

# NLP Models

---

*Including LSTMs and GRUs*

*Author: Anton Zhitomirsky*

## Contents

<b>1 Sparsity</b>	<b>3</b>
1.1 Problem . . . . .	3
1.2 Add-one smoothing . . . . .	3
1.2.1 Problems . . . . .	4
1.3 Back-off smoothing . . . . .	4
1.3.1 Problems . . . . .	4
1.4 Interpolation . . . . .	4
<b>2 Feed-forward neural language models</b>	<b>5</b>
<b>3 Vanilla RNNs for language modelling</b>	<b>6</b>
3.1 Many-to-many RNN . . . . .	6
3.1.1 Teacher forcing . . . . .	6
3.1.2 Weight tying, reducing the no. of parameters . . . . .	7
<b>4 Bi-directional RNNs</b>	<b>7</b>
4.1 Bi-directional RNNs . . . . .	7
4.2 Multi-layered RNNs . . . . .	8
4.3 Bi-directional multi-layered RNNs . . . . .	9
<b>5 LSTMs</b>	<b>9</b>
5.1 How it works . . . . .	10
5.1.1 Step 1: we start from what we know already . . . . .	10
5.1.2 Step 2: we define our cell state . . . . .	10
5.1.3 Step 2. The forget gate . . . . .	10
5.1.4 Step 2. The input gate . . . . .	10
5.1.5 Step 3. prepare next hidden state . . . . .	10
5.2 Full equations . . . . .	10
5.2.1 Dimensions . . . . .	11
5.3 Why do LSTMs help with vanishing gradients? . . . . .	11

<b>6 GRU</b>	<b>11</b>
6.1 How it works . . . . .	11
6.1.1 Step 1. Current value . . . . .	11
6.1.2 Step 1. Update gate . . . . .	12
6.1.3 Step 2. Long term memory . . . . .	12
6.1.4 Step 2. Reset gate . . . . .	12
6.1.5 No output gate . . . . .	12
<b>Bibliography</b>	<b>12</b>

# 1 Sparsity

## 1.1 Problem

WSJ corpus: built over 10 years ago. What would happen if tested on today's News articles?

If certain words are absent then the probability in the n-gram is zero. We can use smoothing to solve this or give it unknown word tokens.

## 1.2 Add-one smoothing

	i	want	to	eat	chinese	food	lunch	spend
i	6	828	1	10	1	1	1	3
want	3	1	609	2	7	7	6	2
to	3	1	5	687	3	1	7	212
eat	1	1	3	1	17	3	43	1
chinese	2	1	1	1	1	83	2	1
food	16	1	16	1	2	5	1	1
lunch	3	1	1	1	1	2	1	1
spend	2	1	2	1	1	1	1	1

(a) Counts after one-smoothing (we've added +1 to EVERYTHING)

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

(b) probabilities before one-smoothing

	i	want	to	eat	chinese	food	lunch	spend
i	0.0015	0.21	0.00025	0.0025	0.00025	0.00025	0.00025	0.00075
want	0.0013	0.00042	0.26	0.00084	0.0029	0.0029	0.0025	0.00084
to	0.00078	0.00026	0.0013	0.18	0.00078	0.00026	0.0018	0.055
eat	0.00046	0.00046	0.0014	0.00046	0.0078	0.0014	0.02	0.00046
chinese	0.0012	0.00062	0.00062	0.00062	0.00062	0.052	0.0012	0.00062
food	0.0063	0.00039	0.0063	0.00039	0.00079	0.002	0.00039	0.00039
lunch	0.0017	0.00056	0.00056	0.00056	0.00056	0.0011	0.00056	0.00056
spend	0.0012	0.00058	0.0012	0.00058	0.00058	0.00058	0.00058	0.00058

(c) probabilities after one-smoothing

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

(d) total instances of words

Figure 1: difference between smoothing and not

$$P_{add-1}(w_n|w_{n-1}) = \frac{C(w_{n-1}, w_n) + 1}{C(w_{n-1}) + V}$$

Given words with sparse statistics, they steal probability mass from more frequent words. This is because smaller instances of vocabulary are more influenced by the  $V$  and  $+1$  term appearing in the new fraction. However, for larger occurrences they shrink because the nominator is large (so  $+1$ ) won't change it by much, but the denominator grows larger, making the overall probability less.

The change here is great between the highlighted terms in Figure 1. “If we have very few instances of the word ‘want’ the smoothing will impact this a lot and this number will therefore change a lot. Whereas, if we have a lot of counts within the word ‘i’ then it won't change the probability a lot.” Indeed, looking at the total instances of word  $i$ , the denominator is so large already that the number wasn't impacted, and ‘want’ was impacted.

