

# Recurrent Neural Networks

Cognitive Robotics

**Marco Ciccone**

Dipartimento di Informatica Elettronica e Bioingegneria  
Politecnico di Milano

# Outline

- Introduction
- Recurrent Neural Networks
- Vanishing and Exploding Gradient problem
- LSTM
- Applications

# Recap

CNNs are good for images because  
exploit their inherent spatial structure

Now we want to deal with sequential data

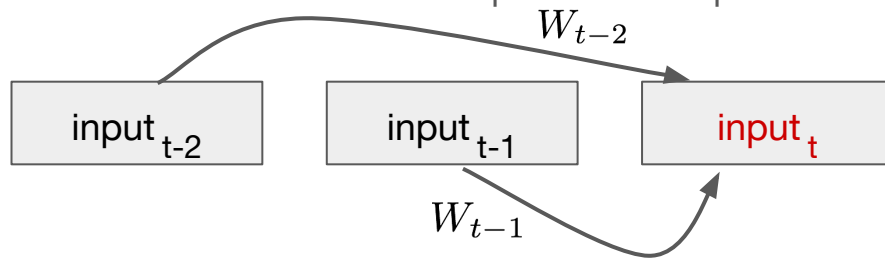
Feedforward Neural Networks treat input samples independently of the others

Information is propagated from the inputs to the outputs and only goes in one direction

# Memoryless models for sequences

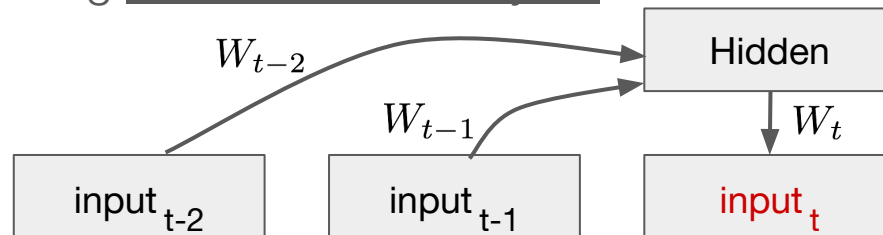
## Autoregressive models

Predict the next input in a sequence from a fixed number of previous inputs using “*delay taps*”.



## Feed-forward neural networks

Generalize autoregressive models by using nonlinear hidden layers.



# Beyond memoryless models

If we give add to a generative model **some hidden state**, and this hidden state has its own internal dynamics (**memory**), we get a much more interesting kind of model (State-Space models, Dynamical Systems... )

- It can store information in its hidden state for a long time.
- **If the dynamics is noisy** and the way it generates outputs from its hidden state is noisy, we can never know its exact hidden state.
- The best we can do is to infer a probability distribution over the space of hidden state vectors.

**This inference is only tractable for two types of hidden state model!**

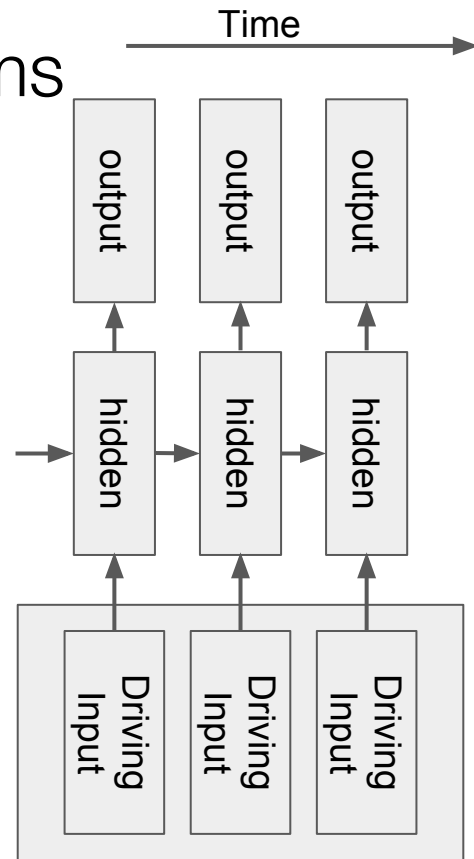
# Example 1: Linear Dynamical Systems

These are generative models.

They have a real valued hidden state that cannot be observed directly.

- The ***hidden state*** has linear dynamics with *Gaussian noise* and produces the observations using a ***linear model*** with *Gaussian noise*.
- (There may also be driving inputs)

To predict the next output we need to infer the hidden state. **How?**





A linearly transformed Gaussian is again Gaussian!

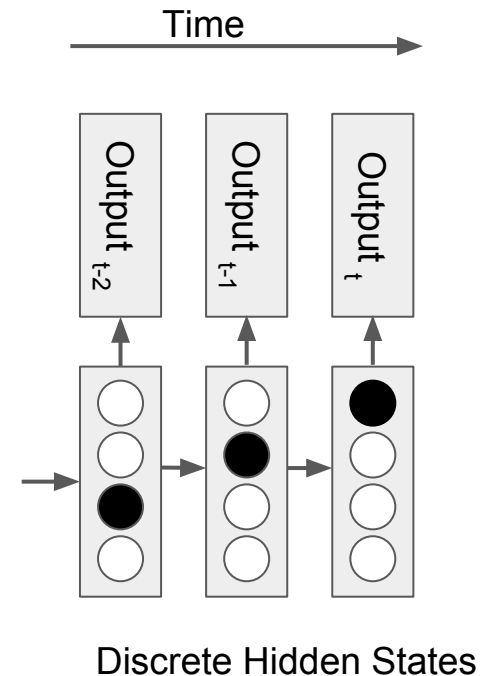
So the distribution over the hidden state given the data so far is Gaussian.

It can be computed using “Kalman filtering”.

## Example 2: Hidden Markov Models

Discrete state, arbitrary observation type

- State is not observed, **must be inferred**.
- Represent probability across  $N$  states with  $N$  numbers.
- Efficient algorithms exist for HMM inference (Viterbi Algorithm)



# RNNs properties

RNNs are very powerful, because they combine two properties:

- **Distributed hidden state** that allows them to store a lot of information about the past efficiently.
- **Nonlinear dynamics** that allows them to update their hidden state in complicated ways.

# Stochastic vs Deterministic

Linear dynamical systems and hidden Markov models are stochastic models:

- But the posterior probability distribution over their hidden states given the observed data so far is a deterministic function of the data.

