SFU NatLangLab

CMPT 413/713: Natural Language Processing

Recurrent Neural Networks
LSTM and GRUs

How to model sequences using neural networks?

Spring 2024
2024-02-05

Adapted from slides from Danqi Chen, Karthik Narasimhan, and Justin Johnson

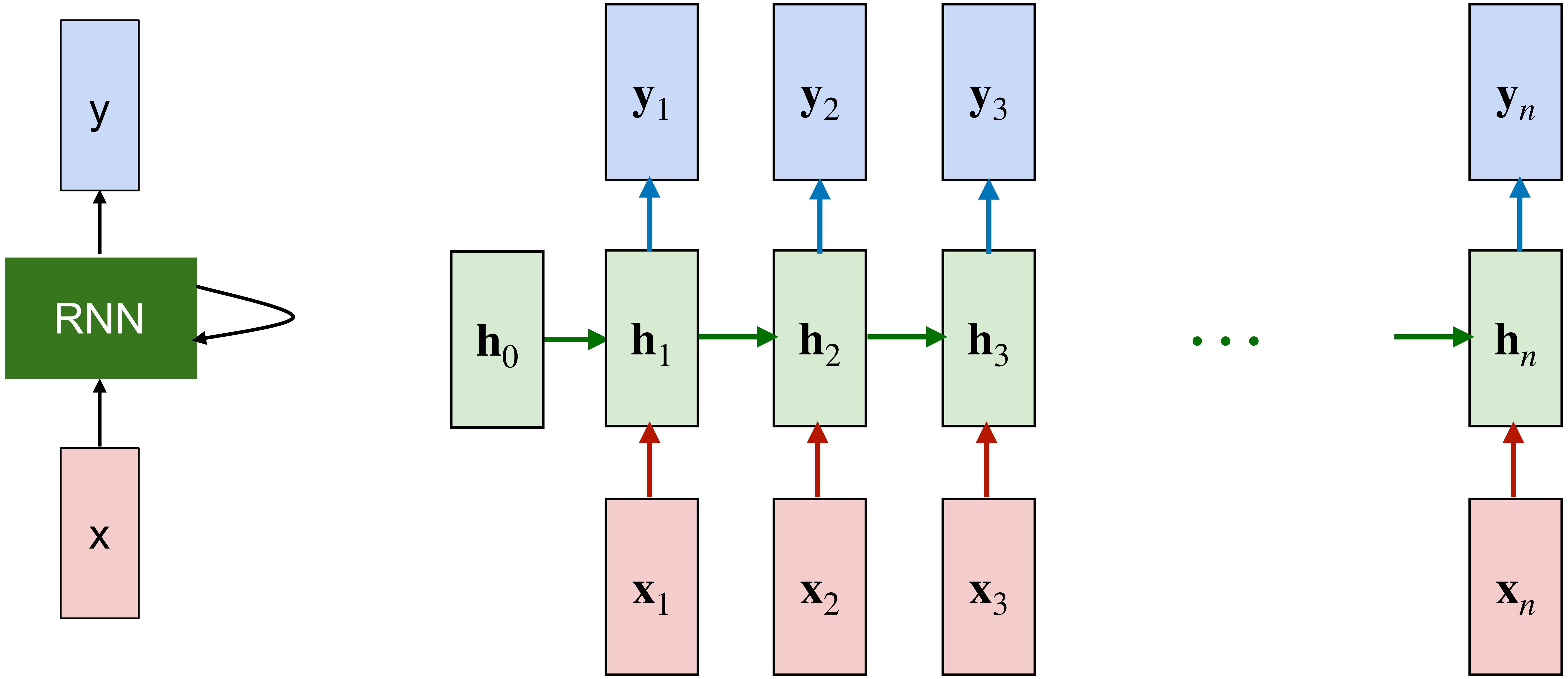
(Some slides adapted from Chris Manning, Abigail See, Andrej Karpathy)

Overview

- Review of Vanilla RNN
- Training RNNs
- Issues with Gradient Flows
- LSTMs and GRUs
- Applications
- Variants: Stacked RNNs, Bidirectional RNNs

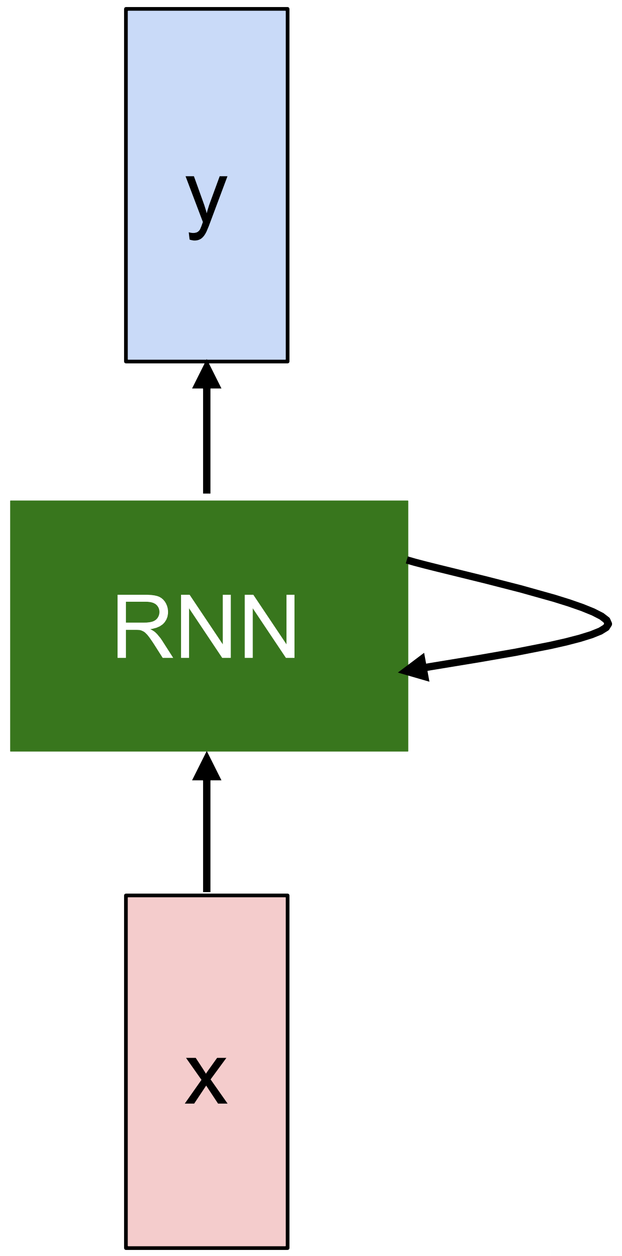
Simple RNNs

Recurrent Neural Networks (RNNs)



Structure of cell and weights are shared across time steps

Simple (vanilla) RNNs



$\mathbf{h}_0 \in \mathbb{R}^d$ is an initial state

function with weights W

$$\boxed{\mathbf{h}_t = f_{\mathbf{W}}(\mathbf{h}_{t-1}, \mathbf{x}_t) \in \mathbb{R}^d}$$

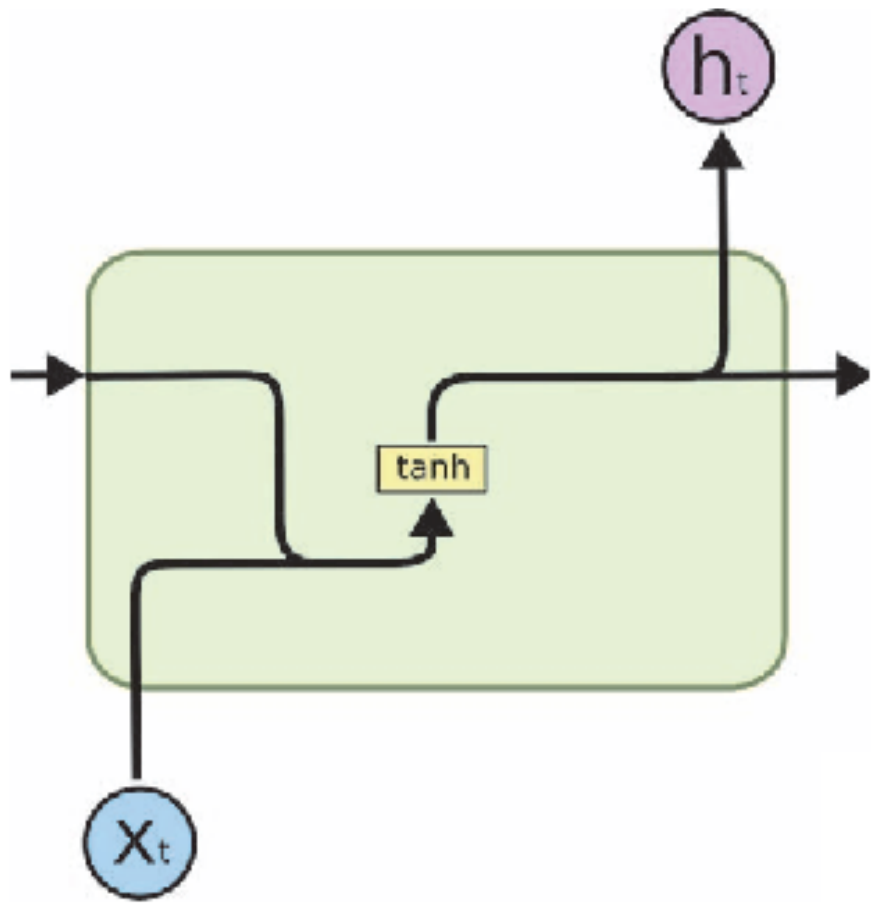
new state　old state　input at time t

$\mathbf{h}_t$: hidden states which store information from $\mathbf{x}_1$ to $\mathbf{x}_t$

Output label for each time step: Denote $\hat{\mathbf{y}}_t = \mathrm{softmax}(\mathbf{W}_o \mathbf{h}_t)$, $\mathbf{W}_o \in \mathbb{R}^{|L| \times d}$

Simple (vanilla) RNNs:

$$\boxed{\mathbf{h}_t = g(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}) \in \mathbb{R}^d}$$



g: nonlinearity (e.g. tanh),

$\mathbf{W}_h \in \mathbb{R}^{d \times d}$, $\mathbf{W}_x \in \mathbb{R}^{d \times d_{in}}$, $\mathbf{b} \in \mathbb{R}^d$

RNN Language Model

Output label size: |V|

output distribution

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

hidden states

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

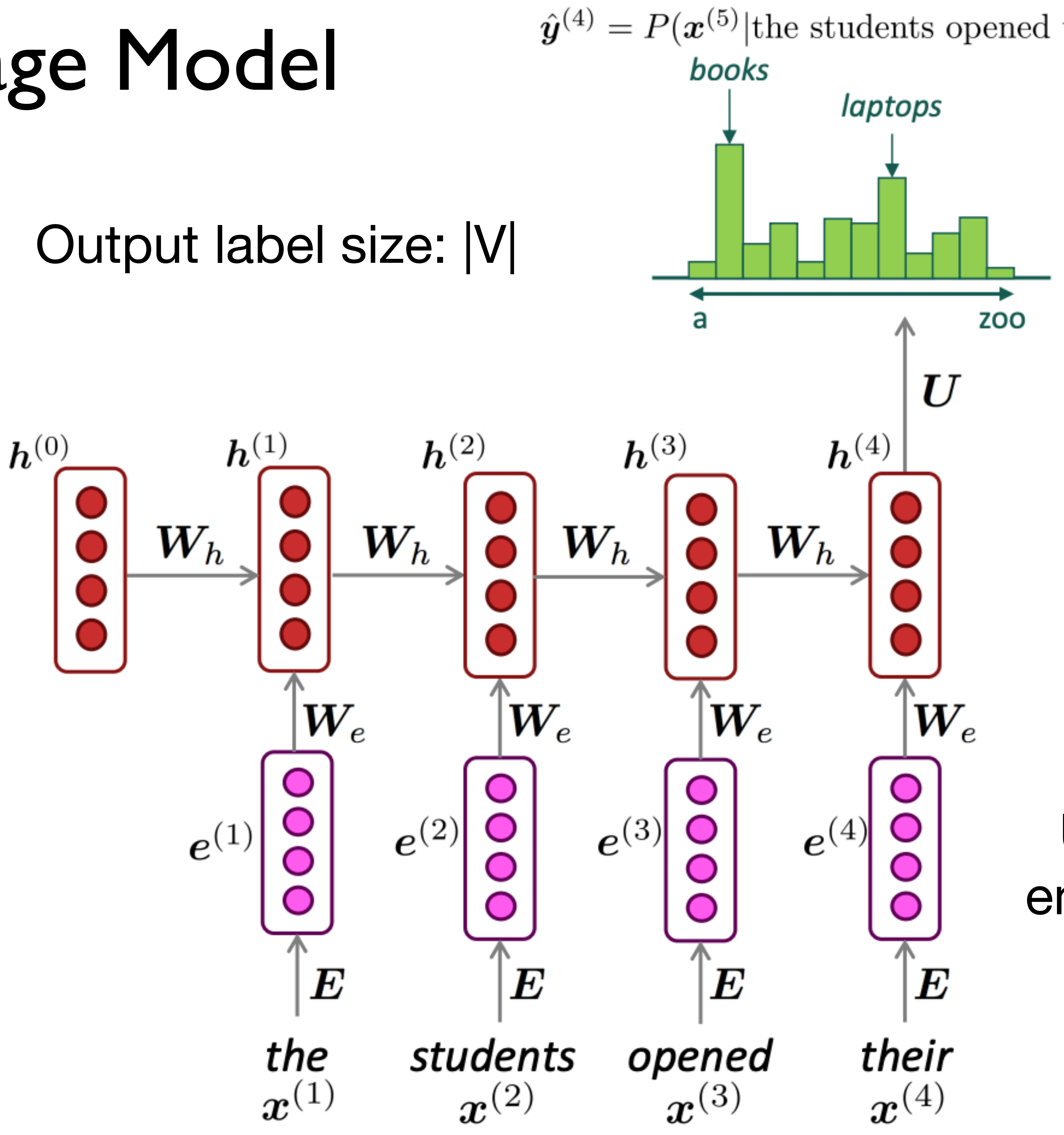
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

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

words / one-hot vectors

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



Use word embeddings

Slide credit: Chris Manning

Training the RNN

Training an RNN Language Model

- Get a big corpus of text (sequence of words $x^{(1)}, \dots, x^{(T)}$)

- Feed into RNN-LM and compute output distribution $\hat{y}^{(t)}$ for every step t (i.e predict for every word, given words so far)

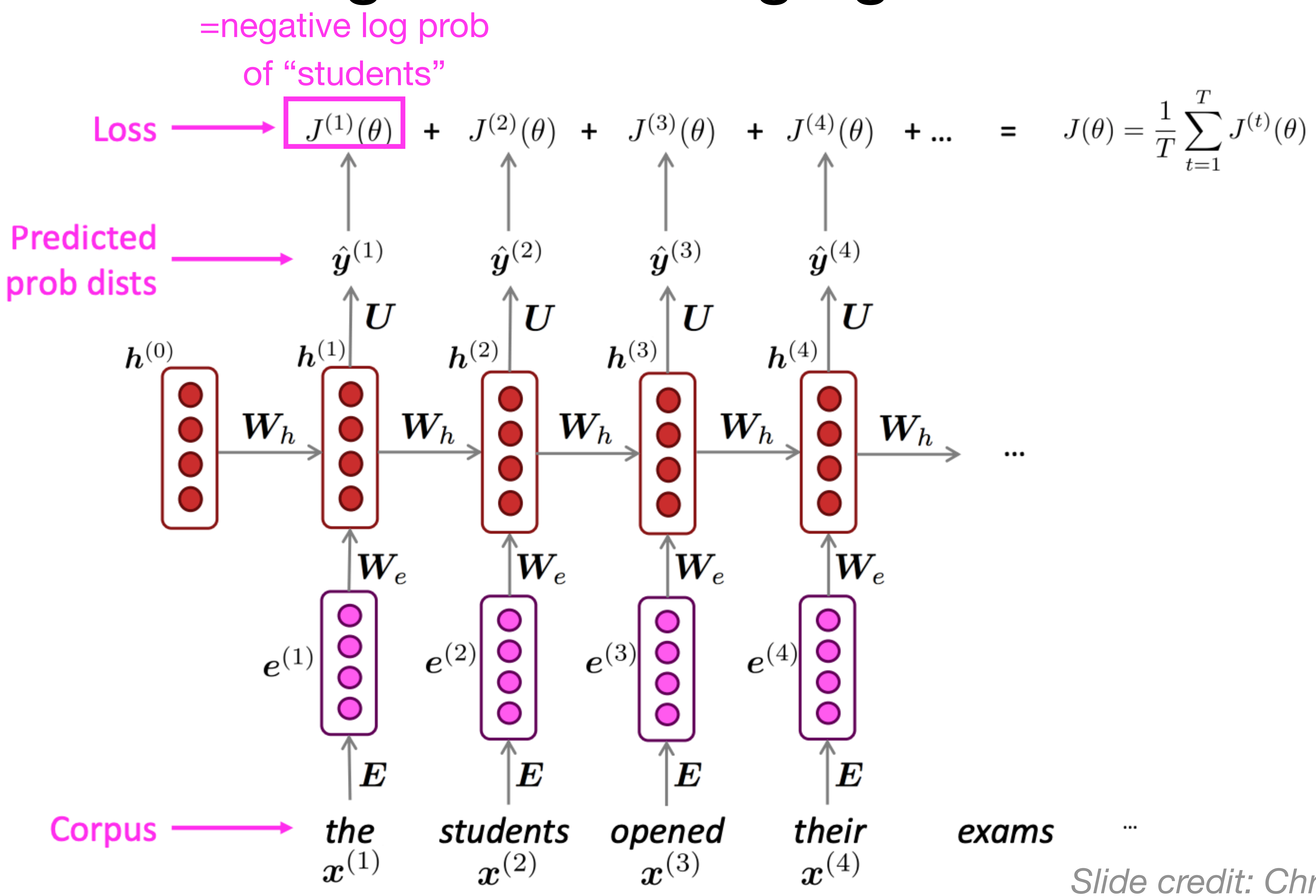
- Loss function on step t is cross-entropy between predicted probability distribution $\hat{y}^{(t)}$, and the true next word $y^{(t)}$ (one-hot for $x^{(t+1)}$)

$$J^{(t)}(\theta) = CE(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)}) = -\sum_{w \in V} \mathbf{y}_w^{(t)} \log \hat{\mathbf{y}}_w^{(t)} = -\log \hat{\mathbf{y}}_{x_{t+1}}^{(t)}$$

- Average to get overall loss for the entire training set

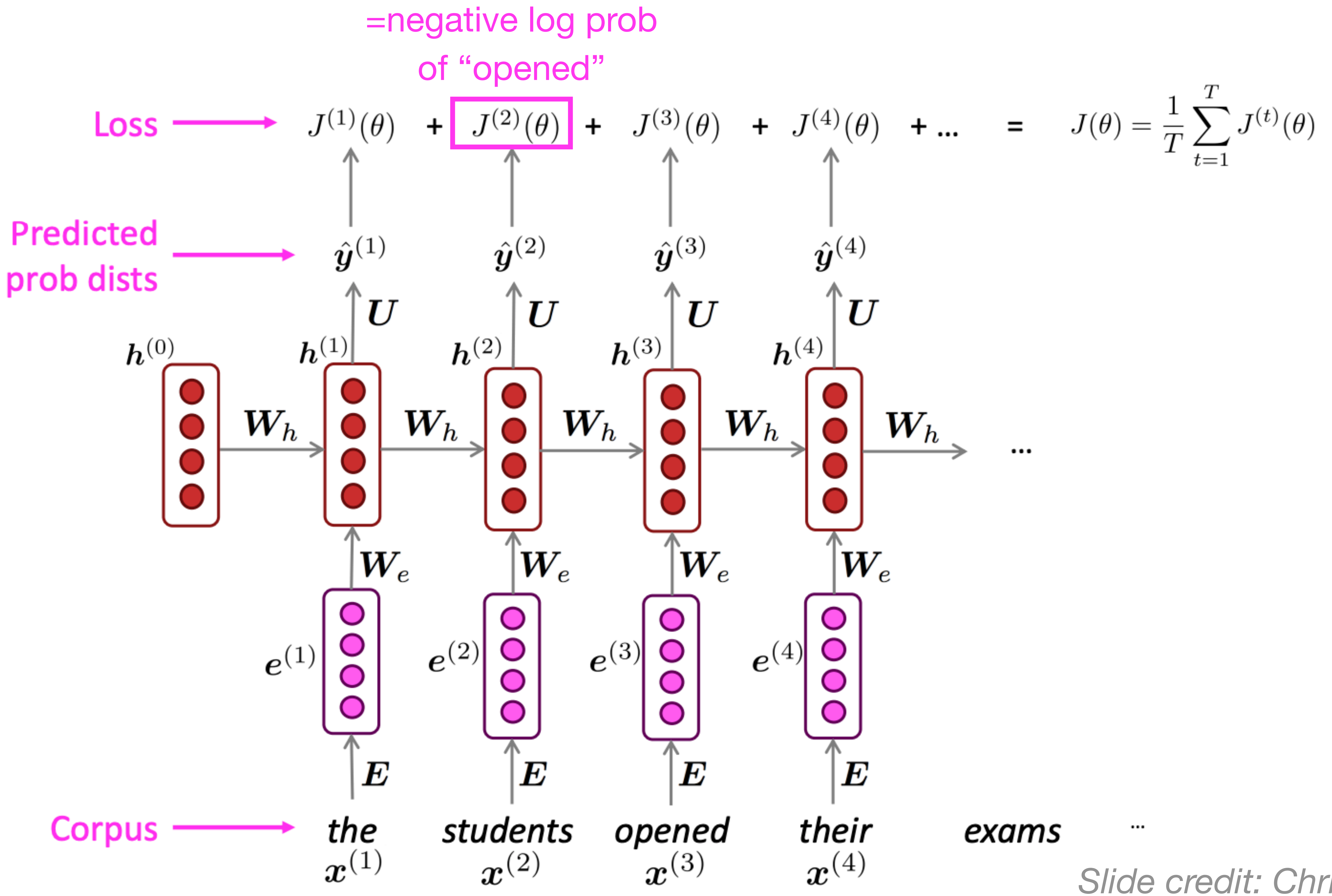
$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \log \hat{\mathbf{y}}_{x_{t+1}}^{(t)}$$

