tf.train.Supervisor

Contents

Class `Supervisor`

Defined in `tensorflow/python/training/supervisor.py`.

See the guide: Training > Distributed execution

A training helper that checkpoints models and computes summaries.

The Supervisor is a small wrapper around a `Coordinator`, a `Saver`, and a `SessionManager` that takes care of common needs of TensorFlow training programs.

Use for a single program

```
with tf.Graph().as_default():
  ...add operations to the graph...
  # Create a Supervisor that will checkpoint the model in '/tmp/mydir'.
  sv = Supervisor(logdir='/tmp/mydir')
  # Get a TensorFlow session managed by the supervisor.
  with sv.managed_session(FLAGS.master) as sess:
    # Use the session to train the graph.
    while not sv.should_stop():
      sess.run(<my_train_op>)
```

Within the `with sv.managed_session()` block all variables in the graph have been initialized. In addition, a few services have been started to checkpoint the model and add summaries to the event log.

If the program crashes and is restarted, the managed session automatically reinitialize variables from the most recent checkpoint.

The supervisor is notified of any exception raised by one of the services. After an exception is raised, `should_stop()` returns `True`. In that case the training loop should also stop. This is why the training loop has to check for `sv.should_stop()`.

Exceptions that indicate that the training inputs have been exhausted, `tf.errors.OutOfRangeError`, also cause `sv.should_stop()` to return `True` but are not re-raised from the `with` block: they indicate a normal termination.

Use for multiple replicas

To train with replicas you deploy the same program in a `Cluster`. One of the tasks must be identified as the *chief*: the task that handles initialization, checkpoints, summaries, and recovery. The other tasks depend on the *chief* for these services.

The only change you have to do to the single program code is to indicate if the program is running as the *chief*.

```
# Choose a task as the chief. This could be based on server_def.task_index,
# or job_def.name, or job_def.tasks. It's entirely up to the end user.
# But there can be only one *chief*.
is_chief = (server_def.task_index == 0)
server = tf.train.Server(server_def)

with tf.Graph().as_default():
  ...add operations to the graph...
  # Create a Supervisor that uses log directory on a shared file system.
  # Indicate if you are the 'chief'
  sv = Supervisor(logdir='/shared_directory/...', is_chief=is_chief)
  # Get a Session in a TensorFlow server on the cluster.
  with sv.managed_session(server.target) as sess:
    # Use the session to train the graph.
    while not sv.should_stop():
      sess.run(<my_train_op>)
```

In the *chief* task, the `Supervisor` works exactly as in the first example above. In the other tasks `sv.managed_session()` waits for the Model to have been initialized before returning a session to the training code. The non-chief tasks depend on the chief task for initializing the model.

If one of the tasks crashes and restarts, `managed_session()` checks if the Model is initialized. If yes, it just creates a session and returns it to the training code that proceeds normally. If the model needs to be initialized, the chief task takes care of reinitializing it; the other tasks just wait for the model to have been initialized.

NOTE: This modified program still works fine as a single program. The single program marks itself as the chief.

What `master` string to use

Whether you are running on your machine or in the cluster you can use the following values for the –master flag:

- Specifying `''` requests an in-process session that does not use RPC.
- Specifying `'local'` requests a session that uses the RPC-based "Master interface" to run TensorFlow programs. See `tf.train.Server.create_local_server` for details.
- Specifying `'grpc://hostname:port'` requests a session that uses the RPC interface to a specific host, and also allows the in-process master to access remote tensorflow workers. Often, it is appropriate to pass `server.target` (for some `tf.train.Server` named `server).

Advanced use

Launching additional services

`managed_session()` launches the Checkpoint and Summary services (threads). If you need more services to run you can simply launch them in the block controlled by `managed_session()`.

Example: Start a thread to print losses. We want this thread to run every 60 seconds, so we launch it with `sv.loop()`.

```
...
sv = Supervisor(logdir='/tmp/mydir')
with sv.managed_session(FLAGS.master) as sess:
  sv.loop(60, print_loss, (sess, ))
  while not sv.should_stop():
    sess.run(my_train_op)
```

Launching fewer services

`managed_session()` launches the "summary" and "checkpoint" threads which use either the optionally `summary_op` and `saver` passed to the constructor, or default ones created automatically by the supervisor. If you want to run your own summary and checkpointing logic, disable these services by passing `None` to the `summary_op` and `saver` parameters.

