LSTMs Explained: A Complete, Technically Accurate, Conceptual Guide with Keras

September 2, 2020

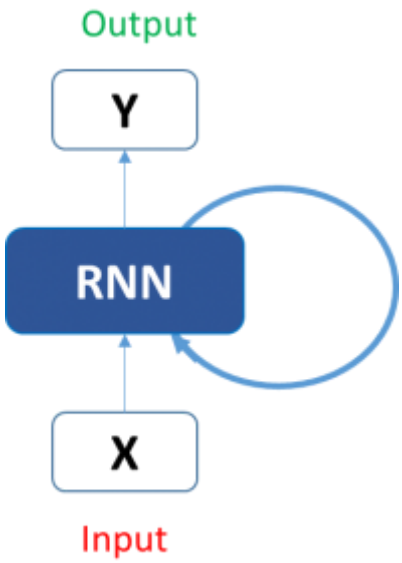
I know, I know — yet another guide on **LSTMs** / **RNNs** / Keras / whatever. There are SO many guides out there — half of them full of false information, with inconsistent terminology — that I felt frustrated enough to write this one-stop guide + resource directory, even for future reference. This guide aims to be a glossary of technical terms and concepts consistent with Keras and the Deep Learning literature. This article assumes a very basic conceptual familiarity with the concept of Neural Networks in general. If you spot something that's inconsistent with your understanding, please feel free to drop a comment / correct me!

Content Page

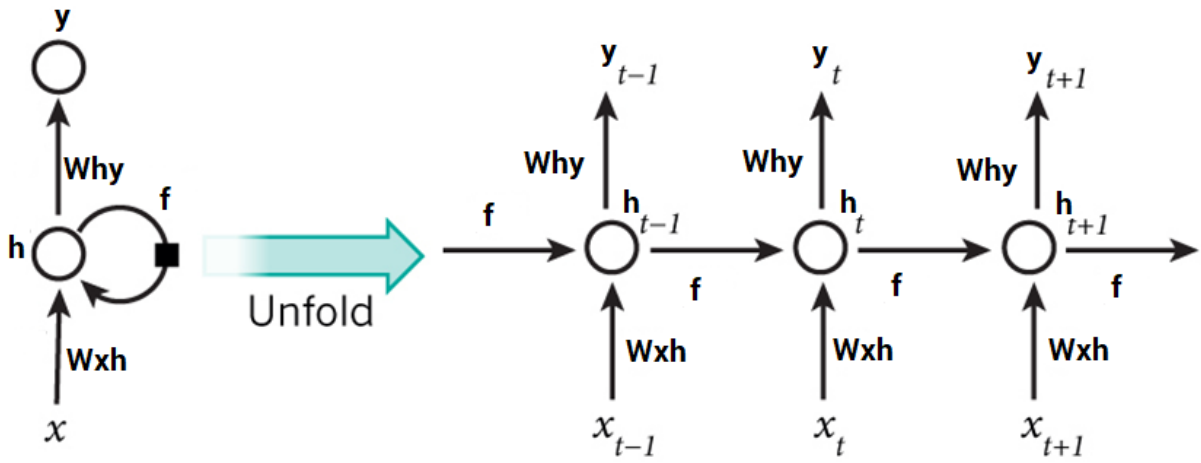
1. RNNs and LSTMs
2. Hidden State vs Cell State
3. General Gate Mechanisms
4. Gate Operation Dimensions & "Hidden Size"
5. "Hidden Layers"
6. Model Complexity
7. Quirks with Keras — Return Sequences? Return States?

Long-Short-Term Memory Networks and RNNs — How do they work?

First off, LSTMs are a special kind of RNN (Recurrent Neural Network). In fact, LSTMs are one of the about 2 kinds (at present) of practical, usable RNNs — LSTMs and Gated Recurrent Units (GRUs). What's a "regular" RNN, then, you might ask? Well, I don't suppose there's a "regular" RNN; rather, RNNs are a broad concept referring to networks that are full of cells that look like this:



A single RNN cell. Image Credits to: https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/



The same single RNN cell, “unfolded” or “unrolled”. Image Credits to: https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/

