

# Optimize TensorFlow GPU performance with the TensorFlow Profiler

## Overview

This guide will show you how to use the TensorFlow Profiler with TensorBoard to gain insight into and get the maximum performance out of your GPUs, and debug when one or more of your GPUs are underutilized.

If you are new to the Profiler:

- Get started with the [TensorFlow Profiler: Profile model performance](https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras) ([https://www.tensorflow.org/tensorboard/tensorboard\\_profiling\\_keras](https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras)) notebook with a Keras example and [TensorBoard](https://www.tensorflow.org/tensorboard) (<https://www.tensorflow.org/tensorboard>).
- Learn about various profiling tools and methods available for optimizing TensorFlow performance on the host (CPU) with the [Optimize TensorFlow performance using the Profiler](https://www.tensorflow.org/guide/profiler#profiler_tools) ([https://www.tensorflow.org/guide/profiler#profiler\\_tools](https://www.tensorflow.org/guide/profiler#profiler_tools)) guide.

Keep in mind that offloading computations to GPU may not always be beneficial, particularly for small models. There can be overhead due to:

- Data transfer between the host (CPU) and the device (GPU); and
- Due to the latency involved when the host launches GPU kernels.

## Performance optimization workflow

This guide outlines how to debug performance issues starting with a single GPU, then moving to a single host with multiple GPUs.

It is recommended to debug performance issues in the following order:

1. Optimize and debug the performance on one GPU:
  1. Check if the input pipeline is a bottleneck.
  2. Debug the performance of one GPU.
  3. Enable mixed precision (with `fp16` (float16)) and optionally enable [XLA](https://www.tensorflow.org/xla) (<https://www.tensorflow.org/xla>).
2. Optimize and debug the performance on the multi-GPU single host.

For example, if you are using a TensorFlow [distribution strategy](https://www.tensorflow.org/guide/distributed_training) ([https://www.tensorflow.org/guide/distributed\\_training](https://www.tensorflow.org/guide/distributed_training)) to train a model on a single host with multiple GPUs and notice suboptimal GPU utilization, you should first optimize and debug the performance for one GPU before debugging the multi-GPU system.

As a baseline for getting performant code on GPUs, this guide assumes you are already using `tf.function`. The Keras `Model.compile` and `Model.fit` APIs will utilize `tf.function` automatically under the hood. When writing a custom training loop with `tf.GradientTape`, refer to the [Better performance with `tf.function`](https://www.tensorflow.org/guide/function) (<https://www.tensorflow.org/guide/function>) on how to enable `tf.function`s.

The next sections discuss suggested approaches for each of the scenarios above to help identify and fix performance bottlenecks.

## 1. Optimize the performance on one GPU

In an ideal case, your program should have high GPU utilization, minimal CPU (the host) to GPU (the device) communication, and no overhead from the input pipeline. The first step in analyzing the performance is to get a profile for a model running with one GPU.

TensorBoard's Profiler [overview page](https://www.tensorflow.org/guide/profiler#overview_page) ([https://www.tensorflow.org/guide/profiler#overview\\_page](https://www.tensorflow.org/guide/profiler#overview_page))—which shows a top level view of how your model performed during a profile run—can provide an idea of how far away your program is from the ideal scenario.



## Summary of input-pipeline analysis

Your program is **HIGHLY** input-bound because 81.4% of the total step time sampled is waiting for input. Therefore, you should first focus on reducing the input time.

### Recommendation for next step:

Look at Section 3 for the breakdown of input time on the host.

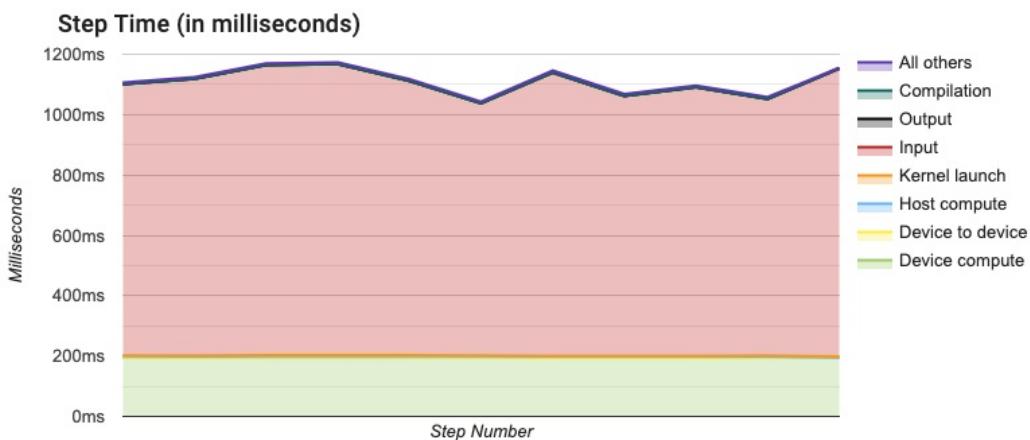
## Device-side analysis details

### Device step time

#### Device step-time statistics

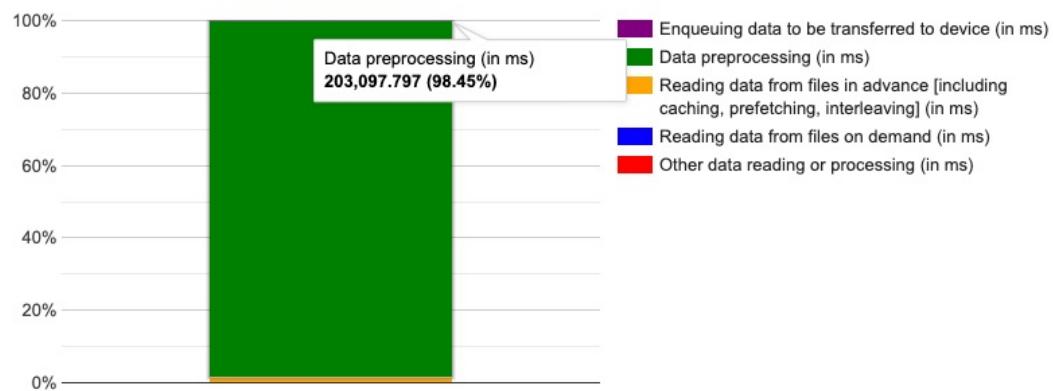
Average: 1114.5 ms ( $\sigma = 44.8$  ms)

Range: 1043.4 - 1173.6 ms



## Host-side analysis details

### Breakdown of input processing time on the host



### What can be done to reduce above components of the host input time:

Enqueuing data: you may want to combine small input data chunks into fewer but larger chunks.

Data preprocessing: you may increase num\_parallel\_calls in [Dataset map\(\)](#) or preprocess the data OFFLINE.

Reading data from files in advance: you may tune parameters in the following tf.data API ([prefetch size](#), [interleave cycle\\_length](#), [reader buffer\\_size](#))

Reading data from files on demand: you should read data IN ADVANCE using the following tf.data API ([prefetch](#), [interleave](#), [reader buffer](#))

Other data reading or processing: you may consider using the [tf.data API](#) (if you are not using it now)

You can take the following potential actions if your input-pipeline contributes significantly to step time:

- You can use the `tf.data`-specific [guide](https://www.tensorflow.org/guide/data_performance_analysis) ([https://www.tensorflow.org/guide/data\\_performance\\_analysis](https://www.tensorflow.org/guide/data_performance_analysis)) to learn how to debug your input pipeline.
- Another quick way to check if the input pipeline is the bottleneck is to use randomly generated input data that does not need any pre-processing. [Here is an example](#) ([https://github.com/tensorflow/models/blob/4a5770827edf1c3974274ba3e4169d0e5ba7478a/official/vision/image\\_classification/resnet/resnet\\_runnable.py#L50-L57](https://github.com/tensorflow/models/blob/4a5770827edf1c3974274ba3e4169d0e5ba7478a/official/vision/image_classification/resnet/resnet_runnable.py#L50-L57)) of using this technique for a ResNet model. If the input pipeline is optimal, you should experience similar performance with real data and with generated random/synthetic data. The only overhead in the synthetic data case will be due to input data copy which again can be prefetched and optimized.

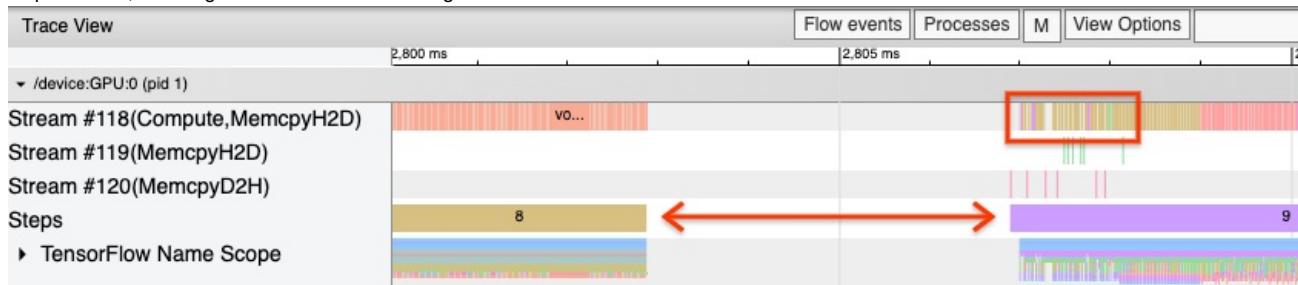
In addition, refer to the [best practices for optimizing the input data pipeline](#) ([https://www.tensorflow.org/guide/profiler#optimize\\_the\\_input\\_data\\_pipeline](https://www.tensorflow.org/guide/profiler#optimize_the_input_data_pipeline)).

## 2. Debug the performance of one GPU

There are several factors that can contribute to low GPU utilization. Below are some scenarios commonly observed when looking at the [trace viewer](https://www.tensorflow.org/guide/profiler#trace_viewer) ([https://www.tensorflow.org/guide/profiler#trace\\_viewer](https://www.tensorflow.org/guide/profiler#trace_viewer)) and potential solutions.

## 1. Analyze gaps between steps

A common observation when your program is not running optimally is gaps between training steps. In the image of the trace view below, there is a large gap between steps 8 and 9, meaning that the GPU is idle during that time.



If your trace viewer shows large gaps between steps, this could be an indication that your program is input bound. In that case you should refer to the previous section on debugging your input pipeline if you have not already done so.

However, even with an optimized input pipeline, you can still have gaps between the end of one step and the start of another due to CPU thread contention. `tf.data` makes use of background threads to parallelize pipeline processing. These threads may interfere with GPU host-side activity that happens at the beginning of each step, such as copying data or scheduling GPU operations.

If you notice large gaps on the host side, which schedules these ops on the GPU, you can set the environment variable `TF_GPU_THREAD_MODE=gpu_private`. This ensures that GPU kernels are launched from their own dedicated threads, and don't get queued behind `tf.data` work.

Gaps between steps can also be caused by metric calculations, Keras callbacks, or ops outside of `tf.function` that run on the host. These ops don't have as good performance as the ops inside a TensorFlow graph. Additionally, some of these ops run on the CPU and copy tensors back and forth from the GPU.

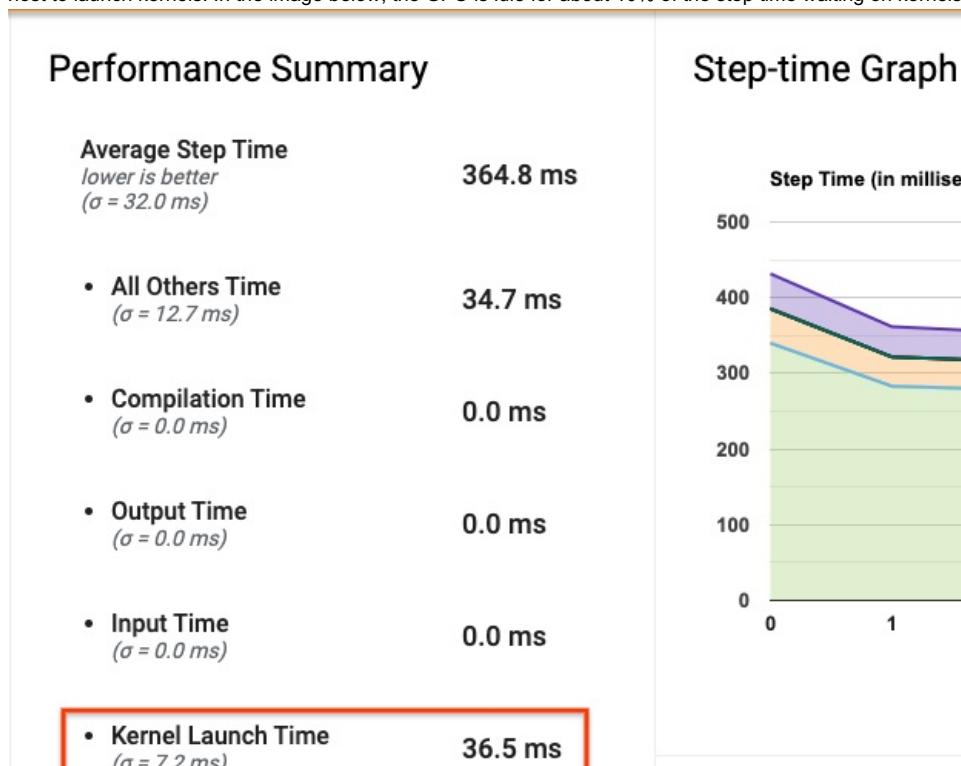
If after optimizing your input pipeline you still notice gaps between steps in the trace viewer, you should look at the model code between steps and check if disabling callbacks/metrics improves performance. Some details of these ops are also on the trace viewer (both device and host side). The recommendation in this scenario is to amortize the overhead of these ops by executing them after a fixed number of steps instead of every step. When using the `compile` method in the `tf.keras` API, setting the `experimental_steps_per_execution` flag does this automatically. For custom training loops, use `tf.while_loop`.

## 2. Achieve higher device utilization

### 1. Small GPU kernels and host kernel launch delays

The host enqueues kernels to be run on the GPU, but there is a latency (around 20-40 µs) involved before kernels are actually executed on the GPU. In an ideal case, the host enqueues enough kernels on the GPU such that the GPU spends most of its time executing, rather than waiting on the host to enqueue more kernels.

The Profiler's [overview page](https://www.tensorflow.org/guide/profiler#overview_page) ([https://www.tensorflow.org/guide/profiler#overview\\_page](https://www.tensorflow.org/guide/profiler#overview_page)) on TensorBoard shows how much time the GPU was idle due to waiting on the host to launch kernels. In the image below, the GPU is idle for about 10% of the step time waiting on kernels to be launched.



### Recommendation

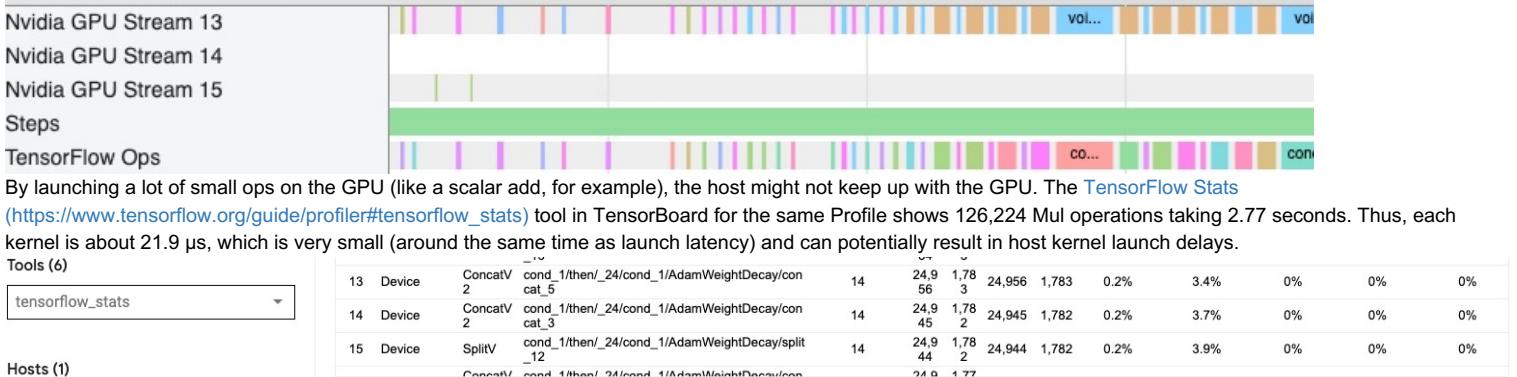
- Your program is POTENTIALLY waiting on others' time (which could be GPU compute time).
- 10.0 % of the total step time is spent in `tf.data`. In this case, you can try to reduce the number of steps per execution.

Tool troubleshooting / FAQ

- Refer to the TF2 Profiler documentation for more information.

### Next tools to use for redundancy

The [trace viewer](https://www.tensorflow.org/guide/profiler#trace_viewer) ([https://www.tensorflow.org/guide/profiler#trace\\_viewer](https://www.tensorflow.org/guide/profiler#trace_viewer)) for this same program shows small gaps between kernels where the host is busy launching kernels on the GPU.



If your trace viewer ([https://www.tensorflow.org/guide/profiler#trace\\_viewer](https://www.tensorflow.org/guide/profiler#trace_viewer)) shows many small gaps between ops on the GPU like in the image above, you can:

- Concatenate small tensors and use vectorized ops or use a larger batch size to make each launched kernel do more work, which will keep the GPU busy for longer.
- Make sure you are using `tf.function` to create TensorFlow graphs, so that you are not running ops in a pure eager mode. If you are using `Model.fit` (as oppose to a custom training loop with `tf.GradientTape`), then `tf.keras.Model.compile` will automatically do this for you.
- Fuse kernels using XLA with `tf.function(jit_compile=True)` or auto-clustering. For more details, go to the [Enable mixed precision and XLA](#) section below to learn how to enable XLA to get higher performance. This feature can lead to high device utilization.

## 2. TensorFlow op placement

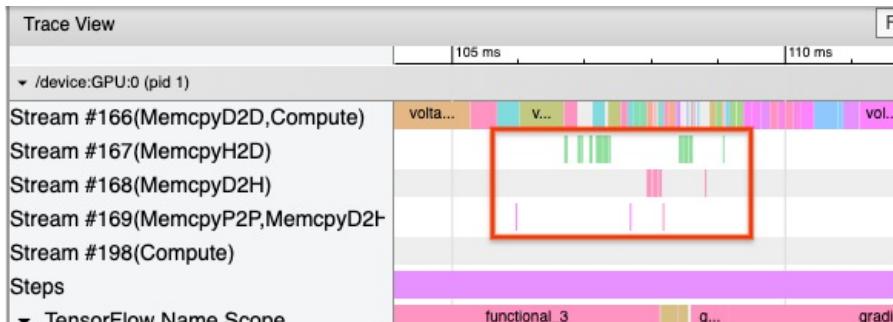
The Profiler overview page ([https://www.tensorflow.org/guide/profiler#overview\\_page](https://www.tensorflow.org/guide/profiler#overview_page)) shows you the percentage of ops placed on the host vs. the device (you can also verify the placement of specific ops by looking at the trace viewer ([https://www.tensorflow.org/guide/profiler#trace\\_viewer](https://www.tensorflow.org/guide/profiler#trace_viewer))). Like in the image below, you want the percentage of ops on the host to be very small compared to the device.



Ideally, most of the compute intensive ops should be placed on the GPU.

To find out which devices the operations and tensors in your model are assigned to, set `tf.debugging.set_log_device_placement(True)` as the first statement of your program.

Note that in some cases, even if you specify an op to be placed on a particular device, its implementation might override this condition (example: `tf.unique`). Even for single GPU training, specifying a distribution strategy, such as `tf.distribute.OneDeviceStrategy`, can result in more deterministic placement of ops on your device. One reason for having the majority of ops placed on the GPU is to prevent excessive memory copies between the host and the device (memory copies for model input/output data between host and device are expected). An example of excessive copying is demonstrated in the trace view below on GPU streams #167, #168, and #169.



These copies can sometimes hurt the performance if they block GPU kernels from executing. Memory copy operations in the trace viewer ([https://www.tensorflow.org/guide/profiler#trace\\_viewer](https://www.tensorflow.org/guide/profiler#trace_viewer)) have more information about the ops that are the source of these copied tensors, but it might not always be easy to associate a memCopy with an op. In these cases, it is helpful to look at the ops nearby to check if the memory copy happens at the same location in every step.

## 3. More efficient kernels on GPUs

Once your program's GPU utilization is acceptable, the next step is to look into increasing the efficiency of the GPU kernels by utilizing Tensor Cores or fusing ops.

### 1. Utilize Tensor Cores

Modern NVIDIA® GPUs have specialized [Tensor Cores](https://www.nvidia.com/en-gb/data-center/tensor-cores/) (<https://www.nvidia.com/en-gb/data-center/tensor-cores/>) that can significantly improve the performance of eligible kernels.

You can use TensorBoard's [GPU kernel stats](https://www.tensorflow.org/guide/profiler#gpu_kernel_stats) ([https://www.tensorflow.org/guide/profiler#gpu\\_kernel\\_stats](https://www.tensorflow.org/guide/profiler#gpu_kernel_stats)) to visualize which GPU kernels are Tensor Core-eligible, and which kernels are using Tensor Cores. Enabling `fp16` (see [Enabling Mixed Precision](#) section below) is one way to make your program's General Matrix Multiply (GEMM) kernels (matmul ops) utilize the Tensor Core. GPU kernels use the Tensor Cores efficiently when the precision is `fp16` and input/output tensor dimensions are divisible by 8 or 16 (for `int8`).

Note: With cuDNN v7.6.3 and later, convolution dimensions will automatically be padded where necessary to leverage Tensor Cores.

For other detailed recommendations on how to make kernels efficient for GPUs, refer to the [NVIDIA® deep learning performance](https://docs.nvidia.com/deeplearning/performance/index.html#perf-guidelines) (<https://docs.nvidia.com/deeplearning/performance/index.html#perf-guidelines>) guide.

### 2. Fuse ops

Use `tf.function(jit_compile=True)` to fuse smaller ops to form bigger kernels leading to significant performance gains. To learn more, refer to the [XLA](#) (<https://www.tensorflow.org/xla>) guide.

### 3. Enable mixed precision and XLA

After following the above steps, enabling mixed precision and XLA are two optional steps you can take to improve performance further. The suggested approach is to enable them one by one and verify that the performance benefits are as expected.

#### 1. Enable mixed precision

The TensorFlow Mixed precision ([https://www.tensorflow.org/guide/keras/mixed\\_precision](https://www.tensorflow.org/guide/keras/mixed_precision)) guide shows how to enable `fp16` precision on GPUs. Enable AMP (<https://developer.nvidia.com/automatic-mixed-precision>) on NVIDIA® GPUs to use Tensor Cores and realize up to 3x overall speedups when compared to using just `fp32` (float32) precision on Volta and newer GPU architectures.

Make sure that matrix/tensor dimensions satisfy requirements for calling kernels that use Tensor Cores. GPU kernels use the Tensor Cores efficiently when the precision is `fp16` and input/output dimensions are divisible by 8 or 16 (for `int8`).

Note that with cuDNN v7.6.3 and later, convolution dimensions will automatically be padded where necessary to leverage Tensor Cores.

Follow the best practices below to maximize the performance benefits of `fp16` precision.

##### 1. Use optimal `fp16` kernels

With `fp16` enabled, your program's matrix multiplications (GEMM) kernels, should use the corresponding `fp16` version that utilizes the Tensor Cores. However, in some cases, this does not happen and you do not experience the expected speedup from enabling `fp16`, as your program falls back to the inefficient implementation instead.

##### Tools (6)

kernel\_stats

### GPU Kernels

Kernel Name Op Name

#### Hosts (1)

localhost

Rank	Kernel Name	Registers per thread	Shared Mem bytes	Block dim	Grid dim	Kernel uses TensorCore	Op is TensorCore eligible
1	nccAllReduceRingLLK ernel_sum_f32(nccCol l)	0	0	1,1,1	1,1, 1	x	x
2	volta_fp16_s884gemm _fp16_256x128_ldg8_f 2f_nn	0	0	1,1,1	1,1, 1	✓	✓
3	volta_fp16_s884gemm _fp16_256x128_ldg8_f 2f_nn	0	0	1,1,1	1,1, 1	✓	✓

The [GPU kernel](https://www.tensorflow.org/guide/profiler#gpu_kernel_stats) ([https://www.tensorflow.org/guide/profiler#gpu\\_kernel\\_stats](https://www.tensorflow.org/guide/profiler#gpu_kernel_stats)) stats page shows which ops are Tensor Core eligible and which kernels are actually using the efficient Tensor Core. The [NVIDIA® guide on deep learning performance](https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html#opt-tensor-cores) (<https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html#opt-tensor-cores>) contains additional suggestions on how to leverage Tensor Cores. Additionally, the benefits of using `fp16` will also show in kernels that were previously memory bound, as now the ops will take half the time.

#### 2. Dynamic vs. static loss scaling

Loss scaling is necessary when using `fp16` to prevent underflow due to low precision. There are two types of loss scaling, dynamic and static, both of which are explained in greater detail in the [Mixed Precision guide](https://www.tensorflow.org/guide/keras/mixed_precision) ([https://www.tensorflow.org/guide/keras/mixed\\_precision](https://www.tensorflow.org/guide/keras/mixed_precision)). You can use the `mixed_float16` policy to automatically enable loss scaling within the Keras optimizer.

Note: The Keras mixed precision API defaults to evaluating standalone softmax ops (ops not part of a Keras loss function) as `fp16` which can lead to numerical issues and poor convergence. Cast such ops to `fp32` for optimal performance.

When trying to optimize performance, it is important to remember that dynamic loss scaling can introduce additional conditional ops that run on the host, and lead to gaps that will be visible between steps in the trace viewer. On the other hand, static loss scaling does not have such overheads and can be a better option in terms of performance with the catch that you need to specify the correct static-loss scale value.

#### 2. Enable XLA with `tf.function(jit_compile=True)` or auto-clustering

As a final step in getting the best performance with a single GPU, you can experiment with enabling XLA, which will fuse ops and lead to better device utilization and a lower memory footprint. For details on how to enable XLA in your program with `tf.function(jit_compile=True)` or auto-clustering, refer to the [XLA](#) (<https://www.tensorflow.org/xla>) guide.

You can set the global JIT level to `-1` (off), `1`, or `2`. A higher level is more aggressive and may reduce parallelism and use more memory. Set the value to `1` if you have memory restrictions. Note that XLA does not perform well for models with variable input tensor shapes as the XLA compiler would have to keep compiling kernels whenever it encounters new shapes.

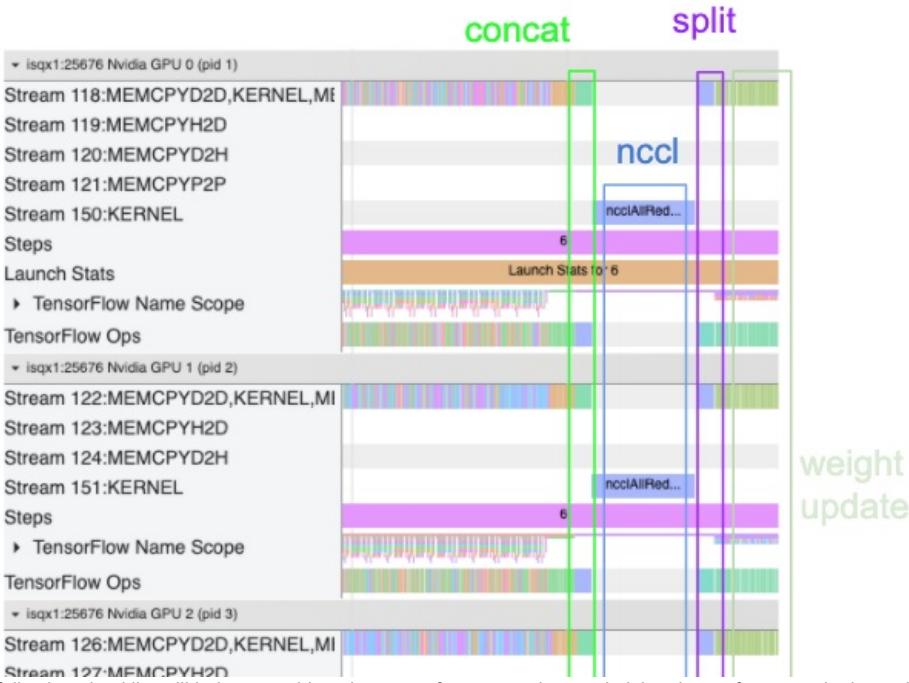
## 2. Optimize the performance on the multi-GPU single host

The `tf.distribute.MirroredStrategy` API can be used to scale model training from one GPU to multiple GPUs on a single host. (To learn more about how to do distributed training with TensorFlow, refer to the [Distributed training with TensorFlow](https://www.tensorflow.org/guide/distributed_training) ([https://www.tensorflow.org/guide/distributed\\_training](https://www.tensorflow.org/guide/distributed_training)), [Use a GPU](#) (<https://www.tensorflow.org/guide/gpu>), and [Use TPUs](https://www.tensorflow.org/guide/tpu) (<https://www.tensorflow.org/guide/tpu>) guides and the [Distributed training with Keras](#) (<https://www.tensorflow.org/tutorials/distribute/keras>) tutorial.)

Although the transition from one GPU to multiple GPUs should ideally be scalable out of the box, you can sometimes encounter performance issues.

When going from training with a single GPU to multiple GPUs on the same host, ideally you should experience the performance scaling with only the additional overhead of gradient communication and increased host thread utilization. Because of this overhead, you will not have an exact 2x speedup if you move from 1 to 2 GPUs, for example.

The trace view below shows an example of the extra communication overhead when training on multiple GPUs. There is some overhead to concatenate the gradients, communicate them across replicas, and split them before doing the weight update.



The following checklist will help you achieve better performance when optimizing the performance in the multi-GPU scenario:

1. Try to maximize the batch size, which will lead to higher device utilization and amortize the costs of communication across multiple GPUs. Using the [memory profiler](https://www.tensorflow.org/guide/profiler#memory_profile_summary) ([https://www.tensorflow.org/guide/profiler#memory\\_profile\\_summary](https://www.tensorflow.org/guide/profiler#memory_profile_summary)) helps get a sense of how close your program is to peak memory utilization. Note that while a higher batch size can affect convergence, this is usually outweighed by the performance benefits.
2. When moving from a single GPU to multiple GPUs, the same host now has to process much more input data. So, after (1), it is recommended to re-check the input pipeline performance and make sure it is not a bottleneck.
3. Check the GPU timeline in your program's trace view for any unnecessary AllReduce calls, as this results in a synchronization across all devices. In the trace view shown above, the AllReduce is done via the [NCCL](https://developer.nvidia.com/nccl) (<https://developer.nvidia.com/nccl>) kernel, and there is only one NCCL call on each GPU for the gradients on each step.
4. Check for unnecessary D2H, H2D and D2D copy operations that can be minimized.
5. Check the step time to make sure each replica is doing the same work. For example, it can happen that one GPU (typically, GPU0) is oversubscribed because the host mistakenly ends up putting more work on it.
6. Lastly, check the training step across all GPUs in your trace view for any ops that are executing sequentially. This usually happens when your program includes control dependencies from one GPU to another. In the past, debugging the performance in this situation has been solved on a case-by-case basis. If you observe this behavior in your program, [file a GitHub issue](https://github.com/tensorflow/tensorflow/issues/new/choose) (<https://github.com/tensorflow/tensorflow/issues/new/choose>) with images of your trace view.

## 1. Optimize gradient AllReduce

When training with a synchronous strategy, each device receives a portion of the input data.

After computing the forward and backwards passes through the model, the gradients calculated on each device need to be aggregated and reduced. This *gradient AllReduce* happens after the gradient calculation on each device, and before the optimizer updates the model weights.

Each GPU first concatenates the gradients across the model layers, communicates them across GPUs using `tf.distribute.CrossDeviceOps` (`tf.distribute.NcclAllReduce` is the default), and then returns the gradients after reduction per layer.

The optimizer will use these reduced gradients to update the weights of your model. Ideally, this process should happen at the same time on all GPUs to prevent any overheads.

The time to AllReduce should be approximately the same as:

$$(\text{number of parameters} * 4\text{bytes}) / (\text{communication bandwidth})$$

This calculation is useful as a quick check to understand whether the performance you have when running a distributed training job is as expected, or if you need to do further performance debugging. You can get the number of parameters in your model from `Model.summary`.

Note that each model parameter is 4 bytes in size since TensorFlow uses `fp32` (float32) to communicate gradients. Even when you have `fp16` enabled, NCCL AllReduce utilizes `fp32` parameters.

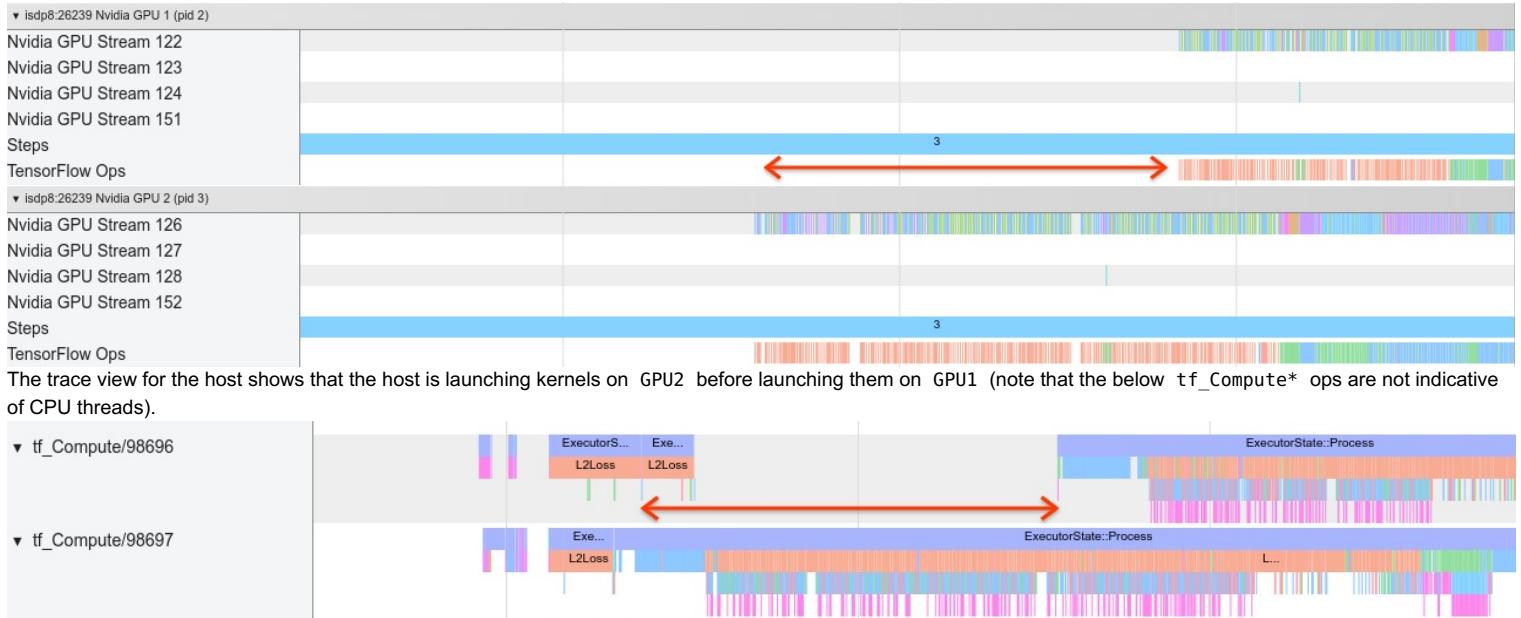
To get the benefits of scaling, the step-time needs to be much higher compared to these overheads. One way to achieve this is to use a higher batch size as batch size affects step time, but does not impact the communication overhead.

## 2. GPU host thread contention

When running multiple GPUs, the CPU's job is to keep all of the devices busy by efficiently launching GPU kernels across the devices.

However, when there are a lot of independent operations that the CPU can schedule on one GPU, the CPU can decide to use a lot of its host threads to keep one GPU busy, and then launch kernels on another GPU in a non-deterministic order. This can cause a skew or negative scaling, which can negatively affect the performance.

The [trace viewer](https://www.tensorflow.org/guide/profiler#trace_viewer) ([https://www.tensorflow.org/guide/profiler#trace\\_viewer](https://www.tensorflow.org/guide/profiler#trace_viewer)) below shows the overhead when the CPU staggers GPU kernel launches inefficiently, as GPU1 is idle and then starts running ops after GPU2 has started.



If you experience this kind of staggering of GPU kernels in your program's trace view, the recommended action is to:

- Set the TensorFlow environment variable `TF_GPU_THREAD_MODE` to `gpu_private`. This environment variable will tell the host to keep threads for a GPU private.
- By default, `TF_GPU_THREAD_MODE= gpu_private` sets the number of threads to 2, which is sufficient in most cases. However, that number can be changed by setting the TensorFlow environment variable `TF_GPU_THREAD_COUNT` to the desired number of threads.

## TensorFlow version compatibility

This document is for users who need backwards compatibility across different versions of TensorFlow (either for code or data), and for developers who want to modify TensorFlow while preserving compatibility.

### Semantic versioning 2.0

TensorFlow follows Semantic Versioning 2.0 ([semver \(http://semver.org\)](http://semver.org)) for its public API. Each release version of TensorFlow has the form `MAJOR.MINOR.PATCH`. For example, TensorFlow version 1.2.3 has `MAJOR` version 1, `MINOR` version 2, and `PATCH` version 3. Changes to each number have the following meaning:

- **MAJOR**: Potentially backwards incompatible changes. Code and data that worked with a previous major release will not necessarily work with the new release. However, in some cases existing TensorFlow graphs and checkpoints may be migratable to the newer release; see [Compatibility of graphs and checkpoints](#) for details on data compatibility.
- **MINOR**: Backwards compatible features, speed improvements, etc. Code and data that worked with a previous minor release *and* which depends only on the non-experimental public API will continue to work unchanged. For details on what is and is not the public API, see [What is covered](#).
- **PATCH**: Backwards compatible bug fixes.

For example, release 1.0.0 introduced backwards *incompatible* changes from release 0.12.1. However, release 1.1.1 was backwards *compatible* with release 1.0.0.

### What is covered

Only the public APIs of TensorFlow are backwards compatible across minor and patch versions. The public APIs consist of

- All the documented [Python \(./api\\_docs/python\)](#) functions and classes in the `tensorflow` module and its submodules, except for
  - Private symbols: any function, class, etc., whose name start with `_`
  - Experimental and `tf.contrib` symbols, see [below](#) for details.

Note that the code in the `examples/` and `tools/` directories is not reachable through the `tensorflow` Python module and is thus not covered by the compatibility guarantee.

If a symbol is available through the `tensorflow` Python module or its submodules, but is not documented, then it is **not** considered part of the public API.

- The compatibility API (in Python, the `tf.compat` module). At major versions, we may release utilities and additional endpoints to help users with the transition to a new major version. These API symbols are deprecated and not supported (i.e., we will not add any features, and we will not fix bugs other than to fix vulnerabilities), but they do fall under our compatibility guarantees.
- The C API ([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/c/c\\_api.h](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/c/c_api.h)).
- The following protocol buffer files:
  - `attr_value` ([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/attr\\_value.proto](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/attr_value.proto))
  - `config` (<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/config.proto>)
  - `event` (<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/util/event.proto>)
  - `graph` (<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/graph.proto>)
  - `op_def` ([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/op\\_def.proto](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/op_def.proto))
  - `reader_base` ([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/reader\\_base.proto](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/reader_base.proto))
  - `summary` (<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/summary.proto>)
  - `tensor` (<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/tensor.proto>)
  - `tensor_shape` ([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/tensor\\_shape.proto](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/tensor_shape.proto))
  - `types` (<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/types.proto>)

### What is *not* covered

Some parts of TensorFlow can change in backward incompatible ways at any point. These include:

- **Experimental APIs**: To facilitate development, we exempt some API symbols clearly marked as experimental from the compatibility guarantees. In particular, the following are not covered by any compatibility guarantees:
  - any symbol in the `tf.contrib` module or its submodules;
  - any symbol (module, function, argument, property, class, or constant) whose name contains `experimental` or `Experimental` ; or

- any symbol whose fully qualified name includes a module or class which is itself experimental. This includes fields and submessages of any protocol buffer called experimental .
- **Other languages:** TensorFlow APIs in languages other than Python and C, such as:
  - C++ (`./install/lang_c.md`) (exposed through header files in `tensorflow/cc` (<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/cc>)).
  - Java (`./install/lang_java.md`),
  - Go (`./install/lang_go.md`)
  - JavaScript (<https://www.tensorflow.org/js>)
- **Details of composite ops:** Many public functions in Python expand to several primitive ops in the graph, and these details will be part of any graphs saved to disk as GraphDef s. These details may change for minor releases. In particular, regression tests that check for exact matching between graphs are likely to break across minor releases, even though the behavior of the graph should be unchanged and existing checkpoints will still work.
- **Floating point numerical details:** The specific floating point values computed by ops may change at any time. Users should rely only on approximate accuracy and numerical stability, not on the specific bits computed. Changes to numerical formulas in minor and patch releases should result in comparable or improved accuracy, with the caveat that in machine learning improved accuracy of specific formulas may result in decreased accuracy for the overall system.
- **Random numbers:** The specific random numbers computed may change at any time. Users should rely only on approximately correct distributions and statistical strength, not the specific bits computed. See the [random number generation \(random\\_numbers.ipynb\)](#) guide for details.
- **Version skew in distributed Tensorflow:** Running two different versions of TensorFlow in a single cluster is unsupported. There are no guarantees about backwards compatibility of the wire protocol.
- **Bugs:** We reserve the right to make backwards incompatible behavior (though not API) changes if the current implementation is clearly broken, that is, if it contradicts the documentation or if a well-known and well-defined intended behavior is not properly implemented due to a bug. For example, if an optimizer claims to implement a well-known optimization algorithm but does not match that algorithm due to a bug, then we will fix the optimizer. Our fix may break code relying on the wrong behavior for convergence. We will note such changes in the release notes.
- **Unused API:** We reserve the right to make backwards incompatible changes to APIs for which we find no documented uses (by performing audit of TensorFlow usage through GitHub search). Before making any such changes, we will announce our intention to make the change on the [announce@ mailing list](mailto:announce@ mailing list) (<https://groups.google.com/a/tensorflow.org/forum/#tfforum/announce>), providing instructions for how to address any breakages (if applicable), and wait for two weeks to give our community a chance to share their feedback.
- **Error behavior:** We may replace errors with non-error behavior. For instance, we may change a function to compute a result instead of raising an error, even if that error is documented. We also reserve the right to change the text of error messages. In addition, the type of an error may change unless the exception type for a specific error condition is specified in the documentation.

## Compatibility of SavedModels, graphs and checkpoints

SavedModel is the preferred serialization format to use in TensorFlow programs. SavedModels contain two parts: One or more graphs encoded as `GraphDefs` and a `Checkpoint`. The graphs describe the data flow of ops to be run, and checkpoints contain the saved tensor values of variables in a graph.

Many TensorFlow users create SavedModels, and load and execute them with a later release of TensorFlow. In compliance with [semver \(https://semver.org\)](https://semver.org), SavedModels written with one version of TensorFlow can be loaded and evaluated with a later version of TensorFlow with the same major release.

We make additional guarantees for *supported* SavedModels. We call a SavedModel which was created using **only non-deprecated, non-experimental, non-compatibility APIs** in TensorFlow major version N a *SavedModel supported in version N*. Any SavedModel supported in TensorFlow major version N can be loaded and executed with TensorFlow major version N+1. However, the functionality required to build or modify such a model may not be available any more, so this guarantee only applies to the unmodified SavedModel.

We will endeavor to preserve backwards compatibility as long as possible, so that the serialized files are usable over long periods of time.

## GraphDef compatibility

Graphs are serialized via the `GraphDef` protocol buffer. To facilitate backwards incompatible changes to graphs, each `GraphDef` has a version number separate from the TensorFlow version. For example, `GraphDef` version 17 deprecated the `inv` op in favor of `reciprocal`. The semantics are:

- Each version of TensorFlow supports an interval of `GraphDef` versions. This interval will be constant across patch releases, and will only grow across minor releases. Dropping support for a `GraphDef` version will only occur for a major release of TensorFlow (and only aligned with the version support guaranteed for SavedModels).
- Newly created graphs are assigned the latest `GraphDef` version number.
- If a given version of TensorFlow supports the `GraphDef` version of a graph, it will load and evaluate with the same behavior as the TensorFlow version used to generate it (except for floating point numerical details and random numbers as outlined above), regardless of the major version of TensorFlow. In particular, a `GraphDef` which is compatible with a checkpoint file in one version of TensorFlow (such as is the case in a SavedModel) will remain compatible with that checkpoint in subsequent versions, as long as the `GraphDef` is supported.

Note that this applies only to serialized Graphs in `GraphDefs` (and SavedModels): Code which reads a checkpoint may not be able to read checkpoints generated by the same code running a different version of TensorFlow.

- If the `GraphDef` *upper* bound is increased to X in a (minor) release, there will be at least six months before the *lower* bound is increased to X. For example (we're using hypothetical version numbers here):
  - TensorFlow 1.2 might support `GraphDef` versions 4 to 7.
  - TensorFlow 1.3 could add `GraphDef` version 8 and support versions 4 to 8.
  - At least six months later, TensorFlow 2.0.0 could drop support for versions 4 to 7, leaving version 8 only.

Note that because major versions of TensorFlow are usually published more than 6 months apart, the guarantees for supported SavedModels detailed above are much stronger than the 6 months guarantee for `GraphDefs`.

Finally, when support for a `GraphDef` version is dropped, we will attempt to provide tools for automatically converting graphs to a newer supported `GraphDef` version.

## Graph and checkpoint compatibility when extending TensorFlow

This section is relevant only when making incompatible changes to the `GraphDef` format, such as when adding ops, removing ops, or changing the functionality of existing ops. The previous section should suffice for most users.

## Backward and partial forward compatibility

Our versioning scheme has three requirements:

- **Backward compatibility** to support loading graphs and checkpoints created with older versions of TensorFlow.
- **Forward compatibility** to support scenarios where the producer of a graph or checkpoint is upgraded to a newer version of TensorFlow before the consumer.
- Enable evolving TensorFlow in incompatible ways. For example, removing ops, adding attributes, and removing attributes.

Note that while the `GraphDef` version mechanism is separate from the TensorFlow version, backwards incompatible changes to the `GraphDef` format are still restricted by Semantic Versioning. This means functionality can only be removed or changed between MAJOR versions of TensorFlow (such as 1.7 to 2.0). Additionally, forward compatibility is enforced within Patch releases (1.x.1 to 1.x.2 for example).

To achieve backward and forward compatibility and to know when to enforce changes in formats, graphs and checkpoints have metadata that describes when they were produced. The sections below detail the TensorFlow implementation and guidelines for evolving `GraphDef` versions.

## Independent data version schemes

There are different data versions for graphs and checkpoints. The two data formats evolve at different rates from each other and also at different rates from TensorFlow. Both versioning systems are defined in `core/public/version.h` (<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/public/version.h>). Whenever a new version is added, a note is added to the header detailing what changed and the date.

## Data, producers, and consumers

We distinguish between the following kinds of data version information: **producers**: binaries that produce data. Producers have a version (producer) and a minimum consumer version that they are compatible with (min\_consumer). **consumers**: binaries that consume data. Consumers have a version (consumer) and a minimum producer version that they are compatible with (min\_producer).

Each piece of versioned data has a `VersionDef` versions (<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/versions.proto>) field which records the producer that made the data, the min\_consumer that it is compatible with, and a list of bad\_consumers versions that are disallowed.

By default, when a producer makes some data, the data inherits the producer's producer and min\_consumer versions. bad\_consumers can be set if specific consumer versions are known to contain bugs and must be avoided. A consumer can accept a piece of data if the following are all true:

- consumer >= data's min\_consumer
- data's producer >= consumer's min\_producer
- consumer not in data's bad\_consumers

Since both producers and consumers come from the same TensorFlow code base, `core/public/version.h`

(<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/public/version.h>) contains a main data version which is treated as either producer or consumer depending on context and both min\_consumer and min\_producer (needed by producers and consumers, respectively). Specifically,

- For GraphDef versions, we have `TF_GRAPH_DEF_VERSION`, `TF_GRAPH_DEF_VERSION_MIN_CONSUMER`, and `TF_GRAPH_DEF_VERSION_MIN_PRODUCER`.
- For checkpoint versions, we have `TF_CHECKPOINT_VERSION`, `TF_CHECKPOINT_VERSION_MIN_CONSUMER`, and `TF_CHECKPOINT_VERSION_MIN_PRODUCER`.

## Add a new attribute with default to an existing op

Following the guidance below gives you forward compatibility only if the set of ops has not changed:

1. If forward compatibility is desired, set `strip_defaultAttrs` to True while exporting the model using either the `tf.saved_model.SavedModelBuilder.add_meta_graph_and_variables` and `tf.saved_model.SavedModelBuilder.add_meta_graph` methods of the `SavedModelBuilder` class, or `tf.estimator.Estimator.export_saved_model`
2. This strips off the default valued attributes at the time of producing/exporting the models. This makes sure that the exported `tf.MetaGraphDef` does not contain the new op-attribute when the default value is used.
3. Having this control could allow out-of-date consumers (for example, serving binaries that lag behind training binaries) to continue loading the models and prevent interruptions in model serving.

## Evolving GraphDef versions

This section explains how to use this versioning mechanism to make different types of changes to the `GraphDef` format.

### Add an op

Add the new op to both consumers and producers at the same time, and do not change any `GraphDef` versions. This type of change is automatically backward compatible, and does not impact forward compatibility plan since existing producer scripts will not suddenly use the new functionality.

### Add an op and switch existing Python wrappers to use it

1. Implement new consumer functionality and increment the `GraphDef` version.
2. If it is possible to make the wrappers use the new functionality only in cases that did not work before, the wrappers can be updated now.
3. Change Python wrappers to use the new functionality. Do not increment `min_consumer`, since models that do not use this op should not break.

### Remove or restrict an op's functionality

1. Fix all producer scripts (not TensorFlow itself) to not use the banned op or functionality.
2. Increment the `GraphDef` version and implement new consumer functionality that bans the removed op or functionality for GraphDefs at the new version and above. If possible, make TensorFlow stop producing GraphDefs with the banned functionality. To do so, add the `REGISTER_OP(...).Deprecated(deprecated_at_version, message)` ([https://github.com/tensorflow/tensorflow/blob/b289bc7a50fc0254970c60aaeba01c33de61a728/tensorflow/core/ops/array\\_ops.cc#L1009](https://github.com/tensorflow/tensorflow/blob/b289bc7a50fc0254970c60aaeba01c33de61a728/tensorflow/core/ops/array_ops.cc#L1009)).
3. Wait for a major release for backward compatibility purposes.
4. Increase `min_producer` to the `GraphDef` version from (2) and remove the functionality entirely.

### Change an op's functionality

1. Add a new similar op named `SomethingV2` or similar and go through the process of adding it and switching existing Python wrappers to use it. To ensure forward compatibility use the checks suggested in `compat.py` (<https://www.tensorflow.org/code/tensorflow/python/compat/compat.py>) when changing the Python wrappers.
2. Remove the old op (Can only take place with a major version change due to backward compatibility).
3. Increase `min_consumer` to rule out consumers with the old op, add back the old op as an alias for `SomethingV2`, and go through the process to switch existing Python wrappers to use it.
4. Go through the process to remove `SomethingV2`.

### Ban a single unsafe consumer version

1. Bump the `GraphDef` version and add the bad version to `bad_consumers` for all new GraphDefs. If possible, add to `bad_consumers` only for GraphDefs which contain a certain op or similar.
2. If existing consumers have the bad version, push them out as soon as possible.

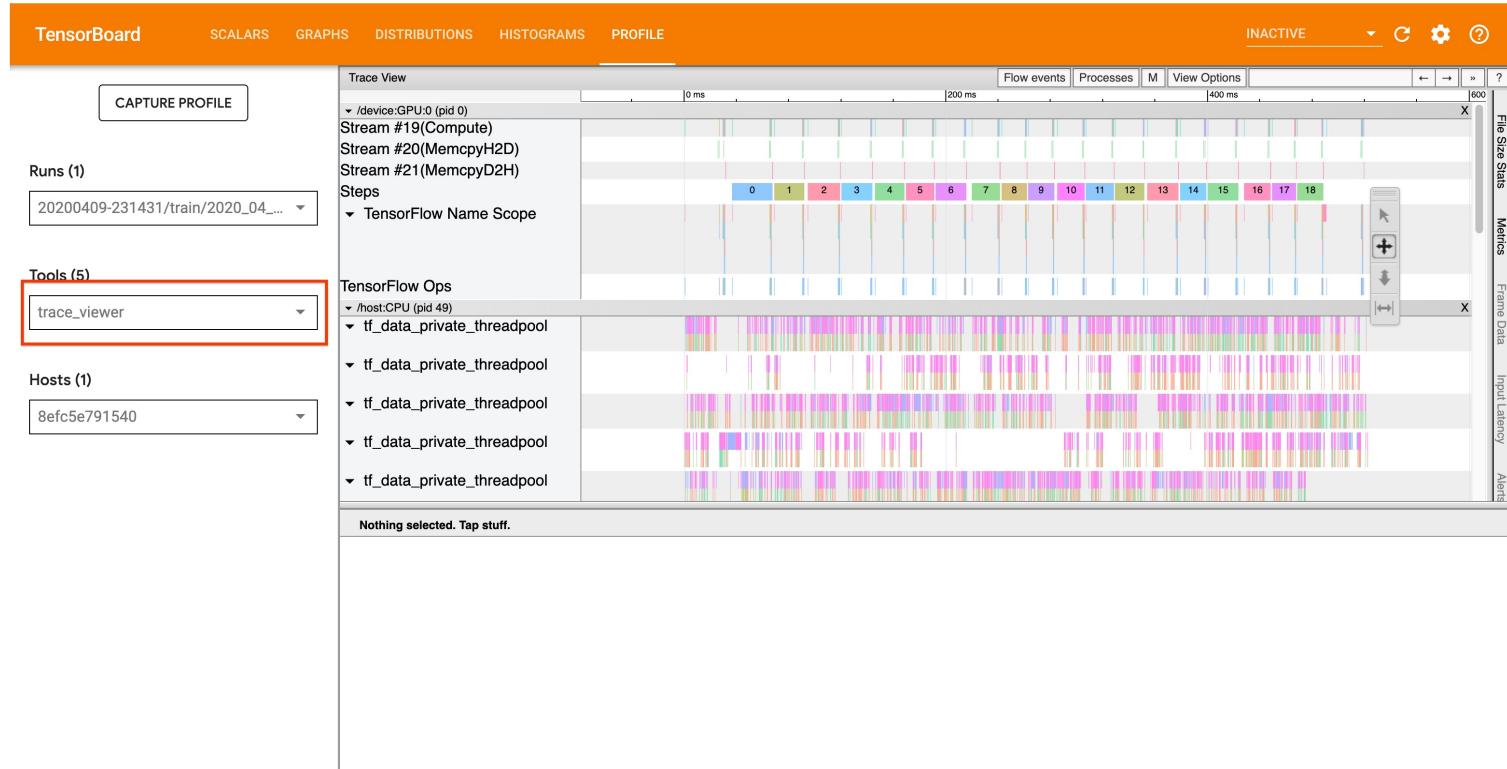
# Analyze tf.data performance with the TF Profiler

## Overview

This guide assumes familiarity with the TensorFlow Profiler (<https://www.tensorflow.org/guide/profiler>) and `tf.data` (<https://www.tensorflow.org/guide/data>). It aims to provide step by step instructions with examples to help users diagnose and fix input pipeline performance issues.

To begin, collect a profile of your TensorFlow job. Instructions on how to do so are available for CPUs/GPUs

([https://www.tensorflow.org/guide/profiler#collect\\_performance\\_data](https://www.tensorflow.org/guide/profiler#collect_performance_data)) and Cloud TPUs ([https://cloud.google.com/tpu/docs/cloud-tpu-tools#capture\\_profile](https://cloud.google.com/tpu/docs/cloud-tpu-tools#capture_profile)).



The analysis workflow detailed below focuses on the trace viewer tool in the Profiler. This tool displays a timeline that shows the duration of ops executed by your TensorFlow program and allows you to identify which ops take the longest to execute. For more information on the trace viewer, check out [this section](https://www.tensorflow.org/guide/profiler#trace_viewer) ([https://www.tensorflow.org/guide/profiler#trace\\_viewer](https://www.tensorflow.org/guide/profiler#trace_viewer)) of the TF Profiler guide. In general, `tf.data` events will appear on the host CPU timeline.

## Analysis Workflow

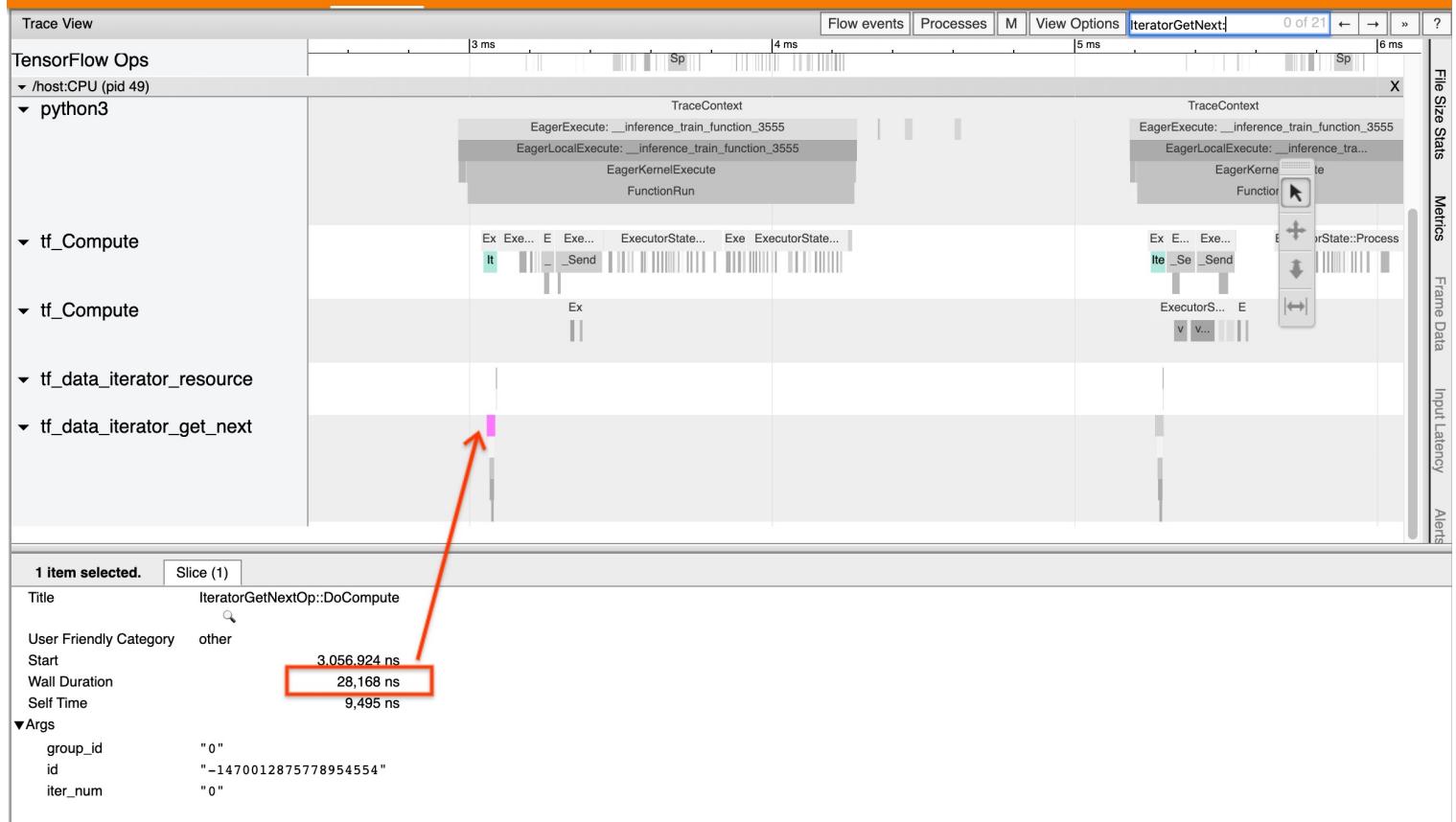
Please follow the workflow below. If you have feedback to help us improve it, please [create a github issue](https://github.com/tensorflow/tensorflow/issues/new/choose) (<https://github.com/tensorflow/tensorflow/issues/new/choose>) with the label "comp:data".

### 1. Is your `tf.data` pipeline producing data fast enough?

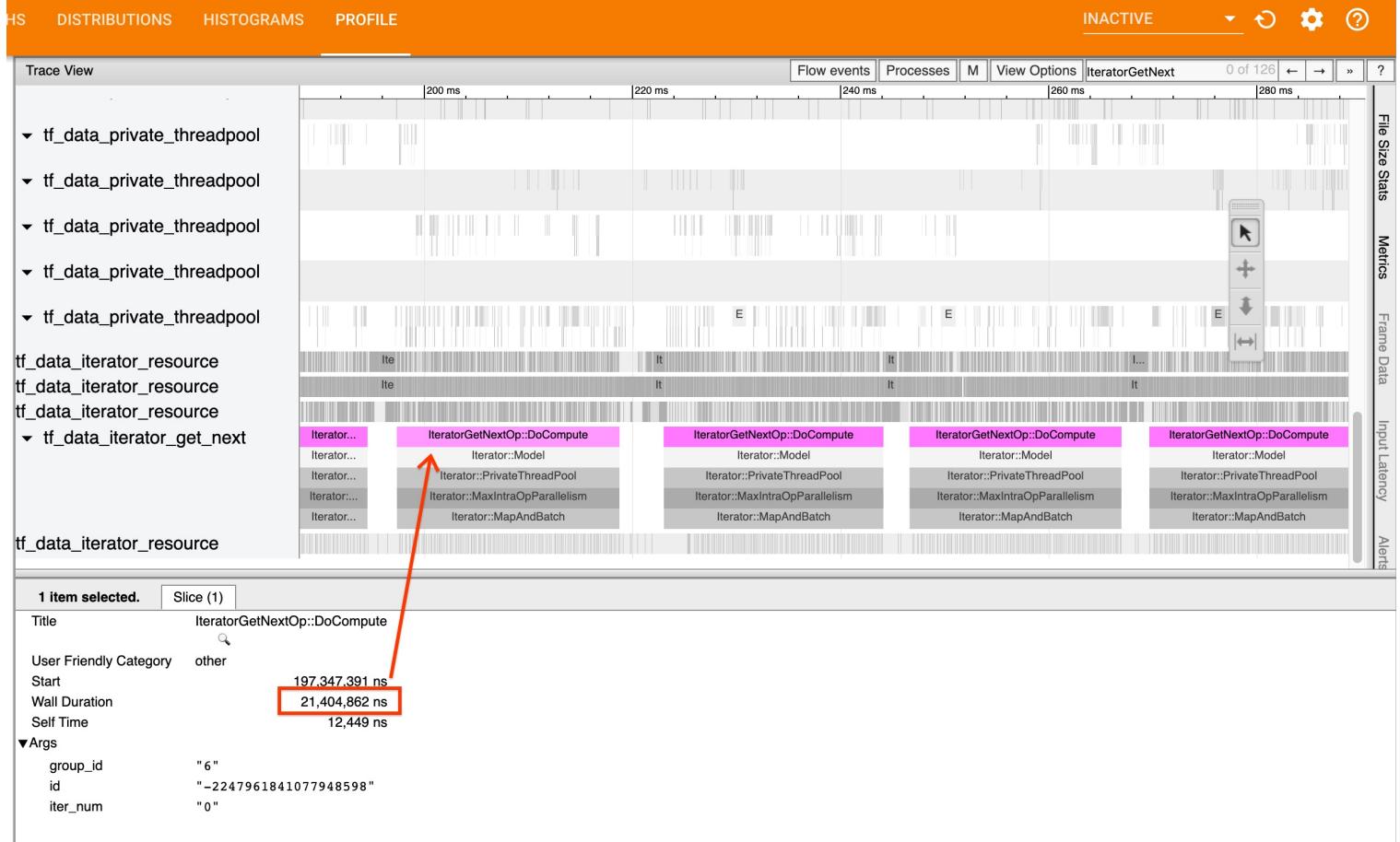
Begin by ascertaining whether the input pipeline is the bottleneck for your TensorFlow program.

To do so, look for `IteratorGetNext::DoCompute` ops in the trace viewer. In general, you expect to see these at the start of a step. These slices represent the time it takes for your input pipeline to yield a batch of elements when it is requested. If you're using keras or iterating over your dataset in a `tf.function`, these should be found in `tf_data_iterator_get_next` threads.

Note that if you're using a [distribution strategy](https://www.tensorflow.org/guide/distributed_training) ([https://www.tensorflow.org/guide/distributed\\_training](https://www.tensorflow.org/guide/distributed_training)), you may see `IteratorGetNextAsOptional::DoCompute` events instead of `IteratorGetNext::DoCompute` (as of TF 2.3).



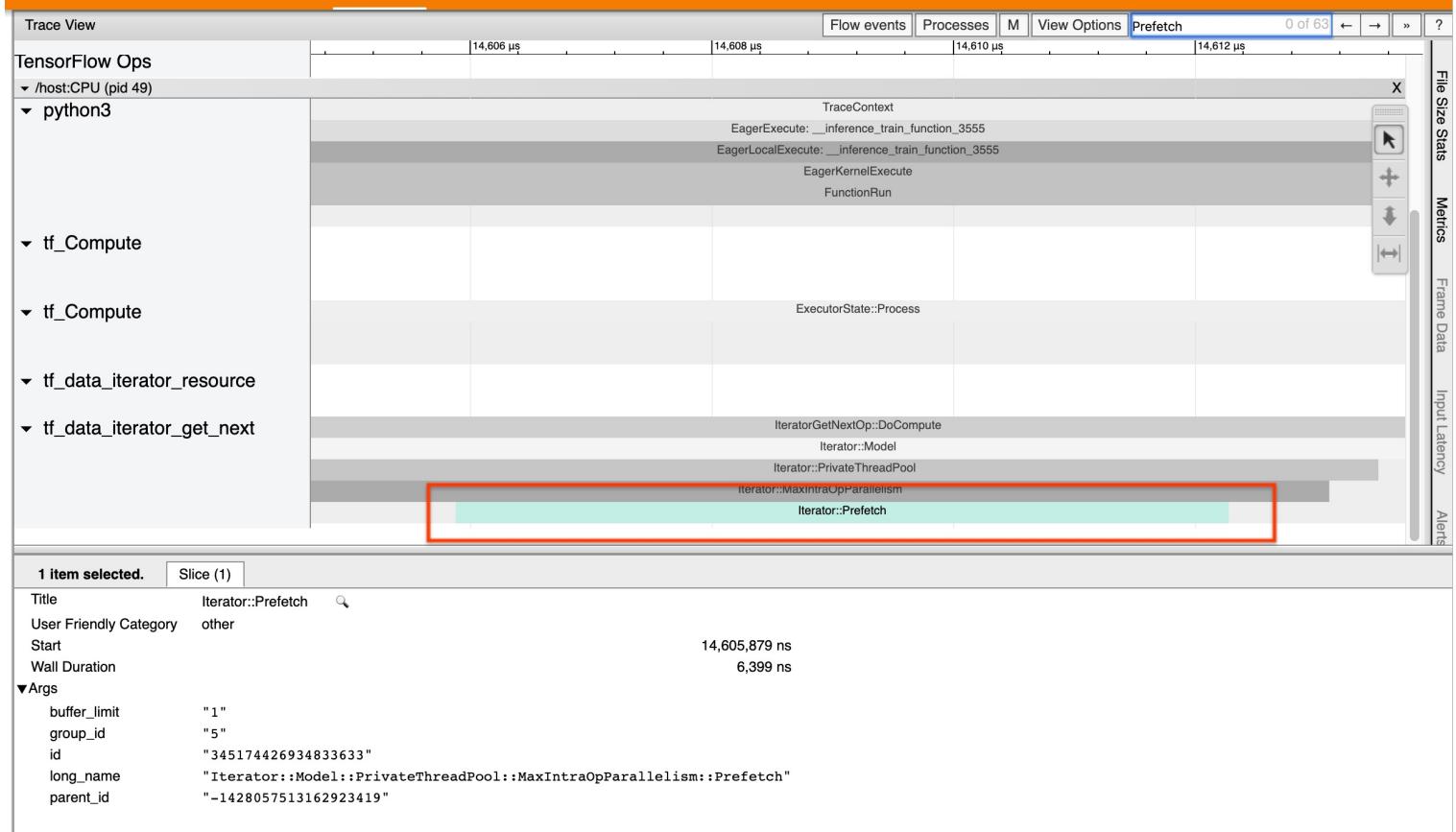
If the calls return quickly (<= 50 us), this means that your data is available when it is requested. The input pipeline is not your bottleneck; see the Profiler guide (<https://www.tensorflow.org/guide/profiler>) for more generic performance analysis tips.



If the calls return slowly, tf.data is unable to keep up with the consumer's requests. Continue to the next section.

## 2. Are you prefetching data?

The best practice for input pipeline performance is to insert a `tf.data.Dataset.prefetch` transformation at the end of your `tf.data` pipeline. This transformation overlaps the input pipeline's preprocessing computation with the next step of model computation and is required for optimal input pipeline performance when training your model. If you're prefetching data, you should see a `Iterator::Prefetch` slice on the same thread as the `IteratorGetNext::DoCompute` op.



If you don't have a `prefetch` at the end of your pipeline, you should add one. For more information about `tf.data` performance recommendations, see the [tf.data performance guide](#) ([https://www.tensorflow.org/guide/data\\_performance#prefetching](https://www.tensorflow.org/guide/data_performance#prefetching)).

If you're already prefetching data, and the input pipeline is still your bottleneck, continue to the next section to further analyze performance.

### 3. Are you reaching high CPU utilization?

`tf.data` achieves high throughput by trying to make the best possible use of available resources. In general, even when running your model on an accelerator like a GPU or TPU, the `tf.data` pipelines are run on the CPU. You can check your utilization with tools like `sar` (<https://linux.die.net/man/1/sar>) and `htop` (<https://en.wikipedia.org/wiki/Htop>), or in the `cloud monitoring console` ([https://cloud.google.com/monitoring/docs/monitoring\\_in\\_console](https://cloud.google.com/monitoring/docs/monitoring_in_console)) if you're running on GCP.

If your utilization is low, this suggests that your input pipeline may not be taking full advantage of the host CPU. You should consult the [tf.data performance guide](#) ([https://www.tensorflow.org/guide/data\\_performance](https://www.tensorflow.org/guide/data_performance)) for best practices. If you have applied the best practices and utilization and throughput remain low, continue to [Bottleneck analysis](#) below.

If your utilization is approaching the resource limit, in order to improve performance further, you need to either improve the efficiency of your input pipeline (for example, avoiding unnecessary computation) or offload computation.

You can improve the efficiency of your input pipeline by avoiding unnecessary computation in `tf.data`. One way of doing this is inserting a `tf.data.Dataset.cache` ([https://www.tensorflow.org/guide/data\\_performance#caching](https://www.tensorflow.org/guide/data_performance#caching)) transformation after computation-intensive work if your data fits into memory; this reduces computation at the cost of increased memory usage. Additionally, disabling intra-op parallelism in `tf.data` has the potential to increase efficiency by > 10%, and can be done by setting the following option on your input pipeline:

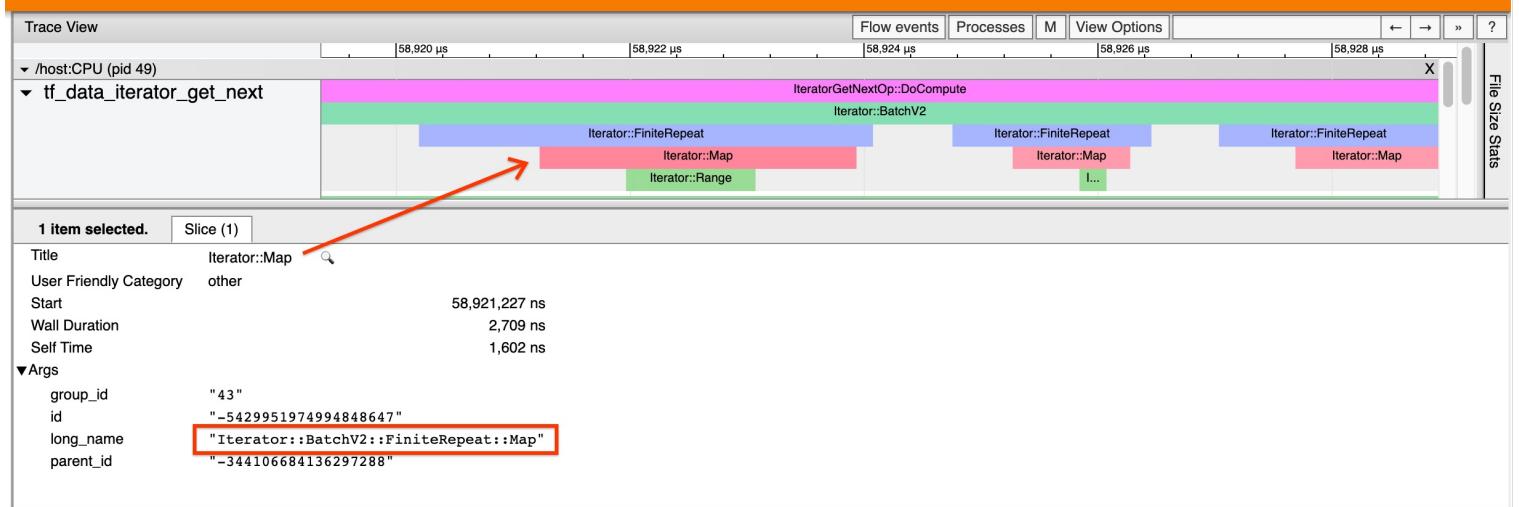
```
python
dataset = ...
options = tf.data.Options()
options.experimental_threading.max_intra_op_parallelism = 1
dataset = dataset.with_options(options)
```

### 4. Bottleneck Analysis

The following section walks through how to read `tf.data` events in the trace viewer to understand where the bottleneck is and possible mitigation strategies.

#### Understanding `tf.data` events in the Profiler

Each `tf.data` event in the Profiler has the name `Iterator::<Dataset>`, where `<Dataset>` is the name of the dataset source or transformation. Each event also has the long name `Iterator::<Dataset_1>::...::<Dataset_n>`, which you can see by clicking on the `tf.data` event. In the long name, `<Dataset_n>` matches `<Dataset>` from the (short) name, and the other datasets in the long name represent downstream transformations.



For example, the above screenshot was generated from the following code:

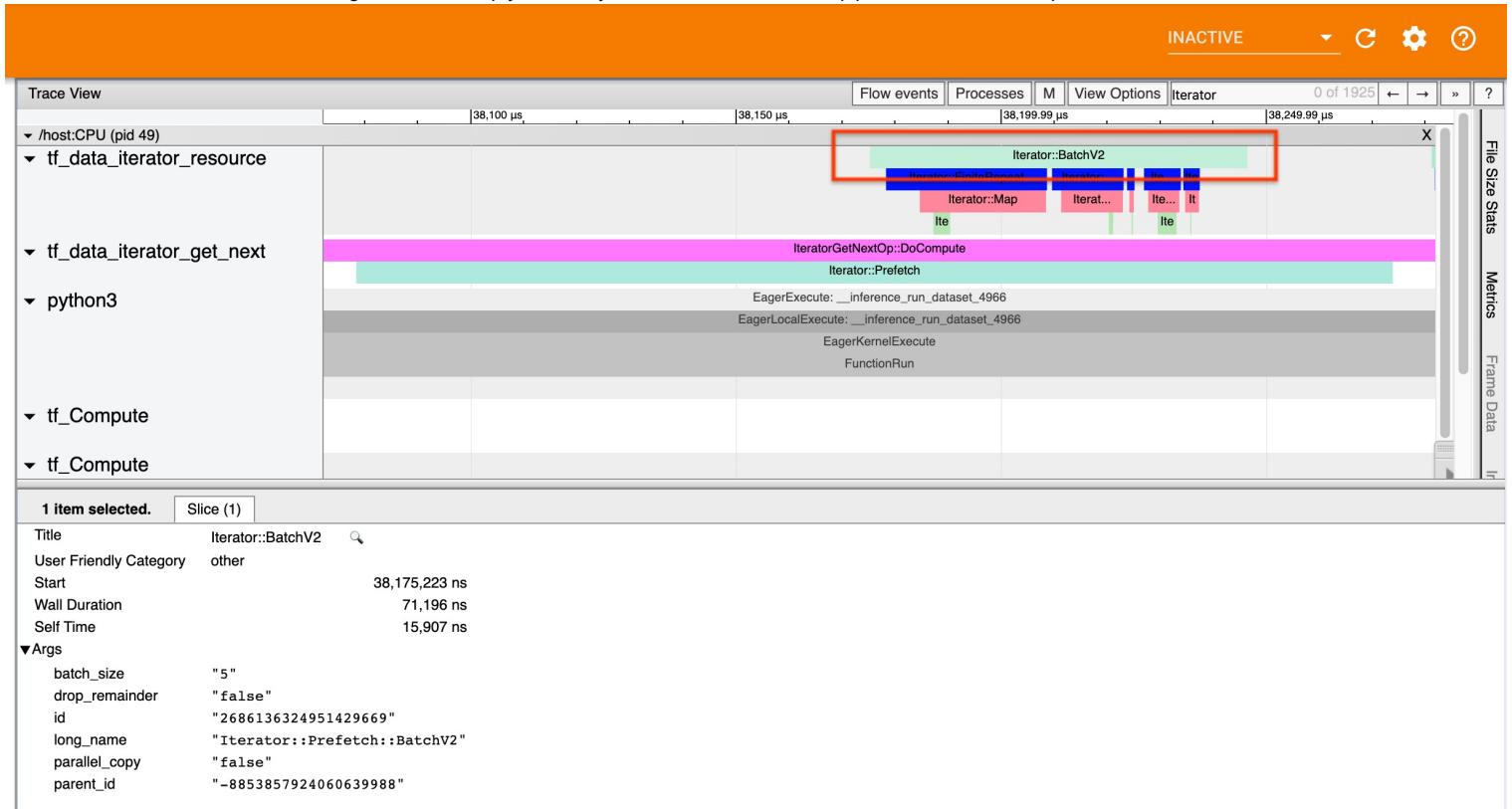
```
python
dataset = tf.data.Dataset.range(10)
dataset = dataset.map(lambda x: x)
dataset = dataset.repeat(2)
dataset = dataset.batch(5)
```

Here, the `Iterator::Map` event has the long name `Iterator::BatchV2::FiniteRepeat::Map`. Note that the datasets name may differ slightly from the python API (for example, `FiniteRepeat` instead of `Repeat`), but should be intuitive enough to parse.

#### Synchronous and asynchronous transformations

For synchronous `tf.data` transformations (such as `Batch` and `Map`), you will see events from upstream transformations on the same thread. In the above example, since all the transformations used are synchronous, all the events appear on the same thread.

For asynchronous transformations (such as `Prefetch`, `ParallelMap`, `ParallelInterleave` and `MapAndBatch`) events from upstream transformations will be on a different thread. In such cases, the “long name” can help you identify which transformation in a pipeline an event corresponds to.



For example, the above screenshot was generated from the following code:

```
python
dataset = tf.data.Dataset.range(10)
dataset = dataset.map(lambda x: x)
dataset = dataset.repeat(2)
dataset = dataset.batch(5)
dataset = dataset.prefetch(1)
```

Here, the `Iterator::Prefetch` events are on the `tf_data_iterator_get_next` threads. Since `Prefetch` is asynchronous, its input events (`BatchV2`) will be on a different thread, and can be located by searching for the long name `Iterator::Prefetch::BatchV2`. In this case, they are on the `tf_data_iterator_resource` thread. From its long name, you can deduce that `BatchV2` is upstream of `Prefetch`. Furthermore, the `parent_id` of the `BatchV2` event will match the ID of the `Prefetch` event.

#### Identifying the bottleneck

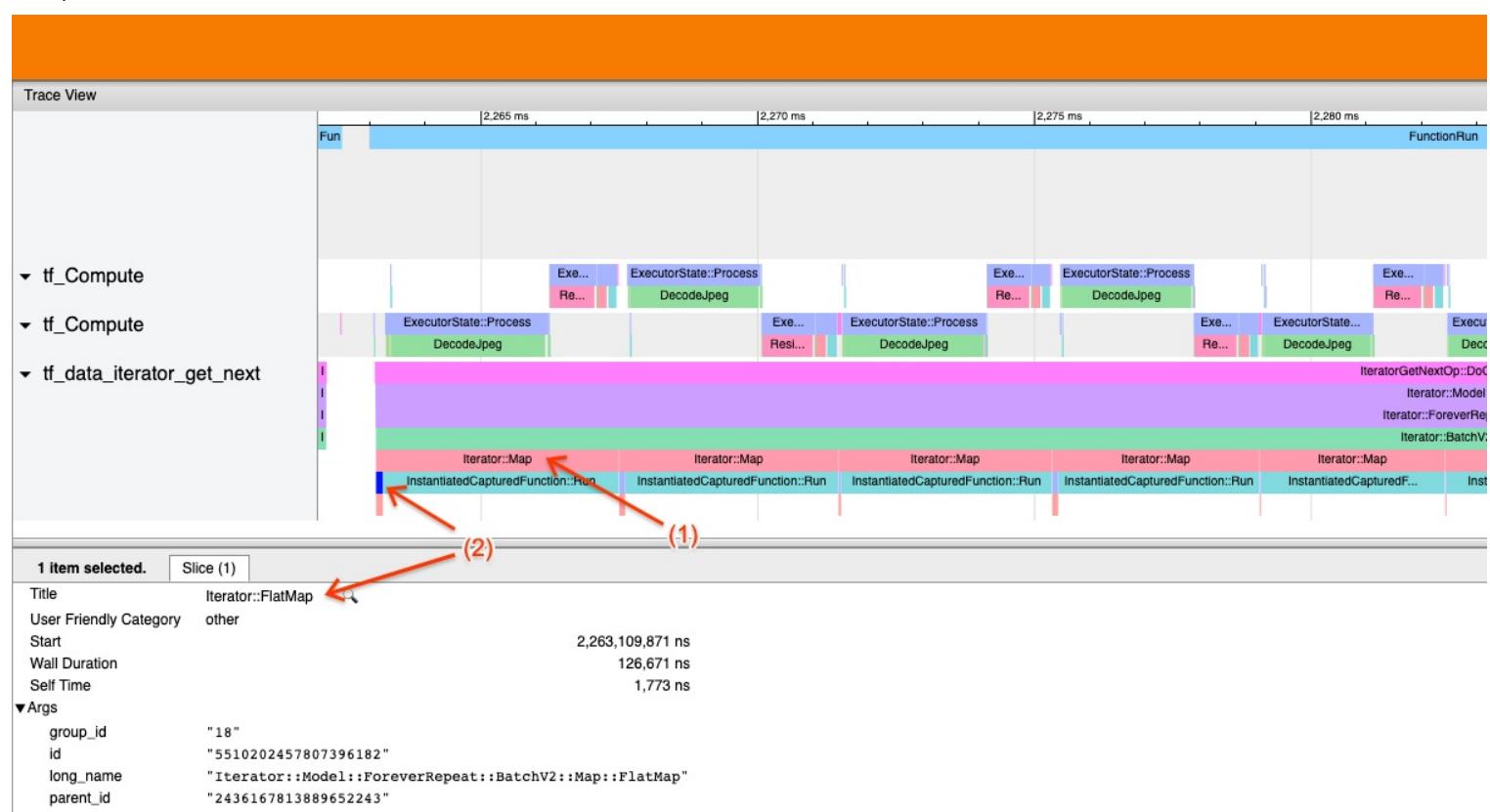
In general, to identify the bottleneck in your input pipeline, walk the input pipeline from the outermost transformation all the way to the source. Starting from the final transformation in your pipeline, recurse into upstream transformations until you find a slow transformation or reach a source dataset, such as `TFRecord`. In the example above, you would start from `Prefetch`, then walk upstream to `BatchV2`, `FiniteRepeat`, `Map`, and finally `Range`.

In general, a slow transformation corresponds to one whose events are long, but whose input events are short. Some examples follow below.

Note that the final (outermost) transformation in most host input pipelines is the `Iterator::Model` event. The Model transformation is introduced automatically by the `tf.data` runtime and is used for instrumenting and autotuning the input pipeline performance.

If your job is using a [distribution strategy](https://www.tensorflow.org/guide/distributed_training) ([https://www.tensorflow.org/guide/distributed\\_training](https://www.tensorflow.org/guide/distributed_training)), the trace viewer will contain additional events that correspond to the device input pipeline. The outermost transformation of the device pipeline (nested under `IteratorGetNextOp::DoCompute` or `IteratorGetNextAsOptionalOp::DoCompute`) will be an `Iterator::Prefetch` event with an upstream `Iterator::Generator` event. You can find the corresponding host pipeline by searching for `Iterator::Model` events.

#### Example 1



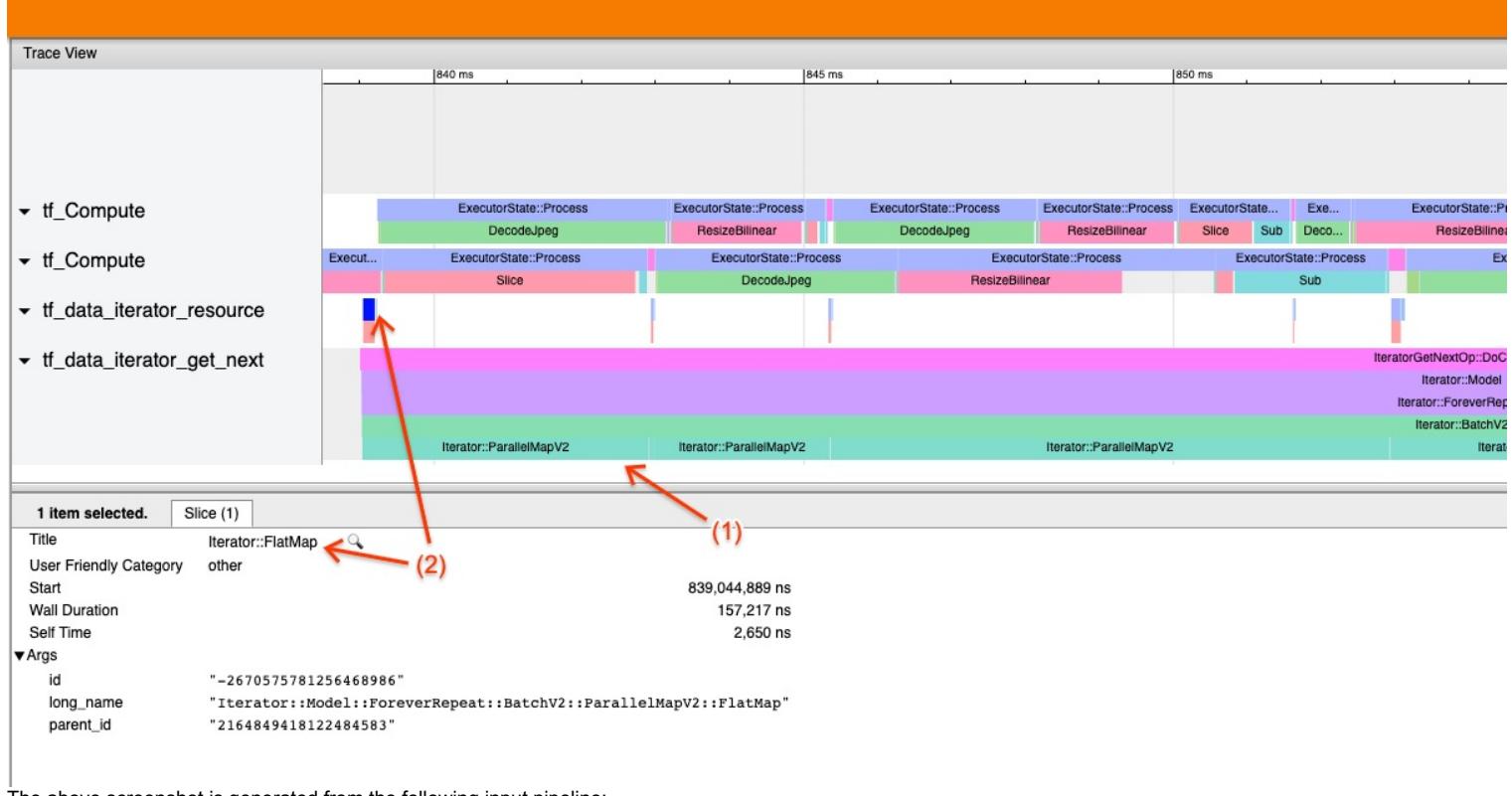
The above screenshot is generated from the following input pipeline:

```
python
dataset = tf.data.TFRecordDataset(filename)
dataset = dataset.map(parse_record)
dataset = dataset.batch(32)
dataset = dataset.repeat()
```

In the screenshot, observe that (1) `Iterator::Map` events are long, but (2) its input events ( `Iterator::FlatMap` ) return quickly. This suggests that the sequential Map transformation is the bottleneck.

Note that in the screenshot, the `InstantiatedCapturedFunction::Run` event corresponds to the time it takes to execute the map function.

#### Example 2



The above screenshot is generated from the following input pipeline:

```
python
dataset = tf.data.TFRecordDataset(filename)
dataset = dataset.map(parse_record, num_parallel_calls=2)
dataset = dataset.batch(32)
dataset = dataset.repeat()
```

This example is similar to the above, but uses `ParallelMap` instead of `Map`. We notice here that (1) `Iterator::ParallelMap` events are long, but (2) its input events `Iterator::FlatMap` (which are on a different thread, since `ParallelMap` is asynchronous) are short. This suggests that the `ParallelMap` transformation is the bottleneck.

## Addressing the bottleneck

### Source datasets

If you've identified a dataset source as the bottleneck, such as reading from TFRecord files, you can improve performance by parallelizing data extraction. To do so, ensure that your data is sharded across multiple files and use `tf.data.Dataset.interleave` with the `num_parallel_calls` parameter set to `tf.data.AUTOTUNE`. If determinism is not important to your program, you can further improve performance by setting the `deterministic=False` flag on `tf.data.Dataset.interleave` as of TF 2.2. For example, if you're reading from TFRecords, you can do the following:

```
python
dataset = tf.data.Dataset.from_tensor_slices(filenames)
dataset = dataset.interleave(tf.data.TFRecordDataset,
    num_parallel_calls=tf.data.AUTOTUNE,
    deterministic=False)
```

Note that sharded files should be reasonably large to amortize the overhead of opening a file. For more details on parallel data extraction, see [this section](https://www.tensorflow.org/guide/data_performance#parallelizing_data_extraction) ([https://www.tensorflow.org/guide/data\\_performance#parallelizing\\_data\\_extraction](https://www.tensorflow.org/guide/data_performance#parallelizing_data_extraction)) of the `tf.data` performance guide.

### Transformation datasets

If you've identified an intermediate `tf.data` transformation as the bottleneck, you can address it by parallelizing the transformation or [caching the computation](https://www.tensorflow.org/guide/data_performance#caching) ([https://www.tensorflow.org/guide/data\\_performance#caching](https://www.tensorflow.org/guide/data_performance#caching)) if your data fits into memory and it is appropriate. Some transformations such as `Map` have parallel counterparts; the `tf.data` performance guide demonstrates ([https://www.tensorflow.org/guide/data\\_performance#parallelizing\\_data\\_transformation](https://www.tensorflow.org/guide/data_performance#parallelizing_data_transformation)) how to parallelize these. Other transformations, such as `Filter`, `Unbatch`, and `Batch` are inherently sequential; you can parallelize them by introducing "outer parallelism". For example, supposing your input pipeline initially looks like the following, with `Batch` as the bottleneck:

```
python
filenames = tf.data.Dataset.list_files(file_path, shuffle=is_training)
dataset = filenames_to_dataset(filenames)
dataset = dataset.batch(batch_size)
```

You can introduce "outer parallelism" by running multiple copies of the input pipeline over sharded inputs and combining the results:

```
```python
filenames = tf.data.Dataset.list_files(file_path, shuffle=is_training)
def make_dataset(shard_index): filenames = filenames.shard(NUM_SHARDS, shard_index) dataset = filenames_to_dataset(filenames) Return dataset.batch(batch_size)
indices = tf.data.Dataset.range(NUM_SHARDS) dataset = indices.interleave(make_dataset, num_parallel_calls=tf.data.AUTOTUNE) dataset =
dataset.prefetch(tf.data.AUTOTUNE)
```
```

## Additional resources

- `tf.data` performance guide ([https://www.tensorflow.org/guide/data\\_performance](https://www.tensorflow.org/guide/data_performance)) on how to write performance `tf.data` input pipelines
- Inside TensorFlow video: `tf.data` best practices (<https://www.youtube.com/watch?v=ZnukSLKEw34>)
- Profiler guide (<https://www.tensorflow.org/guide/profiler>)
- Profiler tutorial with colab ([https://www.tensorflow.org/tensorboard/tensorboard\\_profiling\\_keras](https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras))

## Optimize TensorFlow performance using the Profiler

This guide demonstrates how to use the tools available with the TensorFlow Profiler to track the performance of your TensorFlow models. You will learn how to understand how your model performs on the host (CPU), the device (GPU), or on a combination of both the host and device(s).

Profiling helps understand the hardware resource consumption (time and memory) of the various TensorFlow operations (ops) in your model and resolve performance bottlenecks and, ultimately, make the model execute faster.

This guide will walk you through how to install the Profiler, the various tools available, the different modes of how the Profiler collects performance data, and some recommended best practices to optimize model performance.

If you want to profile your model performance on Cloud TPUs, refer to the [Cloud TPU guide](https://cloud.google.com/tpu/docs/cloud-tpu-tools#capture_profile) ([https://cloud.google.com/tpu/docs/cloud-tpu-tools#capture\\_profile](https://cloud.google.com/tpu/docs/cloud-tpu-tools#capture_profile)).

## Install the Profiler and GPU prerequisites

Install the Profiler plugin for TensorBoard with pip. Note that the Profiler requires the latest versions of TensorFlow and TensorBoard (>=2.2).

```
shell  
pip install -U tensorflow_plugin_profile
```

To profile on the GPU, you must:

1. Meet the NVIDIA® GPU drivers and CUDA® Toolkit requirements listed on [TensorFlow GPU support software requirements](https://www.tensorflow.org/install/gpu#linux_setup) ([https://www.tensorflow.org/install/gpu#linux\\_setup](https://www.tensorflow.org/install/gpu#linux_setup)).
2. Make sure the NVIDIA® CUDA® Profiling Tools Interface (<https://developer.nvidia.com/cupti>) (CUPTI) exists on the path:  

```
shell  
/sbin/ldconfig -N -v $(sed 's/:/ /g' <<< $LD_LIBRARY_PATH) | \  
grep libcupti
```

If you don't have CUPTI on the path, prepend its installation directory to the `$LD_LIBRARY_PATH` environment variable by running:

```
shell  
export LD_LIBRARY_PATH=/usr/local/cuda/extras/CUPTI/lib64:$LD_LIBRARY_PATH
```

Then, run the `ldconfig` command above again to verify that the CUPTI library is found.

## Resolve privilege issues

When you run profiling with CUDA® Toolkit in a Docker environment or on Linux, you may encounter issues related to insufficient CUPTI privileges (`CUPTI_ERROR_INSUFFICIENT_PRIVILEGES`). Go to the [NVIDIA Developer Docs](https://developer.nvidia.com/nvidia-development-tools-solutions-ERR_NVGPUCTRPERM-permission-issue-performance-counters) ([https://developer.nvidia.com/nvidia-development-tools-solutions-ERR\\_NVGPUCTRPERM-permission-issue-performance-counters](https://developer.nvidia.com/nvidia-development-tools-solutions-ERR_NVGPUCTRPERM-permission-issue-performance-counters)){:external} to learn more about how you can resolve these issues on Linux.

To resolve CUPTI privilege issues in a Docker environment, run

```
shell  
docker run option '--privileged=true'
```

## Profiler tools

Access the Profiler from the **Profile** tab in TensorBoard, which appears only after you have captured some model data.

Note: The Profiler requires internet access to load the [Google Chart libraries](https://developers.google.com/chart/interactive/docs/basic_load_libs#basic-library-loading) ([https://developers.google.com/chart/interactive/docs/basic\\_load\\_libs#basic-library-loading](https://developers.google.com/chart/interactive/docs/basic_load_libs#basic-library-loading)).

Some charts and tables may be missing if you run TensorBoard entirely offline on your local machine, behind a corporate firewall, or in a data center.

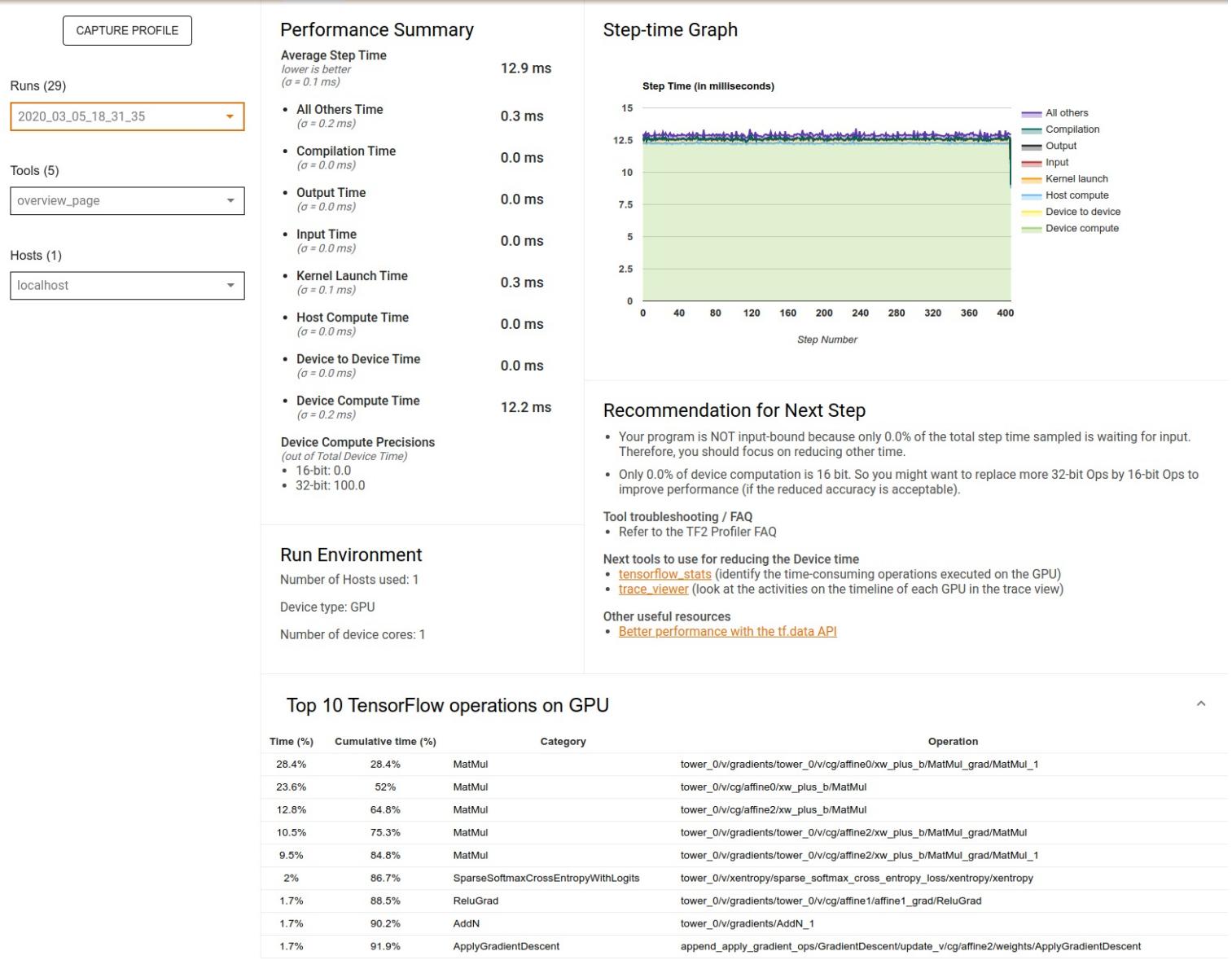
The Profiler has a selection of tools to help with performance analysis:

- Overview Page
- Input Pipeline Analyzer
- TensorFlow Stats
- Trace Viewer
- GPU Kernel Stats
- Memory Profile Tool
- Pod Viewer

## Overview page

The overview page provides a top level view of how your model performed during a profile run. The page shows you an aggregated overview page for your host and all devices, and some recommendations to improve your model training performance. You can also select individual hosts in the Host dropdown.

The overview page displays data as follows:



### Top 10 TensorFlow operations on GPU

| Time (%) | Cumulative time (%) | Category                            | Operation  |
|----------|---------------------|-------------------------------------|--|
| 28.4%    | 28.4%               | MatMul                              | tower_0/v/gradients/tower_0/v/cg/affine0/xw_plus_b/MatMul_grad/MatMul_1                    |
| 23.6%    | 52%                 | MatMul                              | tower_0/v/cg/affine0/xw_plus_b/MatMul  |
| 12.8%    | 64.8%               | MatMul                              | tower_0/v/cg/affine2/xw_plus_b/MatMul  |
| 10.5%    | 75.3%               | MatMul                              | tower_0/v/gradients/tower_0/v/cg/affine2/xw_plus_b/MatMul_grad/MatMul                      |
| 9.5%     | 84.8%               | MatMul                              | tower_0/v/gradients/tower_0/v/cg/affine2/xw_plus_b/MatMul_grad/MatMul_1                    |
| 2%       | 86.7%               | SparseSoftmaxCrossEntropyWithLogits | tower_0/v/xentropy/sparse_softmax_cross_entropy_loss/xentropy/xentropy                     |
| 1.7%     | 88.5%               | ReluGrad                            | tower_0/v/gradients/tower_0/v/cg/affine1/affine1_grad/ReluGrad                             |
| 1.7%     | 90.2%               | AddN                                | tower_0/v/gradients/AddN_1   |
| 1.7%     | 91.9%               | ApplyGradientDescent                | append_apply_gradient_ops/GradientDescent/update_v/cg/affine2/weights/ApplyGradientDescent |

- Performance Summary:** Displays a high-level summary of your model performance. The performance summary has two parts:

- Step-time breakdown: Breaks down the average step time into multiple categories of where time is spent:

- Compilation: Time spent compiling kernels.
- Input: Time spent reading input data.
- Output: Time spent reading output data.
- Kernel launch: Time spent by the host to launch kernels
- Host compute time..
- Device-to-device communication time.
- On-device compute time.
- All others, including Python overhead.

- Device compute precisions - Reports the percentage of device compute time that uses 16 and 32-bit computations.

- Step-time Graph:** Displays a graph of device step time (in milliseconds) over all the steps sampled. Each step is broken into the multiple categories (with different colors) of where time is spent. The red area corresponds to the portion of the step time the devices were sitting idle waiting for input data from the host. The green area shows how much of time the device was actually working.

- Top 10 TensorFlow operations on device (e.g. GPU):** Displays the on-device ops that ran the longest.

Each row displays an op's self time (as the percentage of time taken by all ops), cumulative time, category, and name.

- Run Environment:** Displays a high-level summary of the model run environment including:

- Number of hosts used.
- Device type (GPU/TPU).
- Number of device cores.

- Recommendation for Next Step:** Reports when a model is input bound and recommends tools you can use to locate and resolve model performance bottlenecks.

## Input pipeline analyzer

When a TensorFlow program reads data from a file it begins at the top of the TensorFlow graph in a pipelined manner. The read process is divided into multiple data processing stages connected in series, where the output of one stage is the input to the next one. This system of reading data is called the *input pipeline*.

A typical pipeline for reading records from files has the following stages:

- File reading.
- File preprocessing (optional).
- File transfer from the host to the device.

An inefficient input pipeline can severely slow down your application. An application is considered **input bound** when it spends a significant portion of time in the input pipeline. Use the insights obtained from the input pipeline analyzer to understand where the input pipeline is inefficient.

The input pipeline analyzer tells you immediately whether your program is input bound and walks you through device- and host-side analysis to debug performance bottlenecks at any stage in the input pipeline.

Check the guidance on input pipeline performance for recommended best practices to optimize your data input pipelines.

## Input pipeline dashboard

To open the input pipeline analyzer, select **Profile**, then select **input\_pipeline\_analyzer** from the **Tools** dropdown.

INACTIVE ▾ C ⚙️ ?

### Summary of input-pipeline analysis

Your program is **HIGHLY** input-bound because 81.4% of the total step time sampled is waiting for input. Therefore, you should first focus on reducing the input time.

#### Recommendation for next step:

Look at Section 3 for the breakdown of input time on the host.

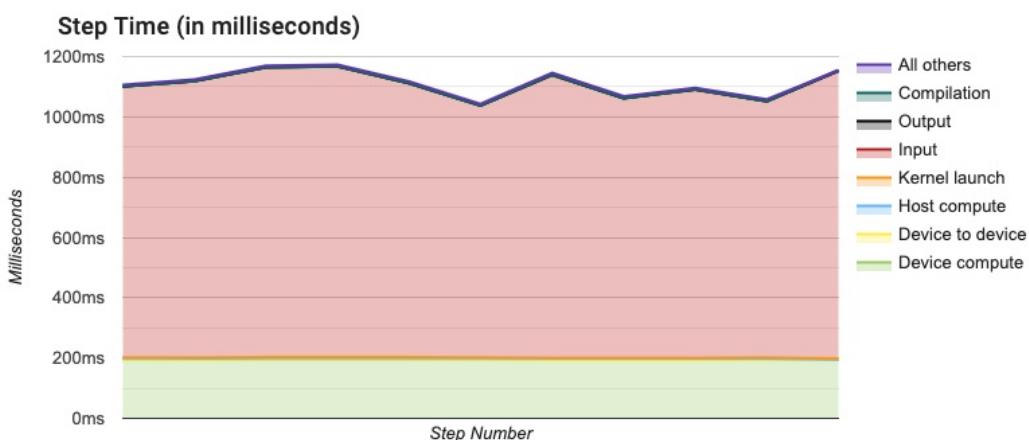
### Device-side analysis details

#### Device step time

##### Device step-time statistics

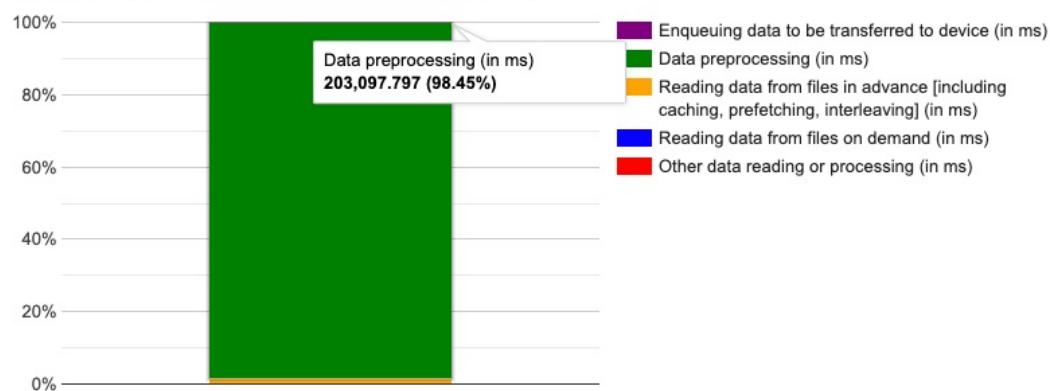
Average: 1114.5 ms ( $\sigma = 44.8$  ms)

Range: 1043.4 - 1173.6 ms



### Host-side analysis details

#### Breakdown of input processing time on the host



#### What can be done to reduce above components of the host input time:

Enqueuing data: you may want to combine small input data chunks into fewer but larger chunks.

Data preprocessing: you may increase num\_parallel\_calls in [Dataset map\(\)](#) or preprocess the data OFFLINE.

Reading data from files in advance: you may tune parameters in the following tf.data API ([prefetch size](#), [interleave cycle length](#), [reader buffer size](#))

Reading data from files on demand: you should read data IN ADVANCE using the following tf.data API ([prefetch](#), [interleave](#), [reader buffer](#))

Other data reading or processing: you may consider using the [tf.data API](#) (if you are not using it now)

### Input Op statistics

The dashboard contains three sections:

- Summary:** Summarizes the overall input pipeline with information on whether your application is input bound and, if so, by how much.
- Device-side analysis:** Displays detailed, device-side analysis results, including the device step-time and the range of device time spent waiting for input data across cores at each step.

3. **Host-side analysis:** Shows a detailed analysis on the host side, including a breakdown of input processing time on the host.

## Input pipeline summary

The **Summary** reports if your program is input bound by presenting the percentage of device time spent on waiting for input from the host. If you are using a standard input pipeline that has been instrumented, the tool reports where most of the input processing time is spent.

## Device-side analysis

The device-side analysis provides insights on time spent on the device versus on the host and how much device time was spent waiting for input data from the host.

1. **Step time plotted against step number:** Displays a graph of device step time (in milliseconds) over all the steps sampled. Each step is broken into the multiple categories (with different colors) of where time is spent. The red area corresponds to the portion of the step time the devices were sitting idle waiting for input data from the host. The green area shows how much of the time the device was actually working.
2. **Step time statistics:** Reports the average, standard deviation, and range ([minimum, maximum]) of the device step time.

## Host-side analysis

The host-side analysis reports a breakdown of the input processing time (the time spent on `tf.data API ops`) on the host into several categories:

- **Reading data from files on demand:** Time spent on reading data from files without caching, prefetching, and interleaving.
- **Reading data from files in advance:** Time spent reading files, including caching, prefetching, and interleaving.
- **Data preprocessing:** Time spent on preprocessing ops, such as image decompression.
- **Enqueuing data to be transferred to device:** Time spent putting data into an infeed queue before transferring the data to the device.

Expand **Input Op Statistics** to inspect the statistics for individual input ops and their categories broken down by execution time.

## Input Op statistics

| Input Op  | Count | Total Time (in ms) | Total Time (as % of total input-processing time) | Total Self Time (in ms) | Total Self Time (as % of total input-processing time) | Category           |
|---|-------|--------------------|--|-------------------------|---|--------------------|
| Iterator::Prefetch::Generator   | 80    | 89,282             | 43.3%  | 89,282                  | 43.3%   | Preprocessing      |
| Iterator::Prefetch  | 88    | 80,228             | 38.9%  | 80,228                  | 38.9%   | Preprocessing      |
| Iterator::Model::Prefetch::Rebatch::Prefetch::MapAndBatch   | 10    | 11,161             | 5.4%   | 11,161                  | 5.4%  | Preprocessing      |
| Iterator::Model::Prefetch   | 87    | 11,161             | 5.4%   | 11,161                  | 5.4%  | Preprocessing      |
| Iterator::Model::Prefetch::Rebatch::Prefetch  | 10    | 11,160             | 5.4%   | 11,160                  | 5.4%  | Preprocessing      |
| Iterator::Model::Prefetch::Rebatch::Prefetch::MapAndBatch::ShuffleAndRepeat::ParallelInterleaveV3[94]::FlatMap[0]::TFRecord | 3     | 868                | 0.4%   | 868                     | 0.4%  | Advanced file read |
| Iterator::Model::Prefetch::Rebatch::Prefetch::MapAndBatch::ShuffleAndRepeat::ParallelInterleaveV3[91]::FlatMap[0]::TFRecord | 3     | 743                | 0.4%   | 743                     | 0.4%  | Advanced file read |
| Iterator::Model::Prefetch::Rebatch::Prefetch::MapAndBatch::ShuffleAndRepeat::ParallelInterleaveV3[97]::FlatMap[0]::TFRecord | 3     | 521                | 0.3%   | 521                     | 0.3%  | Advanced file read |

A source data table will appear with each entry containing the following information:

1. **Input Op:** Shows the TensorFlow op name of the input op.
2. **Count:** Shows the total number of instances of op execution during the profiling period.
3. **Total Time (in ms):** Shows the cumulative sum of time spent on each of those instances.
4. **Total Time %:** Shows the total time spent on an op as a fraction of the total time spent in input processing.
5. **Total Self Time (in ms):** Shows the cumulative sum of the self time spent on each of those instances. The self time here measures the time spent inside the function body, excluding the time spent in the function it calls.
6. **Total Self Time %:** Shows the total self time as a fraction of the total time spent in input processing.
7. **Category:** Shows the processing category of the input op.

## TensorFlow stats

The TensorFlow Stats tool displays the performance of every TensorFlow op (op) that is executed on the host or device during a profiling session.

# TensorFlow Stats

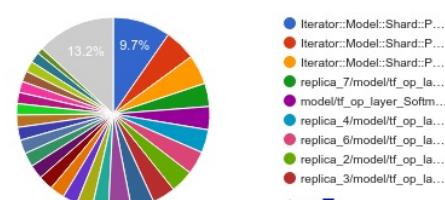
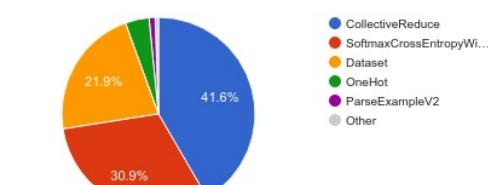
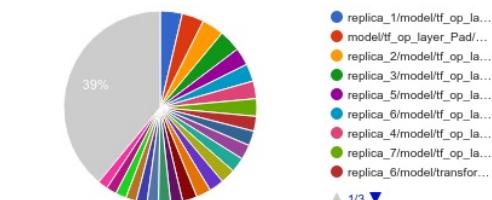
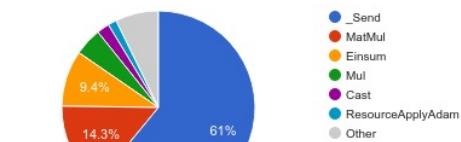
(1) In the charts and table below, "IDLE" represents the portion of the total execution time on device (or host) that is idle.

(2) In the pie charts, the "Other" sector represents the sum of sectors that are too small to be shown individually.

 Export as CSV

### Include IDLE time in statistics

No ▾



TensorFlow operations

## Host/device



### Type



Operations



Note: To avoid sluggishness, only the 1000 most time-consuming operations out of the total 36340 operations are shown in the table below.

| Rank | Host/device | Type  | Operation                                 | #Occurrences | Total time (us)   | Avg. time (us) | Total self-time (us) | Avg. self-time (us) | Total self-time on Device (%) | Cumulative total-self time on Device (%) | Total self-time on Host (%) | Cumulative total-self time on Host (%) |
|------|-------------|-------|---|--------------|-------------------|----------------|----------------------|---------------------|-------------------------------|--|-----------------------------|--|
| 1    | Device      | _Send | replica_1/model/tf_op_layer_Pad/Pad/_2775 | 11           | 1,98<br>8,06<br>5 | 180,<br>733    | 1,98<br>8,06<br>5    | 180,<br>733         | 3.7%                          | 3.7%                                     | 0%                          | 0%                                     |
| 2    | Device      | _Send | model/tf_op_layer_Pad/Pad/_2771           | 11           | 1,98<br>1,24<br>0 | 180,<br>113    | 1,98<br>1,24<br>0    | 180,<br>113         | 3.7%                          | 7.4%                                     | 0%                          | 0%                                     |
| 3    | Device      | _Send | replica_2/model/tf_op_layer_Pad/Pad/_2779 | 11           | 1,97<br>9,88<br>9 | 179,<br>990    | 1,97<br>9,88<br>9    | 179,<br>990         | 3.7%                          | 11%                                      | 0%                          | 0%                                     |
| 4    | Device      | _Send | replica_3/model/tf_op_layer_Pad/Pad/_2783 | 11           | 1,97<br>1,44<br>6 | 179,<br>222    | 1,97<br>1,44<br>6    | 179,<br>222         | 3.7%                          | 14.7%                                    | 0%                          | 0%                                     |
| 5    | Device      | _Send | replica_5/model/tf_op_layer_Pad/Pad/_2791 | 11           | 1,58<br>8,52<br>1 | 144,<br>411    | 1,58<br>8,52<br>1    | 144,<br>411         | 2.9%                          | 17.6%                                    | 0%                          | 0%                                     |

The tool displays performance information in two panes:

- The upper pane displays up to four pie charts:
    1. The distribution of self-execution time of each op on the host.
    2. The distribution of self-execution time of each op type on the host.
    3. The distribution of self-execution time of each op on the device.
    4. The distribution of self-execution time of each op type on the device.
  - The lower pane shows a table that reports data about TensorFlow ops with one row for each op and one column for each type of data (sort columns by clicking the heading of the column). Click the **Export as CSV button** on the right side of the upper pane to export the data from this table as a CSV file.  
Note that:
    - If any ops have child ops:
      - The total "accumulated" time of an op includes the time spent inside the child ops.
      - The total "self" time of an op does not include the time spent inside the child ops.
    - If an op executes on the host:
      - The percentage of the total self-time on device incurred by the op on will be 0.
      - The cumulative percentage of the total self-time on device up to and including this op will be 0.
    - If an op executes on the device:
      - The percentage of the total self-time on host incurred by this op will be 0.
      - The cumulative percentage of the total self-time on host up to and including this op will be 0.

You can choose to include or exclude Idle time in the pie charts and table

## Trace viewer

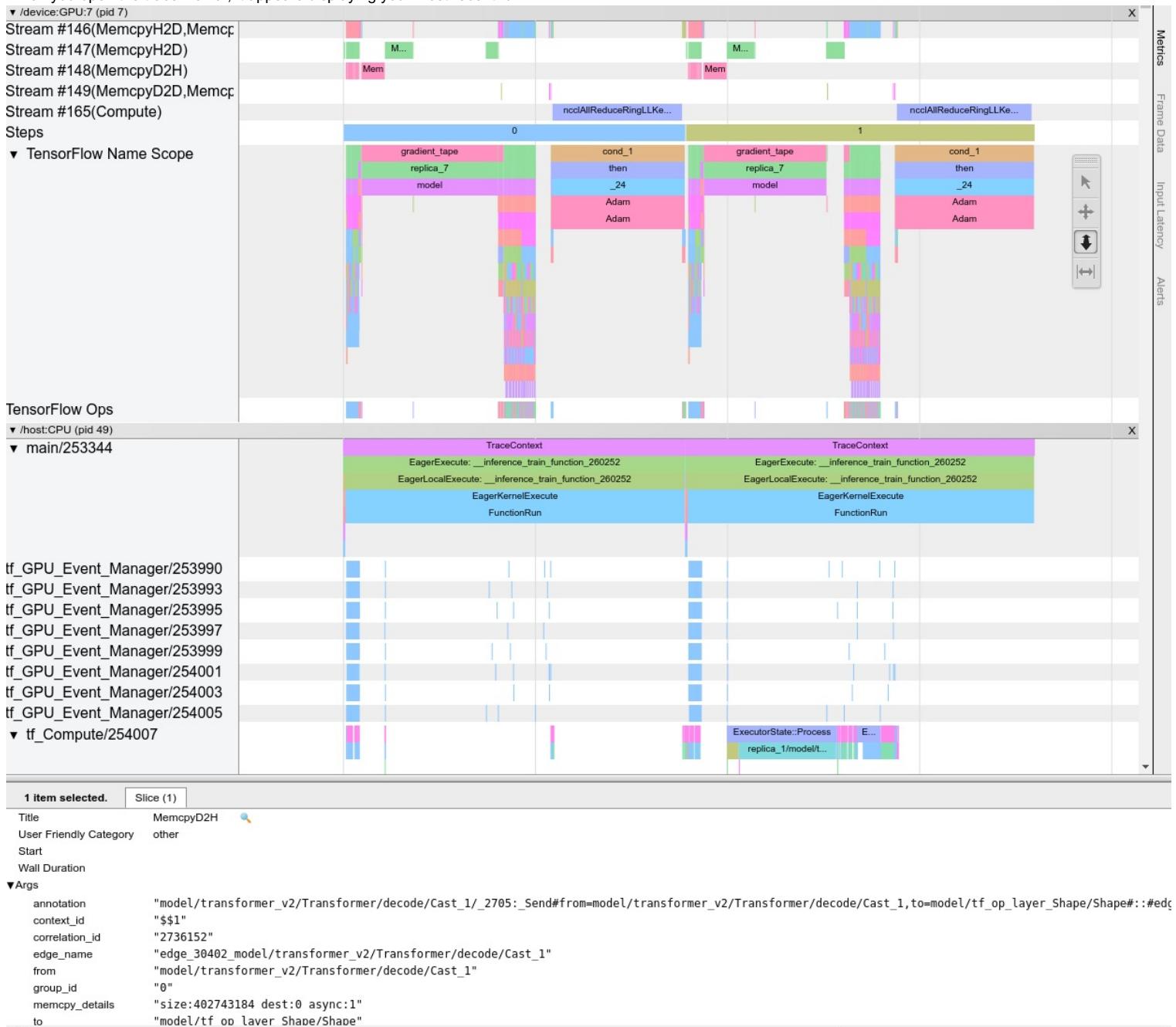
The trace viewer displays a timeline that shows:

- Durations for the ops that were executed by your TensorFlow model
- Which part of the system (host or device) executed an op. Typically, the host executes input operations, preprocesses training data and transfers it to the device, while the device executes the actual model training

The trace viewer allows you to identify performance problems in your model, then take steps to resolve them. For example, at a high level, you can identify whether input or model training is taking the majority of the time. Drilling down, you can identify which ops take the longest to execute. Note that the trace viewer is limited to 1 million events per device.

## Trace viewer interface

When you open the trace viewer, it appears displaying your most recent run:



The Timeline pane contains the following elements:

1. **Top bar:** Contains various auxiliary controls.
2. **Time axis:** Shows time relative to the beginning of the trace.
3. **Section and track labels:** Each section contains multiple tracks and has a triangle on the left that you can click to expand and collapse the section. There is one section for every processing element in the system.
4. **Tool selector:** Contains various tools for interacting with the trace viewer such as Zoom, Pan, Select, and Timing. Use the Timing tool to mark a time interval.
5. **Events:** These show the time during which an op was executed or the duration of meta-events, such as training steps.

## Sections and tracks

The trace viewer contains the following sections:

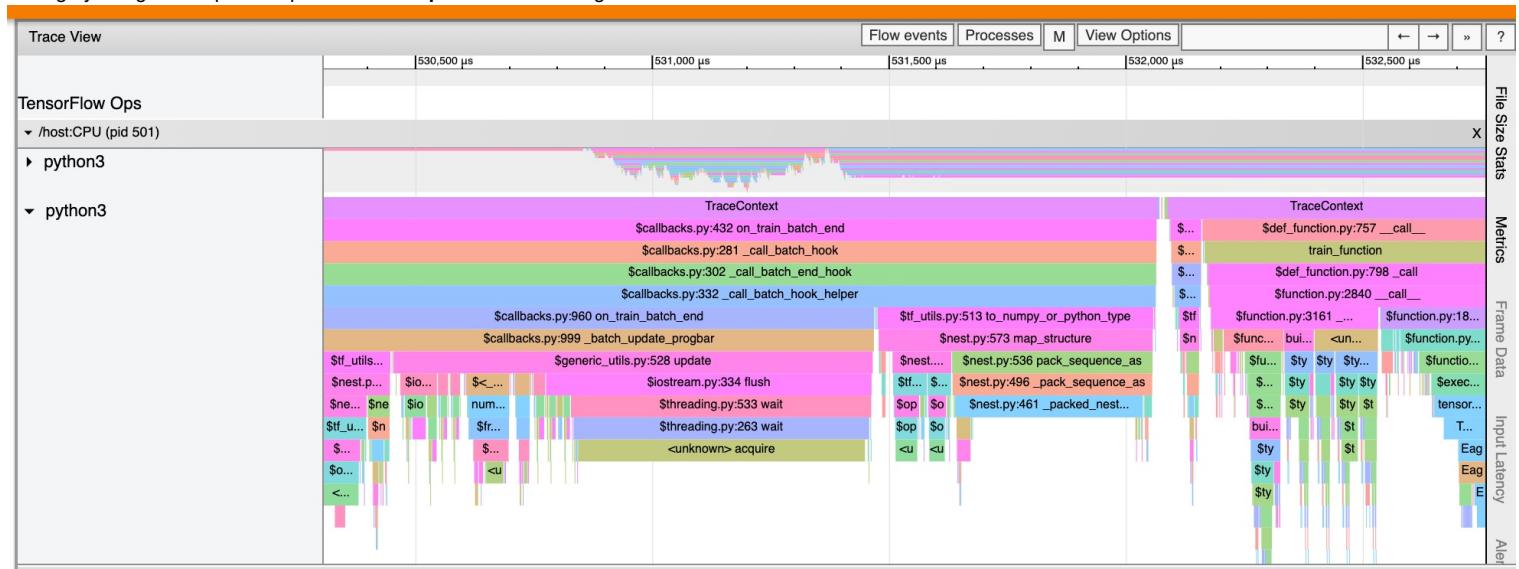
- **One section for each device node**, labeled with the number of the device chip and the device node within the chip (for example, /device:GPU:0 (pid 0)). Each device node section contains the following tracks:
  - **Step:** Shows the duration of the training steps that were running on the device
  - **TensorFlow Ops:** Shows the ops executed on the device
  - **XLA Ops:** Shows XLA (<https://www.tensorflow.org/xla/>) operations (ops) that ran on the device if XLA is the compiler used (each TensorFlow op is translated into one or several XLA ops. The XLA compiler translates the XLA ops into code that runs on the device).
- **One section for threads running on the host machine's CPU**, labeled "**Host Threads**". The section contains one track for each CPU thread. Note that you can

ignore the information displayed alongside the section labels.

## Events

Events within the timeline are displayed in different colors; the colors themselves have no specific meaning.

The trace viewer can also display traces of Python function calls in your TensorFlow program. If you use the `tf.profiler.experimental.start` API, you can enable Python tracing by using the `ProfilerOptions` namedtuple when starting profiling. Alternatively, if you use the sampling mode for profiling, you can select the level of tracing by using the dropdown options in the **Capture Profile** dialog.



## GPU kernel stats

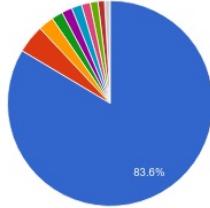
This tool shows performance statistics and the originating op for every GPU accelerated kernel.



### GPU Kernel Stats

[Export as CSV](#)

Top 10 Kernels with highest Total Duration



- nccAllReduceRingLLKernel\_sum\_f32(nccColl)
- volta\_fp16\_sgemm\_fp16\_128x128\_nt
- volta\_fp16\_sgemm\_fp16\_128x64\_nt
- volta\_fp16\_sgemm\_fp16\_64x32\_sliced1x4\_nt
- volta\_fp16\_sgemm\_fp16\_32x128\_nt
- volta\_fp16\_sgemm\_fp16\_128x128\_nn
- volta\_fp16\_sgemm\_fp16\_128x128\_idp0...
- volta\_fp16\_sgemm\_fp16\_128x128\_idp0...
- void tensorflow::func::ApplyAdamKernel<float>...
- Other

## TensorFlow operations

Kernel Name  Op Name

| Rank | Kernel Name                               | Registers per thread | Shared Mem bytes | Block dim | Grid dim | Kernel uses TensorCore | Op is TensorCore eligible | Op Name  | Occurrences | Total Duration (us) | Avg Duration (us) | Min Duration (us) | Max Duration (us) |
|------|---|----------------------|------------------|-----------|----------|------------------------|---------------------------|--|-------------|---------------------|-------------------|-------------------|-------------------|
| 1    | nccAllReduceRingLLKernel_sum_f32(nccColl) | 0                    | 0                | 1,1,1     | 1,1,1    | x                      | x                         |  | 88          | 61,980,7            | 704,326,          | 645,700,          | 800,806,          |
|      |   |                      |                  |           |          |                        |                           |  |             | 61,697,0            | 837,465,          | 637,000           | 741,000           |
| 2    | volta_fp16_sgemm_fp16_32x128_nt           | 0                    | 0                | 1,1,1     | 1,1,1    | x                      | x                         | gradient_tape/replica_2/model/transformer_v2/Transformer/decode/embedding_shared_weights_1/presoftmax_linear/MatMul/MatMul_1 | 11          | 151,729,            | 13,793,5          | 92,636,3          | 13,598,6          |
|      |   |                      |                  |           |          |                        |                           |  |             | 519,000             | 64                | 61,000            | 13,979,5          |
| 3    | volta_fp16_sgemm_fp16_32x128_nt           | 0                    | 0                | 1,1,1     | 1,1,1    | x                      | x                         | gradient_tape/replica_5/model/transformer_v2/Transformer/decode/embedding_shared_weights_1/presoftmax_linear/MatMul/MatMul_1 | 11          | 151,671,            | 13,788,3          | 90,545,4          | 13,595,6          |
|      |   |                      |                  |           |          |                        |                           |  |             | 306,000             | 55                | 30,000            | 13,987,7          |
| 4    | volta_fp16_sgemm_fp16_32x128_nt           | 0                    | 0                | 1,1,1     | 1,1,1    | x                      | x                         | gradient_tape/replica_3/model/transformer_v2/Transformer/decode/embedding_shared_weights_1/presoftmax_linear/MatMul/MatMul_1 | 11          | 151,539,            | 13,776,3          | 47,454,5          | 13,516,7          |
|      |   |                      |                  |           |          |                        |                           |  |             | 822,000             | 45                | 19,000            | 13,986,7          |
| 5    | volta_fp16_sgemm_fp16_32x128_nt           | 0                    | 0                | 1,1,1     | 1,1,1    | x                      | x                         | gradient_tape/replica_1/model/transformer_v2/Transformer/decode/embedding_shared_weights_1/presoftmax_linear/MatMul/MatMul_1 | 11          | 151,483,            | 13,771,2          | 70,272,7          | 13,538,2          |
|      |   |                      |                  |           |          |                        |                           |  |             | 973,000             | 64                | 64,000            | 13,983,7          |
| 6    | volta_fp16_sgemm_fp16_32x128_nt           | 0                    | 0                | 1,1,1     | 1,1,1    | x                      | x                         | gradient_tape/replica_6/model/transformer_v2/Transformer/decode/embedding_shared_weights_1/presoftmax_linear/MatMul/MatMul_1 | 11          | 151,417,            | 13,765,1          | 13,540,2          | 13,985,7          |
|      |   |                      |                  |           |          |                        |                           |  |             | 68,000              | 18                | 81,000            | 19,000            |
| 7    | volta_fp16_sgemm_fp16_32x128_nt           | 0                    | 0                | 1,1,1     | 1,1,1    | x                      | x                         | gradient_tape/replica_7/model/transformer_v2/Transformer/decode/embedding_shared_weights_1/presoftmax_linear/MatMul/MatMul_1 | 11          | 151,407,            | 13,764,3          | 13,539,2          | 13,979,5          |
|      |   |                      |                  |           |          |                        |                           |  |             | 445,000             | 18                | 21,000            | 37,000            |
| 8    | volta_fp16_sgemm_fp16_32x128_nt           | 0                    | 0                | 1,1,1     | 1,1,1    | x                      | x                         | gradient_tape/replica_4/model/transformer_v2/Transformer/decode/embedding_shared_weights_1/presoftmax_linear/MatMul/MatMul_1 | 11          | 151,323,            | 13,756,7          | 20,090,9          | 13,433,7          |
|      |   |                      |                  |           |          |                        |                           |  |             | 921,000             | 09                | 13,433,7          | 13,982,6          |
| 9    | volta_fp16_sgemm_fp16_32x128_nt           | 0                    | 0                | 1,1,1     | 1,1,1    | x                      | x                         | gradient_tape/model/transformer_v2/Transformer/decode/embedding_shared_weights_1/presoftmax_linear/MatMul/MatMul_1           | 11          | 151,292,            | 13,753,9          | 20,272,7          | 13,422,5          |
|      |   |                      |                  |           |          |                        |                           |  |             | 925,000             | 27                | 13,422,5          | 13,980,5          |
| 10   | volta_fp16_sgemm_fp16_128x64_nt           | 0                    | 0                | 1,1,1     | 1,1,1    | x                      | x                         | replica_3/model/transformer_v2/Transformer/decode/embedding_shared_weights_1/presoftmax_linear/MatMul                        | 11          | 151,020,            | 13,729,1          | 50,636,3          | 13,301,6          |
|      |   |                      |                  |           |          |                        |                           |  |             | 657,000             | 64                | 80,000            | 14,015,4          |
| 11   | volta_fp16_sgemm_fp16_128x64_nt           | 0                    | 0                | 1,1,1     | 1,1,1    | x                      | x                         | replica_2/model/transformer_v2/Transformer/decode/embedding_shared_weights_1/presoftmax_linear/MatMul                        | 11          | 150,843,            | 13,713,0          | 69,636,3          | 13,300,6          |
|      |   |                      |                  |           |          |                        |                           |  |             | 766,000             | 64                | 79,000            | 14,040,0          |
| 12   | volta_fp16_sgemm_fp16_128x64_nt           | 0                    | 0                | 1,1,1     | 1,1,1    | x                      | x                         | replica_4/model/transformer_v2/Transformer/decode/embedding_shared_weights_1/presoftmax_linear/MatMul                        | 11          | 150,832,            | 13,712,0          | 37,090,9          | 13,298,6          |
|      |   |                      |                  |           |          |                        |                           |  |             | 408,000             | 09                | 22,000            | 14,004,1          |
| 13   | volta_fp16_sgemm_fp16_128x64_nt           | 0                    | 0                | 1,1,1     | 1,1,1    | x                      | x                         | replica_6/model/transformer_v2/Transformer/decode/embedding_shared_weights_1/presoftmax_linear/MatMul                        | 11          | 150,822,            | 13,711,1          | 126,000           | 13,303,7          |
|      |   |                      |                  |           |          |                        |                           |  |             | 126,000             | 38                | 39,000            | 13,988,7          |

The tool displays information in two panes:

- The upper pane displays a pie chart which shows the CUDA kernels that have the highest total time elapsed.
- The lower pane displays a table with the following data for each unique kernel-op pair:
  - A rank in descending order of total elapsed GPU duration grouped by kernel-op pair.

- The name of the launched kernel.
- The number of GPU registers used by the kernel.
- The total size of shared (static + dynamic shared) memory used in bytes.
- The block dimension expressed as `blockDim.x`, `blockDim.y`, `blockDim.z`.
- The grid dimensions expressed as `gridDim.x`, `gridDim.y`, `gridDim.z`.
- Whether the op is eligible to use [Tensor Cores](https://www.nvidia.com/en-gb/data-center/tensor-cores/) (<https://www.nvidia.com/en-gb/data-center/tensor-cores/>).
- Whether the kernel contains Tensor Core instructions.
- The name of the op that launched this kernel.
- The number of occurrences of this kernel-op pair.
- The total elapsed GPU time in microseconds.
- The average elapsed GPU time in microseconds.
- The minimum elapsed GPU time in microseconds.
- The maximum elapsed GPU time in microseconds.

## Memory profile tool {`: id = 'memory_profile_tool'`}

The **Memory Profile** tool monitors the memory usage of your device during the profiling interval. You can use this tool to:

- Debug out of memory (OOM) issues by pinpointing peak memory usage and the corresponding memory allocation to TensorFlow ops. You can also debug OOM issues that may arise when you run [multi-tenancy](https://arxiv.org/pdf/1901.06887.pdf) (<https://arxiv.org/pdf/1901.06887.pdf>) inference.
- Debug memory fragmentation issues.

The memory profile tool displays data in three sections:

1. **Memory Profile Summary**
2. **Memory Timeline Graph**
3. **Memory Breakdown Table**

### Memory profile summary

This section displays a high-level summary of the memory profile of your TensorFlow program as shown below:

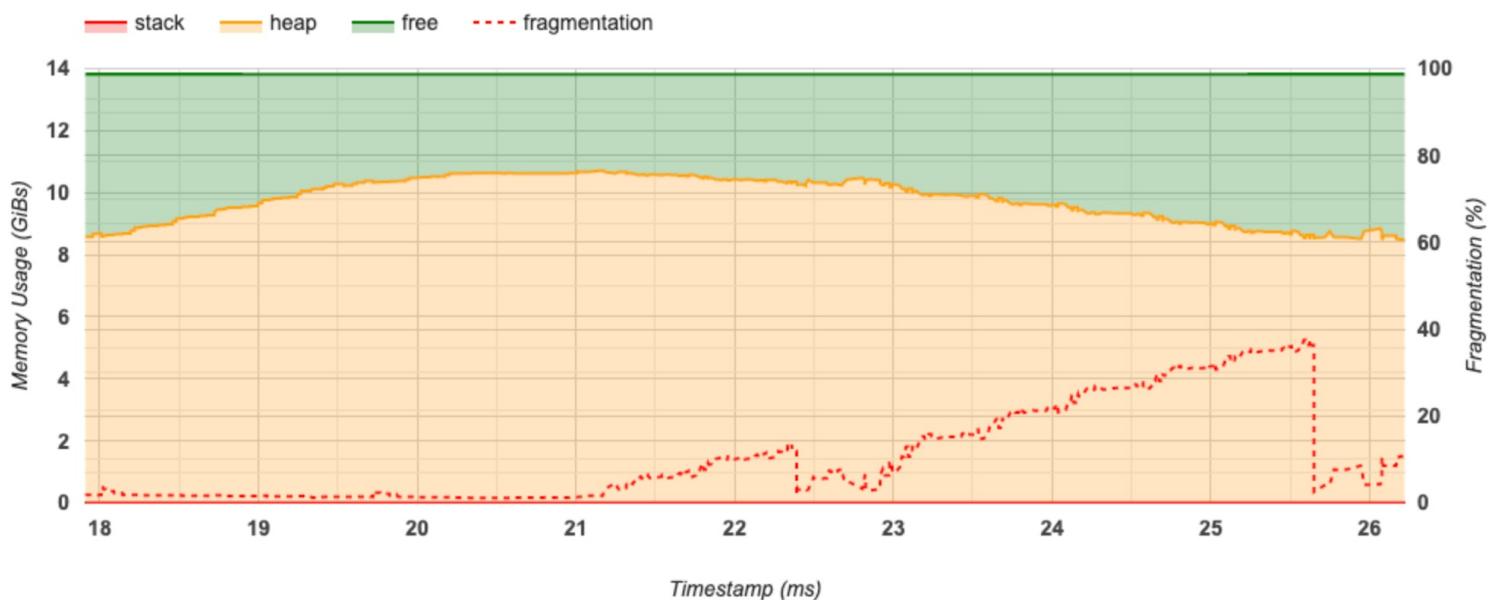
| Memory ID  | GPU_...           |
|--|-------------------|
| <i>show memory profile for selected device</i>                           |                   |
| <b>#Allocation</b>   | <b>489</b>        |
| <b>#Deallocation</b>   | <b>511</b>        |
| <b>Memory Capacity</b>   | <b>13.82 GiBs</b> |
| <b>Peak Heap Usage</b><br><i>high water mark in lifetime</i>             | <b>10.87 GiBs</b> |
| <b>Peak Memory Usage</b><br><i>stack + heap, within profiling window</i> | <b>10.72 GiBs</b> |
| • Timestamp: 21.2 ms   |                   |
| • Stack Reservation: 0.00 GiBs   |                   |
| • Heap Allocation: 10.72 GiBs  |                   |
| • Free Memory: 3.10 GiBs   |                   |
| • Fragmentation: 1.66%   |                   |

The memory profile summary has six fields:

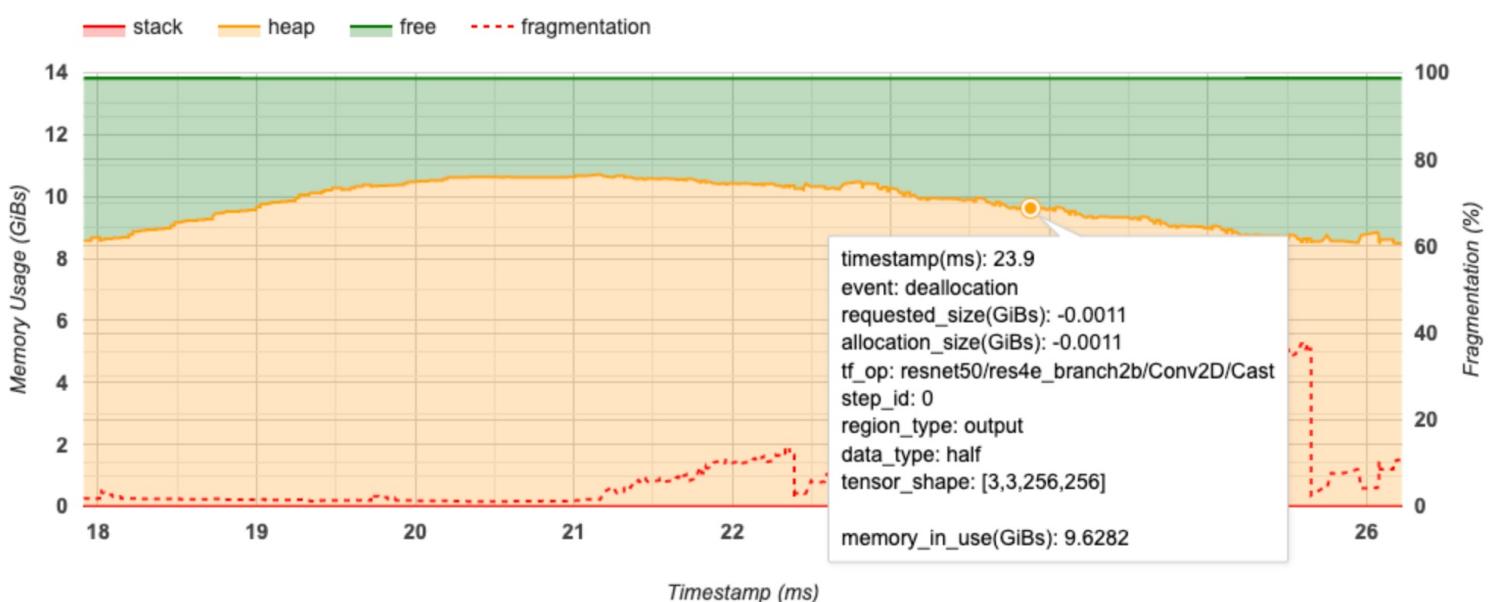
1. **Memory ID**: Dropdown which lists all available device memory systems. Select the memory system you want to view from the dropdown.
2. **#Allocation**: The number of memory allocations made during the profiling interval.
3. **#Deallocation**: The number of memory deallocations in the profiling interval
4. **Memory Capacity**: The total capacity (in GiBs) of the memory system that you select.
5. **Peak Heap Usage**: The peak memory usage (in GiBs) since the model started running.
6. **Peak Memory Usage**: The peak memory usage (in GiBs) in the profiling interval. This field contains the following sub-fields:
  1. **Timestamp**: The timestamp of when the peak memory usage occurred on the Timeline Graph.
  2. **Stack Reservation**: Amount of memory reserved on the stack (in GiBs).
  3. **Heap Allocation**: Amount of memory allocated on the heap (in GiBs).
  4. **Free Memory**: Amount of free memory (in GiBs). The Memory Capacity is the sum total of the Stack Reservation, Heap Allocation, and Free Memory.
  5. **Fragmentation**: The percentage of fragmentation (lower is better). It is calculated as a percentage of  $(1 - \text{Size of the largest chunk of free memory} / \text{Total free memory})$ .