Recurrent neural networks are deterministic models:

- You can think of the hidden state of an RNN as the equivalent of the deterministic probability distribution over hidden states in a linear dynamical system or hidden Markov model.

Remember Universal Approximation thm?  
RNNs can do even more!

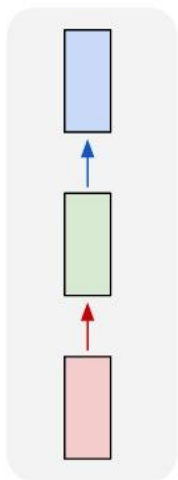
# RNNs are Turing Complete!

With enough neurons and time, RNNs can compute anything that can be computed by a computer.

[http://binds.cs.umass.edu/papers/  
1995\\_Siegelmann\\_Science.pdf](http://binds.cs.umass.edu/papers/1995_Siegelmann_Science.pdf)

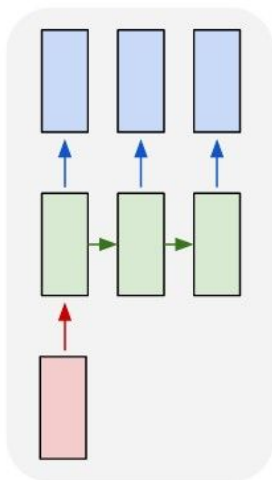
# Sequential data

one to one



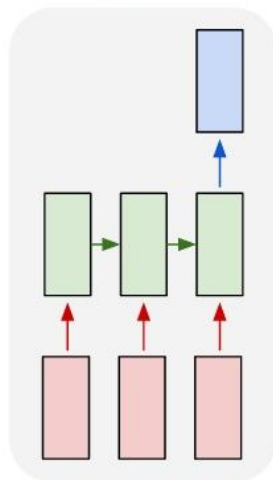
**Fixed-sized input to fixed-sized output** (e.g. image classification)

one to many



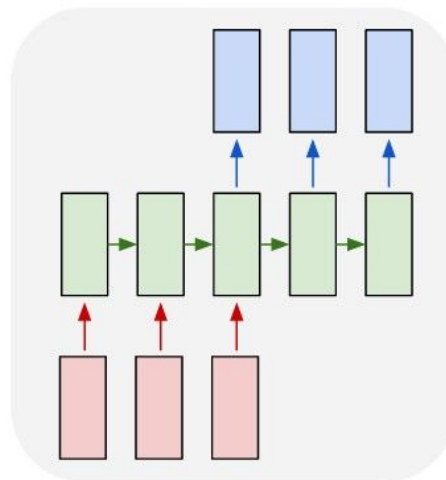
**Sequence output** (e.g. image captioning takes an image and outputs a sentence of words).

many to one



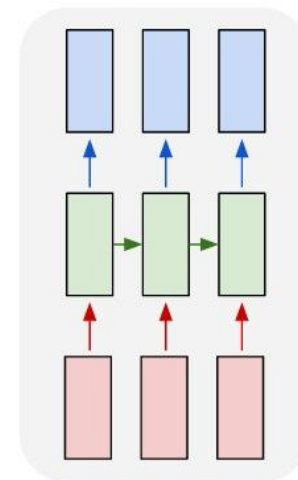
**Sequence input** (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment).

many to many



**Sequence input and sequence output** (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French)

many to many



**Synced sequence input and output** (e.g. video classification where we wish to label each frame of the video)

Each one of these tasks can be considered as a translation from one signal to another



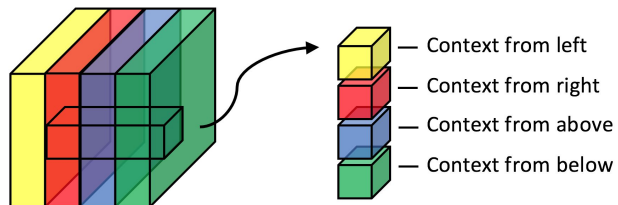
# Remark

Even if your data are not sequential you can decide to process them sequentially

# Example: ReNet Layer

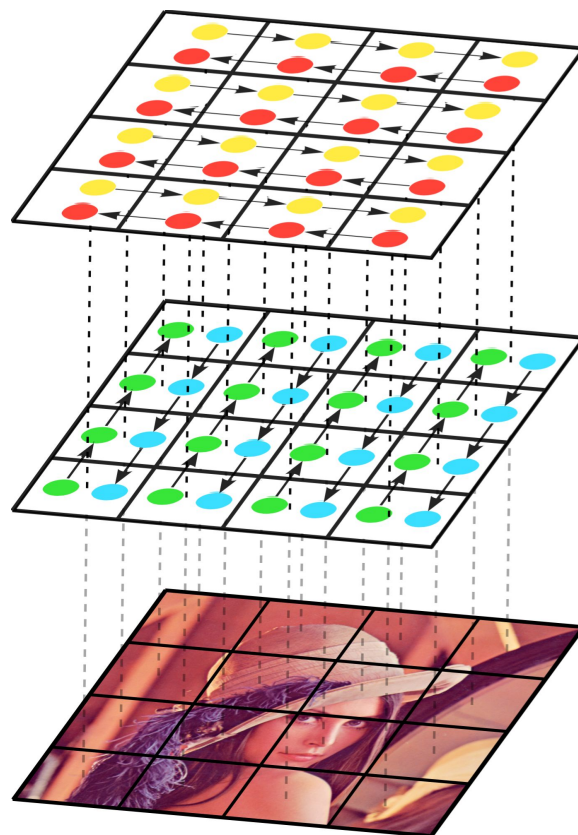
**4 RNNs** that scans the pixels of the image in different directions:

- Top-Down + Bottom-Up
- Left-Right + Right-Left



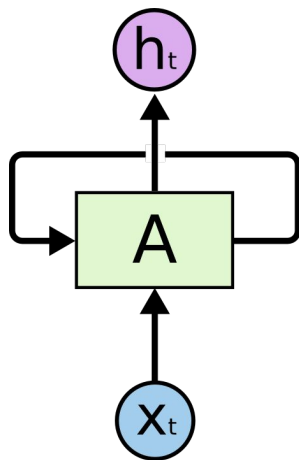
## Reference:

- [ReNet, Visin et Al.](#)
- [ReSeg, Visin, Ciccone et Al.](#)
- Code: <https://github.com/fvisin/reseg>



