tf.nn.raw_rnn

```
raw_rnn(
    cell,
    loop_fn,
    parallel_iterations=None,
    swap_memory=False,
    scope=None
)
```

Defined in `tensorflow/python/ops/rnn.py`.

See the guide: Neural Network > Recurrent Neural Networks

Creates an `RNN` specified by RNNCell `cell` and loop function `loop_fn`.

NOTE: This method is still in testing, and the API may change.

This function is a more primitive version of `dynamic_rnn` that provides more direct access to the inputs each iteration. It also provides more control over when to start and finish reading the sequence, and what to emit for the output.

For example, it can be used to implement the dynamic decoder of a seq2seq model.

Instead of working with `Tensor` objects, most operations work with `TensorArray` objects directly.

The operation of `raw_rnn`, in pseudo-code, is basically the following:

```
time = tf.constant(0, dtype=tf.int32)
(finished, next_input, initial_state, _, loop_state) = loop_fn(
    time=time, cell_output=None, cell_state=None, loop_state=None)
emit_ta = TensorArray(dynamic_size=True, dtype=initial_state.dtype)
state = initial_state
while not all(finished):
  (output, cell_state) = cell(next_input, state)
  (next_finished, next_input, next_state, emit, loop_state) = loop_fn(
      time=time + 1, cell_output=output, cell_state=cell_state,
      loop_state=loop_state)
  # Emit zeros and copy forward state for minibatch entries that are finished.
  state = tf.where(finished, state, next_state)
  emit = tf.where(finished, tf.zeros_like(emit), emit)
  emit_ta = emit_ta.write(time, emit)
  # If any new minibatch entries are marked as finished, mark these.
  finished = tf.logical_or(finished, next_finished)
  time += 1
return (emit_ta, state, loop_state)
```

with the additional properties that output and state may be (possibly nested) tuples, as determined by `cell.output_size` and `cell.state_size`, and as a result the final `state` and `emit_ta` may themselves be tuples.

A simple implementation of `dynamic_rnn` via `raw_rnn` looks like this:

```
inputs = tf.placeholder(shape=(max_time, batch_size, input_depth),
                        dtype=tf.float32)
sequence_length = tf.placeholder(shape=(batch_size,), dtype=tf.int32)
inputs_ta = tf.TensorArray(dtype=tf.float32, size=max_time)
inputs_ta = inputs_ta.unstack(inputs)

cell = tf.contrib.rnn.LSTMCell(num_units)

def loop_fn(time, cell_output, cell_state, loop_state):
  emit_output = cell_output  # == None for time == 0
  if cell_output is None:  # time == 0
    next_cell_state = cell.zero_state(batch_size, tf.float32)
  else:
    next_cell_state = cell_state
  elements_finished = (time >= sequence_length)
  finished = tf.reduce_all(elements_finished)
  next_input = tf.cond(
      finished,
      lambda: tf.zeros([batch_size, input_depth], dtype=tf.float32),
      lambda: inputs_ta.read(time))
  next_loop_state = None
  return (elements_finished, next_input, next_cell_state,
          emit_output, next_loop_state)

outputs_ta, final_state, _ = raw_rnn(cell, loop_fn)
outputs = outputs_ta.stack()
```

Args:

- `cell`: An instance of RNNCell.
- `loop_fn`: A callable that takes inputs `(time, cell_output, cell_state, loop_state)` and returns the tuple `(finished, next_input, next_cell_state, emit_output, next_loop_state)`. Here `time` is an int32 scalar `Tensor`, `cell_output` is a `Tensor` or (possibly nested) tuple of tensors as determined by `cell.output_size`, and `cell_state` is a `Tensor` or (possibly nested) tuple of tensors, as determined by the `loop_fn` on its first call (and should match `cell.state_size`). The outputs are: `finished`, a boolean `Tensor` of shape `[batch_size]`, `next_input`: the next input to feed to `cell`, `next_cell_state`: the next state to feed to `cell`, and `emit_output`: the output to store for this iteration.

  Note that `emit_output` should be a `Tensor` or (possibly nested) tuple of tensors with shapes and structure matching `cell.output_size` and `cell_output` above. The parameter `cell_state` and output `next_cell_state` may be either a single or (possibly nested) tuple of tensors. The parameter `loop_state` and output `next_loop_state` may be either a single or (possibly nested) tuple of `Tensor` and `TensorArray` objects. This last parameter may be ignored by `loop_fn` and the return value may be `None`. If it is not `None`, then the `loop_state` will be propagated through the RNN loop, for use purely by `loop_fn` to keep track of its own state. The `next_loop_state` parameter returned may be `None`.

  The first call to `loop_fn` will be `time = 0`, `cell_output = None`, `cell_state = None`, and `loop_state = None`. For this call: The `next_cell_state` value should be the value with which to initialize the cell's state. It may be a final state from a previous RNN or it may be the output of `cell.zero_state()`. It should be a (possibly nested) tuple structure of tensors. If `cell.state_size` is an integer, this must be a `Tensor` of appropriate type and shape `[batch_size, cell.state_size]`. If `cell.state_size` is a `TensorShape`, this must be a `Tensor` of appropriate type and shape `[batch_size] + cell.state_size`. If `cell.state_size` is a (possibly nested) tuple of ints or `TensorShape`, this will be a tuple having the corresponding shapes. The `emit_output` value may be either `None` or a (possibly nested) tuple structure of tensors, e.g., `(tf.zeros(shape_0, dtype=dtype_0), tf.zeros(shape_1, dtype=dtype_1))`. If this first `emit_output` return value is `None`, then the `emit_ta` result of `raw_rnn` will have the same structure and dtypes as `cell.output_size`. Otherwise `emit_ta` will have the same structure, shapes (prepended with a `batch_size` dimension), and dtypes as `emit_output`. The actual values returned for `emit_output` at this initializing call are ignored. Note, this emit structure must be consistent across all time steps.

- `parallel_iterations` : (Default: 32). The number of iterations to run in parallel. Those operations which do not have any temporal dependency and can be run in parallel, will be. This parameter trades off time for space. Values >> 1 use more memory but take less time, while smaller values use less memory but computations take longer.
- `swap_memory` : Transparently swap the tensors produced in forward inference but needed for back prop from GPU to CPU. This allows training RNNs which would typically not fit on a single GPU, with very minimal (or no) performance penalty.
- `scope` : VariableScope for the created subgraph; defaults to "rnn".

Returns:

A tuple `(emit_ta, final_state, final_loop_state)` where:

`emit_ta` : The RNN output `TensorArray` . If `loop_fn` returns a (possibly nested) set of Tensors for `emit_output` during initialization, (inputs `time = 0` , `cell_output = None` , and `loop_state = None`), then `emit_ta` will have the same structure, dtypes, and shapes as `emit_output` instead. If `loop_fn` returns `emit_output = None` during this call, the structure of `cell.output_size` is used: If `cell.output_size` is a (possibly nested) tuple of integers or `TensorShape` objects, then `emit_ta` will be a tuple having the same structure as `cell.output_size` , containing TensorArrays whose elements' shapes correspond to the shape data in `cell.output_size` .

`final_state` : The final cell state. If `cell.state_size` is an int, this will be shaped `[batch_size, cell.state_size]` . If it is a `TensorShape` , this will be shaped `[batch_size] + cell.state_size` . If it is a (possibly nested) tuple of ints or `TensorShape` , this will be a tuple having the corresponding shapes.

`final_loop_state` : The final loop state as returned by `loop_fn` .

Raises:

- `TypeError` : If `cell` is not an instance of RNNCell, or `loop_fn` is not a `callable` .

Last updated November 2, 2017.