### 1.2.1 Problems

Although easy to implement, it takes too much probability mass from more likely occurrences (see Figure 1) and assigns too much probability to unseen events. Therefore, we could try  $+k$  smoothing with a smaller value of  $k$ .

## 1.3 Back-off smoothing


If we don't have any occurrences in our current (trigram) model, we can back-off and see how many occurrences there are in the smaller (bigram) model.

**Definition 1.1** (Back off smoothing “sutpid back-off”).

$$S(w_i|w_{i-2}w_{i-1}) = \begin{cases} \frac{C(w_{i-2}w_{i-1}w_i)}{C(w_{i-2}w_{i-1})}, & \text{if } C(w_{i-2}w_{i-1}w_i) > 0 \\ 0.4 \cdot S(w_i|w_{i-1}), & \text{otherwise} \end{cases}$$

$$S(w_i|w_{i-1}) = \begin{cases} \frac{C(w_{i-1}w_i)}{C(w_{i-1})}, & \text{if } C(w_{i-1}w_i) > 0 \\ 0.4 \cdot S(w_i), & \text{otherwise} \end{cases}$$

$$s(w_i) = \frac{C(w_i)}{N}$$

 Note, this is not a probability, but a score

Where  $N$  is the number of words in the text

### 1.3.1 Problems

suppose that the bigram “a b” and the unigram “c” are very common, but the trigram “a b c” is never seen. Since “a b” and “c” are very common, it may be significant (that is, not due to chance) that “a b c” is never seen.

Perhaps it's not allowed by the rules of the grammar. Instead of assigning a more appropriate value of 0, the method will back off to the bigram and estimate  $P(c|b)$ , which may be too high

## 1.4 Interpolation

Combine evidence from different n-grams:

**Definition 1.2** (Interpolation).

$$P_{interp}(w_i|w_{i-2}w_{i-1}) = \lambda_1 P(w_i|w_{i-2}w_{i-1}) + \lambda_2 P(w_i|w_{i-1}) + \lambda_3 P(w_i), \quad \text{where } \lambda_1 + \lambda_2 + \lambda_3 = 1$$

