A Review of RNNs and Attention

Tarushii Goel*

March 2022

1 RNNs

"Humans don't start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence.

Traditional neural networks can't do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. It's unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.

Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist." (Christopher Olah's explanation)

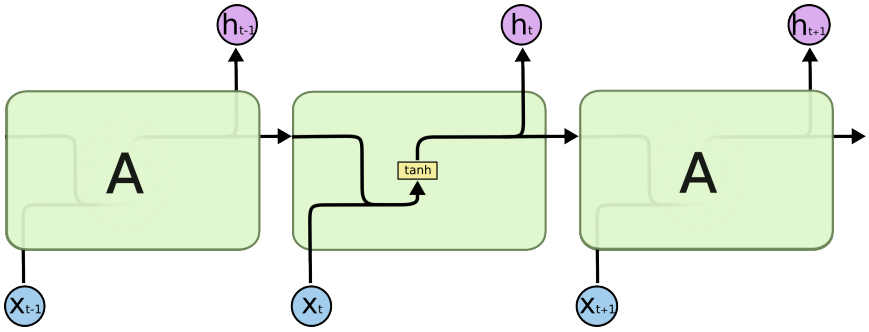


Figure 1: A standard RNN

What makes an RNN different from standard networks is that the previous hidden state from the last input is passed into the function for the current state. For example, with the sentence "I like dogs", the word "I" is passed through the network and results in some output like normal. The previous hidden state that was computed for "I" is passed into the function to calculate the hidden state for "like", and so forth.

*Based on Vinay's RNN lecture

1.1 LSTMs

Our original RNN cell looked like this:

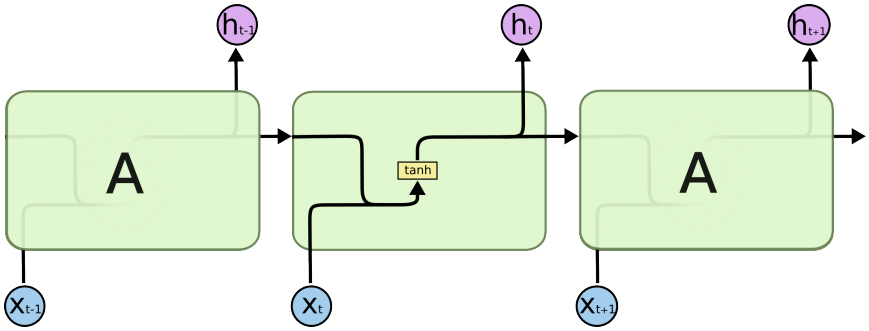


Figure 2: Original RNN Cell

Notice how only the input and the hidden state are controlling the outputs of the cell.

Unfortunately, in practice, RNNs are susceptible to the vanishing gradient and the exploding gradient problem. Essentially, through backpropagation, earlier layers either face an exponentially small gradient or large gradient. This means that in sequences of data, at the current time t, the RNN is likely to forget relevant data from long before.

An LSTM, or Long-short term memory network, seeks to solve these problems. There are four main parts to an LSTM: the cell state (c_t), the input gate (i_t), the forget gate (f_t), and the output gate (o_t).

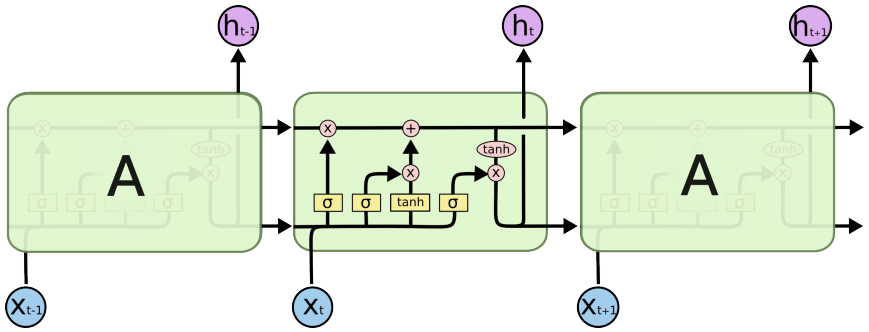


Figure 3: Overview of an LSTM Cell

This cell looks a lot more complicated than the vanilla RNN we saw before. However, these added aspects improve the RNN by a lot so lets walk through them.

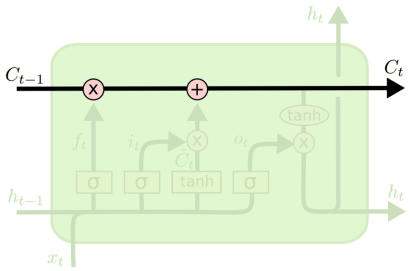
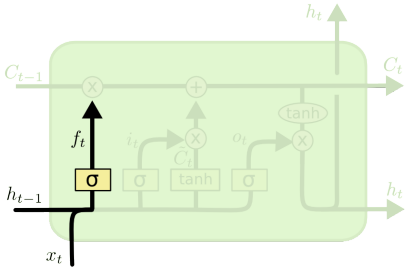


