NVIDIA TensorRT

Quick Start Guide | NVIDIA Docs
# Table of Contents

Chapter 1. Introduction..........................................................................................................................1

Chapter 2. Installing TensorRT............................................................................................................3
  2.1. Container Installation...............................................................................................................................3
  2.2. Debian Installation.....................................................................................................................................3
  2.3. Python Package Index Installation.....................................................................................................5

Chapter 3. The TensorRT Ecosystem............................................................................................... 7
  3.1. Basic TensorRT Workflow.......................................................................................................................7
  3.2. Conversion and Deployment Options................................................................................................9
    3.2.1. Conversion.............................................................................................................................................9
    3.2.2. Deployment........................................................................................................................................ 10
  3.3. Selecting the Correct Workflow....................................................................................................... 11

Chapter 4. Example Deployment Using ONNX..........................................................................13
  4.1. Export the Model.................................................................................................................................... 13
  4.2. Select a Batch Size.................................................................................................................................14
  4.3. Select a Precision....................................................................................................................................14
  4.4. Convert the Model..................................................................................................................................15
  4.5. Deploy the Model.................................................................................................................................... 15

Chapter 5. TF-TRT Framework Integration..................................................................................17

Chapter 6. ONNX Conversion and Deployment........................................................................ 18
  6.1. Exporting with ONNX............................................................................................................................ 18
    6.1.1. Exporting to ONNX from TensorFlow.....................................................................................18
    6.1.2. Exporting to ONNX from PyTorch............................................................................................19
  6.2. Converting ONNX to a TensorRT Engine.......................................................................................20
  6.3. Deploying a TensorRT Engine to the Python Runtime API.................................................... 21

Chapter 7. Using the TensorRT Runtime API.............................................................................22
  7.1. Setting Up the Test Container and Building the TensorRT Engine.................................... 22
  7.2. Running an Engine in C++...................................................................................................................24
  7.3. Running an Engine in Python............................................................................................................ 25

Chapter 8. Additional Resources.....................................................................................................27
  8.1. Glossary....................................................................................................................................................... 27
List of Figures

Figure 1. Typical Deep Learning Development Cycle Using TensorRT............................................ 1
Figure 2. The Five Basic Steps to Convert and Deploy Your Model.................................................... 8
Figure 3. Main Options Available for Conversion and Deployment.................................................. 9
Figure 4. Flowchart for Getting Started with TensorRT................................................................. 12
Figure 5. Deployment Process Using ONNX..................................................................................... 13
Figure 6. TF-TRT Integration with TensorRT..................................................................................... 17
Figure 7. Exporting ONNX from TensorFlow..................................................................................... 19
Figure 8. Exporting ONNX from PyTorch........................................................................................... 20
Figure 9. Test Image, Size 1282x1026.............................................................................................. 23
Figure 10. Test Image, Size 1282x1026............................................................................................. 25
List of Tables

Table 1. Additional TensorRT Resources............................................................... 27
Chapter 1. Introduction

NVIDIA® TensorRT™ is an SDK for optimizing trained deep learning models to enable high-performance inference. TensorRT contains a deep learning inference optimizer for trained deep learning models, and a runtime for execution.

After you have trained your deep learning model in a framework of your choice, TensorRT enables you to run it with higher throughput and lower latency.

Figure 1. Typical Deep Learning Development Cycle Using TensorRT

This guide covers the basic installation, conversion, and runtime options available in TensorRT, and when they are best applied.

Here is a quick summary of each chapter:

**Installing TensorRT**
- We provide multiple, simple ways of installing TensorRT.

**The TensorRT Ecosystem**
- We describe a simple flowchart to show the different types of conversion and deployment workflows and discuss their pros and cons.

**Example Deployment Using ONNX**
- We look at the basic steps to convert and deploy your model. It will introduce concepts used in the rest of the guide, and walk you through the decisions you must make to optimize inference execution.

**TF-TRT Framework Integration**
- We introduce the TensorRT (TRT) inside of Google® TensorFlow (TF) integration.
ONNX Conversion and Deployment
We provide a broad overview of ONNX exports from TensorFlow and PyTorch, as well as pointers to Jupyter notebooks that go into more detail.

Using the TensorRT Runtime API
We provide a tutorial to illustrate semantic segmentation of images using the TensorRT C++ and Python API.

For a higher-level application that allows you to quickly deploy your model, refer to the NVIDIA Triton™ Inference Server Quick Start.
Chapter 2. Installing TensorRT

There are a number of installation methods for TensorRT. This chapter covers the most common options using:

- a container
- a Debian file, or
- a standalone `pip` wheel file.

For other ways to install TensorRT, refer to the NVIDIA TensorRT Installation Guide.

For advanced users who are already familiar with TensorRT and want to get their application running quickly, who are using an NVIDIA CUDA® container with cuDNN included, or want to set up automation, follow the network repo installation instructions (refer to Using The NVIDIA Machine Learning Network Repo For Debian Installation).

2.1. Container Installation

This section contains an introduction to the customized virtual machine images (VMI) that NVIDIA publishes and maintains on a regular basis. NVIDIA NGC™ certified public cloud platform users can access specific setup instructions on how to browse the NGC website and identify an available NGC container and tag to run on their VMI.

