Transformer Notes

Kaili Huang

Feb 1, 2022

1 Transformer

The Transformer is a revolutionary model architecture that tackles the sequence transduction problem in a different way than what we've learnt in the previous lectures (RNNs). By eschewing recurrence and instead relying solely on attention mechanisms to draw the dependencies between input and output, Transformer has outperformed the state-of-the-art models in various tasks with significantly more parallelization, which results in less training time.

2 Encoder-Decoder Structure

The Transformer is an encoder-decoder model, as shown in Figure 1. The encoder maps the input sequence to a sequence of vector representations. Given a sequence of tokens (i.e., words, subwords, etc), $(w_1, w_2, ..., w_n)$, the output of the encoder is a sequence of vectors $Z = (z_1, z_2, ..., z_n)$ where $z_i \in R^{d_{model}}$ is the representation of token w_i. The decoder then takes Z and generates a sequence

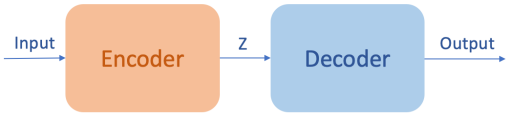

Figure 1: The Encoder-Decoder structure

of output tokens $(y_1, y_2, ..., y_m)$, one token at a time. The decoder is auto-regressive, which means at each step, it takes the previously generated token as an additional input to generate the next token.
We will first introduce attention, one of the core components of the Transformer, then go through the encoder, and then move to the decoder.

3 Attention

3.1 Scaled Dot-Product Attention

The attention mechanism adopted by the Transformer model is *Scaled Dot-Product Attention*. We take the self-attention mechanism in the encoder module (we'll explain the encoder in the next section) as an example.
We are given a sequence of tokens, $s = (w_1, w_2, ..., w_n)$, which have already been mapped into embedding vectors $x_1, x_2, ..., x_n \in R^{d_{model}}$ in the previous modules. We want to determine how $w_i, i \in \{1, 2, ..., n\}$ attends to all the tokens in the sequence. The computation of the attention scores can be decomposed into the following steps:

1. Let $W^Q \in R^{d_{model} \times d_k}, W^K \in R^{d_{model} \times d_k}, W^V \in R^{d_{model} \times d_v}$ denote three parameter matrices that we will train during the training time. With these matrices, we can create a query

vector, a key vector and a value vector for **each** of the tokens.

$$
\begin{aligned}
q_i &= W^Q x_i \\
k_i &= W^K x_i \\
v_i &= W^V x_i \quad \text{for } i \in \{1, 2, ..., n\}
\end{aligned}
$$

We call these vectors "query", "key", and "value" because we want them to represent each token from a particular aspect. The roles they play will be clear in the following steps.

2. Then we calculate a raw attention score for every pair of tokens. For every w_i, we want to calculate how it attends to all the tokens in the sequence. So we use w_i's query vector, and score every token against w_i using their key vectors. Because every token becomes the query for once, we calculate e_{ij} for all the (i, j) pairs.

$$e_{ij} = q_i \cdot k_j \quad \text{for } i, j \in \{1, 2, ..., n\}$$

3. The third step is to multiply all the e_{ij}'s by a scaling factor $\frac{1}{\sqrt{d_k}}$. This leads to more stable gradients.

$$s_{ij} = \frac{e_{ij}}{\sqrt{d_k}}$$

To learn why this specific value is chosen, we can assume the components of q and k are independent random variables with mean 0 and variance 1. Then the dot product $q \cdot k = \sum_{j=1}^{d_k} q_j k_j$ has mean 0 and variance d_k. Hence, we multiply $\frac{1}{\sqrt{d_k}}$ to normalize the variance, thus facilitating the training.

4. Next, we normalize the attention scores with a softmax function.

$$\alpha_{ij} = \text{softmax}(s_{ij}) = \frac{\exp(s_{ij})}{\sum_{k=1}^{n} \exp(s_{ik})}$$

5. The final step is to multiply the attention score with the value vectors and sum them up.

$$z_i = \sum_{j=1}^{d_k} \alpha_{ij} v_j$$

That concludes the self-attention calculation for an input sequence. The resulting vectors $(z_1, z_2, ..., z_n)$ can then be sent to the following neural networks. In conclusion, the self-attention layer calculates a weighted sum of value vectors, where the weights are determined by the dot-product of query vectors and key vectors.

