CHAPTER

10 Transformers and Large Language Models

"How much do we know at any time? Much more, or so I believe, than we know we know."

Agatha Christie, *The Moving Finger*

Fluent speakers of a language bring an enormous amount of knowledge to bear during comprehension and production. This knowledge is embodied in many forms, perhaps most obviously in the vocabulary, the rich representations we have of words and their meanings and usage. This makes the vocabulary a useful lens to explore the acquisition of knowledge from text, by both people and machines.

Estimates of the size of adult vocabularies vary widely both within and across languages. For example, estimates of the vocabulary size of young adult speakers of American English range from 30,000 to 100,000 depending on the resources used to make the estimate and the definition of what it means to know a word. What is agreed upon is that the vast majority of words that mature speakers use in their day-to-day interactions are acquired early in life through spoken interactions with care givers and peers, usually well before the start of formal schooling. This active vocabulary is extremely limited compared to the size of the adult vocabulary (usually on the order of 2000 words for young speakers) and is quite stable, with very few additional words learned via casual conversation beyond this early stage. Obviously, this leaves a very large number of words to be acquired by other means.

A simple consequence of these facts is that children have to learn about 7 to 10 words a day, *every single day*, to arrive at observed vocabulary levels by the time they are 20 years of age. And indeed empirical estimates of vocabulary growth in late elementary through high school are consistent with this rate. How do children achieve this rate of vocabulary growth? Most of this growth is not happening through direct vocabulary instruction in school, which is not deployed at the rate that would be required to result in sufficient vocabulary growth.

The most likely explanation is that the bulk of this knowledge acquisition happens as a by-product of reading, as part of the rich processing and reasoning that we perform when we read. Research into the average amount of time children spend reading, and the lexical diversity of the texts they read, indicate that it is possible to achieve the desired rate. But the mechanism behind this rate of learning must be remarkable indeed, since at some points during learning the rate of vocabulary growth exceeds the rate at which new words are appearing to the learner!

Many of these facts have motivated approaches to word learning based on the *distributional hypothesis*, introduced in Chapter 6. This is the idea that something about what we're loosely calling word meanings can be learned even without any grounding in the real world, solely based on the content of the texts we encounter over our lives. This knowledge is based on the complex association of words with the words they co-occur with (and with the words that those words occur with).

The crucial insight of the distributional hypothesis is that the knowledge that we acquire through this process can be brought to bear long after its initial acquisition.

Of course, adding grounding from vision or from real-world interaction can help build even more powerful models, but even text alone is remarkably useful.

pretraining

In this chapter we formalize this idea of **pretraining**—learning knowledge about language and the world from vast amounts of text—and call the resulting pretrained language models **large language models**. Large language models exhibit remarkable performance on all sorts of natural language tasks because of the knowledge they learn in pretraining, and they will play a role throughout the rest of this book. They have been especially transformative for tasks where we need to produce text, like summarization, machine translation, question answering, or chatbots.

transformer

The standard architecture for building large language models is the **transformer**. We thus begin this chapter by introducing this architecture in detail. The transformer makes use of a novel mechanism called **self-attention**, which developed out of the idea of **attention** that was introduced for RNNs in Chapter 9. Self-attention can be thought of a way to build contextual representations of a word's meaning that integrate information from surrounding words, helping the model learn how words relate to each other over large spans of text.

We'll then see how to apply the transformer to language modeling, in a setting often called causal or autoregressive language models, in which we iteratively predict words left-to-right from earlier words. These language models, like the feedforward and RNN language models we have already seen, are thus self-trained: given a large corpus of text, we iteratively teach the model to guess the next word in the text from the prior words. In addition to training, we'll introduce algorithms for generating texts, including important methods like **greedy decoding**, **beam search**, and **sampling**. And we'll talk about the components of popular large language models like the GPT family.

Finally, we'll see the great power of language models: almost any NLP task can be modeled as word prediction, if we think about it in the right way. We'll work through an example of using large language models to solve one NLP task of **summarization** (generating a short text that summarizes some larger document). The use of a large language model to generate text is one of the areas in which the impact of the last decade of neural algorithms for NLP has been the largest. Indeed, text generation, along with image generation and code generation, constitute a new area of AI that is often called **generative AI**.

We'll save three more areas of large language models for the next three chapters; Chapter 11 will introduce the **bidirectional transformer** encoder and the method of **masked language modeling**, used for the popular BERT family of models. Chapter 12 will introduce the most powerful way to interact with large language models: **prompting** them to perform other NLP tasks by simply giving directions or instructions in natural language to a transformer that is pretrained on language modeling. And Chapter 13 will introduce the use of the encoder-decoder architecture for transformers in the context of machine translation.

10.1 The Transformer: A Self-Attention Network

transformer

In this section we introduce the architecture of the **transformer**, the algorithm that underlies most modern NLP systems. When used for causal language modeling, the input to a transformer is a sequence of words, and the output is a prediction for what word comes next, as well as a sequence of contextual embedding that represents the contextual meaning of each of the input words. Like the LSTMs of Chapter 9,

transformers are a neural architecture that can handle distant information. But unlike LSTMs, transformers are not based on recurrent connections (which can be hard to parallelize), which means that transformers can be more efficient to implement at scale.

Transformers are made up of stacks of transformer **blocks**, each of which is a multilayer network that maps sequences of input vectors $(\mathbf{x}_1, ..., \mathbf{x}_n)$ to sequences of output vectors $(\mathbf{z}_1, ..., \mathbf{z}_n)$ of the same length. These blocks are made by combining simple linear layers, feedforward networks, and **self-attention** layers, the key innovation of transformers. Self-attention allows a network to directly extract and use information from arbitrarily large contexts. We'll start by describing how self-attention works and then return to how it fits into larger transformer blocks. Finally, we'll describe how to use the transformer block together with some input and output mechanisms as a language model, to predict upcoming words from prior words in the context.

self-attention

10.1.1 Transformers: the intuition

The intuition of a transformer is that across a series of layers, we build up richer and richer contextualized representations of the meanings of input words or tokens (we will refer to the input as a sequence of words for convenience, although technically the input is first tokenized by an algorithm like BPE, so it is a series of tokens rather than words). At each layer of a transformer, to compute the representation of a word i we combine information from the representation of i at the previous layer with information from the representations of the neighboring words. The goal is to produce a contextualized representation for each word at each position. We can think of these representations as a contextualized version of the static vectors we saw in Chapter 6, which each represented the meaning of a word type. By contrast, our goal in transformers is to produce a contextualized version, something that represents what this word means in the particular context in which it occurs.

We thus need a mechanism that tells us how to weigh and combine the representations of the different words from the context at the prior level in order to compute our representation at this layer. This mechanism must be able to look broadly in the context, since words have rich linguistic relationships with words that can be many sentences away. Even within the sentence, words have important linguistic relationships with contextual words. Consider these examples, each exhibiting linguistic relationships that we'll discuss in more depth in later chapters:

(10.1) The **keys** to the cabinet **are** on the table.

(10.2) **The chicken** crossed the road because **it** wanted to get to the other side.

(10.3) I walked along the **pond**, and noticed that one of the trees along the **bank** had fallen into the **water** after the storm.

In (10.1), the phrase *The keys* is the subject of the sentence, and in English and many languages, must agree in grammatical number with the verb *are*; in this case both are plural. In English we can't use a singular verb like *is* with a plural subject like *keys*; we'll discuss agreement more in Chapter 17. In (10.2), the pronoun *it* corefers to the chicken; it's the chicken that wants to get to the other side. We'll discuss coreference more in Chapter 26. In (10.3), the way we know that *bank* refers to the side of a pond or river and not a financial institution is from the context, including words like *pond* and *water*. We'll discuss word senses more in Chapter 23. These helpful contextual words can be quite far way in the sentence or paragraph,

so we need a mechanism that can look broadly in the context to help compute representations for words.

Self-attention is just such a mechanism: it allows us to look broadly in the context and tells us how to integrate the representation from words in that context from layer $k-1$ to build the representation for words in layer k.

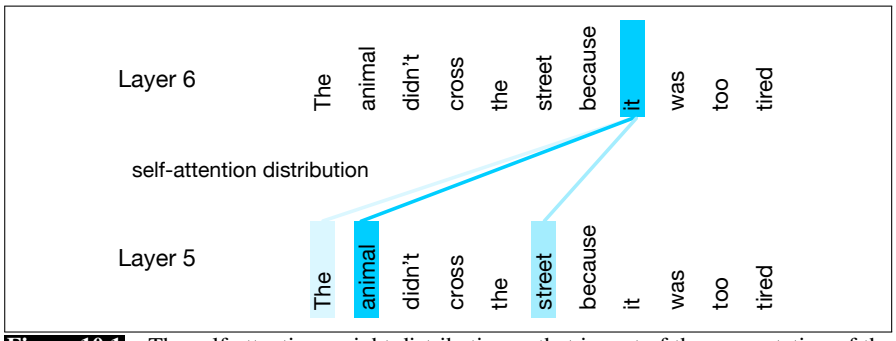

Figure 10.1 The self-attention weight distribution α that is part of the computation of the representation for the word *it* at layer 6. In computing the representation for *it*, we attend differently to the various words at layer 5, with darker shades indicating higher self-attention values. Note that the transformer is attending highly to *animal*, a sensible result, since in this example *it* corefers with the animal, and so we'd like the representation for *it* to draw on the representation for *animal*. Figure simplified from (Uszkoreit, 2017).

Fig. 10.1 shows an schematic example simplified from a real transformer (Uszkoreit, 2017). Here we want to compute a contextual representation for the word *it*, at layer 6 of the transformer, and we'd like that representation to draw on the representations of all the prior words, from layer 5. The figure uses color to represent the attention distribution over the contextual words: the word *animal* has a high attention weight, meaning that as we are computing the representation for *it*, we will draw most heavily on the representation for *animal*. This will be useful for the model to build a representation that has the correct meaning for *it*, which indeed is coreferent here with the word *animal*. (We say that a pronoun like *it* is coreferent with a noun like *animal* if they both refer to the same thing; we'll return to coreference in Chapter 26.)

10.1.2 Causal or backward-looking self-attention

The concept of *context* can be used in two ways in self-attention. In causal, or backward looking self-attention, the context is any of the prior words. In general bidirectional self-attention, the context can include future words. In this chapter we focus on causal, backward looking self-attention; we'll introduce bidirectional self-attention in Chapter 11.

Fig. 10.2 thus illustrates the flow of information in a single causal, or backward looking, self-attention layer. As with the overall transformer, a self-attention layer maps input sequences $(\mathbf{x}_1, ..., \mathbf{x}_n)$ to output sequences of the same length $(\mathbf{a}_1, ..., \mathbf{a}_n)$. When processing each item in the input, the model has access to all of the inputs up to and including the one under consideration, but no access to information about inputs beyond the current one. In addition, the computation performed for each item is independent of all the other computations. The first point ensures that we can use this approach to create language models and use them for autoregressive generation, and the second point means that we can easily parallelize both forward inference and training of such models.

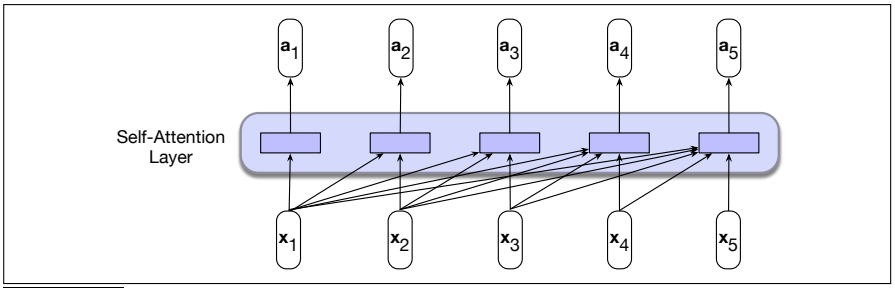

Figure 10.2 Information flow in a causal (or masked) self-attention model. In processing each element of the sequence, the model attends to all the inputs up to, and including, the current one. Unlike RNNs, the computations at each time step are independent of all the other steps and therefore can be performed in parallel.

