# coding=utf-8

# Copyright 2018 Google LLC & Hwalsuk Lee.

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

"""Implementation of Self-Supervised GAN with auxiliary rotation loss."""

from \_\_future\_\_ import absolute\_import

from \_\_future\_\_ import division

from \_\_future\_\_ import print\_function

from absl import flags

from absl import logging

import sys

sys.path.append(‘/content/drive/My Drive/drive/gan/gan\_git/’)

from compare\_gan.architectures import arch\_ops as ops

from compare\_gan.gans import loss\_lib

from compare\_gan.gans import modular\_gan

from compare\_gan.gans import utils

import gin

import numpy as np

import tensorflow as tf

FLAGS = flags.FLAGS

NUM\_ROTATIONS = 4

# pylint: disable=not-callable

@gin.configurable(blacklist=["kwargs"])

class S3GAN(modular\_gan.ModularGAN):

"""S3GAN which enables auxiliary heads for the modular GAN."""

def \_\_init\_\_(self, self\_supervision="rotation",

rotated\_batch\_fraction=gin.REQUIRED,

weight\_rotation\_loss\_d=1.0,

weight\_rotation\_loss\_g=0.2,

project\_y=False,

use\_predictor=False,

use\_soft\_pred=False,

weight\_class\_loss=1.0,

use\_soft\_labels=False,

\*\*kwargs):

"""Instantiates the S3GAN.

Args:

self\_supervision: One of [rotation\_gan, None].

rotated\_batch\_fraction: This must be a divisor of the total batch size.

rotations of each images on each TPU core. For GPU training #CORES is 1.

weight\_rotation\_loss\_d: Weight for the rotation loss for the discriminator

on real images.

weight\_rotation\_loss\_g: Weight for the rotation loss for the generator

on fake images.

project\_y: Boolean, whether an embedding layer as in variant 1) should be

used.

use\_predictor: Boolean, whether a predictor (classifier) should be used.

use\_soft\_pred: Boolean, whether soft labels should be used for the

predicted label vectors in 1).

weight\_class\_loss: weight of the (predictor) classification loss added to

the discriminator loss.

use\_soft\_labels: Boolean, if true assumes the labels passed for real

examples are soft labels and accordingly does not transform

\*\*kwargs: Additional arguments passed to `ModularGAN` constructor.

"""

super(S3GAN, self).\_\_init\_\_(\*\*kwargs)

if use\_predictor and not project\_y:

raise ValueError("Using predictor requires projection.")

assert self\_supervision in {"none", "rotation"}

self.\_self\_supervision = self\_supervision

self.\_rotated\_batch\_fraction = rotated\_batch\_fraction

self.\_weight\_rotation\_loss\_d = weight\_rotation\_loss\_d

self.\_weight\_rotation\_loss\_g = weight\_rotation\_loss\_g

self.\_project\_y = project\_y

self.\_use\_predictor = use\_predictor

self.\_use\_soft\_pred = use\_soft\_pred

self.\_weight\_class\_loss = weight\_class\_loss

self.\_use\_soft\_labels = use\_soft\_labels

# To safe memory ModularGAN supports feeding real and fake samples

# separately through the discriminator. S3GAN does not support this to

# avoid additional additional complexity in create\_loss().

assert not self.\_deprecated\_split\_disc\_calls, \

"Splitting discriminator calls is not supported in S3GAN."

def discriminator\_with\_additonal\_heads(self, x, y, is\_training):

"""Discriminator architecture with additional heads.

Possible heads built on top of feature representation of the discriminator:

(1) Classify the image to the correct class.

(2) Classify the rotation of the image.

Args:

x: An input image tensor.

y: One-hot encoded label. Passing all zeros implies no label was passed.

is\_training: boolean, whether or not it is a training call.

Returns:

Tuple of 5 Tensors: (1) discriminator predictions (in [0, 1]), (2) the

corresponding logits, (3) predictions (logits) of the rotation of x from

the auxiliary head, (4) logits of the class prediction from the auxiliary

head, (5) Indicator vector identifying whether y contained a label or -1.

"""

d\_probs, d\_logits, x\_rep = self.discriminator(

x, y=y, is\_training=is\_training)

use\_sn = self.discriminator.\_spectral\_norm # pylint: disable=protected-access

is\_label\_available = tf.cast(tf.cast(

tf.reduce\_sum(y, axis=1, keepdims=True), tf.float32) > 0.5, tf.float32)

assert x\_rep.shape.ndims == 2, x\_rep.shape

# Predict the rotation of the image.

rotation\_logits = None

if "rotation" in self.\_self\_supervision:

with tf.variable\_scope("discriminator\_rotation", reuse=tf.AUTO\_REUSE):

rotation\_logits = ops.linear(

x\_rep,

NUM\_ROTATIONS,

scope="score\_classify",

use\_sn=use\_sn)

logging.info("[Discriminator] rotation head %s -> %s",

x\_rep.shape, rotation\_logits)

if not self.\_project\_y:

return d\_probs, d\_logits, rotation\_logits, None, is\_label\_available

# Predict the class of the image.

aux\_logits = None

if self.\_use\_predictor:

with tf.variable\_scope("discriminator\_predictor", reuse=tf.AUTO\_REUSE):

aux\_logits = ops.linear(x\_rep, y.shape[1], use\_bias=True,

scope="predictor\_linear", use\_sn=use\_sn)

