CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2022 Marvin Zhang **Discussion 7**

This discussion covers Attention Mechanisms and Transformers.

1 Attention Mechanisms

For many NLP and visual tasks we train our deep models on, features appear on the input text/visual data often contributes unevenly to the output task. For example, in a translation task, not the entirety of the input sentence will be useful (and may even be confusing) for the model to generate a certain output word, or not the entirety of the image contributes to a certain sentence generated in the caption.

While some RNN architectures we previously covered possess the capability to maintain a memory of the previous inputs/outputs, to compute output and to modify the memory accordingly, these memory states need to encompass information of many previous states, which can be difficult especially when performing tasks with long-term dependencies.

Attention mechanisms were developed to improve the network's capability of orienting perception onto parts of the data, and to allow random access to the memory of processing previous inputs. In the context of RNNs, attention mechanisms allow networks to not only utilize the current hidden state, but also the hidden states of the network computed in previous time steps as shown in Figure 5.

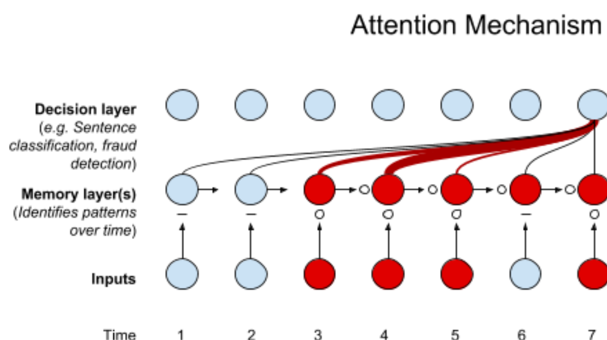

Figure 1: Attention Mechanism Illustrated

1.1 Luong Attention

The Luong attention is one of the commonly used attention mechanism in Neural Machine Translation, which is trained on the task of translating text from source to target language. At each time-step of decoding, the Luong Attention computes a set of alignment (attention) weights a_l based on alignment scores and use this to augment the computation of output probabilities in the translation task.

As shown in Figure 2, we have our encoding states (RNN hidden states when passed a source language into the model first) h_t. At decoding time, we have the original hidden states h_l initially computed with the previous output token and hidden states.

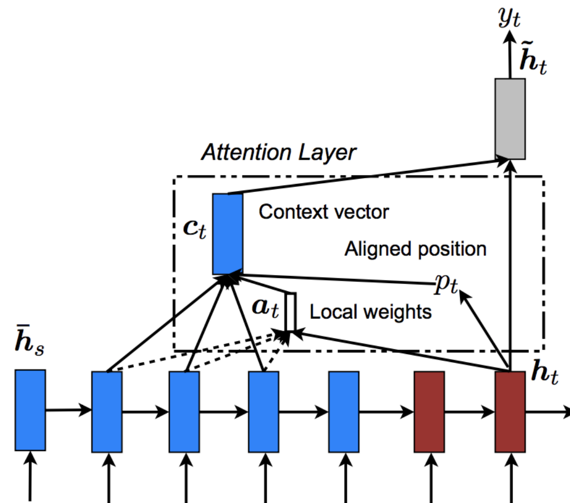

Figure 2: Luong Attention

With h_l and each h_t we compute an alignment score [1], then take a Softmax to obtain attention weights from the alignment scores.

With these attention weights, we compute a context vector c_t, which is a weighted sum of h_s. As we get c_t, we compute a transformation $\tilde{h}_t = \tanh\big(W_f[h_t; c_t]\big)$ applied on the concatenation of the context vector and the original hidden state. This is then used to compute the final output.

The motivation for this kind of machine translation system was to treat the encoder (the blue part) as like a memory which we can access later during the decoding process (colored red). Most of the early neural machine translation systems ran the encoder with the input sentence to get a hidden state, and then that hidden state was the input to the decoder which needed to generate the resulting sentence. With this model, we are able to use the other hidden state outputs from the encoder, not just the last one.

