$$\frac{\partial L_t}{\partial w_{nn}} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_k} \cdot \frac{\partial h_k}{\partial w_{nn}}$$

↓ It has a chain rule in itself.

$$\frac{\partial L_t}{\partial w_{nn}} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \left(\prod_{j=k}^{t-1} \frac{\partial h_{j+1}}{\partial h_j} \right) \frac{\partial h_k}{\partial w_{nn}}$$

$\frac{\partial h_3}{\partial h_1} = \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1}$

where,

$$\prod_{j=k}^{t-1} \cdot \frac{\partial h_{j+1}}{\partial h_j} = \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_{k+1}}{\partial h_k}$$

that because we are taking the derivative of a vector function with respect to a vector, the result is a matrix (called the Jacobian matrix) whose elements are all the pointwise derivatives.

finally, putting it all together,

$$\frac{\partial L}{\partial w_{nn}} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial w_{nn}}$$

$$\frac{\partial L_t}{\partial w_{xh}} = \frac{\partial L_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial w_{xh}}$$

h_t is depends on both w_{xh} & h_{t-1}.

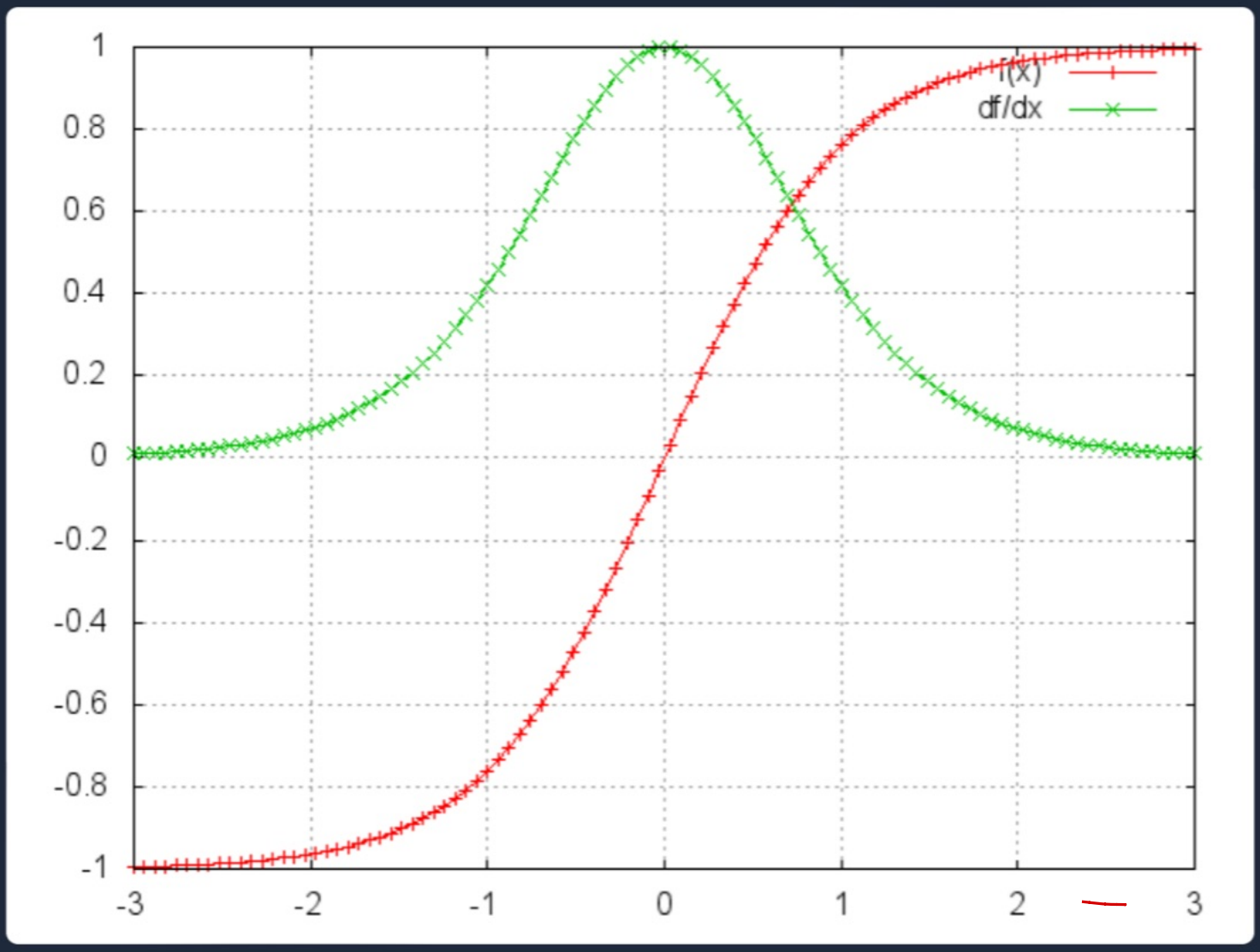
$$\frac{\partial L_t}{\partial w_{xh}} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_k} \cdot \frac{\partial h_k}{\partial w_{xh}}$$

$$\frac{\partial L}{\partial w_{xh}} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_k} \cdot \frac{\partial h_k}{\partial w_{xh}}$$

The Vanishing Gradient Problem

RNNs have difficulties learning long-range dependencies – interactions between words that are several steps apart. That's problematic because the meaning of an English sentence is often determined by words that aren't very close: "The man who wore a wig on his head went inside". The sentence is really about a man going inside, not about the wig. But it's unlikely that a plain RNN would be able capture such information. To understand why, let's take a closer look at the gradient we calculated above:

It turns out (I won't prove it here but this paper goes into detail) that the 2-norm, which you can think of it as an absolute value, of the above Jacobian matrix has an upper bound of 1. This makes intuitive sense because our $tanh$ (or sigmoid) activation function maps all values into a range between -1 and 1, and the derivative is bounded by 1 (1/4 in the case of sigmoid) as well:

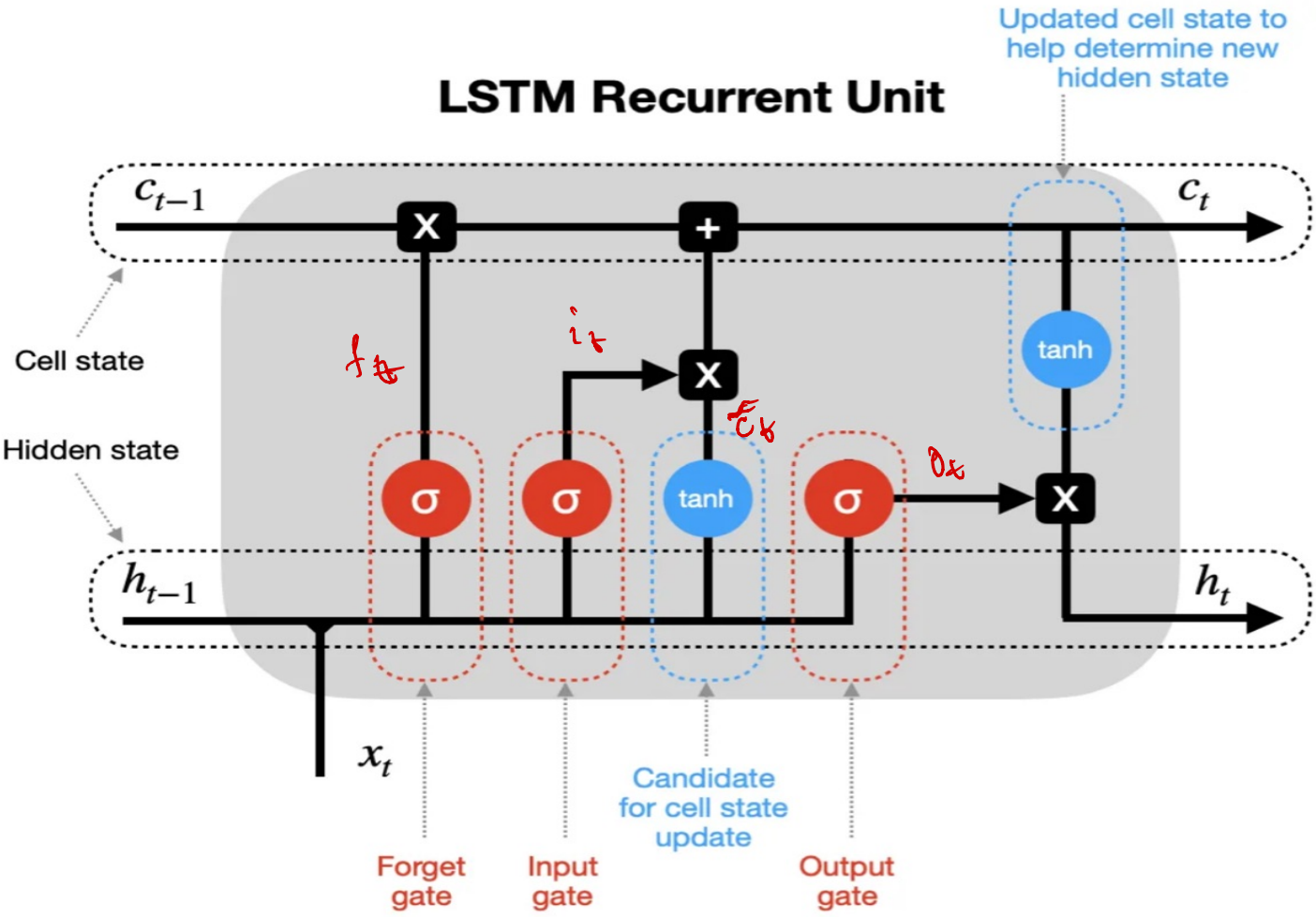


You can see that the $\tanh$ and sigmoid functions have derivatives of 0 at both ends. They approach a flat line. When this happens we say the corresponding neurons are saturated. They have a zero gradient and drive other gradients in previous layers towards 0. Thus, with small values in the matrix and multiple matrix multiplications ($t - k$ in particular) the gradient values are shrinking exponentially fast, eventually vanishing completely after a few time steps. Gradient contributions from "far away" steps become zero, and the state at those steps doesn't contribute to what you are learning: You end up not learning long-range dependencies. Vanishing gradients aren't exclusive to RNNs. They also happen in deep Feedforward Neural Networks. It's just that RNNs tend to be very deep (as deep as the sentence length in our case), which makes the problem common.