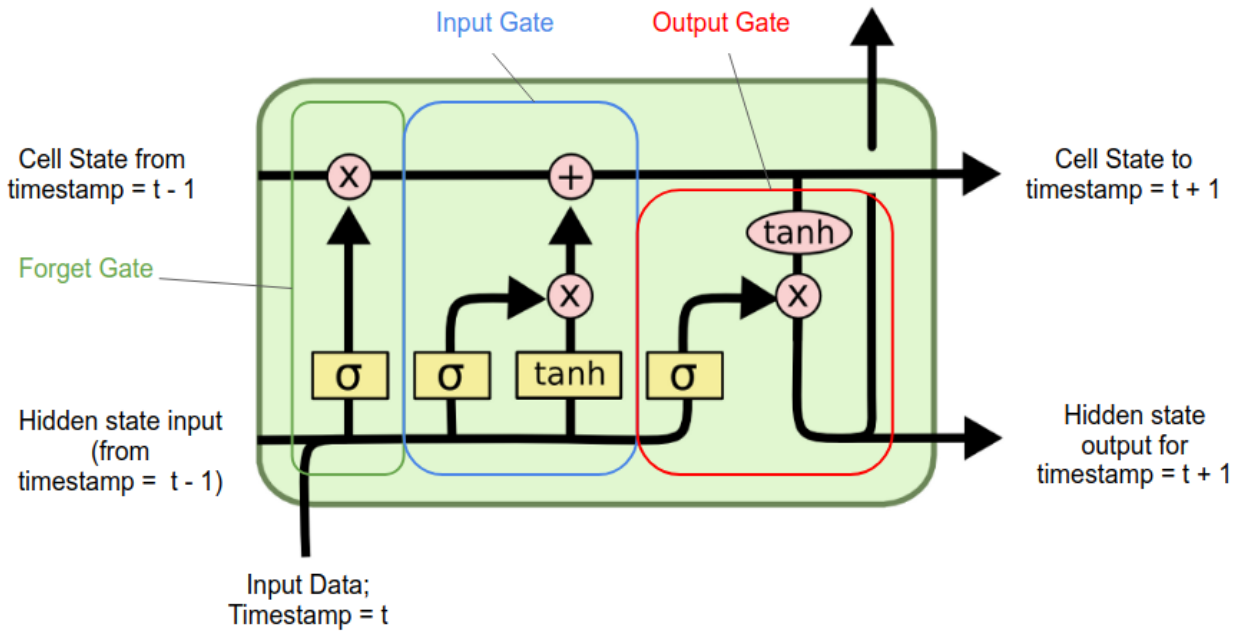
X: Input data at current time-step
Y: Output
Wxh: Weights for transforming X to RNN hidden state (not prediction)
Why: Weights for transforming RNN hidden state to prediction
H: Hidden State
Circle: RNN Cell

A single-cell RNN like the above is very much possible. In that case, assume that we’re trying to process very simple time-series data. Each series contains 3 time-steps worth of data. The RNN cell would:

1. process the first time-step (t = 1), then channel its output(s), as well as the next time-step (t = 2), to itself
2. process those with the same weights as before, and then channel its output(s), as well as the last time-step (t = 3), to itself again
3. process those with the same weights as before, and then output the result to be used (either for training or prediction). More details on the format of this output later on.

That is the big, really high-level picture of what RNNs are. As said before, an RNN cell is merely a concept. In reality, the RNN cell is almost always either an LSTM cell, or a GRU cell. In this article, we're going to focus on LSTMs.

In order to understand why LSTMs work, and get an intuitive understanding of the statistical complexity behind the model that allows it to fit to a variety of data samples, I strongly believe that it's necessary to understand the mathematical operations that go on behind the cell, so here we go!



A single LSTM Cell

Great, big complex diagram. This entire rectangle is called an LSTM "cell". It is analogous to the circle from the previous RNN diagram. These are the parts that make up the LSTM cell:

1. The "Cell State"
2. The "Hidden State"
3. The Gates: "Forget" or also known as "Remember", "Input", and "Output"

"Cell State" vs "Hidden State"

There is usually a lot of confusion between the “Cell State” and the “Hidden State”. The two are clearly different in their function. The cell state is meant to encode a kind of aggregation of data from all previous time-steps that have been processed, while the hidden state is meant to encode a kind of characterization of the previous time-step’s data.