## 2 Feed-forward neural language models

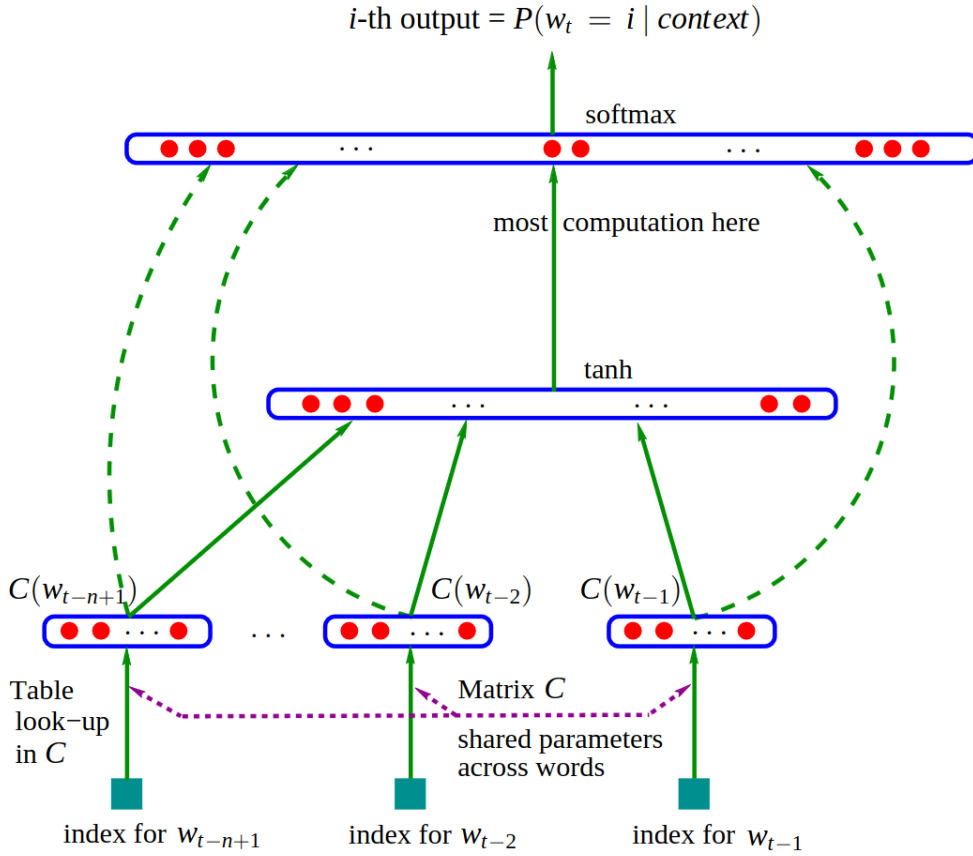
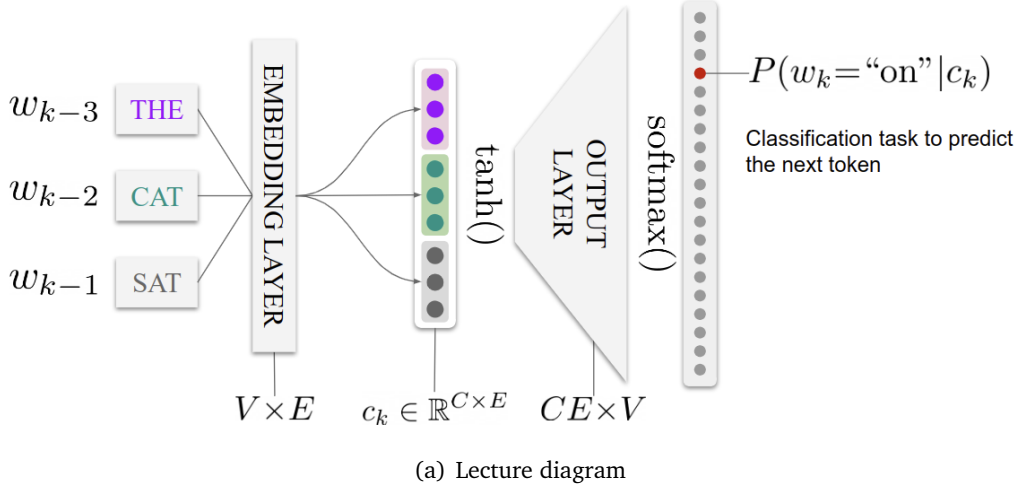
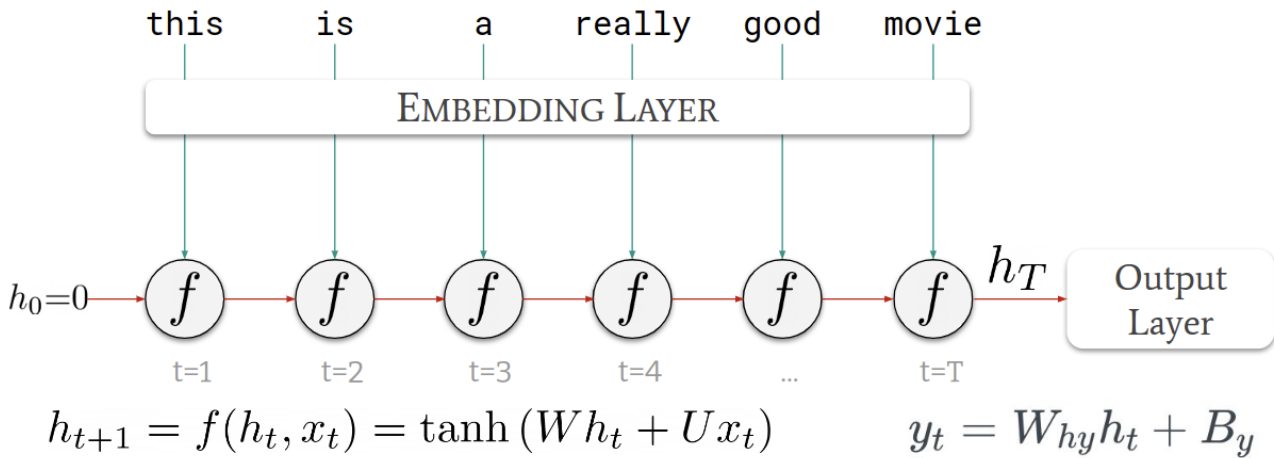


Figure 2: FFLM diagrams

It approximates history with last  $C$  words ( $C$  affects model size) e.g. 4-gram FFLM has a context size of 3. The context is formed by concatenating word embeddings.

