tf.contrib.training.SequenceQueueingStateSaver

Class **`SequenceQueueingStateSaver`**

Defined in `tensorflow/contrib/training/python/training/sequence_queueing_state_saver.py`.

See the guide: Training (contrib) > Splitting sequence inputs into minibatches with state saving

SequenceQueueingStateSaver provides access to stateful values from input.

This class is meant to be used instead of, e.g., a `Queue`, for splitting variable-length sequence inputs into segments of sequences with fixed length and batching them into mini-batches. It maintains contexts and state for a sequence across the segments. It can be used in conjunction with a `QueueRunner` (see the example below).

The `SequenceQueueingStateSaver` (SQSS) accepts one example at a time via the inputs `input_length`, `input_key`, `input_sequences` (a dict), `input_context` (a dict), and `initial_states` (a dict). The sequences, values in `input_sequences`, may have variable first dimension (the `padded_length`), though this dimension must always be a multiple of `num_unroll`. All other dimensions must be fixed and accessible via `get_shape` calls. The length prior to padding can be recorded in `input_length`. The context values in `input_context` must all have fixed and well defined dimensions. The initial state values must all have fixed and well defined dimensions.

The SQSS splits the sequences of an input example into segments of length `num_unroll`. Across examples minibatches of size `batch_size` are formed. These minibatches contain a segment of the sequences, copy the context values, and maintain state, length, and key information of the original input examples. In the first segment of an example the state is still the initial state. It can then be updated; and updated state values are accessible in subsequent segments of the same example. After each segment `batch.save_state()` must be called which is done by the state_saving_rnn. Without this call, the dequeue op associated with the SQSS will not run. Internally, SQSS has a queue for the input examples. Its `capacity` is configurable. If set smaller than `batch_size` then the dequeue op will block indefinitely. A small multiple of `batch_size` is a good rule of thumb to prevent that queue from becoming a bottleneck and slowing down training. If set too large (and note that it defaults to unbounded) memory consumption goes up. Moreover, when iterating over the same input examples multiple times reusing the same `key` the `capacity` must be smaller than the number of examples.

The prefetcher, which reads one unrolled, variable-length input sequence at a time, is accessible via `prefetch_op`. The underlying `Barrier` object is accessible via `barrier`. Processed minibatches, as well as state read and write capabilities are accessible via `next_batch`. Specifically, `next_batch` provides access to all of the minibatched data, including the following, see `NextQueuedSequenceBatch` for details:

- `total_length`, `length`, `insertion_index`, `key`, `next_key`,
- `sequence` (the index each minibatch entry's time segment index),
- `sequence_count` (the total time segment count for each minibatch entry),
- `context` (a dict of the copied minibatched context values),

- `sequences` (a dict of the split minibatched variable-length sequences),
- `state` (to access the states of the current segments of these entries)
- `save_state` (to save the states for the next segments of these entries)

Example usage:

```
batch_size = 32
num_unroll = 20
lstm_size = 8
cell = tf.contrib.rnn.BasicLSTMCell(num_units=lstm_size)
initial_state_values = tf.zeros(cell.state_size, dtype=tf.float32)

raw_data = get_single_input_from_input_reader()
length, key, sequences, context = my_parser(raw_data)
assert "input" in sequences.keys()
assert "label" in context.keys()
initial_states = {"lstm_state": initial_state_value}

stateful_reader = tf.SequenceQueueingStateSaver(
    batch_size, num_unroll,
    length=length, input_key=key, input_sequences=sequences,
    input_context=context, initial_states=initial_states,
    capacity=batch_size*100)

batch = stateful_reader.next_batch
inputs = batch.sequences["input"]
context_label = batch.context["label"]

inputs_by_time = tf.split(value=inputs, num_or_size_splits=num_unroll, axis=1)
assert len(inputs_by_time) == num_unroll

lstm_output, _ = tf.contrib.rnn.static_state_saving_rnn(
  cell,
  inputs_by_time,
  state_saver=batch,
  state_name="lstm_state")

# Start a prefetcher in the background
sess = tf.Session()
num_threads = 3
queue_runner = tf.train.QueueRunner(
    stateful_reader, [stateful_reader.prefetch_op] * num_threads)
tf.train.add_queue_runner(queue_runner)
tf.train.start_queue_runners(sess=session)

while True:
  # Step through batches, perform training or inference...
  session.run([lstm_output])
```

Note: Usually the barrier is given to a QueueRunner as in the examples above. The QueueRunner will close the barrier if the prefetch_op receives an OutOfRange Error from upstream input queues (i.e., reaches the end of the input). If the barrier is closed no further new examples are added to the SQSS. The underlying barrier might, however, still contain further unroll-steps of examples that have not undergone all iterations. To gracefully finish all examples, the flag `allow_small_batch` must be set to true, which causes the SQSS to issue progressively smaller mini-batches with the remaining examples.