Hidden State — Conceptual Interpretation

The *characterization* (not an official term in literature) of a time-step’s data can mean different things. Let’s pretend we are working with Natural Language Processing and are processing the phrase “the sky is blue, therefore the baby elephant is crying”, for example. If we want the LSTM network to be able to classify the sentiment of a word in the context of the sentence, the hidden state at t = 3 would be an encoded version of “is”, which we would then further process (by a mechanism outside the LSTM network) to obtain the predicted sentiment. If we want the LSTM network to be able to predict the next word based on the current series of words, the hidden state at t = 3 would be an encoded version of the prediction for the next word (ideally, “blue” [edited]), which we would again process outside of the LSTM to get the predicted word. As seen, *characterization* takes on different meanings based on what you want the LSTM network to do. In terms of the mathematics behind it, **it can indeed be this flexible** because ultimately, what we want the LSTM to do dictates how we train it and what kind of data we use; the weights will tune themselves accordingly to best approximate the answer that we seek. *Characterization*is an abstract term that merely serves to illustrate how the **hidden state is more concerned with the most recent time-step**.

It is important to note that the **hidden state does not equal the output or prediction,** it is merely an encoding of the most recent time-step. That said, the hidden state, at any point, can be processed to obtain more meaningful data.

Cell State — Conceptual Interpretation

The cell state, however, is more concerned with the entire data so far. If you’re right now processing the word “elephant”, the cell state contains information of **all** words right from the start of the phrase. As you can see in the diagram, each time a time-step of data passes through an LSTM cell, a copy of the time-step data is filtered through a forget gate, and another copy through the input gate; the result of both gates are incorporated into the cell state from processing the previous time-step and gets passed on to get modified by the next time-step yet again. The weights in the forget gate and input gate figure out how to extract features from such information so as to determine which time-steps are important (high forget weights), which are not (low forget weights), and how to encode information from the current time-step into the cell state (input weights). As a result, not all time-steps are incorporated equally into the cell state — some are more significant, or worth remembering, than others. This is what gives LSTMs their characteristic ability of being able to dynamically decide how far back into history to look when working with time-series data.

To summarize, the cell state is basically the *global* or *aggregate* memory of the LSTM network over all time-steps.

General Gate Mechanism / Equation

Before we jump into the specific gates and all the math behind them, I need to point out that there are two types of normalizing equations that are being used in the LSTM. The first is the **sigmoid function** (represented with a lower-case sigma), and the second is the **tanh function**. This is a deliberate choice that has a very intuitive explanation.

Whenever you see a **sigmoid** function in a mechanism, it means that the mechanism is trying to calculate a set of **scalars** by which to multiply (amplify / diminish) something else (apart from preventing vanishing / exploding gradients, of course).

Whenever you see a **tanh** function, it means that the mechanism is trying to transform the data into a **normalized encoding of the data**.

Each of the “Forget”, “Input”, and “Output” gates follow this general format:

Equation for “Forget” Gate

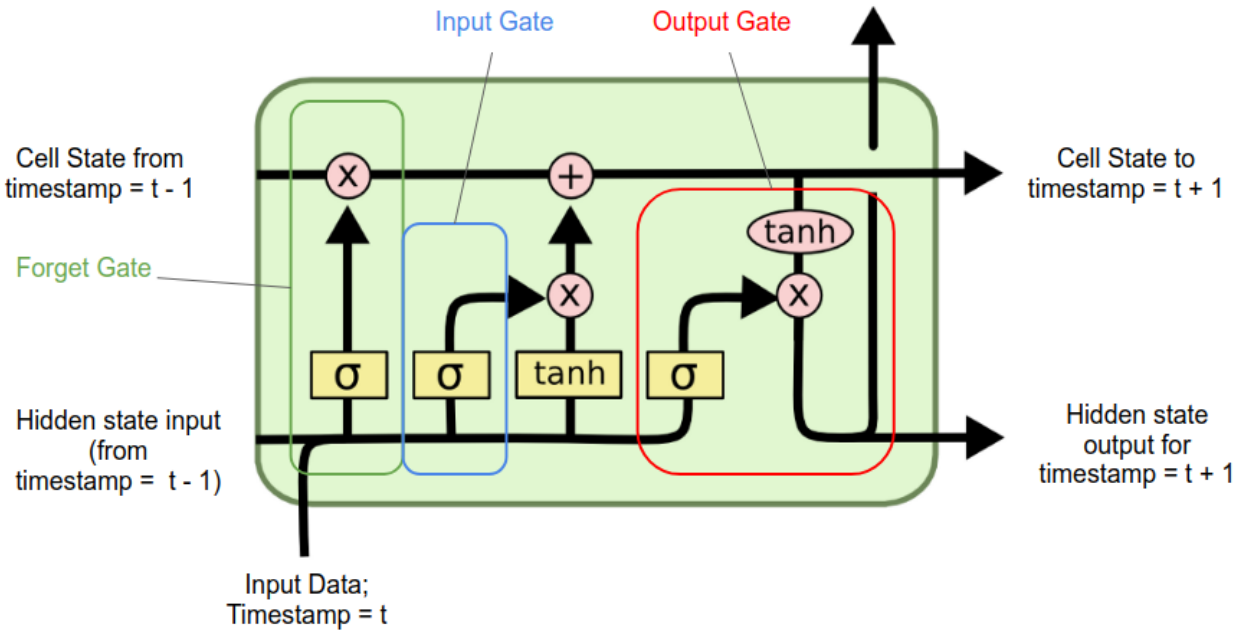
In English, the inputs of these equations are:

1. h_(t-1): A copy of the hidden state from the previous time-step
2. x_t: A copy of the data input at the current time-step

These equation inputs are separately multiplied by their respective matrices of weights at this particular gate, and then added together. The result is then added to a bias, and a sigmoid function is applied to them to squash the result to between 0 and 1. Because the result is between 0 and 1, it is perfect for acting as a scalar by which to amplify or diminish something. You would notice that all these sigmoid gates are followed by a point-wise multiplication operation. This is the amplification / diminishing in operation. For example, at the forget gate, if the forget gate outputs a matrix of values that are all very close to 1, it means that the forget gate has concluded that based on the current input, the time-series’ history is very important, and therefore, when the cell state from the previous time-step is multiplied by the forget gate’s output, the cell state continues to retain most of its original value, or “remember its past”. If the forget gate outputs a matrix of values that are close to 0, the cell state’s values are scaled down to a set of tiny numbers, meaning that the forget gate has told the network to forget most of its past up until this point.

If you have trouble visualizing these operations, it may be worthwhile skipping to the section entitled **Gate Operation Dimensions and “Hidden Size” (Number of “Units”)** where I draw out these matrices in action.

Input Gate



LSTM Cell with differently-drawn input gate

So the above illustration is slightly different from the one at the start of this article; the difference is that in the previous illustration, I boxed up the entire mid-section as the "Input Gate". The terminology here is varied. To be extremely technically precise, the "Input Gate" refers to only the sigmoid gate in the middle. The mechanism is exactly the same as the "Forget Gate", but with an entirely separate set of weights.

What's the **tanh** gate in the middle, then? Sometimes lumped together with the **sigmoid** gate as the "Input Gate", this tanh gate is also called the "Candidate Gate", or in some sources, the "Candidate *Layer*", which I think is a horrible term because it's firstly, not a layer in the sense of what we expect a layer to be in a neural network, and hence secondly, super confusing. Regardless, this is the first time we're seeing a tanh gate, so let's see what it does!

We know that a copy of the current time-step and a copy of the previous hidden state got sent to the sigmoid gate to compute some sort of scalar matrix (an amplifier / diminisher of sorts). **Another copy of both pieces of data are now being sent to the tanh gate to get normalized to between -1 and 1**, instead of between 0 and 1. The matrix operations that are done in this tanh gate are exactly the same as in the sigmoid gates, just that instead of passing the result through the sigmoid function, we pass it through the tanh function.

The output of this tanh gate is then sent to do a point-wise or element-wise multiplication with the sigmoid output. You can think of the tanh output to be an encoded, normalized version of the hidden state combined with the current time-step. We call this data "encoded" because while passing through the tanh gate, the hidden state and the current time-step have already been multiplied by a set of weights, which is the same as being

put through a single-layer densely-connected neural network. In other words, there is already some level of feature-extraction being done on this data while passing through the tanh gate. The same goes for the sigmoid gate.

From this perspective, the sigmoid output — the amplifier / diminisher — is meant to scale the encoded data based on what the data looks like, before being added to the cell state. The rationale is that the presence of certain features can deem the current state to be important to remember, or unimportant to remember.

To summarize what the input gate does, it does feature-extraction once to encode the data that is meaningful to the LSTM for its purposes, and another time to determine how remember-worthy this hidden state and current time-step data are. The feature-extracted matrix is then scaled by its remember-worthiness before getting added to the cell state, which again, is effectively the global "memory" of the LSTM.

