CS 282 Deep Neural Networks **Fall 2022**

Lecture 14: Attention/self-supervision

Lecturer: Anant Sahai *Scribe: Xin Chen, Yun Yeong Choi*

14.1 Attention

We have learned that the natural architecture design of deep learning models is useful for different tasks. For example, convolutional structure for computer vision tasks, sequential structure in RNN for language processing, forget gates in LSTM models, etc. Although the architecture of RNN model is useful for sequential data, it still has limitations in understanding human languages. One reason is that different human languages behave differently (e.g. different grammar, word orders, etc.) To handle this problem, we need to design some new architectures. First, let's look at the problem of machine translation as an example.

14.1.1 Motivation of Attention

When humans read a sentence, we read one word at a time. Therefore, we design the RNN encoder to do the same thing. As shown in Figure 14.1, each word is input into the encoder layers sequentially and each layer outputs a hidden state that is used as an input to the next layer. Each layer shares the same weights.

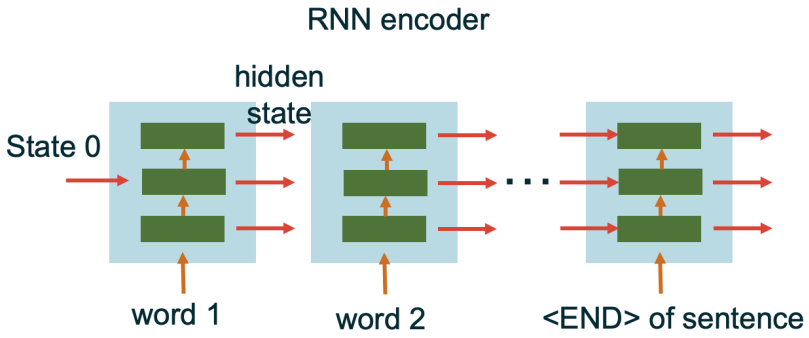

Figure 14.1: A RNN encoder used to process sequential data.

Aside: Why do we call the hidden output a state?

In a dynamic system, we define *"state"* as something that summarizes all the past that is relevant to the future. That is to say, if we know a state, we don't need to care about its past and that state will provide us with the information that we can use for the future. In the sentence example, the hidden state in the last layer should summarize all the previous words.

Up to this point, the normal RNN architecture works well. However, we want the model to translate the sentence into another language. Thus we need a decoder network after the encoder. The decoder will also use the sequential architecture and output the words in other languages, as shown in figure 14.2. The output in each layer will be used as the input into next layer.

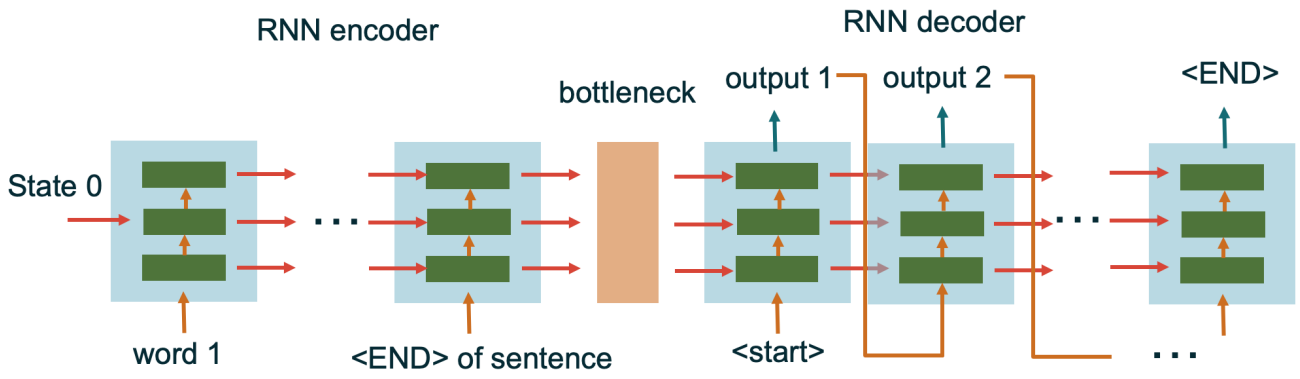

Figure 14.2: Encoder and Decoder in RNN.

