Migrating from Estimator API to TPUEstimator API

This tutorial (https://github.com/tensorflow/tpu/blob/master/models/samples/core/get_started/) describes how to convert a model program using the Estimator API to one using the TPUEstimator API.

Overview

Model programs that use the TPUEstimator API can take full advantage of Tensor Processing Units (TPUs), while remaining compatible with CPUs and GPUs.

After you finish this tutorial, you will know:

- How to convert your code from using the Estimator API to using the TPUEstimator API
- How to run predictions on Cloud TPU

Note: This tutorial does not cover specifics regarding the model logic. It also does not describe performance optimizations.

Before You Begin

Before starting this tutorial, check that your Google Cloud project is correctly set up.

1. Sign in (https://accounts.google.com/Login) to your Google Account.

   If you don't already have one, sign up for a new account (https://accounts.google.com/SignUp).
2. In the Cloud Console, on the project selector page, select or create a Cloud project.

★ **Note**: If you don't plan to keep the resources that you create in this procedure, create a project instead of selecting an existing project. After you finish these steps, you can delete the project, removing all resources associated with the project.

GO TO THE PROJECT SELECTOR PAGE (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/PROJECTSELECT

3. Make sure that billing is enabled for your Google Cloud project. Learn how to confirm billing is enabled for your project (https://cloud.google.com/billing/docs/how-to/modify-project).

   This walkthrough uses billable components of Google Cloud. Check the Cloud TPU pricing page (https://cloud.google.com/tpu/docs/pricing) to estimate your costs. Be sure to clean up (#clean_up) resources you create when you've finished with them to avoid unnecessary charges.

Set up your resources

This section provides information on setting up Cloud Storage storage, VM, and Cloud TPU resources for tutorials.

Important: Set up all resources in the same region/zone to reduce network latency and network costs.

Create a Cloud Storage bucket

You need a Cloud Storage bucket to store the data you use to train your model and the training results. The `ctpu up` tool used in this tutorial sets up default permissions for the Cloud TPU service account. If you want finer-grain permissions, review the access level permissions (https://cloud.google.com/tpu/docs/storage-buckets).

The bucket location must be in the same region as your virtual machine (VM) and your TPU node. VMs and TPU nodes are located in specific zones (https://cloud.google.com/tpu/docs/types-zones#types), which are subdivisions within a region.

1. Go to the Cloud Storage page on the Cloud Console.

   GO TO THE CLOUD STORAGE PAGE (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/STORAGE/BROWSER

2. Create a new bucket, specifying the following options:

- A unique name of your choosing.
- Default storage class: `Standard`
- Location: Specify a bucket location in the same region where you plan to create your TPU node. See TPU types and zones (https://cloud.google.com/tpu/docs/types-zones#types) to learn where various TPU types are available.

Use the `ctpu` tool

This section demonstrates using the Cloud TPU provisioning tool (https://github.com/tensorflow/tpu/tree/master/tools/ctpu) (`ctpu`) for creating and managing Cloud TPU project resources. The resources are comprised of a virtual machine (VM) and a Cloud TPU resource that have the same name. **These resources must reside in the same region/zone as the bucket you just created.**

You can also set up your VM and TPU resources using `gcloud` commands or through the Cloud Console (https://console.cloud.google.com/). See the creating and deleting TPUs (https://cloud.google.com/tpu/docs/creating-deleting-tpus) page to learn all the ways you can set up and manage your Compute Engine VM and Cloud TPU resources.

Run `ctpu up` to create resources

1. Open a Cloud Shell window.

   OPEN CLOUD SHELL (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/?CLOUDSHELL=TRUE)

2. Run `gcloud config set project <Your-Project>` to use the project where you want to create Cloud TPU.
3. Run `ctpu up` specifying the flags shown for either a Cloud TPU device or Pod slice. Refer to CTPU Reference (https://cloud.google.com/tpu/docs/ctpu-reference) for flag options and descriptions.
4. Set up a Cloud TPU device:

```
$ ctpu up
```

★ **Note:** If you have more than one project, you must specify the project name.

The following configuration message appears:

```
ctpu will use the following configuration:

Name: [your TPU's name]
Zone: [your project's zone]
GCP Project: [your project's name]
TensorFlow Version: 1.14
VM:
 Machine Type: [your machine type]
 Disk Size: [your disk size]
 Preemptible: [true or false]
Cloud TPU:
 Size: [your TPU size]
 Preemptible: [true or false]

OK to create your Cloud TPU resources with the above configuration? [Yn]:
```

Press `y` to create your Cloud TPU resources.