It is easy to imagine that, depending on our activation functions and network parameters, we could get exploding instead of vanishing gradients if the values of the Jacobian matrix are large. Indeed, that's called the *exploding gradient problem*. The reason that vanishing gradients have received more attention than exploding gradients is two-fold. For one, exploding gradients are obvious. Your gradients will become NaN (not a number) and your program will crash. Secondly, clipping the gradients at a pre-defined threshold (as discussed in this paper) is a simple and effective solution to exploding gradients. Vanishing gradients are more problematic because it's not obvious when they occur or how to deal with them.

Fortunately, there are a few ways to combat the vanishing gradient problem. Proper initialization of the W matrix can reduce the effect of vanishing gradients. So can regularization. A more preferred solution is to use ReLU instead of $tanh$ or sigmoid activation functions. The ReLU derivative is a constant of either 0 or 1, so it isn't as likely to suffer from vanishing gradients. An even more popular solution is to use Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) architectures. LSTMs were first proposed in 1997 and are the perhaps most widely used models in NLP today. GRUs, first proposed in 2014, are simplified versions of LSTMs. Both of these RNN architectures were explicitly designed to deal with vanishing gradients and efficiently learn long-range dependencies. We'll cover them in the next part of this tutorial.

LSTM



h_{t-1} - hidden state at previous timestep t-1 (short-term memory)

c_{t-1} - cell state at previous timestep t-1 (long-term memory)

x_t - input vector at current timestep t

h_t - hidden state at current timestep t

c_t - cell state at current timestep t

X - vector pointwise multiplication

\+ - vector pointwise addition

tanh - tanh activation function

σ - sigmoid activation function

T - concatenation of vectors

- states
- gates
- updates

9.5 The LSTM

In practice, it is quite difficult to train RNNs for tasks that require a network to make use of information distant from the current point of processing. Despite having access to the entire preceding sequence, the information encoded in hidden states tends to be fairly local, more relevant to the most recent parts of the input sequence and recent decisions. Yet distant information is critical to many language applications. Consider the following example in the context of language modeling.

(9.19) The flights the airline was cancelling were full.

Assigning a high probability to *was* following *airline* is straightforward since *airline* provides a strong local context for the singular agreement. However, assigning an appropriate probability to *were* is quite difficult, not only because the plural *flights* is quite distant, but also because the intervening context involves singular constituents. Ideally, a network should be able to retain the distant information about plural *flights* until it is needed, while still processing the intermediate parts of the sequence correctly.

One reason for the inability of RNNs to carry forward critical information is that the hidden layers, and, by extension, the weights that determine the values in the hidden layer, are being asked to perform two tasks simultaneously: provide information useful for the current decision, and updating and carrying forward information required for future decisions.

A second difficulty with training RNNs arises from the need to backpropagate the error signal back through time. Recall from Section 9.1.2 that the hidden layer at time t contributes to the loss at the next time step since it takes part in that calculation. As a result, during the backward pass of training, the hidden layers are subject to repeated multiplications, as determined by the length of the sequence. A frequent result of this process is that the gradients are eventually driven to zero, a situation called the **vanishing gradients** problem.

vanishing gradients

To address these issues, more complex network architectures have been designed to explicitly manage the task of maintaining relevant context over time, by enabling the network to learn to forget information that is no longer needed and to remember information required for decisions still to come.

The most commonly used such extension to RNNs is the **long short-term memory** (LSTM) network (Hochreiter and Schmidhuber, 1997). LSTMs divide the context management problem into two subproblems: removing information no longer needed from the context, and adding information likely to be needed for later decision making. The key to solving both problems is to learn how to manage this context rather than hard-coding a strategy into the architecture. LSTMs accomplish this by first adding an explicit context layer to the architecture (in addition to the usual recurrent hidden layer), and through the use of specialized neural units that make use of *gates* to control the flow of information into and out of the units that comprise the network layers. These gates are implemented through the use of additional weights that operate sequentially on the input, and previous hidden layer, and previous context layers.

long short-term memory

The gates in an LSTM share a common design pattern; each consists of a feed-forward layer, followed by a sigmoid activation function, followed by a pointwise multiplication with the layer being gated. The choice of the sigmoid as the activation function arises from its tendency to push its outputs to either 0 or 1. Combining this with a pointwise multiplication has an effect similar to that of a binary mask. Values

Let's go through the simplified diagram (weights and biases not shown) to learn how LSTM recurrent unit processes information.

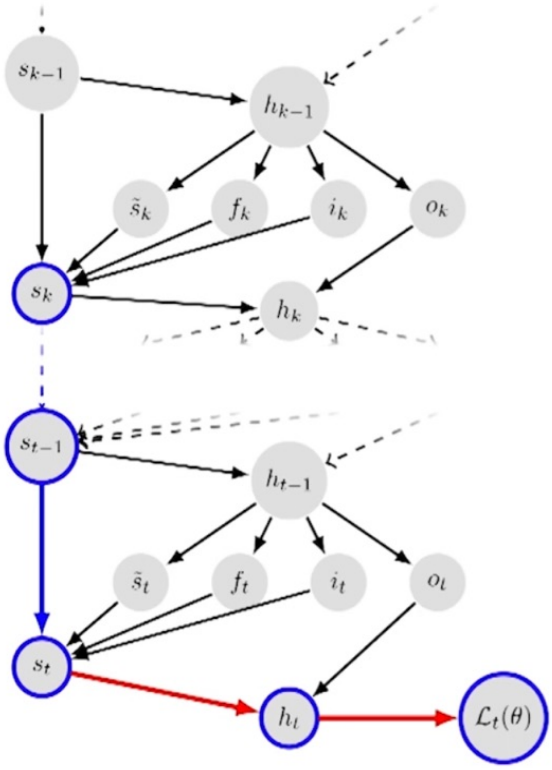
1. **Hidden state & new inputs** — hidden state from a previous timestep (h_t-1) and the input at a current timestep (x_t) are combined before passing copies of it through various gates.
2. **Forget gate** — this gate controls what information should be forgotten. Since the sigmoid function ranges between 0 and 1, it sets which values in the cell state should be discarded (multiplied by 0), remembered (multiplied by 1), or partially remembered (multiplied by some value between 0 and 1).
3. **Input gate** helps to identify important elements that need to be added to the cell state. Note that the results of the input gate get multiplied by the cell state candidate, with only the information deemed important by the input gate being added to the cell state.
4. **Update cell state** —first, the previous cell state (c_t-1) gets multiplied by the results of the forget gate. Then we add new information from [input gate × cell state candidate] to get the latest cell state (c_t).
5. **Update hidden state** — the last part is to update the hidden state. The latest cell state (c_t) is passed through the tanh activation function and multiplied by the results of the output gate.

Finally, the latest cell state (c_t) and the hidden state (h_t) go back into the recurrent unit, and the **process repeats at timestep t+1**. The loop continues until we reach the end of the sequence.

$$
\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
\tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \\
C_t &= f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \\
h_t &= o_t \cdot \tanh(C_t)
\end{aligned}
$$

Where:

- x_t is the input at time step t,
- h_{t-1} is the hidden state from the previous time step,
- W_f, W_i, W_o, W_C are weight matrices,
- b_f, b_i, b_o, b_C are bias vectors,
- σ is the sigmoid activation function,
- $\tanh$ is the hyperbolic tangent activation function.