Questions:

- Why can't we just map each word in the original sentence into the translated sentence?
  → Because different language has different grammatical structures. The first word in a sentence in language A may be the last word in language B.
- There are millions of words, do we need to have millions of output units for the last layer?
  → No. We can embed the word to appropriate high dimensional vectors.
- How do we translate output vectors to word?
  → Look for nearest neighbors of decoded vector.
- How do we propagate errors/loss?
  → 1) Just use output from decoder as a input. 2) Use local softmax for the rounding.
- How do we train the encoder and decoder?
  → Use supervised learning for decoder and self-supervision for encoder.

Note that there is a bottleneck between the encoder and decoder. We hope that this bottleneck can include everything about the input sentence (*recall "state"*). That is to say, we are squeezing all the previous information into this tiny bottleneck. This may cause a problem because there may be some details lost that are irrelevant to figuring out the structure of the output sentence but relevant to which specific word is in that sentence. For example, we input "My green cat is an alien". the workd "green" is used to decorate the "cat". In the output sentence in another language, we need to know which word ("green") is used to decorate the "cat". Then our model needs to look back into the input sentence. However, the architecture we just built cannot look back at things at different scales.

Recall that for image segmentation, the U-Net architecture concatenates features from the earlier layers to the upsampled features in the later layers to allow the model to look back into the encoder layers. Also,

images have a nice matrix structure for the model to process considering that we can clearly depict what the neighbors of pixels are. However, in the case of language, it is hard for the model to figure out the global idea of the neighborhood since it is not well defined in the language itself. **Therefore, we need to add something to our network, that allows the model to look back at the words that are originally embedded, and allows information to flow along the layers.**

Takeaway:

We want to add something in the language model, which allows the decoder layers to look back at what words are embedded. This is like a kind of memory where we are able to store something important in it and retrieve it later. We call this **"Attention."**

14.1.2 Attention structure: queries, keys and values

In the last section, we explained the motivation for "attention" and how it behaves similarly to "memory" in a computer. There are two methods of storing memory. One is like an "array," where we can store the value at an address and retrieve it later by looking into that address. The other method is like a "hash table," where we have a key that corresponds to a specific value and can retrieve the value by querying the key. Attention will use the hash table structure.

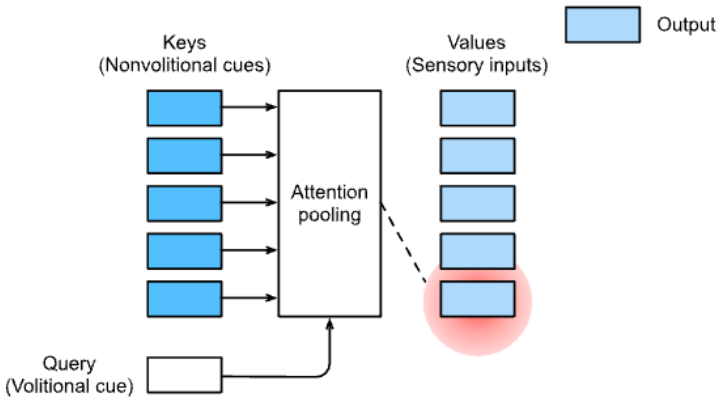

Figure 14.3: Scheme of attention. Query, Key, and Value [1].

Unlike the traditional hash table, now we can have a query vector that is not always exactly matched to keys in the hash table. So we need to design a "pooling" layer which not only can get an appropriate forward value, but also backpropagate gradients. Here are some ideas on how to design this hash table structure for attention.

1. Idea[-1]: A naive idea. Scan for an exact match to the key (like how a normal hash table functions). If found, return it.

