STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

CS 584 Natural Language Processing

More on RNNs

Ping Wang
Department of Computer Science
Stevens Institute of Technology

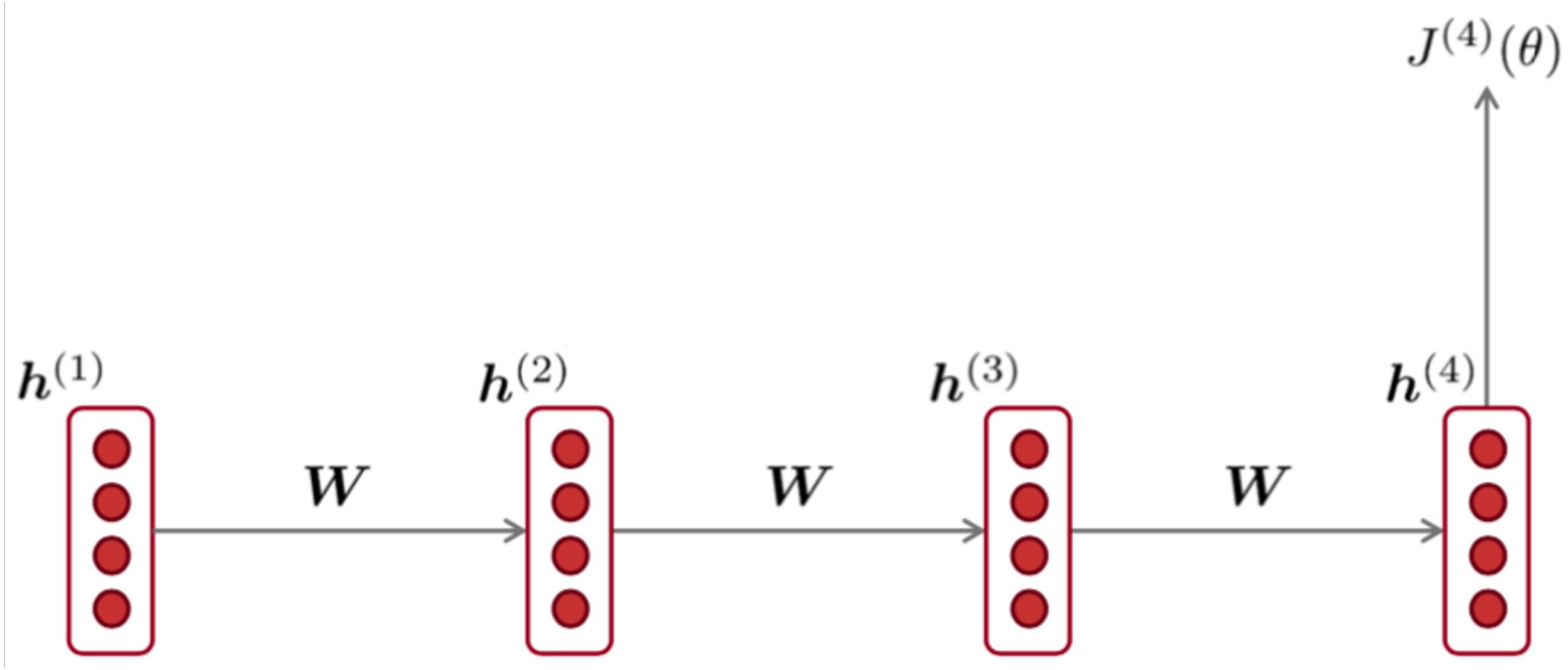
Overview

- We have covered:
  - N-gram
  - Recurrent Neural Networks (RNNs) and why they are good for language modeling (LM)
- Next we will learn:
  - Problems with RNNs and how to fix them
  - More complex RNN variants
- In the future:
  - CNN
  - Neural Machine Translation (NMT)
  - RNN-based architecture called sequence2sequence with attention

