Stanford MS&E 236 / CS 225: Lecture 16
Transformers

Ellen Vitercik*
vitercik@stanford.edu

June 12, 2024

In the final two classes, we will investigate a surprising phenomenon: large neural sequence models can perform *in-context learning* [e.g., 1, 2, 4]. Let T be a *trained* transformer. A test prompt for this transformer is defined by a set of "training points" $x_1, \ldots, x_N$ in some set $\mathcal{X}$, a function $f : \mathcal{X} \to \mathbb{R}$, and a test point $x' \in \mathcal{X}$. The test prompt has the following form:

$$P = \left[(x_1, f(x_1)), (x_2, f(x_2)), \ldots, (x_N, f(x_N)), x'\right].$$

The transformer has already been trained, so we will not retrain it on this test prompt (hence the adjective "test"). Instead, we want to have already trained T so that for any[1] such test prompt P,

$$T(P) \approx f(x'). \tag{1}$$

Intriguingly, the transformer hasn't learned a single function f, it seems to have learned an *algorithm*. For example, suppose f and f' are two linear functions that define two prompts

$$P = \left[(\boldsymbol{x}_1, f(\boldsymbol{x}_1)), (\boldsymbol{x}_2, f(\boldsymbol{x}_2)), \ldots, (\boldsymbol{x}_N, f(\boldsymbol{x}_N)), \boldsymbol{x}'\right]$$
$$P' = \left[(\boldsymbol{x}_1, f'(\boldsymbol{x}_1)), (\boldsymbol{x}_2, f'(\boldsymbol{x}_2)), \ldots, (\boldsymbol{x}_N, f'(\boldsymbol{x}_N)), \boldsymbol{x}'\right].$$

It is not that the transformer has learned a single weight vector $\boldsymbol{w}$ and outputs $T(P) = \boldsymbol{w}^T\boldsymbol{x}$ and $T(P') = \boldsymbol{w}^T\boldsymbol{x}'$. Clearly, Equation (1) would not be even close to satisfied for all prompts. Instead, the transformer seems to have learned a regression *algorithm*. Is it ridge regression, ordinary least-squares, or something else?

We will investigate this phenomenon further in the next class. We will start with an overview of the transformer architecture [5] (these notes are largely based on Chapter 12 of the textbook by Bishop and Bishop [3]).

1 Attention

The concept of *attention*—which we first encountered in Lecture 3—forms the basis of the transformer architecture, as we will illustrate through the following example. Consider the following two sentences:

*These notes are course material and have not undergone formal peer review. Please feel free to send me any typos or comments.

[1]This is likely too much to ask for, so we will discuss relaxations of this goal in the next class.

- He swung the **bat** and hit a foul.
- A **bat** flew out of the cave at dusk.

The word "bat" is a homonym with two different meanings in these two sentences. In the first sentence, the words "swung," "hit," and "foul" indicate that the sentence is about baseball. Meanwhile, in the second sentence, the words "flew" and "cave" indicate that the sentence is about an animal. Intuitively, to properly parse these sentences, a neural network should be able to figure out which words to rely on (i.e., "attend to") to interpret the context. This is all the more challenging because the positions of the important contextual words differ in each sentence.

Old-school word embeddings struggle to handle homonyms: "bat" would map to a single vector, which may not capture its range of meaning. Meanwhile, we can think of transformers as forming word embeddings that depend on the entire context of each word. For example, "bat" would be close to "baseball" in the embedding corresponding to the first sentence, whereas "bat" would be close to "nocturnal" in the second.

