tf.contrib.training.batch_sequences_with_states

```
batch_sequences_with_states(
    input_key,
    input_sequences,
    input_context,
    input_length,
    initial_states,
    num_unroll,
    batch_size,
    num_threads=3,
    capacity=1000,
    allow_small_batch=True,
    pad=True,
    make_keys_unique=False,
    make_keys_unique_seed=None,
    name=None
)
```

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

Creates batches of segments of sequential input.

This method creates a `SequenceQueueingStateSaver` (SQSS) and adds it to the queuerunners. It returns a `NextQueuedSequenceBatch`.

It accepts one example at a time identified by a unique `input_key`. `input_sequence` is a dict with values that are tensors with time as first dimension. This time dimension must be the same across those tensors of an example. It can vary across examples. Although it always has to be a multiple of `num_unroll`. Hence, padding may be necessary and it is turned on by default by `pad=True`.

`input_length` is a Tensor scalar or an int recording the time dimension prior to padding. It should be between 0 and the time dimension. One reason we want to keep track of it is so that we can take it into consideration when computing the loss. If `pad=True` then `input_length` can be `None` and will be inferred.

This methods segments `input_sequence` into segments of length `num_unroll`. It batches input sequences from `batch_size` many examples. These mini-batches are available through the `sequence` property of the output. Moreover, for each entry in the batch we can access its original `input_key` in `key` and its input length in `total_length`. `length` records within this segment how many non-padded time steps there are.

Static features of an example that do not vary across time can be part of the `input_context`, a dict with Tensor values. This method copies the context for each segment and makes it available in the `context` of the output.

This method can maintain and update a state for each example. It accepts some initial_states as a dict with Tensor values. The first mini-batch an example is contained has initial_states as entry of the `state`. If save_state is called then the next segment will have the updated entry of the `state`. See `NextQueuedSequenceBatch` for a complete list of properties and methods.

Example usage:

```
batch_size = 32
num_unroll = 20
num_enqueue_threads = 3
lstm_size = 8
cell = tf.contrib.rnn.BasicLSTMCell(num_units=lstm_size)

key, sequences, context = my_parser(raw_data)
initial_state_values = tf.zeros((state_size,), dtype=tf.float32)
initial_states = {"lstm_state": initial_state_values}
batch = tf.batch_sequences_with_states(
    input_key=key,
    input_sequences=sequences,
    input_context=context,
    input_length=tf.shape(sequences["input"])[0],
    initial_states=initial_states,
    num_unroll=num_unroll,
    batch_size=batch_size,
    num_threads=num_enqueue_threads,
    capacity=batch_size * num_enqueue_threads * 2)

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()

tf.train.start_queue_runners(sess=session)

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

Args:

- `input_key`: A string scalar `Tensor`, the **unique** key for the given input example. 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. Consider setting `make_keys_unique` to True when iterating over the same input multiple times.

  Note: if `make_keys_unique=False` then `input_key`s must be unique. * `input_sequences`: A dict mapping string names to `Tensor` values. The values must all have matching first dimension, called `value_length`. They may vary from input to input. The remainder of the shape (other than the first dimension) must be fully specified. The `SequenceQueueingStateSaver` will split these tensors along this first dimension into minibatch elements of dimension `num_unrolled`. Batched and segmented sequences of the current iteration are made accessible via the `sequences` property.

  Note: if `pad=False`, then `value_length` must always be a multiple of `num_unroll`. * `input_context`: A dict mapping string names to `Tensor` values. The values are treated as "global" across all time splits of the given input example, 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. `input_length`: *None or an int32 scalar* `Tensor`, *the length of the sequence prior to padding. If* `input_length=None` *and* `pad=True` *then the length will be inferred and will*

be equal to `value_length`. *If* `pad=False` *then* `input_length` *cannot be* `None`: `input_length` *must be specified. Its shape of* `input_length` *(scalar) must be fully specified. Its value may be at most* `value_length` *for any given input (see above for the definition of* `value_length`*). Batched and total lengths of the current iteration are made accessible via the* `length` *and* `total_length` *properties.* `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. `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.* `batch_size`: int or int32 scalar `Tensor`, how large minibatches should be when accessing the `state()` method and `context`, `sequences`, etc, properties. `num_threads`*: The int number of threads enqueuing input examples into a queue.* `capacity`: The max capacity of the queue in number of examples. Needs to be at least `batch_size`. Defaults to 1000. When iterating over the same input example multiple times reusing their keys the `capacity` must be smaller than the number of examples. `allow_small_batch`*: If true, the queue 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.* `pad`: If `True`, `input_sequences` will be padded to multiple of `num_unroll`. In that case `input_length` may be `None` and is assumed to be the length of first dimension of values in `input_sequences` (i.e. `value_length`). `make_keys_unique`*: Whether to append a random integer to the* `input_key` *in an effort to make it unique. The seed can be set via* `make_keys_unique_seed`. `make_keys_unique_seed`: If `make_keys_unique=True` this fixes the seed with which a random postfix is generated. * `name`: An op name string (optional).

Returns:

A NextQueuedSequenceBatch with segmented and batched inputs and their states.

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.