10.1.3 Self-attention more formally

We've given the intuition of self-attention (as a way to compute representations of a word at a given layer by integrating information from words at the previous layer) and we've defined context as all the prior words in the input. Let's now introduce the self-attention computation itself.

The core intuition of attention is the idea of *comparing* an item of interest to a collection of other items in a way that reveals their relevance in the current context. In the case of self-attention for language, the set of comparisons are to other words (or tokens) within a given sequence. The result of these comparisons is then used to compute an output sequence for the current input sequence. For example, returning to Fig. 10.2, the computation of $\mathbf{a}_3$ is based on a set of comparisons between the input $\mathbf{x}_3$ and its preceding elements $\mathbf{x}_1$ and $\mathbf{x}_2$, and to $\mathbf{x}_3$ itself.

How shall we compare words to other words? Since our representations for words are vectors, we'll make use of our old friend the **dot product** that we used for computing word similarity in Chapter 6, and also played a role in attention in Chapter 9. Let's refer to the result of this comparison between words i and j as a score (we'll be updating this equation to add attention to the computation of this score):

$$\textit{Verson 1:} \quad \text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j \tag{10.4}$$

The result of a dot product is a scalar value ranging from $-\infty$ to ∞, the larger the value the more similar the vectors that are being compared. Continuing with our example, the first step in computing y_3 would be to compute three scores: $\mathbf{x}_3 \cdot \mathbf{x}_1$, $\mathbf{x}_3 \cdot \mathbf{x}_2$ and $\mathbf{x}_3 \cdot \mathbf{x}_3$. Then to make effective use of these scores, we'll normalize them with a softmax to create a vector of weights, α_{ij}, that indicates the proportional relevance of each input to the input element i that is the current focus of attention.

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \;\; \forall j \leq i \tag{10.5}$$

$$= \frac{\exp(\text{score}(\mathbf{x}_i, \mathbf{x}_j))}{\sum_{k=1}^{i} \exp(\text{score}(\mathbf{x}_i, \mathbf{x}_k))} \;\; \forall j \leq i \tag{10.6}$$

Of course, the softmax weight will likely be highest for the current focus element i, since $vecx_i$ is very similar to itself, resulting in a high dot product. But other context words may also be similar to i, and the softmax will also assign some weight to those words.

Given the proportional scores in α, we generate an output value $\mathbf{a}_i$ by summing

the inputs seen so far, each weighted by its α value.

$$\mathbf{a}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j \tag{10.7}$$

The steps embodied in Equations 10.4 through 10.7 represent the core of an attention-based approach: a set of comparisons to relevant items in some context, a normalization of those scores to provide a probability distribution, followed by a weighted sum using this distribution. The output $\mathbf{a}$ is the result of this straightforward computation over the inputs.

This kind of simple attention can be useful, and indeed we saw in Chapter 9 how to use this simple idea of attention for LSTM-based encoder-decoder models for machine translation. But transformers allow us to create a more sophisticated way of representing how words can contribute to the representation of longer inputs. Consider the three different roles that each input embedding plays during the course of the attention process.

- As *the current focus of attention* when being compared to all of the other preceding inputs. We'll refer to this role as a **query**.
- In its role as *a preceding input* being compared to the current focus of attention. We'll refer to this role as a **key**.
- And finally, as a **value** used to compute the output for the current focus of attention.

query

key

value

To capture these three different roles, transformers introduce weight matrices $\mathbf{W^Q}$, $\mathbf{W^K}$, and $\mathbf{W^V}$. These weights will be used to project each input vector $\mathbf{x}_i$ into a representation of its role as a key, query, or value.

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W^Q}; \quad \mathbf{k}_i = \mathbf{x}_i \mathbf{W^K}; \quad \mathbf{v}_i = \mathbf{x}_i \mathbf{W^V} \tag{10.8}$$

The inputs $\mathbf{x}$ and outputs $\mathbf{y}$ of transformers, as well as the intermediate vectors after the various layers like the attention output vector $\mathbf{a}$, all have the same dimensionality $1 \times d$. We'll have a dimension d_k for the key and query vectors, and a separate dimension d_v for the value vectors. In the original transformer work (Vaswani et al., 2017), d was 512, d_k and d_v were both 64. The shapes of the transform matrices are then $\mathbf{W}^Q \in \mathbb{R}^{d \times d_k}$, $\mathbf{W}^K \in \mathbb{R}^{d \times d_k}$, and $\mathbf{W}^V \in \mathbb{R}^{d \times d_v}$.

Given these projections, the score between a current focus of attention, $\mathbf{x}_i$, and an element in the preceding context, $\mathbf{x}_j$, consists of a dot product between its query vector $\mathbf{q}_i$ and the preceding element's key vectors $\mathbf{k}_j$. This dot product has the right shape since both the query and the key are of dimensionality $1 \times d_k$. Let's update our previous comparison calculation to reflect this, replacing Eq. 10.4 with Eq. 10.9:

$$\textit{Verson 2:} \quad \text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{q}_i \cdot \mathbf{k}_j \tag{10.9}$$

The ensuing softmax calculation resulting in $\alpha_{i,j}$ remains the same, but the output calculation for $\mathbf{a}_i$ is now based on a weighted sum over the value vectors $\mathbf{v}$.

$$\mathbf{a}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{v}_j \tag{10.10}$$

Again, the softmax weight α_{ij} will likely be highest for the current focus element i, and so the value for $\mathbf{y}_i$ will be most influenced by $\mathbf{v}_i$. But the model will also pay attention to other contextual words if they are similar to i, allowing their values to

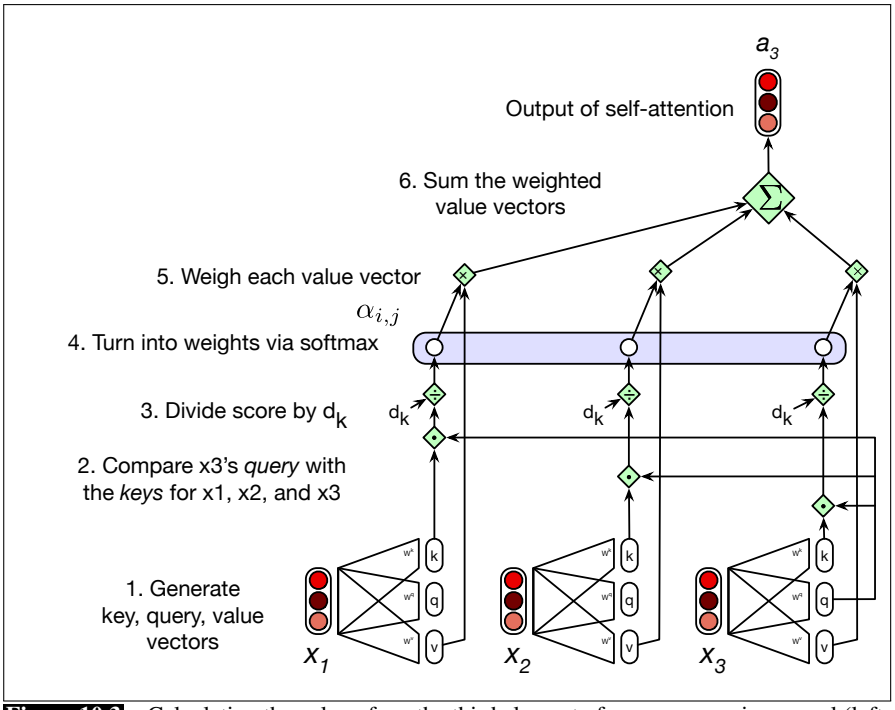

Figure 10.3 Calculating the value of $\mathbf{a}_3$, the third element of a sequence using causal (left-to-right) self-attention.

also influence the final value of $\mathbf{v}_j$. Context words that are not similar to i will have their values downweighted and won't contribute to the final value.

There is one final part of the self-attention model. The result of a dot product can be an arbitrarily large (positive or negative) value. Exponentiating large values can lead to numerical issues and to an effective loss of gradients during training. To avoid this, we scale down the result of the dot product, by dividing it by a factor related to the size of the embeddings. A typical approach is to divide by the square root of the dimensionality of the query and key vectors (d_k), leading us to update our scoring function one more time, replacing Eq. 10.4 and Eq. 10.9 with Eq. 10.12. Here's a final set of equations for computing self-attention for a single self-attention output vector $\mathbf{a}_i$ from a single input vector $\mathbf{x}_i$, illustrated in Fig. 10.3 for the case of calculating the value of the third output $\mathbf{a}_3$ in a sequence.

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^{\mathbf{Q}}; \mathbf{k}_i = \mathbf{x}_i\mathbf{W}^{\mathbf{K}}; \mathbf{v}_i = \mathbf{x}_i\mathbf{W}^{\mathbf{V}} \tag{10.11}$$

$$\textit{Final verson:} \quad \text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \tag{10.12}$$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \;\; \forall j \leq i \tag{10.13}$$

$$\mathbf{a}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{v}_j \tag{10.14}$$

10.1.4 Parallelizing self-attention using a single matrix X

This description of the self-attention process has been from the perspective of computing a single output at a single time step i. However, since each output, $\mathbf{y}_i$, is computed independently, this entire process can be parallelized, taking advantage of

efficient matrix multiplication routines by packing the input embeddings of the N tokens of the input sequence into a single matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$. That is, each row of $\mathbf{X}$ is the embedding of one token of the input. Transformers for large language models can have an input length N = 1024, 2048, or 4096 tokens, so $\mathbf{X}$ has between 1K and 4K rows, each of the dimensionality of the embedding d.

We then multiply $\mathbf{X}$ by the key, query, and value matrices (all of dimensionality $d \times d$) to produce matrices $\mathbf{Q} \in \mathbb{R}^{N \times d}$, $\mathbf{K} \in \mathbb{R}^{N \times d}$, and $\mathbf{V} \in \mathbb{R}^{N \times d}$, containing all the key, query, and value vectors:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}}; \quad \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}}; \quad \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}} \tag{10.15}$$

Given these matrices we can compute all the requisite query-key comparisons simultaneously by multiplying $\mathbf{Q}$ and $\mathbf{K}^\intercal$ in a single matrix multiplication (the product is of shape $N \times N$; Fig. 10.4 shows a visualization). Taking this one step further, we can scale these scores, take the softmax, and then multiply the result by $\mathbf{V}$ resulting in a matrix of shape $N \times d$: a vector embedding representation for each token in the input. We've reduced the entire self-attention step for an entire sequence of N tokens to the following computation:

$$\mathbf{A} = \text{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d_k}}\right)\mathbf{V} \tag{10.16}$$

10.1.5 Masking out the future

The self-attention computation as we've described it has a problem: the calculation in $\mathbf{Q}\mathbf{K}^\intercal$ results in a score for each query value to every key value, *including those that follow the query*. This is inappropriate in the setting of language modeling: guessing the next word is pretty simple if you already know it! To fix this, the elements in the upper-triangular portion of the matrix are zeroed out (set to $-\infty$), thus eliminating any knowledge of words that follow in the sequence. Fig. 10.4 shows this masked $\mathbf{Q}\mathbf{K}^\intercal$ matrix. (we'll see in Chapter 11 how to make use of words in the future for tasks that need it).