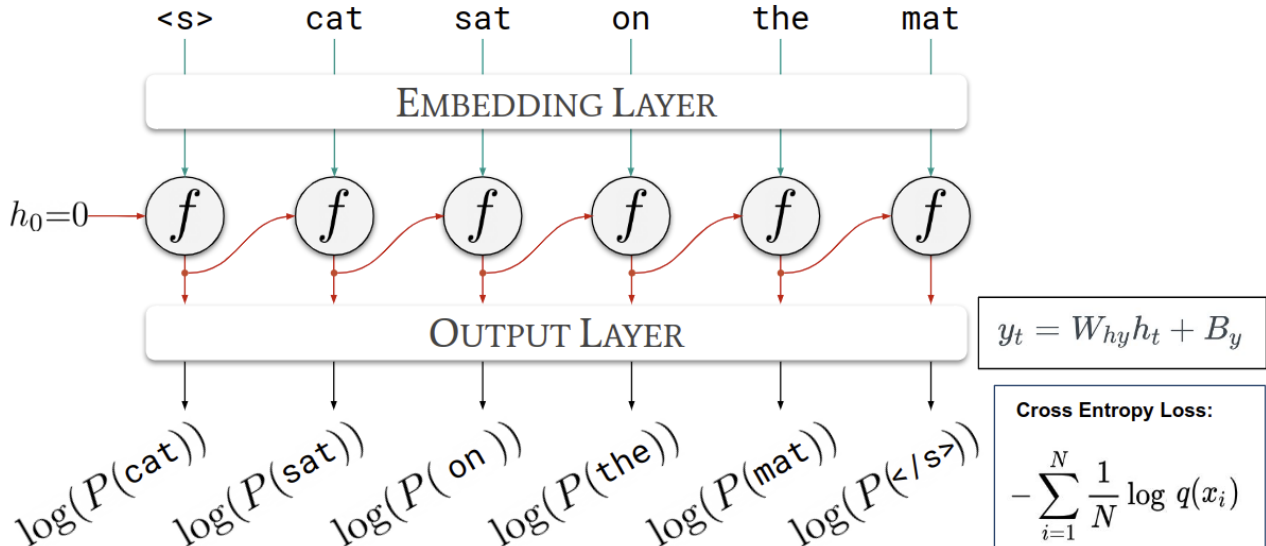
### 3 Vanilla RNNs for language modelling



Before we had classification, each function took a hidden state and output this into the output layer. We could do this differently, but instead of taking the final we could max on each dimension also.

#### 3.1 Many-to-many RNN

Every input has a label, in language modelling you predict the next word. The LM loss is predicted from the cross-entropy losses from each timestep.



In addition to passing the hidden state to the next function, you also use it in the output layer. The size of  $W$ : let's say the hidden state has  $H$  dimensions, it must go to the size of the vocab, because you're predicting which word is going to come next out of the whole vocab, so it is  $V \times H$ .

We take the log probabilities because you want to use the cross entropy loss (which is the average across the outputs)

