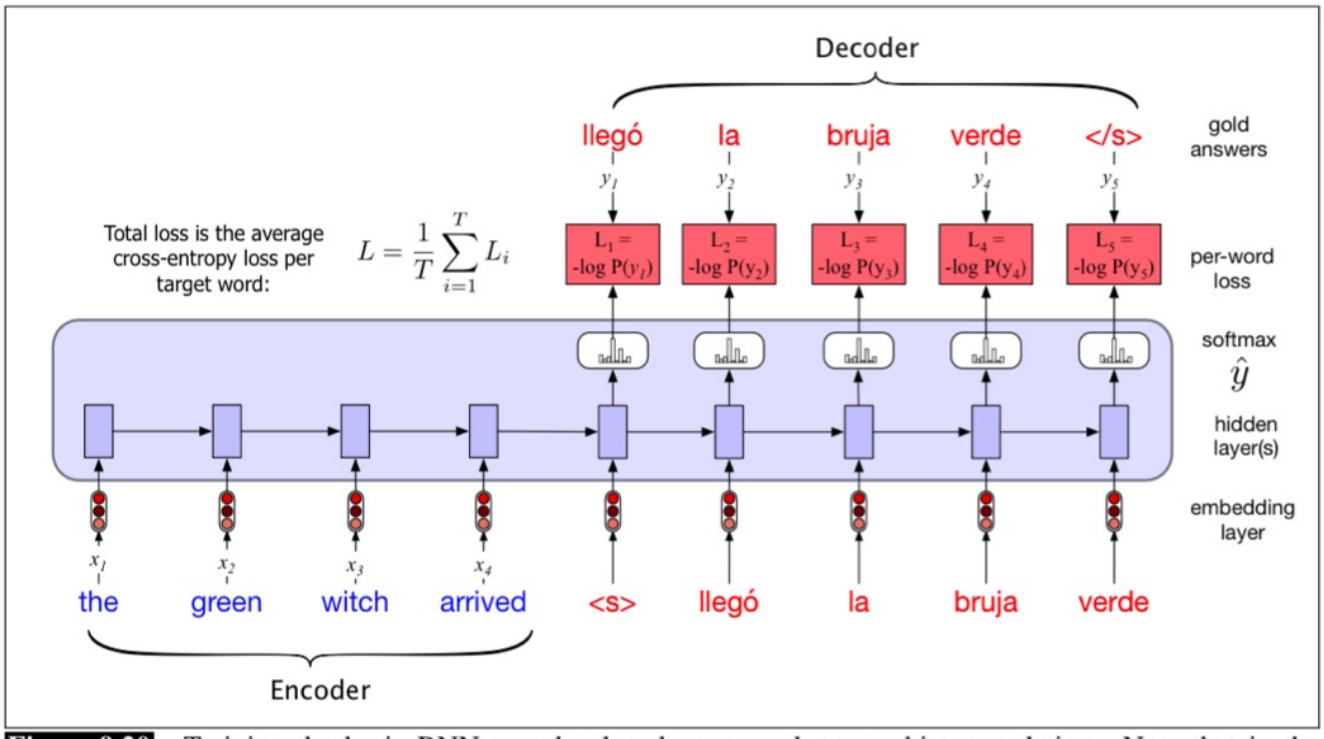


Figure 9.20 Training the basic RNN encoder-decoder approach to machine translation. Note that in the decoder we usually don't propagate the model's softmax outputs $\hat{y}_t$, but use **teacher forcing** to force each input to the correct gold value for training. We compute the softmax output distribution over $\hat{y}$ in the decoder in order to compute the loss at each token, which can then be averaged to compute a loss for the sentence.

context to generate a target text. In the model as we've described it so far, this context vector is h_n, the hidden state of the last (nth) time step of the source text. This final hidden state is thus acting as a **bottleneck**: it must represent absolutely everything about the meaning of the source text, since the only thing the decoder knows about the source text is what's in this context vector (Fig. 9.21). Information at the beginning of the sentence, especially for long sentences, may not be equally well represented in the context vector.

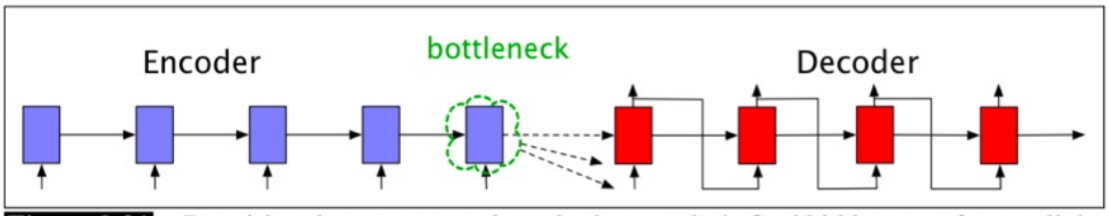


Figure 9.21 Requiring the context c to be only the encoder's final hidden state forces all the information from the entire source sentence to pass through this representational bottleneck.

attention mechanism

The **attention mechanism** is a solution to the bottleneck problem, a way of allowing the decoder to get information from *all* the hidden states of the encoder, not just the last hidden state.

In the attention mechanism, as in the vanilla encoder-decoder model, the context vector $\mathbf{c}$ is a single vector that is a function of the hidden states of the encoder, that is, $\mathbf{c} = f(\mathbf{h}_1^e \ldots \mathbf{h}_n^e)$. Because the number of hidden states varies with the size of the input, we can't use the entire tensor of encoder hidden state vectors directly as the context for the decoder.

The idea of attention is instead to create the single fixed-length vector c by taking a weighted sum of all the encoder hidden states. The weights focus on ('attend to') a particular part of the source text that is relevant for the token the decoder is currently producing. Attention thus replaces the static context vector with one that is dynamically derived from the encoder hidden states, different for each token in

decoding.

This context vector, $\mathbf{c}_i$, is generated anew with each decoding step i and takes all of the encoder hidden states into account in its derivation. We then make this context available during decoding by conditioning the computation of the current decoder hidden state on it (along with the prior hidden state and the previous output generated by the decoder), as we see in this equation (and Fig. 9.22):

$$\mathbf{h}_i^d = g(\hat{y}_{i-1}, \mathbf{h}_{i-1}^d, \mathbf{c}_i) \tag{9.35}$$

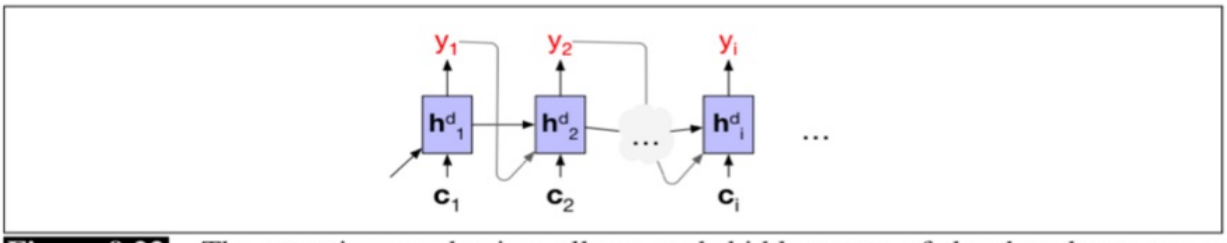


Figure 9.22 The attention mechanism allows each hidden state of the decoder to see a different, dynamic, context, which is a function of all the encoder hidden states.

