CS 282 Deep Neural Networks **Spring 2023**

Lecture 15: From Attention to Auto-encoding

Lecturer: Anant Sahai *Scribe: Yiheng Li, Karim Elmaaroufi*

1 . Attention

1.1 Recap: Attention Mechanism

The attention mechanism is just like some "memory", or to say like a queriable "pooling" operation. It's somewhere between a max pool and average pool of what's stored. It basically consists of two steps:

1. We "store" the pairs $\vec{k}$ and $\vec{v}$ as vectors $< \vec{k_i}, \vec{v_i} >$. They're an analog to *key*, *value* pairs that you'd see in a hash table.
2. Then we *query* them with $\vec{q}$

The specific look up operation using the query can be summarized as three steps:

1. Initial Matching:
$$e_i = \frac{< \vec{q}, \vec{k_i} >}{\sqrt{d}} \tag{15.1}$$
where $\vec{q}, \vec{k_i} \subseteq \mathbb{R}^d, d \subseteq \mathbb{R}$ and $< \vec{q}, \vec{k_i} > = \vec{q}^T \vec{k_i}$. Observe that the keys and querys should have the same dimensions for this operation to work.
2. Compute weights via softmax:
$$\alpha_i = \frac{e^{e_i}}{\sum_j e^{e_j}} \tag{15.2}$$
This produces weights in the form of probabilities, where the sum of all weights is 1 and all weights are non-negative. In this way, the entry corresponding to the highest match would dominate.
3. Return the combined value
$$\sum_j \alpha_j \vec{v_j} \tag{15.3}$$
Note that there are no dimension restrictions on $\vec{v_i}$

Note that there are no learnable parameters are included in this structure.

Another question on this is why we use $\sqrt{d}$ for the normalization. Let's compare some choices. If we do nothing on normalization, the product would increase as the dimension increases. Then e_i would increase, and the domination of the largest α_i would be more significant. In this way all gradients would go to the largest term. Another choice is to use d for normalization. In this way, the Central Limit Theorem tells us that the dot-product in step 1 would behave like a Gaussian where it's standard deviation will be around the square root of the number of terms to be summed. Thus, once we divide by d, the e_i would have an average close to 0 and a standard deviation of n. The values would go too close to 0, and the whole process would be too similar to average pooling. Therefore, there wouldn't be a large difference between the terms

in the gradients when we route them back. Considering problems in these two options, we choose something between them, $\sqrt{d}$, which is also the standard deviation of $< \vec{q}, \vec{k_i} >$.

Finally, a question here is where can we find the $\vec{q}$, $\vec{k}$ and $\vec{v}$? After all we are storing these keys and values but where are we getting them from? The answer is they are all outputs of some Multi-Layer Perceptron (MLP). They are learnable parameters which would be obtained from training.

1.2 Recap: Encoder / Decoder Architecture for Translation

Figure 15.1: Encoder and decoder architecture.

Figure **??** shows an encoder-decoder structure with internal RNNs. We initialize the hidden size as 0 and input a sequence to the RNN encoder (<sos> = Start of Sequence). An embedding of the hidden information is learned by the encoder, and then is put into the decoder. We also input the start token together with the true label of $y_1, ..., y_n$ to the RNN decoder. At the output side of the RNN decoder, we expect to get the decoded information $\hat{y_1}, ..., \hat{y_n}$ and the end token in the last box. We can then compute the loss by comparing y to $\hat{y}$

1.3 Attention

Figure **??** shows where and how we introduce the attention mechanism. The attention module is inserted between two RNN layers in the decoder. The query $\vec{q}$ is from the output of last RNN layer, while the key and the value $\vec{k}$ and $\vec{v}$ may from the hidden states of the encoder or the decoder itself.