1.2 Self-Attention in Transformer Networks

Self attention is an attention mechanism introduced in the Transformer architecture which undergoes similar procedures as the Luong attention. The first step of the attention is to compute Q, K, V using different transformations from the original input embedding as shown in Figure 3.

[1] the original paper proposed multiple alignment scores, the general one they propose is $h_l^\top W_a h_t$, another simple 'location' based attention is just computed using the decoding hidden state at time t: $W_a h_l$

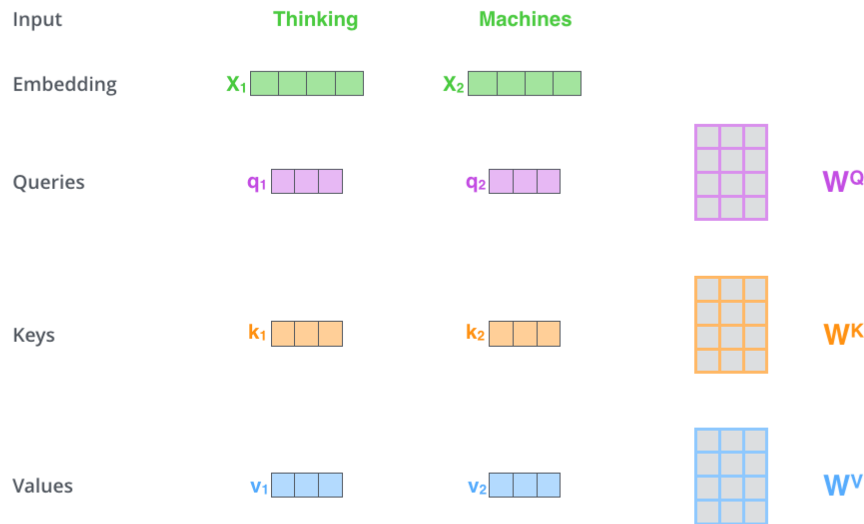

Figure 3: Computing K, Q, V from input embeddings in a Transformer Network

Then, using Q and K, we can compute a dot product as the 'score' of K for Q as shown in Figure 4. Intuitively, Q is the querying term that you would like to find. Its relations for each corresponding K and V pairs (key-value) pairs, can be computed using the key. Note that this dot product is computed across various time steps by matrix multiplication. So we get a score for each K for each Q. We then use a Softmax function to get our attention weights.

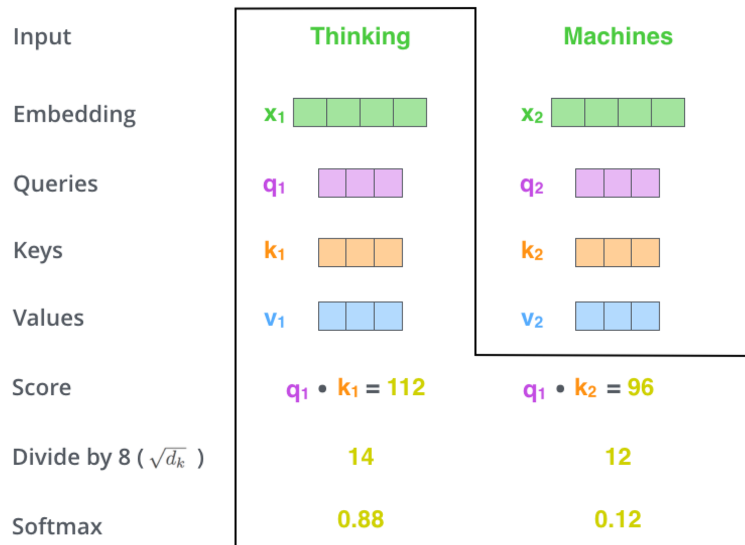

Figure 4: Computing Attention Scores from K, Q, V

Finally, using these weights, we can compute our weighted sum by multiplying the weights with the values. Comparing to the Luong attention, query is analogous to the original h_l, key and query are analogous to the original h_t.

Problem: Attention in RNNs

Explain how we incorporate self-attention into an RNN model at a high-level.

