"""

This module contains utility functions for twml.

"""

import argparse

from datetime import datetime

import itertools

import json

import logging as \_logging

import os

import re

from twitter.ml.common.resources import AuroraPath

from twitter.deepbird.hparam import HParams

from twitter.deepbird.io.util import (

\_get\_feature\_id, # noqa: F401

feature\_id, # noqa: F401

preprocess\_feature\_regex, # noqa: F401

preprocess\_path, # noqa: F401

sanitize\_hdfs\_path, # noqa: F401

is\_string, # noqa: F401

list\_files, # noqa: F401

match\_files, # noqa: F401

)

from twitter.deepbird.io.legacy.util import (

batch\_apply, # noqa: F401

boolean\_mask, # noqa: F401

fixed\_length\_tensor, # noqa: F401

)

from twitter.deepbird.sparse.util import (

convert\_to\_sparse, # noqa: F401

limit\_bits, # noqa: F401

)

from dateutil import rrule

from joblib import delayed, Parallel

from six import string\_types

from absl import logging

from libtwml import CLIB, OPLIB # noqa: F401

import tensorflow.compat.v1 as tf

from tensorflow.python.platform import tf\_logging

import twml

from twml.feature\_config import FeatureConfigBuilder

# big\_prime is less than 2\*\*32

# This just needs to be co-prime with powers of 2

# any large prime is sufficient, but it's not necessary.

HASHING\_PRIME = 2479700537

def multiplicative\_hash(input, hash\_constant=HASHING\_PRIME):

return input \* hash\_constant

def \_return\_tensors\_from\_checkpoint\_folder(init\_dir, model\_name=None):

"""Returns tensors list from a checkpoint folder

Args:

init\_dir: Name of the checkpoint directory.

model\_name: the model which we will use to obtain the checkpoint

(e.g. model.ckpt-50000) if set to None it will default to the

latest model saved in the checkpont file.

"""

if model\_name is None:

# gets the most recently generated model.cpkt file

model\_path = tf.train.latest\_checkpoint(init\_dir)

if model\_path is None:

raise ValueError("Could not find a valid model checkpoint inside the directory")

else:

model\_path = os.path.join(init\_dir, model\_name)

reader = tf.train.NewCheckpointReader(model\_path)

try:

return (reader.debug\_string().decode("utf-8"))

except OSError:

logging.error('Could not decode the string')

def get\_scope\_dict(init\_dir, incoming\_scope\_name, current\_scope\_name, model\_name=None):

"""Returns tensors map from a checkpoint file.

Args:

file\_name:

Name of the checkpoint directory.

incoming\_scope\_name:

scope name of the previous phase

current\_scope\_name:

scope name of current phase

model\_name:

the model which we will use to obtain the checkpoint

(e.g. model.ckpt-50000) if set to None it will default

to the latest model saved in the checkpoint file.

Returns:

init\_map:

init\_map which will be inputted to the checkpoint

"""

init\_map = {}

reader\_dump = \_return\_tensors\_from\_checkpoint\_folder(init\_dir=init\_dir,

model\_name=model\_name).splitlines()

for member in reader\_dump:

# remove global\_step since it is not necessary

if 'global\_step' not in member:

saved\_variables = str(member.split(" ")[0])

saved\_scope = saved\_variables.rsplit('/', 1)[0] + "/"

new\_scope = saved\_scope.replace(incoming\_scope\_name, current\_scope\_name, 1)

# create key in init\_map

if saved\_scope not in init\_map.keys(): # pylint: disable=dict-keys-not-iterating

init\_map[saved\_scope] = new\_scope

return init\_map

def get\_init\_map(

init\_from\_dir,

exclude\_var\_names=None,

exclude\_name\_scopes=None,

name\_scope\_to\_remove=None,

name\_scope\_to\_prepend=None):

"""

Builds a map for initializing from a checkpoint (see tf.train.init\_from\_checkpoint).

