tf.contrib.tpu.TPUEstimator

Contents

Class **`TPUEstimator`**

Inherits From: `Estimator`

Defined in `tensorflow/contrib/tpu/python/tpu/tpu_estimator.py`.

Estimator with TPU support.

TPUEstimator handles many of the details of running on TPU devices, such as replicating inputs and models for each core, and returning to host periodically to run hooks.

If `use_tpu` is false, all training, evaluation, and predict are executed on CPU.

For training, TPUEstimator transforms a global batch size in params to a per-shard batch size when calling the `input_fn` and `model_fn`. Users should specify `train_batch_size` in constructor, and then get the batch size for each shard in `input_fn` and `model_fn` by `params['batch_size']`. If `TPUConfig.per_host_input_for_training` is `True`, `input_fn` is invoked per host rather than per shard. In this case, a global batch size is transformed a per-host batch size in params for `input_fn`, but `model_fn` still gets per-shard batch size.

For evaluation, if `eval_batch_size` is None, it is executed on CPU, even if `use_tpu` is `True`. If `eval_batch_size` is not `None`, it is executed on TPU, which is an experimental feature. In this case, `model_fn` should return `TPUEstimatorSpec` instead of `EstimatorSpec`, which expects the `eval_metrics` for TPU evaluation.

`TPUEstimatorSpec.eval_metrics` is a tuple of `metric_fn` and `tensors`, where `tensors` could be a list of `Tensor`s or dict of names to `Tensor`s. (See `TPUEstimatorSpec` for details). `metric_fn` takes the `tensors` and returns a dict from metric string name to the result of calling a metric function, namely a `(metric_tensor, update_op)` tuple.

Current limitations:

1. TPU evaluation only works on single host.
2. `input_fn` for evaluation should not throw OutOfRange error for all evaluation steps and all batches should have the same size.

Example (MNIST):

```
# The metric Fn which runs on CPU.
def metric_fn(labels, logits):
  predictions = tf.argmax(logits, 1)
  return {
    'accuracy': tf.metrics.precision(
        labels=labels, predictions=predictions),
  }

# Your model Fn which runs on TPU (eval_metrics is list in this example)
def model_fn(features, labels, mode, config, params):
  ...
  logits = ...

  if mode = tf.estimator.ModeKeys.EVAL:
    return tpu_estimator.TPUEstimatorSpec(
        mode=mode,
        loss=loss,
        eval_metrics=(metric_fn, [labels, logits]))

# or specify the eval_metrics tensors as dict.
def model_fn(features, labels, mode, config, params):
  ...
  final_layer_output = ...

  if mode = tf.estimator.ModeKeys.EVAL:
    return tpu_estimator.TPUEstimatorSpec(
        mode=mode,
        loss=loss,
        eval_metrics=(metric_fn, {
            'labels': labels,
            'logits': final_layer_output,
        }))
```

Predict support on TPU is not yet implemented. So, `predict` and `export_savedmodel` are executed on CPU, even if `use_tpu` is true.

Properties

`config`

`model_dir`

`model_fn`

Returns the model_fn which is bound to self.params.

Returns:

The model_fn with following signature: `def model_fn(features, labels, mode, config)`

`params`

Methods

`__init__`

```
__init__(
    model_fn=None,
    model_dir=None,
    config=None,
    params=None,
    use_tpu=True,
    train_batch_size=None,
    eval_batch_size=None,
    batch_axis=None
)
```

Constructs an `TPUEstimator` instance.

Args:

- `model_fn`: Model function as required by `Estimator`. For training, the returned `EstimatorSpec` cannot have hooks as it is not supported in `TPUEstimator`.
- `model_dir`: Directory to save model parameters, graph and etc. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model. If `None`, the model_dir in `config` will be used if set. If both are set, they must be same. If both are `None`, a temporary directory will be used.
- `config`: An `tpu_config.RunConfig` configuration object. Cannot be `None`.
- `params`: An optional `dict` of hyper parameters that will be passed into `input_fn` and `model_fn`. Keys are names of parameters, values are basic python types. There are reserved keys for `TPUEstimator`, including 'batch_size'.
- `use_tpu`: A bool indicating whether TPU support is enabled. Currently,
  - TPU training respects this bit.
  - If true, see `eval_batch_size` for evaluate support.
  - Predict still happens on CPU.
- `train_batch_size`: An int representing the global training batch size. TPUEstimator transforms this global batch size to a per-shard batch size, as params['batch_size'], when calling `input_fn` and `model_fn`. Cannot be `None` if `use_tpu` is `True`. Must be divisible by `config.tpu_config.num_shards`.
- `eval_batch_size`: An int representing the global training batch size. Currently, if `None`, evaluation is still executed on CPU (even when `use_tpu` is True). In near future, `use_tpu` will be the only option to switch between TPU/CPU evaluation.
- `batch_axis`: A python tuple of int values describing how each tensor produced by the Estimator `input_fn` should be split across the TPU compute shards. For example, if your input_fn produced (images, labels) where the images tensor is in `HWCN` format, your shard dimensions would be [3, 0], where 3 corresponds to the `N` dimension of your images Tensor, and 0 corresponds to the dimension along which to split the labels to match up with the corresponding images. If None is supplied, and per_host_input_for_training is True, batches will be sharded based on the major dimension. If tpu_config.per_host_input_for_training is False, batch_axis is ignored.

Raises:

- `ValueError`: `params` has reserved keys already.

`evaluate`

```
evaluate(
    input_fn,
    steps=None,
    hooks=None,
    checkpoint_path=None,
    name=None
)
```

Evaluates the model given evaluation data input_fn.

For each step, calls `input_fn`, which returns one batch of data. Evaluates until: - `steps` batches are processed, or - `input_fn` raises an end-of-input exception (`OutOfRangeError` or `StopIteration`).

Args:

- `input_fn`: Input function returning a tuple of: features - Dictionary of string feature name to `Tensor` or `SparseTensor`. labels - `Tensor` or dictionary of `Tensor` with labels.
- `steps`: Number of steps for which to evaluate model. If `None`, evaluates until `input_fn` raises an end-of-input exception.
- `hooks`: List of `SessionRunHook` subclass instances. Used for callbacks inside the evaluation call.
- `checkpoint_path`: Path of a specific checkpoint to evaluate. If `None`, the latest checkpoint in `model_dir` is used.
- `name`: Name of the evaluation if user needs to run multiple evaluations on different data sets, such as on training data vs test data. Metrics for different evaluations are saved in separate folders, and appear separately in tensorboard.

Returns:

A dict containing the evaluation metrics specified in `model_fn` keyed by name, as well as an entry `global_step` which contains the value of the global step for which this evaluation was performed.

Raises:

- `ValueError`: If `steps <= 0`.
- `ValueError`: If no model has been trained, namely `model_dir`, or the given `checkpoint_path` is empty.

`export_savedmodel`

```
export_savedmodel(
    export_dir_base,
    serving_input_receiver_fn,
    assets_extra=None,
    as_text=False,
    checkpoint_path=None
)
```

Exports inference graph as a SavedModel into given dir.

This method builds a new graph by first calling the serving_input_receiver_fn to obtain feature `Tensor`s, and then calling this `Estimator`'s model_fn to generate the model graph based on those features. It restores the given checkpoint (or, lacking that, the most recent checkpoint) into this graph in a fresh session. Finally it creates a timestamped export directory below the given export_dir_base, and writes a `SavedModel` into it containing a single `MetaGraphDef` saved from this session.

The exported `MetaGraphDef` will provide one `SignatureDef` for each element of the export_outputs dict returned from the model_fn, named using the same keys. One of these keys is always signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY, indicating which signature will be served when a serving request does not specify one. For each signature, the outputs are provided by the corresponding `ExportOutput`s, and the inputs are always the input receivers provided by the serving_input_receiver_fn.

Extra assets may be written into the SavedModel via the extra_assets argument. This should be a dict, where each key gives a destination path (including the filename) relative to the assets.extra directory. The corresponding value gives the full path of the source file to be copied. For example, the simple case of copying a single file without renaming it is specified as `{'my_asset_file.txt': '/path/to/my_asset_file.txt'}`.

