tf.contrib.training.NextQueuedSequenceBatch

Contents

Class **NextQueuedSequenceBatch**

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

NextQueuedSequenceBatch stores deferred SequenceQueueingStateSaver data.

This class is instantiated by `SequenceQueueingStateSaver` and is accessible via its `next_batch` property.

Properties

`batch_size`

The batch_size of the given batch.

Usually, this is the batch_size requested when initializing the SQSS, but if allow_small_batch=True this will become smaller when inputs are exhausted.

Returns:

A scalar integer tensor, the batch_size

`context`

A dict mapping keys of `input_context` to batched context.

Returns:

A dict mapping keys of `input_context` to tensors. If we had at input:

```
context["name"].get_shape() == [d1, d2, ...]
```

then for this property:

```
context["name"].get_shape() == [batch_size, d1, d2, ...]
```

`insertion_index`

The insertion indices of the examples (when they were first added).

These indices start with the value -2**63 and increase with every call to the prefetch op. Each whole example gets its own insertion index, and this is used to prioritize the example so that its truncated segments appear in adjacent iterations, even if new examples are inserted by the prefetch op between iterations.

Returns:

An int64 vector of length `batch_size`, the insertion indices.

`key`

The key names of the given truncated unrolled examples.

The format of the key is:

```
"%05d_of_%05d:%s" % (sequence, sequence_count, original_key)
```

where `original_key` is the unique key read in by the prefetcher.

Returns:

A string vector of length `batch_size`, the keys.

`length`

The lengths of the given truncated unrolled examples.

For initial iterations, for which `sequence * num_unroll < length`, this number is `num_unroll`. For the remainder, this number is between `0` and `num_unroll`.

Returns:

An integer vector of length `batch_size`, the lengths.

`next_key`

The key names of the next (in iteration) truncated unrolled examples.

The format of the key is:

```
"%05d_of_%05d:%s" % (sequence + 1, sequence_count, original_key)
```

if `sequence + 1 < sequence_count`, otherwise:

```
"STOP:%s" % original_key
```

where `original_key` is the unique key read in by the prefetcher.

Returns:

A string vector of length `batch_size`, the keys.

`sequence`

An int32 vector, length `batch_size`: the sequence index of each entry.

When an input is split up, the sequence values

```
0, 1, ..., sequence_count - 1
```

are assigned to each split.

Returns:

An int32 vector `Tensor`.

`sequence_count`

An int32 vector, length `batch_size`: the sequence count of each entry.

When an input is split up, the number of splits is equal to: `padded_length / num_unroll`. This is the sequence_count.

Returns:

An int32 vector `Tensor`.

`sequences`

A dict mapping keys of `input_sequences` to split and rebatched data.

Returns:

A dict mapping keys of `input_sequences` to tensors. If we had at input:

```
sequences["name"].get_shape() == [None, d1, d2, ...]
```

where `None` meant the sequence time was dynamic, then for this property:

```
sequences["name"].get_shape() == [batch_size, num_unroll, d1, d2, ...].
```

`total_length`

The lengths of the original (non-truncated) unrolled examples.

Returns:

An integer vector of length `batch_size`, the total lengths.

Methods

`__init__`

```
__init__(state_saver)
```

save_state

```
save_state(
    state_name,
    value,
    name=None
)
```

Returns an op to save the current batch of state `state_name`.

Args:

- `state_name`: string, matches a key provided in `initial_states`.
- `value`: A `Tensor`. Its type must match that of `initial_states[state_name].dtype`. If we had at input:

  `python initial_states[state_name].get_shape() == [d1, d2, ...]`

  then the shape of `value` must match:

  `python tf.shape(value) == [batch_size, d1, d2, ...]`
- `name`: string (optional). The name scope for newly created ops.

Returns:

A control flow op that stores the new state of each entry into the state saver. This op must be run for every iteration that accesses data from the state saver (otherwise the state saver will never progress through its states and run out of capacity).

Raises:

- `KeyError`: if `state_name` does not match any of the initial states declared in `initial_states`.

state

```
state(state_name)
```

Returns batched state tensors.

Args:

- `state_name`: string, matches a key provided in `initial_states`.

Returns:

A `Tensor`: a batched set of states, either initial states (if this is the first run of the given example), or a value as stored during a previous iteration via `save_state` control flow. Its type is the same as `initial_states["state_name"].dtype`. If we had at input:

```
initial_states[state_name].get_shape() == [d1, d2, ...],
```

then

```
state(state_name).get_shape() == [batch_size, d1, d2, ...]
```

Raises:

- `KeyError`: if `state_name` does not match any of the initial states declared in `initial_states`.

Last updated November 2, 2017.