Example: Create summaries manually every 100 steps in the chief.

```
# Create a Supervisor with no automatic summaries.
sv = Supervisor(logdir='/tmp/mydir', is_chief=is_chief, summary_op=None)
# As summary_op was None, managed_session() does not start the
# summary thread.
with sv.managed_session(FLAGS.master) as sess:
  for step in xrange(1000000):
    if sv.should_stop():
      break
    if is_chief and step % 100 == 0:
      # Create the summary every 100 chief steps.
      sv.summary_computed(sess, sess.run(my_summary_op))
    else:
      # Train normally
      sess.run(my_train_op)
```

Custom model initialization

`managed_session()` only supports initializing the model by running an `init_op` or restoring from the latest checkpoint. If you have special initialization needs, see how to specify a `local_init_op` when creating the supervisor. You can also use the `SessionManager` directly to create a session and check if it could be initialized automatically.

Properties

`coord`

Return the Coordinator used by the Supervisor.

The Coordinator can be useful if you want to run multiple threads during your training.

Returns:

A Coordinator object.

`global_step`

Return the global_step Tensor used by the supervisor.

Returns:

An integer Tensor for the global_step.

`init_feed_dict`

Return the feed dictionary used when evaluating the `init_op`.

Returns:

A feed dictionary or `None`.

`init_op`

Return the Init Op used by the supervisor.

Returns:

An Op or `None`.

`is_chief`

Return True if this is a chief supervisor.

Returns:

A bool.

`ready_for_local_init_op`

`ready_op`

Return the Ready Op used by the supervisor.

Returns:

An Op or `None`.

`save_model_secs`

Return the delay between checkpoints.

Returns:

A timestamp.

`save_path`

Return the save path used by the supervisor.

Returns:

A string.

`save_summaries_secs`

Return the delay between summary computations.

Returns:

A timestamp.

`saver`

Return the Saver used by the supervisor.

Returns:

A Saver object.

`session_manager`

Return the SessionManager used by the Supervisor.

Returns:

A SessionManager object.

`summary_op`

Return the Summary Tensor used by the chief supervisor.

Returns:

A string Tensor for the summary or `None`.

`summary_writer`

Return the SummaryWriter used by the chief supervisor.

Returns:

A SummaryWriter.

Methods

`__init__`

```
__init__(
    graph=None,
    ready_op=USE_DEFAULT,
    ready_for_local_init_op=USE_DEFAULT,
    is_chief=True,
    init_op=USE_DEFAULT,
    init_feed_dict=None,
    local_init_op=USE_DEFAULT,
    logdir=None,
    summary_op=USE_DEFAULT,
    saver=USE_DEFAULT,
    global_step=USE_DEFAULT,
    save_summaries_secs=120,
    save_model_secs=600,
    recovery_wait_secs=30,
    stop_grace_secs=120,
    checkpoint_basename='model.ckpt',
    session_manager=None,
    summary_writer=USE_DEFAULT,
    init_fn=None
)
```

Create a `Supervisor`.

Args:

- `graph`: A `Graph`. The graph that the model will use. Defaults to the default `Graph`. The supervisor may add operations to the graph before creating a session, but the graph should not be modified by the caller after passing it to the supervisor.
- `ready_op`: 1-D string `Tensor`. This tensor is evaluated by supervisors in `prepare_or_wait_for_session()` to check if the model is ready to use. The model is considered ready if it returns an empty array. Defaults to the tensor returned from `tf.report_uninitialized_variables()` If `None`, the model is not checked for readiness.
- `ready_for_local_init_op`: 1-D string `Tensor`. This tensor is evaluated by supervisors in `prepare_or_wait_for_session()` to check if the model is ready to run the local_init_op. The model is considered ready if it returns an empty array. Defaults to the tensor returned from `tf.report_uninitialized_variables(tf.global_variables())`. If `None`, the model is not checked for readiness before running local_init_op.
- `is_chief`: If True, create a chief supervisor in charge of initializing and restoring the model. If False, create a supervisor that relies on a chief supervisor for inits and restore.
- `init_op`: `Operation`. Used by chief supervisors to initialize the model when it can not be recovered. Defaults to an `Operation` that initializes all global variables. If `None`, no initialization is done automatically unless you pass a value for `init_fn`, see below.
- `init_feed_dict`: A dictionary that maps `Tensor` objects to feed values. This feed dictionary will be used when `init_op` is evaluated.
- `local_init_op`: `Operation`. Used by all supervisors to run initializations that should run for every new supervisor instance. By default these are table initializers and initializers for local variables. If `None`, no further per supervisor-instance initialization is done automatically.
- `logdir`: A string. Optional path to a directory where to checkpoint the model and log events for the visualizer. Used by chief supervisors. The directory will be created if it does not exist.
- `summary_op`: An `Operation` that returns a Summary for the event logs. Used by chief supervisors if a `logdir` was specified. Defaults to the operation returned from summary.merge_all(). If `None`, summaries are not computed automatically.
- `saver`: A Saver object. Used by chief supervisors if a `logdir` was specified. Defaults to the saved returned by Saver(). If `None`, the model is not saved automatically.

