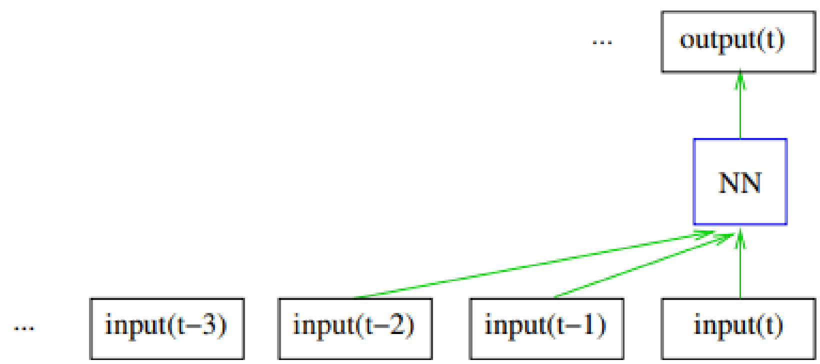
Modeling sequences

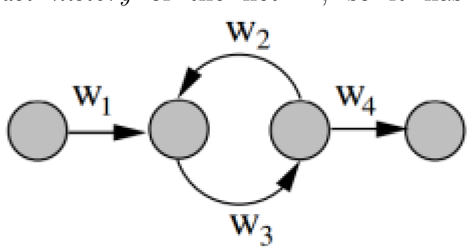
Typical problems: - turn an *input sequence into an output sequence* (possibly in a different domain): - *translation* between different languages - speech/sound recognition - … - predict the *next term in a sequence* - The target output sequence is the input sequence with an advance of 1 step. Blurs the distinction between supervised and unsupervised learning. - *predict a result from a temporal sequence of states*, Typical of Reinforcement learning, and robotics.

Memoryless approach

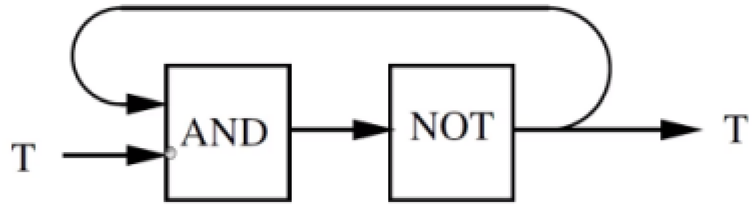
Compute the output as a result of a fixed number of elements in the input se-


quence:
Used e.g. in - Bengio's (first) predictive natural language model - Qlearning for Atari Games

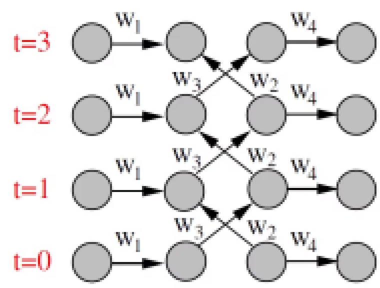
What is a RNN?

(This is an exercerpt from the final part of the previous lesson) An RNN is simply a neural network with cycles in it. The end. This means that, in presence of backward connections, ==hidden states depend on the *past history* of the net==, so it has some kind of memory in a sense.


As we know, in logical circuits having cycles cause some instabilitites…



...but these are solved usually by adding a *clock*. A similar concepts is preserved in RNN thanks to **Temporal Unfolding**, meaning that *activations are updated a precise time steps*. In this way, the RNN is basically a layered net that keeps reusing the same weights:



So it can easily be translated into a traditional feedforward NN. The only thing that we have to keep in mind is that the weights are shared between weights of the same layer at the start; however, they get updated differently after the first update.

Sharing weights through time It is easy to modify the backprop algorithm to *incorporate equality constraints* between weights. We compute the gradients as usual, and then *average gradients* so that they induce a *same update* (and preserve the weights). - If the initial weights satisfied the constraints from the start, they will continue to do. - N.B.: this same update is done if we want to preserve the same weights.

To constrain $w_1 = w_2$ we need: - $\Delta w_1 = \Delta w_2$ - compute $\dfrac{\partial E}{\partial w_1}$ and $\dfrac{\partial E}{\partial w_2}$ and use $\dfrac{\partial E}{\partial w_1} + \dfrac{\partial E}{\partial w_2}$ to update both w_1 and w_2.

