Carnegie Mellon University

Introduction to Deep Learning for Engineers

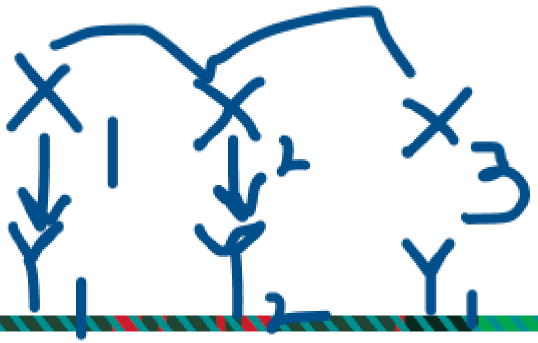
Spring 2025, Deep Learning for Engineers
Feb 18, 2025, Eleventh Session

Amir Barati Farimani
Associate Professor of Mechanical Engineering and Bio-Engineering
Carnegie Mellon University

Motivation behind RNN: Why NN is not enough!

Input feature vectors are sequentially dependent on each other

"The concert was boring for the first 15 minutes while the band warmed up but then was terribly exciting."



Motivation: Recurrent Neural Networks(RNN)

A machine learning model that considers the words in isolation — such as a bag of words model — would probably conclude this sentence is negative. An RNN by contrast should be able to see the words “but” and “terribly exciting” and realize that the sentence turns from negative to positive because it has looked at the entire sequence. Reading a whole sequence gives us a context for processing its meaning, a concept encoded in recurrent neural networks.

Carnegie Mellon University

Motivation: Recurrent Neural Networks(RNN)

Words have different meaning in different sentences/Context

He said, “Teddy bears are on sale!”’, and ‘He said, “Teddy Roosevelt was a great President!”’.

Carnegie Mellon University

Naive Bayes Intuition

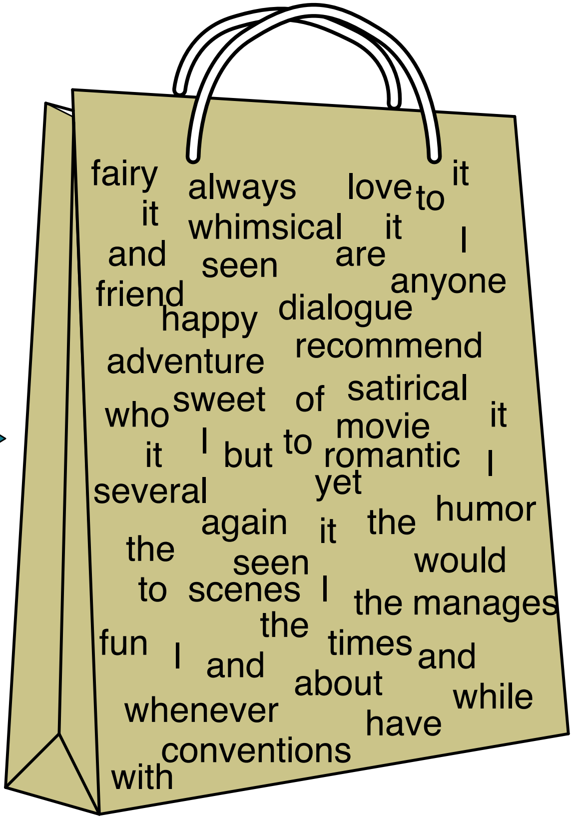
Simple ("naive") classification method based on Bayes rule

Relies on very simple representation of document

- **Bag of words**

The Bag of Words Representation

I love this movie! It's sweet, but with satirical humor. The dialogue is great and the adventure scenes are fun... It manages to be whimsical and romantic while laughing at the conventions of the fairy tale genre. I would recommend it to just about anyone. I've seen it several times, and I'm always happy to see it again whenever I have a friend who hasn't seen it yet!



it	6
I	5
the	4
to	3
and	3
seen	2
yet	1
would	1
whimsical	1
times	1
sweet	1
satirical	1
adventure	1
genre	1
fairy	1
humor	1
have	1
great	1
…	…

The bag of words representation

$\gamma($

seen	2
sweet	1
whimsical	1
recommend	1
happy	1
...	...

$)=c$

Bayes' Rule Applied to Documents and Classes

- FOR A DOCUMENT *D* AND A CLASS *C*

$$P(c \mid d) = \frac{P(d \mid c)P(c)}{P(d)}$$

Naive Bayes Classifier

$$c_{MAP} = \underset{c \in C}{\operatorname{argmax}} P(c \mid d)$$

MAP is "maximum a posteriori" = most likely class

$$= \underset{c \in C}{\operatorname{argmax}} \frac{P(d \mid c)P(c)}{P(d)}$$

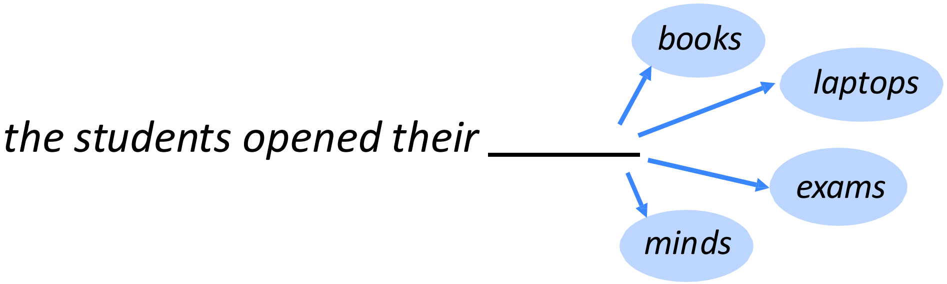
Bayes Rule

$$= \underset{c \in C}{\operatorname{argmax}} P(d \mid c)P(c)$$

Dropping the denominator

Language Modeling

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



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

$$P(\underline{\boldsymbol{x}^{(t+1)}} \mid \underbrace{\boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)}})$$

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

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

n-gram Language Models

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

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

prob of a n-gram

$$= \frac{\boxed{P(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}}{\boxed{P(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}}$$ (definition of conditional prob)

prob of a (n-1)-gram

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

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

Sparsity Problems with n-gram Language Models

Sparsity Problem 1

Problem: What if *"students opened their $\boldsymbol{w}$"* never occurred in data? Then $\boldsymbol{w}$ has probability 0!

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

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

Sparsity Problem 2

Problem: What if *"students opened their"* never occurred in data? Then we can't calculate probability for *any* $\boldsymbol{w}$!

