CHAPTER

10 Transformers and Pretrained 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 bring an enormous amount of knowledge to bear during comprehension and production of language. This knowledge is embodied in many forms, perhaps most obviously in the vocabulary. That is, in the rich representations associated with the words we know, including their grammatical function, meaning, real-world reference, and pragmatic function. 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 in context 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 some 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 given their daily experiences during this period? We know that most of this growth is not happening through direct vocabulary instruction in school since these methods are largely ineffective, and are not deployed at a rate that would result in the reliable acquisition of words at the required rate.

The most likely remaining explanation is that the bulk of this knowledge acquisition happens as a by-product of reading. 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've encountered 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 during language processing long after its initial acquisition in novel contexts. Of course, adding grounding from vision or from real-world interaction into such models can help build even more powerful models, but even text alone is remarkably useful, and we will limit our attention here to purely textual models.

pretraining

In this chapter we formalize this idea under the name **pretraining**. We call **pretraining** the process of learning some sort of representation of meaning for words or sentences by processing very large amounts of text. We say that we pretrain a language model, and then we call the resulting models **pretrained language models**.

While we have seen that the RNNs or even the FFNs of previous chapters can be used to learn language models, in this chapter we introduce the most common architecture for language modeling: the **transformer**.

transformer

The transformer offers new mechanisms (self-attention and positional encodings) that help represent time and help focus on how words relate to each other over long distances. We'll see how to apply this model to the task of language modeling, and then we'll see how a transformer pretrained on language modeling can be used in a zero shot manner to perform other NLP tasks.

10.1 Self-Attention Networks: Transformers

transformers

In this section we introduce the architecture of **transformers**. Like the LSTMs of Chapter 9, transformers 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 map sequences of input vectors $(\mathbf{x}_1, ..., \mathbf{x}_n)$ to sequences of output vectors $(\mathbf{y}_1, ..., \mathbf{y}_n)$ of the same length. Transformers are made up of stacks of transformer **blocks**, each of which is a multilayer network 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 without the need to pass it through intermediate recurrent connections as in RNNs. We'll start by describing how self-attention works and then return to how it fits into larger transformer blocks.

self-attention

Fig. 10.1 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 $(\mathbf{y}_1, ..., \mathbf{y}_n)$. 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.

At the core of an attention-based approach is the ability to *compare* 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, the set of comparisons are to other elements within a given sequence. The result of these comparisons is then used to compute an output for the current input. For example, returning to Fig. 10.1, the

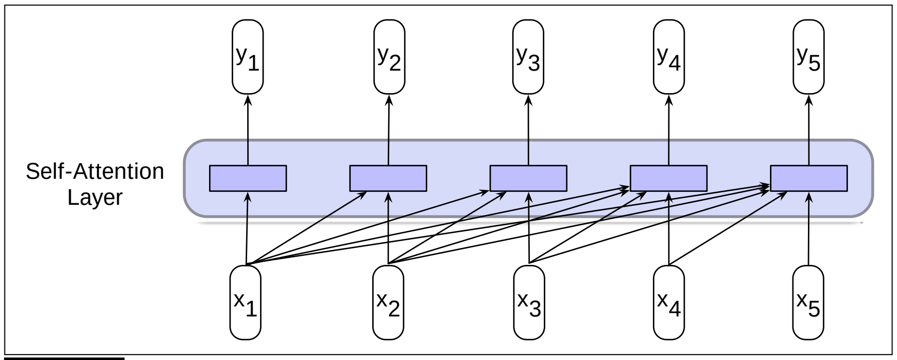

Figure 10.1 Information flow in a causal (or masked) self-attention model. In processing each element of the sequence, the model attends to all the inputs up to, and including, the current one. Unlike RNNs, the computations at each time step are independent of all the other steps and therefore can be performed in parallel.

computation of $\mathbf{y}_3$ 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. The simplest form of comparison between elements in a self-attention layer is a dot product. Let's refer to the result of this comparison as a score (we'll be updating this equation to add attention to the computation of this score):

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j \tag{10.1}$$

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.

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \ \forall j \leq i \tag{10.2}$$

$$= \frac{\exp(\text{score}(\mathbf{x}_i, \mathbf{x}_j))}{\sum_{k=1}^{i} \exp(\text{score}(\mathbf{x}_i, \mathbf{x}_k))} \ \forall j \leq i \tag{10.3}$$

Given the proportional scores in α, we then generate an output value $\mathbf{y}_i$ by taking the sum of the inputs seen so far, weighted by their respective α value.

$$\mathbf{y}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j \tag{10.4}$$

The steps embodied in Equations 10.1 through 10.4 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 $\mathbf{y}$ 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**. 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**. key
- And finally, as a **value** used to compute the output for the current focus of attention. 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.

$$\mathbf{q}_i = \mathbf{W^Q}\mathbf{x}_i;\ \ \mathbf{k}_i = \mathbf{W^K}\mathbf{x}_i;\ \ \mathbf{v}_i = \mathbf{W^V}\mathbf{x}_i \tag{10.5}$$

The inputs $\mathbf{x}$ and outputs $\mathbf{y}$ of transformers, as well as the intermediate vectors after the various layers, all have the same dimensionality $1 \times d$. For now let's assume the dimensionalities of the transform matrices are $\mathbf{W}^Q \in \mathbb{R}^{d\times d}$, $\mathbf{W}^K \in \mathbb{R}^{d\times d}$, and $\mathbf{W}^V \in \mathbb{R}^{d\times d}$. Later we'll need separate dimensions for these matrices when we introduce multi-headed attention, so let's just make a note that we'll have a dimension d_k for the key and query vectors, and a dimension d_v for the value vectors, both of which for now we'll set to d. In the original transformer work (Vaswani et al., 2017), d was 1024.

