NVIDIA TensorRT

Developer Guide | NVIDIA Docs

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

NVIDIA® TensorRT™ is an SDK that facilitates high performance machine learning inference. It is designed to work in a complementary fashion with training frameworks such as TensorFlow, PyTorch, and MXNet. It focuses specifically on running an already-trained network quickly and efficiently on NVIDIA hardware.

Refer to the NVIDIA TensorRT Installation Guide for instructions on how to install TensorRT.

The *NVIDIA TensorRT Quick Start Guide* is for users who want to try out TensorRT SDK; specifically, you'll learn how to quickly construct an application to run inference on a TensorRT engine.

1.1. Structure of this Guide

Chapter 1 provides information about how TensorRT is packaged and supported, and how it fits into the developer ecosystem.

Chapter 2 provides a broad overview of TensorRT capabilities.

Chapters 3 and 4 contain introductions to the C++ and Python APIs respectively.

Subsequent chapters provide more detail about advanced features.

The Appendix contains a layer reference and answers to Frequently Asked Questions.

1.2. Samples

The *NVIDIA TensorRT Sample Support Guide* illustrates many of the topics discussed in this guide. Additional samples focusing on embedded applications can be found here.

1.3. Complementary GPU Features

Multi-Instance GPU, or MIG, is a feature of NVIDIA GPUs with NVIDIA Ampere architecture or later architectures that enables user-directed partitioning of a single GPU into multiple smaller GPUs. The physical partitions provide dedicated compute and memory slices with QoS and independent execution of parallel workloads on fractions of the GPU. For TensorRT applications with low GPU utilization, MIG can produce higher throughput at small or no impact on latency. The optimal partitioning scheme is application-specific.

1.4.   Complementary Software

The NVIDIA Triton™ Inference Server is a higher level library providing optimized inference across CPUs and GPUs. It provides capabilities for starting and managing multiple models, and REST and gRPC endpoints for serving inference.

NVIDIA DALI® provides high performance primitives for preprocessing image, audio, and video data. TensorRT inference can be integrated as a custom operator in a DALI pipeline. A working example of TensorRT inference integrated as a part of DALI can be found here.

TensorFlow-TensorRT (TF-TRT) is an integration of TensorRT directly into TensorFlow. It selects subgraphs of TensorFlow graphs to be accelerated by TensorRT, while leaving the rest of the graph to be executed natively by TensorFlow. The result is still a TensorFlow graph that you can execute as usual. For TF-TRT examples, refer to Examples for TensorRT in TensorFlow.

The PyTorch Quantization Toolkit provides facilities for training models at reduced precision, which can then be exported for optimization in TensorRT.

In addition, the PyTorch Automatic SParsity (ASP) tool provides facilities for training models with structured sparsity, which can then be exported and allows TensorRT to utilize the faster sparse tactics on NVIDIA Ampere GPUs.

TensorRT is integrated with NVIDIA's profiling tools, NVIDIA Nsight™ Systems and NVIDIA® Deep Learning Profiler (DLProf).

A restricted subset of TensorRT is certified for use in NVIDIA DRIVE® products. Some APIs are marked for use only in NVIDIA DRIVE and are not supported for general use.

1.5.   ONNX

TensorRT's primary means of importing a trained model from a framework is via the ONNX interchange format. TensorRT ships with an ONNX parser library to assist in importing models. Where possible, the parser is backward compatible up to opset 7; the ONNX Model Opset Version Converter can assist in resolving incompatibilities.

The GitHub version may support later opsets than the version shipped with TensorRT refer to the ONNX-TensorRT operator support matrix for the latest information on the supported opset and operators.

The ONNX operator support list for TensorRT can be found here.

PyTorch natively supports ONNX export. For TensorFlow, the recommended method is tf2onnx.

A good first step after exporting a model to ONNX is to run constant folding using Polygraphy. This can often solve TensorRT conversion issues in the ONNX parser and generally simplify the workflow. For details, refer to this example. In some cases, it may be necessary to modify the ONNX model further, for example, to replace subgraphs with plugins or reimplement unsupported operations in terms of other operations. To make this process easier, you can use ONNX-GraphSurgeon.

1.6.   Code Analysis Tools

For guidance using the valgrind and clang sanitizer tools with TensorRT, refer to the Troubleshooting chapter.

1.7.   API Versioning

TensorRT version number (MAJOR.MINOR.PATCH) follows Semantic Versioning 2.0.0 for its public APIs and library ABIs. Version numbers change as follows:

1. MAJOR version when making incompatible API or ABI changes
2. MINOR version when adding functionality in a backward-compatible manner
3. PATCH version when making backward-compatible bug fixes

Note that semantic versioning does not extend to serialized objects. To reuse plan files, and timing caches, version numbers must match across major, minor, patch, and build versions. Calibration caches can typically be reused within a major version but compatibility is not guaranteed.

1.8.   Deprecation Policy

Deprecation is used to inform developers that some APIs and tools are no longer recommended for use. Beginning with version 8.0, TensorRT has the following deprecation policy:

▶   Deprecation notices are communicated in the release notes. Deprecated API elements are marked with the `TRT_DEPRECATED` macro where possible.

▶   TensorRT provides a 12-month migration period after the deprecation.

▶   APIs and tools continue to work during the migration period.

▶   After the migration period ends, APIs and tools are removed in a manner consistent with semantic versioning.

For any APIs and tools specifically deprecated in TensorRT 7.x, the 12-month migration period starts from the TensorRT 8.0 GA release date.

1.9.   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 getting involved in discussions with customers, developers, and TensorRT engineers.

1.10.   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**.

Chapter 2. TensorRT's Capabilities

This chapter provides an overview of what you can do with TensorRT. It is intended to be useful to all TensorRT users.

2.1. C++ and Python APIs

TensorRT's API has language bindings for both C++ and Python, with nearly identical capabilities. The Python API facilitates interoperability with Python data processing toolkits and libraries like NumPy and SciPy. The C++ API can be more efficient, and may better meet some compliance requirements, for example in automotive applications.

> **Note:** The Python API is not available for all platforms. For more information, refer to the [NVIDIA TensorRT Support Matrix](#).

2.2. The Programming Model

TensorRT operates in two phases. In the first phase, usually performed offline, you provide TensorRT with a model definition, and TensorRT optimizes it for a target GPU. In the second phase, you use the optimized model to run inference.

2.2.1. The Build Phase

The highest-level interface for the build phase of TensorRT is the `Builder` ([C++](#), [Python](#)). The builder is responsible for optimizing a model, and producing an `Engine`.

In order to build an engine, you need to:

- ▶ Create a network definition
- ▶ Specify a configuration for the builder
- ▶ Call the builder to create the engine

The `NetworkDefinition` interface ([C++](#), [Python](#)) is used to define the model. The most common path to transfer a model to TensorRT is to export it from a framework in ONNX format, and use TensorRT's ONNX parser to populate the network definition. However, you can

also construct the definition step by step using TensorRT's *Layer* (C++, Python) and *Tensor* (C ++, Python) interfaces.

Whichever way you choose, you must also define which tensors are the inputs and outputs of the network. Tensors that are not marked as outputs are considered to be transient values that can be optimized away by the builder. Input and output tensors must be named, so that at runtime, TensorRT knows how to bind the input and output buffers to the model.

The *BuilderConfig* interface (C++, Python) is used to specify how TensorRT should optimize the model. Among the configuration options available, you can control TensorRT's ability to reduce the precision of calculations, control the tradeoff between memory and runtime execution speed, and constrain the choice of CUDA® kernels. Since the builder can take minutes or more to run, you can also control how the builder searches for kernels, and cached search results for use in subsequent runs.

Once you have a network definition and a builder configuration, you can call the builder to create the engine. The builder eliminates dead computations, folds constants, and reorders and combines operations to run more efficiently on the GPU. It can optionally reduce the precision of floating-point computations, either by simply running them in 16-bit floating point, or by quantizing floating point values so that calculations can be performed using 8-bit integers. It also times multiple implementations of each layer with varying data formats, then computes an optimal schedule to execute the model, minimizing the combined cost of kernel executions and format transforms.

The builder creates the engine in a serialized form called a *plan*, which can be deserialized immediately, or saved to disk for later use.

> **Note:**
> ▶ Engines created by TensorRT are specific to both the TensorRT version with which they were created and the GPU on which they were created.
> ▶ TensorRT's network definition does not deep-copy parameter arrays (such as the weights for a convolution). Therefore, you must not release the memory for those arrays until the build phase is complete. When importing a network using the ONNX parser, the parser owns the weights, so it must not be destroyed until the build phase is complete.
> ▶ The builder times algorithms to determine the fastest. Running the builder in parallel with other GPU work may perturb the timings, resulting in poor optimization.

2.2.2.   The Runtime Phase

The highest-level interface for the execution phase of TensorRT is the *Runtime* (C++, Python).

When using the runtime, you will typically carry out the following steps:

▶ Deserialize a plan to create an engine

▶ Create an execution context from the engine

Then, repeatedly:

▶ Populate input buffers for inference

▶ Call `enqueue()` or `execute()` on the execution context to run inference

The *Engine* interface (C++, Python) represents an optimized model. You can query an engine for information about the input and output tensors of the network - the expected dimensions, data type, data format, etc.

The *ExecutionContext* interface (C++, Python), created from the engine, is the main interface for invoking inference. The execution context contains all of the state associated with a particular invocation - thus you can have multiple contexts associated with a single engine, and run them in parallel.

When invoking inference, you must set up the input and output buffers in the appropriate locations. Depending on the nature of the data this may be in either CPU or GPU memory. If not obvious based on your model, you can query the engine to determine in which memory space to provide the buffer.

Once the buffers are set up, inference can be invoked synchronously (`execute`) or asynchronously (`enqueue`). In the latter case, the required kernels are enqueued on a CUDA stream, and control is returned to the application as soon as possible. Some networks require multiple control transfers between CPU and GPU, so control may not return immediately. To wait for completion of asynchronous execution, synchronize on the stream using `cudaStreamSynchronize`.

2.3.   Plugins

TensorRT has a *Plugin* interface to allow applications to provide implementations of operations that TensorRT does not support natively. Plugins that are created and registered with TensorRT's `PluginRegistry` can be found by the ONNX parser while translating the network.

TensorRT ships with a library of plugins, and source for many of these and some additional plugins can be found here.

Refer to the Extending TensorRT With Custom Layers chapter for more details.

2.4.   Types and Precision

TensorRT supports computations using FP32, FP16, INT8, Bool, and INT32 data types.

When TensorRT chooses CUDA kernels to implement floating point operations in the network, it defaults to FP32 implementations. There are two ways to configure different levels of precision:

▶ To control precision at the model level, `BuilderFlag` options (C++, Python) can indicate to TensorRT that it may select lower-precision implementations when searching for the fastest (and because lower precision is generally faster, if allowed to, it typically will).

   Therefore, you can easily instruct TensorRT to use FP16 calculations for your entire model. For regularized models whose input dynamic range is approximately one, this typically produces significant speedups with negligible change in accuracy.

▶ For finer-grained control, where a layer must run at higher precision because part of the network is numerically sensitive or requires high dynamic range, arithmetic precision can be specified for that layer.

Refer to the Reduced Precision section for more details.

2.5.   Quantization

TensorRT supports quantized floating point, where floating-point values are linearly compressed and rounded to 8-bit integers. This significantly increases arithmetic throughput while reducing storage requirements and memory bandwidth. When quantizing a floating-point tensor, TensorRT needs to know its dynamic range - that is, what range of values is important to represent - values outside this range are clamped when quantizing.

Dynamic range information can be calculated by the builder (this is called *calibration*) based on representative input data. Or you can perform quantization-aware training in a framework and import the model to TensorRT with the necessary dynamic range information.

Refer to the Working With INT8 chapter for more details.

2.6.   Tensors and Data Formats

When defining a network, TensorRT assumes that tensors are represented by multidimensional C-style arrays. Each layer has a specific interpretation of its inputs: for example, a 2D convolution will assume that the last three dimensions of its input are in CHW format - there is no option to use, for example an WHC format. Refer to the TensorRT Layers chapter for how each layer interprets its inputs.

Note that tensors are limited to at most 2^31-1 elements.

While optimizing the network, TensorRT performs transformations internally (including to HWC, but also more complex formats) to use the fastest possible CUDA kernels. In general, formats are chosen to optimize performance, and applications have no control over the choices. However, the underlying data formats are exposed at I/O boundaries (network input and output, and passing data to and from plugins) to allow applications to minimize unnecessary format transformations.

Refer to the I/O Formats section for more details.

2.7.   Dynamic Shapes

By default, TensorRT optimizes the model based on the input shapes (batch size, image size, etc) at which it was defined. However, the builder can be configured to allow the input dimensions to be adjusted at runtime. In order to enable this, you specify one or more instances of `OptimizationProfile` (C++, Python) in the builder configuration, containing for each input a minimum and maximum shape, along with an optimization point within that range.

TensorRT creates an optimized engine for each profile, choosing CUDA kernels that work for all shapes within the [minimum, maximum] range and are fastest for the optimization point - typically different kernels for each profile. You can then select among profiles at runtime.

Refer to the Working With Dynamic Shapes chapter for more details.

2.8. DLA

TensorRT supports NVIDIA's Deep Learning Accelerator (DLA), a dedicated inference processor present on many NVIDIA SoCs which supports a subset of TensorRT's layers. TensorRT allows you to execute part of the network on the DLA and the rest on GPU; for layers which can be executed on either device, you can select the target device in the builder configuration on a per-layer basis.

Refer to the Working With DLA chapter for more details.

2.9. Updating Weights

When building an engine, you can specify that it may need to later have its weights updated. This can be useful if you are frequently updating the weights of the model without changing the structure, such as in reinforcement learning or when retraining a model while retaining the same structure. Weight updates are performed via the `Refitter` (C++, Python) interface.

Refer to the Refitting An Engine section for more details.

2.10. `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 three main purposes:

▶ *benchmarking networks* on random or user-provided input data.

▶ *generating serialized engines* from models.

▶ *generating a serialized timing cache* from the builder.

Refer to the trtexec section for more details.

2.11. Polygraphy

Polygraphy is a toolkit designed to assist in running and debugging deep learning models in TensorRT and other frameworks. It includes a Python API and a command-line interface (CLI) built using this API.

Among other things, with Polygraphy you can:

▶ Run inference among multiple backends, like TensorRT and ONNX-Runtime, and compare results (for example API,CLI)

▶ Convert models to various formats, for example, TensorRT engines with post-training quantization (for example API,CLI)

▶ View information about various types of models (for example CLI)

▶ Modify ONNX models on the command-line:

    ▶ Extract subgraphs (for example CLI)

    ▶ Simplify and sanitize (for example CLI)

▶ Isolate faulty tactics in TensorRT (for example CLI)

For more details, refer to the Polygraphy repository.

Chapter 3. The C++ API

This chapter illustrates basic usage of the C++ API, assuming you are starting with an ONNX model. sampleOnnxMNIST illustrates this use case in more detail.

The C++ API can be accessed via the header `NvInfer.h`, and is in the `nvinfer1` namespace. For example, a simple application might begin with:

```
#include "NvInfer.h"

using namespace nvinfer1;
```

Interface classes in the TensorRT C++ API begin with the prefix `I`, for example `ILogger`, `IBuilder`, etc.

A CUDA context is automatically created the first time TensorRT makes a call to CUDA, if none exists prior to that point. It is generally preferable to create and configure the CUDA context yourself before the first call to TensorRT.

In order to illustrate object lifetimes, code in this chapter does not use smart pointers; however, their use is recommended with TensorRT interfaces.

3.1. The Build Phase

To create a builder, you first need to instantiate the `ILogger` interface. This example captures all warning messages but ignores informational messages:

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

You can then create an instance of the builder:

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

3.1.1. Creating a Network Definition

Once the builder has been created, the first step in optimizing a model is to create a network definition:

```
uint32_t flag = 1U <<static_cast<uint32_t>
    (NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
```

```
INetworkDefinition* network = builder->createNetworkV2(flag);
```

The kEXPLICIT_BATCH flag is required in order to import models using the ONNX parser. Refer to the [Explicit vs Implicit Batch](#) section for more information.

3.1.2.   Importing a Model using the ONNX Parser

Now, the network definition needs to be populated from the ONNX representation. The ONNX parser API is in the file NvOnnxParser.h, and the parser is in the nvonnxparser C++ namespace.

```
#include "NvOnnxParser.h"

using namespace nvonnxparser;
```

You can create an ONNX parser to populate the network as follows:

```
IParser*  parser = createParser(*network, logger);
```

Then, read the model file and process any errors.

```
parser->parseFromFile(modelFile,
    static_cast<int32_t>(ILogger::Severity::kWARNING));
for (int32_t i = 0; i < parser.getNbErrors(); ++i)
{
std::cout << parser->getError(i)->desc() << std::endl;
}
```

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. Since the network was created via the parser, the parser owns the memory occupied by the weights, and so the parser object should not be deleted until after the builder has run.

3.1.3.   Building an Engine

The next step is to create a build configuration specifying how TensorRT should optimize the model.

```
IBuilderConfig* config = builder->createBuilderConfig();
```

This interface has many properties that you can set in order to control how TensorRT optimizes the network. One important property is the maximum workspace size. Layer implementations often require a temporary workspace, and this parameter limits the maximum size that any layer in the network can use. If insufficient workspace is provided, it is possible that TensorRT will not be able to find an implementation for a layer. By default the workspace is set to the total global memory size of the given device; restrict it when necessary, for example, when multiple engines are to be built on a single device.

```
config->setMemoryPoolLimit(MemoryPoolType::kWORKSPACE, 1U << 20);
```

Once the configuration has been specified, the engine can be built.

```
IHostMemory*  serializedModel = builder->buildSerializedNetwork(*network, *config);
```

Since the serialized engine contains the necessary copies of the weights, the parser, network definition, builder configuration and builder are no longer necessary and may be safely deleted:

```
delete parser;
delete network;
delete config;
```

```
delete builder;
```

The engine can then be saved to disk, and the buffer into which it was serialized can be deleted.

```
delete serializedModel
```

> **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 platform and the TensorRT version).

3.2.   Deserializing a Plan

Assuming you have previously serialized an optimized model and wish to perform inference, you will need to create an instance of the `Runtime` interface. Like the builder, the runtime requires an instance of the logger:

```
IRuntime* runtime = createInferRuntime(logger);
```

Assuming you have read the model from into a buffer, you can then deserialize it to obtain an engine:

```
ICudaEngine* engine =
  runtime->deserializeCudaEngine(modelData, modelSize);
```

3.3.   Performing Inference

The engine holds the optimized model, but to perform inference we will need to manage additional state for intermediate activations. This is done via the `ExecutionContext` interface:

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

An engine can have multiple execution contexts, allowing one set of weights to be used for multiple overlapping inference tasks. (A current exception to this is when using dynamic shapes, when each optimization profile can only have one execution context.)

To perform inference, you must pass TensorRT buffers for input and output, which TensorRT requires you to specify in an array of pointers. You can query the engine using the names you provided for input and output tensors to find the right positions in the array:

```
int32_t inputIndex = engine->getBindingIndex(INPUT_NAME);
int32_t outputIndex = engine->getBindingIndex(OUTPUT_NAME);
```

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;
```

You can then call TensorRT's enqueue method to start inference asynchronously using a CUDA stream:

```
context->enqueueV2(buffers, stream, nullptr);
```

It is common to enqueue `cudaMemcpyAsync()` 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 that is signaled when the input buffers have been consumed, and their memory can 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.

If you prefer synchronous inference, use the `executeV2` method instead of `enqueueV2`.

Chapter 4. The Python API

This chapter illustrates basic usage of the Python API, assuming you are starting with an ONNX model. The onnx_resnet50.py sample illustrates this use case in more detail.

The Python API can be accessed via the `tensorrt` module:

```
import tensorrt as trt
```

4.1. The Build Phase

To create a builder, you need to first create a logger. The Python bindings include a simple logger implementation that logs all messages above a certain severity to `stdout`.

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

Alternatively, it is possible to define your own implementation of the logger by deriving from the `ILogger` class:

```
class MyLogger(trt.ILogger):
    def __init__(self):
        trt.ILogger.__init__(self)

    def log(self, severity, msg):
        pass # Your custom logging implementation here

logger = MyLogger()
```

You can then create a builder:

```
builder = trt.Builder(logger)
```

4.1.1. Creating a Network Definition in Python

Once the builder has been created, the first step in optimizing a model is to create a network definition:

```
network = builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
```

The `EXPLICIT_BATCH` flag is required in order to import models using the ONNX parser. Refer to the Explicit vs Implicit Batch section for more information.

4.1.2. Importing a Model using the ONNX Parser

Now, the network definition needs to be populated from the ONNX representation. You can create an ONNX parser to populate the network as follows:

```
parser = trt.OnnxParser(network, logger)
```

Then, read the model file and process any errors:

```
success = parser.parse_from_file(model_path)
for idx in range(parser.num_errors):
    print(parser.get_error(idx))

if not success:
    pass # Error handling code here
```

4.1.3. Building an Engine

The next step is to create a build configuration specifying how TensorRT should optimize the model:

```
config = builder.create_builder_config()
```

This interface has many properties that you can set in order to control how TensorRT optimizes the network. One important property is the maximum workspace size. Layer implementations often require a temporary workspace, and this parameter limits the maximum size that any layer in the network can use. If insufficient workspace is provided, it is possible that TensorRT will not be able to find an implementation for a layer:

```
config.set_memory_pool_limit(trt.MemoryPoolType.WORKSPACE, 1 << 20) # 1 MiB
```

Once the configuration has been specified, the engine can be built and serialized with:

```
serialized_engine = builder.build_serialized_network(network, config)
```

It may be useful to save the engine to a file for future use. You can do that like so:

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

> **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 platform and the TensorRT version).

4.2. Deserializing a Plan

To perform inference, you will first need to deserialize the engine using the `Runtime` interface. Like the builder, the runtime requires an instance of the logger.

```
runtime = trt.Runtime(logger)
```

You can then deserialize the engine from a memory buffer:

```
engine = runtime.deserialize_cuda_engine(serialized_engine)
```

If you need to first load the engine from a file, run:

```
with open("sample.engine", "rb") as f:
    serialized_engine = f.read()
```

4.3. Performing Inference

The engine holds the optimized model, but to perform inference requires additional state for intermediate activations. This is done via the `IExecutionContext` interface:

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

An engine can have multiple execution contexts, allowing one set of weights to be used for multiple overlapping inference tasks. (A current exception to this is when using dynamic shapes, when each optimization profile can only have one execution context.)

To perform inference, you must pass TensorRT buffers for inputs and outputs, which TensorRT requires you to specify in a list of GPU pointers. You can query the engine using the names you provided for input and output tensors to find the right positions in the array:

```
input_idx = engine[input_name]
output_idx = engine[output_name]
```

Using these indices, set up GPU buffers for each input and output. Several Python packages allow you to allocate memory on the GPU, including, but not limited to, PyTorch, the Polygraphy CUDA wrapper, and PyCUDA.

Then, create a list of GPU pointers. For example, for PyTorch CUDA tensors, you can access the GPU pointer using the `data_ptr()` method; for Polygraphy `DeviceArray`, use the `ptr` attribute:

```
buffers = [None] * 2 # Assuming 1 input and 1 output
buffers[input_idx] = input_ptr
buffers[output_idx] = output_ptr
```

After populating the input buffer, you can call TensorRT's `execute_async` method to start inference asynchronously using a CUDA stream.

First, create the CUDA stream. If you already have a CUDA stream, you can use a pointer to the existing stream. For example, for PyTorch CUDA streams, i.e. `torch.cuda.Stream()`, you can access the pointer using the `cuda_stream` property; for Polygraphy CUDA streams, use the `ptr` attribute.

Next, start inference:

```
context.execute_async_v2(buffers, stream_ptr)
```

It is common to enqueue asynchronous `memcpy()` before and after the kernels to move data from the GPU if it is not already there.

To determine when the kernel (and possibly `memcpy()`) are complete, use the standard CUDA synchronization mechanisms such as events or waiting on the stream. For example, with Polygraphy, use:

```
stream.synchronize()
```

If you prefer synchronous inference, use the `execute_v2` method instead of `execute_async_v2`.

Chapter 5.   How TensorRT Works

This chapter provides more detail on how TensorRT works.

5.1.   Object Lifetimes

TensorRT's API is class-based, with some classes acting as factories for other classes. For objects owned by the user, the lifetime of a factory object must span the lifetime of objects it creates. For example, the `NetworkDefinition` and `BuilderConfig` classes are created from the builder class, and objects of those classes should be destroyed before the builder factory object.

An important exception to this rule is creating an engine from a builder. Once you have created an engine, you may destroy the builder, network, parser and build config and continue using the engine.

5.2.   Error Handling and Logging

When creating TensorRT top-level interfaces (builder, runtime or refitter), you must provide an implementation of the *Logger* ([C++](#), [Python](#)) interface. The logger is used for diagnostics and informational messages; its verbosity level is configurable. Since the logger may be used to pass back information at any point in the lifetime of TensorRT, its lifetime must span any use of that interface in your application. The implementation must also be thread-safe, since TensorRT may use worker threads internally.

An API call to an object will use the logger associated with the corresponding top-level interface. For example, in a call to `ExecutionContext::enqueue()`, the execution context was created from an engine, which was created from a runtime, so TensorRT will use the logger associated with that runtime.

The primary method of error handling is the *ErrorRecorde* ([C++](#), [Python](#)) interface. You can implement this interface, and attach it to an API object to receive errors associated with that object. The recorder for an object will also be passed to any others it creates - for example, if you attach an error recorder to an engine, and create an execution context from that engine, it will use the same recorder. If you then attach a new error recorder to the execution context, it will receive only errors coming from that context. If an error is generated but no error recorder is found, it will be emitted via the associated logger.

Note that CUDA errors are generally asynchronous - so when performing multiple inferences or other streams of CUDA work asynchronously in a single CUDA context, an asynchronous GPU error may be observed in a different execution context than the one which generated it.

5.3. Memory

TensorRT uses considerable amounts of device memory, i.e. memory directly accessible by the GPU, as opposed to the host memory attached to the CPU). Since device memory is often a constrained resource, it's important to understand how TensorRT uses it.

5.3.1. The Build Phase

During build, TensorRT allocates device memory for timing layer implementations. Some implementations can consume a lot of temporary memory, especially with large tensors. You can control the maximum amount of temporary memory through the builder's `maxWorkspace` attribute. This defaults to the full size of device global memory but can be restricted when necessary. If the builder finds applicable kernels that could not be run because of insufficient workspace, it will emit a logging message indicating this.

Even with relatively little workspace however, timing requires creating buffers for input, output, and weights. TensorRT is robust against the operating system returning out-of-memory for such allocations, but on some platforms the OS may successfully provide memory, and subsequently the out-of-memory killer process observes that the system is low on memory, and kills TensorRT. If this happens free up as much system memory as possible before retrying.

During the build phase, there will typically be at least two copies of the weights in host memory: those from the original network, and those included as part of the engine as it is built. In addition, when TensorRT combines weights (for example convolution with batch normalization) additional temporary weight tensors will be created.

5.3.2. The Runtime Phase

At runtime, TensorRT uses relatively little host memory, but can use considerable amounts of device memory.

An engine, on deserialization, allocates device memory to store the model weights. Since the serialized engine is almost all weights, its size is a good approximation to the amount of device memory the weights require.

An `ExecutionContext` uses two kinds of device memory:

▶ Persistent memory required by some layer implementations - for example, some convolution implementations use edge masks, and this state cannot be shared between contexts as weights are, because its size depends on the layer input shape, which may vary across contexts. This memory is allocated on creation of the execution context, and lasts for its lifetime.

▶ Scratch memory, used to hold intermediate results while processing the network. This memory is used for intermediate activation tensors. It is also used for temporary

storage required by layer implementations, the bound for which is controlled by `IBuilderConfig::setMaxWorkspaceSize()`.

You may optionally create an execution context without scratch memory via `ICudaEngine::createExecutionContextWithoutDeviceMemory()` and provide that memory yourself for the duration of network execution. This allows you to share it between multiple contexts that are not running concurrently, or for other uses while inference is not running. The amount of scratch memory required is returned by `ICudaEngine::getDeviceMemorySize()`.

Information about the amount of persistent memory and scratch memory used by the execution context is emitted by the builder when building the network, at severity `kINFO`. Examining the log, the messages look similar to the following:

```
[08/12/2021-17:39:11] [I] [TRT] Total Host Persistent Memory: 106528
[08/12/2021-17:39:11] [I] [TRT] Total Device Persistent Memory: 29785600
[08/12/2021-17:39:11] [I] [TRT] Total Scratch Memory: 9970688
```

By default, TensorRT allocates device memory directly from CUDA. However, you can attach an implementation of TensorRT's `IGpuAllocator` (C++, Python) interface to the builder or runtime and manage device memory yourself. 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.

TensorRT's dependencies (cuDNN and cuBLAS) can occupy large amounts of device memory. TensorRT allows you to control whether these libraries are used for inference via the `TacticSources` (C++, Python) attribute in the builder configuration. Note that some layer implementations require these libraries, so that when they're excluded, the network may not compile.

The CUDA infrastructure and TensorRT's device code also consume device memory. The amount of memory varies by platform, device, and TensorRT version. You can use `cudaGetMemInfo` to determine the total amount of device memory in use.

> **Note:** Since CUDA is not in control of memory on unified-memory devices, the results returned by `cudaGetMemInfo` may not be accurate on these platforms.

5.4.   Threading

In general TensorRT objects are not thread-safe. The expected runtime concurrency model is that different threads will operate on different execution contexts. The context contains the state of the network (activation values etc) during execution, so using a context concurrently in different threads results in undefined behavior.

To support this model, the following operations are thread-safe:

▶ Non-modifying operations on a runtime or engine.

▶ Deserializing an engine from a TensorRT runtime.

▶ Creating an execution context from an engine.

▶ Registering and deregistering plugins.

There are no thread-safety issues with using multiple builders in different threads; however, the builder uses timing to determine the fastest kernel for the parameters provided, and using multiple builders with the same GPU will perturb the timing and TensorRT's ability to construct optimal engines. There are no such issues using multiple threads to build with different GPUs.

5.5. Determinism

TensorRT builder uses timing to find the fastest kernel to implement a given operator. Timing kernels is subject to noise - other work running on the GPU, fluctuations in GPU clock speed, etc. Timing noise means that on successive runs of the builder, the same implementation may not be selected.

The `AlgorithmSelector` (C++, Python) interface allows you to force the builder to pick a particular implementation for a given layer. You can use this to ensure that the same kernels are picked by the builder from run to run. For more information, refer to the Algorithm Selection and Reproducible Builds section.

Once an engine has been built it is deterministic: providing the same input in the same runtime environment will produce the same output.

Chapter 6. Advanced Topics

6.1. The Timing Cache

To reduce the builder time, TensorRT creates a layer timing cache to keep the layer profiling information during the builder phase. The information it contains is specific to the targeted builder devices, CUDA and TensorRT versions, and BuilderConfig parameters that can change the layer implementation such as `BuilderFlag::kTF32` or `BuilderFlag::kREFIT`.

If there are other layers with the same input/output tensor configuration and layer parameters, the TensorRT builder skips profiling and reuses the cached result for the repeated layers. If a timing query misses in the cache, the builder times the layer and updates the cache.

The timing cache can be serialized and deserialized. You can load a serialized cache from a buffer via `IBuilderConfig::createTimingCache`:

```
ITimingCache* cache =
 config->createTimingCache(cacheFile.data(), cacheFile.size());
```

Setting the buffer size to `0` creates a new empty timing cache.

You then attach the cache to a builder configuration before building.

```
config->setTimingCache(*cache, false);
```

During the build, the timing cache can be augmented with more information as a result of cache misses. After the build, it can be serialized for use with another builder.

