ACCELERATING INFRINGEMENT IN TENSORFLOW WITH TENSORRT (TF-TRT)

User Guide
# TABLE OF CONTENTS

Chapter 1. Integrating Overview..............................................................................1  
  1.1. Introduction................................................................................................1  
  1.2. Benefits Of Integrating TensorFlow With TensorRT.................................3  
  1.3. What Capabilities Does TF-TRT Provide?.................................................3  

Chapter 2. Prerequisites And Dependencies............................................................... 6  
  2.1. Support Matrix.............................................................................................6  

Chapter 3. Using TF-TRT....................................................................................... 8  
  3.1. Supported Ops.............................................................................................8  
  3.2. TF-TRT Workflow With A SavedModel.......................................................10  
  3.3. TF-TRT Workflow With A Frozen Graph.....................................................10  
  3.4. TF-TRT Workflow With MetaGraph And Checkpoint Files........................11  
  3.5. Running Inference With TF-TRT For Image Classification.........................12  
    3.5.1. Setting Up The Environment..................................................................12  
    3.5.2. Obtaining The ImageNet Data...............................................................13  
    3.5.3. General Script Usage...........................................................................13  
    3.5.4. Script Output.......................................................................................15  
    3.5.5. Visualizing TF-TRT Graphs With TensorBoard......................................16  
  3.6. Using TF-TRT With ResNet V1 50.................................................................19  
    3.6.1. Tutorial: Native TensorFlow Using FP32..............................................19  
    3.6.2. Tutorial: TF-TRT Using FP32...............................................................20  
    3.6.3. Tutorial: TF-TRT Using FP16...............................................................21  
    3.6.4. Tutorial: TF-TRT Using INT8...............................................................21  
  3.7. Verified Models......................................................................................... 22  
  3.8. INT8 Quantization..................................................................................... 23  
    3.8.1. Integrating Overview..........................................................................23  
    3.8.2. Quantization-Aware Training..............................................................24  
      3.8.2.1. Where To Add Quantization Nodes.................................................25  
  3.9. Using TF-TRT To Generate A Stand-Alone TensorRT Plan........................25  
  3.10. TensorFlow Ops Used In A TensorRT Op..................................................26  
  3.11. Unconverted TensorFlow Ops.................................................................... 26  
  3.12. Debugging Tools.................................................................................... 27  

Chapter 4. Samples..............................................................................................28  

Chapter 5. Best Practices.....................................................................................29  

Chapter 6. Performance.......................................................................................32  

Chapter 7. Troubleshooting..................................................................................33  
  7.1. Support....................................................................................................33
Chapter 1.
INTEGRATING OVERVIEW

TensorFlow™ integration with TensorRT™ (TF-TRT) optimizes and executes compatible subgraphs, allowing TensorFlow to execute the remaining graph. While you can still use TensorFlow’s wide and flexible feature set, TensorRT will parse the model and apply optimizations to the portions of the graph wherever possible.

You will need to create a SavedModel (or frozen graph) out of a trained TensorFlow model (see Build and load a SavedModel), and give that to the Python API of TF-TRT (see Using TF-TRT), which then:

- returns the TensorRT optimized SavedModel (or frozen graph).
- replaces each supported subgraph with a TensorRT optimized node (called `TRTEngineOp`), producing a new TensorFlow graph.

During the TF-TRT optimization, TensorRT performs several important transformations and optimizations to the neural network graph. First, layers with unused output are eliminated to avoid unnecessary computation. Next, where possible, convolution, bias, and ReLU layers are fused to form a single layer. Another transformation is horizontal layer fusion, or layer aggregation, along with the required division of aggregated layers to their respective output. Horizontal layer fusion improves performance by combining layers that take the same source tensor and apply the same operations with similar parameters.

These graph optimizations do not change the underlying computation in the graph; instead, they look to restructure the graph to perform the operations much faster and more efficiently.

TF-TRT is part of the TensorFlow binary, which means when you install `tensorflow-gpu`, you will be able to use TF-TRT too.

This currently only works if you use the TensorFlow Python API and not the C++ API.

1.1. Introduction
TensorFlow

TensorFlow is an open-source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) that flow between them. This flexible architecture lets you deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device without rewriting code.

TensorFlow was originally developed by researchers and engineers working on the Google Brain team within Google’s Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks (DNNs) research. The system is general enough to be applicable in a wide variety of other domains, as well.

For visualizing TensorFlow results, the Docker® image also contains TensorBoard. TensorBoard is a suite of visualization tools. For example, you can view the training histories as well as what the model looks like.

For information about the optimizations and changes that have been made to TensorFlow, see the TensorFlow Deep Learning Frameworks Release Notes.

TensorRT

The core of NVIDIA TensorRT is a C++ library that facilitates high performance inference on NVIDIA graphics processing units (GPUs). TensorRT takes a trained network, which consists of a network definition and a set of trained parameters, and produces a highly optimized runtime engine which performs inference for that network.

You can describe a TensorRT network using a C++ or Python API, or you can import an existing Caffe, ONNX, or TensorFlow model using one of the provided parsers (UFF for TensorFlow). The recommended method of importing TensorFlow models to TensorRT is using TF-TRT.

The TensorRT API includes import methods to help you express your trained deep learning models for TensorRT to optimize and run. By applying graph optimizations and layer fusion, TensorRT finds the fastest implementation of your models, 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 take advantage of the high speed mixed precision capabilities of Pascal, Volta, and Turing GPUs as an optional optimization.

For information about the optimizations and changes that have been made to TensorRT, see the TensorRT Release Notes. For specific TensorRT product documentation, see TensorRT documentation.
1.2. Benefits Of Integrating TensorFlow With TensorRT

TensorRT optimizes the largest subgraphs possible in the TensorFlow graph. The more compute in the subgraph, the greater benefit obtained from TensorRT. You want most of the graph optimized and replaced with the fewest number of TensorRT nodes for best performance. Based on the operations in your graph, it’s possible that the final graph might have more than one TensorRT node.

With the TensorFlow API, you can specify the minimum number of the nodes in a subgraph for it to be converted to a TensorRT node. Any sub-graph with less than the specified set number of nodes will not be converted to TensorRT engines even if it is compatible with TensorRT. This can be useful for models containing small compatible sub-graphs separated by incompatible nodes, in turn leading to tiny TensorRT engines.

