Accelerating Inference In TensorFlow With TensorRT (TF-TRT)

User Guide

SWE-SWDOCTFT-001-INTG _v001 | May 2021

Table of Contents

Chapter 1. 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, certain layers (such as convolution, bias, and ReLU) 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.

Note: 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.

For more information about TF-TRT, see High performance inference with TensorRT Integration.

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.

TensorRT provides API's via C++ and Python that help to express deep learning models via the Network Definition API or load a pre-defined model via the parsers that allows TensorRT to optimize and run them on an NVIDIA GPU. TensorRT applies graph optimizations, layer fusion, among other optimizations, while also finding the fastest implementation of that model leveraging a diverse collection of highly optimized kernels. TensorRT also supplies a runtime that you can use to execute this network on all of NVIDIA's GPU's from the Kepler generation onwards.

TensorRT also includes high speed mixed precision and Tensor Core routines on the supported GPU architectures.

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

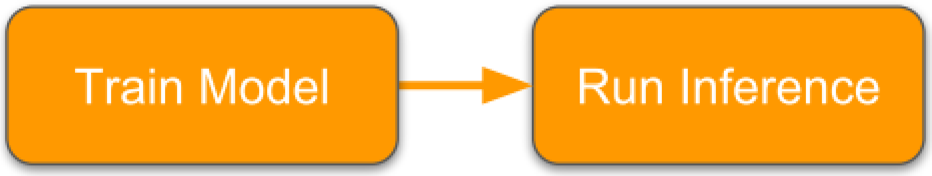
TensorRT optimizes the largest subgraphs possible in the TensorFlow graph. The more compute in the subgraph, the greater benefit obtained from TensorRT. You want most of the graph optimized and replaced with the fewest number of TensorRT nodes for best performance. Based on the operations in your graph, 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 nodes in a subgraph for it to be converted to a TensorRT node. Any sub-graph with less than the specified 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. TF-TRT Workflow

The following diagram shows the typical workflow in deploying a trained model for inference.

Figure 1. Deploying a trained model workflow.



In order to optimize the model using TF-TRT, the workflow changes to one of the following diagrams depending on whether the model is saved in `SavedModel` format or regular checkpoints. Optimizing with TF-TRT is the extra step that is needed to take place before deploying your model for inference.

Figure 2. Showing the SavedModel format.

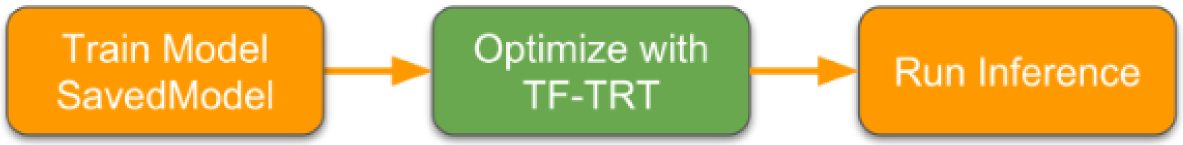
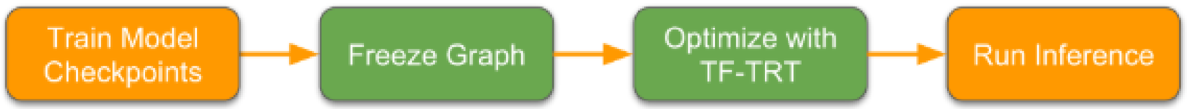


Figure 3. Showing a Frozen graph.



Chapter 2. Using TF-TRT

2.1. Installing TF-TRT

NVIDIA containers of TensorFlow are built with enabling TensorRT, which means TF-TRT is part of the TensorFlow binary in the container and can be used out of the box. The container has all the software dependencies required to run TF-TRT. If you want to build TensorFlow from scratch, follow these instructions. You need to enable TensorRT in bazel configuration (it's disabled by default).

There are also examples provided in the container under the `nvidia-examples` directory which can be executed to test TF-TRT, however, certain additional packages might be required to execute these examples. For step-by-step instructions on how to install TF-TRT, refer to the `README` file of each example.

2.2. Examples

TensorFlow 1.x

The following code snippets show how to use TF-TRT in TensorFlow 1.x (with default configuration) on a given model for each of the formats: `SavedModel` or a frozen graph.

SavedModel format:

```
from tensorflow.python.compiler.tensorrt import trt_convert as trt
converter = trt.TrtGraphConverter(input_saved_model_dir=input_saved_model_dir)
converter.convert()
converter.save(output_saved_model_dir)
```

Frozen graph:

```
from tensorflow.python.compiler.tensorrt import trt_convert as trt
converter = trt.TrtGraphConverter(
 input_graph_def=frozen_graph,
 nodes_blacklist=['logits', 'classes'])
frozen_graph = converter.convert()
```

TensorFlow 2.0

The following code snippet shows how to use TF-TRT in TensorFlow 2.0 (with default configuration) for a given `SavedModel`. The only difference between this code and the one above for TensorFlow 1.x is the suffix V2 in the converter class name. Refer to TF-TRT API In TensorFlow 1.x to learn more about other differences in the TensorFlow 2.0 API.

Note: Converting frozen graphs is no longer supported in TensorFlow 2.0.

```
from tensorflow.python.compiler.tensorrt import trt_convert as trt
converter = trt.TrtGraphConverterV2(input_saved_model_dir=input_saved_model_dir)
converter.convert()
converter.save(output_saved_model_dir)
```

Refer to the following guides on how to save your models in `SavedModel` format or how to turn a graph and checkpoints into a frozen graph:

- SavedModel format
- Freezing a graph
- We use a slightly different method to freeze graphs in our examples

As you can see, `TrtGraphConverter.save` should only be used in case of `SavedModel` because this function converts the optimized graph into `SavedModel`.

