Attention Notes

Piyush Patil

September 19, 2017

The idea behind attention is that given a large input, rather than having our model process the entire input at once, which might be both computationally expensive and might slow down learning as the model is forced to wade through a much higher-dimensional input space, we allow our model to focus on different, small, sub-parts of the input and process each sub-part separately, combining results at the end. This is inspired in part by the visual attention system found in the human occipital lobe, wherein people focus on certain small sub-parts of their visual field with high resolution and ignore the rest.

Thus, attention mechanisms were first introduced in the context of image recognition, but have since been extended to other domains of machine learning such as NLP and even general supervised learning. Another area where attention mechanisms have found promise is in memory-augmented models; it's become increasingly common to enhance neural models with large memory stores they can differentiably read and write to, to intuitively enable them to avoid storing "data", as opposed to processing information, in their "working memory" (i.e. within the network weights themselves). In order to differentiably read and write to a memory matrix, we can introduce an attention mechanism wherein rather than selecting a single index of the memory matrix to read or write to, which would correspond to the undifferentiable step function, we read and write to the entire memory matrix, by taking a weighted sum over all the entries. If we only require data stored in a single or small number of indexes, we simply weight those entries much higher than the rest, allowing us to "focus our attention" on the entries we want without compromising smoothness of the operation.

Generally speaking, an attention function over a memory store can be described as a mapping from a given query and a given set of key-value pairs to an output. Specifically, the output is a weighted sum of values in the key-value store, where the weights are computed using some compatibility or similarity function that acts on the query and the corresponding key. The idea here is that we're given a set of keys and the values they map to, and want to estimate the corresponding value for a new, unseen key: the query. We do this by taking what we already know about the key-value mapping, i.e. by looking at the existing key-value pairs and summing over them, and weighting each term in the sum by how similar it is to the query, based on the assumption that similar keys should map to similar values (i.e. that the key-value mapping is continuous).