TABLE OF CONTENTS

Chapter 1. Overview Of TensorFlow ................................................................. 1
  1.1. Contents Of The NVIDIA TensorFlow Container .......................................... 1
Chapter 2. Pulling The TensorFlow Container .................................................... 3
Chapter 3. Running A TensorFlow Container ..................................................... 4
Chapter 4. Verifying TensorFlow .................................................................... 5
Chapter 5. Customizing And Extending TensorFlow ........................................ 6
  5.1. Benefits And Limitations To Customizing TensorFlow .................................... 7
  5.2. Example 1: Customizing TensorFlow Using Dockerfile .................................. 7
  5.3. Example 2: Customizing TensorFlow Using docker commit ............................ 9
  5.4. Accelerating Inference In TensorFlow With TensorRT ............................... 11
Chapter 6. TensorFlow Parameters ................................................................. 13
  6.1. Added And Modified Parameters ............................................................... 13
Chapter 7. TensorFlow Environment Variables .............................................. 15
  7.1. Added Or Modified Variables .................................................................... 15
Chapter 8. Performance ................................................................................. 20
  8.1. Tensor Core Math .................................................................................... 20
    8.1.1. Float16 Training ............................................................................ 20
  8.2. Automatic Mixed Precision (AMP) ............................................................ 21
    8.2.1. Automatic Mixed Precision Training In TensorFlow ............................. 21
    8.2.2. Conditions And Limitations ............................................................ 22
    8.2.3. FAQs ......................................................................................... 23
Chapter 9. Troubleshooting ............................................................................ 26
  9.1. Support .................................................................................................. 26
Chapter 1.
OVERVIEW OF TENSORFLOW

TensorFlow is an open-source software library for numerical computation using data flow graphs.

Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) that flow between them. This flexible architecture lets you deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device without rewriting code.

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

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

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

1.1. Contents Of The NVIDIA TensorFlow Container

This image contains source and binaries for TensorFlow. The pre-built and installed version of TensorFlow is located in the /usr/local/[bin,lib] directories. The complete source code is located in /opt/tensorflow.

To achieve optimum TensorFlow performance, there are sample scripts within the container image. For more information, see Performance.

TensorFlow includes TensorBoard, a data visualization toolkit developed by Google.

Additionally, this container image also includes several built-in TensorFlow examples that you can run using commands like the following. These examples perform training of convolutional neural networks (CNNs). For more information, see MNIST For ML Beginners. The following Python commands run two of these examples:
python -m tensorflow.models.image.mnist.convolutional
python -m tensorflow.models.image.cifar10.cifar10_multi_gpu_train

The first command uses the MNIST data set, for example, THE MNIST DATABASE. The second command uses the CIFAR-10 dataset, for example, The CIFAR-10 dataset.
Chapter 2.
PULLING THE TENSORFLOW CONTAINER

To pull a TensorFlow container, see Pulling A Container
Chapter 3.  
RUNNING A TENSORFLOW CONTAINER

To run a TensorFlow container, see Running TensorFlow.
Chapter 4.
VERIFYING TENSORFLOW

The simplest way to verify that TensorFlow is running correctly, is to run the examples that are included in the `nvidia-examples/` directory. Each example contains a README that describes the basic usage.
Chapter 5.
CUSTOMIZING AND EXTENDING TENSORFLOW

The nvidia-docker images come prepackaged, tuned, and ready to run; however, you may want to build a new image from scratch or augment an existing image with custom code, libraries, data, or settings for your corporate infrastructure. This section will guide you through exercises that will highlight how to create a container from scratch, customize a container, extend a deep learning framework to add features, develop some code using that extended framework from the developer environment, then package that code as a versioned release.

By default, you do not need to build a container. The NVIDIA container repository, nvcr.io, has a number of containers that can be used immediately including containers for deep learning as well as containers with just the CUDA® Toolkit™.

One of the great things about containers is that they can be used as starting points for creating new containers. This can be referred to as customizing or extending a container. You can create a container completely from scratch, however, since these containers are likely to run on GPUs, it is recommended that you at least start with a nvcr.io container that contains the OS and CUDA®. However, you are not limited to this and can create a container that runs on the CPUs which does not use the GPUs. In this case, you can start with a bare OS container from another location such as Docker. To make development easier, you can still start with a container with CUDA; it is just not used when the container is used.

