Machine Learning Course at MIPT

Recurrent Neural Nets

Valentin Malykh

ml-mipt.github.io, val.maly.hk

iPavlov.ai

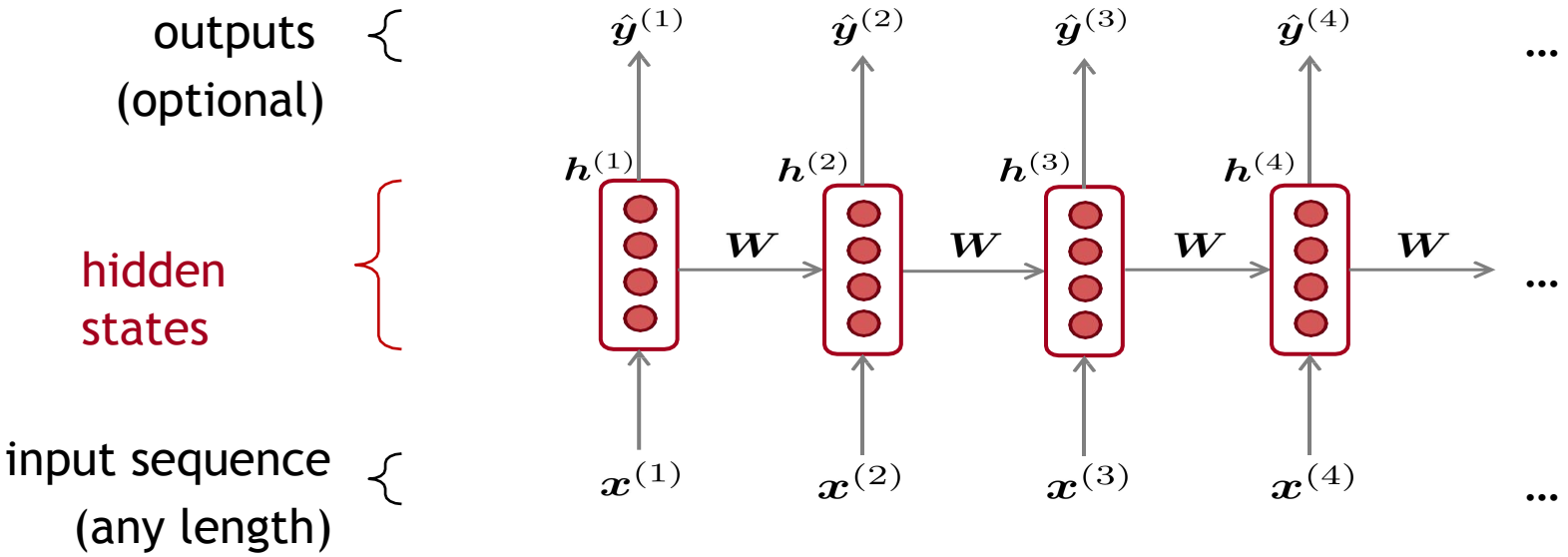
April 24th, 2018

Credit R. Socher

Recurrent Neural Networks (RNN)

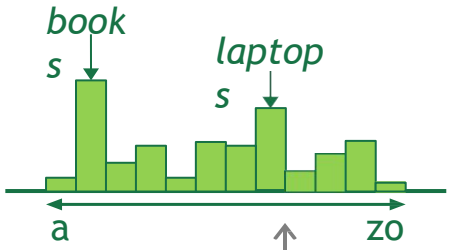
A family of neural architectures

Core idea: Apply the same weights *repeatedly* $\boldsymbol{W}$



A RNN Language Model

$\hat{\boldsymbol{y}}^{(4)} = P(\boldsymbol{x}^{(5)}|\text{the students opened their})$



output

$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$

hidden states

$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$

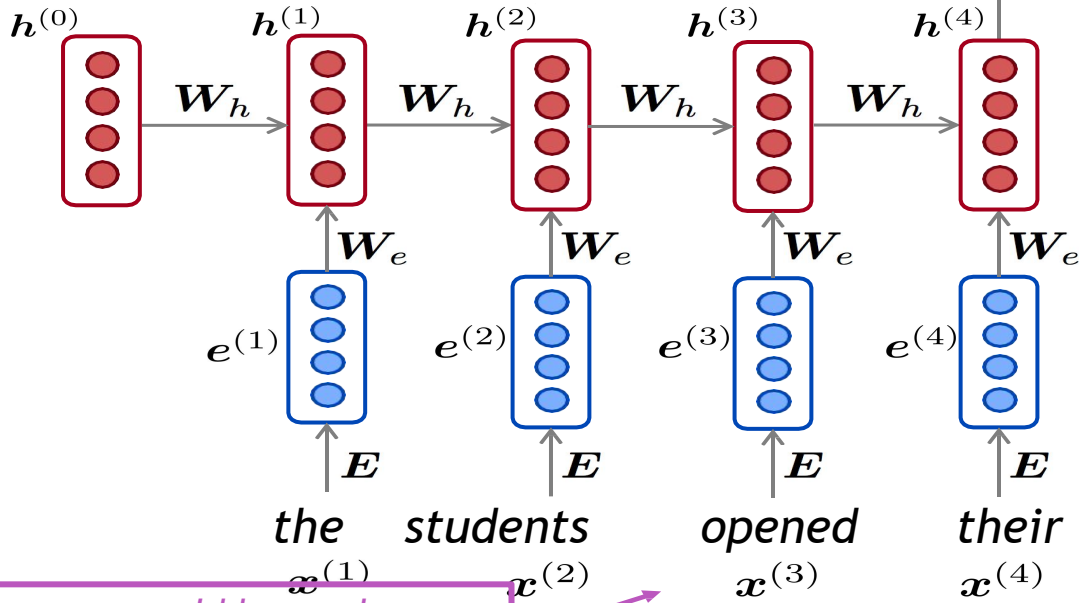
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$

words / one-hot vectors

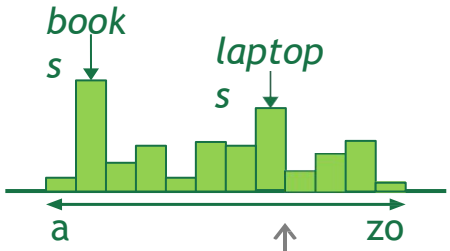
$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$



Note: this input sequence could be much longer, but this slide doesn't have space!

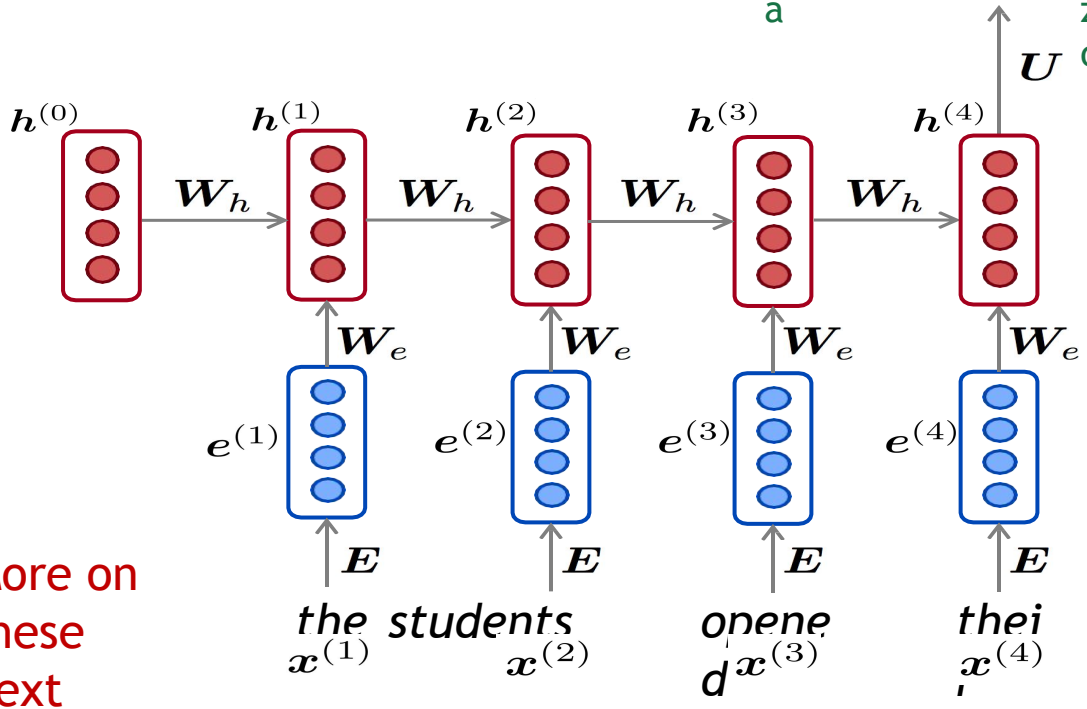
A RNN Language Model

$\hat{\boldsymbol{y}}^{(4)} = P(\boldsymbol{x}^{(5)}|\text{the students opened their})$



RNN **Advantages**:

- Can process any length input
- Model size doesn't increase for longer input
- Computation for step *t* can (in theory) use information from many steps back
- Weights are shared across timesteps → representations are shared



RNN **Disadvantages**:

- Recurrent computation is slow
- In practice, difficult to access information from many steps back

More on these next part

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, and the true next word $\hat{\boldsymbol{y}}^{(t)}$: $\boldsymbol{y}^{(t)} = \boldsymbol{x}^{(t+1)}$

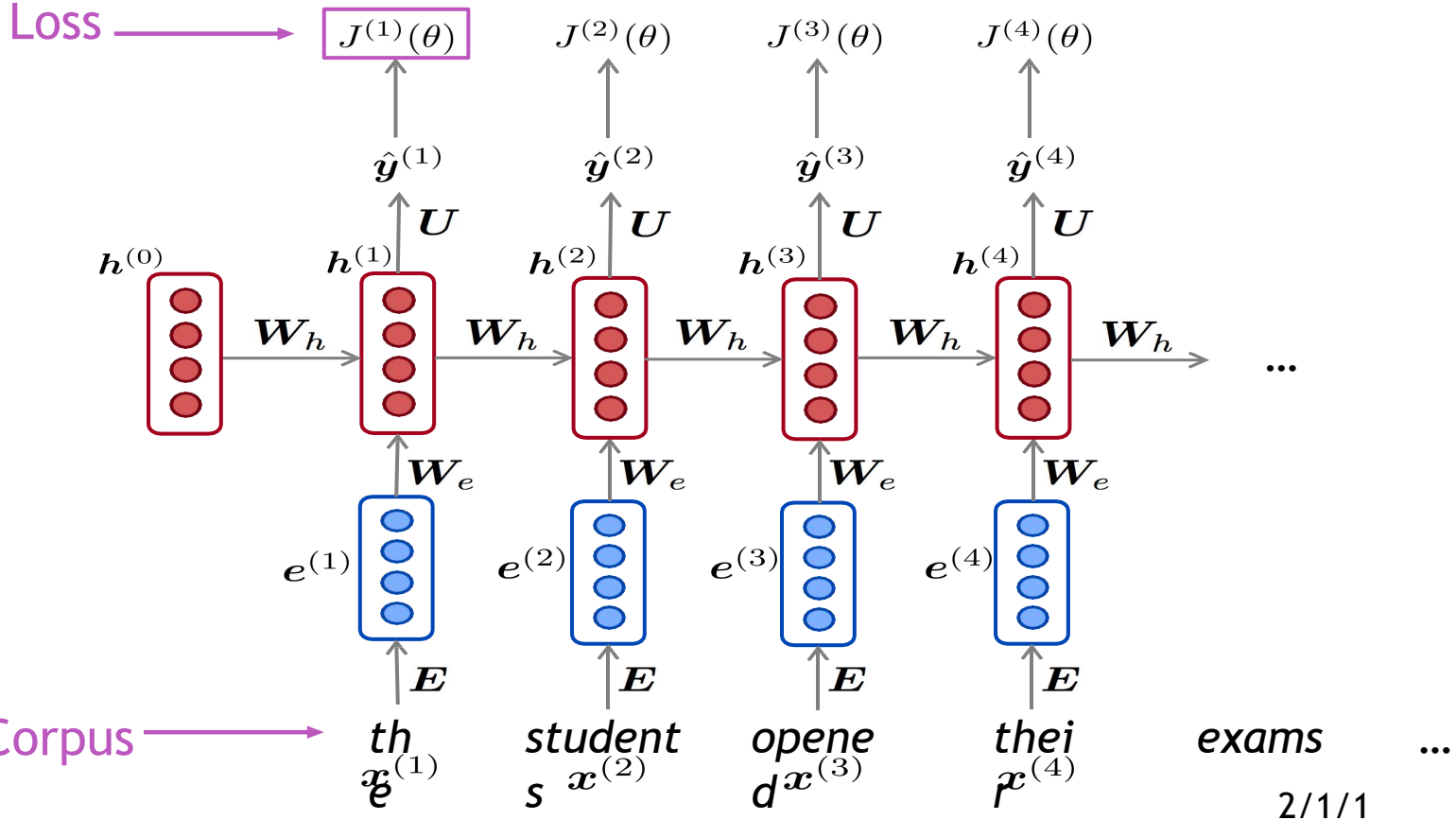
$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{j=1}^{|V|} y_j^{(t)} \log \hat{y}_j^{(t)}$$

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

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

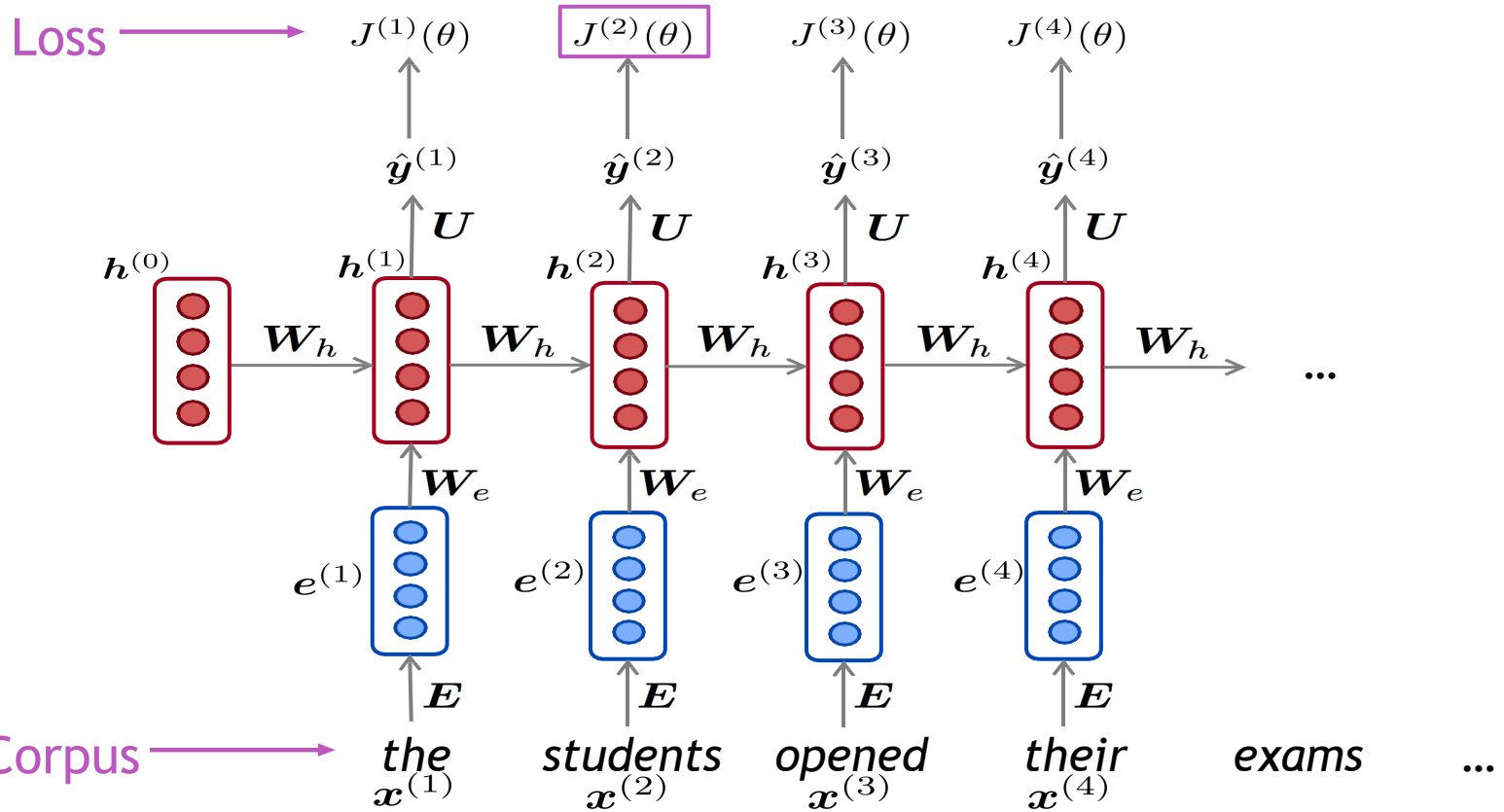
Training a RNN Language Model

= negative log prob of "students"