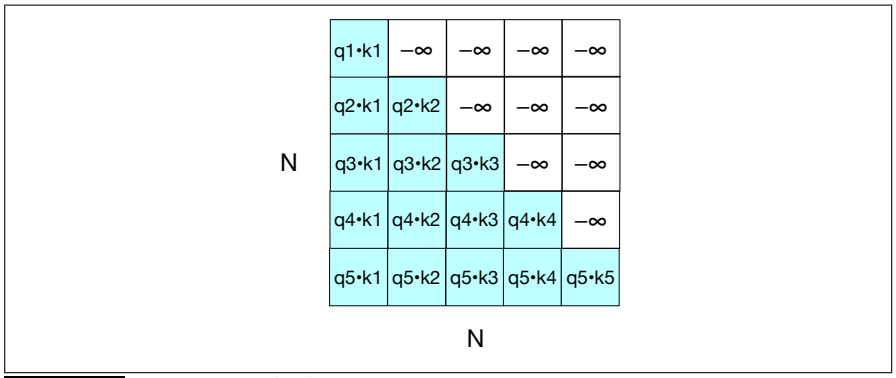

Figure 10.4 The $N \times N$ $\mathbf{Q}\mathbf{K}^\intercal$ matrix showing the $q_i \cdot k_j$ values, with the upper-triangle portion of the comparisons matrix zeroed out (set to $-\infty$, which the softmax will turn to zero).

Fig. 10.4 also makes it clear that attention is quadratic in the length of the input, since at each layer we need to compute dot products between each pair of tokens in the input. This makes it expensive for the input to a transformer to consist of very long documents (like entire novels). Nonetheless modern large language models manage to use quite long contexts of up to 4096 tokens.

10.2 Multihead Attention

multihead self-attention layers

Transformers actually compute a more complex kind of attention than the single self-attention calculation we've seen so far. This is because the different words in a sentence can relate to each other in many different ways simultaneously. For example, distinct syntactic, semantic, and discourse relationships can hold between verbs and their arguments in a sentence. It would be difficult for a single self-attention model to learn to capture all of the different kinds of parallel relations among its inputs. Transformers address this issue with **multihead self-attention layers**. These are sets of self-attention layers, called heads, that reside in parallel layers at the same depth in a model, each with its own set of parameters. By using these distinct sets of parameters, each head can learn different aspects of the relationships among inputs at the same level of abstraction.

To implement this notion, each head, i, in a self-attention layer is provided with its own set of key, query and value matrices: $\mathbf{W}_i^K$, $\mathbf{W}_i^Q$ and $\mathbf{W}_i^V$. These are used to project the inputs into separate key, value, and query embeddings separately for each head, with the rest of the self-attention computation remaining unchanged.

In multi-head attention, as with self-attention, the model dimension d is still used for the input and output, the key and query embeddings have dimensionality d_k, and the value embeddings are of dimensionality d_v (again, in the original transformer paper $d_k = d_v = 64$, $h = 8$, and $d = 512$). Thus for each head i, we have weight layers $\mathbf{W}_i^Q \in \mathbb{R}^{d\times d_k}$, $\mathbf{W}_i^K \in \mathbb{R}^{d\times d_k}$, and $\mathbf{W}_i^V \in \mathbb{R}^{d\times d_v}$, and these get multiplied by the inputs packed into $\mathbf{X}$ to produce $\mathbf{Q} \in \mathbb{R}^{N\times d_k}$, $\mathbf{K} \in \mathbb{R}^{N\times d_k}$, and $\mathbf{V} \in \mathbb{R}^{N\times d_v}$. The output of each of the h heads is of shape $N \times d_v$, and so the output of the multi-head layer with h heads consists of h matrices of shape $N \times d_v$. To make use of these matrices in further processing, they are concatenated to produce a single output with dimensionality $N \times hd_v$. Finally, we use yet another linear projection $\mathbf{W}^O \in \mathbb{R}^{hd_v\times d}$, that reshape it to the original output dimension for each token. Multiplying the concatenated $N \times hd_v$ matrix output by $\mathbf{W}^O \in \mathbb{R}^{hd_v\times d}$ yields the self-attention output A of shape $[N \times d]$, suitable to be passed through residual connections and layer norm.

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_i^Q ; \mathbf{K} = \mathbf{X}\mathbf{W}_i^K ; \mathbf{V} = \mathbf{X}\mathbf{W}_i^V \tag{10.17}$$

$$\mathbf{head}_i = \text{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \tag{10.18}$$

$$\mathbf{A} = \text{MultiHeadAttention}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \ldots \oplus \mathbf{head}_h)\mathbf{W}^O \tag{10.19}$$

Fig. 10.5 illustrates this approach with 4 self-attention heads. In general in transformers, the multihead layer is used instead of a self-attention layer.

10.3 Transformer Blocks

The self-attention calculation lies at the core of what's called a transformer block, which, in addition to the self-attention layer, includes three other kinds of layers: (1) a feedforward layer, (2) residual connections, and (3) normalizing layers (colloquially called "layer norm").

Fig. 10.6 illustrates a standard transformer block consisting of a single attention layer followed by a position-wise feedforward layer with residual connections and layer normalizations following each.

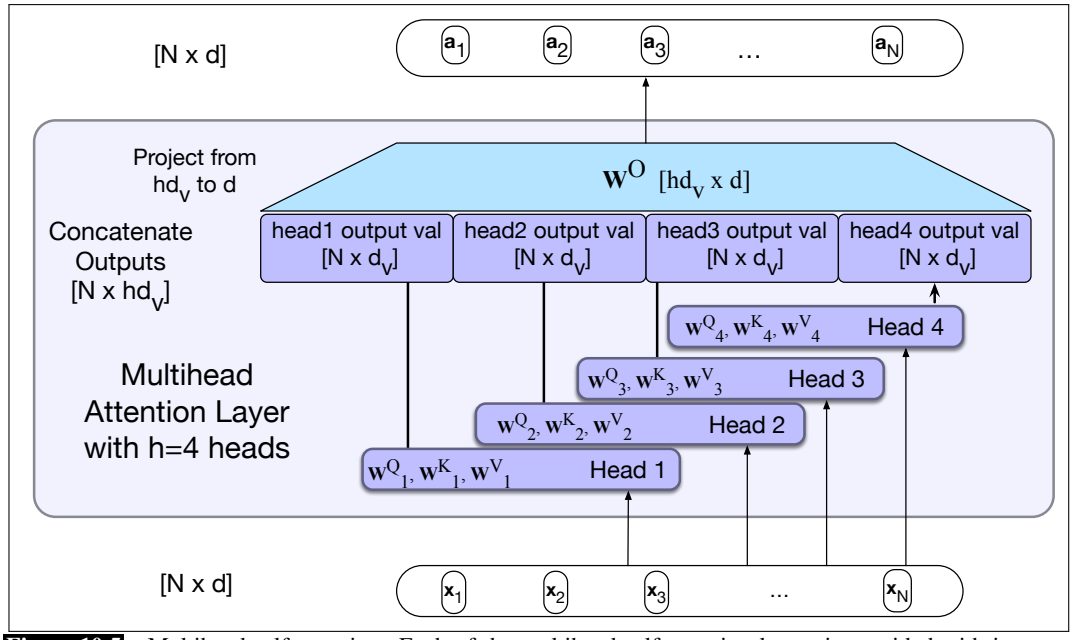

Figure 10.5 Multihead self-attention: Each of the multihead self-attention layers is provided with its own set of key, query and value weight matrices. The outputs from each of the layers are concatenated and then projected to d, thus producing an output of the same size as the input so the attention can be followed by layer norm and feedforward and layers can be stacked.

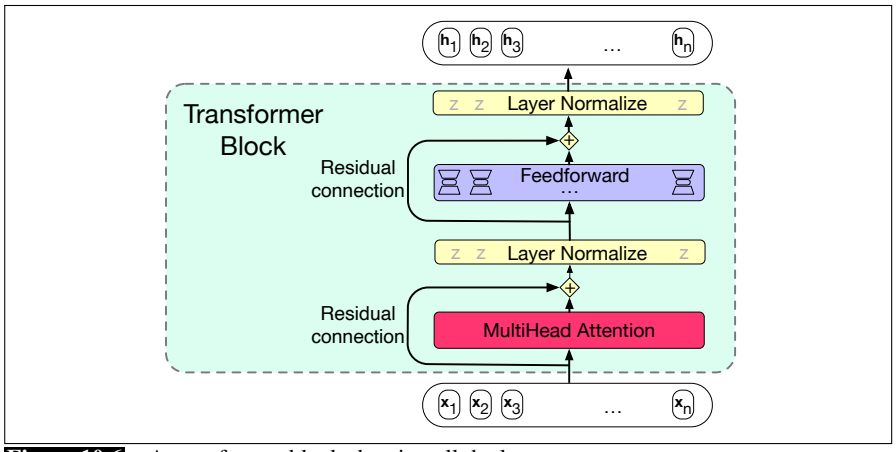

Figure 10.6 A transformer block showing all the layers.

Feedforward layer The feedforward layer contains N position-wise networks, one at each position. Each is a fully-connected 2-layer network, i.e., one hidden layer, two weight matrices, as introduced in Chapter 7. The weights are the same for each position, but the parameters are different from layer to layer. Unlike attention, the feedforward networks are independent for each position and so can be computed in parallel. It is common to make the dimensionality d_{ff} of the hidden layer of the feedforward network be larger than the model dimensionality d. (For example in the original transformer model, $d = 512$ and $d_{\text{ff}} = 2048$.)

Residual connections Residual connections are connections that pass information from a lower layer to a higher layer without going through the intermediate

layer. Allowing information from the activation going forward and the gradient going backwards to skip a layer improves learning and gives higher level layers direct access to information from lower layers (He et al., 2016). Residual connections in transformers are implemented simply by adding a layer's input vector to its output vector before passing it forward. In the transformer block shown in Fig. 10.6, residual connections are used with both the attention and feedforward sublayers.

layer norm

Layer Norm These summed vectors are then normalized using layer normalization (Ba et al., 2016). Layer normalization (usually called **layer norm**) is one of many forms of normalization that can be used to improve training performance in deep neural networks by keeping the values of a hidden layer in a range that facilitates gradient-based training. Layer norm is a variation of the standard score, or z-score, from statistics applied to a single vector in a hidden layer. The input to layer norm is a single vector, for a particular token position i, and the output is that vector normalized. Thus layer norm takes as input a single vector of dimensionality d and produces as output a single vector of dimensionality d. The first step in layer normalization is to calculate the mean, μ, and standard deviation, σ, over the elements of the vector to be normalized. Given a hidden layer with dimensionality d_h, these values are calculated as follows.

$$\mu = \frac{1}{d_h}\sum_{i=1}^{d_h} x_i \tag{10.20}$$

$$\sigma = \sqrt{\frac{1}{d_h}\sum_{i=1}^{d_h}(x_i-\mu)^2} \tag{10.21}$$

Given these values, the vector components are normalized by subtracting the mean from each and dividing by the standard deviation. The result of this computation is a new vector with zero mean and a standard deviation of one.

$$\hat{\mathbf{x}} = \frac{(\mathbf{x}-\mu)}{\sigma} \tag{10.22}$$

Finally, in the standard implementation of layer normalization, two learnable parameters, γ and β, representing gain and offset values, are introduced.

$$\text{LayerNorm} = \gamma\,\hat{\mathbf{x}} + \beta \tag{10.23}$$

Putting it all together The function computed by a transformer block can be expressed as:

$$\mathbf{O} = \text{LayerNorm}(\mathbf{X} + \text{SelfAttention}(\mathbf{X})) \tag{10.24}$$

$$\mathbf{H} = \text{LayerNorm}(\mathbf{O} + \text{FFN}(\mathbf{O})) \tag{10.25}$$

Or we can break it down with one equation for each component computation, using $\mathbf{T}$ (of shape $[N \times d]$) to stand for transformer and superscripts to demarcate each computation inside the block:

$$\mathbf{T^1} = \text{SelfAttention}(\mathbf{X}) \tag{10.26}$$

$$\mathbf{T^2} = \mathbf{X} + \mathbf{T^1} \tag{10.27}$$

$$\mathbf{T_3} = \text{LayerNorm}(\mathbf{T^2}) \tag{10.28}$$

$$\mathbf{T^4} = \text{FFN}(\mathbf{T^3}) \tag{10.29}$$

$$\mathbf{T^5} = \mathbf{T^4} + \mathbf{T^3} \tag{10.30}$$

$$\mathbf{H} = \text{LayerNorm}(\mathbf{T^5}) \tag{10.31}$$

Crucially, the input and output dimensions of transformer blocks are matched so they can be stacked. Each token x_i at the input to the block has dimensionality d, and so the input $\mathbf{X}$ and output $\mathbf{H}$ are both of shape $[N \times d]$.

