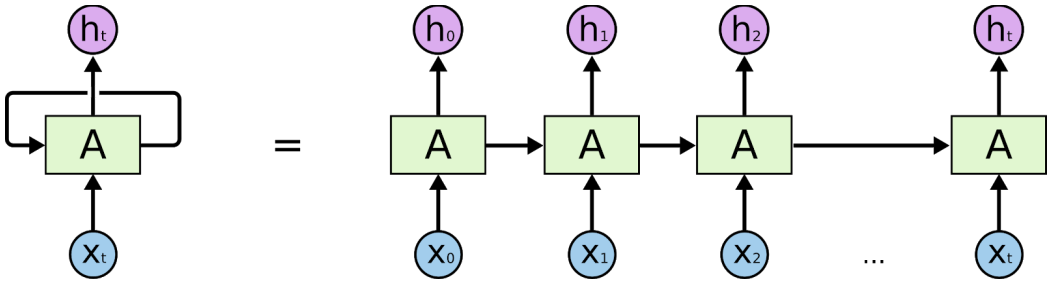
Long-Short Term Memory Networks

Theoretical Knowledge

Recurrent Neural Networks

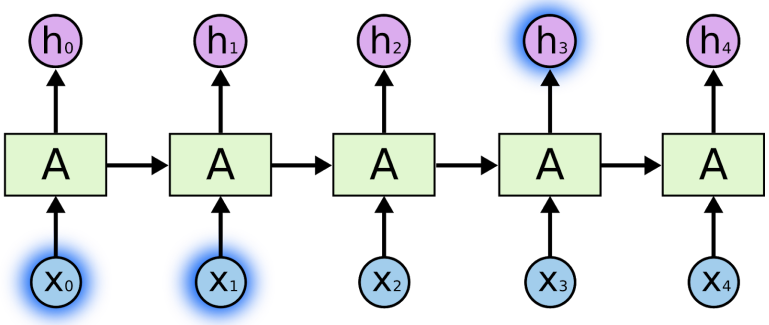
- Use loops in order to allow information to persist.



- In the image above: A looks at input x_t, and outputs h_t. Loop A allows the information to be passed from one time step to the next. The unrolled version of the network in the right shall clarify how the information is propagated forward through time.
- There are lots of applications of RNNs and obvious advantages brought by the ability to use previous information for upcoming decisions.
- But there is more to the story: a few drawbacks in using a Vanilla RNN.

The problem of Long-Term Dependencies

When the gap between relevant information and the place where it is needed is small, RNNs can learn to use the past information. E.g. *"the clouds are in the sky"*.

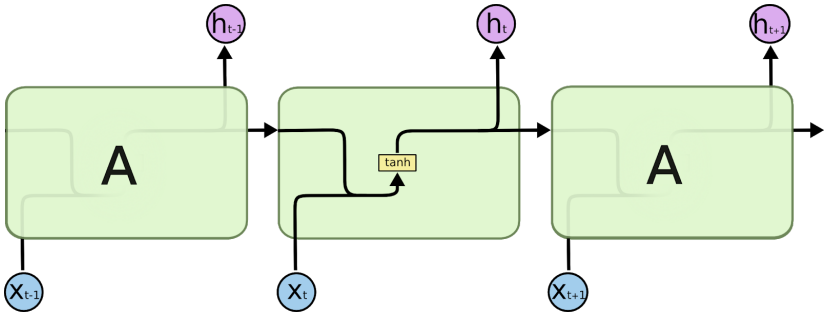


As the gap grows, RNNs become unable to connect the information.

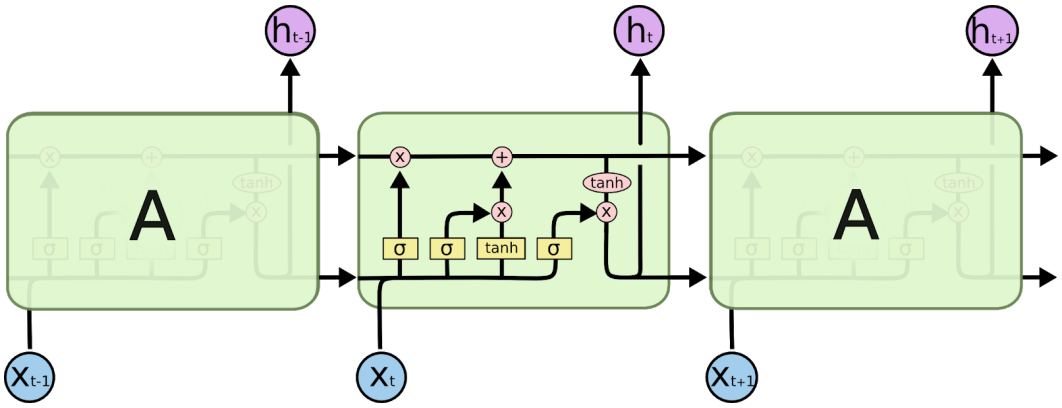
Despite that, in theory, RNNs are perfectly capable of handling such long-term dependencies, Bengio et al. share a few reasons why this might be more than just "piece of cake".

