Natural Language Processing with Deep Learning

CS224N/Ling284

Lecture 7:
Vanishing Gradients
and Fancy RNNs

Abigail See

Announcements

- Assignment 4 released today
  - Due Thursday next week (9 days from now)
  - Based on Neural Machine Translation (NMT)
    - NMT will be covered in Thursday's lecture
  - You'll use Azure to get access to a virtual machine with a GPU
    - Budget extra time if you're not used to working on a remote machine (e.g. ssh, tmux, remote text editing)
  - Get started early
    - The NMT system takes 4 hours to train!
    - **Assignment 4 is quite a lot more complicated than Assignment 3!**
    - Don't be caught by surprise!
    - Thursday's slides + notes are already online

Announcements

- Projects
  - Next week: lectures are all about choosing projects
  - It's fine to delay thinking about projects until next week
  - But if you're already thinking about projects, you can view some info/inspiration on the website's project page
  - Including: project ideas from potential Stanford AI Lab mentors. For these, best to get in contact and get started early!

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**

motivates

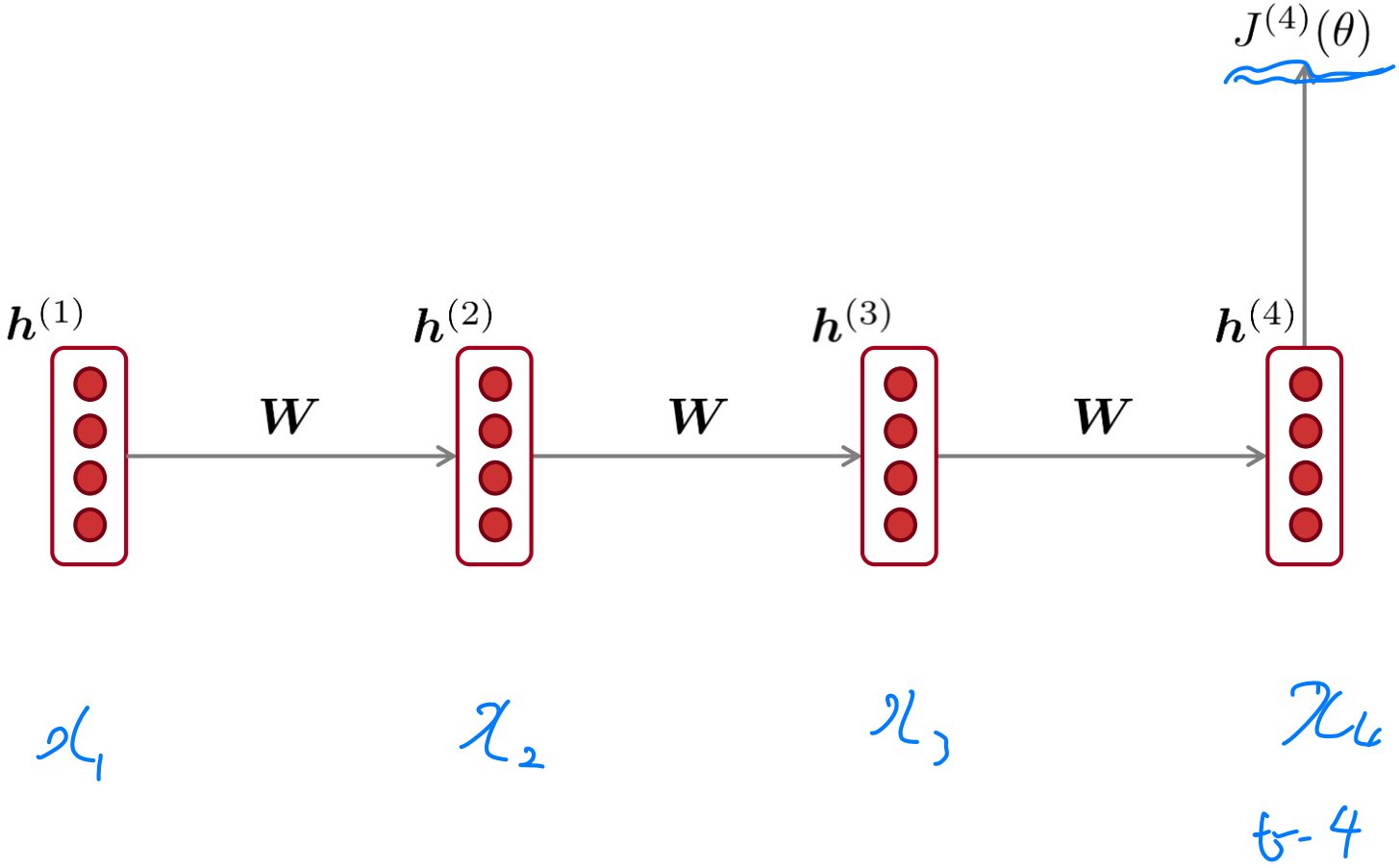
- Two new types of RNN: LSTM and GRU

- Other fixes for vanishing (or exploding) gradient:
  - Gradient clipping ✓
  - Skip connections : residual connection

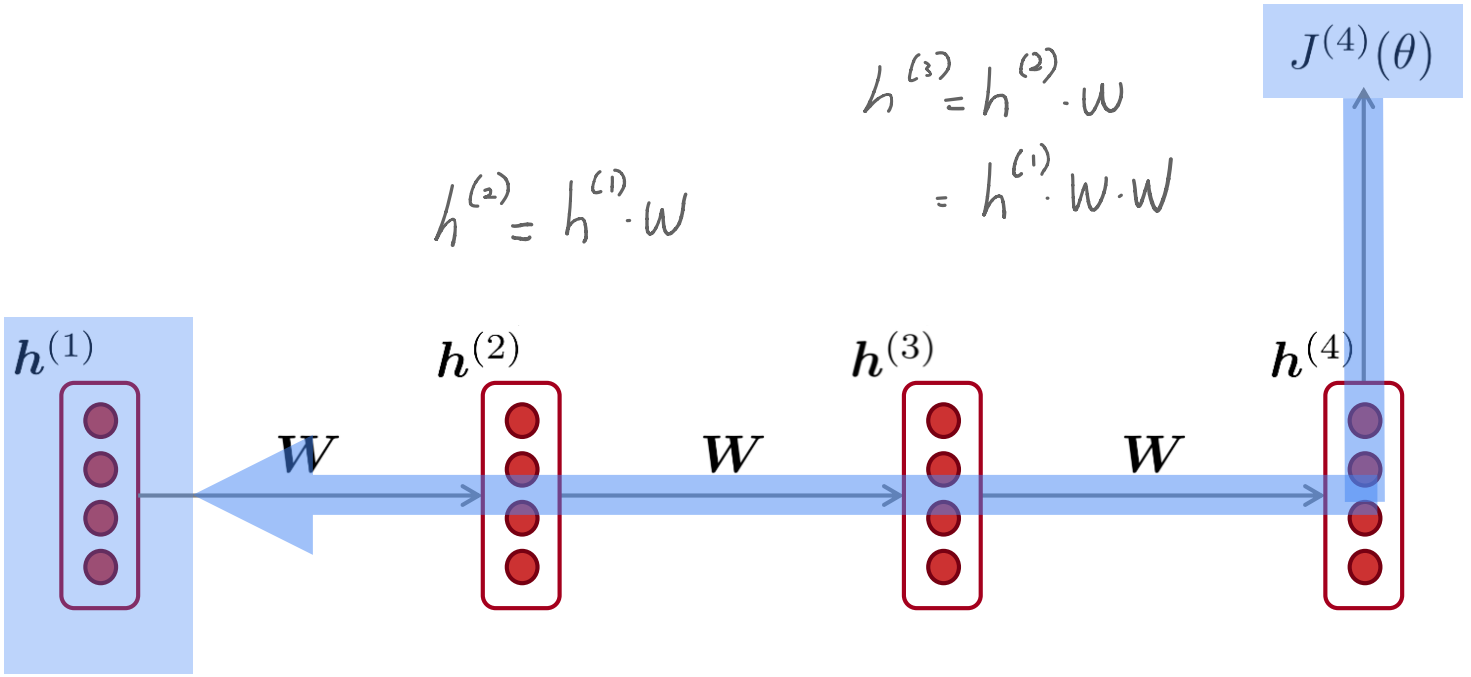
Lots of important definitions today!