```
IHostMemory* serializedCache = cache->serialize();
```

If there is no timing cache attached to a builder, the builder creates its own temporary local cache and destroys it when it is done.

The cache is incompatible with algorithm selection (refer to the Algorithm Selection and Reproducible Builds section). It can be disabled by setting the `BuilderFlag`.

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

6.2. Refitting An Engine

TensorRT can *refit* an engine with new weights without having to rebuild it, however, the option to do so must be specified when building:

```
...
config->setFlag(BuilderFlag::kREFIT)
```

```
builder->buildSerializedNetwork(network, config);
```

Later, you can create a `Refitter` object:

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

Then update the weights. For example, to update the kernel weights for a convolution layer named "MyLayer":

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

The new weights should have the same count as the original weights used to build the engine. `setWeights` returns false if something went wrong, such as a wrong layer name or role or a change in the weights count.

Because of the way the engine is optimized, if you change some weights, you might have to supply some other weights too. The interface can tell you what additional weights need to be supplied.

You can use `INetworkDefinition::setWeightsName()` to name weights at build time - the ONNX parser uses this API to associate the weights with the names used in the ONNX model. Then, later you can use `setNamedWeights` to update the weights:

```
Weights newWeights = ...;
refitter->setNamedWeights("MyWeights", newWeights);
```

`setNamedWeights` and `setWeights` can be used at the same time, i.e., you can update weights with names via `setNamedWeights` and update those unnamed weights via `setWeights`.

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 int32_t n = refitter->getMissing(0, nullptr, nullptr);
std::vector<const char*> layerNames(n);
std::vector<WeightsRole> weightsRoles(n);
refitter->getMissing(n, layerNames.data(),
                        weightsRoles.data());
```

Alternatively, to get the names of all missing weights, run:

```
const int32_t n = refitter->getMissingWeights(0, nullptr);
std::vector<const char*> weightsNames(n);
refitter->getMissingWeights(n, weightsNames.data());
```

You can supply the missing weights, in any order:

```
for (int32_t i = 0; i < n; ++i)
    refitter->setWeights(layerNames[i], weightsRoles[i],
                         Weights{...});
```

The set of missing weights returned is complete, in the sense that supplying only the missing weights does not generate a need for any more weights.

Once all the weights have been provided, you can update the engine:

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

If refit returns false, check the log for a diagnostic, perhaps about weights that are still missing.

You can then delete the refitter:

```
delete refitter;
```

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

To view all refittable weights in an engine, use `refitter->getAll(...)` or `refitter->getAllWeights(...)`; similarly to how `getMissing` and `getMissingWeights` were used above.

6.3. Algorithm Selection and Reproducible Builds

The default behavior of TensorRT's optimizer is to choose the algorithms that globally minimize the execution time of the engine. It does this by timing each implementation, and sometimes, and when implementations have similar timings, it's possible that system noise will determine which will be chosen on any particular run of the builder. Different implementations will typically use different order of accumulation of floating point values, and two implementations may use different algorithms or even run at different precisions. Thus, different invocations of the builder will typically not result in engines which return bit-identical results.

Sometimes it's important to have a deterministic build, or to recreate the algorithm choices of an earlier build. By providing an implementation of the `IAlgorithmSelector` interface and attaching it to a builder configuration with `setAlgorithmSelector`, you can guide algorithm selection manually.

The method `IAlgorithmSelector::selectAlgorithms` receives an *AlgorithmContext* containing information about the algorithm requirements for a layer, and a set of *Algorithm* choices meeting those requirements. It returns the set of algorithms which TensorRT should consider for the layer.

The builder will select from these algorithms the one which minimizes the global runtime for the network. If no choice is returned and `BuilderFlag::kREJECT_EMPTY_ALGORITHMS` is unset, TensorRT interprets this to mean that any algorithm may be used for this layer. To override this behavior and generate an error if an empty list is returned, set the `BuilderFlag::kREJECT_EMPTY_ALGORITHMSS` flag.

After TensorRT has finished optimizing the network for a given profile, it calls `reportAlgorithms`, which can be used to record the final choice made for each layer.

To build a TensorRT engine deterministically, return a single choice from `selectAlgorithms`. To replay choices from an earlier build, use `reportAlgorithms` to record the choices in that build, and return them in `selectAlgorithms`.

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

> **Note:**
>
> ▶ The notion of a "layer" in algorithm selection is different from `ILayer` in `INetworkDefinition`. The "layer" in the former can be equivalent to a collection of multiple network layers due to fusion optimizations.
>
> ▶ Picking the fastest algorithm in `selectAlgorithms` may not produce the best performance for the overall network, as it may increase reformatting overhead.
>
> ▶ The timing of an `IAlgorithm` is `0` in `selectAlgorithms` if TensorRT found that layer to be a no-op.
>
> ▶ `reportAlgorithms` doesn't provide the timing and workspace information for an `IAlgorithm` that are provided to `selectAlgorithms`.

6.4. Creating A Network Definition From Scratch

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.

The following examples create a simple network with Input, Convolution, Pooling, MatrixMultiply, Shuffle, Activation, and Softmax layers.

6.4.1. C++

Code corresponding to this section can be found in sampleMNISTAPI. In this example the weights are loaded into a `weightMap` data structure used in the following code.

First create the builder and network objects. Note that in the following example, the logger is initialized via the `logger.cpp` file common to all C++ samples. The C++ sample helper classes and functions can be found in the `common.h` header file.

```
    auto builder =
 SampleUniquePtr<nvinfer1::IBuilder>(nvinfer1::createInferBuilder(sample::gLogger.getTRTLogger()));
    const auto explicitBatchFlag = 1U <<
 static_cast<uint32_t>(nvinfer1::NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
    auto network = SampleUniquePtr<nvinfer1::INetworkDefinition>(builder-
>createNetworkV2(explicitBatchFlag));
```

Refer to the Explicit vs Implicit Batch section for more information about the `kEXPLICIT_BATCH` flag.

Add the Input layer to the network by specifying the name, datatype, and full dimensions of the input tensor. A network can have multiple inputs, although in this sample there is only one:

```
auto data = network->addInput(INPUT_BLOB_NAME, datatype, Dims4{1, 1, INPUT_H, INPUT_W});
```

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});
```

> 📝 **Note:** Weights passed to TensorRT layers are in host memory.

Add the Pooling layer; note that the output from the previous layer is passed as input.

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

Add a Shuffle layer to reshape the input in preparation for a matrix multiplication:

```
int32_t const batch = input->getDimensions().d[0];
int32_t const mmInputs = input.getDimensions().d[1] * input.getDimensions().d[2] *
 input.getDimensions().d[3];
auto inputReshape = network->addShuffle(*input);
inputReshape->setReshapeDimensions(Dims{2, {batch, mmInputs}});
```

Now, add a MatrixMultiply layer. Here, the model exporter provided transposed weights, so the `kTRANSPOSE` option is specified for those.

```
IConstantLayer* filterConst = network->addConstant(Dims{2, {nbOutputs, mmInputs}},
 mWeightMap["ip1filter"]);
auto mm = network->addMatrixMultiply(*inputReshape->getOutput(0), MatrixOperation::kNONE,
 *filterConst->getOutput(0), MatrixOperation::kTRANSPOSE);
```

Add the bias, which will broadcast across the batch dimension.

```
auto biasConst = network->addConstant(Dims{2, {1, nbOutputs}}, mWeightMap["ip1bias"]);
auto biasAdd = network->addElementWise(*mm->getOutput(0), *biasConst->getOutput(0),
 ElementWiseOperation::kSUM);
```

Add the ReLU Activation layer:

```
auto relu1 = network->addActivation(*ip1->getOutput(0), ActivationType::kRELU);
```

Add the SoftMax layer to calculate the final probabilities:

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

Add a name for the output of the SoftMax layer so that the tensor can be bound to a memory buffer at inference time:

```
prob->getOutput(0)->setName(OUTPUT_BLOB_NAME);
```

Mark it as the output of the entire network:

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

The network representing the MNIST model has now been fully constructed. Refer to sections Building an Engine and Deserializing a Plan for how to build an engine and run inference with this network.

6.4.2. Python

Code corresponding to this section can be found in network_api_pytorch_mnist.

This example uses a helper class to hold some of metadata about the model:

```
class ModelData(object):
    INPUT_NAME = "data"
    INPUT_SHAPE = (1, 1, 28, 28)
    OUTPUT_NAME = "prob"
    OUTPUT_SIZE = 10
    DTYPE = trt.float32
```

In this example, the weights are imported from the Pytorch MNIST model.

```
weights = mnist_model.get_weights()
```

Create the logger, builder, and network classes.

```
TRT_LOGGER = trt.Logger(trt.Logger.ERROR)
builder = trt.Builder(TRT_LOGGER)
EXPLICIT_BATCH = 1 << (int)(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
network = builder.create_network(common.EXPLICIT_BATCH)
```

Refer to the Explicit vs Implicit Batch section for more information about the `kEXPLICIT_BATCH` flag.

Next, create the input tensor for the network, specifying the name, datatype, and shape of the tensor.

```
input_tensor = network.add_input(name=ModelData.INPUT_NAME, dtype=ModelData.DTYPE,
 shape=ModelData.INPUT_SHAPE)
```

Add a convolution layer, specifying the inputs, number of output maps, kernel shape, weights, bias, and stride:

```
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)
```

Add a pooling layer, specifying the inputs (the output of the previous convolution layer), pooling type, window size and stride:

```
pool1 = network.add_pooling(input=conv1.get_output(0), type=trt.PoolingType.MAX,
 window_size=(2, 2))
    pool1.stride = (2, 2)
```

Add the next pair of convolution and pooling layers:

```
    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)
```

Add a Shuffle layer to reshape the input in preparation for a matrix multiplication:

```
batch = input.shape[0]
mm_inputs = np.prod(input.shape[1:])
input_reshape = net.add_shuffle(input)
input_reshape.reshape_dims = trt.Dims2(batch, mm_inputs)
```

Now, add a MatrixMultiply layer. Here, the model exporter provided transposed weights, so the `kTRANSPOSE` option is specified for those.

```
filter_const = net.add_constant(trt.Dims2(nbOutputs, k), weights["fc1.weight"].numpy())
mm = net.add_matrix_multiply(input_reshape.get_output(0), trt.MatrixOperation.NONE,
 filter_const.get_output(0), trt.MatrixOperation.TRANSPOSE);
```

Add bias add which will broadcast across the batch dimension:

```
bias_const = net.add_constant(trt.Dims2(1, nbOutputs), weights["fc1.bias"].numpy())
bias_add = net.add_elementwise(mm.get_output(0), bias_const.get_output(0),
 trt.ElementWiseOperation.SUM)
```

Add the Relu activation layer:

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

Add the final fully connected layer, and mark the output of this layer as the output of the entire network:

```
    fc2_w = weights['fc2.weight'].numpy()
    fc2_b = weights['fc2.bias'].numpy()
```

```
   fc2 = network.add_fully_connected(relu1.get_output(0), ModelData.OUTPUT_SIZE, fc2_w,
 fc2_b)

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

The network representing the MNIST model has now been fully constructed. Refer to sections Building an Engine and Performing Inference for how to build an engine and run inference with this network.

6.5.   Reduced Precision

6.5.1.   Network-level Control of Precision

By default, TensorRT works in 32-bit precision, but can also execute operations using 16-bit floating point, and 8-bit quantized floating point. Using lower precision requires less memory and enables faster computation.

Reduced precision support depends on your hardware (refer to the Hardware And Precision section in the *NVIDIA TensorRT Support Matrix*). You can query the builder to check the supported precision support on a platform:

C++
```
   if (builder->platformHasFastFp16()) { … };
```
Python
```
   if builder.platform_has_fp16:
```

Setting flags in the builder configuration informs TensorRT that it may select lower-precision implementations:

C++
```
   config->setFlag(BuilderFlag::kFP16);
```
Python
```
   config.set_flag(trt.BuilderFlag.FP16)
```

There are three precision flags: FP16, INT8, and TF32, and they may be enabled independently. Note that TensorRT will still choose a higher-precision kernel if it results in overall lower runtime, or if no low-precision implementation exists.

When TensorRT chooses a precision for a layer, it automatically converts weights as necessary to run the layer.

sampleGoogleNet and sampleMNIST provide examples of using these flags.

While using FP16 and TF32 precisions is relatively straightforward, there is additional complexity when working with INT8. Refer to the Working With INT8 chapter for more details.

6.5.2.   Layer-level Control of Precision

The builder-flags provide permissive, coarse-grained control. However, sometimes part of a network requires higher dynamic range or is sensitive to numerical precision. You can constrain the input and output types per layer:

C++
```
   layer->setPrecision(DataType::kFP16)
```

Python
```
layer.precision = trt.fp16
```

This provides a *preferred type* (here, `DataType::kFP16`) for the inputs and outputs.

You may further set preferred types for the layer's outputs:
C++
```
layer->setOutputType(out_tensor_index, DataType::kFLOAT)
```
Python
```
layer.set_output_type(out_tensor_index, trt.fp16)
```

The computation will use the same floating-point type as is preferred for the inputs. Most TensorRT implementations have the same floating-point types for input and output; however, Convolution, Deconvolution, and FullyConnected can support quantized INT8 input and unquantized FP16 or FP32 output, as sometimes working with higher-precision outputs from quantized inputs is necessary to preserve accuracy.

Setting the precision constraint hints to TensorRT that it should select a layer implementation whose inputs and outputs match the preferred types, inserting reformat operations if the outputs of the previous layer and the inputs to the next layer do not match the requested types. Note that TensorRT will only be able to select an implementation with these types if they are also enabled via the flags in the builder configuration.

By default, TensorRT chooses such an implementation only if it results in a higher-performance network. If another implementation is faster, TensorRT uses it and issues a warning. You can override this behavior by preferring the type constraints in the builder configuration.
C++
```
config->setFlag(BuilderFlag::kPREFER_PRECISION_CONSTRAINTS)
```
Python
```
config.set_flag(trt.BuilderFlag.PREFER_PRECISION_CONSTRAINTS)
```

If the constraints are preferred, TensorRT obeys them unless there is no implementation with the preferred precision constraints, in which case it issues a warning and uses the fastest available implementation.

To change the warning to an error, use `OBEY` instead of `PREFER`:
C++
```
config->setFlag(BuilderFlag::kOBEY_PRECISION_CONSTRAINTS);
```
Python
```
config.set_flag(trt.BuilderFlag.OBEY_PRECISION_CONSTRAINTS);
```

sampleINT8API illustrates the use of reduced precision with these APIs.

Precision constraints are optional - you can query to determine whether a constraint has been set using `layer->precisionIsSet()` in C++ or `layer.precision_is_set` in Python. If a precision constraint is not set, then the result returned from `layer->getPrecision()` in C++, or reading the `precision` attribute in Python, is not meaningful. Output type constraints are similarly optional.

Note that there is a distinction between `layer->getOutput(i)->setType()` and `layer->setOutputType()` - the former is an optional type that constrains the implementation that TensorRT will choose for a layer. The latter is mandatory (defaulting to FP32) and specifies the type of a network output. If they are different, TensorRT will insert a cast to ensure that both specifications are respected. Thus if you are calling `setOutputType()` for a layer that

produces a network output, you should in general also configure the corresponding network output to have the same type.

6.5.3. Enabling TF32 Inference Using C++

TensorRT allows the 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 the 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 a high dynamic range for weights or activations.

There is no guarantee that TF32 Tensor Cores are actually used, and there's no way to force the implementation to use them - TensorRT can fall back to FP32 at any time and always falls back if the platform does not support TF32. However you can disable their use by clearing the TF32 builder flag.

C++

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

Python

```
config.clear_flag(trt.BuilderFlag.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 can cause unpredictable precision/performance effects. It is best left unset when an engine is run.**

📝 **Note:** Unless your application requires the higher dynamic range provided by TF32, FP16 will be a better solution since it almost always yields faster performance.

6.6. I/O Formats

TensorRT optimizes a network using many different data formats. In order to allow efficient passing of data between TensorRT and a client application, these underlying data formats are exposed at network I/O boundaries, i.e. for Tensors marked as network input or output, and when passing data to and from plugins. For other tensors, TensorRT picks formats that result in the fastest overall execution, and may insert reformats to improve performance.

You can assemble an optimal data pipeline by profiling the available I/O formats in combination with the formats most efficient for the operations preceding and following TensorRT.

To specify I/O formats, you specify one or more formats in the form of a bitmask.

The following example sets the input tensor format to `TensorFormat::kHWC8`. Note that this format only works for `DataType::kHALF`, so the data type must be set accordingly.

C++

```
auto formats = 1U << TensorFormat::kHWC8;
network->getInput(0)->setAllowedFormats(formats);
network->getInput(0)->setType(DataType::kHALF);
```

Python

```
formats = 1 << int(tensorrt.TensorFormat.HWC8)
network.get_input(0).allowed_formats = formats
network.get_input(0).dtype = tensorrt.DataType.HALF
```

It is possible to make TensorRT avoid inserting reformatting at the network boundaries, by setting the builder configuration flag `DIRECT_IO`. This flag is generally counter-productive for two reasons:

▶ The resulting engine might be slower than if TensorRT had been allowed to insert reformatting. Reformatting may sound like wasted work, but it can allow coupling of the most efficient kernels.

▶ The build will fail if TensorRT cannot build an engine without introducing such reformatting. The failure may happen only for some target platforms, because of what formats are supported by kernels for those platforms.

The flag exists for the sake of users who want full control over whether reformatting happens at I/O boundaries, such as to build engines that run solely on DLA without falling back to the GPU for reformatting.

sampleIOFormats illustrates how to specify IO formats using C++.

The following table shows the supported formats.

Table 1.   Supported I/O formats

Format	kINT32	kFLOAT	kHALF	kINT8
kLINEAR	Only for GPU	Supported	Supported	Supported
kCHW2	N/A	N/A	Only for GPU	N/A
kCHW4	N/A	N/A	Supported	Supported
kHWC8	N/A	N/A	Only for GPU	N/A
kCHW16	N/A	N/A	Supported	N/A
kCHW32	N/A	Only for GPU	Only for GPU	Supported
kDHWC8	N/A	N/A	Only for GPU	N/A
kCDHW32	N/A	N/A	Only for GPU	Only for GPU
kHWC	N/A	Only for GPU	N/A	N/A
kDLA_LINEAR	N/A	N/A	Only for DLA	Only for DLA
kDLA_HWC4	N/A	N/A	Only for DLA	Only for DLA

Format	kINT32	kFLOAT	kHALF	kINT8
kHWC16	N/A	N/A	Only for NVIDIA Ampere GPUs and later	N/A

Note that for the vectorized formats, the channel dimension must be zero-padded to the multiple of the vector size. For example, if a input binding has dimensions of [16,3,224,224], kHALF data type, and kHWC8 format, then the actual required size of the binding buffer would be 16***8***224*224*sizeof(half) bytes, even though the engine->getBindingDimension() API will return tensor dimensions as [16,3,224,224]. The values in the padded part (i.e. where C=3,4,…,7 in this example) must be filled with zeros.

Refer to Data Format Descriptions for how the data are actually laid out in memory for these formats.

6.7.   Compatibility of Serialized Engines

Serialized engines are only guaranteed to work correctly when used with the same operating systems, CPU architectures, GPU models, and TensorRT versions used to serialize the engines.

TensorRT checks the following attributes of the engine and will fail to deserialize if they do not match the environment in which the engine was serialized:

▶ major, minor, patch, and build versions of TensorRT

▶ compute capability (major and minor versions)

This ensures that kernels selected during the build phase are present and can run. In addition, the APIs that TensorRT uses to select and configure kernels from cuDNN and cuBLAS do not support cross-device compatibility, so disable the use of these tactic sources in the builder configuration.

TensorRT additionally checks the following properties and will issue a warning if they do not match:

▶ Global memory bus width

▶ L2 cache size

▶ Maximum shared memory per block and per multiprocessor

▶ Texture alignment requirement

▶ Number of multiprocessors

▶ Whether the GPU device is integrated or discrete

If GPU clock speeds differ between engine serialization and runtime systems, the chosen tactics from the serialization system may not be optimal for the runtime system and may incur some performance degradation.

If the device memory available during deserialization is smaller than the amount during serialization, deserialization may fail due to memory allocation failures.

When building small models on large devices, TensorRT may choose kernels which are less efficient but scale better across the available resources. Thus if optimizing a single TensorRT engine for use on multiple devices in the same architecture, the best approach is to run the builder on the smallest device. Alternatively, you can build the engine on the larger device with limited compute resources (refer to the Limiting Compute Resources section).

6.8. Explicit vs Implicit Batch

TensorRT supports two modes for specifying a network: explicit batch and implicit batch.

In *implicit batch* mode, every tensor has an implicit batch dimension and all other dimensions must have constant length. This mode was used by early versions of TensoRT, and is now deprecated but continues to be supported for backwards compatibility.

In *explicit batch* mode, all dimensions are explicit and can be dynamic, that is their length can change at execution time. Many new features, such as dynamic shapes and loops, are available only in this mode. It is also required by the ONNX parser.

For example, consider a network that processes N images of size HxW with 3 channels, in NCHW format. At runtime, the input tensor has dimensions [N,3,H,W]. The two modes differ in how the `INetworkDefinition` specifies the tensor's dimensions:

▶ In explicit batch mode, the network specifies [N,3,H,W].

▶ In implicit batch mode, the network specifies only [3,H,W]. The batch dimension N is implicit.

Operations that "talk across a batch" are impossible to express in implicit batch mode because there is no way to specify the batch dimension in the network. Examples of inexpressible operations in implicit batch mode:

▶ reducing across the batch dimension

▶ reshaping the batch dimension

▶ transposing the batch dimension with another dimension

The exception is that a tensor can be *broadcast* across the entire batch, via method `ITensor::setBroadcastAcrossBatch` for network inputs, and implicit broadcasting for other tensors.

Explicit batch mode erases the limitations - the batch axis is axis 0. A more accurate term for explicit batch would be "batch oblivious", because in this mode, TensorRT attaches no special semantic meaning to the leading axis, except as required by specific operations. Indeed in explicit batch mode there might not even be a batch dimension (such as a network that handles only a single image) or there might be multiple batch dimensions of unrelated lengths (such as comparison of all possible pairs drawn from two batches).

The choice of explicit vs. implicit batch must be specified when creating the INetworkDefinition, via a flag. Here's the C++ code for explicit batch mode:

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

For implicit batch, use `createNetwork` or pass a 0 to `createNetworkV2`.

Here's the Python code for explicit batch mode:

```
builder = trt.Builder(...)
builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
```

For implicit batch, omit the argument or pass a 0.

6.9.   Sparsity

NVIDIA Ampere Architecture GPUs support Structured Sparsity. To make use of this feature to achieve higher inference performance, the convolution kernel weights and/or the fully-connected weights must meet the following requirements:

For each output channel and for each spatial pixel in the kernel weights, every 4 input channels must have at least 2 zeros. In other words, assuming that the kernel weights have the shape `[K, C, R, S]` and `C % 4 == 0`, then the requirement is:

```
for k in K:
    for r in R:
        for s in S:
            for c_packed in range(0, C // 4):
                num_zeros(weights[k, c_packed*4:(c_packed+1)*4, r, s]) >= 2
```

To enable the sparsity feature, set the `kSPARSE_WEIGHTS` flag in the builder config and make sure that kFP16 and/or kINT8 modes are enabled. For example:

C++

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

Python

```
config.set_flag(trt.BuilderFlag.SPARSE_WEIGHTS)
```

At the end of the TensorRT logs when the TensorRT engine is built, TensorRT reports which layers contain weights that meet the structures sparsity requirement, and in which layers TensorRT selects tactics that make use of the structured sparsity. In some cases, tactics with structured sparsity can be slower than normal tactics and TensorRT will choose normal tactics in these cases. The following output shows an example of TensorRT logs showing information about sparsity:

```
[03/23/2021-00:14:05] [I] [TRT] (Sparsity) Layers eligible for sparse math: conv1, conv2,
 conv3
[03/23/2021-00:14:05] [I] [TRT] (Sparsity) TRT inference plan picked sparse implementation
 for layers: conv2, conv3
```

Forcing kernel weights to have structured sparsity patterns can lead to accuracy loss. To recover lost accuracy with further fine-tuning, refer to the Automatic SParsity tool in PyTorch.

To measure inference performance with structured sparsity using `trtexec`, refer to the trtexec section.

6.10.   Empty Tensors

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.

If an engine binding is an empty tensor, it still needs a non-null memory address, and different tensors should have different addresses.. This is consistent with the C++ rule that every object has a unique address, for example, `new float[0]` returns a non-null pointer. If using a memory allocator that might return a null pointer for zero bytes, ask for at least one byte instead.

Refer to TensorRT Layers for any per-layer special handling of empty tensors.

6.11. Reusing Input Buffers

TensorRT also includes an optional CUDA event as an argument to the `enqueue` method that is signaled once the input buffers are free to be reused. This allows the application to immediately start refilling the input buffer region for the next inference in parallel with finishing the current inference. For example:

C++
```
context->enqueueV2(&buffers[0], stream, &inputReady);
```
Python
```
context.execute_async_v2(buffers, stream_ptr, inputReady)
```

6.12. Engine Inspector

TensorRT provides the `IEngineInspector` API to inspect the information inside a TensorRT engine. Call the `createEngineInspector()` from a deserialized engine to create an engine inspector, and then call `getLayerInformation()` or `getEngineInformation()` inspector APIs to get the information of a specific layer in the engine or the entire engine, respectively. You can print out the information of the first layer of the given engine, as well as the overall information of the engine, as follows:

C++
```
auto inspector = std::unique_ptr<IEngineInspector>(engine->createEngineInspector());
inspector->setExecutionContext(context); // OPTIONAL
std::cout << inspector->getLayerInformation(0, LayerInformationFormat::kJSON); // Print
  the information of the first layer in the engine.
std::cout << inspector->getEngineInformation(LayerInformationFormat::kJSON); // Print the
  information of the entire engine.
```
Python
```
inspector = engine.create_engine_inspector();
inspector.execution_context = context; # OPTIONAL
print(inspector.get_layer_information(0, LayerInformationFormat.JSON); # Print the
  information of the first layer in the engine.
print(inspector.get_engine_information(LayerInformationFormat.JSON); # Print the
  information of the entire engine.
```

Note that the level of detail in the engine/layer information depends on the `ProfilingVerbosity` builder config setting when the engine is built. By default, `ProfilingVerbosity` is set to `kLAYER_NAMES_ONLY`, so only the layer names will be printed. If `ProfilingVerbosity` is set to `kNONE`, then no information will be printed; if it is set to `kDETAILED`, then detailed information will be printed.

Below are some examples of layer information printed by `getLayerInformation()` API depending on the `ProfilingVerbosity` setting:

kLAYER_NAMES_ONLY

```
"node_of_gpu_0/res4_0_branch2a_1 + node_of_gpu_0/res4_0_branch2a_bn_1 + node_of_gpu_0/res4_0_branch2a_bn_2"
```

kDETAILED

```
{
  "Name": "node_of_gpu_0/res4_0_branch2a_1 + node_of_gpu_0/res4_0_branch2a_bn_1 +
 node_of_gpu_0/res4_0_branch2a_bn_2",
  "LayerType": "CaskConvolution",
  "Inputs": [
  {
    "Name": "gpu_0/res3_3_branch2c_bn_3",
    "Dimensions": [16,512,28,28],
    "Format/Datatype": "Thirty-two wide channel vectorized row major Int8 format."
  }],
  "Outputs": [
  {
    "Name": "gpu_0/res4_0_branch2a_bn_2",
    "Dimensions": [16,256,28,28],
    "Format/Datatype": "Thirty-two wide channel vectorized row major Int8 format."
  }],
  "ParameterType": "Convolution",
  "Kernel": [1,1],
  "PaddingMode": "kEXPLICIT_ROUND_DOWN",
  "PrePadding": [0,0],
  "PostPadding": [0,0],
  "Stride": [1,1],
  "Dilation": [1,1],
  "OutMaps": 256,
  "Groups": 1,
  "Weights": {"Type": "Int8", "Count": 131072},
  "Bias": {"Type": "Float", "Count": 256},
  "AllowSparse": 0,
  "Activation": "RELU",
  "HasBias": 1,
  "HasReLU": 1,
  "TacticName":
 "sm80_xmma_fprop_implicit_gemm_interleaved_i8i8_i8i32_f32_nchw_vect_c_32kcrs_vect_c_32_nchw_vect_c_32_ti
  "TacticValue": "0x11bde0e1d9f2f35d"
}
```

In addition, when the engine is built with dynamic shapes, the dynamic dimensions in the engine information will be shown as `-1` and the tensor format information will not be shown because these fields depend on the actual shape at inference phase. To get the engine information for a specific inference shape, create an `IExecutionContext`, set all the input dimensions to the desired shapes, and then call `inspector->setExecutionContext(context)`. After the context is set, the inspector will print the engine information for the specific shape set in the context.

The `trtexec` tool provides the `--profilingVerbosity`, `--dumpLayerInfo`, and `--exportLayerInfo` flags which can be used to get the engine information of a given engine. Refer to the [trtexec](#) section for more details.

Currently, only binding information and layer information, including the dimensions of the intermediate tensors, precisions, formats, tactic indices, layer types, and layer parameters, are included in the engine information. More information may be added into the engine inspector output as new keys in the output JSON object in future TensorRT versions. More specifications about the keys and the fields in the inspector output will also be provided.

Chapter 7. Working With INT8

7.1. Introduction to Quantization

TensorRT supports the use of 8-bit integers to represent quantized floating point values. The quantization scheme is *symmetric uniform* quantization - quantized values are represented in signed INT8, and the transformation from quantized to unquantized values is simply a multiplication. In the reverse direction, quantization uses the reciprocal scale, followed by rounding and clamping.

To enable the use of any quantized operations, the INT8 flag must be set in the builder configuration.

7.1.1. Quantization Workflows

There are two workflows for creating quantized networks:

Post-training quantization (PTQ) derives scale factors after the network has been trained. TensorRT provides a workflow for PTQ, called *calibration*, where it measures the distribution of activations within each activation tensor as the network executes on representative input data, then uses that distribution to estimate a scale value for the tensor.

Quantization-aware training (QAT) computes scale factors during training. This allows the training process to compensate for the effects of the quantization and dequantization operations.

TensorRT's Quantization Toolkit is a PyTorch library that helps produce QAT models that can be optimized by TensorRT. You can also utilize the toolkit's PTQ recipe to perform PTQ in PyTorch and export to ONNX.

7.1.2. Explicit vs Implicit Quantization

Quantized networks can be represented in two ways:

In *implicitly quantized* networks, each quantized tensor has an associated scale. When reading and writing the tensor, the scale is used to implicitly quantize and dequantize values.

When processing implicitly quantized networks, TensorRT treats the model as a floating-point model when applying the graph optimizations, and uses INT8 opportunistically to optimize layer execution time. If a layer runs faster in INT8, then it executes in INT8. Otherwise, FP32 or FP16 are used. In this mode, TensorRT is optimizing for performance only, and you have

little control over where INT8 is used - even if you explicitly set the precision of a layer at the API level, TensorRT may fuse that layer with another during graph optimization, and lose the information that it must execute in INT8. TensorRT's PTQ capability generates an implicitly quantized network.

In *explicitly quantized* networks, the scaling operations to transform between the quantized and unquantized values are represented explicitly by `IQuantizeLayer` (C++, Python) and `IDequantizeLayer` (C++, Python) nodes in the graph - these will henceforth be referred to as Q/DQ nodes. By contrast with implicit quantization, the explicit form specifies exactly where conversion to and from INT8 is performed, and the optimizer will perform only precision conversions that are dictated by the semantics of the model, even if:

▶ adding extra conversions could increase layer precision (for example, choosing an FP16 kernel implementation over an INT8 implementation)

▶ adding extra conversions results in an engine that executes faster (for example, choosing an INT8 kernel implementation to execute a layer specified as having float precision or vice versa)

ONNX uses an explicitly quantized representation - when a model in PyTorch or TensorFlow is exported to ONNX, each fake-quantization operation in the framework's graph is exported as Q followed by DQ. Since TensorRT preserves the semantics of these layers, you can expect task accuracy very close to that seen in the framework. While optimizations preserve the placement of quantization and dequantization, they may change the order of floating-point operations in the model, so results will not be bitwise identical.

Note that by contrast with TensorRT's PTQ, performing either QAT or PTQ in a framework and then exporting to ONNX will result in an explicitly quantized model.

Table 2.   Implicit vs Explicit Quantization

	Implicit Quantization	Explicit Quantization
User control over precision	Little control: INT8 is used in all kernels for which it accelerates performance.	Full control over quantization/dequantization boundaries.
Optimization criterion	Optimize for performance.	Optimize for performance while maintaining arithmetic precision (accuracy).
API	▶ Model + Scales (dynamic range API) ▶ Model + Calibration data	Model with Q/DQ layers.
Quantization scales	Weights: ▶ Set by TensorRT (internal) ▶ Range [-127, 127] Activations: ▶ Set by calibration or specified by the user	Weights and activations: ▶ Specified using Q/DQ ONNX operators ▶ Range [-128, 127]

	Implicit Quantization	Explicit Quantization
	▶ Range `[-128, 127]`	

For more background on quantization, refer to the Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation paper.

7.1.3.   Per-Tensor and Per-Channel Quantization

There are two common quantization scale granularities:

▶ *Per-tensor quantization*: in which a single scale value (scalar) is used to scale the entire tensor.

▶ *Per-channel quantization*: in which a scale tensor is broadcast along the given axis - for convolutional neural networks, this is typically the channel axis.

With explicit quantization, weights can be quantized using per-tensor quantization or they can be quantized using per-channel quantization. In either case, the scale precision is FP32. Activations can only be quantized using per-tensor quantization.

When using per-channel quantization, the axis of quantization must be the output-channel axis. For example, when the weights of 2D convolution are described using `KCRS` notation, `K` is the output-channel axis, and the weights quantization can be described as:

```
For each k in K:
    For each c in C:
        For each r in R:
            For each s in S:
                output[k,c,r,s] := clamp(round(input[k,c,r,s] / scale[k]))
```

The scale is a vector of coefficients and must have the same size as the quantization axis. The quantization scale must consist of all positive float coefficients. The rounding method is rounding-to-nearest ties-to-even and clamping is in the range `[-128, 127]`.

Dequantization is performed similarly except for the pointwise operation which is defined as:

```
output[k,c,r,s] := input[k,c,r,s] * scale[k]
```

TensorRT supports only per-tensor quantization for activation tensors, but supports per-channel weight quantization for convolution, deconvolution, fully connected layers, and MatMul where the second input is constant and both input matrices are 2-dimensional.

7.2.   Setting Dynamic Range

TensorRT provides APIs to set *dynamic range* (the range that must be represented by the quantized tensor) directly, to support implicit quantization where these values have been calculated outside TensorRT.

The API allows setting the dynamic range for a tensor using minimum and maximum values. Since TensorRT currently supports only symmetric range, the scale is calculated using `max(abs(min_float), abs(max_float))`. Note that when `abs(min_float) != abs(max_float)`, TensorRT uses a larger dynamic-range than configured, which may increase the rounding error.

Dynamic range is needed for all floating-point inputs and outputs of an operation that will execute in INT8.

You can set the dynamic range for a tensor as follows:

C++
```
tensor->setDynamicRange(min_float, max_float);
```
Python
```
tensor.dynamic_range = (min_float, max_float)
```

sampleINT8API illustrates the use of these APIs in C++.

7.3.   Post-Training Quantization using Calibration

In post-training quantization, TensorRT computes a scale value for each tensor in the network. This process, called *calibration,* requires you to supply representative input data on which TensorRT runs the network to collect statistics for each activation tensor.

The amount of input data required is application-dependent, but experiments indicate that about 500 images are sufficient for calibrating ImageNet classification networks.

Given the statistics for an activation tensor, deciding on the best scale value is not an exact science - it requires balancing two sources of error in the quantized representation: *discretization error* (which increases as the range represented by each quantized value becomes larger) and *truncation error* (where values are clamped to the limits of the representable range.) Thus, TensorRT provides multiple different calibrators which calculate the scale in different ways. Older calibrators also performed layer fusion for GPU to optimize away unneeded Tensors before performing calibration. This can be problematic when using DLA, where fusion patterns may be different, and can be overridden using the `kCALIBRATE_BEFORE_FUSION` quantization flag.

IInt8EntropyCalibrator2
    Entropy calibration chooses the tensor's scale factor to optimize the quantized tensor's information-theoretic content, and usually suppresses outliers in the distribution. This is the current and recommended entropy calibrator and is required for DLA. Calibration happens before Layer fusion by default. It is recommended for CNN-based networks.

IInt8MinMaxCalibrator
    This calibrator uses the entire range of the activation distribution to determine the scale factor. It seems to work better for NLP tasks. Calibration happens before Layer fusion by default. This is recommended for networks such as NVIDIA BERT (an optimized version of Google's official implementation).

IInt8EntropyCalibrator
    This is the original entropy calibrator. It is less complicated to use than the LegacyCalibrator and typically produces better results. Calibration happens after Layer fusion by default.

IInt8LegacyCalibrator
    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. You can customize this calibrator

to implement percentile max, for example, 99.99% percentile max is observed to have best accuracy for NVIDIA BERT.

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

1. Build a 32-bit engine, run it on the calibration set, and record a histogram for each tensor of the distribution of activation values.
2. Build from the histograms a calibration table providing a scale value for each tensor.
3. Build the INT8 engine from the calibration table and the network definition.

Calibration can be slow; therefore the output of step 2 (the calibration table) can be cached and reused. This is useful when building the same network multiple times, for example, on multiple platforms - in particular, it can simplify workflow to build the calibration table on a machine with a discrete GPU and then reuse it on an embedded platform. A sample calibration table can be found [here](#).

Before running calibration, TensorRT queries the calibrator implementation to see if it has access to a cached table. If so, it proceeds directly to step 3 above. Cached data is passed as a pointer and length.

The calibration cache data is portable between platforms and when building engines for different devices as long as the calibration happens before layer fusion. This means the calibration cache is portable when using the `IInt8EntropyCalibrator2` or `IInt8MinMaxCalibrator` calibrators by default, or when `QuantizationFlag::kCALIBRATE_BEFORE_FUSION` is set. Fusions are not guaranteed to be the same across platforms or devices, so calibrating after layer fusion may not result in a portable calibration cache.

As well as quantizing activations, TensorRT must also quantize weights. It uses symmetric quantization with a quantization scale calculated using the maximum absolute values found in the weight tensor. For convolution, deconvolution and fully-connected weights, scales are per-channel.

> 📄 **Note:** When the builder is configured to use INT8 I/O, TensorRT still expects calibration data to be in FP32. You can create FP32 calibration data by casting INT8 I/O calibration data to FP32 precision. You must also ensure that FP32 cast calibration data is in the range `[-128.0F, 127.0F]` and so can be converted to INT8 data without any precision loss.

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

> 📄 **Note:** Calibration is deterministic - that is, if you provide TensorRT with the same input to calibration in the same order on the same device, the scales generated will be the same across different runs. The data in the calibration cache will be bit-wise identical when generated using the same device with the same batch size when provided with identical calibration inputs. The exact data in the calibration cache is not guaranteed to be bit-wise identical when generated using different devices, different batch sizes, or using different calibration inputs.

7.3.1.   INT8 Calibration Using C++

To provide calibration data to TensorRT, implement the `IInt8Calibrator` interface.

About this task

The builder invokes the calibrator as follows:

- ► First, it queries the interface for the batch size and calls `getBatchSize()` to determine the size of the input batch to expect.

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

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

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

To cache the calibration table, implement the `writeCalibrationCache()` and `readCalibrationCache()` methods.

For more information about configuring INT8 calibrator objects, see [sampleINT8](#).

7.3.2.   Calibration Using Python

The following steps illustrate how to create an INT8 calibrator object using the Python API.

Procedure

1. Import TensorRT:

```
import tensorrt as trt
```

2. Similar to test/validation datasets, use a set of input files as a calibration dataset. Make sure the calibration files are representative of the overall inference data files. For TensorRT to use the calibration files, you must create a `batchstream` object. A `batchstream` object is 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
```

7.4.   Explicit Quantization

When TensorRT detects the presence of Q/DQ layers in a network, it builds an engine using explicit-precision processing logic.

A Q/DQ network must be built with the INT8-precision builder flag enabled:

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

In explicit-quantization, network changes of representation to and from INT8 are explicit, therefore, INT8 must not be used as a type constraint.

7.4.1.   Quantized Weights

Weights of Q/DQ models must be specified using FP32 data type. The weights are quantized by TensorRT using the scale of the `IQuantizeLayer` that operates on the weights. The quantized weights are stored in the `Engine` file. Pre-quantized weights can also be used but must be specified using FP32 data-type. The scale of the Q node must be set to `1.0F`, but the DQ node must be the real scale value.

7.4.2.   ONNX Support

When a model trained in PyTorch or TensorFlow using Quantization Aware Training (QAT) is exported to ONNX, each fake-quantization operation in the framework's graph is exported as a pair of <u>QuantizeLinear</u>and <u>DequantizeLinear</u>ONNX operators.

When TensorRT imports ONNX models, the ONNX `QuantizeLinear` operator is imported as an `IQuantizeLayer` instance, and the ONNX `DequantizeLinear` operator is imported as an `IDequantizeLayer` instance. ONNX using opset 10 introduced support for QuantizeLinear/ DequantizeLinear, and a quantization-axis attribute was added in opset 13 (required for per-channel quantization). PyTorch 1.8 introduced support for exporting PyTorch models to ONNX using opset 13.

> ⚠️ **WARNING: The ONNX GEMM operator is an example that can be quantized per channel. PyTorch `torch.nn.Linear` layers are exported as an ONNX GEMM operator with `(K, C)` weights layout and with the `transB` GEMM attribute enabled (this transposes the weights before performing the GEMM operation). TensorFlow, on the other hand, pre-transposes the weights `(C, K)` before ONNX export:**
>
> ▶ **PyTorch:** $y = xW^T$
> ▶ **TensorFlow:** $y = xW$

PyTorch weights are therefore transposed by TensorRT. The weights are quantized by TensorRT before they are transposed, so GEMM layers originating from ONNX QAT models that were exported from PyTorch use dimension `0` for per-channel quantization (axis `K = 0`); while models originating from TensorFlow use dimension `1` (axis `K = 1`).

TensorRT does not support pre-quantized ONNX models that use INT8 tensors or quantized operators. Specifically, the following ONNX quantized operators are *not* supported and generates an import error if they are encountered when TensorRT imports the ONNX model:

▶ <u>QLinearConv</u>/<u>QLinearMatmul</u>

▶ <u>ConvInteger</u>/<u>MatmulInteger</u>

7.4.3.   TensorRT Processing Of Q/DQ Networks

When TensorRT optimizes a network in Q/DQ-mode, the optimization process is limited to optimizations that do not change the arithmetic correctness of the network. Bit-level accuracy is rarely possible since the order of floating-point operations can produce different results (for example, rewriting $a * s + b * s$ as $(a + b) * s$ is a valid optimization). Allowing these

differences is fundamental to back-end optimization in general, and this also applies to converting a graph with Q/DQ layers to use INT8 computation.

Q/DQ layers control the compute and data precision of a network. An `IQuantizeLayer` instance converts an FP32 tensor to an INT8 tensor by employing quantization, and an `IDequantizeLayer` instance converts an INT8 tensor to an FP32 tensor by means of dequantization. TensorRT expects a Q/DQ layer pair on each of the inputs of quantizable-layers. Quantizable-layers are deep-learning layers that can be converted to quantized layers by fusing with `IQuantizeLayer` and `IDequantizeLayer` instances. When TensorRT performs these fusions, it replaces the quantizable-layers with quantized layers that actually operate on INT8 data using INT8 compute operations.

For the diagrams used in this chapter, green designates INT8 precision and blue designates floating-point precision. Arrows represent network activation tensors and squares represent network layers.

Figure 1.   A quantizable `AveragePool` layer (in blue) is fused with a DQ layer and a Q layer. All three layers are replaced by a quantized `AveragePool` layer (in green).
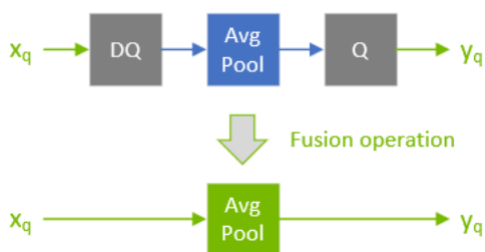
During network optimization, TensorRT moves Q/DQ layers in a process called Q/DQ propagation. The goal in propagation is to maximize the proportion of the graph that can be processed at low precision. Thus, TensorRT propagates Q nodes backwards (so that quantization happens as early as possible) and DQ nodes forward (so that dequantization happens as late as possible). Q-layers can swap places with layers that commute-with-Quantization and DQ-layers can swap places with layers that commute-with-Dequantization.

A layer Op commutes with quantization if $Q\left(\mathrm{Op}\left(x\right)\right) == \mathrm{Op}\left(Q\left(x\right)\right)$

Similarly, a layer Op commutes with dequantization if $\mathrm{Op}\left(DQ\left(x\right)\right) == DQ\left(\mathrm{Op}\left(x\right)\right)$

The following diagram illustrates DQ forward-propagation and Q backward-propagation. These are legal rewrites of the model because Max Pooling has an INT8 implementation and because Max Pooling commutes with DQ and with Q.

Figure 2. An illustration depicting a DQ forward-propagation and Q backward-propagation.
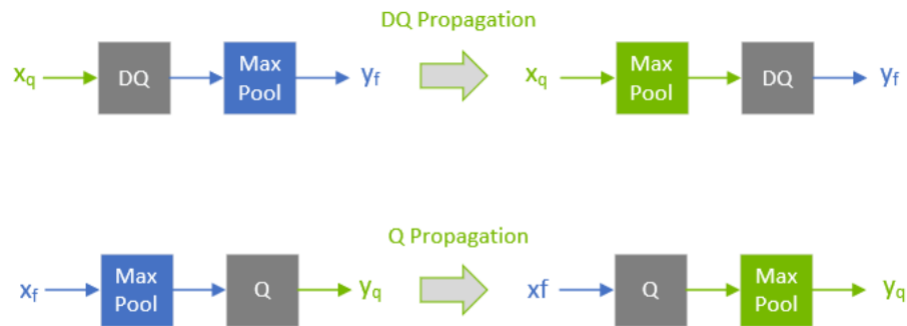
> **Note:**
>
> To understand Max Pooling commutation, let's look at the output of the maximum-pooling operation applied to some arbitrary input. Max Pooling is applied to groups of input coefficients and outputs the coefficient with the maximum value. For group `i` composed of coefficients: $\{x_0 .. x_m\}$:
>
> $$\text{output}_i := \max(\{x_0,\ x_1,\ ... x_m\}) = \max(\{\max(\{\max(\{x_0,\ x_1\}),\ x_2\}),\ ... x_m\})$$
>
> It is therefore enough to look at two arbitrary coefficients without loss of generality (WLOG):
>
> $$x_j = \max(\{x_j,\ x_k\}) \text{ for } x_j >= x_k$$
>
> For quantization function $Q(a,\ \text{scale},\ x_{\max},\ x_{\min}) := \text{truncate}(\text{round}(a/\text{scale}),\ x_{\max},\ x_{\min})$, with $\text{scale} > 0$, note that (without providing proof, and using simplified notation):
>
> $$Q(x_j,\ \text{scale}) >= Q(x_k,\ \text{scale}) \text{ for } x_j >= x_k$$
>
> Therefore:
>
> $$\max(\{Q(x_j,\ \text{scale}),\ Q(x_k,\ \text{scale})\}) = Q(x_j,\ \text{scale}) \text{ for } x_j >= x_k$$
>
> However, by definition:
>
> $$Q(\max(\{x_j,\ x_k\}),\ \text{scale}) = Q(x_j,\ \text{scale}) \text{ for } x_j >= x_k$$
>
> Function max commutes-with-quantization and so does Max Pooling.
>
> Similarly for dequantization, function $\text{DQ}(a, \text{scale}) := a * \text{scale}$ with $\text{scale} > 0$ we can show that:
>
> $$\max(\{\text{DQ}(x_j,\ \text{scale}),\ \text{DQ}(x_k,\ \text{scale})\}) = \text{DQ}(x_j,\ \text{scale}) = \text{DQ}(\max(\{x_j,\ x_k\}),\ \text{scale}) \text{ for } x_j >= x_k$$

There is a distinction between how quantizable-layers and commuting-layers are processed. Both types of layers can compute in INT8, but quantizable-layers also fuse with DQ input layers and a Q output layer. For example, an `AveragePooling` layer (quantizable) does not

commute with either Q or DQ, so it is quantized using Q/DQ fusion as illustrated in the first diagram. This is in contrast to how Max Pooling (commuting) is quantized.

7.4.4.   Q/DQ Layer-Placement Recommendations

The placement of Q/DQ layers in a network affects performance and accuracy. Aggressive quantization can lead to degradation in model accuracy because of the error introduced by quantization. But quantization also enables latency reductions. Listed here are some recommendations for placing Q/DQ layers in your network.

Quantize all inputs of weighted-operations (Convolution, Transposed Convolution and GEMM). Quantization of the weights and activations reduces bandwidth requirements and also enables INT8 computation to accelerate bandwidth-limited and compute-limited layers.

Figure 3.   Two examples of how TensorRT fuses convolutional layers. On the left, only the inputs are quantized. On the right, both inputs and output are quantized.
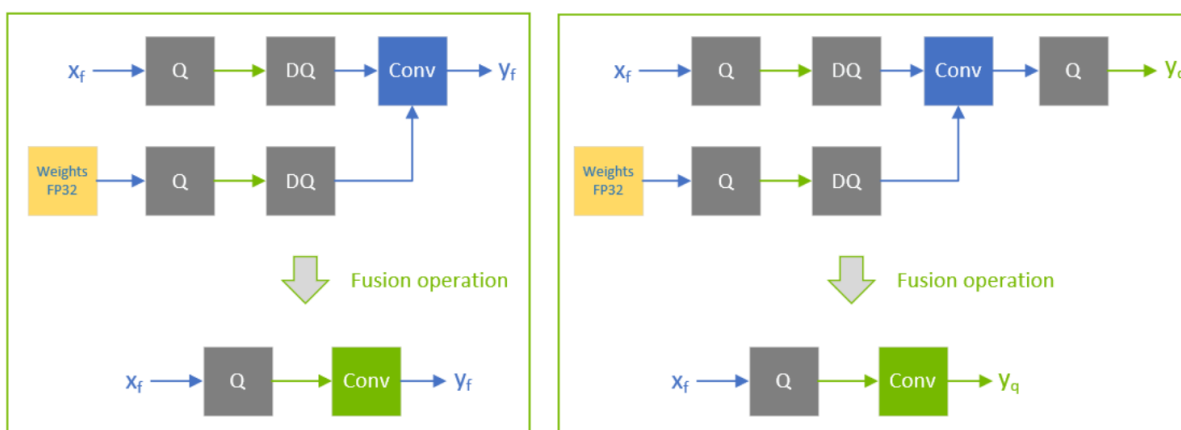
By default, don't quantize the outputs of weighted-operations. It's sometimes useful to preserve the higher-precision dequantized output. For example, if the linear operation is followed by an activation function (SiLU, in the following diagram) that requires higher precision input to produce acceptable accuracy.

Figure 4.   Example of a linear operation followed by an activation function.
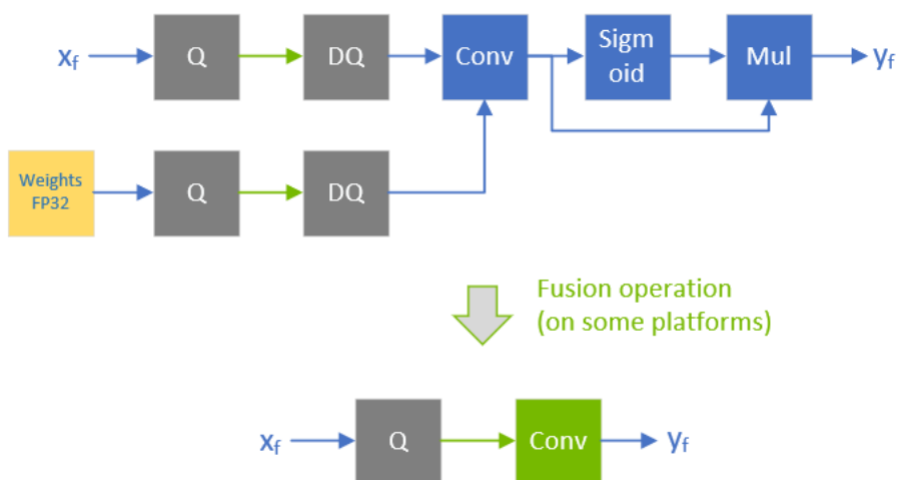
Don't simulate batch-normalization and ReLU fusions in the training framework because TensorRT optimizations guarantee to preserve the arithmetic semantics of these operations.

Figure 5.   Batch normalization is fused with convolution and ReLU while keeping the same execution order as defined in the pre-fusion network. There is no need to simulate BN-folding in the training network.
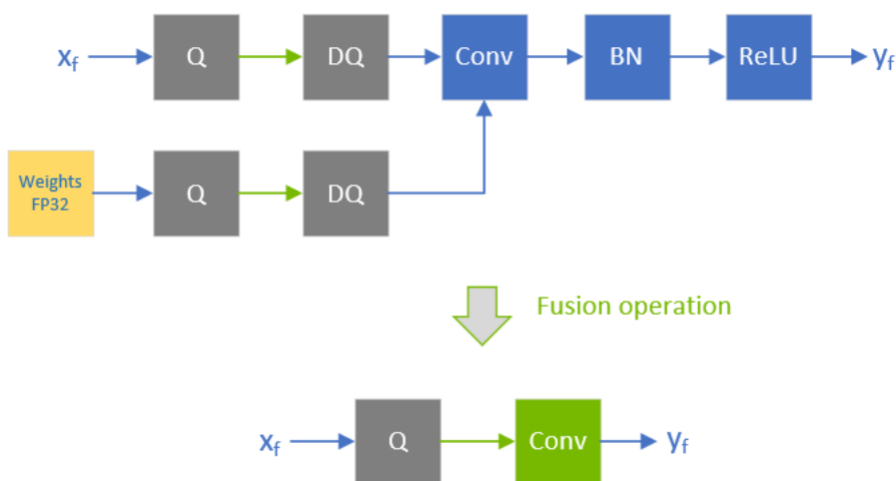
TensorRT can fuse element-wise addition following weighted layers, which is useful for models with skip connections like ResNet and EfficientNet. The precision of the first input to the element-wise addition layer determines the precision of the output of the fusion.

For example, in the following diagram, the precision of xf1 is floating-point, so the output of the fused convolution is limited to floating-point, and the trailing Q-layer cannot be fused with the convolution.

Figure 6.   The precision of x_f^1 is floating-point, so the output of the fused convolution is limited to floating-point, and the trailing Q-layer cannot be fused with the convolution.
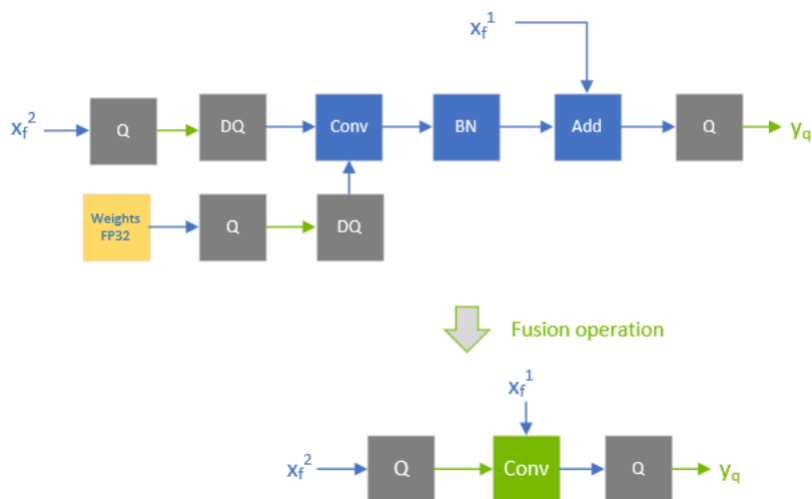
In contrast, when x_f^1 is quantized to INT8, as depicted in the following diagram, the output of the fused convolution is also INT8, and the trailing Q-layer is fused with the convolution.

Figure 7.   When x_f^1 is quantized to INT8, the output of the fused convolution is also INT8, and the trailing Q-layer is fused with the convolution.
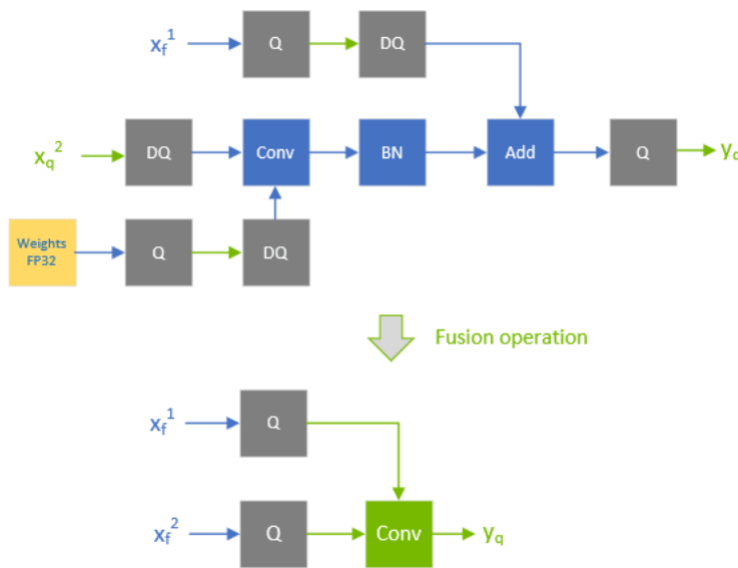
For extra performance, **try quantizing layers that do not commute with Q/DQ**. Currently, non-weighted layers that have INT8 inputs also require INT8 outputs, so quantize both inputs and outputs.

Figure 8.   An example of quantizing a quantizable operation. An element-wise addition is fused with the input DQs and the output Q.
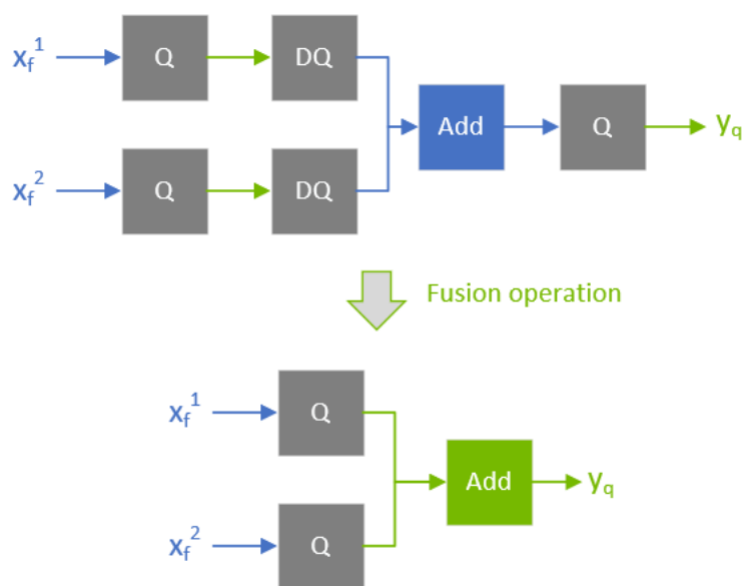
Performance can decrease if TensorRT cannot fuse the operations with the surrounding Q/DQ layers, so **be conservative when adding Q/DQ nodes and experiment with accuracy and TensorRT performance** in mind.

The following figure is an example of suboptimal fusions (the highlighted light green background rectangles) that can result from extra Q/DQ operations. Contrast the following figure with Figure 7, which shows a more performant configuration. The convolution is fused separately from the element-wise addition because each of them is surrounded by Q/DQ pairs. The fusion of the element-wise addition is shown in Figure 8.

Figure 9.   An example of suboptimal quantization fusions: contrast the suboptimal fusion in A and the optimal fusion in B. The extra pair of Q/DQ operations (highlighted with a glowing-green border) forces the separation of the convolution from the element-wise addition.
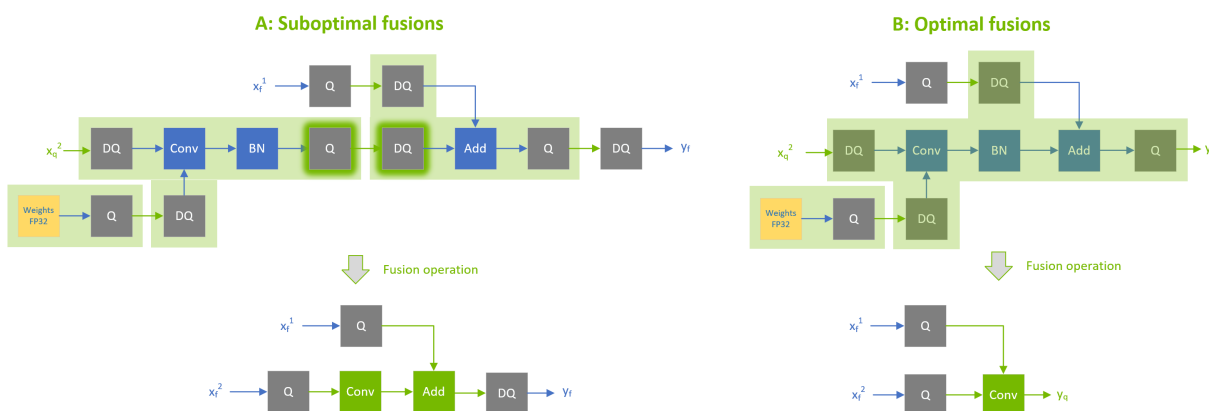
Use per-tensor quantization for activations; and per-channel quantization for weights.This configuration has been demonstrated empirically to lead to the best quantization accuracy.

You can further optimize engine latency by enabling FP16. TensorRT attempts to use FP16 instead of FP32 whenever possible (this is not currently supported for all layer types).

7.4.5.   Q/DQ Limitations

A few of the Q/DQ graph-rewrite optimizations that TensorRT performs compare the values of quantization scales between two or more Q/DQ layers and only perform the graph-rewrite if the compared quantization scales are equal. When a refittable TensorRT engine is refitted, the scales of Q/DQ nodes can be assigned new values. During the refitting operation of Q/DQ engines, TensorRT checks if Q/DQ layers that participated in scale-dependent optimizations are assigned new values that break the rewrite optimizations and throws an exception if true.

Figure 10.   *An example showing scales of Q1 and Q2 are compared for equality, and if equal, they are allowed to propagate backward. If the engine is refitted with new values for Q1 and Q2 such that `Q1 != Q2`, then an exception aborts the refitting process.*
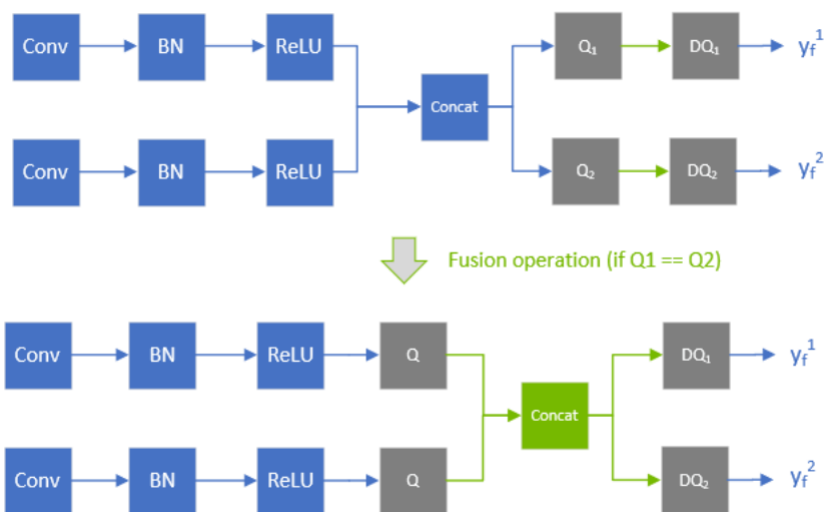
7.4.6.   QAT Networks Using TensorFlow

Currently, there is no TensorFlow quantization toolkit for TensorRT, however, there are several recommended methodologies:

▶ TensorFlow 2 introduced a new API to perform fake-quantization in QAT (Quantization Aware Training): `tf.quantization.quantize_and_dequantize_v2`

This operator uses symmetric quantization which is consistent with TensorRT's quantization scheme. We recommend this API over TensorFlow 1 `tf.quantization.quantize_and_dequantize` API.

When exporting to ONNX, the `quantize_and_dequantize_v2` operator can be converted to a pair of QuantizeLinear and DequantizeLinear operators (Q/DQ operators) by using the [tf2onnx](#) converter. Refer to [quantization_ops_rewriter](#) to understand how this conversion is being performed.

▶ By default, TensorFlow places `tf.quantization.quantize_and_dequantize_v2` operators (Q/DQ nodes after export to ONNX) on operator outputs while TensorRT recommends placing Q/DQ at layer inputs. Refer to [Q/DQ Layer-Placement Recommendations](#) for detailed information.

▶ TensorFlow 1 does not support per-channel quantization (PCQ). PCQ is recommended for weights in order to preserve the accuracy of the model.

7.4.7.   QAT Networks Using PyTorch

PyTorch 1.8.0 and forward support ONNX [QuantizeLinear](#)/[DequantizeLinear](#) which support per channel scales. You can use [pytorch-quantization](#) to do INT8 calibration, run quantization

aware fine-tuning, generate ONNX and finally use TensorRT to run inference on this ONNX model. More detail can be found in PyTorch-Quantization Toolkit User Guide.

7.5.   INT8 Rounding Modes

Backend	Compute Kernel Quantization (FP32 to INT8)	Weights Quantization (FP32 to INT8)	
		Quantized Network (QAT)	Dynamic Range API / Calibration
GPU	`round-to-nearest-with-ties-to-even`	`round-to-nearest-with-ties-to-even`	`round-to-nearest-with-ties-to-positive-infinity`
DLA	`round-to-nearest-with-ties-away-from-zero`	N/A	`round-to-nearest-with-ties-away-from-zero`

Chapter 8. Working With Dynamic Shapes

Dynamic Shapes is 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 will optimize. For more information, refer to <u>Optimization Profiles</u>.