### Memory timeline graph

This section displays a plot of the memory usage (in GiBs) and the percentage of fragmentation versus time (in ms).



The X-axis represents the timeline (in ms) of the profiling interval. The Y-axis on the left represents the memory usage (in GiBs) and the Y-axis on the right represents the percentage of fragmentation. At each point in time on the X-axis, the total memory is broken down into three categories: stack (in red), heap (in orange), and free (in green). Hover over a specific timestamp to view the details about the memory allocation/deallocation events at that point like below:



The pop-up window displays the following information:

- **timestamp(ms)**: The location of the selected event on the timeline.
- **event**: The type of event (allocation or deallocation).
- **requested\_size(GiBs)**: The amount of memory requested. This will be a negative number for deallocation events.
- **allocation\_size(GiBs)**: The actual amount of memory allocated. This will be a negative number for deallocation events.
- **tf\_op**: The TensorFlow op that requests the allocation/deallocation.
- **step\_id**: The training step in which this event occurred.
- **region\_type**: The data entity type that this allocated memory is for. Possible values are `temp` for temporaries, `output` for activations and gradients, and `persist / dynamic` for weights and constants.
- **data\_type**: The tensor element type (e.g., `uint8` for 8-bit unsigned integer).
- **tensor\_shape**: The shape of the tensor being allocated/deallocated.
- **memory\_in\_use(GiBs)**: The total memory that is in use at this point of time.

## Memory breakdown table

This table shows the active memory allocations at the point of peak memory usage in the profiling interval.

## Memory Breakdown Table

Operation



Note: Showing active memory allocations at peak usage within the profiling window. To avoid sluggishness, only the allocations with size over 1MiB are shown in the table below.

| Op Name  | Allocation Size (GiBs) | Requested Size (GiBs) | Occurrences | Region type     | Data type | Shape            |
|--|------------------------|-----------------------|-------------|-----------------|-----------|------------------|
| resnet50/bn5a_branch2c/FusedBatchNormV3                          | 0.048                  | 0.048                 | 1           | output          | half      | [256,7,7,2048]   |
| gradient_tape/resnet50/bn4f_branch2c/FusedBatchNormGradV3        | 0.002                  | 0.001                 | 1           | temp            | uint8     | [1606752]        |
| resnet50/bn4f_branch2c/FusedBatchNormV3                          | 0.009                  | 0.006                 | 1           | output          | float     | [1605888]        |
| gradient_tape/resnet50/res4f_branch2b/Conv2D/Conv2DBackpropInput | 0.024                  | 0.024                 | 1           | temp            | half      | [256,14,14,256]  |
| resnet50/bn4f_branch2a/FusedBatchNormV3                          | 0.024                  | 0.024                 | 1           | output          | half      | [256,14,14,256]  |
| preallocated/unknown   | 0.241                  | 0.241                 | 1           | persist/dynamic | INVALID   | unknown          |
| gradient_tape/resnet50/bn4d_branch2c/FusedBatchNormGradV3        | 0.096                  | 0.096                 | 1           | output          | half      | [256,14,14,1024] |
| gradient_tape/resnet50/bn4d_branch2b/FusedBatchNormGradV3        | 0.024                  | 0.024                 | 1           | output          | half      | [256,14,14,256]  |
| resnet50/bn4e_branch2c/FusedBatchNormV3                          | 0.096                  | 0.096                 | 1           | output          | half      | [256,14,14,1024] |
| gradient_tape/resnet50/res4d_branch2c/Conv2D/Conv2DBackpropInput | 0.024                  | 0.024                 | 1           | output          | half      | [256,14,14,256]  |
| gradient_tape/resnet50/bn4d_branch2c/FusedBatchNormGradV3        | 0.002                  | 0.001                 | 1           | temp            | uint8     | [1606752]        |
| gradient_tape/resnet50/res4d_branch2b/Conv2D/Conv2DBackpropInput | 0.024                  | 0.024                 | 1           | temp            | half      | [256,14,14,256]  |
| gradient_tape/resnet50/bn4f_branch2c/FusedBatchNormGradV3        | 0.096                  | 0.096                 | 1           | output          | half      | [256,14,14,1024] |
| resnet50/bn4f_branch2c/FusedBatchNormV3                          | 0.096                  | 0.096                 | 1           | output          | half      | [256,14,14,1024] |

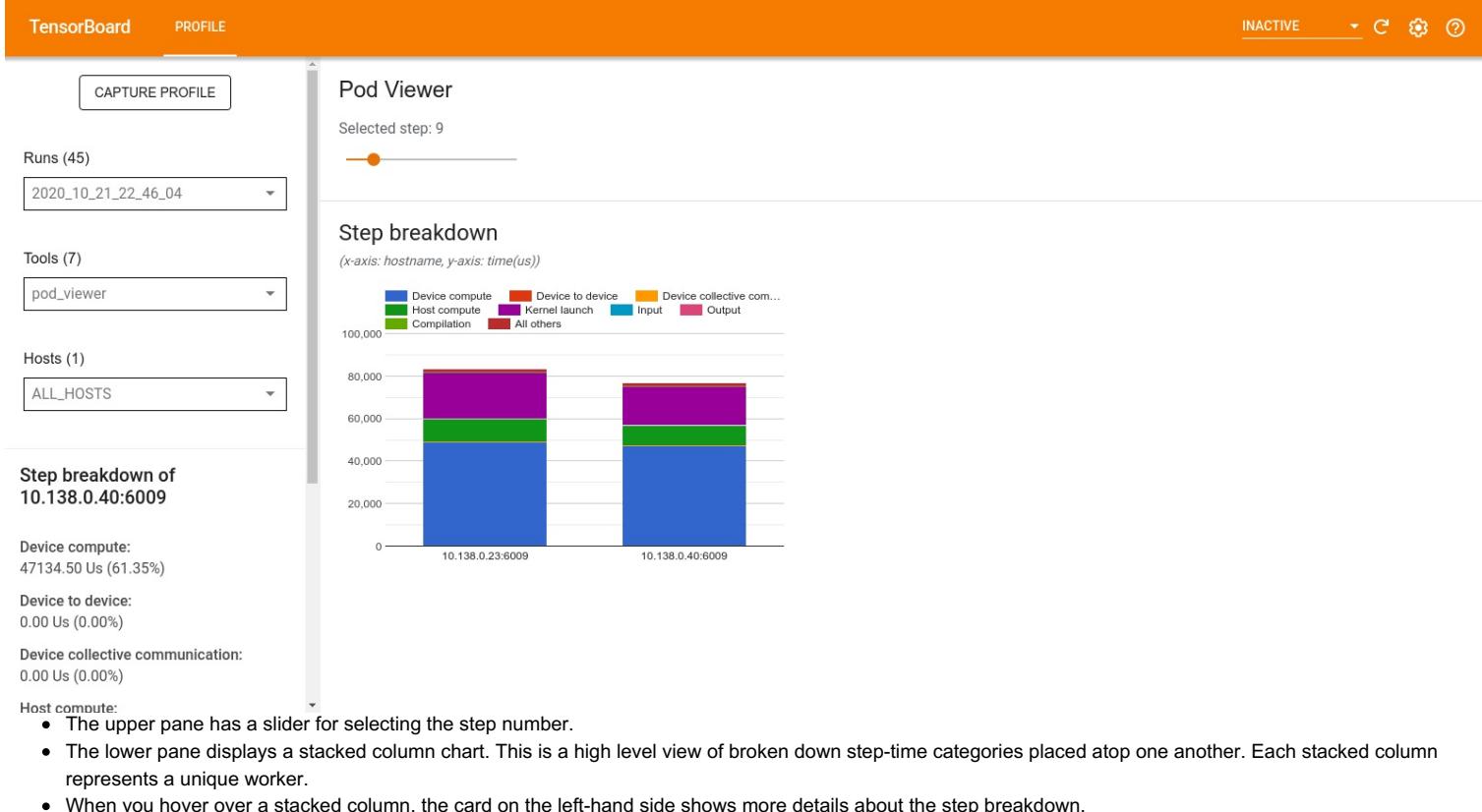
There is one row for each TensorFlow Op and each row has the following columns:

- **Op Name:** The name of the TensorFlow op.
- **Allocation Size (GiBs):** The total amount of memory allocated to this op.
- **Requested Size (GiBs):** The total amount of memory requested for this op.
- **Occurrences:** The number of allocations for this op.
- **Region type:** The data entity type that this allocated memory is for. Possible values are `temp` for temporaries, `output` for activations and gradients, and `persist / dynamic` for weights and constants.
- **Data type:** The tensor element type.
- **Shape:** The shape of the allocated tensors.

Note: You can sort any column in the table and also filter rows by op name.

## Pod viewer

The Pod Viewer tool shows the breakdown of a training step across all workers.



## tf.data bottleneck analysis

Warning: This tool is experimental. Please open a [GitHub Issue](https://github.com/tensorflow/profiler/issues) (<https://github.com/tensorflow/profiler/issues>) if the analysis result seems incorrect.

The `tf.data` bottleneck analysis tool automatically detects bottlenecks in `tf.data` input pipelines in your program and provides recommendations on how to fix them. It works with any program using `tf.data` regardless of the platform (CPU/GPU/TPU). Its analysis and recommendations are based on this [guide](#) ([https://www.tensorflow.org/guide/data\\_performance\\_analysis](https://www.tensorflow.org/guide/data_performance_analysis)).

It detects a bottleneck by following these steps:

1. Find the most input bound host.
2. Find the slowest execution of a `tf.data` input pipeline.
3. Reconstruct the input pipeline graph from the profiler trace.
4. Find the critical path in the input pipeline graph.
5. Identify the slowest transformation on the critical path as a bottleneck.

The UI is divided into three sections: **Performance Analysis Summary**, **Summary of All Input Pipelines** and **Input Pipeline Graph**.

### Performance analysis summary

## Performance Analysis Summary

Your profile has a tf.data input pipeline slower than 50 us. Below shows a bottleneck in the slow input pipeline and a suggestion on how to fix it.

| Host   | Input Pipeline | Max Latency (us) | Bottleneck  | Suggestion  |
|--|----------------|------------------|---|---|
| 0.mkuchnik.dragonfish.2x2.resnet.train.tpu_worker.t<br>pu-perf-team.is.borg.google.com | Host:0         | 450,706          | <b>Iterator Type: TFFRecord</b><br><b>Long Name:</b> Iterator::Model::PrivateThreadPool::MaxIntraOpParallelism::Prefetch::ParallelMapV2::ParallelMapV2::MapAn<br>dBatch::ShuffleAndRepeat::ParallelInterleaveV4[230]::FlatMap[0]::TFFRecord | <ol style="list-style-type: none"><li>1. Check the locality of a host and input data. Ideally, they should be in the same cell (or very close, like the same region).</li><li>2. Parallelize reading from this dataset source. See <a href="#">this</a> for more details.</li><li>3. You should use <a href="#">go/readahead</a> for reading from remote storage.</li><li>4. Find more resources <a href="#">here</a>, <a href="#">here</a> and <a href="#">here</a>.</li></ol> |

This section provides the summary of the analysis. It reports on slow `tf.data` input pipelines detected in the profile. This section also shows the most input bound host and its slowest input pipeline with the max latency. Most importantly, it identifies which part of the input pipeline is the bottleneck and how to fix it. The bottleneck information is provided with the iterator type and its long name.

How to read `tf.data` iterator's long name

A long name is formatted as `Iterator::<Dataset_1>::...::<Dataset_n>`. In the long name, `<Dataset_n>` matches the iterator type and the other datasets in the long name represent downstream transformations.

For example, consider the following input pipeline dataset:

```
python
```

```
dataset = tf.data.Dataset.range(10).map(lambda x: x).repeat(2).batch(5)
```

The long names for the iterators from the above dataset will be:

```
Iterator Type | Long Name :----- | :----- Range | Iterator::Batch::Repeat::Map::Range Map | Iterator::Batch::Repeat::Map Repeat |  
Iterator::Batch::Repeat Batch | Iterator::Batch
```

Summary of all input pipelines

## Summary of All Input Pipelines

| Host        | Input Pipeline | Min (us) | Avg (us)  | Max (us)  | # calls | # slow calls |
|-------------|----------------|----------|-----------|-----------|---------|--------------|
| yoet6:14189 | Host:0         | 69,851   | 1,303,671 | 8,286,438 | 22      | 22           |
| yoet6:14192 | Host:0         | 15,964   | 1,082,697 | 4,750,466 | 23      | 23           |
| yoet8:25428 | Host:0         | 48,455   | 1,118,913 | 4,407,822 | 23      | 23           |
| yoet8:25431 | Host:0         | 19,631   | 995,449   | 7,974,405 | 28      | 28           |

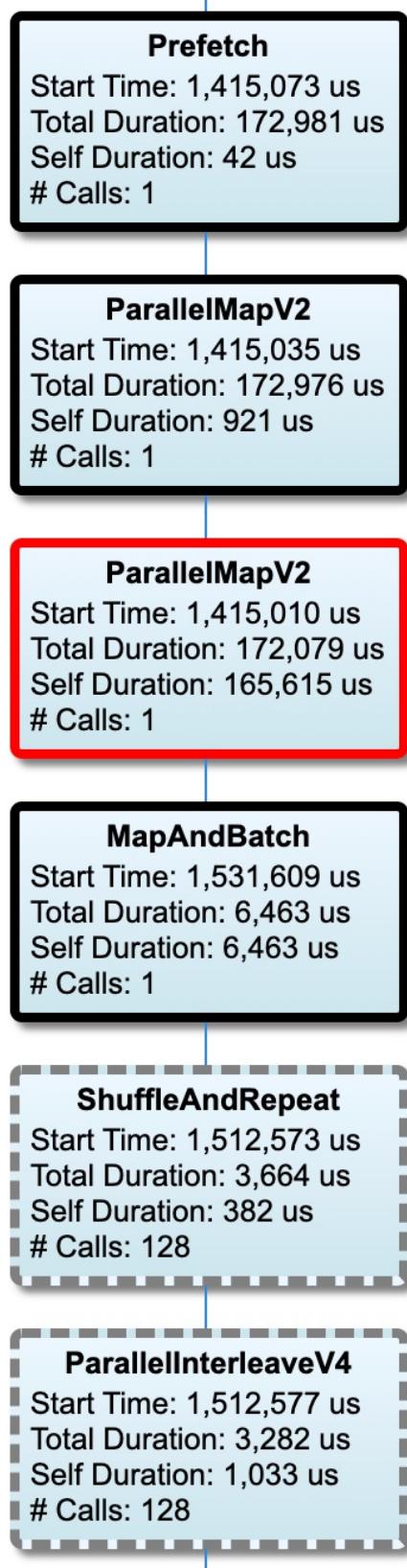
This section provides the summary of all input pipelines across all hosts. Typically there is one input pipeline. When using the distribution strategy, there is one host input pipeline running the program's `tf.data` code and multiple device input pipelines retrieving data from the host input pipeline and transferring it to the devices.

For each input pipeline, it shows the statistics of its execution time. A call is counted as slow if it takes longer than 50  $\mu$ s.

Input pipeline graph

| Host           | Input Pipeline | Rank |
|----------------|----------------|------|
| 0.mkuchnik.... | Host:0         | 4    |

This section shows the input pipeline graph with the execution time information. You can use "Host" and "Input Pipeline" to choose which host and input pipeline to see. Executions of the input pipeline are sorted by the execution time in descending order which you can choose using the **Rank** dropdown.



The nodes on the critical path have bold outlines. The bottleneck node, which is the node with the longest self time on the critical path, has a red outline. The other non-critical nodes have gray dashed outlines.

In each node, **Start Time** indicates the start time of the execution. The same node may be executed multiple times, for example, if there is a `Batch` op in the input pipeline. If it is executed multiple times, it is the start time of the first execution.

**Total Duration** is the wall time of the execution. If it is executed multiple times, it is the sum of the wall times of all executions.

**Self Time** is **Total Time** without the overlapped time with its immediate child nodes.

"# Calls" is the number of times the input pipeline is executed.

## Collect performance data

The TensorFlow Profiler collects host activities and GPU traces of your TensorFlow model. You can configure the Profiler to collect performance data through either the programmatic mode or the sampling mode.

## Profiling APIs

You can use the following APIs to perform profiling.

- Programmatic mode using the TensorBoard Keras Callback (`tf.keras.callbacks.TensorBoard`)
   
```python

# Profile from batches 10 to 15

```
tb_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, profile_batch='10, 15')
```

Train the model and use the TensorBoard Keras callback to collect

performance profiling data

```
model.fit(train_data, steps_per_epoch=20, epochs=5, callbacks=[tb_callback]) ````
```

- Programmatic mode using the `tf.profiler` Function API

```
```python tf.profiler.experimental.start('logdir')
```

Train the model here

```
tf.profiler.experimental.stop() ````
```

- Programmatic mode using the context manager

```
python
with tf.profiler.experimental.Profile('logdir'):
    # Train the model here
    pass
```

Note: Running the Profiler for too long can cause it to run out of memory. It is recommended to profile no more than 10 steps at a time. Avoid profiling the first few batches to avoid inaccuracies due to initialization overhead.

- Sampling mode: Perform on-demand profiling by using `tf.profiler.experimental.server.start` to start a gRPC server with your TensorFlow model run. After starting the gRPC server and running your model, you can capture a profile through the **Capture Profile** button in the TensorBoard profile plugin. Use the script in the Install profiler section above to launch a TensorBoard instance if it is not already running.

As an example,

```
```python
```

Start a profiler server before your model runs.

```
tf.profiler.experimental.server.start(6009)
```

(Model code goes here).

Send a request to the profiler server to collect a trace of your model.

```
tf.profiler.experimental.client.trace('grpc://localhost:6009', 'gs://your_tb_logdir', 2000) ````
```

An example for profiling multiple workers:

```
```python
```

E.g. your worker IP addresses are 10.0.0.2, 10.0.0.3, 10.0.0.4, and you

would like to profile for a duration of 2 seconds.

```
tf.profiler.experimental.client.trace( 'grpc://10.0.0.2:8466,grpc://10.0.0.3:8466,grpc://10.0.0.4:8466', 'gs://your_tb_logdir', 2000) ````
```

Profile Service URL or TPU name \*  
10.138.0.40:6009,10.138.0.23:6009

Address Type:  IP Address  TPU Name

Profiling Duration (milliseconds)  
1000

Automatically retry N times when no trace event is collected  
3

Host Trace (TraceMe) Level  
info

Device Trace Level  
enable

Python Trace Level  
disable

**CAPTURE** **CLOSE**

Use the **Capture Profile** dialog to specify:

- A comma-delimited list of profile service URLs or TPU names.
- A profiling duration.
- The level of device, host, and Python function call tracing.
- How many times you want the Profiler to retry capturing profiles if unsuccessful at first.

## Profiling custom training loops

To profile custom training loops in your TensorFlow code, instrument the training loop with the `tf.profiler.experimental.Trace` API to mark the step boundaries for the Profiler.

The `name` argument is used as a prefix for the step names, the `step_num` keyword argument is appended in the step names, and the `_r` keyword argument makes this trace event get processed as a step event by the Profiler.

As an example,

```
python
for step in range(NUM_STEPS):
    with tf.profiler.experimental.Trace('train', step_num=step, _r=1):
        train_data = next(dataset)
        train_step(train_data)
```

This will enable the Profiler's step-based performance analysis and cause the step events to show up in the trace viewer.

Make sure that you include the dataset iterator within the `tf.profiler.experimental.Trace` context for accurate analysis of the input pipeline.

The code snippet below is an anti-pattern:

Warning: This will result in inaccurate analysis of the input pipeline.

```
python
for step, train_data in enumerate(dataset):
    with tf.profiler.experimental.Trace('train', step_num=step, _r=1):
        train_step(train_data)
```

## Profiling use cases

The profiler covers a number of use cases along four different axes. Some of the combinations are currently supported and others will be added in the future. Some of the use cases are:

- *Local vs. remote profiling*: These are two common ways of setting up your profiling environment. In local profiling, the profiling API is called on the same machine your model is executing, for example, a local workstation with GPUs. In remote profiling, the profiling API is called on a different machine from where your model is executing, for example, on a Cloud TPU.
- *Profiling multiple workers*: You can profile multiple machines when using the distributed training capabilities of TensorFlow.
- *Hardware platform*: Profile CPUs, GPUs, and TPUs.

The table below provides a quick overview of the TensorFlow-supported use cases mentioned above:

Profiling API   Local   Remote   Multiple   Hardware	::::: workers : Platforms :   -----   :-----   :-----   :-----   :-----	TensorBoard Keras
Supported   Not   Not   CPU, GPU   : Callback	Supported : Supported ::   <code>tf.profiler.experimental</code>   Supported   Not   Not   CPU, GPU   : start/stop API ( <a href="https://www.tensorflow.org/api_docs/python/tf/profiler/experimental#functions_2">https://www.tensorflow.org/api_docs/python/tf/profiler/experimental#functions_2</a> )	Supported : Supported ::   <code>tf.profiler.experimental</code>   Supported
Supported   Supported   CPU, GPU,   : client.trace API ( <a href="https://www.tensorflow.org/api_docs/python/tf/profiler/experimental/client/trace">https://www.tensorflow.org/api_docs/python/tf/profiler/experimental/client/trace</a> )	::::: TPU :   Context manager API	Supported
Supported   Not   Not   CPU, GPU   :::: supported : Supported ::		

## Best practices for optimal model performance

Use the following recommendations as applicable for your TensorFlow models to achieve optimal performance.

In general, perform all transformations on the device and ensure that you use the latest compatible version of libraries like cuDNN and Intel MKL for your platform.

## Optimize the input data pipeline

Use the data from the `[#input_pipeline_analyzer]` to optimize your data input pipeline. An efficient data input pipeline can drastically improve the speed of your model execution by reducing device idle time. Try to incorporate the best practices detailed in the [Better performance with the `tf.data` API](https://www.tensorflow.org/guide/data_performance) ([https://www.tensorflow.org/guide/data\\_performance](https://www.tensorflow.org/guide/data_performance)) guide and below to make your data input pipeline more efficient.

- In general, parallelizing any ops that do not need to be executed sequentially can significantly optimize the data input pipeline.

- In many cases, it helps to change the order of some calls or to tune the arguments such that it works best for your model. While optimizing the input data pipeline, benchmark only the data loader without the training and backpropagation steps to quantify the effect of the optimizations independently.
- Try running your model with synthetic data to check if the input pipeline is a performance bottleneck.
- Use `tf.data.Dataset.shard` for multi-GPU training. Ensure you shard very early on in the input loop to prevent reductions in throughput. When working with TFRecords, ensure you shard the list of TFRecords and not the contents of the TFRecords.
- Parallelize several ops by dynamically setting the value of `num_parallel_calls` using `tf.data.AUTOTUNE`.
- Consider limiting the usage of `tf.data.Dataset.from_generator` as it is slower compared to pure TensorFlow ops.
- Consider limiting the usage of `tf.py_function` as it cannot be serialized and is not supported to run in distributed TensorFlow.
- Use `tf.data.Options` to control static optimizations to the input pipeline.

Also read the `tf.data` performance analysis guide ([https://www.tensorflow.org/guide/data\\_performance\\_analysis](https://www.tensorflow.org/guide/data_performance_analysis)) for more guidance on optimizing your input pipeline.

## Optimize data augmentation

When working with image data, make your `data augmentation` ([https://www.tensorflow.org/tutorials/images/data\\_augmentation](https://www.tensorflow.org/tutorials/images/data_augmentation)) more efficient by casting to different data types **after** applying spatial transformations, such as flipping, cropping, rotating, etc.

Note: Some ops like `tf.image.resize` transparently change the `dtype` to `fp32`. Make sure you normalize your data to lie between `0` and `1` if its not done automatically. Skipping this step could lead to `Nan` errors if you have enabled `AMP` (<https://developer.nvidia.com/automatic-mixed-precision>).

## Use NVIDIA® DALI

In some instances, such as when you have a system with a high GPU to CPU ratio, all of the above optimizations may not be enough to eliminate bottlenecks in the data loader caused due to limitations of CPU cycles.

If you are using NVIDIA® GPUs for computer vision and audio deep learning applications, consider using the Data Loading Library (`DALI` (<https://docs.nvidia.com/deeplearning/dali/user-guide/docs/examples/getting%20started.html>)) to accelerate the data pipeline.

Check the `NVIDIA® DALI: Operations` ([https://docs.nvidia.com/deeplearning/dali/user-guide/docs/supported\\_ops.html](https://docs.nvidia.com/deeplearning/dali/user-guide/docs/supported_ops.html)) documentation for a list of supported DALI ops.

## Use threading and parallel execution

Run ops on multiple CPU threads with the `tf.config.threading` API to execute them faster.

TensorFlow automatically sets the number of parallelism threads by default. The thread pool available for running TensorFlow ops depends on the number of CPU threads available.

Control the maximum parallel speedup for a single op by using `tf.config.threading.set_intra_op_parallelism_threads`. Note that if you run multiple ops in parallel, they will all share the available thread pool.

If you have independent non-blocking ops (ops with no directed path between them on the graph), use `tf.config.threading.set_inter_op_parallelism_threads` to run them concurrently using the available thread pool.

## Miscellaneous

When working with smaller models on NVIDIA® GPUs, you can set `tf.compat.v1.ConfigProto.force_gpu_compatible=True` to force all CPU tensors to be allocated with CUDA pinned memory to give a significant boost to model performance. However, exercise caution while using this option for unknown/very large models as this might negatively impact the host (CPU) performance.

## Improve device performance

Follow the best practices detailed here and in the `GPU performance optimization guide` ([https://www.tensorflow.org/guide/gpu\\_performance\\_analysis](https://www.tensorflow.org/guide/gpu_performance_analysis)) to optimize on-device TensorFlow model performance.

If you are using NVIDIA GPUs, log the GPU and memory utilization to a CSV file by running:

```
shell
nvidia-smi
--query-gpu=utilization.gpu,utilization.memory,memory.total,
memory.free,memory.used --format=csv
```

## Configure data layout

When working with data that contains channel information (like images), optimize the data layout format to prefer channels last (NHWC over NCHW).

Channel-last data formats improve `Tensor Core` (<https://www.nvidia.com/en-gb/data-center/tensor-cores/>) utilization and provide significant performance improvements especially in convolutional models when coupled with AMP. NCHW data layouts can still be operated on by Tensor Cores, but introduce additional overhead due to automatic transpose ops.

You can optimize the data layout to prefer NHWC layouts by setting `data_format="channels_last"` for layers such as `tf.keras.layers.Conv2D`, `tf.keras.layers.Conv3D`, and `tf.keras.layers.RandomRotation`.

Use `tf.keras.backend.set_image_data_format` to set the default data layout format for the Keras backend API.

## Max out the L2 cache

When working with NVIDIA® GPUs, execute the code snippet below before the training loop to max out the L2 fetch granularity to 128 bytes.

```
'''python
import ctypes
_libcudart = ctypes.CDLL('libcudart.so')'''
```

## Set device limit on the current device

### `cudaLimitMaxL2FetchGranularity = 0x05`

```
pValue = ctypes.cast((ctypes.c_int*1)(), ctypes.POINTER(ctypes.c_int)) _libcudart.cudaDeviceSetLimit(ctypes.c_int(0x05), ctypes.c_int(128))
_libcudart.cudaDeviceGetLimit(pValue, ctypes.c_int(0x05)) assert pValue.contents.value == 128'''
```

## Configure GPU thread usage

The GPU thread mode decides how GPU threads are used.

Set the thread mode to `gpu_private` to make sure that preprocessing does not steal all the GPU threads. This will reduce the kernel launch delay during training. You can also set the number of threads per GPU. Set these values using environment variables.

```
'''python
import os
os.environ['TF_GPU_THREAD_MODE']='gpu_private' os.environ['TF_GPU_THREAD_COUNT']='1'''
```

## Configure GPU memory options

In general, increase the batch size and scale the model to better utilize GPUs and get higher throughput. Note that increasing the batch size will change the model's accuracy so the model needs to be scaled by tuning hyperparameters like the learning rate to meet the target accuracy.

Also, use `tf.config.experimental.set_memory_growth` to allow GPU memory to grow to prevent all the available memory from being fully allocated to ops that require only a fraction of the memory. This allows other processes which consume GPU memory to run on the same device.

To learn more, check out the [Limiting GPU memory growth](https://www.tensorflow.org/guide/gpu#limiting_gpu_memory_growth) ([https://www.tensorflow.org/guide/gpu#limiting\\_gpu\\_memory\\_growth](https://www.tensorflow.org/guide/gpu#limiting_gpu_memory_growth)) guidance in the GPU guide to learn more.

## Miscellaneous

- Increase the training mini-batch size (number of training samples used per device in one iteration of the training loop) to the maximum amount that fits without an out of memory (OOM) error on the GPU. Increasing the batch size impacts the model's accuracy—so make sure you scale the model by tuning hyperparameters to meet the target accuracy.
- Disable reporting OOM errors during tensor allocation in production code. Set `report_tensor_allocations_upon_oom=False` in `tf.compat.v1.RunOptions`.
- For models with convolution layers, remove bias addition if using batch normalization. Batch normalization shifts values by their mean and this removes the need to have a constant bias term.
- Use TF Stats to find out how efficiently on-device ops run.
- Use `tf.function` to perform computations and optionally, enable the `jit_compile=True` flag (`tf.function(jit_compile=True)`). To learn more, go to [Use XLA tf.function](#) ([https://www.tensorflow.org/xla/tutorials/jit\\_compile](https://www.tensorflow.org/xla/tutorials/jit_compile)).
- Minimize host Python operations between steps and reduce callbacks. Calculate metrics every few steps instead of at every step.
- Keep the device compute units busy.
- Send data to multiple devices in parallel.
- Consider [using 16-bit numerical representations](#) ([https://www.tensorflow.org/guide/mixed\\_precision](https://www.tensorflow.org/guide/mixed_precision)), such as `fp16` —the half-precision floating point format specified by IEEE—or the Brain floating-point `bfloat16` (<https://cloud.google.com/tpu/docs/bfloat16>) format.

## Additional resources

- The [TensorFlow Profiler: Profile model performance](#) ([https://www.tensorflow.org/tensorboard/tensorboard\\_profiling\\_keras](https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras)) tutorial with Keras and TensorBoard where you can apply the advice in this guide.
- The [Performance profiling in TensorFlow 2](#) (<https://www.youtube.com/watch?v=pXHAQlhMhl>) talk from the TensorFlow Dev Summit 2020.
- The [TensorFlow Profiler demo](#) ([https://www.youtube.com/watch?v=e4\\_4D7uNvf8](https://www.youtube.com/watch?v=e4_4D7uNvf8)) from the TensorFlow Dev Summit 2020.

## Known limitations

### Profiling multiple GPUs on TensorFlow 2.2 and TensorFlow 2.3

TensorFlow 2.2 and 2.3 support multiple GPU profiling for single host systems only; multiple GPU profiling for multi-host systems is not supported. To profile multi-worker GPU configurations, each worker has to be profiled independently. From TensorFlow 2.4 multiple workers can be profiled using the `tf.profiler.experimental.client.trace` API.

CUDA® Toolkit 10.2 or later is required to profile multiple GPUs. As TensorFlow 2.2 and 2.3 support CUDA® Toolkit versions only up to 10.1, you need to create symbolic links to `libcudart.so.10.1` and `libcupti.so.10.1`:

```
shell
sudo ln -s /usr/local/cuda/lib64/libcudart.so.10.2 /usr/local/cuda/lib64/libcudart.so.10.1
sudo ln -s /usr/local/cuda/extras/CUPTI/lib64/libcupti.so.10.2 /usr/local/cuda/extras/CUPTI/lib64/libcupti.so.10.1
```

## Create an op

Note: To guarantee that your C++ custom ops are ABI compatible with TensorFlow's official pip packages, please follow the guide at [Custom op repository](#) (<https://github.com/tensorflow/custom-op>). It has an end-to-end code example, as well as Docker images for building and distributing your custom ops.

If you'd like to create an op that isn't covered by the existing TensorFlow library, we recommend that you first try writing the op in Python as a composition of existing Python ops or functions. If that isn't possible, you can create a custom C++ op. There are several reasons why you might want to create a custom C++ op:

- It's not easy or possible to express your operation as a composition of existing ops.
- It's not efficient to express your operation as a composition of existing primitives.
- You want to hand-fuse a composition of primitives that a future compiler would find difficult fusing.

For example, imagine you want to implement something like "median pooling", similar to the "MaxPool" operator, but computing medians over sliding windows instead of maximum values. Doing this using a composition of operations may be possible (e.g., using `ExtractImagePatches` and `TopK`), but may not be as performance- or memory-efficient as a native operation where you can do something more clever in a single, fused operation. As always, it is typically first worth trying to express what you want using operator composition, only choosing to add a new operation if that proves to be difficult or inefficient.

To incorporate your custom op you'll need to:

1. Register the new op in a C++ file. Op registration defines an interface (specification) for the op's functionality, which is independent of the op's implementation. For example, op registration defines the op's name and the op's inputs and outputs. It also defines the shape function that is used for tensor shape inference.
2. Implement the op in C++. The implementation of an op is known as a kernel, and it is the concrete implementation of the specification you registered in Step 1. There can be multiple kernels for different input / output types or architectures (for example, CPUs, GPUs).
3. Create a Python wrapper (optional). This wrapper is the public API that's used to create the op in Python. A default wrapper is generated from the op registration, which can be used directly or added to.
4. Write a function to compute gradients for the op (optional).
5. Test the op. We usually do this in Python for convenience, but you can also test the op in C++. If you define gradients, you can verify them with the Python `tf.test.compute_gradient_error`. See `relu_op_test.py` ([https://www.tensorflow.org/code/tensorflow/python/kernel\\_tests/relu\\_op\\_test.py](https://www.tensorflow.org/code/tensorflow/python/kernel_tests/relu_op_test.py)) as an example that tests the forward functions of Relu-like operators and their gradients.

## Prerequisites

- Some familiarity with C++.
- Must have installed the [TensorFlow binary](#) (<https://www.tensorflow.org/install>), or must have [downloaded TensorFlow source](#) (<https://www.tensorflow.org/install/source.md>), and be able to build it.

## Define the op interface

You define the interface of an op by registering it with the TensorFlow system. In the registration, you specify the name of your op, its inputs (types and names) and outputs (types and names), as well as docstrings and any `attrs` the op might require. To see how this works, suppose you'd like to create an op that takes a tensor of `int32`s and outputs a copy of the tensor, with all but the first element set to zero. To do this, create a file named `zero_out.cc`. Then add a call to the `REGISTER_OP` macro that defines the interface for your op:

```c++

```
include "tensorflow/core/framework/op.h"
include "tensorflow/core/framework/shape_inference.h"
```

using namespace tensorflow;

```
REGISTER_OP("ZeroOut") .Input("to_zero: int32") .Output("zeroed: int32") .SetShapeFn( (:tensorflow::shape_inference::InferenceContext* c) { c->set_output(0, c->input(0)); return Status::OK(); }); ````
```

This `ZeroOut` op takes one tensor `to_zero` of 32-bit integers as input, and outputs a tensor `zeroed` of 32-bit integers. The op also uses a shape function to ensure that the output tensor is the same shape as the input tensor. For example, if the input is a tensor of shape [10, 20], then this shape function specifies that the output shape is also [10, 20].

Note: The op name must be in CamelCase and it must be unique among all other ops that are registered in the binary.

## Implement the kernel for the op

After you define the interface, provide one or more implementations of the op. To create one of these kernels, create a class that extends `OpKernel` and overrides the `Compute` method. The `Compute` method provides one `context` argument of type `OpKernelContext*`, from which you can access useful things like the input and output tensors.

Add your kernel to the file you created above. The kernel might look something like this:

```c++

```
include "tensorflow/core/framework/op_kernel.h"
```

using namespace tensorflow;

```
class ZeroOutOp : public OpKernel { public: explicit ZeroOutOp(OpKernelConstruction* context) : OpKernel(context) {}
void Compute(OpKernelContext* context) override { // Grab the input tensor const Tensor& input_tensor = context->input(0); auto input = input_tensor.flat();
```

```
// Create an output tensor
Tensor* output_tensor = NULL;
OP_REQUIRES_OK(context, context->allocate_output(0, input_tensor.shape(),
&output_tensor));
auto output_flat = output_tensor->flat<int32>();

// Set all but the first element of the output tensor to 0.
const int N = input.size();
for (int i = 1; i < N; i++) {
    output_flat(i) = 0;
}

// Preserve the first input value if possible.
if (N > 0) output_flat(0) = input(0);
```

}; ````

After implementing your kernel, you register it with the TensorFlow system. In the registration, you specify different constraints under which this kernel will run. For example, you might have one kernel made for CPUs, and a separate one for GPUs.

To do this for the `ZeroOut` op, add the following to `zero_out.cc`:

```
c++
REGISTER_KERNEL_BUILDER(Name("ZeroOut").Device(DEVICE_CPU), ZeroOutOp);
```

Important: Instances of your `OpKernel` may be accessed concurrently. Your `Compute` method must be thread-safe. Guard any access to class members with a mutex. Or better yet, don't share state via class members! Consider using a `ResourceMgr` ([https://www.tensorflow.org/code/tensorflow/core/framework/resource\\_mgr.h](https://www.tensorflow.org/code/tensorflow/core/framework/resource_mgr.h)) to keep track of op state.

## Multi-threaded CPU kernels

To write a multi-threaded CPU kernel, the `Shard` function in `work_sharder.h` ([https://www.tensorflow.org/code/tensorflow/core/util/work\\_sharder.h](https://www.tensorflow.org/code/tensorflow/core/util/work_sharder.h)) can be used. This function shards a computation function across the threads configured to be used for intra-op threading (see `intra_op_parallelism_threads` in `config.proto` (<https://www.tensorflow.org/code/tensorflow/core/protobuf/config.proto>)).

## GPU kernels

A GPU kernel is implemented in two parts: the `OpKernel` and the CUDA kernel and its launch code.

Sometimes the `OpKernel` implementation is common between a CPU and GPU kernel, such as around inspecting inputs and allocating outputs. In that case, a suggested implementation is to:

1. Define the `OpKernel` templated on the Device and the primitive type of the tensor.
2. To do the actual computation of the output, the `Compute` function calls a templated functor struct.
3. The specialization of that functor for the `CPUDevice` is defined in the same file, but the specialization for the `GPUDevice` is defined in a `.cu.cc` file, since it will be compiled with the CUDA compiler.

Here is an example implementation.

```c++ // kernel\_example.h

```
ifndef KERNEL_EXAMPLE_H_
```

```
define KERNEL_EXAMPLE_H_
```

```
include
```

```
template struct ExampleFunctor { void operator()(const Device& d, int size, const T in, T out); };
```

```
if GOOGLE_CUDA
```

```
// Partially specialize functor for GpuDevice. template struct ExampleFunctor { void operator()(const Eigen::GpuDevice& d, int size, const T in, T out); };
```

```
endif
```

```
endif KERNEL_EXAMPLE_H_
```

```
...
```

```
```c++ // kernel_example.cc
```

```
include "kernel_example.h"
```

```
include "tensorflow/core/framework/op.h"
```

```
include "tensorflow/core/framework/shape_inference.h"
```

```
include "tensorflow/core/framework/op_kernel.h"
```

```
using namespace tensorflow;
```

```
using CPUDevice = Eigen::ThreadPoolDevice; using GPUDevice = Eigen::GpuDevice;
```

```
REGISTER_OP("Example") .Attr("T: numbertype") .Input("input: T") .Output("input_times_two: T") .SetShapeFn( (:tensorflow::shape_inference::InferenceContext* c) { c->set_output(0, c->input(0)); return Status::OK(); });
```

```
// CPU specialization of actual computation. template struct ExampleFunctor { void operator()(const CPUDevice& d, int size, const T in, T out) { for (int i = 0; i < size; ++i) { out[i] = 2 * in[i]; } } };
```

```
// OpKernel definition. // template parameter is the datatype of the tensors. template class ExampleOp : public OpKernel { public: explicit ExampleOp(OpKernelConstruction* context) : OpKernel(context) {}
```

```
void Compute(OpKernelContext* context) override { // Grab the input tensor const Tensor& input_tensor = context->input(0);
```

```
// Create an output tensor
```

```
Tensor* output_tensor = NULL;  
OP_REQUIRES_OK(context, context->allocate_output(0, input_tensor.shape(),  
&output_tensor));
```

```
// Do the computation.
```

```
OP_REQUIRES(context, input_tensor.NumElements() <= tensorflow::kint32max,  
errors::InvalidArgument("Too many elements in tensor"));
```

```
ExampleFunctor<Device, T>()(  
context->eigen_device<Device>(),  
static_cast<int>(input_tensor.NumElements()),  
input_tensor.flat<T>().data(),  
output_tensor->flat<T>().data());
```

```
};
```

```
// Register the CPU kernels.
```

```
define REGISTER_CPU(T) \
```

```
REGISTER_KERNEL_BUILDER( \ Name("Example").Device(DEVICE_CPU).TypeConstraint("T"), \ ExampleOp); REGISTER_CPU(float); REGISTER_CPU(int32);  
// Register the GPU kernels.
```

```
ifdef GOOGLE_CUDA
```

```
define REGISTER_GPU(T) \
```

```
/ Declare explicit instantiations in kernel_example.cu.cc. / \ extern template class ExampleFunctor; \ REGISTER_KERNEL_BUILDER( \  
Name("Example").Device(DEVICE_GPU).TypeConstraint("T"), \ ExampleOp); REGISTER_GPU(float); REGISTER_GPU(int32);
```

```
endif // GOOGLE_CUDA
```

```
...
```

```
```c++ // kernel_example.cu.cc
```

```
ifdef GOOGLE_CUDA
define EIGEN_USE_GPU
include "kernel_example.h"
include "tensorflow/core/util/gpu_kernel_helper.h"
```

```
using namespace tensorflow;
using GPUDevice = Eigen::GpuDevice;
// Define the CUDA kernel. template global void ExampleCudaKernel(const int size, const T in, T out) { for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < size; i += blockDim.x * gridDim.x) { out[i] = 2 * __ldg(in + i); } }
// Define the GPU implementation that launches the CUDA kernel. template void ExampleFunctor::operator()( const GPUDevice& d, int size, const T in, T out) { // Launch the cuda kernel. // // See core/util/gpu_kernel_helper.h for example of computing // block count and thread_per_block count. int block_count = 1024; int thread_per_block = 20; ExampleCudaKernel <>(size, in, out); }
// Explicitly instantiate functors for the types of OpKernels registered. template struct ExampleFunctor; template struct ExampleFunctor;
```

```
endif // GOOGLE_CUDA
```

```

## Build the op library

### Compile the op using your system compiler (TensorFlow binary installation)

You should be able to compile `zero_out.cc` with a C++ compiler such as `g++` or `clang` available on your system. The binary PIP package installs the header files and the library that you need to compile your op in locations that are system specific. However, the TensorFlow python library provides the `get_include` function to get the header directory, and the `get_lib` directory has a shared object to link against. Here are the outputs of these functions on an Ubuntu machine.

```
```bash $ python
```

```
import tensorflow as tf
tf.sysconfig.get_include() '/usr/local/lib/python3.6/site-packages/tensorflow/include'
tf.sysconfig.get_lib() '/usr/local/lib/python3.6/site-packages/tensorflow' ```

```

```

Assuming you have `g++` installed, here is the sequence of commands you can use to compile your op into a dynamic library.

```
bash
TF_CFLAGS=( $(python -c 'import tensorflow as tf; print(" ".join(tf.sysconfig.get_compile_flags()))') )
TF_LFLAGS=( $(python -c 'import tensorflow as tf; print(" ".join(tf.sysconfig.get_link_flags()))') )
g++ -std=c++14 -shared zero_out.cc -o zero_out.so -fPIC ${TF_CFLAGS[@]} ${TF_LFLAGS[@]} -O2
On macOS, the additional flag "-undefined dynamic_lookup" is required when building the .so file.
```

Note on `gcc` version  $\geq 5$ : `gcc` uses the new C++ ABI (<https://gcc.gnu.org/gcc-5/changes.html#libstdcxx>) since version 5. The binary pip packages available on the TensorFlow website are built with `gcc4` that uses the older ABI. If you compile your op library with `gcc\geq 5`, add `-D_GLIBCXX_USE_CXX11_ABI=0` to the command line to make the library compatible with the older abi.

### Compile the op using bazel (TensorFlow source installation)

If you have TensorFlow sources installed, you can make use of TensorFlow's build system to compile your op. Place a BUILD file with following Bazel build rule in the `tensorflow/core/user_ops` ([https://www.tensorflow.org/code/tensorflow/core/user\\_ops/](https://www.tensorflow.org/code/tensorflow/core/user_ops/)) directory.

```
```python
load("//tensorflow:tensorflow.bzl", "tf_custom_op_library")
tf_custom_op_library(name = "zero_out.so", srcs = ["zero_out.cc"], ) ```

Run the following command to build zero_out.so.
```

```
bash
$ bazel build --config opt //tensorflow/core/user_ops:zero_out.so
```

For compiling the `Example` operation, with the CUDA Kernel, you need to use the `gpu_srcs` parameter of `tf_custom_op_library`. Place a BUILD file with the following Bazel build rule in a new folder inside the `tensorflow/core/user_ops` ([https://www.tensorflow.org/code/tensorflow/core/user\\_ops/](https://www.tensorflow.org/code/tensorflow/core/user_ops/)) directory (e.g. `"example_gpu"`).

```
```python
load("//tensorflow:tensorflow.bzl", "tf_custom_op_library")
tf_custom_op_library( # kernel_example.cc kernel_example.cu.cc kernel_example.h
    name = "kernel_example.so",
    srcs = ["kernel_example.h", "kernel_example.cc"],
    gpu_srcs = ["kernel_example.cu.cc", "kernel_example.h"], ) ```

Run the following command to build kernel_example.so.
```

```
bash
$ bazel build --config opt //tensorflow/core/user_ops/example_gpu:kernel_example.so
```

Note: As explained above, if you are compiling with `gcc\geq 5` add `--cxxopt=-D_GLIBCXX_USE_CXX11_ABI=0` to the Bazel command line arguments.

Note: Although you can create a shared library (a `.so` file) with the standard `cc_library` rule, we strongly recommend that you use the `tf_custom_op_library` macro. It adds some required dependencies, and performs checks to ensure that the shared library is compatible with TensorFlow's plugin loading mechanism.

# Use the op in Python

TensorFlow Python API provides the `tf.load_op_library` function to load the dynamic library and register the op with the TensorFlow framework. `load_op_library` returns a Python module that contains the Python wrappers for the op and the kernel. Thus, once you have built the op, you can do the following to run it from Python:

```
```python
import tensorflow as tf
zero_out_module = tf.load_op_library('./zero_out.so')
print(zero_out_module.zero_out([[1, 2], [3, 4]]).numpy())
```

```

## Prints

```
array([[1, 0], [0, 0]], dtype=int32)
```

Keep in mind, the generated function will be given a snake\_case name (to comply with [PEP8](https://www.python.org/dev/peps/pep-0008/) (<https://www.python.org/dev/peps/pep-0008/>)). So, if your op is named `ZeroOut` in the C++ files, the python function will be called `zero_out`.

To make the op available as a regular function `import`-able from a Python module, it maybe useful to have the `load_op_library` call in a Python source file as follows:

```
```python
import tensorflow as tf
zero_out_module = tf.load_op_library('./zero_out.so')
zero_out = zero_out_module.zero_out
```

```

## Verify that the op works

A good way to verify that you've successfully implemented your op is to write a test for it. Create the file `zero_out_op_test.py` with the contents:

```
```python
import tensorflow as tf
class ZeroOutTest(tf.test.TestCase):
    def testZeroOut(self):
        zero_out_module = tf.load_op_library('./zero_out.so')
        with self.test_session():
            result = zero_out_module.zero_out([5, 4, 3, 2, 1])
            self.assertAllEqual(result.eval(), [5, 0, 0, 0, 0])
if __name__ == "__main__":
    tf.test.main()
```

```

Then run your test (assuming you have tensorflow installed):

```
sh
$ python zero_out_op_test.py
```

## Build advanced features into your op

Now that you know how to build a basic (and somewhat restricted) op and implementation, we'll look at some of the more complicated things you will typically need to build into your op. This includes:

- [Conditional checks and validation](#)
- [Op registration](#)
  - [Attrs](#)
  - [Attr types](#)
  - [Polymorphism](#)
  - [Inputs and outputs](#)
  - [Backwards compatibility](#)
- [GPU support](#)
  - [Compiling the kernel for the GPU device](#)
- [Implement the gradient in Python](#)
- [Shape functions in C++](#)

## Conditional checks and validation

The example above assumed that the op applied to a tensor of any shape. What if it only applied to vectors? That means adding a check to the above `OpKernel` implementation.

```
```c++
void Compute(OpKernelContext* context) override { // Grab the input tensor
    const Tensor& input_tensor = context->input(0);
    OP_REQUIRES(context, TensorShapeUtils::IsVector(input_tensor.shape()),
                errors::InvalidArgument("ZeroOut expects a 1-D vector."));
    // ...
}
```

```

}

This asserts that the input is a vector, and returns having set the `InvalidArgument` status if it isn't. The `OP_REQUIRES` macro (<https://www.tensorflow.org/code/tensorflow/core/platform/errors.h>) takes three arguments:

- The `context`, which can either be an `OpKernelContext` or `OpKernelConstruction` pointer (see `tensorflow/core/framework/op_kernel.h` ([https://www.tensorflow.org/code/tensorflow/core/framework/op\\_kernel.h](https://www.tensorflow.org/code/tensorflow/core/framework/op_kernel.h))), for its `SetStatus()` method.
- The condition. For example, there are functions for validating the shape of a tensor in `tensorflow/core/framework/tensor_shape.h` ([https://www.tensorflow.org/code/tensorflow/core/framework/tensor\\_shape.h](https://www.tensorflow.org/code/tensorflow/core/framework/tensor_shape.h))
- The error itself, which is represented by a `Status` object, see `tensorflow/core/platform/status.h` (<https://www.tensorflow.org/code/tensorflow/core/platform/status.h>). A `Status` has both a type (frequently `InvalidArgument`, but see the list of types) and a message. Functions for constructing an error may be found in `tensorflow/core/platform/errors.h` (<https://www.tensorflow.org/code/tensorflow/core/platform/errors.h>).

Alternatively, if you want to test whether a `Status` object returned from some function is an error, and if so return it, use `OP_REQUIRES_OK` (<https://www.tensorflow.org/code/tensorflow/core/platform/errors.h>). Both of these macros return from the function on error.

## Op registration

### Attrs

Ops can have attrs, whose values are set when the op is added to a graph. These are used to configure the op, and their values can be accessed both within the kernel implementation and in the types of inputs and outputs in the op registration. Prefer using an input instead of an attr when possible, since inputs are more flexible. This is because attrs are constants and must be defined at graph construction time. In contrast, inputs are Tensors whose values can be dynamic; that is, inputs can change every step, be set using a feed, etc. Attrs are used for things that can't be done with inputs: any configuration that affects the signature (number or type of inputs or outputs) or that can't change from step-to-step.

You define an attr when you register the op, by specifying its name and type using the `Attr` method, which expects a spec of the form:

```
<name>: <attr-type-expr>
```

where `<name>` begins with a letter and can be composed of alphanumeric characters and underscores, and `<attr-type-expr>` is a type expression of the form described below.

For example, if you'd like the `ZeroOut` op to preserve a user-specified index, instead of only the 0th element, you can register the op like so:

```
c++  
REGISTER_OP("ZeroOut")  
    .Attr("preserve_index: int")  
    .Input("to_zero: int32")  
    .Output("zeroed: int32");
```

(Note that the set of attribute types is different from the `tf.DType` used for inputs and outputs.)

Your kernel can then access this attr in its constructor via the `context` parameter:

```
c++  
class ZeroOutOp : public OpKernel {  
public:  
    explicit ZeroOutOp(OpKernelConstruction* context) : OpKernel(context) {  
        // Get the index of the value to preserve  
        OP_REQUIRES_OK(context,  
            context->GetAttr("preserve_index", &preserve_index_));  
        // Check that preserve_index is positive  
        OP_REQUIRES(context, preserve_index_ >= 0,  
            errors::InvalidArgument("Need preserve_index >= 0, got ",  
            preserve_index_));  
    }  
    void Compute(OpKernelContext* context) override {  
        // ...  
    }  
private:  
    int preserve_index_;  
};
```

which can then be used in the `Compute` method:

```
```c++ void Compute(OpKernelContext* context) override { // ...  
  
    // We're using saved attr to validate potentially dynamic input  
    // So we check that preserve_index is in range  
    OP_REQUIRES(context, preserve_index_ < input.dimension(0),  
        errors::InvalidArgument("preserve_index out of range"));  
  
    // Set all the elements of the output tensor to 0  
    const int N = input.size();  
    for (int i = 0; i < N; i++) {  
        output_flat(i) = 0;  
    }  
  
    // Preserve the requested input value  
    output_flat(preserve_index_) = input(preserve_index_);  
}
```

## Attr types

The following types are supported in an attr:

- `string` : Any sequence of bytes (not required to be UTF8).
- `int` : A signed integer.
- `float` : A floating point number.
- `bool` : True or false.
- `type` : One of the (non-ref) values of `DataType` (<https://www.tensorflow.org/code/tensorflow/core/framework/types.cc>).
- `shape` : A `TensorShapeProto` ([https://www.tensorflow.org/code/tensorflow/core/framework/tensor\\_shape.proto](https://www.tensorflow.org/code/tensorflow/core/framework/tensor_shape.proto)).
- `list(<type>)` : A list of `<type>`, where `<type>` is one of the above types. Note that `list(list(<type>))` is invalid.

See also: `op_def_builder.cc:FinalizeAttr` ([https://www.tensorflow.org/code/tensorflow/core/framework/op\\_def\\_builder.cc](https://www.tensorflow.org/code/tensorflow/core/framework/op_def_builder.cc)) for a definitive list.

## Default values and constraints

Attrs may have default values, and some types of attrs can have constraints. To define an attr with constraints, you can use the following `<attr-type-expr>`s:

`{'<string1>', '<string2>'}` : The value must be a string that has either the value `<string1>` or `<string2>`. The name of the type, `string`, is implied when you use this syntax. This emulates an enum:

```
c++  
REGISTER_OP("EnumExample")  
    .Attr("e: {'apple', 'orange'}");  
{<type1>, <type2>} : The value is of type type, and must be one of <type1> or <type2>, where <type1> and <type2> are supported tf.DType. You don't specify that the type of the attr is type. This is implied when you have a list of types in {...}. For example, in this case the attr t is a type that must be an int32, a float, or a bool:  
c++  
REGISTER_OP("RestrictedTypeExample")  
    .Attr("t: {int32, float, bool}");
```

There are shortcuts for common type constraints:

- `numbertype` : Type `type` restricted to the numeric (non-string and non-bool) types.
- `realnumbertype` : Like `numbertype` without complex types.
- `quantizedtype` : Like `numbertype` but just the quantized number types.

The specific lists of types allowed by these are defined by the functions (like `NumberTypes()`) in `tensorflow/core/framework/types.h` (<https://www.tensorflow.org/code/tensorflow/core/framework/types.h>). In this example the attr `t` must be one of the numeric types:

```
c++  
REGISTER_OP("NumberType")  
    .Attr("t: numbertype");
```

For this op:

```
python
tf.number_type(t=tf.int32) # Valid
tf.number_type(t=tf.bool) # Invalid
```

Lists can be combined with other lists and single types. The following op allows attr `t` to be any of the numeric types, or the bool type:

```
c++
REGISTER_OP("NumberOrBooleanType")
    .Attr("t: {numbertype, bool}");
```

For this op:

```
python
tf.number_or_boolean_type(t=tf.int32) # Valid
tf.number_or_boolean_type(t=tf.bool) # Valid
tf.number_or_boolean_type(t=tf.string) # Invalid
```

`int >= <n>`: The value must be an int whose value is greater than or equal to `<n>`, where `<n>` is a natural number. For example, the following op registration specifies that the attr `a` must have a value that is at least 2 :

```
c++
REGISTER_OP("MinIntExample")
    .Attr("a: int >= 2");
list(<type>) >= <n> : A list of type <type> whose length is greater than or equal to <n>. For example, the following op registration specifies that the attr a is a list of types (either int32 or float), and that there must be at least 3 of them:
```

`c++`

```
REGISTER_OP("TypeListExample")
    .Attr("a: list({int32, float}) >= 3");
```

To set a default value for an attr (making it optional in the generated code), add `= <default>` to the end, as in:

```
c++
REGISTER_OP("AttrDefaultExample")
    .Attr("i: int = 0");
```

Additionally, both a constraint and a default value can be specified:

```
c++
REGISTER_OP("AttrConstraintAndDefaultExample")
    .Attr("i: int >= 1 = 1");
```

The supported syntax of the default value is what would be used in the proto representation of the resulting GraphDef definition.

Here are examples for how to specify a default for all types:

```
c++
REGISTER_OP("AttrDefaultExampleForAllTypes")
    .Attr("s: string = 'foo'")
    .Attr("i: int = 0")
    .Attr("f: float = 1.0")
    .Attr("b: bool = true")
    .Attr("ty: type = DT_INT32")
    .Attr("sh: shape = { dim { size: 1 } dim { size: 2 } }")
    .Attr("te: tensor = { dtype: DT_INT32 int_val: 5 }")
    .Attr("l_empty: list(int) = []")
    .Attr("l_int: list(int) = [2, 3, 5, 7]");
```

Note in particular that the values of type `type` use `tf.DType`.

## Polymorphism

### Type polymorphism

For ops that can take different types as input or produce different output types, you can specify [an attr in an input or output type](#) in the op registration. Typically you would then register an `OpKernel` for each supported type.

For instance, if you'd like the `ZeroOut` op to work on `float`s in addition to `int32`s, your op registration might look like:

```
c++
REGISTER_OP("ZeroOut")
    .Attr("T: {float, int32}")
    .Input("to_zero: T")
    .Output("zeroed: T");
```

Your op registration now specifies that the input's type must be `float`, or `int32`, and that its output will be the same type, since both have type `T`.

### Naming

Inputs, outputs, and attrs generally should be given `snake_case` names. The one exception is attrs that are used as the type of an input or in the type of an output. Those attrs can be inferred when the op is added to the graph and so don't appear in the op's function. For example, this last definition of `ZeroOut` will generate a Python function that looks like:

```
```python
def zero_out(to_zero, name=None):
    """
    Args:
        to_zero: A Tensor. Must be one of the following types: float32 , int32 .
        name: A name for the operation (optional).
    Returns:
        A Tensor. Has the same type as to_zero.
    ```
```

If `to_zero` is passed an `int32` tensor, then `T` is automatically set to `int32` (well, actually `DT_INT32`). Those inferred attrs are given Capitalized or CamelCase names.

Compare this with an op that has a type attr that determines the output type:

```
c++
REGISTER_OP("StringToNumber")
    .Input("string_tensor: string")
    .Output("output: out_type")
    .Attr("out_type: {float, int32} = DT_FLOAT");
    .Doc(R"doc(
Converts each string in the input Tensor to the specified numeric type.
)doc")
```

Converts each string in the input Tensor to the specified numeric type.

`)doc";`

In this case, the user has to specify the output type, as in the generated Python:

```
```python
def string_to_number(string_tensor, out_type=None, name=None):
    """Converts each string in the input Tensor to the specified numeric type.
    Args:
        string_tensor: A Tensor of type string .
        out_type: An optional tf.DType from: tf.float32, tf.int32 . Defaults to tf.float32 .
        name: A name for the operation (optional).
    Returns:
        A Tensor. Has the same type as string_tensor.
    """
```
```

Args: `string_tensor`: A Tensor of type `string` . `out_type`: An optional `tf.DType` from: `tf.float32`, `tf.int32` . Defaults to `tf.float32` . `name`: A name for the operation (optional).

operation (optional).

Returns: A Tensor of type out\_type . """" ""

Type polymorphism example

```c++

## include "tensorflow/core/framework/op\_kernel.h"

```
class ZeroOutInt32Op : public OpKernel { // as before };
class ZeroOutFloatOp : public OpKernel { public: explicit ZeroOutFloatOp(OpKernelConstruction* context) : OpKernel(context) {} };
void Compute(OpKernelContext* context) override { // Grab the input tensor const Tensor& input_tensor = context->input(0); auto input = input_tensor.flat();

// Create an output tensor
Tensor* output = NULL;
OP_REQUIRES_OK(context,
               context->allocate_output(0, input_tensor.shape(), &output));
auto output_flat = output->template flat<float>();

// Set all the elements of the output tensor to 0
const int N = input.size();
for (int i = 0; i < N; i++) {
  output_flat(i) = 0;
}

// Preserve the first input value
if (N > 0) output_flat(0) = input(0);

}};

// Note that TypeConstraint("T") means that attr "T" (defined // in the op registration above) must be "int32" to use this template // instantiation.
```

REGISTER\_KERNEL\_BUILDER(Name("ZeroOut").Device(DEVICE\_CPU).TypeConstraint("T"), ZeroOutInt32Op); REGISTER\_KERNEL\_BUILDER(Name("ZeroOut").Device(DEVICE\_CPU).TypeConstraint("T"), ZeroOutFloatOp); ````

To preserve [backwards compatibility](#), you should specify a [default value](#) when adding an attr to an existing op:

c++

```
REGISTER_OP("ZeroOut")
    .Attr("T: {float, int32} = DT_INT32")
    .Input("to_zero: T")
    .Output("zeroed: T")
```

Let's say you wanted to add more types, say double :

c++

```
REGISTER_OP("ZeroOut")
    .Attr("T: {float, double, int32}")
```

.Input("to\_zero: T")

.Output("zeroed: T")

Instead of writing another OpKernel with redundant code as above, often you will be able to use a C++ template instead. You will still have one kernel registration (REGISTER\_KERNEL\_BUILDER call) per overload.

```c++ template class ZeroOutOp : public OpKernel { public: explicit ZeroOutOp(OpKernelConstruction\* context) : OpKernel(context) {}

void Compute(OpKernelContext\* context) override { // Grab the input tensor const Tensor& input\_tensor = context->input(0); auto input = input\_tensor.flat();

```
// Create an output tensor
Tensor* output = NULL;
OP_REQUIRES_OK(context,
               context->allocate_output(0, input_tensor.shape(), &output));
auto output_flat = output->template flat<T>();

// Set all the elements of the output tensor to 0
const int N = input.size();
for (int i = 0; i < N; i++) {
  output_flat(i) = 0;
}

// Preserve the first input value
if (N > 0) output_flat(0) = input(0);
```

};

// Note that TypeConstraint("T") means that attr "T" (defined // in the op registration above) must be "int32" to use this template // instantiation.

REGISTER\_KERNEL\_BUILDER(Name("ZeroOut").Device(DEVICE\_CPU).TypeConstraint("T"), ZeroOutOp); REGISTER\_KERNEL\_BUILDER(Name("ZeroOut").Device(DEVICE\_CPU).TypeConstraint("T"), ZeroOutOp); REGISTER\_KERNEL\_BUILDER(Name("ZeroOut").Device(DEVICE\_CPU).TypeConstraint("T"), ZeroOutOp); ````

If you have more than a couple overloads, you can put the registration in a macro.

```c++

## include "tensorflow/core/framework/op\_kernel.h"

### define REGISTER\_KERNEL(type) \

```
REGISTER_KERNEL_BUILDER( \ Name("ZeroOut").Device(DEVICE_CPU).TypeConstraint("T"), \ ZeroOutOp)
REGISTER_KERNEL(int32); REGISTER_KERNEL(float); REGISTER_KERNEL(double);
```

# undef REGISTER\_KERNEL

...  
Depending on the list of types you are registering the kernel for, you may be able to use a macro provided by `tensorflow/core/framework/register_types.h` ([https://www.tensorflow.org/code/tensorflow/core/framework/register\\_types.h](https://www.tensorflow.org/code/tensorflow/core/framework/register_types.h)):

```c++

```
include "tensorflow/core/framework/op_kernel.h"
```

```
include "tensorflow/core/framework/register_types.h"
```

```
REGISTER_OP("ZeroOut") .Attr("T: realnumbertype") .Input("to_zero: T") .Output("zeroed: T");
```

```
template class ZeroOutOp : public OpKernel { ... };
```

```
define REGISTER_KERNEL(type) \
```

```
REGISTER_KERNEL_BUILDER( \ Name("ZeroOut").Device(DEVICE_CPU).TypeConstraint("T"), \ ZeroOutOp)
```

```
TF_CALL_REAL_NUMBER_TYPES(REGISTER_KERNEL);
```

# undef REGISTER\_KERNEL

...

List inputs and outputs

In addition to being able to accept or produce different types, ops can consume or produce a variable number of tensors.

In the next example, the attr `T` holds a *list* of types, and is used as the type of both the input `in` and the output `out`. The input and output are lists of tensors of that type (and the number and types of tensors in the output are the same as the input, since both have type `T`).

```c++

```
REGISTER_OP("PolymorphicListExample")
    .Attr("T: list(type)")
    .Input("in: T")
    .Output("out: T");
```

You can also place restrictions on what types can be specified in the list. In this next case, the input is a list of `float` and `double` tensors. The op accepts, for example, input types `(float, double, float)` and in that case the output type would also be `(float, double, float)`.

```c++

```
REGISTER_OP("ListTypeRestrictionExample")
    .Attr("T: list({float, double})")
    .Input("in: T")
    .Output("out: T");
```

If you want all the tensors in a list to be of the same type, you might do something like:

```c++

```
REGISTER_OP("IntListInputExample")
    .Attr("N: int")
    .Input("in: N * int32")
    .Output("out: int32");
```

This accepts a list of `int32` tensors, and uses an `int` attr `N` to specify the length of the list.

This can be made `type polymorphic` as well. In the next example, the input is a list of tensors (with length `"N"`) of the same (but unspecified) type (`"T"`), and the output is a single tensor of matching type:

```c++

```
REGISTER_OP("SameListInputExample")
    .Attr("N: int")
    .Attr("T: type")
    .Input("in: N * T")
    .Output("out: T");
```

By default, tensor lists have a minimum length of 1. You can change that default using a `">= "` constraint on the corresponding attr. In this next example, the input is a list of at least 2 `int32` tensors:

```c++

```
REGISTER_OP("MinLengthIntListExample")
    .Attr("N: int >= 2")
    .Input("in: N * int32")
    .Output("out: int32");
```

The same syntax works with `"list(type)"` attrs:

```c++

```
REGISTER_OP("MinimumLengthPolymorphicListExample")
    .Attr("T: list(type) >= 3")
    .Input("in: T")
    .Output("out: T");
```

## Inputs and outputs

To summarize the above, an op registration can have multiple inputs and outputs:

```c++

```
REGISTER_OP("MultipleInsAndOuts")
    .Input("y: int32")
    .Input("z: float")
    .Output("a: string")
    .Output("b: int32");
```

Each input or output spec is of the form:

`<name>: <io-type-expr>`

where `<name>` begins with a letter and can be composed of alphanumeric characters and underscores. `<io-type-expr>` is one of the following type expressions:

- `<type>`, where `<type>` is a supported input type (e.g. `float`, `int32`, `string`). This specifies a single tensor of the given type.

See `tf.DType`.

C++

```
REGISTER_OP("BuiltInTypesExample")
    .Input("integers: int32")
    .Input("complex_numbers: complex64");
• <attr-type>, where <attr-type> is the name of an Attr with type type or list(type) (with a possible type restriction). This syntax allows for polymorphic ops.
```

```
```c++ REGISTER_OP("PolymorphicSingleInput") .Attr("T: type") .Input("in: T");
```

```
REGISTER_OP("RestrictedPolymorphicSingleInput") .Attr("T: {int32, int64}") .Input("in: T");```
Referencing an attr of type list(type) allows you to accept a sequence of tensors.
```

```
```c++ REGISTER_OP("ArbitraryTensorSequenceExample") .Attr("T: list(type)") .Input("in: T") .Output("out: T");
REGISTER_OP("RestrictedTensorSequenceExample") .Attr("T: list({int32, int64})") .Input("in: T") .Output("out: T");```
Note that the number and types of tensors in the output out is the same as in the input in, since both are of type T.
```

- For a sequence of tensors with the same type: `<number> * <type>`, where `<number>` is the name of an Attr with type `int`. The `<type>` can either be a `tf.DType`, or the name of an attr with type `type`. As an example of the first, this op accepts a list of `int32` tensors:

C++

```
REGISTER_OP("Int32SequenceExample")
    .Attr("NumTensors: int")
    .Input("in: NumTensors * int32")
```

Whereas this op accepts a list of tensors of any type, as long as they are all the same:

C++

```
REGISTER_OP("SameTypeSequenceExample")
    .Attr("NumTensors: int")
    .Attr("T: type")
    .Input("in: NumTensors * T")
• For a reference to a tensor: Ref(<type>), where <type> is one of the previous types.
```

Any attr used in the type of an input will be inferred. By convention those inferred attrs use capital names (like `T` or `N`). Otherwise inputs, outputs, and attrs have names like function parameters (e.g. `num_outputs`). For more details, see the [earlier section on naming](#).

For more details, see `tensorflow/core/framework/op_def_builder.h` ([https://www.tensorflow.org/code/tensorflow/core/framework/op\\_def\\_builder.h](https://www.tensorflow.org/code/tensorflow/core/framework/op_def_builder.h)).

## Backwards compatibility

Let's assume you have written a nice, custom op and shared it with others, so you have happy customers using your operation. However, you'd like to make changes to the op in some way.

In general, changes to existing, checked-in specifications must be backwards-compatible: changing the specification of an op must not break prior serialized `GraphDef` protocol buffers constructed from older specifications. The details of `GraphDef` compatibility are [described here](#) ([./versions.md#compatibility\\_of\\_graphs\\_and\\_checkpoints](#)).

There are several ways to preserve backwards-compatibility.

1. Any new attrs added to an operation must have default values defined, and with that default value the op must have the original behavior. To change an operation from not polymorphic to polymorphic, you *must* give a default value to the new type attr to preserve the original signature by default. For example, if your operation was:

```
C++
REGISTER_OP("MyGeneralUnaryOp")
    .Input("in: float")
    .Output("out: float");
```

you can make it polymorphic in a backwards-compatible way using:

```
C++
REGISTER_OP("MyGeneralUnaryOp")
    .Input("in: T")
    .Output("out: T")
    .Attr("T: numerictype = DT_FLOAT");
```

2. You can safely make a constraint on an attr less restrictive. For example, you can change from `{int32, int64}` to `{int32, int64, float}` or `type`. Or you may change from `{"apple", "orange"}` to `{"apple", "banana", "orange"}` or `string`.
3. You can change single inputs / outputs into list inputs / outputs, as long as the default for the list type matches the old signature.
4. You can add a new list input / output, if it defaults to empty.
5. Namespace any new ops you create, by prefixing the op names with something unique to your project. This avoids having your op colliding with any ops that might be included in future versions of TensorFlow.
6. Plan ahead! Try to anticipate future uses for the op. Some signature changes can't be done in a compatible way (for example, making a list of the same type into a list of varying types).

The full list of safe and unsafe changes can be found in `tensorflow/core/framework/op_compatibility_test.cc`

([https://www.tensorflow.org/code/tensorflow/core/framework/op\\_compatibility\\_test.cc](https://www.tensorflow.org/code/tensorflow/core/framework/op_compatibility_test.cc)). If you cannot make your change to an operation backwards compatible, then create a new operation with a new name with the new semantics.

Also note that while these changes can maintain `GraphDef` compatibility, the generated Python code may change in a way that isn't compatible with old callers. The Python API may be kept compatible by careful changes in a hand-written Python wrapper, by keeping the old signature except possibly adding new optional arguments to the end. Generally incompatible changes may only be made when TensorFlow changes major versions, and must conform to the `GraphDef` [version semantics](#) ([./versions.md#compatibility\\_of\\_graphs\\_and\\_checkpoints](#)).

## GPU support

You can implement different OpKernels and register one for CPU and another for GPU, just like you can [register kernels for different types](#). There are several examples of kernels with GPU support in `tensorflow/core/kernels/` (<https://www.tensorflow.org/code/tensorflow/core/kernels/>). Notice some kernels have a CPU version in a `.cc` file, a GPU version in a file ending in `_gpu.cu.cc`, and some code shared in common in a `.h` file.

For example, the `tf.pad` has everything but the GPU kernel in `tensorflow/core/kernels/pad_op.cc`

([https://www.tensorflow.org/code/tensorflow/core/kernels/pad\\_op.cc](https://www.tensorflow.org/code/tensorflow/core/kernels/pad_op.cc)). The GPU kernel is in `tensorflow/core/kernels/pad_op_gpu.cc`

([https://www.tensorflow.org/code/tensorflow/core/kernels/pad\\_op\\_gpu.cc](https://www.tensorflow.org/code/tensorflow/core/kernels/pad_op_gpu.cc)), and the shared code is a templated class defined in

`tensorflow/core/kernels/pad_op.h` ([https://www.tensorflow.org/code/tensorflow/core/kernels/pad\\_op.h](https://www.tensorflow.org/code/tensorflow/core/kernels/pad_op.h)). We organize the code this way for two reasons: it allows you to share common code among the CPU and GPU implementations, and it puts the GPU implementation into a separate file so that it can be compiled only by the GPU compiler.

One thing to note, even when the GPU kernel version of `pad` is used, it still needs its "paddings" input in CPU memory. To mark that inputs or outputs are kept on the CPU, add a `HostMemory()` call to the kernel registration, e.g.:

```c++

## define REGISTER\_GPU\_KERNEL(T) \

```
REGISTER_KERNEL_BUILDER(Name("Pad") \ .Device(DEVICE_GPU) \ .TypeConstraint("T") \ .HostMemory("paddings"), \ PadOp) ````
```

### Compiling the kernel for the GPU device

Look at `cuda_op_kernel.cc` ([https://www.tensorflow.org/code/tensorflow/examples/adding\\_an\\_op/cuda\\_op\\_kernel.cc](https://www.tensorflow.org/code/tensorflow/examples/adding_an_op/cuda_op_kernel.cc)) for an example that uses a CUDA kernel to implement an op. The `tf_custom_op_library` accepts a `gpu_srcs` argument in which the list of source files containing the CUDA kernels (`*.cu.cc` files) can be specified. For use with a binary installation of TensorFlow, the CUDA kernels have to be compiled with NVIDIA's `nvcc` compiler. Here is the sequence of commands you can use to compile the `cuda_op_kernel.cc` ([https://www.tensorflow.org/code/tensorflow/examples/adding\\_an\\_op/cuda\\_op\\_kernel.cc](https://www.tensorflow.org/code/tensorflow/examples/adding_an_op/cuda_op_kernel.cc)) and `cuda_op_kernel.cc` ([https://www.tensorflow.org/code/tensorflow/examples/adding\\_an\\_op/cuda\\_op\\_kernel.cc](https://www.tensorflow.org/code/tensorflow/examples/adding_an_op/cuda_op_kernel.cc)) into a single dynamically loadable library:

```
```bash nvcc -std=c++14 -c -o cuda_op_kernel.cc cuda_op_kernel.cc \ ${TF_CFLAGS[@]} -D GOOGLE_CUDA=1 -x cu -Xcompiler -fPIC  
g++ -std=c++14 -shared -o cuda_op_kernel.so cuda_op_kernel.cc \ cuda_op_kernel.cc.o TF_CFLAGS[@] -fPIC -lcudart${TF_LFLAGS[@]} ````
```

`cuda_op_kernel.so` produced above can be loaded as usual in Python, using the `tf.load_op_library` function.

Note that if your CUDA libraries are not installed in `/usr/local/lib64`, you'll need to specify the path explicitly in the second (`g++`) command above. For example, add `-L /usr/local/cuda-8.0/lib64/` if your CUDA is installed in `/usr/local/cuda-8.0`.

Note: In some Linux settings, additional options to `nvcc` compiling step are needed. Add `-D_MWAITXINTRIN_H_INCLUDED` to the `nvcc` command line to avoid errors from `mwaitxintrin.h`.

## Implement the gradient in Python

Given a graph of ops, TensorFlow uses automatic differentiation (backpropagation) to add new ops representing gradients with respect to the existing ops. To make automatic differentiation work for new ops, you must register a gradient function which computes gradients with respect to the ops' inputs given gradients with respect to the ops' outputs.

Mathematically, if an op computes  $y = f(x)$  the registered gradient op converts gradients  $\partial L / \partial y$  of loss  $L$  with respect to  $y$  into gradients  $\partial L / \partial x$  with respect to  $x$  via the chain rule:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial x} = \frac{\partial L}{\partial y} \frac{\partial f}{\partial x}.$$

In the case of `ZeroOut`, only one entry in the input affects the output, so the gradient with respect to the input is a sparse "one hot" tensor. This is expressed as follows:

```
```python from tensorflow.python.framework import ops from tensorflow.python.ops import array_ops from tensorflow.python.ops import sparse_ops  
@ops.RegisterGradient("ZeroOut") def _zero_out_grad(op, grad): """The gradients for zero_out .
```

Args: op: The `zero_out` operation that we are differentiating, which we can use to find the inputs and outputs of the original op. grad: Gradient with respect to the output of the `zero_out` op.

Returns: Gradients with respect to the input of `zero_out`. `to_zero = op.inputs[0]` `shape = array_ops.shape(to_zero)` `index = array_ops.zeros_like(shape)` `first_grad = array_ops.reshape(grad, [-1])[0]` `to_zero_grad = sparse_ops.sparse_to_dense([index], shape, first_grad, 0)` `return [to_zero_grad]` # List of one Tensor, since we have one input ``

Details about registering gradient functions with `tf.RegisterGradient`:

- For an op with one output, the gradient function will take an `tf.Operation`, `op`, and a `tf.Tensor` `grad` and build new ops out of the tensors `op.inputs[i]`, `op.outputs[i]`, and `grad`. Information about any attrs can be found via `tf.Operation.get_attr`.
- If the op has multiple outputs, the gradient function will take `op` and `grads`, where `grads` is a list of gradients with respect to each output. The result of the gradient function must be a list of `Tensor` objects representing the gradients with respect to each input.
- If there is no well-defined gradient for some input, such as for integer inputs used as indices, the corresponding returned gradient should be `None`. For example, for an op taking a floating point tensor `x` and an integer index `i`, the gradient function would `return [x_grad, None]`.
- If there is no meaningful gradient for the op at all, you often will not have to register any gradient, and as long as the op's gradient is never needed, you will be fine. In some cases, an op has no well-defined gradient but can be involved in the computation of the gradient. Here you can use `ops.NotDifferentiable` to automatically propagate zeros backwards.

Note that at the time the gradient function is called, only the data flow graph of ops is available, not the tensor data itself. Thus, all computation must be performed using other tensorflow ops, to be run at graph execution time.

## Shape functions in C++

The TensorFlow API has a feature called "shape inference" that provides information about the shapes of tensors without having to execute the graph. Shape inference is supported by "shape functions" that are registered for each op type in the C++ `REGISTER_OP` declaration, and perform two roles: asserting that the shapes of the inputs are compatible during graph construction, and specifying the shapes for the outputs.

Shape functions are defined as operations on the `shape_inference::InferenceContext` class. For example, in the shape function for `ZeroOut`:

```

```
.SetShapeFn([](::tensorflow::shape_inference::InferenceContext* c) {  
    c->set_output(0, c->input(0));  
    return Status::OK();  
});  
c->set_output(0, c->input(0)); declares that the first output's shape should be set to the first input's shape. If the output is selected by its index as in the above example, the second parameter of set_output should be a ShapeHandle object. You can create an empty ShapeHandle object by its default constructor. The ShapeHandle object for an input with index idx can be obtained by c->input(idx).
```

There are a number of common shape functions that apply to many ops, such as `shape_inference::UnchangedShape` which can be found in `common_shape_fns.h` ([https://www.tensorflow.org/code/tensorflow/core/framework/common\\_shape\\_fns.h](https://www.tensorflow.org/code/tensorflow/core/framework/common_shape_fns.h)) and used as follows:

```

```
REGISTER_OP("ZeroOut")  
.Input("to_zero: int32")  
.Output("zeroed: int32")  
.SetShapeFn(::tensorflow::shape_inference::UnchangedShape);
```

A shape function can also constrain the shape of an input. For the version of `ZeroOut` with a vector shape constraint, the shape function would be as follows:

```

c++
    .SetShapeFn([](::tensorflow::shape_inference::InferenceContext* c) {
        ::tensorflow::shape_inference::ShapeHandle input;
        TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 1, &input));
        c->set_output(0, input);
        return Status::OK();
    });
The WithRank call validates that the input shape c->input(0) has a shape with exactly one dimension (or if the input shape is unknown, the output shape will be a vector with one unknown dimension).
If your op is polymorphic with multiple inputs, you can use members of InferenceContext to determine the number of shapes to check, and Merge to validate that the shapes are all compatible (alternatively, access attributes that indicate the lengths, with InferenceContext::GetAttr, which provides access to the attributes of the op).
c++
    .SetShapeFn([](::tensorflow::shape_inference::InferenceContext* c) {
        ::tensorflow::shape_inference::ShapeHandle input;
        ::tensorflow::shape_inference::ShapeHandle output;
        for (size_t i = 0; i < c->num_inputs(); ++i) {
            TF_RETURN_IF_ERROR(c->WithRank(c->input(i), 2, &input));
            TF_RETURN_IF_ERROR(c->Merge(output, input, &output));
        }
        c->set_output(0, output);
        return Status::OK();
    });

```

Since shape inference is an optional feature, and the shapes of tensors may vary dynamically, shape functions must be robust to incomplete shape information for any of the inputs. The Merge method in InferenceContext ([https://www.tensorflow.org/code/tensorflow/core/framework/shape\\_inference.h](https://www.tensorflow.org/code/tensorflow/core/framework/shape_inference.h)) allows the caller to assert that two shapes are the same, even if either or both of them do not have complete information. Shape functions are defined for all of the core TensorFlow ops and provide many different usage examples.

The InferenceContext class has a number of functions that can be used to define shape function manipulations. For example, you can validate that a particular dimension has a very specific value using InferenceContext::Dim and InferenceContext::WithValue ; you can specify that an output dimension is the sum / product of two input dimensions using InferenceContext::Add and InferenceContext::Multiply . See the InferenceContext class for all of the various shape manipulations you can specify. The following example sets shape of the first output to (n, 3), where first input has shape (n, ...)

```

c++
    .SetShapeFn([](::tensorflow::shape_inference::InferenceContext* c) {
        c->set_output(0, c->Matrix(c->Dim(c->input(0), 0), 3));
        return Status::OK();
    });

```

If you have a complicated shape function, you should consider adding a test for validating that various input shape combinations produce the expected output shape combinations. You can see examples of how to write these tests in some our [core ops tests](#) ([https://www.tensorflow.org/code/tensorflow/core/ops/array\\_ops\\_test.cc](https://www.tensorflow.org/code/tensorflow/core/ops/array_ops_test.cc)). (The syntax of INFER\_OK and INFER\_ERROR are a little cryptic, but try to be compact in representing input and output shape specifications in tests. For now, see the surrounding comments in those tests to get a sense of the shape string specification).

## Build a pip package for your custom op

To build a pip package for your op, see the [tensorflow/custom-op](#) (<https://github.com/tensorflow/custom-op>) example. This guide shows how to build custom ops from the TensorFlow pip package instead of building TensorFlow from source.

**Copyright 2020 The TensorFlow Authors.**

In [ ]:

```

#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

```

## Introduction to modules, layers, and models



[View on TensorFlow.org](https://www.tensorflow.org/guide/intro_to_modules)

([https://www.tensorflow.org/guide/intro\\_to\\_modules](https://www.tensorflow.org/guide/intro_to_modules)) ([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/intro\\_to\\_modules.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/intro_to_modules.ipynb)) ([https://github.com/tensorflow/docs/blob/master/site/en/guide/intro\\_to\\_modules.ipynb](https://github.com/tensorflow/docs/blob/master/site/en/guide/intro_to_modules.ipynb))



[Run in Google Colab](#)

To do machine learning in TensorFlow, you are likely to need to define, save, and restore a model.

A model is, abstractly:

- A function that computes something on tensors (a **forward pass**)
- Some variables that can be updated in response to training

In this guide, you will go below the surface of Keras to see how TensorFlow models are defined. This looks at how TensorFlow collects variables and models, as well as how they are saved and restored.

Note: If you instead want to immediately get started with Keras, please see [the collection of Keras guides \(./keras/\)](#).

## Setup

In [ ]:

```
import tensorflow as tf
from datetime import datetime

%load_ext tensorboard
```

## Defining models and layers in TensorFlow

Most models are made of layers. Layers are functions with a known mathematical structure that can be reused and have trainable variables. In TensorFlow, most high-level implementations of layers and models, such as Keras or [Sonnet \(https://github.com/deepmind/sonnet\)](#), are built on the same foundational class: `tf.Module`.

Here's an example of a very simple `tf.Module` that operates on a scalar tensor:

In [ ]:

```
class SimpleModule(tf.Module):
    def __init__(self, name=None):
        super().__init__(name=name)
        self.a_variable = tf.Variable(5.0, name="train_me")
        self.non_trainable_variable = tf.Variable(5.0, trainable=False, name="do_not_train_me")
    def __call__(self, x):
        return self.a_variable * x + self.non_trainable_variable

simple_module = SimpleModule(name="simple")
simple_module(tf.constant(5.0))
```

Modules and, by extension, layers are deep-learning terminology for "objects": they have internal state, and methods that use that state.

There is nothing special about `__call__` except to act like a [Python callable \(https://stackoverflow.com/questions/111234/what-is-a-callable\)](#); you can invoke your models with whatever functions you wish.

You can set the trainability of variables on and off for any reason, including freezing layers and variables during fine-tuning.

Note: `tf.Module` is the base class for both `tf.keras.layers.Layer` and `tf.keras.Model`, so everything you come across here also applies in Keras. For historical compatibility reasons Keras layers do not collect variables from modules, so your models should use only modules or only Keras layers. However, the methods shown below for inspecting variables are the same in either case.

By subclassing `tf.Module`, any `tf.Variable` or `tf.Module` instances assigned to this object's properties are automatically collected. This allows you to save and load variables, and also create collections of `tf.Module`s.

In [ ]:

```
# All trainable variables
print("trainable variables:", simple_module.trainable_variables)
# Every variable
print("all variables:", simple_module.variables)
```

This is an example of a two-layer linear layer model made out of modules.

First a dense (linear) layer:

In [ ]:

```
class Dense(tf.Module):
    def __init__(self, in_features, out_features, name=None):
        super().__init__(name=name)
        self.w = tf.Variable(
            tf.random.normal([in_features, out_features]), name='w')
        self.b = tf.Variable(tf.zeros([out_features]), name='b')
    def __call__(self, x):
        y = tf.matmul(x, self.w) + self.b
        return tf.nn.relu(y)
```

And then the complete model, which makes two layer instances and applies them:

In [ ]:

```
class SequentialModule(tf.Module):
    def __init__(self, name=None):
        super().__init__(name=name)

        self.dense_1 = Dense(in_features=3, out_features=3)
        self.dense_2 = Dense(in_features=3, out_features=2)

    def __call__(self, x):
        x = self.dense_1(x)
        return self.dense_2(x)

# You have made a model!
my_model = SequentialModule(name="the_model")

# Call it, with random results
print("Model results:", my_model(tf.constant([[2.0, 2.0, 2.0]])))
```

`tf.Module` instances will automatically collect, recursively, any `tf.Variable` or `tf.Module` instances assigned to it. This allows you to manage collections of `tf.Module`s with a single model instance, and save and load whole models.

In [ ]:

```
print("Submodules:", my_model.submodules)
```

In [ ]:

```
for var in my_model.variables:
    print(var, "\n")
```

## Waiting to create variables

You may have noticed here that you have to define both input and output sizes to the layer. This is so the `w` variable has a known shape and can be allocated.

By deferring variable creation to the first time the module is called with a specific input shape, you do not need specify the input size up front.

In [ ]:

```
class FlexibleDenseModule(tf.Module):
    # Note: No need for `in_features`
    def __init__(self, out_features, name=None):
        super().__init__(name=name)
        self.is_built = False
        self.out_features = out_features

    def __call__(self, x):
        # Create variables on first call.
        if not self.is_built:
            self.w = tf.Variable(
                tf.random.normal([x.shape[-1], self.out_features]), name='w')
            self.b = tf.Variable(tf.zeros([self.out_features]), name='b')
            self.is_built = True

        y = tf.matmul(x, self.w) + self.b
        return tf.nn.relu(y)
```

In [ ]:

```
# Used in a module
class MySequentialModule(tf.Module):
    def __init__(self, name=None):
        super().__init__(name=name)

        self.dense_1 = FlexibleDenseModule(out_features=3)
        self.dense_2 = FlexibleDenseModule(out_features=2)

    def __call__(self, x):
        x = self.dense_1(x)
        return self.dense_2(x)

my_model = MySequentialModule(name="the_model")
print("Model results:", my_model(tf.constant([[2.0, 2.0, 2.0]])))
```

This flexibility is why TensorFlow layers often only need to specify the shape of their outputs, such as in `tf.keras.layers.Dense`, rather than both the input and output size.

## Saving weights

You can save a `tf.Module` as both a [checkpoint \(./checkpoint.ipynb\)](#) and a [SavedModel \(./saved\\_model.ipynb\)](#).

Checkpoints are just the weights (that is, the values of the set of variables inside the module and its submodules):

In [ ]:

```
chkp_path = "my_checkpoint"
checkpoint = tf.train.Checkpoint(model=my_model)
checkpoint.write(chkp_path)
```

Checkpoints consist of two kinds of files: the data itself and an index file for metadata. The index file keeps track of what is actually saved and the numbering of checkpoints, while the checkpoint data contains the variable values and their attribute lookup paths.

In [ ]:

```
!ls my_checkpoint*
```

You can look inside a checkpoint to be sure the whole collection of variables is saved, sorted by the Python object that contains them.

In [ ]:

```
tf.train.list_variables(chkp_path)
```

During distributed (multi-machine) training they can be sharded, which is why they are numbered (e.g., '00000-of-00001'). In this case, though, there is only have one shard.

When you load models back in, you overwrite the values in your Python object.

In [ ]:

```
new_model = MySequentialModule()
new_checkpoint = tf.train.Checkpoint(model=new_model)
new_checkpoint.restore("my_checkpoint")

# Should be the same result as above
new_model(tf.constant([[2.0, 2.0, 2.0]]))
```

Note: As checkpoints are at the heart of long training workflows `tf.checkpoint.CheckpointManager` is a helper class that makes checkpoint management much easier. Refer to the [Training checkpoints guide \(./checkpoint.ipynb\)](#) for more details.

## Saving functions

TensorFlow can run models without the original Python objects, as demonstrated by [TensorFlow Serving \(https://tensorflow.org/tfx\)](#) and [TensorFlow Lite \(https://tensorflow.org/lite\)](#), even when you download a trained model from [TensorFlow Hub \(https://tensorflow.org/hub\)](#).

TensorFlow needs to know how to do the computations described in Python, but **without the original code**. To do this, you can make a **graph**, which is described in the [Introduction to graphs and functions guide \(./intro\\_to\\_graphs.ipynb\)](#).

This graph contains operations, or `ops`, that implement the function.

You can define a graph in the model above by adding the `@tf.function` decorator to indicate that this code should run as a graph.

In [ ]:

```
class MySequentialModule(tf.Module):
    def __init__(self, name=None):
        super().__init__(name=name)

        self.dense_1 = Dense(in_features=3, out_features=3)
        self.dense_2 = Dense(in_features=3, out_features=2)

    @tf.function
    def __call__(self, x):
        x = self.dense_1(x)
        return self.dense_2(x)

# You have made a model with a graph!
my_model = MySequentialModule(name="the_model")
```

The module you have made works exactly the same as before. Each unique signature passed into the function creates a separate graph. Check the [Introduction to graphs and functions guide \(./intro\\_to\\_graphs.ipynb\)](#) for details.

In [ ]:

```
print(my_model([[2.0, 2.0, 2.0]]))
print(my_model([[2.0, 2.0, 2.0], [2.0, 2.0, 2.0]]))
```

You can visualize the graph by tracing it within a TensorBoard summary.

In [ ]:

```
# Set up logging.
stamp = datetime.now().strftime("%Y%m%d-%H%M%S")
logdir = "logs/func/%s" % stamp
writer = tf.summary.create_file_writer(logdir)

# Create a new model to get a fresh trace
# Otherwise the summary will not see the graph.
new_model = MySequentialModule()

# Bracket the function call with
# tf.summary.trace_on() and tf.summary.trace_export().
tf.summary.trace_on(graph=True)
tf.profiler.experimental.start(logdir)
# Call only one tf.function when tracing.
z = print(new_model(tf.constant([[2.0, 2.0, 2.0]])))
with writer.as_default():
    tf.summary.trace_export(
        name="my_func_trace",
        step=0,
        profiler_outdir=logdir)
```

Launch TensorBoard to view the resulting trace:

In [ ]:

```
#docs_infra: no_execute
%tensorboard --logdir logs/func
```

Search nodes. Regexes support...

 Fit to Screen Download PNG

Run 20200804-144509 (2)

Tag my\_func\_trace (1)

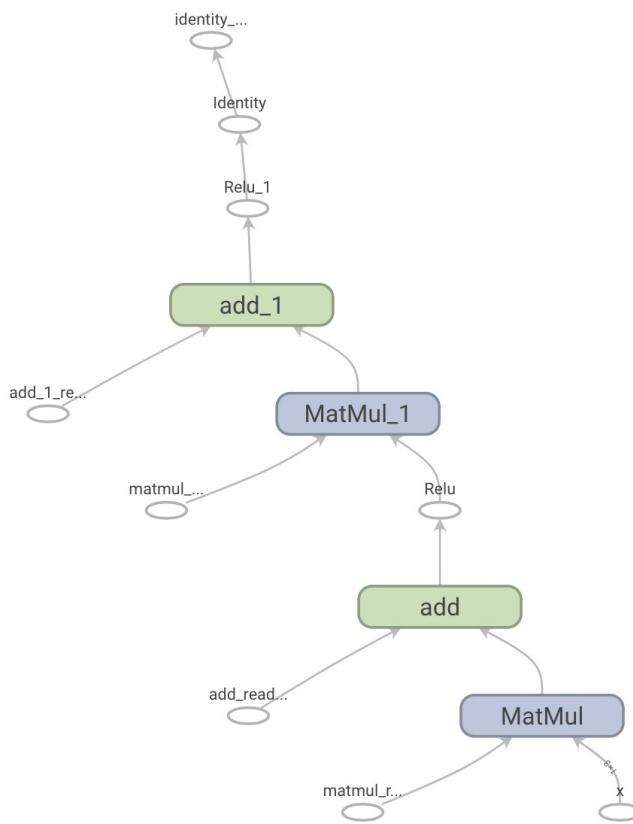
Upload  Choose File Graph Conceptual Graph Profile Trace inputs Show health pillsColor  Structure

▼ Close legend.

Graph (\* = expandable)

- Namespace\* ?
- OpNode ?
- Unconnected series\* ?
- Connected series\* ?
- Constant ?
- Summary ?

## Main Graph



## Creating a SavedModel

The recommended way of sharing completely trained models is to use `SavedModel`. `SavedModel` contains both a collection of functions and a collection of weights.

You can save the model you have just trained as follows:

In [ ]:

```
tf.saved_model.save(my_model, "the_saved_model")
```

In [ ]:

```
# Inspect the SavedModel in the directory
!ls -l the_saved_model
```

In [ ]:

```
# The variables/ directory contains a checkpoint of the variables
!ls -l the_saved_model/variables
```

The `saved_model.pb` file is a [protocol buffer](https://developers.google.com/protocol-buffers) (<https://developers.google.com/protocol-buffers>) describing the functional `tf.Graph`.

Models and layers can be loaded from this representation without actually making an instance of the class that created it. This is desired in situations where you do not have (or want) a Python interpreter, such as serving at scale or on an edge device, or in situations where the original Python code is not available or practical to use.

You can load the model as new object:

In [ ]:

```
new_model = tf.saved_model.load("the_saved_model")
```

`new_model`, created from loading a saved model, is an internal TensorFlow user object without any of the class knowledge. It is not of type `SequentialModule`.

In [ ]:

```
isinstance(new_model, SequentialModule)
```

This new model works on the already-defined input signatures. You can't add more signatures to a model restored like this.

In [ ]:

```
print(my_model([[2.0, 2.0, 2.0]]))
print(my_model([[2.0, 2.0, 2.0], [2.0, 2.0, 2.0]]))
```

Thus, using `SavedModel`, you are able to save TensorFlow weights and graphs using `tf.Module`, and then load them again.

## Keras models and layers

Note that up until this point, there is no mention of Keras. You can build your own high-level API on top of `tf.Module`, and people have.

In this section, you will examine how Keras uses `tf.Module`. A complete user guide to Keras models can be found in the [Keras guide](https://www.tensorflow.org/guide/keras/sequential_model) ([https://www.tensorflow.org/guide/keras/sequential\\_model](https://www.tensorflow.org/guide/keras/sequential_model)).

## Keras layers

`tf.keras.layers.Layer` is the base class of all Keras layers, and it inherits from `tf.Module`.

You can convert a module into a Keras layer just by swapping out the parent and then changing `__call__` to `call`:

In [ ]:

```
class MyDense(tf.keras.layers.Layer):
    # Adding **kwargs to support base Keras layer arguments
    def __init__(self, in_features, out_features, **kwargs):
        super().__init__(**kwargs)

    # This will soon move to the build step; see below
    self.w = tf.Variable(
        tf.random.normal([in_features, out_features]), name='w')
    self.b = tf.Variable(tf.zeros([out_features]), name='b')
    def call(self, x):
        y = tf.matmul(x, self.w) + self.b
        return tf.nn.relu(y)

simple_layer = MyDense(name="simple", in_features=3, out_features=3)
```

Keras layers have their own `__call__` that does some bookkeeping described in the next section and then calls `call()`. You should notice no change in functionality.

In [ ]:

```
simple_layer([[2.0, 2.0, 2.0]])
```

## The build step

As noted, it's convenient in many cases to wait to create variables until you are sure of the input shape.

Keras layers come with an extra lifecycle step that allows you more flexibility in how you define your layers. This is defined in the `build` function.

`build` is called exactly once, and it is called with the shape of the input. It's usually used to create variables (weights).

You can rewrite `MyDense` layer above to be flexible to the size of its inputs:

```
In [ ]:
```

```
class FlexibleDense(tf.keras.layers.Layer):
    # Note the added `**kwargs`, as Keras supports many arguments
    def __init__(self, out_features, **kwargs):
        super().__init__(**kwargs)
        self.out_features = out_features

    def build(self, input_shape): # Create the state of the layer (weights)
        self.w = tf.Variable(
            tf.random.normal([input_shape[-1], self.out_features]), name='w')
        self.b = tf.Variable(tf.zeros([self.out_features]), name='b')

    def call(self, inputs): # Defines the computation from inputs to outputs
        return tf.matmul(inputs, self.w) + self.b

# Create the instance of the layer
flexible_dense = FlexibleDense(out_features=3)
```

At this point, the model has not been built, so there are no variables:

```
In [ ]:
```

```
flexible_dense.variables
```

Calling the function allocates appropriately-sized variables:

```
In [ ]:
```

```
# Call it, with predictably random results
print("Model results:", flexible_dense(tf.constant([[2.0, 2.0, 2.0], [3.0, 3.0, 3.0]])))
```

```
In [ ]:
```

```
flexible_dense.variables
```

Since `build` is only called once, inputs will be rejected if the input shape is not compatible with the layer's variables:

```
In [ ]:
```

```
try:
    print("Model results:", flexible_dense(tf.constant([[2.0, 2.0, 2.0, 2.0]])))
except tf.errors.InvalidArgumentError as e:
    print("Failed:", e)
```

Keras layers have a lot more extra features including:

- Optional losses
- Support for metrics
- Built-in support for an optional `training` argument to differentiate between training and inference use
- `get_config` and `from_config` methods that allow you to accurately store configurations to allow model cloning in Python

Read about them in the [full guide \(./keras/custom\\_layers\\_and\\_models.ipynb\)](#) to custom layers and models.

## Keras models

You can define your model as nested Keras layers.

However, Keras also provides a full-featured model class called `tf.keras.Model`. It inherits from `tf.keras.layers.Layer`, so a Keras model can be used, nested, and saved in the same way as Keras layers. Keras models come with extra functionality that makes them easy to train, evaluate, load, save, and even train on multiple machines.

You can define the `SequentialModule` from above with nearly identical code, again converting `__call__` to `call()` and changing the parent:

In [ ]:

```
class MySequentialModel(tf.keras.Model):
    def __init__(self, name=None, **kwargs):
        super().__init__(**kwargs)

        self.dense_1 = FlexibleDense(out_features=3)
        self.dense_2 = FlexibleDense(out_features=2)
    def call(self, x):
        x = self.dense_1(x)
        return self.dense_2(x)

# You have made a Keras model!
my_sequential_model = MySequentialModel(name="the_model")

# Call it on a tensor, with random results
print("Model results:", my_sequential_model(tf.constant([[2.0, 2.0, 2.0]])))
```

All the same features are available, including tracking variables and submodules.

Note: To emphasize the note above, a raw `tf.Module` nested inside a Keras layer or model will not get its variables collected for training or saving. Instead, nest Keras layers inside of Keras layers.

In [ ]:

```
my_sequential_model.variables
```

In [ ]:

```
my_sequential_model.submodules
```

Overriding `tf.keras.Model` is a very Pythonic approach to building TensorFlow models. If you are migrating models from other frameworks, this can be very straightforward.

If you are constructing models that are simple assemblages of existing layers and inputs, you can save time and space by using the [functional API](#) ([./keras/functional.ipynb](#)), which comes with additional features around model reconstruction and architecture.

Here is the same model with the functional API:

In [ ]:

```
inputs = tf.keras.Input(shape=[3,])
x = FlexibleDense(3)(inputs)
x = FlexibleDense(2)(x)

my_functional_model = tf.keras.Model(inputs=inputs, outputs=x)
my_functional_model.summary()
```

In [ ]:

```
my_functional_model(tf.constant([[2.0, 2.0, 2.0]]))
```

The major difference here is that the input shape is specified up front as part of the functional construction process. The `input_shape` argument in this case does not have to be completely specified; you can leave some dimensions as `None`.

Note: You do not need to specify `input_shape` or an `InputLayer` in a subclassed model; these arguments and layers will be ignored.

## Saving Keras models

Keras models can be checkpointed, and that will look the same as `tf.Module`.

Keras models can also be saved with `tf.saved_model.save()`, as they are modules. However, Keras models have convenience methods and other functionality:

In [ ]:

```
my_sequential_model.save("exname_of_file")
```

Just as easily, they can be loaded back in:

In [ ]:

```
reconstructed_model = tf.keras.models.load_model("exname_of_file")
```

Keras SavedModels also save metric, loss, and optimizer states.

This reconstructed model can be used and will produce the same result when called on the same data:

In [ ]:

```
reconstructed_model(tf.constant([[2.0, 2.0, 2.0]]))
```

There is more to know about saving and serialization of Keras models, including providing configuration methods for custom layers for feature support. Check out the [guide to saving and serialization](https://www.tensorflow.org/guide/keras/save_and_serialize) ([https://www.tensorflow.org/guide/keras/save\\_and\\_serialize](https://www.tensorflow.org/guide/keras/save_and_serialize)).

## What's next

If you want to know more details about Keras, you can follow the existing Keras guides [here \(./keras/\)](#).

Another example of a high-level API built on `tf.module` is Sonnet from DeepMind, which is covered on [their site](#) (<https://github.com/deepmind/sonnet>).

**Copyright 2020 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## TensorFlow basics



[View on TensorFlow.org](https://www.tensorflow.org/guide/basics)

(<https://www.tensorflow.org/guide/basics>)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/basic.ipynb)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/basic.ipynb>)



This guide provides a quick overview of *TensorFlow basics*. Each section of this doc is an overview of a larger topic—you can find links to full guides at the end of each section.

TensorFlow is an end-to-end platform for machine learning. It supports the following:

- Multidimensional-array based numeric computation (similar to [NumPy](https://numpy.org/) (<https://numpy.org/>)).
- GPU and distributed processing
- Automatic differentiation
- Model construction, training, and export
- And more

## Tensors

TensorFlow operates on multidimensional arrays or *tensors* represented as `tf.Tensor` objects. Here is a two-dimensional tensor:

In [ ]:

```
import tensorflow as tf

x = tf.constant([[1., 2., 3.],
                [4., 5., 6.]])

print(x)
print(x.shape)
print(x.dtype)
```

The most important attributes of a `tf.Tensor` are its `shape` and `dtype`:

- `Tensor.shape` : tells you the size of the tensor along each of its axes.
- `Tensor.dtype` : tells you the type of all the elements in the tensor.

TensorFlow implements standard mathematical operations on tensors, as well as many operations specialized for machine learning.

For example:

In [ ]:

```
x + x
```

In [ ]:

```
5 * x
```

In [ ]:

```
x @ tf.transpose(x)
```

In [ ]:

```
tf.concat([x, x, x], axis=0)
```

In [ ]:

```
tf.nn.softmax(x, axis=-1)
```

In [ ]:

```
tf.reduce_sum(x)
```

Running large calculations on CPU can be slow. When properly configured, TensorFlow can use accelerator hardware like GPUs to execute operations very quickly.

In [ ]:

```
if tf.config.list_physical_devices('GPU'):
    print("TensorFlow **IS** using the GPU")
else:
    print("TensorFlow **IS NOT** using the GPU")
```

Refer to the [Tensor guide \(tensor.ipynb\)](#) for details.

## Variables

Normal `tf.Tensor` objects are immutable. To store model weights (or other mutable state) in TensorFlow use a `tf.Variable`.

In [ ]:

```
var = tf.Variable([0.0, 0.0, 0.0])
```

In [ ]:

```
var.assign([1, 2, 3])
```

In [ ]:

```
var.assign_add([1, 1, 1])
```

Refer to the [Variables guide \(variable.ipynb\)](#) for details.

## Automatic differentiation

[Gradient descent](#) ([https://en.wikipedia.org/wiki/Gradient\\_descent](https://en.wikipedia.org/wiki/Gradient_descent)) and related algorithms are a cornerstone of modern machine learning.

To enable this, TensorFlow implements automatic differentiation (autodiff), which uses calculus to compute gradients. Typically you'll use this to calculate the gradient of a model's `error` or `loss` with respect to its weights.

In [ ]:

```
x = tf.Variable(1.0)

def f(x):
    y = x**2 + 2*x - 5
    return y
```

In [ ]:

```
f(x)
```

At  $x = 1.0$ ,  $y = f(x) = (1**2 + 2*1 - 5) = -2$ .

The derivative of  $y$  is  $y' = f'(x) = (2*x + 2) = 4$ . TensorFlow can calculate this automatically:

In [ ]:

```
with tf.GradientTape() as tape:
    y = f(x)

g_x = tape.gradient(y, x) # g(x) = dy/dx

g_x
```

This simplified example only takes the derivative with respect to a single scalar ( $x$ ), but TensorFlow can compute the gradient with respect to any number of non-scalar tensors simultaneously.

Refer to the [Autodiff guide \(autodiff.ipynb\)](#) for details.

## Graphs and tf.function

While you can use TensorFlow interactively like any Python library, TensorFlow also provides tools for:

- **Performance optimization:** to speed up training and inference.
- **Export:** so you can save your model when it's done training.

These require that you use `tf.function` to separate your pure-TensorFlow code from Python.

In [ ]:

```
@tf.function
def my_func(x):
    print('Tracing.\n')
    return tf.reduce_sum(x)
```

The first time you run the `tf.function`, although it executes in Python, it captures a complete, optimized graph representing the TensorFlow computations done within the function.

In [ ]:

```
x = tf.constant([1, 2, 3])
my_func(x)
```

On subsequent calls TensorFlow only executes the optimized graph, skipping any non-TensorFlow steps. Below, note that `my_func` doesn't print *tracing* since `print` is a Python function, not a TensorFlow function.

In [ ]:

```
x = tf.constant([10, 9, 8])
my_func(x)
```

A graph may not be reusable for inputs with a different *signature* (`shape` and `dtype`), so a new graph is generated instead:

In [ ]:

```
x = tf.constant([10.0, 9.1, 8.2], dtype=tf.float32)
my_func(x)
```

These captured graphs provide two benefits:

- In many cases they provide a significant speedup in execution (though not this trivial example).
- You can export these graphs, using `tf.saved_model`, to run on other systems like a [server](https://www.tensorflow.org/tfx/serving/docker) (<https://www.tensorflow.org/tfx/serving/docker>) or a [mobile device](https://www.tensorflow.org/lite/guide) (<https://www.tensorflow.org/lite/guide>), no Python installation required.

Refer to [Intro to graphs \(intro\\_to\\_graphs.ipynb\)](#) for more details.

## Modules, layers, and models

`tf.Module` is a class for managing your `tf.Variable` objects, and the `tf.function` objects that operate on them. The `tf.Module` class is necessary to support two significant features:

1. You can save and restore the values of your variables using `tf.train.Checkpoint`. This is useful during training as it is quick to save and restore a model's state.
2. You can import and export the `tf.Variable` values *and* the `tf.function` graphs using `tf.saved_model`. This allows you to run your model independently of the Python program that created it.

Here is a complete example exporting a simple `tf.Module` object:

In [ ]:

```
class MyModule(tf.Module):
    def __init__(self, value):
        self.weight = tf.Variable(value)

    @tf.function
    def multiply(self, x):
        return x * self.weight
```

In [ ]:

```
mod = MyModule(3)
mod.multiply(tf.constant([1, 2, 3]))
```

Save the Module :

In [ ]:

```
save_path = './saved'
tf.saved_model.save(mod, save_path)
```

The resulting SavedModel is independent of the code that created it. You can load a SavedModel from Python, other language bindings, or [TensorFlow Serving](#) (<https://www.tensorflow.org/tfx/serving/docker>). You can also convert it to run with [TensorFlow Lite](#) (<https://www.tensorflow.org/lite/guide>) or [TensorFlow JS](#) (<https://www.tensorflow.org/js/guide>).

In [ ]:

```
reloaded = tf.saved_model.load(save_path)
reloaded.multiply(tf.constant([1, 2, 3]))
```

The `tf.keras.layers.Layer` and `tf.keras.Model` classes build on `tf.Module` providing additional functionality and convenience methods for building, training, and saving models. Some of these are demonstrated in the next section.

Refer to [Intro to modules \(intro\\_to\\_modules.ipynb\)](#) for details.

## Training loops

Now put this all together to build a basic model and train it from scratch.

First, create some example data. This generates a cloud of points that loosely follows a quadratic curve:

In [ ]:

```
import matplotlib
from matplotlib import pyplot as plt

matplotlib.rcParams['figure.figsize'] = [9, 6]
```

In [ ]:

```
x = tf.linspace(-2, 2, 201)
x = tf.cast(x, tf.float32)

def f(x):
    y = x**2 + 2*x - 5
    return y

y = f(x) + tf.random.normal(shape=[201])

plt.plot(x.numpy(), y.numpy(), '.', label='Data')
plt.plot(x, f(x), label='Ground truth')
plt.legend();
```

Create a model:

In [ ]:

```
class Model(tf.keras.Model):
    def __init__(self, units):
        super().__init__()
        self.dense1 = tf.keras.layers.Dense(units=units,
  activation=tf.nn.relu,
  kernel_initializer=tf.random.normal,
  bias_initializer=tf.random.normal)
        self.dense2 = tf.keras.layers.Dense(1)

    def call(self, x, training=True):
        # For Keras layers/models, implement `call` instead of `__call__`.
        x = x[:, tf.newaxis]
        x = self.dense1(x)
        x = self.dense2(x)
        return tf.squeeze(x, axis=1)
```

In [ ]:

```
model = Model(64)
```

In [ ]:

```
plt.plot(x.numpy(), y.numpy(), '.', label='data')
plt.plot(x, f(x), label='Ground truth')
plt.plot(x, model(x), label='Untrained predictions')
plt.title('Before training')
plt.legend();
```

Write a basic training loop:

In [ ]:

```
variables = model.variables

optimizer = tf.optimizers.SGD(learning_rate=0.01)

for step in range(1000):
    with tf.GradientTape() as tape:
        prediction = model(x)
        error = (y-prediction)**2
        mean_error = tf.reduce_mean(error)
    gradient = tape.gradient(mean_error, variables)
    optimizer.apply_gradients(zip(gradient, variables))

    if step % 100 == 0:
        print(f'Mean squared error: {mean_error.numpy():0.3f}')
```

In [ ]:

```
plt.plot(x.numpy(), y.numpy(), '.', label="data")
plt.plot(x, f(x), label='Ground truth')
plt.plot(x, model(x), label='Trained predictions')
plt.title('After training')
plt.legend();
```

That's working, but remember that implementations of common training utilities are available in the `tf.keras` module. So consider using those before writing your own. To start with, the `Model.compile` and `Model.fit` methods implement a training loop for you:

```
In [ ]:
```

```
new_model = Model(64)
```

```
In [ ]:
```

```
new_model.compile(  
    loss=tf.keras.losses.MSE,  
    optimizer=tf.optimizers.SGD(learning_rate=0.01))  
  
history = new_model.fit(x, y,  
    epochs=100,  
    batch_size=32,  
    verbose=0)  
  
model.save('./my_model')
```

```
In [ ]:
```

```
plt.plot(history.history['loss'])  
plt.xlabel('Epoch')  
plt.ylim([0, max(plt.ylim())])  
plt.ylabel('Loss [Mean Squared Error]')  
plt.title('Keras training progress');
```

Refer to [Basic training loops \(basic\\_training\\_loops.ipynb\)](#) and the [Keras guide \(https://www.tensorflow.org/guide/keras\)](#) for more details.

**Copyright 2020 The TensorFlow Authors.**

```
In [ ]:
```

```
#@title Licensed under the Apache License, Version 2.0 (the "License");  
# you may not use this file except in compliance with the License.  
# You may obtain a copy of the License at  
#  
# https://www.apache.org/licenses/LICENSE-2.0  
#  
# Unless required by applicable law or agreed to in writing, software  
# distributed under the License is distributed on an "AS IS" BASIS,  
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
# See the License for the specific language governing permissions and  
# limitations under the License.
```

## Introduction to Variables



[View on TensorFlow.org](#)

(<https://www.tensorflow.org/guide/variable>)



[Run in Google Colab](#)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/variable.ipynb>)

▶

A TensorFlow **variable** is the recommended way to represent shared, persistent state your program manipulates. This guide covers how to create, update, and manage instances of `tf.Variable` in TensorFlow.

Variables are created and tracked via the `tf.Variable` class. A `tf.Variable` represents a tensor whose value can be changed by running ops on it. Specific ops allow you to read and modify the values of this tensor. Higher level libraries like `tf.keras` use `tf.Variable` to store model parameters.

## Setup

This notebook discusses variable placement. If you want to see on what device your variables are placed, uncomment this line.

```
In [ ]:
```

```
import tensorflow as tf  
  
# Uncomment to see where your variables get placed (see below)  
# tf.debugging.set_log_device_placement(True)
```

## Create a variable

To create a variable, provide an initial value. The `tf.Variable` will have the same `dtype` as the initialization value.

In [ ]:

```
my_tensor = tf.constant([[1.0, 2.0], [3.0, 4.0]])
my_variable = tf.Variable(my_tensor)

# Variables can be all kinds of types, just like tensors
bool_variable = tf.Variable([False, False, False, True])
complex_variable = tf.Variable([5 + 4j, 6 + 1j])
```

A variable looks and acts like a tensor, and, in fact, is a data structure backed by a `tf.Tensor`. Like tensors, they have a `dtype` and a `shape`, and can be exported to NumPy.

In [ ]:

```
print("Shape: ", my_variable.shape)
print("DType: ", my_variable.dtype)
print("As NumPy: ", my_variable.numpy())
```

Most tensor operations work on variables as expected, although variables cannot be reshaped.

In [ ]:

```
print("A variable:", my_variable)
print("\nViewed as a tensor:", tf.convert_to_tensor(my_variable))
print("\nIndex of highest value:", tf.argmax(my_variable))

# This creates a new tensor; it does not reshape the variable.
print("\nCopying and reshaping: ", tf.reshape(my_variable, [1,4]))
```

As noted above, variables are backed by tensors. You can reassign the tensor using `tf.Variable.assign`. Calling `assign` does not (usually) allocate a new tensor; instead, the existing tensor's memory is reused.

In [ ]:

```
a = tf.Variable([2.0, 3.0])
# This will keep the same dtype, float32
a.assign([1, 2])
# Not allowed as it resizes the variable:
try:
    a.assign([1.0, 2.0, 3.0])
except Exception as e:
    print(f"type(e).__name__: {e}")
```

If you use a variable like a tensor in operations, you will usually operate on the backing tensor.

Creating new variables from existing variables duplicates the backing tensors. Two variables will not share the same memory.

In [ ]:

```
a = tf.Variable([2.0, 3.0])
# Create b based on the value of a
b = tf.Variable(a)
a.assign([5, 6])

# a and b are different
print(a.numpy())
print(b.numpy())

# There are other versions of assign
print(a.assign_add([2,3]).numpy()) # [7. 9.]
print(a.assign_sub([7,9]).numpy()) # [0. 0.]
```

## Lifecycles, naming, and watching

In Python-based TensorFlow, `tf.Variable` instances have the same lifecycle as other Python objects. When there are no references to a variable it is automatically deallocated.

Variables can also be named which can help you track and debug them. You can give two variables the same name.

In [ ]:

```
# Create a and b; they will have the same name but will be backed by
# different tensors.
a = tf.Variable(my_tensor, name="Mark")
# A new variable with the same name, but different value
# Note that the scalar add is broadcast
b = tf.Variable(my_tensor + 1, name="Mark")

# These are elementwise-unequal, despite having the same name
print(a == b)
```

Variable names are preserved when saving and loading models. By default, variables in models will acquire unique variable names automatically, so you don't need to assign them yourself unless you want to.

Although variables are important for differentiation, some variables will not need to be differentiated. You can turn off gradients for a variable by setting `trainable` to `false` at creation. An example of a variable that would not need gradients is a training step counter.

In [ ]:

```
step_counter = tf.Variable(1, trainable=False)
```

## Placing variables and tensors

For better performance, TensorFlow will attempt to place tensors and variables on the fastest device compatible with its `dtype`. This means most variables are placed on a GPU if one is available.

However, you can override this. In this snippet, place a float tensor and a variable on the CPU, even if a GPU is available. By turning on device placement logging (see [Setup](#)), you can see where the variable is placed.

Note: Although manual placement works, using [distribution strategies \(distributed\\_training.ipynb\)](#) can be a more convenient and scalable way to optimize your computation.

If you run this notebook on different backends with and without a GPU you will see different logging. *Note that logging device placement must be turned on at the start of the session.*

In [ ]:

```
with tf.device('CPU:0'):

    # Create some tensors
    a = tf.Variable([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
    b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
    c = tf.matmul(a, b)

print(c)
```

It's possible to set the location of a variable or tensor on one device and do the computation on another device. This will introduce delay, as data needs to be copied between the devices.

You might do this, however, if you had multiple GPU workers but only want one copy of the variables.

In [ ]:

```
with tf.device('CPU:0'):
    a = tf.Variable([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
    b = tf.Variable([[1.0, 2.0, 3.0]])

with tf.device('GPU:0'):
    # Element-wise multiply
    k = a * b

print(k)
```

Note: Because `tf.config.set_soft_device_placement` is turned on by default, even if you run this code on a device without a GPU, it will still run. The multiplication step will happen on the CPU.

For more on distributed training, refer to the [guide \(distributed\\_training.ipynb\)](#).

## Next steps

To understand how variables are typically used, see our guide on [automatic differentiation \(autodiff.ipynb\)](#).

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Effective Tensorflow 2



[View on TensorFlow.org](https://www.tensorflow.org/guide/effective_tf2)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/effective_tf2.ipynb)

### Overview

This guide provides a list of best practices for writing code using TensorFlow 2 (TF2), it is written for users who have recently switched over from TensorFlow 1 (TF1). Refer to the [migrate section of the guide](https://tensorflow.org/guide/migrate) (<https://tensorflow.org/guide/migrate>) for more info on migrating your TF1 code to TF2.

### Setup

Import TensorFlow and other dependencies for the examples in this guide.

In [ ]:

```
import tensorflow as tf
import tensorflow_datasets as tfds
```

## Recommendations for idiomatic TensorFlow 2

### Refactor your code into smaller modules

A good practice is to refactor your code into smaller functions that are called as needed. For best performance, you should try to decorate the largest blocks of computation that you can in a `tf.function` (note that the nested python functions called by a `tf.function` do not require their own separate decorations, unless you want to use different `jit_compile` settings for the `tf.function`). Depending on your use case, this could be multiple training steps or even your whole training loop. For inference use cases, it might be a single model forward pass.

### Adjust the default learning rate for some `tf.keras.optimizers`

Some Keras optimizers have different learning rates in TF2. If you see a change in convergence behavior for your models, check the default learning rates.

There are no changes for `optimizers.SGD`, `optimizers.Adam`, or `optimizers.RMSprop`.

The following default learning rates have changed:

- `optimizers.Adagrad` from 0.01 to 0.001
- `optimizers.Adadelta` from 1.0 to 0.001
- `optimizers.Adamax` from 0.002 to 0.001
- `optimizers.Nadam` from 0.002 to 0.001

### Use `tf.Modules` and Keras layers to manage variables

`tf.Modules` and `tf.keras.layers`.`Layer`s offer the convenient `variables` and `trainable_variables` properties, which recursively gather up all dependent variables. This makes it easy to manage variables locally to where they are being used.

Keras layers/models inherit from `tf.train.Checkpointable` and are integrated with `@tf.function`, which makes it possible to directly checkpoint or export SavedModels from Keras objects. You do not necessarily have to use Keras' `Model.fit` API to take advantage of these integrations.

Read the section on [transfer learning and fine-tuning](https://www.tensorflow.org/guide/keras/transfer_learning#transfer_learning_fine-tuning_with_a_custom_training_loop) ([https://www.tensorflow.org/guide/keras/transfer\\_learning#transfer\\_learning\\_fine-tuning\\_with\\_a\\_custom\\_training\\_loop](https://www.tensorflow.org/guide/keras/transfer_learning#transfer_learning_fine-tuning_with_a_custom_training_loop)) in the Keras guide to learn how to collect a subset of relevant variables using Keras.

## Combine `tf.data.Dataset`s and `tf.function`

The [TensorFlow Datasets](https://tensorflow.org/datasets) (<https://tensorflow.org/datasets>) package (`tfds`) contains utilities for loading predefined datasets as `tf.data.Dataset` objects. For this example, you can load the MNIST dataset using `tfds`:

In [ ]:

```
datasets, info = tfds.load(name='mnist', with_info=True, as_supervised=True)
mnist_train, mnist_test = datasets['train'], datasets['test']
```

Then prepare the data for training:

- Re-scale each image.
- Shuffle the order of the examples.
- Collect batches of images and labels.

In [ ]:

```
BUFFER_SIZE = 10 # Use a much larger value for real code
BATCH_SIZE = 64
NUM_EPOCHS = 5

def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255

    return image, label
```

To keep the example short, trim the dataset to only return 5 batches:

In [ ]:

```
train_data = mnist_train.map(scale).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
test_data = mnist_test.map(scale).batch(BATCH_SIZE)

STEPS_PER_EPOCH = 5

train_data = train_data.take(STEPS_PER_EPOCH)
test_data = test_data.take(STEPS_PER_EPOCH)
```

In [ ]:

```
image_batch, label_batch = next(iter(train_data))
```

Use regular Python iteration to iterate over training data that fits in memory. Otherwise, `tf.data.Dataset` is the best way to stream training data from disk. Datasets are [iterables \(not iterators\)](https://docs.python.org/3/glossary.html#term-iterable) (<https://docs.python.org/3/glossary.html#term-iterable>), and work just like other Python iterables in eager execution. You can fully utilize dataset async prefetching/streaming features by wrapping your code in `tf.function`, which replaces Python iteration with the equivalent graph operations using AutoGraph.

```
@tf.function
def train(model, dataset, optimizer):
    for x, y in dataset:
        with tf.GradientTape() as tape:
            # training=True is only needed if there are layers with different
            # behavior during training versus inference (e.g. Dropout).
            prediction = model(x, training=True)
            loss = loss_fn(prediction, y)
            gradients = tape.gradient(loss, model.trainable_variables)
            optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

If you use the Keras `Model.fit` API, you won't have to worry about dataset iteration.

```
model.compile(optimizer=optimizer, loss=loss_fn)
model.fit(dataset)
```

## Use Keras training loops

If you don't need low-level control of your training process, using Keras' built-in `fit`, `evaluate`, and `predict` methods is recommended. These methods provide a uniform interface to train the model regardless of the implementation (sequential, functional, or sub-classed).

The advantages of these methods include:

- They accept Numpy arrays, Python generators and, `tf.data.Datasets`.
- They apply regularization, and activation losses automatically.
- They support `tf.distribute` where the training code remains the same [regardless of the hardware configuration \(distributed\\_training.ipynb\)](#).
- They support arbitrary callables as losses and metrics.
- They support callbacks like `tf.keras.callbacks.TensorBoard`, and custom callbacks.
- They are performant, automatically using TensorFlow graphs.

Here is an example of training a model using a `Dataset`. For details on how this works, check out the [tutorials \(https://tensorflow.org/tutorials\)](#).

In [ ]:

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, 3, activation='relu',
                          kernel_regularizer=tf.keras.regularizers.l2(0.02),
                          input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(10)
])

# Model is the full model w/o custom layers
model.compile(optimizer='adam',
               loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
               metrics=['accuracy'])

model.fit(train_data, epochs=NUM_EPOCHS)
loss, acc = model.evaluate(test_data)

print("Loss {}, Accuracy {}".format(loss, acc))
```

## Customize training and write your own loop

If Keras models work for you, but you need more flexibility and control of the training step or the outer training loops, you can implement your own training steps or even entire training loops. See the Keras guide on [customizing fit \(https://www.tensorflow.org/guide/keras/customizing\\_what\\_happens\\_in\\_fit\)](#) to learn more.

You can also implement many things as a `tf.keras.callbacks.Callback`.

This method has many of the advantages [mentioned previously](#), but gives you control of the train step and even the outer loop.

There are three steps to a standard training loop:

1. Iterate over a Python generator or `tf.data.Dataset` to get batches of examples.
2. Use `tf.GradientTape` to collect gradients.
3. Use one of the `tf.keras.optimizers` to apply weight updates to the model's variables.

Remember:

- Always include a `training` argument on the `call` method of subclassed layers and models.
- Make sure to call the model with the `training` argument set correctly.
- Depending on usage, model variables may not exist until the model is run on a batch of data.
- You need to manually handle things like regularization losses for the model.

There is no need to run variable initializers or to add manual control dependencies. `tf.function` handles automatic control dependencies and variable initialization on creation for you.

In [ ]:

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, 3, activation='relu',
                          kernel_regularizer=tf.keras.regularizers.l2(0.02),
                          input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(10)
])

optimizer = tf.keras.optimizers.Adam(0.001)
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

@tf.function
def train_step(inputs, labels):
    with tf.GradientTape() as tape:
        predictions = model(inputs, training=True)
        regularization_loss = tf.math.add_n(model.losses)
        pred_loss = loss_fn(labels, predictions)
        total_loss = pred_loss + regularization_loss

    gradients = tape.gradient(total_loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

for epoch in range(NUM_EPOCHS):
    for inputs, labels in train_data:
        train_step(inputs, labels)
    print("Finished epoch", epoch)
```

## Take advantage of `tf.function` with Python control flow

`tf.function` provides a way to convert data-dependent control flow into graph-mode equivalents like `tf.cond` and `tf.while_loop`.

One common place where data-dependent control flow appears is in sequence models. `tf.keras.layers.RNN` wraps an RNN cell, allowing you to either statically or dynamically unroll the recurrence. As an example, you could reimplement dynamic unroll as follows.

In [ ]:

```
class DynamicRNN(tf.keras.Model):

    def __init__(self, rnn_cell):
        super(DynamicRNN, self).__init__()
        self.cell = rnn_cell

    @tf.function(input_signature=[tf.TensorSpec(dtype=tf.float32, shape=[None, None, 3])])
    def call(self, input_data):

        # [batch, time, features] -> [time, batch, features]
        input_data = tf.transpose(input_data, [1, 0, 2])
        timesteps = tf.shape(input_data)[0]
        batch_size = tf.shape(input_data)[1]
        outputs = tf.TensorArray(tf.float32, timesteps)
        state = self.cell.get_initial_state(batch_size=batch_size, dtype=tf.float32)
        for i in tf.range(timesteps):
            output, state = self.cell(input_data[i], state)
            outputs = outputs.write(i, output)
        return tf.transpose(outputs.stack(), [1, 0, 2]), state
```

In [ ]:

```
lstm_cell = tf.keras.layers.LSTMCell(units=13)

my_rnn = DynamicRNN(lstm_cell)
outputs, state = my_rnn(tf.random.normal(shape=[10, 20, 3]))
print(outputs.shape)
```

Read the [tf.function guide](https://www.tensorflow.org/guide/function) (<https://www.tensorflow.org/guide/function>) for a more information.

## New-style metrics and losses

Metrics and losses are both objects that work eagerly and in `tf.function`s.

A loss object is callable, and expects (`y_true`, `y_pred`) as arguments:

In [ ]:

```
cce = tf.keras.losses.CategoricalCrossentropy(from_logits=True)
cce([[1, 0]], [[-1.0, 3.0]]).numpy()
```

### Use metrics to collect and display data

You can use `tf.metrics` to aggregate data and `tf.summary` to log summaries and redirect it to a writer using a context manager. The summaries are emitted directly to the writer which means that you must provide the `step` value at the callsite.

```
summary_writer = tf.summary.create_file_writer('/tmp/summaries')
with summary_writer.as_default():
    tf.summary.scalar('loss', 0.1, step=42)
```

Use `tf.metrics` to aggregate data before logging them as summaries. Metrics are stateful; they accumulate values and return a cumulative result when you call the `result` method (such as `Mean.result`). Clear accumulated values with `Model.reset_states`.

