# Accelerating Inference In TensorFlow With TensorRT (TF-TRT)

User Guide

SWE-SWDOCTFT-001-INTG _v002 | May 2024

# Table of Contents

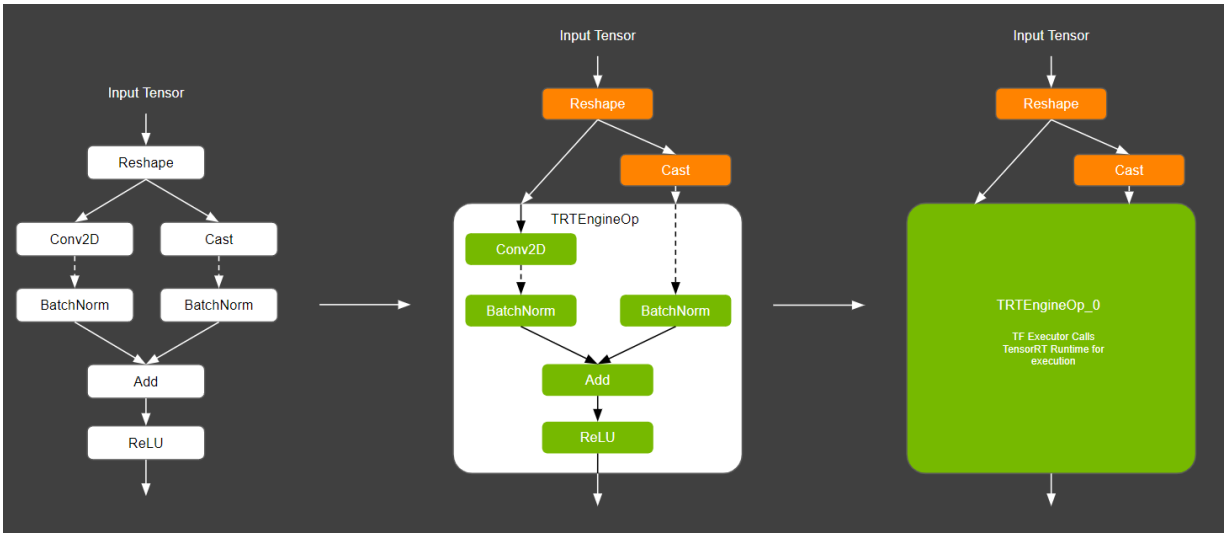
# Chapter 1. Introduction

TF-TRT automatically partitions a TensorFlow graph into subgraphs based on compatibility with TensorRT. These compatible subgraphs are optimized and executed by TensorRT, relegating the execution of the rest of the graph to native TensorFlow. This allows the use of TensorFlow's rich feature set, while optimizing the graph wherever possible with TensorRT, providing both flexibility and performance. To learn more about the optimizations provided by TensorRT, please visit TensorRT's page.

The most important benefit of using TF-TRT is that a user can create and test their model on TensorFlow, and leverage the performance acceleration provided by TensorRT, with just a few additional lines of code, without having to develop in C++ using TensorRT directly. This allows for a seamless workflow from model definition, to training, to deployment on NVIDIA devices.

For expert users requiring complete control of TensorRT's capabilities, exporting the TensorFlow model to ONNX and directly using TensorRT is recommended.

TF-TRT ingests, via its Python or C++ APIs, a TensorFlow SavedModel created from a trained TensorFlow model (see Build and load a SavedModel). Supported subgraphs are replaced with a TensorRT optimized node (called `TRTEngineOp`), producing a new TensorFlow graph that will has both TensorFlow and TensorRT components, as shown in the following figure:



In the process of converting subgraphs to `TRTEngineOp`s, TensorRT performs several important transformations and optimizations to the neural network graph, including constant folding, pruning unnecessary graph nodes, layer fusion, and more. For the full list of optimizations, see TensorRT Documentation.

The more operations converted to a single TensorRT engine, the larger the potential benefit gained from using TensorRT. For that reason, TF-TRT aims to convert the maximum number of operations in the TensorFlow compute graph while limiting the total number of TensorRT engines, maximizing performance. Based on the operations in your graph, it's possible that the final graph might have more than one TensorRT node. Please refer to TensorRT's documentation to understand more about specific graph optimizations.

If conversion of a segment to a TensorRT engine fails or executing the generated TensorRT engine fails, then TFTRT will try to execute the native TensorFlow segment. This is called native segment fallback.

# Chapter 2. Downloading and Installing TF-TRT

NVIDIA NGC containers for TensorFlow are built and tested with TF-TRT support enabled, allowing for out-of-the-box usage in the container, without the hassle of having to set up a custom environment. To learn how to pull the container check out the NGC User Guide, however they can be download with a simple docker command:

```
docker pull nvcr.io/nvidia/tensorflow:<yy.mm>-tf2-py3
# eg: The TF2 container from January 2022 (22.01) would be
docker pull nvcr.io/nvidia/tensorflow:22.01-tf2-py3
```

**Note:** If you are a first-time NGC User, you will need to make an account and log in with your (free) API key. Here are the instructions for running a container, and access to NGC.

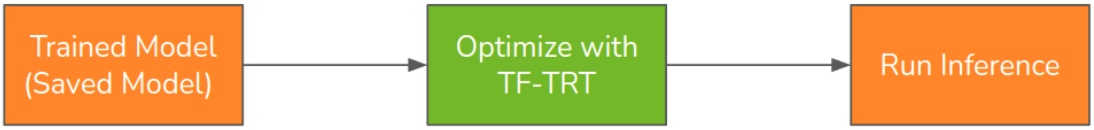
TF-TRT is also available in the TensorFlow repository and can be used with any TensorFlow installation accompanied by TensorRT. This means when you install `tensorflow-gpu`, it includes TF-TRT that can be used directly once TensorRT is installed.