Transformers for large language models stack many of these blocks, from 12 layers (used for the T5 or GPT-3-small language models) to 96 layers (used for GPT-3 large), to even more for more recent models. We'll come back to this issues of stacking in a bit.

10.4 The Residual Stream view of the Transformer Block

The previous sections viewed the transformer block as applied to the entire N-token input $\mathbf{X}$ of shape $[N \times d]$, producing an output also of shape $[N \times d]$.

While packing everything this way is a computationally efficient way to implement the transformer block, it's not always the most perspicuous way to understand what the transformer is doing. It's often clearer to instead visualize what is happening to an individual token vector $\mathbf{x_i}$ in the input as it is processed through each transformer block. After all, most of the components of the transformer are designed to take a single vector of dimensionality d, corresponding to a single token, and produce an output vector also of dimensionality d. For example, the feedforward layer takes a single d-dimensional vector and produces a single d-dimensional vector. Over the N tokens in a batch, we simply use the identical feedforward layer weights (W_1, W_2, b_1 and b_2) for each token i. Similarly, the layer norm function takes a single d-dimensional vector and produces a normalized d-dimensional version.

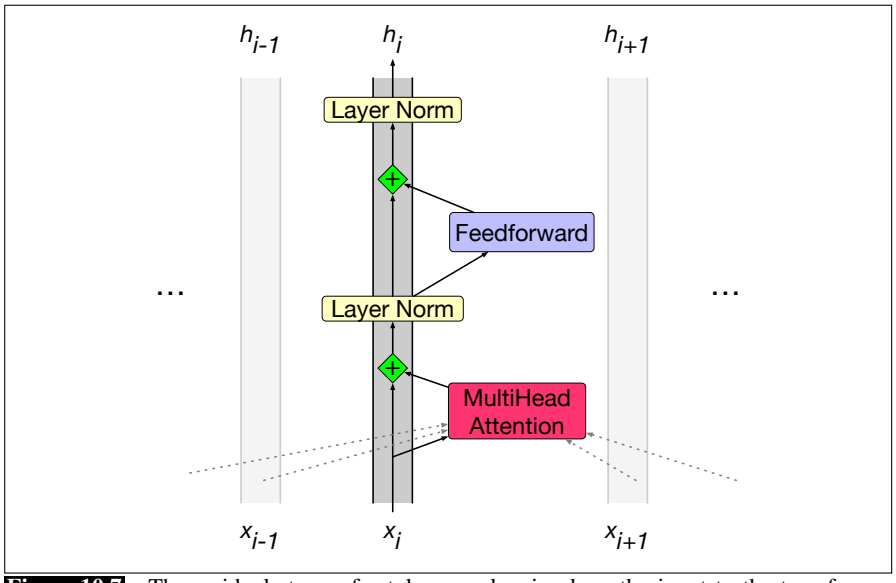

Figure 10.7 The residual stream for token x_i, showing how the input to the transformer block x_i is passed up through residual connections, the output of the feedforward and multi-head attention layers are added in, and processed by layer norm, to produce the output of this block, h_i, which is used as the input to the next layer transformer block. Note that of all the components, only the MultiHeadAttention component reads information from the other residual streams in the context.

We can therefore talk about the processing of an individual token through all

residual stream

these layers as a stream of d-dimensional representations, called the **residual stream** and visualized in Fig. 10.7. The input at the bottom of the stream is an embedding for a token, which has dimensionality d. That initial embedding is passed up by the residual connections and the outputs of feedforward and attention layers get added into it. For each token i, at each block and layer we are passing up an embedding of shape $[1 \times d]$. The residual layers are constantly copying information up from earlier embeddings (hence the metaphor of 'residual stream'), so we can think of the other components as adding new views of this representation back into this constant stream. Feedforward networks add in a different view of the earlier embedding.

Here are the equations for the transformer block, now viewed from this embedding stream perspective.

$$\mathbf{t}_i^1 = \text{MultiHeadAttention}(\mathbf{x}_i, [\mathbf{x}_1, \cdots, \mathbf{x}_N]) \tag{10.32}$$
$$\mathbf{t}_i^2 = \mathbf{t}_i^1 + \mathbf{x}_i \tag{10.33}$$
$$\mathbf{t}_i^3 = \text{LayerNorm}(\mathbf{t}_i^2) \tag{10.34}$$
$$\mathbf{t}_i^4 = \text{FFN}(\mathbf{t}_i^3)) \tag{10.35}$$
$$\mathbf{t}_i^5 = \mathbf{t}_i^4 + \mathbf{t}_i^3 \tag{10.36}$$
$$\mathbf{h}_i = \text{LayerNorm}(\mathbf{t}_i^5) \tag{10.37}$$

Notice that the only component that takes as input information from other tokens (other residual streams) is multi-head attention, which (as we see from (10.32) looks at all the neighboring tokens in the context. The output from attention, however, is then added into to this token's embedding stream. In fact, Elhage et al. (2021) show that we can view attention heads as literally moving attention from the residual stream of a neighboring token into the current stream. The high-dimensional embedding space at each position thus contains information about the current token and about neighboring tokens, albeit in different subspaces of the vector space. Fig. 10.8 shows a visualization of this movement.

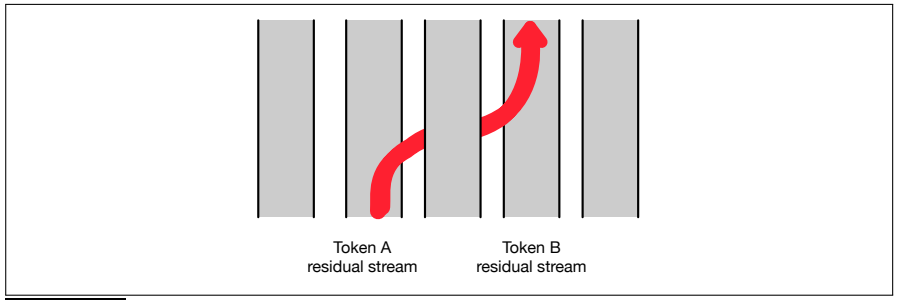

Figure 10.8 An attention head can move information from token A's residual stream into token B's residual stream.

Equation (10.32) and following are just just the equation for a single transformer block, but the residual stream metaphor goes through all the transformer layers, from the first transformer blocks to the 12th, in a 12-layer transformer. At the earlier transformer blocks, the residual stream is representing the current token. At the highest transformer blocks, the residual stream is usual representing the following token, since at the very end it's being trained to predict the next token.

Pre-norm vs. post-norm architecture There is an alternative form of the transformer architecture that is commonly used because it performs better in many cases. In this **prenorm transformer** architecture, the layer norm happens in a slightly dif-

prenorm transformer

ferent place: before the attention layer and before the feedforward layer, rather than afterwards. Fig. 10.9 shows this architecture, with the equations below:

$$\mathbf{t}_i^1 = \text{LayerNorm}(\mathbf{x}_i) \tag{10.38}$$
$$\mathbf{t}_i^2 = \text{MultiHeadAttention}(\mathbf{t}_i^1, [\mathbf{t}_1^1, \cdots, \mathbf{x}_N^1]) \tag{10.39}$$
$$\mathbf{t}_i^3 = \mathbf{t}_i^2 + \mathbf{x}_i \tag{10.40}$$
$$\mathbf{t}_i^4 = \text{LayerNorm}(\mathbf{t}_i^3) \tag{10.41}$$
$$\mathbf{t}_i^5 = \text{FFN}(\mathbf{t}_i^4)) \tag{10.42}$$
$$\mathbf{h}_i = \mathbf{t}_i^5 + \mathbf{t}_i^3 \tag{10.43}$$

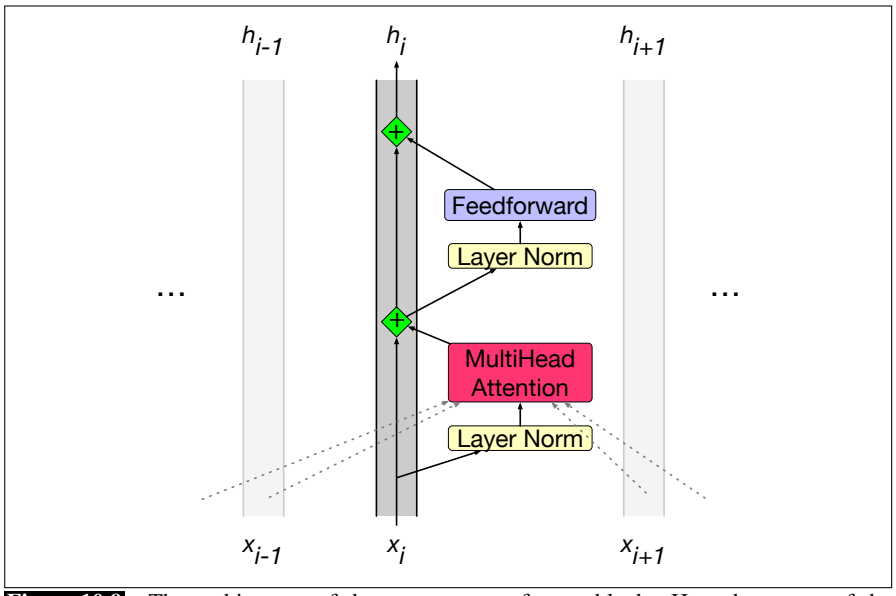

Figure 10.9 The architecture of the prenorm transformer block. Here the nature of the residual stream, passing up information from the input, is even clearer.

The prenorm transformer has one extra requirement: at the very end of the last (highest) transformer block, there is a single extra layer norm that is run on the last $\mathbf{h}_i$ of each token stream (just below the language model head layer that we will define below).

10.5 The input: embeddings for token and position

embedding

Let's talk about where the input $\mathbf{X}$ comes from. Given a sequence of N tokens (N is the context length in tokens), the matrix $\mathbf{X}$ of shape $[N \times d]$ has an **embedding** for each word in the context. The transformer does this by separately computing two embeddings: an input token embedding, and an input positional embedding.

A token embedding, introduced in Chapter 7 and Chapter 9, is a vector of dimension d that will be our initial representation for the input token. (As we pass vectors up through the transformer layers in the residual stream, this embedding representation will change and grow, incorporating context and playing a different

role depending on the kind of language model we are building.) The set of initial embeddings are stored in the embedding matrix **E**, which has a row for each of the $|V|$ tokens in the vocabulary. Thus each each word is a row vector of d dimensions, and **E** has shape $[|V| \times d]$.

Given an input token string like *Thanks for all the* we first convert the tokens into vocabulary indices (these were created when we first tokenized the input using BPE or SentencePiece). So the representation of *thanks for all the* might be $\mathbf{w} = [5, 4000, 10532, 2224]$. Next we use indexing to select the corresponding rows from **E**, (row 5, row 4000, row 10532, row 2224).

Another way to think about selecting token embeddings from the embedding matrix is to represent tokens as one-hot vectors of shape $[1 \times |V|]$, i.e., with one dimension for each word in the vocabulary. Recall that in a **one-hot vector** all the elements are 0 except one, the element whose dimension is the word's index in the vocabulary, which has value 1. So if the word "thanks" has index 5 in the vocabulary, $x_5 = 1$, and $x_i = 0 \ \forall i \neq 5$, as shown here:

one-hot vector

```
[0 0 0 0 1 0 0 ... 0 0 0 0]
 1 2 3 4 5 6 7 ...  ... |V|
```

Multiplying by a one-hot vector that has only one non-zero element $x_i = 1$ simply selects out the relevant row vector for word i, resulting in the embedding for word i, as depicted in Fig. 10.10.

