# TENSORRT

**Developer Guide**

# TABLE OF CONTENTS

# Chapter 1.
# WHAT IS TENSORRT?

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

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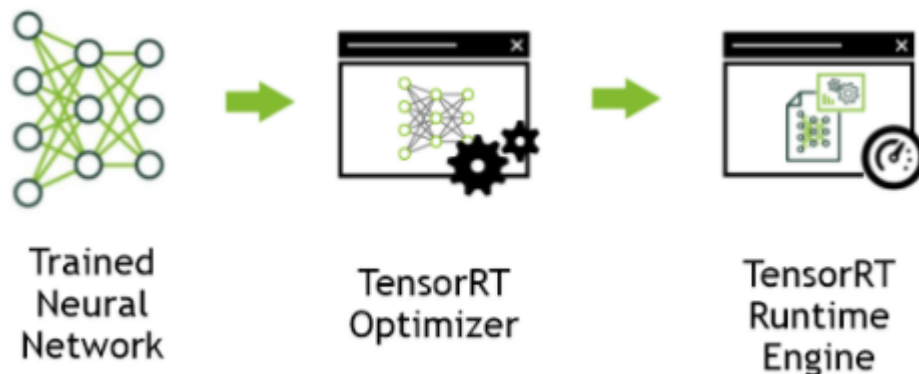


Figure 1   TensorRT is a high performance neural network inference optimizer and runtime engine for production deployment.

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

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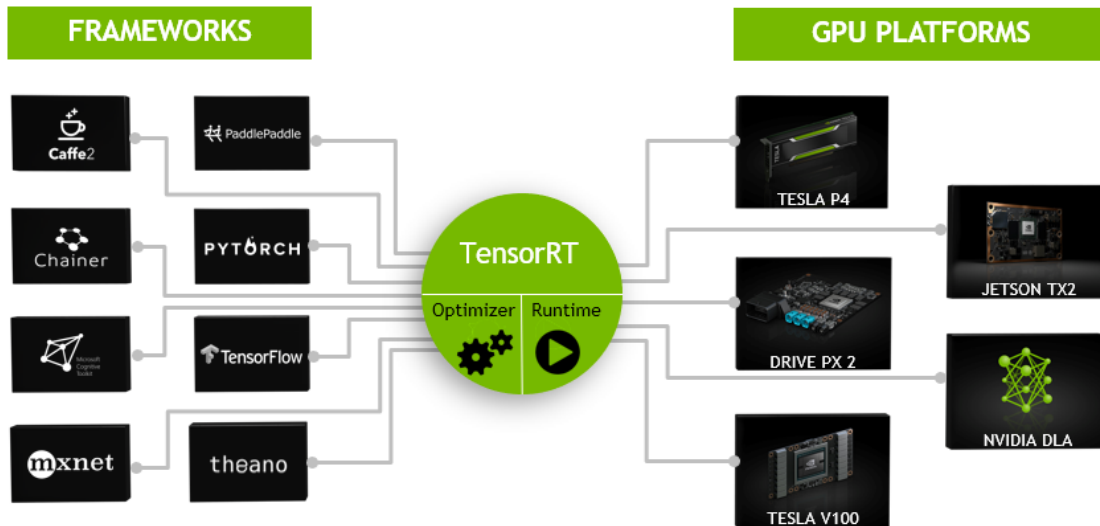


Figure 2   TensorRT is a programmable inference accelerator.

The TensorRT API includes implementations for the most common deep learning layers. For more information about the layers, see TensorRT Layers. You can also use the Plugin API to provide implementations for infrequently used or more innovative layers that are not supported out-of-the-box by TensorRT.

## 1.1. Benefits Of TensorRT

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

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

For deep learning inference, there are 5 critical factors that are used to measure software:
**Throughput**
    The volume of output within a given period. Often measured in inferences/second or samples/second, per-server throughput is critical to cost-effective scaling in data centers.
**Efficiency**
    Amount of throughput delivered per unit-power, often expressed as performance/watt. Efficiency is another key factor to cost effective data center scaling, since servers, server racks and entire data centers must operate within fixed power budgets.

**Latency**

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

**Accuracy**

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

**Memory usage**

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

Alternatives to using TensorRT include:

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

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

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

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

# 1.2. Where Does TensorRT Fit?

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

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

**Phase 1: Training**

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

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

**Phase 2: Developing A Deployment Solution**

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

1. Think about how the neural network functions within the larger system of which it is a part of and design and implement an appropriate solution. The range of systems that might incorporate neural networks are tremendously diverse. Examples include:

   ▶ the autonomous driving system in a vehicle
   ▶ a video security system on a public venue or corporate campus
   ▶ the speech interface to a consumer device
   ▶ an industrial production line automated quality assurance system
   ▶ an online retail system providing product recommendations, or
   ▶ a consumer web service offering entertaining filters users can apply to uploaded images.

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

   ▶ Do you have a single network or many networks? For example, Are you developing a feature or system that is based on a single network (face detection), or will your system be comprised of a mixture or cascade of different models, or perhaps a more general facility that serves up a collection model that may be provided by the end user?
   ▶ What device or compute element will you use to run the network? CPU, GPU, other, or a mixture? If the model is going to run on a GPU, is it a single type of GPU, or do you need to design an application that can run on a variety of GPUs?

▸ How is data going to get to the models? What is the data pipeline? Is the data coming in from a camera or sensor, from a series of files, or being uploaded over a network connection?

▸ What pre-processing will be done? What format will the data come in? If it is an image does it need to be cropped, rotated? If it is text what character set is it and are all characters allowed as inputs to the model? Are there any special tokens?

▸ What latency and throughput requirements will you have?

▸ Will you be able to batch together multiple requests?

▸ Will you need multiple instances of a single network to achieve the required overall system throughput and latency?

▸ What will you do with the output of the network?

▸ What post processing steps are needed?

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

2. After the data scientists and developers define the architecture of their inference solution, by which they determine what their priorities are, they then build an inference engine from the saved network using TensorRT. There are a number of ways to do this depending on the training framework used and the network architecture. Generally, this means you need to take the saved neural network and parse it from its saved format into TensorRT using the ONNX parser (see Figure 3), Caffe parser, or TensorFlow/UFF parser.
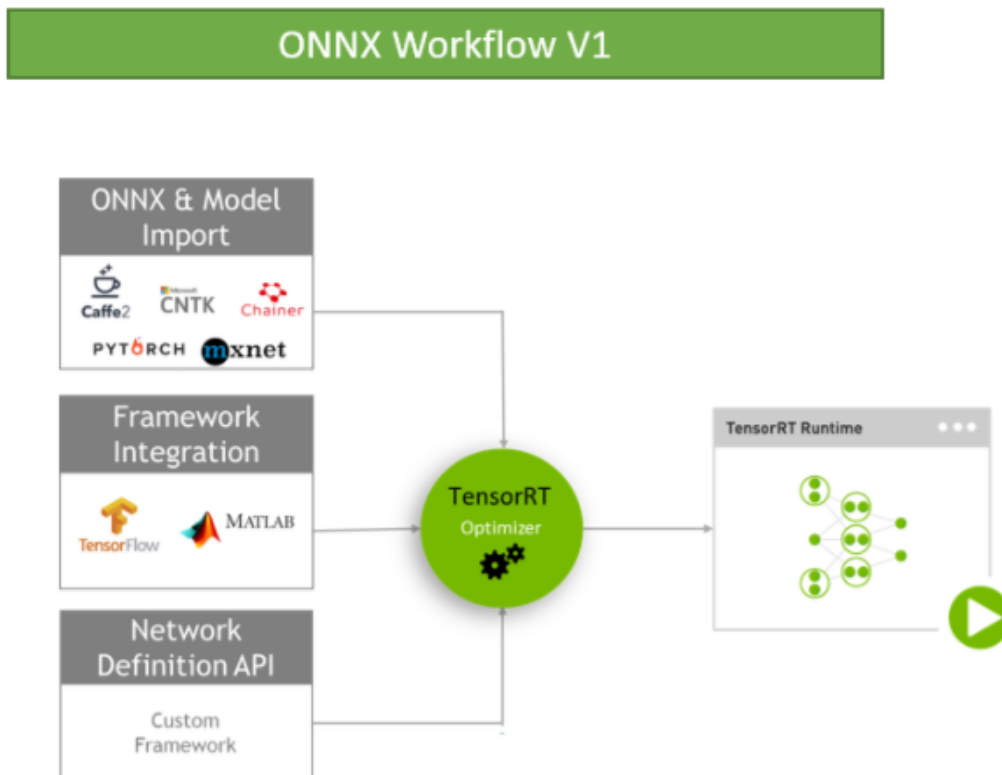
Figure 3   ONNX Workflow V1

3.  After the network is being parsed, you'll need to consider optimization options
    -- batch size, workspace size and mixed precision. These options are chosen and
    specified as part of the TensorRT build step where you actually build an optimized
    inference engine based on your network. Subsequent sections of this guide provide
    detailed instructions and numerous examples on this part of the workflow, parsing
    your model into TensorRT and choosing the optimization parameters (see Figure 4).
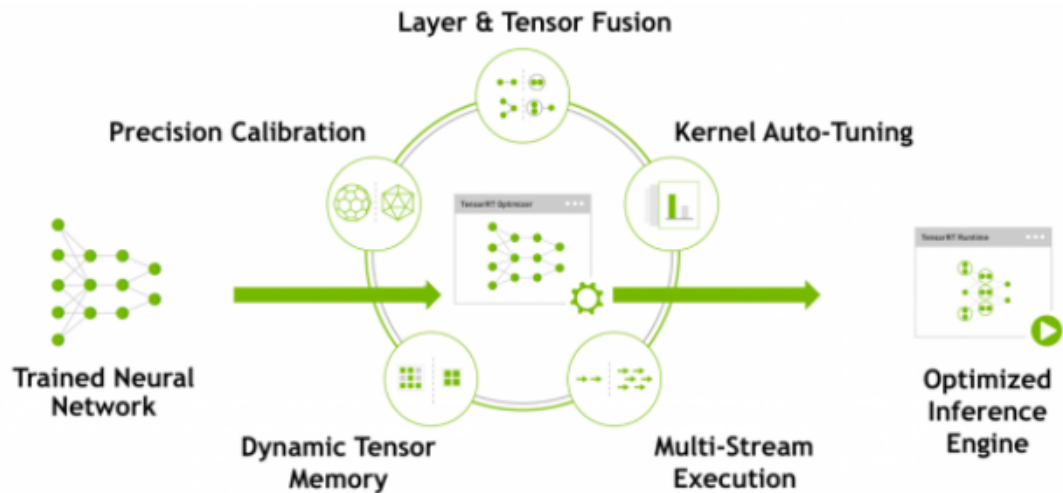
Figure 4   TensorRT optimizes trained neural network models to produce a deployment-ready runtime inference engine.

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

5.  Write out the inference engine in a serialized format. This is also called a plan file.

**Phase 3: Deploying A Solution**

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

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

# 1.3. How Does TensorRT Work?

To optimize your model for inference, TensorRT takes your network definition, performs optimizations including platform specific optimizations, and generates the inference engine. This process is referred to as the build phase. The build phase can take considerable time, especially when running on embedded platforms. Therefore, a typical application will build an engine once, and then serialize it for later use.

> The generated plan file must be retargeted to the specific GPU in case you want to run it on a different GPU.

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

▸ Elimination of layers whose outputs are not used
▸ Fusion of convolution, bias and ReLU operations
▸ Aggregation of operations with sufficiently similar parameters and the same source tensor (for example, the 1x1 convolutions in GoogleNet v5's inception module)
▸ Merging of concatenation layers by directing layer outputs to the correct eventual destination.

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

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

For more information, see Working With Mixed Precision.

# 1.4. What Capabilities Does TensorRT Provide?

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

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

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

The key interfaces in the TensorRT core library are:
**Network Definition**
    The Network Definition interface provides methods for the application to specify the definition of a network. Input and output tensors can be specified, layers can be added, and there is an interface for configuring each supported layer type. As well as layer types, such as convolutional and recurrent layers, and a Plugin layer type allows the application to implement functionality not natively supported by TensorRT. For more information about the Network Definition, see Network Definition API.
**Builder**
    The Builder interface allows creation of an optimized engine from a network definition. It allows the application to specify the maximum batch and workspace size, the minimum acceptable level of precision, timing iteration counts for autotuning, and an interface for quantizing networks to run in 8-bit precision. For more information about the Builder, see Builder API.

**Engine**

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

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

**Caffe Parser**

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

**UFF Parser**

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

**ONNX Parser**

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

# 1.5. How Do I Get TensorRT?

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

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

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

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

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

**C++ API vs Python API**

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

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

## 2.1. Instantiating TensorRT Objects in C++

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

The engine can be created in one of two ways:

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

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

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

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



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

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



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

One of the available parsers is created using the **iNetwork** definition as the input:

- ▸ ONNX: **parser = nvonnxparser::createParser(*network, gLogger);**
- ▸ NVCaffe: **ICaffeParser* parser = createCaffeParser();**
- ▸ UFF: **parser = createUffParser();**

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



Figure 7  Parsing the model file

A method called **buildCudaEngine()** of **iBuilder** is called to create an object of **iCudaEngine** type as shown in Figure 8:
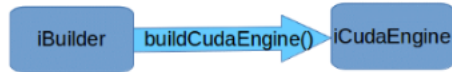
Figure 8   Creating the TensorRT engine

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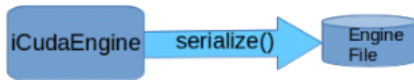


Figure 9   Creating the TensorRT engine

The execution context is used to perform inference.
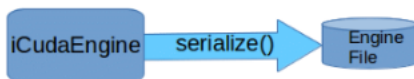


Figure 10   Creating an execution context

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

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



Figure 11   Creating TensorRT runtime

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

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

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

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

# 2.2. Creating A Network Definition In C++

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

▸ sampleMNIST (both BVLC and NVCaffe)
▸ sampleOnnxMNIST 1.0 and 1.1, and
▸ sampleUffMNIST (used for TensorFlow)

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

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

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

## 2.2.1. Creating A Network Definition From Scratch Using The C++ API

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

In the following example, we will create a simple network with Input, Convolution, Pooling, FullyConnected, Activation and SoftMax layers. To see the code in totality, refer to sampleMNISTAPI located in the **/usr/src/tensorrt/samples/sampleMNISTAPI** directory.

1. Create the builder and the network:

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

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

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

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

```
layerName->getOutput(0)
```

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

> 💬 Weights passed to TensorRT layers are in host memory.

4. Add the Pooling layer:

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

5. Add the FullyConnected and Activation layers:

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

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

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

7. Mark the output:

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

## 2.2.2. Importing A Model Using A Parser In C++

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

1. Create the TensorRT builder and network.

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

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

2. Create the TensorRT parser for the specific format.
   **ONNX**
   ```
   auto parser = nvonnxparser::createParser(*network,
           gLogger);
   ```

   **UFF**
   ```
   auto parser = createUffParser();
   ```
   **NVCaffe**
   ```
   ICaffeParser* parser = createCaffeParser();
   ```

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

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

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

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

## 2.2.3. Importing A Caffe Model Using The C++ Parser API

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

1. Create the builder and network:

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

2. Create the Caffe parser:

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

3. Parse the imported model:

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

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

In addition to populating the network definition, the parser returns a dictionary that maps from Caffe blob names to TensorRT tensors. Unlike Caffe, a TensorRT network definition has no notion of in-place operation. When an Caffe model uses an in-place operation, the TensorRT tensor returned in the dictionary corresponds to the last write to that blob. For example, if a convolution writes to a blob and is followed by an in-place ReLU, that blob's name will map to the TensorRT tensor which is the output of the ReLU.

4. Specify the outputs of the network:

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

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

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

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

1. Create the builder and network:

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

2. Create the UFF parser:

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

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

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

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

4. Parse the imported model to populate the network:

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

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

> **Restriction** Since the ONNX format is quickly developing, you may encounter a version mismatch between the model version and the parser version. The ONNX Parser shipped with TensorRT 5.0 RC supports ONNX IR (Intermediate Representation) version 0.0.3, opeset version 7.
>
> In general, the newer version of the ONNX Parser is designed to be backward compatible, therefore, encountering a model file produced by an earlier version of ONNX exporter should not cause a problem. There could be some exceptions when the changes were not backward compatible. In this case, convert the earlier ONNX model file into a later supported version. For more information on this subject, see ONNX Model Opset Version Converter.
>
> It is also possible that the user model was generated by an exporting tool supporting later opsets than supported by the ONNX parser shipped with TensorRT. In this case, check whether the latest version of TensorRT released to GitHub **onnx-tensorrt** supports the required version. The supported version is defined by the **BACKEND_OPSET_VERSION** variable in **onnx_trt_backend.cpp**. Download and build the latest version of ONNX TensorRT Parser from the GitHub. The instructions for building can be found here: TensorRT backend for ONNX.

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

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

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

2. Ingest the model:

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

3. Convert the model to a TensorRT network:

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

4. Obtain the network from the model:

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

# 2.3. Building An Engine In C++

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

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

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

▶ The maximum batch size specifies the batch size for which TensorRT will optimize. At runtime, a smaller batch size may be chosen.

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

1. Build the engine using the builder object:

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

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

2. Dispense with the network, builder, and parser if using one.

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

# 2.4. Serializing A Model In C++

To serialize, you are transforming the engine into a format to store and use at a later time for inference. To use for inference, you would simply deserialize the engine. Serializing

and deserializing are optional. Since creating an engine from the Network Definition can be time consuming, you could avoid rebuilding the engine every time the application reruns by serializing it once and deserializing it while inferencing. Therefore, after the engine is built, users typically want to serialize it for later use.

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

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

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

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

2. Create a runtime object to deserialize:

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

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

## 2.5. Performing Inference In C++

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

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

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

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

2. Use the input and output blob names to get the corresponding input and output index:

```
int inputIndex = engine.getBindingIndex(INPUT_BLOB_NAME);
int outputIndex = engine.getBindingIndex(OUTPUT_BLOB_NAME);
```

3. Using these indices, set up a buffer array pointing to the input and output buffers on the GPU:

```
void* buffers[2];
buffers[inputIndex] = inputbuffer;
buffers[outputIndex] = outputBuffer;
```

4. TensorRT execution is typically asynchronous, so **enqueue** the kernels on a CUDA stream:

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

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

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

## 2.6. Memory Management In C++

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

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

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

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

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

# Chapter 3.
# WORKING WITH TENSORRT USING THE PYTHON API

The following sections highlight the TensorRT user goals and tasks that you can perform using the Python API. These sections focus on using the Python API without any frameworks. Further details are provided in the Samples section and are linked to below where appropriate.

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

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

**Python API vs C++ API**

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

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

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

## 3.1. Importing TensorRT Into Python

1. Import TensorRT:

```
import tensorrt as trt
```

2. Implement a logging interface through which TensorRT reports errors, warnings, and informational messages. The following code shows how to implement the logging interface. In this case, we have suppressed informational messages, and report only warnings and errors. There is a simple logger included in the TensorRT Python bindings.

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

# 3.2. Creating A Network Definition In Python

The first step in performing inference with TensorRT is to create a TensorRT network from your model. The easiest way to achieve this is to import the model using the TensorRT parser library, (see Importing A Model Using A Parser In Python, Importing From Caffe Using Python, Importing From TensorFlow Using Python, and Importing From ONNX Using Python), which supports serialized models in the following formats:

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

An alternative is to define the model directly using the TensorRT API, (see Creating A Network Definition From Scratch Using The Python API). This requires you to make a small number of API calls to define each layer in the network graph, and to implement your own import mechanism for the model's trained parameters.