On each of the major cloud providers, NVIDIA publishes customized GPU-optimized virtual machine images (VMI) with regular updates to OS and drivers. These VMIs are optimized for performance on the latest generations of NVIDIA GPUs. Using these VMIs to deploy NGC hosted containers, models and resources on cloud-hosted virtual machine instances with H100, A100, V100, or T4 GPUs ensures optimum performance for deep learning, machine learning, and HPC workloads.

To deploy a TensorRT container on a public cloud, follow the steps associated with your NGC certified public cloud platform.

2.2. Debian Installation

This section contains instructions for a developer installation. This installation method is for new users or users who want the complete developer installation, including samples and documentation for both the C++ and Python APIs.
For advanced users who are already familiar with TensorRT and want to get their application running quickly, are using an NVIDIA CUDA container, or want to set up automation, follow the network repo installation instructions (refer to Using The NVIDIA CUDA Network Repo For Debian Installation).

Note:
- The following commands are examples for `amd64`, however, the commands are identical for `arm64`.
- When installing Python packages using this method, you must install dependencies manually with `pip`.

Ensure that you have the following dependencies installed.

- CUDA 11.0 update 3, 11.1 update 1, 11.2 update 2, 11.3 update 1, 11.4 update 4, 11.5 update 2, 11.6 update 2, 11.7 update 1, 11.8, 12.0 update 1, 12.1 update 1, 12.2 update 2, 12.3 update 2, or 12.4 update 1
- cuDNN 8.9.7 (Optional and not required for lean or dispatch runtime installations.)

1. Install CUDA according to the CUDA installation instructions.
2. Download the TensorRT local repo file that matches the Ubuntu version and CPU architecture that you are using.
3. Install TensorRT from the Debian local repo package. Replace `ubuntuxx04,10.x.x, and cuda-x.x` with your specific OS version, TensorRT version, and CUDA version.

   ```bash
   os="ubuntuxx04"
   tag="10.x.x-cuda-x.x"
   sudo dpkg -i nv-tensorrt-local-repo-$os-$tag_1.0-1_amd64.deb
   sudo cp /var/nv-tensorrt-local-repo-$os-$tag/*-keyring.gpg /usr/share/keyrings/
   sudo apt-get update
   ```

   For the full C++ and Python runtimes
   ```bash
   sudo apt-get install tensorrt
   ```

   For the lean runtime only, instead of tensorrt
   ```bash
   sudo apt-get install libnvinfer-lean10
   sudo apt-get install libnvinfer-vc-plugin10
   ```

   For lean runtime Python package
   ```bash
   sudo apt-get install python3-libnvinfer-lean
   ```

   For the dispatch runtime only, instead of tensorrt
   ```bash
   sudo apt-get install libnvinfer-dispatch10
   sudo apt-get install libnvinfer-vc-plugin10
   ```

   For dispatch runtime Python package
   ```bash
   sudo apt-get install python3-libnvinfer-dispatch
   ```

   For all TensorRT Python packages without samples
   ```bash
   python3 -m pip install numpy
   sudo apt-get install python3-libnvinfer-dev
   ```

   The following additional packages will be installed:
   ```bash
   python3-libnvinfer
   python3-libnvinfer-lean
   python3-libnvinfer-dispatch
   ```

   If you want to install Python packages for the lean or dispatch runtime only, specify these individually rather than installing the `dev` package.
If you want to run samples that require onnx-graphsurgeon or use the Python module for your own project

```
python3 -m pip install numpy onnx
sudo apt-get install onnx-graphsurgeon
```

4. Verify the installation.
```
dpkg-query -W tensorrt
```
You should see something similar to the following:
```
tensorrt 10.0.1.x-1+cuda12.4
```

2.3. Python Package Index Installation

This section contains instructions for installing TensorRT from the Python Package Index.

When installing TensorRT from the Python Package Index, you’re not required to install TensorRT from a .tar, .deb, .rpm, or .zip package. All required libraries are included in the Python package. However, the header files, which may be needed if you want to access TensorRT C++ APIs or to compile plugins written in C++, are not included. Additionally, if you already have the TensorRT C++ library installed, using the Python package index version will install a redundant copy of this library, which may not be desirable. Refer to Tar File Installation for information on how to manually install TensorRT wheels that do not bundle the C++ libraries. You can stop after this section if you only need Python support.

The tensorrt Python wheel files only support Python versions 3.8 to 3.12 at this time and will not work with other Python versions. Only the Linux and Windows operating systems and the x86_64 CPU architecture are currently supported. These Python wheel files are expected to work on RHEL 8 or newer, Ubuntu 20.04 or newer, and Windows 10 or newer.

Note: If you do not have root access, you are running outside a Python virtual environment, or for any other reason you would prefer a user installation, then append `--user` to any of the pip commands provided.

1. Ensure the pip Python module is up-to-date and the wheel Python module is installed before proceeding or you may encounter issues during the TensorRT Python installation.
```
python3 -m pip install --upgrade pip
python3 -m pip install wheel
```

2. Install the TensorRT Python wheel.
```
python3 -m pip install --upgrade tensorrt
```

The above pip command will pull in all the required CUDA libraries in Python wheel format from PyPI because they are dependencies of the TensorRT Python wheel. Also, it will upgrade tensorrt to the latest version if you had a previous version installed.
Installing TensorRT

A TensorRT Python Package Index installation is split into multiple modules:

- TensorRT libraries (tensorrt_libs)
- Python bindings matching the Python version in use (tensorrt_bindings)
- Frontend source package, which pulls in the correct version of dependent TensorRT modules from pypi.nvidia.com (tensorrt)
- You can append -cu11 or -cu12 to any of the Python modules if you require a different CUDA major version. When unspecified, the TensorRT Python meta-packages default to the CUDA 12.x variants, which is the latest CUDA version supported by TensorRT.