- We now return back to our full expression for t_0:

$$\begin{aligned} t_0 &= \frac{\partial \mathcal{L}_t(\theta)}{\partial h_t}\frac{\partial h_t}{\partial s_t}\frac{\partial s_t}{\partial s_{t-1}}\cdots\frac{\partial s_{k+1}}{\partial s_k} \\ &= \mathcal{L}'_t(h_t).\mathcal{D}(o_t \odot \sigma'(s_t))\mathcal{D}(f_t)\ldots\mathcal{D}(f_{k+1}) \\ &= \mathcal{L}'_t(h_t).\mathcal{D}(o_t \odot \sigma'(s_t))\mathcal{D}(f_t \odot \ldots \odot f_{k+1}) \\ &= \mathcal{L}'_t(h_t).\mathcal{D}(o_t \odot \sigma'(s_t))\mathcal{D}(\odot_{i=k+1}^{t} f_i) \end{aligned}$$

- The red terms don't vanish and the blue terms contain a multiplication of the forget gates
- The forget gates thus regulate the gradient flow depending on the explicit contribution of a state (s_t) to the next state s_{t+1}

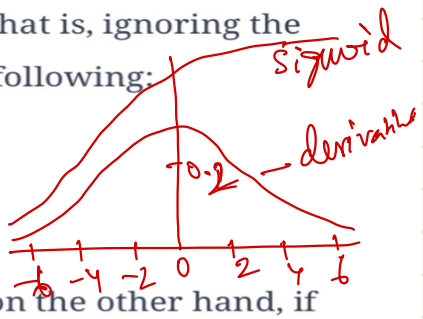
Gradients in LSTMs

Now, let's look at an LSTM cell. More specifically, we'll look at the cell state given by the following equation:

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t$$

This is the product of all the forget gate applications happening in the LSTM. However, if we calculate $\partial c_t / \partial c_{t-k}$ in a similar way for LSTMs (that is, ignoring the $W_{fx}x_t$ terms and b_f because they are nonrecurrent), we get the following:

$$\partial c_t / \partial c_{t-k} = \prod_{i=0}^{k-1} \sigma(W_{fh} h_{t-k+i})$$



In this case, though the gradient will vanish if $W_h h_{t-k+i} << 0$, on the other hand, if $W_h h_{t-k+i} >> 0$, the derivative will decrease much slower than it would in a standard RNN. Therefore, we have one alternative where the gradient will not vanish. Also, as the squashing function is used, the gradients won't explode due to $\partial c_t / \partial c_{t-k}$ being large (which is the thing likely to be the cause of a gradient explosion). In additon, when $W_h h_{t-k+i} >> 0$, we get a maximum gradient close to 1, meaning that the gradient will not rapidly decrease as we saw with RNNs (when the gradient is at maximum). Finally, there is no term such as W_h^k in the derivation. However, derivations are trickier for $\partial h_t / \partial h_{t-k}$. Let's see if such terms are present in the derivation of $\partial h_t / \partial h_{t-k}$. If we calculate the derivatives of this, we'll get something of the following form:

$$\partial h_t / \partial h_{t-k} = \partial(o_t tanh(c_t)) / \partial h_{t-k}$$

Once we solve this, we'll get something of this form:

$$tanh(.)\sigma(.)[1-\sigma(.)]w_{oh} + \sigma(.)[1 - tanh^2(.)]\left\{c_{t-1}\sigma(.)[1-\sigma(.)]w_{fh} + \sigma(.)[1-tanh^2(.)]w_{ch} + tanh(.)\sigma(.)[1-\sigma(.)]u\right.$$

We don't care about the content within $\sigma(.)$ or $tanh(.)$ because no matter the value, it will be bounded by $(0, 1)$ or $(-1, 1)$. If we further reduce the notation by replacing the $\sigma(.)$, $[1-\sigma(.)]$, $tanh(.)$ and $[1-tanh^2(.)]$ terms with a common notation such as $\gamma(.)$, we get something of this form:

$$\gamma(.)w_{oh} + \gamma(.)[c_{t-1}\gamma(.)w_{fh} + \gamma(.)w_{ch} + \gamma(.)w_{ih}]$$

Alternatively, we get the following (assuming that the outside $\gamma(.)$ gets absorbed by each $\gamma(.)$ term present within the square brackets):

$$\gamma(.)w_{oh} + c_{t-1}\gamma(.)w_{fh} + \gamma(.)w_{ch} + \gamma(.)w_{ih}$$

This will give the following:

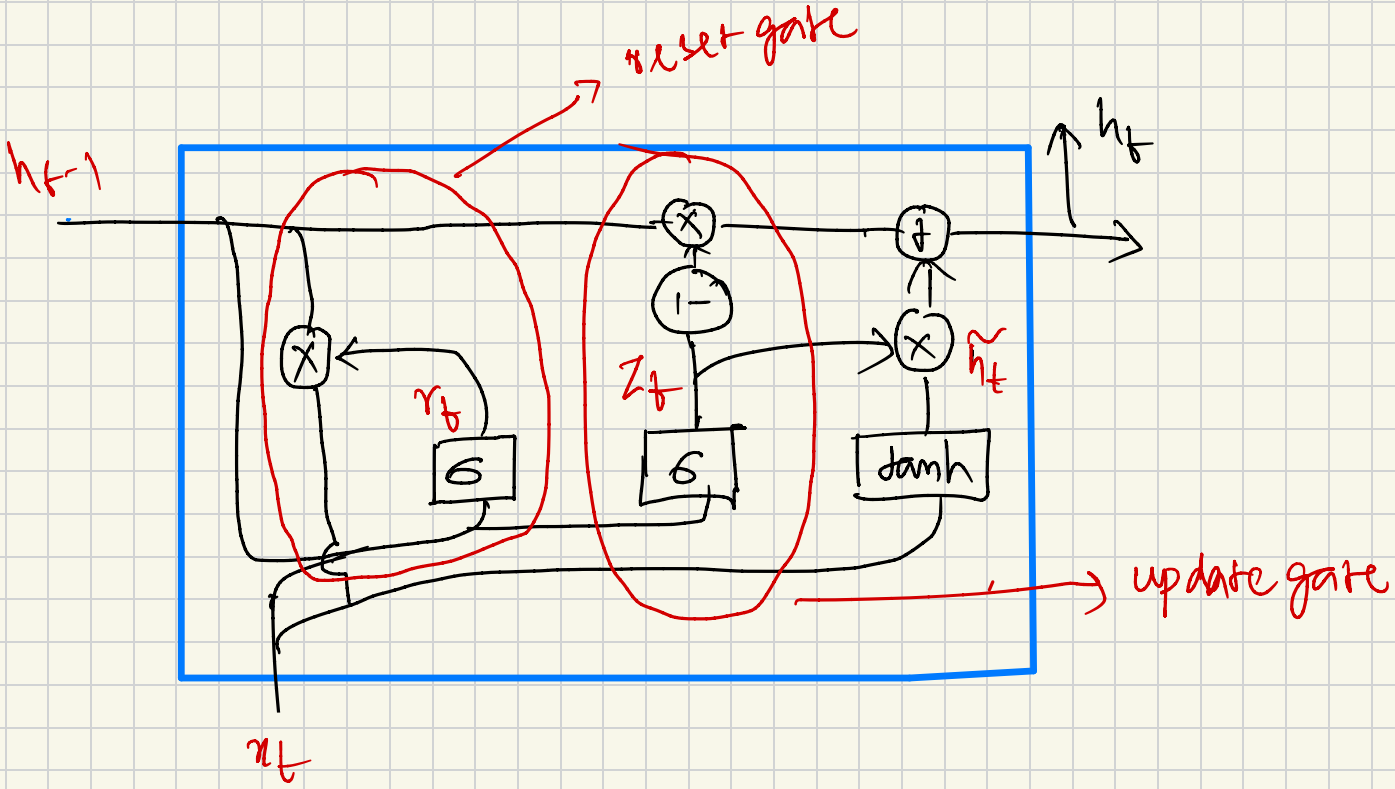
$$\partial h_t/\partial h_{t-k} \approx \prod_{i=0}^{k-1} \gamma(.)w_{oh} + c_{t-1}\gamma(.)w_{fh} + \gamma(.)w_{ch} + \gamma(.)w_{ih}$$

This means that though the term $\partial c_t/\partial c_{t-k}$ is safe from any W_h^k terms, $\partial h_t/\partial h_{t-k}$ is not. Therefore, we must be careful when initializing the weights of the LSTM, and we should use gradient clipping as well.

However, h_t of LSTMs being unsafe from vanishing gradient is not as crucial as it is for RNNs because c_t still can store the long-term dependencies without being affecte by vanishing gradient, and h_t can retrieve the long-term dependencies from c_t, if required to.

GRU

GRU stands for gated recurrent unit, and it is a simplified version of LSTM. It has only two gates: a reset gate and an update gate. The reset gate decides how much of the previous hidden state to keep, and the update gate decides how much of the new input to incorporate into the hidden state. The hidden state also acts as the cell state and the output, so there is no separate output gate. The GRU is easier to implement and requires fewer parameters than the LSTM.



update gate: degree of past information forwarded to the future.

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z)$$

Reset gate: the amount of past information to discard.

$$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r)$$

Candidate hidden state: create new representation considering both input & past hidden layer.

$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t] + b)$$

final hidden state:

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

9.7 The Encoder-Decoder Model with RNNs

In this section we introduce a new model, the encoder-decoder model, which is used when we are taking an input sequence and translating it to an output sequencex that is of a different length than the input, and doesn't align with it in a word-to-word way. Recall that in the sequence labeling task, we have two sequences, but they are

the same length (for example in part-of-speech tagging each token gets an associated tag), each input is associated with a specific output, and the labeling for that output takes mostly local information. Thus deciding whether a word is a verb or a noun, we look mostly at the word and the neighboring words.

By contrast, encoder-decoder models are used especially for tasks like machine translation, where the input sequence and output sequence can have different lengths and the mapping between a token in the input and a token in the output can be very indirect (in some languages the verb appears at the beginning of the sentence; in other languages at the end). We'll introduce machine translation in detail in Chapter 13, but for now we'll just point out that the mapping for a sentence in English to a sentence in Tagalog or Yoruba can have very different numbers of words, and the words can be in a very different order.

encoder-decoder

Encoder-decoder networks, sometimes called **sequence-to-sequence** networks, are models capable of generating contextually appropriate, arbitrary length, output sequences given an input sequence. Encoder-decoder networks have been applied to a very wide range of applications including summarization, question answering, and dialogue, but they are particularly popular for machine translation.