2 The transformer layer

The input to a transformer is a set of vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N \in \mathbb{R}^D$ (called "tokens"). We will use the matrix notation

$$X = \begin{pmatrix} \text{—} & \boldsymbol{x}_1 & \text{—} \\ & \vdots & \\ \text{—} & \boldsymbol{x}_N & \text{—} \end{pmatrix} \in \mathbb{R}^{N \times D}.$$

Our goal is to transform X to a matrix

$$\tilde{X} = \text{TransformerLayer}(X) = \begin{pmatrix} \text{—} & \boldsymbol{y}_1 & \text{—} \\ & \vdots & \\ \text{—} & \boldsymbol{y}_N & \text{—} \end{pmatrix} \in \mathbb{R}^{N \times D}$$

which captures the new, contextual embeddings of these inputs. As such, the new embedding $\boldsymbol{y}_n$ should depend on $\boldsymbol{x}_n$ as well as the entire input $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$. This dependence should be stronger for an input $\boldsymbol{x}_m$ if $\boldsymbol{x}_m$ is important for determining $\boldsymbol{y}_n$. A simple way to capture this dependence is to define

$$\boldsymbol{y}_n = \sum_{m=1}^{N} a_{nm} \boldsymbol{x}_m, \tag{2}$$

where each a_{nm} is the "attention weight." For simplicity, we enforce that $a_{nm} \geq 0$ and $\sum_{m=1}^{N} a_{nm} = 1$.

The next question we must answer is how to define the attention weights. The intuition comes from the field of information retrieval, which we motivate via an analogy. Suppose that a streaming video service's movies can be represented by a D-dimensional feature vector (corresponding to, for example, the movie's genre, main actors, length, etc.). The feature vector is called the movie's "key," whereas the movie itself is called the "value" (you might remember these terms from when you learned about hash tables in an introductory data structures class). Suppose the user provides a D-dimensional vector of attributes describing

the movie they want to see, called the user's "query." The streaming service could return the movie whose key is closest to the user's query. This would correspond to *hard attention* in the realm of neural networks. Meanwhile, *soft attention* is defined by continuous weights, which measure the similarity between the user's query and each movie's key. These weights then determine the likelihood of returning a given key's value.

In Equation (2), we can think of $\boldsymbol{x}_n$ as the query and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ as the keys. The similarity between $\boldsymbol{x}_n$ and $\boldsymbol{x}_m$ is measured as $\boldsymbol{x}_n^T \boldsymbol{x}_m$. Based on these weights, we define the likelihood of returning a given key's value using softmax:

$$a_{nm} = \frac{\exp\left(\boldsymbol{x}_n^T \boldsymbol{x}_m\right)}{\sum_{m'=1}^{N} \exp\left(\boldsymbol{x}_n^T \boldsymbol{x}_{m'}\right)}. \tag{3}$$

Finally, we can rewrite Equation (2) as

$$Y = \text{Softmax}\left(XX^T\right) X. \tag{4}$$

To be clear,

$$\text{Softmax}\left(XX^T\right) = \begin{pmatrix} a_{11} & \ldots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \ldots & a_{NN} \end{pmatrix},$$

where a_{nm} is defined as in Equation (3). Although Y defines a new D-dimensional embedding for each input, we cannot stop here because there are no learnable parameters!

2.1 Network parameters

To introduce learnable parameters, we will define separate query, key, and value matrices

$$\begin{aligned} Q &= XW^{(q)} \\ K &= XW^{(k)} \\ V &= XW^{(v)} \end{aligned}$$

where $W^{(q)}$, $W^{(k)}$, and $W^{(v)}$ are learnable weight matrices. We now generalize Equation (4) as follows: $Y = \text{Attention}(Q, K, V) = \text{Softmax}\left(QK^T\right) V$. It is important that the dimensions of the weight matrices are independent of the sequence length so that the final architecture can be applied to sequences of any length. As such, we set

$$W^{(q)} \in \mathbb{R}^{D \times D},$$

so $Q = XW^{(q)} \in \mathbb{R}^{N \times D}$. Similarly,

$$W^{(k)} \in \mathbb{R}^{D \times D},$$

so $K = XW^{(k)} \in \mathbb{R}^{N \times D}$. As a result, $QK^T \in \mathbb{R}^{N \times N}$, so $\text{Softmax}(QK^T) \in \mathbb{R}^{N \times N}$ as well. Finally, we set

$$W^{(v)} \in \mathbb{R}^{D \times D_v},$$

where D_v is our desired output dimension. This means that $V = XW^{(v)} \in \mathbb{R}^{N \times D_v}$, so $Y \in \mathbb{R}^{N \times D_v}$. (In full generality, we could have set $W^{(q)}, W^{(k)} \in \mathbb{R}^{D \times D_k}$ for some D_k, but oftentimes, $D = D_k$.)