Output Gate

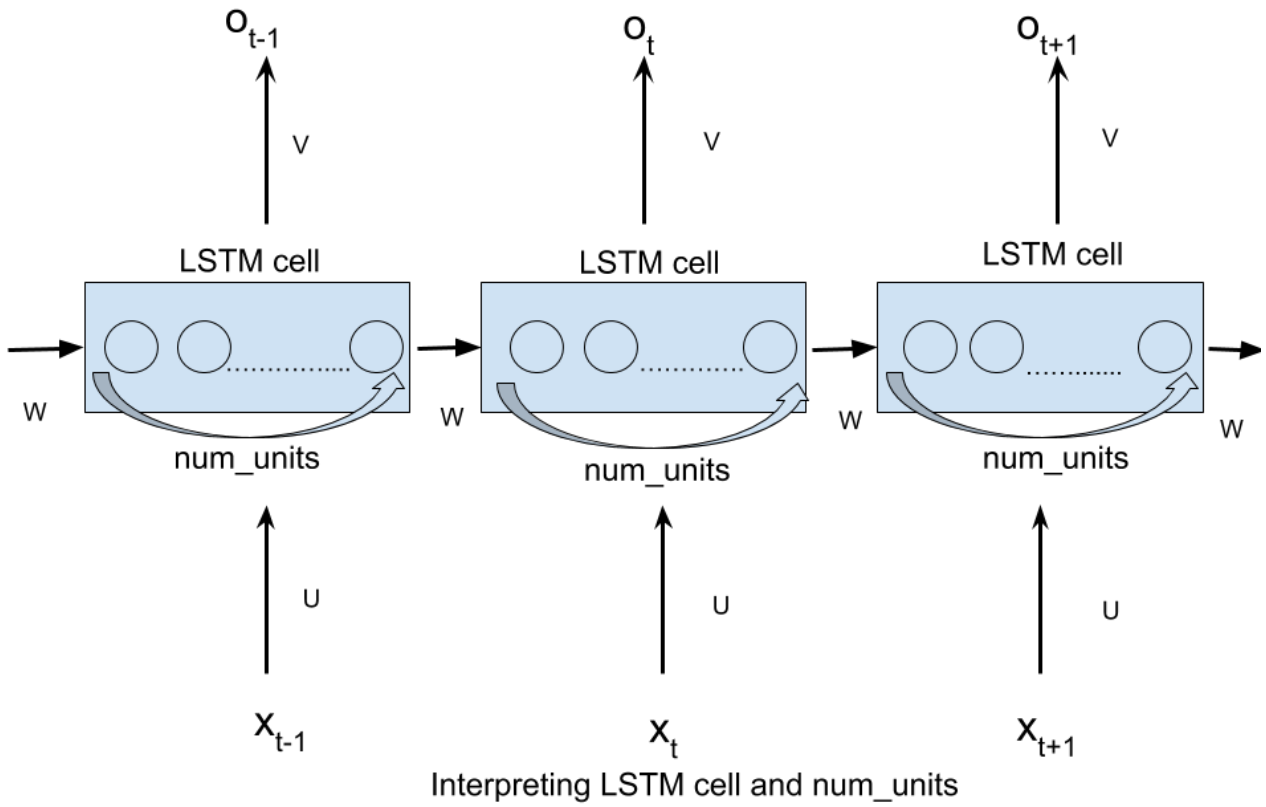
The output gate uses pretty much the same concepts of encoding and scaling to:

1. Get incorporated into the cell state
2. Form an output hidden state that can be used to either make a prediction or be fed back into the LSTM cell for the next time-step.

The conceptual idea behind the operation here is that, since the cell state now holds the information from history up to and including this time-step,

Gate Operation Dimensions and "Hidden Size" (Number of "Units")