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.

8.1. 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 are 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 (refer to 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 previous example, 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 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).
```

> **Note:** The return value of `setBindingDimensions` for an input only indicates consistency with respect to the optimization profile set for that input. After all input binding dimensions are specified, you can check whether the entire network is consistent with respect to the dynamic input shapes by querying the dimensions of the output bindings of the network.
>
> ```
> nvinfer1::Dims out_dim = context->getBindingDimensions(out_index);
>
> if (out_dim.nbDims == -1) {
> gLogError << "Invalid network output, this might be caused by inconsistent input
>  shapes." << std::endl;
> // abort inference
> }
> ```

8.2.   Optimization Profiles

An *optimization profile* describes a range of dimensions for each network input and the dimensions that the auto-tuner will 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. The shapes defined by the optimization profile must define valid input shapes for the network. Here are the calls for the first profile mentioned previously 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`. Note that each execution context must use a separate optimization profile.

To choose the first optimization profile in the example, use:

C++

call `context.setOptimizationProfileAsync(0, stream)`

where `stream` is the CUDA stream that is used for the subsequent `enqueue()` or `enqueueV2()` invocation in this context.

Python

set `context.set_optimization_profile_async(0, stream)`

If the associated CUDA engine has dynamic inputs, the optimization profile must be set at least once with a unique profile index that 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.

`setOptimizationProfileAsync()` 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 that was formerly used but already released by another execution context with different dynamic input dimensions.

`setOptimizationProfileAsync()` function replaces the now deprecated version of the API `setOptimizationProfile()`. Using `setOptimizationProfile()` to switch

between optimization profiles can cause GPU memory copy operations in the subsequent `enqueue()` or `enqueueV2()` operations operation. To avoid these calls during enqueue, use `setOptimizationProfileAsync()` API instead.

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*.

8.2.1. Bindings For Multiple Optimization Profiles

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 11. 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.

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 warns about the mistake and then chooses the correct binding index from the same column.

For the sake of backward semi-compatibility, the interfaces have an "auto-correct" in the scenario where the binding belongs to the *first* profile, but another profile was specified. In this case, TensorRT warns about the mistake and then chooses the correct binding index from the same column.

8.3. 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 is a four-element 1D tensor containing `{2,3,5,7}`. If the input tensor is a scalar, it has dimensions `[]`, and the output tensor is 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))
```

`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.

8.4. 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.

▶ Layers accepting additional shape inputs (`IResizeLayer`, `IShuffleLayer`, `ISliceLayer`) require that the additional shape inputs be compatible with the dimensions of the minimum and maximum optimization profiles as well as with the dimensions of the runtime data input; otherwise, it can lead to either a build-time or runtime error.

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.

8.5.　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.

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`, `Float`, 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 its ultimate use. If the sum is used to reshape another tensor, then it is a "shape tensor."

8.5.1.　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.

`IActivationLayer` has the signature:

`IActivationLayer: E → E`

since it takes an execution tensor as an input and an execution 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 previous signatures 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
```

The signature for `IShapeLayer` allows all four possible combinations E→E, E→S, S→E, and S→S, so it can be written with two independent variables:

IShapeLayer: $t_1 \rightarrow t_2$

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

```
IAssertionLayer: S →
IConcatenationLayer: t × t × ...→ t
IIfConditionalInputLayer: t → t
IIfConditionalOutputLayer: t → t
IConstantLayer: → t
IActivationLayer: t → t
IElementWiseLayer: t × t → t
IFillLayer: S → t
IFillLayer: S × E × E → E
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: t₁ → t₂
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
IUnaryLayer: t → 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 one use that requires an execution tensor and another use that requires a shape tensor. The output of `IConstantLayer` is classified as both and can 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.

8.6.   Shape Tensor I/O (Advanced)

Sometimes the need arises to use a shape tensor as a network I/O tensor. For example, consider a network consisting solely of an `IShuffleLayer`. TensorRT infers that the second input is a shape tensor. `ITensor::isShapeTensor` returns true for it. Because it is an input shape tensor, TensorRT requires 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 sub-networks 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 `Float` or `Bool`. A workaround for `Bool` is to use `Int32` for the I/O tensor, with zeros and ones, and:

▶ Convert to `Bool` via `ElementWiseOperation::kGREATER`, i.e. x > 0.

▶ Convert from `Bool` via `ISelectLayer`, i.e. y ? 1 : 0.

8.7.   INT8 Calibration With Dynamic Shapes

To run INT8 calibration for a network with dynamic shapes, a 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 are 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 the calibration profile is unset. `getBatchSize()` calibrator method must return `1` when running calibration for a network with dynamic shapes.

> 📝 **Note:** If the calibration optimization profile is not set, the first network optimization profile is used as a calibration optimization profile.

Chapter 9. Extending TensorRT With Custom Layers

NVIDIA TensorRT supports many types of layers and its functionality is continually extended; however, there can be cases in which the layers supported do not cater to the specific needs of a model.

You can extend TensorRT by implementing custom layers, often referred to as plugins.

9.1. Adding Custom Layers Using The C++ API

You can implement a custom layer by deriving from one of TensorRT's plugin base classes.

Table 3.   Base classes, ordered from least expressive to most expressive

	Introduced in TensorRT version?	Mixed input/ output formats/ types	Dynamic shapes?	Supports implicit/explicit batch mode?
`IPluginV2Ext`	5.1	Limited	No	Implicit batch mode only
`IPluginV2IOExt`	6.0.1	General	No	Implicit batch mode only
`IPluginV2DynamicExt`	6.0.1	General	Yes	Explicit batch mode only

In order to use a plugin in a network, you must first register it with TensorRT's `PluginRegistry` (C++, Python). Rather than registering the plugin directly, you register an instance of a factory class for the plugin, derived from `PluginCreator` (C++, Python). The plugin creator class also provides other information about the plugin: its name, version, and plugin field parameters.

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 following table summarizes the base classes, ordered from least expressive to most expressive.

> 💬 **Note:** If a plugin is intended for general use, provide an FP32 implementation in order to allow it to properly operate with any network.

TensorRT provides a macro `REGISTER_TENSORRT_PLUGIN` that statically registers the plugin creator with the registry.

TensorRT library contains plugins that can be loaded into your application. For a list of open-sourced plugins, refer to GitHub: TensorRT plugins.

> 💬 **Note:**
>
> ▶ To use TensorRT plugins in your application, the `libnvinfer_plugin.so` library must be loaded, and all plugins must be registered by calling `initLibNvInferPlugins` in your application code.
>
> ▶ If you have your own plugin library, you can include a similar entry point to register all plugins in the registry under a unique namespace. This ensures there are no plugin name collisions during build time across different plugin libraries.

For more information about these plugins, refer to the NvInferPlugin.h file for reference.

Calling `IPluginCreator::createPlugin()` returns a plugin object of type `IPluginV2`. You can add a plugin to the TensorRT network using addPluginV2() which creates a network layer with the given plugin.

For example, you can add a plugin layer to your network as follows:

```
// Look up the plugin in the registry
auto creator = getPluginRegistry()->getPluginCreator(pluginName, pluginVersion);
const PluginFieldCollection* pluginFC = creator->getFieldNames();
// Populate the fields parameters 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
auto layer = network.addPluginV2(&inputs[0], int(inputs.size()), pluginObj);
… (build rest of the network and serialize engine)
// Destroy the plugin object
pluginObj->destroy()
… (destroy network, engine, builder)
… (free allocated pluginData)
```

> 💬 **Note:**
>
> ▶ `pluginData` must allocate the `PluginField` entries on the heap before passing to `createPlugin`.
>
> ▶ The `createPlugin` method described previously creates a new plugin object on the heap and returns a pointer to it. Ensure you destroy the `pluginObj`, as shown previously, to avoid a memory leak.

During serialization, the TensorRT engine internally stores the plugin type, plugin version, and namespace (if it exists) for all `IPluginV2` type plugins. During deserialization, TensorRT looks up the Plugin Creator from the Plugin Registry and calls

`IPluginCreator::deserializePlugin()`. The plugin object created during deserialization is destroyed internally by the TensorRT engine by calling `IPluginV2::destroy()` method.

During serialization, the TensorRT engine internally stores the plugin type, plugin version, and namespace (if it exists) for all `IPluginV2` type plugins. During deserialization, TensorRT looks up the Plugin Creator from the Plugin Registry and calls `IPluginCreator::deserializePlugin()`. The plugin object created during deserialization is destroyed internally by the TensorRT engine by calling the `IPluginV2::destroy()` method.

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

To support dynamic shapes, your plugin must be derived from `IPluginV2DynamicExt`.

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 reminds 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. You can build the expressions from the expressions for the inputs, using the `IExprBuilder` passed into `getOutputDimensions`. In the example, no new expression has to be built for case 1 because the dimensions of the second output are the same as the dimensions of the first input.

```
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: return 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 inspects 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. This 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 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.

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

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

```
class PoolPlugin : 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: Adding A Custom Layer With Dynamic Shape Support Using C++. In this example, the supported I/O tensor format is linear CHW with FP32, FP16, or INT8 data type, but the I/O tensor must have the same data type.

```
bool PoolPlugin::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::kFLOAT) ||
                  (inOut[pos].type == DataType::kHALF) ||
                  (inOut[pos].type == DataType::kINT8));
    condition &= inOut[pos].type == inOut[0].type;
    return condition;
}
```

> **Important:**
>
> ▶ If INT8 calibration must be used with a network with INT8 I/O plugins, the plugin must support FP32 I/O as it is used by the FP32 calibration graph.
>
> ▶ If the FP32 I/O variant is not supported or INT8 calibration is not used, all required INT8 I/O tensors scales must be set explicitly.
>
> ▶ Calibration can't determine the dynamic range of a plugin's internal tensors. Plugins that operate on quantized data must calculate their own dynamic range for internal tensors.

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

```
void PoolPlugin::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 PoolPlugin::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;
}
```

9.2. Adding Custom Layers Using The Python API

Although the C++ API is the preferred language to implement custom layers, due to 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, refer to Creating a Network Definition in Python.

The same custom layer implementation can be used for both C++ and Python.

9.2.1. 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` that 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():
    builder = trt.Builder(TRT_LOGGER)
    network = builder.create_network()
    config = builder.create_builder_config()
    config.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))
```

9.3.   Using Custom Layers When Importing A Model With A Parser

The ONNX parser automatically attempts to import unrecognized nodes as plugins. If a plugin with the same op_type as the node is found in the plugin registry, the parser forwards the attributes of the node to the plugin creator as plugin field parameters in order to create the plugin. By default, the parser uses "1" as the plugin version and "" as the plugin namespace. This behavior can be overridden by setting a plugin_version and/or plugin_namespace string attribute in the corresponding ONNX node.

In some cases, you might want to modify an ONNX graph prior to importing it into TensorRT. For example, to replace a set of ops with a plugin node. To accomplish this, you can use the ONNX GraphSurgeon utility. For details on how to use ONNX-GraphSurgeon to replace a subgraph, refer to this example.

For more examples, refer to the onnx_packnet sample.

9.4.   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().

9.4.1. Migrating Plugins From TensorRT 6.x Or 7.x To TensorRT 8.x.x

`IPluginV2` and `IPluginV2Ext` are still supported for backward compatibility with TensorRT 5.1 and 6.0.x respectively. However, new plugins should target the `IPluginV2DynamicExt` or `IPluginV2IOExt` interfaces, and old ones refactored to use these interfaces.

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.

Refer to the API description in IPluginV2 API Description for more details about the API.

9.4.2. **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, 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. After the optimal format is selected for a plugin, the plugin is once again configured, then it is 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 must be released.

clone

This is called every time a new builder, network, or engine is created that includes this plugin layer. It must 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 does not replicate the input tensor. There is a single copy that the plugin should share across the batch. If it returns `false`, TensorRT replicates 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.

> **Note:** Plugins based on `IPluginV2` are shared at the engine level, not the execution context level, and thus such plugins which may be used simultaneously by multiple threads need to manage their resources in a thread-safe manner. Plugins based on `IPluginV2Ext` and derivative interfaces are cloned when an `ExecutionContext` is created, so this is not required.

9.4.3.   `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
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.

> **Note:** The data passed to the `createPlugin` function should be allocated by the caller and eventually freed by the caller when the program is destroyed. The ownership of the plugin object returned by the `createPlugin` function is passed to the caller and must be destroyed as well.

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. The plugin object created in this function is destroyed by the TensorRT engine when the engine is destroyed.

set/getPluginNamespace

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

9.5. Best Practices For Custom Layers Plugin

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 10. 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.

10.1. 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 inherits 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 12 depicts the structure of a loop and data flow at the boundary. Loop-invariant tensors can be used inside the loop directly, such as shown for FooLayer.

Figure 12.   A TensorRT loop is set by loop boundary layers. Dataflow can leave the loop only via `ILoopOutputLayer`. The only back edges allowed are the second input to `IRecurrenceLayer`.
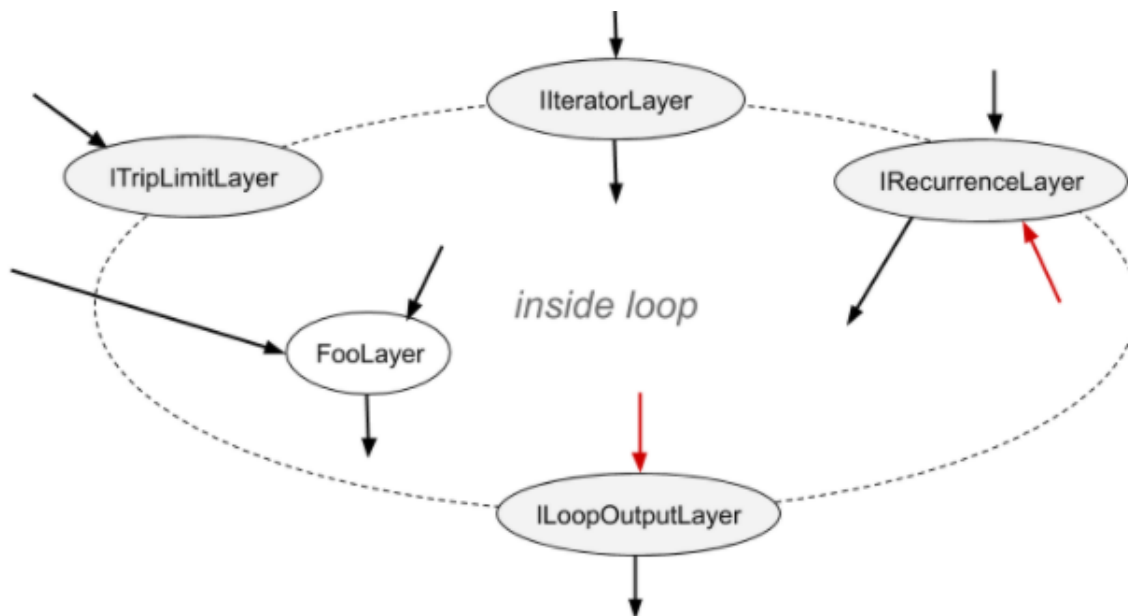
A loop can have multiple `IIteratorLayer`, `IRecurrenceLayer`, and `ILoopOutputLayer`, and at most two `ITripLimitLayers` as explained later. A loop with no `ILoopOutputLayer` has no output and is optimized by TensorRT.

The Layers For Flow-Control Constructs section in the *NVIDIA TensorRT Support Matrix* describes the TensorRT layers which may be used in the loop interior.

Interior layers are free to use tensors defined inside or outside the loop. The interior can contain other loops (refer to Nested Loops) and other conditional constructs (refer to Conditionals Nesting).

To define a loop, first, create an `ILoop` object with the 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 is 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 back edge. If such inputs are removed, the remaining network must be acyclic.

10.2. 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 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 #x0, x1, x2, ...#. 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 the 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 sub-tensors 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.

10.3. Nested Loops

TensorRT infers the nesting of the loops from the data flow. 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.

10.4. Limitations

A loop that refers to more than one dynamic dimension can 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 are 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.

10.5. Replacing **IRNNv2Layer** With Loops

`IRNNv2Layer` was deprecated in TensorRT 7.2.1 and will be removed in TensorRT 9.0. Use the loop API to synthesize a recurrent sub-network. For an example, refer to `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`.

Refer to sampleCharRNN for more information.

Chapter 11. Working With Conditionals

NVIDIA TensorRT supports conditional if-then-else flow control. TensorRT conditionals are used to implement conditional execution of network subgraphs.

11.1. Defining A Conditional

An if-conditional is defined by conditional boundary layers:

▶ `IConditionLayer` represents the predicate and specifies whether the conditional should execute the true-branch (then-branch) or the false-branch (else-branch).

▶ `IIfConditionalInputLayer` specifies an input to one of the two conditional branches.

▶ `IIfConditionalOutputLayer` specifies an output from a conditional.

Each of the boundary layers inherits from class `IIfConditionalBoundaryLayer`, which has a method `getConditional()` for getting its associated `IIfConditional`. The `IIfConditional` instance identifies the conditional. All conditional boundary layers with the same `IIfConditional` belong to that conditional.

A conditional must have exactly one instance of `IConditionLayer`, zero, or more instances of `IIfConditionalInputLayer`, and at least one instance of `IIfConditionalOutputLayer`.

`IIfConditional` implements an if-then-else flow-control construct that provides conditional-execution of a network subgraph based on a dynamic boolean input. It is defined by a boolean scalar predicate `condition`, and two branch subgraphs: a `trueSubgraph` which is executed when `condition` evaluates to `true`, and a `falseSubgraph` which is executed when `condition` evaluates to `false`:

```
If condition is true then:
 output = trueSubgraph(trueInputs);
Else
 output = falseSubgraph(falseInputs);
Emit output
```

Both the true-branch and the false-branch must be defined, similar to the ternary operator in many programming languages.

To define an if-conditional, create an `IIfConditional` instance with the method `INetworkDefinition::addIfConditional`, then add the boundary and branch layers.

```
IIfConditional* simpleIf = network->addIfConditional();
```

The `IIfConditional::setCondition` method takes a single argument: the condition tensor. This 0D boolean tensor (scalar) can be computed dynamically by earlier layers in the network.

It is used to decide which of the branches to execute. An `IConditionLayer` has a single input (the condition) and no outputs since it is used internally by the conditional implementation.

```
// Create a condition predicate that is also a network input.
auto cond = network->addInput("cond", DataType::kBOOL, Dims{0});
IConditionLayer* condition = simpleIf->setCondition(*cond);
```

TensorRT doesn't support a subgraph abstraction for implementing conditional branches and instead uses `IIfConditionalInputLayer` and `IIfConditionalOutputLayer` to define the boundaries of conditionals.

▶ An `IIfConditionalInputLayer`abstracts a single input to one or both of the branch subgraphs of an `IIfConditional`. The output of a specific `IIfConditionalInputLayer`can feed both branches.

```
// Create an if-conditional input.
// x is some arbitrary Network tensor.
IIfConditionalInputLayer* inputX = simpleIf->addInput(*x);
```

Inputs to the then-branch and the else-branch do not need to be of the same type and shape, each branch can independently include zero or more inputs.

`IIfConditionalInputLayer`is optional and is used to control which layers will be part of the branches (refer to Conditional Execution). If all of a branch's outputs do not depend on an `IIfConditionalInputLayer`instance, that branch is empty. An empty else-branch can be useful when there are no layers to evaluate when the condition is `false`, and the network evaluation should proceed following the conditional (refer to Conditional Examples).

▶ An `IIfConditionalOutputLayer`abstracts a single output of the if-conditional. It has two inputs: an output from the true-subgraph (input index 0) and an output from the false-subgraph (input index 1). The output of an `IIfConditionalOutputLayer` can be thought of as a placeholder for the final output which will be determined during runtime.

`IIfConditionalOutputLayer` serves a role similar to that of a Φ (Phi) function node in traditional SSA control-flow graphs. Its semantics are: choose either the output of the true-subgraph or the false-subgraph.

```
// trueSubgraph and falseSubgraph represent network subgraphs
IIfConditionalOutputLayer* outputLayer = simpleIf->addOutput(
    *trueSubgraph->getOutput(0),
    *falseSubgraph->getOutput(0));
```

All outputs of an `IIfConditional` must be sourced at an `IIfConditionalOutputLayer` instance.

An if-conditional without outputs has no effect on the rest of the network, therefore, it is considered ill-formed. Each of the two branches (subgraphs) must also have at least one output. The output of an if-conditional can be marked as the output of the network, unless that if-conditional is nested inside another if-conditional or loop.

The diagram below provides a graphical representation of the abstract model of an if-conditional. The green rectangle represents the interior of the conditional, which is limited to the layer types listed in Layers For Flow-Control Constructs section in the *NVIDIA TensorRT Support Matrix*.

Figure 13.   An if-conditional construct abstract model
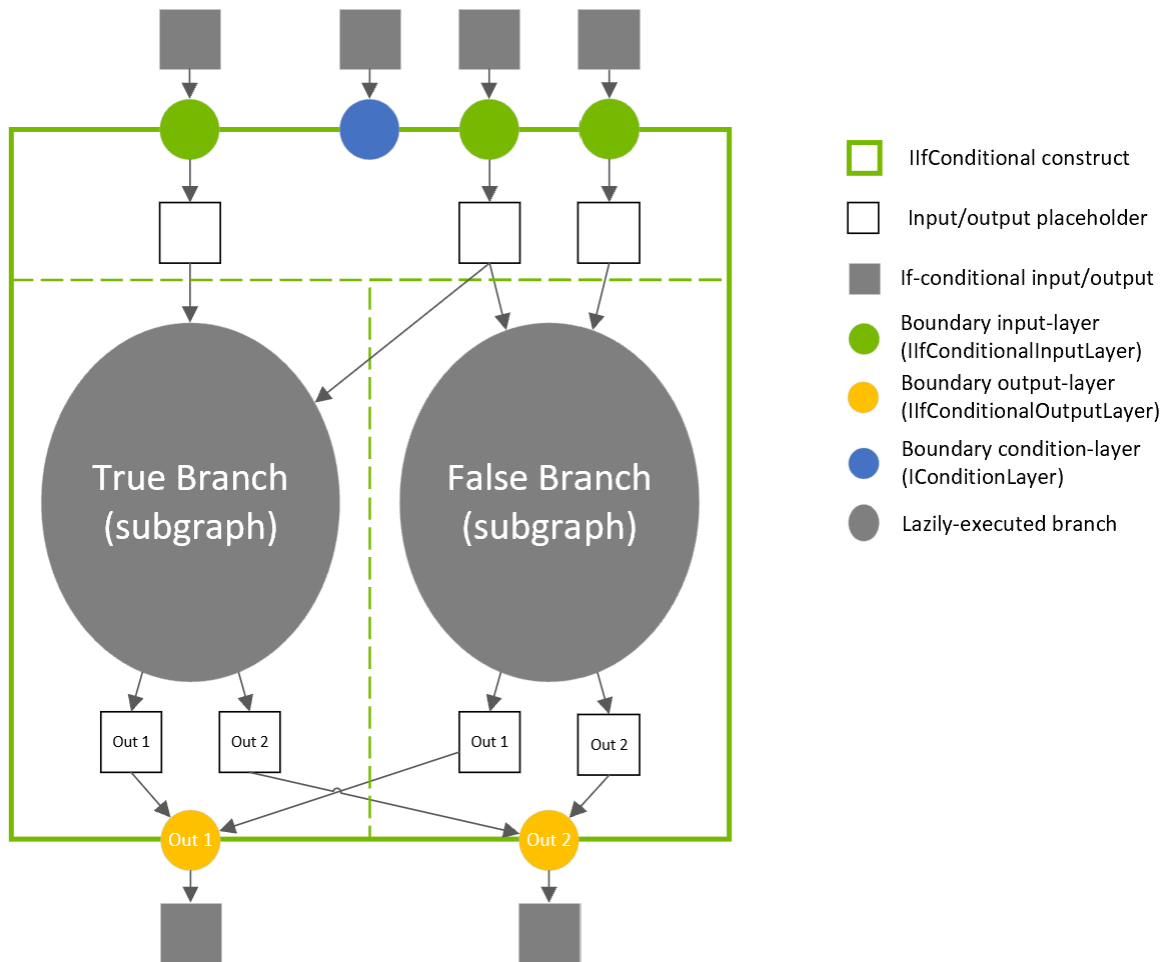
11.2.   Conditional Execution

Conditional-execution of network layers is a network evaluation strategy in which branch-layers (the layers belonging to a conditional subgraph) are executed only if the values of the branch outputs are needed. In conditional-execution, either the true-branch or the false-branch is executed and allowed to change the network state.

In contrast, in predicated-execution, both the true-branch and the false-branch are executed and only one of these is allowed to change the network evaluation state, depending on the value of the condition predicate (i.e. only the outputs of one of the subgraphs is fed into the following layers).

Conditional-execution is sometimes called *lazy evaluation*, and predicated-execution is sometimes referred to as *eager evaluation*.

Instances of `IIfConditionalInputLayer` can be used to specify which layers are invoked eagerly and which are invoked lazily. This is done by tracing the network layers

backwards, starting with each of the conditional outputs. Layers which are data-dependent on the output of at least one `IIfConditionalInputLayer` are considered internal to the conditional and are therefore evaluated lazily. In the extreme case that no instances of `IIfConditionalInputLayer` are added to the conditional, all of the layers are executed eagerly, similarly to `ISelectLayer`.

The three diagrams below depict how the choice of `IIfConditionalInputLayer` placement controls execution scheduling.

Figure 14.   Controlling conditional-execution using
`IIfConditionalInputLayer` **placement**
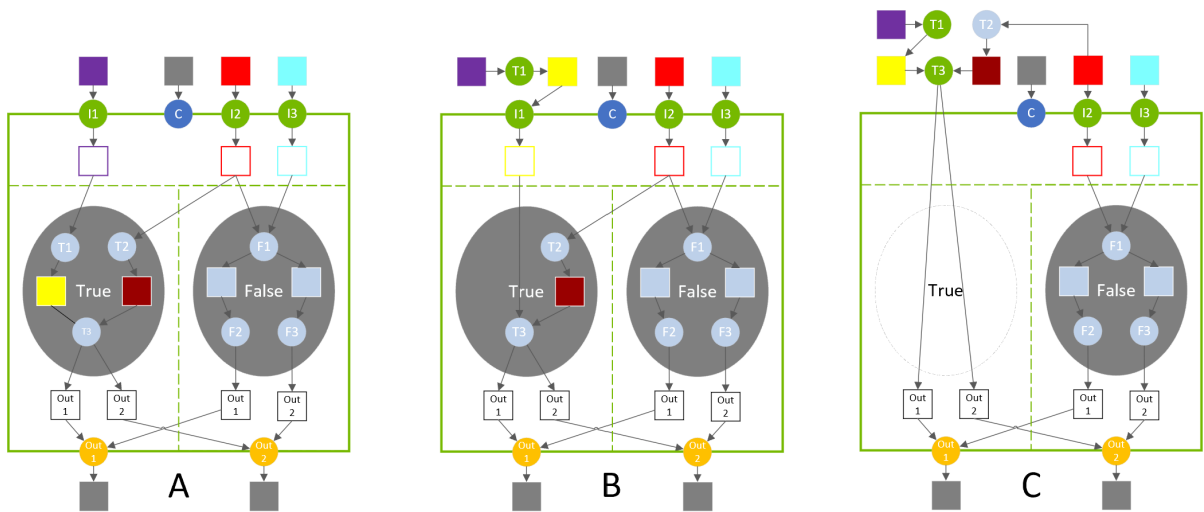
In diagram A, the true-branch is composed of 3 layers (T1, T2, T3). These layers execute lazily when the condition evaluates to `true`.

In diagram B, input-layer I1 is placed after layer T1, which moves T1 out of the true-branch. Layer T1 executes eagerly before evaluating the if-construct.

In diagram C, input-layer I1 is removed altogether which moves T3 outside the conditional. T2's input is reconfigured to create a legal network, and T2 also moves out of the true-branch. When the condition evaluates to `true`, the conditional doesn't compute anything since the outputs have already been eagerly computed (but it does copy the conditional relevant inputs to its outputs).

11.3.   Nesting and Loops

Conditional branches may nest other conditionals and may also nest loops. Loops may nest conditionals. As in loop nesting, TensorRT infers the nesting of the conditionals and loops from the data flow. For example, if conditional B uses a value defined inside loop A, then B is considered to be nested inside of A.

There can be no cross-edges connecting layers in the true-branch to layers in the false-branch, and vice-versa. In other words, the outputs of one branch cannot depend on layers in the other branch.

For example, refer to Conditional Examples for how nesting can be specified.

11.4.   Limitations

The number of output tensors in both true/false subgraph branches must be the same. The type and shape of each output tensor from the branches must be the same.

Note that this is more constrained than the ONNX specification which requires that the true/false subgraphs have the same number of outputs and use the same outputs data-types, but allows for different output shapes.

11.5.   Conditional Examples

11.5.1.   Simple If-Conditional

The following example shows how to implement a simple conditional that conditionally performs an arithmetic operation on two tensors.

Conditional

```
condition = true
If condition is true:
        output = x + y
Else:
        output = x - y
```

Example

```
ITensor* addCondition(INetworkDefinition& n, bool predicate)
{
    // The condition value is a constant int32 input that is cast to boolean because TensorRT
 doesn't support boolean constant layers.

    static const Dims scalarDims = Dims{0, {}};
    static float constexpr zero{0};
    static float constexpr one{1};

    float* const val = predicate ? &one : &zero;

    ITensor* cond =
        n.addConstant(scalarDims, DataType::kINT32, val, 1})->getOutput(0);

    auto* cast = n.addIdentity(cond);
    cast->setOutputType(0, DataType::kBOOL);
    cast->getOutput(0)->setType(DataType::kBOOL);

    return cast->getOutput(0);
}

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

```
INetworkDefinition& n = *builder->createNetworkV2(0U);
auto x = n.addInput("x", DataType::kFLOAT, Dims{1, {5}});
auto y = n.addInput("y", DataType::kFLOAT, Dims{1, {5}});
ITensor* cond = addCondition(n, true);

auto* simpleIf = n.addIfConditional();
simpleIf->setCondition(*cond);

// Add input layers to demarcate entry into true/false branches.
x = simpleIf->addInput(*x)->getOutput(0);
y = simpleIf->addInput(*y)->getOutput(0);

auto* trueSubgraph = n.addElementWise(*x, *y, ElementWiseOperation::kSUM)->getOutput(0);
auto* falseSubgraph = n.addElementWise(*x, *y, ElementWiseOperation::kSUB)->getOutput(0);

auto* output = simpleIf->addOutput(*trueSubgraph, *falseSubgraph)->getOutput(0);
n.markOutput(*output);
```

11.5.2. Exporting from PyTorch

The following example shows how to export scripted PyTorch code to ONNX. The code in function `sum_even` performs an if-conditional nested in a loop.

```python
import torch.onnx
import torch
import tensorrt as trt
import numpy as np

TRT_LOGGER = trt.Logger(trt.Logger.WARNING)
EXPLICIT_BATCH = 1 << (int)(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)

@torch.jit.script
def sum_even(items):
    s = torch.zeros(1, dtype=torch.float)
    for c in items:
        if c % 2 == 0:
            s += c
    return s

class ExampleModel(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, items):
        return sum_even(items)

def build_engine(model_file):
    builder = trt.Builder(TRT_LOGGER)
    network = builder.create_network(EXPLICIT_BATCH)
    config = builder.create_builder_config()
    parser = trt.OnnxParser(network, TRT_LOGGER)

    with open(model_file, 'rb') as model:
        assert parser.parse(model.read())
        return builder.build_engine(network, config)

def export_to_onnx():
    items = torch.zeros(4, dtype=torch.float)
    example = ExampleModel()
    torch.onnx.export(example, (items), "example.onnx", verbose=False, opset_version=13,
 enable_onnx_checker=False, do_constant_folding=True)

export_to_onnx()
build_engine("example.onnx")
```

Chapter 12. 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. DLA does not support Explicit Quantization.

For more information about DLA support in TensorRT layers, refer to DLA Supported Layers. The `trtexec` tool has additional arguments to run networks on DLA, refer to trtexec.

To run the ResNet-50 network on DLA using `trtexec` in FP16 mode, issue:

```
./trtexec --onnx=data/resnet50/ResNet50.onnx --useDLACore=0 --fp16 --allowGPUFallback
```

To run the ResNet-50 network on DLA using `trtexec` in INT8 mode, issue:

```
./trtexec --onnx=data/resnet50/ResNet50.onnx --useDLACore=0 --int8 --allowGPUFallback
```

12.1. 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 FP16 and/or INT8 mode. The `DeviceType` enumeration is used to specify the device that the network or layer executes 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 executes on. If the layer is executing on the GPU, this returns `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 runs 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 is 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, refer to 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 must 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.