- More fancy RNN variants:
  - Bidirectional RNNs
  - Multi-layer RNNs

Vanishing gradient intuition



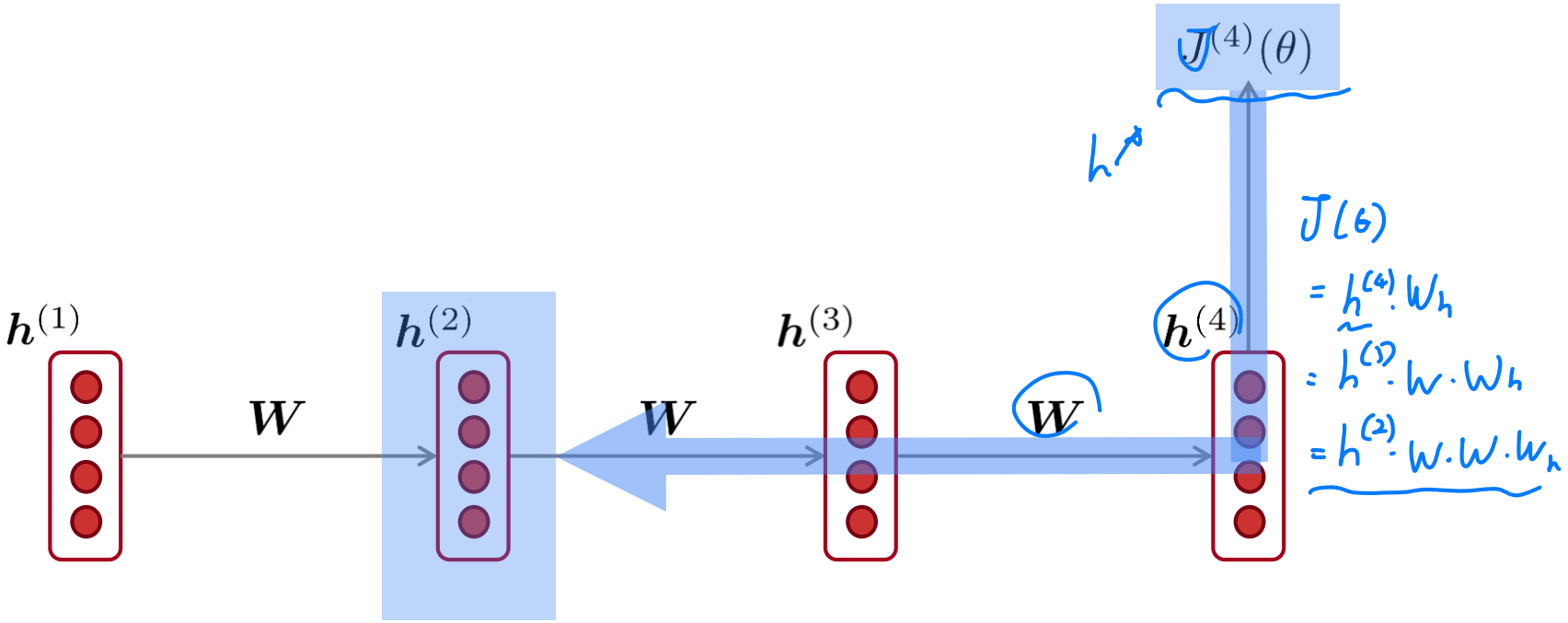
Vanishing gradient intuition



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

w.r.t.

Vanishing gradient intuition



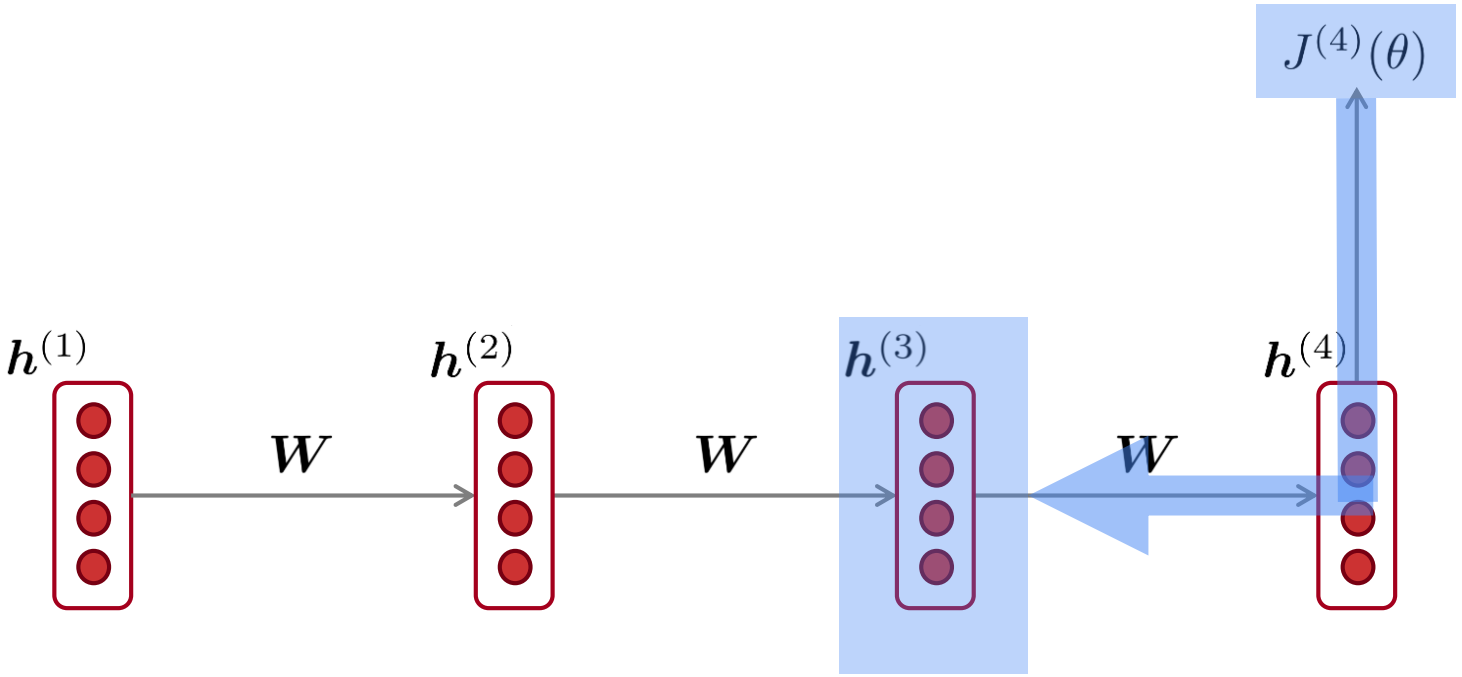
$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial J^{(4)}}{\partial h^{(2)}}$$

chain rule!

