Lecture 7 Notes

Deep Learning AI335

1 Sequence to Sequence (Seq2Seq) [1, 2, 3]

Seq2Seq is a family of machine learning approaches used in Natural Language Processing (NLP). Seq2seq models are used in fields such as Neural Machine Translation (NMT), image captioning, and text summarization.

Seq2seq maps maps an input sequence into a real-numerical vector (sometimes called a *context vector*) by a neural network (*the encoder*), then maps it back to an output sequence using another neural network (*the decoder*).

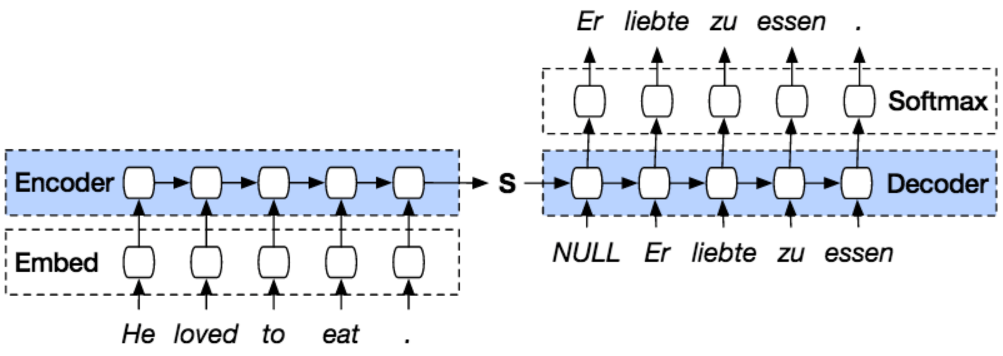

Figure 1: A Seq2Seq model being used for NMT.

1.1 Recurrent Neural Networks (RNNS)

Traditionally, the encoder and decoders used in a seq2seq model were RNNs, which operate by using activations from previous time steps and layers to maintain a *hidden state*, which summarizes the input sequence up to that point, and propagating this state forward through the network.

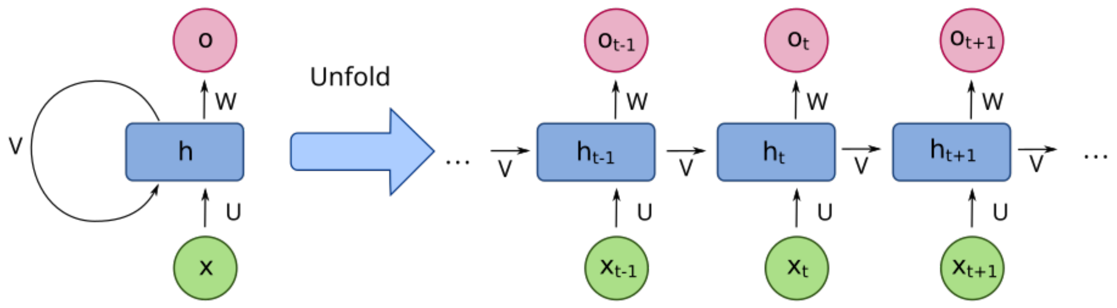

Figure 2: Compressed (left) and unfolded (right) basic RNN.

RNNs process inputs one step at a time, maintaining a hidden state $\mathbf{h}_t$ that captures information from previous steps. At each time step t, the RNN updates its hidden state as follows:

$$\mathbf{h}_t = \sigma\left(\mathbf{W}_{hx} \cdot \mathbf{x}_t + \mathbf{W}_{hh} \cdot \mathbf{h}_{t-1}\right), \tag{1}$$

where $\mathbf{x}_t$ is the input at time t, $\mathbf{h}_{t-1}$ is the previous hidden state, and σ is a non-linear activation function. The output at time t is computed as:

$$\mathbf{y}_t = \sigma\left(\mathbf{W}_{yh} \cdot \mathbf{h}_t\right). \tag{2}$$

1.2 Encoder

The *encoder* in a Seq2Seq model processes the input sequence and compresses it into a fixed-size real-valued vector, often referred to as the *context vector*. It operates over the input sequence $\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T]$ by iteratively updating its hidden state $\mathbf{h}_t$:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}), \tag{3}$$

The final hidden state $\mathbf{h}_T$ serves as the context vector, encapsulating the entire input sequence. This context vector is passed to the decoder to generate the output sequence.