# Apply the projection discriminator if needed.

if self.\_use\_soft\_pred:

y\_predicted = tf.nn.softmax(aux\_logits)

else:

y\_predicted = tf.one\_hot(

tf.arg\_max(aux\_logits, 1), aux\_logits.shape[1])

y = (1.0 - is\_label\_available) \* y\_predicted + is\_label\_available \* y

y = tf.stop\_gradient(y)

logging.info("[Discriminator] %s -> aux\_logits=%s, y\_predicted=%s",

aux\_logits.shape, aux\_logits.shape, y\_predicted.shape)

class\_embedding = self.get\_class\_embedding(

y=y, embedding\_dim=x\_rep.shape[-1].value, use\_sn=use\_sn)

d\_logits += tf.reduce\_sum(class\_embedding \* x\_rep, axis=1, keepdims=True)

d\_probs = tf.nn.sigmoid(d\_logits)

return d\_probs, d\_logits, rotation\_logits, aux\_logits, is\_label\_available

def get\_class\_embedding(self, y, embedding\_dim, use\_sn):

with tf.variable\_scope("discriminator\_projection", reuse=tf.AUTO\_REUSE):

# We do not use ops.linear() below since it does not have an option to

# override the initializer.

kernel = tf.get\_variable(

"kernel", [y.shape[1], embedding\_dim], tf.float32,

initializer=tf.initializers.glorot\_normal())

if use\_sn:

kernel = ops.spectral\_norm(kernel)

embedded\_y = tf.matmul(y, kernel)

logging.info("[Discriminator] embedded\_y for projection: %s",

embedded\_y.shape)

return embedded\_y

def merge\_with\_rotation\_data(self, real, fake, real\_labels, fake\_labels,

num\_rot\_examples):

"""Returns the original data concatenated with the rotated version."""

# Put all rotation angles in a single batch, the first batch\_size are

# the original up-right images, followed by rotated\_batch\_size \* 3

# rotated images with 3 different angles. For NUM\_ROTATIONS=4 and

# num\_rot\_examples=2 we have labels\_rotated [0, 0, 1, 1, 2, 2, 3, 3].

real\_to\_rot, fake\_to\_rot = (

real[-num\_rot\_examples:], fake[-num\_rot\_examples:])

real\_rotated = utils.rotate\_images(real\_to\_rot, rot90\_scalars=(1, 2, 3))

fake\_rotated = utils.rotate\_images(fake\_to\_rot, rot90\_scalars=(1, 2, 3))

all\_features = tf.concat([real, real\_rotated, fake, fake\_rotated], 0)

all\_labels = None

if self.conditional:

real\_rotated\_labels = tf.tile(real\_labels[-num\_rot\_examples:], [3, 1])

fake\_rotated\_labels = tf.tile(fake\_labels[-num\_rot\_examples:], [3, 1])

all\_labels = tf.concat([real\_labels, real\_rotated\_labels,

fake\_labels, fake\_rotated\_labels], 0)

return all\_features, all\_labels

def create\_loss(self, features, labels, params, is\_training=True):

"""Build the loss tensors for discriminator and generator.

This method will set self.d\_loss and self.g\_loss.

Args:

features: Optional dictionary with inputs to the model ("images" should

contain the real images and "z" the noise for the generator).

labels: Tensor will labels. These are class indices. Use

self.\_get\_one\_hot\_labels(labels) to get a one hot encoded tensor.

params: Dictionary with hyperparameters passed to TPUEstimator.

Additional TPUEstimator will set 3 keys: `batch\_size`, `use\_tpu`,

`tpu\_context`. `batch\_size` is the batch size for this core.

is\_training: If True build the model in training mode. If False build the

model for inference mode (e.g. use trained averages for batch norm).

Raises:

ValueError: If set of meta/hyper parameters is not supported.

"""

real\_images = features["images"]

if self.conditional:

if self.\_use\_soft\_labels:

assert labels.shape[1] == self.\_dataset.num\_classes, \

("Need soft labels of dimension {} but got dimension {}".format(

self.\_dataset.num\_classes, labels.shape[1]))

real\_labels = labels

else:

real\_labels = self.\_get\_one\_hot\_labels(labels)

fake\_labels = self.\_get\_one\_hot\_labels(features["sampled\_labels"])

if self.\_experimental\_joint\_gen\_for\_disc:

assert "generated" in features

fake\_images = features["generated"]

else:

logging.warning("Computing fake images for every sub step separately.")