To build from scratch, please 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. Some models may require additional packages to execute properly.

# Chapter 3. Quickstart Guide

In order to optimize a Tensorflow model using TF-TRT, the tensorflow model needs to be saved in the `SavedModel` format. Conversion with TF-TRT is only one extra step to optimize a model for inference on NVIDIA devices.




The TF-TRT workflow is simple. The basic steps are as follows:

1. Convert trained SavedModel using the TF-TRT converter.
2. Build the TRT engines (optional, running the model will automatically build it).
3. Save the converted model for future use (optional).
4. Done! Run an inference using the converted model as normal.

The following is a complete Python example starting from model definition and training to TF-TRT conversion and inference.

1. Create your model using Tensorflow. For illustration purposes, we'll be using a Keras sequential model (from here).

```
import tensorflow as tf
from tensorflow import keras

# Define a simple sequential model
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10)
])

model.compile(optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy'])
```

2. Get the dataset and preprocess it as needed.

```
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

```
x_train = tf.cast(x_train, dtype=tf.float32)
y_train = tf.cast(y_train, dtype=tf.float32)
x_test = tf.cast(x_test, dtype=tf.float32)
y_test = tf.cast(y_test, dtype=tf.float32)
```

3. Train the model and evaluate accuracy as required, as shown here.

```
# Train the model
model.fit(x_train, y_train, epochs=5)

# Evaluate your model accuracy
model.evaluate(x_test,  y_test, verbose=2)
```

4. Save the model in the saved_model format.

```
# Save model in the saved_model format
SAVED_MODEL_DIR="./models/native_saved_model"
model.save(SAVED_MODEL_DIR)
```

5. Convert the model using the TF-TRT converter.

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

# Instantiate the TF-TRT converter
converter = trt.TrtGraphConverterV2(
   input_saved_model_dir=SAVED_MODEL_DIR,
   precision_mode=trt.TrtPrecisionMode.FP32
)

# Convert the model into TRT compatible segments
trt_func = converter.convert()
converter.summary()
```

6. Build the TRT engines.

```
MAX_BATCH_SIZE=128
def input_fn():
   batch_size = MAX_BATCH_SIZE
   x = x_test[0:batch_size, :]
   yield [x]

converter.build(input_fn=input_fn)
```

7. Save the converted model for future use.

```
OUTPUT_SAVED_MODEL_DIR="./models/tftrt_saved_model"
converter.save(output_saved_model_dir=OUTPUT_SAVED_MODEL_DIR)
```

8. Run an inference using the converted model.

```
# Get batches of test data and run inference through them
infer_batch_size = MAX_BATCH_SIZE // 2
for i in range(10):
   print(f"Step: {i}")

   start_idx = i * infer_batch_size
   end_idx   = (i + 1) * infer_batch_size
   x = x_test[start_idx:end_idx, :]

   trt_func(x)
```

For more examples, refer to TF-TRT Examples in the TensorRT repository.

# Chapter 4. Key Capabilities

TF-TRT leverages many of TensorRT's capabilities to accelerate inference. Some of these capabilities are:

- Mixed precision execution (FP32, FP16, and INT8)
- INT8 quantization
- Dynamic Batch and Input shapes

This section will give an overview of the above capabilities & provide usage / best practice examples on how to utilize them.

## 4.1. Supported Precision Levels

TensorRT can convert tensors and weights to lower precisions for faster inference during the optimization. The argument `precision_mode` sets the precision mode; which can be one of `FP32`, `FP16`, or `INT8`. Precisions lower than FP32, such as FP16 and INT8, can extract higher performance out of TensorRT engines. The FP16 mode uses Tensor Cores or half precision hardware instructions, if possible. The INT8 precision mode uses integer hardware instructions.

Users are encouraged to try the reduced precision modes such as FP16 and INT8. FP16 will improve performance without substantial accuracy loss; models trained with AMP should have no loss. INT8 precision mode will have the best performance, however, the quantization error induced by INT8 quantization may introduce an accuracy drop in some models. See the Quantization section to understand this effect in more detail and how to mitigate it.

Depending on the application requirements (performance, memory consumption, accuracy), precision level should be selected. Regardless of choice, model validation is always recommended after conversion to TensorRT.

Users should note that selecting a lower precision mode does not mean that the whole network will run in that precision. TensorRT selects the fastest layer in the chosen precision or higher for best performance (see reduced precision).

## 4.2. Quantization

Quantization in Deep Learning refers to transforming the deep learning model's parameters to perform computation at lower precision. This is a popular optimization which helps reduce the size of deep learning models, thereby speeding up inferences and reducing power consumption. This is useful in all deployments, but can be essential for deployment on embedded devices with lower computational power such as the NVIDIA Jetson.

To illustrate quantization with an example, imagine multiplying 3.999x2.999 and 4x3. The latter (integer quantized) operation is considerably faster to perform than the former. This is the speedup one strives to achieve by quantizing the numbers to a lower precision. Simply put, quantization is a process of mapping input values from a large range and fine granularity to output values in a smaller range and coarser granularity, thereby reducing precision.

In the context of deep learning, we often train deep learning models using floating-point 32-bit representation (FP32) as we can take advantage of a wider range of numbers. During model quantization, the model data–network parameters and activations–are converted from this floating point representation to a lower precision representation, typically using 8-bit integers (INT8). Unfortunately, this approximation may result in a lower model accuracy.

The main quantization method used in TF-TRT is Post-Training Quantization (PTQ). As the name suggests, Post Training Quantization is a technique used on a previously trained model to reduce the size of the model and gain throughput benefits while mitigating the cost to the model accuracy.

