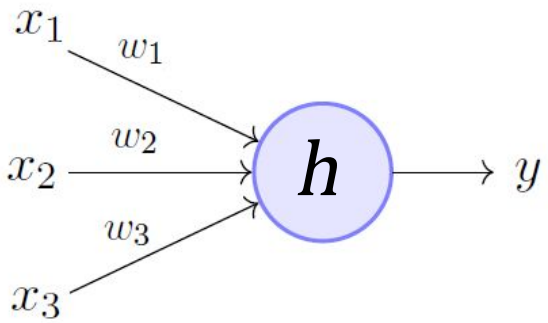
LSTM theory & workflows

First, there was a perceptron

- Simply a weighted sum (plus optional bias term) followed by a nonlinearity
- Originally the nonlinearity was a simple Heaviside step function
- Resulting in a binary classifier
- Aside: Derivative of Heaviside is a Dirac delta function, making this impossible to train via gradient descent
- So we will use the sigmoid as an activation function

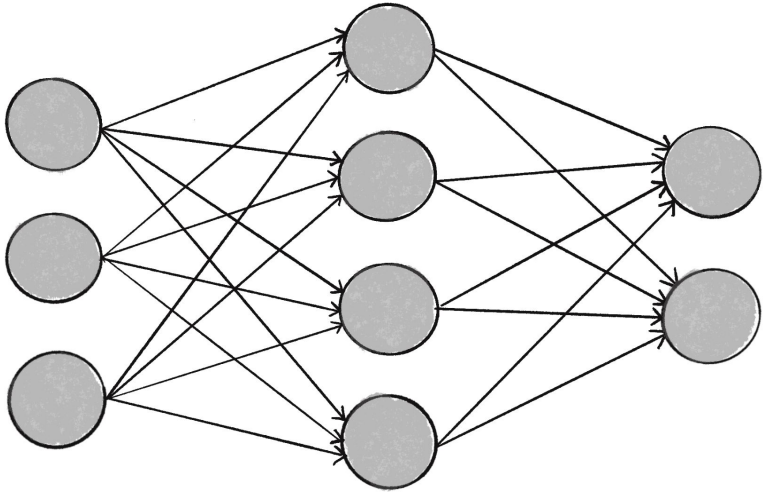


$$f(\mathbf{x}) = h(\mathbf{w} \cdot \mathbf{x} + b)$$

See notebook 1 here.

This can be scaled to a Feed Forward Network

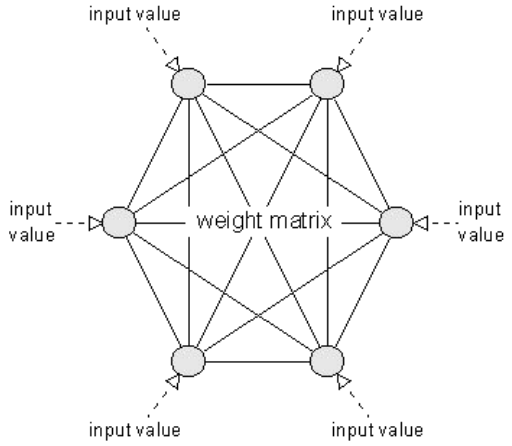
- The idea here is the same, just now you have multiple “nodes” that you can apply the equation in parallel
- Similarly, you can take the output of one equation and put it as the input to another equation with the same form, but different weights
- Voila - you have the FFN/ANN/MLP
- This did not catch on very quickly because the backpropagation algorithm was not yet popular



$$f(\mathbf{x}) = h(\mathbf{w} \cdot \mathbf{x} + b)$$

See notebooks 2 & 3 here.