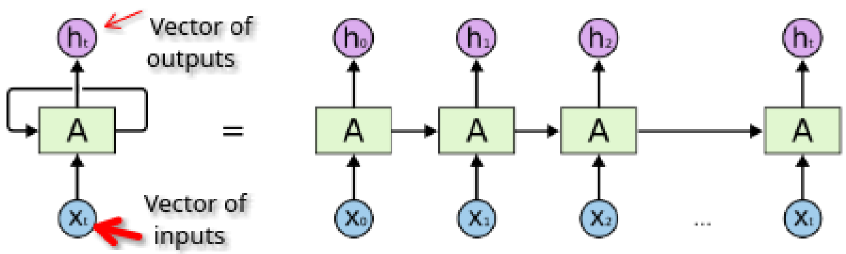
Backpropagation through time - BPTT

- think of the recurrent net as *a layered, feed-forward net* with *shared weights* and train the feed-forward net *with weight constraints*.
- reasoning in the time domain:
  - the forward pass builds up a stack of the activities of all the units at each time step.
  - the *backward pass* peels activities *off the stack* to compute the error derivatives at each time step.

– finally we add together the derivatives at all the different times for each weight.

Hidden state initialization We need to specify the initial activity state of all the *hidden* and *output units*. The best approach is to treat them as parameters, *learning them in the same way as we learn the weights*: - start off with an initial random guess for the initial states. - at the end of each training sequence, backpropagate through time all the way to the initial states to get the gradient of the error function with respect to each initial state. - adjust the initial states by following the negative gradient.

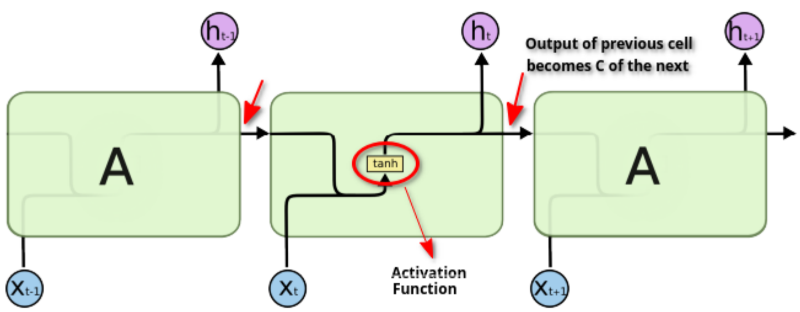
Long-Short Term Memory (LSTM)



Both the vector of inputs and the vector of outputs have the same length t.

A simple, traditional RNN

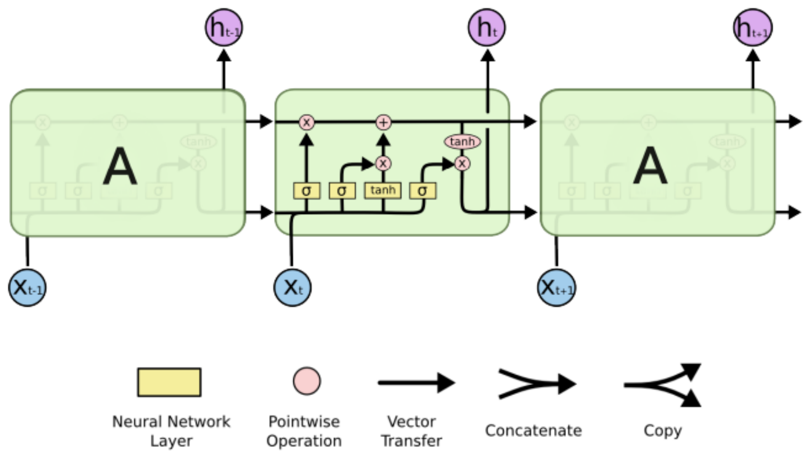
Let's see another example. The content of the memory cell C_t , and the input x_t are combined through a simple neural net to produce the output h_t that *coincides with the new content* of the cell C_{t+1}.