Since small rounding errors can propagate through the network and become increasingly impactful for the model accuracy, different quantization techniques, like quantization-aware training (QAT), have been developed to mitigate this effect. QAT is currently experimental in TF-TRT.

### 4.2.1. Post-Training Quantization

Post-Training Quantization (aka. PTQ) is called INT8-*calibration* in the context of TensorRT.

During the calibration stage, TensorRT uses the supplied input "calibration" data to estimate the best scale and bias values for each tensor of the network given its dynamic range and value distribution. TF-TRT stores this information collected in the converted model.

The TF-TRT workflow for using PTQ is fairly straightforward. A user needs to do the following:

- During converter instantiation:
  - Set the precision mode as INT8.
  - Set the `use_calibration` flag to True.
- During the conversion process, the user needs to pass in a representative input dataloader for calibration. It is important that the calibration input data represent the range of inputs that the model is expected to operate on for calibration to produce meaningful scale factors for activations; the more data, the more accurate the

quantization. For example, the test data set (or some subset of it) is often a good data source.

Users should note that the calibration data is always expected to have a single shape.

The following Python example demonstrates calibration:

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

# Instantiate the TF-TRT converter
converter = trt.TrtGraphConverterV2(
   input_saved_model_dir=SAVED_MODEL_DIR,
   precision_mode=trt.TrtPrecisionMode.INT8,
   use_calibration=True
)

# Use data from the test/validation set to perform INT8 calibration
BATCH_SIZE=32
NUM_CALIB_BATCHES=10
def calibration_input_fn():
   for i in range(NUM_CALIB_BATCHES):
       start_idx = i * BATCH_SIZE
       end_idx = (i + 1) * BATCH_SIZE
       x = x_test[start_idx:end_idx, :]
       yield [x]

# Convert the model with valid calibration data
func = converter.convert(calibration_input_fn=calibration_input_fn)

# Input for dynamic shapes profile generation
MAX_BATCH_SIZE=128
def input_fn():
   batch_size = MAX_BATCH_SIZE
   x = x_test[0:batch_size, :]
   yield [x]

# Build the engine
converter.build(input_fn=input_fn)

OUTPUT_SAVED_MODEL_DIR="./models/tftrt_saved_model"
converter.save(output_saved_model_dir=OUTPUT_SAVED_MODEL_DIR)

converter.summary()

# Run some inferences!
for step in range(10):
   start_idx = step * BATCH_SIZE
   end_idx   = (step + 1) * BATCH_SIZE

   print(f"Step: {step}")
   x = x_test[start_idx:end_idx, :]
   func(x)
```

## 4.3. Dynamic Shapes

A TensorFlow model can have input tensors with fixed or dynamic shapes. Before training, we typically set most of the input dimensions to a fixed value. For example, a model that takes a batch of 8 input images with 224x224 resolution and three color channels could have an input tensor with a fixed shape of [8, 224, 224, 3]. Depending on the model architecture, we can leave some of the input dimensions dynamic to allow inference with a wider range of input shapes: Typical examples of dynamic input shapes are:

- Batch size – for example for an image classification model, the network input tensor can be [?, 224, 224, 3], where the batch size is unknown during model definition and is allowed to take different values during runtime.
- Image size for fully convolutional networks [8, ?, ?, 3]
- Sequence length of transformer models. For example a BERT encoder has input tensors with shape [N, S], where N is the batch size and S is the sequence length, and both of these dimensions can be dynamic.

TF-TRT supports models with dynamic shape via user-provided information about the range of input shapes that the converted model should support.

By default TF-TRT allows dynamic batch size. The maximum batch size (N) is set as the batch size that was used to build the engines for the converted model. Such a model would support any batch size between [1..N]. (also called implicit batch mode). If we try to infer the model with larger batch size, then TF-TRT will build another engine to do so. This has significant performance impacts as engine building is expensive. The `allow_build_at_runtime` and `max_cached engines` conversion parameters control TF-TRT's runtime engine building behavior.

To support dynamic input dimensions other than the batch dimension, we need to enable dynamic shape mode by passing `use_dynamic_shape=True` argument to the converter. The dynamic shape mode in TF-TRT utilizes TensorRT's dynamic shape feature to improve the conversion rate of networks and handle networks with unknown input shapes efficiently. An increased conversion rate means that more of the network can be run in TensorRT. This improves the performance of such networks when used with TF-TRT.

Apart from enabling the `use_dynamic_shape` flag, TF-TRT needs to be provided information about the range of shapes that are expected during inference, as in the following Example.

```
# Instantiate the TF-TRT converter
# Instantiate the TF-TRT converter
PROFILE_STRATEGY="Optimal"
converter = trt.TrtGraphConverterV2(
   input_saved_model_dir=bert_saved_model_path,
   precision_mode=trt.TrtPrecisionMode.FP32,
   use_dynamic_shape=True,
   dynamic_shape_profile_strategy=PROFILE_STRATEGY)

# Convert the model to TF-TRT
converter.convert()

VOCAB_SIZE = 30522  # Model specific, look in the model README.
# Build engines for input sequence lengths of 128, and 384.
input_shapes = [[(1, 128), (1, 128), (1, 128)],
                [(1, 384), (1, 384), (1, 384)]]
def input_fn():
   for shapes in input_shapes:
       # return a list of input tensors
       yield [tf.convert_to_tensor(
         np.random.randint(low=0, high=VOCAB_SIZE, size=x,dtype=np.int32))
         for x in shapes]

