tf.contrib.seq2seq.AttentionWrapper

Contents

Class `AttentionWrapper`

Inherits From: `RNNCell`

Defined in `tensorflow/contrib/seq2seq/python/ops/attention_wrapper.py`.

See the guide: Seq2seq Library (contrib) > Attention

Wraps another `RNNCell` with attention.

Properties

`activity_regularizer`

Optional regularizer function for the output of this layer.

`dtype`

`graph`

`input`

Retrieves the input tensor(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer.

Returns:

Input tensor or list of input tensors.

Raises:

- `AttributeError`: if the layer is connected to more than one incoming layers.

Raises:

- `RuntimeError`: If called in Eager mode.

- `AttributeError`: If no inbound nodes are found.

`input_shape`

Retrieves the input shape(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer, or if all inputs have the same shape.

Returns:

Input shape, as an integer shape tuple (or list of shape tuples, one tuple per input tensor).

Raises:

- `AttributeError`: if the layer has no defined input_shape.
- `RuntimeError`: if called in Eager mode.

`losses`

`name`

`non_trainable_variables`

`non_trainable_weights`

`output`

Retrieves the output tensor(s) of a layer.

Only applicable if the layer has exactly one output, i.e. if it is connected to one incoming layer.

Returns:

Output tensor or list of output tensors.

Raises:

- `AttributeError`: if the layer is connected to more than one incoming layers.
- `RuntimeError`: if called in Eager mode.

`output_shape`

Retrieves the output shape(s) of a layer.

Only applicable if the layer has one output, or if all outputs have the same shape.

Returns:

Output shape, as an integer shape tuple (or list of shape tuples, one tuple per output tensor).

Raises:

- `AttributeError`: if the layer has no defined output shape.
- `RuntimeError`: if called in Eager mode.

output_size

scope_name

state_size

The `state_size` property of `AttentionWrapper`.

Returns:

An `AttentionWrapperState` tuple containing shapes used by this object.

trainable_variables

trainable_weights

updates

variables

Returns the list of all layer variables/weights.

Returns:

A list of variables.

weights

Returns the list of all layer variables/weights.

Returns:

A list of variables.

Methods

__init__

```
__init__(
    cell,
    attention_mechanism,
    attention_layer_size=None,
    alignment_history=False,
    cell_input_fn=None,
    output_attention=True,
    initial_cell_state=None,
    name=None
)
```

Construct the `AttentionWrapper`.

NOTE If you are using the `BeamSearchDecoder` with a cell wrapped in `AttentionWrapper`, then you must ensure that:

- The encoder output has been tiled to `beam_width` via `tf.contrib.seq2seq.tile_batch` (NOT `tf.tile`).
- The `batch_size` argument passed to the `zero_state` method of this wrapper is equal to `true_batch_size * beam_width`.
- The initial state created with `zero_state` above contains a `cell_state` value containing properly tiled final state from the encoder.

An example:

```
tiled_encoder_outputs = tf.contrib.seq2seq.tile_batch(
    encoder_outputs, multiplier=beam_width)
tiled_encoder_final_state = tf.conrib.seq2seq.tile_batch(
    encoder_final_state, multiplier=beam_width)
tiled_sequence_length = tf.contrib.seq2seq.tile_batch(
    sequence_length, multiplier=beam_width)
attention_mechanism = MyFavoriteAttentionMechanism(
    num_units=attention_depth,
    memory=tiled_inputs,
    memory_sequence_length=tiled_sequence_length)
attention_cell = AttentionWrapper(cell, attention_mechanism, ...)
decoder_initial_state = attention_cell.zero_state(
    dtype, batch_size=true_batch_size * beam_width)
decoder_initial_state = decoder_initial_state.clone(
    cell_state=tiled_encoder_final_state)
```

Args:

- `cell`: An instance of `RNNCell`.
- `attention_mechanism`: A list of `AttentionMechanism` instances or a single instance.
- `attention_layer_size`: A list of Python integers or a single Python integer, the depth of the attention (output) layer(s). If None (default), use the context as attention at each time step. Otherwise, feed the context and cell output into the attention layer to generate attention at each time step. If attention_mechanism is a list, attention_layer_size must be a list of the same length.
- `alignment_history`: Python boolean, whether to store alignment history from all time steps in the final output state (currently stored as a time major `TensorArray` on which you must call `stack()`).
- `cell_input_fn`: (optional) A `callable`. The default is: `lambda inputs, attention: array_ops.concat([inputs, attention], -1)`.
- `output_attention`: Python bool. If `True` (default), the output at each time step is the attention value. This is the behavior of Luong-style attention mechanisms. If `False`, the output at each time step is the output of `cell`. This is the beahvior of Bhadanau-style attention mechanisms. In both cases, the `attention` tensor is propagated to the next time step via the state and is used there. This flag only controls whether the attention mechanism is propagated up to the next cell in an RNN stack or to the top RNN output.

