

CSCE 5218 & 4930

Deep Learning

Recurrent Neural Networks

Plan for this lecture

- Recurrent neural networks
 - Basics
 - Training (backprop through time, vanishing gradient)
 - Recurrent networks with gates (GRU, LSTM)
- Applications in NLP and vision
 - Neural machine translation (beam search, attention)
 - Image/video captioning

Recurrent neural networks

Some pre-RNN captioning results



This is a picture of one sky,
one road and one sheep. The
gray sky is over the gray road.
The gray sheep is by the gray
road.



Here we see one road, one
sky and one bicycle. The
road is near the blue sky, and
near the colorful bicycle. The
colorful bicycle is within the
blue sky.



This is a picture of two dogs.
The first dog is near the
second furry dog.

Results with Recurrent Neural Networks



"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."



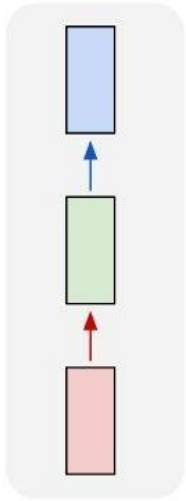
"two young girls are playing with lego toy."



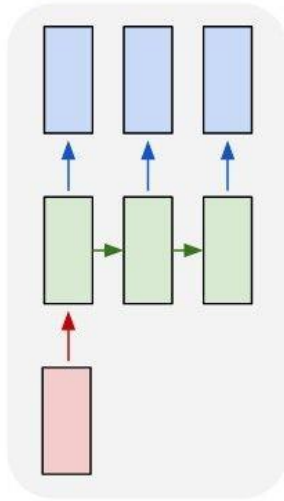
"boy is doing backflip on wakeboard."

Recurrent Networks offer a lot of flexibility:

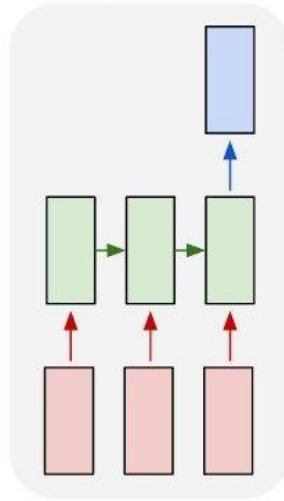
one to one



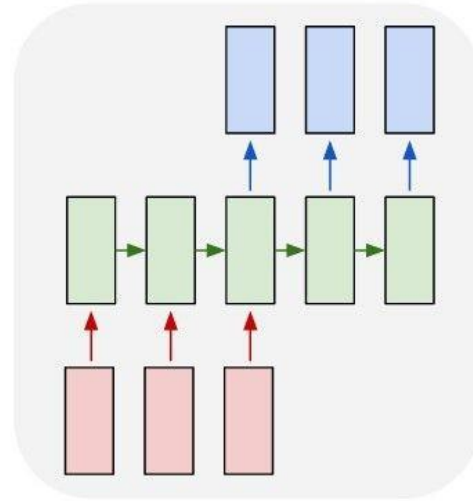
one to many



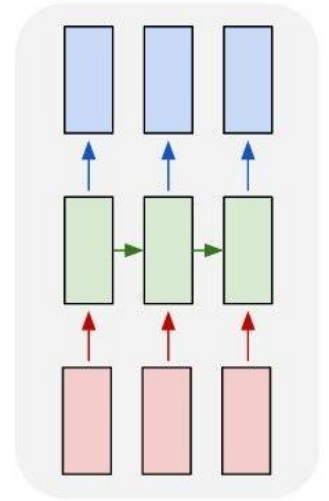
many to one



many to many



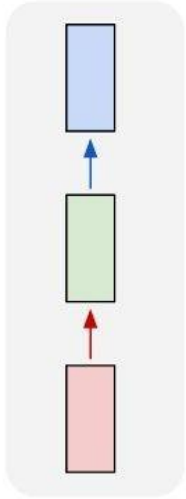
many to many



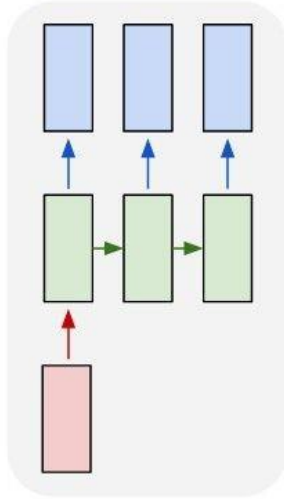
vanilla neural networks

Recurrent Networks offer a lot of flexibility:

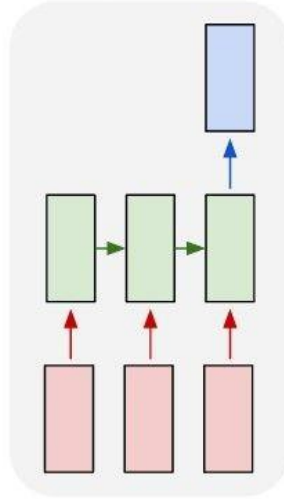
one to one



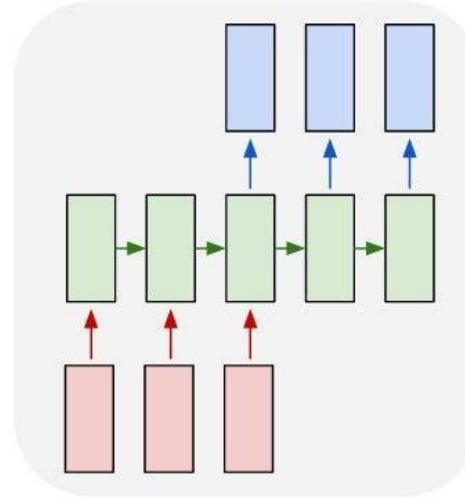
one to many



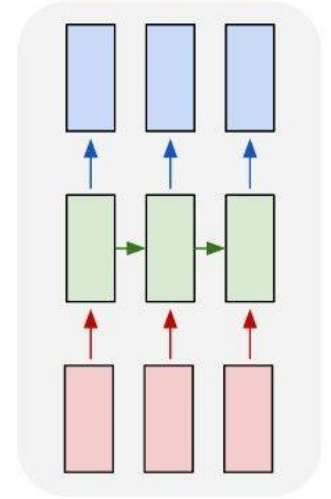
many to one



many to many



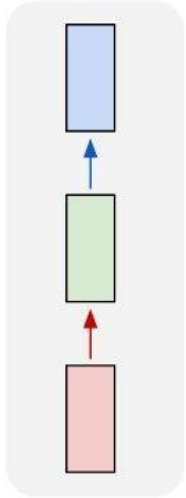
many to many



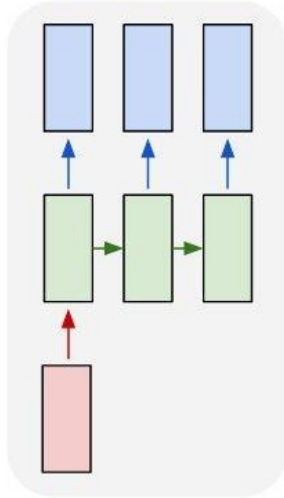
↖ e.g. **image captioning**
image -> sequence of words

Recurrent Networks offer a lot of flexibility:

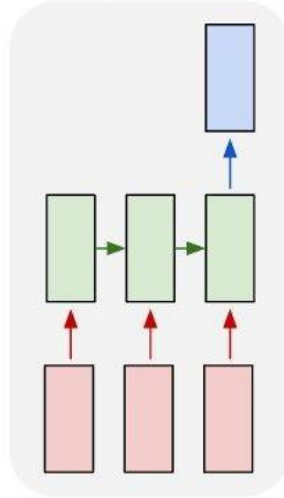
one to one



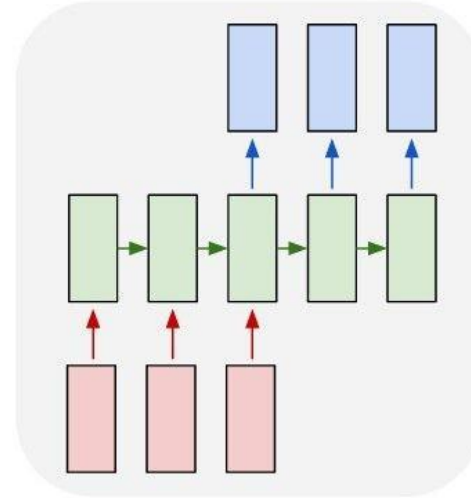
one to many



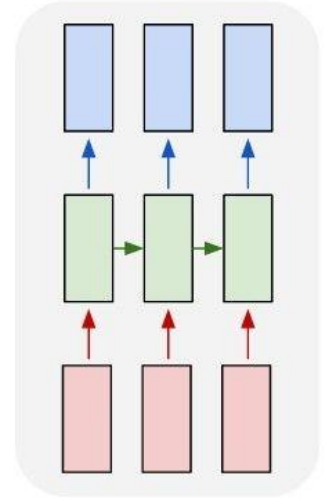
many to one



many to many



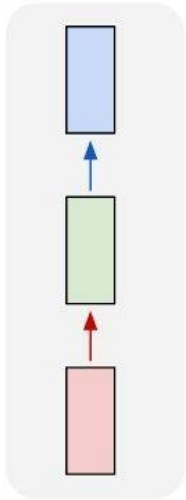
many to many



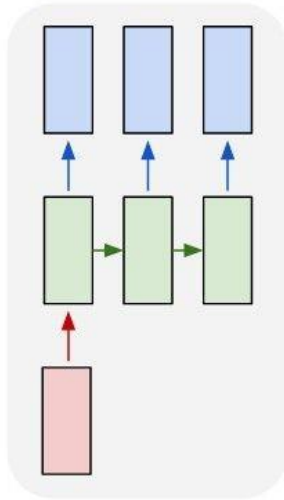
e.g. **sentiment classification**
sequence of words -> sentiment

Recurrent Networks offer a lot of flexibility:

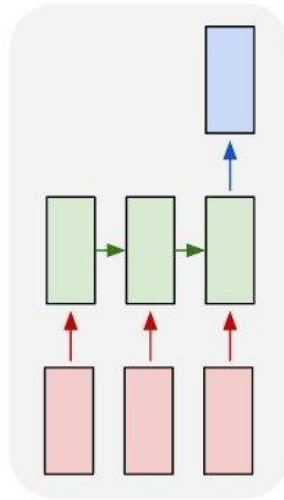
one to one



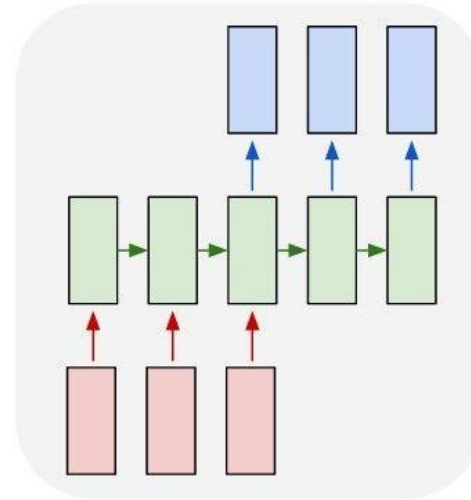
one to many



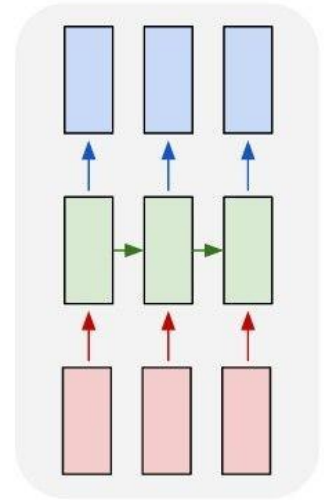
many to one



many to many



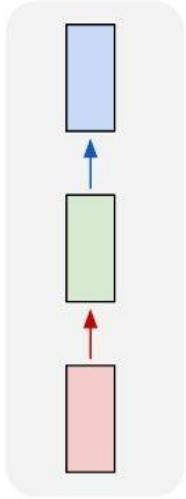
many to many



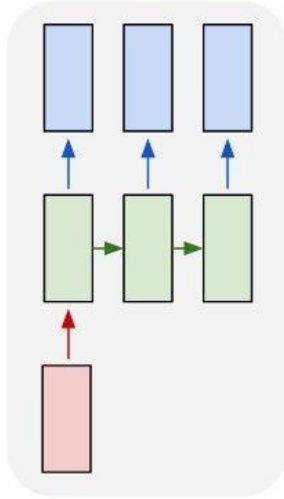
e.g. **machine translation**
seq of words \rightarrow seq of words

Recurrent Networks offer a lot of flexibility:

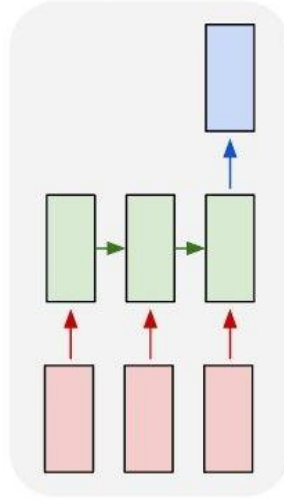
one to one



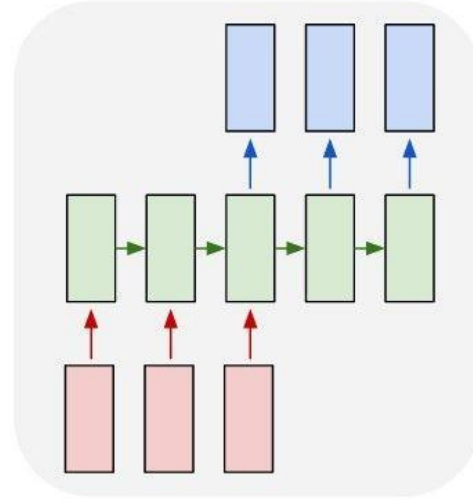
one to many



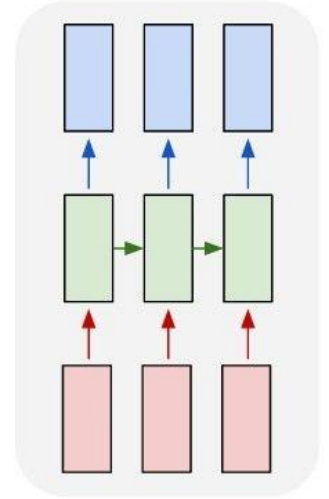
many to one



many to many



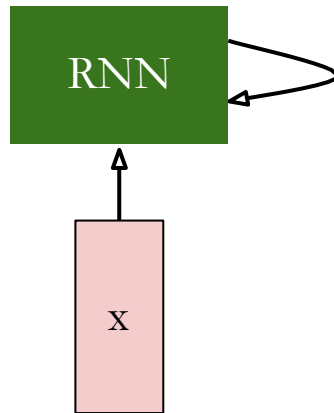
many to many



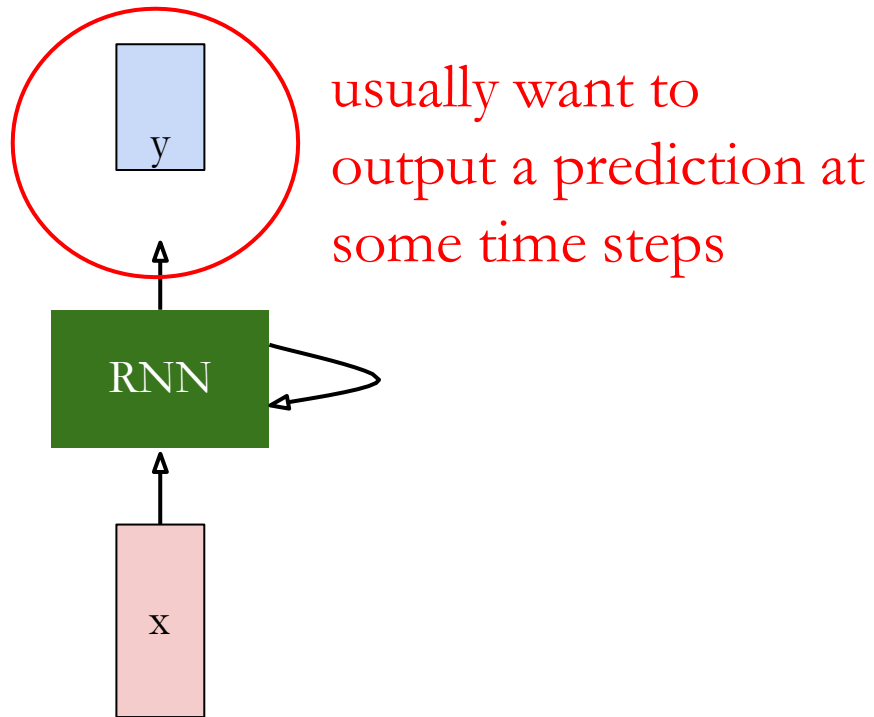
e.g. video classification on frame level



