""" This file contains tf.train.SessionRunHooks defined by TWML """

from datetime import datetime

import json

import operator

import os

from absl import logging

import numpy as np

import tensorflow.compat.v1 as tf

from tensorflow.python.training.basic\_session\_run\_hooks import NeverTriggerTimer, SecondOrStepTimer

import twml

class StepProgressHook(tf.train.SessionRunHook):

"""Hook that displays a progress bar to monitor global step progress """

def \_\_init\_\_(self, max\_step):

"""

Initializes a `StepProgressHook`.

This hook displays a progress bar for max\_steps.

Note that this hook only works for training and calibration.

Args:

max\_steps:

maximum steps to monitor in progress bar.

When this many steps is reached, the progress bar will be full.

"""

self.\_max\_step = max\_step

self.\_start\_step = 0

self.\_global\_step\_tensor = None

self.\_progress\_bar = None

def begin(self):

""" sets the global\_step\_tensor """

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

if self.\_global\_step\_tensor is None:

raise RuntimeError("Global step should be created to use StepProgressHook.")

def after\_create\_session(self, session, coord):

""" creates the progress bar and keeps track of the first global step upon session creation """

global\_step = session.run(self.\_global\_step\_tensor)

self.\_start\_step = global\_step

self.\_progress\_bar = tf.keras.utils.Progbar(self.\_max\_step)

def before\_run(self, run\_context): # pylint: disable=unused-argument

""" invoked before calling session.run """

return tf.train.SessionRunArgs(self.\_global\_step\_tensor)

def after\_run(self, run\_context, run\_values):

""" invoked after run is called. Updates the progress bar. """

step = run\_context.session.run(self.\_global\_step\_tensor)

self.\_progress\_bar.update(step - self.\_start\_step)

class GetMetricsHook(tf.train.SessionRunHook):

"""

Hook used to obtain evaluation metrics.

Typically used for early-stopping by obtaining the value of a

metric at the end of an epoch.

Note that the metric tensor and its commensurate update Op

are responsible for aggregating the metric during the session

(one session per epoch). Used for evaluation.

"""

def \_\_init\_\_(self, get\_metrics\_fn):

"""GetMetricsHook constructor.

Args:

get\_metrics\_fn:

Function that returns a dict mapping metric keys to

tensors as a tf.Tensor.

See Trainer.learn for an example use-case.

"""

self.\_get\_metrics\_fn = get\_metrics\_fn

self.\_metric\_tensors = None

self.metric\_values = None

def begin(self):

""" sets the global\_step\_tensor and metric tensor"""

self.\_metric\_tensors = self.\_get\_metrics\_fn()

assert isinstance(self.\_metric\_tensors, dict)

def end(self, session):

self.metric\_values = session.run(self.\_metric\_tensors)

class EarlyStopHook(GetMetricsHook):

"""

A GetMetricsHook augmented with early-stopping logic for use

within the Trainer.learn method.

"""

def \_\_init\_\_(self,

metric,

patience,

minimize,

get\_estimator\_spec\_fn,

checkpoint\_dir,

file\_path=None,

exit\_on\_end=True,

start\_epoch=0,

tolerance=0):

"""

Prepare early-stopping hook and variables.

Args:

metric:

String specifying the metric to early-stop on. Required with positive

``early\_stop\_patience``. For example, 'accuracy', 'accuracy\_0', 'loss', etc.

The string is used to extract the relevant tensor Op from the dict returned by

the get\_eval\_metric\_ops method. For ``metrics`` pass to the constructor,

the string is one of those. For multi-class (that is, multi-metric)

metrics, the string may be appended with a ``\_0``, ``\_1``, etc. or one

of the ``multi\_metric\_names`` (one per class).

patience:

Maximum number of epochs to wait for an improvement in the early\_stop\_metric

before breaking off training. For example, a patience of 10 means that

training will have 10 epochs to improve the metric before it is killed.

Whenever the metric is improved before running out of patience,

patience is reset to ``early\_stop\_patience``.

minimize:

Set this to True for metrics that need to be minimized

(like ``loss``). Metrics like ``accuracy`` that need to be maximized

should set this to False.

tolerance:

A non-negative tolerance for comparing early\_stop\_metric.

e.g. when maximizing the condition is current\_metric > best\_metric + tolerance."