3.2 The Matrix Form

In practice, it's often more useful to do computations with matrices. Assume we have a batch of embedding vectors, $X \in R^{b \times d_{model}}$, together with the parameter matrices W^Q, W^K, W^V described above. The self-attention calculation in the matrix form can be done as follows:

1. Compute $Q \in R^{b \times d_k}, K \in R^{b \times d_k}, V \in R^{b \times d_v}$ matrices, as shown in Figure 2.

$$
\begin{aligned}
Q &= XW^Q \\
K &= XW^K \\
V &= XW^V
\end{aligned}
$$

2. Calculate the raw attention score matrix $E \in R^{b \times b}$.

$$E = QK^T$$

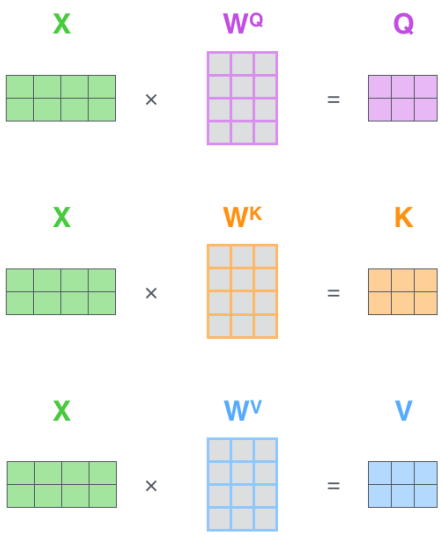

Figure 2: Calculate the Q,K,V matrices

3. Apply the scaling factor to E.

$$S = \frac{E}{\sqrt{d_k}}$$

4. Use softmax to normalize the attention matrix to get $T \in R^{b \times b}$. Every row of T is normalized and has norm 1.

$$T = \text{softmax}(S)$$

5. Calculate the weighted sum $Z \in R^{b \times d_v}$, as shown in Figure 3.

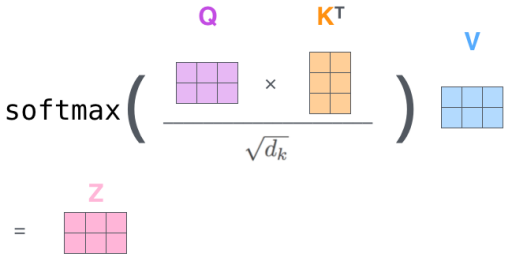

Figure 3: Calculate the self-attention

$$Z = TV = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

3.3 Multi-Head Attention

As discussed above, the parameter matrices W^Q, W^K, W^V are to extract useful information from the sequence from a certain aspect. However, as natural language often contains very rich meanings, a single set of parameter matrices can fall short of generating a good representation of the sequence. Thus, we can benefit from having multiple sets of W^Q, W^K, W^V matrices, which are learned in parallel and ideally can project the input embeddings into different representation subspaces.

This mechanism is called multi-head attention. Each attention output we calculated in the previous sections is called an attention head. $head_i$ (i.e., Z_i) is calculated with parameter matrices W_i^Q, W_i^K, W_i^V:

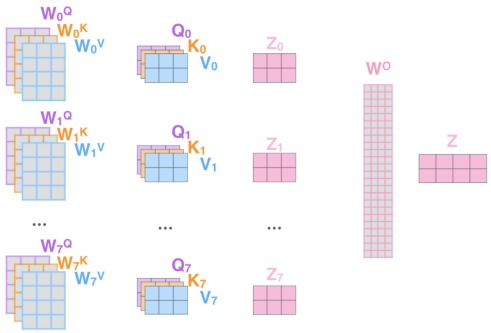

Figure 4: Calculate the multi-head attention

$$head_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i = \text{softmax}\left(\frac{(XW_i^Q)(XW_i^K)^T}{\sqrt{d_k}}\right)(XW_i^V)$$

Assume we're using the h-head attention. Let $W^O \in R^{hd_v \times d}$ denote another parameter matrix that we will train. We first concatenate the h attention heads, and then use W^O to project the embedding matrix back to $R^{b \times d_{model}}$. Figure 4 shows this process.

$$M = \text{Concat}(head_1, head_2, ..., head_h)W^O$$

For the hyper-parameters, Vaswani et al. uses $h = 8$ and $d_k = d_v = d_{model}/h = 64$.