$$f(g(x)) = f'(g(x)) \cdot g'(x)$$

$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$$

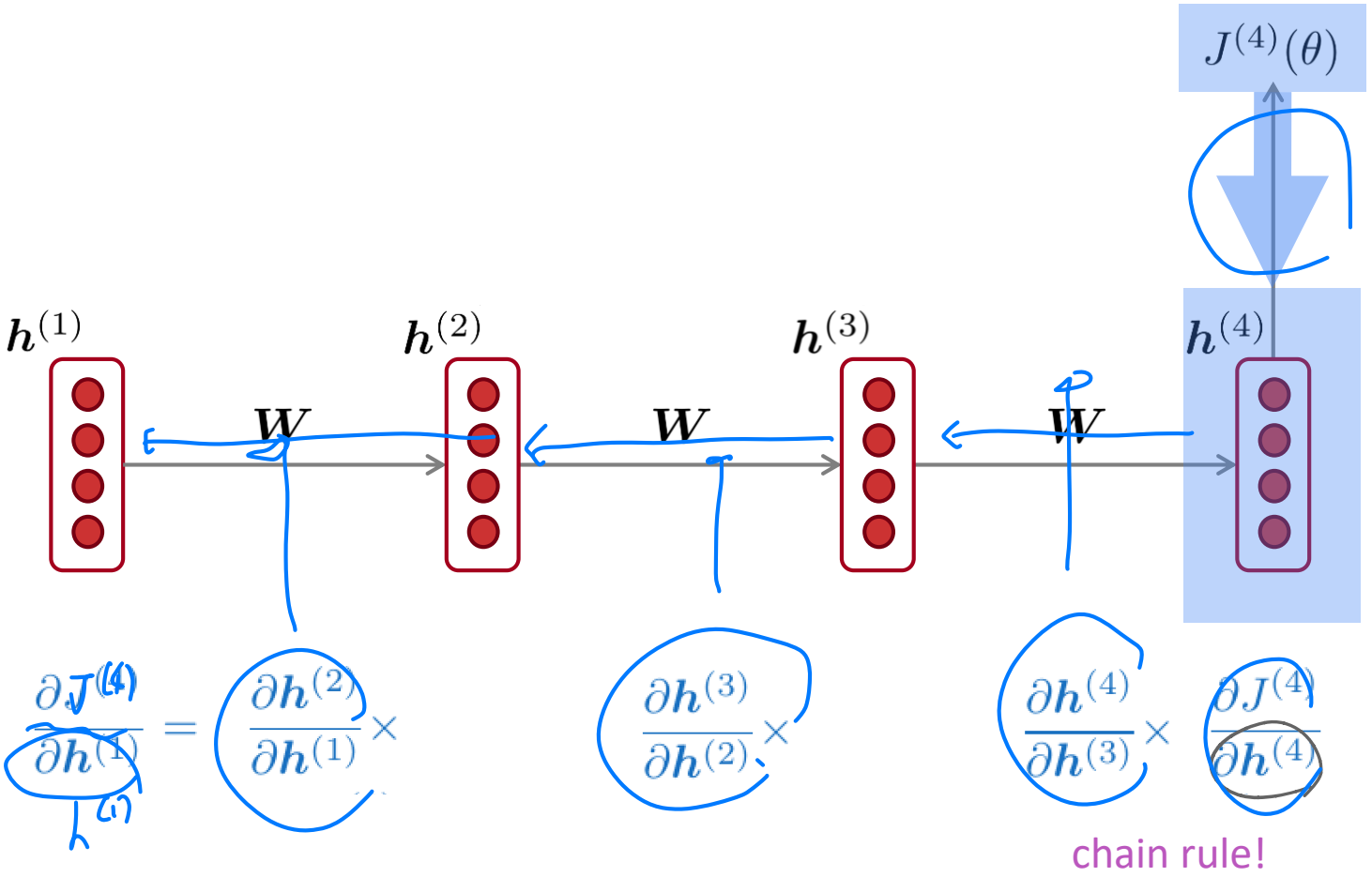
Vanishing gradient intuition



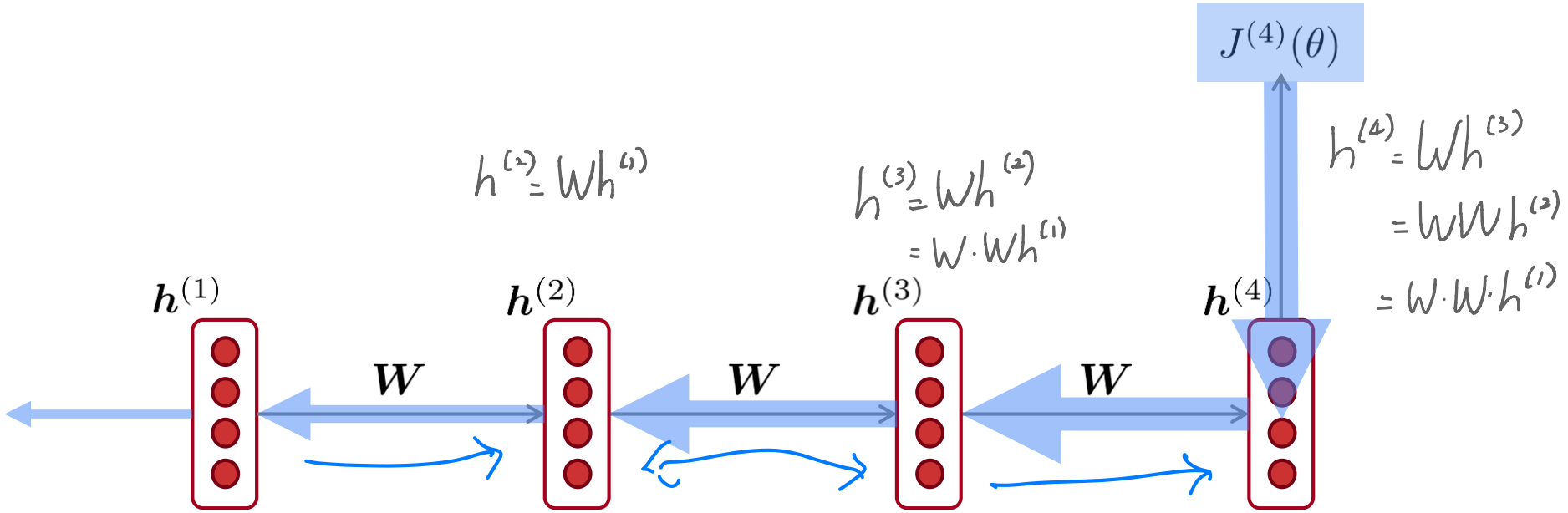
$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial J^{(4)}}{\partial h^{(3)}}$$

chain rule!

Vanishing gradient intuition



Vanishing gradient intuition



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

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

- Therefore: $$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} = \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right) \boldsymbol{W}_h$$ (chain rule)

- Consider the gradient of the loss $J^{(i)}(\theta)$ on step *i*, with respect to the hidden state $\boldsymbol{h}^{(j)}$ on some previous step *j*.

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j<t\leq i} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}$$ (chain rule)

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

If W_h is small, then this term gets vanishingly small as *i* and *j* get further apart

Source: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013. http://proceedings.mlr.press/v28/pascanu13.pdf

· BPTT

- Recall: $h_t = \tanh(W_{xh}^T \cdot x_t + W_{hh}^T \cdot h_{t-1} + b_h) = \phi_h(W^T[x_t : h_{t-1}] + b_h)$

$O_t = W_{yh}^T \cdot h_t + b_y$

$\hat{y}_t = \text{softmax}(O_t)$

C-E

- $L(\hat{y}, y) = \sum_{t=1}^{T} L_t(\hat{y}_t, y_t) = \sum_{t=1}^{T} y_t \cdot \log \hat{y}_t = \sum_{t=1}^{T} y_t \cdot \log[\text{softmax}(O_t)]$

$$\frac{\partial L}{\partial W_{yh}} = \sum_t^T \frac{\partial L_t}{\partial W_{yh}}$$

$$= \sum_t^T \underbrace{\frac{\partial L_t}{\partial \hat{y}_t}}_{①} \cdot \underbrace{\frac{\partial \hat{y}_t}{\partial O_t}}_{②} \cdot \frac{\partial O_t}{\partial W_{yh}}$$

$$\frac{\partial O_t}{\partial W_{yh}} = \frac{\partial [h_t \cdot W_{yh} + b_y]}{\partial W_{yh}} = h_t$$

$$= \sum_t^T (\hat{y}_t - y_t) \otimes h_t$$

gradient w.r.t. softmax: $\frac{\partial L_t}{\partial O_t}$

① Derivative of Loss w.r.t. softmax.

$L = -\sum_j^k [y_j \cdot \log(P_j)]$, $P_j = \text{softmax}(O_i)$

$$\frac{\partial L}{\partial O_t} = -\sum_{j=i} y_j \frac{\partial \log(P_j)}{\partial (O_i)} - \sum_{j \neq i} y_j \frac{\partial \log(P_j)}{\partial O_i}$$

② $\frac{\partial \hat{y}_t}{\partial O_t} = \frac{\partial \text{Softmax}(O_t)}{\partial O_t}$

Softmax는 $\mathbb{R}^k \to \mathbb{R}^k$ mapping function이고,

$$\frac{\partial \text{Softmax}}{\partial x} = \begin{bmatrix} \frac{\partial S_1}{\partial x_1} & \cdots & \frac{\partial S_1}{\partial x_k} \\ \vdots & \ddots & \vdots \\ \frac{\partial S_k}{\partial x_1} & \cdots & \frac{\partial S_k}{\partial x_k} \end{bmatrix}$$ (Jacobian)

대응한 i, j에 대해,

$\frac{\partial S_i}{\partial x_j} = \frac{\partial}{\partial x_j} \left(\frac{e^{x_i}}{\sum_{k=1}^k e^{x_k}} \right)$ → $f(x) = \frac{g(x)}{h(x)}$, $f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{[h(x)]^2}$

1). $\sum_{k=1}^k e^{x_k} = h(x)$, $\frac{\partial}{\partial x_j} \cdot \sum_{k=1}^k e^{x_k} = \sum \frac{\partial e^{x_k}}{\partial x_j} = e^{x_j}$

2). $e^{x_i} = g(x)$, $g'(x) = \frac{\partial e^{x_i}}{\partial x_j}$ 는 $i = j$일 때를 제외하고 0.

따라서 Jacobian의 diagonal인 경우($i = j$).

$$= -\sum_{i=j} y_j \frac{1}{P_j} \frac{\partial P_j}{\partial O_i} - \sum_{j \neq i} y_j \frac{1}{P_j} \frac{\partial P_j}{\partial O_i}$$

$$= -y_i \frac{1}{P_j} P_j (1 - P_i) - \sum_{j \neq i} y_j \frac{1}{P_j} (-P_j P_i)$$

$$= -y_i + y_i P_i + \sum_{j \neq i} y_j P_i$$

$$= P_i \left(\sum_j y_j\right) - y_i = P_i - y_i$$

b_y에 대해서도,

$$\frac{\partial L}{\partial b_y} = \sum_t^T \frac{\partial L_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial b_y}$$

$$= \sum_t^T (\hat{y}_t - y_t)$$

$$\therefore \frac{\partial \frac{e^{x_i}}{\sum e^{x_k}}}{\partial x_j} = \frac{e^{x_i} \cdot \sum e^{x_k} - e^{x_j} \cdot e^{x_i}}{\left[\sum e^{x_k}\right]^2}$$

$$= S_i (1 - S_j)$$

$i \neq j$ 일 경우,

$$\frac{\partial \frac{e^{x_i}}{\sum e^{x_k}}}{\partial x_j} = \frac{0 \cdot \sum e^{x_k} - e^{x_j} \cdot e^{x_i}}{\left[\sum e^{x_k}\right]^2}$$

$$= -S_j S_i$$

where

$$S(x) : \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} \mapsto \begin{bmatrix} S_1 \\ \vdots \\ S_k \end{bmatrix}$$

$t \to t+1$ 시점에서의 W_{hh}에 대한 gradient는,

$$\frac{\partial L_{t+1}}{\partial W_{hh}} = \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \cdot \frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}} \cdot \frac{\partial h_{t+1}}{\partial W_{hh}}$$

여기서 h_t는 h_t에 의존함 ($h_{t+1} = \tanh(W_{xh}^T \cdot X_t + W_{hh}^T \cdot h_t + b_h)$).

이는 $t-1 \to t$에도 적용되고,

$$\frac{\partial L_{t+1}}{\partial W_{hh}} = \sum_{k=1}^{t+1} \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \cdot \frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}} \cdot \frac{\partial h_{t+1}}{\partial h_k} \cdot \frac{\partial h_k}{\partial W_{hh}}$$

chain rule itself: $\frac{\partial h_3}{\partial h_1} = \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1}$

또한, vector에 대해 vector로 derivative를 구하고 있으므로, 그 결과는 Jacobian matrix.