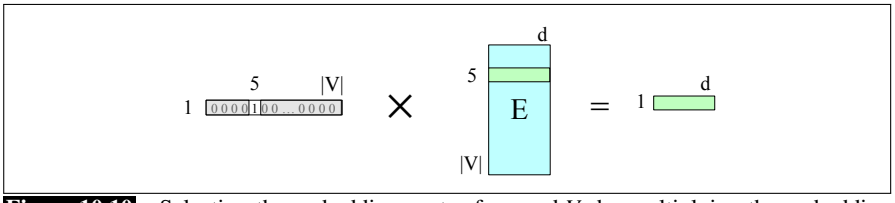

Figure 10.10 Selecting the embedding vector for word V_5 by multiplying the embedding matrix **E** with a one-hot vector with a 1 in index 5.

We can extend this idea to represent the entire token sequence as a matrix of one-hot vectors, one for each of the N positions in the transformer's context window, as shown in Fig. 10.11.

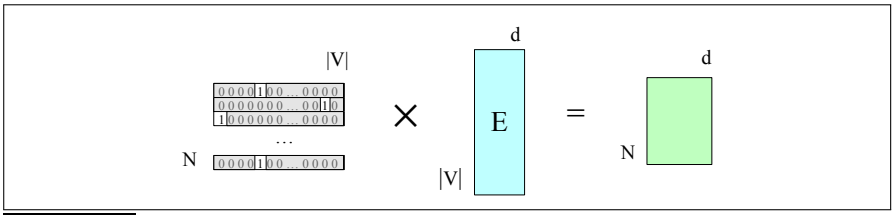

Figure 10.11 Selecting the embedding matrix for the input sequence of token ids W by multiplying a one-hot matrix corresponding to W by the embedding matrix **E**.

These token embeddings are not position-dependent. To represent the position of each token in the sequence, we combine these token embeddings with **positional embeddings** specific to each position in an input sequence.

positional embeddings

Where do we get these positional embeddings? The simplest method, called **absolute position**, is to start with randomly initialized embeddings corresponding to each possible input position up to some maximum length. For example, just as we have an embedding for the word *fish*, we'll have an embedding for the position 3.

absolute position

As with word embeddings, these positional embeddings are learned along with other parameters during training. We can store them in a matrix E_{pos} of shape $[1timesN]$.

To produce an input embedding that captures positional information, we just add the word embedding for each input to its corresponding positional embedding. The individual token and position embeddings are both of size $[1 \times d]$, so their sum is also $[1 \times d]$, This new embedding serves as the input for further processing. Fig. 10.12 shows the idea.

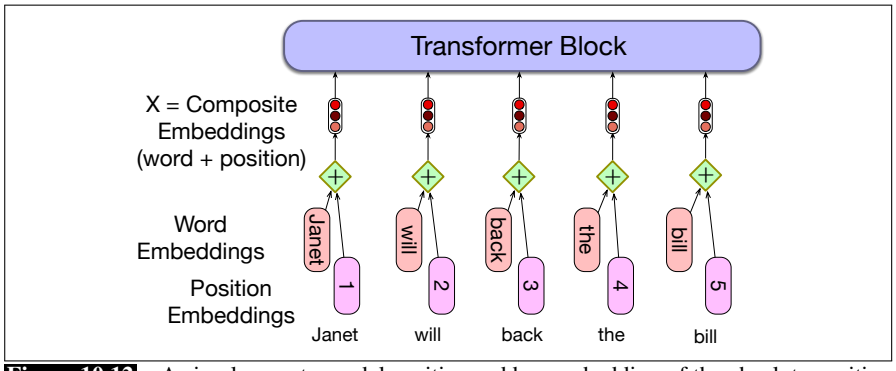

Figure 10.12 A simple way to model position: add an embedding of the absolute position to the token embedding to produce a new embedding of the same dimenionality.

The final representation of the input, the matrix $\mathbf{X}$, is an $[N \times d]$ matrix in which each row i is the representation of the ith token in the input, computed by adding $\mathbf{E}[id(i)]$—the embedding of the id of the token that occurred at position i—, to $\mathbf{P}[i]$, the positional embedding of position i.

A potential problem with the simple absolute position embedding approach is that there will be plenty of training examples for the initial positions in our inputs and correspondingly fewer at the outer length limits. These latter embeddings may be poorly trained and may not generalize well during testing. An alternative approach to absolute positional embeddings is to choose a static function that maps integer inputs to real-valued vectors in a way that captures the inherent relationships among the positions. That is, it captures the fact that position 4 in an input is more closely related to position 5 than it is to position 17. A combination of sine and cosine functions with differing frequencies was used in the original transformer work. Even more complex positional embedding methods exist, such as ones that represent **relative position** instead of absolute position, often implemented in the attention mechanism at each layer rather than being added once at the initial input.

10.6 The Language Modeling Head

language modeling head
head

The last component of the transformer we must introduce is the **language modeling head**. When we apply pretrained transformer models to various tasks, we use the term **head** to mean the additional neural circuitry we add on top of the basic transformer architecture to enable that task. The language modeling head is the circuitry we need to do language modeling.

Recall that language models, from the simple n-gram models of Chapter 3 through the feedforward and RNN language models of Chapter 7 and Chapter 9, are word predictors. Given a context of words, they assign a probability to each possible next

word. For example, if the preceding context is *"Thanks for all the"* and we want to know how likely the next word is *"fish"* we would compute:

$$P(fish|Thanks\ for\ all\ the)$$

Language models give us the ability to assign such a conditional probability to every possible next word, giving us a distribution over the entire vocabulary. The n-gram language models of Chapter 3 compute the probability of a word given counts of its occurrence with the $n-1$ prior words. The context is thus of size $n-1$. For transformer language models, the context is the size of the transformer's context window, which can be quite large: up to 2048 or even 4096 tokens for large models.

The job of the language modeling head is to take the the output of the final transformer layer from the last token N and use it to predict the upcoming word at position $N+1$. Fig. 10.13 shows how to accomplish this task, taking the output of the last token at the last layer (the d-dimensional output embedding of shape $[1 \times d]$) and producing a probability distribution over words (from which we will choose one to generate).

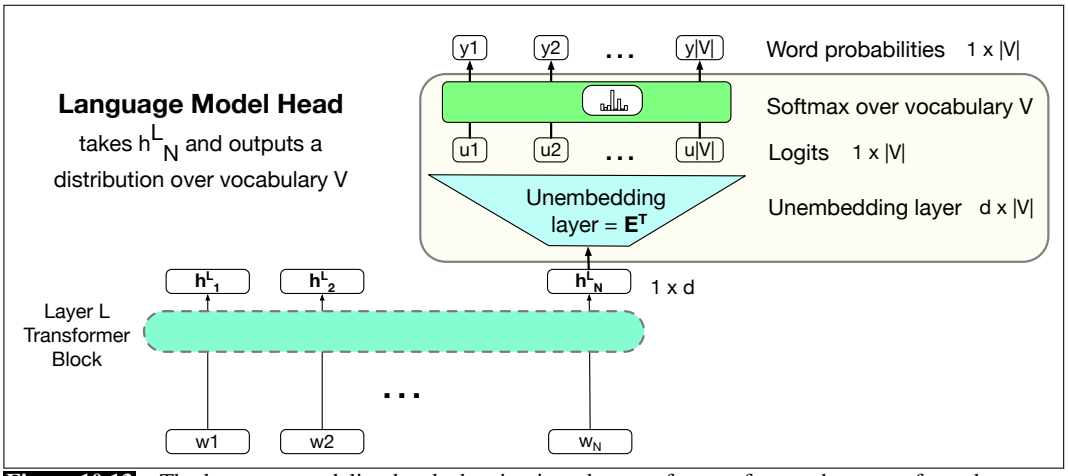

Figure 10.13 The language modeling head: the circuit at the top of a transformer that maps from the output embedding for token N from the last transformer layer ($\mathbf{h}_N^L$) to a probability distribution over words in the vocabulary V.

The first module in Fig. 10.13 is a linear layer, whose job is to project from the output h_N^L, which represents the output token embedding at position N from the final block L, (hence of shape $[1 \times d]$) to the **logit** vector, or score vector, that will have a single score for each of the $|V|$ possible words in the vocabulary V. The logit vector $\mathbf{u}$ is thus of dimensionality $1 \times |V|$.

logit

This linear layer can be learned, but more commonly we tie this matrix to (the transpose of) the embedding matrix $\mathbf{E}$. Recall that in **weight tying**, we use the same weights for two different matrices in the model. Thus at the input stage of the transformer the embedding matrix (of shape $[|V| \times d]$) is used to map from a one-hot vector over the vocabulary (of shape $[1 \times |V|]$) to an embedding (of shape $[1 \times d]$). And then in the language model head, $\mathbf{E}^\mathbf{T}$, the transpose of the embedding matrix (of shape $[d \times |V|]$) is used to map back from an embedding (shape $[1 \times d]$) to a vector over the vocabulary (shape $[1 \times |V|]$). In the learning process, $\mathbf{E}$ will be optimized to be good at doing both of these mappings. We therefore sometimes call the transpose $\mathbf{E}^\mathbf{T}$ the **unembedding** layer because it is performing this reverse mapping.

weight tying

unembedding

A softmax layer turns the logits **u** into the probabilities **y** over the vocabulary.

$$\mathbf{u} = \mathbf{h}_N^L \mathbf{E}^T \tag{10.44}$$

$$\mathbf{y} = \text{softmax}(\mathbf{u}) \tag{10.45}$$

We can use these probabilities to do things like help assign a probability to a given text. But the most important usage to generate text, which we do by **sampling** a word from these probabilities y. We might sample the highest probability word ('greedy' decoding), or use another of the sampling methods we'll introduce in Section 10.8. In either case, whatever entry y_k we choose from the probability vector **y**, we generate the word that has that index k.

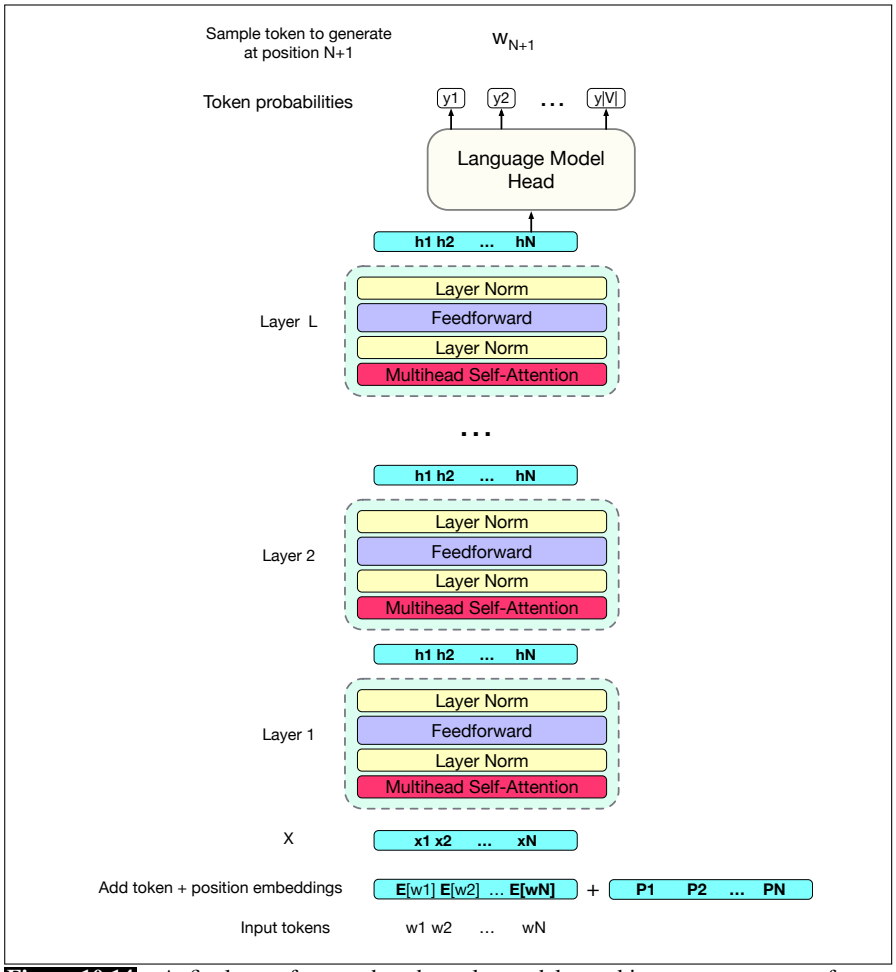

Figure 10.14 A final transformer decoder-only model, stacking post-norm transformer blocks and mapping from a set of input tokens w_1 to w_N to a predicted next word w_{N+1}.