```
def train(model, optimizer, dataset, log_freq=10):
    avg_loss = tf.keras.metrics.Mean(name='loss', dtype=tf.float32)
    for images, labels in dataset:
        loss = train_step(model, optimizer, images, labels)
        avg_loss.update_state(loss)
        if tf.equal(optimizer.iterations % log_freq, 0):
            tf.summary.scalar('loss', avg_loss.result(), step=optimizer.iterations)
            avg_loss.reset_states()

def test(model, test_x, test_y, step_num):
    # training=False is only needed if there are layers with different
    # behavior during training versus inference (e.g. Dropout).
    loss = loss_fn(model(test_x, training=False), test_y)
    tf.summary.scalar('loss', loss, step=step_num)

train_summary_writer = tf.summary.create_file_writer('/tmp/summaries/train')
test_summary_writer = tf.summary.create_file_writer('/tmp/summaries/test')

with train_summary_writer.as_default():
    train(model, optimizer, dataset)

with test_summary_writer.as_default():
    test(model, test_x, test_y, optimizer.iterations)
```

Visualize the generated summaries by pointing TensorBoard to the summary log directory:

```
tensorboard --logdir /tmp/summaries
```

Use the `tf.summary` API to write summary data for visualization in TensorBoard. For more info, read the [tf.summary guide](#) ([https://www.tensorflow.org/tensorboard/migrate#in\\_tf\\_2x](https://www.tensorflow.org/tensorboard/migrate#in_tf_2x)).

In [ ]:

```
# Create the metrics
loss_metric = tf.keras.metrics.Mean(name='train_loss')
accuracy_metric = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')

@tf.function
def train_step(inputs, labels):
    with tf.GradientTape() as tape:
        predictions = model(inputs, training=True)
        regularization_loss=tf.math.add_n(model.losses)
        pred_loss=loss_fn(labels, predictions)
        total_loss=pred_loss + regularization_loss

    gradients = tape.gradient(total_loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    # Update the metrics
    loss_metric.update_state(total_loss)
    accuracy_metric.update_state(labels, predictions)

for epoch in range(NUM_EPOCHS):
    # Reset the metrics
    loss_metric.reset_states()
    accuracy_metric.reset_states()

    for inputs, labels in train_data:
        train_step(inputs, labels)
    # Get the metric results
    mean_loss=loss_metric.result()
    mean_accuracy = accuracy_metric.result()

    print('Epoch: ', epoch)
    print(' loss: {:.3f}'.format(mean_loss))
    print(' accuracy: {:.3f}'.format(mean_accuracy))
```

## Keras metric names

Keras models are consistent about handling metric names. When you pass a string in the list of metrics, that exact string is used as the metric's name . These names are visible in the history object returned by `model.fit` , and in the logs passed to `keras.callbacks` . is set to the string you passed in the metric list.

In [ ]:

```
model.compile(
    optimizer = tf.keras.optimizers.Adam(0.001),
    loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics = ['acc', 'accuracy', tf.keras.metrics.SparseCategoricalAccuracy(name="my_accuracy")])
history = model.fit(train_data)
```

In [ ]:

```
history.history.keys()
```

## Debugging

Use eager execution to run your code step-by-step to inspect shapes, data types and values. Certain APIs, like `tf.function`, `tf.keras`, etc. are designed to use Graph execution, for performance and portability. When debugging, use `tf.config.run_functions_eagerly(True)` to use eager execution inside this code.

For example:

```
@tf.function
def f(x):
    if x > 0:
        import pdb
        pdb.set_trace()
    x = x + 1
    return x

tf.config.run_functions_eagerly(True)
f(tf.constant(1))

>>> f()
-> x = x + 1
(Pdb) l
 6     @tf.function
 7     def f(x):
 8         if x > 0:
 9             import pdb
10             pdb.set_trace()
11 ->         x = x + 1
12     return x
13
14     tf.config.run_functions_eagerly(True)
15     f(tf.constant(1))
[EOF]
```

This also works inside Keras models and other APIs that support eager execution:

```
class CustomModel(tf.keras.models.Model):

    @tf.function
    def call(self, input_data):
        if tf.reduce_mean(input_data) > 0:
            return input_data
        else:
            import pdb
            pdb.set_trace()
            return input_data // 2

tf.config.run_functions_eagerly(True)
model = CustomModel()
model(tf.constant([-2, -4]))

>>> call()
-> return input_data // 2
(Pdb) l
 10         if tf.reduce_mean(input_data) > 0:
 11             return input_data
 12         else:
 13             import pdb
 14             pdb.set_trace()
 15 ->         return input_data // 2
 16
 17
 18     tf.config.run_functions_eagerly(True)
 19     model = CustomModel()
 20     model(tf.constant([-2, -4]))
```

Notes:

- `tf.keras.Model` methods such as `fit`, `evaluate`, and `predict` execute as [graphs](https://www.tensorflow.org/guide/intro_to_graphs) ([https://www.tensorflow.org/guide/intro\\_to\\_graphs](https://www.tensorflow.org/guide/intro_to_graphs)) with `tf.function` under the hood.
- When using `tf.keras.Model.compile`, set `run_eagerly = True` to disable the `Model` logic from being wrapped in a `tf.function`.
- Use `tf.data.experimental.enable_debug_mode` to enable the debug mode for `tf.data`. Read the [API docs](https://www.tensorflow.org/api_docs/python/tf/data/experimental/enable_debug_mode) ([https://www.tensorflow.org/api\\_docs/python/tf/data/experimental/enable\\_debug\\_mode](https://www.tensorflow.org/api_docs/python/tf/data/experimental/enable_debug_mode)) for more details.

## Do not keep `tf.Tensors` in your objects

These tensor objects might get created either in a `tf.function` or in the eager context, and these tensors behave differently. Always use `tf.Tensor`s only for intermediate values.

To track state, use `tf.Variable`s as they are always usable from both contexts. Read the [tf.Variable guide](https://www.tensorflow.org/guide/variable) (<https://www.tensorflow.org/guide/variable>) to learn more.

## Resources and further reading

- Read the TF2 [guides](https://tensorflow.org/guide) (<https://tensorflow.org/guide>) and [tutorials](https://tensorflow.org/tutorials) (<https://tensorflow.org/tutorials>) to learn more about how to use TF2.
- If you previously used TF1.x, it is highly recommended you migrate your code to TF2. Read the [migration guides](https://tensorflow.org/guide/migrate) (<https://tensorflow.org/guide/migrate>) to learn more.

**Copyright 2020 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Better performance with `tf.function`



[View on TensorFlow.org](https://www.tensorflow.org/guide/function)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/function.ipynb)

In TensorFlow 2, [eager execution \(eager.ipynb\)](https://github.com/tensorflow/docs/blob/master/site/en/guide/function.ipynb) is turned on by default. The user interface is intuitive and flexible (running one-off operations is much easier and faster), but this can come at the expense of performance and deployability.

You can use `tf.function` to make graphs out of your programs. It is a transformation tool that creates Python-independent dataflow graphs out of your Python code. This will help you create performant and portable models, and it is required to use `SavedModel`.

This guide will help you conceptualize how `tf.function` works under the hood, so you can use it effectively.

The main takeaways and recommendations are:

- Debug in eager mode, then decorate with `@tf.function`.
- Don't rely on Python side effects like object mutation or list appends.
- `tf.function` works best with TensorFlow ops; NumPy and Python calls are converted to constants.

## Setup

In [ ]:

```
import tensorflow as tf
```

Define a helper function to demonstrate the kinds of errors you might encounter:

In [ ]:

```
import traceback
import contextlib

# Some helper code to demonstrate the kinds of errors you might encounter.
@contextlib.contextmanager
def assert_raises(error_class):
    try:
        yield
    except error_class as e:
        print('Caught expected exception \n  {}'.format(error_class))
        traceback.print_exc(limit=2)
    except Exception as e:
        raise e
    else:
        raise Exception('Expected {} to be raised but no error was raised!'.format(
            error_class))
```

## Basics

### Usage

A `Function` you define (for example by applying the `@tf.function` decorator) is just like a core TensorFlow operation: You can execute it eagerly; you can compute gradients; and so on.

In [ ]:

```
@tf.function # The decorator converts `add` into a `Function`.
def add(a, b):
    return a + b

add(tf.ones([2, 2]), tf.ones([2, 2])) # [[2., 2.], [2., 2.]]
```

In [ ]:

```
v = tf.Variable(1.0)
with tf.GradientTape() as tape:
    result = add(v, 1.0)
tape.gradient(result, v)
```

You can use `Function`s inside other `Function`s.

In [ ]:

```
@tf.function
def dense_layer(x, w, b):
    return add(tf.matmul(x, w), b)

dense_layer(tf.ones([3, 2]), tf.ones([2, 2]), tf.ones([2]))
```

`Function`s can be faster than eager code, especially for graphs with many small ops. But for graphs with a few expensive ops (like convolutions), you may not see much speedup.

In [ ]:

```
import timeit
conv_layer = tf.keras.layers.Conv2D(100, 3)

@tf.function
def conv_fn(image):
    return conv_layer(image)

image = tf.zeros([1, 200, 200, 100])
# Warm up
conv_layer(image); conv_fn(image)
print("Eager conv:", timeit.timeit(lambda: conv_layer(image), number=10))
print("Function conv:", timeit.timeit(lambda: conv_fn(image), number=10))
print("Note how there's not much difference in performance for convolutions")
```

### Tracing

This section exposes how `Function` works under the hood, including implementation details *which may change in the future*. However, once you understand why and when tracing happens, it's much easier to use `tf.function` effectively!

## What is "tracing"?

A `Function` runs your program in a [TensorFlow Graph](https://www.tensorflow.org/guide/intro_to_graphs#what_are_graphs) ([https://www.tensorflow.org/guide/intro\\_to\\_graphs#what\\_are\\_graphs](https://www.tensorflow.org/guide/intro_to_graphs#what_are_graphs)). However, a `tf.Graph` cannot represent all the things that you'd write in an eager TensorFlow program. For instance, Python supports polymorphism, but `tf.Graph` requires its inputs to have a specified data type and dimension. Or you may perform side tasks like reading command-line arguments, raising an error, or working with a more complex Python object; none of these things can run in a `tf.Graph`.

`Function` bridges this gap by separating your code in two stages:

1) In the first stage, referred to as "**tracing**", `Function` creates a new `tf.Graph`. Python code runs normally, but all TensorFlow operations (like adding two `Tensors`) are *deferred*: they are captured by the `tf.Graph` and not run.

2) In the second stage, a `tf.Graph` which contains everything that was deferred in the first stage is run. This stage is much faster than the tracing stage.

Depending on its inputs, `Function` will not always run the first stage when it is called. See "[Rules of tracing](#)" below to get a better sense of how it makes that determination. Skipping the first stage and only executing the second stage is what gives you TensorFlow's high performance.

When `Function` does decide to trace, the tracing stage is immediately followed by the second stage, so calling the `Function` both creates and runs the `tf.Graph`. Later you will see how you can run only the tracing stage with [`get\_concrete\_function`](#).

When you pass arguments of different types into a `Function`, both stages are run:

In [ ]:

```
@tf.function
def double(a):
    print("Tracing with", a)
    return a + a

print(double(tf.constant(1)))
print()
print(double(tf.constant(1.1)))
print()
print(double(tf.constant("a")))
print()
```

Note that if you repeatedly call a `Function` with the same argument type, TensorFlow will skip the tracing stage and reuse a previously traced graph, as the generated graph would be identical.

In [ ]:

```
# This doesn't print 'Tracing with ...'
print(double(tf.constant("b")))
```

You can use `pretty_printed_concrete_signatures()` to see all of the available traces:

In [ ]:

```
print(double.pretty_printed_concrete_signatures())
```

So far, you've seen that `tf.function` creates a cached, dynamic dispatch layer over TensorFlow's graph tracing logic. To be more specific about the terminology:

- A `tf.Graph` is the raw, language-agnostic, portable representation of a TensorFlow computation.
- A `ConcreteFunction` wraps a `tf.Graph`.
- A `Function` manages a cache of `ConcreteFunction`s and picks the right one for your inputs.
- `tf.function` wraps a Python function, returning a `Function` object.
- **Tracing** creates a `tf.Graph` and wraps it in a `ConcreteFunction`, also known as a **trace**.

## Rules of tracing

When called, a `Function` matches the call arguments to existing `ConcreteFunction`s using `tf.types.experimental.TraceType` of each argument. If a matching `ConcreteFunction` is found, the call is dispatched to it. If no match is found, a new `ConcreteFunction` is traced.

If multiple matches are found, the most specific signature is chosen. Matching is done by [subtyping](https://en.wikipedia.org/wiki/Subtyping) (<https://en.wikipedia.org/wiki/Subtyping>), much like normal function calls in C++ or Java, for instance. For example, `TensorShape([1, 2])` is a subtype of `TensorShape([None, None])` and so a call to the `tf.function` with `TensorShape([1, 2])` can be dispatched to the `ConcreteFunction` produced with `TensorShape([None, None])` but if a `ConcreteFunction` with `TensorShape([1, None])` also exists then it will be prioritized since it is more specific.

The `TraceType` is determined from input arguments as follows:

- For `Tensor`, the type is parameterized by the `Tensor`'s `dtype` and `shape`; ranked shapes are a subtype of unranked shapes; fixed dimensions are a subtype of unknown dimensions
- For `Variable`, the type is similar to `Tensor`, but also includes a unique resource ID of the variable, necessary to correctly wire control dependencies
- For Python primitive values, the type corresponds to the `value` itself. For example, the `TraceType` of the value `3` is `LiteralTraceType<3>`, not `int`.
- For Python ordered containers such as `list` and `tuple`, etc., the type is parameterized by the types of their elements; for example, the type of `[1, 2]` is `ListTraceType<LiteralTraceType<1>, LiteralTraceType<2>>` and the type for `[2, 1]` is `ListTraceType<LiteralTraceType<2>, LiteralTraceType<1>>` which is different.
- For Python mappings such as `dict`, the type is also a mapping from the same keys but to the types of values instead of the actual values. For example, the type of `{1: 2, 3: 4}`, is `MappingTraceType<<KeyValue<1, LiteralTraceType<2>>, <KeyValue<3, LiteralTraceType<4>>>`. However, unlike ordered containers, `{1: 2, 3: 4}` and `{3: 4, 1: 2}` have equivalent types.
- For Python objects which implement the `__tf_tracing_type__` method, the type is whatever that method returns
- For any other Python objects, the type is a generic `TraceType` which uses the object's Python equality and hashing for matching. (Note: It relies on [weakref](https://docs.python.org/3/library/weakref.html) (<https://docs.python.org/3/library/weakref.html>) to the object and hence only works as long as the object is in scope/not deleted.)

Note: `TraceType` is based on the `Function` input parameters so changes to global and [free variables](#)

(<https://docs.python.org/3/reference/executionmodel.html#binding-of-names>) alone will not create a new trace. See [this section](#) for recommended practices when dealing with Python global and free variables.

## Controlling retracing

Retracing, which is when your `Function` creates more than one trace, helps ensure that TensorFlow generates correct graphs for each set of inputs. However, tracing is an expensive operation! If your `Function` retraces a new graph for every call, you'll find that your code executes more slowly than if you didn't use `tf.function`.

To control the tracing behavior, you can use the following techniques:

### Pass a fixed `input_signature` to `tf.function`

In [ ]:

```
@tf.function(input_signature=(tf.TensorSpec(shape=[None], dtype=tf.int32),))
def next_collatz(x):
    print("Tracing with", x)
    return tf.where(x % 2 == 0, x // 2, 3 * x + 1)

print(next_collatz(tf.constant([1, 2])))
# You specified a 1-D tensor in the input signature, so this should fail.
with assert_raises(ValueError):
    next_collatz(tf.constant([[1, 2], [3, 4]]))

# You specified an int32 dtype in the input signature, so this should fail.
with assert_raises(ValueError):
    next_collatz(tf.constant([1.0, 2.0]))
```

### Use unknown dimensions for flexibility

Since TensorFlow matches tensors based on their shape, using a `None` dimension as a wildcard will allow `Function`s to reuse traces for variably-sized input. Variably-sized input can occur if you have sequences of different length, or images of different sizes for each batch (See the [Transformer](#) ([..../tutorials/text/transformer.ipynb](#)) and [Deep Dream](#) ([..../tutorials/generative/deepdream.ipynb](#)) tutorials for example).

In [ ]:

```
@tf.function(input_signature=(tf.TensorSpec(shape=[None], dtype=tf.int32),))
def g(x):
    print('Tracing with', x)
    return x

# No retrace!
print(g(tf.constant([1, 2, 3])))
print(g(tf.constant([1, 2, 3, 4, 5])))
```

### Pass tensors instead of python literals

Often, Python arguments are used to control hyperparameters and graph constructions - for example, `num_layers=10` or `training=True` or `nonlinearity='relu'`. So, if the Python argument changes, it makes sense that you'd have to retrace the graph.

However, it's possible that a Python argument is not being used to control graph construction. In these cases, a change in the Python value can trigger needless retracing. Take, for example, this training loop, which AutoGraph will dynamically unroll. Despite the multiple traces, the generated graph is actually identical, so retracing is unnecessary.

In [ ]:

```
def train_one_step():
    pass

@tf.function
def train(num_steps):
    print("Tracing with num_steps = ", num_steps)
    tf.print("Executing with num_steps = ", num_steps)
    for _ in tf.range(num_steps):
        train_one_step()

    print("Retracing occurs for different Python arguments.")
train(num_steps=10)
train(num_steps=20)

print()
print("Traces are reused for Tensor arguments.")
train(num_steps=tf.constant(10))
train(num_steps=tf.constant(20))
```

If you need to force retracing, create a new `Function`. Separate `Function` objects are guaranteed not to share traces.

In [ ]:

```
def f():
    print('Tracing!')
    tf.print('Executing')

tf.function(f)()
tf.function(f)()
```

### Use the tracing protocol

Where possible, you should prefer converting the Python type into a `tf.experimental.ExtensionType` ([www.tensorflow.org/api\\_docs/python/tf/experimental/ExtensionType](https://www.tensorflow.org/api_docs/python/tf/experimental/ExtensionType)) instead. Moreover, the `TraceType` of an `ExtensionType` is the `tf.TypeSpec` associated with it. Therefore, if needed, you can simply `override` ([https://www.tensorflow.org/guide/extension\\_type#customizing\\_the\\_extensiontypes\\_typespec](https://www.tensorflow.org/guide/extension_type#customizing_the_extensiontypes_typespec)) the default `tf.TypeSpec` to take control of an `ExtensionType`'s Tracing Protocol .

Otherwise, for direct control over when `Function` should retrace in regards to a particular Python type, you can implement the `Tracing Protocol` for it yourself.

In [ ]:

```
@tf.function
def get_mixed_flavor(fruit_a, fruit_b):
    return fruit_a.flavor + fruit_b.flavor

class Fruit:
    flavor = tf.constant([0, 0])

class Apple(Fruit):
    flavor = tf.constant([1, 2])

class Mango(Fruit):
    flavor = tf.constant([3, 4])

# As described in the above rules, a generic TraceType for `Apple` and `Mango`
# is generated (and a corresponding ConcreteFunction is traced) but it fails to
# match the second function call since the first pair of Apple() and Mango()
# have gone out of scope by then and deleted.
get_mixed_flavor(Apple(), Mango()) # Traces a new concrete function
get_mixed_flavor(Apple(), Mango()) # Traces a new concrete function again

# However, we, as the designers of the `Fruit` class, know that each subclass
# has a fixed flavor and we can reuse an existing traced concrete function if
# it was the same subclass. Avoiding such unnecessary tracing of concrete
# functions can have significant performance benefits.

class FruitTraceType(tf.types.experimental.TraceType):
    def __init__(self, fruit_type):
        self.fruit_type = fruit_type

    def is_subtype_of(self, other):
        return (type(other) is FruitTraceType and
                self.fruit_type is other.fruit_type)

    def most_specific_common_supertype(self, others):
        return self if all(self == other for other in others) else None

    def __eq__(self, other):
        return type(other) is FruitTraceType and self.fruit_type == other.fruit_type

    def __hash__(self):
        return hash(self.fruit_type)

class FruitWithTraceType:
    def __tf_tracing_type__(self, context):
        return FruitTraceType(type(self))

class AppleWithTraceType(FruitWithTraceType):
    flavor = tf.constant([1, 2])

class MangoWithTraceType(FruitWithTraceType):
    flavor = tf.constant([3, 4])

# Now if we try calling it again:
get_mixed_flavor(AppleWithTraceType(), MangoWithTraceType()) # Traces a new concrete function
get_mixed_flavor(AppleWithTraceType(), MangoWithTraceType()) # Re-uses the traced concrete function
```

## Obtaining concrete functions

Every time a function is traced, a new concrete function is created. You can directly obtain a concrete function, by using `get_concrete_function`.

In [ ]:

```
print("Obtaining concrete trace")
double_strings = double.get_concrete_function(tf.constant("a"))
print("Executing traced function")
print(double_strings(tf.constant("a")))
print(double_strings(a=tf.constant("b")))
```

In [ ]:

```
# You can also call get_concrete_function on an InputSpec
double_strings_from_inputspec = double.get_concrete_function(tf.TensorSpec(shape=[], dtype=tf.string))
print(double_strings_from_inputspec(tf.constant("c")))
```

Printing a `ConcreteFunction` displays a summary of its input arguments (with types) and its output type.

```
In [ ]:
```

```
print(double_strings)
```

You can also directly retrieve a concrete function's signature.

```
In [ ]:
```

```
print(double_strings.structured_input_signature)
print(double_strings.structured_outputs)
```

Using a concrete trace with incompatible types will throw an error

```
In [ ]:
```

```
with assert_raises(tf.errors.InvalidArgumentError):
    double_strings(tf.constant(1))
```

You may notice that Python arguments are given special treatment in a concrete function's input signature. Prior to TensorFlow 2.3, Python arguments were simply removed from the concrete function's signature. Starting with TensorFlow 2.3, Python arguments remain in the signature, but are constrained to take the value set during tracing.

```
In [ ]:
```

```
@tf.function
def pow(a, b):
    return a ** b

square = pow.get_concrete_function(a=tf.TensorSpec(None, tf.float32), b=2)
print(square)
```

```
In [ ]:
```

```
assert square(tf.constant(10.0)) == 100
with assert_raises(TypeError):
    square(tf.constant(10.0), b=3)
```

## Obtaining graphs

Each concrete function is a callable wrapper around a `tf.Graph`. Although retrieving the actual `tf.Graph` object is not something you'll normally need to do, you can obtain it easily from any concrete function.

```
In [ ]:
```

```
graph = double_strings.graph
for node in graph.as_graph_def().node:
    print(f'{node.input} -> {node.name}')
```

## Debugging

In general, debugging code is easier in eager mode than inside `tf.function`. You should ensure that your code executes error-free in eager mode before decorating with `tf.function`. To assist in the debugging process, you can call `tf.config.run_functions_eagerly(True)` to globally disable and reenable `tf.function`.

When tracking down issues that only appear within `tf.function`, here are some tips:

- Plain old Python `print` calls only execute during tracing, helping you track down when your function gets (re)traced.
- `tf.print` calls will execute every time, and can help you track down intermediate values during execution.
- `tf.debugging.enable_check_numerics` is an easy way to track down where NaNs and Inf are created.
- `pdb` (the [Python debugger \(<https://docs.python.org/3/library/pdb.html>\)](https://docs.python.org/3/library/pdb.html)) can help you understand what's going on during tracing. (Caveat: `pdb` will drop you into AutoGraph-transformed source code.)

## AutoGraph transformations

AutoGraph is a library that is on by default in `tf.function`, and transforms a subset of Python eager code into graph-compatible TensorFlow ops. This includes control flow like `if`, `for`, `while`.

TensorFlow ops like `tf.cond` and `tf.while_loop` continue to work, but control flow is often easier to write and understand when written in Python.

In [ ]:

```
# A simple loop

@tf.function
def f(x):
    while tf.reduce_sum(x) > 1:
        tf.print(x)
        x = tf.tanh(x)
    return x

f(tf.random.uniform([5]))
```

If you're curious you can inspect the code AutoGraph generates.

In [ ]:

```
print(tf.autograph.to_code(f.python_function))
```

## Conditionals

AutoGraph will convert some `if <condition>` statements into the equivalent `tf.cond` calls. This substitution is made if `<condition>` is a Tensor. Otherwise, the `if` statement is executed as a Python conditional.

A Python conditional executes during tracing, so exactly one branch of the conditional will be added to the graph. Without AutoGraph, this traced graph would be unable to take the alternate branch if there is data-dependent control flow.

`tf.cond` traces and adds both branches of the conditional to the graph, dynamically selecting a branch at execution time. Tracing can have unintended side effects; check out [AutoGraph tracing effects](#) ([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/autograph/g3doc/reference/control\\_flow.md#effects-of-the-tracing-process](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/autograph/g3doc/reference/control_flow.md#effects-of-the-tracing-process)) for more information.

In [ ]:

```
@tf.function
def fizzbuzz(n):
    for i in tf.range(1, n + 1):
        print('Tracing for loop')
        if i % 15 == 0:
            print('Tracing fizzbuzz branch')
            tf.print('fizzbuzz')
        elif i % 3 == 0:
            print('Tracing fizz branch')
            tf.print('fizz')
        elif i % 5 == 0:
            print('Tracing buzz branch')
            tf.print('buzz')
        else:
            print('Tracing default branch')
            tf.print(i)

fizzbuzz(tf.constant(5))
fizzbuzz(tf.constant(20))
```

See the [reference documentation](#) ([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/autograph/g3doc/reference/control\\_flow.md#if-statements](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/autograph/g3doc/reference/control_flow.md#if-statements)) for additional restrictions on AutoGraph-converted if statements.

## Loops

AutoGraph will convert some `for` and `while` statements into the equivalent TensorFlow looping ops, like `tf.while_loop`. If not converted, the `for` or `while` loop is executed as a Python loop.

This substitution is made in the following situations:

- `for x in y`: if `y` is a Tensor, convert to `tf.while_loop`. In the special case where `y` is a `tf.data.Dataset`, a combination of `tf.data.Dataset` ops are generated.
- `while <condition>`: if `<condition>` is a Tensor, convert to `tf.while_loop`.

A Python loop executes during tracing, adding additional ops to the `tf.Graph` for every iteration of the loop.

A TensorFlow loop traces the body of the loop, and dynamically selects how many iterations to run at execution time. The loop body only appears once in the generated `tf.Graph`.

See the [reference documentation](#)

([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/autograph/g3doc/reference/control\\_flow.md#while-statements](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/autograph/g3doc/reference/control_flow.md#while-statements)) for additional restrictions on AutoGraph-converted `for` and `while` statements.

## Looping over Python data

A common pitfall is to loop over Python/NumPy data within a `tf.function`. This loop will execute during the tracing process, adding a copy of your model to the `tf.Graph` for each iteration of the loop.

If you want to wrap the entire training loop in `tf.function`, the safest way to do this is to wrap your data as a `tf.data.Dataset` so that AutoGraph will dynamically unroll the training loop.

In [ ]:

```
def measure_graph_size(f, *args):
    g = f.get_concrete_function(*args).graph
    print("{}({}) contains {} nodes in its graph".format(
        f.__name__, ', '.join(map(str, args)), len(g.as_graph_def().node)))

@tf.function
def train(dataset):
    loss = tf.constant(0)
    for x, y in dataset:
        loss += tf.abs(y - x) # Some dummy computation.
    return loss

small_data = [(1, 1)] * 3
big_data = [(1, 1)] * 10
measure_graph_size(train, small_data)
measure_graph_size(train, big_data)

measure_graph_size(train, tf.data.Dataset.from_generator(
    lambda: small_data, (tf.int32, tf.int32)))
measure_graph_size(train, tf.data.Dataset.from_generator(
    lambda: big_data, (tf.int32, tf.int32)))
```

When wrapping Python/NumPy data in a Dataset, be mindful of `tf.data.Dataset.from_generator` versus `tf.data.Dataset.from_tensors`. The former will keep the data in Python and fetch it via `tf.py_function` which can have performance implications, whereas the latter will bundle a copy of the data as one large `tf.constant()` node in the graph, which can have memory implications.

Reading data from files via `TFRecordDataset`, `CsvDataset`, etc. is the most effective way to consume data, as then TensorFlow itself can manage the asynchronous loading and prefetching of data, without having to involve Python. To learn more, see the [tf.data : Build TensorFlow input pipelines](#) ([../guide/data](#)) guide.

## Accumulating values in a loop

A common pattern is to accumulate intermediate values from a loop. Normally, this is accomplished by appending to a Python list or adding entries to a Python dictionary. However, as these are Python side effects, they will not work as expected in a dynamically unrolled loop. Use `tf.TensorArray` to accumulate results from a dynamically unrolled loop.

In [ ]:

```
batch_size = 2
seq_len = 3
feature_size = 4

def rnn_step(inp, state):
    return inp + state

@tf.function
def dynamic_rnn(rnn_step, input_data, initial_state):
    # [batch, time, features] -> [time, batch, features]
    input_data = tf.transpose(input_data, [1, 0, 2])
    max_seq_len = input_data.shape[0]

    states = tf.TensorArray(tf.float32, size=max_seq_len)
    state = initial_state
    for i in tf.range(max_seq_len):
        state = rnn_step(input_data[i], state)
        states = states.write(i, state)
    return tf.transpose(states.stack(), [1, 0, 2])

dynamic_rnn(rnn_step,
            tf.random.uniform([batch_size, seq_len, feature_size]),
            tf.zeros([batch_size, feature_size]))
```

## Limitations

TensorFlow Function has a few limitations by design that you should be aware of when converting a Python function to a Function .

### Executing Python side effects

Side effects, like printing, appending to lists, and mutating globals, can behave unexpectedly inside a Function , sometimes executing twice or not at all. They only happen the first time you call a Function with a set of inputs. Afterwards, the traced `tf.Graph` is reexecuted, without executing the Python code.

The general rule of thumb is to avoid relying on Python side effects in your logic and only use them to debug your traces. Otherwise, TensorFlow APIs like `tf.data` , `tf.print` , `tf.summary` , `tf.Variable.assign` , and `tf.TensorArray` are the best way to ensure your code will be executed by the TensorFlow runtime with each call.

In [ ]:

```
@tf.function
def f(x):
    print("Traced with", x)
    tf.print("Executed with", x)

f(1)
f(1)
f(2)
```

If you would like to execute Python code during each invocation of a Function , `tf.py_function` is an exit hatch. The drawback of `tf.py_function` is that it's not portable or particularly performant, cannot be saved with SavedModel, and does not work well in distributed (multi-GPU, TPU) setups. Also, since `tf.py_function` has to be wired into the graph, it casts all inputs/outputs to tensors.

### Changing Python global and free variables

Changing Python global and free variables (<https://docs.python.org/3/reference/executionmodel.html#binding-of-names>) counts as a Python side effect, so it only happens during tracing.

In [ ]:

```
external_list = []

@tf.function
def side_effect(x):
    print('Python side effect')
    external_list.append(x)

side_effect(1)
side_effect(1)
side_effect(1)
# The list append only happened once!
assert len(external_list) == 1
```

Sometimes unexpected behaviors are very hard to notice. In the example below, the `counter` is intended to safeguard the increment of a variable. However because it is a python integer and not a TensorFlow object, its value is captured during the first trace. When the `tf.function` is used, the `assign_add` will be recorded unconditionally in the underlying graph. Therefore `v` will increase by 1, every time the `tf.function` is called. This issue is common among users that try to migrate their Graph-mode Tensorflow code to Tensorflow 2 using `tf.function` decorators, when python side-effects (the `counter` in the example) are used to determine what ops to run (`assign_add` in the example). Usually, users realize this only after seeing suspicious numerical results, or significantly lower performance than expected (e.g. if the guarded operation is very costly).

In [ ]:

```
class Model(tf.Module):
    def __init__(self):
        self.v = tf.Variable(0)
        self.counter = 0

    @tf.function
    def __call__(self):
        if self.counter == 0:
            # A python side-effect
            self.counter += 1
            self.v.assign_add(1)

    return self.v

m = Model()
for n in range(3):
    print(m().numpy()) # prints 1, 2, 3
```

A workaround to achieve the expected behavior is using `tf.init_scope` ([https://www.tensorflow.org/api\\_docs/python/tf/init\\_scope](https://www.tensorflow.org/api_docs/python/tf/init_scope)) to lift the operations outside of the function graph. This ensures that the variable increment is only done once during tracing time. It should be noted `init_scope` has other side effects including cleared control flow and gradient tape. Sometimes the usage of `init_scope` can become too complex to manage realistically.

In [ ]:

```
class Model(tf.Module):
    def __init__(self):
        self.v = tf.Variable(0)
        self.counter = 0

    @tf.function
    def __call__(self):
        if self.counter == 0:
            # Lifts ops out of function-building graphs
            with tf.init_scope():
                self.counter += 1
                self.v.assign_add(1)

    return self.v

m = Model()
for n in range(3):
    print(m().numpy()) # prints 1, 1, 1
```

In summary, as a rule of thumb, you should avoid mutating python objects such as integers or containers like lists that live outside the `Function`. Instead, use arguments and TF objects. For example, the section "[Accumulating values in a loop](#)" has one example of how list-like operations can be implemented.

You can, in some cases, capture and manipulate state if it is a `tf.Variable` (<https://www.tensorflow.org/guide/variable>). This is how the weights of Keras models are updated with repeated calls to the same `ConcreteFunction`.

## Using Python iterators and generators

Many Python features, such as generators and iterators, rely on the Python runtime to keep track of state. In general, while these constructs work as expected in eager mode, they are examples of Python side effects and therefore only happen during tracing.

In [ ]:

```
@tf.function
def buggy_consume_next(iterator):
    tf.print("Value:", next(iterator))

iterator = iter([1, 2, 3])
buggy_consume_next(iterator)
# This reuses the first value from the iterator, rather than consuming the next value.
buggy_consume_next(iterator)
buggy_consume_next(iterator)
```

Just like how TensorFlow has a specialized `tf.TensorArray` for list constructs, it has a specialized `tf.data.Iterator` for iteration constructs. See the section on [AutoGraph transformations](#) for an overview. Also, the `tf.data` (<https://www.tensorflow.org/guide/data>) API can help implement generator patterns:

In [ ]:

```
@tf.function
def good_consume_next(iterator):
    # This is ok, iterator is a tf.data.Iterator
    tf.print("Value:", next(iterator))

ds = tf.data.Dataset.from_tensor_slices([1, 2, 3])
iterator = iter(ds)
good_consume_next(iterator)
good_consume_next(iterator)
good_consume_next(iterator)
```

## All outputs of a `tf.function` must be return values

With the exception of `tf.Variable`s, a `tf.function` must return all its outputs. Attempting to directly access any tensors from a function without going through return values causes "leaks".

For example, the function below "leaks" the tensor `a` through the Python global `x`:

In [ ]:

```
x = None

@tf.function
def leaky_function(a):
    global x
    x = a + 1 # Bad - leaks local tensor
    return a + 2

correct_a = leaky_function(tf.constant(1))

print(correct_a.numpy()) # Good - value obtained from function's returns
try:
    x.numpy() # Bad - tensor leaked from inside the function, cannot be used here
except AttributeError as expected:
    print(expected)
```

This is true even if the leaked value is also returned:

In [ ]:

```
@tf.function
def leaky_function(a):
    global x
    x = a + 1 # Bad - leaks local tensor
    return x # Good - uses local tensor

correct_a = leaky_function(tf.constant(1))

print(correct_a.numpy()) # Good - value obtained from function's returns
try:
    x.numpy() # Bad - tensor leaked from inside the function, cannot be used here
except AttributeError as expected:
    print(expected)

@tf.function
def captures_leaked_tensor(b):
    b += x # Bad - `x` is leaked from `leaky_function`
    return b

with assert_raises(TypeError):
    captures_leaked_tensor(tf.constant(2))
```

Usually, leaks such as these occur when you use Python statements or data structures. In addition to leaking inaccessible tensors, such statements are also likely wrong because they count as Python side effects, and are not guaranteed to execute at every function call.

Common ways to leak local tensors also include mutating an external Python collection, or an object:

In [ ]:

```
class MyClass:

    def __init__(self):
        self.field = None

external_list = []
external_object = MyClass()

def leaky_function():
    a = tf.constant(1)
    external_list.append(a) # Bad - leaks tensor
    external_object.field = a # Bad - leaks tensor
```

## Recursive tf.functions are not supported

Recursive Functions are not supported and could cause infinite loops. For example,

In [ ]:

```
@tf.function
def recursive_fn(n):
    if n > 0:
        return recursive_fn(n - 1)
    else:
        return 1

with assert_raises(Exception):
    recursive_fn(tf.constant(5)) # Bad - maximum recursion error.
```

Even if a recursive Function seems to work, the python function will be traced multiple times and could have performance implication. For example,

In [ ]:

```
@tf.function
def recursive_fn(n):
    if n > 0:
        print('tracing')
        return recursive_fn(n - 1)
    else:
        return 1

recursive_fn(5) # Warning - multiple tracings
```

## Known Issues

If your Function is not evaluating correctly, the error may be explained by these known issues which are planned to be fixed in the future.

## Depending on Python global and free variables

`Function` creates a new `ConcreteFunction` when called with a new value of a Python argument. However, it does not do that for the Python closure, globals, or nonlocals of that `Function`. If their value changes in between calls to the `Function`, the `Function` will still use the values they had when it was traced. This is different from how regular Python functions work.

For that reason, you should follow a functional programming style that uses arguments instead of closing over outer names.

In [ ]:

```
@tf.function
def buggy_add():
    return 1 + foo

@tf.function
def recommended_add(foo):
    return 1 + foo

foo = 1
print("Buggy:", buggy_add())
print("Correct:", recommended_add(foo))
```

In [ ]:

```
print("Updating the value of `foo` to 100!")
foo = 100
print("Buggy:", buggy_add()) # Did not change!
print("Correct:", recommended_add(foo))
```

Another way to update a global value, is to make it a `tf.Variable` and use the `Variable.assign` method instead.

In [ ]:

```
@tf.function
def variable_add():
    return 1 + foo

foo = tf.Variable(1)
print("Variable:", variable_add())
```

In [ ]:

```
print("Updating the value of `foo` to 100!")
foo.assign(100)
print("Variable:", variable_add())
```

## Depending on Python objects

The recommendation to pass Python objects as arguments into `tf.function` has a number of known issues, that are expected to be fixed in the future. In general, you can rely on consistent tracing if you use a Python primitive or `tf.nest`-compatible structure as an argument or pass in a *different* instance of an object into a `Function`. However, `Function` will *not* create a new trace when you pass **the same object and only change its attributes**.

In [ ]:

```
class SimpleModel(tf.Module):
    def __init__(self):
        # These values are *not* tf.Variables.
        self.bias = 0.
        self.weight = 2.

@tf.function
def evaluate(model, x):
    return model.weight * x + model.bias

simple_model = SimpleModel()
x = tf.constant(10.)
print(evaluate(simple_model, x))
```

In [ ]:

```
print("Adding bias!")
simple_model.bias += 5.0
print(evaluate(simple_model, x)) # Didn't change :(
```

Using the same `Function` to evaluate the updated instance of the model will be buggy since the updated model has the [same cache key](#) as the original model.

For that reason, you're recommended to write your `Function` to avoid depending on mutable object attributes or create new objects.

If that is not possible, one workaround is to make new `Function`s each time you modify your object to force retracing:

In [ ]:

```
def evaluate(model, x):
    return model.weight * x + model.bias

new_model = SimpleModel()
evaluate_no_bias = tf.function(evaluate).get_concrete_function(new_model, x)
# Don't pass in `new_model`, `Function` already captured its state during tracing.
print(evaluate_no_bias(x))
```

In [ ]:

```
print("Adding bias!")
new_model.bias += 5.0
# Create new Function and ConcreteFunction since you modified new_model.
evaluate_with_bias = tf.function(evaluate).get_concrete_function(new_model, x)
print(evaluate_with_bias(x)) # Don't pass in `new_model`.
```

As [retracing can be expensive](#) ([https://www.tensorflow.org/guide/intro\\_to\\_graphs#tracing\\_and\\_performance](https://www.tensorflow.org/guide/intro_to_graphs#tracing_and_performance)), you can use `tf.Variable`s as object attributes, which can be mutated (but not changed, careful!) for a similar effect without needing a retrace.

In [ ]:

```
class BetterModel:

    def __init__(self):
        self.bias = tf.Variable(0.)
        self.weight = tf.Variable(2.)

    @tf.function
    def evaluate(model, x):
        return model.weight * x + model.bias

better_model = BetterModel()
print(evaluate(better_model, x))
```

In [ ]:

```
print("Adding bias!")
better_model.bias.assign_add(5.0) # Note: instead of better_model.bias += 5
print(evaluate(better_model, x)) # This works!
```

## Creating `tf.Variables`

`Function` only supports singleton `tf.Variable`s created once on the first call, and reused across subsequent function calls. The code snippet below would create a new `tf.Variable` in every function call, which results in a `ValueError` exception.

Example:

In [ ]:

```
@tf.function
def f(x):
    v = tf.Variable(1.0)
    return v

with assert_raises(ValueError):
    f(1.0)
```

A common pattern used to work around this limitation is to start with a Python `None` value, then conditionally create the `tf.Variable` if the value is `None`:

In [ ]:

```
class Count(tf.Module):
    def __init__(self):
        self.count = None

    @tf.function
    def __call__(self):
        if self.count is None:
            self.count = tf.Variable(0)
        return self.count.assign_add(1)

c = Count()
print(c())
print(c())
```

### Using with multiple Keras optimizers

You may encounter `ValueError: tf.function only supports singleton tf.Variables created on the first call.` when using more than one Keras optimizer with a `tf.function`. This error occurs because optimizers internally create `tf.Variables` when they apply gradients for the first time.

In [ ]:

```
opt1 = tf.keras.optimizers.Adam(learning_rate = 1e-2)
opt2 = tf.keras.optimizers.Adam(learning_rate = 1e-3)

@tf.function
def train_step(w, x, y, optimizer):
    with tf.GradientTape() as tape:
        L = tf.reduce_sum(tf.square(w*x - y))
    gradients = tape.gradient(L, [w])
    optimizer.apply_gradients(zip(gradients, [w]))

w = tf.Variable(2.)
x = tf.constant([-1.])
y = tf.constant([2.])

train_step(w, x, y, opt1)
print("Calling `train_step` with different optimizer...")
with assert_raises(ValueError):
    train_step(w, x, y, opt2)
```

If you need to change the optimizer during training, a workaround is to create a new `Function` for each optimizer, calling the `ConcreteFunction` directly.

In [ ]:

```
opt1 = tf.keras.optimizers.Adam(learning_rate = 1e-2)
opt2 = tf.keras.optimizers.Adam(learning_rate = 1e-3)

# Not a tf.function.
def train_step(w, x, y, optimizer):
    with tf.GradientTape() as tape:
        L = tf.reduce_sum(tf.square(w*x - y))
    gradients = tape.gradient(L, [w])
    optimizer.apply_gradients(zip(gradients, [w]))

w = tf.Variable(2.)
x = tf.constant([-1.])
y = tf.constant([2.])

# Make a new Function and ConcreteFunction for each optimizer.
train_step_1 = tf.function(train_step).get_concrete_function(w, x, y, opt1)
train_step_2 = tf.function(train_step).get_concrete_function(w, x, y, opt2)
for i in range(10):
    if i % 2 == 0:
        train_step_1(w, x, y) # `opt1` is not used as a parameter.
    else:
        train_step_2(w, x, y) # `opt2` is not used as a parameter.
```

## Using with multiple Keras models

You may also encounter `ValueError: tf.function only supports singleton tf.Variables created on the first call.` when passing different model instances to the same `Function`.

This error occurs because Keras models (which [do not have their input shape defined](#) ([https://www.tensorflow.org/guide/keras/custom\\_layers\\_and\\_models#best\\_practice\\_deferring\\_weight\\_creation\\_until\\_the\\_shape\\_of\\_the\\_inputs\\_is\\_known](https://www.tensorflow.org/guide/keras/custom_layers_and_models#best_practice_deferring_weight_creation_until_the_shape_of_the_inputs_is_known))) and Keras layers create `tf.Variables`s when they are first called. You may be attempting to initialize those variables inside a `Function`, which has already been called. To avoid this error, try calling `model.build(input_shape)` to initialize all the weights before training the model.

## Further reading

To learn about how to export and load a `Function`, see the [SavedModel guide \(../../guide/saved\\_model\)](#). To learn more about graph optimizations that are performed after tracing, see the [Grappler guide \(../../guide/graph\\_optimization\)](#). To learn how to optimize your data pipeline and profile your model, see the [Profiler guide \(../../guide/profiler.md\)](#).

*Copyright 2020 The TensorFlow Authors.*

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Introduction to gradients and automatic differentiation



[View on TensorFlow.org](#)

(<https://www.tensorflow.org/guide/autodiff>) (<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/autodiff.ipynb>) (<https://github.com/tensorflow/docs>)



[Run in Google Colab](#)

## Automatic Differentiation and Gradients

[Automatic differentiation](#) ([https://en.wikipedia.org/wiki/Automatic\\_differentiation](https://en.wikipedia.org/wiki/Automatic_differentiation)) is useful for implementing machine learning algorithms such as [backpropagation](#) (<https://en.wikipedia.org/wiki/Backpropagation>) for training neural networks.

In this guide, you will explore ways to compute gradients with TensorFlow, especially in eager execution.

## Setup

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt

import tensorflow as tf
```

## Computing gradients

To differentiate automatically, TensorFlow needs to remember what operations happen in what order during the *forward* pass. Then, during the *backward* pass, TensorFlow traverses this list of operations in reverse order to compute gradients.

## Gradient tapes

TensorFlow provides the `tf.GradientTape` API for automatic differentiation; that is, computing the gradient of a computation with respect to some inputs, usually `tf.Variable`s. TensorFlow "records" relevant operations executed inside the context of a `tf.GradientTape` onto a "tape". TensorFlow then uses that tape to compute the gradients of a "recorded" computation using [reverse mode differentiation](https://en.wikipedia.org/wiki/Automatic_differentiation) ([https://en.wikipedia.org/wiki/Automatic\\_differentiation](https://en.wikipedia.org/wiki/Automatic_differentiation)).

Here is a simple example:

In [ ]:

```
x = tf.Variable(3.0)

with tf.GradientTape() as tape:
    y = x**2
```

Once you've recorded some operations, use `GradientTape.gradient(target, sources)` to calculate the gradient of some target (often a loss) relative to some source (often the model's variables):

In [ ]:

```
# dy = 2x * dx
dy_dx = tape.gradient(y, x)
dy_dx.numpy()
```

The above example uses scalars, but `tf.GradientTape` works as easily on any tensor:

In [ ]:

```
w = tf.Variable(tf.random.normal((3, 2)), name='w')
b = tf.Variable(tf.zeros(2, dtype=tf.float32), name='b')
x = [[1., 2., 3.]]
```

```
with tf.GradientTape(persistent=True) as tape:
    y = x @ w + b
    loss = tf.reduce_mean(y**2)
```

To get the gradient of `loss` with respect to both variables, you can pass both as sources to the `gradient` method. The tape is flexible about how sources are passed and will accept any nested combination of lists or dictionaries and return the gradient structured the same way (see `tf.nest` ).

In [ ]:

```
[dl_dw, dl_db] = tape.gradient(loss, [w, b])
```

The gradient with respect to each source has the shape of the source:

In [ ]:

```
print(w.shape)
print(dl_dw.shape)
```

Here is the gradient calculation again, this time passing a dictionary of variables:

In [ ]:

```
my_vars = {
    'w': w,
    'b': b
}

grad = tape.gradient(loss, my_vars)
grad['b']
```

## Gradients with respect to a model

It's common to collect `tf.Variables` into a `tf.Module` or one of its subclasses (`layers.Layer`, `keras.Model`) for [checkpointing \(checkpoint.ipynb\)](#) and [exporting \(saved\\_model.ipynb\)](#).

In most cases, you will want to calculate gradients with respect to a model's trainable variables. Since all subclasses of `tf.Module` aggregate their variables in the `Module.trainable_variables` property, you can calculate these gradients in a few lines of code:

In [ ]:

```
layer = tf.keras.layers.Dense(2, activation='relu')
x = tf.constant([[1., 2., 3.]])

with tf.GradientTape() as tape:
    # Forward pass
    y = layer(x)
    loss = tf.reduce_mean(y**2)

# Calculate gradients with respect to every trainable variable
grad = tape.gradient(loss, layer.trainable_variables)
```

In [ ]:

```
for var, g in zip(layer.trainable_variables, grad):
    print(f'{var.name}, shape: {g.shape}')
```

## Controlling what the tape watches

The default behavior is to record all operations after accessing a trainable `tf.Variable`. The reasons for this are:

- The tape needs to know which operations to record in the forward pass to calculate the gradients in the backwards pass.
- The tape holds references to intermediate outputs, so you don't want to record unnecessary operations.
- The most common use case involves calculating the gradient of a loss with respect to all a model's trainable variables.

For example, the following fails to calculate a gradient because the `tf.Tensor` is not "watched" by default, and the `tf.Variable` is not trainable:

In [ ]:

```
# A trainable variable
x0 = tf.Variable(3.0, name='x0')
# Not trainable
x1 = tf.Variable(3.0, name='x1', trainable=False)
# Not a Variable: A variable + tensor returns a tensor.
x2 = tf.Variable(2.0, name='x2') + 1.0
# Not a variable
x3 = tf.constant(3.0, name='x3')

with tf.GradientTape() as tape:
    y = (x0**2) + (x1**2) + (x2**2)

grad = tape.gradient(y, [x0, x1, x2, x3])

for g in grad:
    print(g)
```

You can list the variables being watched by the tape using the `GradientTape.watched_variables` method:

In [ ]:

```
[var.name for var in tape.watched_variables()]
```

`tf.GradientTape` provides hooks that give the user control over what is or is not watched.

To record gradients with respect to a `tf.Tensor`, you need to call `GradientTape.watch(x)`:

In [ ]:

```
x = tf.constant(3.0)
with tf.GradientTape() as tape:
    tape.watch(x)
    y = x**2

# dy = 2x * dx
dy_dx = tape.gradient(y, x)
print(dy_dx.numpy())
```

Conversely, to disable the default behavior of watching all `tf.Variables`, set `watch_accessed_variables=False` when creating the gradient tape. This calculation uses two variables, but only connects the gradient for one of the variables:

In [ ]:

```
x0 = tf.Variable(0.0)
x1 = tf.Variable(10.0)

with tf.GradientTape(watch_accessed_variables=False) as tape:
    tape.watch(x1)
    y0 = tf.math.sin(x0)
    y1 = tf.nn.softplus(x1)
    y = y0 + y1
    ys = tf.reduce_sum(y)
```

Since `GradientTape.watch` was not called on `x0`, no gradient is computed with respect to it:

In [ ]:

```
# dys/dx1 = exp(x1) / (1 + exp(x1)) = sigmoid(x1)
grad = tape.gradient(ys, {'x0': x0, 'x1': x1})

print('dy/dx0:', grad['x0'])
print('dy/dx1:', grad['x1'].numpy())
```

## Intermediate results

You can also request gradients of the output with respect to intermediate values computed inside the `tf.GradientTape` context.

In [ ]:

```
x = tf.constant(3.0)

with tf.GradientTape() as tape:
    tape.watch(x)
    y = x * x
    z = y * y

# Use the tape to compute the gradient of z with respect to the
# intermediate value y.
# dz_dy = 2 * y and y = x ** 2 = 9
print(tape.gradient(z, y).numpy())
```

By default, the resources held by a `GradientTape` are released as soon as the `GradientTape.gradient` method is called. To compute multiple gradients over the same computation, create a gradient tape with `persistent=True`. This allows multiple calls to the `gradient` method as resources are released when the tape object is garbage collected. For example:

In [ ]:

```
x = tf.constant([1, 3.0])
with tf.GradientTape(persistent=True) as tape:
    tape.watch(x)
    y = x * x
    z = y * y

print(tape.gradient(z, x).numpy()) # [4.0, 108.0] (4 * x**3 at x = [1.0, 3.0])
print(tape.gradient(y, x).numpy()) # [2.0, 6.0] (2 * x at x = [1.0, 3.0])
```

In [ ]:

```
del tape # Drop the reference to the tape
```

## Notes on performance

- There is a tiny overhead associated with doing operations inside a gradient tape context. For most eager execution this will not be a noticeable cost, but you should still use tape context around the areas only where it is required.
- Gradient tapes use memory to store intermediate results, including inputs and outputs, for use during the backwards pass.

For efficiency, some ops (like `ReLU`) don't need to keep their intermediate results and they are pruned during the forward pass. However, if you use `persistent=True` on your tape, *nothing is discarded* and your peak memory usage will be higher.

## Gradients of non-scalar targets

A gradient is fundamentally an operation on a scalar.

In [ ]:

```
x = tf.Variable(2.0)
with tf.GradientTape(persistent=True) as tape:
    y0 = x**2
    y1 = 1 / x

print(tape.gradient(y0, x).numpy())
print(tape.gradient(y1, x).numpy())
```

Thus, if you ask for the gradient of multiple targets, the result for each source is:

- The gradient of the sum of the targets, or equivalently
- The sum of the gradients of each target.

In [ ]:

```
x = tf.Variable(2.0)
with tf.GradientTape() as tape:
    y0 = x**2
    y1 = 1 / x

print(tape.gradient({'y0': y0, 'y1': y1}, x).numpy())
```

Similarly, if the target(s) are not scalar the gradient of the sum is calculated:

In [ ]:

```
x = tf.Variable(2.)

with tf.GradientTape() as tape:
    y = x * [3., 4.]

print(tape.gradient(y, x).numpy())
```

This makes it simple to take the gradient of the sum of a collection of losses, or the gradient of the sum of an element-wise loss calculation.

If you need a separate gradient for each item, refer to [Jacobians \(advanced\\_autodiff.ipynb#Jacobians\)](#).

In some cases you can skip the Jacobian. For an element-wise calculation, the gradient of the sum gives the derivative of each element with respect to its input-element, since each element is independent:

In [ ]:

```
x = tf.linspace(-10.0, 10.0, 200+1)

with tf.GradientTape() as tape:
    tape.watch(x)
    y = tf.nn.sigmoid(x)

dy_dx = tape.gradient(y, x)
```

In [ ]:

```
plt.plot(x, y, label='y')
plt.plot(x, dy_dx, label='dy/dx')
plt.legend()
_ = plt.xlabel('x')
```

## Control flow

Because a gradient tape records operations as they are executed, Python control flow is naturally handled (for example, `if` and `while` statements).

Here a different variable is used on each branch of an `if`. The gradient only connects to the variable that was used:

In [ ]:

```
x = tf.constant(1.0)
v0 = tf.Variable(2.0)
v1 = tf.Variable(2.0)

with tf.GradientTape(persistent=True) as tape:
    tape.watch(x)
    if x > 0.0:
        result = v0
    else:
        result = v1**2

dv0, dv1 = tape.gradient(result, [v0, v1])

print(dv0)
print(dv1)
```

Just remember that the control statements themselves are not differentiable, so they are invisible to gradient-based optimizers.

Depending on the value of `x` in the above example, the tape either records `result = v0` or `result = v1**2`. The gradient with respect to `x` is always `None`.

In [ ]:

```
dx = tape.gradient(result, x)

print(dx)
```

## Getting a gradient of `None`

When a target is not connected to a source you will get a gradient of `None`.

In [ ]:

```
x = tf.Variable(2.)
y = tf.Variable(3.)

with tf.GradientTape() as tape:
    z = y * y
print(tape.gradient(z, x))
```

Here `z` is obviously not connected to `x`, but there are several less-obvious ways that a gradient can be disconnected.

### 1. Replaced a variable with a tensor

In the section on ["controlling what the tape watches"](#) you saw that the tape will automatically watch a `tf.Variable` but not a `tf.Tensor`.

One common error is to inadvertently replace a `tf.Variable` with a `tf.Tensor`, instead of using `Variable.assign` to update the `tf.Variable`. Here is an example:

In [ ]:

```
x = tf.Variable(2.0)

for epoch in range(2):
    with tf.GradientTape() as tape:
        y = x+1

    print(type(x).__name__, ":", tape.gradient(y, x))
    x = x + 1 # This should be `x.assign_add(1)`
```

### 2. Did calculations outside of TensorFlow

The tape can't record the gradient path if the calculation exits TensorFlow. For example:

In [ ]:

```
x = tf.Variable([[1.0, 2.0],  
                 [3.0, 4.0]], dtype=tf.float32)  
  
with tf.GradientTape() as tape:  
    x2 = x**2  
  
    # This step is calculated with NumPy  
    y = np.mean(x2, axis=0)  
  
    # Like most ops, reduce_mean will cast the NumPy array to a constant tensor  
    # using `tf.convert_to_tensor`.  
    y = tf.reduce_mean(y, axis=0)  
  
print(tape.gradient(y, x))
```

### 3. Took gradients through an integer or string

Integers and strings are not differentiable. If a calculation path uses these data types there will be no gradient.

Nobody expects strings to be differentiable, but it's easy to accidentally create an `int` constant or variable if you don't specify the `dtype`.

In [ ]:

```
x = tf.constant(10)  
  
with tf.GradientTape() as g:  
    g.watch(x)  
    y = x * x  
  
print(g.gradient(y, x))
```

TensorFlow doesn't automatically cast between types, so, in practice, you'll often get a type error instead of a missing gradient.

### 4. Took gradients through a stateful object

State stops gradients. When you read from a stateful object, the tape can only observe the current state, not the history that lead to it.

A `tf.Tensor` is immutable. You can't change a tensor once it's created. It has a `value`, but no `state`. All the operations discussed so far are also stateless: the output of a `tf.matmul` only depends on its inputs.

A `tf.Variable` has internal state—its value. When you use the variable, the state is read. It's normal to calculate a gradient with respect to a variable, but the variable's state blocks gradient calculations from going farther back. For example:

In [ ]:

```
x0 = tf.Variable(3.0)  
x1 = tf.Variable(0.0)  
  
with tf.GradientTape() as tape:  
    # Update x1 = x1 + x0.  
    x1.assign_add(x0)  
    # The tape starts recording from x1.  
    y = x1**2    # y = (x1 + x0)**2  
  
    # This doesn't work.  
print(tape.gradient(y, x0))    #dy/dx0 = 2*(x1 + x0)
```

Similarly, `tf.data.Dataset` iterators and `tf.queue`s are stateful, and will stop all gradients on tensors that pass through them.

## No gradient registered

Some `tf.Operation`s are **registered as being non-differentiable** and will return `None`. Others have **no gradient registered**.

The `tf.raw_ops` page shows which low-level ops have gradients registered.

If you attempt to take a gradient through a float op that has no gradient registered the tape will throw an error instead of silently returning `None`. This way you know something has gone wrong.

For example, the `tf.image.adjust_contrast` function wraps `raw_ops.AdjustContrastv2`, which could have a gradient but the gradient is not implemented:

In [ ]:

```
image = tf.Variable([[[0.5, 0.0, 0.0]]])
delta = tf.Variable(0.1)

with tf.GradientTape() as tape:
    new_image = tf.image.adjust_contrast(image, delta)

try:
    print(tape.gradient(new_image, [image, delta]))
    assert False # This should not happen.
except LookupError as e:
    print(f'{type(e).__name__}: {e}')
```

If you need to differentiate through this op, you'll either need to implement the gradient and register it (using `tf.RegisterGradient`) or re-implement the function using other ops.

## Zeros instead of None

In some cases it would be convenient to get 0 instead of `None` for unconnected gradients. You can decide what to return when you have unconnected gradients using the `unconnected_gradients` argument:

In [ ]:

```
x = tf.Variable([2., 2.])
y = tf.Variable(3.)

with tf.GradientTape() as tape:
    z = y**2
print(tape.gradient(z, x, unconnected_gradients=tf.UnconnectedGradients.ZERO))
```

**Copyright 2019 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Better performance with the `tf.data` API



[View on TensorFlow.org](https://www.tensorflow.org/guide/data_performance)

([https://www.tensorflow.org/guide/data\\_performance](https://www.tensorflow.org/guide/data_performance))



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/data_performance.ipynb)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/data\\_performance.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/data_performance.ipynb)) ([https://github.com/tensorflow/docs/blob/master/site/en/guide/data\\_performance.ipynb](https://github.com/tensorflow/docs/blob/master/site/en/guide/data_performance.ipynb))

## Overview

GPUs and TPUs can radically reduce the time required to execute a single training step. Achieving peak performance requires an efficient input pipeline that delivers data for the next step before the current step has finished. The `tf.data` API helps to build flexible and efficient input pipelines. This document demonstrates how to use the `tf.data` API to build highly performant TensorFlow input pipelines.

Before you continue, check the [Build TensorFlow input pipelines \(./data.ipynb\)](#) guide to learn how to use the `tf.data` API.

## Resources

- [Build TensorFlow input pipelines \(./data.ipynb\)](#)
- `tf.data.Dataset` API
- [Analyze `tf.data` performance with the TF Profiler \(./data\\_performance\\_analysis.md\)](#)

## Setup

In [ ]:

```
import tensorflow as tf
import time
```

Throughout this guide, you will iterate across a dataset and measure the performance. Making reproducible performance benchmarks can be difficult. Different factors affecting reproducibility include:

- The current CPU load
- The network traffic
- Complex mechanisms, such as cache

To get a reproducible benchmark, you will build an artificial example.

## The dataset

Start with defining a class inheriting from `tf.data.Dataset` called `ArtificialDataset`. This dataset:

- Generates `num_samples` samples (default is 3)
- Sleeps for some time before the first item to simulate opening a file
- Sleeps for some time before producing each item to simulate reading data from a file

In [ ]:

```
class ArtificialDataset(tf.data.Dataset):
    def __generator__(num_samples):
        # Opening the file
        time.sleep(0.03)

        for sample_idx in range(num_samples):
            # Reading data (line, record) from the file
            time.sleep(0.015)

            yield (sample_idx,)

    def __new__(cls, num_samples=3):
        return tf.data.Dataset.from_generator(
            cls.__generator__,
            output_signature = tf.TensorSpec(shape = (1,), dtype = tf.int64),
            args=(num_samples,))
)
```

This dataset is similar to the `tf.data.Dataset.range` one, adding a fixed delay at the beginning of and in-between each sample.

## The training loop

Next, write a dummy training loop that measures how long it takes to iterate over a dataset. Training time is simulated.

In [ ]:

```
def benchmark(dataset, num_epochs=2):
    start_time = time.perf_counter()
    for epoch_num in range(num_epochs):
        for sample in dataset:
            # Performing a training step
            time.sleep(0.01)
    print("Execution time:", time.perf_counter() - start_time)
```

## Optimize performance

To exhibit how performance can be optimized, you will improve the performance of the `ArtificialDataset`.

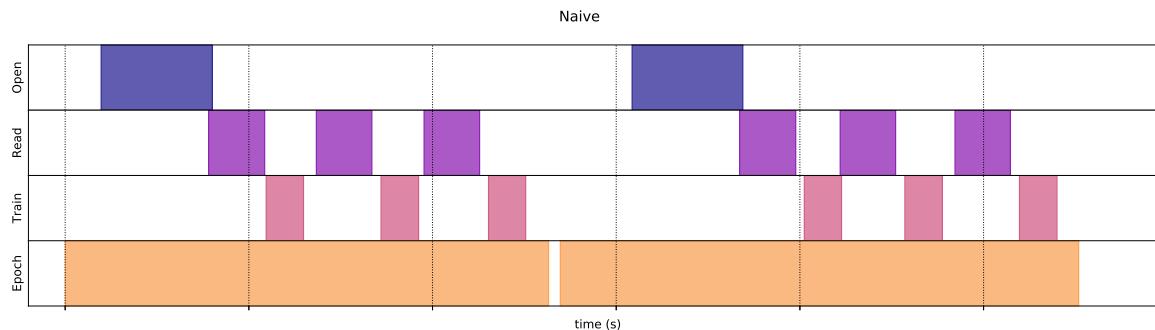
### The naive approach

Start with a naive pipeline using no tricks, iterating over the dataset as-is.

In [ ]:

```
benchmark(ArtificialDataset())
```

Under the hood, this is how your execution time was spent:



The plot shows that performing a training step involves:

- Opening a file if it hasn't been opened yet
- Fetching a data entry from the file
- Using the data for training

However, in a naive synchronous implementation like here, while your pipeline is fetching the data, your model is sitting idle. Conversely, while your model is training, the input pipeline is sitting idle. The training step time is thus the sum of opening, reading and training times.

The next sections build on this input pipeline, illustrating best practices for designing performant TensorFlow input pipelines.

## Prefetching

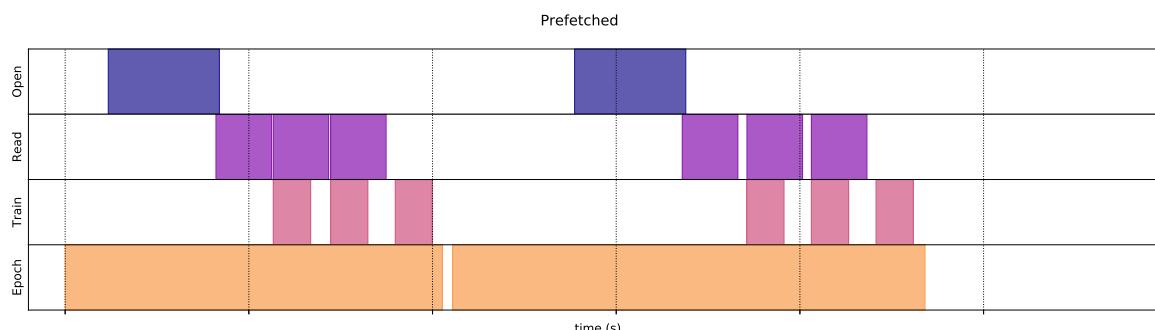
Prefetching overlaps the preprocessing and model execution of a training step. While the model is executing training step `s`, the input pipeline is reading the data for step `s+1`. Doing so reduces the step time to the maximum (as opposed to the sum) of the training and the time it takes to extract the data.

The `tf.data` API provides the `tf.data.Dataset.prefetch` transformation. It can be used to decouple the time when data is produced from the time when data is consumed. In particular, the transformation uses a background thread and an internal buffer to prefetch elements from the input dataset ahead of the time they are requested. The number of elements to prefetch should be equal to (or possibly greater than) the number of batches consumed by a single training step. You could either manually tune this value, or set it to `tf.data.AUTOTUNE`, which will prompt the `tf.data` runtime to tune the value dynamically at runtime.

Note that the prefetch transformation provides benefits any time there is an opportunity to overlap the work of a "producer" with the work of a "consumer."

In [ ]:

```
benchmark(  
    ArtificialDataset()  
    .prefetch(tf.data.AUTOTUNE)  
)
```



Now, as the data execution time plot shows, while the training step is running for sample 0, the input pipeline is reading the data for the sample 1, and so on.

## Parallelizing data extraction

In a real-world setting, the input data may be stored remotely (for example, on Google Cloud Storage or HDFS). A dataset pipeline that works well when reading data locally might become bottlenecked on I/O when reading data remotely because of the following differences between local and remote storage:

- **Time-to-first-byte:** Reading the first byte of a file from remote storage can take orders of magnitude longer than from local storage.
- **Read throughput:** While remote storage typically offers large aggregate bandwidth, reading a single file might only be able to utilize a small fraction of this bandwidth.

In addition, once the raw bytes are loaded into memory, it may also be necessary to deserialize and/or decrypt the data (e.g. [protobuf](#) (<https://developers.google.com/protocol-buffers/>)), which requires additional computation. This overhead is present irrespective of whether the data is stored locally or remotely, but can be worse in the remote case if data is not prefetched effectively.

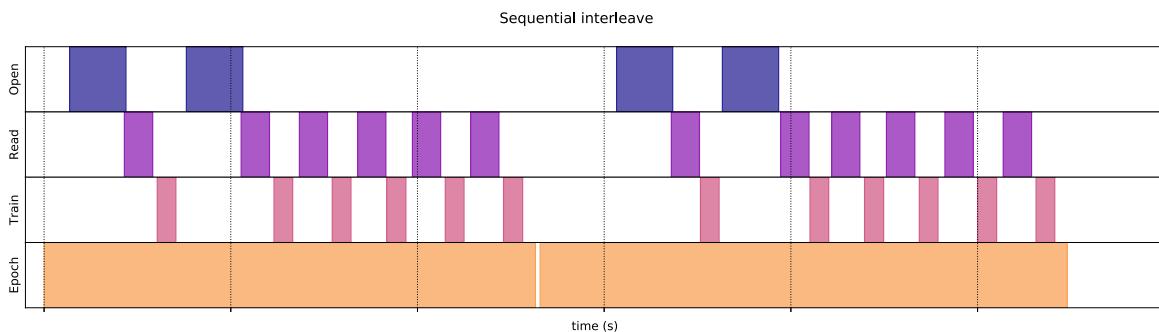
To mitigate the impact of the various data extraction overheads, the `tf.data.Dataset.interleave` transformation can be used to parallelize the data loading step, interleaving the contents of other datasets (such as data file readers). The number of datasets to overlap can be specified by the `cycle_length` argument, while the level of parallelism can be specified by the `num_parallel_calls` argument. Similar to the `prefetch` transformation, the `interleave` transformation supports `tf.data.AUTOTUNE`, which will delegate the decision about what level of parallelism to use to the `tf.data` runtime.

### Sequential interleave

The default arguments of the `tf.data.Dataset.interleave` transformation make it interleave single samples from two datasets sequentially.

In [ ]:

```
benchmark(  
    tf.data.Dataset.range(2)  
    .interleave(lambda _: ArtificialDataset())  
)
```



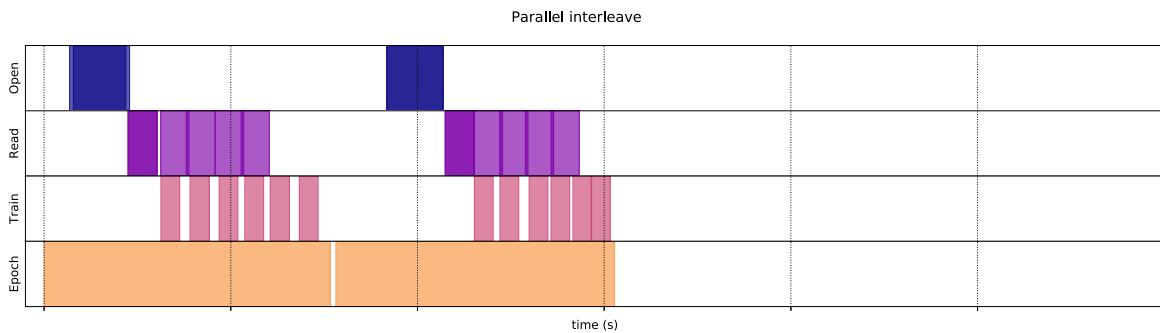
This data execution time plot allows to exhibit the behavior of the `interleave` transformation, fetching samples alternatively from the two datasets available. However, no performance improvement is involved here.

### Parallel interleave

Now, use the `num_parallel_calls` argument of the `interleave` transformation. This loads multiple datasets in parallel, reducing the time waiting for the files to be opened.

In [ ]:

```
benchmark(  
    tf.data.Dataset.range(2)  
    .interleave(  
        lambda _: ArtificialDataset(),  
        num_parallel_calls=tf.data.AUTOTUNE  
    )  
)
```



This time, as the data execution time plot shows, the reading of the two datasets is parallelized, reducing the global data processing time.

## Parallelizing data transformation

When preparing data, input elements may need to be pre-processed. To this end, the `tf.data` API offers the `tf.data.Dataset.map` transformation, which applies a user-defined function to each element of the input dataset. Because input elements are independent of one another, the pre-processing can be parallelized across multiple CPU cores. To make this possible, similarly to the `prefetch` and `interleave` transformations, the `map` transformation provides the `num_parallel_calls` argument to specify the level of parallelism.

Choosing the best value for the `num_parallel_calls` argument depends on your hardware, characteristics of your training data (such as its size and shape), the cost of your map function, and what other processing is happening on the CPU at the same time. A simple heuristic is to use the number of available CPU cores. However, as for the `prefetch` and `interleave` transformation, the `map` transformation supports `tf.data.AUTOTUNE` which will delegate the decision about what level of parallelism to use to the `tf.data` runtime.

In [ ]:

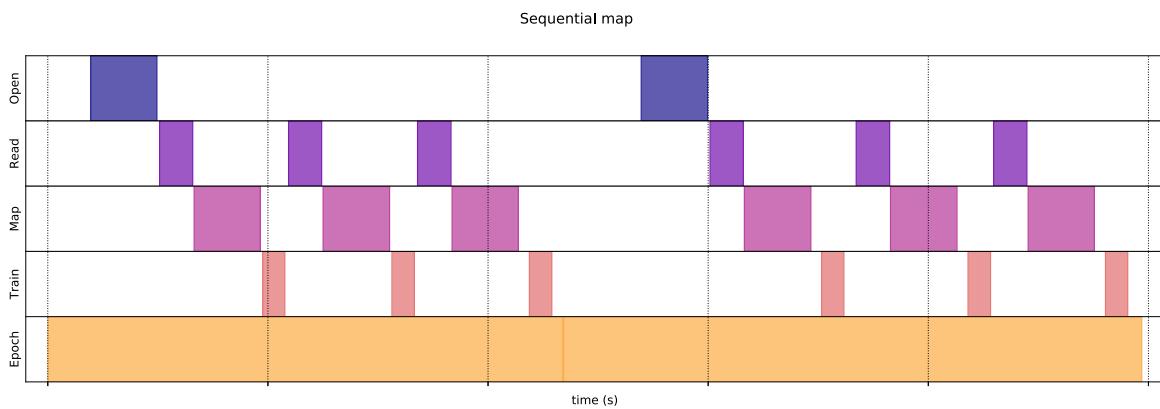
```
def mapped_function(s):
    # Do some hard pre-processing
    tf.py_function(lambda: time.sleep(0.03), [], ())
    return s
```

### Sequential mapping

Start by using the `map` transformation without parallelism as a baseline example.

In [ ]:

```
benchmark(
    ArtificialDataset()
    .map(mapped_function)
)
```



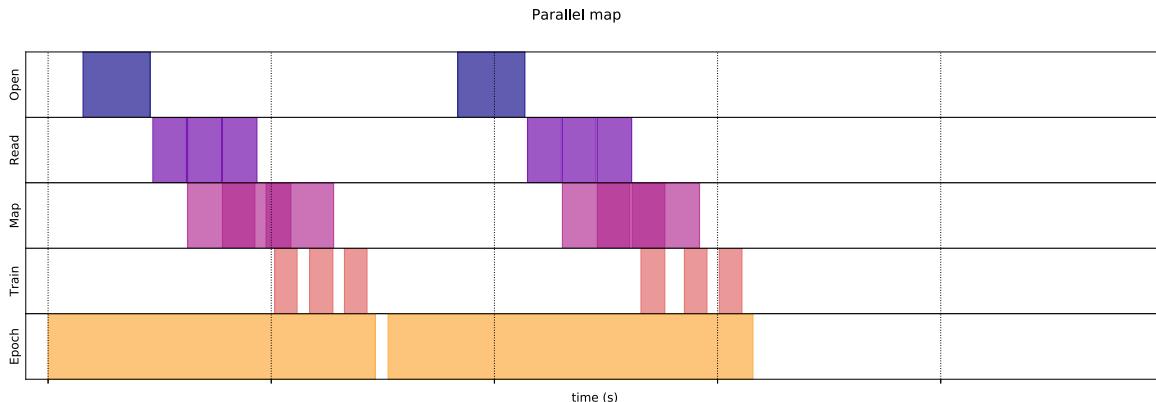
As for the [naive approach](#), here, as the plot shows, the times spent for opening, reading, pre-processing (mapping) and training steps sum together for a single iteration.

### Parallel mapping

Now, use the same pre-processing function but apply it in parallel on multiple samples.

In [ ]:

```
benchmark(  
    ArtificialDataset()  
    .map(  
        mapped_function,  
        num_parallel_calls=tf.data.AUTOTUNE  
    )  
)
```



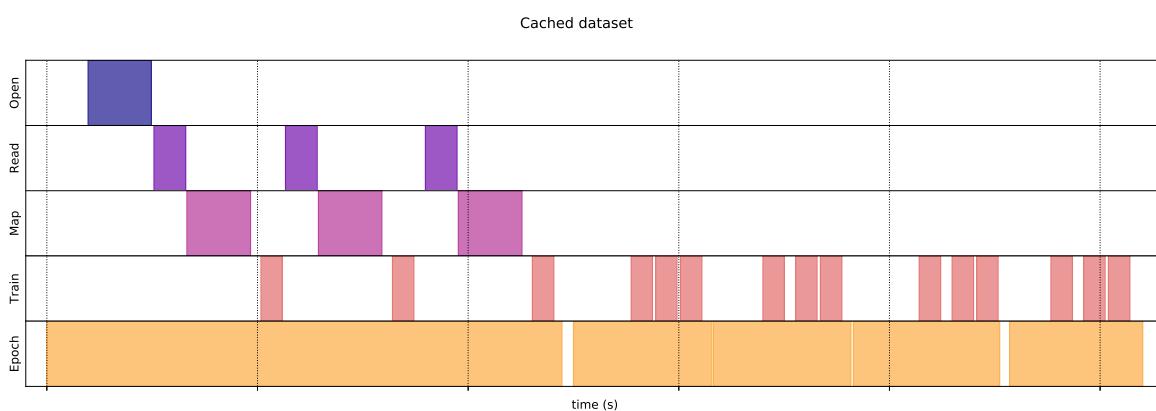
As the data plot demonstrates, the pre-processing steps overlap, reducing the overall time for a single iteration.

## Caching

The `tf.data.Dataset.cache` transformation can cache a dataset, either in memory or on local storage. This will save some operations (like file opening and data reading) from being executed during each epoch.

In [ ]:

```
benchmark(  
    ArtificialDataset()  
    .map( # Apply time consuming operations before cache  
        mapped_function  
    ).cache(  
    ),  
    5  
)
```



Here, the data execution time plot shows that when you cache a dataset, the transformations before the `cache` one (like the file opening and data reading) are executed only during the first epoch. The next epochs will reuse the data cached by the `cache` transformation.

If the user-defined function passed into the `map` transformation is expensive, apply the `cache` transformation after the `map` transformation as long as the resulting dataset can still fit into memory or local storage. If the user-defined function increases the space required to store the dataset beyond the cache capacity, either apply it after the `cache` transformation or consider pre-processing your data before your training job to reduce resource usage.

## Vectorizing mapping

Invoking a user-defined function passed into the `map` transformation has overhead related to scheduling and executing the user-defined function. Vectorize the user-defined function (that is, have it operate over a batch of inputs at once) and apply the `batch` transformation *before* the `map` transformation.

To illustrate this good practice, your artificial dataset is not suitable. The scheduling delay is around 10 microseconds (10e-6 seconds), far less than the tens of milliseconds used in the `ArtificialDataset`, and thus its impact is hard to see.

For this example, use the base `tf.data.Dataset.range` function and simplify the training loop to its simplest form.

In [ ]:

```
fast_dataset = tf.data.Dataset.range(10000)

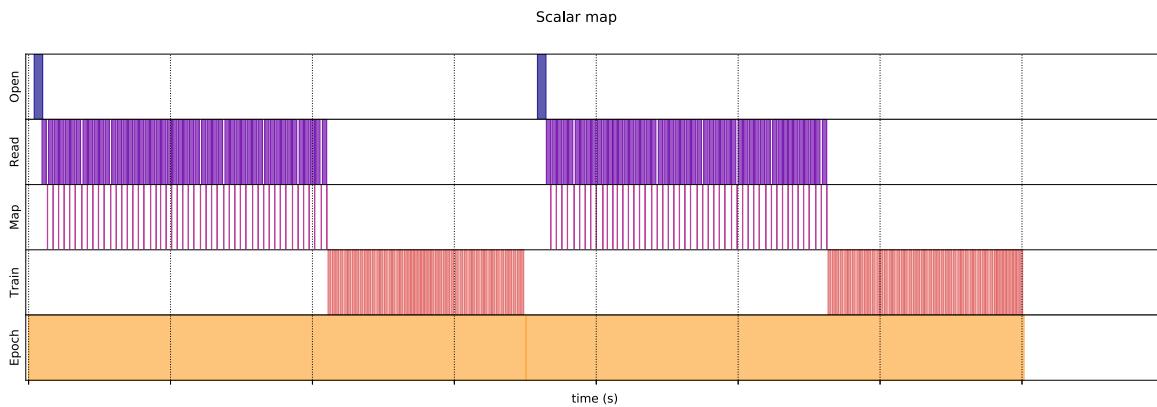
def fast_benchmark(dataset, num_epochs=2):
    start_time = time.perf_counter()
    for _ in tf.data.Dataset.range(num_epochs):
        for _ in dataset:
            pass
    tf.print("Execution time:", time.perf_counter() - start_time)

def increment(x):
    return x+1
```

## Scalar mapping

In [ ]:

```
fast_benchmark(
    fast_dataset
    # Apply function one item at a time
    .map(increment)
    # Batch
    .batch(256)
)
```

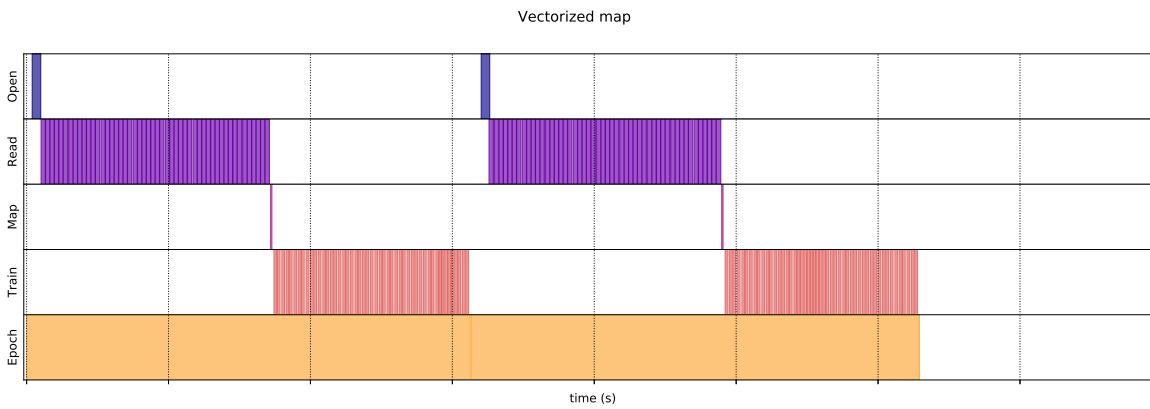


The plot above illustrates what is going on (with less samples) using the scalar mapping method. It shows that the mapped function is applied for each sample. While this function is very fast, it has some overhead that impact the time performance.

## Vectorized mapping

In [ ]:

```
fast_benchmark(
    fast_dataset
    .batch(256)
    # Apply function on a batch of items
    # The tf.Tensor.__add__ method already handle batches
    .map(increment)
)
```



This time, the mapped function is called once and applies to a batch of sample. As the data execution time plot shows, while the function could takes more time to execute, the overhead appear only once, improving the overall time performance.

## Reducing memory footprint

A number of transformations, including `interleave`, `prefetch`, and `shuffle`, maintain an internal buffer of elements. If the user-defined function passed into the `map` transformation changes the size of the elements, then the ordering of the map transformation and the transformations that buffer elements affects the memory usage. In general, choose the order that results in lower memory footprint, unless different ordering is desirable for performance.

### Caching partial computations

It is recommended to cache the dataset after the `map` transformation except if this transformation makes the data too big to fit in memory. A trade-off can be achieved if your mapped function can be split in two parts: a time consuming one and a memory consuming part. In this case, you can chain your transformations like below:

```
dataset.map(time-consuming_mapping).cache().map(memory-consuming_mapping)
```

This way, the time consuming part is only executed during the first epoch, and you avoid using too much cache space.

## Best practice summary

Here is a summary of the best practices for designing performant TensorFlow input pipelines:

- [Use the `prefetch` transformation](#) to overlap the work of a producer and consumer
- [Parallelize the data reading transformation](#) using the `interleave` transformation
- [Parallelize the `map` transformation](#) by setting the `num_parallel_calls` argument
- [Use the `cache` transformation](#) to cache data in memory during the first epoch
- [Vectorize user-defined functions](#) passed in to the `map` transformation
- [Reduce memory usage](#) when applying the `interleave`, `prefetch`, and `shuffle` transformations

## Reproducing the figures

Note: The rest of this notebook is about how to reproduce the above figures. Feel free to play around with this code, but understanding it is not an essential part of this tutorial.

To go deeper in the `tf.data.Dataset` API understanding, you can play with your own pipelines. Below is the code used to plot the images from this guide. It can be a good starting point, showing some workarounds for common difficulties such as:

- Execution time reproducibility
- Mapped functions eager execution
- `interleave` transformation callable

In [ ]:

```
import itertools
from collections import defaultdict

import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
```

## The dataset

Similar to the `ArtificialDataset` you can build a dataset returning the time spent in each step.

In [ ]:

```
class TimeMeasuredDataset(tf.data.Dataset):
    # OUTPUT: (steps, timings, counters)
    OUTPUT_TYPES = (tf.dtypes.string, tf.dtypes.float32, tf.dtypes.int32)
    OUTPUT_SHAPES = ((2, 1), (2, 2), (2, 3))

    _INSTANCES_COUNTER = itertools.count() # Number of datasets generated
    _EPOCHS_COUNTER = defaultdict(itertools.count) # Number of epochs done for each dataset

    def __generator(instance_idx, num_samples):
        epoch_idx = next(TimeMeasuredDataset._EPOCHS_COUNTER[instance_idx])

        # Opening the file
        open_enter = time.perf_counter()
        time.sleep(0.03)
        open_elapsed = time.perf_counter() - open_enter

        for sample_idx in range(num_samples):
            # Reading data (line, record) from the file
            read_enter = time.perf_counter()
            time.sleep(0.015)
            read_elapsed = time.perf_counter() - read_enter

            yield (
                ["Open", "Read"],
                [(open_enter, open_elapsed), (read_enter, read_elapsed)],
                [(instance_idx, epoch_idx, -1), (instance_idx, epoch_idx, sample_idx)])
        open_enter, open_elapsed = -1., -1. # Negative values will be filtered

    def __new__(cls, num_samples=3):
        return tf.data.Dataset.from_generator(
            cls.__generator,
            output_types=cls.OUTPUT_TYPES,
            output_shapes=cls.OUTPUT_SHAPES,
            args=(next(cls._INSTANCES_COUNTER), num_samples)
        )
```

This dataset provides samples of shape `[[2, 1], [2, 2], [2, 3]]` and of type `[tf.dtypes.string, tf.dtypes.float32, tf.dtypes.int32]`. Each sample is:

```
(  
    [("Open"), ("Read")],  
    [(t0, d), (t0, d)],  
    [(i, e, -1), (i, e, s)]  
)
```

Where:

- Open and Read are steps identifiers
- t0 is the timestamp when the corresponding step started
- d is the time spent in the corresponding step
- i is the instance index
- e is the epoch index (number of times the dataset has been iterated)
- s is the sample index

## The iteration loop

Make the iteration loop a little bit more complicated to aggregate all timings. This will only work with datasets generating samples as detailed above.

In [ ]:

```
def timelined_benchmark(dataset, num_epochs=2):
    # Initialize accumulators
    steps_acc = tf.zeros([0, 1], dtype=tf.dtypes.string)
    times_acc = tf.zeros([0, 2], dtype=tf.dtypes.float32)
    values_acc = tf.zeros([0, 3], dtype=tf.dtypes.int32)

    start_time = time.perf_counter()
    for epoch_num in range(num_epochs):
        epoch_enter = time.perf_counter()
        for (steps, times, values) in dataset:
            # Record dataset preparation informations
            steps_acc = tf.concat((steps_acc, steps), axis=0)
            times_acc = tf.concat((times_acc, times), axis=0)
            values_acc = tf.concat((values_acc, values), axis=0)

        # Simulate training time
        train_enter = time.perf_counter()
        time.sleep(0.01)
        train_elapsed = time.perf_counter() - train_enter

        # Record training informations
        steps_acc = tf.concat((steps_acc, [[{"Train"}]]), axis=0)
        times_acc = tf.concat((times_acc, [(train_enter, train_elapsed)]), axis=0)
        values_acc = tf.concat((values_acc, [values[-1]]), axis=0)

    epoch_elapsed = time.perf_counter() - epoch_enter
    # Record epoch informations
    steps_acc = tf.concat((steps_acc, [{"Epoch"}]), axis=0)
    times_acc = tf.concat((times_acc, [(epoch_enter, epoch_elapsed)]), axis=0)
    values_acc = tf.concat((values_acc, [[-1, epoch_num, -1]]), axis=0)
    time.sleep(0.001)

    tf.print("Execution time:", time.perf_counter() - start_time)
    return {"steps": steps_acc, "times": times_acc, "values": values_acc}
```

## The plotting method

Finally, define a function able to plot a timeline given the values returned by the `timelined_benchmark` function.

In [ ]:

```
def draw_timeline(timeline, title, width=0.5, annotate=False, save=False):
    # Remove invalid entries (negative times, or empty steps) from the timelines
    invalid_mask = np.logical_and(timeline['times'] > 0, timeline['steps'] != b'')[ :,0]
    steps = timeline['steps'][invalid_mask].numpy()
    times = timeline['times'][invalid_mask].numpy()
    values = timeline['values'][invalid_mask].numpy()

    # Get a set of different steps, ordered by the first time they are encountered
    step_ids, indices = np.stack(np.unique(steps, return_index=True))
    step_ids = step_ids[np.argsort(indices)]

    # Shift the starting time to 0 and compute the maximal time value
    min_time = times[:,0].min()
    times[:,0] = (times[:,0] - min_time)
    end = max(width, (times[:,0]+times[:,1]).max() + 0.01)

    cmap = mpl.cm.get_cmap("plasma")
    plt.close()
    fig, axs = plt.subplots(len(step_ids), sharex=True, gridspec_kw={'hspace': 0})
    fig.suptitle(title)
    fig.set_size_inches(17.0, len(step_ids))
    plt.xlim(-0.01, end)

    for i, step in enumerate(step_ids):
        step_name = step.decode()
        ax = axs[i]
        ax.set_ylabel(step_name)
        ax.set_ylim(0, 1)
        ax.set_yticks([])
        ax.set_xlabel("time (s)")
        ax.set_xticklabels([])
        ax.grid(which="both", axis="x", color="k", linestyle ":")

        # Get timings and annotation for the given step
        entries_mask = np.squeeze(steps==step)
        serie = np.unique(times[entries_mask], axis=0)
        annotations = values[entries_mask]

        ax.broken_barh(serie, (0, 1), color=cmap(i / len(step_ids)), linewidth=1, alpha=0.66)
        if annotate:
            for j, (start, width) in enumerate(serie):
                annotation = "\n".join([f"\{l\}: {v}" for l,v in zip(["i", "e", "s"], annotations[j])])
                ax.text(start + 0.001 + (0.001 * (j % 2)), 0.55 - (0.1 * (j % 2)), annotation,
                        horizontalalignment='left', verticalalignment='center')
        if save:
            plt.savefig(title.lower().translate(str.maketrans(" ", "_")) + ".svg")
```

## Use wrappers for mapped function

To run mapped function in an eager context, you have to wrap them inside a `tf.py_function` call.

In [ ]:

```
def map_decorator(func):
    def wrapper(steps, times, values):
        # Use a tf.py_function to prevent auto-graph from compiling the method
        return tf.py_function(
            func,
            inp=(steps, times, values),
            Tout=(steps.dtype, times.dtype, values.dtype)
        )
    return wrapper
```

## Pipelines comparison

In [ ]:

```
_batch_map_num_items = 50

def dataset_generator_fun(*args):
    return TimeMeasuredDataset(num_samples=_batch_map_num_items)
```

Naive

In [ ]:

```
@map_decorator
def naive_map(steps, times, values):
    map_enter = time.perf_counter()
    time.sleep(0.001) # Time consuming step
    time.sleep(0.0001) # Memory consuming step
    map_elapsed = time.perf_counter() - map_enter

    return (
        tf.concat((steps, [[ "Map"]]), axis=0),
        tf.concat((times, [[map_enter, map_elapsed]]), axis=0),
        tf.concat((values, [values[-1]]), axis=0)
    )

naive_timeline = timelined_benchmark(
    tf.data.Dataset.range(2)
    .flat_map(dataset_generator_fun)
    .map(naive_map)
    .batch(_batch_map_num_items, drop_remainder=True)
    .unbatch(),
    5
)
```

## Optimized

In [ ]:

```
@map_decorator
def time-consuming_map(steps, times, values):
    map_enter = time.perf_counter()
    time.sleep(0.001 * values.shape[0]) # Time consuming step
    map_elapsed = time.perf_counter() - map_enter

    return (
        tf.concat((steps, tf.tile([[["1st map"]]], [steps.shape[0], 1, 1])), axis=1),
        tf.concat((times, tf.tile([[map_enter, map_elapsed]]), [times.shape[0], 1, 1])), axis=1),
        tf.concat((values, tf.tile([[values[:][-1][0]]], [values.shape[0], 1, 1])), axis=1)
    )

@map_decorator
def memory-consuming_map(steps, times, values):
    map_enter = time.perf_counter()
    time.sleep(0.0001 * values.shape[0]) # Memory consuming step
    map_elapsed = time.perf_counter() - map_enter

    # Use tf.tile to handle batch dimension
    return (
        tf.concat((steps, tf.tile([[["2nd map"]]], [steps.shape[0], 1, 1])), axis=1),
        tf.concat((times, tf.tile([[map_enter, map_elapsed]]), [times.shape[0], 1, 1])), axis=1),
        tf.concat((values, tf.tile([[values[:][-1][0]]], [values.shape[0], 1, 1])), axis=1)
    )

optimized_timeline = timelined_benchmark(
    tf.data.Dataset.range(2)
    .interleave( # Parallelize data reading
        dataset_generator_fun,
        num_parallel_calls=tf.data.AUTOTUNE
    )
    .batch( # Vectorize your mapped function
        _batch_map_num_items,
        drop_remainder=True
    )
    .map( # Parallelize map transformation
        time-consuming_map,
        num_parallel_calls=tf.data.AUTOTUNE
    )
    .cache() # Cache data
    .map( # Reduce memory usage
        memory-consuming_map,
        num_parallel_calls=tf.data.AUTOTUNE
    )
    .prefetch( # Overlap producer and consumer works
        tf.data.AUTOTUNE
    )
    .unbatch(),
    5
)
```

```
In [ ]:
```

```
draw_timeline(naive_timeline, "Naive", 15)
```

```
In [ ]:
```

```
draw_timeline(optimized_timeline, "Optimized", 15)
```

Copyright 2021 The TensorFlow Authors.

```
In [ ]:
```

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Extension types



[View on TensorFlow.org](https://www.tensorflow.org/guide/extension_type)

([https://www.tensorflow.org/guide/extension\\_type](https://www.tensorflow.org/guide/extension_type))



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/extension_type.ipynb)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/extension\\_type.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/extension_type.ipynb))

## Setup

```
In [ ]:
```

```
!pip install -q tf_nightly
import tensorflow as tf
import numpy as np
from typing import Tuple, List, Mapping, Union, Optional
import tempfile
```

## Extension types

User-defined types can make projects more readable, modular, maintainable. However, most TensorFlow APIs have very limited support for user-defined Python types. This includes both high-level APIs (such as [Keras](https://www.tensorflow.org/guide/keras) (<https://www.tensorflow.org/guide/keras>), [tf.function](https://www.tensorflow.org/guide/function) (<https://www.tensorflow.org/guide/function>), [tf.SavedModel](https://www.tensorflow.org/guide/saved_model) ([https://www.tensorflow.org/guide/saved\\_model](https://www.tensorflow.org/guide/saved_model))) and lower-level APIs (such as `tf.while_loop` and `tf.concat`). TensorFlow **extension types** can be used to create user-defined object-oriented types that work seamlessly with TensorFlow's APIs. To create an extension type, simply define a Python class with `tf.experimental.ExtensionType` as its base, and use [type annotations](https://www.python.org/dev/peps/pep-0484/) (<https://www.python.org/dev/peps/pep-0484/>) to specify the type for each field.

```
In [ ]:
```

```
class TensorGraph(tf.experimental.ExtensionType):
    """A collection of labeled nodes connected by weighted edges."""
    edge_weights: tf.Tensor          # shape=[num_nodes, num_nodes]
    node_labels: Mapping[str, tf.Tensor] # shape=[num_nodes]; dtype=any

class MaskedTensor(tf.experimental.ExtensionType):
    """A tensor paired with a boolean mask, indicating which values are valid."""
    values: tf.Tensor
    mask: tf.Tensor      # shape=values.shape; false for missing/invalid values.

class CSRSparseMatrix(tf.experimental.ExtensionType):
    """Compressed sparse row matrix (https://en.wikipedia.org/wiki/Sparse\_matrix)."""
    values: tf.Tensor          # shape=[num_nonzero]; dtype=any
    col_index: tf.Tensor        # shape=[num_nonzero]; dtype=int64
    row_index: tf.Tensor        # shape=[num_rows+1]; dtype=int64
```

The `tf.experimental.ExtensionType` base class works similarly to [typing.NamedTuple](https://docs.python.org/3/library/typing.html#typing.NamedTuple) (<https://docs.python.org/3/library/typing.html#typing.NamedTuple>) and [@dataclasses.dataclass](https://docs.python.org/3/library/dataclasses.html#dataclasses.dataclass) (<https://docs.python.org/3/library/dataclasses.html#dataclasses.dataclass>) from the standard Python library. In particular, it automatically adds a constructor and special methods (such as `__repr__` and `__eq__`) based on the field type annotations.

Typically, extension types tend to fall into one of two categories:

- **Data structures**, which group together a collection of related values, and can provide useful operations based on those values. Data structures may be fairly general (such as the `TensorGraph` example above); or they may be highly customized to a specific model.
- **Tensor-like types**, which specialize or extend the concept of "Tensor." Types in this category have a `rank`, a `shape`, and usually a `dtype`; and it makes sense to use them with Tensor operations (such as `tf.stack`, `tf.add`, or `tf.matmul`). `MaskedTensor` and `CSRSParseMatrix` are examples of tensor-like types.

## Supported APIs

Extension types are supported by the following TensorFlow APIs:

- **Keras**: Extension types can be used as inputs and outputs for Keras Models and Layers .
- **tf.data.Dataset**: Extension types can be included in Datasets , and returned by dataset Iterators .
- **Tensorflow hub**: Extension types can be used as inputs and outputs for `tf.hub` modules.
- **SavedModel**: Extension types can be used as inputs and outputs for `SavedModel` functions.
- **tf.function**: Extension types can be used as arguments and return values for functions wrapped with the `@tf.function` decorator.
- **while loops**: Extension types can be used as loop variables in `tf.while_loop` , and can be used as arguments and return values for the while-loop's body.
- **conditionals**: Extension types can be conditionally selected using `tf.cond` and `tf.case` .
- **py\_function**: Extension types can be used as arguments and return values for the `func` argument to `tf.py_function` .
- **Tensor ops**: Extension types can be extended to support most TensorFlow ops that accept Tensor inputs (e.g., `tf.matmul` , `tf.gather` , and `tf.reduce_sum` ). See the "*Dispatch*" section below for more information.
- **distribution strategy**: Extension types can be used as per-replica values.

For more details, see the section on "TensorFlow APIs that support ExtensionTypes" below.

## Requirements

### Field types

All fields (aka instance variables) must be declared, and a type annotation must be provided for each field. The following type annotations are supported:

Type	Example
Python integers	i: int
Python floats	f: float
Python strings	s: str
Python booleans	b: bool
Python None	n: None
<a href="https://www.tensorflow.org/api_docs/python/tf/TensorShape">Tensor shapes (https://www.tensorflow.org/api_docs/python/tf/TensorShape)</a>	shape: tf.TensorShape
<a href="https://www.tensorflow.org/api_docs/python/tf/dtypes/DType">Tensor dtypes (https://www.tensorflow.org/api_docs/python/tf/dtypes/DType)</a>	dtype: tf.DType
<a href="https://www.tensorflow.org/api_docs/python/tf/Tensor">Tensors (https://www.tensorflow.org/api_docs/python/tf/Tensor)</a>	t: tf.Tensor
<a href="https://www.tensorflow.org/api_docs/python/tf/experimental/ExtensionType">Extension types (https://www.tensorflow.org/api_docs/python/tf/experimental/ExtensionType)</a>	mt: MyMaskedTensor
<a href="https://www.tensorflow.org/api_docs/python/tf/RaggedTensor">Ragged Tensors (https://www.tensorflow.org/api_docs/python/tf/RaggedTensor)</a>	rt: tf.RaggedTensor
<a href="https://www.tensorflow.org/api_docs/python/tf/sparse/SparseTensor">Sparse Tensors (https://www.tensorflow.org/api_docs/python/tf/sparse/SparseTensor)</a>	st: tf.SparseTensor
<a href="https://www.tensorflow.org/api_docs/python/tf/IndexedSlices">Indexed Slices (https://www.tensorflow.org/api_docs/python/tf/IndexedSlices)</a>	s: tf.IndexedSlices
<a href="https://www.tensorflow.org/api_docs/python/tf/experimental/Optional">Optional Tensors (https://www.tensorflow.org/api_docs/python/tf/experimental/Optional)</a>	o: tf.experimental.Optional
<a href="https://docs.python.org/3/library/typing.html#typing.Union">Type unions (https://docs.python.org/3/library/typing.html#typing.Union)</a>	int_or_float: typing.Union[int, float]
<a href="https://docs.python.org/3/library/typing.html#typing.Tuple">Tuples (https://docs.python.org/3/library/typing.html#typing.Tuple)</a>	params: typing.Tuple[int, float, tf.Tensor, int]
<a href="https://docs.python.org/3/library/typing.html#typing.Tuple">Var-length tuples (https://docs.python.org/3/library/typing.html#typing.Tuple)</a>	lengths: typing.Tuple[int, ...]
<a href="https://docs.python.org/3/library/typing.html#typing.Mapping">Mappings (https://docs.python.org/3/library/typing.html#typing.Mapping)</a>	tags: typing.Mapping[str, tf.Tensor]
<a href="https://docs.python.org/3/library/typing.html#typing.Optional">Optional values (https://docs.python.org/3/library/typing.html#typing.Optional)</a>	weight: typing.Optional[tf.Tensor]

### Mutability

Extension types are required to be immutable. This ensures that they can be properly tracked by TensorFlow's graph-tracing mechanisms. If you find yourself wanting to mutate an extension type value, consider instead defining methods that transform values. For example, rather than defining a `set_mask` method to mutate a `MaskedTensor` , you could define a `replace_mask` method that returns a new `MaskedTensor` :

In [ ]:

```
class MaskedTensor(tf.experimental.ExtensionType):
    values: tf.Tensor
    mask: tf.Tensor

    def replace_mask(self, new_mask):
        self.values.shape.assert_is_compatible_with(new_mask.shape)
        return MaskedTensor(self.values, new_mask)
```

## Functionality added by ExtensionType

The `ExtensionType` base class provides the following functionality:

- A constructor (`__init__`).
- A printable representation method (`__repr__`).
- Equality and inequality operators (`__eq__`).
- A validation method (`__validate__`).
- Enforced immutability.
- A nested `TypeSpec`.
- Tensor API dispatch support.

See the "Customizing ExtensionTypes" section below for more information on customizing this functionality.

## Constructor

The constructor added by `ExtensionType` takes each field as a named argument (in the order they were listed in the class definition). This constructor will type-check each parameter, and convert them where necessary. In particular, `Tensor` fields are converted using `tf.convert_to_tensor`; `Tuple` fields are converted to `tuples`; and `Mapping` fields are converted to immutable dicts.

In [ ]:

```
class MaskedTensor(tf.experimental.ExtensionType):
    values: tf.Tensor
    mask: tf.Tensor

    # Constructor takes one parameter for each field.
    mt = MaskedTensor(values=[[1, 2, 3], [4, 5, 6]],
                      mask=[[True, True, False], [True, False, True]])

    # Fields are type-checked and converted to the declared types.
    # E.g., mt.values is converted to a Tensor.
    print(mt.values)
```

The constructor raises an `TypeError` if a field value can not be converted to its declared type:

In [ ]:

```
try:
    MaskedTensor([1, 2, 3], None)
except TypeError as e:
    print(f"Got expected TypeError: {e}")
```

The default value for a field can be specified by setting its value at the class level:

In [ ]:

```
class Pencil(tf.experimental.ExtensionType):
    color: str = "black"
    has_erasor: bool = True
    length: tf.Tensor = 1.0

Pencil()
```

In [ ]:

```
Pencil(length=0.5, color="blue")
```

## Printable representation

`ExtensionType` adds a default printable representation method (`__repr__`) that includes the class name and the value for each field:

```
In [ ]:
```

```
print(MaskedTensor(values=[1, 2, 3], mask=[True, True, False]))
```

## Equality operators

ExtensionType adds default equality operators (`__eq__` and `__ne__`) that consider two values equal if they have the same type and all their fields are equal. Tensor fields are considered equal if they have the same shape and are elementwise equal for all elements.

```
In [ ]:
```

```
a = MaskedTensor([1, 2], [True, False])
b = MaskedTensor([[3, 4], [5, 6]], [[False, True], [True, True]])
print(f'a == a: {a==a}')
print(f'a == b: {a==b}')
print(f'a == a.values: {a==a.values}')
```

Note: if any field contains a `Tensor`, then `__eq__` may return a scalar boolean `Tensor` (rather than a Python boolean value).

## Validation method

ExtensionType adds a `__validate__` method, which can be overridden to perform validation checks on fields. It is run after the constructor is called, and after fields have been type-checked and converted to their declared types, so it can assume that all fields have their declared types.

The following example updates `MaskedTensor` to validate the `shape`s and `dtype`s of its fields:

```
In [ ]:
```

```
class MaskedTensor(tf.experimental.ExtensionType):
    """A tensor paired with a boolean mask, indicating which values are valid."""
    values: tf.Tensor
    mask: tf.Tensor
    def __validate__(self):
        self.values.shape.assert_is_compatible_with(self.mask.shape)
        assert self.mask.dtype.is_bool, 'mask.dtype must be bool'
```

```
In [ ]:
```

```
try:
    MaskedTensor([1, 2, 3], [0, 1, 0]) # wrong dtype for mask.
except AssertionError as e:
    print(f'Got expected AssertionError: {e}')
```

```
In [ ]:
```

```
try:
    MaskedTensor([1, 2, 3], [True, False]) # shapes don't match.
except ValueError as e:
    print(f'Got expected ValueError: {e}')
```

## Enforced immutability

ExtensionType overrides the `__setattr__` and `__delattr__` methods to prevent mutation, ensuring that extension type values are immutable.

```
In [ ]:
```

```
mt = MaskedTensor([1, 2, 3], [True, False, True])
```

```
In [ ]:
```

```
try:
    mt.mask = [True, True, True]
except AttributeError as e:
    print(f'Got expected AttributeError: {e}')
```

```
In [ ]:
```

```
try:
    mt.mask[0] = False
except TypeError as e:
    print(f'Got expected TypeError: {e}')
```

```
In [ ]:
```

```
try:  
    del mt.mask  
except AttributeError as e:  
    print(f"Got expected AttributeError: {e}")
```

## Nested TypeSpec

Each `ExtensionType` class has a corresponding `TypeSpec` class, which is created automatically and stored as `<extension_type_name>.Spec`.

This class captures all the information from a value except for the values of any nested tensors. In particular, the `TypeSpec` for a value is created by replacing any nested Tensor, ExtensionType, or CompositeTensor with its `TypeSpec`.

```
In [ ]:
```

```
class Player(tf.experimental.ExtensionType):  
    name: tf.Tensor  
    attributes: Mapping[str, tf.Tensor]  
  
anne = Player("Anne", {"height": 8.3, "speed": 28.1})  
anne_spec = tf.type_spec_from_value(anne)  
print(anne_spec.name) # Records dtype and shape, but not the string value.  
print(anne_spec.attributes) # Records keys and TensorSpecs for values.
```

`TypeSpec` values can be constructed explicitly, or they can be built from an `ExtensionType` value using `tf.type_spec_from_value`:

```
In [ ]:
```

```
spec1 = Player.Spec(name=tf.TensorSpec([], tf.float32), attributes={})  
spec2 = tf.type_spec_from_value(anne)
```

`TypeSpec`s are used by TensorFlow to divide values into a **static component** and a **dynamic component**:

- The **static component** (which is fixed at graph-construction time) is encoded with a `tf.TypeSpec`.
- The **dynamic component** (which can vary each time the graph is run) is encoded as a list of `tf.Tensors`.

For example, `tf.function` retraces its wrapped function whenever an argument has a previously unseen `TypeSpec`:

```
In [ ]:
```

```
@tf.function  
def anonymize_player(player):  
    print("<<TRACING>>")  
    return Player("<anonymous>", player.attributes)
```

```
In [ ]:
```

```
# Function gets traced (first time the function has been called):  
anonymize_player(Player("Anne", {"height": 8.3, "speed": 28.1}))
```

```
In [ ]:
```

```
# Function does NOT get traced (same TypeSpec: just tensor values changed)  
anonymize_player(Player("Bart", {"height": 8.1, "speed": 25.3}))
```

```
In [ ]:
```

```
# Function gets traced (new TypeSpec: keys for attributes changed):  
anonymize_player(Player("Chuck", {"height": 11.0, "jump": 5.3}))
```

For more information, see the [tf.function Guide](https://www.tensorflow.org/guide/function#rules_of_tracing) ([https://www.tensorflow.org/guide/function#rules\\_of\\_tracing](https://www.tensorflow.org/guide/function#rules_of_tracing)).

## Customizing ExtensionTypes

In addition to simply declaring fields and their types, extension types may:

- Override the default printable representation (`__repr__`).
- Define methods.
- Define classmethods and staticmethods.
- Define properties.
- Override the default constructor (`__init__`).
- Override the default equality operator (`__eq__`).
- Define operators (such as `__add__` and `__lt__`).
- Declare default values for fields.
- Define subclasses.

### Overriding the default printable representation

You can override this default string conversion operator for extension types. The following example updates the `MaskedTensor` class to generate a more readable string representation when values are printed in Eager mode.

In [ ]:

```
class MaskedTensor(tf.experimental.ExtensionType):
    """A tensor paired with a boolean mask, indicating which values are valid."""
    values: tf.Tensor
    mask: tf.Tensor      # shape=values.shape; false for invalid values.

    def __repr__(self):
        return masked_tensor_str(self.values, self.mask)

    def masked_tensor_str(values, mask):
        if isinstance(values, tf.Tensor):
            if hasattr(values, 'numpy') and hasattr(mask, 'numpy'):
                return f'<MaskedTensor {masked_tensor_str(values.numpy(), mask.numpy())}>'
            else:
                return f'MaskedTensor(values={values}, mask={mask})'
        if len(values.shape) == 1:
            items = [repr(v) if m else '_' for (v, m) in zip(values, mask)]
        else:
            items = [masked_tensor_str(v, m) for (v, m) in zip(values, mask)]
        return '[%s]' % ', '.join(items)

mt = MaskedTensor(values=[[1, 2, 3], [4, 5, 6]],
                  mask=[[True, True, False], [True, False, True]])
print(mt)
```

### Defining methods

Extension types may define methods, just like any normal Python class. For example, the `MaskedTensor` type could define a `with_default` method that returns a copy of `self` with masked values replaced by a given `default` value. Methods may optionally be annotated with the `@tf.function` decorator.

In [ ]:

```
class MaskedTensor(tf.experimental.ExtensionType):
    values: tf.Tensor
    mask: tf.Tensor

    def with_default(self, default):
        return tf.where(self.mask, self.values, default)

MaskedTensor([1, 2, 3], [True, False, True]).with_default(0)
```

### Defining classmethods and staticmethods

Extension types may define methods using the `@classmethod` and `@staticmethod` decorators. For example, the `MaskedTensor` type could define a factory method that masks any element with a given value:

```
In [ ]:
```

```
class MaskedTensor(tf.experimental.ExtensionType):
    values: tf.Tensor
    mask: tf.Tensor

    def __repr__(self):
        return masked_tensor_str(self.values, self.mask)

    @staticmethod
    def from_tensor_and_value_to_mask(values, value_to_mask):
        return MaskedTensor(values, values == value_to_mask)

x = tf.constant([[1, 0, 2], [3, 0, 0]])
MaskedTensor.from_tensor_and_value_to_mask(x, 0)
```

## Defining properties

Extension types may define properties using the `@property` decorator, just like any normal Python class. For example, the `MaskedTensor` type could define a `dtype` property that's a shorthand for the `dtype` of the `values`:

```
In [ ]:
```

```
class MaskedTensor(tf.experimental.ExtensionType):
    values: tf.Tensor
    mask: tf.Tensor

    @property
    def dtype(self):
        return self.values.dtype

MaskedTensor([1, 2, 3], [True, False, True]).dtype
```

## Overriding the default constructor

You can override the default constructor for extension types. Custom constructors must set a value for every declared field; and after the custom constructor returns, all fields will be type-checked, and values will be converted as described above.

```
In [ ]:
```

```
class Toy(tf.experimental.ExtensionType):
    name: str
    price: tf.Tensor
    def __init__(self, name, price, discount=0):
        self.name = name
        self.price = price * (1 - discount)

print(Toy("ball", 5.0, discount=0.2)) # On sale -- 20% off!
```

Alternatively, you might consider leaving the default constructor as-is, but adding one or more factory methods. E.g.:

```
In [ ]:
```

```
class Toy(tf.experimental.ExtensionType):
    name: str
    price: tf.Tensor

    @staticmethod
    def new_toy_with_discount(name, price, discount):
        return Toy(name, price * (1 - discount))

print(Toy.new_toy_with_discount("ball", 5.0, discount=0.2))
```

## Overriding the default equality operator (`__eq__`)

You can override the default `__eq__` operator for extension types. The follow example updates `MaskedTensor` to ignore masked elements when comparing for equality.

```
In [ ]:
```

```
class MaskedTensor(tf.experimental.ExtensionType):
    values: tf.Tensor
    mask: tf.Tensor

    def __repr__(self):
        return masked_tensor_str(self.values, self.mask)

    def __eq__(self, other):
        result = tf.math.equal(self.values, other.values)
        result = result | ~(self.mask & other.mask)
        return tf.reduce_all(result)

x = MaskedTensor([1, 2, 3, 4], [True, True, False, True])
y = MaskedTensor([5, 2, 0, 4], [False, True, False, True])
print(x == y)
```

Note: You generally don't need to override `__ne__`, since its default implementation simply calls `__eq__` and negates the result.

## Using forward references

If the type for a field has not been defined yet, you may use a string containing the name of the type instead. In the following example, the string "Node" is used to annotate the `children` field because the `Node` type hasn't been (fully) defined yet.

```
In [ ]:
```

```
class Node(tf.experimental.ExtensionType):
    value: tf.Tensor
    children: Tuple["Node", ...] = ()

Node(3, [Node(5), Node(2)])
```

## Defining subclasses

Extension types may be subclassed using the standard Python syntax. Extension type subclasses may add new fields, methods, and properties; and may override the constructor, the printable representation, and the equality operator. The following example defines a basic `TensorGraph` class that uses three `Tensor` fields to encode a set of edges between nodes. It then defines a subclass that adds a `Tensor` field to record a "feature value" for each node. The subclass also defines a method to propagate the feature values along the edges.

```
In [ ]:
```

```
class TensorGraph(tf.experimental.ExtensionType):
    num_nodes: tf.Tensor
    edge_src: tf.Tensor # edge_src[e] = index of src node for edge e.
    edge_dst: tf.Tensor # edge_dst[e] = index of dst node for edge e.

class TensorGraphWithNodeFeature(TensorGraph):
    node_features: tf.Tensor # node_features[n] = feature value for node n.

    def propagate_features(self, weight=1.0) -> 'TensorGraphWithNodeFeature':
        updates = tf.gather(self.node_features, self.edge_src) * weight
        new_node_features = tf.tensor_scatter_nd_add(
            self.node_features, tf.expand_dims(self.edge_dst, 1), updates)
        return TensorGraphWithNodeFeature(
            self.num_nodes, self.edge_src, self.edge_dst, new_node_features)

g = TensorGraphWithNodeFeature( # Edges: 0->1, 4->3, 2->2, 2->1
    num_nodes=5, edge_src=[0, 4, 2, 2], edge_dst=[1, 3, 2, 1],
    node_features=[10.0, 0.0, 2.0, 5.0, -1.0, 0.0])

print("Original features:", g.node_features)
print("After propagating:", g.propagate_features().node_features)
```

## Defining private fields

An extension type's fields may be marked private by prefixing them with an underscore (following standard Python conventions). This does not impact the way that TensorFlow treats the fields in any way; but simply serves as a signal to any users of the extension type that those fields are private.

## Customizing the ExtensionType's TypeSpec

Each `ExtensionType` class has a corresponding `TypeSpec` class, which is created automatically and stored as `<extension_type_name>.Spec`. For more information, see the section "Nested TypeSpec" above.

To customize the `TypeSpec`, simply define your own nested class named `Spec`, and `ExtensionType` will use that as the basis for the automatically constructed `TypeSpec`. You can customize the `Spec` class by:

- Overriding the default printable representation.
- Overriding the default constructor.
- Defining methods, classmethods, staticmethods, and properties.

The following example customizes the `MaskedTensor.Spec` class to make it easier to use:

In [ ]:

```
class MaskedTensor(tf.experimental.ExtensionType):
    values: tf.Tensor
    mask: tf.Tensor

    shape = property(lambda self: self.values.shape)
    dtype = property(lambda self: self.values.dtype)

    def __repr__(self):
        return masked_tensor_str(self.values, self.mask)

    def with_values(self, new_values):
        return MaskedTensor(new_values, self.mask)

    class Spec:
        def __init__(self, shape, dtype=tf.float32):
            self.values = tf.TensorSpec(shape, dtype)
            self.mask = tf.TensorSpec(shape, tf.bool)

        def __repr__(self):
            return f"MaskedTensor.Spec(shape={self.shape}, dtype={self.dtype})"

        shape = property(lambda self: self.values.shape)
        dtype = property(lambda self: self.values.dtype)
```

Note: The custom `Spec` class may not use any instance variables that were not declared in the original `ExtensionType`.

## Tensor API dispatch

Extension types can be "tensor-like", in the sense that they specialize or extend the interface defined by the `tf.Tensor` type. Examples of tensor-like extension types include `RaggedTensor`, `SparseTensor`, and `MaskedTensor`. **Dispatch decorators** can be used to override the default behavior of TensorFlow operations when applied to tensor-like extension types. TensorFlow currently defines three dispatch decorators:

- `@tf.experimental.dispatch_for_api(tf_api)`
- `@tf.experimental.dispatch_for_unary_elementwise_api(x_type)`
- `@tf.experimental.dispatch_for_binary_elementwise_apis(x_type, y_type)`

## Dispatch for a single API

The `tf.experimental.dispatch_for_api` decorator overrides the default behavior of a specified TensorFlow operation when it is called with the specified signature. For example, you can use this decorator to specify how `tf.stack` should process `MaskedTensor` values:

In [ ]:

```
@tf.experimental.dispatch_for_api(tf.stack)
def masked_stack(values: List[MaskedTensor], axis = 0):
    return MaskedTensor(tf.stack([v.values for v in values], axis),
                        tf.stack([v.mask for v in values], axis))
```

This overrides the default implementation for `tf.stack` whenever it is called with a list of `MaskedTensor` values (since the `values` argument is annotated with `typing.List[MaskedTensor]`):

In [ ]:

```
x = MaskedTensor([1, 2, 3], [True, True, False])
y = MaskedTensor([4, 5, 6], [False, True, True])
tf.stack([x, y])
```

To allow `tf.stack` to handle lists of mixed `MaskedTensor` and `Tensor` values, you can refine the type annotation for the `values` parameter and update the body of the function appropriately:

In [ ]:

```
tf.experimental.unregister_dispatch_for(masked_stack)

def convert_to_masked_tensor(x):
    if isinstance(x, MaskedTensor):
        return x
    else:
        return MaskedTensor(x, tf.ones_like(x, tf.bool))

@tf.experimental.dispatch_for_api(tf.stack)
def masked_stack_v2(values: List[Union[MaskedTensor, tf.Tensor]], axis = 0):
    values = [convert_to_masked_tensor(v) for v in values]
    return MaskedTensor(tf.stack([v.values for v in values], axis),
                        tf.stack([v.mask for v in values], axis))
x = MaskedTensor([1, 2, 3], [True, True, False])
y = tf.constant([4, 5, 6])
tf.stack([x, y, x])
```

For a list of APIs that can be overridden, see the API documentation for `tf.experimental.dispatch_for_api`.

## Dispatch for all unary elementwise APIs

The `tf.experimental.dispatch_for_unary_elementwise_apis` decorator overrides the default behavior of *all* unary elementwise ops (such as `tf.math.cos`) whenever the value for the first argument (typically named `x`) matches the type annotation `x_type`. The decorated function should take two arguments:

- `api_func`: A function that takes a single parameter and performs the elementwise operation (e.g., `tf.abs`).
- `x`: The first argument to the elementwise operation.

The following example updates all unary elementwise operations to handle the `MaskedTensor` type:

In [ ]:

```
@tf.experimental.dispatch_for_unary_elementwise_apis(MaskedTensor)
def masked_tensor_unary_elementwise_api_handler(api_func, x):
    return MaskedTensor(api_func(x.values), x.mask)
```

This function will now be used whenever a unary elementwise operation is called on a `MaskedTensor`.

In [ ]:

```
x = MaskedTensor([1, -2, -3], [True, False, True])
print(tf.abs(x))
```

In [ ]:

```
print(tf.ones_like(x, dtype=tf.float32))
```

## Dispatch for binary all elementwise APIs

Similarly, `tf.experimental.dispatch_for_binary_elementwise_apis` can be used to update all binary elementwise operations to handle the `MaskedTensor` type:

In [ ]:

```
@tf.experimental.dispatch_for_binary_elementwise_apis(MaskedTensor, MaskedTensor)
def masked_tensor_binary_elementwise_api_handler(api_func, x, y):
    return MaskedTensor(api_func(x.values, y.values), x.mask & y.mask)
```

In [ ]:

```
x = MaskedTensor([1, -2, -3], [True, False, True])
y = MaskedTensor([[4], [5]], [[True], [False]])
tf.math.add(x, y)
```

For a list of the elementwise APIs that are overridden, see the API documentation for `tf.experimental.dispatch_for_unary_elementwise_apis` and `tf.experimental.dispatch_for_binary_elementwise_apis`.

## Batchable ExtensionTypes

An `ExtensionType` is *batchable* if a single instance can be used to represent a batch of values. Typically, this is accomplished by adding batch dimensions to all nested `Tensor`s. The following TensorFlow APIs require that any extension type inputs be batchable:

- `tf.data.Dataset` (`batch`, `unbatch`, `from_tensor_slices`)
- `tf.Keras` (`fit`, `evaluate`, `predict`)
- `tf.map_fn`

By default, `BatchableExtensionType` creates batched values by batching any nested `Tensor`s, `CompositeTensor`s, and `ExtensionType`s. If this is not appropriate for your class, then you will need to use `tf.experimental.ExtensionTypeBatchEncoder` to override this default behavior. For example, it would not be appropriate to create a batch of `tf.SparseTensor` values by simply stacking individual sparse tensors' `values`, `indices`, and `dense_shape` fields -- in most cases, you can't stack these tensors, since they have incompatible shapes; and even if you could, the result would not be a valid `SparseTensor`.

**Note:** `BatchableExtensionType`s do *not* automatically define dispatchers for `tf.stack`, `tf.concat`, `tf.slice`, etc. If your class needs to be supported by these APIs, then use the dispatch decorators described above.

### BatchableExtensionType example: Network

As an example, consider a simple `Network` class used for load balancing, which tracks how much work is left to do at each node, and how much bandwidth is available to move work between nodes:

In [ ]:

```
class Network(tf.experimental.ExtensionType): # This version is not batchable.
    work: tf.Tensor      # work[n] = work left to do at node n
    bandwidth: tf.Tensor # bandwidth[n1, n2] = bandwidth from n1->n2

net1 = Network([5., 3, 8], [[0., 2, 0], [2, 0, 3], [0, 3, 0]])
net2 = Network([3., 4, 2], [[0., 2, 2], [2, 0, 2], [2, 2, 0]])
```

To make this type batchable, change the base type to `BatchableExtensionType`, and adjust the shape of each field to include optional batch dimensions. The following example also adds a `shape` field to keep track of the batch shape. This `shape` field is not required by `tf.data.Dataset` or `tf.map_fn`, but it *is* required by `tf.Keras`.

In [ ]:

```
class Network(tf.experimental.BatchableExtensionType):
    shape: tf.TensorShape # batch shape. A single network has shape=[].
    work: tf.Tensor        # work[*shape, n] = work left to do at node n
    bandwidth: tf.Tensor   # bandwidth[*shape, n1, n2] = bandwidth from n1->n2

    def __init__(self, work, bandwidth):
        self.work = tf.convert_to_tensor(work)
        self.bandwidth = tf.convert_to_tensor(bandwidth)
        work_batch_shape = self.work.shape[:-1]
        bandwidth_batch_shape = self.bandwidth.shape[:-2]
        self.shape = work_batch_shape.merge_with(bandwidth_batch_shape)

    def __repr__(self):
        return network_repr(self)

    def network_repr(network):
        work = network.work
        bandwidth = network.bandwidth
        if hasattr(work, 'numpy'):
            work = ' '.join(str(work.numpy()).split())
        if hasattr(bandwidth, 'numpy'):
            bandwidth = ' '.join(str(bandwidth.numpy()).split())
        return f"<Network shape={network.shape} work={work} bandwidth={bandwidth}>"
```

In [ ]:

```
net1 = Network([5., 3, 8], [[0., 2, 0], [2, 0, 3], [0, 3, 0]])
net2 = Network([3., 4, 2], [[0., 2, 2], [2, 0, 2], [2, 2, 0]])
batch_of_networks = Network(
    work=tf.stack([net1.work, net2.work]),
    bandwidth=tf.stack([net1.bandwidth, net2.bandwidth]))
print(f"net1={net1}")
print(f"net2={net2}")
print(f"batch={batch_of_networks}")
```

You can then use `tf.data.Dataset` to iterate through a batch of networks:

In [ ]:

```
dataset = tf.data.Dataset.from_tensor_slices(batch_of_networks)
for i, network in enumerate(dataset):
    print(f"Batch element {i}: {network}")
```

And you can also use `map_fn` to apply a function to each batch element:

In [ ]:

```
def balance_work_greedy(network):
    delta = (tf.expand_dims(network.work, -1) - tf.expand_dims(network.work, -2))
    delta /= 4
    delta = tf.maximum(tf.minimum(delta, network.bandwidth), -network.bandwidth)
    new_work = network.work + tf.reduce_sum(delta, -1)
    return Network(new_work, network.bandwidth)

tf.map_fn(balance_work_greedy, batch_of_networks)
```

## TensorFlow APIs that support ExtensionTypes

### @tf.function

`tf.function` (<https://www.tensorflow.org/guide/function>) is a decorator that precomputes TensorFlow graphs for Python functions, which can substantially improve the performance of your TensorFlow code. Extension type values can be used transparently with `@tf.function`-decorated functions.

In [ ]:

```
class Pastry(tf.experimental.ExtensionType):
    sweetness: tf.Tensor # 2d embedding that encodes sweetness
    chewiness: tf.Tensor # 2d embedding that encodes chewiness

@tf.function
def combine_pastry_features(x: Pastry):
    return (x.sweetness + x.chewiness) / 2

cookie = Pastry(sweetness=[1.2, 0.4], chewiness=[0.8, 0.2])
combine_pastry_features(cookie)
```

If you wish to explicitly specify the `input_signature` for `tf.function`, then you can do so using the extension type's `TypeSpec`.

In [ ]:

```
pastry_spec = Pastry.Spec(tf.TensorSpec([2]), tf.TensorSpec(2))

@tf.function(input_signature=[pastry_spec])
def increase_sweetness(x: Pastry, delta=1.0):
    return Pastry(x.sweetness + delta, x.chewiness)

increase_sweetness(cookie)
```

### Concrete functions

Concrete functions encapsulate individual traced graphs that are built by `tf.function`. Extension types can be used transparently with concrete functions.

In [ ]:

```
cf = combine_pastry_features.get_concrete_function(pastry_spec)
cf(cookie)
```

## Control flow operations

Extension types are supported by TensorFlow's control-flow operations:

- `tf.cond`
- `tf.case`
- `tf.while_loop`
- `tf.identity`

In [ ]:

```
# Example: using tf.cond to select between two MaskedTensors. Note that the
# two MaskedTensors don't need to have the same shape.
a = MaskedTensor([1., 2, 3], [True, False, True])
b = MaskedTensor([22., 33, 108, 55], [True, True, True, False])
condition = tf.constant(True)
print(tf.cond(condition, lambda: a, lambda: b))
```

In [ ]:

```
# Example: using tf.while_loop with MaskedTensor.
cond = lambda i, _: i < 10
def body(i, mt):
    return i + 1, mt.with_values(mt.values + 3 / 7)
print(tf.while_loop(cond, body, [0, b])[1])
```

## Autograph control flow

Extension types are also supported by control flow statements in `tf.function` (using autograph). In the following example, the `if` statement and `for` statements are automatically converted to `tf.cond` and `tf.while_loop` operations, which support extension types.

In [ ]:

```
@tf.function
def fn(x, b):
    if b:
        x = MaskedTensor(x, tf.less(x, 0))
    else:
        x = MaskedTensor(x, tf.greater(x, 0))
    for i in tf.range(5 if b else 7):
        x = x.with_values(x.values + 1 / 2)
    return x

print(fn(tf.constant([1., -2, 3]), tf.constant(True)))
print(fn(tf.constant([1., -2, 3]), tf.constant(False)))
```

## Keras

[tf.keras](https://www.tensorflow.org/guide/keras) (<https://www.tensorflow.org/guide/keras>) is TensorFlow's high-level API for building and training deep learning models. Extension types may be passed as inputs to a Keras model, passed between Keras layers, and returned by Keras models. Keras currently puts two requirements on extension types:

- They must be batchable (see "Batchable ExtensionTypes" above).
- They must have a field or property named `shape`. `shape[0]` is assumed to be the batch dimension.

The following two subsections give examples showing how extension types can be used with Keras.

### Keras example: Network

For the first example, consider the `Network` class defined in the "Batchable ExtensionTypes" section above, which can be used for load balancing work between nodes. Its definition is repeated here:

In [ ]:

```
class Network(tf.experimental.BatchableExtensionType):
    shape: tf.TensorShape # batch shape. A single network has shape=[].
    work: tf.Tensor       # work[*shape, n] = work left to do at node n
    bandwidth: tf.Tensor  # bandwidth[*shape, n1, n2] = bandwidth from n1->n2

    def __init__(self, work, bandwidth):
        self.work = tf.convert_to_tensor(work)
        self.bandwidth = tf.convert_to_tensor(bandwidth)
        work_batch_shape = self.work.shape[:-1]
        bandwidth_batch_shape = self.bandwidth.shape[:-2]
        self.shape = work_batch_shape.merge_with(bandwidth_batch_shape)

    def __repr__(self):
        return network_repr(self)
```

In [ ]:

```
single_network = Network( # A single network w/ 4 nodes.  
    work=[8.0, 5, 12, 2],  
    bandwidth=[[0.0, 1, 2, 2], [1, 0, 0, 2], [2, 0, 0, 1], [2, 2, 1, 0]])  
  
batch_of_networks = Network( # Batch of 2 networks, each w/ 2 nodes.  
    work=[[8.0, 5], [3, 2]],  
    bandwidth=[[0.0, 1], [1, 0]], [[0, 2], [2, 0]])
```

You can define a new Keras layer that processes Network s.

In [ ]:

```
class BalanceNetworkLayer(tf.keras.layers.Layer):  
    """Layer that balances work between nodes in a network.  
    Shifts work from more busy nodes to less busy nodes, constrained by bandwidth.  
    """  
    def call(self, inputs):  
        # This function is defined above, in "Batchable ExtensionTypes" section.  
        return balance_work_greedy(inputs)
```

You can then use this layers to create a simple model. To feed an ExtensionType into a model, you can use a `tf.keras.layers.Input` layer with `type_spec` set to the extension type's `TypeSpec`. If the Keras model will be used to process batches, then the `type_spec` must include the batch dimension.

In [ ]:

```
input_spec = Network.Spec(shape=None,  
                           work=tf.TensorSpec(None, tf.float32),  
                           bandwidth=tf.TensorSpec(None, tf.float32))  
model = tf.keras.Sequential([  
    tf.keras.layers.Input(type_spec=input_spec),  
    BalanceNetworkLayer(),  
])
```

Finally, you can apply the model to a single network and to a batch of networks.

In [ ]:

```
model(single_network)
```

In [ ]:

```
model(batch_of_networks)
```

### Keras example: MaskedTensor

In this example, `MaskedTensor` is extended to support Keras. `shape` is defined as a property that is calculated from the `values` field. Keras requires that you add this property to both the extension type and its `TypeSpec`. `MaskedTensor` also defines a `__name__` variable, which will be required for `SavedModel` serialization (below).

In [ ]:

```
class MaskedTensor(tf.experimental.BatchableExtensionType):
    # __name__ is required for serialization in SavedModel; see below for details.
    __name__ = 'extension_type_colab.MaskedTensor'

    values: tf.Tensor
    mask: tf.Tensor

    shape = property(lambda self: self.values.shape)
    dtype = property(lambda self: self.values.dtype)

    def with_default(self, default):
        return tf.where(self.mask, self.values, default)

    def __repr__(self):
        return masked_tensor_str(self.values, self.mask)

    class Spec:
        def __init__(self, shape, dtype=tf.float32):
            self.values = tf.TensorSpec(shape, dtype)
            self.mask = tf.TensorSpec(shape, tf.bool)

        shape = property(lambda self: self.values.shape)
        dtype = property(lambda self: self.values.dtype)

        def with_shape(self):
            return MaskedTensor.Spec(tf.TensorSpec(shape, self.values.dtype),
                                    tf.TensorSpec(shape, self.mask.dtype))
```

Next, the dispatch decorators are used to override the default behavior of several TensorFlow APIs. Since these APIs are used by standard Keras layers (such as the Dense layer), overriding these will allow us to use those layers with MaskedTensor. For the purposes of this example, matmul for masked tensors is defined to treat the masked values as zeros (i.e., to not include them in the product).

In [ ]:

```
@tf.experimental.dispatch_for_unary_elementwise_apis(MaskedTensor)
def unary_elementwise_op_handler(op, x):
    return MaskedTensor(op(x.values), x.mask)

@tf.experimental.dispatch_for_binary_elementwise_apis(
    Union[MaskedTensor, tf.Tensor],
    Union[MaskedTensor, tf.Tensor])
def binary_elementwise_op_handler(op, x, y):
    x = convert_to_masked_tensor(x)
    y = convert_to_masked_tensor(y)
    return MaskedTensor(op(x.values), y.values), x.mask & y.mask

@tf.experimental.dispatch_for_api(tf.matmul)
def masked_matmul(a: MaskedTensor, b,
                  transpose_a=False, transpose_b=False,
                  adjoint_a=False, adjoint_b=False,
                  a_is_sparse=False, b_is_sparse=False,
                  output_type=None):
    if isinstance(a, MaskedTensor):
        a = a.with_default(0)
    if isinstance(b, MaskedTensor):
        b = b.with_default(0)
    return tf.matmul(a, b, transpose_a, transpose_b, adjoint_a,
                    adjoint_b, a_is_sparse, b_is_sparse, output_type)
```

You can then construct a Keras model that accepts MaskedTensor inputs, using standard Keras layers:

In [ ]:

```
input_spec = MaskedTensor.Spec([None, 2], tf.float32)

masked_tensor_model = tf.keras.Sequential([
    tf.keras.layers.Input(type_spec=input_spec),
    tf.keras.layers.Dense(16, activation="relu"),
    tf.keras.layers.Dense(1)])
masked_tensor_model.compile(loss='binary_crossentropy', optimizer='rmsprop')
```

In [ ]:

```
a = MaskedTensor([[1., 2], [3, 4], [5, 6]],
                 [[True, False], [False, True], [True, True]])
masked_tensor_model.fit(a, tf.constant([[1], [0], [1]]), epochs=3)
print(masked_tensor_model(a))
```

## SavedModel

A [SavedModel](https://www.tensorflow.org/guide/saved_model) ([https://www.tensorflow.org/guide/saved\\_model](https://www.tensorflow.org/guide/saved_model)) is a serialized TensorFlow program, including both weights and computation. It can be built from a Keras model or from a custom model. In either case, extension types can be used transparently with the functions and methods defined by a SavedModel.

SavedModel can save models, layers, and functions that process extension types, as long as the extension types have a `__name__` field. This name is used to register the extension type, so it can be located when the model is loaded.

### Example: saving a Keras model

Keras models that use extension types may be saved using `SavedModel`.

In [ ]:

```
masked_tensor_model_path = tempfile.mkdtemp()
tf.saved_model.save(masked_tensor_model, masked_tensor_model_path)
imported_model = tf.saved_model.load(masked_tensor_model_path)
imported_model(a)
```

### Example: saving a custom model

SavedModel can also be used to save custom `tf.Module` subclasses with functions that process extension types.

In [ ]:

```
class CustomModule(tf.Module):
    def __init__(self, variable_value):
        super().__init__()
        self.v = tf.Variable(variable_value)

    @tf.function
    def grow(self, x: MaskedTensor):
        """Increase values in `x` by multiplying them by `self.v`."""
        return MaskedTensor(x.values * self.v, x.mask)

module = CustomModule(100.0)

module.grow.get_concrete_function(MaskedTensor.Spec(shape=None,
  dtype=tf.float32))
custom_module_path = tempfile.mkdtemp()
tf.saved_model.save(module, custom_module_path)
imported_model = tf.saved_model.load(custom_module_path)
imported_model.grow(MaskedTensor([1., 2, 3], [False, True, False]))
```

### Loading a SavedModel when the ExtensionType is unavailable

If you load a `SavedModel` that uses an `ExtensionType`, but that `ExtensionType` is not available (i.e., has not been imported), then you will see a warning and TensorFlow will fall back to using an "anonymous extension type" object. This object will have the same fields as the original type, but will lack any further customization you have added for the type, such as custom methods or properties.

### Using ExtensionTypes with TensorFlow serving

Currently, [TensorFlow serving](https://www.tensorflow.org/tfx/guide/serving) (<https://www.tensorflow.org/tfx/guide/serving>) (and other consumers of the SavedModel "signatures" dictionary) require that all inputs and outputs be raw tensors. If you wish to use TensorFlow serving with a model that uses extension types, then you can add wrapper methods that compose or decompose extension type values from tensors. E.g.:

In [ ]:

```
class CustomModuleWrapper(tf.Module):
    def __init__(self, variable_value):
        super().__init__()
        self.v = tf.Variable(variable_value)

    @tf.function
    def var_weighted_mean(self, x: MaskedTensor):
        """Mean value of unmasked values in x, weighted by self.v."""
        x = MaskedTensor(x.values * self.v, x.mask)
        return (tf.reduce_sum(x.with_default(0)) /
                tf.reduce_sum(tf.cast(x.mask, x.dtype)))

    @tf.function()
    def var_weighted_mean_wrapper(self, x_values, x_mask):
        """Raw tensor wrapper for var_weighted_mean."""
        return self.var_weighted_mean(MaskedTensor(x_values, x_mask))

module = CustomModuleWrapper([3., 2., 8., 5.])

module.var_weighted_mean_wrapper.get_concrete_function(
    tf.TensorSpec(None, tf.float32), tf.TensorSpec(None, tf.bool))
custom_module_path = tempfile.mkdtemp()
tf.saved_model.save(module, custom_module_path)
imported_model = tf.saved_model.load(custom_module_path)
x = MaskedTensor([1., 2., 3., 4.], [False, True, False, True])
imported_model.var_weighted_mean_wrapper(x.values, x.mask)
```

## Datasets

[tf.data](https://www.tensorflow.org/guide/data) (<https://www.tensorflow.org/guide/data>) is an API that enables you to build complex input pipelines from simple, reusable pieces. Its core data structure is `tf.data.Dataset`, which represents a sequence of elements, in which each element consists of one or more components.

### Building Datasets with extension types

Datasets can be built from extension type values using `Dataset.from_tensors`, `Dataset.from_tensor_slices`, or `Dataset.from_generator`:

In [ ]:

```
ds = tf.data.Dataset.from_tensors(Pastry(5, 5))
iter(ds).next()
```

In [ ]:

```
mt = MaskedTensor(tf.reshape(range(20), [5, 4]), tf.ones([5, 4]))
ds = tf.data.Dataset.from_tensor_slices(mt)
for value in ds:
    print(value)
```

In [ ]:

```
def value_gen():
    for i in range(2, 7):
        yield MaskedTensor(range(10), [j%i != 0 for j in range(10)])

ds = tf.data.Dataset.from_generator(
    value_gen, output_signature=MaskedTensor.Spec(shape=[10], dtype=tf.int32))
for value in ds:
    print(value)
```

### Batching and unbatching Datasets with extension types

Datasets with extension types can be batched and unbatched using `Dataset.batch` and `Dataset.unbatch`.

In [ ]:

```
batched_ds = ds.batch(2)
for value in batched_ds:
    print(value)
```

In [ ]:

```
unbatched_ds = batched_ds.unbatch()
for value in unbatched_ds:
    print(value)
```

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Introduction to tensor slicing



[View on TensorFlow.org](https://www.tensorflow.org/guide/tensor_slicing)

([https://www.tensorflow.org/guide/tensor\\_slicing](https://www.tensorflow.org/guide/tensor_slicing))



[Run in Google Colab](#)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/tensor\\_slicing.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/tensor_slicing.ipynb))

When working on ML applications such as object detection and NLP, it is sometimes necessary to work with sub-sections (slices) of tensors. For example, if your model architecture includes routing, where one layer might control which training example gets routed to the next layer. In this case, you could use tensor slicing ops to split the tensors up and put them back together in the right order.

In NLP applications, you can use tensor slicing to perform word masking while training. For example, you can generate training data from a list of sentences by choosing a word index to mask in each sentence, taking the word out as a label, and then replacing the chosen word with a mask token.

In this guide, you will learn how to use the TensorFlow APIs to:

- Extract slices from a tensor
- Insert data at specific indices in a tensor

This guide assumes familiarity with tensor indexing. Read the indexing sections of the [Tensor](https://www.tensorflow.org/guide/tensor#indexing) (<https://www.tensorflow.org/guide/tensor#indexing>) and [TensorFlow NumPy](https://www.tensorflow.org/guide/tf_numpy#indexing) ([https://www.tensorflow.org/guide/tf\\_numpy#indexing](https://www.tensorflow.org/guide/tf_numpy#indexing)) guides before getting started with this guide.

## Setup

In [ ]:

```
import tensorflow as tf
import numpy as np
```

## Extract tensor slices

Perform NumPy-like tensor slicing using `tf.slice`.

In [ ]:

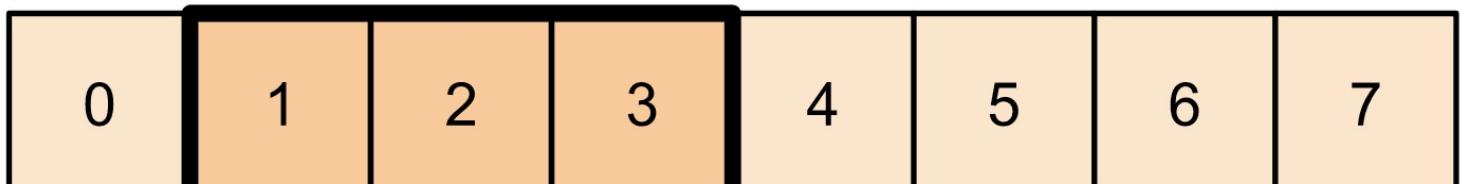
```
t1 = tf.constant([0, 1, 2, 3, 4, 5, 6, 7])

print(tf.slice(t1,
              begin=[1],
              size=[3]))
```

Alternatively, you can use a more Pythonic syntax. Note that tensor slices are evenly spaced over a start-stop range.

In [ ]:

```
print(t1[1:4])
```



In [ ]:

```
print(t1[-3:])
```

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

For 2-dimensional tensors, you can use something like:

In [ ]:

```
t2 = tf.constant([[0, 1, 2, 3, 4],  
                 [5, 6, 7, 8, 9],  
                 [10, 11, 12, 13, 14],  
                 [15, 16, 17, 18, 19]])
```

```
print(t2[:-1, 1:3])
```

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

You can use `tf.slice` on higher dimensional tensors as well.

In [ ]:

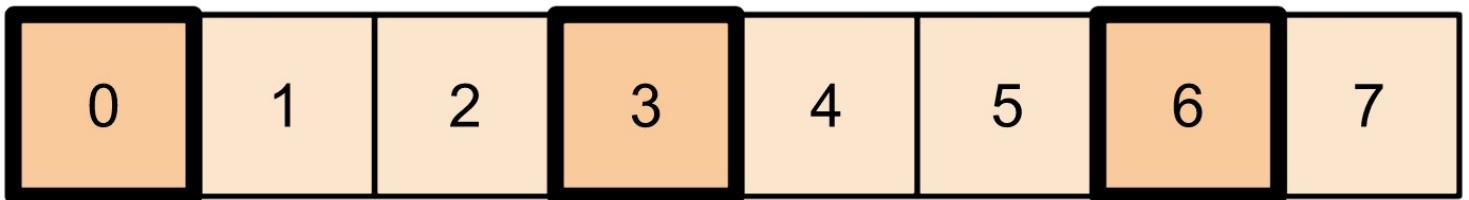
```
t3 = tf.constant([[[1, 3, 5, 7],  
                  [9, 11, 13, 15]],  
                  [[17, 19, 21, 23],  
                   [25, 27, 29, 31]]])  
  
print(tf.slice(t3,  
              begin=[1, 1, 0],  
              size=[1, 1, 2]))
```

You can also use `tf.strided_slice` to extract slices of tensors by 'striding' over the tensor dimensions.

Use `tf.gather` to extract specific indices from a single axis of a tensor.

In [ ]:

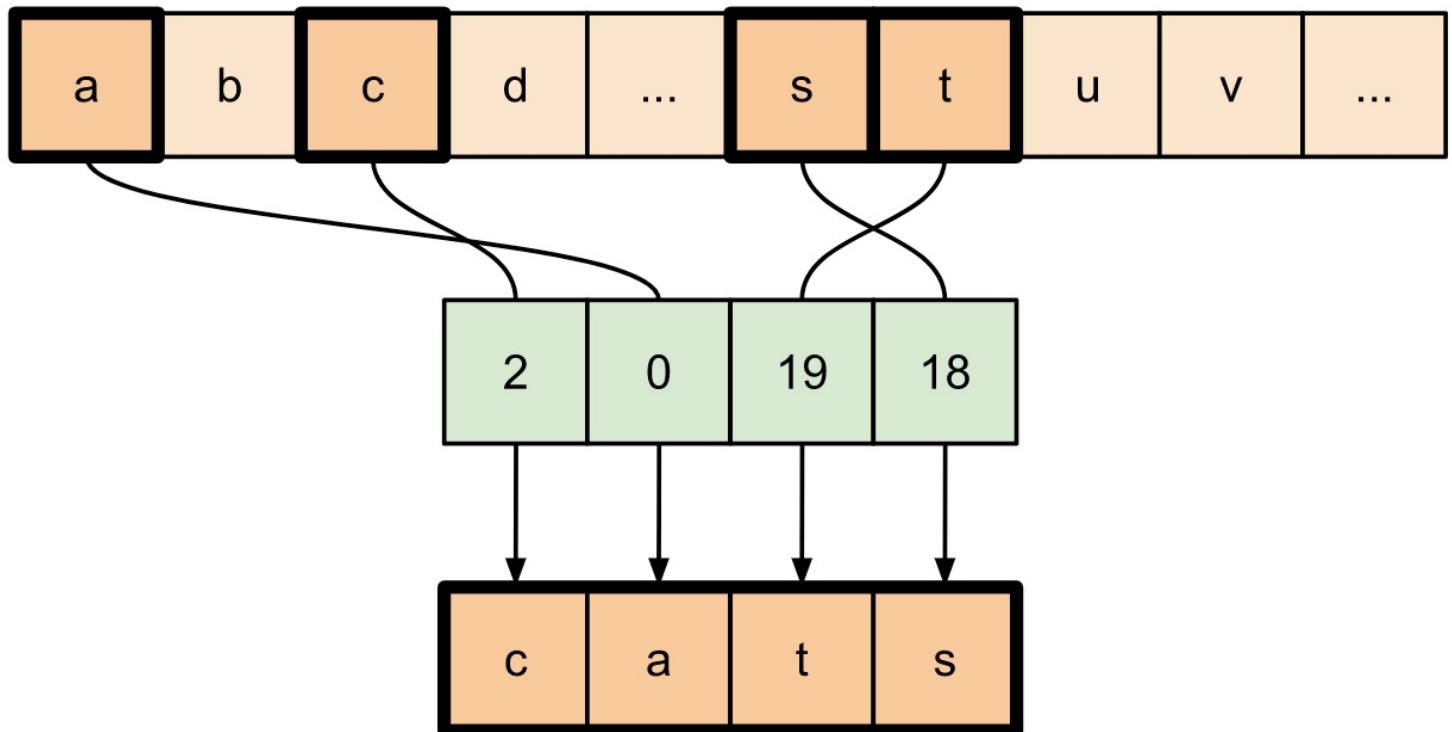
```
print(tf.gather(t1,  
                indices=[0, 3, 6]))  
  
# This is similar to doing  
t1[::3]
```



`tf.gather` does not require indices to be evenly spaced.