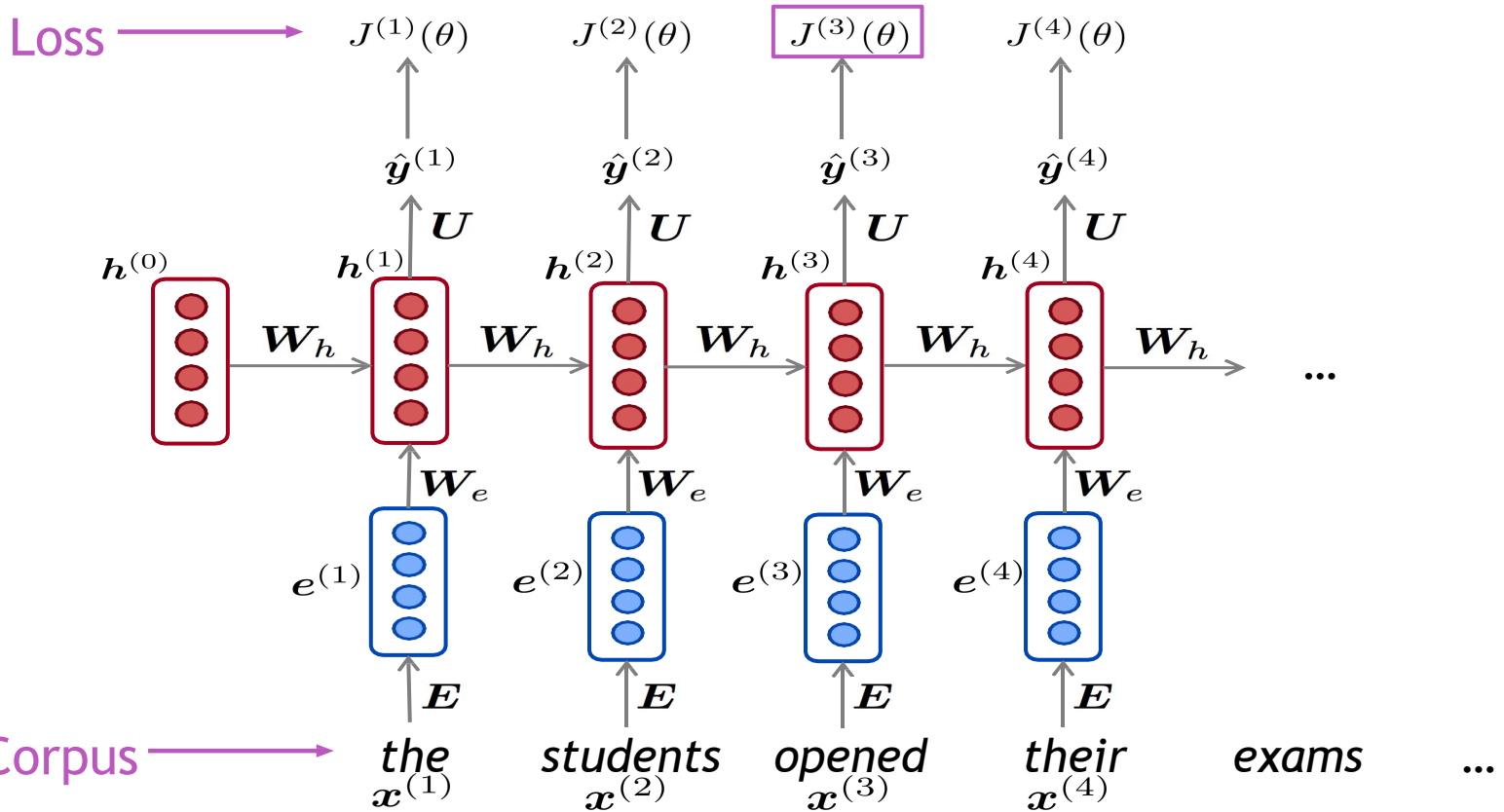
Training a RNN Language Model

= negative log prob of "opened"



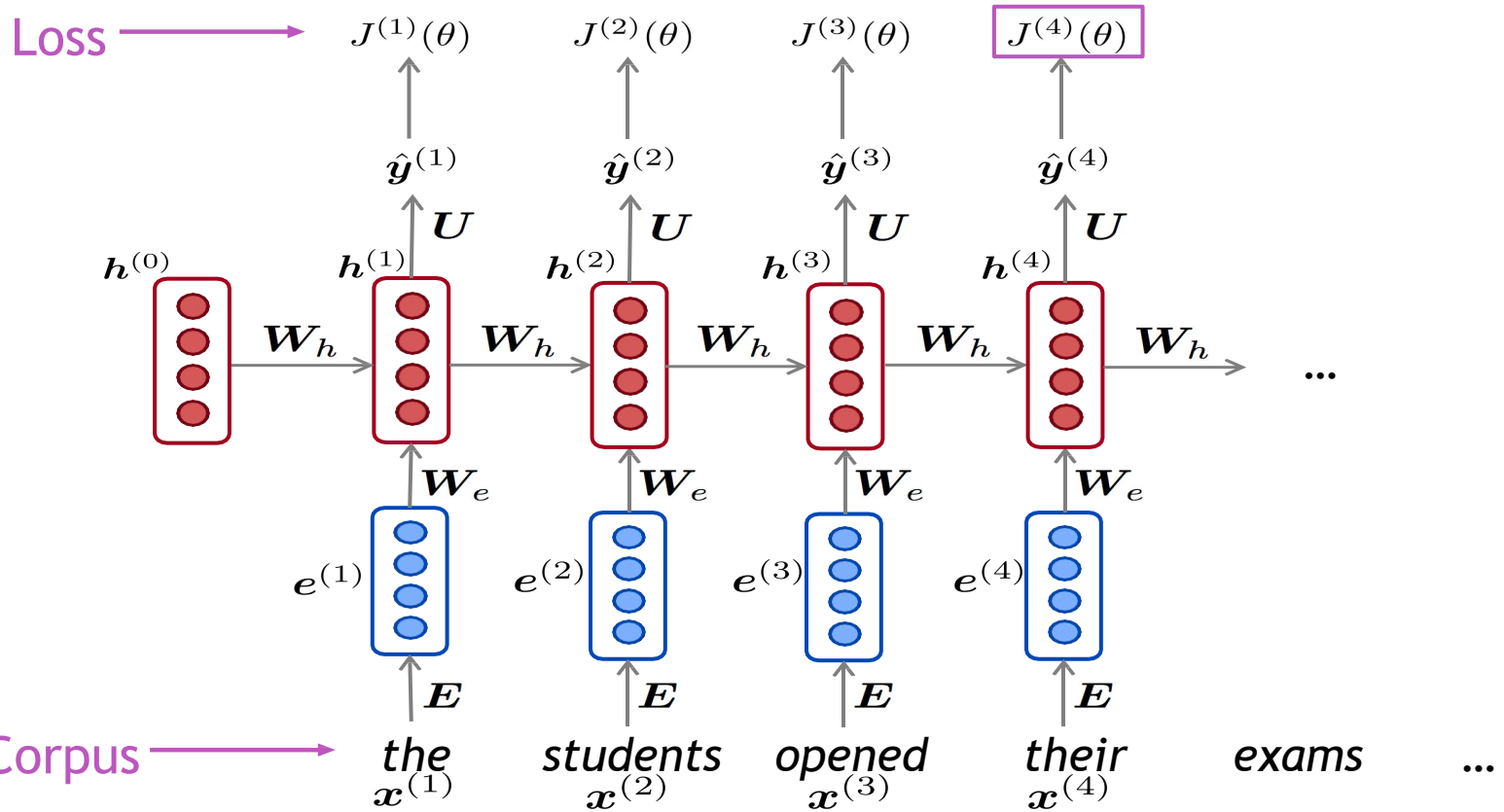
Training a RNN Language Model

= negative log prob of "their"

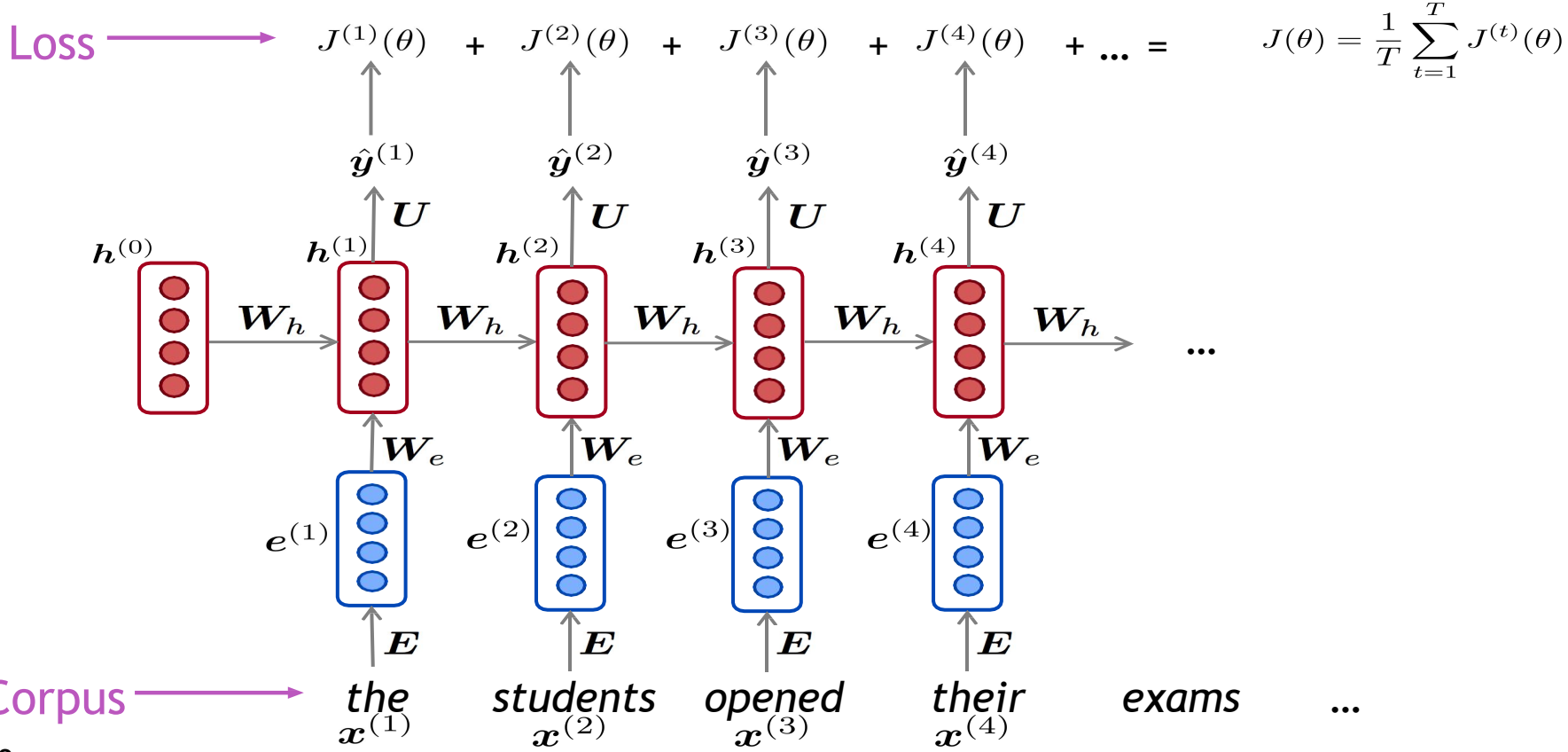


Training a RNN Language Model

= negative log prob of "exams"



Training a RNN Language Model



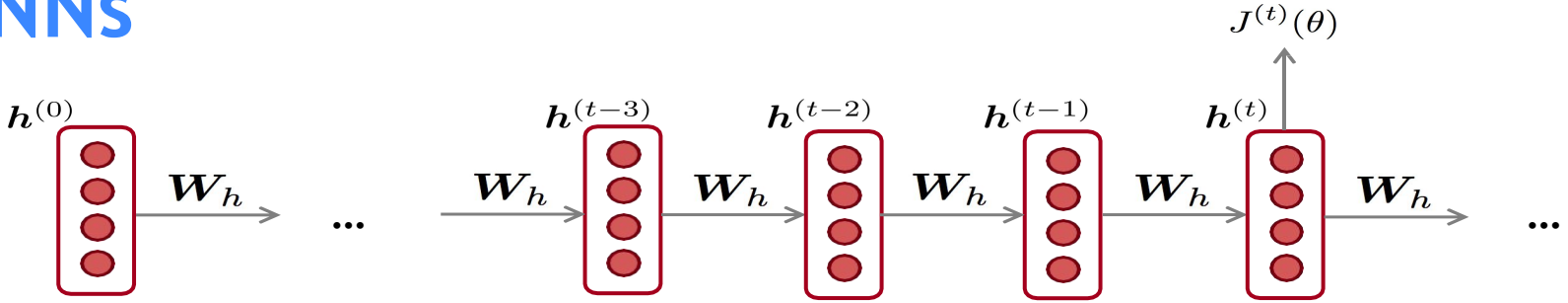
Training a RNN Language Model

- However: Computing loss and gradients across entire corpus is too expensive!
- Recall: Stochastic Gradient Descent allows us to compute loss and gradients for small chunk of data, and update. $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$

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

- → In practice, consider as a sentence
- Compute loss $J(\theta)$ a sentence (actually usually a batch of sentences), compute gradients and update weights. Repeat.

Backpropagation for RNNs



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

Answer:

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

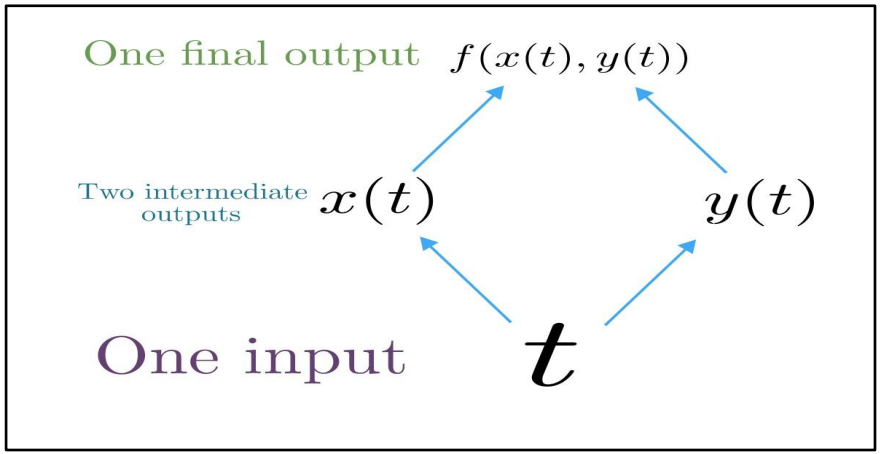
"The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears"

Why?

