TENSORRT 3.0

DU-08602-001_v3.0 | November 2017

Developer Guide

TABLE OF CONTENTS

Chapter 1. OVERVIEW

NVIDIA TensorRT™ is a C++ library that facilitates high performance inference on NVIDIA graphics processing units (GPUs). TensorRT takes a network definition and optimizes it by merging tensors and layers, transforming weights, choosing efficient intermediate data formats, and selecting from a large kernel catalog based on layer parameters and measured performance.

TensorRT includes import methods to help you express your trained deep learning model for TensorRT to optimize and run. It is an optimization tool that applies graph optimization and layer fusion and finds the fastest implementation of that model leveraging a diverse collection of highly optimized kernels, and a runtime that you can use to execute this network in an inference context.

TensorRT includes an infrastructure that allows you to leverage high speed reduced precision capabilities of Pascal and Volta GPUs as an optional optimization.

TensorRT is built with gcc 4.8.

1.1. TensorRT Layers

TensorRT directly supports the following layer types:

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

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

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

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

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

Flatten
The Flatten layer flattens the input while maintaining the batch_size. Assumes that the first dimension represents the batch. The Flatten layer can only be placed in front of the Fully Connected layer.

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

LRN
The LRN layer implements cross-channel Local Response Normalization.

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

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

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

RNN
The RNN layer implements recurrent layers, such as, recurrent neural network (RNN), Gated Recurrent Units (GRU), and long short-term memory (LSTM). Supported types are **`RNN`**, **`GRU`**, and **`LSTM`**.

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

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

SoftMax
The SoftMax layer implements a cross-channel SoftMax.

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

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

- Batch Normalization can be implemented using the TensorRT Scale layer.
- The operation the Convolution layer performs is actually a correlation. Therefore, it is a consideration if you are formatting weights to import via TensorRT API, rather than via the NVCaffe™ parser library.

1.2. Key Concepts

Ensure you are familiar with the following key concepts:

Network definition

A network definition consists of a sequence of layers and a set of tensors.

Layer

Each layer computes a set of output tensors from a set of input tensors. Layers have parameters, for example, **`convolution size`**, **`stride`**, and **`convolution filter weights`**.

Tensor

A tensor is either an input to the network, or an output of a layer. Tensors have a data-type specifying their precision, for example, 16- and 32-bit floats, and three dimensions, for example, channels, width, and height. The dimensions of an input tensor are defined by the application, and for output tensors they are inferred by the builder. The supported dimension is **`N (P_1 P_2 ...) CHW`**, where **`P_1`**, **`P_2`**,and so on are index dimensions. Tensors can have at most **`Dims::MAX_DIMENSIONS`** dimensions in total, where that constant is set to 8.

Each layer and tensor has a name, which is useful when profiling or reading TensorRT build log.

When using NvCaffeParser, tensor and layer names are taken from the NVCaffe prototxt file.

1.3. TensorRT API's

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

In addition to the primary API in C++. TensorRT includes TensorRT python API bindings. The TensorRT python API currently supports all functionality except for RNNs. It introduces compatibility with NumPy arrays for layer weights and through the use of PyCUDA, input and output data. A set of utility functions have also been provided to address common tasks developers may face including NVCaffe model parsing, UFF model parsing from a stream, and from a UFF file loading and writing PLAN files. These are located in **`tensorrt.utils`**.

A workflow is provided for those looking to implement custom layers for use in Python in **`tensorrt.examples.plugin`**.

Additional dependencies are needed, for example, **`swig >= 3.0`**, **`libnvinfer-dev`**, where the developer can define a C++ implementation which can then be used in Python.

1.3.1. Python Samples

Python interfaces support all of the functionality that was previously available only through C++ interfaces. These include:

- the NvCaffeParser
- the nvinfer API's for graph definition
- the builder to create optimized inference engines
- the inference-time interface used to execute the engine
- the calls used to register a custom layer implementation

You can find the Python examples in the **`{PYTHON LIB DIR}/site-packages/tensorrt/examples`** directory.

The TensorRT package comes with a couple sample application implementations. These can be found depending on whether you installed TensorRT across the system or just for the user.

Table 1 Sample application package location

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

For more information about importing a trained model to TensorRT using Python, see NVCaffe Python Workflow, TensorFlow Python Workflow, and Converting A Model From An Unsupported Framework To TensorRT With The TensorRT Python API.

1.3.2. Python Workflows

Python is a popular and highly productive language for data science in general and is used in many deep learning frameworks. Sample applications are provided for the following use cases:

1. There is an existing TensorFlow™ (or other UFF compatible framework) model that a developer wants to try out with TensorRT.
   a. Convert a TensorFlow model to TensorRT
2. There is a NVCaffe model that a developer wants to try out with TensorRT.
   a. Convert a NVCaffe model to TensorRT
3. A developer wants to deploy a TensorRT engine as part of a larger application such as a web backend.
4. A developer wants to try out TensorRT with a model trained with a framework not currently supported by UFF and not trained by NVCaffe.

For steps on how to complete these use cases, see NVCaffe Workflow, TensorFlow Workflow, and Converting A Model From An Unsupported Framework To TensorRT With The TensorRT Python API.

Chapter 2. TENSORRT WORKFLOWS

The following table lists TensorRT features and their supported API's.

Table 2

Feature	C++	Python	NvCaffeParser	NvUffParser
CNNs	yes	yes	yes	yes
RNNs	yes	yes	no	no
INT8 Calibration	yes	yes	NA	NA
Asymmetric Padding	yes	yes	no	no

The following table lists TensorRT features and their supported platforms.

Table 3

Feature	Linux x86	Linux aarch64	Android aarch64	QNX aarch64
Supported Compute Unified Device Architecture® (CUDA) versions	8.0, 9.0	8.0, 9.0	9.0	9.0
Supported CUDA® Deep Neural Network library™ (cuDNN) versions	7.0	7.0	7.0	7.0
TensorRT python API	yes	no	no	no
NvUffParser	yes	no	no	no

Serialized engines are not portable across platforms or TensorRT versions.

2.1. Key Concepts

Ensure you are familiar with the following key concepts:

UFF

Universal Framework Format (UFF) is a data format that describes an execution graph for a DNN (Deep Neural Network), and bindings from that execution graph to its inputs and outputs. It has a well-specified core language, but will also support extensions to its core operators, and entirely custom operators.

The format consists of:

- Concrete syntax for the serialization format, in the form of a protobuf schema.
- A definition of validity for each operator, expressed as a set of python descriptors.
- Documentation of the execution behavior of each core operator (for future delivery).

PLAN file

The PLAN file is the serialized data that the runtime engine uses to execute the network. It includes weights, a schedule for the kernels to execute the network, and information about the network that the application can query in order to determine how to bind input and output buffers.

2.2. Workflow Diagrams

Figure 1 shows a typical development workflow, where the user trains the model on data to produce a trained network. That trained network can then be used for inference.

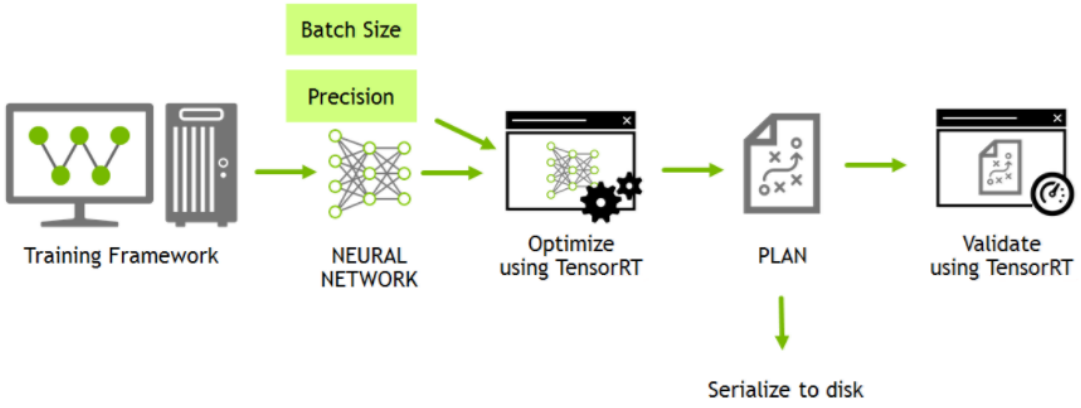

Figure 1 Typical development workflow.

Figure 1 is importing the trained network into TensorRT. The user imports the trained network into TensorRT, which optimizes the network to produce a PLAN. That PLAN is

then used for inference, for example, to validate that optimization has been performed correctly.

The PLAN can also be serialized to disk so that it can be later reloaded into the TensorRT runtime without having to perform the optimization step again (see Figure 2).

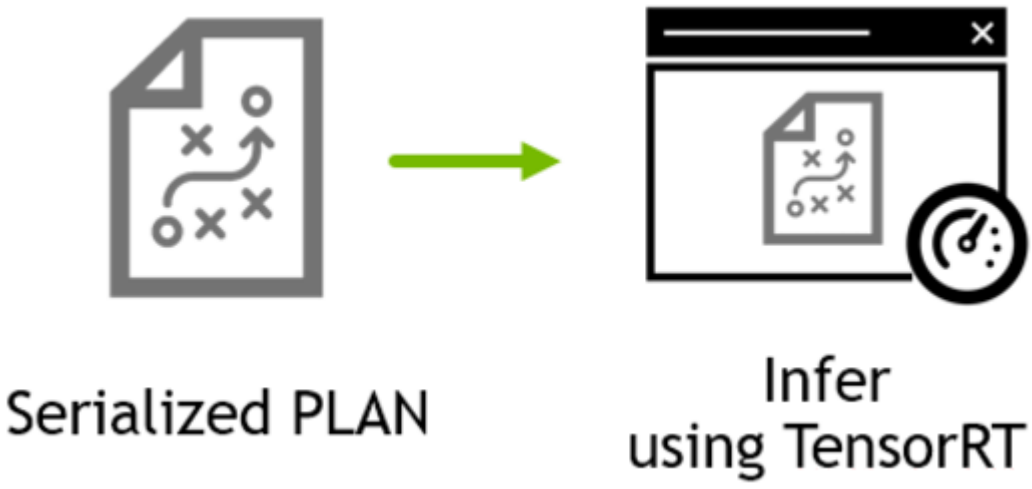

Figure 2 Typical production workflow.

2.3. Exporting From Frameworks

2.3.1. NVCaffe Workflow

2.3.1.1. NVCaffe C++ Workflow

TensorRT can import NVCaffe models directly, via the NvCaffeParser interface.