(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

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

hidden layer

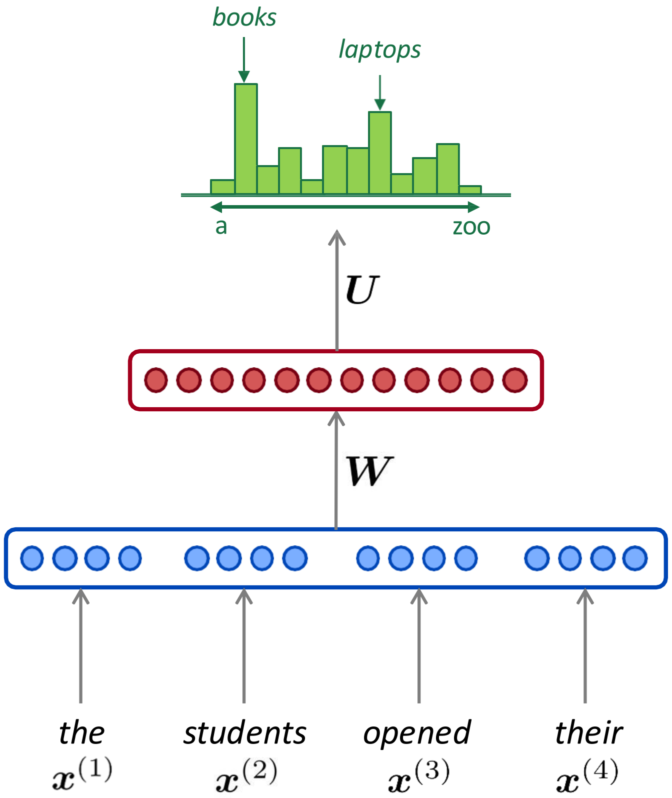
$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$

concatenated word embeddings

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

words / one-hot vectors

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



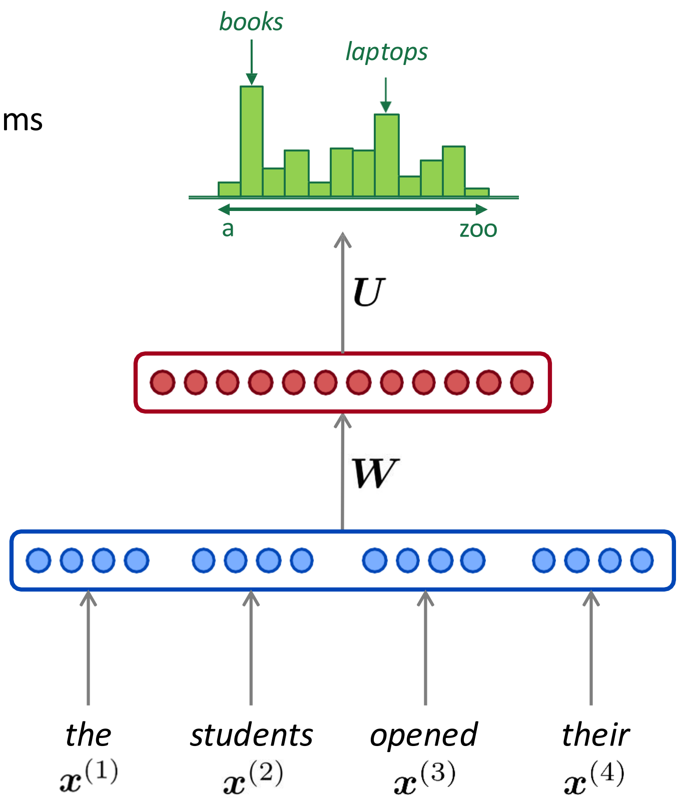
Carnegie Mellon University

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 $\boldsymbol{W}$
- Window can never be large enough!
- $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(2)}$ are multiplied by completely different weights in $\boldsymbol{W}$. No symmetry in how the inputs are processed.

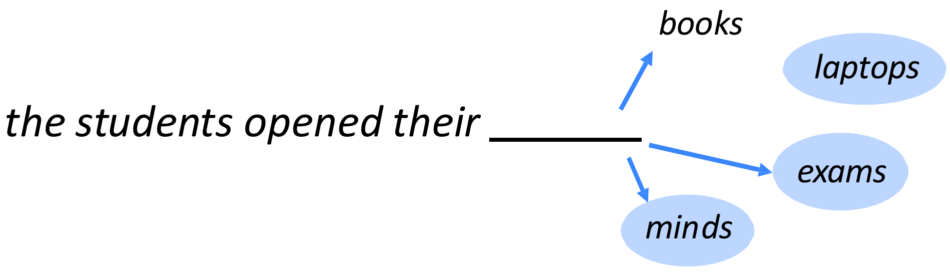
We need a neural architecture that can process *any length input*



Why not NN or CNN?

One issue with vanilla neural nets (and also CNNs) is that they only work with pre-determined sizes: they take fixed-size inputs and produce fixed-size outputs. RNNs are useful because they let us have variable-length sequences as both inputs and outputs.

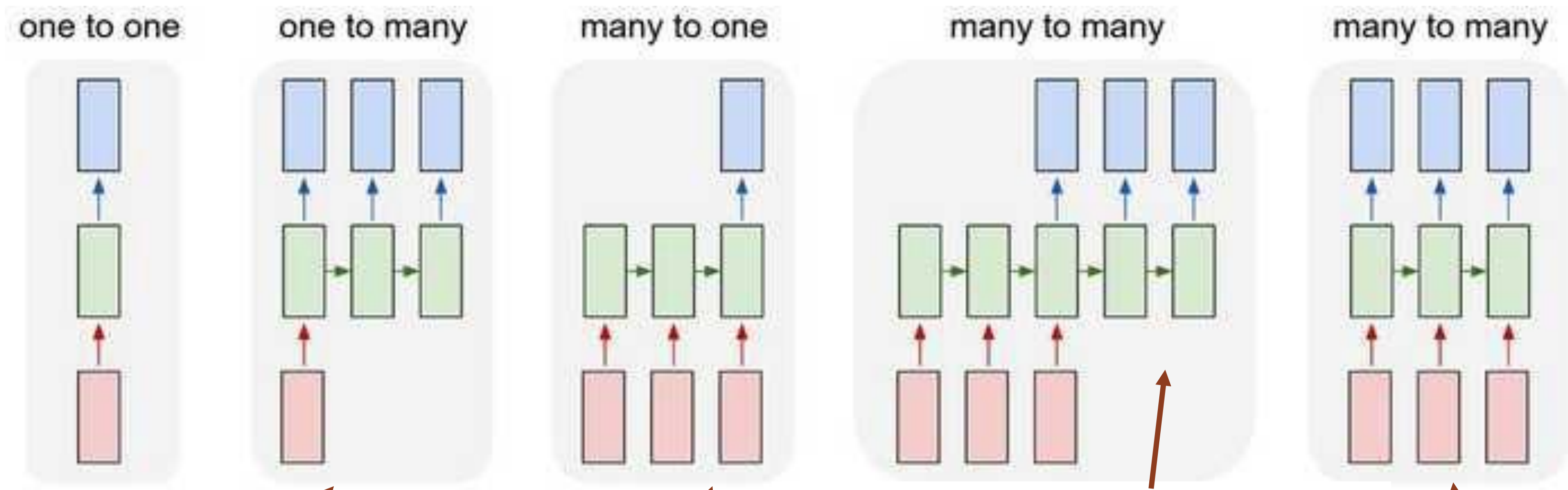
How can we model and learn sequential dependencies?



