tensors are being stored, let us consider an example of a `BasicLSTMCell`.

Figure 7 illustrates what the tensor looks like within the TensorFlow checkpoint.

Figure 7. Tensors within a TensorFlow checkpoint



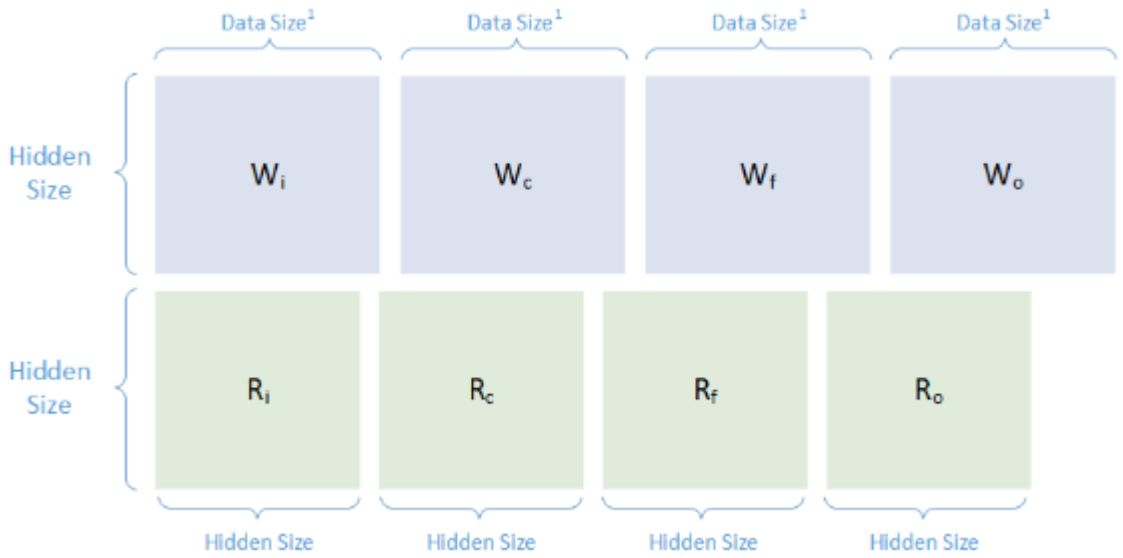
Note: `DS/Data Size` is distinct from `Hidden Size` for the first layer. For all the following sublayers `Data Size` is equal to `Hidden Size`.

In Figure 7, `W` represents the input weights, `R` represents the hidden weights, `DS` represents the data size, and `HS` represents the hidden size.

Note: 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 8.

Figure 8. Converted tensors



Note: `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`.

14.1.4.7.2. BasicLSTMCell Bias Tensor

Figure 9 illustrates the format in which the bias tensor is stored.

Figure 9. 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.

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

14.1.4.8. 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.

Note: 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 the setting process is described in Building An RNN Network Layer By Layer (sampleCharRNN). For more information, consult the code in this sample.

14.1.5. 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 Object Detection With A TensorFlow SSD Network (sampleUffSSD) located in the GitHub repository.

For more details about the `graphsurgeon` API, see the graph surgeon API.

14.2. Working With PyTorch And Other Frameworks

PyTorch models can be exported to ONNX which can then be consumed by TensorRT.

If ONNX export is not possible, exporting to TensorRT is possible by 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 "Hello World" For TensorRT Using PyTorch And Python (network_api_pytorch_mnist) sample.

Chapter 15. Working With DALI

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

NVIDIA® 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.

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

15.1. Benefits Of Integration

The benefits of integrating DALI with TensorRT include the following.

- Running DNN models requires input data pre-processing
- The computational complexity of the I/O pipeline has increased. Hence, the GPU starves for data. DALI helps to accelerate 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 the 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 16. Troubleshooting

The following sections help answer the most commonly asked questions regarding typical use cases with NVIDIA® TensorRT™.

16.1. FAQs

This section is to help troubleshoot the problem and answer our most asked questions.

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, create optimization profiles at the dimensions that are assigned to `OptProfilerSelector::kOPT`.

Q: Why is TensorRT ignoring an attribute I set via `IBuilder` when I build using `IBuilder::buildEngineWithConfig`?

A: If using `IBuilder::buildEngineWithConfig`, configuration attributes should be set on the configuration, not the builder. Here's a list of the `IBuilder` setters for attributes that should instead be set via `IBuilderConfig`:

- `allowGPUFallback`
- `setAverageFindIterations`
- `setDebugSync`
- `setDefaultDeviceType`
- `setDeviceType`
- `setDLACore`
- `setEngineCapability`
- `setFp16Mode`
- `setHalf2Mode`
- `setInt8Calibrator`
- `setInt8Mode`

- `setMaxWorkspaceSize`
- `setMinFindIterations`
- `setRefittable`
- `setStrictTypeConstraints`

Furthermore, these methods are deprecated, so migrating code to use `IBuilderConfig` and `IBuilder::buildEngineWithConfig` is recommended. For more information, see the Deprecated Features section in the *TensorRT Release Notes* for version 7.1.0 EA.

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 `IBuilderConfig::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 `IBuilderConfig::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 an 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.

How do I determine how much device memory TensorRT will need to run my network?

A: TensorRT engines use device memory for three purposes: to hold the weights required by the network, to hold per context persistent state information, 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 per context persistent state information will be allocated using the GPU allocator during the creation of the execution context (even when `createExecutionContextWithoutDeviceMemory()` API is used).The size of persistent memory allocated is reported in the verbose logs generated during the creation of the execution context. 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 at runtime to run the network.

> **Note:** 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: How much memory does the TensorRT builder use to build my network?

`IBuilder` may temporarily use more device memory than what the engine runtime 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 device 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 the overall memory consumption of a network.

> **Note:** 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.

On systems with unified CPU/GPU memory, the `IBuilder` CPU memory consumption may further impact memory requirements. `IBuilder` may be holding in CPU memory not only the original weights provided via the API but a copy of weights at a different precision or calculated from multiple weights from the original network.

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.

Q: How do I implement batch normalization in TensorRT?

A: Batch normalization can be implemented using a sequence of `IElementWiseLayer` in TensorRT. More specifically:

```
adjustedScale = scale / sqrt(variance + epsilon)
```

```
batchNorm = (input + bias - (adjustedScale * mean)) * adjustedScale
```

Q: Why does my network run slower when using DLA compared to without DLA?

A: DLA was designed to maximize energy efficiency. Depending on the features supported by DLA and the features supported by the GPU, either implementation may be more performant. Which implementation to use depends on your latency or throughput requirements and your power budget. Since all DLA engines are independent from the GPU and each other, you could also use both implementations at the same time to further increase the throughput of your network.

Q: Which platforms and/or layers support INT8 quantization?

A: List of Layers supporting INT8 precision:

- `IConcatenationLayer`
- `IDeconvolutionLayer`
- `IElementWiseLayer`
- `IFullyConnectedLayer`
- `IPaddingLayer`
- `IPoolingLayer`
- `IScaleLayer`

For more information, refer to the Support Matrix.

Q: Is INT4 quantization or INT16 quantization supported by TensorRT?

A: Neither INT4 nor INT16 quantization are supported by TensorRT at this time.

Q: When will TensorRT support layer XYZ required by my network in the UFF parser?

A: UFF is deprecated and we will not be doing further development work on UFF; we recommend users switch their workflows to ONNX. The TensorRT ONNX parser is an open source project.