Defaults to 0.

get\_estimator\_spec\_fn:

function that returns the current EstimatorSpec.

The EstimatorSpec is used to obtain the current eval\_metric\_ops.

checkpoint\_dir:

path to directory containing the Estimator checkpoints.

file\_path:

path to file that is used by this hook to communicate early-stopping

to StopIfExistsHook. This hook would be used for evaluation, while

the StopIfExistsHooks (the listeners) would be used for training.

When the file is created, the StopIfExistsHooks detect and terminate training.

This argument is used by ``Trainer.train\_and\_evaluate``.

exit\_on\_end:

when the end() method is called to indicate that the session is terminating,

and exit\_on\_end is True, twml.errors.EarlyStopError() is triggered to stop the evaluation job.

This is set to False by the trainer for non distributed jobs.

start\_epoch:

Specifies the starting epoch number. This is used for logging purposes only.

"""

if not isinstance(metric, str):

raise ValueError("Expecting string for metric arg")

if not isinstance(patience, int):

raise ValueError("Expecting positive number for metric arg")

self.should\_stop = False

self.\_metric = metric

self.\_patience = patience

self.\_current\_patience = patience

self.\_checkpoint\_dir = checkpoint\_dir

self.\_exit\_on\_end = exit\_on\_end

self.\_latest\_checkpoint\_path = None

# used for distributed training (tf.estimator.train\_and\_evaluate)

self.\_file\_path = file\_path

self.\_epoch = start\_epoch

if self.\_file\_path is not None:

# TODO try to read epoch from a file that we create

if tf.io.gfile.exists(self.\_file\_path):

# delete the file if it exists (not sure this makes sense)

logging.info("EarlyStopHook: Removing existing file: %s.", self.\_file\_path)

tf.io.gfile.remove(self.\_file\_path)

# best\_checkpoint dir will contain the best checkpoint

self.\_best\_checkpoint\_path = os.path.join(checkpoint\_dir, 'best\_checkpoint')

self.\_eval\_checkpoint\_path = os.path.join(checkpoint\_dir, 'eval\_checkpoint')

self.\_best\_metric\_path = os.path.join(self.\_best\_checkpoint\_path, self.\_metric)

if tf.io.gfile.exists(self.\_best\_metric\_path):

with tf.io.gfile.GFile(self.\_best\_metric\_path, mode="r") as f:

best\_metric\_from\_file = float(f.read())

else:

best\_metric\_from\_file = None

if minimize:

# current < best : is better

self.\_is\_better\_than = operator.lt

# worse metric possible

if best\_metric\_from\_file is None:

self.\_best\_metric = np.inf

else:

self.\_best\_metric = best\_metric\_from\_file - tolerance

# used for printing

self.\_early\_stop\_name = "minimum"

else:

# current > best : is better

self.\_is\_better\_than = operator.gt

# worse metric possible

if best\_metric\_from\_file is None:

self.\_best\_metric = -np.inf

else:

self.\_best\_metric = best\_metric\_from\_file + tolerance

# used for printing

self.\_early\_stop\_name = "maximum"

def get\_metrics\_fn():

""" function to get metric tensors to early-stopping """

estimator\_spec = get\_estimator\_spec\_fn()

eval\_metric\_ops = estimator\_spec.eval\_metric\_ops

if metric not in eval\_metric\_ops:

raise ValueError(

"Expecting early\_stop\_metric '%s' key in eval\_metric\_ops dict"

% (metric))

# get the value\_op from the (value\_op, update\_op) value

return {k: v[0] for k, v in eval\_metric\_ops.items()}

# initialize GetMetricsHook to get current value of metric from session

super(EarlyStopHook, self).\_\_init\_\_(get\_metrics\_fn=get\_metrics\_fn)

def early\_stop(self, epoch):

"""

Looks at the current value of the early stopping metric.

Decrements current patience. If metric improves, patience is reset

and latest checkpoint is moved to checkpoint\_dir/best\_checkpoint.

