

Natural Language Processing with Deep Learning

CS224N/Ling284



Lecture 7:
Vanishing Gradients
and Fancy RNNs

Abigail See

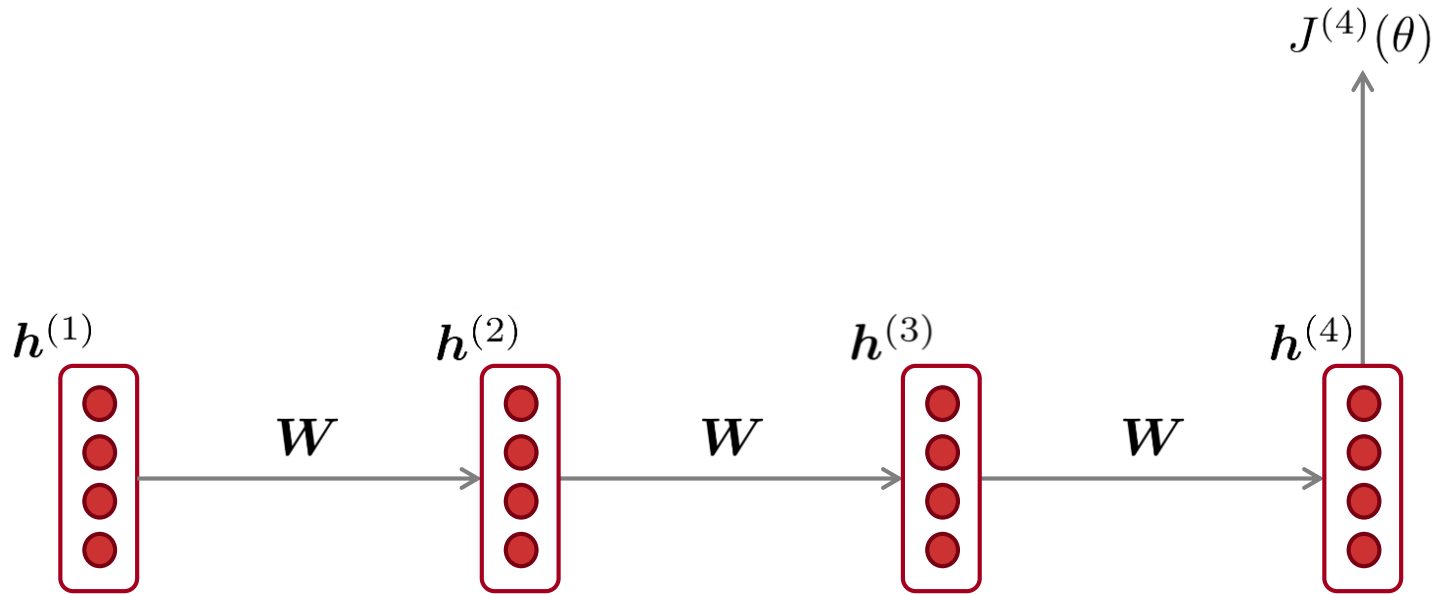
Overview

- Last lecture we learned:
 - Recurrent Neural Networks (RNNs) and why they're great for Language Modeling (LM).
- Today we'll learn:
 - Problems with RNNs and how to fix them
 - More complex RNN variants
- Next lecture we'll learn:
 - How we can do Neural Machine Translation (NMT) using an RNN-based architecture called sequence-to-sequence with attention

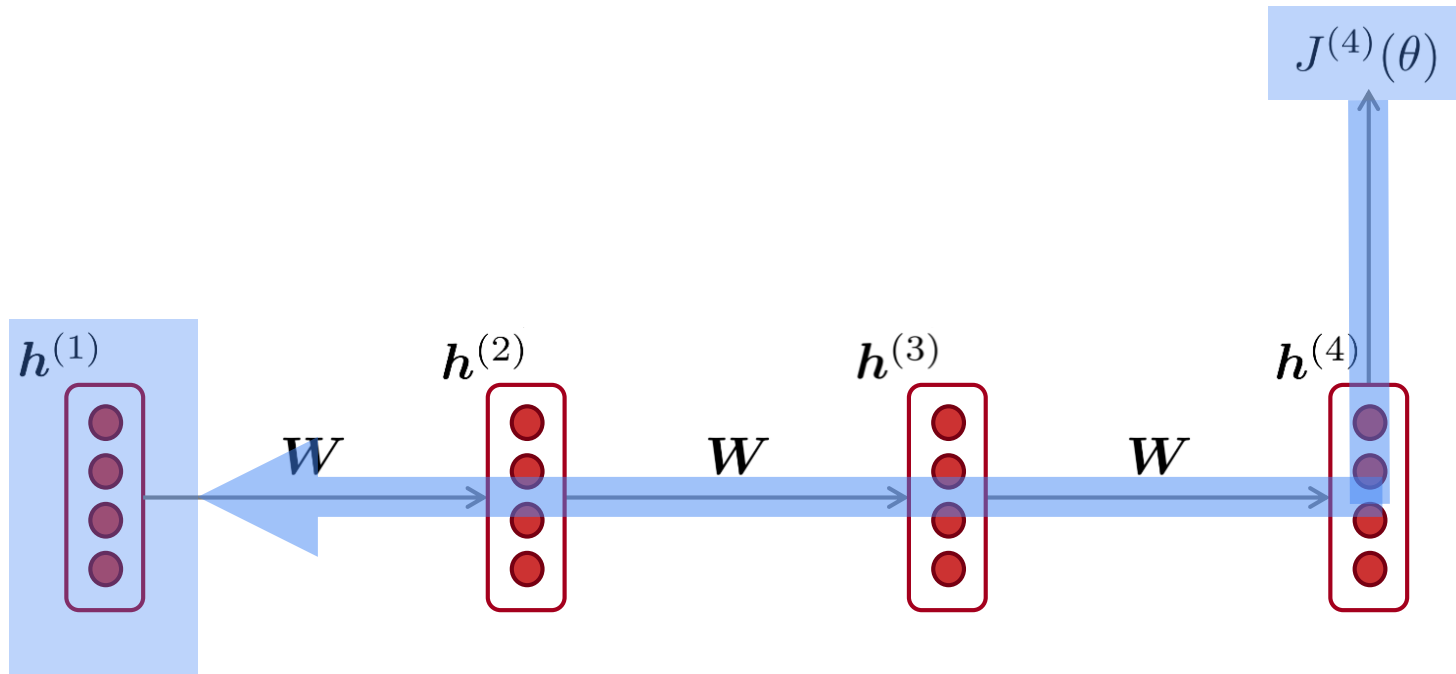
Today's lecture

- Vanishing gradient problem
 - Two new types of RNN: LSTM and GRU
 - Other fixes for vanishing (or exploding) gradient:
 - Gradient clipping
 - Skip connections
 - More fancy RNN variants:
 - Bidirectional RNNs
 - Multi-layer RNNs
- motivates
- Lots of important definitions today!