Solution: Attention in RNNs

To incorporate self-attention, we can let each hidden state attend to themselves. In other words, every hidden state attends to the previous hidden states. Put more formally, h_t attends to previous states by,

$$e_{t,l} = \text{score}(h_t, h_l)$$

We apply Softmax to get attention distribution over previous states,

$$\alpha_{t,l} = \frac{\exp(e_{t,l})}{\sum_j \exp(e_{t,j})}$$

We then compute attention output,

$$a_t = \sum_{j=1}^{t-1} \alpha_{t,j} h_j$$

Problem: Generalized Attention in Matrix Form

Consider a form of attention that matches query q to keys $k_1, \ldots, k_t$ in order to attend over associated values $v_1, \ldots, v_t$.

If we have multiple queries $q_1, \ldots, q_l$, how can we write this version of attention in matrix notation?

Solution: Generalized Attention in Matrix Form

Stack queries into a matrix Q, keys into K and values V. Then,

$$a(Q, K, V) = \text{Softmax}(QK^\top)V$$

where Softmax is applied row-wise

Problem: Justifying Scaled Self-Attention

In practice, Transformers use a Scaled Self-Attention. Suppose $q, k \in \mathbb{R}^d$ are two random vectors with $q, k \sim \mathcal{N}(\mu, \sigma^2 I)$, where $\mu \in \mathbb{R}^d$ and $\sigma \in \mathbb{R}^+$

1. Define $\mathbb{E}[q^\top k]$ in terms of μ, σ, d
2. Define $Var(q^\top k)$ in terms of μ, σ, d
3. How does $Var(q^\top k)$ scale with d? Why might this be problematic?
4. Suppose we scale the dot product by $\frac{1}{s}$, i.e. $Var(q^\top k/s)$. What value of s should we choose to address the issue from 3.?

Solution: Justifying Scaled Self-Attention

1.

$$\begin{aligned}\mathbb{E}[q^\top k] &= \mathbb{E}\left[\sum_{i=1}^{d} q_i k_i\right] \\ &= \sum_{i=1}^{d} \mathbb{E}[q_i k_i] \\ &= \sum_{i=1}^{d} \mu_i^2 \\ &= \mu^\top \mu\end{aligned}$$

2. First, notice that if random variables are uncorrelated, then, we have

$$Var\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} Var(X_i)$$

Then,

$$\begin{aligned}Var(q^\top k) &= \mathbb{E}[(q^\top k)^2] - \mathbb{E}[q^\top k]^2 \\ &= \mathbb{E}[q^\top k k^\top q] - (\mu^\top \mu)^2 \\ &= \mathbb{E}[Tr(q q^\top k k^\top)] - (\mu^\top \mu)^2 \\ &= Tr(\mathbb{E}[q q^\top]\mathbb{E}[k k^\top]) - (\mu^\top \mu)^2 \\ &= Tr\left((\mathbb{E}[q]\mathbb{E}[q^\top] + \sigma^2 I)(\mathbb{E}[k]\mathbb{E}[k^\top] + \sigma^2 I)\right) - (\mu^\top \mu)^2 \\ &= Tr\left((\mu\mu^\top + \sigma^2 I)(\mu\mu^\top + \sigma^2 I)\right) - (\mu^\top \mu)^2 \\ &= Tr(\mu\mu^\top\mu\mu^\top) + 2\sigma^2 Tr(\mu\mu^\top) + Tr(\sigma^4 I) - (\mu^\top \mu)^2 \\ &= \mu^\top\mu\mu^\top\mu + 2\sigma^2\mu^\top\mu + d\sigma^4 - (\mu^\top \mu)^2 \\ &= 2d\sigma^2\mu^\top\mu + d\sigma^4\end{aligned}$$

3. $Var(q^\top k)$ is linear in d. As d grows larger, the variance increases, which is problematic.

4. With the scaling term, the variance becomes $\frac{1}{s^2}(2d\sigma^2\mu^\top\mu + d\sigma^4)$. We would like $s = \sqrt{d}$ for the variance to no longer be linear in d.