- `initial_cell_state`: The initial state value to use for the cell when the user calls `zero_state()`. Note that if this value is provided now, and the user uses a `batch_size` argument of `zero_state` which does not match the batch size of `initial_cell_state`, proper behavior is not guaranteed.
- `name`: Name to use when creating ops.

Raises:

- `TypeError`: `attention_layer_size` is not None and (`attention_mechanism` is a list but `attention_layer_size` is not; or vice versa).
- `ValueError`: if `attention_layer_size` is not None, `attention_mechanism` is a list, and its length does not match that of `attention_layer_size`.

__call__

```
__call__(
    inputs,
    state,
    scope=None
)
```

Run this RNN cell on inputs, starting from the given state.

Args:

- `inputs`: `2-D` tensor with shape `[batch_size x input_size]`.
- `state`: if `self.state_size` is an integer, this should be a `2-D Tensor` with shape `[batch_size x self.state_size]`. Otherwise, if `self.state_size` is a tuple of integers, this should be a tuple with shapes `[batch_size x s] for s in self.state_size`.
- `scope`: VariableScope for the created subgraph; defaults to class name.

Returns:

A pair containing:

- Output: A `2-D` tensor with shape `[batch_size x self.output_size]`.
- New state: Either a single `2-D` tensor, or a tuple of tensors matching the arity and shapes of `state`.

__deepcopy__

```
__deepcopy__(memo)
```

add_loss

```
add_loss(
    losses,
    inputs=None
)
```

Add loss tensor(s), potentially dependent on layer inputs.

Some losses (for instance, activity regularization losses) may be dependent on the inputs passed when calling a layer.

Hence, when reusing a same layer on different inputs `a` and `b`, some entries in `layer.losses` may be dependent on `a` and some on `b`. This method automatically keeps track of dependencies.

The `get_losses_for` method allows to retrieve the losses relevant to a specific set of inputs.

Arguments:

- `losses`: Loss tensor, or list/tuple of tensors.
- `inputs`: Optional input tensor(s) that the loss(es) depend on. Must match the `inputs` argument passed to the `__call__` method at the time the losses are created. If `None` is passed, the losses are assumed to be unconditional, and will apply across all dataflows of the layer (e.g. weight regularization losses).

Raises:

- `RuntimeError`: If called in Eager mode.

add_update

```
add_update(
    updates,
    inputs=None
)
```

Add update op(s), potentially dependent on layer inputs.

Weight updates (for instance, the updates of the moving mean and variance in a BatchNormalization layer) may be dependent on the inputs passed when calling a layer. Hence, when reusing a same layer on different inputs `a` and `b`, some entries in `layer.updates` may be dependent on `a` and some on `b`. This method automatically keeps track of dependencies.

The `get_updates_for` method allows to retrieve the updates relevant to a specific set of inputs.

This call is ignored in Eager mode.

Arguments:

- `updates`: Update op, or list/tuple of update ops.
- `inputs`: Optional input tensor(s) that the update(s) depend on. Must match the `inputs` argument passed to the `__call__` method at the time the updates are created. If `None` is passed, the updates are assumed to be unconditional, and will apply across all dataflows of the layer.

add_variable

```
add_variable(
    name,
    shape,
    dtype=None,
    initializer=None,
    regularizer=None,
    trainable=True,
    constraint=None
)
```

Adds a new variable to the layer, or gets an existing one; returns it.

Arguments:

- `name`: variable name.
- `shape`: variable shape.
- `dtype`: The type of the variable. Defaults to `self.dtype` or `float32`.
- `initializer`: initializer instance (callable).
- `regularizer`: regularizer instance (callable).
- `trainable`: whether the variable should be part of the layer's "trainable_variables" (e.g. variables, biases) or "non_trainable_variables" (e.g. BatchNorm mean, stddev).
- `constraint`: constraint instance (callable).

Returns:

The created variable.

Raises:

- `RuntimeError`: If called in Eager mode with regularizers.