Vanishing gradient intuition

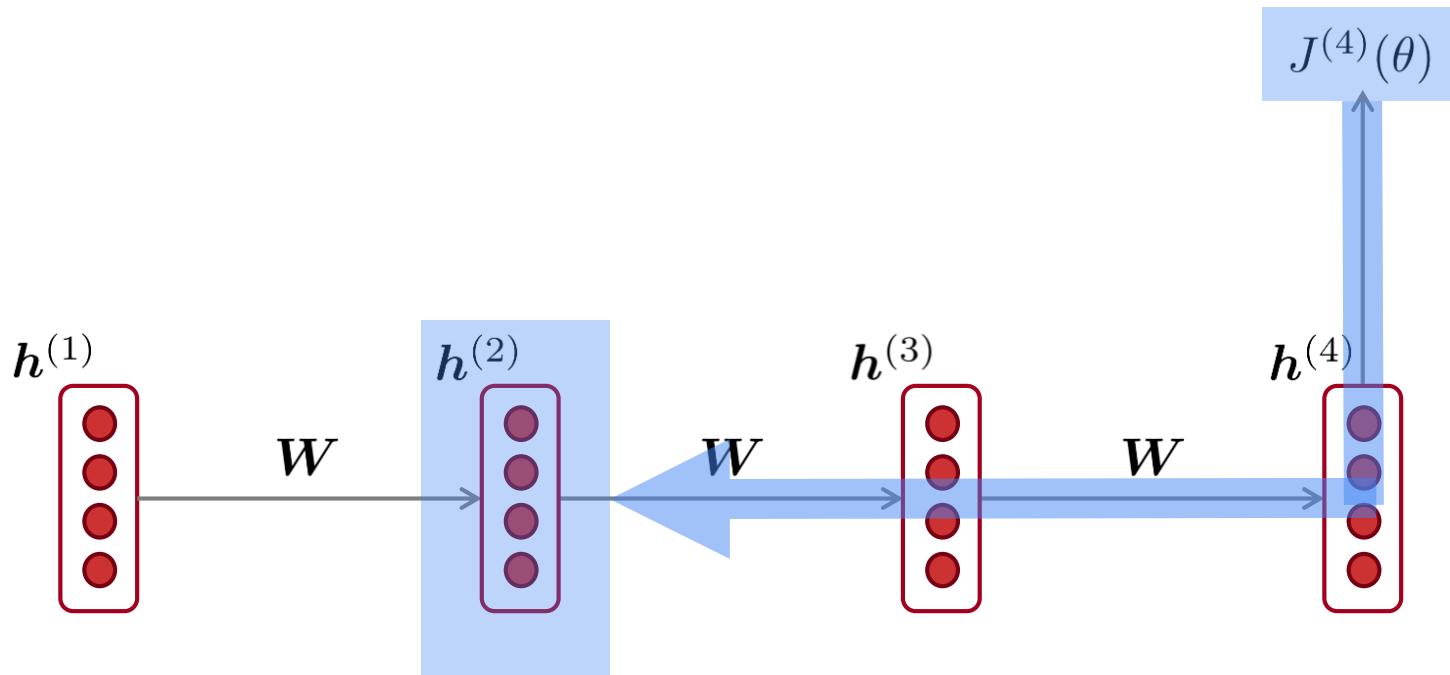


Vanishing gradient intuition



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

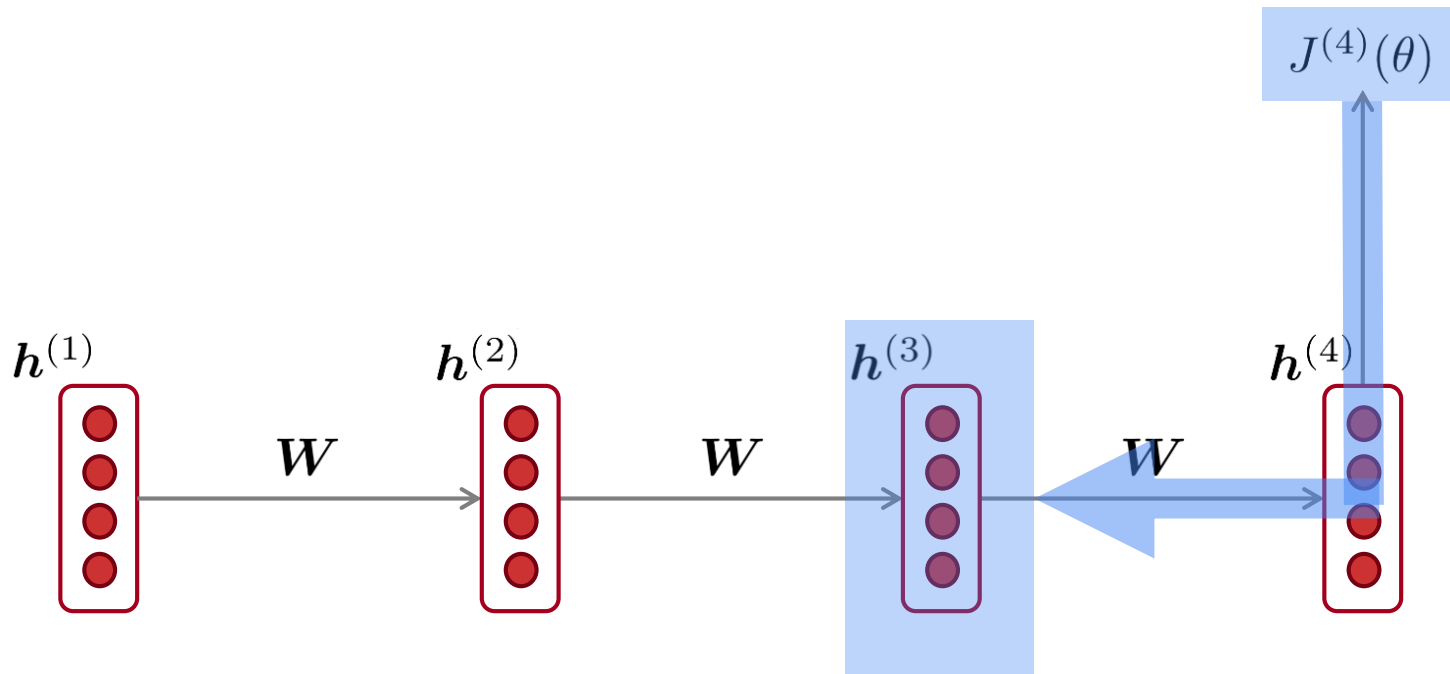
Vanishing gradient intuition



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

chain rule!