It assumes that the latter part of the variable names are consistent between the checkpoint and

the new model, but their name\_scopes may be different. If the checkpoint model has variable names

of the form old/scope/var/foo, and the corresponding variable names for the new model should be

my/new/scope/var/foo, then you should set name\_scope\_to\_remove = 'old/' and

name\_scope\_to\_prepend = 'my/new/'.

This function can be used to

1. Generate an ``init\_map`` map that can be passed to the ``Trainer`` init or

2. Used to generate an ``init\_map`` directly inside ``build\_graph\_fn``, in

which case it should be passed directly to ``tf.train.init\_from\_checkpoint`` inside

``build\_graph\_fn``, in which case you do not also need to specify the ``init\_map`` argument to

the trainer.

Parameters

----------

init\_from\_dir: Directory containing checkpoint

exclude\_var\_names: list[str]

List of variables in the checkpoint that should be excluded from the map.

exclude\_name\_scopes: list[str]

List of name\_scopes in the checkpoint model that should be excluded from the map.

name\_scope\_to\_remove: str

portion of name\_scope for checkpoint variables that should not be included in variable names

for new model.

name\_scope\_to\_prepend: str

name\_scope to prepend to variable names in checkpoint to give variable names for new model.

Returns

-------

dict

keys are variable names in the checkpoint and values are variable names in the new model,

into which the checkpoint parameters should be loaded.

"""

vars\_to\_restore = get\_checkpoint\_variable\_names(

init\_from\_dir,

exclude\_var\_names=exclude\_var\_names,

exclude\_scopes=exclude\_name\_scopes,

)

if name\_scope\_to\_prepend is not None:

if not name\_scope\_to\_prepend.endswith('/'):

name\_scope\_to\_prepend += '/'

if name\_scope\_to\_remove is not None:

if not name\_scope\_to\_remove.endswith('/'):

name\_scope\_to\_remove += '/'

init\_map = {}

for var\_name in vars\_to\_restore:

var\_name\_checkpoint = var\_name

if name\_scope\_to\_remove is not None:

var\_name = var\_name.replace(name\_scope\_to\_remove, '')

var\_name\_new\_model = var\_name

if name\_scope\_to\_prepend is not None:

var\_name\_new\_model = name\_scope\_to\_prepend + var\_name\_new\_model

init\_map[var\_name\_checkpoint] = var\_name\_new\_model

return init\_map

def get\_checkpoint\_variable\_names(model\_dir, exclude\_var\_names=None, exclude\_scopes=None):

"""

Gets a list of variable names from the latest checkpoint in model\_dir.

Removes variables with scope defined by exclude\_scopes, and/or with names defined by

exclude\_var\_names.

Args:

model\_dir (str): Directory containing checkpoint file for the pre-trained model

exclude\_var\_names (list): Optional variable names to exclude (can include full/partial scope)

exclude\_scopes (list): Optional scopes to exclude

Returns:

list: variable names

"""

checkpoint\_path = tf.train.latest\_checkpoint(model\_dir)

variables\_and\_shapes = tf.train.list\_variables(checkpoint\_path)

def \_keep(name):

if exclude\_scopes and any(name.startswith(exc\_scope) for exc\_scope in exclude\_scopes):

return False

if exclude\_var\_names and any(name.endswith(exc\_var) for exc\_var in exclude\_var\_names):

return False

return True

names = [x[0] for x in variables\_and\_shapes if \_keep(x[0])]

return names

def to\_snake\_case(name):

"""

Changes name to snake case

"""

intermediate = re.sub('(.)([A-Z][a-z0-9]+)', r'\1\_\2', name)

insecure = re.sub('([a-z])([A-Z])', r'\1\_\2', intermediate).lower()

# If the class is private the name starts with "\_" which is not secure

# for creating scopes. We prefix the name with "private" in this case.

if insecure[0] != '\_':

return insecure

return 'private' + insecure

def copy\_phase\_inputs(init\_dir, dest\_dir):

"""Automatically copies the .json.tf from the init\_dir to save\_dir

so we can load multiple parameters at the same time.

