Lecture 4

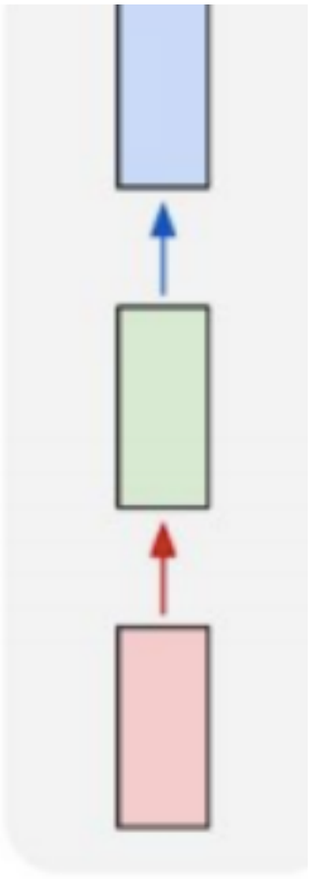
Recurrent Neural Networks

Lecture Plan

- RNN Intro
- RNN Applications
- RNN Scheme
- Vanishing Gradient Problem
- LSTM Cell

Sequence Processing

- Given a sequence of vectors $\mathbf{x} \in \mathbf{R}^{[D * T]}$
- D - a single vector dimensionality
- T - number of elements in the sequence
- Usual Neural Networks can’t process a sequence, they work with all elements all together



Recurrent Neural Nets

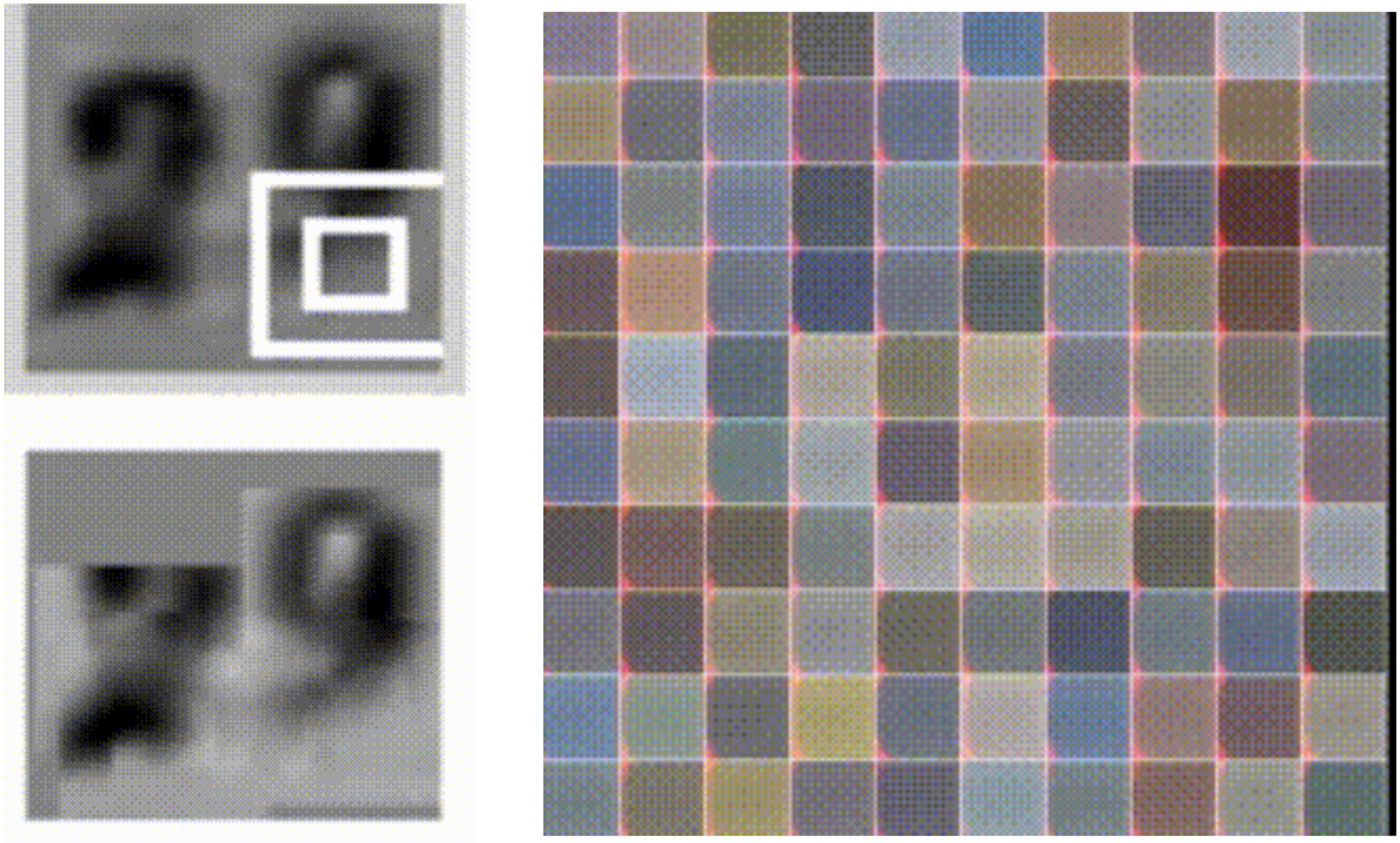
- Recurrent Neural Nets (RNN) - a class of neural networks, designated for sequence processing

- A network with a feedback loop, with connections between layers making a directed loop

- They preserve an internal state, capable of updating while processing the data sequence

Application Areas

- RNN are widely used in the tasks related to temporal changes or other dynamic processes
  - time series classification and prediction
  - speech and video recognition
  - machine translation
  - various NLP tasks



Left: RNN learns to read house numbers. Right: RNN learns to paint house numbers

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

NN Schemes

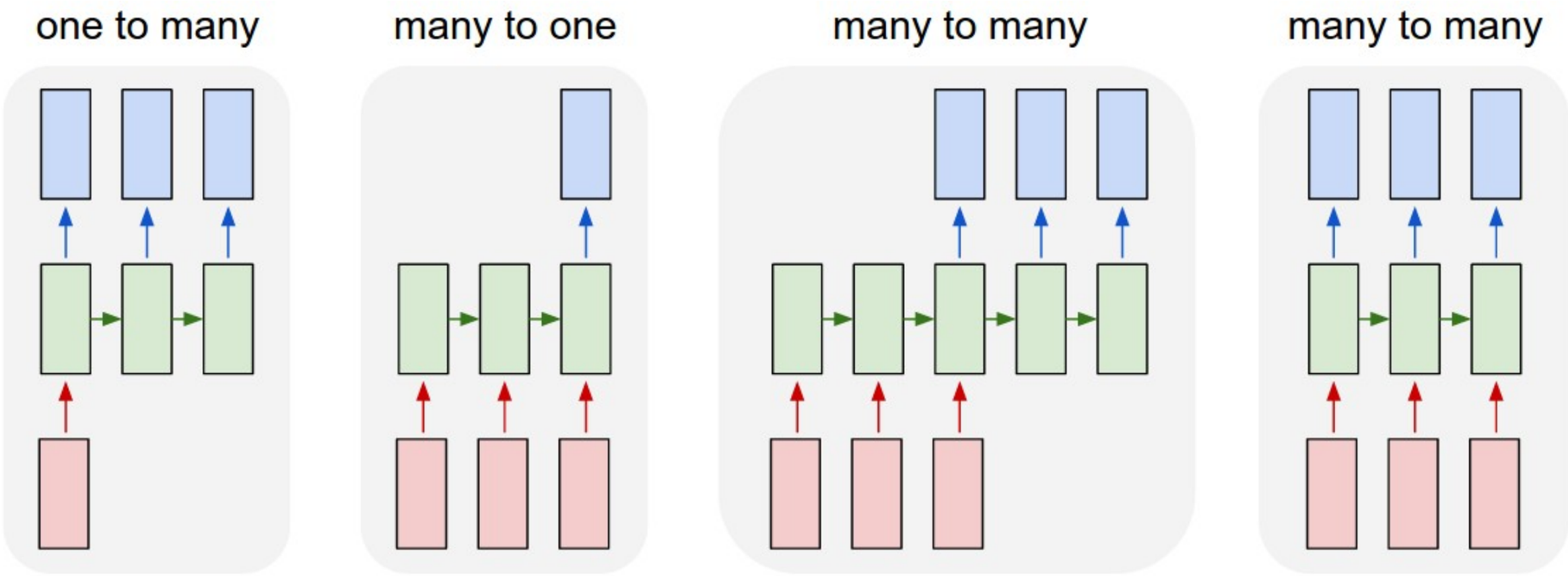


Image Annotation
(image -> token sequence)

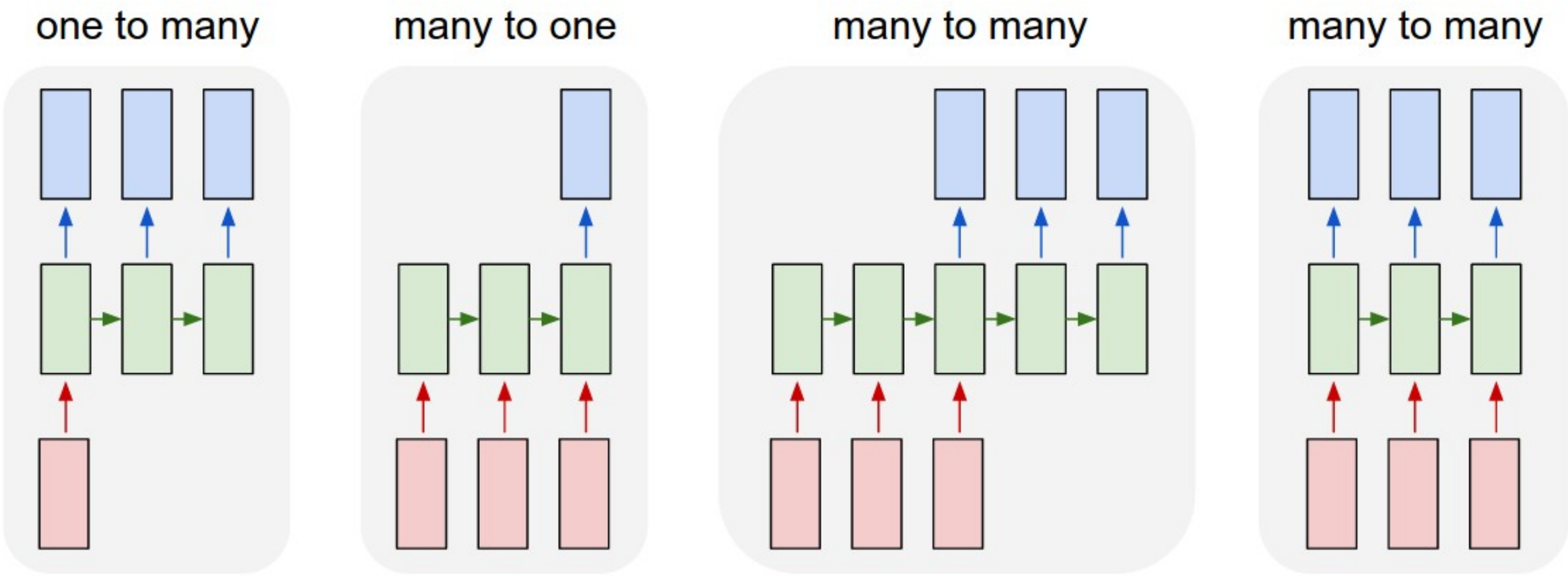
A person on a beach flying a kite.

A black and white photo of a train on a train track.

A person skiing down a snow covered slope.