Quick history lesson: Layerwise pre-training & other stuff



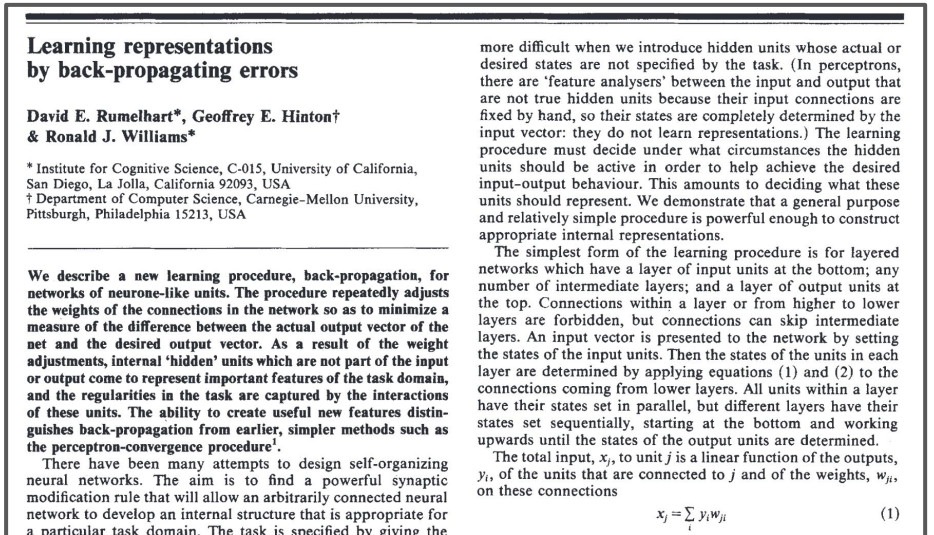
2024 Nobel Prize in Physics went to Hopfield and Hinton for work leading to “modern AI”

The Hopfield network was the first neural network I ever coded up ~2012

Geoffrey Hinton was a co-author of the paper on the backprop algorithm (https://www.nature.com/articles/323533a0)

Until it became computationally feasible to use backprop, it was popular to simply train “deep” models in a layerwise fashion.

This is what Yoshua Bengio is known most for. (https://proceedings.neurips.cc/paper_files/paper/2006/file/5da713a690c067105aeb2fae32403405-Paper.pdf)



Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

$$x_j = \sum_i y_i w_{ji} \qquad (1)$$

NN(NN(NN(NN(x)))) to the rescue

Rumelhart & Hinton had a strong hand in developing RNNs, but first “modern” take seems to be from Michael I Jordan (https://cseweb.ucsd.edu/~gary/PAPER-SUGGESTIONS/Jordan-TR-8604-OCRed.pdf)

Reading this paper shows how disconnected neural networks/AI was at this time from traditional ML and statistics/applied math

Diagrams & general intuition is mostly intact today though

What's in a recurrent neural network

With modern sensibilities, and an eye toward physical systems it's easy to see how RNNs are powerful approximators for dynamical systems

Instead of having to encode "time" as separate feature variables to input into your model, you get an explicit encoding of "time" ordering (replace time for sequence as you see fit)

To train these you use "back propagation through time" (BPTT), which essentially unrolls the recurrence, creating an arbitrarily deep network

RNN

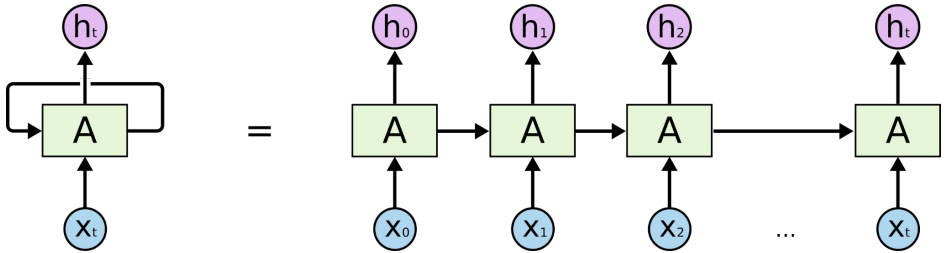
$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

Runge Kutta

$$
\begin{aligned}
y_{n+1} &= y_n + \tfrac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right), \\
t_{n+1} &= t_n + h
\end{aligned}
$$

for $n = 0, 1, 2, 3, \ldots$, using[2]

$$
\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right), \\
k_3 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right),
\end{aligned}
$$



See notebook 4 here.

