# pylint: disable=too-many-lines

"""

``twml.trainers.Trainer`` is a wrapper around `tf.estimator.Estimator

<https://www.tensorflow.org/versions/master/api\_docs/python/tf/estimator/Estimator>`\_

to expose an easier to use API by

hiding rarely used config knobs and supplying default values.

The `Trainer` facilitates multi-phase training commonly used at Twitter: e.g.

MDL calibration -> MLP training -> Isotonic calibration.

The `Trainer` also facilitates hyperparameters tuning,

with its simple `add\_parser\_arguments()` method.

Learning rate decay functions

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Please note that we have four learning rate decay functions to choose from.

Additionally, each trainer can only take one learning rate decay function and its parameters.

If that is not the case, it will throw an error.

Also, please note that the learning rate decay is a positional argument and should be placed as

the last argument to the trainer, as you can see in the example above.

The four learning decays options are:

1. inverse\_learning\_rate\_decay:

The function returns the decayed learning rate. It is computed as:

::

decayed\_learning\_rate = learning\_rate / (1 + decay\_rate \* global\_step /decay\_step)

final\_decayed\_learning\_rate = max(decayed\_learning\_rate, min\_learning\_rate)

2. polynomial\_learning\_rate\_decay:

The function returns the decayed learning rate. It is computed as:

::

global\_step = min(global\_step, decay\_steps)

decayed\_learning\_rate = (learning\_rate - end\_learning\_rate) \*

(1 - global\_step / decay\_steps) ^ (power) +

end\_learning\_rate

3. piecewise\_constant\_learning\_rate\_decay:

Piecewise constant from boundaries and interval values.

Example: use a learning rate that's 1.0 for the first 100001 steps, 0.5 for

the next 10000 steps, and 0.1 for any additional steps.

::

global\_step = tf.Variable(0, trainable=False)

boundaries = [100000, 110000]

values = [1.0, 0.5, 0.1]

learning\_rate = tf.train.piecewise\_constant(global\_step, boundaries, values)

4. exponential\_learning\_rate\_decay:

The function returns the decayed learning rate. It is computed as:

::

decayed\_learning\_rate = learning\_rate \* decay\_rate ^ (global\_step / decay\_steps)

"""

import datetime

import functools

import math

from operator import itemgetter

import os

import pprint as pp

import random

from string import Template

import subprocess

import sys

import time

from threading import Thread

from twitter.common.metrics import AtomicGauge

from twitter.deepbird.stats\_server import utils as stats\_server\_utils

from twitter.deepbird.stats\_server.stats\_exporter import StatsExporter

from twitter.ml.common import metrics

from twitter.ml.common.kubernetes import kubectl\_delete\_by\_name, Resource

from twitter.ml.twml.status import get\_distributed\_training\_job\_status, TrainingJobStatus

from absl import logging

from twml.optimizers import LazyAdamOptimizer, optimize\_loss, OPTIMIZER\_SUMMARIES

from twml.contrib.optimizers import DeepGradientCompressionOptimizer

from twml.tracking import ExperimentTracker

from twml.util import (delete\_file\_or\_dir,

get\_distributed\_training\_job\_path,

sanitize\_hdfs\_path)

try:

from urllib import quote as encode\_url

except ImportError:

from urllib.parse import quote as encode\_url

import tensorflow.compat.v1 as tf

import tensorflow

import tensorflow\_hub as hub

import twitter.ml.twml.kubernetes.status as k8s\_status

import twml

import twml.export\_output\_fns

import twml.learning\_rate\_decay

import twml.metrics

\_CLUSTER\_TEMPLATE = Template('''{

"cluster": {

"ps": [$PS],

"chief": [$CHIEF],

"worker": [$WORKER]

},

"task": {"type": "$TYPE", "index": $INDEX}

}

''')

def init\_from\_checkpoint(init\_dir, init\_map):

"""

Wrapper around tf.train.init\_from\_checkpoint

"""

if init\_dir:

init\_dir = sanitize\_hdfs\_path(init\_dir)

tf.train.init\_from\_checkpoint(init\_dir, init\_map)

class Trainer(object):

"""

This class wraps ``tf.estimator.Estimator`` to make construction, saving, and loading easier.

Supports multi-phase training (for example, use a Trainer for MDL calibration, then

another for training the rest of the model, then another for isotonic calibration).

The Trainer also implements a training and evaluation loop via the ``learn()`` method.

Each Trainer is associated to a fixed set of hyper parameters (params), and a single model

specified by ``build\_graph``. Given these constraints, a single Trainer can be called

multiple times for training and evaluation over multiple epochs.

However, if you intend to try different sets of hyper-parameters, we recommend you instantiate

a different Trainer for each such experiment. That way, each experiment can be tracked

in a different ``save\_dir``. Indeed, after calling ``learn``, a Trainer's save\_dir will contain

checkpoints of the model (its graph, and variables), and the history of metrics (for example,

evaluation accuracy at each epoch), and other store observations like the average time per step.

The latter metrics can be viewed by pointing

TensorBoard to the save\_dir and accessing TensorBoard via your browser.

"""

def \_\_init\_\_(self, name, params, build\_graph\_fn,

metric\_fn=None,

optimize\_loss\_fn=None,

run\_config=None,

save\_dir=None,

init\_from\_dir=None,

init\_map=None,

warm\_start\_from=None,

profiler\_steps=None,

\*\*kwargs):

"""

Args:

name (String):

string name of this estimator; used as scope names for variables and tensors.

params (HParams, Namespace, or Dict):

hyper-parameters to be passed to Estimator constructor.

Must include params.train\_batch\_size and params.eval\_batch\_size.

Note that params is passed to twml.util.convert\_to\_hparams() to produce an HParams.

build\_graph\_fn:

A function for building tensorflow graphs.

This matches TensorFlow Estimator's model\_fn signature.

For example,

.. code-block:: python

def build\_graph(features, label, mode, params, config=None):

# Implements a simple binary logistic regression model

sparse\_tf = twml.util.convert\_to\_sparse(features, params.input\_size\_bits)

logits = twml.layers.full\_sparse(sparse\_tf, 1 << params.input\_size\_bits, 1)

if mode == 'infer':

loss = None

else:

loss = tf.nn.sigmoid\_cross\_entropy\_with\_logits(labels=label, logits=logits)

loss = twml.util.weighted\_average(loss, features['weights'])

output = tf.nn.sigmoid(logits)

return {'output': output, 'loss': loss}

Args:

features (dict of Tensor keyed by a string name):

input tensors.

mode (tf.estimator.ModeKeys / String):

one of 'train', 'eval', 'infer'.

label (Tensor):

if in ``mode == 'train'`` mode, these contain the corresponding labels for input.

params (HParams):

hyper parameters that control how to build a graph.

config:

the RunConfig object passed to Estimator constructor.

This function is expected to return a dictionary containing the following keys:

\* 'output': a node representing model output; required.

\* 'loss': (required) a loss node used for optimization; required for training and

evaluation.

\* 'train\_op': (optional) an operation that minimizes the loss (as output by

`tf.train.Optimizer.minimize`). If train\_op is specified, train\_op is used

for optimization as opposed to loss. Loss is always logged to tensorboard.

Notes:

\* any tf.summary written inside build graph are logged to tensorboard during training.

\* the ``build\_graph\_fn`` is called once or twice per epoch (once per training,

once per evaluation). All data loading (and preprocessing) logic not required

for serving should be in the ``input\_fn`` passed to ``learn``, ``train``,

``evalulate``, etc.

optimize\_loss\_fn:

Defaults to Trainer.get\_train\_op. A function that takes params and loss as arguments

and returns a training op. The training op is used to update parameters (that is, to learn).

metric\_fn:

A function that returns the eval\_metric\_ops dict given graph\_output, labels and weights.

Defaults to None.

Use ``twml.metrics.get\_binary\_class\_metric\_fn()`` to return a ``metric\_fn``

which implements many binary classification metrics.

run\_config (RunConfig):

optional configuration to be passed to Estimator constructor. Defaults to None.

save\_dir (String):

optional directory where to save model checkpoints,

tensorboard event files and trained parameters.

Overwrites and defaults to run\_config.model\_dir.

init\_from\_dir (String):

optional directory to load weights from.

if set to None (the default), do not init from any directory.

init\_map (map from String to String):

Must be specified if init\_from\_dir is specified.

Defines which scopes and variables to load.

Keys are the variables and scopes to load from the directory.

Values are the destinations (in the current graph) to load into.

See tf.init\_from\_checkpoint for more information.

