Supervisor: Training Helper for Days-Long Trainings.

To train a model with TensorFlow you can simply run a training op a number of times and save a checkpoint of the trained parameters when you're done. This works well for small models that can train in a few hours.

Larger models that require days of training, possibly across multiple replicas, need a more robust training process that:

- Handles shutdowns and crashes cleanly.
- Can be resumed after a shutdown or a crash.
- Can be monitored through TensorBoard.

To be able to resume training after a shutdown or a crash the training process must save checkpoints regularly. On restart, it must look for the most recent checkpoint and load it before resuming training.

To be monitored through TensorBoard, the training process must run summary ops regularly and append the returned values to an events file as explained in TensorBoard: Visualizing Learning (https://www.tensorflow.org/get_started/summaries_and_tensorboard). TensorBoard monitors events files and displays graphs reporting training progress over time.

The **`tf.train.Supervisor`** (https://www.tensorflow.org/api_docs/python/tf/train/Supervisor) provides a set of services that helps implement a robust training process.

This how-to shows how to use the supervisor directly. Please also consider using one of several frameworks built on top of the supervisor that provide richer training loops, and numerous customization options: **`tf.learn`** (https://www.tensorflow.org/api_guides/python/contrib.learn) is a good choice.

Note that the supervisor is very helpful for training large models, but can also be used for smaller models without any penalty.

Very Simple Scenario

The simplest scenario for using a supervisor is to:

- Create a **`Supervisor`** object, passing it the path to a directory where to save checkpoints and summaries.
- Ask the supervisor for a session with **`tf.train.Supervisor.managed_session`** (https://www.tensorflow.org/api_docs/python/tf/train/Supervisor#managed_session).
- Use the session to execute a train op, checking at each step if the supervisor requests that the training stops.

```
...create graph...
my_train_op = ...

sv = tf.train.Supervisor(logdir="/my/training/directory")
with sv.managed_session() as sess:
  for step in range(100000):
    if sv.should_stop():
      break
    sess.run(my_train_op)
```

Started Services

In the very simple scenario, the **`managed_session()`** call starts a few services, which run in their own threads, and use the managed session to run ops in your graph.

If your graph contains an integer variable named **`global_step`**, the services use its value to measure the number of training steps executed. See the MNIST training tutorial (https://www.tensorflow.org/get_started/mnist/mechanics#training) for how to create a **`global_step`** variable.

- *Checkpointing* service: Saves a copy of the graph variables in the logdir. The checkpoint filename uses the value of the **`global_step`** variable if one was added to your graph. Runs every 10 minutes by default.
- *Summary* service: Runs all the summary ops and appends their output to an events file (https://www.tensorflow.org/get_started/summaries_and_tensorboard) in the logdir. Runs every 2 minutes by default.

- *Step counter*: Counts how many steps have been executed, by looking at changes in the **`global_step`** variable. Appends a summary to the events file reporting the number of global steps per second. The summary tag is "global_step/sec". This also runs every 2 minutes by default.
- *Queue Runners*: If any **`tf.train.QueueRunner`** (https://www.tensorflow.org/api_docs/python/tf/train/QueueRunner) were added to the graph, the supervisor launches them in their own threads.

All time intervals can be changed when constructing the supervisor object. See the supervisor reference (#supervisor_reference) for details.

Checking for Stop

The check for stop in the main training loop is important and necessary.

Exceptions raised in the service threads are reported to the supervisor which then sets its **`should_stop()`** condition to true. Other service threads notice that condition and terminate properly. The main training loop, within the **`managed_session()`** block, must also check for the stop condition and terminate.

Note that **`managed_session()`** takes care of catching exceptions raised from the training loop to report them to the supervisor. The main loop does not need to do anything special about exceptions. It only needs to check for the stop condition.

Recovery

If the training program shuts down or crashes, its most recent checkpoint and event files are left in the logdir. When you restart the program, **`managed_session()`** restores the graph from the most recent checkpoint and resumes training where it stopped.

A new events file is created. If you start TensorBoard and point it to the logdir, it will know how to merge the contents of the two events files and will show the training resuming at the last global step from the checkpoint.

Larger Model Scenario

The very simple scenario is sufficient for most small to medium sized models. Larger models may run out memory when the summary service runs: The summary ops are run in parallel with the main loop running the train op. This can cause memory usage to peak to up to two times the normal use.

For a larger model you can tell the supervisor to not run the summary service and instead run it yourself in your main training loop: pass **`summary_op=None`** when constructing the supervisor.

