# pylint: disable=arguments-differ, invalid-name

"""

This file contains the DataRecordTrainer class.

"""

import warnings

import twml

from twml.trainers import DataRecordTrainer

class BatchPredictionRequestTrainer(DataRecordTrainer): # pylint: disable=abstract-method

"""

The ``BatchPredictionRequestTrainer`` implementation is intended to satisfy use cases

that input is BatchPredictionRequest at Twitter and also where only the build\_graph methods

needs to be overridden. For this reason, ``Trainer.[train,eval]\_input\_fn`` methods

assume a DataRecord dataset partitioned into part files stored in compressed (e.g. gzip) format.

For use-cases that differ from this common Twitter use-case,

further Trainer methods can be overridden.

If that still doesn't provide enough flexibility, the user can always

use the tf.estimator.Esimator or tf.session.run directly.

"""

def \_\_init\_\_(

self, name, params,

build\_graph\_fn,

feature\_config=None,

\*\*kwargs):

"""

The BatchPredictionRequestTrainer constructor builds a

``tf.estimator.Estimator`` and stores it in self.estimator.

For this reason, BatchPredictionRequestTrainer accepts the same Estimator constructor arguments.

It also accepts additional arguments to facilitate metric evaluation and multi-phase training

(init\_from\_dir, init\_map).

Args:

parent arguments:

See the `Trainer constructor <#twml.trainers.Trainer.\_\_init\_\_>`\_ documentation

for a full list of arguments accepted by the parent class.

name, params, build\_graph\_fn (and other parent class args):

see documentation for twml.Trainer and twml.DataRecordTrainer doc.

feature\_config:

An object of type FeatureConfig describing what features to decode.

Defaults to None. But it is needed in the following cases:

- `get\_train\_input\_fn()` / `get\_eval\_input\_fn()` is called without a `parse\_fn`

- `learn()`, `train()`, `eval()`, `calibrate()` are called without providing `\*input\_fn`.

\*\*kwargs:

further kwargs can be specified and passed to the Estimator constructor.

"""

# Check and update train\_batch\_size and eval\_batch\_size in params before initialization

# to print correct parameter logs and does not stop running

# This overwrites batch\_size parameter constrains in twml.trainers.Trainer.check\_params

updated\_params = self.check\_batch\_size\_params(params)

super(BatchPredictionRequestTrainer, self).\_\_init\_\_(

name=name, params=updated\_params, build\_graph\_fn=build\_graph\_fn, \*\*kwargs)

def check\_batch\_size\_params(self, params):

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

# updated\_params is an instance of tensorflow.contrib.training.HParams

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

param\_values = updated\_params.values()

# twml.trainers.Trainer.check\_params already checks other constraints,

# such as being an integer

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

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

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

if param\_values['train\_batch\_size'] != 1:

# This can be a bit annoying to force users to pass the batch sizes,

# but it is good to let them know what they actually use in the models

# Use warning instead of ValueError in there to continue the run

# and print out that train\_batch\_size is changed

warnings.warn('You are processing BatchPredictionRequest data, '

'train\_batch\_size is always 1.\n'

'The number of DataRecords in a batch is determined by the size '

'of each BatchPredictionRequest.\n'

'If you did not pass train.batch\_size or eval.batch\_size, and '

'the default batch\_size 32 was in use,\n'

'please pass --train.batch\_size 1 --eval.batch\_size 1')

# If the upper error warning, change/pass --train.batch\_size 1

# so that train\_batch\_size = 1

updated\_params.train\_batch\_size = 1

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

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

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

if param\_values['eval\_batch\_size'] != 1:

# This can be a bit annoying to force users to pass the batch sizes,

# but it is good to let them know what they actually use in the models

# Use warning instead of ValueError in there to continue the run

# and print out that eval\_batch\_size is changed

warnings.warn('You are processing BatchPredictionRequest data, '

'eval\_batch\_size is also always 1.\n'

'The number of DataRecords in a batch is determined by the size '

'of each BatchPredictionRequest.\n'

'If you did not pass train.batch\_size or eval.batch\_size, and '

'the default batch\_size 32 was in use,\n'

'please pass --train.batch\_size 1 --eval.batch\_size 1')

# If the upper warning raises, change/pass --eval.batch\_size 1

# so that eval\_batch\_size = 1

updated\_params.eval\_batch\_size = 1

if 'eval\_batch\_size' not in param\_values:

updated\_params.eval\_batch\_size = 1

if not updated\_params.eval\_batch\_size:

updated\_params.eval\_batch\_size = 1

return updated\_params

@staticmethod

def add\_batch\_prediction\_request\_arguments():

"""

Add commandline args to parse typically for the BatchPredictionRequestTrainer 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.

"""

parser = super(BatchPredictionRequestTrainer,

BatchPredictionRequestTrainer).add\_parser\_arguments()

# mlp arguments

parser.add\_argument(

'--model.use\_existing\_discretizer', action='store\_true',

dest="model\_use\_existing\_discretizer",

help='Load a pre-trained calibration or train a new one')

parser.add\_argument(

'--model.use\_binary\_values', action='store\_true',

dest='model\_use\_binary\_values',

help='Use the use\_binary\_values optimization')

# control hom many featues we keep in sparse tensors

# 12 is enough for learning-to-rank for now

parser.add\_argument(

'--input\_size\_bits', type=int, default=12,

help='Number of bits allocated to the input size')

parser.add\_argument(

'--loss\_function', type=str, default='ranknet',

dest='loss\_function',

help='Options are pairwise: ranknet (default), lambdarank, '

'listnet, listmle, attrank, '

'pointwise')

# whether convert sparse tensors to dense tensor

# in order to use dense normalization methods

parser.add\_argument(

'--use\_dense\_tensor', action='store\_true',

dest='use\_dense\_tensor',

default=False,

help='If use\_dense\_tensor is False, '

'sparse tensor and spare normalization are in use. '

'If use\_dense\_tensor is True, '

'dense tensor and dense normalization are in use.')

parser.add\_argument(

'--dense\_normalization', type=str, default='mean\_max\_normalizaiton',

dest='dense\_normalization',

help='Options are mean\_max\_normalizaiton (default), standard\_normalizaiton')

parser.add\_argument(

'--sparse\_normalization', type=str, default='SparseMaxNorm',

dest='sparse\_normalization',

help='Options are SparseMaxNorm (default), SparseBatchNorm')

# so far only used in pairwise learning-to-rank

parser.add\_argument(

'--mask', type=str, default='full\_mask',

dest='mask',

help='Options are full\_mask (default), diag\_mask')

return parser