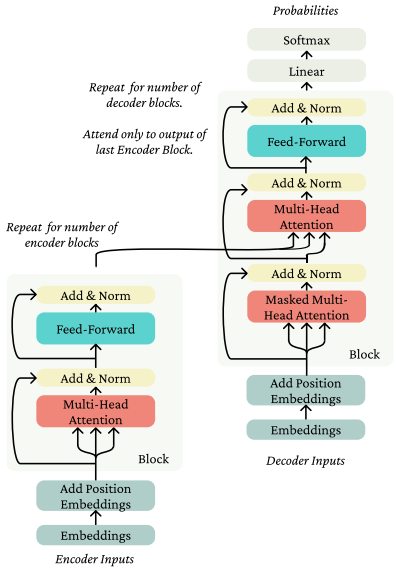


Transformer Encoder-Decoder

This note was originally written for Stanford CS 224n by John Hewitt.

Neural architectures and their properties

Progress in natural language processing is often accelerated by general-purpose techniques that perform better than earlier methods across a wide range of settings. Sometimes, a new idea is proposed to solve old problems; other times, old techniques become newly relevant as data or computation power becomes newly available. A few examples of these include hidden Markov models [Baum and Petrie, 1966], conditional random fields [Lafferty et al., 2001], recurrent neural networks [Rumelhart et al., 1985], convolutional neural networks [LeCun et al., 1989], and support vector machines [Cortes and Vapnik, 1995].

In this section, we'll discuss a bit about the neural modeling approaches we've discussed in Cs 224n so far, and how their limitations (and changes in the world) inspired the modern (as of 2023) zeitgeist of self-attention and Transformer-based architectures.

Notation and basics

Let $\mathbf{w}_{1:n}$ be a sequence, where each $\mathbf{w}_i \in \mathcal{V}$, a finite vocabulary. We'll also overload $\mathbf{w}_{1:n}$ to be a matrix of one-hot vectors, $\mathbf{w}_{1:n} \in \mathbb{R}^{n \times |\mathcal{V}|}$. We'll use $\mathbf{w} \in \mathcal{V}$ to represent an arbitrary vocabulary element, and $\mathbf{w}_i \in \mathcal{V}$ to pick out a specific indexed element of a sequence $\mathbf{w}_{1:n}$. We'll use the notation,

$$\mathbf{w}_t \sim \text{softmax}(f(\mathbf{w}_{1:t-1})), \tag{1}$$

to mean that under a model, w_t "is drawn from" the probability distribution defined by the right-hand-side of the tilde, $\sim$. So in this case, $f(\mathbf{w}_{1:t-1})$ should be in $\mathbb{R}^{|\mathcal{V}|}$. When we use the *softmax* function (as above), we'll use it without direct reference to the dimension being normalized over, and it should be interpreted as follows. If A is a

tensor of shape $\mathbb{R}^{\ell,d}$, the softmax is computed as follows:

$$\text{softmax}(A)_{i,j} = \frac{\exp A_{i,j}}{\sum_{j'=1}^{d} \exp A_{i,j'}}, \tag{2}$$

for all $i \in 1, \ldots, \ell$, $j \in 1, \ldots, d$, and similarly for tensors of more than two axes. That is, if we had a tensor $B \in \mathbb{R}^{m,\ell,d}$, we would define the softmax over the last dimension, similarly. At the risk of being verbose, we'll write it out:

$$\text{softmax}(B)_{q,i,j} = \frac{\exp B_{q,i,j}}{\sum_{j'=1}^{d} \exp B_{q,i,j'}}. \tag{3}$$

In all of our methods, we'll assume an *embedding* matrix, $E \in \mathbb{R}^{d \times |\mathcal{V}|}$, mapping from the vocabulary space to the *hidden dimensionality* d, written as $E\mathbf{x} \in \mathbb{R}^d$.

Embedding definition

The embedding $E\mathbf{w}_i$ of a token in sequence $\mathbf{w}_{1:n}$ is what's known as a **non-contextual** representation; despite $\mathbf{w}_i$ appearing in a sequence, the representation $E\mathbf{w}_i$ is independent of context. Since we'll almost always be working on the embedded version of $\mathbf{w}_{1:n}$, we'll let $\mathbf{x} = E\mathbf{w}$, and $\mathbf{x}_{1:n} = \mathbf{w}_{1:n}E^\top \in \mathbb{R}^{n \times d}$. An overarching goal of the methods discussed in this note is to develop strong **contextual** representations of tokens; that is, a representation $\mathbf{h}_i$ that represents $\mathbf{w}_i$ but is a function of the entire sequence $\mathbf{x}_{1:n}$ (or a prefix $\mathbf{x}_{1:i}$, as in the case of language modeling.).

A non-contextual representation of a token $\mathbf{x}_i$ of sequence $\mathbf{x}_{1:n}$ depends only on the identity of $\mathbf{x}_i$; a contextual representation of $\mathbf{x}_i$ depends on the entire sequence (or a prefix $\mathbf{x}_{1:i}$.)

The default circa 2017: recurrent neural networks

General-purpose modeling techniques and representations have a long history in NLP, with individual techniques falling in and out of favor. Word embeddings, for example, have a much longer history than the word2vec embeddings we studied in the first few lectures [Schütze, 1992]. Likewise, recurrent neural networks have a long and non-monotonic history in modeling problems [Elman, 1990, Bengio et al., 2000]. By 2017, however, the basic strategy to solve a natural language processing task was to begin with a recurrent neural network.

We've gone over RNNs earlier in the course, but the general form bears repeating here. A simple form of RNN is as follows:

Dependence on the sequence index

$$\mathbf{h}_t = \sigma(W \mathbf{h}_{t-1} + U\mathbf{x}_t), \tag{4}$$

where $\mathbf{h}_t \in \mathbb{R}^d$, $U \in \mathbb{R}^{d \times d}$, and $W \in \mathbb{R}^{d \times d}$. By 2017, the intuition was that there were twofold issues with the recurrent neural network form, and they both had to do with the the depenence on the sequence index (often called the dependence on "time") highlighted in Equation 4.