1.3. What Capabilities Does TF-TRT Provide?

The Python TF-TRT API that can be used to optimize a TensorFlow frozen graph is `create_inference_graph`. This function has a number of parameters to configure the optimization.

```python
def create_inference_graph(input_graph_def, outputs, max_batch_size=1,
                         max_workspace_size_bytes=2 << 20,
                         precision_mode="fp32",
                         minimum_segment_size=3,
                         is_dynamic_op=False,
                         maximum_cached_engines=1,
                         cached_engine_batch_sizes=None,
                         use_calibration=True,
                         rewriter_config=None,
                         input_saved_model_dir=None,
                         input_saved_model_tags=None,
                         output_saved_model_dir=None,
                         session_config=None):
    """Python wrapper for the TRT transformation."
```

Where:

- **input_graph_def**
  This parameter is the `GraphDef` object that contains the model to be transformed.

- **outputs**
  This parameter lists the output nodes in the graph. Tensors which are not marked as outputs are considered to be transient values that may be optimized away by the builder.

- **max_batch_size**
  This parameter is the maximum batch size that specifies the batch size for which TensorRT will optimize. At runtime, a smaller batch size may be chosen. At runtime, larger batch size is not supported.
**max_workspace_size_bytes**

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

**precision_mode**

TF-TRT only supports models trained in FP32, in other words all the weights of the model should be stored in FP32 precision. That being said, TensorRT can convert tensors and weights to lower precisions during the optimization. The `precision_mode` parameter sets the precision mode; which can be one of `fp32`, `fp16`, or `int8`. Precision lower than FP32, meaning FP16 and INT8, would improve the performance of inference. The FP16 mode uses Tensor Cores or half precision hardware instructions, if possible. The INT8 precision mode uses integer hardware instructions.

**minimum_segment_size**

This parameter determines the minimum number of TensorFlow nodes in a TensorRT engine, which means the TensorFlow subgraphs that have fewer nodes than this number will not be converted to TensorRT. Therefore, in general smaller numbers such as 5 are preferred. This can also be used to change the minimum number of nodes in the optimized INT8 engines to change the final optimized graph to fine tune result accuracy.

**is_dynamic_op**

If this parameter is set to `True`, the conversion and building the TensorRT engines will happen during the runtime, which would be necessary if there are tensors in the graphs with unknown initial shapes or dynamic shapes. For more information, see #unique_7.

Conversion during runtime may increase the latency, depending on your model and how you use it.

**maximum_cached_engines**

In dynamic mode, this sets the maximum number of cached TensorRT engines per `TRTEngineOp`. For more information, see #unique_7.

**cached_engine_batch_sizes**

The list of batch sizes used to create cached engines, only used when `is_dynamic_op` is `True`. The length of the list should be smaller than `maximum Cached Engines`, and the dynamic TensorRT op will use this list to determine the batch sizes of the cached engines, instead of making the decision while in progress. This is useful when we know the most common batch size(s) the application is going to generate.

**cached_engine_batches**

The batch sizes used to pre-create cached engines.

**use_calibration**

This argument is ignored if `precision_mode` is not INT8.

- If set to `True`, a calibration graph will be created to calibrate the missing ranges. The calibration graph must be converted to an inference graph using `calib_graph_to_infer_graph()` after running calibration.
If set to `False`, quantization nodes will be expected for every tensor in the graph (excluding those which will be fused). If a range is missing, an error will occur.

Accuracy may be negatively affected if there is a mismatch between which tensors TensorRT quantizes and which tensors were trained with fake quantization.

**rewriter_config**

A `RewriterConfig` proto to append the `TensorRTOptimizer` to. If `None`, it will create one with default settings.

**input_saved_model_dir**

The directory to load the `SavedModel` containing the input graph to transform. Used only when `input_graph_def` is `None`.

**input_saved_model_tags**

A list of tags used to identify the `MetaGraphDef` of the `SavedModel` to load.

**output_saved_model_dir**

If not `None`, construct a `SavedModel` using the returned `GraphDef` and save it to the specified directory. This option only works when the input graph is loaded from a `SavedModel`, in other words, when `input_saved_model_dir` is specified and `input_graph_def` is `None`.

**session_config**

The `ConfigProto` used to create a `Session`. If not specified, a default `ConfigProto` will be used.

- Returns:
  - New `GraphDef` with `TRTEngineOps` placed in graph replacing subgraphs.

- Raises:
  - `ValueError`: If the provided precision mode is invalid.
  - `RuntimeError`: If the returned status message is malformed.
Chapter 2.
PREREQUISITES AND DEPENDENCIES

TF-TRT is part of the TensorFlow binary, which means when you install `tensorflow-gpu`, you will be able to use TF-TRT too. There are two ways to install TF-TRT:

- If you pull a TensorFlow container, for example 18.10, you will have all the software dependencies required by TF-TRT, therefore, you don’t need to install anything inside the container.

  Some scripts under the `nvidia-examples` directory require additional dependencies, such as the Python `requests` package. Refer to the README files for installation instructions for these additional dependencies.

- Or:

  follow these instructions to compile TensorFlow with TensorRT integration from its source.

2.1. Support Matrix

Table 1  Support matrix for required software

<table>
<thead>
<tr>
<th>TensorRT 5.1.2</th>
<th>Ubuntu</th>
<th>CUDA Toolkit</th>
<th>TensorFlow</th>
<th>Container Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>18.04</td>
<td>10.1.105</td>
<td>1.13.1</td>
<td>19.03</td>
</tr>
<tr>
<td></td>
<td>16.04</td>
<td></td>
<td></td>
<td>19.04</td>
</tr>
<tr>
<td>TensorRT 5.0.2</td>
<td>18.04</td>
<td>10.0.130</td>
<td>1.13.0-rc0</td>
<td>19.02</td>
</tr>
<tr>
<td></td>
<td>16.04</td>
<td></td>
<td>1.12.0</td>
<td>19.01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18.12</td>
</tr>
<tr>
<td>TensorRT 5.0.2</td>
<td>18.04</td>
<td>10.0.130</td>
<td>1.12.0-rc2</td>
<td>18.11</td>
</tr>
<tr>
<td></td>
<td>16.04</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Support matrix for TensorRT layers

For support matrix tables about the layers, see:

- [Layers Features Support Matrix](#)
- [Layers And Precision Support Matrix](#)

For a description of each supported TensorRT layer, see [TensorRT Layers](#).
The minimal setup in Python for TensorFlow integration with TensorRT requires that you import the TensorRT module and, in addition to your regular inference workflow, create a TensorRT inference graph, using the `create_inference_graph` function. For more information about this function, see What Capabilities Does TF-TRT Provide?

The specific workflows for creating a TensorRT inference graph from a TensorFlow model vary slightly depending on the format of the representation of your model. We offer three such workflows below, for SavedModel, frozen graph, and separate MetaGraph and checkpoint files. For more information, refer to the following articles:

- Saving and restoring TensorFlow models
- TensorFlow model files

### 3.1. Supported Ops

The following table lists the operators that are supported by TF-TRT. For a list of ops that TensorRT supports, see the TensorRT Support Matrix.

