from collections import OrderedDict

import json

import os

from os.path import join

from twitter.magicpony.common import file\_access

import twml

from .model\_utils import read\_config

import numpy as np

from scipy import stats

import tensorflow.compat.v1 as tf

# checkstyle: noqa

def get\_model\_type\_to\_tensors\_to\_change\_axis():

model\_type\_to\_tensors\_to\_change\_axis = {

"magic\_recs/model/batch\_normalization/beta": ([0], "continuous"),

"magic\_recs/model/batch\_normalization/gamma": ([0], "continuous"),

"magic\_recs/model/batch\_normalization/moving\_mean": ([0], "continuous"),

"magic\_recs/model/batch\_normalization/moving\_stddev": ([0], "continuous"),

"magic\_recs/model/batch\_normalization/moving\_variance": ([0], "continuous"),

"magic\_recs/model/batch\_normalization/renorm\_mean": ([0], "continuous"),

"magic\_recs/model/batch\_normalization/renorm\_stddev": ([0], "continuous"),

"magic\_recs/model/logits/EngagementGivenOONC\_logits/clem\_net\_1/block2\_4/channel\_wise\_dense\_4/kernel": (

[1],

"all",

),

"magic\_recs/model/logits/OONC\_logits/clem\_net/block2/channel\_wise\_dense/kernel": ([1], "all"),

}

return model\_type\_to\_tensors\_to\_change\_axis

def mkdirp(dirname):

if not tf.io.gfile.exists(dirname):

tf.io.gfile.makedirs(dirname)

def rename\_dir(dirname, dst):

file\_access.hdfs.mv(dirname, dst)

def rmdir(dirname):

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

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

tf.io.gfile.rmtree(dirname)

else:

tf.io.gfile.remove(dirname)

def get\_var\_dict(checkpoint\_path):

checkpoint = tf.train.get\_checkpoint\_state(checkpoint\_path)

var\_dict = OrderedDict()

with tf.Session() as sess:

all\_var\_list = tf.train.list\_variables(checkpoint\_path)

for var\_name, \_ in all\_var\_list:

# Load the variable

var = tf.train.load\_variable(checkpoint\_path, var\_name)

var\_dict[var\_name] = var

return var\_dict

def get\_continunous\_mapping\_from\_feat\_list(old\_feature\_list, new\_feature\_list):

"""

get var\_ind for old\_feature and corresponding var\_ind for new\_feature

"""

new\_var\_ind, old\_var\_ind = [], []

for this\_new\_id, this\_new\_name in enumerate(new\_feature\_list):

if this\_new\_name in old\_feature\_list:

this\_old\_id = old\_feature\_list.index(this\_new\_name)

new\_var\_ind.append(this\_new\_id)

old\_var\_ind.append(this\_old\_id)

return np.asarray(old\_var\_ind), np.asarray(new\_var\_ind)

def get\_continuous\_mapping\_from\_feat\_dict(old\_feature\_dict, new\_feature\_dict):

"""

get var\_ind for old\_feature and corresponding var\_ind for new\_feature

"""

old\_cont = old\_feature\_dict["continuous"]

old\_bin = old\_feature\_dict["binary"]

new\_cont = new\_feature\_dict["continuous"]

new\_bin = new\_feature\_dict["binary"]

\_dummy\_sparse\_feat = [f"sparse\_feature\_{\_idx}" for \_idx in range(100)]

cont\_old\_var\_ind, cont\_new\_var\_ind = get\_continunous\_mapping\_from\_feat\_list(old\_cont, new\_cont)

all\_old\_var\_ind, all\_new\_var\_ind = get\_continunous\_mapping\_from\_feat\_list(

old\_cont + old\_bin + \_dummy\_sparse\_feat, new\_cont + new\_bin + \_dummy\_sparse\_feat

)

\_res = {

"continuous": (cont\_old\_var\_ind, cont\_new\_var\_ind),

"all": (all\_old\_var\_ind, all\_new\_var\_ind),

}

return \_res

def warm\_start\_from\_var\_dict(

old\_ckpt\_path,

var\_ind\_dict,

output\_dir,

new\_len\_var,

var\_to\_change\_dict\_fn=get\_model\_type\_to\_tensors\_to\_change\_axis,

):

"""

Parameters:

old\_ckpt\_path (str): path to the old checkpoint path

new\_var\_ind (array of int): index to overlapping features in new var between old and new feature list.

old\_var\_ind (array of int): index to overlapping features in old var between old and new feature list.

output\_dir (str): dir that used to write modified checkpoint

new\_len\_var ({str:int}): number of feature in the new feature list.

var\_to\_change\_dict\_fn (dict): A function to get the dictionary of format {var\_name: dim\_to\_change}

"""

old\_var\_dict = get\_var\_dict(old\_ckpt\_path)

ckpt\_file\_name = os.path.basename(old\_ckpt\_path)

mkdirp(output\_dir)

output\_path = join(output\_dir, ckpt\_file\_name)

tensors\_to\_change = var\_to\_change\_dict\_fn()

tf.compat.v1.reset\_default\_graph()

with tf.Session() as sess:

var\_name\_shape\_list = tf.train.list\_variables(old\_ckpt\_path)

count = 0

for var\_name, var\_shape in var\_name\_shape\_list:

old\_var = old\_var\_dict[var\_name]

if var\_name in tensors\_to\_change.keys():

