# Introduction to Deep Learning

## LECTURE 5

---

Elad Hoffer
Nir Ailon
Ran El-Yaniv

Technion Israel Institute Of Technology,
Computer-Science department

# Overview

1. Sequence modeling

2. Recurrent neural networks

2.1 Training recurrent networks

2.2 Vanishing and exploding gradients

3. Gated architectures

3.1 Long-short-term-memory

3.2 Gated recurrent unit

4. Tricks of the trade

# Sequence modeling

# Sequence modeling

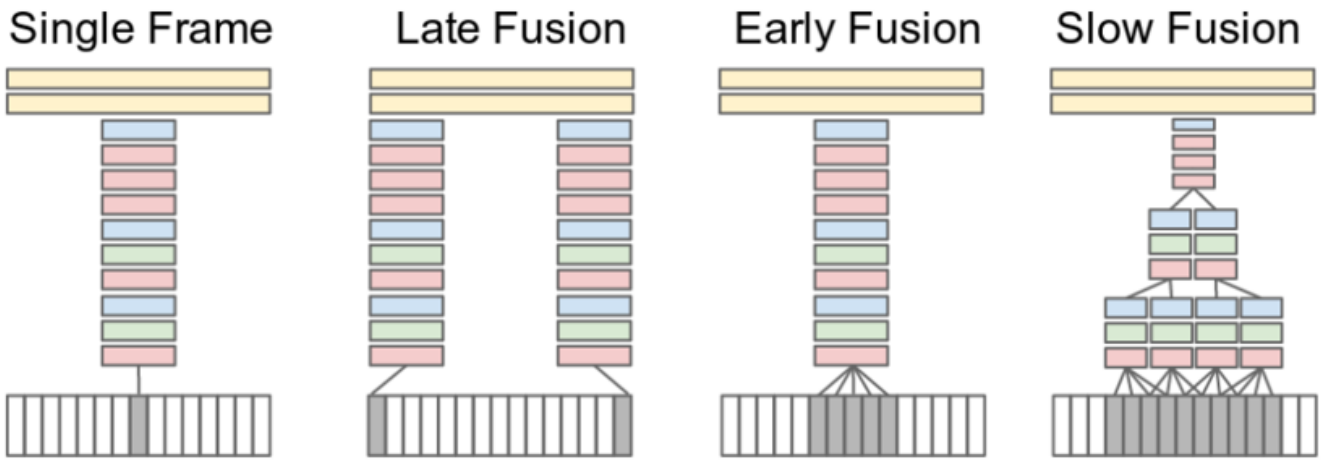
Many interesting problems are time dependent - where our model is expected to infer using a sequence of inputs, or output a sequence of targets. Some popular examples:

- Speech recognition
- Handwriting recognition
- Language translation
- Caption generation from images

# Time-delayed neural networks

The most basic approach is using *time-delayed neural networks*. TDNNs are feed-forward NNs that represent time explicitly with a fixed-length window of the recent history.

For example, a convolutional network trained to classify video instances (Karapathy, 14')



# Time-delayed neural networks

Properties of time-delayed neural networks:

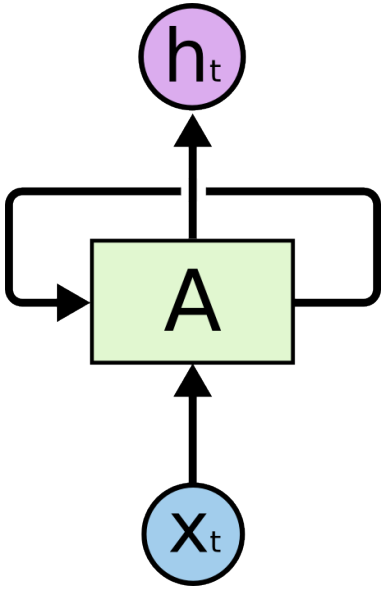
- Fixing the window size makes long-term dependency hard (even impossible) to learn
- Linear increase of the number of parameters for larger time-window
- Time-shift invariant since it has no internal state

Any better way to model temporal data?

# Recurrent neural networks

# Recurrent neural networks

Recurrent neural networks (RNNs) are networks where connections between units form a directed cycle.




$$h_t = \phi\left(W_i x_t + W_r h_{t-1} + b\right)$$

$x_t \in \mathbb{R}^N$ – input at time $t$

$h_t \in \mathbb{R}^M$ – *state* at time $t$

$W_i \in \mathbb{R}^{M \times N}, W_r \in \mathbb{R}^{M \times M}, b \in \mathbb{R}^M$