Q: Can I use multiple TensorRT builders to compile on different targets?

A: TensorRT assumes that all resources for the device it is building on are available for optimization purposes. Concurrent use of multiple TensorRT builders (for example, multiple `trtexec` instances) to compile on different targets (DLA0, DLA1 and GPU) may oversubscribe system resources causing undefined behavior (meaning, inefficient plans, builder failure, or system instability).

It is recommended to use `trtexec` with the `--saveEngine` argument to compile for different targets (DLA and GPU) separately and save their plan files. Such plan files can then be reused

for loading (using `trtexec` with the `--loadEngine` argument) and submitting multiple inference jobs on the respective targets (DLA0, DLA1, GPU). This two step process alleviates over-subscription of system resources during the build phase while also allowing execution of the plan file to proceed without interference by the builder.

Q: How can I safely reuse execution context device memory in TensorRT?

A: `IExecutionContext` requires device memory for both persistent and stateless purposes. Users can reuse the stateless portion of the context memory by using a combination of `createExecutionContextWithoutDeviceMemory()` and `setDeviceMemory()`. Any memory that the user provides to TensorRT using `setDeviceMemory()` API can be reused by the application after context execution. However, the persistent memory allocated by TensorRT during context creation cannot be reused between engine executions. If the user uses the `createExecutionContext()` API, there is no simple way to distinguish between the persistent and stateless memory.

16.2. How Do I Report A Bug?

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

Procedure

1. Register for the NVIDIA Developer website.
2. Log in to 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**.

16.3. 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 the 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

Error Message	Description
`Invalid scale mode, nbWeights: <X>`	should always be specified with the implicit batch dimension *not* included in the specification.
`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 the 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 IBuilderConfig::setMaxWorkspaceSize() if using IBuilder::buildEngineWithConfig, or IBuilder::setMaxWorkspaceSize() if using IBuilder::buildCudaEngine.`	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

	Error Message	Description
		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. 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.`	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 the previous layer; emit a warning here.[2]

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

	Error Message	Description
		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. For more information, refer to the Performing Inference In INT8 Using Custom Calibration (sampleINT8) located in the `/opensource/sampleINT8` directory in the GitHub repository to setup calibration correctly.
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

	Error Message	Description
		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`
	`Allocation failed during deserialization of weights.`
	`GPU does not meet the minimum memory requirements to run this engine …`
		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.

FP16 Errors	`Network needs native FP16 and platform does not have native FP16`	This error message can occur if you attempt 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.

16.4. 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, making connections, and to get involved in discussions with customers, developers, and TensorRT engineers.

Appendix A. Appendix

A.1. TensorRT Layers

In NVIDIA® TensorRT™, layers represent distinct flavours of mathematical and/or programmatic operations. The following sections describe every layer that is supported by TensorRT. TensorRT layers and packaged plugins are expected to work with zero workspace size, however, the precision requested may be ignored if there's no implementation that uses zero workspace. In the latter case, the layer will run on FP32 (or TF32 if enabled and available) even if the precision is set to something else.

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 TensorRT Best Practices guide.

For details about the types of precision and features that are supported per layer, see the TensorRT Support Matrix.

A.1.1. `IActivationLayer`

The `IActivationLayer` implements element-wise activation functions.

Layer Description

Apply an activation function on an 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++ class `IActivationLayer` or the Python class `IActivationLayer` 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 **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++ class `IConcatenationLayer` or the Python class `IConcatenationLayer` 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 **d** and weight vector **w**, the constant layer will output a tensor **B** of dimensions **d** with the constant values in **w**. This layer takes no input tensor. The number of elements in the weight vector **w** is equal to the volume of **d**.

Conditions And Limitations

The output can be a tensor of zero to seven dimensions. Boolean weights are not supported.

See the C++ class `IConstantLayer` or the Python class `IConstantLayer` for further details.

A.1.4. IConvolutionLayer

The `IConvolutionLayer` computes a 2D (channel, height, and width) convolution or 3D (channel, depth, height and width) convolution, with or without bias.

> **Note:** The operation that 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: 2D convolution

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:

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

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

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

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

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

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:

$$\mathbf{B}_{i,j,k,l} = \sum (\mathbf{C}_{i,:,k:kk,l:ll} \times \mathbf{K}_{j,:,:,:}) + \mathbf{x}_j$$

where $kk = k+\mathbf{t}_0-1$, and $ll = l+\mathbf{t}_1-1$.

Layer Description: 3D convolution

Compute a cross-correlation with 3D filters on a 5D tensor **A**, of dimensions **a**, to produce a 5D 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:

- $\mathbf{b} = [\mathbf{a}_0\ m\ \mathbf{b}_2\ \mathbf{b}_3\ \mathbf{b}_4]$

- $\mathbf{b}_2 = (\mathbf{a}_2+2\mathbf{p}_0-\mathbf{t}_0)/\mathbf{s}_0+1$
- $\mathbf{b}_3 = (\mathbf{a}_3+2\mathbf{p}_1-\mathbf{t}_1)/\mathbf{s}_1+1$
- $\mathbf{b}_4 = (\mathbf{a}_4+2\mathbf{p}_2-\mathbf{t}_2)/\mathbf{s}_2+1$

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

- **w** is ordered according to shape $[m\ \mathbf{a}_1/g\ \mathbf{r}_0\ \mathbf{r}_1\ \mathbf{r}_2]$
- **x** has length m

Let tensor **K** with dimensions $\mathbf{k} = [m\ \mathbf{a}_1/g\ \mathbf{t}_0\ \mathbf{t}_1\ \mathbf{t}_2]$ be defined as the zero-filled tensor, such that:

- $\mathbf{k}_{i,j,dd,hh,ll} = \mathbf{w}_{i,j,d,h,l}$
- $dd = \{0 \text{ if } d = 0,\ d + \mathbf{d}_0(d-1) \text{ otherwise}\}$, $hh = \{0 \text{ if } h = 0,\ h + \mathbf{d}_1(h-1) \text{ otherwise}\}$, and $ll = \{0 \text{ if } l = 0,\ l + \mathbf{d}_2(l-1) \text{ otherwise}\}$.

and tensor **C** the zero-padded copy of **A** with dimensions $[\mathbf{a}_0\ \mathbf{a}_1\ \mathbf{a}_2+\mathbf{p}_0\ \mathbf{a}_3+\mathbf{p}_1\ \mathbf{a}_4+\mathbf{p}_2]$, then tensor **B** is defined as:

$$\mathbf{B}_{i,j,d,k,l} = \sum (\mathbf{C}_{i,:,d:dd,k:kk,l:ll} \times \mathbf{K}_{j,:,:,:,:}) + \mathbf{x}_j$$

where $dd = d+\mathbf{t}_0-1$, $kk = k+\mathbf{t}_1-1$, and $ll = l+\mathbf{t}_2-1$.

Conditions And Limitations

2D or 3D is determined by the number of input kernel dimensions. For 2D convolution, 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. For 3D convolution, similar to 2D convolution, if input or output has more than 5 dimensions, all dimensions beyond 5 are treated as multipliers on the batch size. 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++ class `IConvolutionLayer` or the Python class `IConvolutionLayer` for further details.

A.1.5. `IDeconvolutionLayer`

The `IDeconvolutionLayer` computes a 2D (channel, height, and width) or 3D (channel, depth, height and width) deconvolution, with or without bias.

