CHAPTER 9

Transformers

Transformers are a very recent family of architectures that have revolutionized fields like natural language processing (NLP), image processing, and multi-modal generative AI.

Transformers were originally introduced in the field of NLP in 2017, as an approach to process and understand human language. Human language is inherently sequential in nature (e.g., characters form words, words form sentences, and sentences form paragraphs and documents). Prior to the advent of the transformers architecture, recurrent neural networks (RNNs) briefly dominated the field for their ability to process sequential information (RNNs are described in Appendix C for reference). However, RNNs, like many other architectures, processed sequential information in an iterative/sequential fashion, whereby each item of a sequence was individually processed one after another. Transformers offer many advantages over RNNs, including their ability to process all items in a sequence in a *parallel* fashion (as do CNNs).

Like CNNs, transformers factorize the signal processing problem into stages that involve independent and identically processed chunks. However, they also include layers that mix information across the chunks, called *attention layers*, so that the full pipeline can model dependencies between the chunks.

In this chapter, we describe transformers from the bottom up. We start with the idea of embeddings and tokens (Section 9.1). We then describe the attention mechanism (Section 9.2). And finally we then assemble all these ideas together to arrive at the full transformer architecture in Section 9.3.

9.1 Vector embeddings and tokens

Before we can understand the attention mechanism in detail, we need to first introduce a new data structure and a new way of thinking about neural processing for language.

The field of NLP aims to represent words with vectors (aka *word embeddings*) such that they capture semantic meaning. More precisely, the degree to which any two words are related in the 'real-world' to us humans should be reflected by their corresponding vectors (in terms of their numeric values). So, words such as 'dog' and 'cat' should be represented by vectors that are more similar to one another than, say, 'cat' and 'table' are. Nowadays, it's also typical for every individual occurrence of a word to have its own distinct representation/vector. So, a story about a dog may mention the word 'dog' a dozen times, with

each vector being slightly different based on its context in the sentence and story at large.

To measure how similar any two word embeddings are (in terms of their numeric values) it is common to use *cosine similarity* as the metric:

$$\frac{u^T v}{|u|\,|v|} = \cos\langle u, v\rangle\,, \tag{9.1}$$

where $|u|$ and $|v|$ are the lengths of the vectors, and $\langle u, v\rangle$ is the angle between u and v. The cosine similarity is $+1$ when $u = v$, zero when the two vectors are perpendicular to each other, and -1 when the two vectors are diametrically opposed to each other. Thus, higher values correspond to vectors that are numerically more similar to each other.

While word embeddings – and various approaches to create them – have existed for decades, the first approach that produced astonishingly effective word embeddings was *word2vec* in 2012. This revolutionary approach was the first highly-successful approach of applying deep learning to NLP, and it enabled all subsequent progress in the field, including Transformers. The details of *word2vec* are beyond the scope of this course, but we note two facts: (1) it created a single word embedding for each distinct word in the training corpus (not on a per-occurrence basis); (2) it produced word embeddings that were so useful, many relationships between the vectors corresponded with real-world semantic relatedness. For example, when using *Euclidean distance* as a distance metric between two vectors, word2vec produced word embeddings with properties such as (where $v_{\texttt{word}}$ is the vector for `word`):

$$v_{\texttt{paris}} - v_{\texttt{france}} + v_{\texttt{italy}} \approx v_{\texttt{rome}} \tag{9.2}$$

This corresponds with the real-world property that Paris is to France what Rome is to Italy. This incredible finding existed not only for geographic words but all sorts of real-world concepts in the vocabulary. Nevertheless, to some extent, the exact values in each embedding is arbitrary, and what matters most is the holistic relation between all embeddings, along with how performant/useful they are for the exact task that we care about.

For example, an embedding may be considered good if it accurately captures the conditional probability for a given word to appear next in a sequence of words. You probably have a good idea of what words might typically fill in the blank at the end of this sentence:

> After the rain, the grass was _____

Or a model could be built that tries to correctly predict words in the middle of sentences:

> The child fell _____ during the long car ride

The model can be built by minimizing a loss function that penalizes incorrect word guesses, and rewards correct ones. This is done by training a model on a very large corpus of written material, such as all of Wikipedia, or even all the accessible digitized written materials produced by humans.