# Recurrent Neural Networks (RNNs)

- We introduce the notion of *sequentiality* to the model
- RNNs extend classical Feed-forward NNs with feedback connections to the hidden units.
- Through these connections the model can retain information about the past, enabling it to discover correlations between input samples



$$h_t = f_W(h_{t-1}, x_t)$$

new state

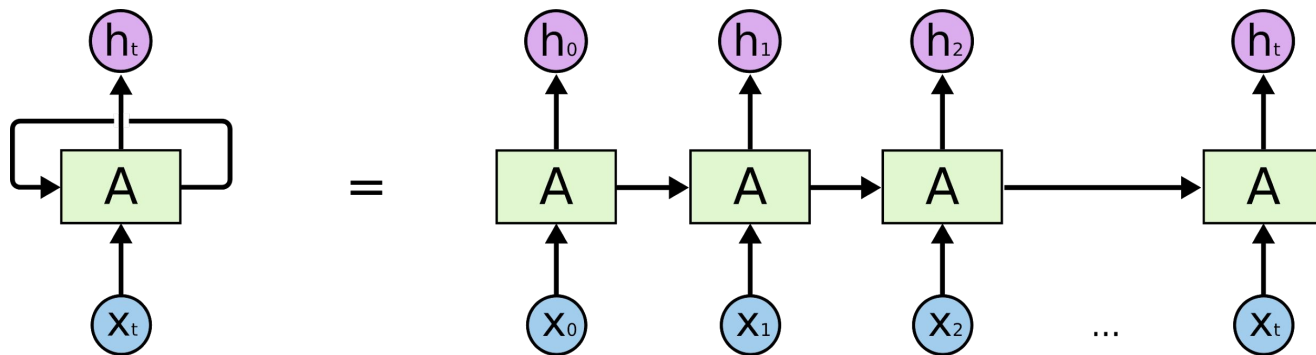
Parametric function

Old state Input

# Weight sharing through time steps

A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.

The **same function** and the **same parameters** are shared at every time step.



Unrolled view of a Recurrent Neural Network

# Remark

An unrolled RNN can be seen as a Deep  
Feedforward Neural Network

# Backpropagation Through Time (BPTT)

Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient.

*Gradients are obtained by applying chain rule on the unrolled graph*

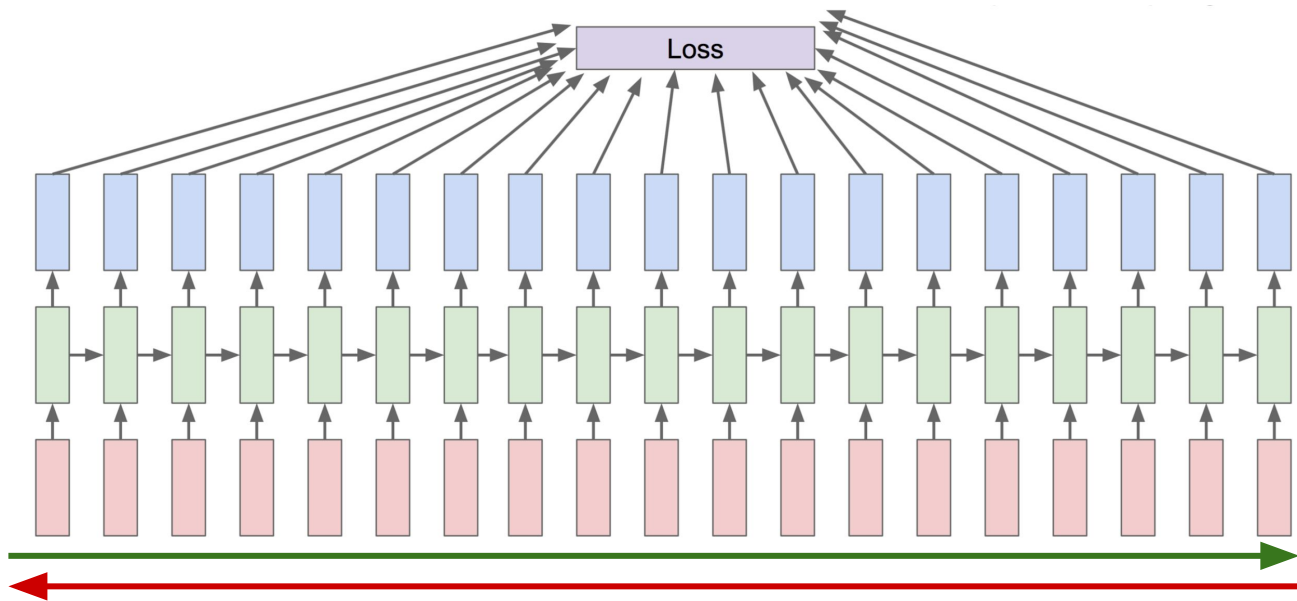


Image from  
Karpathy's  
class CS231n

# Truncated Backpropagation Through Time

Because of memory constraint if the sequence is very long we cannot backpropagate through the entire sequence length.