Vanishing gradient intuition

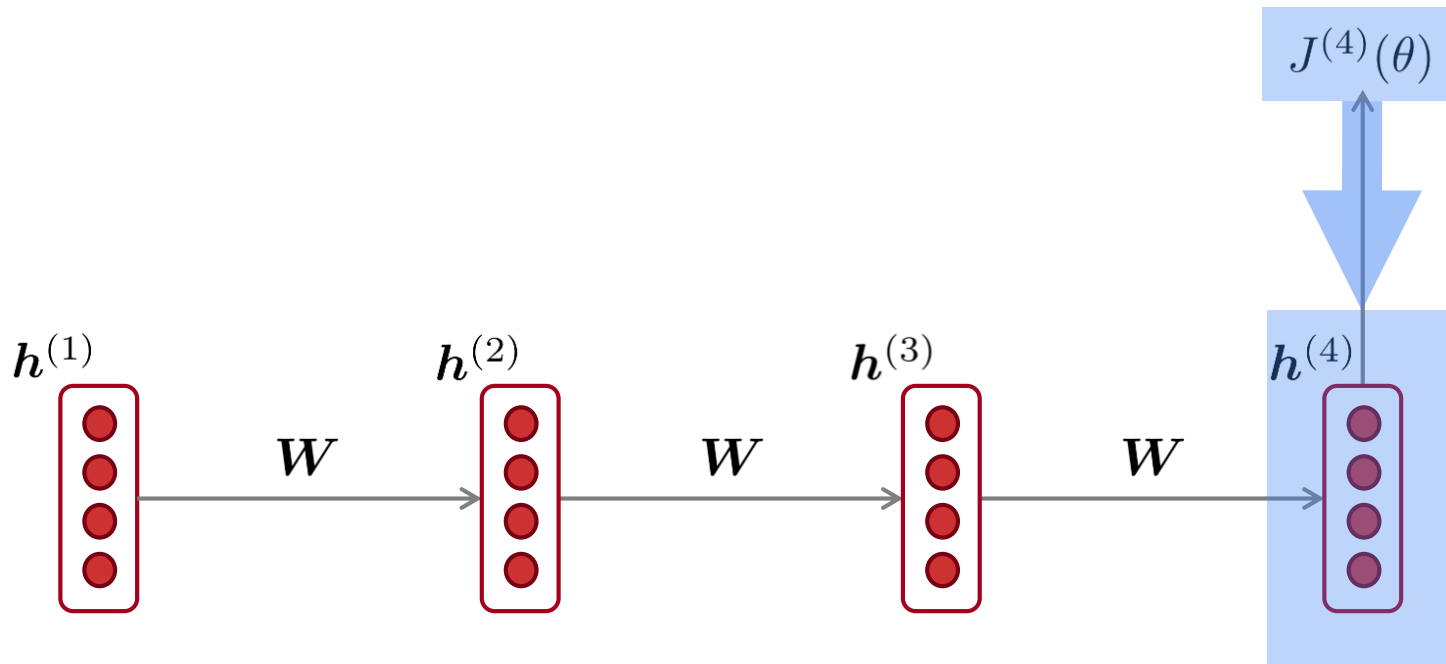


$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times$$

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

chain rule!

Vanishing gradient intuition



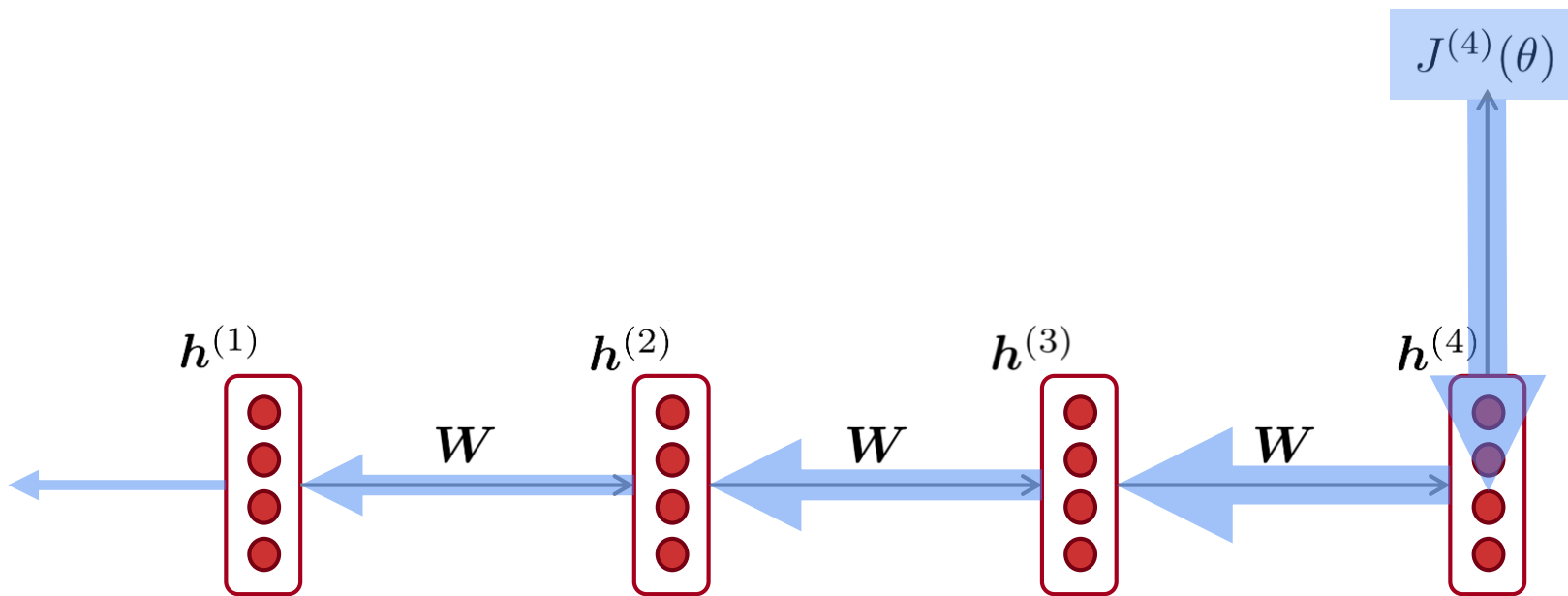
$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times$$

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

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

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

Vanishing gradient proof sketch

- Recall: $\mathbf{h}^{(t)} = \sigma \left(\mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} + \mathbf{b}_1 \right)$
- Therefore: $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} = \text{diag} \left(\sigma' \left(\mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} + \mathbf{b}_1 \right) \right) \mathbf{W}_h$ (chain rule)
- Consider the gradient of the loss $J^{(i)}(\theta)$ on step i , with respect to the hidden state $\mathbf{h}^{(j)}$ on some previous step j .

$$\begin{aligned} \frac{\partial J^{(i)}(\theta)}{\partial \mathbf{h}^{(j)}} &= \frac{\partial J^{(i)}(\theta)}{\partial \mathbf{h}^{(i)}} \prod_{j < t \leq i} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} && \text{(chain rule)} \\ &= \frac{\partial J^{(i)}(\theta)}{\partial \mathbf{h}^{(i)}} \boxed{\mathbf{W}_h^{(i-j)}} \prod_{j < t \leq i} \text{diag} \left(\sigma' \left(\mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} + \mathbf{b}_1 \right) \right) && \left(\text{value of } \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} \right) \end{aligned}$$


If \mathbf{W}_h is small, then this term gets vanishingly small as i and j get further apart