Multivariable Chain Rule

- Given a multivariable function $f(x, y)$, and two single variable functions $x(t)$ and $y(t)$, here's what the multivariable chain rule says:

$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$



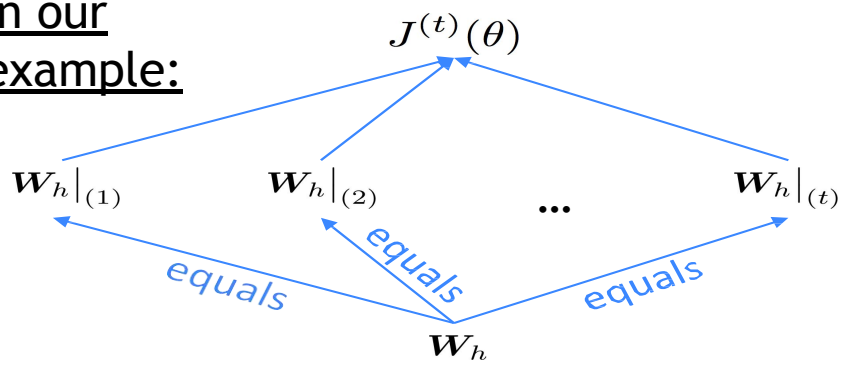
Source:
https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/multivariable-chain-rule-simple-version

Backpropagation for RNNs: Proof sketch

- Given a multivariable function $f(x, y)$, and two single variable functions $x(t)$ and $y(t)$, here's what the multivariable chain rule says:

$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

In our example:



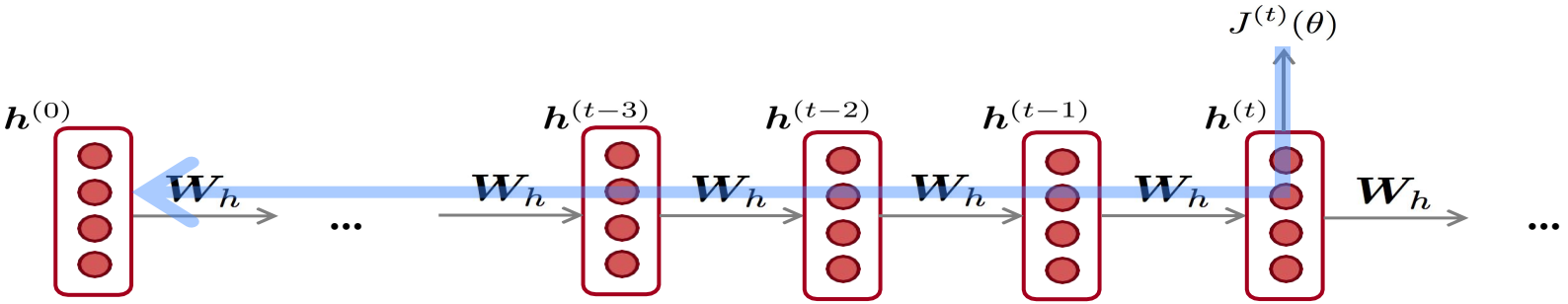
Apply the multivariable chain rule:

$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial W_h}\Big|_{(i)} \frac{\partial W_h|_{(i)}}{\partial W_h} = 1$$

$$= \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial W_h}\Big|_{(i)}$$

Source:
https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/multivariable-chain-rule-simple-version

Backpropagation for RNNs



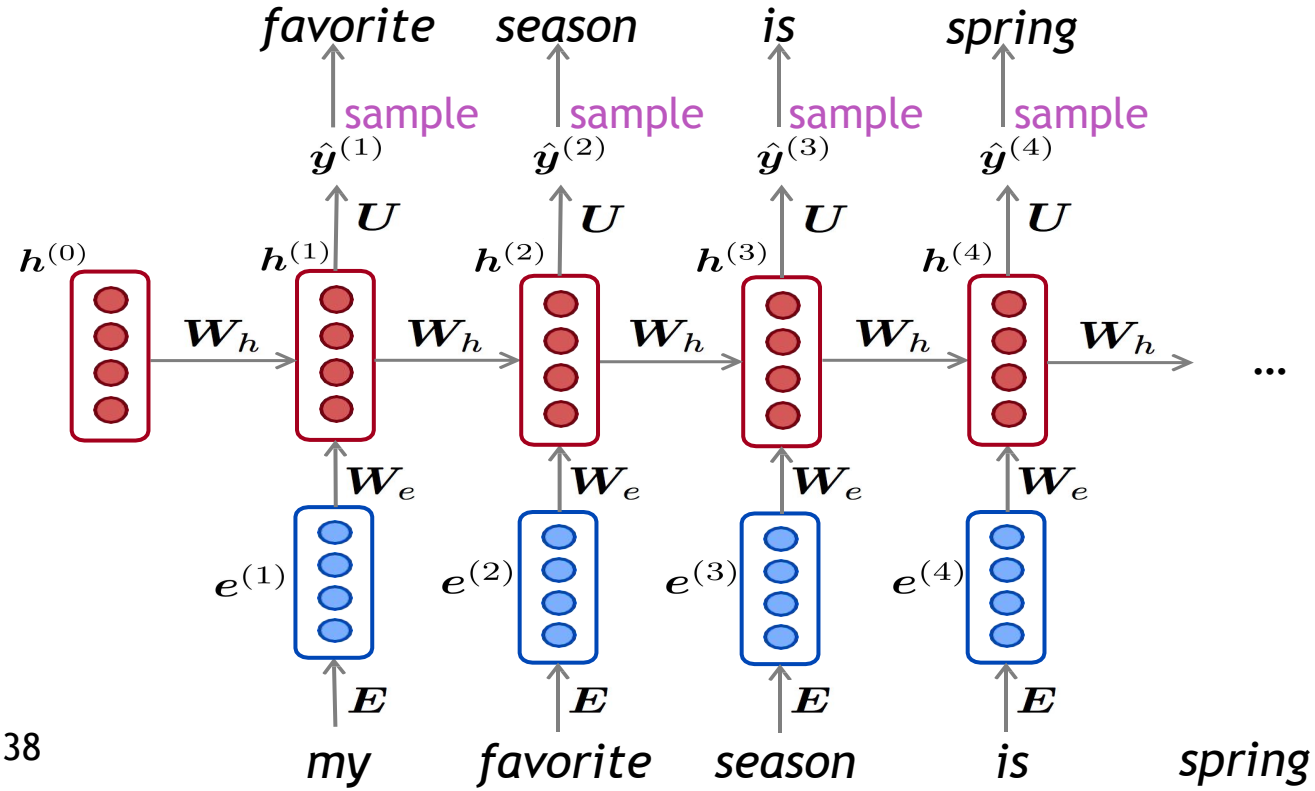
$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}}\Big|_{(i)}$$

Question: How do we calculate this?

Answer: Backpropagate over timesteps *i*=*t*,…,0, summing gradients as you go.
This algorithm is called **"backpropagation through time"**

Generating text with a RNN Language Model

Just like a n-gram Language Model, you can use a RNN Language Model to generate text by repeated sampling. Sampled output is next step's input.



Generating text with a RNN Language Model

- Let's have some fun!
- You can train a RNN-LM on any kind of text, then generate text in that style.
- RNN-LM trained on *Harry Potter*:

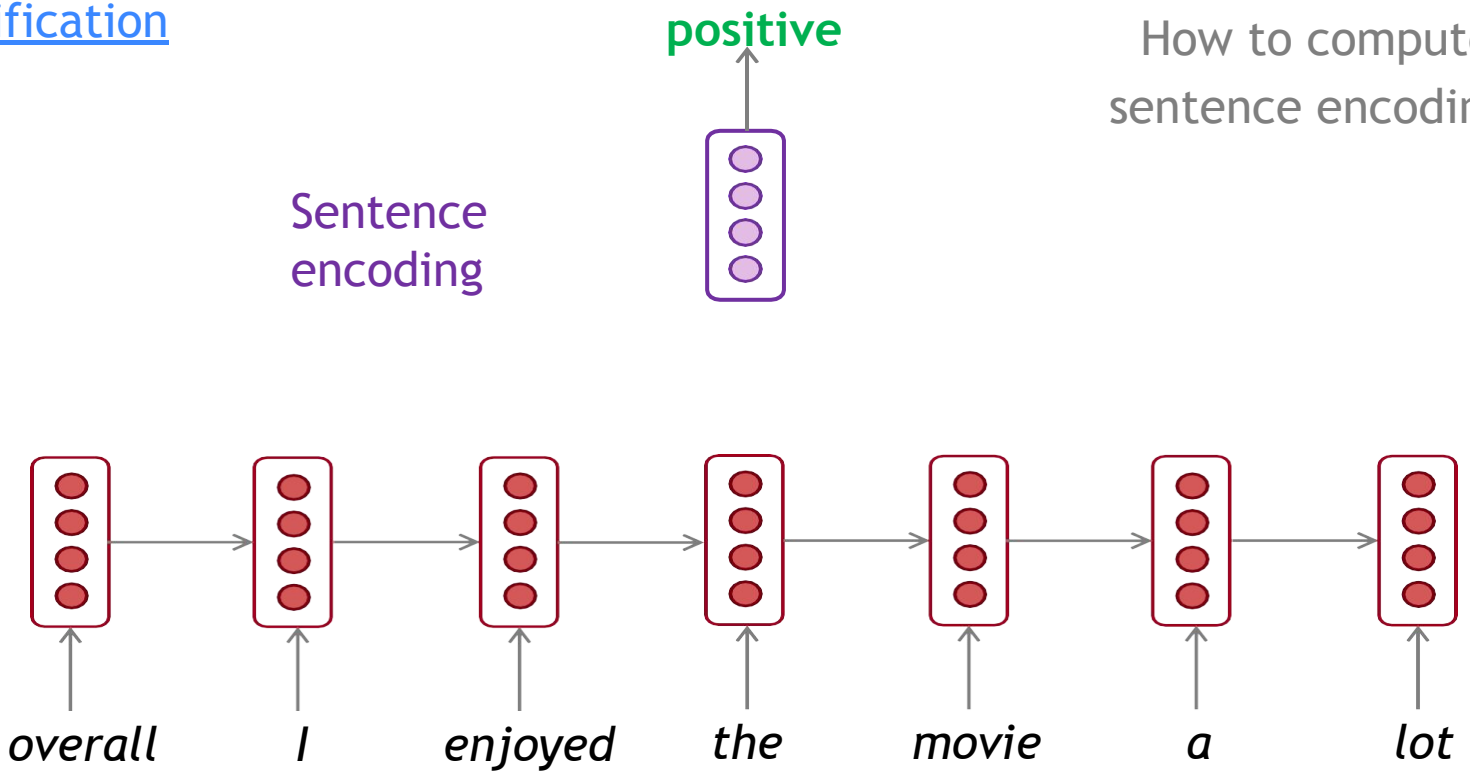
"Sorry," Harry shouted, panicking—"I'll leave those brooms in London, are they?"

"No idea," said Nearly Headless Nick, casting low close by Cedric, carrying the last bit of treacle Charms, from Harry's shoulder, and to answer him the common room perched upon it, four arms held a shining knob from when the spider hadn't felt it seemed. He reached the teams too.

Source: https://medium.com/deep-writing/harry-potter-written-by-artificial-intelligence-8a9431803da6

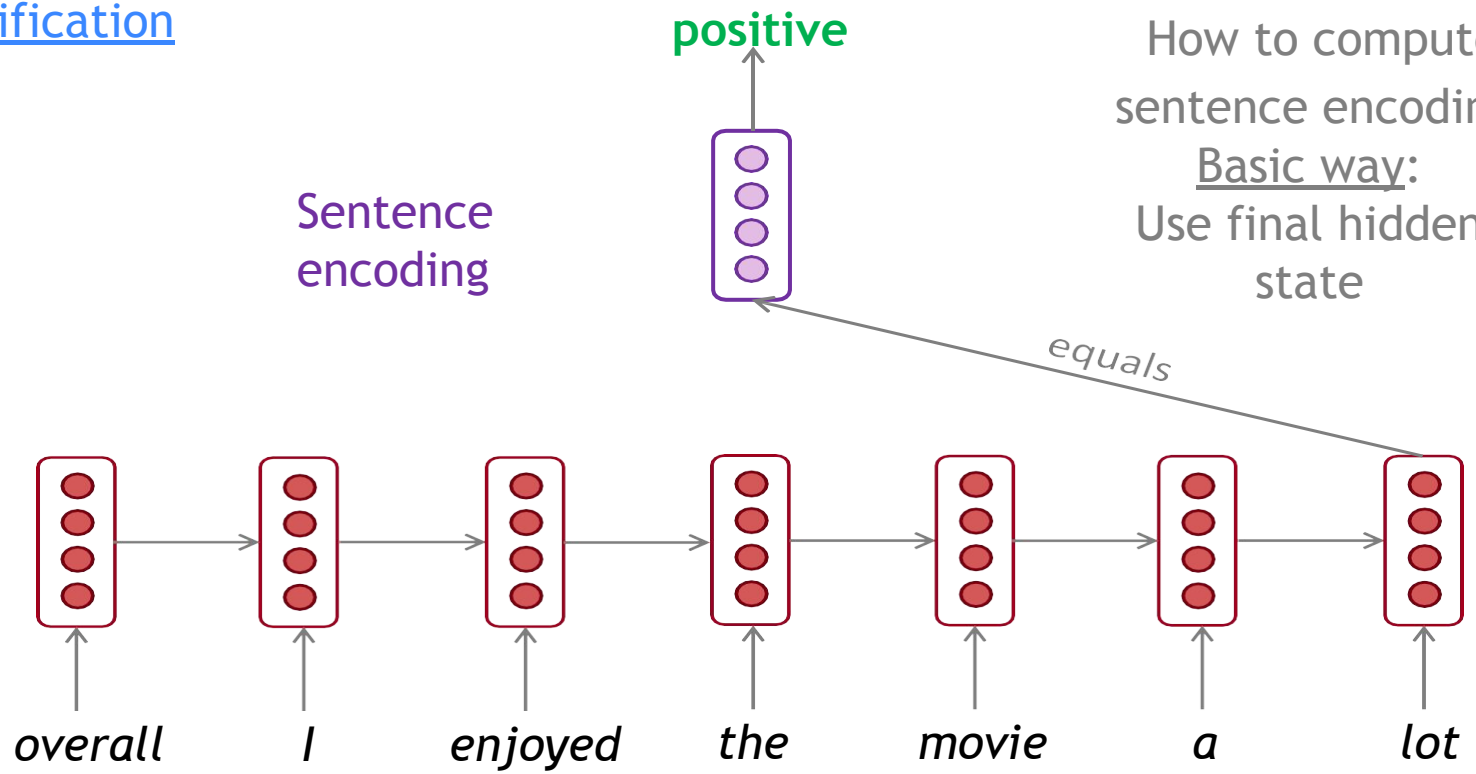
RNNs can be used for sentence classification

e.g. sentiment classification



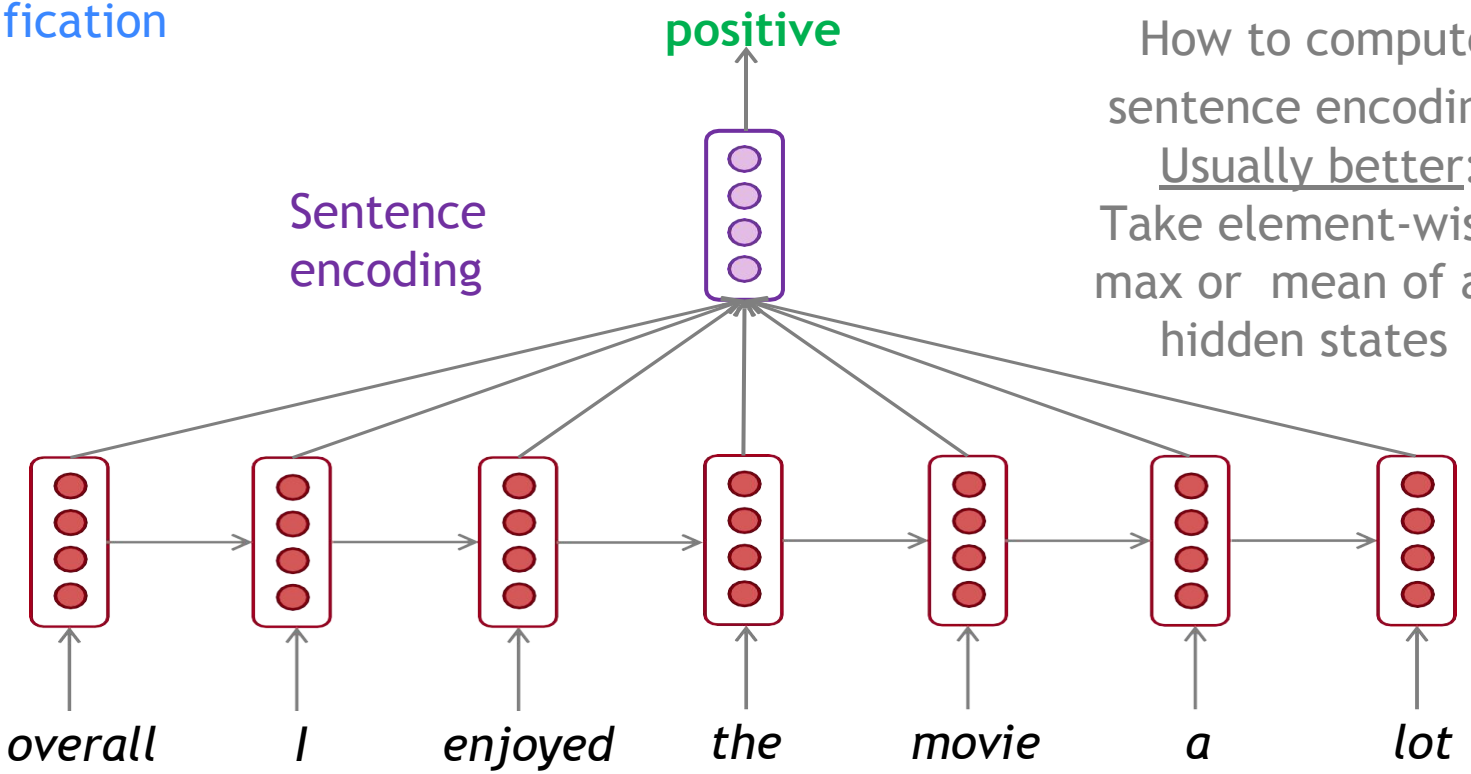
RNNs can be used for sentence classification

e.g. sentiment classification



RNNs can be used for sentence classification

e.g. sentiment classification

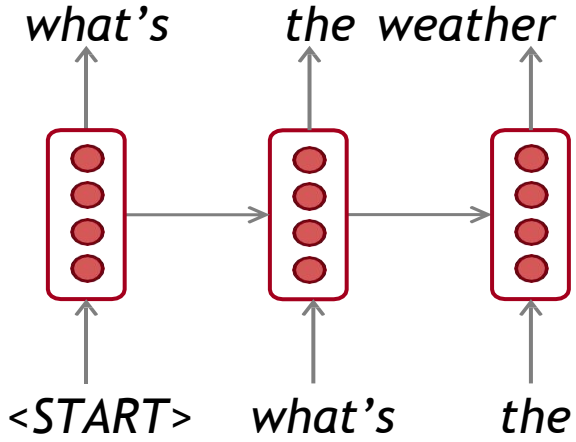


How to compute sentence encoding?