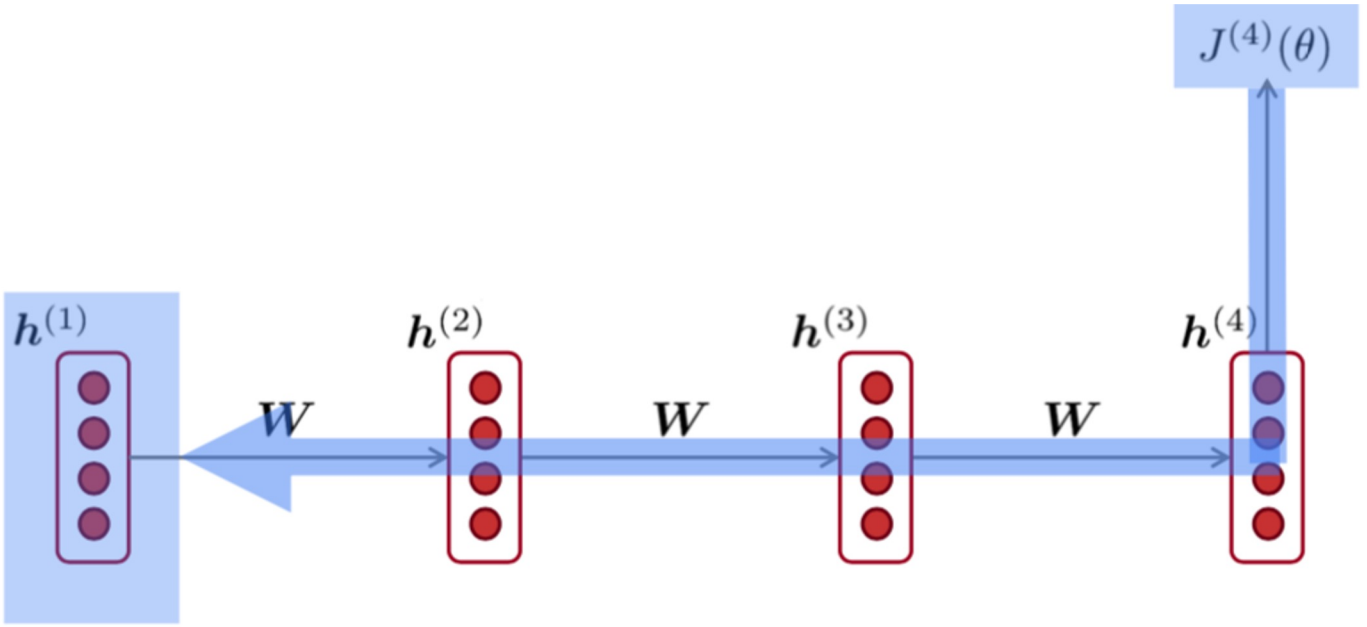
Today's lecture

- 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

Vanishing gradient intuition

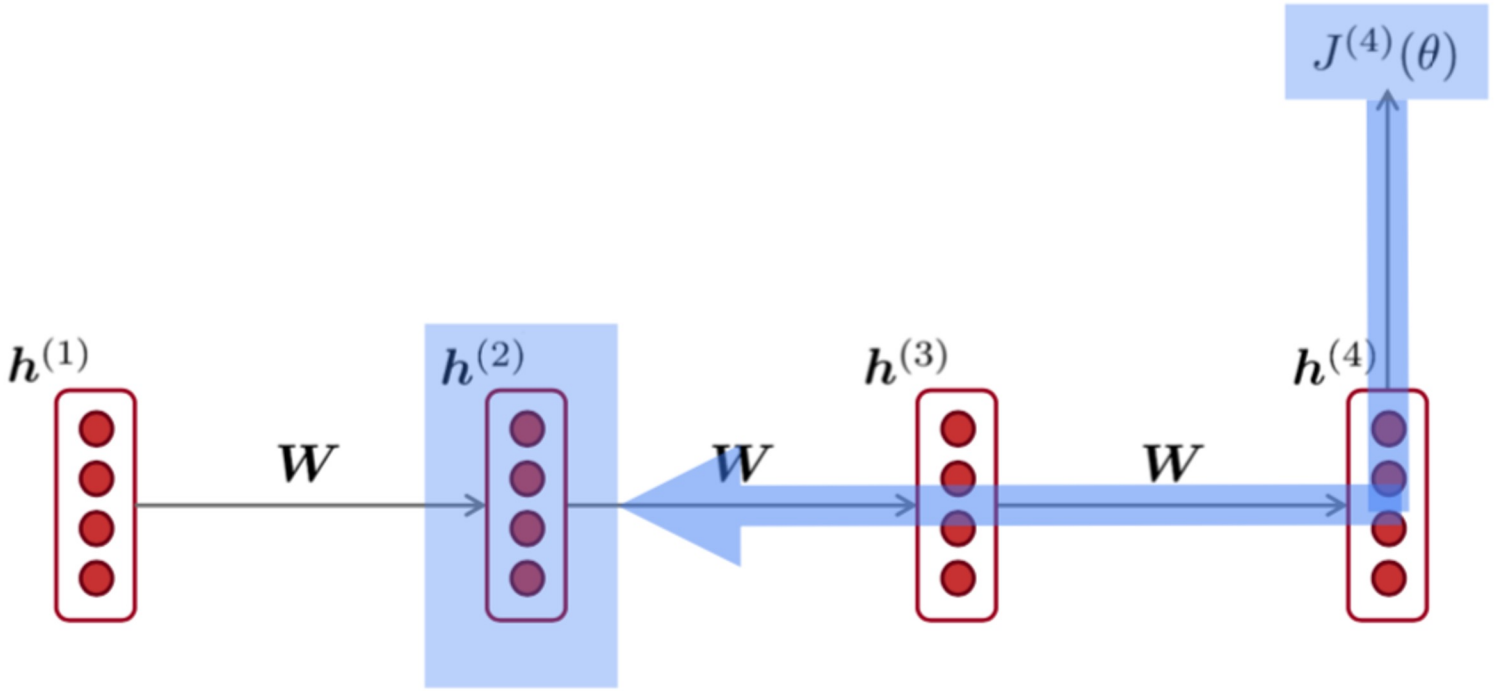


Vanishing gradient intuition



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

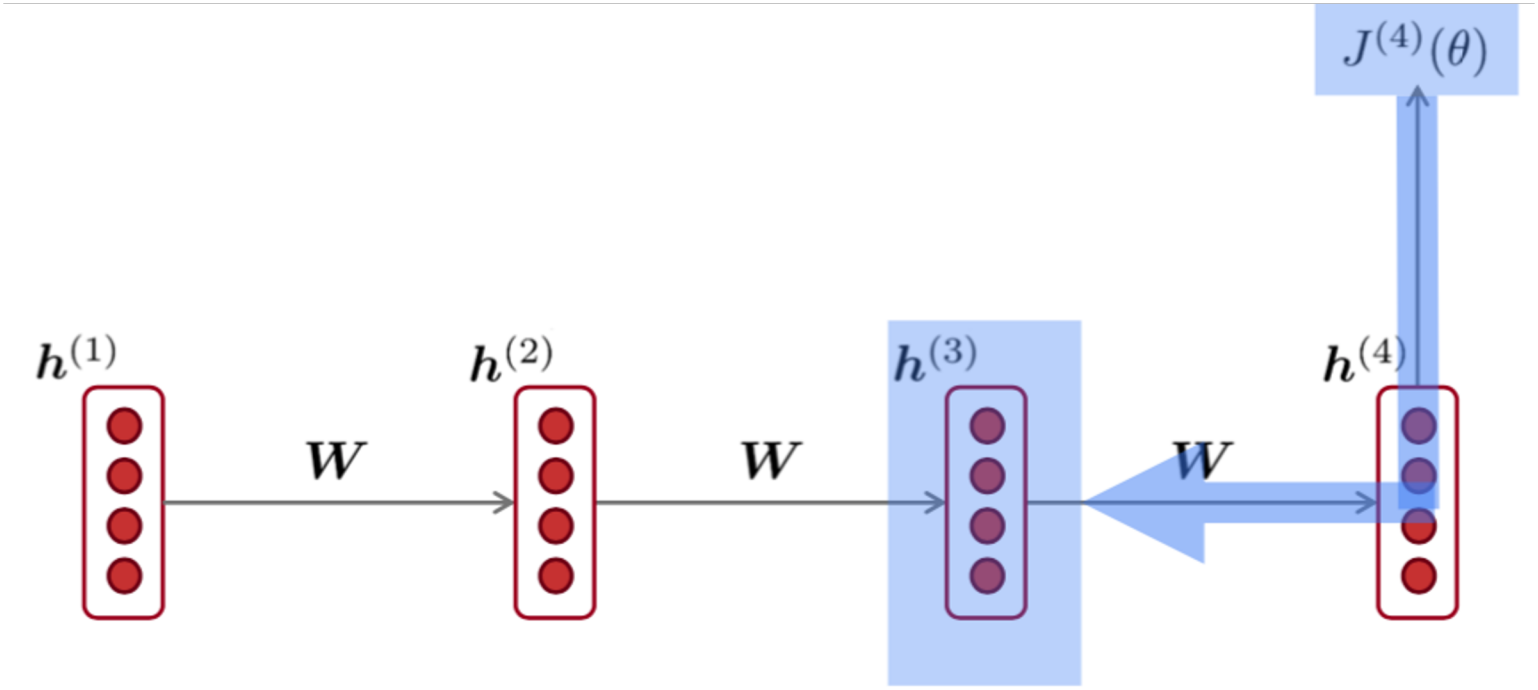
Vanishing gradient intuition



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

chain rule!