The customized or extended containers can be saved to a user’s private container repository. They can also be shared with other users but this requires some administrator help.
It is important to note that all NGC deep learning framework images include the source to build the framework itself as well as all of the prerequisites.

**Attention** Do not install an NVIDIA driver into the Docker image at docker build time.

A best-practice is to *avoid docker commit* usage for developing new Docker images, and to use Dockerfiles instead. The Dockerfile method provides visibility and capability to efficiently version-control changes made during the development of a Docker image. The Docker commit method is appropriate for short-lived, disposable images only.

For more information on writing a Docker file, see the best practices documentation.

### 5.1. Benefits And Limitations To Customizing TensorFlow

You can customize a container to fit your specific needs for numerous reasons; for example, you depend upon specific software that is not included in the container that NVIDIA provides. No matter your reasons, you can customize a container.

The container images do not contain sample data-sets or sample model definitions unless they are included with the framework source. Be sure to check the container for sample data-sets or models.

### 5.2. Example 1: Customizing TensorFlow Using Dockerfile

Before customizing the container, you should ensure the TensorFlow 17.04 container has been loaded into the NGC container registry using the `docker pull` command before proceeding. For example:

```
$ docker pull nvcr.io/nvidia/tensorflow:17.04
```

The Docker containers on nvcr.io also provide a sample Dockerfile that explains how to patch a framework and rebuild the Docker image. In the directory, `/workspace/docker-examples`, there are two sample Dockerfiles that you can use. The first one, `Dockerfile.addpackages`, can be used to add packages to the TensorFlow image. The second one, `Dockerfile.customtensorflow`, illustrates how to patch TensorFlow and rebuild the image.

FROM nvcr.io/nvidia/tensorflow:17.04

# Bring in changes from outside container to /tmp
# (assumes my-tensorflow-modifications.patch is in same directory as Dockerfile)
COPY my-tensorflow-modifications.patch /tmp
# Change working directory to TensorFlow source path
WORKDIR /opt/tensorflow

# Apply modifications
RUN patch -p1 < /tmp/my-tensorflow-modifications.patch

# Rebuild TensorFlow
RUN yes "" &> /dev/null && \
    bazel build -c opt --config=cuda
    tensorflow/tools/pip_package:build_pip_package &> /dev/null && \
    bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/pip &> /dev/null && \
    pip install --upgrade /tmp/pip/tensorflow-*.whl &> /dev/null && \
    rm -rf /tmp/pip/tensorflow-*.whl &> /dev/null && \
    bazel clean --expunge

# Apply modifications
WORKDIR /workspace

This DockerFile will rebuild the TensorFlow image in the same way as it was built in the original image. For more information, see Dockerfile reference.

To better understand the Dockerfile, let's walk through the major commands. The first line in the Dockerfile is the following:

FROM nvcr.io/nvidia/tensorflow:17.04

This line starts with the NVIDIA 17.04 version image for TensorFlow being used as the starting point.

The second line is the following:

COPY my-tensorflow-modifications.patch /tmp

It brings in changes from outside the container into your /tmp directory. This assumes that the my-tensorflow-modifications.patch file is in the same directory as Dockerfile.

The next important line in the file changes the working directory to the TensorFlow source path.

WORKDIR /opt/tensorflow

This is followed by the command to apply the modifications patch to the source.

RUN patch -p1 < /tmp/my-tensorflow-modifications.patch

After the patch is applied, the TensorFlow image can be rebuilt. This is done via the RUN command in the Dockerfile.
Finally, the last major line in the DockerFile resets the default working directory.

```bash
WORKDIR /workspace
```

## 5.3. Example 2: Customizing TensorFlow Using `docker commit`

This example uses the `docker commit` command to flush the current state of the container to a Docker image. This is not a recommended best practice, however, this is useful when you have a container running to which you have made changes and want to save them. In this example, we are using the `apt-get` tag to install a package that requires the user run as root.