> **Note:** This layer actually applies a 2D/3D transposed convolution operator over a 2D/3D input. It is also known as fractionally-strided convolution or transposed convolution.

Layer Description: 2D deconvolution

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:

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

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

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

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

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

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

Layer Description: 3D deconvolution

Compute a cross-correlation with 3D filters on a 5D tensor **A**, of dimensions **a**, to produce a 5D 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:

- $\mathbf{b} = [\mathbf{a}_0\ m\ \mathbf{b}_2\ \mathbf{b}_3]$
- $\mathbf{b}_2 = (\mathbf{a}_2-1)*\mathbf{s}_0 + \mathbf{t}_0 - 2\mathbf{p}_0$
- $\mathbf{b}_3 = (\mathbf{a}_3-1)*\mathbf{s}_1 + \mathbf{t}_1 - 2\mathbf{p}_1$
- $\mathbf{b}_4 = (\mathbf{a}_4-1)*\mathbf{s}_2 + \mathbf{t}_2 - 2\mathbf{p}_2$

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

- **w** is ordered according to shape $[\mathbf{a}_1/g\ m\ \mathbf{r}_0\ \mathbf{r}_1\ \mathbf{r}_2]$
- **x** has length m

Let tensor **K** with dimensions $\mathbf{k} = [m\ \mathbf{b}_1/g\ \mathbf{t}_0\ \mathbf{t}_1\ \mathbf{t}_2]$ be defined as the zero-filled tensor, such that:

- $\mathbf{k}_{i,j,dd,hh,ll} = \mathbf{w}_{i,j,d,h,l}$
- $dd = \{0 \text{ if } d = 0,\ d + \mathbf{d}_0(d-1) \text{ otherwise}\}$, $hh = \{0 \text{ if } h = 0,\ h + \mathbf{d}_1(h-1) \text{ otherwise}\}$, and $ll = \{0 \text{ if } l = 0,\ l + \mathbf{d}_2(l-1) \text{ otherwise}\}$.

and tensor **C** the zero-padded copy of **A** with dimensions $[\mathbf{a}_0\ \mathbf{a}_1\ \mathbf{a}_2+\mathbf{p}_0\ \mathbf{a}_3+\mathbf{p}_1\ \mathbf{a}_4+\mathbf{p}_2]$, then tensor **B** is defined as:

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

where *u* ranges from 0 to $\min(\mathbf{t}_0-1,\ k)$, and *v* ranges from 0 to $\min(\mathbf{t}_1-1,\ l)$, and *w* ranges from 0 to $\min(\mathbf{t}_2-1,\ m)$.

Conditions And Limitations

2D or 3D is determined by the number of input kernel dimensions. For 2D deconvolution, 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. For 3D deconvolution, similar with 2D deconvolution, dimensions beyond 5 are treated as multipliers on the batch size. 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++ class `IDeconvolutionLayer` or the Python class `IDeconvolutionLayer` 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
Floor division : C = floor(A/B)
And : C = A & B
Or : C = A | B
```

```
Xor : C = A xor B
Equal : C = (A == B)
Greater : C = A > B
Less: 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++ class `IElementWiseLayer` or the Python class `IElementWiseLayer` 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. `IFillLayer`

The `IFillLayer` is used to generate an output tensor with the specified mode.

Layer Description

Given an output tensor size, the layer will generate data with the specified mode and fill the tensor. The alpha and beta performs as different parameters for different modes.

The `IFillLayer` supports the following operations:

- `LINSPACE`: Output = `alpha(scalar) + beta(different on each axis) * element_index`
- `RANDOM_UNIFORM`: Output = `Random(min = alpha, max = beta)`

Conditions And Limitations

The layer can only generate 1D tensor if using static tensor size. When using the dynamic tensor size, the alpha and beta's dimensions should match each mode's requirement.

See the C++ class `IFillLayer` or the Python class `IFillLayer` for further details.

A.1.8. `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 ... a_{n-1}], it is first reshaped into a tensor **A'** of dimensions **a'**=[a_0 ... 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, configured 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 ... a_{n-4} k].

Finally, **B'** is reshaped again into the output tensor **B** of dimensions **b**=[a_0 ... 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 ... a_{n-1}], the output tensor **B** will have dimensions **b**=[a_0 ... a_{n-4} k 1 1].

Conditions And Limitations

A must have three dimensions or more.

See the C++ class `IFullyConnectedLayer` or the Python class `IFullyConnectedLayer` for further details.

A.1.9. `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 $\mathbf{c}_k=\{\mathbf{a}_k \text{ if } k<x, \text{ and } \mathbf{a}_{k+1} \text{ if } k>x\}$ and **c** has length equal to one less than the length of **a**. In this case, $\mathbf{C}_i=\mathbf{A}_j$ where $\mathbf{j}_k=\{b \text{ if } k=x, \mathbf{i}_k \text{ if } k<x, \text{ and } \mathbf{i}_{k-1} \text{ if } k>x\}$.

If **B** is a tensor of dimensions **b** (with length b), then $\mathbf{c}_k=\{\mathbf{a}_k \text{ if } k<x, \mathbf{b}_{k-x} \text{ if } k\geq x \text{ and } k<x+b, \text{ and } \mathbf{a}_{k-b+1} \text{ otherwise}\}$. In this case, $\mathbf{C}_i=\mathbf{A}_j$ where $\mathbf{j}_k=\{\mathbf{B}_{X(\mathbf{i})} \text{ if } k=x, \mathbf{i}_k \text{ if } k<x, \text{ and } \mathbf{i}_{k-b} \text{ if } k>x\}$ and $X(\mathbf{i})=\mathbf{i}_{x,..,x+b-1}$.

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++ class `IGatherLayer` or the Python class `IGatherLayer` for further details.

A.1.10. `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++ class `IIdentityLayer` or the Python class `IIdentityLayer` for further details.

A.1.11. `IIteratorLayer`

The `IIteratorLayer` enables a loop to iterate over a tensor. A loop is defined by *loop boundary layers*.

For more information about the `IIteratorLayer`, including how loops work and its limitations, see Working With Loops.

See the C++ class `IIteratorLayer` or the Python class `IIteratorLayer` for further details.

A.1.12. `ILoopBoundaryLayer`

Class `ILoopBoundaryLayer`, derived from class `ILayer`, is the base class for the loop-related layers, specifically `ITripLimitLayer`, `ILoopIteratorLayer`, `IRecurrenceLayer`, and

`ILoopOutputLayer`. Class `ILoopBoundaryLayer` defines a virtual method `getLoop()` that returns a pointer to the associated `ILoop`.

For more information about the `ILoopBoundaryLayer`, including how loops work and its limitations, see Working With Loops.

See the C++ class `ILoopBoundaryLayer` or the Python class `ILoopBoundaryLayer` for further details.

A.1.13. `ILoopOutputLayer`

The `ILoopOutputLayer` specifies an output from the loop. A loop is defined by *loop boundary layers*.

For more information about the `ILoopOutputLayer`, including how loops work and its limitations, see Working With Loops.

See the C++ class `ILoopOutputLayer` or the Python class `ILoopOutputLayer` for further details.

A.1.14. `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++ class `IIdentityLayer` or the Python class `IIdentityLayer` for further details.

