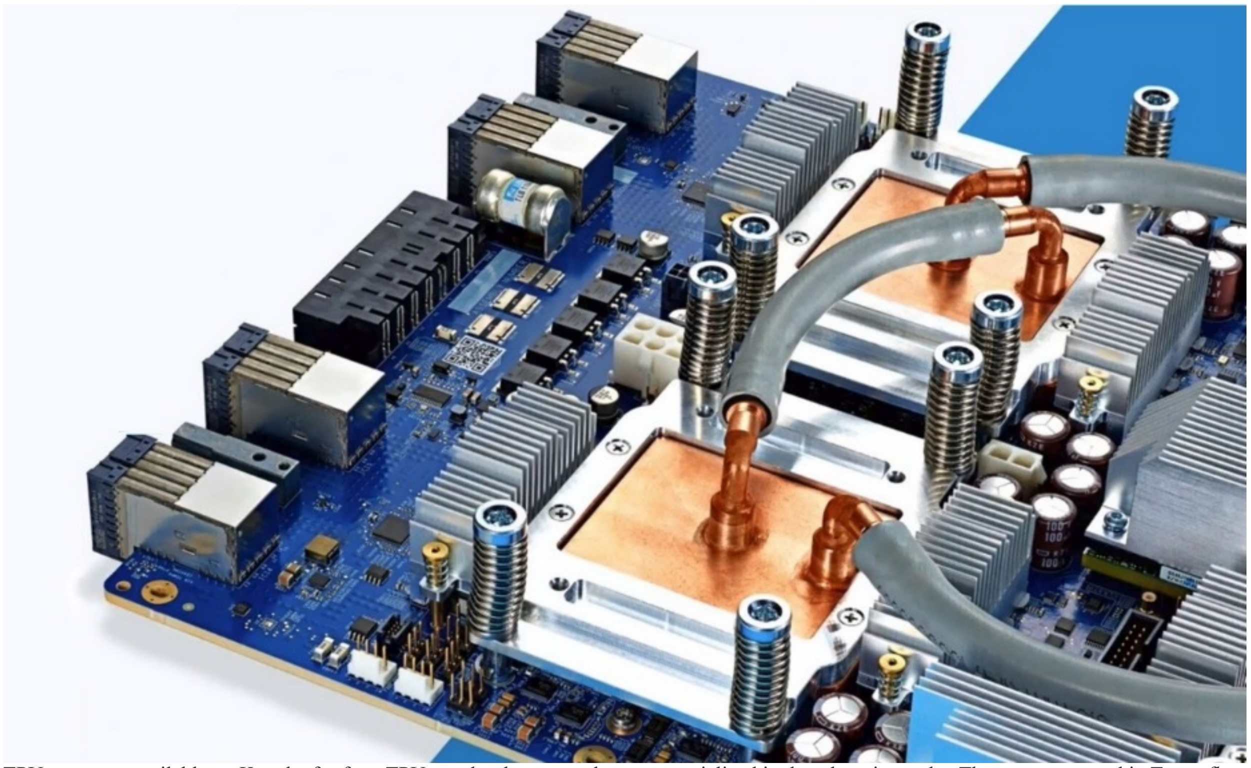
Tensor Processing Units (TPUs)



TPUs are now available on Kaggle, for free. TPUs are hardware accelerators specialized in deep learning tasks. They are supported in Tensorflow 2.1 both through the Keras high-level API and, at a lower level, in models using a custom training loop.

You can use up to 20 hours per week of TPUs and up to 9h at a time in a single session.

This page explains how to: 1) Enable TPUs in Tensorflow and Keras, 2) adjust the batch size and learning rate, 3) optimize your data pipeline for a fast accelerator

If you'd like to jump straight into a sample, here it is: Five flowers with Keras and Xception on TPU

The following documentation was written for earlier TPU versions that Kaggle no longer supports. For help with newer versions, see official TPU documentation.

TPUs in Keras

Once you have flipped the "Accelerator" switch in your notebook to "TPU v3-8", this is how to enable TPU training in Tensorflow Keras:

```
# detect and init the TPU
tpu = tf.distribute.cluster_resolver.TPUClusterResolver()

# instantiate a distribution strategy
tf.tpu.experimental.initialize_tpu_system(tpu)
tpu_strategy = tf.distribute.TPUStrategy(tpu)

# instantiating the model in the strategy scope creates the model on the TPU
with tpu_strategy.scope():
    model = tf.keras.Sequential( … ) # define your model normally
    model.compile( … )

# train model normally
model.fit(training_dataset, epochs=EPOCHS, steps_per_epoch=…)
```

TPUs are network-connected accelerators and you must first locate them on the network. This is what `TPUClusterResolver.connect()` does.