An example of using the NvCaffeParser parser can be found in SampleMNIST. There, TensorRT network definition structure is populated directly from a NVCaffe model using the NvCaffeParser library:

```
INetworkDefinition* network = builder->createNetwork();
CaffeParser* parser = createCaffeParser();
std::unordered_map<std::string, infer1::Tensor> blobNameToTensor;
const IBlobNameToTensor* blobNameToTensor =
    parser->parse(locateFile(deployFile).c_str(),
                              locateFile(modelFile).c_str(),
                              *network,
                              DataType::kFLOAT);
```

The NvCaffeParser is instructed to generate a network whose weights are 32-bit floats, but we could instead have passed **`DataType::kHALF`** to generate a model with 16-bit weights.

As well as populating the network definition, the parser returns a dictionary that maps from NVCaffe blob names to TensorRT tensors.

A TensorRT network definition has no notion of in-place operation, for example, the input and output tensors of a ReLU are different. When a NVCaffe network 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 creates 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.

Since the NVCaffe model does not tell us which tensors are the outputs of the network, we need to specify these explicitly after parsing:

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

There is no restriction on the number of output tensors, however, marking a tensor as an output, may prohibit some optimizations on that tensor.

Do not immediately release the parser object because the network definition holds weights by reference into the NVCaffe model, not by value. It is only during the build process that the weights are read from the NVCaffe model.

2.3.1.2. NVCaffe Python Workflow

TensorRT 3.0 adds support for a TensorRT python API to load and optimize NVCaffe models which can then be executed and stored as portable PLAN files. The following sample explains the workflow you can use to do so. In this sample, you will learn how to use TensorRT to optimize NVCaffe models in Python.

In Python, the TensorRT library is referred to as **`tensorrt`**.

You can import **`tensorrt`** as you would import any other package. For example:

```
import tensorrt as trt
```

There are also some common tools that are used With TensorRT python API, there are typically some common tools used, for example, PyCUDA and NumPy. PyCUDA handles the CUDA operations needed to allocate memory on your GPU and to transfer data to the GPU and results back to the CPU. NumPy is a well used tool to store and move data.

```
import pycuda.driver as cuda
import pycuda.autoinit
import numpy as np
```

For this example, let's import an image processing library (pillow in this case) and **`randint`**.

```
from random import randint
from PIL import Image
```

Since we are converting an NVCaffe model, we also need to use the **`caffeparser`**, which is located in **`tensorrt.parsers`**.

```
from tensorrt.parsers import caffeparser
```

Typically, the first thing you will do is create a logger, which is used in many places during the model conversion and inference process. There is a simple logger included in **tensorrt.infer.ConsoleLogger**.

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

Next, define some constants about your model. In this example, we will classify Deep Learning GPU Training System™ (DIGITS) from the MNIST dataset.

```
INPUT_LAYERS = ['data']
OUTPUT_LAYERS = ['prob']
INPUT_H = 28
INPUT_W =  28
OUTPUT_SIZE = 10
```

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

```
MODEL_PROTOTXT = '/data/mnist/mnist.prototxt'
CAFFE_MODEL = '/data/mnist/mnist.caffemodel'
DATA = '/data/mnist/'
IMAGE_MEAN = '/data/mnist/mnist_mean.binaryproto'
```

Now it's time to create the engine. The TensorRT python API provides some nice utilities to make this much simpler. Let's use the NVCaffe model converter utility in **tensorrt.utils**. We provide a logger, a path to the model prototxt, the model file, the max batch size, the max workspace size, the output layers and the data type of the weights.

```
engine = tensorrt.utils.caffe_to_trt_engine(G_LOGGER,
                                            MODEL_PROTOTXT,
                                            CAFFE_MODEL,
                                            1,
                                            1 << 20,
                                            OUTPUT_LAYERS,
                                            trt.infer.DataType.FLOAT)
```

Let's generate a test case for our engine.

```
rand_file = randint(0,9)
path = DATA + str(rand_file) + '.pgm'
im = Image.open(path)
arr = np.array(im)
img = arr.ravel()
print("Test Case: " + str(rand_file))
```

Next, apply the mean to the input image. We have this stored in a **.binaryproto** file which we use the **caffeparser** to read.

```
parser = caffeparser.create_caffe_parser()
mean_blob = parser.parse_binary_proto(IMAGE_MEAN)
parser.destroy()
mean = mean_blob.get_data(INPUT_W ** 2) #NOTE: This is different than the C++
 API, you must provide the size of the data
data = np.empty([INPUT_W ** 2])
for i in range(INPUT_W ** 2):
    data[i] = float(img[i]) - mean[i]
mean_blob.destroy()
```

Create a runtime for inference and create a context for your engine.

```
runtime = trt.infer.create_infer_runtime(G_LOGGER)
context = engine.create_execution_context()
```

Run inference. Start by making sure your data is in the correct data-type (for example, FP32 for this model). Then, create an empty array on the CPU to hold your results from inference.

```
assert(engine.get_nb_bindings() == 2)
#convert input data to Float32
img = img.astype(np.float32)
#create output array to receive data
output = np.empty(OUTPUT_SIZE, dtype = np.float32)
```

Now, allocate memory on the GPU with PyCUDA and register them with the engine. The size of the allocations is the size of the input and expected output ***** the batch size.

```
d_input = cuda.mem_alloc(1 * img.size * img.dtype.itemsize)
d_output = cuda.mem_alloc(1 * output.size * output.dtype.itemsize)
```

The engine needs bindings provided as pointers to the GPU memory. PyCUDA lets us do this for memory allocations by casting those allocations to **ints**.

```
bindings = [int(d_input), int(d_output)]
```

We also are going to create a CUDA stream to run inference in.

```
stream = cuda.Stream()
```

Next, transfer the data to the GPU, run inference and then copy the results back.

```
#transfer input data to device
cuda.memcpy_htod_async(d_input, img, stream)
#execute model
context.enqueue(1, bindings, stream.handle, None)
#transfer predictions back
cuda.memcpy_dtoh_async(output, d_output, stream)
#synchronize threads
stream.synchronize()
```

Now we have our results. We can just run ArgMax to get a prediction.

```
print("Test Case: " + str(rand_file))
print ("Prediction: " + str(np.argmax(output)))
```

We can also save our engine to a file to use later.

```
trt.utils.write_engine_to_file("/data/mnist/new_mnist.engine",
    engine.serialize())
```

You can then load this engine later by running the following command:

```
new_engine =
trt.utils.load_engine(G_LOGGER,
                      "/data/mnist/new_mnist.engine")
```

As a final step, clean up the context, engine, and runtime:

```
context.destroy()
engine.destroy()
new_engine.destroy()
```

```
runtime.destroy()
```

2.3.1.3. NvCaffeParser

While TensorRT is independent of any framework, the package does include a parser for NVCaffe models named NvCaffeParser.

NvCaffeParser provides a simple mechanism for importing network definitions. NvCaffeParser uses TensorRT layers to implement NVCaffe layers. For example:

- Convolution
- rectified linear unit (ReLU)
- Sigmoid
- hyperbolic tangent (tanh)
- Pooling
- Power
- BatchNorm
- ElementWise (Eltwise)
- Local Response Normalization (LRN)
- InnerProduct (which is what NVCaffe calls the FullyConnected layer)
- SoftMax
- Scale
- Deconvolution

NVCaffe features not currently supported by TensorRT include:

- parametric rectified linear unit (PReLU)
- leaky rectified linear unit (Leaky ReLU)
- Scale (other than per-channel scaling)
- ElementWise (Eltwise) with more than two inputs

NvCaffeParser does not support legacy formats in NVCaffe prototxt; in particular, layer types are expected to be expressed in the prototxt as strings delimited by double quotes.

2.3.2. TensorFlow™ Workflow

2.3.2.1. TensorFlow Python Workflow

TensorRT 3.0 introduces the UFF (Universal Framework Format) parser, a way to import UFF models and generate TensorRT engines. The UFF Toolkit is included in TensorRT 3.0 to support converting TensorFlow models to UFF, thereby allowing TensorFlow users to access the performance gains of TensorRT. In this sample, you will learn how to convert a TensorFlow model (or other UFF compatible framework) to TensorRT with the TensorRT python API.

With the TensorRT python API, you can now go from training in TensorFlow to deploying in TensorRT without leaving a single python interpreter session. For this

example, we are going to train a model to classify handwritten DIGITS and then generate a TensorRT engine for inference.

In Python, the TensorRT library is referred to as **tensorrt**.

Import TensorFlow and its various packages.

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

There are also some common tools that are used with TensorRT python API, there are typically some common tools used, for example, PyCUDA and NumPy. PyCUDA handles the CUDA operations needed to allocate memory on your GPU and to transfer data to the GPU and results back to the CPU. NumPy is a well used tool to store and move data.

```
import pycuda.driver as cuda
import pycuda.autoinit
import numpy as np
from random import randint # generate a random test case
from PIL import Image
import time #import system tools
import os
```

Finally, import the UFF Toolkit to convert the graph from a serialized frozen TensorFlow model to UFF.

```
import uff
```

You can import TensorRT and its parsers by running the following commands:

```
import tensorrt as trt
from tensorrt.parsers import uffparser
```

It is critical to ensure the version of UFF matches the required UFF version by TensorRT. The TensorRT package provides the API to ensure this condition:

```
trt.utils.get_uff_version()
parser = uffparser.create_uff_parser()


def get_uff_required_version(parser):
 return str(parser.get_uff_required_version_major()) + '.'
 + str(parser.get_uff_required_version_minor()) + '.' +
 str(parser.get_uff_required_version_patch())

if trt.utils.get_uff_version() != get_uff_required_version(parser):
 raise ImportError("""ERROR: UFF TRT Required version mismatch""")
```

2.3.2.1.1. Training A Model In TensorFlow

In this example, start by defining some hyper parameters and then define some helper functions to make the code a bit less verbose.

```
STARTER_LEARNING_RATE = 1e-4
BATCH_SIZE = 10
NUM_CLASSES = 10
MAX_STEPS = 5000
IMAGE_SIZE = 28
IMAGE_PIXELS = IMAGE_SIZE ** 2
```

```
OUTPUT_NAMES = ["fc2/Relu"]
```

Notice that we are padding our Conv2d layer. TensorRT expects symmetric padding for layers.

```
def WeightsVariable(shape):
    return tf.Variable(tf.truncated_normal(shape, stddev=0.1, name='weights'))

def BiasVariable(shape):
    return tf.Variable(tf.constant(0.1, shape=shape, name='biases'))

def Conv2d(x, W, b, strides=1):
    # Conv2D wrapper, with bias and relu activation
    filter_size = W.get_shape().as_list()
    pad_size = filter_size[0]//2
    pad_mat = np.array([[0,0],
                        [pad_size,pad_size],
                        [pad_size,pad_size],
                        [0,0]])
    x = tf.pad(x, pad_mat)

    x = tf.nn.conv2d(x,
                     W,
                     strides=[1, strides, strides, 1],
     padding='VALID')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)

def MaxPool2x2(x, k=2):
    # MaxPool2D wrapper
    pad_size = k//2
    pad_mat = np.array([[0,0],
                        [pad_size,pad_size],
                        [pad_size,pad_size],
                        [0,0]])
    return tf.nn.max_pool(x,
                          ksize=[1, k, k, 1],
                          strides=[1, k, k, 1],
                          padding='VALID')
```