The first step in computing $\mathbf{c}_i$ is to compute how much to focus on each encoder state, how *relevant* each encoder state is to the decoder state captured in $\mathbf{h}_{i-1}^d$. We capture relevance by computing— at each state i during decoding—a $score(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e)$ for each encoder state j.

dot-product attention

The simplest such score, called **dot-product attention**, implements relevance as similarity: measuring how similar the decoder hidden state is to an encoder hidden state, by computing the dot product between them:

$$\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e) = \mathbf{h}_{i-1}^d \cdot \mathbf{h}_j^e \tag{9.36}$$

The score that results from this dot product is a scalar that reflects the degree of similarity between the two vectors. The vector of these scores across all the encoder hidden states gives us the relevance of each encoder state to the current step of the decoder.

To make use of these scores, we'll normalize them with a softmax to create a vector of weights, α_{ij}, that tells us the proportional relevance of each encoder hidden state j to the prior hidden decoder state, h_{i-1}^d.

$$\begin{aligned}\alpha_{ij} &= \text{softmax}(\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e) \;\; \forall j \in e) \\ &= \frac{\exp(\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e)}{\sum_k \exp(\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_k^e))}\end{aligned} \tag{9.37}$$

Finally, given the distribution in α, we can compute a fixed-length context vector for the current decoder state by taking a weighted average over all the encoder hidden states.

$$\mathbf{c}_i = \sum_j \alpha_{ij} \mathbf{h}_j^e \tag{9.38}$$

With this, we finally have a fixed-length context vector that takes into account information from the entire encoder state that is dynamically updated to reflect the needs of the decoder at each step of decoding. Fig. 9.23 illustrates an encoder-decoder network with attention, focusing on the computation of one context vector $\mathbf{c}_i$.

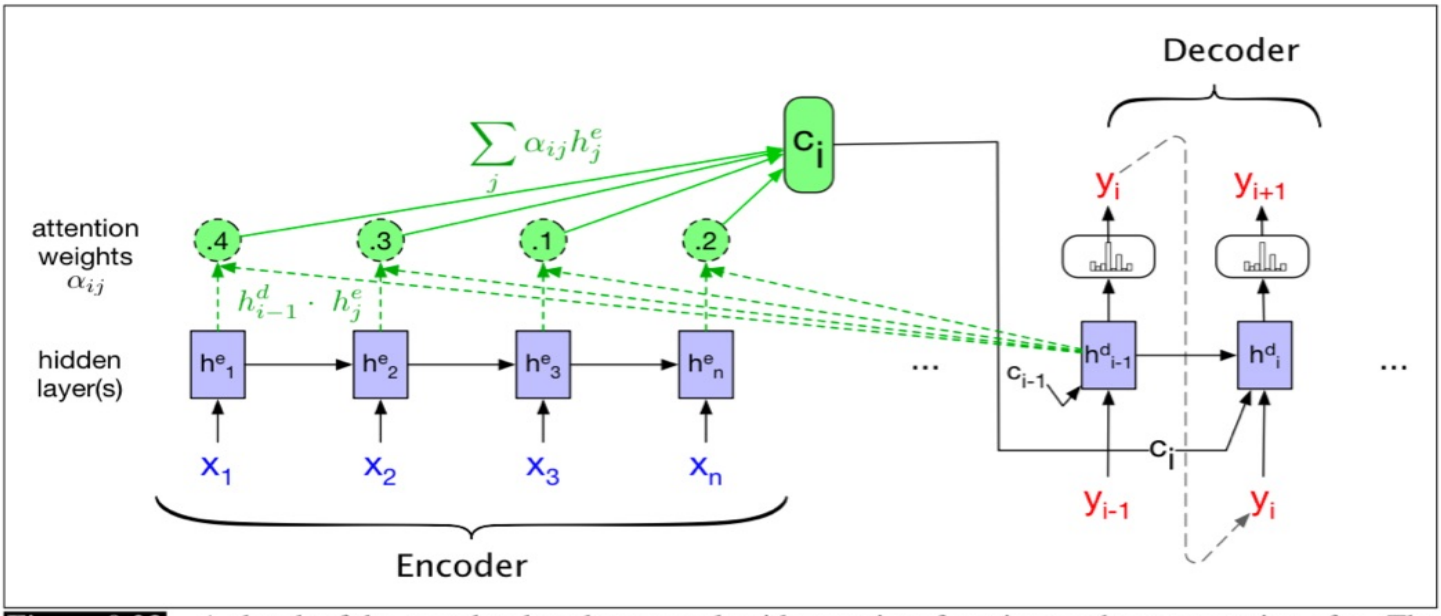


Figure 9.23 A sketch of the encoder-decoder network with attention, focusing on the computation of $\mathbf{c}_i$. The context value $\mathbf{c}_i$ is one of the inputs to the computation of $\mathbf{h}^d_i$. It is computed by taking the weighted sum of all the encoder hidden states, each weighted by their dot product with the prior decoder hidden state $\mathbf{h}^d_{i-1}$.

It's also possible to create more sophisticated scoring functions for attention models. Instead of simple dot product attention, we can get a more powerful function that computes the relevance of each encoder hidden state to the decoder hidden state by parameterizing the score with its own set of weights, $\mathbf{W}_s$.

$$\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j) = \mathbf{h}^d_{t-1} \mathbf{W}_s \mathbf{h}^e_j$$

The weights W_s, which are then trained during normal end-to-end training, give the network the ability to learn which aspects of similarity between the decoder and encoder states are important to the current application. This bilinear model also allows the encoder and decoder to use different dimensional vectors, whereas the simple dot-product attention requires that the encoder and decoder hidden states have the same dimensionality.

We'll return to the concept of attention when we defined the transformer architecture in Chapter 10, which is based on a slight modification of attention called **self-attention**.