In [ ]:

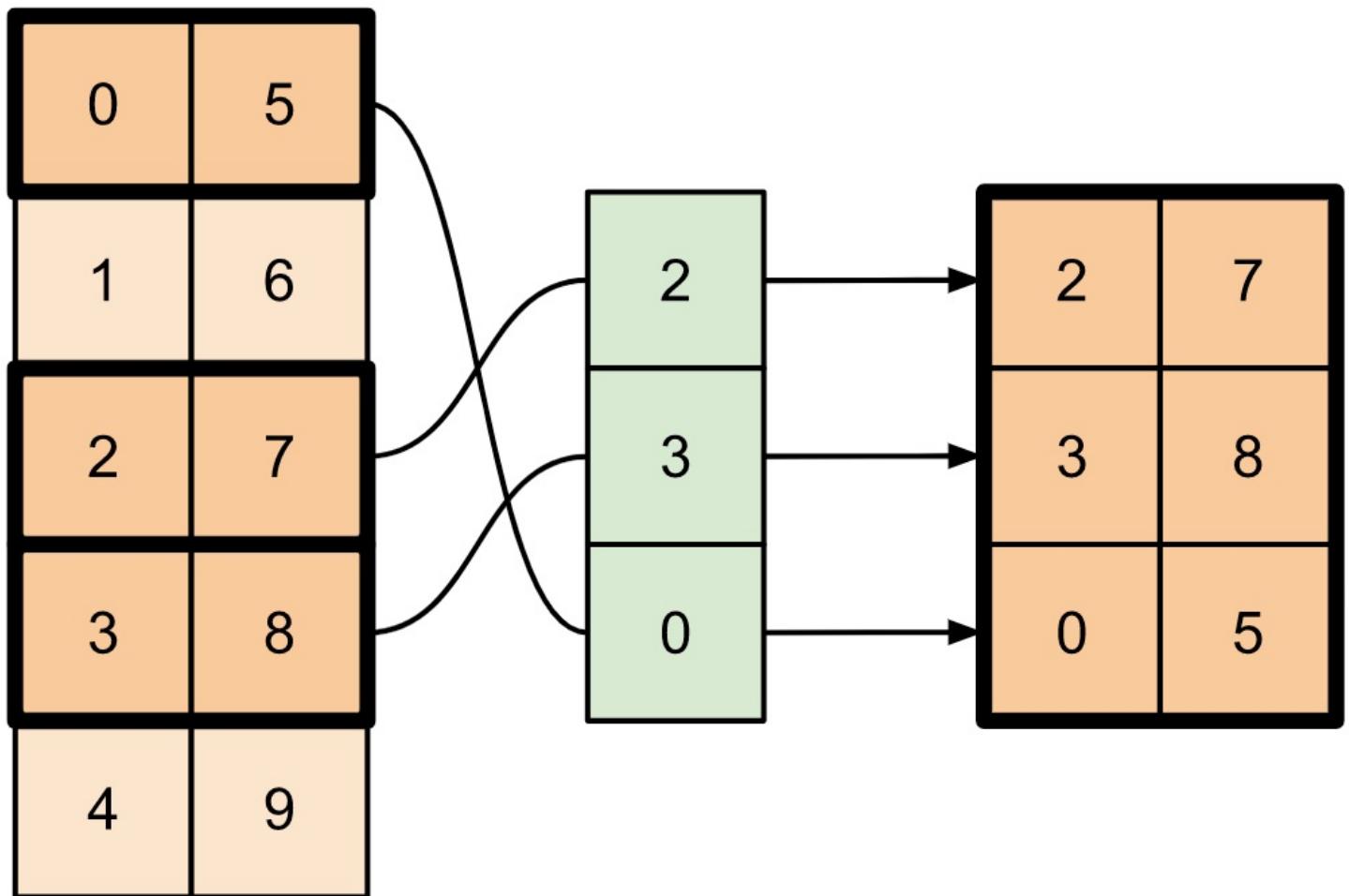
```
alphabet = tf.constant(list('abcdefghijklmnopqrstuvwxyz'))  
  
print(tf.gather(alphabet,  
                indices=[2, 0, 19, 18]))
```



To extract slices from multiple axes of a tensor, use `tf.gather_nd`. This is useful when you want to gather the elements of a matrix as opposed to just its rows or columns.

In [ ]:

```
t4 = tf.constant([[0, 5],  
                 [1, 6],  
                 [2, 7],  
                 [3, 8],  
                 [4, 9]])  
  
print(tf.gather_nd(t4,  
                   indices=[[2], [3], [0]]))
```



In [ ]:

```
t5 = np.reshape(np.arange(18), [2, 3, 3])  
  
print(tf.gather_nd(t5,  
                   indices=[[0, 0, 0], [1, 2, 1]]))
```

In [ ]:

```
# Return a list of two matrices  
  
print(tf.gather_nd(t5,  
                   indices=[[ [0, 0], [0, 2] ], [[1, 0], [1, 2]]]))
```

In [ ]:

```
# Return one matrix  
  
print(tf.gather_nd(t5,  
                   indices=[[0, 0], [0, 2], [1, 0], [1, 2]]))
```

## Insert data into tensors

Use `tf.scatter_nd` to insert data at specific slices/indices of a tensor. Note that the tensor into which you insert values is zero-initialized.

In [ ]:

```
t6 = tf.constant([10])
indices = tf.constant([[1], [3], [5], [7], [9]])
data = tf.constant([2, 4, 6, 8, 10])

print(tf.scatter_nd(indices=indices,
                    updates=data,
                    shape=t6))
```

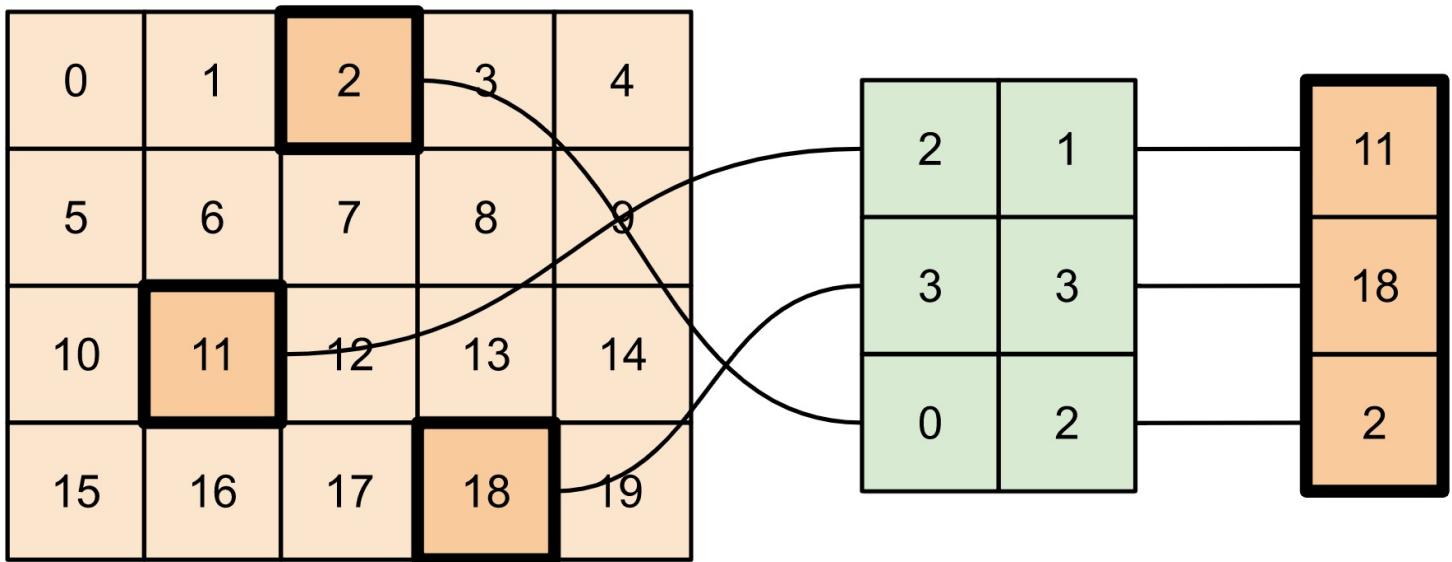
Methods like `tf.scatter_nd` which require zero-initialized tensors are similar to sparse tensor initializers. You can use `tf.gather_nd` and `tf.scatter_nd` to mimic the behavior of sparse tensor ops.

Consider an example where you construct a sparse tensor using these two methods in conjunction.

In [ ]:

```
# Gather values from one tensor by specifying indices

new_indices = tf.constant([[0, 2], [2, 1], [3, 3]])
t7 = tf.gather_nd(t2, indices=new_indices)
```



In [ ]:

```
# Add these values into a new tensor

t8 = tf.scatter_nd(indices=new_indices, updates=t7, shape=tf.constant([4, 5]))

print(t8)
```

This is similar to:

In [ ]:

```
t9 = tf.SparseTensor(indices=[[0, 2], [2, 1], [3, 3]],
                      values=[2, 11, 18],
                      dense_shape=[4, 5])

print(t9)
```

In [ ]:

```
# Convert the sparse tensor into a dense tensor

t10 = tf.sparse.to_dense(t9)

print(t10)
```

To insert data into a tensor with pre-existing values, use `tf.tensor_scatter_nd_add`.

In [ ]:

```
t11 = tf.constant([[2, 7, 0],  
                  [9, 0, 1],  
                  [0, 3, 8]])  
  
# Convert the tensor into a magic square by inserting numbers at appropriate indices  
  
t12 = tf.tensor_scatter_nd_add(t11,  
                                indices=[[0, 2], [1, 1], [2, 0]],  
                                updates=[6, 5, 4])  
  
print(t12)
```

Similarly, use `tf.tensor_scatter_nd_sub` to subtract values from a tensor with pre-existing values.

In [ ]:

```
# Convert the tensor into an identity matrix  
  
t13 = tf.tensor_scatter_nd_sub(t11,  
                                indices=[[0, 0], [0, 1], [1, 0], [1, 1], [1, 2], [2, 1], [2, 2]],  
                                updates=[1, 7, 9, -1, 1, 3, 7])  
  
print(t13)
```

Use `tf.tensor_scatter_nd_min` to copy element-wise minimum values from one tensor to another.

In [ ]:

```
t14 = tf.constant([[-2, -7, 0],  
                  [-9, 0, 1],  
                  [0, -3, -8]])  
  
t15 = tf.tensor_scatter_nd_min(t14,  
                                indices=[[0, 2], [1, 1], [2, 0]],  
                                updates=[-6, -5, -4])  
  
print(t15)
```

Similarly, use `tf.tensor_scatter_nd_max` to copy element-wise maximum values from one tensor to another.

In [ ]:

```
t16 = tf.tensor_scatter_nd_max(t14,  
                                indices=[[0, 2], [1, 1], [2, 0]],  
                                updates=[6, 5, 4])  
  
print(t16)
```

## Further reading and resources

In this guide, you learned how to use the tensor slicing ops available with TensorFlow to exert finer control over the elements in your tensors.

- Check out the slicing ops available with TensorFlow NumPy such as `tf.experimental.numpy.take_along_axis` and `tf.experimental.numpy.take`.
- Also check out the [Tensor guide](https://www.tensorflow.org/guide/tensor) (<https://www.tensorflow.org/guide/tensor>) and the [Variable guide](https://www.tensorflow.org/guide/variable) (<https://www.tensorflow.org/guide/variable>).

**Copyright 2018 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");  
# you may not use this file except in compliance with the License.  
# You may obtain a copy of the License at  
#  
# https://www.apache.org/licenses/LICENSE-2.0  
#  
# Unless required by applicable law or agreed to in writing, software  
# distributed under the License is distributed on an "AS IS" BASIS,  
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
# See the License for the specific language governing permissions and  
# limitations under the License.
```

# Ragged tensors



[View on TensorFlow.org](https://www.tensorflow.org/guide/ragged_tensor)

([https://www.tensorflow.org/guide/ragged\\_tensor](https://www.tensorflow.org/guide/ragged_tensor))



[Run in Google Colab](#)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/ragged\\_tensor.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/ragged_tensor.ipynb))

**API Documentation:** [tf.RaggedTensor](https://www.tensorflow.org/api_docs/python/tf/RaggedTensor) ([https://www.tensorflow.org/api\\_docs/python/tf/RaggedTensor](https://www.tensorflow.org/api_docs/python/tf/RaggedTensor)) [tf.ragged](https://www.tensorflow.org/api_docs/python/tf.ragged) ([https://www.tensorflow.org/api\\_docs/python/tf.ragged](https://www.tensorflow.org/api_docs/python/tf.ragged))

## Setup

In [ ]:

```
import math
import tensorflow as tf
```

## Overview

Your data comes in many shapes; your tensors should too. *Ragged tensors* are the TensorFlow equivalent of nested variable-length lists. They make it easy to store and process data with non-uniform shapes, including:

- Variable-length features, such as the set of actors in a movie.
- Batches of variable-length sequential inputs, such as sentences or video clips.
- Hierarchical inputs, such as text documents that are subdivided into sections, paragraphs, sentences, and words.
- Individual fields in structured inputs, such as protocol buffers.

## What you can do with a ragged tensor

Ragged tensors are supported by more than a hundred TensorFlow operations, including math operations (such as `tf.add` and `tf.reduce_mean`), array operations (such as `tf.concat` and `tf.tile`), string manipulation ops (such as `tf.substr`), control flow operations (such as `tf.while_loop` and `tf.map_fn`), and many others:

In [ ]:

```
digits = tf.ragged.constant([[3, 1, 4, 1], [], [5, 9, 2], [6], []])
words = tf.ragged.constant([["So", "long"], ["thanks", "for", "all", "the", "fish"]])
print(tf.add(digits, 3))
print(tf.reduce_mean(digits, axis=1))
print(tf.concat([digits, [[5, 3]]], axis=0))
print(tf.tile(digits, [1, 2]))
print(tf.strings.substr(words, 0, 2))
print(tf.map_fn(tf.math.square, digits))
```

There are also a number of methods and operations that are specific to ragged tensors, including factory methods, conversion methods, and value-mapping operations. For a list of supported ops, see the [tf.ragged package documentation](#).

Ragged tensors are supported by many TensorFlow APIs, including [Keras](https://www.tensorflow.org/guide/keras) (<https://www.tensorflow.org/guide/keras>), [Datasets](https://www.tensorflow.org/guide/datasets) (<https://www.tensorflow.org/guide/datasets>), [tf.function](https://www.tensorflow.org/guide/function) (<https://www.tensorflow.org/guide/function>), [SavedModels](https://www.tensorflow.org/guide/saved_model) ([https://www.tensorflow.org/guide/saved\\_model](https://www.tensorflow.org/guide/saved_model)), and [tf.Example](https://www.tensorflow.org/tutorials/load_data/tfrecord) ([https://www.tensorflow.org/tutorials/load\\_data/tfrecord](https://www.tensorflow.org/tutorials/load_data/tfrecord)). For more information, check the section on [TensorFlow APIs](#) below.

As with normal tensors, you can use Python-style indexing to access specific slices of a ragged tensor. For more information, refer to the section on [Indexing](#) below.

In [ ]:

```
print(digits[0])      # First row
```

In [ ]:

```
print(digits[:, :2])  # First two values in each row.
```

In [ ]:

```
print(digits[:, -2:]) # Last two values in each row.
```

And just like normal tensors, you can use Python arithmetic and comparison operators to perform elementwise operations. For more information, check the section on **Overloaded operators** below.

In [ ]:

```
print(digits + 3)
```

In [ ]:

```
print(digits + tf.ragged.constant([[1, 2, 3, 4], [], [5, 6, 7], [8], []]))
```

If you need to perform an elementwise transformation to the values of a `RaggedTensor`, you can use `tf.ragged.map_flat_values`, which takes a function plus one or more arguments, and applies the function to transform the `RaggedTensor`'s values.

In [ ]:

```
times_two_plus_one = lambda x: x * 2 + 1
print(tf.ragged.map_flat_values(times_two_plus_one, digits))
```

Ragged tensors can be converted to nested Python `list`s and NumPy `array`s:

In [ ]:

```
digits.to_list()
```

In [ ]:

```
digits.numpy()
```

## Constructing a ragged tensor

The simplest way to construct a ragged tensor is using `tf.ragged.constant`, which builds the `RaggedTensor` corresponding to a given nested Python `list` or NumPy `array`:

In [ ]:

```
sentences = tf.ragged.constant([
    ["Let's", "build", "some", "ragged", "tensors", "!"],
    ["We", "can", "use", "tf.ragged.constant", "."]])
print(sentences)
```

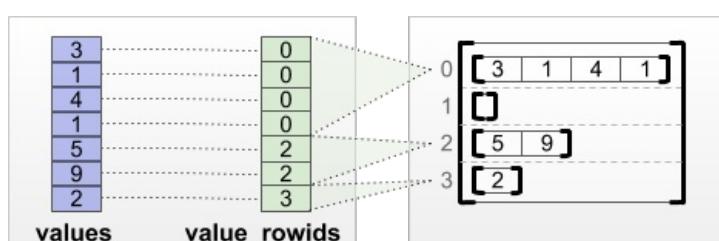
In [ ]:

```
paragraphs = tf.ragged.constant([
    [['I', 'have', 'a', 'cat'], ['His', 'name', 'is', 'Mat']],
    [['Do', 'you', 'want', 'to', 'come', 'visit'], ["I'm", 'free', 'tomorrow']]],
])
print(paragraphs)
```

Ragged tensors can also be constructed by pairing flat `values` tensors with `row-partitioning` tensors indicating how those values should be divided into rows, using factory classmethods such as `tf.RaggedTensor.from_value_rowids`, `tf.RaggedTensor.from_row_lengths`, and `tf.RaggedTensor.from_row_splits`.

### `tf.RaggedTensor.from_value_rowids`

If you know which row each value belongs to, then you can build a `RaggedTensor` using a `value_rowids` row-partitioning tensor:

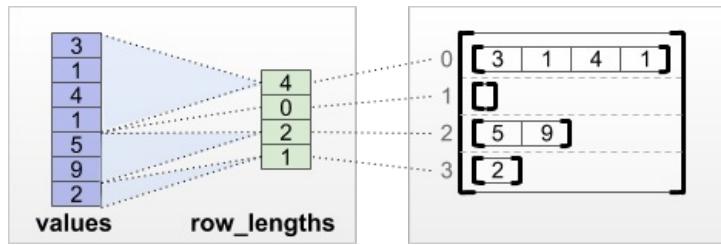


In [ ]:

```
print(tf.RaggedTensor.from_value_rowids(
    values=[3, 1, 4, 1, 5, 9, 2],
    value_rowids=[0, 0, 0, 0, 2, 2, 3]))
```

## `tf.RaggedTensor.from_row_lengths`

If you know how long each row is, then you can use a `row_lengths` row-partitioning tensor:

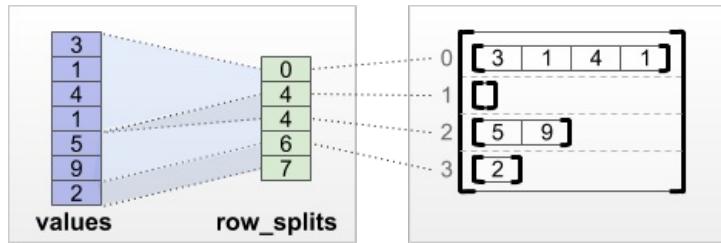


In [ ]:

```
print(tf.RaggedTensor.from_row_lengths(  
    values=[3, 1, 4, 1, 5, 9, 2],  
    row_lengths=[4, 0, 2, 1]))
```

## `tf.RaggedTensor.from_row_splits`

If you know the index where each row starts and ends, then you can use a `row_splits` row-partitioning tensor:



In [ ]:

```
print(tf.RaggedTensor.from_row_splits(  
    values=[3, 1, 4, 1, 5, 9, 2],  
    row_splits=[0, 4, 4, 6, 7]))
```

See the `tf.RaggedTensor` class documentation for a full list of factory methods.

Note: By default, these factory methods add assertions that the row partition tensor is well-formed and consistent with the number of values. The `validate=False` parameter can be used to skip these checks if you can guarantee that the inputs are well-formed and consistent.

## What you can store in a ragged tensor

As with normal `Tensor`s, the values in a `RaggedTensor` must all have the same type; and the values must all be at the same nesting depth (the `rank` of the tensor):

In [ ]:

```
print(tf.ragged.constant([[["Hi"], ["How", "are", "you"]]])) # ok: type=string, rank=2
```

In [ ]:

```
print(tf.ragged.constant([[1, 2], [3]], [[4, 5]]))) # ok: type=int32, rank=3
```

In [ ]:

```
try:  
    tf.ragged.constant([["one", "two"], [3, 4]]) # bad: multiple types  
except ValueError as exception:  
    print(exception)
```

In [ ]:

```
try:  
    tf.ragged.constant(["A", ["B", "C"]]) # bad: multiple nesting depths  
except ValueError as exception:  
    print(exception)
```

## Example use case

The following example demonstrates how `RaggedTensor`s can be used to construct and combine unigram and bigram embeddings for a batch of variable-length queries, using special markers for the beginning and end of each sentence. For more details on the ops used in this example, check the `tf.ragged` package documentation.

In [ ]:

```
queries = tf.ragged.constant([['Who', 'is', 'Dan', 'Smith'],
                             ['Pause'],
                             ['Will', 'it', 'rain', 'later', 'today']])

# Create an embedding table.
num_buckets = 1024
embedding_size = 4
embedding_table = tf.Variable(
    tf.random.truncated_normal([num_buckets, embedding_size],
                               stddev=1.0 / math.sqrt(embedding_size)))

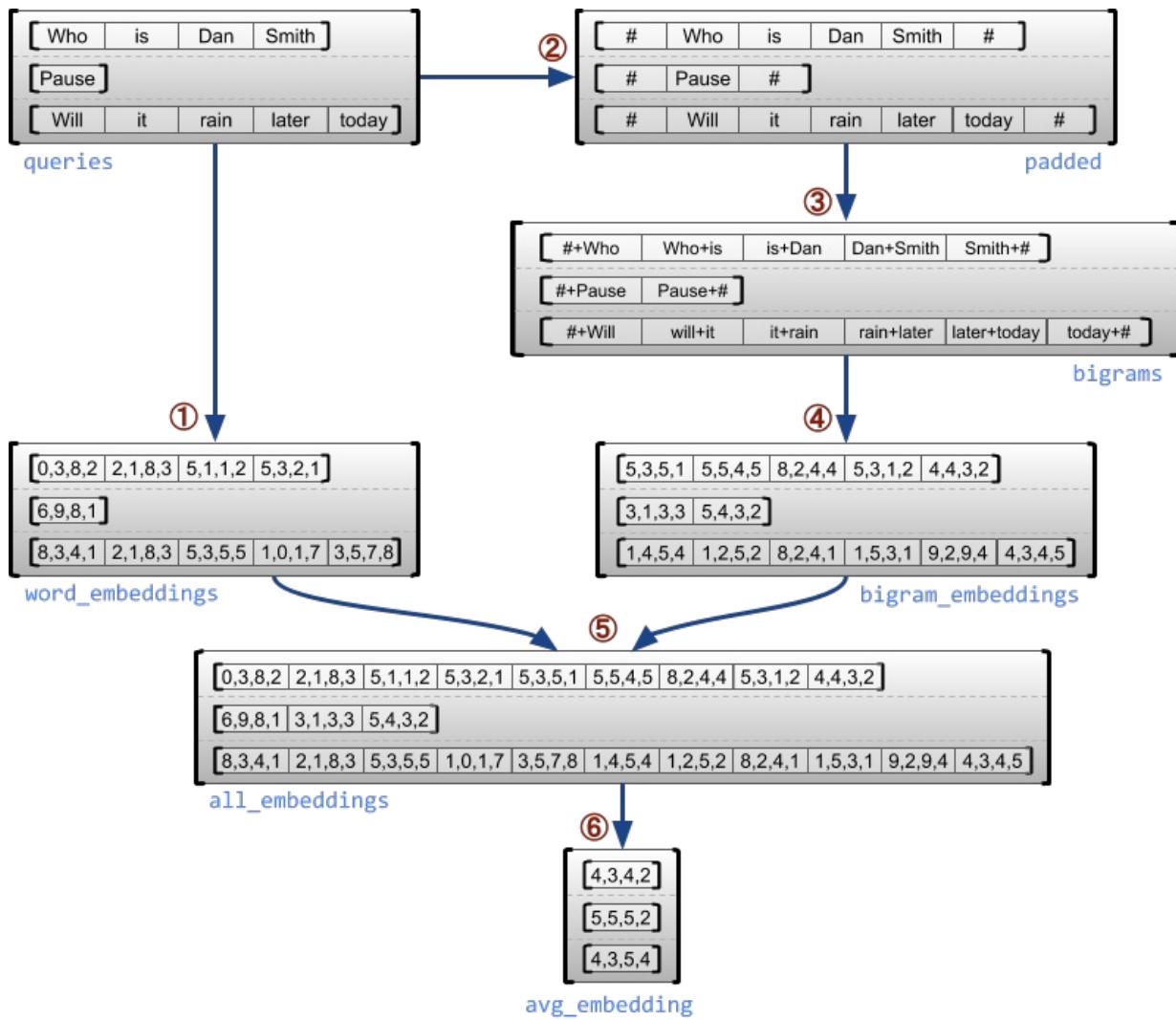
# Look up the embedding for each word.
word_buckets = tf.strings.to_hash_bucket_fast(queries, num_buckets)
word_embeddings = tf.nn.embedding_lookup(embedding_table, word_buckets) # ①

# Add markers to the beginning and end of each sentence.
marker = tf.fill([queries.nrows(), 1], '#')
padded = tf.concat([marker, queries, marker], axis=1) # ②

# Build word bigrams and look up embeddings.
bigrams = tf.strings.join([padded[:, :-1], padded[:, 1:]], separator='+') # ③

bigram_buckets = tf.strings.to_hash_bucket_fast(bigrams, num_buckets)
bigram_embeddings = tf.nn.embedding_lookup(embedding_table, bigram_buckets) # ④

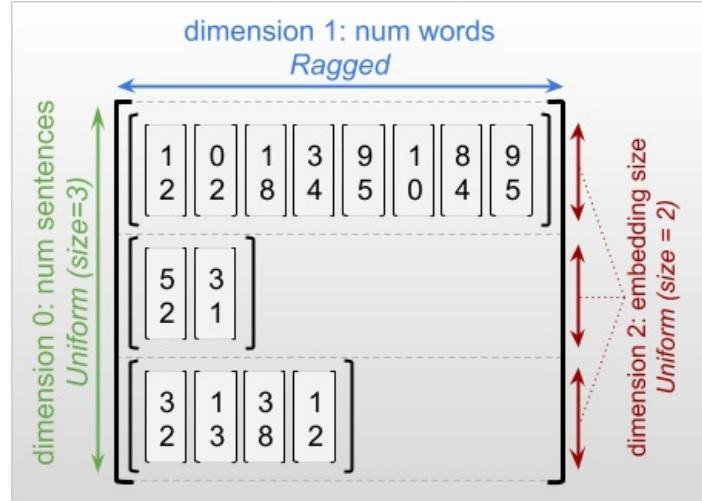
# Find the average embedding for each sentence
all_embeddings = tf.concat([word_embeddings, bigram_embeddings], axis=1) # ⑤
avg_embedding = tf.reduce_mean(all_embeddings, axis=1) # ⑥
print(avg_embedding)
```



## Ragged and uniform dimensions

A **ragged dimension** is a dimension whose slices may have different lengths. For example, the inner (column) dimension of `rt=[[3, 1, 4, 1], [], [5, 9, 2], [6], []]` is ragged, since the column slices (`rt[0, :]`, ..., `rt[4, :]`) have different lengths. Dimensions whose slices all have the same length are called *uniform dimensions*.

The outermost dimension of a ragged tensor is always uniform, since it consists of a single slice (and, therefore, there is no possibility for differing slice lengths). The remaining dimensions may be either ragged or uniform. For example, you may store the word embeddings for each word in a batch of sentences using a ragged tensor with shape `[num_sentences, (num_words), embedding_size]`, where the parentheses around `(num_words)` indicate that the dimension is ragged.



Ragged tensors may have multiple ragged dimensions. For example, you could store a batch of structured text documents using a tensor with shape `[num_documents, (num_paragraphs), (num_sentences), (num_words)]` (where again parentheses are used to indicate ragged dimensions).

As with `tf.Tensor`, the **rank** of a ragged tensor is its total number of dimensions (including both ragged and uniform dimensions). A **potentially ragged tensor** is a value that might be either a `tf.Tensor` or a `tf.RaggedTensor`.

When describing the shape of a `RaggedTensor`, ragged dimensions are conventionally indicated by enclosing them in parentheses. For example, as you saw above, the shape of a 3D `RaggedTensor` that stores word embeddings for each word in a batch of sentences can be written as `[num_sentences, (num_words), embedding_size]`.

The `RaggedTensor.shape` attribute returns a `tf.TensorShape` for a ragged tensor where ragged dimensions have size `None`:

In [ ]:

```
tf.ragged.constant([["Hi"], ["How", "are", "you"]]).shape
```

The method `tf.RaggedTensor.bounding_shape` can be used to find a tight bounding shape for a given `RaggedTensor`:

In [ ]:

```
print(tf.ragged.constant([["Hi"], ["How", "are", "you"]]).bounding_shape())
```

## Ragged vs sparse

A ragged tensor should *not* be thought of as a type of sparse tensor. In particular, sparse tensors are *efficient encodings for* `tf.Tensor` that model the same data in a compact format; but ragged tensor is an *extension to* `tf.Tensor` that models an expanded class of data. This difference is crucial when defining operations:

- Applying an op to a sparse or dense tensor should always give the same result.
- Applying an op to a ragged or sparse tensor may give different results.

As an illustrative example, consider how array operations such as `concat`, `stack`, and `tile` are defined for ragged vs. sparse tensors.

Concatenating ragged tensors joins each row to form a single row with the combined length:

$$\text{ragged\_concat} \left( \left[ \begin{bmatrix} ["John"], \\ ["a", "big", "dog"], \\ ["my", "cat"], \end{bmatrix}, \begin{bmatrix} ["fell", "asleep"], \\ ["barked"], \\ ["is", "fuzzy"], \end{bmatrix} \right], \text{axis}=1 \right) = \begin{bmatrix} ["John", "fell", "asleep"], \\ ["a", "big", "dog", "barked"], \\ ["my", "cat", "is", "fuzzy"], \end{bmatrix}$$

In [ ]:

```
ragged_x = tf.ragged.constant([["John"], ["a", "big", "dog"], ["my", "cat"]])
ragged_y = tf.ragged.constant([["fell", "asleep"], ["barked"], ["is", "fuzzy"]])
print(tf.concat([ragged_x, ragged_y], axis=1))
```

However, concatenating sparse tensors is equivalent to concatenating the corresponding dense tensors, as illustrated by the following example (where  $\emptyset$  indicates missing values):

$$\text{sparse\_concat} \left( \begin{bmatrix} ["John", \emptyset, \emptyset] \\ ["a", "big", "dog"] \\ ["my", "cat", \emptyset] \end{bmatrix}, \begin{bmatrix} ["fell", "asleep"] \\ ["barked", \emptyset] \\ ["is", "fuzzy"] \end{bmatrix}, \text{axis}=1 \right) = \begin{bmatrix} ["John", \emptyset, \emptyset, "fell", "asleep"] \\ ["a", "big", "dog", "barked", \emptyset] \\ ["my", "cat", \emptyset, "is", "fuzzy"] \end{bmatrix}$$

In [ ]:

```
sparse_x = ragged_x.to_sparse()
sparse_y = ragged_y.to_sparse()
sparse_result = tf.sparse.concat(sp_inputs=[sparse_x, sparse_y], axis=1)
print(tf.sparse.to_dense(sparse_result, ''))
```

For another example of why this distinction is important, consider the definition of “the mean value of each row” for an op such as `tf.reduce_mean`. For a ragged tensor, the mean value for a row is the sum of the row’s values divided by the row’s width. But for a sparse tensor, the mean value for a row is the sum of the row’s values divided by the sparse tensor’s overall width (which is greater than or equal to the width of the longest row).

## TensorFlow APIs

### Keras

`tf.keras` (<https://www.tensorflow.org/guide/keras>) is TensorFlow’s high-level API for building and training deep learning models. Ragged tensors may be passed as inputs to a Keras model by setting `ragged=True` on `tf.keras.Input` or `tf.keras.layers.InputLayer`. Ragged tensors may also be passed between Keras layers, and returned by Keras models. The following example shows a toy LSTM model that is trained using ragged tensors.

In [ ]:

```
# Task: predict whether each sentence is a question or not.
sentences = tf.constant([
    'What makes you think she is a witch?',
    'She turned me into a newt.',
    'A newt?',
    'Well, I got better.'])
is_question = tf.constant([True, False, True, False])

# Preprocess the input strings.
hash_buckets = 1000
words = tf.strings.split(sentences, ' ')
hashed_words = tf.strings.to_hash_bucket_fast(words, hash_buckets)

# Build the Keras model.
keras_model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=[None], dtype=tf.int64, ragged=True),
    tf.keras.layers.Embedding(hash_buckets, 16),
    tf.keras.layers.LSTM(32, use_bias=False),
    tf.keras.layers.Dense(32),
    tf.keras.layers.Activation(tf.nn.relu),
    tf.keras.layers.Dense(1)
])

keras_model.compile(loss='binary_crossentropy', optimizer='rmsprop')
keras_model.fit(hashed_words, is_question, epochs=5)
print(keras_model.predict(hashed_words))
```

### tf.Example

`tf.Example` ([https://www.tensorflow.org/tutorials/load\\_data/tfrecord](https://www.tensorflow.org/tutorials/load_data/tfrecord)) is a standard `protobuf` (<https://developers.google.com/protocol-buffers/>) encoding for TensorFlow data. Data encoded with `tf.Example`s often includes variable-length features. For example, the following code defines a batch of four `tf.Example` messages with different feature lengths:

In [ ]:

```
import google.protobuf.text_format as pbtext

def build_tf_example(s):
    return pbtext.Merge(s, tf.train.Example()).SerializeToString()

example_batch = [
    build_tf_example(r"""
        features {
            feature {key: "colors" value {bytes_list {value: ["red", "blue"]}} }
            feature {key: "lengths" value {int64_list {value: [7]}} }
        }"""),
    build_tf_example(r"""
        features {
            feature {key: "colors" value {bytes_list {value: ["orange"]}} }
            feature {key: "lengths" value {int64_list {value: []}} }
        }"""),
    build_tf_example(r"""
        features {
            feature {key: "colors" value {bytes_list {value: ["black", "yellow"]}} }
            feature {key: "lengths" value {int64_list {value: [1, 3]}} }
        }"""),
    build_tf_example(r"""
        features {
            feature {key: "colors" value {bytes_list {value: ["green"]}} }
            feature {key: "lengths" value {int64_list {value: [3, 5, 2]}} }
        }"""),
]
```

You can parse this encoded data using `tf.io.parse_example`, which takes a tensor of serialized strings and a feature specification dictionary, and returns a dictionary mapping feature names to tensors. To read the variable-length features into ragged tensors, you simply use `tf.io.RaggedFeature` in the feature specification dictionary:

In [ ]:

```
feature_specification = {
    'colors': tf.io.RaggedFeature(tf.string),
    'lengths': tf.io.RaggedFeature(tf.int64),
}
feature_tensors = tf.io.parse_example(example_batch, feature_specification)
for name, value in feature_tensors.items():
    print("{}={}".format(name, value))
```

`tf.io.RaggedFeature` can also be used to read features with multiple ragged dimensions. For details, refer to the [API documentation](#) ([https://www.tensorflow.org/api\\_docs/python/tf/io/RaggedFeature](https://www.tensorflow.org/api_docs/python/tf/io/RaggedFeature)).

## Datasets

`tf.data` (<https://www.tensorflow.org/guide/data>) is an API that enables you to build complex input pipelines from simple, reusable pieces. Its core data structure is `tf.data.Dataset`, which represents a sequence of elements, in which each element consists of one or more components.

In [ ]:

```
# Helper function used to print datasets in the examples below.
def print_dictionary_dataset(dataset):
    for i, element in enumerate(dataset):
        print("Element {}:".format(i))
        for (feature_name, feature_value) in element.items():
            print('{:>14} = {}'.format(feature_name, feature_value))
```

### Building Datasets with ragged tensors

Datasets can be built from ragged tensors using the same methods that are used to build them from `tf.Tensors` or NumPy arrays, such as `Dataset.from_tensor_slices`:

In [ ]:

```
dataset = tf.data.Dataset.from_tensor_slices(feature_tensors)
print_dictionary_dataset(dataset)
```

Note: `Dataset.from_generator` does not support ragged tensors yet, but support will be added soon.

### Batching and unbatching Datasets with ragged tensors

Datasets with ragged tensors can be batched (which combines  $n$  consecutive elements into a single elements) using the `Dataset.batch` method.

In [ ]:

```
batched_dataset = dataset.batch(2)
print_dictionary_dataset(batched_dataset)
```

Conversely, a batched dataset can be transformed into a flat dataset using `Dataset.unbatch`.

In [ ]:

```
unbatched_dataset = batched_dataset.unbatch()
print_dictionary_dataset(unbatched_dataset)
```

### Batching Datasets with variable-length non-ragged tensors

If you have a Dataset that contains non-ragged tensors, and tensor lengths vary across elements, then you can batch those non-ragged tensors into ragged tensors by applying the `dense_to_ragged_batch` transformation:

In [ ]:

```
non_ragged_dataset = tf.data.Dataset.from_tensor_slices([1, 5, 3, 2, 8])
non_ragged_dataset = non_ragged_dataset.map(tf.range)
batched_non_ragged_dataset = non_ragged_dataset.apply(
    tf.data.experimental.dense_to_ragged_batch(2))
for element in batched_non_ragged_dataset:
    print(element)
```

### Transforming Datasets with ragged tensors

You can also create or transform ragged tensors in Datasets using `Dataset.map`:

In [ ]:

```
def transform_lengths(features):
    return {
        'mean_length': tf.math.reduce_mean(features['lengths']),
        'length_ranges': tf.ragged.range(features['lengths'])}
transformed_dataset = dataset.map(transform_lengths)
print_dictionary_dataset(transformed_dataset)
```

## tf.function

[tf.function](https://www.tensorflow.org/guide/function) (<https://www.tensorflow.org/guide/function>) is a decorator that precomputes TensorFlow graphs for Python functions, which can substantially improve the performance of your TensorFlow code. Ragged tensors can be used transparently with `@tf.function`-decorated functions. For example, the following function works with both ragged and non-ragged tensors:

In [ ]:

```
@tf.function
def make_palindrome(x, axis):
    return tf.concat([x, tf.reverse(x, [axis])], axis)
```

In [ ]:

```
make_palindrome(tf.constant([[1, 2], [3, 4], [5, 6]]), axis=1)
```

In [ ]:

```
make_palindrome(tf.ragged.constant([[1, 2], [3], [4, 5, 6]]), axis=1)
```

If you wish to explicitly specify the `input_signature` for the `tf.function`, then you can do so using `tf.RaggedTensorSpec`.

In [ ]:

```
@tf.function(
    input_signature=[tf.RaggedTensorSpec(shape=[None, None], dtype=tf.int32)])
def max_and_min(rt):
    return (tf.math.reduce_max(rt, axis=-1), tf.math.reduce_min(rt, axis=-1))

max_and_min(tf.ragged.constant([[1, 2], [3], [4, 5, 6]]))
```

## Concrete functions

Concrete functions ([https://www.tensorflow.org/guide/function#obtaining\\_concrete\\_functions](https://www.tensorflow.org/guide/function#obtaining_concrete_functions)) encapsulate individual traced graphs that are built by `tf.function`. Ragged tensors can be used transparently with concrete functions.

In [ ]:

```
@tf.function
def increment(x):
    return x + 1

rt = tf.ragged.constant([[1, 2], [3], [4, 5, 6]])
cf = increment.get_concrete_function(rt)
print(cf(rt))
```

## SavedModels

A [SavedModel](https://www.tensorflow.org/guide/saved_model) ([https://www.tensorflow.org/guide/saved\\_model](https://www.tensorflow.org/guide/saved_model)) is a serialized TensorFlow program, including both weights and computation. It can be built from a Keras model or from a custom model. In either case, ragged tensors can be used transparently with the functions and methods defined by a SavedModel.

### Example: saving a Keras model

In [ ]:

```
import tempfile

keras_module_path = tempfile.mkdtemp()
tf.saved_model.save(keras_model, keras_module_path)
imported_model = tf.saved_model.load(keras_module_path)
imported_model(hashed_words)
```

### Example: saving a custom model

In [ ]:

```
class CustomModule(tf.Module):
    def __init__(self, variable_value):
        super(CustomModule, self).__init__()
        self.v = tf.Variable(variable_value)

    @tf.function
    def grow(self, x):
        return x * self.v

module = CustomModule(100.0)

# Before saving a custom model, you must ensure that concrete functions are
# built for each input signature that you will need.
module.grow.get_concrete_function(tf.RaggedTensorSpec(shape=[None, None],
   dtype=tf.float32))

custom_module_path = tempfile.mkdtemp()
tf.saved_model.save(module, custom_module_path)
imported_model = tf.saved_model.load(custom_module_path)
imported_model.grow(tf.ragged.constant([[1.0, 4.0, 3.0], [2.0]]))
```

Note: SavedModel [signatures](https://www.tensorflow.org/guide/saved_model#specifying_signatures_during_export) ([https://www.tensorflow.org/guide/saved\\_model#specifying\\_signatures\\_during\\_export](https://www.tensorflow.org/guide/saved_model#specifying_signatures_during_export)) are concrete functions. As discussed in the section on Concrete Functions above, ragged tensors are only handled correctly by concrete functions starting with TensorFlow 2.3. If you need to use SavedModel signatures in a previous version of TensorFlow, then it's recommended that you decompose the ragged tensor into its component tensors.

## Overloaded operators

The `RaggedTensor` class overloads the standard Python arithmetic and comparison operators, making it easy to perform basic elementwise math:

In [ ]:

```
x = tf.ragged.constant([[1, 2], [3], [4, 5, 6]])
y = tf.ragged.constant([[1, 1], [2], [3, 3, 3]])
print(x + y)
```

Since the overloaded operators perform elementwise computations, the inputs to all binary operations must have the same shape or be broadcastable to the same shape. In the simplest broadcasting case, a single scalar is combined elementwise with each value in a ragged tensor:

In [ ]:

```
x = tf.ragged.constant([[1, 2], [3], [4, 5, 6]])
print(x + 3)
```

For a discussion of more advanced cases, check the section on [Broadcasting](#).

Ragged tensors overload the same set of operators as normal `Tensor`s: the unary operators `-`, `~`, and `abs()`; and the binary operators `+`, `-`, `*`, `/`, `//`, `%`, `**`, `&`, `|`, `^`, `==`, `<`, `<=`, `>`, and `>=`.

## Indexing

Ragged tensors support Python-style indexing, including multidimensional indexing and slicing. The following examples demonstrate ragged tensor indexing with a 2D and a 3D ragged tensor.

### Indexing examples: 2D ragged tensor

In [ ]:

```
queries = tf.ragged.constant(
    [['Who', 'is', 'George', 'Washington'],
     ['What', 'is', 'the', 'weather', 'tomorrow'],
     ['Goodnight']])
```

In [ ]:

```
print(queries[1])          # A single query
```

In [ ]:

```
print(queries[1, 2])      # A single word
```

In [ ]:

```
print(queries[1:])        # Everything but the first row
```

In [ ]:

```
print(queries[:, :3])     # The first 3 words of each query
```

In [ ]:

```
print(queries[:, -2:])    # The last 2 words of each query
```

### Indexing examples: 3D ragged tensor

In [ ]:

```
rt = tf.ragged.constant([[[1, 2, 3], [4]],
                        [[5], [], [6]],
                        [[7]],
                        [[8, 9], [10]]])
```

In [ ]:

```
print(rt[1])            # Second row (2D RaggedTensor)
```

In [ ]:

```
print(rt[3, 0])         # First element of fourth row (1D Tensor)
```

In [ ]:

```
print(rt[:, 1:3])       # Items 1-3 of each row (3D RaggedTensor)
```

In [ ]:

```
print(rt[:, -1:])        # Last item of each row (3D RaggedTensor)
```

RaggedTensor's support multidimensional indexing and slicing with one restriction: indexing into a ragged dimension is not allowed. This case is problematic because the indicated value may exist in some rows but not others. In such cases, it's not obvious whether you should (1) raise an IndexError; (2) use a default value; or (3) skip that value and return a tensor with fewer rows than you started with. Following the [guiding principles of Python](https://www.python.org/dev/peps/pep-0020/) ("In the face of ambiguity, refuse the temptation to guess"), this operation is currently disallowed.

## Tensor type conversion

The RaggedTensor class defines methods that can be used to convert between RaggedTensor's and tf.Tensors or tf.SparseTensors:

In [ ]:

```
ragged_sentences = tf.ragged.constant([
    ['Hi'], ['Welcome', 'to', 'the', 'fair'], ['Have', 'fun']])
```

In [ ]:

```
# RaggedTensor -> Tensor
print(ragged_sentences.to_tensor(default_value=' ', shape=[None, 10]))
```

In [ ]:

```
# Tensor -> RaggedTensor
x = [[1, 3, -1, -1], [2, -1, -1, -1], [4, 5, 8, 9]]
print(tf.RaggedTensor.from_tensor(x, padding=-1))
```

In [ ]:

```
#RaggedTensor -> SparseTensor
print(ragged_sentences.to_sparse())
```

In [ ]:

```
# SparseTensor -> RaggedTensor
st = tf.SparseTensor(indices=[[0, 0], [2, 0], [2, 1]],
                      values=['a', 'b', 'c'],
                      dense_shape=[3, 3])
print(tf.RaggedTensor.from_sparse(st))
```

## Evaluating ragged tensors

To access the values in a ragged tensor, you can:

1. Use `tf.RaggedTensor.to_list` to convert the ragged tensor to a nested Python list.
2. Use `tf.RaggedTensor.numpy` to convert the ragged tensor to a NumPy array whose values are nested NumPy arrays.
3. Decompose the ragged tensor into its components, using the `tf.RaggedTensor.values` and `tf.RaggedTensor.row_splits` properties, or row-partitioning methods such as `tf.RaggedTensor.row_lengths` and `tf.RaggedTensor.value_rowids`.
4. Use Python indexing to select values from the ragged tensor.

In [ ]:

```
rt = tf.ragged.constant([[1, 2], [3, 4, 5], [6], [], [7]])
print("Python list:", rt.to_list())
print("NumPy array:", rt.numpy())
print("Values:", rt.values.numpy())
print("Splits:", rt.row_splits.numpy())
print("Indexed value:", rt[1].numpy())
```

## Broadcasting

Broadcasting is the process of making tensors with different shapes have compatible shapes for elementwise operations. For more background on broadcasting, refer to:

- [NumPy: Broadcasting](https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html) (<https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html>)
- `tf.broadcast_dynamic_shape`
- `tf.broadcast_to`

The basic steps for broadcasting two inputs `x` and `y` to have compatible shapes are:

1. If `x` and `y` do not have the same number of dimensions, then add outer dimensions (with size 1) until they do.
2. For each dimension where `x` and `y` have different sizes:
  - If `x` or `y` have size 1 in dimension `d`, then repeat its values across dimension `d` to match the other input's size.
  - Otherwise, raise an exception (`x` and `y` are not broadcast compatible).

Where the size of a tensor in a uniform dimension is a single number (the size of slices across that dimension); and the size of a tensor in a ragged dimension is a list of slice lengths (for all slices across that dimension).

## Broadcasting examples

In [ ]:

```
# x      (2D ragged): 2 x (num_rows)
# y      (scalar)
# result (2D ragged): 2 x (num_rows)
x = tf.ragged.constant([[1, 2], [3]])
y = 3
print(x + y)
```

In [ ]:

```
# x      (2d ragged): 3 x (num_rows)
# y      (2d tensor): 3 x 1
# Result (2d ragged): 3 x (num_rows)
x = tf.ragged.constant(
    [[10, 87, 12],
     [19, 53],
     [12, 32]])
y = [[1000], [2000], [3000]]
print(x + y)
```

In [ ]:

```
# x      (3d ragged): 2 x (r1) x 2
# y      (2d ragged): 1 x 1
# Result (3d ragged): 2 x (r1) x 2
x = tf.ragged.constant(
    [[[1, 2], [3, 4], [5, 6]],
     [[7, 8]]],
    ragged_rank=1)
y = tf.constant([[10]])
print(x + y)
```

In [ ]:

```
# x      (3d ragged): 2 x (r1) x (r2) x 1
# y      (1d tensor): 3
# Result (3d ragged): 2 x (r1) x (r2) x 3
x = tf.ragged.constant(
    [
        [
            [[1], [2]],
            [],
            [[3]],
            [[4]],
        ],
        [
            [[5], [6]],
            [[7]]
        ]
    ],
    ragged_rank=2)
y = tf.constant([10, 20, 30])
print(x + y)
```

Here are some examples of shapes that do not broadcast:

In [ ]:

```
# x      (2d ragged): 3 x (r1)
# y      (2d tensor): 3 x 4 # trailing dimensions do not match
x = tf.ragged.constant([[1, 2], [3, 4, 5, 6], [7]])
y = tf.constant([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
try:
    x + y
except tf.errors.InvalidArgumentError as exception:
    print(exception)
```

In [ ]:

```
# x      (2d ragged): 3 x (r1)
# y      (2d ragged): 3 x (r2) # ragged dimensions do not match.
x = tf.ragged.constant([[1, 2, 3], [4], [5, 6]])
y = tf.ragged.constant([[10, 20], [30, 40], [50]])
try:
    x + y
except tf.errors.InvalidArgumentError as exception:
    print(exception)
```

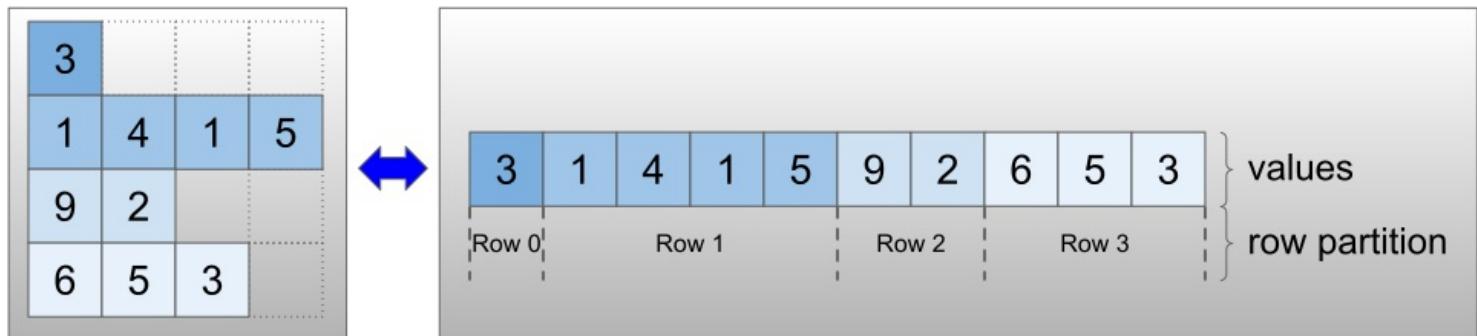
In [ ]:

```
# x      (3d ragged): 3 x (r1) x 2
# y      (3d ragged): 3 x (r1) x 3 # trailing dimensions do not match
x = tf.ragged.constant([[[1, 2], [3, 4], [5, 6]],
                      [[7, 8], [9, 10]]])
y = tf.ragged.constant([[[[1, 2, 0], [3, 4, 0], [5, 6, 0]],
                        [[7, 8, 0], [9, 10, 0]]]])
try:
    x + y
except tf.errors.InvalidArgumentError as exception:
    print(exception)
```

## RaggedTensor encoding

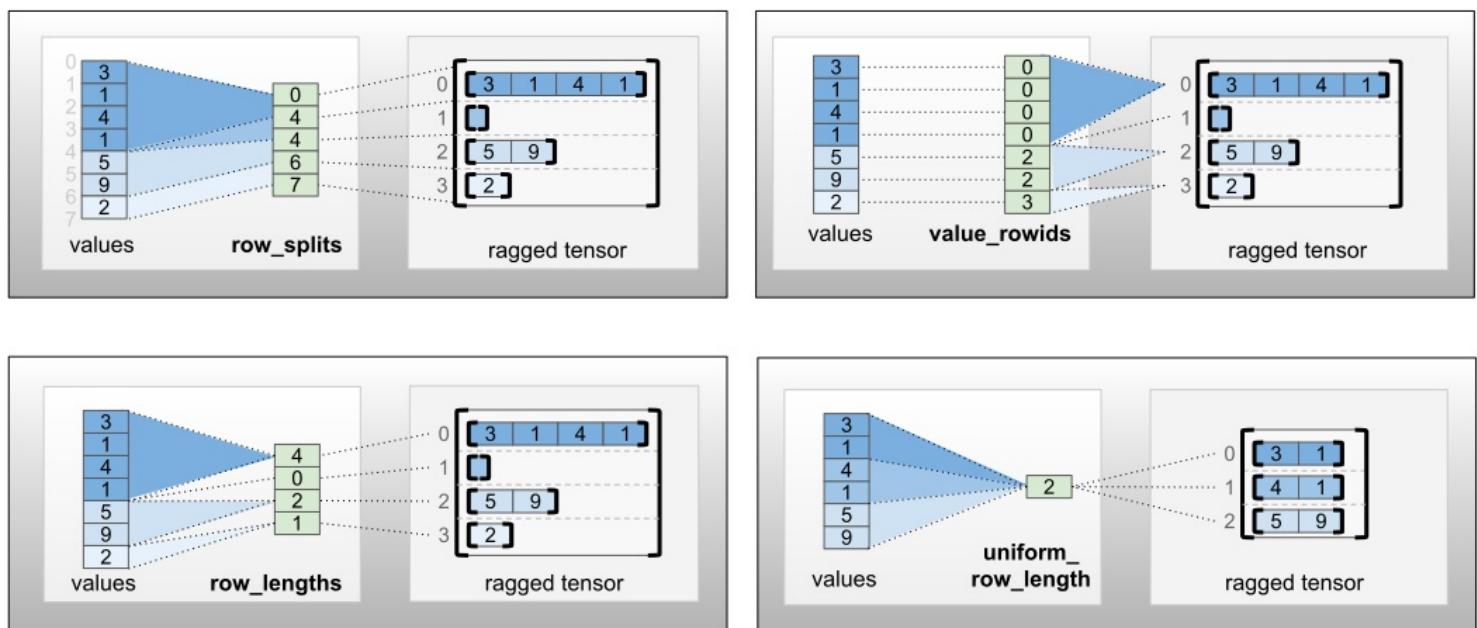
Ragged tensors are encoded using the `RaggedTensor` class. Internally, each `RaggedTensor` consists of:

- A `values` tensor, which concatenates the variable-length rows into a flattened list.
- A `row_partition`, which indicates how those flattened values are divided into rows.



The `row_partition` can be stored using four different encodings:

- `row_splits` is an integer vector specifying the split points between rows.
- `value_rowids` is an integer vector specifying the row index for each value.
- `row_lengths` is an integer vector specifying the length of each row.
- `uniform_row_length` is an integer scalar specifying a single length for all rows.



An integer scalar `nrows` can also be included in the `row_partition` encoding to account for empty trailing rows with `value_rowids` or empty rows with `uniform_row_length`.

In [ ]:

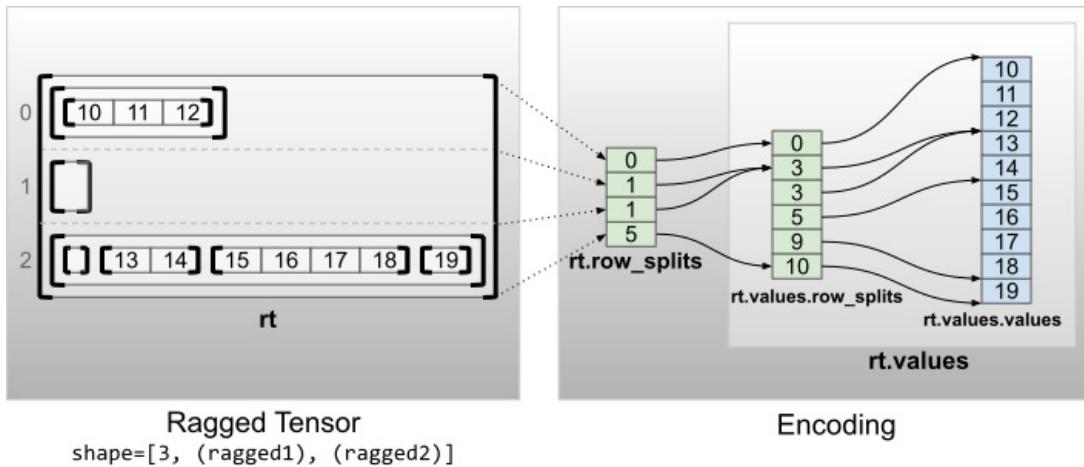
```
rt = tf.RaggedTensor.from_row_splits(  
    values=[3, 1, 4, 1, 5, 9, 2],  
    row_splits=[0, 4, 4, 6, 7])  
print(rt)
```

The choice of which encoding to use for row partitions is managed internally by ragged tensors to improve efficiency in some contexts. In particular, some of the advantages and disadvantages of the different row-partitioning schemes are:

- **Efficient indexing:** The `row_splits` encoding enables constant-time indexing and slicing into ragged tensors.
- **Efficient concatenation:** The `row_lengths` encoding is more efficient when concatenating ragged tensors, since row lengths do not change when two tensors are concatenated together.
- **Small encoding size:** The `value_rowids` encoding is more efficient when storing ragged tensors that have a large number of empty rows, since the size of the tensor depends only on the total number of values. On the other hand, the `row_splits` and `row_lengths` encodings are more efficient when storing ragged tensors with longer rows, since they require only one scalar value for each row.
- **Compatibility:** The `value_rowids` scheme matches the [segmentation \(\[https://www.tensorflow.org/api\\\_docs/python/tf/math#about\\\_segmentation\]\(https://www.tensorflow.org/api\_docs/python/tf/math#about\_segmentation\)\)](https://www.tensorflow.org/api_docs/python/tf/math#about_segmentation) format used by operations, such as `tf.segment_sum`. The `row_limits` scheme matches the format used by ops such as `tf.sequence_mask`.
- **Uniform dimensions:** As discussed below, the `uniform_row_length` encoding is used to encode ragged tensors with uniform dimensions.

## Multiple ragged dimensions

A ragged tensor with multiple ragged dimensions is encoded by using a nested `RaggedTensor` for the `values` tensor. Each nested `RaggedTensor` adds a single ragged dimension.



In [ ]:

```
rt = tf.RaggedTensor.from_row_splits(
    values=tf.RaggedTensor.from_row_splits(
        values=[10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
        row_splits=[0, 3, 3, 5, 9, 10]),
    row_splits=[0, 1, 1, 5])
print(rt)
print("Shape: {}".format(rt.shape))
print("Number of partitioned dimensions: {}".format(rt.ragged_rank))
```

The factory function `tf.RaggedTensor.from_nested_row_splits` may be used to construct a `RaggedTensor` with multiple ragged dimensions directly by providing a list of `row_splits` tensors:

In [ ]:

```
rt = tf.RaggedTensor.from_nested_row_splits(
    flat_values=[10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
    nested_row_splits=([0, 1, 1, 5], [0, 3, 3, 5, 9, 10]))
print(rt)
```

## Ragged rank and flat values

A ragged tensor's **ragged rank** is the number of times that the underlying `values` tensor has been partitioned (i.e. the nesting depth of `RaggedTensor` objects). The innermost `values` tensor is known as its **flat\_values**. In the following example, `conversations` has `ragged_rank=3`, and its `flat_values` is a 1D Tensor with 24 strings:

In [ ]:

```
# shape = [batch, (paragraph), (sentence), (word)]
conversations = tf.ragged.constant(
    [[[["I", "like", "ragged", "tensors."]],
      [["Oh", "yeah?"], ["What", "can", "you", "use", "them", "for?"]],
      [["Processing", "variable", "length", "data!"]]],
     [[[["I", "like", "cheese."], ["Do", "you?"]]],
      [["Yes."], ["I", "do."]]]])
conversations.shape
```

In [ ]:

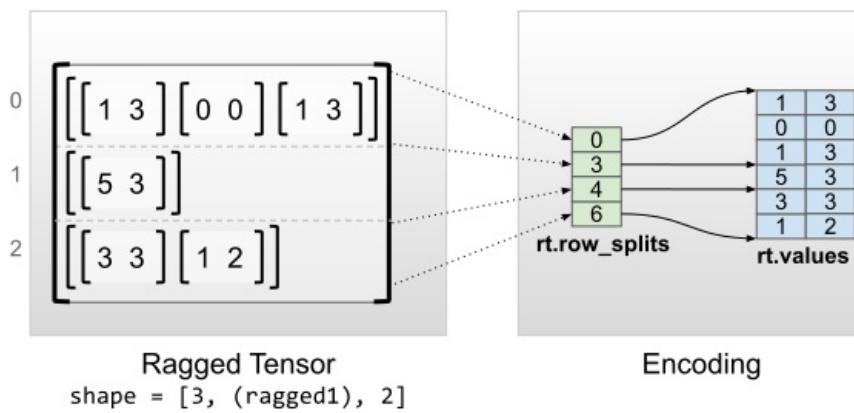
```
assert conversations.ragged_rank == len(conversations.nested_row_splits)
conversations.ragged_rank # Number of partitioned dimensions.
```

In [ ]:

```
conversations.flat_values.numpy()
```

## Uniform inner dimensions

Ragged tensors with uniform inner dimensions are encoded by using a multidimensional `tf.Tensor` for the `flat_values` (i.e., the innermost `values`).

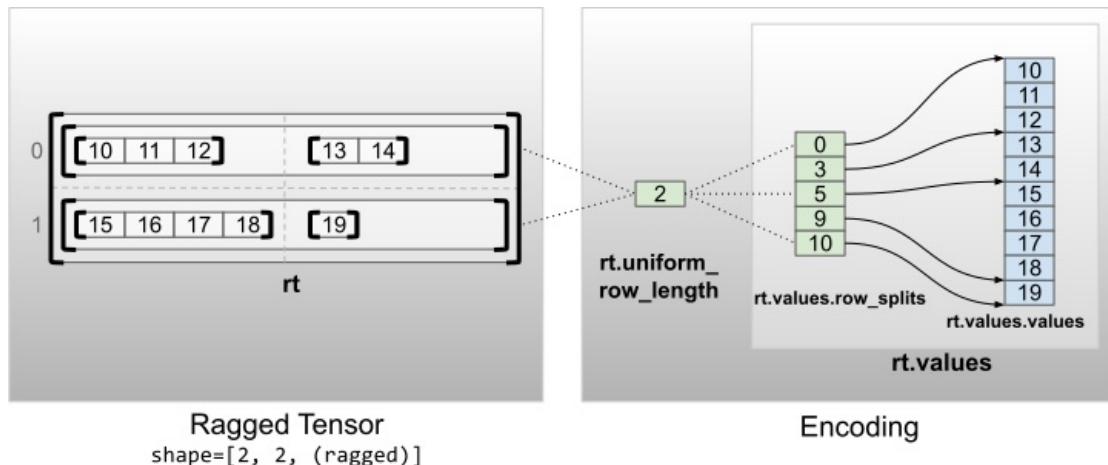


In [ ]:

```
rt = tf.RaggedTensor.from_row_splits(  
    values=[[1, 3], [0, 0], [1, 3], [5, 3], [3, 3], [1, 2]],  
    row_splits=[0, 3, 4, 6])  
print(rt)  
print("Shape: {}".format(rt.shape))  
print("Number of partitioned dimensions: {}".format(rt.ragged_rank))  
print("Flat values shape: {}".format(rt.flat_values.shape))  
print("Flat values:\n{}".format(rt.flat_values))
```

## Uniform non-inner dimensions

Ragged tensors with uniform non-inner dimensions are encoded by partitioning rows with `uniform_row_length`.



In [ ]:

```
rt = tf.RaggedTensor.from_uniform_row_length(  
    values=tf.RaggedTensor.from_row_splits(  
        values=[10, 11, 12, 13, 14, 15, 16, 17, 18, 19],  
        row_splits=[0, 3, 5, 9, 10]),  
    uniform_row_length=2)  
print(rt)  
print("Shape: {}".format(rt.shape))  
print("Number of partitioned dimensions: {}".format(rt.ragged_rank))
```

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Using the SavedModel format



[View on TensorFlow.org](https://www.tensorflow.org/guide/saved_model)

([https://www.tensorflow.org/guide/saved\\_model](https://www.tensorflow.org/guide/saved_model))



[Run in Google Colab](#)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/saved\\_model.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/saved_model.ipynb))

([https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/saved\\_model](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/saved_model))

A SavedModel contains a complete TensorFlow program, including trained parameters (i.e., `tf.Variable`s) and computation. It does not require the original model building code to run, which makes it useful for sharing or deploying with [TFLite](#) (<https://tensorflow.org/lite>), [TensorFlow.js](#) (<https://js.tensorflow.org/>), [TensorFlow Serving](#) ([https://www.tensorflow.org/tfx/serving/tutorials/Serving\\_REST\\_simple](https://www.tensorflow.org/tfx/serving/tutorials/Serving_REST_simple)), or [TensorFlow Hub](#) (<https://tensorflow.org/hub>).

You can save and load a model in the SavedModel format using the following APIs:

- Low-level `tf.saved_model` API. This document describes how to use this API in detail.
  - Save: `tf.saved_model.save(model, path_to_dir)`
  - Load: `model = tf.saved_model.load(path_to_dir)`
- High-level `tf.keras.Model` API. Refer to [the keras save and serialize guide](#) ([https://www.tensorflow.org/guide/keras/save\\_and\\_serialize](https://www.tensorflow.org/guide/keras/save_and_serialize)).
- If you just want to save/load weights during training, refer to [the checkpoints guide](#) ([./checkpoint.ipynb](#)).

## Creating a SavedModel from Keras

For a quick introduction, this section exports a pre-trained Keras model and serves image classification requests with it. The rest of the guide will fill in details and discuss other ways to create SavedModels.

In [ ]:

```
import os
import tempfile

from matplotlib import pyplot as plt
import numpy as np
import tensorflow as tf

tmpdir = tempfile.mkdtemp()
```

In [ ]:

```
physical_devices = tf.config.list_physical_devices('GPU')
for device in physical_devices:
    tf.config.experimental.set_memory_growth(device, True)
```

In [ ]:

```
file = tf.keras.utils.get_file(
    "grace_hopper.jpg",
    "https://storage.googleapis.com/download.tensorflow.org/example_images/grace_hopper.jpg")
img = tf.keras.utils.load_img(file, target_size=[224, 224])
plt.imshow(img)
plt.axis('off')
x = tf.keras.utils.img_to_array(img)
x = tf.keras.applications.mobilenet.preprocess_input(
    x[tf.newaxis,...])
```

You'll use an image of Grace Hopper as a running example, and a Keras pre-trained image classification model since it's easy to use. Custom models work too, and are covered in detail later.

In [ ]:

```
labels_path = tf.keras.utils.get_file(  
    'ImageNetLabels.txt',  
    'https://storage.googleapis.com/download.tensorflow.org/data/ImageNetLabels.txt')  
imagenet_labels = np.array(open(labels_path).read().splitlines())
```

In [ ]:

```
pretrained_model = tf.keras.applications.MobileNet()  
result_before_save = pretrained_model(x)  
  
decoded = imagenet_labels[np.argsort(result_before_save)[0,:-1][:5]+1]  
  
print("Result before saving:\n", decoded)
```

The top prediction for this image is "military uniform".

In [ ]:

```
mobilenet_save_path = os.path.join(tmpdir, "mobilenet/1/")  
tf.saved_model.save(pretrained_model, mobilenet_save_path)
```

The save-path follows a convention used by TensorFlow Serving where the last path component ( 1/ here) is a version number for your model - it allows tools like Tensorflow Serving to reason about the relative freshness.

You can load the SavedModel back into Python with `tf.saved_model.load` and see how Admiral Hopper's image is classified.

In [ ]:

```
loaded = tf.saved_model.load(mobilenet_save_path)  
print(list(loaded.signatures.keys())) # ["serving_default"]
```

Imported signatures always return dictionaries. To customize signature names and output dictionary keys, see [Specifying signatures during export](#).

In [ ]:

```
infer = loaded.signatures["serving_default"]  
print(infer.structured_outputs)
```

Running inference from the SavedModel gives the same result as the original model.

In [ ]:

```
labeling = infer(tf.constant(x))[pretrained_model.output_names[0]]  
  
decoded = imagenet_labels[np.argsort(labeling)[0,:-1][:5]+1]  
  
print("Result after saving and loading:\n", decoded)
```

## Running a SavedModel in TensorFlow Serving

SavedModels are usable from Python (more on that below), but production environments typically use a dedicated service for inference without running Python code. This is easy to set up from a SavedModel using TensorFlow Serving.

See the [TensorFlow Serving REST tutorial](#) ([https://www.tensorflow.org/tfx/tutorials/serving/rest\\_simple](https://www.tensorflow.org/tfx/tutorials/serving/rest_simple)) for an end-to-end tensorflow-serving example.

## The SavedModel format on disk

A SavedModel is a directory containing serialized signatures and the state needed to run them, including variable values and vocabularies.

In [ ]:

```
!ls {mobilenet_save_path}
```

The `saved_model.pb` file stores the actual TensorFlow program, or model, and a set of named signatures, each identifying a function that accepts tensor inputs and produces tensor outputs.

SavedModels may contain multiple variants of the model (multiple `v1.MetaGraphDefs`, identified with the `--tag_set` flag to `saved_model_cli`), but this is rare. APIs which create multiple variants of a model include `tf.Estimator.experimental_export_all_saved_models` ([https://www.tensorflow.org/api\\_docs/python/tf/estimator/Estimator#experimental\\_export\\_all\\_saved\\_models](https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator#experimental_export_all_saved_models)) and in TensorFlow 1.x `tf.saved_model.Builder`.

In [ ]:

```
!saved_model_cli show --dir {mobilenet_save_path} --tag_set serve
```

The variables directory contains a standard training checkpoint (see the [guide to training checkpoints \(./checkpoint.ipynb\)](#)).

In [ ]:

```
!ls {mobilenet_save_path}/variables
```

The assets directory contains files used by the TensorFlow graph, for example text files used to initialize vocabulary tables. It is unused in this example.

SavedModels may have an assets.extra directory for any files not used by the TensorFlow graph, for example information for consumers about what to do with the SavedModel. TensorFlow itself does not use this directory.

## Saving a custom model

`tf.saved_model.save` supports saving `tf.Module` objects and its subclasses, like `tf.keras.Layer` and `tf.keras.Model`.

Let's look at an example of saving and restoring a `tf.Module`.

In [ ]:

```
class CustomModule(tf.Module):
    def __init__(self):
        super(CustomModule, self).__init__()
        self.v = tf.Variable(1.)

    @tf.function
    def __call__(self, x):
        print('Tracing with', x)
        return x * self.v

    @tf.function(input_signature=[tf.TensorSpec([], tf.float32)])
    def mutate(self, new_v):
        self.v.assign(new_v)

module = CustomModule()
```

When you save a `tf.Module`, any `tf.Variable` attributes, `tf.function`-decorated methods, and `tf.Module`s found via recursive traversal are saved. (See the [Checkpoint tutorial \(./checkpoint.ipynb\)](#) for more about this recursive traversal.) However, any Python attributes, functions, and data are lost. This means that when a `tf.function` is saved, no Python code is saved.

If no Python code is saved, how does SavedModel know how to restore the function?

Briefly, `tf.function` works by tracing the Python code to generate a `ConcreteFunction` (a callable wrapper around `tf.Graph`). When saving a `tf.function`, you're really saving the `tf.function`'s cache of `ConcreteFunctions`.

To learn more about the relationship between `tf.function` and `ConcreteFunctions`, refer to the [tf.function guide \(function.ipynb\)](#).

In [ ]:

```
module_no_signatures_path = os.path.join(tmpdir, 'module_no_signatures')
module(tf.constant(0.))
print('Saving model...')
tf.saved_model.save(module, module_no_signatures_path)
```

## Loading and using a custom model

When you load a SavedModel in Python, all `tf.Variable` attributes, `tf.function`-decorated methods, and `tf.Module`s are restored in the same object structure as the original saved `tf.Module`.

In [ ]:

```
imported = tf.saved_model.load(module_no_signatures_path)
assert imported(tf.constant(3.)).numpy() == 3
imported.mutate(tf.constant(2.))
assert imported(tf.constant(3.)).numpy() == 6
```

Because no Python code is saved, calling a `tf.function` with a new input signature will fail:

```
imported(tf.constant([3.]))  
  
ValueError: Could not find matching function to call for canonicalized inputs ((,), {}). Only existing signatures are [((TensorSpec(shape=(), dtype=tf.float32, name=u'x')),), {})].
```

## Basic fine-tuning

Variable objects are available, and you can backprop through imported functions. That is enough to fine-tune (i.e. retrain) a SavedModel in simple cases.

In [ ]:

```
optimizer = tf.optimizers.SGD(0.05)  
  
def train_step():  
    with tf.GradientTape() as tape:  
        loss = (10. - imported(tf.constant(2.))) ** 2  
    variables = tape.watched_variables()  
    grads = tape.gradient(loss, variables)  
    optimizer.apply_gradients(zip(grads, variables))  
    return loss
```

In [ ]:

```
for _ in range(10):  
    # "v" approaches 5, "loss" approaches 0  
    print("loss={:.2f} v={:.2f}".format(train_step(), imported.v.numpy()))
```

## General fine-tuning

A SavedModel from Keras provides [more details \(<https://github.com/tensorflow/community/blob/master/rfcs/20190509-keras-saved-model.md#serialization-details>\)](https://github.com/tensorflow/community/blob/master/rfcs/20190509-keras-saved-model.md#serialization-details) than a plain `__call__` to address more advanced cases of fine-tuning. TensorFlow Hub recommends to provide the following of those, if applicable, in SavedModels shared for the purpose of fine-tuning:

- If the model uses dropout or another technique in which the forward pass differs between training and inference (like batch normalization), the `__call__` method takes an optional, Python-valued `training` argument that defaults to `False` but can be set to `True`.
- Next to the `__call__` attribute, there are `.variable` and `.trainable_variable` attributes with the corresponding lists of variables. A variable that was originally trainable but is meant to be frozen during fine-tuning is omitted from `.trainable_variables`.
- For the sake of frameworks like Keras that represent weight regularizers as attributes of layers or sub-models, there can also be a `.regularization_losses` attribute. It holds a list of zero-argument functions whose values are meant for addition to the total loss.

Going back to the initial MobileNet example, you can see some of those in action:

In [ ]:

```
loaded = tf.saved_model.load(mobilenet_save_path)  
print("MobileNet has {} trainable variables: {}, ...".format(  
    len(loaded.trainable_variables),  
    ", ".join([v.name for v in loaded.trainable_variables[:5]])))
```

In [ ]:

```
trainable_variable_ids = {id(v) for v in loaded.trainable_variables}  
non_trainable_variables = [v for v in loaded.variables  
                           if id(v) not in trainable_variable_ids]  
print("MobileNet also has {} non-trainable variables: {}, ...".format(  
    len(non_trainable_variables),  
    ", ".join([v.name for v in non_trainable_variables[:3]])))
```

## Specifying signatures during export

Tools like TensorFlow Serving and `saved_model_cli` can interact with SavedModels. To help these tools determine which ConcreteFunctions to use, you need to specify serving signatures. `tf.keras.Model`'s automatically specify serving signatures, but you'll have to explicitly declare a serving signature for our custom modules.

IMPORTANT: Unless you need to export your model to an environment other than TensorFlow 2.x with Python, you probably don't need to export signatures explicitly. If you're looking for a way of enforcing an input signature for a specific function, see the `input_signature` ([https://www.tensorflow.org/api\\_docs/python/tf/function#args\\_1](https://www.tensorflow.org/api_docs/python/tf/function#args_1)) argument to `tf.function`.

By default, no signatures are declared in a custom `tf.Module`.

```
In [ ]:
```

```
assert len(imported.signatures) == 0
```

To declare a serving signature, specify a `ConcreteFunction` using the `signatures` kwarg. When specifying a single signature, its signature key will be '`serving_default`', which is saved as the constant `tf.saved_model.DEFAULT_SERVING_SIGNATURE_DEF_KEY`.

```
In [ ]:
```

```
module_with_signature_path = os.path.join(tmpdir, 'module_with_signature')
call = module.__call__.get_concrete_function(tf.TensorSpec(None, tf.float32))
tf.saved_model.save(module, module_with_signature_path, signatures=call)
```

```
In [ ]:
```

```
imported_with_signatures = tf.saved_model.load(module_with_signature_path)
list(imported_with_signatures.signatures.keys())
```

To export multiple signatures, pass a dictionary of signature keys to `ConcreteFunctions`. Each signature key corresponds to one `ConcreteFunction`.

```
In [ ]:
```

```
module_multiple_signatures_path = os.path.join(tmpdir, 'module_with_multiple_signatures')
signatures = {"serving_default": call,
              "array_input": module.__call__.get_concrete_function(tf.TensorSpec([None], tf.float32))}

tf.saved_model.save(module, module_multiple_signatures_path, signatures=signatures)
```

```
In [ ]:
```

```
imported_with_multiple_signatures = tf.saved_model.load(module_multiple_signatures_path)
list(imported_with_multiple_signatures.signatures.keys())
```

By default, the output tensor names are fairly generic, like `output_0`. To control the names of outputs, modify your `tf.function` to return a dictionary that maps output names to outputs. The names of inputs are derived from the Python function arg names.

```
In [ ]:
```

```
class CustomModuleWithOutputName(tf.Module):
    def __init__(self):
        super(CustomModuleWithOutputName, self).__init__()
        self.v = tf.Variable(1.)

    @tf.function(input_signature=[tf.TensorSpec([], tf.float32)])
    def __call__(self, x):
        return {'custom_output_name': x * self.v}

module_output = CustomModuleWithOutputName()
call_output = module_output.__call__.get_concrete_function(tf.TensorSpec(None, tf.float32))
module_output_path = os.path.join(tmpdir, 'module_with_output_name')
tf.saved_model.save(module_output, module_output_path,
                    signatures={'serving_default': call_output})
```

```
In [ ]:
```

```
imported_with_output_name = tf.saved_model.load(module_output_path)
imported_with_output_name.signatures['serving_default'].structured_outputs
```

## Load a SavedModel in C++

The C++ version of the `SavedModel` [loader](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/cc/saved_model/loader.h) ([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/cc/saved\\_model/loader.h](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/cc/saved_model/loader.h)) provides an API to load a `SavedModel` from a path, while allowing `SessionOptions` and `RunOptions`. You have to specify the tags associated with the graph to be loaded. The loaded version of `SavedModel` is referred to as `SavedModelBundle` and contains the `MetaGraphDef` and the session within which it is loaded.

```
const string export_dir = ...
SavedModelBundle bundle;
...
LoadSavedModel(session_options, run_options, export_dir, {kSavedModelTagTrain},
               &bundle);
```

## Details of the `SavedModel` command line interface

You can use the SavedModel Command Line Interface (CLI) to inspect and execute a SavedModel. For example, you can use the CLI to inspect the model's `SignatureDef`s. The CLI enables you to quickly confirm that the input Tensor `dtype` and shape match the model. Moreover, if you want to test your model, you can use the CLI to do a sanity check by passing in sample inputs in various formats (for example, Python expressions) and then fetching the output.

## Install the SavedModel CLI

Broadly speaking, you can install TensorFlow in either of the following two ways:

- By installing a pre-built TensorFlow binary.
- By building TensorFlow from source code.

If you installed TensorFlow through a pre-built TensorFlow binary, then the SavedModel CLI is already installed on your system at pathname `bin/saved_model_cli`.

If you built TensorFlow from source code, you must run the following additional command to build `saved_model_cli`:

```
$ bazel build tensorflow/python/tools:saved_model_cli
```

## Overview of commands

The SavedModel CLI supports the following two commands on a SavedModel:

- `show`, which shows the computations available from a SavedModel.
- `run`, which runs a computation from a SavedModel.

### show command

A SavedModel contains one or more model variants (technically, `v1.MetaGraphDef`s), identified by their tag-sets. To serve a model, you might wonder what kind of `SignatureDef`s are in each model variant, and what are their inputs and outputs. The `show` command let you examine the contents of the SavedModel in hierarchical order. Here's the syntax:

```
usage: saved_model_cli show [-h] --dir DIR [--all]
                            [--tag_set TAG_SET] [--signature_def SIGNATURE_DEF_KEY]
```

For example, the following command shows all available tag-sets in the SavedModel:

```
$ saved_model_cli show --dir /tmp/saved_model_dir
The given SavedModel contains the following tag-sets:
serve
serve, gpu
```

The following command shows all available `SignatureDef` keys for a tag set:

```
$ saved_model_cli show --dir /tmp/saved_model_dir --tag_set serve
The given SavedModel `MetaGraphDef` contains `SignatureDefs` with the
following keys:
SignatureDef key: "classify_x2_to_y3"
SignatureDef key: "classify_x_to_y"
SignatureDef key: "regress_x2_to_y3"
SignatureDef key: "regress_x_to_y"
SignatureDef key: "regress_x_to_y2"
SignatureDef key: "serving_default"
```

If there are *multiple* tags in the tag-set, you must specify all tags, each tag separated by a comma. For example:

```
$ saved_model_cli show --dir /tmp/saved_model_dir --tag_set serve,gpu
```

To show all inputs and outputs `TensorInfo` for a specific `SignatureDef`, pass in the `SignatureDef` key to `signature_def` option. This is very useful when you want to know the tensor key value, `dtype` and shape of the input tensors for executing the computation graph later. For example:

```
$ saved_model_cli show --dir \
/tmp/saved_model_dir --tag_set serve --signature_def serving_default
The given SavedModel SignatureDef contains the following input(s):
  inputs['x'] tensor_info:
    dtype: DT_FLOAT
    shape: (-1, 1)
    name: x:0
The given SavedModel SignatureDef contains the following output(s):
  outputs['y'] tensor_info:
    dtype: DT_FLOAT
    shape: (-1, 1)
    name: y:0
Method name is: tensorflow/serving/predict
```

To show all available information in the SavedModel, use the `--all` option. For example:

```
$ saved_model_cli show --dir /tmp/saved_model_dir --all
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

signature_def['classify_x2_to_y3']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['inputs'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 1)
      name: x2:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['scores'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 1)
      name: y3:0
  Method name is: tensorflow/serving/classify

...
signature_def['serving_default']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['x'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 1)
      name: x:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['y'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 1)
      name: y:0
  Method name is: tensorflow/serving/predict
```

## run command

Invoke the `run` command to run a graph computation, passing inputs and then displaying (and optionally saving) the outputs. Here's the syntax:

```
usage: saved_model_cli run [-h] --dir DIR --tag_set TAG_SET --signature_def
                           SIGNATURE_DEF_KEY [--inputs INPUTS]
                           [--input_exprs INPUT_EXPRS]
                           [--input_examples INPUT_EXAMPLES] [--outdir OUTDIR]
                           [--overwrite] [--tf_debug]
```

The `run` command provides the following three ways to pass inputs to the model:

- `--inputs` option enables you to pass numpy ndarray in files.
- `--input_exprs` option enables you to pass Python expressions.
- `--input_examples` option enables you to pass `tf.train.Example`.

### --inputs

To pass input data in files, specify the `--inputs` option, which takes the following general format:

```
bsh
--inputs <INPUTS>
```

where `INPUTS` is either of the following formats:

- `<input_key>=<filename>`
- `<input_key>=<filename>[<variable_name>]`

You may pass multiple `INPUTS`. If you do pass multiple inputs, use a semicolon to separate each of the `INPUTS`.

`saved_model_cli` uses `numpy.load` to load the `filename`. The `filename` may be in any of the following formats:

- `.npy`
- `.npz`
- pickle format

A `.npy` file always contains a numpy ndarray. Therefore, when loading from a `.npy` file, the content will be directly assigned to the specified input tensor. If you specify a `variable_name` with that `.npy` file, the `variable_name` will be ignored and a warning will be issued.

When loading from a `.npz` (zip) file, you may optionally specify a `variable_name` to identify the variable within the zip file to load for the input tensor key. If you don't specify a `variable_name`, the SavedModel CLI will check that only one file is included in the zip file and load it for the specified input tensor key.

When loading from a pickle file, if no `variable_name` is specified in the square brackets, whatever that is inside the pickle file will be passed to the specified input tensor key. Otherwise, the SavedModel CLI will assume a dictionary is stored in the pickle file and the value corresponding to the `variable_name` will be used.

### --input\_exprs

To pass inputs through Python expressions, specify the `--input_exprs` option. This can be useful for when you don't have data files lying around, but still want to sanity check the model with some simple inputs that match the dtype and shape of the model's `SignatureDef`s. For example:

```
bsh
`<input_key>=[[1],[2],[3]]`
```

In addition to Python expressions, you may also pass numpy functions. For example:

```
bsh
`<input_key>=np.ones((32,32,3))`
```

(Note that the `numpy` module is already available to you as `np`.)

### --input\_examples

To pass `tf.train.Example` as inputs, specify the `--input_examples` option. For each input key, it takes a list of dictionary, where each dictionary is an instance of `tf.train.Example`. The dictionary keys are the features and the values are the value lists for each feature. For example:

```
bsh
`<input_key>=[{"age":[22,24],"education":["BS","MS"]}]`
```

### Save output

By default, the SavedModel CLI writes output to stdout. If a directory is passed to `--outdir` option, the outputs will be saved as `.npy` files named after output tensor keys under the given directory.

Use `--overwrite` to overwrite existing output files.

**Copyright 2018 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Distributed training with TensorFlow



[View on TensorFlow.org](https://www.tensorflow.org/guide/distributed_training)

([https://www.tensorflow.org/guide/distributed\\_training](https://www.tensorflow.org/guide/distributed_training)) ([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/distributed\\_training.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/distributed_training.ipynb)) ([https://github.com/tensorflow/docs/blob/master/site/en/guide/distributed\\_training.ipynb](https://github.com/tensorflow/docs/blob/master/site/en/guide/distributed_training.ipynb))



[Run in Google Colab](#)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/distributed\\_training.ipynb](#)) ([https://github.com/tensorflow/docs/blob/master/site/en/guide/distributed\\_training.ipynb](https://github.com/tensorflow/docs/blob/master/site/en/guide/distributed_training.ipynb))

# Overview

`tf.distribute.Strategy` is a TensorFlow API to distribute training across multiple GPUs, multiple machines, or TPUs. Using this API, you can distribute your existing models and training code with minimal code changes.

`tf.distribute.Strategy` has been designed with these key goals in mind:

- Easy to use and support multiple user segments, including researchers, machine learning engineers, etc.
- Provide good performance out of the box.
- Easy switching between strategies.

You can distribute training using `tf.distribute.Strategy` with a high-level API like Keras `Model.fit`, as well as [custom training loops](#) ([https://www.tensorflow.org/guide/keras/writing\\_a\\_training\\_loop\\_from\\_scratch](https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch)) (and, in general, any computation using TensorFlow).

In TensorFlow 2.x, you can execute your programs eagerly, or in a graph using `tf.function` (`function.ipynb`). `tf.distribute.Strategy` intends to support both these modes of execution, but works best with `tf.function`. Eager mode is only recommended for debugging purposes and not supported for `tf.distribute.TPUStrategy`. Although training is the focus of this guide, this API can also be used for distributing evaluation and prediction on different platforms.

You can use `tf.distribute.Strategy` with very few changes to your code, because the underlying components of TensorFlow have been changed to become strategy-aware. This includes variables, layers, models, optimizers, metrics, summaries, and checkpoints.

In this guide, you will learn about various types of strategies and how you can use them in different situations. To learn how to debug performance issues, check out the [Optimize TensorFlow GPU performance \(gpu\\_performance\\_analysis.md\)](#) guide.

Note: For a deeper understanding of the concepts, watch the deep-dive presentation—[Inside TensorFlow: tf.distribute.Strategy](#) (<https://youtu.be/jKV53r9-H14>). This is especially recommended if you plan to write your own training loop.

## Set up TensorFlow

In [ ]:

```
import tensorflow as tf
```

## Types of strategies

`tf.distribute.Strategy` intends to cover a number of use cases along different axes. Some of these combinations are currently supported and others will be added in the future. Some of these axes are:

- *Synchronous vs asynchronous training*: These are two common ways of distributing training with data parallelism. In sync training, all workers train over different slices of input data in sync, and aggregating gradients at each step. In async training, all workers are independently training over the input data and updating variables asynchronously. Typically sync training is supported via all-reduce and async through parameter server architecture.
- *Hardware platform*: You may want to scale your training onto multiple GPUs on one machine, or multiple machines in a network (with 0 or more GPUs each), or on Cloud TPUs.

In order to support these use cases, TensorFlow has `MirroredStrategy`, `TPUStrategy`, `MultiWorkerMirroredStrategy`, `ParameterServerStrategy`, `CentralStorageStrategy`, as well as other strategies available. The next section explains which of these are supported in which scenarios in TensorFlow. Here is a quick overview:

Training API	<code>MirroredStrategy</code>	<code>TPUStrategy</code>	<code>MultiWorkerMirroredStrategy</code>	<code>CentralStorageStrategy</code>	<code>ParameterServerStrategy</code>
<code>Keras Model.fit</code>	Supported	Supported	Supported	Experimental support	Experimental support
<code>Custom training loop</code>	Supported	Supported	Supported	Experimental support	Experimental support
<code>Estimator API</code>	Limited Support	Not supported	Limited Support	Limited Support	Limited Support

Note: [Experimental support](#) ([https://www.tensorflow.org/guide/versions#what\\_is\\_not\\_covered](https://www.tensorflow.org/guide/versions#what_is_not_covered)) means the APIs are not covered by any compatibility guarantees.

Warning: Estimator support is limited. Basic training and evaluation are experimental, and advanced features—such as scaffold—are not implemented. You should be using Keras or custom training loops if a use case is not covered. Estimators are not recommended for new code. Estimators run `v1.Session`-style code which is more difficult to write correctly, and can behave unexpectedly, especially when combined with TF 2 code. Estimators do fall under our [compatibility guarantees](#) (<https://tensorflow.org/guide/versions>), but will receive no fixes other than security vulnerabilities. Go to the [migration guide](#) (<https://tensorflow.org/guide/migrate>) for details.

## MirroredStrategy

`tf.distribute.MirroredStrategy` supports synchronous distributed training on multiple GPUs on one machine. It creates one replica per GPU device. Each variable in the model is mirrored across all the replicas. Together, these variables form a single conceptual variable called `MirroredVariable`. These variables are kept in sync with each other by applying identical updates.

Efficient all-reduce algorithms are used to communicate the variable updates across the devices. All-reduce aggregates tensors across all the devices by adding them up, and makes them available on each device. It's a fused algorithm that is very efficient and can reduce the overhead of synchronization significantly. There are many all-reduce algorithms and implementations available, depending on the type of communication available between devices. By default, it uses the NVIDIA Collective Communication Library ([NCCL](https://developer.nvidia.com/nccl) (<https://developer.nvidia.com/nccl>)) as the all-reduce implementation. You can choose from a few other options or write your own.

Here is the simplest way of creating `MirroredStrategy` :

In [ ]:

```
mirrored_strategy = tf.distribute.MirroredStrategy()
```

This will create a `MirroredStrategy` instance, which will use all the GPUs that are visible to TensorFlow, and NCCL—as the cross-device communication.

If you wish to use only some of the GPUs on your machine, you can do so like this:

In [ ]:

```
mirrored_strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
```

If you wish to override the cross device communication, you can do so using the `cross_device_ops` argument by supplying an instance of `tf.distribute.CrossDeviceOps`. Currently, `tf.distribute.HierarchicalCopyAllReduce` and `tf.distribute.ReductionToOneDevice` are two options other than `tf.distribute.NcclAllReduce`, which is the default.

In [ ]:

```
mirrored_strategy = tf.distribute.MirroredStrategy(  
    cross_device_ops=tf.distribute.HierarchicalCopyAllReduce())
```

## TPUStrategy

`tf.distribute.TPUStrategy` lets you run your TensorFlow training on [Tensor Processing Units \(TPUs\) \(tpu.ipynb\)](#). TPUs are Google's specialized ASICs designed to dramatically accelerate machine learning workloads. They are available on [Google Colab](#) (<https://colab.research.google.com/>), the [TPU Research Cloud](#) (<https://sites.research.google/trc/>), and [Cloud TPU](#) (<https://cloud.google.com/tpu>).

In terms of distributed training architecture, `TPUStrategy` is the same `MirroredStrategy` —it implements synchronous distributed training. TPUs provide their own implementation of efficient all-reduce and other collective operations across multiple TPU cores, which are used in `TPUStrategy`.

Here is how you would instantiate `TPUStrategy` :

Note: To run any TPU code in Colab, you should select TPU as the Colab runtime. Refer to the [Use TPUs \(tpu.ipynb\)](#) guide for a complete example.

```
cluster_resolver = tf.distribute.cluster_resolver.TPUClusterResolver(  
    tpu=tpu_address)  
tf.config.experimental_connect_to_cluster(cluster_resolver)  
tf.tpu.experimental.initialize_tpu_system(cluster_resolver)  
tpu_strategy = tf.distribute.TPUStrategy(cluster_resolver)
```

The `TPUClusterResolver` instance helps locate the TPUs. In Colab, you don't need to specify any arguments to it.

If you want to use this for Cloud TPUs:

- You must specify the name of your TPU resource in the `tpu` argument.
- You must initialize the TPU system explicitly at the *start* of the program. This is required before TPUs can be used for computation. Initializing the TPU system also wipes out the TPU memory, so it's important to complete this step first in order to avoid losing state.

## MultiWorkerMirroredStrategy

`tf.distribute.MultiWorkerMirroredStrategy` is very similar to `MirroredStrategy`. It implements synchronous distributed training across multiple workers, each with potentially multiple GPUs. Similar to `tf.distribute.MirroredStrategy`, it creates copies of all variables in the model on each device across all workers.

Here is the simplest way of creating `MultiWorkerMirroredStrategy` :

In [ ]:

```
strategy = tf.distribute.MultiWorkerMirroredStrategy()
```

`MultiWorkerMirroredStrategy` has two implementations for cross-device communications. `CommunicationImplementation.RING` is [RPC](https://en.wikipedia.org/wiki/Remote_procedure_call) ([https://en.wikipedia.org/wiki/Remote\\_procedure\\_call](https://en.wikipedia.org/wiki/Remote_procedure_call))-based and supports both CPUs and GPUs. `CommunicationImplementation.NCCL` uses NCCL and provides state-of-art performance on GPUs but it doesn't support CPUs. `CollectiveCommunication.AUTO` defers the choice to Tensorflow. You can specify them in the following way:

In [ ]:

```
communication_options = tf.distribute.experimental.CommunicationOptions(
    implementation=tf.distribute.experimental.CommunicationImplementation.NCCL)
strategy = tf.distribute.MultiWorkerMirroredStrategy(
    communication_options=communication_options)
```

One of the key differences to get multi worker training going, as compared to multi-GPU training, is the multi-worker setup. The '`TF_CONFIG`' environment variable is the standard way in TensorFlow to specify the cluster configuration to each worker that is part of the cluster. Learn more in the [setting up TF\\_CONFIG section](#) of this document.

For more details about `MultiWorkerMirroredStrategy`, consider the following tutorials:

- [Multi-worker training with Keras Model.fit \(./tutorials/distribute/multi\\_worker\\_with\\_keras.ipynb\)](#)
- [Multi-worker training with a custom training loop \(./tutorials/distribute/multi\\_worker\\_with\\_ctl.ipynb\)](#)

## ParameterServerStrategy

Parameter server training is a common data-parallel method to scale up model training on multiple machines. A parameter server training cluster consists of workers and parameter servers. Variables are created on parameter servers and they are read and updated by workers in each step. Check out the [Parameter server training \(./tutorials/distribute/parameter\\_server\\_training.ipynb\)](#) tutorial for details.

In TensorFlow 2, parameter server training uses a central coordinator-based architecture via the `tf.distribute.experimental.coordinator.ClusterCoordinator` class.

In this implementation, the worker and parameter server tasks run `tf.distribute.Server`s that listen for tasks from the coordinator. The coordinator creates resources, dispatches training tasks, writes checkpoints, and deals with task failures.

In the programming running on the coordinator, you will use a `ParameterServerStrategy` object to define a training step and use a `ClusterCoordinator` to dispatch training steps to remote workers. Here is the simplest way to create them:

```
strategy = tf.distribute.experimental.ParameterServerStrategy(
    tf.distribute.cluster_resolver.TFConfigClusterResolver(),
    variable_partitioner=variable_partitioner)
coordinator = tf.distribute.experimental.coordinator.ClusterCoordinator(
    strategy)
```

To learn more about `ParameterServerStrategy`, check out the [Parameter server training with Keras Model.fit and a custom training loop \(./tutorials/distribute/parameter\\_server\\_training.ipynb\)](#) tutorial.

Note: You will need to configure the '`TF_CONFIG`' environment variable if you use `TFConfigClusterResolver`. It is similar to '`TF_CONFIG`' in `MultiWorkerMirroredStrategy` but has additional caveats.

In TensorFlow 1, `ParameterServerStrategy` is available only with an Estimator via `tf.compat.v1.distribute.experimental.ParameterServerStrategy` symbol.

Note: This strategy is `experimental` ([https://www.tensorflow.org/guide/versions#what\\_is\\_not\\_covered](https://www.tensorflow.org/guide/versions#what_is_not_covered)) as it is currently under active development.

## CentralStorageStrategy

`tf.distribute.experimental.CentralStorageStrategy` does synchronous training as well. Variables are not mirrored, instead they are placed on the CPU and operations are replicated across all local GPUs. If there is only one GPU, all variables and operations will be placed on that GPU.

Create an instance of `CentralStorageStrategy` by:

In [ ]:

```
central_storage_strategy = tf.distribute.experimental.CentralStorageStrategy()
```

This will create a `CentralStorageStrategy` instance which will use all visible GPUs and CPU. Update to variables on replicas will be aggregated before being applied to variables.

## Other strategies

In addition to the above strategies, there are two other strategies which might be useful for prototyping and debugging when using `tf.distribute` APIs.

### Default Strategy

The Default Strategy is a distribution strategy which is present when no explicit distribution strategy is in scope. It implements the `tf.distribute.Strategy` interface but is a pass-through and provides no actual distribution. For instance, `Strategy.run(fn)` will simply call `fn`. Code written using this strategy should behave exactly as code written without any strategy. You can think of it as a "no-op" strategy.

The Default Strategy is a singleton—and one cannot create more instances of it. It can be obtained using `tf.distribute.get_strategy` outside any explicit strategy's scope (the same API that can be used to get the current strategy inside an explicit strategy's scope).

In [ ]:

```
default_strategy = tf.distribute.get_strategy()
```

This strategy serves two main purposes:

- It allows writing distribution-aware library code unconditionally. For example, in `tf.optimizers` you can use `tf.distribute.get_strategy` and use that strategy for reducing gradients—it will always return a strategy object on which you can call the `Strategy.reduce` API.

In [ ]:

```
# In optimizer or other library code
# Get currently active strategy
strategy = tf.distribute.get_strategy()
strategy.reduce("SUM", 1., axis=None) # reduce some values
```

- Similar to library code, it can be used to write end users' programs to work with and without distribution strategy, without requiring conditional logic. Here's a sample code snippet illustrating this:

In [ ]:

```
if tf.config.list_physical_devices('GPU'):
    strategy = tf.distribute.MirroredStrategy()
else: # Use the Default Strategy
    strategy = tf.distribute.get_strategy()

with strategy.scope():
    # Do something interesting
    print(tf.Variable(1.))
```

### OneDeviceStrategy

`tf.distribute.OneDeviceStrategy` is a strategy to place all variables and computation on a single specified device.

```
strategy = tf.distribute.OneDeviceStrategy(device="/gpu:0")
```

This strategy is distinct from the Default Strategy in a number of ways. In the Default Strategy, the variable placement logic remains unchanged when compared to running TensorFlow without any distribution strategy. But when using `OneDeviceStrategy`, all variables created in its scope are explicitly placed on the specified device. Moreover, any functions called via `OneDeviceStrategy.run` will also be placed on the specified device.

Input distributed through this strategy will be prefetched to the specified device. In the Default Strategy, there is no input distribution.

Similar to the Default Strategy, this strategy could also be used to test your code before switching to other strategies which actually distribute to multiple devices/machines. This will exercise the distribution strategy machinery somewhat more than the Default Strategy, but not to the full extent of using, for example, `MirroredStrategy` or `TPUStrategy`. If you want code that behaves as if there is no strategy, then use the Default Strategy.

So far you've learned about different strategies and how you can instantiate them. The next few sections show the different ways in which you can use them to distribute your training.

## Use `tf.distribute.Strategy` with Keras Model.fit

`tf.distribute.Strategy` is integrated into `tf.keras`, which is TensorFlow's implementation of the [Keras API specification \(https://keras.io\)](https://keras.io). `tf.keras` is a high-level API to build and train models. By integrating into the `tf.keras` backend, it's seamless for you to distribute your training written in the Keras training framework [using Model.fit \(/keras/customizing\\_what\\_happens\\_in\\_fit.ipynb\)](#).

Here's what you need to change in your code:

1. Create an instance of the appropriate `tf.distribute.Strategy`.
2. Move the creation of Keras model, optimizer and metrics inside `strategy.scope`.

TensorFlow distribution strategies support all types of Keras models—[Sequential \(/keras/sequential\\_model.ipynb\)](#), [Functional \(/keras/functional.ipynb\)](#), and [subclassed \(/keras/custom\\_layers\\_and\\_models.ipynb\)](#).

Here is a snippet of code to do this for a very simple Keras model with one `Dense` layer:

In [ ]:

```
mirrored_strategy = tf.distribute.MirroredStrategy()

with mirrored_strategy.scope():
    model = tf.keras.Sequential([tf.keras.layers.Dense(1, input_shape=(1,))])

model.compile(loss='mse', optimizer='sgd')
```

This example uses `MirroredStrategy`, so you can run this on a machine with multiple GPUs. `strategy.scope()` indicates to Keras which strategy to use to distribute the training. Creating models/optimizers/metrics inside this scope allows you to create distributed variables instead of regular variables. Once this is set up, you can fit your model like you would normally. `MirroredStrategy` takes care of replicating the model's training on the available GPUs, aggregating gradients, and more.

In [ ]:

```
dataset = tf.data.Dataset.from_tensors(([1.], [1.])).repeat(100).batch(10)
model.fit(dataset, epochs=2)
model.evaluate(dataset)
```

Here a `tf.data.Dataset` provides the training and eval input. You can also use NumPy arrays:

In [ ]:

```
import numpy as np

inputs, targets = np.ones((100, 1)), np.ones((100, 1))
model.fit(inputs, targets, epochs=2, batch_size=10)
```

In both cases—with `Dataset` or NumPy—each batch of the given input is divided equally among the multiple replicas. For instance, if you are using the `MirroredStrategy` with 2 GPUs, each batch of size 10 will be divided among the 2 GPUs, with each receiving 5 input examples in each step. Each epoch will then train faster as you add more GPUs. Typically, you would want to increase your batch size as you add more accelerators, so as to make effective use of the extra computing power. You will also need to re-tune your learning rate, depending on the model. You can use `strategy.num_replicas_in_sync` to get the number of replicas.

In [ ]:

```
# Compute a global batch size using a number of replicas.
BATCH_SIZE_PER_REPLICA = 5
global_batch_size = (BATCH_SIZE_PER_REPLICA *
                     mirrored_strategy.num_replicas_in_sync)
dataset = tf.data.Dataset.from_tensors(([1.], [1.])).repeat(100)
dataset = dataset.batch(global_batch_size)

LEARNING_RATES_BY_BATCH_SIZE = {5: 0.1, 10: 0.15}
learning_rate = LEARNING_RATES_BY_BATCH_SIZE[global_batch_size]
```

## What's supported now?

Training API	MirroredStrategy	TPUStrategy	MultiWorkerMirroredStrategy	ParameterServerStrategy	CentralStorageStrategy
Keras Model.fit	Supported	Supported	Supported	Experimental support	Experimental support

## Examples and tutorials

Here is a list of tutorials and examples that illustrate the above integration end-to-end with Keras Model.fit :

1. [Tutorial \(../tutorials/distribute/keras.ipynb\)](#): Training with Model.fit and MirroredStrategy .
2. [Tutorial \(../tutorials/distribute/multi\\_worker\\_with\\_keras.ipynb\)](#): Training with Model.fit and MultiWorkerMirroredStrategy .
3. [Guide \(tpu.ipynb\)](#): Contains an example of using Model.fit and TPUStrategy .
4. [Tutorial \(../tutorials/distribute/parameter\\_server\\_training.ipynb\)](#): Parameter server training with Model.fit and ParameterServerStrategy .
5. [Tutorial \(https://www.tensorflow.org/text/tutorials/bert\\_glue\)](#): Fine-tuning BERT for many tasks from the GLUE benchmark with Model.fit and TPUStrategy .
6. TensorFlow Model Garden [repository \(https://github.com/tensorflow/models/tree/master/official\)](https://github.com/tensorflow/models/tree/master/official) containing collections of state-of-the-art models implemented using various strategies.

## Use tf.distribute.Strategy with custom training loops

As demonstrated above, using `tf.distribute.Strategy` with Keras Model.fit requires changing only a couple lines of your code. With a little more effort, you can also use `tf.distribute.Strategy` [with custom training loops \(/keras/writing\\_a\\_training\\_loop\\_from\\_scratch.ipynb\)](#).

If you need more flexibility and control over your training loops than is possible with Estimator or Keras, you can write custom training loops. For instance, when using a GAN, you may want to take a different number of generator or discriminator steps each round. Similarly, the high level frameworks are not very suitable for Reinforcement Learning training.

The `tf.distribute.Strategy` classes provide a core set of methods to support custom training loops. Using these may require minor restructuring of the code initially, but once that is done, you should be able to switch between GPUs, TPUs, and multiple machines simply by changing the strategy instance.

Below is a brief snippet illustrating this use case for a simple training example using the same Keras model as before.

First, create the model and optimizer inside the strategy's scope. This ensures that any variables created with the model and optimizer are mirrored variables.

In [ ]:

```
with mirrored_strategy.scope():
    model = tf.keras.Sequential([tf.keras.layers.Dense(1, input_shape=(1,))])
    optimizer = tf.keras.optimizers.SGD()
```

Next, create the input dataset and call `tf.distribute.Strategy.experimental_distribute_dataset` to distribute the dataset based on the strategy.

In [ ]:

```
dataset = tf.data.Dataset.from_tensors(([1.], [1.])).repeat(100).batch(
    global_batch_size)
dist_dataset = mirrored_strategy.experimental_distribute_dataset(dataset)
```

Then, define one step of the training. Use `tf.GradientTape` to compute gradients and optimizer to apply those gradients to update your model's variables. To distribute this training step, put it in a function `train_step` and pass it to `tf.distribute.Strategy.run` along with the dataset inputs you got from the `dist_dataset` created before:

In [ ]:

```
loss_object = tf.keras.losses.BinaryCrossentropy(
    from_logits=True,
    reduction=tf.keras.losses.Reduction.NONE)

def compute_loss(labels, predictions):
    per_example_loss = loss_object(labels, predictions)
    return tf.nn.compute_average_loss(per_example_loss, global_batch_size=global_batch_size)

def train_step(inputs):
    features, labels = inputs

    with tf.GradientTape() as tape:
        predictions = model(features, training=True)
        loss = compute_loss(labels, predictions)

    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss

@tf.function
def distributed_train_step(dist_inputs):
    per_replica_losses = mirrored_strategy.run(train_step, args=(dist_inputs,))
    return mirrored_strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses,
                                    axis=None)
```

A few other things to note in the code above:

1. You used `tf.nn.compute_average_loss` to compute the loss. `tf.nn.compute_average_loss` sums the per example loss and divides the sum by the `global_batch_size`. This is important because later after the gradients are calculated on each replica, they are aggregated across the replicas by **summing** them.
2. You also used the `tf.distribute.Strategy.reduce` API to aggregate the results returned by `tf.distribute.Strategy.run`.  
`tf.distribute.Strategy.run` returns results from each local replica in the strategy, and there are multiple ways to consume this result. You can `reduce` them to get an aggregated value. You can also do `tf.distribute.Strategy.experimental_local_results` to get the list of values contained in the result, one per local replica.
3. When you call `apply_gradients` within a distribution strategy scope, its behavior is modified. Specifically, before applying gradients on each parallel instance during synchronous training, it performs a sum-over-all-replicas of the gradients.

Finally, once you have defined the training step, you can iterate over `dist_dataset` and run the training in a loop:

In [ ]:

```
for dist_inputs in dist_dataset:
    print(distributed_train_step(dist_inputs))
```

In the example above, you iterated over the `dist_dataset` to provide input to your training. You are also provided with the `tf.distribute.Strategy.make_experimental_numpy_dataset` to support NumPy inputs. You can use this API to create a dataset before calling `tf.distribute.Strategy.experimental_distribute_dataset`.

Another way of iterating over your data is to explicitly use iterators. You may want to do this when you want to run for a given number of steps as opposed to iterating over the entire dataset. The above iteration would now be modified to first create an iterator and then explicitly call `next` on it to get the input data.

In [ ]:

```
iterator = iter(dist_dataset)
for _ in range(10):
    print(distributed_train_step(next(iterator)))
```

This covers the simplest case of using `tf.distribute.Strategy` API to distribute custom training loops.

## What's supported now?

Training API	MirroredStrategy	TPUStrategy	MultiWorkerMirroredStrategy	ParameterServerStrategy	CentralStorageStrategy
Custom training loop	Supported	Supported	Supported	Experimental support	Experimental support

## Examples and tutorials

Here are some examples for using distribution strategies with custom training loops:

1. [Tutorial \(./tutorials/distribute/custom\\_training.ipynb\)](#): Training with a custom training loop and `MirroredStrategy`.
2. [Tutorial \(./tutorials/distribute/multi\\_worker\\_with\\_ctl.ipynb\)](#): Training with a custom training loop and `MultiWorkerMirroredStrategy`.
3. [Guide \(tpu.ipynb\)](#): Contains an example of a custom training loop with `TPUStrategy`.
4. [Tutorial \(./tutorials/distribute/parameter\\_server\\_training.ipynb\)](#): Parameter server training with a custom training loop and `ParameterServerStrategy`.
5. TensorFlow Model Garden [repository](#) (<https://github.com/tensorflow/models/tree/master/official>) containing collections of state-of-the-art models implemented using various strategies.

## Other topics

This section covers some topics that are relevant to multiple use cases.

### Setting up the TF\_CONFIG environment variable

For multi-worker training, as mentioned before, you need to set up the '`TF_CONFIG`' environment variable for each binary running in your cluster. The '`TF_CONFIG`' environment variable is a JSON string which specifies what tasks constitute a cluster, their addresses and each task's role in the cluster. The [tensorflow/ecosystem](#) (<https://github.com/tensorflow/ecosystem>) repo provides a Kubernetes template, which sets up '`TF_CONFIG`' for your training tasks.

There are two components of '`TF_CONFIG`' : a cluster and a task.

- A cluster provides information about the training cluster, which is a dict consisting of different types of jobs such as workers. In multi-worker training, there is usually one worker that takes on a little more responsibility like saving checkpoint and writing summary file for TensorBoard in addition to what a regular worker does. Such worker is referred to as the "chief" worker, and it is customary that the worker with index `0` is appointed as the chief worker (in fact this is how `tf.distribute.Strategy` is implemented).
- A task on the other hand provides information about the current task. The first component cluster is the same for all workers, and the second component task is different on each worker and specifies the type and index of that worker.

One example of '`TF_CONFIG`' is:

```
os.environ["TF_CONFIG"] = json.dumps({
    "cluster": {
        "worker": ["host1:port", "host2:port", "host3:port"],
        "ps": ["host4:port", "host5:port"]
    },
    "task": {"type": "worker", "index": 1}
})
```

This '`TF_CONFIG`' specifies that there are three workers and two "ps" tasks in the "cluster" along with their hosts and ports. The "task" part specifies the role of the current task in the "cluster" —worker 1 (the second worker). Valid roles in a cluster are "chief", "worker", "ps", and "evaluator". There should be no "ps" job except when using `tf.distribute.experimental.ParameterServerStrategy`.

## What's next?

`tf.distribute.Strategy` is actively under development. Try it out and provide your feedback using [GitHub issues](#) (<https://github.com/tensorflow/tensorflow/issues/new>).

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Use TPUs



[View on TensorFlow.org](https://www.tensorflow.org/guide/tpu)  
(<https://www.tensorflow.org/guide/tpu>)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/tpu.ipynb)  
(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/tpu.ipynb>)



Before you run this Colab notebook, make sure that your hardware accelerator is a TPU by checking your notebook settings: **Runtime > Change runtime type > Hardware accelerator > TPU**.

## Setup

In [ ]:

```
import tensorflow as tf

import os
import tensorflow_datasets as tfds
```

## TPU initialization

TPUs are typically Cloud TPU workers, which are different from the local process running the user's Python program. Thus, you need to do some initialization work to connect to the remote cluster and initialize the TPUs. Note that the `tpu` argument to `tf.distribute.cluster_resolver.TPUClusterResolver` is a special address just for Colab. If you are running your code on Google Compute Engine (GCE), you should instead pass in the name of your Cloud TPU.

Note: The TPU initialization code has to be at the beginning of your program.

In [ ]:

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='')
tf.config.experimental_connect_to_cluster(resolver)
# This is the TPU initialization code that has to be at the beginning.
tf.tpu.experimental.initialize_tpu_system(resolver)
print("All devices: ", tf.config.list_logical_devices('TPU'))
```

## Manual device placement

After the TPU is initialized, you can use manual device placement to place the computation on a single TPU device:

In [ ]:

```
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])

with tf.device('/TPU:0'):
    c = tf.matmul(a, b)

print("c device: ", c.device)
print(c)
```

## Distribution strategies

Usually you run your model on multiple TPUs in a data-parallel way. To distribute your model on multiple TPUs (or other accelerators), TensorFlow offers several distribution strategies. You can replace your distribution strategy and the model will run on any given (TPU) device. Check the [distribution strategy guide](#) ([./distributed\\_training.ipynb](#)) for more information.

To demonstrate this, create a `tf.distribute.TPUStrategy` object:

In [ ]:

```
strategy = tf.distribute.TPUStrategy(resolver)
```

To replicate a computation so it can run in all TPU cores, you can pass it into the `strategy.run` API. Below is an example that shows all cores receiving the same inputs (`a`, `b`) and performing matrix multiplication on each core independently. The outputs will be the values from all the replicas.

In [ ]:

```
@tf.function
def matmul_fn(x, y):
    z = tf.matmul(x, y)
    return z

z = strategy.run(matmul_fn, args=(a, b))
print(z)
```

## Classification on TPUs

Having covered the basic concepts, consider a more concrete example. This section demonstrates how to use the distribution strategy — `tf.distribute.TPUStrategy` — to train a Keras model on a Cloud TPU.

### Define a Keras model

Start with a definition of a `Sequential` Keras model for image classification on the MNIST dataset using Keras. It's no different than what you would use if you were training on CPUs or GPUs. Note that Keras model creation needs to be inside `strategy.scope`, so the variables can be created on each TPU device. Other parts of the code are not necessary to be inside the strategy scope.

In [ ]:

```
def create_model():
    return tf.keras.Sequential(
        [tf.keras.layers.Conv2D(256, 3, activation='relu', input_shape=(28, 28, 1)),
         tf.keras.layers.Conv2D(256, 3, activation='relu'),
         tf.keras.layers.Flatten(),
         tf.keras.layers.Dense(256, activation='relu'),
         tf.keras.layers.Dense(128, activation='relu'),
         tf.keras.layers.Dense(10)])
```

### Load the dataset

Efficient use of the `tf.data.Dataset` API is critical when using a Cloud TPU, as it is impossible to use the Cloud TPUs unless you can feed them data quickly enough. You can learn more about dataset performance in the [Input pipeline performance guide \(./data\\_performance.ipynb\)](#).

For all but the simplest experiments (using `tf.data.Dataset.from_tensor_slices` or other in-graph data), you need to store all data files read by the Dataset in Google Cloud Storage (GCS) buckets.

For most use cases, it is recommended to convert your data into the `TFRecord` format and use a `tf.data.TFRecordDataset` to read it. Check the [TFRecord and tf.Example tutorial \(./tutorials/load\\_data/tfrecord.ipynb\)](#) for details on how to do this. It is not a hard requirement and you can use other dataset readers, such as `tf.data.FixedLengthRecordDataset` or `tf.data.TextLineDataset`.

You can load entire small datasets into memory using `tf.data.Dataset.cache`.

Regardless of the data format used, it is strongly recommended that you use large files on the order of 100MB. This is especially important in this networked setting, as the overhead of opening a file is significantly higher.

As shown in the code below, you should use the `tensorflow_datasets` module to get a copy of the MNIST training and test data. Note that `try_gcs` is specified to use a copy that is available in a public GCS bucket. If you don't specify this, the TPU will not be able to access the downloaded data.

In [ ]:

```
def get_dataset(batch_size, is_training=True):
    split = 'train' if is_training else 'test'
    dataset, info = tfds.load(name='mnist', split=split, with_info=True,
                             as_supervised=True, try_gcs=True)

    # Normalize the input data.
    def scale(image, label):
        image = tf.cast(image, tf.float32)
        image /= 255.0
        return image, label

    dataset = dataset.map(scale)

    # Only shuffle and repeat the dataset in training. The advantage of having an
    # infinite dataset for training is to avoid the potential last partial batch
    # in each epoch, so that you don't need to think about scaling the gradients
    # based on the actual batch size.
    if is_training:
        dataset = dataset.shuffle(10000)
        dataset = dataset.repeat()

    dataset = dataset.batch(batch_size)

    return dataset
```

## Train the model using Keras high-level APIs

You can train your model with Keras `fit` and `compile` APIs. There is nothing TPU-specific in this step—you write the code as if you were using multiple GPUs and a `MirroredStrategy` instead of the `TPUStrategy`. You can learn more in the [Distributed training with Keras](#) (<https://www.tensorflow.org/tutorials/distribute/keras>) tutorial.

In [ ]:

```
with strategy.scope():
    model = create_model()
    model.compile(optimizer='adam',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['sparse_categorical_accuracy'])

batch_size = 200
steps_per_epoch = 60000 // batch_size
validation_steps = 10000 // batch_size

train_dataset = get_dataset(batch_size, is_training=True)
test_dataset = get_dataset(batch_size, is_training=False)

model.fit(train_dataset,
          epochs=5,
          steps_per_epoch=steps_per_epoch,
          validation_data=test_dataset,
          validation_steps=validation_steps)
```

To reduce Python overhead and maximize the performance of your TPU, pass in the argument—`steps_per_execution`—to `Model.compile`. In this example, it increases throughput by about 50%:

In [ ]:

```
with strategy.scope():
    model = create_model()
    model.compile(optimizer='adam',
                  # Anything between 2 and `steps_per_epoch` could help here.
                  steps_per_execution = 50,
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['sparse_categorical_accuracy'])

model.fit(train_dataset,
          epochs=5,
          steps_per_epoch=steps_per_epoch,
          validation_data=test_dataset,
          validation_steps=validation_steps)
```

## Train the model using a custom training loop

You can also create and train your model using `tf.function` and `tf.distribute` APIs directly. You can use the `strategy.experimental_distribute_datasets_from_function` API to distribute the dataset given a dataset function. Note that in the example below the batch size passed into the dataset is the per-replica batch size instead of the global batch size. To learn more, check out the [Custom training with `tf.distribute.Strategy`](https://www.tensorflow.org/tutorials/distribute/custom_training) ([https://www.tensorflow.org/tutorials/distribute/custom\\_training](https://www.tensorflow.org/tutorials/distribute/custom_training)) tutorial.

First, create the model, datasets and `tf.functions`:

In [ ]:

```
# Create the model, optimizer and metrics inside the strategy scope, so that the
# variables can be mirrored on each device.
with strategy.scope():
    model = create_model()
    optimizer = tf.keras.optimizers.Adam()
    training_loss = tf.keras.metrics.Mean('training_loss', dtype=tf.float32)
    training_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
        'training_accuracy', dtype=tf.float32)

# Calculate per replica batch size, and distribute the datasets on each TPU
# worker.
per_replica_batch_size = batch_size // strategy.num_replicas_in_sync

train_dataset = strategy.experimental_distribute_datasets_from_function(
    lambda _: get_dataset(per_replica_batch_size, is_training=True))

@tf.function
def train_step(iterator):
    """The step function for one training step."""

    def step_fn(inputs):
        """The computation to run on each TPU device."""
        images, labels = inputs
        with tf.GradientTape() as tape:
            logits = model(images, training=True)
            loss = tf.keras.losses.sparse_categorical_crossentropy(
                labels, logits, from_logits=True)
            loss = tf.nn.compute_average_loss(loss, global_batch_size=batch_size)
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(list(zip(grads, model.trainable_variables)))
        training_loss.update_state(loss * strategy.num_replicas_in_sync)
        training_accuracy.update_state(labels, logits)

    strategy.run(step_fn, args=(next(iterator),))
```

Then, run the training loop:

In [ ]:

```
steps_per_eval = 10000 // batch_size

train_iterator = iter(train_dataset)
for epoch in range(5):
    print('Epoch: {}/5'.format(epoch))

    for step in range(steps_per_epoch):
        train_step(train_iterator)
        print('Current step: {}, training loss: {}, accuracy: {}'.format(
            optimizer.iterations.numpy(),
            round(float(training_loss.result()), 4),
            round(float(training_accuracy.result()) * 100, 2)))
    training_loss.reset_states()
    training_accuracy.reset_states()
```

## Improving performance with multiple steps inside `tf.function`

You can improve the performance by running multiple steps within a `tf.function`. This is achieved by wrapping the `strategy.run` call with a `tf.range` inside `tf.function`, and AutoGraph will convert it to a `tf.while_loop` on the TPU worker.

Despite the improved performance, there are tradeoffs with this method compared to running a single step inside `tf.function`. Running multiple steps in a `tf.function` is less flexible—you cannot run things eagerly or arbitrary Python code within the steps.

In [ ]:

```
@tf.function
def train_multiple_steps(iterator, steps):
    """The step function for one training step."""

    def step_fn(inputs):
        """The computation to run on each TPU device."""
        images, labels = inputs
        with tf.GradientTape() as tape:
            logits = model(images, training=True)
            loss = tf.keras.losses.sparse_categorical_crossentropy(
                labels, logits, from_logits=True)
            loss = tf.nn.compute_average_loss(loss, global_batch_size=batch_size)
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(list(zip(grads, model.trainable_variables)))
        training_loss.update_state(loss * strategy.num_replicas_in_sync)
        training_accuracy.update_state(labels, logits)

    for _ in tf.range(steps):
        strategy.run(step_fn, args=(next(iterator),))

# Convert `steps_per_epoch` to `tf.Tensor` so the `tf.function` won't get
# retracted if the value changes.
train_multiple_steps(train_iterator, tf.convert_to_tensor(steps_per_epoch))

print('Current step: {}, training loss: {}, accuracy: {}%'.format(
    optimizer.iterations.numpy(),
    round(float(training_loss.result()), 4),
    round(float(training_accuracy.result()) * 100, 2)))
```

## Next steps

- [Google Cloud TPU documentation](https://cloud.google.com/tpu/docs/) (<https://cloud.google.com/tpu/docs/>): How to set up and run a Google Cloud TPU.
- [Google Cloud TPU Colab notebooks](https://cloud.google.com/tpu/docs/colabs) (<https://cloud.google.com/tpu/docs/colabs>): End-to-end training examples.
- [Google Cloud TPU performance guide](https://cloud.google.com/tpu/docs/performance-guide) (<https://cloud.google.com/tpu/docs/performance-guide>): Enhance Cloud TPU performance further by adjusting Cloud TPU configuration parameters for your application
- [Distributed training with TensorFlow](#) ([./distributed\\_training.ipynb](#)): How to use distribution strategies—including `tf.distribute.TPUStrategy`—with examples showing best practices.

Copyright 2019 The TensorFlow Authors.

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Estimators



[View on TensorFlow.org](https://www.tensorflow.org/guide/estimator)

(<https://www.tensorflow.org/guide/estimator>) (<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/estimator.ipynb>) (<https://github.com/tensorflow/d>)



[Run in Google Colab](#)

Warning: Estimators are not recommended for new code. Estimators run `v1.Session`-style code which is more difficult to write correctly, and can behave unexpectedly, especially when combined with TF 2 code. Estimators do fall under our [compatibility guarantees](#) (<https://tensorflow.org/guide/versions>), but will receive no fixes other than security vulnerabilities. See the [migration guide](#) (<https://tensorflow.org/guide/migrate>) for details.

This document introduces `tf.estimator` —a high-level TensorFlow API. Estimators encapsulate the following actions:

- Training
- Evaluation
- Prediction
- Export for serving

TensorFlow implements several pre-made Estimators. Custom estimators are still supported, but mainly as a backwards compatibility measure. **Custom estimators should not be used for new code.** All Estimators—pre-made or custom ones—are classes based on the `tf.estimator.Estimator` class.

For a quick example, try [Estimator tutorials \(./tutorials/estimator/linear.ipynb\)](#). For an overview of the API design, check the [white paper \(https://arxiv.org/abs/1708.02637\)](#).

## Setup

In [ ]:

```
!pip install -U tensorflow_datasets
```

In [ ]:

```
import tempfile
import os

import tensorflow as tf
import tensorflow_datasets as tfds
```

## Advantages

Similar to a `tf.keras.Model`, an estimator is a model-level abstraction. The `tf.estimator` provides some capabilities currently still under development for `tf.keras`. These are:

- Parameter server based training
- Full [TFX \(http://tensorflow.org/tfx\)](#) integration

## Estimators Capabilities

Estimators provide the following benefits:

- You can run Estimator-based models on a local host or on a distributed multi-server environment without changing your model. Furthermore, you can run Estimator-based models on CPUs, GPUs, or TPUs without recoding your model.
- Estimators provide a safe distributed training loop that controls how and when to:
  - Load data
  - Handle exceptions
  - Create checkpoint files and recover from failures
  - Save summaries for TensorBoard

When writing an application with Estimators, you must separate the data input pipeline from the model. This separation simplifies experiments with different datasets.

## Using pre-made Estimators

Pre-made Estimators enable you to work at a much higher conceptual level than the base TensorFlow APIs. You no longer have to worry about creating the computational graph or sessions since Estimators handle all the "plumbing" for you. Furthermore, pre-made Estimators let you experiment with different model architectures by making only minimal code changes. `tf.estimator.DNNClassifier`, for example, is a pre-made Estimator class that trains classification models based on dense, feed-forward neural networks.

A TensorFlow program relying on a pre-made Estimator typically consists of the following four steps:

## 1. Write an input functions

For example, you might create one function to import the training set and another function to import the test set. Estimators expect their inputs to be formatted as a pair of objects:

- A dictionary in which the keys are feature names and the values are Tensors (or SparseTensors) containing the corresponding feature data
- A Tensor containing one or more labels

The `input_fn` should return a `tf.data.Dataset` that yields pairs in that format.

For example, the following code builds a `tf.data.Dataset` from the Titanic dataset's `train.csv` file:

In [ ]:

```
def train_input_fn():
    titanic_file = tf.keras.utils.get_file("train.csv", "https://storage.googleapis.com/tf-datasets/titanic/train.csv")
    titanic = tf.data.experimental.make_csv_dataset(
        titanic_file, batch_size=32,
        label_name="survived")
    titanic_batches = (
        titanic.cache().repeat().shuffle(500)
        .prefetch(tf.data.AUTOTUNE))
    return titanic_batches
```

The `input_fn` is executed in a `tf.Graph` and can also directly return a `(features_dicts, labels)` pair containing graph tensors, but this is error prone outside of simple cases like returning constants.

## 2. Define the feature columns.

Each `tf.feature_column` identifies a feature name, its type, and any input pre-processing.

For example, the following snippet creates three feature columns.

- The first uses the `age` feature directly as a floating-point input.
- The second uses the `class` feature as a categorical input.
- The third uses the `embark_town` as a categorical input, but uses the `hashing trick` to avoid the need to enumerate the options, and to set the number of options.

For further information, check the [feature columns tutorial](https://www.tensorflow.org/tutorials/keras/feature_columns) ([https://www.tensorflow.org/tutorials/keras/feature\\_columns](https://www.tensorflow.org/tutorials/keras/feature_columns)).

In [ ]:

```
age = tf.feature_column.numeric_column('age')
cls = tf.feature_column.categorical_column_with_vocabulary_list('class', ['First', 'Second', 'Third'])
embark = tf.feature_column.categorical_column_with_hash_bucket('embark_town', 32)
```

## 3. Instantiate the relevant pre-made Estimator.

For example, here's a sample instantiation of a pre-made Estimator named `LinearClassifier`:

In [ ]:

```
model_dir = tempfile.mkdtemp()
model = tf.estimator.LinearClassifier(
    model_dir=model_dir,
    feature_columns=[embark, cls, age],
    n_classes=2
)
```

For more information, you can go the [linear classifier tutorial](https://www.tensorflow.org/tutorials/estimator/linear) (<https://www.tensorflow.org/tutorials/estimator/linear>).

## 4. Call a training, evaluation, or inference method.

All Estimators provide `train`, `evaluate`, and `predict` methods.

In [ ]:

```
model = model.train(input_fn=train_input_fn, steps=100)
```

```
In [ ]:
```

```
result = model.evaluate(train_input_fn, steps=10)

for key, value in result.items():
    print(key, ":", value)
```

```
In [ ]:
```

```
for pred in model.predict(train_input_fn):
    for key, value in pred.items():
        print(key, ":", value)
    break
```

## Benefits of pre-made Estimators

Pre-made Estimators encode best practices, providing the following benefits:

- Best practices for determining where different parts of the computational graph should run, implementing strategies on a single machine or on a cluster.
- Best practices for event (summary) writing and universally useful summaries.

If you don't use pre-made Estimators, you must implement the preceding features yourself.

## Custom Estimators

The heart of every Estimator—whether pre-made or custom—is its *model function*, `model_fn`, which is a method that builds graphs for training, evaluation, and prediction. When you are using a pre-made Estimator, someone else has already implemented the model function. When relying on a custom Estimator, you must write the model function yourself.

Note: A custom `model_fn` will still run in 1.x-style graph mode. This means there is no eager execution and no automatic control dependencies. You should plan to migrate away from `tf.estimator` with custom `model_fn`. The alternative APIs are `tf.keras` and `tf.distribute`. If you still need an `Estimator` for some part of your training you can use the `tf.keras.estimator.model_to_estimator` converter to create an `Estimator` from a `keras.Model`.

## Create an Estimator from a Keras model

You can convert existing Keras models to Estimators with `tf.keras.estimator.model_to_estimator`. This is helpful if you want to modernize your model code, but your training pipeline still requires Estimators.

Instantiate a Keras MobileNet V2 model and compile the model with the optimizer, loss, and metrics to train with:

```
In [ ]:
```

```
keras_mobilenet_v2 = tf.keras.applications.MobileNetV2(
    input_shape=(160, 160, 3), include_top=False)
keras_mobilenet_v2.trainable = False

estimator_model = tf.keras.Sequential([
    keras_mobilenet_v2,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(1)
])

# Compile the model
estimator_model.compile(
    optimizer='adam',
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
    metrics=['accuracy'])
```

Create an `Estimator` from the compiled Keras model. The initial model state of the Keras model is preserved in the created `Estimator`:

```
In [ ]:
```

```
est_mobilenet_v2 = tf.keras.estimator.model_to_estimator(keras_model=estimator_model)
```

Treat the derived `Estimator` as you would with any other `Estimator`.

```
In [ ]:
```

```
IMG_SIZE = 160 # All images will be resized to 160x160
```

```
def preprocess(image, label):
    image = tf.cast(image, tf.float32)
    image = (image/127.5) - 1
    image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
    return image, label
```

```
In [ ]:
```

```
def train_input_fn(batch_size):
    data = tfds.load('cats_vs_dogs', as_supervised=True)
    train_data = data['train']
    train_data = train_data.map(preprocess).shuffle(500).batch(batch_size)
    return train_data
```

To train, call Estimator's train function:

```
In [ ]:
```

```
est_mobilenet_v2.train(input_fn=lambda: train_input_fn(32), steps=50)
```

Similarly, to evaluate, call the Estimator's evaluate function:

```
In [ ]:
```

```
est_mobilenet_v2.evaluate(input_fn=lambda: train_input_fn(32), steps=10)
```

For more details, please refer to the documentation for `tf.keras.estimator.model_to_estimator`.

## Saving object-based checkpoints with Estimator

Estimators by default save checkpoints with variable names rather than the object graph described in the [Checkpoint guide \(checkpoint.ipynb\)](#).

`tf.train.Checkpoint` will read name-based checkpoints, but variable names may change when moving parts of a model outside of the Estimator's `model_fn`. For forwards compatibility saving object-based checkpoints makes it easier to train a model inside an Estimator and then use it outside of one.

```
In [ ]:
```

```
import tensorflow.compat.v1 as tf_compat
```

```
In [ ]:
```

```
def toy_dataset():
    inputs = tf.range(10.)[:, None]
    labels = inputs * 5. + tf.range(5.)[None, :]
    return tf.data.Dataset.from_tensor_slices(
        dict(x=inputs, y=labels)).repeat().batch(2)
```

```
In [ ]:
```

```
class Net(tf.keras.Model):
    """A simple linear model."""

    def __init__(self):
        super(Net, self).__init__()
        self.l1 = tf.keras.layers.Dense(5)

    def call(self, x):
        return self.l1(x)
```

In [ ]:

```
def model_fn(features, labels, mode):
    net = Net()
    opt = tf.keras.optimizers.Adam(0.1)
    ckpt = tf.train.Checkpoint(step=tf.compat.train.get_global_step(),
                               optimizer=opt, net=net)
    with tf.GradientTape() as tape:
        output = net(features['x'])
        loss = tf.reduce_mean(tf.abs(output - features['y']))
    variables = net.trainable_variables
    gradients = tape.gradient(loss, variables)
    return tf.estimator.EstimatorSpec(
        mode,
        loss=loss,
        train_op=tf.group(opt.apply_gradients(zip(gradients, variables)),
                          ckpt.step.assign_add(1)),
        # Tell the Estimator to save "ckpt" in an object-based format.
        scaffold=tf_compat.train.Scaffold(saver=ckpt))

tf.keras.backend.clear_session()
est = tf.estimator.Estimator(model_fn, './tf_estimator_example/')
est.train(toy_dataset, steps=10)
```

`tf.train.Checkpoint` can then load the Estimator's checkpoints from its `model_dir`.

In [ ]:

```
opt = tf.keras.optimizers.Adam(0.1)
net = Net()
ckpt = tf.train.Checkpoint(
    step=tf.Variable(1, dtype=tf.int64), optimizer=opt, net=net)
ckpt.restore(tf.train.latest_checkpoint('./tf_estimator_example/'))
ckpt.step.numpy() # From est.train(..., steps=10)
```

## SavedModels from Estimators

Estimators export SavedModels through `tf.Estimator.export_saved_model`.

In [ ]:

```
input_column = tf.feature_column.numeric_column("x")

estimator = tf.estimator.LinearClassifier(feature_columns=[input_column])

def input_fn():
    return tf.data.Dataset.from_tensor_slices(
        {"x": [1., 2., 3., 4.], [1, 1, 0, 0]}).repeat(200).shuffle(64).batch(16)
estimator.train(input_fn)
```

To save an `Estimator` you need to create a `serving_input_receiver`. This function builds a part of a `tf.Graph` that parses the raw data received by the SavedModel.

The `tf.estimator.export` module contains functions to help build these receivers.

The following code builds a receiver, based on the `feature_columns`, that accepts serialized `tf.Example` protocol buffers, which are often used with [tf-serving \(<https://tensorflow.org/serving>\)](https://tensorflow.org/serving).

In [ ]:

```
tmpdir = tempfile.mkdtemp()

serving_input_fn = tf.estimator.export.build_parsing_serving_input_receiver_fn(
    tf.feature_column.make_parse_example_spec([input_column]))

estimator_base_path = os.path.join(tmpdir, 'from_estimator')
estimator_path = estimator.export_saved_model(estimator_base_path, serving_input_fn)
```

You can also load and run that model, from python:

In [ ]:

```
imported = tf.saved_model.load(estimator_path)

def predict(x):
    example = tf.train.Example()
    example.features.feature["x"].float_list.value.extend([x])
    return imported.signatures["predict"](
        examples=tf.constant([example.SerializeToString()]))
```

In [ ]:

```
print(predict(1.5))
print(predict(3.5))
```

`tf.estimator.export.build_raw_serving_input_receiver_fn` allows you to create input functions which take raw tensors rather than `tf.train.Example`s.

## Using `tf.distribute.Strategy` with Estimator (Limited support)

`tf.estimator` is a distributed training TensorFlow API that originally supported the async parameter server approach. `tf.estimator` now supports `tf.distribute.Strategy`. If you're using `tf.estimator`, you can change to distributed training with very few changes to your code. With this, Estimator users can now do synchronous distributed training on multiple GPUs and multiple workers, as well as use TPUs. This support in Estimator is, however, limited. Check out the [What's supported now](#) section below for more details.

Using `tf.distribute.Strategy` with Estimator is slightly different than in the Keras case. Instead of using `strategy.scope`, now you pass the `strategy` object into the `RunConfig` for the Estimator.

You can refer to the [distributed training guide \(distributed\\_training.ipynb\)](#) for more information.

Here is a snippet of code that shows this with a premade Estimator `LinearRegressor` and `MirroredStrategy`:

In [ ]:

```
mirrored_strategy = tf.distribute.MirroredStrategy()
config = tf.estimator.RunConfig(
    train_distribute=mirrored_strategy, eval_distribute=mirrored_strategy)
regressor = tf.estimator.LinearRegressor(
    feature_columns=[tf.feature_column.numeric_column('feats')],
    optimizer='SGD',
    config=config)
```

Here, you use a premade Estimator, but the same code works with a custom Estimator as well. `train_distribute` determines how training will be distributed, and `eval_distribute` determines how evaluation will be distributed. This is another difference from Keras where you use the same strategy for both training and eval.

Now you can train and evaluate this Estimator with an input function:

In [ ]:

```
def input_fn():
    dataset = tf.data.Dataset.from_tensors(({ "feats": [1.]}, [1.]))
    return dataset.repeat(1000).batch(10)
regressor.train(input_fn=input_fn, steps=10)
regressor.evaluate(input_fn=input_fn, steps=10)
```

Another difference to highlight here between Estimator and Keras is the input handling. In Keras, each batch of the dataset is split automatically across the multiple replicas. In Estimator, however, you do not perform automatic batch splitting, nor automatically shard the data across different workers. You have full control over how you want your data to be distributed across workers and devices, and you must provide an `input_fn` to specify how to distribute your data.

Your `input_fn` is called once per worker, thus giving one dataset per worker. Then one batch from that dataset is fed to one replica on that worker, thereby consuming N batches for N replicas on 1 worker. In other words, the dataset returned by the `input_fn` should provide batches of size `PER_REPLICA_BATCH_SIZE`. And the global batch size for a step can be obtained as `PER_REPLICA_BATCH_SIZE * strategy.num_replicas_in_sync`.

When performing multi-worker training, you should either split your data across the workers, or shuffle with a random seed on each. You can check an example of how to do this in the [Multi-worker training with Estimator \(./tutorials/distribute/multi\\_worker\\_with\\_estimator.ipynb\)](#) tutorial.

And similarly, you can use multi worker and parameter server strategies as well. The code remains the same, but you need to use `tf.estimator.train_and_evaluate`, and set `TF_CONFIG` environment variables for each binary running in your cluster.

## What's supported now?

There is limited support for training with Estimator using all strategies except `TPUStrategy`. Basic training and evaluation should work, but a number of advanced features such as `v1.train.Scaffold` do not. There may also be a number of bugs in this integration and there are no plans to actively improve this support (the focus is on Keras and custom training loop support). If at all possible, you should prefer to use `tf.distribute` with those APIs instead.

Training API	MirroredStrategy	TPUStrategy	MultiWorkerMirroredStrategy	CentralStorageStrategy	ParameterServerStrategy
Estimator API	Limited support	Not supported	Limited support	Limited support	Limited support

## Examples and tutorials

Here are some end-to-end examples that show how to use various strategies with Estimator:

1. The [Multi-worker Training with Estimator tutorial \(..../tutorials/distribute/multi\\_worker\\_with\\_estimator.ipynb\)](#) shows how you can train with multiple workers using `MultiWorkerMirroredStrategy` on the MNIST dataset.
2. An end-to-end example of [running multi-worker training with distribution strategies \(https://github.com/tensorflow/ecosystem/tree/master/distribution\\_strategy\)](#) in tensorflow/ecosystem using Kubernetes templates. It starts with a Keras model and converts it to an Estimator using the `tf.keras.estimator.model_to_estimator` API.
3. The official [ResNet50 \(https://github.com/tensorflow/models/blob/master/official/vision/image\\_classification/resnet\\_imagenet\\_main.py\)](#) model, which can be trained using either `MirroredStrategy` or `MultiWorkerMirroredStrategy`.

Copyright 2020 The TensorFlow Authors.

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## TensorFlow graph optimization with Grappler



[View on TensorFlow.org](#)

([https://www.tensorflow.org/guide/graph\\_optimization](https://www.tensorflow.org/guide/graph_optimization))



[Run in Google Colab](#)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/graph\\_optimization.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/graph_optimization.ipynb))

## Overview

TensorFlow uses both graph and eager executions to execute computations. A `tf.Graph` contains a set of `tf.Operation` objects (ops) which represent units of computation and `tf.Tensor` objects which represent the units of data that flow between ops.