Fig. 10.14 shows the total stacked architecture. Note that the input to the first transformer block is represented as **X**, which is the N indexed word embeddings + position embeddings, $\mathbf{E}[\mathbf{w}] + \mathbf{P}$), but the input to all the other layers is the output **H** from the layer just below the current one).

Now that we see all these transformer layers spread out on the page, we can point out another useful feature of the unembedding layer: as a tool for interpretability of

logit lens

the internals of the transformer that we call the **logit lens** (Nostalgebraist, 2020). We can take a vector from any layer of the transformer and, pretending that it is the prefinal embedding, simply multiply it by the unembedding layer to get logits, and compute a softmax to see the distribution over words that that vector might be representing. This can be a useful window into the internal representations of the model. Since the network wasn't trained to make the internal representations function in this way, the logit lens doesn't always work perfectly, but this can still be a useful trick.

Anyhow, the Fig. 10.14 thus sketches out the entire process of taking a series of words $w_1 \dots w_N$ and using the model to predict the next word w_{N+1}.

decoder-only model

A terminological note before we conclude: You will sometimes see a transformer used for this kind of unidirectional causal language model called a **decoder-only model**. This is because this model constitutes roughly half of the **encoder-decoder model** for transformers that we'll see how to apply to machine translation in Chapter 13. (Confusingly, the original introduction of the transformer had an encoder-decoder architecture, and it was only later that the standard paradigm for causal language model was defined by using only the decoder part of this original architecture).

In the next sections we'll introduce what kind of tasks large language models can be used for, discuss various generation methods for sampling possible next words, and show how to train a transformer-based large language model. In the following chapters we'll expand on these ideas to introduce fine-tuning, prompting, and encoder-decoder architectures for transformer-based large language models.

10.7 Large Language Models with Transformers

We've now seen most of the components of a transformer for language modeling (what remains is **sampling** and **training**, which we'll get to in the following sections). Before we do that, we use this section to talk about why and how we apply transformer-based large language models to NLP tasks.

All of these tasks are cases of **conditional generation**, the task of generating text conditioned on an input piece of text, a prompt. The fact that transformers have such long contexts (1024 or even 4096 tokens) makes them very powerful for conditional generation, because they can look back so far into the prompting text.

Consider the simple task of text completion, illustrated in Fig. 10.15. Here a language model is given a text prefix and is asked to generate a possible completion. Note that as the generation process proceeds, the model has direct access to the priming context as well as to all of its own subsequently generated outputs (at least as much as fits in the large context window).. This ability to incorporate the entirety of the earlier context and generated outputs at each time step is the key to the power of large language models built from transformers.

So why should we care about predicting upcoming words? The insight of large language modeling is that **many practical NLP tasks can be cast as word prediction**, and that a powerful-enough language model can solve them with a high degree of accuracy. For example, we can cast sentiment analysis as language modeling by giving a language model a context like:

> The sentiment of the sentence "I like Jackie Chan" is:

and comparing the following conditional probability of the words "positive" and the

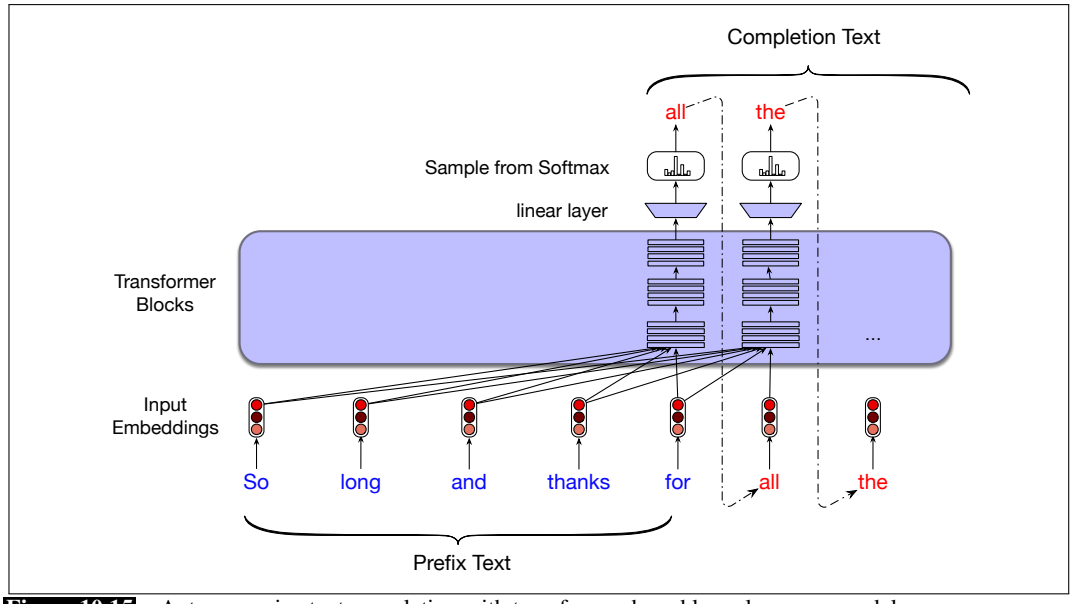

Figure 10.15 Autoregressive text completion with transformer-based large language models.

word "negative" to see which is higher:

$$P(positive|The\ sentiment\ of\ the\ sentence\ "I\ like\ Jackie\ Chan"\ is:)$$
$$P(negative|The\ sentiment\ of\ the\ sentence\ "I\ like\ Jackie\ Chan"\ is:)$$

If the word "positive" is more probable, we say the sentiment of the sentence is positive, otherwise we say the sentiment is negative.

We can also cast more complex tasks as word prediction. Consider the task of answering simple questions, a task we return to in Chapter 14. In this task the system is given some question and must give a textual answer. We can cast the task of question answering as word prediction by giving a language model a question and a token like `A:` suggesting that an answer should come next:

```
Q: Who wrote the book ‘‘The Origin of Species"?  A:
```

If we ask a language model to compute

$$P(w|Q:\ Who\ wrote\ the\ book\ "The\ Origin\ of\ Species"?\ A:)$$

and look at which words w have high probabilities, we might expect to see that *Charles* is very likely, and then if we choose *Charles* and continue and ask

$$P(w|Q:\ Who\ wrote\ the\ book\ "The\ Origin\ of\ Species"?\ A:\ Charles)$$

we might now see that *Darwin* is the most probable word, and select it.

text summarization

Conditional generation can even be used to accomplish tasks that must generate longer responses. Consider the task of **text summarization**, which is to take a long text, such as a full-length article, and produce an effective shorter summary of it. We can cast summarization as language modeling by giving a large language model a text, and follow the text by a token like `tl;dr`; this token is short for something like 'too long; don't read' and in recent years people often use this token, especially in informal work emails, when they are going to give a short summary. We can then do conditional generation: give the language model this prefix, and then ask

it to generate the following words, one by one, and take the entire response as a summary. Fig. 10.16 shows an example of a text and a human-produced summary from a widely-used summarization corpus consisting of CNN and Daily Mirror news articles.

Original Article

The only thing crazier than a guy in snowbound Massachusetts boxing up the powdery white stuff and offering it for sale online? People are actually buying it. For $89, self-styled entrepreneur Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says.

But not if you live in New England or surrounding states. "We will not ship snow to any states in the northeast!" says Waring's website, ShipSnowYo.com. "We're in the business of expunging snow!"

His website and social media accounts claim to have filled more than 133 orders for snow – more than 30 on Tuesday alone, his busiest day yet. With more than 45 total inches, Boston has set a record this winter for the snowiest month in its history. Most residents see the huge piles of snow choking their yards and sidewalks as a nuisance, but Waring saw an opportunity.

According to Boston.com, it all started a few weeks ago, when Waring and his wife were shoveling deep snow from their yard in Manchester-by-the-Sea, a coastal suburb north of Boston. He joked about shipping the stuff to friends and family in warmer states, and an idea was born. His business slogan: "Our nightmare is your dream!" At first, ShipSnowYo sold snow packed into empty 16.9-ounce water bottles for $19.99, but the snow usually melted before it reached its destination...

Summary

Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says. But not if you live in New England or surrounding states.

Figure 10.16 Examples of articles and summaries from the CNN/Daily Mail corpus (Hermann et al., 2015), (Nallapati et al., 2016).

If we take this full article and append the token `tl;dr`, we can use this as the context to prime the generation process to produce a summary as illustrated in Fig. 10.17. Again, what makes transformers able to succeed at this task (as compared, say, to the primitive n-gram language model) is that the ability of self-attention to incorporate information from the large context windows means that the model has access to the original article as well as to the newly generated text throughout the process.

greedy decoding

Which words do we generate at each step? One simple way to generate words is to always generate the most likely word given the context. Generating the most likely word given the context is called **greedy decoding**. A greedy algorithm is one that make a choice that is locally optimal, whether or not it will turn out to have been the best choice with hindsight. Thus in greedy decoding, at each time step in generation, the output y_t is chosen by computing the probability for each possible outputs (every word in the vocabulary) and then choosing the highest probability word (the argmax):

$$\hat{w}_t = \text{argmax}_{w \in V} P(w|\mathbf{w}_{<t}) \tag{10.46}$$

In practice, however, we don't use greedy decoding with large language models. A major problem with greedy decoding is that because the words it chooses are (by definition) extremely predictable, the resulting text is generic and often quite repetitive. Indeed, greedy decoding is so predictable that it is deterministic; if the context

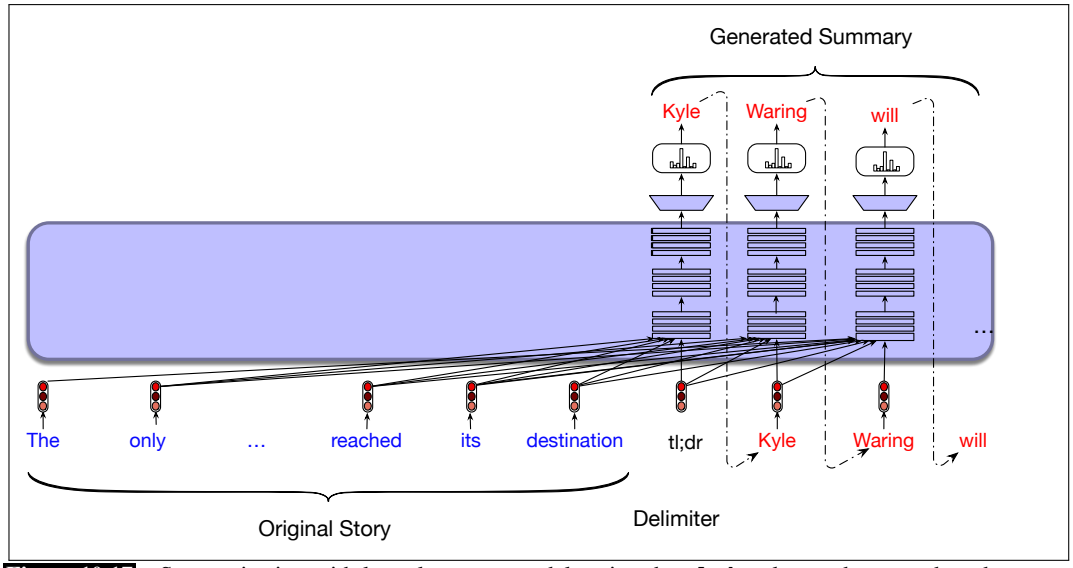

Figure 10.17 Summarization with large language models using the `tl;dr` token and context-based autoregressive generation.

is identical, and the probabilistic model is the same, greedy decoding will always result in generating exactly the same string. We'll see in Chapter 13 that an extension to greedy decoding called **beam search** works well in tasks like machine translation, which are very constrained in that we are always generating a text in one language conditioned on a very specific text in another language. In most other tasks, however, people prefer text which has been generated by more sophisticated methods, called **sampling methods**, that introduce a bit more diversity into the generations. We'll see how to do that in the next few sections.