This section gives a number of examples for how to use TensorFlow Python APIs to run inference on a model. See GitHub: TF-TRT for a more complete set of examples. TensorFlow has also C++ API that can be used for running inference on models.

2.2.1. TF-TRT 1.x 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:

```
import tensorflow as tf
from tensorflow.python.compiler.tensorrt import trt_convert as trt

converter = trt.TrtGraphConverter(
    input_saved_model_dir=input_saved_model_dir,
    max_workspace_size_bytes=(11<32),
    precision_mode=”FP16”,
    maximum_cached_engines=100)
converter.convert()
converter.save(output_saved_model_dir)

with tf.Session() as sess:
    # First load the SavedModel into the session
    tf.saved_model.loader.load(
        sess, [tf.saved_model.tag_constants.SERVING],
       output_saved_model_dir)
    output = sess.run([output_tensor], feed_dict={input_tensor: input_data})
```

2.2.2. TF-TRT 1.x 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:

```
import tensorflow as tf
from tensorflow.python.compiler.tensorrt import trt_convert as trt
with tf.Session() as sess:
    # First deserialize your frozen graph:
    with tf.gfile.GFile(“/path/to/your/frozen/graph.pb”, ‘rb’) as f:
        frozen_graph = tf.GraphDef()
        frozen_graph.ParseFromString(f.read())
    # Now you can create a TensorRT inference graph from your
    # frozen graph:
    converter = trt.TrtGraphConverter(
     input_graph_def=frozen_graph,
     nodes_blacklist=['logits', 'classes']) #output nodes
    trt_graph = converter.convert()
    # Import the TensorRT graph into a new graph and run:
    output_node = tf.import_graph_def(
        trt_graph,
        return_elements=['logits', 'classes'])
    sess.run(output_node)
```

2.2.3. TF-TRT 1.x 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 SavedModel or a frozen graph to then feed into TF-TRT. The following example shows how to freeze a graph from checkpoints:

```
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:
    frozen_graph = tf.graph_util.convert_variables_to_constants(
        sess,
        tf.get_default_graph().as_graph_def(),
        output_node_names=['logits', 'classes'])
```

2.2.4. TF-TRT 2.0 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:

```
import tensorflow as tf
from tensorflow.python.compiler.tensorrt import trt_convert as trt

conversion_params = trt.DEFAULT_TRT_CONVERSION_PARAMS
conversion_params = conversion_params._replace(
    max_workspace_size_bytes=(1<<32))
conversion_params = conversion_params._replace(precision_mode="FP16")
conversion_params = conversion_params._replace(
    maximum_cached_engines=100)

converter = trt.TrtGraphConverterV2(
    input_saved_model_dir=input_saved_model_dir,
    conversion_params=conversion_params)
converter.convert()
def my_input_fn():
# Input for a single inference call, for a network that has two input tensors:
  Inp1 = np.random.normal(size=(8, 16, 16, 3)).astype(np.float32)
  inp2 = np.random.normal(size=(8, 16, 16, 3)).astype(np.float32)
  yield (inp1, inp2)
converter.build(input_fn=my_input_fn)
converter.save(output_saved_model_dir)
```

```
saved_model_loaded = tf.saved_model.load(
    output_saved_model_dir, tags=[tag_constants.SERVING])
graph_func = saved_model_loaded.signatures[
    signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY]
frozen_func = convert_to_constants.convert_variables_to_constants_v2(
    graph_func)
output = frozen_func(input_data)[0].numpy()
```

2.3. TF-TRT API In TensorFlow 1.x

The TF-TRT API includes a single Python class called `TrtGraphConverter` in addition to a few methods that you can call. The API is defined in a Python module in the TensorFlow source code.

Note: The original Python function `create_inference_graph` that was used in TensorFlow 1.13 and earlier is deprecated in TensorFlow >1.13 and removed in TensorFlow 2.0.

The constructor of `TrtGraphConverter` supports the following optional arguments. The most important of these arguments that are used to configure TF-TRT or TensorRT to get better performance are explained in the following sections of this chapter.

`input_saved_model_dir`

Default value is `None`. This is the directory to load the SavedModel which contains the input graph to transforms and is used only when `input_graph_def` is `None`.

`input_saved_model_tags`

Default value is `None`. This is a list of tags to load the SavedModel.

`input_saved_model_signature_key`

Default value is `None`. This is the key of the signature to optimize the graph for.

`input_graph_def`

Default value is `None`. This is a GraphDef object containing a model to be transformed. If set to `None`, the graph will be read from the SavedModel loaded from `input_saved_model_dir`.

`nodes_blacklist`

Default value is `None`. This is a list of node names to prevent the converter from touching.

`session_config`

Default value is `None`. This is the `ConfigProto` used to create a `Session`. It's also used as a template to create a TRT-enabled `ConfigProto` for conversion. If not specified, a default `ConfigProto` is used.

`max_batch_size`

Default value is `1`. This is the max size for the input batch.

`max_workspace_size_bytes`

Default value is 1GB. This is the maximum GPU temporary memory which the TensorRT engine can use at execution time. This corresponds to the `workspaceSize` parameter of `nvinfer1::IBuilder::setMaxWorkspaceSize()`.

precision_mode

Default value is `TrtPrecisionMode.FP32`. This is one of `TrtPrecisionMode.supported_precision_modes()` , in other words, "FP32", "FP16" or "INT8" (lowercase is also supported).

minimum_segment_size

Default value is `3`. This is the minimum number of nodes required for a subgraph to be replaced by `TRTEngineOp`.

is_dynamic_op

Default value is `False`. Whether to generate dynamic TensorRT ops which will build the TensorRT network and engine at run time.

maximum_cached_engines

Default value is `1`. This is the maximum number of cached TensorRT engines in TensorFlow for each TensorRT subgraph. If the number of cached engines is already at max but none of them can serve the input, one of the cached engines gets evicted based on the LRU policy and a new engine gets built and added to the cache.

use_calibration

Default value is `True`. 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 by running calibration with `calibrate()`. 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.