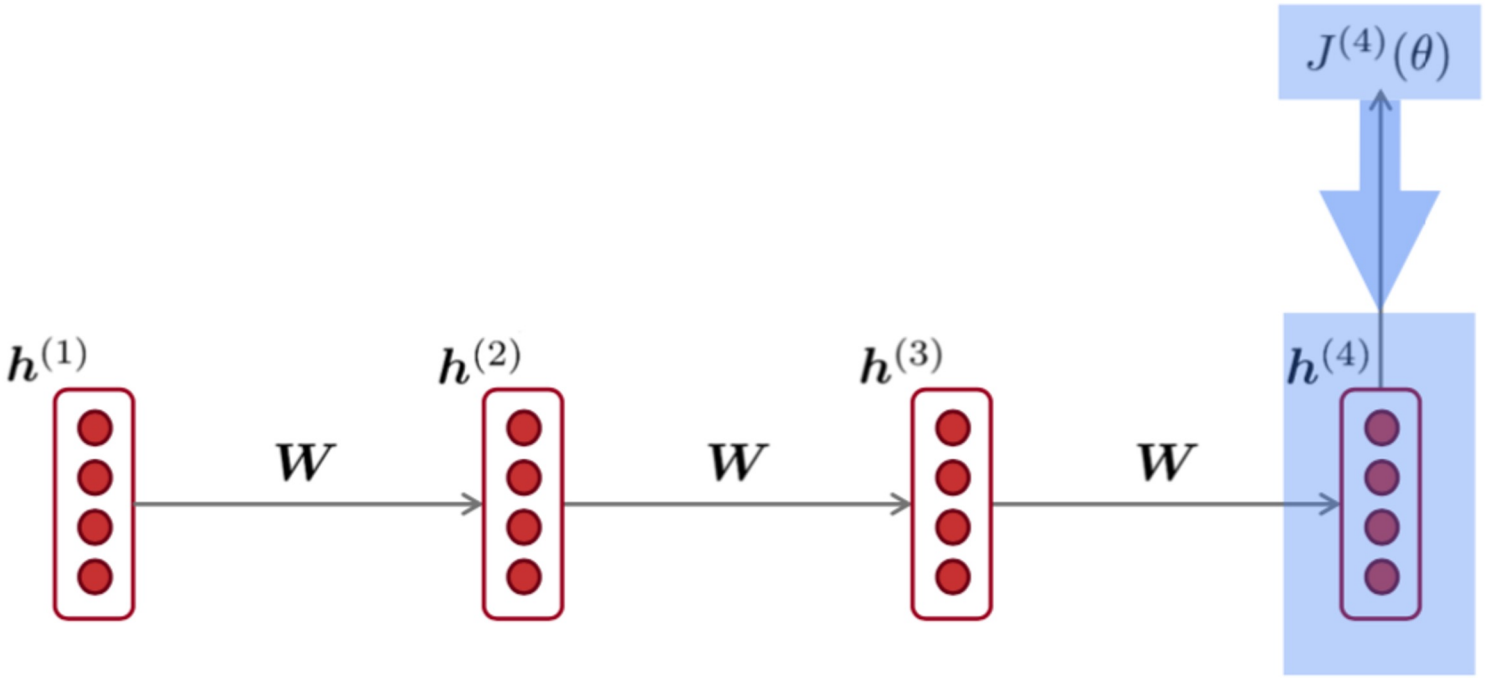
Vanishing gradient intuition



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

chain rule!