converter.build(input_fn)
```

Before saving the converted model, it is built to handle a certain range of input parameters, by using the input_fn. Unlike calibration inputs, these inputs do not need

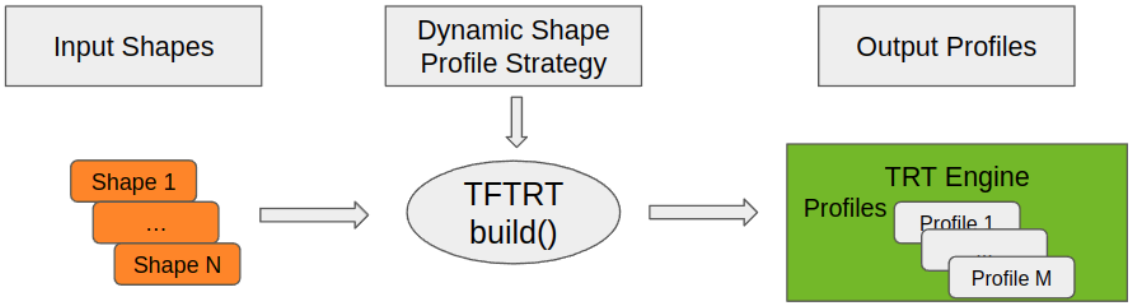
to represent real input data, for most of the models only the input shapes matter; data-dependent shapes are the exception to this.

The example above illustrates a BERT like model, which has three input tensors. Our input_fn defines two different input shapes one with sequence length 128 and one with sequence length 384.

Dynamic inputs can be further specified with the `dynamic_shape_profile_strategy` argument. This parameter selects the strategy for defining optimization profiles for TensorRT (where "optimization profile" is TensorRT's terminology for describing input shape information). The following are options for optimization profiles:

- **Range**: create one profile that works for inputs with dimension values in the range of [min_dims, max_dims] where min_dims and max_dims are derived from the provided inputs.
- **Optimal**: create one profile for each input. The profile only works for inputs with the same dimensions as the input it is created for. The GPU engine will be run with optimal performance with such inputs.
- **Range+Optimal**: create the profiles for both Range and Optimal.
- **ImplicitBatchModeCompatible**: create the profiles that will produce the same GPU engines as the implicit_batch_mode would produce.

The following image and table illustrate how the profile strategy influences the range of shapes accepted by the converted model.




| Input Shapes | Dynamic Shape Profile Strategy | Output Profiles | Use Case |
|---|---|---|---|
| [8, 128], [4, 384] | Range | [4-8, 128-384] | Handles a range of inputs for both dimensions |
| [8, 128], [4, 384] | Optimal | [8, 128], [4, 384] | Best performance for concrete input shapes |
| [8, 128], [4, 384] | Range + Optimal | [8, 128], [4, 384],<br>[4-8, 128-384] | Best performance for the concrete inputs, handles any input in the range |
| [8, 128], [4, 384] | ImplicitBatchModeCompatible | [1-8, 128], [1-4, 384] | Flexible batch size for each sequence length |

If only a small number of concrete input shapes are expected, then it is recommended to use the “Optimal” strategy.

If `build()` is not called, then the TensoRT engine creation will take place when the converted model is first inferred. The input shape used during this inference will set the TensorRT profile strategy to the default strategy, Range, with parameters min_dims=max_dims.

**Note:** Users can set “`use_dynamic_shapes=True`” for graphs that have static inputs, and it often results in improved conversion rate.

# 4.4. Simple Examples

## 4.4.1. A Python Example

The following is a simple Python example demonstrating conversion of a BERT model with random inputs.

```
# Prerequisite: Install the python module below before running this example.
# pip install -q tf-models-official

import tensorflow as tf
import tensorflow_hub as hub

tfhub_handle_encoder = 'https://tfhub.dev/tensorflow/
bert_en_uncased_L-12_H-768_A-12/3'
bert_saved_model_path = './models/bert_base'

bert_model = hub.load(tfhub_handle_encoder)
tf.saved_model.save(bert_model, bert_saved_model_path)

import numpy as np
from tensorflow.python.saved_model import signature_constants
from tensorflow.python.saved_model import tag_constants
from tensorflow.python.compiler.tensorrt import trt_convert as trt

# Instantiate the TF-TRT converter
PROFILE_STRATEGY="Optimal"
converter = trt.TrtGraphConverterV2(
   input_saved_model_dir=bert_saved_model_path,
   precision_mode=trt.TrtPrecisionMode.FP32,
   use_dynamic_shape=True,
   dynamic_shape_profile_strategy=PROFILE_STRATEGY)

# Convert the model to TF-TRT
converter.convert()

VOCAB_SIZE = 30522  # Model specific, look in the model README.
# Build engines for input sequence lengths of 128, and 384.
input_shapes = [[(1, 128), (1, 128), (1, 128)],
                [(1, 384), (1, 384), (1, 384)]]
def input_fn():
   for shapes in input_shapes:
       # return a list of input tensors
       yield [tf.convert_to_tensor(
         np.random.randint(low=0, high=VOCAB_SIZE, size=x,dtype=np.int32))
         for x in shapes]

converter.build(input_fn)
```

```
# Save the converted model
bert_trt_path = "./models/tftrt_bert_base"
converter.save(bert_trt_path)
converter.summary()

# Some helper functions
def get_func_from_saved_model(saved_model_dir):
   saved_model_loaded = tf.saved_model.load(
       saved_model_dir, tags=[tag_constants.SERVING])
   graph_func = saved_model_loaded.signatures[
       signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY]
   return graph_func, saved_model_loaded


def get_random_input(batch_size, seq_length):
   # Generate random input data
   mask = tf.convert_to_tensor(np.ones((batch_size, seq_length), dtype=np.int32))
   type_id = tf.convert_to_tensor(np.zeros((batch_size, seq_length),
 dtype=np.int32))
   word_id = tf.convert_to_tensor(
     np.random.randint(0, VOCAB_SIZE, size=[batch_size, seq_length],
 dtype=np.int32))
   return {'input_mask':mask, 'input_type_ids': type_id, 'input_word_ids':word_id}