1.3 Tutorials on Attention Networks and Eager Execution on Tensorflow

Here is a tutorial that we recommend you run through in RNNs with `tf.eager` and `keras`: https://colab.research.google.com/github/tensorflow/tensorflow/blob/r1.9/tensorflow/contrib/eager/python/examples/nmt_with_attention/nmt_with_attention.ipynb

2 Transformers

At a high-level, transformers consist of the Transformer Encoder and Transformer Decoders.

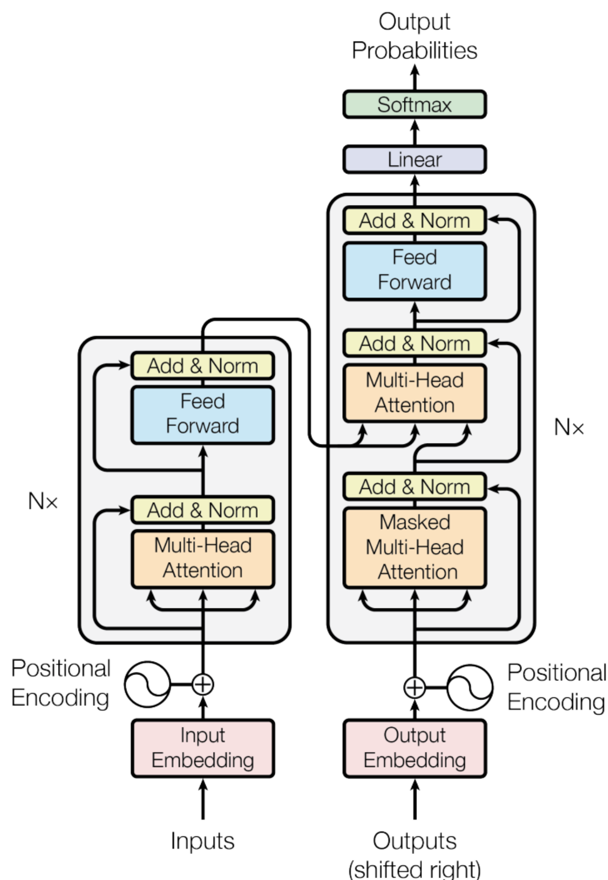

Figure 5: Overview of Transformer architecture

Both operate similarly, except the Transformer Decoder takes x_{target} as input, but Transformer Encoder takes in x_{source} as input. In addition, there are several differences in cross-attention and self-attention operations. In particular, transformers are novel in that they add,

- Positional Encoding: Addresses lack of sequence information
- Multi-headed Attention: Allows querying multiple positions at each layer
- Non-linearities
- Masked Decoding: Prevent attention lookups into the future

2.1 Notations

To ensure a level of clarity, we will let B be the batch size, L_{source} represent the source sequence length, L_{target} be the target sequence length, D represent the model hidden dimension and H represent the number of attention heads.

In particular, transformers receive two sequences as input. The first is $x_{source} \in \mathbb{Z}^{B \times L_{source}}$ and the second is $x_{target} \in \mathbb{Z}^{B \times L_{target}}$. These are integer tensors, and each integer represents a word or token.

2.2 Transformer Encoders

Input & Positional Embedding The source tensor is embedded into the model hidden dimension, and produces a tensor $X_{source} \in \mathbb{R}^{B \times L_{source} \times D}$. We then add a positional encoding that differs for each sequence position in order to enable the model to differentiate the positions in the sequence. In general, we need this information since position of words in a sentence carries information.

