LSTM Notes By Soham.

2024-07-26

Now as we know RNN deals with sequential information but in practice, it generally loosed the information as it goes further in the network cause of the vanishing or exploding gradient problem.

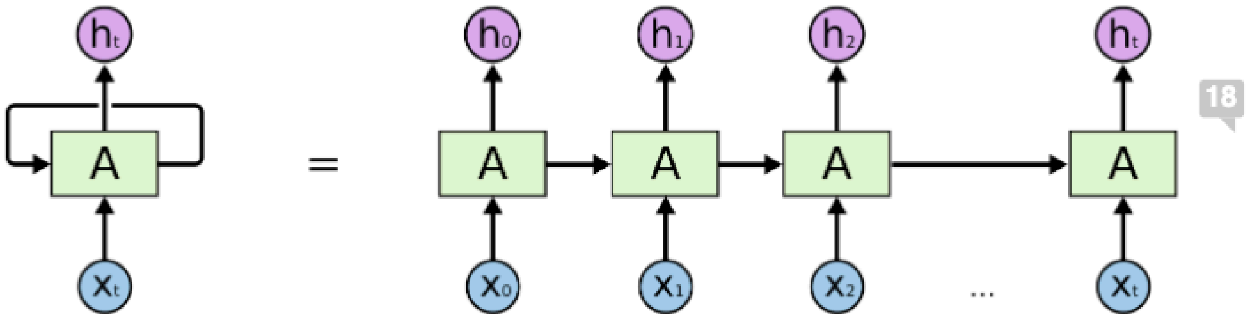
To tackle this we have LSTM, that is a special RNN structure that is more efficient with long sequences and it can dodge the problem of data loss more efficiently.
Lets take an example where RNN works great and other where it fails.

Consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in “the clouds are in the *sky*,” we don’t need any further context – it’s pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it’s needed is small, RNNs can learn to use the past information.

But there are also cases where we need more context. Consider trying to predict the last word in the text “I grew up in France… I speak fluent *French*.” Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It’s entirely possible for the gap between the relevant information and the point where it is needed to become very large.

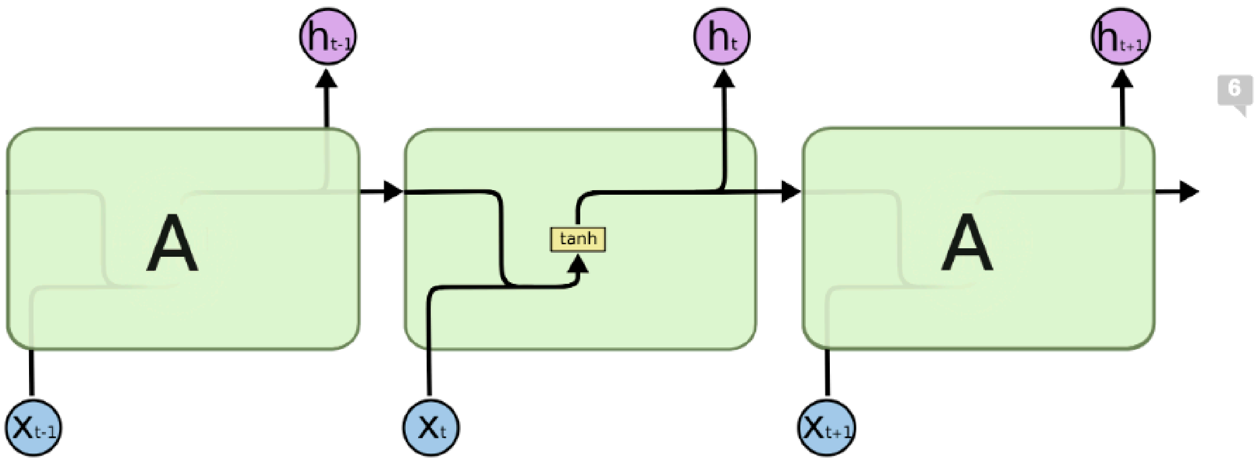
Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

Basic structure of RNN:



An unrolled recurrent neural network.