Training an RNN Language Model



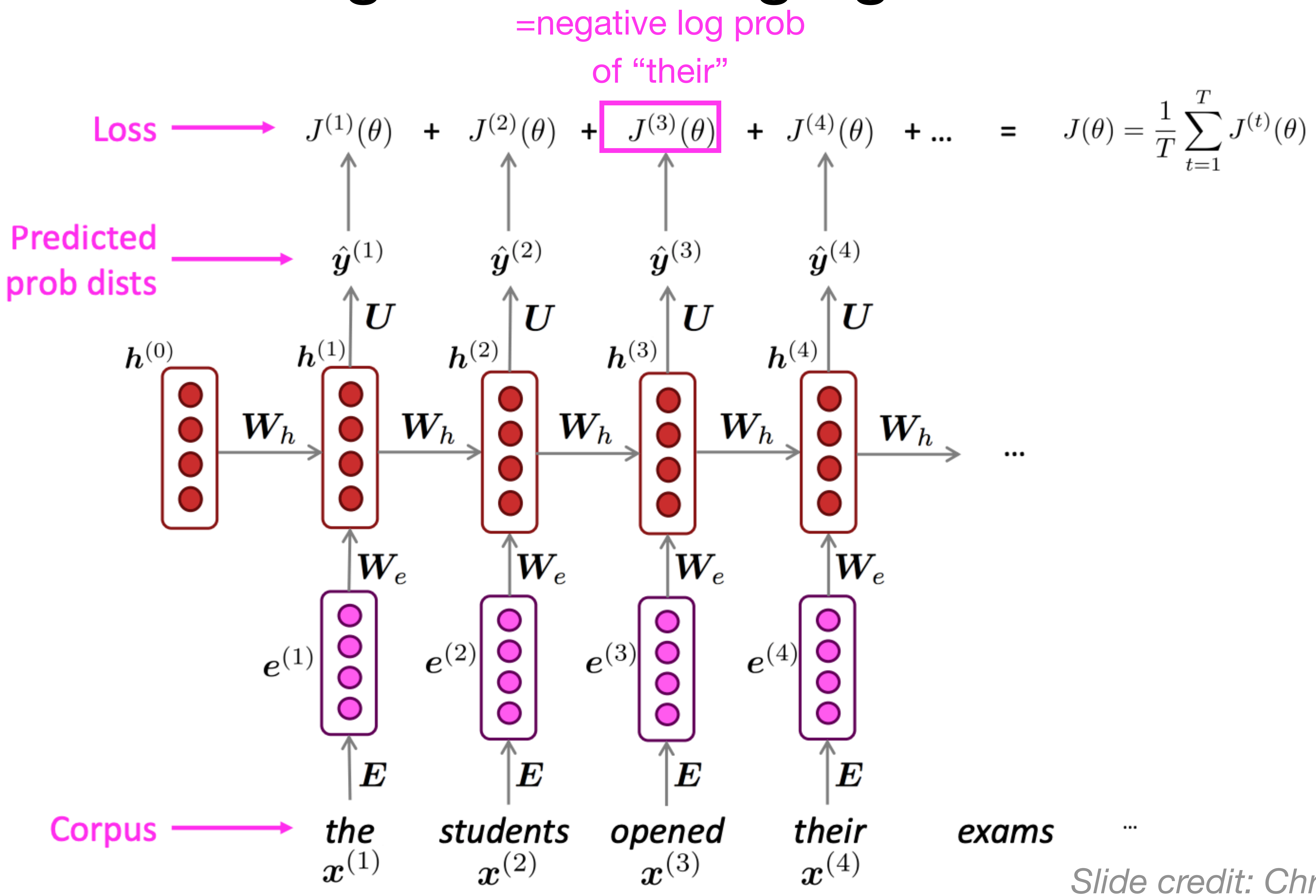
Slide credit: Chris Manning

Training an RNN Language Model



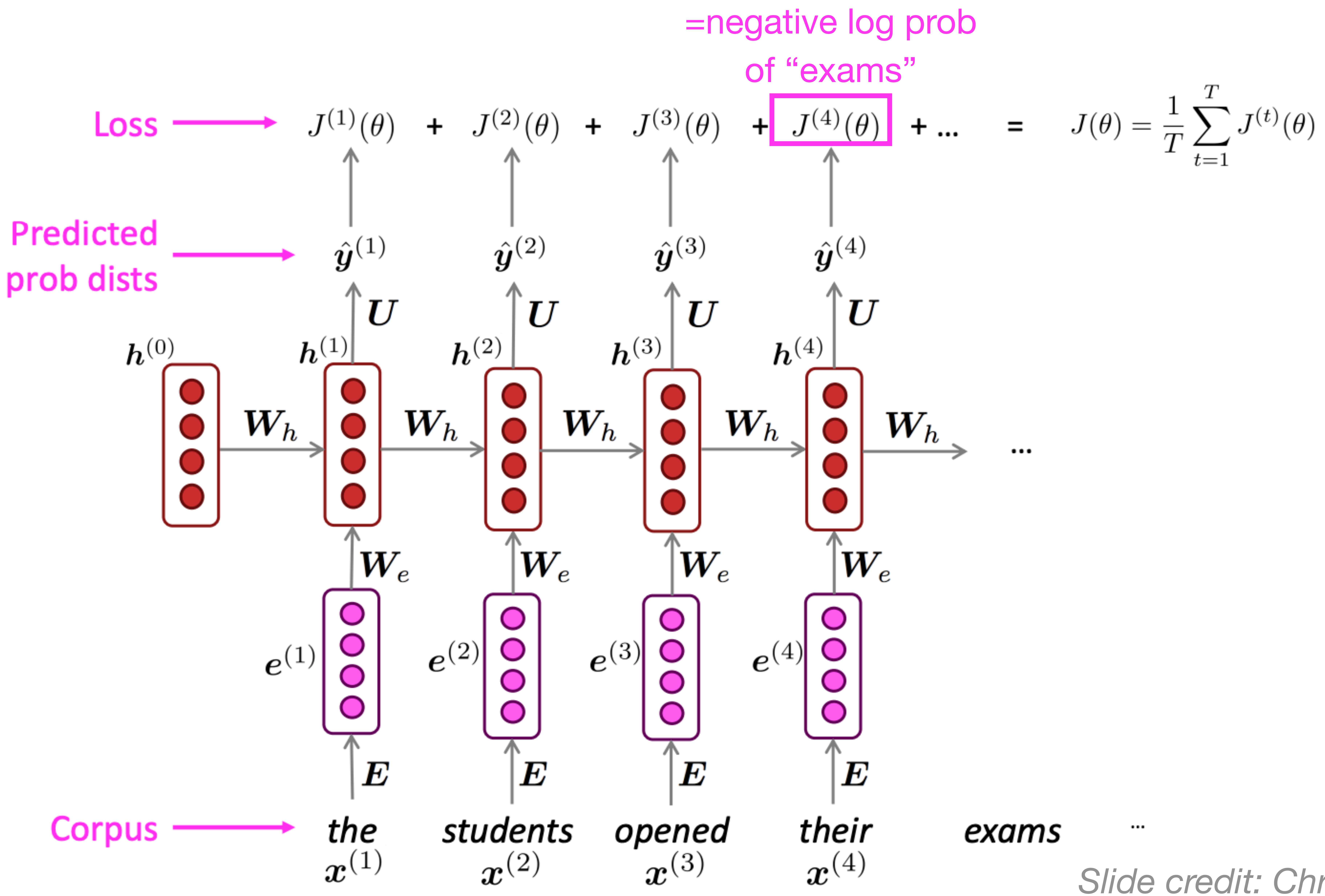
Slide credit: Chris Manning

Training an RNN Language Model



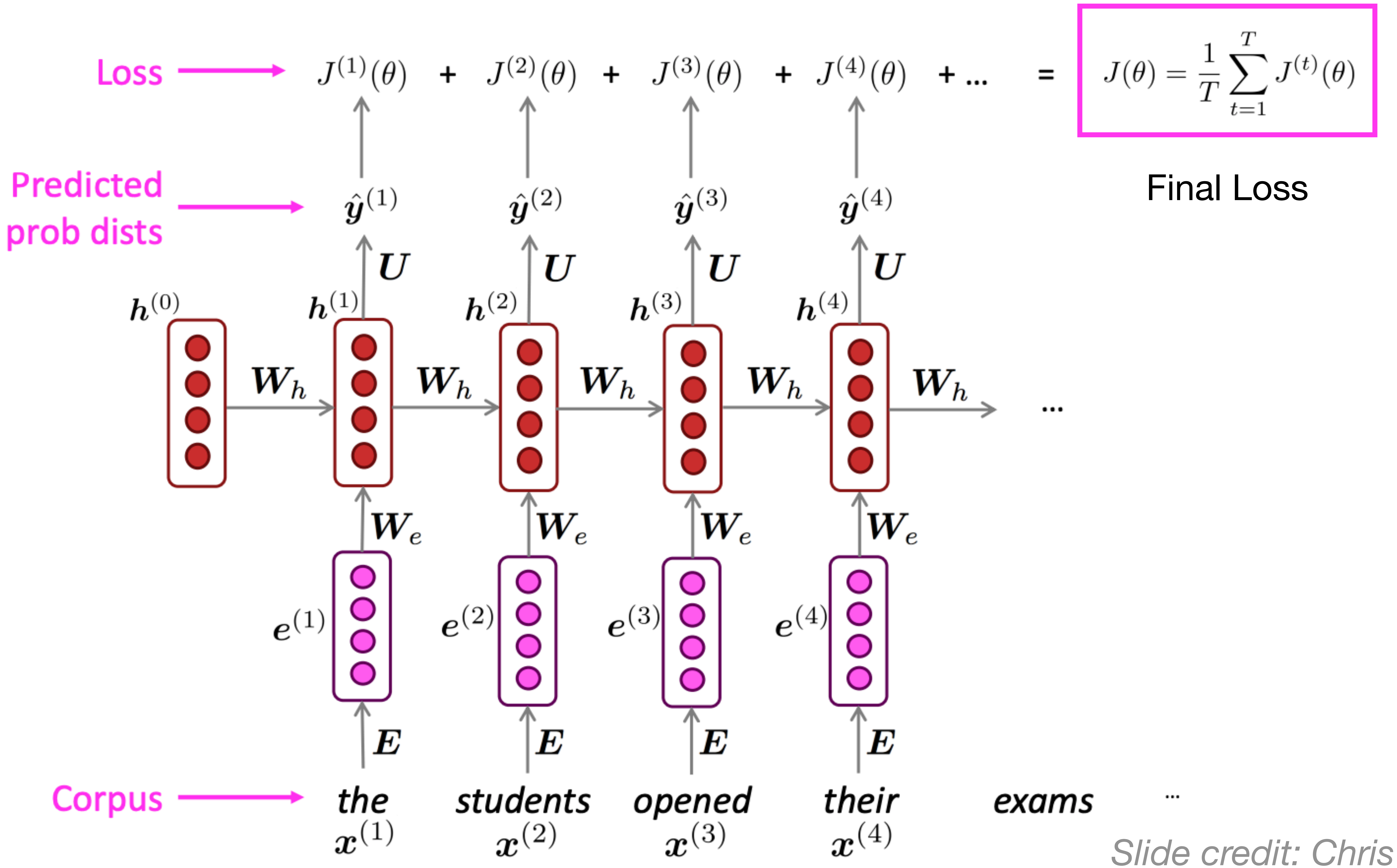
Slide credit: Chris Manning

Training an RNN Language Model



Slide credit: Chris Manning

Training an RNN Language Model

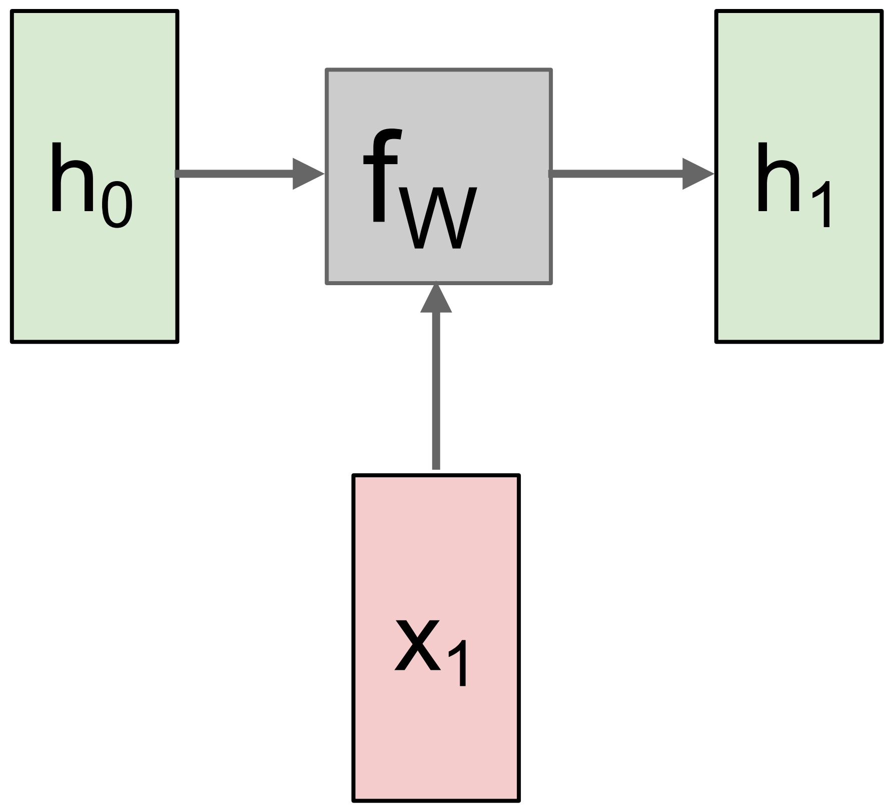


Slide credit: Chris Manning

Training an RNN language Model

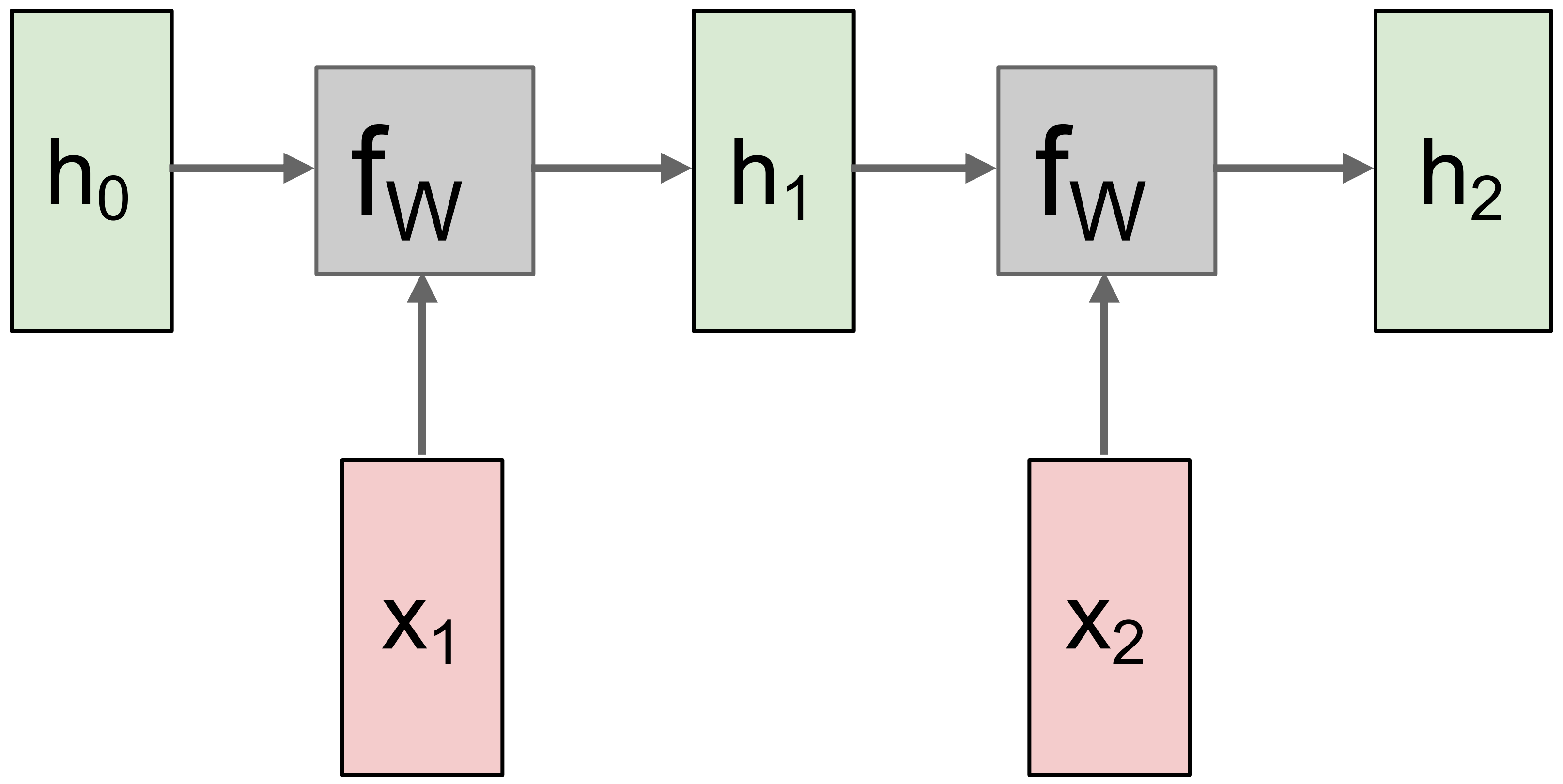
- Note that computing loss and gradients for the whole corpus at once is too expensive
- In practice, consider $x^{(1)}, \ldots, x^{(T)}$ for a sentence (or a document)
- Use batching to parallelize computation over sentences
- Use SGD to estimate parameters
- Use computation graph with backprop