2.2 Multi-head attention

The transformation $Y = \text{Attention}(Q, K, V)$ defines a single *attention head.* However, multiple attention patterns may be relevant at the same time. For example, some words may be relevant to determine tense, whereas others may be relevant to determine vocabulary. Therefore, we can compute multiple "attention heads" with different weight parameters. We denote these computations across M different heads as $H_h = \text{Attention}(Q_h, K_h, V_h)$ for $h \in [M]$. We concatenate these M outputs together, forming a matrix $\text{Concat}(H_1, \ldots, H_M) \in \mathbb{R}^{N \times MD_v}$. To compute a final $(N \times D)$-dimensional matrix, we use a learned matrix $W^{(o)} \in \mathbb{R}^{MD_v \times D}$, defining

$$Y(X) = \text{Concat}(H_1, \ldots, H_M)W^{(o)} \in \mathbb{R}^{N \times D}.$$

2.3 Transformer layers

There are two more improvements we can include in the final architecture. First, it may be useful to include residual layers so that some elements of X are directly retained in the output. We can achieve this by adding X to the output $Y(X)$, i.e., $Y(X) + X$. Moreover, $Y(X)$ is primarily computed using linear transformations, with a few non-linearities in the form of the softmax from Equation (3). Therefore, we might like to include additional non-linearities. To do so, we run *each row* of $Y(X) + X$ through a standard, feed-forward neural network with D inputs and D outputs. We denote this computation as $\text{MLP}(\cdot)$, where MLP stands for *multi-layer perceptron.* Thus, we finally define

$$\tilde{X} = \text{TransformerLayer}(X) = \text{MLP}(Y(X) + X).$$

This is the basic building block of the transformer architecture. Many of these transformer layers are stacked upon each other to build the transformer architecture.

3 Positional encoding

Interestingly, the transformer architecture is *equivariant* with respect to input permutations. To see why, note that to compute $Q = XW^{(q)}$, each row of X (i.e., token) is multiplied by the same matrix $W^{(q)}$. The same holds when compute $K = XW^{(k)}$ and $V = XW^{(v)}$. Moreover, each row of $Y(X) + X$ is run through the same MLP. Therefore, if we permute the input sequence $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$, the output will be the same, except permuted in the same way. However, this property is not always desirable, as illustrated by the following two sentences:

- The food was bad, not good at all.
- The food was good, not bad at all.

These sentences have the exact same words, but when permuted, they have very different meanings. To break permutation equivariance, it is common to use a positional encoding vector $\boldsymbol{r}_n$ and redefine the input tokens to be $\tilde{\boldsymbol{x}}_n = \boldsymbol{x}_n + \boldsymbol{r}_n$.

As a first idea, we might associate each position with its index $1, 2, 3, \ldots$. However, this may limit generalization to longer inputs than the transformer was trained on since it will encounter positional embeddings it was never trained on. Alternatively, we might associate

each position with a number in $(0, 1)$. However, each position's encoding will depend on the entire sequence length, which will differ across sequences; we'd rather the first token, for example, have the same positional encoding across sequences.

To address these issues, there are two common approaches. First, we can make $\boldsymbol{r}_n$ a learnable weight vector, breaking permutation equivariance. Another common approach is to use a sinusoidal encoding of each position—reminiscent of binary encodings—with $\boldsymbol{r}_n \in (-1, 1)^D$. Check out Section 12.1.9 of the textbook linked to on the course webpage for more details [3].

References

[1] Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. What learning algorithm is in-context learning? investigations with linear models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

[2] Yu Bai, Fan Chen, Huan Wang, Caiming Xiong, and Song Mei. Transformers as statisticians: Provable in-context learning with in-context algorithm selection. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[3] Christopher M. Bishop and Hugh Bishop. *Deep learning: foundations and concepts*. Springer Cham, 2023.

[4] Shivam Garg, Dimitris Tsipras, Percy S Liang, and Gregory Valiant. What can transformers learn in-context? a case study of simple function classes. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Conference on Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.