##### 3.1.1 Teacher forcing

In the event where this network predicts an incorrect label some way along the prediction, we force it to use the actual expected label because otherwise the network would be learning some strange

behaviour if you are given incorrect predictions somewhere along the line.  
If you don't use it, then there is instability in the model.

### 3.1.2 Weight tying, reducing the no. of parameters

We have very big models; big vocabulary causes matrices of dimension  $V \times \cdot$ .

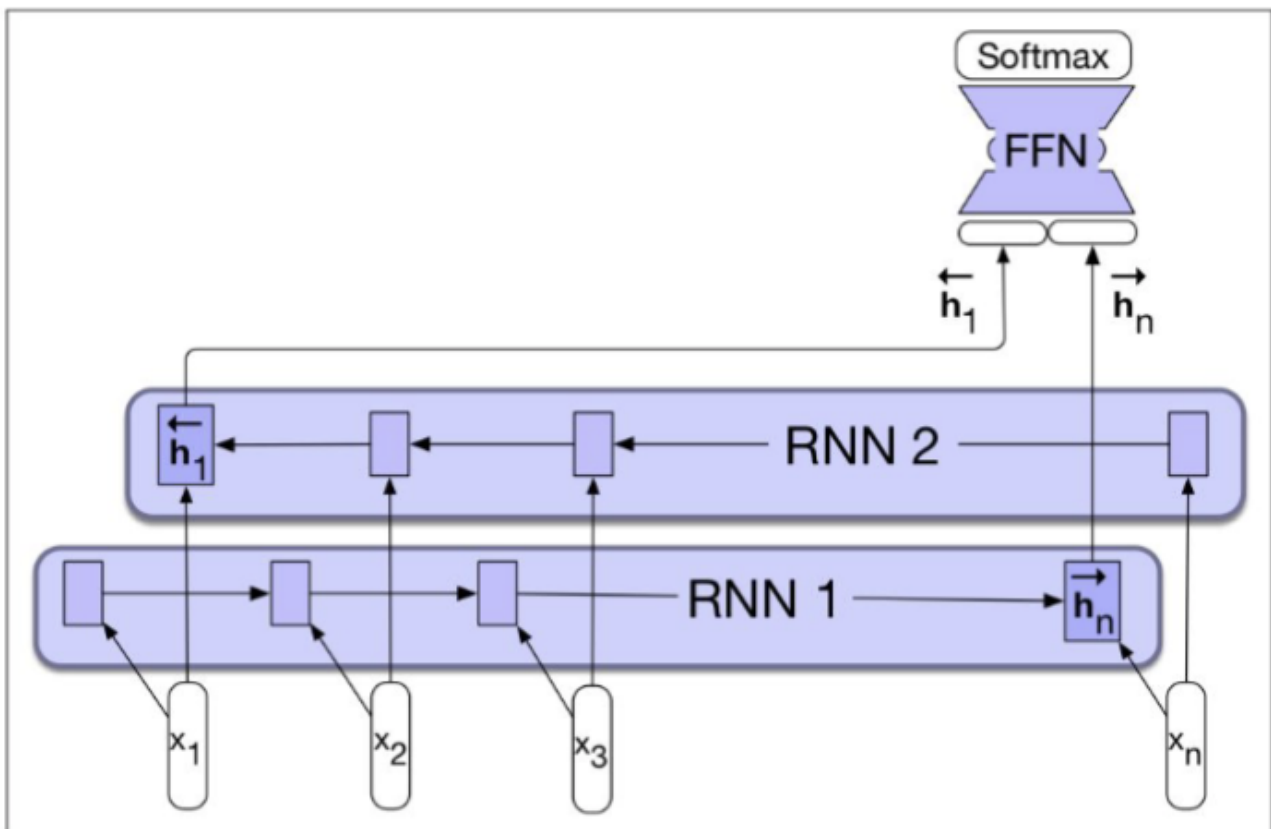
We can use the same embedding weights in our output layer: ( $E$  has dimensions:  $H \times |V|$ ). If you have a matrix  $E$  that maps from a one-hot vector into an embedding, you can use  $E$  transpose in the output layer.