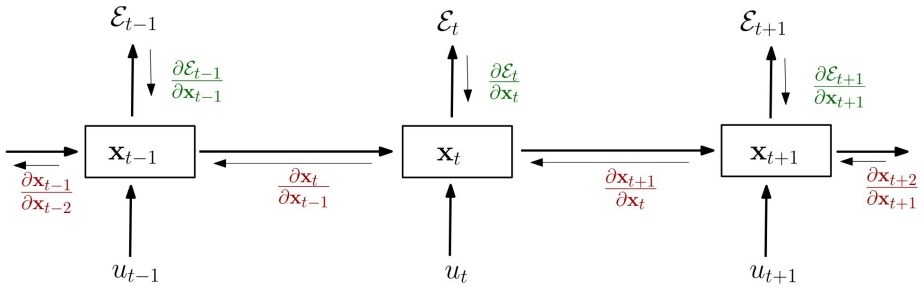
BPTT and exploding/vanishing gradients

It became practically a meme that RNNs were “difficult” to train - mostly due to the exploding/vanishing gradient problem.

For a mathematical treatment of the causes see “On the difficulty of training Recurrent Neural Networks” (https://doi.org/10.48550/arXiv.1211.5063)

This can happen with other deep networks as well, and are partially why ResNets are so good.

There are also other ways to avoid this issue, like batch normalization, gradient clipping, choosing smart activations, etc



$$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{1 \leq t \leq T} \frac{\partial \mathcal{E}_t}{\partial \theta}$$

$$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \leq k \leq t} \left(\frac{\partial \mathcal{E}_t}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} \frac{\partial^+ \mathbf{x}_k}{\partial \theta} \right)$$

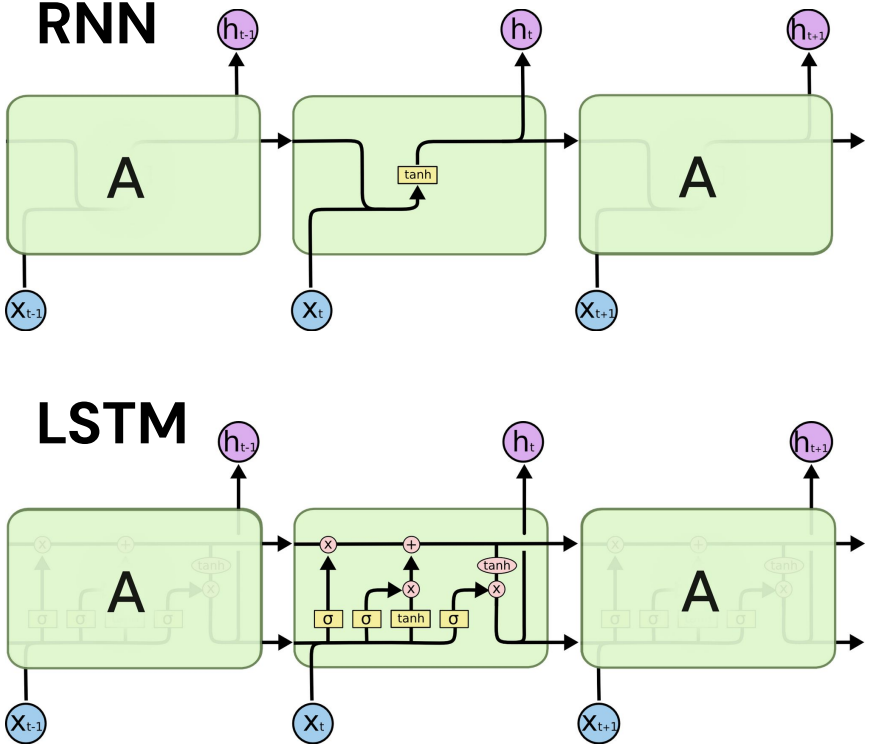
$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \prod_{t \geq i > k} \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_{i-1}} = \prod_{t \geq i > k} \mathbf{W}_{rec}^T diag(\sigma'(\mathbf{x}_{i-1}))$$