Grappler is the default graph optimization system in the TensorFlow runtime. Grappler applies optimizations in graph mode (within `tf.function`) to improve the performance of your TensorFlow computations through graph simplifications and other high-level optimizations such as inlining function bodies to enable inter-procedural optimizations. Optimizing the `tf.Graph` also reduces the device peak memory usage and improves hardware utilization by optimizing the mapping of graph nodes to compute resources.

Use `tf.config.optimizer.set_experimental_options()` for finer control over your `tf.Graph` optimizations.

## Available graph optimizers

Grappler performs graph optimizations through a top-level driver called the `MetaOptimizer`. The following graph optimizers are available with TensorFlow:

- *Constant folding optimizer* - Statically infers the value of tensors when possible by folding constant nodes in the graph and materializes the result using constants.
- *Arithmetic optimizer* - Simplifies arithmetic operations by eliminating common subexpressions and simplifying arithmetic statements.
- *Layout optimizer* - Optimizes tensor layouts to execute data format dependent operations such as convolutions more efficiently.
- *Remapper optimizer* - Remaps subgraphs onto more efficient implementations by replacing commonly occurring subgraphs with optimized fused monolithic kernels.
- *Memory optimizer* - Analyzes the graph to inspect the peak memory usage for each operation and inserts CPU-GPU memory copy operations for swapping GPU memory to CPU to reduce the peak memory usage.
- *Dependency optimizer* - Removes or rearranges control dependencies to shorten the critical path for a model step or enables other optimizations. Also removes nodes that are effectively no-ops such as Identity.
- *Pruning optimizer* - Prunes nodes that have no effect on the output from the graph. It is usually run first to reduce the size of the graph and speed up processing in other Grappler passes.
- *Function optimizer* - Optimizes the function library of a TensorFlow program and inlines function bodies to enable other inter-procedural optimizations.
- *Shape optimizer* - Optimizes subgraphs that operate on shape and shape related information.
- *Autoparallel optimizer* - Automatically parallelizes graphs by splitting along the batch dimension. This optimizer is turned OFF by default.
- *Loop optimizer* - Optimizes the graph control flow by hoisting loop-invariant subgraphs out of loops and by removing redundant stack operations in loops. Also optimizes loops with statically known trip counts and removes statically known dead branches in conditionals.
- *Scoped allocator optimizer* - Introduces scoped allocators to reduce data movement and to consolidate some operations.
- *Pin to host optimizer* - Swaps small operations onto the CPU. This optimizer is turned OFF by default.
- *Auto mixed precision optimizer* - Converts data types to float16 where applicable to improve performance. Currently applies only to GPUs.
- *Debug stripper* - Strips nodes related to debugging operations such as `tf.debugging.Assert`, `tf.debugging.check_numerics`, and `tf.print` from the graph. This optimizer is turned OFF by default.

## Setup

In [ ]:

```
import numpy as np
import timeit
import traceback
import contextlib

import tensorflow as tf
```

Create a context manager to easily toggle optimizer states.

In [ ]:

```
@contextlib.contextmanager
def options(options):
    old_opts = tf.config.optimizer.get_experimental_options()
    tf.config.optimizer.set_experimental_options(options)
    try:
        yield
    finally:
        tf.config.optimizer.set_experimental_options(old_opts)
```

## Compare execution performance with and without Grappler

TensorFlow 2 and beyond executes [eagerly \(../eager.md\)](#) by default. Use `tf.function` to switch the default execution to Graph mode. Grappler runs automatically in the background to apply the graph optimizations above and improve execution performance.

### Constant folding optimizer

As a preliminary example, consider a function which performs operations on constants and returns an output.

```
In [ ]:
```

```
def test_function_1():
    @tf.function
    def simple_function(input_arg):
        print('Tracing!')
        a = tf.constant(np.random.randn(2000,2000), dtype = tf.float32)
        c = a
        for n in range(50):
            c = c@a
        return tf.reduce_mean(c+input_arg)

    return simple_function
```

Turn off the constant folding optimizer and execute the function:

```
In [ ]:
```

```
with options({'constant_folding': False}):
    print(tf.config.optimizer.get_experimental_options())
    simple_function = test_function_1()
    # Trace once
    x = tf.constant(2.2)
    simple_function(x)
    print("Vanilla execution:", timeit.timeit(lambda: simple_function(x), number = 1), "s")
```

Enable the constant folding optimizer and execute the function again to observe a speed-up in function execution.

```
In [ ]:
```

```
with options({'constant_folding': True}):
    print(tf.config.optimizer.get_experimental_options())
    simple_function = test_function_1()
    # Trace once
    x = tf.constant(2.2)
    simple_function(x)
    print("Constant folded execution:", timeit.timeit(lambda: simple_function(x), number = 1), "s")
```

## Debug stripper optimizer

Consider a simple function that checks the numeric value of its input argument and returns it.

```
In [ ]:
```

```
def test_function_2():
    @tf.function
    def simple_func(input_arg):
        output = input_arg
        tf.debugging.check_numerics(output, "Bad!")
        return output
    return simple_func
```

First, execute the function with the debug stripper optimizer turned off.

```
In [ ]:
```

```
test_func = test_function_2()
p1 = tf.constant(float('inf'))
try:
    test_func(p1)
except tf.errors.InvalidArgumentError as e:
    traceback.print_exc(limit=2)
```

`tf.debugging.check_numerics` raises an invalid argument error because of the `Inf` argument to `test_func`.

Enable the debug stripper optimizer and execute the function again.

```
In [ ]:
```

```
with options({'debug_stripper': True}):
    test_func2 = test_function_2()
    p1 = tf.constant(float('inf'))
    try:
        test_func2(p1)
    except tf.errors.InvalidArgumentError as e:
        traceback.print_exc(limit=2)
```

The debug stripper optimizer strips the `tf.debug.check_numerics` node from the graph and executes the function without raising any errors.

## Summary

The TensorFlow runtime uses Grappler to optimize graphs automatically before execution. Use `tf.config.optimizer.set_experimental_options` to enable or disable the various graph optimizers.

For more information on Grappler, see [TensorFlow Graph Optimizations \(http://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf\)](http://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf).

**Copyright 2020 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Advanced automatic differentiation



[View on TensorFlow.org](https://www.tensorflow.org/guide/advanced_autodiff)

([https://www.tensorflow.org/guide/advanced\\_autodiff](https://www.tensorflow.org/guide/advanced_autodiff))



[Run in Google Colab](#)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/advanced\\_autodiff.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/advanced_autodiff.ipynb))

The [Introduction to gradients and automatic differentiation \(autodiff.ipynb\)](#) guide includes everything required to calculate gradients in TensorFlow. This guide focuses on deeper, less common features of the `tf.GradientTape` API.

## Setup

In [ ]:

```
import tensorflow as tf
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rcParams['figure.figsize'] = (8, 6)
```

## Controlling gradient recording

In the [automatic differentiation guide \(autodiff.ipynb\)](#) you saw how to control which variables and tensors are watched by the tape while building the gradient calculation.

The tape also has methods to manipulate the recording.

### Stop recording

If you wish to stop recording gradients, you can use `tf.GradientTape.stop_recording` to temporarily suspend recording.

This may be useful to reduce overhead if you do not wish to differentiate a complicated operation in the middle of your model. This could include calculating a metric or an intermediate result:

In [ ]:

```
x = tf.Variable(2.0)
y = tf.Variable(3.0)

with tf.GradientTape() as t:
    x_sq = x * x
    with t.stop_recording():
        y_sq = y * y
    z = x_sq + y_sq

grad = t.gradient(z, {'x': x, 'y': y})

print('dz/dx:', grad['x']) # 2*x => 4
print('dz/dy:', grad['y'])
```

## Reset/start recording from scratch

If you wish to start over entirely, use `tf.GradientTape.reset`. Simply exiting the gradient tape block and restarting is usually easier to read, but you can use the `reset` method when exiting the tape block is difficult or impossible.

In [ ]:

```
x = tf.Variable(2.0)
y = tf.Variable(3.0)
reset = True

with tf.GradientTape() as t:
    y_sq = y * y
    if reset:
        # Throw out all the tape recorded so far.
        t.reset()
    z = x * x + y_sq

grad = t.gradient(z, {'x': x, 'y': y})

print('dz/dx:', grad['x']) # 2*x => 4
print('dz/dy:', grad['y'])
```

## Stop gradient flow with precision

In contrast to the global tape controls above, the `tf.stop_gradient` function is much more precise. It can be used to stop gradients from flowing along a particular path, without needing access to the tape itself:

In [ ]:

```
x = tf.Variable(2.0)
y = tf.Variable(3.0)

with tf.GradientTape() as t:
    y_sq = y**2
    z = x**2 + tf.stop_gradient(y_sq)

grad = t.gradient(z, {'x': x, 'y': y})

print('dz/dx:', grad['x']) # 2*x => 4
print('dz/dy:', grad['y'])
```

## Custom gradients

In some cases, you may want to control exactly how gradients are calculated rather than using the default. These situations include:

1. There is no defined gradient for a new op you are writing.
2. The default calculations are numerically unstable.
3. You wish to cache an expensive computation from the forward pass.
4. You want to modify a value (for example, using `tf.clip_by_value` or `tf.math.round`) without modifying the gradient.

For the first case, to write a new op you can use `tf.RegisterGradient` to set up your own (refer to the API docs for details). (Note that the gradient registry is global, so change it with caution.)

For the latter three cases, you can use `tf.custom_gradient`.

Here is an example that applies `tf.clip_by_norm` to the intermediate gradient:

In [ ]:

```
# Establish an identity operation, but clip during the gradient pass.
@tf.custom_gradient
def clip_gradients(y):
    def backward(dy):
        return tf.clip_by_norm(dy, 0.5)
    return y, backward

v = tf.Variable(2.0)
with tf.GradientTape() as t:
    output = clip_gradients(v * v)
print(t.gradient(output, v)) # calls "backward", which clips 4 to 2
```

Refer to the `tf.custom_gradient` decorator API docs for more details.

## Custom gradients in SavedModel

Note: This feature is available from TensorFlow 2.6.

Custom gradients can be saved to SavedModel by using the option `tf.saved_model.SaveOptions(experimental_custom_gradients=True)`.

To be saved into the SavedModel, the gradient function must be traceable (to learn more, check out the [Better performance with `tf.function` \(`function.ipynb`\) guide](#)).

In [ ]:

```
class MyModule(tf.Module):

    @tf.function(input_signature=[tf.TensorSpec(None)])
    def call_custom_grad(self, x):
        return clip_gradients(x)

model = MyModule()
```

In [ ]:

```
tf.saved_model.save(
    model,
    'saved_model',
    options=tf.saved_model.SaveOptions(experimental_custom_gradients=True))

# The loaded gradients will be the same as the above example.
v = tf.Variable(2.0)
loaded = tf.saved_model.load('saved_model')
with tf.GradientTape() as t:
    output = loaded.call_custom_grad(v * v)
print(t.gradient(output, v))
```

A note about the above example: If you try replacing the above code with `tf.saved_model.SaveOptions(experimental_custom_gradients=False)`, the gradient will still produce the same result on loading. The reason is that the gradient registry still contains the custom gradient used in the function `call_custom_op`. However, if you restart the runtime after saving without custom gradients, running the loaded model under the `tf.GradientTape` will throw the error: `LookupError: No gradient defined for operation 'IdentityN' (op type: IdentityN)`.

## Multiple tapes

Multiple tapes interact seamlessly.

For example, here each tape watches a different set of tensors:

In [ ]:

```
x0 = tf.constant(0.0)
x1 = tf.constant(0.0)

with tf.GradientTape() as tape0, tf.GradientTape() as tape1:
    tape0.watch(x0)
    tape1.watch(x1)

    y0 = tf.math.sin(x0)
    y1 = tf.nn.sigmoid(x1)

    y = y0 + y1

    ys = tf.reduce_sum(y)
```

```
In [ ]:
```

```
tape0.gradient(ys, x0).numpy() # cos(x) => 1.0
```

```
In [ ]:
```

```
tape1.gradient(ys, x1).numpy() # sigmoid(x1)*(1-sigmoid(x1)) => 0.25
```

## Higher-order gradients

Operations inside of the `tf.GradientTape` context manager are recorded for automatic differentiation. If gradients are computed in that context, then the gradient computation is recorded as well. As a result, the exact same API works for higher-order gradients as well.

For example:

```
In [ ]:
```

```
x = tf.Variable(1.0) # Create a Tensorflow variable initialized to 1.0

with tf.GradientTape() as t2:
    with tf.GradientTape() as t1:
        y = x * x * x

    # Compute the gradient inside the outer `t2` context manager
    # which means the gradient computation is differentiable as well.
    dy_dx = t1.gradient(y, x)
d2y_dx2 = t2.gradient(dy_dx, x)

print('dy_dx:', dy_dx.numpy()) # 3 * x**2 => 3.0
print('d2y_dx2:', d2y_dx2.numpy()) # 6 * x => 6.0
```

While that does give you the second derivative of a *scalar* function, this pattern does not generalize to produce a Hessian matrix, since `tf.GradientTape.gradient` only computes the gradient of a scalar. To construct a [Hessian matrix](https://en.wikipedia.org/wiki/Hessian_matrix) ([https://en.wikipedia.org/wiki/Hessian\\_matrix](https://en.wikipedia.org/wiki/Hessian_matrix)), go to the [Hessian example](#) under the [Jacobian section](#).

"Nested calls to `tf.GradientTape.gradient`" is a good pattern when you are calculating a scalar from a gradient, and then the resulting scalar acts as a source for a second gradient calculation, as in the following example.

### Example: Input gradient regularization

Many models are susceptible to "adversarial examples". This collection of techniques modifies the model's input to confuse the model's output. The simplest implementation—such as the [Adversarial example using the Fast Gradient Signed Method attack](#) ([https://www.tensorflow.org/tutorials/generative/adversarial\\_fgsm](https://www.tensorflow.org/tutorials/generative/adversarial_fgsm))—takes a single step along the gradient of the output with respect to the input; the "input gradient".

One technique to increase robustness to adversarial examples is [input gradient regularization](#) (<https://arxiv.org/abs/1905.11468>) (Finlay & Oberman, 2019), which attempts to minimize the magnitude of the input gradient. If the input gradient is small, then the change in the output should be small too.

Below is a naive implementation of input gradient regularization. The implementation is:

1. Calculate the gradient of the output with respect to the input using an inner tape.
2. Calculate the magnitude of that input gradient.
3. Calculate the gradient of that magnitude with respect to the model.

```
In [ ]:
```

```
x = tf.random.normal([7, 5])

layer = tf.keras.layers.Dense(10, activation=tf.nn.relu)
```

```
In [ ]:
```

```
with tf.GradientTape() as t2:
    # The inner tape only takes the gradient with respect to the input,
    # not the variables.
    with tf.GradientTape(watch_accessed_variables=False) as t1:
        t1.watch(x)
        y = layer(x)
        out = tf.reduce_sum(layer(x)**2)

    # 1. Calculate the input gradient.
    g1 = t1.gradient(out, x)
    # 2. Calculate the magnitude of the input gradient.
    g1_mag = tf.norm(g1)

    # 3. Calculate the gradient of the magnitude with respect to the model.
    dg1_mag = t2.gradient(g1_mag, layer.trainable_variables)
```

In [ ]:

```
[var.shape for var in dg1_mag]
```

## Jacobians

All the previous examples took the gradients of a scalar target with respect to some source tensor(s).

The [Jacobian matrix](https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant) ([https://en.wikipedia.org/wiki/Jacobian\\_matrix\\_and\\_determinant](https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant)) represents the gradients of a vector valued function. Each row contains the gradient of one of the vector's elements.

The `tf.GradientTape.jacobian` method allows you to efficiently calculate a Jacobian matrix.

Note that:

- Like `gradient`: The `sources` argument can be a tensor or a container of tensors.
- Unlike `gradient`: The target tensor must be a single tensor.

### Scalar source

As a first example, here is the Jacobian of a vector-target with respect to a scalar-source.

In [ ]:

```
x = tf.linspace(-10.0, 10.0, 200+1)
delta = tf.Variable(0.0)

with tf.GradientTape() as tape:
    y = tf.nn.sigmoid(x+delta)

dy_dx = tape.jacobian(y, delta)
```

When you take the Jacobian with respect to a scalar the result has the shape of the `target`, and gives the gradient of the each element with respect to the source:

In [ ]:

```
print(y.shape)
print(dy_dx.shape)
```

In [ ]:

```
plt.plot(x.numpy(), y, label='y')
plt.plot(x.numpy(), dy_dx, label='dy/dx')
plt.legend()
_ = plt.xlabel('x')
```

### Tensor source

Whether the input is scalar or tensor, `tf.GradientTape.jacobian` efficiently calculates the gradient of each element of the source with respect to each element of the target(s).

For example, the output of this layer has a shape of (10, 7) :

In [ ]:

```
x = tf.random.normal([7, 5])
layer = tf.keras.layers.Dense(10, activation=tf.nn.relu)

with tf.GradientTape(persistent=True) as tape:
    y = layer(x)

y.shape
```

And the layer's kernel's shape is (5, 10) :

In [ ]:

```
layer.kernel.shape
```

The shape of the Jacobian of the output with respect to the kernel is those two shapes concatenated together:

In [ ]:

```
j = tape.jacobian(y, layer.kernel)
j.shape
```

If you sum over the target's dimensions, you're left with the gradient of the sum that would have been calculated by `tf.GradientTape.gradient`:

In [ ]:

```
g = tape.gradient(y, layer.kernel)
print('g.shape:', g.shape)

j_sum = tf.reduce_sum(j, axis=[0, 1])
delta = tf.reduce_max(tf.abs(g - j_sum)).numpy()
assert delta < 1e-3
print('delta:', delta)
```

</hessian>

## Example: Hessian

While `tf.GradientTape` doesn't give an explicit method for constructing a [Hessian matrix](https://en.wikipedia.org/wiki/Hessian_matrix) ([https://en.wikipedia.org/wiki/Hessian\\_matrix](https://en.wikipedia.org/wiki/Hessian_matrix)) it's possible to build one using the `tf.GradientTape.jacobian` method.

Note: The Hessian matrix contains  $N^2$  parameters. For this and other reasons it is not practical for most models. This example is included more as a demonstration of how to use the `tf.GradientTape.jacobian` method, and is not an endorsement of direct Hessian-based optimization. A Hessian-vector product can be [calculated efficiently with nested tapes](#) ([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/eager/benchmarks/resnet50/hvp\\_test.py](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/eager/benchmarks/resnet50/hvp_test.py)), and is a much more efficient approach to second-order optimization.

In [ ]:

```
x = tf.random.normal([7, 5])
layer1 = tf.keras.layers.Dense(8, activation=tf.nn.relu)
layer2 = tf.keras.layers.Dense(6, activation=tf.nn.relu)

with tf.GradientTape() as t2:
    with tf.GradientTape() as t1:
        x = layer1(x)
        x = layer2(x)
        loss = tf.reduce_mean(x**2)

    g = t1.gradient(loss, layer1.kernel)
    h = t2.jacobian(g, layer1.kernel)
```

In [ ]:

```
print(f'layer.kernel.shape: {layer1.kernel.shape}')
print(f'h.shape: {h.shape}')
```

To use this Hessian for a [Newton's method](https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization) ([https://en.wikipedia.org/wiki/Newton%27s\\_method\\_in\\_optimization](https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization)) step, you would first flatten out its axes into a matrix, and flatten out the gradient into a vector:

In [ ]:

```
n_params = tf.reduce_prod(layer1.kernel.shape)
g_vec = tf.reshape(g, [n_params, 1])
h_mat = tf.reshape(h, [n_params, n_params])
```

The Hessian matrix should be symmetric:

In [ ]:

```
def imshow_zero_center(image, **kwargs):
    lim = tf.reduce_max(tf.abs(image))
    plt.imshow(image, vmin=-lim, vmax=lim, cmap='seismic', **kwargs)
    plt.colorbar()
```

In [ ]:

```
imshow_zero_center(h_mat)
```

The Newton's method update step is shown below:

In [ ]:

```
eps = 1e-3
eye_eps = tf.eye(h_mat.shape[0])*eps
```

Note: [Don't actually invert the matrix \(https://www.johndcook.com/blog/2010/01/19/dont-invert-that-matrix/\)](https://www.johndcook.com/blog/2010/01/19/dont-invert-that-matrix/).

In [ ]:

```
# X(k+1) = X(k) - ( $\nabla^2 f(X(k))$ ) $^{-1}$  @  $\nabla f(X(k))$ 
# h_mat =  $\nabla^2 f(X(k))$ 
# g_vec =  $\nabla f(X(k))$ 
update = tf.linalg.solve(h_mat + eye_eps, g_vec)

# Reshape the update and apply it to the variable.
_ = layer1.kernel.assign_sub(tf.reshape(update, layer1.kernel.shape))
```

While this is relatively simple for a single `tf.Variable`, applying this to a non-trivial model would require careful concatenation and slicing to produce a full Hessian across multiple variables.

## Batch Jacobian

In some cases, you want to take the Jacobian of each of a stack of targets with respect to a stack of sources, where the Jacobians for each target-source pair are independent.

For example, here the input `x` is shaped `(batch, ins)` and the output `y` is shaped `(batch, outs)`:

In [ ]:

```
x = tf.random.normal([7, 5])

layer1 = tf.keras.layers.Dense(8, activation=tf.nn.elu)
layer2 = tf.keras.layers.Dense(6, activation=tf.nn.elu)

with tf.GradientTape(persistent=True, watch_accessed_variables=False) as tape:
    tape.watch(x)
    y = layer1(x)
    y = layer2(y)

y.shape
```

The full Jacobian of `y` with respect to `x` has a shape of `(batch, ins, batch, outs)`, even if you only want `(batch, ins, outs)`:

In [ ]:

```
j = tape.jacobian(y, x)
j.shape
```

If the gradients of each item in the stack are independent, then every `(batch, batch)` slice of this tensor is a diagonal matrix:

In [ ]:

```
imshow_zero_center(j[:, 0, :, 0])
_ = plt.title('A (batch, batch) slice')
```

In [ ]:

```
def plot_as_patches(j):
    # Reorder axes so the diagonals will each form a contiguous patch.
    j = tf.transpose(j, [1, 0, 3, 2])
    # Pad in between each patch.
    lim = tf.reduce_max(tf.abs(j))
    j = tf.pad(j, [[0, 0], [1, 1], [0, 0], [1, 1]], constant_values=-lim)
    # Reshape to form a single image.
    s = j.shape
    j = tf.reshape(j, [s[0]*s[1], s[2]*s[3]])
    imshow_zero_center(j, extent=[-0.5, s[2]-0.5, s[0]-0.5, -0.5])

plot_as_patches(j)
_ = plt.title('All (batch, batch) slices are diagonal')
```

To get the desired result, you can sum over the duplicate batch dimension, or else select the diagonals using `tf.einsum`:

In [ ]:

```
j_sum = tf.reduce_sum(j, axis=2)
print(j_sum.shape)
j_select = tf.einsum('bxby->bxy', j)
print(j_select.shape)
```

It would be much more efficient to do the calculation without the extra dimension in the first place. The `tf.GradientTape.batch_jacobian` method does exactly that:

In [ ]:

```
jb = tape.batch_jacobian(y, x)
jb.shape
```

In [ ]:

```
error = tf.reduce_max(abs(jb - j_sum))
assert error < 1e-3
print(error.numpy())
```

Caution: `tf.GradientTape.batch_jacobian` only verifies that the first dimension of the source and target match. It doesn't check that the gradients are actually independent. It's up to you to make sure you only use `batch_jacobian` where it makes sense. For example, adding a `tf.keras.layers.BatchNormalization` destroys the independence, since it normalizes across the batch dimension:

In [ ]:

```
x = tf.random.normal([7, 5])

layer1 = tf.keras.layers.Dense(8, activation=tf.nn.elu)
bn = tf.keras.layers.BatchNormalization()
layer2 = tf.keras.layers.Dense(6, activation=tf.nn.elu)

with tf.GradientTape(persistent=True, watch_accessed_variables=False) as tape:
    tape.watch(x)
    y = layer1(x)
    y = bn(y, training=True)
    y = layer2(y)

j = tape.jacobian(y, x)
print(f'j.shape: {j.shape}')
```

In [ ]:

```
plot_as_patches(j)

_ = plt.title('These slices are not diagonal')
_ = plt.xlabel("Don't use `batch_jacobian`")
```

In this case, `batch_jacobian` still runs and returns something with the expected shape, but its contents have an unclear meaning:

In [ ]:

```
jb = tape.batch_jacobian(y, x)
print(f'jb.shape: {jb.shape}')
```

**Copyright 2019 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

# Random number generation



[View on TensorFlow.org](https://www.tensorflow.org/guide/random_numbers)

([https://www.tensorflow.org/guide/random\\_numbers](https://www.tensorflow.org/guide/random_numbers))



[Run in Google Colab](#)

TensorFlow provides a set of pseudo-random number generators (RNG), in the `tf.random` module. This document describes how you can control the random number generators, and how these generators interact with other tensorflow sub-systems.

Note: The random numbers are not guaranteed to be consistent across TensorFlow versions. See: [Version Compatibility](#) ([https://www.tensorflow.org/guide/versions#what\\_is\\_not\\_covered](https://www.tensorflow.org/guide/versions#what_is_not_covered))

TensorFlow provides two approaches for controlling the random number generation process:

1. Through the explicit use of `tf.random.Generator` objects. Each such object maintains a state (in `tf.Variable`) that will be changed after each number generation.
2. Through the purely-functional stateless random functions like `tf.random.stateless_uniform`. Calling these functions with the same arguments (which include the seed) and on the same device will always produce the same results.

Warning: The old RNGs from TF 1.x such as `tf.random.uniform` and `tf.random.normal` are not yet deprecated but strongly discouraged.

## Setup

In [ ]:

```
import tensorflow as tf

# Creates some virtual devices (cpu:0, cpu:1, etc.) for using distribution strategy
physical_devices = tf.config.list_physical_devices("CPU")
tf.config.experimental.set_virtual_device_configuration(
    physical_devices[0], [
        tf.config.experimental.VirtualDeviceConfiguration(),
        tf.config.experimental.VirtualDeviceConfiguration(),
        tf.config.experimental.VirtualDeviceConfiguration()
    ])
```

## The `tf.random.Generator` class

The `tf.random.Generator` class is used in cases where you want each RNG call to produce different results. It maintains an internal state (managed by a `tf.Variable` object) which will be updated every time random numbers are generated. Because the state is managed by `tf.Variable`, it enjoys all facilities provided by `tf.Variable` such as easy checkpointing, automatic control-dependency and thread safety.

You can get a `tf.random.Generator` by manually creating an object of the class or call `tf.random.get_global_generator()` to get the default global generator:

In [ ]:

```
g1 = tf.random.Generator.from_seed(1)
print(g1.normal(shape=[2, 3]))
g2 = tf.random.get_global_generator()
print(g2.normal(shape=[2, 3]))
```

There are multiple ways to create a generator object. The easiest is `Generator.from_seed`, as shown above, that creates a generator from a seed. A seed is any non-negative integer. `from_seed` also takes an optional argument `alg` which is the RNG algorithm that will be used by this generator:

In [ ]:

```
g1 = tf.random.Generator.from_seed(1, alg='philox')
print(g1.normal(shape=[2, 3]))
```

See the *Algorithms* section below for more information about it.

Another way to create a generator is with `Generator.from_non_deterministic_state`. A generator created this way will start from a non-deterministic state, depending on e.g. time and OS.

In [ ]:

```
g = tf.random.Generator.from_non_deterministic_state()
print(g.normal(shape=[2, 3]))
```

There are yet other ways to create generators, such as from explicit states, which are not covered by this guide.

When using `tf.random.get_global_generator` to get the global generator, you need to be careful about device placement. The global generator is created (from a non-deterministic state) at the first time `tf.random.get_global_generator` is called, and placed on the default device at that call. So, for example, if the first site you call `tf.random.get_global_generator` is within a `tf.device("gpu")` scope, the global generator will be placed on the GPU, and using the global generator later on from the CPU will incur a GPU-to-CPU copy.

There is also a function `tf.random.set_global_generator` for replacing the global generator with another generator object. This function should be used with caution though, because the old global generator may have been captured by a `tf.function` (as a weak reference), and replacing it will cause it to be garbage collected, breaking the `tf.function`. A better way to reset the global generator is to use one of the "reset" functions such as `Generator.reset_from_seed`, which won't create new generator objects.

In [ ]:

```
g = tf.random.Generator.from_seed(1)
print(g.normal([]))
print(g.normal([]))
g.reset_from_seed(1)
print(g.normal([]))
```

## Creating independent random-number streams

In many applications one needs multiple independent random-number streams, independent in the sense that they won't overlap and won't have any statistically detectable correlations. This is achieved by using `Generator.split` to create multiple generators that are guaranteed to be independent of each other (i.e. generating independent streams).

In [ ]:

```
g = tf.random.Generator.from_seed(1)
print(g.normal([]))
new_gs = g.split(3)
for new_g in new_gs:
    print(new_g.normal([]))
print(g.normal([]))
```

`split` will change the state of the generator on which it is called (`g` in the above example), similar to an RNG method such as `normal`. In addition to being independent of each other, the new generators (`new_gs`) are also guaranteed to be independent of the old one (`g`).

Spawning new generators is also useful when you want to make sure the generator you use is on the same device as other computations, to avoid the overhead of cross-device copy. For example:

In [ ]:

```
with tf.device("cpu"): # change "cpu" to the device you want
    g = tf.random.get_global_generator().split(1)[0]
    print(g.normal()) # use of g won't cause cross-device copy, unlike the global generator
```

Note: In theory, you can use constructors such as `from_seed` instead of `split` here to obtain a new generator, but by doing so you lose the guarantee that the new generator is independent of the global generator. You will also run the risk that you may accidentally create two generators with the same seed or with seeds that lead to overlapping random-number streams.

You can do splitting recursively, calling `split` on splitted generators. There are no limits (barring integer overflow) on the depth of recursions.

## Interaction with `tf.function`

`tf.random.Generator` obeys the same rules as `tf.Variable` when used with `tf.function`. This includes three aspects.

### Creating generators outside `tf.function`

`tf.function` can use a generator created outside of it.

In [ ]:

```
g = tf.random.Generator.from_seed(1)
@tf.function
def foo():
    return g.normal([])
print(foo())
```

The user needs to make sure that the generator object is still alive (not garbage-collected) when the function is called.

## Creating generators inside `tf.function`

Creation of generators inside a `tf.function` can only happen during the first run of the function.

In [ ]:

```
g = None
@tf.function
def foo():
    global g
    if g is None:
        g = tf.random.Generator.from_seed(1)
    return g.normal([])
print(foo())
print(foo())
```

## Passing generators as arguments to `tf.function`

When used as an argument to a `tf.function`, different generator objects will cause retracing of the `tf.function`.

In [ ]:

```
num_traces = 0
@tf.function
def foo(g):
    global num_traces
    num_traces += 1
    return g.normal([])
foo(tf.random.Generator.from_seed(1))
foo(tf.random.Generator.from_seed(2))
print(num_traces)
```

Note that this retracing behavior is consistent with `tf.Variable`:

In [ ]:

```
num_traces = 0
@tf.function
def foo(v):
    global num_traces
    num_traces += 1
    return v.read_value()
foo(tf.Variable(1))
foo(tf.Variable(2))
print(num_traces)
```

## Interaction with distribution strategies

There are two ways in which `Generator` interacts with distribution strategies.

### Creating generators outside distribution strategies

If a generator is created outside strategy scopes, all replicas' access to the generator will be serialized, and hence the replicas will get different random numbers.

In [ ]:

```
g = tf.random.Generator.from_seed(1)
strat = tf.distribute.MirroredStrategy(devices=["cpu:0", "cpu:1"])
with strat.scope():
    def f():
        print(g.normal([]))
    results = strat.run(f)
```

Note that this usage may have performance issues because the generator's device is different from the replicas.

### Creating generators inside distribution strategies

If a generator is created inside a strategy scope, each replica will get a different and independent stream of random numbers.

In [ ]:

```
strat = tf.distribute.MirroredStrategy(devices=["cpu:0", "cpu:1"])
with strat.scope():
    g = tf.random.Generator.from_seed(1)
    print(strat.run(lambda: g.normal([])))
    print(strat.run(lambda: g.normal([])))
```

Note: Currently `tf.random.Generator` doesn't provide an option to let different replicas get identical (instead of different) streams (which is technically not hard). If you have a use case for this feature, please let the TensorFlow developers know.

If the generator is seeded (e.g. created by `Generator.from_seed`), the random numbers are determined by the seed, even though different replicas get different and uncorrelated numbers. One can think of a random number generated on a replica as a hash of the replica ID and a "primary" random number that is common to all replicas. Hence, the whole system is still deterministic.

`tf.random.Generator` can also be created inside `Strategy.run`:

In [ ]:

```
strat = tf.distribute.MirroredStrategy(devices=["cpu:0", "cpu:1"])
with strat.scope():
    def f():
        g = tf.random.Generator.from_seed(1)
        a = g.normal([])
        b = g.normal([])
        return tf.stack([a, b])
    print(strat.run(f))
    print(strat.run(f))
```

We no longer recommend passing `tf.random.Generator` as arguments to `Strategy.run`, because `Strategy.run` generally expects the arguments to be tensors, not generators.

## Saving generators

Generally for saving or serializing you can handle a `tf.random.Generator` the same way you would handle a `tf.Variable` or a `tf.Module` (or its subclasses). In TF there are two mechanisms for serialization: [Checkpoint](https://www.tensorflow.org/guide/checkpoint) (<https://www.tensorflow.org/guide/checkpoint>) and [SavedModel](https://www.tensorflow.org/guide/saved_model) ([https://www.tensorflow.org/guide/saved\\_model](https://www.tensorflow.org/guide/saved_model)).

### Checkpoint

Generators can be freely saved and restored using `tf.train.Checkpoint`. The random-number stream from the restoring point will be the same as that from the saving point.

In [ ]:

```
filename = "./checkpoint"
g = tf.random.Generator.from_seed(1)
cp = tf.train.Checkpoint(generator=g)
print(g.normal([]))
```

In [ ]:

```
cp.write(filename)
print("RNG stream from saving point:")
print(g.normal([]))
print(g.normal([]))
```

In [ ]:

```
cp.restore(filename)
print("RNG stream from restoring point:")
print(g.normal([]))
print(g.normal([]))
```

You can also save and restore within a distribution strategy:

In [ ]:

```
filename = "./checkpoint"
strat = tf.distribute.MirroredStrategy(devices=["cpu:0", "cpu:1"])
with strat.scope():
    g = tf.random.Generator.from_seed(1)
    cp = tf.train.Checkpoint(my_generator=g)
    print(strat.run(lambda: g.normal([])))
```

```
In [ ]:
```

```
with strat.scope():
    cp.write(filename)
    print("RNG stream from saving point:")
    print(strat.run(lambda: g.normal([])))
    print(strat.run(lambda: g.normal([])))
```

```
In [ ]:
```

```
with strat.scope():
    cp.restore(filename)
    print("RNG stream from restoring point:")
    print(strat.run(lambda: g.normal([])))
    print(strat.run(lambda: g.normal([])))
```

You should make sure that the replicas don't diverge in their RNG call history (e.g. one replica makes one RNG call while another makes two RNG calls) before saving. Otherwise, their internal RNG states will diverge and `tf.train.Checkpoint` (which only saves the first replica's state) won't properly restore all the replicas.

You can also restore a saved checkpoint to a different distribution strategy with a different number of replicas. Because a `tf.random.Generator` object created in a strategy can only be used in the same strategy, to restore to a different strategy, you have to create a new `tf.random.Generator` in the target strategy and a new `tf.train.Checkpoint` for it, as shown in this example:

```
In [ ]:
```

```
filename = "./checkpoint"
strat1 = tf.distribute.MirroredStrategy(devices=["cpu:0", "cpu:1"])
with strat1.scope():
    g1 = tf.random.Generator.from_seed(1)
    cp1 = tf.train.Checkpoint(my_generator=g1)
    print(strat1.run(lambda: g1.normal([])))
```

```
In [ ]:
```

```
with strat1.scope():
    cp1.write(filename)
    print("RNG stream from saving point:")
    print(strat1.run(lambda: g1.normal([])))
    print(strat1.run(lambda: g1.normal([])))
```

```
In [ ]:
```

```
strat2 = tf.distribute.MirroredStrategy(devices=["cpu:0", "cpu:1", "cpu:2"])
with strat2.scope():
    g2 = tf.random.Generator.from_seed(1)
    cp2 = tf.train.Checkpoint(my_generator=g2)
    cp2.restore(filename)
    print("RNG stream from restoring point:")
    print(strat2.run(lambda: g2.normal([])))
    print(strat2.run(lambda: g2.normal([])))
```

Although `g1` and `cp1` are different objects from `g2` and `cp2`, they are linked via the common checkpoint file `filename` and object name `my_generator`. Overlapping replicas between strategies (e.g. `cpu:0` and `cpu:1` above) will have their RNG streams properly restored like in previous examples. This guarantee doesn't cover the case when a generator is saved in a strategy scope and restored outside of any strategy scope or vice versa, because a device outside strategies is treated as different from any replica in a strategy.

## SavedModel

`tf.random.Generator` can be saved to a `SavedModel`. The generator can be created within a strategy scope. The saving can also happen within a strategy scope.

In [ ]:

```
filename = "./saved_model"

class MyModule(tf.Module):

    def __init__(self):
        super(MyModule, self).__init__()
        self.g = tf.random.Generator.from_seed(0)

    @tf.function
    def __call__(self):
        return self.g.normal([])

    @tf.function
    def state(self):
        return self.g.state

strat = tf.distribute.MirroredStrategy(devices=["cpu:0", "cpu:1"])
with strat.scope():
    m = MyModule()
    print(strat.run(m))
    print("state:", m.state())
```

In [ ]:

```
with strat.scope():
    tf.saved_model.save(m, filename)
    print("RNG stream from saving point:")
    print(strat.run(m))
    print("state:", m.state())
    print(strat.run(m))
    print("state:", m.state())
```

In [ ]:

```
imported = tf.saved_model.load(filename)
print("RNG stream from loading point:")
print("state:", imported.state())
print(imported())
print("state:", imported.state())
print(imported())
print("state:", imported.state())
```

Loading a SavedModel containing `tf.random.Generator` into a distribution strategy is not recommended because the replicas will all generate the same random-number stream (which is because replica ID is frozen in SavedModel's graph).

Loading a distributed `tf.random.Generator` (a generator created within a distribution strategy) into a non-strategy environment, like the above example, also has a caveat. The RNG state will be properly restored, but the random numbers generated will be different from the original generator in its strategy (again because a device outside strategies is treated as different from any replica in a strategy).

## Stateless RNGs

Usage of stateless RNGs is simple. Since they are just pure functions, there is no state or side effect involved.

In [ ]:

```
print(tf.random.stateless_normal(shape=[2, 3], seed=[1, 2]))
print(tf.random.stateless_normal(shape=[2, 3], seed=[1, 2]))
```

Every stateless RNG requires a `seed` argument, which needs to be an integer Tensor of shape `[2]`. The results of the op are fully determined by this seed.

The RNG algorithm used by stateless RNGs is device-dependent, meaning the same op running on a different device may produce different outputs.

## Algorithms

## General

Both the `tf.random.Generator` class and the `stateless` functions support the Philox algorithm (written as "philox" or `tf.random.Algorithm.PHILOX`) on all devices.

Different devices will generate the same integer numbers, if using the same algorithm and starting from the same state. They will also generate "almost the same" float-point numbers, though there may be small numerical discrepancies caused by the different ways the devices carry out the float-point computation (e.g. reduction order).

## XLA devices

On XLA-driven devices (such as TPU, and also CPU/GPU when XLA is enabled) the ThreeFry algorithm (written as "threefry" or `tf.random.Algorithm.THREEFRY`) is also supported. This algorithm is fast on TPU but slow on CPU/GPU compared to Philox.

See paper '[Parallel Random Numbers: As Easy as 1, 2, 3](https://www.thesalmons.org/john/random123/papers/random123sc11.pdf)' (<https://www.thesalmons.org/john/random123/papers/random123sc11.pdf>) for more details about these algorithms.

**Copyright 2020 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Working with sparse tensors



[View on TensorFlow.org](https://www.tensorflow.org/guide/sparse_tensor)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/sparse_tensor.ipynb)

When working with tensors that contain a lot of zero values, it is important to store them in a space- and time-efficient manner. Sparse tensors enable efficient storage and processing of tensors that contain a lot of zero values. Sparse tensors are used extensively in encoding schemes like [TF-IDF](#) (<https://en.wikipedia.org/wiki/Tf%E2%80%93idf>) as part of data pre-processing in NLP applications and for pre-processing images with a lot of dark pixels in computer vision applications.

## Sparse tensors in TensorFlow

TensorFlow represents sparse tensors through the `tf.SparseTensor` object. Currently, sparse tensors in TensorFlow are encoded using the coordinate list (COO) format. This encoding format is optimized for hyper-sparse matrices such as embeddings.

The COO encoding for sparse tensors is comprised of:

- `values` : A 1D tensor with shape `[N]` containing all nonzero values.
- `indices` : A 2D tensor with shape `[N, rank]`, containing the indices of the nonzero values.
- `dense_shape` : A 1D tensor with shape `[rank]`, specifying the shape of the tensor.

A **nonzero** value in the context of a `tf.SparseTensor` is a value that's not explicitly encoded. It is possible to explicitly include zero values in the `values` of a COO sparse matrix, but these "explicit zeros" are generally not included when referring to nonzero values in a sparse tensor.

Note: `tf.SparseTensor` does not require that indices/values be in any particular order, but several ops assume that they're in row-major order. Use `tf.sparse.reorder` to create a copy of the sparse tensor that is sorted in the canonical row-major order.

## Creating a `tf.SparseTensor`

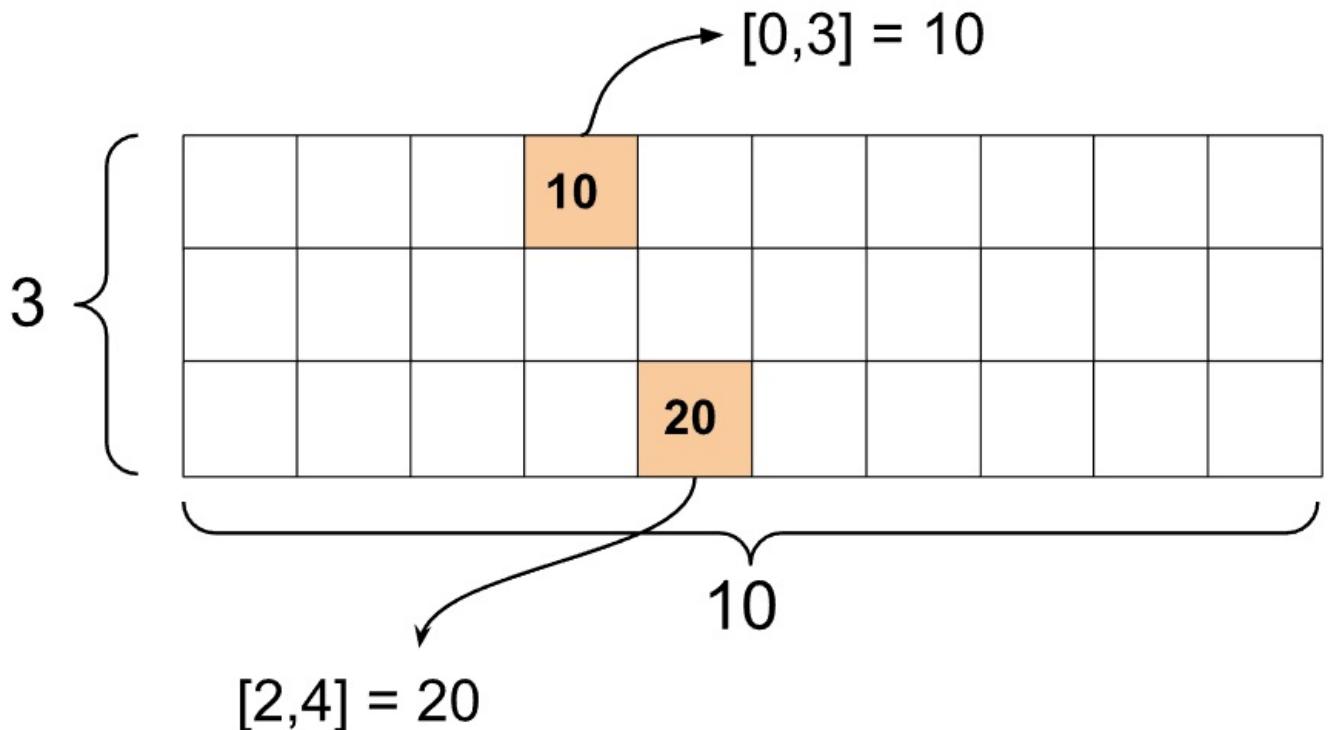
Construct sparse tensors by directly specifying their `values`, `indices`, and `dense_shape`.

```
In [ ]:
```

```
import tensorflow as tf
```

```
In [ ]:
```

```
st1 = tf.SparseTensor(indices=[[0, 3], [2, 4]],  
                      values=[10, 20],  
                      dense_shape=[3, 10])
```



When you use the `print()` function to print a sparse tensor, it shows the contents of the three component tensors:

```
In [ ]:
```

```
print(st1)
```

It is easier to understand the contents of a sparse tensor if the nonzero `values` are aligned with their corresponding `indices`. Define a helper function to pretty-print sparse tensors such that each nonzero value is shown on its own line.

```
In [ ]:
```

```
def pprint_sparse_tensor(st):  
    s = "<SparseTensor shape=%s \n values={" % (st.dense_shape.numpy().tolist(),)  
    for (index, value) in zip(st.indices, st.values):  
        s += f"\n    %s: %s" % (index.numpy().tolist(), value.numpy().tolist())  
    return s + "}>"
```

```
In [ ]:
```

```
print(pprint_sparse_tensor(st1))
```

You can also construct sparse tensors from dense tensors by using `tf.sparse.from_dense`, and convert them back to dense tensors by using `tf.sparse.to_dense`.

```
In [ ]:
```

```
st2 = tf.sparse.from_dense([[1, 0, 0, 8], [0, 0, 0, 0], [0, 0, 3, 0]])  
print(pprint_sparse_tensor(st2))
```

```
In [ ]:
```

```
st3 = tf.sparse.to_dense(st2)  
print(st3)
```

## Manipulating sparse tensors

Use the utilities in the `tf.sparse` package to manipulate sparse tensors. Ops like `tf.math.add` that you can use for arithmetic manipulation of dense tensors do not work with sparse tensors.

Add sparse tensors of the same shape by using `tf.sparse.add`.

In [ ]:

```
st_a = tf.SparseTensor(indices=[[0, 2], [3, 4]],
                       values=[31, 2],
                       dense_shape=[4, 10])

st_b = tf.SparseTensor(indices=[[0, 2], [7, 0]],
                       values=[56, 38],
                       dense_shape=[4, 10])

st_sum = tf.sparse.add(st_a, st_b)

print(pprint_sparse_tensor(st_sum))
```

Use `tf.sparse.sparse_dense_matmul` to multiply sparse tensors with dense matrices.

In [ ]:

```
st_c = tf.SparseTensor(indices=[[0, 1], [1, 0], [1, 1]],
                       values=[13, 15, 17],
                       dense_shape=(2,2))

mb = tf.constant([[4], [6]])
product = tf.sparse.sparse_dense_matmul(st_c, mb)

print(product)
```

Put sparse tensors together by using `tf.sparse.concat` and take them apart by using `tf.sparse.slice`.

In [ ]:

```
sparse_pattern_A = tf.SparseTensor(indices = [[2,4], [3,3], [3,4], [4,3], [4,4], [5,4]],
                                     values = [1,1,1,1,1,1],
                                     dense_shape = [8,5])
sparse_pattern_B = tf.SparseTensor(indices = [[0,2], [1,1], [1,3], [2,0], [2,4], [2,5], [3,5],
   [4,5], [5,0], [5,4], [5,5], [6,1], [6,3], [7,2]],
                                     values = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
                                     dense_shape = [8,6])
sparse_pattern_C = tf.SparseTensor(indices = [[3,0], [4,0]],
                                     values = [1,1],
                                     dense_shape = [8,6])

sparse_patterns_list = [sparse_pattern_A, sparse_pattern_B, sparse_pattern_C]
sparse_pattern = tf.sparse.concat(axis=1, sp_inputs=sparse_patterns_list)
print(tf.sparse.to_dense(sparse_pattern))
```

In [ ]:

```
sparse_slice_A = tf.sparse.slice(sparse_pattern_A, start = [0,0], size = [8,5])
sparse_slice_B = tf.sparse.slice(sparse_pattern_B, start = [0,5], size = [8,6])
sparse_slice_C = tf.sparse.slice(sparse_pattern_C, start = [0,10], size = [8,6])
print(tf.sparse.to_dense(sparse_slice_A))
print(tf.sparse.to_dense(sparse_slice_B))
print(tf.sparse.to_dense(sparse_slice_C))
```

If you're using TensorFlow 2.4 or above, use `tf.sparse.map_values` for elementwise operations on nonzero values in sparse tensors.

In [ ]:

```
st2_plus_5 = tf.sparse.map_values(tf.add, st2, 5)
print(tf.sparse.to_dense(st2_plus_5))
```

Note that only the nonzero values were modified – the zero values stay zero.

Equivalently, you can follow the design pattern below for earlier versions of TensorFlow:

In [ ]:

```
st2_plus_5 = tf.SparseTensor(  
    st2.indices,  
    st2.values + 5,  
    st2.dense_shape)  
print(tf.sparse.to_dense(st2_plus_5))
```

## Using `tf.SparseTensor` with other TensorFlow APIs

Sparse tensors work transparently with these TensorFlow APIs:

- `tf.keras`
- `tf.data`
- `tf.Train.Example` protobuf
- `tf.function`
- `tf.while_loop`
- `tf.cond`
- `tf.identity`
- `tf.cast`
- `tf.print`
- `tf.saved_model`
- `tf.io.serialize_sparse`
- `tf.io.serialize_many_sparse`
- `tf.io.deserialize_many_sparse`
- `tf.math.abs`
- `tf.math.negative`
- `tf.math.sign`
- `tf.math.square`
- `tf.math.sqrt`
- `tf.math.erf`
- `tf.math.tanh`
- `tf.math.bessel_i0e`
- `tf.math.bessel_i1e`

Examples are shown below for a few of the above APIs.

### `tf.keras`

A subset of the `tf.keras` API supports sparse tensors without expensive casting or conversion ops. The Keras API lets you pass sparse tensors as inputs to a Keras model. Set `sparse=True` when calling `tf.keras.Input` or `tf.keras.layers.InputLayer`. You can pass sparse tensors between Keras layers, and also have Keras models return them as outputs. If you use sparse tensors in `tf.keras.layers.Dense` layers in your model, they will output dense tensors.

The example below shows you how to pass a sparse tensor as an input to a Keras model if you use only layers that support sparse inputs.

In [ ]:

```
x = tf.keras.Input(shape=(4,), sparse=True)  
y = tf.keras.layers.Dense(4)(x)  
model = tf.keras.Model(x, y)  
  
sparse_data = tf.SparseTensor(  
    indices = [(0,0),(0,1),(0,2),  
              (4,3),(5,0),(5,1)],  
    values = [1,1,1,1,1,1],  
    dense_shape = (6,4)  
)  
  
model(sparse_data)  
model.predict(sparse_data)
```

## tf.data

The `tf.data` API enables you to build complex input pipelines from simple, reusable pieces. Its core data structure is `tf.data.Dataset`, which represents a sequence of elements in which each element consists of one or more components.

### Building datasets with sparse tensors

Build datasets from sparse tensors using the same methods that are used to build them from `tf.Tensor`s or NumPy arrays, such as `tf.data.Dataset.from_tensor_slices`. This op preserves the sparsity (or sparse nature) of the data.

In [ ]:

```
dataset = tf.data.Dataset.from_tensor_slices(sparse_data)
for element in dataset:
    print(pprint_sparse_tensor(element))
```

### Batching and unbatching datasets with sparse tensors

You can batch (combine consecutive elements into a single element) and unbatch datasets with sparse tensors using the `Dataset.batch` and `Dataset.unbatch` methods respectively.

In [ ]:

```
batched_dataset = dataset.batch(2)
for element in batched_dataset:
    print (pprint_sparse_tensor(element))
```

In [ ]:

```
unbatched_dataset = batched_dataset.unbatch()
for element in unbatched_dataset:
    print (pprint_sparse_tensor(element))
```

You can also use `tf.data.experimental.dense_to_sparse_batch` to batch dataset elements of varying shapes into sparse tensors.

### Transforming Datasets with sparse tensors

Transform and create sparse tensors in Datasets using `Dataset.map`.

In [ ]:

```
transform_dataset = dataset.map(lambda x: x*2)
for i in transform_dataset:
    print(pprint_sparse_tensor(i))
```

## tf.train.Example

`tf.train.Example` is a standard protobuf encoding for TensorFlow data. When using sparse tensors with `tf.train.Example`, you can:

- Read variable-length data into a `tf.SparseTensor` using `tf.io.VarLenFeature`. However, you should consider using `tf.io.RaggedFeature` instead.
- Read arbitrary sparse data into a `tf.SparseTensor` using `tf.io.SparseFeature`, which uses three separate feature keys to store the `indices`, `values`, and `dense_shape`.

## tf.function

The `tf.function` decorator precomputes TensorFlow graphs for Python functions, which can substantially improve the performance of your TensorFlow code. Sparse tensors work transparently with both `tf.function` and [concrete functions](#) ([https://www.tensorflow.org/guide/function#obtaining\\_concrete\\_functions](https://www.tensorflow.org/guide/function#obtaining_concrete_functions)).

In [ ]:

```
@tf.function
def f(x,y):
    return tf.sparse.sparse_dense_matmul(x,y)

a = tf.SparseTensor(indices=[[0, 3], [2, 4]],
                    values=[15, 25],
                    dense_shape=[3, 10])

b = tf.sparse.to_dense(tf.sparse.transpose(a))

c = f(a,b)

print(c)
```

## Distinguishing missing values from zero values

Most ops on `tf.SparseTensor`s treat missing values and explicit zero values identically. This is by design — a `tf.SparseTensor` is supposed to act just like a dense tensor.

However, there are a few cases where it can be useful to distinguish zero values from missing values. In particular, this allows for one way to encode missing/unknown data in your training data. For example, consider a use case where you have a tensor of scores (that can have any floating point value from `-Inf` to `+Inf`), with some missing scores. You can encode this tensor using a sparse tensor where the explicit zeros are known zero scores but the implicit zero values actually represent missing data and not zero.

Note: This is generally not the intended usage of `tf.SparseTensor`s; and you might want to also consider other techniques for encoding this such as for example using a separate mask tensor that identifies the locations of known/unknown values. However, exercise caution while using this approach, since most sparse operations will treat explicit and implicit zero values identically.

Note that some ops like `tf.sparse.reduce_max` do not treat missing values as if they were zero. For example, when you run the code block below, the expected output is `0`. However, because of this exception, the output is `-3`.

In [ ]:

```
print(tf.sparse.reduce_max(tf.sparse.from_dense([-5, 0, -3])))
```

In contrast, when you apply `tf.math.reduce_max` to a dense tensor, the output is 0 as expected.

In [ ]:

```
print(tf.math.reduce_max([-5, 0, -3]))
```

## Further reading and resources

- Refer to the [tensor guide](https://www.tensorflow.org/guide/tensor) (<https://www.tensorflow.org/guide/tensor>) to learn about tensors.
- Read the [ragged tensor guide](https://www.tensorflow.org/guide/ragged_tensor) ([https://www.tensorflow.org/guide/ragged\\_tensor](https://www.tensorflow.org/guide/ragged_tensor)) to learn how to work with ragged tensors, a type of tensor that lets you work with non-uniform data.
- Check out this object detection model in the [TensorFlow Model Garden](https://github.com/tensorflow/models) (<https://github.com/tensorflow/models>) that uses sparse tensors in a `tf.Example` [data decoder](#) ([https://github.com/tensorflow/models/blob/9139a7b90112562aec1d7e328593681bd410e1e7/research/object\\_detection/data\\_decoders/tf\\_example\\_decoder.py](https://github.com/tensorflow/models/blob/9139a7b90112562aec1d7e328593681bd410e1e7/research/object_detection/data_decoders/tf_example_decoder.py))

Copyright 2020 The TensorFlow Authors.

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Basic training loops



[View on TensorFlow.org](https://www.tensorflow.org/guide/basic_training_loops)

([https://www.tensorflow.org/guide/basic\\_training\\_loops](https://www.tensorflow.org/guide/basic_training_loops))



[Run in Google Colab](#)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/basic\\_training\\_loops.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/basic_training_loops.ipynb))

([https://github.com/tensorflow/docs/blob/master/site/en/guide/basic\\_training\\_loops.ipynb](https://github.com/tensorflow/docs/blob/master/site/en/guide/basic_training_loops.ipynb))

In the previous guides, you have learned about [tensors](#) ([./tensor.ipynb](#)), [variables](#) ([./variable.ipynb](#)), [gradient tape](#) ([autodiff.ipynb](#)), and [modules](#) ([./intro\\_to\\_modules.ipynb](#)). In this guide, you will fit these all together to train models.

TensorFlow also includes the [tf.Keras API](#) (<https://www.tensorflow.org/guide/keras/overview>), a high-level neural network API that provides useful abstractions to reduce boilerplate. However, in this guide, you will use basic classes.

## Setup

In [ ]:

```
import tensorflow as tf
import matplotlib.pyplot as plt
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
```

## Solving machine learning problems

Solving a machine learning problem usually consists of the following steps:

- Obtain training data.
- Define the model.
- Define a loss function.
- Run through the training data, calculating loss from the ideal value
- Calculate gradients for that loss and use an *optimizer* to adjust the variables to fit the data.
- Evaluate your results.

For illustration purposes, in this guide you'll develop a simple linear model,  $f(x) = x * W + b$ , which has two variables:  $W$  (weights) and  $b$  (bias).

This is the most basic of machine learning problems: Given  $x$  and  $y$ , try to find the slope and offset of a line via [simple linear regression](#) ([https://en.wikipedia.org/wiki/Linear\\_regression#Simple\\_and\\_multiple\\_linear\\_regression](https://en.wikipedia.org/wiki/Linear_regression#Simple_and_multiple_linear_regression)).

## Data

Supervised learning uses *inputs* (usually denoted as  $x$ ) and *outputs* (denoted  $y$ , often called *labels*). The goal is to learn from paired inputs and outputs so that you can predict the value of an output from an input.

Each input of your data, in TensorFlow, is almost always represented by a tensor, and is often a vector. In supervised training, the output (or value you'd like to predict) is also a tensor.

Here is some data synthesized by adding Gaussian (Normal) noise to points along a line.

In [ ]:

```
# The actual line
TRUE_W = 3.0
TRUE_B = 2.0

NUM_EXAMPLES = 201

# A vector of random x values
x = tf.linspace(-2, 2, NUM_EXAMPLES)
x = tf.cast(x, tf.float32)

def f(x):
    return x * TRUE_W + TRUE_B

# Generate some noise
noise = tf.random.normal(shape=[NUM_EXAMPLES])

# Calculate y
y = f(x) + noise
```

In [ ]:

```
# Plot all the data
plt.plot(x, y, '.')
plt.show()
```

Tensors are usually gathered together in *batches*, or groups of inputs and outputs stacked together. Batching can confer some training benefits and works well with accelerators and vectorized computation. Given how small this dataset is, you can treat the entire dataset as a single batch.

## Define the model

Use `tf.Variable` to represent all weights in a model. A `tf.Variable` stores a value and provides this in tensor form as needed. See the [variable guide \(./variable.ipynb\)](#) for more details.

Use `tf.Module` to encapsulate the variables and the computation. You could use any Python object, but this way it can be easily saved.

Here, you define both  $w$  and  $b$  as variables.

In [ ]:

```
class MyModel(tf.Module):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # Initialize the weights to `5.0` and the bias to `0.0`
        # In practice, these should be randomly initialized
        self.w = tf.Variable(5.0)
        self.b = tf.Variable(0.0)

    def __call__(self, x):
        return self.w * x + self.b

model = MyModel()

# List the variables tf.modules's built-in variable aggregation.
print("Variables:", model.variables)

# Verify the model works
assert model(3.0).numpy() == 15.0
```

The initial variables are set here in a fixed way, but Keras comes with any of a number of [initializers](#) ([https://www.tensorflow.org/api\\_docs/python/tf/keras/initializers](https://www.tensorflow.org/api_docs/python/tf/keras/initializers)) you could use, with or without the rest of Keras.

## Define a loss function

A loss function measures how well the output of a model for a given input matches the target output. The goal is to minimize this difference during training. Define the standard L2 loss, also known as the "mean squared" error:

In [ ]:

```
# This computes a single loss value for an entire batch
def loss(target_y, predicted_y):
    return tf.reduce_mean(tf.square(target_y - predicted_y))
```

Before training the model, you can visualize the loss value by plotting the model's predictions in red and the training data in blue:

In [ ]:

```
plt.plot(x, y, '.', label="Data")
plt.plot(x, f(x), label="Ground truth")
plt.plot(x, model(x), label="Predictions")
plt.legend()
plt.show()

print("Current loss: %1.6f" % loss(y, model(x)).numpy())
```

## Define a training loop

The training loop consists of repeatedly doing three tasks in order:

- Sending a batch of inputs through the model to generate outputs
- Calculating the loss by comparing the outputs to the output (or label)
- Using gradient tape to find the gradients
- Optimizing the variables with those gradients

For this example, you can train the model using [gradient descent](#) ([https://en.wikipedia.org/wiki/Gradient\\_descent](https://en.wikipedia.org/wiki/Gradient_descent)).

There are many variants of the gradient descent scheme that are captured in `tf.keras.optimizers`. But in the spirit of building from first principles, here you will implement the basic math yourself with the help of `tf.GradientTape` for automatic differentiation and `tf.assign_sub` for decrementing a value (which combines `tf.assign` and `tf.sub`):

In [ ]:

```
# Given a callable model, inputs, outputs, and a learning rate...
def train(model, x, y, learning_rate):

    with tf.GradientTape() as t:
        # Trainable variables are automatically tracked by GradientTape
        current_loss = loss(y, model(x))

    # Use GradientTape to calculate the gradients with respect to W and b
    dw, db = t.gradient(current_loss, [model.w, model.b])

    # Subtract the gradient scaled by the learning rate
    model.w.assign_sub(learning_rate * dw)
    model.b.assign_sub(learning_rate * db)
```

For a look at training, you can send the same batch of  $x$  and  $y$  through the training loop, and see how  $W$  and  $b$  evolve.

In [ ]:

```
model = MyModel()

# Collect the history of W-values and b-values to plot later
weights = []
biases = []
epochs = range(10)

# Define a training loop
def report(model, loss):
    return f"W = {model.w.numpy():1.2f}, b = {model.b.numpy():1.2f}, loss={loss:2.5f}"

def training_loop(model, x, y):

    for epoch in epochs:
        # Update the model with the single giant batch
        train(model, x, y, learning_rate=0.1)

        # Track this before I update
        weights.append(model.w.numpy())
        biases.append(model.b.numpy())
        current_loss = loss(y, model(x))

        print(f"Epoch {epoch:2d}:")
        print("    ", report(model, current_loss))
```

Do the training

In [ ]:

```
current_loss = loss(y, model(x))

print(f"Starting:")
print("    ", report(model, current_loss))

training_loop(model, x, y)
```

Plot the evolution of the weights over time:

In [ ]:

```
plt.plot(epochs, weights, label='Weights', color=colors[0])
plt.plot(epochs, [TRUE_W] * len(epochs), '--',
         label="True weight", color=colors[0])

plt.plot(epochs, biases, label='bias', color=colors[1])
plt.plot(epochs, [TRUE_B] * len(epochs), '--',
         label="True bias", color=colors[1])

plt.legend()
plt.show()
```

Visualize how the trained model performs

In [ ]:

```
plt.plot(x, y, '.', label="Data")
plt.plot(x, f(x), label="Ground truth")
plt.plot(x, model(x), label="Predictions")
plt.legend()
plt.show()

print("Current loss: %1.6f" % loss(model(x), y).numpy())
```

## The same solution, but with Keras

It's useful to contrast the code above with the equivalent in Keras.

Defining the model looks exactly the same if you subclass `tf.keras.Model`. Remember that Keras models inherit ultimately from module.

In [ ]:

```
class MyModelKeras(tf.keras.Model):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # Initialize the weights to `5.0` and the bias to `0.0`
        # In practice, these should be randomly initialized
        self.w = tf.Variable(5.0)
        self.b = tf.Variable(0.0)

    def call(self, x):
        return self.w * x + self.b

keras_model = MyModelKeras()

# Reuse the training loop with a Keras model
training_loop(keras_model, x, y)

# You can also save a checkpoint using Keras's built-in support
keras_model.save_weights("my_checkpoint")
```

Rather than write new training loops each time you create a model, you can use the built-in features of Keras as a shortcut. This can be useful when you do not want to write or debug Python training loops.

If you do, you will need to use `model.compile()` to set the parameters, and `model.fit()` to train. It can be less code to use Keras implementations of L2 loss and gradient descent, again as a shortcut. Keras losses and optimizers can be used outside of these convenience functions, too, and the previous example could have used them.

In [ ]:

```
keras_model = MyModelKeras()

# compile sets the training parameters
keras_model.compile(
    # By default, fit() uses tf.function(). You can
    # turn that off for debugging, but it is on now.
    run_eagerly=False,

    # Using a built-in optimizer, configuring as an object
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.1),

    # Keras comes with built-in MSE error
    # However, you could use the loss function
    # defined above
    loss=tf.keras.losses.mean_squared_error,
)
```

Keras `fit` expects batched data or a complete dataset as a NumPy array. NumPy arrays are chopped into batches and default to a batch size of 32.

In this case, to match the behavior of the hand-written loop, you should pass `x` in as a single batch of size 1000.

In [ ]:

```
print(x.shape[0])
keras_model.fit(x, y, epochs=10, batch_size=1000)
```

Note that Keras prints out the loss after training, not before, so the first loss appears lower, but otherwise this shows essentially the same training performance.

## Next steps

In this guide, you have seen how to use the core classes of tensors, variables, modules, and gradient tape to build and train a model, and further how those ideas map to Keras.

This is, however, an extremely simple problem. For a more practical introduction, see [Custom training walkthrough](#) ([./tutorials/customization/custom\\_training\\_walkthrough.ipynb](#)).

For more on using built-in Keras training loops, see [this guide](#) ([https://www.tensorflow.org/guide/keras/train\\_and\\_evaluate](https://www.tensorflow.org/guide/keras/train_and_evaluate)). For more on training loops and Keras, see [this guide](#) ([https://www.tensorflow.org/guide/keras/writing\\_a\\_training\\_loop\\_from\\_scratch](https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch)). For writing custom distributed training loops, see [this guide](#) ([distributed\\_training.ipynb#using\\_tfdistributestrategy\\_with\\_basic\\_training\\_loops\\_loops](#)).

**Copyright 2020 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## NumPy API on TensorFlow



[View on TensorFlow.org](#)

([https://www.tensorflow.org/guide/tf\\_numpy](https://www.tensorflow.org/guide/tf_numpy))



[Run in Google Colab](#)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/tf\\_numpy.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/tf_numpy.ipynb))

([https://github.com/tensorflow/docs/blob/master/site/en/guide/tf\\_numpy.ipynb](https://github.com/tensorflow/docs/blob/master/site/en/guide/tf_numpy.ipynb))

### Overview

TensorFlow implements a subset of the [NumPy API](#) (<https://numpy.org/doc/1.16>), available as `tf.experimental.numpy`. This allows running NumPy code, accelerated by TensorFlow, while also allowing access to all of TensorFlow's APIs.

### Setup

In [ ]:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import tensorflow.experimental.numpy as tnp
import timeit

print("Using TensorFlow version %s" % tf.__version__)
```

### Enabling NumPy behavior

In order to use `tnp` as NumPy, enable NumPy behavior for TensorFlow:

In [ ]:

```
tnp.experimental_enable_numpy_behavior()
```

This call enables type promotion in TensorFlow and also changes type inference, when converting literals to tensors, to more strictly follow the NumPy standard.

Note: This call will change the behavior of entire TensorFlow, not just the `tf.experimental.numpy` module.

## TensorFlow NumPy ND array

An instance of `tf.experimental.numpy.ndarray`, called **ND Array**, represents a multidimensional dense array of a given `dtype` placed on a certain device. It is an alias to `tf.Tensor`. Check out the ND array class for useful methods like `ndarray.T`, `ndarray.reshape`, `ndarray.ravel` and others.

First create an ND array object, and then invoke different methods.

In [ ]:

```
# Create an ND array and check out different attributes.
ones = tnp.ones([5, 3], dtype=tnp.float32)
print("Created ND array with shape = %s, rank = %s, "
      "dtype = %s on device = %s\n" %
      (ones.shape, ones.ndim, ones.dtype, ones.device))

# `ndarray` is just an alias to `tf.Tensor`.
print("Is `ones` an instance of tf.Tensor: %s\n" % isinstance(ones, tf.Tensor))

# Try commonly used member functions.
print("ndarray.T has shape %s" % str(ones.T.shape))
print("ndarray.reshape(-1) has shape %s" % ones.reshape(-1).shape)
```

## Type promotion

TensorFlow NumPy APIs have well-defined semantics for converting literals to ND array, as well as for performing type promotion on ND array inputs. Please see `np.result_type` ([https://numpy.org/doc/1.16/reference/generated/numpy.result\\_type.html](https://numpy.org/doc/1.16/reference/generated/numpy.result_type.html)) for more details.

TensorFlow APIs leave `tf.Tensor` inputs unchanged and do not perform type promotion on them, while TensorFlow NumPy APIs promote all inputs according to NumPy type promotion rules. In the next example, you will perform type promotion. First, run addition on ND array inputs of different types and note the output types. None of these type promotions would be allowed by TensorFlow APIs.

In [ ]:

```
print("Type promotion for operations")
values = [tnp.asarray(1, dtype=d) for d in
          (tnp.int32, tnp.int64, tnp.float32, tnp.float64)]
for i, v1 in enumerate(values):
    for v2 in values[i + 1:]:
        print("%s + %s => %s" %
              (v1.dtype.name, v2.dtype.name, (v1 + v2).dtype.name))
```

Finally, convert literals to ND array using `ndarray.asarray` and note the resulting type.

In [ ]:

```
print("Type inference during array creation")
print("tnp.asarray(1).dtype == tnp.%s" % tnp.asarray(1).dtype.name)
print("tnp.asarray(1.).dtype == tnp.%s\n" % tnp.asarray(1.).dtype.name)
```

When converting literals to ND array, NumPy prefers wide types like `tnp.int64` and `tnp.float64`. In contrast, `tf.convert_to_tensor` prefers `tf.int32` and `tf.float32` types for converting constants to `tf.Tensor`. TensorFlow NumPy APIs adhere to the NumPy behavior for integers. As for floats, the `prefer_float32` argument of `experimental_enable_numpy_behavior` lets you control whether to prefer `tf.float32` over `tf.float64` (default to `False`). For example:

In [ ]:

```
tnp.experimental_enable_numpy_behavior(prefer_float32=True)
print("When prefer_float32 is True:")
print("tnp.asarray(1.).dtype == tnp.%s" % tnp.asarray(1.).dtype.name)
print("tnp.add(1., 2.).dtype == tnp.%s" % tnp.add(1., 2.).dtype.name)

tnp.experimental_enable_numpy_behavior(prefer_float32=False)
print("When prefer_float32 is False:")
print("tnp.asarray(1.).dtype == tnp.%s" % tnp.asarray(1.).dtype.name)
print("tnp.add(1., 2.).dtype == tnp.%s" % tnp.add(1., 2.).dtype.name)
```

## Broadcasting

Similar to TensorFlow, NumPy defines rich semantics for "broadcasting" values. You can check out the [NumPy broadcasting guide](#) (<https://numpy.org/doc/1.16/user/basics.broadcasting.html>) for more information and compare this with [TensorFlow broadcasting semantics](#) (<https://www.tensorflow.org/guide/tensor#broadcasting>).

In [ ]:

```
x = tnp.ones([2, 3])
y = tnp.ones([3])
z = tnp.ones([1, 2, 1])
print("Broadcasting shapes %s, %s and %s gives shape %s" % (
    x.shape, y.shape, z.shape, (x + y + z).shape))
```

## Indexing

NumPy defines very sophisticated indexing rules. See the [NumPy Indexing guide](https://numpy.org/doc/1.16/reference/arrays.indexing.html) (<https://numpy.org/doc/1.16/reference/arrays.indexing.html>). Note the use of ND arrays as indices below.

In [ ]:

```
x = tnp.arange(24).reshape(2, 3, 4)

print("Basic indexing")
print(x[1, tnp.newaxis, 1:3, ...], "\n")

print("Boolean indexing")
print(x[:, (True, False, True)], "\n")

print("Advanced indexing")
print(x[1, (0, 0, 1), tnp.asarray([0, 1, 1])])
```

In [ ]:

```
# Mutation is currently not supported
try:
    tnp.arange(6)[1] = -1
except TypeError:
    print("Currently, TensorFlow NumPy does not support mutation.")
```

## Example Model

Next, you can see how to create a model and run inference on it. This simple model applies a relu layer followed by a linear projection. Later sections will show how to compute gradients for this model using TensorFlow's `GradientTape`.

In [ ]:

```
class Model(object):
    """Model with a dense and a linear layer."""

    def __init__(self):
        self.weights = None

    def predict(self, inputs):
        if self.weights is None:
            size = inputs.shape[1]
            # Note that type `tnp.float32` is used for performance.
            stddev = tnp.sqrt(size).astype(tnp.float32)
            w1 = tnp.random.randn(size, 64).astype(tnp.float32) / stddev
            bias = tnp.random.randn(64).astype(tnp.float32)
            w2 = tnp.random.randn(64, 2).astype(tnp.float32) / 8
            self.weights = (w1, bias, w2)
        else:
            w1, bias, w2 = self.weights
            y = tnp.matmul(inputs, w1) + bias
            y = tnp.maximum(y, 0) # Relu
            return tnp.matmul(y, w2) # Linear projection

model = Model()
# Create input data and compute predictions.
print(model.predict(tnp.ones([2, 32], dtype=tnp.float32)))
```

## TensorFlow NumPy and NumPy

TensorFlow NumPy implements a subset of the full NumPy spec. While more symbols will be added over time, there are systematic features that will not be supported in the near future. These include NumPy C API support, Swig integration, Fortran storage order, views and `stride_tricks`, and some `dtype`s (like `np.recarray` and `np.object`). For more details, please see the [TensorFlow NumPy API Documentation](https://www.tensorflow.org/api_docs/python/tf/experimental/numpy) ([https://www.tensorflow.org/api\\_docs/python/tf/experimental/numpy](https://www.tensorflow.org/api_docs/python/tf/experimental/numpy)).

## NumPy interoperability

TensorFlow ND arrays can interoperate with NumPy functions. These objects implement the `__array__` interface. NumPy uses this interface to convert function arguments to `np.ndarray` values before processing them.

Similarly, TensorFlow NumPy functions can accept inputs of different types including `np.ndarray`. These inputs are converted to an ND array by calling `ndarray.asarray` on them.

Conversion of the ND array to and from `np.ndarray` may trigger actual data copies. Please see the section on [buffer copies](#) for more details.

In [ ]:

```
# ND array passed into NumPy function.  
np_sum = np.sum(tnp.ones([2, 3]))  
print("sum = %s. Class: %s" % (float(np_sum), np_sum.__class__))  
  
# `np.ndarray` passed into TensorFlow NumPy function.  
tnp_sum = tnp.sum(np.ones([2, 3]))  
print("sum = %s. Class: %s" % (float(tnp_sum), tnp_sum.__class__))
```

In [ ]:

```
# It is easy to plot ND arrays, given the __array__ interface.  
labels = 15 + 2 * tnp.random.randn(1, 1000)  
_ = plt.hist(labels)
```

## Buffer copies

Intermixing TensorFlow NumPy with NumPy code may trigger data copies. This is because TensorFlow NumPy has stricter requirements on memory alignment than those of NumPy.

When a `np.ndarray` is passed to TensorFlow NumPy, it will check for alignment requirements and trigger a copy if needed. When passing an ND array CPU buffer to NumPy, generally the buffer will satisfy alignment requirements and NumPy will not need to create a copy.

ND arrays can refer to buffers placed on devices other than the local CPU memory. In such cases, invoking a NumPy function will trigger copies across the network or device as needed.

Given this, intermixing with NumPy API calls should generally be done with caution and the user should watch out for overheads of copying data. Interleaving TensorFlow NumPy calls with TensorFlow calls is generally safe and avoids copying data. See the section on [TensorFlow interoperability](#) for more details.

## Operator precedence

TensorFlow NumPy defines an `__array_priority__` higher than NumPy's. This means that for operators involving both ND array and `np.ndarray`, the former will take precedence, i.e., `np.ndarray` input will get converted to an ND array and the TensorFlow NumPy implementation of the operator will get invoked.

In [ ]:

```
x = tnp.ones([2]) + np.ones([2])  
print("x = %s\nclass = %s" % (x, x.__class__))
```

## TF NumPy and TensorFlow

TensorFlow NumPy is built on top of TensorFlow and hence interoperates seamlessly with TensorFlow.

### tf.Tensor and ND array

ND array is an alias to `tf.Tensor`, so obviously they can be intermixed without triggering actual data copies.

In [ ]:

```
x = tf.constant([1, 2])  
print(x)  
  
# `asarray` and `convert_to_tensor` here are no-ops.  
tnp_x = tnp.asarray(x)  
print(tnp_x)  
print(tf.convert_to_tensor(tnp_x))  
  
# Note that tf.Tensor.numpy() will continue to return `np.ndarray`.  
print(x.numpy(), x.numpy().__class__)
```

## TensorFlow interoperability

An ND array can be passed to TensorFlow APIs, since ND array is just an alias to `tf.Tensor`. As mentioned earlier, such interoperation does not do data copies, even for data placed on accelerators or remote devices.

Conversely, `tf.Tensor` objects can be passed to `tf.experimental.numpy` APIs, without performing data copies.

In [ ]:

```
# ND array passed into TensorFlow function.  
tf_sum = tf.reduce_sum(tpn.ones([2, 3], tnp.float32))  
print("Output = %s" % tf_sum)  
  
# `tf.Tensor` passed into TensorFlow NumPy function.  
tnp_sum = tnp.sum(tf.ones([2, 3]))  
print("Output = %s" % tnp_sum)
```

## Gradients and Jacobians: `tf.GradientTape`

TensorFlow's `GradientTape` can be used for backpropagation through TensorFlow and TensorFlow NumPy code.

Use the model created in [Example Model](#) section, and compute gradients and jacobians.

In [ ]:

```
def create_batch(batch_size=32):  
    """Creates a batch of input and labels."""  
    return (tnp.random.randn(batch_size, 32).astype(tnp.float32),  
           tnp.random.randn(batch_size, 2).astype(tnp.float32))  
  
def compute_gradients(model, inputs, labels):  
    """Computes gradients of squared loss between model prediction and labels."""  
    with tf.GradientTape() as tape:  
        assert model.weights is not None  
        # Note that `model.weights` need to be explicitly watched since they  
        # are not tf.Variables.  
        tape.watch(model.weights)  
        # Compute prediction and loss  
        prediction = model.predict(inputs)  
        loss = tnp.sum(tnp.square(prediction - labels))  
        # This call computes the gradient through the computation above.  
    return tape.gradient(loss, model.weights)  
  
inputs, labels = create_batch()  
gradients = compute_gradients(model, inputs, labels)  
  
# Inspect the shapes of returned gradients to verify they match the  
# parameter shapes.  
print("Parameter shapes:", [w.shape for w in model.weights])  
print("Gradient shapes:", [g.shape for g in gradients])  
# Verify that gradients are of type ND array.  
assert isinstance(gradients[0], tnp.ndarray)
```

In [ ]:

```
# Computes a batch of jacobians. Each row is the jacobian of an element in the  
# batch of outputs w.r.t. the corresponding input batch element.  
def prediction_batch_jacobian(inputs):  
    with tf.GradientTape() as tape:  
        tape.watch(inputs)  
        prediction = model.predict(inputs)  
    return prediction, tape.batch_jacobian(prediction, inputs)  
  
inp_batch = tnp.ones([16, 32], tnp.float32)  
output, batch_jacobian = prediction_batch_jacobian(inp_batch)  
# Note how the batch jacobian shape relates to the input and output shapes.  
print("Output shape: %s, input shape: %s" % (output.shape, inp_batch.shape))  
print("Batch jacobian shape:", batch_jacobian.shape)
```

## Trace compilation: `tf.function`

TensorFlow's `tf.function` works by "trace compiling" the code and then optimizing these traces for much faster performance. See the [Introduction to Graphs and Functions \(`./intro\_to\_graphs.ipynb`\)](#).

`tf.function` can be used to optimize TensorFlow NumPy code as well. Here is a simple example to demonstrate the speedups. Note that the body of `tf.function` code includes calls to TensorFlow NumPy APIs.

In [ ]:

```
inputs, labels = create_batch(512)
print("Eager performance")
compute_gradients(model, inputs, labels)
print(timeit.timeit(lambda: compute_gradients(model, inputs, labels),
                    number=10) * 100, "ms")

print("\n\ntf.function compiled performance")
compiled_compute_gradients = tf.function(compute_gradients)
compiled_compute_gradients(model, inputs, labels) # warmup
print(timeit.timeit(lambda: compiled_compute_gradients(model, inputs, labels),
                    number=10) * 100, "ms")
```

## Vectorization: `tf.vectorized_map`

TensorFlow has inbuilt support for vectorizing parallel loops, which allows speedups of one to two orders of magnitude. These speedups are accessible via the `tf.vectorized_map` API and apply to TensorFlow NumPy code as well.

It is sometimes useful to compute the gradient of each output in a batch w.r.t. the corresponding input batch element. Such computation can be done efficiently using `tf.vectorized_map` as shown below.

In [ ]:

```
@tf.function
def vectorized_per_example_gradients(inputs, labels):
    def single_example_gradient(arg):
        inp, label = arg
        return compute_gradients(model,
                               tnp.expand_dims(inp, 0),
                               tnp.expand_dims(label, 0))
    # Note that a call to `tf.vectorized_map` semantically maps
    # `single_example_gradient` over each row of `inputs` and `labels`.
    # The interface is similar to `tf.map_fn`.
    # The underlying machinery vectorizes away this map loop which gives
    # nice speedups.
    return tf.vectorized_map(single_example_gradient, (inputs, labels))

batch_size = 128
inputs, labels = create_batch(batch_size)

per_example_gradients = vectorized_per_example_gradients(inputs, labels)
for w, p in zip(model.weights, per_example_gradients):
    print("Weight shape: %s, batch size: %s, per example gradient shape: %s" % (
        w.shape, batch_size, p.shape))
```

In [ ]:

```
# Benchmark the vectorized computation above and compare with
# unvectorized sequential computation using `tf.map_fn`.
@tf.function
def unvectorized_per_example_gradients(inputs, labels):
    def single_example_gradient(arg):
        inp, label = arg
        return compute_gradients(model,
                               tnp.expand_dims(inp, 0),
                               tnp.expand_dims(label, 0))

    return tf.map_fn(single_example_gradient, (inputs, labels),
                    fn_output_signature=(tf.float32, tf.float32, tf.float32))

print("Running vectorized computation")
print(timeit.timeit(lambda: vectorized_per_example_gradients(inputs, labels),
                    number=10) * 100, "ms")

print("\nRunning unvectorized computation")
per_example_gradients = unvectorized_per_example_gradients(inputs, labels)
print(timeit.timeit(lambda: unvectorized_per_example_gradients(inputs, labels),
                    number=10) * 100, "ms")
```

## Device placement

TensorFlow NumPy can place operations on CPUs, GPUs, TPUs and remote devices. It uses standard TensorFlow mechanisms for device placement. Below a simple example shows how to list all devices and then place some computation on a particular device.

TensorFlow also has APIs for replicating computation across devices and performing collective reductions which will not be covered here.

## List devices

`tf.config.list_logical_devices` and `tf.config.list_physical_devices` can be used to find what devices to use.

In [ ]:

```
print("All logical devices:", tf.config.list_logical_devices())
print("All physical devices:", tf.config.list_physical_devices())

# Try to get the GPU device. If unavailable, fallback to CPU.
try:
    device = tf.config.list_logical_devices(device_type="GPU")[0]
except IndexError:
    device = "/device:CPU:0"
```

## Placing operations: `tf.device`

Operations can be placed on a device by calling it in a `tf.device` scope.

In [ ]:

```
print("Using device: %s" % str(device))
# Run operations in the `tf.device` scope.
# If a GPU is available, these operations execute on the GPU and outputs are
# placed on the GPU memory.
with tf.device(device):
    prediction = model.predict(create_batch(5)[0])

print("prediction is placed on %s" % prediction.device)
```

## Copying ND arrays across devices: `tnp.copy`

A call to `tnp.copy`, placed in a certain device scope, will copy the data to that device, unless the data is already on that device.

In [ ]:

```
with tf.device("/device:CPU:0"):
    prediction_cpu = tnp.copy(prediction)
print(prediction.device)
print(prediction_cpu.device)
```

## Performance comparisons

TensorFlow NumPy uses highly optimized TensorFlow kernels that can be dispatched on CPUs, GPUs and TPUs. TensorFlow also performs many compiler optimizations, like operation fusion, which translate to performance and memory improvements. See [TensorFlow graph optimization with Grappler](#) (`./graph_optimization.ipynb`) to learn more.

However TensorFlow has higher overheads for dispatching operations compared to NumPy. For workloads composed of small operations (less than about 10 microseconds), these overheads can dominate the runtime and NumPy could provide better performance. For other cases, TensorFlow should generally provide better performance.

Run the benchmark below to compare NumPy and TensorFlow NumPy performance for different input sizes.

In [ ]:

```
def benchmark(f, inputs, number=30, force_gpu_sync=False):
    """Utility to benchmark `f` on each value in `inputs`."""
    times = []
    for inp in inputs:
        def _g():
            if force_gpu_sync:
                one = tnp.asarray(1)
            f(inp)
            if force_gpu_sync:
                with tf.device("CPU:0"):
                    tnp.copy(one) # Force a sync for GPU case

        _g() # warmup
        t = timeit.timeit(_g, number=number)
        times.append(t * 1000. / number)
    return times

def plot(np_times, tnp_times, compiled_tnp_times, has_gpu, tnp_times_gpu):
    """Plot the different runtimes."""
    plt.xlabel("size")
    plt.ylabel("time (ms)")
    plt.title("Sigmoid benchmark: TF NumPy vs NumPy")
    plt.plot(sizes, np_times, label="NumPy")
    plt.plot(sizes, tnp_times, label="TF NumPy (CPU)")
    plt.plot(sizes, compiled_tnp_times, label="Compiled TF NumPy (CPU)")
    if has_gpu:
        plt.plot(sizes, tnp_times_gpu, label="TF NumPy (GPU)")
    plt.legend()
```

In [ ]:

```
# Define a simple implementation of `sigmoid`, and benchmark it using
# NumPy and TensorFlow NumPy for different input sizes.

def np_sigmoid(y):
    return 1. / (1. + np.exp(-y))

def tnp_sigmoid(y):
    return 1. / (1. + tnp.exp(-y))

@tf.function
def compiled_tnp_sigmoid(y):
    return tnp_sigmoid(y)

sizes = (2 ** 0, 2 ** 5, 2 ** 10, 2 ** 15, 2 ** 20)
np_inputs = [np.random.randn(size).astype(np.float32) for size in sizes]
np_times = benchmark(np_sigmoid, np_inputs)

with tf.device("/device:CPU:0"):
    tnp_inputs = [tnp.random.randn(size).astype(np.float32) for size in sizes]
    tnp_times = benchmark(tnp_sigmoid, tnp_inputs)
    compiled_tnp_times = benchmark(compiled_tnp_sigmoid, tnp_inputs)

has_gpu = len(tf.config.list_logical_devices("GPU"))
if has_gpu:
    with tf.device("/device:GPU:0"):
        tnp_inputs = [tnp.random.randn(size).astype(np.float32) for size in sizes]
        tnp_times_gpu = benchmark(compiled_tnp_sigmoid, tnp_inputs, 100, True)
else:
    tnp_times_gpu = None
plot(np_times, tnp_times, compiled_tnp_times, has_gpu, tnp_times_gpu)
```

## Further reading

- [TensorFlow NumPy: Distributed Image Classification Tutorial](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/numpy_ops/g3doc/TensorFlow_Numpy_Distributed_Image_Classification.ipynb)  
([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/numpy\\_ops/g3doc/TensorFlow\\_Numpy\\_Distributed\\_Image\\_Classification.ipynb](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/numpy_ops/g3doc/TensorFlow_Numpy_Distributed_Image_Classification.ipynb))
- [TensorFlow NumPy: Keras and Distribution Strategy](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/numpy_ops/g3doc/TensorFlow_NumPy_Keras_and_Distribution_Strategy.ipynb)  
([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/numpy\\_ops/g3doc/TensorFlow\\_NumPy\\_Keras\\_and\\_Distribution\\_Strategy.ipynb](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/numpy_ops/g3doc/TensorFlow_NumPy_Keras_and_Distribution_Strategy.ipynb))
- [Sentiment Analysis with Trax and TensorFlow NumPy](https://github.com/google/trax/blob/master/trax/tf_numpy_and_keras.ipynb) ([https://github.com/google/trax/blob/master/trax/tf\\_numpy\\_and\\_keras.ipynb](https://github.com/google/trax/blob/master/trax/tf_numpy_and_keras.ipynb))

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Mixed precision



[View on TensorFlow.org](https://www.tensorflow.org/guide/mixed_precision)

([https://www.tensorflow.org/guide/mixed\\_precision](https://www.tensorflow.org/guide/mixed_precision))



[Run in Google Colab](#)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/mixed\\_precision.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/mixed_precision.ipynb))

([https://github.com/tensorflow/docs/blob/master/site/en/guide/mixed\\_precision.ipynb](https://github.com/tensorflow/docs/blob/master/site/en/guide/mixed_precision.ipynb))

### Overview

Mixed precision is the use of both 16-bit and 32-bit floating-point types in a model during training to make it run faster and use less memory. By keeping certain parts of the model in the 32-bit types for numeric stability, the model will have a lower step time and train equally as well in terms of the evaluation metrics such as accuracy. This guide describes how to use the Keras mixed precision API to speed up your models. Using this API can improve performance by more than 3 times on modern GPUs and 60% on TPUs.

Today, most models use the float32 dtype, which takes 32 bits of memory. However, there are two lower-precision dtypes, float16 and bfloat16, each which take 16 bits of memory instead. Modern accelerators can run operations faster in the 16-bit dtypes, as they have specialized hardware to run 16-bit computations and 16-bit dtypes can be read from memory faster.

NVIDIA GPUs can run operations in float16 faster than in float32, and TPUs can run operations in bfloat16 faster than float32. Therefore, these lower-precision dtypes should be used whenever possible on those devices. However, variables and a few computations should still be in float32 for numeric reasons so that the model trains to the same quality. The Keras mixed precision API allows you to use a mix of either float16 or bfloat16 with float32, to get the performance benefits from float16/bfloat16 and the numeric stability benefits from float32.

Note: In this guide, the term "numeric stability" refers to how a model's quality is affected by the use of a lower-precision dtype instead of a higher precision dtype. An operation is "numerically unstable" in float16 or bfloat16 if running it in one of those dtypes causes the model to have worse evaluation accuracy or other metrics compared to running the operation in float32.

### Setup

In [ ]:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras import mixed_precision
```

### Supported hardware

While mixed precision will run on most hardware, it will only speed up models on recent NVIDIA GPUs and Cloud TPUs. NVIDIA GPUs support using a mix of float16 and float32, while TPUs support a mix of bfloat16 and float32.

Among NVIDIA GPUs, those with compute capability 7.0 or higher will see the greatest performance benefit from mixed precision because they have special hardware units, called Tensor Cores, to accelerate float16 matrix multiplications and convolutions. Older GPUs offer no math performance benefit for using mixed precision, however memory and bandwidth savings can enable some speedups. You can look up the compute capability for your GPU at NVIDIA's [CUDA GPU web page](https://developer.nvidia.com/cuda-gpus) (<https://developer.nvidia.com/cuda-gpus>). Examples of GPUs that will benefit most from mixed precision include RTX GPUs, the V100, and the A100.

Note: If running this guide in Google Colab, the GPU runtime typically has a P100 connected. The P100 has compute capability 6.0 and is not expected to show a significant speedup.

You can check your GPU type with the following. The command only exists if the NVIDIA drivers are installed, so the following will raise an error otherwise.

In [ ]:

```
!nvidia-smi -L
```

All Cloud TPUs support bfloat16.

Even on CPUs and older GPUs, where no speedup is expected, mixed precision APIs can still be used for unit testing, debugging, or just to try out the API. On CPUs, mixed precision will run significantly slower, however.

## Setting the dtype policy

To use mixed precision in Keras, you need to create a `tf.keras.mixed_precision.Policy`, typically referred to as a *dtype policy*. Dtype policies specify the dtypes layers will run in. In this guide, you will construct a policy from the string '`'mixed_float16'`' and set it as the global policy. This will cause subsequently created layers to use mixed precision with a mix of float16 and float32.

In [ ]:

```
policy = mixed_precision.Policy('mixed_float16')
mixed_precision.set_global_policy(policy)
```

For short, you can directly pass a string to `set_global_policy`, which is typically done in practice.

In [ ]:

```
# Equivalent to the two lines above
mixed_precision.set_global_policy('mixed_float16')
```

The policy specifies two important aspects of a layer: the dtype the layer's computations are done in, and the dtype of a layer's variables. Above, you created a `mixed_float16` policy (i.e., a `mixed_precision.Policy` created by passing the string '`'mixed_float16'`' to its constructor). With this policy, layers use float16 computations and float32 variables. Computations are done in float16 for performance, but variables must be kept in float32 for numeric stability. You can directly query these properties of the policy.

In [ ]:

```
print('Compute dtype: %s' % policy.compute_dtype)
print('Variable dtype: %s' % policy.variable_dtype)
```

As mentioned before, the `mixed_float16` policy will most significantly improve performance on NVIDIA GPUs with compute capability of at least 7.0. The policy will run on other GPUs and CPUs but may not improve performance. For TPUs, the `mixed_bfloat16` policy should be used instead.

## Building the model

Next, let's start building a simple model. Very small toy models typically do not benefit from mixed precision, because overhead from the TensorFlow runtime typically dominates the execution time, making any performance improvement on the GPU negligible. Therefore, let's build two large `Dense` layers with 4096 units each if a GPU is used.

In [ ]:

```
inputs = keras.Input(shape=(784,), name='digits')
if tf.config.list_physical_devices('GPU'):
    print('The model will run with 4096 units on a GPU')
    num_units = 4096
else:
    # Use fewer units on CPUs so the model finishes in a reasonable amount of time
    print('The model will run with 64 units on a CPU')
    num_units = 64
dense1 = layers.Dense(num_units, activation='relu', name='dense_1')
x = dense1(inputs)
dense2 = layers.Dense(num_units, activation='relu', name='dense_2')
x = dense2(x)
```

Each layer has a policy and uses the global policy by default. Each of the `Dense` layers therefore have the `mixed_float16` policy because you set the global policy to `mixed_float16` previously. This will cause the dense layers to do float16 computations and have float32 variables. They cast their inputs to float16 in order to do float16 computations, which causes their outputs to be float16 as a result. Their variables are float32 and will be cast to float16 when the layers are called to avoid errors from dtype mismatches.

In [ ]:

```
print(dense1.dtype_policy)
print('x.dtype: %s' % x.dtype.name)
# 'kernel' is dense1's variable
print('dense1.kernel.dtype: %s' % dense1.kernel.dtype.name)
```

Next, create the output predictions. Normally, you can create the output predictions as follows, but this is not always numerically stable with float16.

In [ ]:

```
# INCORRECT: softmax and model output will be float16, when it should be float32
outputs = layers.Dense(10, activation='softmax', name='predictions')(x)
print('Outputs dtype: %s' % outputs.dtype.name)
```

A softmax activation at the end of the model should be float32. Because the dtype policy is `mixed_float16`, the softmax activation would normally have a float16 compute dtype and output float16 tensors.

This can be fixed by separating the Dense and softmax layers, and by passing `dtype='float32'` to the softmax layer:

In [ ]:

```
# CORRECT: softmax and model output are float32
x = layers.Dense(10, name='dense_logits')(x)
outputs = layers.Activation('softmax', dtype='float32', name='predictions')(x)
print('Outputs dtype: %s' % outputs.dtype.name)
```

Passing `dtype='float32'` to the softmax layer constructor overrides the layer's dtype policy to be the `float32` policy, which does computations and keeps variables in float32. Equivalently, you could have instead passed `dtype=mixed_precision.Policy('float32')`; layers always convert the dtype argument to a policy. Because the `Activation` layer has no variables, the policy's variable dtype is ignored, but the policy's compute dtype of float32 causes softmax and the model output to be float32.

Adding a float16 softmax in the middle of a model is fine, but a softmax at the end of the model should be in float32. The reason is that if the intermediate tensor flowing from the softmax to the loss is float16 or bfloat16, numeric issues may occur.

You can override the dtype of any layer to be float32 by passing `dtype='float32'` if you think it will not be numerically stable with float16 computations. But typically, this is only necessary on the last layer of the model, as most layers have sufficient precision with `mixed_float16` and `mixed_bfloat16`.

Even if the model does not end in a softmax, the outputs should still be float32. While unnecessary for this specific model, the model outputs can be cast to float32 with the following:

In [ ]:

```
# The linear activation is an identity function. So this simply casts 'outputs'
# to float32. In this particular case, 'outputs' is already float32 so this is a
# no-op.
outputs = layers.Activation('linear', dtype='float32')(outputs)
```

Next, finish and compile the model, and generate input data:

In [ ]:

```
model = keras.Model(inputs=inputs, outputs=outputs)
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=keras.optimizers.RMSprop(),
              metrics=['accuracy'])

(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.reshape(60000, 784).astype('float32') / 255
x_test = x_test.reshape(10000, 784).astype('float32') / 255
```

This example cast the input data from int8 to float32. You don't cast to float16 since the division by 255 is on the CPU, which runs float16 operations slower than float32 operations. In this case, the performance difference is negligible, but in general you should run input processing math in float32 if it runs on the CPU. The first layer of the model will cast the inputs to float16, as each layer casts floating-point inputs to its compute dtype.

The initial weights of the model are retrieved. This will allow training from scratch again by loading the weights.

In [ ]:

```
initial_weights = model.get_weights()
```

## Training the model with Model.fit

Next, train the model:

In [ ]:

```
history = model.fit(x_train, y_train,
                     batch_size=8192,
                     epochs=5,
                     validation_split=0.2)
test_scores = model.evaluate(x_test, y_test, verbose=2)
print('Test loss:', test_scores[0])
print('Test accuracy:', test_scores[1])
```

Notice the model prints the time per step in the logs: for example, "25ms/step". The first epoch may be slower as TensorFlow spends some time optimizing the model, but afterwards the time per step should stabilize.

If you are running this guide in Colab, you can compare the performance of mixed precision with float32. To do so, change the policy from `mixed_float16` to `float32` in the "Setting the dtype policy" section, then rerun all the cells up to this point. On GPUs with compute capability 7.X, you should see the time per step significantly increase, indicating mixed precision sped up the model. Make sure to change the policy back to `mixed_float16` and rerun the cells before continuing with the guide.

On GPUs with compute capability of at least 8.0 (Ampere GPUs and above), you likely will see no performance improvement in the toy model in this guide when using mixed precision compared to float32. This is due to the use of [TensorFloat-32](#) ([https://www.tensorflow.org/api\\_docs/python/tf/config/experimental/enable\\_tensor\\_float\\_32\\_execution](https://www.tensorflow.org/api_docs/python/tf/config/experimental/enable_tensor_float_32_execution)), which automatically uses lower precision math in certain float32 ops such as `tf.linalg.matmul`. TensorFloat-32 gives some of the performance advantages of mixed precision when using float32. However, in real-world models, you will still typically see significant performance improvements from mixed precision due to memory bandwidth savings and ops which TensorFloat-32 does not support.

If running mixed precision on a TPU, you will not see as much of a performance gain compared to running mixed precision on GPUs, especially pre-Ampere GPUs. This is because TPUs do certain ops in bfloat16 under the hood even with the default dtype policy of float32. This is similar to how Ampere GPUs use TensorFloat-32 by default. Compared to Ampere GPUs, TPUs typically see less performance gains with mixed precision on real-world models.

For many real-world models, mixed precision also allows you to double the batch size without running out of memory, as float16 tensors take half the memory. This does not apply however to this toy model, as you can likely run the model in any dtype where each batch consists of the entire MNIST dataset of 60,000 images.

## Loss scaling

Loss scaling is a technique which `tf.keras.Model.fit` automatically performs with the `mixed_float16` policy to avoid numeric underflow. This section describes what loss scaling is and the next section describes how to use it with a custom training loop.

## Underflow and Overflow

The float16 data type has a narrow dynamic range compared to float32. This means values above 65504 will overflow to infinity and values below  $6.0 \times 10^{-8}$  will underflow to zero. float32 and bfloat16 have a much higher dynamic range so that overflow and underflow are not a problem.

For example:

In [ ]:

```
x = tf.constant(256, dtype='float16')
(x ** 2).numpy() # Overflow
```

In [ ]:

```
x = tf.constant(1e-5, dtype='float16')
(x ** 2).numpy() # Underflow
```

In practice, overflow with float16 rarely occurs. Additionally, underflow also rarely occurs during the forward pass. However, during the backward pass, gradients can underflow to zero. Loss scaling is a technique to prevent this underflow.

## Loss scaling overview

The basic concept of loss scaling is simple: simply multiply the loss by some large number, say 1024, and you get the *loss scale* value. This will cause the gradients to scale by 1024 as well, greatly reducing the chance of underflow. Once the final gradients are computed, divide them by 1024 to bring them back to their correct values.

The pseudocode for this process is:

```
loss_scale = 1024
loss = model(inputs)
loss *= loss_scale
# Assume `grads` are float32. You do not want to divide float16 gradients.
grads = compute_gradient(loss, model.trainable_variables)
grads /= loss_scale
```

Choosing a loss scale can be tricky. If the loss scale is too low, gradients may still underflow to zero. If too high, the opposite the problem occurs: the gradients may overflow to infinity.

To solve this, TensorFlow dynamically determines the loss scale so you do not have to choose one manually. If you use `tf.keras.Model.fit`, loss scaling is done for you so you do not have to do any extra work. If you use a custom training loop, you must explicitly use the special optimizer wrapper `tf.keras.mixed_precision.LossScaleOptimizer` in order to use loss scaling. This is described in the next section.

## Training the model with a custom training loop

So far, you have trained a Keras model with mixed precision using `tf.keras.Model.fit`. Next, you will use mixed precision with a custom training loop. If you do not already know what a custom training loop is, please read the [Custom training guide](#) ([./tutorials/customization/custom\\_training\\_walkthrough.ipynb](#)) first.

Running a custom training loop with mixed precision requires two changes over running it in float32:

1. Build the model with mixed precision (you already did this)
2. Explicitly use loss scaling if `mixed_float16` is used.

For step (2), you will use the `tf.keras.mixed_precision.LossScaleOptimizer` class, which wraps an optimizer and applies loss scaling. By default, it dynamically determines the loss scale so you do not have to choose one. Construct a `LossScaleOptimizer` as follows.

In [ ]:

```
optimizer = keras.optimizers.RMSprop()
optimizer = mixed_precision.LossScaleOptimizer(optimizer)
```

If you want, it is possible choose an explicit loss scale or otherwise customize the loss scaling behavior, but it is highly recommended to keep the default loss scaling behavior, as it has been found to work well on all known models. See the `tf.keras.mixed_precision.LossScaleOptimizer` documentation if you want to customize the loss scaling behavior.

Next, define the loss object and the `tf.data.Dataset`s:

In [ ]:

```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy()
train_dataset = (tf.data.Dataset.from_tensor_slices((x_train, y_train))
                 .shuffle(10000).batch(8192))
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(8192)
```

Next, define the training step function. You will use two new methods from the loss scale optimizer to scale the loss and unscale the gradients:

- `get_scaled_loss(loss)` : Multiplies the loss by the loss scale
- `get_unscaled_gradients(gradients)` : Takes in a list of scaled gradients as inputs, and divides each one by the loss scale to unscale them

These functions must be used in order to prevent underflow in the gradients. `LossScaleOptimizer.apply_gradients` will then apply gradients if none of them have `Inf`s or `Nan`s. It will also update the loss scale, halving it if the gradients had `Inf`s or `Nan`s and potentially increasing it otherwise.

```
In [ ]:
```

```
@tf.function
def train_step(x, y):
    with tf.GradientTape() as tape:
        predictions = model(x)
        loss = loss_object(y, predictions)
        scaled_loss = optimizer.get_scaled_loss(loss)
    scaled_gradients = tape.gradient(scaled_loss, model.trainable_variables)
    gradients = optimizer.get_unscaled_gradients(scaled_gradients)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss
```

The `LossScaleOptimizer` will likely skip the first few steps at the start of training. The loss scale starts out high so that the optimal loss scale can quickly be determined. After a few steps, the loss scale will stabilize and very few steps will be skipped. This process happens automatically and does not affect training quality.

Now, define the test step:

```
In [ ]:
```

```
@tf.function
def test_step(x):
    return model(x, training=False)
```

Load the initial weights of the model, so you can retrain from scratch:

```
In [ ]:
```

```
model.set_weights(initial_weights)
```

Finally, run the custom training loop:

```
In [ ]:
```

```
for epoch in range(5):
    epoch_loss_avg = tf.keras.metrics.Mean()
    test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
        name='test_accuracy')
    for x, y in train_dataset:
        loss = train_step(x, y)
        epoch_loss_avg(loss)
    for x, y in test_dataset:
        predictions = test_step(x)
        test_accuracy.update_state(y, predictions)
    print('Epoch {}: loss={}, test accuracy={}'.format(epoch, epoch_loss_avg.result(), test_accuracy.result()))
```

## GPU performance tips

Here are some performance tips when using mixed precision on GPUs.

### Increasing your batch size

If it doesn't affect model quality, try running with double the batch size when using mixed precision. As float16 tensors use half the memory, this often allows you to double your batch size without running out of memory. Increasing batch size typically increases training throughput, i.e. the training elements per second your model can run on.

### Ensuring GPU Tensor Cores are used

As mentioned previously, modern NVIDIA GPUs use a special hardware unit called Tensor Cores that can multiply float16 matrices very quickly. However, Tensor Cores requires certain dimensions of tensors to be a multiple of 8. In the examples below, an argument is bold if and only if it needs to be a multiple of 8 for Tensor Cores to be used.

- `tf.keras.layers.Dense(units=64)`
- `tf.keras.layers.Conv2d(filters=48, kernel_size=7, stride=3)`
  - And similarly for other convolutional layers, such as `tf.keras.layers.Conv3d`
- `tf.keras.layers.LSTM(units=64)`
  - And similar for other RNNs, such as `tf.keras.layers.GRU`
- `tf.keras.Model.fit(epochs=2, batch_size=128)`

You should try to use Tensor Cores when possible. If you want to learn more, [NVIDIA deep learning performance guide](https://docs.nvidia.com/deeplearning/sdk/dl-performance-guide/index.html) (<https://docs.nvidia.com/deeplearning/sdk/dl-performance-guide/index.html>) describes the exact requirements for using Tensor Cores as well as other Tensor Core-related performance information.

### XLA

XLA is a compiler that can further increase mixed precision performance, as well as float32 performance to a lesser extent. Refer to the [XLA guide](https://www.tensorflow.org/xla) (<https://www.tensorflow.org/xla>) for details.

## Cloud TPU performance tips

As with GPUs, you should try doubling your batch size when using Cloud TPUs because bfloat16 tensors use half the memory. Doubling batch size may increase training throughput.

TPUs do not require any other mixed precision-specific tuning to get optimal performance. They already require the use of XLA. TPUs benefit from having certain dimensions being multiples of 128, but this applies equally to the float32 type as it does for mixed precision. Check the [Cloud TPU performance guide](https://cloud.google.com/tpu/docs/performance-guide) (<https://cloud.google.com/tpu/docs/performance-guide>) for general TPU performance tips, which apply to mixed precision as well as float32 tensors.

## Summary

- You should use mixed precision if you use TPUs or NVIDIA GPUs with at least compute capability 7.0, as it will improve performance by up to 3x.
- You can use mixed precision with the following lines:

```
# On TPUs, use 'mixed_bfloat16' instead
mixed_precision.set_global_policy('mixed_float16')
```

- If your model ends in softmax, make sure it is float32. And regardless of what your model ends in, make sure the output is float32.
- If you use a custom training loop with `mixed_float16`, in addition to the above lines, you need to wrap your optimizer with `tf.keras.mixed_precision.LossScaleOptimizer`. Then call `optimizer.get_scaled_loss` to scale the loss, and `optimizer.get_unscaled_gradients` to unscale the gradients.
- Double the training batch size if it does not reduce evaluation accuracy
- On GPUs, ensure most tensor dimensions are a multiple of 8 to maximize performance

For an example of mixed precision using the `tf.keras.mixed_precision` API, check [functions and classes related to training performance](https://github.com/tensorflow/models/blob/master/official/modeling/performance.py) (<https://github.com/tensorflow/models/blob/master/official/modeling/performance.py>). Check out the official models, such as [Transformer](https://github.com/tensorflow/models/blob/master/official/nlp/modeling/layers/transformer_encoder_block.py) ([https://github.com/tensorflow/models/blob/master/official/nlp/modeling/layers/transformer\\_encoder\\_block.py](https://github.com/tensorflow/models/blob/master/official/nlp/modeling/layers/transformer_encoder_block.py)), for details.

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Training checkpoints



[View on TensorFlow.org](https://www.tensorflow.org/guide/checkpoint)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/checkpoint.ipynb)

(<https://www.tensorflow.org/guide/checkpoint>)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/checkpoint.ipynb>)

(<https://github.com/tensorflow>)

The phrase "Saving a TensorFlow model" typically means one of two things:

1. Checkpoints, OR
2. SavedModel.

Checkpoints capture the exact value of all parameters (`tf.Variable` objects) used by a model. Checkpoints do not contain any description of the computation defined by the model and thus are typically only useful when source code that will use the saved parameter values is available.

The SavedModel format on the other hand includes a serialized description of the computation defined by the model in addition to the parameter values (checkpoint). Models in this format are independent of the source code that created the model. They are thus suitable for deployment via TensorFlow Serving, TensorFlow Lite, TensorFlow.js, or programs in other programming languages (the C, C++, Java, Go, Rust, C# etc. TensorFlow APIs).

This guide covers APIs for writing and reading checkpoints.

## Setup

In [ ]:

```
import tensorflow as tf
```

In [ ]:

```
class Net(tf.keras.Model):
    """A simple linear model."""

    def __init__(self):
        super(Net, self).__init__()
        self.l1 = tf.keras.layers.Dense(5)

    def call(self, x):
        return self.l1(x)
```

In [ ]:

```
net = Net()
```

## Saving from `tf.keras` training APIs

See the [tf.keras guide on saving and restoring](https://www.tensorflow.org/guide/keras/save_and_serialize) ([https://www.tensorflow.org/guide/keras/save\\_and\\_serialize](https://www.tensorflow.org/guide/keras/save_and_serialize)).

`tf.keras.Model.save_weights` saves a TensorFlow checkpoint.

In [ ]:

```
net.save_weights('easy_checkpoint')
```

## Writing checkpoints

The persistent state of a TensorFlow model is stored in `tf.Variable` objects. These can be constructed directly, but are often created through high-level APIs like `tf.keras.layers` or `tf.keras.Model`.

The easiest way to manage variables is by attaching them to Python objects, then referencing those objects.

Subclasses of `tf.train.Checkpoint`, `tf.keras.layers.Layer`, and `tf.keras.Model` automatically track variables assigned to their attributes. The following example constructs a simple linear model, then writes checkpoints which contain values for all of the model's variables.

You can easily save a model-checkpoint with `Model.save_weights`.

## Manual checkpointing

### Setup

To help demonstrate all the features of `tf.train.Checkpoint`, define a toy dataset and optimization step:

In [ ]:

```
def toy_dataset():
    inputs = tf.range(10.)[:, None]
    labels = inputs * 5. + tf.range(5.)[None, :]
    return tf.data.Dataset.from_tensor_slices(
        dict(x=inputs, y=labels)).repeat().batch(2)
```

In [ ]:

```
def train_step(net, example, optimizer):
    """Trains `net` on `example` using `optimizer`."""
    with tf.GradientTape() as tape:
        output = net(example['x'])
        loss = tf.reduce_mean(tf.abs(output - example['y']))
    variables = net.trainable_variables
    gradients = tape.gradient(loss, variables)
    optimizer.apply_gradients(zip(gradients, variables))
    return loss
```

### Create the checkpoint objects

Use a `tf.train.Checkpoint` object to manually create a checkpoint, where the objects you want to checkpoint are set as attributes on the object.

A `tf.train.CheckpointManager` can also be helpful for managing multiple checkpoints.

In [ ]:

```
opt = tf.keras.optimizers.Adam(0.1)
dataset = toy_dataset()
iterator = iter(dataset)
ckpt = tf.train.Checkpoint(step=tf.Variable(1), optimizer=opt, net=net, iterator=iterator)
manager = tf.train.CheckpointManager(ckpt, './tf_ckpts', max_to_keep=3)
```

### Train and checkpoint the model

The following training loop creates an instance of the model and of an optimizer, then gathers them into a `tf.train.Checkpoint` object. It calls the training step in a loop on each batch of data, and periodically writes checkpoints to disk.

In [ ]:

```
def train_and_checkpoint(net, manager):
    ckpt.restore(manager.latest_checkpoint)
    if manager.latest_checkpoint:
        print("Restored from {}".format(manager.latest_checkpoint))
    else:
        print("Initializing from scratch.")

    for _ in range(50):
        example = next(iterator)
        loss = train_step(net, example, opt)
        ckpt.step.assign_add(1)
        if int(ckpt.step) % 10 == 0:
            save_path = manager.save()
            print("Saved checkpoint for step {}: {}".format(int(ckpt.step), save_path))
            print("loss {:.2f}".format(loss.numpy()))
```

In [ ]:

```
train_and_checkpoint(net, manager)
```

## Restore and continue training

After the first training cycle you can pass a new model and manager, but pick up training exactly where you left off:

In [ ]:

```
opt = tf.keras.optimizers.Adam(0.1)
net = Net()
dataset = toy_dataset()
iterator = iter(dataset)
ckpt = tf.train.Checkpoint(step=tf.Variable(1), optimizer=opt, net=net, iterator=iterator)
manager = tf.train.CheckpointManager(ckpt, './tf_ckpts', max_to_keep=3)

train_and_checkpoint(net, manager)
```

The `tf.train.CheckpointManager` object deletes old checkpoints. Above it's configured to keep only the three most recent checkpoints.

In [ ]:

```
print(manager.checkpoints) # List the three remaining checkpoints
```

These paths, e.g. `'./tf_ckpts/ckpt-10'`, are not files on disk. Instead they are prefixes for an `index` file and one or more data files which contain the variable values. These prefixes are grouped together in a single `checkpoint` file (`'./tf_ckpts/checkpoint'`) where the `CheckpointManager` saves its state.

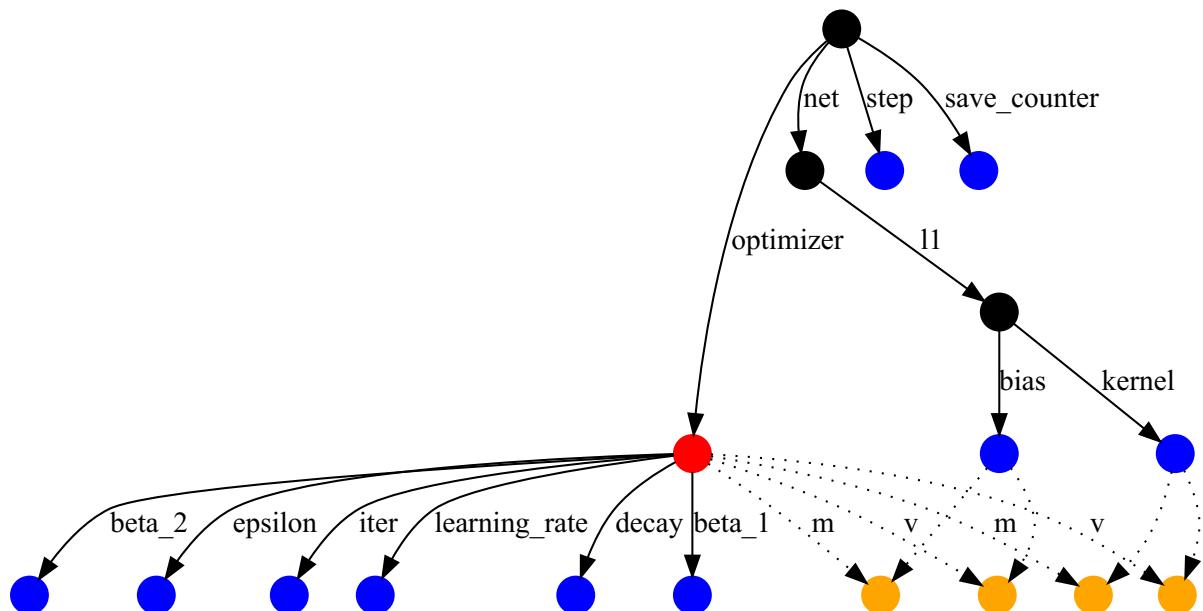
In [ ]:

```
!ls ./tf_ckpts
```

## Loading mechanics

TensorFlow matches variables to checkpointed values by traversing a directed graph with named edges, starting from the object being loaded. Edge names typically come from attribute names in objects, for example the "l1" in `self.l1 = tf.keras.layers.Dense(5)`. `tf.train.Checkpoint` uses its keyword argument names, as in the "step" in `tf.train.Checkpoint(step=...)`.

The dependency graph from the example above looks like this:



The optimizer is in red, regular variables are in blue, and the optimizer slot variables are in orange. The other nodes—for example, representing the `tf.train.Checkpoint`—are in black.

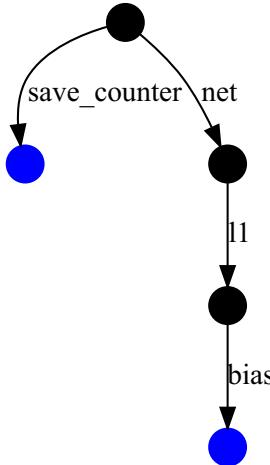
Slot variables are part of the optimizer's state, but are created for a specific variable. For example, the 'm' edges above correspond to momentum, which the Adam optimizer tracks for each variable. Slot variables are only saved in a checkpoint if the variable and the optimizer would both be saved, thus the dashed edges.

Calling `restore` on a `tf.train.Checkpoint` object queues the requested restorations, restoring variable values as soon as there's a matching path from the `Checkpoint` object. For example, you can load just the bias from the model you defined above by reconstructing one path to it through the network and the layer.

In [ ]:

```
to_restore = tf.Variable(tf.zeros([5]))
print(to_restore.numpy()) # All zeros
fake_layer = tf.train.Checkpoint(bias=to_restore)
fake_net = tf.train.Checkpoint(l1=fake_layer)
new_root = tf.train.Checkpoint(net=fake_net)
status = new_root.restore(tf.train.latest_checkpoint('./tf_ckpts/'))
print(to_restore.numpy()) # This gets the restored value.
```

The dependency graph for these new objects is a much smaller subgraph of the larger checkpoint you wrote above. It includes only the bias and a save counter that `tf.train.Checkpoint` uses to number checkpoints.



`restore` returns a `status` object, which has optional assertions. All of the objects created in the new `Checkpoint` have been restored, so `status.assert_existing_objects_matched` passes.

In [ ]:

```
status.assert_existing_objects_matched()
```

There are many objects in the checkpoint which haven't matched, including the layer's kernel and the optimizer's variables.

`status.assert_consumed` only passes if the checkpoint and the program match exactly, and would throw an exception here.

## Deferred restorations

`Layer` objects in TensorFlow may defer the creation of variables to their first call, when input shapes are available. For example, the shape of a `Dense` layer's kernel depends on both the layer's input and output shapes, and so the output shape required as a constructor argument is not enough information to create the variable on its own. Since calling a `Layer` also reads the variable's value, a restore must happen between the variable's creation and its first use.

To support this idiom, `tf.train.Checkpoint` defers restores which don't yet have a matching variable.

In [ ]:

```
deferred_restore = tf.Variable(tf.zeros([1, 5]))
print(deferred_restore.numpy()) # Not restored; still zeros
fake_layer.kernel = deferred_restore
print(deferred_restore.numpy()) # Restored
```

## Manually inspecting checkpoints

`tf.train.load_checkpoint` returns a `CheckpointReader` that gives lower level access to the checkpoint contents. It contains mappings from each variable's key, to the shape and dtype for each variable in the checkpoint. A variable's key is its object path, like in the graphs displayed above.

Note: There is no higher level structure to the checkpoint. It only knows the paths and values for the variables, and has no concept of `models`, `layers` or how they are connected.

In [ ]:

```
reader = tf.train.load_checkpoint('./tf_ckpts/')
shape_from_key = reader.get_variable_to_shape_map()
dtype_from_key = reader.get_variable_to_dtype_map()

sorted(shape_from_key.keys())
```

So if you're interested in the value of `net.l1.kernel` you can get the value with the following code:

In [ ]:

```
key = 'net/l1/kernel/.ATTRIBUTES/VARIABLE_VALUE'

print("Shape:", shape_from_key[key])
print("Dtype:", dtype_from_key[key].name)
```

It also provides a `get_tensor` method allowing you to inspect the value of a variable:

In [ ]:

```
reader.get_tensor(key)
```

## Object tracking

Checkpoints save and restore the values of `tf.Variable` objects by "tracking" any variable or trackable object set in one of its attributes. When executing a save, variables are gathered recursively from all of the reachable tracked objects.

As with direct attribute assignments like `self.l1 = tf.keras.layers.Dense(5)`, assigning lists and dictionaries to attributes will track their contents.

In [ ]:

```
save = tf.train.Checkpoint()
save.listed = [tf.Variable(1.)]
save.listed.append(tf.Variable(2.))
save.mapped = {'one': save.listed[0]}
save.mapped['two'] = save.listed[1]
save_path = save.save('./tf_list_example')

restore = tf.train.Checkpoint()
v2 = tf.Variable(0.)
assert 0. == v2.numpy() # Not restored yet
restore.mapped = {'two': v2}
restore.restore(save_path)
assert 2. == v2.numpy()
```

You may notice wrapper objects for lists and dictionaries. These wrappers are checkpointable versions of the underlying data-structures. Just like the attribute based loading, these wrappers restore a variable's value as soon as it's added to the container.

In [ ]:

```
restore.listed = []
print	restore.listed # ListWrapper([])
v1 = tf.Variable(0.)
restore.listed.append(v1) # Restores v1, from restore() in the previous cell
assert 1. == v1.numpy()
```

Trackable objects include `tf.train.Checkpoint`, `tf.Module` and its subclasses (e.g. `keras.layers.Layer` and `keras.Model`), and recognized Python containers:

- `dict` (and `collections.OrderedDict`)
- `list`
- `tuple` (and `collections.namedtuple`, `typing.NamedTuple`)

Other container types are **not supported**, including:

- `collections.defaultdict`
- `set`

All other Python objects are **ignored**, including:

- `int`
- `string`
- `float`

## Summary

TensorFlow objects provide an easy automatic mechanism for saving and restoring the values of variables they use.

**Copyright 2020 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Introduction to graphs and `tf.function`



[View on TensorFlow.org](https://www.tensorflow.org/guide/intro_to_graphs)  
([https://www.tensorflow.org/guide/intro\\_to\\_graphs](https://www.tensorflow.org/guide/intro_to_graphs))



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/intro_to_graphs.ipynb)  
([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/intro\\_to\\_graphs.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/intro_to_graphs.ipynb))

## Overview

This guide goes beneath the surface of TensorFlow and Keras to demonstrate how TensorFlow works. If you instead want to immediately get started with Keras, check out the [collection of Keras guides](https://www.tensorflow.org/guide/keras) (<https://www.tensorflow.org/guide/keras>).

In this guide, you'll learn how TensorFlow allows you to make simple changes to your code to get graphs, how graphs are stored and represented, and how you can use them to accelerate your models.

Note: For those of you who are only familiar with TensorFlow 1.x, this guide demonstrates a very different view of graphs.

**This is a big-picture overview that covers how `tf.function` allows you to switch from eager execution to graph execution.** For a more complete specification of `tf.function`, go to the [tf.function guide \(function.ipynb\)](https://www.tensorflow.org/guide/function.ipynb).

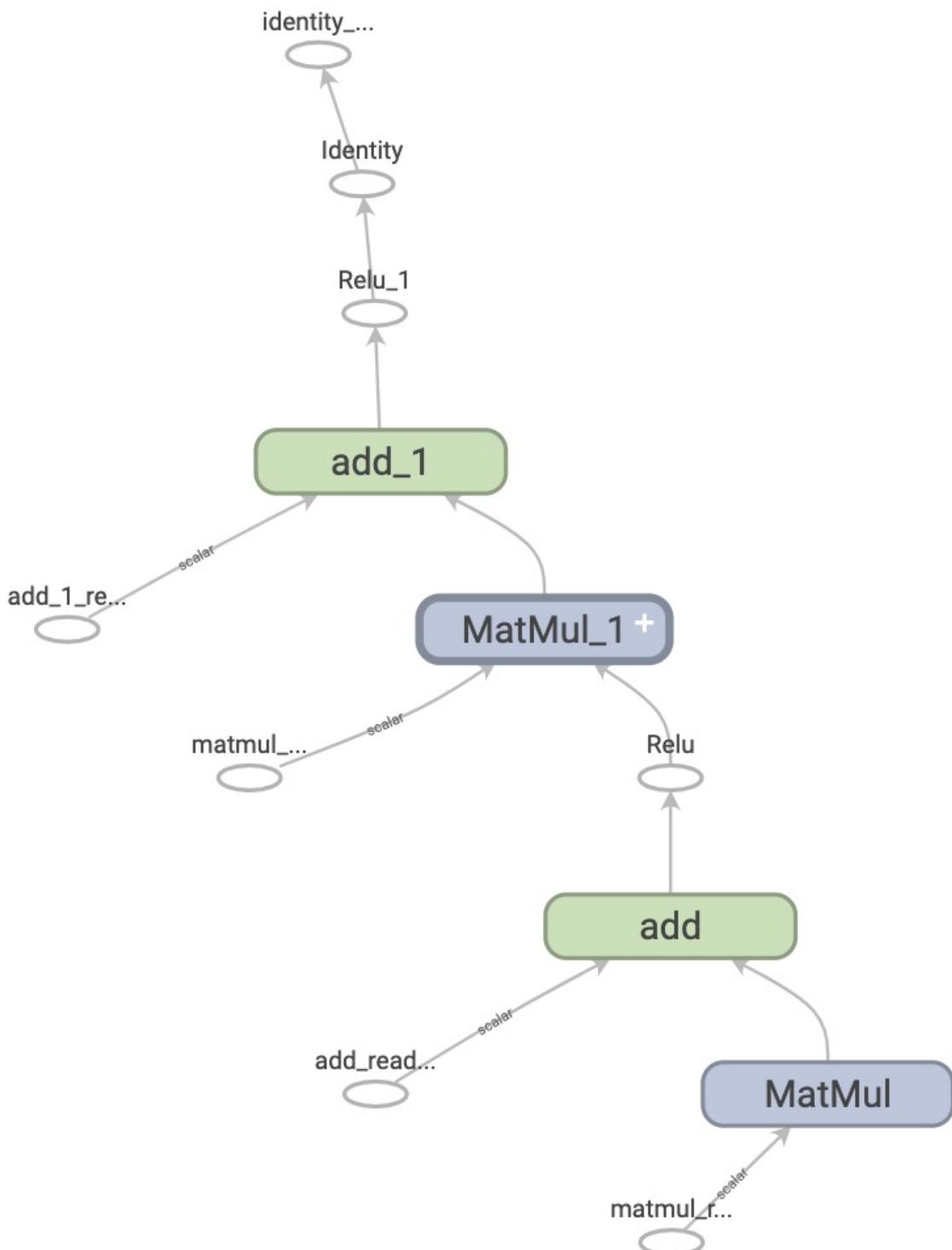
## What are graphs?

In the previous three guides, you ran TensorFlow **eagerly**. This means TensorFlow operations are executed by Python, operation by operation, and returning results back to Python.

While eager execution has several unique advantages, graph execution enables portability outside Python and tends to offer better performance. **Graph execution** means that tensor computations are executed as a *TensorFlow graph*, sometimes referred to as a `tf.Graph` or simply a "graph."

Graphs are data structures that contain a set of `tf.Operation` objects, which represent units of computation; and `tf.Tensor` objects, which represent the units of data that flow between operations. They are defined in a `tf.Graph` context. Since these graphs are data structures, they can be saved, run, and restored all without the original Python code.

This is what a TensorFlow graph representing a two-layer neural network looks like when visualized in TensorBoard.



## The benefits of graphs

With a graph, you have a great deal of flexibility. You can use your TensorFlow graph in environments that don't have a Python interpreter, like mobile applications, embedded devices, and backend servers. TensorFlow uses graphs as the format for [saved models \(saved\\_model\)](#) when it exports them from Python.

Graphs are also easily optimized, allowing the compiler to do transformations like:

- Statically infer the value of tensors by folding constant nodes in your computation ("*constant folding*").
- Separate sub-parts of a computation that are independent and split them between threads or devices.
- Simplify arithmetic operations by eliminating common subexpressions.

There is an entire optimization system, [Grappler \(./graph\\_optimization.ipynb\)](#), to perform this and other speedups.

In short, graphs are extremely useful and let your TensorFlow run **fast**, run in **parallel**, and run efficiently **on multiple devices**.

However, you still want to define your machine learning models (or other computations) in Python for convenience, and then automatically construct graphs when you need them.

## Setup

In [ ]:

```
import tensorflow as tf
import timeit
from datetime import datetime
```

## Taking advantage of graphs

You create and run a graph in TensorFlow by using `tf.function`, either as a direct call or as a decorator. `tf.function` takes a regular function as input and returns a `Function`. A `Function` is a Python callable that builds TensorFlow graphs from the Python function. You use a `Function` in the same way as its Python equivalent.

In [ ]:

```
# Define a Python function.
def a_regular_function(x, y, b):
    x = tf.matmul(x, y)
    x = x + b
    return x

# `a_function_that_uses_a_graph` is a TensorFlow `Function`.
a_function_that_uses_a_graph = tf.function(a_regular_function)

# Make some tensors.
x1 = tf.constant([[1.0, 2.0]])
y1 = tf.constant([[2.0], [3.0]])
b1 = tf.constant(4.0)

orig_value = a_regular_function(x1, y1, b1).numpy()
# Call a `Function` like a Python function.
tf_function_value = a_function_that_uses_a_graph(x1, y1, b1).numpy()
assert(orig_value == tf_function_value)
```

On the outside, a `Function` looks like a regular function you write using TensorFlow operations. [Underneath \(\[https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/eager/def\\\_function.py\]\(https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/eager/def\_function.py\)\)](#), however, it is *very different*. A `Function` **encapsulates several `tf.Graph`s behind one API**. That is how `Function` is able to give you the [benefits of graph execution](#), like speed and deployability.

`tf.function` applies to a function *and all other functions it calls*:

In [ ]:

```
def inner_function(x, y, b):
    x = tf.matmul(x, y)
    x = x + b
    return x

# Use the decorator to make `outer_function` a `Function`.
@tf.function
def outer_function(x):
    y = tf.constant([[2.0], [3.0]])
    b = tf.constant(4.0)

    return inner_function(x, y, b)

# Note that the callable will create a graph that
# includes `inner_function` as well as `outer_function`.
outer_function(tf.constant([[1.0, 2.0]])).numpy()
```

If you have used TensorFlow 1.x, you will notice that at no time did you need to define a `Placeholder` or `tf.Session`.

## Converting Python functions to graphs

Any function you write with TensorFlow will contain a mixture of built-in TF operations and Python logic, such as `if-then` clauses, loops, `break`, `return`, `continue`, and more. While TensorFlow operations are easily captured by a `tf.Graph`, Python-specific logic needs to undergo an extra step in order to become part of the graph. `tf.function` uses a library called AutoGraph (`tf.autograph`) to convert Python code into graph-generating code.

In [ ]:

```
def simple_relu(x):
    if tf.greater(x, 0):
        return x
    else:
        return 0

# `tf_simple_relu` is a TensorFlow `Function` that wraps `simple_relu`.
tf_simple_relu = tf.function(simple_relu)

print("First branch, with graph:", tf_simple_relu(tf.constant(1)).numpy())
print("Second branch, with graph:", tf_simple_relu(tf.constant(-1)).numpy())
```

Though it is unlikely that you will need to view graphs directly, you can inspect the outputs to check the exact results. These are not easy to read, so no need to look too carefully!

In [ ]:

```
# This is the graph-generating output of AutoGraph.
print(tf.autograph.to_code(simple_relu))
```

In [ ]:

```
# This is the graph itself.
print(tf_simple_relu.get_concrete_function(tf.constant(1)).graph.as_graph_def())
```

Most of the time, `tf.function` will work without special considerations. However, there are some caveats, and the [tf.function guide \(./function.ipynb\)](#) can help here, as well as the [complete AutoGraph reference](#) (<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/autograph/g3doc/reference/index.md>)

## Polymorphism: one Function , many graphs

A `tf.Graph` is specialized to a specific type of inputs (for example, tensors with a specific `dtype` ([https://www.tensorflow.org/api\\_docs/python/tf/dtypes/DType](https://www.tensorflow.org/api_docs/python/tf/dtypes/DType)) or objects with the same `id()` (<https://docs.python.org/3/library/functions.html#id>)).

Each time you invoke a `Function` with a set of arguments that can't be handled by any of its existing graphs (such as arguments with new `dtypes` or incompatible shapes), `Function` creates a new `tf.Graph` specialized to those new arguments. The type specification of a `tf.Graph`'s inputs is known as its **input signature** or just a **signature**. For more information regarding when a new `tf.Graph` is generated and how that can be controlled, see the [rules of retracing](#) ([https://www.tensorflow.org/guide/function#rules\\_of\\_tracing](https://www.tensorflow.org/guide/function#rules_of_tracing)).

The `Function` stores the `tf.Graph` corresponding to that signature in a `ConcreteFunction`. A **ConcreteFunction** is a wrapper around a `tf.Graph`.

In [ ]:

```
@tf.function
def my_relu(x):
    return tf.maximum(0., x)

# `my_relu` creates new graphs as it observes more signatures.
print(my_relu(tf.constant(5.5)))
print(my_relu([1, -1]))
print(my_relu(tf.constant([3., -3.])))
```

If the `Function` has already been called with that signature, `Function` does not create a new `tf.Graph`.

In [ ]:

```
# These two calls do *not* create new graphs.
print(my_relu(tf.constant(-2.5))) # Signature matches `tf.constant(5.5)`.
print(my_relu(tf.constant([-1., 1.]))) # Signature matches `tf.constant([3., -3.])`.
```

Because it's backed by multiple graphs, a `Function` is **polymorphic**. That enables it to support more input types than a single `tf.Graph` could represent, as well as to optimize each `tf.Graph` for better performance.

In [ ]:

```
# There are three `ConcreteFunction`s (one for each graph) in `my_relu`.
# The `ConcreteFunction` also knows the return type and shape!
print(my_relu.pretty_printed_concrete_signatures())
```

## Using `tf.function`

So far, you've learned how to convert a Python function into a graph simply by using `tf.function` as a decorator or wrapper. But in practice, getting `tf.function` to work correctly can be tricky! In the following sections, you'll learn how you can make your code work as expected with `tf.function`.

### Graph execution vs. eager execution

The code in a `Function` can be executed both eagerly and as a graph. By default, `Function` executes its code as a graph:

In [ ]:

```
@tf.function
def get_MSE(y_true, y_pred):
    sq_diff = tf.pow(y_true - y_pred, 2)
    return tf.reduce_mean(sq_diff)
```

In [ ]:

```
y_true = tf.random.uniform([5], maxval=10, dtype=tf.int32)
y_pred = tf.random.uniform([5], maxval=10, dtype=tf.int32)
print(y_true)
print(y_pred)
```

In [ ]:

```
get_MSE(y_true, y_pred)
```

To verify that your `Function`'s graph is doing the same computation as its equivalent Python function, you can make it execute eagerly with `tf.config.run_functions_eagerly(True)`. This is a switch that **turns off `Function`'s ability to create and run graphs**, instead executing the code normally.

In [ ]:

```
tf.config.run_functions_eagerly(True)
```

In [ ]:

```
get_MSE(y_true, y_pred)
```

In [ ]:

```
# Don't forget to set it back when you are done.
tf.config.run_functions_eagerly(False)
```

However, `Function` can behave differently under graph and eager execution. The Python `print` (<https://docs.python.org/3/library/functions.html#print>) function is one example of how these two modes differ. Let's check out what happens when you insert a `print` statement to your function and call it repeatedly.

In [ ]:

```
@tf.function
def get_MSE(y_true, y_pred):
    print("Calculating MSE!")
    sq_diff = tf.pow(y_true - y_pred, 2)
    return tf.reduce_mean(sq_diff)
```

Observe what is printed:

In [ ]:

```
error = get_MSE(y_true, y_pred)
error = get_MSE(y_true, y_pred)
error = get_MSE(y_true, y_pred)
```

Is the output surprising? `get_MSE` only printed once even though it was called three times.

To explain, the `print` statement is executed when `Function` runs the original code in order to create the graph in a process known as "[tracing](#)" ([function.ipynb#tracing](#)). Tracing captures the TensorFlow operations into a graph, and `print` is not captured in the graph. That graph is then executed for all three calls without ever running the Python code again.

As a sanity check, let's turn off graph execution to compare:

In [ ]:

```
# Now, globally set everything to run eagerly to force eager execution.
tf.config.run_functions_eagerly(True)
```

In [ ]:

```
# Observe what is printed below.
error = get_MSE(y_true, y_pred)
error = get_MSE(y_true, y_pred)
error = get_MSE(y_true, y_pred)
```

In [ ]:

```
tf.config.run_functions_eagerly(False)
```

`print` is a *Python side effect*, and there are other differences that you should be aware of when converting a function into a `Function`. Learn more in the [Limitations](#) section of the [Better performance with tf.function](#) ([./function.ipynb#limitations](#)) guide.

Note: If you would like to print values in both eager and graph execution, use `tf.print` instead.

## Non-strict execution

Graph execution only executes the operations necessary to produce the observable effects, which includes:

- The return value of the function
- Documented well-known side-effects such as:
  - Input/output operations, like `tf.print`
  - Debugging operations, such as the assert functions in `tf.debugging`
  - Mutations of `tf.Variable`

This behavior is usually known as "Non-strict execution", and differs from eager execution, which steps through all of the program operations, needed or not.

In particular, runtime error checking does not count as an observable effect. If an operation is skipped because it is unnecessary, it cannot raise any runtime errors.

In the following example, the "unnecessary" operation `tf.gather` is skipped during graph execution, so the runtime error `InvalidArgumentError` is not raised as it would be in eager execution. Do not rely on an error being raised while executing a graph.

```
In [ ]:
```

```
def unused_return_eager(x):
    # Get index 1 will fail when `len(x) == 1`
    tf.gather(x, [1]) # unused
    return x

try:
    print(unused_return_eager(tf.constant([0.0])))
except tf.errors.InvalidArgumentError as e:
    # All operations are run during eager execution so an error is raised.
    print(f'{type(e).__name__}: {e}'')
```

```
In [ ]:
```

```
@tf.function
def unused_return_graph(x):
    tf.gather(x, [1]) # unused
    return x

# Only needed operations are run during graph exection. The error is not raised.
print(unused_return_graph(tf.constant([0.0])))
```

## tf.function best practices

It may take some time to get used to the behavior of `Function`. To get started quickly, first-time users should play around with decorating toy functions with `@tf.function` to get experience with going from eager to graph execution.

*Designing for `tf.function`* may be your best bet for writing graph-compatible TensorFlow programs. Here are some tips:

- Toggle between eager and graph execution early and often with `tf.config.run_functions_eagerly` to pinpoint if/ when the two modes diverge.
- Create `tf.Variable`s outside the Python function and modify them on the inside. The same goes for objects that use `tf.Variable`, like `keras.layers`, `keras.Models` and `tf.optimizers`.
- Avoid writing functions that [depend on outer Python variables \(function#depending\\_on\\_python\\_global\\_and\\_free\\_variables\)](#), excluding `tf.Variable`s and Keras objects.
- Prefer to write functions which take tensors and other TensorFlow types as input. You can pass in other object types but [be careful \(function#depending\\_on\\_python\\_objects\)](#)!
- Include as much computation as possible under a `tf.function` to maximize the performance gain. For example, decorate a whole training step or the entire training loop.