We now are going to define a network and then define our loss metrics, training and test steps, our input nodes, and a data loader.

```
 def network(images):
    # Convolution 1
    with tf.name_scope('conv1'):
        weights = WeightsVariable([5,5,1,32])
        biases = BiasVariable([32])
        conv1 = tf.nn.relu(Conv2d(images, weights, biases))
        pool1 = MaxPool2x2(conv1)

    # Convolution 2
    with tf.name_scope('conv2'):
        weights = WeightsVariable([5,5,32,64])
        biases = BiasVariable([64])
        conv2 = tf.nn.relu(Conv2d(pool1, weights, biases))
        pool2 = MaxPool2x2(conv2)
        pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])

    # Fully Connected 1
    with tf.name_scope('fc1'):
        weights = WeightsVariable([7 * 7 * 64, 1024])
        biases = BiasVariable([1024])
        fc1 = tf.nn.relu(tf.matmul(pool2_flat, weights) + biases)
```

```
    # Fully Connected 2
    with tf.name_scope('fc2'):
        weights = WeightsVariable([1024, 10])
        biases = BiasVariable([10])
        fc2 = tf.reshape(tf.matmul(fc1,weights) + biases, shape= [-1,10] ,
 name='Relu')

    return fc2

 def loss_metrics(logits, labels):
    cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
labels=labels,
logits=logits,
name='softmax')
    return tf.reduce_mean(cross_entropy, name='softmax_mean')

def training(loss):
    tf.summary.scalar('loss', loss)
    global_step = tf.Variable(0, name='global_step', trainable=False)
    learning_rate = tf.train.exponential_decay(STARTER_LEARNING_RATE,
    global_step,
    100000,
    0.75,
    staircase=True)
    tf.summary.scalar('learning_rate', learning_rate)
    optimizer = tf.train.MomentumOptimizer(learning_rate, 0.9)
    train_op = optimizer.minimize(loss, global_step=global_step)
    return train_op

def evaluation(logits, labels):
    correct = tf.nn.in_top_k(logits, labels, 1)
    return tf.reduce_sum(tf.cast(correct, tf.int32))

def do_eval(sess,
            eval_correct,
            images_placeholder,
            labels_placeholder,
            data_set,
            summary):
    true_count = 0
    steps_per_epoch = data_set.num_examples // BATCH_SIZE
    num_examples = steps_per_epoch * BATCH_SIZE


for step in range(steps_per_epoch):
        feed_dict = fill_feed_dict(data_set,
                                   images_placeholder,
                                   labels_placeholder)
        log, correctness = sess.run([summary, eval_correct],
    feed_dict=feed_dict)
        true_count += correctness
    precision = float(true_count) / num_examples
    tf.summary.scalar('precision', tf.constant(precision))
    print('Num examples %d, Num Correct: %d Precision @ 1: %0.04f' %
 (num_examples, true_count, precision))
    return log

def placeholder_inputs(batch_size):
    images_placeholder = tf.placeholder(tf.float32,
                                        shape=(None, 28, 28, 1))
    labels_placeholder = tf.placeholder(tf.int32, shape=(None))
    return images_placeholder, labels_placeholder
```

```
def fill_feed_dict(data_set, images_pl, labels_pl):
    images_feed, labels_feed = data_set.next_batch(BATCH_SIZE)
    feed_dict = {
        images_pl: np.reshape(images_feed, (-1,28,28,1)),
        labels_pl: labels_feed,
    }
    return feed_dict
```

We are going to define our training pipeline in function that will return a frozen model with the training nodes removed.

```
def run_training(data_sets):
    with tf.Graph().as_default():
        images_placeholder, labels_placeholder = placeholder_inputs(BATCH_SIZE)
        logits = network(images_placeholder)
        loss = loss_metrics(logits, labels_placeholder)
        train_op = training(loss)
        eval_correct = evaluation(logits, labels_placeholder)
        summary = tf.summary.merge_all()
        init = tf.global_variables_initializer()
        saver = tf.train.Saver()
        gpu_options = tf.GPUOptions(
         per_process_gpu_memory_fraction=0.5)
        sess = tf.Session(
config=tf.ConfigProto(gpu_options=gpu_options))
        summary_writer = tf.summary.FileWriter(
   "/tmp/tensorflow/mnist/log",
   graph=tf.get_default_graph())
        test_writer = tf.summary.FileWriter(
    "/tmp/tensorflow/mnist/log/validation",
    graph=tf.get_default_graph())
        sess.run(init)
        for step in range(MAX_STEPS):
            start_time = time.time()
            feed_dict = fill_feed_dict(data_sets.train,
                                       images_placeholder,
                                       labels_placeholder)
            _, loss_value = sess.run([train_op, loss],
    feed_dict=feed_dict)
            duration = time.time() - start_time
            if step % 100 == 0:
                print('Step %d: loss = %.2f (%.3f sec)' %
(step, loss_value, duration))
                summary_str = sess.run(summary, feed_dict=feed_dict)
                summary_writer.add_summary(summary_str, step)
                summary_writer.flush()
            if (step + 1) % 1000 == 0 or (step + 1) == MAX_STEPS:
                Checkpoint_file = os.path.join(
"/tmp/tensorflow/mnist/log",
"model.ckpt")
                saver.save(sess, checkpoint_file, global_step=step)
                print('Validation Data Eval:')
                log = do_eval(sess,
                              eval_correct,
                              images_placeholder,
                              labels_placeholder,
                              data_sets.validation,
                              summary)
                test_writer.add_summary(log, step)
        graphdef = tf.get_default_graph().as_graph_def()
        frozen_graph = tf.graph_util.convert_variables_to_constants(sess,
 graphdef, OUTPUT_NAMES)
        return tf.graph_util.remove_training_nodes(frozen_graph)
```

Now we are going to load the TensorFlow MNIST data loader and run training. The model has summaries included so you can take at look at the training in TensorBoard.

```
MNIST_DATASETS = input_data.read_data_sets(
'/tmp/tensorflow/mnist/input_data')
tf_model = run_training(MNIST_DATASETS)
```

2.3.2.1.2. Converting A TensorFlow Model To UFF

In this example, we are going to convert a TensorFlow model into a serialized UFF model using the UFF Toolkit and the helper function **`uff.from_tensorflow`**. To convert a model we need to at least provide the model stream and the names of the desired output nodes. The UFF Toolkit also includes a **`uff.from_tensorflow_frozen_model`** function which takes a path to a frozen TensorFlow graph protobuf file.

Both utilities, for example **`uff.from_tensorflow`** (serialized graph) and **`uff.from_tensorflow_frozen_model`** (protobuf file), have options for:

`quiet`
Suppresses conversion logging.

`input_nodes`
Defines a set of input nodes in the graph. The defaults are placeholder nodes.

`text`
Enables you to save a readable version of a UFF model alongside the binary UFF.

`list_nodes`
Lists the nodes in the graph.

`output_filename`
If provided, writes the model out to the specified filepath, instead of returning a serialized model.

To convert a model to UFF, run the following command:

```
uff_model = uff.from_tensorflow(tf_model, ["fc2/Relu"])
```

2.3.2.1.3. Importing A UFF Model Into TensorRT

Now that we have a UFF model, we can generate a TensorRT engine by creating a logger for TensorRT.

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

Create a UFF parser and identify the desired input and output nodes.

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

Pass the logger, parser, the UFF model stream, and some settings (max batch size and max workspace size) to a utility function that will create the engine for us.

```
engine = trt.utils.uff_to_trt_engine(G_LOGGER,
    uff_model,
    parser,
    1,
    1 << 20)
```

Allocate some memory on the CPU to use while we have an active engine.

```
host_mem = parser.hidden_plugin_memory()
```

Destroy the parser.

```
parser.destroy()
```

Get a test case from the TensorFlow data loader (converting it to FP32).

```
img, label = MNIST_DATASETS.test.next_batch(1)
img = img[0]
#convert input data to Float32
img = input_img.astype(np.float32)
label = label[0]
```

Create a runtime and an execution context for the engine.

```
runtime = trt.infer.create_infer_runtime(G_LOGGER)
context = engine.create_execution_context()
```

Next, allocate the memory on the GPU and allocate memory on the CPU to hold results after inference. The size of the allocations is the size of the input and expected output ***** the batch size.

```
output = np.empty(10, dtype = np.float32)
#allocate device memory
d_input = cuda.mem_alloc(1 * img.size * img.dtype.itemsize)
d_output = cuda.mem_alloc(1 * output.size * output.dtype.itemsize)
```

The engine needs bindings provided as pointers to the GPU memory. PyCUDA lets us do this for memory allocations by casting those allocations to **ints**.

```
bindings = [int(d_input), int(d_output)]
```

Create a CUDA stream to run inference in.

```
stream = cuda.Stream()
```

Transfer the data to the GPU, run inference, and the copy the results back.

```
#transfer input data to device
cuda.memcpy_htod_async(d_input, img, stream)
#execute model
context.enqueue(1, bindings, stream.handle, None)
#transfer predictions back
cuda.memcpy_dtoh_async(output, d_output, stream)
#synchronize threads
stream.synchronize()
```

Now that we have our results, run ArgMax to get a prediction.

```
print("Test Case: " + str(label))
print ("Prediction: " + str(np.argmax(output)))
```

We can also save our engine to a file to use later.

```
trt.utils.write_engine_to_file("/data/mnist/tf_mnist.engine",
 engine.serialize())
```

You can then load this engine later by using **tensorrt.utils.load_engine**.

```
new_engine = trt.utils.load_engine(G_LOGGER, "/data/mnist/new_mnist.engine")
```

As a final step, clean up your context, engine, and runtime.

```
context.destroy()
engine.destroy()
new_engine.destroy()
runtime.destroy()
```

2.3.2.2. Exporting TensorFlow To A UFF File

TensorRT has a UFF parser that is able to read the **.uff** file and create the inference engine. Therefore, to convert a TensorFlow model to a TensorRT runnable file, you must freeze the TensorFlow code to **.pb** and convert it to **.uff** using the **convert-to-uff** utility.

Although networks can use NHWC and NCHW, TensorFlow users are encouraged to convert their networks to use NCHW data ordering explicitly in order to achieve the best possible performance.

1. From a TensorFlow or Keras (with TensorFlow backend) code, freeze the graph to generate a **.pb** (protobuf) file. For more information and sample code, see Sample Code 1: Freezing the Graph from TensorFlow to .pb and Sample Code 2: Freezing Keras Code to .pb.
2. Using the **convert-to-uff.py** utility, convert the **.pb** frozen graph to **.uff** format file. For more information and sample code, see Sample Code 3: Running convert_to_uff.py.