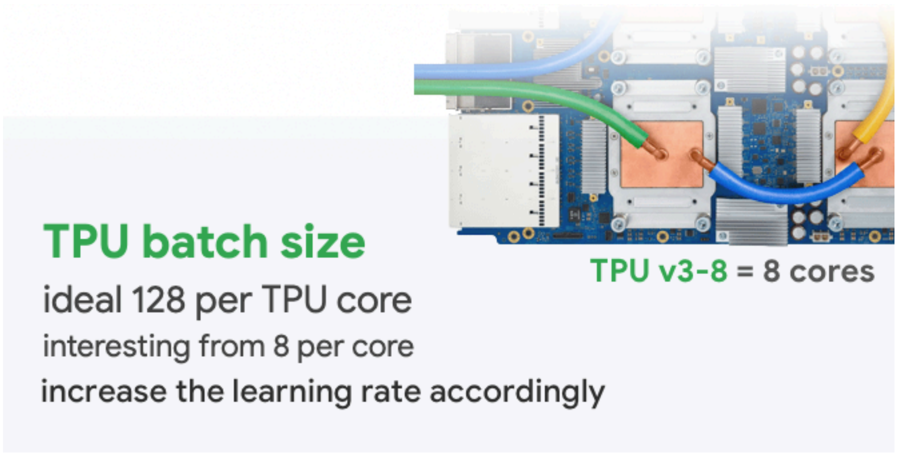
You then instantiate a `TPUStrategy`. This object contains the necessary distributed training code that will work on TPUs with their 8 compute cores (see hardware section below).

Finally, you use the `TPUStrategy` by instantiating your model in the scope of the strategy. This creates the model on the TPU. Model size is constrained by the TPU RAM only, not by the amount of memory available on the VM running your Python code. Model creation and model training use the usual Keras APIs.

Batch size, learning rate, steps_per_execution

To go fast on a TPU, increase the batch size. The rule of thumb is to use batches of 128 elements per core (ex: batch size of 128*8=1024 for a TPU with 8 cores). At this size, the 128x128 hardware matrix multipliers of the TPU (see hardware section below) are most likely to be kept busy. You start seeing interesting speedups from a batch size of 8 per core though. In the sample above, the batch size is scaled with the core count through this line of code:
`BATCH_SIZE = 16 * tpu_strategy.num_replicas_in_sync`

With a TPUStrategy running on a single TPU v3-8, the core count is 8. This is the hardware available on Kaggle. It could be more on larger configurations called TPU pods available on Google Cloud.


TPU batch size
ideal 128 per TPU core
interesting from 8 per core
increase the learning rate accordingly
TPU v3-8 = 8 cores

With larger batch sizes, TPUs will be crunching through the training data faster. This is only useful if the larger training batches produce more "training work" and get your model to the desired accuracy faster. That is why the rule of thumb also calls for increasing the learning rate with the batch size. You can start with a proportional increase but additional tuning may be necessary to find the optimal learning rate schedule for a given model and accelerator.

Starting with Tensorflow 2.4, model.compile() accepts a new `steps_per_execution` parameter. This parameter instructs Keras to send multiple batches to the TPU at once. In addition to lowering communications overheads, this gives the XLA compiler the opportunity to optimize TPU hardware utilization across multiple batches. With this option, it is no longer necessary to push batch sizes to very high values to optimize TPU performance. As long as you use batch sizes of at least 8 per core (>=64 for a TPUv3-8) performance should be acceptable. Example:

```
    model.compile( … ,
                  steps_per_execution=32)
```

tf.data.Dataset and TFRecords

Because TPUs are very fast, many models ported to TPU end up with a data bottleneck. The TPU is sitting idle, waiting for data for the most part of each training epoch. TPUs read training data exclusively from GCS (Google Cloud Storage). And GCS can sustain a pretty large throughput if it is continuously streaming from multiple files in parallel. Following a couple of best practices will optimize the throughput:

> *For TPU training, organize your data in GCS in a reasonable number (10s to 100s) of reasonably large files (10s to 100s of MB).*

With too few files, GCS will not have enough streams to get max throughput. With too many files, time will be wasted accessing each individual file.