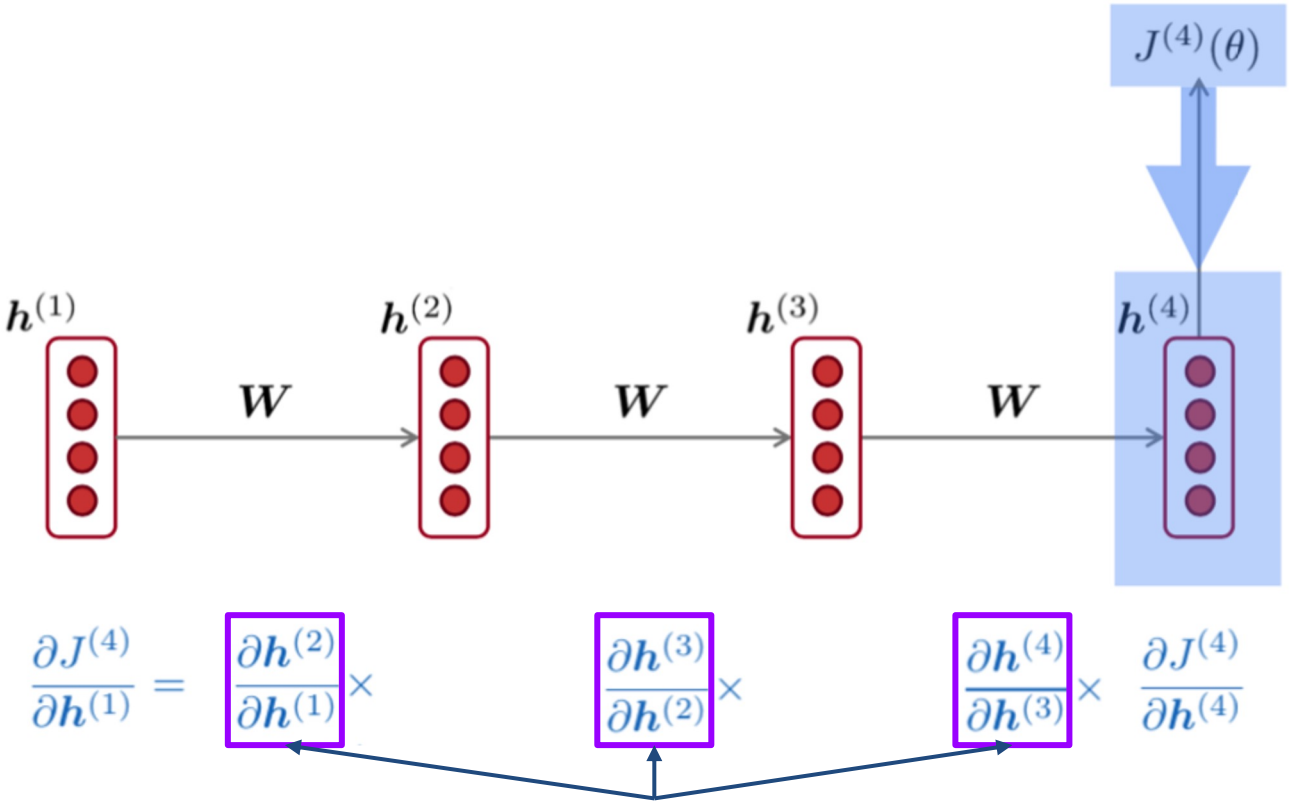
Vanishing gradient intuition



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

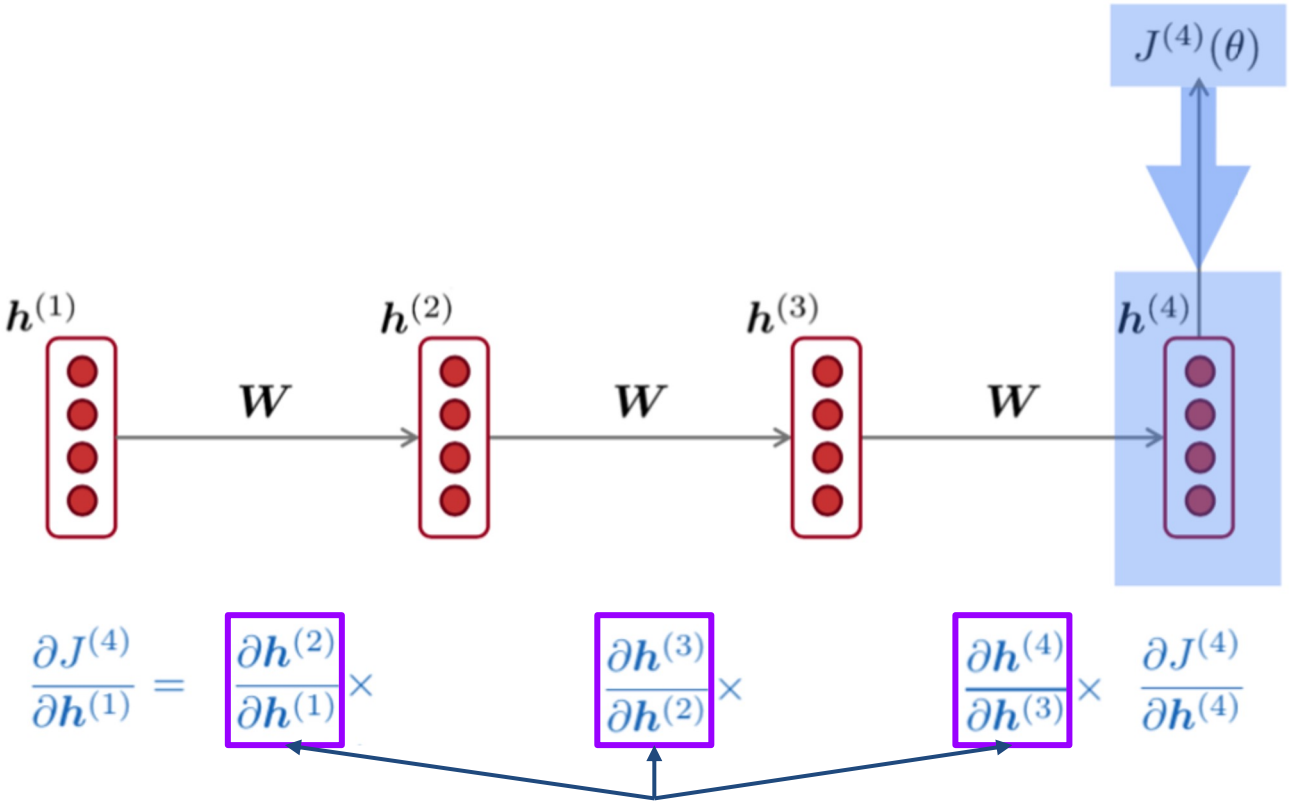
chain rule!

Vanishing gradient intuition



What if these are small?

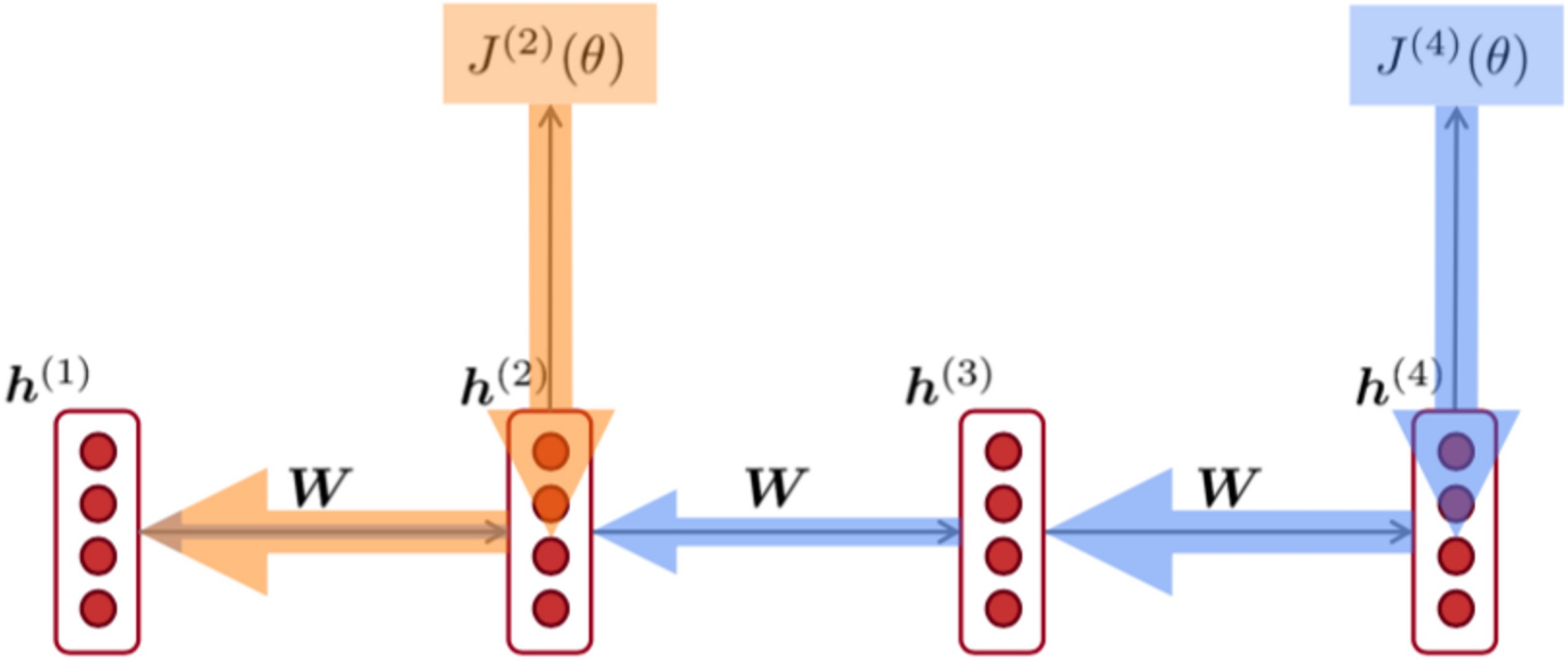
Vanishing gradient intuition



What 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 far away is lost because it's much smaller than gradient signal from close-by.

So model weights are only updated w.r.t 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:
  - There's no dependency between step t and t+n in the data
  - We have wrong parameters to capture the true dependency between t and t+n

Effect of vanishing gradient on RNN-LM

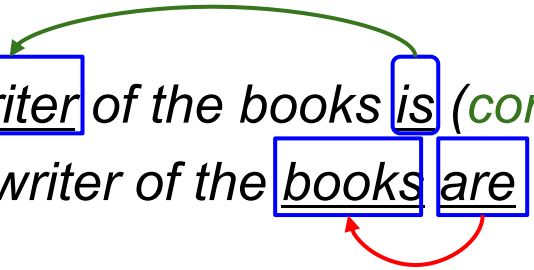
- LM task: *When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her_____*
- To learn from this training example, the RNN-LM needs 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 _____ (is? are?)*
- 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:

learning rate

$$\theta^{\text{new}} = \theta^{\text{old}} - \eta \nabla_\theta J(\theta)$$

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

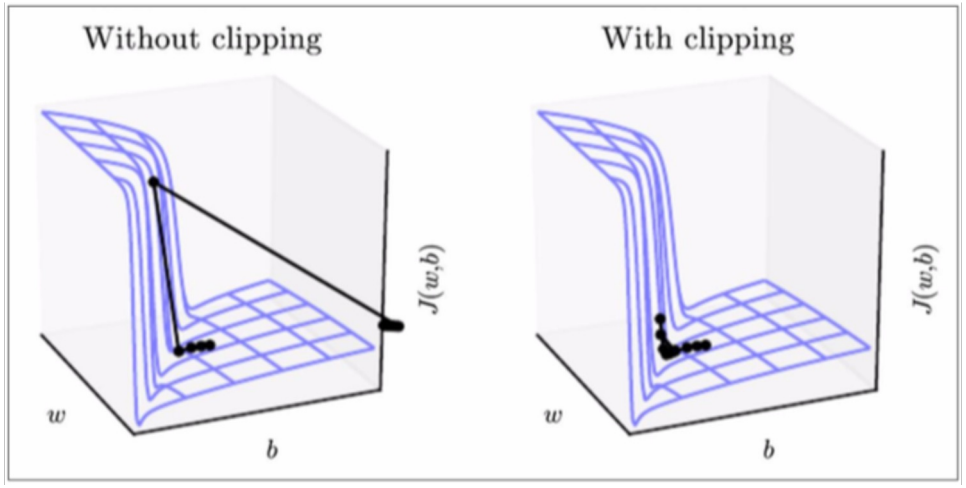
- Gradient clipping: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update