9.9 Summary

This chapter has introduced the concepts of recurrent neural networks and how they can be applied to language problems. Here's a summary of the main points that we covered:

- In simple Recurrent Neural Networks sequences are processed one element at a time, with the output of each neural unit at time t based both on the current input at t and the hidden layer from time $t-1$.
- RNNs can be trained with a straightforward extension of the backpropagation algorithm, known as **backpropagation through time** (BPTT).
- Simple recurrent networks fail on long inputs because of problems like **vanishing gradients**; instead modern systems use more complex gated architec-

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

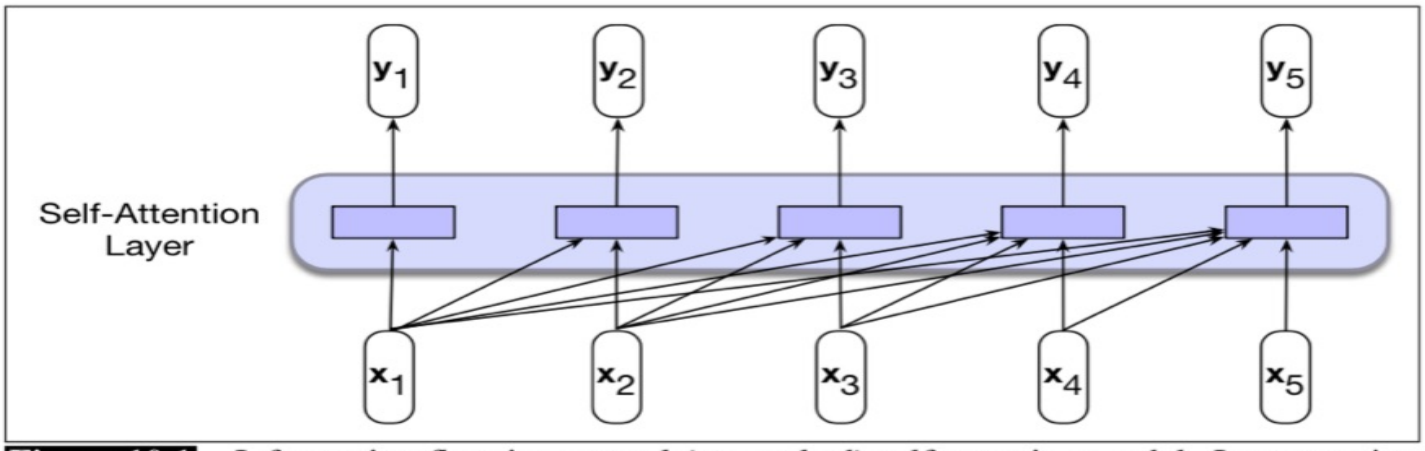


Figure 10.1 Information flow in a causal (or masked) self-attention model. In processing each element of the sequence, the model attends to all the inputs up to, and including, the current one. Unlike RNNs, the computations at each time step are independent of all the other steps and therefore can be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

query
- 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**.

key
- In its role as *a preceding input* being compared to the current focus of attention. We'll refer to this role as a **key**.

value
- And finally, as a **value** used to compute the output for the current focus of attention.

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.

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

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 $\mathbf{v}$.

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

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

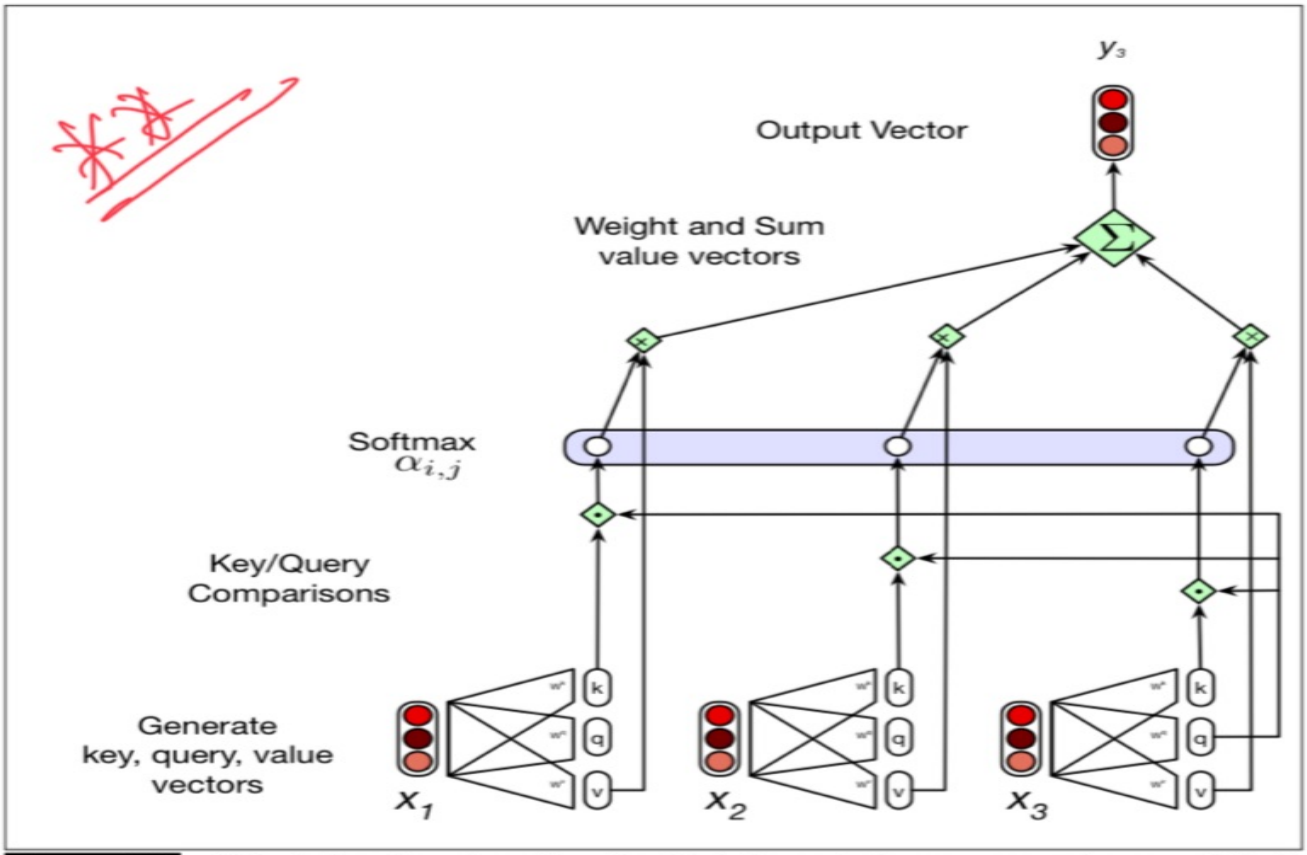


Figure 10.2 Calculating the value of $\mathbf{y}_3$, the third element of a sequence using causal (left-to-right) self-attention.

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:

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

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).