Optionally, install the TensorRT lean or dispatch runtime wheels, which are similarly split into multiple Python modules. If you are only using TensorRT to run pre-built version compatible engines, you can install these wheels without installing the regular TensorRT wheel.

```bash
python3 -m pip install --upgrade tensorrt_lean
python3 -m pip install --upgrade tensorrt_dispatch
```

3. To verify that your installation is working, use the following Python commands to:

- Import the tensorrt Python module.
- Confirm that the correct version of TensorRT has been installed.
- Create a Builder object to verify that your CUDA installation is working.

```python
>>> import tensorrt
>>> print(tensorrt.__version__)
>>> assert tensorrt.Builder(tensorrt.Logger())
```

Use a similar procedure to verify that the lean and dispatch modules work as expected:

```python
>>> import tensorrt_lean as trt
>>> print(trt.__version__)
>>> assert trt.Runtime(trt.Logger())
```

```python
>>> import tensorrt_dispatch as trt
>>> print(trt.__version__)
>>> assert trt.Runtime(trt.Logger())
```

If the final Python command fails with an error message similar to the error message below, then you may not have the NVIDIA driver installed or the NVIDIA driver may not be working properly. If you are running inside a container, then try starting from one of the nvidia/cuda:x.y-base-<os> containers.

```
[TensorRT] ERROR: CUDA initialization failure with error 100. Please check your CUDA installation: ...
```

If the preceding Python commands worked, then you should now be able to run any of the TensorRT Python samples to further confirm that your TensorRT installation is working. For more information about TensorRT samples, refer to the NVIDIA TensorRT Sample Support Guide.
Chapter 3. The TensorRT Ecosystem

TensorRT is a large and flexible project. It can handle a variety of conversion and deployment workflows, and which workflow is best for you will depend on your specific use case and problem setting.

TensorRT provides several options for deployment, but all workflows involve the conversion of your model to an optimized representation, which TensorRT refers to as an engine. Building a TensorRT workflow for your model involves picking the right deployment option, and the right combination of parameters for engine creation.

3.1. Basic TensorRT Workflow

You must follow five basic steps to convert and deploy your model:
Figure 2. The Five Basic Steps to Convert and Deploy Your Model

1. **Export The Model**
2. **Select A Batch Size**
3. **Select A Precision**
4. **Convert The Model**
5. **Deploy The Model**

It is easiest to understand these steps in the context of a complete, end-to-end workflow: In Example Deployment Using ONNX, we will cover a simple framework-agnostic deployment workflow to convert and deploy a trained ResNet-50 model to TensorRT using ONNX conversion and TensorRT’s standalone runtime.
3.2. Conversion and Deployment Options

The TensorRT ecosystem breaks down into two parts:

1. The various paths users can follow to convert their models to optimized TensorRT engines.
2. The various runtimes users can target with TensorRT when deploying their optimized TensorRT engines.

Figure 3. Main Options Available for Conversion and Deployment

3.2.1. Conversion

There are three main options for converting a model with TensorRT:

- using TF-TRT
- automatic ONNX conversion from .onnx files
- manually constructing a network using the TensorRT API (either in C++ or Python)

For converting TensorFlow models, the TensorFlow integration (TF-TRT) provides both model conversion and a high-level runtime API, and has the capability to fall back to TensorFlow implementations where TensorRT does not support a particular operator. For more information about supported operators, refer to ONNX Operator Support.

A more performant option for automatic model conversion and deployment is to convert using ONNX. ONNX is a framework agnostic option that works with models in
TensorFlow, PyTorch, and more. TensorRT supports automatic conversion from ONNX files using either the TensorRT API, or `trtexec` - the latter being what we will use in this guide. ONNX conversion is all-or-nothing, meaning all operations in your model must be supported by TensorRT (or you must provide custom plug-ins for unsupported operations). The result of ONNX conversion is a singular TensorRT engine that allows less overhead than using TF-TRT.

For the most performance and customizability possible, you can also construct TensorRT engines manually using the TensorRT network definition API. This essentially involves building an identical network to your target model in TensorRT operation by operation, using only TensorRT operations. After a TensorRT network is created, you will then export just the weights of your model from the framework and load them into your TensorRT network. For this approach, more information about constructing the model using TensorRT’s network definition API, can be found here:

- [Creating A Network Definition From Scratch Using The C++ API](#)
- [Creating A Network Definition From Scratch Using The Python API](#)

### 3.2.2. Deployment

There are three options for deploying a model with TensorRT:

- deploying within TensorFlow
- using the standalone TensorRT runtime API
- using NVIDIA Triton Inference Server

Your choice for deployment will determine the steps required to convert the model.

When using TF-TRT, the most common option for deployment is to simply deploy within TensorFlow. TF-TRT conversion results in a TensorFlow graph with TensorRT operations inserted into it. This means you can run TF-TRT models like you would any other TensorFlow model using Python.

The TensorRT runtime API allows for the lowest overhead and finest-grained control, but operators that TensorRT does not natively support must be implemented as plug-ins (a library of prewritten plug-ins is available [here](#)). The most common path for deploying with the runtime API is using ONNX export from a framework, which is covered in this guide in the following section.