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

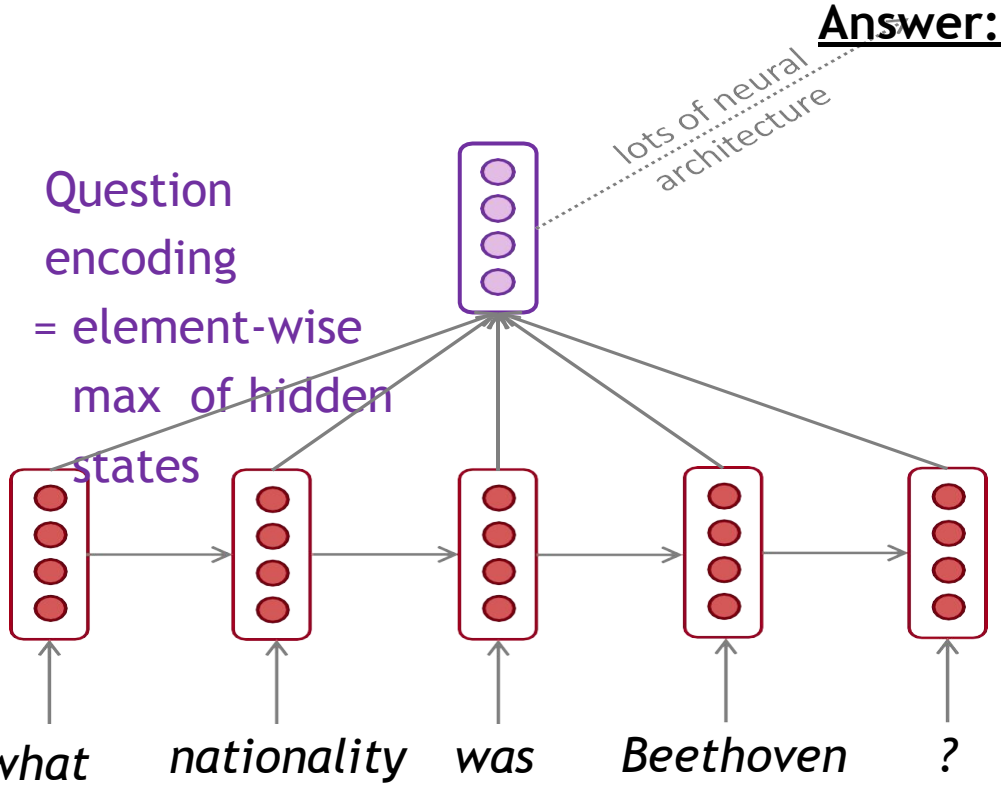


Remember: these are called "conditional language models".

We'll see Machine Translation in more detail later.

RNNs can be used as an encoder module

e.g. question answering, machine translation



Context: *Ludwig van Beethoven was a German composer and pianist. A crucial figure*

…

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

A note on terminology

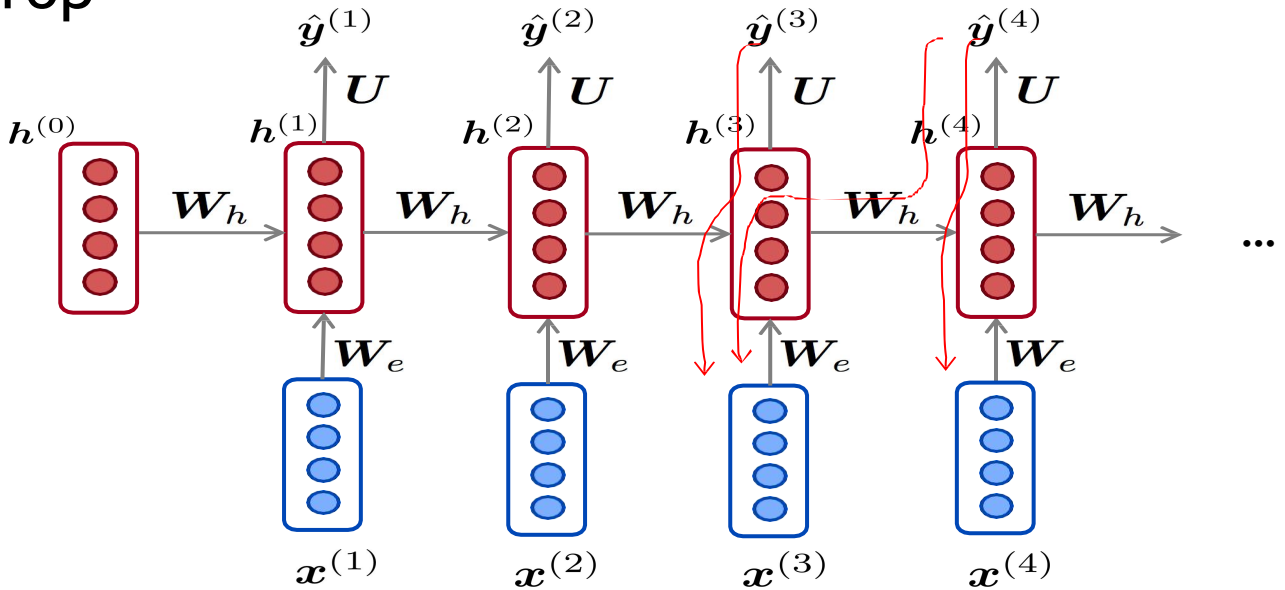
RNN described in this lecture = “vanilla RNN”

Next lecture: You will learn about other RNN flavors like GRU  and LSTM  and multi-layer RNNs 

By the end of the lecture: You will understand phrases like *“stacked bidirectional LSTM with residual connections and self-attention”*

The vanishing/exploding gradient problem

- Multiply the same matrix at each time step during backprop



The vanishing gradient problem - Details

- Similar but simpler RNN formulation:

$$\begin{aligned} h_t &= W f(h_{t-1}) + W^{(hx)} x_{[t]} \\ \hat{y}_t &= W^{(S)} f(h_t) \end{aligned}$$

- Total error is the sum of each error (aka cost function, aka J in previous lectures when it was cross entropy error, could be other cost functions too), but at time steps t

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W}$$

- Hardcore chain rule application:

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

The vanishing gradient problem - Details

- Similar to backprop but less efficient formulation
- Useful for analysis we'll look at:

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \boxed{\frac{\partial h_t}{\partial h_k}} \frac{\partial h_k}{\partial W}$$

We'll show this can quickly become very small or very large

- Remember: $h_t = W f(h_{t-1}) + W^{(hx)} x_{[t]}$
- More chain rule, remember:

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}$$

- Each partial is a Jacobian:

$$\frac{d\mathbf{f}}{d\mathbf{x}} = \left[\frac{\partial \mathbf{f}}{\partial x_1} \quad \cdots \quad \frac{\partial \mathbf{f}}{\partial x_n} \right] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

The vanishing gradient problem - Details

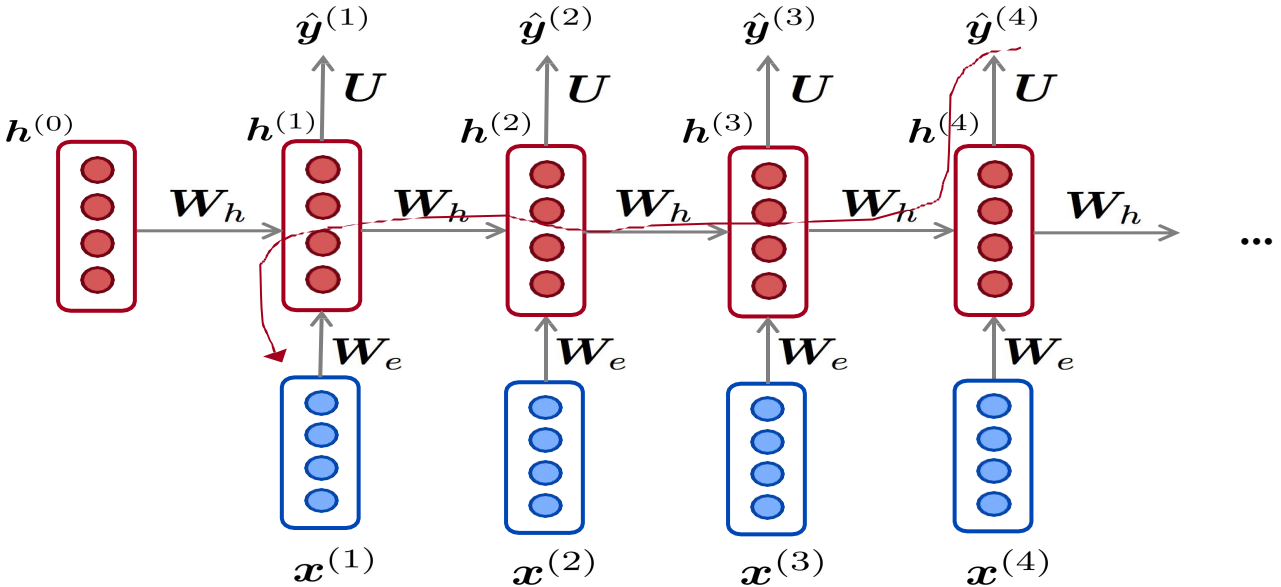
- Analyzing the norms of the Jacobians yields:

$$\left\| \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq \|W^T\| \|\text{diag}[f'(h_{j-1})]\| \leq \beta_W \beta_h$$

- Where we defined β's as upper bounds of the norms
- The gradient is a product of Jacobian matrices, each associa $\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq (\beta_W \beta_h)^{t-k}$ computation.
- This can become very small or very large quickly [Bengio et al 1994], and the locality assumption of gradient descent breaks down. → **Vanishing or exploding gradient**

Why is the vanishing gradient a problem?

- Ideally, the error E^t on step t can flow backwards, via backprop, and allow the weights on a previous timestep (maybe many timesteps ago) to change.



Why is the vanishing gradient a problem?

1. Gradients can be seen as a measure of influence of the past on the future

2. How does the perturbation at time t affect predictions at t+n?

Why is the vanishing gradient a problem?

When we only observe

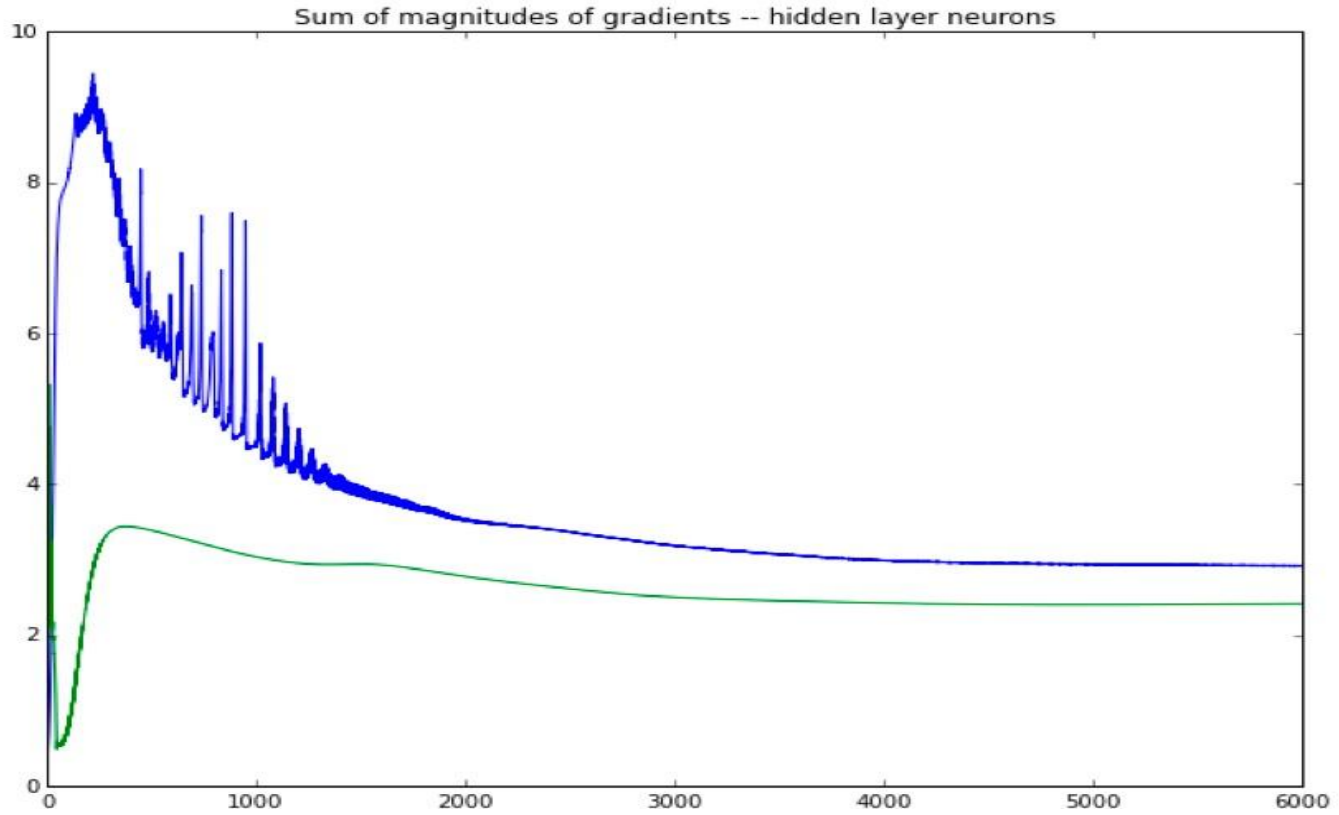
$$\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq (\beta_W \beta_h)^{t-k}$$

We cannot tell whether it's going to 0

1. No dependency between *t* and *t+n* in data, or
2. Wrong configuration of parameters

In [21]:
```
plt.plot(np.array(relu_array[:6000]),color='blue')
plt.plot(np.array(sigm_array[:6000]),color='green')
plt.title('Sum of magnitudes of gradients -- hidden layer neurons')
```

Out[21]: <matplotlib.text.Text at 0x10a331310>



Trick for exploding gradient: clipping trick

- The solution first introduced by Mikolov is to clip gradients so that their norm has some maximum value.