If current patience reaches zero, returns True.

Args:

epoch:

The current epoch number.

Returns:

True when early-stopped. False otherwise.

"""

# decrement patience

self.\_current\_patience -= 1

# get the current metric value

current\_metric = self.metric\_values[self.\_metric]

if self.\_is\_better\_than(current\_metric, self.\_best\_metric):

# save best version of model

self.\_best\_metric = current\_metric

logging.info(

"Found new %s %s=%f @ epoch %d",

self.\_early\_stop\_name, self.\_metric, self.\_best\_metric, epoch)

# backup the file to checkpoint\_dir/best\_checkpoint

assert self.\_latest\_checkpoint\_path, "expecting latest checkpoint"

logging.info("Backing up " + self.\_latest\_checkpoint\_path)

try:

eval\_checkpoint = tf.train.latest\_checkpoint(self.\_eval\_checkpoint\_path)

twml.util.backup\_checkpoint(

checkpoint\_path\_prefix=eval\_checkpoint,

backup\_path=self.\_best\_checkpoint\_path)

except twml.errors.CheckpointNotFoundError as ex:

msg = "Consider increasing 'keep\_checkpoint\_max' or 'save\_checkpoint\_secs'"

raise twml.errors.CheckpointNotFoundError(str(ex) + "\n" + msg)

tf.io.gfile.makedirs(os.path.dirname(self.\_best\_metric\_path))

with tf.io.gfile.GFile(self.\_best\_metric\_path, mode="w") as f:

# Write with enough precision

f.write("%.8f" % self.\_best\_metric)

# reset patience

self.\_current\_patience = self.\_patience

elif self.\_current\_patience > 0:

logging.info("No new %s found after %d epochs",

self.\_early\_stop\_name, self.\_patience - self.\_current\_patience)

elif self.\_current\_patience == 0:

logging.info(

"No new %s found after %d epochs. Early-stopping experiment.",

self.\_early\_stop\_name, self.\_patience)

return True

return False

def cleanup\_checkpoints(self):

"""

makes it so that the best checkpoint is the only checkpoint

in checkpoint\_dir.

"""

raise NotImplementedError("cleanup\_checkpoints is no longer supported")

def end(self, session):

"""

This method is called at the end of an evaluation/epoch.

When file\_path constructor argument is provided, this

will call ``early\_stop()``.

When ``early\_stop()`` returns True, it creates the file\_path,

which will be detected by StopIfExistsHooks

and stop training for all workers and the chief. It will

also call ``cleanup\_checkpoints()``.

"""

super(EarlyStopHook, self).end(session)

# Checks for early stopping criteria and makes a backup

self.should\_stop = self.early\_stop(self.\_epoch)

if self.\_file\_path is not None:

if self.should\_stop:

# create a file to inform workers

with tf.io.gfile.GFile(self.\_file\_path, "wb") as gfile:

gfile.write("early-stop\n")

# makes the best checkpoint the only checkpoint in save\_dir.

msg = "early-stopping evaluation at epoch %d" % self.\_epoch

logging.info(msg)

if self.\_exit\_on\_end:

raise twml.errors.EarlyStopError(msg)

else:

self.\_latest\_checkpoint\_path = None

self.\_epoch += 1

def begin(self):

"""

Saves the latest\_checkpoint in case it gets superseded by another checkpoint.

Remember that when used with train\_and\_evaluate, the chief saves checkpoints

continuouly. The chief could save a checkpoint after evaluation started.

So saving the checkpoint at the beginning of evaluation ensures that we

later save the correct best checkpoint.

"""

super(EarlyStopHook, self).begin()

self.\_latest\_checkpoint\_path = tf.train.latest\_checkpoint(self.\_checkpoint\_dir)

assert self.\_latest\_checkpoint\_path, "expecting latest checkpoint"

# Backup to temporary directory

try:

twml.util.backup\_checkpoint(

checkpoint\_path\_prefix=self.\_latest\_checkpoint\_path,

backup\_path=self.\_eval\_checkpoint\_path)

except twml.errors.CheckpointNotFoundError as ex:

msg = "Consider increasing 'keep\_checkpoint\_max' or 'save\_checkpoint\_secs'"

raise twml.errors.CheckpointNotFoundError(str(ex) + "\n" + msg)

class MetricsUpdateHook(GetMetricsHook):

"""

A GetMetricsHook augmented with logic to map SessionRun events to metrics updates.