10.8 Large Language Models: Generation by Sampling

The core of the generation process for large language models is the task of choosing the single word to generate next based on the context and based on the probabilities that the model assigns to possible words. This task of choosing a word to generate based on the model's probabilities is called **decoding**. Decoding from a language model in a left-to-right manner (or right-to-left for languages like Arabic in which we read from right to left), and thus repeatedly choosing the next word conditioned on our previous choices is called **autoregressive generation** or **causal LM generation**.[1] (As we'll see, alternatives like the masked language models of Chapter 11 are non-causal because they can predict words based on both past and future words).

decoding

autoregressive generation

The most common method for decoding in large language models is **sampling**. Recall from Chapter 3 that **sampling** from a model's distribution over words means to choose random words according to their probability assigned by the model. That is, we iteratively choose a word to generate according to its probability in context

sampling

[1] Technically an **autoregressive** model predicts a value at time t based on a linear function of the values at times $t-1$, $t-2$, and so on. Although language models are not linear (since they have many layers of non-linearities), we loosely refer to this generation technique as autoregressive since the word generated at each time step is conditioned on the word selected by the network from the previous step.

as defined by the model. Thus we are more likely to generate words that the model thinks have a high probability in the context and less likely to generate words that the model thinks have a low probability.

We saw back in Chapter 3 on page **??** how to generate text from a unigram language model , by repeatedly randomly sampling words according to their probability until we either reach a pre-determined length or select the end-of-sentence token. To generate text from a trained transformer language model we'll just generalize this model a bit: at each step we'll sample words according to their probability *conditioned on our previous choices*, and we'll use a transformer language model as the probability model that tells us this probability.

We can formalize this algorithm for generating a sequence of words $W = w_1, w_2, \ldots, w_N$ until we hit the end-of-sequence token, using $x \sim p(x)$ to mean 'choose x by sampling from the distribution $p(x)$:

i $\leftarrow$ 1
$w_i \sim$ p(w)
while w_i != EOS
  i $\leftarrow$ i + 1
  $w_i \sim$ p($w_i \mid w_{<i}$)

random sampling

The algorithm above is called **random sampling**, and it turns out random sampling doesn't work well enough. The problem is that even though random sampling is mostly going to generate sensible, high-probable words, there are many odd, low-probability words in the tail of the distribution, and even though each one is low-probability, if you add up all the rare words, they constitute a large enough portion of the distribution that they get chosen often enough to result in generating weird sentences. For this reason, instead of random sampling, we usually use sampling methods that avoid generating the very unlikely words.

The sampling methods we introduce below each have parameters that enable trading off two important factors in generation: **quality** and **diversity**. Methods that emphasize the most probable words tend to produce generations that are rated by people as more accurate, more coherent, and more factual, but also more boring and more repetitive. Methods that give a bit more weight to the middle-probability words tend to be more creative and more diverse, but less factual and more likely to be incoherent or otherwise low-quality.

10.8.1 Top-*k* sampling

top-k sampling

Top-k sampling is a simple generalization of greedy decoding. Instead of choosing the single most probable word to generate, we first truncate the distribution to the top k most likely words, renormalize to produce a legitimate probability distribution, and then randomly sample from within these k words according to their renormalized probabilities. More formally:

1. Choose in advance a number of words k
2. For each word in the vocabulary V, use the language model to compute the likelihood of this word given the context $p(w_t|\mathbf{w}_{<t})$
3. Sort the words by their likelihood, and throw away any word that is not one of the top k most probable words.
4. Renormalize the scores of the k words to be a legitimate probability distribution.

5. Randomly sample a word from within these remaining k most-probable words according to its probability.

When $k = 1$, top-k sampling is identical to greedy decoding. Setting k to a larger number than 1 leads us to sometimes select a word which is not necessarily the most probable, but is still probable enough, and whose choice results in generating more diverse but still high-enough-quality text.

10.8.2 Nucleus or top-p sampling

One problem with top-k sampling is that k is fixed, but the shape of the the probability distribution over words differs in different contexts. If we set $k = 10$, sometimes the top 10 words will be very likely and include most of the probability mass, but other times the probability distribution will be flatter and the top 10 words will only include a small part of the probability mass.

top-p sampling

An alternative, called **top-p sampling** or **nucleus sampling** (Holtzman et al., 2020), is to keep not the top k words, but the top p percent of the probability mass. The goal is the same; to truncate the distribution to remove the very unlikely words. But by measuring probability rather than the number of words, the hope is that the measure will be more robust in very different contexts, dynamically increasing and decreasing the pool of word candidates.

Given a distribution $P(w_t|\mathbf{w}_{<t})$, the top-p vocabulary $V^{(p)}$ is the smallest set of words such that

$$\sum_{w \in V^{(p)}} P(w|\mathbf{w}_{<t}) \geq p. \tag{10.47}$$

10.8.3 Temperature sampling

temperature sampling

In **temperature sampling**, we don't truncate the distribution, but instead reshape it. The intuition for temperature sampling comes from thermodynamics, where a system at a high temperature is very flexible and can explore many possible states, while a system at a lower temperature is likely to explore a subset of lower energy (better) states. In low-temperature sampling, we smoothly increase the probability of the most probable words and decrease the probability of the rare words.

We implement this intuition by simply dividing the logit by a temperature parameter τ before we normalize it by passing it through the softmax. In low-temperature sampling, $\tau \in (0, 1]$. Thus instead of computing the probability distribution over the vocabulary directly from the logit as in the following (repeated from (10.45)):

$$\mathbf{y} = \text{softmax}(u) \tag{10.48}$$

we instead first divide the logits by τ, computing the probability vector y as

$$\mathbf{y} = \text{softmax}(u/\tau) \tag{10.49}$$

Why does this work? When τ is close to 1 the distribution doesn't change much. But the lower τ is, the larger the scores being passed to the softmax (dividing by a smaller fraction $\tau \leq 1$ results in making each score larger). Recall that one of the useful properties of a softmax is that it tends to push high values toward 1 and low values toward 0. Thus when larger numbers are passed to a softmax the result is a distribution with increased probabilities of the most high-probability words and decreased probabilities of the low probability words, making the distribution more greedy. As τ approaches 0 the probability of the most likely word approaches 1.

Note, by the way, that there can be other situations where we may want to do something quite different and flatten the word probability distribution instead of making it greedy. Temperature sampling can help with this situation too, in this case **high-temperature** sampling, in which case we use $\tau > 1$.

10.9 Large Language Models: Training Transformers

How do we teach a transformer to be a language model? What is the algorithm and what data do we train on?

10.9.1 Self-supervised training algorithm

self-supervision

To train a transformer as a language model, we use the same **self-supervision** (or **self-training**) algorithm we saw in Section **??**: we take a corpus of text as training material and at each time step t ask the model to predict the next word. We call such a model self-supervised because we don't have to add any special gold labels to the data; the natural sequence of words is its own supervision! We simply train the model to minimize the error in predicting the true next word in the training sequence, using cross-entropy as the loss function.

Recall that the cross-entropy loss measures the difference between a predicted probability distribution and the correct distribution.

$$L_{CE} = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w] \tag{10.50}$$

In the case of language modeling, the correct distribution $\mathbf{y}_t$ comes from knowing the next word. This is represented as a one-hot vector corresponding to the vocabulary where the entry for the actual next word is 1, and all the other entries are 0. Thus, the cross-entropy loss for language modeling is determined by the probability the model assigns to the correct next word. So at time t the CE loss in (10.50) can be simplified as the negative log probability the model assigns to the next word in the training sequence.

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}] \tag{10.51}$$

Thus at each word position t of the input, the model takes as input the correct sequence of tokens $w_{1:t}$, and uses them to compute a probability distribution over possible next words so as to compute the model's loss for the next token w_{t+1}. Then we move to the next word, we ignore what the model predicted for the next word and instead use the correct sequence of tokens $w_{1:t+1}$ to estimate the probability of token w_{t+2}. This idea that we always give the model the correct history sequence to predict the next word (rather than feeding the model its best case from the previous time step) is called **teacher forcing**.

teacher forcing

Fig. 10.18 illustrates the general training approach. At each step, given all the preceding words, the final transformer layer produces an output distribution over the entire vocabulary. During training, the probability assigned to the correct word is used to calculate the cross-entropy loss for each item in the sequence. As with RNNs, the loss for a training sequence is the average cross-entropy loss over the entire sequence. The weights in the network are adjusted to minimize the average CE loss over the training sequence via gradient descent.

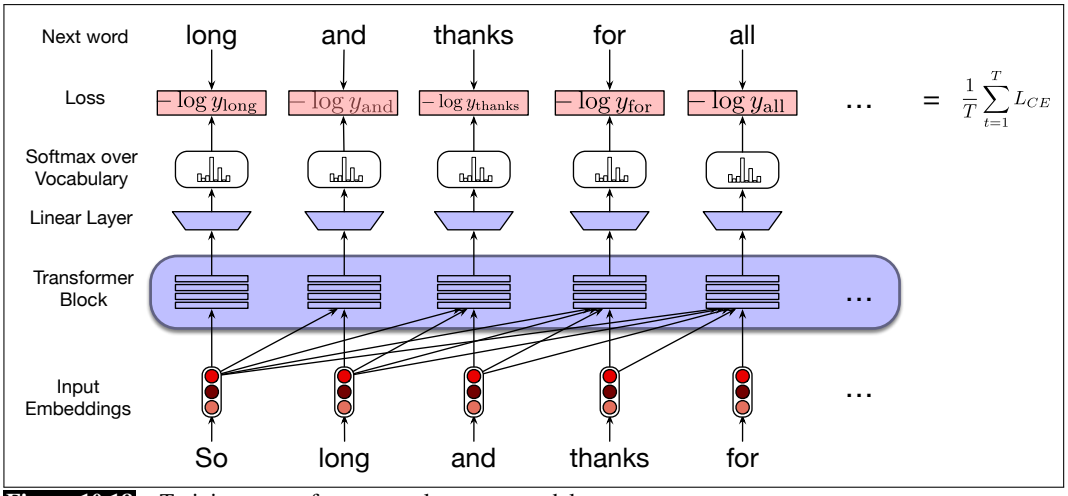

Figure 10.18 Training a transformer as a language model.

Note the key difference between this figure and the earlier RNN-based version shown in Fig. **??**. There the calculation of the outputs and the losses at each step was inherently serial given the recurrence in the calculation of the hidden states. With transformers, each training item can be processed in parallel since the output for each element in the sequence is computed separately.

Large models are generally trained by filling the full context window (for example 2048 or 4096 tokens for GPT3 or GPT4) with text. If documents are shorter than this, multiple documents are packed into the window with a special end-of-text token between them. The batch size for gradient descent is usually quite large (the largest GPT-3 model uses a batch size of 3.2 million tokens).

10.9.2 Training corpora for large language models

Large language models are mainly trained on text scraped from the web, augmented by more carefully curated data. Because these training corpora are so large, they are likely to contain many natural examples that can be helpful for NLP tasks, such as question and answer pairs (for example from FAQ lists), translations of sentences between various languages, documents together with their summaries, and so on.

common crawl

Web text is usually taken from corpora of automatically-crawled web pages like the **common crawl**, a series of snapshots of the entire web produced by the non-profit Common Crawl (`https://commoncrawl.org/`) that each have billions of webpages. Various cleanups of common crawl data exist, such as the Colossal Clean Crawled Corpus (C4; Raffel et al. 2020), a corpus of 156 billion tokens of English that is filtered in various ways (deduplicated, removing non-natural language like code, sentences with offensive words from a blocklist). What is in this data? An analysis suggests that in large part it's patent text documents, Wikipedia, and news sites (Dodge et al., 2021). Wikipedia plays a role in lots of language model training, as do corpora of books. The GPT3 models, for example, are trained mostly on the web (429 billion tokens), some text from books (67 billion tokens) and Wikipedia (3 billion tokens).