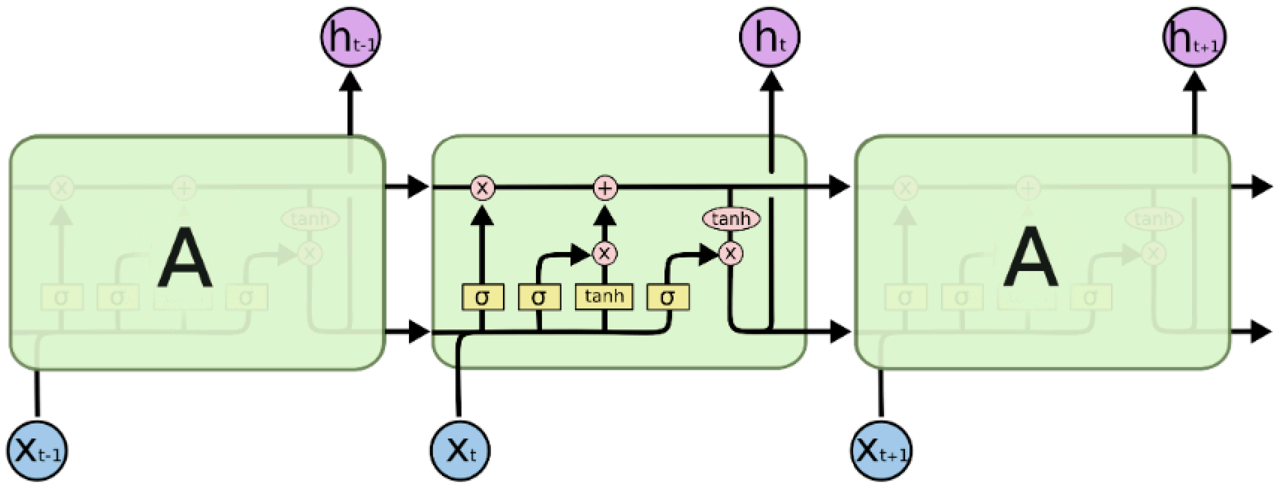
Now in LSTM we have cell state, which is nothing but an accumulation of mathematical functions to help store and pass the relevant information. Think of it as a kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It’s very easy for information to just flow along it unchanged. But before studying the structure of LSTM lets visualize the structure of RNN in the cell representation so that it could create less confusion as we look through the changes.



The repeating module in a standard RNN contains a single layer.

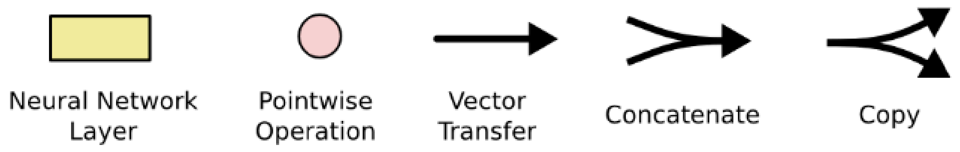
Now we would see the structure of LSTM

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



The repeating module in an LSTM contains four interacting layers.

Don't worry about the details of what's going on. We'll walk through the LSTM diagram step by step later. For now, let's just try to get comfortable with the notation we'll be using.

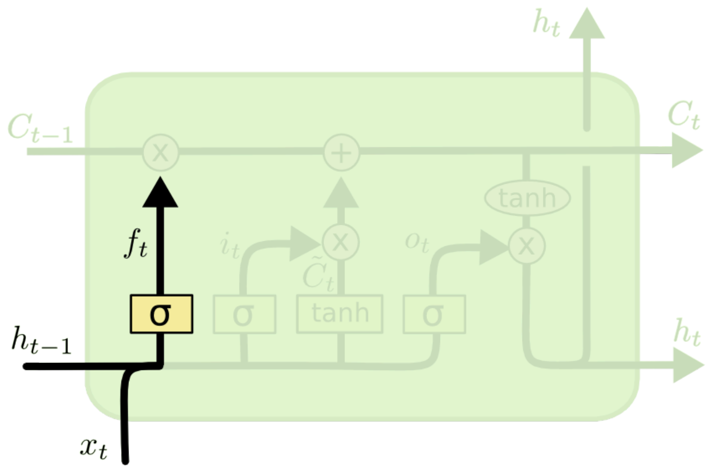


We can see the difference where there is only one line of information passing, I like calling it the main line in RNN compared to the extra memory lane in case of the LSTM network. The memory lane is responsible to carry on important and relevant information further so it could be used even at a distance. Although we would be updating this memory lane based on the input, its like we would want the model to carry the most significant info and delete the previously stored info that is useless now. we would study these all in very detail now.

STEP BY STEP EXPLANATION OF WORKING OF LSTM IN A CHRONOLOGICAL ORDER

The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at ht−1 and xt , and outputs a number between 0 and 1 for each number in the cell state Ct−1 . A 1 represents "completely keep this" while a 0 represents "completely get rid of this."

Let's go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.