Args:

init\_dir:

Name of the checkpoint directory.

dest\_dir:

Name of the output directory.

"""

if init\_dir is not None:

# we are using tf.io.gfile so we can use it with both local and hdfs paths

for files in tf.io.gfile.listdir(init\_dir):

if files.endswith(".json.tf"):

src\_file = os.path.join(init\_dir, files)

dest\_file = os.path.join(dest\_dir, files)

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

# creates the folder

try:

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

# to prevent racing condition

except OSError:

if not tf.io.gfile.isdir(dest\_dir):

raise

# dest\_file may be old if it exists and

# dest\_file gets copied several times in distributed training

tf.io.gfile.copy(src\_file, dest\_file, overwrite=True)

def rehash\_sparse\_features\_nbits(sp\_a, nbits, hash\_fn=multiplicative\_hash):

"""

Rehash the feature ids of the sparse tensor,

and limit the output to n bits.

This is useful for making the distribution of

feature\_ids more uniform, which may improve performance

in some situations.

This would typically be used on the output of

PercentileDiscretizer, since it assigns many

bins to low-valued output feature ids.

Input feature IDs should take values less than 2\*\*32,

and nbits should be less than 32

Args:

sp\_a:

a tf.SparseTensor object

nbits:

integer number of bits to mask output feature\_ids

hash\_fn:

Function that takes integer values and returns hashes of these values.

The output does not need to be masked to the desired number of bits,

as this masking will be taken care of. Default value = multiplicative\_hash.

Returns:

a new tf.SparseTensor

"""

feature\_ids = sp\_a.indices[:, 1]

feature\_ids = hash\_fn(feature\_ids)

sample\_ids = sp\_a.indices[:, 0]

values = sp\_a.values

dense\_shape = sp\_a.dense\_shape

indices = tf.stack([sample\_ids, feature\_ids], axis=1)

sp\_a = tf.SparseTensor(indices, values, dense\_shape)

# note - we need 2\*\*nbits >= batch size

# otherwise, sample\_ids will be squashed by the mask.

return limit\_sparse\_tensor\_size(sp\_a, nbits)

def convert\_to\_hparams(opt):

"""

Converts argparse.Namespace object to twitter.deepbird.hparam.hparam.HParams.

Note that tensorflow.contrib.training.HParams is gone in TF 2.x, and we forward ported

tensorflow.contrib.training.HParams to twitter.deepbird.hparam.hapram.HParams.

NOTE: If you are using estimators, please don't call this method and directly pass python dict

to TensorFlow estimator. Starting TensorFlow 2.0, Estimator will only accept dicts.

"""

# Convert to dict so we can iterate through it cleanly.

if isinstance(opt, argparse.Namespace):

params\_dict = vars(opt)

elif isinstance(opt, dict):

params\_dict = opt

elif isinstance(opt, HParams):

logging.warning('If you are using Estimator, please pass python dict directly to Estimator.')

params\_dict = opt.values()

else:

raise ValueError("Input can not be of type %s. "

"It can be one of { argparse.Namespace, dict, "

"twitter.deepbird.hparam.HParams}."

% type(opt))

params = HParams()

# Hack to convert all parameters from hdfs:/// format to hdfs://default/

# Note: .items() makes a copy in python 2.7, but that is fine since the performance isn't critical.

for key, val in params\_dict.items():

val = params\_dict[key]

# Fix the path if the value is a string

if isinstance(val, str):

params.add\_hparam(key, sanitize\_hdfs\_path(val))

else:

params.add\_hparam(key, val)

return params

def dynamic\_partition(features, partitions, num\_partitions=2, name=None):

"""

Partitions each of the tensor in features using the provided mask.

Args:

features:

A single tensor or an iterable of tensors (list, tuple, dict)

partitions:

A bool or integer tensor representing the partitions.

Returns partitioned outputs as a list. Each element of the list is the same type as features.