For example this code runs the summary op every 100 steps in the training loop:

```
...create graph...
my_train_op = ...
my_summary_op = tf.summary.merge_all()
sv = tf.train.Supervisor(logdir="/my/training/directory",
                         summary_op=None) # Do not run the summary service
with sv.managed_session() as sess:
  for step in range(100000):
    if sv.should_stop():
      break
    if step % 100 == 0:
      _, summ = session.run([my_train_op, my_summary_op])
      sv.summary_computed(sess, summ)
    else:
      session.run(my_train_op)
```

Pre-trained Model Scenario

The **`managed_session()`** call takes care of initializing the model in the session. The model is restored from a checkpoint if one is available, or initialized from scratch otherwise.

One common scenario is to initialize the model by loading a "pre-trained" checkpoint that was saved while training a usually slightly different model using a different dataset.

You can load a pre-trained checkpoint by passing an "init function" to the supervisor. This function is called only if the model needs to be initialized from scratch, not when the model can be recovered from a checkpoint from the logdir.

To load the pre-trained model, the init function needs a **`tf.train.Saver`** (https://www.tensorflow.org/api_docs/python/tf/train/Saver) object, so you should create a saver for this purpose. This is usually a good idea because the new model may contain variables that are not present in the pre-trained checkpoint: This saver must only restore the pre-trained variables. If you were using the default saver, you could get an error trying to restore all the variables of the new model from the pre-trained checkpoint.

```
...create graph...
# Create a saver that restores only the pre-trained variables.
pre_train_saver = tf.train.Saver([pre_train_var1, pre_train_var2])

# Define an init function that loads the pretrained checkpoint.
def load_pretrain(sess):
  pre_train_saver.restore(sess, "<path to pre-trained-checkpoint>")

# Pass the init function to the supervisor.
#
# The init function is called _after_ the variables have been initialized
# by running the init_op.
sv = tf.train.Supervisor(logdir="/my/training/directory",
                         init_fn=load_pretrain)
with sv.managed_session() as sess:
  # Here sess was either initialized from the pre-trained-checkpoint or
  # recovered from a checkpoint saved in a previous run of this code.
  ...
```

Running Your Own Services

Supervisor services, such as the checkpointing service, run in threads parallel to the main training loop. You sometimes want to add your own services, for example to fetch different sets of summaries on a different schedule than the usual summary service.

Use the **`tf.train.Supervisor.loop`** (https://www.tensorflow.org/api_docs/python/tf/train/Supervisor#loop) method of the supervisor for this purpose. It repeatedly calls a function of your choice on a timer until the supervisor stop condition becomes true, so it plays nicely with the other services.

Example: Call **`my_additional_summaries()`** every 20mn:

```
def my_additional_sumaries(sv, sess):
 ...fetch and write summaries, see below...

...
  sv = tf.train.Supervisor(logdir="/my/training/directory")
  with sv.managed_session() as sess:
    # Call my_additional_sumaries() every 1200s, or 20mn,
    # passing (sv, sess) as arguments.
    sv.loop(1200, my_additional_sumaries, args=(sv, sess))
    ...main training loop...
```

Writing Summaries

The supervisor always creates an events file in its logdir, as well as a **`tf.summary.FileWriter`** (https://www.tensorflow.org/api_docs/python/tf/summary/FileWriter) to append events and summaries to that file. If you want to write your own summaries it is a good idea to append them to that same events file: TensorBoard likes it better when only one events file in a directory is being actively appended to.

The supervisor provides a helper function to append summaries: **`tf.train.Supervisor.summary_computed`** (https://www.tensorflow.org/api_docs/python/tf/train/Supervisor#summary_computed). Just pass to the function the output returned by a summary op. Here is an example of using that function to implement **`my_additional_sumaries()`** from the previous example:

```
def my_additional_sumaries(sv, sess):
  summaries = sess.run(my_additional_summary_op)
  sv.summary_computed(sess, summaries)
```

For more advanced usages, the supervisor provides access to its summary writer through its **`tf.train.Supervisor.summary_writer`** (https://www.tensorflow.org/api_docs/python/tf/train/Supervisor#summary_writer) attribute.

Supervisor Reference

The Very Simple Scenario (#very_simple_scenario), and the Larger Model Scenario (#larger_model_scenario) show basic uses of a supervisor. More advanced scenarios can be constructed by using the many options provided by the supervisor

Checkpointing: Where and When.

The **`managed_session()`** call launches the checkpointing service, which can be configured by the following keyword arguments to the **`Supervisor()`** constructor:

- **`logdir`**: path to a directory where the checkpointing service creates checkpoints. The directory is created if needed. Passing **`None`** disables the checkpointing and the summary services.
- **`checkpoint_basename`**: Name of the checkpoint files to create, defaults to "model.ckpt".

If the model contains a scalar integer variable named **`global_step`**, the value of that variable is appended to the checkpoint filename.

For example, at global step 1234 the checkpoint filename is "model.ckpt-1234".

- **`save_model_secs`**: Number of seconds between each checkpoint. Defaults to 600, or 10 minutes.

When choosing a value, consider how much work you want to lose in case of a crash: you will never lose more than **`save_model_secs`** seconds of work. Setting this to 0 disables the checkpointing service.