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

# 3.2.1. Creating A Network Definition From Scratch Using The Python API

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

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

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

```
input_tensor = network.add_input(name=INPUT_NAME, dtype=trt.float32,
shape=INPUT_SHAPE)

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

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

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

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

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

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

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

## 3.2.2. Importing A Model Using A Parser In Python

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

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

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

The builder must be created before the network because it serves as a factory for the network. Different parsers have different mechanisms for marking network outputs. For more information, see the Python Parser API.

## 3.2.3. Importing From Caffe Using Python

The following steps illustrate how to import a Caffe model directly using the CaffeParser and the Python API. Refer to the introductory_parser_samples sample for more information.

1.  Import TensorRT.

    ```
    import tensorrt as trt
    ```

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

    ```
    datatype = trt.float32
    ```

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

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

4.  Create the builder, network, and parser:

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

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

## 3.2.4. Importing From TensorFlow Using Python

The following steps illustrate how to import a TensorFlow model directly using the UffParser and the Python API. This sample can be found in the **<site-packages>/ tensorrt/samples/python/end_to_end_tensorflow_mnist** directory. For more information, see the end_to_end_tensorflow_mnist Python sample.

1.  Import TensorRT:

    ```
    import tensorrt as trt
    ```

2.  Create a frozen TensorFlow model for the **tensorflow** model. The instructions on freezing a TensorFlow model into a stream can be found in Freezing A TensorFlow Graph.

3.  Use the UFF converter to convert a frozen **tensorflow** model to a UFF file. Typically, this is as simple as:

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

    Alternatively, you can use the **uff** python module and convert the TensorFlow GraphDef directly.

4.  Define some paths. Change the following paths to reflect where you placed the model that is included with the samples:

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

5.  Create the builder, network, and parser:

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

## 3.2.5. Importing From ONNX Using Python

> **Restriction** Since the ONNX format is quickly developing, you may encounter a version mismatch between the model version and the parser version. The ONNX Parser shipped with TensorRT 5.0 RC supports ONNX IR (Intermediate Representation) version 0.0.3, opset version 7.
>
> In general, the newer version of the ONNX Parser is designed to be backward compatible, therefore, encountering a model file produced by an earlier version of ONNX exporter should not cause a problem. There could be some exceptions when the changes were not backward compatible. In this case, convert the earlier ONNX model file into a later supported version. For more information on this subject, see ONNX Model Opset Version Converter.
>
> It is also possible that the user model was generated by an exporting tool supporting later opsets than supported by the ONNX parser shipped with TensorRT. In this case, check whether the latest version of TensorRT released to GitHub `onnx-tensorrt` supports the required version. The supported version is defined by the `BACKEND_OPSET_VERSION` variable in `onnx_trt_backend.cpp`. Download and build the latest version of ONNX TensorRT Parser from the GitHub. The instructions for building can be found here: TensorRT backend for ONNX.

The following steps illustrate how to import an ONNX model directly using the OnnxParser and the Python API. For more information, see the introductory_parser_samples Python sample.

1. Import TensorRT:

   ```
   import tensorrt as trt
   ```

2. Create the build, network, and parser:

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

## 3.2.6. Importing From PyTorch And Other Frameworks

Using TensorRT with PyTorch (or any other framework with NumPy compatible weights) involves replicating the network architecture using the TensorRT API, (see Creating A Network Definition From Scratch Using The Python API), and then copying

the weights from PyTorch. For more information, see Working With PyTorch And Other Frameworks.

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

## 3.3. Building An Engine In Python

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

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

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

▸ The maximum batch size specifies the batch size for which TensorRT will optimize. At runtime, a smaller batch size may be chosen.

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

For more information about building an engine in Python, see the introductory_parser_samples sample.

1. Build the engine using the builder object:

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

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

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

## 3.4. Serializing A Model In Python

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

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

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

1. Serialize the model to a modelstream:

   ```
   serialized_engine = engine.serialize()
   ```
2. Deserialize modelstream to perform inference. Deserializing requires creation of a runtime object:

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

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

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

1. Serialize the engine and write to a file:

   ```
   with open("sample.engine", "wb") as f:
     f.write(engine.serialize())
   ```
2. Read the engine from the file and deserialize:

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

# 3.5. Performing Inference In Python

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

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

   ```
   # Determine dimensions and create page-locked memory buffers (i.e. won't be
    swapped to disk) to hold host inputs/outputs.
     h_input = cuda.pagelocked_empty(engine.get_binding_shape(0).volume(),
    dtype=np.float32)
   ```

```
  h_output = cuda.pagelocked_empty(engine.get_binding_shape(1).volume(),
 dtype=np.float32)
  # Allocate device memory for inputs and outputs.
  d_input = cuda.mem_alloc(h_input.nbytes)
  d_output = cuda.mem_alloc(h_output.nbytes)
  # Create a stream in which to copy inputs/outputs and run inference.
  stream = cuda.Stream()
```

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

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

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

# Chapter 4.
# EXTENDING TENSORRT WITH CUSTOM LAYERS

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

## 4.1. Adding Custom Layers Using The C++ API

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

**IPluginExt**

    `IPluginExt` includes versioning support and helps enable custom layers that support other data formats besides NCHW and single precision. It also includes the ability to clone plugin objects in the network, builder, and engine so that they each have their own copy of the plugin object and plugin parameters. If there are multiple builders created for the same network, each builder will have their own copy of the plugin object and can be independently destroyed.

**IPluginCreator**

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

TensorRT also provides the ability to register a plugin by calling `REGISTER_TENSORRT_PLUGIN(pluginCreator)` which statically registers the Plugin Creator to the Plugin Registry. During runtime, the Plugin Registry can be queried using the extern function `getPluginRegistry().` The Plugin Registry stores a pointer to all the registered Plugin Creators and can be used to look up a specific Plugin Creator based on the plugin name and version. TensorRT library contains plugins statically registered in the Plugin Registry. The version of all these plugins is set to 1. The names of these plugins are:

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

> 💬 To use TensorRT registered plugins in your application, the **libnvinfer_plugin.so** library must be loaded. This can be done by calling **registerAllTensorRTPlugins()** in your application code. Alternatively, **dlopen** can also be used.

> 💬 In the current version, the Plugin Registry will only register plugins with a unique **{name, version}** tuple. The API for this is likely to change in future versions to support multiple plugins with same names.

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

Using the Plugin Creator, the **IPluginCreator::createPlugin()** function can be called which returns a plugin object of type **IPluginExt**. This object can be added to the TensorRT network using **addPluginExt()** which creates and adds a layer to a network, and then binds the layer to the given plugin. The method also returns a pointer to the layer (of type **IPluginLayer**), which can be used to access the layer or the plugin itself (via **getPlugin()**).

```
auto creator = getPluginRegistry()->getPluginCreator(pluginName, pluginVersion);
const PluginFieldCollection* pluginFC = creator->getFieldNames();
//populate the field parameters (say layerFields) for the plugin layer
PluginFieldCollection *pluginData = parseAndFillFields(pluginFC, layerFields);
IPluginExt *pluginObj = creator->createPlugin(layerName, pluginData);
auto layer = network.addPluginExt(&inputs[0], int(inputs.size()), pluginObj);
… (build rest of the network and serialize engine)
pluginObj->destroy() // Destroy the plugin object
… (destroy network, engine, builder)
… (free allocated pluginData)
```

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

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

During serialization, the TensorRT engine will internally store the plugin type and plugin version for all **IPluginExt** type plugins. During deserialization, this information is looked up by the TensorRT engine to find the Plugin Creator from the Plugin Registry. This enables the TensorRT engine to internally call the

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

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

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

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

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

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

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

If you are using plugins registered with the TensorRT plugin registry (see Adding Custom Layers Using The C++ API for how to register your plugin), then you do not need to implement the `nvinfer1::IPluginFactory` class. However, you do need to implement the `IPluginCreator` class instead and register it.

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

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

▸ samplePlugin has a user implemented plugin
▸ sampleFasterRCNN uses plugins registered with the TensorRT Plugin Registry

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

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

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

1. Implement the **IPluginExt** and **IPluginCreator** classes as shown in Example 1: Adding A Custom Layer Using C++ For Caffe.

2. Map the TensorFlow operation to the plugin operation. You can use graphsurgeon for this. For example, refer to the following code snippet to map the TensorFlow **Relu6** operation to a plugin:

```
import graphsurgeon as gs
my_relu6 = gs.create_plugin_node(name="MyRelu6", op="Relu6_TRT")
Namespace_plugin_map = { "tf_relu6" : my_relu6 }
def preprocess(dynamic_graph):
 dynamic_graph.collapse_namespaces(namespace_plugin_map)
```

In the above code, **tf_relu6** is the name of the Relu6 node in the TensorFlow graph. Save the code above to a file called **config.py**. If the plugin layer expects parameters, they should be passed in as arguments to **gs.create_plugin_node**.

3. Call the UFF converter with the preprocess flag set:

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

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

   The sampleUffSSD sample illustrates how to add a custom layer that is not supported in UFF using C++. See **config.py** in the sample folder for a demonstration of how to pre-process the graph.

# 4.2. Adding Custom Layers Using The Python API

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

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

The same custom layer implementation can be used for both C++ and Python. For more information, see the fc_plugin_caffe_mnist Python sample located in the **/usr/src/tensorrt/samples/fc_plugin_caffe_mnist/** directory.

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

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

```
import tensorrt as trt
import numpy as np

TRT_LOGGER = trt.Logger()

trt.register_all_tensorrt_plugins()
PLUGIN_CREATORS = trt.get_plugin_registry().plugin_creator_list

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

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

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

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

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

1. Load the libnvinfer_plugin.so by calling **trt.register_all_tensorrt_plugins** (or load the .so file where you have registered your own plugin). This will register the plugin with TensorRT Plugin Registry.

2. Prepare the network and check the TensorFlow output:

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

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

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

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

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

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

```
uff_model = uff.from_tensorflow(dynamic_graph.as_graph_def(), ["trt_lrelu"],
 output_filename=model_path, text=True)
parser = trt.UffParser()
parser.register_input("placeholder", [lrelu_args.size])
parser.register_output("trt_lrelu")
parser.parse(model_path, trt_network)
```

For more information, see the uff_custom_plugin sample.

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

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

**TensorFlow**

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

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

- ▸ `IPluginExt` and `IPluginCreator` in the C++ API
- ▸ `IPluginExt` and `IPluginCreator` in the Python API

**Caffe**

For Caffe models, use the `nvcaffeparser1::IPluginFactoryExt` class. The Plugin Registry is not used by the NvCaffeParser, since there is no way to store and parse custom layer parameters in a Caffe prototxt file. The `setPluginFactoryExt` method of the parser sets the factory in the parser to enable custom layers. While parsing a model description, for each layer, the parser invokes `isPluginExt` to check with the factory if the layer name corresponds to a custom layer; if it does, the parser instantiates the plugin invoking `createPlugin` with the name of the layer (so that the factory can instantiate the corresponding plugin), a `Weights` array, and the number of weights as arguments. There is no restriction on the number of plugins that a single factory can support if they are associated with different layer names.

> For both Caffe and UFF parser use cases, if `setPluginFactoryExt` and `IPluginFactoryExt` are used, the plugin object created during deserialization will be internally destroyed by the engine by calling `IPluginExt::destroy()`. User is only responsible for destroying the plugin object created during network creation step as shown in Adding Custom Layers Using The C++ API.

The samplePlugin sample illustrates how to extend `nvcaffeparser1::IPluginFactoryExt` to use custom layers, while sampleUffSSD uses the UFF Parser to use custom layers. For the Python usage of custom layers with TensorRT, refer to the fc_plugin_caffe_mnist sample for Caffe networks, and the uff_custom_plugin sample for UFF networks.

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

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

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

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

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

## 4.4. Plugin API Description

All new plugins should implement both the **IPluginExt** and **IPluginCreator** classes. In addition, new plugins should also call the **REGISTER_TENSORRT_PLUGIN(...)** macro to register the plugin with the TensorRT Plugin Registry.

### 4.4.1. Migrating Plugins From TensorRT 4.0.x To TensorRT 5.0 RC

TensorRT 5.0 introduces 4 new methods to the **IPluginExt** class. If you have a custom plugin implemented of type **IPluginExt**, you must implement these new methods and re-compile your code, see samplePlugin for an example. The description of these methods are as follows:

**virtual void const char* getPluginType() const = 0**
> This method returns the plugin type or name of the plugin implemented.

**virtual void const char* getPluginVersion() const = 0**
> This method returns the plugin version.

**virtual void destroy() = 0**
> This method is 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, network, or engine is destroyed.

**virtual IPluginExt* clone() const = 0**
> This method clones the plugin object. This method returns a new plugin object after copying over the plugin parameters, if any.

For the simplest migration, a typical implementation will choose unique values for type and version. **clone()** would call a copy constructor and **destroy()** would call the object's destructor.

### 4.4.2. **IPluginExt** API Description

The following section describes the functions of the **IPluginExt** class.

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

**getNbOutputs**
> Used to specify the number of output tensors.

**getOutputDimensions**
> Used to specify the dimensions of an output as a function of the input dimensions.

**supportsFormat**
> Used to check if a plugin supports a given data format.

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

▶ **NCHW** single and half precision tensors
▶ **NC/2HW2** and **NHWC8** half precision tensors

The formats are enumerated by **PluginFormatType**.

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

During both build and inference time, the plugin layer is configured and executed, possibly multiple times. At build time, to discover optimal configurations, the layer is configured, initialized, executed, and terminated. Once the optimal format is selected for a plugin, the plugin is once again configured, and then it will be initialized once and executed as many 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:

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

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

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

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

In addition, the plugins also implement the following methods for plugin object memory management:

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

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

## 4.4.3. **IPluginCreator** API Description

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

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

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

**getFieldNames**

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

**createPlugin**

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

**deserializePlugin**

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

# 4.5. Best Practices For Custom Layers

## Converting User-Defined Layers

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

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

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

## Using The UFF Plugin API

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

## Debuggin Custom Layer Issues

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

# Chapter 5.
# WORKING WITH MIXED PRECISION

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

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

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

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

## 5.1. Enabling FP16 Inference Using C++

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

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

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

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

See sampleGoogleNet for an example of running FP16 inference.

## 5.2. Enabling FP16 Inference Using Python

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

```
builder.fp16_mode = True;
```

For more information, see sampleMNIST for C++.

# 5.3. Optimizing INT8 Calibration Using C++ API

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

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

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

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

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

1. Builds a 32-bit engine, runs it on the calibration set, and records a histogram for each tensor of the distribution of activation values.
2. Builds a calibration table from the histograms.
3. Builds the INT8 engine from the calibration table and the network definition.

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

The cache is used as follows:

▸ if a calibration table is found, calibration is skipped, otherwise:

   ▸ the calibration table is built from the histograms and parameters
▸ then the INT8 network is built from the network definition and the calibration table.

Cached data is passed as a pointer and length.

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

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

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

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

# 5.4. Optimizing INT8 Calibration Using Python

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

1. Import TensorRT:

```
import tensorrt as trt
```

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

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

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

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

4. Set INT8 mode and INT8 Calibrator:

```
trt_builder = trt.Builder(TRT_LOGGER)
trt_builder.int8_mode = True
trt_builder.int8_calibrator = Int8_calibrator
```

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

# Chapter 6.
# WORKING WITH DLA

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

For more information about DLA support in TensorRT layers, see DLA Supported Layers. The **trtexec** tool has additional arguments to run networks on DLA, see Command Line Wrapper. To run the AlexNet network on DLA using **trtexec**, issue:

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

## 6.1. Running On DLA During TensorRT Inference

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

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

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

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

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

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

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

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

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

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

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

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

## 6.1.1. Example 1: sampleMNIST With DLA

This section provides details on how to run a TensorRT sample with DLA enabled. The sampleMNIST sample demonstrates how to import a trained Caffe model, build the TensorRT engine, serialize and deserialize the engine and finally use the engine to perform inference.

The sample first creates the builder:

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

Then, enable **`GPUFallback`** mode:

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

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

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

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

```
./sample_mnist --useDLA=1
```

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

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

1. Create the builder and the network:

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

2. Add the Input layer to the network, with the input dimensions.

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

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

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

4. Set the convolution layer to run on DLA:

```
if(canRunOnDLA(conv1))
{
builder->setFp16Mode(true);
builder->setDeviceType(conv1, DeviceType::kDLA0);

}
```

5. Mark the output:

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

# 6.2. DLA Supported Layers

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

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

▶ Max batch size supported is 32.

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

▶ Input and output tensor data format should be FP16.

**Layer specific restrictions**
Convolution, Deconvolution, and Fully Connected Layers
**Convolution and Deconvolution Layers**

   ▶ Width and height of kernel size must be in the range [1, 32]
   ▶ Width and height of padding must be in the range [0, 31]

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

**Pooling Layer**

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

**Activation Layer**

- ▸ Functions supported: **ReLU**, **Sigmoid**, **Hyperbolic Tangent**

  - ▸ Negative slope not supported for ReLU

**ElementWise Layer**

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

**Scale Layer**

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

**LRN (Local Response Normalization) Layer**

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

**Concatenation Layer**

- ▸ DLA supports concatenation only along the channel axis

# 6.3. GPU Fallback Mode

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

1. The **layer** operation is not supported on DLA.
2. The parameters specified are out of supported range for DLA.
3. The given batch size exceeds the maximum permissible DLA batch size. For more information, see DLA Supported Layers.

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

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

# Chapter 7.
# DEPLOYING A TENSORRT OPTIMIZED MODEL

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

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

## 7.1. Deploying In The Cloud

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

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

## 7.2. Deploying To An Embedded System

TensorRT can also be used to deploy trained networks to embedded systems such as NVIDIA Drive PX. In this context, deployment means taking the network and using it

in a software application running on the embedded device, such as an object detection or mapping service. Deploying a trained network to an embedded system involves the following steps:

1. Export the trained network to a format such as UFF or ONNX which can be imported into TensorRT (see Working With Deep Learning Frameworks for more details).

2. Write a program that uses the TensorRT C++ API to import, optimize, and serialize the trained network to a plan file (see sections Working With Deep Learning Frameworks, Working With Mixed Precision, and Performing Inference In C++). For the purpose of discussion, let's call this program `make_plan`.

   a) Optionally, perform INT8 calibration and export a calibration cache (see Working With Mixed Precision).

3. Build and run `make_plan` on the host system to validate the trained model before deployment to the target system.

4. Copy the trained network (and INT8 calibration cache, if applicable) to the target system. Re-build and re-run the `make_plan` program on the target system to generate a plan file.

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

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

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

▸ Deploying INT8 Inference For Autonomous Vehicles for DRIVE PX
▸ GitHub for Jetson and Jetpack

# Chapter 8.
# WORKING WITH DEEP LEARNING FRAMEWORKS

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

## 8.1. Supported Operations By Framework

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

**Caffe**
These are the operations that are supported in a Caffe framework:

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

‣ **`Reshape`**
‣ **`Permute`**
‣ **`Dropout`**

## TensorFlow

These are the operations that are supported in a TensorFlow framework:

‣ **`Placeholder`**
‣ **`Const`**
‣ **`Add, Sub, Mul, Div, Minimum`** and **`Maximum`**
‣ **`BiasAdd`**
‣ **`Negative, Abs, Sqrt, Rsqrt, Pow, Exp`** and **`Log`**

> The NvUffParser supports **`Neg, Abs, Sqrt, Rsqrt, Exp`** and **`Log`** for const nodes only.

‣ **`FusedBatchNorm`**
‣ **`ReLU, TanH,`** and **`Sigmoid`**
‣ **`SoftMax`**
‣ **`Mean`**
‣ **`ConcatV2`**
‣ **`Reshape`**
‣ **`Transpose`**
‣ **`Conv2D`**
‣ **`DepthwiseConv2dNative`**
‣ **`ConvTranspose2D`**
‣ **`MaxPool`**
‣ **`AvgPool`**
‣ **`Pad`** is supported if followed by one of these TensorFlow layers: **`Conv2D`**, **`DepthwiseConv2dNative, MaxPool`**, and **`AvgPool`**

## ONNX

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

> **Restriction** Since the ONNX format is quickly developing, you may encounter a version mismatch between the model version and the parser version. The ONNX Parser shipped with TensorRT 5.0 RC supports ONNX IR (Intermediate Representation) version 0.0.3, opeset version 7.
>
> In general, the newer version of the ONNX Parser is designed to be backward compatible, therefore, encountering a model file produced by an earlier version of ONNX exporter should not cause a problem. There could be some exceptions when the

changes were not backward compatible. In this case, convert the earlier ONNX model file into a later supported version. For more information on this subject, see ONNX Model Opset Version Converter.

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

These are the operations that are supported in the ONNX framework:

- `Abs`
- `Add`
- `AveragePool`
- `BatchNormalization`
- `Ceil`
- `Clip`
- `Concat`
- `Constant`
- `Conv`
- `ConvTranspose`
- `DepthToSpace`
- `Div`
- `Dropout`
- `Elu`
- `Exp`
- `Flatten`
- `Floor`
- `Gemm`
- `GlobalAveragePool`
- `GlobalMaxPool`
- `HardSigmoid`
- `Identity`
- `InstanceNormalization`
- `LRN`
- `LeakyRelu`
- `Log`
- `LogSoftmax`
- `MatMul`

- **Max**
- **MaxPool**
- **Mean**
- **Min**
- **Mul**
- **Neg**
- **PRelu**
- **Pad**
- **Pow**
- **Reciprocal**
- **ReduceL1**
- **ReduceL2**
- **ReduceLogSum**
- **ReduceLogSumExp**
- **ReduceMax**
- **ReduceMean**
- **ReduceMin**
- **ReduceProd**
- **ReduceSum**
- **ReduceSumSquare**
- **Relu**
- **Reshape**
- **Selu**
- **Shape**
- **Sigmoid**
- **Size**
- **Softmax**
- **Softplus**
- **SpaceToDepth**
- **Split**
- **Squeeze**
- **Sub**
- **Sum**
- **Tanh**
- **TopK**
- **Transpose**
- **Unsqueeze**
- **Upsample**

## 8.2. Working With TensorFlow

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

▸ The end_to_end_tensorflow_mnist Python sample
▸ Generate TensorRT Engines from Tensorflow (or other UFF Compatible Frameworks)

### 8.2.1. Freezing A TensorFlow Graph

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

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

### 8.2.2. Freezing A Keras Model

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

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

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

### 8.2.3. Converting A Frozen Graph To UFF

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

```
convert-to-uff input_file [-o output_file] [-O output_node]
```

You can list the TensorFlow layers:

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

## 8.2.4. Working With TensorFlow RNN Weights

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

### 8.2.4.1. TensorFlow RNN Cells Supported In TensorRT

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

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

**TensorRT RNN Relu/Tanh Layer**

1.  `BasicRNNCell`

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

**TensorRT LSTM Layer**

1.  `BasicLSTMCell`

    a.  `forget_bias` must be set to `0` when creating an instance of this cell in TensorFlow. To support a non-zero forget bias, you need to preprocess the bias by adding the parameterized forget bias to the dumped TensorFlow forget biases.
    b.  This is a platform independent cell.