Recurrent Neural Network



Recurrent Neural Network



Recurrent Neural Network

We can process a sequence of vectors \mathbf{x} by applying a recurrence formula at every time step:

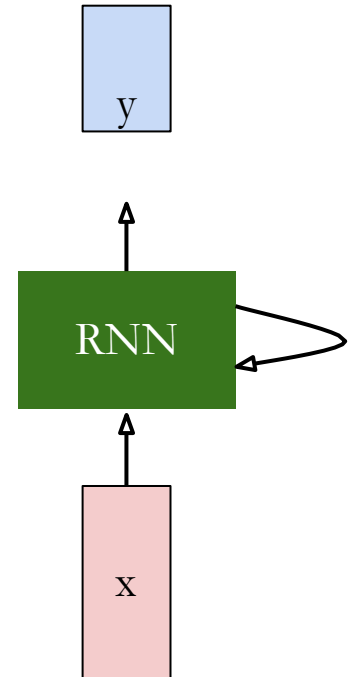
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

some function with parameters W

old state

input vector at some time step

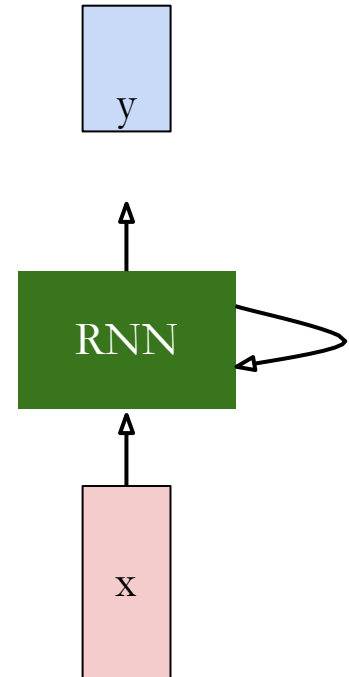


Recurrent Neural Network

We can process a sequence of vectors \mathbf{x} by applying a recurrence formula at every time step:

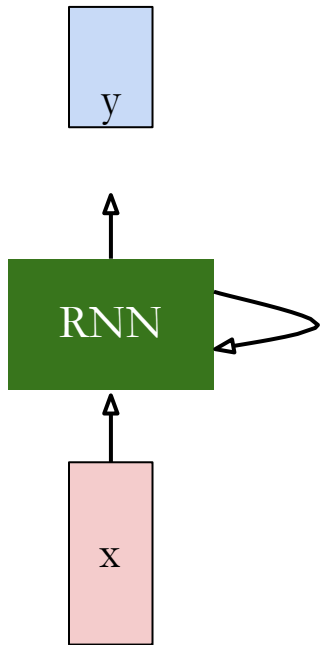
$$h_t = f_W(h_{t-1}, x_t)$$

Notice: the same function and the same set of parameters are used at every time step.



(Vanilla) Recurrent Neural Network

The state consists of a single “*hidden*” vector \mathbf{h} :



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



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

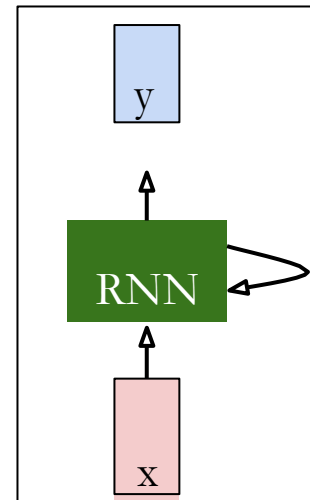
$$y_t = W_{hy}h_t$$

Example

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence: **“hello”**

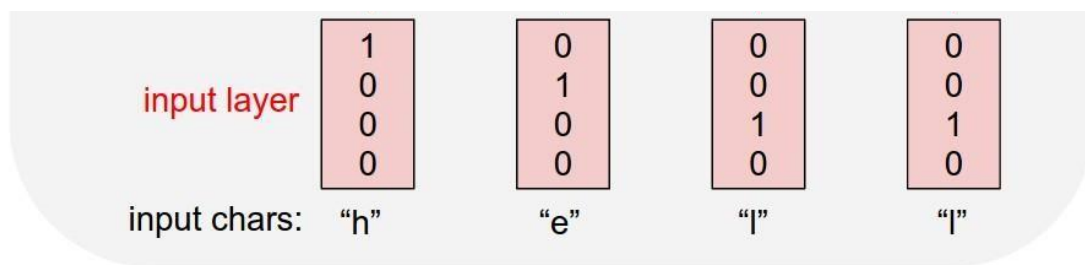


Example

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence: **“hello”**



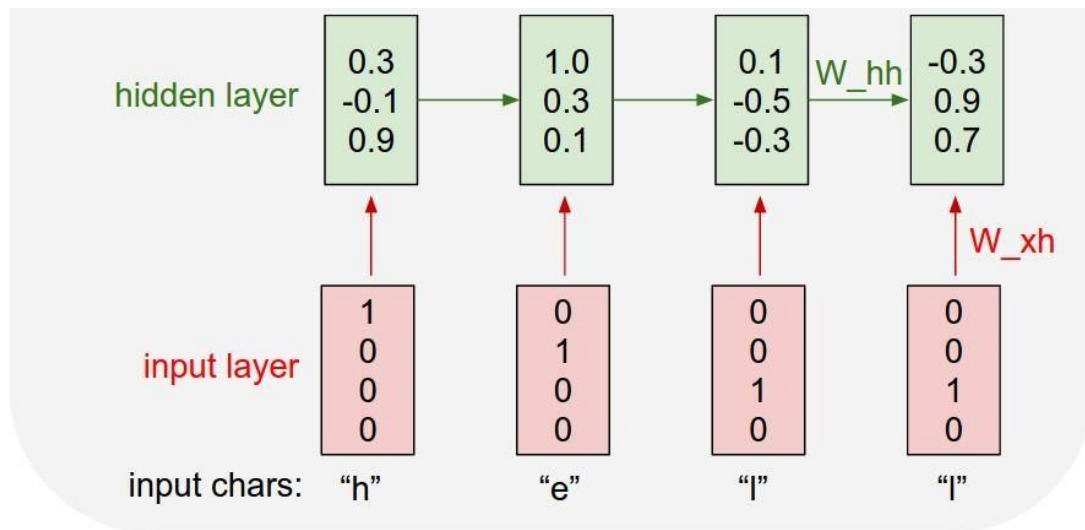
Example

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence: “hello”

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

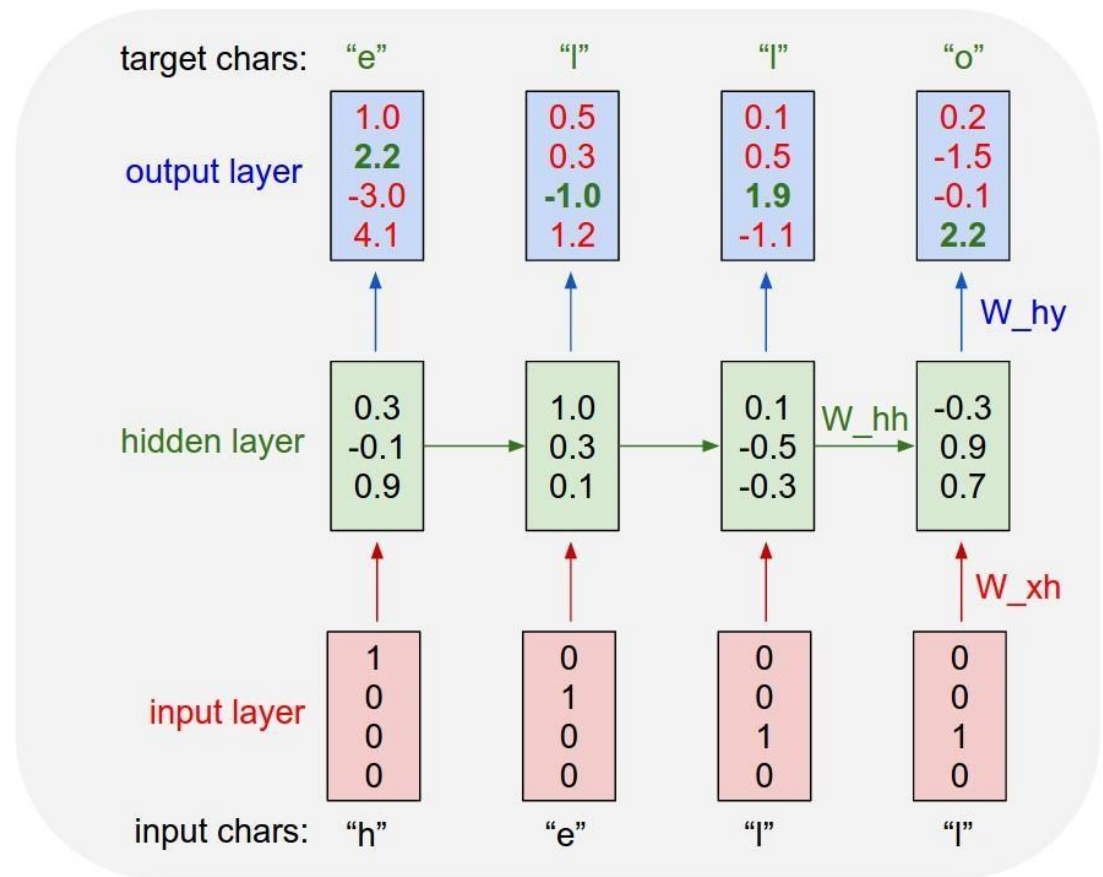


Example

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence: “hello”



What kind of loss can we formulate?

Training a Recurrent Neural Network

- Get a **big corpus of text** which is a sequence of words $x^{(1)}, \dots, x^{(T)}$
- Feed into RNN; compute output distribution $\hat{y}^{(t)}$ **for every step t** .
 - i.e. predict probability distribution of *every word*, given words so far
- **Loss function** on step t is **cross-entropy** between predicted probability distribution $\hat{y}^{(t)}$, and true next word $y^{(t)}$ (one-hot); V is vocabulary

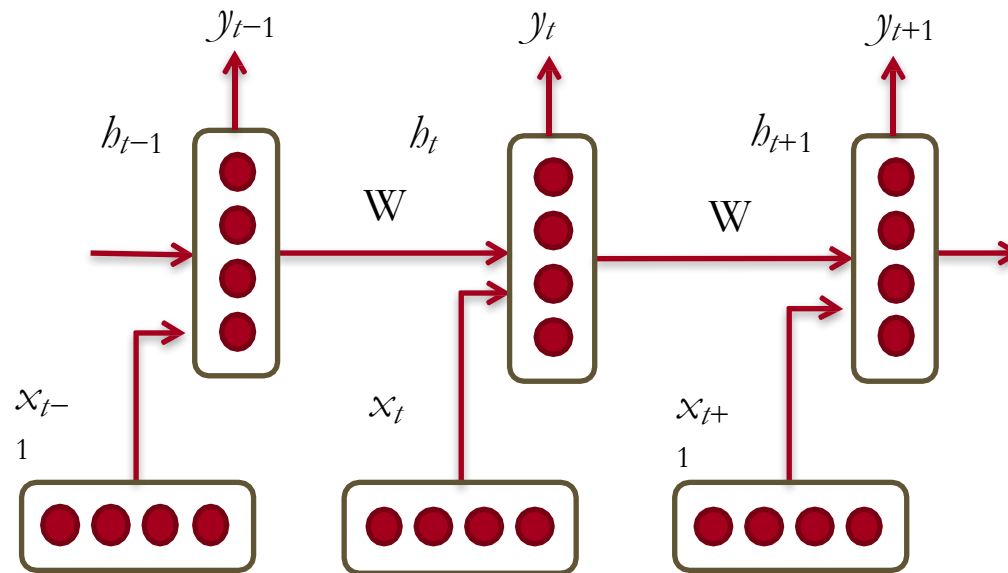
$$J^{(t)}(\theta) = CE(y^{(t)}, \hat{y}^{(t)}) = - \sum_{w \in V} y_w^{(t)} \log \hat{y}_w^{(t)} = - \log \hat{y}_{x_{t+1}}^{(t)}$$

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

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

The vanishing/exploding gradient problem

- The error at a time step ideally can tell a previous time step from many steps away to change during backprop
- Multiply the same matrix at each time step during backprop



The vanishing gradient problem

- Total error is the sum of each error at time steps t

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

- Chain rule:

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

- More chain rule:

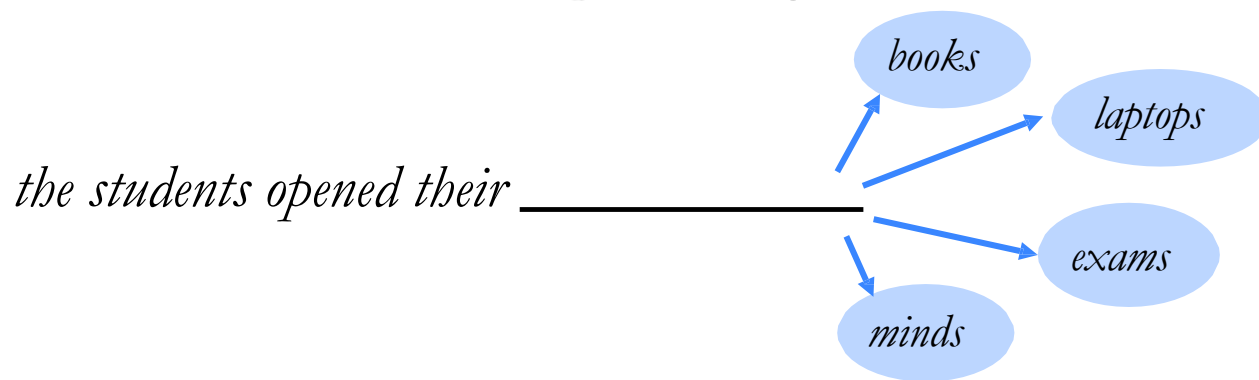
$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{j-1}}$$

- Derivative of vector wrt vector is a Jacobian matrix of partial derivatives; norm of this matrix can become very small or very large quickly [Bengio et al 1994, Pascanu et al. 2013], leading to vanishing/exploding gradient

Now in more detail...