- **`saver`**: A **`tf.train.Saver`** (https://www.tensorflow.org/api_docs/python/tf/train/Saver) object to use for checkpointing.

If you do not pass one, the supervisor creates one for you by calling **`tf.train.Saver()`**, which add ops to save and restore all variables in your model. This is usually what you need.

Example: Use a custom Saver and checkpoint every 30 seconds.

```
...create graph...
my_saver = tf.train.Saver(<only some variables>)
sv = tf.train.Supervisor(logdir="/my/training/directory",
                         saver=my_saver,
                         save_model_secs=30)
with sv.managed_session() as sess:
  ...training loop...
```

Summaries: Where and When.

The **`managed_session()`** call launches the summary service which fetches summaries and reports the number of steps executed per second. It can be configured by the following keyword arguments to the **`Supervisor()`** constructor:

- **`logdir`**: Path to a directory where the summary service creates event files. The directory is created if needed. Passing **`None`** disables the summary service as well as the checkpointing services.
- **`save_summaries_secs`**: Number of seconds between each run of the summary service. Defaults to 120, or 2 minutes.

When choosing a value, consider how expensive your summaries are, and how much disk they will occupy. Pass 0 to disable the summary service.

- **`summary_op`**: Op to use to fetch the summaries.

If not specified, the supervisor use the first op in the **`tf.GraphKeys.SUMMARY_OP`** graph collection (https://www.tensorflow.org/api_docs/python/tf/Graph#add_to_collection). If the collection is empty the supervisor creates an op that aggregates all summaries in the graph using **`tf.summary.merge_all()`**.

Passing **`None`** disables the summary service.

- **`global_step`**: Tensor to use to count the global step.

If not specified, the supervisor uses the first tensor in the `tf.GraphKeys.GLOBAL_STEP` graph collection (https://www.tensorflow.org/api_docs/python/tf/Graph#add_to_collection). If the collection is empty, the supervisor looks for a scalar integer variable named `global_step` in the graph.

If found, the global step tensor is used to measure the number of training steps executed. Note that your training op is responsible for incrementing the global step value.

Model Initialization and Recovery

The `managed_session()` call takes care of initializing or recovering a session. It returns a session with a fully initialized model, ready to run ops. If a checkpoint exists in the logdir when `managed_session()` is called, the model is initialized by loading that checkpoint, otherwise it is initialized by calling an init op and optionally an init function.

When no checkpoint is available, model initialization is controlled by the following keyword arguments to the `Supervisor()` constructor:

- `init_op`: Op to run to initialize the model.

If not specified, the supervisor uses the first op in the `tf.GraphKeys.INIT_OP` collection. If the collection is empty, the supervisor adds an op to initialize all the variables in the graph by calling `tf.global_variables_initializer()`.

Pass `None` to not use an init op.

- `init_fn`: Python function to call to initialize the model.

If specified, called as `init_fn(sess)` where `sess` is the managed session. If an init op is also used, the init function is called *after* the init op.

- `local_init_op`: An additional op to initialize parts of the graph that are not saved in checkpoints such as tables and local variables (https://www.tensorflow.org/api_docs/python/tf/contrib/framework/local_variable). The local init op is run *before* the init op and the init function.

If not specified, the supervisor uses the first op in the `tf.GraphKeys.LOCAL_INIT_OP` collection. If the collection is empty the supervisor adds an op to initialize all the tables and local variables in the graph by calling `tf.initialize_all_tables()` and `tf.initialize_all_local_variables()`.

Pass `None` to not use a local init op.

- `ready_op`: Op to check if the model is initialized.

After running the local init op, the init op, and the init function, the supervisor verifies that the model is fully initialized by running the ready op. This is an op that returns an empty string if the model is initialized, or a description of what parts of the model are not initialized if not.

If not specified, the supervisor uses the first op in the `tf.GraphKeys.READY_OP` collection. If the collection is empty the supervisor creates a ready op that verifies that all variables are initialized by calling `tf.report_uninitialized_variables()`.

Pass `None` to disable the ready op. In that case the model is not checked after initialization.

Checkpoint recovery is controlled by the following keyword arguments to the `Supervisor()` constructor:

- `logdir`: Path to a directory in which to look for checkpoints. The checkpoint service saves a metadata file, named "checkpoint", in the checkpoint directory that indicates the path to the most recent checkpoint.

This file is in text format. When in a pinch, you can edit it manually to recover from a different checkpoint than the most recent one.

- `ready_op`: (see above). The ready op is run before and after loading the checkpoint. The first run checks if the model needs to be initialized and the second run verifies that the model is fully initialized.
- `local_init_op`: (see above). The local init op is run before running the ready op the first time, to initialize local variables and tables.
- `saver`: (see above). Saver object used to load the checkpoint.