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 $1 \times d$. Let's update our previous comparison calculation to reflect this, replacing Eq. 10.1 with Eq. 10.6:

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) \ = \ \mathbf{q}_i \cdot \mathbf{k}_j \tag{10.6}$$

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

$$\mathbf{y}_i \ = \ \sum_{j\le i} \alpha_{ij}\mathbf{v}_j \tag{10.7}$$

Fig. 10.2 illustrates this calculation in the case of computing the third output $\mathbf{y}_3$ in a sequence.

The result of a dot product can be an arbitrarily large (positive or negative) value. Exponentiating such large values can lead to numerical issues and to an effective loss of gradients during training. To avoid this, the dot product needs to be scaled in a suitable fashion. A scaled dot-product approach divides the result of the dot product by a factor related to the size of the embeddings before passing them through the softmax. A typical approach is to divide the dot product 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. 10.1 and Eq. 10.6 with Eq. 10.8:

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) \ = \ \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \tag{10.8}$$

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 by taking advantage of efficient matrix multiplication routines by packing the input embeddings of the N

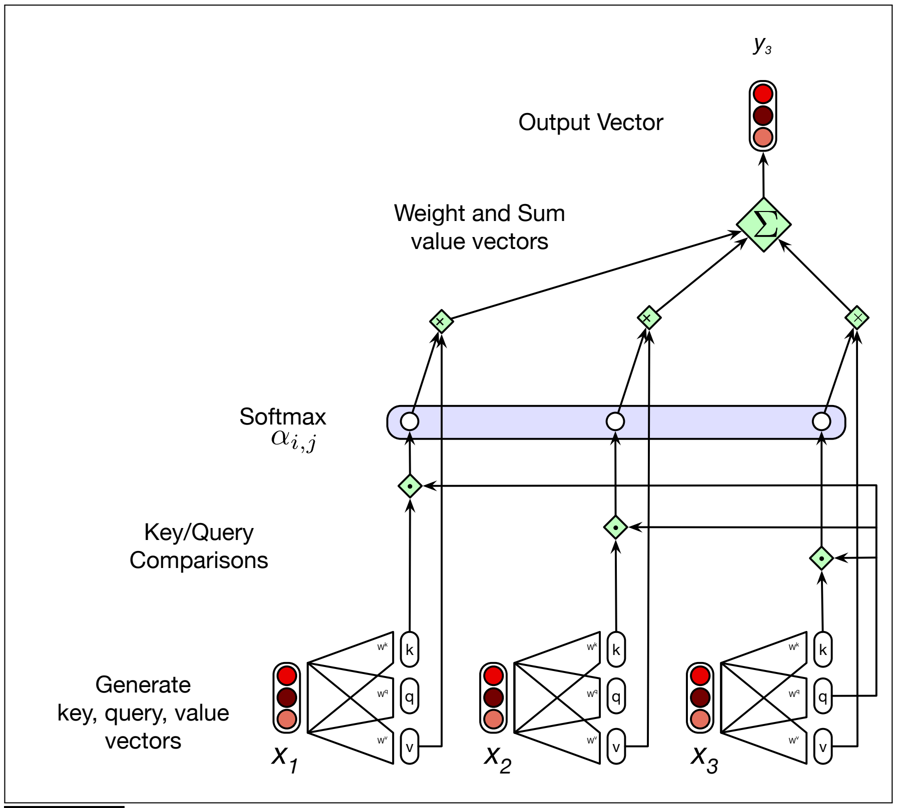

Figure 10.2 Calculating the value of $\mathbf{y}_3$, the third element of a sequence using causal (left-to-right) self-attention.

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. 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:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}}; \;\; \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}}; \;\; \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}} \tag{10.9}$$

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. 10.3 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:

$$\text{SelfAttention}(\mathbf{Q},\mathbf{K},\mathbf{V}) \;=\; \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\intercal}}{\sqrt{d_k}}\right)\mathbf{V} \tag{10.10}$$

Unfortunately, this process goes a bit too far since the calculation of the comparisons 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 since 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. 10.3

depicts the $\mathbf{QK}^\intercal$ matrix. (we'll see in Chapter 11 how to make use of words in the future for tasks that need it).