1.3 Decoder

The *decoder* generates the output sequence $\mathbf{y} = [\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_N]$ from the context vector produced by the encoder. At each time step t, the decoder computes the hidden state $\mathbf{h}'_t$ and the output $\mathbf{y}_t$ as:

$$\mathbf{h}'_t = \sigma(\mathbf{y}_{t-1}, \mathbf{h}'_{t-1}, \mathbf{c}), \tag{4}$$

$$\mathbf{y}_t = \text{softmax}\left(\mathbf{W}_y \cdot \mathbf{h}'_t\right), \tag{5}$$

where $\mathbf{c}$ is the context vector from the encoder, and $\mathbf{W}_y$ is the weight matrix for generating the output.

The decoder operates *autoregressively*, where the previously generated output $\mathbf{y}_{t-1}$ is fed as input to predict the next token $\mathbf{y}_t$.

1.4 Drawbacks

The traditional Seq2Seq architecture faces several challenges:

- **Bottleneck Problem:** The fixed-size context vector $\mathbf{c}$ serves as the sole representation of the input sequence. For long or complex sequences, critical information may be lost, leading to degraded performance (*vanishing gradients*).
- **Limited Parallelization:** The sequential nature of RNNs makes training and inference slower compared to modern architectures like Transformers, which process sequences in parallel.

2 Attention Mechanism [4, 5, 6]

The *attention mechanism* was developed to address the weaknesses of leveraging information from the hidden layers of recurrent neural networks. Attention allows a token equal access to any part of a sentence directly, rather than only through the previous state.

2.1 Attention

Attention computes a weighted sum of the encoder's hidden states, where the weights are determined dynamically based on the relevance of each hidden state to the current decoding step. For a decoder hidden state $\mathbf{h}'_t$, the attention score for each encoder hidden state $\mathbf{h}_i$ is calculated as:

$$\alpha_{t,i} = \frac{\exp\left(\text{score}(\mathbf{h}'_t, \mathbf{h}_i)\right)}{\sum_{j=1}^{T} \exp\left(\text{score}(\mathbf{h}'_t, \mathbf{h}_j)\right)}, \tag{6}$$

where $\alpha_{t,i}$ is the attention weight for encoder state $\mathbf{h}_i$ at decoding step t, and $\text{score}(\mathbf{h}'_t, \mathbf{h}_i)$ is a similarity measure (e.g., dot product, additive, or scaled dot-product).

The context vector $\mathbf{c}_t$ for each decoding step is then computed as:

$$\mathbf{c}_t = \sum_{i=1}^{T} \alpha_{t,i} \cdot \mathbf{h}_i. \tag{7}$$

The decoder combines $\mathbf{c}_t$ with its hidden state to generate the output token. Attention allows the model to dynamically "attend" to the most relevant parts of the input sequence for each output step.

