adanet_tpu

April 24, 2019


```
In [0]: #@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.
```

1 AdaNet on TPU

<a target="_blank" href="https://colab.research.google.com/github/tensorflow/adanet/blob/maste

<a target="_blank" href="https://github.com/tensorflow/adanet/blob/master/adanet/examples/tuto

AdaNet supports training on Google's custom machine learning accelerators known as Tensor Processing Units (TPU). Conveniently, we provide `adanet.TPUEstimator` which handles TPU support behind the scenes. There are only a few minor changes needed to switch from `adanet.Estimator` to `adanet.TPUEstimator`. We highlight the necessary changes in this tutorial.

If the reader is not familiar with AdaNet, it is reccommended they take a look at The AdaNet Objective and in particular Customizing AdaNet as this tutorial builds upon the latter.

NOTE: you must provide a valid GCS bucket to use TPUEstimator.

To begin, we import the necessary packages, obtain the Colab's TPU master address, and give the TPU permissions to write to our GCS Bucket. Follow the instructions here to connect to a Colab TPU runtime.

```
In [0]: #@test {"skip": true}
        # If you're running this in Colab, first install the adanet package:
        !pip install adanet
```

```
In [0]: from __future__ import absolute_import
        from __future__ import division
```

```
from __future__ import print_function

import functools
import json
import os
import six
import time

import adanet
from google.colab import auth
import tensorflow as tf

BUCKET = ''  #@param {type: 'string'}
MODEL_DIR = 'gs://{}/{}'.format(
    BUCKET, time.strftime('adanet-tpu-estimator/%Y-%m-%d-%H-%M-%S'))

MASTER = ''
if 'COLAB_TPU_ADDR' in os.environ:
  auth.authenticate_user()

  MASTER = 'grpc://' + os.environ['COLAB_TPU_ADDR']

  # Authenticate TPU to use GCS Bucket.
  with tf.Session(MASTER) as sess:
    with open('/content/adc.json', 'r') as file_:
      auth_info = json.load(file_)
    tf.contrib.cloud.configure_gcs(sess, credentials=auth_info)


# The random seed to use.
RANDOM_SEED = 42
```

1.1 Fashion MNIST

We focus again on the Fashion MNIST dataset and download the data via Keras.

```
In [0]: (x_train, y_train), (x_test, y_test) = (
            tf.keras.datasets.fashion_mnist.load_data())
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-
32768/29515 [=================================] - 0s 0us/step
40960/29515 [==========================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-
26427392/26421880 [==============================] - 0s 0us/step
26435584/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-
16384/5148 [==================================================================================
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-
```

```
4423680/4422102 [==============================] - 0s 0us/step
4431872/4422102 [==============================] - 0s 0us/step
```

1.2 input_fn Changes

There are two minor changes we must make to `input_fn` to support running on TPU:

1. TPUs dynamically shard the input data depending on the number of cores used. Because of this, we augment `input_fn` to take a dictionary `params` argument. When running on TPU, `params` contains a `batch_size` field with the appropriate batch size.
2. Once the input is batched, we drop the last batch if it is smaller than `batch_size`. This can simply be done by specifying `drop_remainder=True` to the `tf.data.DataSet.batch()` function. It is important to specify this option since TPUs do not support dynamic shapes. Note that we only drop the remainder batch during training since evaluation is still done on the CPU.

In [0]:
```
FEATURES_KEY = "images"


def generator(images, labels):
  """Returns a generator that returns image-label pairs."""

  def _gen():
    for image, label in zip(images, labels):
      yield image, label

  return _gen


def preprocess_image(image, label):
  """Preprocesses an image for an `Estimator`."""
  image = image / 255.
  image = tf.reshape(image, [28, 28, 1])
  features = {FEATURES_KEY: image}
  return features, label


def input_fn(partition, training, batch_size):
  """Generate an input_fn for the Estimator."""

  def _input_fn(params):  # TPU: specify `params` argument.

    # TPU: get the TPU set `batch_size`, if available.
    batch_size_ = params.get("batch_size", batch_size)

    if partition == "train":
      dataset = tf.data.Dataset.from_generator(
```

```
        generator(x_train, y_train), (tf.float32, tf.int32), ((28, 28), ()))
  elif partition == "predict":
    dataset = tf.data.Dataset.from_generator(
        generator(x_test[:10], y_test[:10]), (tf.float32, tf.int32),
        ((28, 28), ()))
  else:
    dataset = tf.data.Dataset.from_generator(
        generator(x_test, y_test), (tf.float32, tf.int32), ((28, 28), ()))

  if training:
    dataset = dataset.shuffle(10 * batch_size_, seed=RANDOM_SEED).repeat()

  # TPU: drop the remainder batch when training on TPU.
  dataset = dataset.map(preprocess_image).batch(
      batch_size_, drop_remainder=training)
  iterator = dataset.make_one_shot_iterator()
  features, labels = iterator.get_next()
  return features, labels

return _input_fn
```