Args:

- `export_dir_base`: A string containing a directory in which to create timestamped subdirectories containing exported SavedModels.
- `serving_input_receiver_fn`: A function that takes no argument and returns a `ServingInputReceiver`.
- `assets_extra`: A dict specifying how to populate the assets.extra directory within the exported SavedModel, or `None` if no extra assets are needed.
- `as_text`: whether to write the SavedModel proto in text format.
- `checkpoint_path`: The checkpoint path to export. If `None` (the default), the most recent checkpoint found within the model directory is chosen.

Returns:

The string path to the exported directory.

Raises:

- `ValueError`: if no serving_input_receiver_fn is provided, no export_outputs are provided, or no checkpoint can be found.

`get_variable_names`

```
get_variable_names()
```

Returns list of all variable names in this model.

Returns:

List of names.

Raises:

- `ValueError`: If the Estimator has not produced a checkpoint yet.

`get_variable_value`

```
get_variable_value(name)
```

Returns value of the variable given by name.

Args:

- `name`: string or a list of string, name of the tensor.

Returns:

Numpy array - value of the tensor.

Raises:

- `ValueError`: If the Estimator has not produced a checkpoint yet.

`latest_checkpoint`

```
latest_checkpoint()
```

Finds the filename of latest saved checkpoint file in `model_dir`.

Returns:

The full path to the latest checkpoint or `None` if no checkpoint was found.

`predict`

```
predict(
    input_fn,
    predict_keys=None,
    hooks=None,
    checkpoint_path=None
)
```

Yields predictions for given features.

Args:

- `input_fn`: Input function returning features which is a dictionary of string feature name to `Tensor` or `SparseTensor`. If it returns a tuple, first item is extracted as features. Prediction continues until `input_fn` raises an end-of-input exception (`OutOfRangeError` or `StopIteration`).
- `predict_keys`: list of `str`, name of the keys to predict. It is used if the `EstimatorSpec.predictions` is a `dict`. If `predict_keys` is used then rest of the predictions will be filtered from the dictionary. If `None`, returns all.
- `hooks`: List of `SessionRunHook` subclass instances. Used for callbacks inside the prediction call.
- `checkpoint_path`: Path of a specific checkpoint to predict. If `None`, the latest checkpoint in `model_dir` is used.

Yields:

Evaluated values of `predictions` tensors.

Raises:

- `ValueError`: Could not find a trained model in model_dir.
- `ValueError`: if batch length of predictions are not same.

- `ValueError`: If there is a conflict between `predict_keys` and `predictions`. For example if `predict_keys` is not `None` but `EstimatorSpec.predictions` is not a `dict`.

train

```
train(
    input_fn,
    hooks=None,
    steps=None,
    max_steps=None,
    saving_listeners=None
)
```

Trains a model given training data input_fn.

Args:

- `input_fn`: Input function returning a tuple of: features - `Tensor` or dictionary of string feature name to `Tensor`. labels - `Tensor` or dictionary of `Tensor` with labels.
- `hooks`: List of `SessionRunHook` subclass instances. Used for callbacks inside the training loop.
- `steps`: Number of steps for which to train model. If `None`, train forever or train until input_fn generates the `OutOfRange` error or `StopIteration` exception. 'steps' works incrementally. If you call two times train(steps=10) then training occurs in total 20 steps. If `OutOfRange` or `StopIteration` occurs in the middle, training stops before 20 steps. If you don't want to have incremental behavior please set `max_steps` instead. If set, `max_steps` must be `None`.
- `max_steps`: Number of total steps for which to train model. If `None`, train forever or train until input_fn generates the `OutOfRange` error or `StopIteration` exception. If set, `steps` must be `None`. If `OutOfRange` or `StopIteration` occurs in the middle, training stops before `max_steps` steps. Two calls to `train(steps=100)` means 200 training iterations. On the other hand, two calls to `train(max_steps=100)` means that the second call will not do any iteration since first call did all 100 steps.
- `saving_listeners`: list of `CheckpointSaverListener` objects. Used for callbacks that run immediately before or after checkpoint savings.

Returns:

`self`, for chaining.

Raises:

- `ValueError`: If both `steps` and `max_steps` are not `None`.
- `ValueError`: If either `steps` or `max_steps` is <= 0.