A.1.15. `ILRNLayer`

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

Layer Description

Given an input **A**, the LRN layer performs a cross-channel LRN 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 Cchannels, 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 #[-1×10^{20}, 1×10^{20}]
- β #[0.01, 1×10^{5}]
- k #[1×10^{-5}, 1×10^{10}]

See the C++ class `ILRNLayer` or the Python class `ILRNLayer` for further details.

A.1.16. `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 $\mathbf{c}_i=\{max(\mathbf{a}_i, \mathbf{b}_i)$ if $i<n-2$, $\mathbf{a}^I_i$ if $i=n-2$, and $\mathbf{b}^I_i$ 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 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++ class `IMatrixMultiplyLayer` or the Python class `IMatrixMultiplyLayer` for further details.

A.1.16.1. 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.1.17. `IParametricReluLayer`

The `IParametricReluLayer` represents a parametric ReLU operation, meaning, a leaky ReLU where the slopes for $x < 0$ can be different for each element.

Layer Description

Users provide a data tensor **X** and a slopes tensor **S**. At each element, the layer computes $y = x$ if $x \geq 0$ and $y = x \cdot s$ if $x < 0$. The slopes tensor may be broadcast to the size of the data tensor and vice versa.

Conditions And Limitations

Parametric ReLU is not supported in many fusions, therefore the performance may be worse than with standard ReLU.

See the C++ class `IParametricReluLayer` or the Python class `IParametricReluLayer` for further details.

A.1.18. `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 $\mathbf{b}_i=\{\mathbf{x}_0+\mathbf{a}_{n-2}+\mathbf{y}_0 \text{ if } i=n-2;\ \mathbf{x}_1+\mathbf{a}_{n-1}+\mathbf{y}_1 \text{ if } i=n-1; \text{ and } \mathbf{a}_i \text{ otherwise}\}$. Accordingly, the values of $\mathbf{B}_w$ are zeros if $\mathbf{w}_{n-2}<\mathbf{x}_0$ or $\mathbf{x}_0+a_{n-2} \leq \mathbf{w}_{n-2}$ or $\mathbf{w}_{n-1}<\mathbf{x}_1$ or $\mathbf{x}_1+\mathbf{a}_{n-2} \leq \mathbf{w}_{n-1}$. Otherwise, $\mathbf{B}_w=\mathbf{A}_z$ where $\mathbf{z}_{n-2}=\mathbf{w}_{n-2}+\mathbf{x}_0$, $\mathbf{z}_{n-1}=\mathbf{w}_{n-1}+\mathbf{x}_1$, and $\mathbf{z}_i=\mathbf{w}_i$ 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++ class `IPaddingLayer` or the Python class `IPaddingLayer` for further details.

A.1.19. `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++ class `IPluginLayer` or the Python class `IPluginLayer` for further details.

A.1.20. `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++ class `IPluginV2Layer` or the Python class `IPluginV2Layer` for further details.

A.1.21. `IPoolingLayer`

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

Layer Description: 2D pooling

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:

- $\mathbf{b} = [\mathbf{a}_0\ \mathbf{a}_1 \ldots \mathbf{a}_{n-3}\ \mathbf{b}_{n-2}\ \mathbf{b}_{n-1}]$
- $\mathbf{b}_{n-2} = (\mathbf{a}_{n-2}+2\mathbf{p}_0-\mathbf{r}_0)/\mathbf{s}_0+1$
- $\mathbf{b}_{n-1} = (\mathbf{a}_{n-1}+2\mathbf{p}_1-\mathbf{r}_1)/\mathbf{s}_1+1$

Let tensor **C** be the zero-padded copy of **A** with dimensions $[\mathbf{a}_0\ \mathbf{a}_1 \ldots\ \mathbf{a}_{n-2}+2\mathbf{p}_0\ \mathbf{a}_{n-1}+2\mathbf{p}_1]$ then, $\mathbf{B}_{j\ldots\ldots kl} = \mathbf{func}(\mathbf{C}_{j\ldots.\ k:kk\ l:ll})$ where $kk = k+\mathbf{r}_0-1$, and $ll = l+\mathbf{r}_1-1$.

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.

Layer Description: 3D pooling

Compute a pooling with 3D 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:

- $\mathbf{b} = [\mathbf{a}_0\ \mathbf{a}_1 \ldots \mathbf{a}_{n-4}\ \mathbf{b}_{n-3}\ \mathbf{b}_{n-2}\ \mathbf{b}_{n-1}]$
- $\mathbf{b}_{n-3} = (\mathbf{a}_{n-3}+2\mathbf{p}_0-\mathbf{r}_0)/\mathbf{s}_0+1$
- $\mathbf{b}_{n-2} = (\mathbf{a}_{n-2}+2\mathbf{p}_1-\mathbf{r}_1)/\mathbf{s}_1+1$
- $\mathbf{b}_{n-1} = (\mathbf{a}_{n-1}+2\mathbf{p}_2-\mathbf{r}_2)/\mathbf{s}_2+1$

Let tensor **C** be the zero-padded copy of **A** with dimensions $[\mathbf{a}_0\ \mathbf{a}_1 \ldots\ \mathbf{a}_{n-3}+2\mathbf{p}_0\ \mathbf{a}_{n-2}+2\mathbf{p}_1\ \mathbf{a}_{n-1}+2\mathbf{p}_2]$ then, $\mathbf{B}_{j\ldots\ldots klm} = \mathbf{func}(\mathbf{C}_{j\ldots.\ k:kk\ l:ll\ m:mm})$ where $kk = k+\mathbf{r}_0-1$, $ll = l+\mathbf{r}_1-1$, and $mm = m+\mathbf{r}_2-1$.

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

2D or 3D is determined by the number of input kernel dimensions. For 2D pooling, input and output tensors should have 3 or more dimensions. For 3D pooling, input and output tensors should have 4 or more dimensions.

See the C++ class `IPoolingLayer` or the Python class `IPoolingLayer` for further details.

A.1.22. `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++ class `IRaggedSoftMaxLayer` or the Python class `IRaggedSoftMaxLayer` for further details.

A.1.23. `IRecurrenceLayer`

The `IRecurrenceLayer` specifies a recurrent definition. A loop is defined by *loop boundary layers*.

For more information about the `IRecurrenceLayer`, including how loops work and its limitations, see Working With Loops.

See the C++ class `IRecurrenceLayer` or the Python class `IRecurrenceLayer` for further details.

A.1.24. `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 $\mathbf{b}_i$=`{1 if i#`**r**`, and` $\mathbf{a}_i$ `otherwise}`; if the dimensions are not kept, then $\mathbf{b}_{j-m(j)}=\mathbf{a}_j$ 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**, $\mathbf{B}_i$=`op(`$\mathbf{A}_j$`)`, where the sequence of indexes **j** is such that $\mathbf{j}_k$=`{:` `if k#`**r**`, and` $\mathbf{i}_k$ `otherwise}`.

Conditions And Limitations

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

See the C++ class `IReduceLayer` or the Python class `IReduceLayer` for further details.

A.1.25. `IResizeLayer`

The `IResizeLayer` implements the resize operation on an input tensor.

Layer Description

The `IResizeLayer` resizes input tensor `A`, of dimension `a`, to produce an output tensor `B`, of dimensions `b`, using a given resize mode `m`. Output dimension `b` can either be provided directly or can be computed using resize scales `s`. If resize scales `s` are provided, `bi={floor(ai * si)}`.