$\phi$ is usually a bounded non-linearity $(Tanh, Sigmoid)$

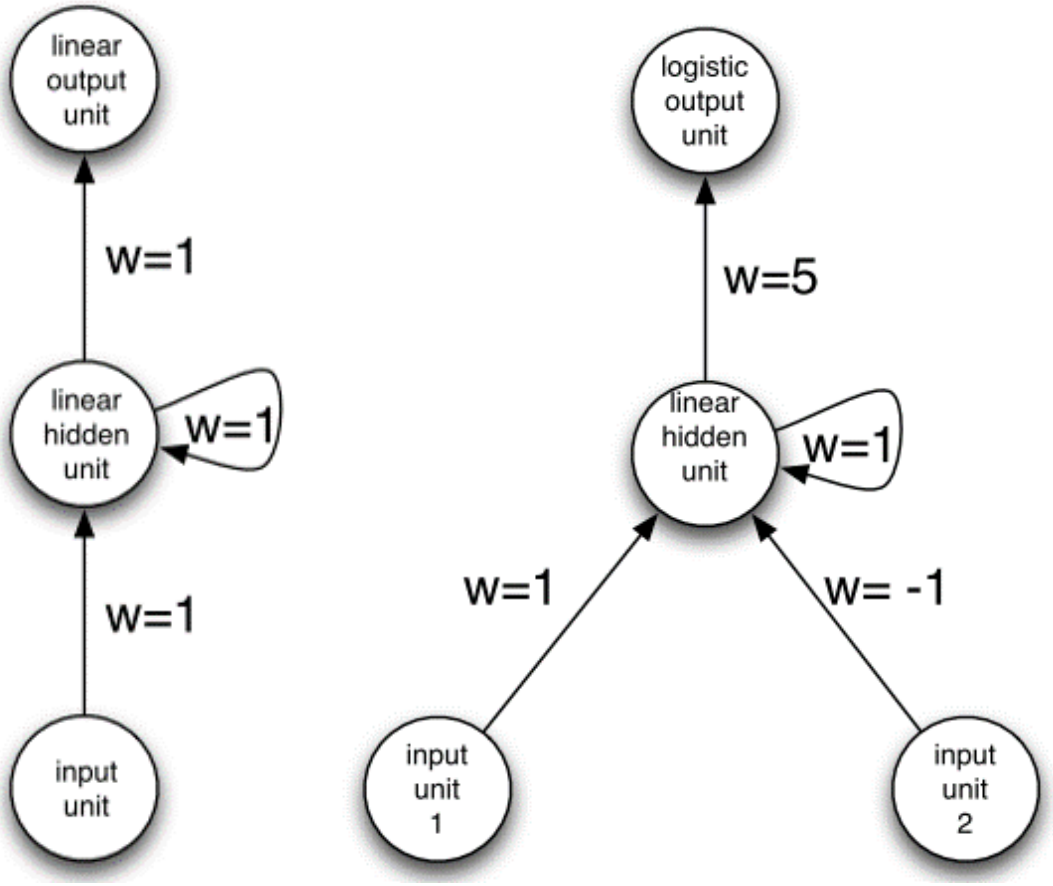
# Recurrent neural networks

Recurrent neural networks properties:

- RNN hidden states work as memory on previous inputs and states
- Temporal dependency and internal memory allow processing of sequences of inputs
- Able to address wide range of time-dependencies
- Able to learn and infer sequences of varying length