Last, NVIDIA Triton Inference Server is an open source inference-serving software that enables teams to deploy trained AI models from any framework (TensorFlow, TensorRT, PyTorch, ONNX Runtime, or a custom framework), from local storage or Google Cloud Platform or AWS S3 on any GPU- or CPU-based infrastructure (cloud, data center, or edge). It is a flexible project with several unique features - such as concurrent model execution of both heterogeneous models and multiple copies of the same model (multiple model copies can reduce latency further) as well as load balancing and model analysis. It is a good option if you must serve your models over HTTP - such as in a cloud inferencing solution. You can find the NVIDIA Triton Inference Server home page [here](#) and the documentation [here](#).
3.3. Selecting the Correct Workflow

Two of the most important factors in selecting how to convert and deploy your model are:

1. your choice of framework.
2. your preferred TensorRT runtime to target.

The following flowchart covers the different workflows covered in this guide. This flowchart will help you select a path based on these two factors.

For more information on the runtime options available, refer to the Jupyter notebook included with this guide on Understanding TensorRT Runtimes.
Figure 4. Flowchart for Getting Started with TensorRT

- TensorFlow
  - Deploy in TensorFlow?
    - Yes: Export To ONNX (see Exporting To ONNX From TensorFlow)
      - TF-TRT (see TF-TRT Framework Integration)
        - Deploy in C++ or Python?
          - C++: C++ Runtime (see Running An Engine In C++)
          - Python: Python Runtime (see Running An Engine In Python)
    - No: In what format is my model?
      - ONNX
      - PyTorch

- PyTorch
  - Export To ONNX (see Exporting To ONNX From PyTorch)
  - Deploy in C++ or Python?
ONNX is a framework-agnostic model format that can be exported from most major frameworks, including TensorFlow and PyTorch. TensorRT provides a library to perform direct conversion of ONNX into a TensorRT engine through the ONNX-TRT parser.

In this section, we will go through the five steps to convert a pretrained ResNet-50 model from the ONNX model zoo into a TensorRT engine. Visually, this is the process we will follow:

Figure 5. Deployment Process Using ONNX

After you understand the basic steps of the TensorRT workflow, you can dive into the more in-depth Jupyter notebooks (refer to the following topics) for using TensorRT using TF-TRT or ONNX. You can follow along in the introductory Jupyter notebook here, which covers these workflow steps in more detail, using the TensorFlow framework.

4.1. Export the Model

The two main automatic paths for TensorRT conversion require different model formats to successfully convert a model:
TF-TRT uses TensorFlow SavedModels.

The ONNX path requires that models are saved in ONNX.

In this example, we are using ONNX, so we need an ONNX model. We are going to use ResNet-50; a basic backbone vision model that can be used for a variety of purposes. We will perform classification using a pretrained ResNet-50 ONNX model included with the ONNX model zoo.

Download a pretrained ResNet-50 model from the ONNX model zoo using `wget` and untar it.

```
wget https://download.onnxruntime.ai/onnx/models/resnet50.tar.gz

tar xzf resnet50.tar.gz
```

This will unpack a pretrained ResNet-50 `.onnx` file to the path `resnet50/model.onnx`.

You can see how we export ONNX models that will work with this same deployment workflow in Exporting to ONNX from TensorFlow or Exporting to ONNX from PyTorch.

### 4.2. Select a Batch Size

Batch size can have a large effect on the optimizations TensorRT performs on our model. Generally speaking, at inference, we pick a small batch size when we want to prioritize latency and a larger batch size when we want to prioritize throughput. Larger batches take longer to process but reduce the average time spent on each sample.

TensorRT is capable of handling the batch size dynamically if you do not know until runtime what batch size you will need. That said, a fixed batch size allows TensorRT to make additional optimizations. For this example workflow, we use a fixed batch size of 64. For more information on handling dynamic input size, refer to dynamic shapes.

We set the batch size during the original export process to ONNX. This is demonstrated in the Exporting to ONNX from TensorFlow or Exporting to ONNX from PyTorch sections. The sample `model.onnx` file downloaded from the ONNX model zoo has its batch size set to 64 already. We will want to remember this when we deploy our model:

```
BATCH_SIZE=64
```

For more information about batch sizes, refer to Batching.

### 4.3. Select a Precision

Inference typically requires less numeric precision than training. With some care, lower precision can give you faster computation and lower memory consumption without sacrificing any meaningful accuracy. TensorRT supports TF32, FP32, FP16, FP8, BF16, and INT8 precisions.

TensorRT has two type systems:

- **Weak typing** allows TensorRT’s optimizer freedom to reduce precision to improve performance.
- **Strong typing** requires TensorRT to statically infer a type for each tensor in the network based on the types of the inputs and then adhere strictly to those types,
which is useful if you have already reduced precision before export, and want
TensorRT to conform.

For more information, refer to Strong Typing vs Weak Typing.

This guide demonstrates the use of a weakly typed network.

FP32 is the default training precision of most frameworks, so we will start by using FP32
for inference here.

```python
import numpy as np
PRECISION = np.float32
```

We set the precision that our TensorRT engine should use at runtime, which we will do in
the next section.

### 4.4. Convert the Model

The ONNX conversion path is one of the most universal and performant paths for
automatic TensorRT conversion. It works for TensorFlow, PyTorch, and many other
frameworks.