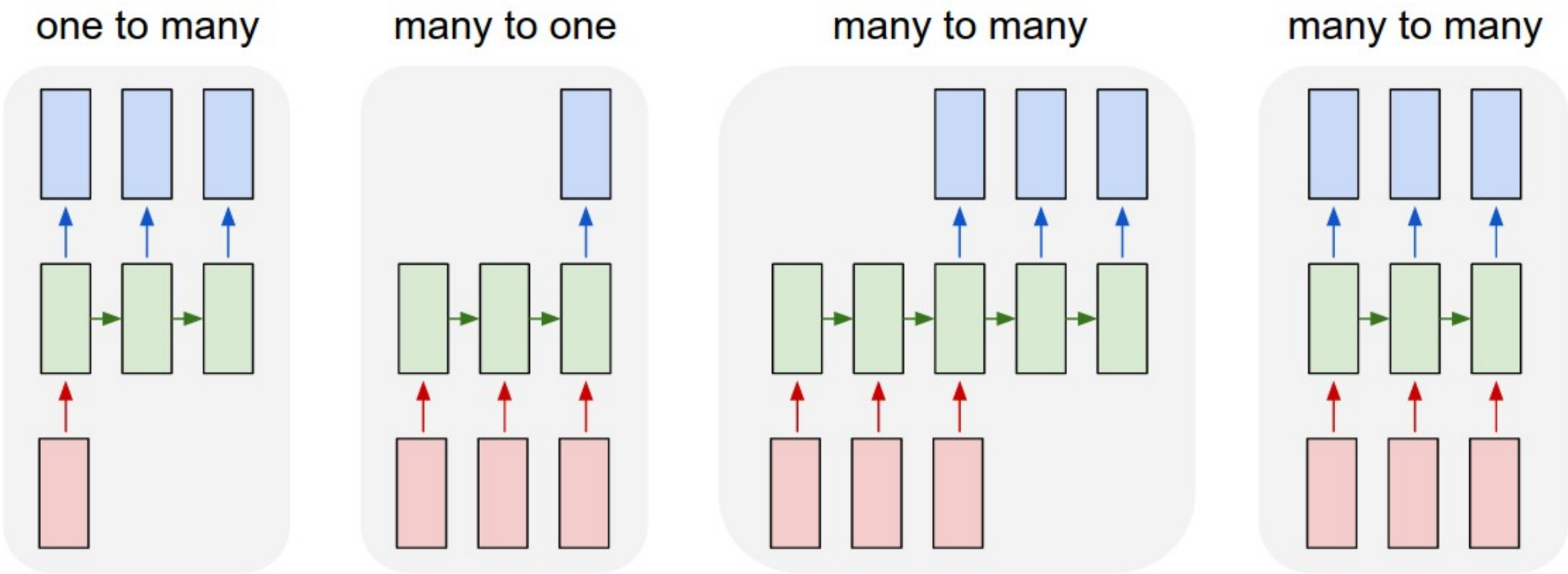
A group of giraffe standing next to each other.

NN Schemes

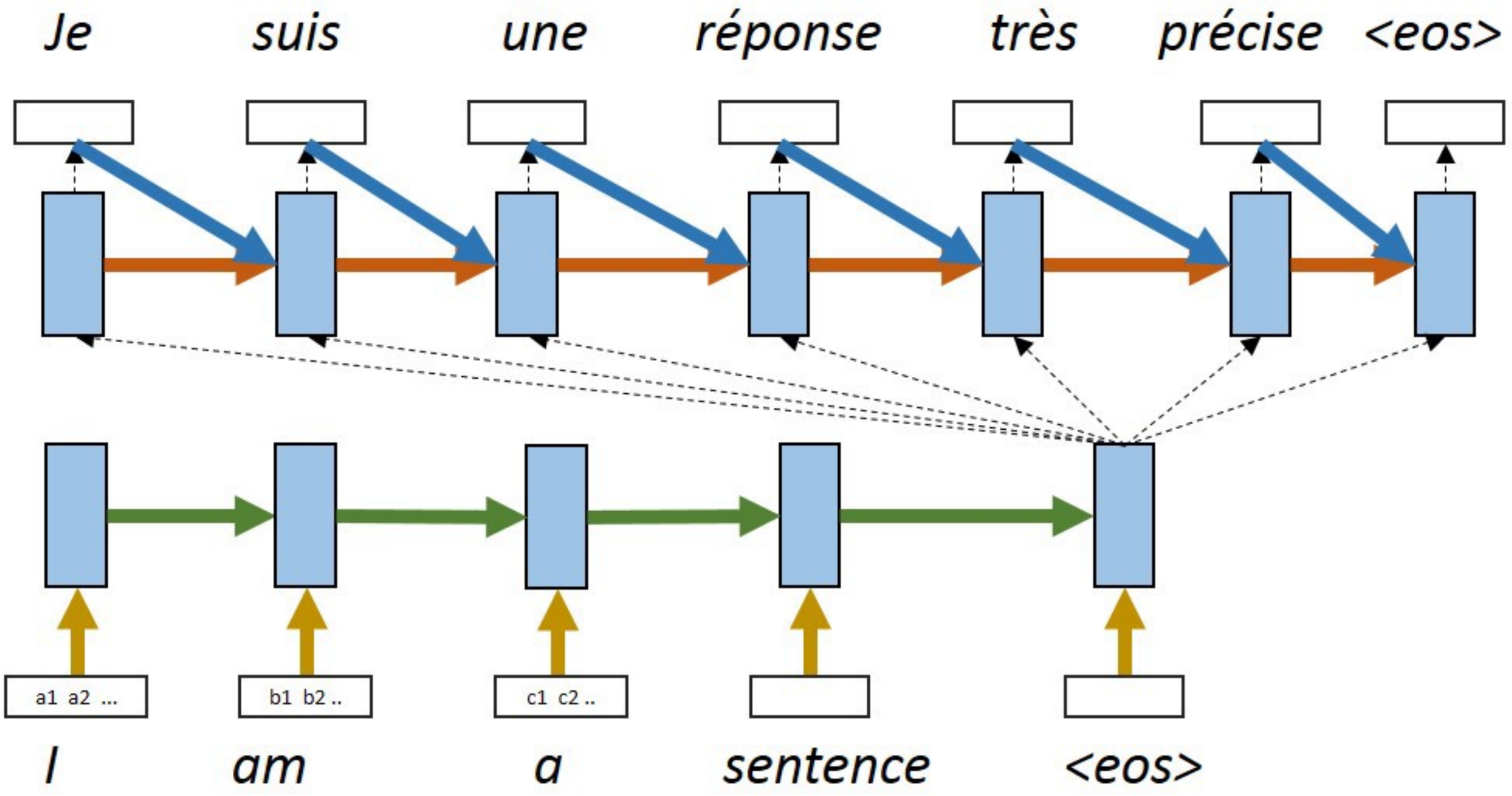


Sentiment Analysis
(token sequence -> sentiment estimate)

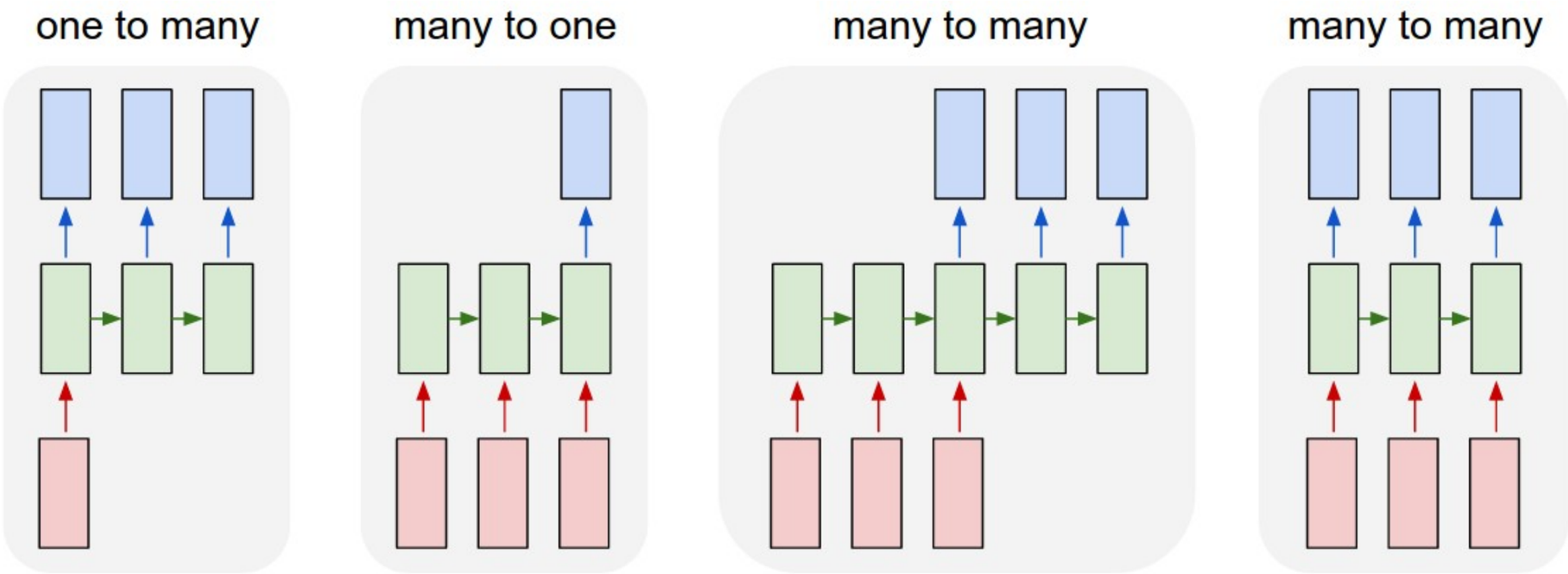
NN Schemes



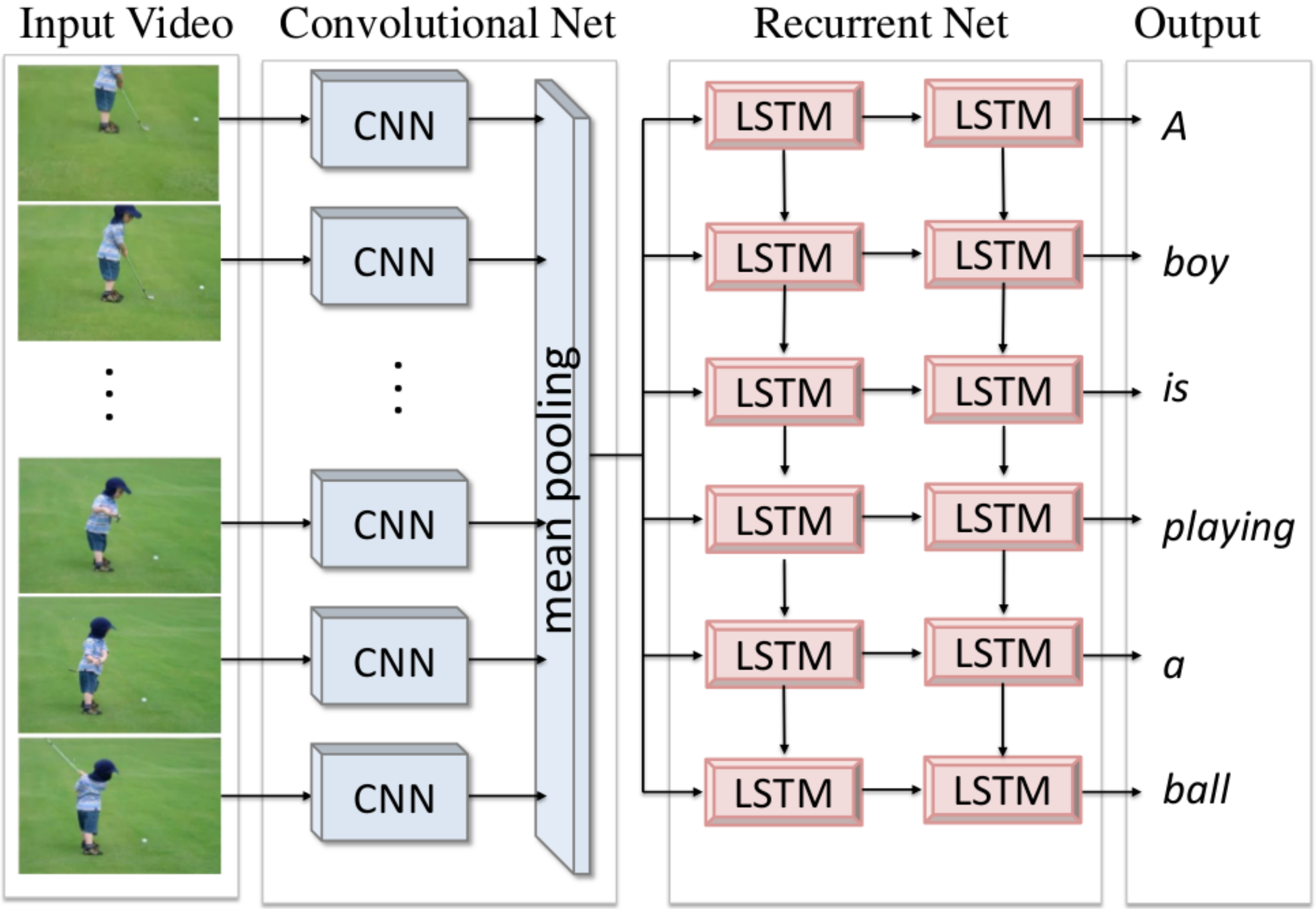
Machine Translation
token sequence->token sequence