This uses tf.dynamic\_partition but adds the following niceties:

- features can be a list or dict of different tensor types.

- only a partition tensor is used to partition all the feature tensors recursively.

- the partition tensor is automatically converted into an integer tensor.

- defaults to num\_partitions == 2

"""

if not isinstance(features, (dict, list, tuple, tf.Tensor)):

raise AssertionError("features container must be a dict, list, or tuple, tf.Tensor")

if isinstance(partitions, tf.Tensor):

partitions = tf.cast(partitions, tf.int32)

if isinstance(features, tf.Tensor):

return tf.dynamic\_partition(features, partitions, num\_partitions, name)

outputs = []

for \_ in range(num\_partitions):

if isinstance(features, (tuple, list)):

# Create an empty list of lists first, will be converted to right type afterwards.

outputs.append([None for \_ in range(len(features))])

else:

outputs.append(dict())

iterable = features.items() if isinstance(features, dict) else enumerate(features)

# Handling partitions of nested classes handled here:

# Recursively call dynamic\_partition for containers

for key, feature in iterable:

name\_key = None if name is None else name + "\_" + str(key)

if isinstance(partitions, tf.Tensor):

results = tf.dynamic\_partition(feature, partitions, num\_partitions, name\_key)

else:

results = tf.dynamic\_partition(feature, partitions[key], num\_partitions[key], name\_key)

# Append the result to the proper output container

for idx, result in enumerate(results):

outputs[idx][key] = result

# if input is tuple, convert list of lists back to list of tuples

if isinstance(features, tuple):

outputs = [type(features)(output) for output in outputs]

return outputs

def write\_file(filename, contents, encode=False):

'''

Optionally encodes contents and writes contents to a file.

Arguments:

filename:

path to file where the contents will be saved.

Accepts HDFS and local paths.

contents:

contents to save to the file.

Must be a string when encode is False.

encode:

False | 'json'. When encode='json', contents is encoded

with json.dumps.

'''

if encode == 'json':

contents = json.dumps(contents)

elif not is\_string(contents):

raise ValueError("Expecting string for encode=False")

graph = tf.Graph()

with graph.as\_default():

write = tf.write\_file(filename, contents)

with tf.Session(graph=graph) as sess:

sess.run(write)

def read\_file(filename, decode=False):

'''

Reads contents from a file and optionally decodes it.

Arguments:

filename:

path to file where the contents will be loaded from.

Accepts HDFS and local paths.

decode:

False | 'json'. When decode='json', contents is decoded

with json.loads. When False, contents is returned as is.

Returns:

contents

'''

graph = tf.Graph()

with graph.as\_default():

read = tf.read\_file(filename)

with tf.Session(graph=graph) as sess:

contents = (sess.run(read))

# particular version of TF and/or Python may or may not perform decoding step from utf-8 to str

if not isinstance(contents, str):

contents = contents.decode()

if decode == 'json':

contents = json.loads(contents)

return contents

def setup\_tf\_logging\_formatter():

formatter = \_logging.Formatter(

'%(asctime)s [%(levelname)s] %(name)s: %(message)s',

None)

# Setting up absl logging verbosity

logging.set\_verbosity('info')

logging.set\_stderrthreshold('info')

logging.get\_absl\_handler().setFormatter(formatter)

tf.logging.set\_verbosity(tf.logging.INFO)

# Set tensorflow logging handler format

if len(tf\_logging.get\_logger().handlers) > 0:

tf\_logging.get\_logger().handlers[0].setFormatter(formatter)

def set\_tensorflow\_log\_level(log\_level):

"""

Sets tensorflow's default logging level.

0. all logs are shown.

1. filter out INFO logs.

2. filter out WARNINGs and INFOs.

3. filter out ERRORs, WARNINGs, and INFOs.

Note that tf.Print output are INFO logs, so setting log\_level above 0 would hide

output from tf.Print.

"""

assert isinstance(log\_level, int) and log\_level >= 0 and log\_level <= 3

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = str(log\_level)

def weighted\_average(values, weights):

"""

Compute a weighted average using the given values and weights.

