Deep Learning for Text-p2

2021.7.16

11.4 The Transformer architecture

Neural attention

11.4.1 Understanding self-attention

Not all input information seen by a model is equally important to the task at hand, so models should "pay more attention" to some features and "pay less attention" to other features.

We've already encountered a similiar concept twice in this book:

Max pooling

TF-IDF normalization: assign scores to tokens based on how much information information different tokens are likely to carry.

They all started by computing important scores for a set of features, with higher scores for more relevant features and lower scores for less relevant features. How these scores should be computed, and what you should do with them, will vary from approach to approach.

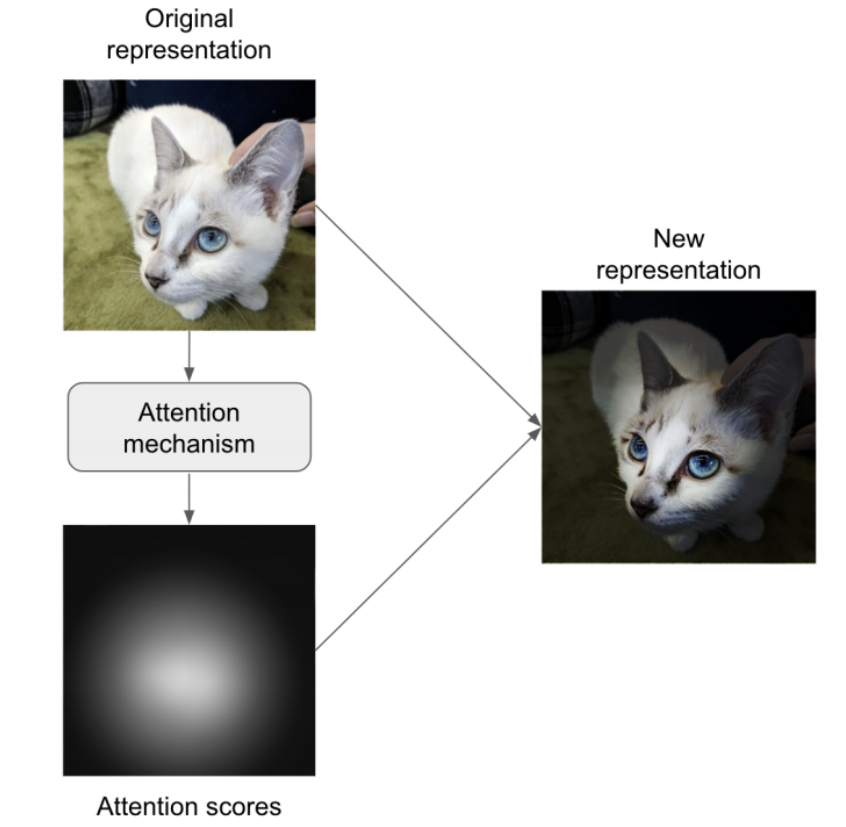

Figure 11.5 The general concept of "attention" in deep learning: input features get assigned "attention scores", which can be used to inform the next representation of the input.

Crucailly, this kind of attention mechanism can be used for more than just highlighting or erasing certain features. It can be used to make features *context-aware*. We've just learned about word embedding-vector spaces that capture the "shape" of the semantic relationships between different words. In an embedding space, a single word has a fixed position-a fixed set of relationships with every other word in the space. But that's not quite how language works: the meaning of a word is usually context-specific.

Clearly, a smart embedding space would provide a different vector representation for a word depending on the other words surrounding it. That's where *self-attention* comes in. The purpose of self-attention is to modulate the representation of a token by using the representation of related tokens in the sequence. This produces context-aware token representations.