It is mainly used by `TrackRun` to persist model metrics via Model Repo.

"""

def \_\_init\_\_(self,

get\_estimator\_spec\_fn,

add\_metrics\_fn,

every\_n\_iter=None,

every\_n\_secs=None

):

"""

Args:

get\_estimator\_spec\_fn:

function that returns the current EstimatorSpec.

The EstimatorSpec is used to obtain the current eval\_metric\_ops.

add\_metrics\_fn: `function` callback used to report metrics, called automatically

at the end of every epoch.

every\_n\_iter: `int`, log the metrics once every N local

steps taken in the current epoch.

every\_n\_secs: `int` or `float`, log the metrics once every N

seconds passed in the current epoch. Exactly one of `every\_n\_iter` and `every\_n\_secs`

should be provided.

Raises:

ValueError: if `every\_n\_iter` is non-positive or if not exactly one of `every\_n\_iter` and

`every\_n\_secs` is set when `add\_progress\_metrics\_fn` is provided.

"""

only\_log\_at\_end = (every\_n\_iter is None) and (every\_n\_secs is None)

if (not only\_log\_at\_end and every\_n\_iter and every\_n\_secs):

raise ValueError(

'exactly one of every\_n\_iter and every\_n\_secs must be provided'

)

# TODO: should have a minimum to avoid too many calls to ModelRepo?

if every\_n\_iter is not None and every\_n\_iter <= 0:

raise ValueError("invalid every\_n\_iter=%s." % every\_n\_iter)

self.\_timer = (

NeverTriggerTimer() if only\_log\_at\_end else

SecondOrStepTimer(every\_secs=every\_n\_secs, every\_steps=every\_n\_iter)

)

self.\_should\_trigger = False

self.\_iter\_count = 0

self.\_add\_metrics\_fn = add\_metrics\_fn

def get\_metrics\_fn():

"""

Function that returns the current EstimatorSpec.

The EstimatorSpec is used to obtain the current eval\_metric\_ops.

"""

estimator\_spec = get\_estimator\_spec\_fn()

eval\_metric\_ops = estimator\_spec.eval\_metric\_ops

# get the value\_op from the (value\_op, update\_op) value

return {k: v[0] for k, v in eval\_metric\_ops.items()}

super(MetricsUpdateHook, self).\_\_init\_\_(get\_metrics\_fn=get\_metrics\_fn)

def report\_metrics(self):

"""

Triggers a metrics report.

"""

self.\_timer.update\_last\_triggered\_step(self.\_iter\_count)

if self.metric\_values is not None:

self.\_add\_metrics\_fn(self.metric\_values)

def begin(self):

"""

Triggered before each epoch.

"""

self.\_timer.reset()

self.\_iter\_count = 0

return super(MetricsUpdateHook, self).begin()

def before\_run(self, run\_context):

"""

Triggered before each step.

"""

self.\_should\_trigger = self.\_timer.should\_trigger\_for\_step(self.\_iter\_count)

return super(MetricsUpdateHook, self).before\_run(run\_context)

def after\_run(self, run\_context, run\_values):

"""

Triggered after each step.

"""

if self.\_should\_trigger:

self.report\_metrics()

self.\_iter\_count += 1

return super(MetricsUpdateHook, self).after\_run(run\_context, run\_values)

def end(self, session):

"""

Triggered after each epoch.

"""

self.report\_metrics()

return super(MetricsUpdateHook, self).end(session)

class EarlyStopDuration(tf.train.SessionRunHook):

"""

Hook that can be used to terminate a job (training or validation) after a certain duration.

The hook is fault tolerant, i.e., if a job is allotted 1 hour to run and fails after 45 minutes,

then it will only run for 15 minutes once restarted.