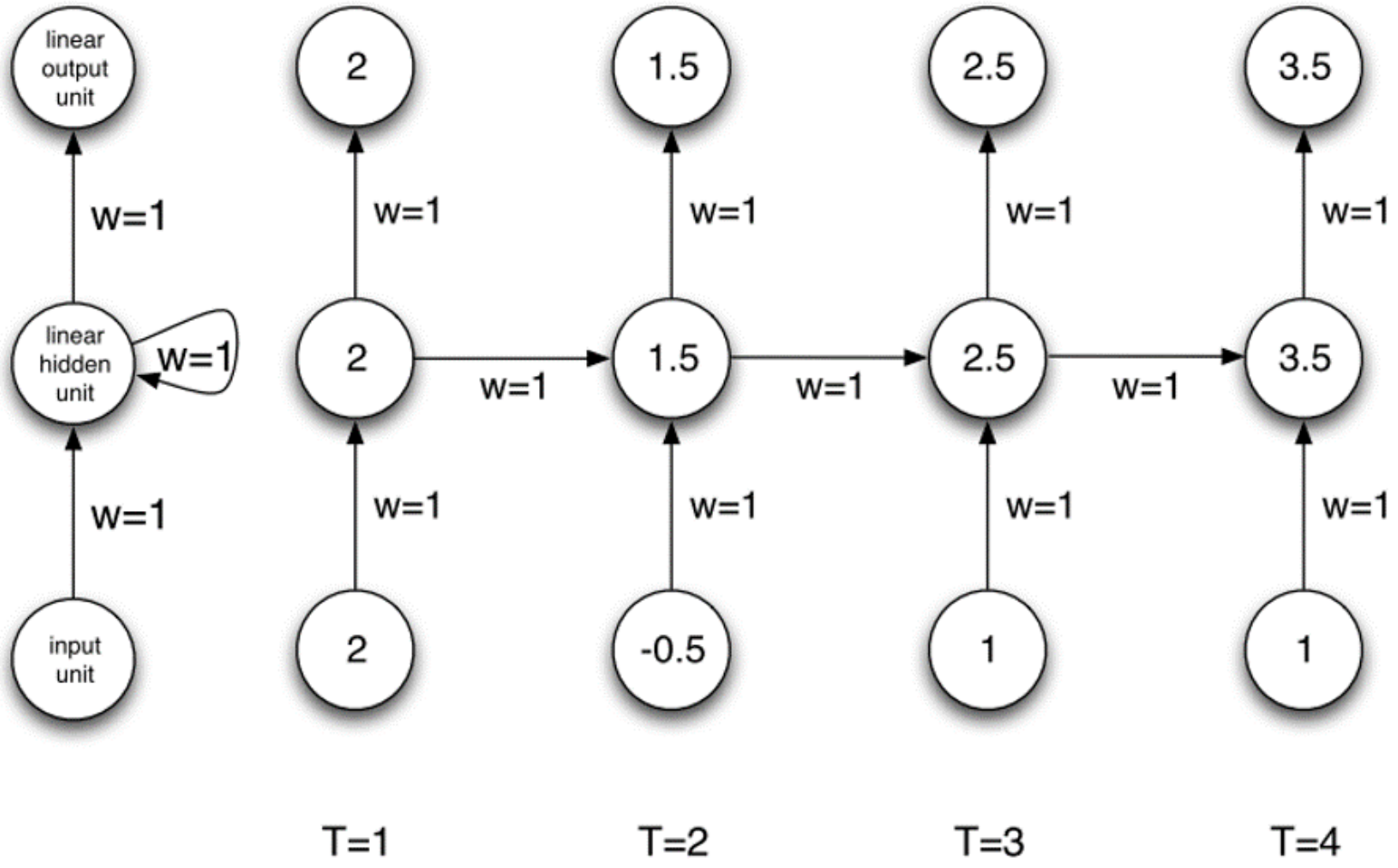
# Recurrent neural networks

What theses RNNs do?



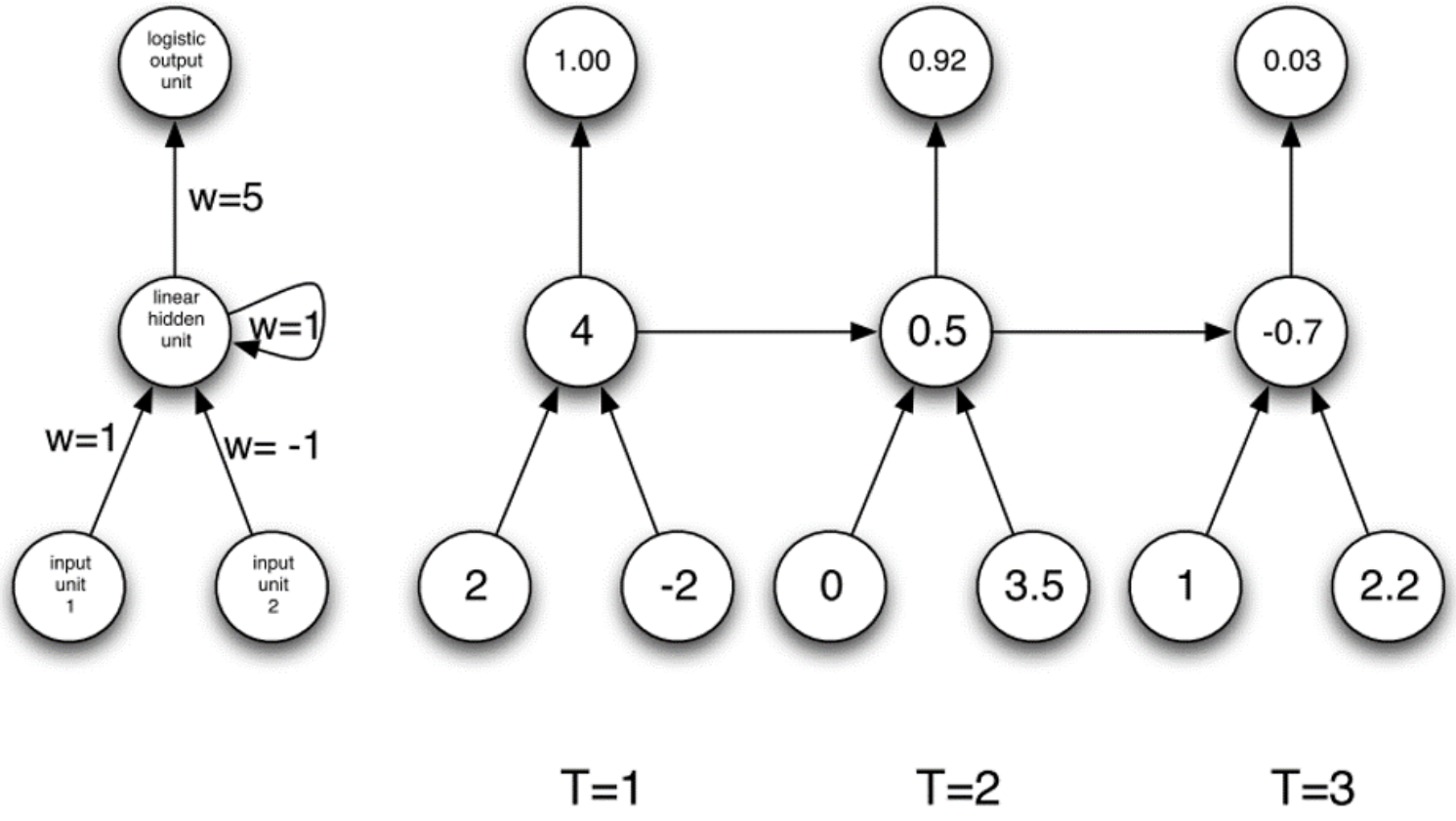
# Recurrent neural networks

This one calculates the sum of inputs over time:



# Recurrent neural networks

This one determines if the total value of the first or second inputs are larger:



