from copy import deepcopy

import random

import types

from twitter.deepbird.util.thrift.simple\_converters import (

bytes\_to\_thrift\_object, thrift\_object\_to\_bytes)

from tensorflow.compat.v1 import logging

from com.twitter.ml.api.ttypes import DataRecord # pylint: disable=import-error

import tensorflow.compat.v1 as tf

import twml

class PermutedInputFnFactory(object):

def \_\_init\_\_(self, data\_dir, record\_count, file\_list=None, datarecord\_filter\_fn=None):

"""

Args:

data\_dir (str): The location of the records on hdfs

record\_count (int): The number of records to process

file\_list (list<str>, default=None): The list of data files on HDFS. If provided, use this instead

of data\_dir

datarecord\_filter\_fn (function): a function takes a single data sample in com.twitter.ml.api.ttypes.DataRecord format

and return a boolean value, to indicate if this data record should be kept in feature importance module or not.

"""

if not (data\_dir is None) ^ (file\_list is None):

raise ValueError("Exactly one of data\_dir and file\_list can be provided. Got {} for data\_dir and {} for file\_list".format(

data\_dir, file\_list))

file\_list = file\_list if file\_list is not None else twml.util.list\_files(twml.util.preprocess\_path(data\_dir))

\_next\_batch = twml.input\_fns.default\_input\_fn(file\_list, 1, lambda x: x,

num\_threads=2, shuffle=True, shuffle\_files=True)

self.records = []

# Validate datarecord\_filter\_fn

if datarecord\_filter\_fn is not None and not isinstance(datarecord\_filter\_fn, types.FunctionType):

raise TypeError("datarecord\_filter\_fn is not function type")

with tf.Session() as sess:

for i in range(record\_count):

try:

record = bytes\_to\_thrift\_object(sess.run(\_next\_batch)[0], DataRecord)

if datarecord\_filter\_fn is None or datarecord\_filter\_fn(record):

self.records.append(record)

except tf.errors.OutOfRangeError:

logging.info("Stopping after reading {} records out of {}".format(i, record\_count))

break

if datarecord\_filter\_fn:

logging.info("datarecord\_filter\_fn has been applied; keeping {} records out of {}".format(len(self.records), record\_count))

def \_get\_record\_generator(self):

return (thrift\_object\_to\_bytes(r) for r in self.records)

def get\_permuted\_input\_fn(self, batch\_size, parse\_fn, fname\_ftypes):

"""Get an input function that passes in a preset number of records that have been feature permuted

Args:

parse\_fn (function): The function to parse inputs

fname\_ftypes: (list<(str, str)>): The names and types of the features to permute

"""

def permuted\_parse\_pyfn(bytes\_array):

out = []

for b in bytes\_array:

rec = bytes\_to\_thrift\_object(b, DataRecord)

if fname\_ftypes:

rec = \_permutate\_features(rec, fname\_ftypes=fname\_ftypes, records=self.records)

out.append(thrift\_object\_to\_bytes(rec))

return [out]

def permuted\_parse\_fn(bytes\_tensor):

parsed\_bytes\_tensor = parse\_fn(tf.py\_func(permuted\_parse\_pyfn, [bytes\_tensor], tf.string))

return parsed\_bytes\_tensor

def input\_fn(batch\_size=batch\_size, parse\_fn=parse\_fn, factory=self):

return (tf.data.Dataset

.from\_generator(self.\_get\_record\_generator, tf.string)

.batch(batch\_size)

.map(permuted\_parse\_fn, 4)

.make\_one\_shot\_iterator()

.get\_next())

return input\_fn

def \_permutate\_features(rec, fname\_ftypes, records):

"""Replace a feature value with a value from random selected record

Args:

rec: (datarecord): A datarecord returned from DataRecordGenerator

fname\_ftypes: (list<(str, str)>): The names and types of the features to permute

records: (list<datarecord>): The records to sample from

Returns:

The record with the feature permuted

"""

rec\_new = deepcopy(rec)

rec\_replace = random.choice(records)

# If the replacement datarecord does not have the feature type entirely, add it in

# to make the logic a bit simpler

for fname, feature\_type in fname\_ftypes:

fid = twml.feature\_id(fname)[0]

if rec\_replace.\_\_dict\_\_.get(feature\_type, None) is None:

rec\_replace.\_\_dict\_\_[feature\_type] = (

dict() if feature\_type != 'binaryFeatures' else set())

if rec\_new.\_\_dict\_\_.get(feature\_type, None) is None:

rec\_new.\_\_dict\_\_[feature\_type] = (

dict() if feature\_type != 'binaryFeatures' else set())

if feature\_type != 'binaryFeatures':

if fid not in rec\_replace.\_\_dict\_\_[feature\_type] and fid in rec\_new.\_\_dict\_\_.get(feature\_type, dict()):

# If the replacement datarecord does not contain the feature but the original does

del rec\_new.\_\_dict\_\_[feature\_type][fid]

elif fid in rec\_replace.\_\_dict\_\_[feature\_type]:

# If the replacement datarecord does contain the feature

if rec\_new.\_\_dict\_\_[feature\_type] is None:

rec\_new.\_\_dict\_\_[feature\_type] = dict()

rec\_new.\_\_dict\_\_[feature\_type][fid] = rec\_replace.\_\_dict\_\_[feature\_type][fid]

else:

# If neither datarecord contains this feature

pass

else:

if fid not in rec\_replace.\_\_dict\_\_[feature\_type] and fid in rec\_new.\_\_dict\_\_.get(feature\_type, set()):

# If the replacement datarecord does not contain the feature but the original does

rec\_new.\_\_dict\_\_[feature\_type].remove(fid)

elif fid in rec\_replace.\_\_dict\_\_[feature\_type]:

# If the replacement datarecord does contain the feature

if rec\_new.\_\_dict\_\_[feature\_type] is None:

rec\_new.\_\_dict\_\_[feature\_type] = set()

rec\_new.\_\_dict\_\_[feature\_type].add(fid)

# If neither datarecord contains this feature

else:

# If neither datarecord contains this feature

pass

return rec\_new