The key idea underlying these networks is the use of an **encoder** network that takes an input sequence and creates a contextualized representation of it, often called the **context**. This representation is then passed to a **decoder** which generates a task-specific output sequence. Fig. 9.16 illustrates the architecture

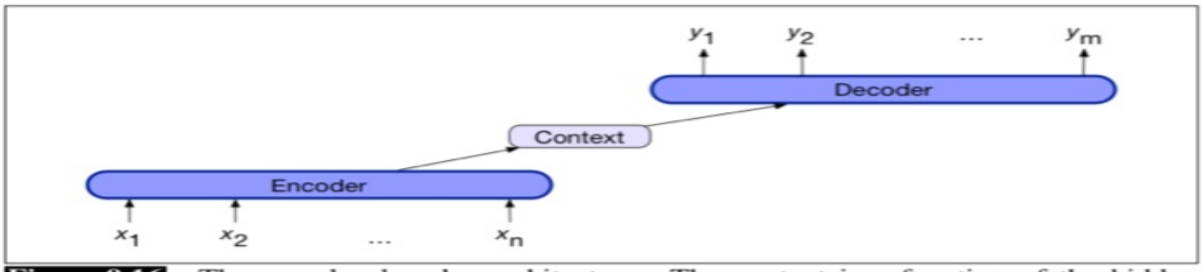


Figure 9.16 The encoder-decoder architecture. The context is a function of the hidden representations of the input, and may be used by the decoder in a variety of ways.

Encoder-decoder networks consist of three components:

1. An **encoder** that accepts an input sequence, x_1^n, and generates a corresponding sequence of contextualized representations, h_1^n. LSTMs, convolutional networks, and Transformers can all be employed as encoders.
2. A **context vector**, c, which is a function of h_1^n, and conveys the essence of the input to the decoder.
3. A **decoder**, which accepts c as input and generates an arbitrary length sequence of hidden states h_1^m, from which a corresponding sequence of output states y_1^m, can be obtained. Just as with encoders, decoders can be realized by any kind of sequence architecture.

In this section we'll describe an encoder-decoder network based on a pair of RNNs, but we'll see in Chapter 13 how to apply them to transformers as well. We'll build up the equations for encoder-decode models by starting with the conditional RNN language model $p(y)$, the probability of a sequence y.

Recall that in any language model, we can break down the probability as follows:

$$p(y) = p(y_1)p(y_2|y_1)p(y_3|y_1,y_2)...P(y_m|y_1,...,y_{m-1}) \quad (9.28)$$

In RNN language modeling, at a particular time t, we pass the prefix of $t-1$ tokens through the language model, using forward inference to produce a sequence of hidden states, ending with the hidden state corresponding to the last word of the prefix. We then use the final hidden state of the prefix as our starting point to generate the next token.

More formally, if g is an activation function like *tanh* or ReLU, a function of the input at time t and the hidden state at time $t-1$, and f is a softmax over the set of possible vocabulary items, then at time t the output $\mathbf{y}_t$ and hidden state $\mathbf{h}_t$ are computed as:

$$\mathbf{h}_t = g(\mathbf{h}_{t-1}, \mathbf{x}_t) \tag{9.29}$$

$$\mathbf{y}_t = f(\mathbf{h}_t) \tag{9.30}$$

sentence separation

We only have to make one slight change to turn this language model with autoregressive generation into an encoder-decoder model that is a translation model that can translate from a **source text** in one language to a **target text** in a second: add a **sentence separation** marker at the end of the source text, and then simply concatenate the target text.

Let's use <s> for our sentence separate token, and let's think about translating an English source text ("the green witch arrived"), to a Spanish sentence ("*llegó la bruja verde*" (which can be glossed word-by-word as 'arrived the witch green'). We could also illustrate encoder-decoder models with a question-answer pair, or a text-summarization pair, but m

Let's use x to refer to the source text (in this case in English) plus the separator token <s>, and y to refer to the target text y (in this case in Spanish). Then an encoder-decoder model computes the probability $p(y|x)$ as follows:

$$p(y|x) = p(y_1|x)p(y_2|y_1,x)p(y_3|y_1,y_2,x)...P(y_m|y_1,...,y_{m-1},x) \tag{9.31}$$

Fig. 9.17 shows the setup for a simplified version of the encoder-decoder model (we'll see the full model, which requires the new concept of **attention**, in the next section).

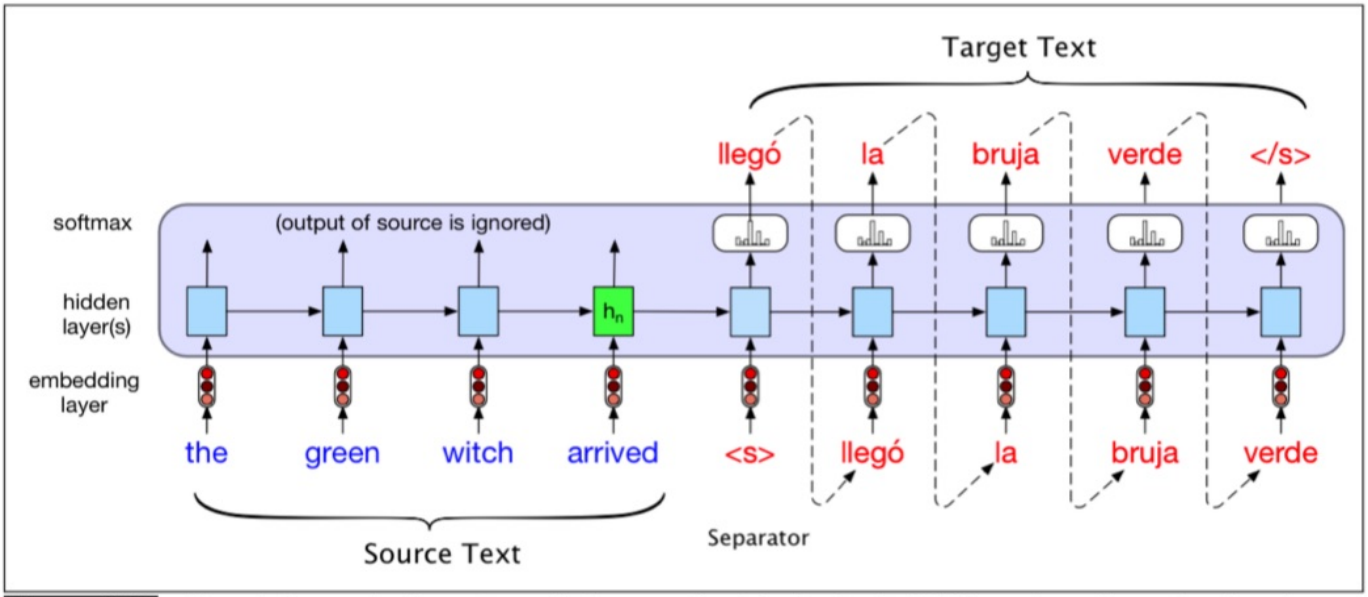


Figure 9.17 Translating a single sentence (inference time) in the basic RNN version of encoder-decoder approach to machine translation. Source and target sentences are concatenated with a separator token in between, and the decoder uses context information from the encoder's last hidden state.

Fig. 9.17 shows an English source text ("the green witch arrived"), a sentence separator token (<s>, and a Spanish target text ("*llegó la bruja verde*"). To trans-

late a source text, we run it through the network performing forward inference to generate hidden states until we get to the end of the source. Then we begin autoregressive generation, asking for a word in the context of the hidden layer from the end of the source input as well as the end-of-sentence marker. Subsequent words are conditioned on the previous hidden state and the embedding for the last word generated.

Let's formalize and generalize this model a bit in Fig. 9.18. (To help keep things straight, we'll use the superscripts e and d where needed to distinguish the hidden states of the encoder and the decoder.) The elements of the network on the left process the input sequence x and comprise the **encoder**. While our simplified figure shows only a single network layer for the encoder, stacked architectures are the norm, where the output states from the top layer of the stack are taken as the final representation. A widely used encoder design makes use of stacked biLSTMs where the hidden states from top layers from the forward and backward passes are concatenated as described in Chapter 9 to provide the contextualized representations for each time step.

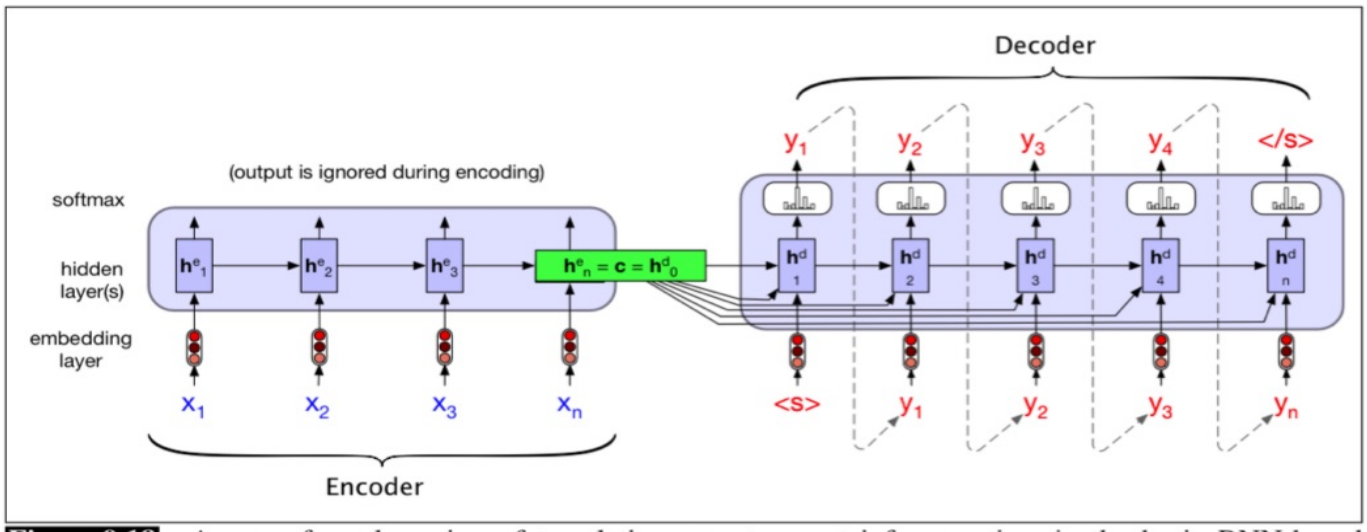


Figure 9.18 A more formal version of translating a sentence at inference time in the basic RNN-based encoder-decoder architecture. The final hidden state of the encoder RNN, h_n^e, serves as the context for the decoder in its role as h_0^d in the decoder RNN.