Data for TPU training typically comes sharded across the appropriate number of larger files. The usual container format is TFRecords. You can load a dataset from TFRecords files by writing:

```
# On Kaggle you can also use KaggleDatasets().get_gcs_path() to obtain the GCS path of a Kaggle dataset
filenames = tf.io.gfile.glob("gs://flowers-public/tfrecords-jpeg-512x512/*.tfrec") # list files on GCS
dataset = tf.data.TFRecordDataset(filenames)
dataset = dataset.map(...) # TFRecord decoding here...
```

To enable parallel streaming from multiple TFRecord files, modify the code like this:

```
    AUTO = tf.data.experimental.AUTOTUNE
    ignore_order = tf.data.Options()
    ignore_order.experimental_deterministic = False

    # On Kaggle you can also use KaggleDatasets().get_gcs_path() to obtain the GCS path of a Kaggle dataset
    filenames = tf.io.gfile.glob("gs://flowers-public/tfrecords-jpeg-512x512/*.tfrec") # list files on GCS
    dataset = tf.data.TFRecordDataset(filenames, num_parallel_reads=AUTO)
    dataset = dataset.with_options(ignore_order)
    dataset = dataset.map(...) # TFRecord decoding here...
```

There are two settings here:

- `num_parallel_reads=AUTO` instructs the API to read from multiple files if available. It figures out how many automatically.
- `experimental_deterministic = False` disables data order enforcement. We will be shuffling the data anyway so order is not important. With this setting the API can use any TFRecord as soon as it is streamed in.

Some details have been omitted from these code snippets so check the sample for the full data pipeline code. In Keras and TensorFlow 2.1, it is also possible to send training data to TPUs as numpy arrays in memory. This works but is not the most efficient way, although for datasets that fit in memory, it can be OK.

Private Datasets with TPUs

TPUs work with both public Kaggle Datasets as well as private Kaggle Datasets. The only difference is that if you want to use a private Kaggle Dataset then you need to: (1) enable "Google Cloud SDK" in the "Add-ons" menu of the notebook editor; (2) Initialize the TPU and then run the "Google Cloud SDK credentials" code snippet; finally (3) take note of the Google Cloud Storage path that is returned.

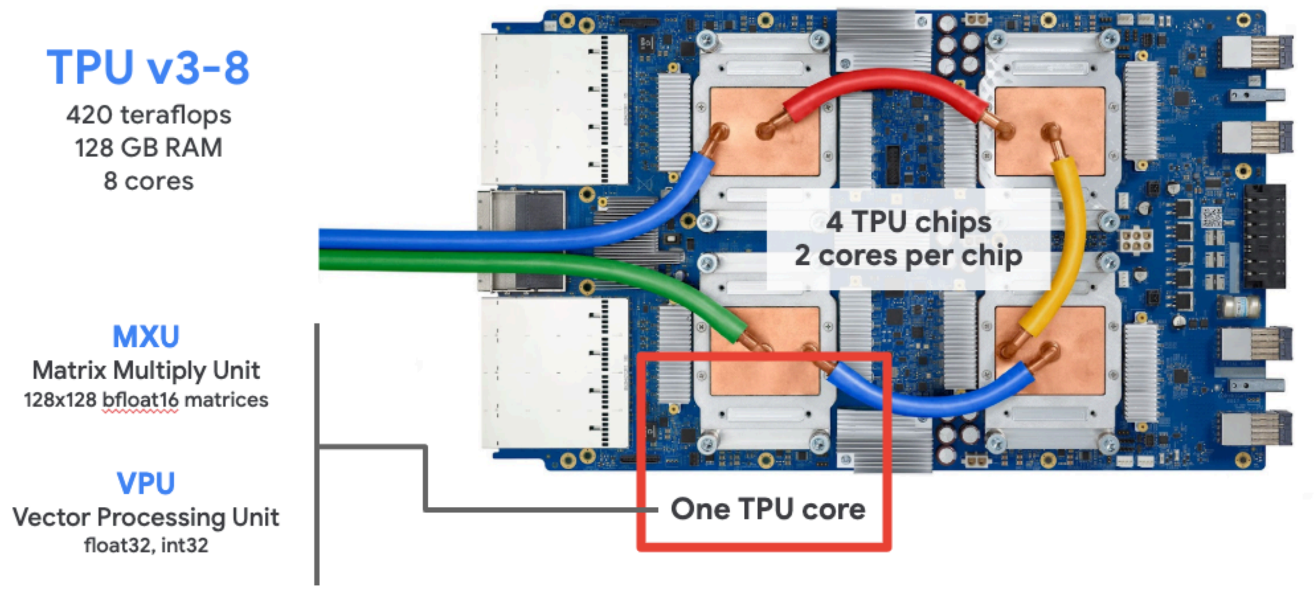
```
    # Step 1: Get the credential from the Cloud SDK
    from kaggle_secrets import UserSecretsClient
    user_secrets = UserSecretsClient()
    user_credential = user_secrets.get_gcloud_credential()

    # Step 2: Set the credentials
    user_secrets.set_tensorflow_credential(user_credential)
    # Step 3: Use a familiar call to get the GCS path of the dataset
    from kaggle_datasets import KaggleDatasets
    GCS_DS_PATH = KaggleDatasets().get_gcs_path()
```