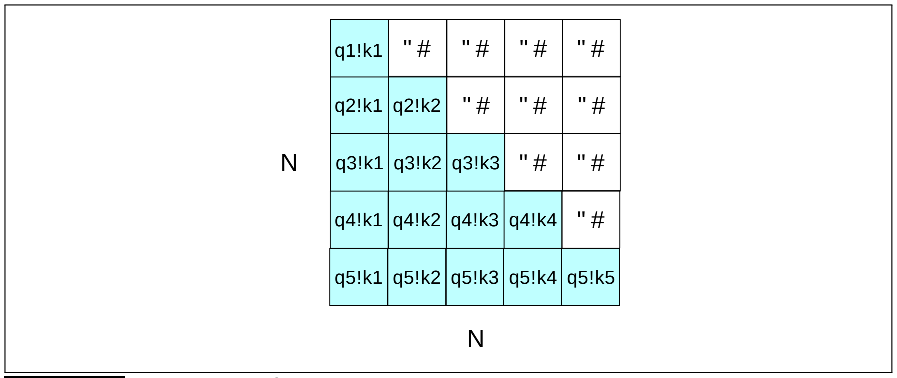

Figure 10.3 The $N \times N$ $\mathbf{QK}^\intercal$ matrix showing the $q_i \cdot k_j$ values, with the upper-triangle portion of the comparisons matrix zeroed out (set to $-\infty$, which the softmax will turn to zero).

Fig. 10.3 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 extremely expensive for the input to a transformer to consist of long documents (like entire Wikipedia pages, or novels), and so most applications have to limit the input length, for example to at most a page or a paragraph of text at a time. Finding more efficient attention mechanisms is an ongoing research direction.

10.1.1 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 additional feedforward layers, residual connections, and normalizing layers. The input and output dimensions of these blocks are matched so they can be stacked just as was the case for stacked RNNs.

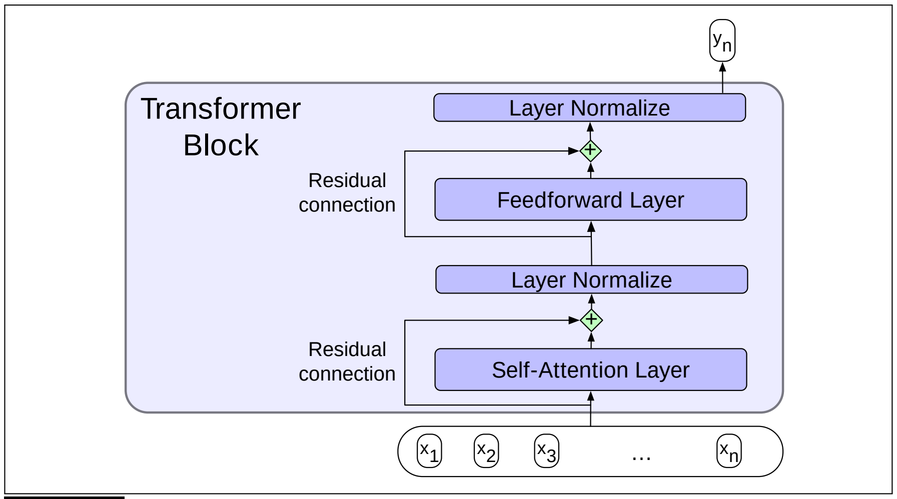

Figure 10.4 A transformer block showing all the layers.

Fig. 10.4 illustrates a standard transformer block consisting of a single attention layer followed by a fully-connected feedforward layer with residual connections and layer normalizations following each. We've already seen feedforward layers in Chapter 7, but what are residual connections and layer norm? In deep networks, 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 by adding a layer's input vector to its output vector before passing it forward. In the transformer block shown in Fig. 10.4, residual connections are used with both the attention and feedforward sublayers. These summed vectors are then normalized using layer normalization (Ba et al., 2016). If we think of a layer as one long vector of units, the resulting function computed in a transformer block can be expressed as:

$$\mathbf{z} = \text{LayerNorm}(\mathbf{x} + \text{SelfAttention}(\mathbf{x})) \tag{10.11}$$

$$\mathbf{y} = \text{LayerNorm}(\mathbf{z} + \text{FFN}(\mathbf{z})) \tag{10.12}$$

layer norm

Layer normalization (or **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 hidden layer. 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.

$$\mu = \frac{1}{d_h}\sum_{i=1}^{d_h} x_i \tag{10.13}$$

$$\sigma = \sqrt{\frac{1}{d_h}\sum_{i=1}^{d_h}(x_i - \mu)^2} \tag{10.14}$$

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.

$$\hat{\mathbf{x}} = \frac{(\mathbf{x} - \mu)}{\sigma} \tag{10.15}$$

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

$$\text{LayerNorm} = \gamma\hat{\mathbf{x}} + \beta \tag{10.16}$$

10.1.2 Multihead Attention

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 transformer block 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.

multihead self-attention layers

Given these distinct sets of parameters, each head can learn different aspects of the relationships that exist 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, instead of using the model dimension d that's used for the input and output from the model, the key and query embeddings have dimensionality d_k, and the value embeddings are of dimensionality d_v (in the original transformer paper $d_k = d_v = 64$). 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 vectors of shape $N \times d_v$. To make use of these vectors in further processing, they are combined and then reduced down to the original input dimension d. This is accomplished by concatenating the outputs from each head and then using yet another linear projection, $\mathbf{W}^O \in \mathbb{R}^{hd_v \times d}$, to reduce it to the original output dimension for each token, or a total $N \times d$ output.

$$\text{MultiHeadAttention}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 ... \oplus \mathbf{head}_h)\mathbf{W}^O \tag{10.17}$$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_i^Q \; ; \; \mathbf{K} = \mathbf{X}\mathbf{W}_i^K \; ; \; \mathbf{V} = \mathbf{X}\mathbf{W}_i^V \tag{10.18}$$

$$\mathbf{head}_i = \text{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \tag{10.19}$$

Fig. 10.5 illustrates this approach with 4 self-attention heads. This multihead layer replaces the single self-attention layer in the transformer block shown earlier in Fig. 10.4. The rest of the transformer block with its feedforward layer, residual connections, and layer norms remains the same.