The Long Short Term Memory network

Hochreiter & Schmidhuber (1997) examined the exploding/vanishing gradient from a "gradient flow" perspective, and introduced the "Constant Error Carousel", memory cells, and gating mechanisms.
(https://www.bioinf.jku.at/publications/older/2604.pdf)

Original paper does not look particularly modern - I think the modern explanation/representation of LSTMs is mostly due to Alex Graves.
(https://doi.org/10.1007/978-3-642-24797-2_4)

Also most of my explanation is based on Chris Olah's blogpost on LSTMS.
(https://colah.github.io/posts/2015-08-Understanding-LSTMs/)

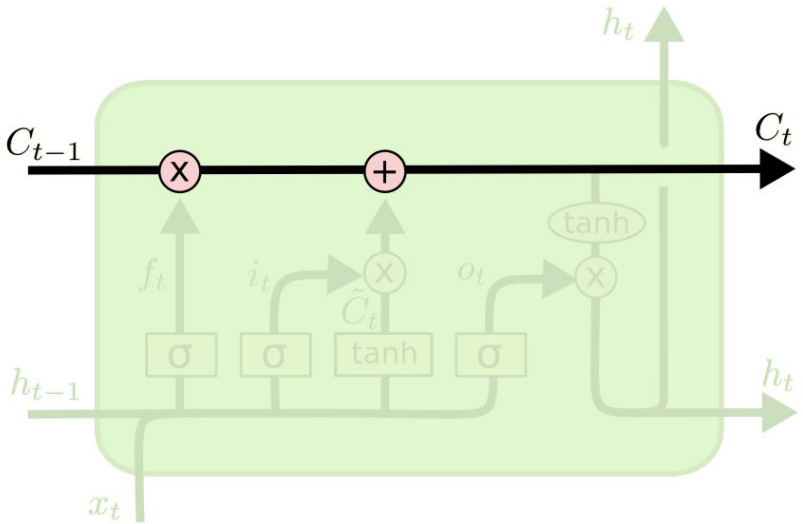


The Long Short Term Memory network

The “constant error carousel” and “memory cell” can be seen as this top path through the network. Not that it only interacts with the rest of the network through an add and multiply step.

This lets information flow through time freely - like a conveyer belt or “carousel”

The add/multiply are used to retain new information or forget old information using gated layers.



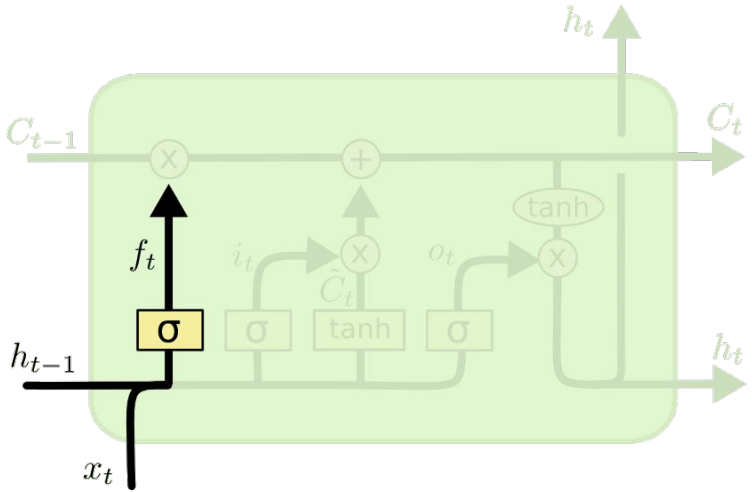
The Long Short Term Memory network

The first gate is the “forget gate”, which tells the network how to remove information based on the previous hidden state and current inputs.

