Machine learning (ML) research shows that many machine learning models can tolerate lower precision arithmetic without degradation of converged accuracy. Many models reach results with the same converged accuracy using bfloat16 as when using 32 bit floating point numerics and some models even show improved converged accuracy with bfloat16.

This document concerns mixed precision training in the sense of storing activations and gradients in memory using the bfloat16 format. (For more background on mixed precision training see Mixed Precision Training (https://arxiv.org/abs/1710.03740).) This document does not discuss the use of bfloat16 in the MXU on Cloud TPU.

The following topics apply to ML models using TensorFlow:

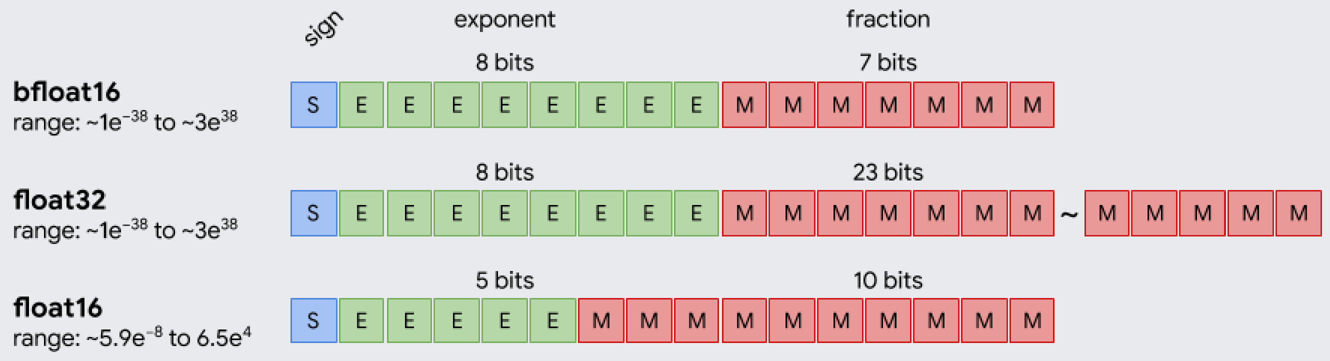
- Description of Google's custom 16-bit brain floating-point, *bfloat16*.
- Performance advantages of using bfloat16 in memory for ML models on hardware that supports it, such as Cloud TPU.
- How to store activations and gradients in memory using bfloat16 for a TPU model in TensorFlow.

By default, TensorFlow stores all variables in 32-bit floating-point (fp32). Using bfloat16 for the activations and gradients speeds up device step time and decreases memory usage. See Changing your model (#changing) for determining the benefits of using bfloat16 for activations and gradients in your model.

The bfloat16 format is [1:8:7], which has one sign bit, eight exponent bits, and seven mantissa bits plus one implicit mantissa bit. By comparison, the standard 16-bit floating-point (fp16) format is [1:5:10]. Notice that the fp16 format has only 5 exponent bits. Because of these characteristics, bfloat16 has a greater dynamic range than fp16. The bfloat16 range is useful for things like gradients that can be outside the dynamic range of fp16 and thus require loss scaling; bfloat16 can represent such gradients directly. In addition, you can use the bfloat16 format to accurately represent all integers [-256, 256], which means you can encode an int8 in bfloat16 without loss of accuracy.

The following figure shows three floating-points formats

- fp32 - IEEE single-precision floating-point
- fp16 - IEEE half-precision floating point
- bfloat16 - 16-bit *brain floating point*

