for each count c, an adjusted count c* is

$$c^* = \frac{(c+1)\, N_{c+1}}{N_c}$$

N_c = the number of n-grams exactly seen c times.

distribute probability mass

N_c → No. of words occurs c times

N_{c+1} → No of words occurs c+1 times

$$c^* = \frac{(c+1)\, N_{c+1}}{N}$$

P^*_{GT} (things with frequency c) = $\frac{c^*}{N}$

distributing this probability mass that occurs c times.

If there are N_c words.

so, each of them will get

$$\frac{(c+1)\, N_{c+1}}{N N_c}$$

effective count

$$\Rightarrow \frac{c^*}{N} = \frac{(c+1)\, N_{c+1}}{N_c}$$

if c = 0 that means it will distribute the probability using the probabilities that occurs 1.

P^*_{GT} (things with frequency c) = $\frac{N_1}{N}$ where

N denotes the total no of bigrams that actually occurs in train data.

RNN

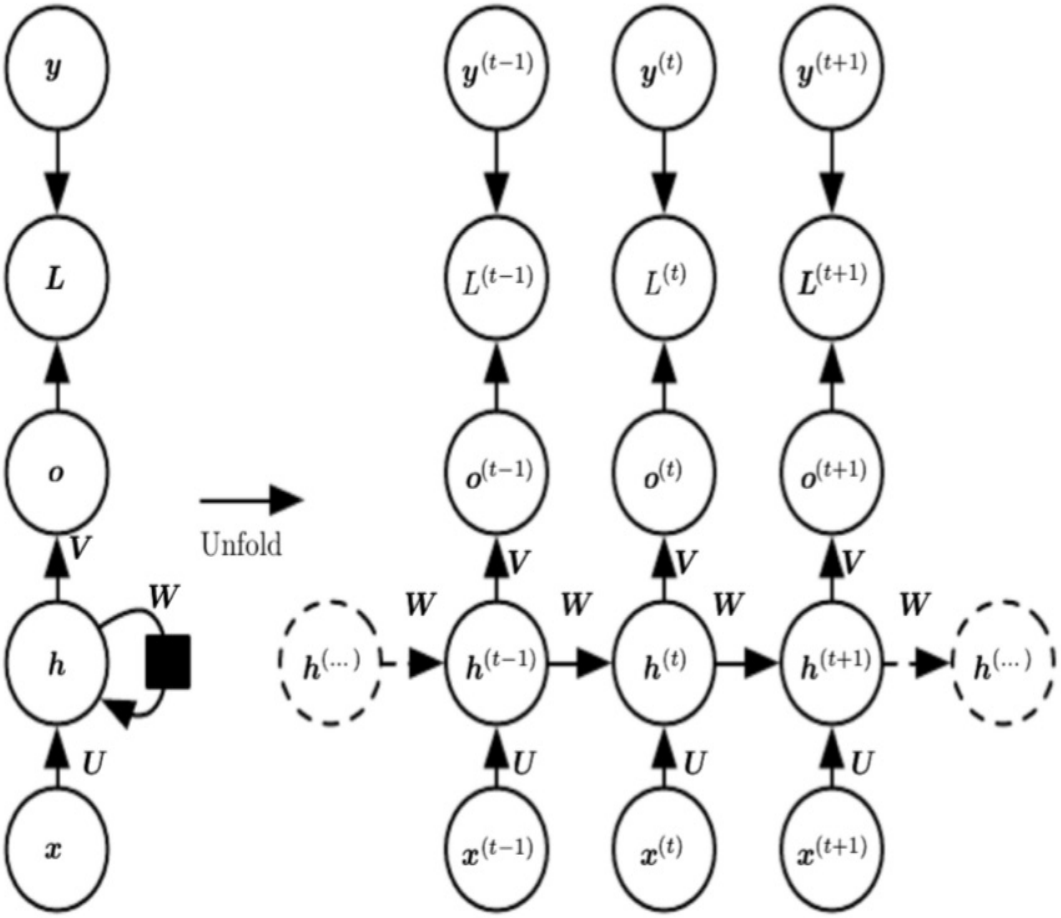
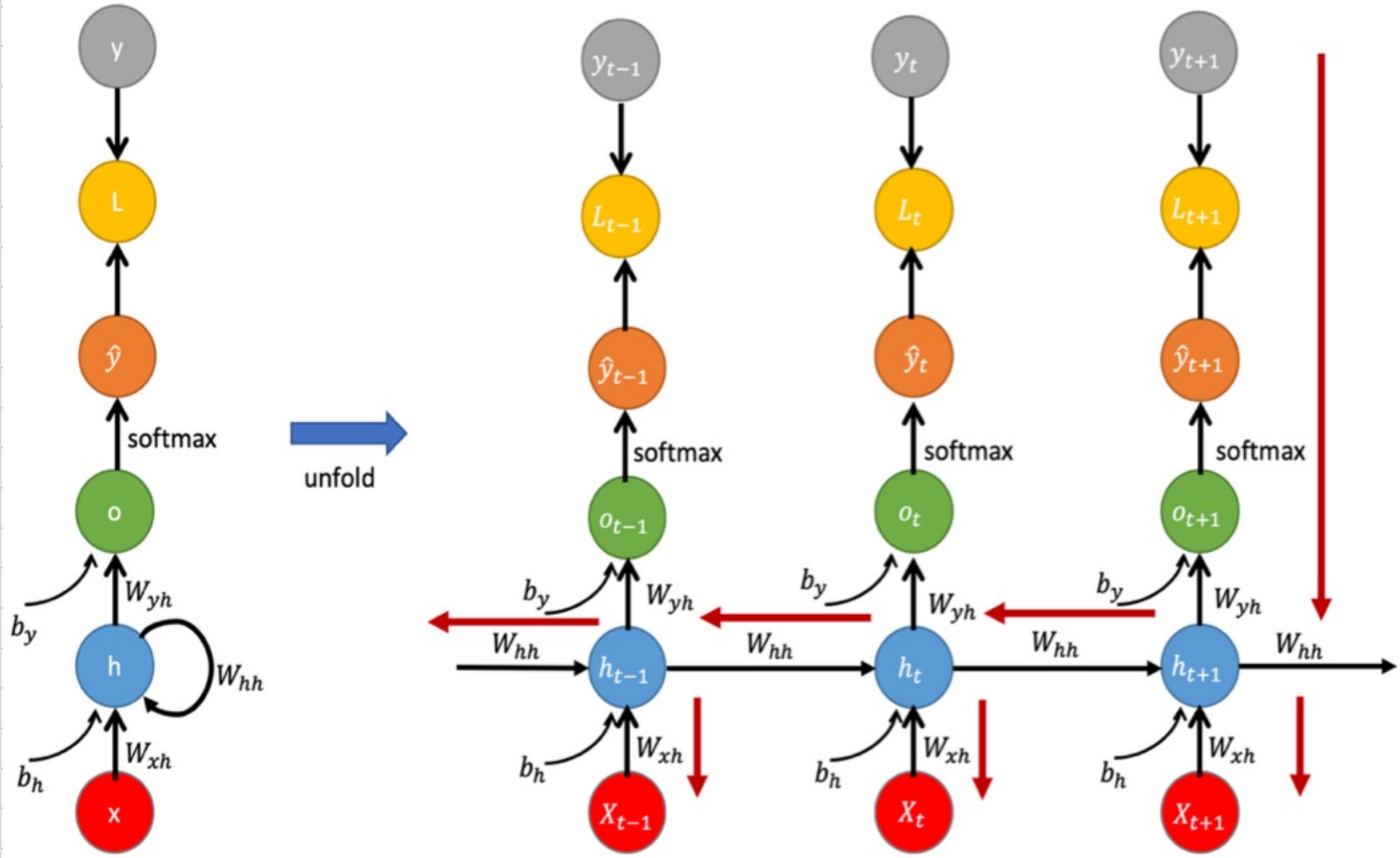