2.  `CudnnCompatibleLSTMCell`

    a.  Same condition for the forget bias applies to this cell as it does to the `BasicLSTMCell`.
    b.  TensorRT does not currently support peepholes so `use_peepholes` must be set to `False`.
    c.  This is a cuDNN compatible cell.

**TensorRT GRU Layer**

1.  `CudnnCompatibleGRUCell`

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

### 8.2.4.2. Maintaining Model Consistency Between TensorFlow And TensorRT

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

## 8.2.4.3. Workflow

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



Figure 12   TensorFlow RNN Workflow

## 8.2.4.4. Dumping The TensorFlow Weights

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

## 8.2.4.5. Loading Dumped Weights

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

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

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

## 8.2.4.6. Converting The Weights To A TensorRT Format

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

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

### 8.2.4.6.1. TensorFlow Checkpoint Storage Format

There are two possible TensorFlow checkpoint storage formats:

1. Platform independent format - separated by layer

   a. `Cell_i_kernel <Weights>`
   b. `Cell_i_bias <Weights>`

2. cuDNN compatible format - separated by input and recurrent

    a. **`Cell_i_Candidate_Input_kernel <Weights>`**
    b. **`Cell_i_Candidate_Hidden_kernel <Weights>`**

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

## 8.2.4.6.2. TensorFlow Kernel Tensor Storage Format

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

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

1. RNN relu/tanh:

    a. input gate (**`i`**)
2. LSTM:

    a. input gate (**`i`**), cell gate (**`c`**) , forget gate (**`f`**), output gate (**`o`**)
3. GRU:

    a. reset (**`r`**), update (**`u`**)

## 8.2.4.6.3. Kernel Weights Conversion To A TensorRT Format

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

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

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

## 8.2.4.6.4. TensorFlow Bias Weights Storage Format

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

## 8.2.4.6.5. Bias Tensor Conversion To TensorRT Format

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

## 8.2.4.7. `BasicLSTMCell` Example

### 8.2.4.7.1. BasicLSTMCell Kernel Tensor

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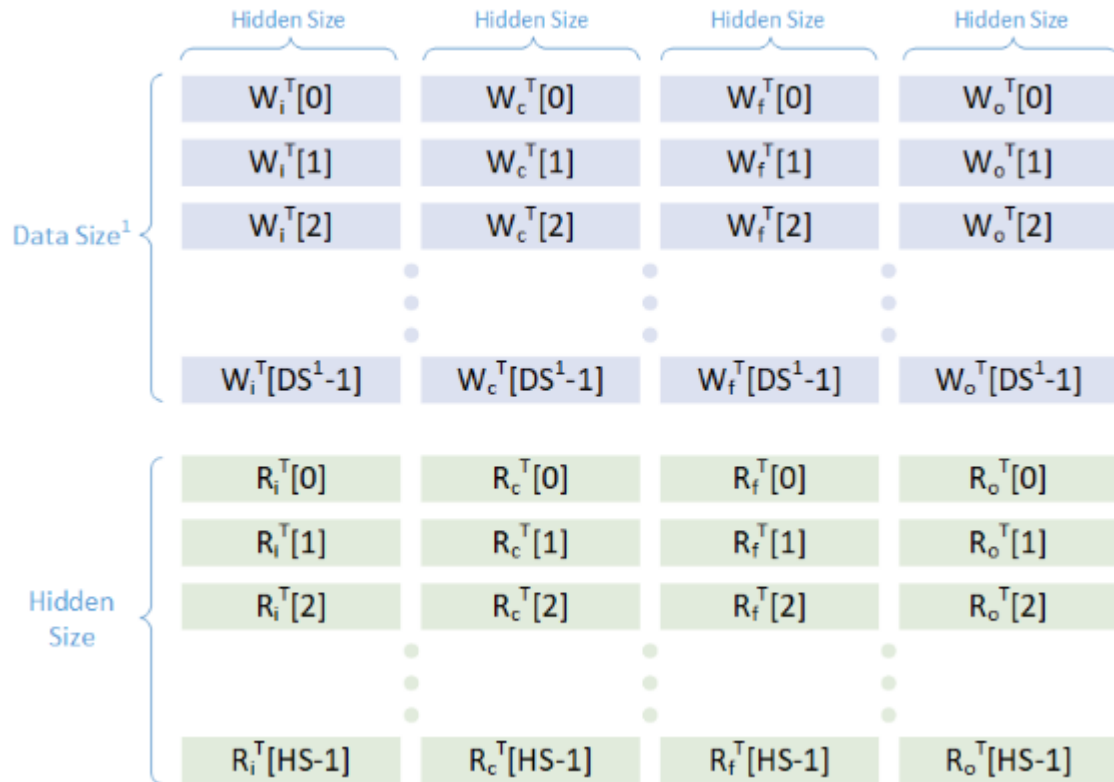


Figure 13  Tensors within a TensorFlow checkpoint

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

In Figure 13, `W` represents the input weights, `R` represents the hidden weights, `DS` represents the data size, and `HS` represents hidden size.

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

Applying the conversion process discussed earlier will result in the converted tensor shown in Figure 14.

Figure 14  Converted tensors

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

### 8.2.4.7.2. BasicLSTMCell Bias Tensor

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



Figure 15  Bias tensor stored format

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

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

### 8.2.4.8. Setting The Converted Weights And Biases

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

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

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

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

## 8.2.5. Preprocessing A TensorFlow Graph Using the Graph Surgeon API

The Graph Surgeon API, also known as **graphsurgeon**, allows you to transform TensorFlow graphs. Its capabilities are broadly divided into two categories:
**Search**
   The search functions allow you to find nodes in a TensorFlow graph.
**Manipulation**
   The manipulation functions allow you to modify, add, or remove nodes.

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

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

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

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

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

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

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

For a working **graphsurgeon** example, see sampleUffSSD for C++.

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

# 8.3. Working With PyTorch And Other Frameworks

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

▶ the network_api_pytorch_mnist Python sample
▶ Generate TensorRT Engines from Tensorflow (or other UFF Compatible Frameworks)

# Chapter 9.
# SAMPLES

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

## 9.1. C++ Samples

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

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

**Running C++ Samples**

If you installed TensorRT using the debian files, copy **/usr/src/tensorrt** to a new directory first before building the C++ samples. If you installed TensorRT using the tar file, then the samples are located in **{TAR_EXTRACT_PATH}/samples**. To build all the samples and then run one of the samples, use the following commands:

```
$ cd <samples_dir>
```

```
$ make -j4
$ cd ../bin
$ ./<sample_bin>
```

## 9.1.1. sampleMNIST

**What Does This Sample Do?**

The sampleMNIST sample demonstrates how to:

▶ Perform the basic setup and initialization of TensorRT

▶ Import a trained Caffe model using Caffe parser (see Importing A Caffe Model Using The C++ Parser API)

▶ Build an engine (see Building An Engine In C++)

▶ Serialize and deserialize the engine (see Serializing A Model In C++)

▶ Use the engine to perform inference on an input image (see Performing Inference In C++)

**Where Is This Sample Located?**

The sampleMNIST sample is installed in the **/usr/src/tensorrt/samples/ sampleMNIST** directory.

**Notes About This Sample:**

The Caffe model was trained on the MNIST dataset, where the dataset is from the NVIDIA DIGITS tutorial.

To verify whether the engine is operating correctly, sampleMNIST picks a 28x28 image of a digit at random and runs inference on it using the engine it created. The output of the network is a probability distribution on the digits, showing which digit is most probably that in the image.

An example of ASCII rendering of the input image with digit 8:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@%+-:    =@@@@@@@@@@@
@@@@@@@%=        -@@@**@@@@@@@
@@@@@@@     :%#@-#@@@.  #@@@@@@
@@@@@@*    +@@@@:*@@@   *@@@@@@
@@@@@@#    +@@@@ @@@%   @@@@@@@
@@@@@@.    :%@@.@@@.  *@@@@@@@
@@@@@@@-    =@@@@.  -@@@@@@@@
@@@@@@@@%:    +@-  :@@@@@@@@@
@@@@@@@@@@%.   :  -@@@@@@@@@
@@@@@@@@@@@@+    #@@@@@@@@@@
@@@@@@@@@@@@@+   :@@@@@@@@@@
@@@@@@@@@@@@@+    *@@@@@@@@@
@@@@@@@@@@@@@:  =  @@@@@@@@@
@@@@@@@@@@@@@  :@  @@@@@@@@@
@@@@@@@@@@@@@  -@  @@@@@@@@@
@@@@@@@@@@@@@# +@  @@@@@@@@@
@@@@@@@@@@@@@* ++  @@@@@@@@@
@@@@@@@@@@@@@*    *@@@@@@@@@
@@@@@@@@@@@@@#   =@@@@@@@@@@
@@@@@@@@@@@@@.  +@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```
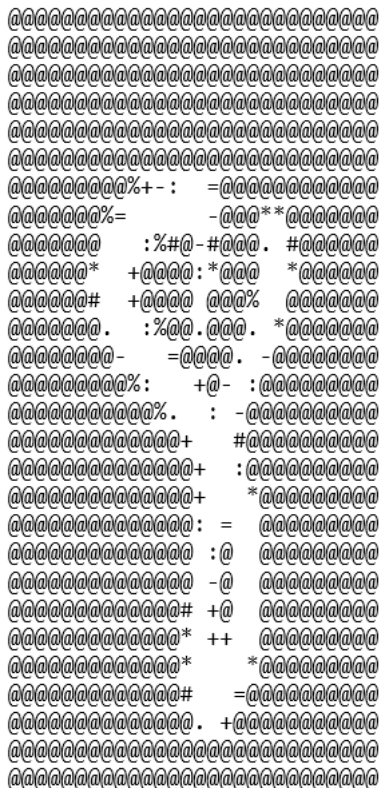
Figure 16    ASCII output

An example of the output from network, classifying the digit 8 from the above image:

```
0:
1:
2:
3:
4:
5:
6:
7:
8: **********
9:
```

Figure 17    Decision output

# 9.1.2. sampleMNISTAPI

**What Does This Sample Do?**
The sampleMNISTAPI sample is similar to sampleMNIST sample. Both of these samples use the same model, handle the same input, and expect similar output. In contrast to sampleMNIST, the sampleMNISTAPI demonstrates how to:

▸    Build a network by individually creating every layer

▸ Load the layers with theirs weights and connecting the layers by linking their inputs and outputs

**Where Is This Sample Located?**

The sampleMNISTAPI sample is installed in the **/usr/src/tensorrt/samples/ sampleMNISTAPI** directory.

**Notes About This Sample:**

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

**Notes About Weights:**

When you build a network by individually creating every layer, ensure you provide the per-layer weights to TensorRT in host memory. You will need to extract weights from their pre-trained model and deep learning framework and have these per-layer weights loaded in host memory to pass to TensorRT during network creation.

## 9.1.3. sampleUffMNIST

**What Does This Sample Do?**

The sampleUffMNIST sample demonstrates how to:

▸ Implement a TensorFlow model trained on the MNIST dataset

▸ Create the UFF Parser (see Importing From TensorFlow Using Python)

▸ Use the UFF Parser, register inputs and outputs, provide the dimensions and the order of the input tensor

▸ Load a trained TensorFlow model converted to UFF

▸ Build an engine (see Building An Engine In C++)

▸ Use the engine to perform inference (see Performing Inference In C++)

**Where Is This Sample Located?**

The sampleUffMNIST sample is installed in the **/usr/src/tensorrt/samples/ sampleUffMNIST** directory.

**Notes About This Sample:**

The TensorFlow model has been converted to UFF using the explanation described in Working With TensorFlow.

The UFF is designed to store neural networks as a graph. The NvUffParser that we use in this sample parses the format in order to create an inference engine based on that neural network.

With TensorRT, you can take a TensorFlow trained model, export it into a UFF protobuf file, and convert it to run in TensorRT. The TensorFlow to UFF converter creates an output file in a format called UFF which can then be read in TensorRT.

# 9.1.4. sampleOnnxMNIST

**What Does This Sample Do?**

The sampleOnnxMNIST sample demonstrates how to:

- ▸ Configure the ONNX parser
- ▸ Convert an MNIST network in ONNX format to a TensorRT network
- ▸ Build the engine and run inference using the generated TensorRT network
- ▸ Covers Importing An ONNX Model Using The C++ Parser API and Importing From ONNX Using Python

The sampleOnnxMNIST sample shows the conversion of an MNIST network in Open Neural Network Exchange (ONNX) format to a TensorRT network. ONNX is a standard for representing deep learning models that enable models to be transferred between frameworks. For more information about the ONNX format, see GitHub: ONNX. You can find a collection of ONNX networks at GitHub: ONNX Models. The network used in this sample can be found here.

**Where Is This Sample Located?**

The sampleOnnxMNIST sample is installed in the **/usr/src/tensorrt/samples/ sampleOnnxMNIST** directory.

## 9.1.4.1. Configuring The ONNX Parser

The **IOnnxConfig** class is the configuration manager class for the ONNX parser. The configuration parameters can be set by creating an object of this class and set the model file.

Set the appropriate ONNX model in the **config** object where **onnx_filename** is a **c** string of the path to the filename containing that model:

```
IOnnxConfig config;
config.setModelFileName(onnx_filename);
```

The **createONNXParser** method requires a **config** object as an argument:

```
nvonnxparser::IONNXParser* parser = nvonnxparser::createONNXParser(*config);
```

The ONNX model file is then passed onto the parser:

```
if (!parser->parse(onnx_filename, dataType))
{
string msg("failed to parse onnx file");
 gLogger->log(nvinfer1::ILogger::Severity::kERROR, msg.c_str());
     exit(EXIT_FAILURE);
}
```

To view additional information about the network, including layer information and individual layer dimensions, issue the following call:

```
config.setPrintLayerInfo(true)
parser->reportParsingInfo();
```

## 9.1.4.2. Converting The ONNX Model To A TensorRT Network

The parser can convert the ONNX model to a TensorRT network which can be used for inference:

```
if (!parser->convertToTRTNetwork()) {
string msg("ERROR, failed to convert onnx network into TRT network");
gLogger->log(nvinfer1::ILogger::Severity::kERROR, msg.c_str());
      exit(EXIT_FAILURE);
    }
```

To get the TensorRT network, issue the following call:

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

After the TensorRT network is built from the model, you can build the TensorRT engine and run inference.

## 9.1.4.3. Building The Engine And Running Inference

Before you can run inference, you must first build the engine. To build the engine, create the builder and pass a logger created for TensorRT which is used for reporting errors, warnings and informational messages in the network:

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

To build the engine from the generated TensorRT network, issue the following call:

```
nvinfer1::ICudaEngine* engine = builder->buildCudaEngine(*network);
```

To run inference using the created engine, see Performing Inference In C++ or Performing Inference In Python.

> It's important to preprocess the data and convert it to the format accepted by the network. In this example, the sample input is in PGM (portable graymap) format. The model expects an input of image `1x28x28` scaled to between `[0,1]`.

After you build the engine, verify that the engine is running properly by confirming the output is what you expected. The output format of this sample should be the same as the output of the sampleMNIST described in sampleMNIST.

## 9.1.5. sampleGoogleNet

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

▶ Use FP16 mode in TensorRT

▶ Use TensorRT **Half2Mode**
▶ Use layer-based profiling

**Where Is This Sample Located?**
The sampleGoogleNet sample is installed in the **/usr/src/tensorrt/samples/sampleGoogleNet** directory.

## 9.1.5.1. Configuring The Builder

The sampleGoogleNet sample builds a network based on a saved Caffe model and network description. For more information, see Importing A Caffe Model Using The C++ Parser API or Importing From Caffe Using Python.

This sample uses optimized FP16 mode (see Enabling FP16 Inference Using C++ or Enabling FP16 Inference Using Python). To use **Half2Mode**, two additional steps are required:

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

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

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

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

## 9.1.5.2. Profiling

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

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

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

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

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

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

# 9.1.6. sampleCharRNN

**What Does This Sample Do?**

The sampleCharRNN sample demonstrates how to generate a simple RNN based on the charRNN network using the Penn Treebank (PTB) dataset. For more information about character level modeling, see char-rnn.

**Where Is This Sample Located?**

The sampleCharRNN sample is installed in the **/usr/src/tensorrt/samples/sampleCharRNN** directory.

**Notes About This Sample:**

Use the TensorRT API documentation to familiarize yourself with the following layers:

▸ RNNv2 layer

  ▸ Weights are set for each gate and layer individually.
  ▸ The input format for RNNv2 is BSE (Batch, Sequence, Embedding).
▸ MatrixMultiply
▸ ElementWise
▸ TopK

## 9.1.6.1. Network Configuration

The CharRNN network is a fairly simple RNN network. The input into the network is a single character that is embedded into a vector of size 512. This embedded input is then supplied to a RNN layer containing two stacked LSTM cells. The output from the RNN layer is then supplied to a fully connected layer, which can be represented in TensorRT by a Matrix Multiply layer followed by an ElementWise sum layer. Constant layers are used to supply the weights and biases to the Matrix Multiply and ElementWise Layers, respectively. A TopK operation is then performed on the output of the ElementWise sum layer where `K = 1` to find the next predicted character in the sequence. For more information about these layers, see the TensorRT API documentation.

### 9.1.6.1.1. RNNv2 Layer Setup

The first layer in the network is an RNN layer. This is added and configured in the `addRNNv2Layer()` function. This layer consists of the following configuration parameters:

**Operation**

This defines the operation of the RNN cell. Supported operations are currently `relu, LSTM, GRU`, and `tanh`.

**Direction**

This defines whether the RNN is unidirectional or bidirectional (BiRNN).

**Input mode**

This defines whether the first layer of the RNN carries out a matrix multiply (linear mode), or the matrix multiply is skipped (skip mode).

For the purpose of the CharRNN network, we will be using a linear, unidirectional LSTM cell containing **LAYER_COUNT** number of stacked layers. The code below shows how to create this RNNv2 layer.

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

> For the RNNv2 layer, weights and bias need to be set separately. For more information, see RNNv2 Layer - Optional Inputs.

For more information, see the TensorRT API documentation.

## 9.1.6.1.2. RNNv2 Layer - Optional Inputs

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

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

## 9.1.6.1.3. MatrixMultiply Layer Setup

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

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

For more information, see the TensorRT API documentation.

## 9.1.6.1.4. ElementWise Layer Setup

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

```
auto fcbias = network->addConstant(Dims2(VOCAB_SIZE, 1), weightMap[FCB_NAME]);
```

```
auto addBiasLayer = network->addElementWise(
*matrixMultLayer->getOutput(0),
*fcbias->getOutput(0), ElementWiseOperation::kSUM);
assert(addBiasLayer != nullptr);
addBiasLayer-getOutput(0)->setName("Add Bias output");
```

For more information, see the TensorRT API documentation.

### 9.1.6.1.5. TopK Layer Setup

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

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

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

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

For more information, see the TensorRT API documentation.

### 9.1.6.1.6. Marking The Network Outputs

After the network is defined, mark the required outputs. RNN output tensors that are not marked as network outputs or used as inputs to another layer are dropped.

```
network->markOutput(*pred->getOutput(1));
pred->getOutput(1)->setType(DataType::kINT32);
rnn->getOutput(1)->setName(HIDDEN_OUT_BLOB_NAME);
network->markOutput(*rnn->getOutput(1));
if (rnn->getOperation() == RNNOperation::kLSTM)
{
rnn->getOutput(2)->setName(CELL_OUT_BLOB_NAME);
network->markOutput(*rnn->getOutput(2));
};
```

## 9.1.6.2. RNNv2 Workflow - From TensorFlow To TensorRT

The following sections provide an end-to-end walkthrough of how to train your model in TensorFlow and convert the weights into a format that TensorRT can use.

### 9.1.6.2.1. Training A CharRNN Model With TensorFlow

TensorFlow has a useful RNN Tutorial which can be used to train a word level model. Word level models learn a probability distribution over a set of all possible word sequence. Since our goal is to train a char level model, which learns a probability distribution over a set of all possible characters, a few modifications will need to be made to get the TensorFlow sample to work. These modifications can be seen here.