12.1.1. Example: sampleMNIST With DLA

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

sampleMNIST located in the GitHub repository demonstrates how to import a trained 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);
```

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=0 [--int8|--fp16]
```

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

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

Procedure

1. Create the builder, builder configuration, and the network:

   ```
   IBuilder* builder = createInferBuilder(gLogger);
   IBuilderConfig* config = builder.createBuilderConfig();
   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 core to execute on:

   ```
   config->setDLACore(0)
   ```

12.2.   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 4096.
- DLA does not support dynamic dimensions. Thus, for wildcard dimensions, the `min`, `max`, and `opt` values of the profile must be equal.
- TensorRT can split a DLA network into multiple sections if any restriction is violated and `GpuFallback` is enabled. Otherwise, TensorRT can emit an error and fallback. For more information, refer to GPU Fallback Mode.
- At most, four DLA loadables can be in use concurrently due to hardware and software memory limitations.

> **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 two spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Each dimension of the kernel size 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]`.
- Operations are not supported if the CBUF size requirement `wtBanksForOneKernel + minDataBanks` exceeds the `numConvBufBankAllotted` limitation 16, where CBUF is the internal convolution cache that stores input weights and activation before operating on them, `wtBanksForOneKernel` is the minimum banks for one kernel to store the minimum weight/kernel elements needed for convolution, and `minDataBanks` is the

minimum banks to store the minimum activation data needed for convolution. The pseudo code details are as follows:

▶ `wtBanksForOneKernel = uint32(ceil(roundUp(inputDims_c * kernelSize_h * kernelSize_w * (INT8 ? 1 : 2), 128) / 32768.0))`

▶ `minDataBanks = uint32(ceil(float(entriesPerDataSlice * dilatedKernelHt) / 256.0)) where entriesPerDataSlice = uint32(ceil(ceil(inputDims_c * (INT8 ? 1 : 2) / 32.0) * inputDims_w / 4.0)) and dilatedKernelHt = (kernelSize_h - 1) * dilation_h + 1`

▶ `FAIL if wtBanksForOneKernel + minDataBanks > 16, PASS otherwise.`

Deconvolution layer

▶ Only two 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`.

▶ TensorRT has disabled deconvolution square kernels and strides in the range `[23 – 32]` on DLA as they significantly slow down compilation.

▶ 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]`.

Pooling layer

▶ Only two 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]`.

▶ With INT8 mode, input and output tensor scales must be the same.

Activation layer

▶ Only two 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]`.

  ▶ `TanH`, `Sigmoid` INT8 support is supported by auto-upgrading to FP16.

Parametric ReLU layer

▶ Slopes input must be a build-time constant.

ElementWise layer

▶ Only two spatial dimension operations are supported.

▶ Both FP16 and INT8 are supported.

▶ Operations supported: `Sum`, `Sub`, `Product`, `Max`, and `Min`.

> 📝 **Note:** On Xavier, 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 Xavier DLA ElementWise layer.

Scale layer

▶ Only two 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 3, 5, 7, or 9.

▶ Normalization region supported is `ACROSS_CHANNELS`.

▶ `LRN` INT8 is supported by auto-upgrading to FP16.

Concatenation layer

▶ DLA supports concatenation only along the channel axis.

▶ Concat must have at least two inputs.

▶ All the inputs must have the same spatial dimensions.

▶ With INT8 mode, the dynamic range of all the inputs must be the same.

▶ With INT8 mode, the dynamic range of output must be equal to each of the inputs.

Resize layer

▶ 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]` in nearest-neighbor mode and `[1, 4]` in bilinear mode.

Unary layer

▶ Only the ABS operation is supported.

Softmax layer

▶ Only supported on NVIDIA Orin™, not Xavier™.

▶ Only supports a single input with batch size of 1.

▶ The non-batch, non-axis dimensions of input should all have size 1. For example, for softmax with axis = 1 (i.e. on C dimension), both the H and W dimensions should be of size 1.

> **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.

12.3. 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 cannot 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, refer to [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 results in an error. However, with `GPUFallbackMode` set to `true`, it continues 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 results in an error if any of the layers can't run on DLA. With `GPUFallbackMode` set to `true`, it reports 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 executes on the GPU.

12.4. I/O Formats on DLA

DLA supports formats that are unique to the device and have constraints on their layout due to vector width byte requirements.

For DLA input, `kDLA_LINEAR`(FP16, INT8), `kDLA_HWC4`(FP16, INT8), `kCHW16`(FP16), and `kCHW32`(INT8) are supported. For DLA output, only `kDLA_LINEAR`(FP16, INT8), `kCHW16`(FP16), and `kCHW32`(INT8) are supported. For `kCHW16` and `kCHW32` formats, if `C` is not an integer multiple, then it must be padded to the next 32-byte boundary.

For `kDLA_LINEAR` format, the stride along the `W` dimension must be padded up to 64 bytes. The memory format is equivalent to a `C` array with dimensions `[N][C][H][roundUp(W, 64/elementSize)]` where `elementSize` is 2 for `FP16` and 1 for `Int8`, with the tensor coordinates `(n, c, h, w)` mapping to array subscript `[n][c][h][w]`.

For `kDLA_HWC4` format, the stride along the `W` dimension must be a multiple of 32 bytes on Xavier and 64 bytes on Orin.

▶ When `C == 1`, TensorRT maps the format to the native grayscale image format.

▶ When `C == 3` or `C == 4`, it maps to the native color image format. If `C == 3`, the stride for stepping along the `W` axis needs to be padded to 4 in elements.

    In this case, the padded channel is located at the 4th-index. Ideally, the padding value doesn't matter because the 4th channel in the weights is padded to zero by the DLA compiler; however, it is safe for the application to allocate a zero-filled buffer of four channels and populate three valid channels.

▶ When `C` is {1, 3, 4}, then padded `C'` is {1, 4, 4} respectively, the memory layout is equivalent to a `C` array with dimensions `[N][H][roundUp(W, 32/C'/elementSize)][C']` where `elementSize` is 2 for `FP16` and 1 for `Int8`. The tensor coordinates `(n, c, h, w)` mapping to array subscript `[n][h][w][c]`, `roundUp` calculates the smallest multiple of `64/elementSize` greater than or equal to `W`.

When using `kDLA_HWC4` as DLA input format, it has the following requirements:

▶ `C` must be 1, 3, or 4

▶ The first layer must be convolution.

▶ The convolution parameters must meet DLA requirements, refer to DLA Supported Layers.

When the `EngineCapability` is `EngineCapability::kSTANDARD` and TensorRT cannot generate a reformat-free network for the given input/output formats, the unsupported DLA formats can be automatically converted into supported DLA format. For example, if the layers connected to the network inputs or outputs cannot run on DLA or if the network does not meet other DLA requirements, reformat operations are inserted to satisfy constraints. In all cases, the strides that TensorRT expects data to be formatted with can be obtained by querying `IExecutionContext::getStrides`.

12.5. DLA Standalone Mode

If you are using a separate DLA runtime component, you can use `EngineCapability::kDLA_STANDALONE` to generate a DLA loadable. See the documentation for the DLA runtime component in question for how to use the loadable.

When using `kDLA_STANDALONE`, TensorRT generates a reformat free network for the given input/output formats. For DLA input, `kLINEAR`(FP16, INT8), `kCHW4`(FP16, INT8), `kCHW16`(FP16), and `kCHW32`(INT8) are supported. While for DLA output, only `kLINEAR`(FP16,

`INT8`), `kCHW16`(`FP16`), and `kCHW32`(`INT8`) are supported. For `kCHW16` and `kCHW32` formats, the `C` channel count is recommended to be equivalent to a positive integer multiple of the vector size. If `C` is not an integer multiple, then it must be padded to the next 32-byte boundary.

12.6. Customizing DLA Memory Pools

You can customize the size of the memory pools allocated to each DLA loadable in a network. There are three types of DLA memory pools (see `enum class MemoryPoolType` for descriptions of each pool):

▶ Managed SRAM

▶ Local DRAM

▶ Global DRAM

For each pool type, use the API `IBuilderConfig::setMemoryPoolLimit` and `IBuilderConfig::getMemoryPoolLimit` to set and query the size of the pool in question to allocate larger memory pools to each DLA loadable. The amount of memory actually required for each loadable may be less than the pool size, in which case the smaller amount will be allocated. The pool size serves only as an upper bound.

Note that all DLA memory pools require sizes that are powers of 2, with a minimum of 4 KiB. Violating this requirement results in a DLA loadable compilation failure.

Managed SRAM is distinct from other DRAM pools mainly due to the difference in role. Here are some aspects of managed SRAM worth noting:

▶ It is similar to a cache in that the resource is scarce and DLA can run without it by falling back to local DRAM.

▶ Any amount allocated tends to be fully exploited. Therefore, the reported SRAM is usually the same as the amount of SRAM pool allocated (which may differ from user-specified size in certain cases).

▶ Due to the cache-like nature, DLA falls back to DRAM upon insufficient SRAM instead of failing. Thus, if affordable, try increasing the amount of SRAM even after a successful engine construction, to see if there is any gain in inference speed. This applies particularly to networks with many subgraphs being offloaded.

▶ There is a difference between Orin and Xavier in the maximum amount of SRAM available per core: Xavier provides 4 MiB total across 4 cores including 2 DLA cores, whereas Orin dedicates 1 MiB SRAM to each DLA core. The implication is that when running multiple networks on a device, Xavier requires an explicit control of aggregate SRAM consumption whereas Orin has less to worry about in that aspect.

In multi-subgraph situations, it is important to keep in mind that the pool sizes apply per DLA subgraph, not for the whole network.

Chapter 13. Performance Best Practices

13.1. Measuring Performance

Before starting any optimization effort with TensorRT, it's essential to determine what should be measured. Without measurements, it's impossible to make reliable progress or measure whether success has been achieved.

Latency

A performance measurement for network inference is how much time elapses from an input being presented to the network until an output is available. This is the *latency* of the network for a single inference. Lower latencies are better. In some applications, low latency is a critical safety requirement. In other applications, latency is directly visible to users as a quality of service issue. For bulk processing, latency may not be important at all.

Throughput

Another performance measurement is how many inferences can be completed in a fixed unit of time. This is the *throughput* of the network. Higher throughput is better. Higher throughputs indicate a more efficient utilization of fixed compute resources. For bulk processing, the total time taken will be determined by the throughput of the network.

Another way of looking at latency and throughput is to fix the maximum latency and measure throughput at that latency. A quality-of-service measurement like this can be a reasonable compromise between the user experience and system efficiency.

Before measuring latency and throughput, you need to choose the exact points at which to start and stop timing. Depending on the network and application, it might make sense to choose different points.

In many applications, there is a processing pipeline, and the overall system performance can be measured by the latency and throughput of the entire processing pipeline. Because the pre- and post-processing steps depend so strongly on the particular application, this section considers the latency and throughput of the network inference only.

13.1.1. Wall-clock Timing

Wall-clock time (the elapsed time between the start of a computation and its end) can be useful for measuring the overall throughput and latency of the application, and for placing inference times in context within a larger system. C++11 provides high precision timers in the `<chrono>` standard library. For example, `std::chrono::system_clock` represents system-wide wall-clock time, and `std::chrono::high_resolution_clock` measures time in the highest precision available.

The following example code snippet shows measuring network inference host time:

```
#include <chrono>

auto startTime = std::chrono::high_resolution_clock::now();
context->enqueueV2(&buffers[0], stream, nullptr);
cudaStreamSynchronize(stream);
auto endTime = std::chrono::high_resolution_clock::now();
float totalTime = std::chrono::duration<float, std::milli>
(endTime - startTime).count();
```

If there is only one inference happening on the device at one time, then this can be a simple way of profiling the time various operations take. Inference is typically asynchronous, so ensure you add an explicit CUDA stream or device synchronization to wait for results to become available.

13.1.2. CUDA Events

One problem with timing on the host exclusively is that it requires host/device synchronization. Optimized applications may have many inferences running in parallel on the device with overlapping data movement. In addition, the synchronization itself adds some amount of noise to timing measurements.

To help with these issues, CUDA provides an Event API. This API allows you to place events into CUDA streams that will be time-stamped by the GPU as they are encountered. Differences in timestamps can then tell you how long different operations took.

The following example code snippet shows computing the time between two CUDA events:

```
cudaEvent_t start, end;
cudaEventCreate(&start);
cudaEventCreate(&end);

cudaEventRecord(start, stream);
context->enqueueV2(&buffers[0], stream, nullptr);
cudaEventRecord(end, stream);

cudaEventSynchronize(end);
float totalTime;
cudaEventElapsedTime(&totalTime, start, end);
```

13.1.3. Built-In TensorRT Profiling

Digging deeper into the performance of inference requires more fine-grained timing measurements within the optimized network.

TensorRT has a *Profiler* (C++, Python) interface, which you can implement in order to have TensorRT pass profiling information to your application. When called, the network will run in a profiling mode. After finishing inference, the profiler object of your class is called to report

the timing for each layer in the network. These timings can be used to locate bottlenecks, compare different versions of a serialized engine, and debug performance issues.

The profiling information can be collected from a regular inference `enqueueV2()` launch or a CUDA graph launch. Refer to `IExecutionContext::setProfiler()` and `IExecutionContext::reportToProfiler()` (C++, Python) for more information.

Layers inside a loop compile into a single monolithic layer, therefore, separate timings for those layers are not available.

An example showing how to use the `IProfiler` interface is provided in the common sample code (`common.h`), and then used in sampleNMT located in the GitHub repository.

You can also use `trtexec` to profile a network with TensorRT given an input network or plan file. Refer to the trtexec section for details.

13.1.4. CUDA Profiling Tools

The recommended CUDA profiler is NVIDIA Nsight™ Systems. Some CUDA developers may be more familiar with nvprof and nvvp, however, these are being deprecated. In any case, these profilers can be used on any CUDA program to report timing information about the kernels launched during execution, data movement between host and device, and CUDA API calls used.

Nsight Systems can be configured in various ways to report timing information for only a portion of the execution of the program or to also report traditional CPU sampling profile information together with GPU information.

Profile only the inference phase

When profiling a TensorRT application, you should enable profiling only after the engine has been built. During the build phase, all possible tactics are tried and timed. Profiling this portion of the execution will not show any meaningful performance measurements and will include all possible kernels, not the ones actually selected for inference. One way to limit the scope of profiling is to:

▶ **First phase:** Structure the application to build and then serialize the engines in one phase.

▶ **Second phase:** Load the serialized engines and run inference in a second phase and profile this second phase only.

If the application cannot serialize the engines, or if the application must run through the two phases consecutively, you can also add `cudaProfilerStart()`/`cudaProfilerStop()` CUDA APIs around the second phase and add `-c cudaProfilerApi` flag to Nsight Systems command to profile only the part between `cudaProfilerStart()` and `cudaProfilerStop()`.

Utilize the NVTX tracing in Nsight Systems

Enabling NVIDIA Tools Extension SDK (NVTX) tracing allows Nsight Compute and Nsight Systems to collect data generated by TensorRT applications. NVTX is a C-based API for marking events and ranges in your applications.

Decoding the kernel names back to layers in the original network can be complicated. Because of this, TensorRT uses NVTX to mark a range for each layer, which then allows the CUDA profilers to correlate each layer with the kernels called to implement it. In TensorRT, NVTX helps to correlate the runtime engine layer execution with CUDA kernel calls. Nsight Systems supports collecting and visualizing these events and ranges on the timeline. Nsight Compute also supports collecting and displaying the state of all active NVTX domains and ranges in a given thread when the application is suspended.

In TensorRT, each layer may launch one or more kernels to perform its operations. The exact kernels launched depend on the optimized network and the hardware present. Depending on the choices of the builder, there may be multiple additional operations that reorder data interspersed with layer computations; these reformat operations may be implemented as either device-to-device memory copies or as custom kernels.

For example, the following screenshots are from Nsight Systems.

Figure 15.   The layer execution and the kernel being launched on the CPU side.
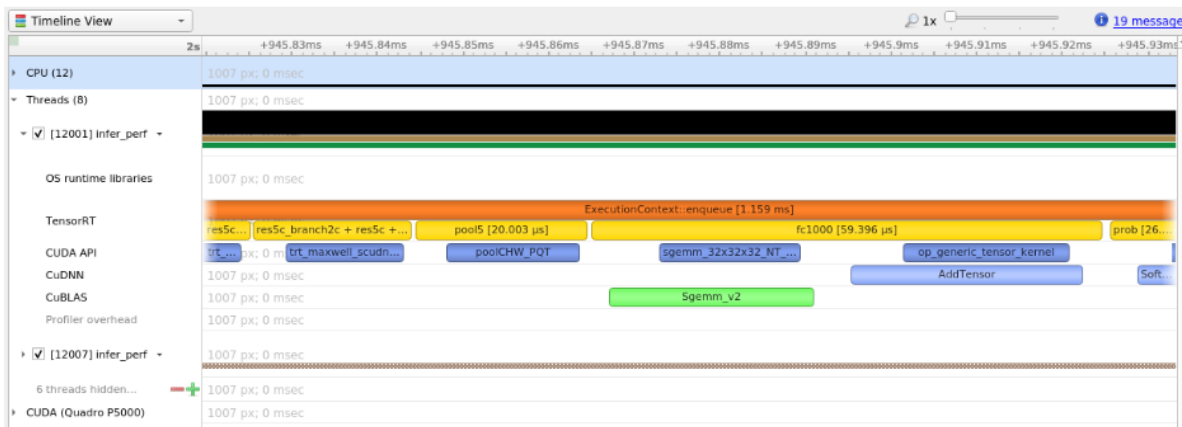
Figure 16.   The kernels actually run on the GPU, in other words, it shows the correlation between the layer execution and kernel launch on the CPU side and their execution on the GPU side.

Control the level of details in NVTX tracing

By default, TensorRT only shows layer names in the NVTX markers, while users can control the level of details by setting the ProfilingVerbosity in the `IBuilderConfig` when the engine is built. For example, to disable NVTX tracing, set the ProfilingVerbosity to `kNONE`:

C++
```
builderConfig->setProfilingVerbosity(ProfilingVerbosity::kNONE);
```
Python
```
builder_config.profilling_verbosity = trt.ProfilingVerbosity.NONE
```

On the other hand, you can choose to allow TensorRT to print more detailed layer information in the NVTX markers, including input and output dimensions, operations, parameters, tactic numbers, etc., by setting the `ProfilingVerbosity` to `kDETAILED`:

C++
```
builderConfig->setProfilingVerbosity(ProfilingVerbosity::kDETAILED);
```
Python
```
builder_config.profilling_verbosity = trt.ProfilingVerbosity.DETAILED
```

Run Nsight Systems with `trtexec`

Below is an example of the commands to gather Nsight Systems profiles using trtexec tool:

```
trtexec --onnx=foo.onnx --profilingVerbosity=detailed --saveEngine=foo.plan
nsys profile -o foo_profile trtexec --loadEngine=foo.plan --warmUp=0 --duration=0 --
iterations=50
```

The first command builds and serializes the engine to `foo.plan`, and the second command runs the inference using `foo.plan` and generates a `foo_profile.qdrep` file which can then be opened in the Nsight Systems GUI interface for visualization.

The `--profilingVerbosity=detailed` flag allows TensorRT to show more detailed layer information in the NVTX marking, and the `--warmUp=0 --duration=0 --iterations=50` flags allow you to control how many inference iterations to run. By default, `trtexec` runs inference for three seconds, which may result in a very large output qdrep file.

13.1.5. Tracking Memory

Tracking memory usage can be as important as execution performance. Usually, the memory will be more constrained on the device than on the host. To keep track of device memory, the recommended mechanism is to create a simple custom GPU allocator that internally keeps some statistics then uses the regular CUDA memory allocation functions `cudaMalloc` and `cudaFree`.

A custom GPU allocator can be set for the builder `IBuilder` for network optimizations, and for `IRuntime` when deserializing engines using the `IGpuAllocator` APIs. One idea for the custom allocator is to keep track of the current amount of memory allocated, and to push an allocation event with a timestamp and other information onto a global list of allocation events. Looking through the list of allocation events allows profiling memory usage over time.

On mobile platforms, GPU memory and CPU memory share the system memory. On devices with very limited memory size, like Nano, system memory might run out with large networks; even the required GPU memory is smaller than system memory. In this case, increasing the system swap size could solve some problems. An example script is:

```
echo "######alloc swap######"
if [ ! -e /swapfile ];then
    sudo fallocate -l 4G /swapfile
    sudo chmod 600 /swapfile
    sudo mkswap /swapfile
    sudo /bin/sh -c 'echo  "/swapfile \t none \t swap \t defaults \t 0 \t 0" >> /etc/fstab'
    sudo swapon -a
fi
```

13.2.   Optimizing TensorRT Performance

The following sections focus on the general inference flow on GPUs and some of the general strategies to improve performance. These ideas are applicable to most CUDA programmers but may not be as obvious to developers coming from other backgrounds.

13.2.1. Batching

The most important optimization is to compute as many results in parallel as possible using batching. In TensorRT, a *batch* is a collection of inputs that can all be processed uniformly. Each instance in the batch has the same shape and flows through the network in exactly the same way. Each instance can, therefore, be trivially computed in parallel.

Each layer of the network will have some amount of overhead and synchronization required to compute forward inference. By computing more results in parallel, this overhead is paid off more efficiently. In addition, many layers are performance-limited by the smallest dimension in the input. If the batch size is one or small, this size can often be the performance limiting dimension. For example, the FullyConnected layer with `V` inputs and `K` outputs can be implemented for one batch instance as a matrix multiply of an `1xV` matrix with a `VxK` weight matrix. If `N` instances are batched, this becomes an `NxV` multiplied by the `VxK` matrix. The vector-matrix multiplier becomes a matrix-matrix multiplier, which is much more efficient.

Larger batch sizes are almost always more efficient on the GPU. Extremely large batches, such as `N > 2^16`, can sometimes require extended index computation and so should be avoided if possible. But generally, increasing the batchsize improves total throughput. In addition, when the network contains MatrixMultiply layers or FullyConnected layers, batch sizes of multiples of 32 tend to have the best performance for FP16 and INT8 inference because of the utilization of Tensor Cores, if the hardware supports them.

Sometimes batching inference work is not possible due to the organization of the application. In some common applications, such as a server that does inference per request, it can be possible to implement opportunistic batching. For each incoming request, wait for a time `T`. If other requests come in during that time, batch them together. Otherwise, continue with a single instance inference. This type of strategy adds fixed latency to each request but can improve the maximum throughput of the system by orders of magnitude.

Using batching

If the explicit batch mode is used when the network is created, then the batch dimension is part of the tensor dimensions, and you can specify the range of the batch sizes and the batch size to optimize the engine for by adding optimization profiles. Refer to the Working With Dynamic Shapes section for more details.

If the implicit batch mode is used when the network is created, the `IExecutionContext::execute` (`IExecutionContext.execute` in Python) and `IExecutionContext::enqueue` (`IExecutionContext.execute_async` in Python) methods take an batch size parameter. The maximum batch size should also be set for the builder when building the optimized network with `IBuilder::setMaxBatchSize` (`Builder.max_batch_size` in Python). When calling `IExecutionContext::execute` or `enqueue`, the bindings passed as the `bindings` parameter are organized per tensor and not per instance. In other words, the data for one input instance is not grouped together into one contiguous region of memory. Instead, each tensor binding is an array of instance data for that tensor.

Another consideration is that building the optimized network optimizes for the given maximum batch size. The final result will be tuned for the maximum batch size but will still work correctly for any smaller batch size. It is possible to run multiple build operations to create multiple optimized engines for different batch sizes, then choose which engine to use based on the actual batch size at runtime.

13.2.2. Streaming

In general, CUDA programming streams are a way of organizing asynchronous work. Asynchronous commands put into a stream are guaranteed to run in sequence but may execute out of order with respect to other streams. In particular, asynchronous commands in two streams may be scheduled to run concurrently (subject to hardware limitations).

In the context of TensorRT and inference, each layer of the optimized final network will require work on the GPU. However, not all layers will be able to fully utilize the computation capabilities of the hardware. Scheduling requests in separate streams allows work to be scheduled immediately as the hardware becomes available without unnecessary synchronization. Even if only some layers can be overlapped, overall performance will improve.

Using streaming

1. Identify the batches of inferences that are independent.
2. Create a single engine for the network.
3. Create a CUDA stream using `cudaStreamCreate` for each independent batch and an `IExecutionContext` for each independent batch.
4. Launch inference work by requesting asynchronous results using `IExecutionContext::enqueue` from the appropriate `IExecutionContext` and passing in the appropriate stream.
5. After all the work has been launched, synchronize with all the streams to wait for results. The execution contexts and streams can be reused for later batches of independent work.

Multiple streams

Running multiple concurrent streams often leads to situations where several streams share compute resources at the same time. This means the network may have less compute resources available during inference than when the TensorRT engine was being optimized. This difference in resource availability can cause TensorRT to choose a kernel that is suboptimal for the actual runtime conditions. In order to mitigate this effect, you can limit the amount of available compute resources during engine creation to more closely resemble actual runtime conditions. This approach generally promotes throughput at the expense of latency. For more information, refer to Limiting Compute Resources.

It is also possible to use multiple host threads with streams. A common pattern is incoming requests dispatched to a pool of waiting for worker threads. In this case, the pool of worker threads will each have one execution context and CUDA stream. Each thread will request work in its own stream as the work becomes available. Each thread will synchronize with its stream to wait for results without blocking other worker threads.

13.2.3. CUDA Graphs

[CUDA graphs](#) are a way to represent a sequence (or more generally a graph) of kernels in a way that allows their scheduling to be optimized by CUDA. This can be particularly useful when your application performance is sensitive to the CPU time taken to enqueue the kernels.

TensorRT's `enqueuev2()` method supports CUDA graph capture for models which require no CPU interaction mid-pipeline. For example:

C++

```
// Capture a CUDA graph instance
cudaGraph_t graph;
cudaGraphExec_t instance;
cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
context->enqueueV2(buffers, stream, nullptr);
cudaStreamEndCapture(stream, &graph);
cudaGraphInstantiate(&instance, graph, NULL, NULL, 0);