There are several tools to help you convert models from ONNX to a TensorRT engine. One
common approach is to use `trtexec` - a command-line tool included with TensorRT that
can, among other things, convert ONNX models to TensorRT engines and profile them.

We can run this conversion as follows:

```
trtexec --onnx=resnet50/model.onnx --saveEngine=resnet_engine.trt
```

This will convert our `resnet50/model.onnx` to a TensorRT engine named
`resnet_engine.trt`.

**Note:**
- To tell `trtexec` where to find our ONNX model, run:
  ```
  --onnx=resnet50/model.onnx
  ```
- To tell `trtexec` where to save our optimized TensorRT engine, run:
  ```
  --saveEngine=resnet_engine_intro.trt
  ```

### 4.5. Deploy the Model

After we have our TensorRT engine created successfully, we must decide how to run it
with TensorRT.

There are two types of TensorRT runtimes: a standalone runtime that has C++ and
Python bindings, and a native integration into TensorFlow. In this section, we will use a
simplified wrapper (`ONNXClassifierWrapper`) which calls the standalone runtime. We will
generate a batch of randomized “dummy” data and use our `ONNXClassifierWrapper` to
run inference on that batch. For more information on TensorRT runtimes, refer to the
Understanding TensorRT Runtimes Jupyter notebook.
1. Set up the ONNXClassifierWrapper (using the precision we determined in Select a Precision).

   ```python
   from onnx_helper import ONNXClassifierWrapper
   N_CLASSES = 1000  # Our ResNet-50 is trained on a 1000 class ImageNet task
   trt_model = ONNXClassifierWrapper("resnet_engine.trt", [BATCH_SIZE, N_CLASSES],
   target_dtype = PRECISION)
   ```

2. Generate a dummy batch.

   ```python
   BATCH_SIZE=32
   dummy_input_batch = np.zeros((BATCH_SIZE, 224, 224, 3), dtype = PRECISION)
   ```

3. Feed a batch of data into our engine and get our predictions.

   ```python
   predictions = trt_model.predict(dummy_input_batch)
   ```

Note that the wrapper does not load and initialize the engine until running the first batch, so this batch will generally take a while. For more information about batching, refer to Batching.

For more information about TensorRT APIs, refer to the NVIDIA TensorRT API Reference. For more information on the ONNXClassifierWrapper, see its implementation on GitHub here.
Chapter 5. TF-TRT Framework Integration

The TF-TRT integration provides a simple and flexible way to get started with TensorRT. TF-TRT is a high-level Python interface for TensorRT that works directly with TensorFlow models. It allows you to convert TensorFlow SavedModels to TensorRT optimized models and run them within Python using a high-level API.

TF-TRT provides both a conversion path and a Python runtime that allows you to run an optimized model the way you would any other TensorFlow model. This has a number of advantages, notably that TF-TRT is able to convert models that contain a mixture of supported and unsupported layers without having to create custom plug-ins, by analyzing the model and passing subgraphs to TensorRT where possible to convert into engines independently.

This notebook provides a basic introduction and wrapper that simplifies the process of working with basic Keras/TensorFlow 2 models. In the notebook, we take a pretrained ResNet-50 model from the keras.applications model zoo, convert it using TF-TRT, and run it in the TF-TRT Python runtime.

Visually, the TF-TRT notebook demonstrates how to follow this path through TensorRT:

Figure 6. TF-TRT Integration with TensorRT
Chapter 6. ONNX Conversion and Deployment

The ONNX interchange format provides a way to export models from many frameworks, including PyTorch, TensorFlow, and TensorFlow 2, for use with the TensorRT runtime. Importing models using ONNX requires the operators in your model to be supported by ONNX, and for you to supply plug-in implementations of any operators TensorRT does not support. (A library of plug-ins for TensorRT can be found here).

6.1. Exporting with ONNX

ONNX models can be easily generated from TensorFlow models using the ONNX project's tf2onnx tool.

This notebook shows how to generate ONNX models from a Keras/T2 ResNet-50 model, how to convert those ONNX models to TensorRT engines using trtexec, and how to use the Python TensorRT runtime to feed a batch of data into the TensorRT engine at inference time.

6.1.1. Exporting to ONNX from TensorFlow

TensorFlow can be exported through ONNX and run in one of our TensorRT runtimes. Here, we provide the steps needed to export an ONNX model from TensorFlow. For more information, refer to the Using Tensorflow 2 through ONNX notebook. The notebook will walk you through this path, starting from the below export steps:
1. Import a ResNet-50 model from `keras.applications`. This will load a copy of ResNet-50 with pretrained weights.
   ```python
   from tensorflow.keras.applications import ResNet50
   model = ResNet50(weights='imagenet')
   
   2. Convert the ResNet-50 model to ONNX format.
   ```
   ```python
   import tf2onnx
   model.save('my_model')
   !python -m tf2onnx.convert --saved-model my_model --output temp.onnx
   onnx_model = onnx.load_model('temp.onnx')
   ```
   ```python
   BATCH_SIZE = 64
   inputs = onnx_model.graph.input
   for input in inputs:
       dim1 = input.type.tensor_type.shape.dim[0]
       dim1.dim_value = BATCH_SIZE
   
   4. Save the ONNX file.
   ```
   ```python
   model_name = "resnet50_onnx_model.onnx"
   onnx.save_model(onnx_model, model_name)
   ```