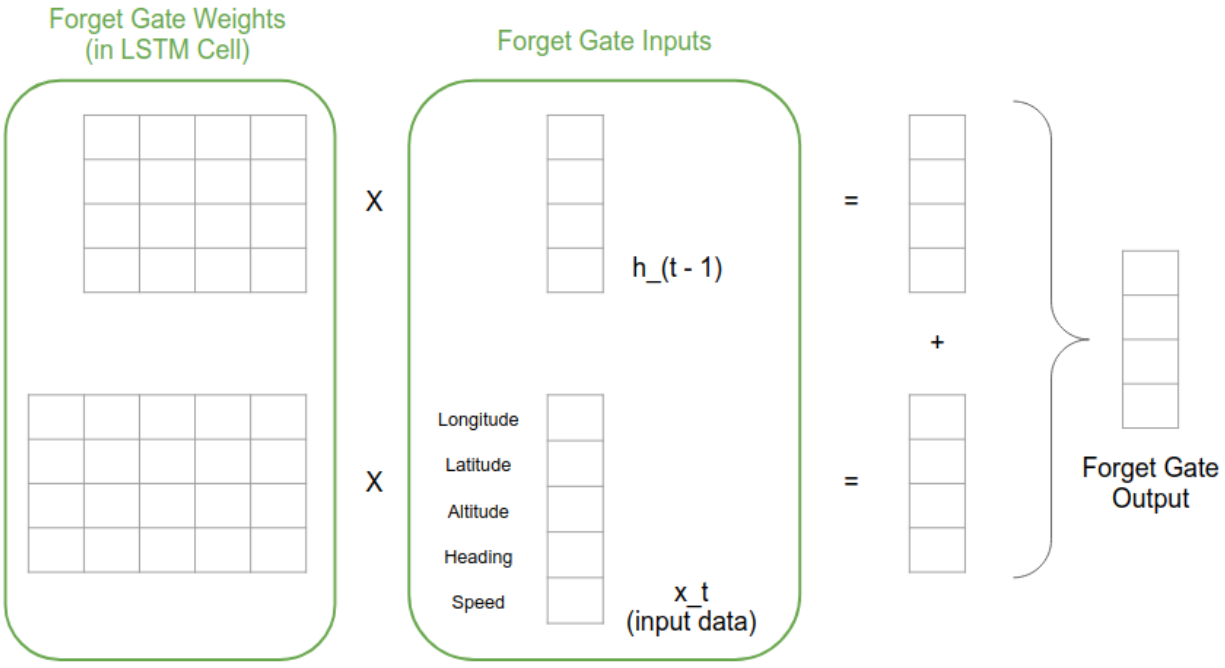
I've been talking about matrices involved in multiplicative operations of gates, and that may be a little unwieldy to deal with. What are the dimensions of these matrices, and how do we decide them? This is where I'll start introducing another parameter in the LSTM cell, called "hidden size", which some people call "num_units". If you're familiar with other types of neural networks like Dense Neural Networks (DNNs), or Convolutional Neural Networks (CNNs), this concept of "hidden size" is analogous to the number of "neurons" (aka "perceptrons") in a given layer of the network.



A graphic illustrating hidden units within LSTM cells

Although the above diagram is a fairly common depiction of hidden units within LSTM cells, I believe that it's far more intuitive to see the matrix operations directly and understand what these units are in conceptual terms.

To do so, let's now switch things up a little and pretend that we're working with time-series data from aircraft, where each data sample is a series of pings from aircraft, each containing the aircraft's longitude, latitude, altitude, heading, and speed (input 5 variables) over time. Further pretend that we have a hidden size of 4 (4 hidden units inside an LSTM cell). This is how the operations in the "Forget" (and "Input" and "Output" too) Gate would look:

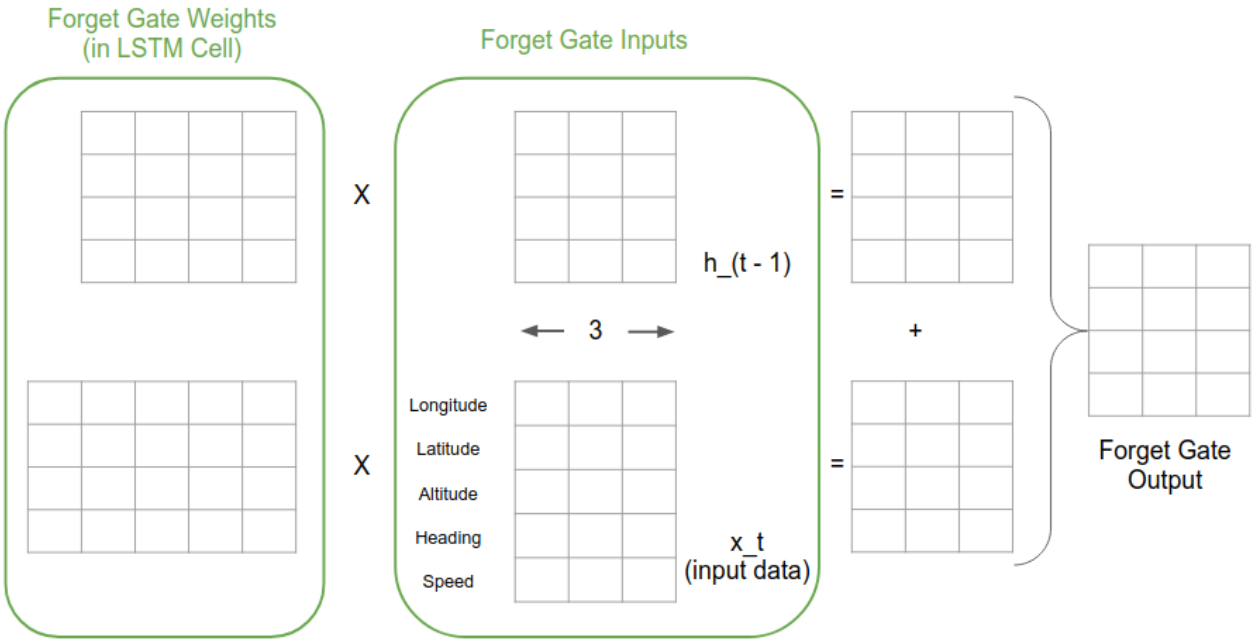


Forget Gate Operations and Dimensions

A few things to note about the **dimensions of the weight matrices:**

1. w_h (Weight Matrix for Hidden State) = `hidden_size * hidden_size`
2. w_x (Weight Matrix for Input Data) = `hidden_size * input_variables`
3. Output = `1 * hidden_size`
4. Output is further passed through the sigmoid function

Notice that the dimensions of the weight matrices are entirely determined by the Hidden Size and Number of Input Variables, which makes sense. In reality, we're processing a huge bunch of data with Keras, so you will rarely be running time-series data samples (flight samples) through the LSTM model one at a time. Rather, you'll be processing them in batches, so there's an added parameter of `batch_size` . The gate operation then looks like this:



Forget Gate Operations when Batch Size = 3

A few things to note:

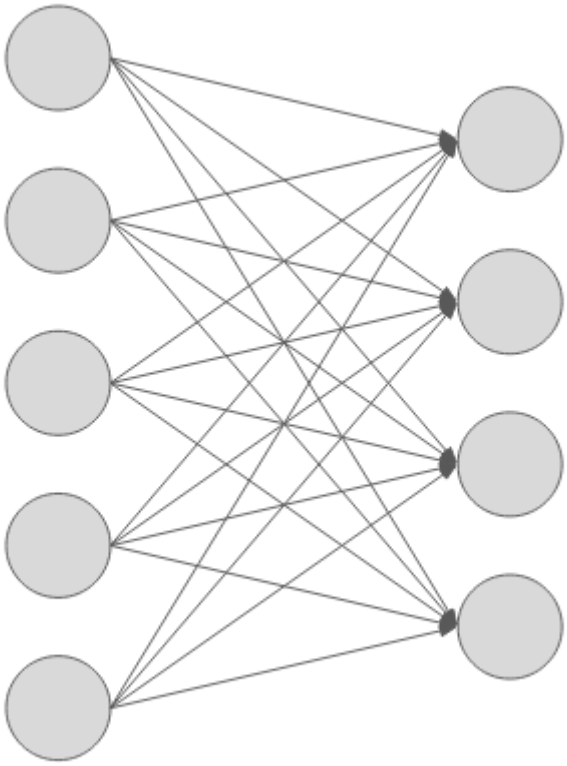
1. Inputs and Outputs, instead of being 1-column vectors, are now 3-column matrices.
2. Dimensions of weights (number of parameters) are of batch size

From Matrix Representation to Neuron / Nodal Representation

A fun thing I love to do to really ensure I understand the nature of the connections between the weights and the data, is to try and visualize these mathematical operations using the symbol of an actual neuron. It nicely ties these mere matrix transformations to its neural origins.

For this forget gate, it's rather straightforward. For the `w_x * x` portion of the forget gate, consider this diagram:

In this familiar diagramatic format, can you figure out what's going on? The left 5 nodes represent the input variables, and the right 4 nodes represent the hidden cells. Each connection (arrow) represents a multiplication operation by a certain weight. Since there are 20 arrows here in total, that means there are 20 weights in total, which is consistent with the 4 x 5 weight matrix we saw in the previous diagram. Pretty much the same thing is going on with the hidden state, just that it's 4 nodes connecting to 4 nodes through 16 connections. Okay, that was just a fun spin-off from what we were doing.