- `global_step`: An integer Tensor of size 1 that counts steps. The value from 'global_step' is used in summaries and checkpoint filenames. Default to the op named 'global_step' in the graph if it exists, is of rank 1, size 1, and of type tf.int32 or tf.int64. If `None` the global step is not recorded in summaries and checkpoint files. Used by chief supervisors if a `logdir` was specified.
- `save_summaries_secs`: Number of seconds between the computation of summaries for the event log. Defaults to 120 seconds. Pass 0 to disable summaries.
- `save_model_secs`: Number of seconds between the creation of model checkpoints. Defaults to 600 seconds. Pass 0 to disable checkpoints.
- `recovery_wait_secs`: Number of seconds between checks that the model is ready. Used by supervisors when waiting for a chief supervisor to initialize or restore the model. Defaults to 30 seconds.
- `stop_grace_secs`: Grace period, in seconds, given to running threads to stop when `stop()` is called. Defaults to 120 seconds.
- `checkpoint_basename`: The basename for checkpoint saving.
- `session_manager`: `SessionManager`, which manages Session creation and recovery. If it is `None`, a default `SessionManager` will be created with the set of arguments passed in for backwards compatibility.
- `summary_writer`: `SummaryWriter` to use or `USE_DEFAULT`. Can be `None` to indicate that no summaries should be written.
- `init_fn`: Optional callable used to initialize the model. Called after the optional `init_op` is called. The callable must accept one argument, the session being initialized.

Returns:

A `Supervisor`.

Loop

```
Loop(
    timer_interval_secs,
    target,
    args=None,
    kwargs=None
)
```

Start a LooperThread that calls a function periodically.

If `timer_interval_secs` is None the thread calls `target(*args, **kwargs)` repeatedly. Otherwise it calls it every `timer_interval_secs` seconds. The thread terminates when a stop is requested.

The started thread is added to the list of threads managed by the supervisor so it does not need to be passed to the `stop()` method.

Args:

- `timer_interval_secs`: Number. Time boundaries at which to call `target`.
- `target`: A callable object.
- `args`: Optional arguments to pass to `target` when calling it.
- `kwargs`: Optional keyword arguments to pass to `target` when calling it.

Returns:

The started thread.

PrepareSession

```
PrepareSession(
    master='',
    config=None,
    wait_for_checkpoint=False,
    max_wait_secs=7200,
    start_standard_services=True
)
```

Make sure the model is ready to be used.

Create a session on 'master', recovering or initializing the model as needed, or wait for a session to be ready. If running as the chief and `start_standard_service` is set to True, also call the session manager to start the standard services.

Args:

- `master`: name of the TensorFlow master to use. See the `tf.Session` constructor for how this is interpreted.
- `config`: Optional ConfigProto proto used to configure the session, which is passed as-is to create the session.
- `wait_for_checkpoint`: Whether we should wait for the availability of a checkpoint before creating Session. Defaults to False.
- `max_wait_secs`: Maximum time to wait for the session to become available.
- `start_standard_services`: Whether to start the standard services and the queue runners.

Returns:

A Session object that can be used to drive the model.

RequestStop

```
RequestStop(ex=None)
```

Request that the coordinator stop the threads.

See `Coordinator.request_stop()`.

Args:

- `ex`: Optional `Exception`, or Python `exc_info` tuple as returned by `sys.exc_info()`. If this is the first call to `request_stop()` the corresponding exception is recorded and re-raised from `join()`.

ShouldStop

```
ShouldStop()
```

Check if the coordinator was told to stop.

See `Coordinator.should_stop()`.

Returns:

True if the coordinator was told to stop, False otherwise.