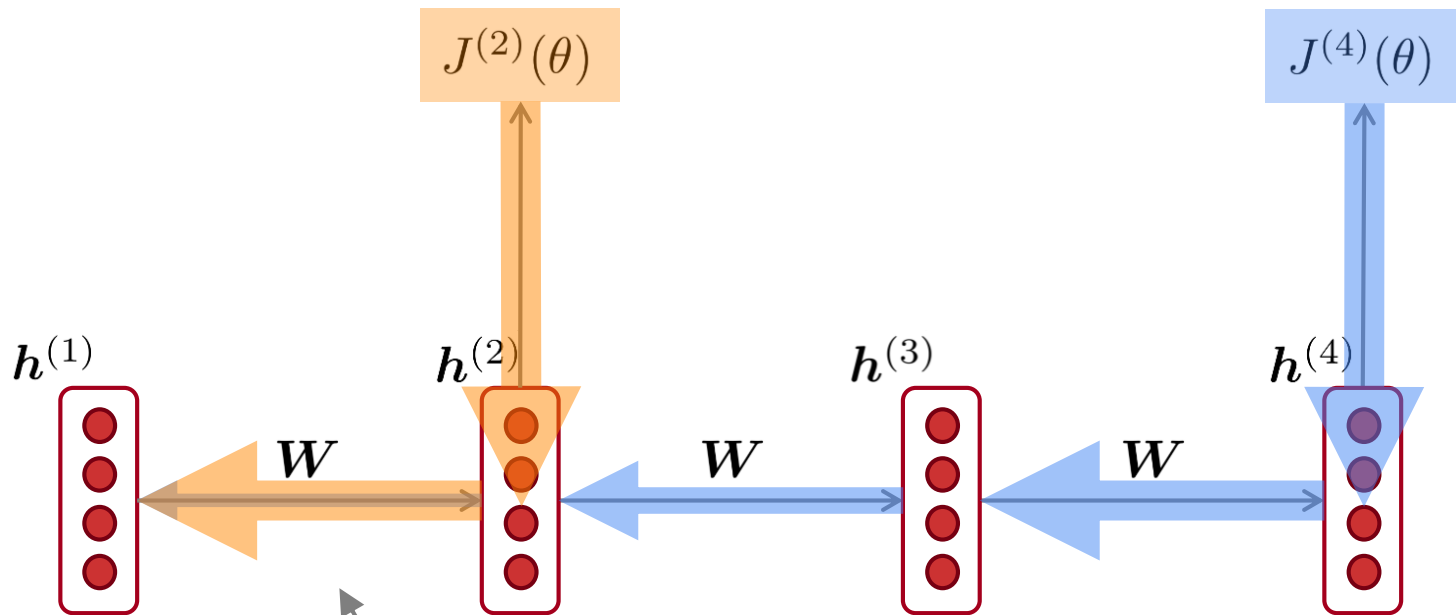
Vanishing gradient proof sketch

- Consider matrix L2 norms:

$$\left\| \frac{\partial J^{(i)}(\theta)}{\partial \mathbf{h}^{(j)}} \right\| \leq \left\| \frac{\partial J^{(i)}(\theta)}{\partial \mathbf{h}^{(i)}} \right\| \|\mathbf{W}_h\|^{(i-j)} \prod_{j < t \leq i} \left\| \text{diag} \left(\sigma' \left(\mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} + \mathbf{b}_1 \right) \right) \right\|$$

- Pascanu et al showed that that if the **largest eigenvalue** of \mathbf{W}_h is **less than 1**, then the gradient $\left\| \frac{\partial J^{(i)}(\theta)}{\partial \mathbf{h}^{(j)}} \right\|$ will **shrink** exponentially
 - Here the bound is 1 because we have sigmoid nonlinearity
- There's a similar proof relating a **largest eigenvalue >1** to **exploding gradients**

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.




Why is vanishing gradient a problem?

- Another explanation: Gradient can be viewed as a measure of *the effect of the past on the future*
- If the gradient becomes vanishingly small over longer distances (step t to step $t+n$), then we can't tell whether:
 1. There's **no dependency** between step t and $t+n$ in the data
 2. We have **wrong parameters** to capture the true dependency between t and $t+n$

Effect of vanishing gradient on RNN-LM

- **LM task:** *When she tried to print her tickets, 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 _____*
- 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:

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

- 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

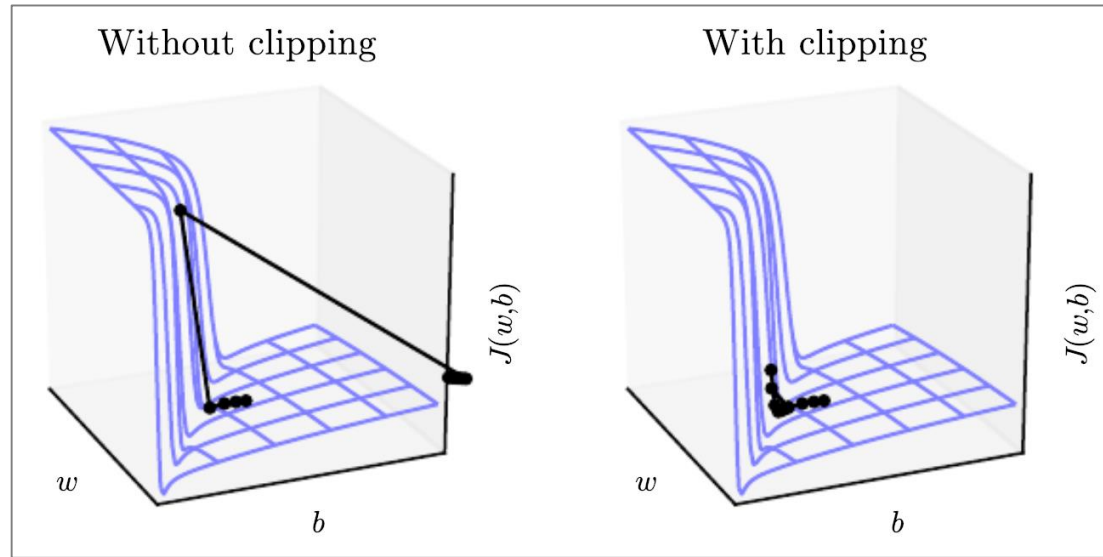
- Gradient clipping: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update

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

Gradient clipping: solution for exploding gradient



- This shows the loss surface of a simple RNN (hidden state is a scalar not a vector)
- The “cliff” is dangerous because it has steep gradient
- On the left, gradient descent takes two very big steps due to steep gradient, resulting in climbing the cliff then shooting off to the right (both bad updates)
- On the right, gradient clipping reduces the size of those steps, so effect is less drastic

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

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

- How about a RNN with separate memory?

Long Short-Term Memory (LSTM)

- A type of RNN proposed by Hochreiter and Schmidhuber in 1997 as a solution to the vanishing gradients problem.
- On step t , there is a **hidden state** $\mathbf{h}^{(t)}$ and a **cell state** $\mathbf{c}^{(t)}$
 - Both are vectors length n
 - The cell stores **long-term information**
 - The LSTM can **erase**, **write** and **read** information from the cell
- The selection of which information is erased/written/read is controlled by three corresponding **gates**
 - The gates are also vectors length n
 - On each timestep, each element of the gates can be **open** (1), **closed** (0), or somewhere in-between.
 - The gates are **dynamic**: their value is computed based on the current context