$$\begin{aligned} e_t &= Ex_t \\ h_{t+1} &= \tanh(Wh_t + Ue_t) \\ y_t &= \text{softmax}(E^\top h_t) \end{aligned} \quad E^\top \text{ has dimensions : } |V| \times H$$

$U$  could be from the size of the imbedding to the size of the hidden state, but if we do the weight tying then it has to be  $H \times H$ .

## 4 Bi-directional RNNs

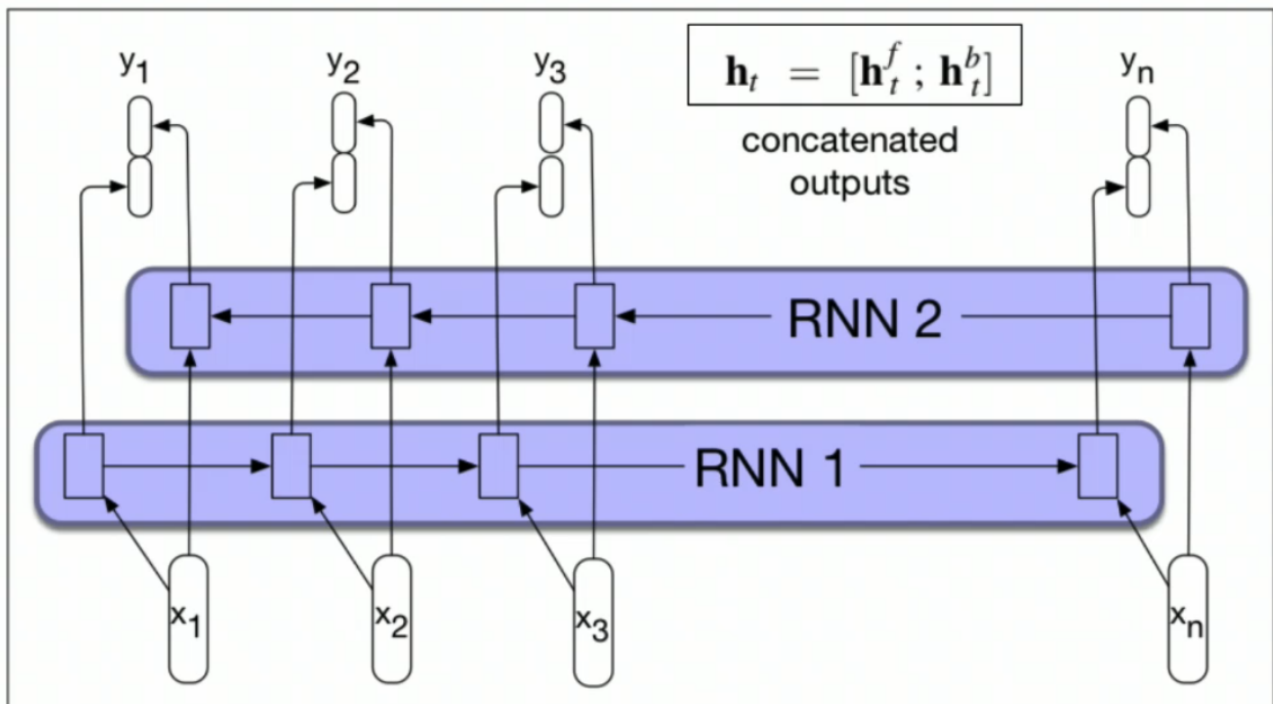
### 4.1 Bi-directional RNNs



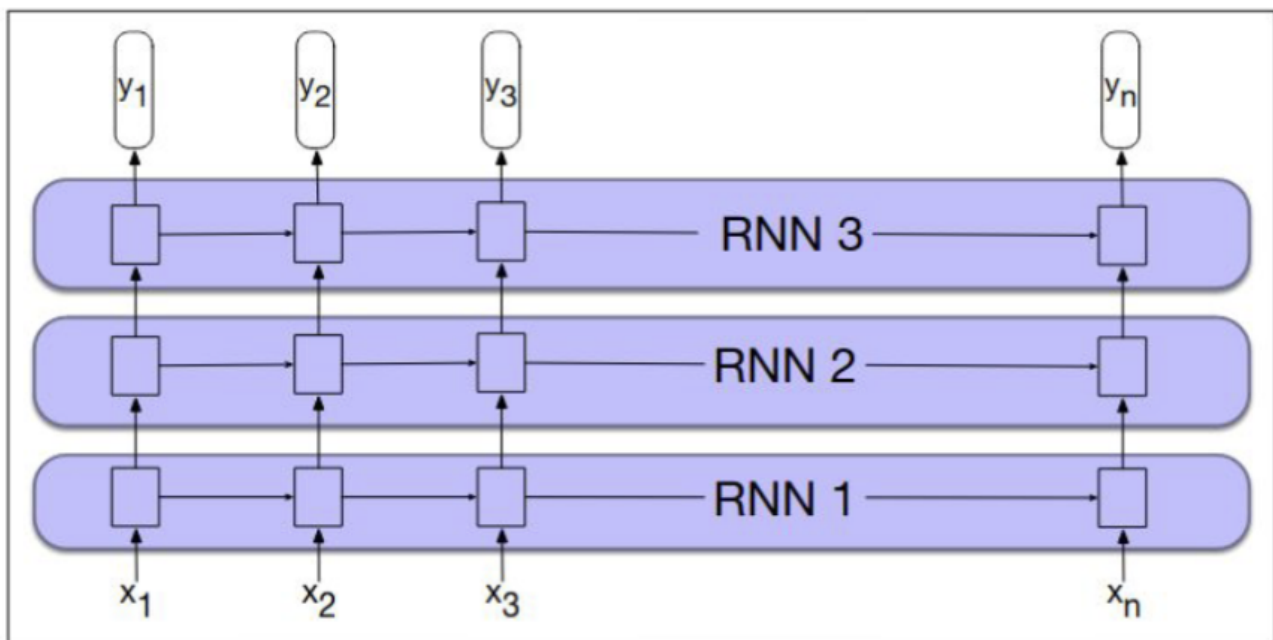
We can concatenate the representations at the end of the RNNs for both directions.

When comparing the number of parameters in this vs a single directional rnn, it doubles. The output layer also doubles (because we are given a matrix  $H \times O$  twice from each direction making the output layer have dimension  $H \times H \times O$ ).