If you are working with a public Kaggle Dataset then only Step #3 is necessary.

TPU hardware

At approximately 20 inches (50 cm), a TPU v3-8 board is a fairly sizeable piece of hardware. It sports 4 dual-core TPU chips for a total of 8 TPU cores.



Each TPU core has a traditional vector processing part (VPU) as well as dedicated matrix multiplication hardware capable of processing 128x128 matrices. This is the part that specifically accelerates machine learning workloads.

TPUs are equipped with 128GB of high-speed memory allowing larger batches, larger models and also larger training inputs. In the sample above, you can try using 512x512 px input images, also provided in the dataset, and see the TPU v3-8 handle them easily.

Model saving/loading on TPUs

When loading and saving models TPU models from/to the local disk, the experimental_io_device option must be used. The technical explanation is at the end of this section. It can be omitted if writing to GCS because TPUs have direct access to GCS. This option does nothing on GPUs.

Saving a TPU model locally

```
save_locally = tf.saved_model.SaveOptions(experimental_io_device='/job:localhost')
model.save('./model', options=save_locally) # saving in Tensorflow's "SavedModel" format
```

Loading a TPU model from local disk

```
with strategy.scope():
    load_locally = tf.saved_model.LoadOptions(experimental_io_device='/job:localhost')
    model = tf.keras.models.load_model('./model', options=load_locally) # loading in Tensorflow's "SavedModel" format
```

Writing checkpoints locally from a TPU model

```
save_locally = tf.saved_model.SaveOptions(experimental_io_device='/job:localhost')
checkpoints_cb = tf.keras.callbacks.ModelCheckpoint('./checkpoints', options=save_locally)
model.fit(…, callbacks=[checkpoints_cb])
```

Loading a model from Tensorflow Hub to TPU directly

```
import tensorflow_hub as hub
with strategy.scope():
    load_locally = tf.saved_model.LoadOptions(experimental_io_device='/job:localhost')
    pretrained_model = hub.KerasLayer('https://tfhub.dev/tensorflow/efficientnet/b6/feature-vector/1', trainable=True, input_shape=[512,512,3], lo
```

Example in this EfficientNetB7 Notebook.

experimental_io_device explained

To understand what the experimental_io_device='/job:localhost' flag does, some background info is needed first. TPU users will remember that in order to train a model on TPU, you have to instantiate the model in a TPUStrategy scope. Like this:

```
# connect to a TPU and instantiate a distribution strategy
tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='local')
tf.tpu.experimental.initialize_tpu_system(tpu)
tpu_strategy = tf.distribute.TPUStrategy(tpu)

# instantiate the model in the strategy scope
with tpu_strategy.scope():
    model = tf.keras.Sequential( … )
```

This boilerplate code actually does 2 things:

The strategy scope instructs Tensorflow to instantiate all the variables of the model in the memory of the TPU. The TPUClusterResolver.connect() call automatically enters the TPU device scope which instructs Tensorflow to run Tensorflow operations on the TPU. Now if you call model.save('./model') when you are connected to a TPU, Tensorflow will try to run the save operations on the TPU and since the TPU is a network-connected accelerator that has no access to your local disk, the operation will fail. Notice that saving to GCS will work though. The TPU does have access to GCS.

If you want to save a TPU model to your local disk, you need to run the saving operation on your local machine and that is what the experimental_io_device='/job:localhost' flag does.