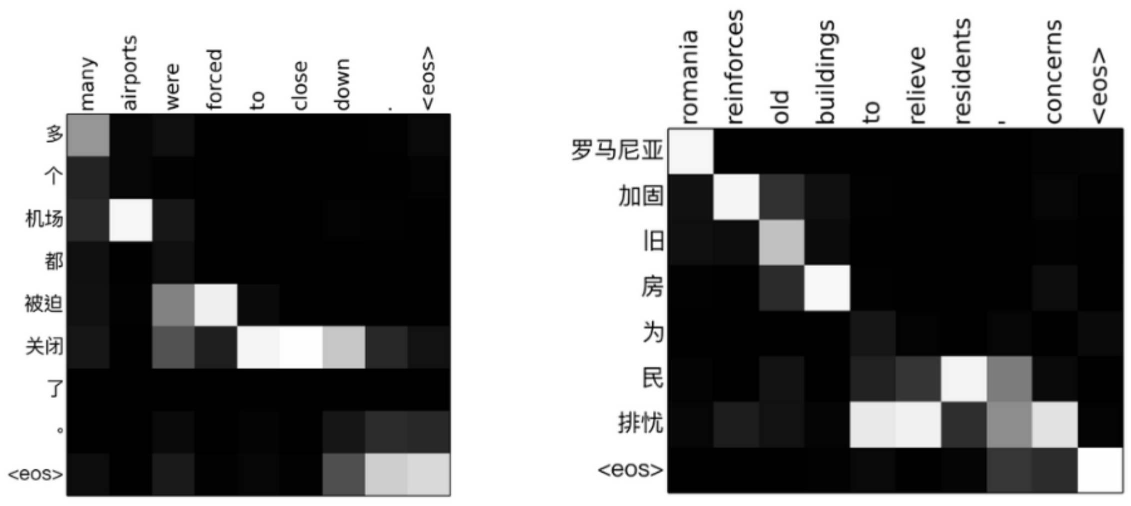

Figure 3: Visualizations of the attention mechanism for NMT. Brighter squares indicate a higher attention score $\alpha_{t,i}$ between the input and output sequences.

Scaled Dot-Product Attention

Scaled dot-product attention is a modification of the attention mechanism where we project input sequences into three distinct spaces using a set of weight matrices.

A single *head* of scaled dot-product attention takes three sequences, a query sequence $\mathbf{X}_q$, a key sequence $\mathbf{X}_k$, and a value sequence $\mathbf{X}_v$. Each input sequence is projected into it's new space using it's respective weight matrix:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V. \tag{8}$$

The attention scores are computed as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}, \tag{9}$$

where d_k is the dimensionality of the keys. Scaling by $\sqrt{d_k}$ ensures stable gradients.

Scaled dot-product attention may be applied to a single input sequence ($\mathbf{X}_q, \mathbf{X}_k, \mathbf{X}_v = \mathbf{X}$), this is called *self-attention*, or to different input sequences ($\mathbf{X}_q = \mathbf{X}_1$, $\mathbf{X}_k$, $\mathbf{X}_v = \mathbf{X}_2$), this is called *cross-attention*.

2.2 Multiheaded Attention

Multiheaded attention improves the expressiveness of the attention mechanism by using multiple attention heads, each learning to focus on different parts of the input. Each head computes scaled dot-product attention independently:

$$\text{MultiheadedAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}_{i \in [n_{\text{heads}}]}(\text{Attention}(\mathbf{X} \cdot \mathbf{W}_i^Q, \mathbf{X} \cdot \mathbf{W}_i^K, \mathbf{X} \cdot \mathbf{W}_i^V))\mathbf{W}^O, \tag{10}$$

where the matrix $\mathbf{X}$ is the concatenation of word embeddings, and the matrices $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V$ are "projection matrices" owned by individual attention head i, and $\mathbf{W}^O$ is a final projection matrix owned by the whole multi-headed attention head.