The entire purpose of the encoder is to generate a contextualized representation of the input. This representation is embodied in the final hidden state of the encoder, $\mathbf{h}_n^e$. This representation, also called $\mathbf{c}$ for **context**, is then passed to the decoder.

The **decoder** network on the right takes this state and uses it to initialize the first hidden state of the decoder. That is, the first decoder RNN cell uses c as its prior hidden state $\mathbf{h}_0^d$. The decoder autoregressively generates a sequence of outputs, an element at a time, until an end-of-sequence marker is generated. Each hidden state is conditioned on the previous hidden state and the output generated in the previous state.

One weakness of this approach as described so far is that the influence of the context vector, c, will wane as the output sequence is generated. A solution is to make the context vector c available at each step in the decoding process by adding it as a parameter to the computation of the current hidden state, using the following equation (illustrated in Fig. 9.19):

$$\mathbf{h}_t^d = g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c}) \tag{9.32}$$

Now we're ready to see the full equations for this version of the decoder in the basic

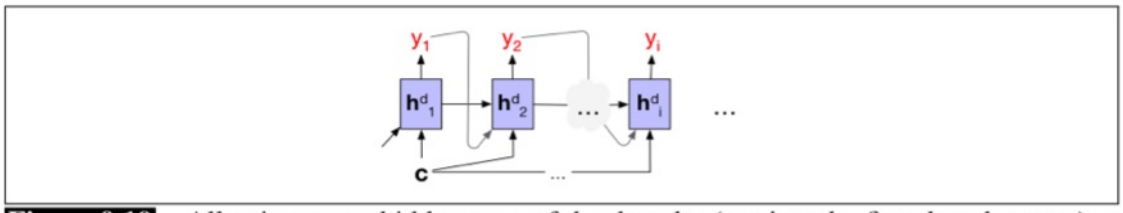


Figure 9.19 Allowing every hidden state of the decoder (not just the first decoder state) to be influenced by the context c produced by the encoder.

encoder-decoder model, with context available at each decoding timestep. Recall that g is a stand-in for some flavor of RNN and $\hat{y}_{t-1}$ is the embedding for the output sampled from the softmax at the previous step:

$$\begin{aligned} \mathbf{c} &= \mathbf{h}_n^e \\ \mathbf{h}_0^d &= \mathbf{c} \\ \mathbf{h}_t^d &= g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c}) \\ \mathbf{z}_t &= f(\mathbf{h}_t^d) \\ y_t &= \text{softmax}(\mathbf{z}_t) \end{aligned} \tag{9.33}$$

Finally, as shown earlier, the output y at each time step consists of a softmax computation over the set of possible outputs (the vocabulary, in the case of language modeling or MT). We compute the most likely output at each time step by taking the argmax over the softmax output:

$$\hat{y}_t = \text{argmax}_{w \in V} P(w|x, y_1 \ldots y_{t-1}) \tag{9.34}$$

9.7.1 Training the Encoder-Decoder Model

Encoder-decoder architectures are trained end-to-end, just as with the RNN language models of Chapter 9. Each training example is a tuple of paired strings, a source and a target. Concatenated with a separator token, these source-target pairs can now serve as training data.

For MT, the training data typically consists of sets of sentences and their translations. These can be drawn from standard datasets of aligned sentence pairs, as we'll discuss in Section 13.2.2. Once we have a training set, the training itself proceeds as with any RNN-based language model. The network is given the source text and then starting with the separator token is trained autoregressively to predict the next word, as shown in Fig. 9.20.

Note the differences between training (Fig. 9.20) and inference (Fig. 9.17) with respect to the outputs at each time step. The decoder during inference uses its own estimated output $\hat{y}_t$ as the input for the next time step x_{t+1}. Thus the decoder will tend to deviate more and more from the gold target sentence as it keeps generating more tokens. In training, therefore, it is more common to use **teacher forcing** in the decoder. Teacher forcing means that 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 $\hat{y}_t$. This speeds up training.

teacher forcing

9.8 Attention

The simplicity of the encoder-decoder model is its clean separation of the encoder—which builds a representation of the source text—from the decoder, which uses this

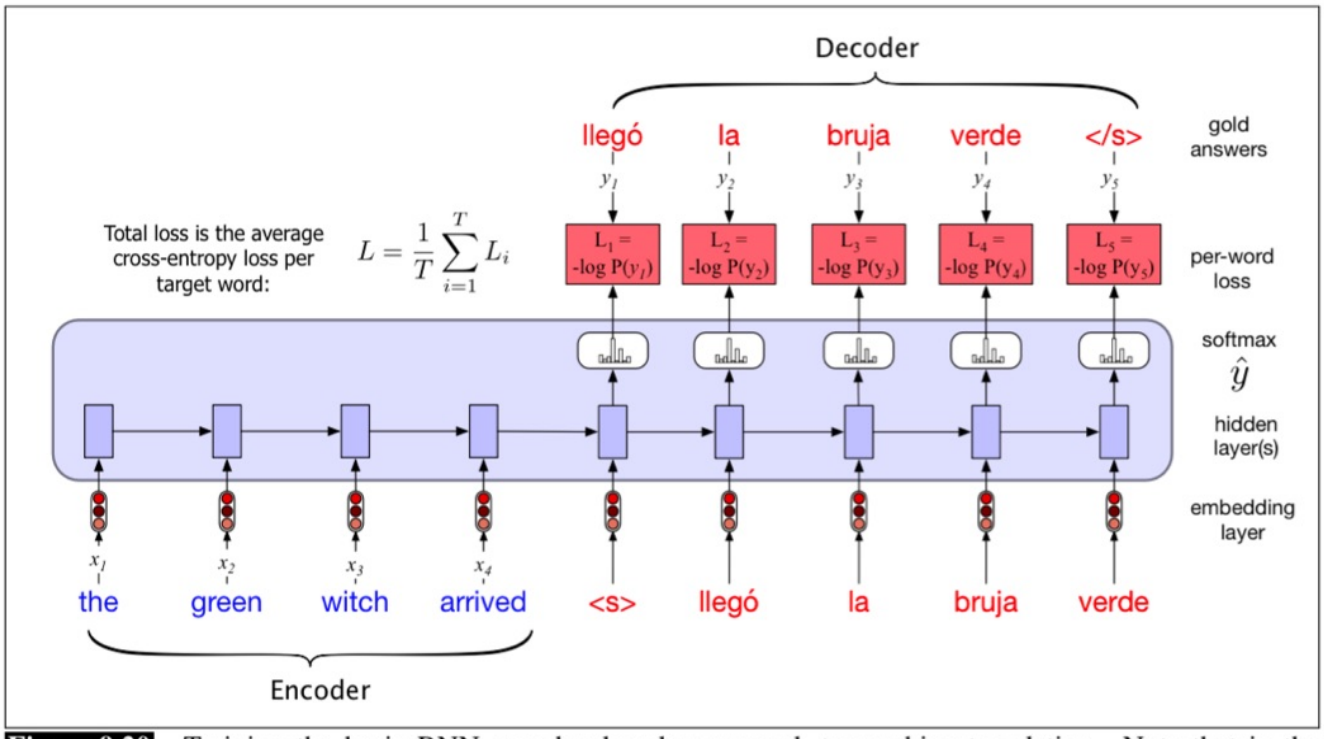


Figure 9.20 Training the basic RNN encoder-decoder approach to machine translation. Note that in the decoder we usually don't propagate the model's softmax outputs $\hat{y}_t$, but use **teacher forcing** to force each input to the correct gold value for training. We compute the softmax output distribution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

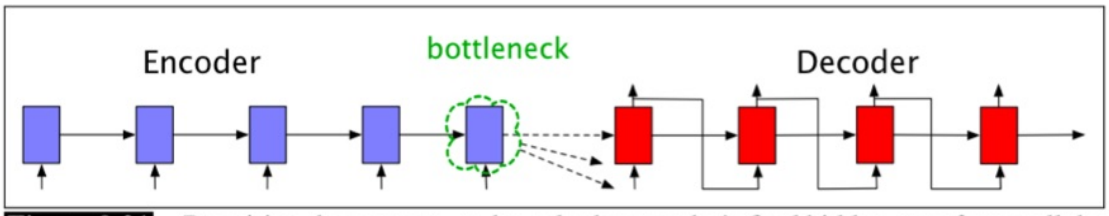


Figure 9.21 Requiring the context c to be only the encoder's final hidden state forces all the information from the entire source sentence to pass through this representational bottleneck.