- **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.*

Carnegie Mellon University

Sequential models we need



e.g. **Machine Translation**
seq of words -> seq of words

e.g. **Image Captioning**
image -> sequence of words

e.g. **Sentiment Classification**
sequence of words -> sentiment

e.g. **Video classification on frame level**

CS231N lecture slides, Stanford

Examples of Sequences

•**Machine Translation** (e.g. Google Translate) is done with "many to many" RNNs. The original text sequence is fed into an RNN, which then produces translated text as output.
•**Sentiment Analysis** (e.g. *Is this a positive or negative review?*) is often done with "many to one" RNNs. The text to be analyzed is fed into an RNN, which then produces a single output classification (e.g. *This is a positive review*).

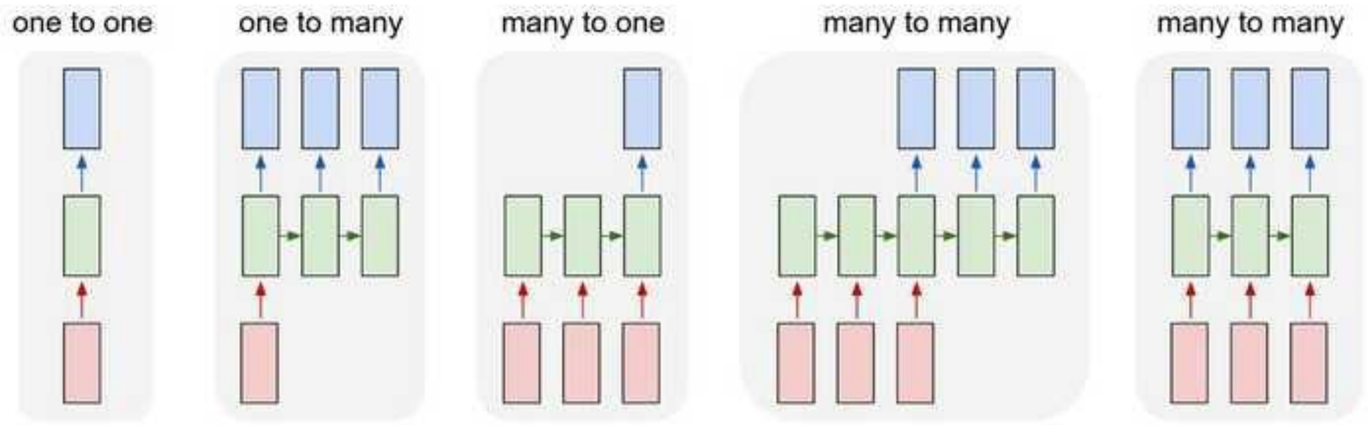


Image Captioning: Sequence on the outputs

"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."

"a young boy is holding a baseball bat."

"a cat is sitting on a couch with a remote control."

"a woman holding a teddy bear in front of a mirror."

"a horse is standing in the middle of a road."

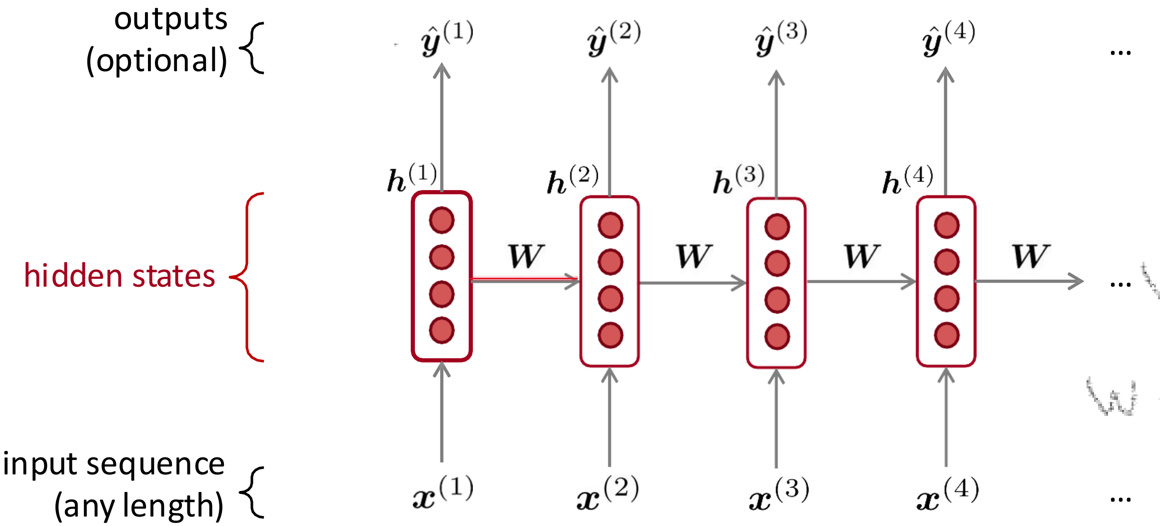
Carnegie Mellon University

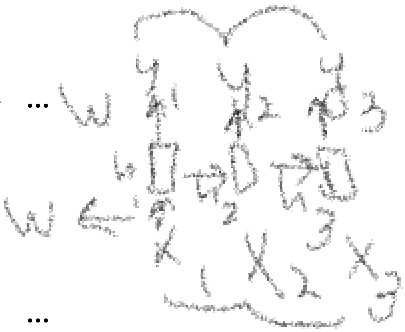
Andrej Karpathy

Recurrent Neural Networks (RNN)