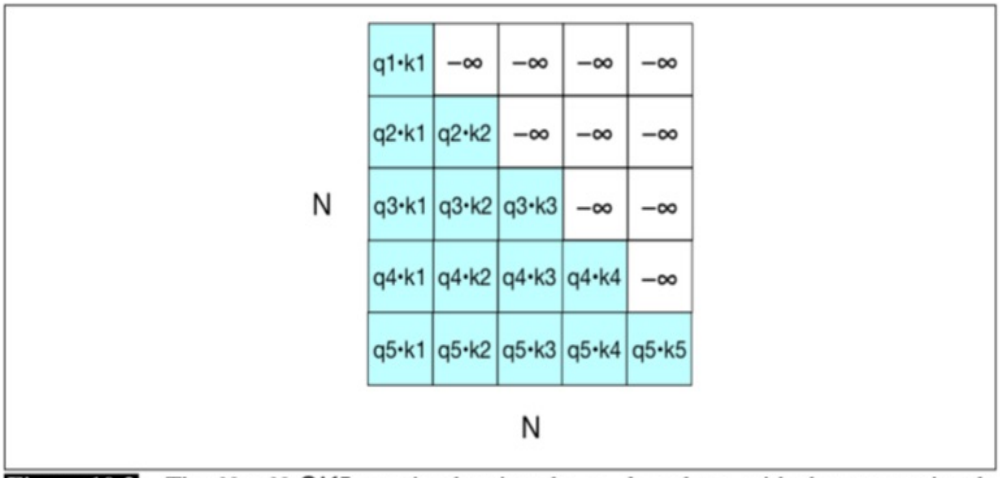


Figure 10.3 The $N \times N$ $\mathbf{QK}^\intercal$ matrix showing the $q_i \cdot k_j$ values, with the upper-triangle portion of the comparisons matrix zeroed out (set to $-\infty$, which the softmax will turn to zero).

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.

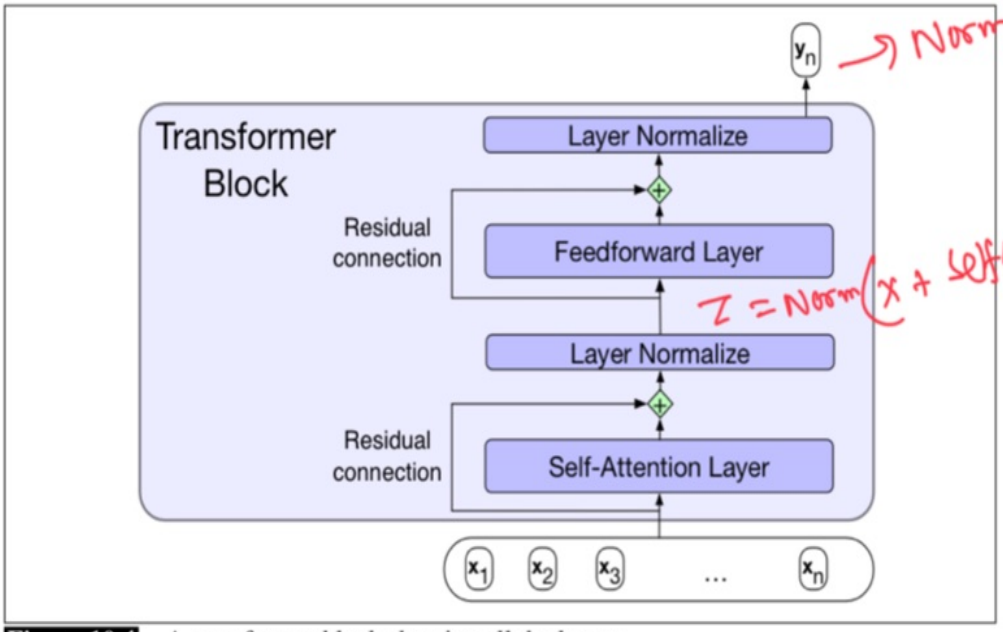


Figure 10.4 A transformer block showing all the layers.

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.

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

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

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

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.