Parallelization issues with dependence on the sequence index. Modern graphics processing units (GPUs) are excellent at crunching through a lot of simple operations (like addition) in *parallel*. For example, when I have a matrix $A \in \mathbb{R}^{n \times k}$ and a matrix $B \in \mathbb{R}^{k \times d}$, a GPU is just blazing fast at computing $AB \in \mathbb{R}^{n \times d}$. The constraint of the operations occuring in parallel, however, is crucial – when computing AB the simplest way, I'm performing a bunch of multiplies and then a bunch of sums, most of which don't depend on the output of each other. However, in a recurrent neural network, when I compute

$$\mathbf{h}_2 = \sigma(W\mathbf{h}_1 + U\mathbf{x}_2), \tag{5}$$

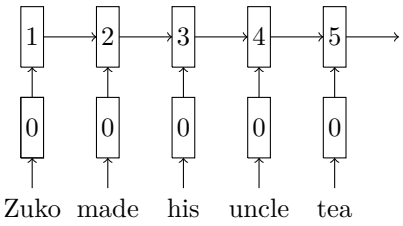


Figure 1: A RNN unrolled in time. The rectangles are intermediate states of the RNN (e.g., the first row is the embedding layer, and the second row is the RNN hidden state at each time step) and the number in the rectangle is the number of serial operations that need to be performed before this intermediate state can be computed

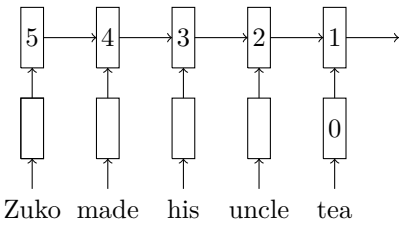


Figure 2: A RNN unrolled in time. The rectangles are intermediate states of the RNN (e.g., the first row is the embedding layer, and the second row is the RNN hidden state at each time step) and the number in the rectangle is roughly the number of operations separating lexical information of the word *tea* from each intermediate state.

I can't compute h_2 until I know the value of $\mathbf{h}_1$, so we can write it out as

$$\mathbf{h}_2 = \sigma(W\sigma(W\mathbf{h}_0 + U\mathbf{x}_1) + U\mathbf{x}_2). \tag{6}$$

Likewise if I wanted to compute $\mathbf{h}_3$, I can't compute it until I know $\mathbf{h}_2$, which I can't compute until I know $\mathbf{h}_1$, etc. Visually, this looks like Figure 1. As the sequence gets longer, **there is only so much I can parallelize the computation of the network on a GPU** because of the number of serial dependencies. (Serial meaning one-after-the-other.)

As GPUs (and later, other accelerators like Tensor Processing Units (TPUs) became more powerful and researchers wanted to take fuller advantage of them, this dependence in time became untenable.

Linear interaction distance. A related issue with RNNs is the difficulty with which distant tokens in a sequence can *interact* with each other. By interact, we mean that the presence of one token (already observed in the past) gainfully affects the processing of another token. For example, in the sentence

> The chef_1 who ran out of blackberries and went to the stores is_1

the number of intermediate computations—matrix multiplies and nonlinearities, for example—that separate *chef* from *is* scales with the number of words between them. We visualize this in Figure 2.

Intuitively, researchers believe there's an issue with linear interaction distance because it can be difficult for networks to precisely "recall" the presence of a word when a large

number of operations occur after observing that word. This can make it difficult to learn how distant words should impact the representation of the current word.

This notion of **direct interaction between elements** of a sequence might remind you of the attention mechanism [Bahdanau et al., 2014] in machine translation. In that context, while generating a translation, we learned how to look back into the source sequence once per token of the translation. In this note, we'll present an entire replacement for recurrent neural networks just based on attention. This will solve both the parallelization issues and the linear interaction distance issues with recurrent neural networks.

A minimal self-attention architecture

Attention, broadly construed, is a method for taking a query, and softly looking up information in a key-value store by picking the value(s) of the key(s) most like the query. By "picking" and "most like," we mean averaging overall values, putting more weight on those which correspond to the keys more like the query. In **self-attention**, we mean that we use the same elements to help us define the querys as we do the keys and values.

In this section, we'll discuss how to develop contextual representations with methods wherein the main mechanism for contextualization is not recurrence, but attention.

The key-query-value self-attention mechanism

There are many forms of self-attention; the form we'll discuss here is currently the most popular. It's called key-query-value self-attention.

Consider a token $\mathbf{x}_i$ in the sequence $\mathbf{x}_{1:n}$. From it, we define a query $\mathbf{q}_i = Q\mathbf{x}_i$, for matrix $Q \in \mathbb{R}^{d\times d}$. Then, for each token in the sequence $\mathbf{x}_j \in \{x_1 \ldots, x_n\}$, we define both a key and a value similarly, with two other weight matrices: $\mathbf{k}_j = K\mathbf{x}_j$, and $\mathbf{v}_j = V\mathbf{x}_j$ for $K \in \mathbb{R}^{d\times d}$ and $V \in \mathbb{R}^{d\times d}$.

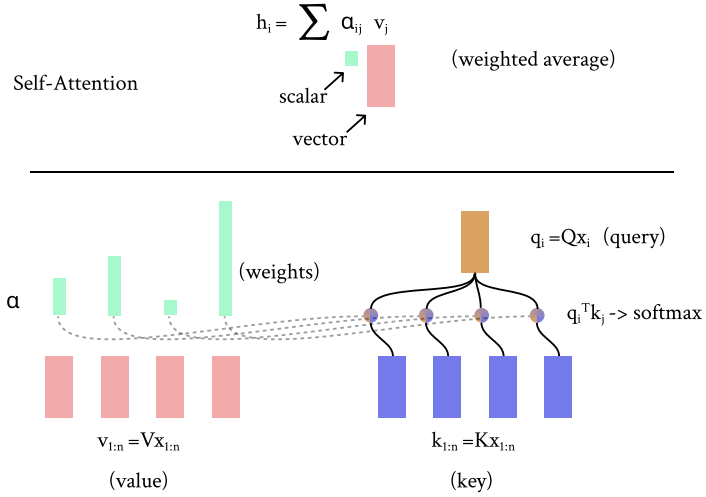
Our contextual representation $\mathbf{h}_i$ of $\mathbf{x}_i$ is a linear combination (that is, a weighted sum) of the values of the sequence,

$$\mathbf{h}_i = \sum_{j=1}^{n} \alpha_{ij}\mathbf{v}_j, \tag{7}$$

where the weights, these α_{ij} control the strength of contribution of each $\mathbf{v}_j$. Going back to our key-value store analogy, the α_{ij} softly selects what data to look up. We define these weights by computing the affinities between the keys and the query, $q_i^\top k_j$, and then computing the softmax over the sequence:

$$\alpha_{ij} = \frac{\exp(\mathbf{q}_i^\top \mathbf{k}_j)}{\sum_{j'=1}^{n} \exp(\mathbf{q}_i^\top \mathbf{k}_{j'})} \tag{8}$$