// To run inferences:
cudaGraphLaunch(instance, stream);
cudaStreamSynchronize(stream);
```

Models for which graphs are not supported include those with loops or conditionals. In this case, `cudaStreamEndCapture()` will return `cudaErrorStreamCapture*` errors, indicating that the graph capturing has failed, but the context can continue to be used for normal inference without CUDA graphs.

When capturing a graph, it's important to account for the two-phase execution strategy used in the presence of dynamic shapes.

1. Update internal state of the model to account for any changes in input size
2. Stream work to the GPU

For models where input size is fixed at build time, the first phase requires no per-invocation work. Otherwise, if the input sizes have changed since the last invocation, some work may be required to update derived properties.

The first phase of work is not designed to be captured, and even if the capture is successful may increase model execution time. Therefore, after changing the shapes of inputs or the values of shape tensors, call `enqueueV2()` once to flush deferred updates before capturing the graph.

Graphs captured with TensorRT are specific to the input size for which they were captured, and also to the state of the execution context. Modifying the context from which the graph was captured will result in undefined behavior when executing the graph - in particular, if the application is providing its own memory for activations via `createExecutionContextWithoutDeviceMemory()`, the memory address is also captured as part of the graph. Binding locations are also captured as part of the graph.

`trtexec` allows you to check whether the built TensorRT engine is compatible with CUDA graph capture. Refer to the [trtexec](#) section for more information.

13.2.4. Enabling Fusion

13.2.4.1. Layer Fusion

TensorRT attempts to perform many different types of optimizations in a network during the build phase. In the first phase, layers are fused together whenever possible. Fusions transform the network into a simpler form but preserve the same overall behavior. Internally, many layer implementations have extra parameters and options that are not directly accessible when creating the network. Instead, the fusion optimization step detects supported patterns of operations and fuses multiple layers into one layer with internal options set.

Consider the common case of a convolution followed by ReLU activation. To create a network with these operations, it involves adding a Convolution layer with `addConvolution`, following it with an Activation layer using `addActivation` with an `ActivationType` of `kRELU`. The unoptimized graph will contain separate layers for convolution and activation. The internal implementation of convolution supports computing the ReLU function on the output in one step directly from the convolution kernel without requiring a second kernel call. The fusion optimization step will detect the convolution followed by ReLU, verify that the operations are supported by the implementation, then fuse them into one layer.

To investigate which fusions have happened, or have not happened, the builder logs its operations to the logger object provided during construction. Optimization steps are at the `kINFO` log level. To see these messages, ensure you log them in the `ILogger` callback.

Fusions are normally handled by creating a new layer with a name containing the names of both of the layers which were fused. For example, in MNIST, a FullyConnected layer (InnerProduct) named `ip1` is fused with a ReLU Activation layer named `relu1` to create a new layer named `ip1 + relu1`.

13.2.4.2. Types Of Fusions

The following list describes the types of supported fusions.

Supported Layer Fusions

ReLU ReLU Activation

An Activation layer performing ReLU followed by an activation performing ReLU will be replaced by a single activation layer.

Convolution and ReLU Activation

The Convolution layer can be of any type and there are no restrictions on values. The Activation layer must be ReLU type.

Convolution and GELU Activation

The precision of input and output should be the same; with both of them FP16 or INT8. The Activation layer must be GELU type. TensorRT should be running on a Turing or later device with CUDA version 10.0 or later.

Convolution and Clip Activation

The Convolution layer can be any type and there are no restrictions on values. The Activation layer must be Clip type.

Scale and Activation

The Scale layer followed by an Activation layer can be fused into a single Activation layer.

Convolution And ElementWise Operation

A Convolution layer followed by a simple sum, min, or max in an ElementWise layer can be fused into the Convolution layer. The sum must not use broadcasting, unless the broadcasting is across the batch size.

Padding and Convolution/Deconvolution

Padding followed by a Convolution or Deconvolution can be fused into a single Convolution/ Deconvolution layer if all the padding sizes are non-negative.

Shuffle and Reduce

A Shuffle layer without reshape, followed by a Reduce layer can be fused into a single Reduce layer. The Shuffle layer can perform permutations but cannot perform any reshape operation. The Reduce layer must have a `keepDimensions` set of dimensions.

Shuffle and Shuffle

Each Shuffle layer consists of a transpose, a reshape, and a second transpose. A Shuffle layer followed by another Shuffle layer can be replaced by a single Shuffle (or nothing). If both Shuffle layers perform reshape operations, this fusion is only allowed if the second transpose of the first shuffle is the inverse of the first transpose of the second shuffle.

Scale

A Scale layer that adds `0`, multiplied by `1`, or computes powers to the `1` can be erased.

Convolution and Scale

A Convolution layer followed by a Scale layer that is `kUNIFORM` or `kCHANNEL` can be fused into a single convolution by adjusting the convolution weights. This fusion is disabled if the scale has a non-constant `power` parameter.

Reduce

A Reduce layer that performs average pooling will be replaced by a Pooling layer. The Reduce layer must have a `keepDimensions` set, reduced across `H` and `W` dimensions from `CHW` input format before batching, using the `kAVG` operation.

Convolution and Pooling

The Convolution and Pooling layers must have the same precision. The Convolution layer may already have a fused activation operation from a previous fusion.

Depthwise Separable Convolution

A depthwise convolution with activation followed by a convolution with activation may sometimes be fused into a single optimized DepSepConvolution layer. The precision of both convolutions must be INT8 and the device's computes capability must be 7.2 or later.

SoftMax and Log

It can be fused into a single Softmax layer if the SoftMax has not already been fused with a previous log operation.

SoftMax and TopK

Can be fused into a single layer. The SoftMax may or may not include a Log operation.

FullyConnected

The FullyConnected layer will be converted into the Convolution layer, all fusions for convolution will take effect.

Supported Reduction Operation Fusions

GELU

A group of Unary layerand ElementWise layer which represent the following equations can be fused into a single GELU reduction operation.

$$0.5x \times \left(1 + \tanh\left(2 \big/ \pi \left(x + 0.044715 x^3\right)\right)\right)$$

Or the alternative representation:

$$0.5x \times \left(1 + \mathrm{erf}\left(x \big/ \sqrt{2}\right)\right)$$

L1Norm

A Unary layer `kABS` operation followed by a Reduce layer `kSUM` operation can be fused into a single L1Norm reduction operation.

Sum of Squares

A product ElementWise layer with the same input (square operation) followed by a `kSUM` reduction can be fused into a single square Sum reduction operation.

L2Norm

A sum of squares operation followed by a `kSQRT` UnaryOperation can be fused into a single L2Norm reduction operation.

LogSum

A Reduce layer `kSUM` followed by a `kLOG` UnaryOperation can be fused into a single LogSum reduction operation.

LogSumExp

A Unary `kEXP` ElementWise operation followed by a LogSum fusion can be fused into a single LogSumExp reduction.

13.2.4.3. PointWise Fusion

Multiple adjacent PointWise layers can be fused into a single PointWise layer, to improve performance.

The following types of PointWise layers are supported, with some limitations:

`Activation`

Every `ActivationType` is supported.

`Constant`

Only constant with a single value (size == 1).

`ElementWise`

Every `ElementWiseOperation` is supported.

`PointWise`

`PointWise` itself is also a PointWise layer.

`Scale`

Only support `ScaleMode::kUNIFORM`.

`Unary`

Every `UnaryOperation` is supported.

The size of the fused PointWise layer is not unlimited, therefore, some PointWise layers may not be fused.

Fusion creates a new layer with a name consisting of both of the layers which were fused. For example, an ElementWise layer named `add1` is fused with a ReLU Activation layer named `relu1` with a new layer name: `fusedPointwiseNode(add1, relu1)`.

13.2.4.4. Q/DQ Fusion

Quantized INT8 graphs generated from QAT tools like NVIDIA's Quantization Toolkit for PyTorch consists of `onnx::QuantizeLinear` and `onnx::DequantizeLinear` pair of nodes (Q/DQ) with scales and zero-points. Starting in TensorRT 7.0, it's required that `zero_point` is 0.

Q/DQ nodes help convert from FP32 values to INT8 and vice-versa. Such a graph would still have weights and bias in FP32 precision.

Weights are followed by a Q/DQ node pair so that they can be quantized/dequantized if required. Bias quantization is performed using scales from activations and weights, thus no extra Q/DQ node pair is required for bias input. Assumption for bias quantization is that $S\_\text{weights} * S\_\text{input} = S\_\text{bias}$.

Fusions related to Q/DQ nodes include quantizing/dequantizing weights, commutating Q/DQ nodes without changing the mathematical equivalence of the model, and erasing redundant Q/DQ nodes. After applying Q/DQ fusions, the rest of the builder optimizations would be applied to the graph.

Fuse Q/DQ with weighted node (Conv, FC, Deconv)

If we have a
```
[DequantizeLinear (Activations), DequantizeLinear (weights)] > Node >
      QuantizeLinear
```
(`[DQ, DQ] > Node > Q`) sequence, then it is fused to the quantized node (`QNode`).

Supporting Q/DQ node pairs for `weights` requires weighted nodes to support more than one input. Thus we support adding a second input (for weights tensor) and third input (for bias tensor). Additional inputs can be set using `setInput(index, tensor)` API for Convolution, Deconvolution and FullyConnected layers where index = 2 for weights tensor and index = 3 for bias tensor.

During fusion with weighted nodes, we would quantize FP32 weights to INT8 and fuse it with the corresponding weighted node. Similarly, FP32 bias would be quantized to INT32 and fused.

Fuse Q/DQ with non-weighted node

If we have a `DequantizeLinear > Node > QuantizeLinear` (`DQ > Node > Q`) sequence, then it is fused to the quantized node (`QNode`).

Commutate Q/DQ nodes

`DequantizeLinear` commutation is allowed when $\Phi(\text{DQ}(x)) == \text{DQ}(\Phi(x))$.

`QuantizeLinear` commutation is allowed when $Q(\Phi(x)) == \Phi(Q(x))$.

Also, commutation logic also accounts for available kernel implementations such that mathematical equivalence is guaranteed.

Insert missing Q/DQ nodes

If a node has a missing Q/DQ nodes pair, and $\max\left(\mathrm{abs}\left(\Phi\left(x\right)\right)\right) = = \max\left(\mathrm{abs}\left(x\right)\right)$; (for example, MaxPool), missing Q/DQ pairs would be inserted to run more node with INT8 precision.

Erase redundant Q/DQ nodes

It's possible that after applying all the optimizations, the graph still has Q/DQ node pairs which are in itself a no-op. Q/DQ node erasure fusion would remove such redundant pairs.

13.2.5. Limiting Compute Resources

Limiting the amount of compute resources available to TensorRT during engine creation is beneficial when the reduced amount better represents the expected conditions during runtime. For example, when the GPU is expected to be performing additional work in parallel to the TensorRT engine or when the engine is expected to be run on a different GPU with less resources (note that the recommended approach is to build the engine on the GPU that will be used for inference, but this may not always be feasible).

You can limit the amount of available compute resources with the following steps:

1. Start the CUDA MPS control daemon.
   ```
   nvidia-cuda-mps-control -d
   ```
2. Set the amount of compute resources to use with the `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` environment variable. For example, `export CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=50`.
3. Build the network engine.
4. Stop the CUDA MPS control daemon.
   ```
   echo quit | nvidia-cuda-mps-control
   ```

The resulting engine is optimized to the reduced amount of compute cores (50% in this example) and provides better throughput when using similar conditions during inference. You are encouraged to experiment with different amounts of streams and different MPS values to determine the best performance for your network.

For more details about `nvidia-cuda-mps-control`, refer to the nvidia-cuda-mps-control documentation and the relevant GPU requirements here.

13.3. Optimizing Layer Performance

The following descriptions detail how you can optimize the listed layers.

Concatenation Layer

If using an implicit batch dimension, the main consideration with the Concatenation layer is that if multiple outputs are concatenated together, they can not be broadcasted across the batch dimension and must be explicitly copied. Most layers support broadcasting across the batch dimension to avoid copying data unnecessarily, but this will be disabled if the output is concatenated with other tensors.

Gather Layer

To get the maximum performance out of a Gather layer, use an `axis` of `0`. There are no fusions available for a Gather layer.

Reduce Layer

To get the maximum performance out of a Reduce layer, perform the reduction across the last dimensions (tail reduce). This allows optimal memory to read/write patterns through sequential memory locations. If doing common reduction operations, express the reduction in a way that will be fused to a single operation if possible.

RNN Layer

If possible, opt to use the newer RNNv2 interface in preference to the legacy RNN interface. The newer interface supports variable sequence lengths and variable batch sizes, as well as having a more consistent interface. To get maximum performance, larger batch sizes are better. In general, sizes that are multiples of 64 achieve highest performance. Bidirectional RNN-mode prevents wavefront propagation because of the added dependency, therefore, it tends to be slower.

In addition, the newly introduced ILoop-based API provides a much more flexible mechanism to use general layers within recurrence without being limited to a small set of predefined RNNv2 interface. The ILoop recurrence enables a rich set of automatic loop optimizations, including loop fusion, unrolling, and loop-invariant code motion, to name a few. For example, significant performance gains are often obtained when multiple instances of the same MatrixMultiply or FullyConnected layer are properly combined to maximize machine utilization after loop unrolling along the sequence dimension. This works best if you can avoid a MatrixMultiply or FullyConnected layer with a recurrent data dependence along the sequence dimension.

Shuffle

Shuffle operations that are equivalent to identity operations on the underlying data are 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 does not execute additional kernels or memory copies for such operations.

TopK

To get the maximum performance out of a TopK layer, use small values of `K` reducing the last dimension of data to allow optimal sequential memory accesses. Reductions along multiple dimensions at once can be simulated by using a Shuffle layer to reshape the data, then reinterpreting the index values appropriately.

For more information about layers, refer to TensorRT Layers.

13.4.   Optimizing for Tensor Cores

Tensor Core is a key technology to deliver high-performance inference on NVIDIA GPUs. In TensorRT, Tensor Core operations are supported by all compute-intensive layers - MatrixMultiply, FullyConnected, Convolution, and Deconvolution.

Tensor Core layers tend to achieve better performance if the input/output tensor dimensions are aligned to a certain minimum granularity:

▶ In Convolution and Deconvolution layers the alignment requirement is on input/output channel dimension

▶ In MatrixMultiply and FullyConnected layers the alignment requirement is on matrix dimensions `K` and `N` in a MatrixMultiply that is `M x K` times `K x N`

The following table captures the suggested tensor dimension alignment for better Tensor Core performance.

Table 4.   Types of Tensor Cores

Tensor Core Operation Type	Suggested Tensor Dimension Alignment in Elements
TF32	4
FP16	8 for dense math, 16 for sparse math
INT8	32

When using Tensor Core implementations in cases where these requirements are not met, TensorRT implicitly pads the tensors to the nearest multiple of alignment rounding up the dimensions in the model definition instead to allow for extra capacity in the model without increasing computation or memory traffic.

TensorRT always uses the fastest implementation for a layer, and thus in some cases may not use a Tensor Core implementation even if available.

13.5.   Optimizing Plugins

TensorRT provides a mechanism for registering custom plugins that perform layer operations. After a plugin creator is registered, you can look up the registry to find the creator and add the corresponding plugin object to the network during serialization/deserialization.

All TensorRT plugins are automatically registered once the plugin library is loaded. For more information about custom plugins, refer to Extending TensorRT With Custom Layers.

The performance of plugins depends on the CUDA code performing the plugin operation. Standard CUDA best practices apply. When developing plugins, it can be helpful to start with simple standalone CUDA applications that perform the plugin operation and verify correctness. The plugin program can then be extended with performance measurements, more unit testing, and alternate implementations. After the code is working and optimized, it can be integrated as a plugin into TensorRT.

To get the best performance possible, it is important to support as many formats as possible in the plugin. This removes the need for internal reformat operations during the execution of the network. Refer to the Extending TensorRT With Custom Layers section for examples.

13.6.   Optimizing Python Performance

When using the Python API, most of the same performance considerations apply. When building engines, the builder optimization phase will normally be the performance bottleneck; not API calls to construct the network. Inference time should be nearly identical between the Python API and C++ API.

Setting up the input buffers in the Python API involves using `pycuda` or another CUDA Python library, like `cupy`, to transfer the data from the host to device memory. The details of how this

works will depend on where the host data is coming from. Internally, `pycuda` supports the Python Buffer Protocol which allows efficient access to memory regions. This means that if the input data is available in a suitable format in `numpy` arrays or another type that also has support for the buffer protocol, this allows efficient access and transfer to the GPU. For even better performance, ensure that you allocate a page-locked buffer using `pycuda` and write your final preprocessed input there.

For more information about using the Python API, refer to The Python API.

13.7.   Improving Model Accuracy

TensorRT can execute a layer in FP32, FP16, or INT8 precision depending on the builder configuration. By default, TensorRT chooses to run a layer in a precision which results in optimal performance. Sometimes this can result in poor accuracy. Generally, running a layer in higher precision helps improve accuracy with some performance hit.

There are several steps we can take to improve model accuracy:

1. Validate layer outputs:

    a). Use Polygraphy to dump layer outputs and verify there are no NaNs or Infs. The `--validate` option can check for NaNs and Infs. Also, we can compare layer outputs with golden values from, for example, ONNX runtime.

    b). For FP16, it is possible that a model might require retraining to ensure that intermediate layer output can be represented in FP16 precision without overflow/underflow.

    c). For INT8, consider recalibrating with a more representative calibration data set. If your model comes from PyTorch, we also provide NVIDIA's Quantization Toolkit for PyTorch for QAT in the framework besides PTQ in TensorRT. You can try both approaches and choose the one with more accuracy.

2. Manipulate layer precision:

    a). Sometimes running a layer in certain precision results in incorrect output. This can be due to inherent layer constraints (for example, LayerNorm output should not be INT8), model constraints (output gets diverged resulting in poor accuracy), or report a TensorRT bug.

    b). You can control layer execution precision and output precision.

    c). An experimental debug precision tool can help automatically find layers to run in high precision.

3. Use an Algorithm Selection and Reproducible Builds to disable flaky tactics:

    a). When accuracy changes between build+run to build+run, it might be due to a selection of a bad tactic for a layer.

    b). Use an algorithm selector to dump tactics from both good and bad runs. Configure the algorithm selector to allow only a subset of tactics (i.e. just allow tactics from a good run etc).

    c). You can use Polygraphy to automate this process.

Accuracy from run-to-run variation should not change; once the engine is built for a specific GPU, it should result in bit accurate outputs in multiple runs. If not, file a TensorRT bug.

Chapter 14. Troubleshooting

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

14.1. 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 can 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: Are engines and calibration tables portable across TensorRT versions?

A: No. Internal implementations and formats are continually optimized and can change between versions. For this reason, engines and calibration tables are not guaranteed to be binary compatible with different versions of TensorRT. Applications must 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::setMemoryPoolLimit()` controls the maximum amount of workspace that can be allocated and prevents 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 is no more than is required, even if the amount set in `IBuilderConfig::setMemoryPoolLimit()` is much higher. Applications should therefore allow the TensorRT builder as much workspace as they can afford; at runtime, TensorRT allocates 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 following example:

```
$ nm -D libnvinfer.so.* | grep tensorrt_version
00000000abcd1234 B tensorrt_version_#_#_#_#
```

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

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

▶ Turn on `VERBOSE` 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 verify if they match what you are expecting.

> 🗨 **Note:** Marking tensors as outputs can inhibit optimizations, and therefore, can change the results.

You can use [Polygraphy](#) to assist you with debugging and diagnosis.

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 can be more performant. Which implementation to use depends on your latency or throughput requirements and your power budget. Since all DLA engines are independent of the GPU and each other, you could also use both implementations at the same time to further increase the throughput of your network.

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

A: Neither INT4 nor INT16 quantization is 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. 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) can 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: Which layers are accelerated by Tensor Cores?

Most math-bound operations will be accelerated with tensor cores - convolution, deconvolution, fully connected, and matrix multiply. In some cases, particularly for small channel counts or small group sizes, another implementation may be faster and be selected instead of a tensor core implementation.

14.2. Understanding Error Messages

If an error is encountered during execution, TensorRT reports an error message that is intended to help in debugging the problem. Some common error messages that can 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 should always be specified with the implicit batch dimension *not* included in the specification.
`Invalid scale mode, nbWeights: <X>`	
`kernel weights has count <X> but <Y> was expected`	
`<NODE> Axis node has op <OP>, expected Const. The axis must be specified as a Const node.`	As indicated by the error message, the axis must be a build-time constant in order for UFF to parse the node correctly.

ONNX Parser Error Messages

The following table captures the common ONNX parser error messages. For more information on specific ONNX node support, refer to the operators support document.

Error Message	Description
`<X> must be an initializer!`	These error messages signify that an ONNX node input tensor is expected to be an initializer in TensorRT. A possible fix is to run constant folding on the model using TensorRT's Polygraphy tool: `polygraphy surgeon sanitize model.onnx --fold-constants --output model_folded.onnx`
`!inputs.at(X).is_weights()`	
`getPluginCreator() could not find Plugin <operator name> version 1`	This is an error stating that the ONNX parser does not have an import function defined for a particular operator, and did not find a corresponding plugin in the loaded registry for the operator.

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`	This error message can occur if the CUDA or NVIDIA driver

	Error Message	Description
	`<code>. Please check cuda installation:` http://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html.	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::setMemoryPoolLimit().`	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 (refer to 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 must contain a null pointer; otherwise, the count must be non-zero, and values must contain a non-null pointer.
	`Builder was created on device different from current device.`	This error message can 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`.

	Error Message	Description
	You can 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.[1] 4. User does not provide any calibration images; emit a warning here.[1]
	`Could not find scales for tensor <X>.`	This error message indicates that a calibration failure occurred with no scaling factors detected. This could be due to no INT8 calibrator or insufficient custom scales for network layers. For more information, refer to sampleINT8 located in the `opensource/ sampleINT8` directory in the GitHub repository to set up calibration correctly.
	`The engine plan file is not compatible with this version of TensorRT, expecting`	This error message can occur if you are running TensorRT

---

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

	Error Message	Description
	`(format|library) version <X> got <Y>, please rebuild.`	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.
	`Using an engine plan file across different models of devices is not recommended and is likely to affect performance or even cause errors.`	This warning message can occur 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. As indicated by the warning, it is highly recommended to use a device of the same model when generating the engine and deploying it to avoid compatibility issues.
	`GPU memory allocation failed during initialization of (tensor|layer): <name> GPU memory`	These error messages can occur if there is insufficient GPU memory available to instantiate a given TensorRT engine. Verify that the GPU has sufficient available memory to contain the required layer weights and activation tensors.
	`Allocation failed during deserialization of weights.`	
	`GPU does not meet the minimum memory requirements to run this engine …`	
	`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 either need to rebuild the engine without FP16 precision inference or upgrade your GPU to a model

	Error Message	Description
		that supports FP16 precision inference.
	`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, refer to TensorRT Layers.

14.3.   Code Analysis Tools

14.3.1.   Compiler Sanitizers

Google sanitizers are a set of code analysis tools.

14.3.1.1. Issues With `dlopen` And Address Sanitizer

There is a known issue with sanitizers, documented here. When using `dlopen` on TensorRT under a sanitizer, there will be reports of memory leaks unless one of two solutions is adopted:

1. Do not call `dlclose` when running under the sanitizers.
2. Pass the flag `RTLD_NODELETE` to `dlopen` when running under sanitizers.

14.3.1.2. Issues With `dlopen` And Thread Sanitizer

The thread sanitizer can list errors when using `dlopen` from multiple threads. In order to suppress this warning, create a file called `tsan.supp` and add the following to the file:

```
race::dlopen
```

When running applications under thread sanitizer, set the environment variable using:

```
export TSAN_OPTIONS="suppressions=tsan.supp"
```

14.3.1.3. Issues With CUDA And Address Sanitizer

The address sanitizer has a known issue with CUDA applications documented here. In order to successfully run CUDA libraries such as TensorRT under the address sanitizer, add the option `protect_shadow_gap=0` to the `ASAN_OPTIONS` environment variable.

On CUDA 11.4, there is a known bug which can trigger mismatched allocation-and-free errors in the address sanitizer. Add `alloc_dealloc_mismatch=0` to `ASAN_OPTIONS` to disable these errors.

14.3.1.4. Issues With Undefined Behavior Sanitizer

UndefinedBehaviorSanitizer (UBSan) reports false positives with the -fvisibility=hidden option as documented here. You must add the `-fno-sanitize=vptr` option to avoid UBSan reporting such false positives.

14.3.2. Valgrind

Valgrindis a framework for dynamic analysis tools which can be used to automatically detect memory management and threading bugs in applications.

Some versions of valgrind and glibc are affected by a bug which causes false memory leaks to be reported when `dlopen` is used, which can generate spurious errors when running a TensorRT application under valgrind's `memcheck` tool. To work around this, add the following to a valgrind suppressions file as documented here:

```
{
    Memory leak errors with dlopen
    Memcheck:Leak
    match-leak-kinds: definite
    ...
    fun:*dlopen*
    ...
}
```

On CUDA 11.4, there is a known bug which can trigger mismatched allocation-and-free errors in valgrind. Add the option `--show-mismatched-frees=no` to the valgrind command line to suppress these errors.

14.3.3. Compute Sanitizer

When running a TensorRT application under compute-sanitizer, `cuGetProcAddress` can fail with error code 500 due to missing functions. This error can be ignored or suppressed with `--report-api-errors no` option. This is due to CUDA backward compatibility checking if a function is usable on the CUDA toolkit/driver combination. The functions are introduced in a later version of CUDA but are not available on the current platform.

Appendix A. Appendix

A.1. TensorRT Layers

In TensorRT, layers represent distinct flavors of mathematical and/or programmatic operations. The following sections describe every layer that TensorRT supports. The minimum workspace required by TensorRT depends on the operators used by the network. A suggested minimum build-time setting is 16 MB. Regardless of the maximum workspace value provided to the builder, TensorRT will allocate at runtime no more than the workspace it requires. To view a list of the specific attributes supported by each layer, refer to the NVIDIA TensorRT API Reference documentation.

TensorRT can optimize performance by fusing layers. For information about how to enable layer fusion optimizations, refer to Types Of Fusions. For information about optimizing individual layer performance, refer to Optimizing Layer Performance.

For details about the types of precision and features supported per layer, refer to the NVIDIA 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

Refer to the C++ class `IActivationLayer` or the Python class `IActivationLayer` for further details.

A.1.2. **IAssertionLayer**

The `IAssertionLayer` causes the builder or runtime to report an error if its input tensor contains any false values.

Layer Description

The layer has a single boolean input tensor and no outputs. The input tensor must be a shape tensor. If the builder can prove that any element is always false at build time, a build-time error is reported. Otherwise, the tensor is evaluated at runtime, and if any element is false, an error is reported.

Asserting equality of input dimensions may help the optimizer. For example, if a network has two inputs whose leading dimensions must be equal, extracting those dimensions with `IGatherLayer` (or `ISliceLayer`), comparing them with `ElementWiseOperation::kEQUAL`, and feeding the output to an `IAssertionLayer` lets TensorRT know the dimension must be equal.

Conditions And Limitations

The input tensor must be a boolean shape tensor.

Refer to the C++ class `IAssertionLayer` or the Python class `IAssertionLayer` for further details.

A.1.3. **IConcatenationLayer**

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

Layer Description

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

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

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

Conditions And Limitations

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

Refer to the C++ class `IConcatenationLayer` or the Python class `IConcatenationLayer` for further details.

A.1.4.   **IConditionLayer**

An `IConditionLayer` represents the condition boundary associated with a Conditional construct. It is created by TensorRT when using `IIfConditional::setCondition`. An if-conditional is defined by conditional boundary layers.

Layer Description

An `IConditionLayer` has exactly one input which accepts a boolean tensor that controls the evaluation flow of the Conditional associated with this `IConditionLayer`.

Conditions And Limitations

The input to `IConditionLayer` must be a boolean scalar (zero-dimensional tensor).

Refer to the C++ class `IConditionLayer` or the Python class `IConditionLayer` for further details.

A.1.5.   **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.

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

A.1.6.   **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 parsers.

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 $\mathbf{k}$, symmetric padding $\mathbf{p}$, stride $\mathbf{s}$, dilation $\mathbf{d}$, and dilated kernel size $\mathbf{t} = \mathbf{1} + \mathbf{d}(\mathbf{k} - \mathbf{1})$, such that height and width are adjusted accordingly as follows:

▶ $\mathbf{b} = [\mathbf{a}_0\, m\, \mathbf{b}_2\, \mathbf{b}_3]$

▶ $\mathbf{b}_2 = \left\lfloor (\mathbf{a}_2 + 2\mathbf{p}_0 - \mathbf{t}_0)/\mathbf{s}_0 \right\rfloor + 1$

▶ $\mathbf{b}_3 = \left\lfloor (\mathbf{a}_3 + 2\mathbf{p}_1 - \mathbf{t}_1)/\mathbf{s}_1 \right\rfloor + 1$

The kernel weights $\mathbf{w}$ and bias weights $\mathbf{x}$ (optional) for the number of groups g are such that:

▶ $\mathbf{w}$ is ordered according to shape $[m\,\mathbf{a}_1/g\,\mathbf{r}_0\,\mathbf{r}_1]$

▶ $\mathbf{x}$ has length m

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

▶ $\mathbf{k}_{i,\,j,\,\mathrm{hh},\,\mathrm{ll}} = \mathbf{w}_{i,\,j,\,h,\,l}$

▶ $\mathrm{hh} = \{0\ \text{if}\ h = 0,\ h + \mathbf{d}_0(h - 1)\,\text{otherwise}\}$

▶ $\mathrm{ll} = \{0\ \text{if}\ l = 0,\ l + \mathbf{d}_1(l - 1)\,\text{otherwise}\}$

and tensor $\mathbf{C}$ the zero-padded copy of $\mathbf{A}$ with dimensions $[\mathbf{a}_0\,\mathbf{a}_1\,\mathbf{a}_2 + \mathbf{p}_0\,\mathbf{a}_3 + \mathbf{p}_1]$, then tensor $\mathbf{B}$ is defined as $\mathbf{B}_{i,\,j,\,k,\,l} = \Sigma\left(\mathbf{C}_{i,\,:,\,k:\mathrm{kk},\,\mathrm{l:ll}} \times \mathbf{K}_{j,\,:,\,:,\,:}\right) + \mathbf{x}_j$ where $\mathrm{kk} = k + \mathbf{t}_0 - 1$ and $\mathrm{ll} = l + \mathbf{t}_1 - 1$.

Layer Description: 3D convolution

Compute a cross-correlation with 3D filters on a 5D tensor $\mathbf{A}$, of dimensions $\mathbf{a}$, to produce a 5D tensor $\mathbf{B}$, of dimensions $\mathbf{b}$. The dimensions of $\mathbf{B}$ depend on the dimensions of $\mathbf{A}$, the number of output maps m, kernel size $\mathbf{k}$, symmetric padding $\mathbf{p}$, stride $\mathbf{s}$, dilation $\mathbf{d}$, and dilated kernel size $\mathbf{t} = \mathbf{1} + \mathbf{d}(\mathbf{k} - \mathbf{1})$, such that height and width are adjusted accordingly as follows:

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

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

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

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

The kernel weights $\mathbf{w}$ and bias weights $\mathbf{x}$ (optional) for the number of groups g, are such that:

▶ $\mathbf{w}$ is ordered according to shape $[m\,\mathbf{a}_1/g\,\mathbf{r}_0\,\mathbf{r}_1\,\mathbf{r}_2]$

▶ $\mathbf{x}$ has length m

Let tensor $\mathbf{K}$ 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:

▶ $\mathbf{k}_{i,\,j,\,\mathrm{dd},\,\mathrm{hh},\,\mathrm{ll}} = \mathbf{w}_{i,\,j,\,d,\,h,\,l}$

▶ $\text{dd} = \{0 \text{ if } d = 0, \; d + \mathbf{d}_0(d\text{-}1) \text{ otherwise}\}$

▶ $\text{hh} = \{0 \text{ if } h = 0, \; h + \mathbf{d}_1(h\text{-}1) \text{ otherwise}\}$

▶ $\text{ll} = \{0 \text{ if } l = 0, \; l + \mathbf{d}_2(l\text{-}1) \text{ otherwise}\}$

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 $\mathbf{B}_{i,\,j,\,d,\,k,\,l} = \Sigma(\mathbf{C}_{i,\,:,\,\text{d:dd},\,\text{k:kk},\,\text{l:ll}} \times \mathbf{K}_{j,\,:,\,:,\,:,\,:}) + \mathbf{x}_j$ where $\text{dd} = d + \mathbf{t}_0\text{-}1$, $\text{kk} = k + \mathbf{t}_1\text{-}1$, and $\text{ll} = l + \mathbf{t}_2\text{-}1$.

Conditions And Limitations

The number of input kernel dimensions determine 2D or 3D. For 2D convolution, input and output may have more than four dimensions; beyond four, 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 five 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 four for both input and output.

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.

Refer to the [C++ class `IConvolutionLayer`](#) or the [Python class `IConvolutionLayer`](#) for further details.

A.1.7. `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 **k**, symmetric padding **p**, stride **s**, dilation **d**, and dilated kernel size $\mathbf{t} = \mathbf{1} + \mathbf{d}(\mathbf{k}\text{-}\mathbf{1})$, such that height and width are adjusted accordingly as follows:

▶ $\mathbf{b} = [\mathbf{a}_0\,m\,\mathbf{b}_2\,\mathbf{b}_3]$

▶ $\mathbf{b}_2 = (\mathbf{a}_2\text{-}1) * \mathbf{s}_0 + t_0 - 2\mathbf{p}_0$

▶ $\mathbf{b}_3 = \left(\mathbf{a}_3 - 1\right) * \mathbf{s}_1 + t_1 - 2\mathbf{p}_1$

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

▶ **w** is ordered according to shape $\left[\mathbf{a}_1 \big/ g\, m\, \mathbf{r}_0\, \mathbf{r}_1\right]$

▶ **x** has length m

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

▶ $\mathbf{k}_{i,\,j,\,\text{hh},\,\text{ll}} = \mathbf{w}_{i,\,j,\,h,\,l}$

▶ $\text{hh} = \left\{0\,\text{if}\,h = 0,\ h + \mathbf{d}_0\!\left(h - 1\right)\text{otherwise}\right\}$

▶ $\text{ll} = \left\{0\,\text{if}\,l = 0,\ l + \mathbf{d}_1\!\left(l - 1\right)\text{otherwise}\right\}$

and tensor **C** the zero-padded copy of **A** with dimensions $\left[\mathbf{a}_0\, \mathbf{a}_1\, \mathbf{a}_2 + \mathbf{p}_0\, \mathbf{a}_3 + \mathbf{p}_1\right]$, then tensor **B** is defined as $\mathbf{B}_{i,\,j,\,k,\,l} = \Sigma_{u,\,v}\!\left(\mathbf{C}_{i,\,j,\,k\text{-}u,\,l\text{-}v}\,\mathbf{K}\right) + \mathbf{x}_j$ where u ranges from 0 to $\min\!\left(\mathbf{t}_0 - 1,\ k\right)$, and v ranges from 0 to $\min\!\left(\mathbf{t}_1 - 1,\ l\right)$.

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 **k**, symmetric padding **p**, stride **s**, dilation **d**, and dilated kernel size $\mathbf{t} = \mathbf{1} + \mathbf{d}\!\left(\mathbf{k} - \mathbf{1}\right)$, such that height and width are adjusted accordingly as follows:

▶ $\mathbf{b} = \left[\mathbf{a}_0\, m\, \mathbf{b}_2\, \mathbf{b}_3\right]$

▶ $\mathbf{b}_2 = \left(\mathbf{a}_2 - 1\right) * \mathbf{s}_0 + t_0 - 2\mathbf{p}_0$

▶ $\mathbf{b}_3 = \left(\mathbf{a}_3 - 1\right) * \mathbf{s}_1 + t_1 - 2\mathbf{p}_1$

▶ $\mathbf{b}_4 = \left(\mathbf{a}_4 - 1\right) * \mathbf{s}_2 + t_2 - 2\mathbf{p}_2$

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

▶ **w** is ordered according to shape $\left[\mathbf{a}_1 \big/ g\, m\, \mathbf{r}_0\, \mathbf{r}_1\, \mathbf{r}_2\right]$

▶ **x** has length m

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

▶ $\mathbf{k}_{i,\,j,\,\text{dd},\,\text{hh},\,\text{ll}} = \mathbf{w}_{i,\,j,\,d,\,h,\,l}$

▶ $\text{dd} = \left\{0\,\text{if}\,d = 0,\ d + \mathbf{d}_0\!\left(d - 1\right)\text{otherwise}\right\}$

▶ $\text{hh} = \left\{0\,\text{if}\,h = 0,\ h + \mathbf{d}_1\!\left(h - 1\right)\text{otherwise}\right\}$

▶ $\text{ll} = \left\{0\,\text{if}\,l = 0,\ l + \mathbf{d}_2\!\left(l - 1\right)\text{otherwise}\right\}$

and tensor **C** the zero-padded copy of **A** with dimensions $\left[\mathbf{a}_0\,\mathbf{a}_1\,\mathbf{a}_2 + \mathbf{p}_0\,\mathbf{a}_3 + \mathbf{p}_1\,\mathbf{a}_4 + \mathbf{p}_2\right]$, then tensor **B** is defined as $\mathbf{B}_{i,j,k,l,m} = \Sigma_{u,v,w}\left(\mathbf{C}_{i,j,\,k\text{-}u,\,l\text{-}v,\,m\text{-}w}\,\mathbf{K}\right) + \mathbf{x}_j$ where u ranges from 0 to $\min\left(\mathbf{t}_0 - 1,\ k\right)$, and v ranges from 0 to $\min\left(\mathbf{t}_1 - 1,\ l\right)$, and w ranges from 0 to $\min\left(\mathbf{t}_2 - 1,\ m\right)$.

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 to 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.

Refer to the C++ class `IDeconvolutionLayer` or the Python class `IDeconvolutionLayer` for further details.

A.1.8. **IDequantizeLayer**

The `IDequantizeLayer` implements dequantization operators.

Layer Description

The `IDequantizeLayer` layer accepts a signed 8-bit integer input tensor, and uses the configured scale and zero-point inputs to dequantize the input according to:

```
output = (input - zeroPt) * scale
```

The first input (`index 0`) is the tensor to be quantized. The second input (`index 1`), and third input (`index 2`), are the scale and zero-point respectively.

Refer to the C++ class `IDequantizeLayer` or the Python class `IDequantizeLayer` for further details.

A.1.9. **IEinsumLayer**

The `IEinsumLayer` implements an Einsum operator.

Layer Description

The `IEinsumLayer` implements a summation over the elements of the inputs along dimensions specified by the equation parameter, based on the Einstein summation convention.

▶ The equation specifies ASCII lower-case letters for each dimension in the inputs in the same order as the dimensions, separated by a comma for each input.

▶ The equation is represented as `term1,term2…->output-term` where each term corresponds to an operand tensor and the characters within the terms correspond to operands dimensions.

▶ The dimensions labeled with the same subscript must match.

- Repeated subscript labels in one input take the diagonal.
- Repeating a label across multiple inputs means that those axes will be multiplied
- Omitting a label from the output means values along those axes will be summed.
- The output subscripts must appear at least once for some input operand and at most once for the output.
- In implicit mode, i.e. if the equation does not contain ->, the indices which appear once in the expression will be part of the output in increasing alphabetical order.
- In explicit mode, the output can be controlled by specifying output subscript labels by adding an arrow (->) followed by subscripts for the output. For example, `ij,jk->ik` is equivalent to `ij,jk`.
- An empty string ("") is valid for scalar operands.
- The equation may contain spaces (SPC- 0x20) between the different elements (subscripts, ellipsis, arrow and comma).

Conditions And Limitations

TensorRT does not support ellipsis, diagonal operations, or more than two inputs for Einsum. All the inputs must be of the same data type and that data type must be `DataType::kFLOAT` or `DataType::kHALF`.

Refer to the C++ class `IEinsumLayer` or the Python class `IEinsumLayer` for further details.

A.1.10. **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. The output tensor has the same number of dimensions as the inputs. For each output dimension, its length is equal to the lengths of the corresponding input dimensions if they match; otherwise, it is equal to the corresponding input dimension that is not one.

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.

The `IElementWiseLayer` layer can accept different input datatypes depending on the operation that is used. The `IElementWiseLayer` supports the following types for each of these ops:

```
| Operation    | DataTypes                  |
| ----------   | -------------------------- |
| kSUM         | kFLOAT, kHALF, kINT8, kINT32 |
| kPROD        | kFLOAT, kHALF, kINT8, kINT32 |
| kMAX         | kFLOAT, kHALF, kINT8, kINT32 |
| kMIN         | kFLOAT, kHALF, kINT8, kINT32 |
| kSUB         | kFLOAT, kHALF, kINT8, kINT32 |
| kDIV         | kFLOAT, kHALF, kINT8, kINT32 |
| kPOW         | kFLOAT, kHALF, kINT8          |
| kFLOOR\_DIV  | kFLOAT, kHALF, kINT8, kINT32 |
| kAND         | kBOOL                        |
| kOR          | kBOOL                        |
| kXOR         | kBOOL                        |
| kEQUAL       | kFLOAT, kHALF, kINT8, kINT32 |
| kGREATER     | kFLOAT, kHALF, kINT8, kINT32 |
| kLESS        | kFLOAT, kHALF, kINT8, kINT32 |
```

Refer to the C++ class `IElementWiseLayer` or the Python class `IElementWiseLayer` for further details.

A.1.10.1. 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:

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

For more information, refer to the *NVIDIA TensorRT API Reference*.

A.1.11. `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 perform 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 a 1D tensor if using static tensor size. When using the dynamic tensor size, the dimensions for alpha and beta should match each mode's requirement.

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

A.1.12. **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 $\mathbf{a} = [\mathbf{a}_0 \ldots \mathbf{a}_{n-1}]$, it is first reshaped into a tensor **A'** of dimensions $\mathbf{a'} = [\mathbf{a}_0 \ldots \mathbf{a}_{n-4} (\mathbf{a}_{n-3} * \mathbf{a}_{n-2} * \mathbf{a}_{n-1})]$ by squeezing the last three dimensions into one dimension.

Then, the layer performs the operation $\mathbf{B'} = \mathbf{WA'} + \mathbf{X}$ where **W** is the weight tensor of dimensions $\mathbf{w} = [(\mathbf{a}_{n-3} * \mathbf{a}_{n-2} * \mathbf{a}_{n-1}) \, k]$, **X** is the bias tensor of dimensions $\mathbf{x} = (k)$ 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 $\mathbf{b'} = [\mathbf{a}_0 \ldots \mathbf{a}_{n-4} \, k]$.

Finally, **B'** is reshaped again into the output tensor **B** of dimensions $\mathbf{b} = [\mathbf{a}_0 \ldots \mathbf{a}_{n-4} \, k \, 1 \, 1]$ by inserting two lower dimensions each of size 1.

In summary, for input tensor **A** of dimensions $\mathbf{a} = [\mathbf{a}_0 \ldots \mathbf{a}_{n-1}]$, the output tensor **B** will have dimensions $\mathbf{b} = [\mathbf{a}_0 \ldots \mathbf{a}_{n-4} \, k \, 1 \, 1]$.

Conditions And Limitations

A must have three dimensions or more.

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

A.1.13. **IGatherLayer**

The `IGatherLayer` implements three variants of the `gather` operation.

Layer Description

Default Gather Mode: The `kDEFAULT` mode of `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 a 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

$$\mathbf{c}_k = \{\mathbf{a}_k \text{ if } k < x, \ \mathbf{b}_{k-x} \text{ if } k \geq x \text{ and } k < x + b, \text{ and } \mathbf{a}_{k-b+1} \text{ otherwise}\}. \text{ In this case, } \mathbf{C}_i = \mathbf{A}_j \text{ where}$$

$$\mathbf{j}_k = \{\mathbf{B}_{X(i)} \text{ if } k = x, \ \mathbf{i}_k \text{ if } k < x, \text{ and } \mathbf{i}_{k-b} \text{ if } k > x\} \text{ and } X(\mathbf{i}) = \mathbf{i}_{x, \dots, x+b-1}.$$

GatherND Mode: The `GatherMode::kND` mode of `IGatherLayer` gathers elements of the data tensor **A** of dimension **a** using indices tensor **B** of dimension **b** to produce output **C** of dimension **c**.

Let us denote the shape of the input tensors as follows:
$\text{shape}(\mathbf{A}) = [f_1, \dots f_a]$, $\text{shape}(\mathbf{B}) = [g_1, \dots g_{b-1}, k]$. Let us denote the number of elementwise dims as **n**. Then, $c = \mathbf{a} + \mathbf{b} - k - 1 - \mathbf{n}$.

1. If $k = \mathbf{a} - \mathbf{n}$

   a). $\text{shape}(\mathbf{C}) = [g_1, \dots g_{b-1}]$

   b). $\mathbf{C}(i_1 \dots i_{n-1}, i_n \dots i_{b-1}) = \mathbf{A}(i_1 \dots i_{n-1}, \mathbf{B}(i_1, \dots i_{b-1}))$

2. If $k < \mathbf{a} - \mathbf{n}$

   a). $\text{shape}(\mathbf{C}) = [g_1, \dots g_{b-1}, f_{n+k+1}, \dots f_{-a}]$

   b). $\mathbf{C}(i_1, \dots i_{n-1}, i_n \dots i_{a-1}, i_a \dots i_{a+b-k-1-n}) = \mathbf{A}(i_1, \dots i_{n-1}, \mathbf{B}(i_1, \dots i_{b-1}), i_b, \dots i_{a+b-k-1-n})$

   c). where $\mathbf{B}(i_1, \dots i_{b-1})$ is a tensor with k dimensions, and $\text{shape}(\mathbf{B}(i_1, \dots i_{b-1})) = [i_n, \dots i_{n+k+1}]$.

3. If $k > \mathbf{a} - \mathbf{n}$

   a). This is a build time error.

GatherElements Mode: The `GatherMode::kELEMENT` mode of `IGatherLayer` gathers elements of the data tensor **A** of dimension **a** on a specified **axis** using indices tensor **B** of dimension **b** to produce output **C** of dimension **c**.

The output **C** has $\text{shape}(\mathbf{C}) = \text{shape}(\mathbf{B})$. The following pseudocode illustrates how **C** is computed.

```
For each element X of B:
Let J denote a sequence for the subscripts of X
Let K = sequence J with element [axis] replaced by X
C[J] = A[K]
```

Conditions And Limitations

▶ The indices tensor **B** must contain only INT32 values.

▶ If there are any invalid indices elements in the indices tensor, then zeros will be stored at the appropriate locations in the output tensor.

▶ If an axis of the data tensor **A** has a dynamic length, using a negative index for it has undefined behavior.

Default Gather Mode: Applicable to implicit batch mode:

▶ 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 parameter `axis` is zero-indexed and starts at the first non-batch dimension of data tensor **A**.

Applicable to explicit batch mode:

▶ The data tensor **A** must contain at least one dimension.

▶ The data tensor **A** must contain at least `axis` + 1 dimensions.

▶ The parameter axis is zero-indexed and starts at the first dimension of data tensor **A**.

▶ `axis` must be larger than or equal to the number of element-wise dimensions set by `IGatherLayer::setNbElementWiseDimensions()`.

▶ The number of ElementWise dimensions can only be set to 0 or 1.

▶ If the number of ElementWise dimensions is set to 1, the behavior will be similar to implicit batch mode. The leading dimension will be treated like batch dimension in implicit batch mode, which is not part of the gather operation. For example, data tensor **A** has dimensions of `[N, C, H, W]`, and indices tensor **B** has the dimensions of `[N, K]`. If `nbElementWiseDimensions` is 1 and the *axis* is set to 1, the dimensions of the result will be `[N, K, H, W]`. If `nbElementWiseDimensions` is 0 and the `axis` is set to 1, the dimensions of the result will be `[N, N, K, H, W]`.

▶ ElementWise dimensions support broadcasts like `IElementWiseLayer`.

GatherND Mode:

▶ The data tensor **A** must contain at least one dimension.

▶ The indices tensor **B** must contain at least one dimension.

▶ The first **n** dimensions of **A** and **B** must be the equal, where **n** is the number of elementwise dimensions.

▶ **n** must be less than both the number of dimensions of **A** and the number of dimensions of **B**.

▶ The innermost dimension of **B** should have a value between 1 and **a–n**, inclusive.

GatherElements Mode:

- ▶ The data tensor **A** must contain at least one dimension.
- ▶ The indices tensor **B** must contain at least one dimension.
- ▶ `axis` must be between `0` and **a**-1, inclusive.

Refer to the C++ class `IGatherLayer` or the Python class `IGatherLayer` 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

Refer to the C++ class `IIdentityLayer` or the Python class `IIdentityLayer` for further details.

A.1.15. **IIfConditionalBoundaryLayer**

`IIfConditionalBoundaryLayer` is the base class for the if-conditional-related layers, specifically `IConditionLayer`, `IIfConditionalOutputLayer`, and `IIfConditionalInputLayer`. An if-conditional is defined by conditional boundary layers.

Layer Description

Class `IIfConditionalBoundaryLayer` defines a virtual method `getConditional()` that returns a pointer to the associated `IIfConditional`.

Conditions And Limitations

None

For more information about the `IIfConditionalBoundaryLayer` , including how conditionals work and their limitations, refer to Working With Conditionals.

A.1.16. **IIfConditionalOutputLayer**

The `IfConditionalOutputLayer` specifies an output from an if-conditional. An if-conditional is defined by conditional boundary layers.

Layer Description

An if-conditional must have one or more outputs, each defined by a unique instance of `IfConditionalOutputLayer`. An `IfConditionalOutputLayer` has exactly two inputs, one from the true-branch (then-branch) and one from the false-branch (else-branch). When an if-conditional executes an `IfConditionalOutputLayer` it chooses which of the two inputs to copy to the output depending on the value of the condition associated with the if-conditional.

Conditions And Limitations

Must have exactly two inputs and one output.

For more information about the `IfConditionalOutputLayer`, including how conditionals work and their limitations, refer to Working With Conditionals.

A.1.17. **IIfConditionalInputLayer**

The `IfConditionalInputLayer` specifies an input to an if-conditional. An if-conditional is defined by conditional boundary layers.

Layer Description

The `IfConditionalInputLayer` is a placeholder for an input to an if-conditional that can be used by either or both of the if-conditional's branches. An `IfConditionalInputLayer` demarcates the input boundary of an if-conditional.

Conditions And Limitations

Must have exactly one input and one output.

For more information about the `IfConditionalOutputLayer`, including how conditionals work and their limitations, refer to Working With Conditionals.

A.1.18. **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 their limitations, refer to Working With Loops.

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

A.1.19. **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 their limitations, refer to Working With Loops.

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

A.1.20. **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 their limitations, refer to Working With Loops.

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

A.1.21. **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 four constant values: w is the size of the cross-channel window over which the normalization will occur, α, β, and k are normalization parameters. This formula shows the operation performed by the layer:

$$\mathbf{B}_I = \frac{A_I}{\left(k + \mathrm{a}A_{j(I)}^2\right)\beta}$$

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

Conditions And Limitations

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

$$W \in \{1,\ 3,\ 5,\ 7,\ 9,\ 11,\ 13,\ 15\}$$
$$\alpha \in [-1 \times 10^{20},\ 1 \times 10^{20}]$$
$$\beta \in [0.01,\ 1 \times 10^5]$$
$$k \in [1 \times 10^{-5},\ 1 \times 10^{10}]$$

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

A.1.22. **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

$\mathbf{C}_i = \{\max(\mathbf{a}_i, \mathbf{b}_i) \text{ if } i < n\text{-}2,\ \mathbf{a}^I_i \text{ if } i = n\text{-}2,\ \text{and } \mathbf{b}^I_i \text{ if } i = n\text{-}1\}$; 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.

Refer to the [C++ class `IMatrixMultiplyLayer`](#) or the [Python class `IMatrixMultiplyLayer`](#) for further details.

A.1.22.1. 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.

```
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");
```

For more information, refer to the [TensorRT API documentation](#).

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.

A.1.23. **`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 ≥ 0` and `y = x·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.

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

A.1.24. **`IPaddingLayer`**

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

> **Note:** `IPaddingLayer` is deprecated in TensorRT 8.2 and will be removed in TensorRT 10.0. Use `ISliceLayer` to pad the tensor, which supports new non-constant, reflects padding mode and clamp, and supports padding output with dynamic shape.

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 $\mathbf{b}_i = \left\{ \mathbf{x}_0 + \mathbf{a}_{n-2} + \mathbf{y}_0 \text{ if } i = n-2; \ \mathbf{x}_1 + \mathbf{a}_{n-1} + \mathbf{y}_1 \text{ if } i = n-1; \ \textbf{and} \ \mathbf{a}_i \text{ otherwise} \right\}$. Accordingly, the values of $\mathbf{B_w}$ are zeros if $\mathbf{w}_{n-2} < \mathbf{x}_0$ or $\mathbf{x}_0 + a_{n-2} \le \mathbf{w}_{n-2}$ or $\mathbf{w}_{n-1} < \mathbf{x}_1$ or $\mathbf{x}_1 + \mathbf{a}_{n-2} \le \mathbf{w}_{n-1}$. 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.

Refer to the C++ class `IPaddingLayer` or the Python class `IPaddingLayer` for further details.

A.1.25. `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. Refer to the IPluginV2 API Description for more details on the API. TensorRT also has support for a Plugin Registry, which is 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

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 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 following fragment 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.

Refer to the C++ class `IPluginV2Layer` or the Python class `IPluginV2Layer` for further details.

A.1.26. **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:

▶ $\mathbf{b} = [\mathbf{a_0}\,\mathbf{a}_1 \dots \mathbf{a}_{n\text{-}3}\,\mathbf{b}_{n\text{-}2}\,\mathbf{b}_{n\text{-}1}]$

▶ $\mathbf{b}_{n\text{-}2} = (\mathbf{a}_{n\text{-}2} + 2\mathbf{p}_0 + \mathbf{r}_0)/\mathbf{s}_0 + 1$

▶ $\mathbf{b}_{n\text{-}1} = (\mathbf{a}_{n\text{-}1} + 2\mathbf{p}_1 + \mathbf{r}_1)/\mathbf{s}_1 + 1$

Let tensor **C** be the zero-padded copy of **A** with dimensions

$[\mathbf{a}_0\,\mathbf{a}_1 \dots \mathbf{a}_{n\text{-}2} + 2\mathbf{p}_0\,\mathbf{a}_{n\text{-}1} + 2\mathbf{p}_1]$ then, $\mathbf{B}_{j\dots\dots kl} = \mathbf{func}(\mathbf{C}_{j\dots kkk\ lll})$ 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:

▶ $\mathbf{b} = [\mathbf{a_0}\,\mathbf{a}_1 \dots \mathbf{a}_{n\text{-}4}\,\mathbf{b}_{n\text{-}3}\,\mathbf{b}_{n\text{-}2}\,\mathbf{b}_{n\text{-}1}]$

▶ $\mathbf{b}_{n\text{-}3} = (\mathbf{a}_{n\text{-}3} + 2\mathbf{p}_0 + \mathbf{r}_0)/\mathbf{s}_0 + 1$

▶ $\mathbf{b}_{n\text{-}2} = \left(\mathbf{a}_{n\text{-}2} + 2\mathbf{p}_1 + \mathbf{r}_1 \right) / \mathbf{s}_1 + 1$

▶ $\mathbf{b}_{n\text{-}1} = \left(\mathbf{a}_{n\text{-}1} + 2\mathbf{p}_2 + \mathbf{r}_2 \right) / \mathbf{s}_2 + 1$

Let tensor **C** be the zero-padded copy of **A** with dimensions

$\left[\mathbf{a}_0\, \mathbf{a}_1 ... \mathbf{a}_{n\text{-}3} + 2\mathbf{p}_0\, \mathbf{a}_{n\text{-}2} + 2\mathbf{p}_1\, \mathbf{a}_{n\text{-}1} + 2\mathbf{p}_2 \right]$ then, $\mathbf{B}_{j.....\text{klm}} = \mathbf{func}\left(\mathbf{C}_{j.... k\text{kk}\, l\text{ll}\, m\text{mm}} \right)$ where $\text{kk} = k + \mathbf{r}_0 - 1$, $\text{ll} = l + \mathbf{r}_1 - 1$, and $\text{mm} = m + \mathbf{r}_2 - 1$.

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

The number of input kernel dimensions determine 2D or 3D. For 2D pooling, input and output tensors should have three or more dimensions. For 3D pooling, input and output tensors should have four or more dimensions.

If `PoolingType::kMAX` is used and the padding size is equal to or greater than the window size on any dimension, the boundaries of the output tensor (corresponding to the input windows that lie completely outside of the input tensor) will be filled with negative infinity (-inf) for FP32/FP16 or -128 for INT8. This may lead to NaNs in the subsequent layers.

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

A.1.27. **`IQuantizeLayer`**

This `IQuantizeLayer` layer implements quantization operators.

Layer Description

The `IQuantizeLayer` layer accepts a floating-point data input tensor and uses the scale and zero-point inputs to quantize the data to an 8-bit signed integer according to:

```
output = clamp(round(input / scale) +  zeroPt)
```

Rounding type is <u>rounding-to-nearest ties-to-even</u>. Clamping is in the range `[-128, 127]`. The first input (`index 0`) is the tensor to be quantized. The second (`index 1`) and third (`index 2`) are the scale and zero-point respectively.

Refer to the <u>C++ class `IQuantizeLayer`</u> or the <u>Python class `IQuantizeLayer`</u> for further details.

A.1.28. `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 $A_{i,0:B_i}$ like in the SoftMax layer.

Conditions And Limitations

None

Refer to the <u>C++ class `IRaggedSoftMaxLayer`</u> or the <u>Python class `IRaggedSoftMaxLayer`</u> for further details.

A.1.29. `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 their limitations, refer to <u>Working With Loops</u>.

Refer to the <u>C++ class `IRecurrenceLayer`</u> or the <u>Python class `IRecurrenceLayer`</u> for further details.

A.1.30. `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

$\mathbf{b}_i = \{1 \text{ if } i \in \mathbf{r}, \text{ and } \mathbf{a}_i \text{ otherwise}\}$; if the dimensions are not kept, then $\mathbf{b}_{j\text{-}m(j)} = \mathbf{a}_j$ where $j \in \mathbf{r}$ and $m(j)$ is the number of reduction indexes in $\mathbf{r}$ less than or equal to j.

With the sequence of indexes $\mathbf{i}$, $\mathbf{B}_i = \mathrm{op}(\mathbf{A}_j)$, where the sequence of indexes $\mathbf{j}$ is such that $\mathbf{j}_k = \{: \text{ if } k \in \mathbf{r}, \text{ and } \mathbf{i}_k \text{ otherwise}\}$.

Conditions And Limitations

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

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:

Table 5.   Empty tensor exceptions

Reduction Operation	kFLOAT and kHALF	kINT32	kINT8
kSUM	0	0	0
kPROD	1	1	1
kMAX	∞	INT_MAX	-128
kMIN	$-\infty$	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.

Refer to the C++ class IReduceLayer or the Python class IReduceLayer for further details.

A.1.31. 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 dimension `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 modes such as Nearest and Linear are supported for the resize operation. The nearest mode resizes innermost `d` dimensions of `N-D` tensors, where $d \epsilon (0, \min(8, N))$ and `N` > 0. The linear mode resizes innermost `d` dimensions of `N-D` tensors, where $d \epsilon (0, \min(3, N))$ and `N` > 0.

The coordinate transformation mapping function, while interpolating, can be configured to align corners, asymmetric and half pixel. When resized to a single pixel, we support using either the coordinate transformation or using the upper left pixel. When resize mode is `Nearest`, we support different rounding modes like half down, half up, round to floor, and round to ceiling.

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.

Refer to the [C++ class `IResizeLayer`](#) or the [Python class `IResizeLayer`](#) for further details.

A.1.32. **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."

> 📝 **Note:** The `IRNNv2Layer` is deprecated in favor of the loop API; however, it is still available for backward compatibility. For more information about the loop API, refer to [sampleCharRNN](#) with the `--Iloop` option.

Layer Description

This layer accepts an input sequence **X**, initial hidden state $\mathbf{H}_0$, and if the cell is a long short-term memory (LSTM) cell, initial cell state $\mathbf{C}_0$, and produces an output **Y** which represents the output of the final RNN "sub-layer" computed across T timesteps (refer 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 **x**, **h**, and **c**, and produces up to two vector outputs **h'** and **c'**, representing the hidden and cell state after the cell operation has been performed.

In the default (unidirectional) configuration, the RNNv2 layer applies **G** as shown in the following diagram:
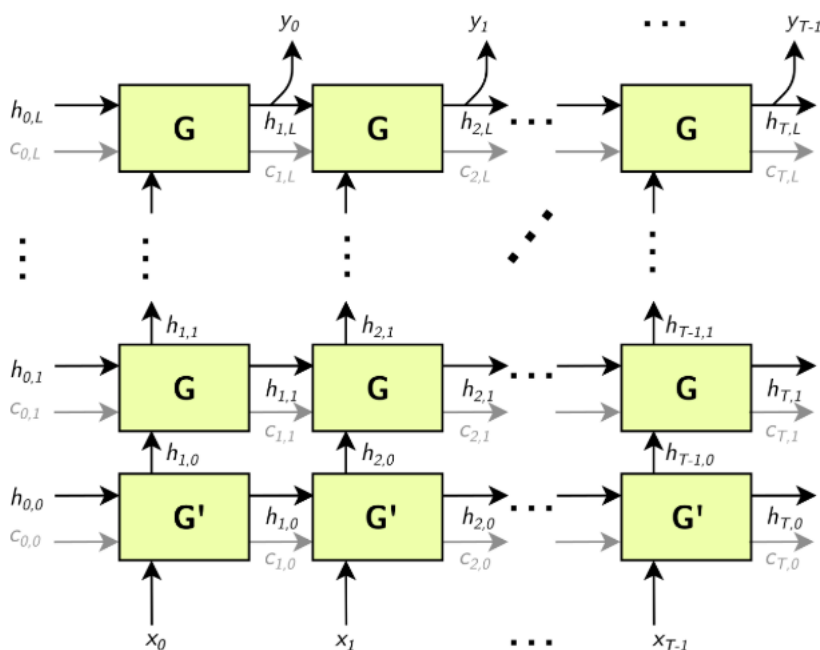
G' is a variant of **G,** .

Arrows leading into boxes are function inputs, and arrows leading away from boxes are function outputs.

- $\mathbf{X} = [\mathbf{x}_0,\ \mathbf{x}_1,\ ...,\ \mathbf{x}_T]$
- $\mathbf{Y} = [\mathbf{y}_0,\ \mathbf{y}_1,\ ...,\ \mathbf{y}_T]$
- $\mathbf{H_i} = [\mathbf{h}_{i,\,0},\ \mathbf{h}_{i,\,1},\ ...,\ \mathbf{h}_{i,\,L}]$
- $\mathbf{C_i} = [\mathbf{c}_{i,\,0},\ \mathbf{c}_{i,\,1},\ ...,\ \mathbf{c}_{i,\,L}]$

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

Bidirectional RNNs (BiRNNs): The RNN can be configured to be bidirectional. In that case, each sub-layer consists of a "forward" layer and a "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:
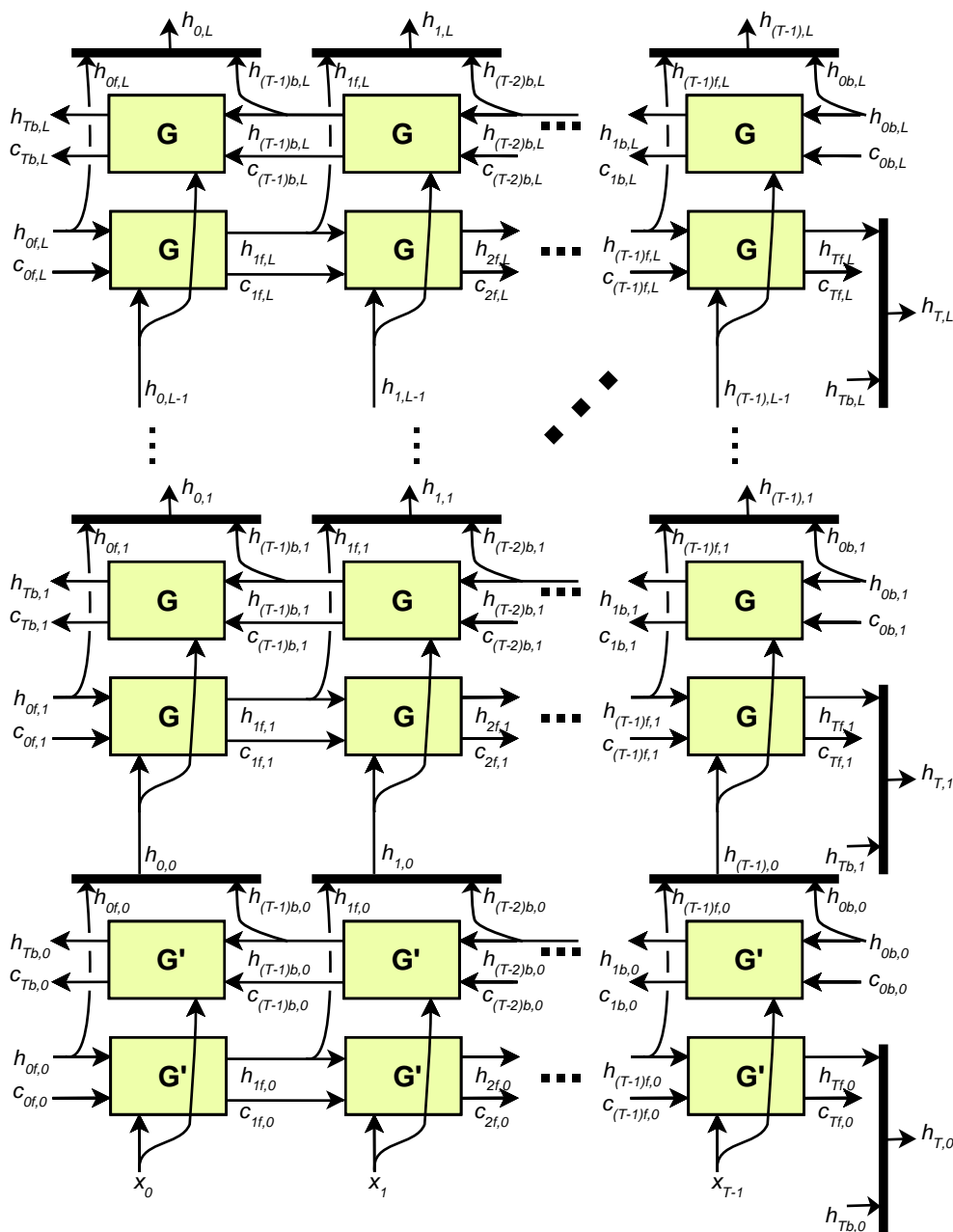
Black bars in the diagram above represent concatenation. The full hidden state $\mathbf{h}_t$ is defined by the concatenation of the forward hidden state $\mathbf{h_{tf}}$ and the backward hidden state $\mathbf{h_{tb}}$:

▶ $\mathbf{h}_{t, i} = \left[h_{\mathrm{tf},i}, \ h_{\mathrm{tb},i} \right]$

▶ $\mathbf{h_t} = \left[\mathbf{h}_{t, 0}, \ \mathbf{h}_{t, 1}, \ ..., \ \mathbf{h}_{t, L} \right]$

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: $G(x, h, c) := \max(W_i x + R_i h + W_b + R_b, \ 0)$** (c not used)

- **tanh: $G(x, h, c) := \tanh(W_i x + R_i h + W_b + R_b)$** (c not used)

- **GRU: $Z := \text{sigmoid}(W_z x + R_z h + W_{bz} + R_{bz})$**

- **GRU: $M := \text{sigmoid}(W_r x + R_r h + W_{br} + R_{br})$**

- **GRU: $G(x, h, c) := \tanh(W_h x + M(h + R_{bh}) + W_{bh})$** (c not used)

- **LSTM: $I := \text{sigmoid}(W_I x + R_I h + W_{bi} + R_{bi})$**

- **LSTM: $F := \text{sigmoid}(W_f x + R_f h + W_{bf} + R_{bf})$**

- **LSTM: $O := \text{sigmoid}(W_o x + R_o h + W_{bo} + R_{bo})$**

- **LSTM: $C := \tanh(W_c x + R_c h + W_{bc} + R_{bc})$**

- **LSTM: $C' := F \times C$**

- **LSTM: $H := O \times \tanh(C')$**

- **LSTM: $G(x, h, c) := \{H, \ C\}$**

For GRU and LSTM, we refer to the intermediate computations for **Z**, **M**, **I**, **F**, for example, 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, refer below). In the bidirectional case, the dimensionality of the **W** matrices is HxE for the first forward/backward layer and $Hx2H$ for subsequent layers.

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

Skip mode: The default mode used by RNNv2 is "linear mode." In this mode, the first sub-layer of the RNNv2 layer uses the cell $G'(x, \ h, \ c)$, which accepts a vector **x** of size E (embedding size), and vectors **h** and **c** of size H (hidden state size), and is defined by the cell operation formula. Subsequent layers use the cell $G(x, \ h, \ c)$, where **x**, **h**, and **c** are all vectors of size H, and are 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 (**H₀** and **C₀**) tensors have at least two non-batch dimensions. Additional dimensions are considered batch dimensions.

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

The data (**Y**) output and final hidden/cell state (**H_T** and **C_T**) tensors have at least two 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));
};
```

Refer to the C++ class `IRNNv2Layer` or the Python class `IRNNv2Layer` for further details.

A.1.32.1. 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, refer to RNNv2 Layer - Optional Inputs.

For more information, refer to the TensorRT API documentation.

A.1.32.2. 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
```

Empty Tensors

`IRNNv2Layer` works for empty tensors but was deprecated in TensorRT 7.2.1 and removed in TensorRT 9.0. Use a loop to synthesize a recurrent sub-network as discussed in the Working With Loops section.

A.1.33. **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

$$B = \left(A * scale + \text{shift}\right)^{power}$$

ScaleMode::kCHANNEL channel-wise transformation

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

ScaleMode::kELEMENTWISE element-wise transformation

$$B_I = \left(A_I * scale_I + shift_I\right)^{power_I}$$

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 three dimensions in implicit batch mode and at least four 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`.

Refer to the C++ class `IScaleLayer` or the Python class `IScaleLayer` for further details.

A.1.34. **IScatterLayer**

The `IScatterLayer` has three input tensors: Data D, Indices I, and Updates U, one output tensor S, and a scatter mode. An optional axis parameter is available when `kELEMENT` mode is used.

Layer Description

Data tensor is of rank $r >= 1$ and has the same shape as the output tensor. Generally, the layer writes to the output tensor either the Data tensor or the Updates tensor depending on the Indices tensor. The interpretation of the indices tensor and mode of operation depends on the scatter mode.

If `mode == kELEMENT`, for each value in updates, its output index is specified by its index in updates for dimension != axis and by the corresponding value in indices for dimension = axis.