# Recurrent neural networks

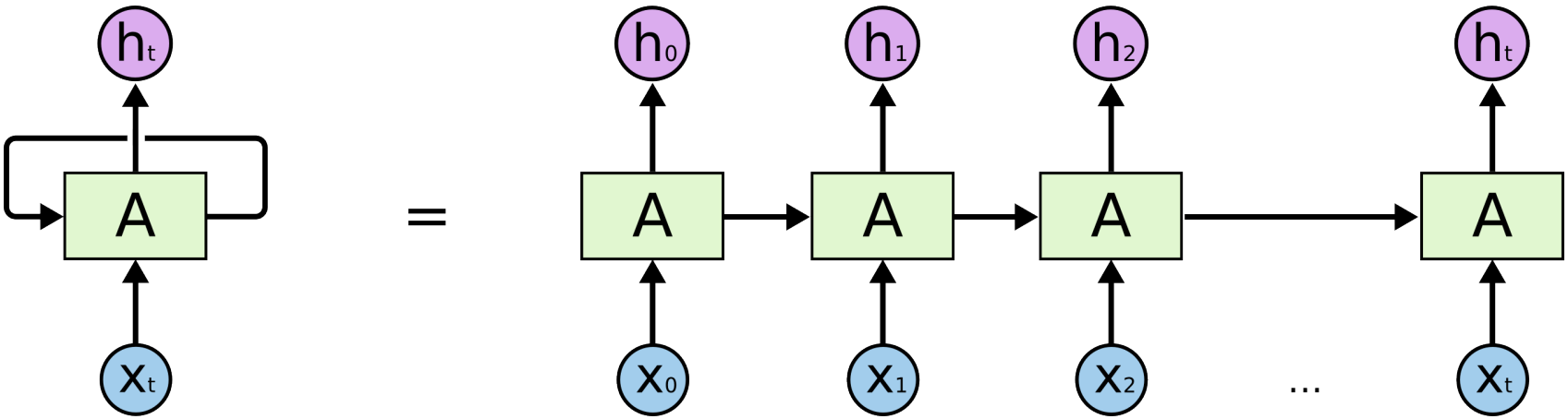
- RNNs essentially describe programs.
- They are proven to be Turing-complete (Siegelmann, Sontag) in the sense that they can to simulate arbitrary programs (with proper weights).
- Similar to universal approximation theorem for feed-forward neural nets - does not tell us what kind/size is needed for a given problem, or how hard it is to learn it.

So, how do we train recurrent networks?

# Back-propagation through time

In order to learn time-dependent signals, we need to use *Back-propagation through time*:

- Unfold network in time (remember all intermediate steps)
- Apply gradient back-propagation recursively back in time from T to 1
- Sum over the whole sequence ($t = 1..T$) to get the derivatives with respect to the weights



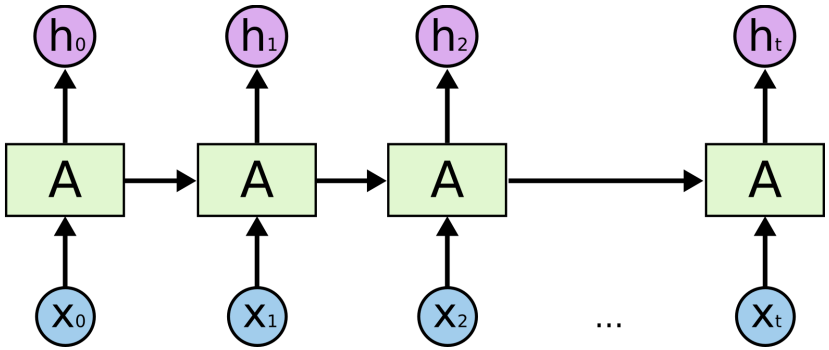
# Vanishing and exploding gradients

Recurrent networks tend to suffer from the infamous problem of *vanishing or exploding gradients* (Hochreiter, 91')

- Back-propagated error signals either shrink rapidly, or grow out of bounds.
- In fact, they decay exponentially in the number of layers, or they explode.

Why does it happen?

# Vanishing and exploding gradients




$$h_t = \phi\left(W_i x_t + W_r h_{t-1} + b\right)$$

We wish to optimize the weight according to some error $E$:

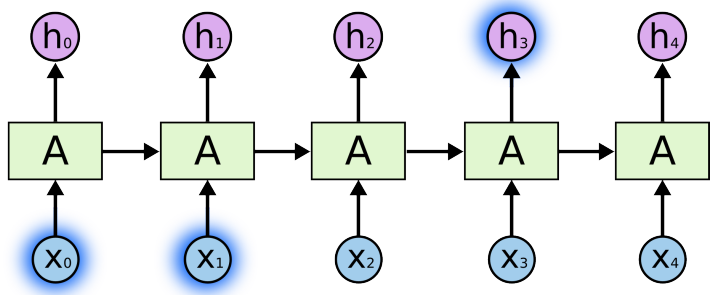
$$\frac{\partial E}{\partial W} = \sum_{1 \le t \le T} \frac{\partial E_t}{\partial W}, \qquad \frac{\partial E_t}{\partial W} = \sum_{1 \le k \le t} \left( \frac{\partial E_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W} \right)$$

$$\frac{\partial h_t}{\partial h_k} = \prod_{t \ge i > k} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{t \ge i > k} \mathbf{W}_r^T \, diag(\phi'(h_{i-1}))$$

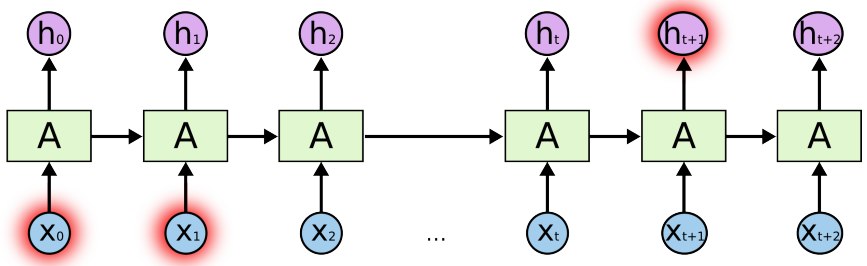
# Vanishing and exploding gradients

The vanishing or exploding gradients problem can be seen as a symptom for our attempt at modeling varying time dependencies