1.3 `model_fn` Changes

We use a similar CNN architecture as used in the Customizing AdaNet tutorial. The only TPU specific change we need to make is wrap the optimizer in a `tf.contrib.tpu.CrossShardOptimizer`.

```
In [0]: #@title Define the Builder and Generator
        class SimpleCNNBuilder(adanet.subnetwork.Builder):
          """Builds a CNN subnetwork for AdaNet."""

          def __init__(self, learning_rate, max_iteration_steps, seed):
            """Initializes a `SimpleCNNBuilder`.

            Args:
              learning_rate: The float learning rate to use.
              max_iteration_steps: The number of steps per iteration.
              seed: The random seed.

            Returns:
              An instance of `SimpleCNNBuilder`.
            """
            self._learning_rate = learning_rate
            self._max_iteration_steps = max_iteration_steps
            self._seed = seed

          def build_subnetwork(self,
                               features,
                               logits_dimension,
```

```python
                           training,
                           iteration_step,
                           summary,
                           previous_ensemble=None):
    """See `adanet.subnetwork.Builder`."""
    images = list(features.values())[0]
    kernel_initializer = tf.keras.initializers.he_normal(seed=self._seed)
    x = tf.keras.layers.Conv2D(
        filters=16,
        kernel_size=3,
        padding="same",
        activation="relu",
        kernel_initializer=kernel_initializer)(
            images)
    x = tf.keras.layers.MaxPool2D(pool_size=2, strides=2)(x)
    x = tf.keras.layers.Flatten()(x)
    x = tf.keras.layers.Dense(
        units=64, activation="relu", kernel_initializer=kernel_initializer)(
            x)

    logits = tf.keras.layers.Dense(
        units=10, activation=None, kernel_initializer=kernel_initializer)(
            x)

    complexity = tf.constant(1)

    return adanet.Subnetwork(
        last_layer=x,
        logits=logits,
        complexity=complexity,
        persisted_tensors={})

  def build_subnetwork_train_op(self,
                                subnetwork,
                                loss,
                                var_list,
                                labels,
                                iteration_step,
                                summary,
                                previous_ensemble=None):
    """See `adanet.subnetwork.Builder`."""

    learning_rate = tf.train.cosine_decay(
        learning_rate=self._learning_rate,
        global_step=iteration_step,
        decay_steps=self._max_iteration_steps)
    optimizer = tf.train.MomentumOptimizer(learning_rate, .9)
    # TPU: wrap the optimizer in a CrossShardOptimizer.
```

```python
    optimizer = tf.contrib.tpu.CrossShardOptimizer(optimizer)
    return optimizer.minimize(loss=loss, var_list=var_list)

  def build_mixture_weights_train_op(self, loss, var_list, logits, labels,
                                     iteration_step, summary):
    """See `adanet.subnetwork.Builder`."""
    return tf.no_op("mixture_weights_train_op")

  @property
  def name(self):
    """See `adanet.subnetwork.Builder`."""
    return "simple_cnn"


class SimpleCNNGenerator(adanet.subnetwork.Generator):
  """Generates a `SimpleCNN` at each iteration."""

  def __init__(self, learning_rate, max_iteration_steps, seed=None):
    """Initializes a `Generator` that builds `SimpleCNNs`.

    Args:
      learning_rate: The float learning rate to use.
      max_iteration_steps: The number of steps per iteration.
      seed: The random seed.

    Returns:
      An instance of `Generator`.
    """
    self._seed = seed
    self._dnn_builder_fn = functools.partial(
        SimpleCNNBuilder,
        learning_rate=learning_rate,
        max_iteration_steps=max_iteration_steps)

  def generate_candidates(self, previous_ensemble, iteration_number,
                          previous_ensemble_reports, all_reports):
    """See `adanet.subnetwork.Generator`."""
    seed = self._seed
    # Change the seed according to the iteration so that each subnetwork
    # learns something different.
    if seed is not None:
      seed += iteration_number
    return [self._dnn_builder_fn(seed=seed)]
```

1.4 Launch TensorBoard

Let's run TensorBoard to visualize model training over time. We'll use ngrok to tunnel traffic to localhost.