\_info\_tuple = tensors\_to\_change[var\_name]

dims\_to\_remove\_from, var\_type = \_info\_tuple

new\_var\_ind, old\_var\_ind = var\_ind\_dict[var\_type]

this\_shape = list(old\_var.shape)

for this\_dim in dims\_to\_remove\_from:

this\_shape[this\_dim] = new\_len\_var[var\_type]

stddev = np.std(old\_var)

truncated\_norm\_generator = stats.truncnorm(-0.5, 0.5, loc=0, scale=stddev)

size = np.prod(this\_shape)

new\_var = truncated\_norm\_generator.rvs(size).reshape(this\_shape)

new\_var = new\_var.astype(old\_var.dtype)

new\_var = copy\_feat\_based\_on\_mapping(

new\_var, old\_var, dims\_to\_remove\_from, new\_var\_ind, old\_var\_ind

)

count = count + 1

else:

new\_var = old\_var

var = tf.Variable(new\_var, name=var\_name)

assert count == len(tensors\_to\_change.keys()), "not all variables are exchanged.\n"

saver = tf.train.Saver()

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

saver.save(sess, output\_path)

return output\_path

def copy\_feat\_based\_on\_mapping(new\_array, old\_array, dims\_to\_remove\_from, new\_var\_ind, old\_var\_ind):

if dims\_to\_remove\_from == [0, 1]:

for this\_new\_ind, this\_old\_ind in zip(new\_var\_ind, old\_var\_ind):

new\_array[this\_new\_ind, new\_var\_ind] = old\_array[this\_old\_ind, old\_var\_ind]

elif dims\_to\_remove\_from == [0]:

new\_array[new\_var\_ind] = old\_array[old\_var\_ind]

elif dims\_to\_remove\_from == [1]:

new\_array[:, new\_var\_ind] = old\_array[:, old\_var\_ind]

else:

raise RuntimeError(f"undefined dims\_to\_remove\_from pattern: ({dims\_to\_remove\_from})")

return new\_array

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 or 'json'. When decode='json', contents is decoded

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

"""

graph = tf.Graph()

with graph.as\_default():

read = tf.read\_file(filename)

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

contents = sess.run(read)

if not isinstance(contents, str):

contents = contents.decode()

if decode == "json":

contents = json.loads(contents)

return contents

def read\_feat\_list\_from\_disk(file\_path):

return read\_file(file\_path, decode="json")

def get\_feature\_list\_for\_light\_ranking(feature\_list\_path, data\_spec\_path):

feature\_list = read\_config(feature\_list\_path).items()

string\_feat\_list = [f[0] for f in feature\_list if f[1] != "S"]

feature\_config\_builder = twml.contrib.feature\_config.FeatureConfigBuilder(

data\_spec\_path=data\_spec\_path

)

feature\_config\_builder = feature\_config\_builder.extract\_feature\_group(

feature\_regexes=string\_feat\_list,

group\_name="continuous",

default\_value=-1,

type\_filter=["CONTINUOUS"],

)

feature\_config = feature\_config\_builder.build()

feature\_list = feature\_config\_builder.\_feature\_group\_extraction\_configs[0].feature\_map[

"CONTINUOUS"

]

return feature\_list

def get\_feature\_list\_for\_heavy\_ranking(feature\_list\_path, data\_spec\_path):

feature\_list = read\_config(feature\_list\_path).items()

string\_feat\_list = [f[0] for f in feature\_list if f[1] != "S"]

feature\_config\_builder = twml.contrib.feature\_config.FeatureConfigBuilder(

data\_spec\_path=data\_spec\_path

)

feature\_config\_builder = feature\_config\_builder.extract\_feature\_group(

feature\_regexes=string\_feat\_list,

group\_name="continuous",

default\_value=-1,

type\_filter=["CONTINUOUS"],

)

feature\_config\_builder = feature\_config\_builder.extract\_feature\_group(

feature\_regexes=string\_feat\_list,

group\_name="binary",

default\_value=False,

type\_filter=["BINARY"],

)

feature\_config\_builder = feature\_config\_builder.build()

continuous\_feature\_list = feature\_config\_builder.\_feature\_group\_extraction\_configs[0].feature\_map[

"CONTINUOUS"

]

binary\_feature\_list = feature\_config\_builder.\_feature\_group\_extraction\_configs[1].feature\_map[

"BINARY"

]

return {"continuous": continuous\_feature\_list, "binary": binary\_feature\_list}

def warm\_start\_checkpoint(

old\_best\_ckpt\_folder,

old\_feature\_list\_path,

feature\_allow\_list\_path,

data\_spec\_path,

output\_ckpt\_folder,

\*args,

):

"""

Reads old checkpoint and the old feature list, and create a new ckpt warm started from old ckpt using new features .

Arguments:

old\_best\_ckpt\_folder:

path to the best\_checkpoint\_folder for old model

old\_feature\_list\_path:

path to the json file that stores the list of continuous features used in old models.

feature\_allow\_list\_path:

yaml file that contain the feature allow list.

data\_spec\_path:

path to the data\_spec file

output\_ckpt\_folder:

folder that contains the modified ckpt.

Returns:

path to the modified ckpt."""

old\_ckpt\_path = tf.train.latest\_checkpoint(old\_best\_ckpt\_folder, latest\_filename=None)

new\_feature\_dict = get\_feature\_list(feature\_allow\_list\_path, data\_spec\_path)

old\_feature\_dict = read\_feat\_list\_from\_disk(old\_feature\_list\_path)

var\_ind\_dict = get\_continuous\_mapping\_from\_feat\_dict(new\_feature\_dict, old\_feature\_dict)

new\_len\_var = {

"continuous": len(new\_feature\_dict["continuous"]),

"all": len(new\_feature\_dict["continuous"] + new\_feature\_dict["binary"]) + 100,

}

warm\_started\_ckpt\_path = warm\_start\_from\_var\_dict(

old\_ckpt\_path,

var\_ind\_dict,

output\_dir=output\_ckpt\_folder,

new\_len\_var=new\_len\_var,

)

return warm\_started\_ckpt\_path