- Short term dependency




- Long term dependency



# Gated architectures

# Long-short-term-memory

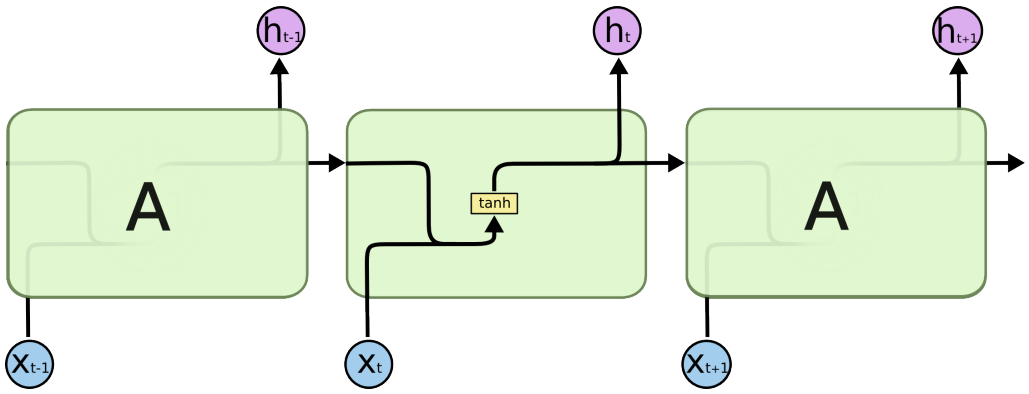
Long Short-Term Memory (LSTM) architecture was proposed by Hochreiter and Schmidhuber, '97.

- Designed specifically to counter the vanishing and exploding gradient problem
- It uses *gate units* to control the information flow within the network
- Empirically found to be much better at finding and exploiting long range dependencies within the data (compared to simple RNNs)
- Very popular today as a recurrent network model