A family of neural architectures

Core idea: Apply the same weights $\boldsymbol{W}$ *repeatedly*





A RNN Language Model

output distribution

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

hidden states

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

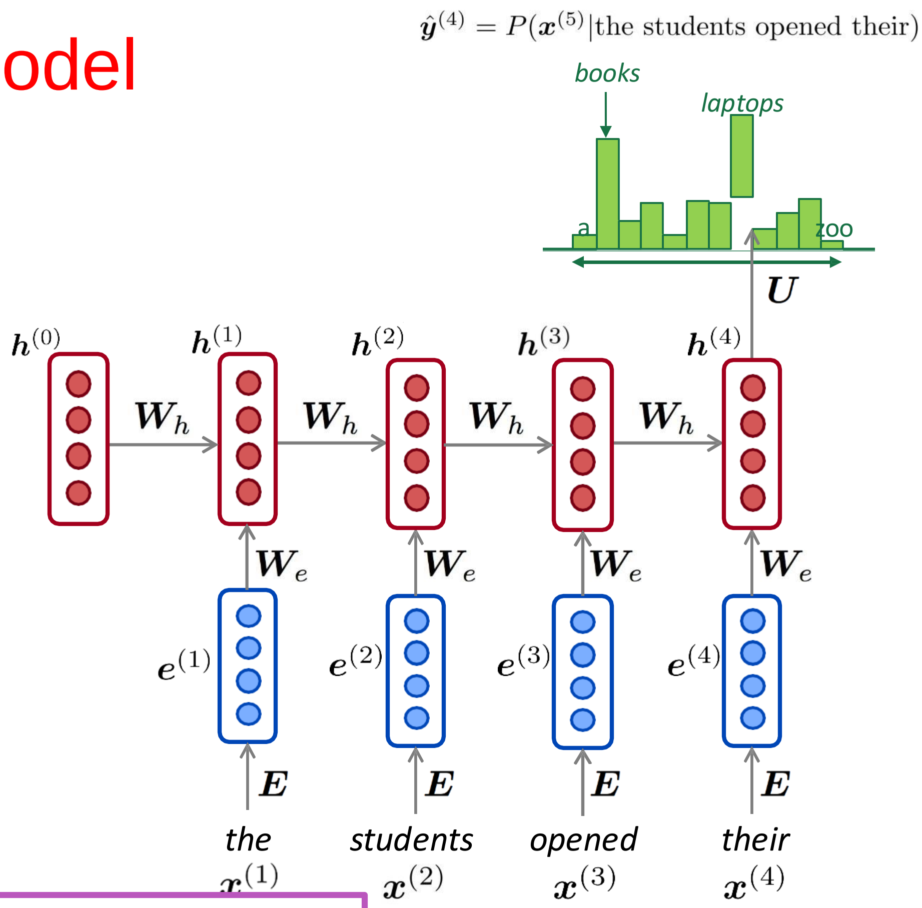
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

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

words / one-hot vectors

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

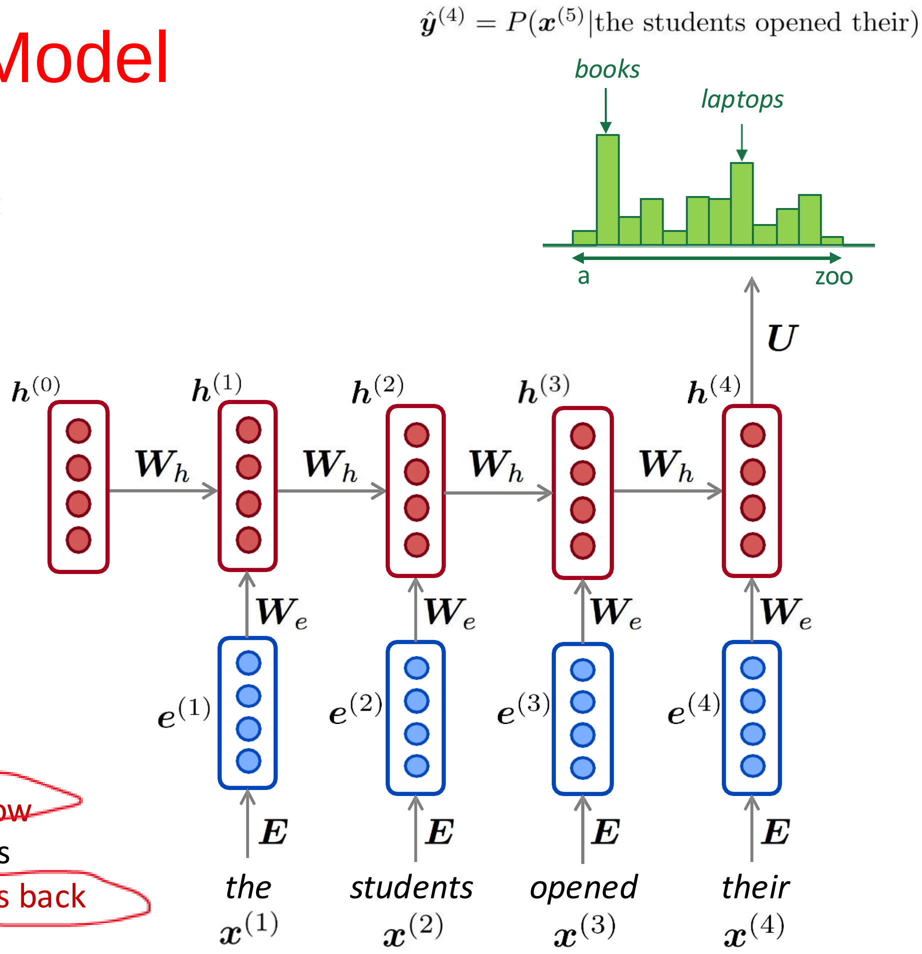


***Note**: this input sequence could be much longer, but this slide doesn't have space!*

A RNN Language Model

RNN **Advantages**:

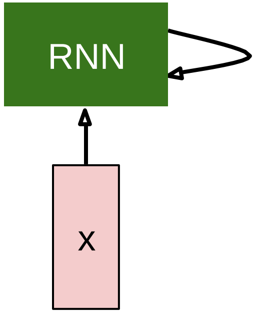
- Can process any length input
- Computation for step *t* can (in theory) use information from many steps back
- Model size doesn't increase for longer input
- Same weights applied on every timestep, so there is symmetry in how inputs are processed



RNN **Disadvantages**:

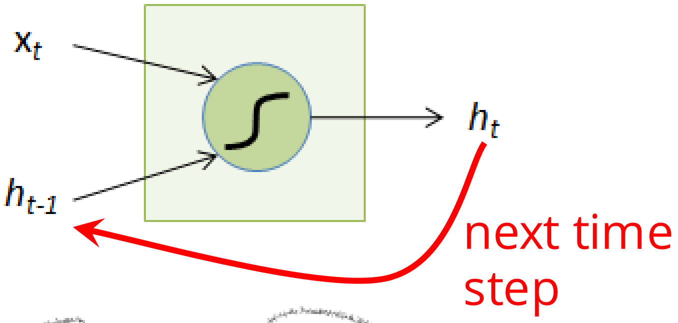
- Recurrent computation is slow
- In practice, difficult to access information from many steps back

Recurrent Neural Network



The Recurrent Neuron

- x_t: Input at time t
- h_{t-1}: State at time t-1



$$h_t = f(W_h h_{t-1} + W_x x_t)$$

h_2 h_1

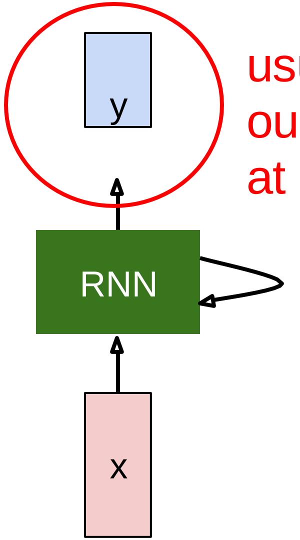
Carnegie Mellon University

Recurrent Neural Networks(RNN)

Recurrent means the output at the current time step becomes the input to the next time step. At each element of the sequence, the model considers not just the current input, but what it remembers about the preceding elements.