There are also multiple GitHub repositories that contain CharRNN implementations that will work out of the box. Tensorflow-char-rnn is one such implementation.

### 9.1.6.2.2. Exporting Weights From A TensorFlow Model Checkpoint

A python script **/usr/src/tensorrt/samples/common/dumpTFWts.py** has been provided to extract the weights from the model checkpoint files that are created during training. Use **dumpTFWts.py -h** for directions on the usage of the script.

### 9.1.6.2.3. Loading And Converting Weights Format

After the TensorFlow weights have been exported into a single **WTS** file, the next step is to load the weights and convert them into the TensorRT weights format. This is done by the **loadWeights** and then the **convertRNNWeights** and **convertRNNBias** functions. The functions contain detailed descriptions of the loading and conversion process. You can use those as guides in case you need to write your own conversion functions. After the conversion has taken place, the memory holding the converted weights is added to the weight map so that it can be deallocated once the engine has been built.

```
Weights rnnwL0 = convertRNNWeights(weightMap[RNNW_L0_NAME]);
Weights rnnbL0 = convertRNNBias(weightMap[RNNB_L0_NAME]);
Weights rnnwL1 = convertRNNWeights(weightMap[RNNW_L1_NAME]);
Weights rnnbL1 = convertRNNBias(weightMap[RNNB_L1_NAME]);

...
weightMap["rnnwL0"] = rnnwL0;
weightMap["rnnbL0"] = rnnbL0;
weightMap["rnnwL1"] = rnnwL1;
weightMap["rnnbL1"] = rnnbL1;
```

### 9.1.6.2.4. RNNv2: Setting Weights And Bias

After the conversion to the TensorRT format, the RNN weights and biases are stored in their respective contiguous arrays. They are stored in the format of $[ W_Lf, W_Li, W_LC, W_LO, R_Lf, R_Li, R_LC, R_LO ]$, where:

**W**

The weights for the input.

**R**

The weights for the recurrent input.

**f**

Corresponds to the forget gate.

**i**

Corresponds to the input gate.

**c**

Corresponds to the cell gate.

**o**

Corresponds to the output gate.

The code below takes advantage of this memory layout and iterates over the two layers and the eight gates to extract and set the correct gate weights and gate biases for the RNN layer.

```
for (int gateIndex = 0; gateIndex < NUM_GATES; gateIndex++)
```

```
{
    // extract weights and bias for a given gate and layer
    Weights gateWeightL0{.type = dataType,
.values = (void*)(wtsL0 + kernelOffset),
.count = DATA_SIZE * HIDDEN_SIZE};
    Weights gateBiasL0{.type = dataType,
.values = (void*)(biasesL0 + biasOffset),
.count = HIDDEN_SIZE};
    Weights gateWeightL1{.type = dataType,
.values = (void*)(wtsL1 + kernelOffset),
.count = DATA_SIZE * HIDDEN_SIZE};
    Weights gateBiasL1{.type = dataType,
.values = (void*)(biasesL1 + biasOffset),
.count = HIDDEN_SIZE};

    // set weights and bias for given gate
    rnn->setWeightsForGate(0, gateOrder[gateIndex % 4],
(gateIndex < 4), gateWeightL0);
    rnn->setBiasForGate(0, gateOrder[gateIndex % 4],
(gateIndex < 4), gateBiasL0);
    rnn->setWeightsForGate(1, gateOrder[gateIndex % 4],
(gateIndex < 4), gateWeightL1);
    rnn->setBiasForGate(1, gateOrder[gateIndex % 4],
(gateIndex < 4), gateBiasL1);

    // Update offsets
    kernelOffset = kernelOffset + DATA_SIZE * HIDDEN_SIZE;
    biasOffset = biasOffset + HIDDEN_SIZE;
}
```

### 9.1.6.3. Seeding The Network

After the network is built, it is seeded with preset inputs so that the RNN can start generating data. Inside **stepOnce**, the output states are preserved for use as inputs on the next timestep.

```
for (auto &a : input)
{
    std::copy(static_cast<const float*>(embed.values) +
 char_to_id[a]*DATA_SIZE,
            static_cast<const float*>(embed.values) + char_to_id[a]*DATA_SIZE +
 DATA_SIZE,
            data[INPUT_IDX]);
    stepOnce(data, output, buffers, indices, stream, context);
    cudaStreamSynchronize(stream);

    // Copy Ct/Ht to the Ct-1/Ht-1 slots.
    std::memcpy(data[HIDDEN_IN_IDX], data[HIDDEN_OUT_IDX],
 gSizes[HIDDEN_IN_IDX] * sizeof(float));
    std::memcpy(data[CELL_IN_IDX], data[CELL_OUT_IDX], gSizes[CELL_IN_IDX] *
 sizeof(float));

    genstr.push_back(a);
}
// Extract first predicted character
uint32_t predIdx = *reinterpret_cast<uint32_t*>(data[OUTPUT_IDX]);
genstr.push_back(id_to_char[predIdx]);
```

### 9.1.6.4. Generating Data

The following code is similar to the seeding code, however, this code generates an output character based on the output probability distribution. The following code

simply selects the character with the highest probability. The final result is stored in **genstr**.

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

    stepOnce(data, output, buffers, indices, stream, context);
    cudaStreamSynchronize(stream);

    // Copy Ct/Ht to the Ct-1/Ht-1 slots.
    std::memcpy(data[HIDDEN_IN_IDX], data[HIDDEN_OUT_IDX],
 gSizes[HIDDEN_IN_IDX] * sizeof(float));
    std::memcpy(data[CELL_IN_IDX], data[CELL_OUT_IDX], gSizes[CELL_IN_IDX] *
 sizeof(float));

uint32_t predIdx = *(output);
    genstr.push_back(id_to_char[predIdx]);
}
```

## 9.1.7. sampleINT8

**What Does This Sample Do?**

The sampleINT8 sample provides the steps involved when performing inference in 8-bit integer (INT8).

> INT8 inference is available only on GPUs with compute capability 6.1 or 7.x.

The sampleINT8 sample demonstrates how to:

▸ Perform INT8 calibration

▸ Perform INT8 inference

▸ Calibrate a network for execution in INT8

▸ Cache the output of the calibration to avoid repeating the process

▸ Repo your own experiments with Caffe in order to validate your results on ImageNet networks

**Where Is This Sample Located?**

The sampleINT8 sample is installed in the **/usr/src/tensorrt/samples/ sampleINT8** directory.

**Notes About This Sample:**

INT8 engines are built from 32-bit network definitions and require significantly more investment than building a 32-bit or 16-bit engine. In particular, the TensorRT builder must perform a process called calibration to determine how best to represent the weights and activations as 8-bit integers.

The sample is accompanied by the MNIST training set, but may also be used to calibrate and score other networks. To run the sample on MNIST, use the command line:

```
./sample_int8 mnist
```

## 9.1.7.1. Defining The Network

Defining a network for INT8 execution is exactly the same as for any other precision. Weights should be imported as FP32 values, and TensorRT will calibrate the network to find appropriate quantization factors to reduce the network to INT8 precision. This sample imports the network using the NvCaffeParser:

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

## 9.1.7.2. Building The Engine

Calibration is an additional step required when building networks for INT8. The application must provide TensorRT with sample input. TensorRT will then perform inference in FP32 and gather statistics about intermediate activation layers that it will use to build the reduce precision INT8 engine.

### 9.1.7.2.1. Calibrating The Network

The application must specify the calibration set and parameters by implementing the IInt8Calibrator interface. Because calibration is an expensive process that may need to run multiple times, the interface provides methods for caching intermediate values. Follow this sample to learn more about how to configure a calibrator object.

### 9.1.7.2.2. Calibration Set

Calibration must be performed using images representative of those which will be used at runtime. Since the sample is based around Caffe, any image preprocessing that Caffe would perform prior to running the network (such as scaling, cropping, or mean subtraction) will be done in Caffe and captured as a set of files. The sample uses a utility class (**BatchStream**) to read these files and create appropriate input for calibration. Generation of these files is discussed in Batch Files For Calibration.

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

```
bool getBatch(void* bindings[], const char* names[], int
 nbBindings) override
{
    if (!mStream.next())
        return false;
```

```
    CHECK(cudaMemcpy(mDeviceInput, mStream.getBatch(),
 mInputCount * sizeof(float), cudaMemcpyHostToDevice));
    assert(!strcmp(names[0], INPUT_BLOB_NAME));
    bindings[0] = mDeviceInput;
    return true;
}
```

For each input tensor, a pointer to input data in GPU memory must be written into the bindings array. The names array contains the names of the input tensors. The position for each tensor in the bindings array matches the position of its name in the names array. Both arrays have size **nbBindings**.

> The calibration set must be representative of the input provided to TensorRT at runtime; for example, for image classification networks, it should not consist of images from just a small subset of categories. For ImageNet networks, around 500 calibration images is adequate.

## 9.1.7.2.3. Loading A Calibration File

A calibration file stores activation scales for each network layer. The calibration file is called **CalibrationTable<NetworkName>**, where **<NetworkName>** is the name of your network, for example **mnist**. The file is located in the **TensorRT-x.x.x.x/data/mnist** directory, where **x.x.x.x** is your installed version of TensorRT.

If the **CalibrationTable** file is not found, the builder will run the calibration algorithm again to create it. The **CalibrationTable** contents include:

```
1
data: 3c000889
pool2: 3d9ccc94
ip1: 3daeff07
prob: 3c010a14
conv2: 3dd33169
pool1: 3c88e7e3
ip2: 3e7d50ec
conv1: 3c8954be
```

Where **1** stands for the calibration algorithm, for example, Entropy Calibration.

The **CalibrationTable** file is generated during the build phase while running the calibration algorithm. Specifically, to create the calibration file, you first need to provide a calibrator object and pass it to the builder. The calibrator object should be configured to use the calibration image batches. During the build phase, the builder will create the calibration file using the calibrator object.

After the calibration file is created, the file must get loaded. You cannot manually load a calibration file using an API, the builder first checks whether the file exists. If it does, it will not calibrate again and instead will load that same calibration file for every runtime. Therefore, the calibration file needs to be created only once.

## 9.1.7.3. Configuring The Builder

There are two additional methods to call on the builder:

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

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

## 9.1.7.4. Running The Engine

After the network has been built, it can be used just like an FP32 network, for example, inputs and outputs remain in 32-bit floating point.

## 9.1.7.5. Verifying The Output

This sample outputs Top-1 and Top-5 metrics for both FP32 and INT8 precision, as well as for FP16 if it is natively supported by the hardware. These numbers should be within 1%.

## 9.1.7.6. Batch Files For Calibration

The sampleINT8 sample uses batch files in order to calibrate for the INT8 data. The INT8 batch file is a binary file containing a set of **N** images, whose format is as follows:

▶ Four 32-bit integer values representing **{N,C, H, W}** representing the number of images **N** in the file, and the dimensions **{C, H, W}** of each image.
▶ **N** 32-bit floating point data blobs of dimensions **{C, H, W}** that are used as inputs to the network.

### 9.1.7.6.1. Generating Batch Files For Caffe Users

Calibration requires that the images passed to the calibrator are in the same format as those that will be passed to TensorRT at runtime. For developers using Caffe for training, or who can easily transfer their network to Caffe, a supplied patchset supports capturing images after image preprocessing.

These instructions are provided so that users can easily use the sample code to test accuracy and performance on classification networks. In typical production use cases, applications will have such preprocessing already implemented, and should integrate with the calibrator directly.

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

1. Apply the patch. The patch can be applied by going to the root directory of the Caffe source tree and applying the patch with the command:

   ```
   patch -p1 < int8_caffe.patch
   ```
2. Rebuild Caffe and set the environment variable **TENSORRT_INT8_BATCH_DIRECTORY** to the location where the batch files are to be generated.

After training for 1000 iterations, there are 1003 batch files in the directory specified. This occurs because Caffe preprocesses three batches in advance of the current iteration.

These batch files can then be used with the `BatchStream` and `Int8Calibrator` to calibrate the data for INT8.

> When running Caffe to generate the batch files, the training prototxt, and not the deployment prototxt, is required to be used.

The following example depicts the sequence of commands to run `./sample_int8 mnist` with Caffe generated batch files.

1. Navigate to the samples data directory and create an INT8 `mnist` directory:

```
cd <TensorRT>/samples/data
mkdir -p int8/mnist
cd int8/mnist
```

> If Caffe is not installed anywhere, ensure you clone, checkout, patch, and build Caffe at the specific commit:
>
> ```
> git clone https://github.com/BVLC/caffe.git
> cd caffe
> git checkout 473f143f9422e7fc66e9590da6b2a1bb88e50b2f
> patch -p1 < <TensorRT>/samples/mnist/int8_caffe.patch
> mkdir build
> pushd build
> cmake -DUSE_OPENCV=FALSE -DUSE_CUDNN=OFF ../
> make -j4
> popd
> ```

2. Download the `mnist` dataset from Caffe and create a link to it:

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

3. Set the directory to store the batch data, execute Caffe, and link the `mnist` files:

```
mkdir batches
export TENSORRT_INT8_BATCH_DIRECTORY=batches
caffe/build/tools/caffe test -gpu 0 -iterations 1000 -model examples/mnist/
lenet_train_test.prototxt -weights
<TensorRT>/samples/mnist/mnist.caffemodel
ln -s <TensorRT>/samples/mnist/mnist.caffemodel .
ln -s <TensorRT>/samples/mnist/mnist.prototxt .
```

4. Execute sampleINT8 from the `bin` directory after being built with the following command:

```
 ./sample_int8 mnist
```

## 9.1.7.6.2. Generating Batch Files For Non-Caffe Users

For developers that are not using Caffe, or cannot easily convert to Caffe, the batch files can be generated via the following sequence of steps on the input training data.

1. Subtract out the normalized mean from the dataset.

2. Crop all of the input data to the same dimensions.

3. Split the data into batch files where each batch file has **N** preprocessed images and labels.

4. Generate the batch files based on the format specified in Batch Files for Calibration.

The following example depicts the sequence of commands to run **./sample_int8 mnist** without Caffe.

1. Navigate to the samples data directory and create an INT8 **mnist** directory:

```
cd <TensorRT>/samples/data
mkdir -p int8/mnist/batches
cd int8/mnist
ln -s <TensorRT>/samples/mnist/mnist.caffemodel .
ln -s <TensorRT>/samples/mnist/mnist.prototxt .
```

2. Copy the generated batch files to the **int8/mnist/batches/** directory.

3. Execute sampleINT8 from the **bin** directory after being built with the command **./sample_int8 mnist**.

```
./sample_int8 mnist
```

# 9.1.8. samplePlugin

**What Does This Sample Do?**

The samplePlugin demonstrates how to add a Custom layer to TensorRT. This sample implements the MNIST model with the difference that the final FullyConnected layer is replaced by a Custom layer. To read more information about MNIST, see sampleMNIST, sampleMNISTAPI, and sampleUffMNIST.

The samplePlugin sample demonstrates how to:

▸ Define a Custom layer that supports multiple data formats

▸ Define a Custom layer that can be serialized and deserialized

▸ Enable a Custom layer in NvCaffeParser

**Where Is This Sample Located?**

The samplePlugin sample is installed in the **/usr/src/tensorrt/samples/samplePlugin** directory.

**Notes About This Sample:**

The Custom layer implements the FullyConnected layer using *gemm* routines (Matrix Multiplication) in cuBLAS, and tensor addition in cuDNN (bias offset). This sample illustrates the definition of the **FCPlugin** for the Custom layer, and the integration with NvCaffeParser.

## 9.1.8.1. Defining The Network

The **FCPlugin** redefines the FullyConnected layer, which in this case has a single output. Accordingly, **getNbOutputs** returns **1** and **getOutputDimensions** includes validation checks and returns the dimensions of the output:

```
Dims getOutputDimensions(int index, const Dims* inputDims,
                         int nbInputDims) override
{
    assert(index == 0 && nbInputDims == 1 &&
           inputDims[0].nbDims == 3);
    assert(mNbInputChannels == inputDims[0].d[0] *
                               inputDims[0].d[1] *
                               inputDims[0].d[2]);
    return DimsCHW(mNbOutputChannels, 1, 1);
}
```

## 9.1.8.2. Enabling Custom Layers In NvCaffeParser

The model is imported using NvCaffeParser (see Importing A Caffe Model Using The C ++ Parser API and Using Custom Layers When Importing A Model From A Framework). To use the **FCPlugin** implementation for the FullyConnected layer, a plugin factory is defined which recognizes the name of the FullyConnected layer (inner product **ip2** in Caffe).

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

The factory can then instantiate **FCPlugin** objects as directed by the parser. The **createPlugin** method receives the layer name, and a set of weights extracted from the Caffe model file, which are then passed to the plugin constructor. Since the lifetime of the weights and that of the newly created plugin are decoupled, the plugin makes a copy of the weights in the constructor.

```
virtual nvinfer1::IPlugin* createPlugin(const char* layerName, const
 nvinfer1::Weights* weights, int nbWeights) override
{
    …
    mPlugin =
      std::unique_ptr<FCPlugin>(new FCPlugin(weights,nbWeights));

    return mPlugin.get();
}
```

## 9.1.8.3. Building The Engine

**FCPlugin** does not need any scratch space, therefore, for building the engine, the most important methods deal with the formats supported and the configuration. **FCPlugin** supports two formats: NCHW in both single and half precision as defined in the **supportsFormat** method.

```
bool supportsFormat(DataType type, PluginFormat format) const override
{
    return (type == DataType::kFLOAT || type == DataType::kHALF) &&
           format == PluginFormat::kNCHW;
}
```

Supported configurations are selected in the building phase. The builder selects a configuration with the networks **configureWithFormat()** method, to give it a chance

to select an algorithm based on its inputs. In this example, the inputs are checked to ensure they are in a supported format, and the selected format is recorded in a member variable. No other information needs to be stored in this simple case; in more complex cases, you may need to do so or even choose an ad-hoc algorithm for the given configuration.

```
void configureWithFormat(..., DataType type, PluginFormat format, ...) override
{
    assert((type == DataType::kFLOAT || type == DataType::kHALF) &&
            format == PluginFormat::kNCHW);
    mDataType = type;
}
```

The configuration takes place at build time, therefore, any information or state determined here that is required at runtime should be stored as a member variable of the plugin, and serialized and deserialized.

## 9.1.8.4. Serializing And Deserializing

Fully complaint plugins support serialization and deserialization, as described in Serializing A Model In C++. In the example, **FCPlugin** stores the number of channels and weights, the format selected, and the actual weights. The size of these variables makes up for the size of the serialized image; the size is returned by **getSerializationSize**:

```
virtual size_t getSerializationSize() override
{
    return sizeof(mNbInputChannels) + sizeof(mNbOutputChannels) +
            sizeof(mBiasWeights.count) + sizeof(mDataType) +
            (mKernelWeights.count + mBiasWeights.count) *
            type2size(mDataType);
}
```

Eventually, when the engine is serialized, these variables are serialized, the weights converted is needed, and written on a buffer:

```
virtual void serialize(void* buffer) override
{
    char* d = static_cast<char*>(buffer), *a = d;
    write(d, mNbInputChannels);
    ...
    convertAndCopyToBuffer(d, mKernelWeights);
    convertAndCopyToBuffer(d, mBiasWeights);
    assert(d == a + getSerializationSize());
}
```

Then, when the engine is deployed, it is deserialized. As the runtime scans the serialized image, when a plugin image is encountered, it create a new plugin instance via the factory. The plugin object created during deserialization (shows below using **new**) is destroyed when the engine is destroyed by calling **FCPlugin::destroy()**.

```
IPlugin* createPlugin(...) override
{
    …

    return new FCPlugin(serialData, serialLength);
}
```

In the same order as in the serialization, the variables are read and their values restored. In addition, at this point the weights have been converted to selected format and can be stored directly on the device.

```
FCPlugin(const void* data, size_t length)
{
    const char* d = static_cast<const char*>(data), *a = d;
    read(d, mNbInputChannels);
    ...
    deserializeToDevice(d, mDeviceKernel,
                        mKernelWeights.count*type2size(mDataType));
    deserializeToDevice(d, mDeviceBias,
                        mBiasWeights.count*type2size(mDataType));
    assert(d == a + length);
}
```

## 9.1.8.5. Resource Management And Execution

Before a custom layer is executed, the plugin is initialized. This is where resources are held for the lifetime of the plugin and can be acquired and initialized. In this example, weights are kept in CPU memory at first, so that during the build phase, for each configuration tested, weights can be converted to the desired format and then copied to the device in the initialization of the plugin. The method **initialize** creates the required cuBLAS and cuDNN handles, sets up tensor descriptors, allocates device memory, and copies the weights to device memory. Conversely, **terminate** destroys the handles and frees the memory allocated on the device.