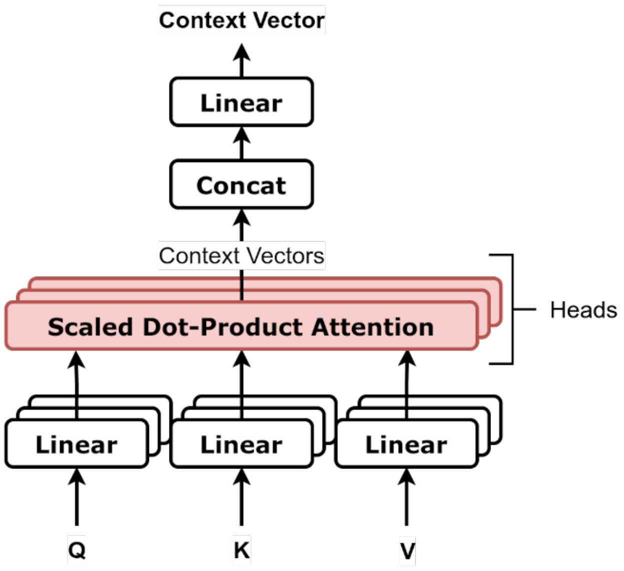

Figure 4: Multiheaded attention.

3 Transformers [6]

Transformers are a type of Seq2Seq model that replaces RNNs with self-attention mechanisms. Transformers consist of two main components: an encoder and a decoder, both built using self-attention and feedforward layers.

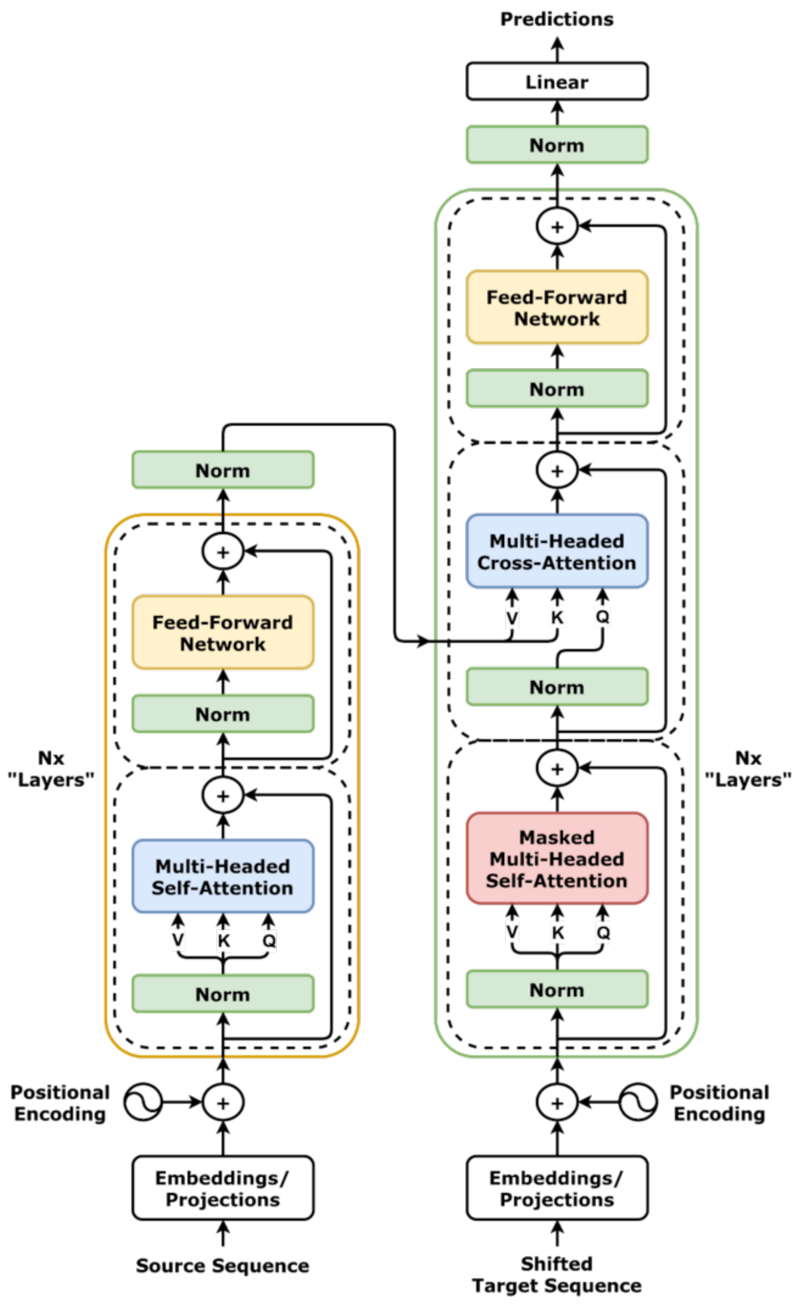

Figure 5: Transformer architecture.

