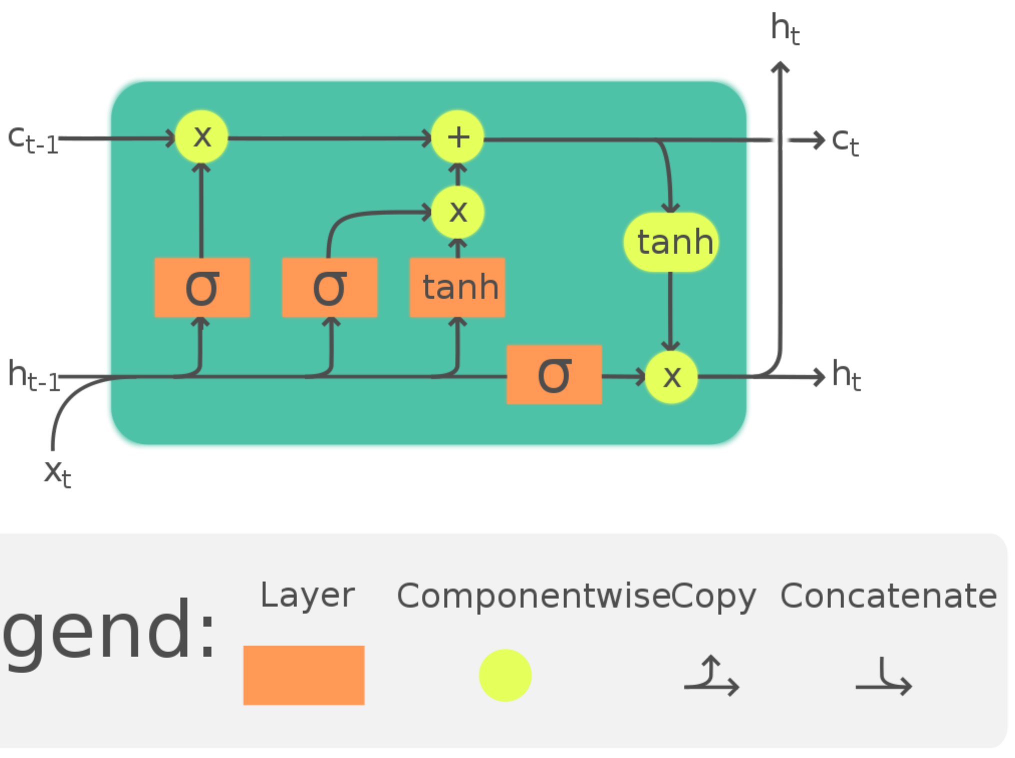
LSTM

Author: Jakub Dylag
www.linkedin.com/in/jakubdylag

Want to take your Time Series forecasting to the next level? Long-Short Term Memory (LSTM) Networks will fix the Vanishing Gradient problem you have encountered with Simple RNNs (Elman Networks). They are used in many applications, such as predicting stock prices, translating languages, and even generating new text or music! By the end of this article you will have a good understanding of how LSTMs work and be confident to apply them to your own projects.

LSTM are complex systems built up from a standardised LSTM cell shown below. It consists of several components, including an input gate, an output gate, and a forget gate, each of which is controlled by a sigmoid activation function. These gates determine the flow of information into and out of the cell.



Source: https://en.wikipedia.org/wiki/Long_short-term_memory

Instead of using a single feedback loop like many basic Recurrent Neural Networks do, LSTM use two separate loops in the shape of Short-term memory and Long-term memory, as the name suggests. As with other RNNs Hidden States (h) act like the Short-term memory and store recent events and are constantly altered by multiple gates. A new Cell State (c) is added to LSTMs to remember long-term dependencies. This cell state is only minorly changed throughout the forward pass. Both the Hidden States and Cell States are represented as fixed size vectors. Larger vectors allow the model to remember more information, however raise the complexity of the model, risking its overfitting.

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

Source: https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html

A more detailed formulaic representation of an LSTM cell can be seen above. Importantly, the forget gate (f_t) chooses which information is no longer relevant in the cell state. The input gate (i_t) and output gate (o_t) transform the previous hidden state (h_{t-1}), computed from the previous input. The output gate decides what parts of the old hidden state join onto the new hidden state (h_t). Whereas the input gate decides which parts are joined to the new cell state (c_t) . All these gates use a hidden neural network layer using the sigmoid activation function, which is commonly used in classification models.

Architecture Types

LSTMs are a flexible architecture that has been adapted into a range of varied configurations!

Bidirectional LSTM (BiLSTM) are similar to a vanilla LSTM, however the sequence is processed in both the forward and backwards direction, which requires the entire sequence to be available at once. This allows the network to have a more detailed view from different perspectives.

Convolutional LSTMs (ConvLSTM) incorporate a one dimensional convolutional layer previous to the initial LSTM layer, in order to allow for deeper understanding. These have been exceedingly successful in multivariate time series prediction, where multiple variables are given to the model in parallel. For example the electricity price and the outside temperature.

Gated Recurrent Unit (GRU) with the addition of peephole connections, simplify the LSTM architecture by combining the input and output gates into a single update gate. This reduces training cost, whilst retaining the hidden and cell states. However LSTM are still commonly preferred in most task without specific power constraints.

A challenging problem with LSTM architecture most prevalent with stacked LSTMs, which have multiple layers, is overfitting. A solution to overcome this is the use of **Drop Out layers** in between LSTM layers for better generalisation. This randomly discards some elements in the hidden states, altering the following layers perspective and promoting generalisation.

Build your own!

For beginners we highly recommend using the Keras LSTM module (https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM) and for more experienced programmers the PyTorch implementation (https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html), which allows for deeper analysis and customisation.

As discussed in our RNN workshop (http://aisoc.uk/workshops/neural-networks) it is important to choose the correct parameters to get the best performance out of your model.

- **Input size:** The larger the input the more "context" the model can see before making a decision.
- **Output size:** Smaller output sizes are best as the model doesn't have to predict far in the future, which would increase difficulty.
- **Hidden and Cell size:** The size of the hidden state must be the same as the cell state. More complex problems require a larger size.
- **Number of Layers:** The more layers the more abstractions can be made by the model. Should be minimised as training time is significantly lengthened with additional layers.

Being able to explain your models performance is just as vital as creating it. A method introduced by Wang et al (https://arxiv.org/pdf/1512.08849.pdf) analysed the values forget gate when performing a forward pass. Remember that, forget gates indicate the importance of the previous cell state, and therefore higher values of the forget gate mean the previous data point was important to remember. This will help you explain some of your models decisions, and which layers focus on what aspects.

Inspecting the Hidden and Cell states is also a good idea, as some elements may remain unused (equal to 0), which allows you to decrease their size and train future models faster. An example of this can be found in my Undergraduate Thesis: (https://github.com/JakubDylag/CGM_Prediction_LSTM/blob/main/code/xai_0.ipynb)