**Solution I:** Run forward and backward through chunks of the sequence instead of whole sequence

**Solution II:** Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps

# Truncated Backpropagation Through Time I

Run forward and backward  
through chunks of the  
sequence

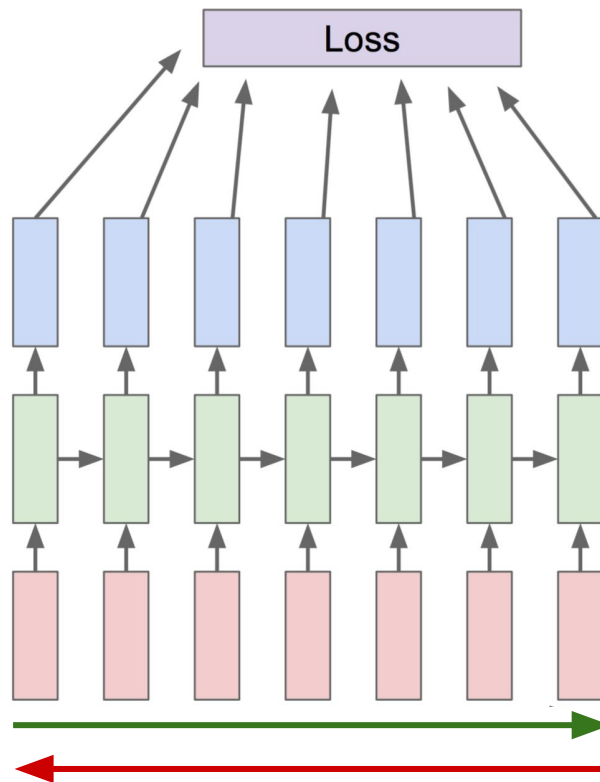


Image from  
Karpathy's  
class CS231n



# Truncated Backpropagation Through Time II

Forward in time forever, but only backpropagate for some smaller number of steps

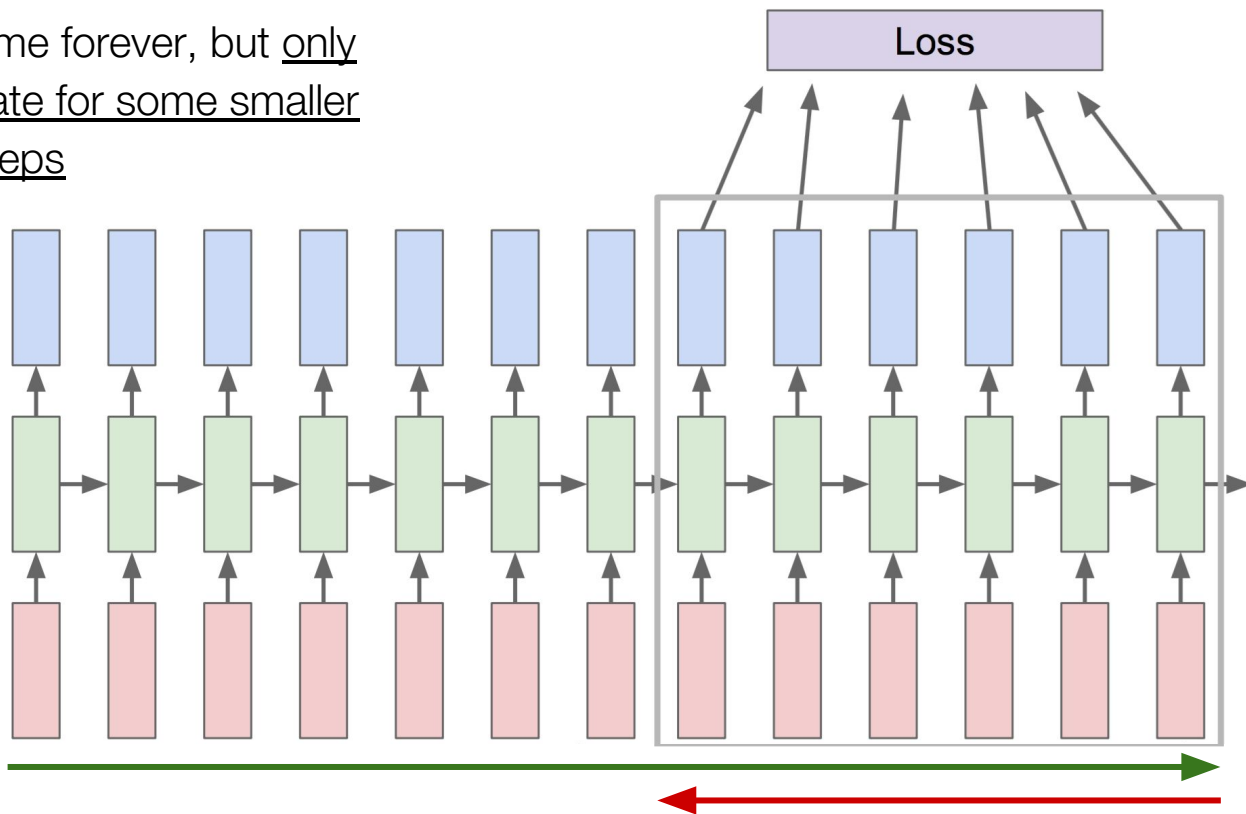


Image from  
Karpathy's  
class CS231n

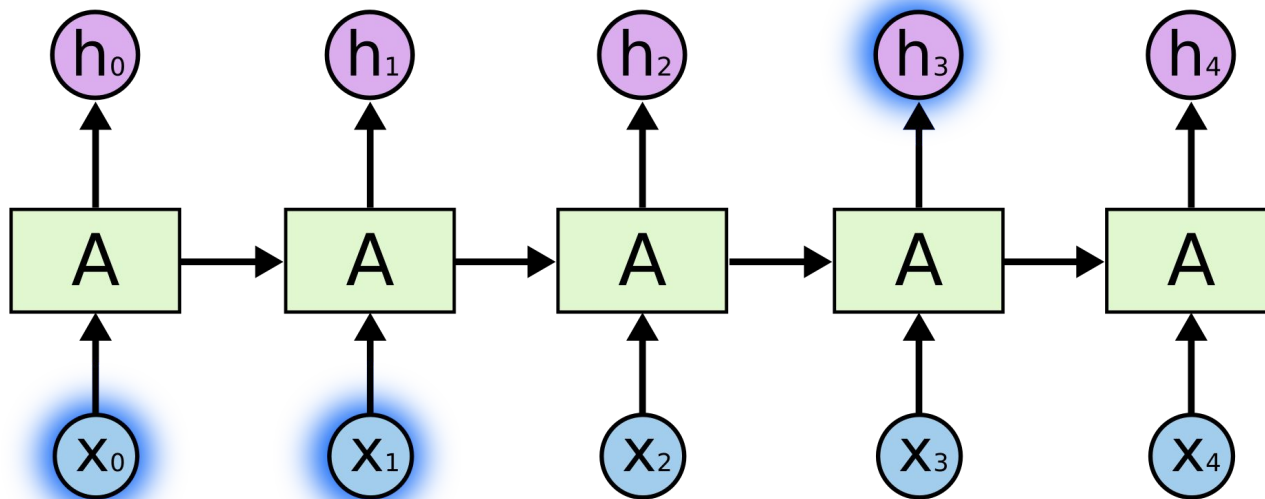
# Learning long-term dependencies is hard

With internal state, the network can “remember” things for a long time.