이를 다시 쓰면,

$$\frac{\partial L_{t+1}}{\partial W_{hh}} = \sum_{k=1}^{t+1} \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \cdot \frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}} \cdot \left(\prod_{j=k}^{t} \frac{\partial h_{j+1}}{\partial h_j} \right) \cdot \frac{\partial h_k}{\partial W_{hh}}$$

why?

$\Rightarrow$ 2-norm의 최댓값이 1

$\Rightarrow$ tanh/sigmoid의 값이 1/0.25

1 / 0.25

$\frac{\partial \tanh}{\partial x}$

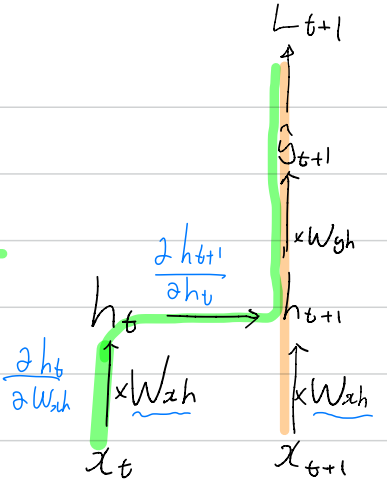
gradient vanishing

모든 time step에서의 gradient w.r.t. W_{hh}를 모으면,

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t}^{T} \sum_{k=1}^{t+1} \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \cdot \frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}} \cdot \frac{\partial h_{t+1}}{\partial h_k} \cdot \frac{\partial h_k}{\partial W_{hh}}$$

W_{xh} 도 이와 비슷하게,

$$\frac{\partial L_{t+1}}{\partial W_{xh}} = \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \cdot \frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}} \cdot \frac{\partial h_{t+1}}{\partial W_{xh}} + \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \cdot \frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}} \cdot \frac{\partial h_{t+1}}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{xh}}$$



이를 $1 \to t+1$ 까지 확장하면,

$$\frac{\partial L_{t+1}}{\partial W_{xh}} = \sum_{k=1}^{t+1} \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \cdot \frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}} \cdot \frac{\partial h_{t+1}}{\partial h_k} \cdot \frac{\partial h_k}{\partial W_{xh}}$$

모든 step에서의 Loss를 계산하면,

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t}^{T} \sum_{k=1}^{t+1} \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \cdot \frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}} \cdot \frac{\partial h_{t+1}}{\partial h_k} \cdot \frac{\partial h_k}{\partial W_{xh}}$$

Vanishing / Exploding Gradient Desent.

위에서 언급한 바와 같이 hidden state를 derivative 하면,

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

이중에 하나 $\frac{\partial h_{j+1}}{\partial h_j}$ 를 보자. diag는 vector를 diagonal matrix로 변환한다 (Jacobian)

$$\frac{\partial h_{j+1}}{\partial h_j} = \text{diag}\left(\phi_h'\left(W_{xh}^T \cdot x_{j+1} + W_{hh}^T \cdot h_j + b_h\right) \cdot W_{hh}\right)$$

- 따라서 t-k step 만큼 back prop을 한다면,

$$\prod_{j=k}^{t} \frac{\partial h_{j+1}}{\partial h_j} = \prod_{j=k}^{t} \mathrm{diag}\left(\phi_h' \left(W_{xh}^T \cdot x_{j+1} + W_{hh}^T \cdot h_j + b_h\right) \cdot W_{hh}\right)$$

- 이에 대해 eigen decomposition을 수행하면 $\lambda_1, \cdots, \lambda_n$과 이에 대응하는 $v_1, \cdots, v_n$을 얻음.
- h_{j+1}의 변화 Δh_{j+1}가 v_i 방향으로 갈 때, 이에 해당하는 eigenvalue λ_i로 나타낼 수 있음.

$$\therefore \lambda_i \Delta h_{j+1}$$

- 이러한 Jacobian을 연속해서 곱하는 것은 λ_i^t을 곱하는 것으로 이해할 수 있음.
- 따라서 eigenvalue가 1보다 크거나 작다면, explode/vanish 함

$$\left\| \frac{\partial h_{j+1}}{\partial h_j} \right\| \leq \|W_{hh}\| \left\| \mathrm{diag}\left(\phi_h'\left(W_{xh}^T \cdot x_{j+1} + W_{hh}^T \cdot h_j + b_h\right)\right) \right\| \leq \gamma_w \cdot \gamma_h$$

eigenvalue

→ 이게 1보다 작아야 upper bound?

$\phi_h' = \sigma \Rightarrow \gamma_h = 0.25$

$\phi_h' = \tanh \Rightarrow \gamma_h = 1$

$$\left\| \frac{\partial h_{j+1}}{\partial h_j} \right\| = \left\| \prod_{j=k}^{t} \frac{\partial h_{j+1}}{\partial h_j} \right\| \leq (\gamma W \gamma_h)^{t-k}$$

↳ explode / vanish

Vanishing gradient proof sketch

- Consider matrix L2 norms:

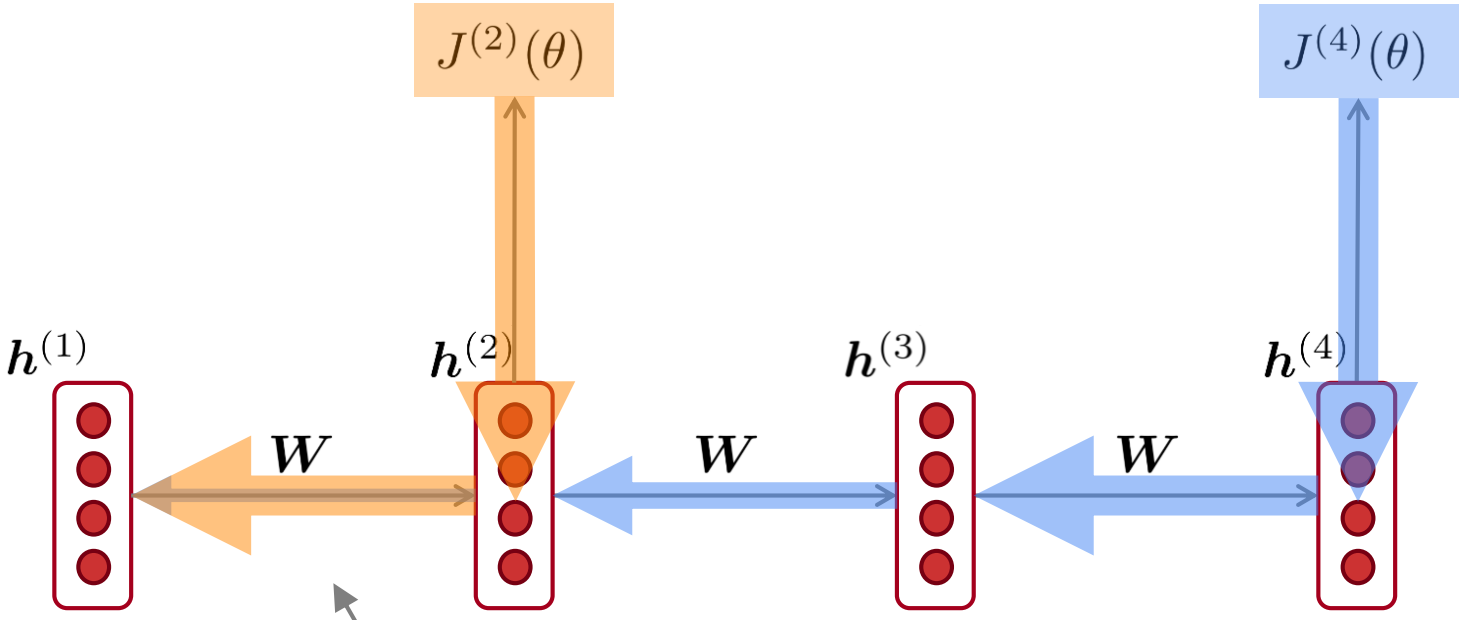
$$\left\| \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} \right\| \leq \left\| \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \right\| \|\boldsymbol{W}_h\|^{(i-j)} \prod_{j<t\leq i} \left\| \text{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x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

- Pascanu et al showed that that if the largest eigenvalue of W_h is less than 1, then the gradient $\left\| \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} \right\|$ 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

Source: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013. http://proceedings.mlr.press/v28/pascanu13.pdf

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

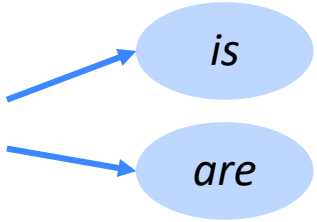
- **LM task:** *When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her* _____

- 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]

"Assessing the Ability of LSTMs to Learn Syntax-Sensitive Dependencies", Linzen et al, 2016. https://arxiv.org/pdf/1611.01368.pdf

Why is exploding gradient a problem?