Note: The first time you run **`ctpu up`** on a project it takes about 5 minutes to perform startup tasks such as SSH key propagation and API turnup.

The **`ctpu up`** command creates a virtual machine (VM) and Cloud TPU services.

From this point on, a prefix of **`(vm)$`** means you should run the command on the Compute Engine VM instance.

Verify your Compute Engine VM

When the **`ctpu up`** command has finished executing, verify that your shell prompt has changed from **`username@project`** to **`username@tpuname`**. This change shows that you are now logged into your Compute Engine VM.

Install pandas

Install or upgrade pandas by typing the following command:

```
pip install pandas
```

Additional TensorFlow Concepts

In addition, you should be familiar with the following TensorFlow concepts:

- **Estimators.** For more information, see the TensorFlow (https://www.tensorflow.org/guide/premade_estimators) guide.
- **Custom estimators.** For more information on custom estimators, see the Create Custom Estimators (https://www.tensorflow.org/guide/custom_estimators) section of the TensorFlow guide.

Define Hyperparameters

In this code section you add several hyperparameters that TPU requires. You add these hyperparameters as flags, which allows you to change them at runtime.

The parameters you add are:

- **tpu.** This parameter identifies the name or IP address of the TPU node on which to run the model.
- **model_dir.** The path to save model checkpoints. This path must be a Cloud Storage bucket.
- **iterations.** The number of iterations per training loop.
- **use_tpu.** Specifies if you want to run the model on TPUs or GPU/CPUs, based on availability.

ESTIMATOR API | TPUESTIMATOR API

```
# Model specific parameters
tf.flags.DEFINE_integer("batch_size",
    default=50,
    help="Batch size.")
tf.flags.DEFINE_integer("train_steps",
    default=1000,
    help="Total number of training steps.")
FLAGS = tf.flags.FLAGS
```

Loading the data

This code section specifies how to read and load the data.

TPUs support the following data types:

- `tf.float32`
- `tf.complex64`
- `tf.int64`
- `tf.bool`
- `tf.bfloat64`

ESTIMATOR API | TPUESTIMATOR API

```
def load_data(y_name='Species'):
  """Returns the iris dataset as (train_x, train_y), (test_x, test_y)."""
  train_path, test_path = maybe_download()

  train = pd.read_csv(train_path, names=CSV_COLUMN_NAMES, header=0)
  train_x, train_y = train, train.pop(y_name)

  test = pd.read_csv(test_path, names=CSV_COLUMN_NAMES, header=0)
  test_x, test_y = test, test.pop(y_name)

  return (train_x, train_y), (test_x, test_y)
```

Define the input functions

A key difference between the Estimator API and the TPUEstimator API is the function signature of input functions. With the Estimator API, you can write input functions with any number of parameters. With the TPUEstimator API, input functions can take only a single parameter, `params`. This `params` has all the key-value pairs from the TPUEstimator (https://www.tensorflow.org/api_docs/python/tf/contrib/tpu/TPUEstimator) object, along with extra keys like `batch_size`.

One way to address this difference is to use lambda functions when calling the input functions. With lambda functions, you need to make only minor changes to your existing input functions.

The following sections demonstrate how to update your input functions. Later, you will see how to use lambda functions to convert these input functions to work with the TPUEstimator API.

Training input function

With the TPUEstimator API, your training input function, `train_input_fn`, must return a number of input samples that can be sharded by the number of Cloud TPU cores. For example, if you are using 8 cores, each batch size must be divisible by 8.

To accomplish this, the preceding code uses the `dataset.batch(batch_size, drop_remainder=True)` function. This function batches using the `batch_size` parameter and discards the remainder.

ESTIMATOR API | TPUESTIMATOR API

```
def train_input_fn(features, labels, batch_size):
  """An input function for training"""

  # Convert the inputs to a Dataset.
  dataset = tf.data.Dataset.from_tensor_slices((dict(features), labels))

  # Shuffle, repeat, and batch the examples.
  dataset = dataset.shuffle(1000).repeat().batch(batch_size)

  # Return the dataset.
  return dataset
```

Evaluation input function

In this step, you update the evaluation input function, `eval_input_fn`, to ensure that the input samples can be sharded by the number of TPU cores. To accomplish this, use the `dataset.batch(batch_size, drop_remainder=True)` function.