`apply`

```
apply(
    inputs,
    *args,
    **kwargs
)
```

Apply the layer on a input.

This simply wraps `self.__call__`.

Arguments:

- `inputs`: Input tensor(s).
- `*args`: additional positional arguments to be passed to `self.call`.
- `**kwargs`: additional keyword arguments to be passed to `self.call`.

Returns:

Output tensor(s).

`build`

```
build(_)
```

`call`

```
call(
    inputs,
    state
)
```

Perform a step of attention-wrapped RNN.

- Step 1: Mix the `inputs` and previous step's `attention` output via `cell_input_fn`.
- Step 2: Call the wrapped `cell` with this input and its previous state.
- Step 3: Score the cell's output with `attention_mechanism`.
- Step 4: Calculate the alignments by passing the score through the `normalizer`.
- Step 5: Calculate the context vector as the inner product between the alignments and the attention_mechanism's values (memory).
- Step 6: Calculate the attention output by concatenating the cell output and context through the attention layer (a linear layer with `attention_layer_size` outputs).

Args:

- `inputs`: (Possibly nested tuple of) Tensor, the input at this time step.
- `state`: An instance of `AttentionWrapperState` containing tensors from the previous time step.

Returns:

A tuple `(attention_or_cell_output, next_state)`, where:

- `attention_or_cell_output` depending on `output_attention`.
- `next_state` is an instance of `AttentionWrapperState` containing the state calculated at this time step.

Raises:

- `TypeError`: If `state` is not an instance of `AttentionWrapperState`.

count_params

```
count_params()
```

Count the total number of scalars composing the weights.

Returns:

An integer count.

Raises:

- `ValueError`: if the layer isn't yet built (in which case its weights aren't yet defined).

get_input_at

```
get_input_at(node_index)
```

Retrieves the input tensor(s) of a layer at a given node.

Arguments:

- `node_index`: Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns:

A tensor (or list of tensors if the layer has multiple inputs).

Raises:

- `RuntimeError`: If called in Eager mode.

`get_input_shape_at`

```
get_input_shape_at(node_index)
```

Retrieves the input shape(s) of a layer at a given node.

Arguments:

- `node_index`: Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns:

A shape tuple (or list of shape tuples if the layer has multiple inputs).

Raises:

- `RuntimeError`: If called in Eager mode.

`get_losses_for`

```
get_losses_for(inputs)
```

Retrieves losses relevant to a specific set of inputs.

Arguments:

- `inputs`: Input tensor or list/tuple of input tensors. Must match the `inputs` argument passed to the `__call__` method at the time the losses were created. If you pass `inputs=None`, unconditional losses are returned, such as weight regularization losses.

Returns:

List of loss tensors of the layer that depend on `inputs`.

Raises:

- `RuntimeError`: If called in Eager mode.

`get_output_at`

```
get_output_at(node_index)
```

Retrieves the output tensor(s) of a layer at a given node.

Arguments:

- `node_index`: Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns:

A tensor (or list of tensors if the layer has multiple outputs).

Raises:

- `RuntimeError`: If called in Eager mode.

`get_output_shape_at`

```
get_output_shape_at(node_index)
```

Retrieves the output shape(s) of a layer at a given node.

Arguments:

- `node_index`: Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns:

A shape tuple (or list of shape tuples if the layer has multiple outputs).

Raises:

- `RuntimeError`: If called in Eager mode.

`get_updates_for`

```
get_updates_for(inputs)
```

Retrieves updates relevant to a specific set of inputs.

Arguments:

- `inputs`: Input tensor or list/tuple of input tensors. Must match the `inputs` argument passed to the `__call__` method at the time the updates were created. If you pass `inputs=None`, unconditional updates are returned.

Returns:

List of update ops of the layer that depend on `inputs`.

Raises:

- `RuntimeError`: If called in Eager mode.

zero_state

```
zero_state(
    batch_size,
    dtype
)
```

Return an initial (zero) state tuple for this `AttentionWrapper`.

NOTE Please see the initializer documentation for details of how to call `zero_state` if using an `AttentionWrapper` with a `BeamSearchDecoder`.

Args:

- `batch_size`: `0D` integer tensor: the batch size.
- `dtype`: The internal state data type.

Returns:

An `AttentionWrapperState` tuple containing zeroed out tensors and, possibly, empty `TensorArray` objects.

Raises:

- `ValueError`: (or, possibly at runtime, InvalidArgument), if `batch_size` does not match the output size of the encoder passed to the wrapper object at initialization time.