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 the 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

$$\mathbf{h}^{(t)} = \sigma\left(W_h \mathbf{h}^{(t-1)} + W_x \mathbf{x}^{(t)} + \mathbf{b}_1\right)$$

- How about a RNN with separate memory?
  →LSTM and GRU

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 $h^{(t)}$ and a cell state $c^{(t)}$
  - Both are vectors of 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 x(t), and we will compute a sequence of hidden states h(t), and cell states c(t). on time step t:

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

$$\mathbf{f}^{(t)} = \sigma(W_f \mathbf{h}^{(t-1)} + U_f \mathbf{x}^{(t)} + \mathbf{b}_f)$$

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

$$\mathbf{i}^{(t)} = \sigma(W_i \mathbf{h}^{(t-1)} + U_i \mathbf{x}^{(t)} + \mathbf{b}_i)$$

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

$$\mathbf{o}^{(t)} = \sigma(W_o \mathbf{h}^{(t-1)} + U_o \mathbf{x}^{(t)} + \mathbf{b}_o)$$

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

$$\tilde{\mathbf{c}}^{(t)} = \tanh(W_c \mathbf{h}^{(t-1)} + U_c \mathbf{x}^{(t)} + \mathbf{b}_c)$$

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

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

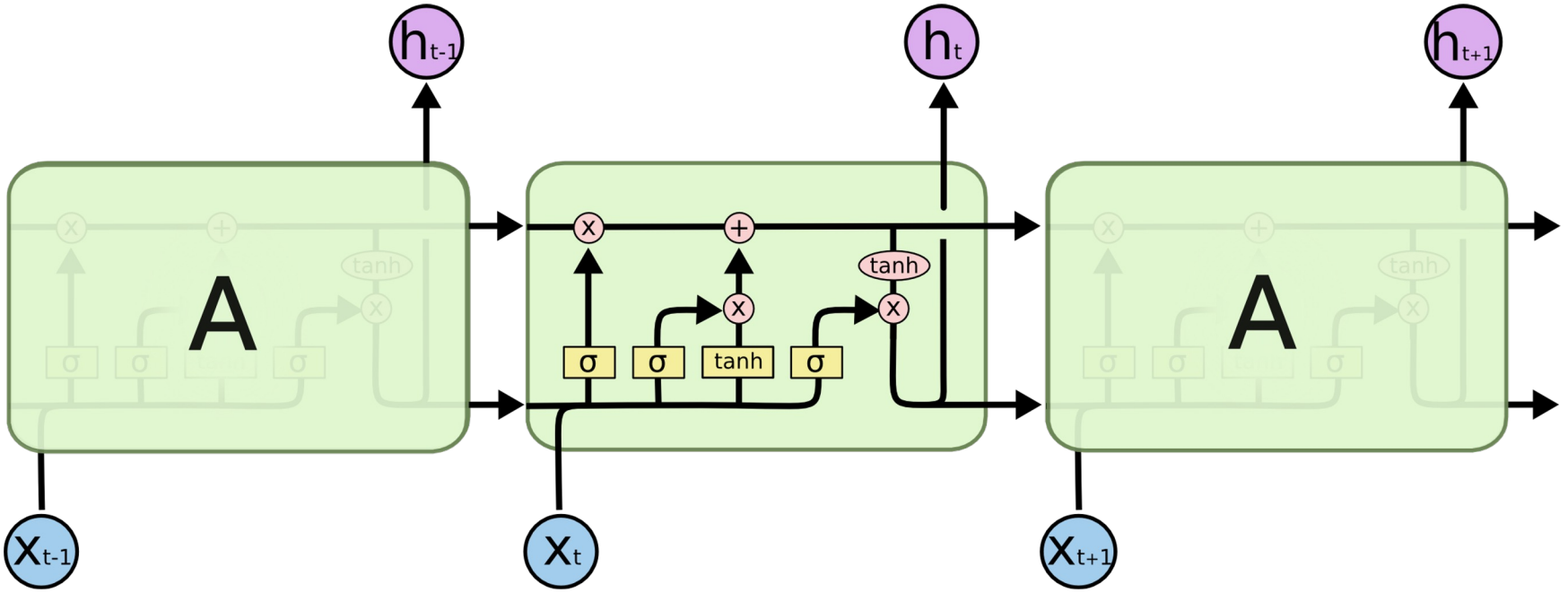
Hidden state: read ("output") some content from the cell

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

Gates are applied using element-wise product

Long Short-Term Memory (LSTM)

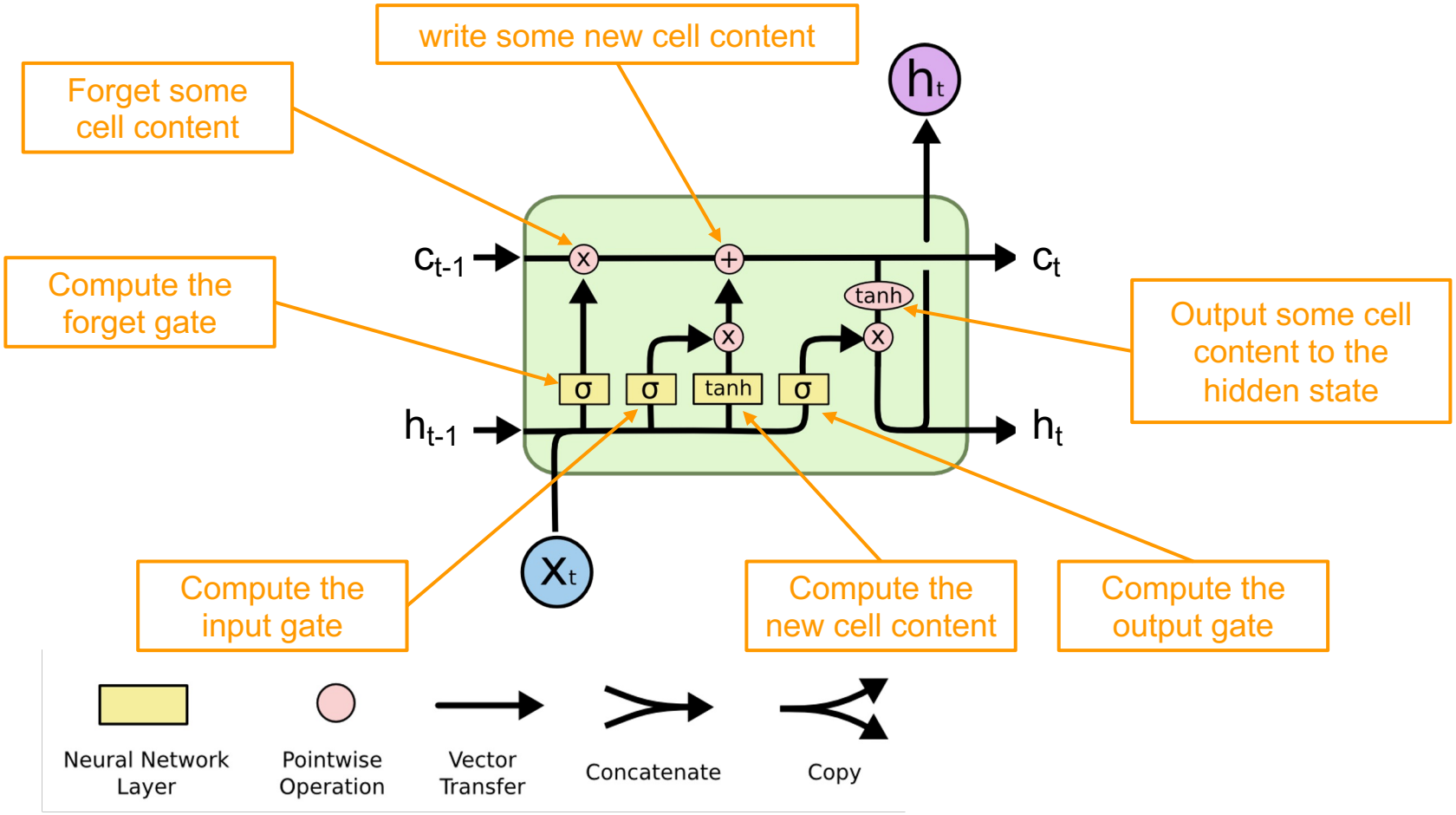
Think of the LSTM equations visually like this:



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

Long Short-Term Memory (LSTM)

Think of the LSTM equations visually like this:



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

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

Gated Recurrent Units (GRU)

Proposed by Cho et al. in 2014 as a simpler alternative to the LSTM
On each time step t, we have input $x^{(t)}$ and hidden state $h^{(t)}$ (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

$$\mathbf{u}^{(t)} = \sigma(W_u \mathbf{h}^{(t-1)} + U_u \mathbf{x}^{(t)} + \mathbf{b}_u)$$

$$\mathbf{r}^{(t)} = \sigma(W_r \mathbf{h}^{(t-1)} + U_r \mathbf{x}^{(t)} + \mathbf{b}_r)$$

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

$$\tilde{\mathbf{h}}^{(t)} = \tanh(W_h(\mathbf{r}^{(t)} \circ \mathbf{h}^{(t-1)}) + U_h \mathbf{x}^{(t)} + \mathbf{b}_h)$$

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

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)

"Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation", Cho et al. 2014, https://arxiv.org/pdf/1406.1078v3.pdf

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 network much easier to train

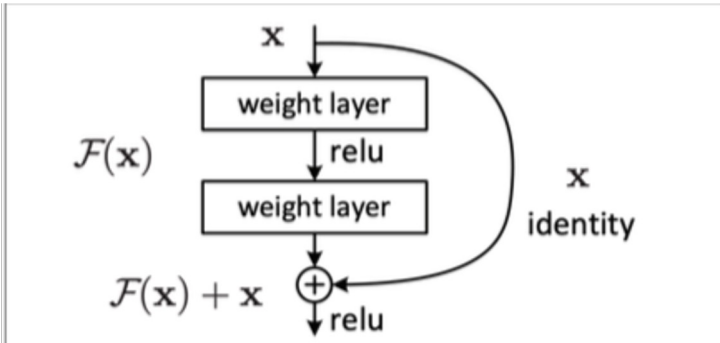


Figure 2. Residual learning: a building block.

"Deep Residual Learning for Image Recognition", He et al, 2015. https://arxiv.org/pdf/1512.03385.pdf
2023 Future Science Prize: http://www.futureprize.org/en/index.html

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 connection aka “DenseNet”
- Connects each layer to every other layer in a feed-forward fashion
- Directly connect everything to everything

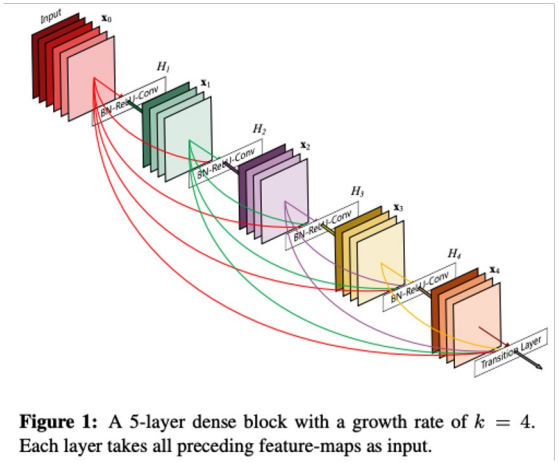


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

"Densely Connected Convolutional Networks", Huang et al, 2017. https://arxiv.org/pdf/1608.06993.pdf

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

"Highway Networks", Srivastava et al, 2015. https://arxiv.org/pdf/1505.00387.pdf

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:

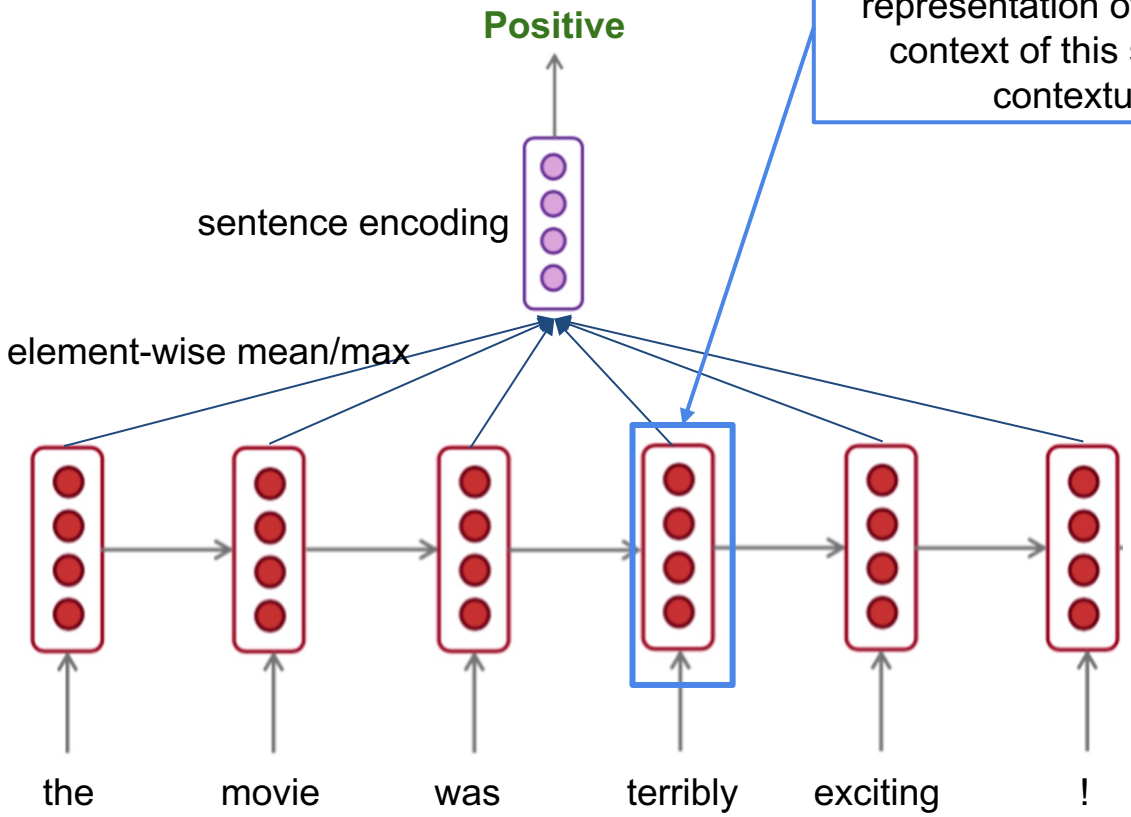
- Though vanishing/exploding gradients are a general problem, RNNs are particularly unstable due to the repeated multiplication by the same weight matrix [Bengio et al, 1994]