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

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

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

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

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

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

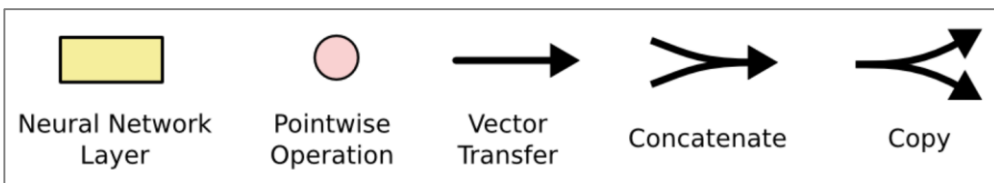
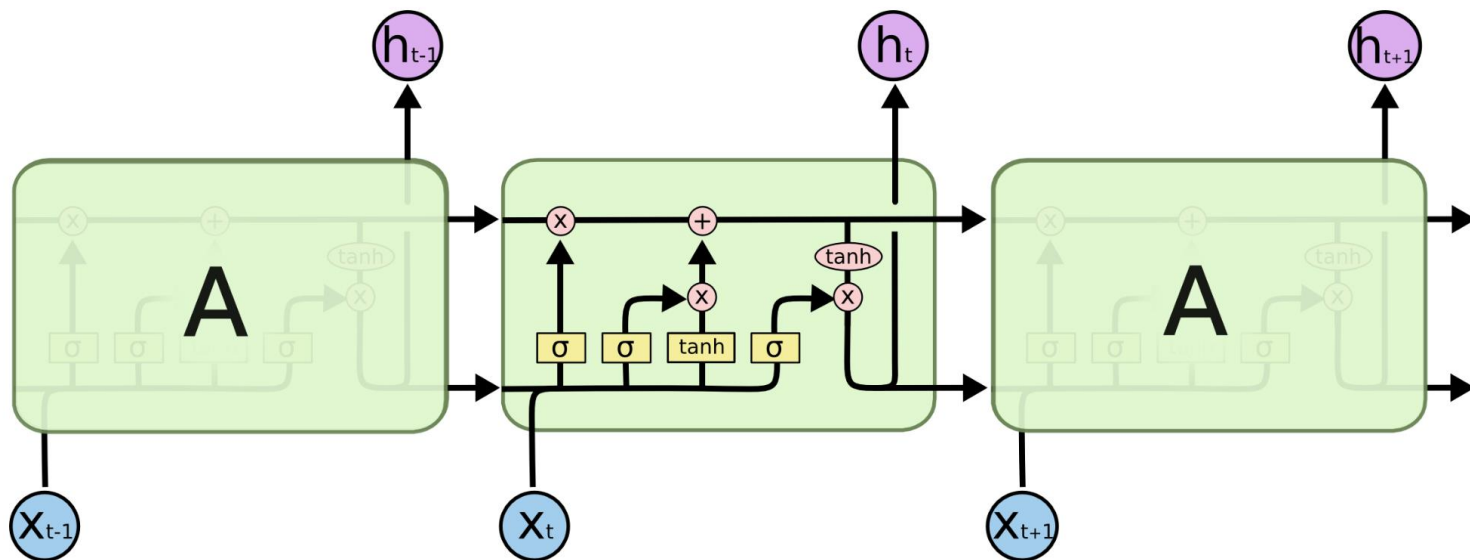
$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \circ \tanh \mathbf{c}^{(t)}$$

Gates are applied using element-wise product

All these are vectors of same length n

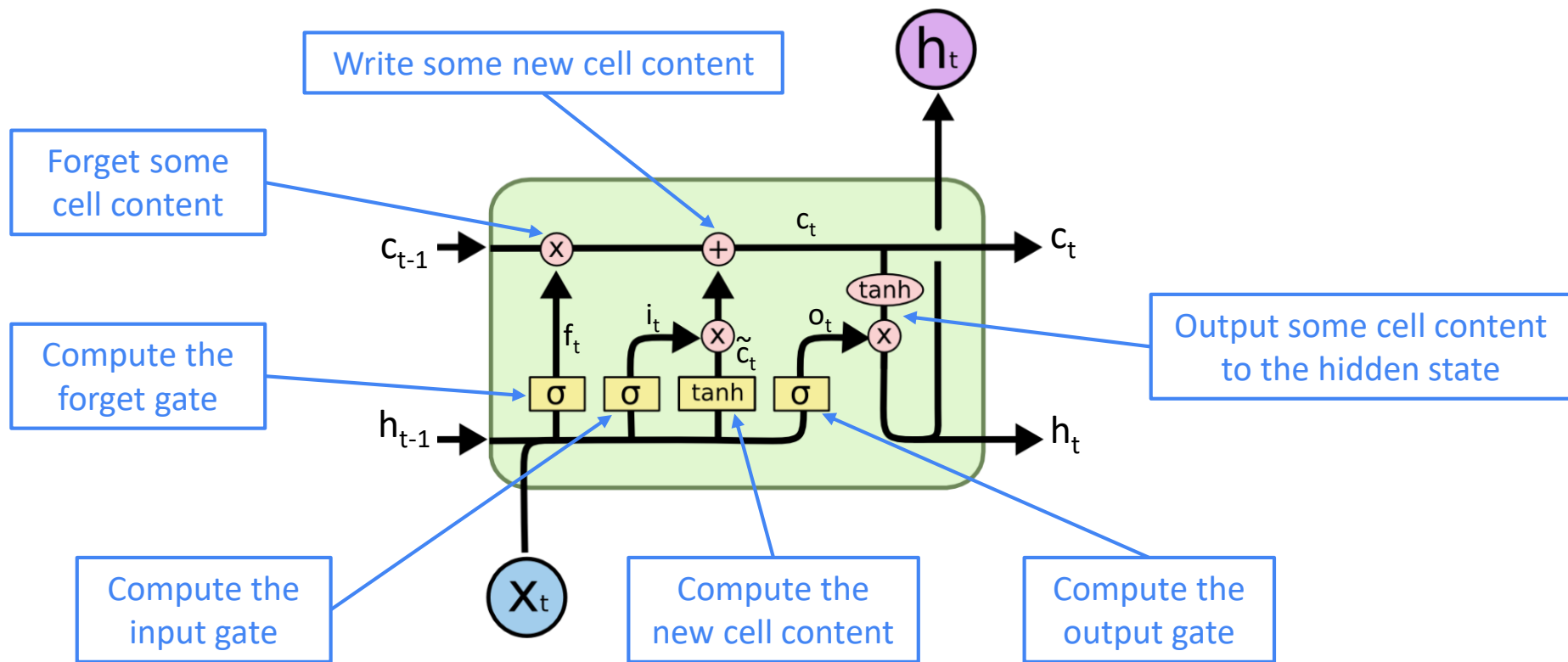
Long Short-Term Memory (LSTM)

You can think of the LSTM equations visually like this:



Long Short-Term Memory (LSTM)

You can think of the LSTM equations visually like this:



Neural Network
Layer

Pointwise
Operation

Vector
Transfer

Concatenate

Copy

How does LSTM solve vanishing gradients?

- The LSTM architecture makes it **easier** for the RNN to **preserve information over many timesteps**
 - e.g. if the forget gate is set to remember everything on every timestep, then the info in the cell is preserved indefinitely
 - By contrast, it's harder for vanilla RNN to learn a recurrent weight matrix W_h that preserves info in hidden state
- 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: real-world success

- In 2013-2015, LSTMs started achieving state-of-the-art results
 - Successful tasks include: handwriting recognition, speech recognition, machine translation, parsing, image captioning
 - LSTM became the dominant approach
- Now (2019), other approaches (e.g. Transformers) have become more dominant for certain tasks.
 - For example in WMT (a MT conference + competition):
 - In WMT 2016, the summary report contains "RNN" 44 times
 - In WMT 2018, the report contains "RNN" 9 times and "Transformer" 63 times

Source: "Findings of the 2016 Conference on Machine Translation (WMT16)", Bojar et al. 2016, <http://www.statmt.org/wmt16/pdf/W16-2301.pdf>
Source: "Findings of the 2018 Conference on Machine Translation (WMT18)", Bojar et al. 2018, <http://www.statmt.org/wmt18/pdf/WMT028.pdf>

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

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

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

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

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

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

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

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

How does this solve vanishing gradient?