```
int initialize() override
{
    CHECK(cudnnCreate(&mCudnn));
    CHECK(cublasCreate(&mCublas));
    …
    if (mKernelWeights.values != nullptr)
        convertAndCopyToDevice(mDeviceKernel, mKernelWeights);
    …
}
```

The core of the plugin is **enqueue**, which is used to execute the custom layer at runtime. The **call** parameters include the actual batch size, inputs, and outputs. The handles for cuBLAS and cuDNN operations are placed on the given stream; then, according to the data type and format configured, the plugin executes in single or half precision.

> The two handles are part of the plugin object, therefore, the same engine cannot be executed concurrently on multiple streams. In order to enable multiple streams of execution, plugins must be re-entrant and handle stream-specific data accordingly.

```
virtual int enqueue(int batchSize, const void*const * inputs, void**
 outputs, ...) override
{
    ...
    cublasSetStream(mCublas, stream);
    cudnnSetStream(mCudnn, stream);
    if (mDataType == DataType::kFLOAT)
    {...}
    else
    {
        CHECK(cublasHgemm(mCublas, CUBLAS_OP_T, CUBLAS_OP_N,
```

```
                          mNbOutputChannels, batchSize,
                          mNbInputChannels, &oneh,
                          mDeviceKernel), mNbInputChannels,
                          inputs[0], mNbInputChannels, &zeroh,
                          outputs[0], mNbOutputChannels));
    }
    if (mBiasWeights.count)
    {
        cudnnDataType_t cudnnDT = mDataType == DataType::kFLOAT ?
                              CUDNN_DATA_FLOAT : CUDNN_DATA_HALF;
        ...
    }
    return 0;
}
```

The plugin object created in the sample is cloned by each of the network, builder, and engine by calling the **FCPlugin::clone()** method. The **clone()** method calls the plugin constructor and can also clone plugin parameters, if necessary.

```
IPluginExt* clone()
    {
        return new FCPlugin(&mKernelWeights, mNbWeights, mNbOutputChannels);
    }
```

The cloned plugin objects are deleted when the network, builder, or engine are destroyed. This is done by invoking the **FCPlugin::destroy()** method.

```
void destroy() { delete this; }
```

## 9.1.9. sampleNMT

**What Does This Sample Do?**

sampleNMT is a highly modular sample for inferencing using C++ and TensorRT API so that you can consider using it as a reference point in your projects. Neural Machine Translation (NMT) using sequence to sequence (seq2seq) models has garnered a lot of attention and is used in various NMT frameworks.

The sampleNMT sample demonstrates how to:

▶ Create an attention based seq2seq type NMT inference engine using a checkpoint from TensorFlow

▶ Convert trained weights using Python and import trained weights data into TensorRT

▶ Build relevant engines and run inference using the generated TensorRT network

▶ Use layers, such as:

  **RNNv2**

  The RNNv2 layer is used in the **lstm_encoder.cpp** and **lstm_decoder.cpp** files.

  **Constant**

  The Constant layer is used in the **slp_attention.cpp**, **slp_embedder.cpp** and **slp_projection.cpp** files.

**MatrixMultiply**

The MatrixMultiply layer is used in the `context.cpp`, `multiplicative_alignment.cpp`, `slp_attention.cpp`, and `slp_projection.cpp` files.

**Shuffle**

The Shuffle layer is used in the `lstm_encoder.cpp` and `lstm_decoder.cpp` files.

**RaggedSoftmax**

The RaggedSoftmax layer is used in the `context.cpp` file.

**TopK**

The TopK layer is used in the `softmax_likelihood.cpp` file.

**Gather**

The Gather layer is used in the `slp_embedder.cpp` file.

### Where Is This Sample Located?

The sampleNMT sample is installed in the `tensorrt/samples/sampleNMT` directory. For more information about how to run the sample, see the `README.txt` file in the `samples/sampleNMT/` directory.

## 9.1.9.1. Overview

At a high level, the basic architecture of the NMT model consists of two sides: an encoder and a decoder. Incoming sentences are translated into sequences of words in a fixed vocabulary. The incoming sequence goes through the encoder and is transformed by a network of Recurrent Neural Network (RNN) layers into an internal state space that represents a language-independent "meaning" of the sentence. The decoder works the opposite way, transforming from the internal state space back into a sequence of words in the output vocabulary.

### Encoding And Embedding

The encoding process requires a fixed vocabulary of words from the source language. Words not appearing in the vocabulary are replaced with an `UNKNOWN` token. Special symbols also represent `START-OF-SENTENCE` and `END-OF-SENTENCE`. After the input is finished, a `START-OF-SENTENCE` is fed in to mark the switch to decoding. The decoder will then produce the `END-OF-SENTENCE` symbol to indicate it is finished translating.

Vocabulary words are not just represented as single numbers, they are encoded as word vectors of a fixed size. The mapping from vocabulary word to embedding vector is learned during training.

### Attention

Attention mechanisms sit between the encoder and decoder and allow the network to focus on one part of the translation task at a time. It is possible to directly connect the

encoding and decoding stages but this would mean the internal state representing the meaning of the sentence would have to cover sentences of all possible lengths at once.

This sample implements Luong attention. In this model, at each decoder step the target hidden state is combined with all source states using the attention weights. A scoring function weighs each contribution from the source states. The attention vector is then fed into the next decoder stage as an input.

### Beam Search And Projection

There are several ways to organize the decode stage. The output of the RNN layer is not a single word. The simplest method, is to choose the most likely word at each time step, assume that is the correct output, and continue until the decoder generates the **END-OF-SENTENCE** symbol.

A better way to perform the decoding is to keep track of multiple candidate possibilities in parallel and keep updating the possibilities with the most likely sequences. In practice, a small fixed size of candidates works well. This method is called beam search. The beam width is the number of simultaneous candidate sequences that are in consideration at each time step.

As part of beam search we need a mechanism to convert output states into probability vectors over the vocabulary. This is accomplished with the projection layer using a fixed dense matrix.

For more information related to SampleNMT, see Creating A Network Definition In C++, Working With Deep Learning Frameworks, and Enabling FP16 Inference Using C++.

## 9.1.9.2. Preparing The Data

The NMT sample can be run with pre-trained weights. Link to the weights in the correct format can be found in the **samples/sampleNMT/README.txt** file.

Running the sample also requires text and vocabulary data. For the De-En model, the data can be fetched and processed using the script: wmt16_en_de.sh. Running this script may take some time, since it prepares 4.5M samples for training as well as inference.

Run the script **wmt16_de_en.sh** and collect the following files into a directory:

▶ **newstest2015.tok.bpe.32000.de**
▶ **newstest2015.tok.bpe.32000.en**
▶ **vocab.bpe.32000.de**
▶ **vocab.bpe.32000.en**

The weights **.bin** files from the link in the **README.txt** should be put in a subdirectory named **weights** in this directory.

In the event that the data files change, as of March 26, 2018 the MD5SUM for the data files are:

```
3c0a6e29d67b081a961febc6e9f53e4c  newstest2015.tok.bpe.32000.de


875215f2951b21a5140e4f3734b47d6c  newstest2015.tok.bpe.32000.en


c1d0ca6d4994c75574f28df7c9e8253f  vocab.bpe.32000.de


c1d0ca6d4994c75574f28df7c9e8253f  vocab.bpe.32000.en
```

## 9.1.9.3. Running The Sample

The sample executable is located in the **tensorrt/bin** directory. Running the sample requires pre-trained weights and the data files mentioned in Preparing The Data. After the data directory is setup, pass the location of the data directory to the sample with the following option:

```
--data_dir=<path_to_data_directory>
```

To generate example translation output, issue:

```
sample_nmt --data_dir=<path> --data_writer=text
```

The example translations can then be found in the **translation_output.txt** file.

To get the BLEU score for the first 100 sentences, issue:

```
sample_nmt --data_dir=<path> --max_inference_samples=100
```

The following options are available when running the sample:
**--help**
  Output help message and exit.
**--data_writer=bleu/text/benchmark**
  Type of the output the app generates (default = **bleu**).
**--output_file=<path_to_file>**
  Path to the output file when **data_writer=text**.
**--batch=<N>**
  Batch size (default = **128**).
**--beam=<N>**
  Beam width (default = **5**).
**--max_input_sequence_length=<N>**
  Maximum length for input sequences (default = **150**).
**--max_output_sequence_length=<N>**
  Maximum length for output sequences (default = **-1**), negative value indicates no limit.
**--max_inference_samples=<N>**
  Maximum sample count to run inference for, negative values indicates no limit is set (default = **-1**).
**--verbose**
  Output information level messages by TensorRT.

**--max_workspace_size=<N>**
　　Maximum workspace size (default = **268435456**).
**--data_dir=<path_to_data_directory>**
　　Path to the directory where data and weights are located (default = **../../../../
　　data/samples/nmt/deen**).
**--profile**
　　Profile TensorRT execution layer by layer. Use benchmark **data_writer** when
　　profiling on, disregard benchmark results.
**--aggregate_profile**
　　Merge profiles from multiple TensorRT engines.
**--fp16**
　　Switch on FP16 math.

## 9.1.9.4. Training The Model

Training the NMT model can be done in TensorFlow. This sample was trained following
the general outline of the TensorFlow Neural Machine Translation Tutorial. The first step
is to obtain training data, which is handled by the steps in Preparing The Data.

The next step is to fetch the TensorFlow NMT framework, for example:

```
git clone https://github.com/tensorflow/nmt.git
```

The model description is located in the **nmt/nmt/standard_hparams/wmt16.json**
file. This file encodes values for all the hyperparameters available for NMT models.
Not all variations are supported by the current NMT sample code so this file should be
edited with appropriate values. For example, only unidirectional LSTMs and the Luong
attention model are supported. The exact parameters used for the pre-trained weights
are available in the sample **README.txt** file.

After the model description is ready and the training data is available in the **<path>/
wmt16_de_en** directory, the command to train the model is:

```
python -m nmt.nmt \
--src=de --tgt=en \
--hparams_path=<path_to_json_config>/wmt16.json \
--out_dir=/tmp/deen_nmt \
--vocab_prefix=/tmp/wmt16_de_en/vocab.bpe.32000 \
--train_prefix=/tmp/wmt16_de_en/train.tok.clean.bpe.32000 \
--dev_prefix=/tmp/wmt16_de_en/newstest2013.tok.bpe.32000 \
--test_prefix=/tmp/wmt16_de_en/newstest2015.tok.bpe.32000
```

## 9.1.9.5. Importing Weights From A Checkpoint

Training the model generates various output files describing the state of the model. In
order to use the model with TensorRT, model weights must be loaded into the TensorRT
network. The weight values themselves are included in the TensorFlow checkpoint
produced during training. In the sample directory, we provide a Python script that
extracts the weights from a TensorFlow checkpoint into a set of binary weight files that
can be directly loaded by the sample.

To use the script, run the command:

> The `chpt_to_bin.py` script is located in the `/usr/src/tensorrt/samples/sampleNMT` directory.

```
python ./chpt_to_bin.py \
    --src=de --tgt=en \
    --ckpt=/tmp/deen_nmt/translate.ckpt-340000 \
    --hparams_path=<path_to_json_config>/wmt16.json \
    --out_dir=/tmp/deen \
    --vocab_prefix=<path>/wmt16_de_en/vocab.bpe.32000 \
    --inference_input_file=\
        <path>/wmt16_de_en/newstest2015.tok.bpe.32000.de \
    --inference_output_file=/tmp/deen/output_infer \
    --inference_ref_file=\
        <path>/wmt16_de_en/newstest2015.tok.bpe.32000.en
```

This generates 7 binary weight files for all the pieces of the model. The binary format is just a raw dump of the floating point values in order, followed by a metadata. The script was tested against TensorFlow 1.6.

# 9.1.10. sampleFasterRCNN

**What Does This Sample Do?**

The sampleFasterRCNN sample demonstrates how to:

▸ Use the Faster R-CNN plugin which allows for end-to-end inferencing

▸ Implement the Faster R-CNN network in TensorRT

▸ Perform a quick performance test in TensorRT

▸ Implement a fused custom layer

▸ Construct the basis for further optimization, for example using INT8 calibration, user trained network, etc.

**Where Is This Sample Located?**

The sampleFasterRCNN sample is installed in the `/usr/src/tensorrt/samples/sampleFasterRNN` directory.

The Faster R-CNN Caffe model is too large to include in the product bundle. To run this sample, download the model using the instructions in the `README.txt` in the sample directory. The README is located in the `<TensorRT directory>/samples/sampleFasterRCNN` directory. Once the model is downloaded and extracted as per the instructions, the sample can be run by invoking `sample_fasterRCNN` binary.

**Notes About This Sample:**

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

## 9.1.10.1. Overview

The sampleFasterRCNN is a more complex sample. The Faster R-CNN network is based on the paper Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.

Faster R-CNN is a fusion of Fast R-CNN and RPN (Region Proposal Network). The latter is a fully convolutional network that simultaneously predicts object bounds and objectness scores at each position. It can be merged with Fast R-CNN into a single network because it is trained end-to-end along with the Fast R-CNN detection network and thus shares with it the full-image convolutional features, enabling nearly cost-free region proposals. These region proposals will then be used by Fast R-CNN for detection.

The sampleFasterRCNN sample uses a plugin from the TensorRT plugin library to include a fused implementation of Faster R-CNN's Region Proposal Network (RPN) and ROIPooling layers. These particular layers are from the Faster R-CNN paper and are implemented together as a single plugin called `RPNROIPlugin`. This plugin is registered in the TensorRT Plugin Registry with the name `RPROI_TRT`.

Faster R-CNN is faster and more accurate than its predecessors (RCNN, Fast R-CNN) because it allows for an end-to-end inferencing and does not need standalone region proposal algorithms (like selective search in Fast R-CNN) or classification method (like SVM in RCNN).

## 9.1.10.2. Preprocessing The Input

The input to the Faster R-CNN network is 3 channel 375x500 images.

Since TensorRT does not depend on any computer vision libraries, the images are represented in binary `R`, `G`, and `B` values for each pixels. The format is Portable PixMap (PPM), which is a netpbm color image format. In this format, the `R`, `G`, and `B` values for each pixel are represented by a byte of integer (0-255) and they are stored together, pixel by pixel.

However, the authors of SSD have trained the network such that the first Convolution layer sees the image data in `B`, `G`, and `R` order. Therefore, we reverse the channel order when the PPM images are being put into the network buffer.

```
float* data = new float[N*INPUT_C*INPUT_H*INPUT_W];
// pixel mean used by the Faster R-CNN's author
float pixelMean[3]{ 102.9801f, 115.9465f, 122.7717f }; // also in BGR order
for (int i = 0, volImg = INPUT_C*INPUT_H*INPUT_W; i < N; ++i)
{
 for (int c = 0; c < INPUT_C; ++c)
 {
  // the color image to input should be in BGR order
  for (unsigned j = 0, volChl = INPUT_H*INPUT_W; j < volChl; ++j)
data[i*volImg + c*volChl + j] =  float(ppms[i].buffer[j*INPUT_C + 2 - c]) –
 pixelMean[c];
 }
}
```

There is a simple PPM reading function called `readPPMFile`.

> The `readPPMFile` function will not work correctly if the header of the PPM image contains any annotations starting with `#`.

Furthermore, within the sample, there is another function called **writePPMFileWithBBox**, that plots a given bounding box in the image with one-pixel width red lines.

In order to obtain PPM images, you can easily use the command-line tools such as ImageMagick to perform the resizing and conversion from JPEG images.

If you choose to use off-the-shelf image processing libraries to preprocess the inputs, ensure that the TensorRT inference engine sees the input data in the form that it is supposed to.

## 9.1.10.3. Defining The Network

The network is defined in a prototxt file which is shipped with the sample and located in the **data/faster-rcnn** directory. The prototxt file is very similar to the one used by the inventors of Faster R-CNN except that the RPN and the ROI pooling layer is fused and replaced by a custom layer named **RPROIFused**.

Similar to samplePlugin, in order to add Custom layers via NvCaffeParser, you need to create a factory by implementing the **nvcaffeParser::IPluginFactory** interface and then pass an instance to ICaffeParser::parse(). But unlike samplePlugin, in which the **FCPlugin** is defined in the sample, the **RPROIFused** plugin layer instance can be created by the **create** function implemented in the TensorRT plugin library **createRPNROIPlugin**. This function returns an instance that implements an optimized **RPROIFused** Custom layer and performs the same logic designed by the authors.

## 9.1.10.4. Building The Engine

For details on how to build the TensorRT engine, see Building An Engine In C++.

> 💬 In the case of the Faster R-CNN sample, **maxWorkspaceSize** is set to **10 * (2^20)**, namely 10MB, because there is a need of roughly 6MB of scratch space for the plugin layer for batch size 5.

After the engine is built, the next steps are to serialize the engine, then run the inference with the deserialized engine. For more information, see Serializing A Model In C++.

## 9.1.10.5. Running The Engine

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

In **sampleFasterRCNN.cpp**, there are two inputs to the inference function:
**data**
    **data** is the image input
**imInfo**
    **imInfo** is the image information array which stores the number of rows, columns, and the scale for each image in a batch.
and four outputs:
**bbox_pred**
    **bbox_pred** is the predicted offsets to the heights, widths and center coordinates.

**cls_prob**

    **cls_prob** is the probability associated with each object class of every bounding box.

**rois**

    **rois** is the height, width, and the center coordinates for each bounding box.

**count**

    **count** is deprecated and can be ignored.

> The **count** output was used to specify the number of resulting NMS bounding boxes if the output is not aligned to **nmsMaxOut**. Although it is deprecated, always allocate the engine buffer of size **batchSize * sizeof(int)** for it until it is completely removed from the future version of TensorRT.

## 9.1.10.6. Verifying The Output

The outputs of the Faster R-CNN network need to be post-processed in order to obtain human interpretable results.

First, because the bounding boxes are now represented by the offsets to the center, height, and width, they need to be unscaled back to the raw image space by dividing the scale defined in the **imInfo** (image info).

Ensure you apply the inverse transformation on the bounding boxes and clip the resulting coordinates so that they do not go beyond the image boundaries.

Lastly, overlapped predictions have to be removed by the non-maximum suppression algorithm. The post-processing codes are defined within the CPU because they are neither compute intensive nor memory intensive.

After all of the above work, the bounding boxes are available in terms of the class number, the confidence score (probability), and four coordinates. They are drawn in the output PPM images using the **writePPMFileWithBBox** function.

## 9.1.11. sampleUffSSD

**What Does This Sample Do?**

The sampleUffSSD sample demonstrates how to:

▸ Preprocess the TensorFlow SSD network

▸ Perform inference on the SSD network in TensorRT

▸ Use TensorRT plugins to speed up inference

**Where Is This Sample Located?**

The sampleUffSSD sample is installed in the **tensorrt/samples/sampleUffSSD** directory.

**Notes About This Sample:**

The frozen graph for the SSD network is too large to include in the TensorRT package. Ensure you read the instructions in the README located at `tensorrt/samples/sampleUffSSD` for details on how to generate the network to run inference.

## 9.1.11.1. API Overview

The sampleUffSSD is based on the following paper, SSD: Single Shot MultiBox Detector. The SSD network, built on the VGG-16 network, performs the task of object detection and localization in a single forward pass of the network. This approach discretizes the output space of bounding boxes into a set of default boxes over different aspect ratios and scales per feature map location. At prediction time, the network generates scores for the presence of each object category in each default box and produces adjustments to the box to better match the object shape. Additionally, the network combines predictions from multiple features with different resolutions to naturally handle objects of various sizes.

The sampleUffSSD is based on the TensorFlow implementation of SSD. For more information, see ssd_inception_v2_coco.

Unlike the paper, the TensorFlow SSD network was trained on the InceptionV2 architecture using the MSCOCO dataset which has 91 classes (including the background class). The configuration details of the network can be found at GitHub: TensorFlow models.

The main components of this network are the Preprocessor, FeatureExtractor, BoxPredictor, GridAnchorGenerator and Postprocessor.

**Preprocessor**
> The preprocessor step of the graph is responsible for resizing the image. The image is resized to a 300x300x3 size tensor. The preprocessor step also performs normalization of the image so all pixel values lie between the range [-1, 1].

**FeatureExtractor**
> The FeatureExtractor portion of the graph runs the InceptionV2 network on the preprocessed image. The feature maps generated are used by the anchor generation step to generate default bounding boxes for each feature map.

> In this network, the size of feature maps that are used for anchor generation are [(19x19), (10x10), (5x5), (3x3), (2x2), (1x1)].

**BoxPredictor**
> The BoxPredictor step takes in a high level feature map as input and produces a list of box encodings (x-y coordinates) and a list of class scores for each of these encodings per feature map. This information is passed to the postprocessor.

**GridAnchorGenerator**
> The goal of this step is to generate a set of default bounding boxes (given the scale and aspect ratios mentioned in the config) for each feature map cell. This is implemented as a plugin layer in TensorRT called the `gridAnchorGenerator` plugin. The registered plugin name is `GridAnchor_TRT`.

**Postprocessor**
> The postprocessor step performs the final steps to generate the network output. The bounding box data and confidence scores for all feature maps are fed to

the step along with the pre-computed default bounding boxes (generated in the `GridAnchorGenerator` namespace). It then performs NMS (non-maximum suppression) which prunes away most of the bounding boxes based on a confidence threshold and IoU (Intersection over Union) overlap, thus storing only the top `N` boxes per class. This is implemented as a plugin layer in TensorRT called the `NMS` plugin. The registered plugin name is `NMS_TRT`.

> This sample also implements another plugin called `FlattenConcat` which is used to flatten each input and then concatenate the results. This is applied to the location and confidence data before it is fed to the post processor step since the `NMS` plugin requires the data to be in this format.

For details on how a plugin is implemented, see the implementation of `FlattenConcat` Plugin and `FlattenConcatPluginCreator` in the `sampleUffSSD.cpp` file in the `tensorrt/samples/sampleUffSSD` directory.

## 9.1.11.2. Processing The Input Graph

The TensorFlow SSD graph has some operations that are currently not supported in TensorRT. Using a preprocessor on the graph, we can combine multiple operations in the graph into a single custom operation which can be implemented as a plugin layer in TensorRT. Currently, the preprocessor provides the ability to stitch all nodes within a namespace into one custom node.

To use the preprocessor, the `convert-to-uff` utility should be called with a `-p` flag and a config file. The config script should also include attributes for all custom plugins which will be embedded in the generated `.uff` file. Current sample scripts for SSD is located in `/usr/src/tensorrt/samples/sampleUffSSD/config.py`.

Using the preprocessor on the graph, we were able to remove the preprocessor namespace from the graph, stitch the `GridAnchorGenerator` namespace to create the `GridAnchorGenerator` plugin, stitch the postprocessor namespace to the `NMS` plugin and mark the concat operations in the BoxPredictor as `FlattenConcat` plugins.

The TensorFlow graph has some operations like `Assert` and `Identity` which can be removed for the inferencing. Operations like `Assert` are removed and leftover nodes (with no outputs once assert is deleted) are then recursively removed.

Identity operations are deleted and the input is forwarded to all the connected outputs. TensorRT does not currently support `Relu6(x)` operation, so the preprocessor also replaces this operation with a `Relu(x) - Relu(x-6)`.

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

## 9.1.11.3. Preparing The Data

The generated network has an input node called `Input` and the output node is given the name `MarkOutput_0` by the UFF converter. These nodes are registered by the UFF Parser in the sample.