<table>
<thead>
<tr>
<th>Table 2 TF-TRT supported ops</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Op</strong></td>
</tr>
<tr>
<td>Abs</td>
</tr>
<tr>
<td>Add</td>
</tr>
<tr>
<td>AvgPool</td>
</tr>
<tr>
<td>BatchMatMul</td>
</tr>
<tr>
<td>BiasAdd</td>
</tr>
<tr>
<td>ConcatV2</td>
</tr>
<tr>
<td>Const</td>
</tr>
<tr>
<td>Conv2D</td>
</tr>
<tr>
<td>DepthwiseConv2dNative</td>
</tr>
<tr>
<td>Op</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>Div</td>
</tr>
<tr>
<td>Exp</td>
</tr>
<tr>
<td>ExpandDims</td>
</tr>
<tr>
<td>FusedBatchNorm</td>
</tr>
<tr>
<td>FusedBatchNormV2</td>
</tr>
<tr>
<td>Identity</td>
</tr>
<tr>
<td>Log</td>
</tr>
<tr>
<td>MatMul</td>
</tr>
<tr>
<td>Max</td>
</tr>
<tr>
<td>Maximum</td>
</tr>
<tr>
<td>MaxPool</td>
</tr>
<tr>
<td>Mean</td>
</tr>
<tr>
<td>Min</td>
</tr>
<tr>
<td>Minimum</td>
</tr>
<tr>
<td>Mul</td>
</tr>
<tr>
<td>Neg</td>
</tr>
<tr>
<td>Pad</td>
</tr>
<tr>
<td>Prod</td>
</tr>
<tr>
<td>RealDiv</td>
</tr>
<tr>
<td>Reciprocal</td>
</tr>
<tr>
<td>Relu</td>
</tr>
<tr>
<td>Relu6</td>
</tr>
<tr>
<td>Reshape</td>
</tr>
<tr>
<td>Rsqrt</td>
</tr>
<tr>
<td>Sigmoid</td>
</tr>
<tr>
<td>Snapshot</td>
</tr>
<tr>
<td>Softmax</td>
</tr>
<tr>
<td>Sqrt</td>
</tr>
<tr>
<td>Square</td>
</tr>
<tr>
<td>Squeeze</td>
</tr>
<tr>
<td>StrideSlice</td>
</tr>
<tr>
<td>Sub</td>
</tr>
<tr>
<td>Sum</td>
</tr>
<tr>
<td>Tanh</td>
</tr>
<tr>
<td>TopKV2</td>
</tr>
</tbody>
</table>
3.2. TF-TRT Workflow With A SavedModel

If you have a SavedModel representation of your TensorFlow model, you can create a TensorRT inference graph directly from your SavedModel, for example:

```python
# Import TensorFlow and TensorRT
import tensorflow as tf
import tensorflow.contrib.tensorrt as trt

# Inference with TF-TRT 'SavedModel' workflow:
graph = tf.Graph()
with graph.as_default():
    with tf.Session() as sess:
        # Create a TensorRT inference graph from a SavedModel:
        trt_graph = trt.create_inference_graph(  
            input_graph_def=None,
            outputs=None,
            input_saved_model_dir="/path/to/your/saved/model",
            input_saved_model_tags=['your_saved_model_tags'],
            max_batch_size=your_batch_size,
            max_workspace_size_bytes=max_GPU_mem_size_for_TRT,
            precision_mode="your_precision_mode")

        # Import the TensorRT graph into a new graph and run:
        output_node = tf.import_graph_def(  
            trt_graph,
            return_elements=['your_outputs'])

        sess.run(output_node)
```

Where, in addition to `max_batch_size`, `max_workspace_size_bytes` and `precision_mode`, you need to supply the following arguments to `create_inference_graph`:

- `input_saved_model_dir`: Path to your SavedModel directory.
- `input_saved_model_tags`: A list of tags used to identify the MetaGraphDef of the SavedModel to load.

And where:
- `['your_outputs']`: A list of the name strings for the final result nodes in your graph.

3.3. TF-TRT Workflow With A Frozen Graph

If you have a frozen graph of your TensorFlow model, you first need to load the frozen graph file and parse it to create a deserialized GraphDef. Then you can use the GraphDef to create a TensorRT inference graph, for example:

```python
# Import TensorFlow and TensorRT
import tensorflow as tf
import tensorflow.contrib.tensorrt as trt

# Inference with TF-TRT frozen graph workflow:
graph = tf.Graph()
```
with graph.as_default():
    with tf.Session() as sess:
        # First deserialize your frozen graph:
        with tf.gfile.GFile("/path/to/your/frozen/graph.pb", 'rb') as f:
            graph_def = tf.GraphDef()
            graph_def.ParseFromString(f.read())
        # Now you can create a TensorRT inference graph from your
        # frozen graph:
        trt_graph = trt.create_inference_graph(
            input_graph_def=graph_def,
            outputs=['your_output_node_names'],
            max_batch_size=your_batch_size,
            max_workspace_size_bytes=max_GPU_mem_size_for_TRT,
            precision_mode="your_precision_mode")
        # Import the TensorRT graph into a new graph and run:
        output_node = tf.import_graph_def(
            trt_graph,
            return_elements=['your_outputs'])
        sess.run(output_node)

Where, again in addition to max_batch_size, max_workspace_size_bytes,
and precision_mode, you need to supply the following argument to
create_inference_graph:
outputs
   A list of the name strings for the final result nodes in your graph.
And where:
"/path/to/your/frozen/graph.pb"
   Path to the frozen graph of your model.
["your_outputs"]
   A list of the name strings for the final result nodes in your graph. Same as outputs
   above.

3.4. TF-TRT Workflow With MetaGraph And
Checkpoint Files

If you don’t have a SavedModel or a frozen graph representation of your TensorFlow
model but have separate MetaGraph and checkpoint files, you first need to use these
to create a frozen graph to then feed into the create_inference_graph function, for
equivalence:

# Import TensorFlow and TensorRT
import tensorflow as tf
import tensorflow.contrib.tensorrt as trt
# Inference with TF-TRT `MetaGraph` and checkpoint files workflow:
graph = tf.Graph()
with graph.as_default():
    with tf.Session() as sess:
        # First create a `Saver` object (for saving and rebuilding a
        # model) and import your `MetaGraphDef` protocol buffer into it:
        saver = tf.train.import_meta_graph("/path/to/your/model.ckpt.meta")
        # Then restore your training data from checkpoint files:
        saver.restore(sess, "/path/to/your/model.ckpt")
        # Finally, freeze the graph:
        your_outputs = ['your_output_node_names']
        frozen_graph = tf.graph_util.convert_variables_to_constants(
sess,
tf.get_default_graph().as_graph_def(),
output_node_names=["your_outputs"])
# Now you can create a TensorRT inference graph from your
# frozen graph:
trt_graph = trt.create_inference_graph(
    input_graph_def=frozen_graph,
    outputs=["your_outputs"],
    max_batch_size=your_batch_size,
    max_workspace_size_bytes=max_GPU_mem_size_for_TRT,
    precision_mode="your_precision_mode")
# Import the TensorRT graph into a new graph and run:
output_node = tf.import_graph_def(
    trt_graph,
    return_elements=["your_outputs"])
sess.run(output_node)

Where, again in addition to max_batch_size, max_workspace_size_bytes
and precision_mode, you need to supply the following argument to
create_inference_graph:
outputs
    A list of the name strings for the final result nodes in your graph.