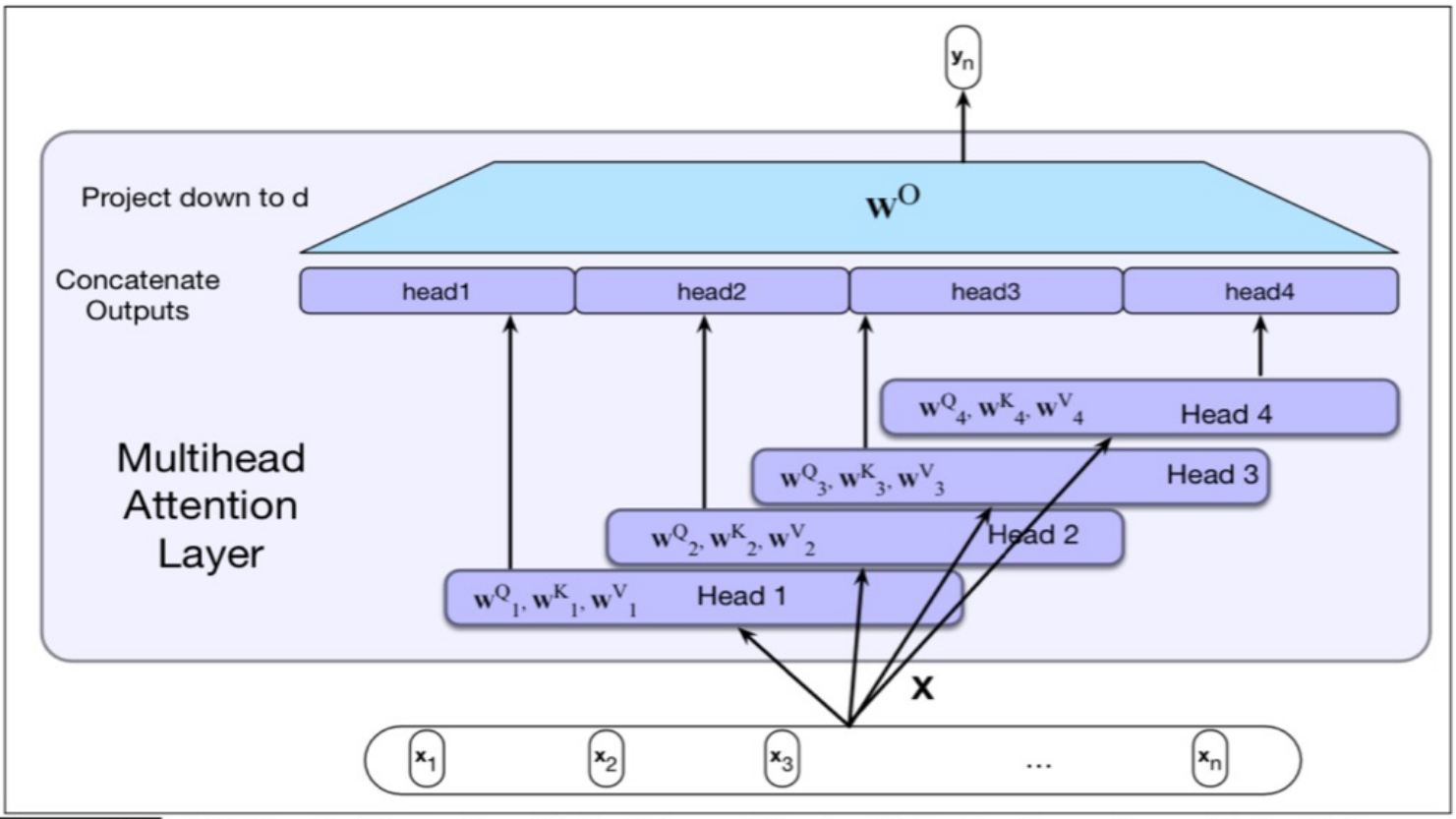


Figure 10.5 Multihead self-attention: Each of the multihead self-attention layers is provided with its own set of key, query and value weight matrices. The outputs from each of the layers are concatenated and then projected down to d, thus producing an output of the same size as the input so layers can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

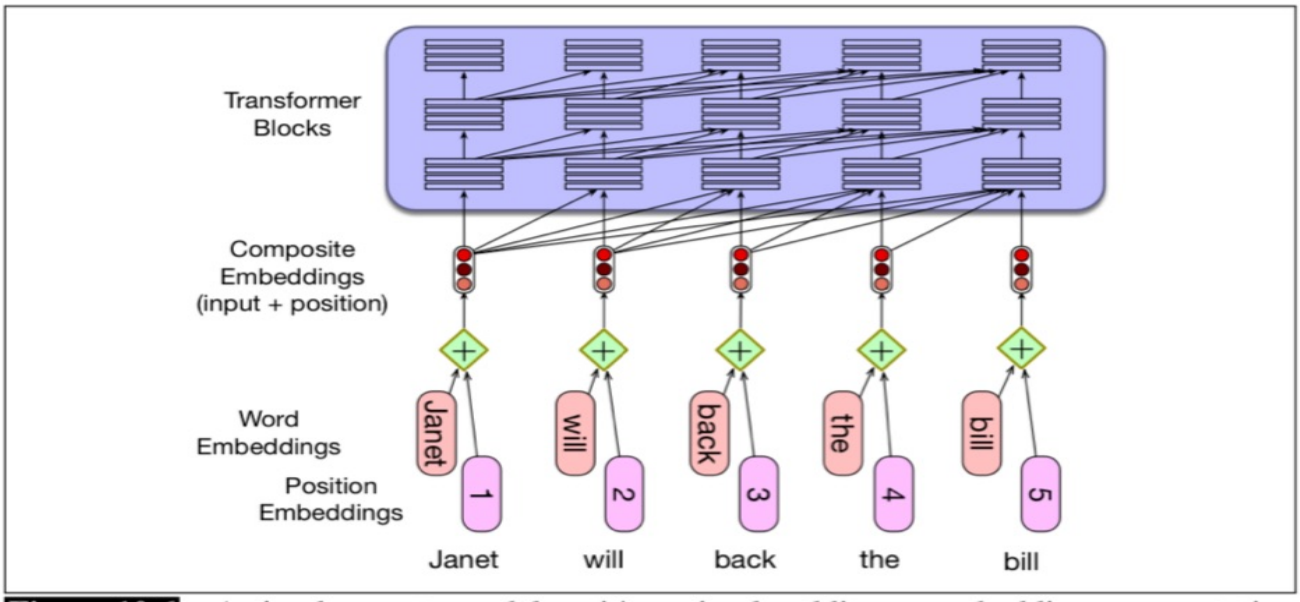


Figure 10.6 A simple way to model position: simply adding an embedding representation of the absolute position to the input word embedding to produce a new embedding of the same dimenionality.

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.

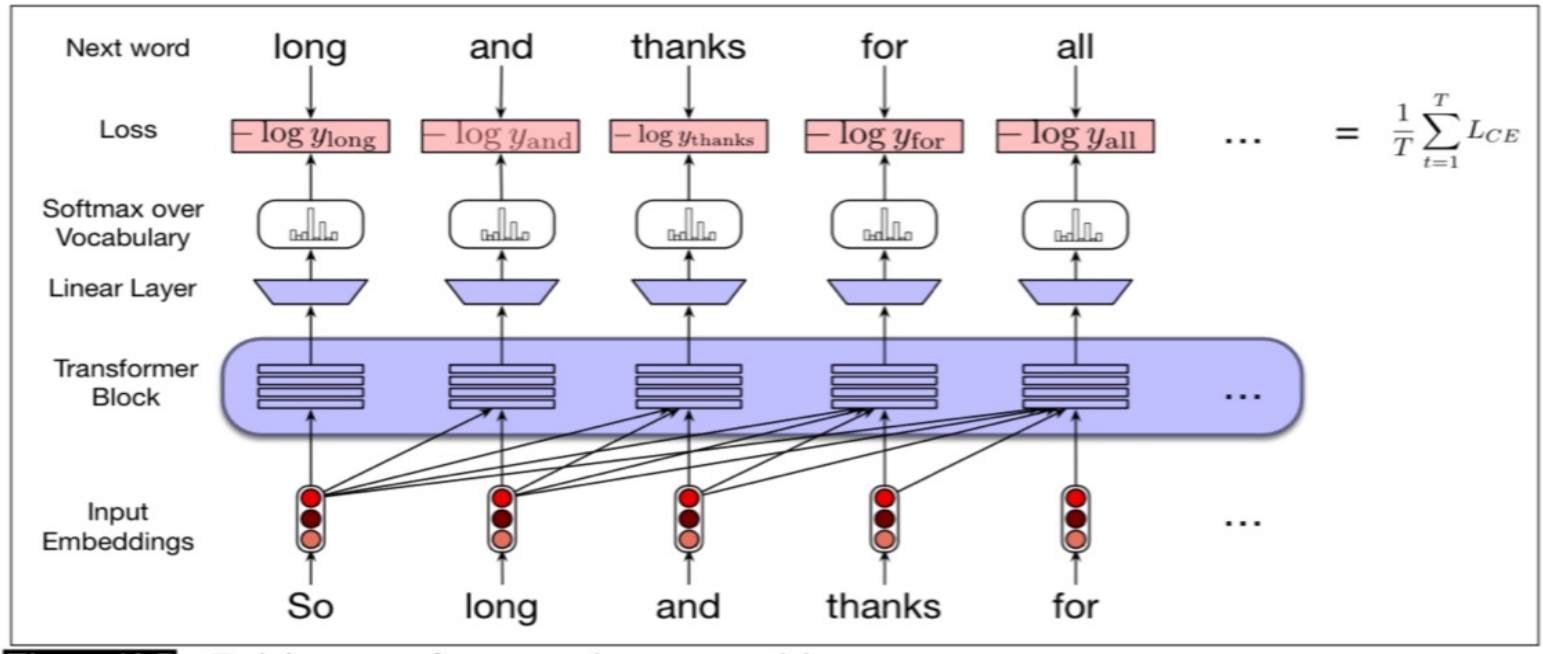


Figure 10.7 Training a transformer as a language model.