While we will not dive into the full details of *tokenization*, the high-level idea is straightforward: the individual inputs of data that are represented and processed by a model are referred to as *tokens*. And, instead of processing each word as a whole, words are typically split into smaller, meaningful pieces (akin to syllables). Thus, when we refer to tokens, know that we're referring to each individual input, and that in practice, nowadays, they tend to be sub-words (e.g., the word 'talked' may be split into two tokens, 'talk' and 'ed').

9.2 Query, key, value, and attention

Attention is a strategy for processing global information efficiently, focusing just on the parts of the signal that are most salient to the task at hand.

It might help our understanding of the "attention" mechanism to think about a dictionary look-up scenario. Consider a dictionary with keys k mapping to some values $v(k)$. For example, let k be the name of some foods, such as `pizza`, `apple`, `sandwich`, `donut`, `chili`, `burrito`, `sushi`, `hamburger`, ... The corresponding values may be information about the food, such as where it is available, how much it costs, or what its ingredients are.

What we present below is the so-called "dot-product attention" mechanism; there can be other variants that involve more complex attention functions

Suppose that instead of looking up foods by a specific name, we wanted to query by cuisine, e.g., "`mexican`" foods. Clearly, we cannot simply look for the word "`mexican`" among the dictionary keys, since that word is not a food. What does work is to utilize again the idea of finding "similarity" between vector embeddings of the query and the keys. The end result we'd hope to get, is a probability distribution over the foods, $p(k|q)$ indicating which are best matches for a given query q. With such a distribution, we can look for keys that are semantically close to the given query.

More concretely, to get such distribution, we follow these steps: First, embed the word we are interested in ("`mexican`" in our example) into a so-called query vector, denoted simply as $q \in \mathbb{R}^{d_k \times 1}$ where d_k is the embedding dimension.

Next, suppose our given dictionary has n number of entries/entries, we embed each one of these into a so-called key vector. In particular, for each of the j^{th} entry in the dictionary, we produce a $k_j \in \mathbb{R}^{d_k \times 1}$ key vector, where $j = 1, 2, 3, \ldots, n$.

We can then obtain the desired probability distribution using a softmax (see Chapter 6) applied to the inner-product between the key and query:

$$p(k|q) = \mathrm{softmax}\left([q^T k_1; q^T k_2; q^T k_3; \ldots, q^T k_n]\right)$$

This vector-based lookup mechanism has come to be known as "attention" in the sense that $p(k|q)$ is a conditional probability distribution that says how much attention should be given to the key k_j for a given query q.

In other words, the conditional probability distribution $p(k|q)$ gives the "attention weights," and the weighted average value

$$\sum_j p(k_j|q)\, v_j \tag{9.3}$$

is the "attention output."

The meaning of this weighted average value may be ambiguous when the values are just words. However, the attention output really becomes meaningful when the value are projected in some semantic embedding space (and such projection are typically done in transformers via learned embedding weights).

The same weighted-sum idea generalizes to multiple query, key, and values. In particular, suppose there are n_q number of queries, n_k number of keys (and therefore n_k number of values), one can compute an attention matrix

$$A = \begin{bmatrix} \mathrm{softmax}\left(\left[q_1^\top k_1 & q_1^\top k_2 & \cdots & q_1^\top k_{n_k}\right]/\sqrt{d_k}\right) \\ \mathrm{softmax}\left(\left[q_2^\top k_1 & q_2^\top k_2 & \cdots & q_2^\top k_{n_k}\right]/\sqrt{d_k}\right) \\ \vdots \\ \mathrm{softmax}\left(\left[q_{n_q}^\top k_1 & q_{n_q}^\top k_2 & \cdots & q_{n_q}^\top k_{n_k}\right]/\sqrt{d_k}\right) \end{bmatrix} \tag{9.4}$$

Here, $\mathrm{softmax}_j$ is a softmax over the n_k-dimensional vector indexed by j, so in Eq. 9.2 this means a softmax computed over keys. In this equation, the normalization by $\sqrt{d_k}$ is

done to reduce the magnitude of the dot product, which would otherwise grow undesirably large with increasing d_k, making it difficult for (overall) training.