Figure 10.3: The computational graph to compute the training loss of a recurrent network that maps an input sequence of $\boldsymbol{x}$ values to a corresponding sequence of output $\boldsymbol{o}$ values. A loss L measures how far each $\boldsymbol{o}$ is from the corresponding training target $\boldsymbol{y}$. When using softmax outputs, we assume $\boldsymbol{o}$ is the unnormalized log probabilities. The loss L internally computes $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{o})$ and compares this to the target $\boldsymbol{y}$. The RNN has input to hidden connections parametrized by a weight matrix $\boldsymbol{U}$, hidden-to-hidden recurrent connections parametrized by a weight matrix $\boldsymbol{W}$, and hidden-to-output connections parametrized by a weight matrix $\boldsymbol{V}$. Equation 10.8 defines forward propagation in this model. *(Left)* The RNN and its loss drawn with recurrent connections. *(Right)* The same seen as a time-unfolded computational graph, where each node is now associated with one particular time instance.



Just like for feedforward neural networks, we can compute a recurrent layer's output in one shot for a whole mini-batch by placing all the inputs at time step t in an input matrix X_t:

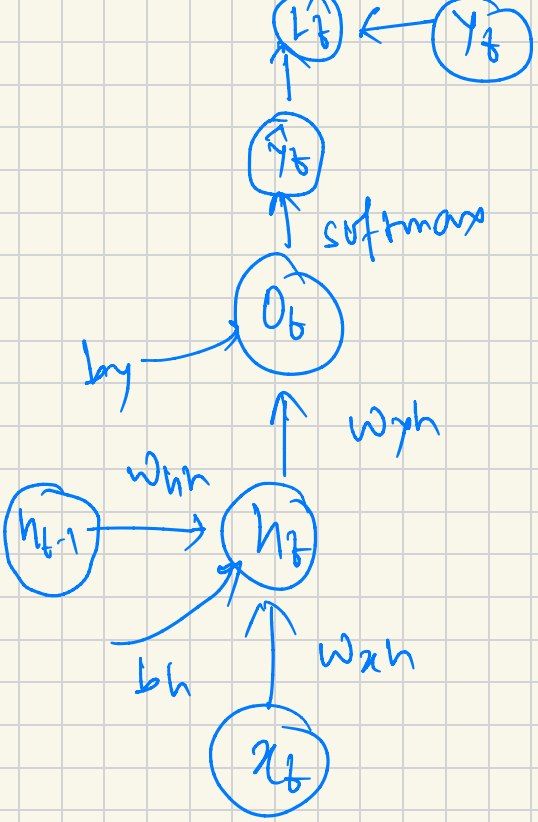
$$
\begin{aligned}
h_t &= tanh(X_t \cdot W_{xh} + h_{t-1} \cdot W_{hh} + b_h) \\
&= \phi_h([X_t h_{t-1}] \cdot W + b_h) \\
o_t &= h_t \cdot W_{yh} + b_y \\
\hat{y}_t &= softmax(o_t)
\end{aligned}
$$

1. The weight matrices W_{xh} and W_{yh} are often concatenated vertically into a single weight matrix W of shape $(n_{inputs} + n_{neurons}) \times n_{neurons}$.
2. The notation $[X_t h_{t-1}]$ represents the horizontal concatenation of the matrices X_t and h_{t-1}, shape of $m \times (n_{inputs} + n_{neurons})$

Let's denote m as the number of instances in the mini-batch, $n_{neurons}$ as the number of neurons, and n_{inputs} as the number of input features.

1. X_t is an $m \times n_{inputs}$ matrix containing the inputs for all instances.
2. h_{t-1} is an $m \times n_{neurons}$ matrix containing the hidden state of the previous time-step for all instances.
3. W_{xh} is an $n_{inputs} \times n_{neurons}$ matrix containing the connection weights between input and the hidden layer.
4. W_{hh} is an $n_{neurons} \times n_{neurons}$ matrix containing the connection weights between two hidden layers.
5. W_{yh} is an $n_{neurons} \times n_{neurons}$ matrix containing the connection weights between the hidden layer and the output.
6. b_h is a vector of size $n_{neurons}$ containing each neuron's bias term.
7. b_y is a vector of size $n_{neurons}$ containing each output's bias term.
8. y_t is an $m \times n_{neurons}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch

NOTE: At the first time step, $t = 0$, there are no previous outputs, so they are typically assumed to be all zeros.



Let's assume we pick a vocabulary = 8000

$x_t \in \mathbb{R}^{8000}$ $o_t \in \mathbb{R}^{8000}$ $h_t \in \mathbb{R}^{100}$

$W_{xh} \in \mathbb{R}^{100 \times 8000}$

$W_{hh} \in \mathbb{R}^{100 \times 100}$

$W_{yh} \in \mathbb{R}^{8000 \times 100}$

total parameter we need to learn

$= XH + YH + H^2$

$= 8000 \times 100 + 8000 \times 100 + (100)^2$

$= 1610000$

So, the dimensions also tell us the bottleneck of our model. As x_t is one-hot vector, multiplying it with W_{xh} is same as selecting a column of W_{xh}, so we don't need to perform the full multiplication. Then the biggest multiplication in our network is

Why. That's why we keep our vocabulary size small if possible.

Training RNN

9.1.2 Training (Why Backpropagation can't be used)

As with feedforward networks, we'll use a training set, a loss function, and backpropagation to obtain the gradients needed to adjust the weights in these recurrent networks. As shown in Fig. 9.2, we now have 3 sets of weights to update: **W**, the weights from the input layer to the hidden layer, **U**, the weights from the previous hidden layer to the current hidden layer, and finally **V**, the weights from the hidden layer to the output layer.

Fig. 9.4 highlights two considerations that we didn't have to worry about with backpropagation in feedforward networks. First, to compute the loss function for the output at time t we need the hidden layer from time $t-1$. Second, the hidden layer at time t influences both the output at time t and the hidden layer at time $t+1$ (and hence the output and loss at $t+1$). It follows from this that to assess the error accruing to $\mathbf{h}_t$, we'll need to know its influence on both the current output *as well as the ones that follow*.