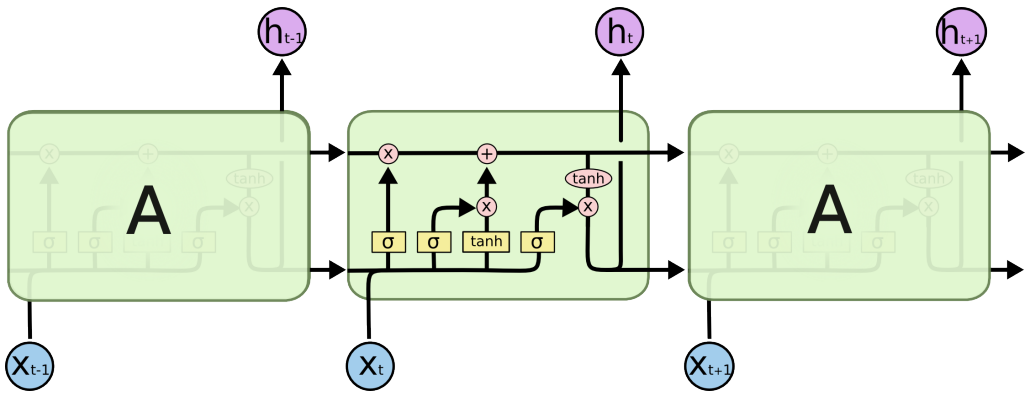
# Long-short-term-memory

Moving from RNNs to LSTMS units

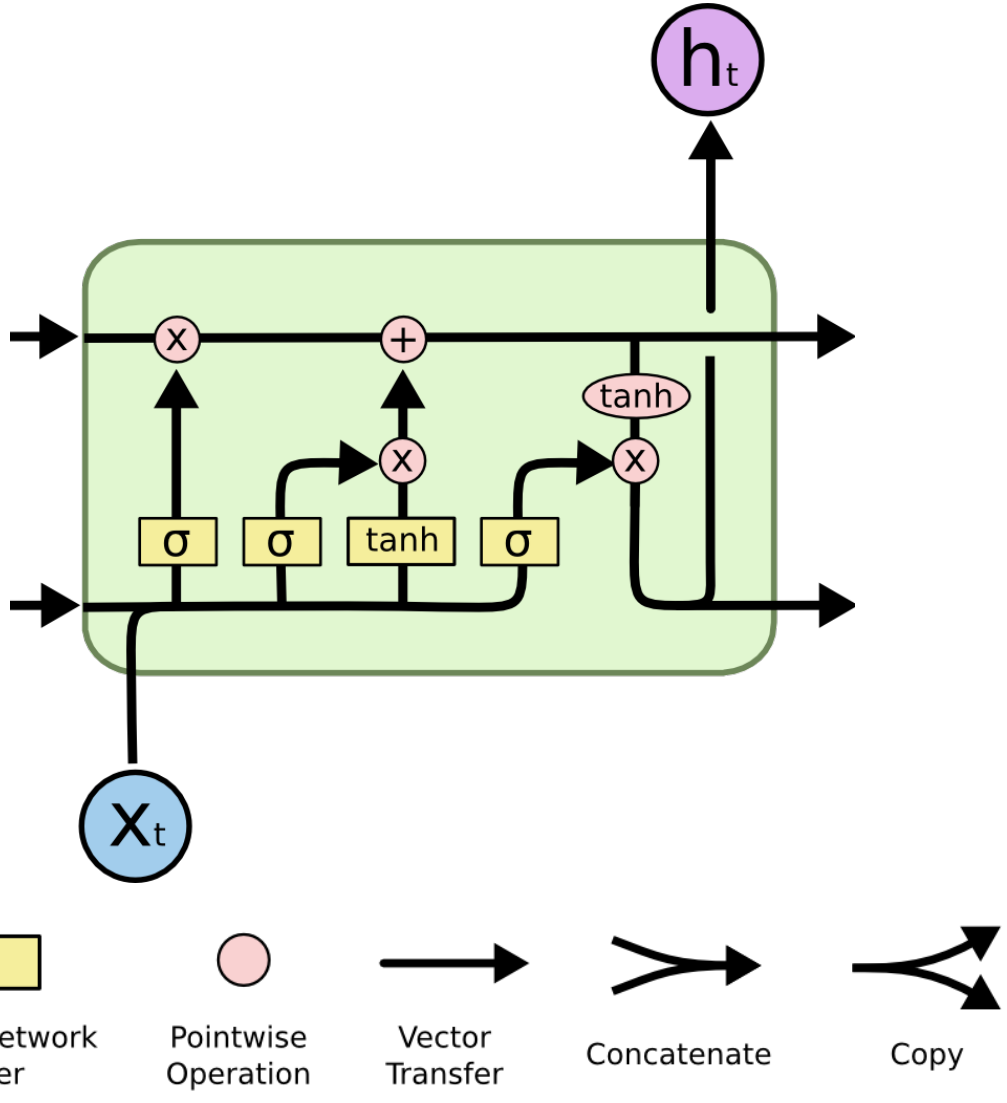
- RNN




- LSTM

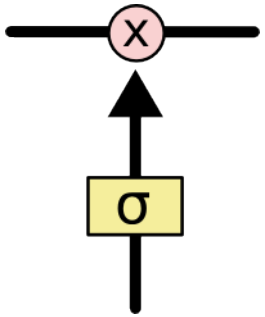


# Long-short-term-memory



# Gate units

- Gates are a way to optionally let information through.
- They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.
- The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through.

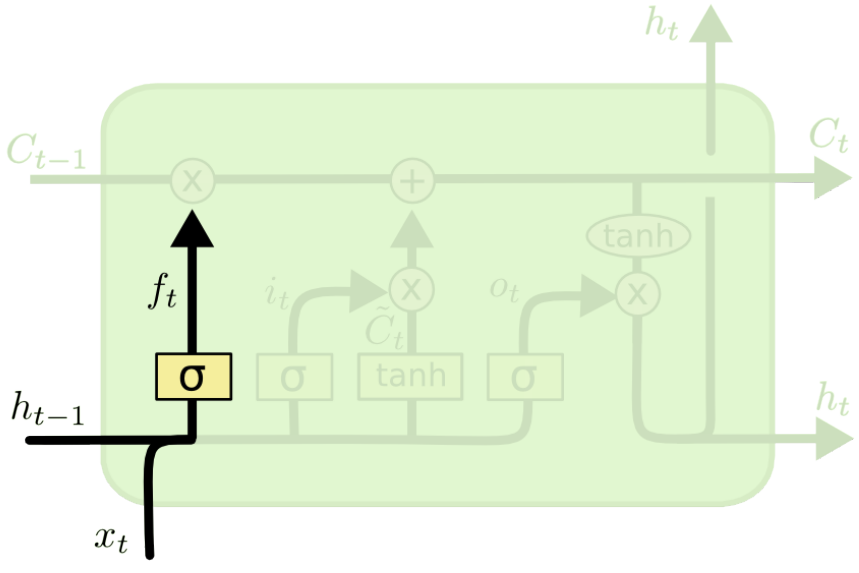



An LSTM has three of these gates

# Forget gate

We'll start with the "forget gate"

- Controls the amount of information kept from previous state. outputs a number between 0 and 1 for each number in the cell state

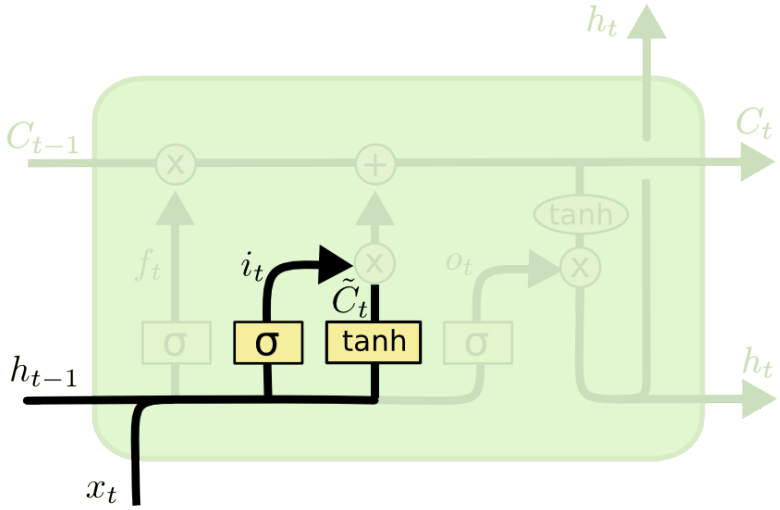



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

# Input gate and candidate state

The next step is to decide what new information we're going to store in the cell state.