## Seeing the speed-up

`tf.function` usually improves the performance of your code, but the amount of speed-up depends on the kind of computation you run. Small computations can be dominated by the overhead of calling a graph. You can measure the difference in performance like so:

```
In [ ]:
```

```
x = tf.random.uniform(shape=[10, 10], minval=-1, maxval=2, dtype=tf.dtypes.int32)

def power(x, y):
    result = tf.eye(10, dtype=tf.dtypes.int32)
    for _ in range(y):
        result = tf.matmul(x, result)
    return result
```

```
In [ ]:
```

```
print("Eager execution:", timeit.timeit(lambda: power(x, 100), number=1000))
```

```
In [ ]:
```

```
power_as_graph = tf.function(power)
print("Graph execution:", timeit.timeit(lambda: power_as_graph(x, 100), number=1000))
```

`tf.function` is commonly used to speed up training loops, and you can learn more about it in [Writing a training loop from scratch \(https://www.tensorflow.org/guide/keras/writing\\_a\\_training\\_loop\\_from\\_scratch#speeding-up\\_your\\_training\\_step\\_with\\_tffunction\)](#) with Keras.

Note: You can also try `tf.function(jit_compile=True)` ([https://www.tensorflow.org/xla#explicit\\_compilation\\_with\\_tffunctionjit\\_compiletrue](https://www.tensorflow.org/xla#explicit_compilation_with_tffunctionjit_compiletrue)) for a more significant performance boost, especially if your code is heavy on TF control flow and uses many small tensors.

## Performance and trade-offs

Graphs can speed up your code, but the process of creating them has some overhead. For some functions, the creation of the graph takes more time than the execution of the graph. **This investment is usually quickly paid back with the performance boost of subsequent executions, but it's important to be aware that the first few steps of any large model training can be slower due to tracing.**

No matter how large your model, you want to avoid tracing frequently. The `tf.function` guide discusses [how to set input specifications and use tensor arguments \(function#controlling\\_retracing\)](#) to avoid retracing. If you find you are getting unusually poor performance, it's a good idea to check if you are retracing accidentally.

## When is a Function tracing?

To figure out when your `Function` is tracing, add a `print` statement to its code. As a rule of thumb, `Function` will execute the `print` statement every time it traces.

In [ ]:

```
@tf.function
def a_function_with_python_side_effect(x):
    print("Tracing!") # An eager-only side effect.
    return x * x + tf.constant(2)

# This is traced the first time.
print(a_function_with_python_side_effect(tf.constant(2)))
# The second time through, you won't see the side effect.
print(a_function_with_python_side_effect(tf.constant(3)))
```

In [ ]:

```
# This retraces each time the Python argument changes,
# as a Python argument could be an epoch count or other
# hyperparameter.
print(a_function_with_python_side_effect(2))
print(a_function_with_python_side_effect(3))
```

New Python arguments always trigger the creation of a new graph, hence the extra tracing.

## Next steps

You can learn more about `tf.function` on the API reference page and by following the [Better performance with `tf.function` \(function.ipynb\)](#) guide.

**Copyright 2018 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Use a GPU



[View on TensorFlow.org](https://www.tensorflow.org/guide/gpu)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/gpu.ipynb)



[\(https://github.com/tensorflow/docs/blob/master/site/en/guide/gpu.ipynb\)](https://github.com/tensorflow/docs/blob/master/site/en/guide/gpu.ipynb)

TensorFlow code, and `tf.keras` models will transparently run on a single GPU with no code changes required.

Note: Use `tf.config.list_physical_devices('GPU')` to confirm that TensorFlow is using the GPU.

The simplest way to run on multiple GPUs, on one or many machines, is using [Distribution Strategies \(distributed\\_training.ipynb\)](#).

This guide is for users who have tried these approaches and found that they need fine-grained control of how TensorFlow uses the GPU. To learn how to debug performance issues for single and multi-GPU scenarios, see the [Optimize TensorFlow GPU Performance \(gpu\\_performance\\_analysis.md\)](#) guide.

## Setup

Ensure you have the latest TensorFlow gpu release installed.

In [ ]:

```
import tensorflow as tf
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
```

## Overview

TensorFlow supports running computations on a variety of types of devices, including CPU and GPU. They are represented with string identifiers for example:

- `"/device:CPU:0"` : The CPU of your machine.
- `"/GPU:0"` : Short-hand notation for the first GPU of your machine that is visible to TensorFlow.
- `"/job:localhost/replica:0/task:0/device:GPU:1"` : Fully qualified name of the second GPU of your machine that is visible to TensorFlow.

If a TensorFlow operation has both CPU and GPU implementations, by default, the GPU device is prioritized when the operation is assigned. For example, `tf.matmul` has both CPU and GPU kernels and on a system with devices `CPU:0` and `GPU:0`, the `GPU:0` device is selected to run `tf.matmul` unless you explicitly request to run it on another device.

If a TensorFlow operation has no corresponding GPU implementation, then the operation falls back to the CPU device. For example, since `tf.cast` only has a CPU kernel, on a system with devices `CPU:0` and `GPU:0`, the `CPU:0` device is selected to run `tf.cast`, even if requested to run on the `GPU:0` device.

## Logging device placement

To find out which devices your operations and tensors are assigned to, put `tf.debugging.set_log_device_placement(True)` as the first statement of your program. Enabling device placement logging causes any Tensor allocations or operations to be printed.

In [ ]:

```
tf.debugging.set_log_device_placement(True)

# Create some tensors
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
c = tf.matmul(a, b)

print(c)
```

The above code will print an indication the `MatMul` op was executed on `GPU:0`.

## Manual device placement

If you would like a particular operation to run on a device of your choice instead of what's automatically selected for you, you can use `with tf.device` to create a device context, and all the operations within that context will run on the same designated device.

In [ ]:

```
tf.debugging.set_log_device_placement(True)

# Place tensors on the CPU
with tf.device('/CPU:0'):
    a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
    b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])

# Run on the GPU
c = tf.matmul(a, b)
print(c)
```

You will see that now `a` and `b` are assigned to `CPU:0`. Since a device was not explicitly specified for the `MatMul` operation, the TensorFlow runtime will choose one based on the operation and available devices (`GPU:0` in this example) and automatically copy tensors between devices if required.

## Limits GPU memory growth

By default, TensorFlow maps nearly all of the GPU memory of all GPUs (subject to `CUDA_VISIBLE_DEVICES` (<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#env-vars>)) visible to the process. This is done to more efficiently use the relatively precious GPU memory resources on the devices by reducing memory fragmentation. To limit TensorFlow to a specific set of GPUs, use the `tf.config.set_visible_devices` method.

In [ ]:

```
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    # Restrict TensorFlow to only use the first GPU
    try:
        tf.config.set_visible_devices(gpus[0], 'GPU')
        logical_gpus = tf.config.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPU")
    except RuntimeError as e:
        # Visible devices must be set before GPUs have been initialized
        print(e)
```

In some cases it is desirable for the process to only allocate a subset of the available memory, or to only grow the memory usage as is needed by the process. TensorFlow provides two methods to control this.

The first option is to turn on memory growth by calling `tf.config.experimental.set_memory_growth`, which attempts to allocate only as much GPU memory as needed for the runtime allocations: it starts out allocating very little memory, and as the program gets run and more GPU memory is needed, the GPU memory region is extended for the TensorFlow process. Memory is not released since it can lead to memory fragmentation. To turn on memory growth for a specific GPU, use the following code prior to allocating any tensors or executing any ops.

In [ ]:

```
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    try:
        # Currently, memory growth needs to be the same across GPUs
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
        logical_gpus = tf.config.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
    except RuntimeError as e:
        # Memory growth must be set before GPUs have been initialized
        print(e)
```

Another way to enable this option is to set the environmental variable `TF_FORCE_GPU_ALLOW_GROWTH` to `true`. This configuration is platform specific.

The second method is to configure a virtual GPU device with `tf.config.set_logical_device_configuration` and set a hard limit on the total memory to allocate on the GPU.

In [ ]:

```
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    # Restrict TensorFlow to only allocate 1GB of memory on the first GPU
    try:
        tf.config.set_logical_device_configuration(
            gpus[0],
            [tf.config.LogicalDeviceConfiguration(memory_limit=1024)])
        logical_gpus = tf.config.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
    except RuntimeError as e:
        # Virtual devices must be set before GPUs have been initialized
        print(e)
```

This is useful if you want to truly bound the amount of GPU memory available to the TensorFlow process. This is common practice for local development when the GPU is shared with other applications such as a workstation GUI.

## Using a single GPU on a multi-GPU system

If you have more than one GPU in your system, the GPU with the lowest ID will be selected by default. If you would like to run on a different GPU, you will need to specify the preference explicitly:

In [ ]:

```
tf.debugging.set_log_device_placement(True)

try:
    # Specify an invalid GPU device
    with tf.device('/device:GPU:2'):
        a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
        b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
        c = tf.matmul(a, b)
except RuntimeError as e:
    print(e)
```

If the device you have specified does not exist, you will get a `RuntimeError : .../device:GPU:2 unknown device`.

If you would like TensorFlow to automatically choose an existing and supported device to run the operations in case the specified one doesn't exist, you can call `tf.config.set_soft_device_placement(True)`.

In [ ]:

```
tf.config.set_soft_device_placement(True)
tf.debugging.set_log_device_placement(True)

# Creates some tensors
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
c = tf.matmul(a, b)

print(c)
```

## Using multiple GPUs

Developing for multiple GPUs will allow a model to scale with the additional resources. If developing on a system with a single GPU, you can simulate multiple GPUs with virtual devices. This enables easy testing of multi-GPU setups without requiring additional resources.

In [ ]:

```
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    # Create 2 virtual GPUs with 1GB memory each
    try:
        tf.config.set_logical_device_configuration(
            gpus[0],
            [tf.config.LogicalDeviceConfiguration(memory_limit=1024),
             tf.config.LogicalDeviceConfiguration(memory_limit=1024)])
        logical_gpus = tf.config.list_logical_devices('GPU')
        print(len(gpus), "Physical GPU,", len(logical_gpus), "Logical GPUs")
    except RuntimeError as e:
        # Virtual devices must be set before GPUs have been initialized
        print(e)
```

Once there are multiple logical GPUs available to the runtime, you can utilize the multiple GPUs with `tf.distribute.Strategy` or with manual placement.

### With `tf.distribute.Strategy`

The best practice for using multiple GPUs is to use `tf.distribute.Strategy`. Here is a simple example:

In [ ]:

```
tf.debugging.set_log_device_placement(True)
gpus = tf.config.list_logical_devices('GPU')
strategy = tf.distribute.MirroredStrategy(gpus)
with strategy.scope():
    inputs = tf.keras.layers.Input(shape=(1,))
    predictions = tf.keras.layers.Dense(1)(inputs)
    model = tf.keras.models.Model(inputs=inputs, outputs=predictions)
    model.compile(loss='mse',
                  optimizer=tf.keras.optimizers.SGD(learning_rate=0.2))
```

This program will run a copy of your model on each GPU, splitting the input data between them, also known as "[data parallelism](#) ([https://en.wikipedia.org/wiki/Data\\_parallelism](https://en.wikipedia.org/wiki/Data_parallelism))".

For more information about distribution strategies, check out the guide [here \(./distributed\\_training.ipynb\)](#).

## Manual placement

`tf.distribute.Strategy` works under the hood by replicating computation across devices. You can manually implement replication by constructing your model on each GPU. For example:

In [ ]:

```
tf.debugging.set_log_device_placement(True)

gpus = tf.config.list_logical_devices('GPU')
if gpus:
    # Replicate your computation on multiple GPUs
    c = []
    for gpu in gpus:
        with tf.device(gpu.name):
            a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
            b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
            c.append(tf.matmul(a, b))

    with tf.device('/CPU:0'):
        matmul_sum = tf.add_n(c)

    print(matmul_sum)
```

**Copyright 2020 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Introduction to Tensors



[View on TensorFlow.org](https://www.tensorflow.org/guide/tensor)

(<https://www.tensorflow.org/guide/tensor>)



[Run in Google Colab](#)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/tensor.ipynb>)

In [ ]:

```
import tensorflow as tf
import numpy as np
```

Tensors are multi-dimensional arrays with a uniform type (called a `dtype`). You can see all supported `dtypes` at `tf.dtypes.DType`.

If you're familiar with [NumPy](https://numpy.org/devdocs/user/quickstart.html) (<https://numpy.org/devdocs/user/quickstart.html>), tensors are (kind of) like `np.arrays`.

All tensors are immutable like Python numbers and strings: you can never update the contents of a tensor, only create a new one.

## Basics

Let's create some basic tensors.

Here is a "scalar" or "rank-0" tensor. A scalar contains a single value, and no "axes".

In [ ]:

```
# This will be an int32 tensor by default; see "dtypes" below.  
rank_0_tensor = tf.constant(4)  
print(rank_0_tensor)
```

A "vector" or "rank-1" tensor is like a list of values. A vector has one axis:

In [ ]:

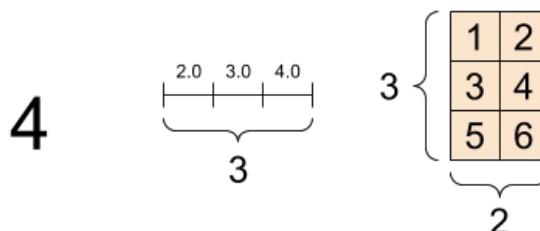
```
# Let's make this a float tensor.  
rank_1_tensor = tf.constant([2.0, 3.0, 4.0])  
print(rank_1_tensor)
```

A "matrix" or "rank-2" tensor has two axes:

In [ ]:

```
# If you want to be specific, you can set the dtype (see below) at creation time  
rank_2_tensor = tf.constant([[1, 2],  
                           [3, 4],  
                           [5, 6]], dtype=tf.float16)  
print(rank_2_tensor)
```

A scalar, shape: []   A vector, shape: [3]   A matrix, shape: [3, 2]

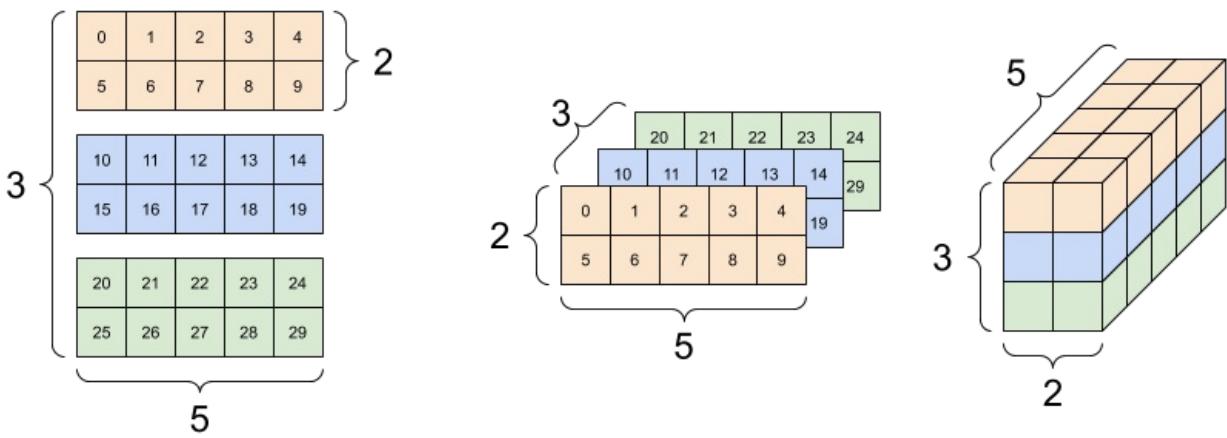


Tensors may have more axes; here is a tensor with three axes:

In [ ]:

```
# There can be an arbitrary number of  
# axes (sometimes called "dimensions")  
rank_3_tensor = tf.constant([  
    [[0, 1, 2, 3, 4],  
     [5, 6, 7, 8, 9]],  
    [[10, 11, 12, 13, 14],  
     [15, 16, 17, 18, 19]],  
    [[20, 21, 22, 23, 24],  
     [25, 26, 27, 28, 29]],])  
  
print(rank_3_tensor)
```

There are many ways you might visualize a tensor with more than two axes.



You can convert a tensor to a NumPy array either using `np.array` or the `tensor.numpy` method:

In [ ]:

```
np.array(rank_2_tensor)
```

In [ ]:

```
rank_2_tensor.numpy()
```

Tensors often contain floats and ints, but have many other types, including:

- complex numbers
- strings

The base `tf.Tensor` class requires tensors to be "rectangular"—that is, along each axis, every element is the same size. However, there are specialized types of tensors that can handle different shapes:

- Ragged tensors (see [RaggedTensor](#) below)
- Sparse tensors (see [SparseTensor](#) below)

You can do basic math on tensors, including addition, element-wise multiplication, and matrix multiplication.

In [ ]:

```
a = tf.constant([[1, 2],
                 [3, 4]])
b = tf.constant([[1, 1],
                 [1, 1]]) # Could have also said `tf.ones([2,2])`

print(tf.add(a, b), "\n")
print(tf.multiply(a, b), "\n")
print(tf.matmul(a, b), "\n")
```

In [ ]:

```
print(a + b, "\n") # element-wise addition
print(a * b, "\n") # element-wise multiplication
print(a @ b, "\n") # matrix multiplication
```

Tensors are used in all kinds of operations (ops).

In [ ]:

```
c = tf.constant([[4.0, 5.0], [10.0, 1.0]])

# Find the largest value
print(tf.reduce_max(c))
# Find the index of the largest value
print(tf.argmax(c))
# Compute the softmax
print(tf.nn.softmax(c))
```

## About shapes

Tensors have shapes. Some vocabulary:

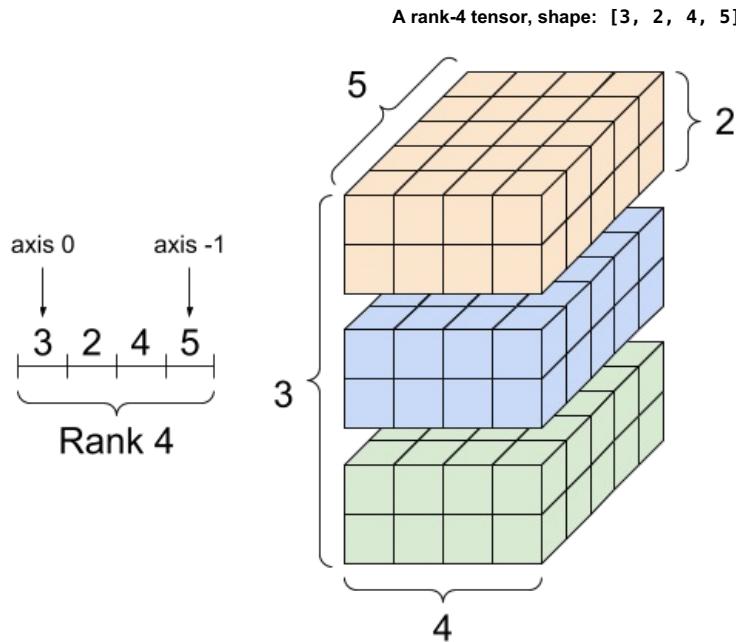
- **Shape:** The length (number of elements) of each of the axes of a tensor.
- **Rank:** Number of tensor axes. A scalar has rank 0, a vector has rank 1, a matrix is rank 2.
- **Axis or Dimension:** A particular dimension of a tensor.
- **Size:** The total number of items in the tensor, the product of the shape vector's elements.

Note: Although you may see reference to a "tensor of two dimensions", a rank-2 tensor does not usually describe a 2D space.

Tensors and `tf.TensorShape` objects have convenient properties for accessing these:

In [ ]:

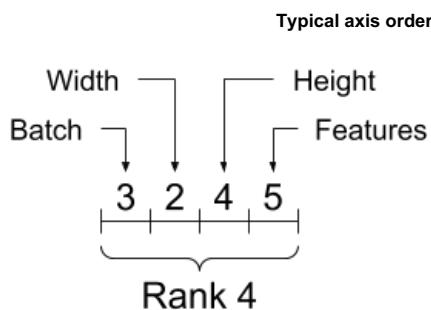
```
rank_4_tensor = tf.zeros([3, 2, 4, 5])
```



In [ ]:

```
print("Type of every element:", rank_4_tensor.dtype)
print("Number of axes:", rank_4_tensor.ndim)
print("Shape of tensor:", rank_4_tensor.shape)
print("Elements along axis 0 of tensor:", rank_4_tensor.shape[0])
print("Elements along the last axis of tensor:", rank_4_tensor.shape[-1])
print("Total number of elements (3*2*4*5): ", tf.size(rank_4_tensor).numpy())
```

While axes are often referred to by their indices, you should always keep track of the meaning of each. Often axes are ordered from global to local: The batch axis first, followed by spatial dimensions, and features for each location last. This way feature vectors are contiguous regions of memory.



## Indexing

## Single-axis indexing

TensorFlow follows standard Python indexing rules, similar to [indexing a list or a string in Python](https://docs.python.org/3/tutorial/introduction.html#strings) (<https://docs.python.org/3/tutorial/introduction.html#strings>), and the basic rules for NumPy indexing.

- indexes start at 0
- negative indices count backwards from the end
- colons, :, are used for slices: start:stop:step

In [ ]:

```
rank_1_tensor = tf.constant([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
print(rank_1_tensor.numpy())
```

Indexing with a scalar removes the axis:

In [ ]:

```
print("First:", rank_1_tensor[0].numpy())
print("Second:", rank_1_tensor[1].numpy())
print("Last:", rank_1_tensor[-1].numpy())
```

Indexing with a : slice keeps the axis:

In [ ]:

```
print("Everything:", rank_1_tensor[:].numpy())
print("Before 4:", rank_1_tensor[:4].numpy())
print("From 4 to the end:", rank_1_tensor[4:].numpy())
print("From 2, before 7:", rank_1_tensor[2:7].numpy())
print("Every other item:", rank_1_tensor[::-2].numpy())
print("Reversed:", rank_1_tensor[::-1].numpy())
```

## Multi-axis indexing

Higher rank tensors are indexed by passing multiple indices.

The exact same rules as in the single-axis case apply to each axis independently.

In [ ]:

```
print(rank_2_tensor.numpy())
```

Passing an integer for each index, the result is a scalar.

In [ ]:

```
# Pull out a single value from a 2-rank tensor
print(rank_2_tensor[1, 1].numpy())
```

You can index using any combination of integers and slices:

In [ ]:

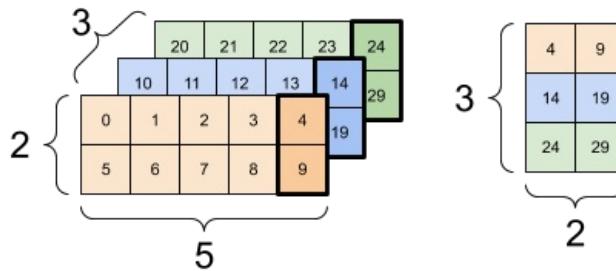
```
# Get row and column tensors
print("Second row:", rank_2_tensor[1, :].numpy())
print("Second column:", rank_2_tensor[:, 1].numpy())
print("Last row:", rank_2_tensor[-1, :].numpy())
print("First item in last column:", rank_2_tensor[0, -1].numpy())
print("Skip the first row:")
print(rank_2_tensor[1:, :].numpy(), "\n")
```

Here is an example with a 3-axis tensor:

In [ ]:

```
print(rank_3_tensor[:, :, 4])
```

Selecting the last feature across all locations in each example in the batch



Read the [tensor slicing guide \(https://tensorflow.org/guide/tensor\\_slicing\)](https://tensorflow.org/guide/tensor_slicing) to learn how you can apply indexing to manipulate individual elements in your tensors.

## Manipulating Shapes

Reshaping a tensor is of great utility.

In [ ]:

```
# Shape returns a `TensorShape` object that shows the size along each axis
x = tf.constant([[1], [2], [3]])
print(x.shape)
```

In [ ]:

```
# You can convert this object into a Python list, too
print(x.shape.as_list())
```

You can reshape a tensor into a new shape. The `tf.reshape` operation is fast and cheap as the underlying data does not need to be duplicated.

In [ ]:

```
# You can reshape a tensor to a new shape.
# Note that you're passing in a list
reshaped = tf.reshape(x, [1, 3])
```

In [ ]:

```
print(x.shape)
print(reshaped.shape)
```

The data maintains its layout in memory and a new tensor is created, with the requested shape, pointing to the same data. TensorFlow uses C-style "row-major" memory ordering, where incrementing the rightmost index corresponds to a single step in memory.

In [ ]:

```
print(rank_3_tensor)
```

If you flatten a tensor you can see what order it is laid out in memory.

In [ ]:

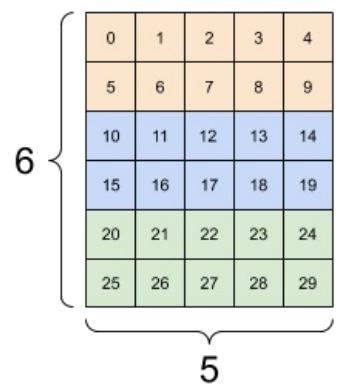
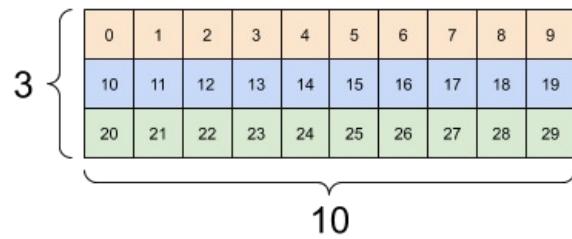
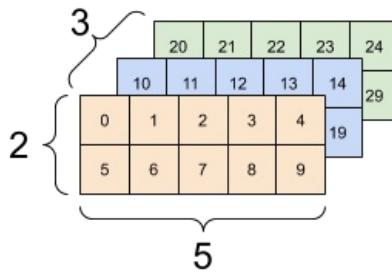
```
# A `-1` passed in the `shape` argument says "Whatever fits".
print(tf.reshape(rank_3_tensor, [-1]))
```

Typically the only reasonable use of `tf.reshape` is to combine or split adjacent axes (or add/remove 1's).

For this 3x2x5 tensor, reshaping to (3x2)x5 or 3x(2x5) are both reasonable things to do, as the slices do not mix:

In [ ]:

```
print(tf.reshape(rank_3_tensor, [3*2, 5]), "\n")
print(tf.reshape(rank_3_tensor, [3, -1]))
```



Reshaping will "work" for any new shape with the same total number of elements, but it will not do anything useful if you do not respect the order of the axes.

Swapping axes in `tf.reshape` does not work; you need `tf.transpose` for that.

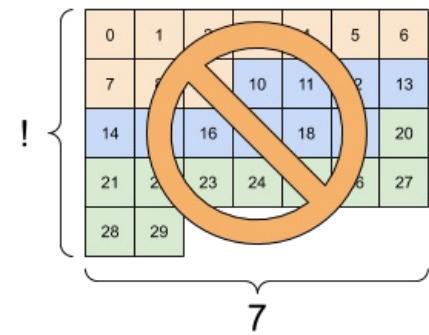
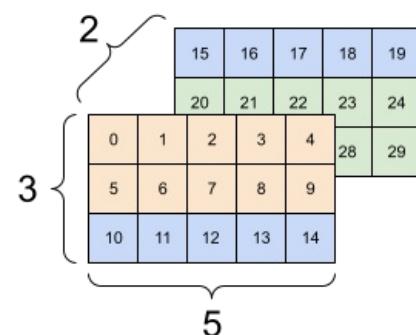
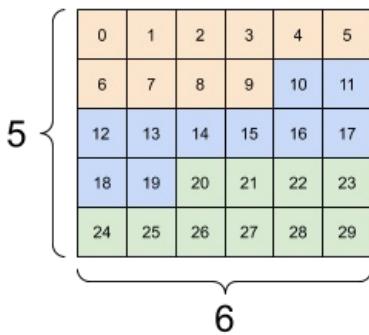
In [ ]:

```
# Bad examples: don't do this

# You can't reorder axes with reshape.
print(tf.reshape(rank_3_tensor, [2, 3, 5]), "\n")

# This is a mess
print(tf.reshape(rank_3_tensor, [5, 6]), "\n")

# This doesn't work at all
try:
    tf.reshape(rank_3_tensor, [7, -1])
except Exception as e:
    print(f"type(e).__name__: {e}")
```



You may run across not-fully-specified shapes. Either the shape contains a `None` (an axis-length is unknown) or the whole shape is `None` (the rank of the tensor is unknown).

Except for [tf.RaggedTensor](#), such shapes will only occur in the context of TensorFlow's symbolic, graph-building APIs:

- [tf.function \(function.ipynb\)](#)
- The [keras functional API \(https://www.tensorflow.org/guide/keras/functional\)](https://www.tensorflow.org/guide/keras/functional).

## More on DTYPES

To inspect a `tf.Tensor`'s data type use the `Tensor.dtype` property.

When creating a `tf.Tensor` from a Python object you may optionally specify the datatype.

If you don't, TensorFlow chooses a datatype that can represent your data. TensorFlow converts Python integers to `tf.int32` and Python floating point numbers to `tf.float32`. Otherwise TensorFlow uses the same rules NumPy uses when converting to arrays.

You can cast from type to type.

In [ ]:

```
the_f64_tensor = tf.constant([2.2, 3.3, 4.4], dtype=tf.float64)
the_f16_tensor = tf.cast(the_f64_tensor, dtype=tf.float16)
# Now, cast to an uint8 and lose the decimal precision
the_u8_tensor = tf.cast(the_f16_tensor, dtype=tf.uint8)
print(the_u8_tensor)
```

## Broadcasting

Broadcasting is a concept borrowed from the [equivalent feature in NumPy](https://numpy.org/doc/stable/user/basics.html) (<https://numpy.org/doc/stable/user/basics.html>). In short, under certain conditions, smaller tensors are "stretched" automatically to fit larger tensors when running combined operations on them.

The simplest and most common case is when you attempt to multiply or add a tensor to a scalar. In that case, the scalar is broadcast to be the same shape as the other argument.

In [ ]:

```
x = tf.constant([1, 2, 3])
y = tf.constant(2)
z = tf.constant([2, 2, 2])
# All of these are the same computation
print(tf.multiply(x, 2))
print(x * y)
print(x * z)
```

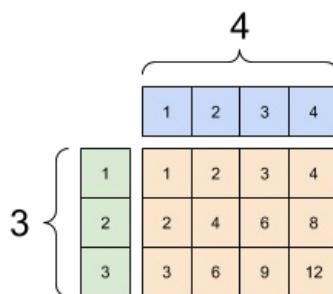
Likewise, axes with length 1 can be stretched out to match the other arguments. Both arguments can be stretched in the same computation.

In this case a 3x1 matrix is element-wise multiplied by a 1x4 matrix to produce a 3x4 matrix. Note how the leading 1 is optional: The shape of y is [4].

In [ ]:

```
# These are the same computations
x = tf.reshape(x,[3,1])
y = tf.range(1, 5)
print(x, "\n")
print(y, "\n")
print(tf.multiply(x, y))
```

A broadcasted add: a [3, 1] times a [1, 4] gives a [3,4]



Here is the same operation without broadcasting:

In [ ]:

```
x_stretch = tf.constant([[1, 1, 1, 1],
                        [2, 2, 2, 2],
                        [3, 3, 3, 3]])
y_stretch = tf.constant([[1, 2, 3, 4],
                        [1, 2, 3, 4],
                        [1, 2, 3, 4]])
print(x_stretch * y_stretch) # Again, operator overloading
```

Most of the time, broadcasting is both time and space efficient, as the broadcast operation never materializes the expanded tensors in memory.

You see what broadcasting looks like using `tf.broadcast_to`.

In [ ]:

```
print(tf.broadcast_to(tf.constant([1, 2, 3]), [3, 3]))
```

Unlike a mathematical op, for example, `broadcast_to` does nothing special to save memory. Here, you are materializing the tensor.

It can get even more complicated. [This section \(https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html\)](https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html) of Jake VanderPlas's book *Python Data Science Handbook* shows more broadcasting tricks (again in NumPy).

## tf.convert\_to\_tensor

Most ops, like `tf.matmul` and `tf.reshape` take arguments of class `tf.Tensor`. However, you'll notice in the above case, Python objects shaped like tensors are accepted.

Most, but not all, ops call `convert_to_tensor` on non-tensor arguments. There is a registry of conversions, and most object classes like NumPy's `ndarray`, `TensorShape`, Python lists, and `tf.Variable` will all convert automatically.

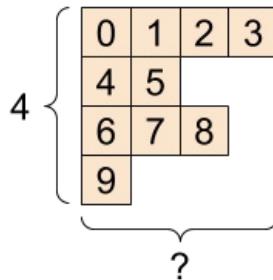
See `tf.register_tensor_conversion_function` for more details, and if you have your own type you'd like to automatically convert to a tensor.

## Ragged Tensors

A tensor with variable numbers of elements along some axis is called "ragged". Use `tf.ragged.RaggedTensor` for ragged data.

For example, This cannot be represented as a regular tensor:

A `tf.RaggedTensor`, shape: [4, None]



In [ ]:

```
ragged_list = [
    [0, 1, 2, 3],
    [4, 5],
    [6, 7, 8],
    [9]]
```

In [ ]:

```
try:
    tensor = tf.constant(ragged_list)
except Exception as e:
    print(f"type(e).__name__: {e}")
```

Instead create a `tf.RaggedTensor` using `tf.ragged.constant`:

In [ ]:

```
ragged_tensor = tf.ragged.constant(ragged_list)
print(ragged_tensor)
```

The shape of a `tf.RaggedTensor` will contain some axes with unknown lengths:

In [ ]:

```
print(ragged_tensor.shape)
```

## String tensors

`tf.string` is a `dtype`, which is to say you can represent data as strings (variable-length byte arrays) in tensors.

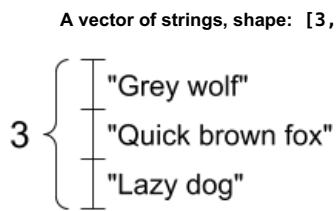
The strings are atomic and cannot be indexed the way Python strings are. The length of the string is not one of the axes of the tensor. See `tf.strings` for functions to manipulate them.

Here is a scalar string tensor:

In [ ]:

```
# Tensors can be strings, too here is a scalar string.  
scalar_string_tensor = tf.constant("Gray wolf")  
print(scalar_string_tensor)
```

And a vector of strings:



In [ ]:

```
# If you have three string tensors of different lengths, this is OK.  
tensor_of_strings = tf.constant(["Gray wolf",  
                                "Quick brown fox",  
                                "Lazy dog"])  
# Note that the shape is (3,). The string length is not included.  
print(tensor_of_strings)
```

In the above printout the `b` prefix indicates that `tf.string` dtype is not a unicode string, but a byte-string. See the [Unicode Tutorial](https://www.tensorflow.org/tutorials/load_data/unicode) ([https://www.tensorflow.org/tutorials/load\\_data/unicode](https://www.tensorflow.org/tutorials/load_data/unicode)) for more about working with unicode text in TensorFlow.

If you pass unicode characters they are utf-8 encoded.

In [ ]:

```
tf.constant("")
```

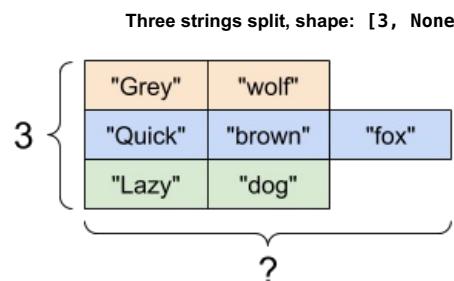
Some basic functions with strings can be found in `tf.strings`, including `tf.strings.split`.

In [ ]:

```
# You can use split to split a string into a set of tensors  
print(tf.strings.split(scalar_string_tensor, sep=" "))
```

In [ ]:

```
# ...but it turns into a `RaggedTensor` if you split up a tensor of strings,  
# as each string might be split into a different number of parts.  
print(tf.strings.split(tensor_of_strings))
```



And `tf.string.to_number`:

```
In [ ]:  
text = tf.constant("1 10 100")  
print(tf.strings.to_number(tf.strings.split(text, " ")))
```

Although you can't use `tf.cast` to turn a string tensor into numbers, you can convert it into bytes, and then into numbers.

In [ ]:

```
byte_strings = tf.strings.bytes_split(tf.constant("Duck"))
byte_ints = tf.io.decode_raw(tf.constant("Duck"), tf.uint8)
print("Byte strings:", byte_strings)
print("Bytes:", byte_ints)
```

In [ ]:

```
# Or split it up as unicode and then decode it
unicode_bytes = tf.constant("アヒル")
unicode_char_bytes = tf.strings.unicode_split(unicode_bytes, "UTF-8")
unicode_values = tf.strings.unicode_decode(unicode_bytes, "UTF-8")

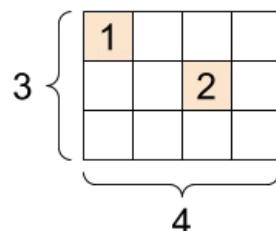
print("\nUnicode bytes:", unicode_bytes)
print("\nUnicode chars:", unicode_char_bytes)
print("\nUnicode values:", unicode_values)
```

The `tf.string` dtype is used for all raw bytes data in TensorFlow. The `tf.io` module contains functions for converting data to and from bytes, including decoding images and parsing csv.

## Sparse tensors

Sometimes, your data is sparse, like a very wide embedding space. TensorFlow supports `tf.sparse.SparseTensor` and related operations to store sparse data efficiently.

A `tf.SparseTensor`, shape: [3, 4]



In [ ]:

```
# Sparse tensors store values by index in a memory-efficient manner
sparse_tensor = tf.sparse.SparseTensor(indices=[[0, 0], [1, 2]],
   values=[1, 2],
   dense_shape=[3, 4])
print(sparse_tensor, "\n")

# You can convert sparse tensors to dense
print(tf.sparse.to_dense(sparse_tensor))
```

**Copyright 2018 The TensorFlow Authors.**

Licensed under the Apache License, Version 2.0 (the "License");

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License"); { display-mode: "form" }
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## tf.data: Build TensorFlow input pipelines



[View on TensorFlow.org](https://www.tensorflow.org/guide/data)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/data.ipynb)

(<https://www.tensorflow.org/guide/data>) (<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/data.ipynb>) (<https://github.com/tensorflow/docs/blob/master/site/en/guide/data.ipynb>)

The `tf.data` API enables you to build complex input pipelines from simple, reusable pieces. For example, the pipeline for an image model might aggregate data from files in a distributed file system, apply random perturbations to each image, and merge randomly selected images into a batch for training. The pipeline for a text model might involve extracting symbols from raw text data, converting them to embedding identifiers with a lookup table, and batching together sequences of different lengths. The `tf.data` API makes it possible to handle large amounts of data, read from different data formats, and perform complex transformations.

The `tf.data` API introduces a `tf.data.Dataset` abstraction that represents a sequence of elements, in which each element consists of one or more components. For example, in an image pipeline, an element might be a single training example, with a pair of tensor components representing the image and its label.

There are two distinct ways to create a dataset:

- A data **source** constructs a `Dataset` from data stored in memory or in one or more files.
- A data **transformation** constructs a dataset from one or more `tf.data.Dataset` objects.

In [ ]:

```
import tensorflow as tf
```

In [ ]:

```
import pathlib
import os
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

np.set_printoptions(precision=4)
```

## Basic mechanics

To create an input pipeline, you must start with a data **source**. For example, to construct a `Dataset` from data in memory, you can use `tf.data.Dataset.from_tensors()` or `tf.data.Dataset.from_tensor_slices()`. Alternatively, if your input data is stored in a file in the recommended TFRecord format, you can use `tf.data.TFRecordDataset()`.

Once you have a `Dataset` object, you can *transform* it into a new `Dataset` by chaining method calls on the `tf.data.Dataset` object. For example, you can apply per-element transformations such as `Dataset.map()`, and multi-element transformations such as `Dataset.batch()`. See the documentation for `tf.data.Dataset` for a complete list of transformations.

The `Dataset` object is a Python iterable. This makes it possible to consume its elements using a for loop:

In [ ]:

```
dataset = tf.data.Dataset.from_tensor_slices([8, 3, 0, 8, 2, 1])
dataset
```

In [ ]:

```
for elem in dataset:
    print(elem.numpy())
```

Or by explicitly creating a Python iterator using `iter` and consuming its elements using `next`:

In [ ]:

```
it = iter(dataset)
print(next(it).numpy())
```

Alternatively, dataset elements can be consumed using the `reduce` transformation, which reduces all elements to produce a single result. The following example illustrates how to use the `reduce` transformation to compute the sum of a dataset of integers.

In [ ]:

```
print(dataset.reduce(0, lambda state, value: state + value).numpy())
```

## Dataset structure

A dataset produces a sequence of *elements*, where each element is the same (nested) structure of *components*. Individual components of the structure can be of any type representable by `tf.TypeSpec`, including `tf.Tensor`, `tf.sparse.SparseTensor`, `tf.RaggedTensor`, `tf.TensorArray`, or `tf.data.Dataset`.

The Python constructs that can be used to express the (nested) structure of elements include `tuple`, `dict`, `NamedTuple`, and `OrderedDict`. In particular, `list` is not a valid construct for expressing the structure of dataset elements. This is because early `tf.data` users felt strongly about `list` inputs (e.g. passed to `tf.data.Dataset.from_tensors`) being automatically packed as tensors and `list` outputs (e.g. return values of user-defined functions) being coerced into a `tuple`. As a consequence, if you would like a `list` input to be treated as a structure, you need to convert it into `tuple` and if you would like a `list` output to be a single component, then you need to explicitly pack it using `tf.stack`.

The `Dataset.element_spec` property allows you to inspect the type of each element component. The property returns a *nested structure* of `tf.TypeSpec` objects, matching the structure of the element, which may be a single component, a tuple of components, or a nested tuple of components. For example:

In [ ]:

```
dataset1 = tf.data.Dataset.from_tensor_slices(tf.random.uniform([4, 10]))  
dataset1.element_spec
```

In [ ]:

```
dataset2 = tf.data.Dataset.from_tensor_slices(  
    (tf.random.uniform([4]),  
     tf.random.uniform([4, 100], maxval=100, dtype=tf.int32)))  
dataset2.element_spec
```

In [ ]:

```
dataset3 = tf.data.Dataset.zip((dataset1, dataset2))  
dataset3.element_spec
```

In [ ]:

```
# Dataset containing a sparse tensor.  
dataset4 = tf.data.Dataset.from_tensors(tf.SparseTensor(indices=[[0, 0], [1, 2]], values=[1, 2], dense_shape=[3, 4]))  
dataset4.element_spec
```

In [ ]:

```
# Use value_type to see the type of value represented by the element spec  
dataset4.element_spec.value_type
```

The `Dataset` transformations support datasets of any structure. When using the `Dataset.map()`, and `Dataset.filter()` transformations, which apply a function to each element, the element structure determines the arguments of the function:

In [ ]:

```
dataset1 = tf.data.Dataset.from_tensor_slices(  
    tf.random.uniform([4, 10], minval=1, maxval=10, dtype=tf.int32))  
dataset1
```

In [ ]:

```
for z in dataset1:  
    print(z.numpy())
```

In [ ]:

```
dataset2 = tf.data.Dataset.from_tensor_slices(  
    (tf.random.uniform([4]),  
     tf.random.uniform([4, 100], maxval=100, dtype=tf.int32)))  
dataset2
```

In [ ]:

```
dataset3 = tf.data.Dataset.zip((dataset1, dataset2))

dataset3
```

In [ ]:

```
for a, (b,c) in dataset3:
    print('shapes: {a.shape}, {b.shape}, {c.shape}'.format(a=a, b=b, c=c))
```

## Reading input data

### Consuming NumPy arrays

See [Loading NumPy arrays \(..../tutorials/load\\_data/numpy.ipynb\)](#) for more examples.

If all of your input data fits in memory, the simplest way to create a `Dataset` from them is to convert them to `tf.Tensor` objects and use `Dataset.from_tensor_slices()`.

In [ ]:

```
train, test = tf.keras.datasets.fashion_mnist.load_data()
```

In [ ]:

```
images, labels = train
images = images/255

dataset = tf.data.Dataset.from_tensor_slices((images, labels))
dataset
```

Note: The above code snippet will embed the `features` and `labels` arrays in your TensorFlow graph as `tf.constant()` operations. This works well for a small dataset, but wastes memory---because the contents of the array will be copied multiple times---and can run into the 2GB limit for the `tf.GraphDef` protocol buffer.

### Consuming Python generators

Another common data source that can easily be ingested as a `tf.data.Dataset` is the python generator.

Caution: While this is a convenient approach it has limited portability and scalability. It must run in the same python process that created the generator, and is still subject to the Python [GIL \(\[https://en.wikipedia.org/wiki/Global\\\_interpreter\\\_lock\]\(https://en.wikipedia.org/wiki/Global\_interpreter\_lock\)\)](#).

In [ ]:

```
def count(stop):
    i = 0
    while i<stop:
        yield i
        i += 1
```

In [ ]:

```
for n in count(5):
    print(n)
```

The `Dataset.from_generator` constructor converts the python generator to a fully functional `tf.data.Dataset`.

The constructor takes a callable as input, not an iterator. This allows it to restart the generator when it reaches the end. It takes an optional `args` argument, which is passed as the callable's arguments.

The `output_types` argument is required because `tf.data` builds a `tf.Graph` internally, and graph edges require a `tf.dtype`.

In [ ]:

```
ds_counter = tf.data.Dataset.from_generator(count, args=[25], output_types=tf.int32, output_shapes = (), )
```

In [ ]:

```
for count_batch in ds_counter.repeat().batch(10).take(10):
    print(count_batch.numpy())
```

The `output_shapes` argument is not *required* but is highly recommended as many TensorFlow operations do not support tensors with an unknown rank. If the length of a particular axis is unknown or variable, set it as `None` in the `output_shapes`.

It's also important to note that the `output_shapes` and `output_types` follow the same nesting rules as other dataset methods.

Here is an example generator that demonstrates both aspects, it returns tuples of arrays, where the second array is a vector with unknown length.

In [ ]:

```
def gen_series():
    i = 0
    while True:
        size = np.random.randint(0, 10)
        yield i, np.random.normal(size=(size,))
        i += 1
```

In [ ]:

```
for i, series in gen_series():
    print(i, ":", str(series))
    if i > 5:
        break
```

The first output is an `int32` the second is a `float32`.

The first item is a scalar, shape `()`, and the second is a vector of unknown length, shape `(None, )`

In [ ]:

```
ds_series = tf.data.Dataset.from_generator(
    gen_series,
    output_types=(tf.int32, tf.float32),
    output_shapes=(((), (None,)))
```

```
ds_series
```

Now it can be used like a regular `tf.data.Dataset`. Note that when batching a dataset with a variable shape, you need to use `Dataset.padded_batch`.

In [ ]:

```
ds_series_batch = ds_series.shuffle(20).padded_batch(10)

ids, sequence_batch = next(iter(ds_series_batch))
print(ids.numpy())
print()
print(sequence_batch.numpy())
```

For a more realistic example, try wrapping `preprocessing.image.ImageDataGenerator` as a `tf.data.Dataset`.

First download the data:

In [ ]:

```
flowers = tf.keras.utils.get_file(
    'flower_photos',
    'https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz',
    untar=True)
```

Create the `image.ImageDataGenerator`

In [ ]:

```
img_gen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255, rotation_range=20)
```

In [ ]:

```
images, labels = next(img_gen.flow_from_directory(flowers))
```

In [ ]:

```
print(images.dtype, images.shape)
print(labels.dtype, labels.shape)
```

```
In [ ]:
```

```
ds = tf.data.Dataset.from_generator(  
    lambda: img_gen.flow_from_directory(flowers),  
    output_types=(tf.float32, tf.float32),  
    output_shapes=([32,256,256,3], [32,5])  
)  
  
ds.element_spec
```

```
In [ ]:
```

```
for images, labels in ds.take(1):  
    print('images.shape: ', images.shape)  
    print('labels.shape: ', labels.shape)
```

## Consuming TFRecord data

See [Loading TFRecords \(./tutorials/load\\_data/tfrecord.ipynb\)](#) for an end-to-end example.

The `tf.data` API supports a variety of file formats so that you can process large datasets that do not fit in memory. For example, the TFRecord file format is a simple record-oriented binary format that many TensorFlow applications use for training data. The `tf.data.TFRecordDataset` class enables you to stream over the contents of one or more TFRecord files as part of an input pipeline.

Here is an example using the test file from the French Street Name Signs (FSNS).

```
In [ ]:
```

```
# Creates a dataset that reads all of the examples from two files.  
fsns_test_file = tf.keras.utils.get_file("fsns_tfrec", "https://storage.googleapis.com/download.tensorflow.org/data/fsns-20160927/testdata/fsns-00000-of-00001")
```

The `filenames` argument to the `TFRecordDataset` initializer can either be a string, a list of strings, or a `tf.Tensor` of strings. Therefore if you have two sets of files for training and validation purposes, you can create a factory method that produces the dataset, taking filenames as an input argument:

```
In [ ]:
```

```
dataset = tf.data.TFRecordDataset(filenames = [fsns_test_file])  
dataset
```

Many TensorFlow projects use serialized `tf.train.Example` records in their TFRecord files. These need to be decoded before they can be inspected:

```
In [ ]:
```

```
raw_example = next(iter(dataset))  
parsed = tf.train.Example.FromString(raw_example.numpy())  
  
parsed.features.feature['image/text']
```

## Consuming text data

See [Loading Text \(./tutorials/load\\_data/text.ipynb\)](#) for an end to end example.

Many datasets are distributed as one or more text files. The `tf.data.TextLineDataset` provides an easy way to extract lines from one or more text files. Given one or more filenames, a `TextLineDataset` will produce one string-valued element per line of those files.

```
In [ ]:
```

```
directory_url = 'https://storage.googleapis.com/download.tensorflow.org/data/illiad/'  
file_names = ['cowper.txt', 'derby.txt', 'butler.txt']  
  
file_paths = [  
    tf.keras.utils.get_file(file_name, directory_url + file_name)  
    for file_name in file_names  
]
```

```
In [ ]:
```

```
dataset = tf.data.TextLineDataset(file_paths)
```

Here are the first few lines of the first file:

```
In [ ]:
```

```
for line in dataset.take(5):
    print(line.numpy())
```

To alternate lines between files use `Dataset.interleave`. This makes it easier to shuffle files together. Here are the first, second and third lines from each translation:

```
In [ ]:
```

```
files_ds = tf.data.Dataset.from_tensor_slices(file_paths)
lines_ds = files_ds.interleave(tf.data.TextLineDataset, cycle_length=3)

for i, line in enumerate(lines_ds.take(9)):
    if i % 3 == 0:
        print()
    print(line.numpy())
```

By default, a `TextLineDataset` yields every line of each file, which may not be desirable, for example, if the file starts with a header line, or contains comments. These lines can be removed using the `Dataset.skip()` or `Dataset.filter()` transformations. Here, you skip the first line, then filter to find only survivors.

```
In [ ]:
```

```
titanic_file = tf.keras.utils.get_file("train.csv", "https://storage.googleapis.com/tf-datasets/titanic/train.csv")
titanic_lines = tf.data.TextLineDataset(titanic_file)
```

```
In [ ]:
```

```
for line in titanic_lines.take(10):
    print(line.numpy())
```

```
In [ ]:
```

```
def survived(line):
    return tf.not_equal(tf.strings.substr(line, 0, 1), "0")

survivors = titanic_lines.skip(1).filter(survived)
```

```
In [ ]:
```

```
for line in survivors.take(10):
    print(line.numpy())
```

## Consuming CSV data

See [Loading CSV Files](#) (`..../tutorials/load_data/csv.ipynb`), and [Loading Pandas DataFrames](#) (`..../tutorials/load_data/pandas_dataframe.ipynb`) for more examples.

The CSV file format is a popular format for storing tabular data in plain text.

For example:

```
In [ ]:
```

```
titanic_file = tf.keras.utils.get_file("train.csv", "https://storage.googleapis.com/tf-datasets/titanic/train.csv")
```

```
In [ ]:
```

```
df = pd.read_csv(titanic_file)
df.head()
```

If your data fits in memory the same `Dataset.from_tensor_slices` method works on dictionaries, allowing this data to be easily imported:

```
In [ ]:
```

```
titanic_slices = tf.data.Dataset.from_tensor_slices(dict(df))

for feature_batch in titanic_slices.take(1):
    for key, value in feature_batch.items():
        print("  {!r:20s}: {}".format(key, value))
```

A more scalable approach is to load from disk as necessary.

The `tf.data` module provides methods to extract records from one or more CSV files that comply with [RFC 4180](https://tools.ietf.org/html/rfc4180) (<https://tools.ietf.org/html/rfc4180>).

The `experimental.make_csv_dataset` function is the high level interface for reading sets of csv files. It supports column type inference and many other features, like batching and shuffling, to make usage simple.

In [ ]:

```
titanic_batches = tf.data.experimental.make_csv_dataset(  
    titanic_file, batch_size=4,  
    label_name="survived")
```

In [ ]:

```
for feature_batch, label_batch in titanic_batches.take(1):  
    print("survived": {}).format(label_batch))  
    print("features:")  
    for key, value in feature_batch.items():  
        print("  {!r:20s}: {}".format(key, value))
```

You can use the `select_columns` argument if you only need a subset of columns.

In [ ]:

```
titanic_batches = tf.data.experimental.make_csv_dataset(  
    titanic_file, batch_size=4,  
    label_name="survived", select_columns=['class', 'fare', 'survived'])
```

In [ ]:

```
for feature_batch, label_batch in titanic_batches.take(1):  
    print("survived": {}).format(label_batch))  
    for key, value in feature_batch.items():  
        print("  {!r:20s}: {}".format(key, value))
```

There is also a lower-level `experimental.CsvDataset` class which provides finer grained control. It does not support column type inference. Instead you must specify the type of each column.

In [ ]:

```
titanic_types = [tf.int32, tf.string, tf.float32, tf.int32, tf.int32, tf.float32, tf.string, tf.string, tf.string]  
dataset = tf.data.experimental.CsvDataset(titanic_file, titanic_types, header=True)  
  
for line in dataset.take(10):  
    print([item.numpy() for item in line])
```

If some columns are empty, this low-level interface allows you to provide default values instead of column types.

In [ ]:

```
%writefile missing.csv  
1,2,3,4  
,2,3,4  
1,,3,4  
1,2,,,4  
1,2,3,  
,,
```

In [ ]:

```
# Creates a dataset that reads all of the records from two CSV files, each with  
# four float columns which may have missing values.
```

```
record_defaults = [999, 999, 999, 999]  
dataset = tf.data.experimental.CsvDataset("missing.csv", record_defaults)  
dataset = dataset.map(lambda *items: tf.stack(items))  
dataset
```

In [ ]:

```
for line in dataset:  
    print(line.numpy())
```

By default, a `CsvDataset` yields every column of every line of the file, which may not be desirable, for example if the file starts with a header line that should be ignored, or if some columns are not required in the input. These lines and fields can be removed with the `header` and `select_cols` arguments respectively.

In [ ]:

```
# Creates a dataset that reads all of the records from two CSV files with
# headers, extracting float data from columns 2 and 4.
record_defaults = [999, 999] # Only provide defaults for the selected columns
dataset = tf.data.experimental.CsvDataset("missing.csv", record_defaults, select_cols=[1, 3])
dataset = dataset.map(lambda *items: tf.stack(items))
dataset
```

In [ ]:

```
for line in dataset:
    print(line.numpy())
```

## Consuming sets of files

There are many datasets distributed as a set of files, where each file is an example.

In [ ]:

```
flowers_root = tf.keras.utils.get_file(
    'flower_photos',
    'https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz',
    untar=True)
flowers_root = pathlib.Path(flowers_root)
```

Note: these images are licensed CC-BY, see LICENSE.txt for details.

The root directory contains a directory for each class:

In [ ]:

```
for item in flowers_root.glob("*"):
    print(item.name)
```

The files in each class directory are examples:

In [ ]:

```
list_ds = tf.data.Dataset.list_files(str(flowers_root/'*/*'))

for f in list_ds.take(5):
    print(f.numpy())
```

Read the data using the `tf.io.read_file` function and extract the label from the path, returning `(image, label)` pairs:

In [ ]:

```
def process_path(file_path):
    label = tf.strings.split(file_path, os.sep)[-2]
    return tf.io.read_file(file_path), label

labeled_ds = list_ds.map(process_path)
```

In [ ]:

```
for image_raw, label_text in labeled_ds.take(1):
    print(repr(image_raw.numpy()[:100]))
    print()
    print(label_text.numpy())
```

## Batching dataset elements

## Simple batching

The simplest form of batching stacks  $n$  consecutive elements of a dataset into a single element. The `Dataset.batch()` transformation does exactly this, with the same constraints as the `tf.stack()` operator, applied to each component of the elements: i.e. for each component  $i$ , all elements must have a tensor of the exact same shape.

In [ ]:

```
inc_dataset = tf.data.Dataset.range(100)
dec_dataset = tf.data.Dataset.range(0, -100, -1)
dataset = tf.data.Dataset.zip((inc_dataset, dec_dataset))
batched_dataset = dataset.batch(4)

for batch in batched_dataset.take(4):
    print([arr.numpy() for arr in batch])
```

While `tf.data` tries to propagate shape information, the default settings of `Dataset.batch` result in an unknown batch size because the last batch may not be full. Note the `None`s in the shape:

In [ ]:

```
batched_dataset
```

Use the `drop_remainder` argument to ignore that last batch, and get full shape propagation:

In [ ]:

```
batched_dataset = dataset.batch(7, drop_remainder=True)
batched_dataset
```

## Batching tensors with padding

The above recipe works for tensors that all have the same size. However, many models (e.g. sequence models) work with input data that can have varying size (e.g. sequences of different lengths). To handle this case, the `Dataset.padded_batch` transformation enables you to batch tensors of different shape by specifying one or more dimensions in which they may be padded.

In [ ]:

```
dataset = tf.data.Dataset.range(100)
dataset = dataset.map(lambda x: tf.fill([tf.cast(x, tf.int32)], x))
dataset = dataset.padded_batch(4, padded_shapes=(None,))

for batch in dataset.take(2):
    print(batch.numpy())
    print()
```

The `Dataset.padded_batch` transformation allows you to set different padding for each dimension of each component, and it may be variable-length (signified by `None` in the example above) or constant-length. It is also possible to override the padding value, which defaults to 0.

## Training workflows

### Processing multiple epochs

The `tf.data` API offers two main ways to process multiple epochs of the same data.

The simplest way to iterate over a dataset in multiple epochs is to use the `Dataset.repeat()` transformation. First, create a dataset of titanic data:

In [ ]:

```
titanic_file = tf.keras.utils.get_file("train.csv", "https://storage.googleapis.com/tf-datasets/titanic/train.csv")
titanic_lines = tf.data.TextLineDataset(titanic_file)
```

In [ ]:

```
def plot_batch_sizes(ds):
    batch_sizes = [batch.shape[0] for batch in ds]
    plt.bar(range(len(batch_sizes)), batch_sizes)
    plt.xlabel('Batch number')
    plt.ylabel('Batch size')
```

Applying the `Dataset.repeat()` transformation with no arguments will repeat the input indefinitely.

The `Dataset.repeat` transformation concatenates its arguments without signaling the end of one epoch and the beginning of the next epoch. Because of this a `Dataset.batch` applied after `Dataset.repeat` will yield batches that straddle epoch boundaries:

In [ ]:

```
titanic_batches = titanic_lines.repeat(3).batch(128)
plot_batch_sizes(titanic_batches)
```

If you need clear epoch separation, put `Dataset.batch` before the repeat:

In [ ]:

```
titanic_batches = titanic_lines.batch(128).repeat(3)
plot_batch_sizes(titanic_batches)
```

If you would like to perform a custom computation (e.g. to collect statistics) at the end of each epoch then it's simplest to restart the dataset iteration on each epoch:

In [ ]:

```
epochs = 3
dataset = titanic_lines.batch(128)

for epoch in range(epochs):
    for batch in dataset:
        print(batch.shape)
    print("End of epoch: ", epoch)
```

## Randomly shuffling input data

The `Dataset.shuffle()` transformation maintains a fixed-size buffer and chooses the next element uniformly at random from that buffer.

Note: While large `buffer_sizes` shuffle more thoroughly, they can take a lot of memory, and significant time to fill. Consider using `Dataset.interleave` across files if this becomes a problem.

Add an index to the dataset so you can see the effect:

In [ ]:

```
lines = tf.data.TextLineDataset(titanic_file)
counter = tf.data.experimental.Counter()

dataset = tf.data.Dataset.zip((counter, lines))
dataset = dataset.shuffle(buffer_size=100)
dataset = dataset.batch(20)
dataset
```

Since the `buffer_size` is 100, and the batch size is 20, the first batch contains no elements with an index over 120.

In [ ]:

```
n,line_batch = next(iter(dataset))
print(n.numpy())
```

As with `Dataset.batch` the order relative to `Dataset.repeat` matters.

`Dataset.shuffle` doesn't signal the end of an epoch until the shuffle buffer is empty. So a shuffle placed before a repeat will show every element of one epoch before moving to the next:

In [ ]:

```
dataset = tf.data.Dataset.zip((counter, lines))
shuffled = dataset.shuffle(buffer_size=100).batch(10).repeat(2)

print("Here are the item ID's near the epoch boundary:\n")
for n, line_batch in shuffled.skip(60).take(5):
    print(n.numpy())
```

In [ ]:

```
shuffle_repeat = [n.numpy().mean() for n, line_batch in shuffled]
plt.plot(shuffle_repeat, label="shuffle().repeat()")
plt.ylabel("Mean item ID")
plt.legend()
```

But a repeat before a shuffle mixes the epoch boundaries together:

In [ ]:

```
dataset = tf.data.Dataset.zip((counter, lines))
shuffled = dataset.repeat(2).shuffle(buffer_size=100).batch(10)

print("Here are the item ID's near the epoch boundary:\n")
for n, line_batch in shuffled.skip(55).take(15):
    print(n.numpy())
```

In [ ]:

```
repeat_shuffle = [n.numpy().mean() for n, line_batch in shuffled]

plt.plot(shuffle_repeat, label="shuffle().repeat()")
plt.plot(repeat_shuffle, label="repeat().shuffle()")
plt.ylabel("Mean item ID")
plt.legend()
```

## Preprocessing data

The `Dataset.map(f)` transformation produces a new dataset by applying a given function `f` to each element of the input dataset. It is based on the `map()` ([https://en.wikipedia.org/wiki/Map\\_\(higher-order\\_function\)](https://en.wikipedia.org/wiki/Map_(higher-order_function))) function that is commonly applied to lists (and other structures) in functional programming languages. The function `f` takes the `tf.Tensor` objects that represent a single element in the input, and returns the `tf.Tensor` objects that will represent a single element in the new dataset. Its implementation uses standard TensorFlow operations to transform one element into another.

This section covers common examples of how to use `Dataset.map()`.

### Decoding image data and resizing it

When training a neural network on real-world image data, it is often necessary to convert images of different sizes to a common size, so that they may be batched into a fixed size.

Rebuild the flower filenames dataset:

In [ ]:

```
list_ds = tf.data.Dataset.list_files(str(flowers_root/'*/*'))
```

Write a function that manipulates the dataset elements.

In [ ]:

```
# Reads an image from a file, decodes it into a dense tensor, and resizes it
# to a fixed shape.
def parse_image(filename):
    parts = tf.strings.split(filename, os.sep)
    label = parts[-2]

    image = tf.io.read_file(filename)
    image = tf.io.decode_jpeg(image)
    image = tf.image.convert_image_dtype(image, tf.float32)
    image = tf.image.resize(image, [128, 128])
    return image, label
```

Test that it works.

In [ ]:

```
file_path = next(iter(list_ds))
image, label = parse_image(file_path)

def show(image, label):
    plt.figure()
    plt.imshow(image)
    plt.title(label.numpy().decode('utf-8'))
    plt.axis('off')

show(image, label)
```

Map it over the dataset.

In [ ]:

```
images_ds = list_ds.map(parse_image)

for image, label in images_ds.take(2):
    show(image, label)
```

## Applying arbitrary Python logic

For performance reasons, use TensorFlow operations for preprocessing your data whenever possible. However, it is sometimes useful to call external Python libraries when parsing your input data. You can use the `tf.py_function()` operation in a `Dataset.map()` transformation.

For example, if you want to apply a random rotation, the `tf.image` module only has `tf.image.rot90`, which is not very useful for image augmentation.

Note: `tensorflow_addons` has a TensorFlow compatible `rotate` in `tensorflow_addons.image.rotate`.

To demonstrate `tf.py_function`, try using the `scipy.ndimage.rotate` function instead:

In [ ]:

```
import scipy.ndimage as ndimage

def random_rotate_image(image):
    image = ndimage.rotate(image, np.random.uniform(-30, 30), reshape=False)
    return image
```

In [ ]:

```
image, label = next(iter(images_ds))
image = random_rotate_image(image)
show(image, label)
```

To use this function with `Dataset.map` the same caveats apply as with `Dataset.from_generator`, you need to describe the return shapes and types when you apply the function:

In [ ]:

```
def tf_random_rotate_image(image, label):
    im_shape = image.shape
    [image,] = tf.py_function(random_rotate_image, [image], [tf.float32])
    image.set_shape(im_shape)
    return image, label
```

In [ ]:

```
rot_ds = images_ds.map(tf_random_rotate_image)

for image, label in rot_ds.take(2):
    show(image, label)
```

## Parsing `tf.Example` protocol buffer messages

Many input pipelines extract `tf.train.Example` protocol buffer messages from a TFRecord format. Each `tf.train.Example` record contains one or more "features", and the input pipeline typically converts these features into tensors.

```
In [ ]:
```

```
fsns_test_file = tf.keras.utils.get_file("fsns_tfrec", "https://storage.googleapis.com/download.tensorflow.org/data/fsns-20160927/testdata/fsns-00000-of-00001")
dataset = tf.data.TFRecordDataset(filenames = [fsns_test_file])
dataset
```

You can work with `tf.train.Example` protos outside of a `tf.data.Dataset` to understand the data:

```
In [ ]:
```

```
raw_example = next(iter(dataset))
parsed = tf.train.Example.FromString(raw_example.numpy())

feature = parsed.features.feature
raw_img = feature['image/encoded'].bytes_list.value[0]
img = tf.image.decode_png(raw_img)
plt.imshow(img)
plt.axis('off')
_ = plt.title(feature["image/text"].bytes_list.value[0])
```

```
In [ ]:
```

```
raw_example = next(iter(dataset))
```

```
In [ ]:
```

```
def tf_parse(eg):
    example = tf.io.parse_example(
        eg[tf.newaxis], {
            'image/encoded': tf.io.FixedLenFeature(shape=(), dtype=tf.string),
            'image/text': tf.io.FixedLenFeature(shape=(), dtype=tf.string)
        })
    return example['image/encoded'][0], example['image/text'][0]
```

```
In [ ]:
```

```
img, txt = tf_parse(raw_example)
print(txt.numpy())
print(repr(img.numpy()[:20]), "...")
```

```
In [ ]:
```

```
decoded = dataset.map(tf_parse)
decoded
```

```
In [ ]:
```

```
image_batch, text_batch = next(iter(decoded.batch(10)))
image_batch.shape
```

## Time series windowing

For an end to end time series example see: [Time series forecasting \(../../tutorials/structured\\_data/time\\_series.ipynb\)](#).

Time series data is often organized with the time axis intact.

Use a simple `Dataset.range` to demonstrate:

```
In [ ]:
```

```
range_ds = tf.data.Dataset.range(100000)
```

Typically, models based on this sort of data will want a contiguous time slice.

The simplest approach would be to batch the data:

## Using batch

In [ ]:

```
batchs = range_ds.batch(10, drop_remainder=True)

for batch in batchs.take(5):
    print(batch.numpy())
```

Or to make dense predictions one step into the future, you might shift the features and labels by one step relative to each other:

In [ ]:

```
def dense_1_step(batch):
    # Shift features and labels one step relative to each other.
    return batch[:-1], batch[1:]

predict_dense_1_step = batchs.map(dense_1_step)

for features, label in predict_dense_1_step.take(3):
    print(features.numpy(), "=>", label.numpy())
```

To predict a whole window instead of a fixed offset you can split the batches into two parts:

In [ ]:

```
batchs = range_ds.batch(15, drop_remainder=True)

def label_next_5_steps(batch):
    return (batch[:-5],      # Inputs: All except the last 5 steps
            batch[-5:])    # Labels: The last 5 steps

predict_5_steps = batchs.map(label_next_5_steps)

for features, label in predict_5_steps.take(3):
    print(features.numpy(), "=>", label.numpy())
```

To allow some overlap between the features of one batch and the labels of another, use `Dataset.zip`:

In [ ]:

```
feature_length = 10
label_length = 3

features = range_ds.batch(feature_length, drop_remainder=True)
labels = range_ds.batch(feature_length).skip(1).map(lambda labels: labels[:label_length])

predicted_steps = tf.data.Dataset.zip((features, labels))

for features, label in predicted_steps.take(5):
    print(features.numpy(), "=>", label.numpy())
```

## Using window

While using `Dataset.batch` works, there are situations where you may need finer control. The `Dataset.window` method gives you complete control, but requires some care: it returns a `Dataset of Datasets`. See [Dataset structure](#) for details.

In [ ]:

```
window_size = 5

windows = range_ds.window(window_size, shift=1)
for sub_ds in windows.take(5):
    print(sub_ds)
```

The `Dataset.flat_map` method can take a dataset of datasets and flatten it into a single dataset:

In [ ]:

```
for x in windows.flat_map(lambda x: x).take(30):
    print(x.numpy(), end=' ')
```

In nearly all cases, you will want to `.batch` the dataset first:

```
In [ ]:
```

```
def sub_to_batch(sub):
    return sub.batch(window_size, drop_remainder=True)

for example in windows.flat_map(sub_to_batch).take(5):
    print(example.numpy())
```

Now, you can see that the `shift` argument controls how much each window moves over.

Putting this together you might write this function:

```
In [ ]:
```

```
def make_window_dataset(ds, window_size=5, shift=1, stride=1):
    windows = ds.window(window_size, shift=shift, stride=stride)

    def sub_to_batch(sub):
        return sub.batch(window_size, drop_remainder=True)

    windows = windows.flat_map(sub_to_batch)
    return windows
```

```
In [ ]:
```

```
ds = make_window_dataset(range_ds, window_size=10, shift = 5, stride=3)

for example in ds.take(10):
    print(example.numpy())
```

Then it's easy to extract labels, as before:

```
In [ ]:
```

```
dense_labels_ds = ds.map(dense_1_step)

for inputs,labels in dense_labels_ds.take(3):
    print(inputs.numpy(), "=>", labels.numpy())
```

## Resampling

When working with a dataset that is very class-imbalanced, you may want to resample the dataset. `tf.data` provides two methods to do this. The credit card fraud dataset is a good example of this sort of problem.

Note: See [Imbalanced Data \(./tutorials/keras/imbalanced\\_data.ipynb\)](#) for a full tutorial.

```
In [ ]:
```

```
zip_path = tf.keras.utils.get_file(
    origin='https://storage.googleapis.com/download.tensorflow.org/data/creditcard.zip',
    fname='creditcard.zip',
    extract=True)

csv_path = zip_path.replace('.zip', '.csv')
```

```
In [ ]:
```

```
creditcard_ds = tf.data.experimental.make_csv_dataset(
    csv_path, batch_size=1024, label_name="Class",
    # Set the column types: 30 floats and an int.
    column_defaults=[float()]*30+[int()])
```

Now, check the distribution of classes, it is highly skewed:

In [ ]:

```
def count(counts, batch):
    features, labels = batch
    class_1 = labels == 1
    class_1 = tf.cast(class_1, tf.int32)

    class_0 = labels == 0
    class_0 = tf.cast(class_0, tf.int32)

    counts['class_0'] += tf.reduce_sum(class_0)
    counts['class_1'] += tf.reduce_sum(class_1)

    return counts
```

In [ ]:

```
counts = creditcard_ds.take(10).reduce(
    initial_state={'class_0': 0, 'class_1': 0},
    reduce_func = count)

counts = np.array([counts['class_0'].numpy(),
                  counts['class_1'].numpy()]).astype(np.float32)

fractions = counts/counts.sum()
print(fractions)
```

A common approach to training with an imbalanced dataset is to balance it. `tf.data` includes a few methods which enable this workflow:

## Datasets sampling

One approach to resampling a dataset is to use `sample_from_datasets`. This is more applicable when you have a separate `data.Dataset` for each class.

Here, just use filter to generate them from the credit card fraud data:

In [ ]:

```
negative_ds = (
    creditcard_ds
    .unbatch()
    .filter(lambda features, label: label==0)
    .repeat())
positive_ds = (
    creditcard_ds
    .unbatch()
    .filter(lambda features, label: label==1)
    .repeat())
```

In [ ]:

```
for features, label in positive_ds.batch(10).take(1):
    print(label.numpy())
```

To use `tf.data.Dataset.sample_from_datasets` pass the datasets, and the weight for each:

In [ ]:

```
balanced_ds = tf.data.Dataset.sample_from_datasets(
    [negative_ds, positive_ds], [0.5, 0.5]).batch(10)
```

Now the dataset produces examples of each class with 50/50 probability:

In [ ]:

```
for features, labels in balanced_ds.take(10):
    print(labels.numpy())
```

## Rejection resampling

One problem with the above `Dataset.sample_from_datasets` approach is that it needs a separate `tf.data.Dataset` per class. You could use `Dataset.filter` to create those two datasets, but that results in all the data being loaded twice.

The `data.Dataset.rejection_resample` method can be applied to a dataset to rebalance it, while only loading it once. Elements will be dropped from the dataset to achieve balance.

`data.Dataset.rejection_resample` takes a `class_func` argument. This `class_func` is applied to each dataset element, and is used to determine which class an example belongs to for the purposes of balancing.

The goal here is to balance the lable distribution, and the elements of `creditcard_ds` are already `(features, label)` pairs. So the `class_func` just needs to return those labels:

In [ ]:

```
def class_func(features, label):
    return label
```

The resampling method deals with individual examples, so in this case you must `unbatch` the dataset before applying that method.

The method needs a target distribution, and optionally an initial distribution estimate as inputs.

In [ ]:

```
resample_ds = (
    creditcard_ds
    .unbatch()
    .rejection_resample(class_func, target_dist=[0.5, 0.5],
                         initial_dist=fractions)
    .batch(10))
```

The `rejection_resample` method returns `(class, example)` pairs where the `class` is the output of the `class_func`. In this case, the example was already a `(feature, label)` pair, so use `map` to drop the extra copy of the labels:

In [ ]:

```
balanced_ds = resample_ds.map(lambda extra_label, features_and_label: features_and_label)
```

Now the dataset produces examples of each class with 50/50 probability:

In [ ]:

```
for features, labels in balanced_ds.take(10):
    print(labels.numpy())
```

## Iterator Checkpointing

Tensorflow supports [taking checkpoints](https://www.tensorflow.org/guide/checkpoint) (<https://www.tensorflow.org/guide/checkpoint>) so that when your training process restarts it can restore the latest checkpoint to recover most of its progress. In addition to checkpointing the model variables, you can also checkpoint the progress of the dataset iterator. This could be useful if you have a large dataset and don't want to start the dataset from the beginning on each restart. Note however that iterator checkpoints may be large, since transformations such as `shuffle` and `prefetch` require buffering elements within the iterator.

To include your iterator in a checkpoint, pass the iterator to the `tf.train.Checkpoint` constructor.

In [ ]:

```
range_ds = tf.data.Dataset.range(20)

iterator = iter(range_ds)
ckpt = tf.train.Checkpoint(step=tf.Variable(0), iterator=iterator)
manager = tf.train.CheckpointManager(ckpt, '/tmp/my_ckpt', max_to_keep=3)

print([next(iterator).numpy() for _ in range(5)])

save_path = manager.save()

print([next(iterator).numpy() for _ in range(5)])
ckpt.restore(manager.latest_checkpoint)

print([next(iterator).numpy() for _ in range(5)])
```

Note: It is not possible to checkpoint an iterator which relies on external state such as a `tf.py_function`. Attempting to do so will raise an exception complaining about the external state.

## Using tf.data with tf.keras

The `tf.keras` API simplifies many aspects of creating and executing machine learning models. Its `.fit()` and `.evaluate()` and `.predict()` APIs support datasets as inputs. Here is a quick dataset and model setup:

In [ ]:

```
train, test = tf.keras.datasets.fashion_mnist.load_data()  
  
images, labels = train  
images = images/255.0  
labels = labels.astype(np.int32)
```

In [ ]:

```
fashion_train_ds = tf.data.Dataset.from_tensor_slices((images, labels))  
fashion_train_ds = fashion_train_ds.shuffle(5000).batch(32)  
  
model = tf.keras.Sequential([  
    tf.keras.layers.Flatten(),  
    tf.keras.layers.Dense(10)  
])  
  
model.compile(optimizer='adam',  
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),  
              metrics=['accuracy'])
```

Passing a dataset of `(feature, label)` pairs is all that's needed for `Model.fit` and `Model.evaluate`:

In [ ]:

```
model.fit(fashion_train_ds, epochs=2)
```

If you pass an infinite dataset, for example by calling `Dataset.repeat()`, you just need to also pass the `steps_per_epoch` argument:

In [ ]:

```
model.fit(fashion_train_ds.repeat(), epochs=2, steps_per_epoch=20)
```

For evaluation you can pass the number of evaluation steps:

In [ ]:

```
loss, accuracy = model.evaluate(fashion_train_ds)  
print("Loss :", loss)  
print("Accuracy :", accuracy)
```

For long datasets, set the number of steps to evaluate:

In [ ]:

```
loss, accuracy = model.evaluate(fashion_train_ds.repeat(), steps=10)  
print("Loss :", loss)  
print("Accuracy :", accuracy)
```

The labels are not required in when calling `Model.predict`.

In [ ]:

```
predict_ds = tf.data.Dataset.from_tensor_slices(images).batch(32)  
result = model.predict(predict_ds, steps = 10)  
print(result.shape)
```

But the labels are ignored if you do pass a dataset containing them:

In [ ]:

```
result = model.predict(fashion_train_ds, steps = 10)  
print(result.shape)
```

Welcome to the warp zone!

# Keras

# TF1.x -> TF2 migration overview

TensorFlow 2 is fundamentally different from TF1.x in several ways. You can still run unmodified TF1.x code ([except for contrib](https://github.com/tensorflow/community/blob/master/rfcs/20180907-contrib-sunset.md) (<https://github.com/tensorflow/community/blob/master/rfcs/20180907-contrib-sunset.md>)) against TF2 binary installations like so:

```
python  
import tensorflow.compat.v1 as tf  
tf.disable_v2_behavior()
```

However, this is *not* running TF2 behaviors and APIs, and may not work as expected with code written for TF2. If you are not running with TF2 behaviors active, you are effectively running TF1.x on top of a TF2 installation. Read the [TF1 vs TF2 behaviors guide](#) ([./tf1\\_vs\\_tf2.ipynb](#)) for more details on how TF2 is different from TF1.x.

This guide provides an overview of the process to migrate your TF1.x code to TF2. This enables you to take advantage of new and future feature improvements and also make your code simpler, more performant, and easier to maintain.

If you are using `tf.keras`'s high level APIs and training exclusively with `model.fit`, your code should more or less be fully compatible with TF2 except for the following caveats:

- TF2 has new default learning rates ([./guide/effective\\_tf2.ipynb#optimizer\\_defaults](#)) for Keras optimizers.
- TF2 [may have changed](#) ([./guide/effective\\_tf2.ipynb#keras\\_metric\\_names](#)) the "name" that metrics are logged to.

## TF2 migration process

Before migrating, learn about the behavior and API differences between TF1.x and TF2 by reading the [guide](#) ([./tf1\\_vs\\_tf2.ipynb](#)).

1. Run the automated script to convert some of your TF1.x API usage to `tf.compat.v1`.
2. Remove old `tf.contrib` symbols (check [TF Addons](#) (<https://github.com/tensorflow/addons>) and [TF-Slim](#) (<https://github.com/google-research/tf-slim>)).
3. Make your TF1.x model forward passes run in TF2 with eager execution enabled.
4. Upgrade your TF1.x code for training loops and saving/loading models to TF2 equivalents.
5. (Optional) Migrate your TF2-compatible `tf.compat.v1` APIs to idiomatic TF2 APIs.

The following sections expand upon the steps outlined above.

## Run the symbol conversion script

This executes an initial pass at rewriting your code symbols to run against TF 2.x binaries, but won't make your code idiomatic to TF 2.x nor will it automatically make your code compatible with TF2 behaviors.

Your code will most likely still make use of `tf.compat.v1` endpoints to access placeholders, sessions, collections, and other TF1.x-style functionality.

Read the [guide](#) ([./upgrade.ipynb](#)) to find out more about the best practices for using the symbol conversion script.

## Remove usage of `tf.contrib`

The `tf.contrib` module has been sunsetting and several of its submodules have been integrated into the core TF2 API. The other submodules are now spun-off into other projects like [TF IO](#) (<https://github.com/tensorflow/io>) and [TF Addons](#) (<https://www.tensorflow.org/addons/overview>).

A large amount of older TF1.x code uses the [Slim](#) (<https://ai.googleblog.com/2016/08/tf-slim-high-level-library-to-define.html>) library, which was packaged with TF1.x as `tf.contrib.layers`. When migrating your Slim code to TF2, switch your Slim API usages to point to the [tf-slim pip package](#) (<https://pypi.org/project/tf-slim/>). Then, read the [model mapping guide](#) ([https://tensorflow.org/guide/migrate/model\\_mapping#a\\_note\\_on\\_slim\\_and\\_contriblayers](https://tensorflow.org/guide/migrate/model_mapping#a_note_on_slim_and_contriblayers)) to learn how to convert Slim code.

Alternatively, if you use Slim pre-trained models you may consider trying out Keras's pre-trained models from `tf.keras.applications` or [TF Hub](#) (<https://tfhub.dev/s?tf-version=tf2&q=slim>)'s TF2 `SavedModel`s exported from the original Slim code.

## Make TF1.x model forward passes run with TF2 behaviors enabled

### Track variables and losses

[TF2 does not support global collections](#). ([./tf1\\_vs\\_tf2.ipynb#no\\_more\\_globals](#))

Eager execution in TF2 does not support `tf.Graph` collection-based APIs. This affects how you construct and track variables.

For new TF2 code you would use `tf.Variable` instead of `v1.get_variable` and use Python objects to collect and track variables instead of `tf.compat.v1.variable_scope`. Typically this would be one of:

- `tf.keras.layers.Layer`
- `tf.keras.Model`
- `tf.Module`

Aggregate lists of variables (like `tf.Graph.get_collection(tf.GraphKeys.VARIABLES)`) with the `.variables` and `.trainable_variables` attributes of the `Layer`, `Module`, or `Model` objects.

The `Layer` and `Model` classes implement several other properties that remove the need for global collections. Their `.losses` property can be a replacement for using the `tf.GraphKeys.LOSSES` collection.

Read the [model mapping guide](#) ([./model\\_mapping.ipynb](#)) to find out more about using the TF2 code modeling shims to embed your existing `get_variable` and `variable_scope` based code inside of `Layers`, `Models`, and `Modules`. This will let you execute forward passes with eager execution enabled without major rewrites.

### Adapting to other behavior changes

If the [model mapping guide](#) ([./model\\_mapping.ipynb](#)) on its own is insufficient to get your model forward pass running other behavior changes that may be more details, see the guide on [TF1.x vs TF2 behaviors](#) ([./tf1\\_vs\\_tf2.ipynb](#)) to learn about the other behavior changes and how you can adapt to them. Also check out the [making new Layers and Models via subclassing guide](#) ([https://tensorflow.org/guide/keras/custom\\_layers\\_and\\_models.ipynb](https://tensorflow.org/guide/keras/custom_layers_and_models.ipynb)) for details.

### Validating your results

See the [model validation guide](#) ([./validate\\_correctness.ipynb](#)) for easy tools and guidance around how you can (numerically) validate that your model is behaving correctly when eager execution is enabled. You may find this especially useful when paired with the [model mapping guide](#) ([./model\\_mapping.ipynb](#)).

### Upgrade training, evaluation, and import/export code

TF1.x training loops built with `tf.Session`-style `tf.estimator.Estimator`s and other collections-based approaches are not compatible with the new behaviors of TF2. It is important you migrate all your TF1.x training code as combining it with TF2 code can cause unexpected behaviors.

You can choose from among several strategies to do this.

The highest-level approach is to use `tf.keras`. The high level functions in Keras manage a lot of the low-level details that might be easy to miss if you write your own training loop. For example, they automatically collect the regularization losses, and set the `training=True` argument when calling the model.

Refer to the [Estimator migration guide \(./migrating\\_estimator.ipynb\)](#) to learn how you can migrate `tf.estimator.Estimator`s code to use [vanilla \(./migrating\\_estimator.ipynb#tf2\\_keras\\_training\\_api\)](#) and [custom \(./migrating\\_estimator.ipynb#tf2\\_keras\\_training\\_api\\_with\\_custom\\_training\\_step\)](#) `tf.keras` training loops.

Custom training loops give you finer control over your model such as tracking the weights of individual layers. Read the guide on [building training loops from scratch \(https://www.tensorflow.org/guide/keras/writing\\_a\\_training\\_loop\\_from\\_scratch\)](#) to learn how to use `tf.GradientTape` to retrieve model weights and use them to update the model.

## Convert TF1.x optimizers to Keras optimizers

The optimizers in `tf.compat.v1.train`, such as the [Adam optimizer \(https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/train/AdamOptimizer\)](#) and the [gradient descent optimizer \(https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/train/GradientDescentOptimizer\)](#), have equivalents in `tf.keras.optimizers`.

The table below summarizes how you can convert these legacy optimizers to their Keras equivalents. You can directly replace the TF1.x version with the TF2 version unless additional steps (such as [updating the default learning rate \(./guide/effective\\_tf2.ipynb#optimizer\\_defaults\)](#)) are required.

Note that converting your optimizers [may make old checkpoints incompatible \(./migrating\\_checkpoints.ipynb\)](#).

TF1.x	TF2	Additional steps
<code>'tf.v1.train.GradientDescentOptimizer'</code>	<code>'tf.keras.optimizers.SGD'</code>	None
<code>'tf.v1.train.MomentumOptimizer'</code>	<code>'tf.keras.optimizers.SGD'</code>	Include the `momentum` argument
<code>'tf.v1.train.AdamOptimizer'</code>	<code>'tf.keras.optimizers.Adam'</code>	Rename `beta1` and `beta2` arguments to `beta_1` and `beta_2`
<code>'tf.v1.train.RMSPropOptimizer'</code>	<code>'tf.keras.optimizers.RMSprop'</code>	Rename the `decay` argument to `rho`
<code>'tf.v1.train.AdadeltaOptimizer'</code>	<code>'tf.keras.optimizers.Adadelta'</code>	None
<code>'tf.v1.train.AdagradOptimizer'</code>	<code>'tf.keras.optimizers.Adagrad'</code>	None
<code>'tf.v1.train.FtrlOptimizer'</code>	<code>'tf.keras.optimizers.Ftrl'</code>	Remove the `accum_name` and `linear_name` arguments
<code>'tf.contrib.AdamaxOptimizer'</code>	<code>'tf.keras.optimizers.Adamax'</code>	Rename the `beta1`, and `beta2` arguments to `beta_1` and `beta_2`
<code>'tf.contrib.Nadam'</code>	<code>'tf.keras.optimizers.Nadam'</code>	Rename the `beta1`, and `beta2` arguments to `beta_1` and `beta_2`

Note: In TF2, all epsilons (numerical stability constants) now default to `1e-7` instead of `1e-8`. This difference is negligible in most use cases.

## Upgrade data input pipelines

There are many ways to feed data to a `tf.keras` model. They will accept Python generators and Numpy arrays as input.

The recommended way to feed data to a model is to use the `tf.data` package, which contains a collection of high performance classes for manipulating data. The `dataset`s belonging to `tf.data` are efficient, expressive, and integrate well with TF2.

They can be passed directly to the `tf.keras.Model.fit` method.

python

```
model.fit(dataset, epochs=5)
```

They can be iterated over directly standard Python:

python

```
for example_batch, label_batch in dataset:  
    break
```

If you are still using `tf.queue`, these are now only supported as data-structures, not as input pipelines.

You should also migrate all feature preprocessing code that uses `tf.feature_columns`. Read the [migration guide \(./migrating\\_feature\\_columns.ipynb\)](#) for more details.

## Saving and loading models

TF2 uses object-based checkpoints. Read the [checkpoint migration guide \(./migrating\\_checkpoints.ipynb\)](#) to learn more about migrating off name-based TF1.x checkpoints. Also read the [checkpoints guide \(https://www.tensorflow.org/guide/checkpoint\)](#) in the core TensorFlow docs.

There are no significant compatibility concerns for saved models. Read the [SavedModel guide \(./saved\\_model.ipynb\)](#) for more information about migrating `SavedModel`s in TF1.x to TF2. In general,

- TF1.x `saved_models` work in TF2.
- TF2 `saved_models` work in TF1.x if all the ops are supported.

Also refer to the [GraphDef section \(./saved\\_model.ipynb#graphdef\\_and\\_metagraphdef\)](#) in the `SavedModel` migration guide for more information on working with `Graph.pb` and `Graph.pbtxt` objects.

## (Optional) Migrate off `tf.compat.v1` symbols

The `tf.compat.v1` module contains the complete TF1.x API, with its original semantics.

Even after following the steps above and ending up with code that is fully compatible with all TF2 behaviors, it is likely there may be many mentions of `compat.v1` apis that happen to be compatible with TF2. You should avoid using these legacy `compat.v1` apis for any new code that you write, though they will continue working for your already-written code.

However, you may choose to migrate the existing usages to non-legacy TF2 APIs. The docstrings of individual `compat.v1` symbols will often explain how to migrate them to non-legacy TF2 APIs. Additionally, the [model mapping guide's section on incremental migration to idiomatic TF2 APIs \(./model\\_mapping.ipynb#incremental\\_migration\\_to\\_native\\_tf2\)](#) may help with this as well.

## Resources and further reading

As mentioned previously, it is a good practice to migrate all your TF1.x code to TF2. Read the guides in the [Migrate to TF2 section \(https://tensorflow.org/guide/migrate\)](#) of the TensorFlow guide to learn more.

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Migrate multi-worker CPU/GPU training



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/multi_worker_cpu_gpu_training)

([https://www.tensorflow.org/guide/migrate/multi\\_worker\\_cpu\\_gpu\\_training](https://www.tensorflow.org/guide/migrate/multi_worker_cpu_gpu_training)) ([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/multi\\_worker\\_cpu\\_gpu\\_training.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/multi_worker_cpu_gpu_training.ipynb))



This guide demonstrates how to migrate your multi-worker distributed training workflow from TensorFlow 1 to TensorFlow 2.

To perform multi-worker training with CPUs/GPUs:

- In TensorFlow 1, you traditionally use the `tf.estimator.train_and_evaluate` and `tf.estimator.Estimator` APIs.
- In TensorFlow 2, use the Keras APIs for writing the model, the loss function, the optimizer, and metrics. Then, distribute the training with Keras `Model.fit` API or a custom training loop (with `tf.GradientTape`) across multiple workers with `tf.distribute.experimental.ParameterServerStrategy` or `tf.distribute.MultiWorkerMirroredStrategy`. For more details, refer to the following tutorials:
  - [Distributed training with TensorFlow](#) ([./guide/distributed\\_training.ipynb](#))
  - [Parameter server training with Keras Model.fit/a custom training loop](#) ([./tutorials/distribute/parameter\\_server\\_training.ipynb](#))
  - [MultiWorkerMirroredStrategy with Keras Model.fit](#) ([./tutorials/distribute/multi\\_worker\\_with\\_keras.ipynb](#))
  - [MultiWorkerMirroredStrategy with a custom training loop](#) ([./tutorials/distribute/multi\\_worker\\_with\\_ctl.ipynb](#)).

## Setup

Start with some necessary imports and a simple dataset for demonstration purposes:

In [ ]:

```
# The notebook uses a dataset instance for `Model.fit` with
# `ParameterServerStrategy`, which depends on symbols in TF 2.7.
# Install a utility needed for this demonstration
!pip install portpicker

import tensorflow as tf
import tensorflow.compat.v1 as tf1
```

In [ ]:

```
features = [[1., 1.5], [2., 2.5], [3., 3.5]]
labels = [[0.3], [0.5], [0.7]]
eval_features = [[4., 4.5], [5., 5.5], [6., 6.5]]
eval_labels = [[0.8], [0.9], [1.]]
```

You will need the `'TF_CONFIG'` configuration environment variable for training on multiple machines in TensorFlow. Use `'TF_CONFIG'` to specify the `'cluster'` and the `'task'` s' addresses. (Learn more in the [Distributed\\_training](#) ([./guide/distributed\\_training.ipynb](#)) guide.)

In [ ]:

```
import json
import os

tf_config = {
    'cluster': {
        'chief': ['localhost:11111'],
        'worker': ['localhost:12345', 'localhost:23456', 'localhost:21212'],
        'ps': ['localhost:12121', 'localhost:13131'],
    },
    'task': {'type': 'chief', 'index': 0}
}

os.environ['TF_CONFIG'] = json.dumps(tf_config)
```

Note: Unfortunately, since multi-worker training with `tf.estimator` APIs in TensorFlow 1 requires multiple clients (which would be especially tricky to be done here in this Colab notebook), you will make the notebook runnable without a '`TF_CONFIG`' environment variable, so it falls back to local training. (Learn more in the *Setting up the '`TF_CONFIG`' environment variable* section in the [Distributed training with TensorFlow \(./guide/distributed\\_training.ipynb\)](#) guide.)

Use the `del` statement to remove the variable (but in real-world multi-worker training in TensorFlow 1, you won't have to do this):

In [ ]:

```
del os.environ['TF_CONFIG']
```

## TensorFlow 1: Multi-worker distributed training with `tf.estimator` APIs

The following code snippet demonstrates the canonical workflow of multi-worker training in TF1: you will use a `tf.estimator.Estimator`, a `tf.estimator.TrainSpec`, a `tf.estimator.EvalSpec`, and the `tf.estimator.train_and_evaluate` API to distribute the training:

In [ ]:

```
def _input_fn():
    return tf1.data.Dataset.from_tensor_slices((features, labels)).batch(1)

def _eval_input_fn():
    return tf1.data.Dataset.from_tensor_slices(
        (eval_features, eval_labels)).batch(1)

def _model_fn(features, labels, mode):
    logits = tf1.layers.Dense(1)(features)
    loss = tf1.losses.mean_squared_error(labels=labels, predictions=logits)
    optimizer = tf1.train.AdagradOptimizer(0.05)
    train_op = optimizer.minimize(loss, global_step=tf1.train.get_global_step())
    return tf1.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)

estimator = tf1.estimator.Estimator(model_fn=_model_fn)
train_spec = tf1.estimator.TrainSpec(input_fn=_input_fn)
eval_spec = tf1.estimator.EvalSpec(input_fn=_eval_input_fn)
tf1.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

## TensorFlow 2: Multi-worker training with distribution strategies

In TensorFlow 2, distributed training across multiple workers with CPUs, GPUs, and TPUs is done via `tf.distribute.Strategy`s.

The following example demonstrates how to use two such strategies: `tf.distribute.experimental.ParameterServerStrategy` and `tf.distribute.MultiWorkerMirroredStrategy`, both of which are designed for CPU/GPU training with multiple workers.

`ParameterServerStrategy` employs a *coordinator* ('chief'), which makes it more friendly with the environment in this Colab notebook. You will be using some utilities here to set up the supporting elements essential for a runnable experience here: you will create an *in-process cluster*, where threads are used to simulate the parameter servers ('ps') and workers ('worker'). For more information about parameter server training, refer to the [Parameter server training with ParameterServerStrategy \(./tutorials/distribute/parameter\\_server\\_training.ipynb\)](#) tutorial.

In this example, first define the '`TF_CONFIG`' environment variable with a `tf.distribute.cluster_resolver.TFConfigClusterResolver` to provide the cluster information. If you are using a cluster management system for your distributed training, check if it provides '`TF_CONFIG`' for you already, in which case you don't need to explicitly set this environment variable. (Learn more in the *Setting up the '`TF_CONFIG`' environment variable* section in the [Distributed training with TensorFlow \(./guide/distributed\\_training.ipynb\)](#) guide.)

In [ ]:

```
# Find ports that are available for the ``chief`` (the coordinator),
# ``worker``'s, and ``ps`` (parameter servers).
import portpicker

chief_port = portpicker.pick_unused_port()
worker_ports = [portpicker.pick_unused_port() for _ in range(3)]
ps_ports = [portpicker.pick_unused_port() for _ in range(2)]

# Dump the cluster information to ``TF_CONFIG``.
tf_config = {
    'cluster': {
        'chief': ["localhost:%s" % chief_port],
        'worker': ["localhost:%s" % port for port in worker_ports],
        'ps': ["localhost:%s" % port for port in ps_ports],
    },
    'task': {'type': 'chief', 'index': 0}
}
os.environ['TF_CONFIG'] = json.dumps(tf_config)

# Use a cluster resolver to bridge the information to the strategy created below.
cluster_resolver = tf.distribute.cluster_resolver.TFConfigClusterResolver()
```

Then, create `tf.distribute.Server`s for the workers and parameter servers one-by-one:

In [ ]:

```
# Workers need some inter_ops threads to work properly.
# This is only needed for this notebook to demo. Real servers
# should not need this.
worker_config = tf.compat.v1.ConfigProto()
worker_config.inter_op_parallelism_threads = 4

for i in range(3):
    tf.distribute.Server(
        cluster_resolver.cluster_spec(),
        job_name="worker",
        task_index=i,
        config=worker_config)

for i in range(2):
    tf.distribute.Server(
        cluster_resolver.cluster_spec(),
        job_name="ps",
        task_index=i)
```

In real-world distributed training, instead of starting all the `tf.distribute.Server`s on the coordinator, you will be using multiple machines, and the ones that are designated as "worker"s and "ps" (parameter servers) will each run a `tf.distribute.Server`. Refer to *Clusters in the real world* section in the [Parameter server training \(../../tutorials/distribute/parameter\\_server\\_training.ipynb\)](#) tutorial for more details.

With everything ready, create the `ParameterServerStrategy` object:

In [ ]:

```
strategy = tf.distribute.experimental.ParameterServerStrategy(cluster_resolver)
```

Once you have created a strategy object, define the model, the optimizer, and other variables, and call the Keras `Model.compile` within the `Strategy.scope` API to distribute the training. (Refer to the `Strategy.scope` API docs for more information.)

If you prefer to customize your training by, for instance, defining the forward and backward passes, refer to *Training with a custom training loop* section in [Parameter server training \(../../tutorials/distribute/parameter\\_server\\_training.ipynb\)](#) tutorial for more details.

In [ ]:

```
dataset = tf.data.Dataset.from_tensor_slices(
    (features, labels)).shuffle(10).repeat().batch(64)

eval_dataset = tf.data.Dataset.from_tensor_slices(
    (eval_features, eval_labels)).repeat().batch(1)

with strategy.scope():
    model = tf.keras.models.Sequential([tf.keras.layers.Dense(1)])
    optimizer = tf.keras.optimizers.Adagrad(learning_rate=0.05)
    model.compile(optimizer, "mse")

model.fit(dataset, epochs=5, steps_per_epoch=10)
```

In [ ]:

```
model.evaluate(eval_dataset, steps=10, return_dict=True)
```

### Partitioners ( `tf.distribute.experimental.partitioners` )

`ParameterServerStrategy` in TensorFlow 2 supports variable partitioning and offers same partitioners as TensorFlow 1, with less confusing names:

- `tf.compat.v1.variable_axis_size_partitioner` -> `tf.distribute.experimental.partitioners.MaxValuePartitioner`: a partitioner that keeps shards under a maximum size).
- `tf.compat.v1.min_max_variable_partitioner` -> `tf.distribute.experimental.partitioners.MinSizePartitioner`: a partitioner that allocates a minimum size per shard.
- `tf.compat.v1.fixed_size_partitioner` -> `tf.distribute.experimental.partitioners.FixedShardsPartitioner`: a partitioner that allocates a fixed number of shards.

Alternatively, you can use a `MultiWorkerMirroredStrategy` object:

In [ ]:

```
# To clean up the `TF_CONFIG` used for `ParameterServerStrategy`.
del os.environ['TF_CONFIG']
strategy = tf.distribute.MultiWorkerMirroredStrategy()
```

You can replace the strategy used above with a `MultiWorkerMirroredStrategy` object to perform training with this strategy.

As with the `tf.estimator` APIs, since `MultiWorkerMirroredStrategy` is a multi-client strategy, there is no easy way to run distributed training in this Colab notebook. Therefore, replacing the code above with this strategy ends up running things locally. The Multi-worker training [with Keras Model.fit](#) ([./tutorials/distribute/multi\\_worker\\_with\\_keras.ipynb](#))/[a custom training loop](#) ([./tutorials/distribute/multi\\_worker\\_with\\_ctl.ipynb](#)) tutorials demonstrate how to run multi-worker training with the '`TF_CONFIG`' variable set up, with two workers on a localhost in Colab. In practice, you would create multiple workers on external IP addresses/ports, and use the '`TF_CONFIG`' variable to specify the cluster configuration for each worker.

## Next steps

To learn more about multi-worker distributed training with `tf.distribute.experimental.ParameterServerStrategy` and `tf.distribute.MultiWorkerMirroredStrategy` in TensorFlow 2, consider the following resources:

- Tutorial: [Parameter server training with ParameterServerStrategy and Keras Model.fit/a custom training loop](#) ([./tutorials/distribute/parameter\\_server\\_training.ipynb](#))
- Tutorial: [Multi-worker training with MultiWorkerMirroredStrategy and Keras Model.fit](#) ([./tutorials/distribute/multi\\_worker\\_with\\_keras.ipynb](#))
- Tutorial: [Multi-worker training with MultiWorkerMirroredStrategy and a custom training loop](#) ([./tutorials/distribute/multi\\_worker\\_with\\_ctl.ipynb](#))
- Guide: [Distributed training with TensorFlow](#) ([./guide/distributed\\_training.ipynb](#))
- Guide: [Optimize TensorFlow GPU performance with the TensorFlow Profiler](#) ([./guide/gpu\\_performance\\_analysis.ipynb](#))
- Guide: [Use a GPU](#) ([./guide/gpu.ipynb](#)) (the Using multiple GPUs section)

**Copyright 2021 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

# Migrate from TPUEstimator to TPUStrategy



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/tpu_estimator)

([https://www.tensorflow.org/guide/migrate/tpu\\_estimator](https://www.tensorflow.org/guide/migrate/tpu_estimator)) ([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/tpu\\_estimator.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/tpu_estimator.ipynb))



[Run in Google Colab](#)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/tpu\\_estimator.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/tpu_estimator.ipynb))

This guide demonstrates how to migrate your workflows running on [TPUs](#) from TensorFlow 1's `TPUEstimator` API to TensorFlow 2's `TPUStrategy` API.

- In TensorFlow 1, the `tf.compat.v1.estimator.tpu.TPUEstimator` API lets you train and evaluate a model, as well as perform inference and save your model (for serving) on (Cloud) TPUs.
- In TensorFlow 2, to perform synchronous training on TPUs and TPU Pods (a collection of TPU devices connected by dedicated high-speed network interfaces), you need to use a TPU distribution strategy—`tf.distribute.TPUStrategy`. The strategy can work with the Keras APIs—including for model building (`tf.keras.Model`), optimizers (`tf.keras.optimizers.Optimizer`), and training (`Model.fit`)—as well as a custom training loop (with `tf.function` and `tf.GradientTape`).

For end-to-end TensorFlow 2 examples, check out the [Use TPUs](#) guide—namely, the *Classification on TPUs* section—and the [Solve GLUE tasks using BERT on TPU](#) tutorial. You may also find the [Distributed training](#) guide useful, which covers all TensorFlow distribution strategies, including `TPUStrategy`.

## Setup

Start with imports and a simple dataset for demonstration purposes:

In [ ]:

```
import tensorflow as tf
import tensorflow.compat.v1 as tf1
```

In [ ]:

```
features = [[1., 1.5]]
labels = [[0.3]]
eval_features = [[4., 4.5]]
eval_labels = [[0.8]]
```

## TensorFlow 1: Drive a model on TPUs with TPUEstimator

This section of the guide demonstrates how to perform training and evaluation with `tf.compat.v1.estimator.tpu.TPUEstimator` in TensorFlow 1.

To use a `TPUEstimator`, first define a few functions: an input function for the training data, an evaluation input function for the evaluation data, and a model function that tells the `TPUEstimator` how the training op is defined with the features and labels:

In [ ]:

```
def _input_fn(params):
    dataset = tf1.data.Dataset.from_tensor_slices((features, labels))
    dataset = dataset.repeat()
    return dataset.batch(params['batch_size'], drop_remainder=True)

def _eval_input_fn(params):
    dataset = tf1.data.Dataset.from_tensor_slices((eval_features, eval_labels))
    dataset = dataset.repeat()
    return dataset.batch(params['batch_size'], drop_remainder=True)

def _model_fn(features, labels, mode, params):
    logits = tf1.layers.Dense(1)(features)
    loss = tf1.losses.mean_squared_error(labels=labels, predictions=logits)
    optimizer = tf1.train.AdagradOptimizer(0.05)
    train_op = optimizer.minimize(loss, global_step=tf1.train.get_global_step())
    return tf1.estimator.tpu.TPUEstimatorSpec(mode, loss=loss, train_op=train_op)
```

With those functions defined, create a `tf.distribute.cluster_resolver.TPUClusterResolver` that provides the cluster information, and a `tf.compat.v1.estimator.tpu.RunConfig` object. Along with the model function you have defined, you can now create a `TPUEstimator`. Here, you will simplify the flow by skipping checkpoint savings. Then, you will specify the batch size for both training and evaluation for the `TPUEstimator`.

```
In [ ]:
```

```
cluster_resolver = tf1.distribute.cluster_resolver.TPUClusterResolver(tpu='')
```

```
print("All devices: ", tf1.config.list_logical_devices('TPU'))
```

```
In [ ]:
```

```
tpu_config = tf1.estimator.tpu.TPUConfig(iterations_per_loop=10)
config = tf1.estimator.tpu.RunConfig(
    cluster=cluster_resolver,
    save_checkpoints_steps=None,
    tpu_config=tpu_config)
estimator = tf1.estimator.tpu.TPUEstimator(
    model_fn=_model_fn,
    config=config,
    train_batch_size=8,
    eval_batch_size=8)
```

Call `TPUEstimator.train` to begin training the model:

```
In [ ]:
```

```
estimator.train(_input_fn, steps=1)
```

Then, call `TPUEstimator.evaluate` to evaluate the model using the evaluation data:

```
In [ ]:
```

```
estimator.evaluate(_eval_input_fn, steps=1)
```

## TensorFlow 2: Drive a model on TPUs with Keras Model.fit and TPUStrategy

In TensorFlow 2, to train on the TPU workers, use `tf.distribute.TPUStrategy` together with the Keras APIs for model definition and training/evaluation. (Refer to the [Use TPUs \(../guide/tpu.ipynb\)](#) guide for more examples of training with Keras `Model.fit` and a custom training loop (with `tf.function` and `tf.GradientTape` ).)

Since you need to perform some initialization work to connect to the remote cluster and initialize the TPU workers, start by creating a `TPUClusterResolver` to provide the cluster information and connect to the cluster. (Learn more in the *TPU initialization* section of the [Use TPUs \(../guide/tpu.ipynb\)](#) guide.)

```
In [ ]:
```

```
cluster_resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='')
tf.config.experimental_connect_to_cluster(cluster_resolver)
tf.tpu.experimental.initialize_tpu_system(cluster_resolver)
print("All devices: ", tf.config.list_logical_devices('TPU'))
```

Next, once your data is prepared, you will create a `TPUStrategy`, define a model, metrics, and an optimizer under the scope of this strategy.

To achieve comparable training speed with `TPUStrategy`, you should make sure to pick a number for `steps_per_execution` in `Model.compile` because it specifies the number of batches to run during each `tf.function` call, and is critical for performance. This argument is similar to `iterations_per_loop` used in a `TPUEstimator`. If you are using custom training loops, you should make sure multiple steps are run within the `tf.function`-ed training function. Go to the *Improving performance with multiple steps inside tf.function* section of the [Use TPUs \(../guide/tpu.ipynb\)](#) guide for more information.

`tf.distribute.TPUStrategy` can support bounded dynamic shapes, which is the case that the upper bound of the dynamic shape computation can be inferred. But dynamic shapes may introduce some performance overhead compared to static shapes. So, it is generally recommended to make your input shapes static if possible, especially in training. One common op that returns a dynamic shape is `tf.data.Dataset.batch(batch_size)`, since the number of samples remaining in a stream might be less than the batch size. Therefore, when training on the TPU, you should use `tf.data.Dataset.batch(..., drop_remainder=True)` for best training performance.

```
In [ ]:
```

```
dataset = tf.data.Dataset.from_tensor_slices(
    (features, labels)).shuffle(10).repeat().batch(
        8, drop_remainder=True).prefetch(2)
eval_dataset = tf.data.Dataset.from_tensor_slices(
    (eval_features, eval_labels)).batch(1, drop_remainder=True)

strategy = tf.distribute.TPUStrategy(cluster_resolver)
with strategy.scope():
    model = tf.keras.models.Sequential([tf.keras.layers.Dense(1)])
    optimizer = tf.keras.optimizers.Adagrad(learning_rate=0.05)
    model.compile(optimizer, "mse", steps_per_execution=10)
```

With that, you are ready to train the model with the training dataset:

In [ ]:

```
model.fit(dataset, epochs=5, steps_per_epoch=10)
```

Finally, evaluate the model using the evaluation dataset:

In [ ]:

```
model.evaluate(eval_dataset, return_dict=True)
```

## Next steps

To learn more about TPUStrategy in TensorFlow 2, consider the following resources:

- Guide: [Use TPUs \(../../guide/tpu.ipynb\)](#) (covering training with Keras Model.fit /a custom training loop with tf.distribute.TPUStrategy , as well as tips on improving the performance with tf.function )
- Guide: [Distributed training with TensorFlow \(../../guide/distributed\\_training.ipynb\)](#)

To learn more about customizing your training, refer to:

- Guide: [Customize what happens in Model.fit \(../../guide/keras/customizing\\_what\\_happens\\_in\\_fit.ipynb\)](#)
- Guide: [Writing a training loop from scratch \(\[https://www.tensorflow.org/guide/keras/writing\\\_a\\\_training\\\_loop\\\_from\\\_scratch\]\(https://www.tensorflow.org/guide/keras/writing\_a\_training\_loop\_from\_scratch\)\)](#)

TPUs—Google's specialized ASICs for machine learning—are available through [Google Colab \(<https://colab.research.google.com/>\)](#), the [TPU Research Cloud \(<https://sites.research.google/trc/>\)](#), and [Cloud TPU \(<https://cloud.google.com/tpu>\)](#).

**Copyright 2021 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Migrate checkpoint saving



[View on TensorFlow.org](#)

([https://www.tensorflow.org/guide/migrate/checkpoint\\_saver](https://www.tensorflow.org/guide/migrate/checkpoint_saver))



[Run in Google Colab](#)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/checkpoint\\_saver.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/checkpoint_saver.ipynb))

Continually saving the "best" model or model weights/parameters has many benefits. These include being able to track the training progress and load saved models from different saved states.

In TensorFlow 1, to configure checkpoint saving during training/validation with the `tf.estimator.Estimator` APIs, you specify a schedule in `tf.estimator.RunConfig` or use `tf.estimator.CheckpointSaverHook`. This guide demonstrates how to migrate from this workflow to TensorFlow 2 Keras APIs.

In TensorFlow 2, you can configure `tf.keras.callbacks.ModelCheckpoint` in a number of ways:

- Save the "best" version according to a metric monitored using the `save_best_only=True` parameter, where `monitor` can be, for example, `'loss'`, `'val_loss'`, `'accuracy'`, or `'val_accuracy'`.
- Save continually at a certain frequency (using the `save_freq` argument).
- Save the weights/parameters only instead of the whole model by setting `save_weights_only` to `True`.

For more details, refer to the `tf.keras.callbacks.ModelCheckpoint` API docs and the *Save checkpoints during training* section in the [Save and load models \(../../tutorials/keras/save\\_and\\_load.ipynb\)](#) tutorial. Learn more about the Checkpoint format in the *TF Checkpoint format* section in the [Save and load Keras models \(https://www.tensorflow.org/guide/keras/save\\_and\\_serialize\)](#) guide. In addition, to add fault tolerance, you can use `tf.keras.callbacks.BackupAndRestore` or `tf.train.Checkpoint` for manual checkpointing. Learn more in the [Fault tolerance migration guide \(fault\\_tolerance.ipynb\)](#).

Keras [callbacks \(https://www.tensorflow.org/guide/keras/custom\\_callback\)](#) are objects that are called at different points during training/evaluation/prediction in the built-in Keras `Model.fit` / `Model.evaluate` / `Model.predict` APIs. Learn more in the *Next steps* section at the end of the guide.

## Setup

Start with imports and a simple dataset for demonstration purposes:

In [ ]:

```
import tensorflow.compat.v1 as tf1
import tensorflow as tf
import numpy as np
import tempfile
```

In [ ]:

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

## TensorFlow 1: Save checkpoints with `tf.estimator` APIs

This TensorFlow 1 example shows how to configure `tf.estimator.RunConfig` to save checkpoints at every step during training/evaluation with the `tf.estimator.Estimator` APIs:

In [ ]:

```
feature_columns = [tf1.feature_column.numeric_column("x", shape=[28, 28])]

config = tf1.estimator.RunConfig(save_summary_steps=1,
                                 save_checkpoints_steps=1)

path = tempfile.mkdtemp()

classifier = tf1.estimator.DNNClassifier(
    feature_columns=feature_columns,
    hidden_units=[256, 32],
    optimizer=tf1.train.AdamOptimizer(0.001),
    n_classes=10,
    dropout=0.2,
    model_dir=path,
    config = config
)

train_input_fn = tf1.estimator.inputs.numpy_input_fn(
    x={"x": x_train},
    y=y_train.astype(np.int32),
    num_epochs=10,
    batch_size=50,
    shuffle=True,
)
test_input_fn = tf1.estimator.inputs.numpy_input_fn(
    x={"x": x_test},
    y=y_test.astype(np.int32),
    num_epochs=10,
    shuffle=False
)
train_spec = tf1.estimator.TrainSpec(input_fn=train_input_fn, max_steps=10)
eval_spec = tf1.estimator.EvalSpec(input_fn=test_input_fn,
                                    steps=10,
                                    throttle_secs=0)

tf1.estimator.train_and_evaluate(estimator=classifier,
                                 train_spec=train_spec,
                                 eval_spec=eval_spec)
```

In [ ]:

```
%ls {classifier.model_dir}
```

## TensorFlow 2: Save checkpoints with a Keras callback for Model.fit

In TensorFlow 2, when you use the built-in Keras `Model.fit` (or `Model.evaluate`) for training/evaluation, you can configure `tf.keras.callbacks.ModelCheckpoint` and then pass it to the `callbacks` parameter of `Model.fit` (or `Model.evaluate`). (Learn more in the API docs and the *Using callbacks* section in the [Training and evaluation with the built-in methods](#) ([https://www.tensorflow.org/guide/keras/train\\_and\\_evaluate](https://www.tensorflow.org/guide/keras/train_and_evaluate)) guide.)

In the example below, you will use a `tf.keras.callbacks.ModelCheckpoint` callback to store checkpoints in a temporary directory:

In [ ]:

```
def create_model():
    return tf.keras.models.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(512, activation='relu'),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10, activation='softmax')
    ])

model = create_model()
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'],
              steps_per_execution=10)

log_dir = tempfile.mkdtemp()

model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=log_dir)

model.fit(x=x_train,
          y=y_train,
          epochs=10,
          validation_data=(x_test, y_test),
          callbacks=[model_checkpoint_callback])
```

In [ ]:

```
%ls {model_checkpoint_callback.filepath}
```

## Next steps

Learn more about checkpointing in:

- API docs: `tf.keras.callbacks.ModelCheckpoint`
- Tutorial: [Save and load models \(./tutorials/keras/save\\_and\\_load.ipynb\)](#) (the *Save checkpoints during training* section)
- Guide: [Save and load Keras models \(https://www.tensorflow.org/guide/keras/save\\_and\\_serialize\)](#) (the *TF Checkpoint format* section)

Learn more about callbacks in:

- API docs: `tf.keras.callbacks.Callback`
- Guide: [Writing your own callbacks \(https://www.tensorflow.org/guide/keras/guide/keras/custom\\_callback\)](#)
- Guide: [Training and evaluation with the built-in methods \(https://www.tensorflow.org/guide/keras/train\\_and\\_evaluate\)](#) (the *Using callbacks* section)

You may also find the following migration-related resources useful:

- The [Fault tolerance migration guide \(fault\\_tolerance.ipynb\)](#): `tf.keras.callbacks.BackupAndRestore` for `Model.fit`, or `tf.train.Checkpoint` and `tf.train.CheckpointManager` APIs for a custom training loop
- The [Early stopping migration guide \(early\\_stopping.ipynb\)](#): `tf.keras.callbacks.EarlyStopping` is a built-in early stopping callback
- The [TensorBoard migration guide \(tensorboard.ipynb\)](#): TensorBoard enables tracking and displaying metrics
- The [LoggingTensorHook and StopAtStepHook to Keras callbacks migration guide \(logging\\_stop\\_hook.ipynb\)](#)
- The [SessionRunHook to Keras callbacks guide \(sessionrunhook\\_callback.ipynb\)](#)

**Copyright 2021 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```



[View on TensorFlow.org](https://www.tensorflow.org)

([https://www.tensorflow.org/guide/migrate/validate\\_correctness](https://www.tensorflow.org/guide/migrate/validate_correctness))



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/validate_correctness.ipynb)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/validate\\_correctness.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/validate_correctness.ipynb))

# Validating correctness & numerical equivalence

When migrating your TensorFlow code from TF1.x to TF2, it is a good practice to ensure that your migrated code behaves the same way in TF2 as it did in TF1.x.

This guide covers migration code examples with the `tf.compat.v1.keras.utils.track_tf1_style_variables` modeling shim applied to `tf.keras.layers.Layer` methods. Read the [model mapping guide \(./model\\_mapping.ipynb\)](#) to find out more about the TF2 modeling shims.

This guide details approaches you can use to:

- Validate the correctness of the results obtained from training models using the migrated code
- Validate the numerical equivalence of your code across TensorFlow versions

## Setup

In [ ]:

```
!pip uninstall -y -q tensorflow
```

In [ ]:

```
# Install tf-nightly as the DeterministicRandomTestTool is available only in
# Tensorflow 2.8
!pip install -q tf-nightly
```

In [ ]:

```
!pip install -q tf_slim
```

In [ ]:

```
import tensorflow as tf
import tensorflow.compat.v1 as v1

import numpy as np
import tf_slim as slim
import sys

from contextlib import contextmanager
```

In [ ]:

```
!git clone --depth=1 https://github.com/tensorflow/models.git
import models.research.slim.nets.inception_resnet_v2 as inception
```

If you're putting a nontrivial chunk of forward pass code into the shim, you want to know that it is behaving the same way as it did in TF1.x. For example, consider trying to put an entire TF-Slim Inception-Resnet-v2 model into the shim as such:

In [ ]:

```
# TF1 Inception resnet v2 forward pass based on slim layers
def inception_resnet_v2(inputs, num_classes, is_training):
    with slim.arg_scope(
        inception.inception_resnet_v2_arg_scope(batch_norm_scale=True)):
        return inception.inception_resnet_v2(inputs, num_classes, is_training=is_training)
```

In [ ]:

```
class InceptionResnetV2(tf.keras.layers.Layer):
    """Slim InceptionResnetV2 forward pass as a Keras layer"""

    def __init__(self, num_classes, **kwargs):
        super().__init__(**kwargs)
        self.num_classes = num_classes

    @tf.compat.v1.keras.utils.track_tf1_style_variables
    def call(self, inputs, training=None):
        is_training = training or False

        # Slim does not accept `None` as a value for is_training,
        # Keras will still pass `None` to layers to construct functional models
        # without forcing the layer to always be in training or in inference.
        # However, `None` is generally considered to run layers in inference.

        with slim.arg_scope(
            inception.inception_resnet_v2_arg_scope(batch_norm_scale=True)):
            return inception.inception_resnet_v2(
                inputs, self.num_classes, is_training=is_training)
```

As it so happens, this layer actually works perfectly fine out of the box (complete with accurate regularization loss tracking).

However, this is not something you want to take for granted. Follow the below steps to verify that it is actually behaving as it did in TF1.x, down to observing perfect numerical equivalence. These steps can also help you triangulate what part of the forward pass is causing a divergence from TF1.x (identify if the divergence arises in the model forward pass as opposed to a different part of the model).

## Step 1: Verify variables are only created once

The very first thing you should verify is that you have correctly built the model in a way that reuses variables in each call rather than accidentally creating and using new variables each time. For example, if your model creates a new Keras layer or calls `tf.Variable` in each forward pass call then it is most likely failing to capture variables and creating new ones each time.

Below are two context manager scopes you can use to detect when your model is creating new variables and debug which part of the model is doing it.

In [ ]:

```
@contextmanager
def assert_no_variable_creations():
    """Assert no variables are created in this context manager scope."""
    def invalid_variable_creator(next_creator, **kwargs):
        raise ValueError("Attempted to create a new variable instead of reusing an existing one. Args: {}".format(kwargs))

    with tf.variable_creator_scope(invalid_variable_creator):
        yield

@contextmanager
def catch_and_raise_created_variables():
    """Raise all variables created within this context manager scope (if any)."""
    created_vars = []
    def variable_catcher(next_creator, **kwargs):
        var = next_creator(**kwargs)
        created_vars.append(var)
        return var

    with tf.variable_creator_scope(variable_catcher):
        yield
    if created_vars:
        raise ValueError("Created vars:", created_vars)
```

The first scope (`assert_no_variable_creations()`) will raise an error immediately once you try creating a variable within the scope. This allows you to inspect the stacktrace (and use interactive debugging) to figure out exactly what lines of code created a variable instead of reusing an existing one.

The second scope (`catch_and_raise_created_variables()`) will raise an exception at the end of the scope if any variables ended up being created. This exception will include the list of all variables created in the scope. This is useful for figuring out what the set of all weights your model is creating is in case you can spot general patterns. However, it is less useful for identifying the exact lines of code where those variables got created.

Use both scopes below to verify that the shim-based InceptionResnetV2 layer does not create any new variables after the first call (presumably reusing them).

In [ ]:

```
model = InceptionResnetV2(1000)
height, width = 299, 299
num_classes = 1000

inputs = tf.ones( (1, height, width, 3))
# Create all weights on the first call
model(inputs)

# Verify that no new weights are created in followup calls
with assert_no_variable_creations():
    model(inputs)
with catch_and_raise_created_variables():
    model(inputs)
```

In the example below, observe how these decorators work on a layer that incorrectly creates new weights each time instead of reusing existing ones.

In [ ]:

```
class BrokenScalingLayer(tf.keras.layers.Layer):
    """Scaling layer that incorrectly creates new weights each time"""

    @tf.compat.v1.keras.utils.track_tf1_style_variables
    def call(self, inputs):
        var = tf.Variable(initial_value=2.0)
        bias = tf.Variable(initial_value=2.0, name='bias')
        return inputs * var + bias
```

In [ ]:

```
model = BrokenScalingLayer()
inputs = tf.ones( (1, height, width, 3))
model(inputs)

try:
    with assert_no_variable_creations():
        model(inputs)
except ValueError as err:
    import traceback
    traceback.print_exc()
```

In [ ]:

```
model = BrokenScalingLayer()
inputs = tf.ones( (1, height, width, 3))
model(inputs)

try:
    with catch_and_raise_created_variables():
        model(inputs)
except ValueError as err:
    print(err)
```

You can fix the layer by making sure it only creates the weights once and then reuses them each time.

In [ ]:

```
class FixedScalingLayer(tf.keras.layers.Layer):
    """Scaling layer that incorrectly creates new weights each time"""

    def __init__(self):
        super().__init__()
        self.var = None
        self.bias = None

    @tf.compat.v1.keras.utils.track_tf1_style_variables
    def call(self, inputs):
        if self.var is None:
            self.var = tf.Variable(initial_value=2.0)
            self.bias = tf.Variable(initial_value=2.0, name='bias')
        return inputs * self.var + self.bias

model = FixedScalingLayer()
inputs = tf.ones( (1, height, width, 3))
model(inputs)

with assert_no_variable_creations():
    model(inputs)
with catch_and_raise_created_variables():
    model(inputs)
```

## Troubleshooting

Here are some common reasons why your model might accidentally be creating new weights instead of reusing existing ones:

1. It uses an explicit `tf.Variable` call without reusing already-created `tf.Variables`. Fix this by first checking if it has not been created then reusing the existing ones.
2. It creates a Keras layer or model directly in the forward pass each time (as opposed to `tf.compat.v1.layers`). Fix this by first checking if it has not been created then reusing the existing ones.
3. It is built on top of `tf.compat.v1.layers` but fails to assign all `compat.v1.layers` an explicit name or to wrap your `compat.v1.layer` usage inside of a named `variable_scope`, causing the autogenerated layer names to increment in each model call. Fix this by putting a named `tf.compat.v1.variable_scope` inside your shim-decorated method that wraps all of your `tf.compat.v1.layers` usage.

## Step 2: Check that variable counts, names, and shapes match

The second step is to make sure your layer running in TF2 creates the same number of weights, with the same shapes, as the corresponding code does in TF1.x.

You can do a mix of manually checking them to see that they match, and doing the checks programmatically in a unit test as shown below.

In [ ]:

```
# Build the forward pass inside a TF1.x graph, and
# get the counts, shapes, and names of the variables
graph = tf.Graph()
with graph.as_default(), tf.compat.v1.Session(graph=graph) as sess:
    height, width = 299, 299
    num_classes = 1000
    inputs = tf.ones( (1, height, width, 3))

    out, endpoints = inception_resnet_v2(inputs, num_classes, is_training=False)

    tf1_variable_names_and_shapes = {
        var.name: (var.trainable, var.shape) for var in tf.compat.v1.global_variables()}
    num_tf1_variables = len(tf.compat.v1.global_variables())
```

Next, do the same for the shim-wrapped layer in TF2. Notice that the model is also called multiple times before grabbing the weights. This is done to effectively test for variable reuse.

In [ ]:

```
height, width = 299, 299
num_classes = 1000

model = InceptionResnetV2(num_classes)
# The weights will not be created until you call the model

inputs = tf.ones( (1, height, width, 3))
# Call the model multiple times before checking the weights, to verify variables
# get reused rather than accidentally creating additional variables
out, endpoints = model(inputs, training=False)
out, endpoints = model(inputs, training=False)

# Grab the name: shape mapping and the total number of variables separately,
# because in TF2 variables can be created with the same name
num_tf2_variables = len(model.variables)
tf2_variable_names_and_shapes = {
    var.name: (var.trainable, var.shape) for var in model.variables}
```

In [ ]:

```
# Verify that the variable counts, names, and shapes all match:
assert num_tf1_variables == num_tf2_variables
assert tf1_variable_names_and_shapes == tf2_variable_names_and_shapes
```

The shim-based InceptionResnetV2 layer passes this test. However, in the case where they don't match, you can run it through a diff (text or other) to see where the differences are.

This can provide a clue as to what part of the model isn't behaving as expected. With eager execution you can use pdb, interactive debugging, and breakpoints to dig into the parts of the model that seem suspicious, and debug what is going wrong in more depth.

## Troubleshooting

- Pay close attention to the names of any variables created directly by explicit `tf.Variable` calls and Keras layers/models as their variable name generation semantics may differ slightly between TF1.x graphs and TF2 functionality such as eager execution and `tf.function` even if everything else is working properly. If this is the case for you, adjust your test to account for any slightly different naming semantics.
- You may sometimes find that the `tf.Variables`, `tf.keras.layers.Layers`, or `tf.keras.Models` created in your training loop's forward pass are missing from your TF2 variables list even if they were captured by the variables collection in TF1.x. Fix this by assigning the variables/layers/models that your forward pass creates to instance attributes in your model. See [here](https://www.tensorflow.org/guide/keras/custom_layers_and_models) ([https://www.tensorflow.org/guide/keras/custom\\_layers\\_and\\_models](https://www.tensorflow.org/guide/keras/custom_layers_and_models)) for more info.

## Step 3: Reset all variables, check numerical equivalence with all randomness disabled

The next step is to verify numerical equivalence for both the actual outputs and the regularization loss tracking when you fix the model such that there is no random number generation involved (such as during inference).

The exact way to do this may depend on your specific model, but in most models (such as this one), you can do this by:

1. Initializing the weights to the same value with no randomness. This can be done by resetting them to a fixed value after they have been created.
2. Running the model in inference mode to avoid triggering any dropout layers which can be sources of randomness.

The following code demonstrates how you can compare the TF1.x and TF2 results this way.

In [ ]:

```
graph = tf.Graph()
with graph.as_default(), tf.compat.v1.Session(graph=graph) as sess:
    height, width = 299, 299
    num_classes = 1000
    inputs = tf.ones((1, height, width, 3))

    out, endpoints = inception_resnet_v2(inputs, num_classes, is_training=False)

    # Rather than running the global variable initializers,
    # reset all variables to a constant value
    var_reset = tf.group([var.assign(tf.ones_like(var) * 0.001) for var in tf.compat.v1.global_variables()])
    sess.run(var_reset)

    # Grab the outputs & regularization loss
    reg_losses = tf.compat.v1.get_collection(tf.compat.v1.GraphKeys.REGULARIZATION_LOSSES)
    tf1_regularization_loss = sess.run(tf.math.add_n(reg_losses))
    tf1_output = sess.run(out)

print("Regularization loss:", tf1_regularization_loss)
tf1_output[0][:5]
```

Get the TF2 results.

In [ ]:

```
height, width = 299, 299
num_classes = 1000

model = InceptionResnetV2(num_classes)

inputs = tf.ones((1, height, width, 3))
# Call the model once to create the weights
out, endpoints = model(inputs, training=False)

# Reset all variables to the same fixed value as above, with no randomness
for var in model.variables:
    var.assign(tf.ones_like(var) * 0.001)
tf2_output, endpoints = model(inputs, training=False)

# Get the regularization loss
tf2_regularization_loss = tf.math.add_n(model.losses)

print("Regularization loss:", tf2_regularization_loss)
tf2_output[0][:5]
```

In [ ]:

```
# Create a dict of tolerance values
tol_dict={'rtol':1e-06, 'atol':1e-05}
```

In [ ]:

```
# Verify that the regularization loss and output both match
# when we fix the weights and avoid randomness by running inference:
np.testing.assert_allclose(tf1_regularization_loss, tf2_regularization_loss.numpy(), **tol_dict)
np.testing.assert_allclose(tf1_output, tf2_output.numpy(), **tol_dict)
```

The numbers match between TF1.x and TF2 when you remove sources of randomness, and the TF2-compatible InceptionResnetV2 layer passes the test.

If you are observing the results diverging for your own models, you can use printing or pdb and interactive debugging to identify where and why the results start to diverge. Eager execution can make this significantly easier. You can also use an ablation approach to run only small portions of the model on fixed intermediate inputs and isolate where the divergence happens.

Conveniently, many slim nets (and other models) also expose intermediate endpoints that you can probe.

## Step 4: Align random number generation, check numerical equivalence in both training and inference

The final step is to verify that the TF2 model numerically matches the TF1.x model, even when accounting for random number generation in variable initialization and in the forward pass itself (such as dropout layers during the forward pass).

You can do this by using the testing tool below to make random number generation semantics match between TF1.x graphs/sessions and eager execution.

TF1 legacy graphs/sessions and TF2 eager execution use different stateful random number generation semantics.

In `tf.compat.v1.Session`s, if no seeds are specified, the random number generation depends on how many operations are in the graph at the time when the random operation is added, and how many times the graph is run. In eager execution, stateful random number generation depends on the global seed, the operation random seed, and how many times the operation with the given random seed is run. See `tf.random.set_seed` for more info.

The following `v1.keras.utils.DeterministicRandomTestTool` ([https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/keras/utils/DeterministicRandomTestTool](https://www.tensorflow.org/api_docs/python/tf/compat/v1/keras/utils/DeterministicRandomTestTool)) class provides a context manager `scope()` that can make stateful random operations use the same seed across both TF1 graphs/sessions and eager execution.

The tool provides two testing modes:

1. `constant` which uses the same seed for every single operation no matter how many times it has been called and,
2. `num_random_ops` which uses the number of previously-observed stateful random operations as the operation seed.

This applies both to the stateful random operations used for creating and initializing variables, and to the stateful random operations used in computation (such as for dropout layers).

Generate three random tensors to show how to use this tool to make stateful random number generation match between sessions and eager execution.

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool()
with random_tool.scope():
    graph = tf.Graph()
    with graph.as_default(), tf.compat.v1.Session(graph=graph) as sess:
        a = tf.random.uniform(shape=(3,1))
        a = a * 3
        b = tf.random.uniform(shape=(3,3))
        b = b * 3
        c = tf.random.uniform(shape=(3,3))
        c = c * 3
        graph_a, graph_b, graph_c = sess.run([a, b, c])

graph_a, graph_b, graph_c
```

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool()
with random_tool.scope():
    a = tf.random.uniform(shape=(3,1))
    a = a * 3
    b = tf.random.uniform(shape=(3,3))
    b = b * 3
    c = tf.random.uniform(shape=(3,3))
    c = c * 3

a, b, c
```

In [ ]:

```
# Demonstrate that the generated random numbers match
np.testing.assert_allclose(graph_a, a.numpy(), **tol_dict)
np.testing.assert_allclose(graph_b, b.numpy(), **tol_dict)
np.testing.assert_allclose(graph_c, c.numpy(), **tol_dict)
```

However, notice that in `constant` mode, because `b` and `c` were generated with the same seed and have the same shape, they will have exactly the same values.

In [ ]:

```
np.testing.assert_allclose(b.numpy(), c.numpy(), **tol_dict)
```

## Trace order

If you are worried about some random numbers matching in `constant` mode reducing your confidence in your numerical equivalence test (for example if several weights take on the same initializations), you can use the `num_random_ops` mode to avoid this. In the `num_random_ops` mode, the generated random numbers will depend on the ordering of random ops in the program.

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    graph = tf.Graph()
    with graph.as_default(), tf.compat.v1.Session(graph=graph) as sess:
        a = tf.random.uniform(shape=(3,1))
        a = a * 3
        b = tf.random.uniform(shape=(3,3))
        b = b * 3
        c = tf.random.uniform(shape=(3,3))
        c = c * 3
        graph_a, graph_b, graph_c = sess.run([a, b, c])

graph_a, graph_b, graph_c
```

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    a = tf.random.uniform(shape=(3,1))
    a = a * 3
    b = tf.random.uniform(shape=(3,3))
    b = b * 3
    c = tf.random.uniform(shape=(3,3))
    c = c * 3

a, b, c
```

In [ ]:

```
# Demonstrate that the generated random numbers match
np.testing.assert_allclose(graph_a, a.numpy(), **tol_dict)
np.testing.assert_allclose(graph_b, b.numpy(), **tol_dict )
np.testing.assert_allclose(graph_c, c.numpy(), **tol_dict)
```

In [ ]:

```
# Demonstrate that with the 'num_random_ops' mode,
# b & c took on different values even though
# their generated shape was the same
assert not np.allclose(b.numpy(), c.numpy(), **tol_dict)
```

However, notice that in this mode random generation is sensitive to program order, and so the following generated random numbers do not match.

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    a = tf.random.uniform(shape=(3,1))
    a = a * 3
    b = tf.random.uniform(shape=(3,3))
    b = b * 3

random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    b_prime = tf.random.uniform(shape=(3,3))
    b_prime = b_prime * 3
    a_prime = tf.random.uniform(shape=(3,1))
    a_prime = a_prime * 3

assert not np.allclose(a.numpy(), a_prime.numpy())
assert not np.allclose(b.numpy(), b_prime.numpy())
```

To allow for debugging variations due to tracing order, `DeterministicRandomTestTool` in `num_random_ops` mode allows you to see how many random operations have been traced with the `operation_seed` property.

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    print(random_tool.operation_seed)
    a = tf.random.uniform(shape=(3,1))
    a = a * 3
    print(random_tool.operation_seed)
    b = tf.random.uniform(shape=(3,3))
    b = b * 3
    print(random_tool.operation_seed)
```

If you need to account for varying trace order in your tests, you can even set the auto-incrementing `operation_seed` explicitly. For example, you can use this to make random number generation match across two different program orders.

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    print(random_tool.operation_seed)
    a = tf.random.uniform(shape=(3,1))
    a = a * 3
    print(random_tool.operation_seed)
    b = tf.random.uniform(shape=(3,3))
    b = b * 3

random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    random_tool.operation_seed = 1
    b_prime = tf.random.uniform(shape=(3,3))
    b_prime = b_prime * 3
    random_tool.operation_seed = 0
    a_prime = tf.random.uniform(shape=(3,1))
    a_prime = a_prime * 3

np.testing.assert_allclose(a.numpy(), a_prime.numpy(), **tol_dict)
np.testing.assert_allclose(b.numpy(), b_prime.numpy(), **tol_dict)
```

However, `DeterministicRandomTestTool` disallows reusing already-used operation seeds, so make sure the auto-incremented sequences cannot overlap. This is because eager execution generates different numbers for follow-on usages of the same operation seed while TF1 graphs and sessions do not, so raising an error helps keep session and eager stateful random number generation in line.

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    random_tool.operation_seed = 1
    b_prime = tf.random.uniform(shape=(3,3))
    b_prime = b_prime * 3
    random_tool.operation_seed = 0
    a_prime = tf.random.uniform(shape=(3,1))
    a_prime = a_prime * 3
    try:
        c = tf.random.uniform(shape=(3,1))
        raise RuntimeError("An exception should have been raised before this, " +
                           "because the auto-incremented operation seed will " +
                           "overlap an already-used value")
    except ValueError as err:
        print(err)
```

## Verifying Inference

You can now use the `DeterministicRandomTestTool` to make sure the `InceptionResnetV2` model matches in inference, even when using the random weight initialization. For a stronger test condition due to matching program order, use the `num_random_ops` mode.

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    graph = tf.Graph()
    with graph.as_default(), tf.compat.v1.Session(graph=graph) as sess:
        height, width = 299, 299
        num_classes = 1000
        inputs = tf.ones((1, height, width, 3))

        out, endpoints = inception_resnet_v2(inputs, num_classes, is_training=False)

        # Initialize the variables
        sess.run(tf.compat.v1.global_variables_initializer())

        # Grab the outputs & regularization loss
        reg_losses = tf.compat.v1.get_collection(tf.compat.v1.GraphKeys.REGULARIZATION_LOSSES)
        tf1_regularization_loss = sess.run(tf.math.add_n(reg_losses))
        tf1_output = sess.run(out)

        print("Regularization loss:", tf1_regularization_loss)
```

In [ ]:

```
height, width = 299, 299
num_classes = 1000

random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    model = InceptionResnetV2(num_classes)

    inputs = tf.ones((1, height, width, 3))
    tf2_output, endpoints = model(inputs, training=False)

    # Grab the regularization loss as well
    tf2_regularization_loss = tf.math.add_n(model.losses)

    print("Regularization loss:", tf2_regularization_loss)
```

In [ ]:

```
# Verify that the regularization loss and output both match
# when using the DeterministicRandomTestTool:
np.testing.assert_allclose(tf1_regularization_loss, tf2_regularization_loss.numpy(), **tol_dict)
np.testing.assert_allclose(tf1_output, tf2_output.numpy(), **tol_dict)
```

## Verifying Training

Because `DeterministicRandomTestTool` works for *all* stateful random operations (including both weight initialization and computation such as dropout layers), you can use it to verify the models match in training mode as well. You can again use the `num_random_ops` mode because the program order of the stateful random ops matches.