Language Modeling

- **Language Modeling** is the task of predicting what word comes next.



- More formally: given a sequence of words $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(t)}$, compute the probability distribution of the next word $\mathbf{x}^{(t+1)}$:

$$P(\mathbf{x}^{(t+1)} \mid \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)})$$

where $\mathbf{x}^{(t+1)}$ can be any word in the vocabulary $V = \{\mathbf{w}_1, \dots, \mathbf{w}_{|V|}\}$

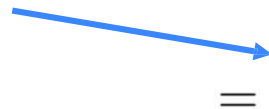
- A system that does this is called a **Language Model**.

n-gram Language Models

- First we make a **simplifying assumption**: $\mathbf{x}^{(t+1)}$ depends only on the preceding $n-1$ words.

$$P(\mathbf{x}^{(t+1)} | \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)}) = P(\mathbf{x}^{(t+1)} | \overbrace{\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)}}^{n-1 \text{ words}}) \quad (\text{assumption})$$

prob of a n-gram



=

$$P(\mathbf{x}^{(t+1)}, \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})$$

prob of a (n-1)-gram



$$P(\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})$$

(definition of
conditional prob)

- Question:** How do we get these n -gram and $(n-1)$ -gram probabilities?

n-gram Language Models

- First we make a **simplifying assumption**: $\mathbf{x}^{(t+1)}$ depends only on the preceding $n-1$ words.

$$P(\mathbf{x}^{(t+1)} | \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)}) = P(\mathbf{x}^{(t+1)} | \overbrace{\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)}}^{n-1 \text{ words}}) \quad (\text{assumption})$$

prob of a n-gram \rightarrow

prob of a (n-1)-gram \rightarrow

$$= \frac{P(\mathbf{x}^{(t+1)}, \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})}{P(\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})} \quad (\text{definition of conditional prob})$$

- Question:** How do we get these n -gram and $(n-1)$ -gram probabilities?
- Answer:** By **counting** them in some large corpus of text!

$$\approx \frac{\text{count}(\mathbf{x}^{(t+1)}, \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})}{\text{count}(\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})} \quad (\text{statistical approximation})$$

Sparsity Problems with n-gram Language Models

Sparsity Problem 1

Problem: What if “*students opened their* w ” never occurred in data? Then w has probability 0!

(Partial) Solution: Add small δ to the count for every $w \in V$. This is called *smoothing*.

$$P(w|\text{students opened their}) = \frac{\text{count}(\text{students opened their } w)}{\text{count}(\text{students opened their})}$$

Sparsity Problem 2

Problem: What if “*students opened their*” never occurred in data? Then we can’t calculate probability for *any* w !

(Partial) Solution: Just condition on “*opened their*” instead. This is called *backoff*.

Note: Increasing n makes sparsity problems *worse*. Typically we can’t have n bigger than 5.

A fixed-window neural Language Model

output distribution

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

hidden layer

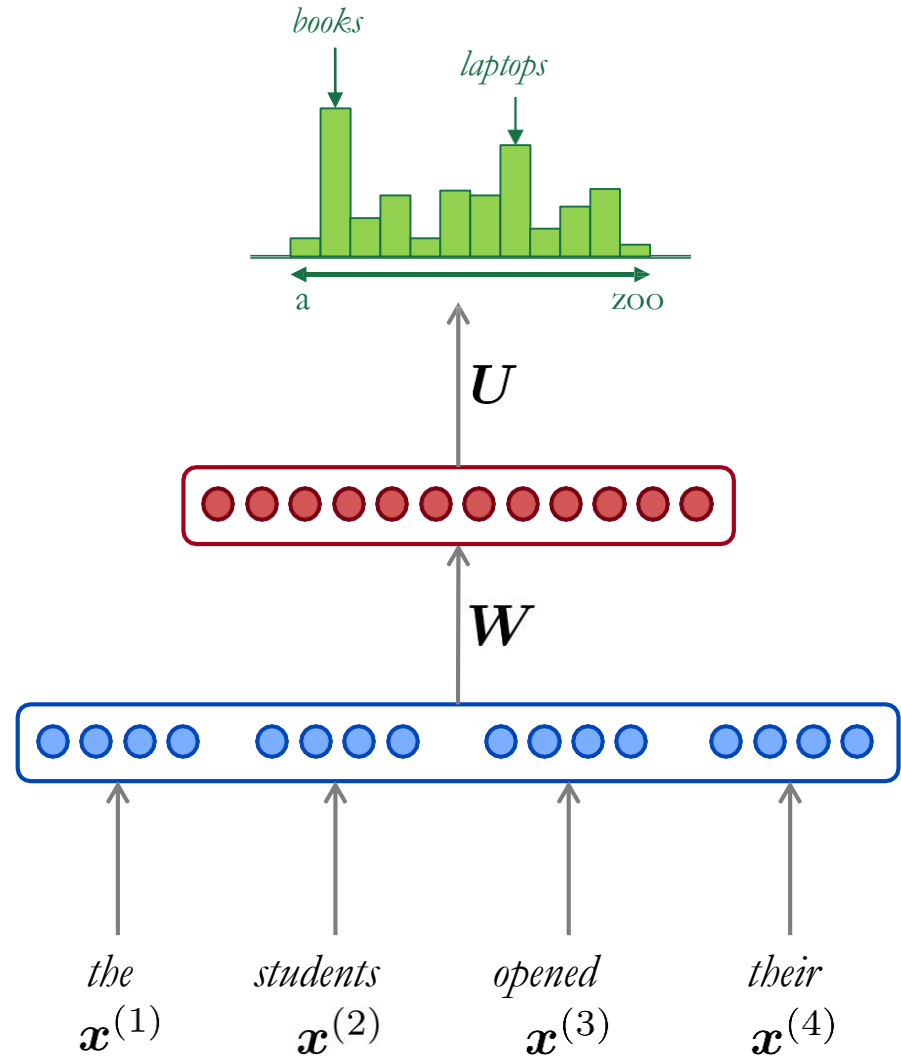
$$\mathbf{h} = f(\mathbf{W}\mathbf{e} + \mathbf{b}_1)$$

concatenated word embeddings

$$\mathbf{e} = [\mathbf{e}^{(1)}; \mathbf{e}^{(2)}; \mathbf{e}^{(3)}; \mathbf{e}^{(4)}]$$

words / one-hot vectors

$$\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \mathbf{x}^{(4)}$$



A fixed-window neural Language Model

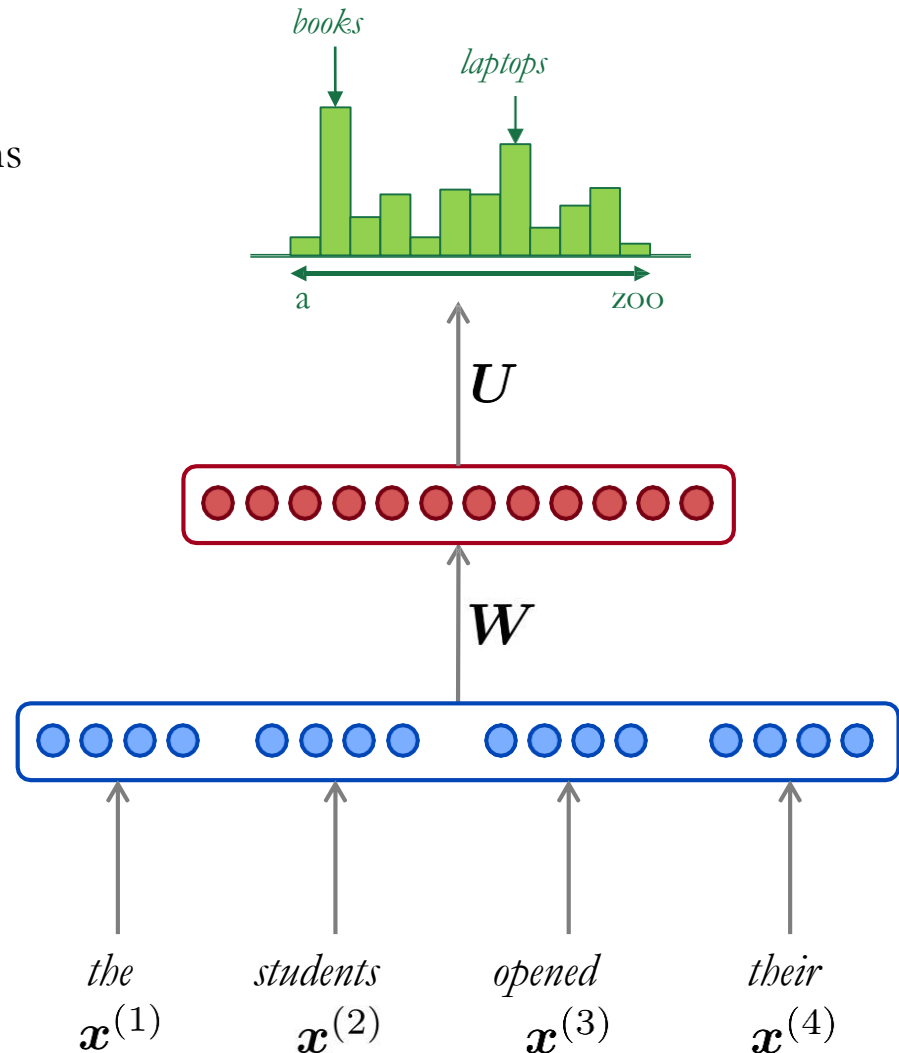
Improvements over n -gram LM:

- No sparsity problem
- Don't need to store all observed n -grams

Remaining **problems**:

- Fixed window is **too small**
- Enlarging window enlarges W
- Window can never be large enough!

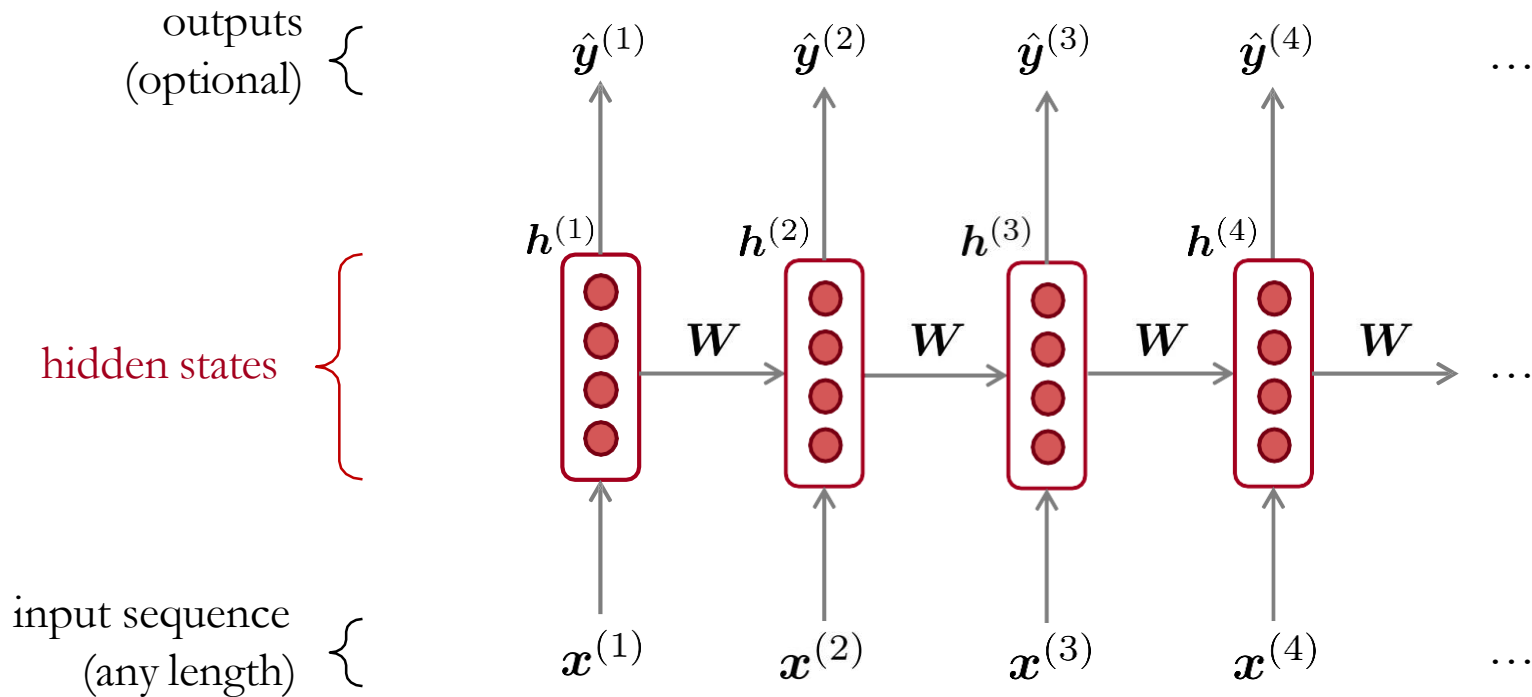
We need a neural architecture that can process *any length input*



Recurrent Neural Networks (RNN)