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

learning rate

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

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

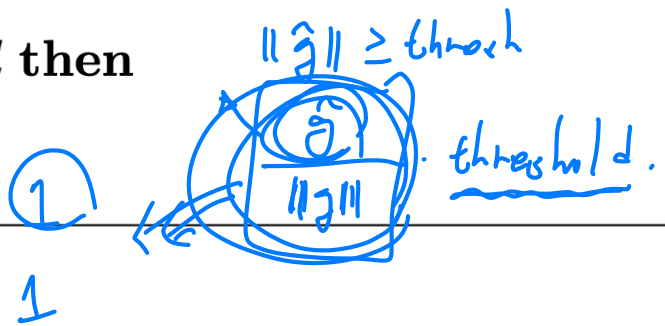
Algorithm 1 Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$

if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**

$\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$

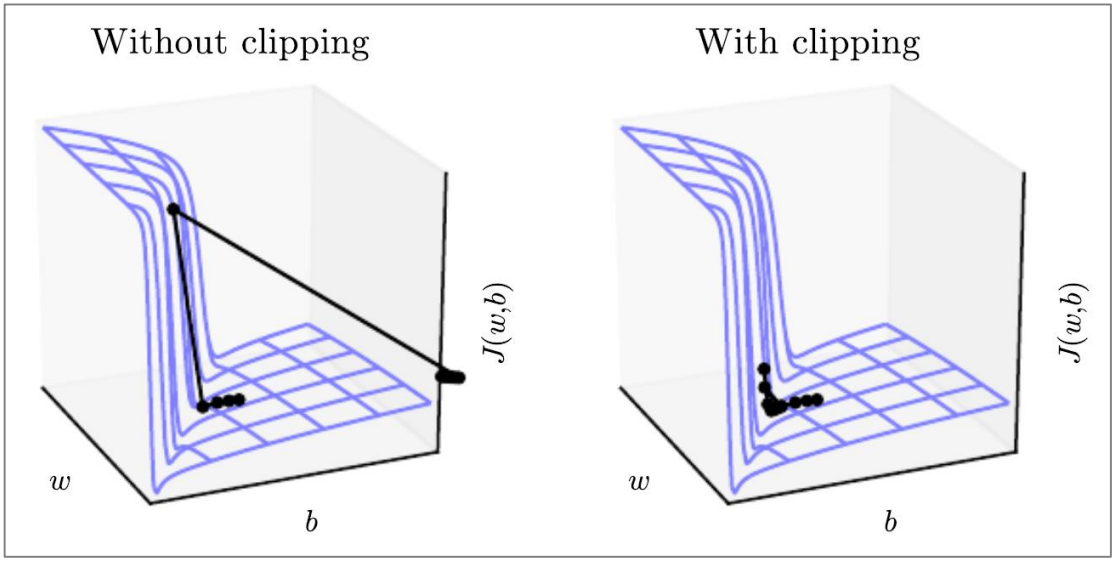
end if



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

Source: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013. http://proceedings.mlr.press/v28/pascanu13.pdf

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

Source: "Deep Learning", Goodfellow, Bengio and Courville, 2016. Chapter 10.11.1. https://www.deeplearningbook.org/contents/rnn.html

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

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

- 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 $\boldsymbol{h}^{(t)}$ *and* a cell state $\boldsymbol{c}^{(t)}$
  - 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", Hochreiter and Schmidhuber, 1997. https://www.bioinf.jku.at/publications/older/2604.pdf

Long Short-Term Memory (LSTM)

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

forget이면 $c^{(t-1)}$의 대해서 무엇인가 계산하려고 할까?

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

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

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

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

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

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

(-1, 1)

tanh의 gradient는 최대 1 (보통 0.25)

Gates are applied using element-wise product

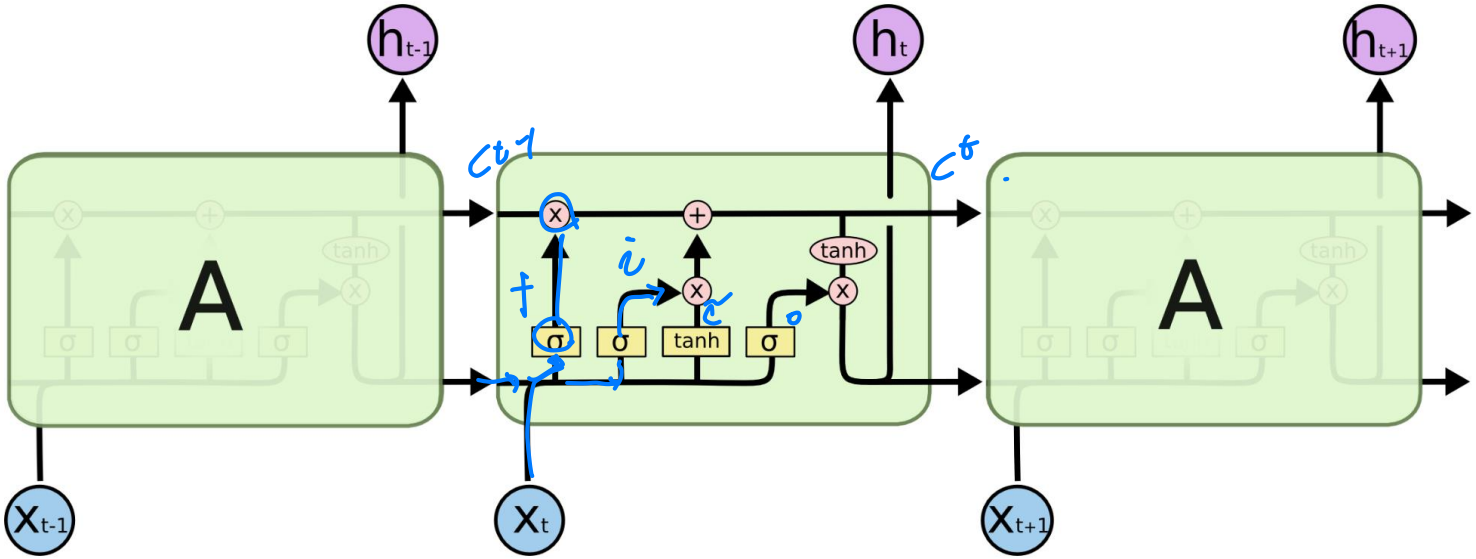
All these are vectors of same length n

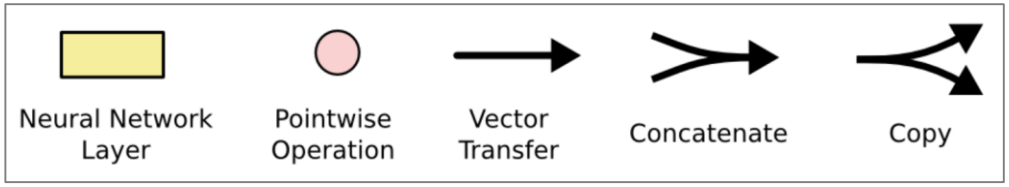
$$\frac{\partial C_t}{\partial C_{t-1}} = \text{diag}(f_t), \quad 0 \le f_t \le 1 \text{ also vanishing.}$$

$b_f = 1$ or 2.

Long Short-Term Memory (LSTM)

You can think of the LSTM equations visually like this:

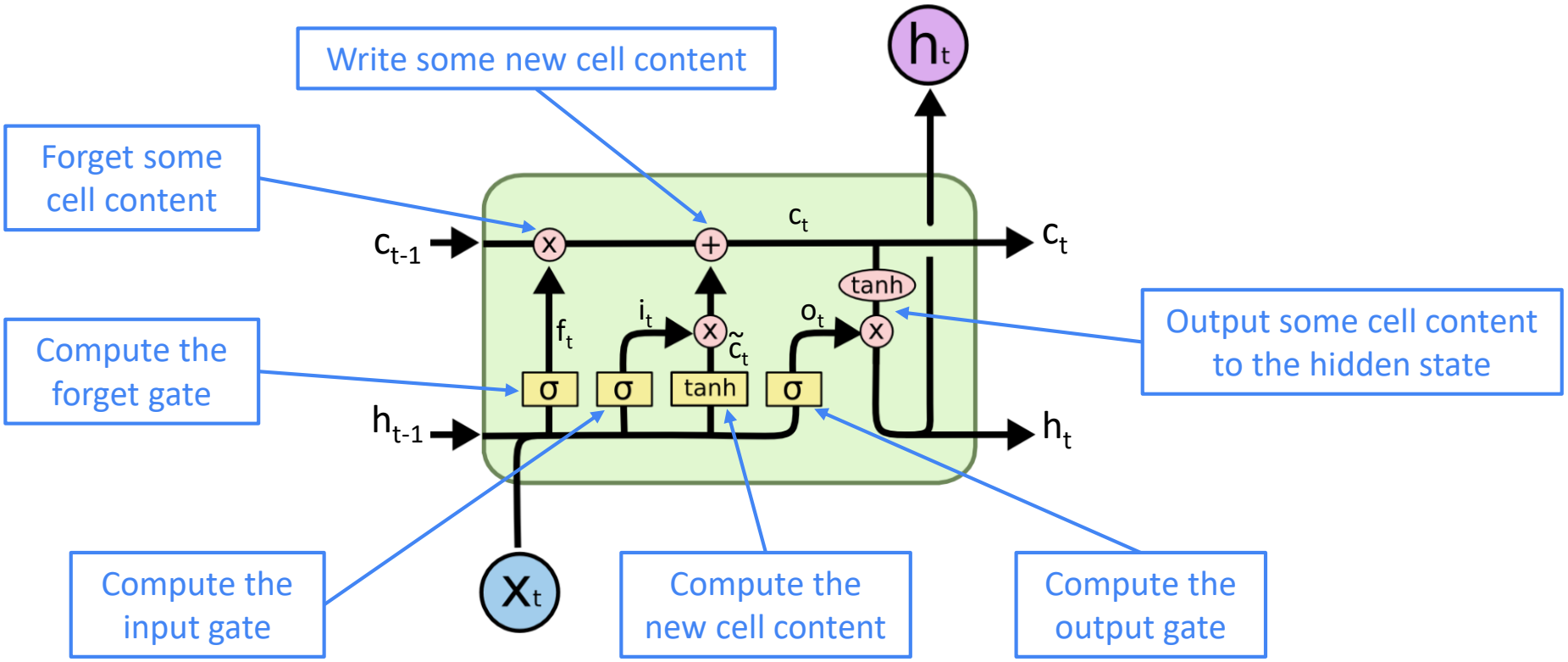


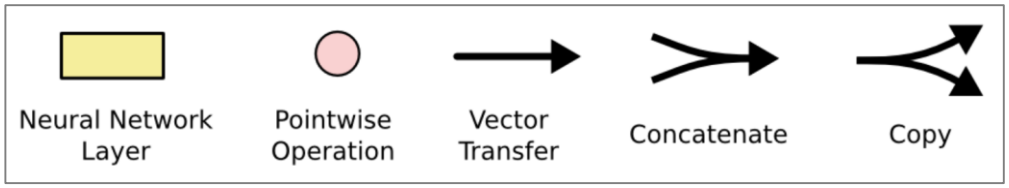


Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

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/

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

In WMT 2019, " 7 / 105 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 $\boldsymbol{x}^{(t)}$ and hidden state $\boldsymbol{h}^{(t)}$ (no cell state).

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

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

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

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

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

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

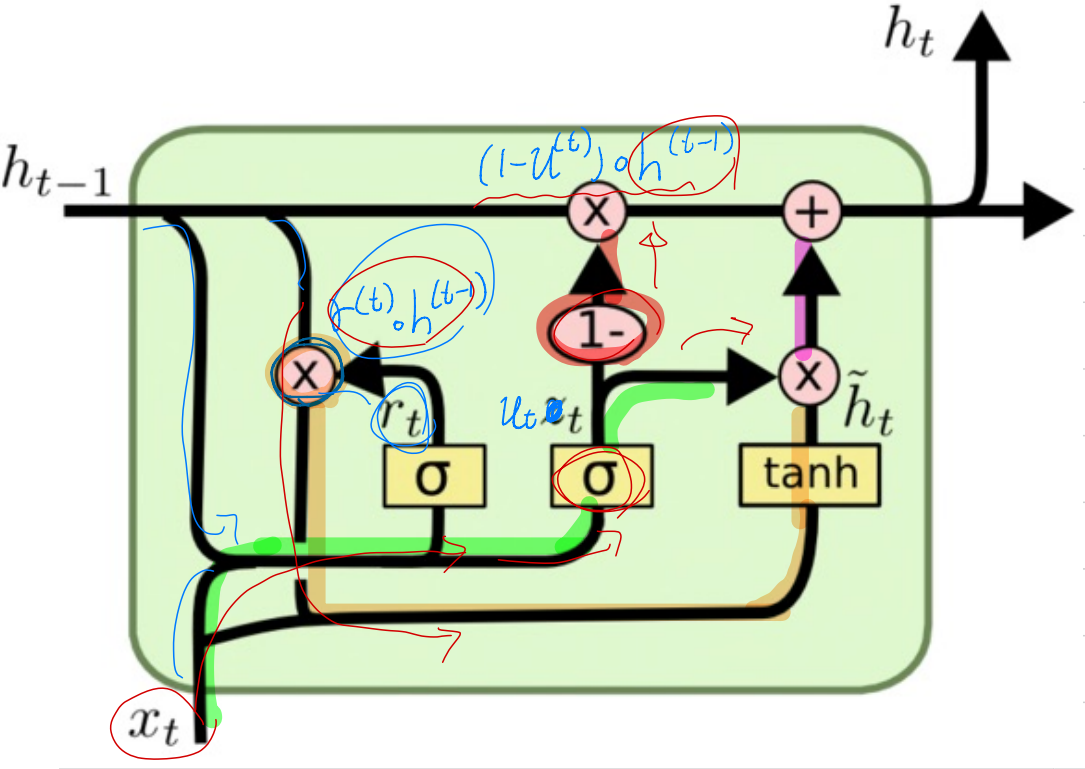
Hidden state: update gate simultaneously controls what is kept from previous hidden state, and what is updated to new hidden state content

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

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)

$h^{(t)} = (1-0) \cdot h^{(t-1)} + 0 \cdot \tilde{h}^{(t)}$

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



$$\tilde{h}^{(t)} = \tanh\left(W_h\left(r^{(t)} \circ h^{(t-1)}\right) + U_h x^{(t)} + b_h\right)$$

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

$$\begin{pmatrix} r^{(t)} \\ u^{(t)} \end{pmatrix} = W h^{(t-1)} + U x^{(t)} + b$$

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 성능이 바쁘하면 GRU

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)

deep NN performs worse: difficult to learn

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

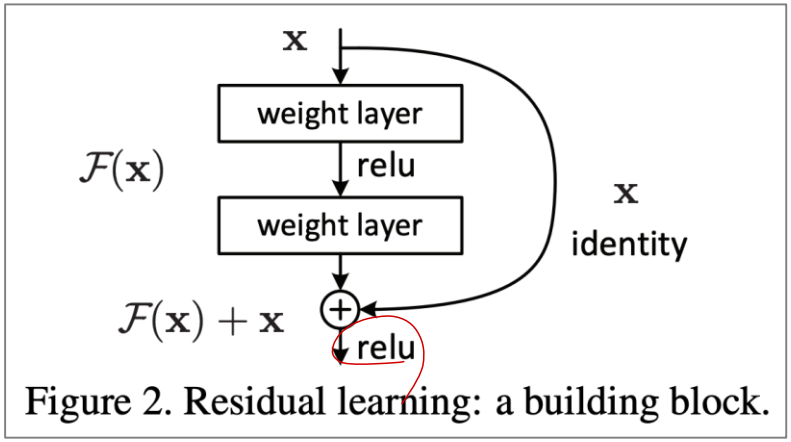


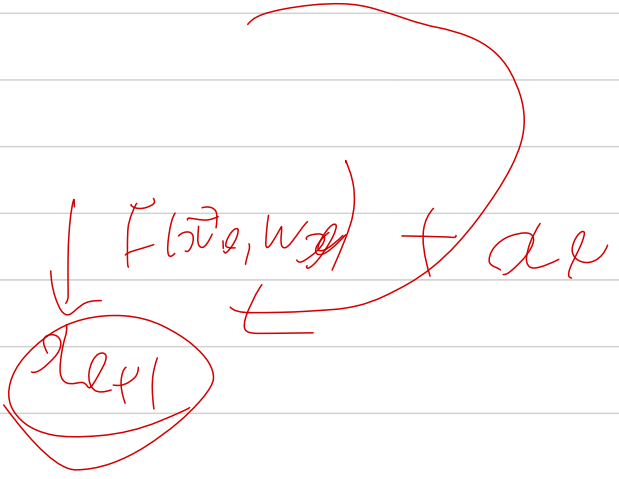
Figure 2. Residual learning: a building block.

"Deep Residual Learning for Image Recognition", He et al, 2015. https://arxiv.org/pdf/1512.03385.pdf

$$\vec{x}_{l+1} = f(\vec{y}_l)$$

$$\vec{y}_l = h(\vec{x}_l) + F(\vec{x}_l, W_l)$$

$$\therefore \quad \vec{x}_{l+1} = f\left(h(\vec{x}_l) + F(\vec{x}_l, W_l)\right) = \vec{x}_l + F(\vec{x}_l, W_l)$$



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!

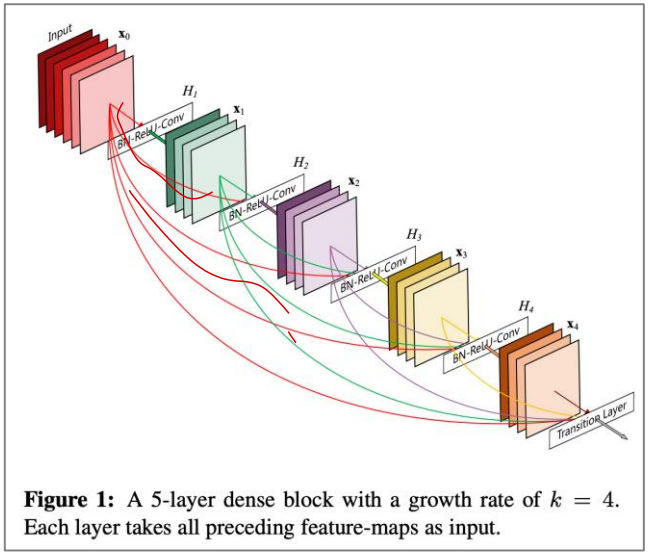


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

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

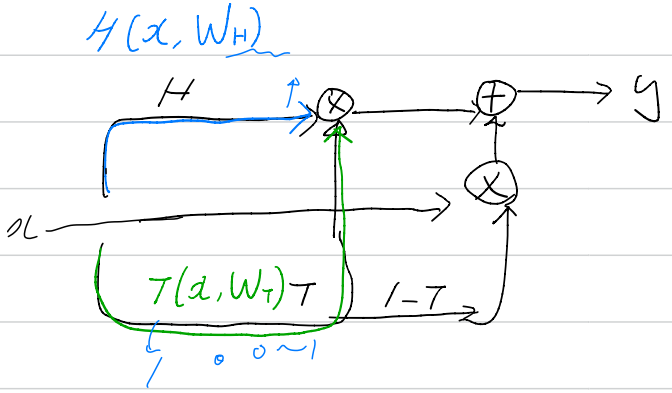
Is vanishing/exploding gradient just a RNN problem?