Intuitively, what we've done by this operation is take our element $\mathbf{x}_i$ and look in its own sequence $\mathbf{x}_{1:n}$ to figure out what information (in an informal sense,) from what other tokens, should be used in representing $\mathbf{x}_i$ in context. The use of matrices K, Q, V intuitively allow us to use different views of the $\mathbf{x}_i$ for the different roles of key, query, and value. We perform this operation to build $\mathbf{h}_i$ for all $i \in \{1, \ldots, n\}$.



Position representations

Consider the sequence *the oven cooked the bread so*. This is a different sequence than *the bread cooked the oven so*, as you might guess. The former sentence has us making delicious bread, and the latter we might interpret as the bread somehow breaking the oven. In a recurrent neural network, the order of the sequence defines the order of the rollout, so the two sequences have different representations. In the self-attention operation, there's no built-in notion of order.

The self-attention operation has **no built-in notion of the sequence order.**

To see this, let's take a look at self-attention on this sequence. We have a set of vectors $\mathbf{x}_{1:n}$ for *the oven cooked the bread so*, which we can write as

$$\mathbf{x}_{1:n} = [\mathbf{x}_{\text{the}}; \mathbf{x}_{\text{oven}}; \mathbf{x}_{\text{cooked}}; \mathbf{x}_{\text{the}}; \mathbf{x}_{\text{bread}}; \mathbf{x}_{\text{so}}] \in \mathbb{R}^{5 \times d} \tag{9}$$

As an example, consider performing self-attention to represent the word *so* in context. The weights over the context are as follows, recalling that $\mathbf{q}_i = Q\mathbf{x}_i$ for all words, and $\mathbf{k}_i = K\mathbf{x}_i$ likewise:

$$\alpha_{\text{so}} = \text{softmax}\left(\left[\mathbf{q}_{\text{so}}^\top \mathbf{k}_{\text{the}}; \mathbf{q}_{\text{so}}^\top \mathbf{k}_{\text{oven}}; \mathbf{q}_{\text{so}}^\top \mathbf{k}_{\text{cooked}}; \mathbf{q}_{\text{so}}^\top \mathbf{k}_{\text{the}}; \mathbf{q}_{\text{so}}^\top \mathbf{k}_{\text{bread}}; \mathbf{q}_{\text{so}}^\top \mathbf{k}_{\text{so}}\right]\right) \tag{10}$$

So, the weight $\alpha_{\text{so},0}$, the amount that we look up the first word, (by writing out the softmax) is,

$$\alpha_{\text{so},0} = \frac{\exp(\mathbf{q}_{\text{so}}^\top \mathbf{k}_{\text{the}})}{\exp(\mathbf{q}_{\text{so}}^\top \mathbf{k}_{\text{the}}) + \cdots + \exp(\mathbf{q}_{\text{so}}^\top \mathbf{k}_{\text{bread}})}. \tag{11}$$

So, $\alpha \in \mathbb{R}^5$ are our weights, and we compute the weighted average in Equation 7 with these weights to compute $\mathbf{h}_{\text{so}}$.

For the reordered sentence *the bread cooked the oven*, note that $\alpha_{\text{so},0}$ is identical. The numerator hasn't changed, and the denominator hasn't changed; we've just rearranged terms in the sum. Likewise for $\alpha_{\text{so,bread}}$ and $\alpha_{\text{so,oven}}$, you can compute that they too are identical independent of the ordering of the sequence. This all comes back down to the two facts that (1) the representation of $\mathbf{x}$ is not position-dependent; it's just $E\mathbf{w}$ for whatever word $\mathbf{w}$, and (2) there's no dependence on position in the self-attention operations.

Non-contextual embedded words $\mathbf{x}_i = E\mathbf{w}_i$ have no dependence on the word's position in a sequence $\mathbf{w}_{1:n}$; only on the identity of the word in $\mathcal{V}$.

Position representation through learned embeddings. To represent position in self-attention, you either need to (1) use vectors that are already position-dependent as inputs, or (2) change the self-attention operation itself. One common solution is a simple implementation of (1). We posit a new parameter matrix, $P \in \mathbb{R}^{N \times d}$, where N is the *maximum length of any sequence* that your model will be able to process.

We then simply add embedded representation of the position of a word to its word embedding:

$$\tilde{x}_i = P_i + \mathbf{x}_i \tag{12}$$

and perform self-attention as we otherwise would. Now, the self-attention operation can use the embedding P_i to look at the word at position i differently than if that word were at position j. This is done, e.g., in the BERT paper [Devlin et al., 2019] (which we go over later in the course.)

Position representation through changing α directly. Instead of changing the input representation, another thing we can do is change the form of self-attention to have a built-in notion of position. One intuition is that all else held equal, self-attention should look at "nearby" words more than "far" words. Attention with Linear Biases [Press et al., 2022] is one implementation of this idea. One implementation of this would be as follows:

$$\alpha_i = \text{softmax}\left(\mathbf{k}_{1:n}\mathbf{q}_i + [-i, \ldots, -1, 0, -1, \ldots, -(n-i)]\right), \tag{13}$$

where $\mathbf{k}_{1:n}\mathbf{q}_i \in \mathbb{R}^n$ are the original attention scores, and the bias we add makes attention focus more on nearby words than far away words, all else held equal. In some sense, it's odd that this works; but interesting!