Carnegie Mellon University

Recurrent Neural Network



usually want to output a prediction at some time steps

Adapted from Andrej Karpathy

Recurrent Neural Network

We can process a sequence of vectors **x** by applying a recurrence formula at every time step:

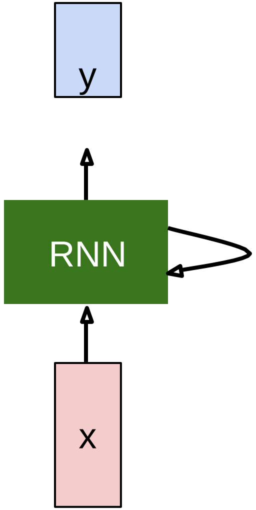
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

old state

input vector at some time step

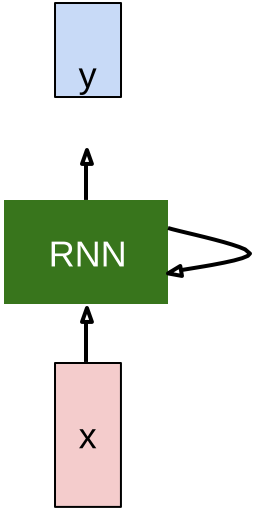
some function with parameters W



Recurrent Neural Network

We can process a sequence of vectors **x** by applying a recurrence formula at every time step:

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

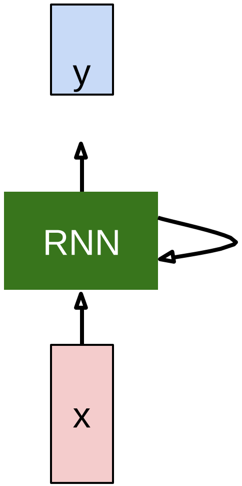


Notice: the same function and the same set of parameters are used at every time step.

Andrej Karpathy

(Vanilla) Recurrent Neural Network

The state consists of a single *"hidden"* vector **h**:



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

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

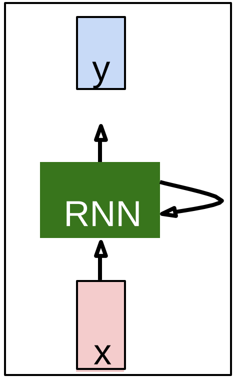
$$y_t = W_{hy}h_t$$

Example

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”

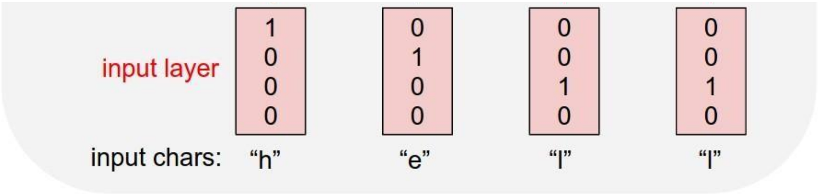


Example

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”



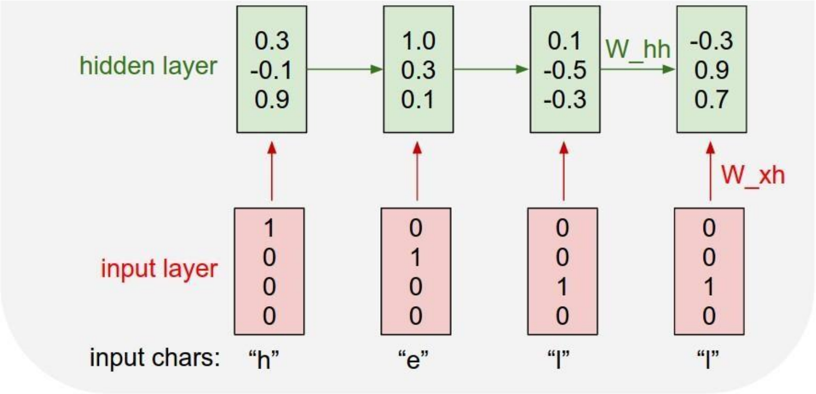
Example

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”

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



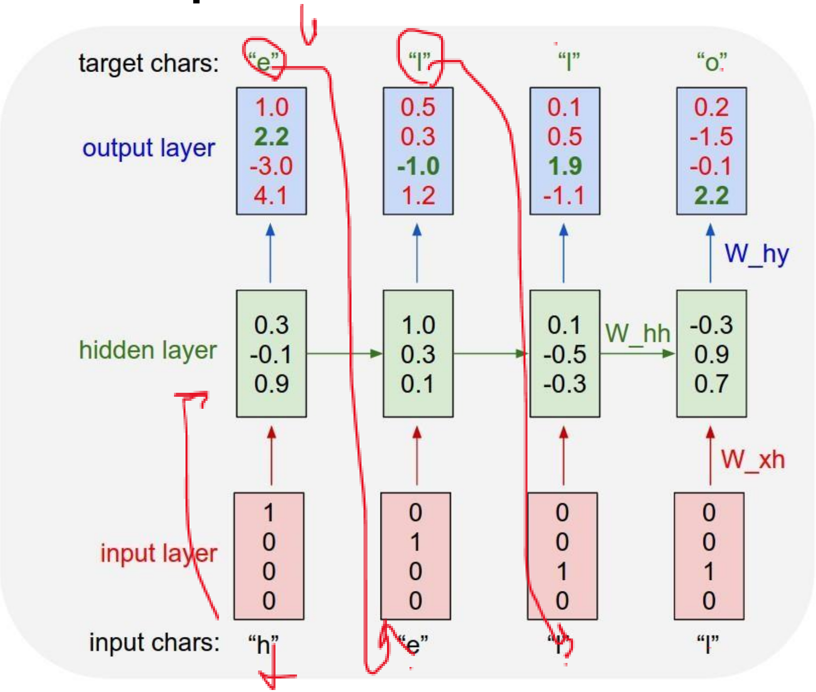
Carnegie Mellon University

Example

Character-level language model example

Vocabulary: [h,e,l,o]

Example training sequence: **“hello”**



What do we still need to specify, for this to work?

What kind of loss can we formulate?