4 Encoder

Let's first take a look at a single layer in the encoder. As Figure 5 shows, we can break an encoder layer into two sub-layers: a self-attention layer, followed by a feed-forward neural network (FFN). Let $X = (x_1, x_2, ..., x_n)$ denote the input of a sub-layer, where each $x_i \in R^{d_{model}}$ is a vector

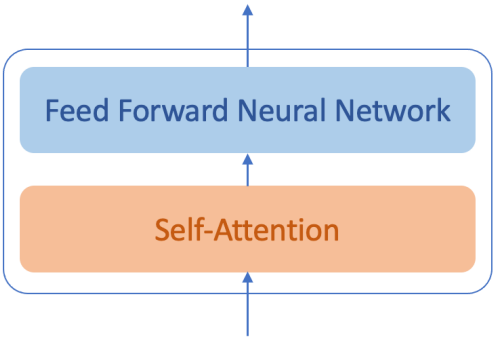

Figure 5: A (simplified) encoder layer

representation of the original input token w_i. Then let's analyze the outputs of the two sub-layers.

1. Self-attention. As the previous section introduced, it applies linear transformations to X and produces a sequence of query, key, and value vectors Q, K, V. Then the multi-head scaled dot-product attention is calculated.

2. Feed-forward neural network. It consists of two linear transformations with a RELU activation in between, and it operates on each x_i.

$$\text{FFN}(x_i) = \text{RELU}(x_i W_1 + b_1)W_2 + b_2 \quad \text{for } i \in \{1, 2, ..., n\}$$

The FFN is called point-wise, or position-wise, because it applies exactly the same two linear transformations for every position i within the same layer.

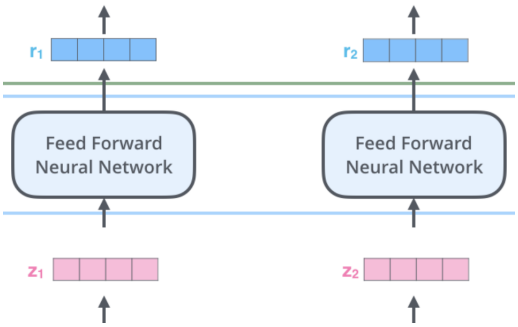

Figure 6: Position-wise FFN: apply the same linear transformations to every position within the same layer

The encoder stacks N such layers. The layers are of exactly the same architecture, but they don't share parameters. All sub-layers produce outputs of dimension d_{model}. The vanilla Transformer proposed by Vaswani et al. sets $N = 6$ and $d_{model} = 512$.

5 Positional Encoding

Since the model does not use a recurrent structure, it cannot take the order of the input sequence into account. As a result, we lose a significant amount of information. Therefore, the Transformer adds the positional encoding (PE) to the input embeddings before sending the embeddings into the encoder layers.
The positional encoding uses sine and cosine functions of different frequencies. Here pos is the position of a token and $2i, 2i+1$ represent the dimensions.

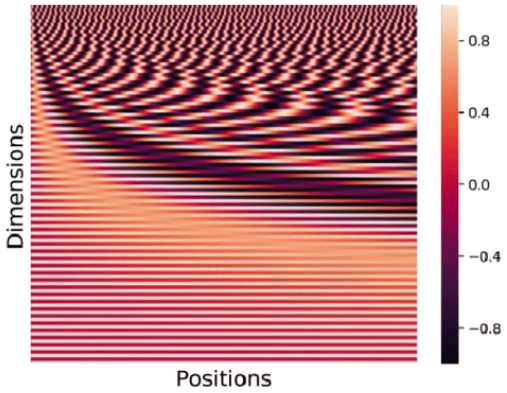

Figure 7: Positional encoding

$$\begin{aligned} \mathrm{PE}_{(pos,2i)} &= \sin\left(pos/10000^{2i/d_{\text{model}}}\right) \\ \mathrm{PE}_{(pos,2i+1)} &= \cos\left(pos/10000^{2i/d_{\text{model}}}\right) \\ \text{for } i &\in \{0, 1, ..., d_{model}/2 - 1\} \end{aligned}$$