RNN Computation Graph



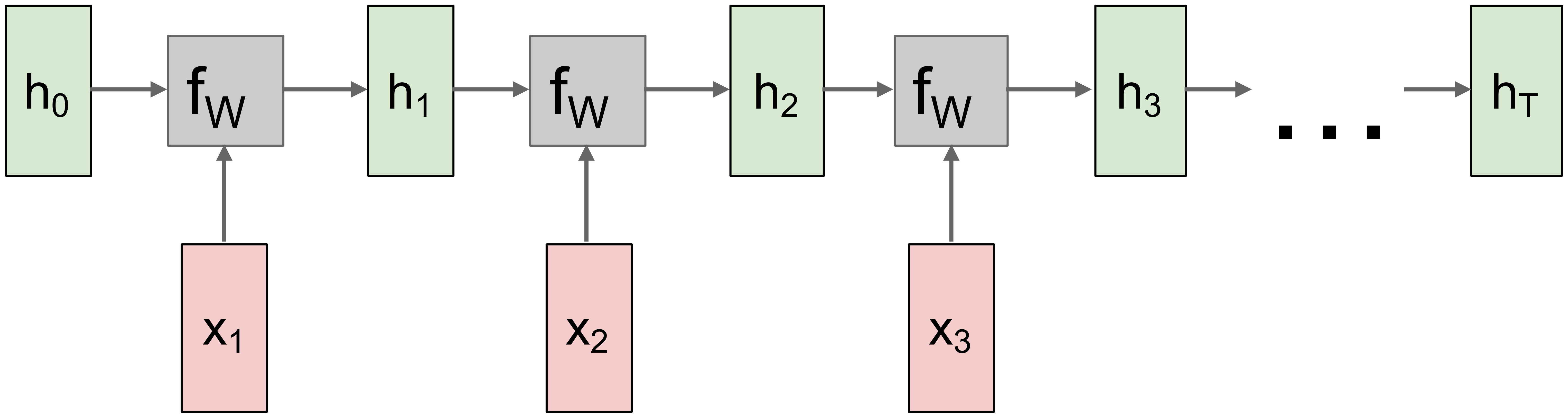
Slide credit: Justin Johnson

RNN Computation Graph



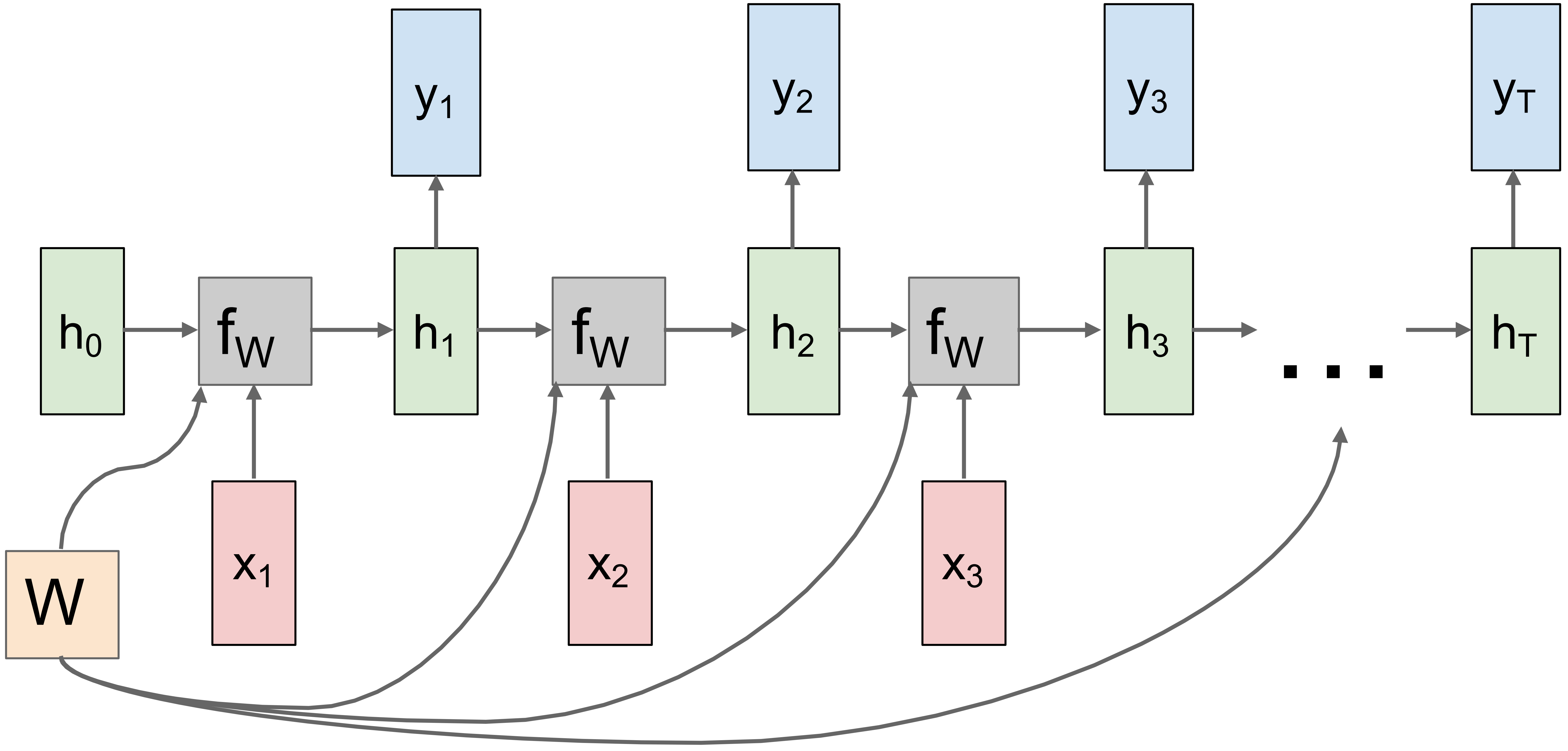
Slide credit: Justin Johnson

RNN Computation Graph



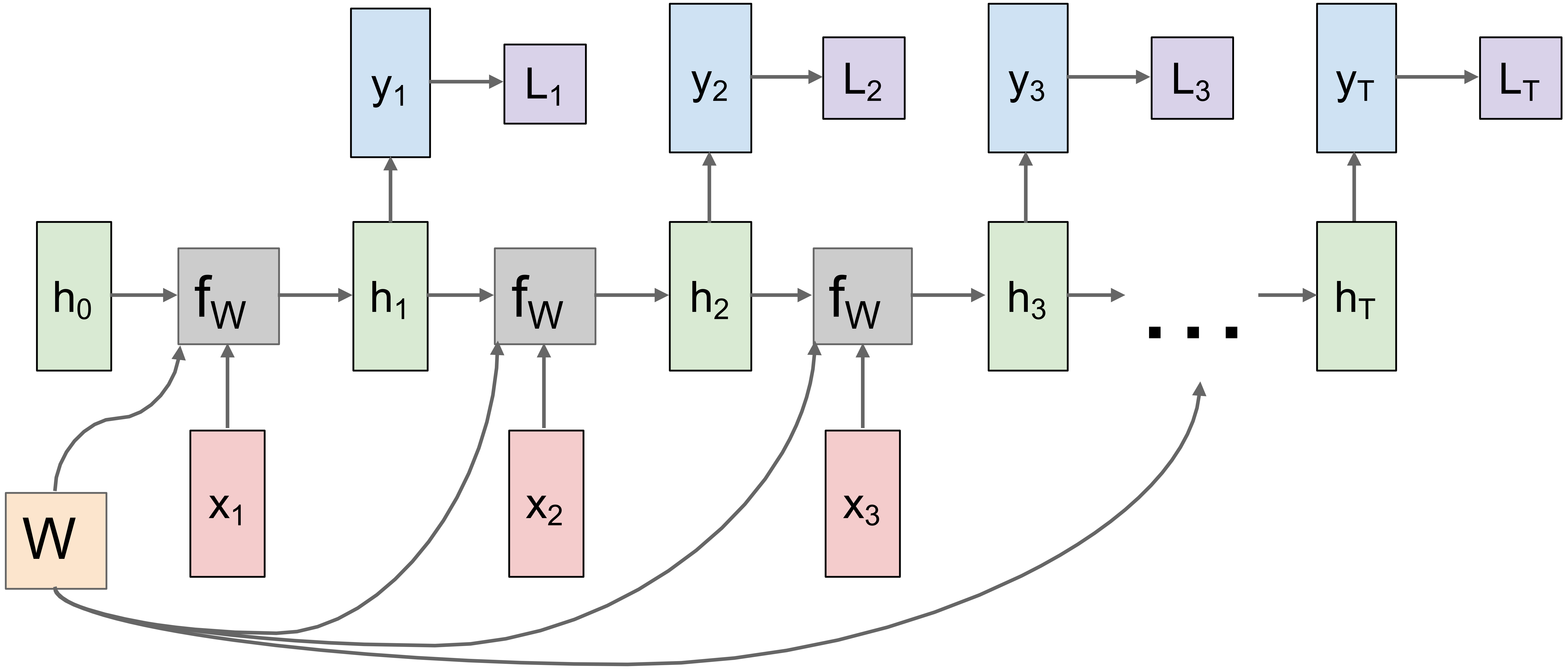
Slide credit: Justin Johnson

RNN Computation Graph



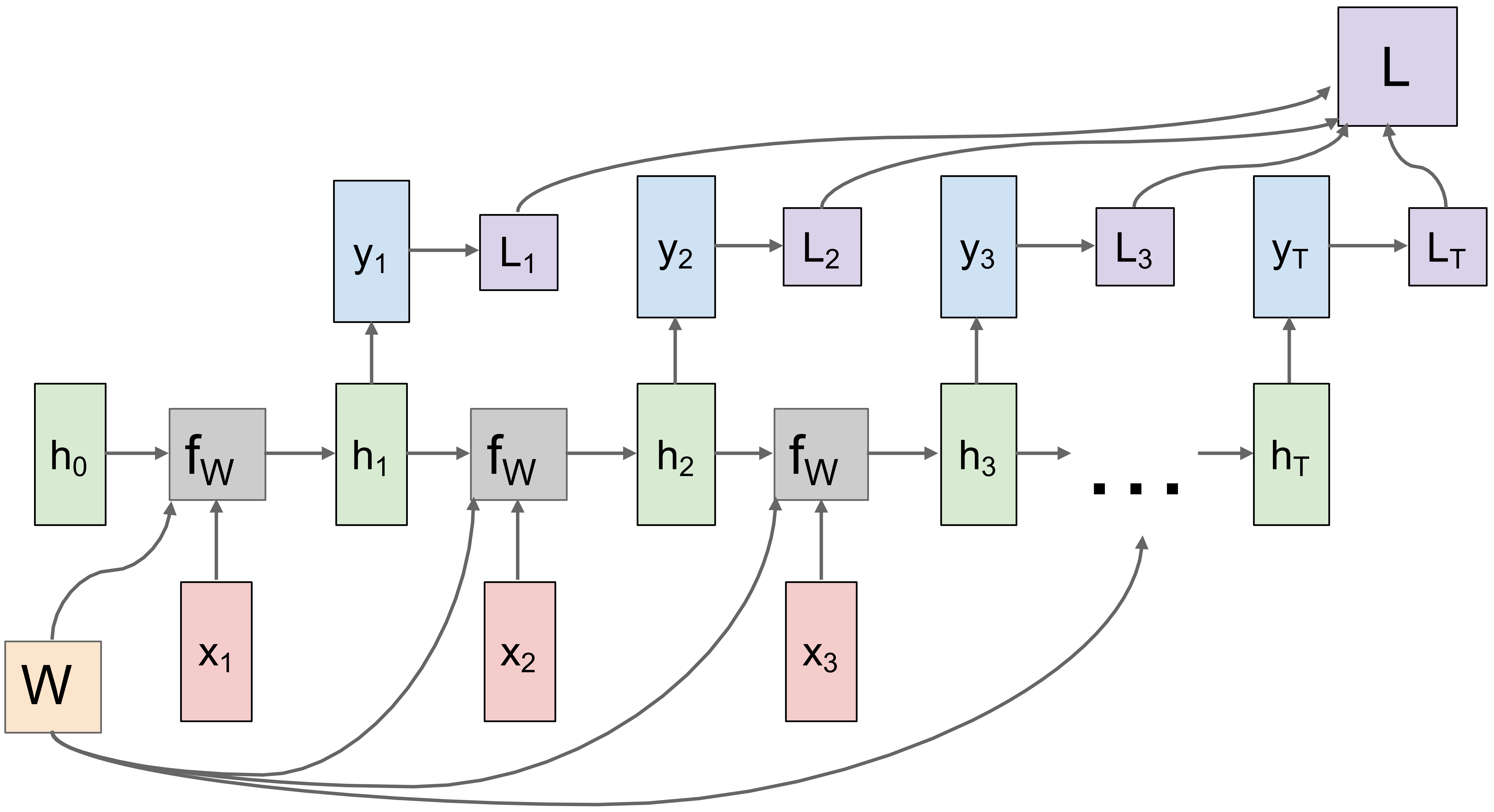
Slide credit: Justin Johnson

RNN Computation Graph



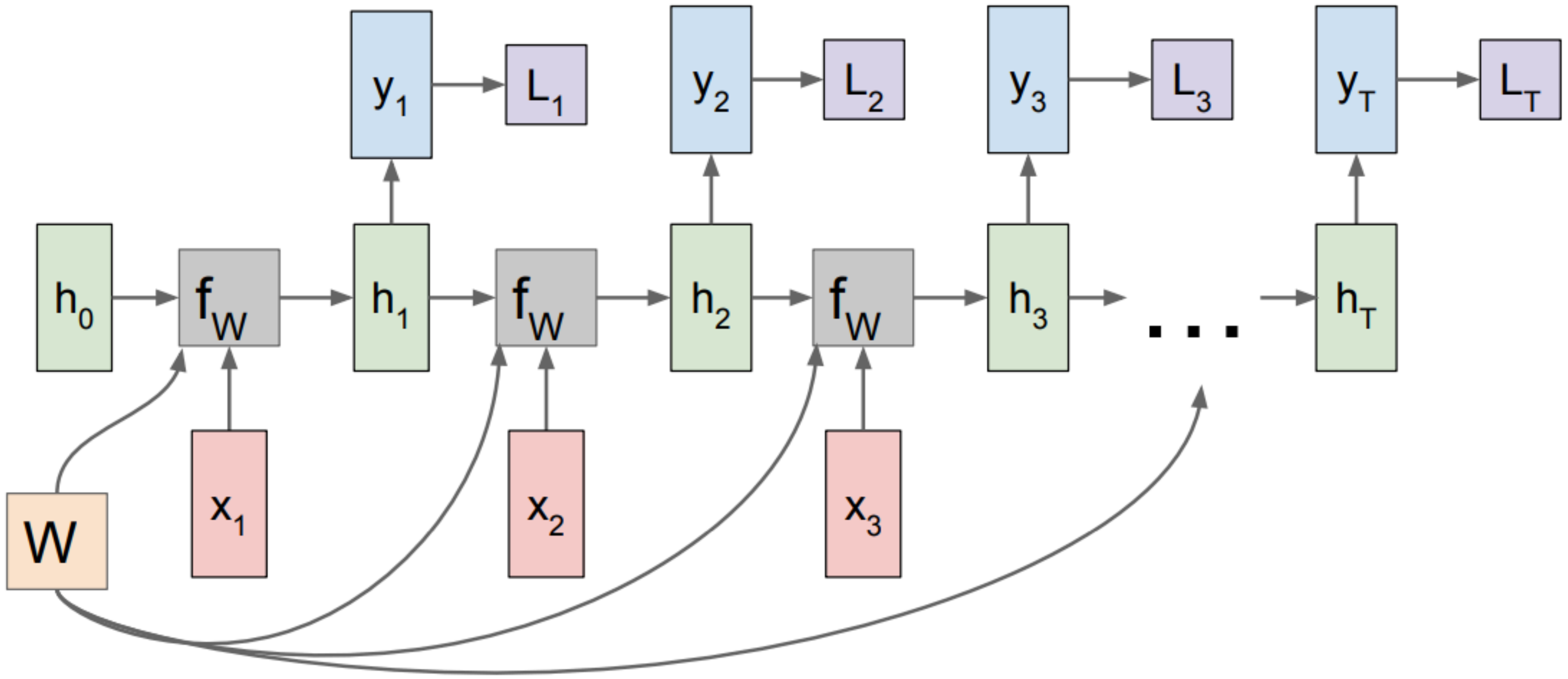
Slide credit: Justin Johnson

RNN Computation Graph



Training RNNLMs

- Backpropagation? Yes, but not that simple!

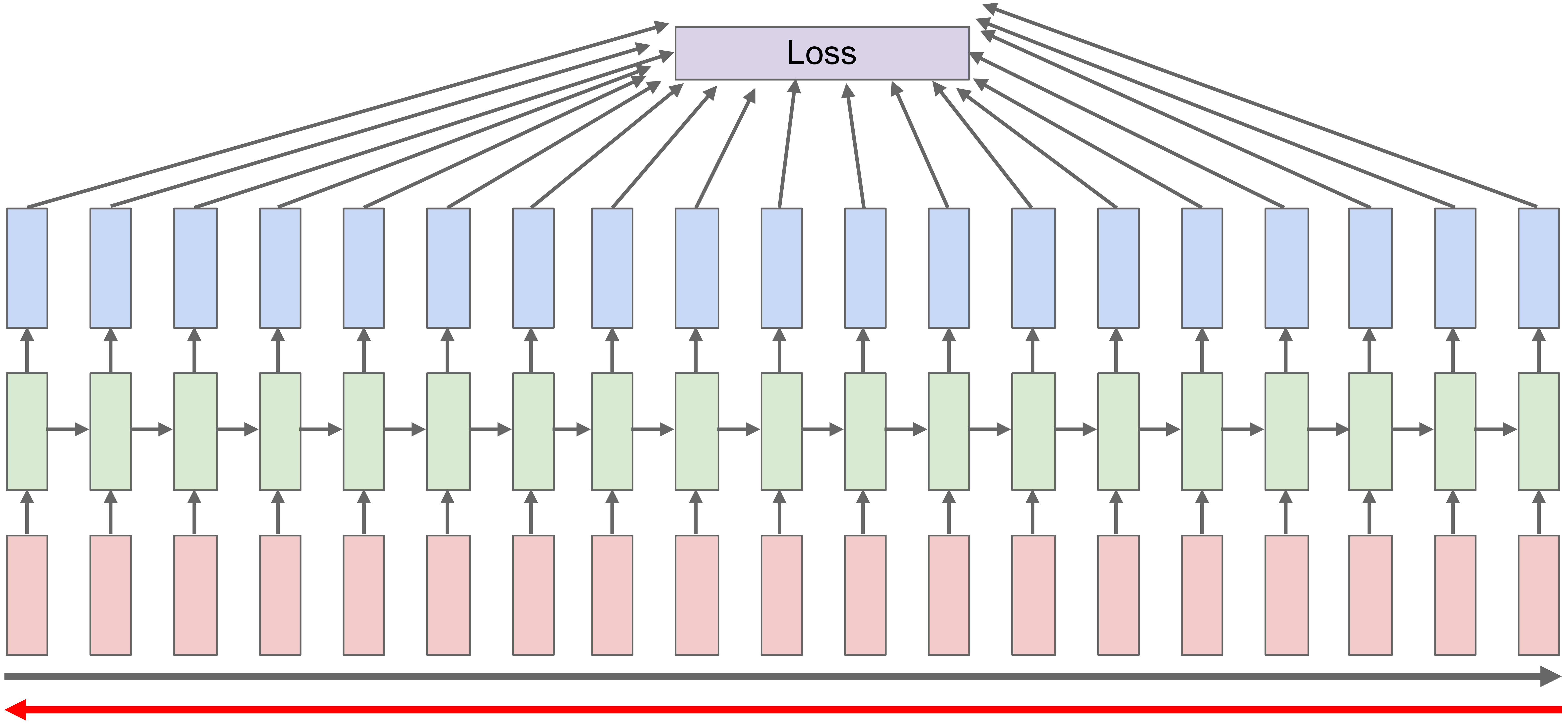


- The algorithm is called Backpropagation Through Time (BPTT).

Slide credit: Justin Johnson

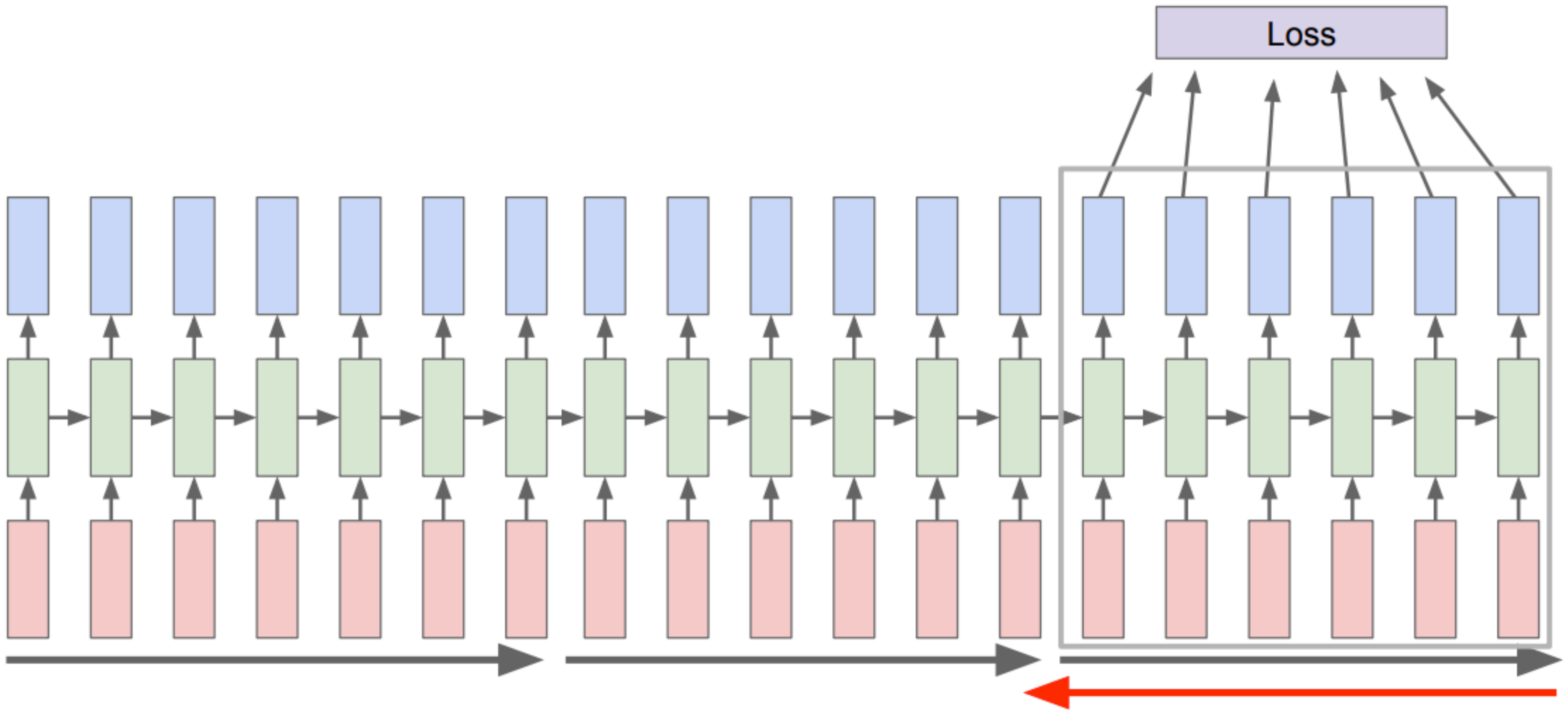
Backpropagation through time

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



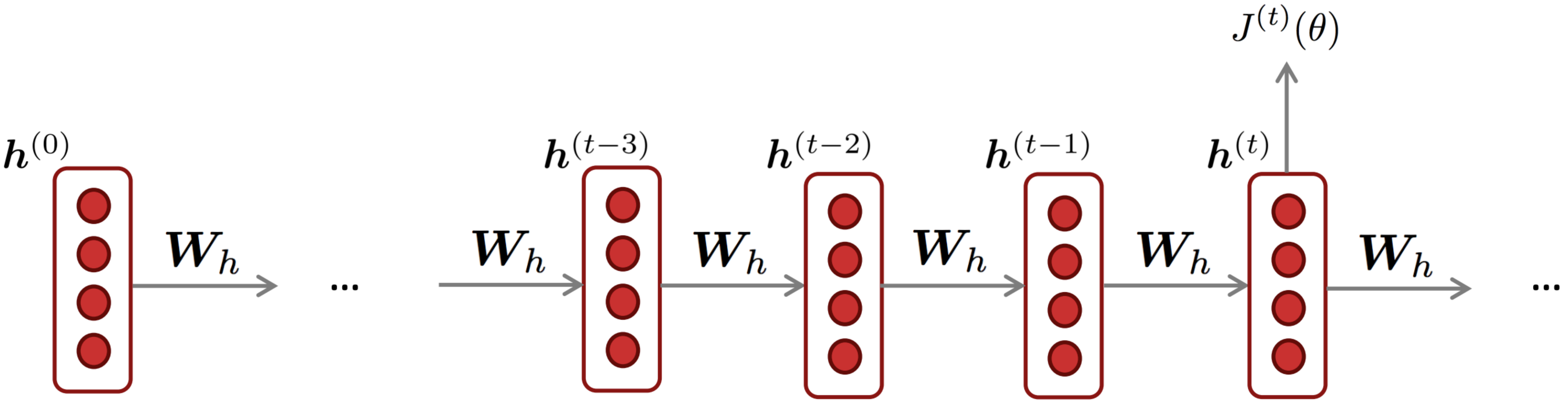
Truncated backpropagation through time

- Backpropagation is very expensive if you handle long sequences



- Run forward and backward through chunks of the sequence instead of whole sequence
- Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps

Let's consider the gradient wrt the weight matrix



$$\frac{\partial J}{\partial \mathbf{W}_h} = -\frac{1}{n}\sum_{t=1}^{n}\frac{\partial J^{(t)}}{\partial \mathbf{W}_h}$$

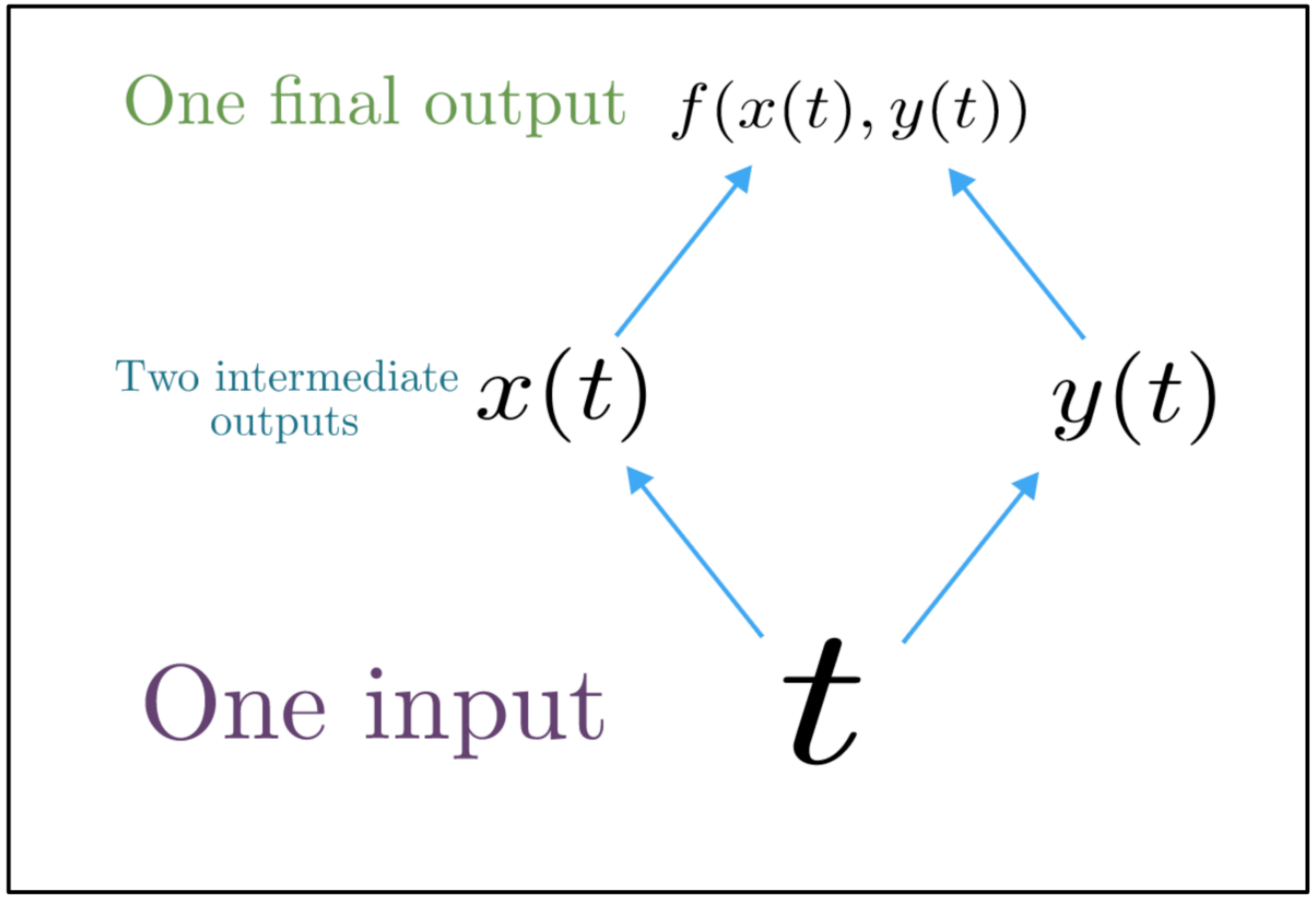
$$\frac{\partial J^{(t)}}{\partial \mathbf{W}_h} = \sum_{i=1}^{t}\left.\frac{\partial J^{(t)}}{\partial \mathbf{W}_h}\right|_{(i)}$$

Gradient wrt a repeated weight is the sum of the gradient wrt each time it appears

Recall: Gradient sum at branches

Multivariate Chain Rule

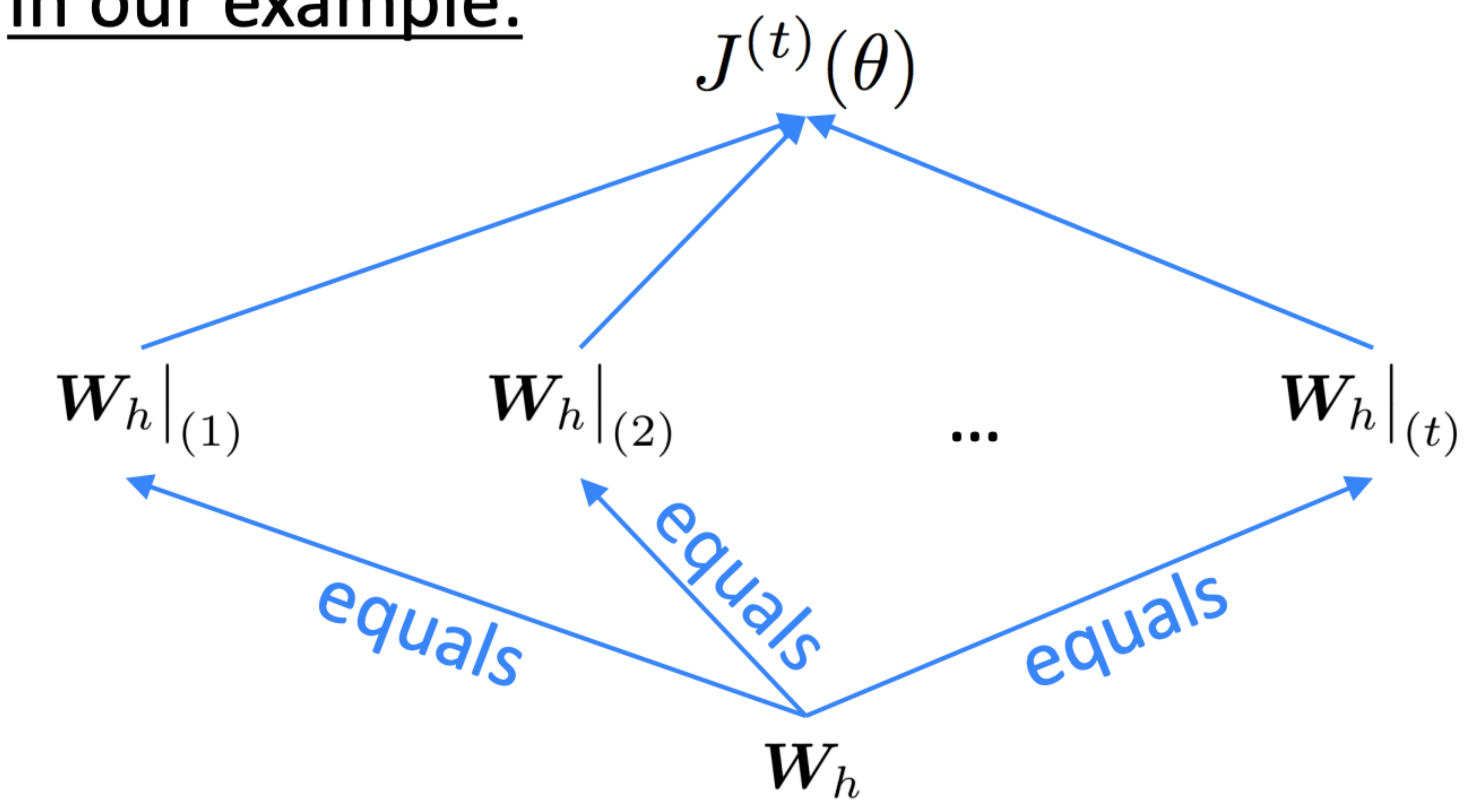
$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$



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

Recall: Gradient sum at branches

<u>In our example:</u>



<u>Apply the multivariable chain rule:</u>

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} = \sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} \right|_{(i)} \overbrace{\boxed{\frac{\partial \boldsymbol{W}_h \big|_{(i)}}{\partial \boldsymbol{W}_h}}}^{=1}$$

$$= \sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} \right|_{(i)}$$

$$\frac{\partial J^{(t)}}{\partial \mathbf{W}_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \mathbf{W}_h}\bigg|_{(i)}$$