- First, a gate layer called the "input gate" decides which values we'll update.
- Next, a tanh layer creates a vector of new candidate values, $\tilde{C}_t$.



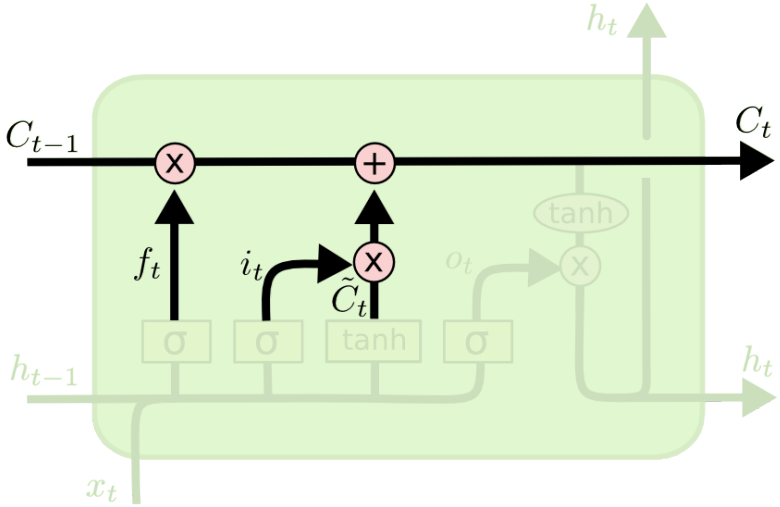

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

# Candidate state update

It's now time to update the old cell state,$C_{t-1}$, into the new cell state $C_t$.

- We multiply the old state by $f_t$, forgetting the things we decided to forget earlier.
- We then add it $i_t * \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.

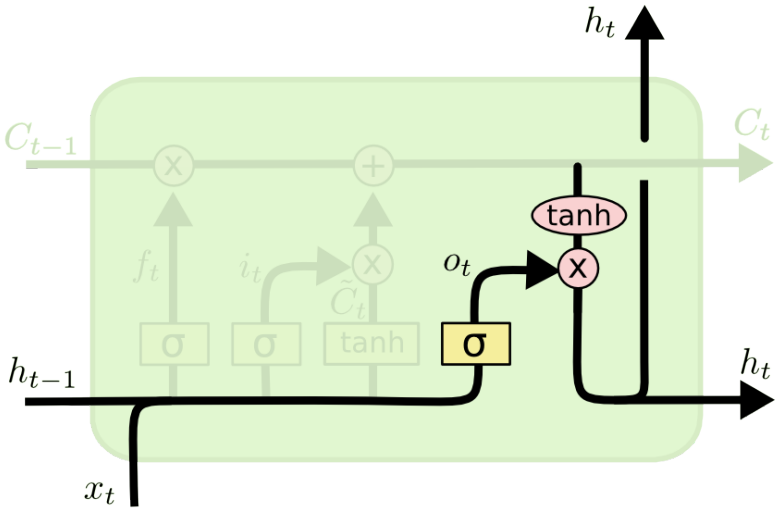



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

# Output gate

Finally, we need to decide what we're going to output. This output will be a filtered version of the state.

- First, we compute an "output gate" which decides what parts of the cell state we're going to output.
- Then, we apply a tanh on the cell state and output the gated version of it.



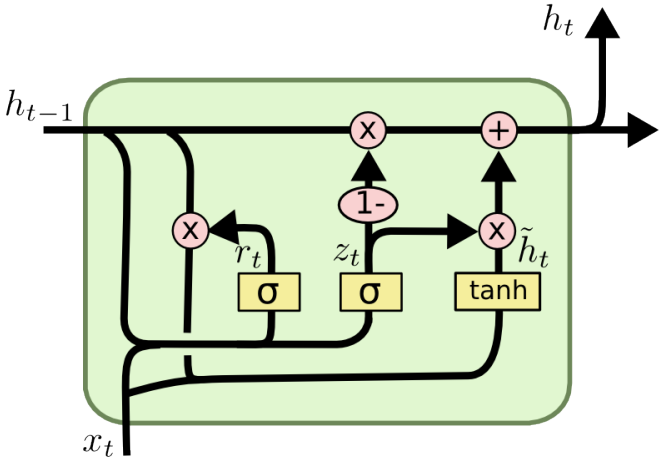

$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

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

# Gated recurrent unit

Gated recurrent unit (GRU) is a recently introduced alternative:

- Similarly to the LSTM unit, the GRU has gating units that modulate the flow of information.
- The activation $h_t$ of the GRU at time $t$ is a linear interpolation between the previous activation $h_{t-1}$ and the candidate activation $\tilde{h}_t$
- GRU exposes the whole state each time.