E.g. this is usually used to compute a weighted loss given sample weights.

"""

return tf.reduce\_sum(tf.multiply(values, weights)) / tf.reduce\_sum(weights)

def backup\_checkpoint(checkpoint\_path\_prefix,

backup\_path='backup',

empty\_backup=True):

"""

Creates a backup copy of a checkpoint in backup\_dir.

This function is used by the Trainer for early-stopping.

Arguments:

checkpoint\_path\_prefix:

Prefix of the path to the checkpoint files.

backup\_path:

path to a directory where checkpoint files will be backed up.

empty\_backup:

When True (the default), the current contents of the backup directory

are removed before the backup is performed.

Returns:

The number of backed up files.

"""

checkpoint\_file\_prefix = os.path.basename(checkpoint\_path\_prefix)

if tf.io.gfile.exists(backup\_path) and empty\_backup:

tf.io.gfile.rmtree(backup\_path)

tf.io.gfile.mkdir(backup\_path)

n\_backup = 0

# copy all checkpoint files to backup directory (TODO use gfile.glob instead)

try:

checkpoint\_files = tf.io.gfile.glob(checkpoint\_path\_prefix + "\*")

if len(checkpoint\_files) == 0:

raise twml.errors.CheckpointNotFoundError("%s not found" % checkpoint\_path\_prefix)

for filename in checkpoint\_files:

n\_backup += 1

tf.io.gfile.copy(

src=filename,

dst=os.path.join(backup\_path, os.path.basename(filename))

)

except tf.errors.OpError as ex:

raise twml.errors.CheckpointNotFoundError(

f"{str(ex)}\n {checkpoint\_path\_prefix} not found."

)

# tf.train.latest\_checkpoint needs the 'checkpoint' file.

with tf.io.gfile.GFile(os.path.join(backup\_path, 'checkpoint'), 'w') as f:

f.write('model\_checkpoint\_path: "%s"\n' % checkpoint\_file\_prefix)

return n\_backup

def set\_only\_checkpoint(source\_path, dest\_path, remove\_source=True):

"""

Removes the checkpoint and model.ckpt\* files from dest\_path.

Moves the latest checkpoint from source\_path to dest\_path.

Arguments:

source\_path:

path to directory containing the latest checkpoint.

Should contain a valid checkpoint file and model.ckpt files.

For early-stopping, this should be the save\_dir/best\_checkpoint dir.

dest\_path:

path to directory where the latest checkpoint files will be moved.

All its checkpoint and model.ckpt\* files will be removed.

For early-stopping, this should be the save\_dir.

remove\_source:

When True (the default), deletes the source directory.

Note that even when False, its checkpoint files are moved to

dest\_path anyway.

This deletes the source directory (and any remaining contents).

"""

# make it so that source\_path checkpoint is the only checkpoint

source\_path\_prefix = tf.train.latest\_checkpoint(source\_path)

if source\_path\_prefix is not None:

# remove intermediate checkpoints

for filename in tf.io.gfile.listdir(dest\_path):

if filename.startswith("model.ckpt"):

tf.io.gfile.Remove(os.path.join(dest\_path, filename))

# move contents of source\_path to dest\_path

for filename in tf.io.gfile.listdir(source\_path):

tf.io.gfile.rename(

oldname=os.path.join(source\_path, filename),

newname=os.path.join(dest\_path, filename),

overwrite=True) # overwrite "checkpoint" file

# delete the source\_path dir

if remove\_source:

tf.io.gfile.rmtree(source\_path)

def list\_files\_by\_datetime(

base\_path,

start\_datetime,

end\_datetime=None,

datetime\_prefix\_format='%Y/%m/%d/%H',

extension='lzo',

parallelism=1,

hour\_resolution=1,

sort=False

):

"""List files matching `base\_path/dt\_prefix\_format/\*.extension` for the requested datetime range.