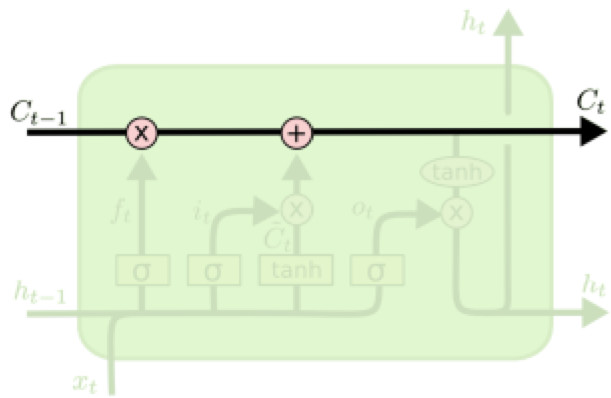
Why $C_{t+1} = h_t$? Better trying to *preserve the memory cell*, letting the neural net *learn how and when to update it.* - Many times, though, using these kind of the structure the memory may be lost in a way (because of the input x_t). Nevertheless, we try to preserve the memory as much as possible.

Also, $C_t \neq h_t$, since the content of a cell, before becoming the output, goes through some kind of post processing.

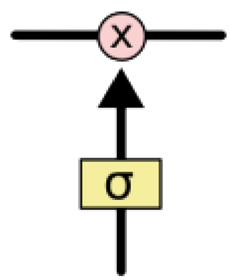
The overall structure of a LSTM



C-line and gates The LSTM has the ability *to remove or add information to* ***the cell state***, in a way regulated by suitable **gates**. **Gates** are a way to optionally let information through: *the product with a sigmoid neural net layer simu-*



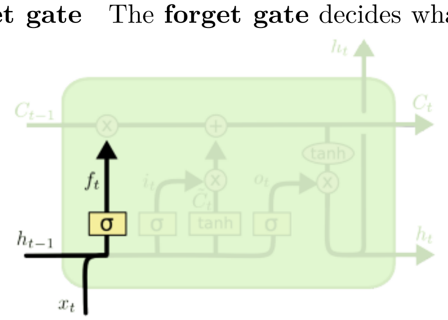
the C-line



a gate

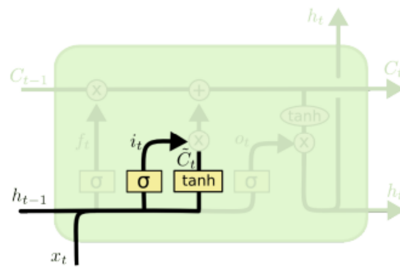
*lates a **boolean mask**.*

The forget gate The **forget gate** decides what part of the memory cell to



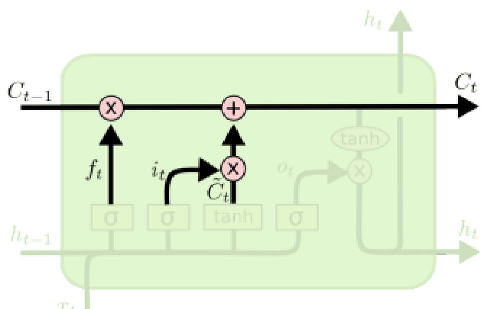
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

preserve.
In particular, by concatenating the input of the current cell w/ the output of the previous, which is then passed to a network layer, it generates a *mask* which decides which part of the content of the previous to keep and which of them to ignore.
This is a form of *attention*, as we will see.



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

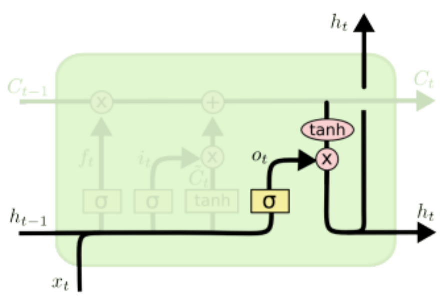
The update gate
As we've seen, the *input gate* decides *what part of the input to preserve.* The *tanh* layer creates a vector of new candidate values $\tilde{C}_t$ to be added to the state. Here's how the content of a cell is updated.



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

We multiply the old state by the boolean mask f_t . Then we add $i_t * \tilde{C}_t$.

The output gate The output h_t is a filtered version of the content of the cell.



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

The output gate decides what parts of the cell state to output. The *tanh* function is *used to **renormalize** values* in the interval [−1, 1].

Applications

They were used for NLP until the birth of transformers.

Asperti then did a long ass demo. You can find the demo here.