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

Bidirectional RNNs: motivation

Task: sentiment classification



We can regard this hidden state as a representation of the word "terribly" in the context of this sentence, we call this a contextual representation

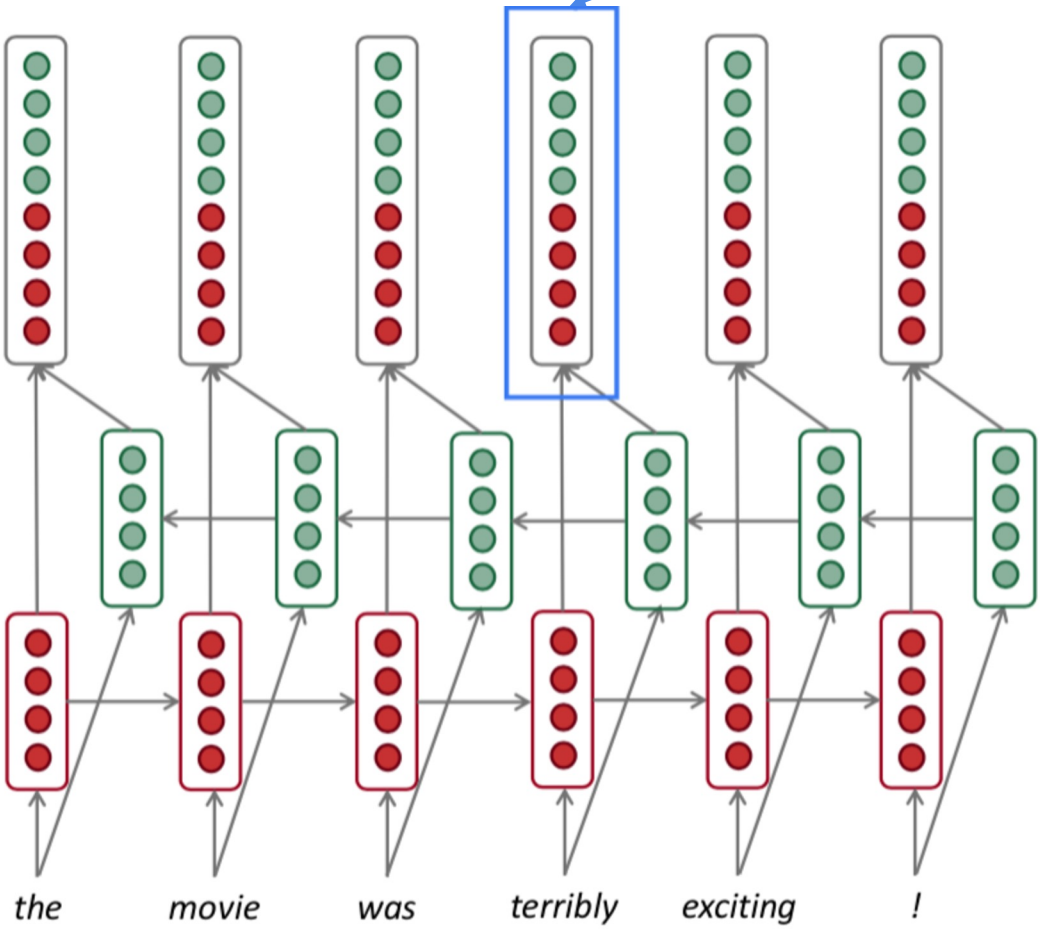
These contextual representations only contain information about the left context (e.g. "the movie was").

What about right context?

In this example, "exciting" is in the right context and this modifies the meaning of "terribly" (from negative to positive)

Bidirectional RNNs

This contextual representation of “terribly” has both left and right context!



Concatenated hidden states

Backward RNN

Forward RNN

Bidirectional RNNs

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

$$\overrightarrow{\mathbf{h}}^{(t)} = \mathrm{RNN}_{\mathrm{FW}}(\overrightarrow{\mathbf{h}}^{(t-1)}, \mathbf{x}^{(t)})$$

Backward RNN

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

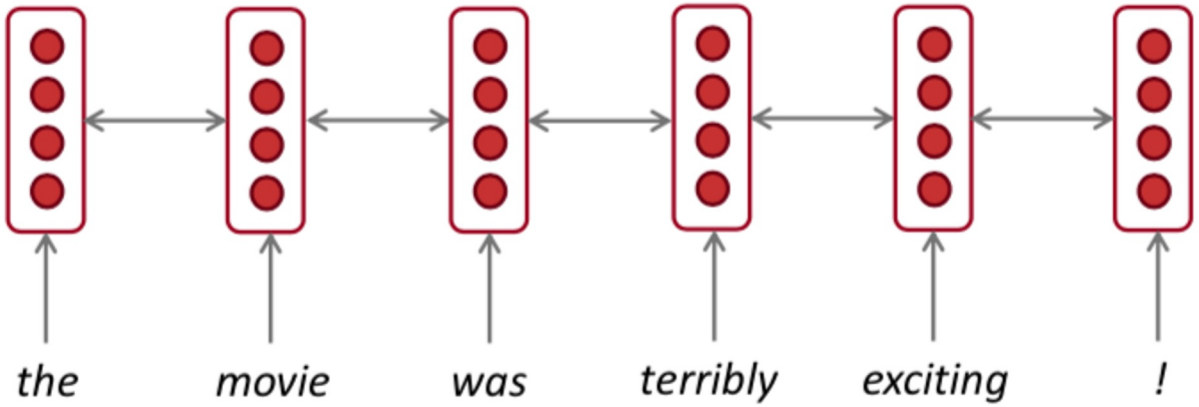
Generally, these two RNNs have separate weights

Concatenated hidden states

$$\mathbf{h}^{(t)} = [\overrightarrow{\mathbf{h}}^{(t)}; \overleftarrow{\mathbf{h}}^{(t)}]$$

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
- 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.