Args:

base\_path:

The base path. If `None`, returns `None`.

start\_datetime:

A `datetime.datetime` or string representing the start of the range (inclusive).

If `None`, it returns `list\_files(base\_path, extension, sort)`.

end\_datetime:

A `datetime.datetime` or string representing the end of the range (inclusive).

If `None`, assumed to be the same as start\_datetime.

datetime\_prefix\_format:

Format compatible with `datetime.datetime.strftime`

(https://docs.python.org/2/library/datetime.html#strftime-and-strptime-behavior).

extension:

The extension of the files composing the dataset (e.g. 'lzo').

parallelism:

The number of threads used to process list patterns (this is mostly useful

when dealing with filesystems such as HDFS in which listing files is a potentially expensive

operation).

hour\_resolution:

The separation between consecutive hours. The default value is 1.

sort:

bool, whether to return a sorted list of files. Default False.

Returns:

A list with all the matching files.

Raises:

errors.OpError: If there are filesystem / directory listing errors.

"""

if hour\_resolution is None:

hour\_resolution = 1

if base\_path is None:

return None

if start\_datetime is None:

return list\_files(base\_path, extension, sort)

# Do this in case people want to use a single day for training.

if end\_datetime is None:

end\_datetime = start\_datetime

assert parallelism > 0

assert start\_datetime <= end\_datetime

if isinstance(start\_datetime, str):

start\_datetime = datetime.strptime(start\_datetime, datetime\_prefix\_format)

if isinstance(end\_datetime, str):

end\_datetime = datetime.strptime(end\_datetime, datetime\_prefix\_format)

assert isinstance(start\_datetime, datetime)

assert isinstance(end\_datetime, datetime)

base\_path = preprocess\_path(base\_path)

def \_handle\_missing\_globs(pattern):

try:

return tf.io.gfile.glob(pattern)

except tf.errors.NotFoundError as e:

tf.logging.warning(e.message)

return []

# a set is used because there might be some repeated globs depending on dt\_prefix\_format

globs = {

os.path.join(base\_path, dt.strftime(datetime\_prefix\_format), '\*.%s' % extension)

for dt in rrule.rrule(

freq=rrule.HOURLY, interval=hour\_resolution, dtstart=start\_datetime, until=end\_datetime)

}

nested\_files = Parallel(n\_jobs=parallelism, backend='threading')(

delayed(\_handle\_missing\_globs)(p) for p in globs

)

flattened\_files = list(itertools.chain.from\_iterable(nested\_files))

if not flattened\_files:

error\_msg = "Files list is empty: base\_path={base\_path}, start\_datetime={start\_datetime}, end\_datetime={end\_datetime}".format(

base\_path=base\_path, start\_datetime=start\_datetime, end\_datetime=end\_datetime

)

raise OSError(error\_msg)

if sort:

flattened\_files = sorted(flattened\_files)

return flattened\_files

def limit\_sparse\_tensor\_size(sparse\_tf, input\_size\_bits, mask\_indices=True):

"""

Returns a ``tf.SparseTensor`` which is the input SparseTensor

limited to the specified input\_size\_bits

Args:

sparse\_tf:

twml.SparseTensor or tf.SparseTensor

input\_size\_bits:

The number of bits allocated to the input size.

Input size will be power(2,input\_size\_bits).

Note that twml.limit\_bits truncates any feature keys that

exceed the input size.

mask\_indices:

If mask indices is False; only the shape is changed. Defaults to True.

"""

if isinstance(sparse\_tf, twml.SparseTensor):

sparse\_tf = sparse\_tf.to\_tf()

if not isinstance(sparse\_tf, tf.SparseTensor):

raise TypeError('Input argument `sparse\_tf` should either be of type'

'twml.SparseTensor of tf.SparseTensor. Found type: {}'.

format(type(sparse\_tf)))

if mask\_indices:

indices = twml.limit\_bits(sparse\_tf.indices, input\_size\_bits)

else:

indices = sparse\_tf.indices

dense\_shape = tf.stack([sparse\_tf.dense\_shape[0], 1 << input\_size\_bits])

return tf.SparseTensor(indices=indices, values=sparse\_tf.values,

dense\_shape=dense\_shape)

def create\_module\_spec(mlp\_fn, mode, params, drop\_collections=None):

"""

Creates a standard tags\_and\_args which should be passed to the create\_module\_spec

spec = hub.create\_module\_spec(mlp\_fn, tags\_and\_args=tags\_and\_args).