Je
suis
une
réponse
très
précise
<eos>
a1 a2 ...
b1 b2 ..
c1 c2 ..
I
am
a
sentence
<eos>

NN Schemes



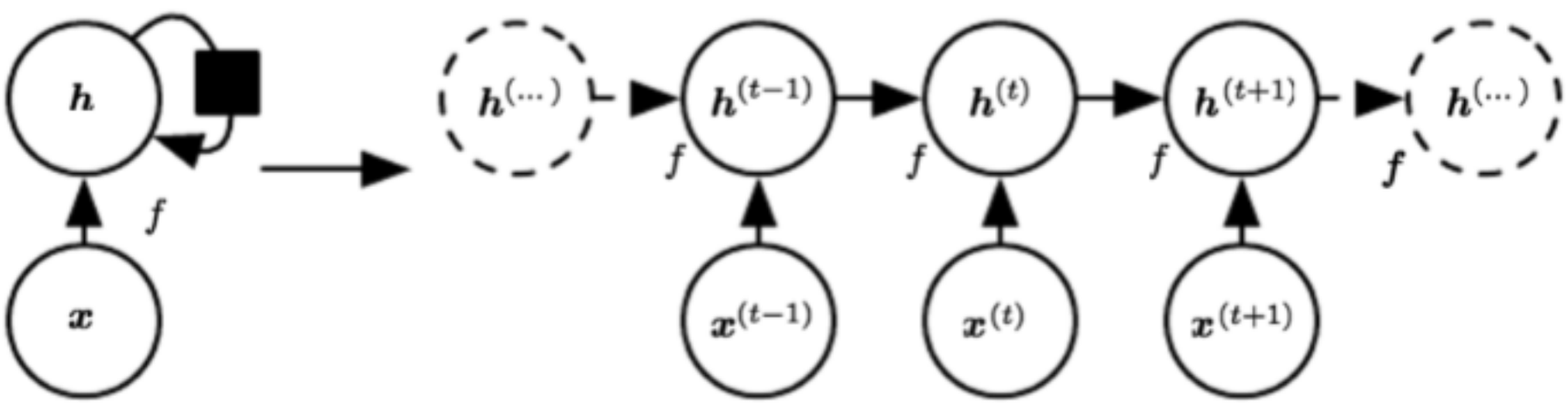
Video Annotation
(Sequence of images -> sequence of tokens)


Input Video
Convolutional Net
Recurrent Net
Output
CNN
CNN
CNN
CNN
CNN
mean pooling
LSTM
A
boy
is
playing
a
ball

Principal Scheme

- The main sequence is **x**
- $\mathbf{x}^t$ - t element of a sequence
- $\mathbf{h}^t$ - hidden RNN state when processing $\mathbf{x}^t$
- **Θ** - all the paramaters
- The hidden state update is calculated with a recurrent formula

$$h^{(t)} = f(h^{(t-1)}, \theta, x^{(t)})$$



Recurrent formula

Non-linearity of W parameters

Input vector at timestamp t

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

The new state

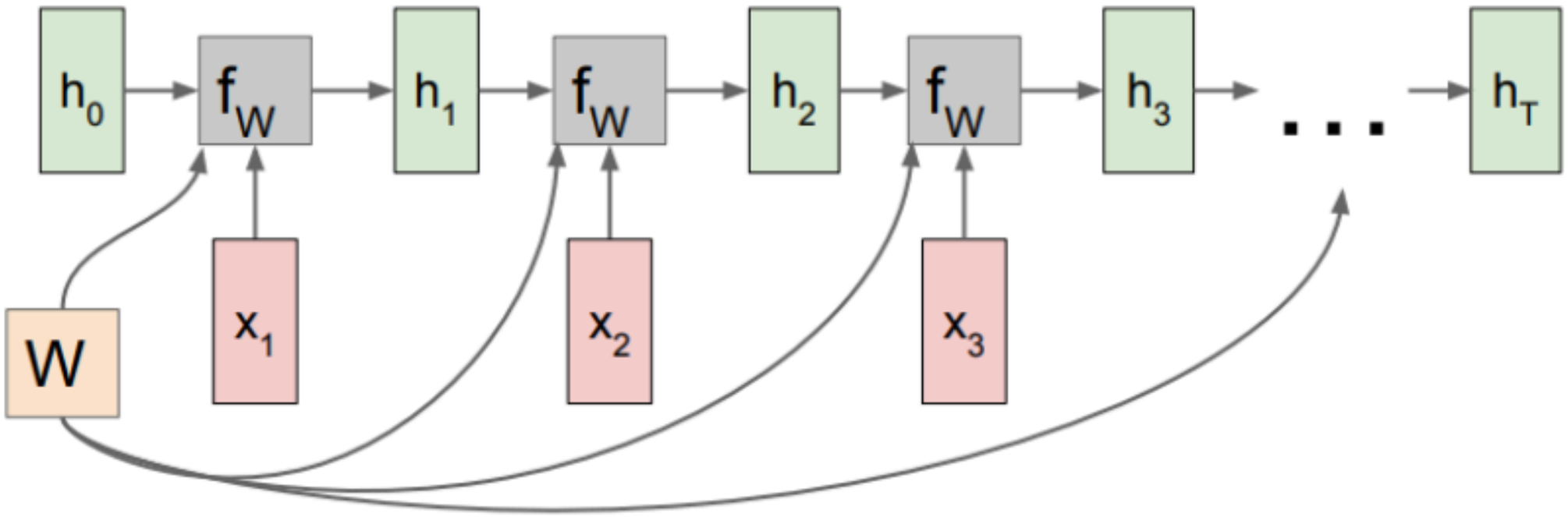
The previous state

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

- It can handle a sequence of any length, applying the formula for each input
- **IMPORTANT:** each function with the same weights is applied for every input vector

RNN Computational Graph



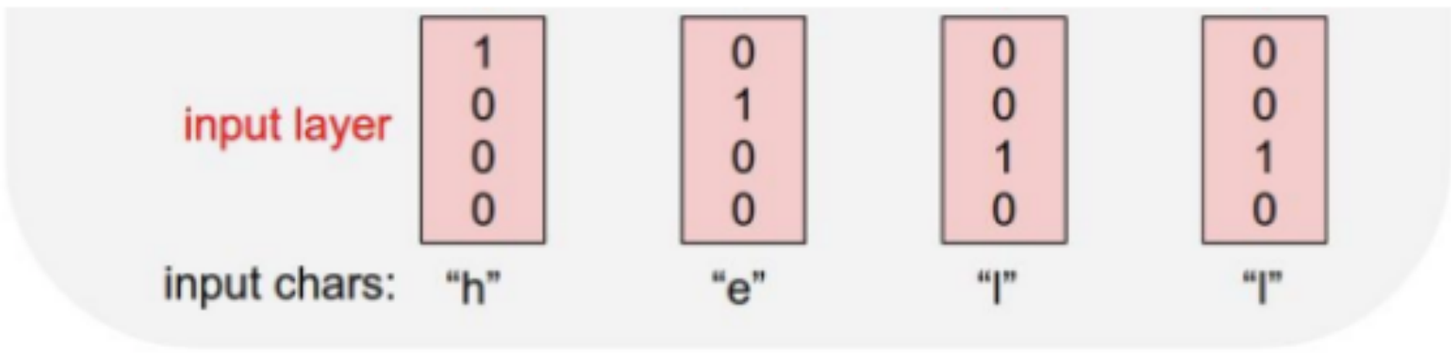
- The same set of parameters **W** is reused on each step t
- This lets RNN effectively generalise on a variable length sequences

Example: character language model

- The goal: to predict next letter/character

- Training sequence: **‘hello’**

- Vocabulary: [h,e,l,o]