Elementwise nonlinearity

Imagine if we were to stack self-attention layers. Would this be sufficient for a replacement for stacked LSTM layers? Intuitively, there's one thing that's missing: the elementwise nonlinearities that we've come to expect in standard deep learning architectures. In fact, if we stack two self-attention layers, we get something that looks a lot like a single self-attention layer:

$$o_i = \sum_{j=1}^{n} \alpha_{ij} V^{(2)} \left(\sum_{k=1}^{n} \alpha_{jk} V^{(1)} \mathbf{x}_k \right) \tag{14}$$

$$= \sum_{k=1}^{n} \left(\alpha_{jk} \sum_{j=1}^{n} \alpha_{ij} \right) V^{(2)} V^{(1)} \mathbf{x}_k \tag{15}$$

$$= \sum_{k=1}^{n} \alpha^*_{ij} V^* \mathbf{x}_k, \tag{16}$$

where $\alpha^*_{ij} = \left(\alpha_{jk} \sum_{j=1}^{n} \alpha_{ij} \right)$, and $V^* = V^{(2)} V^{(1)}$. So, this is just a linear combination of a linear transformation of the input, much like a single layer of self-attention! Is this good enough?[1]

[1] This question ends up having a nuanced answer that's out-of-scope for this note; ask me if you're interested in knowing more!

In practice, after a layer of self-attention, it's common to apply feed-forward network independently to each word representation:

$$h_{\text{FF}} = W_2 \text{ ReLU}(W_1 h_{\text{self-attention}} + b_1) + b_2, \tag{17}$$

where often, $W_1 \in \mathbb{R}^{5d \times d}$, and $W_2 \in \mathbb{R}^{d \times 5d}$. That is, the hidden dimension of the feed-forward network is substantially larger than the hidden dimension of the network, d—this is done because this matrix multiply is an efficiently parallelizable operation, so it's an efficient place to put a lot of computation and parameters.

Future masking

When performing language modeling like we've seen in this course (often called autoregressive modeling), we predict a word given all words so far:

$$\mathbf{w}_t \sim \text{softmax}(f(\mathbf{w}_{1:t-1})). \tag{18}$$

where f is function to map a sequence to a vector in $\mathbb{R}^{|\mathcal{V}|}$.

One crucial aspect of this process is that we can't look at the future when predicting it—otherwise the problem becomes trivial. This idea is built-in to unidirectional RNNs. If we want to use an RNN for the function f, we can use the hidden state for word w_{t-1}:

$$\mathbf{w}_t \sim \text{softmax}(\mathbf{h}_{t-1} E) \tag{19}$$

$$h_{t-1} = \sigma(W \mathbf{h}_{t-2} + U \mathbf{x}_{t-1}), \tag{20}$$

and by the rollout of the RNN, we haven't looked at the future. (In this case, the future is all the words $\mathbf{w}_t, \ldots, \mathbf{w}_n$.)

In a Transformer, there's nothing explicit in the self-attention weight α that says not to look at indices $j > i$ when representing token i. In practice, we enforce this constraint simply adding a large negative constant to the input to the softmax (or equivalently, setting $\alpha_{ij} = 0$ where $j > i$.)[2]

$$\alpha_{ij,\text{masked}} = \begin{cases} \alpha_{ij} & j \leq i \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

In a diagram, it looks like Figure 3.

Summary of a minimal self-attention architecture

Our minimal self-attention architecture has (1) the self-attention operation, (2) position representations, (3) elementwise nonlinearities, and (4) future masking (in the context of language modeling.)

Intuitively, these are the biggest components to understand. However, as of 2023, by far the most-used architecture in NLP is called the *Transformer*, introduced by [Vaswani et al., 2017], and it contains a number of components that end up being quite important. So now we'll get into the details of that architecture.

[2]It might seem like one should use $-\infty$ as the constant, to "really" ensure that you can't see the future. However, this is not done; a modest constant within even the float range of the 'float16' encoding is used instead, like -10^5. Using infinity can lead to NaNs and it's sort of undefined how each library should treat infinite inputs, so we tend to avoid using it. And because of finite precision, a large enough negative constant will still set the attention weight to exactly zero.

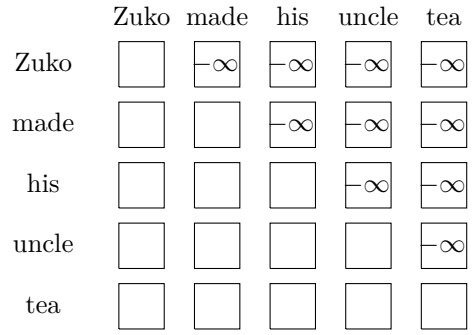


Figure 3: Diagram of autoregressive future masking in self-attention. Words in each row have words in the future masked out (e.g., "Zuko" can only attend to "Zuko", while "made" can attend to "Zuko" and "made".)

The Transformer

The Transformer is an architecture based on self-attention that consists of stacked *Blocks*, each of which contains self-attention and feed-forward layers, and a few other components we'll discuss. If you'd like to take a peek for intuition, we have a diagram of a Transformer language model architecture in Figure 4. The components we haven't gone over are **multi-head** self-attention, **layer normalization**, **residual connections**, and **attention scaling**—and of course, we'll discuss how these components are combined to form the Transformer.

Multi-head Self-Attention

Intuitively, a single call of self-attention is best at picking out a *single* value (on average) from the input value set. It does so softly, by averaging over all of the values, but it requires a balancing game in the key-query dot products in order to carefully average two or more things. In Assignment 5, you'll work through a bit of this intuition more carefully. What we'll present now, **multi-head self-attention**, intuitively applies self-attention multiple times at once, each with different key, query, and value transformations of the same input, and then combines the outputs.

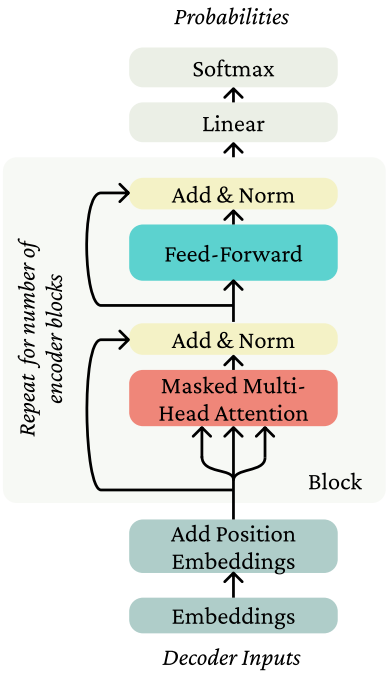
For an integer number of heads k, we define matrices $K^{(\ell)}, Q^{(\ell)}, V^{(\ell)} \in \mathbb{R}^{d \times d/k}$ for ℓ in $\{1, \ldots, k\}$. (We'll see why we have the dimensionality reduction to d/k soon.) These our the key, query, and value matrices for each head. Correspondingly, we get keys, queries, and values $\mathbf{k}_{1:n}^{(\ell)}, \mathbf{q}_{1:n}^{(\ell)}, \mathbf{v}_{1:n}^{(\ell)}$, as in single-head self-attention.