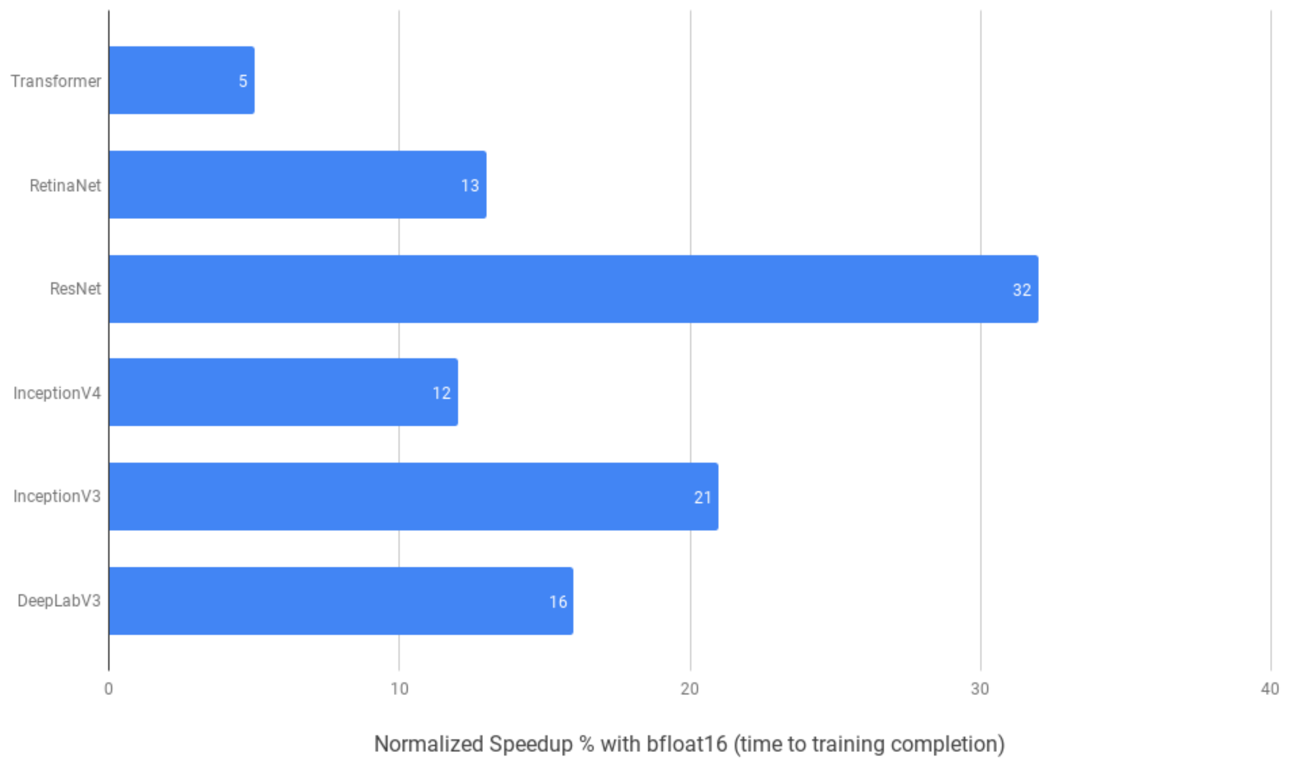


The dynamic range of bfloat16 is greater than that of fp16.

Cloud TPU supports storing values such as activations and gradients in bfloat16 format. Using bfloat16 reduces the size of data in memory and allows larger models to fit in the same amount of memory. Using bfloat16 can also reduce rematerialization which improves step time.

Some operations are memory-bandwidth-bound, which means the memory bandwidth determines the time spent in such operations. Storing inputs and outputs of memory-bandwidth-bound operations in the bfloat16 format reduces the amount of data that must be transferred, thus improving the speed of the operations.

The following chart shows improvements seen in our internal experiments.



By default, activations, gradients, and weights are stored in fp32 in memory. You can use bfloat16 for activations and gradients, leaving weights in fp32, and then compare your model's performance between using bfloat16 and fp32 to determine the benefits.

1. Run the model in fp32 using capture_tpu_profile (/tpu/docs/cloud-tpu-tools#install_cloud_tpu_profiler).
2. To view the model's step time and converged accuracy, use the profile viewer in TensorBoard. (See Using Cloud TPU tools (/tpu/docs/cloud-tpu-tools) for details.)
3. Cast the input to bfloat16 in your input pipeline within the record parser so that the conversion can be done in parallel rather than at the end of the input_fn. Doing this converts all of the activations and gradients in the model to bfloat16.

   For example:

4. Create your network under the bfloat16 scope and then cast the outputs of the model to float32. (https://github.com/tensorflow/tpu/blob/master/models/official/resnet/resnet_main.py#L306)

After you configure your model to use tf.bfloat16 for activations, check the following to see the impact of bfloat16 on your model:

1. Run the model with bfloat16 using capture_tpu_profile (/tpu/docs/cloud-tpu-tools#install_cloud_tpu_profiler).
2. To view the model's step time and converged accuracy, use the profile viewer in TensorBoard. (See Using Cloud TPU tools (/tpu/docs/cloud-tpu-tools) for details.)
3. Compare the step time for bfloat16 and fp32. Step time typically improves for bfloat16.
4. Compare converged accuracy for bfloat16 versus fp32. Usually they are identical, but values can be better or worse than expected.

If you do not already know what range of variation to expect of your model, you might need multiple runs to determine run-to-run variation in converged accuracy.

If your profile shows that the processing time is faster but the input pipeline has become a bottleneck, optimize your input pipeline to realize an even greater speed advantage. See Data Input Pipeline Performance (https://www.tensorflow.org/guide/performance/datasets) for general guidance on improving TensorFlow pipeline performance.

A best practice for training and inference is to use the same precision for both. It is possible to train using fp32 for activations, and then run inference with bfloat16 (or vice versa). If you opt for mismatched precision, verify converged accuracy using the precision that was used for inference.