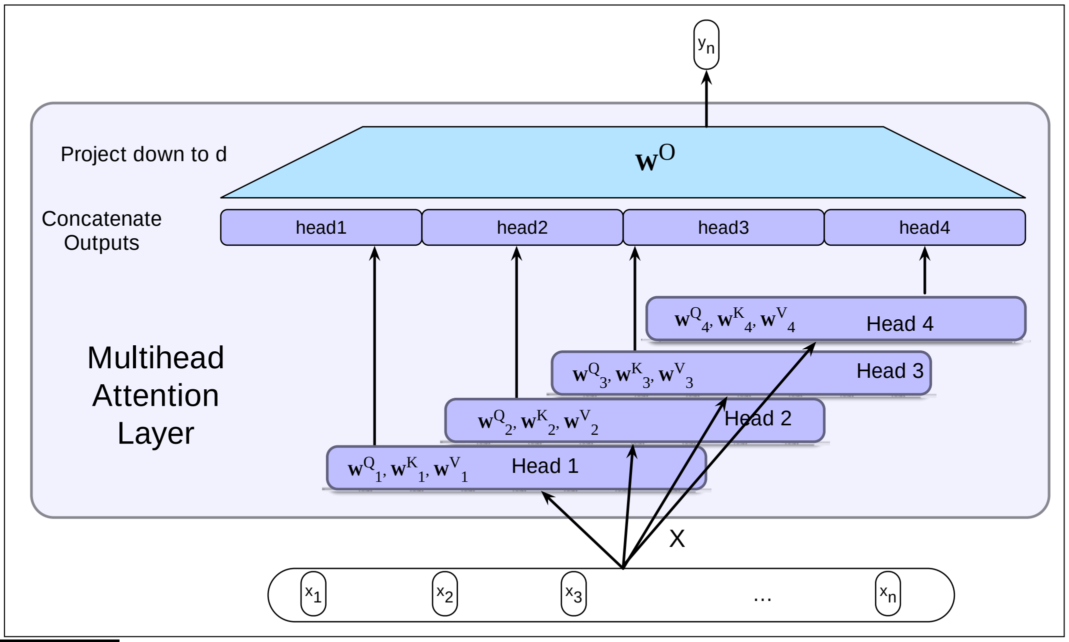

Figure 10.5 Multihead self-attention: Each of the multihead self-attention layers is provided with its own set of key, query and value weight matrices. The outputs from each of the layers are concatenated and then projected down to d, thus producing an output of the same size as the input so layers can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

10.1.3 Modeling word order: positional embeddings

How does a transformer model the position of each token in the input sequence? With RNNs, information about the order of the inputs was built into the structure of the model. Unfortunately, the same isn't true for transformers; the models as we've described them so far don't have any notion of the relative, or absolute, positions of the tokens in the input. This can be seen from the fact that if you scramble the order of the inputs in the attention computation in Fig. 10.2 you get exactly the same answer.

positional embeddings

One simple solution is to modify the input embeddings by combining them with **positional embeddings** specific to each position in an input sequence.

Where do we get these positional embeddings? The simplest method 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. As with word embeddings, these positional embeddings are learned along with other parameters during training. To produce an input embedding that captures positional information, we just add the word embedding for each input to its corresponding positional embedding. (We don't concatenate the two embeddings, we just add them to produce a new vector of the same dimensionality.). This new embedding serves as the input for further processing. Fig. 10.6 shows the idea.

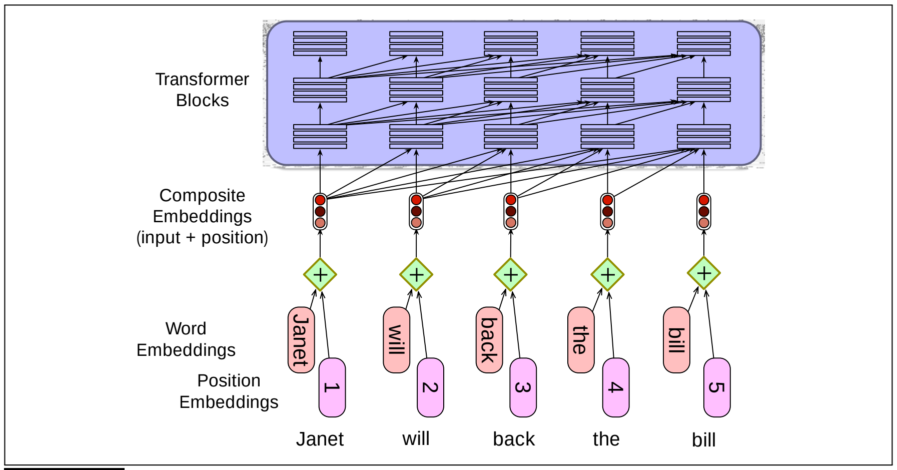

Figure 10.6 A simple way to model position: simply adding an embedding representation of the absolute position to the input word embedding to produce a new embedding of the same dimenionality.