Algorithm 1 Pseudo-code for norm clipping the gradients whenever they explode

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
end if

s

- Makes a big difference in RNNs and many other unstable models

Gradient clipping intuition

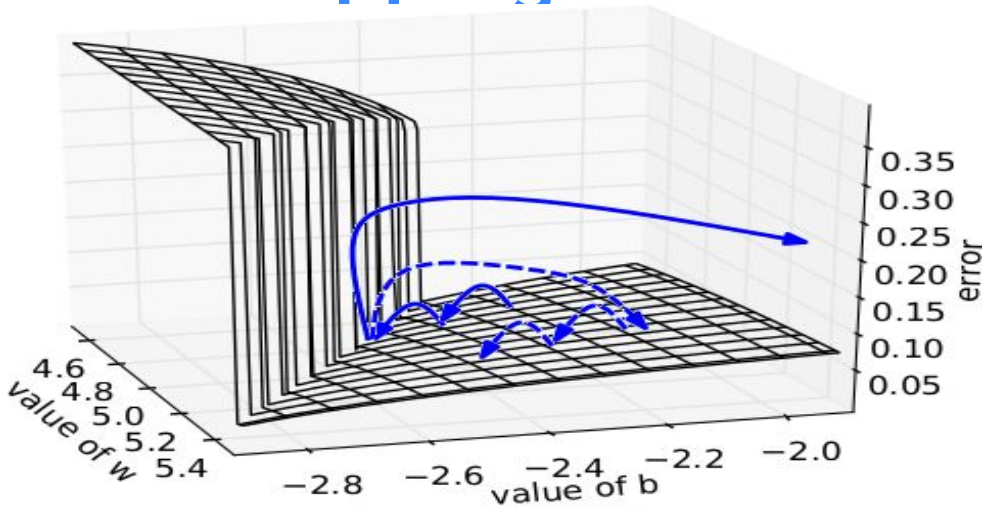


Figure from paper: On the difficulty of training Recurrent Neural Networks, Pascanu et al. 2013

- Error surface of a single hidden unit RNN,
- High curvature walls
- Solid lines: standard gradient descent trajectories
- Dashed lines gradients rescaled to fixed size

Main solution for better RNNs: Better Units

- The main solution to the Vanishing Gradient Problem is to use a more complex hidden unit computation in recurrence!
- Gated Recurrent Units (GRU) introduced by [Cho et al. 2014] and LSTMs [Hochreiter & Schmidhuber, 1999]
- Main ideas:
  - keep around memories to capture long distance dependencies
  - allow error messages to flow at different strengths depending on the inputs

GRUs

- Standard RNN computes hidden layer at next time step directly: $$h_t = f\left(W^{(hh)}h_{t-1} + W^{(hx)}x_t\right)$$
- #Simplified from: $$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$
- GRU first computes an update **gate** (another layer) based on current input word vector and hidden state
$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$
$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$
- Compute reset gate similarly but with different weights

GRUs

- Update gate $$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$
- Reset gate $$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$
- New memory content: $$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right)$$
  If reset gate unit is ~0, then this ignores previous memory and only stores the new word information
- Final memory at time step combines current and previous time steps: $$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

GRU illustration

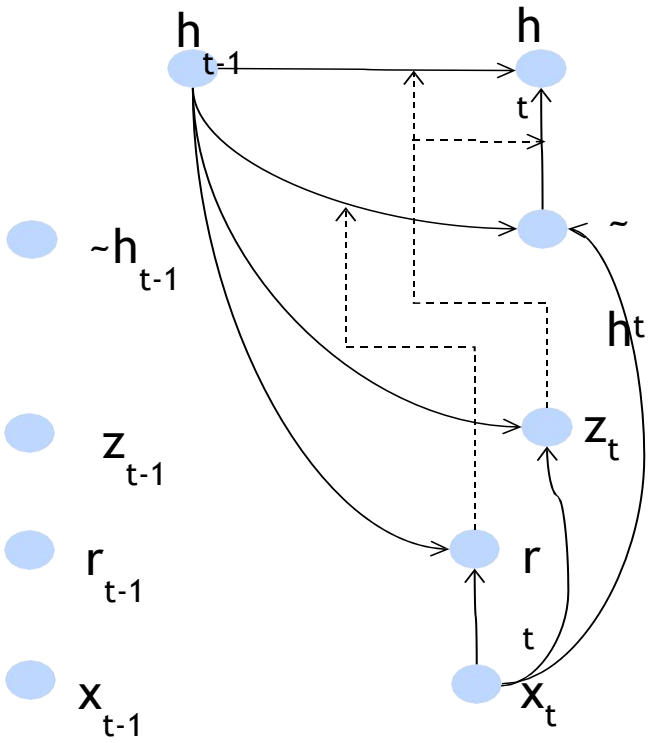
Final memory

Memory (reset)

Update gate

Reset gate

Input :



$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$

$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$

$$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right)$$

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

GRU intuition

- If reset is close to 0,
  ignore previous hidden state
  → Allows model to drop information that is irrelevant in the future

$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$
$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$
$$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right)$$

- Update gate z controls how much of past state s

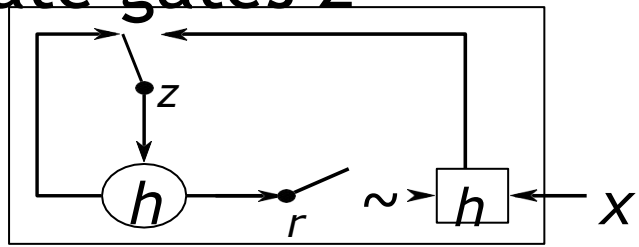
$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

  - If z close to 1, then we can copy information in that unit through many time steps! **Less vanishing gradient**!

- Units with short-term dependencies often have reset gates very active

GRU intuition

- Units with long term dependencies have active update gates z
- Illustration:

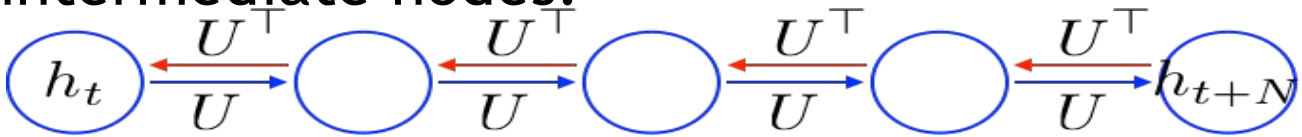


$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$
$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$
$$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right)$$
$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

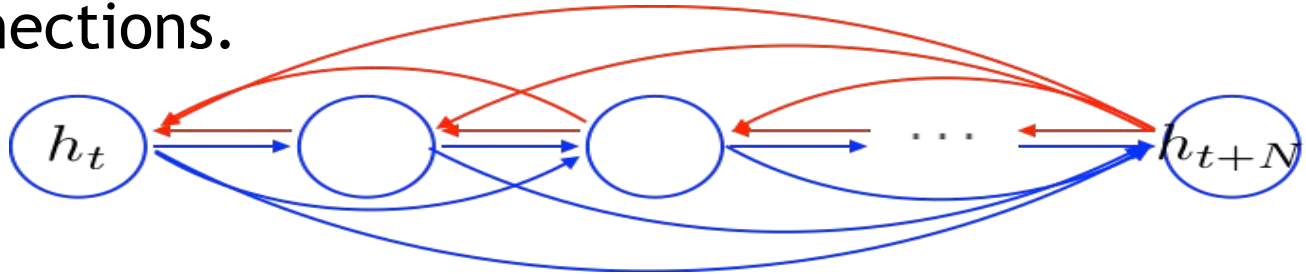
- Derivative of $\frac{\partial}{\partial x_1} x_1 x_2$? → rest is same chain rule, but implement with **modularization** or automatic differentiation

How do Gated Recurrent Units fix vanishing gradient problems?

- Is the problem with standard RNNs the naïve transition function? $h_t = f\left(W^{(hh)} h_{t-1} + W^{(hx)} x_t\right)$
- It implies that the error must backpropagate through all the intermediate nodes:

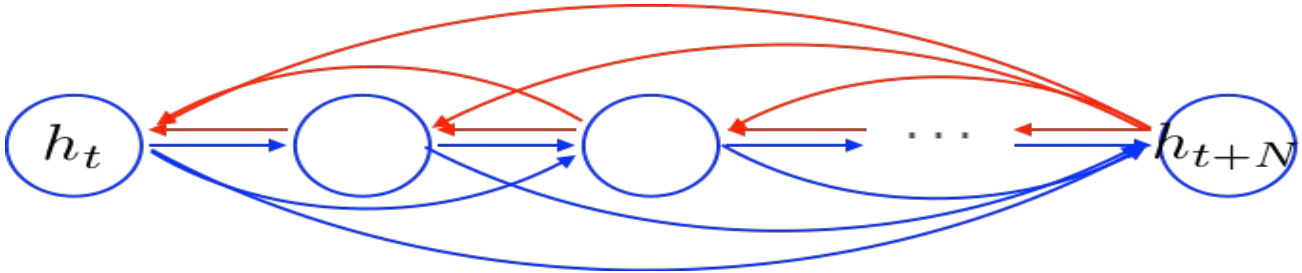


- Perhaps we can create shortcut connections.



How do Gated Recurrent Units fix vanishing gradient problems?

- Perhaps we can create *adaptive* shortcut connections.
- Let the net prune unnecessary connections *adaptively*.

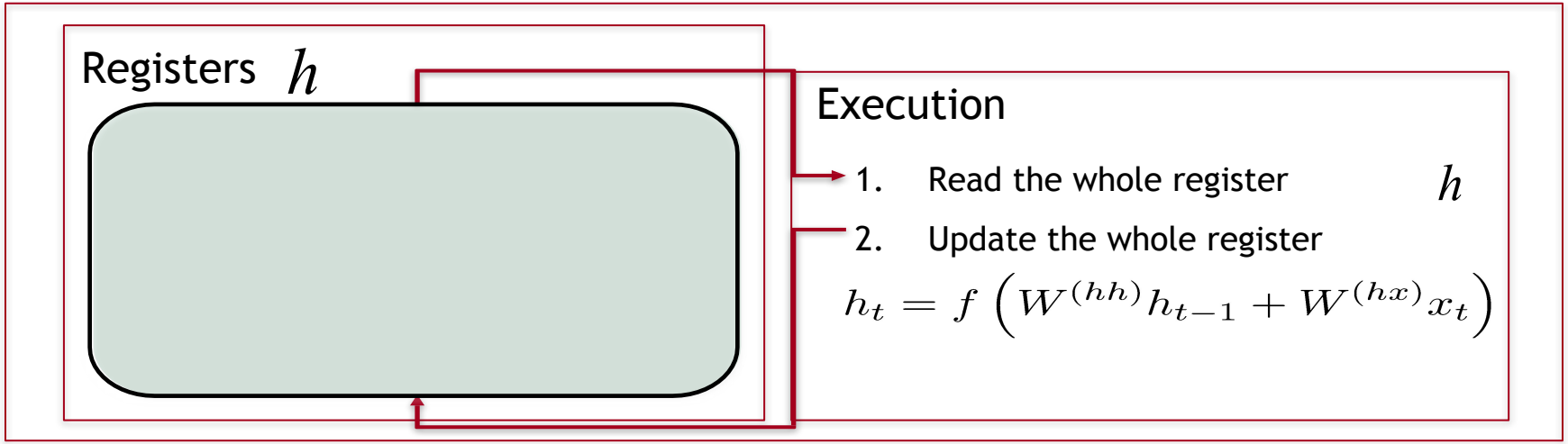


- That's what the gates do.

$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$
$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$
$$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right)$$
$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

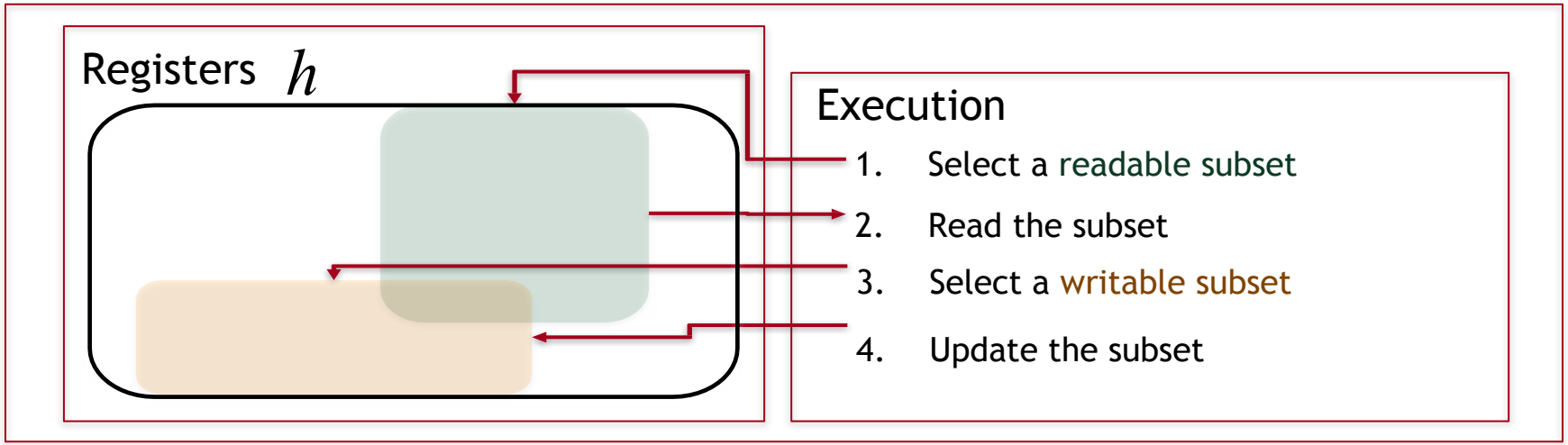
GRU Comparison to Standard tanh-RNN

Vanilla RNN …



GRU Comparison to Standard tanh-RNN

GRU …

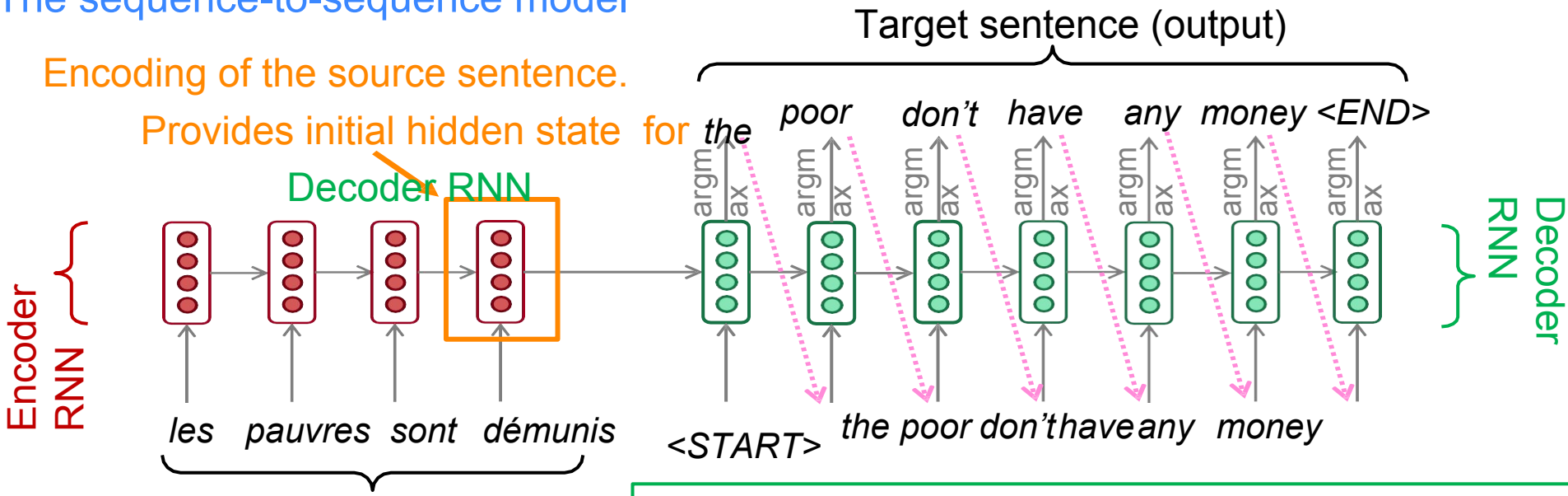


$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

Gated recurrent units are much more versatile and adaptive in which elements of the hidden vector h they update!

Neural Machine Translation (NMT)

The sequence-to-sequence model

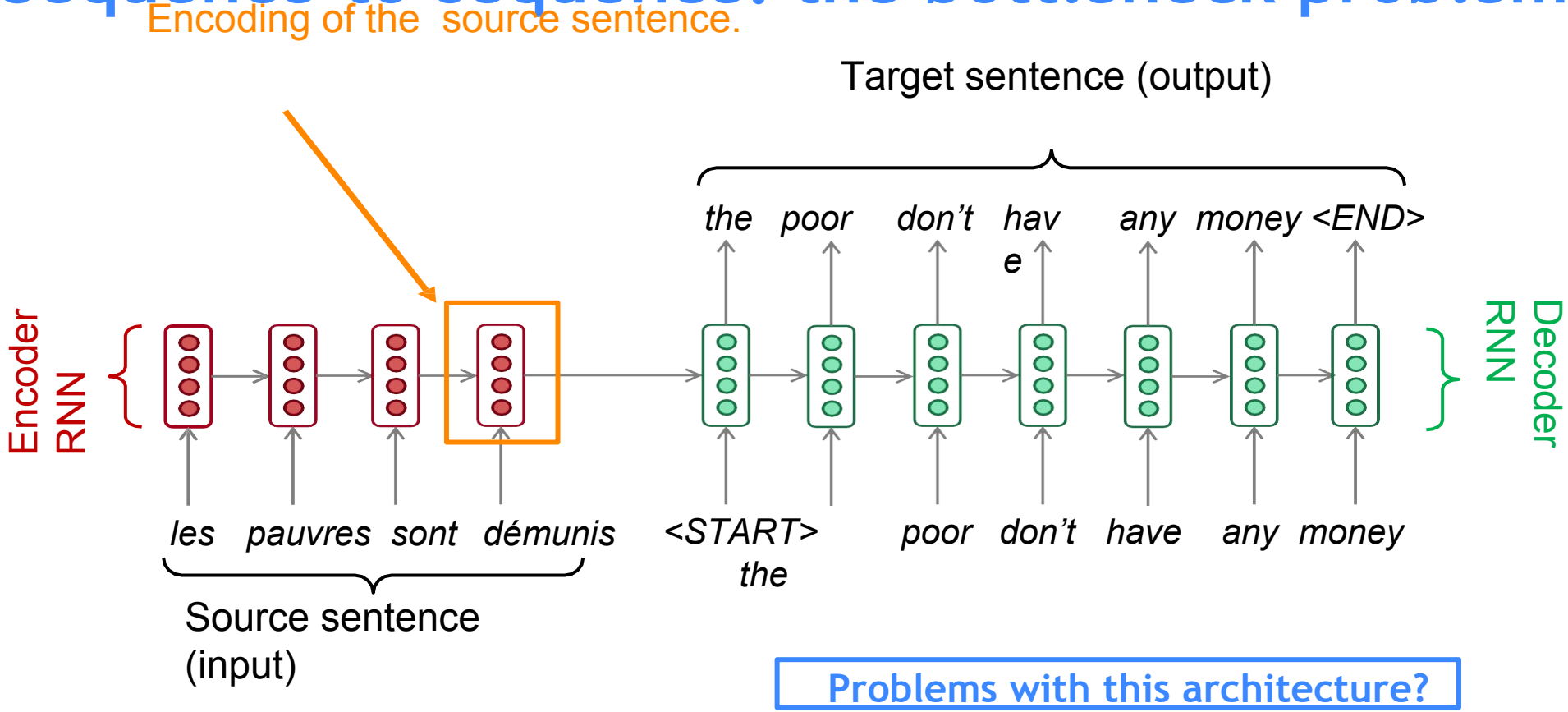


Decoder RNN is a Language Model that generates target sentence conditioned on encoding.