Connections between time-series data inputs and weights at the forget gate

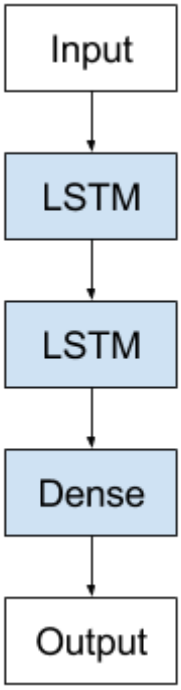
“Hidden Layers” (Number of Layers)

So far, these are the things we’ve covered:

- RNNs and LSTMs
- Gate Functionality
- Gate Operations, Dimensions, and “Hidden Size”

In terms of hyperparameters, there’s only “Hidden Layers” left. A “multi-layer LSTM” is also sometimes called “stacked LSTMs”. It looks like this:

The concept of increasing number of layers in an LSTM network is rather straightforward. All time-steps get put through the first LSTM layer / cell to generate a whole set of hidden states (one per time-step). These hidden states are then used as inputs for the second LSTM layer / cell to generate another set of hidden states, and so on and so forth.



A stacked LSTM. Image Credits: https://machinelearningmastery.com/stacked-long-short-term-memory-networks/

Model Complexity

Estimating what hyperparameters to use to fit the complexity of your data is a main course in any deep learning task. There are several rules of thumb out there that you may search, but I'd like to point out what I believe to be the conceptual rationale for increasing either types of complexity (hidden size and hidden layers).

Generally, when you believe the input variables in your time-series data have a lot of interdependence — and I don't mean linear dependence like "speed", "displacement", and "travel time" — a **bigger hidden size would be necessary** to allow the model to figure out a greater number of ways that the input variables could be talking to each other.

Generally, too, when you believe that the patterns in your time-series data are very high-level, which means to say that it can be abstracted a lot, **a greater model depth, or number of hidden layers, is necessary.** In Speech Recognition, this would be akin to identifying small millisecond-long textures in speech, then further abstracting multiple textures to distinct soundbites, then further abstracting these soundbites to consonants and vowels, then to word segments, then to words.

It also generally follows that the two types of complexities cannot have too large a difference, as they are complementary.

Quirks with Keras

The terminology that I've been using so far are consistent with Keras. I've included technical resources at the end of this article if you've not managed to find all the answers from this article. However, there are some other quirks that I haven't yet explained.

One major quirk that results in a *LOT* of confusion is the presence of the arguments: `return_sequences` and `return_states`.

Remember that in an LSTM, there are 2 data states that are being maintained — the "Cell State" and the "Hidden State". By default, an LSTM cell returns the hidden state for a single time-step (the latest one). However, Keras still records the hidden state outputted by the LSTM at each time-step. Hence,

- `return_sequences` means "return all hidden states" Default: `False`
- `return_states` means "return cell state". Default: `False`

Confusing stuff. For the most part, you won't have to care about `return_states`. But, if you're working with a multi-layer LSTM (Stacked LSTMs), you will have to set `return_sequences = True`, because you need the entire series of hidden states to feed forward into each successive LSTM layer / cell. The exception to this is with the last LSTM layer / cell. In the last layer, you may or may not to set it to `True`, depending on what kind of output you want. If you want an output of same dimensions as your input, entire time-series with the same number of time-step, then it's `True`, but if you're expecting only a representation for the last time-step, then it's `False`. The outputs here are typically put through a Dense layer to transform the hidden state into something more useful, like a class prediction.

This post summarizes this fantastically, with code examples: https://machinelearningmastery.com/return-sequences-and-return-states-for-lstms-in-keras/.

Conclusion

Hopefully, I've helped you to understand the specifics of LSTMs with the correct jargon, much of which are glossed over by most of the application-based guides that are sometimes what seems to be all we can find. This guide was written from my experience working with data scientists and deep learning engineers, and I hope the research behind this guide reflects that.

Resources:

- Conceptual Explanation of Gates + Simple Text Prediction Code Example https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/

- Conceptual Explanation of LSTM Gate Operations https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21
- Gate Equations https://medium.com/@divyanshu132/lstm-and-its-equations-5ee9246d04af

- Conventional Machine Learning Terminology vs Keras / TF Terminology https://stats.stackexchange.com/questions/241985/understanding-lstm-units-vs-cells
- Stacked LSTMs https://machinelearningmastery.com/stacked-long-short-term-memory-networks/
- return_sequences vs return_states https://machinelearningmastery.com/return-sequences-and-return-states-for-lstms-in-keras/