```
parser->registerInput("Input", DimsCHW(3, 300, 300), UffInputOrder::kNCHW);
parser->registerOutput("MarkOutput_0");
```

The input to the SSD network in this sample is 3 channel 300x300 images. In the sample, we normalize the image so the pixel values lie in the range [-1,1]. This is equivalent to the preprocessing stage of the network.

Since TensorRT does not depend on any computer vision libraries, the images are represented in binary **R**, **G**, and **B** values for each pixels. The format is Portable PixMap (PPM), which is a netpbm color image format. In this format, the **R**, **G**, and **B** values for each pixel are represented by a byte of integer (0-255) and they are stored together, pixel by pixel. There is a simple PPM reading function called **readPPMFile**.

## 9.1.11.4. Defining The Network And Plugins

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

The **config.py** defined for the **convert-to-uff** command should have the custom layers mapped to the plugin names in TensorRT by modifying the **op** field. The names of the plugin parameters should also exactly match those expected by the TensorRT plugins. For example, for the GridAnchor Plugin, the **config.py** should have the following:

```
PriorBox = gs.create_plugin_node(name="GridAnchor", op="GridAnchor_TRT",
    numLayers=6,
    minSize=0.2,
    maxSize=0.95,
    aspectRatios=[1.0, 2.0, 0.5, 3.0, 0.33],
    variance=[0.1,0.1,0.2,0.2],
    featureMapShapes=[19, 10, 5, 3, 2, 1])
```

Here, **GridAnchor_TRT** matches the registered plugin name and the parameters have the same name and type as expected by the plugin.

If the **config.py** is defined as above, the NvUffParser will be able to parser the network and call the appropriate plugins with the correct parameters.

Alternately, the older flow of using the **IPluginFactory** can also be used. In this case, the **pluginFactory** object created needs to be passed to an instance of **IUffParser::parse()** which will invoke the **createPlugin()** function for each Custom layer which has to be implemented by the user. Details about some of the plugin layers implemented for SSD in TensorRT are given below.

**GridAnchorGeneration Plugin**
   This plugin layer implements the grid anchor generation step in the TensorFlow SSD network. For each feature map we calculate the bounding boxes for each grid cell. In this network, there are 6 feature maps and the number of boxes per grid cell are as follows:

   ▸ [19x19] feature map: 3 boxes (19x19x3x4(co-ordinates/box))
   ▸ [10x10] feature map: 6 boxes (10x10x6x4)
   ▸ [5x5] feature map: 6 boxes (5x5x6x4)
   ▸ [3x3] feature map: 6 boxes (3x3x6x4)
   ▸ [2x2] feature map: 6 boxes (2x2x6x4)
   ▸ [1x1] feature map: 6 boxes (1x1x6x4)

**NMS Plugin**

The **NMS** plugin generates the detection output based on location and confidence predictions generated by the BoxPredictor. This layer has three input tensors corresponding to location data (**locData**), confidence data (**confData**) and priorbox data (**priorData**).

The inputs to detection output plugin have to be flattened and concatenated across all the feature maps. We use the **FlattenConcat** plugin implemented in the sample to achieve this. The location data generated from the box predictor has the following dimensions:

```
19x19x12 -> Reshape -> 1083x4 -> Flatten -> 4332x1
10x10x24 -> Reshape -> 600x4 -> Flatten -> 2400x1
```

and so on for the remaining feature maps.

After concatenating, the input dimensions for **locData** input are of the order of 7668x1.

The confidence data generated from the box predictor has the following dimensions:

```
19x19x273 -> Reshape -> 1083x91 -> Flatten -> 98553x1
10x10x546 -> Reshape -> 600x91 -> Flatten -> 54600x1
```

and so on for the remaining feature maps.

After concatenating, the input dimensions for **confData** input are of the order of 174447x1.

The prior data generated from the grid anchor generator plugin has the following dimensions, for example 19x19 feature map has 2x4332x1 (there are two channels here because one channel is used to store variance of each coordinate that is used in the NMS step). After concatenating, the input dimensions for **priorData** input are of the order of 2x7668x1.

```
struct DetectionOutputParameters
{
        bool shareLocation, varianceEncodedInTarget;
        int backgroundLabelId, numClasses, topK, keepTopK;
        float confidenceThreshold, nmsThreshold;
        CodeTypeSSD codeType;
        int inputOrder[3];
        bool confSigmoid;
        bool isNormalized;
};
```

**shareLocation** and **varianceEncodedInTarget** are used for the Caffe implementation, so for the TensorFlow network they should be set to **true** and **false** respectively. The **confSigmoid** and **isNormalized** parameters are necessary for the TensorFlow implementation. If **confSigmoid** is set to **true**, it calculates the sigmoid values of all the confidence scores. The **isNormalized** flag specifies if the data is normalized and is set to **true** for the TensorFlow graph.

## 9.1.11.5. Verifying The Output

After the builder is created (see Building An Engine In C++) and the engine is serialized (see Serializing A Model In C++), we can perform inference. Steps for deserialization and running inference are outlined in Performing Inference In C++.

The outputs of the SSD network are human interpretable. The post-processing work, such as the final NMS, is done in the **NMS** plugin. The results are organized as tuples of 7. In each tuple, the 7 elements are respectively image ID, object label, confidence score, (**x,y**) coordinates of the lower left corner of the bounding box, and (**x,y**) coordinates of the upper right corner of the bounding box. This information can be drawn in the output PPM image using the **writePPMFileWithBBox** function. The **visualizeThreshold** parameter can be used to control the visualization of objects in the image. It is currently set to 0.5 so the output will display all objects with confidence score of 50% and above.

# 9.1.12. sampleMovieLens

**What Does This Sample Do?**

The sampleMovieLens sample demonstrates a simple movie recommender system using Neural Collaborative Filter (NCF). The network is trained in TensorFlow on the MovieLens dataset containing 6040 users and 3706 movies. For more information about the recommender system network, see Neural Collaborative Filtering.

**Where Is This Sample Located?**

The sampleMovieLens sample in installed in the **usr/src/tensorrt/samples/ sampleMovieLens** directory.

**Notes About This Sample:**

Each query to the network consists of a **userID** and list of **MovieIDs**. The network predicts the highest-rated movie for each user. As trained parameters, the network has embeddings for users and movies, and weights for a sequence of Multi-Layer Perceptrons (MLPs).

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

## 9.1.12.1. Importing Network To TensorRT

The network is converted from TensorFlow using the UFF converter (see Converting A Frozen Graph To UFF), and imported using the UFF parser. Constant layers are used to represent the trained parameters within the network, and the MLPs are implemented using FullyConnected layers. A TopK operation is added manually after parsing to find the highest rated movie for the given user.

## 9.1.12.2. Running With MPS

MPS (Multi-Process Service) allows multiple CUDA processes to share single GPU context. With MPS, multiple overlapping kernel execution and **memcpy** operations from different processes can be scheduled concurrently to achieve maximum utilization. This

can be especially effective in increasing parallelism for small networks with low resource utilization such as those primarily consisting of a series of small MLPs. For more information about MPS, see Multi-Process Service documentation or in the `README.txt` file for the sample.

MPS requires a server process. To start the process:

```
export CUDA_VISIBLE_DEVICES=<GPU_ID>
nvidia-smi -i <GPU_ID> -c EXCLUSIVE_PROCESS
nvidia-cuda-mps-control -d
```

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

### 9.1.12.3. Verifying The Output

The output of the MLP based NCF network is in human readable format. The final output is `movieID` with probability rating for give `userID`.

## 9.1.13. sampleSSD

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

▸ Preprocess the input to the SSD network

▸ Perform inference on the SSD network in TensorRT

▸ Use TensorRT plugins to speed up inference

▸ Perform INT8 calibration on an SSD network

**Where Is This Sample Located?**
The sampleSSD sample is installed in the `/usr/src/tensorrt/samples/sampleSSD` directory.

**Notes About This Sample:**
The SSD Caffe model is too large to include in the product bundle. To run this sample, download the model using the instructions in the README.md in the sample `<TensorRT directory>/samples/sampleSSD` directory. The original Caffe model (prototxt) has been modified to include the SSD's customized Plugin layers.

### 9.1.13.1. Overview

The SSD network is based on the following paper SSD: Single Shot MultiBox Detector. This network is based on the VGG-16 network. It can perform object detection and localization in a single forward pass.

Unlike Faster R-CNN, SSD completely eliminates proposal generation and subsequent pixel or feature resampling stages and encapsulates all computation in a single network. This makes SSD straightforward to integrate into systems that require a detection component.

## 9.1.13.2. Preprocessing The Input

The input to the SSD network in this sample is a RGB 300x300 image. The image format is Portable PixMap (PPM), which is a netpbm color image format. In this format, the **R**, **G**, and **B** values for each pixel are represented by a byte of integer (0-255) and they are stored together, pixel by pixel.

The authors of SSD have trained the network such that the first Convolution layer sees the image data in **B**, **G**, and **R** order. Therefore, the channel order needs to be changed when the PPM image is being put into the network's input buffer.