Figure 4: LSTM Cell State

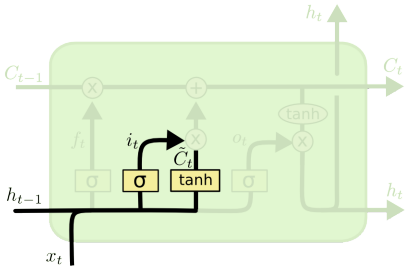
The cell state serves as the main flow across the cells. The gates modify the information that is passed through the cell state. Note, the X denotes element-wise multiplication (also known as the Hadamard product) and the + denotes element-wise addition.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure 5: LSTM Forget Gate

The forget gate chooses whether or not the information should be added to the cell state. This gate is a sigmoid layer that takes in h_{t-1} and x_t. This forget gate output will be multiplied into the cell state, so if the forget gate has values of 0, then it would forget the information and if it has values of 1, then it would remember all the information.

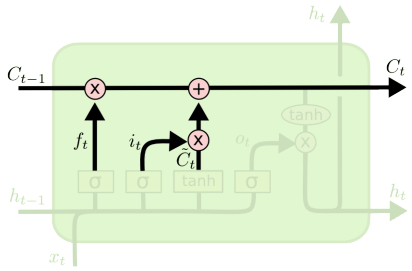


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Figure 6: LSTM Input Gate

Next is the input gate. This passes the input from the h_{t-1} and x_t values

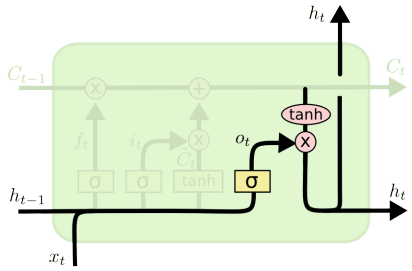
and combines it with the $\tilde{C}_t$, or new candidate values. $\tilde{C}_t$ takes in the inputs and looks to find new values that are possible. Notice that this is a hyperbolic tan function, which outputs values from -1 to 1. This allows the layer to give negative correlations to a value as well.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 7: LSTM Combining Values

These values are all combined and put into the cell state which passes the values through.



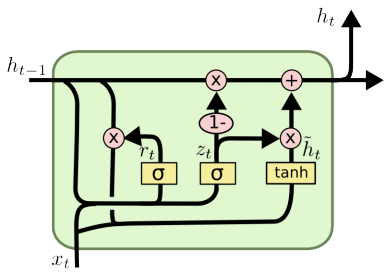
$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 8: LSTM Output Gate

Lastly, there is the output gate. This is just another sigmoid layer, which is combined with the tanh of the cell state to give the h_t final value that gets passed into the next cell.

All these gates would be trained through backpropagation as they're just layers in a network. However, with all the layers, LSTMs can be computationally expensive, which leads us into our next section.

1.2 GRUs



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 9: GRU Cell

GRUs, or Gated recurrent units, combine the forget and input gate into one "update" gate. Additionally, it merges the cell state and the hidden state, and adds other changes for efficiency. Fewer layers to train means a faster model.

GRUs are a relatively new concept as well, being discovered by Cho, et al. in 2014. LSTMs, on the other hand, have been around since 1997, from the discovery by Hochreiter and Schmidhuber.

2 Attention

The main motivation for attention was Neural Machine Translation (NMT), which as you'll recall, can be performed through seq2seq networks composed of an encoder and decoder. The issue (again!) is long-term memory: when the sentences get increasingly large the chance that all of them can be effectively encoded into a low-dimensional vector becomes increasingly small. Attention directly links the hidden states from the encoder to the decoder input (these are kind of like information "highways", allowing the words to travel directly instead of through a roundabout route involving multiple hidden states).

Say, we have a source sequence $\mathbf{x}$ of length n and want to output a target sequence $\mathbf{y}$ of length m.

$$\begin{aligned} \mathbf{x} &= [x_1, x_2, \ldots, x_n] \\ \mathbf{y} &= [y_1, y_2, \ldots, y_m] \end{aligned} \tag{1}$$

If we use a bi-directional RNN as our decoder, at every point, we will have two hidden states for each word. One is the hidden state outputted right after the forward RNN takes the ith word as input, which includes information mostly about the ith word but also about all preceding words, and the other is the hidden state right after the backwards RNN takes the ith word as input, which includes information mostly about the ith word but also about all following words. These two hidden states are appended to get the hidden state for that word.

$$\boldsymbol{h}_i = [\overrightarrow{\boldsymbol{h}}_i^\top; \overleftarrow{\boldsymbol{h}}_i^\top]^\top, i = 1, \ldots, n \tag{2}$$

Now, we want to connect these hidden states directly to the decoder input. We will accomplish this by appending a context vector $\mathbf{c}_t$, a sum of hidden states of the encoder, weighted by alignment scores, to the decoder hidden state. The new hidden state is $\boldsymbol{s}_t = f(\boldsymbol{s}_{t-1}, y_{t-1}, \mathbf{c}_t)$ for the output word at position t, $t = 1, \ldots, m$:

$$\begin{aligned} \mathbf{c}_t &= \sum_{i=1}^{n} \alpha_{t,i} \boldsymbol{h}_i \\ \alpha_{t,i} &= \text{align}(y_t, x_i) \\ &= \frac{\exp(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_i))}{\sum_{i'=1}^{n} \exp(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_{i'}))}. \end{aligned} \tag{3}$$

2.1 Self-Attention

Self-attention is practically regular attention, but instead of the context vectors being calculated from the hidden states of one RNN and being the input to another RNN, the context vectors are calculated from the hidden states of a single RNN, *for that* RNN.