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

use_function_backup

Default value is `True`. If set to `True`, it will create a `FunctionDef` for each subgraph that is converted to TensorRT op, and if TensorRT ops fail to execute at runtime, it'll invoke that function as a fallback.

The main methods you can use in the `TrtGraphConverter` class are the following:

TrtGraphConverter.convert()

This method runs the conversion and returns the converted `GraphDef`. The conversion and optimization that are performed depends on the arguments passed to the constructor as explained above.

In dynamic mode, where the TensorRT engines are built at runtime, this method only segments the graph in order to separate the TensorRT subgraphs, i.e. optimizing each TensorRT subgraph happens later during runtime. In static mode, the optimization also happens in this method and thus this method becomes time consuming.

TrtGraphConverter.calibrate(fetch_names, num_runs, feed_dict_fn, input_map_fn)

This method runs the INT8 calibration and returns the calibrated `GraphDef`. This method should be called after `convert()` in order to execute the calibration on the converted graph. The method accepts the following arguments:

`fetch_names`

A list of output tensor name to fetch during calibration.

`num_runs`

Number of runs of the graph during calibration.

`feed_dict_fn`

A function that returns a dictionary mapping input names (as strings) in the GraphDef to be calibrated to values (e.g. `Python list`, `NumPy arrays`, etc).One and only one of `feed_dict_fn` and `input_map_fn` should be specified.

`input_map_fn`

A function that returns a dictionary mapping input names (as strings) in the `GraphDef` to be calibrated to Tensor objects. The values of the named input tensors in the `GraphDef` to be calibrated will be re-mapped to the respective `Tensor` values during calibration. One and only one of `feed_dict_fn` and `input_map_fn` should be specified.

`TrtGraphConverter.save(output_saved_model_dir)`

This method saves the converted graph as a `SavedModel`.

Note: The saved TensorFlow model is still not optimized yet with TensorRT (engines are not built) in case of dynamic mode.

2.4. TF-TRT API In TensorFlow 2.0

The TF-TRT API in TensorFlow 2.0 has a number of differences with the API in TensorFlow 1.x which are explained here. The name of the Python class has the suffix V2, i.e. `TrtGraphConverterV2`.

Note: The original Python function `create_inference_graph` that was used in TensorFlow 1.13 and earlier is removed in to TensorFlow 2.0.

The constructor of `TrtGraphConverterV2` supports the following optional arguments. The most important of these arguments that are used to configure TF-TRT or TensorRT to get better performance are stored in the `namedtuple TrtConversionParams` as explained in the following sections of this chapter.

`input_saved_model_dir`

Default value is `None`. This is the directory to load the `SavedModel` which contains the input graph to transforms.

`input_saved_model_tags`

Default value is `None`. This is a list of tags to load the `SavedModel`.

`input_saved_model_signature_key`

Default value is `None`. This is the key of the signature to optimize the graph for.

`conversion_params`

Default value is `DEFAULT_TRT_CONVERSION_PARAMS`. An instance of `namedtuple TrtConversionParams` consisting the following items:

`rewriter_config_template`

A template `RewriterConfig` proto used to create a TRT-enabled `RewriterConfig`. If `None`, it will use a default one.

`max_workspace_size_bytes`

Default value is 1GB. The maximum GPU temporary memory which the TensorRT engine can use at execution time. This corresponds to the `workspaceSize` parameter of `nvinfer1::IBuilder::setMaxWorkspaceSize()`.

`precision_mode`

Default value is `TrtPrecisionMode.FP32`. This is one of `TrtPrecisionMode.supported_precision_modes()`, in other words, `FP32`, `FP16` or `INT8` (lowercase is also supported).

`minimum_segment_size`

Default value is `3`. This is the minimum number of nodes required for a subgraph to be replaced by `TRTEngineOp`.

`maximum_cached_engines`

Default value is `1`. This is the maximum number of cached TensorRT engines in dynamic TensorRT ops. If the number of cached engines is already at max but none of them can serve the input, the `TRTEngineOp` will fall back to run the TensorFlow function based on which the `TRTEngineOp` is created.

`use_calibration`

Default value is `True`. 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 by running calibration with `calibrate()`. 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.

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

The main methods you can use in the `TrtGraphConverter` class are the following:

`TrtGraphConverter.convert(calibration_input_fn)`

This method runs the conversion and returns the converted TensorFlow function (note that this method returns the converted `GraphDef` in TensorFlow 1.x). The conversion and optimization that are performed depends on the arguments passed to the constructor as explained above.

This method only segments the graph in order to separate the TensorRT subgraphs, i.e. optimizing each TensorRT subgraph happens later during runtime (in TensorFlow 1.x this behaviour depends on `is_dynamic_mode` but this argument is not supported in TensorFlow 2.0 anymore; i.e. only `is_dynamic_op=True` is supported).

This method has only one optional argument which should be used in case INT8 calibration is desired. The argument `calibration_input_fn` is a generator function that yields input data as a list or tuple, which will be used to execute the converted signature for INT8 calibration. All the returned input data should have the same shape. Note that in TensorFlow 1.x, the INT8 calibration was performed using the separate method `calibrate()` which is removed from TensorFlow 2.0.

