tf.train.SyncReplicasOptimizer

Class **SyncReplicasOptimizer**

Inherits From: `Optimizer`

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

Class to synchronize, aggregate gradients and pass them to the optimizer.

In a typical asynchronous training environment, it's common to have some stale gradients. For example, with a N-replica asynchronous training, gradients will be applied to the variables N times independently. Depending on each replica's training speed, some gradients might be calculated from copies of the variable from several steps back (N-1 steps on average). This optimizer avoids stale gradients by collecting gradients from all replicas, averaging them, then applying them to the variables in one shot, after which replicas can fetch the new variables and continue.

The following accumulators/queue are created: *N* `gradient accumulators`, *one per variable to train. Gradients are pushed to them and the chief worker will wait until enough gradients are collected and then average them before applying to variables. The accumulator will drop all stale gradients (more details in the accumulator op).* 1 `token` queue where the optimizer pushes the new global_step value after all variables are updated.

The following local variable is created: * `sync_rep_local_step`, one per replica. Compared against the global_step in each accumulator to check for staleness of the gradients.

The optimizer adds nodes to the graph to collect gradients and pause the trainers until variables are updated. For the Parameter Server job: 1. An accumulator is created for each variable, and each replica pushes the gradients into the accumulators instead of directly applying them to the variables. 2. Each accumulator averages once enough gradients (replicas_to_aggregate) have been accumulated. 3. Apply the averaged gradients to the variables. 4. Only after all variables have been updated, increment the global step. 5. Only after step 4, pushes `global_step` in the `token_queue`, once for each worker replica. The workers can now fetch the global step, use it to update its local_step variable and start the next batch.

For the replicas: 1. Start a step: fetch variables and compute gradients. 2. Once the gradients have been computed, push them into gradient accumulators. Each accumulator will check the staleness and drop the stale. 3. After pushing all the gradients, dequeue an updated value of global_step from the token queue and record that step to its local_step variable. Note that this is effectively a barrier. 4. Start the next batch.

Usage

```
# Create any optimizer to update the variables, say a simple SGD:
opt = GradientDescentOptimizer(learning_rate=0.1)

# Wrap the optimizer with sync_replicas_optimizer with 50 replicas: at each
# step the optimizer collects 50 gradients before applying to variables.
# Note that if you want to have 2 backup replicas, you can change
# total_num_replicas=52 and make sure this number matches how many physical
# replicas you started in your job.
opt = tf.SyncReplicasOptimizer(opt, replicas_to_aggregate=50,
                               total_num_replicas=50)

# Some models have startup_delays to help stabilize the model but when using
# sync_replicas training, set it to 0.

# Now you can call `minimize()` or `compute_gradients()` and
# `apply_gradients()` normally
training_op = opt.minimize(total_loss, global_step=self.global_step)

# You can create the hook which handles initialization and queues.
sync_replicas_hook = opt.make_session_run_hook(is_chief)
```

In the training program, every worker will run the train_op as if not synchronized.

```
with training.MonitoredTrainingSession(
    master=workers[worker_id].target, is_chief=is_chief,
    hooks=[sync_replicas_hook]) as mon_sess:
  while not mon_sess.should_stop():
    mon_sess.run(training_op)
```

To use SyncReplicasOptimizer with an `Estimator`, you need to send sync_replicas_hook while calling the fit.

```
my_estimator = DNNClassifier(..., optimizer=opt)
my_estimator.fit(..., hooks=[sync_replicas_hook])
```

Methods

`__init__`

```
__init__(
    opt,
    replicas_to_aggregate,
    total_num_replicas=None,
    variable_averages=None,
    variables_to_average=None,
    use_locking=False,
    name='sync_replicas'
)
```

Construct a sync_replicas optimizer.

Args:

- `opt`: The actual optimizer that will be used to compute and apply the gradients. Must be one of the Optimizer classes.
- `replicas_to_aggregate`: number of replicas to aggregate for each variable update.
- `total_num_replicas`: Total number of tasks/workers/replicas, could be different from replicas_to_aggregate. If total_num_replicas > replicas_to_aggregate: it is backup_replicas + replicas_to_aggregate. If total_num_replicas <

replicas_to_aggregate: Replicas compute multiple batches per update to variables.

- `variable_averages`: Optional `ExponentialMovingAverage` object, used to maintain moving averages for the variables passed in `variables_to_average`.
- `variables_to_average`: a list of variables that need to be averaged. Only needed if variable_averages is passed in.
- `use_locking`: If True use locks for update operation.
- `name`: string. Optional name of the returned operation.

`apply_gradients`

```
apply_gradients(
    grads_and_vars,
    global_step=None,
    name=None
)
```

Apply gradients to variables.