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 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. Developing better position representations is an ongoing research topic.

10.2 Transformers as Language Models

Now that we've seen all the major components of transformers, let's examine how to deploy them as language models via self-supervised learning. To do this, we'll use the same self-supervision model we used for training RNN language models in Chapter 9. Given a training corpus of plain text we'll train the model autoregressively to predict the next token in a sequence $\mathbf{y}_t$, using cross-entropy loss. Recall from Eq. 9.11 that 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 is the negative log probability the model assigns to the next word in the training sequence:

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}] \tag{10.20}$$

teacher forcing

As in that case, we use **teacher forcing** . Recall that in teacher forcing, at each time step in decoding we force the system to use the gold target token from training as the next input x_{t+1}, rather than allowing it to rely on the (possibly erroneous) decoder output $\hat{y}_t$.

Fig. 10.7 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.

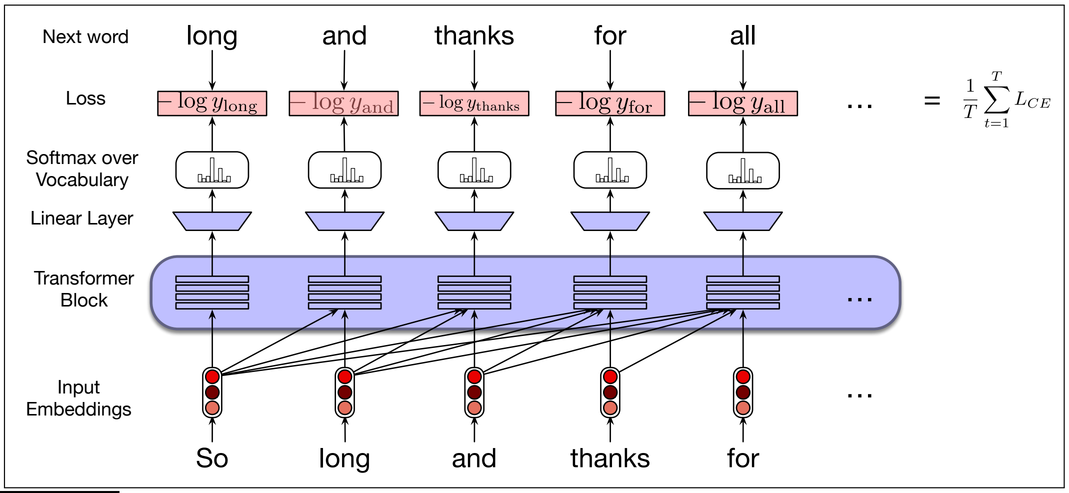

Figure 10.7 Training a transformer as a language model.

Note the key difference between this figure and the earlier RNN-based version shown in Fig. 9.6. 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.

Once trained, we can autoregressively generate novel text just as with RNN-based models. Recall from Section 9.3.3 that using a language model to incrementally generate words by repeatedly sampling the next word conditioned on our previous choices is called **autoregressive generation** or **causal LM generation**.

autoregressive generation

Recall back in Chapter 3 we saw how to generate text from an n-gram language model by adapting a **sampling** technique suggested at about the same time by Claude Shannon (Shannon, 1951) and the psychologists George Miller and Jennifer Selfridge (Miller and Selfridge, 1950). We first randomly sample a word to begin a sequence based on its suitability as the start of a sequence. We then continue to sample words *conditioned on our previous choices* until we reach a pre-determined length, or an end of sequence token is generated.

The procedure for generation from transformer LMs is basically the same as that described on page 40, but adapted to a neural context:

- Sample a word in the output from the softmax distribution that results from using the beginning of sentence marker, `<s>`, as the first input.
- Use the word embedding for that first word as the input to the network at the next time step, and then sample the next word in the same fashion.
- Continue generating until the end of sentence marker, `</s>`, is sampled or a fixed length limit is reached.

Technically an **autoregressive** model is a model that predicts a value at time t based on a linear function of the previous 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 generation** since the word generated at each time step is conditioned on the word selected by the network from the previous step.

The use of a language model to generate text is one of the areas in which the impact of neural language models on NLP has been the largest. Text generation, along with image generation and code generation, constitute a new area of AI that is often called **generative AI**.

More formally, for generating from a trained language model, at each time step in decoding, the output y_t is chosen by computing a softmax over the set of possible outputs (the vocabulary) and then choosing the highest probability token (the argmax):

$$\hat{y}_t = \text{argmax}_{w \in V} P(w|y_1 \ldots y_{t-1}) \tag{10.21}$$

greedy

Choosing the single most probable token to generate at each step 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. We'll see in following sections that there are other options to greedy decoding.

10.3 Sampling

TBD: nucleus, top k, temperature sampling.