`TrtGraphConverter.build(input_fn)`

This method optimizes the converted function (returned by `convert()`) by building TensorRT engines. This is useful in case the user wants to perform the optimizations before runtime. The optimization is done by running inference on the converted function using the input data received from the argument `input_fn`. This argument is a generator function that yields input data as a list or tuple.

`TrtGraphConverter.save`

This method saves the converted function as a `SavedModel`. Note that the saved TensorFlow model is still not optimized yet with TensorRT (engines are not built) in case `build()` is not called.

2.5. Precision Mode

The model that TF-TRT optimizes can have the graph or parameters stored in float32 (FP32) or float16 (FP16). Regardless of the datatype of the model, TensorRT can convert tensors and weights to lower precisions during the optimization. The argument `precision_mode` sets the precision mode; which can be one of `FP32`, `FP16`, or `INT8`. Precisions 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. Refer to the Performance section for more information about how lower precision can improve performance. Also, see the INT8 Quantization section for more information about it and the various ways of using it.

Use FP16 to get the best performance without losing accuracy. If your experiments show that INT8 quantization doesn't degrade the accuracy of your model, use INT8 because it provides a much higher performance.

2.6. Static Or Dynamic Mode

There are two modes in TF-TRT: static (default mode) and dynamic. The static mode is enabled when `is_dynamic_op=False` and otherwise dynamic mode is enabled. The main difference between these two modes is that the TensorRT engines are built offline (by `TrtGraphConverter.convert`) when you are in static mode, whereas in the dynamic mode, the TensorRT engines are built during runtime when the actual inference happens.

Use dynamic mode if you have a graph that has undefined shapes (dimensions that are `None` or `-1`). If you try to convert a model which has undefined shapes while in static mode, TF-TRT will issue the following warning:

```
Input shapes must be fully defined when in static mode. Please try is_dynamic_op=True.
```

Dynamic mode allows you to have unknown shapes in your model, despite the fact that TensorRT requires all shapes to be fully defined. In this mode, TF-TRT creates a new TensorRT engine for each unique input shape that is supplied to the model.

Dynamic mode allows you to have unknown shapes in your model, despite the fact that TensorRT requires all shapes to be fully defined. In this mode, TF-TRT creates a new TensorRT engine for each unique input shape that is supplied to the model. For example, you may have an image classification network that works on images of any size where the input placeholder has the shape `[?, ?, ?, 3]`. If you were to first send a batch of images to the model with shape `[8, 224, 224, 3]`, a new TensorRT engine will be created that is optimized for those dimensions. Since the engine will have to be built at this time, this first batch will take longer to execute than usual. If you later send more images with the same shape of `[8, 224, 224, 3]`, the previously built engine will be used immediately with no additional overhead. If you instead send a batch with a different shape, a new engine would have to be created for that shape. The argument `maximum_cached_engines` can be used to control how many engines will be stored at a time, for each individual `TRTEngineOp` in the graph.

Static mode does not support post-training quantization (INT8 calibration). The conversion switches to dynamic mode in case users try to use static mode for INT8 calibration.

2.6.1. Cache And Variable Batch Sizes

TensorRT engines are cached in an LRU cache located in the `TRTEngineOp` op. The key to this cache are the shapes of the op inputs. For example, a new engine is created if the cache is empty or if an engine for a given input shape does not exist in the cache. You can control the number of engines cached with the argument `maximum_cached_engines`.

TensorRT uses the batch size of the inputs as one of the parameters to select the highest performing CUDA kernels. The batch size is provided as the first dimension of the inputs. The batch size is determined by input shapes during execution when `is_dynamic_op` is `true` (TF 2 default), and by the argument `max_batch_size` when `is_dynamic_op` is `false` (TF 1 default). An engine can be reused for a new input, if:

- the engine batch size is greater than or equal to the batch size of new input, and
- the non-batch dims match the new input

If you want to have a conservative memory usage, set `maximum_cached_engines` to 1 to force any existing cache to be evicted each time a new engine is created. On the other hand, if your main goal is to reduce latency, then increase `maximum_cached_engines` to prevent the recreation of engines as much as possible. Caching more engines uses more resources on the machine, however, that is not a problem for typical models.

Note: Setting `maximum_cached_engines` to a very large number like 100 doesn't increase the memory usage until that many engines actually get built during runtime (`maximum_cached_engines` is just an upper bound on the number of engines in the cache).

2.7. Controlling Minimum Number Of Nodes In TensorRT Subgraphs

TensorFlow subgraphs that are optimized by TensorRT include a certain number of operators. If the number of operators included in the subgraph is very small, then launching a TensorRT engine for that subgraph may not be efficient compared to executing the original subgraph. You can control the size of subgraphs by using the argument `minimum_segment_size`. Setting this value to `x` (default is 3) will not generate TensorRT engines for subgraphs consisting of less than `x` nodes.

If you want to avoid having very small TensorRT engines (meaning, include a very small number of layers), then increase `minimum_segment_size`. That might help with avoiding potential overheads introduced by those small TensorRT engines, and also can get around any possible errors that arise from those engines. We have observed that the default value `3` gives the best performance for most models.

Note: If you use a very large value for `minimum_segment_size`, then the optimization is only applied to very large subgraphs which could potentially leave possible optimizations that are applicable to smaller subgraphs.

2.8. Memory Management

TensorRT stores weights and activations on GPUs. The size of each engine stored in the cache of `TRTEngineOp` is about the same as the size of weights. TensorRT allocates memory through TensorFlow allocators, therefore, all TensorFlow memory configurations also apply to TensorRT. For example, if the TensorFlow session configuration `config.gpu_options.per_process_gpu_memory_fraction` is set to `0.3`, it means 30% of the GPU memory is allocated by TensorFlow to be used for all of its internal usage including TF-TRT and TensorRT. That means if TensorRT asks TensorFlow to allocate memory with the amount more than what is available to TensorFlow, then it will run out of memory.