   Problem: Really bad at initialization. When we initialize the model, we set a random key, and try to get the value with a random query. Then we are unable to get the value. Additionally, we are unable to calculate the gradient for backpropagation since even if we change the keys and queries by a small amount, there is still no matches, so there is no gradient.

2. Idea[0]: Scan for the closest match of query to key in the hash table, and return the value.

   Problem: Now we can retrieve a value and the gradient can reach the value. However, we still have the problem of calculating a gradient for the query and key. Note that we are using the approximation of query and key. This means that changing the query or the key by a small amount will result in the same value, so the gradients will be 0 and cannot update the weights.

   Whenever we make a hard decision, there's a gradient problem. We need to soften this hard decision.

3. Idea[1]: Scan for the closest matches of query to the keys. Then return the weighted average of the values.

   Since the weights now depend on both the query and the key, we have gradients on query, key and values. In this perspective, attention can be thought of as "queryable pooling", because keys and query values are also learnable parameters, but there are no learnable weights in the attention mechanism itself. As shown in figure 14.4, the hidden states in the encoder are used as keys and values, and the hidden state from a decoder layer is used as a query. The attention layer will output the weighted value which is used together with the hidden state from the decoder to generate the output. This output is then fed into the next decoder layer.

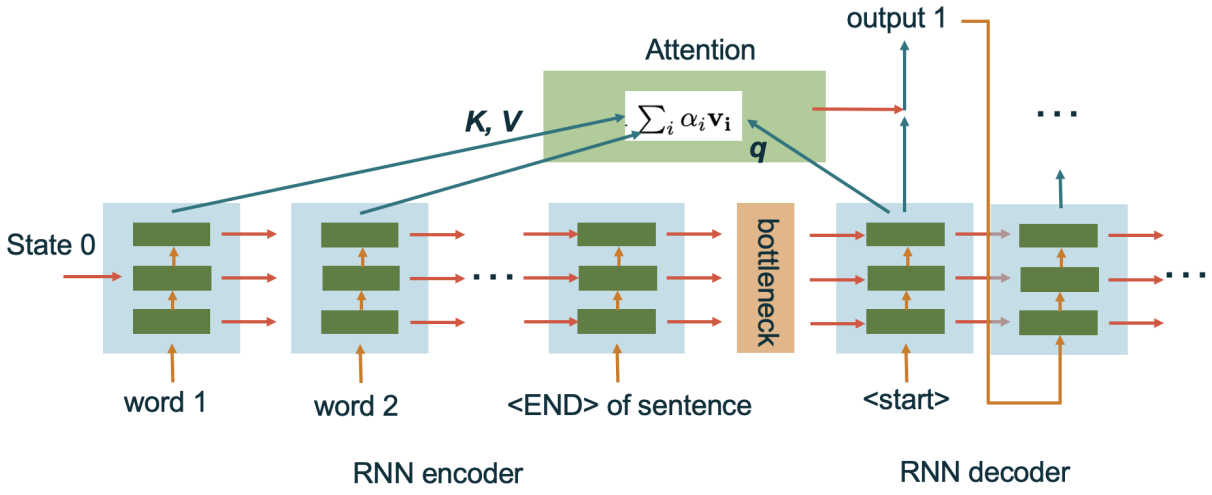

Figure 14.4: Scheme of attention in RNN layers.

Details:
With proper similarity function, weighted average will be

$$\textbf{Weighted Average} = \sum_{i=1}^{n} \text{Sim}(\textbf{query}, \textbf{key}_\textbf{i})\textbf{v}_\textbf{i} \tag{14.1}$$

where *Sim* is similarity function, **query** or **q** stands for query vector, $\textbf{key}_\textbf{i}$ or $\textbf{k}_\textbf{i}$ means ith key vector, and $\textbf{v}_\textbf{i}$ is a value corresponding to the ith value vector. Vectors/scalars are represented as bold/plain letters.

We have seen a similar ideas before - softmax and kernel-based methods. What should our similarity function be?