- The TensorFlow image release 17.04 is used in the example instructions for illustrative purposes.
- Do not use the `--rm` flag when running the container. If you use the `--rm` flag when running the container your changes will be lost when exiting the container.

1. Pull the Docker container from the `nvcr.io` repository to your DGX™ system. For example, the following command will pull the TensorFlow container:

```bash
$ docker pull nvcr.io/nvidia/tensorflow:17.04
```

2. Run the container on your DGX.

```bash
docker run --gpus all -ti nvcr.io/nvidia/tensorflow:17.04
```

---

NVIDIA Release 17.04 (build 21630)

Container image Copyright (c) 2017, NVIDIA CORPORATION. All rights reserved.

Copyright 2017 The TensorFlow Authors. All rights reserved.

Various files include modifications (c) NVIDIA CORPORATION. All rights reserved.

NVIDIA modifications are covered by the license terms that apply to the underlying project or file.

NOTE: The SHMEM allocation limit is set to the default of 64MB. This may be insufficient for TensorFlow. NVIDIA recommends the use of the following flags:
3. You should now be the root user in the container (notice the prompt). You can use the `apt` command to pull down a package and put it in the container.

```
docker run --gpus all --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 ...
```

```
root@8db6076d82c4:/workspace#
```

The NVIDIA containers are built using Ubuntu which uses the `apt-get` package manager. Check the container release notes Deep Learning Documentation for details on the specific container you are using.

In this example, we will install octave; the GNU clone of MATLAB, into the container.

```
# apt-get update
# apt install octave
```

You have to first issue `apt-get update` before you install Octave using `apt`.

4. Exit the workspace.

```
# exit
```

5. Display the list of running containers.

```
$ docker ps -a
```

As an example, here is some of the output from the `docker ps -a` command:

```
CONTAINER ID     IMAGE                                 CREATED
...              nvcr.io/nvidia/tensorflow:17.04      3
8db6076d82c4     nvcr.io/nvidia/tensorflow:17.04      3
minutes ago   ...
```

6. Now you can create a new image from the container that is running where you have installed Octave. You can commit the container with the following command.

```
$ docker commit 8db6076d82c4 nvcr.io/nvidian_sas/tensorflow_octave:17.04
```

```
sha256:25198e37ae2e3416bebcf1d3084ff3a95600d978811fe7f4f184de0af3878b51
```

7. Display the list of images.

```
$ docker images
```

```
REPOSITORY                              TAG
nvvidian_sas/tensorflow_octave          17.04
```

8. To verify, run the container again and see if Octave is actually there.

```
docker run --gpus all -ti nvvidian_sas/tensorflow_octave:17.04
```

```
== TensorFlow ==
```

```
NVIDIA Release 17.04 (build 21630)
```
NOTE: The SHMEM allocation limit is set to the default of 64MB. This may be insufficient for TensorFlow. NVIDIA recommends the use of the following flags:

docker run --gpus all --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 ...

root@87e8dde4be6d:/workspace# octave
octave: X11 DISPLAY environment variable not set
octave: disabling GUI features
GNU Octave, version 4.0.0
Copyright (C) 2015 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. For details, type 'warranty'.

Octave was configured for "x86_64-pc-linux-gnu".
Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/get-involved.html
Read http://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.

octave:1>

Since the Octave prompt displayed, Octave is installed.

9. If you are using a DGX-1 or DGX Station, and you want to save the container into your private repository (Docker uses the phrase "push"), then you can use the docker push ... command.

$ docker push nvcr.io/nvidian_sas/tensorflow_octave:17.04

5.4. Accelerating Inference In TensorFlow With TensorRT

For step-by-step instructions on how to use TensorRT with the TensorFlow framework, see Accelerating Inference In TensorFlow With TensorRT User Guide. To view the key
features, software enhancements and improvements, and known issues, see the Release Notes.
Chapter 6.
TENSORFLOW PARAMETERS

The TensorFlow container in the NGC container registry (nvcr.io) comes pre-configured as defined by the following parameters. These parameters are used to pre-compile GPUs, enable support for the Accelerated Linear Algebra (XLA) backend, and disable support for Google Cloud Platform (GCP) and the Hadoop Distributed File System (HDFS).