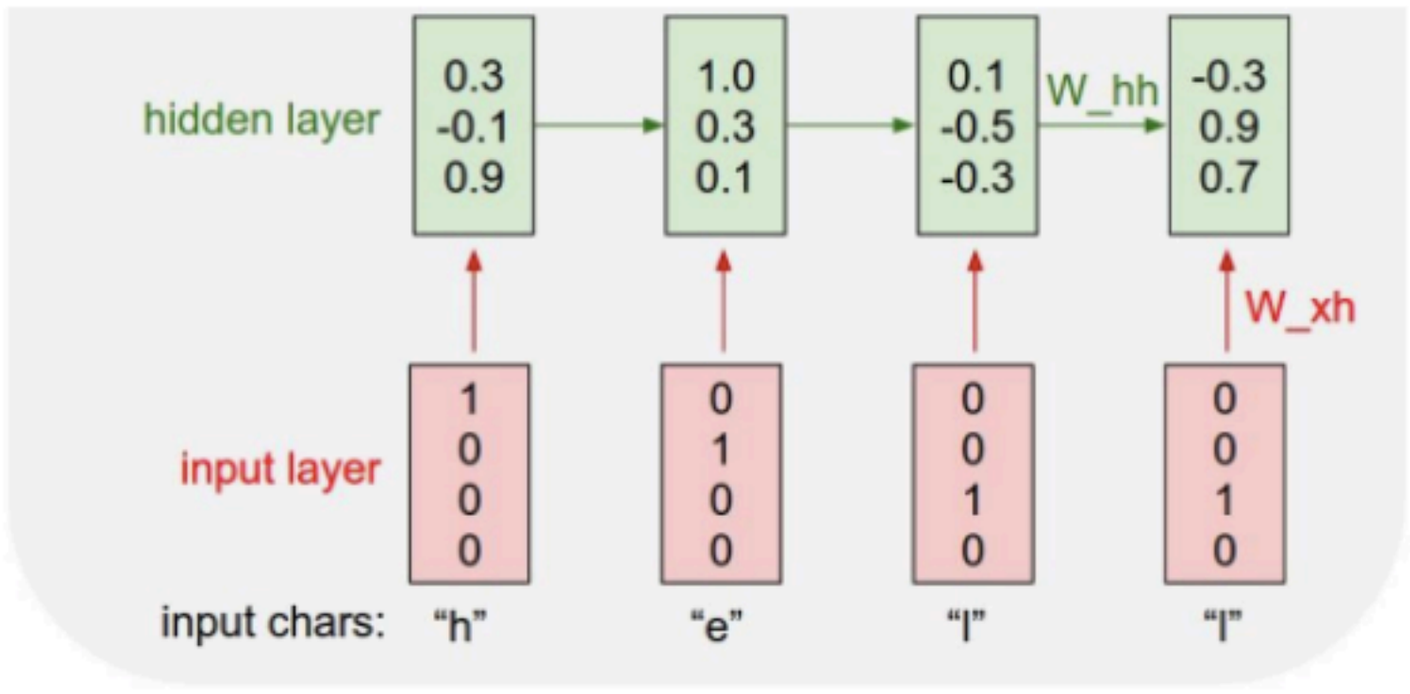


Example: character language model

$$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

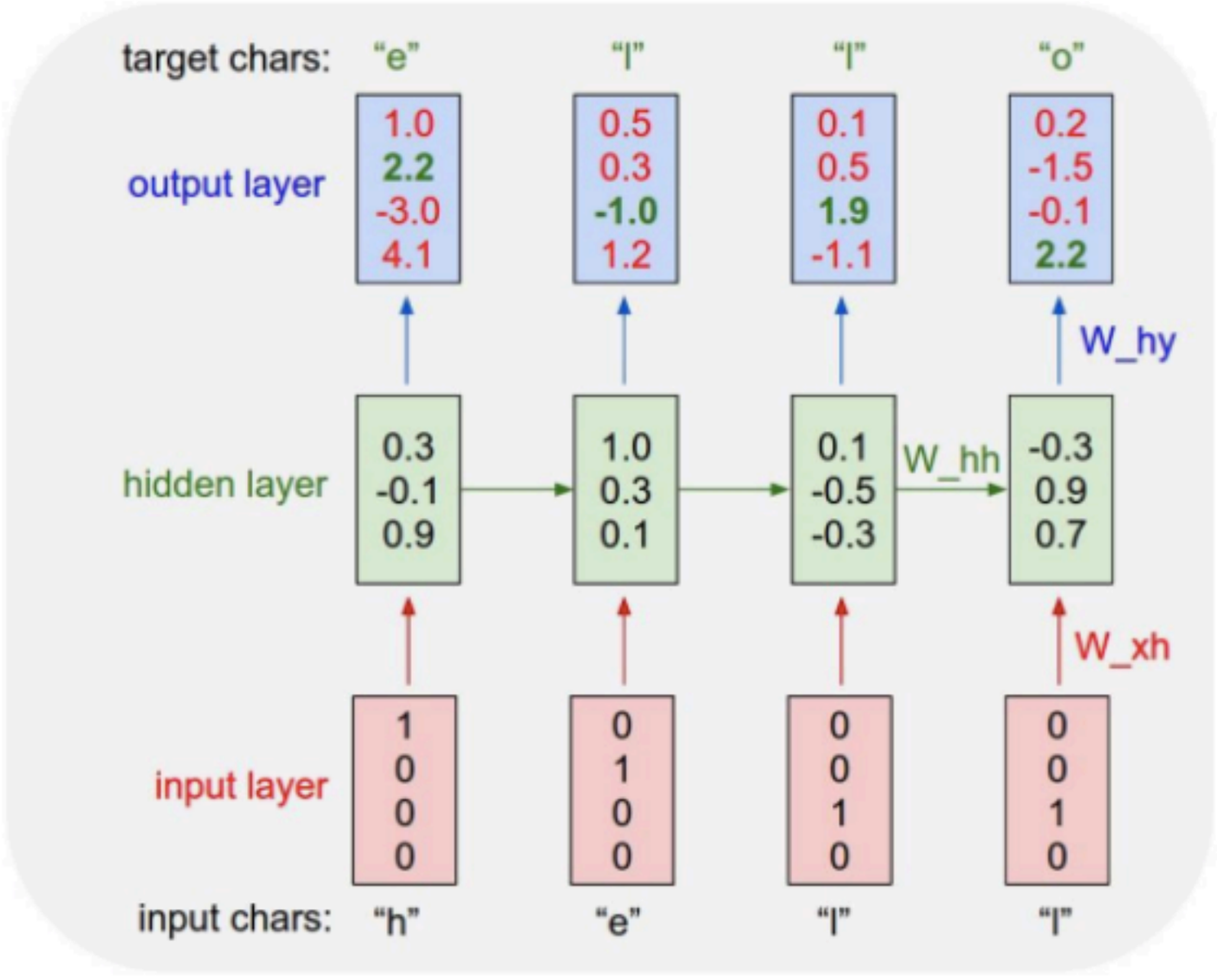
- Training sequence:
  ‘hello’

- Vocabulary:
  [h,e,l,o]



Example: character language model

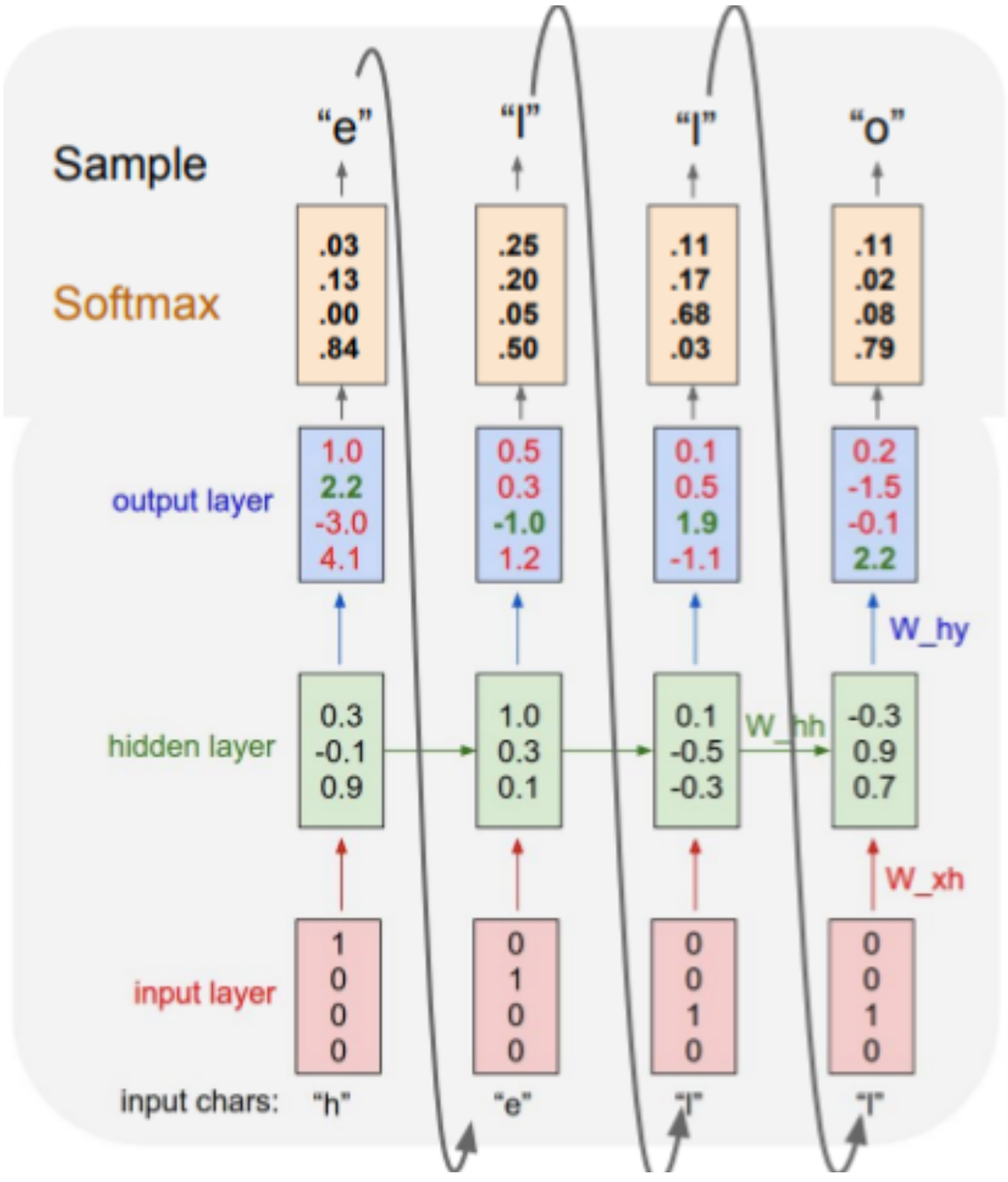
- Training sequence: **‘hello’**
- Vocabulary: [h,e,l,o]
- Input: [h, e, l, l]
- Targets: [e, l, l, o]



Example: text generation

- Step by step, we generate the probable next characters

- And we pass this new character as an input to the next cell



Training a RNN Language Model

- Get a big corpus of text which is a sequence of words $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$
- Feed into RNN-LM; compute output distribution $\hat{\boldsymbol{y}}^{(t)}$ for *every step t.*
  - i.e. predict probability dist of *every word*, given words so far

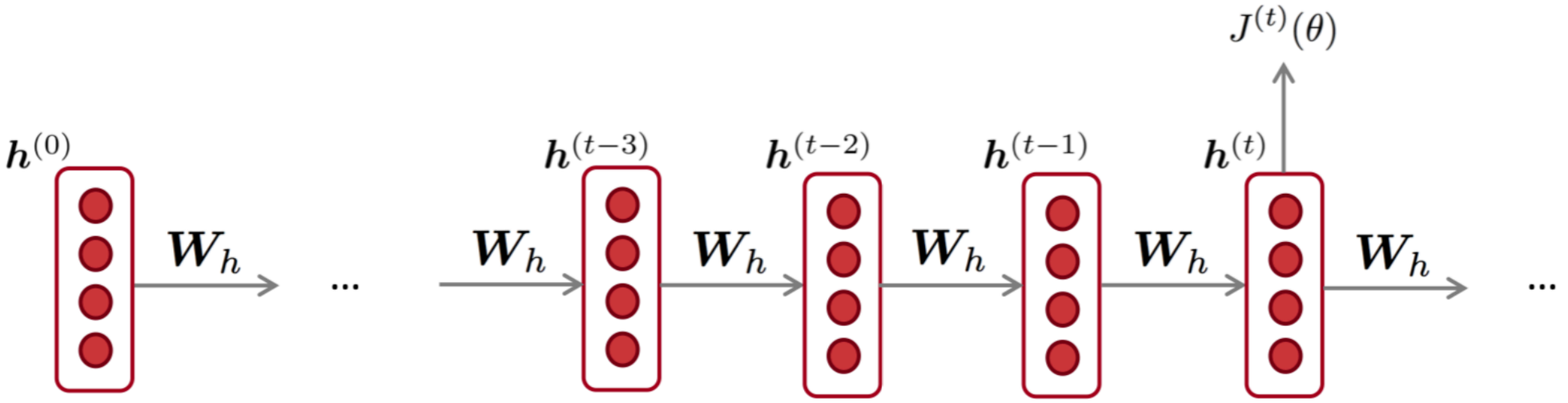
- Loss function on step *t* is usual cross-entropy between our predicted probability distribution $\hat{\boldsymbol{y}}^{(t)}$, and the true next word $\boldsymbol{y}^{(t)} = \boldsymbol{x}^{(t+1)}$:

$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{j=1}^{|V|} y_j^{(t)} \log \hat{y}_j^{(t)}$$

- Average this to get overall loss for entire training set:

$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta)$$

Backpropagation for RNNs



Question: What's the derivative of $J^{(t)}(\theta)$ w.r.t. the repeated weight matrix $\boldsymbol{W}_h$?

Answer: $$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}}\bigg|_{(i)}$$

"The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears"