A family of neural architectures

Core idea: Apply the same weights \mathbf{W} repeatedly



A RNN Language Model

output distribution

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

hidden states

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

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

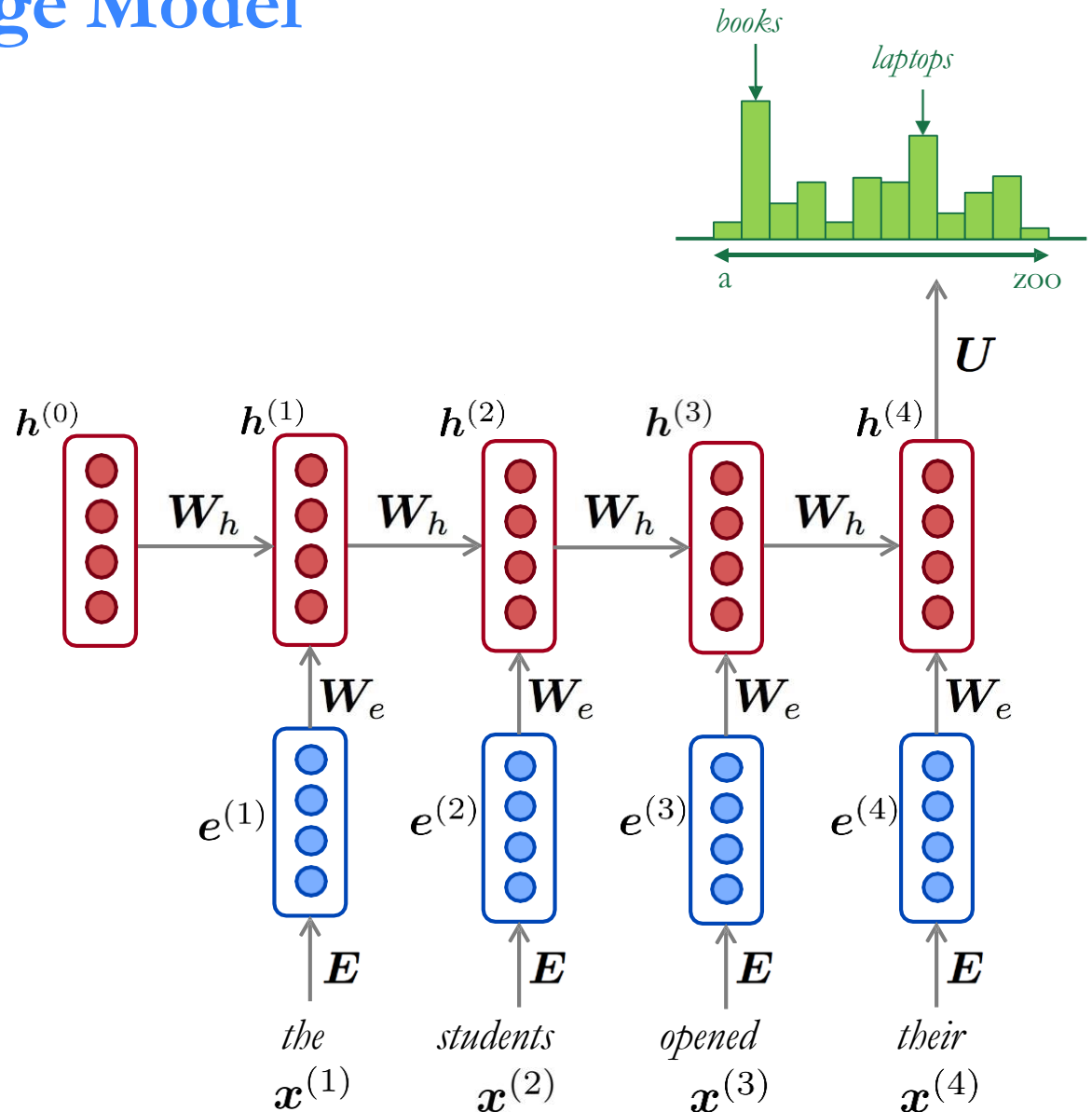
word embeddings

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

words / one-hot vectors

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

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



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

A RNN Language Model

RNN

Advantages:

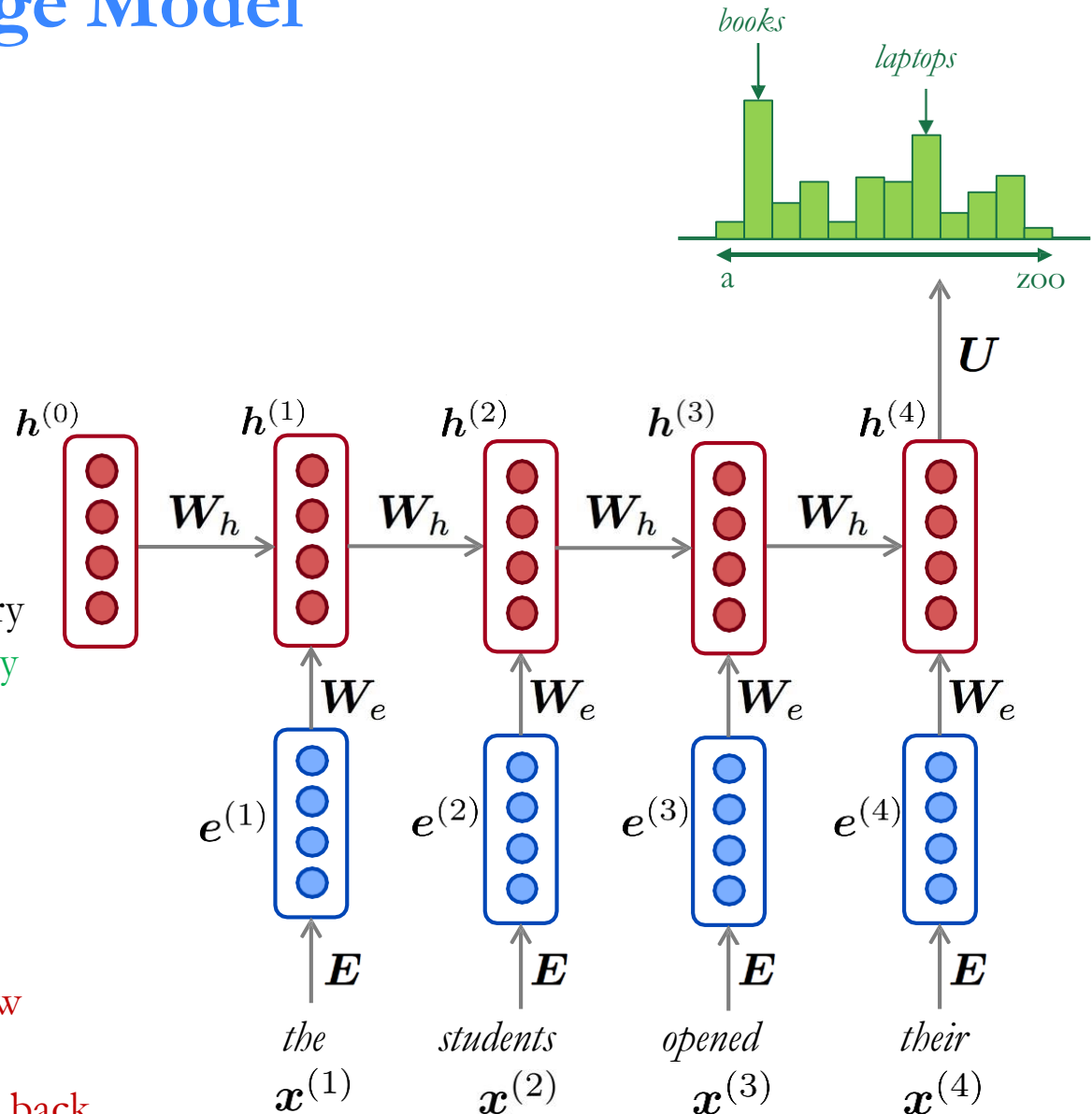
- Can process **any length** input
- Computation for step t can (in theory) use information from **many steps back**
- **Model size doesn't increase** for longer input
- Same weights applied on every timestep, so there is **symmetry** in how inputs are processed

RNN

Disadvantages:

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

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



Recall: Training a RNN Language Model

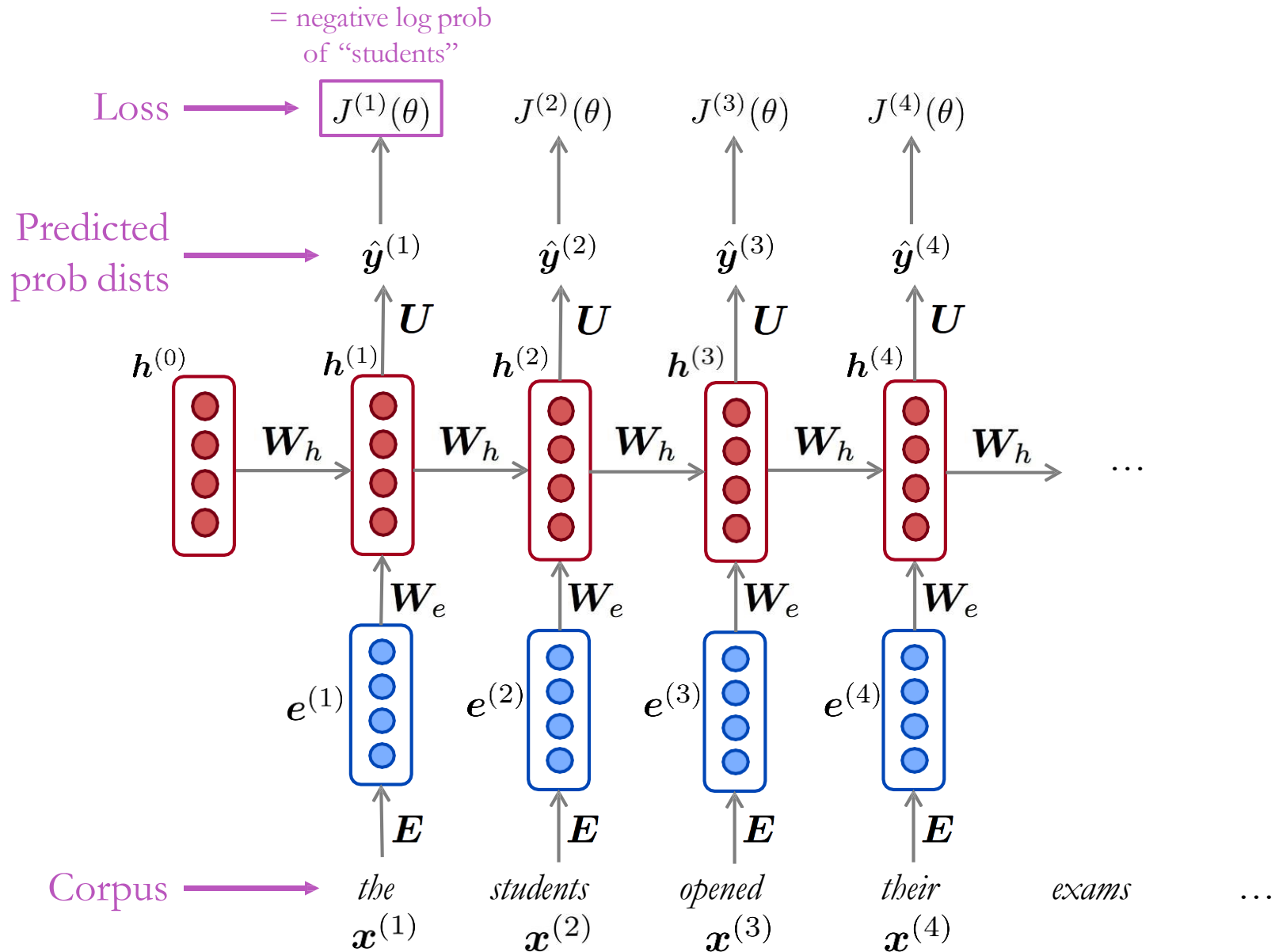
- Get a **big corpus of text** which is a sequence of words $x^{(1)}, \dots, x^{(T)}$
- Feed into RNN-LM; compute output distribution $\hat{y}^{(t)}$ **for every step t** .
 - i.e. predict probability distribution of *every word*, given words so far
- **Loss function** on step t is **cross-entropy** between predicted probability distribution $\hat{y}^{(t)}$, and the true next word $y^{(t)}$ (one-hot for $x^{(t+1)}$):

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

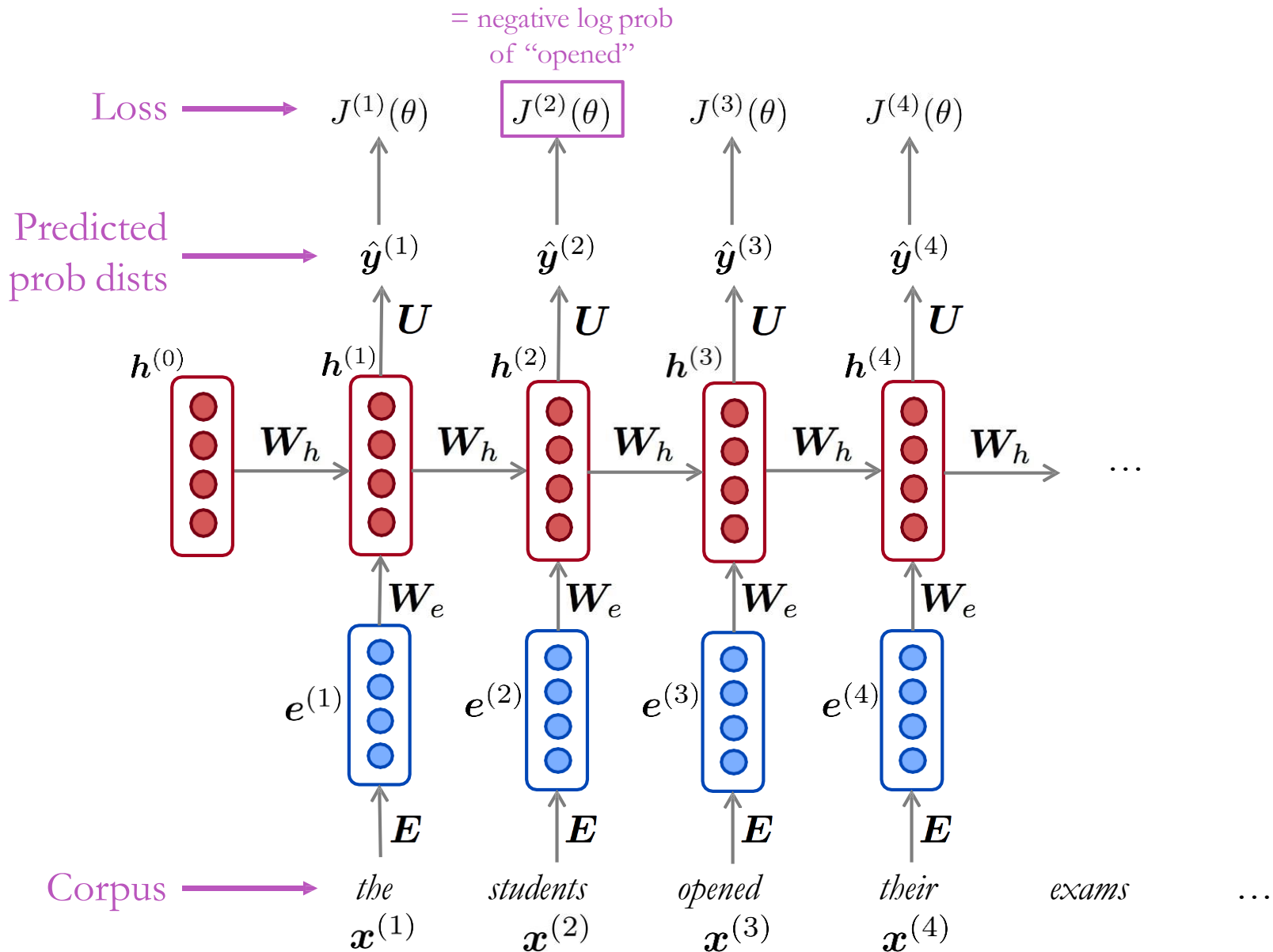
- Average this to get **overall loss** for entire training set:

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

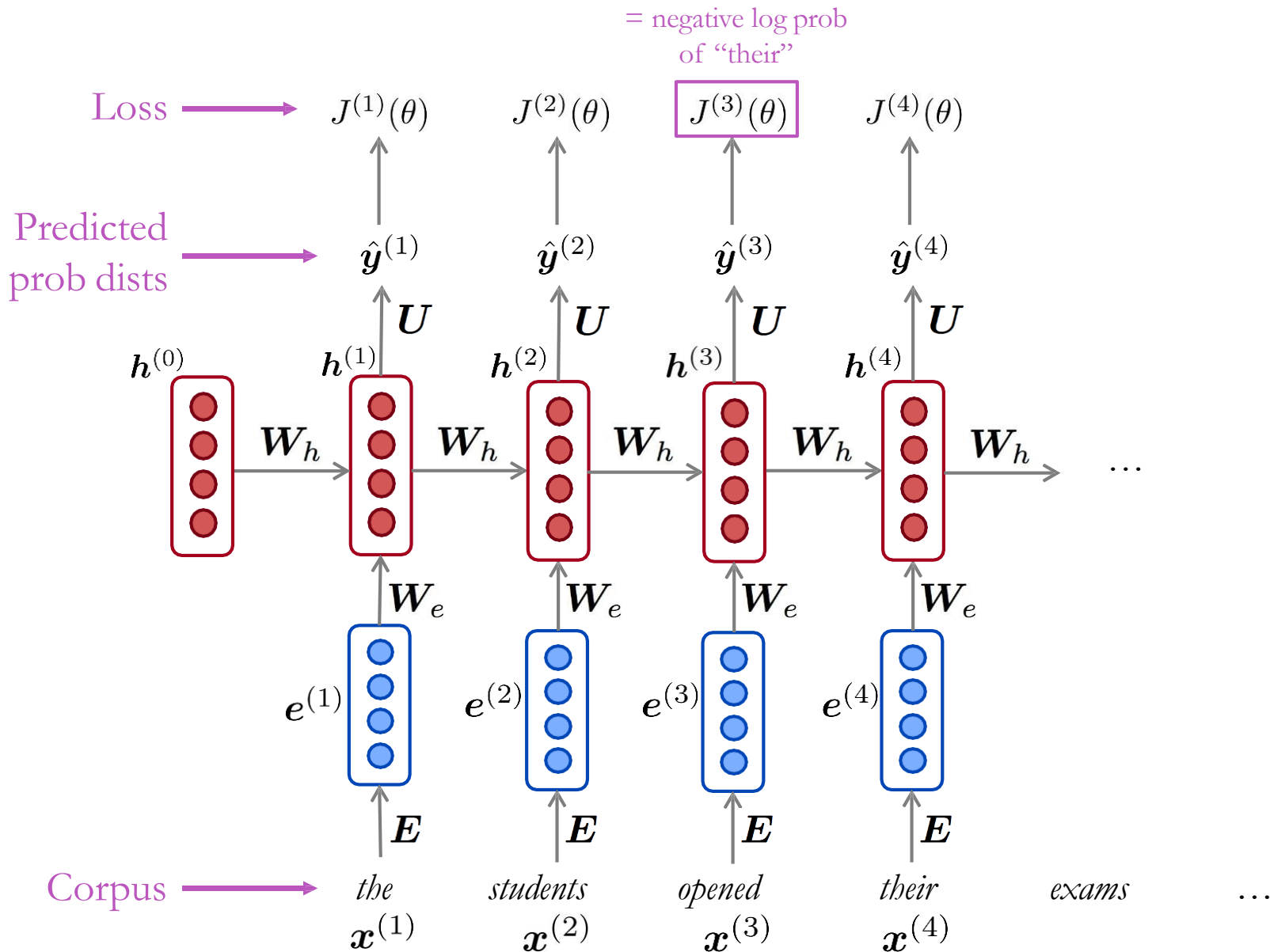
Training a RNN Language Model



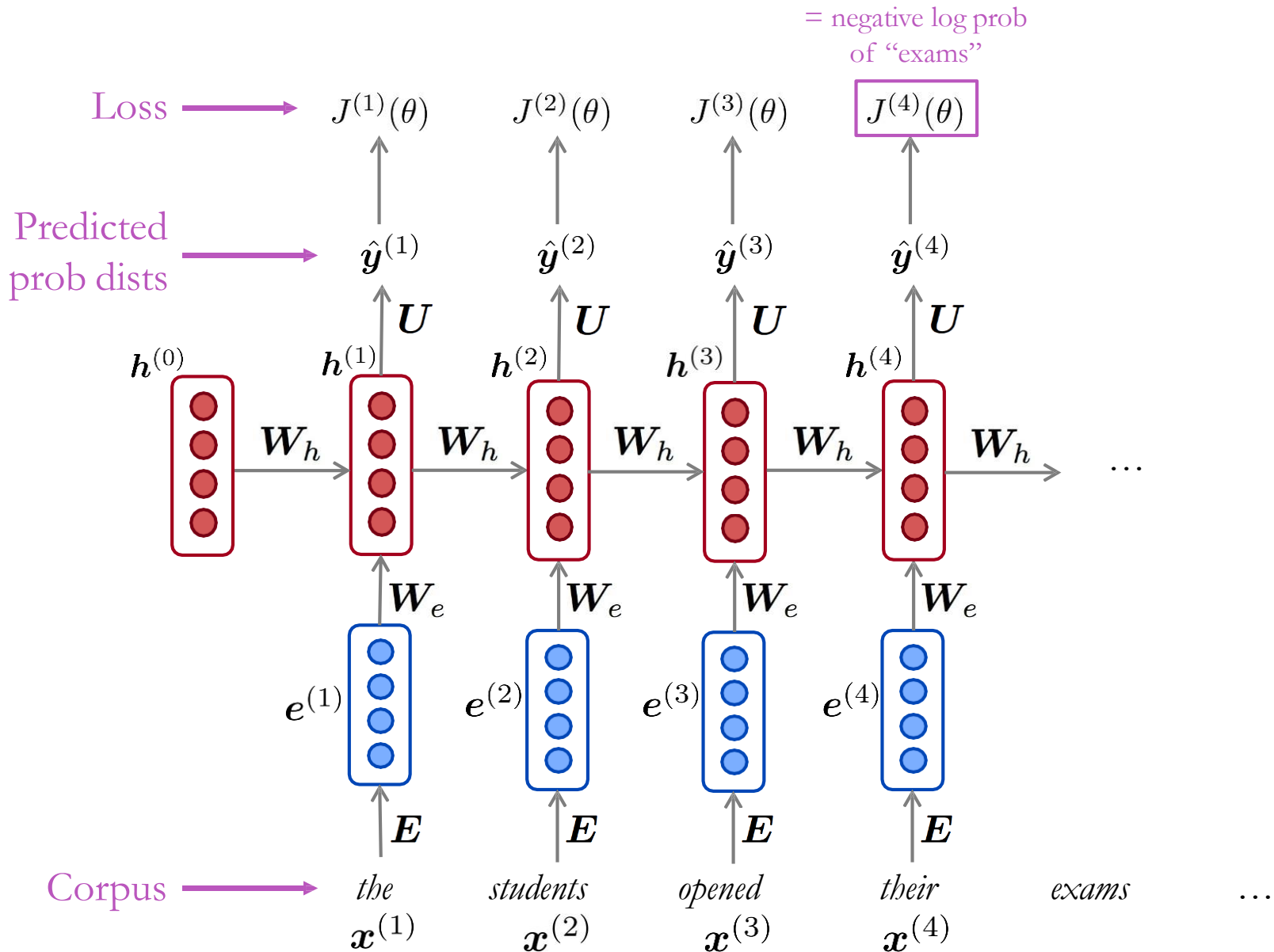
Training a RNN Language Model



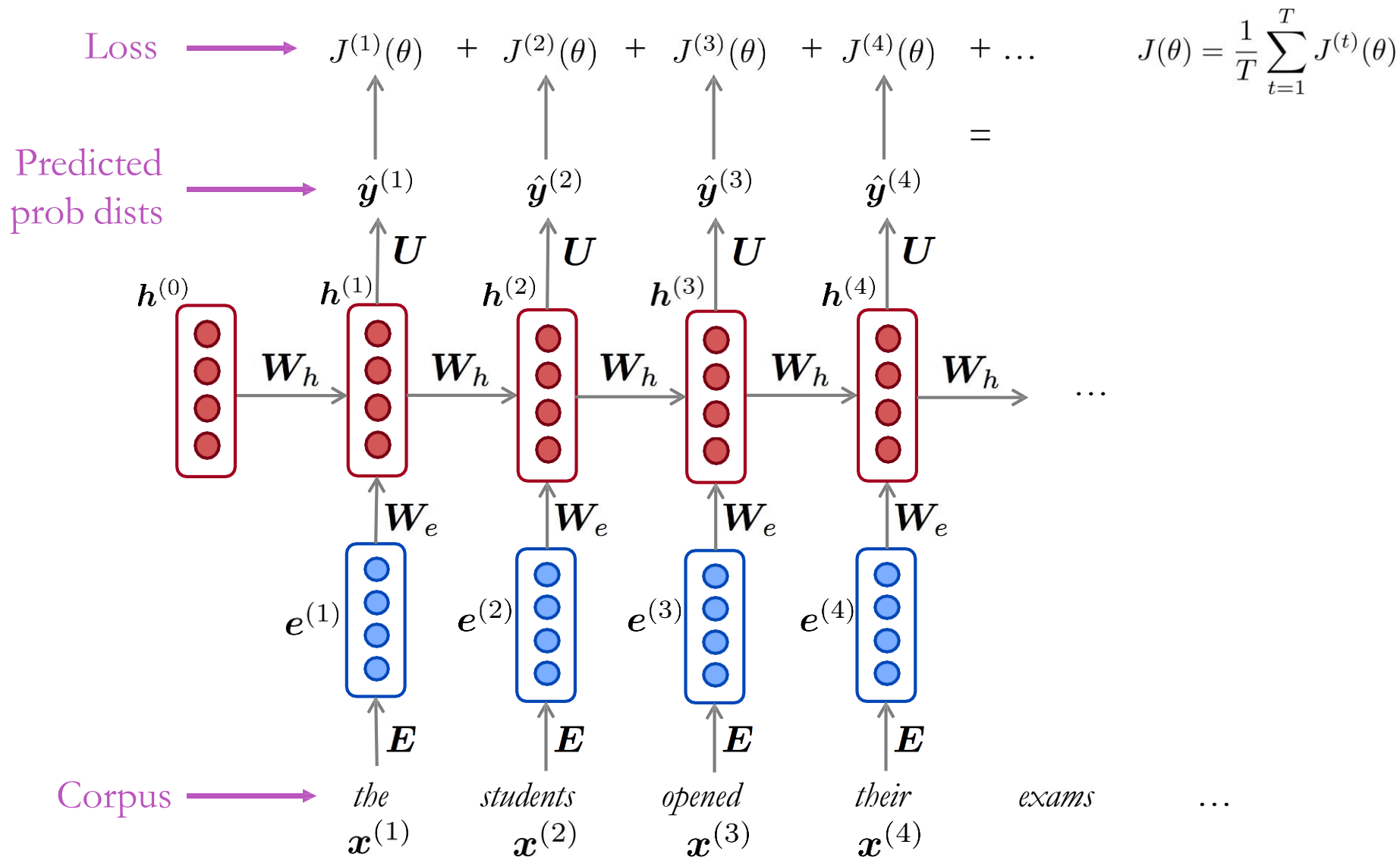
Training a RNN Language Model



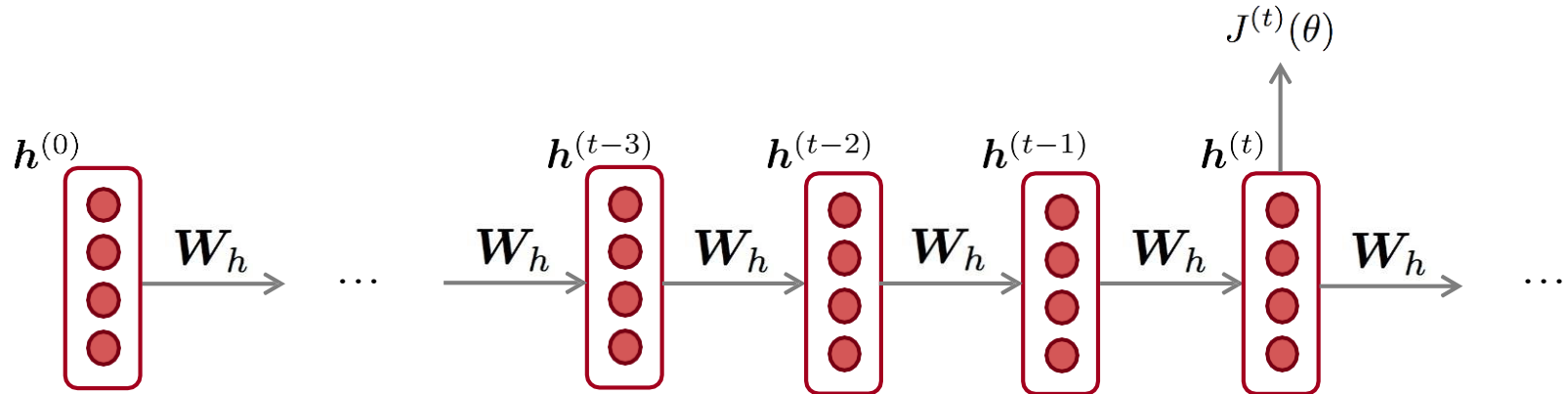
Training a RNN Language Model



Training a RNN Language Model



Backpropagation for RNNs



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

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

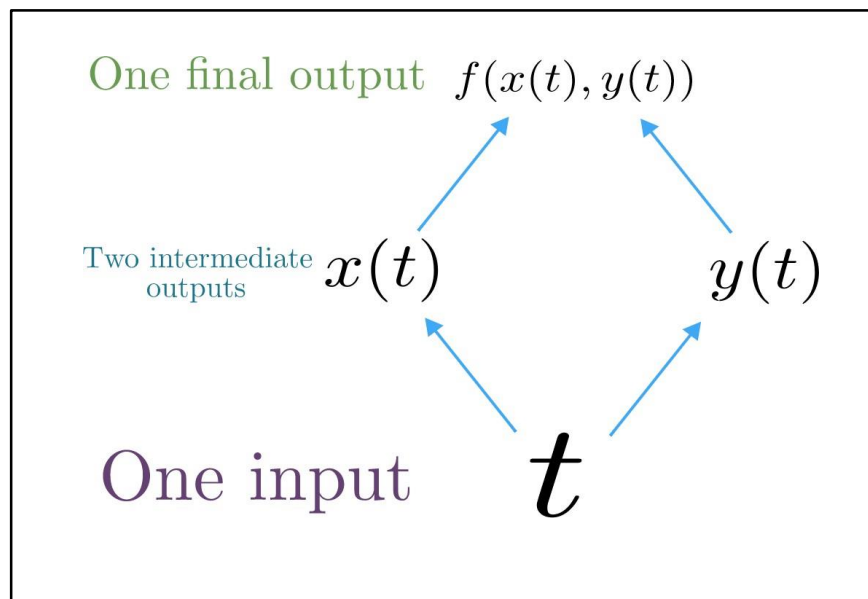
“The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears”