Args:

module\_fn:

a function to build a graph for the Module.

mode:

mode in which the Estimator is run

params:

parameters passed to the Estimator

"""

import tensorflow\_hub as hub # noqa: F402

tags\_and\_args = [(set(), {"params": params, "mode": mode}), # serving graph

({"train"}, {"params": params, "mode": mode}) # training graph

]

spec = hub.create\_module\_spec(mlp\_fn, tags\_and\_args=tags\_and\_args, drop\_collections=drop\_collections)

return spec

def change\_name\_scope\_from\_dir(init\_scope\_name, final\_scope\_name, save\_dir):

"""

Changes the name of the saved scope to the desired name and saves it

to the same save\_dir.

Args:

init\_scope\_name:

initial scope name

final\_scope\_name:

desired (final) scope name

save\_dir:

directory which the scopes are saved

In the follwing section we:

- Read all the variables from the latest checkpoint.

- Make a copy of the variables with new name scope.

- Store both sets of variables into the latest checkpoint.

This essentially doubles up the size of the checkpoint.

But when a job is restarted after this part is done, the checkpoint size doubles again.

To avoid doing this, we create a copy in backup if a backup isn't found.

This allows us always read (from backup) and write same sized checkpoint files.

"""

# Create a backup\_checkpoints dir

backup\_dir = os.path.join(save\_dir, "change\_name\_scope\_backups")

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

latest\_checkpoint = tf.train.latest\_checkpoint(save\_dir)

if latest\_checkpoint is None:

raise OSError("No checkpoints found in save\_dir: %s" % save\_dir)

latest\_backup\_checkpoint = tf.train.latest\_checkpoint(backup\_dir)

if (latest\_backup\_checkpoint is None or

(os.path.basename(latest\_checkpoint) !=

os.path.basename(latest\_backup\_checkpoint))):

backup\_checkpoint(latest\_checkpoint, backup\_dir, empty\_backup=False)

variables = tf.train.list\_variables(backup\_dir)

with tf.Graph().as\_default(), tf.Session().as\_default() as sess:

new\_variables = []

for name, \_ in variables:

var = tf.train.load\_variable(backup\_dir, name)

# Append both the rename and the original variable

new\_variables.append(

tf.Variable(var, name=name.replace(init\_scope\_name, final\_scope\_name)))

new\_variables.append(tf.Variable(var, name=name))

# Save this to the checkpoint in the save\_dir

saver = tf.train.Saver(new\_variables)

sess.run(tf.global\_variables\_initializer())

saver.save(sess, latest\_checkpoint) # pylint: disable=no-member

def hub\_import(input, module, module\_name, trainable=False):

"""

Loads exported hub module.

Args:

input:

input to hub module

module:

module path

module\_name:

signature of the exported hub module

"""

import tensorflow\_hub as hub # noqa: F402

hub\_module = hub.Module(module, trainable=trainable)

output = hub\_module(input, signature=module\_name)

return output

def \_extract\_hash\_space\_bits(feature\_config):

"""

Extract Sparse Shapes for contrib.FeatureConfig.

Arguments:

feature\_config:

Feature Configuration of the type contrib.FeatureConfig

Returns:

Dictionary of tensor names and hash space bits.