And where:
"/path/to/your/model.ckpt.meta"
    Path to the MetaGraphDef protocol buffer of your model. This is usually created and
    saved during training.
"/path/to/your/model.ckpt"
    Path to your latest checkpoint file saved during training.
["your_outputs"]
    A list of the name strings for the final result nodes in your graph. Same as outputs
    above.

3.5. Running Inference With TF-TRT For Image Classification

The NVIDIA TensorFlow Docker containers include example scripts for using a few
popular pre-trained image classification models to run inference on the ImageNet
validation set. The following sections walk you through how to use these scripts. These
scripts are copied from Examples for TensorRT in TensorFlow (TF-TRT) to the container.

3.5.1. Setting Up The Environment

We’ll be working in a recent NVIDIA TensorFlow Docker container (18.09 or 18.10) with
TensorFlow 1.10 or newer.

1. Clone the tensorflow/tensorrt repository.

        git clone https://github.com/tensorflow/tensorrt.git

2. Clone the tensorflow/models repository.

        git clone https://github.com/tensorflow/models.git
3. Add the directory where we cloned the `tensorflow/models` repository to
`PYTHONPATH` to install `tensorflow/models`.

```
cd models
export PYTHONPATH="$PYTHONPATH:$PWD"
```


```
cd research/slim
python setup.py install
```

5. Install the `requests` package.

```
 pip install requests
```

The `PYTHONPATH` environment variable is not saved between different shell sessions. To avoid having to set `PYTHONPATH` in each new shell session, you can add the following line to your `.bashrc` file:

```
export PYTHONPATH="$PYTHONPATH:/path/to/tensorflow/models"
```

### 3.5.2. Obtaining The ImageNet Data

The example Python scripts under `tensorrt/tftrt/examples/image-classification` support only the TFRecord format for inputting data. The scripts also assume that validation TFRecords are named according to the pattern: `validation-*-of-00128`. To download and process the ImageNet data, you can:

- Use the scripts provided in the `nvidia-examples/build_imagenet_data` directory in the NVIDIA TensorFlow Docker container `workspace` directory. Follow the instructions in the `README` file in that directory on how to use these scripts.

Or

- Use the scripts provided by TensorFlow Slim in the `tensorflow/models` repository under `research/slim`. Refer to the `README` file in the `research/slim` directory for instructions on how to use these scripts.

### 3.5.3. General Script Usage

The main Python script under `tensorrt/tftrt/examples/image-classification` is `image_classification.py`. Assuming that the ImageNet validation data are located under `~/data/imagenet/train-val-tfrecord`, we can evaluate inference with TF-TRT integration using the pre-trained ResNet V1 50 model as follows:

```
python image_classification.py --model resnet_v1_50
    --data_dir /data/imagenet/train-val-tfrecord
    --use_trt
    --precision fp16
```

Where:

- `--model` Which model to use to run inference, in this case ResNet V1 50.
- `--data_dir` Path to the ImageNet TFRecord validation files.
--use_trt
  Convert the graph to a TensorRT graph.

--precision
  Precision mode to use, in this case FP16.

The full set of options for the image_classification.py script follows.

-h, --help
  Show a help message and exit.

--model
  Which model to use to run inference. Currently supported models are
  mobilenet_v1, mobilenet_v2, nasnet_mobile, nasnet_large, resnet_v1_50,
  resnet_v2_50, resnet_v2_152, vgg_16, vgg_19, inception_v3, inception_v4.

--data_dir
  Path to the directory containing validation set TFRecord files.

--calib_data_dir
  Path to the directory containing TFRecord files for calibrating int8.

--model_dir
  Path to the directory containing model checkpoints. If not provided, checkpoints may
  be downloaded automatically and stored in --default_models_dir or --model for
  future use.

--default_models_dir
  Path to the directory where downloaded model checkpoints will be stored and loaded
  from if --model_dir is not provided.

--use_trt
  Convert the graph to a TensorRT graph.

--use_trt_dynamic_op
  Generate dynamic TensorRT operations that build the TensorRT graph and engine at
  run time.

--precision
  Precision mode to use. One of fp32 (default), fp16, or int8. fp16 and int8 only
  work in conjunction with --use_trt.

--batch_size
  Number of images per batch. (Default: 8).

--minimum_segment_size
  Minimum number of TensorFlow ops in a TensorRT engine. (Default: 2).

--num_iterations
  How many iterations (batches) to evaluate. If not supplied, the whole set will be
  evaluated.

--display_every
  Number of iterations executed between two consecutive displays of metrics. (Default: 100).

--use_synthetic
  Generate one batch of random data and use at every iteration.

--num_warmup_iterations
  Number of initial iterations skipped from timing. (Default: 50).

--num_calib_inputs
  Number of inputs (for example, images) used for calibration (the last batch is skipped
  in case it is not full). (Default: 500).
--max_workspace_size
Maximum allowable workspace size. (Default: 1<<32).

--cache
Save the graph to a graphs directory where the script is located. If a converted graph is present on disk under the graphs directory, it is loaded instead of building the graph again.

3.5.4. Script Output

The script first loads the pre-trained model. If given the flag --use_trt, the model is converted to a TensorRT graph, and the script displays (in addition to its initial configuration options):

- the number of nodes before conversion (num_nodes(native_tf))
- the number of nodes after conversion (num_nodes(trt_total))
- the number of separate TensorRT nodes (num_nodes(trt_only))
- the size of the graph before conversion (graph_size(MB)(native_tf))
- the size of the graph after conversion (graph_size(MB)(trt))
- how long the conversion took (time(s)(trt_conversion))

For example:

num_nodes(native_tf): 741
num_nodes(tftrt_total): 10
num_nodes(trt_only): 1
graph_size(MB)(native_tf): ***
graph_size(MB)(trt): ***
time(s)(trt_conversion): ***

For a list of supported operations that can be converted to a TensorRT graph, see Supported Ops.

The script then begins running inference on the ImageNet validation set, displaying run times of each iteration after the interval defined by the --display_every option (default: 100):

running inference...
step 100/6250, iter_time(ms)=**.****, images/sec=***
step 200/6250, iter_time(ms)=**.****, images/sec=***
step 300/6250, iter_time(ms)=**.****, images/sec=***
...