### 6.1.2. Exporting to ONNX from PyTorch

One approach to converting a PyTorch model to TensorRT is to export a PyTorch model to ONNX and then convert into a TensorRT engine. For more details, refer to [Using](#)
**PyTorch with TensorRT through ONNX.** The notebook will walk you through this path, starting from the below export steps:

**Figure 8.** Exporting ONNX from PyTorch

1. Import a ResNet-50 model from `torchvision`. This will load a copy of ResNet-50 with pretrained weights.
   ```python
   import torchvision.models as models
   resnext50_32x4d = models.resnext50_32x4d(pretrained=True)
   ```

2. Save the ONNX file from PyTorch.
   ```python
   import torch
   BATCH_SIZE = 64
   dummy_input = torch.randn(BATCH_SIZE, 3, 224, 224)
   ```

3. Save the ONNX file.
   ```python
   import torch.onnx
   torch.onnx.export(resnext50_32x4d, dummy_input, "resnet50_onnx_model.onnx", verbose=False)
   ```

### 6.2. Converting ONNX to a TensorRT Engine

There are two main ways of converting ONNX files to TensorRT engines:

- using `trtexec`
- using the TensorRT API
In this guide, we will focus on using `trtexec`. To convert one of the preceding ONNX models to a TensorRT engine using `trtexec`, we can run this conversion as follows:

```
trtexec --onnx=resnet50_onnx_model.onnx --saveEngine=resnet_engine.trt
```

This will convert our `resnet50_onnx_model.onnx` to a TensorRT engine named `resnet_engine.trt`.

### 6.3. Deploying a TensorRT Engine to the Python Runtime API

There are a number of runtimes available to target with TensorRT. When performance is important, the TensorRT API is a great way of running ONNX models. We will go into the deployment of a more complex ONNX model using the TensorRT runtime API in both C++ and Python in the following section.

For the purposes of the preceding model, you can see how to deploy it in Jupyter with the Python runtime API in the notebooks [Using Tensorflow 2 through ONNX](#) and [Using PyTorch through ONNX](#). Another simple option is to use the [ONNXClassifierWrapper](#) provided with this guide, as demonstrated in [Deploy the Model](#).
Chapter 7. Using the TensorRT Runtime API

One of the most performant and customizable options for both model conversion and deployment are to use the TensorRT API, which has both C++ and Python bindings. TensorRT includes a standalone runtime with C++ and Python bindings, which are generally more performant and more customizable than using the TF-TRT integration and running in TensorFlow. The C++ API has lower overhead, but the Python API works well with Python data loaders and libraries like NumPy and SciPy, and is easier to use for prototyping, debugging and testing.

The following tutorial illustrates semantic segmentation of images using the TensorRT C++ and Python API. A fully convolutional model with ResNet-101 backbone is used for this task. The model accepts images of arbitrary sizes and produces per-pixel predictions.

The tutorial consists of the following steps:

1. Setup–launch the test container, and generate the TensorRT engine from a PyTorch model exported to ONNX and converted using `trtexec`
2. C++ runtime API–run inference using engine and TensorRT’s C++ API
3. Python runtime AP–run inference using engine and TensorRT’s Python API

7.1. Setting Up the Test Container and Building the TensorRT Engine

1. Download the source code for this quick start tutorial from the TensorRT Open Source Software repository.
   
   ```
   $ git clone https://github.com/NVIDIA/TensorRT.git
   $ cd TensorRT/quickstart
   ```

2. Convert a pre-trained FCN-ResNet-101 model from torch.hub to ONNX.
   
   Here we use the export script that is included with the tutorial to generate an ONNX model and save it to `fcn-resnet101.onnx`. For details on ONNX conversion refer to ONNX Conversion and Deployment. The script also generates a test image of size 1282x1026 and saves it to `input.ppm`.
Using the TensorRT Runtime API

Figure 9. Test Image, Size 1282x1026

a). Launch the NVIDIA PyTorch container for running the export script.

```
$ docker run --rm -it --gpus all -p 8888:8888 -v `pwd`:/workspace -w /workspace/
SemanticSegmentation nvcr.io/nvidia/pytorch:20.12-py3 bash
```

b). Run the export script to convert the pretrained model to ONNX.

```
$ python export.py
```

Note: FCN-ResNet-101 has one input of dimension [batch, 3, height, width] and one output of dimension [batch, 21, height, weight] containing unnormalized probabilities corresponding to predictions for 21 class labels. When exporting the model to ONNX, we append an argmax layer at the output to produce per-pixel class labels of highest probability.

3. Build a TensorRT engine from ONNX using the trtexec tool.

trtexec can generate a TensorRT engine from an ONNX model that can then be deployed using the TensorRT runtime API. It leverages the TensorRT ONNX parser to load the ONNX model into a TensorRT network graph, and the TensorRT Builder API to generate an optimized engine. Building an engine can be time-consuming, and is usually performed offline.

Building an engine can be time-consuming, and is usually performed offline.

```
trtexec --onnx=fcn-resnet101.onnx --fp16 --workspace=64 --minShapes=input:1x3x256x256 --optShapes=input:1x3x1026x1282 --maxShapes=input:1x3x1440x2560 --buildOnly --saveEngine=fcn-resnet101.engine
```

Successful execution should result in an engine file being generated and see something similar to Successful in the command output.

trtexec can build TensorRT engines with the build configuration options as described in the NVIDIA TensorRT Developer Guide.