We then perform self-attention with each head:

$$\mathbf{h}_i^{(\ell)} = \sum_{j=1}^{n} \alpha_{ij}^{(\ell)} \mathbf{v}_j^{(\ell)} \tag{22}$$

$$\alpha_{ij}^{(\ell)} = \frac{\exp(\mathbf{q}_i^{(\ell)\top} \mathbf{k}_j^{(\ell)})}{\sum_{j'=1}^{n} \exp(\mathbf{q}_i^{(\ell)\top} \mathbf{k}_{j'}^{(\ell)})} \tag{23}$$

Note that the output $\mathbf{h}_i^{(\ell)}$ of each head is in reduced dimension d/k. Finally, we define the output of multi-head self-attention as a linear transformation of the concatenation



Transformer Decoder

Figure 4: Diagram of the Transformer Decoder (without corresponding Encoder, and so no cross-attention.

of the head outputs, letting $O \in \mathbb{R}^{d \times d}$:

$$\mathbf{h}_i = O\left[\mathbf{v}_i^{(1)}; \cdots; \mathbf{v}_i^{(k)}\right], \tag{24}$$

where we concatenate the head outputs each of dimensionality $d \times d/k$ at their second axis, such that their concatenation has dimension $d \times d$.

Sequence-tensor form. To understand why we have the reduced dimension of each head output, it's instructive to get a bit closer to how multi-head self-attention is implemented in code. In practice, **multi-head self-attention is no more expensive than single-head** due to the low-rankness of the transformations we apply.

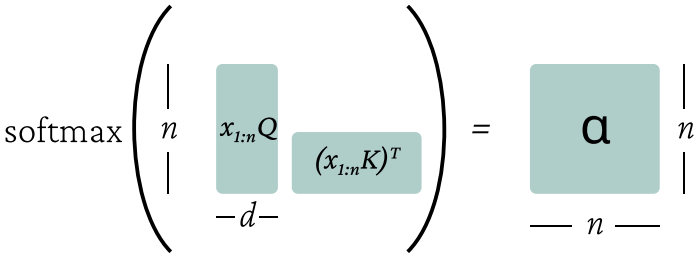
For a single head, recall that $\mathbf{x}_{1:n}$ is a matrix in $\mathbb{R}^{n \times d}$. Then we can compute our value vectors as a matrix as $\mathbf{x}_{1:n}V$, and likewise our keys and queries $\mathbf{x}_{1:n}K$ and $\mathbf{x}_{1:n}Q$, all matrices in $\mathbb{R}^{n \times d}$. To compute self-attention, we can compute our weights in matrix operations:

$$\alpha = \text{softmax}(\mathbf{x}_{1:n}QK^\top \mathbf{x}_{1:n}^\top) \in \mathbb{R}^{n \times n} \tag{25}$$

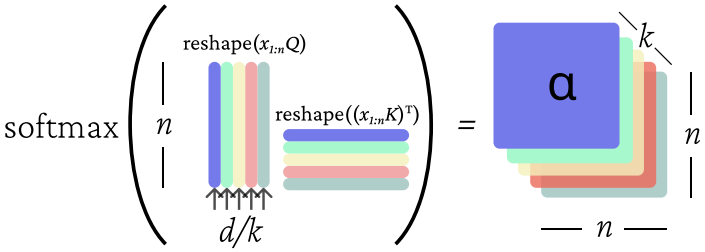
and then compute the self-attention operation for all $\mathbf{x}_{1:n}$ via:

$$\mathbf{h}_{1:n} = \text{softmax}(\mathbf{x}_{1:n}QK^\top \mathbf{x}_{1:n}^\top)\mathbf{x}_{1:n}V \in \mathbb{R}^{n \times d}. \tag{26}$$

Here's a diagram showing the matrix ops:



When we perform multi-head self-attention in this matrix form, we first reshape $\mathbf{x}_{1:n}Q$, $\mathbf{x}_{1:n}K$, and $\mathbf{x}_{1:n}V$ each into a matrix of shape $\mathbb{R}^{n,k,d/k}$, splitting the model dimensionality into two axes, for the number of heads and the number of dimensions per head. We can then transpose the matrices to $\mathbb{R}^{k,n,d/k}$, which intuitively should look like k sequences of length n and dimensionality d/k. This allows us to perform the batched softmax operation in parallel across the heads, using the number of heads kind of like a **batch** axis (and indeed in practice we'll also have a separate batch axis.) So, the total computation (except the last linear transformation to combine the heads) is the same, just distributed across the (each lower-rank) heads. Here's a diagram like the single-head diagram, demonstrating how the multi-head operation ends up much like the single-head operation:



Layer Norm

One important learning aid in Transformers is *layer normalization* [Ba et al., 2016]. The intuition of layer norm is to reduce uninformative variation in the activations at a layer, providing a more stable input to the next layer. Further work shows that this may be most useful not in normalizing the forward pass, but actually in improving gradients in the backward pass [Xu et al., 2019].

To do this, layer norm (1) computes statistics across the activations at a layer to estimate the mean and variance of the activations, and (2) normalizes the activations with respect to those estimates, while (3) optionally learning (as parameters) an elementwise additive bias and multiplicative gain by which to sort of de-normalize the activations in a predictable way. The third part seems not to be crucial, and may even be harmful [Xu et al., 2019], so we omit it in our presentation.

One question to ask when understanding how layer norm affects a network is, "computing statistics over *what*?" That is, what constitutes a layer? In Transformers, the answer is always that statistics computed independently for a single index into the sequence length (and a single example in the batch) and shared across the d hidden dimensions. Put another way, the statistics for the token at index i won't affect the token at index $j \neq i$.