ESTIMATOR API | TPUESTIMATOR API

```
def eval_input_fn(features, labels, batch_size):
  """An input function for evaluation or prediction"""
  features=dict(features)
  if labels is None:
      # No labels, use only features.
```

```
    inputs = features
else:
    inputs = (features, labels)

# Convert the inputs to a Dataset.
dataset = tf.data.Dataset.from_tensor_slices(inputs)

# Batch the examples
assert batch_size is not None, "batch_size must not be None"
dataset = dataset.batch(batch_size)

# Return the dataset.
return dataset
```

Prediction input function

For predictions in TPUEstimators, the input dataset must have tensors with the additional outer dimension of `batch_size`. As a result, you must add a prediction input function, which takes `features` and `batch_size` as parameters. This function allows you to have fewer input samples than `batch_size`.

A prediction input function is optional if you are using the Estimator API.

ESTIMATOR API | TPUESTIMATOR API

A prediction input function is optional for the Estimator API, because the evaluation function, eval_input_fn performs this task.

Update the custom model function

Your next task is to update the custom model function:

- Replace instances of `tf.estimator.EstimatorSpec` to use `tf.contrib.tpu.TPUEstimatorSpec`.
- Remove any instances of `tf.summary`. The TPUEstimator API does not support custom summaries for tensorboard. However, basic summaries are automatically recorded to event files in the model directory.

- Wrap the optimizer using `tf.contrib.tpu.CrossShardOptimizer`. The `CrossShardOptimizer` uses an `allreduce` to aggregate gradients and broadcast the result to each shard. As the `CrossShardOptimizer` is not compatible with local training, you must also check for the `use_tpu` flag.

ESTIMATOR API | TPUESTIMATOR API

```
def my_model(features, labels, mode, params):
  """DNN with three hidden layers, and dropout of 0.1 probability."""

  # Create three fully connected layers each layer having a dropout
  # probability of 0.1.
  net = tf.feature_column.input_layer(features, params['feature_columns'])
  for units in params['hidden_units']:
      net = tf.layers.dense(net, units=units, activation=tf.nn.relu)

  # Compute logits (1 per class).
  logits = tf.layers.dense(net, params['n_classes'], activation=None)

  # Compute predictions.
  predicted_classes = tf.argmax(logits, 1)
  if mode == tf.estimator.ModeKeys.PREDICT:
      predictions = {
          'class_ids': predicted_classes[:, tf.newaxis],
          'probabilities': tf.nn.softmax(logits),
          'logits': logits,
      }
      return tf.estimator.EstimatorSpec(mode, predictions=predictions)

  # Compute loss.
  loss = tf.losses.sparse_softmax_cross_entropy(labels=labels,
                                                logits=logits)

  # Compute evaluation metrics.
  accuracy = tf.metrics.accuracy(labels=labels,
                                 predictions=predicted_classes,
                                 name='acc_op')
  metrics = {'accuracy': accuracy}
  tf.summary.scalar('accuracy', accuracy[1])
  if mode == tf.estimator.ModeKeys.EVAL:
      return tf.estimator.EstimatorSpec(
          mode, loss=loss, eval_metric_ops=metrics)

  # Create training op.
```

```
  if mode == tf.estimator.ModeKeys.TRAIN
      optimizer = tf.train.AdagradOptimizer(learning_rate=0.1)
      train_op = optimizer.minimize(loss, global_step=tf.train.get_global_step())
      return tf.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)
```

Add an evaluation metric function

Another difference between the Estimator API and the TPUEstimator API is how they handle metrics. With the Estimator API, you can pass metrics as a normal dictionary. For the TPUEstimator API, you must use a function instead.

ESTIMATOR API | TPUESTIMATOR API

Optional. The `my_model` function generates the metrics.

Update the main function

Configure TPUs

In this step, you configure the TPU cluster.

To configure the cluster, you can use the values assigned to the hyperparameters. See Define Hyperparameters (#defining_hyperparameters) for more information. In addition, you must set the following values:

- `allow_soft_placement`. When set to `true`, this parameter allows TensorFlow to use a GPU device if a TPU is unavailable. If a GPU device is also unavailable, a CPU device is used.
- `log_device_placement`. Indicates that TensorFlow should log device placements.

ESTIMATOR API | TPUESTIMATOR API

Not required, as this code section only affects TPUs.

Add TPU-specific parameters to the classifier

In this section of the code, you update the classifier variable to use the TPUEstimator class. This change requires that you add the following parameters:

- `use_tpu`
- `train_batch_size`
- `eval_batch_size`
- `predict_batch_size`
- `config`

ESTIMATOR API TPUESTIMATOR API

```
# Build 2 hidden layer DNN with 10, 10 units respectively.
classifier = tf.estimator.Estimator(
    model_fn=my_model,
    params={
        'feature_columns': my_feature_columns,
        # Two hidden layers of 10 nodes each.
        'hidden_units': [10, 10],
        # The model must choose between 3 classes.
        'n_classes': 3,
    })
```

Call the train method

The next change is to update the train method. Notice the use of a lambda function to call the `train_input_fn` function. This methodology makes it easier to use your existing functions with the TPUEstimator API.