Optimisation problems

- Let’s check a simple RNN model with a linear dependency between states

$$h^{(t)} = W^T h^{(t-1)}$$

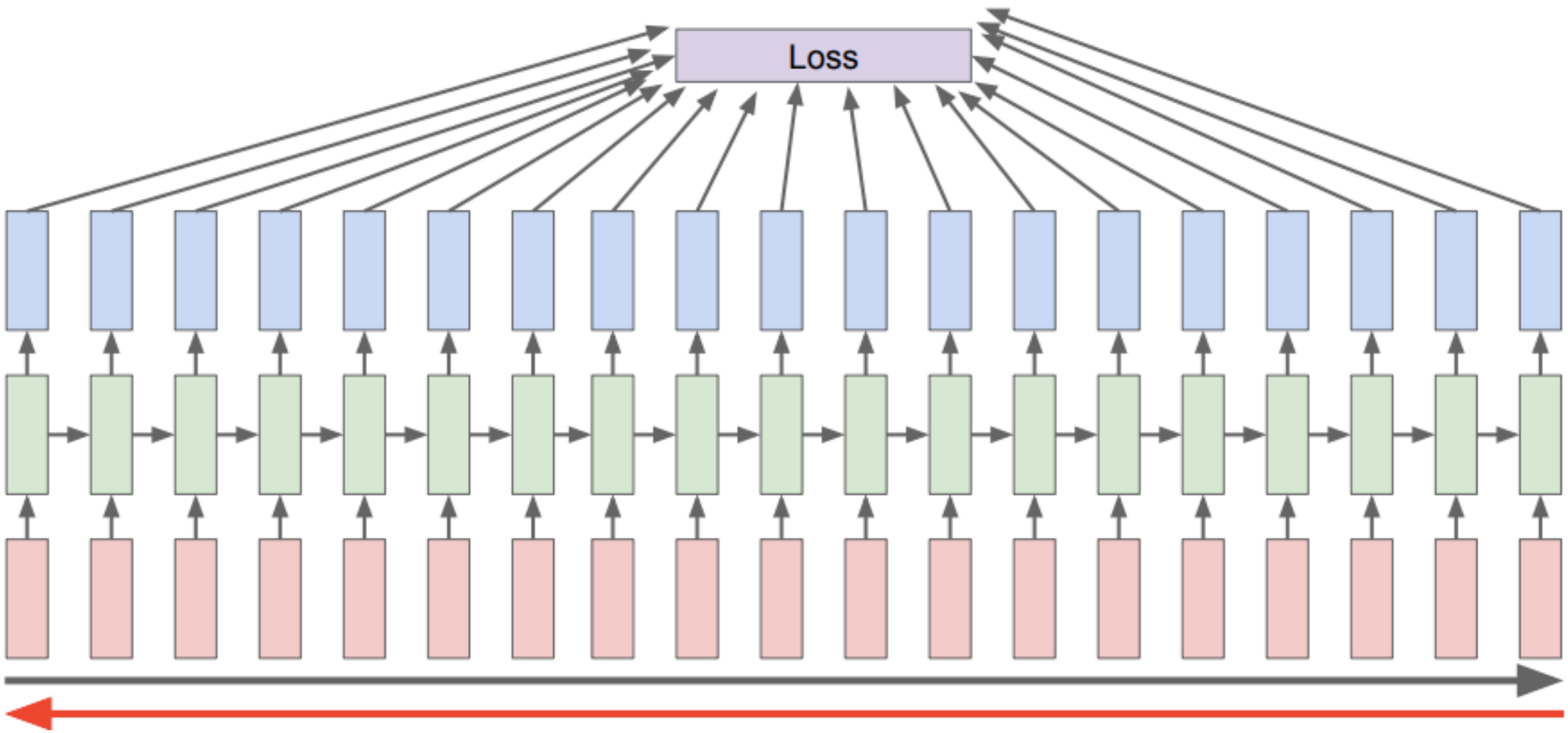
- We can state the step h^t through h^0

$$h^{(t)} = (W^t)^T h^{(0)}$$

- Wafter applying the eigenvalue decomposition of W we get the following

$$h^{(t)} = Q^T \Lambda^t Q h^{(0)}$$

Backprop through RNN



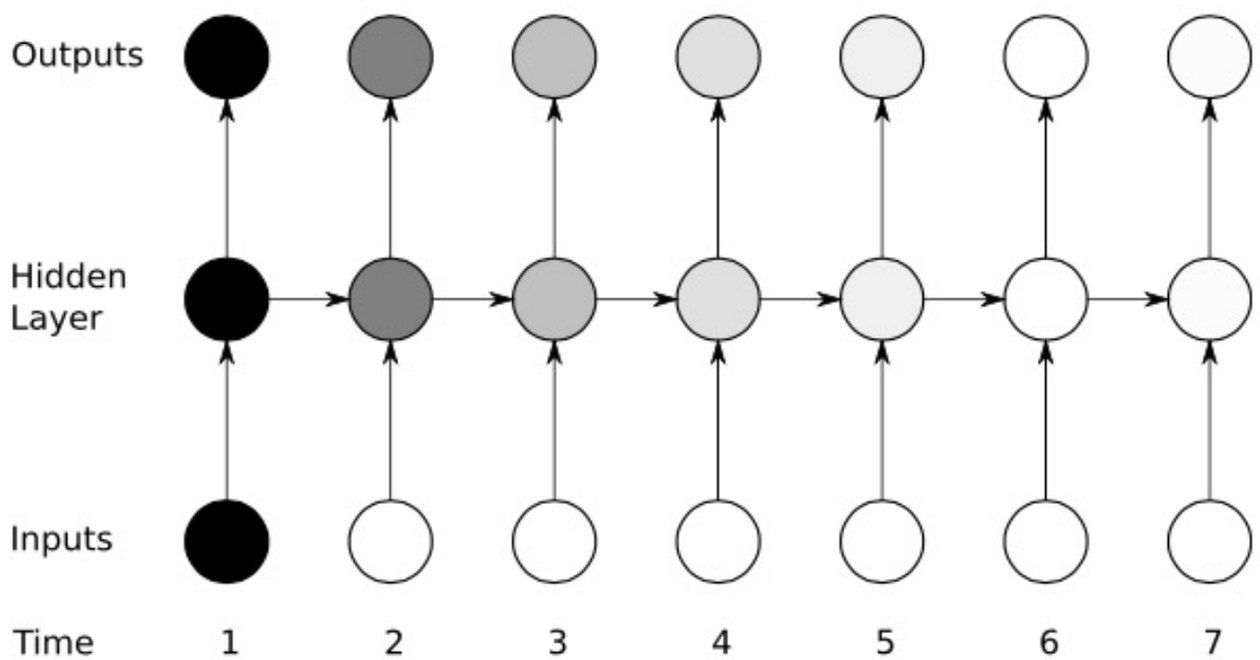
- RNN parameters optimisation — a computationally complex task, as we have to apply the backpropogation of error recursively for each node

Vanishing Gradient

$$h^{(t)} = Q^T \Lambda^t Q h^{(0)}$$

- When put into t power, собственные eigenvalues Λ<1 vanish (limit to zero).

- We would like the error on a step t to flow backward and update the weights based on the input of a far away step



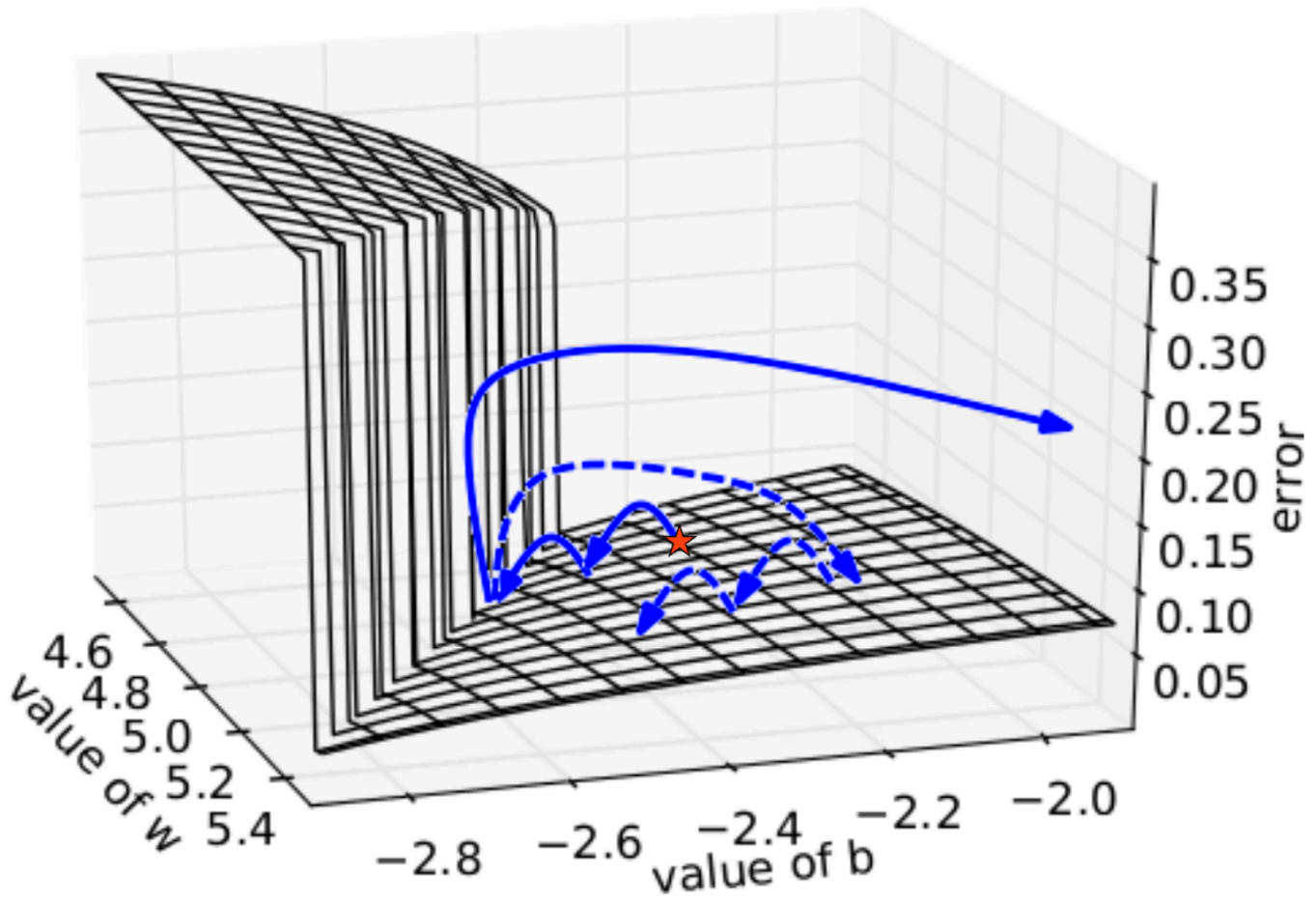
Gradient Explosion

- When put into t power, eigenvalues Λ>1 grow uncontrollably (limit to ∞).

$$h^{(t)} = Q^T \Lambda^t Q h^{(0)}$$

- It is more rare, than vanishing, but has a much worse impact on the optimisation process

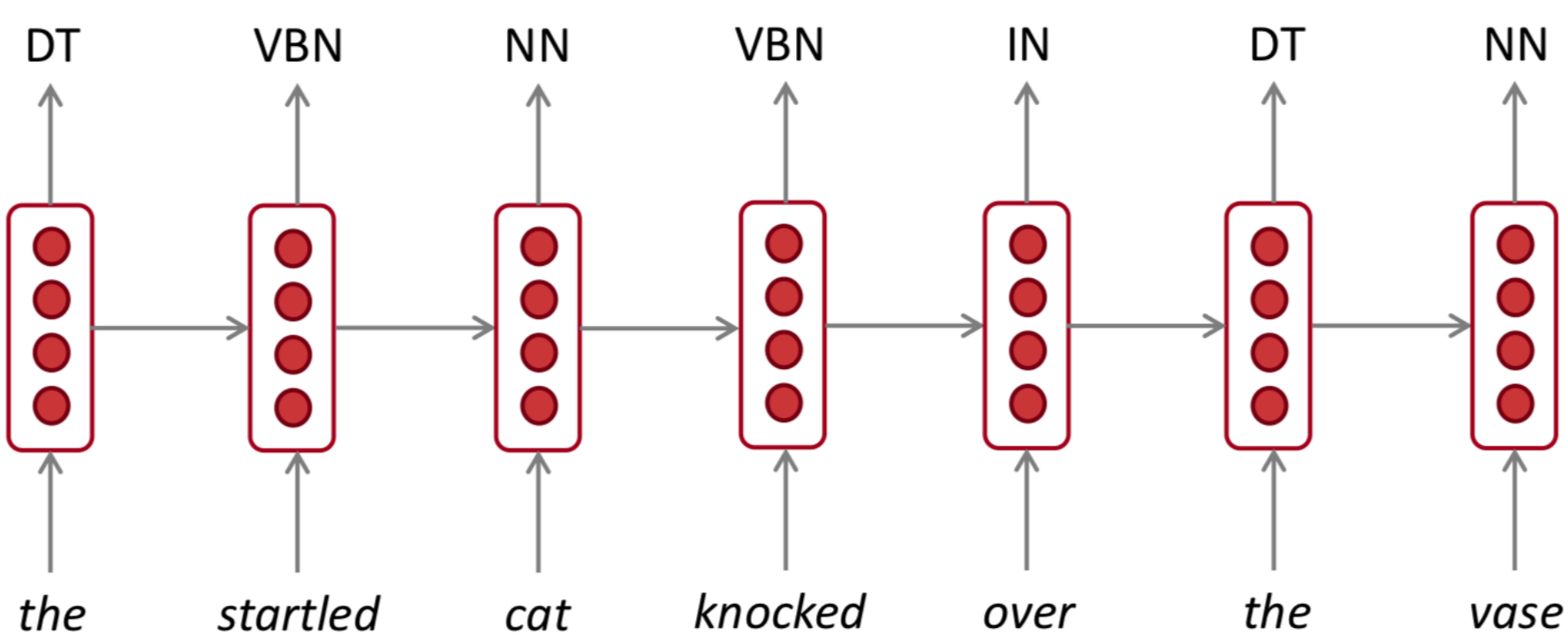
- Solution: clipping. Set the norm of the gradient to some max value