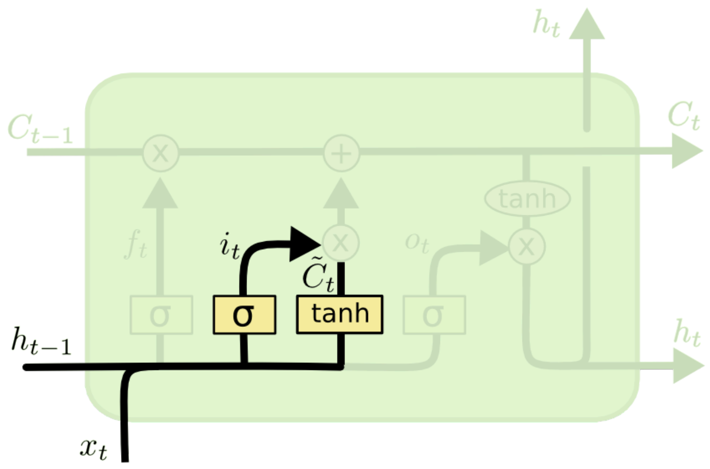


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

Here we can consider this as a complete neural network layer inputting values ht-1 (output from the same layer at time t-1) and the new input xt and determines whether the information from the previous state's output should be pushed to the memory lane and if yes then at what extent (that is determined by a sigmoid function ranging from 0 to 1)

The second step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, C̃ t , that could be added to the state. In the next step, we'll combine these two to create an update to the state.

In the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting.



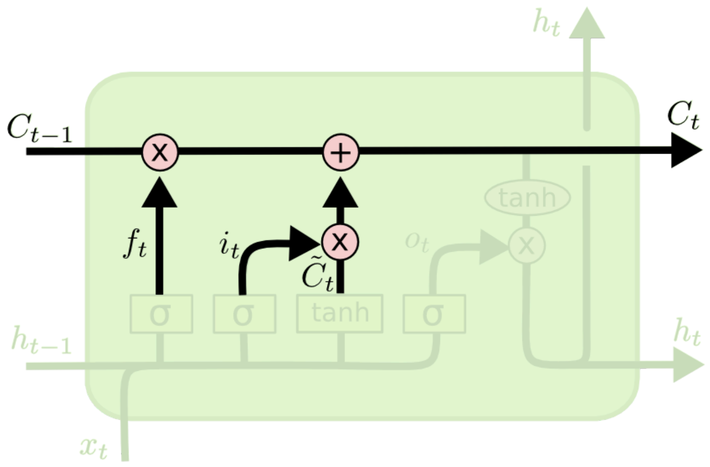
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Multiple questions flooded my mind while learning this concept like why do we need both tanh and sigmoid to add information and why not just sigmoid or tan distinctively add or push the information to the memory lane, the answer is that tanh like provides the complete information that you need to add and the sigmoid determines its weight like to what extent the complete information should be added.

STEP 3 It's now time to update the old cell state, Ct−1 , into the new cell state Ct . The previous steps already decided what to do, we just need to actually do it.

We multiply the old state by ft , forgetting the things we decided to forget earlier. Then we add i∗C̃ t . This is the new candidate values, scaled by how much we decided to update each state value.

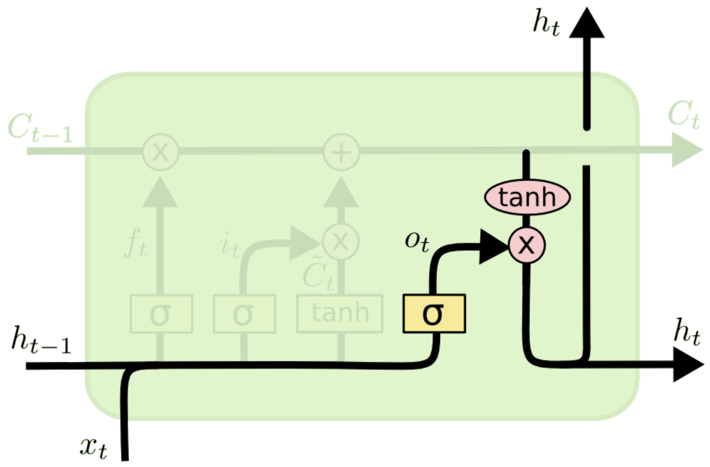
In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.



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

STEP 4: Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.



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

I would explain this step in more easy terms, the input going from ht-1 and xt is nearly same as RNN, where we need to process the input getting suitable output for passing it to the next state. But here the memory lanes also contributes where the information from Ct-1 (the non highlighted line in the diagram), passes through the activation function that is tanh later adding the information information that's flowing in to memory lane to the output giving better context of the previous informations (The output has better context of the information that was given to the model much before unlike RNN as information did flowed smoothly in the memory lane without much amputation).

After all of these steps fr getting the final output, backpropogation starts to update the weights. Provided below are the values that are updated.

1. **Hidden State**: Update based on gradients from the output gate and cell state.
2. **Gates**: Update weights and biases for forget, input, and output gates based on gradients calculated from their respective contributions to the loss.
3. **Candidate Cell State**: Update weights and biases based on gradients derived from its contribution to the cell state.
4. **Cell State**: Update based on gradients derived from the forget gate, input gate, and candidate cell state.

By handling these updates, LSTMs effectively manage information flow and learn to preserve long-term dependencies, making them more capable than standard RNNs for tasks involving sequences.

Reference:https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (https://colah.github.io/posts/2015-08-Understanding-LSTMs/)