2.3.2.2.1. Freezing The Graph From TensorFlow To .pb

TensorFlow provides a **freeze_graph** function whose usage can be seen here: freeze_graph_test.py. To use this path, the graph must be saved as a **.pb** using **graph_io.write_graph**, along with checkpoints. Then, **freeze_graph** can be called to convert variables to const ops to a new frozen **.pb**. Additional information about **freeze_graph** can be found at the links below.

Brief overview of why **freeze_graph** is needed and what it does: freezing

Tutorials with examples of using **freeze_graph**:

- Exporting trained TensorFlow models to C++ the RIGHT way!
- TensorFlow: How to freeze a model and serve it with a python API

2.3.2.2.2. Sample Code 1: Freezing Keras Code To .pb

To convert a Keras model, use the following code:

```
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, "")
```

2.3.2.2.3. Sample Code 2: Running `convert-to-uff`

In order to convert the **`.pb`** frozen graph to **`.uff`** format file, see the following sample code:

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

In order the figure out **`name_of_output_tensor`** you can list the TensorFlow layers:

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

2.3.2.2.4. Supported TensorFlow Operations

The current exporter supports the following native layers of TensorFlow:

- Placeholder is converted into an UFF Input layer.
- Const is converted into a UFF Const layer.
- Add, Sub, Mul, Div, Minimum and Maximum are converted into a UFF Binary layer.
- BiasAdd is converted into a UFF Binary layer.
- Negative, Abs, Sqrt, Rsqrt, Pow, Exp and Log are converted into a UFF Unary layer.
- FusedBatchNorm is converted into a UFF Batchnorm layer.
- Tanh, Relu and Sigmoid are converted into a UFF Activation layer.
- SoftMax is converted into a UFF SoftMax layer.
- Mean is converted into a UFF Reduce layer.
- ConcatV2 is converted into a UFF Concat layer.
- Reshape is converted into a UFF Reshape layer.
- Transpose is converted into a UFF Transpose layer.
- Conv2D and DepthwiseConv2dNative are converted into a UFF Conv layer.
- ConvTranspose2D are converted into a UFF ConvTranspose layer.
- MaxPool and AvgPool are converted into a UFF Pooling layer.
- Pad is supported if followed by one of these TensorFlow layers: Conv2D, DepthwiseConv2dNative, MaxPool, and AvgPool.

2.3.2.3. NvUffParser

For an illustration of how to import UFF files into TensorRT, see SampleUffMNIST UFF Usage.

2.3.3. Converting A Model From An Unsupported Framework To TensorRT With The TensorRT Python API

With the release of UFF (Universal Framework Format), converting models from compatible frameworks to TensorRT engines is much easier. However, there may be frameworks that do not currently have UFF exporters or never will. The TensorRT python API provides a path forward for Python based frameworks with its NumPy compatible layer weights.

For this example we are going to be using PyTorch™ , and show how you can train a model then manually convert the model into a TensorRT engine.

In Python, the TensorRT library is referred to as **tensorrt**.

There are also some common tools that are used with TensorRT python API, there are typically some common tools used, for example, PyCUDA and NumPy. PyCUDA handles the CUDA operations needed to allocate memory on your GPU and to transfer data to the GPU and results back to the CPU. NumPy is a well used tool to store and move data.

```
import pycuda.driver as cuda
import pycuda.autoinit
import numpy as np
```

Import PyTorch and its various packages.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.autograd import Variable
```

You can import **tensorrt** as you would import any other package. For example:

```
import tensorrt as trt
```

2.3.3.1. Training A Model In PyTorch

In this example, start by setting some hyper parameters, create a data loader, define your network, set your optimizer, and define your train and test steps.

```
BATCH_SIZE = 64
TEST_BATCH_SIZE = 1000
EPOCHS = 3
LEARNING_RATE = 0.001
SGD_MOMENTUM = 0.5
SEED = 1
LOG_INTERVAL = 10  #Enable Cuda

torch.cuda.manual_seed(SEED)  #Dataloader
kwargs = {'num_workers': 1, 'pin_memory': True}

train_loader  = torch.utils.data.DataLoader(
```

```
    datasets.MNIST('/tmp/mnist/data', train=True, download=True,
 transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
        ])),
    batch_size=BATCH_SIZE,
    shuffle=True,
    **kwargs)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('/tmp/mnist/data', train=False,
 transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
        ])),
    batch_size=TEST_BATCH_SIZE,
    shuffle=True,
    **kwargs)  #Network

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, kernel_size=5)
        self.conv2 = nn.Conv2d(20, 50, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(800, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.max_pool2d(self.conv1(x), kernel_size=2, stride=2)
        x = F.max_pool2d(self.conv2(x), kernel_size=2, stride=2)
        x = x.view(-1, 800)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)

model = Net()
model.cuda()  optimizer = optim.SGD(model.parameters(),
   lr=LEARNING_RATE,
   momentum=SGD_MOMENTUM)

def train(epoch):
    model.train()
    for batch, (data, target) in enumerate(train_loader):
        data, target = data.cuda(), target.cuda()
        data, target = Variable(data), Variable(target)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch % LOG_INTERVAL == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'
.format(epoch,
   batch * len(data),
   len(train_loader.dataset),
   100. * batch / len(train_loader),

   loss.data[0]))

def test(epoch):
    model.eval()
    test_loss = 0
    correct = 0
    for data, target in test_loader:
```

```
        data, target = data.cuda(), target.cuda()
        data, target = Variable(data, volatile=True), Variable(target)
        output = model(data)
        test_loss += F.nll_loss(output, target).data[0]
        pred = output.data.max(1)[1]
        correct += pred.eq(target.data).cpu().sum()
    test_loss /= len(test_loader)
    print('Test: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'
.format(test_loss,
  correct,
  len(test_loader.dataset),
  100. * correct / len(test_loader.dataset)))
```

Next, train this model.

```
for e in range(EPOCHS):
    train(e + 1)
    test(e + 1)
```

2.3.3.2. Converting The Model Into A TensorRT Engine

Now that we have a trained model, we need to start converting the model by first extracting the layer weights by getting the **state_dict**.

```
weights = model.state_dict()
```

Start converting the model to TensorRT by first creating a builder and a logger for the build process.

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

Next, create the network by replicating the network structure above and extracting the weights in the form of NumPy arrays from PyTorch. The NumPy arrays from PyTorch reflect the dimensionality of the layers, however, we reshape the layers to flatten the arrays.

TensorRT expects weights in NCHW format, therefore, if your framework uses another format, you may need to pre-process your weights before flattening.

```
network = builder.create_network()

#Name for the input layer, data type, tuple for dimension
data = network.add_input("data",
   trt.infer.DataType.FLOAT,
   (1, 28, 28))
assert(data)

#-------------
conv1_w = weights['conv1.weight'].cpu().numpy().reshape(-1)
conv1_b = weights['conv1.bias'].cpu().numpy().reshape(-1)
conv1 = network.add_convolution(data, 20, (5,5), conv1_w, conv1_b)
assert(conv1)
conv1.set_stride((1,1))


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

```
assert(pool1)
pool1.set_stride((2,2))

#-------------
conv2_w = weights['conv2.weight'].cpu().numpy().reshape(-1)
conv2_b = weights['conv2.bias'].cpu().numpy().reshape(-1)
conv2 = network.add_convolution(pool1.get_output(0), 50, (5,5), conv2_w,
 conv2_b)
assert(conv2)
conv2.set_stride((1,1))

#-------------
pool2 = network.add_pooling(conv2.get_output(0), trt.infer.PoolingType.MAX,
 (2,2))
assert(pool2)
pool2.set_stride((2,2))

#-------------
fc1_w = weights['fc1.weight'].cpu().numpy().reshape(-1)
fc1_b = weights['fc1.bias'].cpu().numpy().reshape(-1)
fc1 = network.add_fully_connected(pool2.get_output(0), 500, fc1_w, fc1_b)
assert(fc1)

#-------------
relu1 = network.add_activation(fc1.get_output(0), trt.infer.ActivationType.RELU)
assert(relu1)

#-------------
fc2_w = weights['fc2.weight'].cpu().numpy().reshape(-1)
fc2_b = weights['fc2.bias'].cpu().numpy().reshape(-1)
fc2 = network.add_fully_connected(relu1.get_output(0), 10, fc2_w, fc2_b)
assert(fc2)
```

Mark you output layer.

```
fc2.get_output(0).set_name("prob")
network.mark_output(fc2.get_output(0))
```

Set the rest of the parameters for the network (max batch size and max workspace) and build the engine.

```
builder.set_max_batch_size(1)
builder.set_max_workspace_size(1 <<  20)

engine = builder.build_cuda_engine(network)
network.destroy()
builder.destroy()
```

Create the engine runtime and generate a test case from the torch data loader.

```
runtime = trt.infer.create_infer_runtime(G_LOGGER)
img, target = next(iter(test_loader))
img = img.numpy()[0]
target = target.numpy()[0]
print("Test Case: " + str(target))
img = img.ravel()
```

Create an execution context for the engine.

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

Allocate the memory on the GPU and allocate memory on the CPU to hold results after inference. The size of the allocations is the size of the input and expected output * the batch size.

```
output = np.empty(10, dtype = np.float32)
#allocate device memory
d_input = cuda.mem_alloc(1 * img.size * img.dtype.itemsize)
d_output = cuda.mem_alloc(1 * output.size * output.dtype.itemsize)
```

The engine needs bindings provided as pointers to the GPU memory. PyCUDA lets us do this for memory allocations by casting those allocations to **ints**.

```
bindings = [int(d_input), int(d_output)]
```

Create a CUDA stream to run inference in.

```
stream = cuda.Stream()
```

Transfer the data to the GPU, run inference, and then copy the results back.

```
#transfer input data to device
cuda.memcpy_htod_async(d_input, img, stream)
#execute model
context.enqueue(1, bindings, stream.handle, None)
#transfer predictions back
cuda.memcpy_dtoh_async(output, d_output, stream)
#synchronize threads
stream.synchronize()
```

Now that you have the results, run ArgMax to get a prediction.

```
print("Test Case: " + str(target))
print ("Prediction: " + str(np.argmax(output)))
```

We can also save our engine to a file to use later.

```
trt.utils.write_engine_to_file("/data/mnist/pyt_mnist.engine",
 engine.serialize())
```

You can then load this engine later by using **tensorrt.utils.load_engine**.

```
new_engine = trt.utils.load_engine(G_LOGGER,
"/data/mnist/new_mnist.engine")
```

Finally, clean up your context, engine, and runtime.

```
context.destroy()
engine.destroy()
new_engine.destroy()
runtime.destroy()
```

2.4. Build Phase

In the *build phase*, TensorRT takes a network definition, performs optimizations, and generates the inference engine.

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 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

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.

2.5. Execution Phase

In the *execution phase*, the following tasks are run:

- The runtime executes the optimized engine.
- The engine runs inference tasks using input and output buffers on the GPU.