BPTT: Example for t=3

$$L_t = J^{(t)}$$

$$\mathbf{h}_1 = g(\mathbf{W}_h \mathbf{h}_0 + \mathbf{W}_x \mathbf{x}_1 + \mathbf{b})$$
$$\mathbf{h}_2 = g(\mathbf{W}_h \mathbf{h}_1 + \mathbf{W}_x \mathbf{x}_2 + \mathbf{b})$$
$$\mathbf{h}_3 = g(\mathbf{W}_h \mathbf{h}_2 + \mathbf{W}_x \mathbf{x}_3 + \mathbf{b})$$
$$L_3 = -\log \hat{\mathbf{y}}_3(w_4)$$

If i and t are far away, the gradients are likely to grow/shrink exponentially (called the exploding or vanishing gradient problem)

You should know how to compute: $\frac{\partial L_3}{\partial \mathbf{h}_3}$

$$\frac{\partial L_3}{\partial \mathbf{W}_h} = \frac{\partial L_3}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h} + \frac{\partial L_3}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{W}_h} + \frac{\partial L_3}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial \mathbf{W}_h}$$

$$\boxed{\frac{\partial L}{\partial \mathbf{W}_h} = -\frac{1}{n}\sum_{t=1}^{n}\sum_{i=1}^{t}\frac{\partial L_t}{\partial \mathbf{h}_t}\left(\prod_{j=i+1}^{t}\frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right)\frac{\partial \mathbf{h}_i}{\partial \mathbf{W}_h}}$$

Exploding and vanishing gradients

(advanced)

Vanishing/exploding gradients

- Consider the gradient of L_t at step t, with respect to the hidden state $\mathbf{h}_k$ at some previous step k ($k < t$):

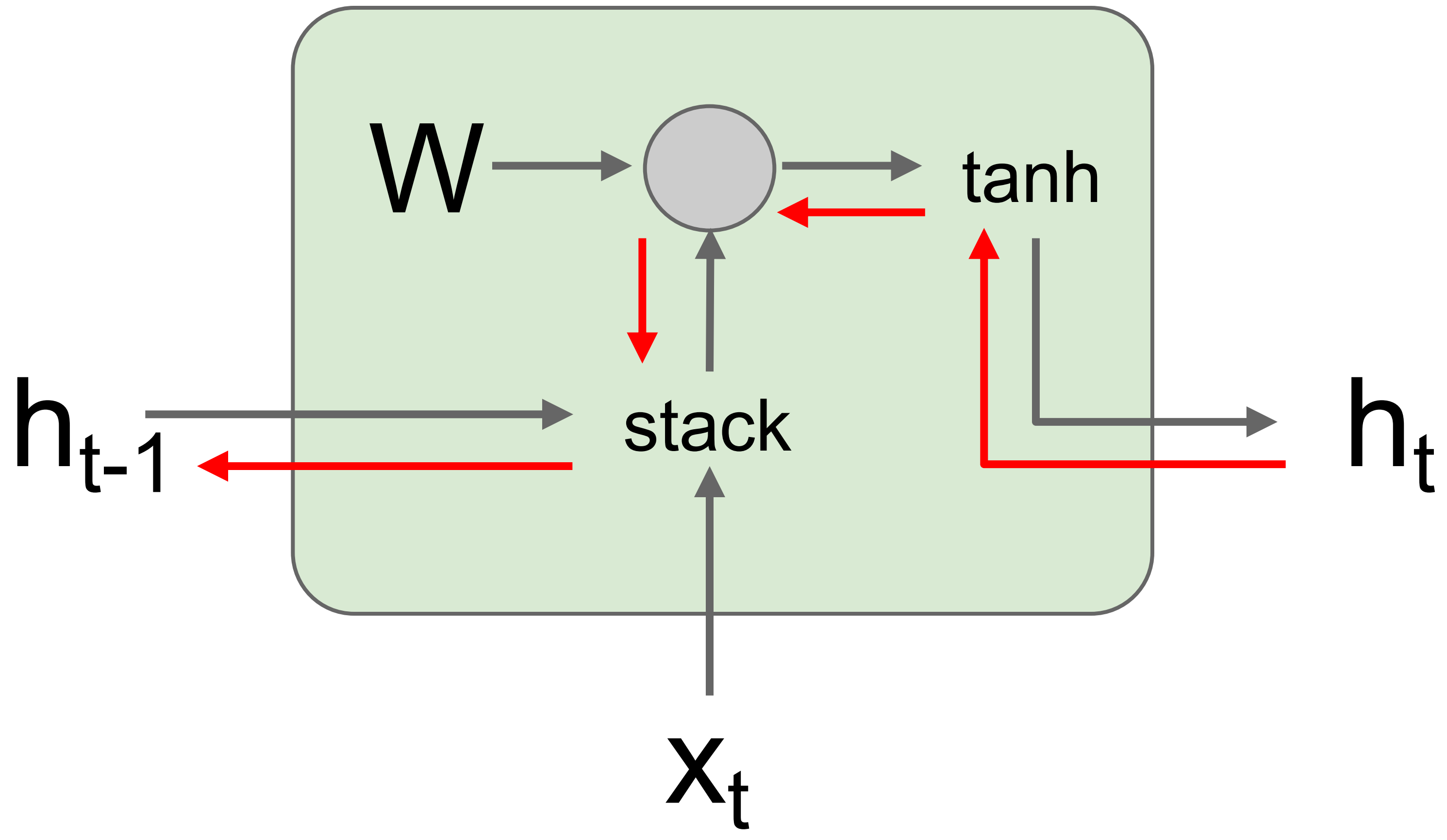
$$\frac{\partial L_t}{\partial \mathbf{h}_k} = \frac{\partial L_t}{\partial \mathbf{h}_t}\left(\prod_{t \geq j > k} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right)$$

$$= \frac{\partial L_t}{\partial \mathbf{h}_t} \times \prod_{t \geq j > k}\left(\text{diag}\left(g'(\mathbf{W}\mathbf{h}_{j-1} + \mathbf{U}\mathbf{x}_j + \mathbf{b})\right)\mathbf{W}\right)$$

- (Pascanu et al, 2013) showed that if the largest eigenvalue of $\mathbf{W}$ is less than 1 for $g = \tanh$, then the gradient will shrink exponentially. This problem is called **vanishing gradients**.

- In contrast, if the gradients are getting too large, it is called **exploding gradients**.

Gradient flow through Vanilla RNN cell



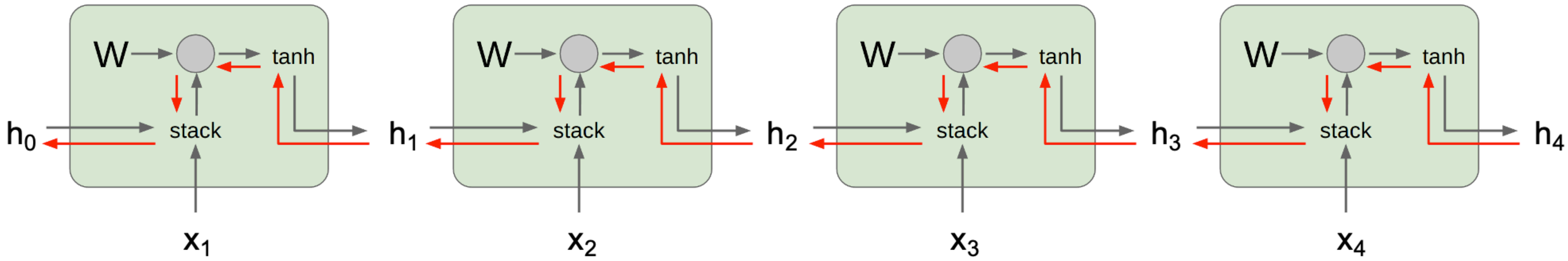
First, using matrix notation

$$
\begin{aligned}
\mathbf{h}_t &= \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{hx}\mathbf{x}_t) \\
&= \tanh\left(\begin{pmatrix}\mathbf{W}_{hh} & \mathbf{W}_{hx}\end{pmatrix}\begin{pmatrix}\mathbf{h}_{t-1} \\ \mathbf{x}_t\end{pmatrix}\right) \\
&= \tanh\left(\mathbf{W}\begin{pmatrix}\mathbf{h}_{t-1} \\ \mathbf{x}_t\end{pmatrix}\right)
\end{aligned}
$$

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994

Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

Exploding and Vanishing Gradients



Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1:
Exploding gradients

→ Difficult for model to converge!

Largest singular value < 1:
Vanishing gradients

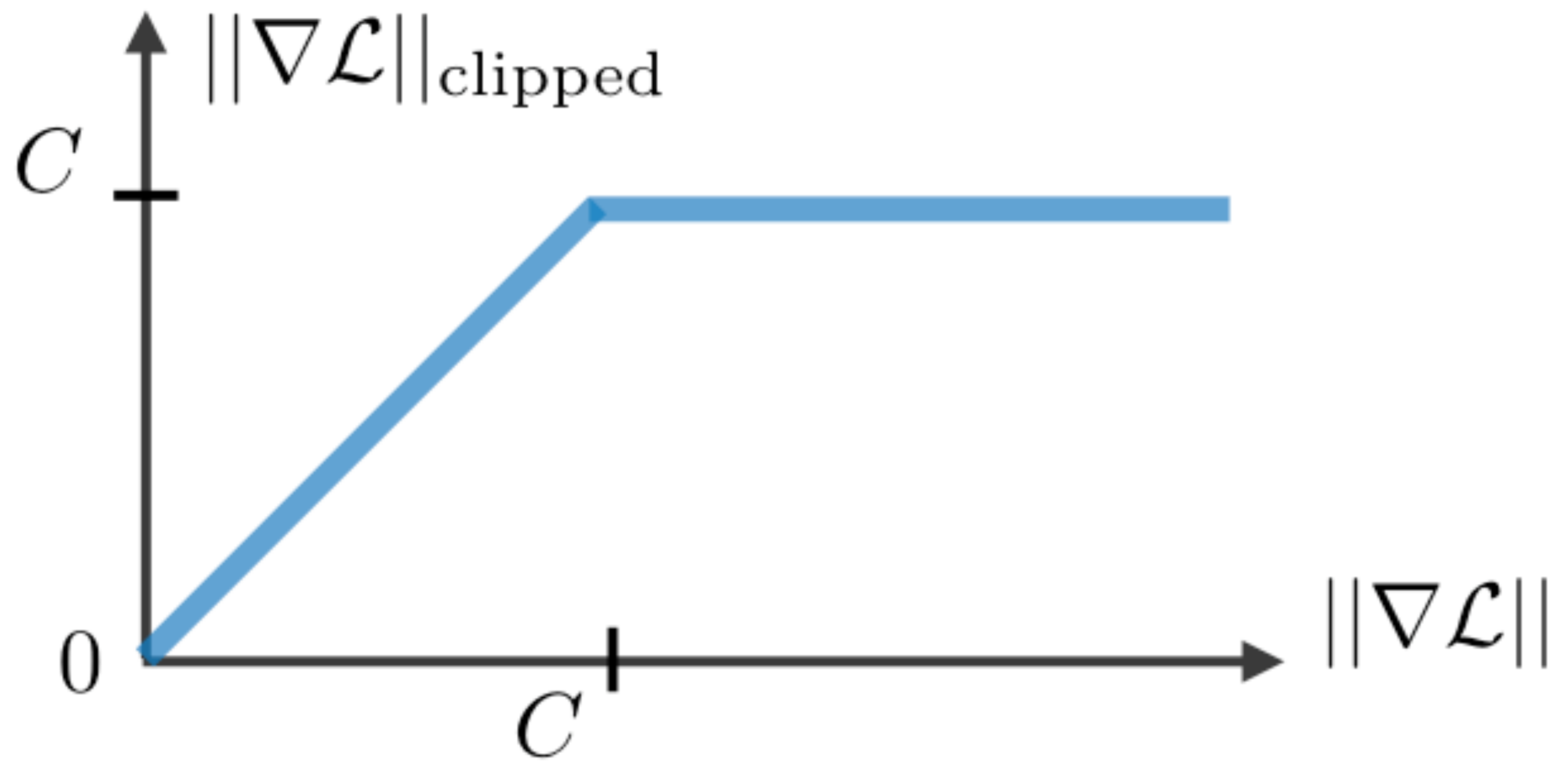
Why is exploding gradient a problem?

- Gradients become too big and we take a very large step in SGD.
- **Solution**: Gradient clipping — if the norm of the gradient is greater than some threshold, scale it down before applying SGD update.

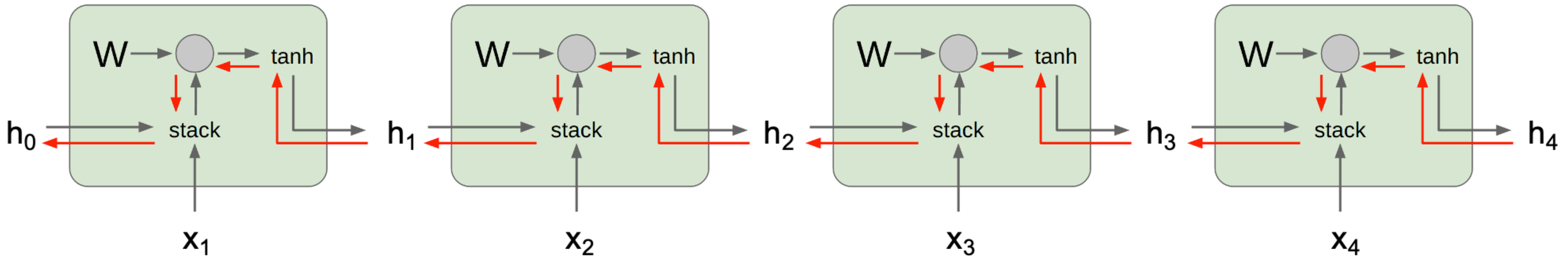
Difficult for model to converge!

Algorithm 1 Pseudo-code for norm clipping

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



Exploding and Vanishing Gradients

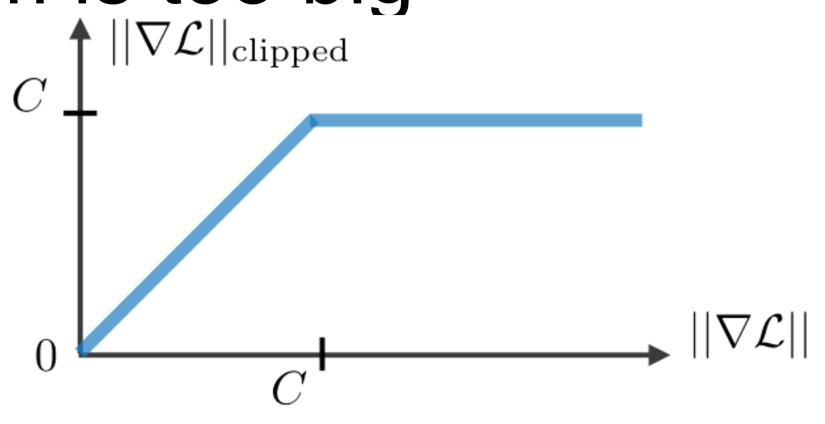


Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1:
Exploding gradients

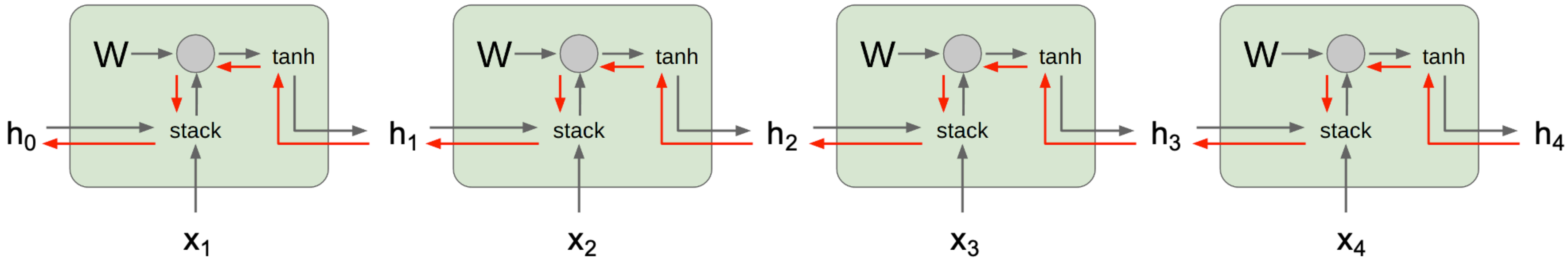
Gradient clipping:
Scale gradient if its norm is too big

```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
  grad *= (threshold / grad_norm)
```



Largest singular value < 1:
Vanishing gradients

Exploding and Vanishing Gradients



Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1:
Exploding gradients

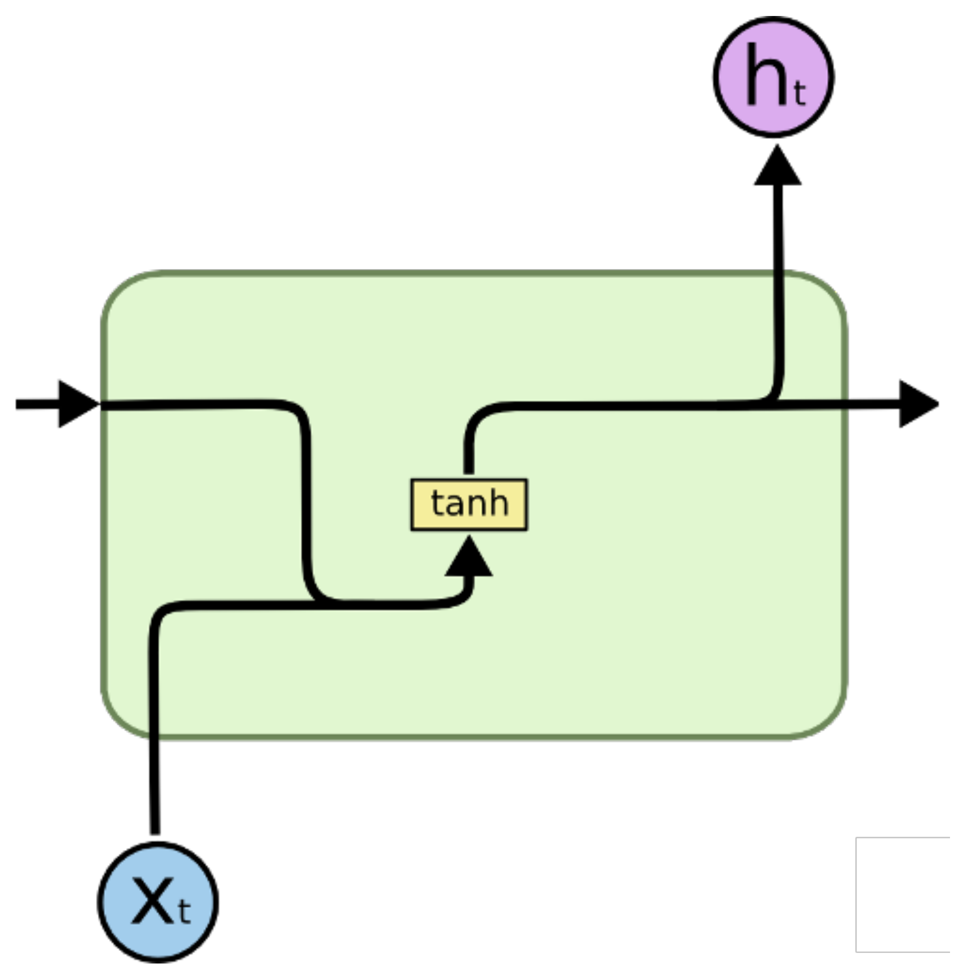
Largest singular value < 1:
Vanishing gradients

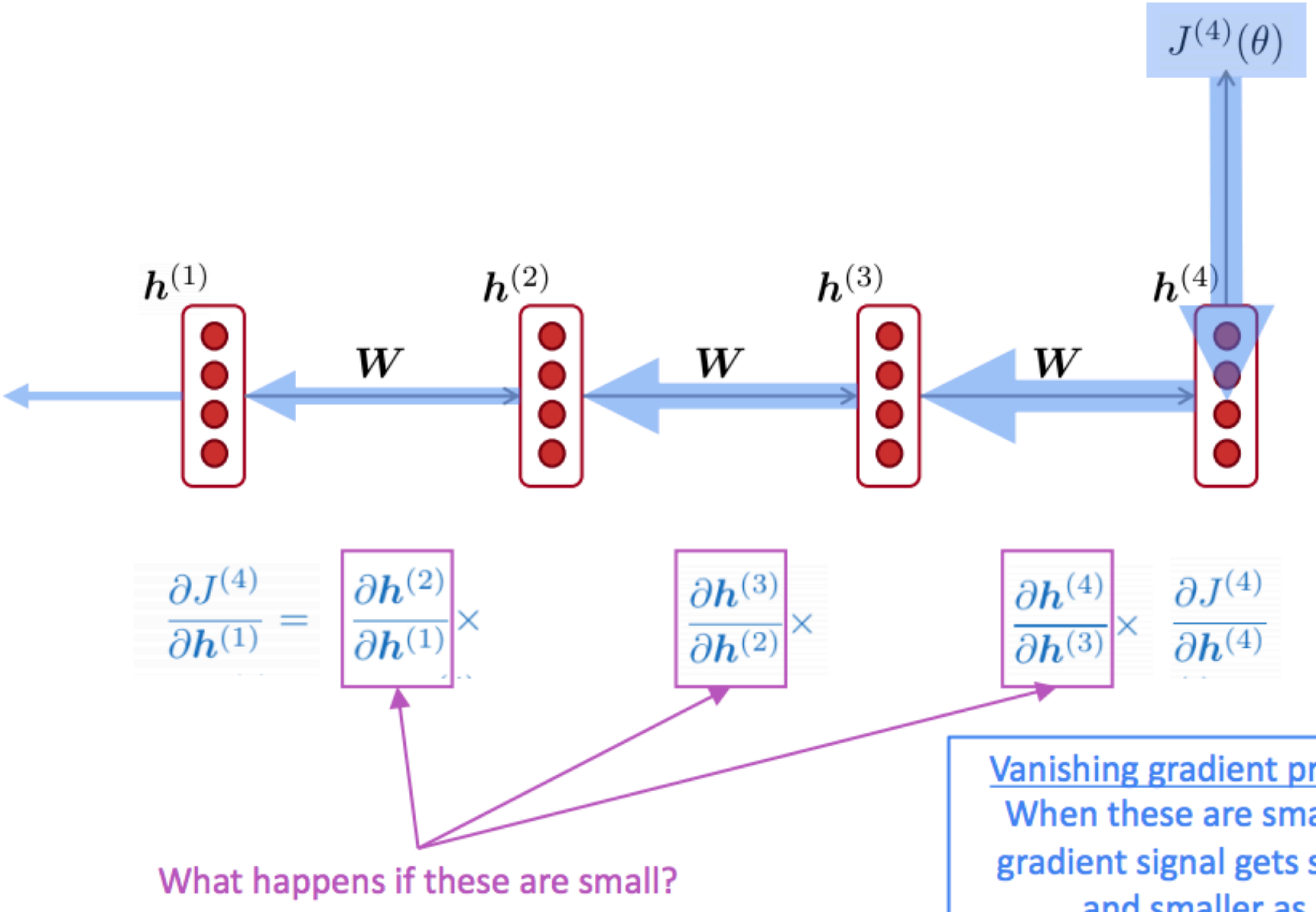
→ Can't capture long distance dependencies.