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):

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

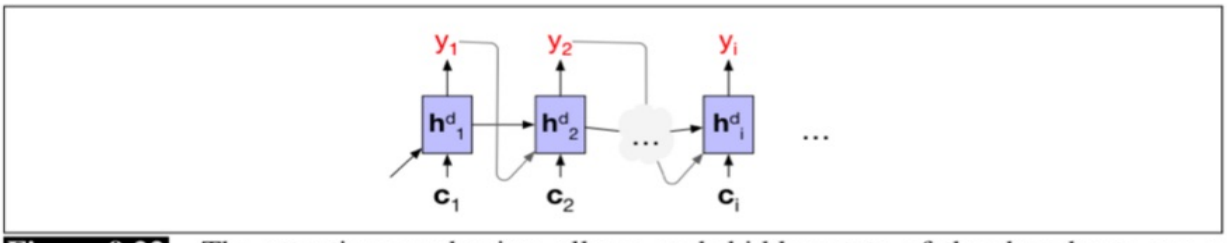


Figure 9.22 The attention mechanism allows each hidden state of the decoder to see a different, dynamic, context, which is a function of all the encoder hidden states.

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 $\mathbf{c}_i$.

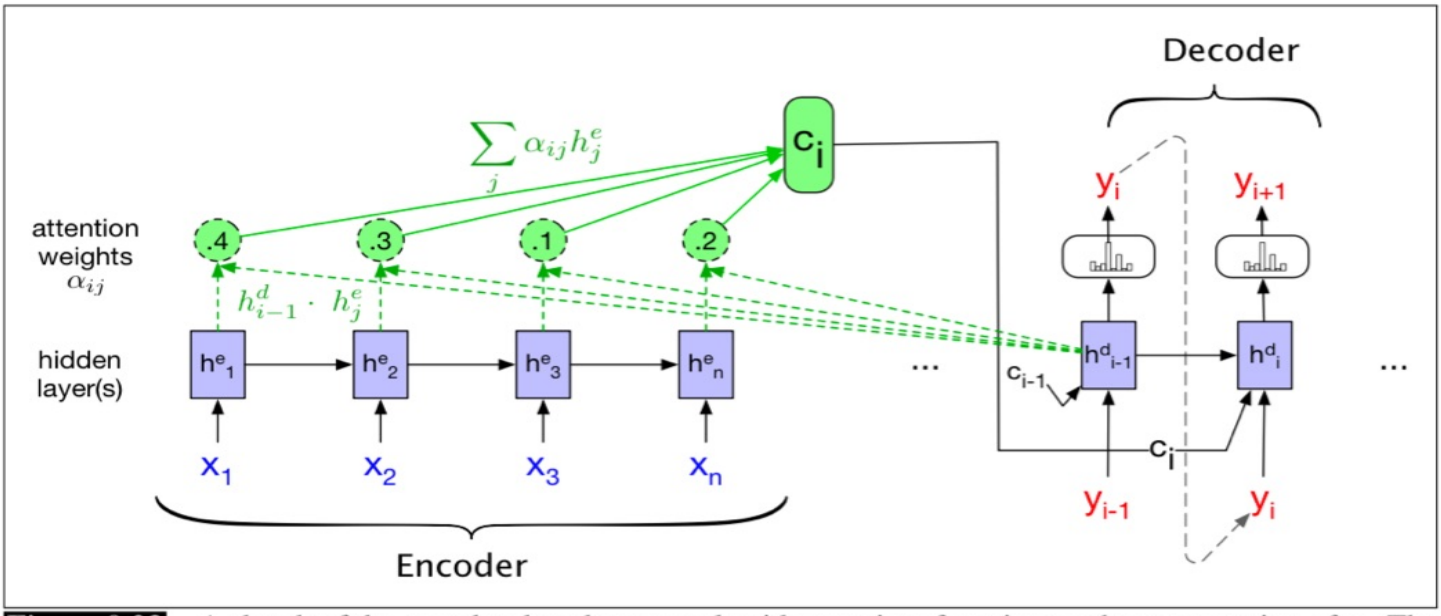


Figure 9.23 A sketch of the encoder-decoder network with attention, focusing on the computation of $\mathbf{c}_i$. The context value $\mathbf{c}_i$ is one of the inputs to the computation of $\mathbf{h}^d_i$. It is computed by taking the weighted sum of all the encoder hidden states, each weighted by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

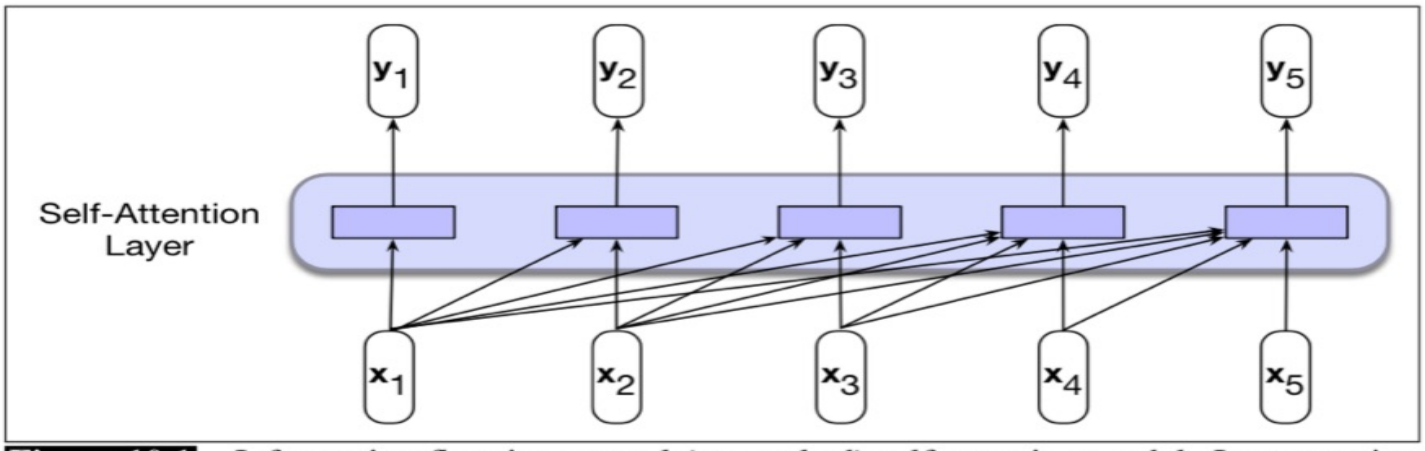


Figure 10.1 Information flow in a causal (or masked) self-attention model. In processing each element of the sequence, the model attends to all the inputs up to, and including, the current one. Unlike RNNs, the computations at each time step are independent of all the other steps and therefore can be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

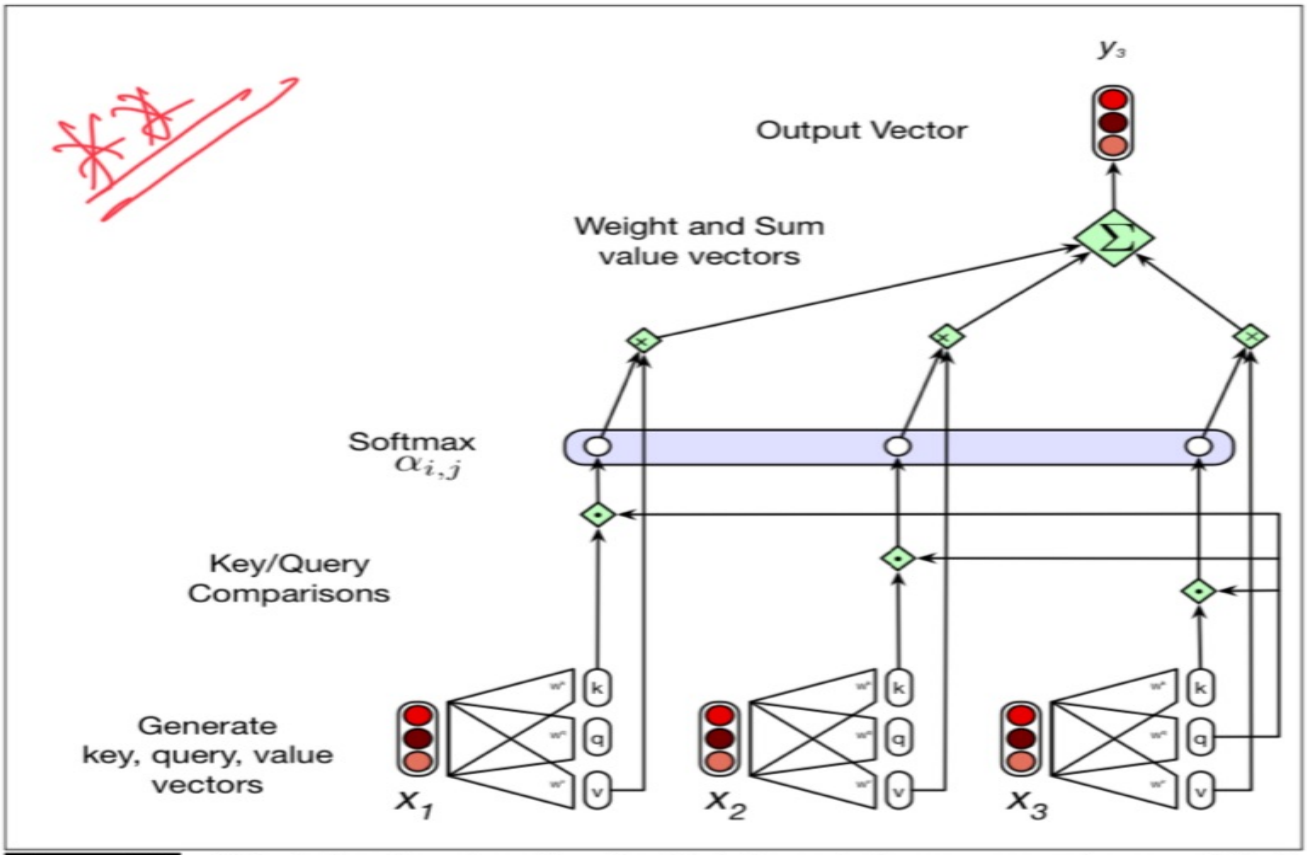


Figure 10.2 Calculating the value of $\mathbf{y}_3$, the third element of a sequence using causal (left-to-right) self-attention.