Like LSTM, GRU makes it easier to retain info long-term (e.g. by setting update gate to 0)

LSTM vs GRU

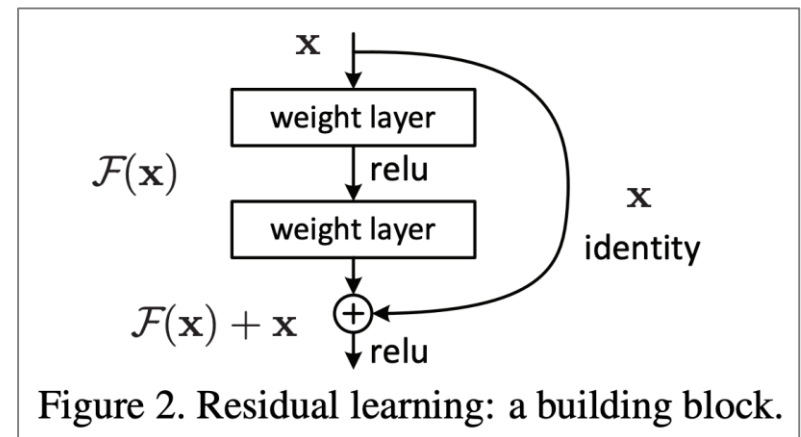
- Researchers have proposed many gated RNN variants, but LSTM and GRU are the most widely-used
- The biggest difference is that GRU is quicker to compute and has fewer parameters
- There is no conclusive evidence that one consistently performs better than the other
- LSTM is a good default choice (especially 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

Is vanishing/exploding gradient just a RNN problem?

- No! It can be a problem for all neural architectures (including **feed-forward** and **convolutional**), especially **deep** ones.
 - Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small as it backpropagates
 - Thus lower layers are learnt very slowly (hard to train)
 - Solution: lots of new deep feedforward/convolutional architectures that **add more direct connections** (thus allowing the gradient to flow)

For example:

- **Residual connections** aka “ResNet”
- Also known as **skip-connections**
- The **identity connection** **preserves information** by default
- This makes **deep** networks much **easier to train**

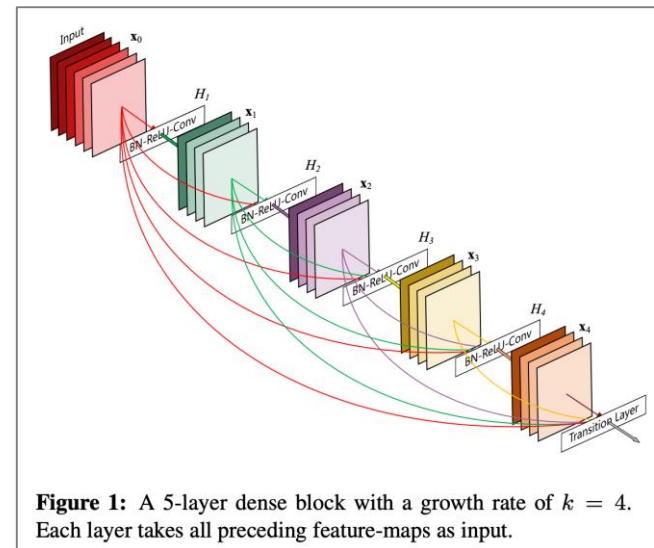


Is vanishing/exploding gradient just a RNN problem?

- No! It can be a problem for all neural architectures (including **feed-forward** and **convolutional**), especially **deep** ones.
 - Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small as it backpropagates
 - Thus lower layers are learnt very slowly (hard to train)
 - Solution: lots of new deep feedforward/convolutional architectures that **add more direct connections** (thus allowing the gradient to flow)

For example:

- **Dense connections** aka “DenseNet”
- Directly connect everything to everything!



Is vanishing/exploding gradient just a RNN problem?

- No! It can be a problem for all neural architectures (including **feed-forward** and **convolutional**), especially **deep** ones.
 - Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small as it backpropagates
 - Thus lower layers are learnt very slowly (hard to train)
 - Solution: lots of new deep feedforward/convolutional architectures that **add more direct connections** (thus allowing the gradient to flow)

For example:

- **Highway connections** aka “HighwayNet”
- Similar to residual connections, but the identity connection vs the transformation layer is controlled by a **dynamic gate**
- Inspired by LSTMs, but applied to deep feedforward/convolutional networks

Is vanishing/exploding gradient just a RNN problem?

- No! It can be a problem for all neural architectures (including feed-forward and convolutional), especially deep ones.
 - Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small as it backpropagates
 - Thus lower layers are learnt very slowly (hard to train)
 - Solution: lots of new deep feedforward/convolutional architectures that add more direct connections (thus allowing the gradient to flow)

Is vanishing/exploding gradient just a RNN problem?

- No! It can be a problem for all neural architectures (including **feed-forward** and **convolutional**), especially **deep** ones.
 - Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small as it backpropagates
 - Thus lower layers are learnt very slowly (hard to train)
 - Solution: lots of new deep feedforward/convolutional architectures that **add more direct connections** (thus allowing the gradient to flow)
- Conclusion: Though vanishing/exploding gradients are a general problem, **RNNs are particularly unstable** due to the repeated multiplication by the **same** weight matrix [Bengio et al, 1994]