More Detailed Applications

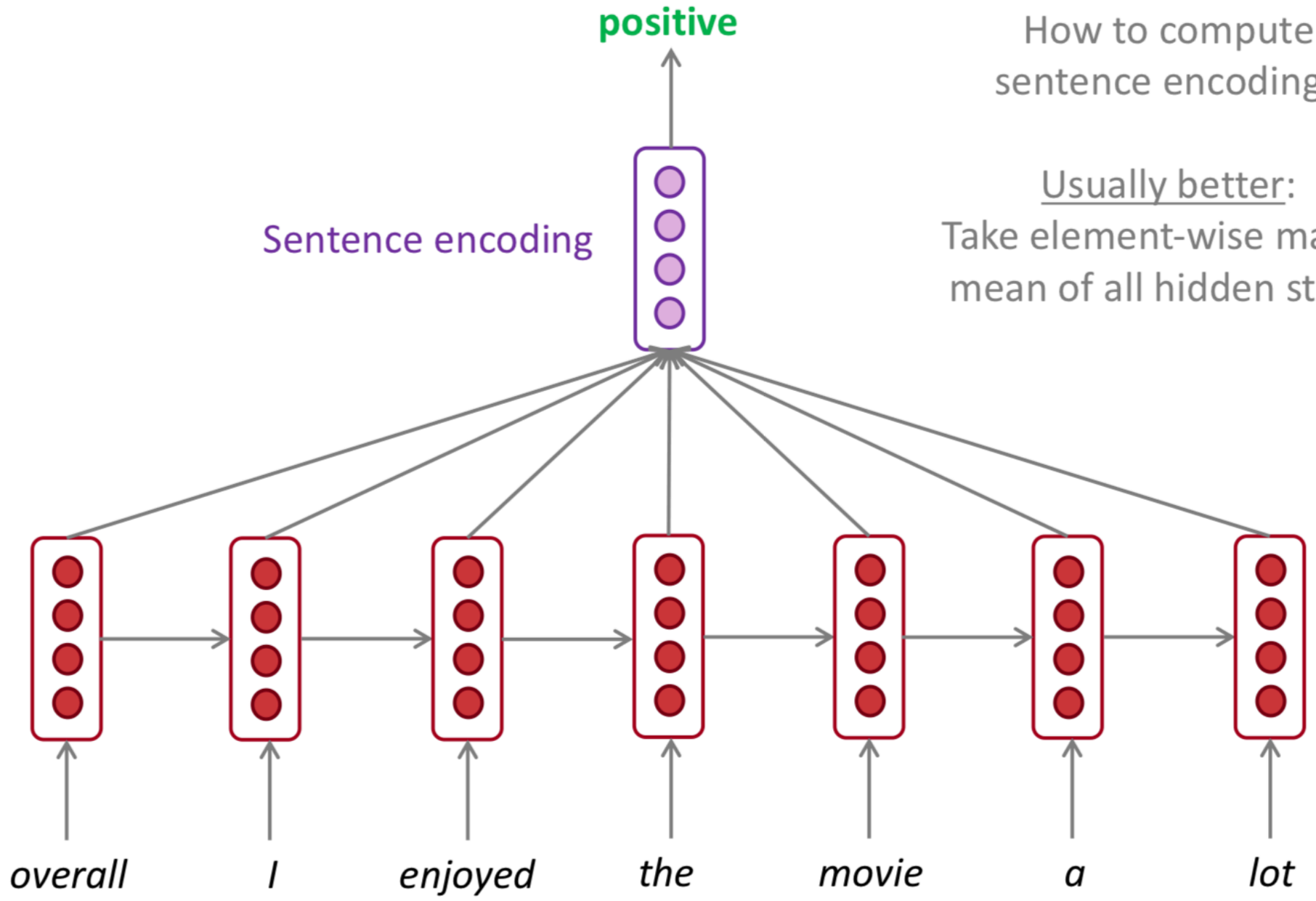
RNNs can be used for tagging

e.g. part-of-speech tagging, named entity recognition



RNNs can be used for sentence classification

e.g. sentiment classification

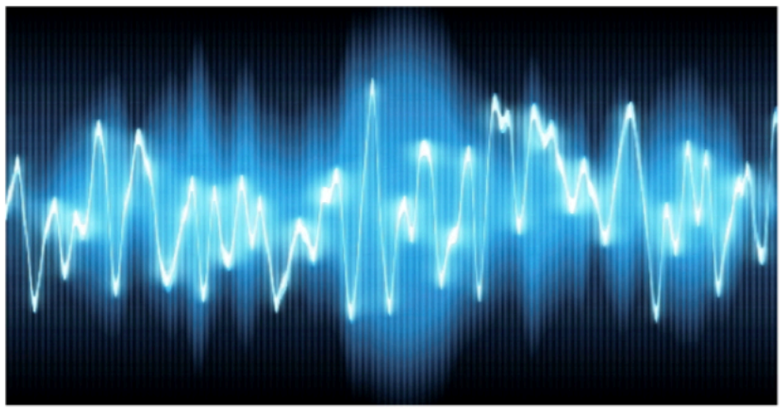


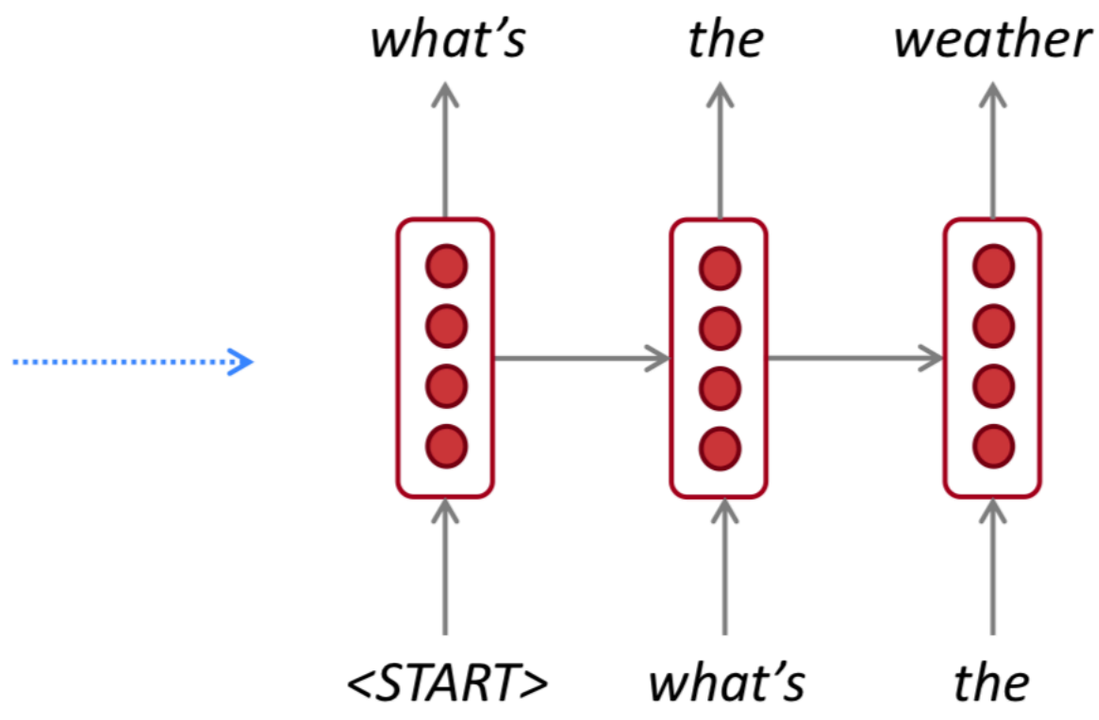
How to compute sentence encoding?

Usually better:
Take element-wise max or mean of all hidden states

RNNs can be used to generate text

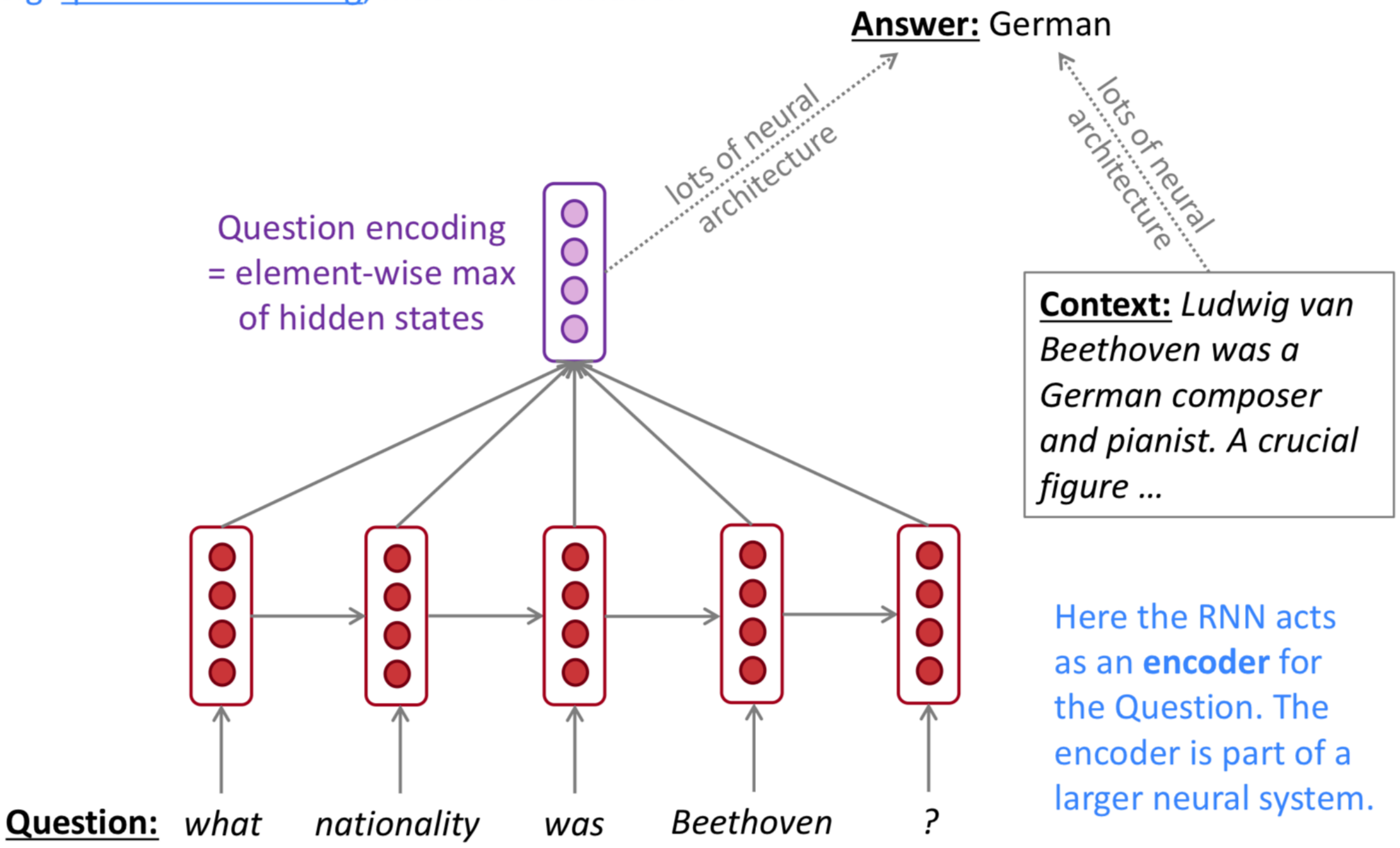
e.g. speech recognition, machine translation, summarization





RNNs can be used as an encoder module

e.g. question answering, machine translation



Here the RNN acts as an **encoder** for the Question. The encoder is part of a larger neural system.