6.1. Added And Modified Parameters

In addition to the parameters within the Dockerfile that is included in the Google TensorFlow container, the following parameters have either been added for modified with the NVIDIA version of TensorFlow.

For parameters not mentioned in this guide, see the TensorFlow documentation.

6.1.1. TF_CUDA_COMPUTE_CAPABILITIES

The TF_CUDA_COMPUTE_CAPABILITIES parameter enables the code to be pre-compiled for specific GPU architectures.

The container comes built with the following setting, which targets Pascal, Volta, and Turing GPUs:

```
TF_CUDA_COMPUTE_CAPABILITIES "6.0,6.1,7.0,7.5"
```

Where the numbers correspond to GPU architectures:

- 6.0+6.1: Pascal
- 7.0: Volta
- 7.5: Turing

6.1.2. TF_NEED_GCP
The TF_NEED_GCP parameter, as defined, disables support for the Google Cloud Platform (GCP).

The container comes built with the following setting, which turns off support for GCP:

```
TF_NEED_GCP 0
```

### 6.1.3. TF_NEED_HDFS

The TF_NEED_HDFS parameter, as defined, disables support for the Hadoop Distributed File System (HDFS).

The container comes built with the following setting, which turns off support for HDFS:

```
TF_NEED_HDFS 0
```

### 6.1.4. TF_ENABLE_XLA

The TF_ENABLE_XLA parameter, as defined, enables support for the Accelerated Linear Algebra (XLA) backend.

The container comes built with the following setting, which turns on support for XLA:

```
TF_ENABLE_XLA 1
```
Chapter 7.
TENSORFLOW ENVIRONMENT VARIABLES

The following environment variable settings enable certain features within TensorFlow. They change and reduce the precision of the computation slightly and are enabled by default.

7.1. Added Or Modified Variables

In addition to the variables within the Dockerfile that are included in the Google TensorFlow container, the following variables have either been added or modified with the NVIDIA version of TensorFlow.

For variables not mentioned in this guide, see the TensorFlow documentation.

7.1.1. TF_ADJUST_HUE_FUSED

The TF_ADJUST_HUE_FUSED variable enables the use of fused kernels for the image hue.

This variable is enabled by default:

```
TF_ADJUST_HUE_FUSED 1
```

To disable the variable, run the following command:

```
export TF_ADJUST_HUE_FUSED=0
```

7.1.2. TF_ADJUST_SATURATION_FUSED

The TF_ADJUST_SATURATION_FUSED variable enables the use of fused kernels for the saturation adjustment.

This variable is enabled by default:

```
TF_ADJUST_SATURATION_FUSED 1
```

To disable the variable, run the following command:
export TF_ADJUST_SATURATION_FUSED=0

7.1.3. **TF_ENABLE_WINOGRAD_NONFUSED**

The *TF_ENABLE_WINOGRAD_NONFUSED* variable enables the use of the non-fused Winograd convolution algorithm.

This variable is enabled by default:

```
TF_ENABLE_WINOGRAD_NONFUSED 1
```

To disable the variable, run the following command:

```
export TF_ENABLE_WINOGRAD_NONFUSED=0
```

7.1.4. **TF_AUTOTUNE_THRESHOLD**

The *TF_AUTOTUNE_THRESHOLD* variable improves the stability of the auto-tuning process used to select the fastest convolution algorithms. Setting it to a higher value improves stability, but requires a larger number of trial steps at the beginning of training before the best algorithms are found.

Within the container, this variable is set to the following:

```
export TF_AUTOTUNE_THRESHOLD=2
```

To set this variable to its default setting, run the following command:

```
export TF_AUTOTUNE_THRESHOLD=1
```

7.1.5. **CUDA_DEVICE_MAX_CONNECTIONS**

The *CUDA_DEVICE_MAX_CONNECTIONS* variable solves performance issues related to streams on Tesla K80 GPUs.