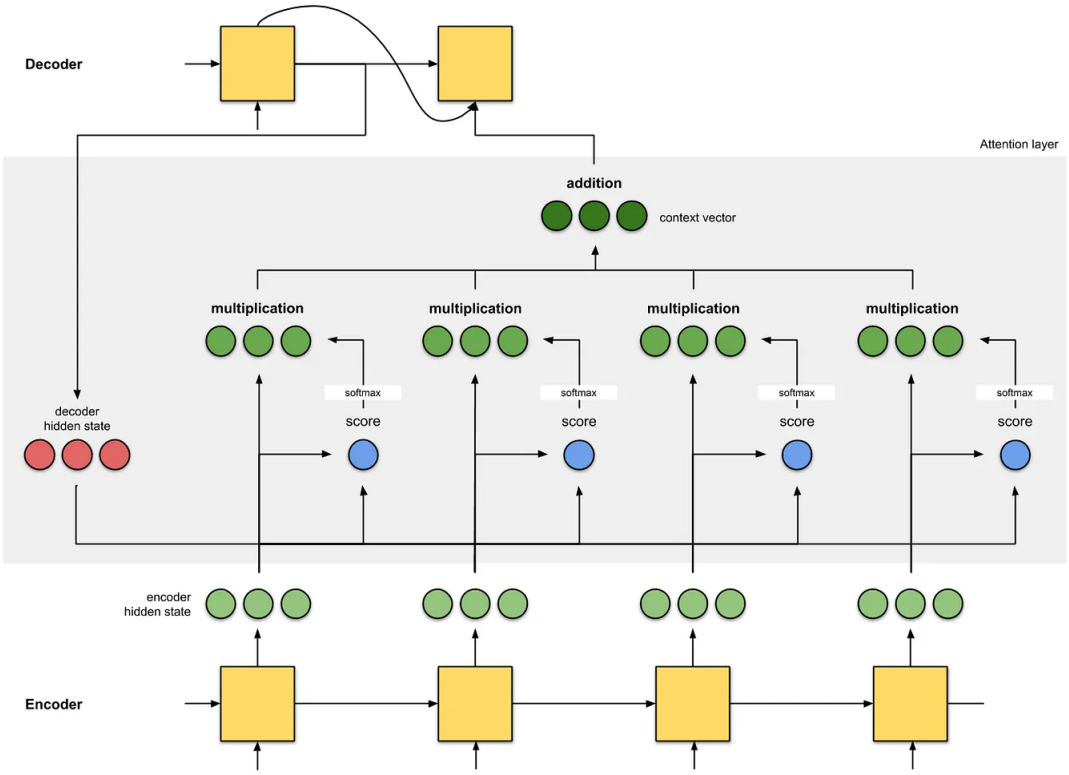

Figure 15.2: One attention layer

If keys and values are obtained from the encoder, we call the attention layer Cross Attention. The key and the value can use the same information as the input (like layer output / input etc.), but be generated through different attention layers, so that they can focus on different parts of the info. If the key and the value are from the decoder, we call it Self Attention. The benefit of this includes creating more routes to prevent the gradient from dying. Note that in this way the key and the value must be causal, which means they can only depend on the previous or current timestep. If time is current, only layers below can be seen as the input. In a word, only the input side of the decoder can be used here.

1.4 Info routes in Cross Attention

In Cross Attention, there are two ways for the gradients to go backwards: One through the hidden states through the RNN encoder, and the other through the attention mechanism and the key / value to the encoder. The hidden-state route can keep track of the time info (e.g. what's the last word input?), while the attention is normally unordered across input subscript i. However, we can also add time to the attention input directly, which is called positional encoding.

1.5 Lots of Attention

At this point, you may step back and begin to wonder what have we done? We've built memory with attention that lets us look back at any state in the encoder and decoder. So, what's the purpose of the RNN? Well the one thing that remains is that it's keeping a sense of time. It seems a bit expensive to keep many neural networks around just to keep track of time so let's see how we can encode time as positional encoding.

2. Positional Encoding

The problem of positional encoding is how to add time to the input vector? We come up with different ideas to do this:

- **Idea** -1 Just increment a counter. For example, we mark the first place as 1, the second as 2 etc. However, in this way, the number would be huge when it comes to later places, like 1000 for the 1000th place. This would cause an unbalanced encoding as the later places would be shining in the model.
- **Idea** 0 Normalized counter. For example we can set the largest step as 1 and the smallest as 0. However, in this way, if we also have a sequence of 1000, the difference between each step would be very tiny: just 0.001.
- **Idea** $1a$ Use "circular" time. In this way, we utilize the equation:

$$e^{jwt} = cos(wt) + isin(wt) \tag{15.4}$$

  In this way, the norm of the positional encoding is always 1. However, the difference between two steps would still be very tiny when w goes down, just like the time difference between two moments of a clock which rotates a cycle in 12 centuries. The only way to use this is not to depend too much on the difference. There is a question on whether we can use difference sequence for different inputs. However, this would make it hard to calculate the previous moment from the current one, as the difference is not known. Therefore, a single w is preferred.
- **Idea** $1b$ Use an ordered list to store the key and the value instead of using unordered dictionary.