10.9.3 Scaling laws

The performance of large language models has shown to be mainly determined by 3 factors: model size (the number of parameters not counting embeddings), dataset size (the amount of training data), and the amount of computer used for training. That is, we can improve a model by adding parameters (adding more layers or having wider contexts or both), by training on more data, or by training for more iterations.

scaling laws

The relationships between these factors and performance are known as **scaling laws**. Roughly speaking, the performance of a large language model (the loss) scales as a power-law with each of these three properties of model training.

For example, Kaplan et al. (2020) found the following three relationships for loss L as a function of the number of non-embedding parameters N, the dataset size D, and the compute budget C, for models training with limited parameters, dataset, or compute budget, if in each case the other two properties are held constant:

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N} \tag{10.52}$$

$$L(D) = \left(\frac{D_c}{D}\right)^{\alpha_D} \tag{10.53}$$

$$L(C) = \left(\frac{C_c}{C}\right)^{\alpha_C} \tag{10.54}$$

The number of (non-embedding) parameters N can be roughly computed as follows (ignoring biases, and with d as the input and output dimensionality of the model, d_{attn} as the self-attention layer size, and d_{ff} the size of the feedforward layer):

$$\begin{aligned} N &\approx 2\, d\, n_{\text{layer}}(2\, d_{\text{attn}} + d_{\text{ff}}) \\ &\approx 12\, n_{\text{layer}}\, d^2 \\ &\quad (\text{assuming } d_{\text{attn}} = d_{\text{ff}}/4 = d) \end{aligned} \tag{10.55}$$

Thus GPT-3, with $n = 96$ layers and dimensionality $d = 12288$, has $12 \times 96 \times 12288^2 \approx 175$ billion parameters.

The values of N_c, D_c, C_c, α_N, α_D, and α_C depend on the exact transformer architecture, tokenization, and vocabulary size, so rather than all the precise values, scaling laws focus on the relationship with loss.[2]

Scaling laws can be useful in deciding how to train a model to a particular performance, for example by looking at early in the training curve, or performance with smaller amounts of data, to predict what the loss would be if we were to add more data or increase model size. Other aspects of scaling laws can also tell us how much data we need to add when scaling up a model.

10.10 Potential Harms from Language Models

Large pretrained neural language models exhibit many of the potential harms discussed in Chapter 4 and Chapter 6. Many of these harms become realized when pretrained language models are used for any downstream task, particularly those

[2] For the initial experiment in Kaplan et al. (2020) the precise values were $\alpha_N = 0.076$, $N_c = 8.8 \times 10^{13}$ (parameters), $\alpha_D = 0.095$, $D_c = 5.4 \times 10^{13}$ (tokens), $\alpha_C = 0.050$, $C_c = 3.1 \times 10^8$ (petaflop-days).

involving text generation, whether question answering, machine translation, or in assistive technologies like writing aids or web search query completion, or predictive typing for email (Olteanu et al., 2020).

hallucination

For example, language models are prone to saying things that are false, a problem called **hallucination**. Language models are trained to generate text that is predictable and coherent, but the training algorithms we have seen so far don't have any way to enforce that the text that is generated is correct or true. This causes enormous problems for any application where the facts matter!

toxic language

A second source of harm is that language models can generate **toxic language**. Gehman et al. (2020) show that even completely non-toxic prompts can lead large language models to output hate speech and abuse their users. Language models also generate stereotypes (Cheng et al., 2023) and negative attitudes (Brown et al., 2020; Sheng et al., 2019) about many demographic groups.

One source of biases is the training data. Gehman et al. (2020) shows that large language model training datasets include toxic text scraped from banned sites. There are other biases than toxicity: the training data is disproportionately generated by authors from the US and from developed countries. Such biased population samples likely skew the resulting generation toward the perspectives or topics of this group alone. Furthermore, language models can amplify demographic and other biases in training data, just as we saw for embedding models in Chapter 6.

Language models can also be used by malicious actors for generating text for **misinformation**, phishing, or other socially harmful activities (Brown et al., 2020). McGuffie and Newhouse (2020) show how large language models generate text that emulates online extremists, with the risk of amplifying extremist movements and their attempt to radicalize and recruit.

Language models also present **privacy** issues since they can **leak** information about their training data. It is thus possible for an adversary to extract training-data text from a language model such as an individual person's name, phone number, and address (Henderson et al. 2017, Carlini et al. 2021). This is a problem if large language models are trained on private datasets such as electronic health records.

Related to privacy is the issue of **copyright**. Large language models are trained on text that is copyrighted. In some countries, like the United States, the fair use doctrine allows copyrighted content to be used to build language models, but possibly not if they are used to generate text that competes with the market for the text they are trained on.

Finding ways to mitigate all these harms is an important current research area in NLP. At the very least, carefully analyzing the data used to pretrain large language models is important as a way of understanding issues of toxicity, bias, privacy, and fair use, making it extremely important that language models include **datasheets** (page **??**) or **model cards** (page **??**) giving full replicable information on the corpora used to train them. Open-source models can specify their exact training data. Requirements that models are transparent in such ways is also in the process of being incorporated into the regulations of various national governments.

10.11 Summary

This chapter has introduced the transformer, and how it can be applied to build large language models. Here's a summary of the main points that we covered:

- Transformers are non-recurrent networks based on **self-attention**. A self-attention layer maps input sequences to output sequences of the same length, using attention heads that model how the surrounding words are relevant for the processing of the current word.
- A transformer block consists of a single attention layer followed by a feed-forward layer with residual connections and layer normalizations following each. Transformer blocks can be stacked to make deeper and more powerful networks.
- Language models can be built out of stacks of transformer blocks, with a linear and softmax max layer at the top.
- Transformer-based language models have a wide context window (as wide as 4096 tokens for current models) allowing them to draw on enormous amounts of context to predict upcoming words.
- Many NLP tasks—such as question answering, summarization, sentiment, and machine translation—can be cast as tasks of word prediction and hence addressed with Large language models.
- The choice of which word to generate in large language models is generally done by using a **sampling** algorithm.
- Because of their ability to be used in so many ways, language models also have the potential to cause harms. Some harms include hallucinations, bias, stereotypes, misinformation and propaganda, and violations of privacy and copyright.

Bibliographical and Historical Notes

The transformer (Vaswani et al., 2017) was developed drawing on two lines of prior research: **self-attention** and **memory networks**. Encoder-decoder attention, the idea of using a soft weighting over the encodings of input words to inform a generative decoder (see Chapter 13) was developed by Graves (2013) in the context of handwriting generation, and Bahdanau et al. (2015) for MT. This idea was extended to self-attention by dropping the need for separate encoding and decoding sequences and instead seeing attention as a way of weighting the tokens in collecting information passed from lower layers to higher layers (Ling et al., 2015; Cheng et al., 2016; Liu et al., 2016). Other aspects of the transformer, including the terminology of key, query, and value, came from **memory networks**, a mechanism for adding an external read-write memory to networks, by using an embedding of a query to match keys representing content in an associative memory (Sukhbaatar et al., 2015; Weston et al., 2015; Graves et al., 2014).

MORE HISTORY TBD IN NEXT DRAFT.

Ba, J. L., J. R. Kiros, and G. E. Hinton. 2016. Layer normalization. *NeurIPS workshop*.

Bahdanau, D., K. H. Cho, and Y. Bengio. 2015. Neural machine translation by jointly learning to align and translate. *ICLR 2015*.

Brown, T., B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. 2020. Language models are few-shot learners. *NeurIPS*, volume 33.

Carlini, N., F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson, et al. 2021. Extracting training data from large language models. *30th USENIX Security Symposium (USENIX Security 21)*.

Cheng, J., L. Dong, and M. Lapata. 2016. Long short-term memory-networks for machine reading. *EMNLP*.

Cheng, M., E. Durmus, and D. Jurafsky. 2023. Marked personas: Using natural language prompts to measure stereotypes in language models. *ACL 2023*.

Dodge, J., M. Sap, A. Marasović, W. Agnew, G. Ilharco, D. Groeneveld, M. Mitchell, and M. Gardner. 2021. Documenting large webtext corpora: A case study on the colossal clean crawled corpus. *EMNLP*.

Elhage, N., N. Nanda, C. Olsson, T. Henighan, N. Joseph, B. Mann, A. Askell, Y. Bai, A. Chen, T. Conerly, N. DasSarma, D. Drain, D. Ganguli, Z. Hatfield-Dodds, D. Hernandez, A. Jones, J. Kernion, L. Lovitt, K. Ndousse, D. Amodei, T. Brown, J. Clark, J. Kaplan, S. McCandlish, and C. Olah. 2021. A mathematical framework for transformer circuits. White paper.

Gehman, S., S. Gururangan, M. Sap, Y. Choi, and N. A. Smith. 2020. RealToxicityPrompts: Evaluating neural toxic degeneration in language models. *Findings of EMNLP*.

Graves, A. 2013. Generating sequences with recurrent neural networks. ArXiv.

Graves, A., G. Wayne, and I. Danihelka. 2014. Neural Turing machines. ArXiv.

He, K., X. Zhang, S. Ren, and J. Sun. 2016. Deep residual learning for image recognition. *CVPR*.

Henderson, P., K. Sinha, N. Angelard-Gontier, N. R. Ke, G. Fried, R. Lowe, and J. Pineau. 2017. Ethical challenges in data-driven dialogue systems. *AAAI/ACM AI Ethics and Society Conference*.

Hermann, K. M., T. Kočiský, E. Grefenstette, L. Espeholt, W. Kay, M. Suleyman, and P. Blunsom. 2015. Teaching machines to read and comprehend. *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. MIT Press.

Holtzman, A., J. Buys, L. Du, M. Forbes, and Y. Choi. 2020. The curious case of neural text degeneration. *ICLR*.

Kaplan, J., S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. 2020. Scaling laws for neural language models. ArXiv preprint.

Ling, W., C. Dyer, A. W. Black, I. Trancoso, R. Fermandez, S. Amir, L. Marujo, and T. Luís. 2015. Finding function in form: Compositional character models for open vocabulary word representation. *EMNLP*.

Liu, Y., C. Sun, L. Lin, and X. Wang. 2016. Learning natural language inference using bidirectional LSTM model and inner-attention. ArXiv.

McGuffie, K. and A. Newhouse. 2020. The radicalization risks of GPT-3 and advanced neural language models. ArXiv preprint arXiv:2009.06807.

Nallapati, R., B. Zhou, C. dos Santos, Ç. Gulçehre, and B. Xiang. 2016. Abstractive text summarization using sequence-to-sequence RNNs and beyond. *CoNLL*.

Nostalgebraist. 2020. Interpreting gpt: the logit lens. White paper.

Olteanu, A., F. Diaz, and G. Kazai. 2020. When are search completion suggestions problematic? *CSCW*.

Raffel, C., N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR*, 21(140):1–67.

Sheng, E., K.-W. Chang, P. Natarajan, and N. Peng. 2019. The woman worked as a babysitter: On biases in language generation. *EMNLP*.

Sukhbaatar, S., A. Szlam, J. Weston, and R. Fergus. 2015. End-to-end memory networks. *NeurIPS*.

Uszkoreit, J. 2017. Transformer: A novel neural network architecture for language understanding. Google Research blog post, Thursday August 31, 2017.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. 2017. Attention is all you need. *NeurIPS*.

Weston, J., S. Chopra, and A. Bordes. 2015. Memory networks. *ICLR 2015*.