fake\_images = self.generator(

features["z"], y=fake\_labels, is\_training=is\_training)

bs = real\_images.shape[0].value

if self.\_self\_supervision:

assert bs % self.\_rotated\_batch\_fraction == 0, (

"Rotated batch fraction is invalid: %d doesn't divide %d" %

self.\_rotated\_batch\_fraction, bs)

rotated\_bs = bs // self.\_rotated\_batch\_fraction

num\_rot\_examples = rotated\_bs // NUM\_ROTATIONS

logging.info("bs=%s, rotated\_bs=%s, num\_rot\_examples=%s", bs, rotated\_bs,

num\_rot\_examples)

assert num\_rot\_examples > 0

# Append the data obtained by rotating the last 'num\_rotated\_samples'

# from the true and the fake data.

if self.\_self\_supervision == "rotation":

assert num\_rot\_examples <= bs, (num\_rot\_examples, bs)

all\_features, all\_labels = self.merge\_with\_rotation\_data(

real\_images, fake\_images, real\_labels, fake\_labels, num\_rot\_examples)

else:

all\_features = tf.concat([real\_images, fake\_images], 0)

all\_labels = None

if self.conditional:

all\_labels = tf.concat([real\_labels, fake\_labels], axis=0)

d\_predictions, d\_logits, rot\_logits, aux\_logits, is\_label\_available = (

self.discriminator\_with\_additonal\_heads(

x=all\_features, y=all\_labels, is\_training=is\_training))

expected\_batch\_size = 2 \* bs

if self.\_self\_supervision == "rotation":

expected\_batch\_size += 2 \* (NUM\_ROTATIONS - 1) \* num\_rot\_examples

if d\_logits.shape[0].value != expected\_batch\_size:

raise ValueError("Batch size unexpected: got %r expected %r" % (

d\_logits.shape[0].value, expected\_batch\_size))

prob\_real, prob\_fake = tf.split(d\_predictions, 2)

prob\_real, prob\_fake = prob\_real[:bs], prob\_fake[:bs]

logits\_real, logits\_fake = tf.split(d\_logits, 2)

logits\_real, logits\_fake = logits\_real[:bs], logits\_fake[:bs]

# Get the true/fake GAN loss.

self.d\_loss, \_, \_, self.g\_loss = loss\_lib.get\_losses(

d\_real=prob\_real, d\_fake=prob\_fake,

d\_real\_logits=logits\_real, d\_fake\_logits=logits\_fake)

# At this point we have the classic GAN loss with possible regularization.

# We now add the rotation loss and summaries if required.

if self.\_self\_supervision == "rotation":

# Extract logits for the rotation task.

rot\_real\_logits, rot\_fake\_logits = tf.split(rot\_logits, 2)

rot\_real\_logits = rot\_real\_logits[-rotated\_bs:]

rot\_fake\_logits = rot\_fake\_logits[-rotated\_bs:]

labels\_rotated = tf.constant(np.repeat(

np.arange(NUM\_ROTATIONS, dtype=np.int32), num\_rot\_examples))

rot\_onehot = tf.one\_hot(labels\_rotated, NUM\_ROTATIONS)

rot\_real\_logp = tf.log(tf.nn.softmax(rot\_real\_logits) + 1e-10)

rot\_fake\_logp = tf.log(tf.nn.softmax(rot\_fake\_logits) + 1e-10)

real\_loss = -tf.reduce\_mean(tf.reduce\_sum(rot\_onehot \* rot\_real\_logp, 1))

fake\_loss = -tf.reduce\_mean(tf.reduce\_sum(rot\_onehot \* rot\_fake\_logp, 1))

self.d\_loss += real\_loss \* self.\_weight\_rotation\_loss\_d

self.g\_loss += fake\_loss \* self.\_weight\_rotation\_loss\_g

rot\_real\_labels = tf.one\_hot(

tf.arg\_max(rot\_real\_logits, 1), NUM\_ROTATIONS)

rot\_fake\_labels = tf.one\_hot(

tf.arg\_max(rot\_fake\_logits, 1), NUM\_ROTATIONS)

accuracy\_real = tf.metrics.accuracy(rot\_onehot, rot\_real\_labels)

accuracy\_fake = tf.metrics.accuracy(rot\_onehot, rot\_fake\_labels)

self.\_tpu\_summary.scalar("loss/real\_loss", real\_loss)

self.\_tpu\_summary.scalar("loss/fake\_loss", fake\_loss)

self.\_tpu\_summary.scalar("accuracy/real", accuracy\_real)

self.\_tpu\_summary.scalar("accuracy/fake", accuracy\_fake)

# Training the predictor on the features of real data and real labels.

if self.\_use\_predictor:

real\_aux\_logits, \_ = tf.split(aux\_logits, 2)

real\_aux\_logits = real\_aux\_logits[:bs]

is\_label\_available, \_ = tf.split(is\_label\_available, 2)

is\_label\_available = tf.squeeze(is\_label\_available[:bs])

class\_loss\_real = tf.losses.softmax\_cross\_entropy(

real\_labels, real\_aux\_logits, weights=is\_label\_available)

# Add the loss to the discriminator

self.d\_loss += self.\_weight\_class\_loss \* class\_loss\_real

self.\_tpu\_summary.scalar("loss/class\_loss\_real", class\_loss\_real)

self.\_tpu\_summary.scalar("label\_frac", tf.reduce\_mean(is\_label\_available))