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).

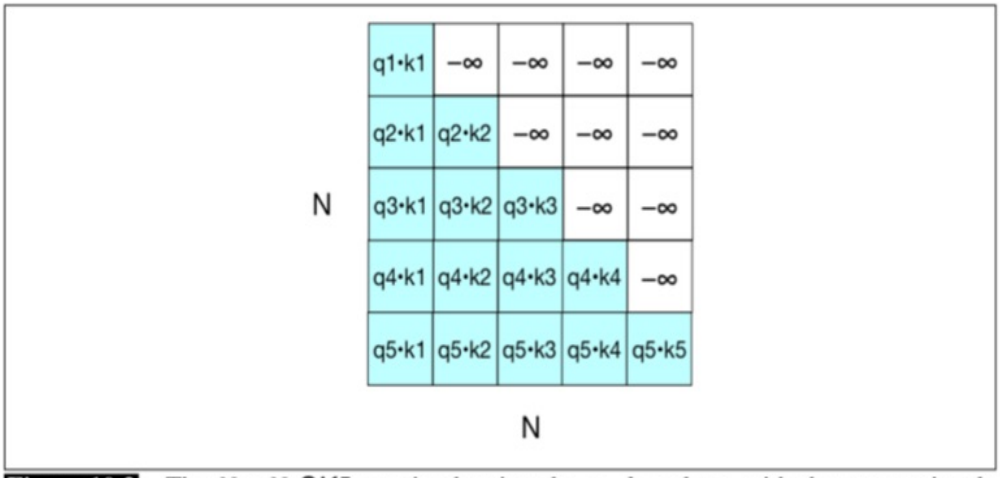


Figure 10.3 The $N \times N$ $\mathbf{QK}^\intercal$ matrix showing the $q_i \cdot k_j$ values, with the upper-triangle portion of the comparisons matrix zeroed out (set to $-\infty$, which the softmax will turn to zero).

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.

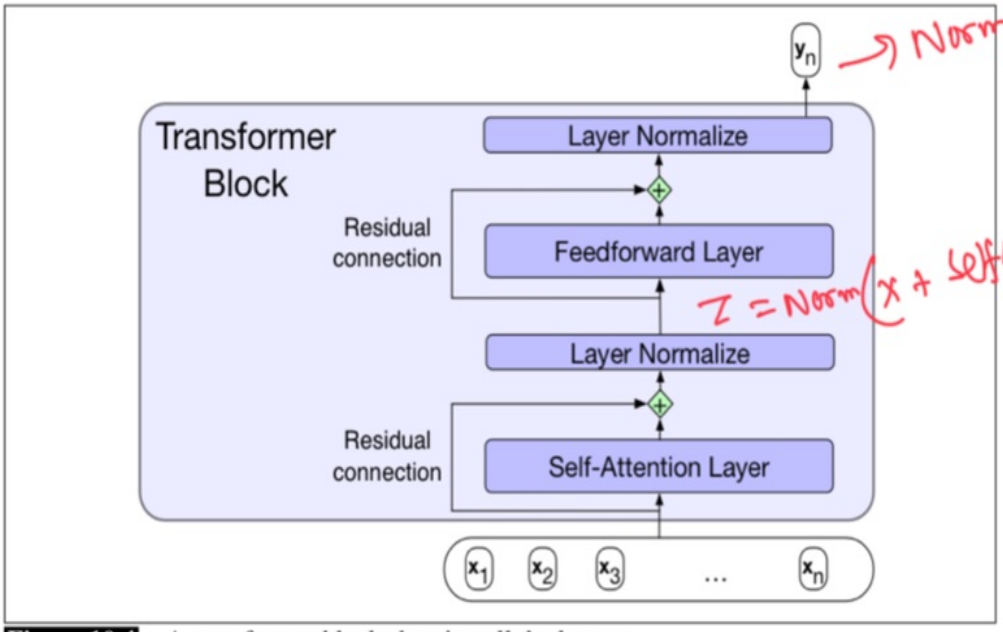


Figure 10.4 A transformer block showing all the layers.

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.

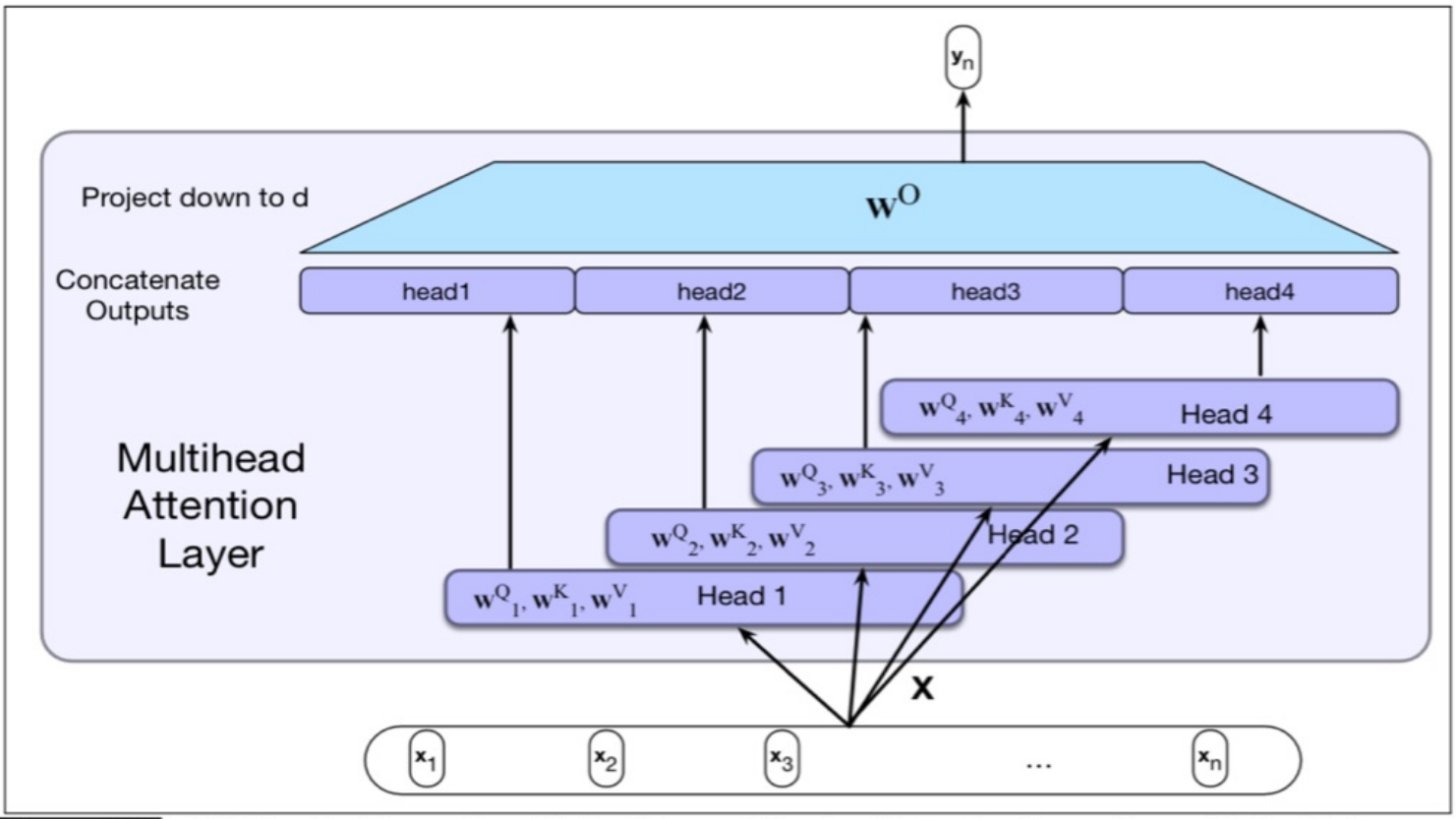


Figure 10.5 Multihead self-attention: Each of the multihead self-attention layers is provided with its own set of key, query and value weight matrices. The outputs from each of the layers are concatenated and then projected down to d, thus producing an output of the same size as the input so layers can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