Carnegie Mellon University

Andrej Karpathy

Motivation: Recurrent Neural Networks(RNN)

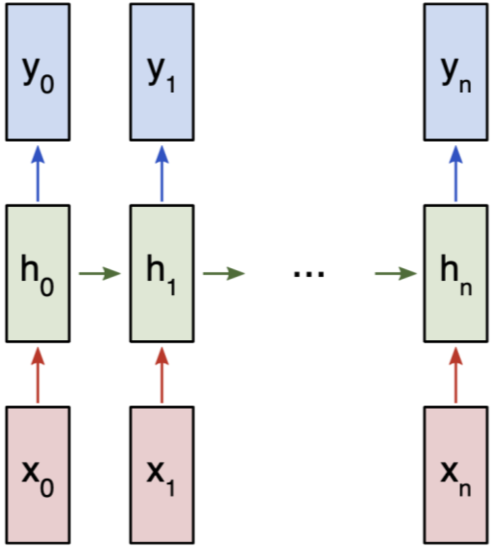
- Language Modelling and Generating Text
- Machine Translation
- Speech Recognition
- Generating Image Descriptions
- Video Tagging
- Text Summarization
- Call Center Analysis
- Face detection, OCR Applications as Image Recognition
- Other applications like Music composition

RNN Sequence Types

RNNs work by iteratively updating a hidden state h, which is a vector that can also have arbitrary dimension. At any given step t,

1.The next hidden state h_t is calculated using the previous hidden state h_{t-1} and the next input x_t.

2.The next output y_t is calculated using h_t.

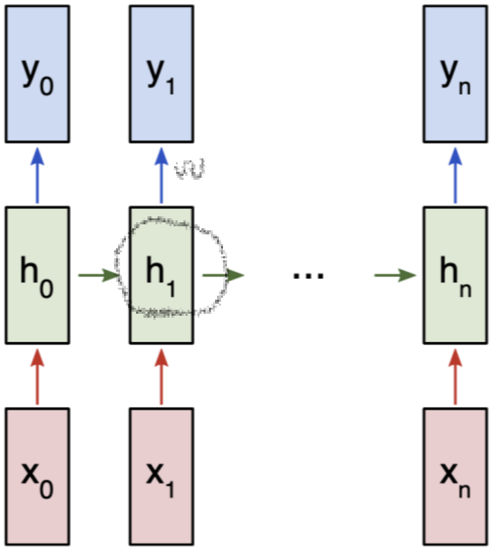


•*Wxh*, used for all x_t → h_t links.
•*Whh*, used for all h_{t-1} → h_t links.
•*Why*, used for all h_t → y_t links.
We'll also use two biases for our RNN:
•b_h, added when calculating h_t.
•b_y, added when calculating y_t.