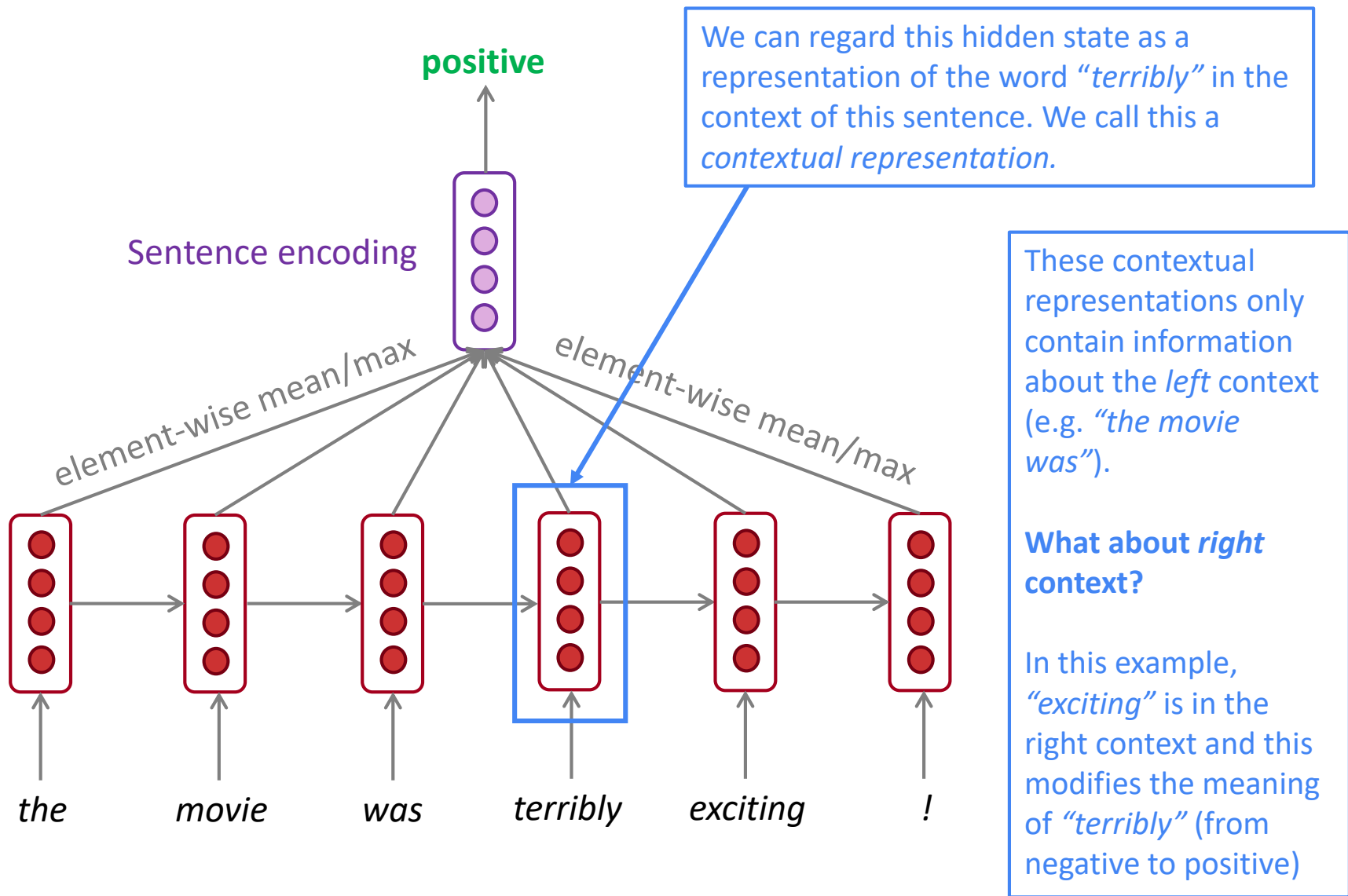
Recap

- Today we've learnt:
 - **Vanishing gradient problem:** what it is, why it happens, and why it's bad for RNNs
 - **LSTMs and GRUs:** more complicated RNNs that use gates to control information flow; they are more resilient to vanishing gradients
- Remainder of this lecture:
 - **Bidirectional RNNs**
 - **Multi-layer RNNs**

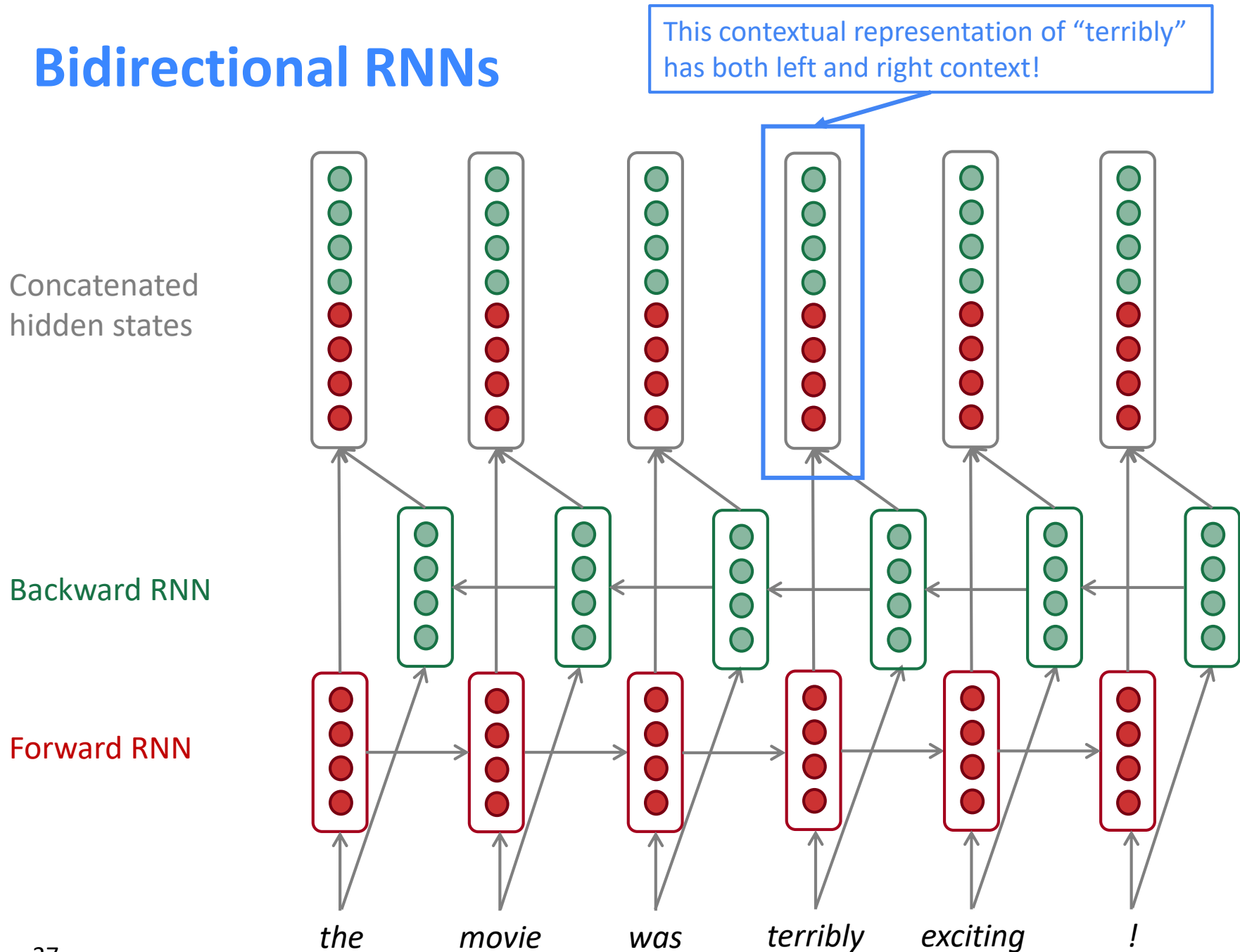
} Both of these are pretty simple

Bidirectional RNNs: motivation

Task: Sentiment Classification



Bidirectional RNNs



Bidirectional RNNs

On timestep t :

This is a general notation to mean “compute one forward step of the RNN” – it could be a vanilla, LSTM or GRU computation.