Within the container, this variable is set to the following:

```
export CUDA_DEVICE_MAX_CONNECTIONS=12
```

To set this variable to its default setting, run the following command:

```
export CUDA_DEVICE_MAX_CONNECTIONS=8
```

7.1.6. **TF_DISABLE_CUDNN_TENSOR_OP_MATH**

The *TF_DISABLE_CUDNN_TENSOR_OP_MATH* variable enables and disables Tensor Core math for cuDNN convolutions in TensorFlow. Tensor Core math is enabled by default, but can be disabled by setting this variable to 1. For more information, see [Tensor Core Math](#).

This variable is disabled by default:

```
export TF_DISABLE_CUDNN_TENSOR_OP_MATH=0
```

To enable the variable, run the following command:
7.1.7. **TF_DISABLE_CUDNN_RNN_TENSOR_OP_MATH**

The `TF_DISABLE_CUDNN_RNN_TENSOR_OP_MATH` variable enables and disables Tensor Core math for cuDNN RNNs in TensorFlow. Tensor Core math is enabled by default, but can be disabled by setting this variable to 1. For more information, see Tensor Core Math.

This variable is disabled by default:

```bash
export TF_DISABLE_CUDNN_RNN_TENSOR_OP_MATH=1
```

To enable the variable, run the following command:

```bash
export TF_DISABLE_CUDNN_RNN_TENSOR_OP_MATH=0
```

7.1.8. **TF_DISABLE_CUBLAS_TENSOR_OP_MATH**

The `TF_DISABLE_CUBLAS_TENSOR_OP_MATH` variable enables and disables Tensor Core math for cuBLAS convolutions in TensorFlow. Tensor Core math is enabled by default, but can be disabled by setting this variable to 1. For more information, see Tensor Core Math.

This variable is disabled by default:

```bash
export TF_DISABLE_CUBLAS_TENSOR_OP_MATH=1
```

To enable the variable, run the following command:

```bash
export TF_DISABLE_CUBLAS_TENSOR_OP_MATH=0
```

7.1.9. **TF_ENABLE_CUBLAS_TENSOR_OP_MATH_FP32**

The `TF_ENABLE_CUBLAS_TENSOR_OP_MATH_FP32` variable enables and disables Tensor Core math for float32 matrix multiplication operations in TensorFlow. Tensor Core math for float32 operations is disabled by default, but can be enabled by setting this variable to 1. For more information, see Tensor Core Math.

This variable is disabled by default:

```bash
export TF_ENABLE_CUBLAS_TENSOR_OP_MATH_FP32=0
```

To enable this variable, run the following command:

```bash
export TF_ENABLE_CUBLAS_TENSOR_OP_MATH_FP32=1
```

7.1.10. **TF_ENABLE_CUDNN_TENSOR_OP_MATH_FP32**

The `TF_ENABLE_CUDNN_TENSOR_OP_MATH_FP32` variable enables and disables Tensor Core math for float32 convolution operations in TensorFlow. Tensor Core math for float32 operations is disabled by default, but can be enabled by setting this variable to 1. For more information, see Tensor Core Math.
This variable is disabled by default:

```
export TF_ENABLE_CUDNN_TENSOR_OP_MATH_FP32=0
```

To enable this variable, run the following command:

```
export TF_ENABLE_CUDNN_TENSOR_OP_MATH_FP32=1
```

### 7.1.11. **TF_ENABLE_CUDNN_RNN_TENSOR_OP_MATH_FP32**

The `TF_ENABLE_CUDNN_RNN_TENSOR_OP_MATH_FP32` variable enables and disables Tensor Core math for float32 cuDNN RNN operations in TensorFlow. Tensor Core math for float32 operations is disabled by default, but can be enabled by setting this variable to 1. For more information, see Tensor Core Math.

This variable is disabled by default:

```
export TF_ENABLE_CUDNN_RNN_TENSOR_OP_MATH_FP32=0
```

To enable this variable, run the following command:

```
export TF_ENABLE_CUDNN_RNN_TENSOR_OP_MATH_FP32=1
```

### 7.1.12. **TF_DISABLE_NVTX_RANGES**

The `TF_DISABLE_NVTX_RANGES` variable enables and disables NVTX ranges in TensorFlow. NVTX ranges add operation name annotations to the execution timeline when profiling an application with Nsight Systems or the NVIDIA Visual Profiler. These NVTX ranges are enabled by default, but can be disabled by setting this variable to 1. For more information on NVTX, see CUDA Toolkit Documentation: NVIDIA Tools Extension.