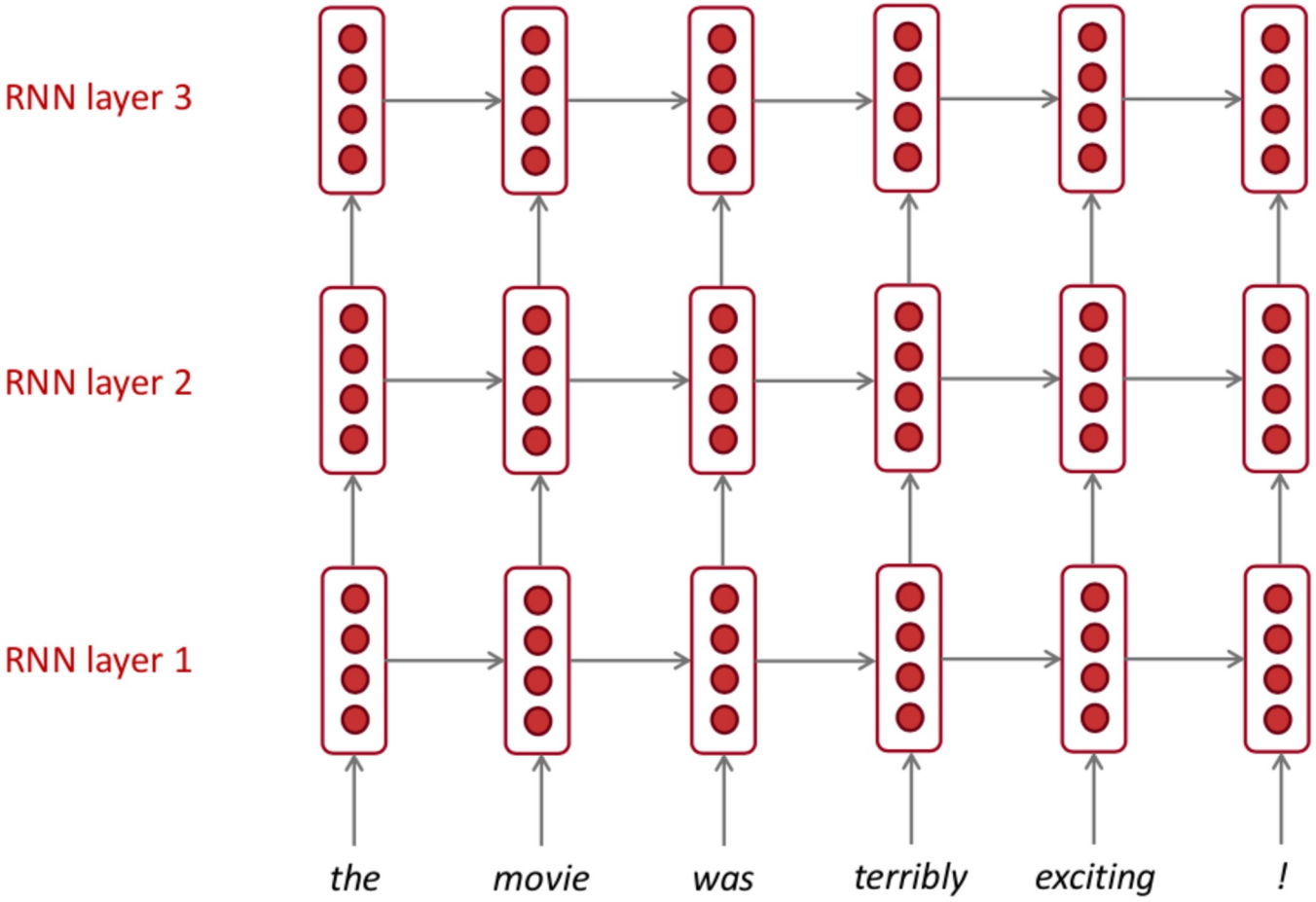
- If you do have entire input sequence (e.g. any kind of encoding), bidirectionality is powerful (you should use it by default).

- For example, BERT (**Bidirectional** Encoder Representations from Transformers) is a powerful pretrained contextual representation system built on bidirectionality.

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

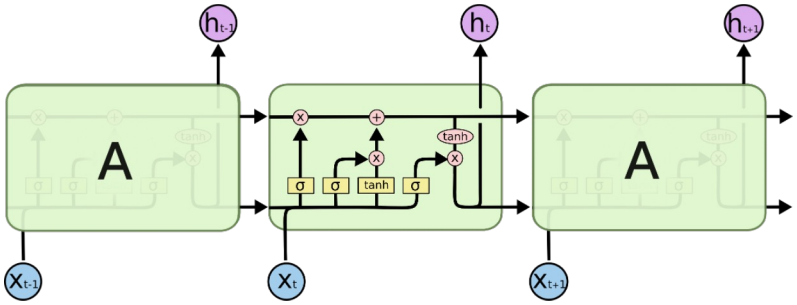
Multi-layer RNNs in practice

- High-performing RNNs are often multi-layer (but aren't as deep as convolutional or feed-forward networks)

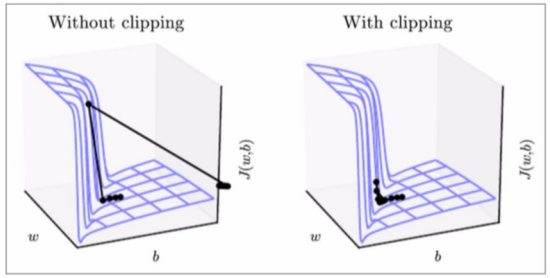
- For example: In a 2017 paper, Britz et al find that for Neural Machine Translation, 2 to 4 layers is best for the encoder RNN, and 4 layers is best for the decoder RNN
  - However, skip-connections/dense-connections are needed to train deeper RNNs (e.g. 8 layers)
- Transformer-based networks (e.g. BERT) can be up to 24 layers

Summary of RNN

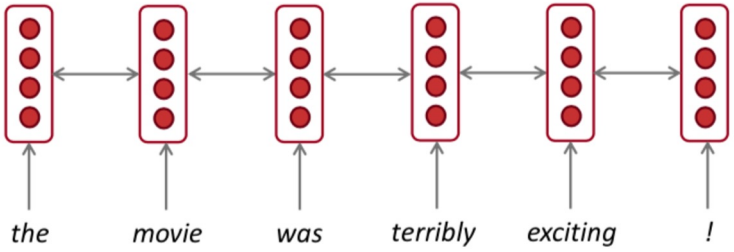
- What are the practical takeaways?



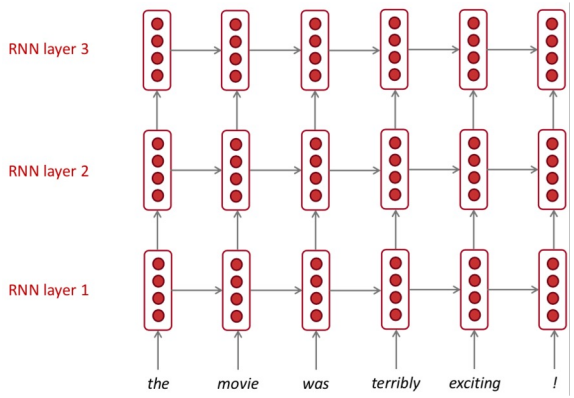
1. LSTM are powerful but GRUs are faster



2. Clip your gradients



3. Use bidirectionality when possible



4. Multi-layer RNNs are powerful, but you might need skip/dense-connections if it's deep

Readings

- **CH10 DL**; **CH15-16 NNLP**
- Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. 1997 (LSTM)
- Cho et al. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. 2014

STEVENS

INSTITUTE *of* TECHNOLOGY

THE INNOVATION UNIVERSITY®

stevens.edu

Thank You