4. Optionally, validate the generated engine for random-valued input using trtexec.

```
trtexec --shapes=input:1x3x1026x1282 --loadEngine=fcn-resnet101.engine
```
Using the TensorRT Runtime API

Where `--shapes` sets the input sizes for the dynamic shaped inputs to be used for inference.

If successful, you should see something similar to the following:

```bash
&&&& PASSED TensorRT.trtexec # trtexec --shapes=input:1x3x1026x1282 --loadEngine=fcn-resnet101.engine
```

7.2. Running an Engine in C++

1. Compile and run the C++ segmentation tutorial within the test container.

```bash
$ make
$ ./bin/segmentation_tutorial
```

The following steps show how to use the [Deserializing A Plan](#) for inference.

1. Deserialize the TensorRT engine from a file. The file contents are read into a buffer and deserialized in-memory.

```cpp
std::vector<char> engineData(fsize);
engineFile.read(engineData.data(), fsize);

std::unique_ptr<nvinfer1::IRuntime> runtime{nvinfer1::createInferRuntime(sample::gLogger.getTRTLogger())};
std::unique_ptr<nvinfer1::ICudaEngine> mEngine(runtime->deserializeCudaEngine(engineData.data(), fsize, nullptr));
```

2. A TensorRT execution context encapsulates execution state such as persistent device memory for holding intermediate activation tensors during inference.

Since the segmentation model was built with dynamic shapes enabled, the shape of the input must be specified for inference execution. The network output shape may be queried to determine the corresponding dimensions of the output buffer.

```cpp
auto input_idx = mEngine->getBindingIndex("input");
assert(mEngine->getBindingDataType(input_idx) == nvinfer1::DataType::kFLOAT);
auto input_dims = nvinfer1::Dims4{1, 3 /* channels */, height, width};
context->setBindingDimensions(input_idx, input_dims);
auto input_size = util::getMemorySize(input_dims, sizeof(float));
auto output_idx = mEngine->getBindingIndex("output");
assert(mEngine->getBindingDataType(output_idx) == nvinfer1::DataType::kINT32);
auto output_dims = context->getBindingDimensions(output_idx);
auto output_size = util::getMemorySize(output_dims, sizeof(int32_t));
```

Note: The binding indices for network I/O can be queried by name.

3. In preparation for inference, CUDA device memory is allocated for all inputs and outputs, image data is processed and copied into input memory, and a list of engine bindings is generated.

For semantic segmentation, input image data is processed by fitting into a range of \([0, 1]\) and normalized using mean \([0.485, 0.456, 0.406]\) and std deviation \([0.229, 0.224, 0.225]\). Refer to the input-preprocessing requirements for the `torchvision` models [here](#). This operation is abstracted by the utility class `RGBImageReader`.

```cpp
void* input_mem{nullptr};
cudaMalloc(&input_mem, input_size);
```
void* output_mem{nullptr};
cudaMalloc(&output_mem, output_size);
const std::vector<float> mean{0.485f, 0.456f, 0.406f};
const std::vector<float> stddev{0.229f, 0.224f, 0.225f};
auto input_image{util::RGBImageReader(input_filename, input_dims, mean, stddev)};
input_image.read();
cudaStream_t stream;
auto input_buffer = input_image.process();
cudaMemcpyAsync(input_mem, input_buffer.get(), input_size, cudaMemcpyHostToDevice, stream);

4. Inference execution is kicked off using the context’s `executeV2` or `enqueueV2` methods. After the execution is complete, we copy the results back to a host buffer and release all device memory allocations.

    void* bindings[] = {input_mem, output_mem};
    bool status = context->enqueueV2(bindings, stream, nullptr);
    auto output_buffer = std::unique_ptr<int>{new int[output_size]};
    cudaMemcpyAsync(output_buffer.get(), output_mem, output_size, cudaMemcpyDeviceToHost, stream);
    cudaStreamSynchronize(stream);
    cudaFree(input_mem);
    cudaFree(output_mem);

5. To visualize the results, a pseudo-color plot of per-pixel class predictions is written out to `output.ppm`. This is abstracted by the utility class `ArgmaxImageWriter`.

    const int num_classes{21};
    const std::vector<int> palette{0x1 << 25) - 1, (0x1 << 15) - 1, (0x1 << 21) - 1};
    auto output_image{util::ArgmaxImageWriter(output_filename, output_dims, palette, num_classes)};
    output_image.process(output_buffer.get());
    output_image.write();

For the test image, the expected output is as follows:

Figure 10. Test Image, Size 1282x1026

7.3. Running an Engine in Python
1. Install the required Python packages.
   $ pip install pycuda

2. Launch Jupyter and use the provided token to log in using a browser http://<host-ip-address>:8888.
   $ jupyter notebook --port=8888 --no-browser --ip=0.0.0.0 --allow-root

3. Open the tutorial-runtime.ipynb notebook and follow its steps.

The TensorRT Python runtime APIs map directly to the C++ API described in Running an Engine in C++.
## Chapter 8. Additional Resources