- 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



$$y = H(x, W_H) \cdot T(x, W_T) + x \cdot (1 - T(x, W_T))$$

where $T = \sigma$

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]

”Learning Long-Term Dependencies with Gradient Descent is Difficult", Bengio et al. 1994, http://ai.dinfo.unifi.it/paolo//ps/tnn-94-gradient.pdf

Recap

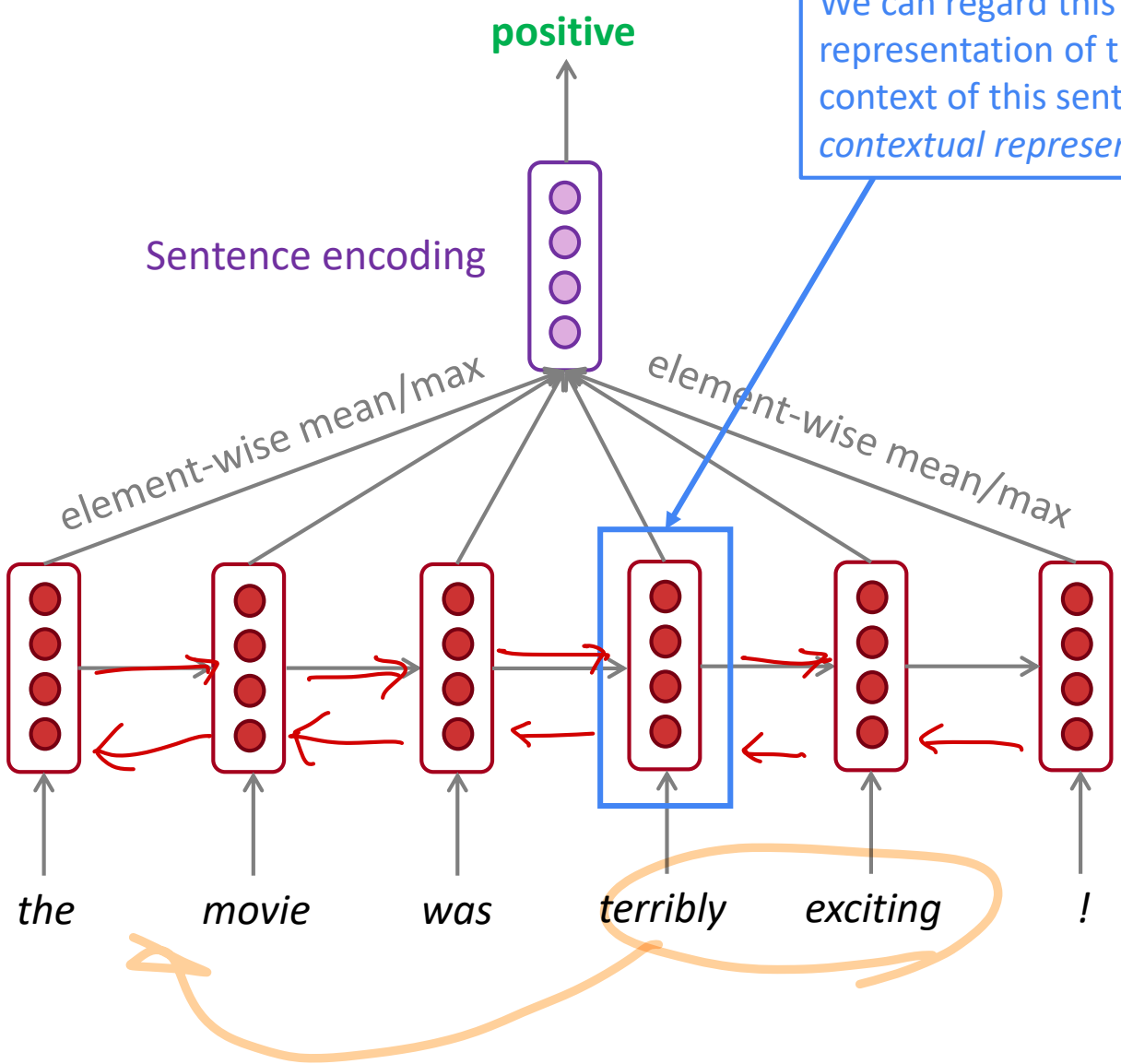
- Today we've learnt:
  - Vanishing gradient problem: what it is, why it happens, and why it's bad for RNNs
  - LSTMs and GRUs: more complicated RNNs that use gates to control information flow; they are more resilient to vanishing gradients

- Remainder of this lecture:
  - Bidirectional RNNs
  - Multi-layer RNNs

} Both of these are pretty simple

Bidirectional RNNs: motivation

Task: Sentiment Classification



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.*

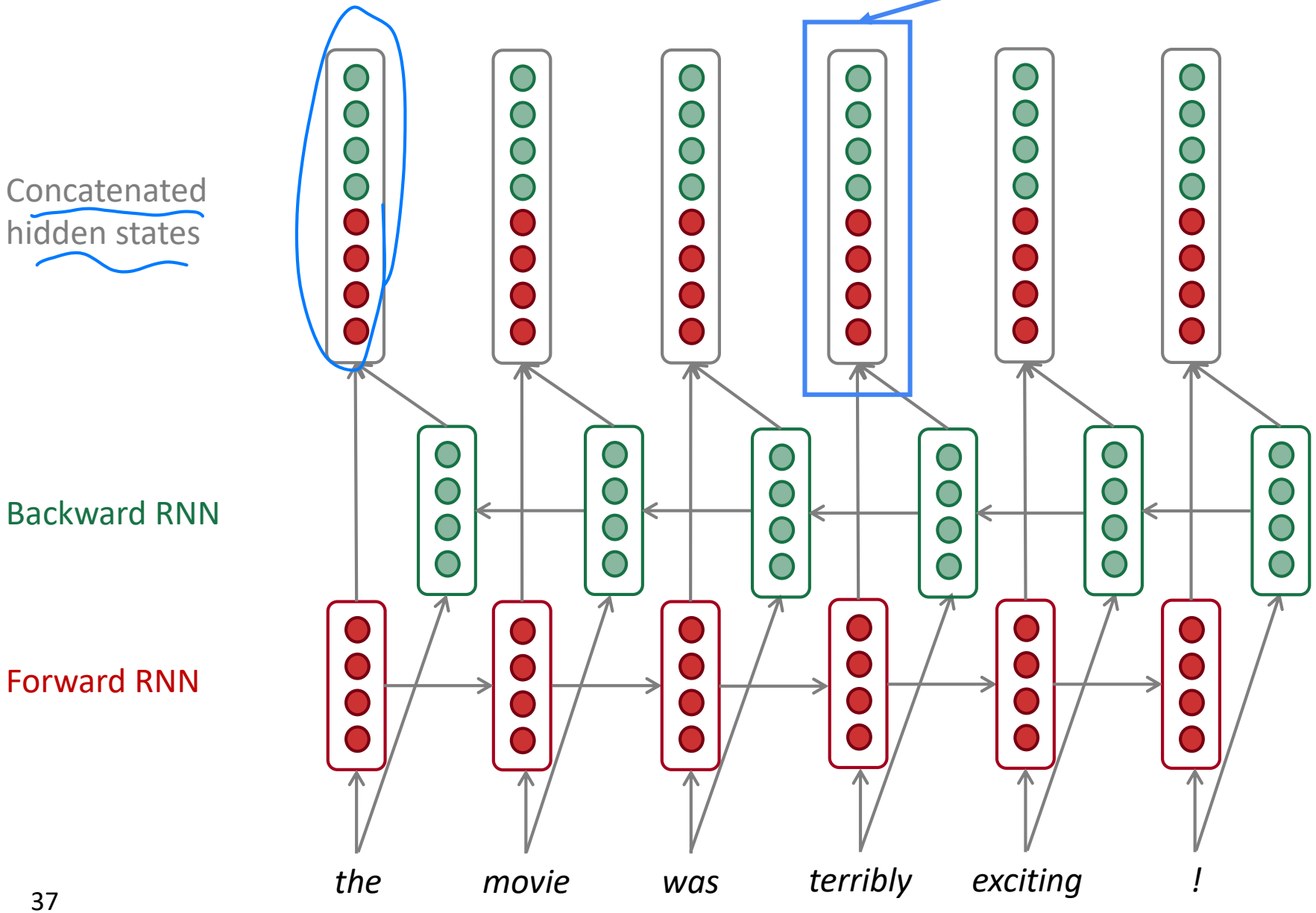
These contextual representations only contain information about the *left* context (e.g. *"the movie was"*).

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!