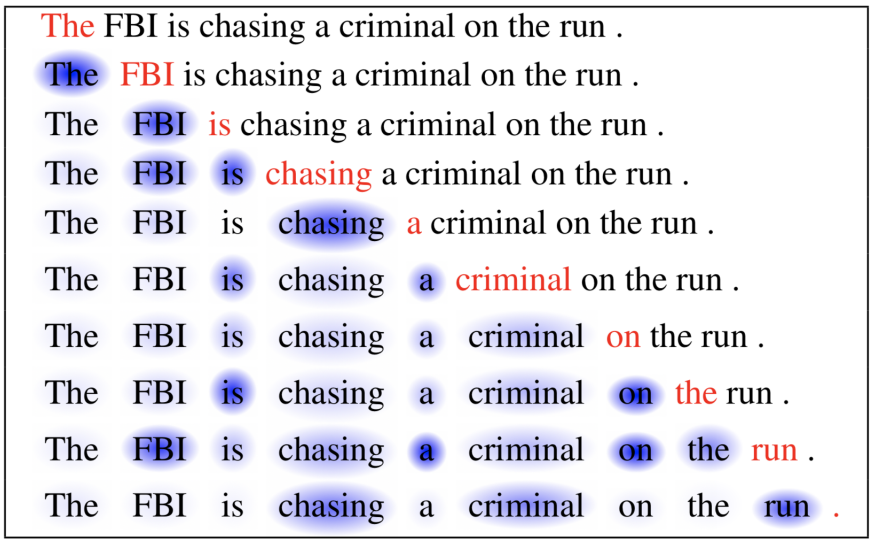


Figure 10: Self-Attention

2.2 Multi-Head Attention

Symbol	Meaning
d	The model size / hidden state dimension
h	The number of heads in multi-head attention layer.
L	The segment length of input sequence.
$\mathbf{X} \in R^{L \times d}$	The input sequence where each element has been mapped into an embedding vector of shape d, same as the model size.
$\mathbf{W}^k \in R^{d \times d_k}$	The key weight matrix.
$\mathbf{W}^q \in R^{d \times d_k}$	The query weight matrix.
$\mathbf{W}^v \in R^{d \times d_v}$	The value weight matrix. Often we have $d\_k = d\_v = d$.
$\mathbf{W}_i^k, \mathbf{W}_i^q \in R^{d \times d_k/h}; \mathbf{W}_i^v \in R^{d \times d_v/h}$	The weight matrices per head.
$\mathbf{W}^o \in R^{d_v \times d}$	The output weight matrix.
$\mathbf{Q} = \mathbf{X}\mathbf{W}^q \in R^{L \times d_k}$	The query embedding inputs.
$\mathbf{K} = \mathbf{X}\mathbf{W}^k \in R^{L \times d_k}$	The key embedding inputs.
$\mathbf{V} = \mathbf{X}\mathbf{W}^v \in R^{L \times d_v}$	The value embedding inputs.
$\mathbf{A} \in R^{L \times L}$	The self-attention matrix between a input sequence of lenght L and itself. $\mathbf{A} = \text{softmax}(\mathbf{Q}\mathbf{K}^\top / \sqrt{d_k})$.
$a_{ij} \in \mathbf{A}$	The scalar attention score between query $\mathbf{q}_i$ and key $\mathbf{k}_j$

$$\text{MultiHeadAttention}(\mathbf{X}_q, \mathbf{X}_k, \mathbf{X}_v) = [\text{head}_1; \ldots; \text{head}_h]\mathbf{W}^o$$
$$\text{where head}_i = \text{Attention}(\mathbf{X}_q\mathbf{W}_i^q, \mathbf{X}_k\mathbf{W}_i^k, \mathbf{X}_v\mathbf{W}_i^v)$$

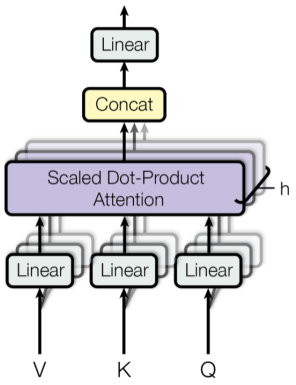


Figure 11: The multi-head scaled dot-product attention mechanism

3 Sources

- Lillian Weng's Transformer Blog Post
- Lillian Weng's Attention Blog Post
- Saahith's Attention Lecture
- Vinay's RNN Lecture
- Christopher Olah's LSTM Walkthrough