From the equation - the end result is from a sigmoid, resulting a range from 0 to 1.

When multiplied agains the cell state this essentially tells the network what information to throw away (by outputting small values).

Similarly, higher values for elements of f result in retained information.



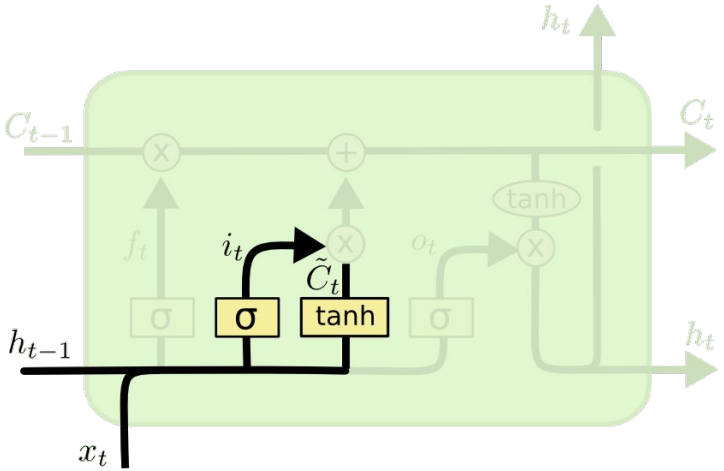
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

The Long Short Term Memory network

The second gate has two pieces. First is the “input gate layer” and second is the creation of “candidate values”

Input gate layer essentially chooses which locations in the cell state will be updated

Candidate values are the information to store in those locations.



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

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

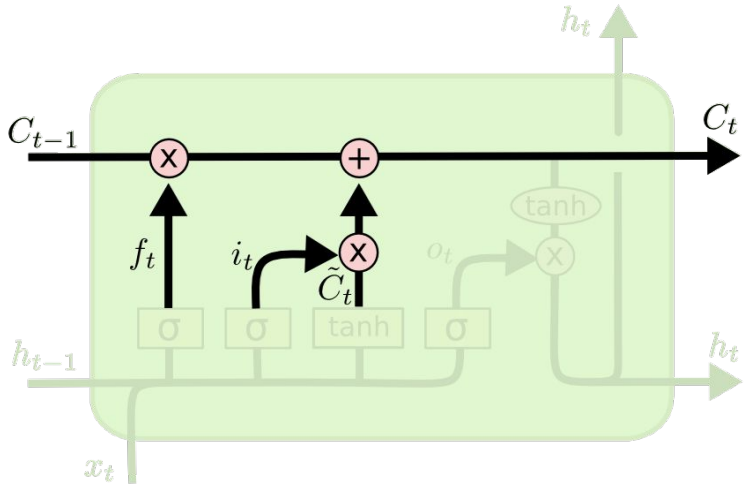
The Long Short Term Memory network

The second gate has two pieces. First is the “input gate layer” and second is the creation of “candidate values”

Input gate layer essentially chooses which locations in the cell state will be updated

Candidate values are the information to store in those locations.