So, we compute the statistics for a single index $i \in \{1, \ldots, n\}$ as

$$\hat{\mu}_i = \frac{1}{d}\sum_{j=1}^{d} \mathbf{h}_{ij} \qquad \hat{\sigma}_i = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(\mathbf{h}_{ij} - \mu_i)^2}, \tag{27}$$

where (as a reminder), $\hat{\mu}_i$ and $\hat{\sigma}_i$ are scalars, and we compute the layer norm as

$$\text{LN}(\mathbf{h}_i) = \frac{\mathbf{h}_i - \hat{\mu}_i}{\hat{\sigma}_i}, \tag{28}$$

where we've broadcasted the $\hat{\mu}_i$ and $\hat{\sigma}_i$ across the d dimensions of $\mathbf{h}_i$. Layer normalization is a great tool to have in your deep learning toolbox more generally.

Residual Connections

Residual connections simply add the *input* of a layer to the *output* of that layer:

$$f_{\text{residual}}(\mathbf{h}_{1:n}) = f(\mathbf{h}_{1:n}) + \mathbf{h}_{1:n}, \tag{29}$$

the intuition being that (1) the gradient flow of the identity function is *great* (the local gradient is 1 everywhere!) so the connection allows for learning much deeper networks, and (2) it is easier to learn the difference of a function from the identity function than it is to learn the function from scratch. As simple as these seem, they're massively useful in deep learning, not just in Transformers!

Add & Norm. In the Transformer diagrams you'll see, including Figure 4, the application of layer normalization and residual connection are often combined in a single visual block labeled *Add & Norm*. Such a layer might look like:

$$\mathbf{h}_{\text{pre-norm}} = f(\text{LN}(\mathbf{h})) + \mathbf{h}, \tag{30}$$

where f is either a feed-forward operation or a self-attention operation, (this is known as *pre-normalization*), or like:

$$\mathbf{h}_{\text{post-norm}} = \text{LN}(f(\mathbf{h}) + \mathbf{h}), \tag{31}$$

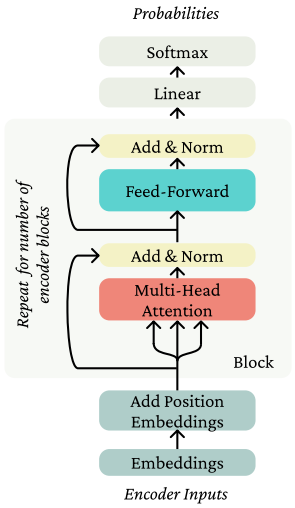


Figure 5: Diagram of the Transformer Encoder.

which is known as *post-normalization*. It turns out that the gradients of **pre-normalization** are much better at initialization, leading to much faster training [Xiong et al., 2020].

Attention logit scaling

Another trick introduced in [Vaswani et al., 2017] they dub *scaled dot product attention*. The dot product part comes from the fact that we're computing dot products $\mathbf{q}_i^\top k_j$. The intuition of scaling is that, when the dimensionality d of the vectors we're dotting grows large, the dot product of even random vectors (e.g., at initialization) grows roughly as $\sqrt{d}$. So, we normalize the dot products by $\sqrt{d}$ to stop this scaling:

$$\alpha = \text{softmax}(\frac{\mathbf{x}_{1:n} Q K^\top \mathbf{x}_{1:n}^\top}{\sqrt{d}}) \in \mathbb{R}^{n \times n} \tag{32}$$

Transformer Encoder

A Transformer Encoder takes a single sequence $\mathbf{w}_{1:n}$, and performs no future masking. It embeds the sequence with E to make $\mathbf{x}_{1:n}$, adds the position representation, and then applies a stack of independently parameterized *Encoder Blocks*, each of which consisting of (1) multi-head attention and Add & Norm, and (2) feed-forward and Add & Norm. So, the output of each Block is the input to the next. Figure 5 presents this. In the case that one wants probabilities out of the tokens of a Transformer Encoder (as in masked language modeling for BERT [Devlin et al., 2019], which we'll cover later), one applies a linear transformation to the output space followed by a softmax.

Uses of the Transformer Encoder. A Transformer Encoder is great in contexts where you aren't trying to generate text autoregressively (there's no masking in the encoder so each position index can see the whole sequence,) and want strong representations for the whole sequence (again, possible because even the first token can see the whole future of the sequence when building its representation.)

Transformer Decoder

To build a Transformer autoregressive language model, one uses a Transformer Decoder. These differ from Transformer Encoders simply by using future masking at each application of self-attention. This ensures that the informational constraint (no cheating by looking at the future!) holds throughout the architecture. We show a diagram of this architecture in Figure 4. Famous examples of this are GPT-2 [Radford et al., 2019], GPT-3 [Brown et al., 2020] and BLOOM [Workshop et al., 2022].

Transformer Encoder-Decoder

A Transformer encoder-decoder takes as input two sequences. Figure 6 shows the whole encoder-decoder structure. The first sequence $\mathbf{x}_{1:n}$ is passed through a Transformer Encoder to build contextual representations. The second sequence $\mathbf{y}_{1:m}$ is encoded through a modified Transformer Decoder architecture in which **cross-attention** (which we haven't yet defined!) is applied from the encoded representation of $\mathbf{y}_{1:m}$ to the output of the Encoder. So, let's take a quick detour to discuss cross-attention; it's not too different from what we've already seen.

Cross-Attention. Cross-attention uses one sequence to define the keys and values of self-attention, and another sequence to define the queries. You might think, hey wait, isn't that just what attention always was before we got into this self-attention business? Yeah, pretty much. So if

$$\mathbf{h}^{(x)}_{1:n} = \text{TransformerEncoder}(\mathbf{w}_{1:n}), \tag{33}$$

and we have some intermediate representation $\mathbf{h}^{(y)}$ of sequence $\mathbf{y}_{1:m}$, then we let the queries come from the decoder (the $\mathbf{h}^{(y)}$ sequence) while the keys and values come from the encoder:

$$\mathbf{q}_i = Q\mathbf{h}^{(y)}_i \qquad i \in \{1, \ldots, m\} \tag{34}$$

$$\mathbf{k}_j = K\mathbf{h}^{(x)}_j \qquad j \in \{1, \ldots, n\} \tag{35}$$

$$\mathbf{v}_j = V\mathbf{h}^{(x)}_j \qquad j \in \{1, \ldots, n\}, \tag{36}$$

and compute the attention on $\mathbf{q}, \mathbf{k}, \mathbf{v}$ as we defined for self-attention. Note in Figure 6 that in the Transformer Encoder-Decoder, cross-attention always applies to the output of the Transformer encoder.