Tailoring the backpropagation algorithm to this situation leads to a two-pass algorithm for training the weights in RNNs. In the first pass, we perform forward inference, computing $\mathbf{h}_t$, $\mathbf{y}_t$, accumulating the loss at each step in time, saving the value of the hidden layer at each step for use at the next time step. In the second phase, we process the sequence in reverse, computing the required gradients as we go, computing and saving the error term for use in the hidden layer for each step backward in time. This general approach is commonly referred to as **backpropagation through time** (Werbos 1974, Rumelhart et al. 1986, Werbos 1990).

We need to calculate 5 derivatives w.r.t loss

$\frac{\partial L}{\partial W_{yh}}$, $\frac{\partial L}{\partial W_{hh}}$, $\frac{\partial L}{\partial W_{xh}}$, $\frac{\partial L}{\partial b_y}$, $\frac{\partial L}{\partial b_h}$

While calculating $\frac{\partial L}{\partial W_{hh}}$ we face problem with traditional Backpropagation.

Because the parameters are shared by all time steps in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps. So, slightly modified version of Backpropagation called backpropagation through time (BPTT) is being used.

Let's calculate these derivatives one by one.

$$\text{Loss } L(\hat{Y}, Y) = \sum_{t=1}^{T} L_t(\hat{Y}_t, Y_t)$$

$$= -\sum_{t=1}^{T} Y_t \log \hat{Y}_t$$

$$= -\sum_{t=1}^{T} Y_t \log\left[\text{softmax}(O_t)\right]$$

$$\boxed{\frac{\partial L}{\partial W_{yh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W_{yh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{Y}_t} \cdot \frac{\partial \hat{Y}_t}{\partial O_t} \left(\frac{\partial O_t}{\partial W_{yh}}\right) \to h_t}$$

$$= \sum_{t=1}^{T} (\hat{Y}_t - Y_t) \otimes h_t$$

$$\frac{\partial L}{\partial b_y} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{Y}_t} \cdot \frac{\partial \hat{Y}_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial b_y} = \sum_{t=1}^{T} (\hat{Y}_t - Y_t)$$

$$\frac{\partial L_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial \hat{Y}_t} \frac{\partial \hat{Y}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{hh}}$$

we know $h_t = \tanh\left(W_{xh}^T x_t + W_{hh}^T \cdot h_{t-1} + b_h\right)$

$$\Rightarrow \frac{\partial L_t}{\partial \hat{Y}_t} \cdot \frac{\partial \hat{Y}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial W_{hh}}$$

Thus, at timestamp t, we can compute the gradient & further go through time till 1 to compute overall gradient w.r.t W_{hh}.

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

↓ 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}.

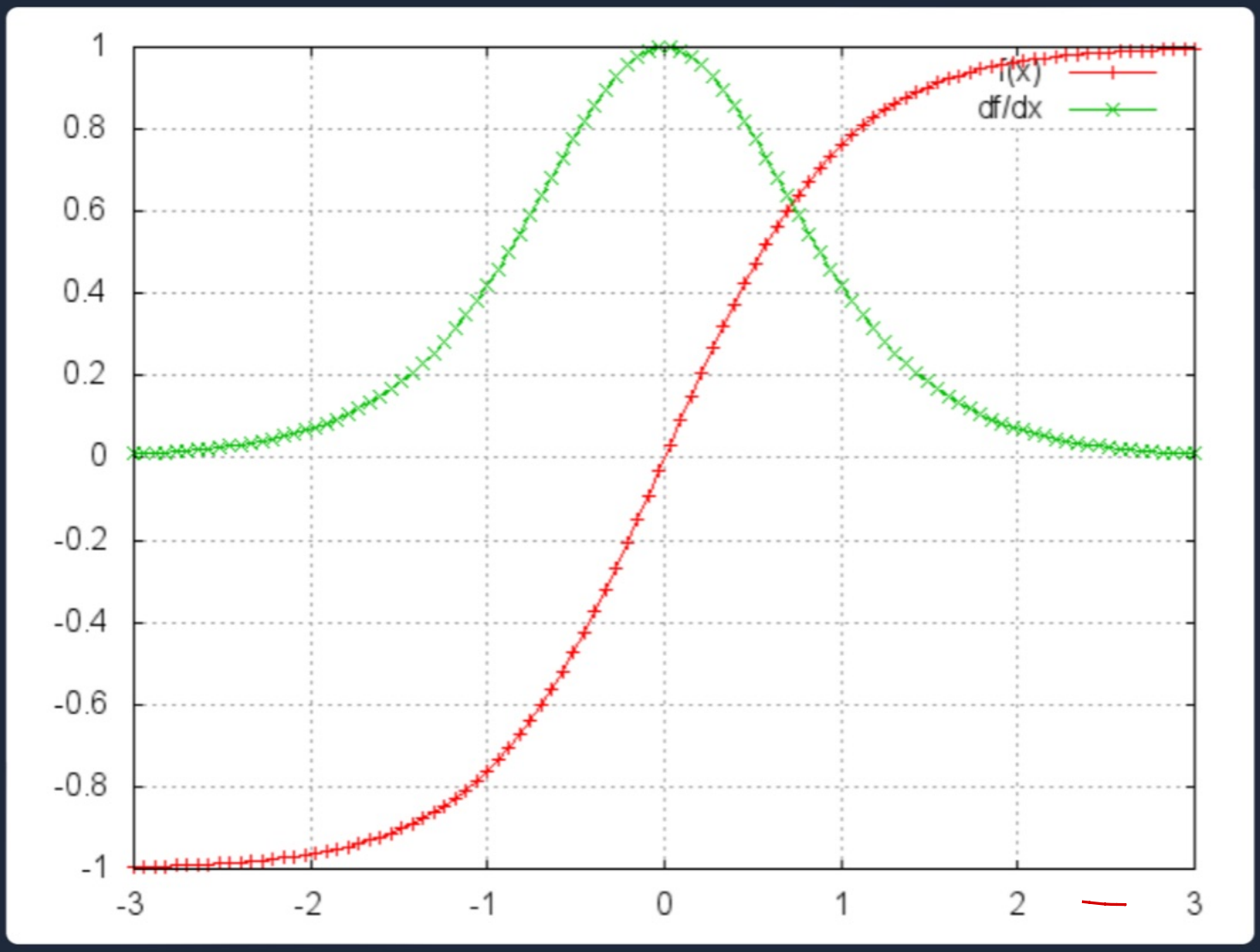
$$\frac{\partial L_t}{\partial w_{xh}} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_k} \cdot \frac{\partial h_k}{\partial w_{xh}}$$