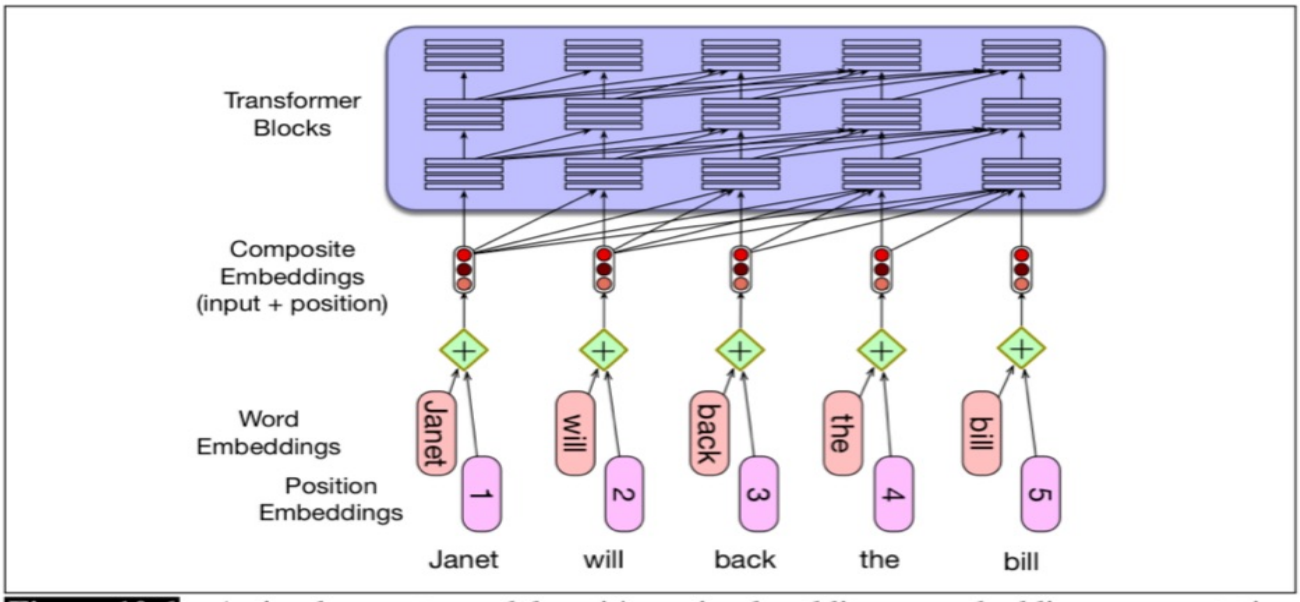


Figure 10.6 A simple way to model position: simply adding an embedding representation of the absolute position to the input word embedding to produce a new embedding of the same dimenionality.

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.

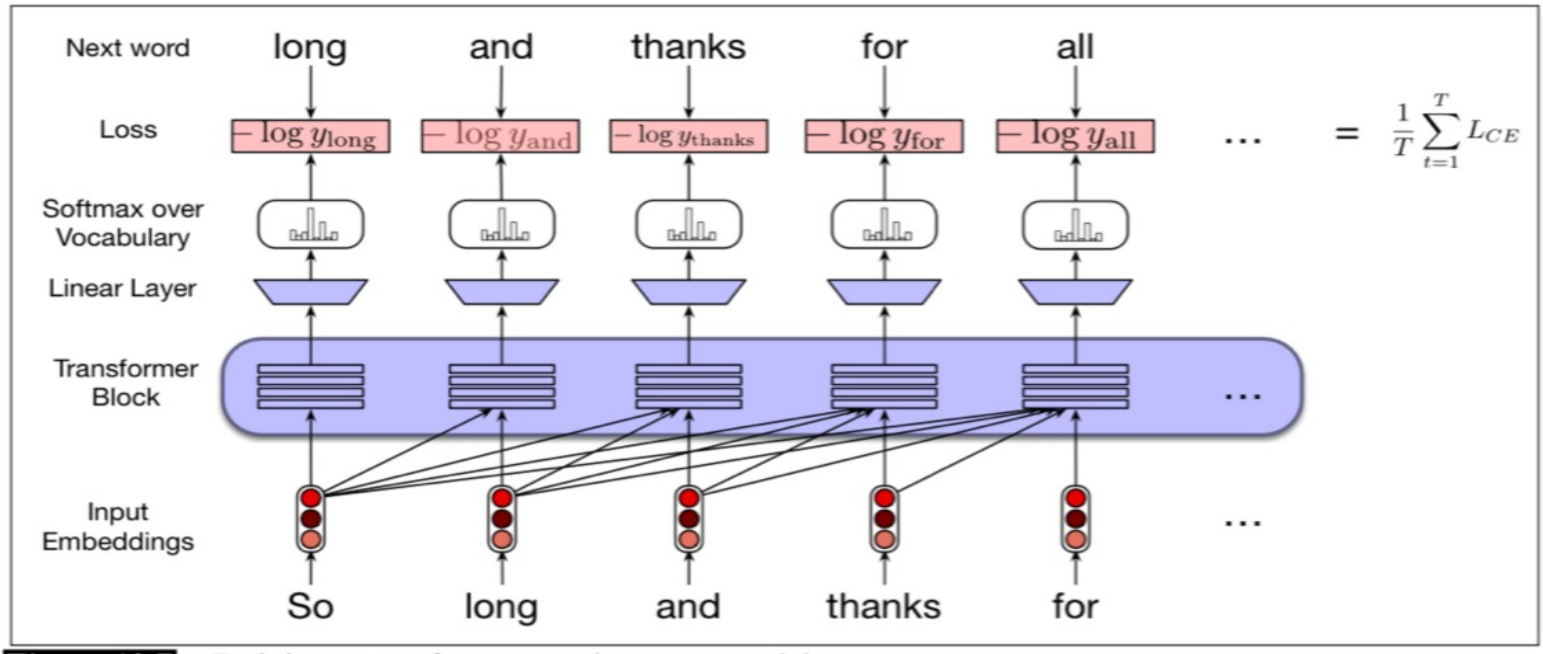


Figure 10.7 Training a transformer as a language model.

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

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!

Obtain all the n-gram probabilities P((I| <s>), P(NLP|<s>), P(am|I) P(do|I) P(NLP|am) from the following set of sentences[2 marks]

<s> I am NLP </s>

<s> NLP I am </s>

<s> I do not like Exams and Marks </s>

Solution:

P((I| <s>)=2/3=0.67; 0.5 marks

P(NLP|<s>)=1/2=0.5; 0.5 marks

P(am|I)=2/3=0.67; 0.5 marks

P(do|I)=1/3=0.33; both together 0.5 marks

P(NLP|am)=1/2=0.5

Suppose in our training corpus [2marks]

- girl appears 8 times as a noun and 4 times as a verb
- sleep appears twice as a noun and 6 times as a verb what is the Emission probabilities of the below sentence

girl sleep

Solution:

Noun

P(girl|noun) 0.8 -0.5marks

P(sleep| noun) 0.2 -0.5 marks

Verb

P(girl| verb) 0.4 -0.5 marks

P(sleep| verb) 0.6 -0.5 marks

c. Given the emission probabilities and transition probabilities find the correct pos tag for the sentence using HMM [3marks]

"I go to school"

Emission probabilities

	I	GO	TO	SCHOOL
VB	0	0.1	0	0.8
TO	0	0	0.99	0
NN	0	0.85	0	0.68
PPSS	0.37	0	0	0

Transition probabilities

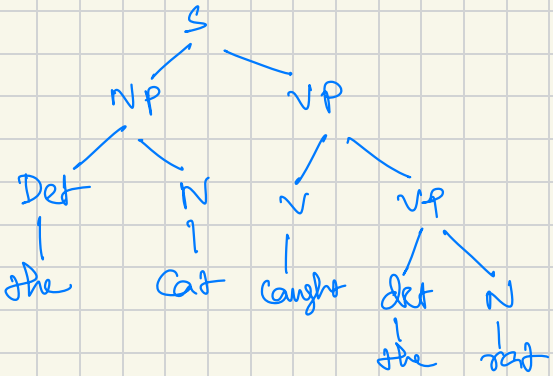
	VB	TO	NN	PPSS
<s>	0.19	0.43	0.41	0.67
VB	0.38	0.35	0.47	0.70
TO	0.83	0	0.47	0
NN	0.40	0.16	0.87	0.45
PPSS	0.23	0.79	0.12	0.14

Given the grammar and lexicon below, derive the parse tree using the top-down parsing method for the sentence [3 marks]

S: the cat caught the rat

S->NP VP VP->VNP NP->Det N

N->rat, N->cat ,Det ->the V->caught



Congratulations! You passed!
TO PASS 80% or higher

Keep Learning

GRADE
100%

Recurrent Neural Networks

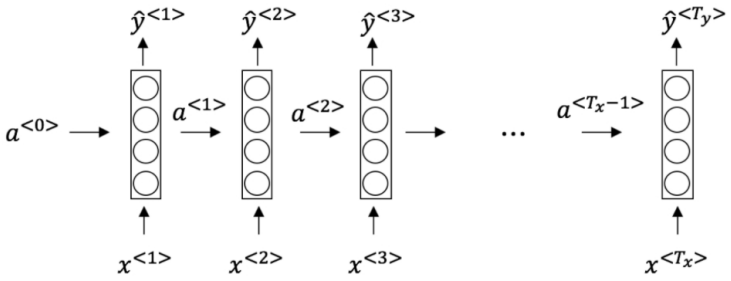
LATEST SUBMISSION GRADE
100%

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**

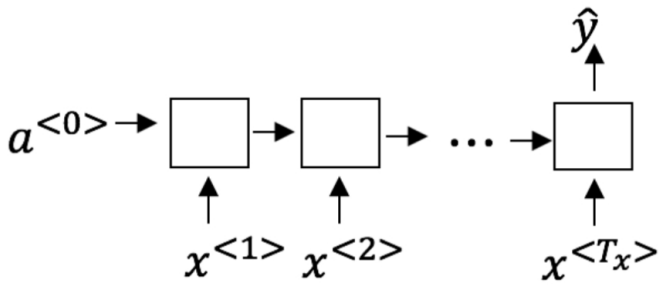


This specific type of architecture is appropriate when:

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

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)