Forward RNN $\vec{h}^{(t)} = \text{RNN}_{\text{FW}}(\vec{h}^{(t-1)}, \mathbf{x}^{(t)})$

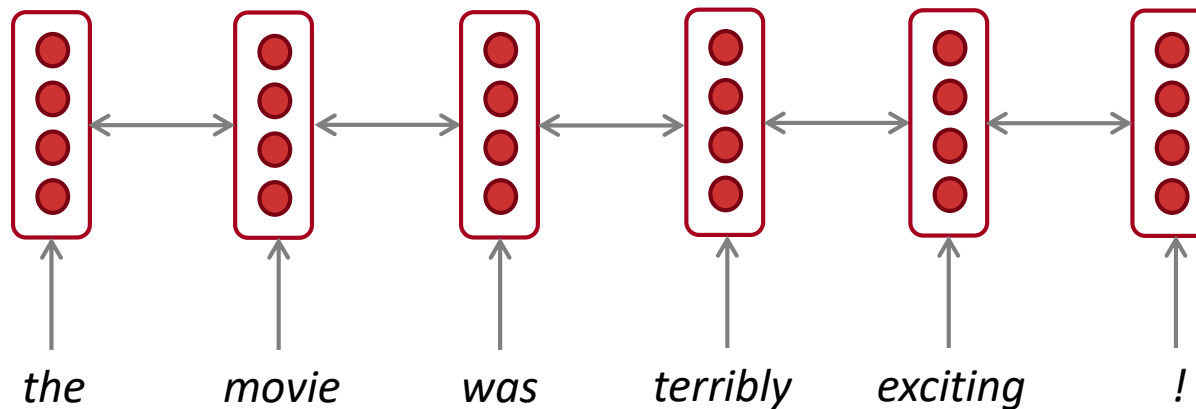
Backward RNN $\overleftarrow{h}^{(t)} = \text{RNN}_{\text{BW}}(\overleftarrow{h}^{(t+1)}, \mathbf{x}^{(t)})$

Generally, these two RNNs have separate weights

Concatenated hidden states $\mathbf{h}^{(t)} = [\vec{h}^{(t)}; \overleftarrow{h}^{(t)}]$

We regard this as “the hidden state” of a bidirectional RNN. This is what we pass on to the next parts of the network.

Bidirectional RNNs: simplified diagram



The two-way arrows indicate bidirectionality and the depicted hidden states are assumed to be the concatenated forwards+backwards states.

Bidirectional RNNs

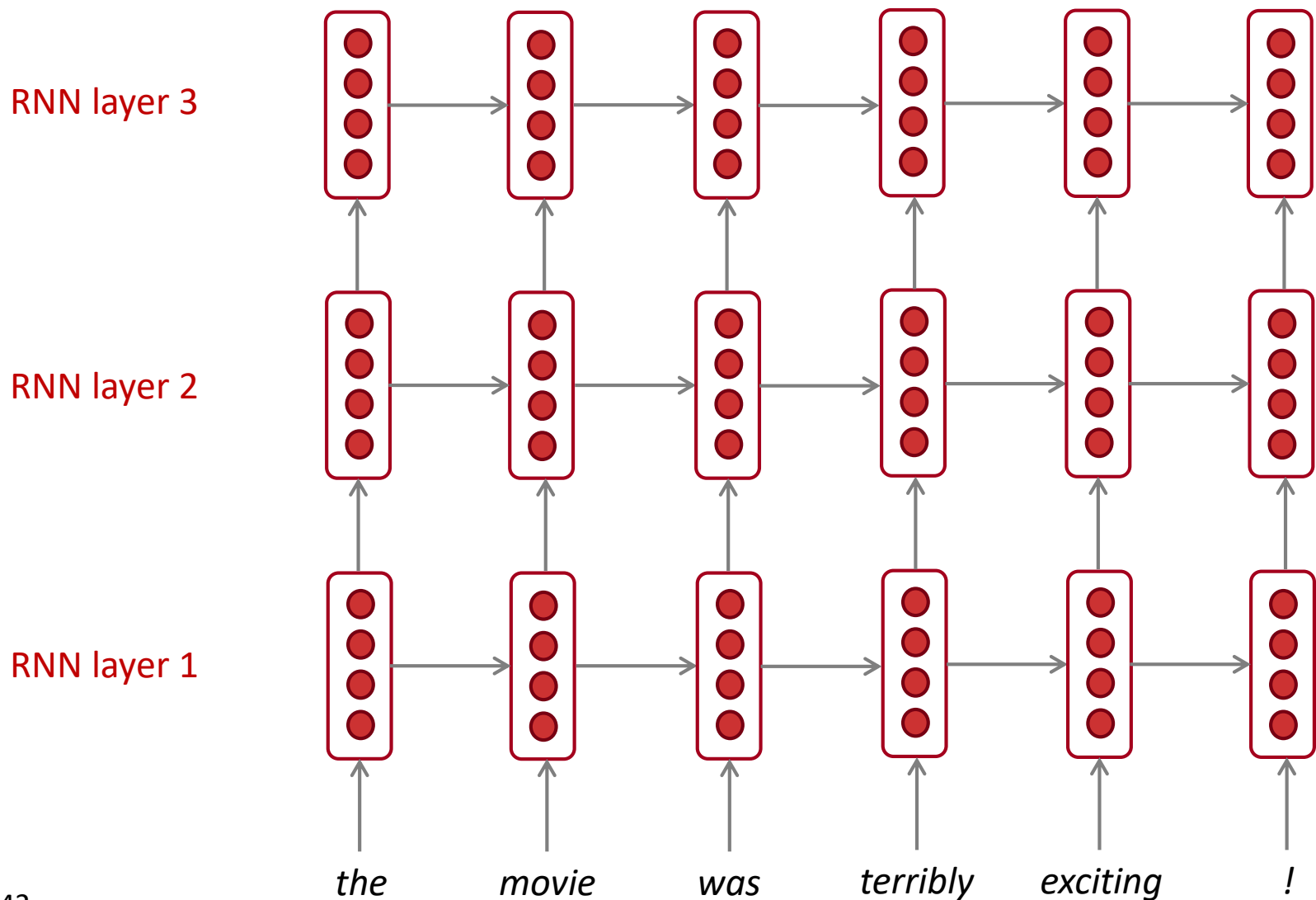
- Note: bidirectional RNNs are only applicable if you have access to the **entire input sequence**.
 - They are **not** applicable to Language Modeling, because in LM you *only* have left context available.
- If you do have entire input sequence (e.g. any kind of encoding), **bidirectionality is powerful** (you should use it by default).
- For example, **BERT** (**Bidirectional** Encoder Representations from Transformers) is a powerful pretrained contextual representation system **built on bidirectionality**.
 - You will learn more about BERT later in the course!

Multi-layer RNNs

- RNNs are already “deep” on one dimension (they unroll over many timesteps)
- We can also make them “deep” in another dimension by *applying multiple RNNs* – this is a multi-layer RNN.
- This allows the network to compute *more complex representations*
 - The *lower RNNs* should compute *lower-level features* and the *higher RNNs* should compute *higher-level features*.
- Multi-layer RNNs are also called *stacked RNNs*.

Multi-layer RNNs

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



Multi-layer RNNs in practice

- High-performing RNNs are often multi-layer (but aren't as deep as convolutional or feed-forward networks)
- For example: In a 2017 paper, Britz et al find that for Neural Machine Translation, 2 to 4 layers is best for the encoder RNN, and 4 layers is best for the decoder RNN
 - However, skip-connections/dense-connections are needed to train deeper RNNs (e.g. 8 layers)
- Transformer-based networks (e.g. BERT) can be up to 24 layers
 - You will learn about Transformers later; they have a lot of skipping-like connections