LSTM Networks

- Special type of RNN, capable of learning long-term dependencies.
- The repeating module in a standard RNN has a single layer:

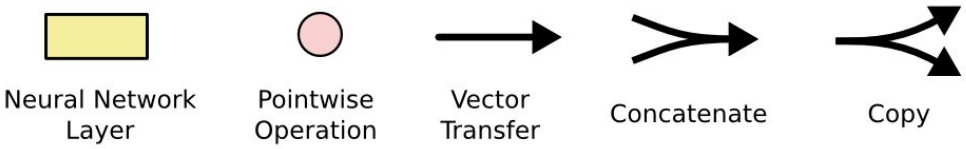


- LSTMs are similar (apparently), but the repeating module has a different structure:

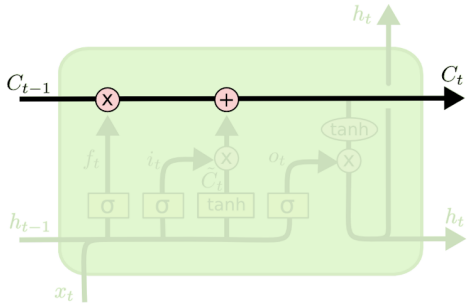


Instead of a single neural layer, there are **4 layers** (sigmoid, sigmoid, tanh, sigmoid).

Notations:



Core idea behind LSTMs: the cell state: the line running straight down the entire chain, with only some minor linear interactions. Information flows along it (almost) unchanged.

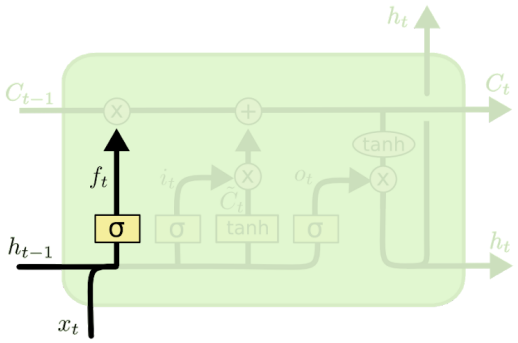


The LSTM can add information to the cell state through gates which are composed of a **sigmoid neural layer**(outputs 0 or 1 - how much of each component should be let through) and a **pointwise operation**.

In a LSTM there are 3 such gates to protect and control the cell state.

Step-by-Step LSTM Walk Through

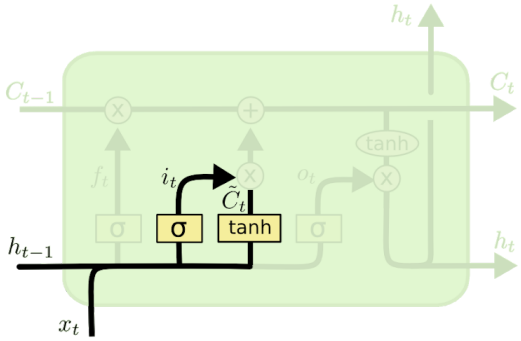
1. **Forget gate layer**: decide what information is thrown away from the cell state. The sigmoid layer does this by looking at h_{t-1} and x_t. A number (0 or 1) is outputted for each number in the cell state C_{t-1}.



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

E.g. forget the gender of a subject previously mentioned, as we want to use the correct pronoun for the gender of the current subject.