3.1 Positional Encoding

Since Transformers lack the inherent sequential structure of RNNs, *positional encodings* are added to embeddings to provide order information, so that, for example, the input sequence 'man bites dog' is processed differently from 'dog bites man'.

The positional encoding is defined as a function $f : \mathbb{R} \to \mathbb{R}^d$:

$$f(t)_{2k}, f(t)_{2k+1}) = (\sin(\theta), \cos(\theta)) \quad \forall k \in \{0, 1, \ldots, d/2 - 1\}, \tag{11}$$

where:

$$\theta = \frac{t}{r^k}, r = N^{2/d},$$

here N is a parameter that should be significantly larger than the biggest k that would be input into the positional encoding function e.g, 10000.

3.2 Encoder

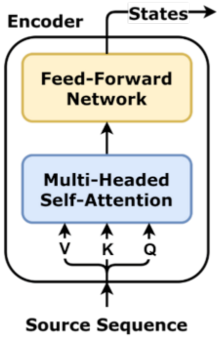

Figure 6: A single encoder layer in a transformer.

The encoder processes the input sequence into a sequence of high-dimensional representations. Each encoder block/layer contains:

- **Multiheaded Self-Attention:** Takes in input sequence and produces an intermediate sequence of vectors $[\mathbf{h}_0, \mathbf{h}_1, \cdots]$, which are combined into a matrix $\mathbf{H}$.
- **Feedforward Network:** Applies transformations to each row (position) in the matrix H.

$$\text{EncoderLayer}(\mathbf{H}) = \text{FFN}(\text{MultiheadedAttention}(\mathbf{H}, \mathbf{H}, \mathbf{H})) \tag{12}$$

3.3 Decoder

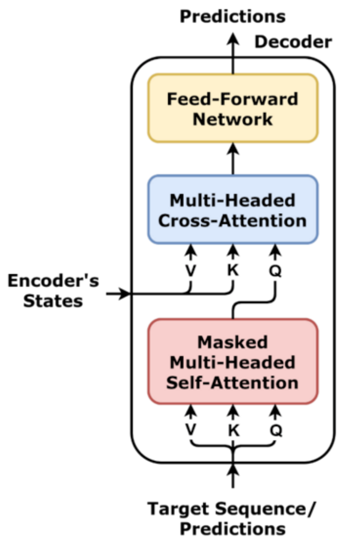
Predictions
Decoder
Feed-Forward Network
Multi-Headed Cross-Attention
V
K
Q
Encoder's States
Masked Multi-Headed Self-Attention
V
K
Q
Target Sequence/ Predictions

Figure 7: A single decoder layer from a transformer

Each decoder block/layer contains:

- **Masked Multiheaded Self-Attention:** Takes in the output/target sequence as a matrix of embedding $\mathbf{H}$ and produces an intermediate matrix $\mathbf{H}'$. The decoder uses a mask so that future output cannot be used to predict current output:

$$\text{MaskedAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\mathbf{M} + \frac{\mathbf{Q} \cdot \mathbf{K}^{\mathrm{T}}}{\sqrt{d_k}}\right)\mathbf{V}, \tag{13}$$

  where M is a matrix that *masks* future values (sometimes called a *causal mask*):

$$\mathbf{M}_{\text{causal}} = \begin{bmatrix} 0 & -\infty & -\infty & \dots & -\infty \\ 0 & 0 & -\infty & \dots & -\infty \\ 0 & 0 & 0 & \dots & -\infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