Interpolation mode such as Nearest and Linear are supported for the resize operation. The nearest mode resizes innermost `d` dimensions of `N-D` tensors, where `d# (0, min(8, N)` and `N > 0`. The linear mode resizes innermost `d` dimensions of `N-D` tensors, where d `# (0, min(3, N)` and `N > 0`. The resize operation can also be configured to align corners while interpolating.

Conditions And Limitations

Either output dimension `b` or resize scales `s` must be known and valid. Number of scales must be equal to the number of input dimensions. Number of output dimensions must be equal to the number of input dimensions.

See the C++ class `IResizeLayer` or the Python class `IResizeLayer` for further details.

A.1.26. `IRNNLayer`

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

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

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++ class `IRNNLayer` or the Python class `IRNNLayer` for further details.

A.1.27. `IRNNv2Layer`

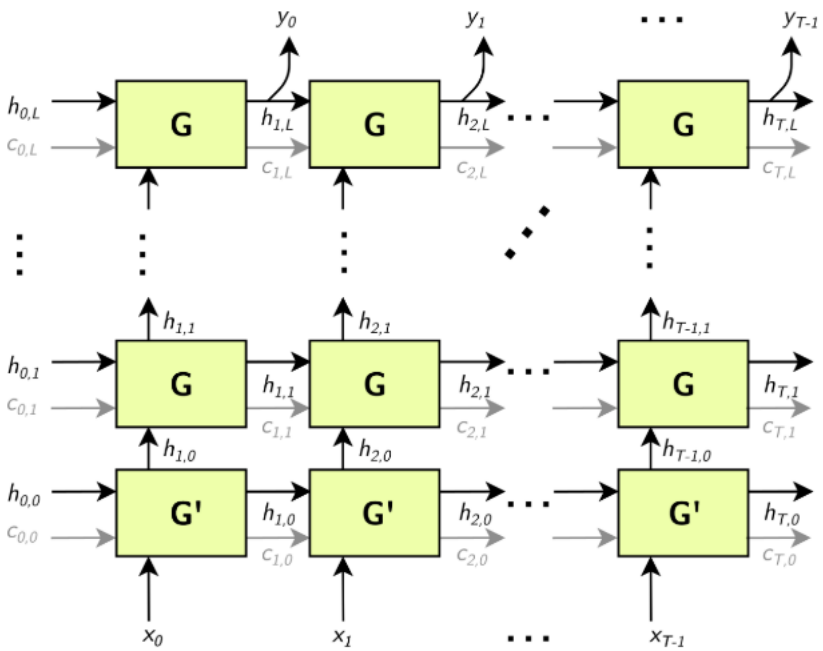
The `IRNNv2Layer` implements recurrent layers such as Recurrent Neural Network (RNN), Gated Recurrent Units (GRU), and Long Short-Term Memory (LSTM). Supported types are RNN, GRU, and LSTM. It performs a recurrent operation, where the operation is defined by one of several well-known recurrent neural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

Layer Description

This layer accepts an input sequence $\mathbf{X}$,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 $\mathbf{Y}$ 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 $\mathbf{G}(\mathbf{x}, \mathbf{h}, \mathbf{c})$. This function takes vector inputs $\mathbf{x}$, $\mathbf{h}$, and $\mathbf{c}$, and produces up to two vector outputs, $\mathbf{h}'$ and $\mathbf{c}'$, representing the hidden and cell state after the cell operation has been performed.

In the default (unidirectional) configuration, the RNNv2 layer applies $\mathbf{G}$as shown in the following diagram:



G' is a variant of **G**, .

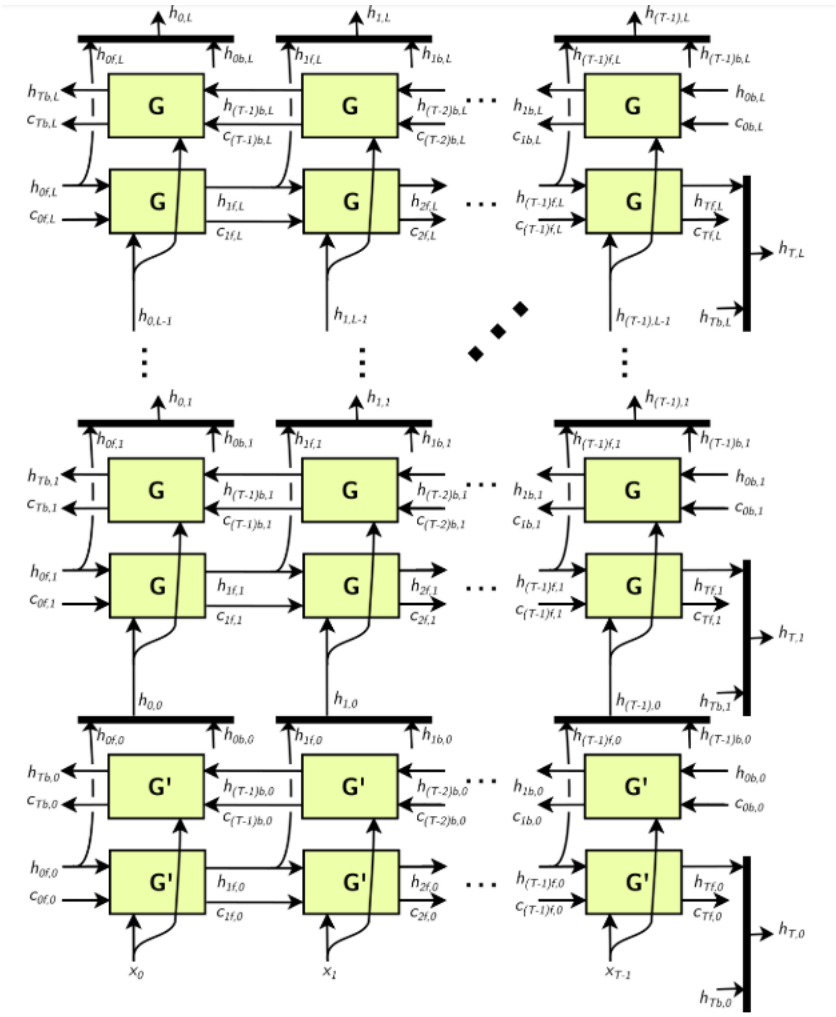
Arrows leading into boxes are function inputs, and arrows leading away from boxes are function outputs. **X** = [x_0, x_1, ..., x_T], **Y** = [y_0, y_1, ..., y_T], H_i= [$h_{i,0}$, $h_{i,1}$, ..., $h_{i,L}$], and C_i= [$c_{i,0}$, $c_{i,1}$, ..., $c_{i,L}$].

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

> **Note:** 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 Tmay 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 $\mathbf{h}_t$ is defined by the concatenation of the forward hidden state $\mathbf{h}_{tf}$ and the backward hidden state $\mathbf{h}_{tb}$:

- $\mathbf{h}_{t,i} = [\ h_{tf,i}\ , h_{tb,i}]$
- $\mathbf{h}_t = [\ \mathbf{h}_{t,0},\ \ \mathbf{h}_{t,1},\ \ldots, \mathbf{h}_{t,L}]$.

Similarly, for the cell state (not shown). Each **ht,i** is used as input to the next sub-layer, as shown above.