2.6. Command Line Wrapper

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

Optional params:
  --model=<file>       Caffe model file (default = no model,
 random weights
       used)
  --batch=N            Set batch size (default = 1)
  --device=N           Set cuda device to N (default = 0)
  --iterations=N       Run N iterations (default = 10)
  --avgRuns=N          Set avgRuns to N - perf is measured as an
 average of
       avgRuns (default=10)
  --workspace=N        Set workspace size in megabytes (default =
 16)
  --half2              Run in paired fp16 mode (default = false)
  --int8               Run in int8 mode (default = false)
```

```
  --verbose             Use verbose logging (default = false)
  --hostTime            Measure host time rather than GPU time
 (default =
        false)
  --engine=<file>       Generate a serialized GIE engine
  --calib=<file>        Read INT8 calibration cache file
```

For example:

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

If no model is supplied, random weights are generated.

2.7. TensorRT Lite

Included within the Python API is a highly abstracted interface called TensorRT Lite. TensorRT Lite API handles almost everything when it comes to building an engine and executing inference, therefore, users are able to just create an engine and start processing data.

The TensorRT Lite API is located in **`tensorrt.lite`** and contains a single class called **`Engine`**. The engine constructor takes the model definition and the input and output layers and constructs a full engine around it that is ready for inference.

Internally, the Lite engine creates the logger, TensorRT engine, runtime and context for you, then allocates the GPU memory for the engine as well.

Features like custom loggers, plugins, calibrators, and profilers can be passed from the constructor.

2.7.1. Creating A TensorRT Lite Engine

2.7.1.1. Creating A TensorRT Lite Engine From A TensorFlow Model

Unlike the TensorRT utility functions, the Lite Engine supports TensorFlow models directly, performing the conversion to UFF internally. The model can be provided as a serialized graph or a path to a protobuf file by using the appropriate keyword argument. For example:

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

2.7.1.2. Creating A TensorRT Lite Engine From A UFF Model

Similar to creating a lite engine from a TensorFlow model, the Lite Engine API can accept a UFF model stream or a path to a UFF file. For example:

```
stream = uff.from_tensorflow_frozen_model("resnet50-infer-5.pb" ,["GPU_0/
tower_0/Softmax"])
    resnet = tensorrt.lite.Engine(framework="uff",
 stream=stream,input_nodes={"in":(3,224,224)},output_nodes=["GPU_0/tower_0/
Softmax"])
```

2.7.1.3. Creating A TensorRT Lite Engine From An NVCaffe Model

For a NVCaffe model you must provide a path to the deploy and model files. For example:

```
c = tensorrt.lite.Engine(framework="caffe",
                                      deployfile= "/mnist/mnist.prototxt",
                                      modelfile= "/mnist/mnist.caffemodel",
                                      max_batch_size=10,
                                      input_nodes={"data":(1,28,28)},
                                      output_nodes=["prob"])
```

2.7.1.4. Creating A TensorRT Lite Engine From A PLAN File

Ensure you have a TensorRT engine and a PLAN file for it. After you have both, you can then import it to use the Lite API. For example:

```
engine = tensorrt.lite.Engine(PLAN="model.plan")
```

2.7.1.5. Creating A TensorRT Lite Engine From A Serialized Engine

If you have a engine in memory or you are manually converting a model using the Network Definition API, you can import that engine into the Lite API.

```
import tensorrt as trt
import torch
import numpy as np

GLOGGER = trt.infer.ColorLogger(trt.infer.LogSeverity.ERROR)

#Create an Engine from a PyTorch model
def create_pytorch_engine(max_batch_size, builder, dt, model):
    network = builder.create_network()

    data = network.add_input("in", dt, (1, 28, 28))
    assert(data)

    #-------------
    conv1_w = model['conv1.weight'].cpu().numpy().reshape(-1)
    conv1_b = model['conv1.bias'].cpu().numpy().reshape(-1)
    conv1 = network.add_convolution(data, 20, (5,5),  conv1_w, conv1_b)
    assert(conv1)
    conv1.set_stride((1,1))

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

    #-------------
    conv2_w = model['conv2.weight'].cpu().numpy().reshape(-1)
    conv2_b = model['conv2.bias'].cpu().numpy().reshape(-1)
    conv2 = network.add_convolution(pool1.get_output(0), 50, (5,5), conv2_w,
 conv2_b)
```

```
    assert(conv2)
    conv2.set_stride((1,1))

    #-------------
    pool2 = network.add_pooling(conv2.get_output(0), trt.infer.PoolingType.MAX,
 (2,2))
    assert(pool2)
    pool2.set_stride((2,2))

    #-------------
    fc1_w = model['fc1.weight'].cpu().numpy().reshape(-1)
    fc1_b = model['fc1.bias'].cpu().numpy().reshape(-1)
    fc1 = network.add_fully_connected(pool2.get_output(0), 500, fc1_w, fc1_b)
    assert(fc1)

    #-------------
    relu1 = network.add_activation(fc1.get_output(0),
 trt.infer.ActivationType.RELU)
    assert(relu1)

    #-------------
    fc2_w = model['fc2.weight'].cpu().numpy().reshape(-1)
    fc2_b = model['fc2.bias'].cpu().numpy().reshape(-1)
    fc2 = network.add_fully_connected(relu1.get_output(0), 10, fc2_w, fc2_b)
    assert(fc2)

    #-------------
    # NOTE: Before release
    # Using log_softmax in training, cutting out log softmax here since no log
 softmax in TRT
    fc2.get_output(0).set_name("out")
    network.mark_output(fc2.get_output(0))


    builder.set_max_batch_size(max_batch_size)
    builder.set_max_workspace_size(1 << 20)

    engine = builder.build_cuda_engine(network)
    network.destroy()

    return engine

def main():
    #Load pretrained PyTorch model
    model = torch.load(DATA_DIR + "/mnist/trained_lenet5_mnist.pyt")

    #Build an engine from PyTorch
    builder = trt.infer.create_infer_builder(GLOGGER)
    engine = create_pytorch_engine(10, builder, trt.infer.DataType.FLOAT, model)

    #Create a Lite Engine from the engine
    mnist_engine = trt.lite.Engine(engine_stream=engine.serialize(),    #Use a
 serialized engine
                                                        max_batch_size=10)
                #Max batch size
    #Destroy the old engine
    engine.destroy()
    ...
```

2.7.2. Running Inference

After your engine is created, you can now use the **infer** call to run inference on a set of data. Your data can be provided to the Lite engine in a couple of ways. Fundamentally,

each input must be a NumPy array matching the input shape defined in the constructor. For example, if the input shape for a layer is **1,28,28**, then each input must be in the shape **1,28,28**. From the base structure, the input data can be formatted as:

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

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

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

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

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

For more information, see the examples in Creating A TensorRT Lite Engine.

2.7.3. Preprocessing And Postprocessing Function Tables

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

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

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

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

#Lamba to apply argmax to each result after inference to get prediction
```

```
#Instead of having to reshape, you can replace the 3D array provided to the
 postprocessor with #the object of your choosing (e.g. the top class)
argmax = lambda res: np.argmax(res.reshape(10))

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


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

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

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

2.7.4. Saving The Engine

Save the TensorRT engine within your Lite engine with **`engine.save(path)`**. You can later rebuild your engine by passing in the path to the PLAN file and the pre and post processors if applicable.

Chapter 3. SAMPLES

The following samples show how to use TensorRT in numerous use cases. The samples demonstrate the different capabilities of the interface. Each sample includes the build phase and an execution phase. The samples begin with a simple and basic usage sample and continues to the more complex samples.

The samples are listed sequentially, therefore, you need to implement each sample in the order provided.

3.1. Getting Started With The C++ Samples

Each C++ sample contains a section on:

- How to define the network
- How to build the network
- How to run the network
- How to verify that the output is correct

3.1.1. Initializing The TensorRT Library

There are two ways to initialize the TensorRT library:

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

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

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

```
            std::cout << msg << std::endl;
    }
} gLogger;
```

There are two ways to initialize the library:

1. You want to create a builder object by issuing:

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

   For more information about creating a build object, see Building the Engine.
2. You want to create a runtime object by issuing:

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

   For more information about creating a runtime object, see Running the Engine.

3.1.2. Defining A Network

After a builder object has been created, a network definition object can be created as follows:

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

Populate this object by:

- Defining the set of input tensors to the network
- Creating parameterized layers
- Marking the network outputs tensors

You can perform these operations individually via the API, or, more commonly, use one of the supported parsers to import your network from a training framework.

3.1.3. Building The Engine

After the network has been defined, build the engine by configuring the builder and calling:

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

The builder configuration includes the precision at which the network should run and auto-tuning parameters, for example, how many times TensorRT should time each kernel when ascertaining which is fastest. This increases the number of timing iterations ameliorates which affects the system noise upon measurement but increases the build time.

You can also query the builder to find out what reduced precision types are natively supported by the hardware.

3.1.4. Running The Engine

After the engine has been created, you can use it to perform inference. To use the engine, create an execution context:

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

The engine holds constant values (such as weights), while the context holds per-invocation values (such as temporary activations, and workspace required by the per-layer execution algorithms.) For example, if you want to process images in parallel in multiple CUDA streams, you should create a single engine, and then one context per stream.

It is also necessary to bind the network input and output tensors to memory buffers on the GPU. The buffers are passed in an array, and to find the array index for a buffer you can query the engine using the tensor name assigned when the network was created.

```
int inputIndex = engine->getBindingIndex(INPUT_BLOB_NAME),
     outputIndex = engine->getBindingIndex(OUTPUT_BLOB_NAME);
```

The engine also has methods to iterate over the bindings, such as identifying the name of each binding, whether it is an input or output binding, etc.

After the buffer array has been created, you can run inference. There are two invocation methods, synchronous:

```
context.execute(batchSize, buffers);
```

and asynchronous:

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

The batch size must be no greater than the batch size specified for the builder.

In the case of asynchronous execution, synchronize on the CUDA stream to await completion:

```
cudaStreamSynchronize(stream);
```

The last argument to **`enqueue()`** is a CUDA event which is signaled when the input buffers may be reused, to facilitate pipelining.

3.1.5. Serializing And Deserializing The Engine

In a typical use case, you will run the builder offline, and the engine will be serialized to a file:

```
gieModelStream = engine->serialize();
```

Next, at the application initialization time, create a TensorRT runtime and deserialize the engine:

```
IRuntime* runtime = createInferRuntime(gLogger);
 ICudaEngine* engine = runtime->deserializeCudaEngine(gieModelStream->data(),
 gieModelStream->size(), nullptr);
```

Serialized engines are not portable across platforms or TensorRT versions.

Next, you need to deserialize the plan to create the engine. To deserialize the engine, create a TensorRT runtime object:

```
IRuntime* runtime = createInferRuntime(gLogger);
ICudaEngine* engine = runtime->deserializeCudaEngine(gieModelStream->data(),
 gieModelStream->size(), nullptr);
```

Create an execution context. One engine can support multiple contexts, allowing inference to be performed on multiple batches simultaneously while sharing the same weights.

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

Assemble the parameters to input into the engine. The input to the engine is an array of pointers to input and output buffers on the GPU.

All TensorRT inputs and outputs are in contiguous NCHW format.

3.2. SampleMNIST Simple Usage

The SampleMNIST sample implements an NVCaffe model trained on the MNIST dataset, where the dataset is from the DIGITS tutorial. This section explains how to build and execute the sample.

This sample covers the basic setting-up of TensorRT.

3.2.1. Key Concepts

The key concepts explained in this sample are:

- How to initialize the library
- How to export the engine to a PLAN file, and reimport it at runtime
- How to use the imported engine to perform inference

Ensure you are familiar with the following key concepts for the MNIST sample:

Initializing the library
When initializing TensorRT you must implement a logging interface through which TensorRT reports errors, warnings, and informational messages.

Serializing the engine
Serialization is the process of storing the engine object into a PLAN file.

Deserializing the engine to PLAN files
In order to create the engine, you must first deserialize the PLAN.

3.2.2. Configuring The Builder

Build the engine from the network definition:

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

where:

- **`maxBatchSize`** is the size for which the engine will be tuned. At execution time, smaller batches may be used, but not larger.

  

  The execution of smaller batch sizes may be slower than with a TensorRT engine optimized for that size.

- **`maxWorkspaceSize`** is the maximum amount of scratch space which the engine may use at runtime.

With the following code, the engine is serialized to a memory block, which you could then serialize to a file or stream:

```
gieModelStream = engine->serialize();
```

3.2.3. Verifying The Output

After you run the engine, it is a good idea to verify that the engine is running properly and you see the output you would expect. For example, in this sample, SampleMNIST picks a random digit from 10 images and the results look similar to the following.

The output is an ASCII art rendering of the image followed by a histogram of the probability distribution of what the network thinks the image is. In this example, the network thinks the image is an 8.

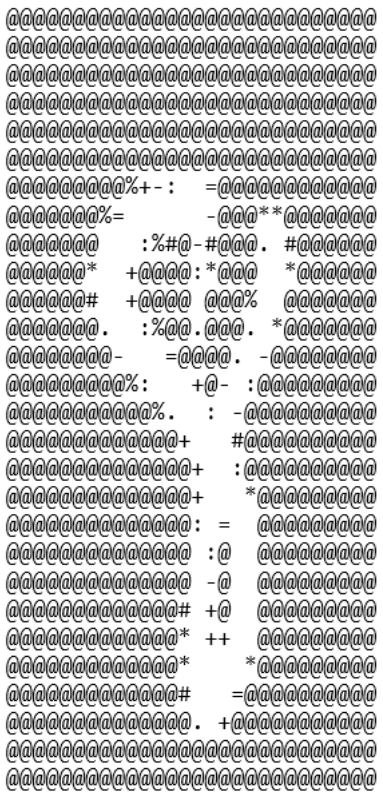

Figure 3 ASCII output

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

Figure 4 Decision output

3.3. SampleUffMNIST UFF Usage

The SampleUffMNIST sample implements a TensorFlow model trained on the MNIST dataset. The TensorFlow model has been converted to UFF using the explanation described in section Exporting TensorFlow To A UFF File.

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 3.0, 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.

3.3.1. Key Concepts

The key concepts explained in this sample are:

- How to use the UFF Parser

Ensure you are familiar with the following key concepts for the SampleUffMNIST sample:

Registering input layers
You need to register the inputs layers with the dimensions in NCHW (even if the exported network was in NHWC).

Registering output layers
You need to register the output layer with their name.

UFF converter
A UFF converter is a tool that uses the UFF Toolkit. The converter converts your neural network from a framework, such as TensorFlow, to a UFF file. Currently, only TensorFlow is automatically supported.

3.3.2. Defining The Network Using NvUffParser

To define a network, you must import the network definition (including the weights) from a UFF converter.

In order to create the network definition structure, you need to populate from a UFF model using the UFF parser library:

```
INetworkDefinition* network = builder->createNetwork();
auto parser = createUffParser();
```

You also need to register the input dimensions with NCHW format (even if the exported network was in NHWC):

```
parser->registerInput("Input_0", DimsCHW(1, 28, 28));
```

If the UFF converter did not add an output node with the MarkOutput special operation of UFF, you need to register the output as well:

```
parser->registerOutput("Binary_3");
```

The `uffFile` argument is a C string of the filename. The `parse` method returns a boolean that represents whether the parse succeeded or not. You can create an input network with either 32-bit weight or 16-bit weights by supplying DataType::kFLOAT and DataType::kHALF parameters respectively to the parser. For example:

```
if (!parser->parse(uffFile, *network, nvinfer1::DataType::kFLOAT))
        RETURN_AND_LOG(nullptr, ERROR, "Fail to parse");
```

The **`registerInput`** method of the parser accepts an optional argument called **`inputOrder`** which allows the user to define the ordering of the input. This feature

is not yet implemented. For now, provide the input in NCHW format even if your network used to be in NHWC.

3.4. SampleMNISTAPI C++ API Usage

The SampleMNISTAPI example demonstrates how to use the C++ API in order to produce the same network as SampleMNIST but without using NvCaffeParser.

3.4.1. Key Concepts

The key concepts explained in this sample are:

- How to specify networks using the TensorRT API rather than via a parser.

3.4.2. Defining The Network Using The C++ API

When using the C++ API, in order to define a network you must:

1. Creating The Network
2. Adding Input Into Your Network
3. Creating A Layer
4. Setting The Network Output

3.4.2.1. Creating The Network

To create the network, issue the following command:

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

3.4.2.2. Adding Input Into Your Network

All networks must specify an input; as the input is the entry point to the network. You must provide a name for the input.

```
INetworkDefinition* network = builder->createNetwork();
//  Create input of shape { 1, 1, 28, 28 } with name referenced
 by INPUT_BLOB_NAME
auto data = network->addInput(INPUT_BLOB_NAME, dt, DimsCHW{ 1,
 INPUT_H, INPUT_W});
```

3.4.2.3. Creating A Layer

You can create multiple layers directly from the TensorRT C++ API.

In the following code, both **power** and **shift** are using the default values for their weights and the scale parameter is being provided to the layer. The scaling mode is uniform scaling.

The following code is an example of the creation of a single scale layer:

```
// Create a scale layer with default power/shift and specified
 scale
parameter.
float scale_param = 0.0125f;
Weights power{DataType::kFLOAT, nullptr, 0};
Weights shift{DataType::kFLOAT, nullptr, 0};
Weights scale{DataType::kFLOAT, &scale_param, 1};
auto scale_1 = network->addScale(*data, ScaleMode::kUNIFORM,
 shift, scale, power);
```

For layers that require weights, you need to load the weights. In this sample, since NVCaffe is not generating the weights automatically, you must allocate and manage the weight memory, which is stored in the **weightMap** and read from the filesystem.

```
std::map<std::string, Weights> weightMap =
      loadWeights(locateFile("mnistapi.wts"));
```

3.4.2.4. Setting The Network Output

The network must know which layers set which outputs.

It is recommended to name the outputs.

If a name is not provided, TensorRT will generate a name.

```
// Add a softmax layer to determine the probability.
auto prob = network->addSoftMax(*ip2->getOutput(0));
prob->getOutput(0)->setName(OUTPUT_BLOB_NAME);
network->markOutput(*prob->getOutput(0));
```

3.4.3. Memory Management Requirements

Memory needs to be made available to the builder until after the engine is created. In this sample, the memory for weights are stored in a map after being loaded from the filesystem. After the engine has been created and the network has been destroyed, it is safe to deallocate memory.

Deallocating memory before creating the engine has undefined behavior.

3.5. SampleGoogleNet - Profiling And 16-bit Inference

The SampleGoogleNet example demonstrates the layer-based profiling, and TensorRT half2 mode, which runs the network in 16-bit floating point precision.

3.5.1. Key Concepts

The key concepts explained this sample are:

- How to use FP16 mode in TensorRT
- How to execute a profile at the per-layer level

Ensure you are familiar with the following key concepts:

Profiling a network
Profiling enables you to know which assets are running on a particular network.

FP16
Half-precision (also known as FP16) data compared to higher precision FP32 or FP64 reduces memory usage of the neural network, allowing training and deployment of larger networks, and FP16 data transfers take less time than FP32 or FP64 transfers.

`Half2Mode`
`Half2Mode` is an execution mode where internal tensors interleave 16-bits from adjacent pairs of images, and is the fastest mode of operation for batch sizes greater than one.

3.5.2. Configuring The Builder

TensorRT can use 16-bit instead of 32-bit arithmetic and tensors, but this alone may not deliver significant performance benefits. **`Half2Mode`** is an execution mode where internal tensors interleave 16-bits from adjacent pairs of images, and is the fastest mode of operation for batch sizes greater than one.

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. For example:

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

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

```
builder->setHalf2Mode(true);
```

3.5.3. 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.

3.6. SampleCharRNN - RNNs And Converting Weights From TensorFlow To TensorRT

The SampleCharRNN example demonstrates how to generate a simple RNN based on the charRNN network using the PTB dataset.

RNN layers are like any other TensorRT layer. Each RNN has three output tensors and three input tensors; with 2 tensors that are option in each case. For more information, see the TensorRT API.

3.6.1. Key Concepts

Ensure you are familiar with the following key concepts:

TensorRT RNN layer

RNN layers are like any other TensorRT layer. Each RNN has three output tensors and three input tensors; with two tensors that are options in each case. For more information, see the TensorRT API.

3.6.2. Defining The Network Using The C++ API

3.6.2.1. Weight Conversion

TensorFlow weights are exported with each layer concatenated into a single WTS file. The file format is defined by the **`loadWeights`** function. The weights that were previously loaded by **`loadWeights()`** are now converted into an the format required by TensorRT. The memory holding the converted weights is added to the weight map so that it can be deallocated once the engine has been built.

```
// Create an RNN layer w/ 2 layers and 512 hidden states
auto tfwts = weightMap["rnnweight"];
Weights rnnwts{convertRNNWeights(tfwts)};
auto tfbias = weightMap["rnnbias"];
Weights rnnbias{convertRNNBias(tfbias)};
...
weightMap["rnnweight2"] = rnnwts;
weightMap["rnnbias2"] = rnnbias;
```

3.6.2.2. Layer Generation

After the RNN weights are converted, the next step is to create the RNN layer. There are multiple different RNN types and modes that are supported. This specific RNN is a single directional LSTM layer where the input is transformed to match the same size as the hidden weight matrix.

```
auto rnn = network->addRNN(*data, LAYER_COUNT, HIDDEN_SIZE,
 SEQ_SIZE,
```

```
        RNNOperation::kLSTM, RNNInputMode::kLINEAR,
 RNNDirection::kUNIDIRECTION,
        rnnwts, rnnbias);
```

3.6.2.3. Optional Inputs

If there are cases where the hidden and cell states need to be pre-initialized, 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);
```

3.6.2.4. Marking The Resulting Output

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.

```
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));
}
```

3.6.2.5. Reshaping Data To Fit The Format Of The Next Layer

The output of an RNN is optimized to feed into another RNN layer as efficiently as possible. When out-puting to another layer that has a different layer requirement, a reshaping is required.