Inner product $\textbf{q}^\textbf{T}\textbf{k}$ or radial basis function (RBF) $\exp(-\gamma\|\textbf{q}-\textbf{k}\|^\textbf{2})$ are good choices. Note that the inner product should be normalized, otherwise if we have a long vector but low similarity, the inner product can likely still be large.

In "attention," we will use the normalized inner product:

$$e_i = \frac{\textbf{q}^\textbf{T}\textbf{k}_\textbf{i}}{\sqrt{d}} \tag{14.2}$$

where d is the dimensions of **q** and **k**. This scales the variance to 1 for random keys and queries, but still allows similarity to increase with $\mathcal{O}(\sqrt{d})$ when key and query are aligned.
After that, we use softmax to compute the weights:

$$\alpha_i = \frac{\exp(e_i)}{\sum_j \exp(e_j)} \tag{14.3}$$

Then "attention" will return $\sum_i \alpha_i \textbf{v}_\textbf{i}$ upon query.

Question:

- What can act as keys, values?
  → There are different components that can act as keys and values. For example, we can use a hidden state in the encoder as the key and another hidden state after that layer as the value. Also we can use the outputs in the decoder as the query.

14.1.3 Embed "order" information in Attention

Sometimes, we might need to take care about order information in the keys. For example, in natural language processing, "Prof. Sahai bites dog" and "Dog bites Prof. Sahai" makes the meaning totally different. One possible way to encode order/position/time is using complex vectors. If we let t be the position in sentence we want to encode, this can be differentiated from other sentence positions using complex expression.

$$e^{j\omega t} = \cos\omega t + j\sin\omega t \text{ or } \begin{bmatrix} \cos\omega_1 t \\ \sin\omega_1 t \\ \cos\omega_2 t \\ \vdots \\ \sin\omega_n t \end{bmatrix} \tag{14.4}$$

Note that we don't want to use the complex number in Neural Networks, so in practice, vector expression will be used. This positional encoding is useful since not only we can express relative shift using matrix multiplication ($e^{j\omega(t+\phi)} = e^{j\omega t} \times e^{j\omega\phi}$) but also we can differentiate vectors as on the unit circle in the complex plane. By concatenating the positional encoding to the key vector, we have both a regular key part and a positional encoding. Now we can query one or a combination of them.

Questions:

- Why do we need periodicity of time? We don't have that in our sentence.
  → The periodicity allows us to have an easier distinction between coarse and fine degrees of time. Additionally, we do not want to make our activations really big, and this periodicity confines the activations to a compact space.
- Is ω is learnable?
  → We can set it as learnable parameter. As far as we know, it has not proven beneficial to have learnable ω. People usually fix ω.

14.2 Self-supervision

In many contexts, we lack enough labeled data for supervising learning, so we may need to turn to unsupervised learning. There are two kinds of unsupervised learning: (1) Dimensionality reduction style (pre-regression) (2) Clustering style (classification).

Note that in these styles of unsupervised learning, there are no gradients or loss functions. But what if we can understand (1) and (2) in terms of loss function and gradient? That is to say, can we design these unsupervised learning techniques in terms of loss function, and then do gradient descent to update weights, which transforms this unsupervised learning problem into a "supervised learning", or **"self-supervision"** problem. Next lecture, we will see how we can turn the dimensionality reduction style of unsupervised learning into two different kinds of self-supervised learning problems – one we call the "autoencode style," and the other we call the "masked reconstruction style.

14.3 What we wish this lecture also had to make things clearer?

1. When talking about query, key and values, please use more clear examples to explain this concepts.
2. It would be better if the professor can give examples of word embedding.
3. We wish the professor can spend more time clarifying how position/time information is included in Attention. It would be better if the professor could present a trivial example.
4. It would be better if the professor talk more about the details of how the encoder/decoder is trained. For example, the professor mentioned we can just train the decoder alone without the encoder but we are not sure how. Currently it is very hand-waving.

References

[1] Aston Zhang, Zachary C. Lipton, Mu Li, Alexander J. Smola, Dive into Deep Learn-

ing, *arXiv preprint arXiv:2106.11342* (2021).