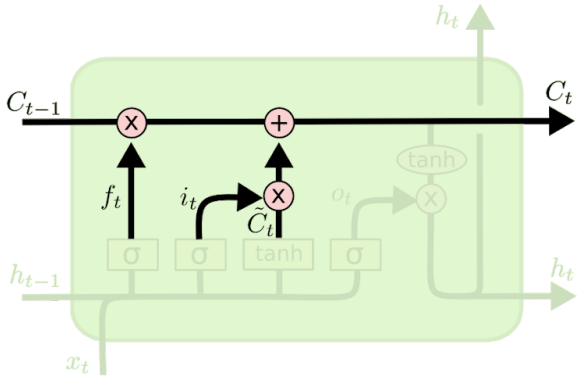
2. **Input gate layer**: decides what values are updated in the cell state.
3. **Candidate values**: tanh layer deciding what values could be added to the state. The next step consists of these two in order to create an ***update to the state***.



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

E.g. add the new gender to the cell state and replace the one that we are forgetting.

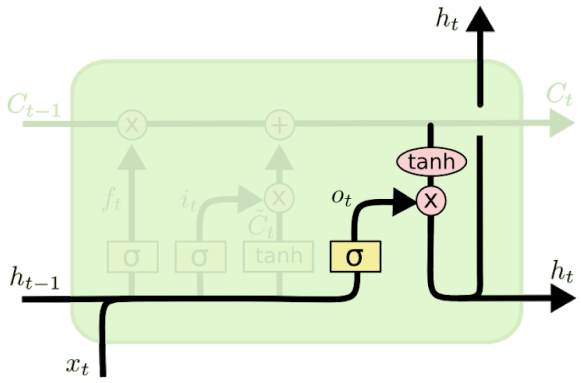
4. Update the old cell state, C_{t-1}, into the new state C_t. We need to just do what we have decided in the previous step:



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

5. Decide what is outputted: a filtered version of the cell state. First, run a sigmoid layer which decides what parts of the cell state we're going to output. Then, put the cell state

through tanh (to push the values to be between −1 and) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.



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

Conclusions
LSTMs represent a big step in what we can accomplish with RNNs. And one step even further is attention (you can look this up and read more about it online).

Implementation & Practice

The suggestion for today's practice is for you to implement a LSTM neural network that learns sequences of characters from one of the books in the Project Gutenberg.
One example of this kind of book is Heart of Darkness by Joseph Conrad.

Follow the steps below for a basic, easy to implement LSTM network:

1. **Download the data file**. Use the link above for *Heart of Darkness*, or try another book from the Project Gutenberg library.
2. **Load data and create character to integer mappings.**
    a. Open the text file, read the data and then convert it to lowercase letters. Note: remove the header and footer from the
    b. Map each character to a respective number. Keep two dictionaries in order to have access more easily to the mappings both ways around (character to index, index to character).
3. **Prepare the data.**
    a. Think in sequences of 100 characters: 99 characters in the input (X), 1 in the output (Y). E.g.: for the sequence *['h', 'e', 'l', 'l']* as input, we will have *['o']* as the expected output.
    b. Reshape X such that it has the shape expected by a LSTM: [samples, time steps, features].
        i. samples: number of data points (len(X));

    ii. time steps: number of time-dependent steps that are in a single data point (100);
    iii. features: number of variables for the true value in Y (1).
  c. Scale the value of X to be in [0, 1].
  d. One-hot encode the true values in Y.
4. **Define the LSTM model**
  a. Instantiate the model: a linear stack of layers.
  b. First layer: LSTM with 256 memory units, and specify the input shape as well X.shape[1], X.shape[2].
  c. Add dropout 20%.
  d. The last layer is going to be Dense, with softmax and Y.shape[1] classes.
  e. Compile the model.
5. **Train the model** - 5 epochs and a batch size of 128 should be enough in order to see something generated. However, the text generated in the end is not going to make a lot of sense. You are going to need to let the network train much more epochs (at least 20 in this case) in order for the generated text to make some sense. This is out of the scope of this laboratory as it takes too much time.
6. **Generate characters.** Fix a random seed and start generating characters. The prediction of the model gives out the character encoding of the predicted character. This is then decoded back to the character value and appended to the pattern.

Starting from this suggestion, you can add layers, use different values for the hyperparameters - which essentially means to change the architecture in order to obtain better generated text in the end.

Bonus

Another approach that you can take is using words (their embeddings, to be more specific) as features. We can use the same dataset as above, in order to compare the performance of this model against that of the previous. Do not forget to add an embedding layer in this case.

Bibliography

https://colah.github.io/posts/2015-08-Understanding-LSTMs/
https://adventuresinmachinelearning.com/keras-lstm-tutorial/
http://karpathy.github.io/2015/05/21/rnn-effectiveness/
https://skymind.ai/wiki/lstm
http://karpathy.github.io/2015/05/21/rnn-effectiveness/