TPUs in Code Competitions

Due to technical limitations for certain kinds of code-only competitions we aren't able to support notebook submissions that run on TPUs, made clear in the competition's rules. But that doesn't mean you can't use TPUs to train your models!

A workaround to this restriction is to run your model training in a separate notebook that uses TPUs, and then to save the resulting model. You can then load that model into the notebook you use for your submission and use a GPU to run inference and generate your predictions.

Here's how that would work in practice:

Step 1: Save the Model

```
# Save your model to disk using the .save() functionality. Here we save in .h5 format
# This step will be replaced with an alternative call to save models in Tensorflow 2.3
model.save('model.h5')
```

Step 2: Put your model in a dataset

You can easily create a dataset from the output of your notebook from the dataviewer. For more details, you can see our Dataset Documentation

Step 3: Load your model into inference Notebook

```
# You can now load your model and run inference using a GPU in this notebook.
# Because this notebook only uses a GPU, you can submit it to competitions

model = tf.keras.models.load_model('../input/yourDataset/model.h5')
```

More information and tutorials

A hands-on TPU tutorial containing more information, best practices and samples is available here:
Keras and modern convnets, on TPUs.

You can also check out our TPU video tutorial, Learn With Me: Getting Started With TPUs, on our YouTube channel!

TPU playground competition

We have prepared a dataset of 13,000 images of flowers for you to play with. You can give TPUs a try in this playground competition: Flower Classification with TPUs

For an easy way to begin, check out this tutorial notebook and starter project, a part of our Deep Learning course:

TPUs in PyTorch

Once you have flipped the "Accelerator" switch in your notebook to "TPU v3-8", this is how to enable TPU training in Tensorflow PyTorch:

```
# Step 1: Install Torch-XLA (PyTorch with Accelerated Linear Algebra (XLA) support)
!curl https://raw.githubusercontent.com/pytorch/xla/master/contrib/scripts/env-setup.py -o pytorch-xla-env-setup.py
!python pytorch-xla-env-setup.py --version nightly --apt-packages libomp5 libopenblas-dev

# Step 2: Run your PyTorch code
TPUs (TPU v3-8) have 8 cores, and each core is itself an XLA device.
You can run code on a single XLA device, but to take full advantage of
the TPU you will want to run your code on all 8 cores simultaneously.
For examples that demonstrate how to do this, you can refer to
The Ultimate PyTorch TPU Tutorial,
I Like Clean TPU Training Kernels and I Can Not Lie,
Super Duper Fast PyTorch TPU Kernel,
and  XLM Roberta Large Pytorch TPU
```

You should also note the following when using TPUs with PyTorch:

```
#1: Startup Script
https://raw.githubusercontent.com/pytorch/xla/master/contrib/scripts/env-setup.py

#2: Distributed training function mp_fn
xmp.spawn(_mp_fn, nprocs=8, start_method='fork')

#3: Instantiate model outside of mp_fn and use MpModelWrapper
MX = JigsawModel()   =>    MX = xmp.MpModelWrapper(JigsawModel())

#4: Send model to TPU device
device = xm.xla_device()
model = MX.to(device)

#5: Changes to training loop: send data to device
ids = ids.to(device, dtype=torch.long)
token_type_ids = token_type_ids.to(device, dtype=torch.long)
mask = mask.to(device, dtype=torch.long)
targets = targets.to(device, dtype=torch.float)


#6: Printing messages
```

```
xm.master_print


#7: Loading data
train_dataset = … # user-defined, can be outside of mp_fn
# in mp_fn:
train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset,
            num_replicas=xm.xrt_world_size(),rank=xm.get_ordinal(), …)
train_data_loader = torch.utils.data.DataLoader(train_dataset,
                            sampler=train_sampler, …)


#8: Training on data
for epoch in range(EPOCHS):
    para_loader = pl.ParallelLoader(train_data_loader, [device])
    train_fn(para_loader.per_device_loader(device), …)


#9: Results from TPU
xm.mesh_reduce


#10: Model save / restore (memory-optimized)
import torch_xla.utils.serialization as xser
xser.save(model.state_dict(), f"model.bin", master_only=True)
model.load_state_dict(xser.load(f"model.bin"))

#11: Model save / restore (PyTorch standard)
torch_xla.core.xla_model.save
torch.load(...)

#12: Out of memory datasets:
Can be loaded from localhost
Of loaded from GCS in TFRecord format, a TFRecords PyTorch loader exists
```