StartQueueRunners

```
StartQueueRunners(
    sess,
    queue_runners=None
)
```

Start threads for `QueueRunners`.

Note that the queue runners collected in the graph key `QUEUE_RUNNERS` are already started automatically when you create a session with the supervisor, so unless you have non-collected queue runners to start you do not need to call this explicitly.

Args:

- `sess`: A `Session`.
- `queue_runners`: A list of `QueueRunners`. If not specified, we'll use the list of queue runners gathered in the graph under the key `GraphKeys.QUEUE_RUNNERS`.

Returns:

The list of threads started for the `QueueRunners`.

StartStandardServices

```
StartStandardServices(sess)
```

Start the standard services for 'sess'.

This starts services in the background. The services started depend on the parameters to the constructor and may include:

- A Summary thread computing summaries every save_summaries_secs.
- A Checkpoint thread saving the model every save_model_secs.
- A StepCounter thread measure step time.

Args:

- `sess`: A Session.

Returns:

A list of threads that are running the standard services. You can use the Supervisor's Coordinator to join these threads with: sv.coord.Join()

Raises:

- `RuntimeError`: If called with a non-chief Supervisor.
- `ValueError`: If not `logdir` was passed to the constructor as the services need a log directory.

Stop

```
Stop(
    threads=None,
    close_summary_writer=True
)
```

Stop the services and the coordinator.

This does not close the session.

Args:

- `threads`: Optional list of threads to join with the coordinator. If `None`, defaults to the threads running the standard services, the threads started for `QueueRunners`, and the threads started by the `loop()` method. To wait on additional threads, pass the list in this parameter.
- `close_summary_writer`: Whether to close the `summary_writer`. Defaults to `True` if the summary writer was created by the supervisor, `False` otherwise.

StopOnException

```
StopOnException()
```

Context handler to stop the supervisor when an exception is raised.

See `Coordinator.stop_on_exception()`.

Returns:

A context handler.

SummaryComputed

```
SummaryComputed(
    sess,
    summary,
    global_step=None
)
```

Indicate that a summary was computed.

Args:

- `sess`: A `Session` object.
- `summary`: A Summary proto, or a string holding a serialized summary proto.
- `global_step`: Int. global step this summary is associated with. If `None`, it will try to fetch the current step.

Raises:

- `TypeError`: if 'summary' is not a Summary proto or a string.
- `RuntimeError`: if the Supervisor was created without a `logdir`.

WaitForStop

```
WaitForStop()
```

Block waiting for the coordinator to stop.

`loop`

```
loop(
    timer_interval_secs,
    target,
    args=None,
    kwargs=None
)
```

Start a LooperThread that calls a function periodically.

If `timer_interval_secs` is None the thread calls `target(*args, **kwargs)` repeatedly. Otherwise it calls it every `timer_interval_secs` seconds. The thread terminates when a stop is requested.

The started thread is added to the list of threads managed by the supervisor so it does not need to be passed to the `stop()` method.

Args:

- `timer_interval_secs`: Number. Time boundaries at which to call `target`.
- `target`: A callable object.
- `args`: Optional arguments to pass to `target` when calling it.
- `kwargs`: Optional keyword arguments to pass to `target` when calling it.

Returns:

The started thread.

`managed_session`

```
managed_session(
    *args,
    **kwds
)
```

Returns a context manager for a managed session.

This context manager creates and automatically recovers a session. It optionally starts the standard services that handle checkpoints and summaries. It monitors exceptions raised from the `with` block or from the services and stops the supervisor as needed.

The context manager is typically used as follows:

```
def train():
  sv = tf.train.Supervisor(...)
  with sv.managed_session(<master>) as sess:
    for step in xrange(..):
      if sv.should_stop():
        break
      sess.run(<my training op>)
      ...do other things needed at each training step...
```

An exception raised from the `with` block or one of the service threads is raised again when the block exits. This is done after stopping all threads and closing the session. For example, an `AbortedError` exception, raised in case of preemption of one of the workers in a distributed model, is raised again when the block exits.

If you want to retry the training loop in case of preemption you can do it as follows:

```
def main(...):
  while True
    try:
      train()
    except tf.errors.Aborted:
      pass
```

As a special case, exceptions used for control flow, such as `OutOfRangeError` which reports that input queues are exhausted, are not raised again from the `with` block: they indicate a clean termination of the training loop and are considered normal termination.