```
float pixelMean[3]{ 104.0f, 117.0f, 123.0f }; // also in BGR order
float* data = new float[N * kINPUT_C * kINPUT_H * kINPUT_W];
 for (int i = 0, volImg = kINPUT_C * kINPUT_H * kINPUT_W; i < N; ++i)
 {
  for (int c = 0; c < kINPUT_C; ++c)
  {
   // the color image to input should be in BGR order
   for (unsigned j = 0, volChl = kINPUT_H * kINPUT_W; j < volChl; ++j){
    Data[i * volImg + c * volChl + j] = float(ppms[i].buffer[j * kINPUT_C + 2 -
c]) - pixelMean[c];
   }
  }
 }
```

The **readPPMFile** and **writePPMFileWithBBox** functions read a PPM image and produce output images with red colored bounding boxes respectively.

> 💬 The **readPPMFile** function will not work correctly if the header of the PPM image contains any annotations starting with **#**.

## 9.1.13.3. Defining The Network

The network is defined in a prototxt file which is shipped with the sample and located in the **data/ssd** directory. The original prototxt file provided by the authors is modified and included in the TensorRT in-built plugin layers in the prototxt file.

The built-in plugin layers used in sampleSSD are Normalize, PriorBox, and DetectionOutput. The corresponding registered plugins for these layers are **Normalize_TRT**, **PriorBox_TRT** and **NMS_TRT**.

To initialize and register these TensorRT plugins to the plugin registry, the **initLibNvInferPlugins** method is used. After registering the plugins and while parsing the prototxt file, the NvCaffeParser creates plugins for the layers based on the parameters that were provided in the prototxt file automatically. The details about each parameter is provided in the README.md and can be modified similar to the Caffe Layer parameter.

## 9.1.13.4. Building The Engine

The sampleSSD sample builds a network based on a Caffe model and network description. For details on importing a Caffe model, see Importing A Caffe Model Using The C++ Parser API. The SSD network has few non-natively supported layers which

are implemented as plugins in TensorRT. The Caffe parser can create plugins for these layers internally which avoids creating additional code for plugin factory like in the sampleFasterRCNN sample.

This sample can run in FP16 and INT8 modes based on the user input. For more details, see Optimizing INT8 Calibration Using C++ API and Enabling FP16 Inference Using C++. The sample selects the entropy calibrator as a default choice. The `CalibrationMode` parameter in the sample code needs to be set to `0` to switch to the Legacy calibrator.

For details on how to build the TensorRT engine, see Building An Engine In C++. After the engine is built, the next steps are to serialize the engine and run the inference with the deserialized engine. For more information about these steps, see Serializing A Model In C++.

### 9.1.13.5. Verifying The Output

After deserializing the engine, you can perform inference. To perform inference, see Performing Inference In C++.

In sampleSSD, there is a single input:
`data`
    Namely the image input.
and 2 outputs:
`detectionOut`
    The detection array, containing the image ID, label, confidence, and 4 coordinates.
`keepCount`
    The number of valid detections.

The outputs of the SSD network are directly human interpretable. The results are organized as tuples of 7. In each tuple, the 7 elements are:

- image ID
- object label
- confidence score
- (x,y) coordinates of the lower left corner of the bounding box
- (x,y) coordinates of the upper right corner of the bounding box

This information can be drawn in the output PPM image using the `writePPMFileWithBBox` function. The `kVISUAL_THRESHOLD` parameter can be used to control the visualization of objects in the image. It is currently set to 0.6, therefore, the output will display all objects with confidence score of 60% and above.

# 9.2. Python Samples

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

- introductory_parser_samples
- end_to_end_tensorflow_mnist

- ▸ network_api_pytorch_mnist
- ▸ fc_plugin_caffe_mnist
- ▸ uff_custom_plugin

**Running Python Samples**

Every Python sample includes a `README.md` and `requirements.txt` file. To run one of the Python samples, the process typically involves two steps:

1. Install the sample requirements:

   ```
   python<x> -m pip install -r requirements.txt
   ```

   where `python<x>` is either `python2` or `python3`.

2. Run the sample code with the `data` directory provided if the TensorRT sample data is not in the default location. For example:

   ```
   python<x> sample.py [-d DATA_DIR]
   ```

For more information on running samples, see the `README.md` file included with the sample.

# 9.2.1. introductory_parser_samples

**What Does This Sample Do?**
This sample demonstrates how to use TensorRT and its included suite of parsers (the UFF, Caffe and ONNX parsers), to perform inference with ResNet-50 models trained with various different frameworks.

This sample includes the following:
**caffe_resnet50**
   This sample demonstrates how to build an engine from a trained Caffe model using the Caffe parser and then run inference.

**onnx_resnet50**
   This sample demonstrates how to build an engine from an ONNX model file using the open-source ONNX parser and then run inference.

**uff_resnet50**
   This sample demonstrates how to build an engine from a UFF model file (converted from a TensorFlow protobuf) and then run inference.

**Where Is This Sample Located?**
The introductory_parser_samples sample is installed in the `/usr/src/tensorrt/samples/python/introductory_parser_samples` directory.

# 9.2.2. end_to_end_tensorflow_mnist

**What Does This Sample Do?**

This sample demonstrates how to first train a model using TensorFlow and Keras, freeze the model and write it to a protobuf file, convert it to UFF, and finally run inference using TensorRT.

**Where Is This Sample Located?**

The end_to_end_tensorflow_mnist sample is installed in the **/usr/src/tensorrt/ samples/python/end_to_end_tensorflow_mnist** directory.

# 9.2.3. network_api_pytorch_mnist

**What Does This Sample Do?**

This sample demonstrates how to train a model in PyTorch, recreate the network in TensorRT and import weights from the trained model, and finally run inference with a TensorRT engine.

**Where Is This Sample Located?**

The network_api_pytorch_mnist sample is installed in the **/usr/src/tensorrt/ samples/python/network_api_pytorch_mnist** directory.

**Notes About This Sample:**

The **sample.py** script imports the functions from the **mnist.py** script for training the PyTorch model, as well as retrieving test cases from the PyTorch Data Loader.

# 9.2.4. fc_plugin_caffe_mnist

**What Does This Sample Do?**

This sample demonstrates how to use plugins written in C++ with the TensorRT Python bindings and CaffeParser. More specifically, this sample implements a FullyConnected layer using cuBLAS and cuDNN, wraps the implementation in a TensorRT plugin (with a corresponding plugin factory) and then generates Python bindings for it using **pybind11**. These bindings are then used to register the plugin factory with the CaffeParser.

**Where Is This Sample Located?**

The fc_plugin_caffe_mnist sample is installed in the **/usr/src/tensorrt/samples/ python/fc_plugin_caffe_mnist** directory.

# 9.2.5. uff_custom_plugin

## What Does This Sample Do?

This sample demonstrates how to use plugins written in C++ with the TensorRTPython bindings and UFF Parser. More specifically, this sample implementsa clip layer (as a CUDA kernel), wraps the implementation in a TensorRT plugin(with a corresponding plugin creator) and then generates a shared library modulecontaining its code. The user then dynamically links this library in Python,which causes plugin to be registered in TensorRT's PluginRegistry andmakes it available for UFF parser.

## Where Is This Sample Located?

The uff_custom_plugin sample is installed in the **/usr/src/tensorrt/samples/ python/uff_custom_plugin** directory.

# Chapter 10.
# TROUBLESHOOTING

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

## 10.1. FAQs

**Q: How do you create an engine that is optimized for several different batch sizes?**
A: While TensorRT allows an engine optimized for a given batch size to run at any smaller size, the performance for those smaller sizes may not be as well-optimized. To optimize for multiple different batch sizes, run the builder and serialize an engine for each batch size.

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

**Q: How do you choose the optimal workspace size?**
A: Some TensorRT algorithms require additional workspace on the GPU. The method `IBuilder::setMaxWorkspaceSize()` controls the maximum amount of workspace that may be allocated, and will prevent algorithms that require more workspace from being considered by the builder. At runtime, the space is allocated automatically when creating an `IExecutionContext`. The amount allocated will be no more than is required, even if the amount set in `IBuilder::setMaxWorkspaceSize()` is much higher. Applications should therefore allow the TensorRT builder as much workspace as they can afford; at runtime TensorRT will allocate no more than this, and typically less.

**Q: How do you use TensorRT on multiple GPUs?**

A: Each `ICudaEngine` object is bound to a specific GPU when it is instantiated, either by the builder or on deserialization. To select the GPU, use `cudaSetDevice()` before calling the builder or deserializing the engine. Each `IExecutionContext` is bound to the same GPU as the engine from which it was created. When calling `execute()` or `enqueue()`, ensure that the thread is associated with the correct device by calling `cudaSetDevice()` if necessary.

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

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

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

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

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

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

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

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

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

# 10.2. Support

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

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

# Appendix A.
# APPENDIX

## A.1. TensorRT Layers

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

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

### A.1.1. Activation Layer

The Activation layer implements element-wise activation functions.

**Layer Description**

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

The Activation layer supports the following operations:

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

**Layer Creation And Configuration Methods**

Usage with the C++ API:

▶ `INetworkDefinition::addActivation(A, f)`

▶ `IActivationLayer::setActivationType(f)`

Usage with the Python API:

- ▶ `NetworkDefinition::add_activation(A, f)`
- ▶ `ActivationLayer::set_activation(f)`

**Conditions And Limitations**

None

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

# A.1.2. Concatenation Layer

The Concatenation layer 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 $\mathbf{a}$ channel axis $c$.

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

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

**Layer Creation And Configuration Methods**

Usage with the C++ API:

- ▶ `INetworkDefinition::addConcatenationLayer(A_i, m)`
- ▶ `IConcatenationLayer::setAxis(c)`

Usage with the Python API:

- ▶ `NetworkDefinition::add_concatenation(A_i, m)`
- ▶ `ConcatenationLayer::set_axis(c)`

**Conditions And Limitations**

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

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

## A.1.3. Constant Layer

The Constant layer 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 $d$.

**Layer Creation And Configuration Methods**

Usage with the C++ API:

▸ `INetworkDefinition::addConstant(d,w)`
▸ `IConstantLayer::setDimensions(d)`
▸ `IConstLayer::setWeights(w)`

Usage with the Python API:

▸ `NetworkDefinition::add_constant(d, w)`
▸ `ConstantLayer::set_dimensions(d)`
▸ `ConstantLayer::set_weights(w)`

**Conditions And Limitations**

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

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

## A.1.4. Convolution Layer

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

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

**Layer Description**

Compute a cross-correlation with 2D filters on a 4D tensor $A$, of dimensions $a$, to produce a 4D tensor $B$, of dimensions $b$. The dimensions of $B$ depend on the dimensions of $A$, the number of output maps $m$, kernel size $r$, symmetric padding $p$, stride $s$, dilation

**d**, and dilated kernel size $t = r+d(r-1)$, such that height and width are adjusted accordingly as follows:

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

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

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

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

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

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

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

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

## Layer Creation And Configuration Methods

Usage with the C++ API:

- `INetworkDefinition::addConvolution(A, m, r, w, x)`
- `IConvolutionLayer::setKernelSize(r)`
- `IConvolutionLayer::setNbOutputMaps(m)`
- `IConvolutionLayer::setStride(s) - default s = [1 1]`
- `IConvolutionLayer::setPadding(p) - default p = [0 0]`
- `IConvolutionLayer::setNbGroups(g) - default g = 1`
- `IConvolutionLayer::setKernelWeights(w)`
- `IConvolutionLayer::setBiasWeights(x)`
- `IConvolutionLayer::setDilation(d) - default d = [0 0]`

Usage with the Python API:

- `NetworkDefinition::add_convolution(A, m, r, w, x)`
- `ConvolutionLayer::set_kernel_size(r)`
- `ConvolutionLayer::set_nb_output_maps(m)`
- `ConvolutionLayer::set_stride(s) - default s = [1 1]`

- ▸ `ConvolutionLayer::set_padding(p) - default p = [0 0]`
- ▸ `ConvolutionLayer::set_nb_groups(g) - default g = 1`
- ▸ `ConvolutionLayer::set_kernel_weights(w)`
- ▸ `ConvolutionLayer::set_bias_weights(x)`
- ▸ `ConvolutionLayer::set_dilation(d) - default d = [0 0]`

**Conditions And Limitations**

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

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

# A.1.5. Deconvolution Layer

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

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

**Layer Description**

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

- ▸ `b = [a₀ m b₂ b₃]` — $b = [a_0\ m\ b_2\ b_3]$
- ▸ `b₂ = ( a₂-1)*s₀ + t₀ - 2p₀` — $b_2 = (a_2-1)*s_0 + t_0 - 2p_0$
- ▸ `b₃ = ( a₃-1)*s₁ + t₁ - 2p₀` — $b_3 = (a_3-1)*s_1 + t_1 - 2p_0$

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

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

Let tensor $K$ with dimensions $k = [m\ b_1/g\ t_0\ t_1]$ be defined as the zero-filled tensor, such that:

- ▸ $k_{i,j,hh,ll} = w_{i,j,h,l}$

- ▶ `hh = {0 if h = 0, h + d`$_0$`(h-1) otherwise}`, and `ll = {0 if l = 0, l + d`$_1$`(l-1) otherwise}`.

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

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

where $u$ ranges from `0` to `min(t`$_0$`-1, k)`, and $v$ ranges from `0` to `min(t`$_1$`-1, l)`.

**Layer Creation And Configuration Methods**

Usage with the C++ API:

- ▶ `INetworkDefinition::addDeconvolution(A, `*`m`*`, `*`r`*`, `*`w`*`, `*`x`*`)`
- ▶ `IDeconvolutionLayer::setKernelSize(`*`r`*`)`
- ▶ `IDeconvolutionLayer::setNbOutputMaps(`*`m`*`)`
- ▶ `IDeconvolutionLayer::setStride(`*`s`*`) - default s = [1 1]`
- ▶ `IDeconvolutionLayer::setPadding(`*`p`*`) - default p = [0 0]`
- ▶ `IDeconvolutionLayer::setNbGroups(`*`g`*`) - default g = 1`
- ▶ `IDeconvolutionLayer::setKernelWeights(`*`w`*`)`
- ▶ `IDeconvolutionLayer::setBiasWeights(`*`x`*`)`

Usage with the Python API:

- ▶ `NetworkDefinition::add_deconvolution(A, `*`m`*`, `*`r`*`, `*`w`*`, `*`x`*`)`
- ▶ `DeconvolutionLayer::set_kernel_size(`*`r`*`)`
- ▶ `DeconvolutionLayer::set_nb_output_maps(`*`m`*`)`
- ▶ `DeconvolutionLayer::set_stride(`*`s`*`) - default s = [1 1]`
- ▶ `DeconvolutionLayer::set_padding(`*`p`*`) - default p = [0 0]`
- ▶ `DeconvolutionLayer::set_nb_groups(`*`g`*`) - default g = 1`
- ▶ `DeconvolutionLayer::set_kernel_weights(`*`w`*`)`
- ▶ `DeconvolutionLayer::set_bias_weights(`*`x`*`)`

**Conditions And Limitations**

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

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

# A.1.6. ElementWise Layer

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

**Layer Description**

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

The ElementWise layer supports the following operations:

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

**Layer Creation And Configuration Methods**

Usage with the C++ API:

▸ `INetworkDefinition::addElementWise(A, B, operation)`

▸ `IElementWiseLayer::setOperation(operation)`

Usage with the Python API:

▸ `NetworkDefinition::add_element_wise(A, B, operation)`

▸ `ElementWiseLayer::set_operation(operation)`

**Conditions And Limitations**

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

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

## A.1.7. FullyConnected Layer

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

**Layer Description**

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

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

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

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

## Layer Creation And Configuration Methods

Usage with the C++ API:

▸ `INetworkDefinition::addFullyConnected(A, k, W, x)`

▸ `IFullyConnectedLayer::setNbOutputChannels(k)`

▸ `IFullyConnectedLayer::setKernelWeights(W)`

▸ `IFullyConnectedLayer::setBiasWeights(x)`

Usage with the Python API:

▸ `NetworkDefinition::add_fully_connected(A, k, W, x)`

▸ `FullyConnectedLayer::set_nb_output_channels(k)`

▸ `FullyConnectedLayer::set_kernel_weights(W)`

▸ `FullyConnectedLayer::set_bias_weights(x)`

## Conditions And Limitations

`A` must have three dimensions or more.

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

# A.1.8. Gather Layer

The Gather layer implements the `gather` operation on a given axis.

## Layer Description

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

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

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

### Layer Creation And Configuration Methods

Usage with the C++ API:

▶ `INetworkDefinition::addGather(A, B, axis)`
▶ `IGatherLayer::setGatherAxis(axis)`

Usage with the Python API:

▶ `NetworkDefinition:add_gather(A, B, axis)`
▶ `GatherLayer::set_gather_axis(axis)`

### Conditions And Limitations

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

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

## A.1.9. Identity Layer

The Identity layer 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.

### Layer Creation And Configuration Methods

Usage with the C++ API:

▶ `INetworkDefinition::addIdentity()`
▶ `ILayer::setPrecision(precision)`

Usage with the Python API:

▶ `NetworkDefinition:add_identity()`
▶ `Layer::precision(precision)`

**Conditions And Limitations**

None

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

# A.1.10. LRN Layer

The LRN layer implements cross-channel Local Response Normalization.

**Layer Description**

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

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

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

**Layer Creation And Configuration Methods**

Usage with the C++ API:

▶ **INetworkDefinition::addLRN(A, w, α, β, k)**
▶ **ILRNLayer::setWindowSize(w)**
▶ **ILRNLayer::setAlpha(α)**
▶ **ILRNLayer::setBeta(β)**
▶ **ILRNLayer::setK(k)**

Usage with the Python API:

▶ **NetworkDefinition::add_lrn(A, w, α, β, k)**
▶ **LRNLayer::set_window_size(w)**
▶ **LRNLayer::set_alpha(α)**
▶ **LRNLayer::set_beta(β)**
▶ **LRNLayer::set_k(k)**

**Conditions And Limitations**

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

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

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

# A.1.11. MatrixMultiply Layer

The MatrixMultiply layer implements matrix multiplication for a collection of matrices.

**Layer Description**

The matrix multiply layer 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 $A^I$ and $B^I$ be the input tensors after the optional transpose, then

`C`$_{i0,..,in-3,:,:}$`=A`$^I_{i0,..,in-3,:,:}$`*B`$^I_{i0,..,in-3,:,:}$`.`

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

**Layer Creation And Configuration Methods**

Associate parameters to names used in the description (tensors etc.)

Usage with the C++ API:

- ▸ `INetworkDefinition::addMatrixMultiply(A, transposeA, B, transposeB)`
- ▸ `IMatrixMultiplyLayer::setTranspose(0, transposeA) // set whether A has to be transposed`
- ▸ `IMatrixMultiplyLayer::setTranspose(1, transposeB) // set whether B has to be transposed`

Usage with the Python API:

- ▸ `NetworkDefinition::add_matrix_multiply(A, transposeA, B, transposeB)`

- **`IMatrixMultiply::set_transpose(0, transposeA) // set whether A has to be transposed`**
- **`IMatrixMultiply::set_transpose(1, transposeB) // set whether B has to be transposed`**

### Conditions And Limitations

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

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

## A.1.12. Padding Layer

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