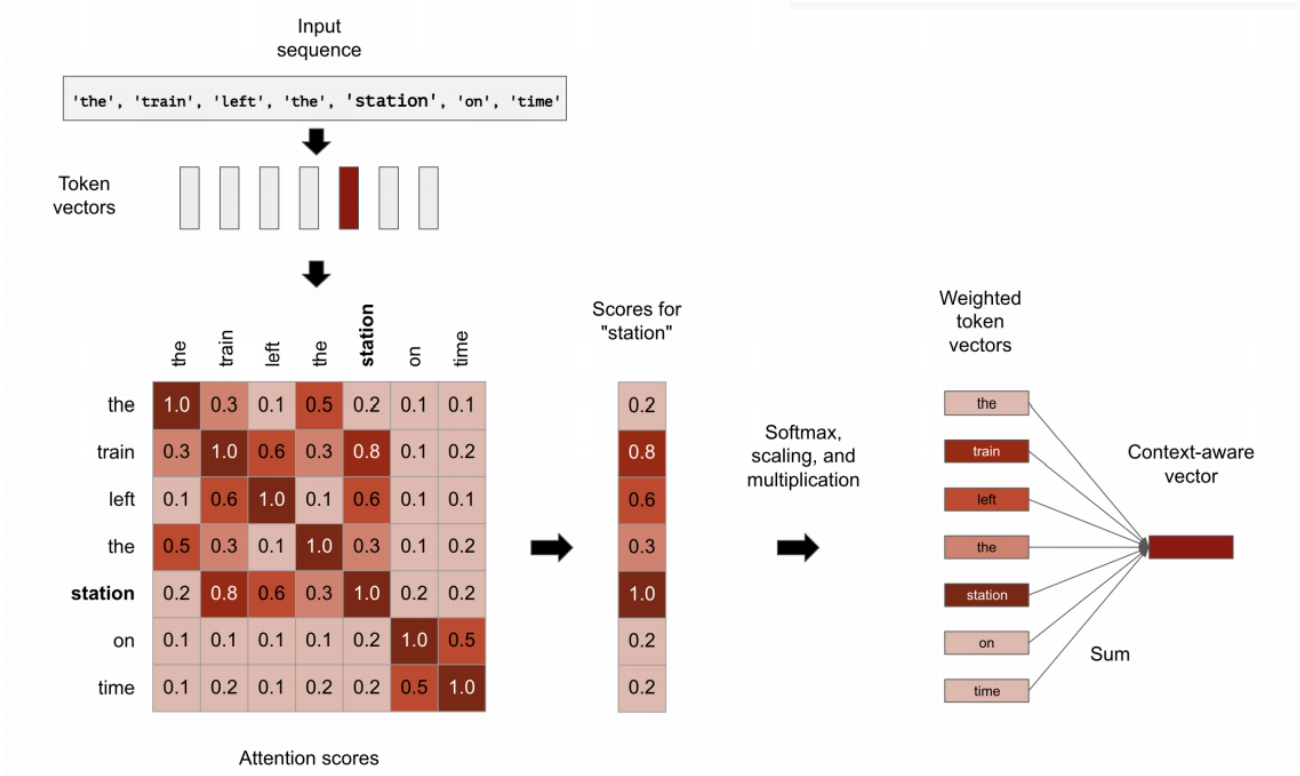

Figure 11.6 Self-attention: attention scores are computed between "station" and every other word in the sequence, which are then used to weight a sum of word vectors that becomes the new "station" vector

step1: compute relevancy scores between the vector for "station" and every other word in the sentences. These are our "attention scores". We're simply going to use the dot product between two word vectors as a measure of the strength of their relationship. It's a very computationally-efficient distance function, and it was already the standard way to relate two word embeddings to each other long before Transformers. In practice, these scores will also go through a scaling function and a softmax, but for now, that's just an implementation detail.

step2: compute the sum of all word vectors in the sentence, weighted by our relevancy scores. Words closely related to "station" will contribute more to the sum (including the word "station" itself), while irrelevant words will contribute almost nothing. The resulting vector is our new representation for "station": a representation that incorporates the surrounding context. In particular, it includes part of the "train" vector, clarifying that it is, in fact, a "train station".

We'd repeat this process for every word in the sentence, producing a new sequences of vectors encoding the sentence. Let's see its pseudo-code:

```
def self_attention(input_sequence):
    output = np.zeros(shape=input_sequence.shape)
    # iterate over each token in the input sequence
    for i, pivot_vector in enumerate(input_sequence):
        scores = np.zeros(shape=(len(input_sequence),))
        for j, vector in enumerate(input_sequence):
            # compute the dot product between the token and every other token
            scores[j] = np.dot(pivot_vector, vector.T)
        # scale by a normalization and apply a softmax
        scores /= np.sqrt(input_sequence.shape[1])
        scores = softmax(scores)
        new_pivot_representation = np.zeros(shape=pivot_vector.shape)
        for j, vector in enumerate(input_sequence):
            # take the sum of all tokens weighted by the attention scores
            new_pivot_representation += vector * scores[j]
```

```
        # that sum is our output
        output[i] = new_pivot_representation
    return output
```