Note that the the trainer prepends name\_scope of the form `name`/model/ to the name\_scope

of any variable defined inside `build\_graph\_fn` and this should be taken into account when

defining the values.

warm\_start\_from:

Optional string filepath to a checkpoint to warm-start from,

or a tf.estimator.WarmStartSettings object to fully configure warm-starting.

If the string filepath is provided instead of a WarmStartSettings,

then all variables are warm-started, and it is assumed that

vocabularies and Tensor names are unchanged.

profiler\_steps (Integer):

Defaults to None. If set defines the number of steps in the

`tf.train.ProfileHook <https://www.tensorflow.org/api\_docs/python/tf/train/ProfilerHook>`\_.

Captures CPU/GPU profiling information every ``profiler\_steps`` steps or seconds.

When executing ``learn``, ``train`` or ``predict`` methods,

with ``profiler\_steps`` set to a number,

a ``timeline\_X.json`` file is created in the save\_dir. This file contains profiling data

storedin Chrome trace format. To view stored data, use the Chrome browser to follow

these steps:

1) Go to the page chrome://tracing.

2) In the upper left corner, you will find Load button.

3) Press it and load our JSON file, which can be found in the ``save\_dir``

\*Warning\*: This could create too many these json files which can be a potential problem,

e.g. for HDFS there is normally quota forfile count, so use with caution.

Note: this argument is ignored when a non-None ``hooks`` argument is pasesd to

``train``, ``learn``, or ``predict`` methods. The hook can be added manually by passing

``trainer.train(..., hooks=myhooks.extend(trainer.get\_train\_hooks()))``, for example.

"""

if tensorflow.\_\_version\_\_ >= "2.0":

RuntimeError("Trainer not yet supported for Tensorflow >= 2.0")

self.\_name = name

self.\_build\_graph\_fn = build\_graph\_fn

self.\_metric\_fn = metric\_fn

self.\_tensorboard\_handle = None

self.\_current\_estimator\_spec = None # holds the current estimator spec

self.\_profiler\_steps = profiler\_steps

self.\_export\_output\_fn = None

self.\_is\_early\_stopping = False

# NOTE: Sanitize all HDFS paths first.

save\_dir = sanitize\_hdfs\_path(save\_dir)

init\_from\_dir = sanitize\_hdfs\_path(init\_from\_dir)

# warm\_start\_from can be of type tf.estimator.WarmStartSettings.

if isinstance(warm\_start\_from, str):

warm\_start\_from = sanitize\_hdfs\_path(warm\_start\_from)

# convert to twitter.deepbird.hparam.hparam.HParams object

params = twml.util.convert\_to\_hparams(params)

# keep a copy of the params because calling self.\_estimator.params creates a deepcopy

self.\_params = params

self.check\_params()

self.\_using\_hogwild = True if os.environ.get('TWML\_HOGWILD\_PORTS') else False

# configure Hogwild (needs to be called before RunConfig is created)

self.\_hogwild\_setup()

if not run\_config:

session\_config = tf.ConfigProto()

# By default each process tries to allocate (almost) all of the memory.

# This option ensures the gpu memory grows dynamically instead.

session\_config.gpu\_options.allow\_growth = True # pylint: disable=no-member

if 'TWML\_NUM\_CPUS' in os.environ:

num\_available\_cpus = int(os.environ.get("TWML\_MESOS\_CPU", "8"))

if params.num\_mkl\_threads > 1:

os.environ["OMP\_NUM\_THREADS"] = str(params.num\_mkl\_threads)

os.environ["MKL\_NUM\_THREADS"] = str(params.num\_mkl\_threads)

session\_config.inter\_op\_parallelism\_threads = num\_available\_cpus // params.num\_mkl\_threads

session\_config.intra\_op\_parallelism\_threads = params.num\_mkl\_threads

run\_config = tf.estimator.RunConfig(

session\_config=session\_config,

keep\_checkpoint\_max=self.\_params.get('keep\_checkpoint\_max', 20),

log\_step\_count\_steps=10000,

save\_checkpoints\_secs=self.\_params.get('save\_checkpoints\_secs', 600),

tf\_random\_seed=self.\_tf\_random\_seed())

elif not isinstance(run\_config, tf.estimator.RunConfig):

raise ValueError("Expecting run\_config argument of type None or tf.estimator.RunConfig"

"Got %s instead." % type(run\_config).\_\_name\_\_)

elif os.environ.get('TWML\_HOGWILD\_PORTS'):

raise ValueError("Custom RunConfig not supported with Hogwild")

if run\_config.model\_dir is None and save\_dir is None:

raise ValueError(

"Expecting either save\_dir or run\_config.model\_dir to be specified. Got None for each.")

elif run\_config.model\_dir is None:

run\_config = run\_config.replace(model\_dir=save\_dir)

elif save\_dir is None:

save\_dir = run\_config.model\_dir

self.\_save\_dir = save\_dir

self.experiment\_tracker = ExperimentTracker(self.\_params, run\_config, self.\_save\_dir)

# Check if should delete the tsd running this training job. In certain use case when

# there are other tf operations following trainer.train\_and\_evaluate (or trainer.learn),

# additional state files need to be specified to ensure those steps are executed after job restart.

kwargs['gke\_state\_files'] = kwargs.get('gke\_state\_files', ['\_SUCCESS'])

self.\_maybe\_del\_tsd\_exit(kwargs['gke\_state\_files'])

logging.info("Checkpoint and event files will be saved at save\_dir=%s", save\_dir)

self.\_optimize\_loss\_fn = self.get\_train\_op if optimize\_loss\_fn is None else optimize\_loss\_fn

# overwrite the current save\_dir

if self.\_params.get('overwrite\_save\_dir') and tf.io.gfile.exists(self.\_save\_dir):

logging.info("Trainer overwriting existing save directory: %s (params.overwrite\_save\_dir)"

% self.\_save\_dir)

# if distributed or hogwild:

if self.\_params.get('distributed', False):

# sleep for 30 seconds to allow each worker to get to this point.

time.sleep(30)

if run\_config.is\_chief:

logging.info("Chief deleting the save\_dir now")

delete\_file\_or\_dir(self.\_save\_dir)

# sleep for 30 seconds to allow each worker to get to this point.

time.sleep(30)

else:

delete\_file\_or\_dir(self.\_save\_dir)

# Exposing stats to a /vars.json endpoint that will be collected

# by the absorber

if self.\_params.get('stats\_port'):

try:

stats\_server\_utils.start\_stats\_server(self.\_params.get('stats\_port'), self.\_save\_dir)

except Exception as err:

logging.error('Failed to start the stats server. Error: %s', str(err))

checkpoint = os.path.join(self.\_save\_dir, 'checkpoint')

if tf.io.gfile.exists(checkpoint):

logging.info("The provided save\_dir directory %s already exists."

" Training will be resumed."

% checkpoint)

self.\_maybe\_restore\_checkpoint = lambda: init\_from\_checkpoint(init\_from\_dir, init\_map)

if init\_from\_dir is not None and init\_map is None:

raise ValueError("Need to provide init\_map when init\_from\_dir is provided.")

if not tf.io.gfile.exists(self.\_save\_dir):

# so tensorboard can point to a directory that exists

tf.io.gfile.mkdir(self.\_save\_dir)

self.\_estimator = tf.estimator.Estimator(

model\_fn=self.\_model\_fn,

params=self.\_params, # HParams

config=run\_config, # RunConfig

warm\_start\_from=warm\_start\_from,

model\_dir=self.\_save\_dir, # By this point it is same as run\_config.model\_dir

)

# Log parameters that are used to construct trainer. This allows people to see default values.

logging.info("Trainer constructed using the following parameters: ")

pp\_params = pp.pformat(self.\_params.values())

logging.info(pp\_params)

# Start TensorBoard

if self.\_params.get('disable\_tensorboard', False):

logging.info("Skipping launching TensorBoard [--disable\_tensorboard is set]")

elif "tensorboard\_port" in self.\_params.values() and self.\_params.tensorboard\_port is not None:

self.start\_tensorboard(self.\_params.tensorboard\_port)

# Export gauge that will track whether a model was exported

self.stats\_exporter = StatsExporter("twml.trainer")

self.export\_gauge = AtomicGauge('export\_model')

self.stats\_exporter.register\_metrics(self.export\_gauge)

def \_hogwild\_setup(self):

"""

Setup the parameters required for hogwild.