10.4 Beam Search

Greedy search is not optimal, and may not find the highest probability translation. The problem is that the token that looks good to the decoder now might turn out later to have been the wrong choice!

search tree

Let's see this by looking at the **search tree**, a graphical representation of the choices the decoder makes in generating the next token. in which we view the decoding problem as a heuristic state-space search and systematically explore the space of possible outputs. In such a search tree, the branches are the actions, in this case the action of generating a token, and the nodes are the states, in this case the state of having generated a particular prefix. We are searching for the best action sequence, i.e. the target string with the highest probability. Fig. 10.8 demonstrates the problem, using a made-up example. Notice that the most probable sequence is *ok ok* `</s>` (with a probability of .4*.7*1.0), but a greedy search algorithm will fail to find it, because it incorrectly chooses *yes* as the first word since it has the highest local probability.

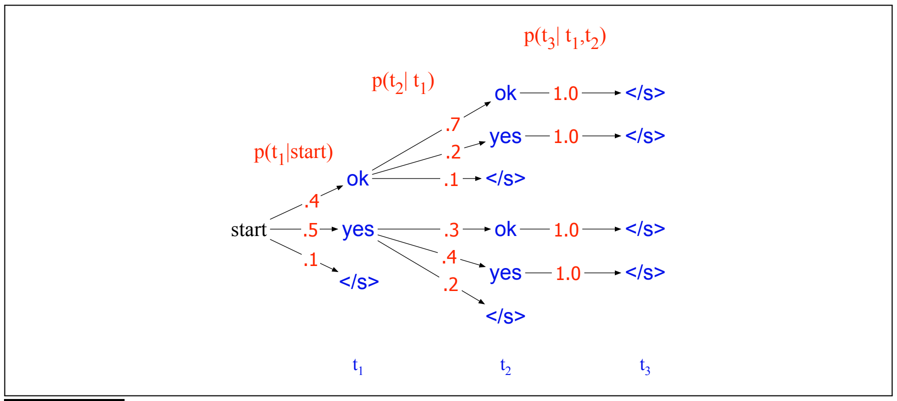

Figure 10.8 A search tree for generating the target string $T = t_1, t_2, \ldots$ from the vocabulary $V = \{\text{yes}, \text{ok}, \texttt{<s>}\}$, showing the probability of generating each token from that state. Greedy search would choose *yes* at the first time step followed by *yes*, instead of the globally most probable sequence *ok ok*.

Recall from Chapter 8 that for part-of-speech tagging we used dynamic programming search (the Viterbi algorithm) to address this problem. Unfortunately, dynamic programming is not applicable to generation problems with long-distance dependencies between the output decisions. The only method guaranteed to find the best solution is exhaustive search: computing the probability of every one of the V^T possible sentences (for some length value T) which is obviously too slow.

beam search

beam width

Instead, decoding in sequence generation problems generally uses a method called **beam search**. In beam search, instead of choosing the best token to generate at each timestep, we keep k possible tokens at each step. This fixed-size memory footprint k is called the **beam width**, on the metaphor of a flashlight beam that can be parameterized to be wider or narrower.

Thus at the first step of decoding, we compute a softmax over the entire vocabulary, assigning a probability to each word. We then select the k-best options from this softmax output. These initial k outputs are the search frontier and these k initial words are called **hypotheses**. A hypothesis is an output sequence, a translation-so-far, together with its probability.

At subsequent steps, each of the k best hypotheses is extended incrementally by being passed to distinct decoders, which each generate a softmax over the entire vocabulary to extend the hypothesis to every possible next token. Each of these $k * V$ hypotheses is scored by $P(y_i|x, y_{<i})$: the product of the probability of current word choice multiplied by the probability of the path that led to it. We then prune the $k * V$ hypotheses down to the k best hypotheses, so there are never more than k hypotheses