3. Self-supervision

As we eluded to earlier, when we have an RNN and Attention together, a problem arrises: what is the smallest acceptable RNN size? After all the RNN is only keeping time for us at this point since our memory is accessed through attention. Prior work reduces the RNN size to 0, which let the network becomes a pure MLP, and it works very well. See the paper Attention Is All You Need

Since attention is in fact a giant pooling process, it doesn't have a strong inductive bias compared to fully-connected layers. Generally speaking, the less inductive bias introduced, the more data is necessary to train the model. However, in the real world, we frequently encounter a lack of labeled data, while many other times we do have a lot of unlabeled data. We hope to make use of these data, but they do not have labels, so they have no loss function and gradient defined. Therefore, we need self-supervision here, which is also called unsupervised learning.

Historically, unsupervised learning is also called data mining Chances are that if you previously took a Machine Learning course, you learned two methods to approach unsupervised learning: Principal Component Analysis (PCA) (and friends) for dimension reduction and K-Means (and friends) for clustering.

We focus more on the PCA here. Remember when doing PCA, we de-mean the normalized data, throw it into a large matrix and compute the Singular Value Decomposition (SVD), which is,

$$X = U\Sigma V^T \tag{15.5}$$

where $X \in R^{d\times n}$. Here d is the dimension of features and n is the number of data points. We then get the n largest eigenvalues $\sigma_1, \sigma_2, ..., \sigma_k$ and the corresponding principle features $\vec{v_1}, \vec{v_2}, ..., \vec{v_k} \in R^d$, which are a group of orthogonal vectors. In this way, we can use the combination of these principle features $\vec{v_i}^T\vec{x}$ to get the i^{th} principle feature of a new point $\vec{x}$. We choose the largest eigenvectors, as they give us the directions of largest variance which in turn tells us where the most interesting things about the data is happening. In this way, we reduce the dimensions and the data necessary to express the same features. However, the problem of no defined loss or gradients still exists as the SVD is a black box for us. So we still cannot use our standard machine learning approaches. Further, for large datasets, computing the SVD is not straight forward.

To solve this problem, notice that justified by the Eckart-Young-Mirsky theorem, the best rank-k approximation of a matrix X is

$$\hat{X} = \sum_{i=1}^{k} \sigma_i \vec{u_i}\vec{v_i}^T \tag{15.6}$$

The word "best" is in terms of

$$||X - \hat{X}||_F^2 \tag{15.7}$$

which is the square of the Frobenius norm of the difference between the original matrix and the approximation. Inspired by this, we can also choose the Frobenius norm as our loss function. Note that the word "best" can also be in terms of the 2-norm, but we choose the Frobenius norm as it's a simple norm to compute.

We can imagine a linear autoencoder:

$$X \to f(w, x) \tag{15.8}$$

where $X \subseteq R^{d\times n}$. d is the number of features and n is the number of data samples. In order to get the autoencoder to be of rank k, we can use:

$$\hat{X} = W_2W_1X \tag{15.9}$$

where $W_2 \subseteq R^{d\times k}$, $W_1 \subseteq R^{k\times d}$. In this way the rank of W_2W_1 is lower than k. Therefore, the problem comes to

$$argmin_{W_1,W_2}||X - W_2W_1X||_F^2 \tag{15.10}$$

Some may ask why we use this instead of directly doing SVD. This is because doing SVD could be computationally painful if there are many data points. However, in the autoencoder problem shown in Equation **??**, each X vector is independent, so Stochastic Gradient Descent (SGD) can be used here to calculate the minimum W_1, W_2, which is computationally much cheaper.

For the next time, we will talk more about the autoencoder, and how to generalize the idea (counter parts of W_2 and W_1 in other networks).

4. What we wish this lecture also have to make things clearer?

4.1 About the normalization of the attention

We thought that it might be good to address more on why we use $\sqrt{d}$ for the normalization. We put our ideas here, as these equations are not included in the lecture. If we assume

$$\vec{q}, \vec{k}_i \sim i.i.d. \sim N(0,1) \tag{15.11}$$

Then,

$$< \vec{q}, \vec{k}_i > \sim N(0,d) \tag{15.12}$$

Therefore, the standard deviation of $< \vec{q}, \vec{k}_i >$ should be $\sqrt{d}$, and that's the reason why we use $\sqrt{d}$ here for normalization