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

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:

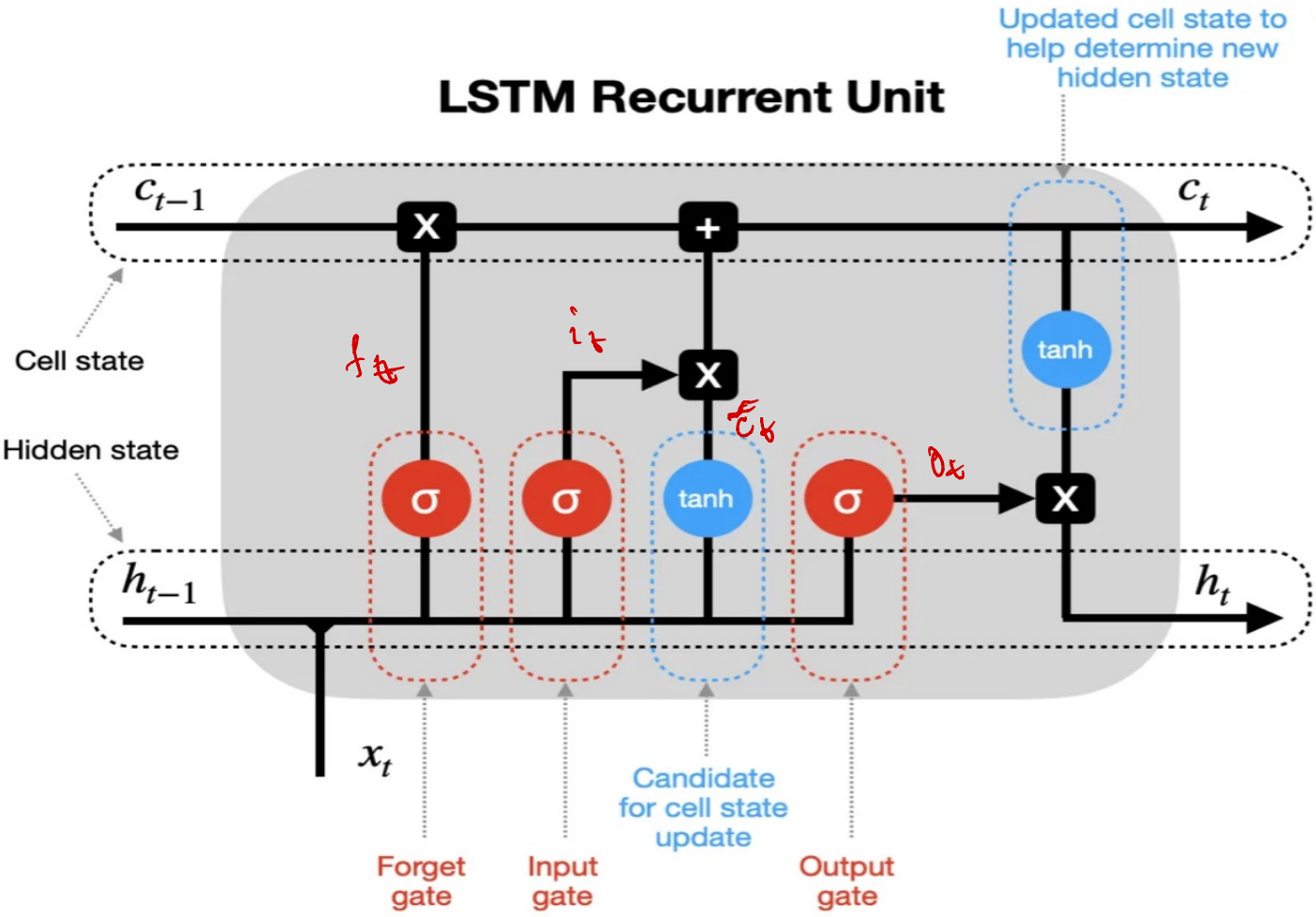


You can see that the $\tanh$ and sigmoid functions have derivatives of 0 at both ends. They approach a flat line. When this happens we say the corresponding neurons are saturated. They have a zero gradient and drive other gradients in previous layers towards 0. Thus, with small values in the matrix and 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.

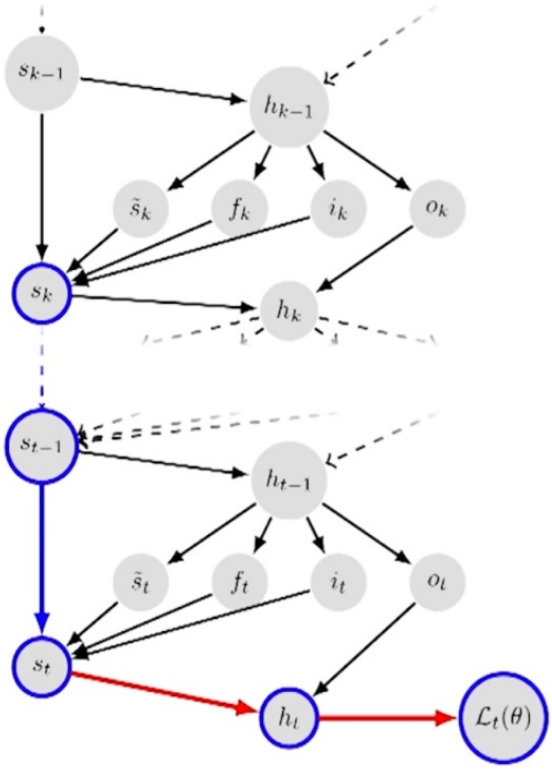