Can decide to ignore input for a while if it wants to... **BUT**

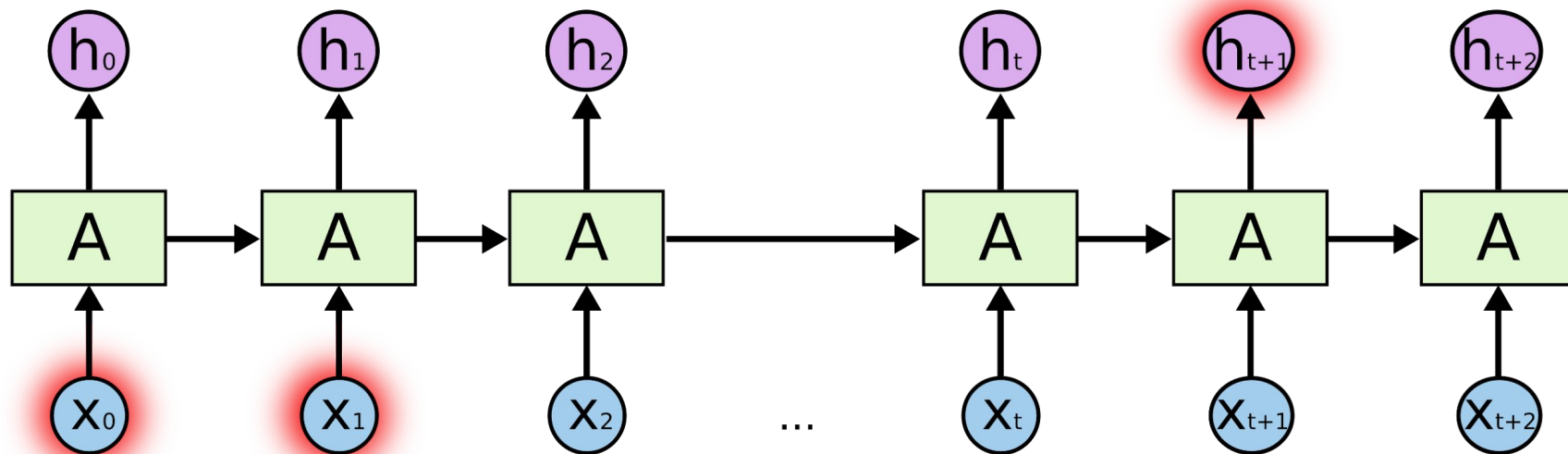
- **It is very hard to train an RNN to store information that's not needed for a long time.**
- In principle, the internal state can carry information about a potentially unbounded number of previous inputs.

# Short-term dependencies



Where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.

# Long-term dependencies



Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

# Why training RNNs is hard?

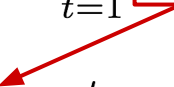
- **Vanishing gradient:** the error signal from later time steps cannot go enough back in time to influence the network at earlier time steps. This makes it difficult to learn long-term dependencies making it impossible for the model to learn correlation between temporally distant events.
- **Exploding gradient:** the error signal accumulates and explode

# Vanishing and Exploding Gradient

$$\mathbf{h}_t = \boldsymbol{\theta} \phi(\mathbf{h}_{t-1}) + \boldsymbol{\theta}_x \mathbf{x}$$

$$\mathbf{y}_t = \boldsymbol{\theta}_y \phi(\mathbf{h}_t)$$

$$\frac{\partial E}{\partial \boldsymbol{\theta}} = \sum_{t=1}^S \boxed{\frac{\partial E_t}{\partial \boldsymbol{\theta}}}$$

$$\frac{\partial E_t}{\partial \boldsymbol{\theta}} = \sum_{k=1}^t \frac{\partial E_t}{\partial \mathbf{y}_t} \frac{\partial \mathbf{y}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \boldsymbol{\theta}}$$


Sorry for changing the notation :)

# Vanishing and Exploding Gradient

$$\frac{\partial E_t}{\partial \theta} = \sum_{k=1}^t \frac{\partial E_t}{\partial \mathbf{y}_t} \frac{\partial \mathbf{y}_t}{\partial \mathbf{h}_t} \boxed{\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}} \frac{\partial \mathbf{h}_k}{\partial \theta}$$

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \prod_{i=k+1}^t \boxed{\frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}}} = \prod_{i=k+1}^t \theta^T \text{diag}[\phi'(\mathbf{h}_{i-1})]$$

$$\left\| \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \right\| = \|\theta^T\| \|\text{diag}[\phi'(\mathbf{h}_{i-1})]\|$$

$$\left\| \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \right\| \leq (\gamma_\theta \gamma_\phi)^{t-k}$$

$$\gamma_\theta = \|\theta\|_2 = \sqrt{\lambda_{\max}(\theta^* \theta)} = \sigma_{\max}(\theta)$$

Max singular value

# Vanishing and Exploding Gradient

We define the *spectral radius*  $\rho$  of a square matrix as the supremum among the absolute values of the elements in its spectrum, formally, let  $\lambda_1, \dots, \lambda_n$  be the (real or complex) eigenvalues of a matrix  $\mathbf{A} \in \mathbb{C}^{n \times n}$ , then its spectral radius is defined as

$$\rho(\mathbf{A}) = \max\{|\lambda_1|, \dots, |\lambda_n|\}$$

The spectral radius is closely related to the behavior of the convergence of the power sequence of a matrix and the following theorem holds:



# Vanishing and Exploding Gradient

**Theorem.** Let  $\mathbf{A} \in \mathbb{C}^{n \times n}$  with spectral radius  $\rho(\mathbf{A})$  then  $\rho(\mathbf{A}) < 1$  if and only if