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    graph = tf.Graph()
    with graph.as_default(), tf.compat.v1.Session(graph=graph) as sess:
        height, width = 299, 299
        num_classes = 1000
        inputs = tf.ones( (1, height, width, 3))

        out, endpoints = inception_resnet_v2(inputs, num_classes, is_training=True)

    # Initialize the variables
    sess.run(tf.compat.v1.global_variables_initializer())

    # Grab the outputs & regularization loss
    reg_losses = tf.compat.v1.get_collection(tf.compat.v1.GraphKeys.REGULARIZATION_LOSSES)
    tf1_regularization_loss = sess.run(tf.math.add_n(reg_losses))
    tf1_output = sess.run(out)

print("Regularization loss:", tf1_regularization_loss)
```

In [ ]:

```
height, width = 299, 299
num_classes = 1000

random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    model = InceptionResnetV2(num_classes)

    inputs = tf.ones((1, height, width, 3))
    tf2_output, endpoints = model(inputs, training=True)

    # Grab the regularization loss as well
    tf2_regularization_loss = tf.math.add_n(model.losses)

print("Regularization loss:", tf2_regularization_loss)
```

In [ ]:

```
# Verify that the regularization loss and output both match
# when using the DeterministicRandomTestTool
np.testing.assert_allclose(tf1_regularization_loss, tf2_regularization_loss.numpy(), **tol_dict)
np.testing.assert_allclose(tf1_output, tf2_output.numpy(), **tol_dict)
```

You have now verified that the `InceptionResnetV2` model running eagerly with decorators around `tf.keras.layers.Layer` numerically matches the slim network running in TF1 graphs and sessions.

Note: When using the `DeterministicRandomTestTool` in `num_random_ops` mode, it is suggested you directly use and call the `tf.keras.layers.Layer` method decorator when testing for numerical equivalence. Embedding it within a Keras functional model or other Keras models can produce differences in stateful random operation tracing order that can be tricky to reason about or match exactly when comparing TF1.x graphs/sessions and eager execution.

For example, calling the `InceptionResnetV2` layer directly with `training=True` interleaves variable initialization with the dropout order according to the network creation order.

On the other hand, first putting the `tf.keras.layers.Layer` decorator in a Keras functional model and only then calling the model with `training=True` is equivalent to initializing all variables then using the dropout layer. This produces a different tracing order and a different set of random numbers.

However, the default `mode='constant'` is not sensitive to these differences in tracing order and will pass without extra work even when embedding the layer in a Keras functional model.

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool()
with random_tool.scope():
    graph = tf.Graph()
    with graph.as_default(), tf.compat.v1.Session(graph=graph) as sess:
        height, width = 299, 299
        num_classes = 1000
        inputs = tf.ones( (1, height, width, 3))

        out, endpoints = inception_resnet_v2(inputs, num_classes, is_training=True)

    # Initialize the variables
    sess.run(tf.compat.v1.global_variables_initializer())

    # Get the outputs & regularization losses
    reg_losses = tf.compat.v1.get_collection(tf.compat.v1.GraphKeys.REGULARIZATION_LOSSES)
    tf1_regularization_loss = sess.run(tf.math.add_n(reg_losses))
    tf1_output = sess.run(out)

print("Regularization loss:", tf1_regularization_loss)
```

In [ ]:

```
height, width = 299, 299
num_classes = 1000

random_tool = v1.keras.utils.DeterministicRandomTestTool()
with random_tool.scope():
    keras_input = tf.keras.Input(shape=(height, width, 3))
    layer = InceptionResnetV2(num_classes)
    model = tf.keras.Model(inputs=keras_input, outputs=layer(keras_input))

    inputs = tf.ones((1, height, width, 3))
    tf2_output, endpoints = model(inputs, training=True)

    # Get the regularization loss
    tf2_regularization_loss = tf.math.add_n(model.losses)

print("Regularization loss:", tf2_regularization_loss)
```

In [ ]:

```
# Verify that the regularization loss and output both match
# when using the DeterministicRandomTestTool
np.testing.assert_allclose(tf1_regularization_loss, tf2_regularization_loss.numpy(), **tol_dict)
np.testing.assert_allclose(tf1_output, tf2_output.numpy(), **tol_dict)
```

## Step 3b or 4b (optional): Testing with pre-existing checkpoints

After step 3 or step 4 above, it can be useful to run your numerical equivalence tests when starting from pre-existing name-based checkpoints if you have some. This can test both that your legacy checkpoint loading is working correctly and that the model itself is working right. The [Reusing TF1.x checkpoints guide \(./reuse\\_checkpoints.ipynb\)](#) covers how to reuse your pre-existing TF1.x checkpoints and transfer them over to TF2 checkpoints.

## Additional Testing & Troubleshooting

As you add more numerical equivalence tests, you may also choose to add a test that verifies your gradient computation (or even your optimizer updates) match.

Backpropagation and gradient computation are more prone to floating point numerical instabilities than model forward passes. This means that as your equivalence tests cover more non-isolated parts of your training, you may begin to see non-trivial numerics differences between running fully eagerly and your TF1 graphs. This may be caused by TensorFlow's graph optimizations that do things such as replace subexpressions in a graph with fewer mathematical operations.

To isolate whether this is likely to be the case, you can compare your TF1 code to TF2 computation happening inside of a `tf.function` (which applies graph optimization passes like your TF1 graph) rather than to a purely eager computation. Alternatively, you can try using `tf.config.optimizer.set_experimental_options` to disable optimization passes such as "`arithmetic_optimization`" before your TF1 computation to see if the result ends up numerically closer to your TF2 computation results. In your actual training runs it is recommended you use `tf.function` with optimization passes enabled for performance reasons, but you may find it useful to disable them in your numerical equivalence unit tests.

Similarly, you may also find that `tf.compat.v1.train` optimizers and TF2 optimizers have slightly different floating point numerics properties than TF2 optimizers, even if the mathematical formulas they are representing are the same. This is less likely to be an issue in your training runs, but it may require a higher numerical tolerance in equivalence unit tests.

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Automatically rewrite TF 1.x and compat.v1 API symbols



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/upgrade)

(<https://www.tensorflow.org/guide/migrate/upgrade>) (<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/upgrade.ipynb>) (<https://github.com/tensorflow/docs/blob/master/site/en/guide/migrate/upgrade.ipynb>)



[Run in Google Colab](#)

TensorFlow 2.x includes many API changes from TF 1.x and the `tf.compat.v1` APIs, such as reordering arguments, renaming symbols, and changing default values for parameters. Manually performing all of these modifications would be tedious and prone to error. To streamline the changes, and to make your transition to TF 2.x as seamless as possible, the TensorFlow team has created the `tf_upgrade_v2` utility to help transition legacy code to the new API.

Note: `tf_upgrade_v2` is installed automatically for TensorFlow 1.13 and later (including all TF 2.x builds).

Typical usage is like this:

```
tf_upgrade_v2 \
--intree my_project/ \
--outtree my_project_v2/ \
--reportfile report.txt
```

It will accelerate your upgrade process by converting existing TensorFlow 1.x Python scripts to TensorFlow 2.x.

The conversion script automates many mechanical API transformations, though many APIs cannot be automatically migrated. It is also not able to fully make your code compatible with TF2 behaviors and APIs. So, it is only a part of your migration journey.

## Compatibility modules

Certain API symbols can not be upgraded simply by using a string replacement. Those that cannot be automatically upgraded will be mapped to their locations in the `compat.v1` module. This module replaces TF 1.x symbols like `tf.foo` with the equivalent `tf.compat.v1.foo` reference. If you are already using `compat.v1` APIs by importing TF via `import tensorflow.compat.v1 as tf`, the `tf_upgrade_v2` script will attempt to convert these usages to the non-compat APIs where possible. Note that while some `compat.v1` APIs are compatible with TF2.x behaviors, many are not. So, we recommend that you manually proofread replacements and migrate them to new APIs in the `tf.*` namespace instead of `tf.compat.v1` namespace as quickly as possible.

Because of TensorFlow 2.x module deprecations (for example, `tf.flags` and `tf.contrib`), some changes can not be worked around by switching to `compat.v1`. Upgrading this code may require using an additional library (for example, `absl.flags` (<https://github.com/abseil/abseil-py>)) or switching to a package in `tensorflow/addons` (<http://www.github.com/tensorflow/addons>).

## Recommended upgrade process

The rest of this guide demonstrates how to use the symbol-rewriting script. While the script is easy to use, it is strongly recommended that you use the script as part of the following process:

1. **Unit Test:** Ensure that the code you're upgrading has a unit test suite with reasonable coverage. This is Python code, so the language won't protect you from many classes of mistakes. Also ensure that any dependency you have has already been upgraded to be compatible with TensorFlow 2.x.
2. **Install TensorFlow 1.15:** Upgrade your TensorFlow to the latest TensorFlow 1.x version, at least 1.15. This includes the final TensorFlow 2.0 API in `tf.compat.v2`.
3. **Test With 1.15:** Ensure your unit tests pass at this point. You'll be running them repeatedly as you upgrade so starting from green is important.
4. **Run the upgrade script:** Run `tf_upgrade_v2` on your entire source tree, tests included. This will upgrade your code to a format where it only uses symbols available in TensorFlow 2.0. Deprecated symbols will be accessed with `tf.compat.v1`. These will eventually require manual attention, but not immediately.
5. **Run the converted tests with TensorFlow 1.15:** Your code should still run fine in TensorFlow 1.15. Run your unit tests again. Any error in your tests here means there's a bug in the upgrade script. [Please let us know \(<https://github.com/tensorflow/tensorflow/issues>\)](https://github.com/tensorflow/tensorflow/issues).
6. **Check the upgrade report for warnings and errors:** The script writes a report file that explains any conversions you should double-check, or any manual action you need to take. For example: Any remaining instances of contrib will require manual action to remove. Please consult [the RFC for more instructions \(<https://github.com/tensorflow/community/blob/master/rfcs/20180907-contrib-sunset.md>\)](https://github.com/tensorflow/community/blob/master/rfcs/20180907-contrib-sunset.md).
7. **Install TensorFlow 2.x:** At this point it should be safe to switch to TensorFlow 2.x binaries, even if you are running with legacy behaviors
8. **Test with `v1.disable_v2_behavior`:** Re-running your tests with a `v1.disable_v2_behavior()` in the tests' main function should give the same results as running under 1.15.
9. **Enable V2 Behavior:** Now that your tests work using the TF2 binaries, you can now begin migrating your code to avoiding `tf.estimator`s and only using supported TF2 behaviors (with no TF2 behavior disabling). See the [Migration guides \(<https://tensorflow.org/guide/migrate>\)](https://tensorflow.org/guide/migrate) for details.

## Using the symbol-rewriting `tf_upgrade_v2` script

### Setup

Before getting started ensure that TensorFlow 2.x is installed.

In [ ]:

```
import tensorflow as tf
print(tf.__version__)
```

Clone the [tensorflow/models \(<https://github.com/tensorflow/models>\)](https://github.com/tensorflow/models) git repository so you have some code to test on:

In [ ]:

```
!git clone --branch r1.13.0 --depth 1 https://github.com/tensorflow/models
```

### Read the help

The script should be installed with TensorFlow. Here is the builtin help:

In [ ]:

```
!tf_upgrade_v2 -h
```

### Example TF1 code

Here is a simple TensorFlow 1.0 script:

In [ ]:

```
!head -n 65 models/samples/cookbook/regression/custom_regression.py | tail -n 10
```

With TensorFlow 2.x installed it does not run:

In [ ]:

```
!(cd models/samples/cookbook/regression && python custom_regression.py)
```

## Single file

The script can be run on a single Python file:

In [ ]:

```
!tf_upgrade_v2 N
--infile models/samples/cookbook/regression/custom_regression.py \
--outfile /tmp/custom_regression_v2.py
```

The script will print errors if it can not find a fix for the code.

## Directory tree

Typical projects, including this simple example, will use much more than one file. Typically want to update an entire package, so the script can also be run on a directory tree:

In [ ]:

```
# update the .py files and copy all the other files to the outtree
!tf_upgrade_v2 N
--intree models/samples/cookbook/regression/ \
--outtree regression_v2/ \
--reportfile tree_report.txt
```

Note the one warning about the `dataset.make_one_shot_iterator` function.

Now the script works in with TensorFlow 2.x:

Note that because the `tf.compat.v1` module is included in TF 1.15, the converted script will also run in TensorFlow 1.15.

In [ ]:

```
!(cd regression_v2 && python custom_regression.py 2>&1) | tail
```

## Detailed report

The script also reports a list of detailed changes. In this example it found one possibly unsafe transformation and included a warning at the top of the file:

In [ ]:

```
!head -n 20 tree_report.txt
```

Note again the one warning about the `Dataset.make_one_shot_iterator` function.

In other cases the output will explain the reasoning for non-trivial changes:

In [ ]:

```
%%writefile dropout.py
import tensorflow as tf

d = tf.nn.dropout(tf.range(10), 0.2)
z = tf.zeros_like(d, optimize=False)
```

In [ ]:

```
!tf_upgrade_v2 N
--infile dropout.py \
--outfile dropout_v2.py \
--reportfile dropout_report.txt > /dev/null
```

In [ ]:

```
!cat dropout_report.txt
```

Here is the modified file contents, note how the script adds argument names to deal with moved and renamed arguments:

In [ ]:

```
!cat dropout_v2.py
```

A larger project might contain a few errors. For example convert the deeplab model:

In [ ]:

```
!tf_upgrade_v2 --intree models/research/deeplab \  
--outtree deeplab_v2 \  
--reportfile deeplab_report.txt > /dev/null
```

It produced the output files:

In [ ]:

```
!ls deeplab_v2
```

But there were errors. The report will help you pin-point what you need to fix before this will run. Here are the first three errors:

In [ ]:

```
!cat deeplab_report.txt | grep -i models/research/deeplab | grep -i error | head -n 3
```

## "Safety" mode

The conversion script also has a less invasive SAFETY mode that simply changes the imports to use the tensorflow.compat.v1 module:

In [ ]:

```
!cat dropout.py
```

In [ ]:

```
!tf_upgrade_v2 --mode SAFETY --infile dropout.py --outfile dropout_v2_safe.py > /dev/null
```

In [ ]:

```
!cat dropout_v2_safe.py
```

As you can see this doesn't upgrade your code, but does allow TensorFlow 1 code to run against TensorFlow 2 binaries. Note that this does not mean your code is running supported TF 2.x behaviors!

## Caveats

- Do not update parts of your code manually before running this script. In particular, functions that have had reordered arguments like `tf.argmax` or `tf.batch_to_space` cause the script to incorrectly add keyword arguments that mismatch your existing code.
- The script assumes that `tensorflow` is imported using `import tensorflow as tf`, or `import tensorflow.compat.v1 as tf`.
- This script does not reorder arguments. Instead, the script adds keyword arguments to functions that have their arguments reordered.
- Check out [tf2up.ml](https://github.com/lc0/tf2up) (<https://github.com/lc0/tf2up>) for a convenient tool to upgrade Jupyter notebooks and Python files in a GitHub repository.

To report upgrade script bugs or make feature requests, please file an issue on [GitHub](https://github.com/tensorflow/tensorflow/issues) (<https://github.com/tensorflow/tensorflow/issues>).

**Copyright 2021 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");  
# you may not use this file except in compliance with the License.  
# You may obtain a copy of the License at  
#  
# https://www.apache.org/licenses/LICENSE-2.0  
#  
# Unless required by applicable law or agreed to in writing, software  
# distributed under the License is distributed on an "AS IS" BASIS,  
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
# See the License for the specific language governing permissions and  
# limitations under the License.
```

# Migration Examples: Canned Estimators



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/canned_estimators)

([https://www.tensorflow.org/guide/migrate/canned\\_estimators](https://www.tensorflow.org/guide/migrate/canned_estimators))



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/canned_estimators.ipynb)

Canned (or Premade) Estimators have traditionally been used in TensorFlow 1 as quick and easy ways to train models for a variety of typical use cases. TensorFlow 2 provides straightforward approximate substitutes for a number of them by way of Keras models. For those canned estimators that do not have built-in TensorFlow 2 substitutes, you can still build your own replacement fairly easily.

This guide walks through a few examples of direct equivalents and custom substitutions to demonstrate how TensorFlow 1's `tf.estimator`-derived models can be migrated to TF2 with Keras.

Namely, this guide includes examples for migrating:

- From `tf.estimator`'s `LinearEstimator`, `Classifier` or `Regressor` in TensorFlow 1 to Keras `tf.compat.v1.keras.models.LinearModel` in TensorFlow 2
- From `tf.estimator`'s `DNNEstimator`, `Classifier` or `Regressor` in TensorFlow 1 to a custom Keras DNN ModelKeras in TensorFlow 2
- From `tf.estimator`'s `DNNLinearCombinedEstimator`, `Classifier` or `Regressor` in TensorFlow 1 to `tf.compat.v1.keras.models.WideDeepModel` in TensorFlow 2
- From `tf.estimator`'s `BoostedTreesEstimator`, `Classifier` or `Regressor` in TensorFlow 1 to `tf.compat.v1.keras.models.WideDeepModel` in TensorFlow 2

A common precursor to the training of a model is feature preprocessing, which is done for TensorFlow 1 Estimator models with `tf.feature_column`. For more information on feature preprocessing in TensorFlow 2, see [this guide on migrating feature columns \(migrating\\_feature\\_columns.ipynb\)](#).

## Setup

Start with a couple of necessary TensorFlow imports,

In [ ]:

```
!pip install tensorflow_decision_forests
```

In [ ]:

```
import keras
import pandas as pd
import tensorflow as tf
import tensorflow.compat.v1 as tf1
import tensorflow_decision_forests as tfdf
```

prepare some simple data for demonstration from the standard Titanic dataset,

In [ ]:

```
x_train = pd.read_csv('https://storage.googleapis.com/tf-datasets/titanic/train.csv')
x_eval = pd.read_csv('https://storage.googleapis.com/tf-datasets/titanic/eval.csv')
x_train['sex'].replace(('male', 'female'), (0, 1), inplace=True)
x_eval['sex'].replace(('male', 'female'), (0, 1), inplace=True)

x_train['alone'].replace((0, 1), (0, 1), inplace=True)
x_eval['alone'].replace((0, 1), (0, 1), inplace=True)

x_train['class'].replace(('First', 'Second', 'Third'), (1, 2, 3), inplace=True)
x_eval['class'].replace(('First', 'Second', 'Third'), (1, 2, 3), inplace=True)

x_train.drop(['embark_town', 'deck'], axis=1, inplace=True)
x_eval.drop(['embark_town', 'deck'], axis=1, inplace=True)

y_train = x_train.pop('survived')
y_eval = x_eval.pop('survived')
```

In [ ]:

```
# Data setup for TensorFlow 1 with `tf.estimator`  
def _input_fn():  
    return tf1.data.Dataset.from_tensor_slices((dict(x_train), y_train)).batch(32)  
  
def _eval_input_fn():  
    return tf1.data.Dataset.from_tensor_slices((dict(x_eval), y_eval)).batch(32)  
  
FEATURE_NAMES = [  
    'age', 'fare', 'sex', 'n_siblings_spouses', 'parch', 'class', 'alone'  
]  
  
feature_columns = []  
for fn in FEATURE_NAMES:  
    feat_col = tf1.feature_column.numeric_column(fn, dtype=tf.float32)  
    feature_columns.append(feat_col)
```

and create a method to instantiate a simplistic sample optimizer to use with our various TensorFlow 1 Estimator and TensorFlow 2 Keras models.

In [ ]:

```
def create_sample_optimizer(tf_version):  
    if tf_version == 'tf1':  
        optimizer = lambda: tf.keras.optimizers.Ftrl(  
            l1_regularization_strength=0.001,  
            learning_rate=tf1.train.exponential_decay(  
                learning_rate=0.1,  
                global_step=tf1.train.get_global_step(),  
                decay_steps=10000,  
                decay_rate=0.9))  
    elif tf_version == 'tf2':  
        optimizer = tf.keras.optimizers.Ftrl(  
            l1_regularization_strength=0.001,  
            learning_rate=tf.keras.optimizers.schedules.ExponentialDecay(  
                initial_learning_rate=0.1, decay_steps=10000, decay_rate=0.9))  
    return optimizer
```

## Example 1: Migrating from LinearEstimator

### TF1: Using LinearEstimator

In TensorFlow 1, you can use `tf.estimator.LinearEstimator` to create a baseline linear model for regression and classification problems.

In [ ]:

```
linear_estimator = tf.estimator.LinearEstimator(  
    head=tf.estimator.BinaryClassHead(),  
    feature_columns=feature_columns,  
    optimizer=create_sample_optimizer('tf1'))
```

In [ ]:

```
linear_estimator.train(input_fn=_input_fn, steps=100)  
linear_estimator.evaluate(input_fn=_eval_input_fn, steps=10)
```

### TF2: Using Keras LinearModel

In TensorFlow 2, you can create an instance of the Keras `tf.compat.v1.keras.models.LinearModel` which is the substitute to the `tf.estimator.LinearEstimator`. The `tf.compat.v1.keras` path is used to signify that the pre-made model exists for compatibility.

In [ ]:

```
linear_model = tf.compat.v1.keras.experimental.LinearModel()  
linear_model.compile(loss='mse', optimizer=create_sample_optimizer('tf2'), metrics=['accuracy'])  
linear_model.fit(x_train, y_train, epochs=10)  
linear_model.evaluate(x_eval, y_eval, return_dict=True)
```

## Example 2: Migrating from DNNEstimator

## TF1: Using DNNEstimator

In TensorFlow 1, you can use `tf.estimator.DNNEstimator` to create a baseline DNN model for regression and classification problems.

In [ ]:

```
dnn_estimator = tf.estimator.DNNEstimator(  
    head=tf.estimator.BinaryClassHead(),  
    feature_columns=feature_columns,  
    hidden_units=[128],  
    activation_fn=tf.nn.relu,  
    optimizer=create_sample_optimizer('tf1'))
```

In [ ]:

```
dnn_estimator.train(input_fn=_input_fn, steps=100)  
dnn_estimator.evaluate(input_fn=_eval_input_fn, steps=10)
```

## TF2: Using Keras to Create a Custom DNN Model

In TensorFlow 2, you can create a custom DNN model to substitute for one generated by `tf.estimator.DNNEstimator`, with similar levels of user-specified customization (for instance, as in the previous example, the ability to customize a chosen model optimizer).

A similar workflow can be used to replace `tf.estimator.experimental.RNNEstimator` with a Keras RNN Model. Keras provides a number of built-in, customizable choices by way of `tf.keras.layers.RNN`, `tf.keras.layers.LSTM`, and `tf.keras.layers.GRU` - see [here](https://www.tensorflow.org/guide/keras/rnn#built-in_rnn_layers_a_simple_example) ([https://www.tensorflow.org/guide/keras/rnn#built-in\\_rnn\\_layers\\_a\\_simple\\_example](https://www.tensorflow.org/guide/keras/rnn#built-in_rnn_layers_a_simple_example)) for more details.

In [ ]:

```
dnn_model = tf.keras.models.Sequential(  
    [tf.keras.layers.Dense(128, activation='relu'),  
     tf.keras.layers.Dense(1)])  
  
dnn_model.compile(loss='mse', optimizer=create_sample_optimizer('tf2'), metrics=['accuracy'])
```

In [ ]:

```
dnn_model.fit(x_train, y_train, epochs=10)  
dnn_model.evaluate(x_eval, y_eval, return_dict=True)
```

## Example 3: Migrating from DNNLinearCombinedEstimator

### TF1: Using DNNLinearCombinedEstimator

In TensorFlow 1, you can use `tf.estimator.DNNLinearCombinedEstimator` to create a baseline combined model for regression and classification problems with customization capacity for both its linear and DNN components.

In [ ]:

```
optimizer = create_sample_optimizer('tf1')  
  
combined_estimator = tf.estimator.DNNLinearCombinedEstimator(  
    head=tf.estimator.BinaryClassHead(),  
    # Wide settings  
    linear_feature_columns=feature_columns,  
    linear_optimizer=optimizer,  
    # Deep settings  
    dnn_feature_columns=feature_columns,  
    dnn_hidden_units=[128],  
    dnn_optimizer=optimizer)
```

In [ ]:

```
combined_estimator.train(input_fn=_input_fn, steps=100)  
combined_estimator.evaluate(input_fn=_eval_input_fn, steps=10)
```

### TF2: Using Keras WideDeepModel

In TensorFlow 2, you can create an instance of the Keras `tf.compat.v1.keras.models.WideDeepModel` to substitute for one generated by `tf.estimator.DNNLinearCombinedEstimator`, with similar levels of user-specified customization (for instance, as in the previous example, the ability to customize a chosen model optimizer).

This `WideDeepModel` is constructed on the basis of a constituent `LinearModel` and a custom DNN Model, both of which are discussed in the preceding two examples. A custom linear model can also be used in place of the built-in Keras `LinearModel` if desired.

If you would like to build your own model instead of a canned estimator, check out [how to build a `keras.Sequential` model](#) ([https://www.tensorflow.org/guide/keras/sequential\\_model](https://www.tensorflow.org/guide/keras/sequential_model)). For more information on custom training and optimizers you can also checkout [this guide](#) ([https://www.tensorflow.org/tutorials/customization/custom\\_training\\_walkthrough](https://www.tensorflow.org/tutorials/customization/custom_training_walkthrough)).

In [ ]:

```
# Create LinearModel and DNN Model as in Examples 1 and 2
optimizer = create_sample_optimizer('tf2')

linear_model = tf.compat.v1.keras.experimental.LinearModel()
linear_model.compile(loss='mse', optimizer=optimizer, metrics=['accuracy'])
linear_model.fit(x_train, y_train, epochs=10, verbose=0)

dnn_model = tf.keras.models.Sequential(
    [tf.keras.layers.Dense(128, activation='relu'),
     tf.keras.layers.Dense(1)])
dnn_model.compile(loss='mse', optimizer=optimizer, metrics=['accuracy'])
```

In [ ]:

```
combined_model = tf.compat.v1.keras.experimental.WideDeepModel(linear_model,
   dnn_model)
combined_model.compile(
    optimizer=[optimizer, optimizer], loss='mse', metrics=['accuracy'])
combined_model.fit([x_train, x_train], y_train, epochs=10)
combined_model.evaluate(x_eval, y_eval, return_dict=True)
```

## Example 4: Migrating from BoostedTreesEstimator

### TF1: Using BoostedTreesEstimator

In TensorFlow 1, you could use `tf.estimator.BoostedTreesEstimator` to create a baseline to create a baseline Gradient Boosting model using an ensemble of decision trees for regression and classification problems. This functionality is no longer included in TensorFlow 2.

```
bt_estimator = tf1.estimator.BoostedTreesEstimator(
    head=tf.estimator.BinaryClassHead(),
    n_batches_per_layer=1,
    max_depth=10,
    n_trees=1000,
    feature_columns=feature_columns)

bt_estimator.train(input_fn=_input_fn, steps=1000)
bt_estimator.evaluate(input_fn=_eval_input_fn, steps=100)
```

### TF2: Using TensorFlow Decision Forests

In TensorFlow 2, `tf.estimator.BoostedTreesEstimator` is replaced by [`tfdf.keras.GradientBoostedTreesModel`](#) ([https://www.tensorflow.org/decision\\_forests/api\\_docs/python/tfdf/keras/GradientBoostedTreesModel#attributes](https://www.tensorflow.org/decision_forests/api_docs/python/tfdf/keras/GradientBoostedTreesModel#attributes)) from the [TensorFlow Decision Forests](#) ([https://www.tensorflow.org/decision\\_forests](https://www.tensorflow.org/decision_forests)) package.

TensorFlow Decision Forests provides various advantages over the `tf.estimator.BoostedTreesEstimator`, notably regarding quality, speed, ease of use and flexibility. To learn about TensorFlow Decision Forests, start with the [beginner colab](#) ([https://www.tensorflow.org/decision\\_forests/tutorials/beginner\\_colab](https://www.tensorflow.org/decision_forests/tutorials/beginner_colab)).

The following example shows how to train a Gradient Boosted Trees model using TensorFlow 2:

Install TensorFlow Decision Forests.

In [ ]:

```
!pip install tensorflow_decision_forests
```

Create a TensorFlow dataset. Note that Decision Forests support natively many types of features and do not need pre-processing.

In [ ]:

```
train_dataframe = pd.read_csv('https://storage.googleapis.com/tf-datasets/titanic/train.csv')
eval_dataframe = pd.read_csv('https://storage.googleapis.com/tf-datasets/titanic/eval.csv')

# Convert the Pandas Dataframes into TensorFlow datasets.
train_dataset = tfdf.keras.pd_dataframe_to_tf_dataset(train_dataframe, label="survived")
eval_dataset = tfdf.keras.pd_dataframe_to_tf_dataset(eval_dataframe, label="survived")
```

Train the model on the `train_dataset` dataset.

In [ ]:

```
# Use the default hyper-parameters of the model.
gbt_model = tfdf.keras.GradientBoostedTreesModel()
gbt_model.fit(train_dataset)
```

Evaluate the quality of the model on the `eval_dataset` dataset.

In [ ]:

```
gbt_model.compile(metrics=['accuracy'])
gbt_evaluation = gbt_model.evaluate(eval_dataset, return_dict=True)
print(gbt_evaluation)
```

Gradient Boosted Trees is just one of the many decision forests algorithms available in TensorFlow Decision Forests. For example, Random Forests (available as [tfdf.keras.GradientBoostedTreesModel](https://www.tensorflow.org/decision_forests/api_docs/python/tfdf/keras/RandomForestModel) ([https://www.tensorflow.org/decision\\_forests/api\\_docs/python/tfdf/keras/RandomForestModel](https://www.tensorflow.org/decision_forests/api_docs/python/tfdf/keras/RandomForestModel)) is very resistant to overfitting) while CART (available as [tfdf.keras.CartModel](https://www.tensorflow.org/decision_forests/api_docs/python/tfdf/keras/CartModel) ([https://www.tensorflow.org/decision\\_forests/api\\_docs/python/tfdf/keras/CartModel](https://www.tensorflow.org/decision_forests/api_docs/python/tfdf/keras/CartModel))) is great for model interpretation.

In the next example, we train and plot a Random Forest model.

In [ ]:

```
# Train a Random Forest model
rf_model = tfdf.keras.RandomForestModel()
rf_model.fit(train_dataset)

# Evaluate the Random Forest model
rf_model.compile(metrics=['accuracy'])
rf_evaluation = rf_model.evaluate(eval_dataset, return_dict=True)
print(rf_evaluation)
```

Finally, in the next example, we train and evaluate a CART model.

In [ ]:

```
# Train a CART model
cart_model = tfdf.keras.CartModel()
cart_model.fit(train_dataset)

# Plot the CART model
tfdf.model_plotter.plot_model_in_colab(cart_model, max_depth=2)
```

Copyright 2021 The TensorFlow Authors.

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Migrate TensorBoard: TensorFlow's visualization toolkit



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/tensorboard)

(<https://www.tensorflow.org/guide/migrate/tensorboard>)



[Run in Google Colab](#)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/tensorboard.ipynb>)

TensorBoard (<https://www.tensorflow.org/tensorboard>) is a built-in tool for providing measurements and visualizations in TensorFlow. Common machine learning experiment metrics, such as accuracy and loss, can be tracked and displayed in TensorBoard. TensorBoard is compatible with TensorFlow 1 and 2 code.

In TensorFlow 1, `tf.estimator.Estimator` saves summaries for TensorBoard by default. In comparison, in TensorFlow 2, summaries can be saved using a `tf.keras.callbacks.TensorBoard` [callback](https://keras.io/api/callbacks/) (<https://keras.io/api/callbacks/>).

This guide demonstrates how to use TensorBoard, first, in TensorFlow 1 with Estimators, and then, how to carry out the equivalent process in TensorFlow 2.

### Setup

In [ ]:

```
import tensorflow.compat.v1 as tf1
import tensorflow as tf
import tempfile
import numpy as np
import datetime
%load_ext tensorboard
```

In [ ]:

```
mnist = tf.keras.datasets.mnist # The MNIST dataset.

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

### TensorFlow 1: TensorBoard with `tf.estimator`

In this TensorFlow 1 example, you instantiate a `tf.estimator.DNNClassifier`, train and evaluate it on the MNIST dataset, and use TensorBoard to display the metrics:

In [ ]:

```
%reload_ext tensorboard

feature_columns = [tf1.feature_column.numeric_column("x", shape=[28, 28])]

config = tf1.estimator.RunConfig(save_summary_steps=1,
                                  save_checkpoints_steps=1)

path = tempfile.mkdtemp()

classifier = tf1.estimator.DNNClassifier(
    feature_columns=feature_columns,
    hidden_units=[256, 32],
    optimizer=tf1.train.AdamOptimizer(0.001),
    n_classes=10,
    dropout=0.1,
    model_dir=path,
    config = config
)

train_input_fn = tf1.estimator.inputs.numpy_input_fn(
    x={"x": x_train},
    y=y_train.astype(np.int32),
    num_epochs=10,
    batch_size=50,
    shuffle=True,
)
test_input_fn = tf1.estimator.inputs.numpy_input_fn(
    x={"x": x_test},
    y=y_test.astype(np.int32),
    num_epochs=10,
    shuffle=False
)
train_spec = tf1.estimator.TrainSpec(input_fn=train_input_fn, max_steps=10)
eval_spec = tf1.estimator.EvalSpec(input_fn=test_input_fn,
                                    steps=10,
                                    throttle_secs=0)

tf1.estimator.train_and_evaluate(estimator=classifier,
                                 train_spec=train_spec,
                                 eval_spec=eval_spec)
```

In [ ]:

```
%tensorboard --logdir {classifier.model_dir}
```

## TensorFlow 2: TensorBoard with a Keras callback and Model.fit

In this TensorFlow 2 example, you create and store logs with the `tf.keras.callbacks.TensorBoard` callback, and train the model. The callback tracks the accuracy and loss per epoch. It is passed to `Model.fit` in the `callbacks` list.

In [ ]:

```
%reload_ext tensorboard

def create_model():
    return tf.keras.models.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(512, activation='relu'),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10, activation='softmax')
    ])

model = create_model()
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'],
              steps_per_execution=10)

log_dir = tempfile.mkdtemp()
tensorboard_callback = tf.keras.callbacks.TensorBoard(
    log_dir=log_dir,
    histogram_freq=1) # Enable histogram computation with each epoch.

model.fit(x=x_train,
          y=y_train,
          epochs=10,
          validation_data=(x_test, y_test),
          callbacks=[tensorboard_callback])
```

In [ ]:

```
%tensorboard --logdir {tensorboard_callback.log_dir}
```

## Next steps

- Learn more about TensorBoard in the [Get started \(\[https://www.tensorflow.org/tensorboard/get\\\_started\]\(https://www.tensorflow.org/tensorboard/get\_started\)\)](https://www.tensorflow.org/tensorboard/get_started) guide.
- For lower level APIs, refer to the [tf.summary migration to TensorFlow 2 \(<https://www.tensorflow.org/tensorboard/migrate>\)](https://www.tensorflow.org/tensorboard/migrate) guide.

Copyright 2021 The TensorFlow Authors.

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Migrate the SavedModel workflow



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/saved_model)

([https://www.tensorflow.org/guide/migrate/saved\\_model](https://www.tensorflow.org/guide/migrate/saved_model))



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/saved_model.ipynb)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/saved\\_model.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/saved_model.ipynb))

Once you have migrated your model from TensorFlow 1's graphs and sessions to TensorFlow 2 APIs, such as `tf.function`, `tf.Module`, and `tf.keras.Model`, you can migrate the model saving and loading code. This notebook provides examples of how you can save and load in the SavedModel format in TensorFlow 1 and TensorFlow 2. Here is a quick overview of the related API changes for migration from TensorFlow 1 to TensorFlow 2:

	TensorFlow 1	Migration to TensorFlow 2
Saving	<code>tf.compat.v1.saved_model.Builder</code> <code>tf.compat.v1.saved_model.simple_save</code>	<code>tf.saved_model.save</code> Keras: <code>tf.keras.models.save_model</code>
Loading	<code>tf.compat.v1.saved_model.load</code>	<code>tf.saved_model.load</code> Keras: <code>tf.keras.models.load_model</code>
Signatures: a set of input and output tensors that can be used to run the	Generated using the <code>*.signature_def</code> utils (e.g. <code>tf.compat.v1.saved_model.predict_signature_def</code> )	Write a <code>tf.function</code> and export it using the <code>signatures</code> argument in <code>tf.saved_model.save</code> .
Classification and regression: special types of signatures	Generated with <code>tf.compat.v1.saved_model.classification_signature_def</code> , <code>tf.compat.v1.saved_model.regression_signature_def</code> , and certain Estimator exports.	These two signature types have been removed from TensorFlow 2. If the serving library requires these method names, <code>tf.compat.v1.saved_model.signature_def_utils.MethodNameUpdater</code> .

For a more in-depth explanation of the mapping, refer to the [Changes from TensorFlow 1 to TensorFlow 2](#) section below.

## Setup

The examples below show how to export and load the same dummy TensorFlow model (defined as `add_two` below) to a SavedModel format using the TensorFlow 1 and TensorFlow 2 APIs. Start by setting up the imports and utility functions:

In [ ]:

```
import tensorflow as tf
import tensorflow.compat.v1 as tf1
import shutil

def remove_dir(path):
    try:
        shutil.rmtree(path)
    except:
        pass

def add_two(input):
    return input + 2
```

## TensorFlow 1: Save and export a SavedModel

In TensorFlow 1, you use the `tf.compat.v1.saved_model.Builder`, `tf.compat.v1.saved_model.simple_save`, and `tf.estimator.Estimator.export_saved_model` APIs to build, save, and export the TensorFlow graph and session:

### 1. Save the graph as a SavedModel with SavedModelBuilder

In [ ]:

```
remove_dir("saved-model-builder")

with tf.Graph().as_default() as g:
    with tf1.Session() as sess:
        input = tf1.placeholder(tf.float32, shape=[])
        output = add_two(input)
        print("add two output: ", sess.run(output, {input: 3.}))

    # Save with SavedModelBuilder
    builder = tf1.saved_model.Builder('saved-model-builder')
    sig_def = tf1.saved_model.predict_signature_def(
        inputs={'input': input},
        outputs={'output': output})
    builder.add_meta_graph_and_variables(
        sess, tags=[ "serve" ], signature_def_map={
            tf.saved_model.DEFAULT_SERVING_SIGNATURE_DEF_KEY: sig_def
        })
    builder.save()
```

In [ ]:

```
!saved_model_cli run --dir saved-model-builder --tag_set serve \
--signature_def serving_default --input_expressions input=10
```

## 2. Build a SavedModel for serving

In [ ]:

```
remove_dir("simple-save")

with tf.Graph().as_default() as g:
    with tf.Session() as sess:
        input = tf.placeholder(tf.float32, shape=[])
        output = add_two(input)
        print("add_two output: ", sess.run(output, {input: 3.}))

    tf1.saved_model.simple_save(
        sess, 'simple-save',
        inputs={'input': input},
        outputs={'output': output})
```

In [ ]:

```
!saved_model_cli run --dir simple-save --tag_set serve \
--signature_def serving_default --input_expressions input=10
```

## 3. Export the Estimator inference graph as a SavedModel

In the definition of the Estimator `model_fn` (defined below), you can define signatures in your model by returning `export_outputs` in the `tf.estimator.EstimatorSpec`. There are different types of outputs:

- `tf.estimator.export.ClassificationOutput`
- `tf.estimator.export.RegressionOutput`
- `tf.estimator.export.PredictOutput`

These will produce classification, regression, and prediction signature types, respectively.

When the estimator is exported with `tf.estimator.Estimator.export_saved_model`, these signatures will be saved with the model.

In [ ]:

```
def model_fn(features, labels, mode):
    output = add_two(features['input'])
    step = tf1.train.get_global_step()
    return tf.estimator.EstimatorSpec(
        mode,
        predictions=output,
        train_op=step.assign_add(1),
        loss=tf.constant(0.),
        export_outputs={
            tf.saved_model.DEFAULT_SERVING_SIGNATURE_DEF_KEY: \
            tf.estimator.export.PredictOutput({'output': output}))}
est = tf.estimator.Estimator(model_fn, 'estimator-checkpoints')

# Train for one step to create a checkpoint.
def train_fn():
    return tf.data.Dataset.from_tensors({'input': 3.})
est.train(train_fn, steps=1)

# This utility function `build_raw_serving_input_receiver_fn` takes in raw
# tensor features and builds an "input serving receiver function", which
# creates placeholder inputs to the model.
serving_input_fn = tf.estimator.export.build_raw_serving_input_receiver_fn(
    {'input': tf.constant(3.)}) # Pass in a dummy input batch.
estimator_path = est.export_saved_model('exported-estimator', serving_input_fn)

# Estimator's export_saved_model creates a time stamped directory. Move this
# to a set path so it can be inspected with `saved_model_cli` in the cell below.
!rm -rf estimator-model
import shutil
shutil.move(estimator_path, 'estimator-model')
```

In [ ]:

```
!saved_model_cli run --dir estimator-model --tag_set serve \
--signature_def serving_default --input_expressions input=[10]
```

## TensorFlow 2: Save and export a SavedModel

### Save and export a SavedModel defined with tf.Module

To export your model in TensorFlow 2, you must define a `tf.Module` or a `tf.keras.Model` to hold all of your model's variables and functions. Then, you can call `tf.saved_model.save` to create a SavedModel. Refer to the *Saving a custom model* section in the [Using the SavedModel format \(..../saved\\_model.ipynb\)](#) guide to learn more.

In [ ]:

```
class MyModel(tf.Module):
    @tf.function
    def __call__(self, input):
        return add_two(input)

model = MyModel()

@tf.function
def serving_default(input):
    return {'output': model(input)}

signature_function = serving_default.get_concrete_function(
    tf.TensorSpec(shape=[], dtype=tf.float32))
tf.saved_model.save(
    model, 'tf2-save', signatures={
        tf.saved_model.DEFAULT_SERVING_SIGNATURE_DEF_KEY: signature_function})
```

In [ ]:

```
!saved_model_cli run --dir tf2-save --tag_set serve
--signature_def serving_default --input_exprs input=10
```

### Save and export a SavedModel defined with Keras

The Keras APIs for saving and exporting—`Model.save` or `tf.keras.models.save_model`—can export a SavedModel from a `tf.keras.Model`. Check out the [Save and load Keras models \(..../guide/keras/save\\_and\\_serialize\)](#) for more details.

In [ ]:

```
inp = tf.keras.Input(3)
out = add_two(inp)
model = tf.keras.Model(inputs=inp, outputs=out)

@tf.function(input_signature=[tf.TensorSpec(shape=[], dtype=tf.float32)])
def serving_default(input):
    return {'output': model(input)}

model.save('keras-model', save_format='tf', signatures={
    tf.saved_model.DEFAULT_SERVING_SIGNATURE_DEF_KEY: serving_default})
```

In [ ]:

```
!saved_model_cli run --dir keras-model --tag_set serve
--signature_def serving_default --input_exprs input=10
```

## Loading a SavedModel

A SavedModel saved with any of the above APIs can be loaded using either TensorFlow 1 or TensorFlow 2 APIs.

A TensorFlow 1 SavedModel can generally be used for inference when loaded into TensorFlow 2, but training (generating gradients) is only possible if the SavedModel contains *resource variables*. You can check the `dtype` of the variables—if the variable `dtype` contains “\_ref”, then it is a reference variable.

A TensorFlow 2 SavedModel can be loaded and executed from TensorFlow 1 as long as the SavedModel is saved with signatures.

The sections below contain code samples showing how to load the SavedModels saved in the previous sections, and call the exported signature.

### TensorFlow 1: Load a SavedModel with `tf.saved_model.load`

In TensorFlow 1, you can import a SavedModel directly into the current graph and session using `tf.saved_model.load`. You can call `Session.run` on the tensor input and output names:

In [ ]:

```
def load_tf1(path, input):
    print('Loading from', path)
    with tf.Graph().as_default() as g:
        with tf1.Session() as sess:
            meta_graph = tf1.saved_model.load(sess, ["serve"], path)
            sig_def = meta_graph.signature_def[tf.saved_model.DEFAULT_SERVING_SIGNATURE_DEF_KEY]
            input_name = sig_def.inputs['input'].name
            output_name = sig_def.outputs['output'].name
            print(' Output with input', input, ':',
                  sess.run(output_name, feed_dict={input_name: input}))

load_tf1('saved-model-builder', 5.)
load_tf1('simple-save', 5.)
load_tf1('estimator-model', [5.]) # Estimator's input must be batched.
load_tf1('tf2-save', 5.)
load_tf1('keras-model', 5.)
```

## TensorFlow 2: Load a model saved with `tf.saved_model`

In TensorFlow 2, objects are loaded into a Python object that stores the variables and functions. This is compatible with models saved from TensorFlow 1.

Check out the `tf.saved_model.load` API docs and [Loading and using a custom model](#)

([./guide/saved\\_model#loading\\_and\\_using\\_a\\_custom\\_model](#)) section from the [Using the SavedModel format](#) ([./guide/saved\\_model](#)) guide for details.

In [ ]:

```
def load_tf2(path, input):
    print('Loading from', path)
    loaded = tf.saved_model.load(path)
    out = loaded.signatures[tf.saved_model.DEFAULT_SERVING_SIGNATURE_DEF_KEY](
        tf.constant(input))['output']
    print(' Output with input', input, ': ', out)

load_tf2('saved-model-builder', 5.)
load_tf2('simple-save', 5.)
load_tf2('estimator-model', [5.]) # Estimator's input must be batched.
load_tf2('tf2-save', 5.)
load_tf2('keras-model', 5.)
```

Models saved with the TensorFlow 2 API can also access `tf.function`s and variables that are attached to the model (instead of those exported as signatures). For example:

In [ ]:

```
loaded = tf.saved_model.load('tf2-save')
print('restored __call__', loaded.__call__)
print('output with input 5.', loaded(5))
```

## TensorFlow 2: Load a model saved with Keras

The Keras loading API—`tf.keras.models.load_model`—allows you to reload a saved model back into a Keras Model object. Note that this only allows you to load SavedModels saved with Keras (`Model.save` or `tf.keras.models.save_model`).

Models saved with `tf.saved_model.save` should be loaded with `tf.saved_model.load`. You can load a Keras model saved with `Model.save` using `tf.saved_model.load` but you will only get the TensorFlow graph. Refer to the `tf.keras.models.load_model` API docs and [Save and load Keras models](#) ([https://www.tensorflow.org/guide/keras/save\\_and\\_serialize#savedmodel\\_format](https://www.tensorflow.org/guide/keras/save_and_serialize#savedmodel_format)) guide for details.

In [ ]:

```
loaded_model = tf.keras.models.load_model('keras-model')
loaded_model.predict_on_batch(tf.constant([1, 3, 4]))
```

## GraphDef and MetaGraphDef

<a name="graphdef\_and\_metagraphdef"> </a>

There is no straightforward way to load a raw `GraphDef` or `MetaGraphDef` to TF2. However, you can convert the TF1 code that imports the graph into a TF2 [concrete\\_function](#) ([https://tensorflow.org/guide/concrete\\_function](https://tensorflow.org/guide/concrete_function)) using `v1.wrap_function` ([https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/wrap\\_function](https://www.tensorflow.org/api_docs/python/tf/compat/v1/wrap_function)).

First, save a MetaGraphDef:

In [ ]:

```
# Save a simple multiplication computation:  
with tf.Graph().as_default() as g:  
    x = tf.placeholder(tf.float32, shape=[], name='x')  
    v = tf.Variable(3.0, name='v')  
    y = tf.multiply(x, v, name='y')  
    with tf.Session() as sess:  
        sess.run(v.initializer)  
        print(sess.run(y, feed_dict={x: 5}))  
    s = tf.train.Saver()  
    s.export_meta_graph('multiply.pb', as_text=True)  
    s.save(sess, 'multiply_values.ckpt')
```

Using TF1 APIs, you can use `tf1.train.import_meta_graph` to import the graph and restore the values:

In [ ]:

```
with tf.Graph().as_default() as g:  
    meta = tf1.train.import_meta_graph('multiply.pb')  
    x = g.get_tensor_by_name('x:0')  
    y = g.get_tensor_by_name('y:0')  
    with tf.Session() as sess:  
        meta.restore(sess, 'multiply_values.ckpt')  
        print(sess.run(y, feed_dict={x: 5}))
```

There are no TF2 APIs for loading the graph, but you can still import it into a concrete function that can be executed in eager mode:

In [ ]:

```
def import_multiply():  
    # Any graph-building code is allowed here.  
    tf1.train.import_meta_graph('multiply.pb')  
  
# Creates a tf.function with all the imported elements in the function graph.  
wrapped_import = tf1.wrap_function(import_multiply, [])  
import_graph = wrapped_import.graph  
x = import_graph.get_tensor_by_name('x:0')  
y = import_graph.get_tensor_by_name('y:0')  
  
# Restore the variable values.  
tf1.train.Saver(wrapped_import.variables).restore(  
    sess=None, save_path='multiply_values.ckpt')  
  
# Create a concrete function by pruning the wrap_function (similar to sess.run).  
multiply_fn = wrapped_import.prune(feeds=x, fetches=y)  
  
# Run this function  
multiply_fn(tf.constant(5.)) # inputs to concrete functions must be Tensors.
```

# Changes from TensorFlow 1 to TensorFlow 2

<a id="changes\_from\_tf1\_to\_tf2"> </a>

This section lists out key saving and loading terms from TensorFlow 1, their TensorFlow 2 equivalents, and what has changed.

## SavedModel

[SavedModel \(./guide/saved\\_model.ipynb\)](#) is a format that stores a complete TensorFlow program with parameters and computation. It contains signatures used by serving platforms to run the model.

The file format itself has not changed significantly, so SavedModels can be loaded and served using either TensorFlow 1 or TensorFlow 2 APIs.

### Differences between TensorFlow 1 and TensorFlow 2

The *serving* and *inference* use cases have not been updated in TensorFlow 2, aside from the API changes—the improvement was introduced in the ability to *reuse* and *compose models* loaded from SavedModel.

In TensorFlow 2, the program is represented by objects like `tf.Variable`, `tf.Module`, or higher-level Keras models (`tf.keras.Model`) and layers (`tf.keras.layers`). There are no more global variables that have values stored in a session, and the graph now exists in different `tf.function`s. Consequently, during a model export, SavedModel saves each component and function graphs separately.

When you write a TensorFlow program with the TensorFlow Python APIs, you must build an object to manage the variables, functions, and other resources. Generally, this is accomplished by using the Keras APIs, but you can also build the object by creating or subclassing `tf.Module`.

Keras models (`tf.keras.Model`) and `tf.Module` automatically track variables and functions attached to them. SavedModel saves these connections between modules, variables, and functions, so that they can be restored when loading.

## Signatures

Signatures are the endpoints of a SavedModel—they tell the user how to run the model and what inputs are needed.

In TensorFlow 1, signatures are created by listing the input and output tensors. In TensorFlow 2, signatures are generated by passing in *concrete functions*. (Read more about TensorFlow functions in the [Introduction to graphs and tf.function \(./intro\\_to\\_graphs.ipynb\)](#) guide, particularly the *Polymorphism: one Function, many graphs* section.) In short, a concrete function is generated from a `tf.function`:

```
# Option 1: Specify an input signature.
@tf.function(input_signature=[...])
def fn(...):
    ...
    return outputs

tf.saved_model.save(model, path, signatures={
    'name': fn
})

# Option 2: Call `get_concrete_function`
@tf.function
def fn(...):
    ...
    return outputs

tf.saved_model.save(model, path, signatures={
    'name': fn.get_concrete_function(...)
})
```

## Session.run

In TensorFlow 1, you could call `Session.run` with the imported graph as long as you already know the tensor names. This allows you to retrieve the restored variable values, or run parts of the model that were not exported in the signatures.

In TensorFlow 2, you can directly access a variable, such as a weights matrix (`kernel`):

```
model = tf.Module()
model.dense_layer = tf.keras.layers.Dense(...)
tf.saved_model.save('my_saved_model')
loaded = tf.saved_model.load('my_saved_model')
loaded.dense_layer.kernel
```

or call `tf.function`s attached to the model object: for example, `loaded.__call__`.

Unlike TF1, there is no way to extract parts of a function and access intermediate values. You *must* export all of the needed functionality in the saved object.

## TensorFlow Serving migration notes

SavedModel was originally created to work with [TensorFlow Serving](https://www.tensorflow.org/tfx/guide/serving) (<https://www.tensorflow.org/tfx/guide/serving>). This platform offers different types of prediction requests: classify, regress, and predict.

The **TensorFlow 1** API allows you to create these types of signatures with the utils:

- `tf.compat.v1.saved_model.classification_signature_def`
- `tf.compat.v1.saved_model.regression_signature_def`
- `tf.compat.v1.saved_model.predict_signature_def`

[Classification](https://www.tensorflow.org/tfx/serving/signature_defs#classification_signaturedef) ([https://www.tensorflow.org/tfx/serving/signature\\_defs#classification\\_signaturedef](https://www.tensorflow.org/tfx/serving/signature_defs#classification_signaturedef)) ( `classification_signature_def` ) and [regression](https://www.tensorflow.org/tfx/serving/signature_defs#regression_signaturedef) ([https://www.tensorflow.org/tfx/serving/signature\\_defs#regression\\_signaturedef](https://www.tensorflow.org/tfx/serving/signature_defs#regression_signaturedef)) ( `regression_signature_def` ) restrict the inputs and outputs, so the inputs must be a `tf.Example`, and the outputs must be `classes`, `scores` or `prediction`. Meanwhile, [the predict signature](https://www.tensorflow.org/tfx/serving/signature_defs#predict_signaturedef) ([https://www.tensorflow.org/tfx/serving/signature\\_defs#predict\\_signaturedef](https://www.tensorflow.org/tfx/serving/signature_defs#predict_signaturedef)) ( `predict_signature_def` ) has no restrictions.

SavedModels exported with the **TensorFlow 2** API are compatible with TensorFlow Serving, but will only contain prediction signatures. The classification and regression signatures have been removed.

If you require the use of the classification and regression signatures, you may modify the exported SavedModel using `tf.compat.v1.saved_model.signature_def_utils.MethodNameUpdater`.

## Next steps

To learn more about SavedModels in TensorFlow 2, check out the following guides:

- [Using the SavedModel format](https://www.tensorflow.org/guide/saved_model) ([https://www.tensorflow.org/guide/saved\\_model](https://www.tensorflow.org/guide/saved_model))
- [Save and load Keras models](https://www.tensorflow.org/guide/keras/save_and_serialize) ([https://www.tensorflow.org/guide/keras/save\\_and\\_serialize](https://www.tensorflow.org/guide/keras/save_and_serialize))

If you are using TensorFlow Hub, you may find these guides useful:

- [TensorFlow Hub: Model compatibility for TensorFlow 1/TensorFlow 2](https://www.tensorflow.org/hub/model_compatibility) ([https://www.tensorflow.org/hub/model\\_compatibility](https://www.tensorflow.org/hub/model_compatibility))
- [Migrating from TensorFlow 1 to TensorFlow 2 with TensorFlow Hub](https://www.tensorflow.org/hub/migration_tf2) ([https://www.tensorflow.org/hub/migration\\_tf2](https://www.tensorflow.org/hub/migration_tf2))

**Copyright 2021 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Migrating your TFLite code to TF2



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/tflite)

(<https://www.tensorflow.org/guide/migrate/tflite>) (<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/tflite.ipynb>) (<https://github.com/tensor>)



[Run in Google Colab](#)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/tflite.ipynb](#)) (<https://github.com/tensor>)

[TensorFlow Lite](https://www.tensorflow.org/lite/guide) (<https://www.tensorflow.org/lite/guide>) (TFLite) is a set of tools that helps developers run ML inference on-device (mobile, embedded, and IoT devices). The [TFLite converter](https://www.tensorflow.org/lite/convert) (<https://www.tensorflow.org/lite/convert>) is one such tool that converts existing TF models into an optimized TFLite model format that can be efficiently run on-device.

In this doc, you'll learn what changes you need to make to your TF to TFLite conversion code, followed by a few examples that do the same.

## Changes to your TF to TFLite conversion code

- If you're using a legacy TF1 model format (Keras file, frozen GraphDef, checkpoints, tf.Session, etc), update it to TF1/TF2 SavedModel and use the TF2 converter API `tf.lite.TFLiteConverter.from_saved_model(...)` to convert it to a TFLite model (refer to Table 1).
- Update the converter API flags (refer to Table 2).
- Remove legacy APIs such as `tf.lite.constants`. (eg: Replace `tf.lite.constants.INT8` with `tf.int8`)

// Table 1 // TFLite Python Converter API Update

TF1 API	TF2 API
<code>tf.lite.TFLiteConverter.from_saved_model('saved_model/...')</code>	<i>supported</i>
<code>tf.lite.TFLiteConverter.from_keras_model_file('model.h5',...)</code>	<i>removed (update to SavedModel format)</i>
<code>tf.lite.TFLiteConverter.from_frozen_graph('model.pb',...)</code>	<i>removed (update to SavedModel format)</i>
<code>tf.lite.TFLiteConverter.from_session(sess,...)</code>	<i>removed (update to SavedModel format)</i>

// Table 2 // TFLite Python Converter API Flags Update

TF1 API	TF2 API
<code>allow_custom_ops</code>	<i>supported</i>
<code>optimizations</code>	
<code>representative_dataset</code>	
<code>target_spec</code>	
<code>inference_input_type</code>	
<code>inference_output_type</code>	
<code>experimental_new_converter</code>	
<code>experimental_new_quantizer</code>	
<code>input_tensors</code>	<i>removed (unsupported converter API arguments)</i>
<code>output_tensors</code>	
<code>input_arrays_with_shape</code>	
<code>output_arrays</code>	
<code>experimental_debug_info_func</code>	
<code>change_concat_input_ranges</code>	<i>removed (unsupported quantization workflows)</i>
<code>default_ranges_stats</code>	
<code>get_input_arrays()</code>	
<code>inference_type</code>	
<code>quantized_input_stats</code>	
<code>reorder_across_fake_quant</code>	
<code>conversion_summary_dir</code>	<i>removed (instead, visualize models using <a href="https://lutzroeder.github.io/netron/">Netron</a> (<a href="https://lutzroeder.github.io/netron/">https://lutzroeder.github.io/netron/</a>) or <a href="https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/tools/visualize.py">visualize.py</a> (<a href="https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/tools/visualize.py">https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/tools/visualize.py</a>))</i>
<code>dump_graphviz_dir</code>	
<code>dump_graphviz_video</code>	
<code>output_format</code>	<i>removed (unsupported features in TF2)</i>
<code>drop_control_dependency</code>	

## Examples

You'll now walkthrough some examples to convert legacy TF1 models to TF1/TF2 SavedModels and then convert it to TF2 TFLite models.

### Setup

Start with the necessary TensorFlow imports.

```
In [ ]:
```

```
import tensorflow as tf
import tensorflow.compat.v1 as tf1
import numpy as np

import logging
logger = tf.get_logger()
logger.setLevel(logging.ERROR)

import shutil
def remove_dir(path):
    try:
        shutil.rmtree(path)
    except:
        pass
```

Create all the necessary TF1 model formats.

```
In [ ]:
```

```
# Create a TF1 SavedModel
SAVED_MODEL_DIR = "tf_saved_model/"
remove_dir(SAVED_MODEL_DIR)
with tf1.Graph().as_default() as g:
    with tf1.Session() as sess:
        input = tf1.placeholder(tf.float32, shape=(3,), name='input')
        output = input + 2
        # print("result: ", sess.run(output, {input: [0., 2., 4.]}))
        tf1.saved_model.simple_save(
            sess, SAVED_MODEL_DIR,
            inputs={'input': input},
            outputs={'output': output})
print("TF1 SavedModel path: ", SAVED_MODEL_DIR)

# Create a TF1 Keras model
KERAS_MODEL_PATH = 'tf_keras_model.h5'
model = tf1.keras.models.Sequential([
    tf1.keras.layers.InputLayer(input_shape=(128, 128, 3,), name='input'),
    tf1.keras.layers.Dense(units=16, input_shape=(128, 128, 3,), activation='relu'),
    tf1.keras.layers.Dense(units=1, name='output')
])
model.save(KERAS_MODEL_PATH, save_format='h5')
print("TF1 Keras Model path: ", KERAS_MODEL_PATH)

# Create a TF1 frozen GraphDef model
GRAPH_DEF_MODEL_PATH = tf.keras.utils.get_file(
    'mobilenet_v1_0.25_128',
    origin='https://storage.googleapis.com/download.tensorflow.org/models/mobilenet_v1_0.25_128_frozen.tgz',
    untar=True,
) + '/frozen_graph.pb'

print("TF1 frozen GraphDef path: ", GRAPH_DEF_MODEL_PATH)
```

## 1. Convert a TF1 SavedModel to a TFLite model

### Before: Converting with TF1

This is typical code for TF1-style TFLite conversion.

```
In [ ]:
```

```
converter = tf1.lite.TFLiteConverter.from_saved_model(
    saved_model_dir=SAVED_MODEL_DIR,
    input_arrays=['input'],
    input_shapes={'input' : [3]})
converter.optimizations = {tf.lite.Optimize.DEFAULT}
converter.change_concat_input_ranges = True
tflite_model = converter.convert()
# Ignore warning: "Use '@tf.function' or '@defun' to decorate the function."
```

### After: Converting with TF2

Directly convert the TF1 SavedModel to a TFLite model, with a smaller v2 converter flags set.

In [ ]:

```
# Convert TF1 SavedModel to a TFLite model.
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir=SAVED_MODEL_DIR)
converter.optimizations = {tf.lite.Optimize.DEFAULT}
tflite_model = converter.convert()
```

## 2. Convert a TF1 Keras model file to a TFLite model

### Before: Converting with TF1

This is typical code for TF1-style TFlite conversion.

In [ ]:

```
converter = tf1.lite.TFLiteConverter.from_keras_model_file(model_file=KERAS_MODEL_PATH)
converter.optimizations = {tf.lite.Optimize.DEFAULT}
converter.change_concat_input_ranges = True
tflite_model = converter.convert()
```

### After: Converting with TF2

First, convert the TF1 Keras model file to a TF2 SavedModel and then convert it to a TFLite model, with a smaller v2 converter flags set.

In [ ]:

```
# Convert TF1 Keras model file to TF2 SavedModel.
model = tf.keras.models.load_model(KERAS_MODEL_PATH)
model.save(filepath='saved_model_2/')

# Convert TF2 SavedModel to a TFLite model.
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir='saved_model_2/')
tflite_model = converter.convert()
```

## 3. Convert a TF1 frozen GraphDef to a TFLite model

### Before: Converting with TF1

This is typical code for TF1-style TFlite conversion.

In [ ]:

```
converter = tf1.lite.TFLiteConverter.from_frozen_graph(
    graph_def_file=GRAPH_DEF_MODEL_PATH,
    input_arrays=['input'],
    input_shapes={'input' : [1, 128, 128, 3]},
    output_arrays=['MobilenetV1/Predictions/Softmax'],
)
converter.optimizations = {tf.lite.Optimize.DEFAULT}
converter.change_concat_input_ranges = True
tflite_model = converter.convert()
```

### After: Converting with TF2

First, convert the TF1 frozen GraphDef to a TF1 SavedModel and then convert it to a TFLite model, with a smaller v2 converter flags set.

In [ ]:

```
## Convert TF1 frozen Graph to TF1 SavedModel.

# Load the graph as a v1.GraphDef
import pathlib
gdef = tf.compat.v1.GraphDef()
gdef.ParseFromString(pathlib.Path(GRAPH_DEF_MODEL_PATH).read_bytes())

# Convert the GraphDef to a tf.Graph
with tf.Graph().as_default() as g:
    tf.graph_util.import_graph_def(gdef, name="")

# Lookup the input and output tensors.
input_tensor = g.get_tensor_by_name('input:0')
output_tensor = g.get_tensor_by_name('MobilenetV1/Predictions/Softmax:0')

# Save the graph as a TF1 Savedmodel
remove_dir('saved_model_3/')
with tf.compat.v1.Session(graph=g) as s:
    tf.compat.v1.saved_model.simple_save(
        session=s,
        export_dir='saved_model_3/',
        inputs={'input':input_tensor},
        outputs={'output':output_tensor})

# Convert TF1 SavedModel to a TFLite model.
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir='saved_model_3/')
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()
```

## Further reading

- Refer to the [TFLite Guide](https://www.tensorflow.org/lite/guide) (<https://www.tensorflow.org/lite/guide>) to learn more about the workflows and latest features.
- If you're using TF1 code or legacy TF1 model formats (Keras .h5 files, frozen GraphDef .pb , etc), please update your code and migrate your models to the [TF2 SavedModel format](https://www.tensorflow.org/guide/saved_model) ([https://www.tensorflow.org/guide/saved\\_model](https://www.tensorflow.org/guide/saved_model)).

Copyright 2021 The TensorFlow Authors.

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Debug TF2 Migrated Training Pipeline



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/migration_debugging)

([https://www.tensorflow.org/guide/migrate/migration\\_debugging](https://www.tensorflow.org/guide/migrate/migration_debugging))



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/migration_debugging.ipynb)

This notebook demonstrates how to debug training pipeline when migrating to TF2. It consists of following components:

1. Suggested steps and code samples for debugging training pipeline
2. Tools for debugging
3. Other related resources

One assumption is you have TF1.x code and trained models for comparison, and you want to build a TF2 model that achieves similar validation accuracy.

This notebook does **NOT** cover debugging performance issues for training/inference speed or memory usage.

## Debugging workflow

Below is a general workflow for debugging your TF2 training pipelines. Note that you do not need to follow these steps in order. You can also use a binary search approach where you test the model in an intermediate step and narrow down the debugging scope.

1. Fix compile and runtime errors
2. Single forward pass validation (in a separate [guide \(./validate\\_correctness.ipynb\)](#))
  - a. On single CPU device
    - Verify variables are created only once
    - Check variable counts, names, and shapes match
    - Reset all variables, check numerical equivalence with all randomness disabled
    - Align random number generation, check numerical equivalence in inference
    - (Optional) Check checkpoints are loaded properly and TF1.x/TF2 models generate identical output
  - b. On single GPU/TPU device
  - c. With multi-device strategies
3. Model training numerical equivalence validation for a few steps (code samples available below)
  - a. Single training step validation using small and fixed data on single CPU device. Specifically, check numerical equivalence for the following components
    - losses computation
    - metrics
    - learning rate
    - gradient computation and update
  - b. Check statistics after training 3 or more steps to verify optimizer behaviors like the momentum, still with fixed data on single CPU device
  - c. On single GPU/TPU device
  - d. With multi-device strategies (check the intro for [MultiProcessRunner](#) ([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/distribute/multi\\_process\\_runner.py#L108](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/distribute/multi_process_runner.py#L108)) at the bottom)
4. End-to-end convergence testing on real dataset
  - a. Check training behaviors with TensorBoard
    - use simple optimizers e.g. SGD and simple distribution strategies e.g. `tf.distribute.OneDeviceStrategy` first
    - training metrics
    - evaluation metrics
    - figure out what the reasonable tolerance for inherent randomness is
  - b. Check equivalence with advanced optimizer/learning rate scheduler/distribution strategies
  - c. Check equivalence when using mixed precision
5. Additional product benchmarks

## Setup

In [ ]:

```
!pip uninstall -y -q tensorflow
```

In [ ]:

```
# Install tf-nightly as the DeterministicRandomTestTool is only available in
# Tensorflow 2.8
!pip install -q tf-nightly
```

## Single forward pass validation

Single forward pass validation, including checkpoint loading, is covered in a different [colab \(./validate\\_correctness.ipynb\)](#).

In [ ]:

```
import sys
import unittest
import numpy as np

import tensorflow as tf
import tensorflow.compat.v1 as v1
```

## Model training numerical equivalence validation for a few steps

Set up model configuration and prepare a fake dataset.

In [ ]:

```
params = {  
    'input_size': 3,  
    'num_classes': 3,  
    'layer_1_size': 2,  
    'layer_2_size': 2,  
    'num_train_steps': 100,  
    'init_lr': 1e-3,  
    'end_lr': 0.0,  
    'decay_steps': 1000,  
    'lr_power': 1.0,  
}  
  
# make a small fixed dataset  
fake_x = np.ones((2, params['input_size']), dtype=np.float32)  
fake_y = np.zeros((2, params['num_classes']), dtype=np.int32)  
fake_y[0][0] = 1  
fake_y[1][1] = 1  
  
step_num = 3
```

Define the TF1.x model.

In [ ]:

```
# Assume there is an existing TF1.x model using estimator API
# Wrap the model_fn to log necessary tensors for result comparison
class SimpleModelWrapper():
    def __init__(self):
        self.logged_ops = {}
        self.logs = {
            'step': [],
            'lr': [],
            'loss': [],
            'grads_and_vars': [],
            'layer_out': []}

    def model_fn(self, features, labels, mode, params):
        out_1 = tf.compat.v1.layers.dense(features, units=params['layer_1_size'])
        out_2 = tf.compat.v1.layers.dense(out_1, units=params['layer_2_size'])
        logits = tf.compat.v1.layers.dense(out_2, units=params['num_classes'])
        loss = tf.compat.v1.losses.softmax_cross_entropy(labels, logits)

        # skip EstimatorSpec details for prediction and evaluation
        if mode == tf.estimator.ModeKeys.PREDICT:
            pass
        if mode == tf.estimator.ModeKeys.EVAL:
            pass
        assert mode == tf.estimator.ModeKeys.TRAIN

        global_step = tf.compat.v1.train.get_or_create_global_step()
        lr = tf.compat.v1.train.polynomial_decay(
            learning_rate=params['init_lr'],
            global_step=global_step,
            decay_steps=params['decay_steps'],
            end_learning_rate=params['end_lr'],
            power=params['lr_power'])

        optmizer = tf.compat.v1.train.GradientDescentOptimizer(lr)
        grads_and_vars = optmizer.compute_gradients(
            loss=loss,
            var_list=graph.get_collection(
                tf.compat.v1.GraphKeys.TRAINABLE_VARIABLES))
        train_op = optmizer.apply_gradients(
            grads_and_vars,
            global_step=global_step)

        # log tensors
        self.logged_ops['step'] = global_step
        self.logged_ops['lr'] = lr
        self.logged_ops['loss'] = loss
        self.logged_ops['grads_and_vars'] = grads_and_vars
        self.logged_ops['layer_out'] = {
            'layer_1': out_1,
            'layer_2': out_2,
            'logits': logits}

    return tf.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)

    def update_logs(self, logs):
        for key in logs.keys():
            model_tf1.logs[key].append(logs[key])
```

The following `v1.keras.utils.DeterministicRandomTestTool`

([https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/keras/utils/DeterministicRandomTestTool](https://www.tensorflow.org/api_docs/python/tf/compat/v1/keras/utils/DeterministicRandomTestTool)) class provides a context manager `scope()` that can make stateful random operations use the same seed across both TF1 graphs/sessions and eager execution,

The tool provides two testing modes:

1. `constant` which uses the same seed for every single operation no matter how many times it has been called and,
2. `num_random_ops` which uses the number of previously-observed stateful random operations as the operation seed.

This applies both to the stateful random operations used for creating and initializing variables, and to the stateful random operations used in computation (such as for dropout layers).

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
```

Run the TF1.x model in graph mode. Collect statistics for first 3 training steps for numerical equivalence comparison.

In [ ]:

```
with random_tool.scope():
    graph = tf.Graph()
    with graph.as_default(), tf.compat.v1.Session(graph=graph) as sess:
        model_tf1 = SimpleModelWrapper()
        # build the model
        inputs = tf.compat.v1.placeholder(tf.float32, shape=(None, params['input_size']))
        labels = tf.compat.v1.placeholder(tf.float32, shape=(None, params['num_classes']))
        spec = model_tf1.model_fn(inputs, labels, tf.estimator.ModeKeys.TRAIN, params)
        train_op = spec.train_op

    sess.run(tf.compat.v1.global_variables_initializer())
    for step in range(step_num):
        # log everything and update the model for one step
        logs, _ = sess.run(
            [model_tf1.logged_ops, train_op],
            feed_dict={inputs: fake_x, labels: fake_y})
        model_tf1.update_logs(logs)
```

Define the TF2 model.

In [ ]:

```
class SimpleModel(tf.keras.Model):
    def __init__(self, params, *args, **kwargs):
        super(SimpleModel, self).__init__(*args, **kwargs)
        # define the model
        self.dense_1 = tf.keras.layers.Dense(params['layer_1_size'])
        self.dense_2 = tf.keras.layers.Dense(params['layer_2_size'])
        self.out = tf.keras.layers.Dense(params['num_classes'])
        learning_rate_fn = tf.keras.optimizers.schedules.PolynomialDecay(
            initial_learning_rate=params['init_lr'],
            decay_steps=params['decay_steps'],
            end_learning_rate=params['end_lr'],
            power=params['lr_power'])
        self.optimizer = tf.keras.optimizers.SGD(learning_rate_fn)
        self.compiled_loss = tf.keras.losses.CategoricalCrossentropy(from_logits=True)
        self.logs = {
            'lr': [],
            'loss': [],
            'grads': [],
            'weights': [],
            'layer_out': []}

    def call(self, inputs):
        out_1 = self.dense_1(inputs)
        out_2 = self.dense_2(out_1)
        logits = self.out(out_2)
        # log output features for every layer for comparison
        layer_wise_out = {
            'layer_1': out_1,
            'layer_2': out_2,
            'logits': logits}
        self.logs['layer_out'].append(layer_wise_out)
        return logits

    def train_step(self, data):
        x, y = data
        with tf.GradientTape() as tape:
            logits = self(x)
            loss = self.compiled_loss(y, logits)
            grads = tape.gradient(loss, self.trainable_weights)
        # log training statistics
        step = self.optimizer.iterations.numpy()
        self.logs['lr'].append(self.optimizer.learning_rate(step).numpy())
        self.logs['loss'].append(loss.numpy())
        self.logs['grads'].append(grads)
        self.logs['weights'].append(self.trainable_weights)
        # update model
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        return
```

Run the TF2 model in eager mode. Collect statistics for first 3 training steps for numerical equivalence comparison.

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    model_tf2 = SimpleModel(params)
    for step in range(step_num):
        model_tf2.train_step([fake_x, fake_y])
```

Compare numerical equivalence for first few training steps.

You can also check the [Validating correctness & numerical equivalence notebook \(./validate\\_correctness.ipynb\)](#) for additional advice for numerical equivalence.

In [ ]:

```
np.testing.assert_allclose(model_tf1.logs['lr'], model_tf2.logs['lr'])
np.testing.assert_allclose(model_tf1.logs['loss'], model_tf2.logs['loss'])
for step in range(step_num):
    for name in model_tf1.logs['layer_out'][step]:
        np.testing.assert_allclose(
            model_tf1.logs['layer_out'][step][name],
            model_tf2.logs['layer_out'][step][name])
```

## Unit tests

There are a few types of unit testing that can help debug your migration code.

1. Single forward pass validation
2. Model training numerical equivalence validation for a few steps
3. Benchmark inference performance
4. The trained model makes correct predictions on fixed and simple data points

You can use `@parameterized.parameters` to test models with different configurations. [Details with code sample \(<https://github.com/abseil/abseil-py/blob/master/absl/testing/parameterized.py>\)](#).

Note that it's possible to run session APIs and eager execution in the same test case. The code snippets below show how.

In [ ]:

```
import unittest

class TestNumericalEquivalence(unittest.TestCase):

    # copied from code samples above
    def setup(self):
        # record statistics for 100 training steps
        step_num = 100

        # setup TF 1 model
        random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
        with random_tool.scope():
            # run TF1.x code in graph mode with context management
            graph = tf.Graph()
            with graph.as_default(), tf.compat.v1.Session(graph=graph) as sess:
                self.model_tf1 = SimpleModelWrapper()
                # build the model
                inputs = tf.compat.v1.placeholder(tf.float32, shape=(None, params['input_size']))
                labels = tf.compat.v1.placeholder(tf.float32, shape=(None, params['num_classes']))
                spec = self.model_tf1.model_fn(inputs, labels, tf.estimator.ModeKeys.TRAIN, params)
                train_op = spec.train_op

                sess.run(tf.compat.v1.global_variables_initializer())
                for step in range(step_num):
                    # log everything and update the model for one step
                    logs, _ = sess.run(
                        [self.model_tf1.logged_ops, train_op],
                        feed_dict={inputs: fake_x, labels: fake_y})
                    self.model_tf1.update_logs(logs)

    # setup TF2 model
    random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
    with random_tool.scope():
        self.model_tf2 = SimpleModel(params)
        for step in range(step_num):
            self.model_tf2.train_step([fake_x, fake_y])

    def test_learning_rate(self):
        np.testing.assert_allclose(
            self.model_tf1.logs['lr'],
            self.model_tf2.logs['lr'])

    def test_training_loss(self):
        # adopt different tolerance strategies before and after 10 steps
        first_n_step = 10

        # absolute difference is limited below 1e-5
        # set `equal_nan` to be False to detect potential NaN loss issues
        absolute_tolerance = 1e-5
        np.testing.assert_allclose(
            actual=self.model_tf1.logs['loss'][:first_n_step],
            desired=self.model_tf2.logs['loss'][:first_n_step],
            atol=absolute_tolerance,
            equal_nan=False)

        # relative difference is limited below 5%
        relative_tolerance = 0.05
        np.testing.assert_allclose(self.model_tf1.logs['loss'][first_n_step:],
            self.model_tf2.logs['loss'][first_n_step:],
            rtol=relative_tolerance,
            equal_nan=False)
```

## Debugging tools

## tf.print

tf.print vs print/logging.info

- With configurable arguments, `tf.print` can recursively display show first and last few elements of each dimension for printed tensors. Check the [API docs](https://www.tensorflow.org/api_docs/python/tf/print) ([https://www.tensorflow.org/api\\_docs/python/tf/print](https://www.tensorflow.org/api_docs/python/tf/print)) for details.
- For eager execution, both `print` and `tf.print` print the value of the tensor. But `print` may involve device-to-host copy, which can potentially slow down your code.
- For graph mode including usage inside `tf.function`, you need to use `tf.print` to print the actual tensor value. `tf.print` is compiled into an op in the graph, whereas `print` and `logging.info` only log at tracing time, which is often not what you want.
- `tf.print` also supports printing composite tensors like `tf.RaggedTensor` and `tf.sparse.SparseTensor`.
- You can also use a callback to monitor metrics and variables. Please check how to use custom callbacks with [logs dict](https://www.tensorflow.org/guide/keras/custom_callback#usage_of_logs_dict) ([https://www.tensorflow.org/guide/keras/custom\\_callback#usage\\_of\\_logs\\_dict](https://www.tensorflow.org/guide/keras/custom_callback#usage_of_logs_dict)) and [self.model attribute](https://www.tensorflow.org/guide/keras/custom_callback#usage_of_selfmodel_attribute) ([https://www.tensorflow.org/guide/keras/custom\\_callback#usage\\_of\\_selfmodel\\_attribute](https://www.tensorflow.org/guide/keras/custom_callback#usage_of_selfmodel_attribute)).

tf.print vs print inside tf.function

In [ ]:

```
# `print` prints info of tensor object
# `tf.print` prints the tensor value
@tf.function
def dummy_func(num):
    num += 1
    print(num)
    tf.print(num)
    return num

_ = dummy_func(tf.constant([1.0]))

# Output:
# Tensor("add:0", shape=(1,), dtype=float32)
# [2]
```

tf.distribute.Strategy

- If the `tf.function` containing `tf.print` is executed on the workers, for example when using `TPUStrategy` or `ParameterServerStrategy`, you need to check worker/parameter server logs to find the printed values.
- For `print` or `logging.info`, logs will be printed on the coordinator when using `ParameterServerStrategy`, and logs will be printed on the `STDOUT` on worker0 when using TPUs.

tf.keras.Model

- When using Sequential and Functional API models, if you want to print values, e.g., model inputs or intermediate features after some layers, you have following options.
  - [Write a custom layer](https://www.tensorflow.org/guide/keras/custom_layers_and_models) ([https://www.tensorflow.org/guide/keras/custom\\_layers\\_and\\_models](https://www.tensorflow.org/guide/keras/custom_layers_and_models)) that `tf.print` the inputs.
  - Include the intermediate outputs you want to inspect in the model outputs.
- `tf.keras.layers.Lambda` layers have (de)serialization limitations. To avoid checkpoint loading issues, write a custom subclassed layer instead. Check the [API docs](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Lambda) ([https://www.tensorflow.org/api\\_docs/python/tf/keras/layers/Lambda](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Lambda)) for more details.
- You can't `tf.print` intermediate outputs in a `tf.keras.callbacks.LambdaCallback` if you don't have access to the actual values, but instead only to the symbolic Keras tensor objects.

Option 1: write a custom layer

In [ ]:

```
class PrintLayer(tf.keras.layers.Layer):
    def call(self, inputs):
        tf.print(inputs)
        return inputs

    def get_model():
        inputs = tf.keras.layers.Input(shape=(1,))
        out_1 = tf.keras.layers.Dense(4)(inputs)
        out_2 = tf.keras.layers.Dense(1)(out_1)
        # use custom layer to tf.print intermediate features
        out_3 = PrintLayer()(out_2)
        model = tf.keras.Model(inputs=inputs, outputs=out_3)
        return model

model = get_model()
model.compile(optimizer="adam", loss="mse")
model.fit([1, 2, 3], [0.0, 0.0, 1.0])
```

Option 2: include the intermediate outputs you want to inspect in the model outputs.

Note that in such case, you may need some [customizations](https://www.tensorflow.org/guide/keras/customizing_what_happens_in_fit) ([https://www.tensorflow.org/guide/keras/customizing\\_what\\_happens\\_in\\_fit](https://www.tensorflow.org/guide/keras/customizing_what_happens_in_fit)) to use `Model.fit`.

In [ ]:

```
def get_model():
    inputs = tf.keras.layers.Input(shape=(1,))
    out_1 = tf.keras.layers.Dense(4)(inputs)
    out_2 = tf.keras.layers.Dense(1)(out_1)
    # include intermediate values in model outputs
    model = tf.keras.Model(
        inputs=inputs,
        outputs={
            'inputs': inputs,
            'out_1': out_1,
            'out_2': out_2})
    return model
```

## pdb

You can use [pdb](https://docs.python.org/3/library/pdb.html) (<https://docs.python.org/3/library/pdb.html>) both in terminal and Colab to inspect intermediate values for debugging.

## Visualize graph with TensorBoard

You can [examine the TensorFlow graph with TensorBoard](https://www.tensorflow.org/tensorboard/graphs) (<https://www.tensorflow.org/tensorboard/graphs>). TensorBoard is also [supported on colab](https://www.tensorflow.org/tensorboard/tensorboard_in_notebooks) ([https://www.tensorflow.org/tensorboard/tensorboard\\_in\\_notebooks](https://www.tensorflow.org/tensorboard/tensorboard_in_notebooks)). TensorBoard is a great tool to visualize summaries. You can use it to compare learning rate, model weights, gradient scale, training/validation metrics, or even model intermediate outputs between TF1.x model and migrated TF2 model through the training process and seeing if the values look as expected.

## TensorFlow Profiler

[TensorFlow Profiler](https://www.tensorflow.org/guide/profiler) (<https://www.tensorflow.org/guide/profiler>) can help you visualize the execution timeline on GPUs/TPUs. You can check out this [Colab Demo](https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras) ([https://www.tensorflow.org/tensorboard/tensorboard\\_profiling\\_keras](https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras)) for its basic usage.

## MultiProcessRunner

[MultiProcessRunner](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/distribute/multi_process_runner.py#L108) ([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/distribute/multi\\_process\\_runner.py#L108](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/distribute/multi_process_runner.py#L108)) is a useful tool when debugging with MultiWorkerMirroredStrategy and ParameterServerStrategy. You can take a look at [this concrete example](https://github.com/keras-team/keras/blob/master/keras/integration_test/mwms_multi_process_runner_test.py) ([https://github.com/keras-team/keras/blob/master/keras/integration\\_test/mwms\\_multi\\_process\\_runner\\_test.py](https://github.com/keras-team/keras/blob/master/keras/integration_test/mwms_multi_process_runner_test.py)) for its usage.

Specifically for the cases of these two strategies, you are recommended to 1) not only have unit tests to cover their flow, 2) but also to attempt to reproduce failures using it in unit test to avoid launch real distributed job every time when they attempt a fix.

Copyright 2021 The TensorFlow Authors.

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Migrate from TPU embedding\_columns to TPUEmbedding layer



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/tpu_embedding)

([https://www.tensorflow.org/guide/migrate/tpu\\_embedding](https://www.tensorflow.org/guide/migrate/tpu_embedding))



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/tpu_embedding.ipynb)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/tpu\\_embedding.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/tpu_embedding.ipynb))

This guide demonstrates how to migrate embedding training on on [TPUs \(../guide/tpu.ipynb\)](#) from TensorFlow 1's `embedding_column` API with `TPUEstimator` to TensorFlow 2's `TPUEmbedding` layer API with `TPUStrategy`.

Embeddings are (large) matrices. They are lookup tables that map from a sparse feature space to dense vectors. Embeddings provide efficient and dense representations, capturing complex similarities and relationships between features.

TensorFlow includes specialized support for training embeddings on TPUs. This TPU-specific embedding support allows you to train embeddings that are larger than the memory of a single TPU device, and to use sparse and ragged inputs on TPUs.

- In TensorFlow 1, `tf.compat.v1.estimator.tpu.TPUEstimator` is a high level API that encapsulates training, evaluation, prediction, and exporting for serving with TPUs. It has special support for `tf.compat.v1.tpu.experimental.embedding_column`.
- To implement this in TensorFlow 2, use the TensorFlow Recommenders' `tfrs.layers.embedding.TPUEmbedding` layer. For training and evaluation, use a TPU distribution strategy—`tf.distribute.TPUStrategy`—which is compatible with the Keras APIs for, for example, model building (`tf.keras.Model`), optimizers (`tf.keras.optimizers.Optimizer`), and training with `Model.fit` or a custom training loop with `tf.function` and `tf.GradientTape`.

For additional information, refer to the `tfrs.layers.embedding.TPUEmbedding` layer's API documentation, as well as the `tf.tpu.experimental.embedding.TableConfig` and `tf.tpu.experimental.embedding.FeatureConfig` docs for additional information. For an overview of `tf.distribute.TPUStrategy`, check out the [Distributed training \(../guide/distributed\\_training.ipynb\)](#) guide and the [Use TPUs \(../guide/tpu.ipynb\)](#) guide. If you're migrating from `TPUEstimator` to `TPUStrategy`, check out [the TPU migration guide \(tpu\\_estimator.ipynb\)](#).

## Setup

Start by installing [TensorFlow Recommenders](#) (<https://www.tensorflow.org/recommenders>) and importing some necessary packages:

In [ ]:

```
!pip install tensorflow-recommenders
```

In [ ]:

```
import tensorflow as tf
import tensorflow.compat.v1 as tf1

# TPUEmbedding layer is not part of TensorFlow.
import tensorflow_recommenders as tfrs
```

And prepare a simple dataset for demonstration purposes:

In [ ]:

```
features = [[1., 1.5]]
embedding_features_indices = [[0, 0], [0, 1]]
embedding_features_values = [0, 5]
labels = [[0.3]]
eval_features = [[4., 4.5]]
eval_embedding_features_indices = [[0, 0], [0, 1]]
eval_embedding_features_values = [4, 3]
eval_labels = [[0.8]]
```

## TensorFlow 1: Train embeddings on TPUs with TPUEstimator

In TensorFlow 1, you set up TPU embeddings using the `tf.compat.v1.tpu.experimental.embedding_column` API and train/evaluate the model on TPUs with `tf.compat.v1.estimator.tpu.TPUEstimator`.

The inputs are integers ranging from zero to the vocabulary size for the TPU embedding table. Begin with encoding the inputs to categorical ID with `tf.feature_column.categorical_column_with_identity`. Use "sparse\_feature" for the `key` parameter, since the input features are integer-valued, while `num_buckets` is the vocabulary size for the embedding table ( 10 ).

In [ ]:

```
embedding_id_column = (
    tf1.feature_column.categorical_column_with_identity(
        key="sparse_feature", num_buckets=10))
```

Next, convert the sparse categorical inputs to a dense representation with `tpu.experimental.embedding_column`, where `dimension` is the width of the embedding table. It will store an embedding vector for each of the `num_buckets`.

In [ ]:

```
embedding_column = tf1.tpu.experimental.embedding_column(
    embedding_id_column, dimension=5)
```

Now, define the TPU-specific embedding configuration via `tf.estimator.tpu.experimental.EmbeddingConfigSpec`. You will pass it later to `tf.estimator.tpu.TPUEstimator` as an `embedding_config_spec` parameter.

In [ ]:

```
embedding_config_spec = tf1.estimator.tpu.experimental.EmbeddingConfigSpec(
    feature_columns=(embedding_column,),
    optimization_parameters=(
        tf1.tpu.experimental.AdagradParameters(0.05)))
```

Next, to use a `TPUEstimator`, define:

- An input function for the training data
- An evaluation input function for the evaluation data
- A model function for instructing the `TPUEstimator` how the training op is defined with the features and labels

In [ ]:

```
def _input_fn(params):
    dataset = tf1.data.Dataset.from_tensor_slices((
        {"dense_feature": features,
         "sparse_feature": tf1.SparseTensor(
             embedding_features_indices,
             embedding_features_values, [1, 2])},
        labels))
    dataset = dataset.repeat()
    return dataset.batch(params['batch_size'], drop_remainder=True)

def _eval_input_fn(params):
    dataset = tf1.data.Dataset.from_tensor_slices((
        {"dense_feature": eval_features,
         "sparse_feature": tf1.SparseTensor(
             eval_embedding_features_indices,
             eval_embedding_features_values, [1, 2])},
        eval_labels))
    dataset = dataset.repeat()
    return dataset.batch(params['batch_size'], drop_remainder=True)

def _model_fn(features, labels, mode, params):
    embedding_features = tf1.keras.layers.DenseFeatures(embedding_column)(features)
    concatenated_features = tf1.keras.layers.concatenate(axis=1)(
        [embedding_features, features["dense_feature"]])
    logits = tf1.layers.Dense(1)(concatenated_features)
    loss = tf1.losses.mean_squared_error(labels=labels, predictions=logits)
    optimizer = tf1.train.AdagradOptimizer(0.05)
    optimizer = tf1.tpu.CrossShardOptimizer(optimizer)
    train_op = optimizer.minimize(loss, global_step=tf1.train.get_global_step())
    return tf1.estimator.tpu.TPUEstimatorSpec(mode, loss=loss, train_op=train_op)
```

With those functions defined, create a `tf.distribute.cluster_resolver.TPUClusterResolver` that provides the cluster information, and a `tf.compat.v1.estimator.tpu.RunConfig` object.

Along with the model function you have defined, you can now create a `TPUEstimator`. Here, you will simplify the flow by skipping checkpoint savings. Then, you will specify the batch size for both training and evaluation for the `TPUEstimator`.

In [ ]:

```
cluster_resolver = tf1.distribute.cluster_resolver.TPUClusterResolver(tpu='')
print("All devices: ", tf1.config.list_logical_devices('TPU'))
```

In [ ]:

```
tpu_config = tf1.estimator.tpu.TPUEConfig(
    iterations_per_loop=10,
    per_host_input_for_training=tf1.estimator.tpu.InputPipelineConfig
        .PER_HOST_V2)
config = tf1.estimator.tpu.RunConfig(
    cluster=cluster_resolver,
    save_checkpoints_steps=None,
    tpu_config=tpu_config)
estimator = tf1.estimator.tpu.TPUEstimator(
    model_fn=_model_fn, config=config, train_batch_size=8, eval_batch_size=8,
    embedding_config_spec=embedding_config_spec)
```

Call `TPUEstimator.train` to begin training the model:

In [ ]:

```
estimator.train(_input_fn, steps=1)
```

Then, call `TPUEstimator.evaluate` to evaluate the model using the evaluation data:

In [ ]:

```
estimator.evaluate(_eval_input_fn, steps=1)
```

## TensorFlow 2: Train embeddings on TPUs with TPUStrategy

In TensorFlow 2, to train on the TPU workers, use `tf.distribute.TPUStrategy` together with the Keras APIs for model definition and training/evaluation. (Refer to the [Use TPUs](https://render.githubusercontent.com/guide/tpu.ipynb) (<https://render.githubusercontent.com/guide/tpu.ipynb>) guide for more examples of training with Keras Model.fit and a custom training loop (with `tf.function` and `tf.GradientTape` ).)

Since you need to perform some initialization work to connect to the remote cluster and initialize the TPU workers, start by creating a `TPUClusterResolver` to provide the cluster information and connect to the cluster. (Learn more in the *TPU initialization* section of the [Use TPUs](#) ([./guide/tpu.ipynb](#)) guide.)

In [ ]:

```
cluster_resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='''')
tf.config.experimental_connect_to_cluster(cluster_resolver)
tf.tpu.experimental.initialize_tpu_system(cluster_resolver)
print("All devices: ", tf.config.list_logical_devices('TPU'))
```

Next, prepare your data. This is similar to how you created a dataset in the TensorFlow 1 example, except the `dataset` function is now passed a `tf.distribute.InputContext` object rather than a `params` dict. You can use this object to determine the local batch size (and which host this pipeline is for, so you can properly partition your data).

- When using the `tfrs.layers.embedding.TPUEmbedding` API, it is important to include the `drop_remainder=True` option when batching the dataset with `Dataset.batch`, since `TPUEmbedding` requires a fixed batch size.
- Additionally, the same batch size must be used for evaluation and training if they are taking place on the same set of devices.
- Finally, you should use `tf.keras.utils.experimental.DatasetCreator` along with the special input option — `experimental_fetch_to_device=False` — in `tf.distribute.InputOptions` (which holds strategy-specific configurations). This is demonstrated below:

In [ ]:

```
global_batch_size = 8

def _input_dataset(context: tf.distribute.InputContext):
    dataset = tf.data.Dataset.from_tensor_slices({
        "dense_feature": features,
        "sparse_feature": tf.SparseTensor(
            embedding_features_indices,
            embedding_features_values, [1, 2]),
        labels))
    dataset = dataset.shuffle(10).repeat()
    dataset = dataset.batch(
        context.get_per_replica_batch_size(global_batch_size),
        drop_remainder=True)
    return dataset.prefetch(2)

def _eval_dataset(context: tf.distribute.InputContext):
    dataset = tf.data.Dataset.from_tensor_slices({
        "dense_feature": eval_features,
        "sparse_feature": tf.SparseTensor(
            eval_embedding_features_indices,
            eval_embedding_features_values, [1, 2]),
        eval_labels))
    dataset = dataset.repeat()
    dataset = dataset.batch(
        context.get_per_replica_batch_size(global_batch_size),
        drop_remainder=True)
    return dataset.prefetch(2)

input_options = tf.distribute.InputOptions(
    experimental_fetch_to_device=False)

input_dataset = tf.keras.utils.experimental.DatasetCreator(
    _input_dataset, input_options=input_options)

eval_dataset = tf.keras.utils.experimental.DatasetCreator(
    _eval_dataset, input_options=input_options)
```

Next, once the data is prepared, you will create a `TPUStrategy`, and define a model, metrics, and an optimizer under the scope of this strategy (`Strategy.scope`).

You should pick a number for `steps_per_execution` in `Model.compile` since it specifies the number of batches to run during each `tf.function` call, and is critical for performance. This argument is similar to `iterations_per_loop` used in `TPUEstimator`.

The features and table configuration that were specified in TensorFlow 1 via the `tf.tpu.experimental.embedding_column` (and `tf.tpu.experimental.shared_embedding_column`) can be specified directly in TensorFlow 2 via a pair of configuration objects:

- `tf.tpu.experimental.embedding.FeatureConfig`
- `tf.tpu.experimental.embedding.TableConfig`

(Refer to the associated API documentation for more details.)

In [ ]:

```
strategy = tf.distribute.TPUStrategy(cluster_resolver)
with strategy.scope():
    optimizer = tf.keras.optimizers.Adagrad(learning_rate=0.05)
    dense_input = tf.keras.Input(shape=(2,), dtype=tf.float32, batch_size=global_batch_size)
    sparse_input = tf.keras.Input(shape=(), dtype=tf.int32, batch_size=global_batch_size)
    embedded_input = tfrs.layers.embedding.TPUEmbedding(
        feature_config=tf.tpu.experimental.embedding.FeatureConfig(
            table=tf.tpu.experimental.embedding.TableConfig(
                vocabulary_size=10,
                dim=5,
                initializer=tf.initializers.TruncatedNormal(mean=0.0, stddev=1)),
            name="sparse_input"),
        optimizer=optimizer)(sparse_input)
    input = tf.keras.layers.concatenate([dense_input, embedded_input])
    result = tf.keras.layers.Dense(1)(input)
    model = tf.keras.Model(inputs={"dense_feature": dense_input, "sparse_feature": sparse_input}, outputs=result)
    model.compile(optimizer, "mse", steps_per_execution=10)
```

With that, you are ready to train the model with the training dataset:

In [ ]:

```
model.fit(input_dataset, epochs=5, steps_per_epoch=10)
```

Finally, evaluate the model using the evaluation dataset:

In [ ]:

```
model.evaluate(eval_dataset, steps=1, return_dict=True)
```

## Next steps

Learn more about setting up TPU-specific embeddings in the API docs:

- `tfrs.layers.embedding.TPUEmbedding`: particularly about feature and table configuration, setting the optimizer, creating a model (using the Keras [functional](https://www.tensorflow.org/guide/keras/functional) API or via [subclassing](#) ([../guide/keras/custom\\_layers\\_and\\_models.ipynb](#))) and serving with `tf.keras.Model`, training/evaluation, and model serving with `tf.saved_model`
- `tf.tpu.experimental.embedding.TableConfig`
- `tf.tpu.experimental.embedding.FeatureConfig`

For more information about `TPUStrategy` in TensorFlow 2, consider the following resources:

- Guide: [Use TPUs](#) ([../guide/tpu.ipynb](#)) (covering training with Keras `Model.fit` / a custom training loop with `tf.distribute.TPUStrategy`, as well as tips on improving the performance with `tf.function`)
- Guide: [Distributed training with TensorFlow](#) ([../guide/distributed\\_training.ipynb](#))
- Guide: [Migrate from TPUEstimator to TPUStrategy](#) (`tpu_estimator.ipynb`).

To learn more about customizing your training, refer to:

- Guide: [Customize what happens in Model.fit](#) ([../guide/keras/customizing\\_what\\_happens\\_in\\_fit.ipynb](#))
- Guide: [Writing a training loop from scratch](#) ([https://www.tensorflow.org/guide/keras/writing\\_a\\_training\\_loop\\_from\\_scratch](https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch))

TPUs—Google's specialized ASICs for machine learning—are available through [Google Colab](#) (<https://colab.research.google.com/>), the [TPU Research Cloud](#) (<https://sites.research.google/trc/>), and [Cloud TPU](#) (<https://cloud.google.com/tpu>).

**Copyright 2021 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## TensorFlow 1.x vs TensorFlow 2 - Behaviors and APIs



[View on TensorFlow.org](#)

([https://www.tensorflow.org/guide/migrate/tf1\\_vs\\_tf2](https://www.tensorflow.org/guide/migrate/tf1_vs_tf2)) ([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/tf1\\_vs\\_tf2.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/tf1_vs_tf2.ipynb)) ([https://github.com/tensorflow/docs/blob/master/site/en/guide/migrate/tf1\\_vs\\_tf2.ipynb](https://github.com/tensorflow/docs/blob/master/site/en/guide/migrate/tf1_vs_tf2.ipynb))



[Run in Google Colab](#)

Under the hood, TensorFlow 2 follows a fundamentally different programming paradigm from TF1.x.

This guide describes the fundamental differences between TF1.x and TF2 in terms of behaviors and the APIs, and how these all relate to your migration journey.

## High-level summary of major changes

Fundamentally, TF1.x and TF2 use a different set of runtime behaviors around execution (eager in TF2), variables, control flow, tensor shapes, and tensor equality comparisons. To be TF2 compatible, your code must be compatible with the full set of TF2 behaviors. During migration, you can enable or disable most of these behaviors individually via the `tf.compat.v1.enable_*` or `tf.compat.v1.disable_*` APIs. The one exception is the removal of collections, which is a side effect of enabling/disabling eager execution.

At a high level, TensorFlow 2:

- Removes [redundant APIs](https://github.com/tensorflow/community/blob/master/rfcs/20180827-api-names.md) (<https://github.com/tensorflow/community/blob/master/rfcs/20180827-api-names.md>).
- Makes APIs more consistent - for example, [Unified RNNs](https://github.com/tensorflow/community/blob/master/rfcs/20180920-unify-rnn-interface.md) (<https://github.com/tensorflow/community/blob/master/rfcs/20180920-unify-rnn-interface.md>) and [Unified Optimizers](https://github.com/tensorflow/community/blob/master/rfcs/20181016-optimizer-unification.md) (<https://github.com/tensorflow/community/blob/master/rfcs/20181016-optimizer-unification.md>).
- Prefers [functions over sessions](https://github.com/tensorflow/community/blob/master/rfcs/20180918-functions-not-sessions-20.md) (<https://github.com/tensorflow/community/blob/master/rfcs/20180918-functions-not-sessions-20.md>) and integrates better with the Python runtime with [Eager execution](https://www.tensorflow.org/guide/eager) (<https://www.tensorflow.org/guide/eager>) enabled by default along with `tf.function` that provides automatic control dependencies for graphs and compilation.
- Deprecates global graph [collections](https://github.com/tensorflow/community/blob/master/rfcs/20180905-deprecate-collections.md) (<https://github.com/tensorflow/community/blob/master/rfcs/20180905-deprecate-collections.md>).
- Alters Variable concurrency semantics by using [ResourceVariables over ReferenceVariables](https://github.com/tensorflow/community/blob/master/rfcs/20180817-variables-20.md) (<https://github.com/tensorflow/community/blob/master/rfcs/20180817-variables-20.md>).
- Supports [function-based](https://github.com/tensorflow/community/blob/master/rfcs/20180507-cond-v2.md) (<https://github.com/tensorflow/community/blob/master/rfcs/20180507-cond-v2.md>) and differentiable [control flow](https://github.com/tensorflow/community/blob/master/rfcs/20180821-differentiable-functional-while.md) (<https://github.com/tensorflow/community/blob/master/rfcs/20180821-differentiable-functional-while.md>) (Control Flow v2).
- Simplifies the TensorShape API to hold `int`'s instead of `tf.compat.v1.Dimension` objects.
- Updates tensor equality mechanics. In TF1.x the `==` operator on tensors and variables checks for object reference equality. In TF2 it checks for value equality. Additionally, tensors/variables are no longer hashable, but you can get hashable object references to them via `var.ref()` if you need to use them in sets or as `dict` keys.

The sections below provide some more context on the differences between TF1.x and TF2. To learn more about the design process behind TF2, read the [RFCs](https://github.com/tensorflow/community/pulls?utf8=%E2%9C%93&q=is%3Apr) (<https://github.com/tensorflow/community/pulls?utf8=%E2%9C%93&q=is%3Apr>) and the [design docs](https://github.com/tensorflow/community/tree/master/rfcs) (<https://github.com/tensorflow/community/tree/master/rfcs>).

## API cleanup

Many APIs are either [gone or moved](https://github.com/tensorflow/community/blob/master/rfcs/20180827-api-names.md) (<https://github.com/tensorflow/community/blob/master/rfcs/20180827-api-names.md>) in TF2. Some of the major changes include removing `tf.app`, `tf.flags`, and `tf.logging` in favor of the now open-source `absl-py` (<https://github.com/abseil/abseil-py>), rehoming projects that lived in `tf.contrib`, and cleaning up the main `tf.*` namespace by moving lesser used functions into subpackages like `tf.math`. Some APIs have been replaced with their TF2 equivalents - `tf.summary`, `tf.keras.metrics`, and `tf.keras.optimizers`.

## `tf.compat.v1`: Legacy and Compatibility API Endpoints

Symbols under the `tf.compat` and `tf.compat.v1` namespaces are not considered TF2 APIs. These namespaces expose a mix of compatibility symbols, as well as legacy API endpoints from TF 1.x. These are intended to aid migration from TF1.x to TF2. However, as none of these `compat.v1` APIs are idiomatic TF2 APIs, do not use them for writing brand-new TF2 code.

Individual `tf.compat.v1` symbols may be TF2 compatible because they continue to work even with TF2 behaviors enabled (such as `tf.compat.v1.losses.mean_squared_error`), while others are incompatible with TF2 (such as `tf.compat.v1.metrics.accuracy`). Many `compat.v1` symbols (though not all) contain dedicated migration information in their documentation that explains their degree of compatibility with TF2 behaviors, as well as how to migrate them to TF2 APIs.

The [TF2 upgrade script](https://www.tensorflow.org/guide/migrate/upgrade) (<https://www.tensorflow.org/guide/migrate/upgrade>) can map many `compat.v1` API symbols to equivalent TF2 APIs in the case where they are aliases or have the same arguments but with a different ordering. You can also use the upgrade script to automatically rename TF1.x APIs.

## False friend APIs

There are a set of "false-friend" symbols found in the TF2 `tf` namespace (not under `compat.v1`) that actually ignore TF2 behaviors under-the-hood, and/or are not fully compatible with the full set of TF2 behaviors. As such, these APIs are likely to misbehave with TF2 code, potentially in silent ways.

- `tf.estimator.*`: Estimators create and use graphs and sessions under the hood. As such, these should not be considered TF2-compatible. If your code is running estimators, it is not using TF2 behaviors.
- `keras.Model.model_to_estimator(...)`: This creates an Estimator under the hood, which as mentioned above is not TF2-compatible.
- `tf.Graph().as_default()`: This enters TF1.x graph behaviors and does not follow standard TF2-compatible `tf.function` behaviors. Code that enters graphs like this will generally run them via Sessions, and should not be considered TF2-compatible.
- `tf.feature_column.*`: The feature column APIs generally rely on TF1-style `tf.compat.v1.get_variable` variable creation and assume that the created variables will be accessed via global collections. As TF2 does not support collections, APIs may not work correctly when running them with TF2 behaviors enabled.

## Other API changes

- TF2 features significant improvements to the device placement algorithms which renders the usage of `tf.colocate_with` unnecessary. If removing it causes a performance degrade [please file a bug](https://github.com/tensorflow/tensorflow/issues) (<https://github.com/tensorflow/tensorflow/issues>).
- Replace all usage of `tf.v1.ConfigProto` with equivalent functions from `tf.config`.

## Eager execution

TF1.x required you to manually stitch together an [abstract syntax tree](https://en.wikipedia.org/wiki/Abstract_syntax_tree) ([https://en.wikipedia.org/wiki/Abstract\\_syntax\\_tree](https://en.wikipedia.org/wiki/Abstract_syntax_tree)) (the graph) by making `tf.*` API calls and then manually compile the abstract syntax tree by passing a set of output tensors and input tensors to a `session.run` call. TF2 executes eagerly (like Python normally does) and makes graphs and sessions feel like implementation details.

One notable byproduct of eager execution is that `tf.control_dependencies` is no longer required, as all lines of code execute in order (within a `tf.function`, code with side effects executes in the order written).

## No more globals

TF1.x relied heavily on implicit global namespaces and collections. When you called `tf.Variable`, it would be put into a collection in the default graph, and it would remain there, even if you lost track of the Python variable pointing to it. You could then recover that `tf.Variable`, but only if you knew the name that it had been created with. This was difficult to do if you were not in control of the variable's creation. As a result, all sorts of mechanisms proliferated to attempt to help you find your variables again, and for frameworks to find user-created variables. Some of these include: variable scopes, global collections, helper methods like `tf.get_global_step` and `tf.global_variables_initializer`, optimizers implicitly computing gradients over all trainable variables, and so on. TF2 eliminates all of these mechanisms ([Variables 2.0 RFC](https://github.com/tensorflow/community/pull/11) (<https://github.com/tensorflow/community/pull/11>)) in favor of the default mechanism - you keep track of your variables. If you lose track of a `tf.Variable`, it gets garbage collected.

The requirement to track variables creates some extra work, but with tools like the [modeling shims](#) ([./model\\_mapping.ipynb](#)) and behaviors like [implicit object-oriented variable collections in](#) `tf.Module`s and `tf.keras.layers.Layer`s ([https://www.tensorflow.org/guide/intro\\_to\\_modules](https://www.tensorflow.org/guide/intro_to_modules)), the burden is minimized.

## Functions, not sessions

A `session.run` call is almost like a function call: you specify the inputs and the function to be called, and you get back a set of outputs. In TF2, you can decorate a Python function using `tf.function` to mark it for JIT compilation so that TensorFlow runs it as a single graph ([Functions 2.0 RFC](#) (<https://github.com/tensorflow/community/pull/20>)). This mechanism allows TF2 to gain all of the benefits of graph mode:

- Performance: The function can be optimized (node pruning, kernel fusion, etc.)
- Portability: The function can be exported/reimported ([SavedModel 2.0 RFC](#) (<https://github.com/tensorflow/community/pull/34>)), allowing you to reuse and share modular TensorFlow functions.

```
# TF1.x
outputs = session.run(f(placeholder), feed_dict={placeholder: input})
# TF2
outputs = f(input)
```

With the power to freely intersperse Python and TensorFlow code, you can take advantage of Python's expressiveness. However, portable TensorFlow executes in contexts without a Python interpreter, such as mobile, C++, and JavaScript. To help avoid rewriting your code when adding `tf.function`, use [AutoGraph](#) (<https://tensorflow.org/guide/function>) to convert a subset of Python constructs into their TensorFlow equivalents:

- `for/while -> tf.while_loop` (`break` and `continue` are supported)
- `if -> tf.cond`
- `for _ in dataset -> dataset.reduce`

AutoGraph supports arbitrary nestings of control flow, which makes it possible to performantly and concisely implement many complex ML programs such as sequence models, reinforcement learning, custom training loops, and more.

## Adapting to TF 2.x Behavior Changes

Your migration to TF2 is only complete once you have migrated to the full set of TF2 behaviors. The full set of behaviors can be enabled or disabled via `tf.compat.v1.enable_v2_behaviors` and `tf.compat.v1.disable_v2_behaviors`. The sections below discuss each major behavior change in detail.

### Using `tf.functions`

The largest changes to your programs during migration are likely to come from the fundamental programming model paradigm shift from graphs and sessions to eager execution and `tf.function`. Refer to the [TF2 migration guides](#) (<https://tensorflow.org/guide/migrate>) to learn more about moving from APIs that are incompatible with eager execution and `tf.function` to APIs that are compatible with them.

Note: During migration you may choose to directly enable and disable eager execution with `tf.compat.v1.enable_eager_execution` and `tf.compat.v1.disable_eager_execution`, but this may only be done once during the lifetime of your program.

Below are some common program patterns not tied to any one API that may cause problems when switching from `tf.Graph`s and `tf.compat.v1.Session`s to eager execution with `tf.function`s.

#### Pattern 1: Python object manipulation and variable creation intended to be done only once get run multiple times

In TF1.x programs that rely on graphs and sessions, the expectation is usually that all Python logic in your program will only run once. However, with eager execution and `tf.function` it is fair to expect that your Python logic will be run at least once, but possibly more times (either multiple times eagerly, or multiple times across different `tf.function` traces). Sometimes, `tf.function` will even trace twice on the same input, causing unexpected behaviors (see Example 1 and 2). Refer to the [tf.function guide](#) (<https://www.tensorflow.org/guide/function>) for more details.

Note: This pattern usually causes your code to silently misbehave when executing eagerly without `tf.function`s, but generally raises an `InaccessibleTensorError` or a `ValueError` when attempting to wrap the problematic code inside of a `tf.function`. To discover and debug this issue, it is recommended you wrap your code with `tf.function` early on, and use [pdb](#) (<https://docs.python.org/3/library/pdb.html>) or interactive debugging to identify the source of the `InaccessibleTensorError`.

#### Example 1: Variable creation

Consider the example below, where the function creates a variable when called:

```
def f():
    v = tf.Variable(1.0)
    return v

with tf.Graph().as_default():
    with tf.compat.v1.Session() as sess:
        res = f()
        sess.run(tf.compat.v1.global_variables_initializer())
        sess.run(res)
```

However, naively wrapping the above function that contains variable creation with `tf.function` is not allowed. `tf.function` only supports [singleton variable creations on the first call](#) ([https://www.tensorflow.org/guide/function#creating\\_tfvariables](https://www.tensorflow.org/guide/function#creating_tfvariables)). To enforce this, when `tf.function` detects variable creation in the first call, it will attempt to trace again and raise an error if there is variable creation in the second trace.

```
@tf.function
def f():
    print("trace") # This will print twice because the python body is run twice
    v = tf.Variable(1.0)
    return v

try:
    f()
except ValueError as e:
    print(e)
```

A workaround is caching and reusing the variable after it is created in the first call.

```
class Model(tf.Module):
    def __init__(self):
        self.v = None

    @tf.function
    def __call__(self):
        print("trace") # This will print twice because the python body is run twice
        if self.v is None:
            self.v = tf.Variable(0)
        return self.v

m = Model()
m()
```

### Example 2: Out-of-scope Tensors due to `tf.function` retracing

As demonstrated in Example 1, `tf.function` will retrace when it detects Variable creation in the first call. This can cause extra confusion, because the two tracings will create two graphs. When the second graph from retracing attempts to access a Tensor from the graph generated during the first tracing, Tensorflow will raise an error complaining that the Tensor is out of scope. To demonstrate the scenario, the code below creates a dataset on the first `tf.function` call. This would run as expected.

```
class Model(tf.Module):
    def __init__(self):
        self.dataset = None

    @tf.function
    def __call__(self):
        print("trace") # This will print once: only traced once
        if self.dataset is None:
            self.dataset = tf.data.Dataset.from_tensors([1, 2, 3])
        it = iter(self.dataset)
        return next(it)

m = Model()
m()
```

However, if we also attempt to create a variable on the first `tf.function` call, the code will raise an error complaining that the dataset is out of scope. This is because the dataset is in the first graph, while the second graph is also attempting to access it.

```

class Model(tf.Module):
    def __init__(self):
        self.v = None
        self.dataset = None

    @tf.function
    def __call__(self):
        print("trace") # This will print twice because the python body is run twice
        if self.v is None:
            self.v = tf.Variable(0)
        if self.dataset is None:
            self.dataset = tf.data.Dataset.from_tensors([1, 2, 3])
        it = iter(self.dataset)
        return [self.v, next(it)]

m = Model()
try:
    m()
except TypeError as e:
    print(e) # <tf.Tensor ...> is out of scope and cannot be used here.

```

The most straightforward solution is ensuring that the variable creation and dataset creation are both outside of the `tf.function` call. For example:

```

class Model(tf.Module):
    def __init__(self):
        self.v = None
        self.dataset = None

    def initialize(self):
        if self.dataset is None:
            self.dataset = tf.data.Dataset.from_tensors([1, 2, 3])
        if self.v is None:
            self.v = tf.Variable(0)

    @tf.function
    def __call__(self):
        it = iter(self.dataset)
        return [self.v, next(it)]

m = Model()
m.initialize()
m()

```

However, sometimes it's not avoidable to create variables in `tf.function` (such as slot variables in some [TF keras optimizers](#) ([https://www.tensorflow.org/api\\_docs/python/tf/keras/optimizers/Optimizer#slots](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Optimizer#slots))). Still, we can simply move the dataset creation outside of the `tf.function` call. The reason that we can rely on this is because `tf.function` will receive the dataset as an implicit input and both graphs can access it properly.

```

class Model(tf.Module):
    def __init__(self):
        self.v = None
        self.dataset = None

    def initialize(self):
        if self.dataset is None:
            self.dataset = tf.data.Dataset.from_tensors([1, 2, 3])

    @tf.function
    def __call__(self):
        if self.v is None:
            self.v = tf.Variable(0)
        it = iter(self.dataset)
        return [self.v, next(it)]

m = Model()
m.initialize()
m()

```

**Example 3: Unexpected Tensorflow object re-creations due to dict usage**

`tf.function` has very poor support for python side effects such as appending to a list, or checking/adding to a dictionary. More details are in "[Better performance with `tf.function`](#)" ([https://www.tensorflow.org/guide/function#executing\\_python\\_side\\_effects](https://www.tensorflow.org/guide/function#executing_python_side_effects)). In the example below, the code uses dictionaries to cache datasets and iterators. For the same key, each call to the model will return the same iterator of the dataset.

```
class Model(tf.Module):
    def __init__(self):
        self.datasets = {}
        self.iterators = {}

    def __call__(self, key):
        if key not in self.datasets:
            self.datasets[key] = tf.compat.v1.data.Dataset.from_tensor_slices([1, 2, 3])
            self.iterators[key] = self.datasets[key].make_initializable_iterator()
        return self.iterators[key]

with tf.Graph().as_default():
    with tf.compat.v1.Session() as sess:
        m = Model()
        it = m('a')
        sess.run(it.initializer)
        for _ in range(3):
            print(sess.run(it.get_next())) # prints 1, 2, 3
```

However, the pattern above will not work as expected in `tf.function`. During tracing, `tf.function` will ignore the python side effect of addition to the dictionaries. Instead, it only remembers the creation of a new dataset and iterator. As a result, each call to the model will always return a new iterator. This issue is hard to notice unless the numerical results or performance are significant enough. Hence, we recommend users to think about the code carefully before wrapping `tf.function` naively onto the python code.

```
class Model(tf.Module):
    def __init__(self):
        self.datasets = {}
        self.iterators = {}

    @tf.function
    def __call__(self, key):
        if key not in self.datasets:
            self.datasets[key] = tf.data.Dataset.from_tensor_slices([1, 2, 3])
            self.iterators[key] = iter(self.datasets[key])
        return self.iterators[key]

m = Model()
for _ in range(3):
    print(next(m('a'))) # prints 1, 1, 1
```

We can use `tf.init_scope` ([https://www.tensorflow.org/api\\_docs/python/tf/init\\_scope](https://www.tensorflow.org/api_docs/python/tf/init_scope)) to lift the dataset and iterator creation outside of the graph, to achieve the expected behavior:

```
class Model(tf.Module):
    def __init__(self):
        self.datasets = {}
        self.iterators = {}

    @tf.function
    def __call__(self, key):
        if key not in self.datasets:
            # Lifts ops out of function-building graphs
            with tf.init_scope():
                self.datasets[key] = tf.data.Dataset.from_tensor_slices([1, 2, 3])
                self.iterators[key] = iter(self.datasets[key])
            return self.iterators[key]

m = Model()
for _ in range(3):
    print(next(m('a'))) # prints 1, 2, 3
```

The general rule of thumb is to avoid relying on Python side effects in your logic and only use them to debug your traces.

#### Example 4: Manipulating a global Python list

The following TF1.x code uses a global list of losses that it uses to only maintain the list of losses generated by the current training step. Note that the Python logic that appends losses to the list will only be called once regardless of how many training steps the session is run for.

```

all_losses = []

class Model():
    def __call__(...):
        ...
        all_losses.append(regularization_loss)
        all_losses.append(label_loss_a)
        all_losses.append(label_loss_b)
        ...

g = tf.Graph()
with g.as_default():

    ...
    # initialize all objects
model = Model()
optimizer = ...

...
# train step
model(...)
total_loss = tf.reduce_sum(all_losses)
optimizer.minimize(total_loss)

...
...

sess = tf.compat.v1.Session(graph=g)
sess.run(...)
```

However, if this Python logic is naively mapped to TF2 with eager execution, the global list of losses will have new values appended to it in each training step. This means the training step code which previously expected the list to only contain losses from the current training step now actually sees the list of losses from all training steps run so far. This is an unintended behavior change, and the list will either need to be cleared at the start of each step or made local to the training step.

```

all_losses = []

class Model():
    def __call__(...):
        ...
        all_losses.append(regularization_loss)
        all_losses.append(label_loss_a)
        all_losses.append(label_loss_b)
        ...

    # initialize all objects
model = Model()
optimizer = ...

def train_step(...):
    ...
    model(...)
    total_loss = tf.reduce_sum(all_losses) # global list is never cleared,
    # Accidentally accumulates sum loss across all training steps
    optimizer.minimize(total_loss)
    ...
```

## Pattern 2: A symbolic tensor meant to be recomputed every step in TF1.x is accidentally cached with the initial value when switching to eager.

This pattern usually causes your code to silently misbehave when executing eagerly outside of `tf.functions`, but raises an `InaccessibleTensorError` if the initial value caching occurs inside of a `tf.function`. However, be aware that in order to avoid [Pattern 1](#) above you will often inadvertently structure your code in such a way that this initial value caching will happen *outside* of any `tf.function` that would be able to raise an error. So, take extra care if you know your program may be susceptible to this pattern.

The general solution to this pattern is to restructure the code or use Python callables if necessary to make sure the value is recomputed each time instead of being accidentally cached.

### Example 1: Learning rate/hyperparameter/etc. schedules that depend on global step

In the following code snippet, the expectation is that every time the session is run the most recent `global_step` value will be read and a new learning rate will be computed.

```

g = tf.Graph()
with g.as_default():
    ...
    global_step = tf.Variable(0)
    learning_rate = 1.0 / global_step
    opt = tf.compat.v1.train.GradientDescentOptimizer(learning_rate)
    ...
    global_step.assign_add(1)
    ...
sess = tf.compat.v1.Session(graph=g)
sess.run(...)

```

However, when trying to switch to eager, be wary of ending up with the learning rate only being computed once then reused, rather than following the intended schedule:

```

global_step = tf.Variable(0)
learning_rate = 1.0 / global_step # Wrong! Only computed once!
opt = tf.keras.optimizers.SGD(learning_rate)

def train_step(...):
    ...
    opt.apply_gradients(...)
    global_step.assign_add(1)
    ...

```

Because this specific example is a common pattern and optimizers should only be initialized once rather than at each training step, TF2 optimizers support `tf.keras.optimizers.schedules.LearningRateSchedule` schedules or Python callables as arguments for the learning rate and other hyperparameters.

#### **Example 2: Symbolic random number initializations assigned as object attributes then reused via pointer are accidentally cached when switching to eager**

Consider the following `NoiseAdder` module:

```

class NoiseAdder(tf.Module):
    def __init__(shape, mean):
        self.noise_distribution = tf.random.normal(shape=shape, mean=mean)
        self.trainable_scale = tf.Variable(1.0, trainable=True)

    def add_noise(input):
        return (self.noise_distribution + input) * self.trainable_scale

```

Using it as follows in TF1.x will compute a new random noise tensor every time the session is run:

```

g = tf.Graph()
with g.as_default():
    ...
    # initialize all variable-containing objects
    noise_adder = NoiseAdder(shape, mean)
    ...
    # computation pass
    x_with_noise = noise_adder.add_noise(x)
    ...
    ...
sess = tf.compat.v1.Session(graph=g)
sess.run(...)

```

However, in TF2 initializing the `noise_adder` at the beginning will cause the `noise_distribution` to be only computed once and get frozen for all training steps:

```

...
# initialize all variable-containing objects
noise_adder = NoiseAdder(shape, mean) # Freezes `self.noise_distribution`!
...
# computation pass
x_with_noise = noise_adder.add_noise(x)
...

```

To fix this, refactor `NoiseAdder` to call `tf.random.normal` every time a new random tensor is needed, instead of referring to the same tensor object each time.

```

class NoiseAdder(tf.Module):
    def __init__(shape, mean):
        self.noise_distribution = lambda: tf.random.normal(shape=shape, mean=mean)
        self.trainable_scale = tf.Variable(1.0, trainable=True)

    def add_noise(input):
        return (self.noise_distribution() + input) * self.trainable_scale

```

### Pattern 3: TF1.x code directly relies on and looks up tensors by name

It is common for TF1.x code tests to rely on checking what tensors or operations are present in a graph. In some rare cases, modeling code will also rely on these lookups by name.

Tensor names are not generated when executing eagerly outside of `tf.function` at all, so all usages of `tf.Tensor.name` must happen inside of a `tf.function`. Keep in mind the actual generated names are very likely to differ between TF1.x and TF2 even within the same `tf.function`, and API guarantees do not ensure stability of the generated names across TF versions.

Note: Variable names are still generated even outside of `tf.function`s, but their names also are not guaranteed to match between TF1.x and TF2 except when following the relevant sections of the [model mapping guide \(./model\\_mapping.ipynb\)](#).

### Pattern 4: TF1.x session selectively runs only part of the generated graph

In TF1.x, you can construct a graph and then choose to only selectively run only a subset of it with a session by choosing a set of inputs and outputs that do not require running every op in the graph.

For example, you may have both a generator and a discriminator inside of a single graph, and use separate `tf.compat.v1.Session.run` calls to alternate between only training the discriminator or only training the generator.

In TF2, due to automatic control dependencies in `tf.function` and eager execution, there is no selective pruning of `tf.function` traces. A full graph containing all variable updates would get run even if, for example, only the output of the discriminator or the generator is output from the `tf.function`.

So, you would need to either use multiple `tf.function`s containing different parts of the program, or a conditional argument to the `tf.function` that you branch on so as to execute only the things you actually want to have run.

## Collections Removal

When eager execution is enabled, graph collection-related `compat.v1` APIs (including those that read or write to collections under the hood such as `tf.compat.v1.trainable_variables`) are no longer available. Some may raise `ValueError`s, while others may silently return empty lists.

The most standard usage of collections in TF1.x is to maintain initializers, the global step, weights, regularization losses, model output losses, and variable updates that need to be run such as from `BatchNormalization` layers.

To handle each of these standard usages:

1. Initializers - Ignore. Manual variable initialization is not required with eager execution enabled.
2. Global step - See the documentation of `tf.compat.v1.train.get_or_create_global_step` for migration instructions.
3. Weights - Map your models to `tf.Module`s/`tf.keras.layers.Layer`s/`tf.keras.Model`s by following the guidance in the [model mapping guide \(./model\\_mapping.ipynb\)](#) and then use their respective weight-tracking mechanisms such as `tf.module.trainable_variables`.
4. Regularization losses - Map your models to `tf.Module`s/`tf.keras.layers.Layer`s/`tf.keras.Model`s by following the guidance in the [model mapping guide \(./model\\_mapping.ipynb\)](#) and then use `tf.keras.losses`. Alternatively, you can also manually track your regularization losses.
5. Model output losses - Use `tf.keras.Model` loss management mechanisms or separately track your losses without using collections.
6. Weight updates - Ignore this collection. Eager execution and `tf.function` (with autograph and auto-control-dependencies) means all variable updates will get run automatically. So, you will not have to explicitly run all weight updates at the end, but note that it means the weight updates may happen at a different time than they did in your TF1.x code, depending on how you were using control dependencies.
7. Summaries - Refer to the [migrating summary API guide \(https://www.tensorflow.org/tensorboard/migrate\)](#).

More complex collections usage (such as using custom collections) may require you to refactor your code to either maintain your own global stores, or to make it not rely on global stores at all.

## ResourceVariables instead of ReferenceVariables

`ResourceVariables` have stronger read-write consistency guarantees than `ReferenceVariables`. This leads to more predictable, easier-to-reason about semantics about whether or not you will observe the result of a previous write when using your variables. This change is extremely unlikely to cause existing code to raise errors or to break silently.

However, it is **possible though unlikely** that these stronger consistency guarantees may increase the memory usage of your specific program. Please file an [issue](https://github.com/tensorflow/tensorflow/issues) (<https://github.com/tensorflow/tensorflow/issues>) if you find this to be the case. Additionally, if you have unit tests relying on exact string comparisons against the operator names in a graph corresponding to variable reads, be aware that enabling resource variables may slightly change the name of these operators.

To isolate the impact of this behavior change on your code, if eager execution is disabled you can use

`tf.compat.v1.disable_resource_variables()` and `tf.compat.v1.enable_resource_variables()` to globally disable or enable this behavior change. `ResourceVariables` will always be used if eager execution is enabled.

## Control flow v2

In TF1.x, control flow ops such as `tf.cond` and `tf.while_loop` inline low-level ops such as `Switch`, `Merge` etc. TF2 provides improved functional control flow ops that are implemented with separate `tf.function` traces for every branch and support higher-order differentiation.

To isolate the impact of this behavior change on your code, if eager execution is disabled you can use

`tf.compat.v1.disable_control_flow_v2()` and `tf.compat.v1.enable_control_flow_v2()` to globally disable or enable this behavior change. However, you can only disable control flow v2 if eager execution is also disabled. If it is enabled, control flow v2 will always be used.

This behavior change can dramatically change the structure of generated TF programs that use control flow, as they will contain several nested function traces rather than one flat graph. So, any code that is highly dependent on the exact semantics of produced traces may require some modification. This includes:

- Code relying on operator and tensor names
- Code referring to tensors created within a TensorFlow control flow branch from outside of that branch. This is likely to produce an `InaccessibleTensorError`

This behavior change is intended to be performance neutral to positive, but if you run into an issue where control flow v2 performs worse for you than TF1.x control flow then please file an [issue](https://github.com/tensorflow/tensorflow/issues) (<https://github.com/tensorflow/tensorflow/issues>) with reproduction steps.

## TensorShape API behavior changes

The `TensorShape` class was simplified to hold `int`'s, instead of `tf.compat.v1.Dimension` objects. So there is no need to call `.value` to get an `int`.

Individual `tf.compat.v1.Dimension` objects are still accessible from `tf.TensorShape.dims`.

To isolate the impact of this behavior change on your code, you can use `tf.compat.v1.disable_v2_tensorshape()` and `tf.compat.v1.enable_v2_tensorshape()` to globally disable or enable this behavior change.

The following demonstrate the differences between TF1.x and TF2.

In [ ]:

```
import tensorflow as tf
```

In [ ]:

```
# Create a shape and choose an index
i = 0
shape = tf.TensorShape([16, None, 256])
shape
```

If you had this in TF1.x:

```
value = shape[i].value
```

Then do this in TF2:

In [ ]:

```
value = shape[i]
value
```

If you had this in TF1.x:

```
for dim in shape:  
    value = dim.value  
    print(value)
```

Then, do this in TF2:

In [ ]:

```
for value in shape:  
    print(value)
```

If you had this in TF1.x (or used any other dimension method):

```
dim = shape[i]  
dim.assert_is_compatible_with(other_dim)
```

Then do this in TF2:

In [ ]:

```
other_dim = 16  
Dimension = tf.compat.v1.Dimension  
  
if shape.rank is None:  
    dim = Dimension(None)  
else:  
    dim = shape.dims[i]  
dim.is_compatible_with(other_dim) # or any other dimension method
```

In [ ]:

```
shape = tf.TensorShape(None)  
  
if shape:  
    dim = shape.dims[i]  
    dim.is_compatible_with(other_dim) # or any other dimension method
```

The boolean value of a `tf.TensorShape` is `True` if the rank is known, `False` otherwise.

In [ ]:

```
print(bool(tf.TensorShape([])))      # Scalar  
print(bool(tf.TensorShape([0])))     # 0-length vector  
print(bool(tf.TensorShape([1])))     # 1-length vector  
print(bool(tf.TensorShape([None])))   # Unknown-length vector  
print(bool(tf.TensorShape([1, 10, 100]))) # 3D tensor  
print(bool(tf.TensorShape([None, None, None]))) # 3D tensor with no known dimensions  
print()  
print(bool(tf.TensorShape(None)))    # A tensor with unknown rank.
```

## Potential errors due to `TensorShape` changes

The `TensorShape` behavior changes are unlikely to silently break your code. However, you may see shape-related code begin to raise `AttributeError`s as `int`s and `None`s do not have the same attributes that `tf.compat.v1.Dimension`s do. Below are some examples of these `AttributeError`s:

In [ ]:

```
try:  
    # Create a shape and choose an index  
    shape = tf.TensorShape([16, None, 256])  
    value = shape[0].value  
except AttributeError as e:  
    # 'int' object has no attribute 'value'  
    print(e)
```

In [ ]:

```
try:  
    # Create a shape and choose an index  
    shape = tf.TensorShape([16, None, 256])  
    dim = shape[1]  
    other_dim = shape[2]  
    dim.assert_is_compatible_with(other_dim)  
except AttributeError as e:  
    # 'NoneType' object has no attribute 'assert_is_compatible_with'  
    print(e)
```

## Tensor Equality by Value

The binary `==` and `!=` operators on variables and tensors were changed to compare by value in TF2 rather than comparing by object reference like in TF1.x. Additionally, tensors and variables are no longer directly hashable or usable in sets or dict keys, because it may not be possible to hash them by value. Instead, they expose a `.ref()` method that you can use to get a hashable reference to the tensor or variable.

To isolate the impact of this behavior change, you can use `tf.compat.v1.disable_tensor_equality()` and `tf.compat.v1.enable_tensor_equality()` to globally disable or enable this behavior change.

For example, in TF1.x, two variables with the same value will return false when you use the `==` operator:

In [ ]:

```
tf.compat.v1.disable_tensor_equality()  
x = tf.Variable(0.0)  
y = tf.Variable(0.0)  
  
x == y
```

While in TF2 with tensor equality checks enabled, `x == y` will return `True`.

In [ ]:

```
tf.compat.v1.enable_tensor_equality()  
x = tf.Variable(0.0)  
y = tf.Variable(0.0)  
  
x == y
```

So, in TF2, if you need to compare by object reference make sure to use `is` and `is not`

In [ ]:

```
tf.compat.v1.enable_tensor_equality()  
x = tf.Variable(0.0)  
y = tf.Variable(0.0)  
  
x is y
```

## Hashing tensors and variables

With TF1.x behaviors you used to be able to directly add variables and tensors to data structures that require hashing, such as `set` and `dict` keys.

In [ ]:

```
tf.compat.v1.disable_tensor_equality()  
x = tf.Variable(0.0)  
set([x, tf.constant(2.0)])
```

However, in TF2 with tensor equality enabled, tensors and variables are made unhashable due to the `==` and `!=` operator semantics changing to value equality checks.

In [ ]:

```
tf.compat.v1.enable_tensor_equality()  
x = tf.Variable(0.0)  
  
try:  
    set([x, tf.constant(2.0)])  
except TypeError as e:  
    # TypeError: Variable is unhashable. Instead, use tensor.ref() as the key.  
    print(e)
```

So, in TF2 if you need to use tensor or variable objects as keys or `set` contents, you can use `tensor.ref()` to get a hashable reference that can be used as a key:

In [ ]:

```
tf.compat.v1.enable_tensor_equality()
x = tf.Variable(0.0)

tensor_set = set([x.ref(), tf.constant(2.0).ref()])
assert x.ref() in tensor_set

tensor_set
```

If needed, you can also get the tensor or variable from the reference by using `reference.deref()`:

In [ ]:

```
referenced_var = x.ref().deref()
assert referenced_var is x
referenced_var
```

## Resources and further reading

- Visit the [Migrate to TF2 \(`https://tensorflow.org/guide/migrate`\)](https://tensorflow.org/guide/migrate) section to read more about migrating to TF2 from TF1.x.
- Read the [model mapping guide \(`./model\_mapping.ipynb`\)](#) to learn more mapping your TF1.x models to work in TF2 directly.

*Copyright 2021 The TensorFlow Authors.*

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Migrate metrics and optimizers



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/metrics_optimizers)

(`https://www.tensorflow.org/guide/migrate/metrics_optimizers`)



[Run in Google Colab](#)

(`https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/metrics_optimizers.ipynb`)

In TF1, `tf.metrics` is the API namespace for all the metric functions. Each of the metrics is a function that takes `label` and `prediction` as input parameters and returns the corresponding metrics tensor as result. In TF2, `tf.keras.metrics` contains all the metric functions and objects. The `Metric` object can be used with `tf.keras.Model` and `tf.keras.layers.Layer` to calculate metric values.

## Setup

Let's start with a couple of necessary TensorFlow imports,

In [ ]:

```
import tensorflow as tf
import tensorflow.compat.v1 as tf1
```

and prepare some simple data for demonstration:

In [ ]:

```
features = [[1., 1.5], [2., 2.5], [3., 3.5]]
labels = [0, 0, 1]
eval_features = [[4., 4.5], [5., 5.5], [6., 6.5]]
eval_labels = [0, 1, 1]
```

## TF1: tf.compat.v1.metrics with Estimator

In TF1, the metrics can be added to `EstimatorSpec` as the `eval_metric_ops`, and the op is generated via all the metrics functions defined in `tf.metrics`. You can follow the example to see how to use `tf.metrics.accuracy`.

In [ ]:

```
def _input_fn():
    return tf1.data.Dataset.from_tensor_slices((features, labels)).batch(1)

def _eval_input_fn():
    return tf1.data.Dataset.from_tensor_slices(
        (eval_features, eval_labels)).batch(1)

def _model_fn(features, labels, mode):
    logits = tf1.layers.Dense(2)(features)
    predictions = tf.argmax(input=logits, axis=1)
    loss = tf1.nn.sparse_softmax_cross_entropy_with_logits(labels=labels, logits=logits)
    optimizer = tf1.train.AdagradOptimizer(0.05)
    train_op = optimizer.minimize(loss, global_step=tf1.train.get_global_step())
    accuracy = tf1.metrics.accuracy(labels=labels, predictions=predictions)
    return tf1.estimator.EstimatorSpec(mode,
  predictions=predictions,
  loss=loss,
  train_op=train_op,
  eval_metric_ops={'accuracy': accuracy})

estimator = tf1.estimator.Estimator(model_fn=_model_fn)
estimator.train(_input_fn)
```

In [ ]:

```
estimator.evaluate(_eval_input_fn)
```

Also, metrics could be added to estimator directly via `tf.estimator.add_metrics()`.

In [ ]:

```
def mean_squared_error(labels, predictions):
    labels = tf.cast(labels, predictions.dtype)
    return {"mean_squared_error":
            tf1.metrics.mean_squared_error(labels=labels, predictions=predictions)}

estimator = tf1.estimator.add_metrics(estimator, mean_squared_error)
estimator.evaluate(_eval_input_fn)
```

## TF2: Keras Metrics API with tf.keras.Model

In TF2, `tf.keras.metrics` contains all the metrics classes and functions. They are designed in a OOP style and integrate closely with other `tf.keras` API. All the metrics can be found in `tf.keras.metrics` namespace, and there is usually a direct mapping between `tf.compat.v1.metrics` with `tf.keras.metrics`.

In the following example, the metrics are added in `model.compile()` method. Users only need to create the metric instance, without specifying the label and prediction tensor. The Keras model will route the model output and label to the metrics object.

In [ ]:

```
dataset = tf.data.Dataset.from_tensor_slices((features, labels)).batch(1)
eval_dataset = tf.data.Dataset.from_tensor_slices(
    (eval_features, eval_labels)).batch(1)

inputs = tf.keras.Input((2,))
logits = tf.keras.layers.Dense(2)(inputs)
predictions = tf.argmax(input=logits, axis=1)
model = tf.keras.models.Model(inputs, predictions)
optimizer = tf.keras.optimizers.Adagrad(learning_rate=0.05)

model.compile(optimizer, loss='mse', metrics=[tf.keras.metrics.Accuracy()])
```

In [ ]:

```
model.evaluate(eval_dataset, return_dict=True)
```

With eager execution enabled, `tf.keras.metrics.Metric` instances can be directly used to evaluate numpy data or eager tensors. `tf.keras.metrics.Metric` objects are stateful containers. The metric value can be updated via `metric.update_state(y_true, y_pred)`, and the result can be retrieved by `metrics.result()`.

In [ ]:

```
accuracy = tf.keras.metrics.Accuracy()  
  
accuracy.update_state(y_true=[0, 0, 1, 1], y_pred=[0, 0, 0, 1])  
accuracy.result().numpy()
```

In [ ]:

```
accuracy.update_state(y_true=[0, 0, 1, 1], y_pred=[0, 0, 0, 0])  
accuracy.update_state(y_true=[0, 0, 1, 1], y_pred=[1, 1, 0, 0])  
accuracy.result().numpy()
```

For more details about `tf.keras.metrics.Metric`, please take a look for the API documentation at `tf.keras.metrics.Metric`, as well as the [migration guide \(https://www.tensorflow.org/guide/effective\\_tf2#new-style\\_metrics\\_and\\_losses\)](https://www.tensorflow.org/guide/effective_tf2#new-style_metrics_and_losses).

## Migrate TF1.x optimizers to Keras optimizers

The optimizers in `tf.compat.v1.train`, such as the [Adam optimizer \(https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/train/AdamOptimizer\)](https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/AdamOptimizer) and the [gradient descent optimizer \(https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/train/GradientDescentOptimizer\)](https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/GradientDescentOptimizer), have equivalents in `tf.keras.optimizers`.

The table below summarizes how you can convert these legacy optimizers to their Keras equivalents. You can directly replace the TF1.x version with the TF2 version unless additional steps (such as [updating the default learning rate \(../guide/effective\\_tf2.ipynb#optimizer\\_defaults\)](#)) are required.

Note that converting your optimizers [may make old checkpoints incompatible \(./migrating\\_checkpoints.ipynb\)](#).

TF1.x	TF2	Additional steps
<code>'tf.v1.train.GradientDescentOptimizer'</code>	<code>'tf.keras.optimizers.SGD'</code>	None
<code>'tf.v1.train.MomentumOptimizer'</code>	<code>'tf.keras.optimizers.SGD'</code>	Include the <code>'momentum'</code> argument
<code>'tf.v1.train.AdamOptimizer'</code>	<code>'tf.keras.optimizers.Adam'</code>	Rename <code>'beta1'</code> and <code>'beta2'</code> arguments to <code>'beta_1'</code> and <code>'beta_2'</code>
<code>'tf.v1.train.RMSPropOptimizer'</code>	<code>'tf.keras.optimizers.RMSprop'</code>	Rename the <code>'decay'</code> argument to <code>'rho'</code>
<code>'tf.v1.train.AdadeltaOptimizer'</code>	<code>'tf.keras.optimizers.Adadelta'</code>	None
<code>'tf.v1.train.AdagradOptimizer'</code>	<code>'tf.keras.optimizers.Adagrad'</code>	None
<code>'tf.v1.train.FtrlOptimizer'</code>	<code>'tf.keras.optimizers.Ftrl'</code>	Remove the <code>'accum_name'</code> and <code>'linear_name'</code> arguments
<code>'tf.contrib.AdamaxOptimizer'</code>	<code>'tf.keras.optimizers.Adamax'</code>	Rename the <code>'beta1'</code> , and <code>'beta2'</code> arguments to <code>'beta_1'</code> and <code>'beta_2'</code>
<code>'tf.contrib.Nadam'</code>	<code>'tf.keras.optimizers.Nadam'</code>	Rename the <code>'beta1'</code> , and <code>'beta2'</code> arguments to <code>'beta_1'</code> and <code>'beta_2'</code>

Note: In TF2, all epsilons (numerical stability constants) now default to `1e-7` instead of `1e-8`. This difference is negligible in most use cases.

Copyright 2021 The TensorFlow Authors.