"""

self.\_num\_workers = self.\_params.get('num\_workers') or 1

logging.info("NUM\_WORKERS: %d", self.\_num\_workers)

if self.\_num\_workers <= 1:

self.\_ports = None

return

# a hogwild job is considered distributed

if 'distributed' in self.\_params:

self.\_params.set\_hparam('distributed', True)

else:

self.\_params.add\_hparam('distributed', True)

ports = os.environ.get('TWML\_HOGWILD\_PORTS')

if ports:

self.\_ports = [int(port) for port in ports.strip().split(",")]

if (self.\_num\_workers + 1!= len(self.\_ports)):

raise ValueError("Number of (workers + PS) and ports need to match")

else:

if self.\_num\_workers > 1:

raise ValueError("TWML\_HOGWILD\_PORTS needs to be set to use hogwild training")

# Split the number of data threads across multiple workers

num\_threads = self.\_params.get('num\_threads')

num\_threads\_per\_worker = int(math.ceil(float(num\_threads) / self.\_num\_workers))

self.\_params.set\_hparam('num\_threads', num\_threads\_per\_worker)

hogwild\_task\_type = os.environ.get('TWML\_HOGWILD\_TASK\_TYPE')

hogwild\_task\_id = int(os.environ.get('TWML\_HOGWILD\_TASK\_ID'))

os.environ['TF\_CONFIG'] = self.\_get\_cluster\_config(hogwild\_task\_type, hogwild\_task\_id)

def \_tf\_random\_seed(self):

""" Returns user set seed and deal with Hogwild multiple seeds """

tf\_random\_seed = self.\_params.get('tf\_random\_seed', None)

if tf\_random\_seed is None:

return None

elif self.using\_hogwild and os.environ.get('TWML\_HOGWILD\_TASK\_TYPE') == 'worker':

# chief (tf\_random\_seed), worker\_0 (tf\_random\_seed + 1), worker\_1 (tf\_random\_seed + 2)...

return tf\_random\_seed + 1 + int(os.environ.get('TWML\_HOGWILD\_TASK\_ID'))

else:

return tf\_random\_seed

def check\_params(self):

""" Verify that params has the correct key,values """

param\_values = self.\_params.values()

if 'train\_batch\_size' in param\_values:

if not isinstance(self.\_params.train\_batch\_size, int):

raise ValueError("Expecting params.train\_batch\_size to be an integer.")

if self.\_params.train\_batch\_size <= 0:

raise ValueError("train\_batch\_size needs to be positive")

else:

raise ValueError("train\_batch\_size needs to be present in params")

if 'eval\_batch\_size' in param\_values:

if not isinstance(self.\_params.eval\_batch\_size, int):

raise ValueError("Expecting params.eval\_batch\_size to be an integer.")

if self.\_params.eval\_batch\_size <= 0:

raise ValueError("eval\_batch\_size needs to be positive.")

else:

self.\_params.add\_hparam('eval\_batch\_size', self.\_params.train\_batch\_size)

if (self.\_params.get('distributed\_training\_cleanup') and

not self.\_params.get('distributed')):

# we only need to support training discontinuation for distributed training

# bc we are still using TSDs on GKE for distributed training

raise ValueError(

"Expecting params.distributed to be set if "

"params.distributed\_training\_cleanup is set."

)

def \_get\_cluster\_config(self, name, index):

"""Create a tensorflow cluster config from ports, name and index"""

host = '"localhost:%d"'

ps = host % self.\_ports[0]

chief = host % self.\_ports[1]

workers = ", ".join([host % port for port in self.\_ports[2:]])

config = \_CLUSTER\_TEMPLATE.substitute(

PS=ps,

CHIEF=chief,

WORKER=workers,

TYPE=name,

INDEX=index,

)

return config

@property

def current\_estimator\_spec(self):

"""

returns the current estimator (warning: often reset)

"""

return self.\_current\_estimator\_spec

@property

def estimator(self):

""" returns estimator encapsulated by Trainer """

return self.\_estimator

@property

def num\_workers(self):

""" returns number of workers """

return self.\_estimator.config.num\_worker\_replicas

@property

def worker\_index(self):

"""

returns index of worker in the cluster

chief has index 0

non-chief workers have indices 1 through (num\_workers - 1)

"""

return self.\_estimator.config.global\_id\_in\_cluster

@property

def using\_hogwild(self):

""" returns a bool indicating whether hogwild is being used """

return self.\_using\_hogwild

def set\_estimator(self, estimator):

""" sets the estimator used internally by Trainer """

if not isinstance(estimator, tf.estimator.Estimator):

raise ValueError("Expecting tf.estimator.Estimator")

self.\_estimator = estimator

self.\_params = self.estimator.params

@property

def params(self):

"""

returns the hyper-parameters passed to the constructor.

"""

return self.\_params

@staticmethod

def add\_parser\_arguments():

"""

Add common commandline args to parse for the Trainer class.

Typically, the user calls this function and then parses cmd-line arguments

into an argparse.Namespace object which is then passed to the Trainer constructor

via the params argument.

See the `code <\_modules/twml/argument\_parser.html#get\_trainer\_parser>`\_

for a list and description of all cmd-line arguments.

Returns:

argparse.ArgumentParser instance with some useful args already added.

"""

return twml.argument\_parser.get\_trainer\_parser()

@staticmethod

def get\_train\_op(params, loss):

"""

Return a training Op, that is, a `twml.optimizers.optimize\_loss

<https://www.tensorflow.org/api\_docs/python/tf/contrib/layers/optimize\_loss>`\_

instance given params and loss.

This method can be overwritten by passing the optimize\_loss\_fn to the Trainer

constructor.

Args:

params:

tensorflow.contrib.training.HParams instance. Recognizes the optimizer, optimizer\_summaries,

gradient\_noise\_scale, clip\_gradients and learning\_rate\_decay (including

other learning rate decay arguments).

loss:

scalar Op returned by the build\_graph that specifies the training loss to

be minimized.

"""

optimizer = params.get('optimizer')

if not optimizer:

optimizer = 'SGD'

if optimizer == 'LazyAdam':

optimizer = LazyAdamOptimizer

if optimizer == 'DGC':

optimizer = DeepGradientCompressionOptimizer(

learning\_rate=params.learning\_rate,

use\_locking=False,

name="Sparse",

density=params.get('dgc\_density'),

density\_decay=params.get('dgc\_density\_decay'),

density\_decay\_steps=params.get('dgc\_density\_decay\_steps'),

density\_decay\_rate=params.get('dgc\_density\_decay\_rate'),

min\_density=params.get('dgc\_min\_density'),

accumulation=params.get('dgc\_accumulation')

)

summaries = ['loss']

if params.get('show\_optimizer\_summaries'):

summaries = OPTIMIZER\_SUMMARIES

train\_op = optimize\_loss(

loss=loss,

global\_step=tf.train.get\_global\_step(),

optimizer=optimizer,

learning\_rate=params.learning\_rate,

summaries=summaries,

colocate\_gradients\_with\_ops=True,

gradient\_noise\_scale=params.get('gradient\_noise\_scale'),

clip\_gradients=params.get('clip\_gradients'),

learning\_rate\_decay\_fn=twml.learning\_rate\_decay.get\_learning\_rate\_decay\_fn(params)

)

return train\_op

def export\_model\_effects(self, export\_path, feature\_spec=None, log\_features=True):

# DO NOT CHANGE THE ORDER.

# This needs to be done before registering the model.

if feature\_spec:

if log\_features:

features = feature\_spec['features']

feature\_names = ['.'.join(features[fid]['featureName'].split('.')[1:]) for fid in features.keys()]

features\_to\_log = ','.join(feature\_names)

try:

model\_hash = self.experiment\_tracker.compute\_model\_hash(export\_path)

metrics.log\_usage('dbv2', 'export\_model\_effects', 'v1', custom\_attrs=[model\_hash, "feature config present", features\_to\_log])

except: # noqa: T803

logging.info("Failed to log Feature Config features")

twml.contrib.export.export\_fn.export\_feature\_spec(export\_path, feature\_spec)

export\_start\_time = time.time()

self.experiment\_tracker.export\_feature\_spec(feature\_spec)

logging.info("Exported feature spec to ML Metastore in %s seconds.", time.time() - export\_start\_time)

self.experiment\_tracker.register\_model(str(export\_path))

self.export\_gauge.increment()

@property

def best\_or\_latest\_checkpoint(self):

if self.\_is\_early\_stopping:

best\_checkpoint\_path = os.path.join(self.\_save\_dir, "best\_checkpoint")

checkpoint\_path = tf.train.latest\_checkpoint(best\_checkpoint\_path)

# Return best checkpoint if necessary

if checkpoint\_path:

return checkpoint\_path

else:

raise ValueError("Best checkpoint not found at %s." % best\_checkpoint\_path)

else: # Fallback to latest checkpoint from save directory

return self.latest\_checkpoint

@property

def latest\_checkpoint(self):

return self.estimator.latest\_checkpoint()

def export\_model(self, serving\_input\_receiver\_fn,

export\_output\_fn=None,

export\_dir=None, checkpoint\_path=None,

feature\_spec=None,

log\_features=True):

"""

Export the model for prediction. Typically, the exported model

will later be run in production servers. This method is called

by the user to export the PREDICTgraph to disk.