$$\lim_{k \rightarrow \infty} \mathbf{A}^k = 0.$$

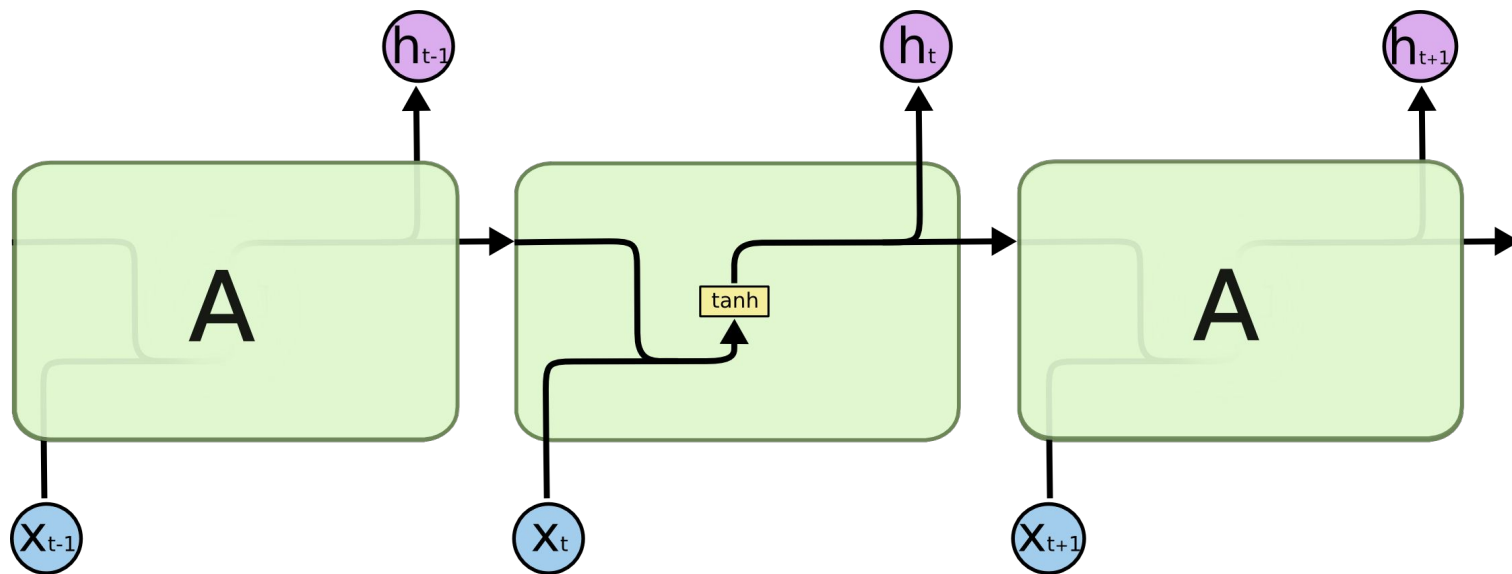
Moreover, if  $\rho(\mathbf{A}) > 1$ ,  $\|\mathbf{A}^k\|$  is not bounded for increasing values of  $k$ .

Considering the spectral radius:

- It is sufficient to be  $\rho < 1$  for long term components to vanish ( $t \rightarrow \infty$ )
- and necessary for for them to explode  $\rho > 1$

**Note:** this result is true for linear functions, but can be generalized for nonlinear ones using the max singular value  $\gamma_\theta$

# Vanilla RNN cell

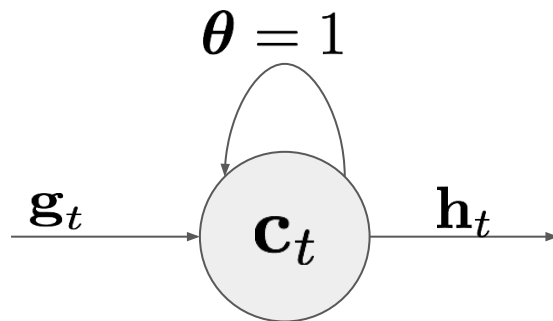


The repeating module in a standard RNN contains a single layer

## Naive (Not Working) Solution

$$\mathbf{c}_t = \theta \mathbf{c}_{t-1} + \theta_g \mathbf{g}_t$$

$$\mathbf{h}_t = \tanh(\mathbf{c}_t)$$



Changing again the notation...  
You can thank me later :)

**Solution:** We don't have vanishing or exploding gradient but...

**Problem:** *It's not a very good memory, it's just accumulating the input!*

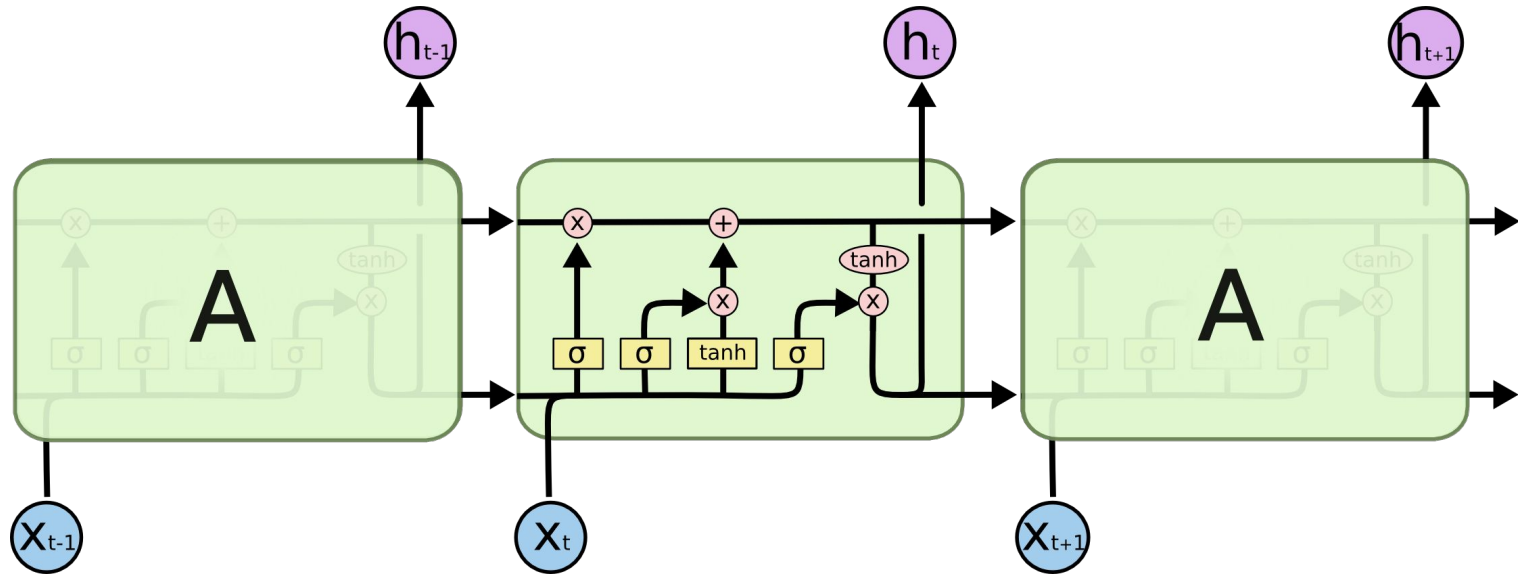
We need to know when and how much to *read, input, output, forget*....