We can also take a look at the positional encoding of a given position k (assume $d_{model} = 256$):

$$p_k = \begin{pmatrix} \sin\left(k/10000^{0/256}\right) \\ \cos\left(k/10000^{0/256}\right) \\ \vdots \\ \sin\left(k/10000^{254/256}\right) \\ \cos\left(k/10000^{254/256}\right) \end{pmatrix}$$

Figure 7 shows a demo of the personal encoding. A row corresponds to a certain dimension, and a column corresponds to a certain position.

One advantage of the sinusoidal positional encoding is that it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.
The positional encoding is added to the token embeddings, and then the embeddings are fed into the encoder.

6 Decoder

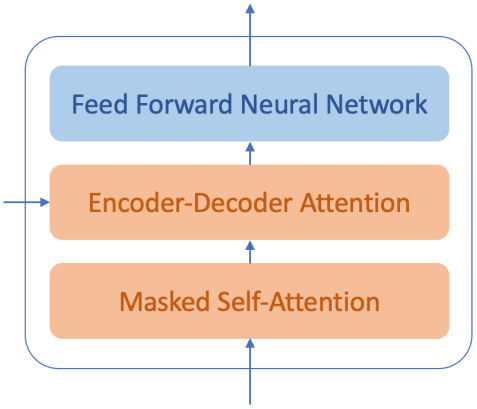

Figure 8: A (simplified) decoder layer

After analyzing the encoder structure, let's take a look at the decoder. As Figure 8 shows, we can break a decoder layer into three sub-layers: a masked self-attention layer, an encoder-decoder attention layer, and a feed-forward neural network.
Again, let $X = (x_1, x_2, ..., x_n)$ denote the input of a sub-layer (the output of the previous decoder sub-layer), where each $x_i \in R^{d_{model}}$ is a vector representation of the original input token w_i, and let's analyze the outputs of the three sub-layers.

1. Masked self-attention. Every position attends to all the history positions and itself, but not to the future positions. That is to say, for vector x_i, it should only attend to $\{x_1, x_2, ..., x_i\}$. This is because the decoder is auto-regressive, which means it generates one token at a time, and it takes the previously generated token as an additional input. We want to prevent the earlier steps from "seeing" the future ones, in order to emulate the auto-regressive process.
   As shown in Figure 9, we can implement this inside of scaled dot-product attention by masking out (setting to $-\infty$) all values in the input of the softmax which correspond to illegal connections. In that way, the output of softmax for the illegal connections will be 0.

2. Encoder-decoder attention. It performs multi-head attention over the output of the encoder stack. This layer takes inputs from two sources: the previous decoder sub-layer, and the encoder layer.
   Let $Z = (z_1, z_2, ..., z_n)$ denote the output of the uppermost encoder layer. In this attention layer, the query is from this decoder sub-layer's input, and the key and value are from the encoder output:

$$Q = XW^Q$$
$$K = ZW^K$$
$$V = ZW^V$$

   Then the multi-head scaled dot-product attention is calculated.