LSTM Cell

- Variation RNN, optimised for vanishing gradient problem solution

- The principal model was proposed in a paper by Hochreiter & Schmidhuber (1997)

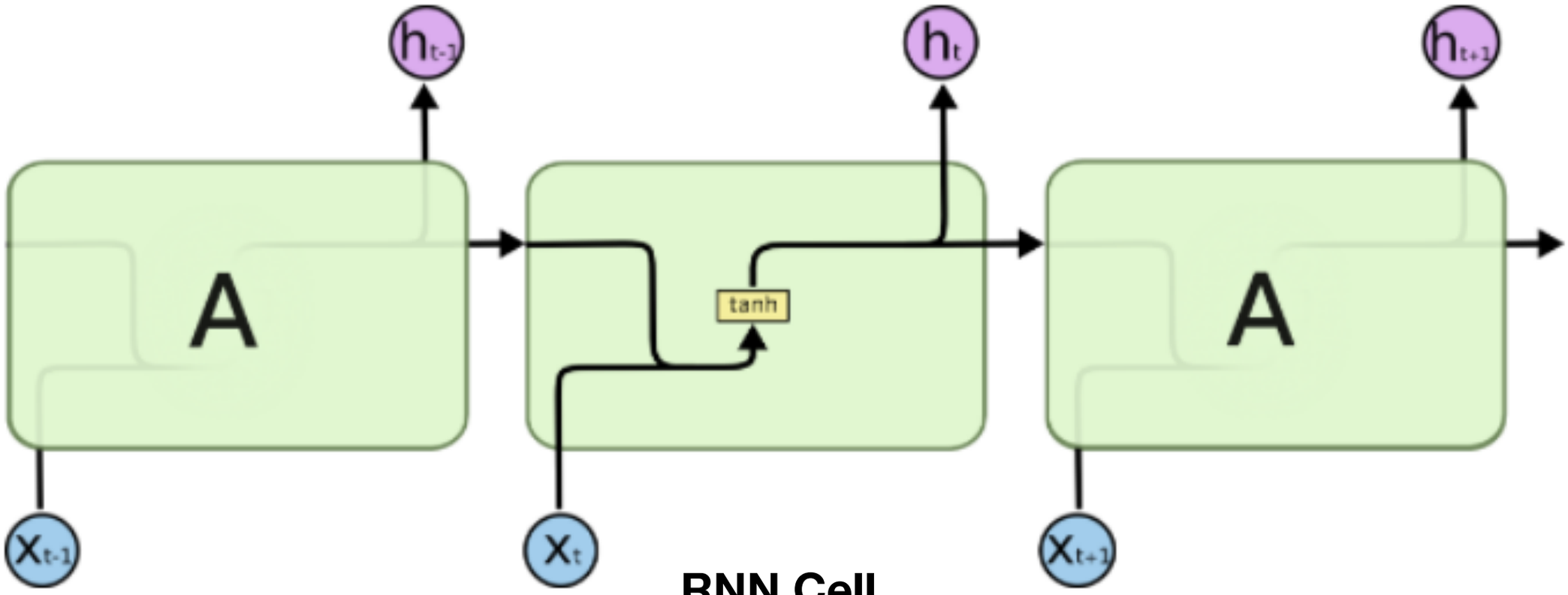
- Main Improvement - new mechanism of a loop, that helps gradient to stay for a long time, without vanishing or explosion

- LSTM cell also has a few gates, controlling information flow inside the cell

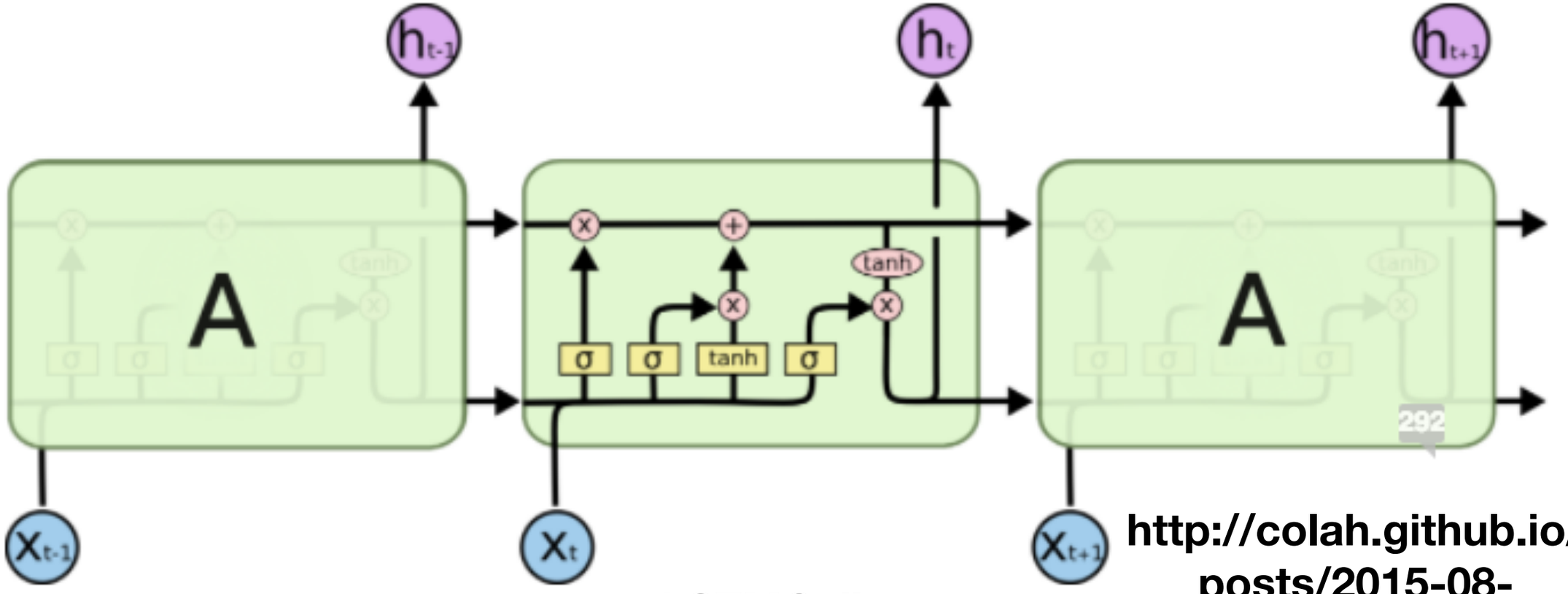
- Currently, the most widespread and stable RNN model

Long-short-term-memories (LSTMs)

- Allow each time step to modify
  - Input gate (current cell matters) $i_t = \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1}\right)$
  - Forget (gate 0, forget past) $f_t = \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1}\right)$
  - Output (how much cell is exposed) $o_t = \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1}\right)$
  - New memory cell $\tilde{c}_t = \tanh\left(W^{(c)}x_t + U^{(c)}h_{t-1}\right)$
- Final memory cell: $c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$
- Final hidden state: $h_t = o_t \circ \tanh(c_t)$