Properties

`barrier`

`batch_size`

`name`

`next_batch`

The `NextQueuedSequenceBatch` providing access to batched output data.

Also provides access to the `state` and `save_state` methods. The first time this gets called, it additionally prepares barrier reads and creates `NextQueuedSequenceBatch` / next_batch objects. Subsequent calls simply return the previously created `next_batch`.

In order to access data in `next_batch` without blocking, the `prefetch_op` must have been run at least `batch_size` times (ideally in a separate thread, or launched via a `QueueRunner`). After processing a segment in `next_batch()`, `batch.save_state()` must be called which is done by the state_saving_rnn. Without this call, the dequeue op associated with the SQSS will not run.

Returns:

A cached `NextQueuedSequenceBatch` instance.

`num_unroll`

`prefetch_op`

The op used to prefetch new data into the state saver.

Running it once enqueues one new input example into the state saver. The first time this gets called, it additionally creates the prefetch_op. Subsequent calls simply return the previously created `prefetch_op`.

It should be run in a separate thread via e.g. a `QueueRunner`.

Returns:

An `Operation` that performs prefetching.

Methods

`__init__`

```
__init__(
    batch_size,
    num_unroll,
    input_length,
    input_key,
    input_sequences,
    input_context,
    initial_states,
    capacity=None,
    allow_small_batch=False,
    name=None
)
```

Creates the SequenceQueueingStateSaver.

Args:

- `batch_size`: int or int32 scalar `Tensor`, how large minibatches should be when accessing the `state()` method and `context`, `sequences`, etc, properties.
- `num_unroll`: Python integer, how many time steps to unroll at a time. The input sequences of length `k` are then split into `k / num_unroll` many segments.
- `input_length`: An int32 scalar `Tensor`, the length of the sequence prior to padding. This value may be at most `padded_length` for any given input (see below for the definition of `padded_length`). Batched and total lengths of the current iteration are made accessible via the `length` and `total_length` properties. The shape of input_length (scalar) must be fully specified.
- `input_key`: A string scalar `Tensor`, the **unique** key for the given input. This is used to keep track of the split minibatch elements of this input. Batched keys of the current iteration are made accessible via the `key` property. The shape of `input_key` (scalar) must be fully specified.
- `input_sequences`: A dict mapping string names to `Tensor` values. The values must all have matching first dimension, called `padded_length`. The `SequenceQueueingStateSaver` will split these tensors along this first dimension into minibatch elements of dimension `num_unroll`. Batched and segmented sequences of the current iteration are made accessible via the `sequences` property.

  Note: `padded_length` may be dynamic, and may vary from input to input, but must always be a multiple of `num_unroll`. The remainder of the shape (other than the first dimension) must be fully specified. * `input_context`: A dict mapping string names to `Tensor` values. The values are treated as "global" across all time splits of the given input, and will be copied across for all minibatch elements accordingly. Batched and copied context of the current iteration are made accessible via the `context` property.

  Note: All input_context values must have fully defined shapes. * `initial_states`: A dict mapping string state names to multi-dimensional values (e.g. constants or tensors). This input defines the set of states that will be kept track of during computing iterations, and which can be accessed via the `state` and `save_state` methods.

  Note: All initial_state values must have fully defined shapes. `capacity`*: The max capacity of the SQSS in number of examples. Needs to be at least* `batch_size`*. Defaults to unbounded.* `allow_small_batch`: If true, the SQSS will return smaller batches when there aren't enough input examples to fill a whole batch and the end of the input has been reached (i.e., the underlying barrier has been closed). * `name`: An op name string (optional).

Raises:

- `TypeError`: if any of the inputs is not an expected type.
- `ValueError`: if any of the input values is inconsistent, e.g. if not enough shape information is available from inputs to build the state saver.

`close`

```
close(
    cancel_pending_enqueues=False,
    name=None
)
```

Closes the barrier and the FIFOQueue.

This operation signals that no more segments of new sequences will be enqueued. New segments of already inserted sequences may still be enqueued and dequeued if there is a sufficient number filling a batch or allow_small_batch is true. Otherwise dequeue operations will fail immediately.

Args:

- `cancel_pending_enqueues`: (Optional.) A boolean, defaulting to `False`. If `True`, all pending enqueues to the underlying queues will be cancelled, and completing already started sequences is not possible.

- `name`: Optional name for the op.

Returns:

The operation that closes the barrier and the FIFOQueue.