```
Pseudocode:
output = data;
for (auto read_idx : Grid(data.extent))
{
 auto write_idx = read_idx[axis];
 write_idx[axis] = indices[read_idx];
 output[write_idx] = updates[read_idx];
}
return output;
```

If `mode == kND`, assume:

- tensor D of rank $r >= 1$ and $\text{shape}(D) = [k_1...k_r]$,
- indices tensor I of rank $q >= 1$ and $\text{shape}(I) = [k_1...k_q]$, and
- updates tensor U of rank $q + r - k_q - 1$

The output of the operation is a tensor G with rank r and $\text{shape}(G) = [k_1...k_r]$.

Mode of operation is defined by the following equations: $G = D$

1. If $k_q = r$, then $G(I(a_1 ... a_{q-1})) = U(a_1 ... a_{q-1})$
2. If $k_q < r$, then $G(I(a_1 ... a_{q-1}), a_{r-K_q} ... a_r) = U(a_1 ... a_{q-1}, a_{r-K_q}, ..., a_r)$

Conditions And Limitations

For both modes:

- repeated indices in I leads to undefined behavior,
- if dynamic shapes are used, using negative indices has undefined behavior,
- only data tensors can have dynamic shapes.

Refer to the C++ class `IScatterLayer` or the Python class `IScatterLayer` for further details.

A.1.35. **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

All three input tensors must have the same number of dimensions; along each axis, each must have the same length or a length of one. If the length is one, the tensor is broadcast along that axis. The output tensor has the dimensions of the inputs after the broadcast rule is applied. The condition tensor is required to be of boolean type. The other two inputs may be FP32, FP16, or INT32.

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

A.1.36. **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. Refer to Execution Tensors vs. Shape Tensors for more information.

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

A.1.37. **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 $c_i = a_{p1(i)}$ 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 $\mathbf{r}^I$, such that $\mathbf{r}_i^I = \left\{\mathbf{r}_i \text{ if } \mathbf{r}_i > 0, \ c_i \text{ if } \mathbf{r}_i = 0, \ \text{inferred if } \mathbf{r}_i = -1\right\}$. Only one dimension can be inferred, such that $\prod \mathbf{r}_i^I = \prod \mathbf{a}_i$.

The special interpretation of $r_i = 0$ as a placeholder, and not the actual dimensions, can be turned off by calling the method `setZeroIsPlaceholder(false)` on the layer. If using dynamic shapes, it is strongly recommended to turn off the placeholder interpretation of `0` because it can interfere with the correct handling of empty tensors and can decrease optimization by TensorRT. For example, consider a tensor **C** with dimensions `[2,x]` that needs to be reshaped to a tensor **R** with dimensions `[x,2]`. With the placeholder interpretation, when `x=0`, the reshape dimensions expand to `[2,2]`, not `[0,2]` as intended.

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. A placeholder `0` or wildcard `-1` occurring in the second input at runtime are interpreted identically to how they are interpreted if they are build-time constants.

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 $\mathbf{b}_i = r_{p2(I)}$ and $\mathbf{B}_{p2(I)} = \mathbf{R}_I$ for a sequence of indexes `I`.

Conditions And Limitations

Product of dimensions $\mathbf{r}^I$ must be equal to the product of input dimensions **a**.

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` are interpreted as placeholders for input dimensions, and the output has 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: $x * (\text{volume of other reshape dimension}) = \text{volume}(\text{input tensor})$

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.

TensorRT reports an error in either case, possibly at build time or run time.

Refer to the C++ class `IShuffleLayer` or the Python class `IShuffleLayer` for further details.

A.1.38. `ISliceLayer`

The `ISliceLayer` implements five variants of the `slice` operations.

Layer Description

Given an input `n`-dimension tensor `A`, the Slice layer generates an output tensor `B` with elements extracted from `A`. Let us denote a, b, s, o, d, f as element coordinates in `A`, element coordinates in `B`, stride, starting offset, the extent of each dimension in `A` and constant value to fill, respectively. The stride can be positive, negative, or zero. The correspondence mapping between `A` and `B` are:

$$B[b_i] = \begin{cases} A[x_i] \text{ in default mode} \\ A[mod(x_i, d_1)] \text{ in warp mode} \\ A[y_i] \text{ where } y_i = \begin{cases} 0, if\ x_i < 0 \\ d_i - 1, if\ x_i \geq d_i \\ x_i, otherwise \end{cases}, in\ clamp\ mode \\ \begin{cases} A[x_i], if\ 0 \leq x_i < d_i \\ f, otherwise \end{cases}, in\ fill\ mode \\ A[z_i], where\ z_i = \begin{cases} 2d_i - 2 - c_i, if\ c_i > d_i \\ c_i \end{cases}, c_i = mod(|x_i|, 2d_i - 2)\ in\ reflect\ mode \end{cases}$$

where $x_i = b_i * s_i + o_i$.

Conditions And Limitations

The corresponding `A` coordinates for every element in `B` must not be out of bounds.

Empty Tensors

The behavior of `ISliceLayer` for empty tensors follows 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 $0 \leq i < size$. 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.

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

A.1.39. **ISoftMaxLayer**

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

Layer Description

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

The SoftMax function **S** for a slice `x` is defined as $S(x) = \exp(x_j) / \sum \exp(x_j)$

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

Conditions And Limitations

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

Refer to the C++ class `ISoftMaxLayer` or the Python class `ISoftMaxLayer` for further details.

A.1.40. **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:

▶ $V_{a0, ..., ai-1, :, ai+1, ..., an-1} = \text{sort}(A_{a0, ..., ai-1, :, ai+1, ..., an-1})_{:K}$

▶ $I_{a0, ..., ai-1, :, ai+1, ..., an-1} = \text{argsort}(A_{a0, ..., ai-1, :, ai+1, ..., an-1})_{:K}$

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.

Refer to the C++ class `ITopKLayer` or the Python class `ITopKLayer` for further details.

A.1.40.1. 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.

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

For more information, refer to the *NVIDIA TensorRT API Reference*.

A.1.41. **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 their limitations, refer to Working With Loops.

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

A.1.42. **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: $B = e^{\wedge}A$

▶ abs: $B = |A|$

▶ log: $B = \ln(A)$

▶ sqrt: $B = \sqrt{A}\left(\text{rounded to nearest even mode}\right)$

- neg : $B = -A$

- recip : $B = 1 / A (\text{reciprocal})$ 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 : $B = \operatorname{asin}(A)$

- Acos : $B = \operatorname{acos}(A)$

- Atan : $B = \tan(A)$

- Asinh : $B = \operatorname{asinh}(A)$

- Acosh : $B = \operatorname{acosh}(A)$

- Atanh : $B = \operatorname{atanh}(A)$

- Ceil : $B = \operatorname{ceil}(A)$

- Floor : $B = \operatorname{floor}(A)$

- ERF : $B = \operatorname{erf}(A)$

- NOT : $B = {\sim} A$

- Sign : $B = $ If $A > 0,\ 1;$ if $A < 0,\ -1;$ if $A == 0,\ 0.$

- Round : $B = $ Round to nearest even (A)

Conditions And Limitations

Input and output can be zero to 7-dimensional tensors.

The `IUnaryLayer` layer can accept different input datatypes depending on the operation that is used. The `IUnaryLayer` supports the following types for each of these ops:

```
| Operation | DataTypes                  |
| --------- | -------------------------- |
| kEXP      | kFLOAT,  kHALF,  kINT8     |
| kLOG      | kFLOAT,  kHALF,  kINT8     |
| kSQRT     | kFLOAT,  kHALF,  kINT8     |
| kRECIP    | kFLOAT,  kHALF,  kINT8     |
| kABS      | kFLOAT,  kHALF,  kINT8     |
| kNEG      | kFLOAT,  kHALF,  kINT8     |
| kSIN      | kFLOAT,  kHALF,  kINT8     |
| kCOS      | kFLOAT,  kHALF,  kINT8     |
| kTAN      | kFLOAT,  kHALF,  kINT8     |
| kSINH     | kFLOAT,  kHALF,  kINT8     |
| kCOSH     | kFLOAT,  kHALF,  kINT8     |
| kASIN     | kFLOAT,  kHALF,  kINT8     |
| kACOS     | kFLOAT,  kHALF,  kINT8     |
| kATAN     | kFLOAT,  kHALF,  kINT8     |
| kASINH    | kFLOAT,  kHALF,  kINT8     |
```

```
| kACOSH    | kFLOAT, kHALF, kINT8          |
| kATANH    | kFLOAT, kHALF, kINT8          |
| kCEIL     | kFLOAT, kHALF, kINT8          |
| kFLOOR    | kFLOAT, kHALF, kINT8          |
| kERF      | kFLOAT, kHALF, kINT8          |
| kNOT      | kBOOL                         |
| kSIGN     | kFLOAT, kHALF, kINT8, kINT32 |
| kROUND    | kFLOAT, kHALF, kINT8          |
```

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

A.2. Data Format Descriptions

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

Data type format

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

Layout format

The layout format determines the ordering in which values are stored. Typically, batch dimensions are the leftmost dimensions, and the other dimensions refer to aspects of each data item, such as C is channel, H is height, and W is width, in images. Ignoring batch sizes, which are always preceding these, C, H, and W are typically sorted as CHW (see Figure 17) or HWC (see Figure 18).

Figure 17.   Layout format for `CHW`: The image is divided into `HxW` matrices, one per channel, and the matrices are stored in sequence; all the values of a channel are stored contiguously.
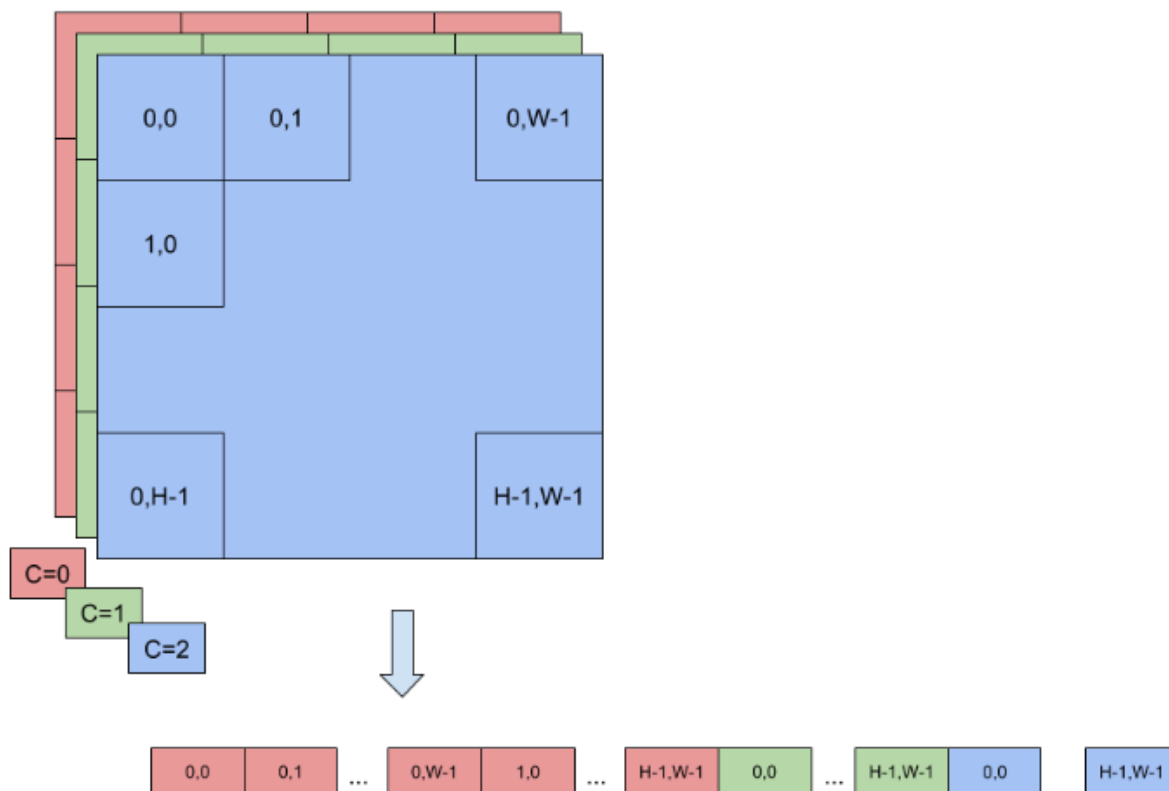
Figure 18.   Layout format for `HWC`: The image is stored as a single `HxW` matrix, whose value is actually C-tuple, with a value per channel; all the values of a point (pixel) are stored contiguously.

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

In `NC/2HW2` (`TensorFormat::kCHW2`), 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 (see Figure 19); 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.

Figure 19.   A pair of channel values are packed together in each `HxW` matrix. The result is a format in which the values of `[C/2] HxW` matrices are pairs of values of two consecutive channels.
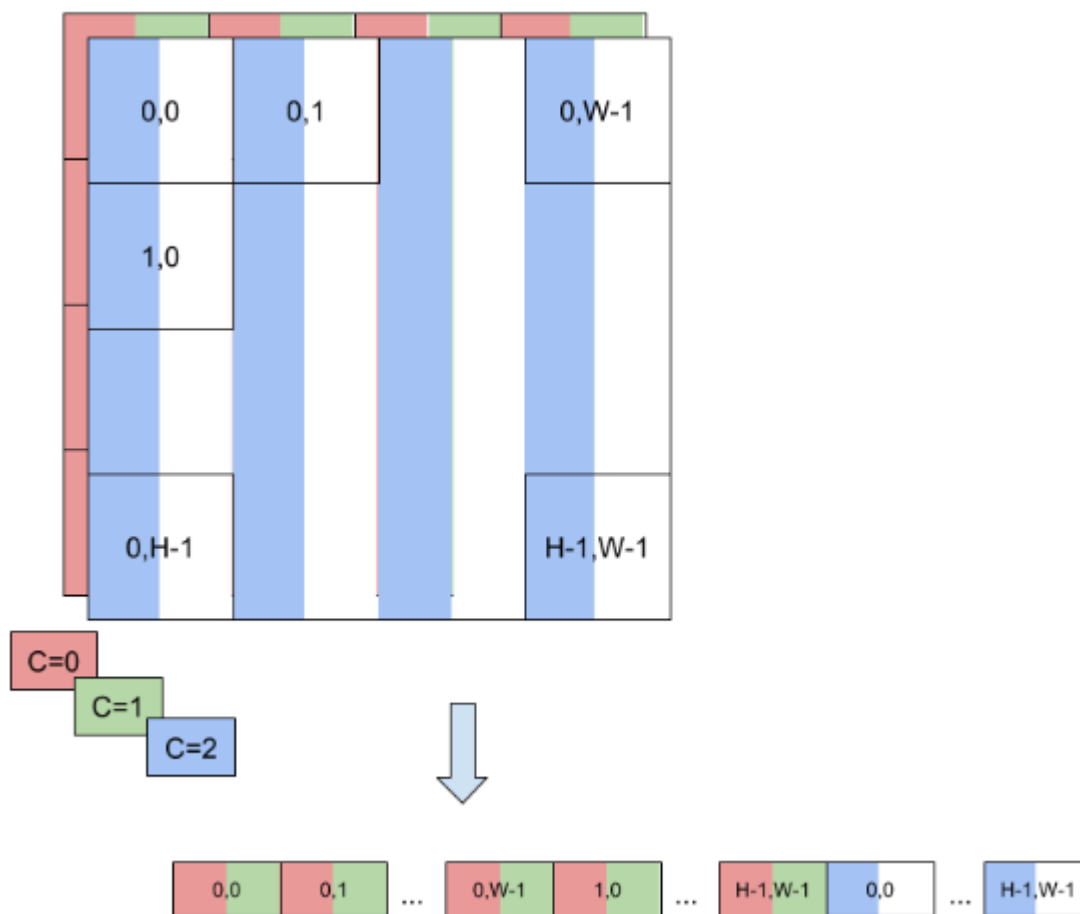
In `NHWC8` (`TensorFormat::kHWC8`), the entries of an `HxW` matrix include the values of all the channels (see [Figure 20](#)). In addition, these values are packed together in #`C`/8# 8-tuples, and `C` is rounded up to the nearest multiple of 8.

Figure 20. In this `NHWC8` format, the entries of an `HxW` matrix include the values of all the channels.
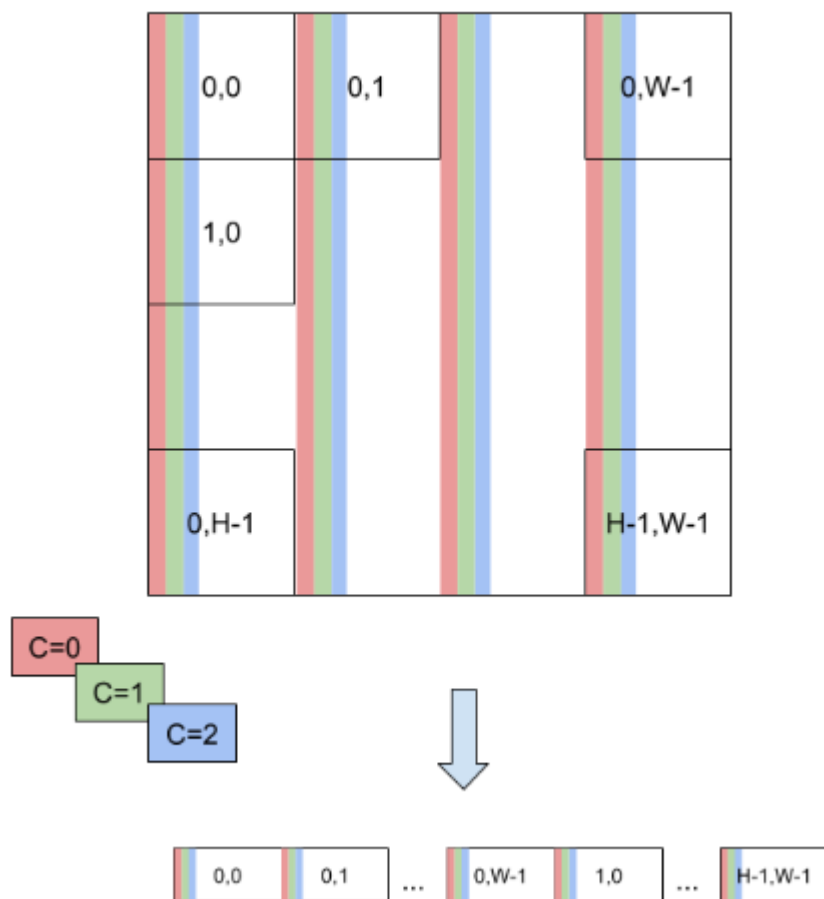
Other `TensorFormat` follow similar rules to `TensorFormat::kCHW2` and `TensorFormat::kHWC8` mentioned previously.

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 three main purposes:

▶ It's useful for *benchmarking networks* on random or user-provided input data.

▶ It's useful for *generating serialized engines* from models.

▶ It's useful for *generating serialized timing cache* from the builder.

A.3.1.1. Benchmarking Network

If you have a model saved as an ONNX file, UFF 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.

To maximize GPU utilization, `trtexec` enqueues the queries one batch ahead of time. In other words, it does the following:

```
enqueue batch 0 -> enqueue batch 1 -> wait until batch 0 is done -> enqueue batch 2 -> wait
 until batch 1 is done -> enqueue batch 3 -> wait until batch 2 is done -> enqueue batch 4 -
> ...
```

If multi-stream (`--streams=N` flag) is used, then `trtexec` follows this pattern on each stream separately.

The `trtexec` tool prints the following performance metrics. The following figure shows an example Nsight System profile of a `trtexec` run with markers showing what each performance metric means.

Throughput
The observed throughput is computed by dividing the number of queries by the Total Host Walltime. If this is significantly lower than the reciprocal of GPU Compute Time, the GPU may be underutilized because of host-side overheads or data transfers. Using CUDA graphs (with `--useCudaGraph`) or disabling H2D/D2H transfers (with `--noDataTransfer`) may improve GPU utilization. The output log provides guidance on which flag to use when `trtexec` detects that the GPU is underutilized.

Host Latency
The summation of H2D Latency, GPU Compute Time, and D2H Latency. This is the latency to infer a single query.

Enqueue Time
The host latency to enqueue a query, including calling H2D/D2H CUDA APIs, running host-side heuristics, and launching CUDA kernels. If this is longer than GPU Compute Time, the GPU may be underutilized and the throughput may be dominated by host-side overhead. Using CUDA graphs (with `--useCudaGraph`) may reduce Enqueue Time.

H2D Latency
The latency for host-to-device data transfers for input tensors of a single query. Add `--noDataTransfer` to disable H2D/D2H data transfers.

D2H Latency
The latency for device-to-host data transfers for output tensors of a single query. Add `--noDataTransfer` to disable H2D/D2H data transfers.

GPU Compute Time
The GPU latency to execute the CUDA kernels for a query.

Total Host Walltime
The host walltime from when the first query (after warm-ups) is enqueued to when the last query was completed.

Total GPU Compute Time
The summation of the GPU Compute Time of all the queries. If this is significantly shorter than Total Host Walltime, the GPU may be under-utilized because of host-side overheads or data transfers.

Figure 21.   Performance metrics in a normal `trtexec` run under Nsight Systems (ShuffleNet, BS=16, best, TitanRTX@1200MHz)
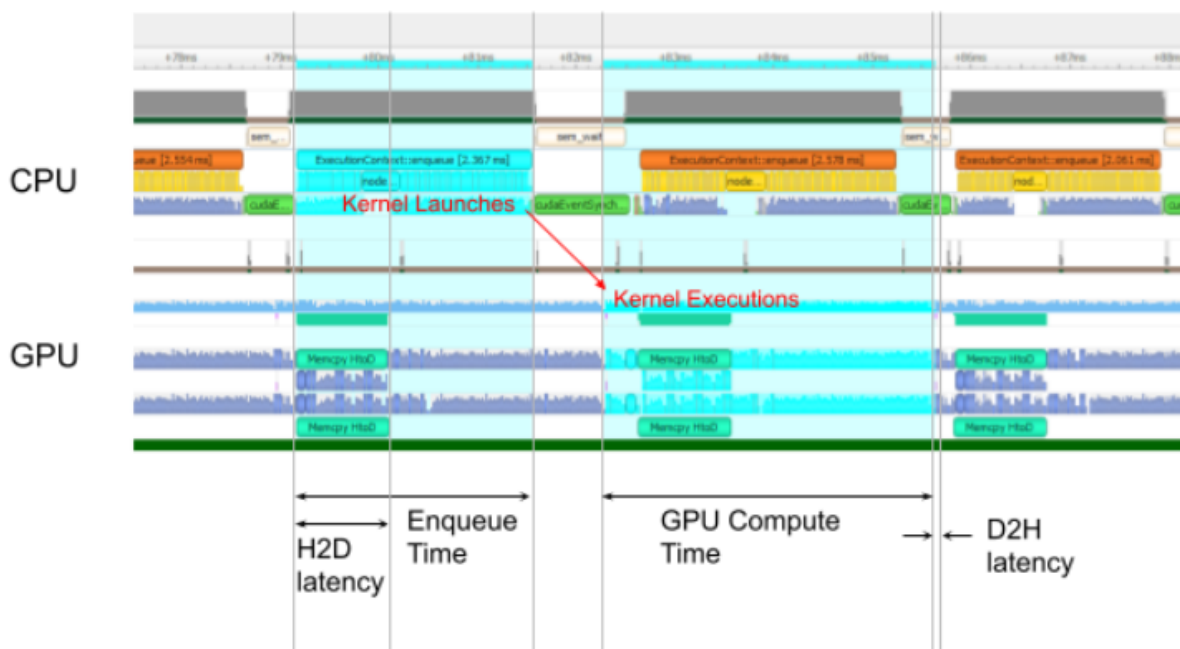
Host latency = (H2D latency) + (GPU Compute latency) + (D2H latency)

Throughput = (Total host wall time for N queries) / N

Add the `--dumpProfile` flag to `trtexec` to show per-layer performance profiles, which allows users to understand which layers in the network take the most time in GPU execution. The per-layer performance profiling works with launching inference as a CUDA graph as well (requires CUDA 11.1 onwards). In addition, build the engine with the `--profilingVerbosity=detailed` flag and add the `--dumpLayerInfo` flag to show detailed engine information, including per-layer detail and binding information. This allows you to understand which operation each layer in the engine corresponds to and their parameters.

A.3.1.2. 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.

A.3.1.3. `trtexec`

If you provide a timing cache file to the `--timingCacheFile` option, the builder can load existing profiling data from it and add new profiling data entries during layer profiling. The

timing cache file can be reused in other builder instances to improve the builder execution time. It is suggested to reuse this cache only in the same hardware/software configurations (for example, CUDA/cuDNN/TensorRT versions, device model, and clock frequency); otherwise, functional or performance issues may occur.

A.3.1.4.   Commonly used command-line flags

The section lists the commonly used `trtexec` command-line flags.

Flags for the build phase

- ▶ `--onnx=<model>`: Specify the input ONNX model.
- ▶ `--deploy=<caffe_prototxt>`: Specify the input Caffe prototxt model.
- ▶ `--uff=<model>`: Specify the input UFF model.
- ▶ `--output=<tensor>`: Specify output tensor names. Only required if the input models are in UFF or Caffe formats.
- ▶ `--maxBatch=<BS>`: Specify the maximum batch size to build the engine with. Only needed if the input models are in UFF or Caffe formats. If the input model is in ONNX format, use the `--minShapes`, `--optShapes`, `--maxShapes` flags to control the range of input shapes including batch size.
- ▶ `--minShapes=<shapes>`, `--optShapes=<shapes>`, `--maxShapes=<shapes>`: Specify the range of the input shapes to build the engine with. Only required if the input model is in ONNX format.
- ▶ `--workspace=<size in MB>`: Specify the maximum size of the workspace that tactics are allowed to use. This flag has been deprecated. You can use the `--memPoolSize=<pool_spec>` flag instead.
- ▶ `--memPoolSize=<pool_spec>`: Specify the maximum size of the workspace that tactics are allowed to use, as well as the sizes of the memory pools that DLA will allocate per loadable.
- ▶ `--saveEngine=<file>`: Specify the path to save the engine to.
- ▶ `--fp16`, `--int8`, `--noTF32`, `--best`: Specify network-level precision.
- ▶ `--sparsity=[disable|enable|force]`: Specify whether to use tactics which support structured sparsity.

  - ▶ `disable`: Disable all tactics using structured sparsity. This is the default.
  - ▶ `enable`: Enable tactics using structured sparsity. Tactics will only be used if the weights in the ONNX file meet the requirements for structured sparsity.
  - ▶ `force`: Enable tactics using structured sparsity and allow trtexec to overwrite the weights in the ONNX file to enforce them to have structured sparsity patterns. Note that the accuracy is not preserved, so this is to get inference performance only.

- ▶ `--timingCacheFile=<file>`: Specify the timing cache to load from and save to.
- ▶ `--verbose`: Turn on verbose logging.

- ▶ `--buildOnly`: Build and save the engine without running inference.
- ▶ `--profilingVerbosity=[layer_names_only|detailed|none]`: Specify the profiling verbosity to build the engine with.
- ▶ `--dumpLayerInfo`, `--exportLayerInfo=<file>`: Print/Save the layer information of the engine.
- ▶ `--precisionConstraints=spec`: Control precision constraint setting.

    - ▶ `none`: No constraints.
    - ▶ `prefer`: Meet precision constraints set by `--layerPrecisions`/`--layerOutputTypes` if possible.
    - ▶ `obey`: Meet precision constraints set by `--layerPrecisions`/`--layerOutputTypes` or fail otherwise.

- ▶ `--layerPrecisions=spec`: Control per-layer precision constraints. Effective only when `precisionConstraints` is set to `obey` or `prefer`. The specs are read left-to-right, and later ones override earlier ones. "`*`" can be used as a `layerName` to specify the default precision for all the unspecified layers.

    - ▶ For example: `--layerPrecisions=*:fp16,layer_1:fp32` sets the precision of all layers to FP16 except for `layer_1`, which will be set to FP32.

- ▶ `--layerOutputTypes=spec`: Control per-layer output type constraints. Effective only when `precisionConstraints` is set to `obey` or `prefer`. The specs are read left-to-right, and later ones override earlier ones. "`*`" can be used as a `layerName` to specify the default precision for all the unspecified layers. If a layer has more than one output, then multiple types separated by "+" can be provided for this layer.

    - ▶ For example: `--layerOutputTypes=*:fp16,layer_1:fp32+fp16` sets the precision of all layer outputs to FP16 except for `layer_1`, whose first output will be set to FP32 and whose second output will be set to FP16.

Flags for the inference phase

- ▶ `--loadEngine=<file>`: Load the engine from a serialized plan file instead of building it from input ONNX, UFF, or Caffe model.
- ▶ `--batch=<N>`: Specify the batch size to run the inference with. Only needed if the input models are in UFF or Caffe formats. If the input model is in ONNX format or if the engine is built with explicit batch dimension, use `--shapes` instead.
- ▶ `--shapes=<shapes>`: Specify the input shapes to run the inference with.
- ▶ `--warmUp=<duration in ms>`, `--duration=<duration in seconds>`, `--iterations=<N>`: Specify the minimum duration of the warm-up runs, the minimum duration for the inference runs, and the minimum iterations of the inference runs. For example, setting `--warmUp=0 --duration=0 --iterations` allows users to control exactly how many iterations to run the inference for.

- ▶ `--useCudaGraph`: Capture the inference to a CUDA graph and run inference by launching the graph. This argument may be ignored when the built TensorRT engine contains operations that are not permitted under CUDA graph capture mode.

- ▶ `--noDataTransfers`: Turn off host-to-device and device-to-host data transfers.

- ▶ `--streams=<N>`: Run inference with multiple streams in parallel.

- ▶ `--verbose`: Turn on verbose logging.

- ▶ `--dumpProfile, --exportProfile=<file>`: Print/Save the per-layer performance profile.

Refer to `trtexec --help` for all the supported flags and detailed explanations.

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

A.4. Glossary

Device
A specific GPU. Two GPUs are considered identical devices if they have the same model name and same configuration.
Platform
A combination of architecture and operating system. Example platforms are Linux on x86 and QNX Standard on Aarch64. Platforms with different architectures or different operating systems are considered different platforms.

A.5. 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:

half.h

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

TensorRT includes CRC routines from FreeBSD.

$FreeBSD: head/COPYRIGHT 260125 2013-12-31 12:18:10Z gjb $

@(#)COPYRIGHT 8.2 (Berkeley) 3/21/94

The compilation of software known as FreeBSD is distributed under the following terms:

Copyright (c) 1992-2014 The FreeBSD Project. All rights reserved.

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 AUTHOR 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 AUTHOR 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.

The 4.4BSD and 4.4BSD-Lite software is distributed under the following terms:

All of the documentation and software included in the 4.4BSD and 4.4BSD-Lite Releases is copyrighted by The Regents of the University of California.

Copyright 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994 The Regents of the University of California. All rights reserved.

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.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed by the University of California, Berkeley and its contributors.
4. Neither the name of the University 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 REGENTS 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 REGENTS 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.

The Institute of Electrical and Electronics Engineers and the American National Standards Committee X3, on Information Processing Systems have given us permission to reprint portions of their documentation.

In the following statement, the phrase ``this text'' refers to portions of the system documentation.

Portions of this text are reprinted and reproduced in electronic form in the second BSD Networking Software Release, from IEEE Std 1003.1-1988, IEEE Standard Portable Operating

System Interface for Computer Environments (POSIX), copyright C 1988 by the Institute of Electrical and Electronics Engineers, Inc. In the event of any discrepancy between these versions and the original IEEE Standard, the original IEEE Standard is the referee document.

In the following statement, the phrase ``This material'' refers to portions of the system documentation.

This material is reproduced with permission from American National Standards Committee X3, on Information Processing Systems. Computer and Business Equipment Manufacturers Association (CBEMA), 311 First St., NW, Suite 500, Washington, DC 20001-2178. The developmental work of Programming Language C was completed by the X3J11 Technical Committee.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official 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 included below.

July 22, 1999

To All Licensees, Distributors of Any Version of BSD:

As you know, certain of the Berkeley Software Distribution ("BSD") source code files require that further distributions of products containing all or portions of the software, acknowledge within their advertising materials that such products contain software developed by UC Berkeley and its contributors.

Specifically, the provision reads:

" * 3. All advertising materials mentioning features or use of this software

* must display the following acknowledgement:

* This product includes software developed by the University of

* California, Berkeley and its contributors."

Effective immediately, licensees and distributors are no longer required to include the acknowledgement within advertising materials. Accordingly, the foregoing paragraph of those BSD Unix files containing it is hereby deleted in its entirety.

William Hoskins

Director, Office of Technology Licensing

University of California, Berkeley

getopt.c

$OpenBSD: getopt_long.c,v 1.23 2007/10/31 12:34:57 chl Exp $

$NetBSD: getopt_long.c,v 1.15 2002/01/31 22:43:40 tv Exp $

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.

ONNX Model Zoo

MIT License

Copyright (c) ONNX Project Contributors

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

RESNET-50 Caffe models

The MIT License (MIT)

Copyright (c) 2016 Shaoqing Ren

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.

OpenSSL

Apache License Version 2.0

Copyright (c) OpenSSL Project Contributors

Apache License

Version 2.0, January 2004

https://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

Boost Beast

Copyright (c) 2016-2017 Vinnie Falco (vinnie dot falco at gmail dot com)

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

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, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Trademarks

NVIDIA, the NVIDIA logo, and BlueField, CUDA, DALI, DRIVE, JetPack, Jetson AGX Xavier, Jetson Nano, Kepler, Maxwell, NGC, Nsight, Orin, Pascal, Quadro, Tegra, TensorRT, Triton, 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