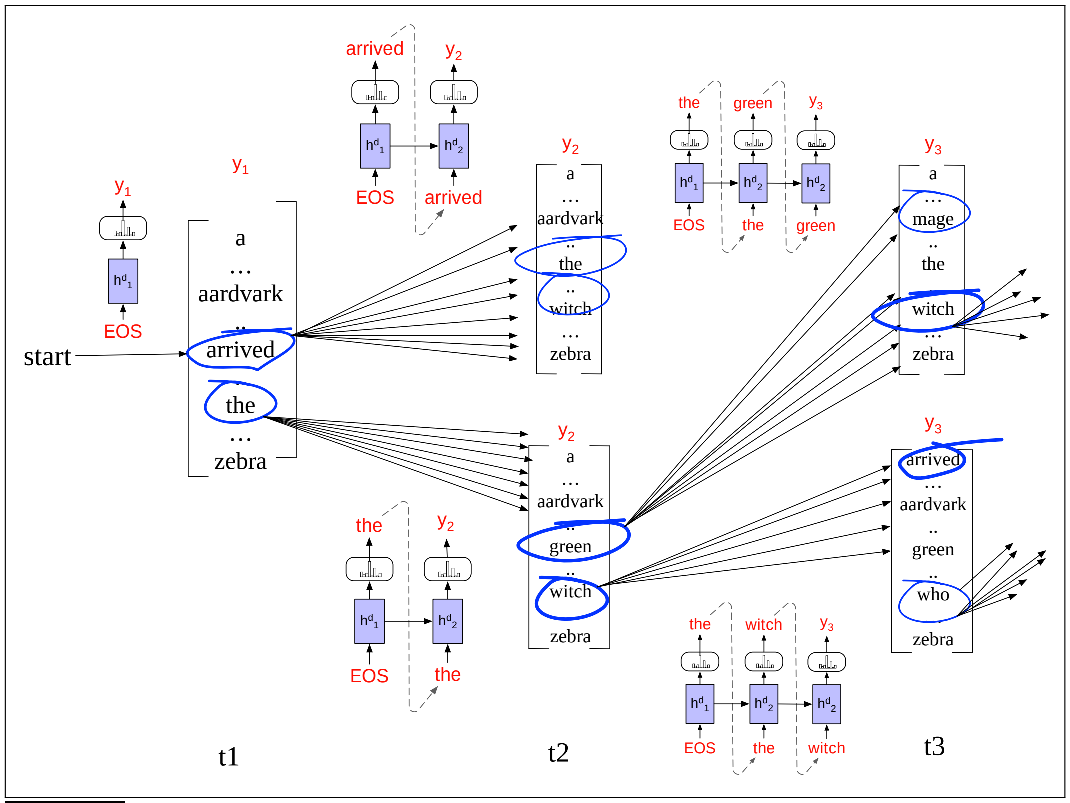

Figure 10.9 Beam search decoding with a beam width of $k = 2$. At each time step, we choose the k best hypotheses, compute the V possible extensions of each hypothesis, score the resulting $k * V$ possible hypotheses and choose the best k to continue. At time 1, the frontier is filled with the best 2 options from the initial state of the decoder: *arrived* and *the*. We then extend each of those, compute the probability of all the hypotheses so far (*arrived the*, *arrived aardvark*, *the green*, *the witch*) and compute the best 2 (in this case *the green* and *the witch*) to be the search frontier to extend on the next step. On the arcs we show the decoders that we run to score the extension words (although for simplicity we haven't shown the context value c_i that is input at each step).

at the frontier of the search, and never more than k decoders.

Fig. 10.9 illustrates this process with a beam width of 2 for the start of our sentence we used in Chapter 9 and will continue to use in Chapter 13 to introduce machine translation, (*The green witch arrived*).

This process continues until a `</s>` is generated indicating that a complete candidate output has been found. At this point, the completed hypothesis is removed from the frontier and the size of the beam is reduced by one. The search continues until the beam has been reduced to 0. The result will be k hypotheses.

Let's see how the scoring works in detail, scoring each node by its log probability. Recall from Eq. 9.31 that we can use the chain rule of probability to break down $p(y|x)$ into the product of the probability of each word given its prior context, which we can turn into a sum of logs (for an output string of length t):

$$\begin{aligned} score(y) &= \log P(y|x) \\ &= \log\left(P(y_1|x)P(y_2|y_1,x)P(y_3|y_1,y_2,x)...P(y_t|y_1,...,y_{t-1},x)\right) \\ &= \sum_{i=1}^{t} \log P(y_i|y_1,...,y_{i-1},x) \end{aligned} \quad (10.22)$$

Thus at each step, to compute the probability of a partial sentence, we simply add the log probability of the prefix sentence so far to the log probability of generating the next token. Fig. 10.10 shows the scoring for the example sentence shown in Fig. 10.9, using some simple made-up probabilities. Log probabilities are negative or 0, and the max of two log probabilities is the one that is greater (closer to 0).

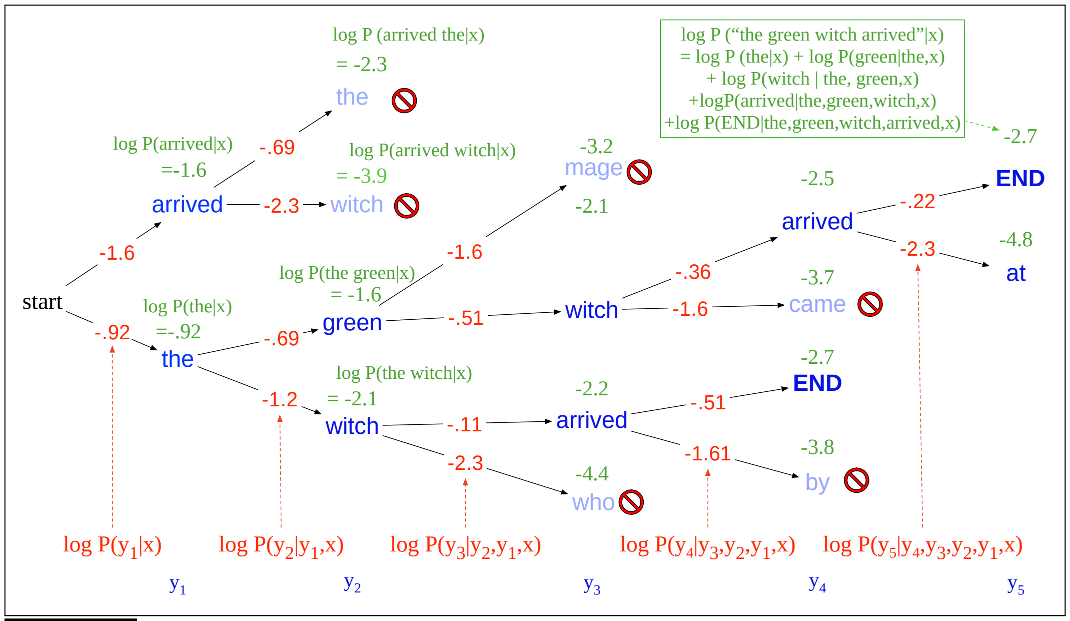

Figure 10.10 Scoring for beam search decoding with a beam width of $k=2$. We maintain the log probability of each hypothesis in the beam by incrementally adding the logprob of generating each next token. Only the top k paths are extended to the next step.