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

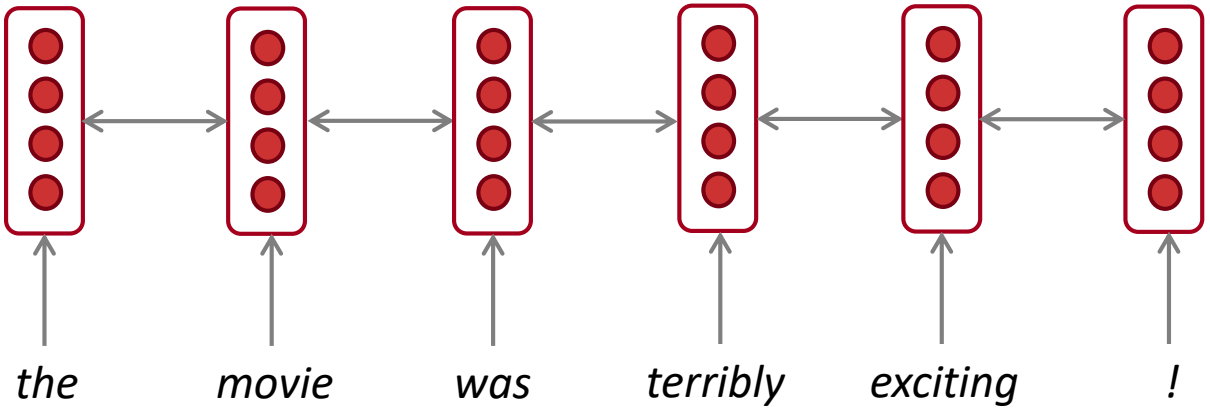
Backward RNN $\overleftarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{BW}}(\overleftarrow{\boldsymbol{h}}^{(t+1)}, \boldsymbol{x}^{(t)})$

Generally, these two RNNs have separate weights

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

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.

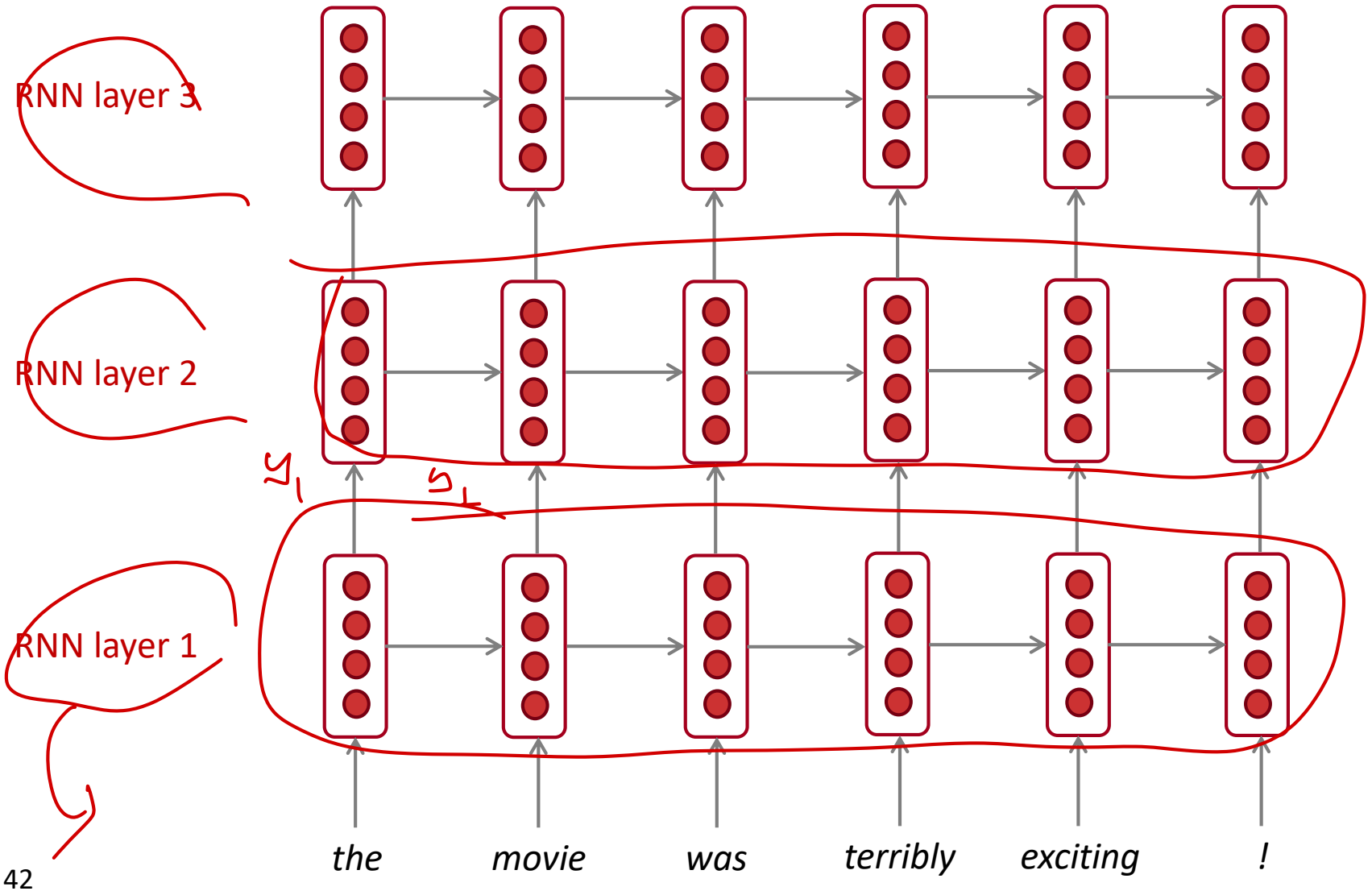
syntax

semantic

- 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) expensive to compute

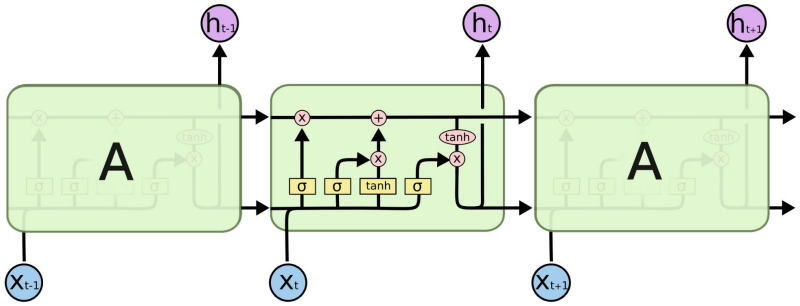
- For example: In a 2017 paper, Britz et al find that for Neural Machine Translation, 2 to 4 layers is best for the encoder RNN, and 4 layers is best for the decoder RNN
  - However, skip-connections/dense-connections are needed to train deeper RNNs (e.g. 8 layers)

- 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 ✓

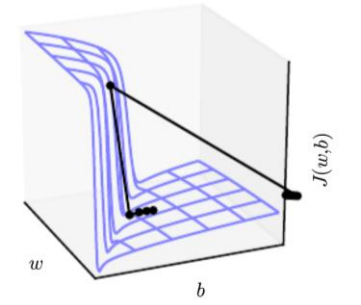
"Massive Exploration of Neural Machine Translation Architecutres", Britz et al, 2017. https://arxiv.org/pdf/1703.03906.pdf

In summary

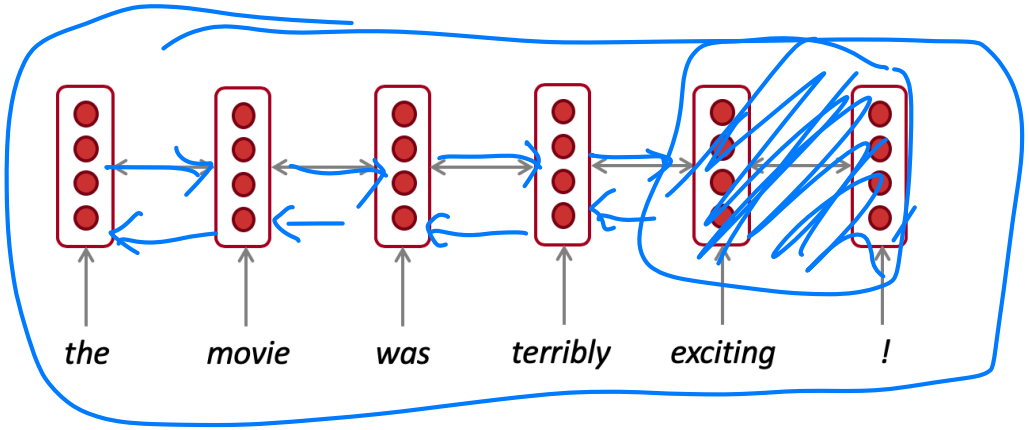
Lots of new information today! What are the practical takeaways?



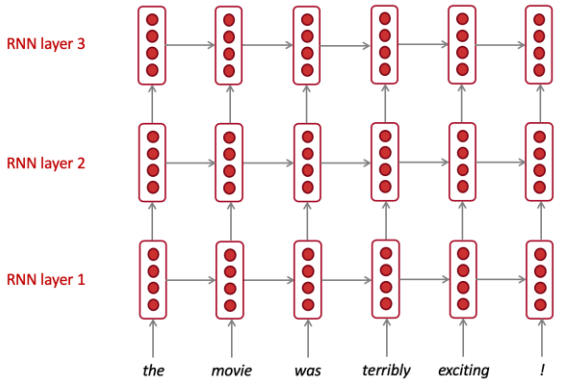
1. LSTMs are powerful but GRUs are faster



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