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:

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

Derivative of composition function



Source:

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

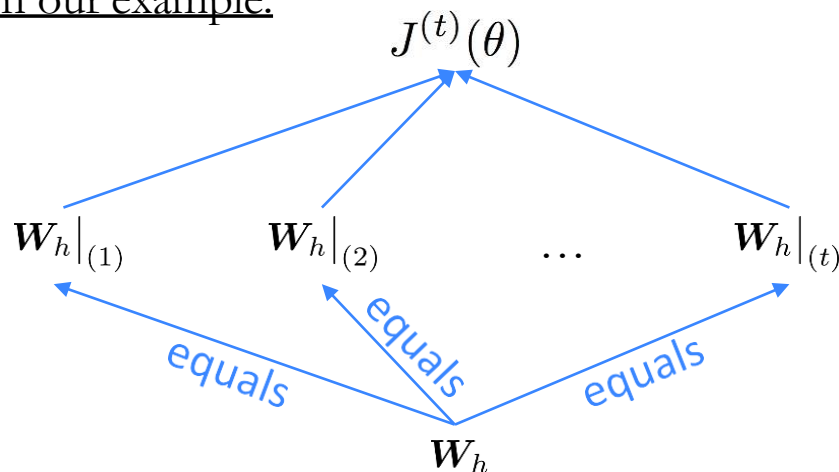
Backpropagation for RNNs: Proof sketch

- 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:

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

Derivative of composition function

In our example:



Apply the multivariable chain rule:

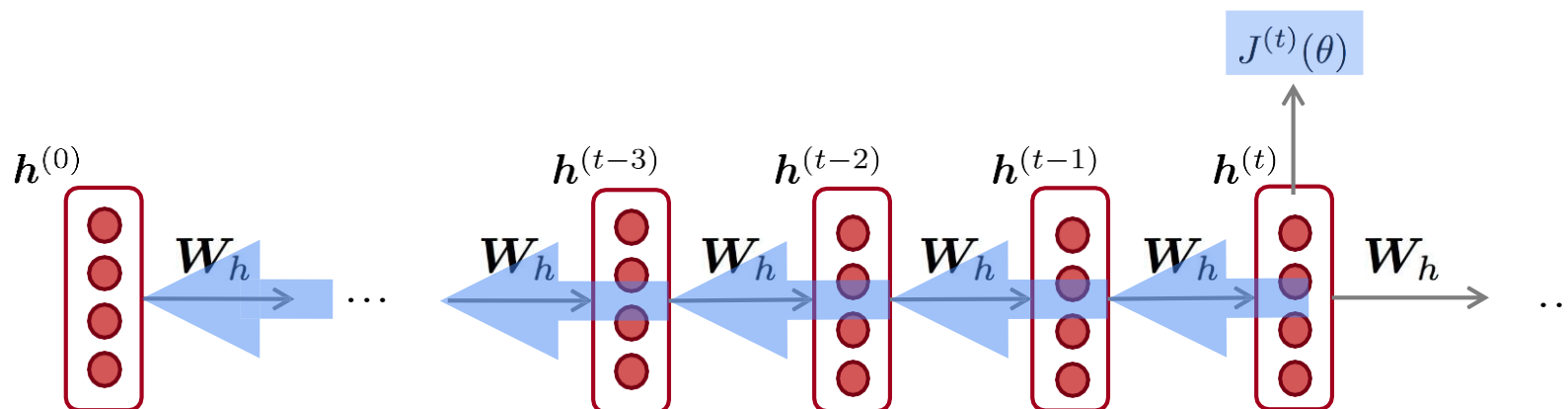
$= 1$

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

Source:

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

Backpropagation for RNNs

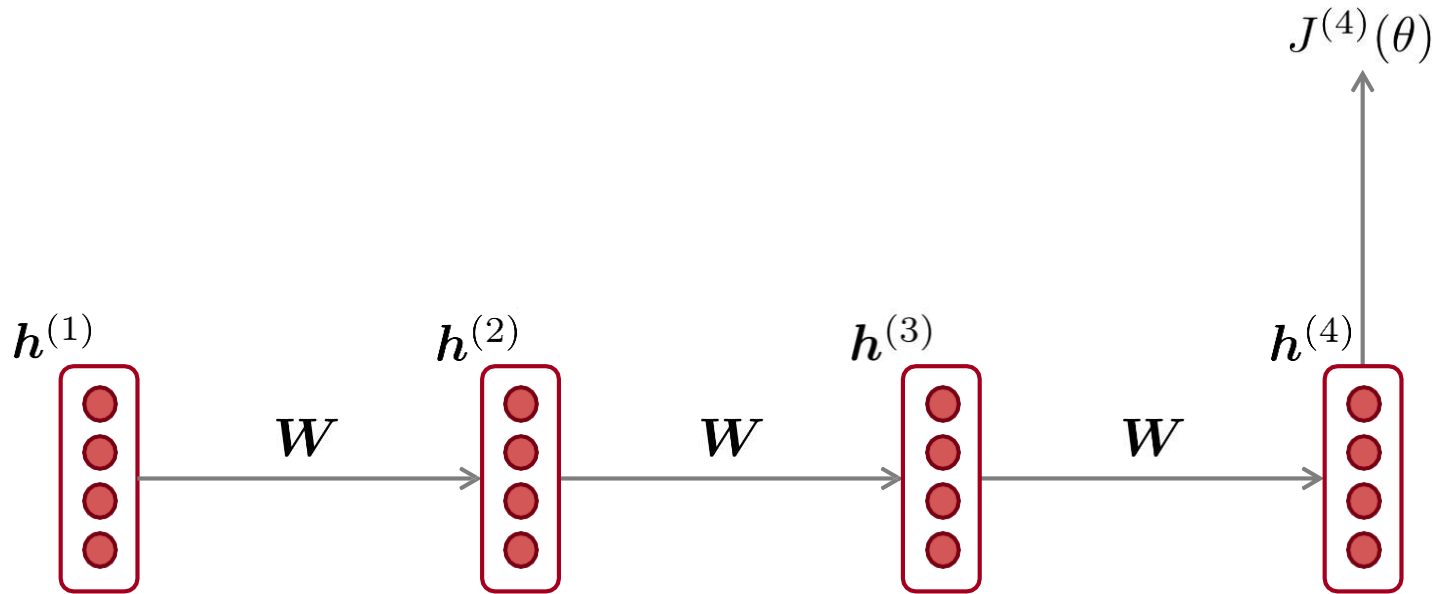


$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^t \left. \frac{\partial J^{(t)}}{\partial W_h} \right|_{(i)}$$

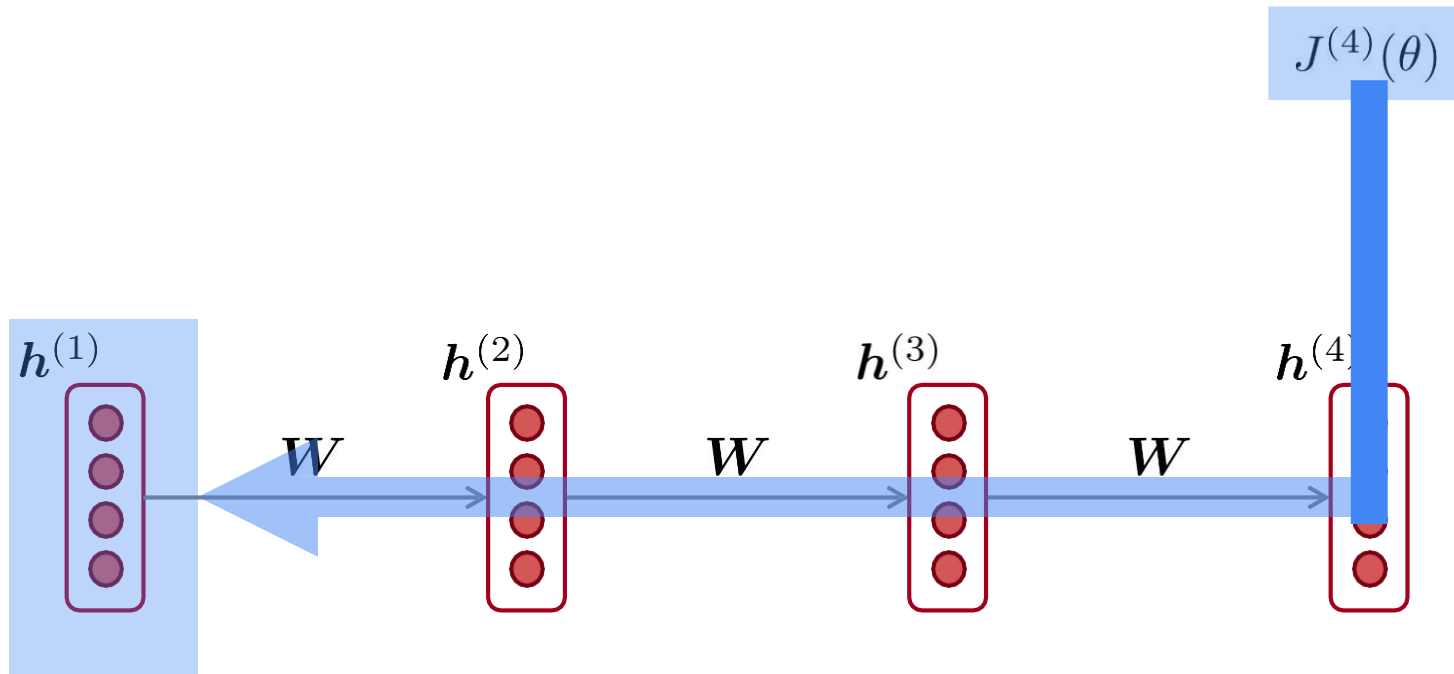
Question: How do we calculate this?

Answer: Backpropagate over timesteps $i=t, \dots, 0$, summing gradients as you go. This algorithm is called “backpropagation through time”