Encoder Attention The Encoder Attention is self-attention. Specifically, in Transformer networks, we use the Scaled QKV Attention (not covered explicitly in lecture). In other words, we would like to build a representation of a single sequence such that every position in the sequence has information about every other position in the sequence. In particular, to enable this, we will use the Query-Key-Values (QKV) Attention. Our queries, keys and values will be tensors in $X_{source} \in \mathbb{R}^{B \times L_{source} \times D}$ and weight matrices will be $W_Q, W_K, W_V \in \mathbb{R}^{D \times D}$. Ultimately, we will retrieve,

$$Q = X_{source} W_Q$$
$$K = X_{source} W_K$$
$$V = X_{source} W_V$$

Using Q, K, V, we will compute the attention scores (tensor in $\mathbb{R}^{B \times L_{source} \times L_{source}}$). For each element in the batch, each entry i, j in the matrix would be $\frac{q_i^\top k_j}{\sqrt{D}}$ for scaled dot product attention. Alternatively, we can compute, $\frac{QK^\top}{\sqrt{D}}$. To produce weights over each position in the sequence, we want each score to sum to one over the keys K. To accomplish this, we take a softmax update over the last dimension of the attention scores. Then, to produce the attention update, we multiply these attention weights by our values V,

$$C_{update} = \mathsf{softmax}\left(\frac{QK^\top}{\sqrt{D}}\right) V$$

where $C_{update} \in \mathbb{R}^{B \times L_{source} \times D}$

One of the key changes in Transformers is the *multi-headed attention* mechanism. To turn it into multi-headed attention, we can take any such update matrices and reshape and permute the matrix from shape $B \times L_{source} \times D$ to $B \times H \times L_{source} \times \frac{D}{H}$.

We finally consider padding. In general, we operate on a batch of B sequences, but these sequences may not be the same length. We pad each sequence to L_{source}. To prevent our model from paying attention to padded positions, we add $-\infty$ to attention scores prior to the Softmax of any position that should be ignored.

Feedforward Layer The feedforward layer applies linear transformation to each position, apply a nonlinear activation, then applies a second linear transformation.

2.3 Transformer Decoder

Masked Decoder Self-Attention Masked decoder self-attention is the same as encoder self-attention, but with different masking. In particular, we would like every position to pay attention to all previous positions, but not future positions. To achieve this, we set attention score to $\frac{q_i^\top k_j}{\sqrt{D}}$ if $i \leq j$ and $-\infty$ otherwise.

Encoder-Decoder Attention Encoder-Decoder attention operated similarly as well, except that we have two sequences: (1) generate queries and (2) generate keys-values. Hence, we let $Q = X_{target} W_Q, K = X_{source} W_K, V = X_{source} W_V$, where X_{source} is the output of the transformer encoder on the source sequences.

Problem: Machine Translation

1. What is the reason for positional encoding? How is it typically implemented?
2. What is the advantage of multi-head attention? Give some examples of structures that can be found using multi-head attention
3. For input sequences of length M and output sequences of length N, what are the complexities of (1) Encoder Self-Attention (2) Decoder-Encoder Attention (3) Decoder Self-Attention. Further let k be the hidden dimension of the network
4. Do activation of the encoder depend on decoder activation? How much additional computation is needed to translate a source sequence into a different target language, in terms of M and N?

Solution: Machine Translation

1. Position encoding is used to ensure that word position is known. Because attention is applied symmetrically to all input vectors from the layer below, there is no way for the network to know which positions were filtered through to the output of the attention block. Position encoding also allows the network to compare words (nearby position encodings have high inner product) and find nearby words. We can either use learned position encodings or precomputed sinusoids such that each dimension of the position encoding corresponds to a different sinusoidal frequency.
2. Multi-Head attention allows for a single attention module to attend to multiple parts of an input sequence. This is useful when the output is dependent on multiple inputs (such as in the case of the tense of a verb in translation). Attention heads find features like start of sentence and paragraph, subject/object relations, pronouns, etc.
3. (1) $\mathcal{O}(M^2k)$ (2) $\mathcal{O}(MNk)$ (3) $\mathcal{O}(N^2k)$
4. No. The encoder activations do not depend on the decoder activations. Thus, you only need $\mathcal{O}(MN + N^2)$ additional computation to decode into a new sequence.

2.4 Why Transformers

In general, transformers are good for long-range connections, are easy to parallelize and transformers can be made much deeper than RNNs. On the other hand, attention computations are complex to implement and computations take $\mathcal{O}(n^2)$ time.

However, in practice, it turns out the benefits vastly outweigh the downsides, and transformers work better than RNNs and LSTMs in many cases.