# *An elegant* solution

Long Short Term Memory, Sepp Hochreiter, Jürgen Schmidhuber, 1997

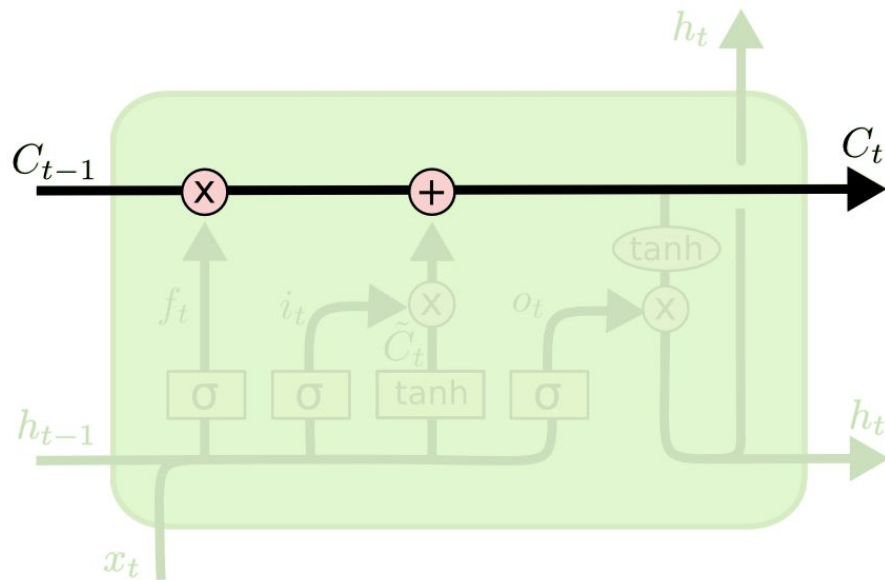
<http://www.bioinf.jku.at/publications/older/2604.pdf>

# Long Short-Term Memory (LSTM)



The repeating module in an LSTM contains four interacting layers.

# Cell Memory



The core idea of LSTM is to have a *cell memory* with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

# Gates

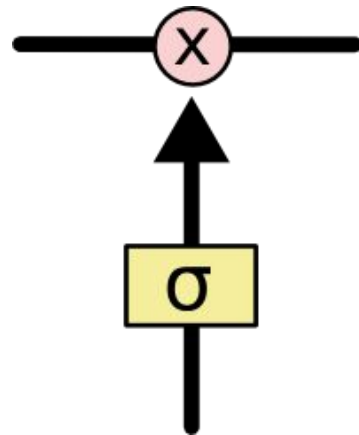
The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates

**Gates are a way to optionally let information through**

They are composed out of:

- a sigmoid neural net layer
- a pointwise multiplication operation

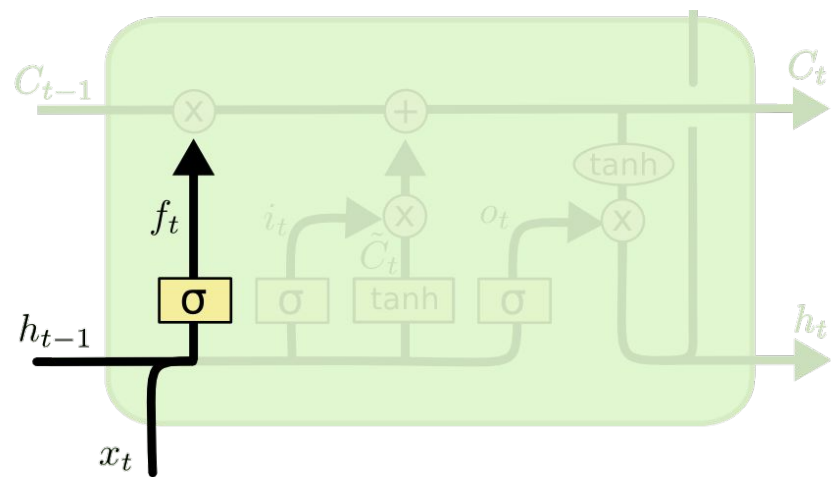
The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”



LSTM learns from data the behavior of the gates



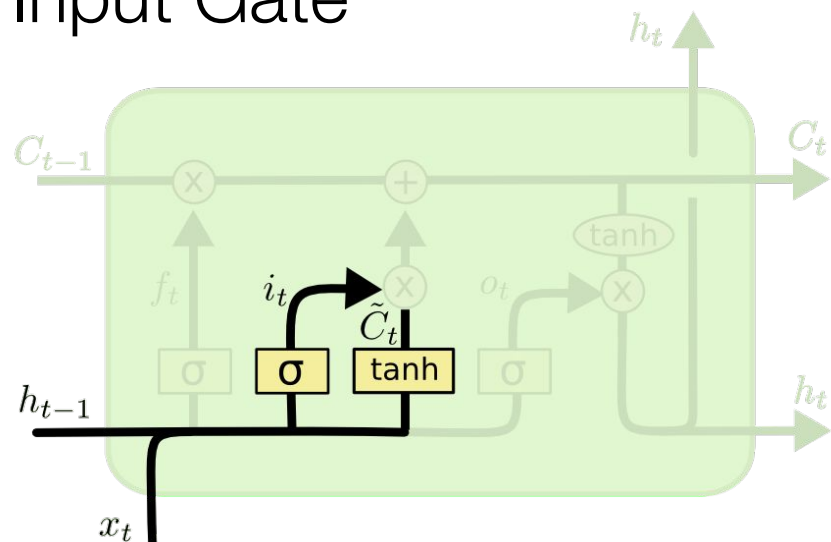
# Forget Gate



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

Decide what (how much) information we're going to throw away from the cell state  
 It uses a sigmoid where 1 represents "completely keep this" while a 0 represents "completely get rid of this."