In addition, you must change the steps parameter to `max_steps`. In the next section, you'll repurpose the steps parameter to specify the number of evaluation steps.

ESTIMATOR API TPUESTIMATOR API

```
# Train the Model.
classifier.train(
    input_fn=lambda:iris_data.train_input_fn(
        train_x, train_y, FLAGS.batch_size),
    steps=FLAGS.train_steps)
```

Call the evaluation method

This change is similar to the one you made to the train method. Again, the use of a lambda function makes it easier to use an existing evaluation input function.

In addition, you must change the `steps` parameter to the value set from the `eval_steps` command line flag.

ESTIMATOR API | TPUESTIMATOR API

```
# Evaluate the model.
eval_result = classifier.evaluate(
    input_fn=lambda:iris_data.eval_input_fn(
        test_x, test_y, FLAGS.batch_size))

print('\nTest set accuracy: {accuracy:0.3f}\n'.format(**eval_result))
```

Call the predict method

As with the train and evaluate methods, you must update the predict method. Again, the use of a lambda function makes it easier to use an existing evaluation input function.

ESTIMATOR API | TPUESTIMATOR API

```
# Generate predictions from the model
predictions = classifier.predict(
    input_fn=lambda: iris_data.eval_input_fn(
        iris_data.PREDICTION_INPUT_DATA,
        labels=None,
        batch_size=FLAGS.batch_size))

for pred_dict, expec in zip(predictions, iris_data.PREDICTION_OUTPUT_DATA):
    template = ('\nPrediction is "{}" ({:.1f}%), expected "{}"')

    class_id = pred_dict['class_ids'][0]
    probability = pred_dict['probabilities'][class_id]

    print(template.format(iris_data.SPECIES[class_id],
                          100 * probability, expec))
```

Clean up

To avoid incurring charges to your GCP account for the resources used in this topic:

1. Disconnect from the Compute Engine VM:

   ```
   (vm)$ exit
   ```

   Your prompt should now be `user@projectname`, showing you are in the Cloud Shell.

2. In your Cloud Shell, run `ctpu delete` with the –zone flag you used when you set up the Cloud TPU to delete your Compute Engine VM and your Cloud TPU:

   ```
   $ ctpu delete [optional: --zone]
   ```

   ★ **Important:** If you set the TPU resources name when you ran `ctpu up`, you must specify that name with the `--name` flag when you run `ctpu delete` in order to shut down your TPU resources.

3. Run `ctpu status` to make sure you have no instances allocated to avoid unnecessary charges for TPU usage. The deletion might take several minutes. A response like the one below indicates there are no more allocated instances:

   ```
   2018/04/28 16:16:23 WARNING: Setting zone to "us-central1-b"
   No instances currently exist.
           Compute Engine VM:     --
           Cloud TPU:             --
   ```

4. Run `gsutil` as shown, replacing `YOUR-BUCKET-NAME` with the name of the Cloud Storage bucket you created for this tutorial:

   ```
   $ gsutil rm -r gs://YOUR-BUCKET-NAME
   ```

Note: For free storage limits and other pricing information, see the Cloud Storage pricing guide (https://cloud.google.com/storage/pricing).

What's Next?

To learn more about the Estimator and TPUEstimator APIs, see the following topics:

- Estimator API
  - Premade Estimators (https://www.tensorflow.org/guide/premade_estimators)
  - Creating Custom Estimators (https://www.tensorflow.org/guide/custom_estimators)
  - `iris_data.py` (https://github.com/tensorflow/models/blob/master/samples/core/get_started/iris_data.py)
  - `custom_estimator.py` (https://github.com/tensorflow/models/blob/master/samples/core/get_started/custom_estimator.py)
- TPUEstimator API
  - `iris_data_tpu.ipynb` (https://github.com/tensorflow/tpu/blob/master/models/samples/core/get_started/iris_data_tpu.py)
  - `custom_tpuestimator.ipynb` (https://github.com/tensorflow/tpu/blob/master/models/samples/core/get_started/custom_tpuestimator.py)
  - Using TPUs (https://www.tensorflow.org/guide/using_tpu)
  - Using the TPUEstimator API on Cloud TPUs (https://cloud.google.com/tpu/docs/using-estimator-api)

Last updated December 4, 2019.