On completion, the script prints overall accuracy and timing information over the inference session:

results of resnet_v1_50:
   accuracy: 75.91
   images/sec: ***
   99th_percentile(ms): ***
   total_time(s): ***
   latency_mean(ms): ***
The accuracy metric measures the percentage of predictions from inference that match the labels on the ImageNet validation set. The remaining metrics capture various performance measurements:

- number of images processed per second (images/sec)
- total time of the inference session (total_time(s))
- the mean duration for each iteration (latency_mean(ms))
- the slowest duration for an iteration (99th_percentile(ms))

If you save this output from `inference.py`, you can verify the accuracy metric by passing this output to the `check_accuracy.py` script:

```
python check_accuracy.py \
   --input /path/to/your/output/from/inference.py \
   --tolerance 0.1
```

The accuracy metrics for most of the pre-trained models in `check_accuracy.py` are taken from the following source: https://github.com/tensorflow/models/tree/master/research/slim#pre-trained-models.

The metrics for the ResNet 50 models are taken from: https://github.com/tensorflow/models/tree/master/official/resnet#pre-trained-model.

The accuracy metric should match the acceptable accuracy of the model used, within the acceptable tolerance for the precision mode used:

```
checking accuracy...
input: /path/to/your/output/from/inference.py
tolerance: 0.1
PASS
```

The acceptable tolerance for FP32 and FP16 is 0.1%. For INT8 the acceptable tolerance is higher at 1.0%, reflecting the tradeoff between performance and precision.

### 3.5.5. Visualizing TF-TRT Graphs With TensorBoard

TensorBoard is a suite of visualization tools that make it easier to understand, debug, and optimize TensorFlow programs. You can use TensorBoard with `image_classification.py` to display how the pre-trained TensorFlow graphs are optimized with TensorRT integration.

After running inference on the ImageNet validation set, `image_classification.py` writes `Estimator` outputs used to evaluate the model (such as checkpoints and event files) to a `.model_dir` directory in the same directory where `image_classification.py` is located. You can launch a TensorBoard session on this directory with the following command:

```
tensorboard --logdir ./model_dir
```

In order to view a TensorBoard session running in a Docker container, you need to run the container with the `--publish` option to publish the port that TensorBoard uses (6006) to the machine hosting the container. The `--publish` option takes the form of `--publish [host machine IP address]:[host machine port]:[container`
You can then navigate a web browser to port 6006 on the machine hosting the Docker container (http://[IP address of host machine]:6006), where you can see an interactive visualization of the graph.

You can then run `inference.py` with the `--use_trt` parameter, and run TensorBoard on the ./model_dir again to see how the ResNet V1 50 graph changes when converted to a TensorRT graph. Figure 1 shows the graph of `image_classification.py` running inference with native TensorFlow using the ResNet V1 50 model (`--model resnet_v1_50`).

![TensorBoard Visualization of Inference with Native TensorFlow Using ResNet V1 50.](image)

This visualization displays all the nodes created by `image_classification.py` for running and evaluating inference using ResNet V1 50, so there are additional nodes for loading and saving data and for evaluating inference. If you double click the `resnet_model` node, you can see the nodes specific to the ResNet V1 50 model, as shown in Figure 2.
Figure 2  TensorBoard Visualization of ResNet V1 50 as a Native TensorFlow Graph.

Notice that the `resnet_model` subgraph contains 459 nodes as a native TensorFlow graph. You can then run `image_classification.py` with the `--use_trt` flag, and run TensorBoard on the `./model_dir` again to see that now the number of nodes in the `resnet_model` subgraph has been reduced from 459 to 4 when converted to a TensorRT graph as shown in Figure 3.
3.6. Using TF-TRT With ResNet V1 50

Here we walk through how to use the example Python scripts in the NVIDIA TensorFlow Docker containers under the nvidia-examples/tensorrt/tftrt/examples/image-classification/ directory with the ResNet V1 50 model.

Using TF-TRT with precision modes lower than FP32, that is, FP16 and INT8, improves the performance of inference. The FP16 precision mode uses Tensor Cores or half-precision hardware instructions, if possible, while the INT8 precision mode uses Tensor Cores or integer hardware instructions. INT8 mode also requires running a calibration step, which the image_classification.py script does automatically.

Below we use image_classification.py to compare the accuracy and timing performance of all the precision modes when running inference using the ResNet V1 50 model.

3.6.1. Tutorial: Native TensorFlow Using FP32
This is our baseline session running inference using native TensorFlow without TensorRT integration/conversion.

```python
python image_classification.py --model resnet_v1_50 \
--data_dir /data/imagenet/train-val-tfrecord \
--cache
```

We use the `--cache` flag above to allow the script to cache checkpoint and frozen graph files to use with future sessions.

Results:

```bash
results of resnet_v1_50:
accuracy: 75.90
images/sec: ***
99th percentile(ms): ***
total_time(s): ***
latency_mean(ms): ***
```

The accuracy metrics for the ResNet 50 models are taken from: Pre-trained model.

### 3.6.2. Tutorial: TF-TRT Using FP32

With this session, we use the same precision mode as in our native TensorFlow session (FP32), but this time we use the `--use_trt` flag to convert the graph to a TensorRT optimized graph.

```python
python image_classification.py --model resnet_v1_50 \
--data_dir /data/imagenet/train-val-tfrecord \
--use_trt \
--cache
```

Before the script starts running inference, it converts the TensorFlow graph to a TensorRT optimized graph with fewer nodes. Here the ResNet V1 50 model gets reduced from 741 native TensorFlow nodes to 10 total TF-TRT nodes:

```bash
num_nodes(native_tf): 741
num_nodes(tfrt_Total): 10
num_nodes(trt_only): 1
graph_size(MB)(native_tf): ***
graph_size(MB)(tft): ***
...
time(s)(trt_conversion): ***
```

For a list of supported operations that can be converted to a TensorRT graph, see Supported Ops.

