C4W2_Masking

January 21, 2024

1 Masking

In this lab, you will implement the masking, that is one of the essential building blocks of the transformer. You will see how to define the masks and test how they work. You will use the masks later in the programming assignment.

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

1.1 1 - Masking

There are two types of masks that are useful when building your Transformer network: the *padding mask* and the *look-ahead mask*. Both help the softmax computation give the appropriate weights to the words in your input sentence.

1.1.1 1.1 - Padding Mask

Oftentimes your input sequence will exceed the maximum length of a sequence your network can process. Let's say the maximum length of your model is five, it is fed the following sequences:

```
[["Do", "you", "know", "when", "Jane", "is", "going", "to", "visit", "Africa"],
 ["Jane", "visits", "Africa", "in", "September" ],
 ["Exciting", "!"]
]
```

which might get vectorized as:

```
[[ 71, 121, 4, 56, 99, 2344, 345, 1284, 15],
 [ 56, 1285, 15, 181, 545],
 [ 87, 600]
]
```

When passing sequences into a transformer model, it is important that they are of uniform length. You can achieve this by padding the sequence with zeros, and truncating sentences that exceed the maximum length of your model:

```
[[ 71, 121, 4, 56, 99],
 [ 2344, 345, 1284, 15, 0],
 [ 56, 1285, 15, 181, 545],
 [ 87, 600, 0, 0, 0],
```

]

Sequences longer than the maximum length of five will be truncated, and zeros will be added to the truncated sequence to achieve uniform length. Similarly, for sequences shorter than the maximum length, zeros will also be added for padding.

When pasing these vectors through the attention layers, the zeros will typically disappear (you will get completely new vectors given the mathematical operations that happen in the attention block). However, you still want the network to attend only to the first few numbers in that vector (given by the sentence length) and this is when a padding mask comes in handy. You will need to define a boolean mask that specifies to which elements you must attend (1) and which elements you must ignore (0) and you do this by looking at all the zeros in the sequence. Then you use the mask to set the values of the vectors (corresponding to the zeros in the initial vector) close to negative infinity (-1e9).

Imagine your input vector is `[87, 600, 0, 0, 0]`. This would give you a mask of `[1, 1, 0, 0, 0]`. When your vector passes through the attention mechanism, you get another (randomly looking) vector, let's say `[1, 2, 3, 4, 5]`, which after masking becomes `[1, 2, -1e9, -1e9, -1e9]`, so that when you take the softmax, the last three elements (where there were zeros in the input) don't affect the score.

The MultiheadAttention layer implemented in Keras, uses this masking logic.

Note: The below function only creates the mask of an *already padded sequence.*

[2]:
```
def create_padding_mask(decoder_token_ids):
    """
    Creates a matrix mask for the padding cells

    Arguments:
        decoder_token_ids (matrix like): matrix of size (n, m)

    Returns:
        mask (tf.Tensor): binary tensor of size (n, 1, m)
    """
    seq = 1 - tf.cast(tf.math.equal(decoder_token_ids, 0), tf.float32)

    # add extra dimensions to add the padding
    # to the attention logits.
    # this will allow for broadcasting later when comparing sequences
    return seq[:, tf.newaxis, :]
```

[3]:
```
x = tf.constant([[7., 6., 0., 0., 0.], [1., 2., 3., 0., 0.], [3., 0., 0., 0., 0.
 ↪]])
print(create_padding_mask(x))
```

```
tf.Tensor(
[[[1. 1. 0. 0. 0.]]

 [[1. 1. 1. 0. 0.]]
 [[1. 0. 0. 0. 0.]]], shape=(3, 1, 5), dtype=float32)
```

If you multiply (1 - mask) by -1e9 and add it to the sample input sequences, the zeros are essentially set to negative infinity. Notice the difference when taking the softmax of the original sequence and the masked sequence:

[4]:
```
# Create the mask for x
mask = create_padding_mask(x)

# Extend the dimension of x to match the dimension of the mask
x_extended = x[:, tf.newaxis, :]

print("Softmax of non-masked vectors:\n")
print(tf.keras.activations.softmax(x_extended))

print("\nSoftmax of masked vectors:\n")
print(tf.keras.activations.softmax(x_extended + (1 - mask) * -1.0e9))
```

```
Softmax of non-masked vectors:

tf.Tensor(
[[[7.2959954e-01 2.6840466e-01 6.6530867e-04 6.6530867e-04 6.6530867e-04]]

 [[8.4437378e-02 2.2952460e-01 6.2391251e-01 3.1062774e-02 3.1062774e-02]]

 [[8.3392531e-01 4.1518696e-02 4.1518696e-02 4.1518696e-02 4.1518696e-02]]],
shape=(3, 1, 5), dtype=float32)

Softmax of masked vectors:

tf.Tensor(
[[[0.7310586  0.26894143 0.         0.         0.        ]]

 [[0.09003057 0.24472848 0.66524094 0.         0.        ]]

 [[1.         0.         0.         0.         0.        ]]], shape=(3, 1, 5),
dtype=float32)
```

1.1.2 1.2 - Look-ahead Mask

The look-ahead mask follows similar intuition. In training, you will have access to the complete correct output of your training example. The look-ahead mask helps your model pretend that it correctly predicted a part of the output and see if, *without looking ahead*, it can correctly predict the next output.

For example, if the expected correct output is `[1, 2, 3]` and you wanted to see if given that the model correctly predicted the first value it could predict the second value, you would mask out the second and third values. So you would input the masked sequence `[1, -1e9, -1e9]` and see if it could generate `[1, 2, -1e9]`.

Just because you've worked so hard, we'll also implement this mask for you . Again, take a close look at the code so you can effectively implement it later.

[5]:
```
def create_look_ahead_mask(sequence_length):
    """
    Returns a lower triangular matrix filled with ones

    Arguments:
        sequence_length (int): matrix size

    Returns:
        mask (tf.Tensor): binary tensor of size (sequence_length,␣
 ↪sequence_length)
    """
    mask = tf.linalg.band_part(tf.ones((1, sequence_length, sequence_length)),␣
 ↪-1, 0)
    return mask
```

[6]:
```
x = tf.random.uniform((1, 3))
temp = create_look_ahead_mask(x.shape[1])
temp
```

[6]:
```
<tf.Tensor: shape=(1, 3, 3), dtype=float32, numpy=
array([[[1., 0., 0.],
        [1., 1., 0.],
        [1., 1., 1.]]], dtype=float32)>
```

Congratulations on finishing this Lab! Now you should have a better understanding of the masking in the transformer and this will surely help you with this week's assignment!

Keep it up!