AdditiveAttention layer

`AdditiveAttention` class

[source]

```
tf.keras.layers.AdditiveAttention(use_scale=True, **kwargs)
```

Additive attention layer, a.k.a. Bahdanau-style attention.

Inputs are `query` tensor of shape `[batch_size, Tq, dim]`, `value` tensor of shape `[batch_size, Tv, dim]` and `key` tensor of shape `[batch_size, Tv, dim]`. The calculation follows the steps:

1. Reshape `query` and `key` into shapes `[batch_size, Tq, 1, dim]` and `[batch_size, 1, Tv, dim]` respectively.
2. Calculate scores with shape `[batch_size, Tq, Tv]` as a non-linear sum: `scores = tf.reduce_sum(tf.tanh(query + key), axis=-1)`
3. Use scores to calculate a distribution with shape `[batch_size, Tq, Tv]`: `distribution = tf.nn.softmax(scores)`.
4. Use `distribution` to create a linear combination of `value` with shape `[batch_size, Tq, dim]`: `return tf.matmul(distribution, value)`.

Arguments

- **use_scale**: If `True`, will create a variable to scale the attention scores.
- **causal**: Boolean. Set to `True` for decoder self-attention. Adds a mask such that position `i` cannot attend to positions `j > i`. This prevents the flow of information from the future towards the past. Defaults to `False`.
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the attention scores. Defaults to 0.0.

Call # Arguments

inputs: List of the following tensors: * query: Query `Tensor` of shape `[batch_size, Tq, dim]`. * value: Value `Tensor` of shape `[batch_size, Tv, dim]`. * key: Optional key `Tensor` of shape `[batch_size, Tv, dim]`. If not given, will use `value` for both `key` and `value`, which is the most common case. mask: List of the following tensors: * query_mask: A boolean mask `Tensor` of shape `[batch_size, Tq]`. If given, the output will be zero at the positions where `mask==False`. * value_mask: A boolean mask `Tensor` of shape `[batch_size, Tv]`. If given, will apply the mask such that values at positions where `mask==False` do not contribute to the result. training: Python boolean indicating whether the layer should behave in training mode (adding dropout) or in inference mode (no dropout). return_attention_scores: bool, it `True`, returns the attention scores (after masking and softmax) as an additional output argument.

Output:

Attention outputs of shape `[batch_size, Tq, dim]`. [Optional] Attention scores after masking and softmax with shape `[batch_size, Tq, Tv]`.

The meaning of `query`, `value` and `key` depend on the application. In the case of text similarity, for example, `query` is the sequence embeddings of the first piece of text and `value` is the sequence embeddings of the second piece of text. `key` is usually the same tensor as `value`.

Here is a code example for using `AdditiveAttention` in a CNN+Attention network:

```
# Variable-length int sequences.
query_input = tf.keras.Input(shape=(None,), dtype='int32')
value_input = tf.keras.Input(shape=(None,), dtype='int32')

# Embedding lookup.
token_embedding = tf.keras.layers.Embedding(max_tokens, dimension)
# Query embeddings of shape [batch_size, Tq, dimension].
query_embeddings = token_embedding(query_input)
# Value embeddings of shape [batch_size, Tv, dimension].
value_embeddings = token_embedding(value_input)

# CNN layer.
cnn_layer = tf.keras.layers.Conv1D(
    filters=100,
    kernel_size=4,
    # Use 'same' padding so outputs have the same shape as inputs.
    padding='same')
# Query encoding of shape [batch_size, Tq, filters].
query_seq_encoding = cnn_layer(query_embeddings)
# Value encoding of shape [batch_size, Tv, filters].
value_seq_encoding = cnn_layer(value_embeddings)

# Query-value attention of shape [batch_size, Tq, filters].
query_value_attention_seq = tf.keras.layers.AdditiveAttention()(
    [query_seq_encoding, value_seq_encoding])

# Reduce over the sequence axis to produce encodings of shape
# [batch_size, filters].
query_encoding = tf.keras.layers.GlobalAveragePooling1D()(
    query_seq_encoding)
query_value_attention = tf.keras.layers.GlobalAveragePooling1D()(
    query_value_attention_seq)

# Concatenate query and document encodings to produce a DNN input layer.
input_layer = tf.keras.layers.Concatenate()(
    [query_encoding, query_value_attention])

# Add DNN layers, and create Model.
# ...
```