Results:

```bash
results of resnet_v1_50:
accuracy: 75.90
images/sec: ***
99th percentile(ms): ***
total_time(s): ***
latency_mean(ms): ***
```
3.6.3. Tutorial: TF-TRT Using FP16

With this session, we continue to use TF-TRT conversion, but we reduce the precision mode to FP16, allowing the use of Tensor Cores for performance improvements during inference, while preserving accuracy within the acceptable tolerance level (0.1%).

```
python image_classification.py --model resnet_v1_50 \
  --data_dir /data/imagenet/train-val-tfrecord \
  --use_trt \n  --precision fp16 \n  --cache
```

Again, we see that the native TensorFlow graph gets converted to a TensorRT graph, from 741 native TensorFlow nodes to 10 total TF-TRT nodes:

```
num_nodes(native_tf): 741
num_nodes(tftrt_total): 10
num_nodes(trt_only): 1
graph_size(MB)(native_tf): ***
graph_size(MB)(trt): ***
... time(s)(trt_conversion): ***
```

Results:

```
results of resnet_v1_50:
  accuracy: 75.91
  images/sec: ***
  99th_percentile(ms): ***
  total_time(s): ***
  latency_mean(ms): ***
```

3.6.4. Tutorial: TF-TRT Using INT8

For this session we continue to use TF-TRT conversion, and we reduce the precision further to INT8 for faster computation. Because INT8 has significantly lower precision and dynamic range than FP32, the INT8 precision mode requires an additional calibration step before performing the type conversion. In this calibration step, inference is first run with FP32 precision on a calibration dataset to generate many INT8 quantizations of the weights and activations in the trained TensorFlow graph, from which are chosen the INT8 quantizations that minimize information loss. For more details on the calibration process, see the 8-bit Inference with TensorRT presentation.

The calibration dataset should closely reflect the distribution of the problem dataset. In this walkthrough, we use the same ImageNet validation set training data for the calibration data, with --calib_data_dir /data/imagenet/train-val-tfrecord.

```
python image_classification.py --model resnet_v1_50 \
  --data_dir /data/imagenet/train-val-tfrecord \
  --use_trt \n  --precision int8 \n  --calib_data_dir /data/imagenet/train-val-tfrecord \n  --cache
```

This time, we see the script performing the calibration step:
Calibrating INT8...
...
INFO:tensorflow:Evaluation [12/62]
INFO:tensorflow:Evaluation [18/62]
...

The process completes with the message:

**INT8 graph created.**

When the calibration step completes, it may take some time; we again see that the native TensorFlow graph gets converted to a TensorRT graph, from 741 native TensorFlow nodes to 10 total TF-TRT nodes:

```
num_nodes(native_tf): 741
num_nodes(tftrt_total): 10
num_nodes(trt_only): 1
graph_size(MB)(native_tf): ***
graph_size(MB)(tft): ***
...
time(s)(trt_conversion): ***
```

Also notice the following INT8-specific timing information:

```
time(s)(trt_calibration): ***
...
time(s)(trt_int8_conversion): ***
```

**Results:**

```
results of resnet_v1_50:
    accuracy: 75.90
    images/sec: ***
    99th_percentile(ms): ***
    total_time(s): ***
    latency_mean(ms): ***
```

### 3.7. Verified Models

We have verified that the following image classification models work with TF-TRT. Refer to the [release notes](https://www.nvidia.com) for any related issues on these models.

Preliminary tests have been performed on other types of models, for example, object detection, translation, recommender systems, and reinforcement learning; which can be potentially optimized with TF-TRT. We will continue to publish more details on them.

In the following table, we’ve listed the accuracy numbers for each model that we validate against. Our validation runs inference on the whole ImageNet validation dataset and provides the top-1 accuracy.
Table 3  Verified Models

<table>
<thead>
<tr>
<th>Model</th>
<th>Native TensorFlow FP32</th>
<th>TF-TRT FP32</th>
<th>TF-TRT FP16</th>
<th>TF-TRT INT8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Volta and Turing</td>
<td>Volta and Turing</td>
<td>Volta and Turing</td>
<td>Volta</td>
</tr>
<tr>
<td>MobileNet v1</td>
<td>71.01</td>
<td>71.01</td>
<td>70.99</td>
<td>69.49</td>
</tr>
<tr>
<td>MobileNet v2</td>
<td>74.08</td>
<td>74.08</td>
<td>74.07</td>
<td>73.96</td>
</tr>
<tr>
<td>NASNet - Large</td>
<td>82.72</td>
<td>82.71</td>
<td>82.70</td>
<td>Work in progress</td>
</tr>
<tr>
<td>NASNet - Mobile</td>
<td>73.97</td>
<td>73.85</td>
<td>73.87</td>
<td>73.19</td>
</tr>
<tr>
<td>ResNet-50 v1.5</td>
<td>76.51</td>
<td>76.51</td>
<td>76.48</td>
<td>76.23</td>
</tr>
<tr>
<td>ResNet-50 v2</td>
<td>76.43</td>
<td>76.37</td>
<td>76.4</td>
<td>76.3</td>
</tr>
<tr>
<td>VGG16</td>
<td>70.89</td>
<td>70.89</td>
<td>70.91</td>
<td>70.84</td>
</tr>
<tr>
<td>VGG19</td>
<td>71.01</td>
<td>71.01</td>
<td>71.01</td>
<td>70.82</td>
</tr>
<tr>
<td>Inception v3</td>
<td>77.99</td>
<td>77.99</td>
<td>77.97</td>
<td>77.92</td>
</tr>
<tr>
<td>Inception v4</td>
<td>80.19</td>
<td>80.19</td>
<td>80.19</td>
<td>80.14</td>
</tr>
</tbody>
</table>

### 3.8. INT8 Quantization

In addition to the default INT8 method of using calibration to determine the quantization scales per tensor (see Tutorial: TF-TRT Using INT8), TF-TRT allows users to bypass the calibration step by providing their own ranges instead. This can be used to support two use cases:

- Providing custom quantization ranges (possibly obtained from a custom calibrator or from other knowledge about your model).
- Deploying a model which has been trained using quantization-aware training.

To use either of these methods, set `precision_mode=“INT8”` and `use_calibration=False` in the arguments to `create_inference_graph`.

Since quantization-aware training requires many considerations, we recommend that most users use the calibration mode described in Tutorial: TF-TRT Using INT8.

### 3.8.1. Integrating Overview

TF-TRT also allows you to supply your own quantization ranges in case you do not want to use TensorRT’s built in calibrator. To do so, augment your TensorFlow model

---

with quantization nodes to provide the converter with the floating point range for each tensor.

You can use any of the following TensorFlow ops to provide quantization ranges:

- `QuantizeAndDequantizeV2`
- `QuantizeAndDequantizeV3`
- `FakeQuantWithMinMaxVars`
- `FakeQuantWithMinMaxArgs`

The following code snippet shows a simple hypothetical TensorFlow graph which has been augmented using `QuantizeAndDequantizeV2` ops to include quantization ranges which can be read by TF-TRT.

This particular graph has inputs which range from -1 to 1, so we set the quantization range for the input tensor to \([-1, 1]\).

The output of this particular `matmul` op has been measured to fit mostly between -9 to 9, so the quantization range for that tensor is set accordingly.