Uses of the encoder-decoder. An encoder-decoder is used when we'd like bidirectional context on something (like an article to summarize) to build strong represenations (i.e., each token can attend to all other tokens), but then generate an output according to an autoregressive decomposition as we can with a decoder. While such an architecture has been found to provide better performance than decoder-only models at modest scale [Raffel et al., 2020], it involves splitting parameters between encoder and decoder, and most of the largest Transformers are decoder-only.

References

[Ba et al., 2016] Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

[Bahdanau et al., 2014] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

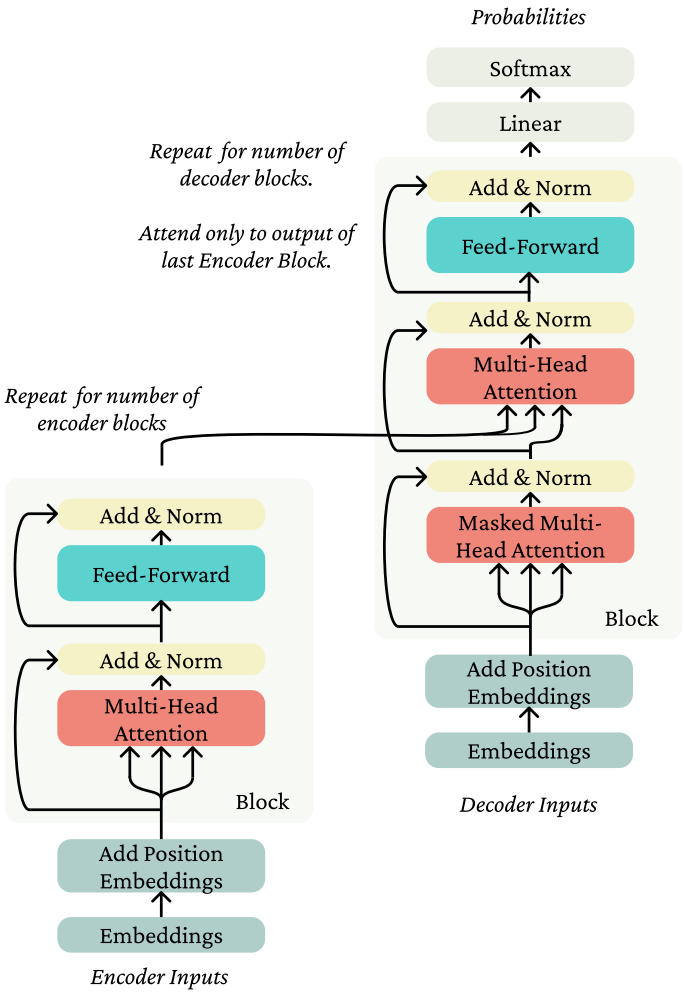


Figure 6: A Transformer encoder-decoder.

[Baum and Petrie, 1966] Baum, L. E. and Petrie, T. (1966). Statistical inference for probabilistic functions of finite state markov chains. *The Annals of Mathematical Statistics*, 37(6):1554–1563.

[Bengio et al., 2000] Bengio, Y., Ducharme, R., and Vincent, P. (2000). A neural probabilistic language model. *Advances in neural information processing systems*, 13.

[Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. (2003). A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155.

[Brown et al., 2020] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

[Collobert et al., 2011] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. P. (2011). Natural language processing (almost) from scratch. *CoRR*, abs/1103.0398.

[Cortes and Vapnik, 1995] Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297.

[Devlin et al., 2019] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

[Elman, 1990] Elman, J. L. (1990). Finding structure in time. *Cognitive science*, 14(2):179–211.

[Fukushima and Miyake, 1982] Fukushima, K. and Miyake, S. (1982). Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer.

[Lafferty et al., 2001] Lafferty, J. D., McCallum, A., and Pereira, F. C. N. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, page 282–289, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[LeCun et al., 1989] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551.

[Manning, 2022] Manning, C. D. (2022). Human Language Understanding & Reasoning. *Daedalus*, 151(2):127–138.

[Mikolov et al., 2013] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781.

[Press et al., 2022] Press, O., Smith, N., and Lewis, M. (2022). Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations*.

[Radford et al., 2019] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners.

[Raffel et al., 2020] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551.

[Rong, 2014] Rong, X. (2014). word2vec parameter learning explained. *CoRR*, abs/1411.2738.

[Rumelhart et al., 1985] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.

[Rumelhart et al., 1988] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA.

[Schütze, 1992] Schütze, H. (1992). Dimensions of meaning. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, page 787–796, Washington, DC, USA. IEEE Computer Society Press.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