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

- **ReLU:** $\mathbf{G}(\mathbf{x}, \mathbf{h}, \mathbf{c}) := \max(\mathbf{W}_i\mathbf{x} + \mathbf{R}_i\mathbf{h} + \mathbf{W}_b + \mathbf{R}_b,\ 0)$ (**c** not used)
- **tanh:** $\mathbf{G}(\mathbf{x}, \mathbf{h}, \mathbf{c}) := \tanh(\mathbf{W}_i\mathbf{x} + \mathbf{R}_i\mathbf{h} + \mathbf{W}_b + \mathbf{R}_b)$ (**c** not used)
- **GRU:**
  - $\mathbf{Z} := \mathrm{sigmoid}(\mathbf{W}_z\mathbf{x} + \mathbf{R}_z\mathbf{h} + \mathbf{W}_{bz} + \mathbf{R}_{bz})$
  - $\mathbf{M} := \mathrm{sigmoid}(\mathbf{W}_r\mathbf{x} + \mathbf{R}_r\mathbf{h} + \mathbf{W}_{br} + \mathbf{R}_{br})$
  - $\mathbf{G}(\mathbf{x}, \mathbf{h}, \mathbf{c}) := \tanh(\mathbf{W}_h\mathbf{x} + \mathbf{M}(\mathbf{h} + \mathbf{R}_{bh}) + \mathbf{W}_{bh})$ (**c** not used)

- **LSTM:**
  - $\mathbf{I} := \text{sigmoid}(\mathbf{W_I x} + \mathbf{R_I h} + \mathbf{W_{bi}} + \mathbf{R_{bi}})$
  - $\mathbf{F} := \text{sigmoid}(\mathbf{W_f x} + \mathbf{R_f h} + \mathbf{W_{bf}} + \mathbf{R_{bf}})$
  - $\mathbf{O} := \text{sigmoid}(\mathbf{W_o x} + \mathbf{R_o h} + \mathbf{W_{bo}} + \mathbf{R_{bo}})$
  - $\mathbf{C} := \tanh(\mathbf{W_C x} + \mathbf{R_C h} + \mathbf{W_{bc}} + \mathbf{R_{bc}})$
  - $\mathbf{C'} := \mathbf{F} \times \mathbf{C}$
  - $\mathbf{H} := \mathbf{O} \times \tanh(\mathbf{C'})$
  - $\mathbf{G(x, h, c)} := \{ \mathbf{H, C'} \}$

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 $\mathbf{W_{bx}}$ and $\mathbf{R_{bx}}$ 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 $\mathbf{G'(x, h, c)}$, 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 $\mathbf{G(x, h, c)}$, 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 ($\mathbf{H_0}$ and $\mathbf{C_0}$) 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 ($\mathbf{H_T}$ and $\mathbf{C_T}$) 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++ class `IRNNv2Layer` or the Python class `IRNNv2Layer` for further details.