Let α_{ij} be the entry in ith row and jth column in the attention matrix A. Then α_{ij} helps answer the question "which tokens $x^{(j)}$ help the most with predicting the corresponding output token $y^{(i)}$?" The attention output is given by a weighted sum over the values:

$$y^{(i)} = \sum_{j=1}^{n} \alpha_{ij} v_j$$

9.2.1 Self Attention

Self-attention is an attention mechanism where the keys, values, and queries are all generated from the same input.

At a very high level, typical transformer with self-attention layers maps $\mathbb{R}^{n \times d} \longrightarrow \mathbb{R}^{n \times d}$. In particular, the transformer takes in data (a sequence of tokens) $X \in \mathbb{R}^{n \times d}$ and for each token $x^{(i)} \in \mathbb{R}^{d \times 1}$, it computes (via learned projection, to be discussed in Section 9.3.1), a query $q_i \in \mathbb{R}^{d_q \times 1}$, key $k_i \in \mathbb{R}^{d_k \times 1}$, and value $v_i \in \mathbb{R}^{d_v \times 1}$. In practice, $d_q = d_k = d_v$ and we often denote all three embedding dimension via a unified d_k.

Note that d_k differs from d: d is the dimension of raw input token $\in \mathbb{R}^{d_q \times 1}$

The self-attention layer then take in these query, key, and values, and compute a self-attention matrix

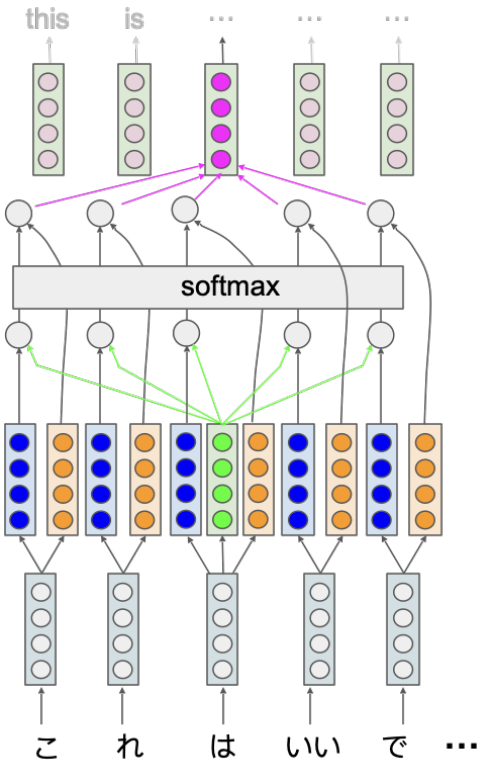
$$A = \begin{bmatrix} \text{softmax}\left(\begin{bmatrix} q_1^\top k_1 & q_1^\top k_2 & \cdots & q_1^\top k_n \end{bmatrix} / \sqrt{d_k}\right) \\ \text{softmax}\left(\begin{bmatrix} q_2^\top k_1 & q_2^\top k_2 & \cdots & q_2^\top k_n \end{bmatrix} / \sqrt{d_k}\right) \\ \vdots \\ \text{softmax}\left(\begin{bmatrix} q_n^\top k_1 & q_n^\top k_2 & \cdots & q_n^\top k_n \end{bmatrix} / \sqrt{d_k}\right) \end{bmatrix} \tag{9.5}$$

Comparing this self-attention matrix with the attention matrix described in Equation 9.2, we notice the only difference lies in the dimensions: since in self-attention, the query, key, and value all come from the same input, we have $n_q = n_k = n_v$, and we often denote all three with a unified n.