3. Feed-forward neural network. Same as the encoder.

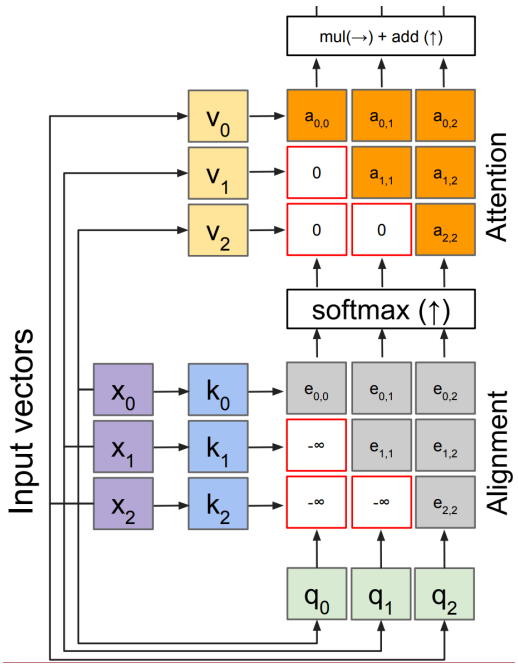

Figure 9: Masked self-attention: assign $-\infty$ to zero the softmax outputs

7 Other Optimization

7.1 Residual Connections

Residual connections were first proposed by He et al. in 2015 and were proven effective to ease the training of deep neural networks. The Transformer makes use of the residual connections by applying them after every sub-layers in the encoder and decoder. Then the output of each sublayer is changed to $x + \text{Sublayer}(x)$.

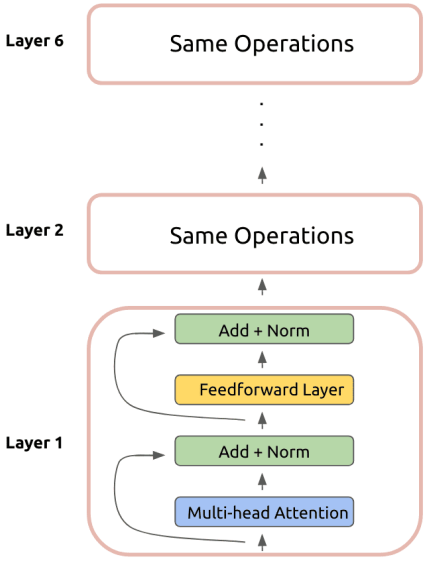

Figure 10: The encoder module with the residual connections and the layer normalization

7.2 Layer Normalization

Layer normalization, proposed by Ba et al. in 2016, is a useful method to stabilize the hidden state and reduce the training time. The Transformer also adopts this optimization in both the

encoder and decoder. For each layer, the layer normalization statistics over all the hidden units in the same layer are calculated as:

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \quad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

And then the inputs are normalized to be:

$$x^{\ell'} = \frac{x^\ell - \mu^\ell}{\sigma^\ell + \epsilon}$$

Combined with the residual connections, the original Sublayer(x) transformation is changed to LayerNorm(x + Sublayer(x)). Figure 10 shows the encoder layers with the residual connections and the layer normalization. The same tricks are applied to the decoder as well.

8 The Whole Picture

Putting all these modules together, we get a whole picture of the Transformer, as shown in Figure 11. The Transformer model achieved a new state-of-the-art (SOTA) on both WMT 2014

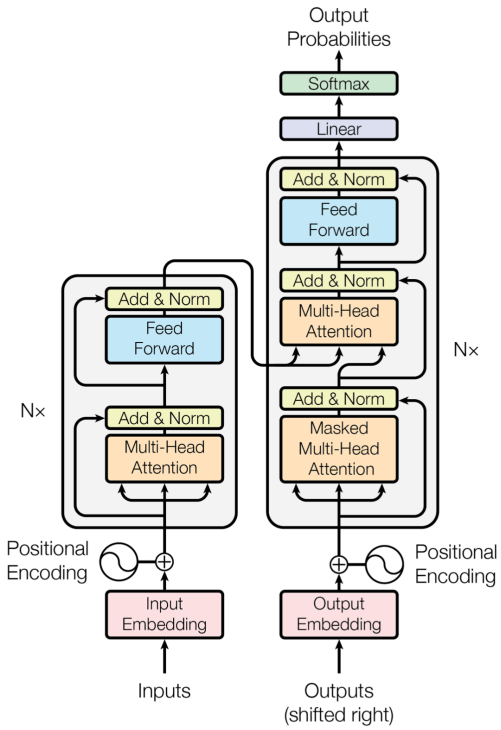

Figure 11: The optimized encoder

English-to-German and WMT 2014 English-to-French translation tasks at the time it was proposed. Nowadays, the Transformer has been applied to address a lot of different NLP tasks and achieves outstanding performance. It also shows promising performance in domains out of NLP. Besides, the Transformer architecture inspired various brilliant research findings, e.g., Transformer-XL, BERT, GPT-2/3, etc. Its revolutionary architecture and remarkable achievements have made it an especially important member of the NLP family.