Internally, this method calls `tf.estimator.Estimator.export\_savedmodel

<https://www.tensorflow.org/api\_docs/python/tf/estimator/Estimator#export\_savedmodel>`\_.

Note that a valid self.\_export\_output\_fn is required.

If export\_ouput\_fn is provided, it is used to set the self.\_export\_output\_fn.

Args:

serving\_input\_receiver\_fn:

function preparing the model for inference requests.

This funtion returns the ``features`` dict passed to ``build\_graph``.

export\_dir:

directory to export a SavedModel for prediction servers.

Defaults to ``[save\_dir]/exported\_models``.

checkpoint\_path:

the checkpoint path to export. If None (the default), the most recent checkpoint

found within the model directory is chosen.

export\_output\_fn:

Function to export the graph\_output (output of build\_graph) for

prediction. Takes a graph\_output dict as sole argument and returns

the export\_output\_fns dict.

Defaults to `twml.export\_output\_fns.default\_output\_fn`.

Return:

returns a string path to exported directory.

# set the export output function

"""

if not self.is\_chief():

logging.info("Trainer.export\_model ignored due to the process not being chief.")

return

self.\_export\_output\_fn = export\_output\_fn or twml.export\_output\_fns.default\_output\_fn

if not callable(self.\_export\_output\_fn):

raise RuntimeError(

"Expecting export\_output\_fn function. Got %s."

% type(self.\_export\_output\_fn).\_\_name\_\_)

if export\_dir:

export\_dir = sanitize\_hdfs\_path(export\_dir)

if checkpoint\_path:

checkpoint\_path = sanitize\_hdfs\_path(checkpoint\_path)

else:

checkpoint\_path = self.best\_or\_latest\_checkpoint

# actually export the model using the Estimator API

export\_path = self.\_estimator.export\_savedmodel(

export\_dir\_base=export\_dir or os.path.join(self.\_save\_dir, 'exported\_models'),

serving\_input\_receiver\_fn=serving\_input\_receiver\_fn,

checkpoint\_path=checkpoint\_path)

# export\_path is bytes, need to convert to string for python3 to work.

logging.info("The exported model path is: " + str(export\_path))

self.export\_model\_effects(export\_path, feature\_spec, log\_features)

return export\_path

def \_model\_fn(self, features, labels, mode, params, config=None):

"""

returns tf.estimator.EstimatorSpec that can be used with tf.estimator.Estimators.

You would probably never need to modify this method.

Instead, you should override build\_graph, which this method calls.

Args:

features:

Dict of input tensors.

labels:

Tensor of target labels.

mode:

an instance of tf.estimator.ModeKeys.

Typically used to toggle TRAINing or EVALuation.

params:

HParams object containing hyper-parameters.

"""

# pylint: disable=too-many-branches

if isinstance(features, dict):

weights = features.get('weights', None)

else:

weights = None

with tf.variable\_scope(self.\_name + '/model'):

graph\_output = self.\_build\_graph\_fn(features, labels, mode, params, config)

loss = graph\_output['loss'] if 'loss' in graph\_output else None

self.\_maybe\_restore\_checkpoint()

with tf.variable\_scope(self.\_name + '/optim'):

train\_op = None

if mode == tf.estimator.ModeKeys.TRAIN:

if 'train\_op' in graph\_output:

train\_op = graph\_output['train\_op']

graph\_output['train\_op'] = None # remove from preds to prevent error

elif loss is not None:

train\_op = self.\_optimize\_loss\_fn(params, loss)

if params.get('train\_log\_metrics') and self.\_metric\_fn:

metric\_ops = self.\_metric\_fn(graph\_output=graph\_output, labels=labels, weights=weights)

for metric\_name in metric\_ops:

tf.summary.scalar(

name="training\_metric\_" + metric\_name,

tensor=metric\_ops[metric\_name][1]) # index 0 contains value\_op, 1 contains update\_op

if mode == tf.estimator.ModeKeys.PREDICT and self.\_export\_output\_fn is not None:

# note that this is ignored by the predict method.

# Estimator only uses export\_output\_fn for export\_model.

export\_outputs = self.\_export\_output\_fn(graph\_output)

else:

export\_outputs = None

if mode == tf.estimator.ModeKeys.EVAL and self.\_metric\_fn:

eval\_metric\_ops = self.\_metric\_fn(graph\_output=graph\_output, labels=labels, weights=weights)

else:

eval\_metric\_ops = None

# None and loss (scalar, not sliceable by TFMA) should be removed from the graph\_output

preds = {key: graph\_output[key] for key in graph\_output if (graph\_output[key] is not None) and (key is not 'loss')}

init\_feed\_dict = twml.contrib.initializers.get\_init\_feed\_dict()

scaffold = tf.train.Scaffold(init\_feed\_dict=init\_feed\_dict)

# Clear the init feed collection to avoid serializing the initializers.

twml.contrib.initializers.clear\_init\_feed\_collection()

# save estimator for use by later methods and hooks (warning: often reset)

self.\_current\_estimator\_spec = tf.estimator.EstimatorSpec(

mode=mode,

predictions=preds,

export\_outputs=export\_outputs,

loss=loss,

train\_op=train\_op,

eval\_metric\_ops=eval\_metric\_ops,

scaffold=scaffold,

)

return self.\_current\_estimator\_spec

def get\_train\_hooks(self):

"""Return SessionRunHooks used during training.

By default training uses one hooks `tf.train.StepCounterHook` for monitoring step speed.

If self.\_profiler\_steps is set then we also use the ProfilerHook `tf.train.ProfilerHook`

for monitoring the profile.

"""

# Instead of having every\_n\_steps be a constant number,

# change it dynamically based on batch size.

# Ideally we should be using every\_n\_secs, but that seems buggy as of 1.7.

# The every\_n\_steps = 20K / batch\_size

every\_n\_steps = ((2048 \* 100) // self.\_params.train\_batch\_size)

step\_counter = tf.train.StepCounterHook(

every\_n\_steps=every\_n\_steps, output\_dir=self.\_save\_dir

)

train\_hooks = [step\_counter]

if self.\_profiler\_steps is not None:

if not self.\_params.get('distributed') or self.\_estimator.config.is\_chief:

profiler = tf.train.ProfilerHook(

save\_steps=self.\_profiler\_steps,

output\_dir=self.\_save\_dir

)

train\_hooks.append(profiler)

return train\_hooks

def is\_task\_type(self, name):

"""

Helper function to specify if the current process is of the given worker type.

Note: This an only be called \*after\* self.\_hogwild\_setup() is called in \_\_init\_\_()

"""

if os.environ.get('TF\_CONFIG'):

if self.\_estimator.config.task\_type == name:

return True

else:

return False

return True

def is\_evaluator(self):

"""

Helper function to let you know if the worker is evaluator.

Note: This an only be called \*after\* self.\_hogwild\_setup() is called in \_\_init\_\_()

"""

return self.is\_task\_type("evaluator")

def is\_chief(self):

"""

Helper function to let you know if the worker is chief.

Note: This an only be called \*after\* self.\_hogwild\_setup() is called in \_\_init\_\_()

"""

return self.is\_task\_type("chief") or self.is\_task\_type("master")

def is\_ps(self):

"""

Helper function to let you know if the task is parameter server.

"""

if os.environ.get('TF\_CONFIG') and self.\_estimator.config.task\_type == 'ps':

return True

return False

def \_exit\_ps\_after\_training\_complete(self):

"""

Helper function to shutdown parameter server after training job complete (either succeed or failed).