The cell state is then updated by applying these two gates



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

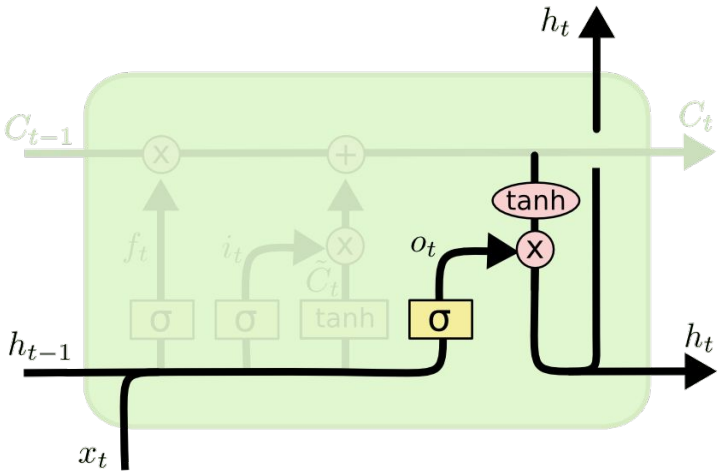
The Long Short Term Memory network

Following the update of the cell state, we then have to decide what to output and additionally update the hidden state.

First, output is calculated strictly from the inputs and hidden state. This let's the cell state be retained only to store information

Then, the hidden state is updated based on the output and a tanh of the cell state.

This completes the forward pass of the LSTM block for a single timestep. This would be repeated for each value in the sequence of inputs.



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

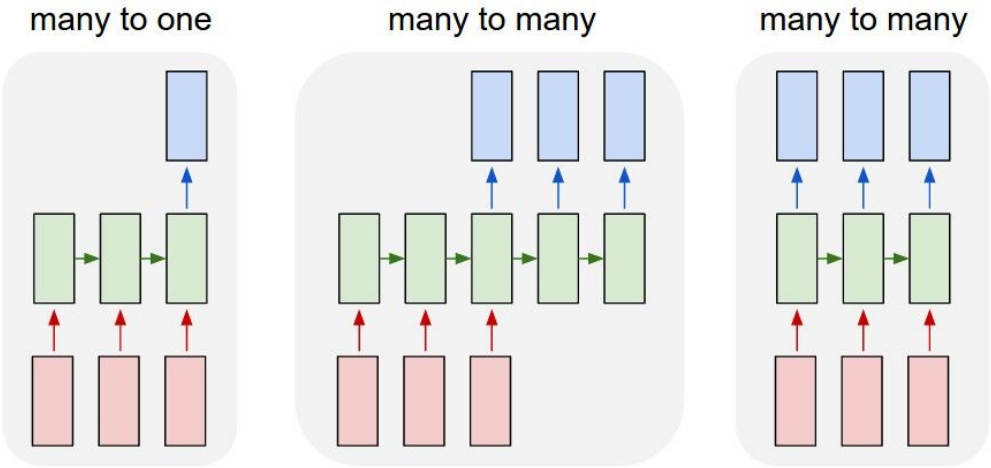
$$h_t = o_t * \tanh\left(C_t\right)$$

How are LSTMS used in practice?

LSTMs (and other RNNs) typically produce an output for every value that is input, but you don't actually have to calculate the loss & BPTT for the entire sequence.

Most commonly we do "many to one", also known as "sequence to value"

The "many to many" case is more commonly practiced in the middle regime, though if you have good initial states the rightmost can be possible.



How are LSTMS used in practice?

LSTMs (and other RNNs) typically produce an output for every value that is input, but you don’t actually have to calculate the loss & BPTT for the entire sequence.

Most commonly we do “many to one”, also known as “sequence to value”

The “many to many” case is more commonly practiced in the middle regime, though if you have good initial states the rightmost can be possible.

See notebook 5 here.

Some things worth noting!

Just because RNNs/LSTMs are sequence models does not imply they are autoregressive! They can be, but often are not treated as such.

When manipulating/preparing data the key thing to think about is what dimensions the model wants to consume. Usually for RNNs it's:

```
[batch, sequence, features]
```

Similarly, understanding what comes out of the layers implemented by pytorch, tensorflow, etc is super important! Understanding what this means is crucial:

```
o, (h, c) = self.lstm(x)
```

probably missing *a lot* of information & examples, hope this was useful!