The reshape parameter uses the plugin API and converts the layer to the format required for the FullyConnected layer. In this case we are reshaping the **`{ T, N, C }`** to **`{N * T, C, 1, 1}`** so that it can be fed properly into the FullyConnected layer.

```
Reshape reshape(SEQ_SIZE * BATCH_SIZE * HIDDEN_SIZE);
ITensor *ptr = rnn->getOutput(0);
auto plugin = network->addPlugin(&ptr, 1, reshape);
plugin->setName("reshape");

auto fc = network->addFullyConnected(*plugin->getOutput(0),
OUTPUT_SIZE, wts, bias);
```

TensorRT network inputs and outputs are 32-bit tensors in contiguous **`NCHW`** format. For weights:

- Convolution weights are in contiguous **`KCRS`** format, where **`K`** indexes over output channels, **`C`** over input channels, and **`R`** and **`S`** over the height and width of the convolution, respectively.
- FullyConnected weights are in contiguous row-major layout.
- Deconvolution weights are in contiguous **`CKRS`** format; where **`C`**, **`K`**, **`R`** and **`S`** are the same as convolution weights.

3.6.3. 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(reinterpret_cast<const float*>(embed.values) +
char_to_id[a]*DATA_SIZE,
               reinterpret_cast<const float*>(embed.values) +
char_to_id[a]*DATA_SIZE + DATA_SIZE,
               data[0]);
     stepOnce(data, buffers, sizes, indices, 6, stream, context);
     cudaStreamSynchronize(stream);
     genstr.push_back(a); }
```

3.6.4. 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 highest probability. The final result is stored in **`genstr`**.

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

     stepOnce(data, buffers, sizes, indices, 6, stream, context);
     cudaStreamSynchronize(stream);

     float* probabilities =
 reinterpret_cast<float*>(data[indices[3]]);
     int idx = std::max_element(probabilities, probabilities +
 sizes[3]) -
probabilities;
     genstr.push_back(id_to_char[idx]);
}
```

3.7. SampleINT8 - Calibration And 8-bit Inference

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.

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 calibrate 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
```

3.7.1. Key Concepts

The key concepts explained this sample are:

- How to calibrate a network for execution in INT8
- How to cache the output of the calibration to avoid repeating the process
- How to repro your own experiments with NVCaffe in order to validate your results on ImageNet networks

3.7.2. 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);
```

3.7.3. 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 reduce precision.

3.7.3.1. 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.

3.7.3.2. Calibration Set

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

```
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. In addition, any image processing, such as, scaling, cropping or mean subtraction, that would occur prior to inference must also be performed prior to calibration.

3.7.4. Configuring the Builder

There are two additional methods to call on the builder:

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

3.7.4.1. Calibration Caching

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:

- then 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.

3.7.5. 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.

3.7.6. 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%, usually much closer.

3.7.7. Batch Files For Calibration

The SampleINT8 example uses batch files in order to calibrate for the INT8 data. The INT8 batch file is a binary file defined as follows:

- Four 32-bit integer values representing **`{N,C, H, W}`** dimensions of the data set.
- There are **`N`** 32-bit floating point data blobs of dimensions **`{C, H, W}`** that are used as inputs to the network.
- There are **`N`** 32-bit integer labels that correspond to the **`N`** input blobs.

3.7.7.1. Generating Batch Files For NVCaffe Users

For developers that use NVCaffe for their training, or can easily transfer their network to NVCaffe, generating the calibration data is done through a supplied patchset.

These instructions are for Caffe™ git commit **`473f143f9422e7fc66e9590da6b2a1bb88e50b2f`**. The patchfile might be slightly different for later versions of NVCaffe. The patch can be applied by going to the root directory of the NVCaffe source tree and applying the patch with the command:

```
patch -p1 < int8_caffe.patch
```

After the patch is applied, NVCaffe needs to be rebuilt and the environment variable *TENSORRT_INT8_BATCH_DIRECTORY* needs to be set 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 NVCaffe pre-processes 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 NVCaffe 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 NVCaffe generated batch files.

First, go to the samples data directory and create an INT8 **mnist** directory.

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

If NVCaffe is not installed anywhere, ensure you clone, checkout, patch, and build the NVCaffe at the specified 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
```

After the build has finished, download the **mnist** data set from NVCaffe and create the link to it.

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

Set the directory to store the batch data, execute NVCaffe, 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 .
```

SampleINT8 can now be executed from the **bin** directory after being built with the command **./sample_int8 mnist**.

3.7.7.2. Generating Batch Files For Non-NVCaffe Users

For developers that are not using NVCaffe, or cannot easily convert to NVCaffe, 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 data set.

2. Crop all of the input data to the same dimensions.
3. Split the data into **N** batch files where each batch file has **M** sets of input data and **M** sets of 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 NVCaffe.

First, go 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 .
```

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

SampleINT8 can now be executed from the **bin** directory after being built with the command **./sample_int8 mnist**.

3.8. SamplePlugin - Implementing A Custom Layer

The SamplePlugin example demonstrates how to add a custom layer to TensorRT. It replaces the final fully connected layer of the MNIST sample with a direct call to CUDA® Basic Linear Algebra Subroutines library™ (cuBLAS).

Attention Currently only FP32 precision is supported by the plugin layer.

3.8.1. Key Concepts

The key concepts explained in this sample are:

- How to create a custom layer
- How to integrate the custom layer with NvCaffeParser and import it into the runtime

3.8.2. Implementing The Plugin Interface

Each phase of TensorRT operation (network creation, build, serialization and runtime) requires implementing methods in the plugin API. The following is an overview of which methods are relevant to each phase.

When creating the network:

These methods are called during network construction if the output size of the layer, or any subsequent layer, is requested through an `ITensor::getDimensions()` call. Otherwise, the methods are called when the builder runs.

- `getNbOutputs()`
- `getOutputDimensions()`

By the builder:

- `configure()`
- `getWorkspaceSize()`

At runtime:

- `initialize()` when an engine context is constructed
- `enqueue()` at inference time
- `terminate()` when an engine context is destroyed

For serialization:

- `getSerializationSize()`
- `serialize()`

3.8.3. Defining The Network

When defining the network, TensorRT needs to know which outputs the layer has.

The dimensions given in the sample are without the batch size, in a similar way to dimensions returned by `ITensor::getDimensions()`. For example, for a typical 3-dimensional convolution, the dimensions provided are given in **`{C, H, W}`** form, and the return value should also be in **`{C, H, W}`** form.

The following methods provide which outputs the layer has:

```
int getNbOutputs() const override
{
     return 1;
}
```

The **`getOutputDimensions`** function has three parameters, the output index, input dimensions and number of inputs. The last two parameters are already calculated by TensorRT internally, so all you need is to calculate the output dimensions based on the given input dimensions and the given index number.

```
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);
}
```

3.8.4. Layer Configuration

The builder calls the network's `configure()` method, to give it a chance to select an algorithm based on its inputs. In this example, the inputs are checked to have the correct form. In a more complex example, you might choose a convolution algorithm based on the input dimensions.

The `configure()` method is only called at build time, therefore, anything determined here that is required at runtime should be stored as a member variable of the plugin, and serialized and/or de-serialized.

3.8.5. Workspace

TensorRT can provide workspace for temporary storage during layer execution, which is shared among layers in order to minimize memory usage. The TensorRT builder calls `getWorkspaceSize()` in order to determine the workspace requirement. In this example, no workspace is used. If workspace is requested, it will be allocated when an `IExecutionContext` is created, and passed to the `enqueue()` method at runtime.

Alternatively, you can also allocate the GPU memory for layer execution in the constructor and release it in the destructor, and just return zero in this callback. The advantage of using workspace allocated by TensorRT is that it can be shared by other plugins during execution.

3.8.6. Resource Management

The `initialize()` and `terminate()` methods are called by the builder when performing auto-tuning, and by the runtime when an **`IExecutionContext`** is created and destroyed. They are used to allocate and release resource that is needed by layer execution - in this example, handles for cuDNN and cuBLAS, and some cuDNN tensor descriptors for the bias addition operation.

In the following sample, handles are created for cuDNN, cuBLAS, and some cuDNN tensor descriptors for the bias addition operation.

```
int initialize() override
{
     CHECK(cudnnCreate(&mCudnn));
     CHECK(cublasCreate(&mCublas));
     CHECK(cudnnCreateTensorDescriptor(&mSrcDescriptor));
     CHECK(cudnnCreateTensorDescriptor(&mDstDescriptor));

     return 0;
}

virtual void terminate() override
{
     CHECK(cublasDestroy(mCublas));
     CHECK(cudnnDestroy(mCudnn));
}
```

3.8.7. Runtime Implementation

The `enqueue()` method is used to execute the layer's runtime implementation.

The batch size passed to `enqueue()` is at most the maximum batch size specified at build time, although it can be smaller.

Except for batch size, dimensional information is not passed to `enqueue()`. Therefore, other dimensional information required at runtime, for example, the number of input and output channels, should be serialized as part of the layer data.

```
virtual int enqueue(int batchSize, const void*const * inputs,
 void**
outputs, void* workspace, cudaStream_t stream) override
{
     int nbOutputChannels = mBiasWeights.count;
     int nbInputChannels = mKernelWeights.count /
 nbOutputChannels;
     float kONE = 1.0f, kZERO = 0.0f;
     cublasSetStream(mCublas, stream);
     cudnnSetStream(mCudnn, stream);
     CHECK(cublasSgemm(mCublas, CUBLAS_OP_T, CUBLAS_OP_N,
nbOutputChannels, batchSize, nbInputChannels, &kONE,
                reinterpret_cast<const
 float*>(mKernelWeights.values),
nbInputChannels,
                reinterpret_cast<const float*>(inputs[0]),
nbInputChannels, &kZERO,
                reinterpret_cast<float*>(outputs[0]),
 nbOutputChannels));
     CHECK(cudnnSetTensor4dDescriptor(mSrcDescriptor,
CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, 1, nbOutputChannels, 1, 1));
     CHECK(cudnnSetTensor4dDescriptor(mDstDescriptor,
CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, batchSize, nbOutputChannels,
 1, 1));
     CHECK(cudnnAddTensor(mCudnn, &kONE, mSrcDescriptor,
mBiasWeights.values, &kONE, mDstDescriptor, outputs[0]));
     return 0;
}
```

3.8.8. Serialization

Layer parameters can be serialized along with the rest of the network. The serialization system calls the following functions:

```
virtual size_t getSerializationSize() override
{
    // 3 integers (number of input channels, number of output
channels, bias size), and then the weights:
    return sizeof(int)*3 + mKernelWeights.count*sizeof(float) +
mBiasWeights.count*sizeof(float);
}

virtual void serialize(void* buffer) override
```

```
{
    char* d = reinterpret_cast<char*>(buffer), *a = d;

    write(d, mNbInputChannels);
    write(d, mNbOutputChannels);
    write(d, (int)mBiasWeights.count);
    serializeFromDevice(d, mKernelWeights);
    serializeFromDevice(d, mBiasWeights);

    assert(d == a + getSerializationSize());
}
```

Deserialization is implemented with the following constructor:

```
// create the plugin at runtime from a byte stream
FCPlugin(const void* data, size_t length)
{
    const char* d = reinterpret_cast<const char*>(data), *a = d;
    mNbInputChannels = read<int>(d);
    mNbOutputChannels = read<int>(d);
    int biasCount = read<int>(d);

    mKernelWeights = deserializeToDevice(d, mNbInputChannels *
mNbOutputChannels);
    mBiasWeights = deserializeToDevice(d, biasCount);
    assert(d == a + length);
}
```

3.8.9. Adding The Plugin Into A Network

There are three ways to add the plugin into a network:

1. Use the `INetwork::addPlugin()` method when defining the network.
2. Create the network via a parser.
3. De-serialize the network after it has been built.