Fig. 10.11 gives the algorithm.

One problem arises from the fact that the completed hypotheses may have different lengths. Because models generally assign lower probabilities to longer strings, a naive algorithm would also choose shorter strings for y. This was not an issue during the earlier steps of decoding; due to the breadth-first nature of beam search all the hypotheses being compared had the same length. The usual solution to this is to apply some form of length normalization to each of the hypotheses, for example simply dividing the negative log probability by the number of words:

$$score(y) = -\log P(y|x) \quad = \quad \frac{1}{T}\sum_{i=1}^{t} -\log P(y_i|y_1,\ldots,y_{i-1},x) \tag{10.23}$$

Beam search is common in language model generation tasks. For tasks like MT, which we'll return to in Chapter 13, we generally use beam widths k between 5 and 10. What do we do with the resulting k hypotheses? In some cases, all we need from our language model generation algorithm is the single best hypothesis, so we can return that. In other cases our downstream application might want to look at all k hypotheses, so we can pass them all (or a subset) to the downstream application with their respective scores.

```
function BEAMDECODE(c, beam_width) returns best paths

  $y_0, h_0 \leftarrow 0$
  path ← ()
  complete_paths ← ()
  state ← (c, $y_0$, $h_0$, path)          ;initial state
  frontier ← ⟨state⟩        ;initial frontier

  while frontier contains incomplete paths and beamwidth > 0
      extended_frontier ← ⟨⟩
      for each state ∈ frontier do
          y ← DECODE(state)
          for each word i ∈ Vocabulary do
              successor ← NEWSTATE(state, i, $y_i$)
              extended_frontier ← ADDTOBEAM(successor, extended_frontier,
                                                          beam_width)

      for each state in extended_frontier do
          if state is complete do
              complete_paths ← APPEND(complete_paths, state)
              extended_frontier ← REMOVE(extended_frontier, state)
              beam_width ← beam_width - 1
      frontier ← extended_frontier

    return completed_paths

function NEWSTATE(state, word, word_prob) returns new state

function ADDTOBEAM(state, frontier, width) returns updated frontier

  if LENGTH(frontier) < width then
      frontier ← INSERT(state, frontier)
  else if SCORE(state) > SCORE(WORSTOF(frontier))
      frontier ← REMOVE(WORSTOF(frontier))
      frontier ← INSERT(state, frontier)
  return frontier
```

Figure 10.11 Beam search decoding.

10.5 Pretraining Large Language Models

TBD: corpora, etc.

10.6 Language Models for Zero-shot Learning

TBD: How to recast NLP tasks as word prediction, simple prompting examples (and then to be continued in Chapter 12)

10.7 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 fine-tuned to downstream tasks, particularly those involving text generation, such as in assistive technologies like web search query completion, or predictive typing for email (Olteanu et al., 2020).

For example, language models can generate toxic language. Gehman et al. (2020) show that many kinds of completely non-toxic prompts can nonetheless lead large language models to output hate speech and abuse. Brown et al. (2020) and Sheng et al. (2019) showed that large language models generate sentences displaying negative attitudes toward minority identities such as being Black or gay.

Indeed, language models are biased in a number of ways by the distributions of their training data. Gehman et al. (2020) shows that large language model training datasets include toxic text scraped from banned sites, such as Reddit communities that have been shut down by Reddit but whose data may still exist in dumps. In addition to problems of toxicity, internet data is disproportionately generated by authors from developed countries, and many large language models trained on data from Reddit, whose authors skew male and young. Such biased population samples likely skew the resulting generation away from the perspectives or topics of underrepresented populations. 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 a tool for generating text for misinformation, phishing, radicalization, and 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.

Finally, there are important privacy issues. Language models, like other machine learning models, can **leak** information about their training data. It is thus possible for an adversary to extract individual training-data phrases from a language model such as an individual person's name, phone number, and address (Henderson et al. 2017, Carlini et al. 2020). This is a problem if large language models are trained on private datasets such as electronic health records (EHRs).

Mitigating all these harms is an important but unsolved research question in NLP. Extra pretraining (Gururangan et al., 2020) on non-toxic subcorpora seems to reduce a language model's tendency to generate toxic language somewhat (Gehman et al., 2020). And analyzing the data used to pretrain large language models is important to understand toxicity and bias in generation, as well as privacy, making it extremely important that language models include **datasheets** (page 16) or **model cards** (page 76) giving full replicable information on the corpora used to train them.

10.8 Summary

This chapter has introduced the transformer and how it can be applied to language problems. 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.
- Common language-based applications for RNNs and transformers include:
  - Probabilistic language modeling: assigning a probability to a sequence, or to the next element of a sequence given the preceding words.
  - Auto-regressive generation using a trained language model.
  - Sequence labeling like part-of-speech tagging, where each element of a sequence is assigned a label.
  - Sequence classification, where an entire text is assigned to a category, as in spam detection, sentiment analysis or topic classification.

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., 2016b). 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).