1. **Hidden state & new inputs** — hidden state from a previous timestep (h_t-1) and the input at a current timestep (x_t) are combined before passing copies of it through various gates.
2. **Forget gate** — this gate controls what information should be forgotten. Since the sigmoid function ranges between 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:

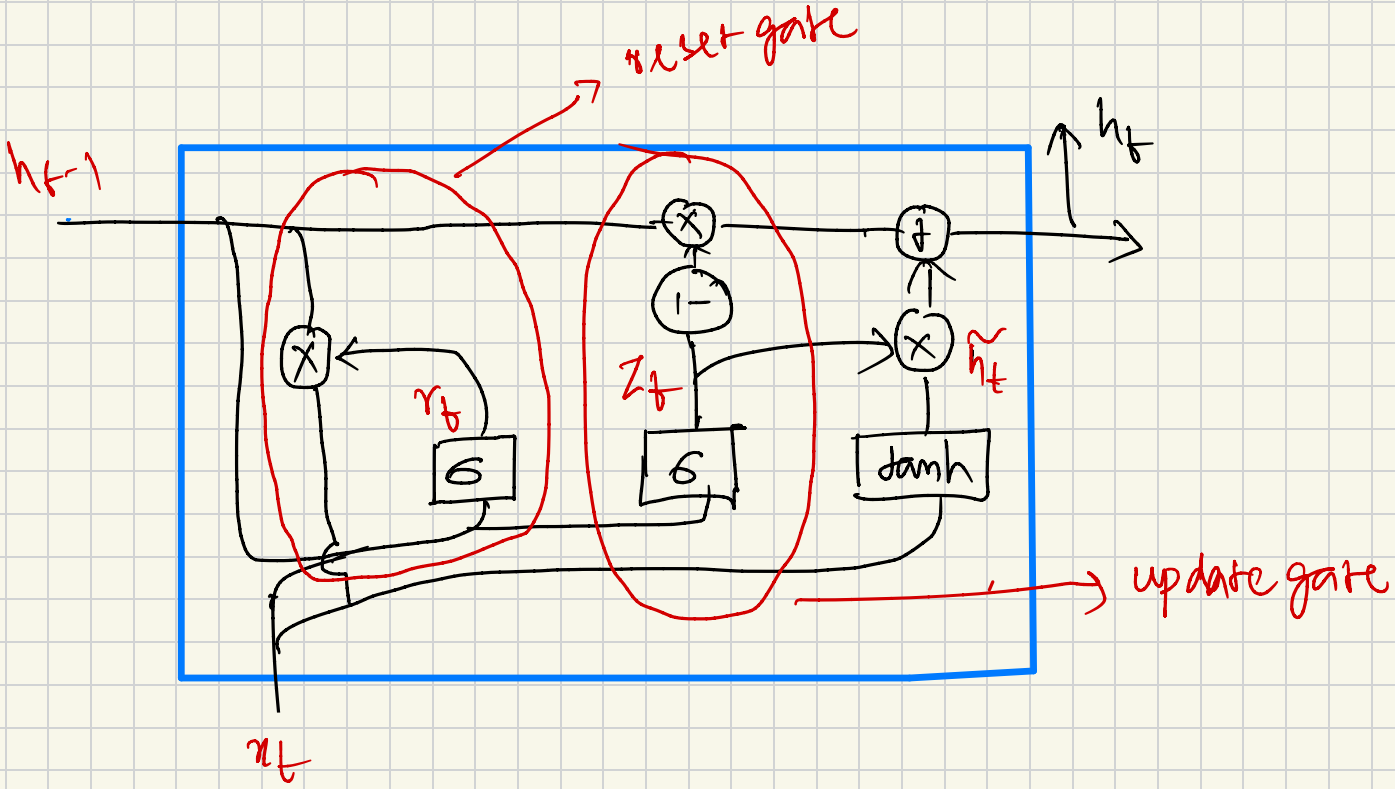
$$\partial h_t/\partial h_{t-k} \approx \prod_{i=0}^{k-1} \gamma(.)w_{oh} + c_{t-1}\gamma(.)w_{fh} + \gamma(.)w_{ch} + \gamma(.)w_{ih}$$