## 4.2 Multi-layered RNNs

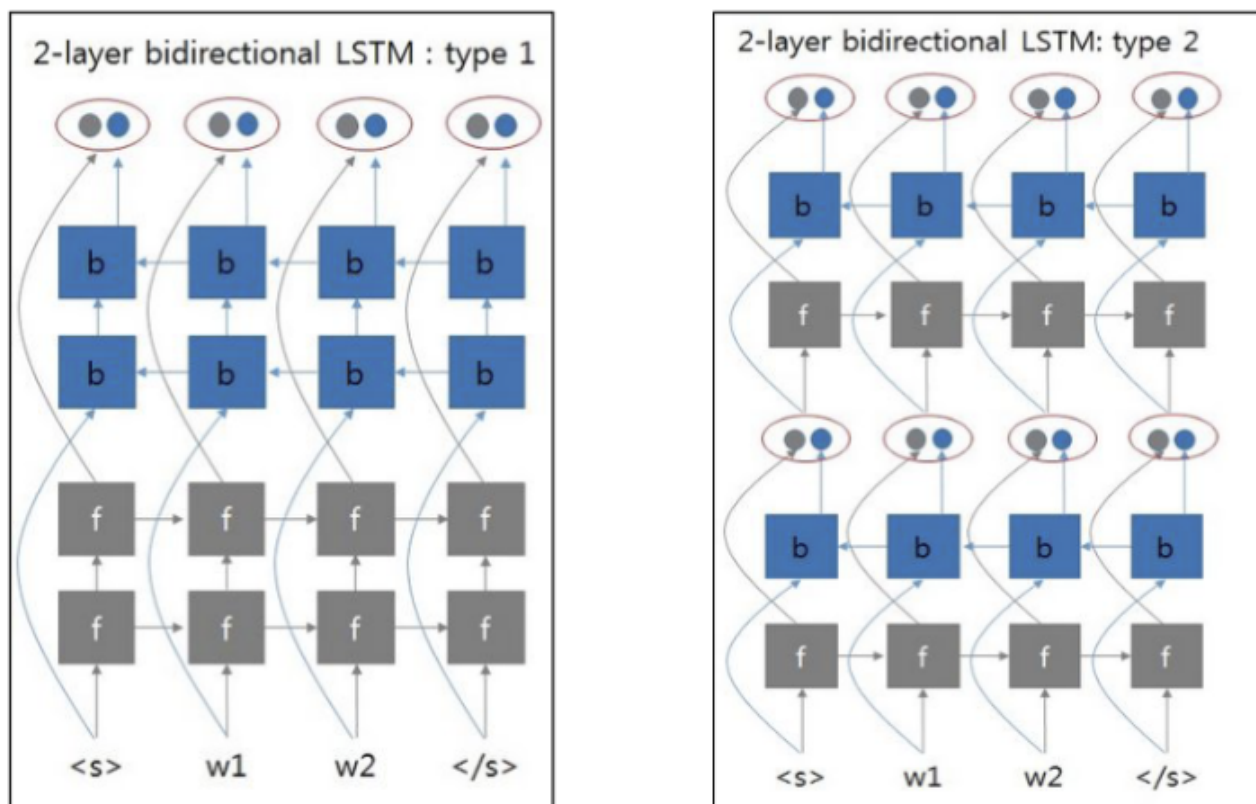


We can also predict every single token (every word we input). We concatenate them and can make a prediction at each time step. Then, following this, we can stack networks on-top of each other.



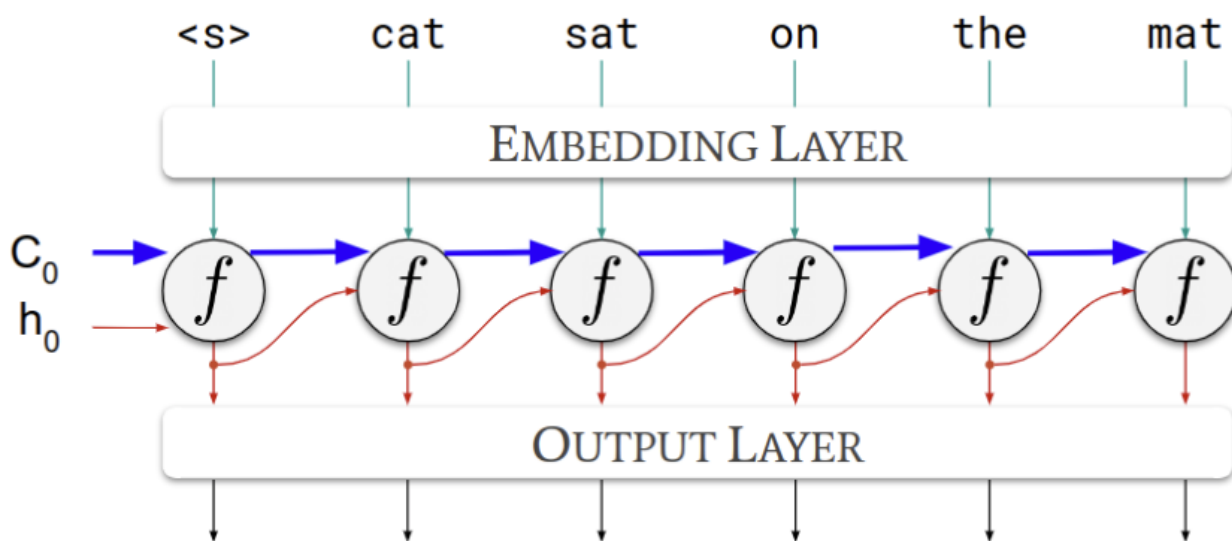


## 4.3 Bi-directional multi-layered RNNs



Currently forward output and backward output is concatenated after each layer. But, for language modeling, we need independent forward rnn and backward rnn until the last layer and output concatenation is only need at the last layer.

## 5 LSTMs



Cell states ( $C_t$ ) represent 'long term memory', and hidden states ( $h_t$ ) is current working memory (same as for vanilla RNN).