# Get a random input tensor
input_tensor = get_random_input(1, 128)

# Specify the output tensor interested in. This output is the 'classifier'
result_key = 'bert_encoder_1'
trt_func, _ = get_func_from_saved_model(bert_trt_path)

## Let's run some inferences!
for i in range(0, 10):
   print(f"Step: {i}")
   preds = trt_func(**input_tensor)
   result = preds[result_key]
```

## 4.4.2. A C++ Example

Tensorflow supports executing models in C++ including TF-TRT converted models. The C++ API for the TF-TRT converter is presently experimental.

First convert the model via the TF-TRT Python APIs.

The C++ workflow for loading and running models is explained below for a model with synthetic data.

1. Initialize the global states required by Tensorflow
2. Load the saved model and initialize the TF session
3. Set up inputs
4. Set up outputs
5. Run the inference
6. Release resources

An example showing the above steps:

1. Initialize the global states required by Tensorflow and the TF session.

```
// We need to call this to set up global state for TensorFlow.
tensorflow::port::InitMain(argv[0], &argc, &argv);
```

```
 if (argc > 1) {
   LOG(ERROR) << "Unknown argument " << argv[1] << "\n" << usage;
   return -1;
 }
```

2. Load the saved model and initialize the TF session.

```
// Some helper functions
// Returns info for nodes listed in the signature definition.
std::vector<tensorflow::TensorInfo> GetNodeInfo(
   const google::protobuf::Map<string, tensorflow::TensorInfo>& signature) {
 std::vector<tensorflow::TensorInfo> info;
 for (const auto& item : signature) {
   info.push_back(item.second);
 }
 return info;
}

// Load the `SavedModel` located at `model_dir`.
Status LoadModel(const string& model_dir, const string& signature_key,
                 tensorflow::SavedModelBundle* bundle,
                 std::vector<tensorflow::TensorInfo>* input_info,
                 std::vector<tensorflow::TensorInfo>* output_info) {
 tensorflow::RunOptions run_options;
 tensorflow::SessionOptions sess_options;

 tensorflow::OptimizerOptions* optimizer_options =
     sess_options.config.mutable_graph_options()->mutable_optimizer_options();
 optimizer_options->set_opt_level(tensorflow::OptimizerOptions::L0);
 optimizer_options->set_global_jit_level(tensorflow::OptimizerOptions::OFF);

 sess_options.config.mutable_gpu_options()->force_gpu_compatible();
 TF_RETURN_IF_ERROR(tensorflow::LoadSavedModel(sess_options, run_options,
                                               model_dir, {"serve"}, bundle));

 // Get input and output names
 auto signature_map = bundle->GetSignatures();
 const tensorflow::SignatureDef& signature = signature_map[signature_key];
 *input_info = GetNodeInfo(signature.inputs());
 *output_info = GetNodeInfo(signature.outputs());

 return Status::OK();
}
```

3. Set up inputs. Here we are using synthetic data and placing it on the device ahead of inference.

```
// Create random inputs matching `input_info`
Status SetupInputs(int32_t batch_size,
                   int32_t input_size,
                   std::vector<tensorflow::TensorInfo>& input_info,
                   std::vector<std::pair<std::string, tensorflow::Tensor>>*
 inputs) {

 //std::vector<std::pair<std::string, tensorflow::Tensor>> input_tensors;
 for (auto& info : input_info) {
     // Set input batch size
     auto* shape = info.mutable_tensor_shape();
     shape->mutable_dim(0)->set_size(batch_size);

     // Set dynamic dims to static size
     for (size_t i = 1; i < shape->dim_size(); i++) {
         auto* dim = shape->mutable_dim(i);
         if (dim->size() < 0) {
           dim->set_size(input_size);
         }
     }

     // Allocate memory and fill host tensor
```

```
    Tensor input_tensor(info.dtype(), *shape);
    std::fill_n((uint8_t*)input_tensor.data(), input_tensor.AllocatedBytes(),
1);

    inputs->push_back({info.name(), input_tensor});
}

  return Status::OK();
}
```

4. Set up outputs.

```
// Get output tensor names based on `output_info`.
Status SetupOutputs(std::vector<tensorflow::TensorInfo>& output_info,
                    std::vector<string>* output_names,
                    std::vector<Tensor>* outputs) {
 for (auto& info : output_info) {
   output_names->push_back(info.name());
   outputs->push_back({});
 }
 return Status::OK();
}
```

5. Run the inference.

```
// Setup inputs
std::vector<std::pair<std::string, tensorflow::Tensor>> inputs;
TFTRT_ENSURE_OK(SetupInputs(batch_size, input_size, input_info, &inputs));

// Setup outputs
std::vector<string> output_names;
std::vector<Tensor> outputs;
TFTRT_ENSURE_OK(SetupOutputs(output_info, &output_names, &outputs));

int num_iterations = 10;
for (int i = 0; i < num_iterations; i++) {
  LOG(INFO) << "Step: " << i;

  TFTRT_ENSURE_OK(
      bundle.session->Run(inputs, output_names, {}, &outputs));
}
```

Here is a program that ties all of the above functions together:

```
int main(int argc, char* argv[]) {
 // Parse arguments
 string model_path = "/path/to/model/";
 string signature_key = "serving_default";
 int32_t batch_size = 64;
 int32_t input_size = 128;
 std::vector<Flag> flag_list = {
     Flag("model_path", &model_path, "graph to be executed"),
     Flag("signature_key", &signature_key, "the serving signature to use"),
     Flag("batch_size", &batch_size, "batch size to use for inference"),
     Flag("input_size", &input_size, "shape to use for -1 input dims"),
 };
 string usage = tensorflow::Flags::Usage(argv[0], flag_list);
 const bool parse_result = tensorflow::Flags::Parse(&argc, argv, flag_list);
 if (!parse_result) {
   LOG(ERROR) << usage;
   return -1;
 }