This contains most of the synchronization implementation and also wraps the apply_gradients() from the real optimizer.

Args:

- `grads_and_vars`: List of (gradient, variable) pairs as returned by compute_gradients().
- `global_step`: Optional Variable to increment by one after the variables have been updated.
- `name`: Optional name for the returned operation. Default to the name passed to the Optimizer constructor.

Returns:

- `train_op`: The op to dequeue a token so the replicas can exit this batch and start the next one. This is executed by each replica.

Raises:

- `ValueError`: If the grads_and_vars is empty.
- `ValueError`: If global step is not provided, the staleness cannot be checked.

`compute_gradients`

```
compute_gradients(
    *args,
    **kwargs
)
```

Compute gradients of "loss" for the variables in "var_list".

This simply wraps the compute_gradients() from the real optimizer. The gradients will be aggregated in the apply_gradients() so that user can modify the gradients like clipping with per replica global norm if needed. The global norm with aggregated gradients can be bad as one replica's huge gradients can hurt the gradients from other replicas.

Args:

- `*args`: Arguments for compute_gradients().

- `**kwargs`: Keyword arguments for compute_gradients().

Returns:

A list of (gradient, variable) pairs.

`get_chief_queue_runner`

```
get_chief_queue_runner()
```

Returns the QueueRunner for the chief to execute.

This includes the operations to synchronize replicas: aggregate gradients, apply to variables, increment global step, insert tokens to token queue.

Note that this can only be called after calling apply_gradients() which actually generates this queuerunner.

Returns:

A `QueueRunner` for chief to execute.

Raises:

- `ValueError`: If this is called before apply_gradients().

`get_init_tokens_op`

```
get_init_tokens_op(num_tokens=-1)
```

Returns the op to fill the sync_token_queue with the tokens.

This is supposed to be executed in the beginning of the chief/sync thread so that even if the total_num_replicas is less than replicas_to_aggregate, the model can still proceed as the replicas can compute multiple steps per variable update. Make sure: `num_tokens >= replicas_to_aggregate - total_num_replicas`.

Args:

- `num_tokens`: Number of tokens to add to the queue.

Returns:

An op for the chief/sync replica to fill the token queue.

Raises:

- `ValueError`: If this is called before apply_gradients().
- `ValueError`: If num_tokens are smaller than replicas_to_aggregate - total_num_replicas.

`get_name`

```
get_name()
```

get_slot

```
get_slot(
    *args,
    **kwargs
)
```

Return a slot named "name" created for "var" by the Optimizer.

This simply wraps the get_slot() from the actual optimizer.

Args:

- `*args`: Arguments for get_slot().
- `**kwargs`: Keyword arguments for get_slot().

Returns:

The `Variable` for the slot if it was created, `None` otherwise.

get_slot_names

```
get_slot_names(
    *args,
    **kwargs
)
```

Return a list of the names of slots created by the `Optimizer`.

This simply wraps the get_slot_names() from the actual optimizer.

Args:

- `*args`: Arguments for get_slot().
- `**kwargs`: Keyword arguments for get_slot().

Returns:

A list of strings.

make_session_run_hook

```
make_session_run_hook(
    is_chief,
    num_tokens=-1
)
```

Creates a hook to handle SyncReplicasHook ops such as initialization.

minimize

```
minimize(
    loss,
    global_step=None,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    name=None,
    grad_loss=None
)
```

Add operations to minimize `loss` by updating `var_list`.

This method simply combines calls `compute_gradients()` and `apply_gradients()`. If you want to process the gradient before applying them call `compute_gradients()` and `apply_gradients()` explicitly instead of using this function.

Args:

- `loss`: A `Tensor` containing the value to minimize.
- `global_step`: Optional `Variable` to increment by one after the variables have been updated.
- `var_list`: Optional list or tuple of `Variable` objects to update to minimize `loss`. Defaults to the list of variables collected in the graph under the key `GraphKeys.TRAINABLE_VARIABLES`.
- `gate_gradients`: How to gate the computation of gradients. Can be `GATE_NONE`, `GATE_OP`, or `GATE_GRAPH`.
- `aggregation_method`: Specifies the method used to combine gradient terms. Valid values are defined in the class `AggregationMethod`.
- `colocate_gradients_with_ops`: If True, try colocating gradients with the corresponding op.
- `name`: Optional name for the returned operation.
- `grad_loss`: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:

An Operation that updates the variables in `var_list`. If `global_step` was not `None`, that operation also increments `global_step`.

Raises:

- `ValueError`: If some of the variables are not `Variable` objects.

Class Members

GATE_GRAPH

GATE_NONE

GATE_OP

Stay Connected

Blog

GitHub

Twitter

Support

Issue Tracker

Release Notes

Stack Overflow

English

Terms | **Privacy**