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

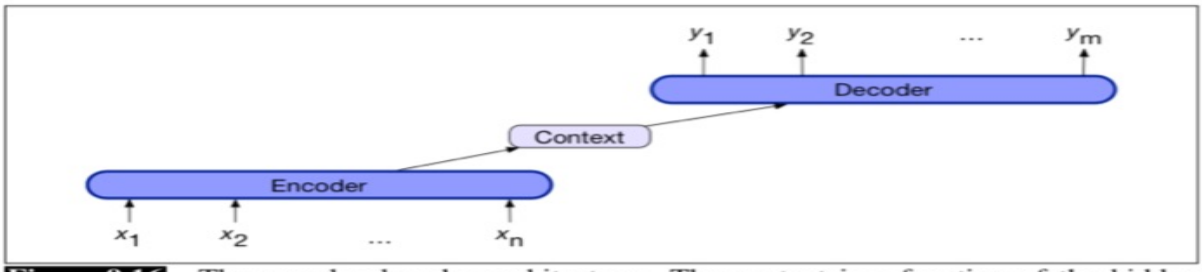


Figure 9.16 The encoder-decoder architecture. The context is a function of the hidden representations of the input, and may be used by the decoder in a variety of ways.

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).

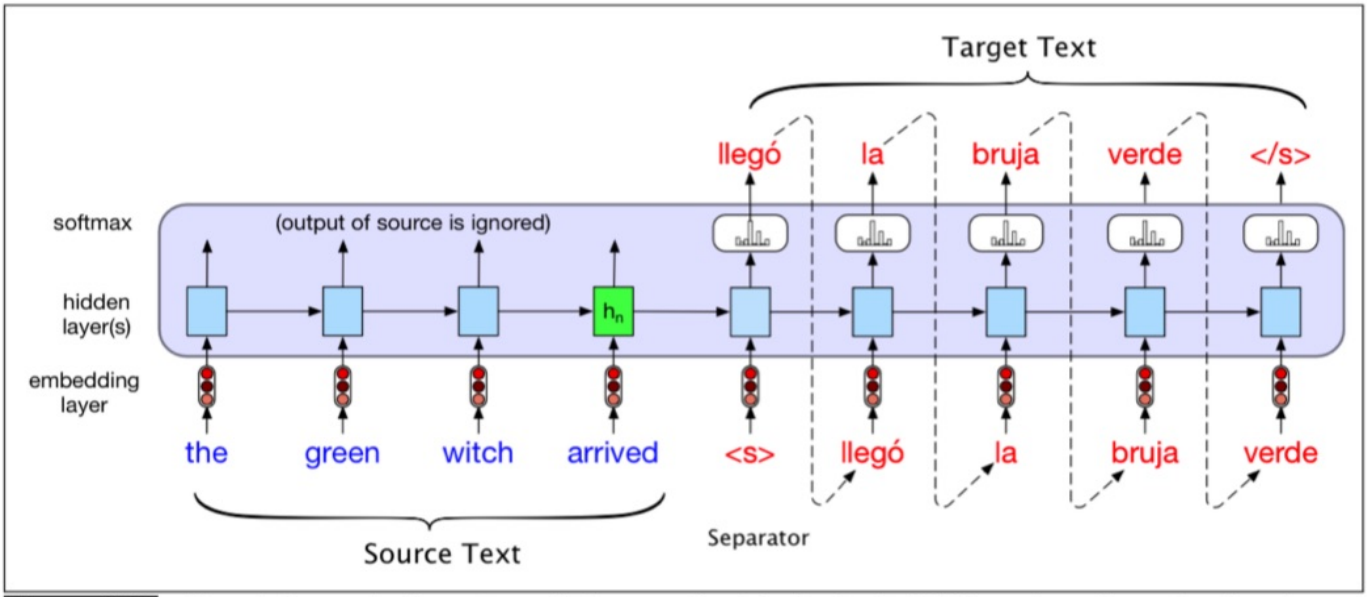


Figure 9.17 Translating a single sentence (inference time) in the basic RNN version of encoder-decoder approach to machine translation. Source and target sentences are concatenated with a separator token in between, and the decoder uses context information from the encoder's last hidden state.