Vanishing gradients

Simple RNN

$$h_t = \tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$





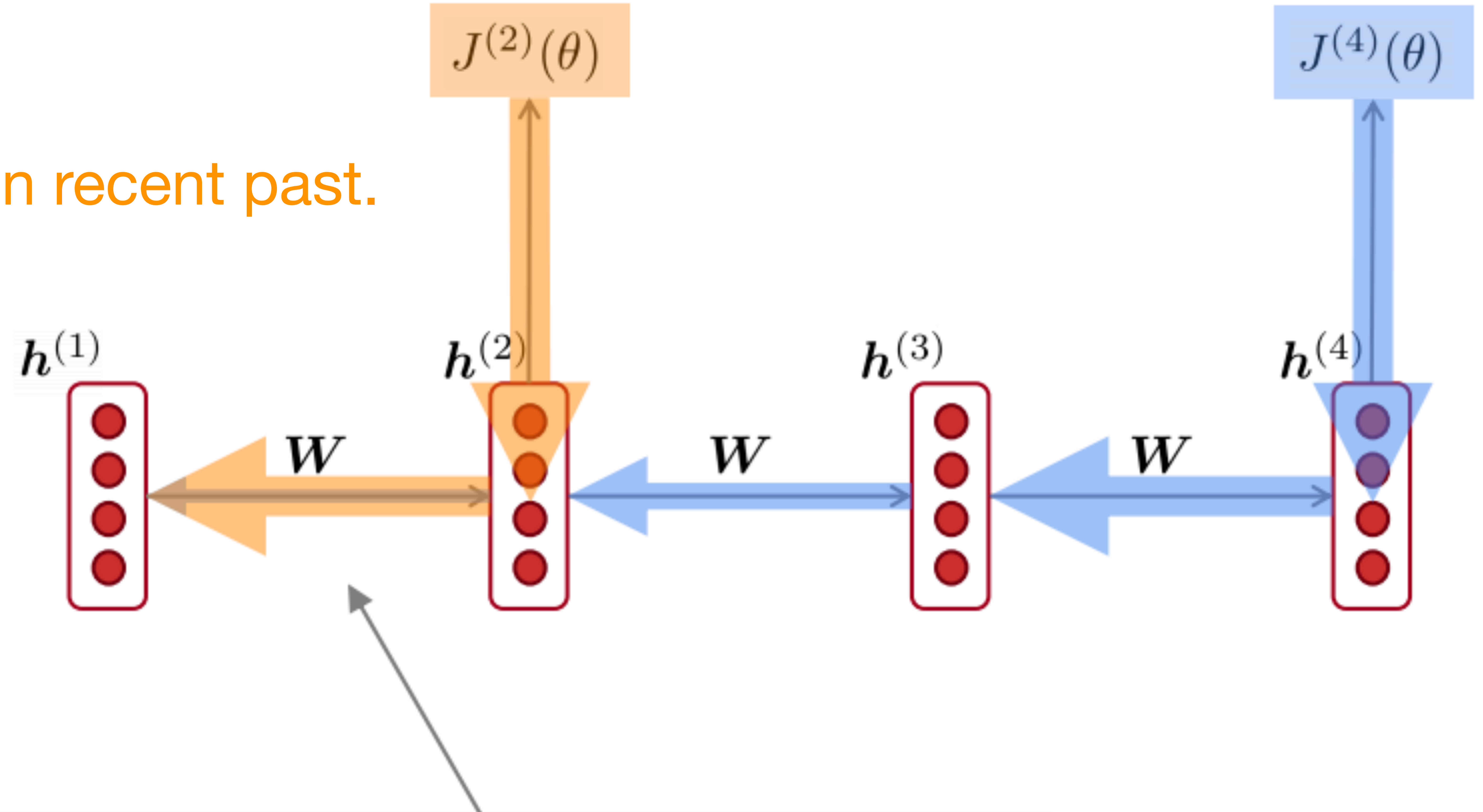
<u>Vanishing gradient problem:</u>
When these are small, the gradient signal gets smaller and smaller as it backpropagates further

(Figure credit: Abigail See)

Vanishing Gradients

Can’t capture long distance dependencies.

Focus more on recent past.

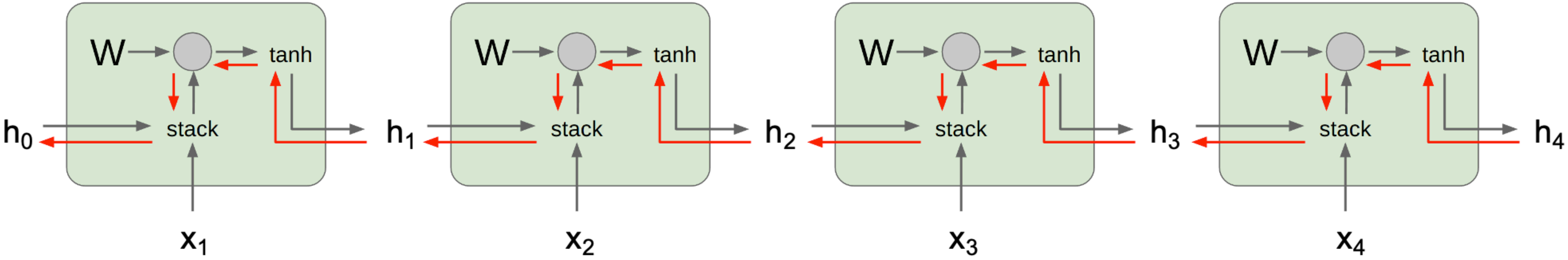


Gradient signal from faraway is lost because it’s much smaller than gradient signal from close-by.

So model weights are updated only with respect to near effects, not long-term effects.

(Figure credit: Abigail See)

Exploding and Vanishing Gradients

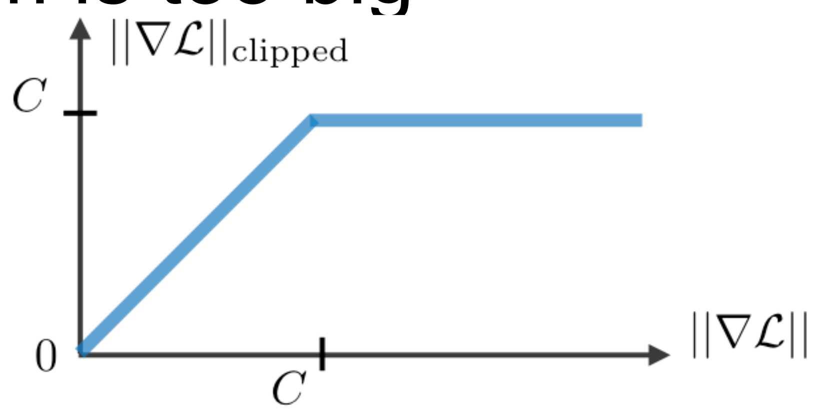


Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1:
Exploding gradients

Gradient clipping:
Scale gradient if its norm is too big

```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
  grad *= (threshold / grad_norm)
```



Largest singular value < 1:
Vanishing gradients

Change RNN architecture

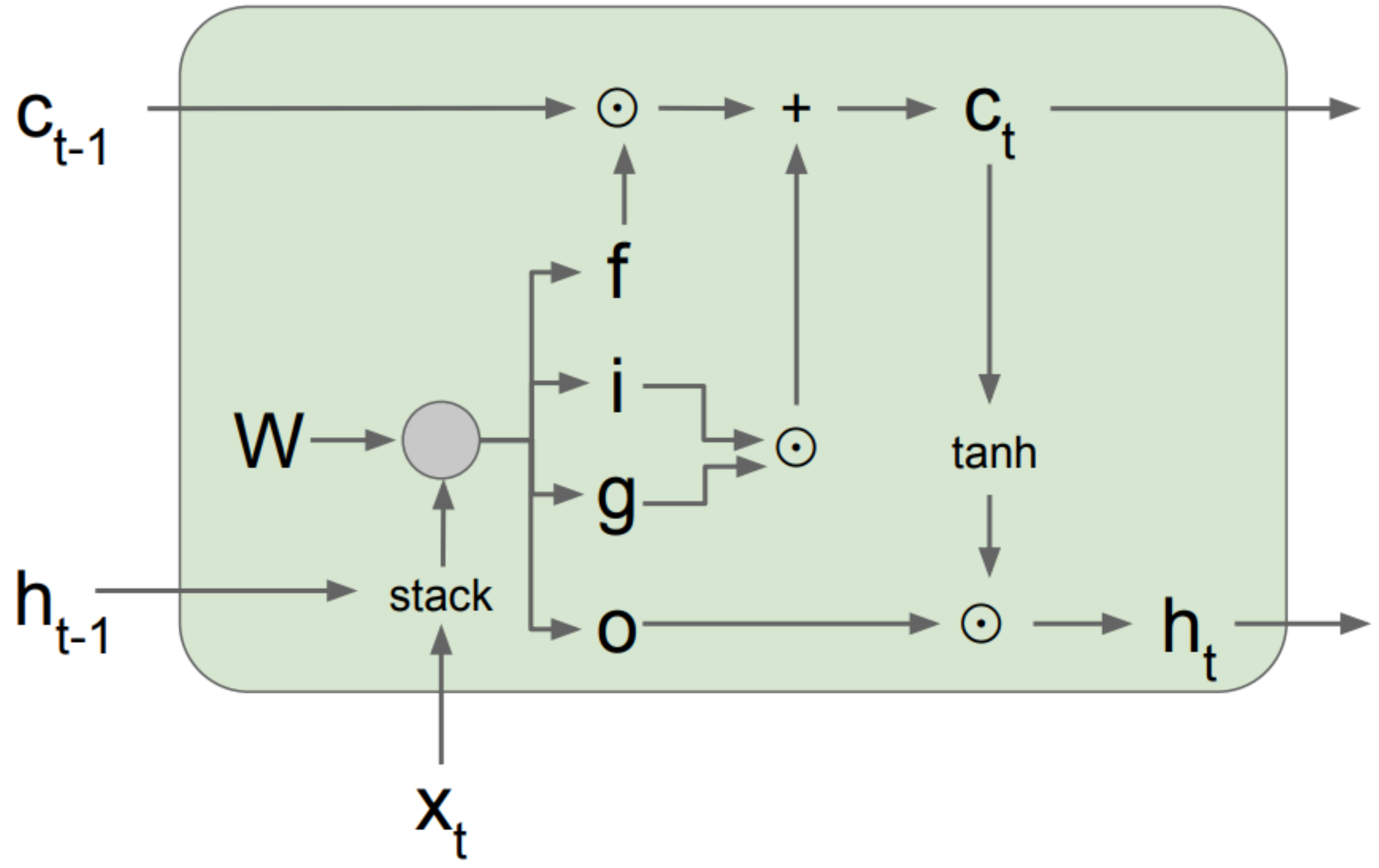
Different RNN cells

Long Short-term Memory (LSTM)

- A type of RNN proposed by Hochreiter and Schmidhuber in 1997 as a solution to the vanishing gradients problem
- Work extremely well in practice
- **Basic idea**: turning multiplication into addition
- Use “gates” to control how much information to add/erase

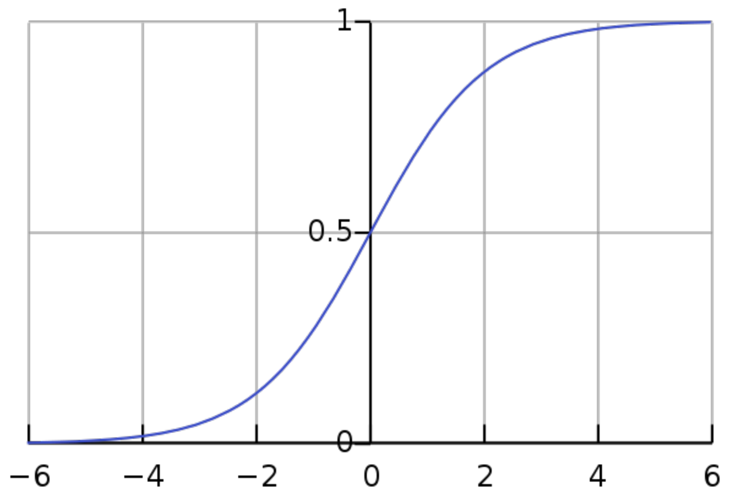
$$\boxed{\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t) \in \mathbb{R}^d}$$

- At each timestep, there is a hidden state $\mathbf{h}_t \in \mathbb{R}^d$ and also a cell state $\mathbf{c}_t \in \mathbb{R}^d$
  - $\mathbf{c}_t$ stores **long-term information**
  - We write/erase $\mathbf{c}_t$ after each step
  - We read $\mathbf{h}_t$ from $\mathbf{c}_t$

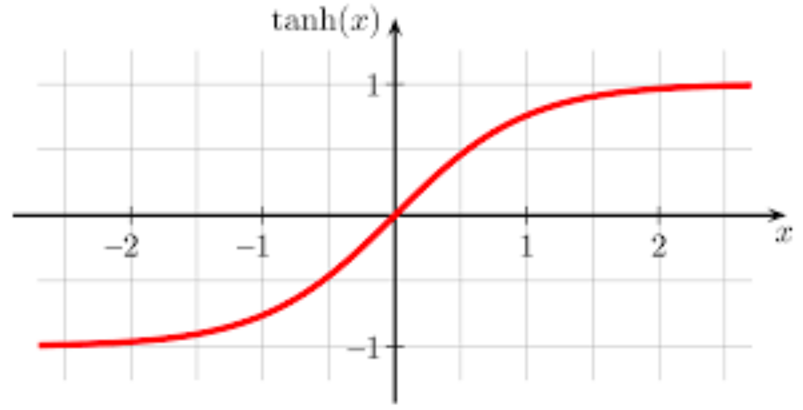


Long Short-term Memory (LSTM)

Use logistic for gating
0 = filter out,
1 = pass through



Use tanh for output
(zero-centered for
feeding into next layer)



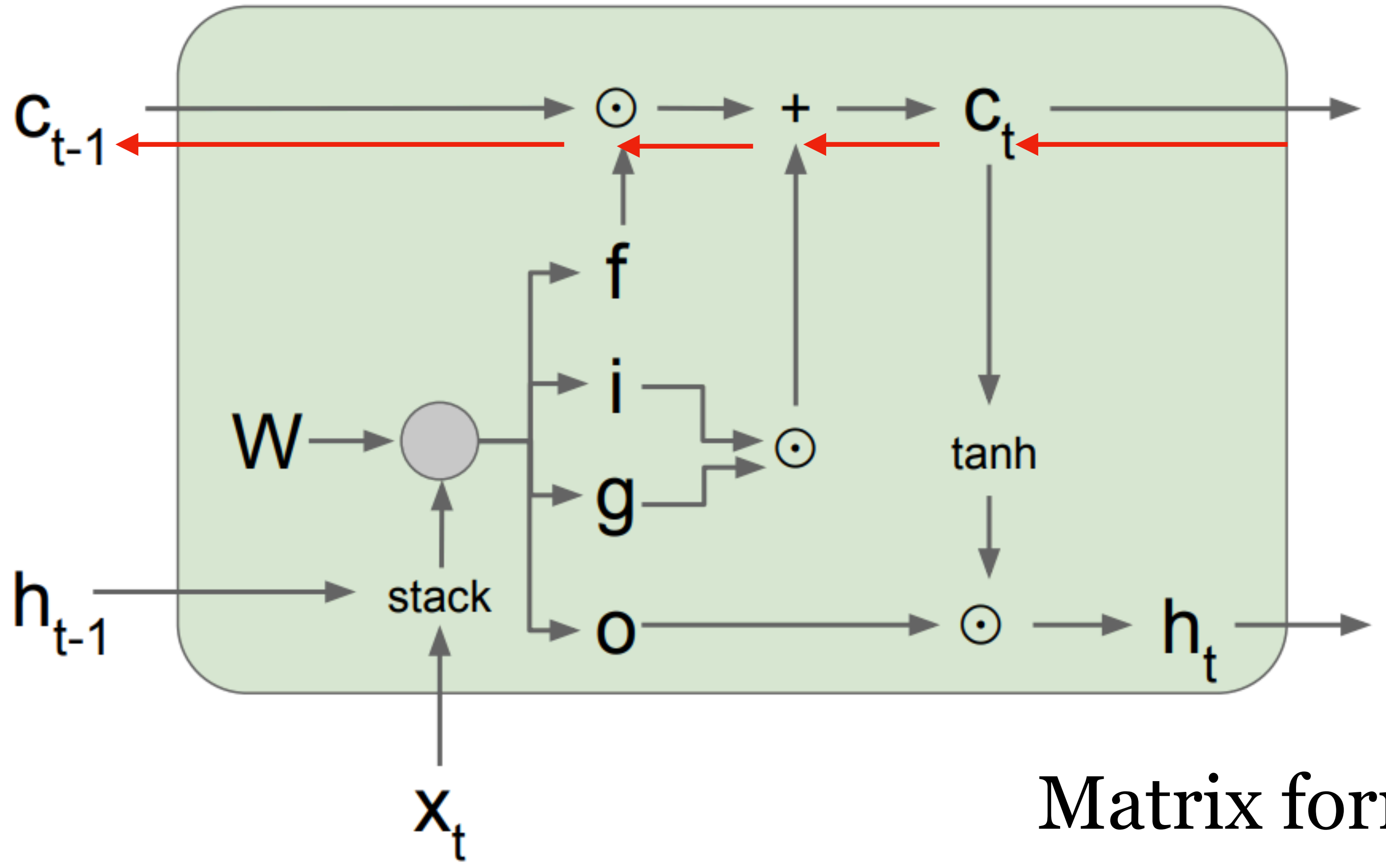
There are 3 gates and a memory cell:

- Input gate (how much to write):
  $\mathbf{i}_t = \sigma(\mathbf{W}^{(i)}\mathbf{h}_{t-1} + \mathbf{U}^{(i)}\mathbf{x}_t + \mathbf{b}^{(i)}) \in \mathbb{R}^d$
- Forget gate (how much to erase):
  $\mathbf{f}_t = \sigma(\mathbf{W}^{(f)}\mathbf{h}_{t-1} + \mathbf{U}^{(f)}\mathbf{x}_t + \mathbf{b}^{(f)}) \in \mathbb{R}^d$
- Output gate (how much to reveal):
  $\mathbf{o}_t = \sigma(\mathbf{W}^{(o)}\mathbf{h}_{t-1} + \mathbf{U}^{(o)}\mathbf{x}_t + \mathbf{b}^{(o)}) \in \mathbb{R}^d$
- New memory cell (what to write):
  $\mathbf{g}_t = \tanh(\mathbf{W}^{(c)}\mathbf{h}_{t-1} + \mathbf{U}^{(c)}\mathbf{x}_t + \mathbf{b}^{(c)}) \in \mathbb{R}^d$

element-wise product

- Final memory cell: $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$
- Final hidden cell: $\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$

Backpropagation from $\mathbf{c}_t$ to $\mathbf{c}_{t-1}$ only element wise multiplication by $\mathbf{f}$, no matrix multiply by $\mathbf{W}$



Matrix form

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

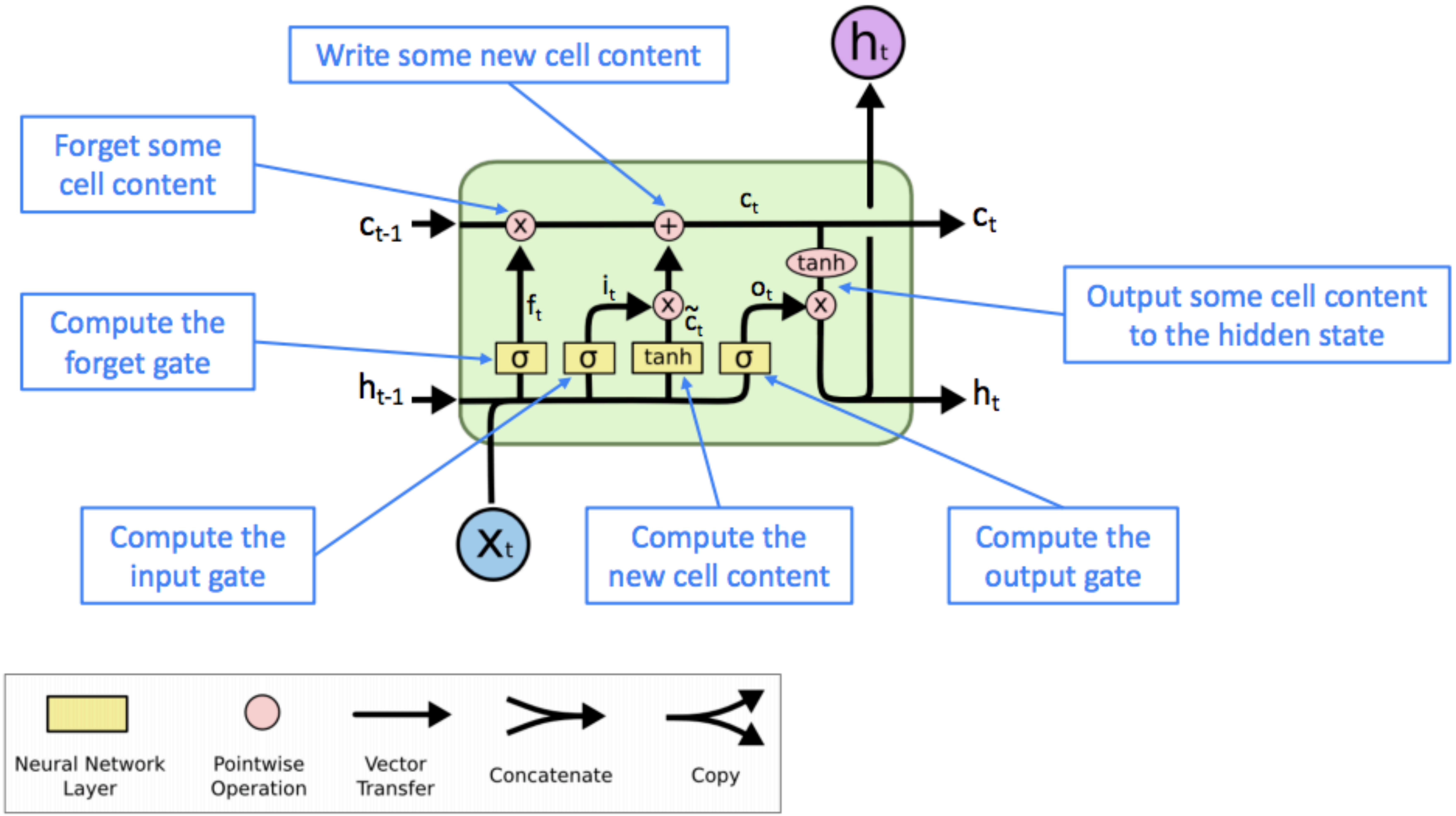
$$h_t = o \odot \tanh(c_t)$$

How many parameters in total? $4 \times (d^2 + dm + d)$

$\mathbf{x}_t \in \mathbb{R}^m$

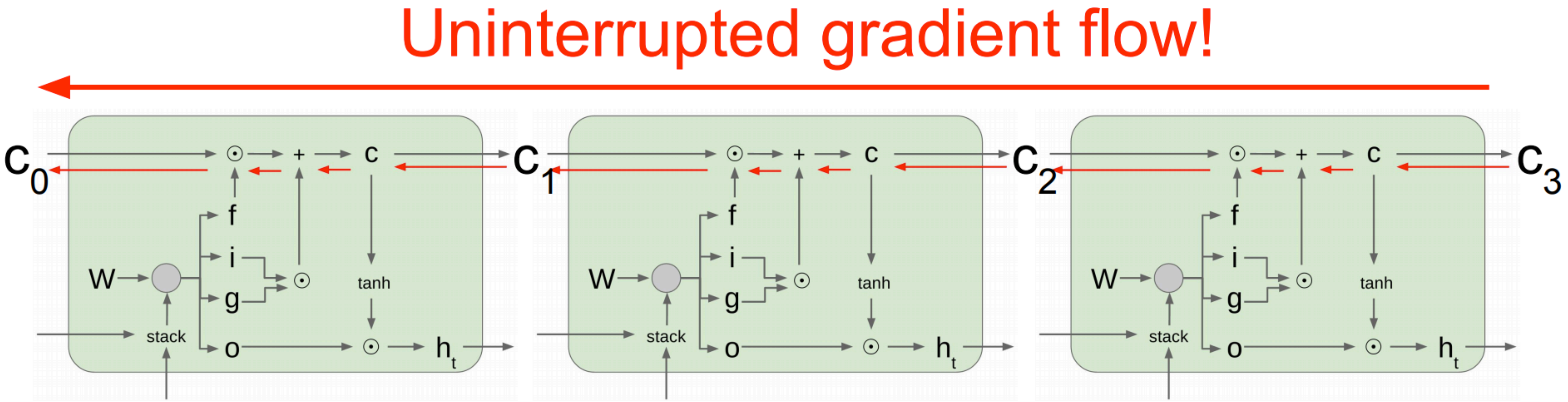
LSTM cell intuitively

You can think of the LSTM equations visually like this:



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

Long Short-term Memory (LSTM)



- LSTM doesn't guarantee that there is no vanishing/exploding gradient, but it does provide an easier way for the model to learn long-distance dependencies

- LSTMs were invented in 1997 but finally got working from 2013-2015.

Is the LSTM architecture optimal?