Finally, the output of this `bias_add` op has been measured to range from -3 to 3, therefore quantization range of the output tensor is set to \([-3, 3]\).

```
def my_graph(x):
    x = tf.quantize_and_dequantize_v2(x, input_min=-1.0, input_max=1.0)
    x = tf.matmul(x, kernel)
    x = tf.quantize_and_dequantize_v2(x, input_min=-9.0, input_max=9.0)
    x = tf.nn.bias_add(x, bias)
    x = tf.quantize_and_dequantize_v2(x, input_min=-3.0, input_max=3.0)
    return x
```

TensorRT may decide to fuse some operations in your graph. If you have provided a quantization range for a tensor which is removed due to fusion, your unnecessary range will be ignored.

You may also provide custom quantization ranges for some tensors and still use calibration to determine the rest of the ranges. To do this, provide quantization ranges in your TensorFlow model as described above using the supported quantization ops and perform the regular calibration procedure as described in the Tutorial: TF-TRT Using INT8 (with `use_calibration=True`).

### 3.8.2. Quantization-Aware Training

TF-TRT can also convert models for INT8 inference which have been trained using quantization-aware training. In quantization aware training, the error from quantizing weights and tensors to INT8 is modeled during training, allowing the model to adapt and mitigate the error.

The procedure for quantization-aware training is similar to that of Integrating Overview. Your TensorFlow graph should be augmented with quantization nodes and then the model will be trained as normal. The quantization nodes will model the error due to
quantization by clipping, scaling, rounding, and unscaling the tensor values, allowing the model to adapt to the error. You can use fixed quantization ranges or make them trainable variables.

**Important** INT8 inference is modeled as closely as possible during training. This means that you must not introduce a TensorFlow quantization node in places that will not be quantized during inference (due to a fusion occurring). Operation patterns such as Conv > Bias > Relu or Conv > Bias > BatchNorm > Relu are usually fused together by TensorRT, therefore, it would be wrong to insert a quantization node in between any of these ops.

### 3.8.2.1. Where To Add Quantization Nodes

We recommend starting by only adding quantization nodes after activation ops such as Relu. You can then try to convert the model using TF-TRT. TF-TRT will give you an error if a quantization range that it needs is missing, so you should add that range to your graph and repeat the process. Once you have enough ranges such that the graph can be converted successfully, you can train your model as usual.

Alternatively, a tool such as tf.contrib.quantize can automatically insert quantization nodes in the correct places in your model, but it is not guaranteed to exactly model inference using TensorRT, which may negatively impact your results.

### 3.9. Using TF-TRT To Generate A Stand-Alone TensorRT Plan

It is possible to execute your TF-TRT accelerated model using TensorRT’s C++ API or through the TensorRT Inference Server, without needing TensorFlow at all. You can use the following code snippet to extract the serialized TensorRT engines from your converted graph, where `trt_graph` is the output of `create_inference_graph`. This feature requires that your entire model converts to TensorRT, `is_dynamic_op=False`, and the precision mode is FP32 or FP16.

The script will display which nodes were excluded for the engine. If there are any nodes listed besides the input placeholders, TensorRT engine, and output identity nodes, your engine does not include the entire model.

```python
for n in trt_graph.node:
    if n.op == "TRTEngineOp":
        print("Node: %s, %s" % (n.op, n.name.replace("/", " ")))
        with tf.gfile.GFile("%s.plan" % (n.name.replace("/", " ")), 'wb') as f:
            f.write(n.attr["serialized_segment"].s)
    else:
        print("Exclude Node: %s, %s" % (n.op, n.name.replace("/", " ")))
```

The data in the `.plan` file can then be provided to `IRuntime: :deserializeCudaEngine` to use the engine in TensorRT. The input bindings will be named `TensorRTInputPH_0`, `TensorRTInputPH_1`, etc and the output bindings will be named `TensorRTOutputPH_0` similarly. For more information, see **Serializing A Model In C++**.
3.10. TensorFlow Ops Used In A TensorRT Op

Each TensorRT op in the optimized graph consists of a TensorRT network with a number of layers resulting from converting TensorFlow ops. To see the original subgraph including the TensorFlow ops that were converted to a particular TensorRT op, you can see the `segment_funcdef_name` attribute stored in the TensorRT op. For example, for a TensorRT op named `TRTEngineOp_0`, the native subgraph is stored as `TRTEngineOp_0_native_segment`. This native segment is also visible on TensorBoard.

3.11. Unconverted TensorFlow Ops

The conversion algorithm optimizes an input graph by converting TensorFlow ops to TensorRT layers, however certain TensorFlow ops (due to their type, input shapes, type of inputs, etc.) can't be converted. There are various ways to check what operators are converted or not converted.

**TF-TRT log**

Look for the following that specifies what operators are not converted.

The operators that are skipped because of segment size being larger than `minimum_segment_size` are not specified in this list.

There are 5 ops of 4 different types in the graph that are not converted to TensorRT: `ArgMax`, `Identity`, `Placeholder`, `NoOp`. For more information see Supported Ops.

**TF-TRT verbose logging**

Increase the verbosity of logging to 1 to see the reason why each op is not converted.

**GraphDef**

Print the nodes (only need the op type) of the graph to see what ops are not converted to TensorRT. Something similar to the following works:

```
frozen_graph = the output of TF-TRT API.

for node in frozen_graph.node:
    print(node.op)
```

As an example, the following output means all ops in the graph are converted to `TRTEngineOp_0` except `NoOp`, `Placeholder`, `Identity`, and `ArgMax`.

```
NoOp
Placeholder
Identity
TRTEngineOp_0
ArgMax
```
TensorBoard
After you write the graph structure to be loaded on TensorBoard, you can then see the graph including the ops that are converted and unconverted.

3.12. Debugging Tools

The following tools can be used to help debug.

Verbose logging
Increase the verbosity level in TensorFlow logs, for example:

```python
TF_CPP_VMODULE=segment=2,convert_graph=2,convert_nodes=2,trt_engine=1 python ...
```

This is the preferred way because most users care about the logs printed from a few C++ files. The other options would increase the verbosity throughout TensorFlow which makes the logs become much harder to read.

There are other ways of increasing the verbosity level, however, they produce unreadable logs, for example:

```python
TF_CPP_MIN_LOG_LEVEL=2 python ...
TF_CPP_MIN_VLOG_LEVEL=2 python ...
```

TensorBoard
TensorBoard is typically used to look at the TensorFlow graph, what nodes are in it, what nodes are not converted to TensorRT, what nodes are attached to TensorRT nodes, for example TRTEngineOp, what TF subgraph was converted to TensorRT node, and even the shape of the tensors in the graph. For more information, see Visualizing TF-TRT Graphs With TensorBoard.