## 5.1 How it works

### 5.1.1 Step 1: we start from what we know already

We used vanilla rnns before. The current value is:

$$g_t = \tanh(W_{ig}x_t + W_{hg}h_{t-1} + b_g)$$

### 5.1.2 Step 2: we define our cell state

With LSTM, we also want a cell state for long term memory

$$c_t = f_t * c_{t-1} + i_t * g_t$$

We define this by the cell state from the previous time step  $c_{t-1}$  and add it to some of what we got from this time step  $g_t$  with some multiplier for each of those.

The model can now choose to keep this state the same or forget everything.

Note that  $f_t \wedge i_t$  are in terms of  $t$  so they're not the same every time step - so we can choose when to remember or when to forget. The decision should be based on what the input is for that time step and what the context is up to that point.

### 5.1.3 Step 2. The forget gate

$f_t$  is the forget gate, and is a vector between 0 and 1. It needs to consider  $x_t \wedge h_{t-1}$  and returns a vector between 0 and 1.

$$f_t = \sigma(W_{if}x_t + W_{hf}h_{t-1} + b_f)$$

### 5.1.4 Step 2. The input gate

Similarly, with  $i_t$  it is the input gate and is a vector between 0 and 1.

$$i_t = \sigma(W_{ii}x_t + W_{hi}h_{t-1} + b_i)$$

### 5.1.5 Step 3. prepare next hidden state

Here, we are transforming  $c_t$  to give us  $h_t$ .

$$h_t = o_t * \tanh(c_t)$$

## 5.2 Full equations

$$\begin{aligned} i_t &= \sigma(W_{ii}x_t + W_{hi}h_{t-1} + b_i) \\ f_t &= \sigma(W_{if}x_t + W_{hf}h_{t-1} + b_f) \\ o_t &= \sigma(W_{io}x_t + W_{ho}h_{t-1} + b_o) \\ g_t &= \tanh(W_{ig}x_t + W_{hg}h_{t-1} + b_g) \end{aligned}$$

$$c_t = f_t * c_{(t-1)} + i_t * g_t$$

$$h_t = o_t * \tanh(c_t)$$

### 5.2.1 Dimensions

If we have  $E$  dimensional embeddings and  $H$  dimensional vectors.

- $W_{ii} = (H \times E)$  is the weight matrix applied to the input. We go from the embeddings to  $i_t$ . We know that  $i_t$  is  $H$  dimensional
- $W_{hi} = (H \times H)$  is the weight matrix applied to the input gate whose dimensions must match the addition of the step prior,

#### EXAMPLE

If we have 100 dimensional hidden states, 500 dimensional embeddings and a classification task with 5 classes (no bias term in the output layer), then excluding any embedding layer, how many parameters do we have?

Look at the dimensions of the weight matrices, we have  $100 \times 500$  and  $100 \times 100$ . The bias term in these intermediate hidden layers is therefore 100 also. Then similarly for the rest so  $(100 \times 500 + 100 \times 100 + 100) \times 4$ .

In the output layer, we have 5 output classes, so we must convert from having  $H$  dimensions to 5 dimensions in our weight matrix i.e.  $100 \times 5$ .

Therefore, we have 24,900 parameters.

#### EXAMPLE BIDIRECTIONAL

Same as above, only we multiply by 2 in the hidden layers, and 2 once again in the output layer because we're getting data sent in from two streams, each without output dimension  $H$ .

## 5.3 Why do LSTMs help with vanishing gradients?

The gradients through the cell states are hard to vanishing

1. Additive formula means we don't have repeated multiplication of the same matrix (we have a derivative that's more 'well behaved')
2. The forget gate means that our model can learn when to let the gradients vanish, and when to preserve them. This gate can take different values at different time steps.

## 6 GRU

### 6.1 How it works

Similar, only instead of having two distinct gates, we have  $z_t \wedge (1 - z_t)$ . Also we have a update gate  $r_t$ .

#### 6.1.1 Step 1. Current value

$$g_t = \tanh(W_{ig}x_t + r_t * (W_{hg}h_{t-1} + b_g))$$

### 6.1.2 Step 1. Update gate

How much do we consider the previous hidden state

$$r_t = \sigma(W_{ir}x_t + W_{hr}h_{t-1} + b_r)$$

### 6.1.3 Step 2. Long term memory

Our long term memory is only controlled by a reset gate ( $z_t$ )

$$h_t = (1 - z_t) * g_t + z_t * h_{t-1}$$

### 6.1.4 Step 2. Reset gate

$$z_t = \sigma(W_{iz}x_t + W_{hz}h_{t-1} + b_z)$$

### 6.1.5 No output gate

## References

- [1] Yoshua Bengio et al. "A neural probabilistic language model". In: 3.null (Mar. 2003), pp. 1137–1155. ISSN: 1532-4435.