Encoder RNN produces an encoding of the source sentence.

Note: This diagram shows **test time** behavior: decoder output is fed in as next step's input

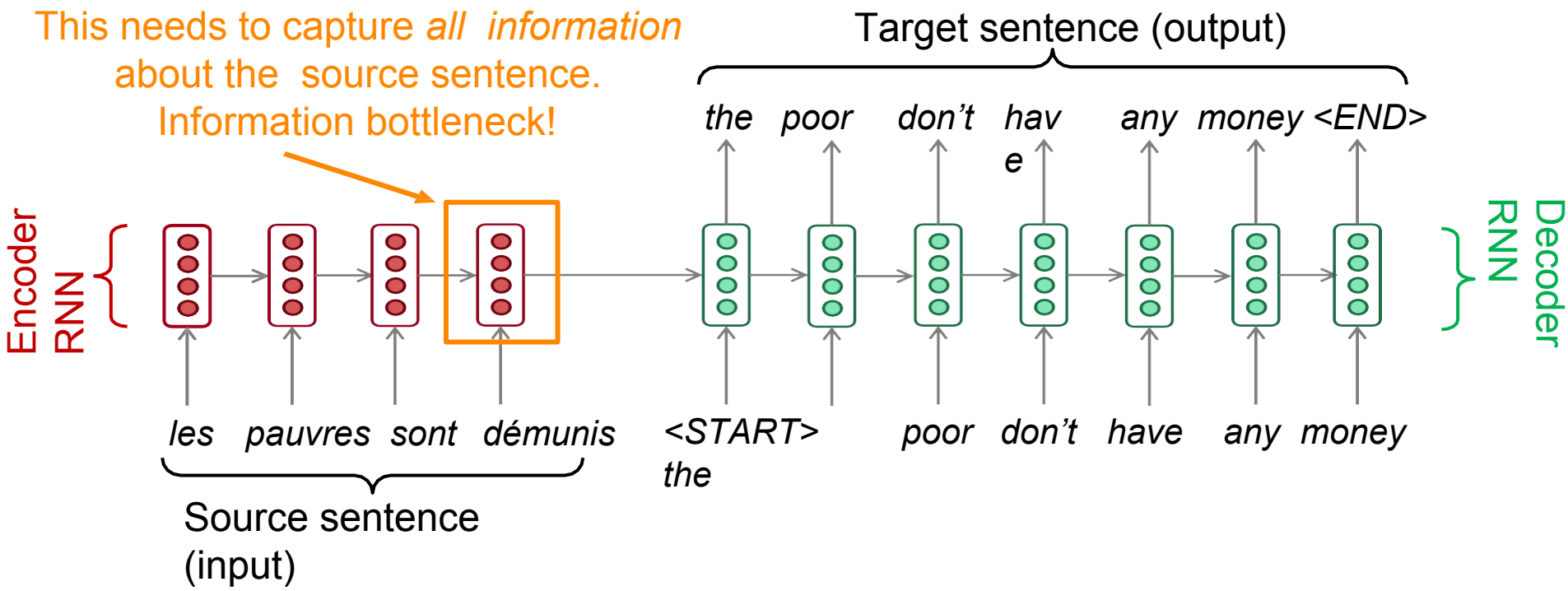
Sequence-to-sequence: the bottleneck problem



Sequence-to-sequence: the bottleneck problem

Encoding of the source sentence.

This needs to capture *all information* about the source sentence.
Information bottleneck!



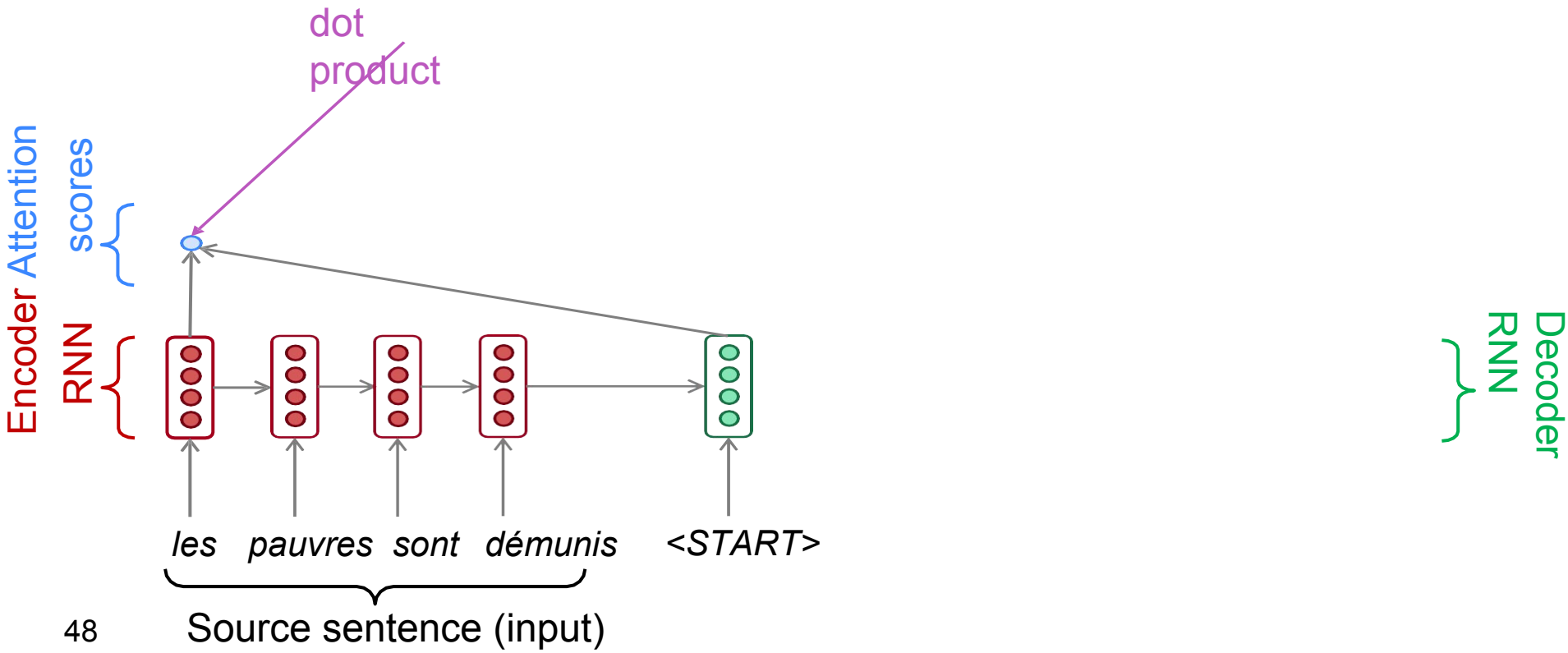
Attention

- **Attention** provides a solution to the bottleneck problem.

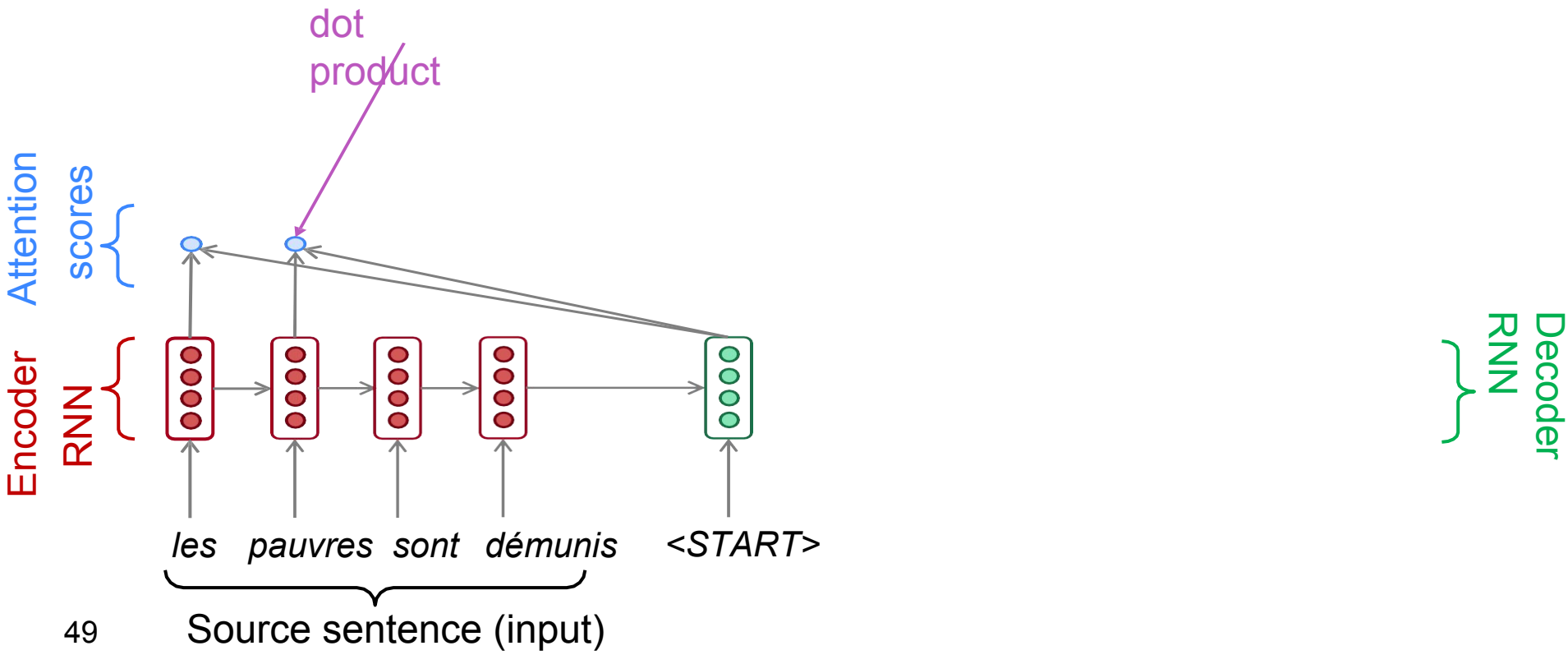
- Core idea: on each step of the decoder, *focus on a particular part* of the source sequence

- First we will show via diagram (no equations), then we will show with equations