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

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.

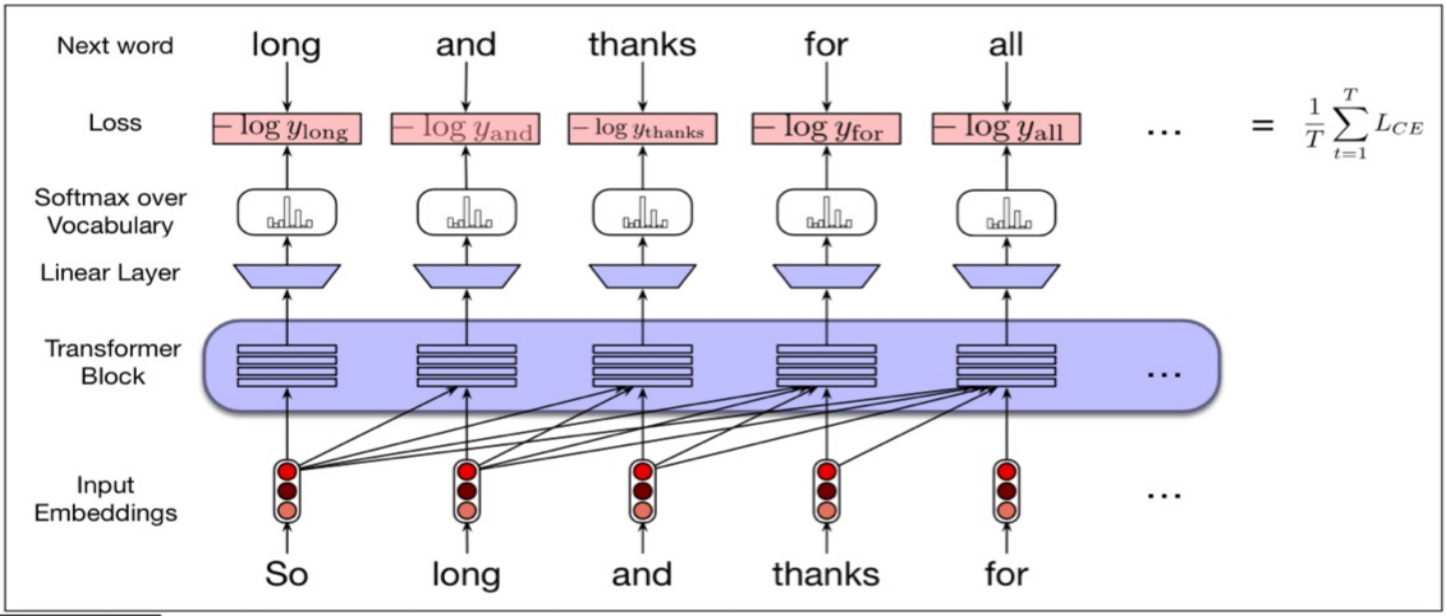


Figure 10.7 Training a transformer as a language model.

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):

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

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!

Congratulations! You passed!
TO PASS 80% or higher

Keep Learning

GRADE
100%

Recurrent Neural Networks

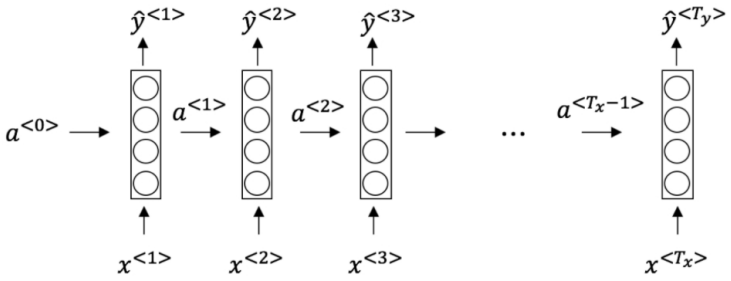
LATEST SUBMISSION GRADE
100%

1. Suppose your training examples are sentences (sequences of words). Which of the following refers to the j^{th} word in the i^{th} training example? **1 / 1 point**

- (●) $x^{(i)<j>}$
- ( ) $x^{<i>(j)}$
- ( ) $x^{(j)<i>}$
- ( ) $x^{<j>(i)}$

Correct
We index into the i^{th} row first to get the i^{th} training example (represented by parentheses), then the j^{th} column to get the j^{th} word (represented by the brackets).

2. Consider this RNN: **1 / 1 point**

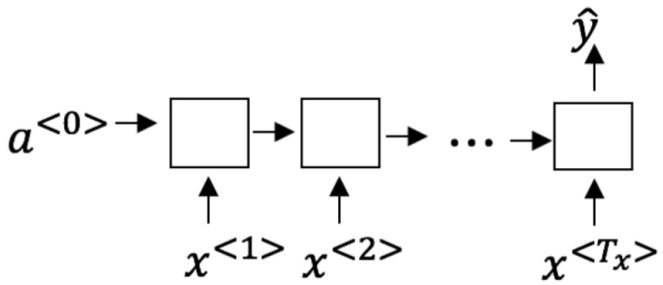


This specific type of architecture is appropriate when:

- (●) $T_x = T_y$
- ( ) $T_x < T_y$
- ( ) $T_x > T_y$
- ( ) $T_x = 1$

Correct
It is appropriate when every input should be matched to an output.

3. To which of these tasks would you apply a many-to-one RNN architecture? (Check all that apply). **1 / 1 point**



- [ ] Speech recognition (input an audio clip and output a transcript)
- [x] Sentiment classification (input a piece of text and output a 0/1 to denote positive or negative sentiment)

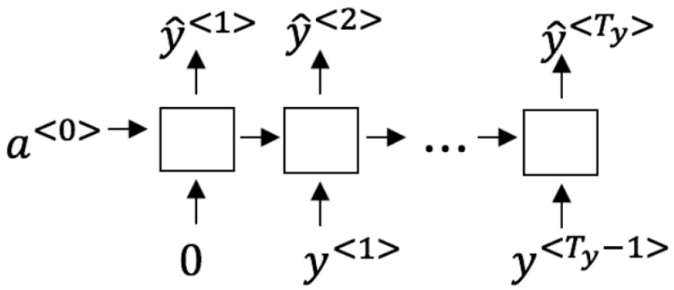
Correct
Correct!

- [ ] Image classification (input an image and output a label)
- [x] Gender recognition from speech (input an audio clip and output a label indicating the speaker's gender)

Correct

Correct!