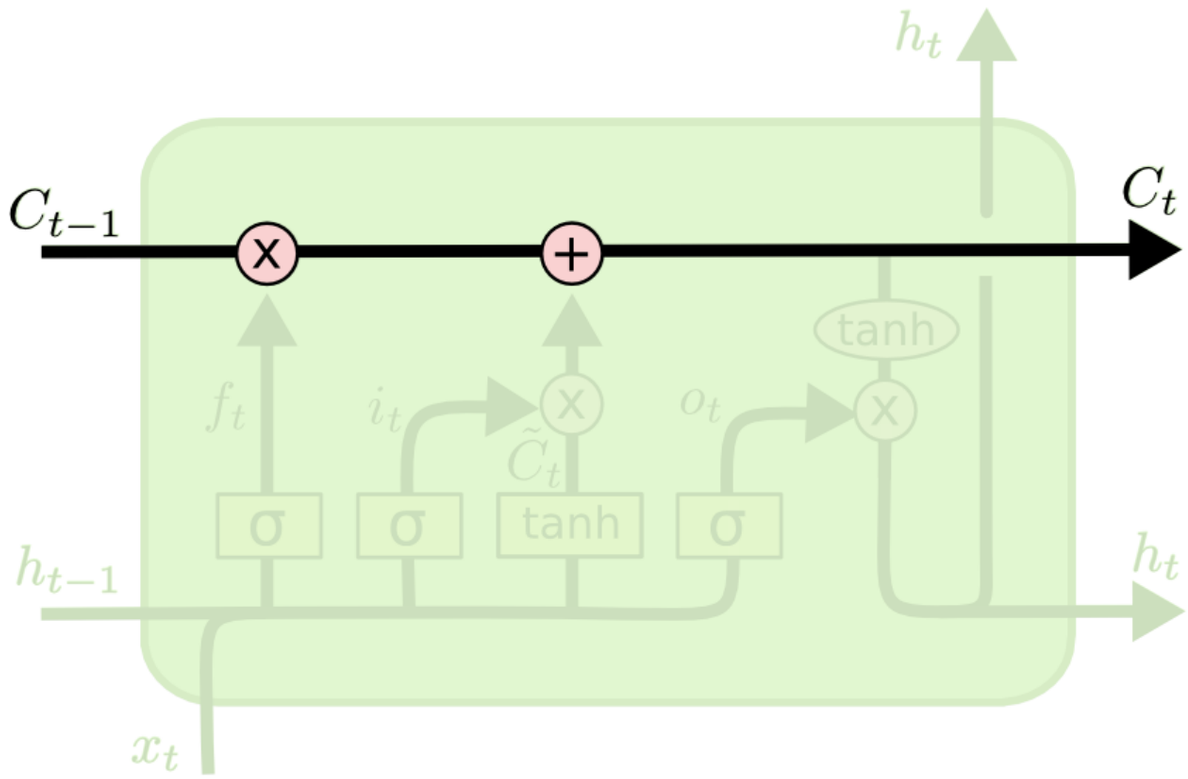
RNN Cell

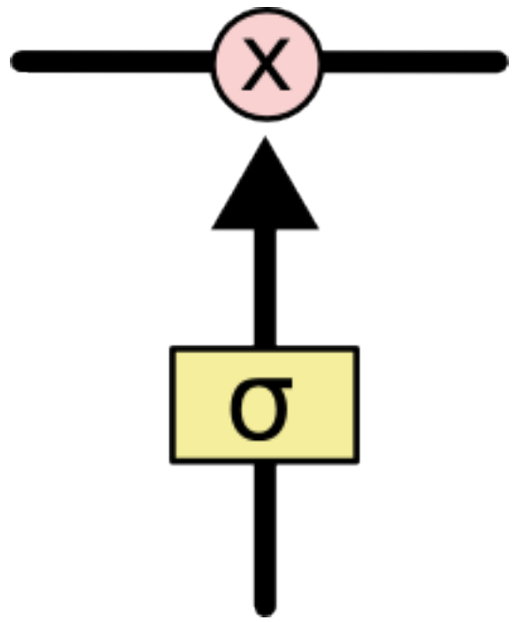


LSTMCell

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

The cell state





Neural Network Layer

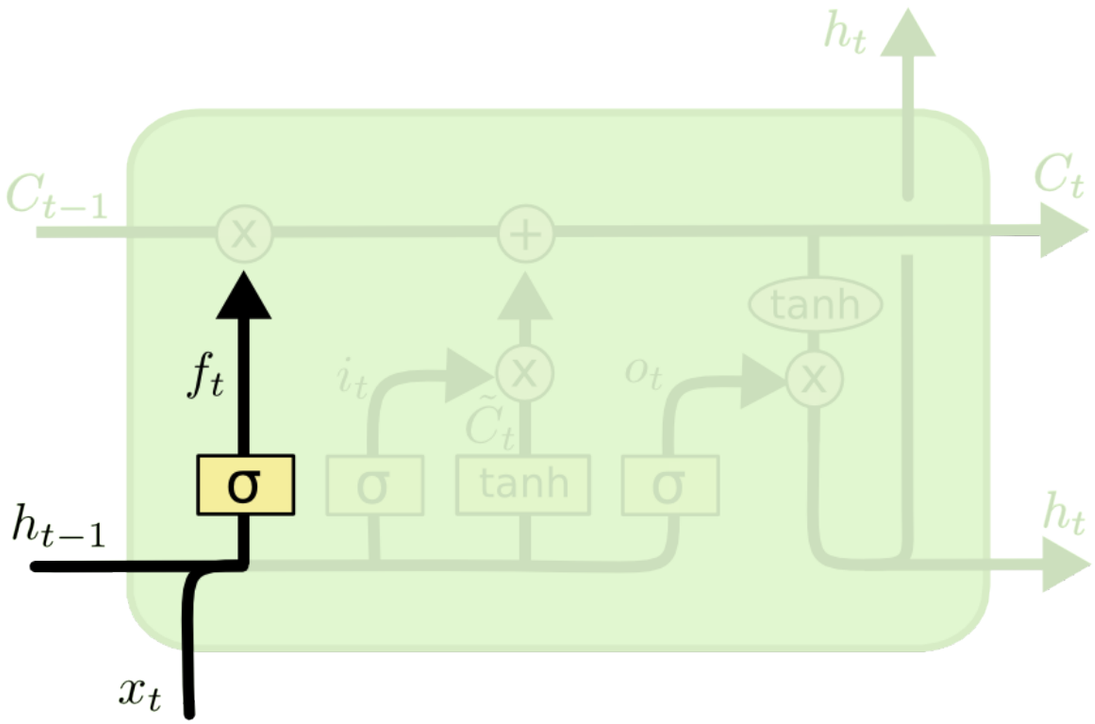
Pointwise Operation

Vector Transfer

Concatenate

Copy

Step 1: Forget gate



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

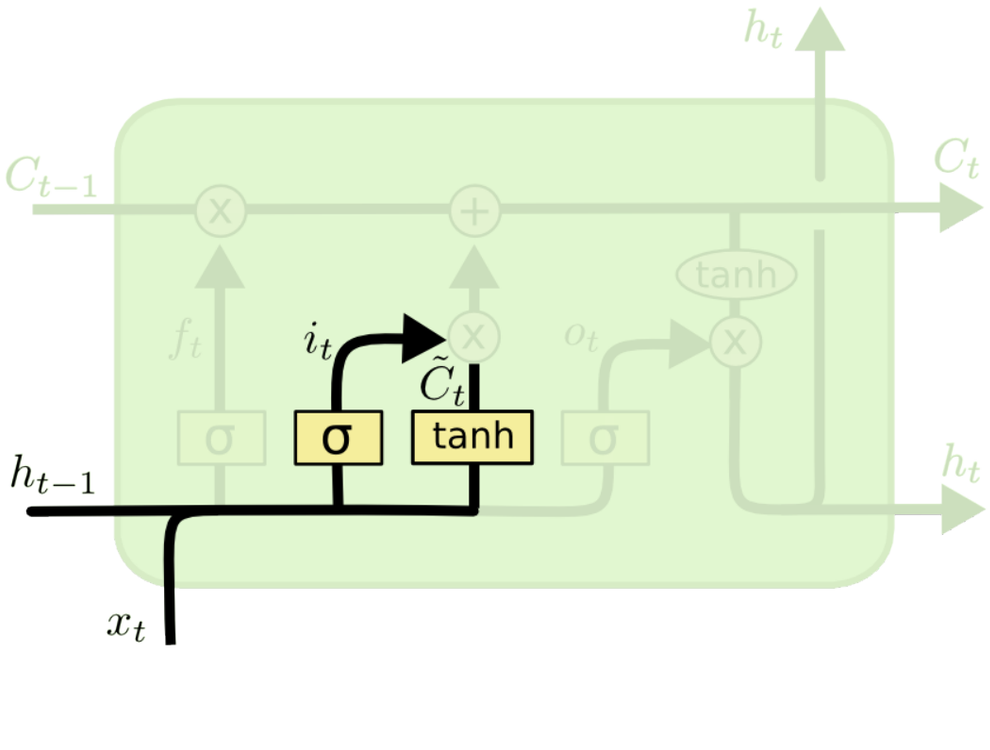
Decides what information we're going to throw away from the cell state.

Looks at h_{t-1}, current x_t and outputs a number between 0 and 1 for each number in the cell state.

0 - completely get rid of this

1 - completely keep this

Step 2: Input gate



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

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

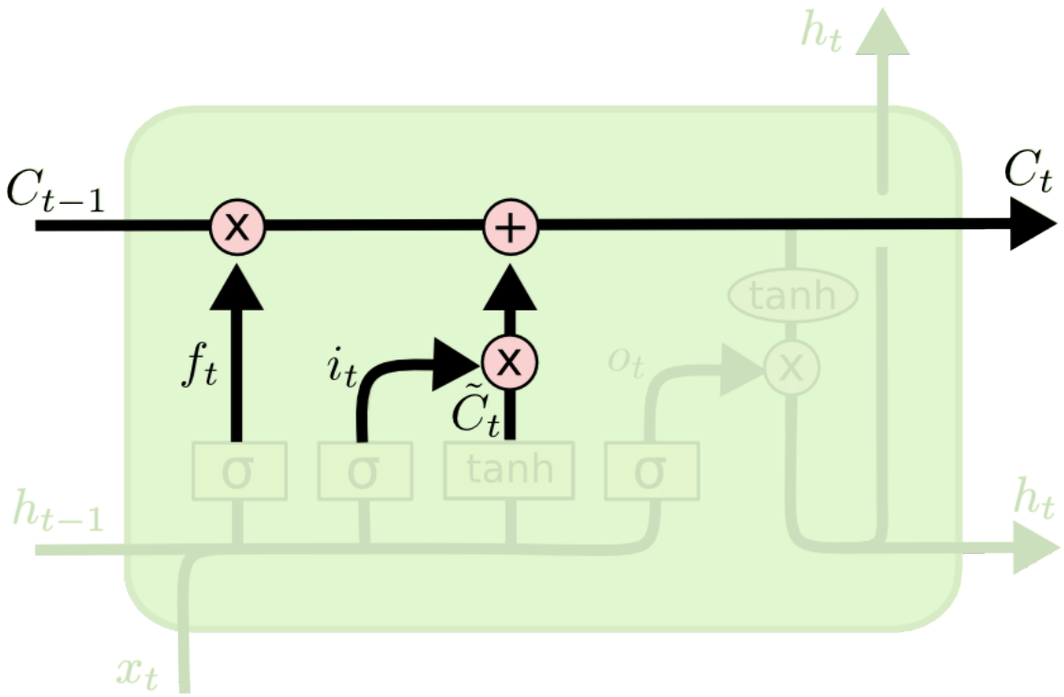
Decides what new information we're going to store in the cell state and what we won't.

First, a **sigmoid layer** called decides which values we'll update.

Next, a ***tanh*** layer creates a vector of new candidate values $\tilde{C}_t$ based on the x_t

We'll combine them in the next step to update the state

Step 3: State update

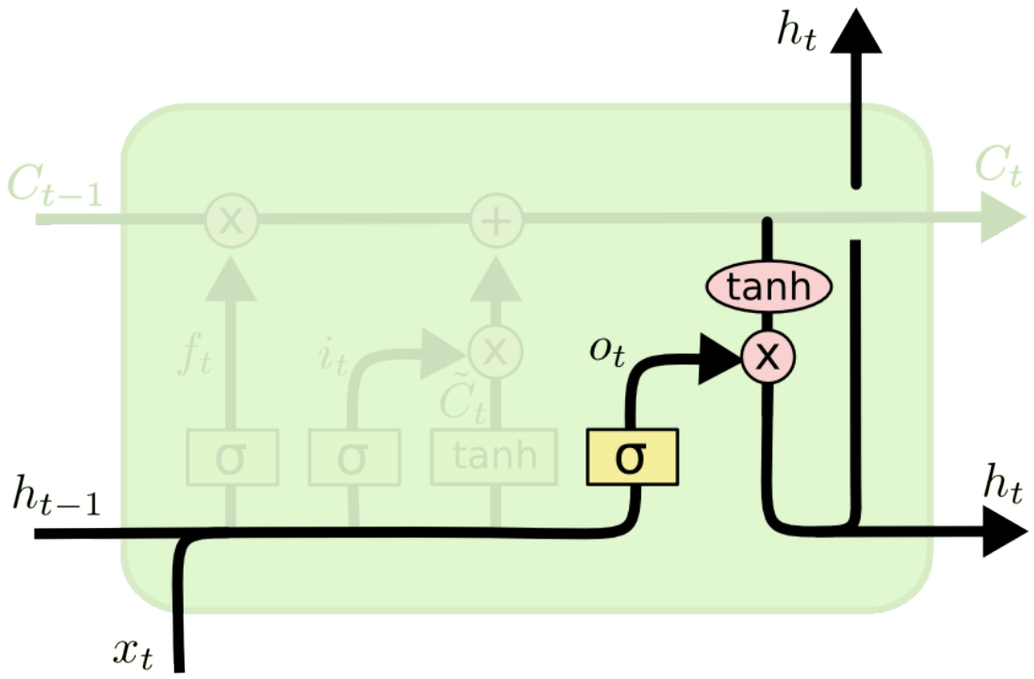


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

We've got the new state by using Step 1 and Step 2 results:

We use previous state, C_{t-1} multiplied by ***forget gate***, and current candidate state $\tilde{C}_t$ multiplied by ***input gate***

Step 4: Output gate



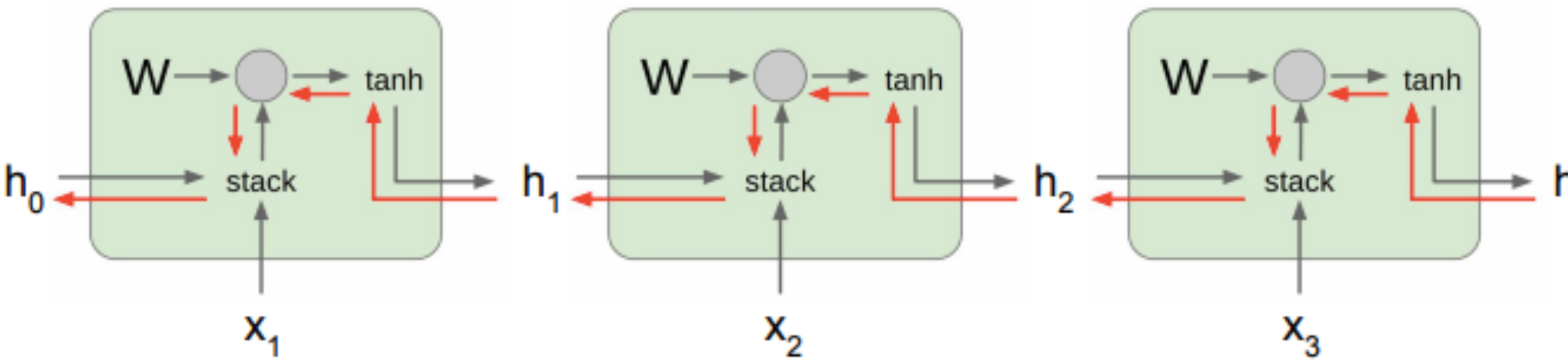
$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

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

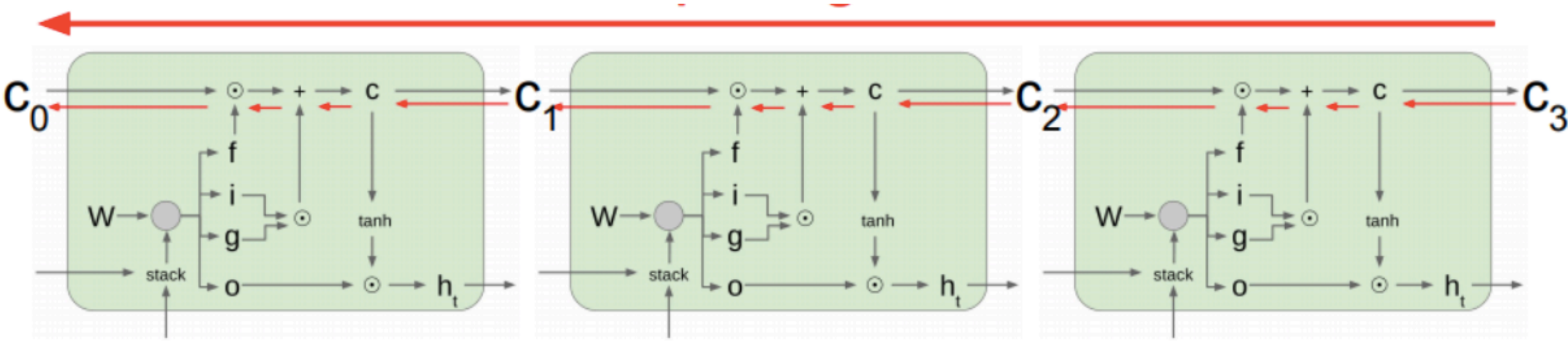
Decide what we're going to output to the next ***t***

First, we run a sigmoid layer which decides what parts of the cell state we're going to output

Then we run the cell state through ***tahn*** to normalise and filter it using the **output gate**. It goes next.



Gradient flow through RNN



Gradient flow through LSTM

This is it, thank you!

- Theory and intuition: Stanford CS224n
- Slides foundation and practical notebooks: Denis Antyukhov https://github.com/gaphex
- Love and Support: my dear wife, Vika
- Work Leave: my Habidatum team
- Armenian Code Academy: Karen and all of you!