[Workshop et al., 2022] Workshop, B., :, Scao, T. L., Fan, A., Akiki, C., Pavlick, E., Ilić, S., Hesslow, D., Castagné, R., Luccioni, A. S., Yvon, F., Gallé, M., Tow, J., Rush, A. M., Biderman, S., Webson, A., Ammanamanchi, P. S., Wang, T., Sagot, B., Muennighoff, N., del Moral, A. V., Ruwase, O., Bawden, R., Bekman, S., McMillan-Major, A., Beltagy, I., Nguyen, H., Saulnier, L., Tan, S., Suarez, P. O., Sanh, V., Laurençon, H., Jernite, Y., Launay, J., Mitchell, M., Raffel, C., Gokaslan, A., Simhi, A., Soroa, A., Aji, A. F., Alfassy, A., Rogers, A., Nitzav, A. K., Xu, C., Mou, C., Emezue, C., Klamm, C., Leong, C., van Strien, D., Adelani, D. I., Radev, D., Ponferrada, E. G., Levkovizh, E., Kim, E., Natan, E. B., De Toni, F., Dupont, G., Kruszewski, G., Pistilli, G., Elsahar, H., Benyamina, H., Tran, H., Yu, I., Abdulmumin, I., Johnson, I., Gonzalez-Dios, I., de la Rosa, J., Chim, J., Dodge, J., Zhu, J., Chang, J., Frohberg, J., Tobing, J., Bhattacharjee, J., Almubarak, K., Chen, K., Lo, K., Von Werra, L., Weber, L., Phan, L., allal, L. B., Tanguy, L., Dey, M., Muñoz, M. R., Masoud, M., Grandury, M., Šaško, M., Huang, M., Coavoux, M., Singh, M., Jiang, M. T.-J., Vu, M. C., Jauhar, M. A., Ghaleb, M., Subramani, N., Kassner, N., Khamis, N., Nguyen, O., Espejel, O., de Gibert, O., Villegas, P., Henderson, P., Colombo, P., Amuok, P., Lhoest, Q., Harliman, R., Bommasani, R., López, R. L., Ribeiro, R., Osei, S., Pyysalo, S., Nagel, S., Bose, S., Muhammad, S. H., Sharma, S., Longpre, S., Nikpoor, S., Silberberg, S., Pai, S., Zink, S., Torrent, T. T., Schick, T., Thrush, T., Danchev, V., Nikoulina, V., Laippala, V., Lepercq, V., Prabhu, V., Alyafeai, Z., Talat, Z., Raja, A., Heinzerling, B., Si, C., Taşar, D. E., Salesky, E., Mielke, S. J., Lee, W. Y., Sharma, A., Santilli, A., Chaffin, A., Stiegler, A., Datta, D., Szczechla, E., Chhablani, G., Wang, H., Pandey, H., Strobelt, H., Fries, J. A., Rozen, J., Gao, L., Sutawika, L., Bari, M. S., Al-shaibani, M. S., Manica, M., Nayak, N., Teehan, R., Albanie, S., Shen, S., Ben-David, S., Bach, S. H., Kim, T., Bers, T., Fevry, T., Neeraj, T., Thakker, U., Raunak, V., Tang, X., Yong, Z.-X., Sun, Z., Brody, S., Uri, Y., Tojarieh, H., Roberts, A., Chung, H. W., Tae, J., Phang, J., Press, O., Li, C., Narayanan, D., Bourfoune, H., Casper, J., Rasley, J., Ryabinin, M., Mishra, M., Zhang, M., Shoeybi, M., Peyrounette, M., Patry, N., Tazi, N., Sanseviero, O., von Platen, P., Cornette, P., Lavallée, P. F., Lacroix, R., Rajbhandari, S., Gandhi, S., Smith, S., Requena, S., Patil, S., Dettmers, T., Baruwa, A., Singh, A., Cheveleva, A., Ligozat, A.-L., Subramonian, A., Névéol, A., Lovering, C., Garrette, D., Tunuguntla, D., Reiter, E., Taktasheva, E., Voloshina, E., Bogdanov, E., Winata, G. I., Schoelkopf, H., Kalo, J.-C., Novikova, J., Forde, J. Z., Clive, J., Kasai, J., Kawamura, K., Hazan, L., Carpuat, M., Clinciu, M., Kim, N., Cheng, N., Serikov, O., Antverg, O., van der Wal, O., Zhang, R., Zhang, R., Gehrmann, S., Mirkin, S., Pais, S., Shavrina, T., Scialom, T., Yun, T., Limisiewicz, T., Rieser, V., Protasov, V., Mikhailov, V., Pruksachatkun, Y., Belinkov, Y., Bamberger, Z., Kasner, Z., Rueda, A., Pestana, A., Feizpour, A., Khan, A., Faranak, A., Santos, A., Hevia, A., Unldreaj, A., Aghagol, A., Abdollahi, A., Tammour, A., HajiHosseini, A., Behroozi, B., Ajibade, B., Saxena, B., Ferrandis, C. M., Contractor, D., Lansky, D., David, D., Kiela, D., Nguyen, D. A., Tan, E., Baylor, E., Ozoani, E., Mirza, F., Ononiwu, F., Rezanejad, H., Jones, H., Bhattacharya, I., Solaiman, I., Sedenko, I., Nejadgholi, I., Passmore, J., Seltzer, J., Sanz, J. B., Dutra, L., Samagaio, M., Elbadri, M., Mieskes, M., Gerchick, M., Akinlolu, M., McKenna, M., Qiu, M., Ghauri, M., Burynok, M., Abrar, N., Rajani, N., Elkott, N., Fahmy, N., Samuel, O., An, R., Kromann, R., Hao, R., Alizadeh, S., Shubber, S., Wang, S., Roy, S., Viguier, S., Le, T., Oyebade, T., Le, T., Yang, Y., Nguyen, Z., Kashyap, A. R., Palasciano, A., Callahan, A., Shukla, A., Miranda-Escalada, A., Singh, A., Beilharz, B., Wang, B., Brito, C., Zhou, C., Jain, C., Xu, C., Fourrier, C., Periñán, D. L., Molano, D., Yu, D., Manjavacas, E., Barth, F., Fuhrimann, F., Altay, G., Bayrak, G., Burns, G., Vrabec, H. U., Bello, I., Dash, I., Kang, J., Giorgi, J., Golde, J., Posada, J. D., Sivaraman, K. R., Bulchandani, L., Liu, L., Shinzato, L., de Bykhovetz, M. H., Takeuchi, M., Pàmies, M., Castillo, M. A., Nezhurina, M., Sänger, M., Samwald, M., Cullan, M., Weinberg, M., De Wolf, M., Mihaljcic, M., Liu, M., Freidank, M., Kang, M., Seelam, N., Dahlberg, N., Broad, N. M., Muellner, N., Fung, P., Haller, P., Chandrasekhar, R., Eisenberg, R., Martin, R., Canalli, R., Su, R., Su, R., Cahyawijaya, S., Garda, S., Deshmukh, S. S., Mishra, S., Kiblawi, S., Ott, S., Sang-aroonsiri, S., Kumar, S., Schweter, S., Bharati, S., Laud, T., Gigant, T., Kainuma, T., Kusa, W., Labrak, Y., Bajaj, Y. S., Venkatraman, Y., Xu, Y., Xu, Y., Xu, Y., Tan, Z., Xie, Z., Ye, Z., Bras, M., Belkada, Y., and Wolf, T. (2022). Bloom: A 176b-parameter open-access multilingual language model.

[Xiong et al., 2020] Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., and Liu, T. (2020). On layer normalization in the transformer architecture. In *International Conference on Machine Learning*, pages 10524–10533. PMLR.

[Xu et al., 2019] Xu, J., Sun, X., Zhang, Z., Zhao, G., and Lin, J. (2019). Understanding and improving layer normalization. *Advances in Neural Information Processing Systems*, 32.