MUT1:

$$\begin{aligned} z &= \mathrm{sigm}(W_{\mathrm{xz}} x_t + b_{\mathrm{z}}) \\ r &= \mathrm{sigm}(W_{\mathrm{xr}} x_t + W_{\mathrm{hr}} h_t + b_{\mathrm{r}}) \\ h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + \tanh(x_t) + b_{\mathrm{h}}) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

MUT2:

$$\begin{aligned} z &= \mathrm{sigm}(W_{\mathrm{xz}} x_t + W_{\mathrm{hz}} h_t + b_{\mathrm{z}}) \\ r &= \mathrm{sigm}(x_t + W_{\mathrm{hr}} h_t + b_{\mathrm{r}}) \\ h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + W_{xh} x_t + b_{\mathrm{h}}) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

MUT3:

$$\begin{aligned} z &= \mathrm{sigm}(W_{\mathrm{xz}} x_t + W_{\mathrm{hz}} \tanh(h_t) + b_{\mathrm{z}}) \\ r &= \mathrm{sigm}(W_{\mathrm{xr}} x_t + W_{\mathrm{hr}} h_t + b_{\mathrm{r}}) \\ h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + W_{xh} x_t + b_{\mathrm{h}}) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

Arch.	Arith.	XML	PTB
Tanh	0.29493	0.32050	0.08782
LSTM	0.89228	0.42470	0.08912
LSTM-f	0.29292	0.23356	0.08808
LSTM-i	0.75109	0.41371	0.08662
LSTM-o	0.86747	0.42117	0.08933
LSTM-b	0.90163	0.44434	0.08952
GRU	0.89565	0.45963	0.09069
MUT1	**0.92135**	**0.47483**	0.08968
MUT2	0.89735	**0.47324**	0.09036
MUT3	0.90728	0.46478	**0.09161**

Arch.	5M-tst	10M-v	20M-v	20M-tst
Tanh	4.811	4.729	4.635	4.582 (97.7)
LSTM	4.699	4.511	4.437	4.399 (81.4)
LSTM-f	4.785	4.752	4.658	4.606 (100.8)
LSTM-i	4.755	4.558	4.480	4.444 (85.1)
LSTM-o	4.708	4.496	4.447	4.411 (82.3)
LSTM-b	4.698	4.437	4.423	**4.380 (79.83)**
GRU	4.684	4.554	4.559	4.519 (91.7)
MUT1	4.699	4.605	4.594	4.550 (94.6)
MUT2	4.707	4.539	4.538	4.503 (90.2)
MUT3	4.692	4.523	4.530	4.494 (89.47)

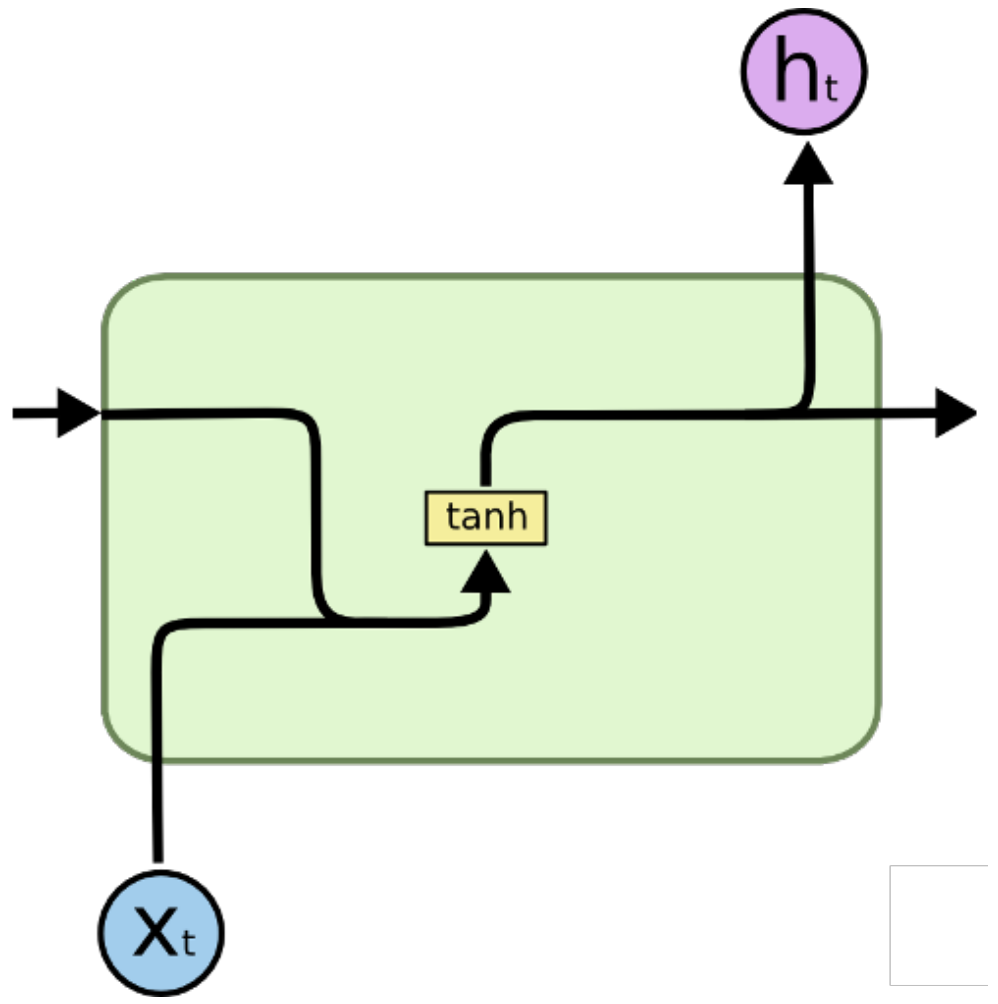
(Jozefowicz et al, 2015): An Empirical Exploration of Recurrent Network Architectures

Simple RNN vs GRU vs LSTM

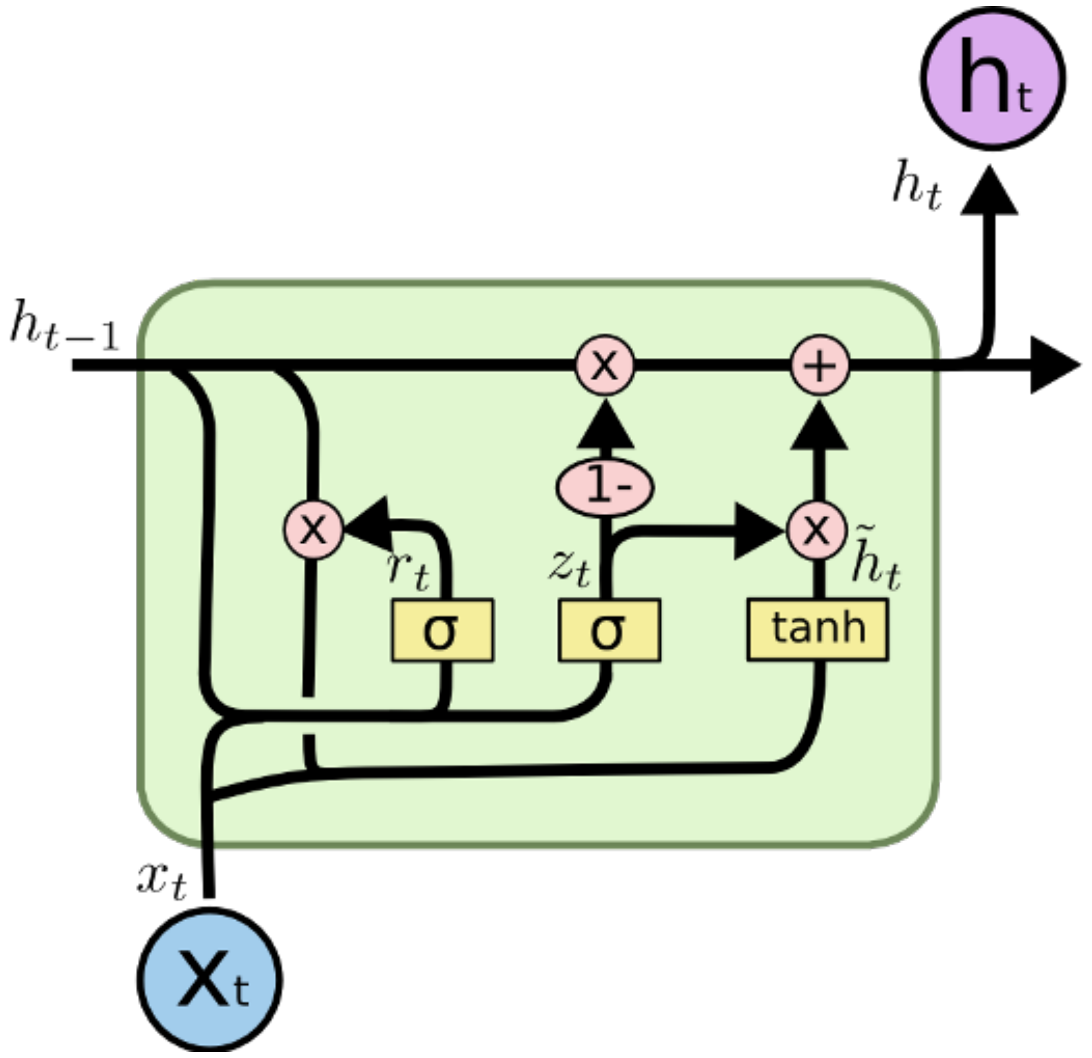
$$\mathbf{h}_t = g(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$$

$$\begin{aligned}
\mathbf{r}_t &= \sigma(\mathbf{W}^r\mathbf{h}_{t-1} + \mathbf{U}^r\mathbf{x}_t + \mathbf{b}^r) \\
\mathbf{z}_t &= \sigma(\mathbf{W}^z\mathbf{h}_{t-1} + \mathbf{U}^z\mathbf{x}_t + \mathbf{b}^z) \\
\tilde{\mathbf{h}}_t &= \tanh(\mathbf{W}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{U}\mathbf{x}_t + \mathbf{b}) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t
\end{aligned}$$

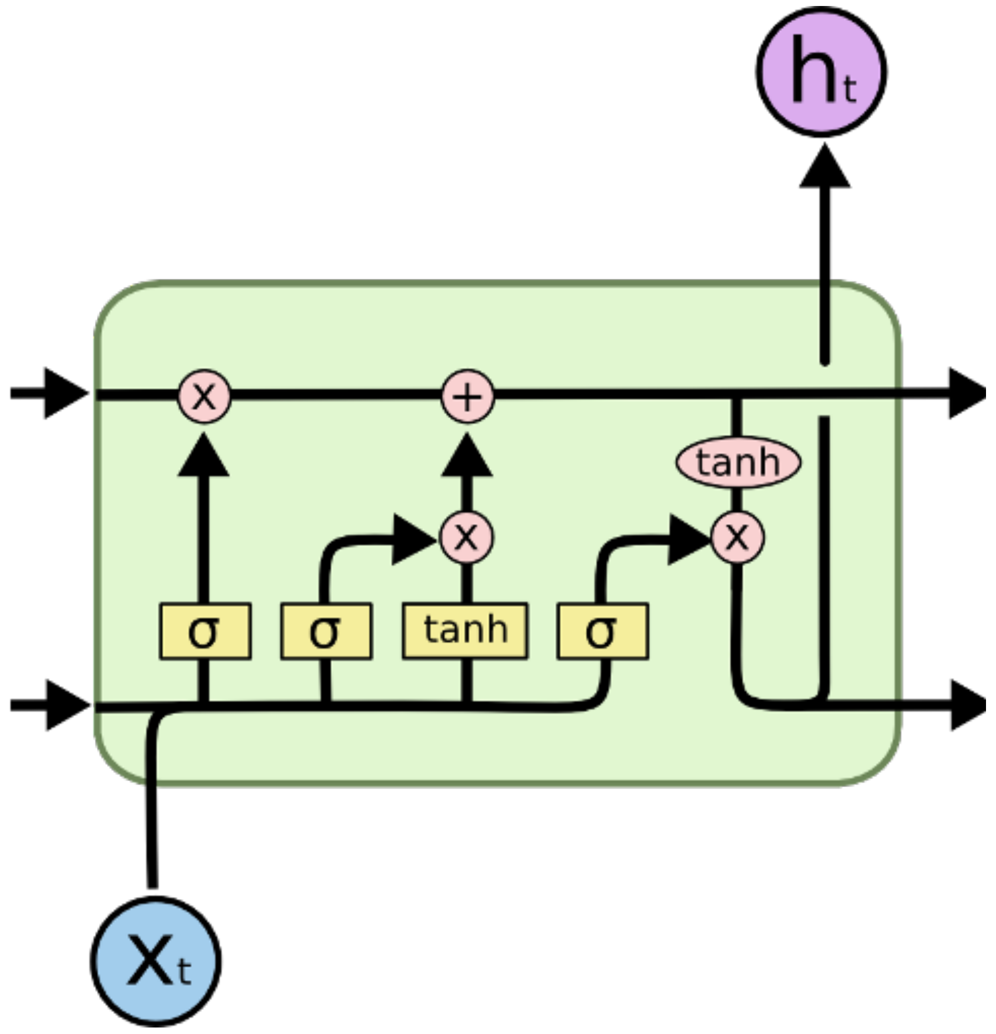
$$\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}^i\mathbf{h}_{t-1} + \mathbf{U}^i\mathbf{x}_t + \mathbf{b}^i) \\
\mathbf{f}_t &= \sigma(\mathbf{W}^f\mathbf{h}_{t-1} + \mathbf{U}^f\mathbf{x}_t + \mathbf{b}^f) \\
\mathbf{o}_t &= \sigma(\mathbf{W}^o\mathbf{h}_{t-1} + \mathbf{U}^o\mathbf{x}_t + \mathbf{b}^o) \\
\mathbf{g}_t &= \tanh(\mathbf{W}^g\mathbf{h}_{t-1} + \mathbf{U}^g\mathbf{x}_t + \mathbf{b}^g) \\
\mathbf{c}_t &= \mathbf{c}_{t-1} \odot \mathbf{f}_t + \mathbf{g}_t \odot \mathbf{i}_t \\
\mathbf{h}_t &= \tanh(\mathbf{c}_t) \odot \mathbf{o}_t
\end{aligned}$$



Simple RNN

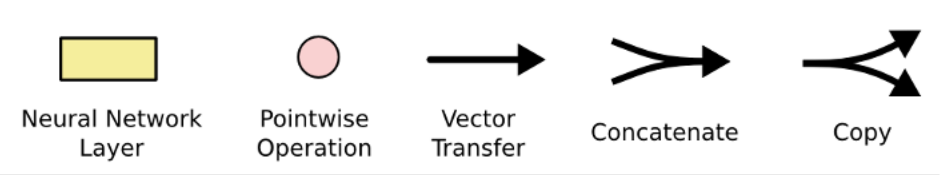


GRU



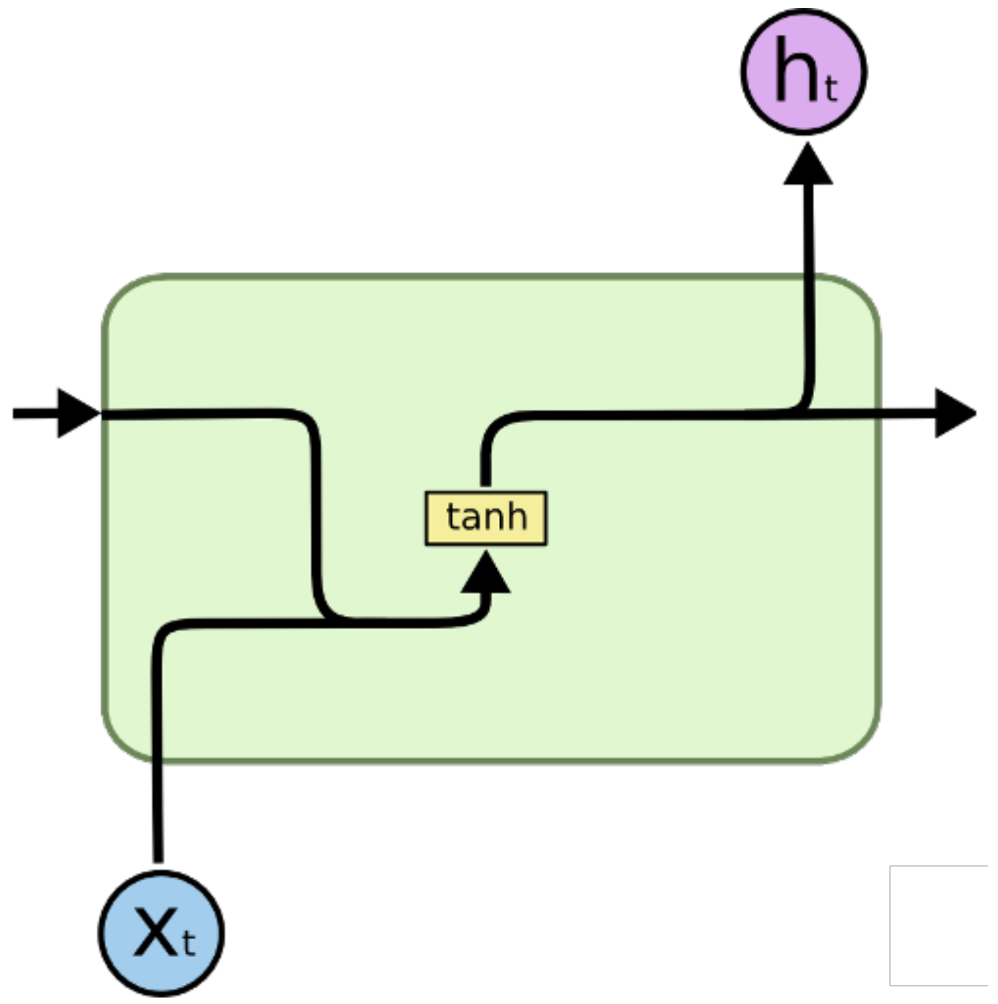
LSTM

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



Simple RNN vs GRU vs LSTM

$$h_t = \tanh\left(W\begin{pmatrix}h_{t-1}\\ x_t\end{pmatrix}\right)$$

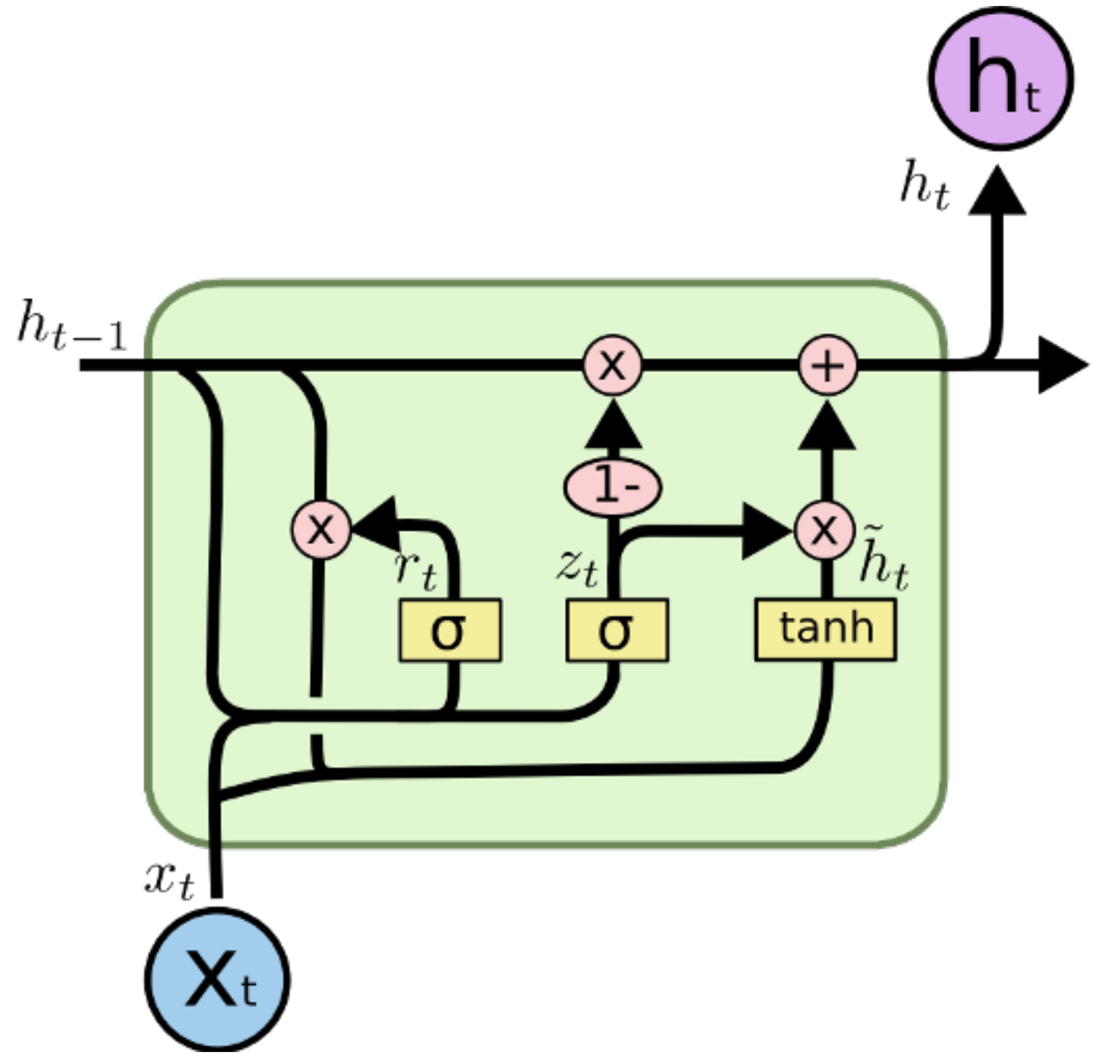


Simple RNN

$$\begin{pmatrix}r\\ z\end{pmatrix} = \begin{pmatrix}\sigma\\ \sigma\end{pmatrix} W_1 \begin{pmatrix}h_{t-1}\\ x_t\end{pmatrix}$$

$$\tilde{h}_t = \tanh\left(W_2\begin{pmatrix}r \odot h_{t-1}\\ x_t\end{pmatrix}\right)$$

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

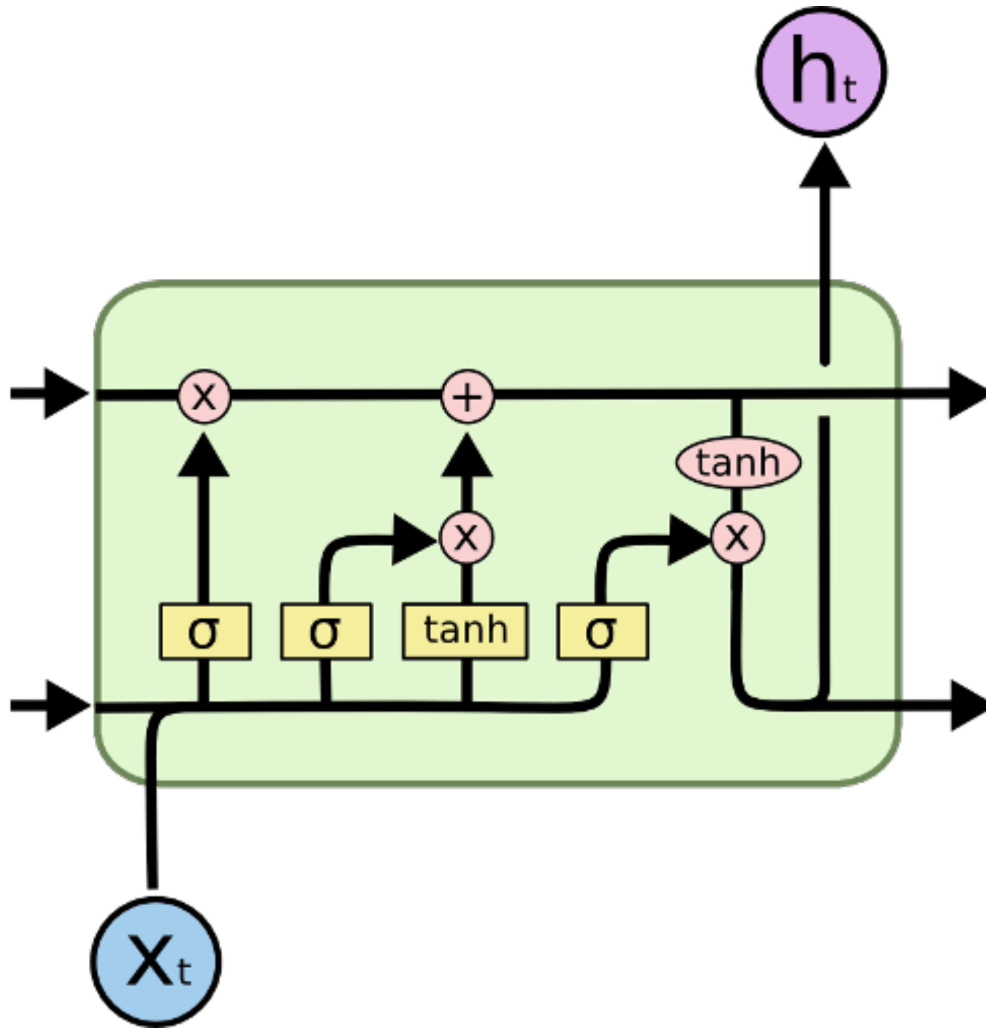


GRU

$$\begin{pmatrix}i\\ f\\ o\\ g\end{pmatrix} = \begin{pmatrix}\sigma\\ \sigma\\ \sigma\\ \tanh\end{pmatrix} W \begin{pmatrix}h_{t-1}\\ x_t\end{pmatrix}$$