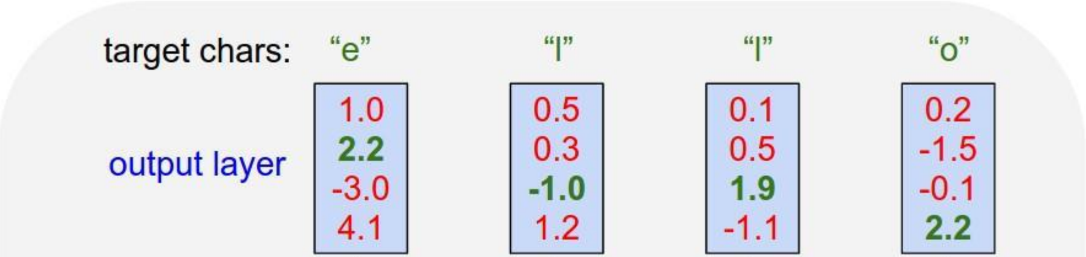
Weights

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

$$y_t = W_{hy}h_t + b_y$$



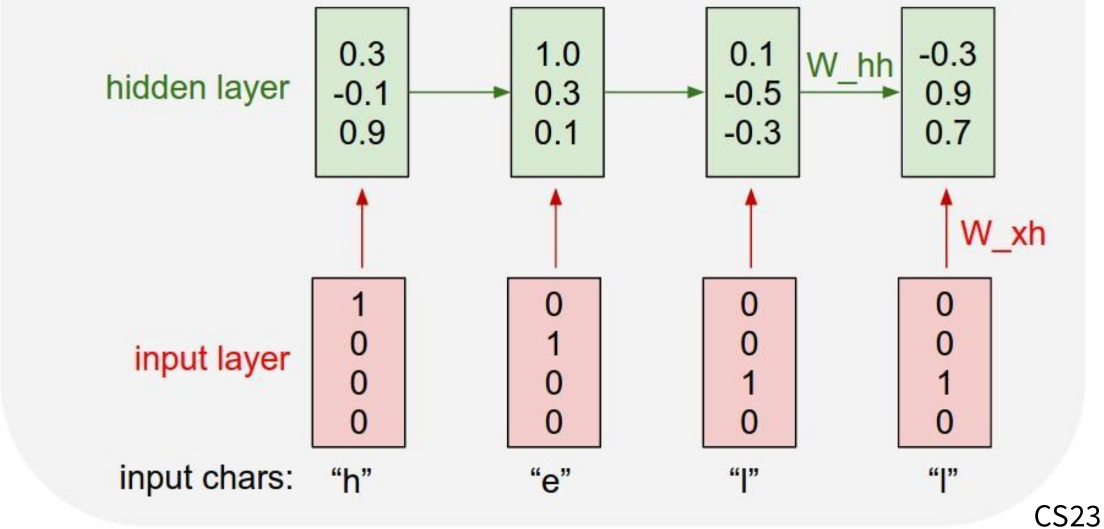
RNN explained with an example: Hello



CS231N lecture slides, Stanford

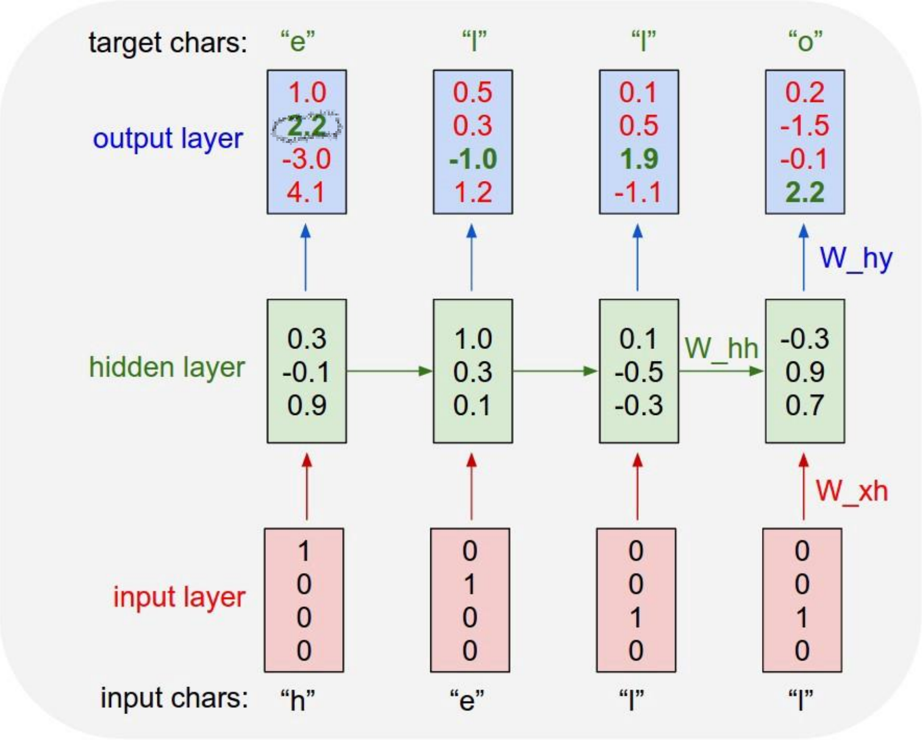
RNN explained with an example: Hello

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



CS231N lecture slides

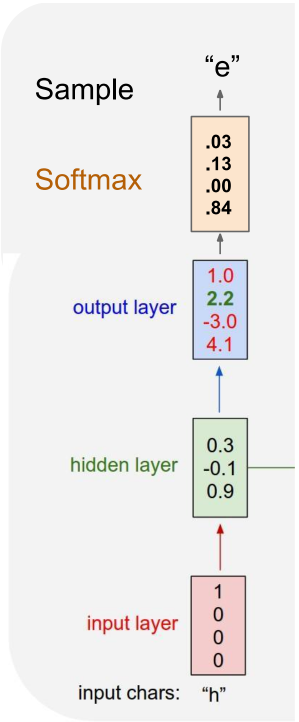
RNN explained with an example: Hello

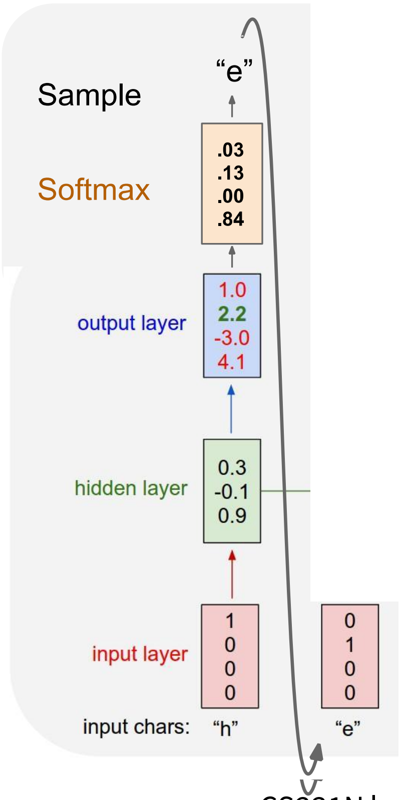


CS231N lecture slides

Carnegie Mellon University

RNN explained with an example: Hello





CS231N lecture slides, Stanford

Training a RNN Language Model

- Get a big corpus of text which is a sequence of words $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$
- Feed into RNN-LM; compute output distribution $\hat{\boldsymbol{y}}^{(t)}$ *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 $\hat{\boldsymbol{y}}^{(t)}$, and the true next word $\boldsymbol{y}^{(t)}$ (one-hot for $\boldsymbol{x}^{(t+1)}$):