This variable is disabled by default:

```
export TF_DISABLE_NVTX_RANGES=0
```

To enable this variable, run the following command:

```
export TF_DISABLE_NVTX_RANGES=1
```

### 7.1.13. **TF_ENABLE_NHWC**

The `TF_ENABLE_NHWC` variable enables and disables the NHWC plumbing in TensorFlow. The NHWC plumbing applies operations directly on data with NHWC (or channels_last) format by skipping the layout optimizer in the grappler pass and eliminating layout transposes per operation. It is disabled by default, but can be enabled by setting this variable to 1. Enabling it can decrease the number of unnecessary data layout transposes.

This variable is disabled by default:

```
export TF_ENABLE_NHWC=0
```

To enable this variable, run the following command:
export TF_ENABLE_NHWC=1

7.1.14. TF_CUDNN_CTC_LOSS

The **TF_CUDNN_CTC_LOSS** variable enables the cuDNN CTC loss backend via `nn.ctc_loss` for Tensorflow 2.x (for example, *19.11-tf2-py3*) or `nn.ctc_loss_v2` for Tensorflow 1.x (for example, *19.11-tf1-py3*).

This variable is disabled by default:

```
export TF_CUDNN_CTC_LOSS=0
```

To enable this variable, run the following command:

```
export TF_CUDNN_CTC_LOSS=1
```
Chapter 8. PERFORMANCE

To achieve optimum TensorFlow performance, for image based training, the container includes sample scripts that demonstrate efficient training of CNNs. The sample scripts may need to be modified to fit your application. The scripts can be found in the /opt/tensorflow/nvidia-examples/cnn/ directory. Along with the training scripts, there is also some documentation that can be found in the /opt/tensorflow/nvidia-examples/cnn/README.md directory.

For more information, see Performance models and Benchmarks.

8.1. Tensor Core Math

The TensorFlow container includes support Tensor Cores starting in Volta’s architecture, available on Tesla V100 GPUs. Tensor Cores deliver up to 12x higher peak TFLOPs for training. The container enables Tensor Core math by default; therefore, any models containing convolutions or matrix multiplies using the tf.float16 data type will automatically take advantage of Tensor Core hardware whenever possible.

Tensor Core math can also be enabled for tf.float32 matrix multiply, convolution, and RNN operations by setting the TF_ENABLE_CUBLAS_TENSOR_OP_MATH_FP32=1, TF_ENABLE_CUDNN_TENSOR_OP_MATH_FP32=1, and TF_ENABLE_CUDNN_RNN_TENSOR_OP_MATH_FP32=1 (for RNNs that use the cudnn_rnno p) environment variables, respectively. This mode causes data to be internally reduced to float16 precision, which may affect training convergence.

With Tensor Core math enabled, inputs of matrix multiply, convolution, and RNN operations are implicitly down-cast from FP32 to FP16. Internal accumulation and outputs remain in FP32. This allows FP32 models to run faster by using GPU Tensor Cores when available. Additionally, users should augment models to include loss scaling (for example, by wrapping the optimizer in a tf.contrib.mixed_precision.loss_scale_optimizer).

For more information about the architecture, see Inside Volta and Inside Turing.

8.1.1. Float16 Training
Training with reduced precision can in some cases lead to poor or unstable convergence. NVIDIA recommends the following strategies to minimize the effects of reduced precision during training (see nvidia-examples/cnn/nvcnn.py for a complete demonstration of float16 training):

1. Keep trainable variables in float32 precision and cast them to float16 before using them in the model. For example:

   ```python
   tf.cast(tf.get_variable(..., dtype=tf.float32), tf.float16)
   ```

2. Apply loss-scaling if the model struggles or fails to converge. Loss scaling involves multiplying the loss by a scale factor before computing gradients and then dividing the resulting gradients by the same scale again to re-normalize them. A typical loss scale factor for recurrent neural network models is 128. For example:

   ```python
   loss, params = ...
   scale = 128
   grads = [grad / scale for grad in tf.gradients(loss * scale, params)]
   ```

### 8.2. Automatic Mixed Precision (AMP)

Using **mixed precision training** requires three steps:

1. Converting the model to use the float16 data type where possible.
2. Keeping float32 master weights to accumulate per-iteration weight updates.
3. Using loss scaling to preserve small gradient values.