On top of the memory used for weights and activations, certain TensorRT algorithms also require temporary workspace. The argument `max_workspace_size_bytes` limits the maximum size that TensorRT can use for the workspace. The default value is 1GB. If the value is too small, TensorRT will not be able to use certain algorithms that need that much workspace and that may lead to poor performance. The workspace is also allocated through TensorFlow allocators.

Although TensorRT is allowed to use algorithms that require at most `max_workspace_size_bytes` amount of workspace, but the maximum workspace requirement of all the TensorRT algorithms may still be smaller than `max_workspace_size_bytes` (meaning, TensorRT may not have any algorithm that needs that much workspace). In such cases, TensorRT only allocates the needed workspace instead of allocating how much the user specifies.

If you observe your inference running out of memory or you want to experiment with whether you can get better performance by using more memory, then try to increase `config.gpu_options.per_process_gpu_memory_fraction` and `max_workspace_size_bytes`. The memory usage highly depends on your model and it's hard to predict a suitable default for `max_workspace_size_bytes`.

2.9. INT8 Quantization

In order to use INT8 precision, the weights and activations of the model need to be quantized so that floating point values can be converted into integers using appropriate ranges. There are different calibration algorithms which can be used to perform the quantization after the model is trained. You can use such algorithms to compute the quantization ranges and then feed those custom ranges into TF-TRT.

TensorRT also provides a quantization algorithm which is integrated into TF-TRT and can be used without worrying about feeding ranges into TF-TRT. It's also possible to quantize the model during training (quantization-aware training) and then feed the ranges into TF-TRT. Since quantization-aware training requires many considerations, we recommend that you use the TensorRT calibration algorithm instead.

2.9.1. Post-Training Quantization Using TensorRT Calibration

TensorRT provides a calibration algorithm which can quantize the weights and activations after training. In the first step, TF-TRT runs the calibration algorithm which results in a calibration table, and in the second step the calibrated model is converted to a graph that is ready to be used by inference. The following code snippet shows how to use this method:

```
from tensorflow.python.compiler.tensorrt import trt_convert as trt
converter = trt.TrtGraphConverter(
 input_graph_def=frozen_graph,
 nodes_blacklist=['logits', 'classes'],
     precision_mode='INT8',
     use_calibration=True)
frozen_graph = converter.convert()
frozen_graph = converter.calibrate(
    fetch_names=['logits', 'classes'],
    num_runs=num_calib_inputs // batch_size,
    input_map_fn=input_map_fn)
```

The function `input_map_fn` should return a dictionary mapping input names (as strings) in the GraphDef to be calibrated to Tensor objects. The values of the named input tensors in the GraphDef to be calibrated will be re-mapped to the respective `Tensor` values during calibration. The following is an example of such a function:

```
dataset = tf.data.TFRecordDataset(data_files)
iterator = dataset.make_one_shot_iterator()
features = iterator.get_next()
def input_map_fn():
    return {'input:0': features}
```

The following code snippet shows how to extract the TensorRT calibration table after the calibration is done:

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

2.9.2. Post-Training Quantization Using Custom Quantization Ranges

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`

You should then call TF-TRT in the following way:

```
from tensorflow.python.compiler.tensorrt import trt_convert as trt
converter = trt.TrtGraphConverter(
 input_graph_def=frozen_graph,
 nodes_blacklist=['logits', 'classes'],
     precision_mode='INT8',
     use_calibration=False)
frozen_graph = converter.convert()
```

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]`.

Note: TensorRT only supports symmetric quantization ranges.

```
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 the TensorRT 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 TensorRT calibration procedure as described in Post-Training Quantization Using TensorRT Calibration (with `use_calibration=True`).

2.9.3. 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 quantizating 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 Post-Training Quantization Using Custom Quantization Ranges. 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 occuring). 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.

2.9.3.1. 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.

2.10. How 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.

```
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++.

Chapter 3. Supported Operators

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 following sections provide a list of supported operators in different versions of NVIDIA TensorFlow containers.

3.1. TensorFlow Container 19.11-19.12 (TensorFlow 1.15 And 2.0)

19.11/19.12-tf2 (TensorFlow 2.0)

- `AddN`
- `AddV2`
- `BatchMatMulV2`
- `FloorDiv`
- `FusedConv2DBiasActivation`

19.11/19.12-tf1 (TensorFlow 1.15)

- `AddN`
- `AddV2`
- `BatchMatMulV2`
- `Conv3D`
- `Conv3DBackpropInputV2`
- `FloorDiv`
- `FusedConv2DBiasActivation`

3.2. TensorFlow Container 19.07-19.10 (TensorFlow 1.14)

- `Acos`
- `Acosh`
- `ArgMax`
- `ArgMin`
- `Asin`
- `Asinh`
- `Atan`
- `Ceil`
- `ClipByValue`
- `CombinedNonMaxSuppression`
- `Conv2DBackpropInput`
- `Cos`
- `Cosh`
- `DepthToSpace`
- `Elu`
- `Floor`
- `FusedBatchNormV3`
- `GatherV2`
- `LeakyRelu`
- `Pack`
- `Selu`
- `Sin`
- `Sinh`
- `Softplus`
- `Softsign`
- `SpaceToDepth`
- `Split`
- `SquaredDifference`
- `StopGradient`