"""

if not isinstance(feature\_config, twml.contrib.feature\_config.FeatureConfig):

fc\_type = type(feature\_config)

raise TypeError(f"Feature config must be of type contrib.FeatureConfig: {fc\_type}")

sparse\_shapes\_dict = {}

for config in feature\_config.sparse\_extraction\_configs:

sparse\_shapes\_dict[config.output\_name] = config.hash\_space\_bits

return sparse\_shapes\_dict

def fix\_shape\_sparse(features, feature\_config):

"""

Modifies the shape of features which are extracted using the hashing trick.

Features itself is changed by this function.

Arguments:

features:

Feature dictionary extracted by the feature config

feature\_config:

Feature Configuration of the type contrib.FeatureConfig

"""

if not isinstance(feature\_config, twml.contrib.feature\_config.FeatureConfig):

raise TypeError(f"Feature config must be of type contrib.FeatureConfig, currently of {type(feature\_config)}")

sparse\_shape = \_extract\_hash\_space\_bits(feature\_config)

if not isinstance(features, dict):

raise TypeError(f"features must be of dictionary type, it is of {type(features)} type")

for key in set(features) & set(sparse\_shape):

features[key] = limit\_sparse\_tensor\_size(features[key], sparse\_shape[key], mask\_indices=False)

def touch\_file\_in\_dir(directory, filename):

"""

Creates a file named filename in directory.

Arguments:

filename: (str)

directory: (str)

"""

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

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

f.write("")

def file\_exist\_in\_dir(directory: str, filename: str) -> bool:

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

return tf.io.gfile.exists(file\_path)

def copy\_to\_local(remote, local, filename, overwrite=False):

"""Function to file from remote directory to local directory."""

assert "hdfs://" not in local

tf.io.gfile.makedirs(local)

return tf.io.gfile.copy(

os.path.join(remote, filename),

os.path.join(local, filename),

overwrite=overwrite,

)

def copy\_recursive(src, dst, overwrite=False):

"""

Function to copy a directory recursively.

Arguments:

src: Source directory.

dst: Destination directory.

overwrite: Specifies if files are to be overwritten if they exist.

"""

src = src.rstrip("/")

dst = dst.rstrip("/")

for dirname, subdirs, files in tf.io.gfile.walk(src):

dst\_dirname = dirname.replace(src, dst)

tf.io.gfile.makedirs(dst\_dirname)

for f in files:

src\_f = os.path.join(dirname, f)

dst\_f = os.path.join(dst\_dirname, f)

tf.logging.info(f"Copying {src\_f} to {dst\_f}")

tf.io.gfile.copy(src\_f, dst\_f, overwrite=overwrite)

def delete\_file\_or\_dir(path):

"""

Delete the file or directory given by `path`

Arguments:

path:

string indicating path of file or directory to remove

"""

if tf.io.gfile.isdir(path):

tf.io.gfile.rmtree(path)

else:

tf.io.gfile.remove(path)

def get\_distributed\_training\_job\_path():

"""

Function to get distributed training job path.

Note: distributed training has three jobs, one parameter server job,

one worker job and one evaluator job. All of these three jobs' name

share a common base job name.

"""

job\_path = AuroraPath(dc=os.environ.get("TWML\_JOB\_CLUSTER"),

role=os.environ.get("TWML\_JOB\_ROLE"),

env=os.environ.get("TWML\_JOB\_ENV"),

job\_name=os.environ.get("TWML\_DISTRIBUTED\_BASE\_JOBNAME"))

return job\_path

def do\_every\_n\_steps(action, num\_steps):

"""

Execute a sequence of TensorFlow operations only once in a while.

Specifically, `action` is performed if `global\_step` is a

multiple of `num\_steps`

Args:

action: callable to be performed at regular intervals. This callable

must return a TF op with no output tensors.

num\_steps: period of performing the action, as measured

in number of training steps

Returns:

A TensorFlow op with no output tensors, like a tf.print() or tf.no\_op().

You must use tf.control\_dependencies() to execute the op.

"""

global\_step = tf.train.get\_or\_create\_global\_step()

condition = tf.math.equal(tf.math.floormod(global\_step, num\_steps), 0)

return tf.cond(condition, action, lambda: tf.no\_op())