"""

if not self.is\_ps():

return

# No need to exit ps if on the same machine

if os.environ.get('TWML\_HOGWILD\_PORTS'):

return

if self.\_params.get('disable\_auto\_ps\_shutdown', False):

logging.info("Skip shutting down parameter server after training complete [--disable\_auto\_ps\_shutdown is set]")

return

# checking job status is different on gke vs aurora

if self.\_is\_on\_gke():

get\_job\_status = functools.partial(

k8s\_status.get\_training\_job\_status,

cluster=None,

namespace=os.environ['TWML\_JOB\_ROLE'],

environment=os.environ['TWML\_JOB\_ENV'],

job\_name=os.environ['TWML\_JOB\_NAME'],

using\_tsd=True)

else:

get\_job\_status = functools.partial(

get\_distributed\_training\_job\_path,

base\_job\_path=get\_distributed\_training\_job\_path()

)

def wait\_complete\_then\_exit():

retry\_max = 60

retry = 0

while True:

try:

training\_status = get\_job\_status()

if training\_status == TrainingJobStatus.FINISHED:

logging.info("Distributed training job succeed, shutting down parameter server.")

os.\_exit(0)

elif training\_status == TrainingJobStatus.FAILED:

logging.info("Distributed training job failed, shutting down parameter server.")

os.\_exit(0)

elif training\_status == TrainingJobStatus.NOT\_FOUND:

raise Exception("Distributed training job status not found.")

else:

poke\_interval = random.randrange(60, 90) # prevent spike QPS to aurora endpoint

time.sleep(poke\_interval)

retry = 0

except Exception as e:

if retry >= retry\_max:

raise e # only exception in this thread, won't fail parameter server thread

retry += 1

poke\_interval = random.randrange(60, 90) + retry \* 10

logging.warn("Error getting distributed training job status, will retry after %s seconds." % poke\_interval)

time.sleep(poke\_interval)

Thread(target=wait\_complete\_then\_exit).start()

def get\_eval\_hooks(self): # pylint: disable=no-self-use

""" Return SessionRunHooks used during evaluation."""

return None

def get\_predict\_hooks(self):

""" Return hooks used during prediction.

If profiler\_steps is set in the constructor to the Trainer,

we pass a tf.Train.ProfilerHook to the estimator's predict function.

"""

hooks = []

if self.\_profiler\_steps is not None:

profiler = tf.train.ProfilerHook(

save\_steps=self.\_profiler\_steps,

output\_dir=self.\_save\_dir

)

hooks.append(profiler)

return hooks

def learn(self, train\_input\_fn=None, eval\_input\_fn=None,

train\_max\_steps=None,

train\_steps=None, eval\_steps=None,

train\_hooks=None, eval\_hooks=None,

early\_stop\_metric=None, early\_stop\_patience=-1,

early\_stop\_minimize=True, early\_stop\_tolerance=0, start\_epoch=0,

exporters=None, export\_output\_fn=None, max\_duration=None):

"""

Train and evaluate the estimator for ``train\_max\_steps`` steps.

Each epoch involves ``train\_steps`` training steps followed

by ``eval\_steps`` evaluation steps. Note that each step

is a ``session.run()``, that is, each batch is a step.

Args:

train\_max\_steps:

maximum number of global steps of training to run.

Defaults to params.train\_max\_steps.

None-values cause learn() to terminate after \*one\* call to train() and evaluate(),

which is usually useful when using train\_steps=-1

Non-positive values trains indefinitely in a loop (use with caution),

which is usually useful when used with early stopping.

train\_steps:

number of training steps per epoch. For example, 100 means each

training epoch will end after processing 100 batches.

Defaults to params.train\_steps.

Non-positive values and None-values go through the entire training set each epoch.

eval\_steps:

number of evaluation steps per epoch.

Defaults to params.eval\_steps.

Non-positive values and None-values go through the entire evaluation set each epoch.

train\_input\_fn:

Function to iterate through training set. It is passed to estimator.train.

eval\_input\_fn:

Function to iterate through evaluation set. It is passed to estimator.evaluate.

train\_hooks:

List of SessionRunHooks uses for training. Defaults to self.get\_train\_hooks().

eval\_hooks:

List of SessionRunHooks uses for evaluation. Defaults to self.get\_eval\_hooks()

start\_epoch:

The epoch from which to start learn. If you want to do training and evaluation

for N epochs, you can call ``learn()`` in a loop as follows:

exporters:

List of exporters called at the end of each evaluation run.

Defaults to none.

export\_output\_fn:

The output format to use for exported models.

Only used if exporters is not None.

.. code-block:: python

for epoch in range(1,max\_epoch):

trainer.learn(start\_epoch=epoch)

Early-stopping arguments:

early\_stop\_metric:

String specifying the metric to early-stop on. Required with positive

``early\_stop\_patience``. For example, 'accuracy', 'accuracy\_0', 'loss', etc.

The string is used to extract the relevant tensor Op from the dict returned by

the get\_eval\_metric\_ops method. For ``metrics`` pass to the constructor,

the string is one of those. For multi-class (that is, multi-metric)

metrics, the string may be appended with a ``\_0``, ``\_1``, etc. or one

of the ``multi\_metric\_names`` (one per class).

early\_stop\_patience:

Maximum number of epochs to wait for an improvement in the early\_stop\_metric

before breaking off training. For example, a patience of 10 means that

training will have 10 epochs to improve the metric before it is killed.

Whenever the metric is improved before running out of patience,

patience is reset to ``early\_stop\_patience``.

Defaults to -1 (that is, no early-stopping).

early\_stop\_minimize:

Set this to True (the default) for metrics that need to be minimized

(like ``loss``). Metrics like ``accuracy`` that need to be maximized

should set this to False.

early\_stop\_tolerance:

A non-negative tolerance for comparing early\_stop\_metric.

E.g. when maximizing the condition is current\_metric > best\_metric + tolerance.

Defaults to 0.

max\_duration:

A float. When this argument is defined, the job will automatically terminate after

`max\_duration` seconds if it has not already compeleted.

Returns:

The directory where the checkpoints were saved.

That is, save\_dir.

You can point TensorBoard to this directory to get metrics,

or pass it to another Trainer via ``init\_from\_dir`` when doing

multi-phase training.

"""

# pylint: disable=too-many-branches

if not callable(train\_input\_fn):

raise ValueError("Expecting callable train\_input\_fn function")

if not callable(eval\_input\_fn):

raise ValueError("Expecting callable eval\_input\_fn function")

if os.environ.get('TF\_CONFIG'):

raise ValueError("trainer.learn() can not be used with distributed / hogwild setups")

if exporters and export\_output\_fn:

self.\_export\_output\_fn = export\_output\_fn

train\_hooks = self.get\_train\_hooks() if train\_hooks is None else train\_hooks

eval\_hooks = self.get\_eval\_hooks() if eval\_hooks is None else eval\_hooks

eval\_hooks = [] if eval\_hooks is None else eval\_hooks

if train\_max\_steps is None:

train\_max\_steps = self.params.get('train\_max\_steps')

if train\_steps is None:

train\_steps = self.params.train\_steps

if train\_steps <= 0:

train\_steps = None

if eval\_steps is None:

eval\_steps = self.params.eval\_steps

if eval\_steps <= 0:

eval\_steps = None

if early\_stop\_patience > 0:

assert train\_max\_steps is not None, "Early stopping and max\_steps=None are not compatible."

# prepare early stopping hook (which also handles logic here)

self.\_is\_early\_stopping = True

early\_stop\_hook = twml.hooks.EarlyStopHook(

metric=early\_stop\_metric,

checkpoint\_dir=self.\_save\_dir,

patience=early\_stop\_patience,

minimize=early\_stop\_minimize,

tolerance=early\_stop\_tolerance,

get\_estimator\_spec\_fn=lambda: self.current\_estimator\_spec,

start\_epoch=start\_epoch)

# add early stop hook to eval hooks

eval\_hooks.append(early\_stop\_hook)

if max\_duration is not None:

train\_early\_stop\_duration\_hook = twml.hooks.EarlyStopDuration(

max\_duration=max\_duration,

exit\_on\_end=False,

save\_dir=self.\_save\_dir,

overwrite=True,

)

train\_hooks.append(train\_early\_stop\_duration\_hook)

eval\_early\_stop\_duration\_hook = twml.hooks.EarlyStopDuration(

max\_duration=max\_duration,

exit\_on\_end=False,

save\_dir=self.\_save\_dir,

overwrite=True,

)

eval\_hooks.append(eval\_early\_stop\_duration\_hook)

if not self.\_is\_early\_stopping:

if (train\_max\_steps is not None) and (train\_max\_steps <= 0):

if ((max\_duration is not None) and (max\_duration < 0)) or (max\_duration is None):

logging.warn("train.max\_steps is non-positive, and no early or duration stopping is configured. "

"Training job will loop forever.")

if train\_max\_steps is not None and train\_max\_steps > 0:

# we can't pass max\_steps AND steps to estimator.train.

# so we pass steps to estimator.train and max\_steps to this hook instead...

stop\_at\_step\_hook = twml.hooks.StopAtStepHook(last\_step=train\_max\_steps)

train\_hooks.append(stop\_at\_step\_hook)

with self.experiment\_tracker.track\_experiment(eval\_hooks,

lambda: self.current\_estimator\_spec):

# alternate training and evaluation epochs

epoch = start\_epoch

while True:

logging.info("Training epoch %d", epoch)

self.\_estimator.train(train\_input\_fn, steps=train\_steps, hooks=train\_hooks)

logging.info("Evaluating epoch %d", epoch)

eval\_result = self.\_estimator.evaluate(

eval\_input\_fn, steps=eval\_steps, hooks=eval\_hooks)

if exporters:

checkpoint\_path = self.estimator.latest\_checkpoint()

for exporter in exporters:

export\_path = os.path.join(self.\_save\_dir, "export", exporter.name)

exporter.export(

estimator=self.estimator, export\_path=export\_path,

checkpoint\_path=checkpoint\_path, eval\_result=eval\_result,

is\_the\_final\_export=False)

# If train\_max\_step is none. Terminate after one loop.

if train\_max\_steps is None:

break

# If stop\_at\_step\_hook requested a stop, break

if train\_max\_steps > 0 and stop\_at\_step\_hook.stop\_requested:

break

# early-stopping logic is handled internally by the hook

if early\_stop\_patience > 0 and early\_stop\_hook.should\_stop:

# but we still need to break here

break

epoch += 1

self.write\_state\_to\_disk(save\_dir=self.\_save\_dir, filename='\_SUCCESS')

return self.\_save\_dir

def get\_train\_spec(self, input\_fn, max\_steps=None, hooks=None):

"""Get the TrainSpec used by ``tf.train.train\_and\_evaluate``."""

if not callable(input\_fn):

raise ValueError("Expecting callable train\_input\_fn")

if max\_steps is None:

max\_steps = self.params.train\_max\_steps

if max\_steps is not None and max\_steps <= 0:

max\_steps = None

hooks = self.get\_train\_hooks() if hooks is None else hooks

return tf.estimator.TrainSpec(input\_fn=input\_fn,

max\_steps=max\_steps,

hooks=hooks)

def get\_eval\_spec(self, input\_fn, steps=None, delay=None, period=None,

hooks=None, exporters=None):

"""Get the EvalSpec used by ``tf.train.train\_and\_evaluate``."""

if not callable(input\_fn):

raise ValueError("Expecting callable eval\_input\_fn")

if steps is None:

steps = self.params.eval\_steps

if steps <= 0:

steps = None

if delay is None:

delay = self.params.eval\_delay

if period is None:

period = self.params.eval\_period

hooks = self.get\_eval\_hooks() if hooks is None else hooks

eval\_name = self.params.get("eval\_name", None)

return tf.estimator.EvalSpec(input\_fn=input\_fn,

steps=steps,

name=eval\_name,

start\_delay\_secs=delay,

throttle\_secs=period,

hooks=hooks,

exporters=exporters)

def train\_and\_evaluate(self, train\_input\_fn=None, eval\_input\_fn=None,

train\_max\_steps=None, eval\_steps=None,

eval\_delay=None, eval\_period=None,

train\_hooks=None, eval\_hooks=None,

early\_stop\_metric=None, early\_stop\_patience=-1,

early\_stop\_minimize=True, early\_stop\_tolerance=0, exporters=None,

export\_output\_fn=None, max\_duration=None):

"""

Train and evaluate the estimator for ``train\_max\_steps``

using ``tf.estimator.train\_and\_evaluate``.

With a cluster configuration provided in the ``TF\_CONFIG`` environment variable, this method

can be used for distributed training (multi-node or multi-process).

Unlike the ``learn`` method, training is continuous with ``train\_max\_steps``.

For distributed use case, evaluation happens periodically.

That is, after ``eval\_delay`` seconds, an evaluation epoch of ``eval\_step`` steps

occurs every ``eval\_period`` seconds. Evaluation happens on the most recent checkpoint.

TF defaults to saving checkpoints every 10 mins.

For local use case, training occurs for train\_max\_steps epochs followed by a

single evaluation. For local use case we therefore recommend using learn() instead

as it provides early-stopping and multiple evaluations.

``train\_and\_evaluate`` will evaluate for ``eval\_steps`` every ``eval\_period`` seconds.

It will stop after ``train\_steps`` is reached.

You must ensure that all workers/servers are assigned the same `save\_dir`.

.. Note::

If the TF\_CONFIG environment variable is set, this function assumes its running a distribute job.

Args:

train\_input\_fn:

Function to iterate through training set. It is passed to estimator.train\_and\_evalute

eval\_input\_fn:

Function to iterate through evaluation set. It is passed to estimator.train\_and\_evalute.

train\_max\_steps:

maximum number of global steps of training to run.

Defaults to params.train\_max\_steps.

Non-positive values and None-values train indefinitely (use with caution).

eval\_steps:

number of steps per evaluation.

Defaults to params.eval\_steps.

Non-positive values and None-values go through

the entire evaluation set for each evaluation.

Note that the number of eval\_steps should be high enough to minimize noise.

This is especially true for early-stopping.

eval\_delay:

Start the first evaluation after eval\_delay. Defaults to params.eval\_delay or 2\*60s.

eval\_period:

Run an evaluation every eval\_period seconds. Defaults to params.eval\_period or 10\*60s.

exporters:

List of exporters called at the end of each evaluation run.

Defaults to none.

export\_output\_fn:

The output format to use for exported models.

Only used if exporters is not None.

Early-stopping arguments:

early\_stop\_metric:

String specifying the metric to early-stop on. Required with positive

``early\_stop\_patience``. For example, 'accuracy', 'accuracy\_0', 'loss', etc.

The string is used to extract the relevant tensor Op from the dict returned by

the get\_eval\_metric\_ops method. For ``metrics`` pass to the constructor,

the string is one of those. For multi-class (that is, multi-metric)

metrics, the string may be appended with a ``\_0``, ``\_1``, etc. or one

of the ``multi\_metric\_names`` (one per class).

early\_stop\_patience:

Maximum number of epochs to wait for an improvement in the early\_stop\_metric

before breaking off training. For example, a patience of 10 means that

training will have 10 epochs to improve the metric before it is killed.

Whenever the metric is improved before running out of patience,

patience is reset to ``early\_stop\_patience``.

Defaults to -1 (that is, no early-stopping).

early\_stop\_minimize:

Set this to True (the default) for metrics that need to be minimized

(like ``loss``). Metrics like ``accuracy`` that need to be maximized

should set this to False.

early\_stop\_tolerance:

A non-negative tolerance for comparing early\_stop\_metric.

E.g. when maximizing the condition is current\_metric > best\_metric + tolerance.

Defaults to 0.

max\_duration:

A float. When this argument is defined, the job will automatically terminate after

`max\_duration` seconds if it has not already compeleted.

Returns:

The directory where the checkpoints were saved.

"""

logging.info("WARNING: Trainer.train\_and\_evaluate is an EXPERIMENTAL API.")

logging.info("Trainer.train\_and\_evaluate may change or be removed in future versions.")

if not callable(train\_input\_fn):

raise ValueError("Expecting callable train\_input\_fn function")

if not callable(eval\_input\_fn):

raise ValueError("Expecting callable eval\_input\_fn function")

self.\_exit\_ps\_after\_training\_complete()

# Maybe export in eval processes.

if self.is\_evaluator():

if self.params.get("eval\_name") is not None:

# Do not export if running special eval.

exporters = None

export\_output\_fn = None

elif exporters and export\_output\_fn:

self.\_export\_output\_fn = export\_output\_fn

else:

# Default option.

self.\_export\_output\_fn = None

train\_hooks = self.get\_train\_hooks() if train\_hooks is None else train\_hooks

train\_hooks = [] if train\_hooks is None else train\_hooks

eval\_hooks = self.get\_eval\_hooks() if eval\_hooks is None else eval\_hooks

eval\_hooks = [] if eval\_hooks is None else eval\_hooks

if train\_max\_steps is None:

train\_max\_steps = self.params.get('train\_max\_steps')

if eval\_steps is None:

eval\_steps = self.params.eval\_steps

if eval\_steps <= 0:

eval\_steps = None

if eval\_delay is None:

eval\_delay = self.params.eval\_delay

if eval\_period is None:

eval\_period = self.params.eval\_period

if early\_stop\_patience > 0:

# when training hooks detect this file, they request a stop to training

early\_stop\_path = os.path.join(self.\_save\_dir, 'earlystop\_now.txt')

# prepare early stopping hook (which also handles logic here)

self.\_is\_early\_stopping = True

eval\_early\_stop\_hook = twml.hooks.EarlyStopHook(

metric=early\_stop\_metric,

checkpoint\_dir=self.\_save\_dir,

patience=early\_stop\_patience,

minimize=early\_stop\_minimize,

tolerance=early\_stop\_tolerance,

get\_estimator\_spec\_fn=lambda: self.current\_estimator\_spec,

file\_path=early\_stop\_path,

exit\_on\_end=os.environ.get('TF\_CONFIG') is not None) # only exit for distributed jobs

# add early stop hook to eval hooks

eval\_hooks.append(eval\_early\_stop\_hook)

# prepare the commensurate training hook

train\_early\_stop\_hook = twml.hooks.StopIfExistsHook(early\_stop\_path)

train\_hooks.append(train\_early\_stop\_hook)

if max\_duration is not None:

train\_early\_stop\_duration\_hook = twml.hooks.EarlyStopDuration(

max\_duration=max\_duration,

exit\_on\_end=False,

save\_dir=self.\_save\_dir,

overwrite=self.is\_chief()

)

eval\_early\_stop\_duration\_hook = twml.hooks.EarlyStopDuration(

max\_duration=max\_duration,

exit\_on\_end=os.environ.get('TF\_CONFIG') is not None,

save\_dir=self.\_save\_dir,

overwrite=False

) # only exit for distributed jobs

train\_hooks.append(train\_early\_stop\_duration\_hook)

eval\_hooks.append(eval\_early\_stop\_duration\_hook)

with self.experiment\_tracker.track\_experiment(eval\_hooks, lambda: self.current\_estimator\_spec):

train\_spec = self.get\_train\_spec(train\_input\_fn, train\_max\_steps, train\_hooks)

eval\_spec = self.get\_eval\_spec(eval\_input\_fn, eval\_steps,

eval\_delay, eval\_period,

eval\_hooks, exporters)

self.\_train\_and\_evaluate(train\_spec, eval\_spec)

if self.is\_chief():

self.write\_state\_to\_disk(save\_dir=self.\_save\_dir, filename='\_SUCCESS')

return self.\_save\_dir

def \_train\_and\_evaluate(self, train\_spec, eval\_spec):

"""

Private method that calls

``tf.estimator.train\_and\_evaluate(self.\_estimator, train\_spec, eval\_spec)``.

"""

try:

tf.estimator.train\_and\_evaluate(self.\_estimator, train\_spec, eval\_spec)

except twml.errors.EarlyStopError:

# Ignore the exception if on evaluator.

if self.is\_evaluator():

pass

else:

raise

def train(self, input\_fn=None, steps=None, hooks=None):

"""

Train the estimator for `steps` training steps.

Args:

steps:

number of steps for which to perform training. For example, 100 means each

evaluation will end after processing 100 batches.

Defaults to None. i.e. trains on the entire dataset a single time.

Non-positive values and None-values go through the entire training set each epoch.

input\_fn:

Function to iterate through training set. It is passed to estimator.train.

hooks:

List of SessionRunHooks uses for training. Defaults to self.get\_train\_hooks().

"""

if os.environ.get('TF\_CONFIG') and "is\_calibrating" not in self.params:

raise ValueError("trainer.train() can not be used with distributed / hogwild setups")

if not callable(input\_fn):

raise ValueError("Expecting callable input\_fn function")

if self.\_is\_early\_stopping:

raise ValueError("Can not call train() after learn() when using early stopping.")

hooks = self.get\_train\_hooks() if hooks is None else hooks

self.\_estimator.train(input\_fn, steps=steps, hooks=hooks)

return self

def evaluate(self, input\_fn=None, steps=None, hooks=None, name=None):

"""

Evaluate the estimator for `steps` evaluation steps.

Args:

steps:

number of steps for which to perform evaluation. For example, 100 means each

evaluation will end after processing 100 batches.

Defaults to None. i.e. evaluates on the entire dataset a single time.

Negative values and None-values go through the entire training set each epoch.

input\_fn:

Function to iterate through evaluation set. It is passed to estimator.evaluate.

hooks:

List of SessionRunHooks used for evaluation. Defaults to None.

Note that, unlike learn(), hooks defaults to None instead of self.get\_eval\_hooks()

as the latter may implement early-stopping, which isn't necessarilty the desired

behavior when calling evaluate() on its own.

name:

Name of the evaluation if user needs to run multiple evaluations on different data sets.

Metrics for different evaluations are saved in separate folders,

and appear separately in tensorboard.

Returns:

If `is\_evaluator()`, returns a dict containing the evaluation metrics specified

in `metric\_fn` keyed by name, as well as an entry `global\_step` that contains

the value of the global step for which this evaluation was performed.

Otherwise (i.e. `is\_evaluator() == False`), returns None.

"""

if not self.is\_evaluator():

return None

if not callable(input\_fn):

raise ValueError("Expecting callable input\_fn function")

hooks = self.get\_eval\_hooks() if hooks is None else hooks

hooks = [] if hooks is None else hooks

# for consistency with train/learn

eval\_steps = None if steps is not None and steps < 0 else steps

with self.experiment\_tracker.track\_experiment(hooks, lambda: self.current\_estimator\_spec, name=name):

checkpoint = self.best\_or\_latest\_checkpoint

computed\_metrics = self.\_estimator.evaluate(

input\_fn,

steps=eval\_steps,

hooks=hooks,

checkpoint\_path=checkpoint,

name=name

)

return computed\_metrics

def start\_tensorboard(self, port=None):

"""

Start tensorboard process to visualize logs in save\_dir.

"""

logging.info("Starting tensorboard.")

if self.\_tensorboard\_handle:

logging.warn("Tensorboard already running. Nothing done.")

return

if port is None:

if 'tensorboard\_port' not in self.params.values():

raise ValueError('You must specify a port for tensorboard to run on.')

elif self.params.tensorboard\_port is None:

return

else:

port = self.params.tensorboard\_port

mldash\_path = 'experiments'

if self.experiment\_tracker.path:

mldash\_path += '/%s' % encode\_url(self.experiment\_tracker.experiment\_id)

tensorboard\_args = ['--logdir=%s' % self.\_save\_dir, '--port=%d' % port]

try:

args = ['email\_and\_launch\_tensorboard', mldash\_path, '--'] + tensorboard\_args

self.\_tensorboard\_handle = subprocess.Popen(args)

except OSError:

try:

self.\_tensorboard\_handle = subprocess.Popen(['tensorboard'] + tensorboard\_args)

except OSError:

try:

# this will work with Twitter internal pants build when run locally

args = ['./pants', 'run', 'twml:tensorboard', '--'] + tensorboard\_args

self.\_tensorboard\_handle = subprocess.Popen(args)

except OSError:

logging.error("No tensorboard installed, won't able to visualize training in tensorboard.")

def stop\_tensorboard(self):

"""

Shutdown this Trainer's associated Tensorboard.

"""

if self.\_tensorboard\_handle:

logging.info("Shutting down tensorboard.")

self.\_tensorboard\_handle.kill()

else:

logging.warn("No known tensorboard process. Nothing done.")

def calibrate(self,

calibrator,

steps=None,

input\_fn=None,

save\_calibrator=True,

hooks=None):

"""

Calibrate the calibrator for `steps` calibration steps using the estimator.train method.

The build\_graph passed to the Trainer constructor should

call calibrator.accumulate using something like tf.py\_func.

That way, when this method calls estimator.train the calibrator will

accumulate one epoch of samples. After which, this method calls calibrator.calibrate().

It is up to the user to then call calibrator.save() to save the calibrated Layer

and other information to disk for multi-phase training.

Args:

calibrator:

a twml.Calibrator instance or a dict of the form {name(str): twml.Calibrator}.

steps:

Maximum steps to accumulate examples for calibration. Optional.

If not specified, examples will be accumulated until all downsampled parts are processed.

input\_fn:

Function to iterate through training set. It is passed to estimator.train.

hooks:

List of SessionRunHooks uses for training. Defaults to self.get\_train\_hooks().

save\_calibrator:

Boolean (default: True). If set to True it will save the calibrator layer.