For use of the `addPlugin()` method, see the TensorRT API.

3.8.9.1. Creating Plugins From NvCaffeParser

To add custom layers via NvCaffeParser, create a factory by implementing the nvcaffeParser::IPluginFactory interface, then pass an instance to `ICaffeParser::parse()`.

The `createPlugin()` method receives the layer name, and a set of weights extracted from the NVCaffe model file, which are then passed to the layer constructor. The name can be used to disambiguate between multiple plugins. There is currently no way to extract parameters other than weights from the NVCaffe network description, therefore, these parameters must be specified in the factory.

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

```
virtual nvinfer1::IPlugin* createPlugin(const char* layerName,
 const
nvinfer1::Weights* weights, int nbWeights) override
{
    // there's no way to pass parameters through from the model
definition, so we have to define it here explicitly
    static const int NB_OUTPUT_CHANNELS = 10;
    assert(isPlugin(layerName) && nbWeights == 2 &&
 weights[0].type ==
DataType::kFLOAT && weights[1].type == DataType::kFLOAT);
    assert(mPlugin.get() == nullptr);
    mPlugin = std::unique_ptr<FCPlugin>(new FCPlugin(weights,
nbWeights, NB_OUTPUT_CHANNELS));
    return mPlugin.get();
}
```

3.8.9.2. Creating Plugins At Runtime

To integrate custom layers with the runtime, implement the nvinfer1::IPlugin interface and pass an instance of the factory to `IInferRuntime::deserializeCudaEngine()`.

```
// deserialization plugin implementation
IPlugin* createPlugin(const char* layerName, const void*
 serialData,
size_t serialLength) override
{
    assert(isPlugin(layerName));
    assert(mPlugin.get() == nullptr);
    mPlugin = std::make_unique<FCPlugin>(serialData,
 serialLength);
    return mPlugin.get();
}
```

When constructed using the NvCaffeParser or deserialized at runtime, the layer implementation may assume that data passed as weights (from NvCaffeParser) or a byte stream (at runtime) will exist until the call to `initialize()`, allowing the data to be copied to the GPU in that function.

3.9. SampleFasterRCNN - Using The Plugin Library

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.

SampleFasterRCNN uses a plugin from the TensorRT plugin library to include a fused implementation of Faster R-CNN's RPN and ROIPooling layers. These particular layers are from the Faster R-CNN paper and are implemented together as a single plugin called the Faster R-CNN plugin.

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). Compared to other end-to-end techniques such as YOLO, YOLO makes a significant number of localization errors, which is not the case of Faster R-CNN. Furthermore, YOLO has relatively low recall compared to Faster R-CNN.

1. The Faster R-CNN NVCaffe model is too large to include in the product bundle. To run this sample, download the model using the instructions in the README in the sample directory. The README is located in the **`<TensorRT directory>/samples/sampleFasterRCNN`** directory.
2. The original NVCaffe model has been modified to include the Faster R-CNN's RPN and ROIPooling layers.

Because TensorRT does not currently support the Reshape layer, it uses plugins to implement reshaping. The Reshape plugin requires a copy operation because the current version of TensorRT does not support in-place plugin layers.

There is code within SampleFasterRCNN, along with factories, that show how you can create and deserialize multiple plugins for a network.

In this sample you will learn:

- How to implement the Faster R-CNN network in TensorRT
- How to perform a quick performance test in TensorRT
- How to implement a fused custom layer
- How to construct the basis for further optimization, for example using INT8 calibration, user trained network, etc.).

3.9.1. Key Concepts

The key concepts explained this sample are:

- How to implement the Faster R-CNN network in TensorRT
- How to perform a quick performance test in TensorRT
- How to implement a fused custom layer
- How to construct the basis for further optimization, for example using INT8 calibration, user trained network, etc.).

3.9.2. Pre-Processing 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 usually represented by a byte of integer (0-255) and they are stored together, pixel by pixel.

However, the authors of Faster R-CNN have trained the network such that the first convolution layer sees the image data in **B**, **G**, and **R** order. Therefore, you need to reverse the order when the PPM images are being put into the network input 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**.

This 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 pre-process the inputs, ensure that the TensorRT inference engine sees the input data in the form that it is supposed to.

3.9.3. 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 for two minor differences.

1. The RPN and the ROI pooling layer is fused and replaced by a custom layer named **RPROIFused**.
2. The reshape layer is replaced with two custom layers: **ReshapeCTo2** and **ReshapeCTo18** and are defined in the sample.

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 NVIDIA plugin library

createFasterRCNNPlugin. This function returns an instance that implements an optimized RPROIFused custom layer and performs the same logic designed by the authors.

3.9.4. Building The Engine

Build the engine from the network definition:

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

where:

- **maxBatchSize** is the size for which the engine will be tuned. At execution time, smaller batches may be used, but not larger.

The execution of smaller batch sizes may be slower than with a TensorRT engine optimized for that size.

- **maxWorkspaceSize** is the maximum amount of scratch space which the engine may use at runtime.

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.

Alternatively, you can run the engine without serializing it to a file or stream. However, a typical application will build an engine once, and then serialize it for later use. Because the building of an engine may take considerable time, and a built network can be saved by serialization and later reloaded into TensorRT runtime without having to perform the optimization step again. In the sampleFasterRCNN sample, we demonstrate how to serialize the engine, then run the inference with the deserialized engine.

With the following code, the engine is serialized to a memory block, which you could then serialize to a file or stream:

```
gieModelStream = engine->serialize();
```

3.9.5. Running The Engine

You need to deserialize the plan to create the engine. To deserialize the engine, create a TensorRT runtime object:

```
IRuntime* runtime = createInferRuntime(gLogger);
ICudaEngine* engine =
runtime->deserializeCudaEngine(gieModelStream->data(),
gieModelStream->size(), &pluginFactory);
```

Next, create an execution context. One engine can support multiple contexts, allowing inference to be performed on multiple batches simultaneously while sharing the same weights.

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

Serialized engines are not portable across platforms or TensorRT versions.

Assemble the parameters to input into the engine. The input to the engine is an array of pointers to input and output buffers on the GPU.

All TensorRT inputs and outputs are in contiguous NCHW format.

The engine can be queried for the buffer indices, using the tensor names assigned when the network was created.

```
int inputIndex0 = engine.getBindingIndex(INPUT_BLOB_NAME0),
    inputIndex1 = engine.getBindingIndex(INPUT_BLOB_NAME1),
    outputIndex0 = engine.getBindingIndex(OUTPUT_BLOB_NAME0),
    outputIndex1 = engine.getBindingIndex(OUTPUT_BLOB_NAME1),
    outputIndex2 = engine.getBindingIndex(OUTPUT_BLOB_NAME2),
    outputIndex3 = engine.getBindingIndex(OUTPUT_BLOB_NAME3);
```

In SampleFasterRCNN there are two inputs:

data
: The **data** input is the image input.

im_info
: The **im_info** input is the image information array which contains the number of rows, columns, and the scale for each image in a batch.

and four outputs:

bbox_pred
: The **bbox_pred** output is the predicted offsets to the heights, widths, and center coordinates.

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

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

count
: The **count** output 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.

The outputs are associated respectively with **INPUT_BLOB_NAME0**, **INPUT_BLOB_NAME1**, **OUTPUT_BLOB_NAME0**, **OUTPUT_BLOB_NAME1**, **OUTPUT_BLOB_NAME2**, **OUTPUT_BLOB_NAME3**.

In a typical production case, TensorRT will execute asynchronously. The `enqueue()` method will add kernels to a CUDA stream specified by the application, which may then wait on that stream for completion. The fourth parameter to `enqueue()` is an optional

cudaEvent which will be signaled when the input buffers are no longer in use and can be refilled.

The following sample code shows the input buffer being copied to the GPU, running inference, then copying the result back and waiting on the stream:

```
cudaMemcpyAsync(<...>, cudaMemcpyHostToDevice, stream);
context.enqueue(batchSize, buffers, stream, nullptr);
cudaMemcpyAsync(<...>, cudaMemcpyDeviceToHost, stream);
cudaStreamSynchronize(stream);
```

The batch size must be at most the value specified when the engine was created.

3.9.6. 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 4 coordinates. They are drawn in the output PPM images using the **`writePPMFileWithBBox`** function.

Chapter 4. PERFORMANCE

4.1. TensorRT Python Bindings

NVIDIA supplies Python bindings for TensorRT in order to put the capabilities of TensorRT in easy reach of data scientists working in Python. In order to interact with TensorRT buffers and streams, the Python bindings treat the buffers as a list of integers; and the streams as a single integer. CUDA operations are handled by external libraries, with the only requirement being that raw handles to device buffers and streams can be extracted.

Any time you build around TensorRT, the goal is to get to the inference execution step. In C++, you can create the buffers for data, output onto the GPU, and then transfer the input data to the device. You would then take the pointers for both buffers in the form of a pointer array, and if applicable a stream handle, and pass those to TensorRT to execute.

To create custom layers for use in Python, you can implement your custom layer in C ++ (so that use of libraries like CUDA and cuDNN are easier). Use the SWIG plugin in Python's setup tools to create a package around the C++ file that you created. Then, the plugin can be loaded into a Python application and run in C++. You can also use the same custom layer implementation for both C++ and Python.

Chapter 5. TROUBLESHOOTING

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

5.1. Creating An Engine That Is Optimized For Several Batch Sizes

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.

5.2. Choosing The Optimal Workspace Size

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.

5.3. Using TensorRT On Multiple GPUs

Each `ICudaEngine` object is bound to a specific GPU when it is instantiated, either by the builder or on de-serialization.

To select the GPU, use `cudaSetDevice()` before calling the builder or de-serializing 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.

Notice

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

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

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

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

Trademarks

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

Copyright