The self-attention output is then given by a weighted sum over the values:

$$y^{(i)} = \sum_{j=1}^{n} \alpha_{ij} v_j$$

This diagram below shows (only) the middle input token generating a query that is then combined with the keys computed with all tokens to generate the attention weights via a softmax. The output of the softmax is then combined with values computed from all tokens, to generate the attention output corresponding to the middle input token. Repeating this for each input token then generates the output.



Study Question: We have five colored tokens in the diagram above (gray, navy blue, orange, lime green, magenta). Could you read off the diagram the correspondence between the color and input, query, value, output?

Note that the size of the output is the same as the size of the input. Also, observe that there is no apparent notion of ordering of the input words in the depicted structure. Positional information can be added by encoding a number for token (giving say, the token's position relative to the start of the sequence) into the vector embedding of each token. And note that a given query need not pay attention to all other tokens in the input; in this example, the token used for the query is not used for a key or value.

More generally, a *mask* may be applied to limit which tokens are used in the attention computation. For example, one common mask limits the attention computation to tokens that occur previously in time to the one being used for the query. This prevents the attention mechanism from "looking ahead" in scenarios where the transformer is being used to generate one token at a time.

Each self-attention stage is trained to have key, value, and query embeddings that lead it to pay specific attention to some particular feature of the input. We generally want to pay attention to many different kinds of features in the input; for example, in translation one feature might be be the verbs, and another might be objects or subjects. A transformer utilizes multiple instances of self-attention, each known as an "attention head," to allow combinations of attention paid to many different features.

9.3 Transformers

A transformer is the composition of a number of transformer blocks, each of which has multiple attention heads. At a very high-level, the goal of a transformer block is to output

a really rich, useful representation for each input token, all for the sake of being high-performant for whatever task the model is trained to learn.

Rather than depicting the transformer graphically, it is worth returning to the beauty of the underlying equations[1].

9.3.1 Learned embedding

For simplicity, we assume the transformer internally uses self-attention. Full general attention layers work out similarly.

Formally, a transformer block is a parameterized function f_θ that maps $\mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d}$, where the input data $X \in \mathbb{R}^{n\times d}$ is often represented as a sequence of n tokens, with each token $x^{(i)} \in \mathbb{R}^{d\times 1}$.

Three projection matrices (weights) W_q, W_k, W_v are to be learned, such that, for each token $x^{(i)} \in \mathbb{R}^{d\times 1}$, we produce 3 distinct vectors: a query vector $q_i = W_q^T x^{(i)}$; a key vector $k_i = W_k^T x^{(i)}$; a value vector $v_i = W_v^T x^{(i)}$, all 3 of these vectors $\mathbb{R}^{d_k\times 1}$ and the learned weights $W_q, W_k, W_v \in \mathbb{R}^{d\times d_k}$.

If we stack these n query, key, value vectors into matrix- form, such that $Q \in \mathbb{R}^{n\times d_k}$, $K \in \mathbb{R}^{n\times d_k}$, and $V \in \mathbb{R}^{n\times d_k}$, then we can more compactly write out the learned transformation from the sequence of input token X:

$$\begin{aligned} Q &= XW_q \\ K &= XW_k \\ V &= XW_v \end{aligned}$$

These Q, K, V triple can then be used to produce one (self)attention-layer output. One such layer is called one "attention head".

One can have more than one "attention head", such that: the queries, keys, and values are embedded via encoding matrices:

$$Q^{(h)} = XW_{h,q} \tag{9.6}$$
$$K^{(h)} = XW_{h,k} \tag{9.7}$$
$$V^{(h)} = XW_{h,v} \tag{9.8}$$

and $W_{h,q}, W_{h,k}, W_{h,v} \in \mathbb{R}^{d\times d_k}$ where d_k is the size of the key/query embedding space, and $h \in \{1, \cdots, H\}$ is an index over "attention heads."

for each attention-head h, we learn one set of $W_{h,q}, W_{h,k}, W_{h,v}$.

We then perform a weighted sum over all the outputs for each head,

$$u'^{(i)} = \sum_{h=1}^{H} W_{h,c}^T \sum_{j=1}^{n} \alpha_{ij}^{(h)} V_j^{(h)}, \tag{9.9}$$

where $W_{h,c} \in \mathbb{R}^{d_k\times d}$, $u'^{(i)} \in \mathbb{R}^{d\times 1}$, the indices $i \in \{1, \cdots, n\}$ and $j \in \{1, \cdots, n\}$ are an integer index over tokens.