Profiler
There are a couple of different ways you can use profiling to debug. For information about the Tensorflow profiler, see How to profile TensorFlow. For information about how to debug using Nsight Systems, see NVIDIA Nsight Systems.

nvprof
You can also use nvprof, which is the easiest. For example,

```python
nvprof python ...
```

NTXX range
TensorFlow inside the NVIDIA container is built with NVTX ranges for TensorFlow operators. This means every operator (including TRTEngineOp) executed by TensorFlow will appear as a range on the visual profiler which can be linked against the CUDA kernels executed by that operator. This way, you can check the kernels executed by TensorRT, the timing of each, and compare that information with the profile of the original TensorFlow graph before conversion.
Chapter 4.
SAMPLES

For specific tutorials and samples, see nvidia-examples/tensorrt inside the TensorFlow container. For more information, see NGC TensorFlow container and the TensorFlow User Guide.

The TensorFlow samples include the following features:

- Download checkpoints or pre-trained models from TensorFlow model zoo.
- Run inference using either native TensorFlow or TF-TRT.
- Achieve the accuracy that matches the accuracy obtained by TensorFlow slim or TensorFlow official scripts.
- Report any metrics including throughput, latency (mean and 99th percentile), node conversion rates, top1 accuracy, total time.
- Support precision modes FP32, FP16, and INT8 for TF-TRT.
- Work with TFRecord dataset only. Tested the scripts with the ImageNet dataset.
- Run benchmark with synthetic data in order to measure the performance of the inference only regardless of I/O pipeline.
Chapter 5. 
BEST PRACTICES

Ensure you are familiar with the following best practice guidelines:

**Batch normalization**

The `FusedBatchNorm` operator is supported, which means this operator is converted to the relevant TensorRT batch normalization layers. This operator has an argument named `is_training` which is a boolean to indicate whether the operation is for training or inference. The operator is converted to TensorRT only if `is_training=False`.

When converting a model from Keras, ensure you call the function `keras.backend.set_learning_phase(0)` to ensure that your batch normalization layers are built in inference mode and therefore are eligible to be converted. We recommend to call this function at the very beginning of your python script, right after `import keras`.

**Batch size**

TensorRT optimizes the graph for a maximum batch size that must be provided during the conversion. During inference, while using this same maximum batch size provides the best performance, batch sizes smaller than this maximum may not give the best possible performance for that batch size. Also, running inference with batch sizes larger than the maximum batch size is not supported by TensorRT.

**Conversion on the target machine**

You need to execute the conversion on the machine on which you will run inference. This is because TensorRT optimizes the graph by using the available GPUs and thus the optimized graph may not perform well on a different GPU.

**I/O bound**

Inference workloads have a very low latency especially if the model is small and the inference engine is optimized with TF-TRT. In such cases, we often see that the bottleneck in the pipeline is loading inputs from disks or networks (such as jpeg images or TFRecords) and preprocessing them before feeding them into the inference engine. We recommend to always profile your inference during runtime and find the bottlenecks. A common method in improving the I/O pipeline is to use all available optimizations in the I/O libraries such as (tf.data or nvidia/dali). These available optimizations include multithreaded I/O and image processing and performing image processing on GPUs.
INT8 calibration conversion
After you convert the graph with INT8 precision mode, the converted TensorRT graph needs to be calibrated. The calibration is a very slow process that can take 1 hour.

Memory usage
TensorRT allocates memory using an allocator provided by TensorFlow. This means that if you specify the fraction of GPU memory allowed for TensorFlow (using the `per_process_gpu_memory_fraction` parameter of the `GPUOptions`), then TensorRT can use the remaining memory. As an example, `per_process_gpu_memory_fraction=0.67` would allocate 67% of GPU memory for TensorFlow, making the remaining 33% available for TensorRT engines. The `max_workspace_size_bytes` parameter can be also used to set the maximum amount of memory that TensorRT should allocate.

Minimum segment size
TensorRT does not convert TensorFlow subgraphs with fewer nodes than the value defined by the `minimum_segment_size` parameter. Therefore, to achieve the best performance we recommend using the smallest possible value of `minimum_segment_size` for which the converter doesn't crash. There are two ways to determine this smallest possible value.

- You can start by setting `minimum_segment_size` to a large number such as 50 and decrease this number until the converter crashes. With this method you should see better performance as you decrease the `minimum_segment_size` parameter.

- Or

- You can start with `minimum_segment_size` set to a small number such as 2 or 3 and increase this number until the converter completes its process without crashing.

Number of nodes
Each TensorFlow graph has a certain number of nodes. The TF-TRT conversion always reduces the number of nodes through replacing a subset of those nodes by a single TensorRT node. For example, converting a TensorFlow graph of Res-Net with 743 nodes, could result in a new graph with 19 nodes out of which 1 node is a TensorRT node that will be executed by a TensorRT engine. A good way to find out whether any optimization has happened or how much of the graph is optimized is to compare the number of nodes before and after the conversion. We expect >90% of nodes to be replaced by TensorRT nodes for the supported models.

Tensor Cores
If you have a GPU with Tensor Core capability, you can simply set the precision mode to FP16 during the conversion, and then TensorRT will run the relevant operators on Tensor Cores.

- Not all GPUs support the ops required for all precisions.
- Tensor Cores can be used only for `MatMul` and convolutions if the dimensions are multiples of 8. To verify whether Tensor Cores are being used in your inference, you can profile your inference run with `nvprof` and check if all the
GEMM CUDA kernels (GEMM is used by MatMul and convolution) have 884 in their name.
Chapter 6.
PERFORMANCE

The amount of speedup we achieve by optimizing the TensorFlow models through TF-TRT varies a lot depending on various factors; such as, the type of nodes, network architecture, batch size, TF-TRT workspace size, and precision mode.

To optimize your performance, ensure you are familiar with the following tips:

- The set of operators supported by TRT or TF-TRT is limited.
- The possibility of node fusion is determined by the type of nodes that are directly connected.
- TF-TRT optimizes the graph for one particular batch size, and thus inference with a batch size smaller than that may not obtain the best achievable performance.
- Certain algorithms in TensorRT need a larger workspace, therefore, decreasing the TF-TRT workspace size might result in not running the fastest TensorRT algorithms possible.
Chapter 7.
TROUBLESHOOTING

7.1. Support

For more information, see:

- TensorFlow Release Notes
- TensorFlow User Guide
- TensorFlow tutorials
- TensorFlow API
- Install TensorFlow on Ubuntu
- GitHub TensorFlow / TensorRT
- NVIDIA Developer Blogs: TensorRT Integration Speeds Up TensorFlow Inference

To ask questions and get involved in discussions in all things related to TensorRT, access the NVIDIA DevTalk TensorRT forum at https://devtalk.nvidia.com/default/board/304/tensorrt/.
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, DGX, DGX-1, DGX-2, and DGX Station are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2019 NVIDIA Corporation. All rights reserved.