**Practical Assignment**

**Objective: - Object Detection with MSCOCO**

Microsoft is in this game also with their Common Objects in Context (COCO) dataset. Containing ~200K images, it’s relatively small but what makes it stand out are its challenges that come associated with the additional features it provides for each image.

COCO challenges are also held annually. But each year’s challenge is slightly different. This year the challenge has four tracks:

1. Object segmentation
2. Panoptic segmentation
3. Keypoint detection
4. DensePose task

**Dataset Link: -** Dataset is pretty big. SO we do not want to train it completely. So please extract any 10 classes images and annotations that you like. Then train it.

Link :-<https://cocodataset.org/#download>

**Task: -** Create a Web Application using Flask. Use the end user should be able to upload an image and get results with the prediction score. Use any CNN architecture launched after 2017.

**Deployment: -** Any Free Platform(Try to look out for free options.)

**Assignment Submission: -** Only submit the hosted app link. OR GitHub Link

# Copyright 2024 The Orbit Authors. All Rights Reserved.

#

# 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.

"""Tests for orbit.standard\_runner."""

from absl.testing import parameterized

from orbit import standard\_runner

from orbit import utils

import tensorflow as tf, tf\_keras

def dataset\_fn(input\_context=None):

del input\_context

def dummy\_data(\_):

return tf.zeros((1, 1), dtype=tf.float32)

dataset = tf.data.Dataset.range(1)

dataset = dataset.repeat()

dataset = dataset.map(

dummy\_data, num\_parallel\_calls=tf.data.experimental.AUTOTUNE)

return dataset

class TestTrainer(standard\_runner.StandardTrainer):

"""A StandardTrainer subclass for tests."""

def \_\_init\_\_(self, options=None):

self.strategy = tf.distribute.get\_strategy()

self.global\_step = utils.create\_global\_step()

dataset = self.strategy.distribute\_datasets\_from\_function(dataset\_fn)

super().\_\_init\_\_(train\_dataset=dataset, options=options)

def train\_loop\_begin(self):

self.global\_step.assign(0)

def train\_step(self, iterator):

def replica\_step(\_):

self.global\_step.assign\_add(1)

self.strategy.run(replica\_step, args=(next(iterator),))

def train\_loop\_end(self):

return self.global\_step.numpy()

class TestEvaluator(standard\_runner.StandardEvaluator):

"""A StandardEvaluator subclass for tests."""

def \_\_init\_\_(self, options=None):

self.strategy = tf.distribute.get\_strategy()

self.global\_step = utils.create\_global\_step()

dataset = self.strategy.distribute\_datasets\_from\_function(dataset\_fn)

super().\_\_init\_\_(eval\_dataset=dataset, options=options)

def eval\_begin(self):

self.global\_step.assign(0)

def eval\_step(self, iterator):

def replica\_step(\_):

self.global\_step.assign\_add(1)

self.strategy.run(replica\_step, args=(next(iterator),))

def eval\_end(self):

return self.global\_step.numpy()

class TestEvaluatorWithOutputsAggregation(standard\_runner.StandardEvaluator):

"""A StandardEvaluator subclass for tests."""

def \_\_init\_\_(self, options=None):

self.strategy = tf.distribute.get\_strategy()

dataset = self.strategy.distribute\_datasets\_from\_function(

lambda \_: tf.data.Dataset.range(10))

super().\_\_init\_\_(eval\_dataset=dataset, options=options)

def eval\_begin(self):

return {"logits": tf.constant((0.0,))}

def eval\_reduce(self, state, step\_outputs):

state["logits"] = tf.concat([state["logits"], step\_outputs], 0)

return state

def eval\_step(self, iterator):

def replica\_step(x):

x = tf.cast(x, tf.float32)

return tf.reduce\_sum(x)

return self.strategy.experimental\_local\_results(

self.strategy.run(replica\_step, args=(next(iterator),)))

def eval\_end(self, outputs):

return tf.reduce\_sum(outputs["logits"])

class StandardRunnerTest(parameterized.TestCase):

def test\_default\_trainer(self):

trainer = TestTrainer()

self.assertEqual(trainer.train(tf.constant(10)), 10)

def test\_trainer\_with\_tpu\_summary\_optimization(self):

options = standard\_runner.StandardTrainerOptions(

use\_tpu\_summary\_optimization=True)

trainer = TestTrainer(options)

self.assertEqual(trainer.train(tf.constant(10)), 10)

@parameterized.named\_parameters(("use\_tf\_while\_loop", True), ("", False))

def test\_default\_evaluator(self, use\_tf\_while\_loop):

options = standard\_runner.StandardEvaluatorOptions(

use\_tf\_while\_loop=use\_tf\_while\_loop)

evaluator = TestEvaluator(options)

self.assertEqual(evaluator.evaluate(tf.constant(10)), 10)

@parameterized.named\_parameters(("use\_tf\_while\_loop", True), ("", False))

def test\_evaluator\_with\_outputs\_aggregation(self, use\_tf\_while\_loop):

options = standard\_runner.StandardEvaluatorOptions(

use\_tf\_while\_loop=use\_tf\_while\_loop)

evaluator = TestEvaluatorWithOutputsAggregation(options)

self.assertEqual(evaluator.evaluate(tf.constant(10)), 45)

@parameterized.named\_parameters(

("recreate\_iterator\_for\_each\_eval", True, 10, 10),

("not\_recreate\_iterator\_for\_each\_eval", False, 10, 35))

def test\_evaluator\_with\_repeat\_dataset(self, recreate\_iterator\_for\_each\_eval,

sum\_for\_1st\_time, sum\_for\_2nd\_time):

options = standard\_runner.StandardEvaluatorOptions(

recreate\_iterator\_for\_each\_eval=recreate\_iterator\_for\_each\_eval)

evaluator = TestEvaluatorWithOutputsAggregation(options)

self.assertEqual(evaluator.evaluate(tf.constant(5)), sum\_for\_1st\_time)

self.assertEqual(evaluator.evaluate(tf.constant(5)), sum\_for\_2nd\_time)

if \_\_name\_\_ == "\_\_main\_\_":

tf.test.main()