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");  
# you may not use this file except in compliance with the License.  
# You may obtain a copy of the License at  
#  
# https://www.apache.org/licenses/LICENSE-2.0  
#  
# Unless required by applicable law or agreed to in writing, software  
# distributed under the License is distributed on an "AS IS" BASIS,  
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
# See the License for the specific language governing permissions and  
# limitations under the License.
```

## Migrating model checkpoints



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/migrating_checkpoints)

([https://www.tensorflow.org/guide/migrate/migrating\\_checkpoints](https://www.tensorflow.org/guide/migrate/migrating_checkpoints))



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/migrating_checkpoints.ipynb)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/migrating\\_checkpoints.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/migrating_checkpoints.ipynb))

Note: Checkpoints saved with `tf.compat.v1.Saver` are often referred as *TF1 or name-based* checkpoints. Checkpoints saved with `tf.train.Checkpoint` are referred as *TF2 or object-based* checkpoints.

## Overview

This guide assumes that you have a model that saves and loads checkpoints with `tf.compat.v1.Saver` ([https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/train/Saver](https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/Saver)), and want to migrate the code use the TF2 `tf.train.Checkpoint` API, or use pre-existing checkpoints in your TF2 model.

Below are some common scenarios that you may encounter:

### Scenario 1

There are existing TF1 checkpoints from previous training runs that need to be loaded or converted to TF2.

- To load the TF1 checkpoint in TF2, see the snippet [Load a TF1 checkpoint in TF2](#).
- To convert the checkpoint to TF2, see [Checkpoint conversion](#).

### Scenario 2

You are adjusting your model in a way that risks changing variable names and paths (such as when incrementally migrating away from `get_variable` to explicit `tf.Variable` creation), and would like to maintain saving/loading of existing checkpoints along the way.

See the section on [How to maintain checkpoint compatibility during model migration](#)

### Scenario 3

You are migrating your training code and checkpoints to TF2, but your inference pipeline continues to require TF1 checkpoints for now (for production stability).

#### Option 1

Save both TF1 and TF2 checkpoints when training.

- see [Save a TF1 checkpoint in TF2](#)

#### Option 2

Convert the TF2 checkpoint to TF1.

- see [Checkpoint conversion](#)

---

The examples below show all the combinations of saving and loading checkpoints in TF1/TF2, so you have some flexibility in determining how to migrate your model.

## Setup

In [ ]:

```
import tensorflow as tf
import tensorflow.compat.v1 as tf1

def print_checkpoint(save_path):
    reader = tf.train.load_checkpoint(save_path)
    shapes = reader.get_variable_to_shape_map()
    dtypes = reader.get_variable_to_dtype_map()
    print(f"Checkpoint at '{save_path}':")
    for key in shapes:
        print(f"  (key='{key}', shape={shapes[key]}, dtype={dtypes[key].name}, "
              f"value={reader.get_tensor(key)})")
```

## Changes from TF1 to TF2

This section is included if you are curious about what has changed between TF1 and TF2, and what we mean by "name-based" (TF1) vs "object-based" (TF2) checkpoints.

The two types of checkpoints are actually saved in the same format, which is essentially a key-value table. The difference lies in how the keys are generated.

The keys in named-based checkpoints are the **names of the variables**. The keys in object-based checkpoints refer to the **path from the root object to the variable** (the examples below will help to get a better sense of what this means).

First, save some checkpoints:

In [ ]:

```
with tf.Graph().as_default() as g:
    a = tf.get_variable('a', shape=[], dtype=tf.float32,
                        initializer=tf.zeros_initializer())
    b = tf.get_variable('b', shape=[], dtype=tf.float32,
                        initializer=tf.zeros_initializer())
    c = tf.get_variable('scoped/c', shape=[], dtype=tf.float32,
                        initializer=tf.zeros_initializer())
with tf.Session() as sess:
    saver = tf.train.Saver()
    sess.run(a.assign(1))
    sess.run(b.assign(2))
    sess.run(c.assign(3))
    saver.save(sess, 'tf1-ckpt')

print_checkpoint('tf1-ckpt')
```

In [ ]:

```
a = tf.Variable(5.0, name='a')
b = tf.Variable(6.0, name='b')
with tf.name_scope('scoped'):
    c = tf.Variable(7.0, name='c')

ckpt = tf.train.Checkpoint(variables=[a, b, c])
save_path_v2 = ckpt.save('tf2-ckpt')
print_checkpoint(save_path_v2)
```

If you look at the keys in `tf2-ckpt`, they all refer to the object paths of each variable. For example, variable `a` is the first element in the `variables` list, so its key becomes `variables/0/...` (feel free to ignore the `.ATTRIBUTES/VARIABLE_VALUE` constant).

A closer inspection of the `Checkpoint` object below:

In [ ]:

```
a = tf.Variable(0.)
b = tf.Variable(0.)
c = tf.Variable(0.)
root = ckpt = tf.train.Checkpoint(variables=[a, b, c])
print("root type =", type(root).__name__)
print("root.variables =", root.variables)
print("root.variables[0] =", root.variables[0])
```

Try experimenting with the below snippet and see how the checkpoint keys change with the object structure:

In [ ]:

```
module = tf.Module()
module.d = tf.Variable(0.)
test_ckpt = tf.train.Checkpoint(v={'a': a, 'b': b},
                                c=c,
                                module=module)
test_ckpt_path = test_ckpt.save('root-tf2-ckpt')
print_checkpoint(test_ckpt_path)
```

*Why does TF2 use this mechanism?*

Because there is no more global graph in TF2, variable names are unreliable and can be inconsistent between programs. TF2 encourages the object-oriented modelling approach where variables are owned by layers, and layers are owned by a model:

```
variable = tf.Variable(...)
layer.variable_name = variable
model.layer_name = layer
```

# How to maintain checkpoint compatibility during model migration

One important step in the migration process is *ensuring that all variables are initialized to the correct values*, which in turn allows you to validate that the ops/functions are doing the correct computations. To accomplish this, you must consider the **checkpoint compatibility** between models in the various stages of migration. Essentially, this section answers the question, *how do I keep using the same checkpoint while changing the model*.

Below are three ways of maintaining checkpoint compatibility, in order of increasing flexibility:

1. The model has the **same variable names** as before.
2. The model has different variable names, and maintains a **assignment map** that maps variable names in the checkpoint to the new names.
3. The model has different variable names, and maintains a **TF2 Checkpoint object** that stores all of the variables.

## When the variable names match

Long title: How to re-use checkpoints when the variable names match.

Short answer: You can directly load the pre-existing checkpoint with either `tf1.train.Saver` or `tf.train.Checkpoint`.

---

If you are using `tf.compat.v1.keras.utils.track_tf1_style_variables`, then it will ensure that your model variable names are the same as before. You can also manually ensure that variable names match.

When the variable names match in the migrated models, you may directly use either `tf.train.Checkpoint` or `tf.compat.v1.train.Saver` to load the checkpoint. Both APIs are compatible with eager and graph mode, so you can use them at any stage of the migration.

Note: You can use `tf.train.Checkpoint` to load TF1 checkpoints, but you cannot use `tf.compat.v1.Saver` to load TF2 checkpoints without complicated name matching.

Below are examples of using the same checkpoint with different models. First, save a TF1 checkpoint with `tf1.train.Saver`:

In [ ]:

```
with tf.Graph().as_default() as g:  
    a = tf1.get_variable('a', shape=[], dtype=tf.float32,  
                         initializer=tf1.zeros_initializer())  
    b = tf1.get_variable('b', shape=[], dtype=tf.float32,  
                         initializer=tf1.zeros_initializer())  
    c = tf1.get_variable('scoped/c', shape=[], dtype=tf.float32,  
                         initializer=tf1.zeros_initializer())  
    with tf1.Session() as sess:  
        saver = tf1.train.Saver()  
        sess.run(a.assign(1))  
        sess.run(b.assign(2))  
        sess.run(c.assign(3))  
        save_path = saver.save(sess, 'tf1-ckpt')  
print_checkpoint(save_path)
```

The example below uses `tf.compat.v1.Saver` to load the checkpoint while in eager mode:

In [ ]:

```
a = tf.Variable(0.0, name='a')  
b = tf.Variable(0.0, name='b')  
with tf.name_scope('scoped'):  
    c = tf.Variable(0.0, name='c')  
  
# With the removal of collections in TF2, you must pass in the list of variables  
# to the Saver object:  
saver = tf1.train.Saver(var_list=[a, b, c])  
saver.restore(sess=None, save_path=save_path)  
print(f"loaded values of [a, b, c]: [{a.numpy()}, {b.numpy()}, {c.numpy()}]")  
  
# Saving also works in eager (sess must be None).  
path = saver.save(sess=None, save_path='tf1-ckpt-saved-in-eager')  
print_checkpoint(path)
```

The next snippet loads the checkpoint using the TF2 API `tf.train.Checkpoint`:

In [ ]:

```
a = tf.Variable(0.0, name='a')
b = tf.Variable(0.0, name='b')
with tf.name_scope('scoped'):
    c = tf.Variable(0.0, name='c')

# Without the name_scope, name="scoped/c" works too:
c_2 = tf.Variable(0.0, name='scoped/c')

print("Variable names: ")
print(f"  a.name = {a.name}")
print(f"  b.name = {b.name}")
print(f"  c.name = {c.name}")
print(f"  c_2.name = {c_2.name}")

# Restore the values with tf.train.Checkpoint
ckpt = tf.train.Checkpoint(variables=[a, b, c, c_2])
ckpt.restore(save_path)
print(f"loaded values of [a, b, c, c_2]: [{a.numpy()}, {b.numpy()}, {c.numpy()}, {c_2.numpy()}]")
```

## Variable names in TF2

- Variables still all have a `name` argument you can set.
- Keras models also take a `name` argument as which they set as the prefix for their variables.
- The `v1.name_scope` function can be used to set variable name prefixes. This is very different from `tf.variable_scope`. It only affects names, and doesn't track variables and reuse.

The `tf.compat.v1.keras.utils.track_tf1_style_variables` decorator is a shim that helps you maintain variable names and TF1 checkpoint compatibility, by keeping the naming and reuse semantics of `tf.variable_scope` and `tf.compat.v1.get_variable` unchanged. See the [Model mapping guide](#) ([./model\\_mapping.ipynb](#)) for more info.

**Note 1: If you are using the shim, use TF2 APIs to load your checkpoints (even when using pre-trained TF1 checkpoints).**

See the section *Checkpoint Keras*.

**Note 2: When migrating to `tf.Variable` from `get_variable`:**

If your shim-decorated layer or module consists of some variables (or Keras layers/models) that use `tf.Variable` instead of `tf.compat.v1.get_variable` and get attached as properties/tracked in an object oriented way, they may have different variable naming semantics in TF1.x graphs/sessions versus during eager execution.

In short, *the names may not be what you expect them to be* when running in TF2.

Warning: Variables may have duplicate names in eager execution, which may cause problems if multiple variables in the name-based checkpoint need to be mapped to the same name. You may be able to explicitly adjust the layer and variable names using `tf.name_scope` and layer constructor or `tf.Variable` `name` arguments to adjust variable names and ensure there are no duplicates.

## Maintaining assignment maps

Assignment maps are commonly used to transfer weights between TF1 models, and can also be used during your model migration if the variable names change.

You can use these maps with `tf.compat.v1.train.init_from_checkpoint` ([https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/train/init\\_from\\_checkpoint](https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/init_from_checkpoint)), `tf.compat.v1.train.Saver` ([https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/train/Saver](https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/Saver)), and `tf.train.load_checkpoint` ([https://www.tensorflow.org/api\\_docs/python/tf/train/load\\_checkpoint](https://www.tensorflow.org/api_docs/python/tf/train/load_checkpoint)) to load weights into models in which the variable or scope names may have changed.

The examples in this section will use a previously saved checkpoint:

In [ ]:

```
print_checkpoint('tf1-ckpt')
```

## Loading with `init_from_checkpoint`

`tf1.train.init_from_checkpoint` ([https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/train/init\\_from\\_checkpoint](https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/init_from_checkpoint)) must be called while in a Graph/Session, because it places the values in the variable initializers instead of creating an assign op.

You can use the `assignment_map` argument to configure how the variables are loaded. From the documentation:

Assignment map supports following syntax:

- '`checkpoint_scope_name/`': '`scope_name/`' - will load all variables in current `scope_name` from `checkpoint_scope_name` with matching tensor names.
- '`checkpoint_scope_name/some_other_variable`': '`scope_name/variable_name`' - will initialize `scope_name/variable_name` variable from `checkpoint_scope_name/some_other_variable`.
- '`scope_variable_name`': `variable` - will initialize given `tf.Variable` object with tensor '`scope_variable_name`' from the checkpoint.
- '`scope_variable_name`': `list(variable)` - will initialize list of partitioned variables with tensor '`scope_variable_name`' from the checkpoint.
- '`/`': '`scope_name/`' - will load all variables in current `scope_name` from checkpoint's root (e.g. no scope).

In [ ]:

```
# Restoring with tf1.train.init_from_checkpoint:

# A new model with a different scope for the variables.
with tf.Graph().as_default() as g:
    with tf1.variable_scope('new_scope'):
        a = tf1.get_variable('a', shape=[], dtype=tf.float32,
                             initializer=tf1.zeros_initializer())
        b = tf1.get_variable('b', shape=[], dtype=tf.float32,
                             initializer=tf1.zeros_initializer())
        c = tf1.get_variable('scoped/c', shape=[], dtype=tf.float32,
                             initializer=tf1.zeros_initializer())
    with tf1.Session() as sess:
        # The assignment map will remap all variables in the checkpoint to the
        # new scope:
        tf1.train.init_from_checkpoint(
            'tf1-ckpt',
            assignment_map={'/': 'new_scope/'})
        # `init_from_checkpoint` adds the initializers to these variables.
        # Use `sess.run` to run these initializers.
        sess.run(tf1.global_variables_initializer())
    print("Restored [a, b, c]: ", sess.run([a, b, c]))
```

## Loading with `tf1.train.Saver`

Unlike `init_from_checkpoint`, `tf.compat.v1.train.Saver` ([https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/train/Saver](https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/Saver)) runs in both graph and eager mode. The `var_list` argument optionally accepts a dictionary, except it must map variable names to the `tf.Variable` object.

In [ ]:

```
# Restoring with tf1.train.Saver (works in both graph and eager):

# A new model with a different scope for the variables.
with tf1.variable_scope('new_scope'):
    a = tf1.get_variable('a', shape=[], dtype=tf.float32,
                         initializer=tf1.zeros_initializer())
    b = tf1.get_variable('b', shape=[], dtype=tf.float32,
                         initializer=tf1.zeros_initializer())
    c = tf1.get_variable('scoped/c', shape=[], dtype=tf.float32,
                         initializer=tf1.zeros_initializer())
# Initialize the saver with a dictionary with the original variable names:
saver = tf1.train.Saver({'a': a, 'b': b, 'scoped/c': c})
saver.restore(sess=None, save_path='tf1-ckpt')
print("Restored [a, b, c]: ", [a.numpy(), b.numpy(), c.numpy()])
```

## Loading with `tf.train.load_checkpoint`

This option is for you if you need precise control over the variable values. Again, this works in both graph and eager modes.

In [ ]:

```
# Restoring with tf.train.load_checkpoint (works in both graph and eager):

# A new model with a different scope for the variables.
with tf.Graph().as_default() as g:
    with tf1.variable_scope('new_scope'):
        a = tf1.get_variable('a', shape=[], dtype=tf.float32,
                             initializer=tf1.zeros_initializer())
        b = tf1.get_variable('b', shape=[], dtype=tf.float32,
                             initializer=tf1.zeros_initializer())
        c = tf1.get_variable('scoped/c', shape=[], dtype=tf.float32,
                             initializer=tf1.zeros_initializer())
    with tf1.Session() as sess:
        # It may be easier writing a loop if your model has a lot of variables.
        reader = tf.train.load_checkpoint('tf1-ckpt')
        sess.run(a.assign(reader.get_tensor('a')))
        sess.run(b.assign(reader.get_tensor('b')))
        sess.run(c.assign(reader.get_tensor('scoped/c')))
    print("Restored [a, b, c]: ", sess.run([a, b, c]))
```

## Maintaining a TF2 Checkpoint object

If the variable and scope names may change a lot during the migration, then use `tf.train.Checkpoint` and TF2 checkpoints. TF2 uses the **object structure** instead of variable names (more details in *Changes from TF1 to TF2*).

In short, when creating a `tf.train.Checkpoint` to save or restore checkpoints, make sure it uses the same **ordering** (for lists) and **keys** (for dictionaries and keyword arguments to the `Checkpoint` initializer). Some examples of checkpoint compatibility:

```
ckpt = tf.train.Checkpoint(foo=[var_a, var_b])

# compatible with ckpt
tf.train.Checkpoint(foo=[var_a, var_b])

# not compatible with ckpt
tf.train.Checkpoint(foo=[var_b, var_a])
tf.train.Checkpoint(bar=[var_a, var_b])
```

The code samples below show how to use the "same" `tf.train.Checkpoint` to load variables with different names. First, save a TF2 checkpoint:

In [ ]:

```
with tf.Graph().as_default() as g:
    a = tf1.get_variable('a', shape=[], dtype=tf.float32,
                         initializer=tf1.constant_initializer(1))
    b = tf1.get_variable('b', shape=[], dtype=tf.float32,
                         initializer=tf1.constant_initializer(2))
    with tf1.variable_scope('scoped'):
        c = tf1.get_variable('c', shape=[], dtype=tf.float32,
                             initializer=tf1.constant_initializer(3))
    with tf1.Session() as sess:
        sess.run(tf1.global_variables_initializer())
        print("[a, b, c]: ", sess.run([a, b, c]))

    # Save a TF2 checkpoint
    ckpt = tf.train.Checkpoint(unscoped=[a, b], scoped=[c])
    tf2_ckpt_path = ckpt.save('tf2-ckpt')
    print_checkpoint(tf2_ckpt_path)
```

You can keep using `tf.train.Checkpoint` even if the variable/scope names change:

In [ ]:

```
with tf.Graph().as_default() as g:
    a = tf.get_variable('a_different_name', shape=[], dtype=tf.float32,
                        initializer=tf.zeros_initializer())
    b = tf.get_variable('b_different_name', shape=[], dtype=tf.float32,
                        initializer=tf.zeros_initializer())
    with tf.variable_scope('different_scope'):
        c = tf.get_variable('c', shape=[], dtype=tf.float32,
                            initializer=tf.zeros_initializer())
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        print("Initialized [a, b, c]: ", sess.run([a, b, c]))

    ckpt = tf.train.Checkpoint(unscoped=[a, b], scoped=[c])
    # `assert_consumed` validates that all checkpoint objects are restored from
    # the checkpoint. `run_restore_ops` is required when running in a TF1
    # session.
    ckpt.restore(tf2_ckpt_path).assert_consumed().run_restore_ops()

    # Removing `assert_consumed` is fine if you want to skip the validation.
    # ckpt.restore(tf2_ckpt_path).run_restore_ops()

    print("Restored [a, b, c]: ", sess.run([a, b, c]))
```

And in eager mode:

In [ ]:

```
a = tf.Variable(0.)
b = tf.Variable(0.)
c = tf.Variable(0.)
print("Initialized [a, b, c]: ", [a.numpy(), b.numpy(), c.numpy()])

# The keys "scoped" and "unscoped" are no longer relevant, but are used to
# maintain compatibility with the saved checkpoints.
ckpt = tf.train.Checkpoint(unscoped=[a, b], scoped=[c])

ckpt.restore(tf2_ckpt_path).assert_consumed().run_restore_ops()
print("Restored [a, b, c]: ", [a.numpy(), b.numpy(), c.numpy()])
```

## TF2 checkpoints in Estimator

The sections above describe how to maintain checkpoint compatibility while migrating your model. These concepts also apply for Estimator models, although the way the checkpoint is saved/loaded is slightly different. As you migrate your Estimator model to use TF2 APIs, you may want to switch from TF1 to TF2 checkpoints *while the model is still using the estimator*. This section shows how to do so.

`tf.estimator.Estimator` ([https://www.tensorflow.org/api\\_docs/python/tf/estimator/Estimator](https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator)) and `MonitoredSession` ([https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/train/monitored\\_session](https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/monitored_session)) have a saving mechanism called the `scaffold`, a `tf.compat.v1.train.Scaffold` ([https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/train/Scaffold](https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/Scaffold)) object. The `Scaffold` can contain a `tf1.train.Saver` or `tf.train.Checkpoint`, which enables `Estimator` and `MonitoredSession` to save TF1- or TF2-style checkpoints.

In [ ]:

```
# A model_fn that saves a TF1 checkpoint
def model_fn_tf1_ckpt(features, labels, mode):
    # This model adds 2 to the variable `v` in every train step.
    train_step = tf1.train.get_or_create_global_step()
    v = tf1.get_variable('var', shape=[], dtype=tf.float32,
                         initializer=tf1.constant_initializer(0))
    return tf.estimator.EstimatorSpec(
        mode,
        predictions=v,
        train_op= tf.group(v.assign_add(2), train_step.assign_add(1)),
        loss= tf.constant(1.),
        scaffold=None
    )

!rm -rf est-tf1
est = tf.estimator.Estimator(model_fn_tf1_ckpt, 'est-tf1')

def train_fn():
    return tf.data.Dataset.from_tensor_slices(([1,2,3], [4,5,6]))
est.train(train_fn, steps=1)

latest_checkpoint = tf.train.latest_checkpoint('est-tf1')
print_checkpoint(latest_checkpoint)
```

In [ ]:

```
# A model_fn that saves a TF2 checkpoint
def model_fn_tf2_ckpt(features, labels, mode):
    # This model adds 2 to the variable `v` in every train step.
    train_step = tf1.train.get_or_create_global_step()
    v = tf1.get_variable('var', shape=[], dtype=tf.float32,
                         initializer=tf1.constant_initializer(0))
    ckpt = tf.train.Checkpoint(var_list={'var': v}, step=train_step)
    return tf.estimator.EstimatorSpec(
        mode,
        predictions=v,
        train_op=tf.group(v.assign_add(2), train_step.assign_add(1)),
        loss=tf.constant(1.),
        scaffold=tf1.train.Scaffold(saver=ckpt)
    )

!rm -rf est-tf2
est = tf.estimator.Estimator(model_fn_tf2_ckpt, 'est-tf2',
                             warm_start_from='est-tf1')

def train_fn():
    return tf.data.Dataset.from_tensor_slices(([1,2,3], [4,5,6]))
est.train(train_fn, steps=1)

latest_checkpoint = tf.train.latest_checkpoint('est-tf2')
print_checkpoint(latest_checkpoint)

assert est.get_variable_value('var_list/var/.ATTRIBUTES/VARIABLE_VALUE') == 4
```

The final value of `v` should be `16`, after being warm-started from `est-tf1`, then trained for an additional 5 steps. The train step value doesn't carry over from the `warm_start` checkpoint.

## Checkpointing Keras

Models built with Keras still use `tf1.train.Saver` and `tf.train.Checkpoint` to load pre-existing weights. When your model is fully migrated, switch to using `model.save_weights` and `model.load_weights`, especially if you are using the `ModelCheckpoint` callback when training.

Some things you should know about checkpoints and Keras:

### Initialization vs Building

Keras models and layers must go through **two steps** before being fully created. First is the *initialization* of the Python object: `layer = tf.keras.layers.Dense(x)`. Second is the *build* step, in which most of the weights are actually created: `layer.build(input_shape)`. You can also build a model by calling it or running a single `train`, `eval`, or `predict` step (the first time only).

If you find that `model.load_weights(path).assert_consumed()` is raising an error, then it is likely that the model/layers have not been built.

### Keras uses TF2 checkpoints

`tf.train.Checkpoint(model).write` is equivalent to `model.save_weights`. Same with `tf.train.Checkpoint(model).read` and `model.load_weights`. Note that `Checkpoint(model) != Checkpoint(model=model)`.

### TF2 checkpoints work with Keras's `build()` step

`tf.train.Checkpoint.restore` has a mechanism called *deferred restoration* which allows `tf.Module` and Keras objects to store variable values if the variable has not yet been created. This allows *initialized* models to load weights and *build* after.

```
m = YourKerasModel()
status = m.load_weights(path)

# This call builds the model. The variables are created with the restored
# values.
m.predict(inputs)

status.assert_consumed()
```

Because of this mechanism, we highly recommend that you use TF2 checkpoint loading APIs with Keras models (even when restoring pre-existing TF1 checkpoints into the [model mapping shims \(`./model\_mapping.ipynb`\)](#)). See more in the [checkpoint guide](#) ([https://www.tensorflow.org/guide/checkpoint#delayed\\_restorations](https://www.tensorflow.org/guide/checkpoint#delayed_restorations)).

## Code Snippets

The snippets below show the TF1/TF2 version compatibility in the checkpoint saving APIs.

## Save a TF1 checkpoint in TF2

In [ ]:

```
a = tf.Variable(1.0, name='a')
b = tf.Variable(2.0, name='b')
with tf.name_scope('scoped'):
    c = tf.Variable(3.0, name='c')

saver = tf.train.Saver(var_list=[a, b, c])
path = saver.save(sess=None, save_path='tf1-ckpt-saved-in-eager')
print_checkpoint(path)
```

## Load a TF1 checkpoint in TF2

In [ ]:

```
a = tf.Variable(0., name='a')
b = tf.Variable(0., name='b')
with tf.name_scope('scoped'):
    c = tf.Variable(0., name='c')
print("Initialized [a, b, c]: ", [a.numpy(), b.numpy(), c.numpy()])
saver = tf.train.Saver(var_list=[a, b, c])
saver.restore(sess=None, save_path='tf1-ckpt-saved-in-eager')
print("Restored [a, b, c]: ", [a.numpy(), b.numpy(), c.numpy()])
```

## Save a TF2 checkpoint in TF1

In [ ]:

```
with tf.Graph().as_default() as g:
    a = tf.get_variable('a', shape=[], dtype=tf.float32,
                        initializer=tf.constant_initializer(1))
    b = tf.get_variable('b', shape=[], dtype=tf.float32,
                        initializer=tf.constant_initializer(2))
    with tf.variable_scope('scoped'):
        c = tf.get_variable('c', shape=[], dtype=tf.float32,
                            initializer=tf.constant_initializer(3))
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        ckpt = tf.train.Checkpoint(
            var_list={v.name.split(':')[0]: v for v in tf.global_variables()})
        tf2_in_tf1_path = ckpt.save('tf2-ckpt-saved-in-session')
        print_checkpoint(tf2_in_tf1_path)
```

## Load a TF2 checkpoint in TF1

In [ ]:

```
with tf.Graph().as_default() as g:
    a = tf.get_variable('a', shape=[], dtype=tf.float32,
                        initializer=tf.constant_initializer(0))
    b = tf.get_variable('b', shape=[], dtype=tf.float32,
                        initializer=tf.constant_initializer(0))
    with tf.variable_scope('scoped'):
        c = tf.get_variable('c', shape=[], dtype=tf.float32,
                            initializer=tf.constant_initializer(0))
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        print("Initialized [a, b, c]: ", sess.run([a, b, c]))
        ckpt = tf.train.Checkpoint(
            var_list={v.name.split(':')[0]: v for v in tf.global_variables()})
        ckpt.restore('tf2-ckpt-saved-in-session-1').run_restore_ops()
        print("Restored [a, b, c]: ", sess.run([a, b, c]))
```

## Checkpoint conversion

You can convert checkpoints between TF1 and TF2 by loading and re-saving the checkpoints. An alternative is `tf.train.load_checkpoint`, shown in the code below.

### Convert TF1 checkpoint to TF2

In [ ]:

```
def convert_tf1_to_tf2(checkpoint_path, output_prefix):
    """Converts a TF1 checkpoint to TF2.

    To load the converted checkpoint, you must build a dictionary that maps
    variable names to variable objects.
    ```

    ckpt = tf.train.Checkpoint(vars={name: variable})
    ckpt.restore(converted_ckpt_path)
    ```

    Args:
        checkpoint_path: Path to the TF1 checkpoint.
        output_prefix: Path prefix to the converted checkpoint.

    Returns:
        Path to the converted checkpoint.
    """
    vars = {}
    reader = tf.train.load_checkpoint(checkpoint_path)
    dtypes = reader.get_variable_to_dtype_map()
    for key in dtypes.keys():
        vars[key] = tf.Variable(reader.get_tensor(key))
    return tf.train.Checkpoint(vars=vars).save(output_prefix)
```

Convert the checkpoint saved in the snippet Save a TF1 checkpoint in TF2 :

In [ ]:

```
# Make sure to run the snippet in `Save a TF1 checkpoint in TF2`.
print_checkpoint('tf1-ckpt-saved-in-eager')
converted_path = convert_tf1_to_tf2('tf1-ckpt-saved-in-eager',
                                    'converted-tf1-to-tf2')
print("\n[Converted]")
print_checkpoint(converted_path)

# Try loading the converted checkpoint.
a = tf.Variable(0.)
b = tf.Variable(0.)
c = tf.Variable(0.)
ckpt = tf.train.Checkpoint(vars={'a': a, 'b': b, 'scoped/c': c})
ckpt.restore(converted_path).assert_consumed()
print("\nRestored [a, b, c]: ", [a.numpy(), b.numpy(), c.numpy()])
```

## Convert TF2 checkpoint to TF1

In [ ]:

```
def convert_tf2_to_tf1(checkpoint_path, output_prefix):
    """Converts a TF2 checkpoint to TF1.

    The checkpoint must be saved using a
    `tf.train.Checkpoint(var_list={name: variable})`.

    To load the converted checkpoint with `tf.compat.v1.Saver`:
    ```

    saver = tf.compat.v1.train.Saver(var_list={name: variable})

    # An alternative, if the variable names match the keys:
    saver = tf.compat.v1.train.Saver(var_list=[variables])
    saver.restore(sess, output_path)
    ````

    ````

    vars = {}
    reader = tf.train.load_checkpoint(checkpoint_path)
    dtypes = reader.get_variable_to_dtype_map()
    for key in dtypes.keys():
        # Get the "name" from the
        if key.startswith('var_list/'):
            var_name = key.split('/')[1]
            # TF2 checkpoint keys use '/', so if they appear in the user-defined name,
            # they are escaped to '.S'.
            var_name = var_name.replace('.S', '/')
            vars[var_name] = tf.Variable(reader.get_tensor(key))

    return tf1.train.Saver(var_list=vars).save(sess=None, save_path=output_prefix)
```

Convert the checkpoint saved in the snippet Save a TF2 checkpoint in TF1 :

In [ ]:

```
# Make sure to run the snippet in `Save a TF2 checkpoint in TF1`.
print_checkpoint('tf2-ckpt-saved-in-session-1')
converted_path = convert_tf2_to_tf1('tf2-ckpt-saved-in-session-1',
                                    'converted-tf2-to-tf1')
print("\n[Converted]")
print_checkpoint(converted_path)

# Try loading the converted checkpoint.
with tf.Graph().as_default() as g:
    a = tf1.get_variable('a', shape=[], dtype=tf.float32,
                         initializer=tf1.constant_initializer(0))
    b = tf1.get_variable('b', shape=[], dtype=tf.float32,
                         initializer=tf1.constant_initializer(0))
    with tf1.variable_scope('scoped'):
        c = tf1.get_variable('c', shape=[], dtype=tf.float32,
                             initializer=tf1.constant_initializer(0))
    with tf1.Session() as sess:
        saver = tf1.train.Saver([a, b, c])
        saver.restore(sess, converted_path)
    print("\nRestored [a, b, c]: ", sess.run([a, b, c]))
```

## Related Guides

- [Validating numerical equivalence and correctness \(./validate\\_correctness.ipynb\)](#)
- [Model mapping guide \(./model\\_mapping.ipynb\)](#) and `tf.compat.v1.keras.utils.track_tf1_style_variables`
- [TF2 Checkpoint guide \(<https://www.tensorflow.org/guide/checkpoint>\)](#)

Copyright 2021 The TensorFlow Authors.

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Migrate the fault tolerance mechanism



[View on TensorFlow.org  
\(\[https://www.tensorflow.org/guide/migrate/fault\\\_tolerance\]\(https://www.tensorflow.org/guide/migrate/fault\_tolerance\)\)](https://www.tensorflow.org/guide/migrate/fault_tolerance)



[Run in Google Colab  
\(\[https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/fault\\\_tolerance.ipynb\]\(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/fault\_tolerance.ipynb\)\)](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/fault_tolerance.ipynb)

Fault tolerance refers to a mechanism of periodically saving the states of trackable objects, such as parameters and models. This enables you to recover them in the event of a program/machine failure during training.

This guide first demonstrates how to add fault tolerance to training with `tf.estimator.Estimator` in TensorFlow 1 by specifying metric saving with `tf.estimator.RunConfig`. Then, you will learn how to implement fault tolerance for training in Tensorflow 2 in two ways:

- If you use the Keras `Model.fit` API, you can pass the `tf.keras.callbacks.BackupAndRestore` callback to it.
- If you use a custom training loop (with `tf.GradientTape`), you can arbitrarily save checkpoints using the `tf.train.Checkpoint` and `tf.train.CheckpointManager` APIs.

Both of these methods will back up and restore the training states in [checkpoint \(../../guide/checkpoint.ipynb\)](#) files.

## Setup

```
In [ ]:
```

```
import tensorflow.compat.v1 as tf1
import tensorflow as tf
import numpy as np
import tempfile
import time
```

```
In [ ]:
```

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

## TensorFlow 1: Save checkpoints with `tf.estimator.RunConfig`

In TensorFlow 1, you can configure a `tf.estimator` to save checkpoints every step by configuring `tf.estimator.RunConfig`.

In this example, start by writing a hook that artificially throws an error during the fifth checkpoint:

```
In [ ]:
```

```
class InterruptHook(tf1.train.SessionRunHook):
    # A hook for artificially interrupting training.
    def begin(self):
        self._step = -1

    def before_run(self, run_context):
        self._step += 1

    def after_run(self, run_context, run_values):
        if self._step == 5:
            raise RuntimeError('Interruption')
```

Next, configure `tf.estimator.Estimator` to save every checkpoint and use the MNIST dataset:

```
In [ ]:
```

```
feature_columns = [tf1.feature_column.numeric_column("x", shape=[28, 28])]
config = tf1.estimator.RunConfig(save_summary_steps=1,
                                 save_checkpoints_steps=1)

path = tempfile.mkdtemp()

classifier = tf1.estimator.DNNClassifier(
    feature_columns=feature_columns,
    hidden_units=[256, 32],
    optimizer=tf1.train.AdamOptimizer(0.001),
    n_classes=10,
    dropout=0.2,
    model_dir=path,
    config = config
)

train_input_fn = tf1.estimator.inputs.numpy_input_fn(
    x={"x": x_train},
    y=y_train.astype(np.int32),
    num_epochs=10,
    batch_size=50,
    shuffle=True,
)
```

Begin training the model. An artificial exception will be raised by the hook you defined earlier.

```
In [ ]:
```

```
try:
    classifier.train(input_fn=train_input_fn,
                      hooks=[InterruptHook()],
                      max_steps=10)
except Exception as e:
    print(f'{type(e).__name__}:{e}')
```

Rebuild the `tf.estimator.Estimator` using the last saved checkpoint and continue training:

In [ ]:

```
classifier = tf1.estimator.DNNClassifier(  
    feature_columns=feature_columns,  
    hidden_units=[256, 32],  
    optimizer=tf1.train.AdamOptimizer(0.001),  
    n_classes=10,  
    dropout=0.2,  
    model_dir=path,  
    config = config  
)  
classifier.train(input_fn=train_input_fn,  
                  max_steps = 10)
```

## TensorFlow 2: Back up and restore with a callback and Model.fit

In TensorFlow 2, if you use the Keras `Model.fit` API for training, you can provide the `tf.keras.callbacks.BackupAndRestore` callback to add the fault tolerance functionality.

To help demonstrate this, let's first start by defining a callback class that artificially throws an error during the fifth checkpoint:

In [ ]:

```
class InterruptingCallback(tf.keras.callbacks.Callback):  
    # A callback for artificially interrupting training.  
    def on_epoch_end(self, epoch, log=None):  
        if epoch == 4:  
            raise RuntimeError('Interruption')
```

Then, define and instantiate a simple Keras model, define the loss function, call `Model.compile`, and set up a `tf.keras.callbacks.BackupAndRestore` callback that will save the checkpoints in a temporary directory:

In [ ]:

```
def create_model():  
    return tf.keras.models.Sequential([  
        tf.keras.layers.Flatten(input_shape=(28, 28)),  
        tf.keras.layers.Dense(512, activation='relu'),  
        tf.keras.layers.Dropout(0.2),  
        tf.keras.layers.Dense(10)  
    ])  
  
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)  
  
model = create_model()  
model.compile(optimizer='adam',  
              loss=loss,  
              metrics=['accuracy'],  
              steps_per_execution=10)  
  
log_dir = tempfile.mkdtemp()  
  
backup_restore_callback = tf.keras.callbacks.BackupAndRestore(  
    backup_dir = log_dir  
)
```

Now, start training the model with `Model.fit`. During training, checkpoints will be saved thanks to the `backup_restore_callback` defined above, while the `InterruptingCallback` will raise an artificial exception to simulate a failure.

In [ ]:

```
try:  
    model.fit(x=x_train,  
              y=y_train,  
              epochs=10,  
              validation_data=(x_test, y_test),  
              callbacks=[backup_restore_callback, InterruptingCallback()])  
except Exception as e:  
    print(f'{type(e).__name__}:{e}')
```

Next, instantiate the Keras model, call `Model.compile`, and continue training the model with `Model.fit` from a previously saved checkpoint:

In [ ]:

```
model = create_model()
model.compile(optimizer='adam',
              loss=loss,
              metrics=['accuracy'],
              steps_per_execution=10)
model.fit(x=x_train,
          y=y_train,
          epochs=10,
          validation_data=(x_test, y_test),
          callbacks=[backup_restore_callback])
```

## TensorFlow 2: Write manual checkpoints with a custom training loop

If you use a custom training loop in TensorFlow 2, you can implement a fault tolerance mechanism with the `tf.train.Checkpoint` and `tf.train.CheckpointManager` APIs.

This example demonstrates how to:

- Use a `tf.train.Checkpoint` object to manually create a checkpoint, where the trackable objects you want to save are set as attributes.
- Use a `tf.train.CheckpointManager` to manage multiple checkpoints.

Start by defining and instantiating the Keras model, the optimizer, and the loss function. Then, create a `Checkpoint` that manages two objects with trackable states (the model and the optimizer), as well as a `CheckpointManager` for logging and keeping several checkpoints in a temporary directory.

In [ ]:

```
model = create_model()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
log_dir = tempfile.mkdtemp()
epochs = 5
steps_per_epoch = 5

checkpoint = tf.train.Checkpoint(model=model, optimizer=optimizer)
checkpoint_manager = tf.train.CheckpointManager(
    checkpoint, log_dir, max_to_keep=2)
```

Now, implement a custom training loop where after the first epoch every time a new epoch starts the last checkpoint is loaded:

In [ ]:

```
for epoch in range(epochs):
    if epoch > 0:
        tf.train.load_checkpoint(save_path)
    print(f"\nStart of epoch {epoch}")

    for step in range(steps_per_epoch):
        with tf.GradientTape() as tape:

            logits = model(x_train, training=True)
            loss_value = loss_fn(y_train, logits)

            grads = tape.gradient(loss_value, model.trainable_weights)
            optimizer.apply_gradients(zip(grads, model.trainable_weights))

        save_path = checkpoint_manager.save()
        print(f"Checkpoint saved to {save_path}")
        print(f"Training loss at step {step}: {loss_value}")
```

## Next steps

To learn more about fault tolerance and checkpointing in TensorFlow 2, consider the following documentation:

- The `tf.keras.callbacks.BackupAndRestore` callback API docs.
- The `tf.train.Checkpoint` and `tf.train.CheckpointManager` API docs.
- The [Training checkpoints \(../../guide/checkpoint.ipynb\)](#) guide, including the *Writing checkpoints* section.

You may also find the following material related to [distributed training \(../../guide/distributed\\_training.ipynb\)](#) useful:

- The *Fault tolerance* section in the [Multi-worker training with Keras \(../../tutorials/distribute/multi\\_worker\\_with\\_keras.ipynb\)](#) tutorial.
- The *Handling task failure* section in the [Parameter server training \(../../tutorials/distribute/parameter\\_server\\_training.ipynb\)](#) tutorial.

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Migrate LoggingTensorHook and StopAtStepHook to Keras callbacks



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/logging_stop_hook)

([https://www.tensorflow.org/guide/migrate/logging\\_stop\\_hook](https://www.tensorflow.org/guide/migrate/logging_stop_hook))



[Run in Google Colab](#)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/logging\\_stop\\_hook.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/logging_stop_hook.ipynb))

In TensorFlow 1, you use `tf.estimator.LoggingTensorHook` to monitor and log tensors, while `tf.estimator.StopAtStepHook` helps stop training at a specified step when training with `tf.estimator.Estimator`. This notebook demonstrates how to migrate from these APIs to their equivalents in TensorFlow 2 using custom Keras callbacks (`tf.keras.callbacks.Callback`) with `Model.fit`.

Keras [callbacks](https://www.tensorflow.org/guide/keras/custom_callback) ([https://www.tensorflow.org/guide/keras/custom\\_callback](https://www.tensorflow.org/guide/keras/custom_callback)) are objects that are called at different points during training/evaluation/prediction in the built-in Keras `Model.fit` / `Model.evaluate` / `Model.predict` APIs. You can learn more about callbacks in the `tf.keras.callbacks.Callback` API docs, as well as the [Writing your own callbacks](#) ([../guide/keras/custom\\_callback.ipynb](#)) and [Training and evaluation with the built-in methods](#) ([https://www.tensorflow.org/guide/keras/train\\_and\\_evaluate](https://www.tensorflow.org/guide/keras/train_and_evaluate)) (the *Using callbacks* section) guides. For migrating from `SessionRunHook` in TensorFlow 1 to Keras callbacks in TensorFlow 2, check out the [Migrate training with assisted logic](#) ([sessionrunhook\\_callback.ipynb](#)) guide.

## Setup

Start with imports and a simple dataset for demonstration purposes:

In [ ]:

```
import tensorflow as tf
import tensorflow.compat.v1 as tf1
```

In [ ]:

```
features = [[1., 1.5], [2., 2.5], [3., 3.5]]
labels = [[0.3], [0.5], [0.7]]

# Define an input function.
def _input_fn():
    return tf1.data.Dataset.from_tensor_slices((features, labels)).batch(1)
```

## TensorFlow 1: Log tensors and stop training with `tf.estimator` APIs

In TensorFlow 1, you define various hooks to control the training behavior. Then, you pass these hooks to `tf.estimator.EstimatorSpec`.

In the example below:

- To monitor/log tensors—for example, model weights or losses—you use `tf.estimator.LoggingTensorHook` (`tf.train.LoggingTensorHook` is its alias).
- To stop training at a specific step, you use `tf.estimator.StopAtStepHook` (`tf.train.StopAtStepHook` is its alias).

In [ ]:

```
def _model_fn(features, labels, mode):
    dense = tf1.layers.Dense(1)
    logits = dense(features)
    loss = tf1.losses.mean_squared_error(labels=labels, predictions=logits)
    optimizer = tf1.train.AdagradOptimizer(0.05)
    train_op = optimizer.minimize(loss, global_step=tf1.train.get_global_step())

    # Define the stop hook.
    stop_hook = tf1.train.StopAtStepHook(num_steps=2)

    # Access tensors to be logged by names.
    kernel_name = tf1.identity(dense.weights[0])
    bias_name = tf1.identity(dense.weights[1])
    logging_weight_hook = tf1.train.LoggingTensorHook(
        tensors=[kernel_name, bias_name],
        every_n_iter=1)
    # Log the training loss by the tensor object.
    logging_loss_hook = tf1.train.LoggingTensorHook(
        {'loss from LoggingTensorHook': loss},
        every_n_secs=3)

    # Pass all hooks to `EstimatorSpec`.
    return tf1.estimator.EstimatorSpec(mode,
                                        loss=loss,
                                        train_op=train_op,
                                        training_hooks=[stop_hook,
                                                       logging_weight_hook,
                                                       logging_loss_hook])

estimator = tf1.estimator.Estimator(model_fn=_model_fn)

# Begin training.
# The training will stop after 2 steps, and the weights/loss will also be logged.
estimator.train(_input_fn)
```

## TensorFlow 2: Log tensors and stop training with custom callbacks and Model.fit

In TensorFlow 2, when you use the built-in Keras `Model.fit` (or `Model.evaluate`) for training/evaluation, you can configure tensor monitoring and training stopping by defining custom Keras `tf.keras.callbacks.Callback`s. Then, you pass them to the `callbacks` parameter of `Model.fit` (or `Model.evaluate`). (Learn more in the [Writing your own callbacks \(../../guide/keras/custom\\_callback.ipynb\)](#) guide.)

In the example below:

- To recreate the functionalities of `StopAtStepHook`, define a custom callback (named `StopAtStepCallback` below) where you override the `on_batch_end` method to stop training after a certain number of steps.
- To recreate the `LoggingTensorHook` behavior, define a custom callback (`LoggingTensorCallback`) where you record and output the logged tensors manually, since accessing to tensors by names is not supported. You can also implement the logging frequency inside the custom callback. The example below will print the weights every two steps. Other strategies like logging every N seconds are also possible.

In [ ]:

```
class StopAtStepCallback(tf.keras.callbacks.Callback):
    def __init__(self, stop_step=None):
        super().__init__()
        self._stop_step = stop_step

    def on_batch_end(self, batch, logs=None):
        if self.model.optimizer.iterations >= self._stop_step:
            self.model.stop_training = True
            print('\nstop training now')

class LoggingTensorCallback(tf.keras.callbacks.Callback):
    def __init__(self, every_n_iter):
        super().__init__()
        self._every_n_iter = every_n_iter
        self._log_count = every_n_iter

    def on_batch_end(self, batch, logs=None):
        if self._log_count > 0:
            self._log_count -= 1
            print("Logging Tensor Callback: dense/kernel:",
                  model.layers[0].weights[0])
            print("Logging Tensor Callback: dense/bias:",
                  model.layers[0].weights[1])
            print("Logging Tensor Callback loss:", logs["loss"])
        else:
            self._log_count -= self._every_n_iter
```

When finished, pass the new callbacks—`StopAtStepCallback` and `LoggingTensorCallback`—to the `callbacks` parameter of `Model.fit`:

In [ ]:

```
dataset = tf.data.Dataset.from_tensor_slices((features, labels)).batch(1)
model = tf.keras.models.Sequential([tf.keras.layers.Dense(1)])
optimizer = tf.keras.optimizers.Adagrad(learning_rate=0.05)
model.compile(optimizer, "mse")

# Begin training.
# The training will stop after 2 steps, and the weights/loss will also be logged.
model.fit(dataset, callbacks=[StopAtStepCallback(stop_step=2),
                               LoggingTensorCallback(every_n_iter=2)])
```

## Next steps

Learn more about callbacks in:

- API docs: `tf.keras.callbacks.Callback`
- Guide: [Writing your own callbacks \(./guide/keras/custom\\_callback.ipynb\)](#)
- Guide: [Training and evaluation with the built-in methods \(\[https://www.tensorflow.org/guide/keras/train\\\_and\\\_evaluate\]\(https://www.tensorflow.org/guide/keras/train\_and\_evaluate\)\)](#) (the *Using callbacks* section)

You may also find the following migration-related resources useful:

- The [Early stopping migration guide \(early\\_stopping.ipynb\)](#): `tf.keras.callbacks.EarlyStopping` is a built-in early stopping callback
- The [TensorBoard migration guide \(tensorboard.ipynb\)](#): TensorBoard enables tracking and displaying metrics
- The [Training with assisted logic migration guide \(sessionrunhook\\_callback.ipynb\)](#): From `SessionRunHook` in TensorFlow 1 to Keras callbacks in TensorFlow 2

**Copyright 2021 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

# Migrating feature\_columns to TF2's Keras Preprocessing Layers



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/migrating_feature_columns)



Ru

([https://www.tensorflow.org/guide/migrate/migrating\\_feature\\_columns](https://www.tensorflow.org/guide/migrate/migrating_feature_columns)) ([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/migrating\\_feature\\_columns.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/migrating_feature_columns.ipynb))

Training a model will usually come with some amount of feature preprocessing, particularly when dealing with structured data. When training a `tf.estimator.Estimator` in TF1, this feature preprocessing is usually done with the `tf.feature_column` API. In TF2, this preprocessing can be done directly with Keras layers, called *preprocessing layers*.

In this migration guide, you will perform some common feature transformations using both feature columns and preprocessing layers, followed by training a complete model with both APIs.

First, start with a couple of necessary imports,

In [ ]:

```
import tensorflow as tf
import tensorflow.compat.v1 as tf1
import math
```

and add a utility for calling a feature column for demonstration:

In [ ]:

```
def call_feature_columns(feature_columns, inputs):
    # This is a convenient way to call a `feature_column` outside of an estimator
    # to display its output.
    feature_layer = tf1.keras.layers.DenseFeatures(feature_columns)
    return feature_layer(inputs)
```

## Input handling

To use feature columns with an estimator, model inputs are always expected to be a dictionary of tensors:

In [ ]:

```
input_dict = {
    'foo': tf.constant([1]),
    'bar': tf.constant([0]),
    'baz': tf.constant([-1])
}
```

Each feature column needs to be created with a key to index into the source data. The output of all feature columns is concatenated and used by the estimator model.

In [ ]:

```
columns = [
    tf1.feature_column.numeric_column('foo'),
    tf1.feature_column.numeric_column('bar'),
    tf1.feature_column.numeric_column('baz'),
]
call_feature_columns(columns, input_dict)
```

In Keras, model input is much more flexible. A `tf.keras.Model` can handle a single tensor input, a list of tensor features, or a dictionary of tensor features. You can handle dictionary input by passing a dictionary of `tf.keras.Input` on model creation. Inputs will not be concatenated automatically, which allows them to be used in much more flexible ways. They can be concatenated with `tf.keras.layers.concatenate`.

In [ ]:

```
inputs = {
    'foo': tf.keras.Input(shape=()),
    'bar': tf.keras.Input(shape=()),
    'baz': tf.keras.Input(shape=()),
}
# Inputs are typically transformed by preprocessing layers before concatenation.
outputs = tf.keras.layers.concatenate()(inputs.values())
model = tf.keras.Model(inputs=inputs, outputs=outputs)
model(input_dict)
```

## One-hot encoding integer IDs

A common feature transformation is one-hot encoding integer inputs of a known range. Here is an example using feature columns:

In [ ]:

```
categorical_col = tf1.feature_column.categorical_column_with_identity(  
    'type', num_buckets=3)  
indicator_col = tf1.feature_column.indicator_column(categorical_col)  
call_feature_columns(indicator_col, {'type': [0, 1, 2]})
```

Using Keras preprocessing layers, these columns can be replaced by a single `tf.keras.layers.CategoryEncoding` layer with `output_mode` set to `'one_hot'`:

In [ ]:

```
one_hot_layer = tf.keras.layers.CategoryEncoding(  
    num_tokens=3, output_mode='one_hot')  
one_hot_layer([0, 1, 2])
```

Note: For large one-hot encodings, it is much more efficient to use a sparse representation of the output. If you pass `sparse=True` to the `CategoryEncoding` layer, the output of the layer will be a `tf.sparse.SparseTensor`, which can be efficiently handled as input to a `tf.keras.layers.Dense` layer.

## Normalizing numeric features

When handling continuous, floating-point features with feature columns, you need to use a `tf.feature_column.numeric_column`. In the case where the input is already normalized, converting this to Keras is trivial. You can simply use a `tf.keras.Input` directly into your model, as shown above.

A `numeric_column` can also be used to normalize input:

In [ ]:

```
def normalize(x):  
    mean, variance = (2.0, 1.0)  
    return (x - mean) / math.sqrt(variance)  
numeric_col = tf1.feature_column.numeric_column('col', normalizer_fn=normalize)  
call_feature_columns(numeric_col, {'col': tf.constant([[0.], [1.], [2.]])})
```

In contrast, with Keras, this normalization can be done with `tf.keras.layers.Normalization`.

In [ ]:

```
normalization_layer = tf.keras.layers.Normalization(mean=2.0, variance=1.0)  
normalization_layer(tf.constant([[0.], [1.], [2.]]))
```

## Bucketizing and one-hot encoding numeric features

Another common transformation of continuous, floating point inputs is to bucketize them into integers of a fixed range.

In feature columns, this can be achieved with a `tf.feature_column.bucketized_column`:

In [ ]:

```
numeric_col = tf1.feature_column.numeric_column('col')  
bucketized_col = tf1.feature_column.bucketized_column(numeric_col, [1, 4, 5])  
call_feature_columns(bucketized_col, {'col': tf.constant([1., 2., 3., 4., 5.])})
```

In Keras, this can be replaced by `tf.keras.layers.Discretization`:

In [ ]:

```
discretization_layer = tf.keras.layers.Discretization(bin_boundaries=[1, 4, 5])  
one_hot_layer = tf.keras.layers.CategoryEncoding(  
    num_tokens=4, output_mode='one_hot')  
one_hot_layer(discretization_layer([1., 2., 3., 4., 5.]))
```

## One-hot encoding string data with a vocabulary

Handling string features often requires a vocabulary lookup to translate strings into indices. Here is an example using feature columns to lookup strings and then one-hot encode the indices:

In [ ]:

```
vocab_col = tf1.feature_column.categorical_column_with_vocabulary_list(  
    'sizes',  
    vocabulary_list=['small', 'medium', 'large'],  
    num_oov_buckets=0)  
indicator_col = tf1.feature_column.indicator_column(vocab_col)  
call_feature_columns(indicator_col, {'sizes': ['small', 'medium', 'large']})
```

Using Keras preprocessing layers, use the `tf.keras.layers.StringLookup` layer with `output_mode` set to `'one_hot'`:

In [ ]:

```
string_lookup_layer = tf.keras.layers.StringLookup(  
    vocabulary=['small', 'medium', 'large'],  
    num_oov_indices=0,  
    output_mode='one_hot')  
string_lookup_layer(['small', 'medium', 'large'])
```

Note: For large one-hot encodings, it is much more efficient to use a sparse representation of the output. If you pass `sparse=True` to the `StringLookup` layer, the output of the layer will be a `tf.sparse.SparseTensor`, which can be efficiently handled as input to a `tf.keras.layers.Dense` layer.

## Embedding string data with a vocabulary

For larger vocabularies, an embedding is often needed for good performance. Here is an example embedding a string feature using feature columns:

In [ ]:

```
vocab_col = tf1.feature_column.categorical_column_with_vocabulary_list(  
    'col',  
    vocabulary_list=['small', 'medium', 'large'],  
    num_oov_buckets=0)  
embedding_col = tf1.feature_column.embedding_column(vocab_col, 4)  
call_feature_columns(embedding_col, {'col': ['small', 'medium', 'large']})
```

Using Keras preprocessing layers, this can be achieved by combining a `tf.keras.layers.StringLookup` layer and an `tf.keras.layers.Embedding` layer. The default output for the `StringLookup` will be integer indices which can be fed directly into an embedding.

Note: The `Embedding` layer contains trainable parameters. While the `StringLookup` layer can be applied to data inside or outside of a model, the `Embedding` must always be part of a trainable Keras model to train correctly.

In [ ]:

```
string_lookup_layer = tf.keras.layers.StringLookup(  
    vocabulary=['small', 'medium', 'large'], num_oov_indices=0)  
embedding = tf.keras.layers.Embedding(3, 4)  
embedding(string_lookup_layer(['small', 'medium', 'large']))
```

## Summing weighted categorical data

In some cases, you need to deal with categorical data where each occurrence of a category comes with an associated weight. In feature columns, this is handled with `tf.feature_column.weighted_categorical_column`. When paired with an `indicator_column`, this has the effect of summing weights per category.

In [ ]:

```
ids = tf.constant([[5, 11, 5, 17, 17]])  
weights = tf.constant([[0.5, 1.5, 0.7, 1.8, 0.2]])  
  
categorical_col = tf1.feature_column.categorical_column_with_identity(  
    'ids', num_buckets=20)  
weighted_categorical_col = tf1.feature_column.weighted_categorical_column(  
    categorical_col, 'weights')  
indicator_col = tf1.feature_column.indicator_column(weighted_categorical_col)  
call_feature_columns(indicator_col, {'ids': ids, 'weights': weights})
```

In Keras, this can be done by passing a `count_weights` input to `tf.keras.layers.CategoryEncoding` with `output_mode='count'` .

In [ ]:

```
ids = tf.constant([[5, 11, 5, 17, 17]])
weights = tf.constant([[0.5, 1.5, 0.7, 1.8, 0.2]])

# Using sparse output is more efficient when `num_tokens` is large.
count_layer = tf.keras.layers.CategoryEncoding(
    num_tokens=20, output_mode='count', sparse=True)
tf.sparse.to_dense(count_layer(ids, count_weights=weights))
```

## Embedding weighted categorical data

You might alternately want to embed weighted categorical inputs. In feature columns, the `embedding_column` contains a `combiner` argument. If any sample contains multiple entries for a category, they will be combined according to the argument setting (by default '`mean`' ).

In [ ]:

```
ids = tf.constant([[5, 11, 5, 17, 17]])
weights = tf.constant([[0.5, 1.5, 0.7, 1.8, 0.2]])

categorical_col = tf.feature_column.categorical_column_with_identity(
    'ids', num_buckets=20)
weighted_categorical_col = tf.feature_column.weighted_categorical_column(
    categorical_col, 'weights')
embedding_col = tf.feature_column.embedding_column(
    weighted_categorical_col, 4, combiner='mean')
call_feature_columns(embedding_col, {'ids': ids, 'weights': weights})
```

In Keras, there is no `combiner` option to `tf.keras.layers.Embedding`, but you can achieve the same effect with `tf.keras.layers.Dense`. The `embedding_column` above is simply linearly combining embedding vectors according to category weight. Though not obvious at first, it is exactly equivalent to representing your categorical inputs as a sparse weight vector of size (`num_tokens`), and multiplying them by a `Dense` kernel of shape (`embedding_size`, `num_tokens`).

In [ ]:

```
ids = tf.constant([[5, 11, 5, 17, 17]])
weights = tf.constant([[0.5, 1.5, 0.7, 1.8, 0.2]])

# For `combiner='mean'`, normalize your weights to sum to 1. Removing this line
# would be equivalent to an `embedding_column` with `combiner='sum'`.
weights = weights / tf.reduce_sum(weights, axis=-1, keepdims=True)

count_layer = tf.keras.layers.CategoryEncoding(
    num_tokens=20, output_mode='count', sparse=True)
embedding_layer = tf.keras.layers.Dense(4, use_bias=False)
embedding_layer(count_layer(ids, count_weights=weights))
```

## Complete training example

To show a complete training workflow, first prepare some data with three features of different types:

In [ ]:

```
features = {
    'type': [0, 1, 1],
    'size': ['small', 'small', 'medium'],
    'weight': [2.7, 1.8, 1.6],
}
labels = [1, 1, 0]
predict_features = {'type': [0], 'size': ['foo'], 'weight': [-0.7]}
```

Define some common constants for both TF1 and TF2 workflows:

In [ ]:

```
vocab = ['small', 'medium', 'large']
one_hot_dims = 3
embedding_dims = 4
weight_mean = 2.0
weight_variance = 1.0
```

## With feature columns

Feature columns must be passed as a list to the estimator on creation, and will be called implicitly during training.

In [ ]:

```
categorical_col = tf1.feature_column.categorical_column_with_identity(
    'type', num_buckets=one_hot_dims)
# Convert index to one-hot; e.g. [2] -> [0,0,1].
indicator_col = tf1.feature_column.indicator_column(categorical_col)

# Convert strings to indices; e.g. ['small'] -> [1].
vocab_col = tf1.feature_column.categorical_column_with_vocabulary_list(
    'size', vocabulary_list=vocab, num_oov_buckets=1)
# Embed the indices.
embedding_col = tf1.feature_column.embedding_column(vocab_col, embedding_dims)

normalizer_fn = lambda x: (x - weight_mean) / math.sqrt(weight_variance)
# Normalize the numeric inputs; e.g. [2.0] -> [0.0].
numeric_col = tf1.feature_column.numeric_column(
    'weight', normalizer_fn=normalizer_fn)

estimator = tf1.estimator.DNNClassifier(
    feature_columns=[indicator_col, embedding_col, numeric_col],
    hidden_units=[1])

def _input_fn():
    return tf1.data.Dataset.from_tensor_slices((features, labels)).batch(1)

estimator.train(_input_fn)
```

The feature columns will also be used to transform input data when running inference on the model.

In [ ]:

```
def _predict_fn():
    return tf1.data.Dataset.from_tensor_slices(predict_features).batch(1)

next(estimator.predict(_predict_fn))
```

## With Keras preprocessing layers

Keras preprocessing layers are more flexible in where they can be called. A layer can be applied directly to tensors, used inside a `tf.data` input pipeline, or built directly into a trainable Keras model.

In this example, you will apply preprocessing layers inside a `tf.data` input pipeline. To do this, you can define a separate `tf.keras.Model` to preprocess your input features. This model is not trainable, but is a convenient way to group preprocessing layers.

In [ ]:

```
inputs = {
    'type': tf.keras.Input(shape=(), dtype='int64'),
    'size': tf.keras.Input(shape=(), dtype='string'),
    'weight': tf.keras.Input(shape=(), dtype='float32'),
}
# Convert index to one-hot; e.g. [2] -> [0,0,1].
type_output = tf.keras.layers.CategoryEncoding(
    one_hot_dims, output_mode='one_hot')(inputs['type'])
# Convert size strings to indices; e.g. ['small'] -> [1].
size_output = tf.keras.layers.StringLookup(vocabulary=vocab)(inputs['size'])
# Normalize the numeric inputs; e.g. [2.0] -> [0.0].
weight_output = tf.keras.layers.Normalization(
    axis=None, mean=weight_mean, variance=weight_variance)(inputs['weight'])
outputs = {
    'type': type_output,
    'size': size_output,
    'weight': weight_output,
}
preprocessing_model = tf.keras.Model(inputs, outputs)
```

Note: As an alternative to supplying a vocabulary and normalization statistics on layer creation, many preprocessing layers provide an `adapt()` method for learning layer state directly from the input data. See the [preprocessing guide](#) ([https://www.tensorflow.org/guide/keras/preprocessing\\_layers#the\\_adapt\\_method](https://www.tensorflow.org/guide/keras/preprocessing_layers#the_adapt_method)) for more details.

You can now apply this model inside a call to `tf.data.Dataset.map`. Please note that the function passed to `map` will automatically be converted into a `tf.function`, and usual caveats for writing `tf.function` code apply (no side effects).

In [ ]:

```
# Apply the preprocessing in tf.data.Dataset.map.
dataset = tf.data.Dataset.from_tensor_slices((features, labels)).batch(1)
dataset = dataset.map(lambda x, y: (preprocessing_model(x), y),
                      num_parallel_calls=tf.data.AUTOTUNE)
# Display a preprocessed input sample.
next(dataset.take(1).as_numpy_iterator())
```

Next, you can define a separate `Model` containing the trainable layers. Note how the inputs to this model now reflect the preprocessed feature types and shapes.

In [ ]:

```
inputs = {
    'type': tf.keras.Input(shape=(one_hot_dims,), dtype='float32'),
    'size': tf.keras.Input(shape=(), dtype='int64'),
    'weight': tf.keras.Input(shape=(), dtype='float32'),
}
# Since the embedding is trainable, it needs to be part of the training model.
embedding = tf.keras.layers.Embedding(len(vocab), embedding_dims)
outputs = tf.keras.layers.concatenate([
    inputs['type'],
    embedding(inputs['size']),
    tf.expand_dims(inputs['weight'], -1),
])
outputs = tf.keras.layers.Dense(1)(outputs)
training_model = tf.keras.Model(inputs, outputs)
```

You can now train the `training_model` with `tf.keras.Model.fit`.

In [ ]:

```
# Train on the preprocessed data.
training_model.compile(
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=True))
training_model.fit(dataset)
```

Finally, at inference time, it can be useful to combine these separate stages into a single model that handles raw feature inputs.

In [ ]:

```
inputs = preprocessing_model.input
outputs = training_model(preprocessing_model(inputs))
inference_model = tf.keras.Model(inputs, outputs)

predict_dataset = tf.data.Dataset.from_tensor_slices(predict_features).batch(1)
inference_model.predict(predict_dataset)
```

This composed model can be saved as a [SavedModel](https://www.tensorflow.org/guide/saved_model) ([https://www.tensorflow.org/guide/saved\\_model](https://www.tensorflow.org/guide/saved_model)) for later use.

In [ ]:

```
inference_model.save('model')
restored_model = tf.keras.models.load_model('model')
restored_model.predict(predict_dataset)
```

Note: Preprocessing layers are not trainable, which allows you to apply them *asynchronously* using `tf.data`. This has performance benefits, as you can both [prefetch](https://www.tensorflow.org/guide/data_performance#prefetching) ([https://www.tensorflow.org/guide/data\\_performance#prefetching](https://www.tensorflow.org/guide/data_performance#prefetching)) preprocessed batches, and free up any accelerators to focus on the differentiable parts of a model. As this guide shows, separating preprocessing during training and composing it during inference is a flexible way to leverage these performance gains. However, if your model is small or preprocessing time is negligible, it may be simpler to build preprocessing into a complete model from the start. To do this you can build a single model starting with `tf.keras.Input`, followed by preprocessing layers, followed by trainable layers.

## Feature column equivalence table

For reference, here is an approximate correspondence between feature columns and preprocessing layers:

Feature Column	Keras Layer
`feature_column.bucketized_column`	`layers.Discretization`
`feature_column.categorical_column_with_hash_bucket`	`layers.Hashing`
`feature_column.categorical_column_with_identity`	`layers.CategoryEncoding`
`feature_column.categorical_column_with_vocabulary_file`	`layers.StringLookup` or `layers.IntegerLookup`
`feature_column.categorical_column_with_vocabulary_list`	`layers.StringLookup` or `layers.IntegerLookup`
`feature_column.crossed_column`	`layers.experimental.preprocessing.HashedCrossing`
`feature_column.embedding_column`	`layers.Embedding`
`feature_column.indicator_column`	`output_mode='one_hot'` or `output_mode='multi_hot'*
`feature_column.numeric_column`	`layers.Normalization`
`feature_column.sequence_categorical_column_with_hash_bucket`	`layers.Hashing`
`feature_column.sequence_categorical_column_with_identity`	`layers.CategoryEncoding`
`feature_column.sequence_categorical_column_with_vocabulary_file`	`layers.StringLookup`, `layers.IntegerLookup`, or `layer.TextVectorization`†
`feature_column.sequence_categorical_column_with_vocabulary_list`	`layers.StringLookup`, `layers.IntegerLookup`, or `layer.TextVectorization`†
`feature_column.sequence_numeric_column`	`layers.Normalization`
`feature_column.weighted_categorical_column`	`layers.CategoryEncoding`

\* `output_mode` can be passed to `layers.CategoryEncoding`, `layers.StringLookup`, `layers.IntegerLookup`, and `layers.TextVectorization`.

† `layers.TextVectorization` can handle freeform text input directly (e.g. entire sentences or paragraphs). This is not one-to-one replacement for categorical sequence handling in TF1, but may offer a convenient replacement for ad-hoc text preprocessing.

Note: Linear estimators, such as `tf.estimator.LinearClassifier`, can handle direct categorical input (integer indices) without an `embedding_column` or `indicator_column`. However, integer indices cannot be passed directly to `tf.keras.layers.Dense` or `tf.keras.experimental.LinearModel`. These inputs should be first encoded with `tf.layers.CategoryEncoding` with `output_mode='count'` (and `sparse=True` if the category sizes are large) before calling into `Dense` or `LinearModel`.

## Next Steps

- For more information on keras preprocessing layers, see [the guide to preprocessing layers](https://www.tensorflow.org/guide/keras/preprocessing_layers) ([https://www.tensorflow.org/guide/keras/preprocessing\\_layers](https://www.tensorflow.org/guide/keras/preprocessing_layers)).
- For a more in-depth example of applying preprocessing layers to structured data, see [the structured data tutorial](https://www.tensorflow.org/tutorials/structured_data/preprocessing_layers) ([https://www.tensorflow.org/tutorials/structured\\_data/preprocessing\\_layers](https://www.tensorflow.org/tutorials/structured_data/preprocessing_layers)).

Copyright 2021 The TensorFlow Authors.

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Migrate SessionRunHook to Keras callbacks



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/sessionrunhook_callback)

([https://www.tensorflow.org/guide/migrate/sessionrunhook\\_callback](https://www.tensorflow.org/guide/migrate/sessionrunhook_callback))



Run in

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/sessionrunhook\\_callback.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/sessionrunhook_callback.ipynb))

In TensorFlow 1, to customize the behavior of training, you use `tf.estimator.SessionRunHook` with `tf.estimator.Estimator`. This guide demonstrates how to migrate from `SessionRunHook` to TensorFlow 2's custom callbacks with the `tf.keras.callbacks.Callback` API, which works with Keras `Model.fit` for training (as well as `Model.evaluate` and `Model.predict`). You will learn how to do this by implementing a `SessionRunHook` and a `Callback` task that measures examples per second during training.

Examples of callbacks are checkpoint saving (`tf.keras.callbacks.ModelCheckpoint`) and [TensorBoard](#) (`tf.keras.callbacks.TensorBoard`) summary writing. Keras [callbacks](#) ([https://www.tensorflow.org/guide/keras/custom\\_callback](https://www.tensorflow.org/guide/keras/custom_callback)) are objects that are called at different points during training/evaluation/prediction in the built-in Keras `Model.fit` / `Model.evaluate` / `Model.predict` APIs. You can learn more about callbacks in the `tf.keras.callbacks.Callback` API docs, as well as the [Writing your own callbacks](#) ([https://www.tensorflow.org/guide/keras/custom\\_callback.ipynb](https://www.tensorflow.org/guide/keras/custom_callback.ipynb)) and [Training and evaluation with the built-in methods](#) ([https://www.tensorflow.org/guide/keras/train\\_and\\_evaluate](https://www.tensorflow.org/guide/keras/train_and_evaluate)) (the *Using callbacks* section) guides.

## Setup

Start with imports and a simple dataset for demonstration purposes:

In [ ]:

```
import tensorflow as tf
import tensorflow.compat.v1 as tf1

import time
from datetime import datetime
from absl import flags
```

In [ ]:

```
features = [[1., 1.5], [2., 2.5], [3., 3.5]]
labels = [[0.3], [0.5], [0.7]]
eval_features = [[4., 4.5], [5., 5.5], [6., 6.5]]
eval_labels = [[0.8], [0.9], [1.]]
```

## TensorFlow 1: Create a custom SessionRunHook with tf.estimator APIs

The following TensorFlow 1 examples show how to set up a custom `SessionRunHook` that measures examples per second during training. After creating the hook (`LoggerHook`), pass it to the `hooks` parameter of `tf.estimator.Estimator.train`.

In [ ]:

```
def _input_fn():
    return tf1.data.Dataset.from_tensor_slices(
        (features, labels)).batch(1).repeat(100)

def _model_fn(features, labels, mode):
    logits = tf1.layers.Dense(1)(features)
    loss = tf1.losses.mean_squared_error(labels=labels, predictions=logits)
    optimizer = tf1.train.AdagradOptimizer(0.05)
    train_op = optimizer.minimize(loss, global_step=tf1.train.get_global_step())
    return tf1.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)
```

In [ ]:

```
class LoggerHook(tf1.train.SessionRunHook):
    """Logs loss and runtime."""

    def begin(self):
        self._step = -1
        self._start_time = time.time()
        self.log_frequency = 10

    def before_run(self, run_context):
        self._step += 1

    def after_run(self, run_context, run_values):
        if self._step % self.log_frequency == 0:
            current_time = time.time()
            duration = current_time - self._start_time
            self._start_time = current_time
            examples_per_sec = self.log_frequency / duration
            print('Time:', datetime.now(), ', Step #:', self._step,
                  ', Examples per second:', examples_per_sec)

estimator = tf1.estimator.Estimator(model_fn=_model_fn)

# Begin training.
estimator.train(_input_fn, hooks=[LoggerHook()])
```

## TensorFlow 2: Create a custom Keras callback for Model.fit

In TensorFlow 2, when you use the built-in Keras `Model.fit` (or `Model.evaluate`) for training/evaluation, you can configure a custom `tf.keras.callbacks.Callback`, which you then pass to the `callbacks` parameter of `Model.fit` (or `Model.evaluate`). (Learn more in the [Writing your own callbacks \(../../guide/keras/custom\\_callback.ipynb\)](#) guide.)

In the example below, you will write a custom `tf.keras.callbacks.Callback` that logs various metrics—it will measure examples per second, which should be comparable to the metrics in the previous `SessionRunHook` example.

In [ ]:

```
class CustomCallback(tf.keras.callbacks.Callback):

    def on_train_begin(self, logs = None):
        self._step = -1
        self._start_time = time.time()
        self.log_frequency = 10

    def on_train_batch_begin(self, batch, logs = None):
        self._step += 1

    def on_train_batch_end(self, batch, logs = None):
        if self._step % self.log_frequency == 0:
            current_time = time.time()
            duration = current_time - self._start_time
            self._start_time = current_time
            examples_per_sec = self.log_frequency / duration
            print('Time:', datetime.now(), ', Step #:', self._step,
                  ', Examples per second:', examples_per_sec)

callback = CustomCallback()

dataset = tf.data.Dataset.from_tensor_slices(
    (features, labels)).batch(1).repeat(100)

model = tf.keras.models.Sequential([tf.keras.layers.Dense(1)])
optimizer = tf.keras.optimizers.Adagrad(learning_rate=0.05)

model.compile(optimizer, "mse")

# Begin training.
result = model.fit(dataset, callbacks=[callback], verbose = 0)
# Provide the results of training metrics.
result.history
```

## Next steps

Learn more about callbacks in:

- API docs: `tf.keras.callbacks.Callback`
- Guide: [Writing your own callbacks \(../guide/keras/custom\\_callback.ipynb/\)](#)
- Guide: [Training and evaluation with the built-in methods \(\[https://www.tensorflow.org/guide/keras/train\\\_and\\\_evaluate\]\(https://www.tensorflow.org/guide/keras/train\_and\_evaluate\)\)](#) (the *Using callbacks* section)

You may also find the following migration-related resources useful:

- The [Early stopping migration guide \(early\\_stopping.ipynb\)](#): `tf.keras.callbacks.EarlyStopping` is a built-in early stopping callback
- The [TensorBoard migration guide \(tensorboard.ipynb\)](#): TensorBoard enables tracking and displaying metrics
- The [LoggingTensorHook and StopAtStepHook to Keras callbacks migration guide \(logging\\_stop\\_hook.ipynb\)](#)

**Copyright 2021 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Migrate evaluation



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/evaluator)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/evaluator.ipynb)

Evaluation is a critical part of measuring and benchmarking models.

This guide demonstrates how to migrate evaluator tasks from TensorFlow 1 to TensorFlow 2. In Tensorflow 1 this functionality is implemented by `tf.estimator.train_and_evaluate`, when the API is running distributedly. In Tensorflow 2, you can use the built-in `tf.keras.utils.SidecarEvaluator`, or a custom evaluation loop on the evaluator task.

There are simple serial evaluation options in both TensorFlow 1 (`tf.estimator.Estimator.evaluate`) and TensorFlow 2 (`Model.fit(..., validation_data=...)` or `Model.evaluate`). The evaluator task is preferable when you would like your workers not switching between training and evaluation, and built-in evaluation in `Model.fit` is preferable when you would like your evaluation to be distributed.

## Setup

In [ ]:

```
import tensorflow.compat.v1 as tf1
import tensorflow as tf
import numpy as np
import tempfile
import time
import os
```

In [ ]:

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

## TensorFlow 1: Evaluating using `tf.estimator.train_and_evaluate`

In TensorFlow 1, you can configure a `tf.estimator` to evaluate the estimator using `tf.estimator.train_and_evaluate`.

In this example, start by defining the `tf.estimator.Estimator` and specifying training and evaluation specifications:

In [ ]:

```
feature_columns = [tf1.feature_column.numeric_column("x", shape=[28, 28])]

classifier = tf1.estimator.DNNClassifier(
    feature_columns=feature_columns,
    hidden_units=[256, 32],
    optimizer=tf1.train.AdamOptimizer(0.001),
    n_classes=10,
    dropout=0.2
)

train_input_fn = tf1.estimator.inputs.numpy_input_fn(
    x={"x": x_train},
    y=y_train.astype(np.int32),
    num_epochs=10,
    batch_size=50,
    shuffle=True,
)

test_input_fn = tf1.estimator.inputs.numpy_input_fn(
    x={"x": x_test},
    y=y_test.astype(np.int32),
    num_epochs=10,
    shuffle=False
)

train_spec = tf1.estimator.TrainSpec(input_fn=train_input_fn, max_steps=10)
eval_spec = tf1.estimator.EvalSpec(input_fn=test_input_fn,
                                    steps=10,
                                    throttle_secs=0)
```

Then, train and evaluate the model. The evaluation runs synchronously between training because it's limited as a local run in this notebook and alternates between training and evaluation. However, if the estimator is used distributedly, the evaluator will run as a dedicated evaluator task. For more information, check the [migration guide on distributed training](https://www.tensorflow.org/guide/migrate/multi_worker_cpu_gpu_training) ([https://www.tensorflow.org/guide/migrate/multi\\_worker\\_cpu\\_gpu\\_training](https://www.tensorflow.org/guide/migrate/multi_worker_cpu_gpu_training)).

In [ ]:

```
tf1.estimator.train_and_evaluate(estimator=classifier,
                                 train_spec=train_spec,
                                 eval_spec=eval_spec)
```

## TensorFlow 2: Evaluating a Keras model

In TensorFlow 2, if you use the Keras `Model.fit` API for training, you can evaluate the model with `tf.keras.utils.SidecarEvaluator`. You can also visualize the evaluation metrics in Tensorboard which is not shown in this guide.

To help demonstrate this, let's first start by defining and training the model:

In [ ]:

```
def create_model():
    return tf.keras.models.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(512, activation='relu'),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10)
    ])

loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

model = create_model()
model.compile(optimizer='adam',
              loss=loss,
              metrics=['accuracy'],
              steps_per_execution=10,
              run_eagerly=True)

log_dir = tempfile.mkdtemp()
model_checkpoint = tf.keras.callbacks.ModelCheckpoint(
    filepath=os.path.join(log_dir, 'ckpt-{epoch}'),
    save_weights_only=True)

model.fit(x=x_train,
          y=y_train,
          epochs=1,
          callbacks=[model_checkpoint])
```

Then, evaluate the model using `tf.keras.utils.SidecarEvaluator`. In real training, it's recommended to use a separate job to conduct the evaluation to free up worker resources for training.

In [ ]:

```
data = tf.data.Dataset.from_tensor_slices((x_test, y_test))
data = data.batch(64)

tf.keras.utils.SidecarEvaluator(
    model=model,
    data=data,
    checkpoint_dir=log_dir,
    max_evaluations=1
).start()
```

## Next steps

- To learn more about sidecar evaluation consider reading the `tf.keras.utils.SidecarEvaluator` API docs.
- To consider alternating training and evaluation in Keras consider reading about [other built-in methods](#) ([https://www.tensorflow.org/guide/keras/train\\_and\\_evaluate](https://www.tensorflow.org/guide/keras/train_and_evaluate)).

**Copyright 2021 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Migrate single-worker multiple-GPU training



[View on TensorFlow.org](https://www.tensorflow.org/)



[Run in Google Colab](#)

([https://www.tensorflow.org/guide/migrate/mirrored\\_strategy](https://www.tensorflow.org/guide/migrate/mirrored_strategy)) ([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/mirrored\\_strategy.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/mirrored_strategy.ipynb))

This guide demonstrates how to migrate the single-worker multiple-GPU workflows from TensorFlow 1 to TensorFlow 2.

To perform synchronous training across multiple GPUs on one machine:

- In TensorFlow 1, you use the `tf.estimator.Estimator` APIs with `tf.distribute.MirroredStrategy`.
- In TensorFlow 2, you can use [Keras Model.fit](https://www.tensorflow.org/tutorials/distribute/keras) (<https://www.tensorflow.org/tutorials/distribute/keras>) or a [custom training loop](https://www.tensorflow.org/tutorials/distribute/custom_training) ([https://www.tensorflow.org/tutorials/distribute/custom\\_training](https://www.tensorflow.org/tutorials/distribute/custom_training)) with `tf.distribute.MirroredStrategy`. Learn more in the [Distributed training with TensorFlow](https://www.tensorflow.org/guide/distributed_training#mirroredstrategy) ([https://www.tensorflow.org/guide/distributed\\_training#mirroredstrategy](https://www.tensorflow.org/guide/distributed_training#mirroredstrategy)) guide.

## Setup

Start with imports and a simple dataset for demonstration purposes:

In [ ]:

```
import tensorflow as tf
import tensorflow.compat.v1 as tf1
```

In [ ]:

```
features = [[1., 1.5], [2., 2.5], [3., 3.5]]
labels = [[0.3], [0.5], [0.7]]
eval_features = [[4., 4.5], [5., 5.5], [6., 6.5]]
eval_labels = [[0.8], [0.9], [1.]]
```

## TensorFlow 1: Single-worker distributed training with `tf.estimator.Estimator`

This example demonstrates the TensorFlow 1 canonical workflow of single-worker multiple-GPU training. You need to set the distribution strategy (`tf.distribute.MirroredStrategy`) through the `config` parameter of the `tf.estimator.Estimator`:

In [ ]:

```
def _input_fn():
    return tf1.data.Dataset.from_tensor_slices((features, labels)).batch(1)

def _eval_input_fn():
    return tf1.data.Dataset.from_tensor_slices(
        (eval_features, eval_labels)).batch(1)

def _model_fn(features, labels, mode):
    logits = tf1.layers.Dense(1)(features)
    loss = tf1.losses.mean_squared_error(labels=labels, predictions=logits)
    optimizer = tf1.train.AdagradOptimizer(0.05)
    train_op = optimizer.minimize(loss, global_step=tf1.train.get_global_step())
    return tf1.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)

strategy = tf1.distribute.MirroredStrategy()
config = tf1.estimator.RunConfig(
    train_distribute=strategy, eval_distribute=strategy)
estimator = tf1.estimator.Estimator(model_fn=_model_fn, config=config)

train_spec = tf1.estimator.TrainSpec(input_fn=_input_fn)
eval_spec = tf1.estimator.EvalSpec(input_fn=_eval_input_fn)
tf1.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

## TensorFlow 2: Single-worker training with Keras

When migrating to TensorFlow 2, you can use the Keras APIs with `tf.distribute.MirroredStrategy`.

If you use the `tf.keras` APIs for model building and Keras `Model.fit` for training, the main difference is instantiating the Keras model, an optimizer, and metrics in the context of `Strategy.scope`, instead of defining a `config` for `tf.estimator.Estimator`.

If you need to use a custom training loop, check out the [Using `tf.distribute.Strategy` with custom training loops](https://www.tensorflow.org/guide/distributed_training#using_tfdistributestrategy_with_custom_training_loops) ([https://www.tensorflow.org/guide/distributed\\_training#using\\_tfdistributestrategy\\_with\\_custom\\_training\\_loops](https://www.tensorflow.org/guide/distributed_training#using_tfdistributestrategy_with_custom_training_loops)) guide.

In [ ]:

```
dataset = tf.data.Dataset.from_tensor_slices((features, labels)).batch(1)
eval_dataset = tf.data.Dataset.from_tensor_slices(
    (eval_features, eval_labels)).batch(1)
```

In [ ]:

```
strategy = tf.distribute.MirroredStrategy()
with strategy.scope():
    model = tf.keras.models.Sequential([tf.keras.layers.Dense(1)])
    optimizer = tf.keras.optimizers.Adagrad(learning_rate=0.05)

model.compile(optimizer=optimizer, loss='mse')
model.fit(dataset)
model.evaluate(eval_dataset, return_dict=True)
```

## Next steps

To learn more about distributed training with `tf.distribute.MirroredStrategy` in TensorFlow 2, check out the following documentation:

- The [Distributed training on one machine with Keras \(../../tutorials/distribute/keras\)](#) tutorial
- The [Distributed training on one machine with a custom training loop \(../../tutorials/distribute/custom\\_training\)](#) tutorial
- The [Distributed training with TensorFlow \(../../guide/distributed\\_training\)](#) guide
- The [Using multiple GPUs \(../../guide/gpu#using\\_multiple\\_gpus\)](#) guide
- The [Optimize the performance on the multi-GPU single host \(with the TensorFlow Profiler\) \(../../guide/gpu\\_performance\\_analysis#2\\_optimize\\_the\\_performance\\_on\\_the\\_multi-gpu\\_single\\_host\)](#) guide

**Copyright 2021 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Migrate from Estimator to Keras APIs



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/migrating_estimator)

([https://www.tensorflow.org/guide/migrate/migrating\\_estimator](https://www.tensorflow.org/guide/migrate/migrating_estimator))



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/migrating_estimator.ipynb)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/migrating\\_estimator.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/migrating_estimator.ipynb))

This guide demonstrates how to migrate from TensorFlow 1's `tf.estimator.Estimator` APIs to TensorFlow 2's `tf.keras` APIs. First, you will set up and run a basic model for training and evaluation with `tf.estimator.Estimator`. Then, you will perform the equivalent steps in TensorFlow 2 with the `tf.keras` APIs. You will also learn how to customize the training step by subclassing `tf.keras.Model` and using `tf.GradientTape`.

- In TensorFlow 1, the high-level `tf.estimator.Estimator` APIs let you train and evaluate a model, as well as perform inference and save your model (for serving).
- In TensorFlow 2, use the Keras APIs to perform the aforementioned tasks, such as [model building](#) ([https://www.tensorflow.org/guide/keras/custom\\_layers\\_and\\_models](https://www.tensorflow.org/guide/keras/custom_layers_and_models)), gradient application, [training](#) ([https://www.tensorflow.org/guide/keras/customizing\\_what\\_happens\\_in\\_fit](https://www.tensorflow.org/guide/keras/customizing_what_happens_in_fit)), evaluation, and prediction.

(For migrating model/checkpoint saving workflows to TensorFlow 2, check out the [SavedModel \(saved\\_model.ipynb\)](#) and [Checkpoint \(checkpoint\\_saved.ipynb\)](#) migration guides.)

## Setup

Start with imports and a simple dataset:

In [ ]:

```
import tensorflow as tf
import tensorflow.compat.v1 as tf1
```

In [ ]:

```
features = [[1., 1.5], [2., 2.5], [3., 3.5]]
labels = [[0.3], [0.5], [0.7]]
eval_features = [[4., 4.5], [5., 5.5], [6., 6.5]]
eval_labels = [[0.8], [0.9], [1.]]
```

## TensorFlow 1: Train and evaluate with `tf.estimator.Estimator`

This example shows how to perform training and evaluation with `tf.estimator.Estimator` in TensorFlow 1.

Start by defining a few functions: an input function for the training data, an evaluation input function for the evaluation data, and a model function that tells the `Estimator` how the training op is defined with the features and labels:

In [ ]:

```
def _input_fn():
    return tf1.data.Dataset.from_tensor_slices((features, labels)).batch(1)

def _eval_input_fn():
    return tf1.data.Dataset.from_tensor_slices(
        (eval_features, eval_labels)).batch(1)

def _model_fn(features, labels, mode):
    logits = tf1.layers.Dense(1)(features)
    loss = tf1.losses.mean_squared_error(labels=labels, predictions=logits)
    optimizer = tf1.train.AdagradOptimizer(0.05)
    train_op = optimizer.minimize(loss, global_step=tf1.train.get_global_step())
    return tf1.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)
```

Instantiate your `Estimator`, and train the model:

In [ ]:

```
estimator = tf1.estimator.Estimator(model_fn=_model_fn)
estimator.train(_input_fn)
```

Evaluate the program with the evaluation set:

In [ ]:

```
estimator.evaluate(_eval_input_fn)
```

## TensorFlow 2: Train and evaluate with the built-in Keras methods

This example demonstrates how to perform training and evaluation with Keras `Model.fit` and `Model.evaluate` in TensorFlow 2. (You can learn more in the [Training and evaluation with the built-in methods \(\[https://www.tensorflow.org/guide/keras/train\\\_and\\\_evaluate\]\(https://www.tensorflow.org/guide/keras/train\_and\_evaluate\)\)](https://www.tensorflow.org/guide/keras/train_and_evaluate) guide.)

- Start by preparing the dataset pipeline with the `tf.data.Dataset` APIs.
- Define a simple Keras [Sequential \(\[https://www.tensorflow.org/guide/keras/sequential\\\_model\]\(https://www.tensorflow.org/guide/keras/sequential\_model\)\)](https://www.tensorflow.org/guide/keras/sequential_model) model with one linear (`tf.keras.layers.Dense`) layer.
- Instantiate an Adagrad optimizer (`tf.keras.optimizers.Adagrad`).
- Configure the model for training by passing the `optimizer` variable and the mean-squared error (`"mse"`) loss to `Model.compile`.

In [ ]:

```
dataset = tf.data.Dataset.from_tensor_slices((features, labels)).batch(1)
eval_dataset = tf.data.Dataset.from_tensor_slices(
    (eval_features, eval_labels)).batch(1)

model = tf.keras.models.Sequential([tf.keras.layers.Dense(1)])
optimizer = tf.keras.optimizers.Adagrad(learning_rate=0.05)

model.compile(optimizer=optimizer, loss="mse")
```

With that, you are ready to train the model by calling `Model.fit`:

In [ ]:

```
model.fit(dataset)
```

Finally, evaluate the model with `Model.evaluate`:

In [ ]:

```
model.evaluate(eval_dataset, return_dict=True)
```

## TensorFlow 2: Train and evaluate with a custom training step and built-in Keras methods

In TensorFlow 2, you can also write your own custom training step function with `tf.GradientTape` to perform forward and backward passes, while still taking advantage of the built-in training support, such as `tf.keras.callbacks.Callback` and `tf.distribute.Strategy`. (Learn more in [Customizing what happens in Model.fit](https://www.tensorflow.org/guide/keras/customizing_what_happens_in_fit) ([https://www.tensorflow.org/guide/keras/customizing\\_what\\_happens\\_in\\_fit](https://www.tensorflow.org/guide/keras/customizing_what_happens_in_fit)) and [Writing custom training loops from scratch](https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch) ([https://www.tensorflow.org/guide/keras/writing\\_a\\_training\\_loop\\_from\\_scratch](https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch)).

In this example, start by creating a custom `tf.keras.Model` by subclassing `tf.keras.Sequential` that overrides `Model.train_step`. (Learn more about [subclassing tf.keras.Model](https://www.tensorflow.org/guide/keras/custom_layers_and_models) ([https://www.tensorflow.org/guide/keras/custom\\_layers\\_and\\_models](https://www.tensorflow.org/guide/keras/custom_layers_and_models))). Inside that class, define a custom `train_step` function that for each batch of data performs a forward pass and backward pass during one training step.

In [ ]:

```
class CustomModel(tf.keras.Sequential):
    """A custom sequential model that overrides `Model.train_step`."""

    def train_step(self, data):
        batch_data, labels = data

        with tf.GradientTape() as tape:
            predictions = self(batch_data, training=True)
            # Compute the loss value (the loss function is configured
            # in `Model.compile`).
            loss = self.compiled_loss(labels, predictions)

            # Compute the gradients of the parameters with respect to the loss.
            gradients = tape.gradient(loss, self.trainable_variables)
            # Perform gradient descent by updating the weights/parameters.
            self.optimizer.apply_gradients(zip(gradients, self.trainable_variables))
            # Update the metrics (includes the metric that tracks the loss).
            self.compiled_metrics.update_state(labels, predictions)
            # Return a dict mapping metric names to the current values.
        return {m.name: m.result() for m in self.metrics}
```

Next, as before:

- Prepare the dataset pipeline with `tf.data.Dataset`.
- Define a simple model with one `tf.keras.layers.Dense` layer.
- Instantiate Adagrad (`tf.keras.optimizers.Adagrad`)
- Configure the model for training with `Model.compile`, while using mean-squared error ("mse") as the loss function.

In [ ]:

```
dataset = tf.data.Dataset.from_tensor_slices((features, labels)).batch(1)
eval_dataset = tf.data.Dataset.from_tensor_slices(
    (eval_features, eval_labels)).batch(1)

model = CustomModel([tf.keras.layers.Dense(1)])
optimizer = tf.keras.optimizers.Adagrad(learning_rate=0.05)

model.compile(optimizer=optimizer, loss="mse")
```

Call `Model.fit` to train the model:

In [ ]:

```
model.fit(dataset)
```

And, finally, evaluate the program with `Model.evaluate`:

In [ ]:

```
model.evaluate(eval_dataset, return_dict=True)
```

## Next steps

Additional Keras resources you may find useful:

- Guide: [Training and evaluation with the built-in methods](https://www.tensorflow.org/guide/keras/train_and_evaluate) ([https://www.tensorflow.org/guide/keras/train\\_and\\_evaluate](https://www.tensorflow.org/guide/keras/train_and_evaluate))
- Guide: [Customize what happens in Model.fit](https://www.tensorflow.org/guide/keras/customizing_what_happens_in_fit) ([https://www.tensorflow.org/guide/keras/customizing\\_what\\_happens\\_in\\_fit](https://www.tensorflow.org/guide/keras/customizing_what_happens_in_fit))
- Guide: [Writing a training loop from scratch](https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch) ([https://www.tensorflow.org/guide/keras/writing\\_a\\_training\\_loop\\_from\\_scratch](https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch))
- Guide: [Making new Keras layers and models via subclassing](https://www.tensorflow.org/guide/keras/custom_layers_and_models) ([https://www.tensorflow.org/guide/keras/custom\\_layers\\_and\\_models](https://www.tensorflow.org/guide/keras/custom_layers_and_models))

The following guides can assist with migrating distribution strategy workflows from `tf.estimator` APIs:

- [Migrate from TPUEstimator to TPUStrategy \(tpu\\_estimator.ipynb\)](#)
- [Migrate single-worker multiple-GPU training \(mirrored\\_strategy.ipynb\)](#)
- [Migrate multi-worker CPU/GPU training \(multi\\_worker\\_cpu\\_gpu\\_training.ipynb\)](#)

**Copyright 2021 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Use TF1.x models in TF2 workflows



[View on TensorFlow.org](https://www.tensorflow.org/guide/migrate/model_mapping)  
([https://www.tensorflow.org/guide/migrate/model\\_mapping](https://www.tensorflow.org/guide/migrate/model_mapping))



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/model_mapping.ipynb)  
([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/model\\_mapping.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/model_mapping.ipynb))

This guide provides an overview and examples of a [modeling code shim](https://en.wikipedia.org/wiki/Shim_(computing)) ([https://en.wikipedia.org/wiki/Shim\\_\(computing\)](https://en.wikipedia.org/wiki/Shim_(computing))) that you can employ to use your existing TF1.x models in TF2 workflows such as eager execution, `tf.function`, and distribution strategies with minimal changes to your modeling code.

## Scope of usage

The shim described in this guide is designed for TF1.x models that rely on:

1. `tf.compat.v1.get_variable` and `tf.compat.v1.variable_scope` to control variable creation and reuse, and
2. Graph-collection based APIs such as `tf.compat.v1.global_variables()`, `tf.compat.v1.trainable_variables`, `tf.compat.v1.losses.get_regularization_losses()`, and `tf.compat.v1.get_collection()` to keep track of weights and regularization losses

This includes most models built on top of `tf.compat.v1.layer`, `tf.contrib.layers` APIs, and [TensorFlow-Slim](https://github.com/google-research/tf-slim) (<https://github.com/google-research/tf-slim>).

The shim is **NOT** necessary for the following TF1.x models:

1. Stand-alone Keras models that already track all of their trainable weights and regularization losses via `model.trainable_weights` and `model.losses` respectively.
2. `tf.Module`s that already track all of their trainable weights via `module.trainable_variables`, and only create weights if they have not already been created.

These models are likely to work in TF2 with eager execution and `tf.function`s out-of-the-box.

## Setup

Import TensorFlow and other dependencies.

```
In [ ]:
```

```
!pip uninstall -y -q tensorflow
```

```
In [ ]:
```

```
# Install tf-nightly as the DeterministicRandomTestTool is available only in
```

```
# Tensorflow 2.8
```

```
!pip install -q tf-nightly
```

```
In [ ]:
```

```
import tensorflow as tf
import tensorflow.compat.v1 as v1
import sys
import numpy as np

from contextlib import contextmanager
```

## The `track_tf1_style_variables` decorator

The key shim described in this guide is `tf.compat.v1.keras.utils.track_tf1_style_variables`, a decorator that you can use within methods belonging to `tf.keras.layers.Layer` and `tf.Module` to track TF1.x-style weights and capture regularization losses.

Decorating a `tf.keras.layers.Layer`'s or `tf.Module`'s call methods with `tf.compat.v1.keras.utils.track_tf1_style_variables` allows variable creation and reuse via `tf.compat.v1.get_variable` (and by extension `tf.compat.v1.layers`) to work correctly inside of the decorated method rather than always creating a new variable on each call. It will also cause the layer or module to implicitly track any weights created or accessed via `get_variable` inside the decorated method.

In addition to tracking the weights themselves under the standard `layer.variable` / `module.variable` /etc. properties, if the method belongs to a `tf.keras.layers.Layer`, then any regularization losses specified via the `get_variable` or `tf.compat.v1.layers` regularizer arguments will get tracked by the layer under the standard `layer.losses` property.

This tracking mechanism enables using large classes of TF1.x-style model-forward-pass code inside of Keras layers or `tf.Module`s in TF2 even with TF2 behaviors enabled.

## Usage examples

The usage examples below demonstrate the modeling shims used to decorate `tf.keras.layers.Layer` methods, but except where they are specifically interacting with Keras features they are applicable when decorating `tf.Module` methods as well.

### Layer built with `tf.compat.v1.get_variable`

Imagine you have a layer implemented directly on top of `tf.compat.v1.get_variable` as follows:

```
def dense(self, inputs, units):
    out = inputs
    with tf.compat.v1.variable_scope("dense"):
        # The weights are created with a `regularizer`,
        kernel = tf.compat.v1.get_variable(
            shape=[out.shape[-1], units],
            regularizer=tf.keras.regularizers.L2(),
            initializer=tf.compat.v1.initializers.glorot_normal,
            name="kernel")
        bias = tf.compat.v1.get_variable(
            shape=[units,],
            initializer=tf.compat.v1.initializers.zeros,
            name="bias")
        out = tf.linalg.matmul(out, kernel)
        out = tf.compat.v1.nn.bias_add(out, bias)
    return out
```

Use the shim to turn it into a layer and call it on inputs.

In [ ]:

```
class DenseLayer(tf.keras.layers.Layer):  
  
    def __init__(self, units, *args, **kwargs):  
        super().__init__(*args, **kwargs)  
        self.units = units  
  
    @tf.compat.v1.keras.utils.track_tf1_style_variables  
    def call(self, inputs):  
        out = inputs  
        with tf.compat.v1.variable_scope("dense"):  
            # The weights are created with a `regularizer`,  
            # so the layer should track their regularization losses  
            kernel = tf.compat.v1.get_variable(  
                shape=[out.shape[-1], self.units],  
                regularizer=tf.keras.regularizers.L2(),  
                initializer=tf.compat.v1.initializers.glorot_normal,  
                name="kernel")  
            bias = tf.compat.v1.get_variable(  
                shape=[self.units,],  
                initializer=tf.compat.v1.initializers.zeros,  
                name="bias")  
            out = tf.linalg.matmul(out, kernel)  
            out = tf.compat.v1.nn.bias_add(out, bias)  
        return out  
  
layer = DenseLayer(10)  
x = tf.random.normal(shape=(8, 20))  
layer(x)
```

Access the tracked variables and the captured regularization losses like a standard Keras layer.

In [ ]:

```
layer.trainable_variables  
layer.losses
```

To see that the weights get reused each time you call the layer, set all the weights to zero and call the layer again.

In [ ]:

```
print("Resetting variables to zero:", [var.name for var in layer.trainable_variables])  
  
for var in layer.trainable_variables:  
    var.assign(var * 0.0)  
  
# Note: layer.losses is not a live view and  
# will get reset only at each layer call  
print("layer.losses:", layer.losses)  
print("calling layer again.")  
out = layer(x)  
print("layer.losses: ", layer.losses)  
out
```

You can use the converted layer directly in Keras functional model construction as well.

In [ ]:

```
inputs = tf.keras.Input(shape=(20))  
outputs = DenseLayer(10)(inputs)  
model = tf.keras.Model(inputs=inputs, outputs=outputs)  
  
x = tf.random.normal(shape=(8, 20))  
model(x)  
  
# Access the model variables and regularization losses  
model.weights  
model.losses
```

## Model built with `tf.compat.v1.layers`

Imagine you have a layer or model implemented directly on top of `tf.compat.v1.layers` as follows:

```
def model(self, inputs, units):
    with tf.compat.v1.variable_scope('model'):
        out = tf.compat.v1.layers.conv2d(
            inputs, 3, 3,
            kernel_regularizer="l2")
        out = tf.compat.v1.layers.flatten(out)
        out = tf.compat.v1.layers.dense(
            out, units,
            kernel_regularizer="l2")
    return out
```

Use the shim to turn it into a layer and call it on inputs.

In [ ]:

```
class CompatV1LayerModel(tf.keras.layers.Layer):

    def __init__(self, units, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.units = units

    @tf.compat.v1.keras.utils.track_tf1_style_variables
    def call(self, inputs):
        with tf.compat.v1.variable_scope('model'):
            out = tf.compat.v1.layers.conv2d(
                inputs, 3, 3,
                kernel_regularizer="l2")
            out = tf.compat.v1.layers.flatten(out)
            out = tf.compat.v1.layers.dense(
                out, self.units,
                kernel_regularizer="l2")
        return out

layer = CompatV1LayerModel(10)
x = tf.random.normal(shape=(8, 5, 5, 5))
layer(x)
```

Warning: For safety reasons, make sure to put all `tf.compat.v1.layers` inside of a non-empty-string `variable_scope`. This is because `tf.compat.v1.layers` with auto-generated names will always auto-increment the name outside of any variable scope. This means the requested variable names will mismatch each time you call the layer/module. So, rather than reusing the already-made weights it will create a new set of variables every call.

Access the tracked variables and captured regularization losses like a standard Keras layer.

In [ ]:

```
layer.trainable_variables
layer.losses
```

To see that the weights get reused each time you call the layer, set all the weights to zero and call the layer again.

In [ ]:

```
print("Resetting variables to zero:", [var.name for var in layer.trainable_variables])

for var in layer.trainable_variables:
    var.assign(var * 0.0)

out = layer(x)
print("layer.losses: ", layer.losses)
out
```

You can use the converted layer directly in Keras functional model construction as well.

In [ ]:

```
inputs = tf.keras.Input(shape=(5, 5, 5))
outputs = CompatV1LayerModel(10)(inputs)
model = tf.keras.Model(inputs=inputs, outputs=outputs)

x = tf.random.normal(shape=(8, 5, 5, 5))
model(x)
```

In [ ]:

```
# Access the model variables and regularization losses
model.weights
model.losses
```

## Capture batch normalization updates and model training args

In TF1.x, you perform batch normalization like this:

```
x_norm = tf.compat.v1.layers.batch_normalization(x, training=training)

# ...

update_ops = tf.compat.v1.get_collection(tf.GraphKeys.UPDATE_OPS)
train_op = optimizer.minimize(loss)
train_op = tf.group([train_op, update_ops])
```

Note that:

1. The batch normalization moving average updates are tracked by `get_collection` which was called separately from the layer
2. `tf.compat.v1.layers.batch_normalization` requires a `training` argument (generally called `is_training` when using TF-Slim batch normalization layers)

In TF2, due to [eager execution \(<https://www.tensorflow.org/guide/eager>\)](https://www.tensorflow.org/guide/eager) and automatic control dependencies, the batch normalization moving average updates will be executed right away. There is no need to separately collect them from the updates collection and add them as explicit control dependencies.

Additionally, if you give your `tf.keras.layers.Layer`'s forward pass method a `training` argument, Keras will be able to pass the current training phase and any nested layers to it just like it does for any other layer. See the API docs for `tf.keras.Model` for more information on how Keras handles the `training` argument.

If you are decorating `tf.Module` methods, you need to make sure to manually pass all `training` arguments as needed. However, the batch normalization moving average updates will still be applied automatically with no need for explicit control dependencies.

The following code snippets demonstrate how to embed batch normalization layers in the shim and how using it in a Keras model works (applicable to `tf.keras.layers.Layer`).

In [ ]:

```
class CompatV1BatchNorm(tf.keras.layers.Layer):

    @tf.compat.v1.keras.utils.track_tf1_style_variables
    def call(self, inputs, training=None):
        print("Forward pass called with `training` =", training)
        with v1.variable_scope('batch_norm_layer'):
            return v1.layers.batch_normalization(x, training=training)
```

In [ ]:

```
print("Constructing model")
inputs = tf.keras.Input(shape=(5, 5, 5))
outputs = CompatV1BatchNorm()(inputs)
model = tf.keras.Model(inputs=inputs, outputs=outputs)

print("Calling model in inference mode")
x = tf.random.normal(shape=(8, 5, 5, 5))
model(x, training=False)

print("Moving average variables before training: ",
      {var.name: var.read_value() for var in model.non_trainable_variables})

# Notice that when running TF2 and eager execution, the batchnorm layer directly
# updates the moving averages while training without needing any extra control
# dependencies
print("calling model in training mode")
model(x, training=True)

print("Moving average variables after training: ",
      {var.name: var.read_value() for var in model.non_trainable_variables})
```

## Variable-scope based variable reuse

Any variable creations in the forward pass based on `get_variable` will maintain the same variable naming and reuse semantics that variable scopes have in TF1.x. This is true as long as you have at least one non-empty outer scope for any `tf.compat.v1.layers` with auto-generated names, as mentioned above.

Note: Naming and reuse will be scoped to within a single layer/module instance. Calls to `get_variable` inside one shim-decorated layer or module will not be able to refer to variables created inside of layers or modules. You can get around this by using Python references to other variables directly if need be, rather than accessing variables via `get_variable`.

## Eager execution & `tf.function`

As seen above, decorated methods for `tf.keras.layers.Layer` and `tf.Module` run inside of eager execution and are also compatible with `tf.function`. This means you can use [pdb](https://docs.python.org/3/library/pdb.html) (<https://docs.python.org/3/library/pdb.html>) and other interactive tools to step through your forward pass as it is running.

Warning: Although it is perfectly safe to call your shim-decorated layer/module methods from *inside* of a `tf.function`, it is not safe to put `tf.function`s inside of your shim-decorated methods if those `tf.functions` contain `get_variable` calls. Entering a `tf.function` resets `variable_scope`s, which means the TF1.x-style variable-scope-based variable reuse that the shim mimics will break down in this setting.

## Distribution strategies

Calls to `get_variable` inside of `@track_tf1_style_variables`-decorated layer or module methods use standard `tf.Variable` variable creations under the hood. This means you can use them with the various distribution strategies available with `tf.distribute` such as `MirroredStrategy` and `TPUStrategy`.

## Nesting `tf.Variables`, `tf.Modules`, `tf.keras.layers` & `tf.keras.models` in decorated calls

Decorating your layer call in `tf.compat.v1.keras.utils.track_tf1_style_variables` will only add automatic implicit tracking of variables created (and reused) via `tf.compat.v1.get_variable`. It will not capture weights directly created by `tf.Variable` calls, such as those used by typical Keras layers and most `tf.Module`s. This section describes how to handle these nested cases.

### (Pre-existing usages) `tf.keras.layers` and `tf.keras.models`

For pre-existing usages of nested Keras layers and models, use `tf.compat.v1.keras.utils.get_or_create_layer`. This is only recommended for easing migration of existing TF1.x nested Keras usages; new code should use explicit attribute setting as described below for `tf.Variables` and `tf.Modules`.

To use `tf.compat.v1.keras.utils.get_or_create_layer`, wrap the code that constructs your nested model into a method, and pass it in to the method. Example:

In [ ]:

```
class NestedModel(tf.keras.Model):
    def __init__(self, units, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.units = units

    def build_model(self):
        inp = tf.keras.Input(shape=(5, 5))
        dense_layer = tf.keras.layers.Dense(
            10, name="dense", kernel_regularizer="l2",
            kernel_initializer=tf.compat.v1.ones_initializer())
        model = tf.keras.Model(inputs=inp, outputs=dense_layer(inp))
        return model

    @tf.compat.v1.keras.utils.track_tf1_style_variables
    def call(self, inputs):
        # Get or create a nested model without assigning it as an explicit property
        model = tf.compat.v1.keras.utils.get_or_create_layer(
            "dense_model", self.build_model)
        return model(inputs)

layer = NestedModel(10)
layer(tf.ones(shape=(5,5)))
```

This method ensures that these nested layers are correctly reused and tracked by tensorflow. Note that the `@track_tf1_style_variables` decorator is still required on the appropriate method. The model builder method passed into `get_or_create_layer` (in this case, `self.build_model`), should take no arguments.

Weights are tracked:

In [ ]:

```
assert len(layer.weights) == 2
weights = {x.name: x for x in layer.variables}

assert set(weights.keys()) == {"dense/bias:0", "dense/kernel:0"}

layer.weights
```

And regularization loss as well:

In [ ]:

```
tf.add_n(layer.losses)
```

## Incremental migration: `tf.Variables` and `tf.Modules`

If you need to embed `tf.Variable` calls or `tf.Module`s in your decorated methods (for example, if you are following the incremental migration to non-legacy TF2 APIs described later in this guide), you still need to explicitly track these, with the following requirements:

- Explicitly make sure that the variable/module/layer is only created once
- Explicitly attach them as instance attributes just as you would when defining a [typical module or layer](https://www.tensorflow.org/guide/intro_to_modules#define_models_and_layers_in_tensorflow) ([https://www.tensorflow.org/guide/intro\\_to\\_modules#define\\_models\\_and\\_layers\\_in\\_tensorflow](https://www.tensorflow.org/guide/intro_to_modules#define_models_and_layers_in_tensorflow))
- Explicitly reuse the already-created object in follow-on calls

This ensures that weights are not created new each call and are correctly reused. Additionally, this also ensures that existing weights and regularization losses get tracked.

Here is an example of how this could look:

In [ ]:

```
class NestedLayer(tf.keras.layers.Layer):
    def __init__(self, units, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.units = units

    @tf.compat.v1.keras.utils.track_tf1_style_variables
    def __call__(self, inputs):
        out = inputs
        with tf.compat.v1.variable_scope("inner_dense"):
            # The weights are created with a `regularizer`,
            # so the layer should track their regularization losses
            kernel = tf.compat.v1.get_variable(
                shape=[out.shape[-1], self.units],
                regularizer=tf.keras.regularizers.L2(),
                initializer=tf.compat.v1.initializers.glorot_normal,
                name="kernel")
            bias = tf.compat.v1.get_variable(
                shape=[self.units,],
                initializer=tf.compat.v1.initializers.zeros,
                name="bias")
            out = tf.linalg.matmul(out, kernel)
            out = tf.compat.v1.nn.bias_add(out, bias)
        return out

class WrappedDenseLayer(tf.keras.layers.Layer):
    def __init__(self, units, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        # Only create the nested tf.variable/module/layer/model
        # once, and then reuse it each time!
        self._dense_layer = NestedLayer(self.units)

    @tf.compat.v1.keras.utils.track_tf1_style_variables
    def call(self, inputs):
        with tf.compat.v1.variable_scope('outer'):
            outputs = tf.compat.v1.layers.dense(inputs, 3)
            outputs = tf.compat.v1.layers.dense(outputs, 4)
        return self._dense_layer(outputs)

layer = WrappedDenseLayer(10)
layer(tf.ones(shape=(5, 5)))
```

Note that explicit tracking of the nested module is needed even though it is decorated with the `track_tf1_style_variables` decorator. This is because each module/layer with decorated methods has its own variable store associated with it.

The weights are correctly tracked:

In [ ]:

```
assert len(layer.weights) == 6
weights = {x.name: x for x in layer.variables}

assert set(weights.keys()) == {"outer/inner_dense/bias:0",
                               "outer/inner_dense/kernel:0",
                               "outer/dense/bias:0",
                               "outer/dense/kernel:0",
                               "outer/dense_1/bias:0",
                               "outer/dense_1/kernel:0"}

layer.trainable_weights
```

As well as regularization loss:

In [ ]:

```
layer.losses
```

Note that if the `NestedLayer` were a non-Keras `tf.Module` instead, variables would still be tracked but regularization losses would not be automatically tracked, so you would have to explicitly track them separately.

## Guidance on variable names

Explicit `tf.Variable` calls and Keras layers use a different layer name / variable name autogeneration mechanism than you may be used to from the combination of `get_variable` and `variable_scopes`. Although the shim will make your variable names match for variables created by `get_variable` even when going from TF1.x graphs to TF2 eager execution & `tf.function`, it cannot guarantee the same for the variable names generated for `tf.Variable` calls and Keras layers that you embed within your method decorators. It is even possible for multiple variables to share the same name in TF2 eager execution and `tf.function`.

You should take special care with this when following the sections on validating correctness and mapping TF1.x checkpoints later on in this guide.

## Using `tf.compat.v1.make_template` in the decorated method

It is highly recommended you directly use `tf.compat.v1.keras.utils.track_tf1_style_variables` instead of using `tf.compat.v1.make_template`, as it is a thinner layer on top of TF2.

Follow the guidance in this section for prior TF1.x code that was already relying on `tf.compat.v1.make_template`.

Because `tf.compat.v1.make_template` wraps code that uses `get_variable`, the `track_tf1_style_variables` decorator allows you to use these templates in layer calls and successfully track the weights and regularization losses.

However, do make sure to call `make_template` only once and then reuse the same template in each layer call. Otherwise, a new template will be created each time you call the layer along with a new set of variables.

For example,

In [ ]:

```
class CompatV1TemplateScaleByY(tf.keras.layers.Layer):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        def my_op(x, scalar_name):
            var1 = tf.compat.v1.get_variable(scalar_name,
                                            shape=[],
                                            regularizer=tf.compat.v1.keras.regularizers.L2(),
                                            initializer=tf.compat.v1.constant_initializer(1.5))
            return x * var1
        self.scale_by_y = tf.compat.v1.make_template('scale_by_y', my_op, scalar_name='y')

    @tf.compat.v1.keras.utils.track_tf1_style_variables
    def call(self, inputs):
        with tf.compat.v1.variable_scope('layer'):
            # Using a scope ensures the `scale_by_y` name will not be incremented
            # for each instantiation of the layer.
            return self.scale_by_y(inputs)

layer = CompatV1TemplateScaleByY()

out = layer(tf.ones(shape=(2, 3)))
print("weights:", layer.weights)
print("regularization loss:", layer.losses)
print("output:", out)
```

Warning: Avoid sharing the same `make_template`-created template across multiple layer instances as it may break the variable and regularization loss tracking mechanisms of the shim decorator. Additionally, if you plan to use the same `make_template` name inside of multiple layer instances then you should nest the created template's usage inside of a `variable_scope`. If not, the generated name for the template's `variable_scope` will increment with each new instance of the layer. This could alter the weight names in unexpected ways.

## Incremental migration to Native TF2

As mentioned earlier, `track_tf1_style_variables` allows you to mix TF2-style object-oriented `tf.Variable` / `tf.keras.layers.Layer` / `tf.Module` usage with legacy `tf.compat.v1.get_variable` / `tf.compat.v1.layers`-style usage inside of the same decorated module/layer.

This means that after you have made your TF1.x model fully-TF2-compatible, you can write all new model components with native (non- `tf.compat.v1`) TF2 APIs and have them interoperate with your older code.

However, if you continue to modify your older model components, you may also choose to incrementally switch your legacy-style `tf.compat.v1` usage over to the purely-native object-oriented APIs that are recommended for newly written TF2 code.

`tf.compat.v1.get_variable` usage can be replaced with either `self.add_weight` calls if you are decorating a Keras layer/model, or with `tf.Variable` calls if you are decorating Keras objects or `tf.Module`s.

Both functional-style and object-oriented `tf.compat.v1.layers` can generally be replaced with the equivalent `tf.keras.layers` layer with no argument changes required.

You may also consider chunks parts of your model or common patterns into individual layers/modules during your incremental move to purely-native APIs, which may themselves use `track_tf1_style_variables`.

### A note on Slim and contrib.layers

A large amount of older TF 1.x code uses the [Slim \(<https://ai.googleblog.com/2016/08/tf-slim-high-level-library-to-define.html>\)](https://ai.googleblog.com/2016/08/tf-slim-high-level-library-to-define.html) library, which was packaged with TF 1.x as `tf.contrib.layers`. Converting code using Slim to native TF 2 is more involved than converting `v1.layers`. In fact, it may make sense to convert your Slim code to `v1.layers` first, then convert to Keras. Below is some general guidance for converting Slim code.

- Ensure all arguments are explicit. Remove `arg_scopes` if possible. If you still need to use them, split `normalizer_fn` and `activation_fn` into their own layers.
- Separable conv layers map to one or more different Keras layers (depthwise, pointwise, and separable Keras layers).
- Slim and `v1.layers` have different argument names and default values.
- Note that some arguments have different scales.

## Migration to Native TF2 ignoring checkpoint compatibility

The following code sample demonstrates an incremental move of a model to purely-native APIs without considering checkpoint compatibility.

In [ ]:

```
class CompatModel(tf.keras.layers.Layer):  
    def __init__(self, units, *args, **kwargs):  
        super().__init__(*args, **kwargs)  
        self.units = units  
  
    @tf.compat.v1.keras.utils.track_tf1_style_variables  
    def call(self, inputs, training=None):  
        with tf.compat.v1.variable_scope('model'):  
            out = tf.compat.v1.layers.conv2d(  
                inputs, 3, 3,  
                kernel_regularizer="l2")  
            out = tf.compat.v1.layers.flatten(out)  
            out = tf.compat.v1.layers.dropout(out, training=training)  
            out = tf.compat.v1.layers.dense(  
                out, self.units,  
                kernel_regularizer="l2")  
        return out
```

Next, replace the `compat.v1` APIs with their native object-oriented equivalents in a piecewise manner. Start by switching the convolution layer to a Keras object created in the layer constructor.

In [ ]:

```
class PartiallyMigratedModel(tf.keras.layers.Layer):

    def __init__(self, units, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.units = units
        self.conv_layer = tf.keras.layers.Conv2D(
            3, 3,
            kernel_regularizer="l2")

    @tf.compat.v1.keras.utils.track_tf1_style_variables
    def call(self, inputs, training=None):
        with tf.compat.v1.variable_scope('model'):
            out = self.conv_layer(inputs)
            out = tf.compat.v1.layers.flatten(out)
            out = tf.compat.v1.layers.dropout(out, training=training)
            out = tf.compat.v1.layers.dense(
                out, self.units,
                kernel_regularizer="l2")
        return out
```

Use the [v1.keras.utils.DeterministicRandomTestTool](#)

([https://www.tensorflow.org/api\\_docs/python/tf/compat/v1/keras/utils/DeterministicRandomTestTool](https://www.tensorflow.org/api_docs/python/tf/compat/v1/keras/utils/DeterministicRandomTestTool)) class to verify that this incremental change leaves the model with the same behavior as before.

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    layer = CompatModel(10)

    inputs = tf.random.normal(shape=(10, 5, 5, 5))
    original_output = layer(inputs)

    # Grab the regularization loss as well
    original_regularization_loss = tf.math.add_n(layer.losses)

print(original_regularization_loss)
```

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')
with random_tool.scope():
    layer = PartiallyMigratedModel(10)

    inputs = tf.random.normal(shape=(10, 5, 5, 5))
    migrated_output = layer(inputs)

    # Grab the regularization loss as well
    migrated_regularization_loss = tf.math.add_n(layer.losses)

print(migrated_regularization_loss)
```

In [ ]:

```
# Verify that the regularization loss and output both match
np.testing.assert_allclose(original_regularization_loss.numpy(), migrated_regularization_loss.numpy())
np.testing.assert_allclose(original_output.numpy(), migrated_output.numpy())
```

You have now replaced all of the individual `compat.v1.layers` with native Keras layers.

In [ ]:

```
class NearlyFullyNativeModel(tf.keras.layers.Layer):  
  
    def __init__(self, units, *args, **kwargs):  
        super().__init__(*args, **kwargs)  
        self.units = units  
        self.conv_layer = tf.keras.layers.Conv2D(  
            3, 3,  
            kernel_regularizer="l2")  
        self.flatten_layer = tf.keras.layers.Flatten()  
        self.dense_layer = tf.keras.layers.Dense(  
            self.units,  
            kernel_regularizer="l2")  
  
    @tf.compat.v1.keras.utils.track_tf1_style_variables  
    def call(self, inputs):  
        with tf.compat.v1.variable_scope('model'):  
            out = self.conv_layer(inputs)  
            out = self.flatten_layer(out)  
            out = self.dense_layer(out)  
        return out
```

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')  
with random_tool.scope():  
    layer = NearlyFullyNativeModel(10)  
  
    inputs = tf.random.normal(shape=(10, 5, 5, 5))  
    migrated_output = layer(inputs)  
  
    # Grab the regularization loss as well  
    migrated_regularization_loss = tf.math.add_n(layer.losses)  
  
print(migrated_regularization_loss)
```

In [ ]:

```
# Verify that the regularization loss and output both match  
np.testing.assert_allclose(original_regularization_loss.numpy(), migrated_regularization_loss.numpy())  
np.testing.assert_allclose(original_output.numpy(), migrated_output.numpy())
```

Finally, remove both any remaining (no-longer-needed) `variable_scope` usage and the `track_tf1_style_variables` decorator itself.

You are now left with a version of the model that uses entirely native APIs.

In [ ]:

```
class FullyNativeModel(tf.keras.layers.Layer):  
  
    def __init__(self, units, *args, **kwargs):  
        super().__init__(*args, **kwargs)  
        self.units = units  
        self.conv_layer = tf.keras.layers.Conv2D(  
            3, 3,  
            kernel_regularizer="l2")  
        self.flatten_layer = tf.keras.layers.Flatten()  
        self.dense_layer = tf.keras.layers.Dense(  
            self.units,  
            kernel_regularizer="l2")  
  
    def call(self, inputs):  
        out = self.conv_layer(inputs)  
        out = self.flatten_layer(out)  
        out = self.dense_layer(out)  
    return out
```

In [ ]:

```
random_tool = v1.keras.utils.DeterministicRandomTestTool(mode='num_random_ops')  
with random_tool.scope():  
    layer = FullyNativeModel(10)  
  
    inputs = tf.random.normal(shape=(10, 5, 5, 5))  
    migrated_output = layer(inputs)  
  
    # Grab the regularization loss as well  
    migrated_regularization_loss = tf.math.add_n(layer.losses)  
  
print(migrated_regularization_loss)
```

In [ ]:

```
# Verify that the regularization loss and output both match
np.testing.assert_allclose(original_regularization_loss.numpy(), migrated_regularization_loss.numpy())
np.testing.assert_allclose(original_output.numpy(), migrated_output.numpy())
```

## Maintaining checkpoint compatibility during migration to Native TF2

The above migration process to native TF2 APIs changed both the variable names (as Keras APIs produce very different weight names), and the object-oriented paths that point to different weights in the model. The impact of these changes is that they will have broken both any existing TF1-style name-based checkpoints or TF2-style object-oriented checkpoints.

However, in some cases, you might be able to take your original name-based checkpoint and find a mapping of the variables to their new names with approaches like the one detailed in the [Reusing TF1.x checkpoints guide \(./reusing\\_checkpoints.ipynb\)](#).

Some tips to making this feasible are as follows:

- Variables still all have a `name` argument you can set.
- Keras models also take a `name` argument as which they set as the prefix for their variables.
- The `v1.name_scope` function can be used to set variable name prefixes. This is very different from `tf.variable_scope`. It only affects names, and doesn't track variables and reuse.

With the above pointers in mind, the following code samples demonstrate a workflow you can adapt to your code to incrementally update part of a model while simultaneously updating checkpoints.

Note: Due to the complexity of variable naming with Keras layers, this is not guaranteed to work for all use cases.

1. Begin by switching functional-style `tf.compat.v1.layers` to their object-oriented versions.

In [ ]:

```
class FunctionalStyleCompatModel(tf.keras.layers.Layer):
    @tf.compat.v1.keras.utils.track_tf1_style_variables
    def call(self, inputs, training=None):
        with tf.compat.v1.variable_scope('model'):
            out = tf.compat.v1.layers.conv2d(
                inputs, 3, 3,
                kernel_regularizer="l2")
            out = tf.compat.v1.layers.conv2d(
                out, 4, 4,
                kernel_regularizer="l2")
            out = tf.compat.v1.layers.conv2d(
                out, 5, 5,
                kernel_regularizer="l2")
        return out

layer = FunctionalStyleCompatModel()
layer(tf.ones(shape=(10, 10, 10, 10)))
[v.name for v in layer.weights]
```

1. Next, assign the `compat.v1.layer` objects and any variables created by `compat.v1.get_variable` as properties of the `tf.keras.layers.Layer / tf.Module` object whose method is decorated with `track_tf1_style_variables` (note that any object-oriented TF2 style checkpoints will now save out both a path by variable name and the new object-oriented path).

In [ ]:

```
class O0StyleCompatModel(tf.keras.layers.Layer):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.conv_1 = tf.compat.v1.layers.Conv2D(
            3, 3,
            kernel_regularizer="l2")
        self.conv_2 = tf.compat.v1.layers.Conv2D(
            4, 4,
            kernel_regularizer="l2")

    @tf.compat.v1.keras.utils.track_tf1_style_variables
    def call(self, inputs, training=None):
        with tf.compat.v1.variable_scope('model'):
            out = self.conv_1(inputs)
            out = self.conv_2(out)
            out = tf.compat.v1.layers.conv2d(
                out, 5, 5,
                kernel_regularizer="l2")
        return out

layer = O0StyleCompatModel()
layer(tf.ones(shape=(10, 10, 10, 10)))
[v.name for v in layer.weights]
```

1. Resave a loaded checkpoint at this point to save out paths both by the variable name (for compat.v1.layers), or by the object-oriented object graph.

In [ ]:

```
weights = {v.name: v for v in layer.weights}
assert weights['model/conv2d/kernel:0'] is layer.conv_1.kernel
assert weights['model/conv2d_1/bias:0'] is layer.conv_2.bias
```

1. You can now swap out the object-oriented `compat.v1.layers` for native Keras layers while still being able to load the recently-saved checkpoint. Ensure that you preserve variable names for the remaining `compat.v1.layers` by still recording the auto-generated `variable_scopes` of the replaced layers. These switched layers/variables will now only use the object attribute path to the variables in the checkpoint instead of the variable name path.

In general, you can replace usage of `compat.v1.get_variable` in variables attached to properties by:

- Switching them to using `tf.Variable`, OR
- Updating them by using `tf.keras.layers.Layer.add_weight` ([https://www.tensorflow.org/api\\_docs/python/tf/keras/layers/Layer#add\\_weight](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Layer#add_weight)). Note that if you are not switching all layers in one go this may change auto-generated layer/variable naming for the remaining `compat.v1.layers` that are missing a `name` argument. If that is the case, you must keep the variable names for remaining `compat.v1.layers` the same by manually opening and closing a `variable_scope` corresponding to the removed `compat.v1.layer`'s generated scope name. Otherwise the paths from existing checkpoints may conflict and checkpoint loading will behave incorrectly.

In [ ]:

```
def record_scope(scope_name):
    """Record a variable_scope to make sure future ones get incremented."""
    with tf.compat.v1.variable_scope(scope_name):
        pass

class PartiallyNativeKerasLayersModel(tf.keras.layers.Layer):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.conv_1 = tf.keras.layers.Conv2D(
            3, 3,
            kernel_regularizer="l2")
        self.conv_2 = tf.keras.layers.Conv2D(
            4, 4,
            kernel_regularizer="l2")

    @tf.compat.v1.keras.utils.track_tf1_style_variables
    def call(self, inputs, training=None):
        with tf.compat.v1.variable_scope('model'):
            out = self.conv_1(inputs)
            record_scope('conv2d') # Only needed if follow-on compat.v1.layers do not pass a `name` arg
            out = self.conv_2(out)
            record_scope('conv2d_1') # Only needed if follow-on compat.v1.layers do not pass a `name` arg
            out = tf.compat.v1.layers.conv2d(
                out, 5, 5,
                kernel_regularizer="l2")
        return out

layer = PartiallyNativeKerasLayersModel()
layer(tf.ones(shape=(10, 10, 10, 10)))
[v.name for v in layer.weights]
```

Saving a checkpoint out at this step after constructing the variables will make it contain **only** the currently-available object paths.

Ensure you record the scopes of the removed `compat.v1.layers` to preserve the auto-generated weight names for the remaining `compat.v1.layers`.

In [ ]:

```
weights = set(v.name for v in layer.weights)
assert 'model/conv2d_2/kernel:0' in weights
assert 'model/conv2d_2/bias:0' in weights
```

1. Repeat the above steps until you have replaced all the `compat.v1.layers` and `compat.v1.get_variables`s in your model with fully-native equivalents.

In [ ]:

```
class FullyNativeKerasLayersModel(tf.keras.layers.Layer):
```

```
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.conv_1 = tf.keras.layers.Conv2D(
            3, 3,
            kernel_regularizer="l2")
        self.conv_2 = tf.keras.layers.Conv2D(
            4, 4,
            kernel_regularizer="l2")
        self.conv_3 = tf.keras.layers.Conv2D(
            5, 5,
            kernel_regularizer="l2")

    def call(self, inputs, training=None):
        with tf.compat.v1.variable_scope('model'):
            out = self.conv_1(inputs)
            out = self.conv_2(out)
            out = self.conv_3(out)
        return out
```

```
layer = FullyNativeKerasLayersModel()
layer(tf.ones(shape=(10, 10, 10, 10)))
[v.name for v in layer.weights]
```

Remember to test to make sure the newly updated checkpoint still behaves as you expect. Apply the techniques described in the [validate numerical correctness guide](#) ([./validate\\_correctness.ipynb](#)) at every incremental step of this process to ensure your migrated code runs correctly.

## Handling TF1.x to TF2 behavior changes not covered by the modeling shims

The modeling shims described in this guide can make sure that variables, layers, and regularization losses created with `get_variable`, `tf.compat.v1.layers`, and `variable_scope` semantics continue to work as before when using eager execution and `tf.function`, without having to rely on collections.

This does not cover **all** TF1.x-specific semantics that your model forward passes may be relying on. In some cases, the shims might be insufficient to get your model forward pass running in TF2 on their own. Read the [TF1.x vs TF2 behaviors guide \(./tf1\\_vs\\_tf2\)](#) to learn more about the behavioral differences between TF1.x and TF2.

**Copyright 2021 The TensorFlow Authors.**

In [ ]:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## Migrate early stopping



[View on TensorFlow.org](#)

([https://www.tensorflow.org/guide/migrate/early\\_stopping](https://www.tensorflow.org/guide/migrate/early_stopping))



[Run in Google Colab](#)

([https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/early\\_stopping.ipynb](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/migrate/early_stopping.ipynb)) (<http://>)

This notebook demonstrates how you can set up model training with early stopping, first, in TensorFlow 1 with `tf.estimator.Estimator` and an early stopping hook, and then, in TensorFlow 2 with Keras APIs or a custom training loop. Early stopping is a regularization technique that stops training if, for example, the validation loss reaches a certain threshold.

In TensorFlow 2, there are three ways to implement early stopping:

- Use a built-in Keras callback—`tf.keras.callbacks.EarlyStopping`—and pass it to `Model.fit`.
- Define a custom callback and pass it to Keras `Model.fit`.
- Write a custom early stopping rule in a [custom training loop](#) ([https://www.tensorflow.org/guide/keras/writing\\_a\\_training\\_loop\\_from\\_scratch](https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch)) (with `tf.GradientTape`).

## Setup

In [ ]:

```
import time
import numpy as np
import tensorflow as tf
import tensorflow.compat.v1 as tf1
import tensorflow_datasets as tfds
```

## TensorFlow 1: Early stopping with an early stopping hook and `tf.estimator`

Start by defining functions for MNIST dataset loading and preprocessing, and model definition to be used with `tf.estimator.Estimator`:

In [ ]:

```
def normalize_img(image, label):
    return tf.cast(image, tf.float32) / 255., label

def _input_fn():
    ds_train = tfds.load(
        name='mnist',
        split='train',
        shuffle_files=True,
        as_supervised=True)

    ds_train = ds_train.map(
        normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
    ds_train = ds_train.batch(128)
    ds_train = ds_train.repeat(100)
    return ds_train

def _eval_input_fn():
    ds_test = tfds.load(
        name='mnist',
        split='test',
        shuffle_files=True,
        as_supervised=True)
    ds_test = ds_test.map(
        normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
    ds_test = ds_test.batch(128)
    return ds_test

def _model_fn(features, labels, mode):
    flatten = tf1.layers.Flatten()(features)
    features = tf1.layers.Dense(128, 'relu')(flatten)
    logits = tf1.layers.Dense(10)(features)

    loss = tf1.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)
    optimizer = tf1.train.AdagradOptimizer(0.005)
    train_op = optimizer.minimize(loss, global_step=tf1.train.get_global_step())

    return tf1.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)
```

In TensorFlow 1, early stopping works by setting up an early stopping hook with `tf.estimator.experimental.make_early_stopping_hook`. You pass the hook to the `make_early_stopping_hook` method as a parameter for `should_stop_fn`, which can accept a function without any arguments. The training stops once `should_stop_fn` returns `True`.

The following example demonstrates how to implement an early stopping technique that limits the training time to a maximum of 20 seconds:

In [ ]:

```
estimator = tf1.estimator.Estimator(model_fn=_model_fn)

start_time = time.time()
max_train_seconds = 20

def should_stop_fn():
    return time.time() - start_time > max_train_seconds

early_stopping_hook = tf1.estimator.experimental.make_early_stopping_hook(
    estimator=estimator,
    should_stop_fn=should_stop_fn,
    run_every_secs=1,
    run_every_steps=None)

train_spec = tf1.estimator.TrainSpec(
    input_fn=_input_fn,
    hooks=[early_stopping_hook])

eval_spec = tf1.estimator.EvalSpec(input_fn=_eval_input_fn)

tf1.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

## TensorFlow 2: Early stopping with a built-in callback and Model.fit

Prepare the MNIST dataset and a simple Keras model:

In [ ]:

```
(ds_train, ds_test), ds_info = tfds.load(
    'mnist',
    split=['train', 'test'],
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)

ds_train = ds_train.map(
    normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
ds_train = ds_train.batch(128)

ds_test = ds_test.map(
    normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
ds_test = ds_test.batch(128)

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
model.compile(
    optimizer=tf.keras.optimizers.Adam(0.005),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
)
```

In TensorFlow 2, when you use the built-in Keras `Model.fit` (or `Model.evaluate`), you can configure early stopping by passing a built-in callback — `tf.keras.callbacks.EarlyStopping` — to the `callbacks` parameter of `Model.fit`.

The `EarlyStopping` callback monitors a user-specified metric and ends training when it stops improving. (Check the [Training and evaluation with the built-in methods](https://www.tensorflow.org/guide/keras/train_and_evaluate#using_callbacks) ([https://www.tensorflow.org/guide/keras/train\\_and\\_evaluate#using\\_callbacks](https://www.tensorflow.org/guide/keras/train_and_evaluate#using_callbacks)) or the [API docs](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping) ([https://www.tensorflow.org/api\\_docs/python/tf/keras/callbacks/EarlyStopping](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping)) for more information.)

Below is an example of an early stopping callback that monitors the loss and stops training after the number of epochs that show no improvements is set to 3 ( `patience` ):

In [ ]:

```
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)

# Only around 25 epochs are run during training, instead of 100.
history = model.fit(
    ds_train,
    epochs=100,
    validation_data=ds_test,
    callbacks=[callback]
)

len(history.history['loss'])
```

## TensorFlow 2: Early stopping with a custom callback and Model.fit

You can also implement a [custom early stopping callback](https://www.tensorflow.org/guide/keras/custom_callback#early_stopping_at_minimum_loss) ([https://www.tensorflow.org/guide/keras/custom\\_callback#early\\_stopping\\_at\\_minimum\\_loss](https://www.tensorflow.org/guide/keras/custom_callback#early_stopping_at_minimum_loss)), which can also be passed to the `callbacks` parameter of `Model.fit` (or `Model.evaluate`).

In this example, the training process is stopped once `self.model.stop_training` is set to be `True` :

In [ ]:

```
class LimitTrainingTime(tf.keras.callbacks.Callback):
    def __init__(self, max_time_s):
        super().__init__()
        self.max_time_s = max_time_s
        self.start_time = None

    def on_train_begin(self, logs):
        self.start_time = time.time()

    def on_train_batch_end(self, batch, logs):
        now = time.time()
        if now - self.start_time > self.max_time_s:
            self.model.stop_training = True
```

In [ ]:

```
# Limit the training time to 30 seconds.
callback = LimitTrainingTime(30)
history = model.fit(
    ds_train,
    epochs=100,
    validation_data=ds_test,
    callbacks=[callback]
)
len(history.history['loss'])
```

## TensorFlow 2: Early stopping with a custom training loop

In TensorFlow 2, you can implement early stopping in a [custom training loop](#)

([https://www.tensorflow.org/tutorials/customization/custom\\_training\\_walkthrough#training\\_loop](https://www.tensorflow.org/tutorials/customization/custom_training_walkthrough#training_loop)) if you're not training and evaluating with the [built-in Keras methods](#) ([https://www.tensorflow.org/guide/keras/train\\_and\\_evaluate](https://www.tensorflow.org/guide/keras/train_and_evaluate)).

Start by using Keras APIs to define another simple model, an optimizer, a loss function, and metrics:

In [ ]:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])

optimizer = tf.keras.optimizers.Adam(0.005)
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

train_acc_metric = tf.keras.metrics.SparseCategoricalAccuracy()
train_loss_metric = tf.keras.metrics.SparseCategoricalCrossentropy()
val_acc_metric = tf.keras.metrics.SparseCategoricalAccuracy()
val_loss_metric = tf.keras.metrics.SparseCategoricalCrossentropy()
```

Define the parameter update functions [with tf.GradientTape](#) ([https://www.tensorflow.org/guide/keras/writing\\_a\\_training\\_loop\\_from\\_scratch](https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch)) and the `@tf.function` decorator [for a speedup](#) (<https://www.tensorflow.org/guide/function>):

In [ ]:

```
@tf.function
def train_step(x, y):
    with tf.GradientTape() as tape:
        logits = model(x, training=True)
        loss_value = loss_fn(y, logits)
    grads = tape.gradient(loss_value, model.trainable_weights)
    optimizer.apply_gradients(zip(grads, model.trainable_weights))
    train_acc_metric.update_state(y, logits)
    train_loss_metric.update_state(y, logits)
    return loss_value

@tf.function
def test_step(x, y):
    logits = model(x, training=False)
    val_acc_metric.update_state(y, logits)
    val_loss_metric.update_state(y, logits)
```

Next, write a custom training loop, where you can implement your early stopping rule manually.

The example below shows how to stop training when the validation loss doesn't improve over a certain number of epochs:

In [ ]:

```
epochs = 100
patience = 5
wait = 0
best = 0

for epoch in range(epochs):
    print("\nStart of epoch %d" % (epoch,))
    start_time = time.time()

    for step, (x_batch_train, y_batch_train) in enumerate(ds_train):
        loss_value = train_step(x_batch_train, y_batch_train)
        if step % 200 == 0:
            print("Training loss at step %d: %.4f" % (step, loss_value.numpy()))
            print("Seen so far: %s samples" % ((step + 1) * 128))
        train_acc = train_acc_metric.result()
        train_loss = train_loss_metric.result()
        train_acc_metric.reset_states()
        train_loss_metric.reset_states()
        print("Training acc over epoch: %.4f" % (train_acc.numpy()))

    for x_batch_val, y_batch_val in ds_test:
        test_step(x_batch_val, y_batch_val)
    val_acc = val_acc_metric.result()
    val_loss = val_loss_metric.result()
    val_acc_metric.reset_states()
    val_loss_metric.reset_states()
    print("Validation acc: %.4f" % (float(val_acc),))
    print("Time taken: %.2fs" % (time.time() - start_time))

# The early stopping strategy: stop the training if `val_loss` does not
# decrease over a certain number of epochs.
wait += 1
if val_loss > best:
    best = val_loss
    wait = 0
if wait >= patience:
    break
```

## Next steps

- Learn more about the Keras built-in early stopping callback API in the [API docs](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping) ([https://www.tensorflow.org/api\\_docs/python/tf/keras/callbacks/EarlyStopping](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping)).
- Learn to [write custom Keras callbacks](https://www.tensorflow.org/guide/keras/custom_callback) ([https://www.tensorflow.org/guide/keras/custom\\_callback](https://www.tensorflow.org/guide/keras/custom_callback)), including [early stopping at a minimum loss](https://www.tensorflow.org/guide/keras/custom_callback/#early_stopping_at_minimum_loss) ([https://www.tensorflow.org/guide/keras/custom\\_callback/#early\\_stopping\\_at\\_minimum\\_loss](https://www.tensorflow.org/guide/keras/custom_callback/#early_stopping_at_minimum_loss)).
- Learn about [Training and evaluation with the Keras built-in methods](https://www.tensorflow.org/guide/keras/train_and_evaluate#using_callbacks) ([https://www.tensorflow.org/guide/keras/train\\_and\\_evaluate#using\\_callbacks](https://www.tensorflow.org/guide/keras/train_and_evaluate#using_callbacks)).
- Explore common regularization techniques in the [Overfit and underfit](https://tensorflow.org/tutorials/keras/overfit_and_underfit) ([tensorflow.org/tutorials/keras/overfit\\_and\\_underfit](https://tensorflow.org/tutorials/keras/overfit_and_underfit)) tutorial that uses the EarlyStopping callback.