Vanishing gradient intuition

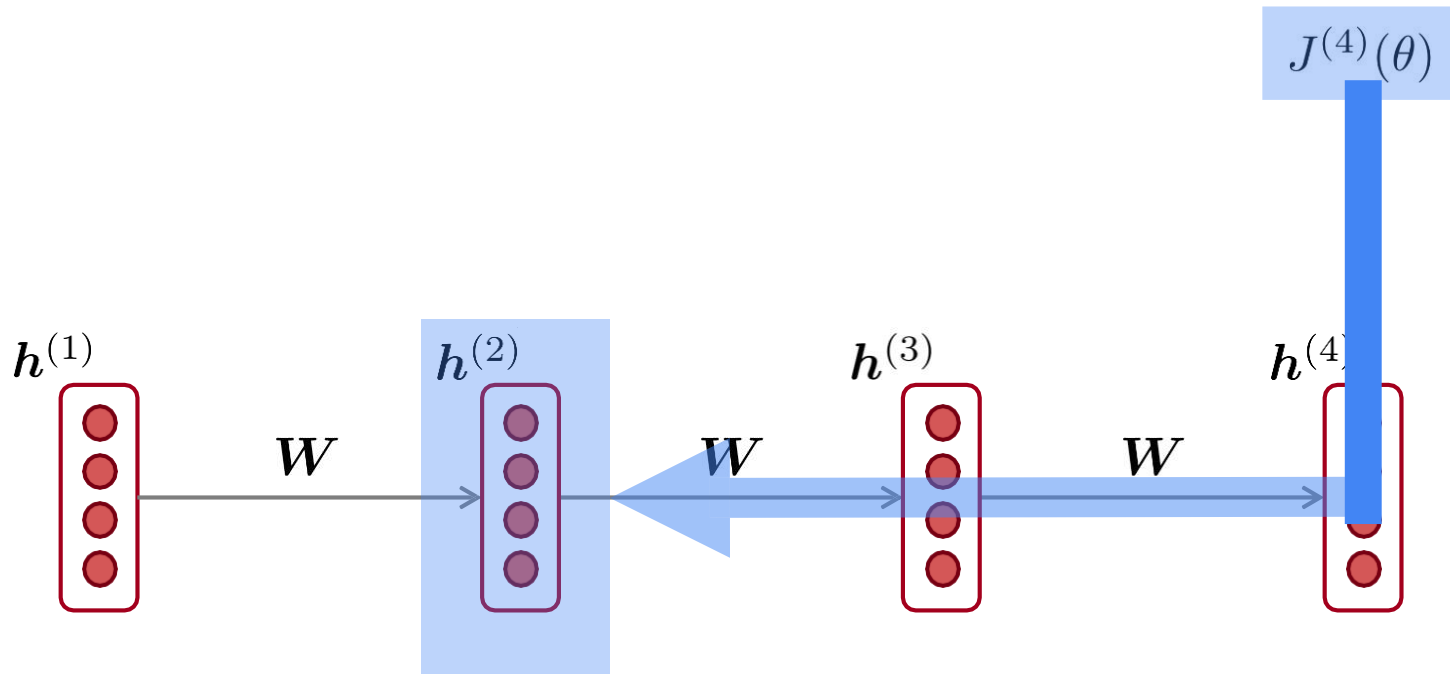


Vanishing gradient intuition



$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = ?$$

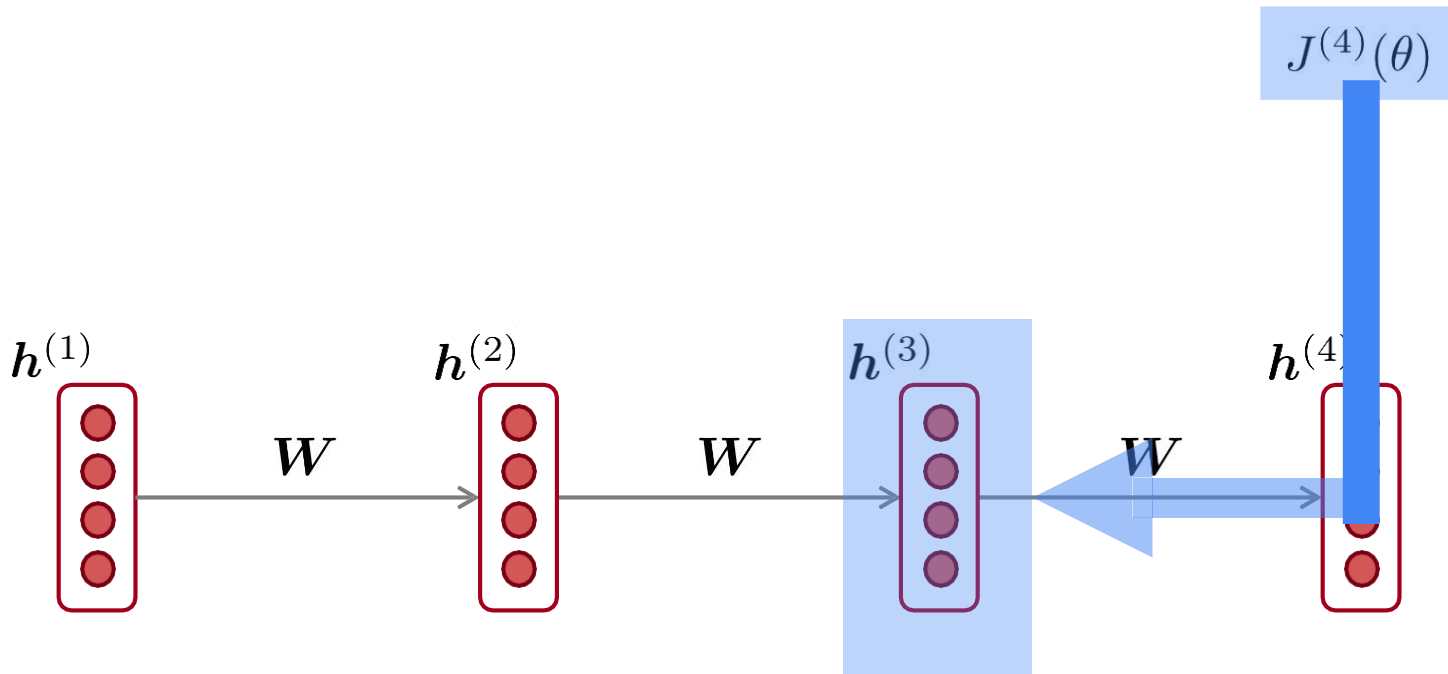
Vanishing gradient intuition



$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial J^{(4)}}{\partial h^{(2)}}$$

chain rule!

Vanishing gradient intuition

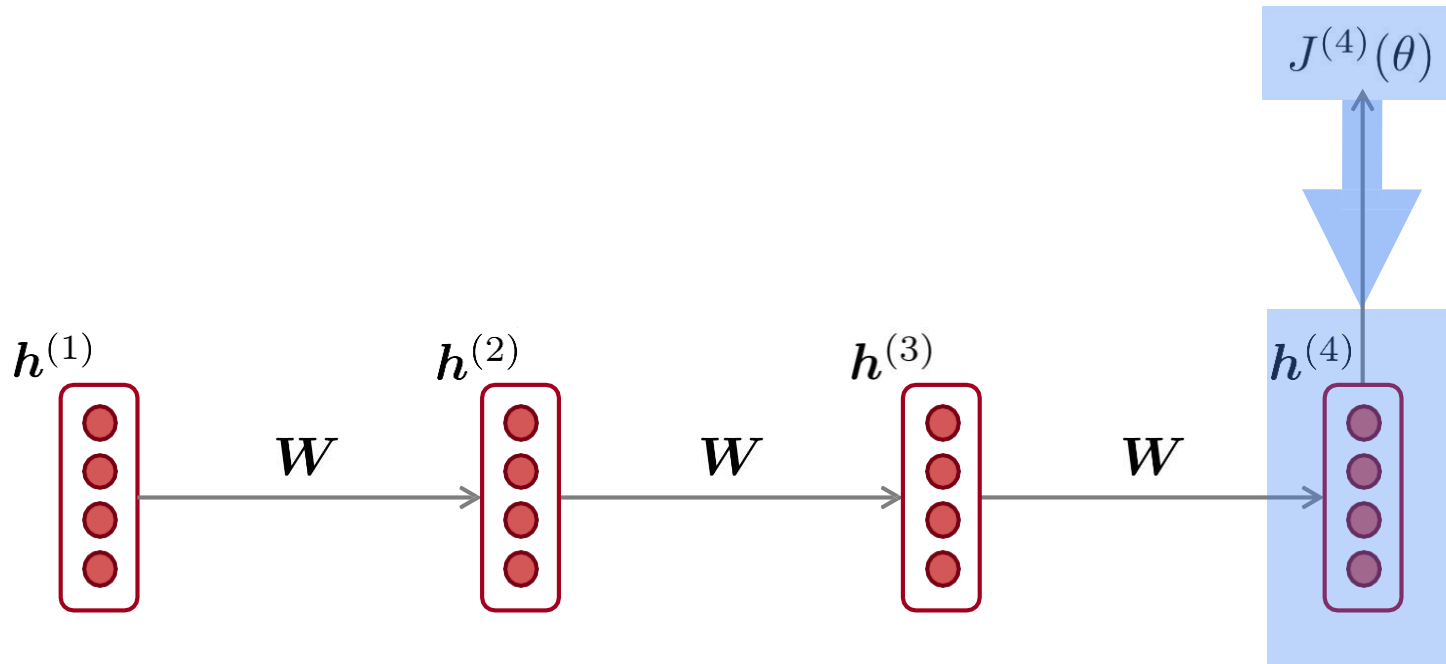


$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times$$

$$\frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial J^{(4)}}{\partial h^{(3)}}$$

chain rule!

Vanishing gradient intuition



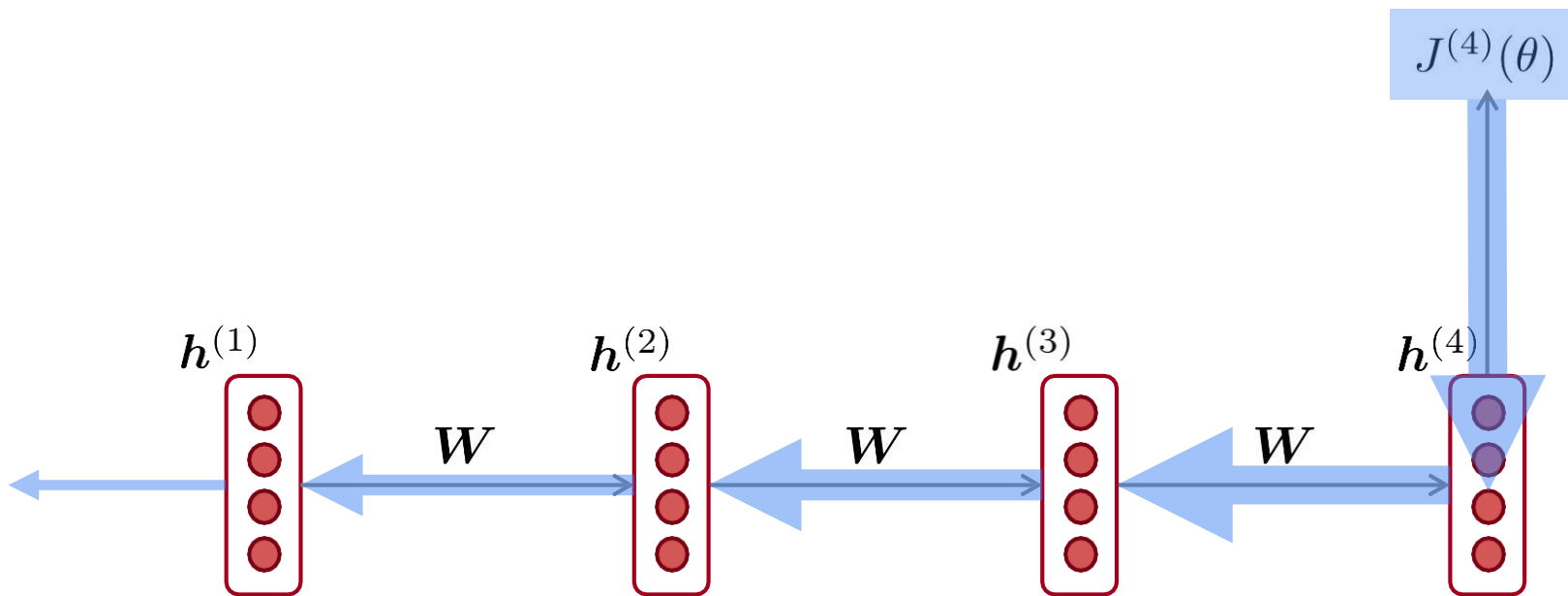
$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times$$

$$\frac{\partial h^{(3)}}{\partial h^{(2)}} \times$$

$$\frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}$$

chain rule!

Vanishing gradient intuition

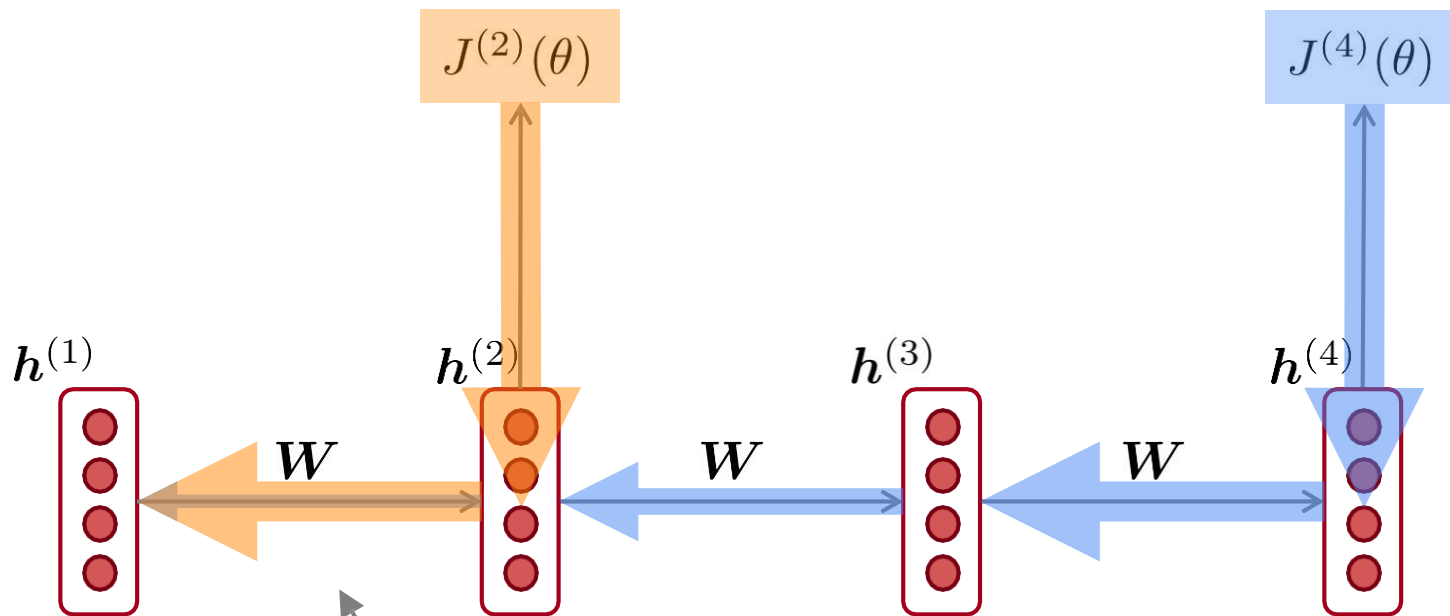


$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}$$

What happens if these are small?

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

Why is vanishing gradient a problem?



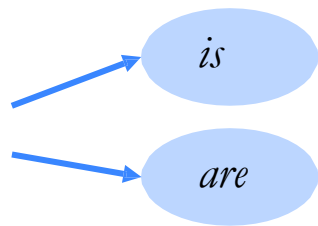


Gradient signal from faraway is lost because it's much smaller than gradient signal from close-by.

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

Effect of vanishing gradient on RNN-LM

- **LM task:** *When she tried to print her _____, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her tickets.*
- To learn from this training example, the RNN-LM needs to **model the dependency** between “*tickets*” on the 7th step and the target word “*tickets*” at the end.
- But if gradient is small, the model **can’t learn this dependency**
 - So the model is **unable to predict similar long-distance dependencies** at test time

Effect of vanishing gradient on RNN-LM

- **LM task:** *The writer of the books _____*
- **Correct answer:** *The writer of the books is planning a sequel*
- **Syntactic recency:** *The writer of the books is* (correct)
- **Sequential recency:** *The writer of the books are* (incorrect)
- Due to vanishing gradient, RNN-LMs are better at learning from sequential recency than syntactic recency, so they make this type of error more often than we'd like [Linzen et al 2016]

Why is exploding gradient a problem?

- If the gradient becomes too big, then the SGD update step becomes too big:

$$\theta^{new} = \theta^{old} - \overbrace{\alpha}^{\text{learning rate}} \underbrace{\nabla_{\theta} J(\theta)}_{\text{gradient}}$$

- This can cause **bad updates**: we take too large a step and reach a bad parameter configuration (with large loss)
- In the worst case, this will result in **Inf** or **NaN** in your network (then you have to restart training from an earlier checkpoint)

Gradient clipping: solution for exploding gradient

- Gradient clipping: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update

Algorithm 1 Pseudo-code for norm clipping

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

- Intuition: take a step in the same direction, but a smaller step

RNNs with Gates

How to fix vanishing gradient problem?

- The main problem is that *it's too difficult for the RNN to learn to preserve information over many timesteps.*
- In a vanilla RNN, the hidden state is constantly being rewritten

$$\mathbf{h}^{(t)} = \sigma \left(\mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} \right)$$

- How about a RNN with separate memory?