4. You are training this RNN language model. 1 / 1 point



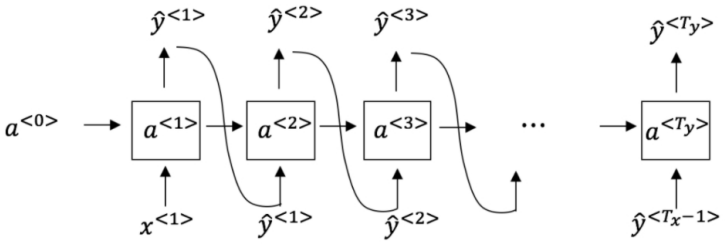
At the t^{th} time step, what is the RNN doing? Choose the best answer.

- Estimating $P(y^{<1>}, y^{<2>}, \ldots, y^{<t-1>})$
- Estimating $P(y^{<t>})$
- (selected) Estimating $P(y^{<t>} \mid y^{<1>}, y^{<2>}, \ldots, y^{<t-1>})$
- Estimating $P(y^{<t>} \mid y^{<1>}, y^{<2>}, \ldots, y^{<t>})$

Correct

Yes, in a language model we try to predict the next step based on the knowledge of all prior steps.

5. You have finished training a language model RNN and are using it to sample random sentences, as follows: 1 / 1 point



What are you doing at each time step t?

- (i) Use the probabilities output by the RNN to pick the highest probability word for that time-step as $\hat{y}^{<t>}$. (ii) Then pass the ground-truth word from the training set to the next time-step.
- (i) Use the probabilities output by the RNN to randomly sample a chosen word for that time-step as $\hat{y}^{<t>}$. (ii) Then pass the ground-truth word from the training set to the next time-step.
- (i) Use the probabilities output by the RNN to pick the highest probability word for that time-step as $\hat{y}^{<t>}$. (ii) Then pass this selected word to the next time-step.
- (selected) (i) Use the probabilities output by the RNN to randomly sample a chosen word for that time-step as $\hat{y}^{<t>}$. (ii) Then pass this selected word to the next time-step.

Correct

Yes!

6. You are training an RNN, and find that your weights and activations are all taking on the value of NaN ("Not a Number"). Which of these is the most likely cause of this problem? 1 / 1 point

- Vanishing gradient problem.
- (selected) Exploding gradient problem.
- ReLU activation function g(.) used to compute g(z), where z is too large.
- Sigmoid activation function g(.) used to compute g(z), where z is too large.

Correct

7. Suppose you are training a LSTM. You have a 10000 word vocabulary, and are using an LSTM with 100-dimensional activations $a^{<t>}$. What is the dimension of Γ_u at each time step? 1 / 1 point

- 1
- (selected) 100
- 300
- 10000

✓ **Correct**

Correct, Γ_u is a vector of dimension equal to the number of hidden units in the LSTM.

8. Here're the update equations for the GRU. **1 / 1 point**

GRU

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$$

$$a^{<t>} = c^{<t>}$$

Alice proposes to simplify the GRU by always removing the Γ_u. I.e., setting Γ_u = 1. Betty proposes to simplify the GRU by removing the Γ_r. I. e., setting Γ_r = 1 always. Which of these models is more likely to work without vanishing gradient problems even when trained on very long input sequences?

- ○ Alice's model (removing Γ_u), because if $\Gamma_r \approx 0$ for a timestep, the gradient can propagate back through that timestep without much decay.
- ○ Alice's model (removing Γ_u), because if $\Gamma_r \approx 1$ for a timestep, the gradient can propagate back through that timestep without much decay.
- ◉ Betty's model (removing Γ_r), because if $\Gamma_u \approx 0$ for a timestep, the gradient can propagate back through that timestep without much decay.
- ○ Betty's model (removing Γ_r), because if $\Gamma_u \approx 1$ for a timestep, the gradient can propagate back through that timestep without much decay.

✓ **Correct**

Yes. For the signal to backpropagate without vanishing, we need $c^{<t>}$ to be highly dependant on $c^{<t-1>}$.

9. Here are the equations for the GRU and the LSTM: **1 / 1 point**

GRU

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$$

$$a^{<t>} = c^{<t>}$$

LSTM

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$$

$$\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$

$$a^{<t>} = \Gamma_o * c^{<t>}$$

From these, we can see that the Update Gate and Forget Gate in the LSTM play a role similar to _____ and _____ in the GRU. What should go in the the blanks?

- ◉ Γ_u and $1 - \Gamma_u$
- ○ Γ_u and Γ_r
- ○ $1 - \Gamma_u$ and Γ_u
- ○ Γ_r and Γ_u

✓ **Correct**

Yes, correct!

10. You have a pet dog whose mood is heavily dependent on the current and past few days' weather. You've collected data for the past 365 days on the weather, which you represent as a sequence as $x^{<1>}, \ldots, x^{<365>}$. You've also collected data on your dog's mood, which you represent as $y^{<1>}, \ldots, y^{<365>}$. You'd like to build a model to map from $x \rightarrow y$. Should you use a Unidirectional RNN or Bidirectional RNN for this problem? **1 / 1 point**

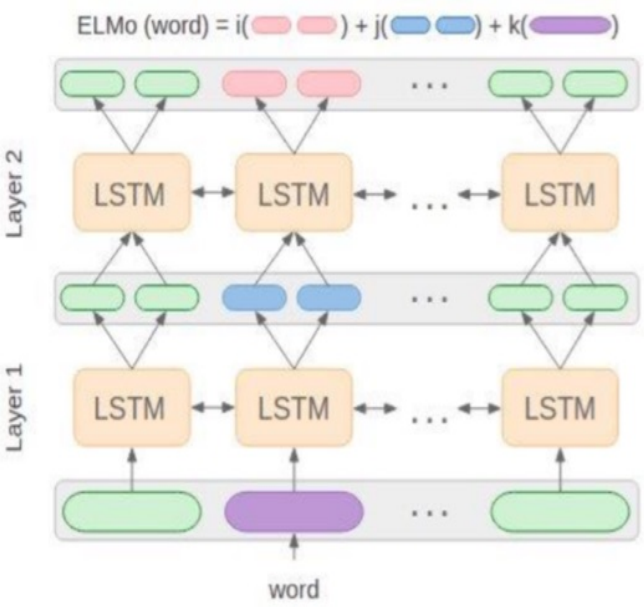
- ○ Bidirectional RNN, because this allows the prediction of mood on day t to take into account more information.
- ○ Bidirectional RNN, because this allows backpropagation to compute more accurate gradients.
- ◉ Unidirectional RNN, because the value of $y^{<t>}$ depends only on $x^{<1>}, \ldots, x^{<t>}$, but not on $x^{<t+1>}, \ldots, x^{<365>}$
- ○ Unidirectional RNN, because the value of $y^{<t>}$ depends only on $x^{<t>}$, and not other days' weather.