$V_j^{(h)}$ is the $d_k \times 1$ value embedding vector that corresponds to the input token x^j for attention head h.

This is then standardized and combined with $x^{(i)}$ using a LayerNorm function (defined below) to become

$$u^{(i)} = \text{LayerNorm}\left(x^{(i)} + u'^{(i)}; \gamma_1, \beta_1\right) \tag{9.10}$$

with parameters $\gamma_1, \beta_1 \in \mathbb{R}^d$.

[1] The presentation here follows the notes by John Thickstun.

To get the final output, we follow the "intermediate output then layer norm" recipe again. In particular, we first get the transformer block output $z'^{(i)}$ given by

$$z'^{(i)} = W_2^T \text{ReLU}\left(W_1^T u^{(i)}\right) \tag{9.11}$$

with weights $W_1 \in \mathbb{R}^{d\times m}$ and $W_2 \in \mathbb{R}^{m\times d}$. This is then standardized and combined with $u^{(i)}$ to give the final output $z^{(i)}$:

$$z^{(i)} = \text{LayerNorm}\left(u^{(i)} + z'^{(i)}; \gamma_2, \beta_2\right), \tag{9.12}$$

with parameters $\gamma_2, \beta_2 \in \mathbb{R}^d$. These vectors are then assembled (e.g., through parallel computation) to produce $z \in \mathbb{R}^{n\times d}$.

The LayerNorm function transforms a d-dimensional input z with parameters $\gamma, \beta \in \mathbb{R}^d$ into

$$\text{LayerNorm}(z; \gamma, \beta) = \gamma \frac{z - \mu_z}{\sigma_z} + \beta, \tag{9.13}$$

where μ_z is the mean and σ_z the standard deviation of z:

$$\mu_z = \frac{1}{d}\sum_{i=1}^{d} z_i \tag{9.14}$$

$$\sigma_z = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(z_i - \mu_z)^2}. \tag{9.15}$$

Layer normalization is done to improve convergence stability during training.

The model parameters comprise the weight matrices $W_{h,q}, W_{h,k}, W_{h,v}, W_{h,c}, W_1, W_2$ and the LayerNorm parameters $\gamma_1, \gamma_2, \beta_1, \beta_2$. A *transformer* is the composition of L transformer blocks, each with its own parameters:

$$f_{\theta_L} \circ \cdots \circ f_{\theta_2} \circ f_{\theta_1}(x) \in \mathbb{R}^{n\times d}. \tag{9.16}$$

The hyperparameters of this model are d, d_k, m, H, and L.

9.3.2 Variations and training

Many variants on this transformer structure exist. For example, the LayerNorm may be moved to other stages of the neural network. Or a more sophisticated attention function may be employed instead of the simple dot product used in Eq. 9.2. Transformers may also be used in pairs, for example, one to process the input and a separate one to generate the output given the transformed input. Self-attention may also be replaced with cross-attention, where some input data are used to generate queries and other input data generate keys and values. Positional encoding and masking are also common, though they are left implicit in the above equations for simplicity.

How are transformers trained? The number of parameters in θ can be very large; modern transformer models like GPT4 have tens of billions of parameters or more. A great deal of data is thus necessary to train such models, else the models may simply overfit small datasets.

Training large transformer models is thus generally done in two stages. A first "pre-training" stage employs a very large dataset to train the model to extract patterns. This is done with unsupervised (or self-supervised) learning and unlabelled data. For example, the well-known BERT model was pre-trained using sentences with words masked. The

model was trained to predict the masked words. BERT was also trained on sequences of sentences, where the model was trained to predict whether two sentences are likely to be contextually close together or not. The pre-training stage is generally very expensive.

The second "fine-tuning" stage trains the model for a specific task, such as classification or question answering. This training stage can be relatively inexpensive, but it generally requires labeled data.