Args:

- `master`: name of the TensorFlow master to use. See the `tf.Session` constructor for how this is interpreted.
- `config`: Optional `ConfigProto` proto used to configure the session. Passed as-is to create the session.
- `start_standard_services`: Whether to start the standard services, such as checkpoint, summary and step counter.
- `close_summary_writer`: Whether to close the summary writer when closing the session. Defaults to True.

Returns:

A context manager that yields a `Session` restored from the latest checkpoint or initialized from scratch if not checkpoint exists. The session is closed when the `with` block exits.

prepare_or_wait_for_session

```
prepare_or_wait_for_session(
    master='',
    config=None,
    wait_for_checkpoint=False,
    max_wait_secs=7200,
    start_standard_services=True
)
```

Make sure the model is ready to be used.

Create a session on 'master', recovering or initializing the model as needed, or wait for a session to be ready. If running as the chief and `start_standard_service` is set to True, also call the session manager to start the standard services.

Args:

- `master`: name of the TensorFlow master to use. See the `tf.Session` constructor for how this is interpreted.
- `config`: Optional ConfigProto proto used to configure the session, which is passed as-is to create the session.
- `wait_for_checkpoint`: Whether we should wait for the availability of a checkpoint before creating Session. Defaults to False.
- `max_wait_secs`: Maximum time to wait for the session to become available.
- `start_standard_services`: Whether to start the standard services and the queue runners.

Returns:

A Session object that can be used to drive the model.

request_stop

```
request_stop(ex=None)
```

Request that the coordinator stop the threads.

See `Coordinator.request_stop()`.

Args:

- `ex`: Optional `Exception`, or Python `exc_info` tuple as returned by `sys.exc_info()`. If this is the first call to `request_stop()` the corresponding exception is recorded and re-raised from `join()`.

should_stop

```
should_stop()
```

Check if the coordinator was told to stop.

See `Coordinator.should_stop()`.

Returns:

True if the coordinator was told to stop, False otherwise.

start_queue_runners

```
start_queue_runners(
    sess,
    queue_runners=None
)
```

Start threads for `QueueRunners`.

Note that the queue runners collected in the graph key `QUEUE_RUNNERS` are already started automatically when you create a session with the supervisor, so unless you have non-collected queue runners to start you do not need to call this explicitly.

Args:

- `sess`: A `Session`.
- `queue_runners`: A list of `QueueRunners`. If not specified, we'll use the list of queue runners gathered in the graph under the key `GraphKeys.QUEUE_RUNNERS`.

Returns:

The list of threads started for the `QueueRunners`.

start_standard_services

```
start_standard_services(sess)
```

Start the standard services for 'sess'.

This starts services in the background. The services started depend on the parameters to the constructor and may include:

- A Summary thread computing summaries every save_summaries_secs.
- A Checkpoint thread saving the model every save_model_secs.
- A StepCounter thread measure step time.

Args:

- `sess`: A Session.

Returns:

A list of threads that are running the standard services. You can use the Supervisor's Coordinator to join these threads with: sv.coord.Join()

Raises:

- `RuntimeError`: If called with a non-chief Supervisor.
- `ValueError`: If not `logdir` was passed to the constructor as the services need a log directory.

stop

```
stop(
    threads=None,
    close_summary_writer=True
)
```

Stop the services and the coordinator.

This does not close the session.

Args:

- `threads`: Optional list of threads to join with the coordinator. If `None`, defaults to the threads running the standard services, the threads started for `QueueRunners`, and the threads started by the `loop()` method. To wait on additional threads, pass the list in this parameter.
- `close_summary_writer`: Whether to close the `summary_writer`. Defaults to `True` if the summary writer was created by the supervisor, `False` otherwise.

stop_on_exception

```
stop_on_exception()
```

Context handler to stop the supervisor when an exception is raised.

See `Coordinator.stop_on_exception()`.

Returns:

A context handler.

summary_computed

```
summary_computed(
    sess,
    summary,
    global_step=None
)
```

Indicate that a summary was computed.

Args:

- `sess`: A `Session` object.
- `summary`: A Summary proto, or a string holding a serialized summary proto.
- `global_step`: Int. global step this summary is associated with. If `None`, it will try to fetch the current step.

Raises:

- `TypeError`: if 'summary' is not a Summary proto or a string.
- `RuntimeError`: if the Supervisor was created without a `logdir`.

wait_for_stop

```
wait_for_stop()
```

Block waiting for the coordinator to stop.

Class Members

USE_DEFAULT

English