 // We need to call this to set up global state for TensorFlow.
 tensorflow::port::InitMain(argv[0], &argc, &argv);
 if (argc > 1) {
   LOG(ERROR) << "Unknown argument " << argv[1] << "\n" << usage;
   return -1;
 }
```

```
// Setup TF session
tensorflow::SavedModelBundle bundle;
std::vector<tensorflow::TensorInfo> input_info;
std::vector<tensorflow::TensorInfo> output_info;
TFTRT_ENSURE_OK(
    LoadModel(model_path, signature_key, &bundle, &input_info, &output_info));

// Setup inputs
std::vector<std::pair<std::string, tensorflow::Tensor>> inputs;
TFTRT_ENSURE_OK(SetupInputs(batch_size, input_size, input_info, &inputs));

// Setup outputs
std::vector<string> output_names;
std::vector<Tensor> outputs;
TFTRT_ENSURE_OK(SetupOutputs(output_info, &output_names, &outputs));

int num_iterations = 10;
for (int i = 0; i < num_iterations; i++) {
  LOG(INFO) << "Step: " << i;

  TFTRT_ENSURE_OK(
      bundle.session->Run(inputs, output_names, {}, &outputs));
}

return 0;
}
```

More C++ examples working on real data are presented here.

# Chapter 5. Deploying TF-TRT

Models accelerated by TensorFlow-TensorRT can be served with NVIDIA Triton Inference Server, which is an open-source inference serving software that helps standardize model deployment and execution and delivers fast and scalable AI in production.

Explore this example and the NVIDIA Triton Inference Server Github Repository for more information!

# Chapter 6. Debugging and Troubleshooting

## 6.1. Minimum Segment Size

TensorFlow subgraphs that are optimized by TensorRT include a certain number of operators. If the number of operators included in the subgraph is too 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.

Using the `minimum_segment_size` parameter allows you to mitigate and avoid potential overheads introduced by those small TensorRT engines, and also can get around any possible errors that arise from those engines.

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

The optimal value for `minimum_segment_size` is model specific. To find the best value, look at the converter summary (see section), observe the number of engines and their sizes, and set a value that maximizes the number of converted nodes while keeping the number of subgraphs (TRTEngineOps) limited.

A good rule is to start with `minimum_segment_size=3` and only update if TF-TRT generates too many engines: this number will vary, but a maximum of 5-10 engines is a good heuristic. Incrementally update the value until the number of engines comes down to a reasonable value at no performance cost. A good heuristic to use to tune the number of engines is the number of nodes in an engine. If the number of nodes in the segment is too small compared to other segments, one could increase the minimum segment size to a value larger than that segment size to avoid converting that segment.

## 6.2. Max Workspace Size

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 workspace is also allocated through TensorFlow allocators. If the value is too small, TensorRT will not be able to use certain algorithms that need a large workspace, leading to engine build failures and native segment fallback.

Although TensorRT is allowed to use algorithms that require at most `max_workspace_size_bytes` amount of workspace, 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.

TensorRT introduced lazy allocation in TensorRT 8.4, and can use up to the total global memory size of a given device if needed. It is therefore recommended to use as large a value as needed by the model, restricting this size only if multiple engines are to be built on a single device.

## 6.3. Conversion Reports

You can use `converter.summary()` to see a list of converted engines along with related information such as the concrete input shapes and types. If you care about non-converted engines, use Conversion Report instead, which shows the ops that did not convert as well as how many instances of such ops exist. This section discusses each of these diagnostic tools in detail.

### 6.3.1. converter.summary()

A nifty API to understand how the TF-TRT conversion process went is the summary API. It can be invoked as shown:

```
converter.convert()
converter.summary()
```

When invoked after conversion, this API prints the summary of the conversion. It includes information such as the name of the engine, the number of nodes per engine, the input and output data types, along with the input shape of each TRTEngineOp. The detailed report also includes the number and types of ops in the engine. An example summary report is shown below, which shows not only the number of engines and their constituent ops, but also the total number of ops that were converted, to give an idea of completeness.

```
TRTEngineOP Name                  Device            # Nodes  # Inputs      # Outputs
     Input DTypes           Output Dtypes           Input Shapes           Output
 Shapes
================================================================================================

---------------------------------

TRTEngineOp_000_000               device:GPU:0     9        8             8
     ['float32', 'float ... ['float32', 'float ... [[-1, -1, 3], [-1, ... [[1, -1,
 -1, 3], [ ...
```

```
   - Const: 1x
   - ExpandDims: 8x

---------------------------------

TRTEngineOp_000_001               device:GPU:0     919      8             2
    ['float32', 'float ... ['float32', 'float32'] [[1, 640, 640, 3], ... [[8, 51150,
 90], [ ...

   - AddV2: 25x
   - BiasAdd: 13x
   - Cast: 11x
   - ConcatV2: 5x
   - Const: 489x
   - Conv2D: 110x
   - Exp: 2x
   - ExpandDims: 2x
   - FusedBatchNormV3: 97x
   - MaxPool: 4x
   - Mul: 17x
   - Pack: 4x
   - Pad: 1x
   - Relu: 16x
   - Relu6: 77x
   - Reshape: 19x
   - Sigmoid: 1x
   - Slice: 1x
   - Split: 1x
   - Squeeze: 8x
   - Sub: 6x
   - Tile: 5x
   - Transpose: 3x
   - Unpack: 2x