### Layer Description

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

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

### Layer Creation And Configuration Methods

Usage with the C++ API:

- **`INetworkDefinition::addPadding(A, pre, post)`**
- **`IPaddingLayer::setPrePadding(pre)`**
- **`IPaddingLayer::setPostPadding(post)`**

Usage with the Python API:

- **`NetworkDefinition::add_padding(A, pre, post)`**
- **`PaddingLayer::set_pre_padding(pre)`**
- **`PaddingLayer::set_post_padding(post)`**

**Conditions And Limitations**

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

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

## A.1.13. Plugin Layer

Plugin layers are user-defined and provide the ability to extend the functionalities of TensorRT. See Extending TensorRT With Custom Layers for more details.

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

## A.1.14. Pooling Layer

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

**Layer Description**

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

- ▸ `b = [a`$_0$` a`$_1$`...a`$_{n-3}$` b`$_{n-2}$` b`$_{n-1}$`]`
- ▸ `b`$_{n-2}$` = (a`$_{n-2}$`+2p`$_0$`-r`$_0$`)/s`$_0$`+1`
- ▸ `b`$_{n-1}$` = (a`$_{n-1}$`+2p`$_1$`-r`$_1$`)/s`$_1$`+1`

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

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

**PoolingType::kMAX**
  Maximum over elements in window.

**PoolingType::kAVERAGE**
  Average over elements in the window.

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

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

**Layer Creation And Configuration Methods**

Usage with the C++ API:

▸ `INetworkDefinition::addPooling(A, t, r)`
▸ `IPoolingLayer::setPoolingType(t)`
▸ `IPoolingLayer::setWindowSize(r)`
▸ `IPoolingLayer::setStride(s)`
▸ `IPoolingLayer::setPadding(p)`
▸ `IPoolingLayer::setBlendFactor(blendFactor)`
▸ `IPoolingLayer::setAverageCountExcludesPadding(exclusive)`

Usage with the Python API:

▸ `NetworkDefinition::add_pooling(A, t, r)`
▸ `PoolingLayer::set_pooling_type(t)`
▸ `PoolingLayer::set_window_size(r)`
▸ `PoolingLayer::set_stride(s)`
▸ `PoolingLayer::set_padding(p)`
▸ `PoolingLayer::set_blend_factor(blendFactor)`
▸ `PoolingLayer::set_average_count_excludes_padding(exclusive)`

**Conditions And Limitations**

Input and output tensors should have 3 or more dimensions.

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

# A.1.15. RaggedSoftMax Layer

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

**Layer Description**

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

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

**Layer Creation And Configuration Methods**

Usage with the C++ API:

▸ `INetworkDefinition::addRaggedSoftMax(A, B)`

Usage with the Python API:

▸ `NetworkDefinition::add_ragged_soft_max(A, B)`

**Conditions And Limitations**

None

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

# A.1.16. Reduce Layer

The Reduce layer implements dimension reduction of tensors using reduce operators.

**Layer Description**

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

With the sequence of indexes `i`, `B`$_i$`=op(A`$_j$`)`, where the sequence of indexes `j` is such that `j`$_k$`={: if k#r, and i`$_k$ `otherwise}`.

**Layer Creation And Configuration Methods**

Usage with the C++ API:

▸ `INetworkDefinition::addReduce(A, op, `*reduceAxes*`, keepDimensions)`
▸ `IReduceLayer::setOperation(op)`
▸ `IReduceLayer::setReduceAxes(`*reduceAxes*`)`
▸ `IReduceLayer::setKeepDimensions(keepDimensions)`

Usage with the Python API:

▸ `NetworkDefinition::add_reduce(A, op, `*reduceAxes*`, keepDimensions)`
▸ `ReduceLayer::set_operation(op)`
▸ `ReduceLayer::set_reduce_axes(`*reduceAxes*`)`
▸ `ReduceLayer::set_keep_dimensions(keepDimensions)`

**Conditions And Limitations**

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

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

# A.1.17. RNNv2 Layer (IRNNv2Layer) Layer

The RNNv2 layer implements recurrent layers such as Recurrent Neural Network (RNN), Gated Recurrent Units (GRU), and Long Short-Term Memory (LSTM). Supported types are RNN, GRU, and LSTM. It performs a recurrent operation, where the operation is defined by one of several well-known recurrent neural network (RNN) "cells".

**Layer Description**

This layer accepts an input sequence `X`, initial hidden state `H`$_0$, and if the cell is a long short-term memory (LSTM) cell, initial cell state `C`$_0$, and produces an output `Y` which represents the output of the final RNN "sub-layer" computed across `T` timesteps (see below). Optionally, the layer can also produce an output `h`$_T$ representing the final hidden state, and, if the cell is an LSTM cell, an output `c`$_T$ representing the final cell state.

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

In the default (unidirectional) configuration, the RNNv2 layer applies `G` as shown in the following diagram:
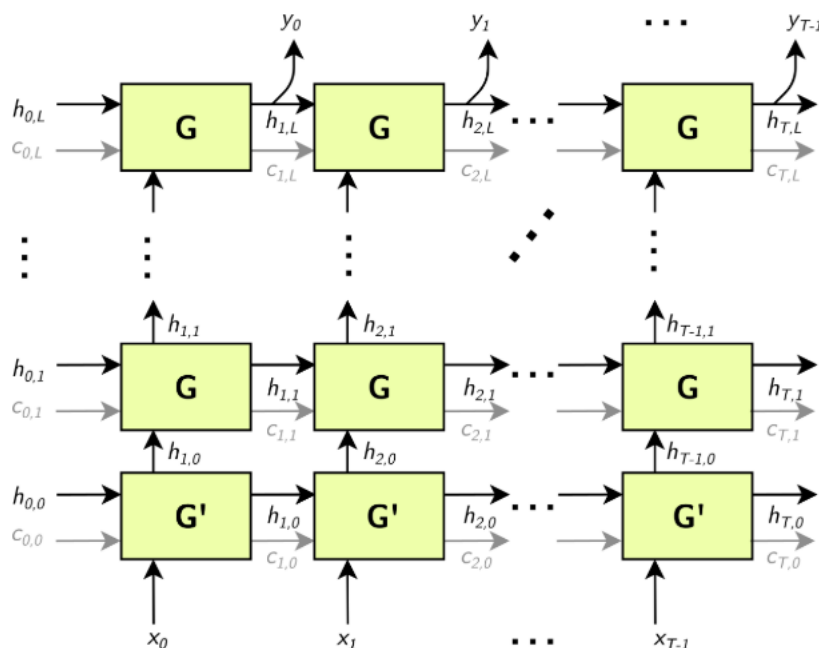
`G'` is a variant of `G,`.

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

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

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

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

**Bidirectional RNNs (BiRNNs)**: The RNN can be configured to be bidirectional. In that case, each sub-layer consists of a "forward" layer and "backward" layer. The forward layer iteratively applies `G` using $x_i$ from `0` to `T`, and the backward layer iteratively applies `G` using $x_i$ from `T` to `0`, as shown in the diagram below:
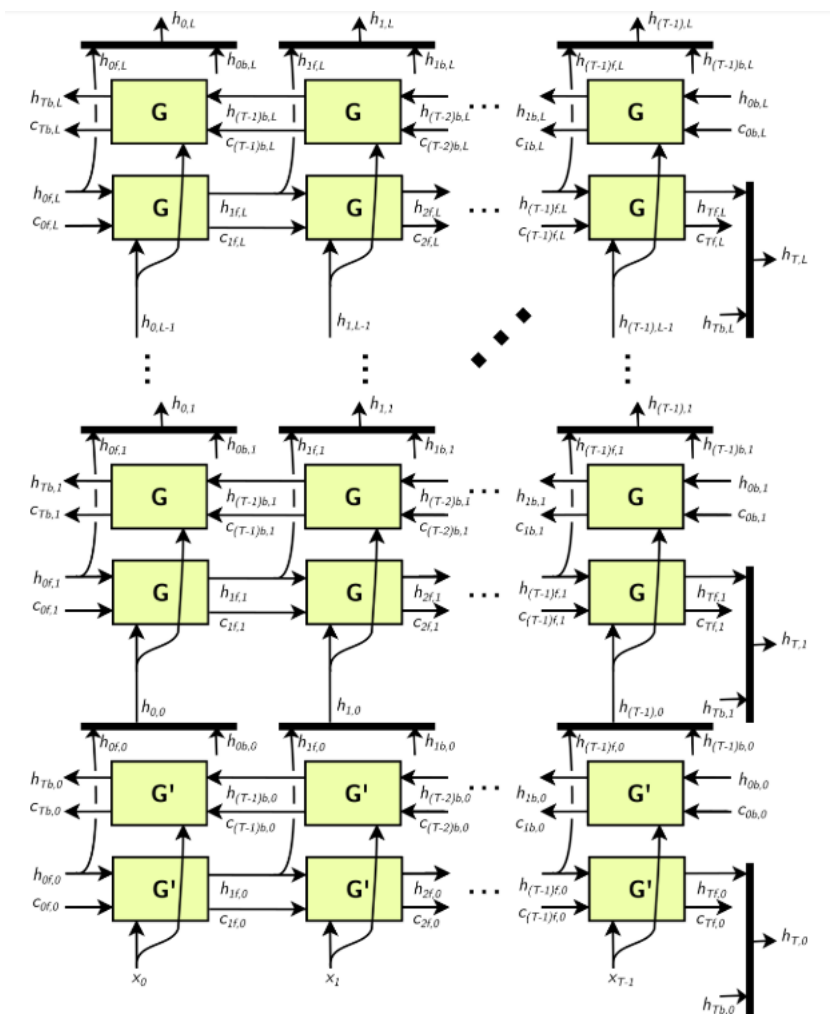
Black bars in the diagram above represent concatenation. The full hidden state $h_t$ is defined by the concatenation of the forward hidden state $h_{tf}$ and the backward hidden state $h_{tb}$:

▶ `h`$_{t,i}$ `= [ ` $h_{tf,i}$ `,`$h_{tb,i}$`]`

▶ `h`$_t$ `= [ h`$_{t,0,}$`, h`$_{t,1}$`, ...,h`$_{t,L}$`].`

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

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

▶ `ReLU: 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 := sigmoid(W`$_z$`x + R`$_z$`h + W`$_{bz}$` + R`$_{bz}$`)`

　　▶ `M := sigmoid(W`$_r$`x + R`$_r$`h + W`$_{br}$` + R`$_{br}$`)`

　　▶ `G(x, h, c) := tanh(W`$_h$`x + M(h + R`$_{bh}$`) + W`$_{bh}$`)` (`c` not used)

- **LSTM:**

  - `I := sigmoid(W`$_I$`x + R`$_I$`h + W`$_{bi}$` + R`$_{bi}$`)`
  - `F := sigmoid(W`$_f$`x + R`$_f$`h + W`$_{bf}$` + R`$_{bf}$`)`
  - `O := sigmoid(W`$_o$`x + R`$_o$`h + W`$_{bo}$` + R`$_{bo}$`)`
  - `C := tanh(W`$_C$`x + R`$_C$`h + W`$_{bc}$` + R`$_{bc}$`)`
  - `C' := F × C`
  - `H := O x tanh(C')`
  - `G(x, h, c) := { H, C' }`

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

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

The dimensionality of the `R` matrices is always *HxH*. The biases `W`$_{bx}$ and `R`$_{bx}$ have dimensionality *H*.

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

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

**Layer Creation And Configuration Methods**

Usage with the C++ API:

- `INetworkDefinition::addRNNv2(X, L, H, `$T_{max}$` for a batch, operation)`
- `IRNNv2Layer::setOperation(operation)`
- `IRNNv2Layer::setDirection(direction)`
- `IRNNv2Layer::setInputMode(mode (skip or linear))`
- `IRNNv2Layer::setWeightsForGate(W or R weight for a cell)`
- `IRNNv2Layer::setBiasForGate(W`$_b$` or R`$_b$` bias for a cell)`
- `IRNNv2Layer::setHiddenState(H`$_0$`)`
- `IRNNv2Layer::setCellState(C`$_0$`)`
- `IRNNv2Layer::setSequenceLengths(`$T$` for each batch element)`

Usage with the Python API:

- ▸ `NetworkDefinition::add_rnnv2(X, L, H, `$T_{max}$` for a batch, operation)`
- ▸ `RNNv2Layer::set_operation(operation)`
- ▸ `RNNv2Layer::set_direction(direction)`
- ▸ `RNNv2Layer::set_input_mode(mode (skip or linear))`
- ▸ `RNNv2Layer::set_weights_for_gate(W or R weight for a cell)`
- ▸ `RNNv2Layer::set_bias_for_gate(`$W_b$` or `$R_b$` bias for a cell)`
- ▸ `RNNv2Layer::set_hidden_state(`$H_0$`)`
- ▸ `RNNv2Layer::set_cell_state(`$C_0$`)`
- ▸ `RNNv2Layer::set_sequence_lengths(T for each batch element)`

**Conditions And Limitations**

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

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

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

RNNv2 supports FP32 and FP16 data type for input and output, hidden, and cell tensors. RNNv2 supports INT32 data type only for the sequence length tensor.

See the C++ IRNNv2 Layer method or the Python RNNv2Layer method for further details.

# A.1.18. RNN Layer (IRNNLayer)

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

**Layer Description**

This layer is identical to the RNNv2 layer (see below) in functionality, but contains additional limitations as described in the Conditions and Limitations section.

**Layer Creation And Configuration Methods**

Usage with the C++ API:

- ▸ `INetworkDefinition::addRNN(X, L, H, `$T_{max}$` for a batch, operation)`
- ▸ `IRNNLayer::setOperation(operation)`
- ▸ `IRNNLayer::setDirection(direction)`
- ▸ `IRNNLayer::setInputMode(mode (skip or linear))`

- ▸ `IRNNLayer::setWeights(structure containing W, R weights for all cells)`
- ▸ `IRNNLayer::setBias(structure containing` $W_b$ `or` $R_b$ `biases for all cells)`
- ▸ `IRNNLayer::setHiddenState(`$H_0$`)`
- ▸ `IRNNLayer::setCellState(`$C_0$`)`

Usage with the Python API:

- ▸ `NetworkDefinition::add_rnn(X, L, H,` $T_{max}$ `for a batch, operation)`
- ▸ `RNNLayer::set_operation(operation)`
- ▸ `RNNLayer::set_direction(direction)`
- ▸ `RNNLayer::set_input_mode(mode (skip or linear))`
- ▸ `RNNLayer::set_weights(structure containing W, R weights for all cells)`
- ▸ `RNNLayer::set_bias(structure containing` $W_b$ `or` $R_b$ `biases for all cells)`
- ▸ `RNNLayer::set_hidden_state(`$H_0$`)`
- ▸ `RNNLayer::set_cell_state(`$C_0$`)`

**Conditions And Limitations**

Unlike the RNNv2 layer, the legacy RNN layer does not support specifying sequence lengths via an input tensor.

The legacy RNN layer does not support arbitrary batch dimensions, and requires that input tensor data be specified using the dimension ordering: sequence length $T$, batch size $N$, embedding size $E$. In contrast, the RNNv2 layer requires that tensor data be specified using the dimension ordering: batch size $N$, sequence length $T$, embedding size $E$.

All limitations that apply to the RNNv2 layer also apply to the legacy RNN layer.

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

# A.1.19. Scale Layer

The Scale layer 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 scale layer performs a per-tensor, per-channel or per-element transformation to produce an output tensor $B$ of the same dimensions. The transformations corresponding to each mode are:

**ScaleMode::kUNIFORM tensor-wise transformation**

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

**ScaleMode::kCHANNEL channel-wise transformation**

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

**ScaleMode::kELEMENTWISE element-wise transformation**

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

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

### Layer Creation And Configuration Methods

Usage with the C++ API:

▸ `INetworkDefinition::addScale(A, mode, shift, scale, power)`

▸ `IScaleLayer::setMode(mode)`

▸ `IScaleLayer::setShift(shift)`

▸ `IScaleLayer::setScale(scale)`

▸ `IScaleLayer::setPower(power)`

Usage with the Python API:

▸ `NetworkDefinition::add_scale(A, mode, shift, scale, power)`

▸ `ScaleLayer::set_mode(mode)`

▸ `ScaleLayer::set_shift(shift)`

▸ `ScaleLayer::set_scale(scale)`

▸ `ScaleLayer::set_power(power)`

### Conditions And Limitations

`A` must have 3 or more dimensions.

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

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

## A.1.20. Shuffle Layer

The Shuffle layer implements a reshape and transpose operator for tensors.

### Layer Description

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

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

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

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

### Layer Creation And Configuration Methods

Usage with the C++ API:

- ▶  `INetworkDefinition::addShuffle(A)`
- ▶  `IShuffleLayer::setFirstTranspose(p1)`
- ▶  `IShuffleLayer::setReshapeDimensions(b)`
- ▶  `IShuffleLayer::setSecondTranspose(p2)`

Usage with the Python API:

- ▶  `NetworkDefinition::add_shuffle(A)`
- ▶  `ShuffleLayer::set_first_transpose(p1)`
- ▶  `ShuffleLayer::set_reshape_dimensions(b)`
- ▶  `ShuffleLayer::set_second_transpose(p2)`

### Conditions And Limitations

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

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

## A.1.21. SoftMax Layer

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

### Layer Description

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

The SoftMax function $S$ for a slice $\mathbf{x}$ is defined as:

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

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

### Layer Creation And Configuration Methods

Usage with the C++ API:

- `INetworkDefinition::addSoftMax(A)`
- `ISoftMaxLayer::setAxes(i)`

Usage with the Python API:

- `NetworkDefinition::add_softmax(A)`
- `SoftmaxLayer::set_axis(i)`

### Conditions And Limitations

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

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

## A.1.22. TopK Layer

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

### Layer Description

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

The output values are:

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

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

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

**Layer Creation And Configuration Methods**

Usage with the C++ API:

▸ `INetworkDefinition::addTopK(A, operation, K, 2`$^i$`)`

▸ `ITopKLayer::setOperation(operation)`

▸ `ITopKLayer::setK(K)`

▸ `ITopKLayer::setReduceAxes(2`$^i$`)`

Usage with the Python API:

▸ `NetworkDefinition::add_top_k(A, operation, K, 2`$^i$`)`

▸ `TopKLayer::set_operation(operation)`

▸ `TopKLayer::set_k(K)`

▸ `TopKLayer::set_reduce_axes(2`$^i$`)`

**Conditions And Limitations**

The $K$ value must be 1024 or less. Only one axis can be searched to find the top $K$ minimum or maximum values; this axis cannot be the batch dimension.

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

# A.1.23. Unary Layer

The Unary layer supports pointwise unary operations.

**Layer Description**

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

▸ `exp: B = e^A`

▸ `abs: B = |A|`

▸ `log: B = ln(A)`

▸ `sqrt: B = √A (rounded to nearest even mode)`

▸ `neg: B = -A`

▸ `recip: B = 1 / A (reciprocal) in rounded to nearest even mode`

**Layer Creation And Configuration Methods**

Usage with the C++ API:

▸ `INetworkDefinition::addUnary(A, operation)`

▸ `IUnaryLayer::setOperation(operation)`

Usage with the Python API:

▸ `NetworkDefinition::add_unary(A, operation)`

▸ **`UnaryLayer::set_operation(operation)`**

**Conditions And Limitations**

Input and output can be zero to 7 dimensional tensors.

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

# A.2. Data Format Descriptions

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

**Data type format**

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

**Layout format**

The layout format determines the ordering in which values are stored. Typically, batch dimensions are the leftmost dimensions, and the other dimensions refer to aspects of each data item such as `C` is channel, `H` is height, and `W` is width, in images. Ignoring batch sizes, which are always preceding these, `C`, `H`, and `W` are typically sorted as `CHW` #unique_193/unique_193_Connect_42_fig1 or `HWC` #unique_193/unique_193_Connect_42_fig2.
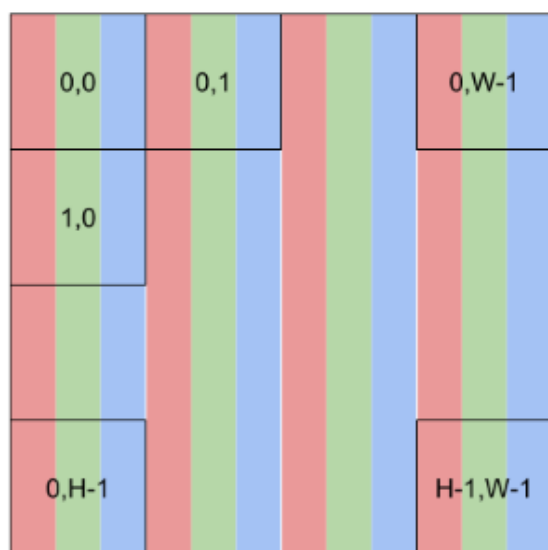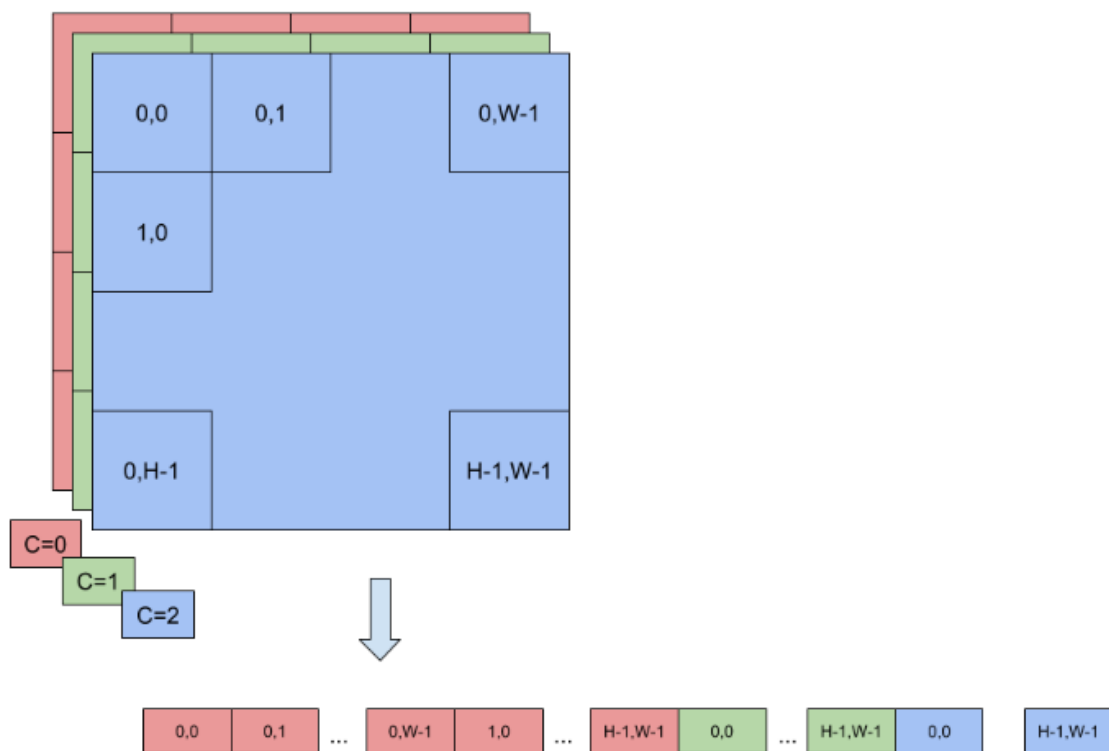
To enable faster computations, more formats are defined to pack together channel values and use reduced precision. For this reason, TensorRT also supports formats `NC/2HW2` and `NHWC8`.

In `NC/2HW2`, pairs of channel values are packed together in each `HxW` matrix (with an empty value in the case of an odd number of channels). The result is a format in which the values of `#C/2#HxW` matrices are pairs of values of two consecutive channels #unique_193/unique_193_Connect_42_fig3; notice that this ordering interleaves dimensions as values of channels that have stride `1` if they are in the same pair and stride `2xHxW` otherwise.
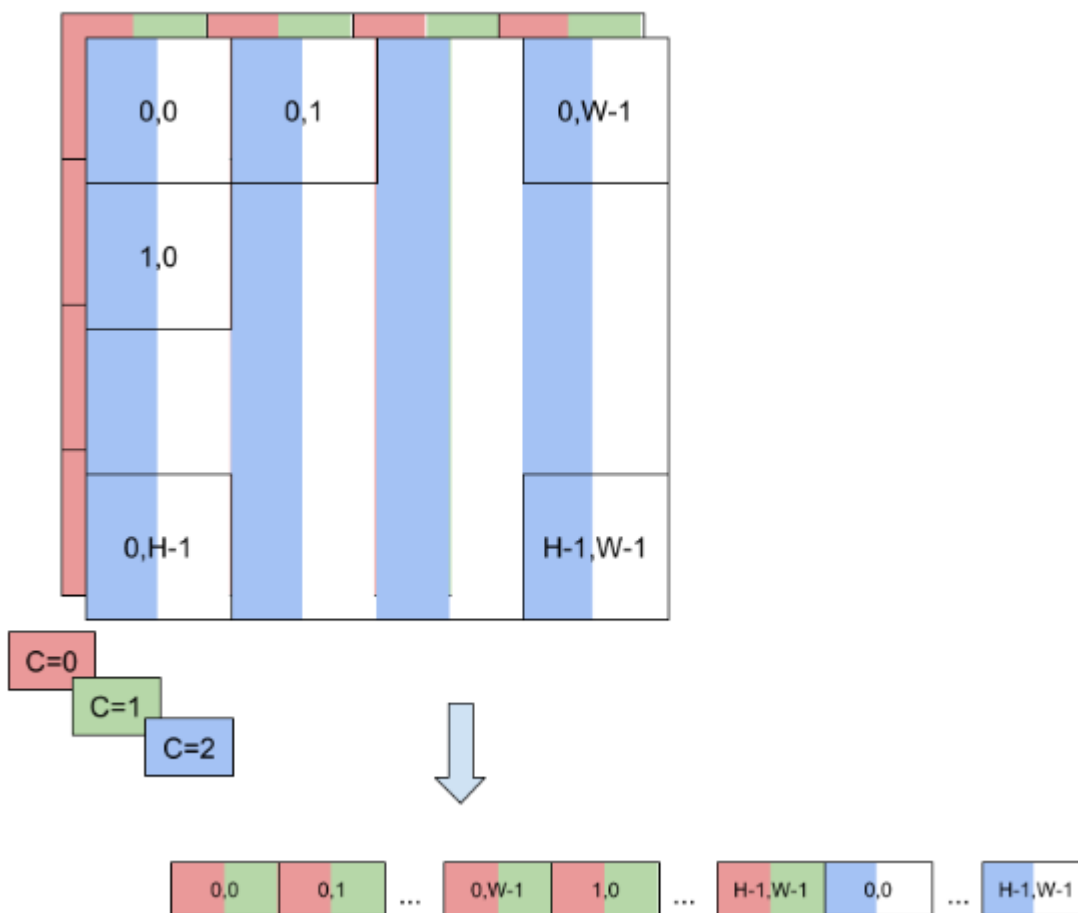


In `NHWC8`, the entries of an `HxW` matrix include the values of all the channels #unique_193/unique_193_Connect_42_fig4. In addition, these values are packed together in `#C/8#` 8-tuples and `C` is rounded up to the nearest multiple of 8.
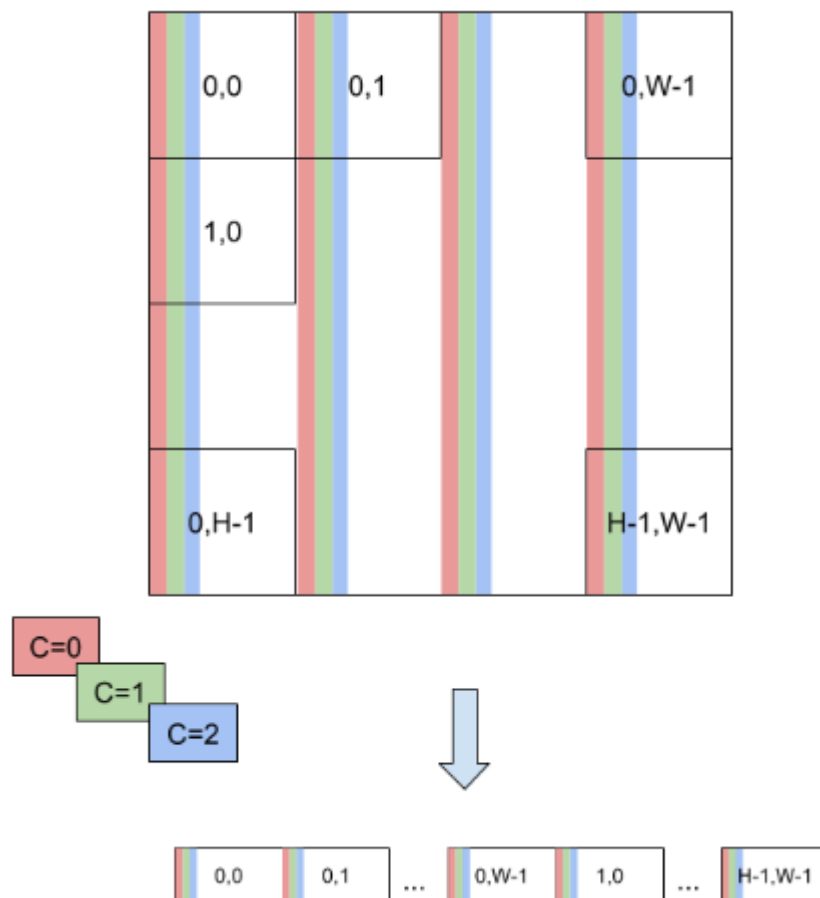
# A.3. Command Line Wrapper

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

The command line arguments are as follows:

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

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

Optional params:
  --uffInput=<name>,C,H,W Input blob names along with their
 dimensions for UFF parser
  --model=<file>        Caffe model file (default = no model,
 random weights used)
```

```
 --batch=N              Set batch size (default = 1)
 --device=N             Set cuda device to N (default = 0)
 --iterations=N         Run N iterations (default = 10)
 --avgRuns=N            Set avgRuns to N - perf is measured as an
average of avgRuns (default=10)
 --percentile=P         For each iteration, report the percentile
time at P percentage (0<P<=100, default = 99.0%)
 --workspace=N          Set workspace size in megabytes (default =
16)
 --fp16                 Run in fp16 mode (default = false).
Permits 16-bit kernels
 --int8                 Run in int8 mode (default = false).
Currently no support for ONNX model.
 --verbose              Use verbose logging (default = false)
 --hostTime             Measure host time rather than GPU time
(default = false)
 --engine=<file>        Generate a serialized TensorRT engine
 --calib=<file>         Read INT8 calibration cache file.
Currently no support for ONNX model.
 --useDLA=N        Enable execution on DLA for all layers that
support DLA. Value can range from 1 to N, where N is the number
of DLA engines on the platform. Set the --fp16 flag as well for
DLA
 --allowGPUFallback   If --useDLA flag is present and if a layer
cannot run on DLA, then run it on GPU.
```

For example:

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

If no model is supplied, random weights are generated.

# A.4. ACKNOWLEDGEMENTS

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

**Google Protobuf**

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

▸ Atomicops support for generic gcc, located in **src/google/protobuf/stubs/ atomicops_internals_generic_gcc.h**. This file is copyrighted by Red Hat Inc.

▸ Atomicops support for AIX/POWER, located in **src/google/protobuf/stubs/ atomicops_internals_power.h**. This file is copyrighted by Bloomberg Finance LP.

Copyright 2014, Google Inc. All rights reserved.

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

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

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

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

### Google Flatbuffers

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

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

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

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

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

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

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

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

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

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

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

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

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where

such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   a. You must give any other recipients of the Work or Derivative Works a copy of this License; and

   b. You must cause any modified files to carry prominent notices stating that You changed the files; and

   c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

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

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

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall

supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

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

END OF TERMS AND CONDITIONS

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

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

Copyright 2014 Google Inc.

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

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

## BVLC Caffe

COPYRIGHT

All contributions by the University of California:

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

All other contributions:

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

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

LICENSE

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

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

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

(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CONTRIBUTION AGREEMENT

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

### half.h

The MIT License

Copyright (c) 2012-2013 Christian Rau

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.

www.nvidia.com