# Input Gate



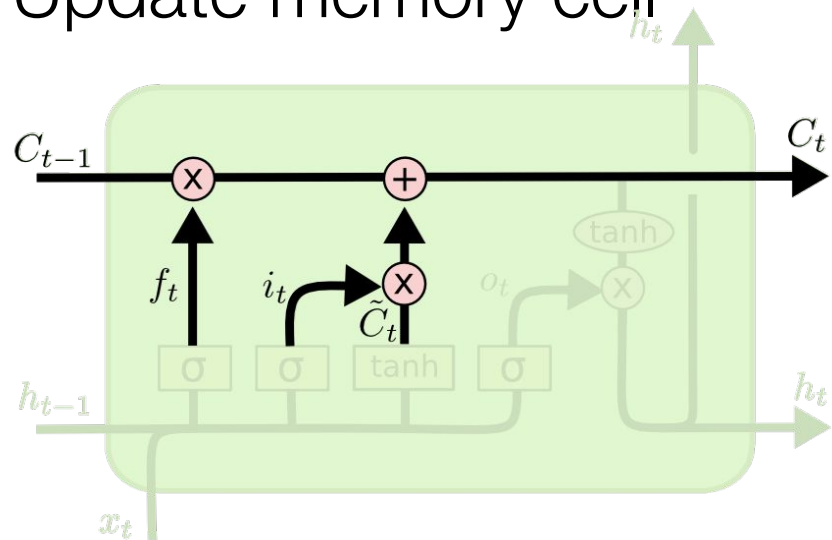
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

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

Decide what new information we're going to store in the cell state. This has two parts:

- A *sigmoid layer* called the "input gate layer" decides which values we'll update.
- A *tanh layer* creates a vector of new candidate values,  $\tilde{C}_t$ , that could be added to the state.

# Update memory cell

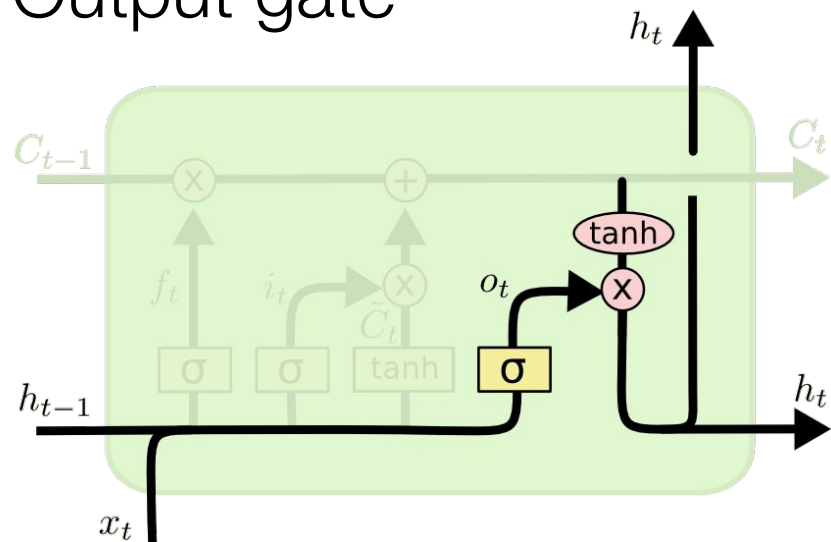


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

We need to update the old cell state,  $C_{t-1}$  into the new cell state  $C_t$

- We multiply the old state by the forget gate, forgetting the things we decided to forget earlier.
- Then we add the new candidate values, scaled by how much we decided to update each state value.

# Output gate



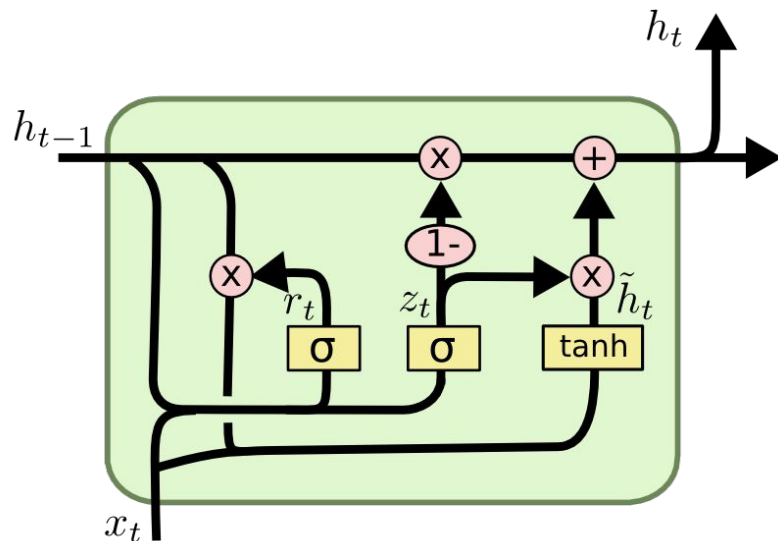
$$o_t = \sigma (W_o [h_{t-1}, x_t] + b_o)$$

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

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version.

- First, compute the output gate which decides what parts of the cell state we're going to output.
- We add a nonlinearity ( $\tanh$ ) to the cell and multiply it by the output gate, so that we only output the parts we decided to.

# Gated Recurrent Unit (GRU)



$$z_t = \sigma (W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma (W_r \cdot [h_{t-1}, x_t])$$

$$\tilde{h}_t = \tanh (W \cdot [r_t * h_{t-1}, x_t])$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

It combines the forget and input gates into a single “update gate.” It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.

# Bidirectional Recurrent Neural Networks

When conditioning on a full input sequence, no obligation to only traverse left-to-right

Bidirectional RNNs exploit this observation

- have one RNNs traverse the sequence left-to-right
- have another RNN traverse the sequence right-to-left
- use concatenation of hidden layers as feature representation

# One more issue: Hidden State initialization

- Need to specify the initial activations of the hidden units and output units.
  - Could initialize them to a fixed value (such as 0).
  - Better to treat the initial state as learned parameters.
  
- Learn these the same way we do with other model parameters:
  - Start off with random guesses of the initial state values.
  - Backpropagate the prediction error through time all the way to the initial state values and compute the gradient of the error with respect to these initial state parameters.
  - Update these parameters by following the negative gradient.

# Acknowledgements

This slides are highly based on material taken from:

- Geoffrey Hinton
- Hugo Larochelle
- Andrej Karpathy
- Nando De Freitas
- Chris Olah

You can find more details on the original slides

The amazing images on LSTM cell are taken from Chris Hola's blog:

<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>