A.1.27.1. 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);
```

Note: 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.1.27.2. 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.1.28. `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

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

`ScaleMode::kCHANNEL` channel-wise transformation

$B_I = (\mathbf{A}_I * scale_{c(I)} + shift_{c(I)})^{power_{c(I)}}$

`ScaleMode::kELEMENTWISE` element-wise transformation

$B_I = (\mathbf{A}_I * scale_1 + shift_1)^{power_1}$

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

Conditions And Limitations

A must have at least 3 dimensions in implicit batch mode and at least 4 dimensions in explicit batch mode.

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++ class `IScaleLayer` or the Python class `IScaleLayer` for further details.

A.1.29. `ISelectLayer`

The `ISelectLayer` returns either of the two inputs depending on the condition.

Layer Description

This layer returns the elements chosen from input tensor **B**(`thenInput`) or **C**(`elseInput`) depending on the condition tensor **A**.

Conditions And Limitations

The length of each dimension of the three input tensors **A**, **B and C** must be equal or equal to one. The condition tensor is required to be of boolean type.

`ISelectLayer` supports only FP32 and FP16 precision.

See the C++ class `ISelectLayer` or the Python class `ISelectLayer` for further details.

A.1.30. `IShapeLayer`

The `IShapeLayer` gets the shape of a tensor.

Layer Description

The `IShapeLayer` outputs the dimensions of its input tensor. The output is a 1D tensor of type INT32.

Conditions And Limitations

The input tensor must have at least one dimension. The output tensor is a "shape tensor", which can be used as an input only for layers that handle shape tensors. See Execution Tensors vs. Shape Tensors for more information.

See the C++ class `IShapeLayer` or the Python class `IShapeLayer` for further details.

A.1.31. `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 must 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 reshape dimensions can be specified as build-time constants in the layer or as runtime values by supplying a second input to the layer, which must be a 1D tensor of type INT32.

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++ class `IShuffleLayer` or the Python class `IShuffleLayer` for further details.

A.1.32. `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: $a_i = b_i * s_i + o_i \quad (0 \le i < n)$, 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 element in `B` must not be out-of-bounds.

See the C++ class `ISliceLayer` or the Python class `ISliceLayer` for further details.

A.1.33. `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 $A_{a_0, \ldots, a_{i-1}, :, a_{i+1}, \ldots, a_{n-1}}$ 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 $C_{a_0, \ldots, a_{i-1}, :, a_{i+1}, \ldots, a_{n-1}}$ 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++ class `ISoftMaxLayer` or the Python class `ISoftMaxLayer` for further details.

A.1.34. `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 $v_j=\{k \text{ if } i\neq j, \text{ and } a_i \text{ otherwise}\}$.

The output values are:

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

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++ class `ITopKLayer` or the Python class `ITopKLayer` for further details.

A.1.34.1. TopK Layer Setup

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

> **Note:** 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.1.35. `ITripLimitLayer`

The `ITripLimitLayer` specifies how many times the loop iterates. A loop is defined by *loop boundary layers*.

For more information about the `ITripLimitLayer`, including how loops work and its limitations, see Working With Loops.

See the C++ class `ITripLayer` or the Python class `ITripLayer` for further details.

A.1.36. `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`
- `sine : B = sin(A)`
- `Cos : B = cos(A)`
- `Tan : B = tan(A)`
- `Tanh : B = tanh(A)`
- `Sinh : B = sinh(A)`
- `Cosh : B = cosh(A)`
- `Asin : B = asin(A)`
- `Acos : B = acos(A)`
- `Atan : B = tan(A)`
- `Asinh : B = asinh(A)`
- `Acosh : B = acosh(A)`
- `Atanh : B = atanh(A)`
- `Ceil : B = ceil(A)`
- `Floor : B = floor(A)`
- `ERF : B = erf(A)`
- `NOT : B = ~A`

Conditions And Limitations

Input and output can be zero to 7 dimensional tensors.