### Table 1. Additional TensorRT Resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer builder API documentation - for manual</td>
<td>The manual layer builder API is useful for when you need the maximum flexibility possible in building a TensorRT engine.</td>
</tr>
<tr>
<td>TensorRT engine construction:</td>
<td>The Layer Builder API lets you construct a network from scratch by hand in TensorRT, and gives you tools to load in weights from your model.</td>
</tr>
<tr>
<td><a href="#">Creating a Network Definition in Python</a></td>
<td>When using the layer builder API, your goal is to essentially build an identical network to your training model using TensorRT layer by layer, and then load in the weights from your model.</td>
</tr>
<tr>
<td><a href="#">Creating a Network Definition in C++</a></td>
<td></td>
</tr>
<tr>
<td>ONNX-TensorRT GitHub</td>
<td>The ONNX-TensorRT integration is a simple high-level interface for ONNX conversion with a Python runtime. It is useful for early prototyping of TensorRT workflows using the ONNX path.</td>
</tr>
<tr>
<td>TF-TRT product documentation</td>
<td>Product documentation page for TF-TRT.</td>
</tr>
<tr>
<td>TensorRT is integrated with NVIDIA’s profiling</td>
<td>This is a great next step for further optimizing and debugging models that you are working on productionizing.</td>
</tr>
<tr>
<td>tools, <a href="#">NVIDIA Nsight Systems</a></td>
<td></td>
</tr>
<tr>
<td>and [NVIDIA Deep Learning Profiler (DLProf)]</td>
<td></td>
</tr>
<tr>
<td>TensorRT product documentation</td>
<td>Product documentation page for the ONNX, layer builder, C++, and legacy APIs.</td>
</tr>
<tr>
<td>TensorRT OSS GitHub</td>
<td>Contains OSS TensorRT components, sample applications, and plug-in examples.</td>
</tr>
<tr>
<td>TensorRT developer page</td>
<td>Contains downloads, posts, and quick reference code samples.</td>
</tr>
</tbody>
</table>

### 8.1. Glossary

B
**Batch**
A batch is a collection of inputs that can all be processed uniformly. Each instance in the batch has the same shape and flows through the network in exactly the same way. All instances can therefore be computed in parallel.

**Builder**
TensorRT’s model optimizer. The builder takes as input a network definition, performs device-independent and device-specific optimizations, and creates an engine. For more information about the builder, refer to the Builder API.

**Dynamic batch**
A mode of inference deployment where the batch size is not known until runtime. Historically, TensorRT treated batch size as a special dimension, and the only dimension that was configurable at runtime. TensorRT 6 and later allow engines to be built such that all dimensions of inputs can be adjusted at runtime.

**Engine**
A representation of a model that has been optimized by the TensorRT builder. For more information about the engine, refer to the Execution API.

**Explicit batch**
An indication to the TensorRT builder that the model includes the batch size as one of the dimensions of the input tensors. TensorRT’s implicit batch mode allows the batch size to be omitted from the network definition and provided by the user at runtime, but this mode has been deprecated and is not supported by the ONNX parser.

**Framework integration**
An integration of TensorRT into a framework such as TensorFlow, which allows model optimization and inference to be performed within the framework.

**Network definition**
A representation of a model in TensorRT. A network definition is a graph of tensors and operators.

**ONNX**
Open Neural Network eXchange. A framework-independent standard for representing machine learning models. For more information about ONNX, refer to onnx.ai.

**ONNX parser**
A parser for creating a TensorRT network definition from an ONNX model. For more details on the C++ ONNX Parser, refer to the NvONNXParser or the Python ONNX Parser.

**Plan**
An optimized inference engine in a serialized format. To initialize the inference engine, the application will first deserialize the model from the plan file. A typical application will build an engine once, and then serialize it as a plan file for later use.
**Precision**
Refers to the numerical format used to represent values in a computational method. This option is specified as part of the TensorRT build step. TensorRT supports mixed precision inference with FP32, TF32, FP16, or INT8 precisions. Devices before NVIDIA Ampere Architecture default to FP32. NVIDIA Ampere Architecture and later devices default to TF32, a fast format using FP32 storage with lower-precision math.

**Runtime**
The component of TensorRT that performs inference on a TensorRT engine. The runtime API supports synchronous and asynchronous execution, profiling, and enumeration and querying of the bindings for an engine inputs and outputs.

**TF-TRT**
TensorFlow integration with TensorRT. Optimizes and executes compatible subgraphs, allowing TensorFlow to execute the remaining graph.
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 acknowledgment, 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.
Arm
Arm, AMBA and Arm Powered are registered trademarks of Arm Limited. Cortex, MPCore and Mali are trademarks of Arm Limited. "Arm" is used to represent Arm Holdings plc; its operating company Arm Limited; and the regional subsidiaries Arm inc.; Arm KK; Arm Korea Limited.; Arm Taiwan Limited; Arm France SAS; Arm Consulting [Shanghai] Co. Ltd.; Arm Germany GmbH; Arm Embedded Technologies Pvt. Ltd.; Arm Norway, AS and Arm Sweden AB.
HDMI
HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.
Blackberry/QNX
Copyright © 2020 BlackBerry Limited. All rights reserved.
 TRADEMARKS, INCLUDING BUT NOT LIMITED TO BLACKBERRY, EMBLEM Design, QNX, AVIAGE, MONTIMENTS, NEUTRINO and QNX CAR are the trademarks or registered trademarks of BlackBerry Limited, used under license, and the exclusive rights to such trademarks are expressly reserved.
Google
Android, Android TV, Google Play and the Google Play logo are trademarks of Google, Inc.