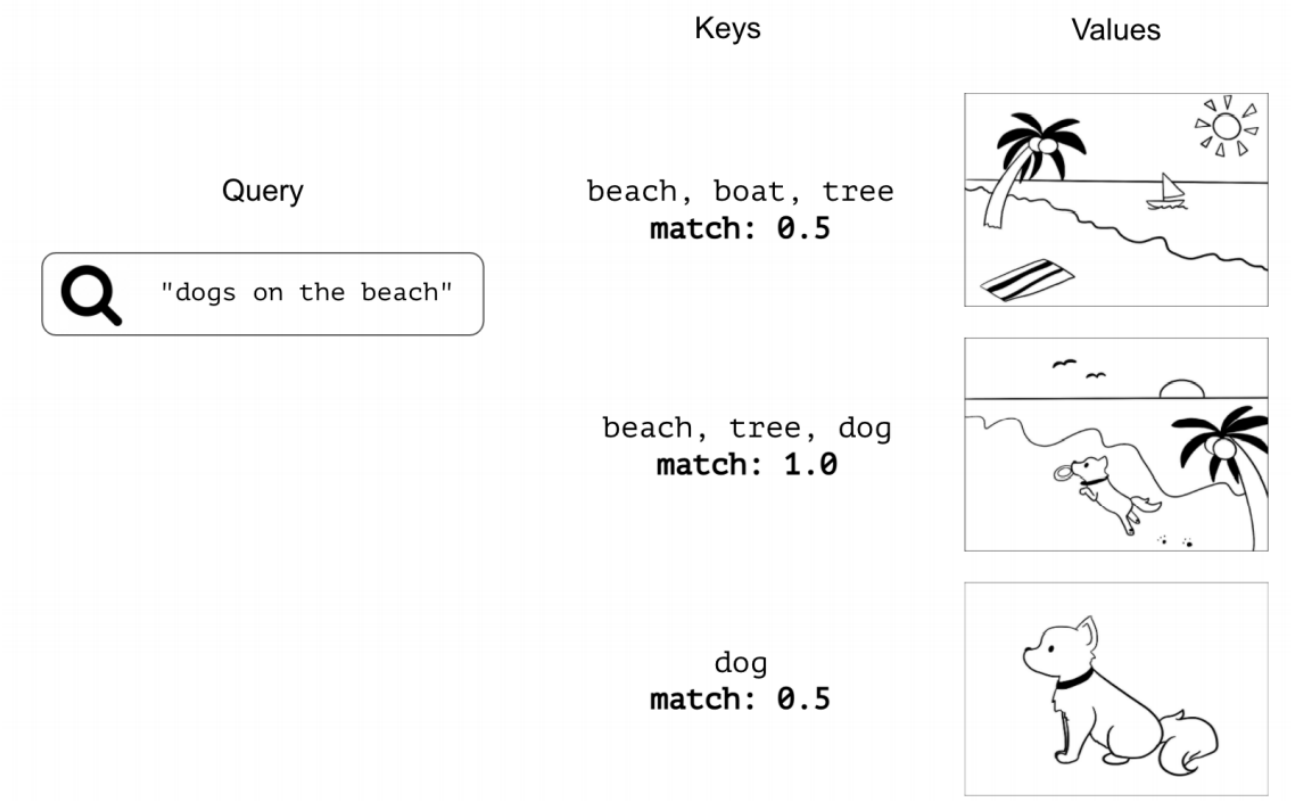

Figure 11.7 Retrieving images from a database: the "query" is compared to a set of "keys", and the match scores are used to rank "values" (images)

11.4.2 Multi-head attention

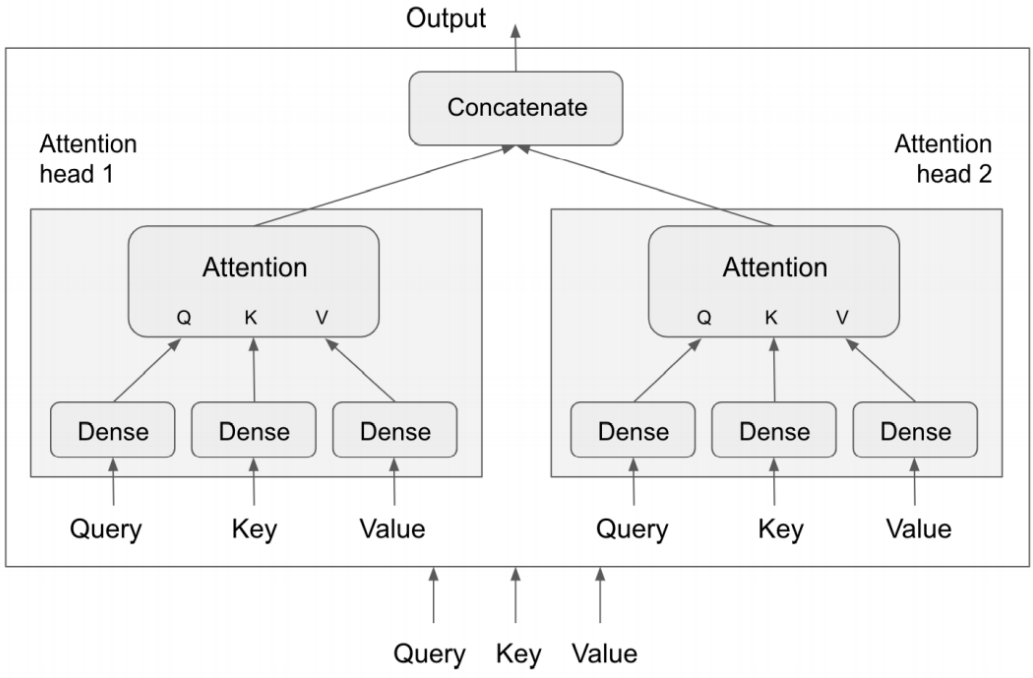

Figure 11.8 The `MultiHeadAttention` layer

11.4.3 The Transformer encoder

Let's go back to code part