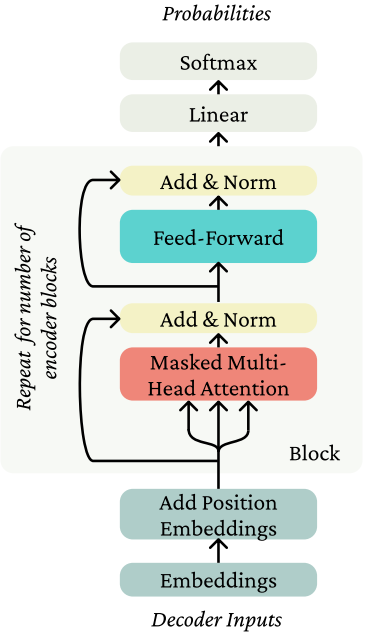
3 *The Transformer*

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.

3.1 *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.



Transformer Decoder

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

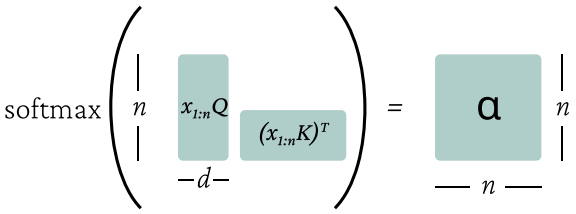
For a single head, recall that $\mathbf{x}_{1:n}$ is a matrix in $\mathbb{R}^{n \times d}$. Then we can compute our value vectors as a matrix as $\mathbf{x}_{1:n} V$, and likewise our keys and queries $\mathbf{x}_{1:n} K$ and $\mathbf{x}_{1:n} Q$, all matrices in $\mathbb{R}^{n \times d}$. To compute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

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

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

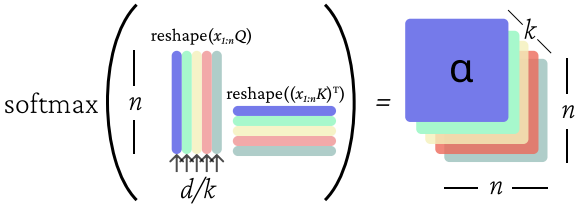
$$\mathbf{h}_{1:n} = \text{softmax}(\mathbf{x}_{1:n} Q K^\top \mathbf{x}_{1:n}^\top) \mathbf{x}_{1:n} V \in \mathbb{R}^{n \times d}. \tag{26}$$

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:



3.2 *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.

3.3 *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:

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

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].

3.4 *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:

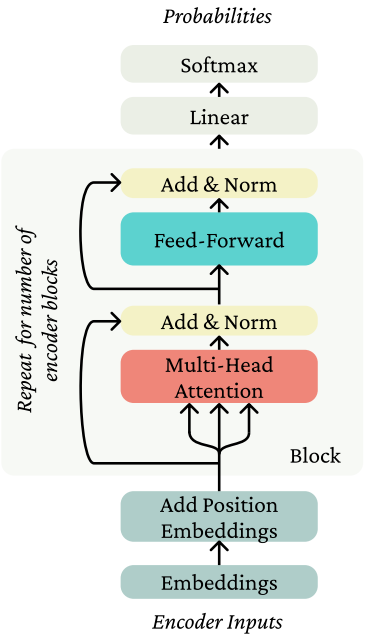
$$\alpha = \text{softmax}(\frac{\mathbf{x}_{1:n} Q K^\top \mathbf{x}_{1:n}^\top}{\sqrt{d}}) \in \mathbb{R}^{n \times n} \tag{32}$$

3.5 *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 Encoder

Figure 5: Diagram of the Transformer Encoder.

3.6 *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].

3.7 *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.