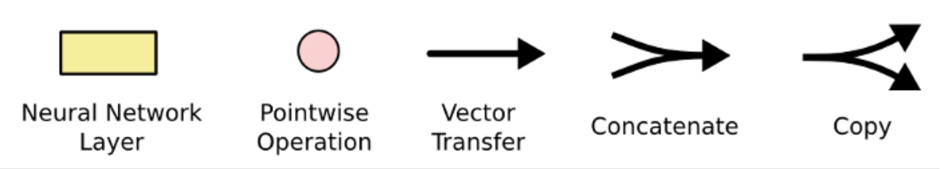
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

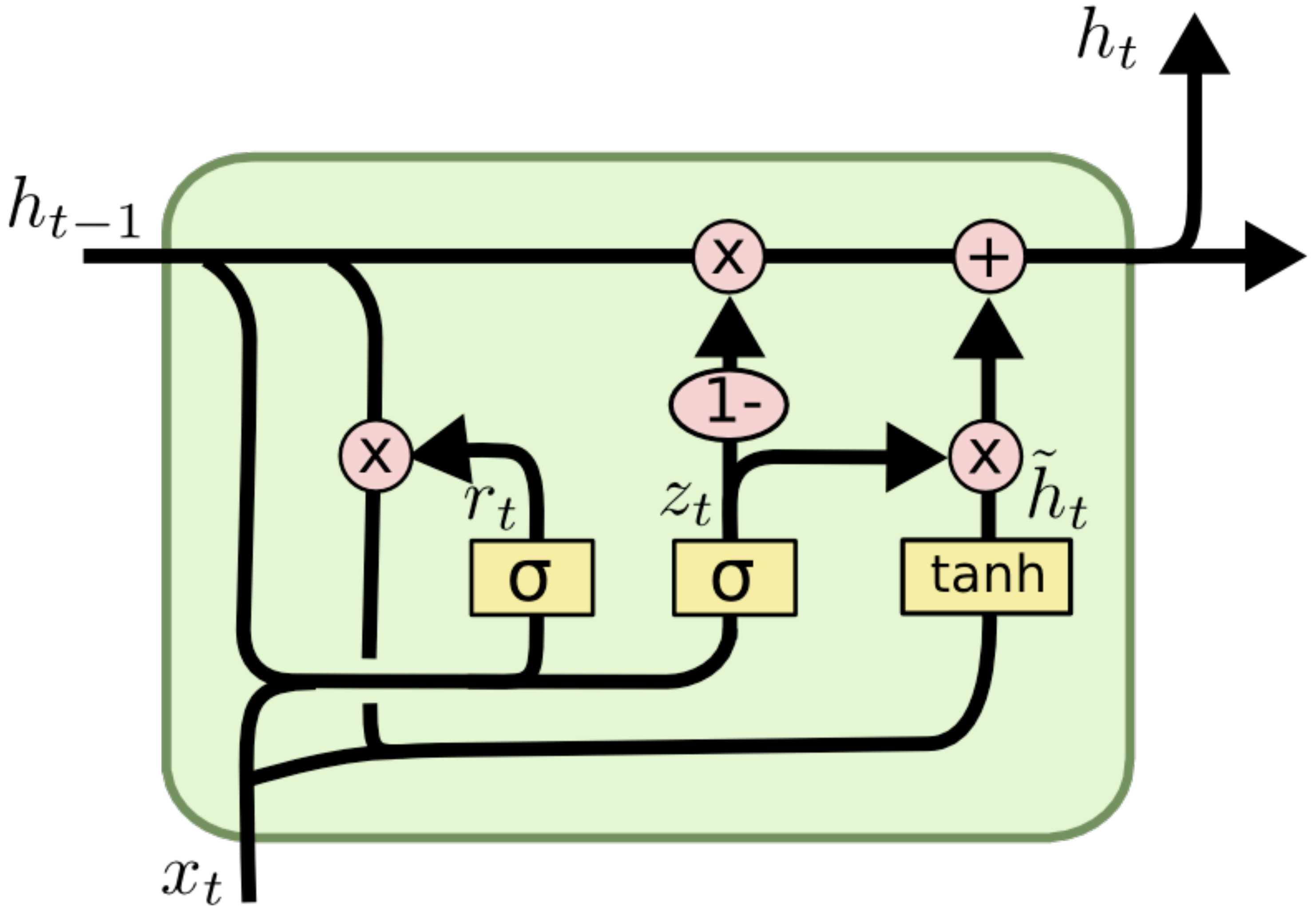


LSTM

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



GRU



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

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

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

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

- If reset is close to 0, ignore previous hidden state
  - Allows model to drop information that is irrelevant in the future
- Update gate z controls how much of past state should matter now.
  - 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

Progress on language models

On the Penn Treebank (PTB) dataset

Metric: perplexity

$$\text{ppl}(S) = 2^x \quad \text{where}$$
$$x = -\frac{1}{W}\sum_{i=1}^{n} \log_2 P(S^i)$$

KN5: Kneser-Ney 5-gram

Model	Individual	+KN5	+KN5+cache
KN5	141.2	-	-
KN5 + cache	125.7	-	-
Feedforward NNLM	140.2	116.7	106.6
Log-bilinear NNLM	144.5	115.2	105.8
Syntactical NNLM	131.3	110.0	101.5
Recurrent NNLM	124.7	105.7	97.5
RNN-LDA LM	113.7	98.3	92.0

(Mikolov and Zweig, 2012): Context dependent recurrent neural network language model

Progress on language models

On the Penn Treebank (PTB) dataset

Metric: perplexity

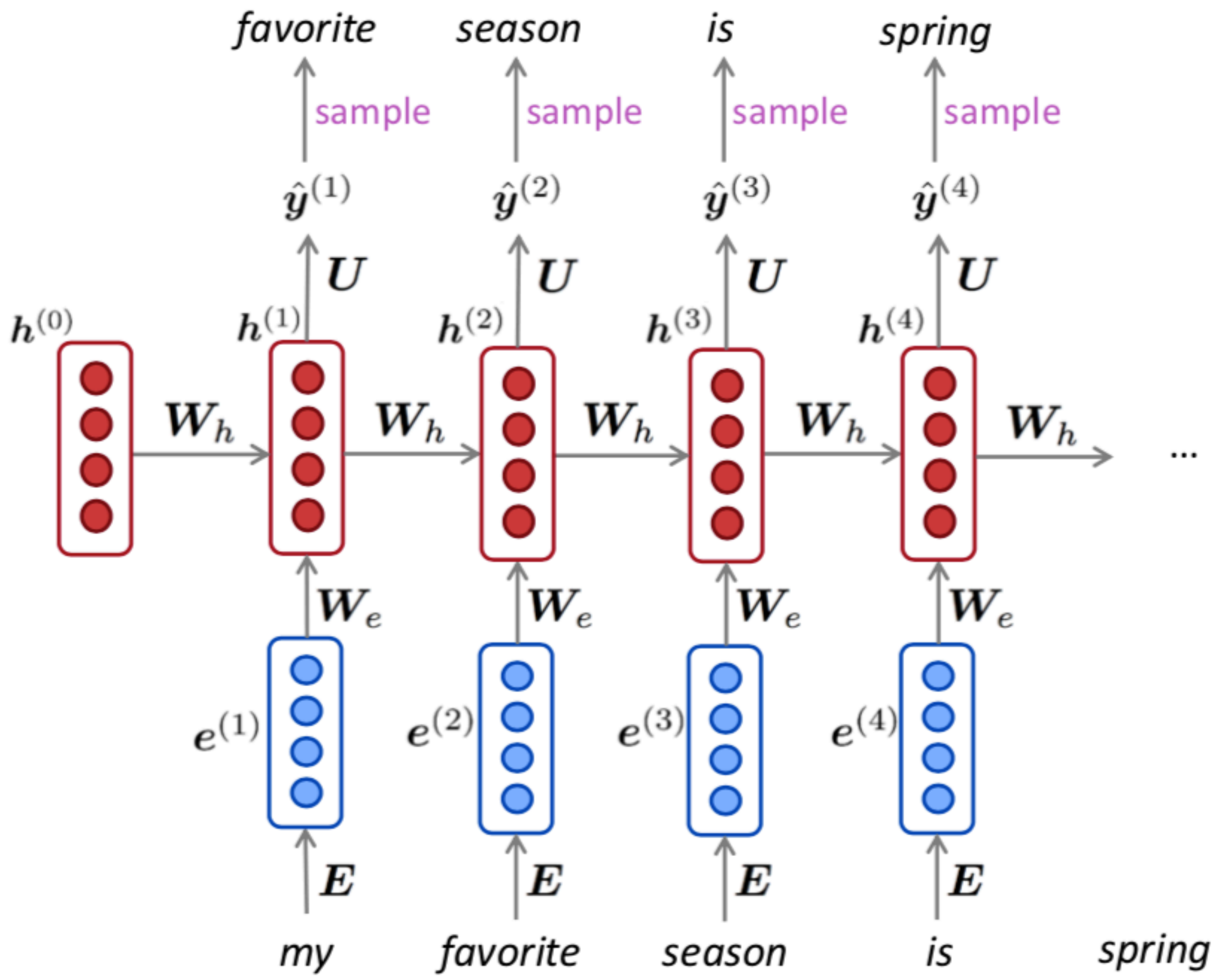
Model	#Param	Validation	Test
Mikolov & Zweig (2012) – RNN-LDA + KN-5 + cache	9M‡	-	92.0
Zaremba et al. (2014) – LSTM	20M	86.2	82.7
Gal & Ghahramani (2016) – Variational LSTM (MC)	20M	-	78.6
Kim et al. (2016) – CharCNN	19M	-	78.9
Merity et al. (2016) – Pointer Sentinel-LSTM	21M	72.4	70.9
Grave et al. (2016) – LSTM + continuous cache pointer†	-	-	72.1
Inan et al. (2016) – Tied Variational LSTM + augmented loss	24M	75.7	73.2
Zilly et al. (2016) – Variational RHN	23M	67.9	65.4
Zoph & Le (2016) – NAS Cell	25M	-	64.0
Melis et al. (2017) – 2-layer skip connection LSTM	24M	60.9	58.3
Merity et al. (2017) – AWD-LSTM w/o finetune	24M	60.7	58.8
Merity et al. (2017) – AWD-LSTM	24M	60.0	57.3
Ours – AWD-LSTM-MoS w/o finetune	22M	58.08	55.97
Ours – AWD-LSTM-MoS	22M	**56.54**	**54.44**
Merity et al. (2017) – AWD-LSTM + continuous cache pointer†	24M	53.9	52.8
Krause et al. (2017) – AWD-LSTM + dynamic evaluation†	24M	51.6	51.1
Ours – AWD-LSTM-MoS + dynamic evaluation†	22M	**48.33**	**47.69**

(Yang et al, 2018): Breaking the Softmax Bottleneck: A High-Rank RNN Language Model

Overview

- Review of Vanilla RNN
- Training RNNs
- Issues with Gradient Flows
- LSTMs and GRUs
- Applications
- Variants: Stacked RNNs, Bidirectional RNNs

Application: Text Generation



You can generate text by **repeated sampling.**

Sampled output is next step's input.

Fun with RNNs

Obama speeches

Good afternoon. God bless you.

The United States will step up to the cost of a new challenges of the American people that will share the fact that we created the problem. They were attacked and so that they have to say that all the task of the final days of war that I will not be able to get this done. The promise of the men and women who were still going to take out the fact that the American people have fought to make sure that they have to be able to protect our part. It was a chance to stand together to completely look for the commitment to borrow from the American people. And the fact is the men and women in uniform and the millions of our country with the law system that we should be a strong stretcks of the forces that we can afford to increase our spirit of the American people and the leadership of our country who are on the Internet of American lives.

Thank you very much. God bless you, and God bless the United States of America.

Latex generation

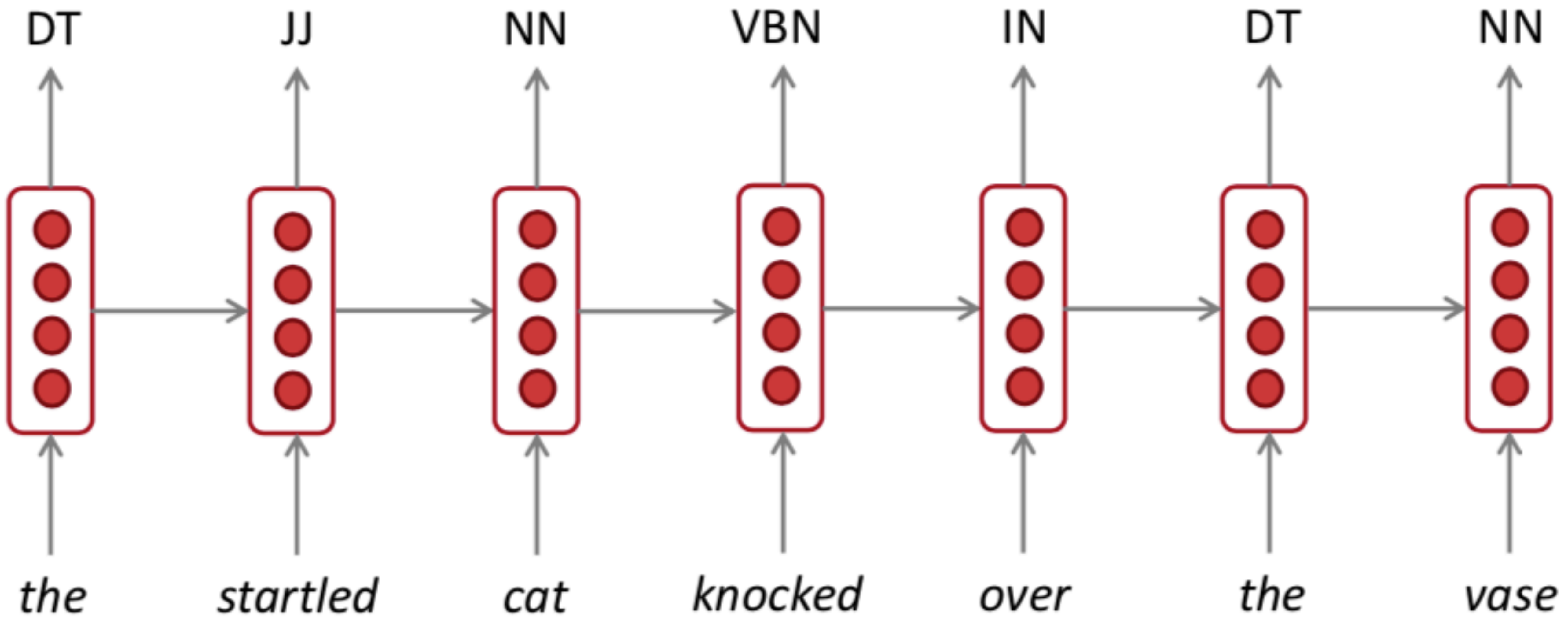
```
\begin{proof}
We may assume that $\mathcal{I}$ is an abelian sheaf on $\mathcal{C}$.
\item Given a morphism $\Delta : \mathcal{F} \to \mathcal{I}$
is an injective and let $\mathfrak q$ be an abelian sheaf on $X$.
Let $\mathcal{F}$ be a fibered complex. Let $\mathcal{F}$ be a category.
\begin{enumerate}
\item \hyperref[setain-construction-phantom]{Lemma}
\label{lemma-characterize-quasi-finite}
Let $\mathcal{F}$ be an abelian quasi-coherent sheaf on $\mathcal{C}$.
Let $\mathcal{F}$ be a coherent $\mathcal{O}_X$-module. Then
$\mathcal{F}$ is an abelian catenary over $\mathcal{C}$.
\item The following are equivalent
\begin{enumerate}
\item $\mathcal{F}$ is an $\mathcal{O}_X$-module.
\end{lemma}
```

Andrej Karpathy “The Unreasonable Effectiveness of Recurrent Neural Networks”

Application: Sequence Tagging

Input: a sentence of n words: $x_1, \ldots, x_n$

Output: $y_1, \ldots, y_n, y_i \in \{1, \ldots C\}$



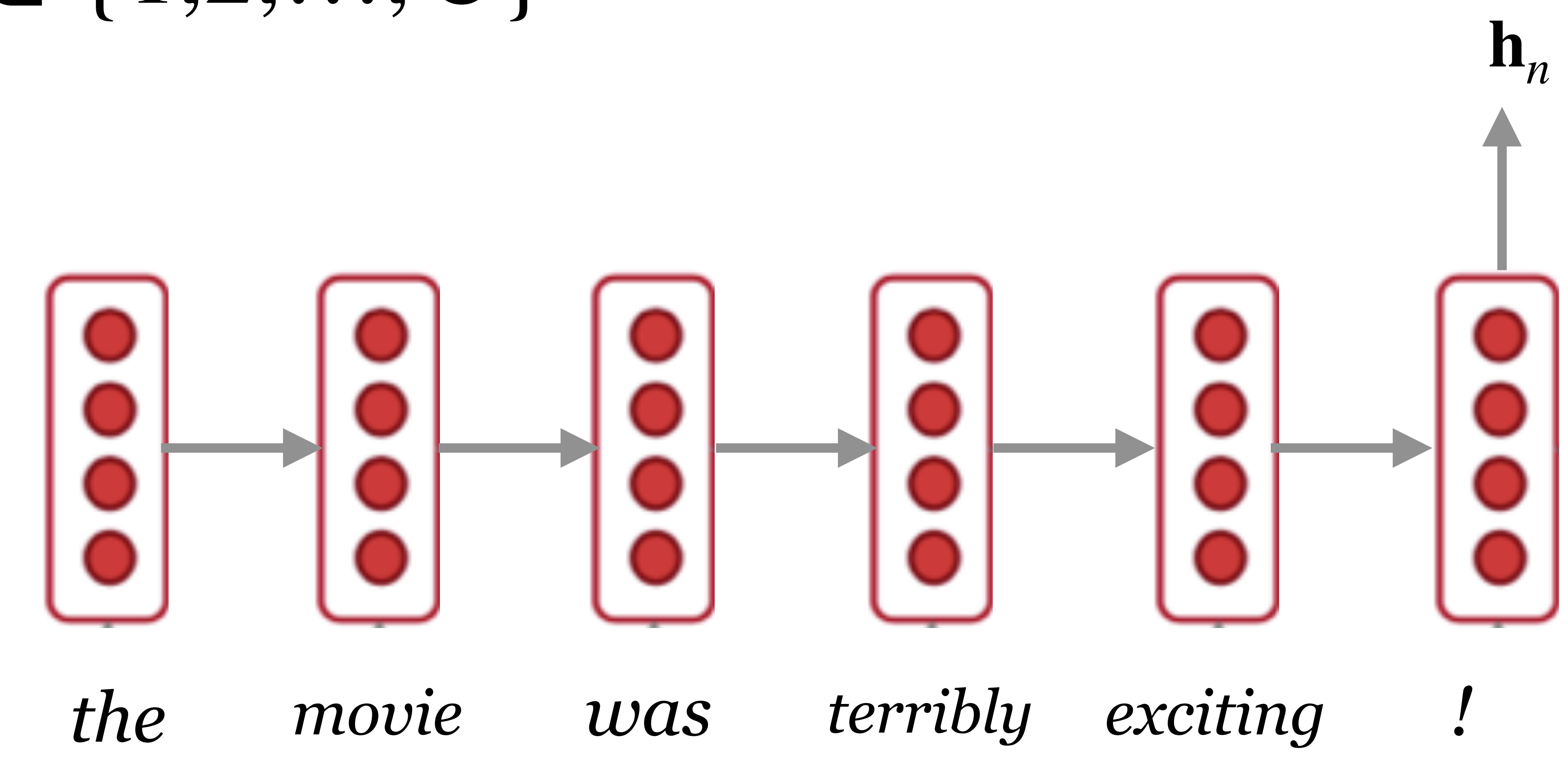
$$P(y_i = k) = softmax_k(\mathbf{W}_o \mathbf{h}_i) \qquad \mathbf{W}_o \in \mathbb{R}^{C \times d}$$

$$L = -\frac{1}{n} \sum_{i=1}^{n} \log P(y_i = k)$$

Application: Text Classification

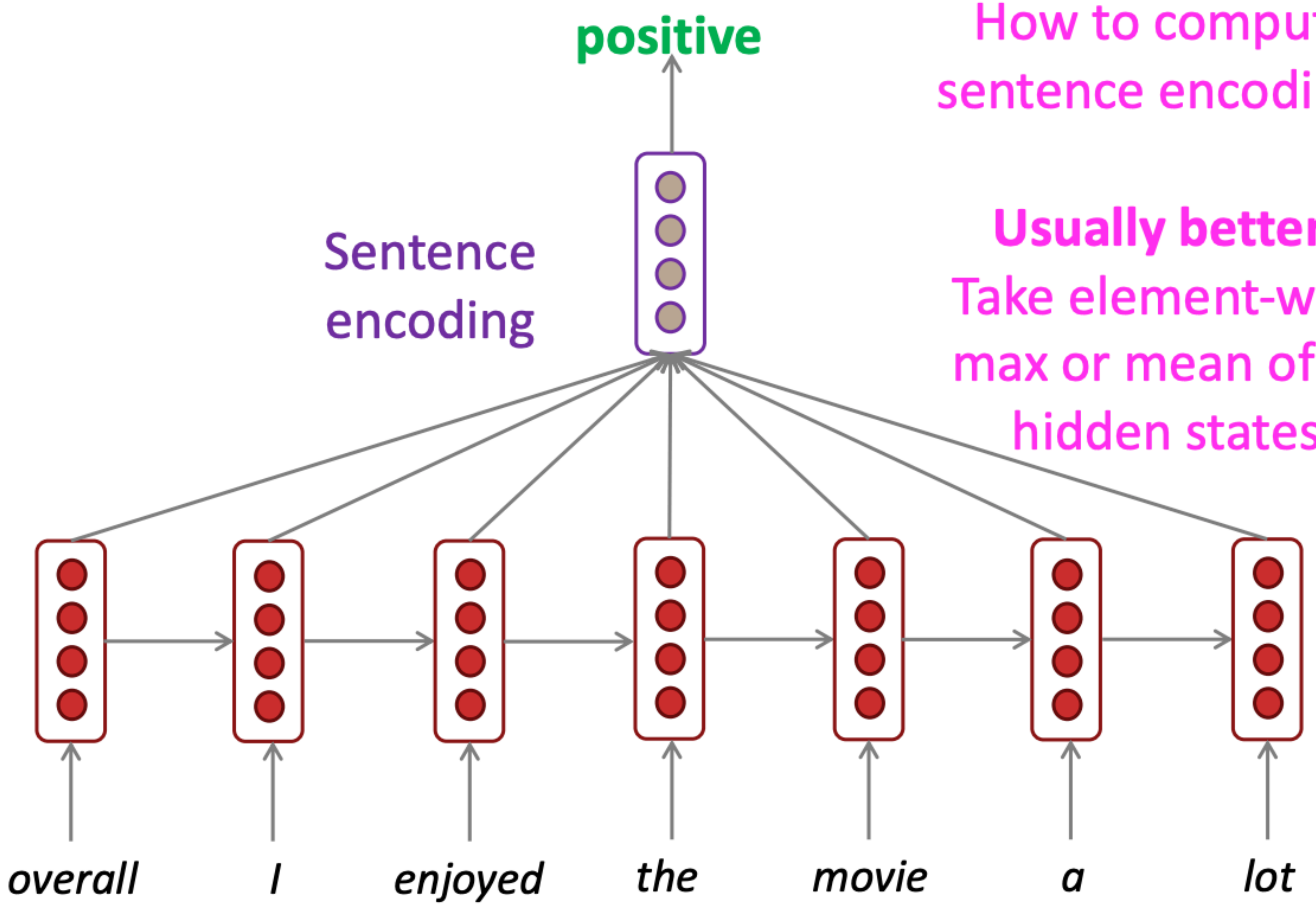
Input: a sentence of n words

Output: $y \in \{1, 2, \ldots, C\}$



$$P(y = k) = softmax_k(\mathbf{W}_o \mathbf{h}_n) \qquad \mathbf{W}_o \in \mathbb{R}^{C \times d}$$

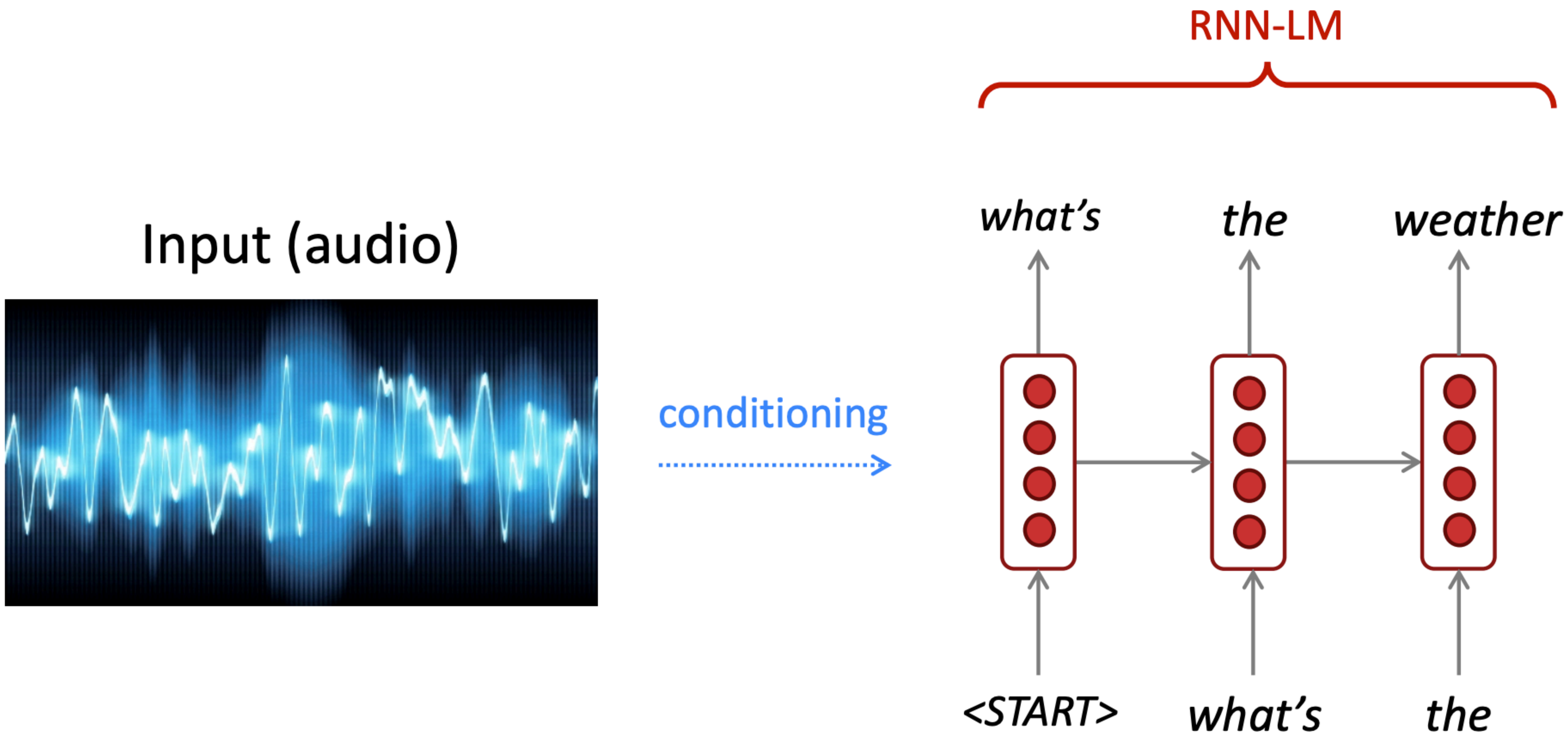
Application: Text classification



How to compute sentence encoding?