See the C++ class `IUnaryLayer` or the Python class `IUnaryLayer` for further details.

A.2. Data Format Descriptions

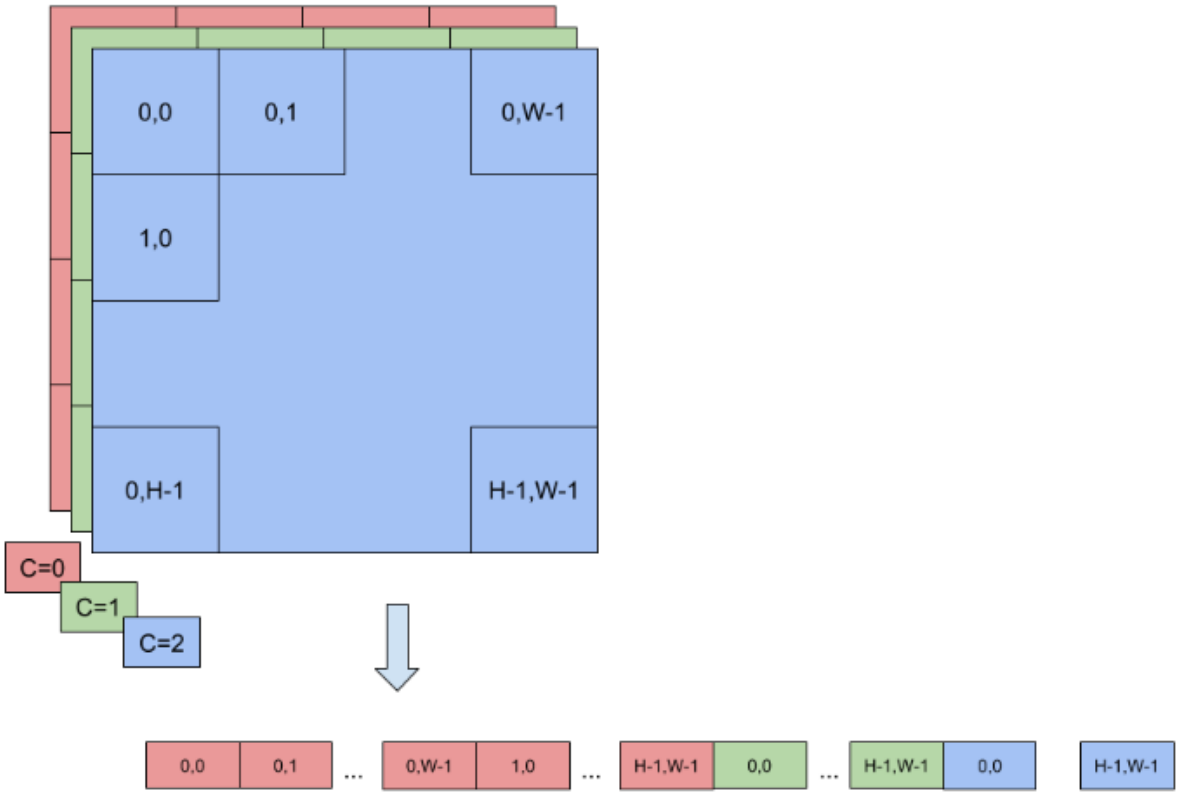
NVIDIA® TensorRT™ supports different data formats. There are two aspects to consider: data type and layout.

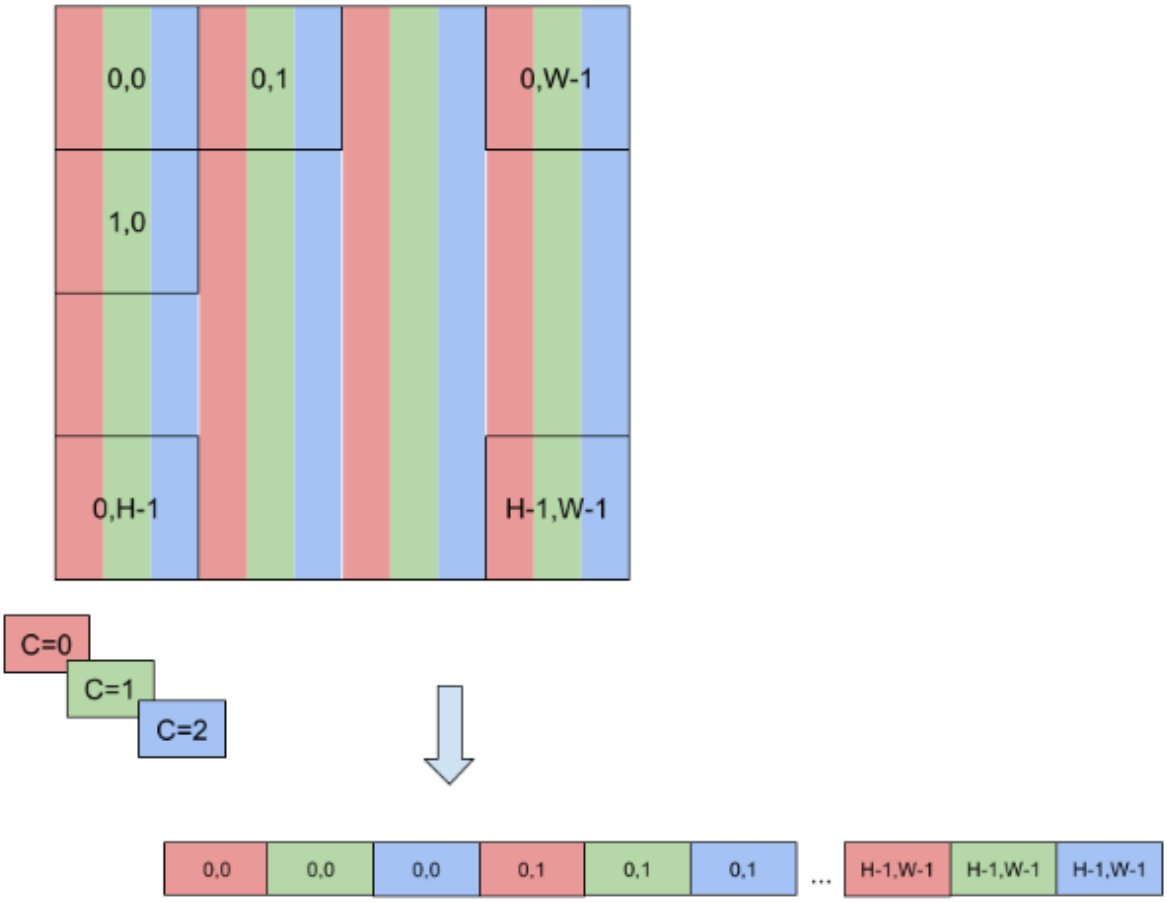
Data type format

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

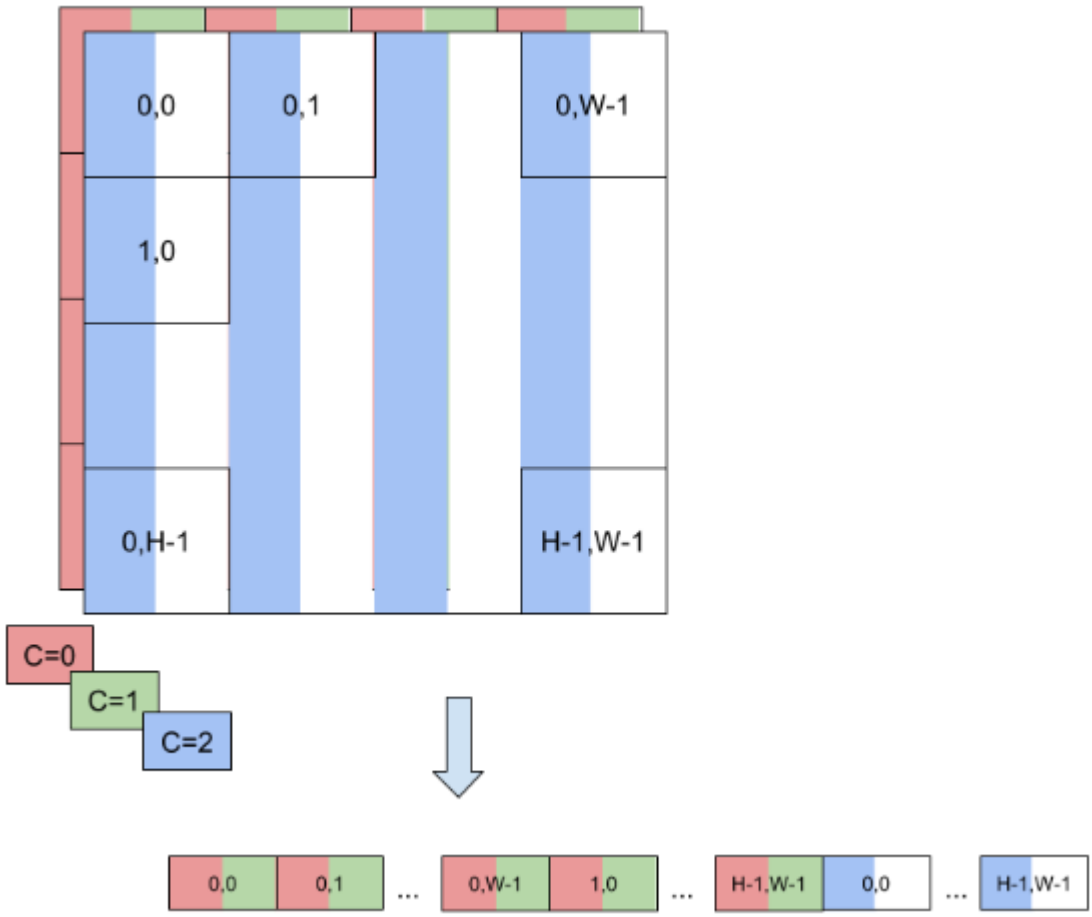
The layout format determines the ordering in which values are stored. Typically, batch dimensions are the leftmost dimensions, and the other dimensions refer to aspects of each data item such as `C` is channel, `H` is height, and `W` is width, in images. Ignoring batch sizes, which are always preceding these, `C`, `H`, and `W` are typically sorted as `CHW` #unique_195/unique_195_Connect_42_fig1 or `HWC` #unique_195/unique_195_Connect_42_fig2.



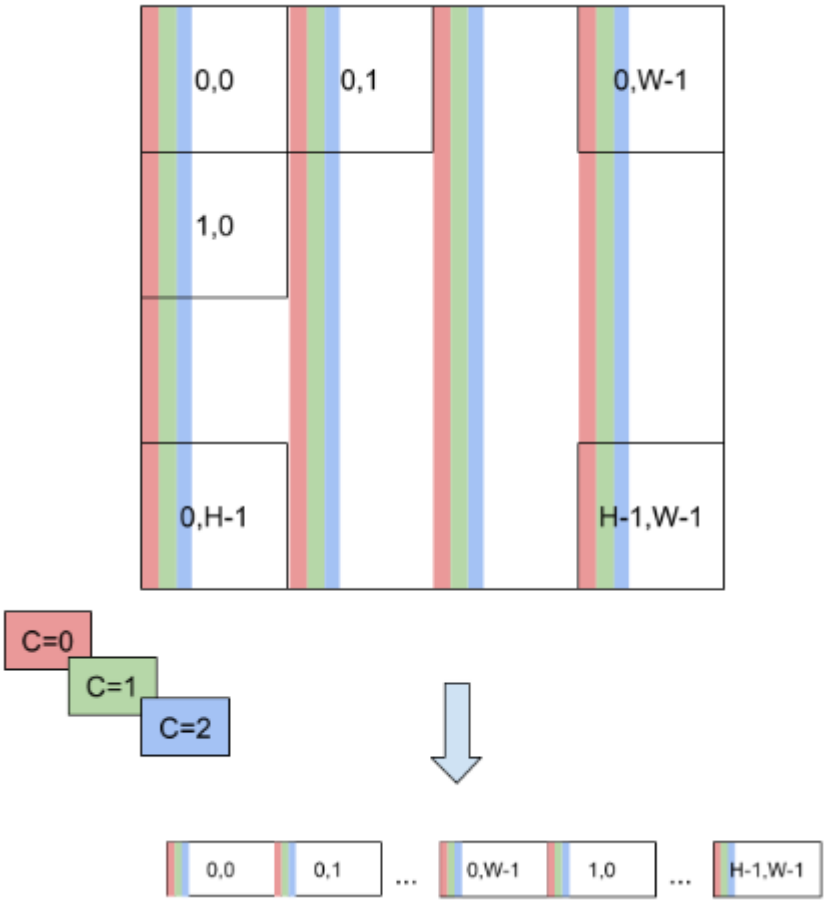


To enable faster computations, more formats are defined to pack together channel values and use reduced precision. For this reason, TensorRT also supports formats `NC/2HW2` and `NHWC8`.

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_195/unique_195_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_195/unique_195_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 Programs

A.3.1. `trtexec`

Included in the `samples` directory is a command line wrapper tool, called `trtexec`. `trtexec` is a tool to quickly utilize TensorRT without having to develop your own application.

The `trtexec` tool has two main purposes:

- It's useful for *benchmarking networks* on random data.
- It's useful for *generating serialized engines* from models.

Benchmarking network: If you have a model saved as a UFF file, ONNX file, or if you have a network description in a Caffe prototxt format, you can use the `trtexec` tool to test the performance of running inference on your network using TensorRT. The `trtexec` tool has many options for specifying inputs and outputs, iterations for performance timing, precision allowed, and other options.

Serialized engine generation: If you generate a saved serialized engine file, you can pull it into another application that runs inference. For example, you can use the TensorRT Laboratory

to run the engine with multiple execution contexts from multiple threads in a fully pipelined asynchronous way to test parallel inference performance. There are some caveats, for example, if you used a Caffe prototxt file and a model is not supplied, random weights are generated. Also, in INT8 mode, random weights are used, meaning `trtexec` does not provide calibration capability.

Refer to GitHub: trtexec/README.md file for detailed information about how to build this tool and examples of its usage.

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.

Copyright 2014, Google Inc. All rights reserved.

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.

Note: 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

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform 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 document. NVIDIA accepts no 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 document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

ARM

ARM, AMBA and ARM Powered are registered trademarks of ARM Limited. Cortex, MPCore and Mali are trademarks of ARM Limited. All other brands or product names are the property of their respective holders. "ARM" is used to represent ARM Holdings plc; its operating company ARM Limited; and the regional subsidiaries ARM Inc.; ARM KK; ARM Korea Limited.; ARM Taiwan Limited; ARM France SAS; ARM Consulting (Shanghai) Co. Ltd.; ARM Germany GmbH; ARM Embedded Technologies Pvt. Ltd.; ARM Norway, AS and ARM Sweden AB.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, CUDA Toolkit, cuDNN, DALI, DIGITS, DGX, DGX-1, DGX-2, DGX Station, DLProf, GPU, JetPack, Jetson, Kepler, Maxwell, NCCL, Nsight Compute, Nsight Systems, NVCaffe, NVIDIA Ampere GPU architecture, NVIDIA Deep Learning SDK, NVIDIA Developer Program, NVIDIA GPU Cloud, NVLink, NVSHMEM, PerfWorks, Pascal, SDK Manager, T4, Tegra, TensorRT, TensorRT Inference Server, Tesla, TF-TRT, Triton Inference Server, Turing, and Volta are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2017-2020 NVIDIA Corporation. All rights reserved.

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com