The instructions for setting up Tensorboard were obtained from https://www.dlology.com/blog/quick-guide-to-run-tensorboard-in-google-colab/

Run the next cells and follow the link to see the TensorBoard in a new tab.

```
In [0]: #@test {"skip": true}

        get_ipython().system_raw(
            'tensorboard --logdir {} --host 0.0.0.0 --port 6006 &'
            .format(MODEL_DIR)
        )

        # Install ngrok binary.
        ! wget https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.zip
        ! unzip ngrok-stable-linux-amd64.zip

        print("Follow this link to open TensorBoard in a new tab.")
        get_ipython().system_raw('./ngrok http 6006 &')
        ! curl -s http://localhost:4040/api/tunnels | python3 -c \
            "import sys, json; print(json.load(sys.stdin)['tunnels'][0]['public_url'])"
```

1.5 Using `adanet.TPUEstimator` to Train and Evaluate

Finally, we switch from `adanet.Estimator` to `adanet.TPUEstimator`. There are two last changes needed:

1. Update the `RunConfig` to be a `tf.contrib.tpu.RunConfig`. We supply the TPU `master` address and set `iterations_per_loop=200`. This choice is fairly arbitrary in our case. A good practice is to set it to the number of steps in between summary writes and metric evals.
2. Finally, we specify the `use_tpu` and `batch_size` parameters `adanet.TPUEstimator`.

There is an important thing to note about the `batch_size`: each TPU chip consists of 2 cores with 4 shards each. In the Customizing AdaNet tutorial, a `batch_size` of 64 was used. To be consistent we use the same `batch_size` per shard and drop the number of training steps accordingly. In other words, since we're running on one TPU we set `batch_size=64*8=512` and `train_steps=1000`. In the ideal case, since we drop the `train_steps` by 5x, this means we're **training 5x faster!**

```
In [0]: #@title AdaNet Parameters
        LEARNING_RATE = 0.25  #@param {type:"number"}
        TRAIN_STEPS = 1000  #@param {type:"integer"}
        BATCH_SIZE = 512  #@param {type:"integer"}
        ADANET_ITERATIONS = 2  #@param {type:"integer"}

        # TPU: switch `tf.estimator.RunConfig` to `tf.contrib.tpu.RunConfig`.
        # The main required changes are specifying `tpu_config` and `master`.
        config = tf.contrib.tpu.RunConfig(
            tpu_config=tf.contrib.tpu.TPUConfig(iterations_per_loop=200),
            master=MASTER,
```

```
    save_checkpoints_steps=200,
    save_summary_steps=200,
    tf_random_seed=RANDOM_SEED)

head = tf.contrib.estimator.multi_class_head(
    n_classes=10, loss_reduction=tf.losses.Reduction.SUM_OVER_BATCH_SIZE)
max_iteration_steps = TRAIN_STEPS // ADANET_ITERATIONS
# TPU: switch `adanet.Estimator` to `adanet.TPUEstimator`.
try:
  estimator = adanet.TPUEstimator(
      head=head,
      subnetwork_generator=SimpleCNNGenerator(
          learning_rate=LEARNING_RATE,
          max_iteration_steps=max_iteration_steps,
          seed=RANDOM_SEED),
      max_iteration_steps=max_iteration_steps,
      evaluator=adanet.Evaluator(
          input_fn=input_fn("train", training=False, batch_size=BATCH_SIZE),
          steps=None),
      adanet_loss_decay=.99,
      config=config,
      model_dir=MODEL_DIR,
      # TPU: specify `use_tpu` and the batch_size parameters.
      use_tpu=True,
      train_batch_size=BATCH_SIZE,
      eval_batch_size=32)
except tf.errors.InvalidArgumentError as e:
  six.raise_from(
      Exception(
          "Invalid GCS Bucket: you must provide a valid GCS bucket in the "
          "`BUCKET` form field of the first cell."), e)

results, _ = tf.estimator.train_and_evaluate(
    estimator,
    train_spec=tf.estimator.TrainSpec(
        input_fn=input_fn("train", training=True, batch_size=BATCH_SIZE),
        max_steps=TRAIN_STEPS),
    eval_spec=tf.estimator.EvalSpec(
        input_fn=input_fn("test", training=False, batch_size=BATCH_SIZE),
        steps=None,
        start_delay_secs=1,
        throttle_secs=1,
    ))

print("Accuracy:", results["accuracy"])
print("Loss:", results["average_loss"])
```

```
Accuracy: 0.8913
```

```
Loss: 0.298405
```

1.6 Conclusion

That was easy! With very few changes we were able to transform our original estimator into one which can harness the power of TPUs.