Args:

max\_duration:

A float. When this argument is defined, the job will automatically terminate after

`max\_duration` seconds if it has not already compeleted.

overwrite:

A boolean. If set to True, this hook will overwrite the file containing the elapsed time

since the beginning of the job. In a distributed setting, this will be used so only one

job writes to the file while all others will have read access. In a distributed setting,

if all executors have this parameter set to False, then it just means that the hook will

not be fault tolerant. When restarted, the job will restart the clock from 0.

save\_dir:

String. A directory (located on a file system that is Tensorflow compatible) where

we can store the file which contains the record of the elapsed time. This file is what makes

the hook faul tolerant.

exit\_on\_end:

when exit\_on\_end is True, twml.errors.EarlyStopError() is triggered to stop the job.

This is usually set to True to kill a validation job in a distributed setting.

"""

def \_\_init\_\_(self, max\_duration: float, exit\_on\_end: bool, save\_dir: str, overwrite: bool):

self.\_overwrite = overwrite

self.\_save\_dir = save\_dir

self.\_exit\_on\_end = exit\_on\_end

self.\_max\_duration = max\_duration

self.\_last\_time\_check = datetime.now()

# Initialize elapse time file

if overwrite:

self.elapsed\_time()

@property

def elapsed\_file\_path(self):

return os.path.join(self.\_save\_dir, "early\_stop\_duration.txt")

def early\_stop(self) -> bool:

return self.elapsed\_time() > self.\_max\_duration

def elapsed\_time(self) -> float:

# Recorded elapsed time is 0 unless it's been recorded in a file already

recorded\_elapsed\_time = 0

if tf.io.gfile.exists(self.elapsed\_file\_path):

with tf.io.gfile.GFile(self.elapsed\_file\_path, mode="r") as file:

recorded\_elapsed\_time = json.loads(file.read())["elapsed\_time"]

elapsed\_time = recorded\_elapsed\_time + (datetime.now() - self.\_last\_time\_check).total\_seconds()

self.\_last\_time\_check = datetime.now()

if self.\_overwrite:

# Record the actualized new elapsed time to the file

tf.io.gfile.makedirs(os.path.dirname(self.elapsed\_file\_path))

with tf.io.gfile.GFile(self.elapsed\_file\_path, mode="w") as file:

record = {

"elapsed\_time": elapsed\_time,

"max\_duration": self.\_max\_duration

}

file.write(json.dumps(record, indent=2))

return elapsed\_time

def before\_run(self, run\_context: tf.estimator.SessionRunContext) -> None:

if self.early\_stop():

message = f"""

Stopping job which now exceeded the maximum duration of {self.\_max\_duration} seconds.

"""

logging.info(message)

run\_context.request\_stop()

if self.\_exit\_on\_end:

raise twml.errors.EarlyStopError(message)

class StopAtStepHook(tf.train.StopAtStepHook):

"""

Overrides ``tf.train.StopAtStepHook`` so that

a ``stop\_requested`` property can be accessed to determine

if this hook requested a stop.

"""

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(StopAtStepHook, self).\_\_init\_\_(\*args, \*\*kwargs)

self.\_stop\_requested = False

@property

def stop\_requested(self):

""" true if this hook requested a stop """

return self.\_stop\_requested

def after\_run(self, run\_context, run\_values):

""" sets self.stop\_requested to true when requesting a stop """

super(StopAtStepHook, self).after\_run(run\_context, run\_values)

self.\_stop\_requested = run\_context.stop\_requested

class StopIfExistsHook(tf.train.SessionRunHook):

"""

Hook that requests stop if a file exists.

This hook is used with the EarlyStopHook to implement

early-stopping for distributed training (tf.estimator.train\_and\_evaluate).

"""

def \_\_init\_\_(self, file\_path):

"""

Arguments:

file\_path:

path to file. When this hook detects that the file exists,

it requests a stop, which effectively kills this worker.

"""

self.\_file\_path = file\_path

self.\_stop\_requested = False

def after\_run(self, run\_context, run\_values):

if tf.io.gfile.exists(self.\_file\_path):

logging.info("Early-stopping file detected; requesting stop")

run\_context.request\_stop()

self.\_stop\_requested = True

@property

def stop\_requested(self):

""" true if this hook requested a stop """

return self.\_stop\_requested