- `Tanh`
- `Transpose`
- `Unpack`

3.3. TensorFlow Container 19.02-19.06 (TensorFlow 1.13)

- `ExpandDims`
- `FakeQuantWithMinMaxArgs`
- `FakeQuantWithMinMaxVars`
- `Pow`
- `QuantizeAndDequantizeV2`
- `QuantizeAndDequantizeV3`
- `Reshape`
- `Sigmoid`
- `Slice`
- `Sqrt`
- `Square`
- `Squeeze`
- `StridedSlice`
- `Tanh`

3.4. TensorFlow Container 18.11-19.01 (TensorFlow 1.12)

- `Abs`
- `Add`
- `AvgPool`
- `BatchMatMul`
- `BiasAdd`
- `ConcatV2`
- `Const`
- `Conv2D`

- `DepthwiseConv2dNative`
- `Div`
- `Exp`
- `FusedBatchNorm`
- `FusedBatchNormV2`
- `Identity`
- `Log`
- `MatMul`
- `Max`
- `Maximum`
- `MaxPool`
- `Mean`
- `Min`
- `Minimum`
- `Mul`
- `Neg`
- `Pad`
- `Prod`
- `RealDiv`
- `Reciprocal`
- `Relu`
- `Relu6`
- `Rsqrt`
- `Snapshot`
- `Softmax`
- `Sub`
- `Sum`
- `TopKV2`

Chapter 4. Debugging And Profiling

4.1. Debugging

4.1.1. Verbose Logging

Increase the verbosity level in TensorFlow logs, for example:

```
TF_CPP_VMODULE=segment=2,convert_graph=2,convert_nodes=2,trt_engine=1,trt_logger=2 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:

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

TensorRT logs appear as part of TensorFlow logs. The verbosity level of TensorFlow logs affects the verbosity level of TensorRT logs.

4.1.2. 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.

4.1.2.1. Visualizing TF-TRT Graphs With TensorBoard

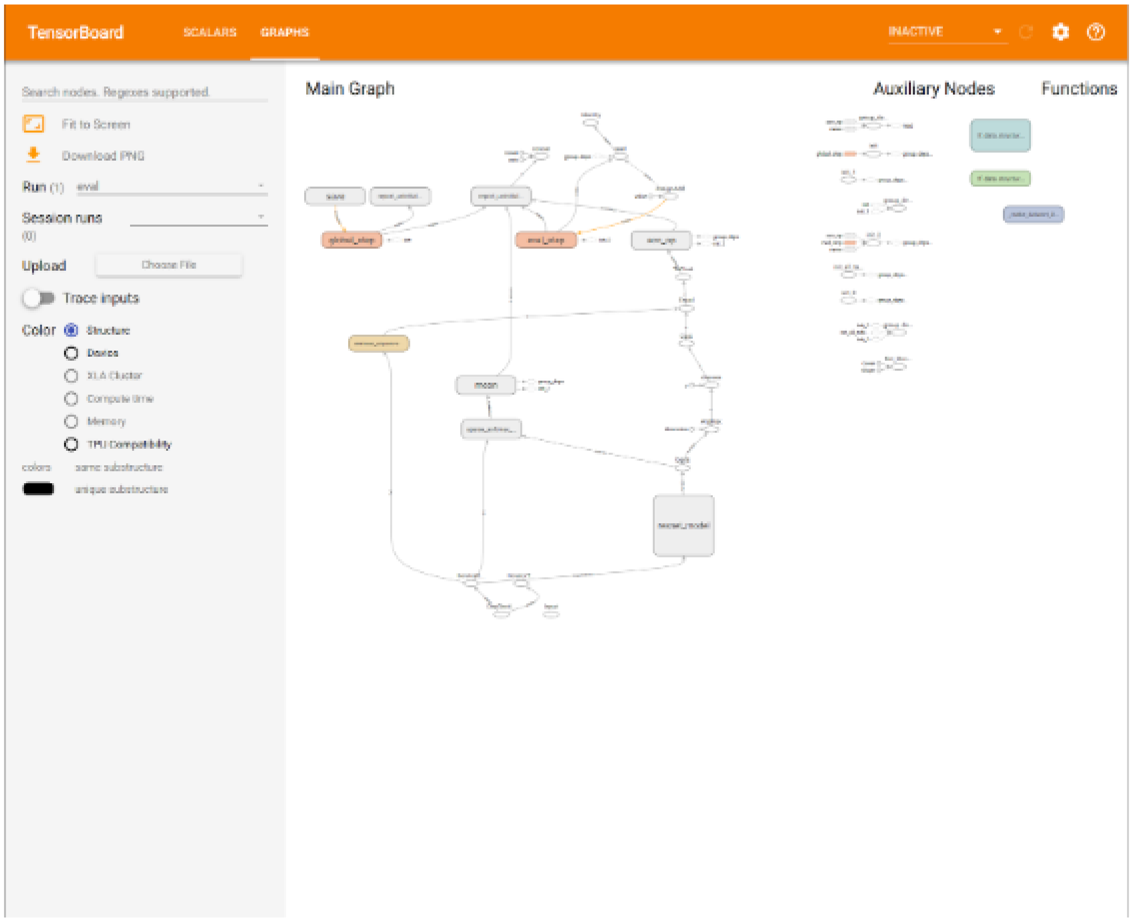
TensorBoard is a suite of visualization tools that make it easier to understand, debug, and optimize TensorFlow programs. You can use TensorBoard to display how the pre-trained TensorFlow graphs are optimized with TF-TRT. See TensorBoard documentation to learn how to generate TensorBoard event files to be used later in TensorBoard (for example, `tf.summary.FileWriter('./tensorboard_events', sess.graph)`). After you generate the event files, then you can launch TensorBoard using the following command:

```
tensorboard --logdir ./tensorboard_events
```

Note: 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 port]`. For example, `--publish 0.0.0.0:6006:6006` publishes TensorBoard's port `6006` to the host machine at port `6006` over all network interfaces (`0.0.0.0`). If you run a Docker container with this option, you can then access a running TensorBoard session at `http://[IP address of host machine]:6006`.

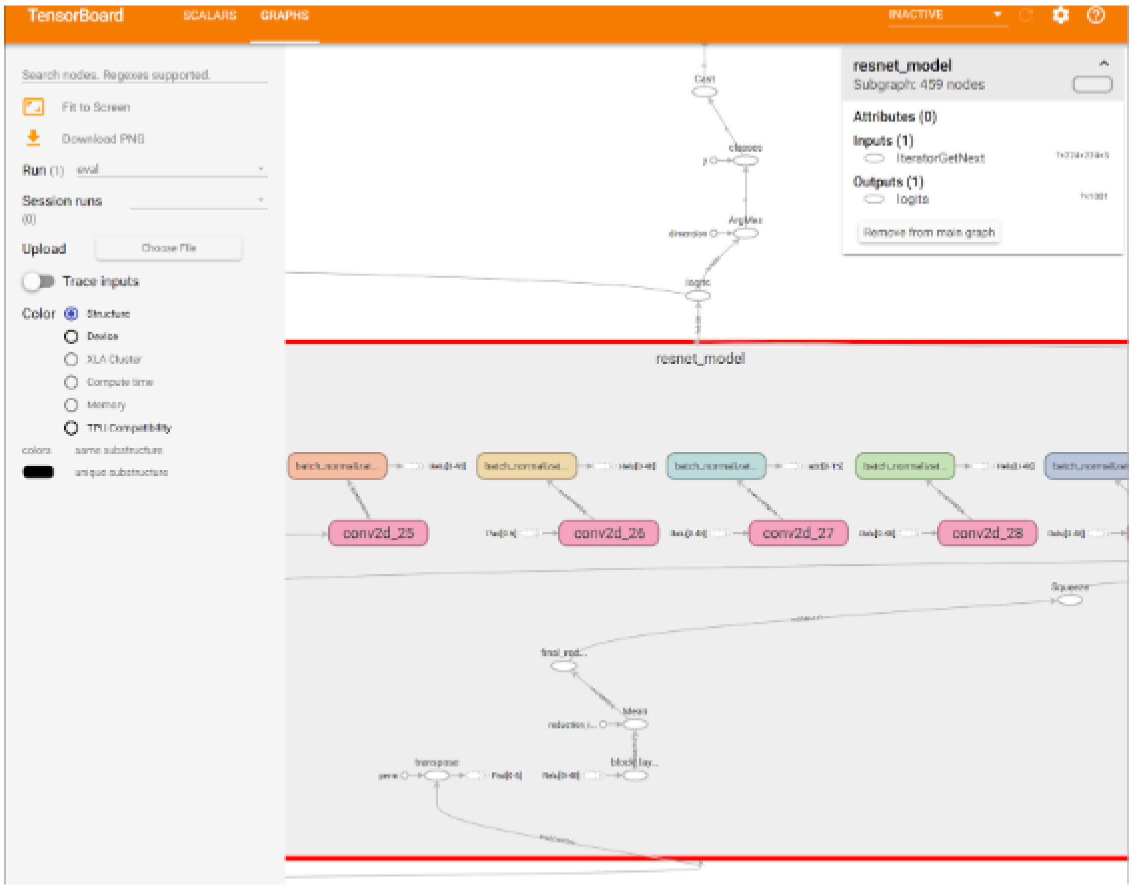
You can then navigate a web browser to port 6006 on the machine hosting the Docker container (`http://[IP address of host machine]:6006`), where you can see an interactive visualization of the graph. Figure 4 shows the graph of ResNet V1 50 model.

Figure 4. TensorBoard Visualization of Inference with Native TensorFlow Using ResNet V1 50.



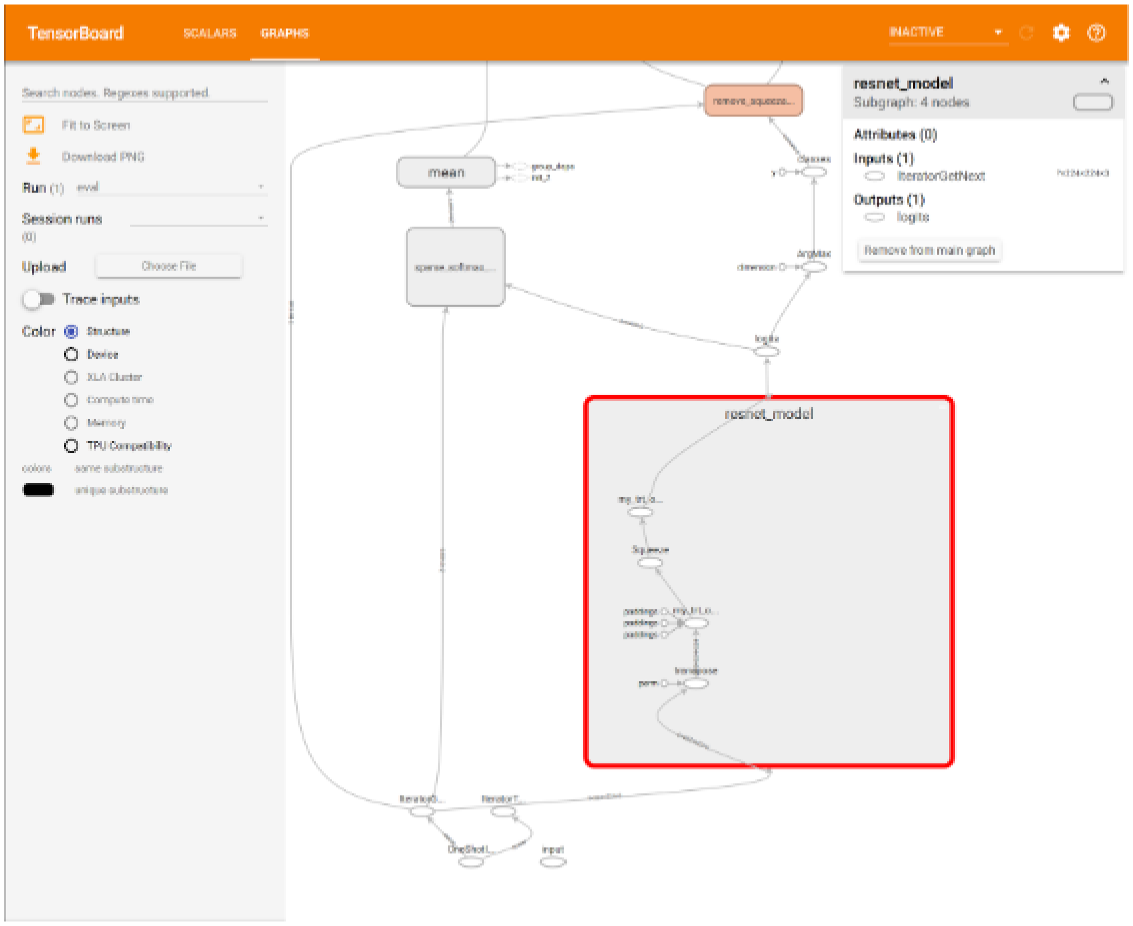
This visualization displays all the nodes including the nodes for running and evaluating inference, so there are additional nodes for loading and saving data and for evaluating inference. If you double click the `resnet_model` node, you can see the nodes specific to the ResNet V1 50 model, as shown in Figure 5.

Figure 5. 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. After you optimize the model with TF-TRT, many nodes in the graph are replaced by a single TensorRT node as shown in Figure 6, and thus reducing the total number of nodes to 4 from 459.

Figure 6. TensorBoard Visualization of ResNet V1 50 Converted to a TensorRT Graph.



4.2. Profiling Tools

There are many tools available for profiling a TF-TRT application, ranging from command-line profiler to GUI tools, including nvprof, NVIDIA NSIGHT Systems, DLProf and TensorFlow Profiler.

nvprof

The easiest profiler to use is nvprof, a command-line light-weight profiler which presents an overview of the GPU kernels and memory copies in your application. You can use nvprof as below:

```
nvprof python run_inference.py
```

NVTX 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.

NVIDIA Nsight Systems

NVIDIA has released a system-wide performance analysis tool to help users investigate bottlenecks, pursue optimizations with higher probability of performance gains. Refer to the blog post High Performance Inference with TensorRT Integration for more information about how to use Nsight Systems with TF-TRT.

TensorFlow Profiler

TensorFlow Profiler is another tool that can be used for visualizing kernel timing information by adding additional parameters to the Python script that defines or runs the graph. For example the argument `run_metadata` can be used to enable the profiler:

```
sess.run(res, options=options, run_metadata=run_metadata)
```

After execution, a `JSON` file with profiled data is generated in Chrome trace format and can be viewed by the Chrome browser.

4.3. TensorFlow Ops Used In A TensorRT Subgraph

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.

4.4. 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.

Note: 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`.

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:

Note: `frozen_graph` is 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.

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`.

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.

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 ResNet 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.

`minimum_segment_size`

Subgraphs with fewer nodes than `minimum_segment_size` are not converted to TensorRT. If you experience a functionality (for example, crash) or performance issue with certain subgraphs, you can prevent their conversion by increasing `minimum_segment_size` to a number larger than the size of that subgraph.

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 TensorRT 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.

See the blog post High Performance Inference with TensorRT Integration for more information about how to achieve better performance with TF-TRT.

6.1. Verified Models

We have verified that the following image classification models work with TF-TRT and achieve good performance. Refer to the release notes 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 1. Verified Models

	Native TensorFlow FP32	TF-TRT FP32	TF-TRT FP16	TF-TRT INT8	
	Volta and Turing	Volta and Turing	Volta and Turing	Volta	Turing
MobileNet v1	71.01	71.01	70.99	69.49	69.49
MobileNet v2	74.08	74.08	74.07	73.96	73.96
NASNet - Large	82.72	82.71	82.70	Work in progress	82.66
NASNet - Mobile	73.97	73.85	73.87	73.19	73.25
ResNet-50 v1.5[1]	76.51	76.51	76.48	76.23	76.23
ResNet-50 v2	76.43	76.37	76.4	76.3	76.3
VGG16	70.89	70.89	70.91	70.84	70.78
VGG19	71.01	71.01	71.01	70.82	70.90
Inception v3	77.99	77.99	77.97	77.92	77.93
Inception v4	80.19	80.19	80.19	80.14	80.08

6.2. 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.

Note:

- 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.

[1] ResNet-50 v1.5 from the official TensorFlow model repository, sometimes labeled as ResNet-50 v1. For more details, see ResNet in TensorFlow.

Chapter 7. Samples

For specific tutorials and samples, see `nvidia-examples/tensorrt` inside the TensorFlow container or GitHub: TF-TRT.

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 and JPEG files. 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 8. Troubleshooting

8.1. 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
- High performance inference with TensorRT Integration.

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/.

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