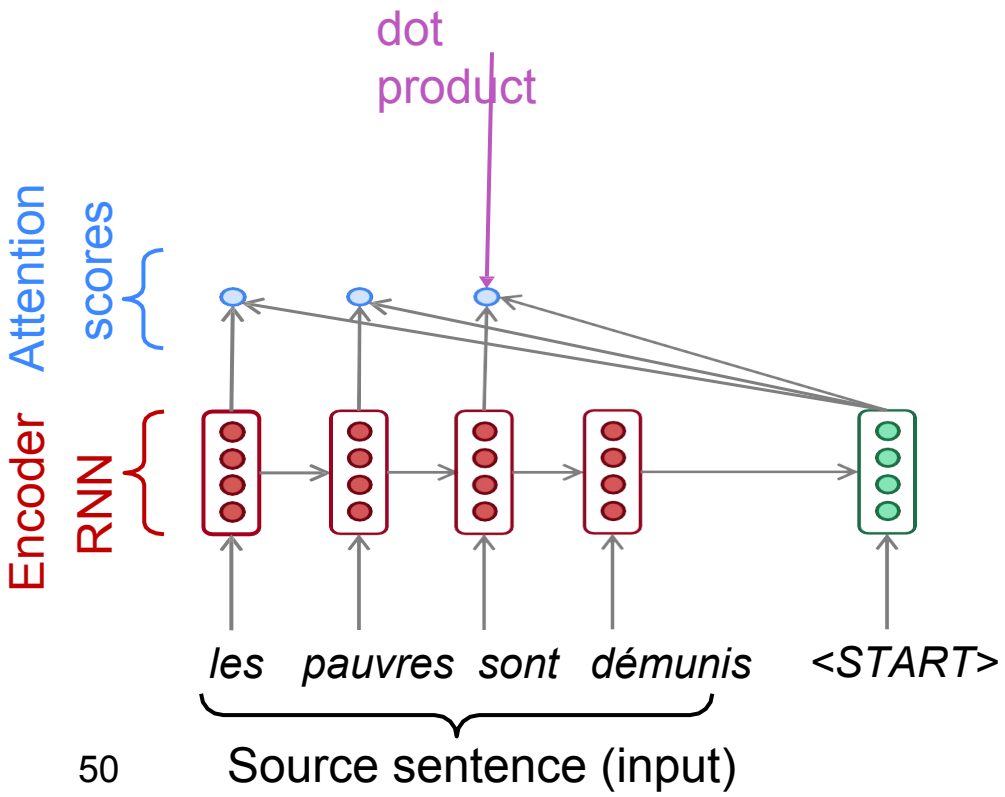
Sequence-to-sequence with attention



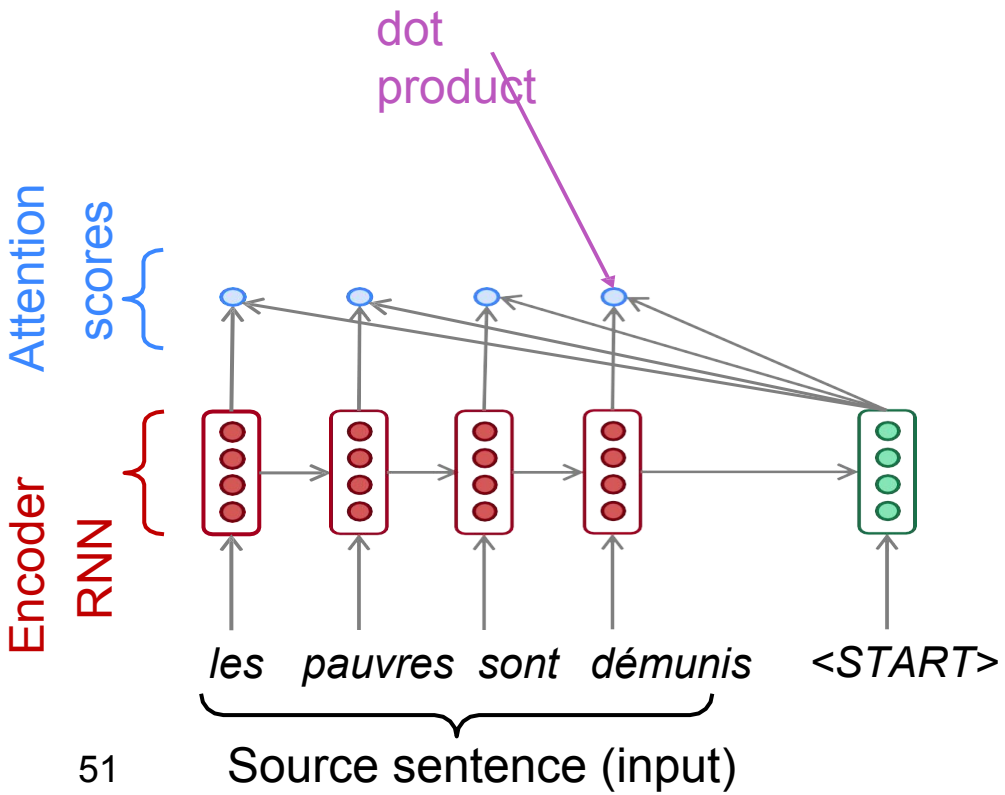
Sequence-to-sequence with attention



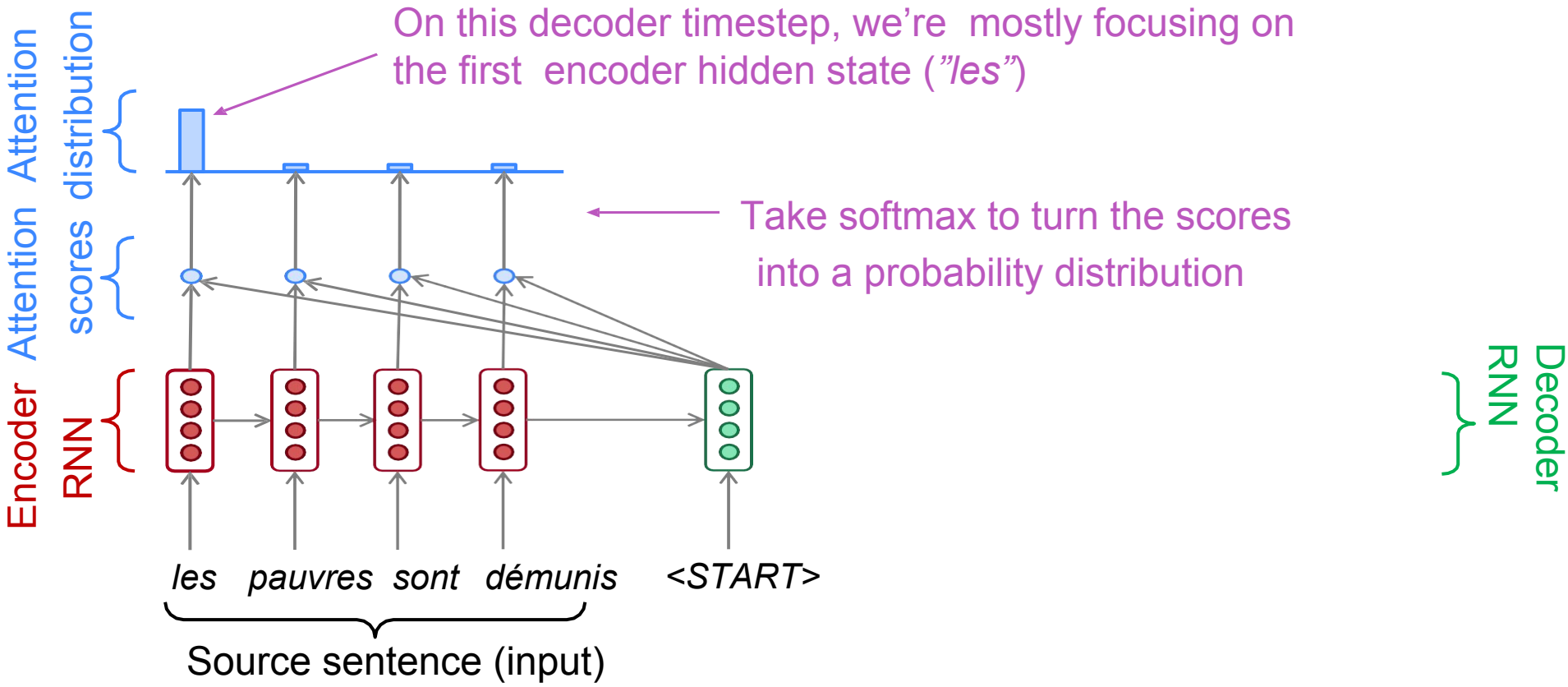
Sequence-to-sequence with attention



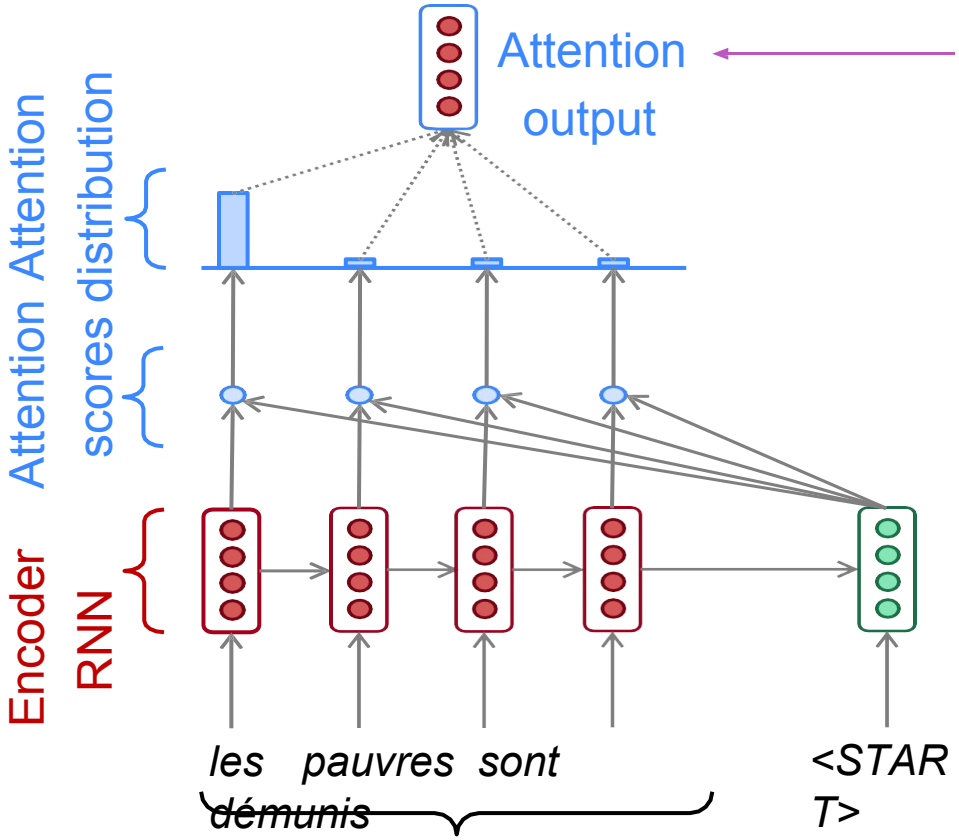
Sequence-to-sequence with attention



Sequence-to-sequence with attention

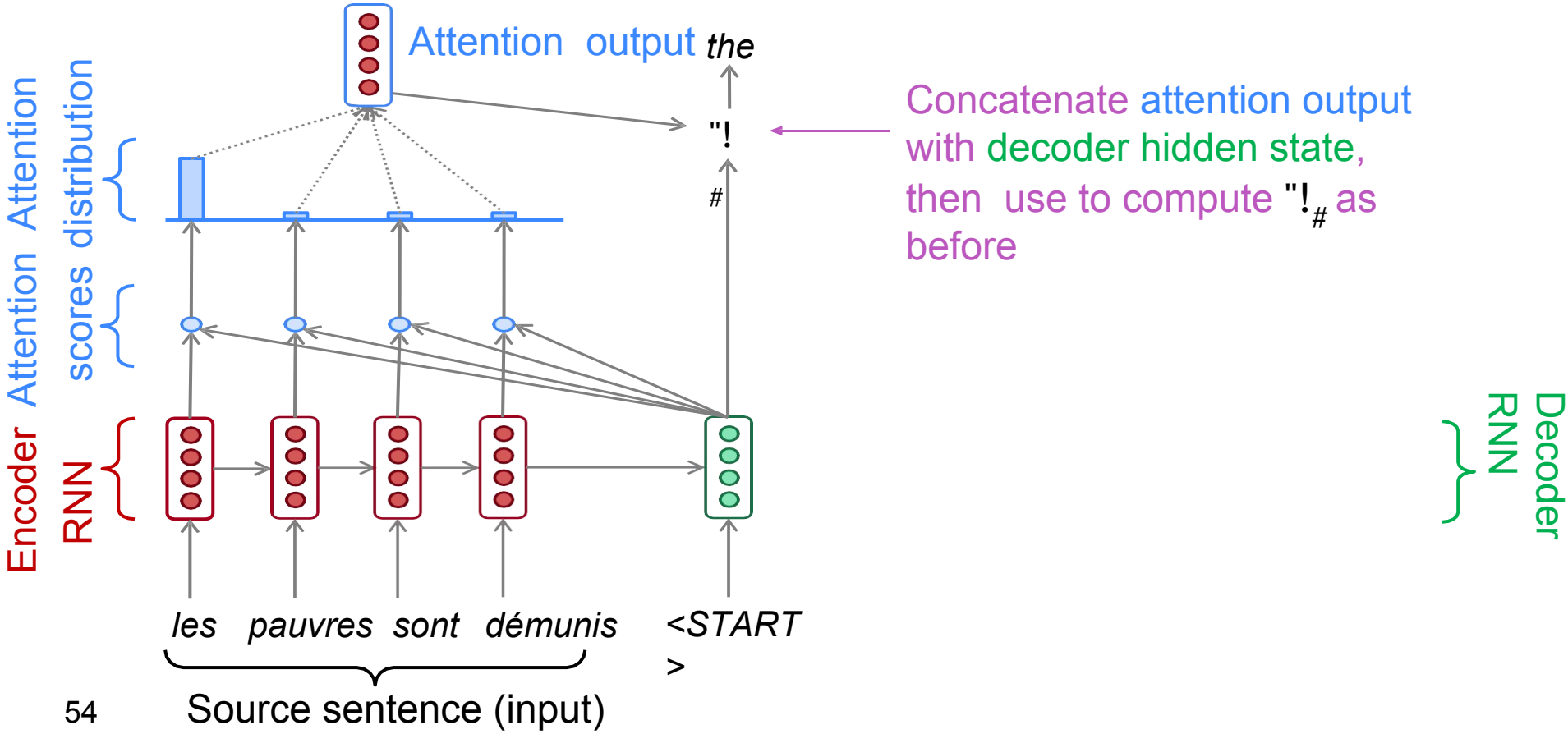


Sequence-to-sequence with attention

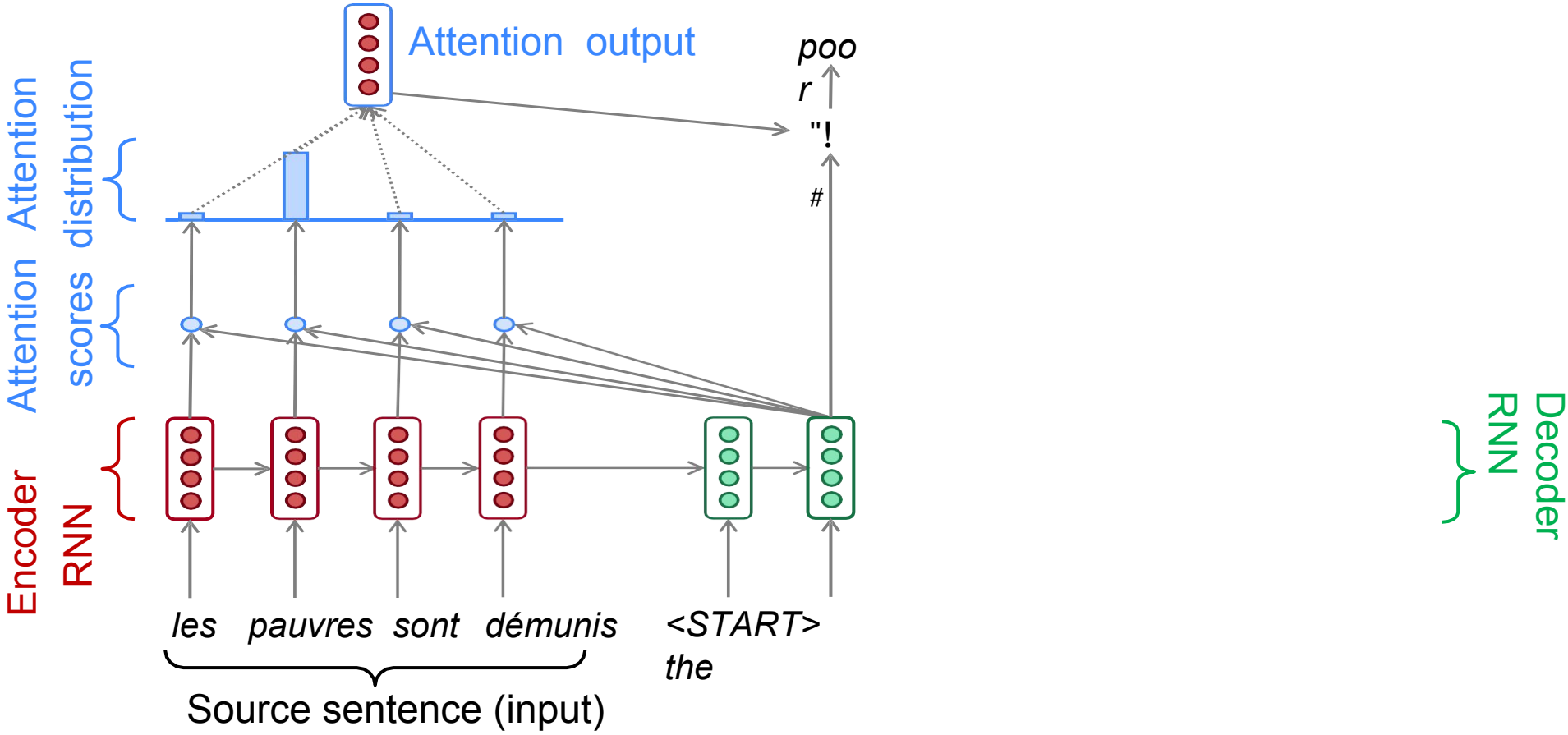


Use the attention distribution to take a **weighted sum** of the encoder hidden states.
The attention output mostly contains information the hidden states that received high attention.