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

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

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

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

# Tricks of the trade

# RNN - tricks of the trade

Some useful techniques for training recurrent networks:

- Gradient clipping - to reduce the chances of exploding gradients, it is helpful to normalize them using a predetermined maximum norm (usually around 5):

$$\text{if } \left\| \frac{\partial E}{\partial W} \right\|_2 > V_{max}$$

$$\left( \frac{\partial E}{\partial W} \right)_i = \frac{V_{max}}{\| \frac{\partial E}{\partial W} \|_2} \left( \frac{\partial E}{\partial W} \right)_i$$

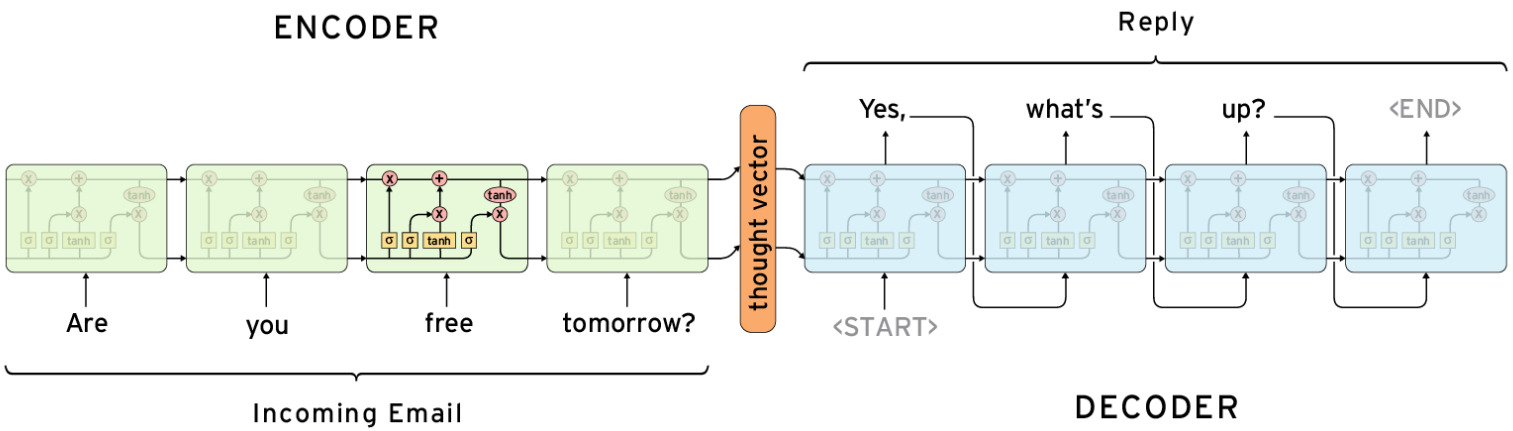
  This is very effective in practice, and should be used in every recurrent training.

# RNN - tricks of the trade

- Per parameter adaptive optimization - can sometime be crucial for convergence. Adam and Rmsprop usually work.
- Dropout, batch-normalization may need special care to work - straight-forward usage will work only on non-recurrent connections.
- Weight initialization - initializing the recurrent weights to be close to identity can speed-up convergence. Similarly, the forget-gate bias in LSTM should be initialized to a large constant (>1).
- Using small batches - recurrent networks are much more affected by batch size. It is therefore beneficial to use small batches compared to CNN training.

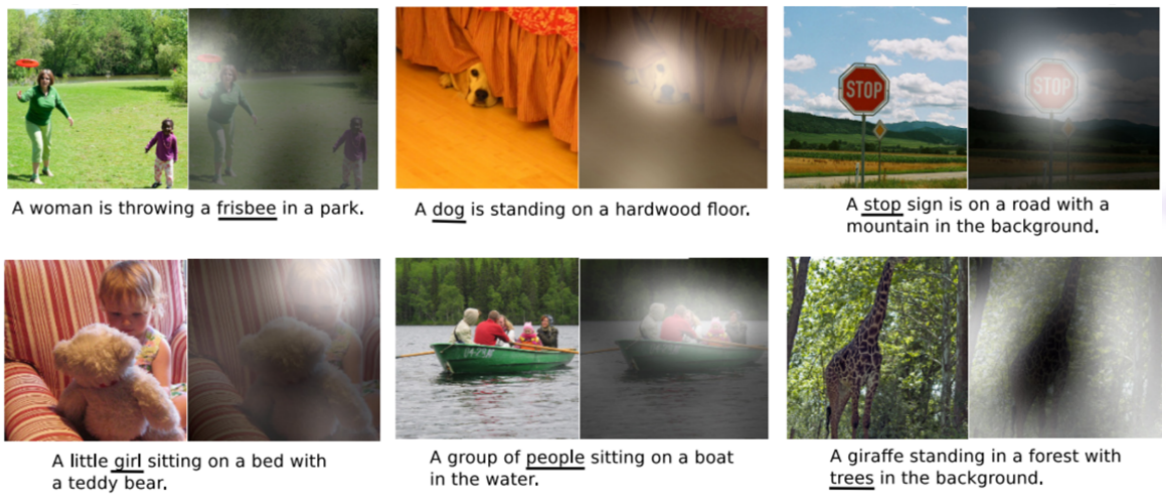
# Recent trends

- Memory augmented architectures - *Memory networks, Neural Turing machines, Stack-augmented networks*
- Sequence-to-sequence learning



# Recent trends

- Attention mechanisms - learn a state-dependent weighting over the input space in order to maximize an objective




of the officers had to have bullet fragments removed from his arm later , according to the *ent315* . the *ent454* reported that the officers had been driving through the neighborhood dressed in plain clothes . the officers returned fire , and several suspects scattered , *ent195* . *ent47* told the newspaper , adding that the officers believed that they were targeted . but a public information officer for the *ent315* disputes that possibility . `` the officers were in plain clothes , '' *ent309* told *ent100* . `` this can not be called targeting . the narcotics officers from the 77th division were driving in an unmarked police vehicle around 64th and *ent223* when they were shot at and they returned fire . '' three individuals were detained for questioning , according to *ent309* , but were not arrested . the names of the injured officers have not been released .

**X** _UNK_ : `` this can not be called targeting ''

# References

Figures were borrowed from Christopher Olah's site:

Understanding LSTMs