Gated Recurrent Units (GRUs)

- More complex hidden unit computation in recurrence!
- Introduced by Cho et al. 2014
- Main ideas:
 - keep around memories to capture long distance dependencies
 - allow error messages to flow at different strengths depending on the inputs

Gated Recurrent Units (GRUs)

- Standard RNN computes hidden layer at next time step directly:
$$h_t = f \left(W^{(hh)} h_{t-1} + W^{(hx)} x_t \right)$$
- GRU first computes an update **gate** (another layer) based on current input word vector and hidden state

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

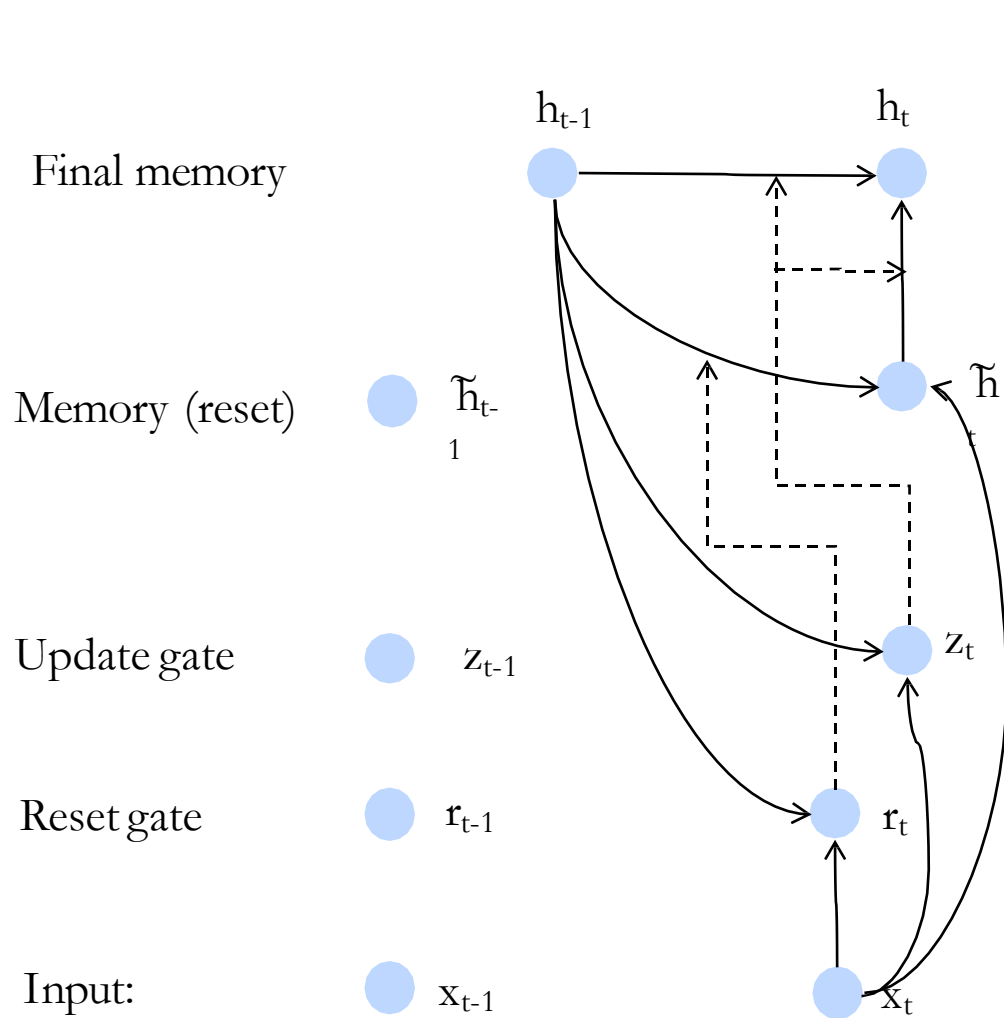
- Compute reset gate similarly but with different weights

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

Gated Recurrent Units (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: $\tilde{h}_t = \tanh (W x_t + r_t \circ U h_{t-1})$
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: $h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$

Gated Recurrent Units (GRUs)



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

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

$$\tilde{h}_t = \tanh \left(W x_t + r_t \circ U h_{t-1} \right)$$

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

Gated Recurrent Units (GRUs)

- If reset r is close to 0, ignore previous hidden state: Allows model to drop information that is irrelevant in the future
$$z_t = \sigma \left(W^{(z)} x_t + U^{(z)} h_{t-1} \right)$$
$$r_t = \sigma \left(W^{(r)} x_t + U^{(r)} h_{t-1} \right)$$
$$\tilde{h}_t = \tanh (W x_t + r_t \circ U h_{t-1})$$
$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$
- If update z is close to 1, can copy information through many time steps, i.e. copy-paste state: Less vanishing gradient!
- Units with short-term dependencies often have reset gates (r) very active; ones with long-term dependencies have active update gates (z)

Long-short-term-memories (LSTMs)

- Proposed by Hochreiter and Schmidhuber in 1997
- We can make the units even more complex
- Allow each time step to modify

- Input gate (current cell matters)
- Forget (gate 0, forget past)
- Output (how much cell is exposed)
- New memory cell

- Final memory cell:

- Final hidden state:

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

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

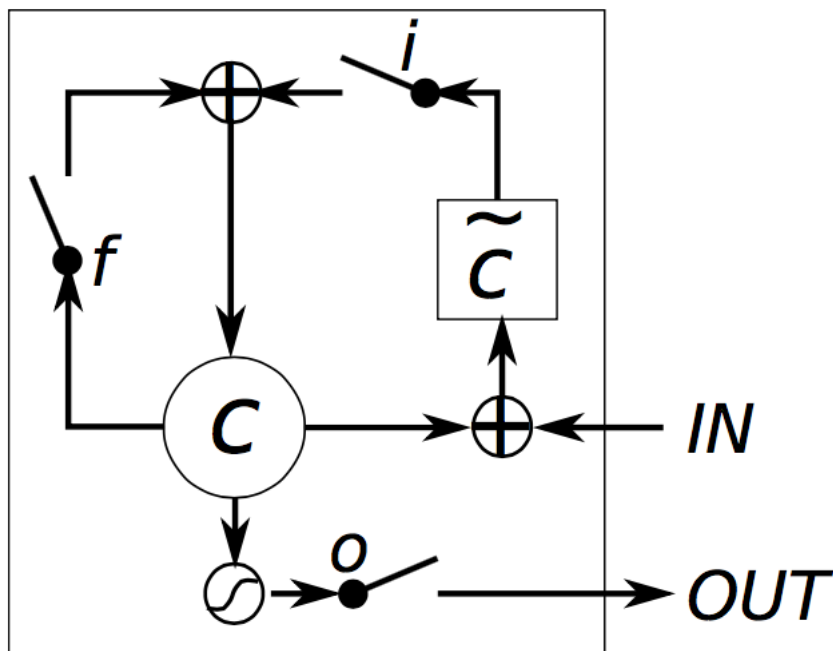
$$o_t = \sigma \left(W^{(o)} x_t + U^{(o)} h_{t-1} \right)$$

$$\tilde{c}_t = \tanh \left(W^{(c)} x_t + U^{(c)} h_{t-1} \right)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

$$h_t = o_t \circ \tanh(c_t)$$

Long-short-term-memories (LSTMs)



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

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

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

$$\tilde{c}_t = \tanh \left(W^{(c)} x_t + U^{(c)} h_{t-1} \right)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

$$h_t = o_t \circ \tanh(c_t)$$

Intuition: memory cells can keep information intact, unless inputs makes them forget it or overwrite it with new input

Cell can decide to output this information or just store it

Review on your own: Gated Recurrent Units (GRU)

- Proposed by Cho et al. in 2014 as a simpler alternative to the LSTM.
- On each timestep t we have input $\mathbf{x}^{(t)}$ and hidden state $\mathbf{h}^{(t)}$ (no cell state).

Update gate: controls what parts of hidden state are updated vs preserved

$$\mathbf{u}^{(t)} = \sigma \left(\mathbf{W}_u \mathbf{h}^{(t-1)} + \mathbf{U}_u \mathbf{x}^{(t)} + \mathbf{b}_u \right)$$

Reset gate: controls what parts of previous hidden state are used to compute new content

$$\mathbf{r}^{(t)} = \sigma \left(\mathbf{W}_r \mathbf{h}^{(t-1)} + \mathbf{U}_r \mathbf{x}^{(t)} + \mathbf{b}_r \right)$$

New hidden state content: reset gate selects useful parts of prev hidden state. Use this and current input to compute new hidden content.

$$\tilde{\mathbf{h}}^{(t)} = \tanh \left(\mathbf{W}_h (\mathbf{r}^{(t)} \circ \mathbf{h}^{(t-1)}) + \mathbf{U}_h \mathbf{x}^{(t)} + \mathbf{b}_h \right)$$

$$\mathbf{h}^{(t)} = (1 - \mathbf{u}^{(t)}) \circ \mathbf{h}^{(t-1)} + \mathbf{u}^{(t)} \circ \tilde{\mathbf{h}}^{(t)}$$

Hidden state: update gate simultaneously controls what is kept from previous hidden state, and what is updated to new hidden state content

How does this solve vanishing gradient?
GRU makes it easier to retain info long-term (e.g. by setting update gate to 0)

Review on your own: Long Short-Term Memory (LSTM)

We have a sequence of inputs $\mathbf{x}^{(t)}$, and we will compute a sequence of hidden states $\mathbf{h}^{(t)}$ and cell states $\mathbf{c}^{(t)}$. On timestep t :

Forget gate: controls what is kept vs forgotten, from previous cell state

Input gate: controls what parts of the new cell content are written to cell

Output gate: controls what parts of cell are output to hidden state

New cell content: this is the new content to be written to the cell

Cell state: erase (“forget”) some content from last cell state, and write (“input”) some new cell content

Hidden state: read (“output”) some content from the cell

Sigmoid function: all gate values are between 0 and 1

$$\mathbf{f}^{(t)} = \sigma \left(\mathbf{W}_f \mathbf{h}^{(t-1)} + \mathbf{U}_f \mathbf{x}^{(t)} + \mathbf{b}_f \right)$$

$$\mathbf{i}^{(t)} = \sigma \left(\mathbf{W}_i \mathbf{h}^{(t-1)} + \mathbf{U}_i \mathbf{x}^{(t)} + \mathbf{b}_i \right)$$

$$\mathbf{o}^{(t)} = \sigma \left(\mathbf{W}_o \mathbf{h}^{(t-1)} + \mathbf{U}_o \mathbf{x}^{(t)} + \mathbf{b}_o \right)$$

$$\tilde{\mathbf{c}}^{(t)} = \tanh \left(\mathbf{W}_c \mathbf{h}^{(t-1)} + \mathbf{U}_c \mathbf{x}^{(t)} + \mathbf{b}_c \right)$$

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \circ \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \circ \tilde{\mathbf{c}}^{(t)}$$

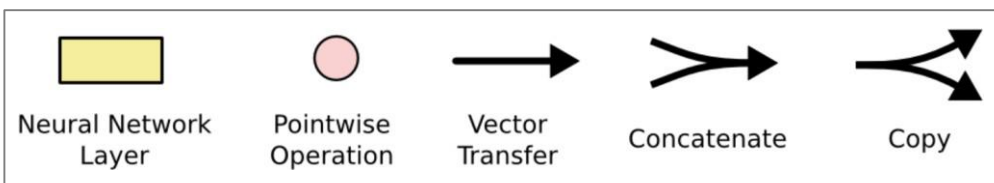
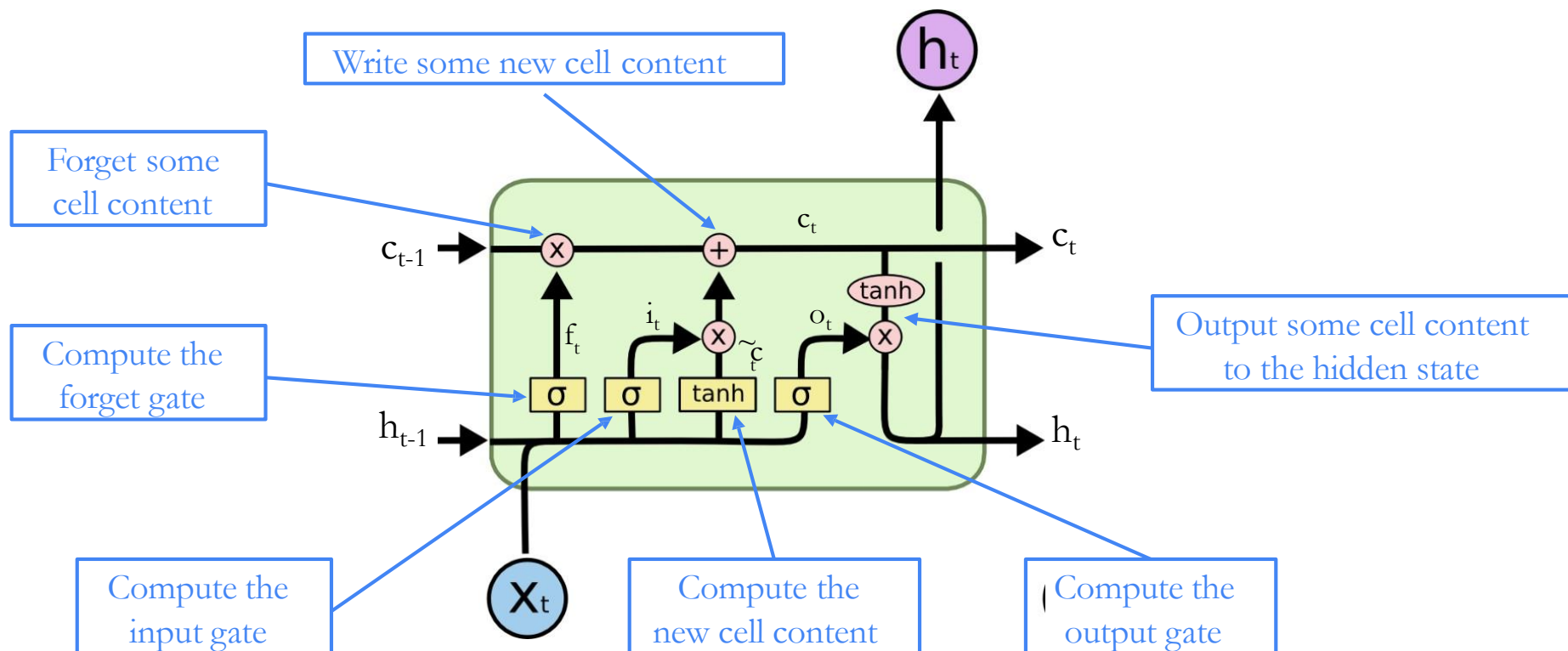
$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \circ \tanh \mathbf{c}^{(t)}$$

Gates are applied using element-wise product

All these are vectors of same length n

Review on your own: Long Short-Term Memory (LSTM)

You can think of the LSTM equations visually like this:



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

LSTM vs GRU

- Researchers have proposed many gated RNN variants, but LSTM and GRU are the most widely-used
- The biggest difference is that GRU is quicker to compute and has fewer parameters
- There is no conclusive evidence that one consistently performs better than the other
- LSTM is a good default choice (especially if your data has particularly long dependencies, or you have lots of training data)
- Rule of thumb: start with LSTM, but switch to GRU if you want something more efficient