✓ **Correct**

Yes!

ELMO Embeddings from Language Model

Embeddings from Language Models (ELMO)

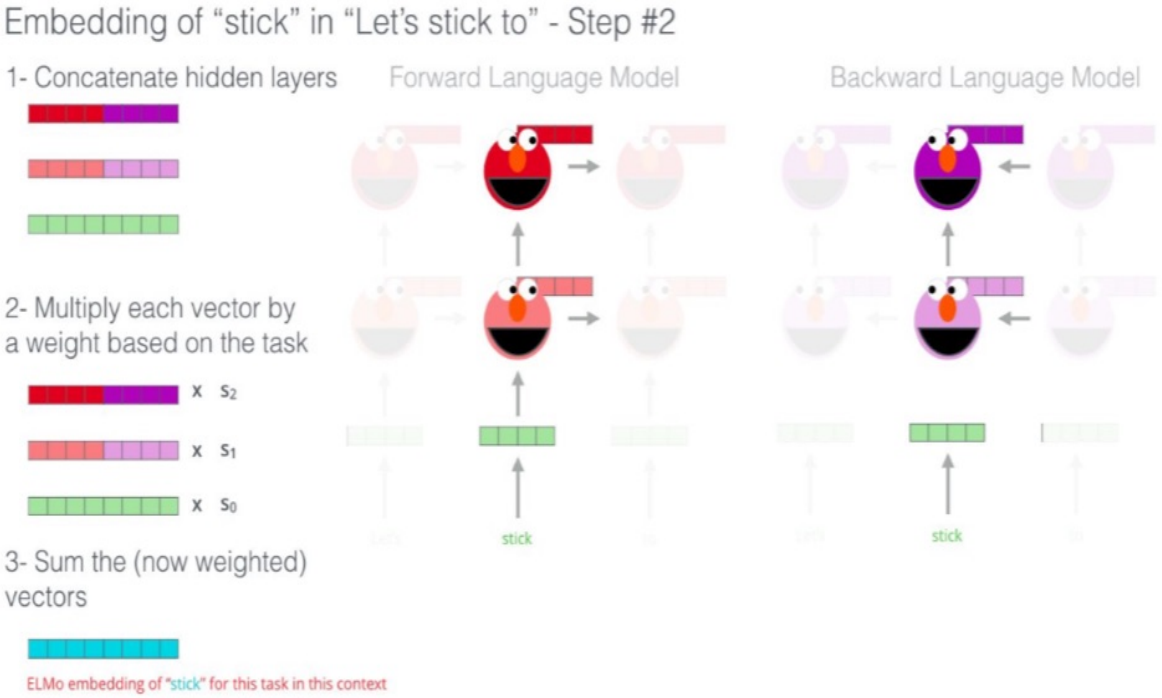


- Stacked LSTMs
- Each LSTM layer i gives a representation h_i of the token t_i
- Final representation h is a combination of the representations from different layers
  - $h=f(h_0,h_1,\ldots,h_L)$
- How to combine these representations?
- How to use in target tasks?

Ref: Peters, Matthew E., et al. "Deep contextualized word representations." NAACL 2018.

Figure from: Biesialska, K.et al. (2020). Sentiment analysis with contextual embeddings and self-attention. In *International Symposium on Methodologies for Intelligent Systems* (pp. 32-41).

Combining Representations from ELMO Layers



Picture courtesy: The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning) by Jay Alammar

- Embeddings from Language Models: **ELMo**
- Learn word embeddings through building *bidirectional language models* (biLMs)
  - biLMs consist of forward and backward LMs
    - Forward: $p(t_1, t_2, \ldots, t_N) = \prod_{k=1}^{N} p(t_k \mid t_1, t_2, \ldots, t_{k-1})$
    - Backward: $p(t_1, t_2, \ldots, t_N) = \prod_{k=1}^{N} p(t_k \mid t_{k+1}, t_{k+2}, \ldots, t_N)$

A biLM combines both forward & backward LM. our formulation jointly maximizes the log likelihood of the forward & backward direction:

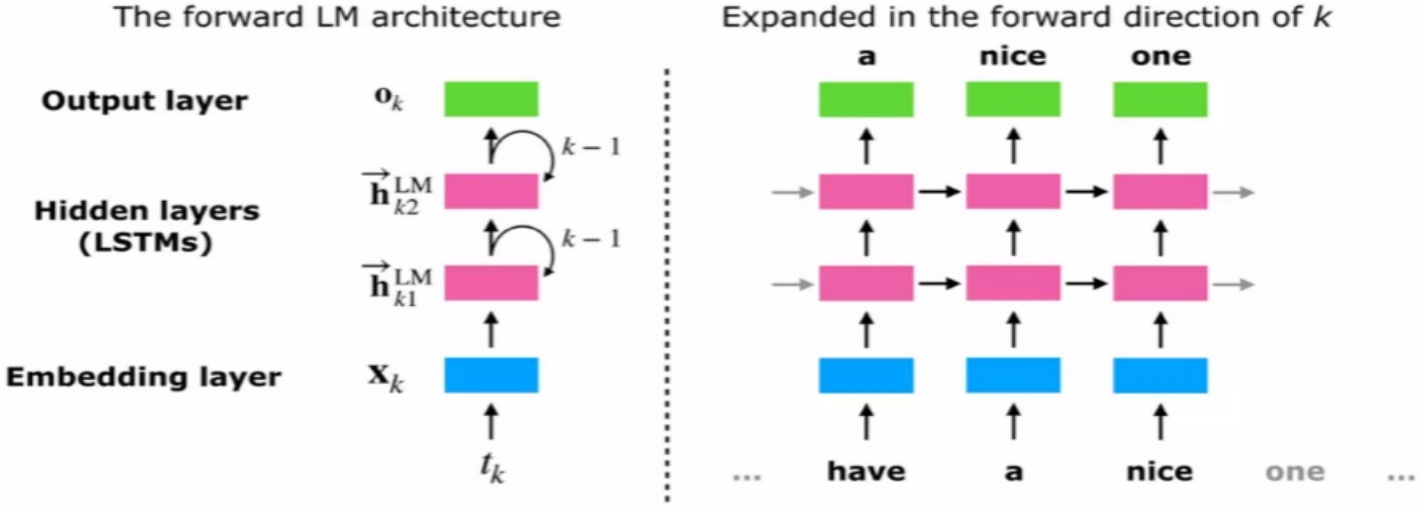
$$\sum_{k=1}^{N} \left(\log p(t_k \mid t_1, \ldots, t_{k-1}, \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s) + \log p(t_{k+1}, \ldots, t_N, \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s) \right)$$

Θ_s → parameter for softmax layer.

35

Method

With long short term memory (LSTM) network, predicting the next words in both directions to build biLMs



36