"""

if not callable(input\_fn):

raise ValueError("Expecting callable input\_fn function")

# making everything a dict to avoid multiple ifs

if isinstance(calibrator, twml.contrib.calibrators.Calibrator):

calibrator = {"default": calibrator}

# This is a dummy call to train, since we cannot predict without training

# from the Estimator API

self.\_estimator.train(input\_fn, steps=1)

max\_steps = steps if steps is not None else -1

for name, clbrt in sorted(calibrator.items(), key=itemgetter(0)):

count = 0

for out in self.\_estimator.predict(input\_fn, hooks=hooks, yield\_single\_examples=False):

if max\_steps > 0 and count > max\_steps:

break

clbrt.accumulate\_feature(out)

count += 1

clbrt.calibrate()

# this step is done to allow us to keep the current phases event file for

# visualization on Tensorboard. It removes all files that

# are not event files. This piece of code should be deprecated when

# we deprecate the MDL calibrator (CX-12329)

for fname in tf.io.gfile.listdir(self.\_save\_dir):

if not fname.startswith("events"):

tf.io.gfile.remove(os.path.join(self.\_save\_dir, fname))

if save\_calibrator:

# If we only have one calibrator, the calibrator signature

# will be set to default

if len(calibrator) == 1:

calibrator = calibrator['default']

calibrator.save(

self.params.save\_dir,

name=calibrator.name,

verbose=True

)

else:

for name, clbrt in calibrator.items():

clbrt.save(

self.params.save\_dir,

name=clbrt.name + str(name),

verbose=True

)

def predict(self, \*args, \*\*kwargs):

"""

Wrapper over the tensorflow `Estimator.predict

<https://www.tensorflow.org/api\_docs/python/tf/estimator/Estimator#predict>`\_.

method. See that documentation for description of arguments accepted.

If hooks is passed as an argument, the specified hooks are used.

Else when profiler\_steps is specified in the constructor of the Trainer, a

tf.train.ProfilerHook is passed to the predict interface.

Otherwise, hooks is set to an empty list.

"""

if 'hooks' not in kwargs and len(args) < 3:

# If hooks is not specified as a keyword argument, nor as a positional argument

# add hooks as a keyword argument.

kwargs['hooks'] = self.get\_predict\_hooks()

return self.estimator.predict(\*args, \*\*kwargs)

def hub\_export(self,

name,

serving\_input\_receiver\_fn,

export\_dir=None,

checkpoint\_path=None,

export\_task\_type\_overrider=None):

"""

Exports registered modules into a save directory.

This method creates a directory under export\_path with the save TF Hub.

One sub-directory (named export\_name) per module registered via register\_module\_for\_export.

Arguments:

name:

unique name of the module to export.

serving\_input\_receiver\_fn:

A function with no arguments that returns a ServingInputReceiver.

This is used with the estimator passed to export() to build the graph (in PREDICT mode)

that registers the modules for export. The model in that graph is never run,

so the actual data provided by this input fn does not matter.

export\_dir:

A string containing a directory where to write the export directories.

Defaults to the save\_dir.

checkpoint\_path:

The checkpoint path to export. Defaults to the latest.

export\_task\_type\_overrider:

Specifies the task type that will override the default task type used for export

(hogwild training defaults to evaluator, otherwise, defaults to chief)

"""

if export\_task\_type\_overrider:

if not self.is\_task\_type(export\_task\_type\_overrider):

logging.info(

f"Trainer.hub\_export ignored due to process not being {export\_task\_type\_overrider}")

return

else:

if self.\_using\_hogwild:

if not self.is\_evaluator():

logging.info("Trainer.hub\_export ignored due to the process not being evaluator.")

return

else:

if not self.is\_chief():

logging.info("Trainer.hub\_export ignored due to the process not being chief.")

return

if export\_dir:

export\_dir = sanitize\_hdfs\_path(export\_dir)

if checkpoint\_path:

checkpoint\_path = sanitize\_hdfs\_path(checkpoint\_path)

else:

checkpoint\_path = self.best\_or\_latest\_checkpoint

export\_dir = export\_dir if export\_dir is not None else self.\_save\_dir

exporter = hub.LatestModuleExporter(name, serving\_input\_receiver\_fn)

# The path\_exporter by default contains a timestamp directory in its path.

path\_exporter = exporter.export(estimator=self.estimator,

export\_path=export\_dir,

checkpoint\_path=checkpoint\_path)

# LatestModuleExporter.export() returns a binary string on Cloud ML Engine

# but tf.io.gfile.listdir() does not; this is an issue when joining paths

if isinstance(path\_exporter, bytes):

path\_exporter = path\_exporter.decode()

# Copying the saved hub module to export\_dir so we don't need to specify

# the timestamp when loading the module.

# This is a workaround due to the current implementation of hub.LatestModuleExporter.

# This works for multiple hub modules.

hub\_exported\_modules = tf.io.gfile.listdir(path\_exporter)

backup\_dir = os.path.join(export\_dir, "backups",

datetime.datetime.now().strftime('%Y-%m-%d\_%H-%M-%S'))

for folder in hub\_exported\_modules:

hub\_module\_oldpath = os.path.join(path\_exporter, folder)

hub\_module\_newpath = os.path.join(export\_dir, folder)

# If the destination already exists, move to backup

if tf.io.gfile.exists(hub\_module\_newpath):

# Ensure backup\_dir exists

tf.io.gfile.makedirs(backup\_dir)

hub\_module\_backup = os.path.join(backup\_dir, folder)

tf.io.gfile.rename(hub\_module\_newpath, hub\_module\_backup)

tf.io.gfile.rename(hub\_module\_oldpath, hub\_module\_newpath)

# Since the timestamped folder exists but is empty, we can delete it.

tf.io.gfile.rmtree(path\_exporter)

def \_is\_on\_gke(self) -> bool:

"""Returns True if running on gke."""

cluster = os.environ.get('TWML\_JOB\_CLUSTER')

if not cluster or cluster in {'smf1', 'atla'}:

return False

return True

def \_maybe\_del\_tsd\_exit(self, state\_files) -> None:

"""Handle potential early exit and TwitterSetDeployment deletion.

If:

- distributed training

- running GKE

- training is finished (all state\_files exists)

we will exit early and not restart work

If --distributed\_training\_cleanup = True then we will also handle

cleaning up the TwitterSetDeployments.

Args:

state\_files: A python list indicate state files to determine the finish

state of the job.

"""

# job type that is responsible for experiment tracking will remain alive

# until it marks the experiment as finished.

if self.experiment\_tracker.\_env\_eligible\_for\_recording\_experiment:

exp\_status = self.experiment\_tracker.get\_run\_status()

if exp\_status and exp\_status not in {'Success', 'Failed'}:

logging.info(

f"Not exiting early because experiment is still {exp\_status}."

)

return

# do not bother if we are on prem

if not self.\_is\_on\_gke():

logging.info("No need to exit early because running on prem.")

return

states = [

twml.util.file\_exist\_in\_dir(self.\_save\_dir, state\_file) for state\_file in state\_files]

do\_not\_restart = (self.\_params.get('distributed') and all(states))

if not do\_not\_restart:

return

logging.info(

f"Exiting early because a \_SUCCESS file already exists in {self.\_save\_dir}")

if self.\_params.get('distributed\_training\_cleanup'):

resource\_name = '-'.join([

os.environ['TWML\_JOB\_NAME'],

os.environ['TWML\_DISTRIBUTED\_JOB\_TYPE'],

os.environ['TWML\_JOB\_ENV'],

])

logging.info(f"Deleting TwitterSetDeployment {resource\_name}")

# each job type will manage its own deletion so that deletion happens

# in the trainer init call for every job type

# otherwise we may kill another job type during an important

# process like experiment tracking management (handled by the evaluator

kubectl\_delete\_by\_name(

zone=None,

namespace=os.environ['TWML\_JOB\_ROLE'],

resource\_type=Resource.TWITTERSETDEPLOYMENTS.value,

resource\_name=resource\_name,

wait=False,

)

sys.exit(0)

def write\_state\_to\_disk(self, save\_dir, filename='\_SUCCESS') -> None:

"""Write state file to disk to indicate the state of training process. This is usually used

to mark the state of training progress and determine the start when job restarts/resumes.

Args:

save\_dir: A str of local/gcs/hdfs dir to write the state file.

file\_name: A str indicate the state file. Default to `\_SUCCESS`.

"""

file\_path = os.path.join(save\_dir, filename)

if tf.io.gfile.exists(file\_path):

tf.logging.warn(f'{file\_path} already exist.')

return

with tf.io.gfile.GFile(file\_path, 'w') as f:

f.write('')