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

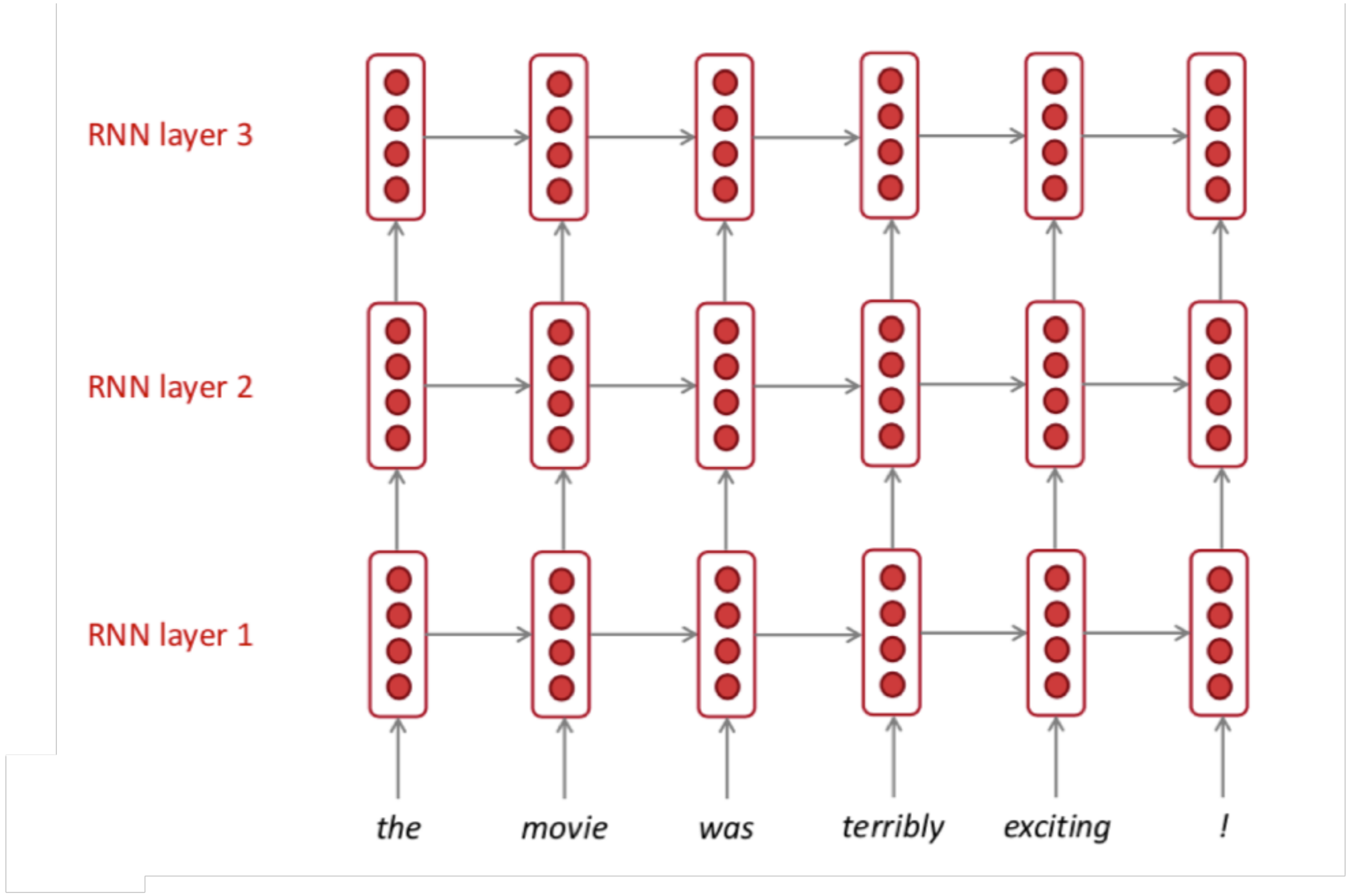
Conditional Text Generation



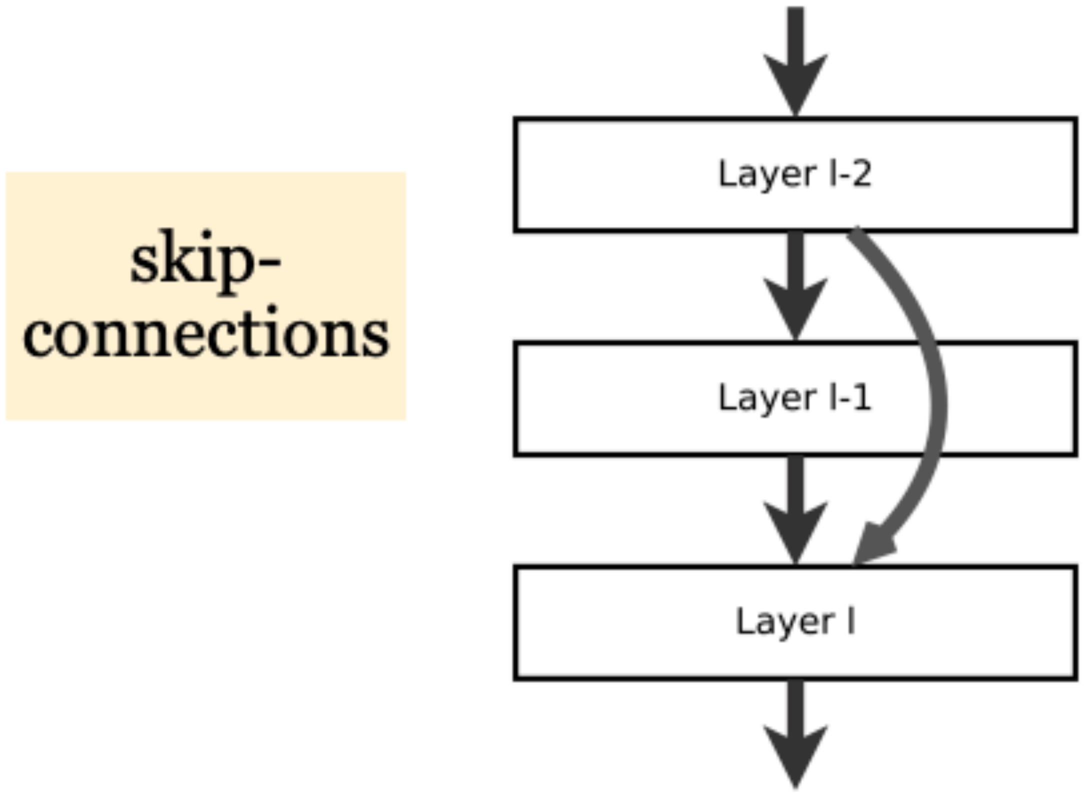
Multi-layer RNNs

- RNNs are already “deep” on one dimension (unroll over time steps)
- We can also make them “deep” in another dimension by applying multiple RNNs
- Multi-layer RNNs are also called stacked RNNs.

Stacking multi-layered RNNs



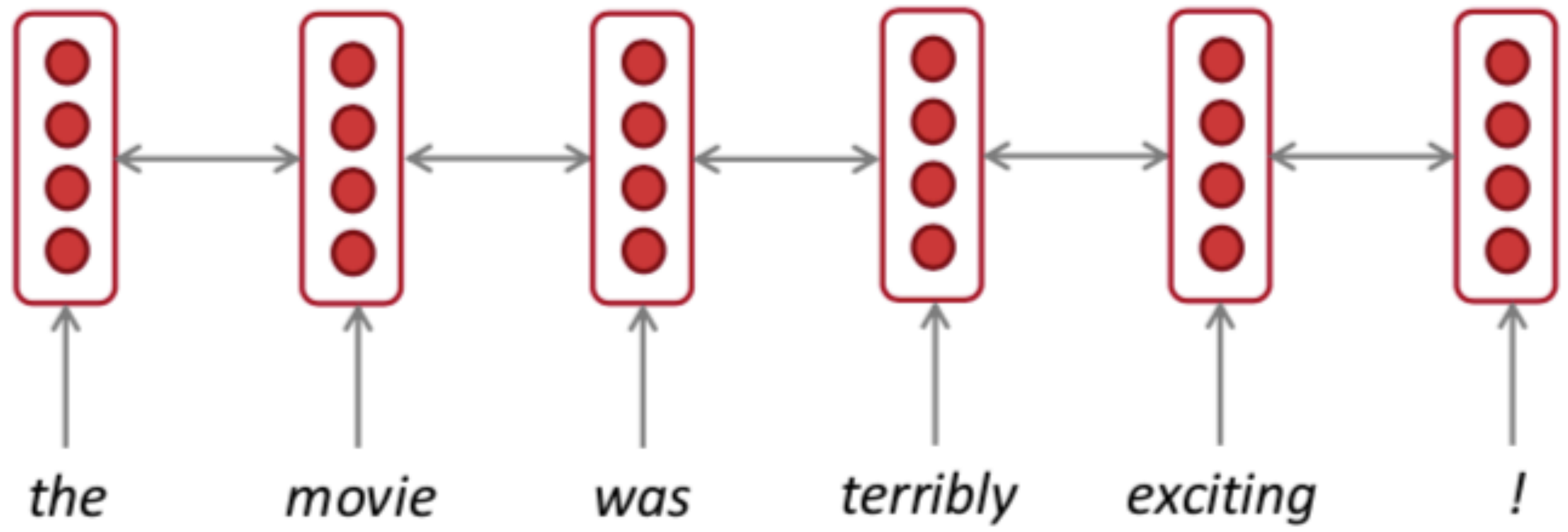
- The hidden states from RNN layer i are the inputs to RNN layer $i + 1$



- In practice, using 2 to 4 layers is common (usually better than 1 layer)
- Transformer-based networks can be up to 24 layers with lots of skip-connections.

Bidirectional RNNs

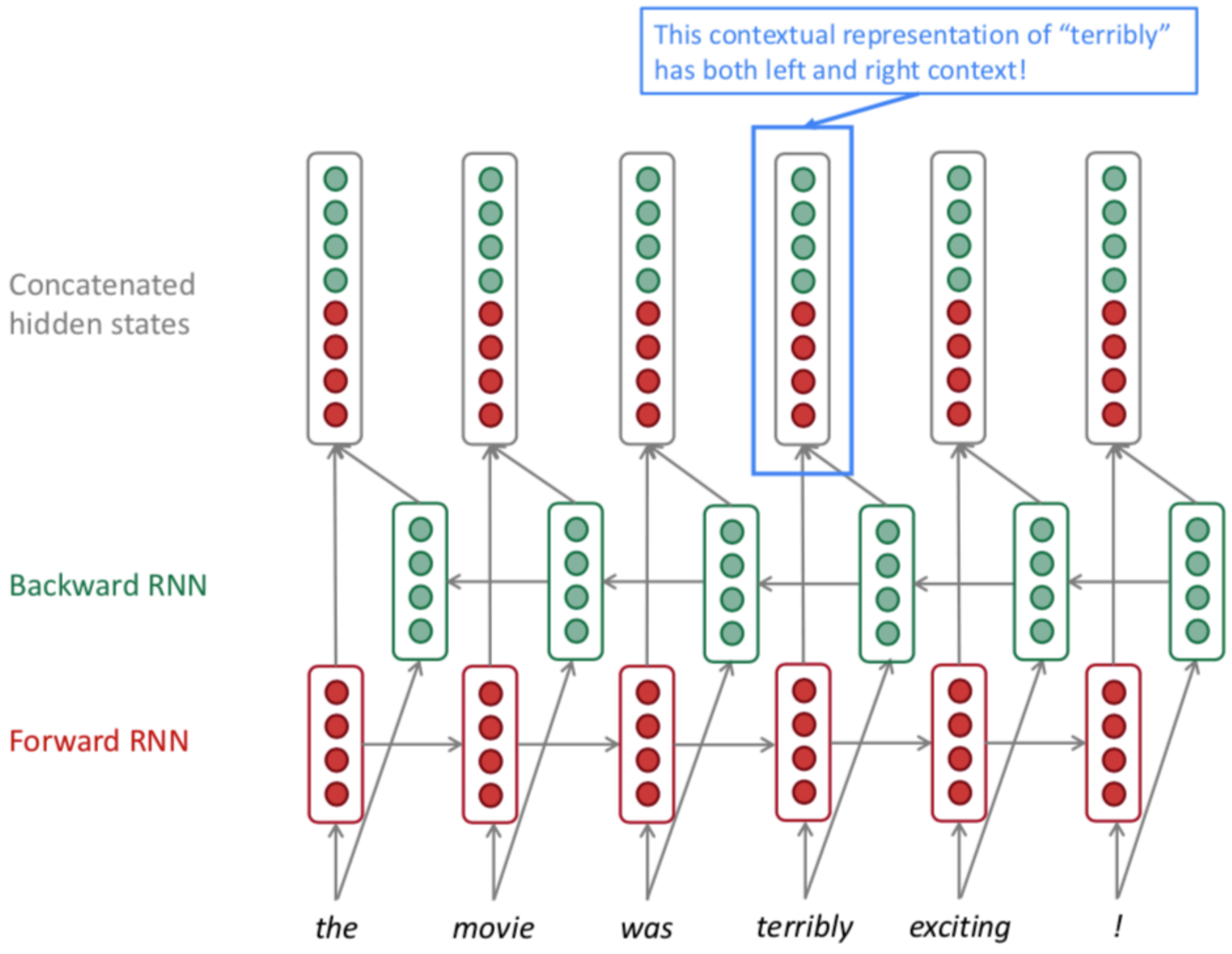
- Bidirectionality is important in language representations:



terribly:

- left context "the movie was"
- right context "exciting !"

Bidirectional RNNs



$$\boxed{\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t) \in \mathbb{R}^d}$$

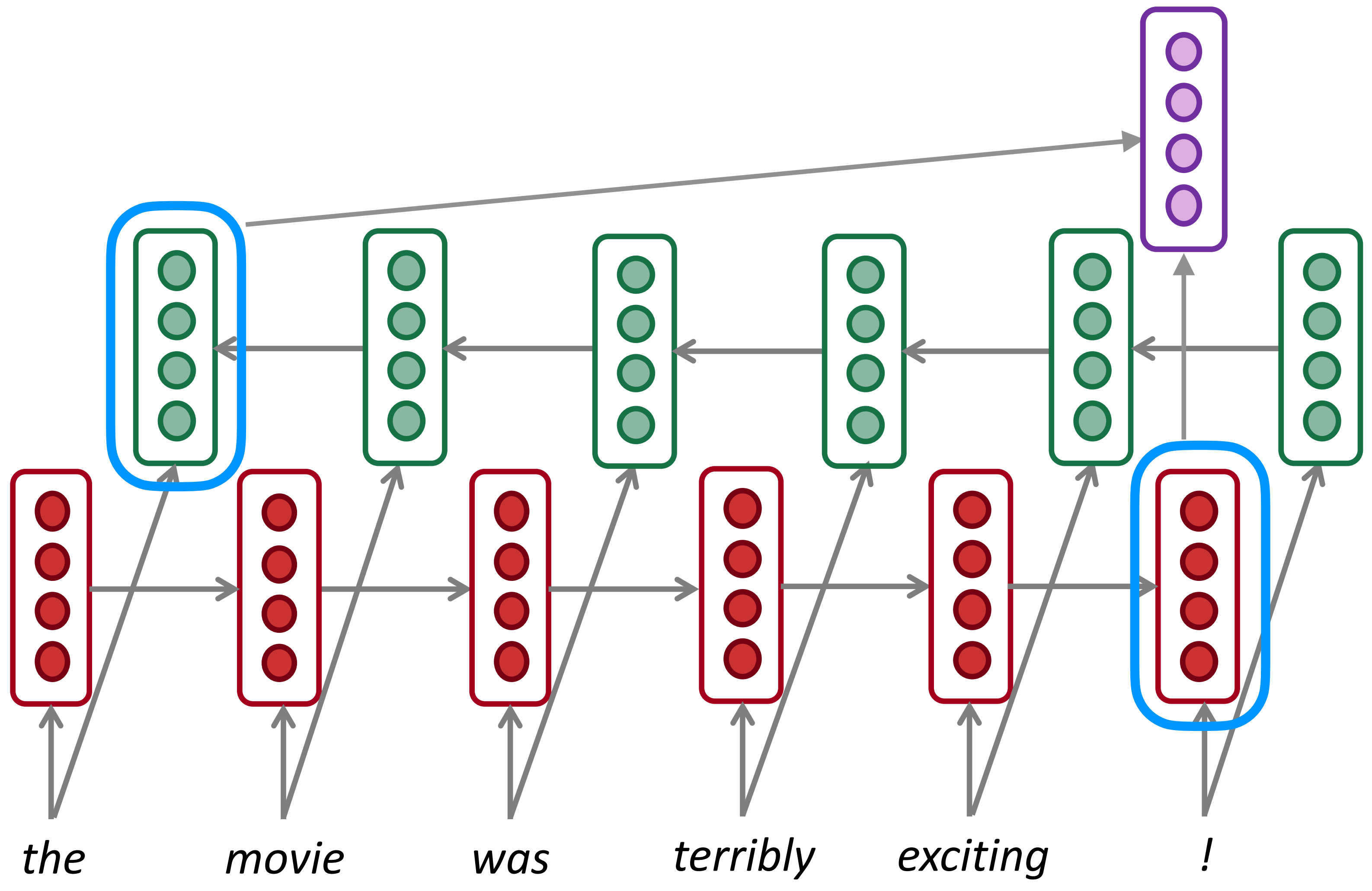
$$\overrightarrow{\mathbf{h}}_t = f_1(\overrightarrow{\mathbf{h}}_{t-1}, \mathbf{x}_t), t = 1,2,\ldots n$$

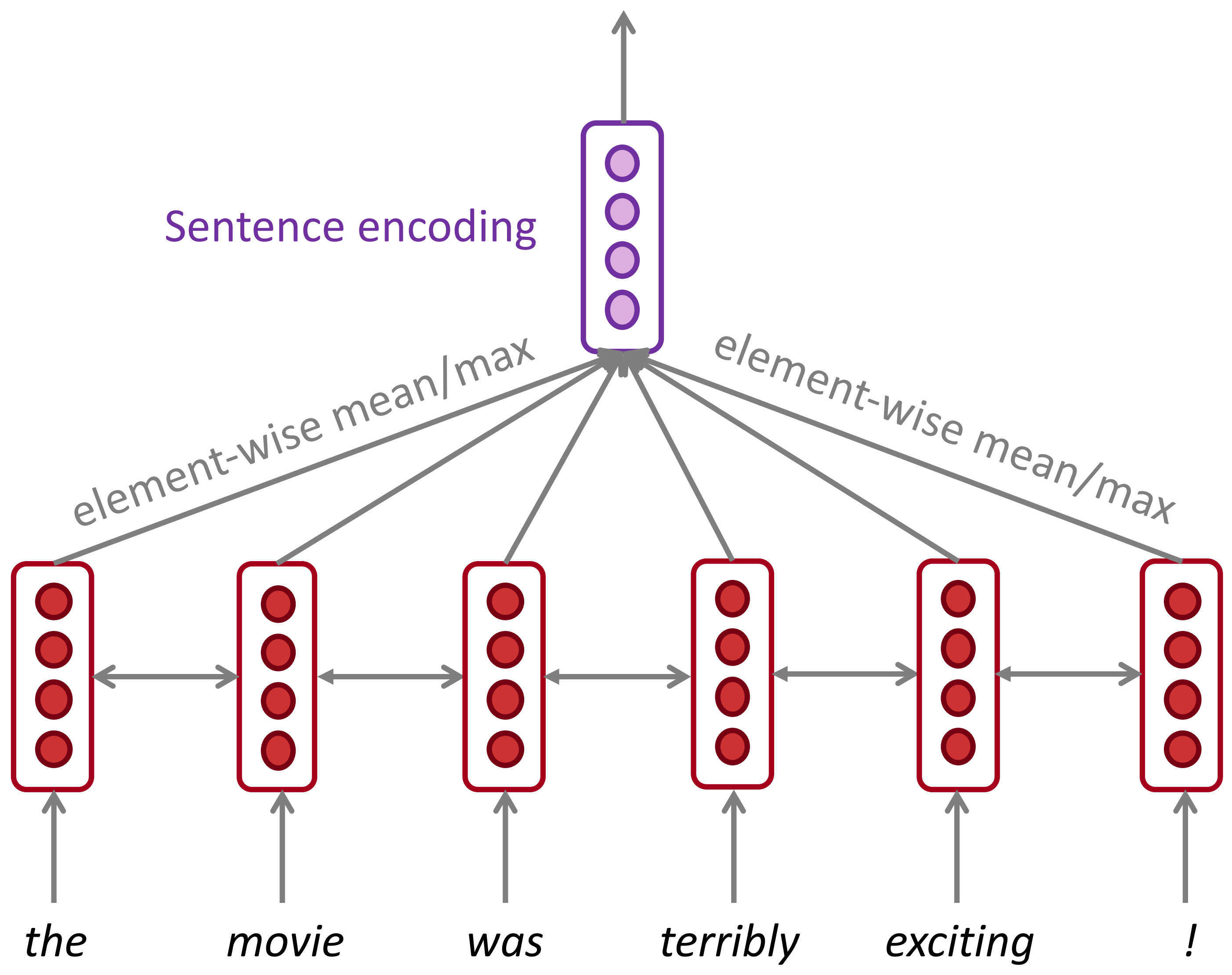
$$\overleftarrow{\mathbf{h}}_t = f_2(\overleftarrow{\mathbf{h}}_{t+1}, \mathbf{x}_t), t = n, n-1,\ldots 1$$

$$\mathbf{h}_t = [\overleftarrow{\mathbf{h}}_t, \overrightarrow{\mathbf{h}}_t] \in \mathbb{R}^{2d}$$

Bidirectional RNNs

- Sequence tagging: Yes!
- Text classification: Yes! With slight modifications.





- Text generation: No. Why?