Using automatic mixed precision with the TensorFlow framework can be as simple as adding one line of code. It accomplishes this by automatically rewriting all computation graphs with the necessary operations to enable mixed precision training and loss scaling. See [Automatic Mixed Precision for Deep Learning](https://www.nvidia.com) for more information.

#### 8.2.1. Automatic Mixed Precision Training In TensorFlow

For models already using an optimizer from `tf.train` or `tf.keras.optimizers` for both `compute_gradients()` and `apply_gradients()` operations (for example, by calling `optimizer.minimize()` or `model.fit()`), automatic mixed precision can be enabled by wrapping the optimizer with `tf.train.experimental.enable_mixed_precision_graph_rewrite()`.

**Graph-based example:**

```python
opt = tf.train.AdamOptimizer()
opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)
train_op = opt.minimize(loss)
```

**Keras-based example:**

```python
opt = tf.keras.optimizers.Adam()
opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)
model.compile(loss=loss, optimizer=opt)
model.fit(...)  
```

For more information on this function, see the TensorFlow documentation [here](https://www.nvidia.com).
For backward compatibility with previous container releases, AMP can also be enabled for **tf.train** optimizers by defining the following environment variable:

```
export TF_ENABLE_AUTO_MIXED_PRECISION=1
```

When enabled, automatic mixed precision will do two things:

1. Insert the appropriate cast operations into your TensorFlow graph to use float16 execution and storage where appropriate -- this enables the use of Tensor Cores along with memory storage and bandwidth savings.
2. Turn on **automatic loss scaling** inside the training Optimizer object.

### 8.2.2. Conditions And Limitations

Ensure you are familiar with the following conditions:

**Additional control**

It is possible to enable the automatic insertion of cast operations without automatic loss scaling. The environment variable for doing so is

```
TF_ENABLE_AUTO_MIXED_PRECISION_GRAPH_REWRITE=1
```

**Caveats**

**Model types**

Convolutional architectures that rely primarily on grouped or depth-separable convolutions (MobileNet and ResNeXt are popular examples) will not presently see speedups from float16 execution. This is due to the library constraints outside the scope of automatic mixed precision, though we expect them to be relaxed soon.

**Optimizers**

Automatic mixed precision loss scaling requires that the model code use a subclass of the built-in **tf.train.optimizer** or **tf.keras.optimizers.optimizer** classes. Furthermore:

- TensorFlow code that directly calls **tf.gradients** and uses those gradients “by hand” will not be supported.
- Instead, automatic mixed precision requires the paired calls to **optimizer.compute_gradients** and **optimizer.apply_gradients**, or a call to the high-level function **optimizer.minimize**.
- If the optimizer class is a custom subclass of **tf.train.optimizer** (not one built into TensorFlow), then it may not be supported by automatic mixed precision loss scaling. In particular, if the custom subclass overrides either **compute_gradients** or **apply_gradients**, it must take care to also call into the superclass implementations of those methods.
Multi-GPU

Prior to TensorFlow 1.14.0, automatic mixed precision did not support TensorFlow "Distributed Strategies." Instead, multi-GPU training needed to use Horovod (or TensorFlow device primitives).

Other notes

If your code already has automatic loss scaling support built-in, it will need to be disabled in order to avoid conflicting with automatic mixed precision own automatic loss scaling. Alternatively, the automatic mixed precision graph rewrite can be enabled without enabling loss scaling by using the option described above.

8.2.3. FAQs

Q: What if my model code already supports mixed precision training?

If the code is already written in such a way to follow the Mixed Precision Training Guide, then automatic mixed precision will leave things as they are. For example, the CNN examples provided inside the NVIDIA TensorFlow container use mixed precision training by default. If you would like to evaluate how they work with automatic mixed precision, be sure to run them with the flag `--precision=fp32`.

Q: How much faster will my model run with automatic mixed precision?