- **Multiheaded Cross-Attention:** Attends to the output vectors of the encoder $\mathbf{H}^E$.
- **Feedforward Network:** Applies position-wise transformations to the output from the cross attention.

$$\mathbf{H}' = \text{MaskedMultiheadedAttention}(\mathbf{H}, \mathbf{H}, \mathbf{H}), \tag{14}$$

$$\text{MultiheadedAttention}(\mathbf{H}', \mathbf{H}^E, \mathbf{H}^E)). \tag{15}$$

The final decoder layer in a transformer is followed by an *un-embedding* layer, to produce the output probabilities over the vocabulary. Then, one of the tokens is sampled according to the probability, and the decoder can be run again to produce the next token, etc, autoregressively generating output text.

3.4 Layer Normalization

Layer normalization (LayerNorm, or LN), while conceptually unnecessary, are necessary for numerical stability and convergence. LayerNorm is applied position-wise similarly to FNN.

LayerNorm can be applied in two ways:

1. **Post-LN -**
$$\text{LayerNorm}(x + \text{Sublayer(x)}) \tag{16}$$

2. **Pre-LN -**
$$x + \text{LayerNorm}(\text{Sublayer(x)}) \tag{17}$$

The original transformer used post-LN. However, subsequent work has shown that pre-LN allows for faster convergence.

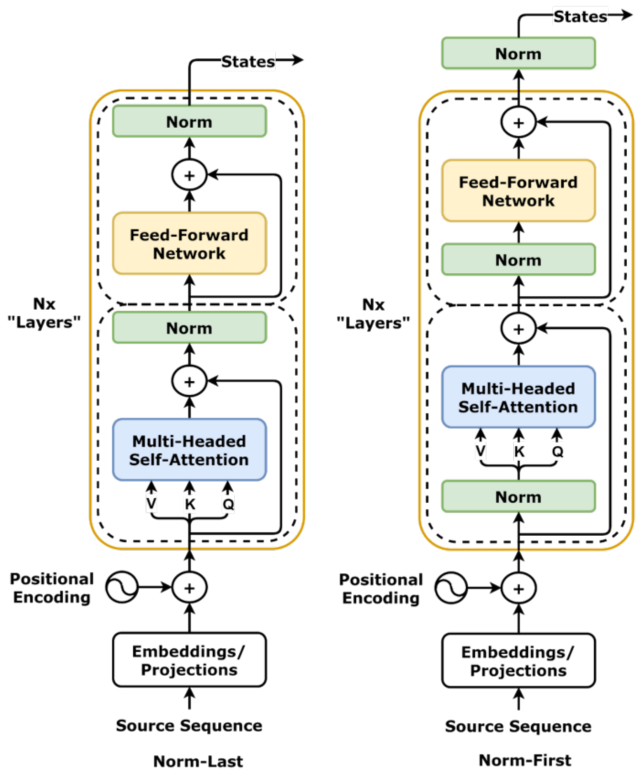

Figure 8: Comparison between post-LN and pre-LN conventions in an encoder block.

References

[1] Wikipedia contributors. Recurrent neural network — Wikipedia, the free encyclopedia, 2024. [Online; accessed 29-December-2024].

[2] Wikipedia contributors. Seq2seq — Wikipedia, the free encyclopedia, 2024. [Online; accessed 29-December-2024].

[3] GeeksforGeeks Contributors. Seq2seq model in machine learning, n.d. Accessed: 2024-12-29.

[4] Wikipedia contributors. Attention (machine learning) — Wikipedia, the free encyclopedia, 2024. [Online; accessed 30-December-2024].

[5] Zhaopeng Tu, Zhengdong Lu, Yang Liu, Xiaohua Liu, and Hang Li. Modeling coverage for neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 2016.

[6] Wikipedia contributors. Transformer (deep learning architecture) — Wikipedia, the free encyclopedia, 2024. [Online; accessed 30-December-2024].