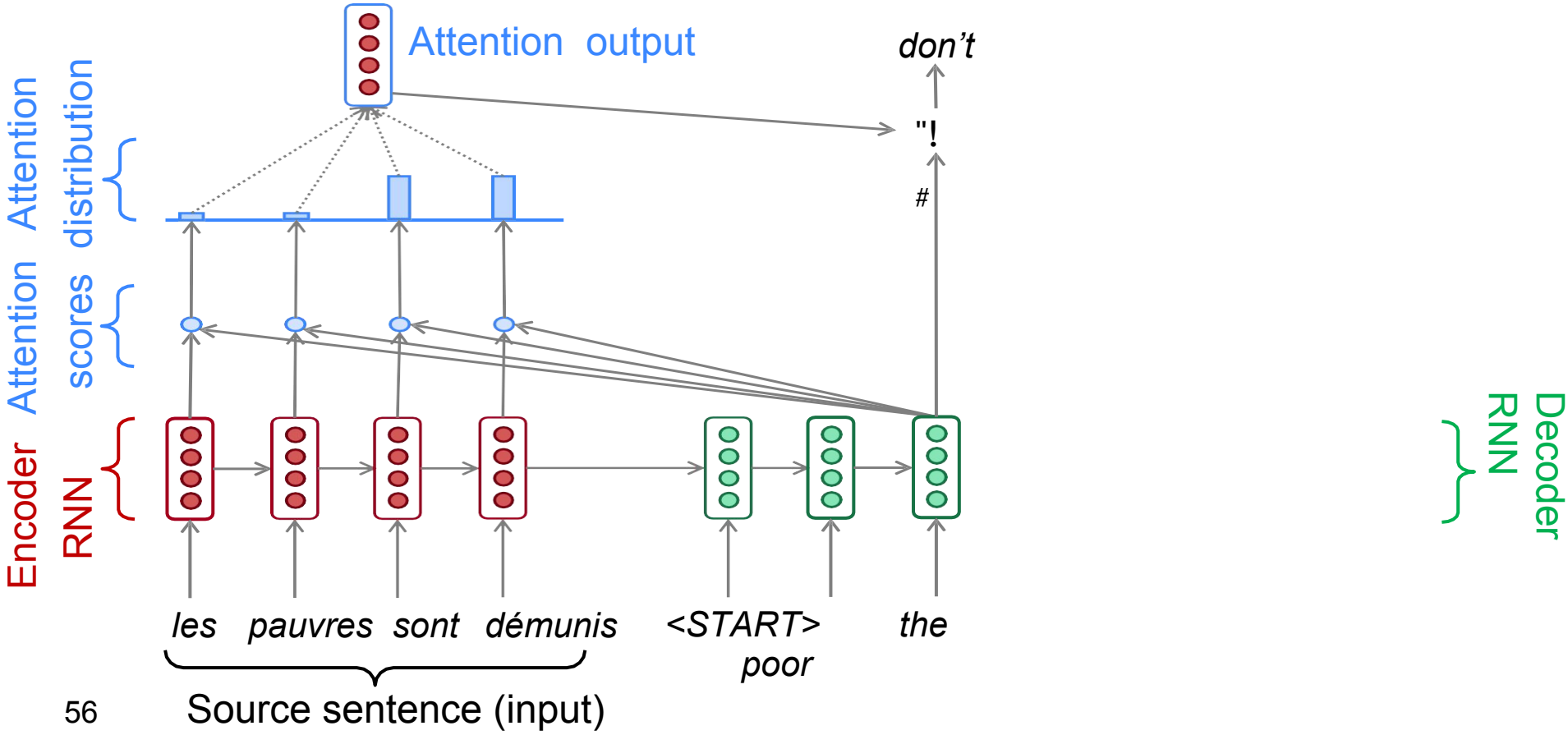
Sequence-to-sequence with attention



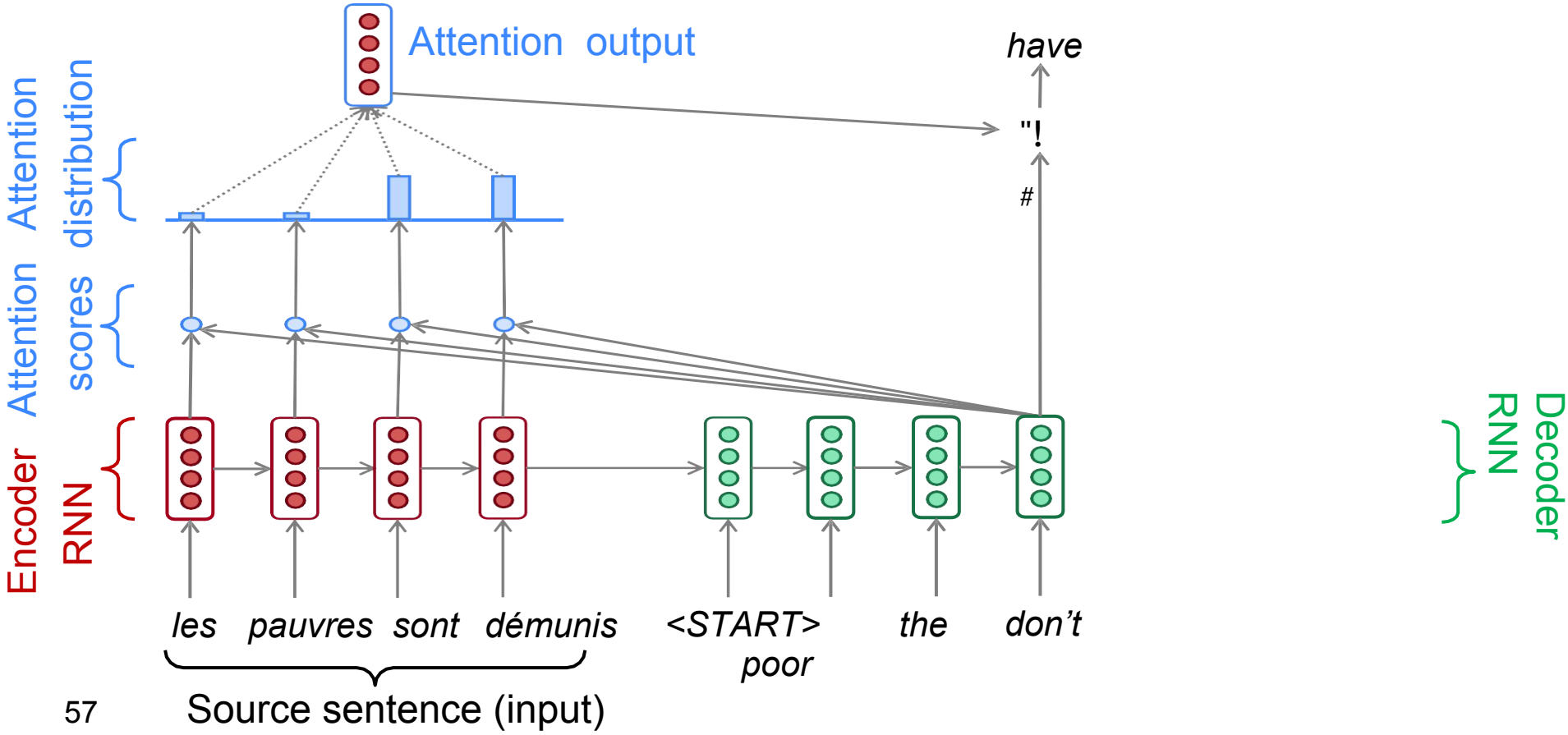
Sequence-to-sequence with attention



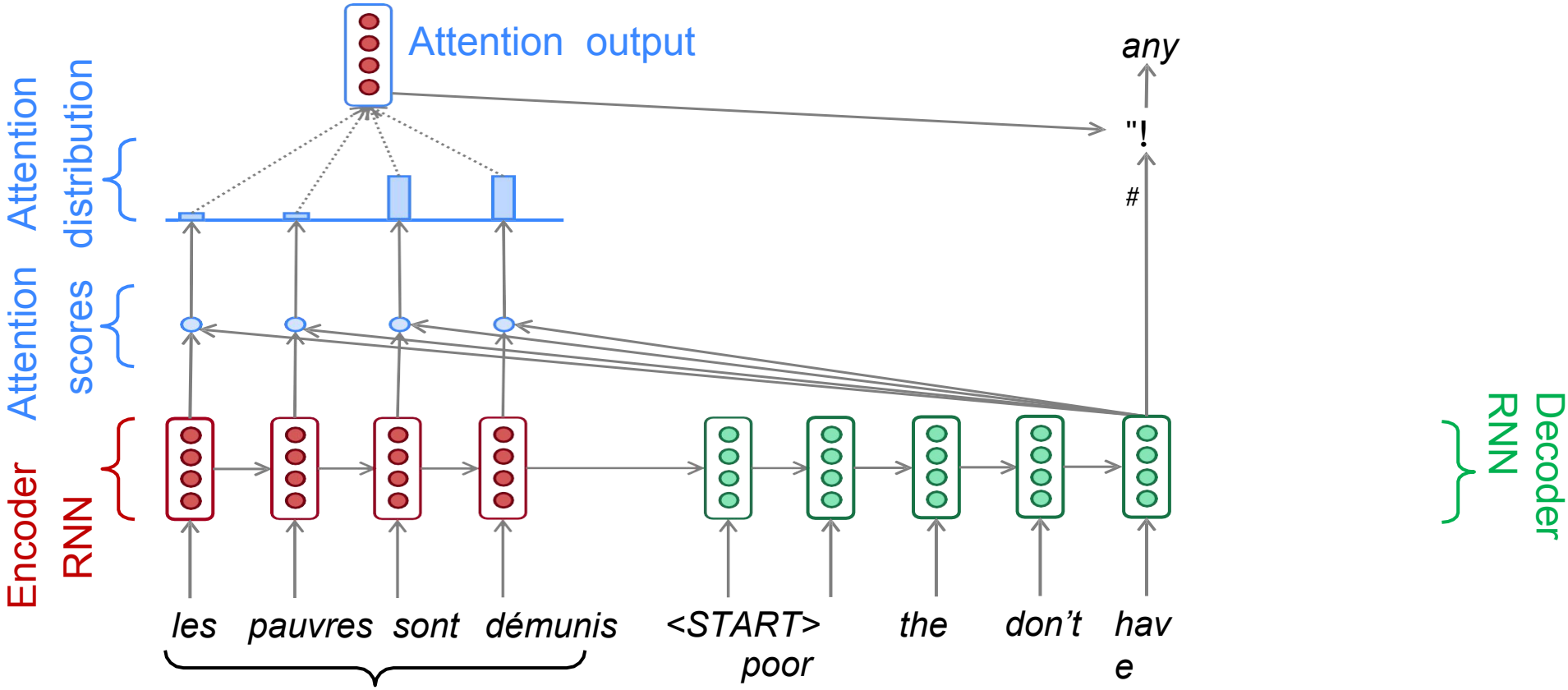
Sequence-to-sequence with attention



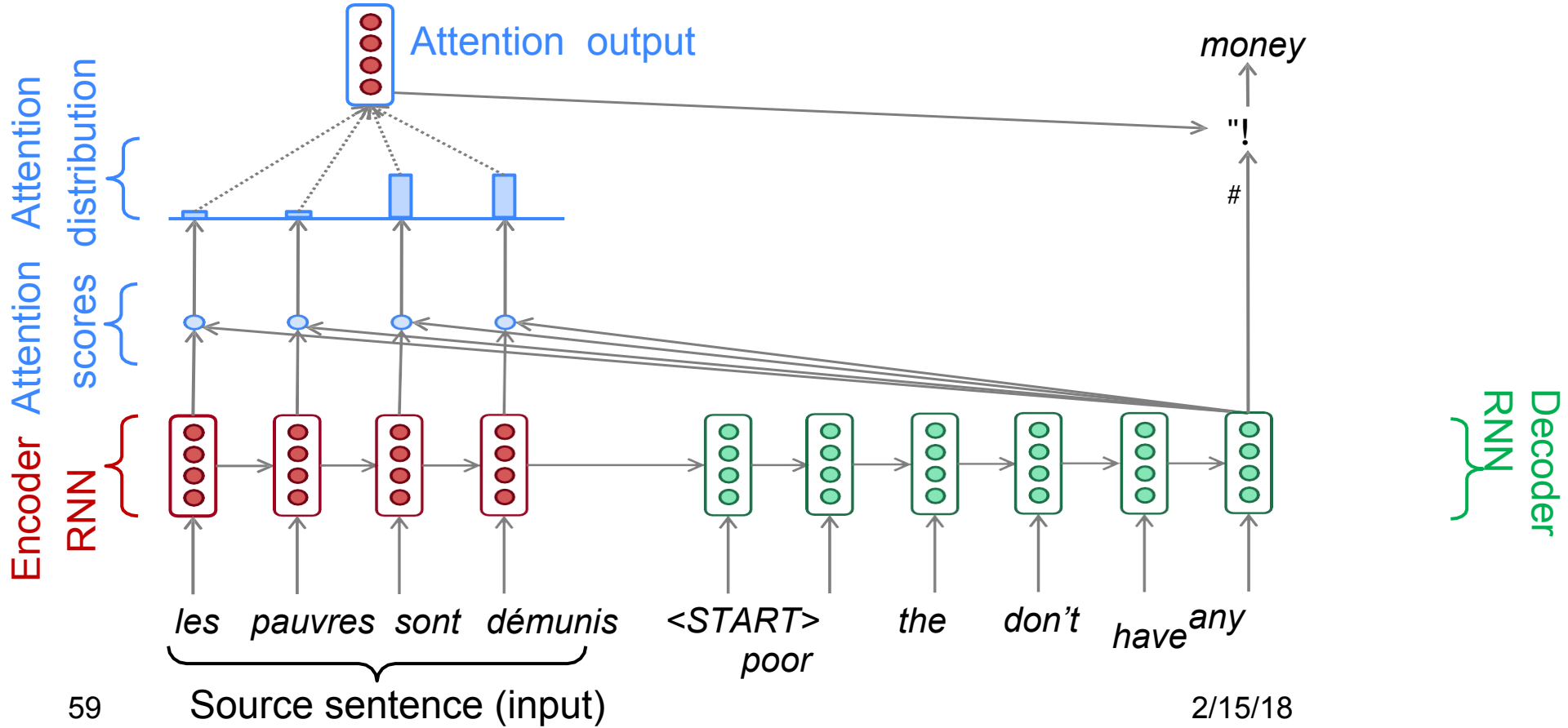
Sequence-to-sequence with attention



Sequence-to-sequence with attention



Sequence-to-sequence with attention



Attention: in equations

- We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$
- On timestep *t*, we have decoder hidden state $s_t \in \mathbb{R}^h$ e^t $e^t = [s_t^T h_1, \ldots, s_t^T h_N] \in \mathbb{R}^N$
- We get the attention scores for this step: α^t
- We take softmax to get the attention for this step (this is a distribution probability distribution and sums to 1)

$$\alpha^t = \text{softmax}(e^t) \in \mathbb{R}^N$$

- We use α^t to take a weighted sum of the encoder hidden states to get the a_t attention output

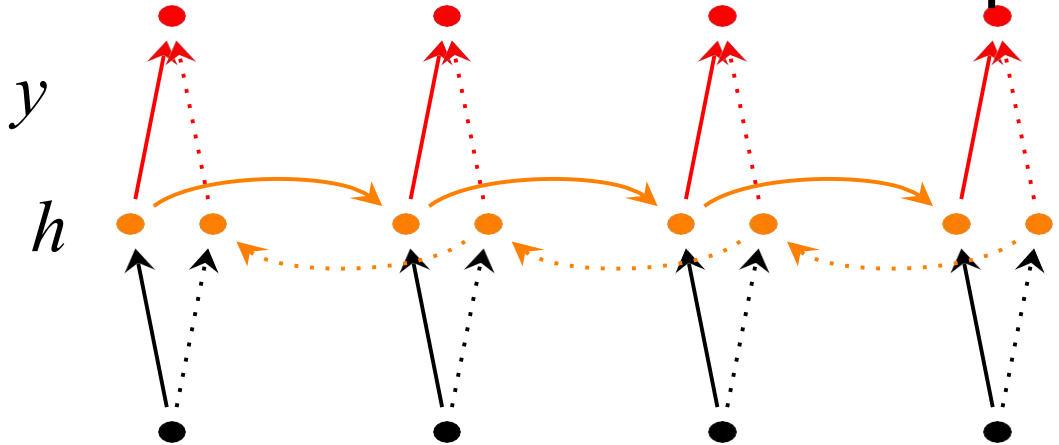
$$a_t = \sum_{i=1}^{N} \alpha_i^t h_i \in \mathbb{R}^h$$

- Finally we concatenate the attention output a_t with the decoder hidden state s_t and proceed as in the non-attention seq2seq model

$$[a_t; s_t] \in \mathbb{R}^{2h}$$

Bidirectional RNNs

Problem: For classification you want to incorporate information from words both preceding and following

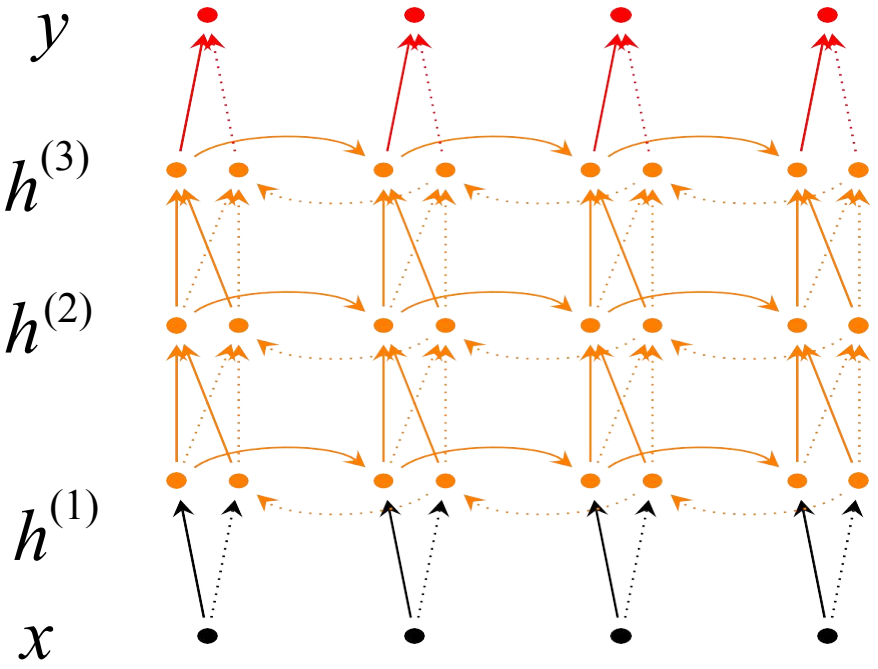


$$h_t = f(Wx_t + Vh_{t+1} + b)$$

$$y_t = g(U[h_t; h_t] + c)$$

$h = [h; h]$ now represents (summarizes) the past and future around a single token.

Deep Bidirectional RNNs



$$\overrightarrow{h}_t^{(i)} = f(\overrightarrow{W}^{(i)} h_t^{(i-1)} + \overrightarrow{V}^{(i)} \overrightarrow{h}_{t-1}^{(i)} + \overrightarrow{b}^{(i)})$$

$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)})$$

$$y_t = g(U[\overrightarrow{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c)$$

Each memory layer passes an intermediate sequential representation to the next.

Administrative

- A3, **Machine Translation** will come out at the end of week at worst

!!! Lecture Feedback https://goo.gl/forms/zeZiu1fSgrpPGp6T2 !!!

Good courses/books

- edu.ipavlov.ai
- udacity.com/course/deep-learning–ud730
- deeplearningbook.org
- github.com/oxford-cs-deepnlp-2017/lectures
- cs224d.stanford.edu/