There are no precise rules for mixed precision speedups, but here are a few guidelines:

- The more time is spent in matrix multiplication (dense layers) or convolutions, the more Tensor Cores can accelerate the model. This means that “bigger” models often see larger speedups. In particular, very small dense and convolution layers will see limited benefit from automatic mixed precision, since there is not enough math to fully exploit Tensor Cores.
- Mixed precision models use less memory than FP32, so it is possible to increase the batch size when running with automatic mixed precision. Thus, you can often increase the speedup by increasing the batch size after enabling automatic mixed precision.

Q: How can I see what changes automatic mixed precision makes to my model?

Because automatic mixed precision operates at the level of TensorFlow graphs, it can be challenging to quickly grasp the changes it makes: often it will tweak thousands of TensorFlow operations, but those correspond to many fewer logical layers. You can set the environment variable `TF_CPP_VMODULE="auto_mixed_precision=2"` to see a full log of the decisions automatic mixed precision makes (note that this may generate a lot of output).
Q: Why do I see only FP32 datatypes in my saved model GraphDef?

When you save a model graph or inspect the graph with `Session.graph` for `Session.graph_def`, TensorFlow returns the unoptimized version of the graph. Automatic mixed precision works as an optimization pass over the original graph, so its changes are not included in the unoptimized graph. You can set the environment variable `TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_LOG_PATH="my/log/path"`, and automatic mixed precision will save out pre- and post-optimization copies of each graph it processes to that directory.

Q: Why do I see step=0 repeated multiple times when training with automatic mixed precision?

The automatic loss scaling algorithm that automatic mixed precision enables can choose to “skip” training iterations as it searches for the optimal loss scale. When it does so, it does not increment the global step count. Since most of the skips occur at the beginning of training (usually fewer than ten iterations), this behavior manifests as multiple iterations where the step counter stays at zero.

Q: How are user-defined custom TensorFlow operations handled?

By default, automatic mixed precision will leave alone any op types it doesn’t know about, including custom operations. That means the types of the op’s inputs and outputs are not changed, and automatic mixed precision will insert casts as necessary to interoperate with the rest of the (possibly-changed) graph.

If you would like to make automatic mixed precision aware of a custom op type, there are three environment variables you can use:

- **TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_WHITELIST_ADD**
  These are ops for which it is worth casting the inputs to FP16 to get FP16 execution. Mostly, they are ops that can take advantage of Tensor Cores.

- **TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_GRAYLIST_ADD**
  These are ops for which FP16 execution is available, so they can use FP16 if the inputs happen to already be in FP16 because of an upstream WHITELIST op.

- **TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_BLACKLIST_ADD**
  These are ops for which FP32 is necessary for numerical precision, and the outputs are not safe to cast back to FP16. Example ops include `Exp` and `Log`.

Each of these environment variables takes a comma-separated list of string op names. For example, you might set export

```
TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_WHITELIST_ADD=MyOp1,MyOp2
```

The op name is the string name used in the call to `REGISTER_OP`, which corresponds to the name attribute on the operation’s `OpDef`.
Q: Can I change the algorithmic behavior of automatic mixed precision?

The primary lever for controlling automatic mixed precision behavior is to manipulate what ops lie on each of the white, gray, and blacklists. You can add ops to each using the three environment variables above, and there is a corresponding variable `TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_{WHITELIST,GRAYLIST,BLACKLIST}_REMOVE` to take built-in ops off of each list.
Chapter 9.
TROUBLESHOOTING

9.1. Support

For more information about TensorFlow, including tutorials, documentation, and examples, see:

- TensorFlow tutorials
- TensorFlow API
- Automatic mixed precision
- Tensor Cores
- Deep Learning Examples optimized for Tensor Cores

For the latest TensorFlow Release Notes, see the Deep Learning Documentation website.
Notice

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

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

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

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

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, cuDNN, DALI, DIGITS, DGX, DGX-1, DGX-2, DGX Station, DLProf, Jetson, Kepler, Maxwell, NCCL, Nsight Compute, Nsight Systems, NvCaffe, PerfWorks, Pascal, SDK Manager, Tegra, TensorRT, TensorRT 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

© 2019 NVIDIA Corporation. All rights reserved.

www.nvidia.com