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.

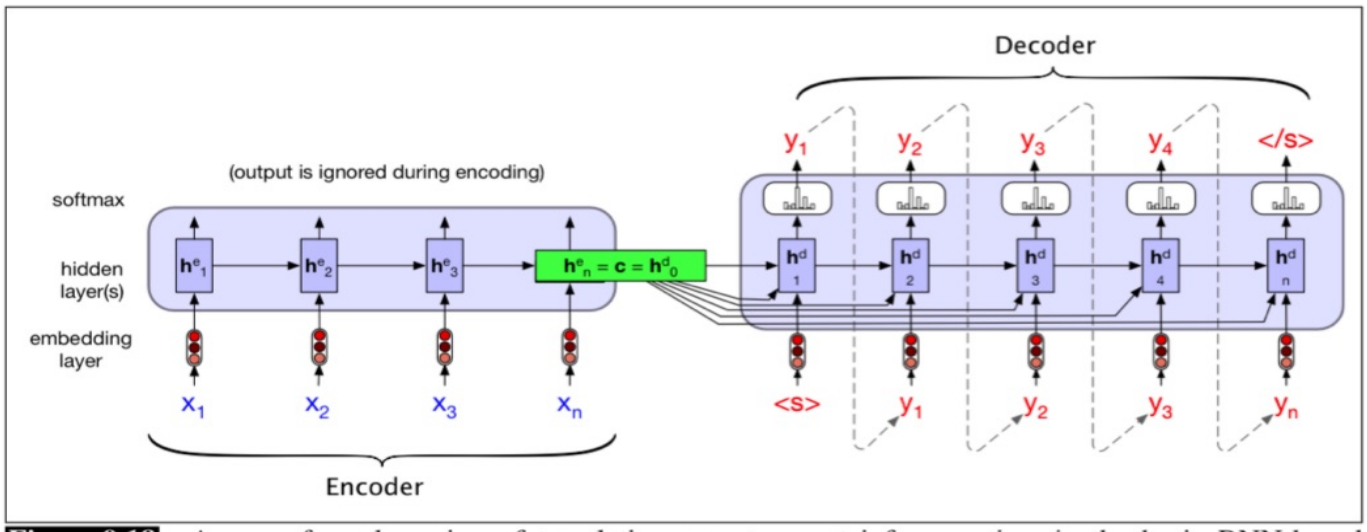


Figure 9.18 A more formal version of translating a sentence at inference time in the basic RNN-based encoder-decoder architecture. The final hidden state of the encoder RNN, h_n^e, serves as the context for the decoder in its role as h_0^d in the decoder RNN.

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

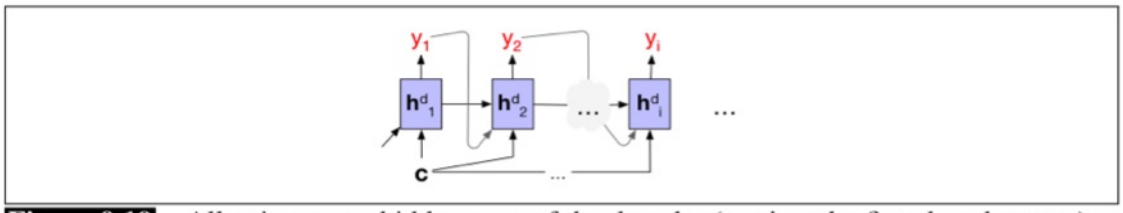


Figure 9.19 Allowing every hidden state of the decoder (not just the first decoder state) to be influenced by the context c produced by the encoder.

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