================================================================================================================
[*] Total number of TensorRT engines: 2
[*] % of OPs Converted: 97.38% [928/953]
```

## 6.3.2. Conversion Report

TF-TRT can now report the results of the conversion process controlled by the environment variable `TF_TRT_SHOW_DETAILED_REPORT`. The conversion report details the type and number of ops that were not converted, and the reason for their failure to convert. This is useful, for example, when debugging a conversion that resulted in a large number of engines.

This environment variable can take the following values:

- `TF_TRT_SHOW_DETAILED_REPORT=1`– this prints the detailed nonconversion report on stdout.
- `TF_TRT_SHOW_DETAILED_REPORT`=a non-empty string – this exports the nonconversion report in CSV format at the path defined by the environment variable.
- If this environment variable is not used, then the default report will be printed to stdout.

The following shows a sample nonconversion report:

```
################################################################################
TensorRT unsupported/non-converted OP Report:
   - ResizeBilinear -> 8x
       - [Count: 8x] Cannot Convert Bilinear Resize when align_corners=False
```

```
   - Identity -> 4x
      - [Count: 4x] excluded by segmenter option. Most likely an input or output
 node.

   - Cast -> 2x
      - [Count: 1x] Cast op is not supported - Allowed input dtypes: [float, half].
 Received: int32
      - [Count: 1x] Failed to convert at least one input to a TRT_TensorOrWeights:
 Unsupported tensorflow data type uint8

   - NoOp -> 2x
      - [Count: 2x] Op type NoOp is not supported.

   - CombinedNonMaxSuppression -> 1x
      - [Count: 1x] Op type CombinedNonMaxSuppression is not supported.

   - Placeholder -> 1x
      - [Count: 1x] excluded by segmenter option. Most likely an input or output
 node.

   - Unpack -> 1x
      - [Count: 1x] The argument `strided_slice_spec` is `absl::nullopt` with
 `dynamic_input_size_indices` non empty.

--------------------------------------------------------------------------------
   - Total nonconverted OPs: 19
   - Total nonconverted OP Types: 7
For more information see https://docs.nvidia.com/deeplearning/frameworks/tf-trt-
user-guide/index.html#supported-ops.
################################################################################
```

# 6.4. Logging

TF-TRT leverages TensorFlow's logging infrastructure wherever possible. Verbose logging can be turned on by using environment variables.

One of the popular methods to debug a failure/regression in TF-TRT is to turn on verbose logging in TF and then run the workload. This is controlled by the environment variable `TF_CPP_VMODULE`. Sample usage for this environment variable is shown here:

```
TF_CPP_VMODULE=trt_logger=2,trt_engine_utils=2,trt_engine_op=2,convert_nodes=2,convert_graph=2,seg
```

Any of the comma-separated values may be omitted, and the value of the variable controls how detailed the logging will be.

# 6.5. Export TRT Engines for Debugging

Engine export is controlled by the `TF_TRT_EXPORT_TRT_ENGINES_PATH` environment variable.

When this variable is populated with a valid path on the filesystem, the TFTRT converter exports all the TRT engines built by the converter to this location. This can be investigated as a standard TRT engine.

## 6.6. Blocking Conversion of Ops for Debugging

Blocking ops from getting converted to TRT is a useful debugging tool when trying to identify one or more problematic ops in TFTRT. This is controlled by the environment variable TF_TRT_OP_DENYLIST. This can take many values, and must be an exact stringwise match. For example:

```
TF_TRT_OP_DENYLIST=Sub,Exp,Conv2D
```

will prevent `Sub`, `Exp`, and `Conv2D` from b converted to TRT.

## 6.7. Using Experimental Features

The environment variable `TF_TRT_EXPERIMENTAL_FEATURES` can be used to enable specific experimental features in TF-TRT.

**Usage:** `TF_TRT_EXPERIMENTAL_FEATURES=feature_1,feature_2`

The following experimental features are supported in TF-TRT currently:

- `disable_graph_freezing`
- `reject_all_fp_cast_ops`

## 6.8. Overriding top_k Threshold for the NMS Plugin

The TRT NMS plugin allows top_k<=4096, where top_k = max(num_boxes, max_total_size). The user can override this by setting the `TF_TRT_ALLOW_NMS_TOPK_OVERRIDE=1` environment variable, but this can result in a loss of accuracy. This variable has effect before TRT 8.2.1, newer TRT versions do not have limitations on top_k.

**Usage:** `TF_TRT_ALLOW_NMS_TOPK_OVERRIDE=1`

## 6.9. Enabling the Tensor Layout Optimizer

The environment variable TF_TRT_ENABLE_LAYOUT_OPTIMIZER is used to enable/disable a layout conversion to NCHW. By default, the layout optimizer is turned on.

**Usage:** `TF_TRT_ENABLE_LAYOUT_OPTIMIZER=0`

## 6.10. Allowing Fallback to TF Native Segment Execution

The environment variable `TF_TRT_ALLOW_ENGINE_NATIVE_SEGMENT_EXECUTION` allows a graph to continue execution if the TRTEngine execution fails.

**Usage:** `TF_TRT_ALLOW_ENGINE_NATIVE_SEGMENT_EXECUTION=1`

## 6.11. Controlling the Number of Engines Generated

The environment variable `TF_TRT_MAX_ALLOWED_ENGINES` allows the user to specify the maximum number of TRT engines that will be generated in a particular model. This comes in handy for networks that are heavily segmented, and prevents the generation of dozens of engines, potentially increasing performance.

**Usage:**

- Don't limit number of engines: `TF_TRT_ENABLE_LAYOUT_OPTIMIZER=0`
- Limit number of engines created: `TF_TRT_MAX_ALLOWED_ENGINES=<N>` where <N> is the maximum number of engines created. If there are more than N segments in the graph, the largest N segments will be converted and engines created.

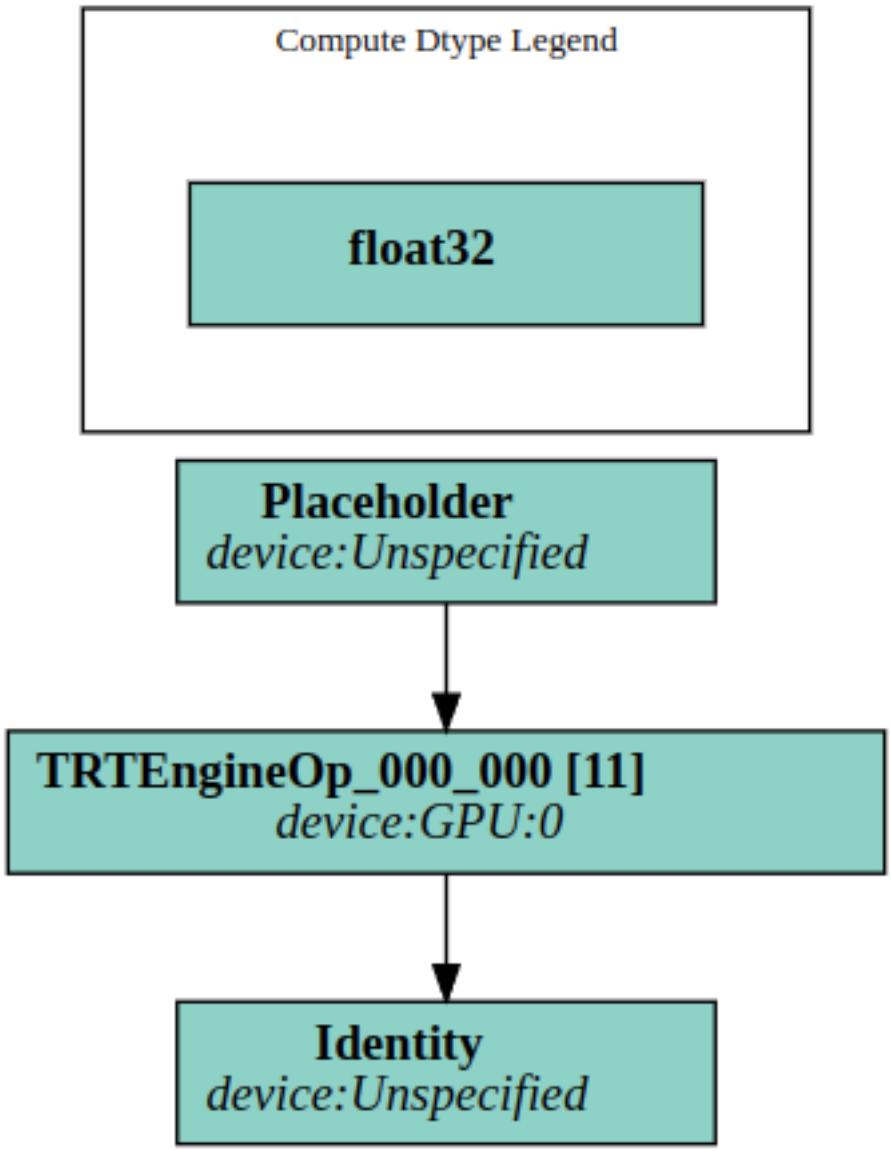
## 6.12. Visualizing the TF-TRT Graph

The environment variable `TF_TRT_EXPORT_GRAPH_VIZ_PATH` enables and controls where the graphical visualization of the network will be generated.

This is a nifty tool for visualizing the converted TRT graph in dot format. A variety of tools can be used to view dot files.

**Usage:** `TF_TRT_EXPORT_GRAPH_VIZ_PATH=/path/to/graphviz-output`

For example, the visualization of the mnist example we described earlier is shown in the following figure:


Compute Dtype Legend
float32
Placeholder
device:Unspecified
TRTEngineOp_000_000 [11]
device:GPU:0
Identity
device:Unspecified

# Chapter 7. Best Practices

The TensorFlow-TensorRT GitHub Repository is the best place to submit issues. When posting issues in GitHub, follow the process outlined in the Stack Overflow document. Ensure that posted examples are:

- Minimal – use as little code as possible that still produces the same problem.
- Complete – provide all parts needed to reproduce the problem. Check if you can strip external dependencies and still show the problem. The less time we spend on reproducing problems the more time we have to fix it.
- Verifiable – test the code you're about to provide to make sure it reproduces the problem. Remove all other problems that are not related to your request/question.

# Chapter 8. Advanced Features

## 8.1. 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 (original model size). TensorRT allocates memory through TensorFlow allocators, therefore, all TensorFlow memory configuration restrictions also apply to TensorRT.

Memory usage highly depends on the model since models can be large (e.g. transformers or small (simple image classification models). If you observe that your inference or engine building is running out of memory, try reducing the batch size.

## 8.2. Max Cached Engines

This is a parameter that is largely used only in implicit batch mode with `dynamic_op_mode` enabled, and not applicable to dynamic shapes mode. This specifies the maximum number of TRT engines that can be cached for dynamic TRT ops (dynamic_op_mode). The TRT engines created for a dynamic dimension are cached. If the number of cached engines is already at max but none of them supports the input shapes, the TRTEngineOp will fall back to run the original TF subgraph (native segment) that corresponds to the TRTEngineOp.

TensorRT engines are cached in an LRU cache located in the `TRTEngineOp` op. 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. 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.

## Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform 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 document. NVIDIA accepts no 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 document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

## HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

## OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

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

### Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, DALI, DGX, DGX-1, DGX-2, DGX Station, DLProf, Jetson, Kepler, Maxwell, NCCL, Nsight Compute, Nsight Systems, NvCaffe, PerfWorks, Pascal, SDK Manager, Tegra, TensorRT, Triton Inference Server, Tesla, TF-TRT, and Volta are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

### Copyright



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