C4W2_Attention

August 19, 2025

1 The Three Ways of Attention and Dot Product Attention: Ungraded Lab Notebook

In this notebook you'll explore the three ways of attention (encoder-decoder attention, causal attention, and bi-directional self attention) and how to implement the latter two with dot product attention.

1.1 Background

As you learned last week, **attention models** constitute powerful tools in the NLP practitioner's toolkit. Like LSTMs, they learn which words are most important to phrases, sentences, paragraphs, and so on. Moreover, they mitigate the vanishing gradient problem even better than LSTMs. You've already seen how to combine attention with LSTMs to build **encoder-decoder models** for applications such as machine translation.

This week, you'll see how to integrate attention into **transformers**. Because transformers do not process one token at a time, they are much easier to parallelize and accelerate. Beyond text summarization, applications of transformers include: * Machine translation * Auto-completion * Named Entity Recognition * Chatbots * Question-Answering * And more!

Along with embedding, positional encoding, dense layers, and residual connections, attention is a crucial component of transformers. At the heart of any attention scheme used in a transformer is **dot product attention**, of which the figures below display a simplified picture:

With basic dot product attention, you capture the interactions between every word (embedding) in your query and every word in your key. If the queries and keys belong to the same sentences, this constitutes **bi-directional self-attention**. In some situations, however, it's more appropriate to consider only words which have come before the current one. Such cases, particularly when the queries and keys come from the same sentences, fall into the category of **causal attention**.

For causal attention, you add a **mask** to the argument of our softmax function, as illustrated below:

Now let's see how to implement the attention mechanism.

1.2 Imports

[1]:
```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

import sys
```

```
import tensorflow as tf

import textwrap
wrapper = textwrap.TextWrapper(width=70)
```

Here is a helper function that will help you display useful information:

- `display_tensor()` prints out the shape and the actual tensor.

[2]:
```
def display_tensor(t, name):
    """Display shape and tensor"""
    print(f'{name} shape: {t.shape}\n')
    print(f'{t}\n')
```

Create some tensors and display their shapes. Feel free to experiment with your own tensors. Keep in mind, though, that the query, key, and value arrays must all have the same embedding dimensions (number of columns), and the mask array must have the same shape as `tf.matmul(query, key_transposed)`.

[3]:
```
q = tf.constant([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
display_tensor(q, 'query')
k = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
display_tensor(k, 'key')
v = tf.constant([[0.0, 1.0, 0.0], [1.0, 0.0, 1.0]])
display_tensor(v, 'value')
m = tf.constant([[1.0, 0.0], [1.0, 1.0]])
display_tensor(m, 'mask')
```

```
query shape: (2, 3)

[[1. 0. 0.]
 [0. 1. 0.]]

key shape: (2, 3)

[[1. 2. 3.]
 [4. 5. 6.]]

value shape: (2, 3)

[[0. 1. 0.]
 [1. 0. 1.]]

mask shape: (2, 2)

[[1. 0.]
 [1. 1.]]
```

1.3 Dot product attention

Here you compute softmax $\left(\frac{QK^T}{\sqrt{d}} + M\right)V$, where the (optional, but default) scaling factor $\sqrt{d}$ is the square root of the embedding dimension.

[4]:
```
def dot_product_attention(q, k, v, mask, scale=True):
    """
    Calculate the attention weights.
      q, k, v must have matching leading dimensions.
      k, v must have matching penultimate dimension, i.e.: seq_len_k =␣
 ↪seq_len_v.
      The mask has different shapes depending on its type(padding or look␣
 ↪ahead)
      but it must be broadcastable for addition.

    Arguments:
        q (tf.Tensor): query of shape (..., seq_len_q, depth)
        k (tf.Tensor): key of shape (..., seq_len_k, depth)
        v (tf.Tensor): value of shape (..., seq_len_v, depth_v)
        mask (tf.Tensor): mask with shape broadcastable
              to (..., seq_len_q, seq_len_k). Defaults to None.
        scale (boolean): if True, the result is a scaled dot-product attention.␣
 ↪Defaults to True.

    Returns:
        attention_output (tf.Tensor): the result of the attention function
    """

    # Multiply q and k transposed.
    matmul_qk = tf.matmul(q, k, transpose_b=True)  # (..., seq_len_q, seq_len_k)

    # scale matmul_qk with the square root of dk
    if scale:
        dk = tf.cast(tf.shape(k)[-1], tf.float32)
        matmul_qk = matmul_qk / tf.math.sqrt(dk)
    # add the mask to the scaled tensor.
    if mask is not None:
        matmul_qk = matmul_qk + (1. - mask) * -1e9

    # softmax is normalized on the last axis (seq_len_k) so that the scores add␣
 ↪up to 1.
    attention_weights = tf.keras.activations.softmax(matmul_qk)

    # Multiply the attention weights by v
    attention_output = tf.matmul(attention_weights, v)  # (..., seq_len_q,␣
 ↪depth_v)
```

```
    return attention_output
```

Finally, you implement the *masked* dot product self-attention (at the heart of causal attention) as a special case of dot product attention

[5]:
```
def causal_dot_product_attention(q, k, v, scale=True):
    """ Masked dot product self attention.
    Args:
        q (numpy.ndarray): queries.
        k (numpy.ndarray): keys.
        v (numpy.ndarray): values.
    Returns:
        numpy.ndarray: masked dot product self attention tensor.
    """

    # Size of the penultimate dimension of the query
    mask_size = q.shape[-2]

    # Creates a matrix with ones below the diagonal and 0s above. It should␣
 ↪have shape (1, mask_size, mask_size)
    mask = tf.experimental.numpy.tril(tf.ones((mask_size, mask_size)))

    return dot_product_attention(q, k, v, mask, scale=scale)
```

[6]:
```
result = causal_dot_product_attention(q, k, v)
display_tensor(result, 'result')
```

```
result shape: (2, 3)

[[0.         1.         0.        ]
 [0.8496746  0.15032543 0.8496746 ]]
```