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

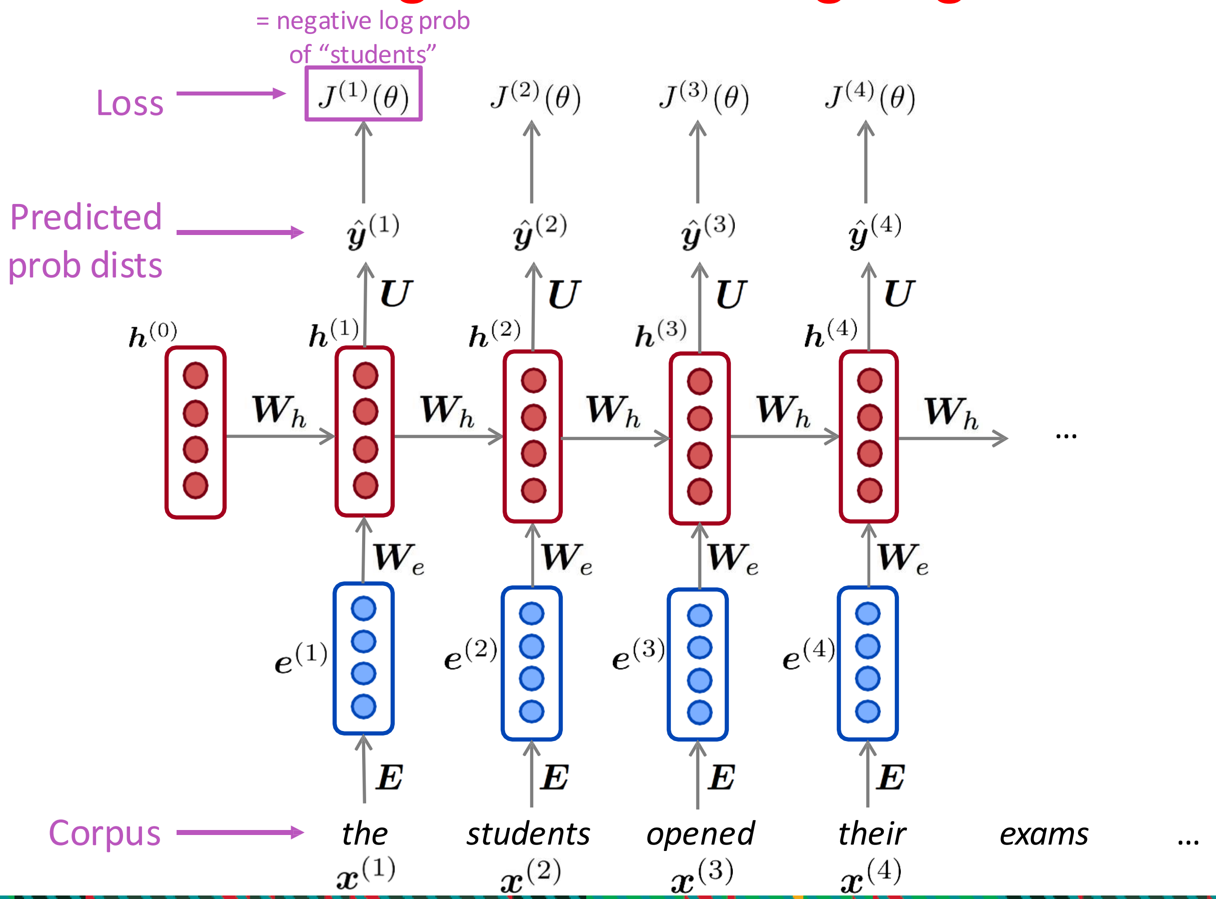
am a ? 1st

I am a ___

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

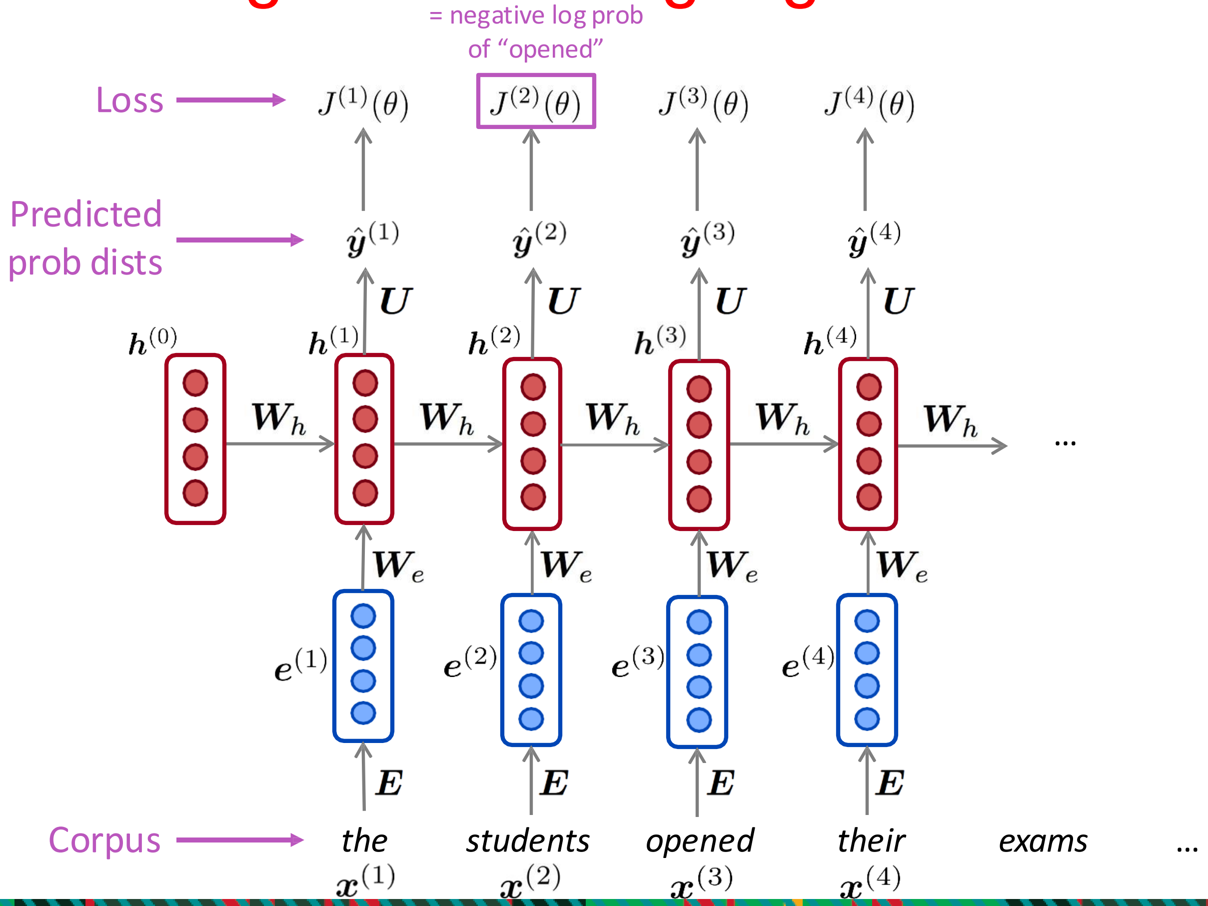
$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T} \sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

Training a RNN Language Model

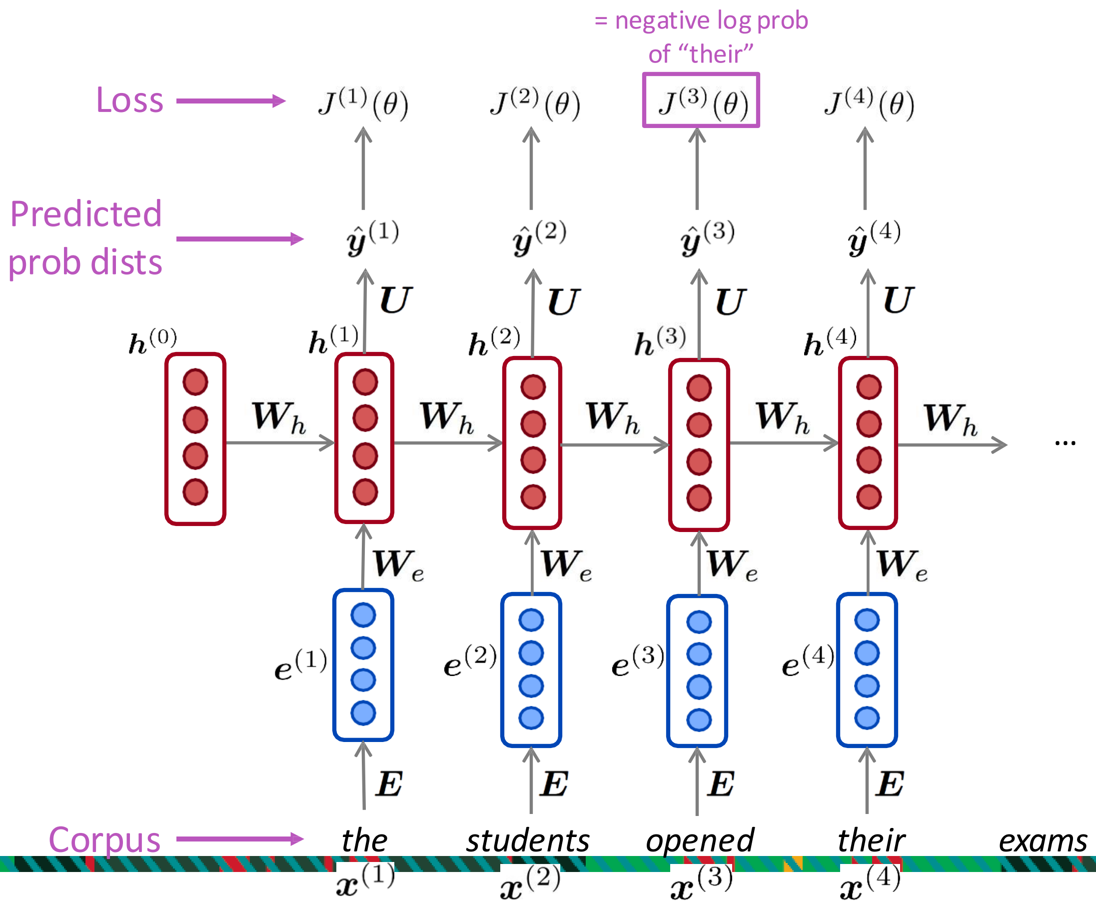


Carnegie Mellon University

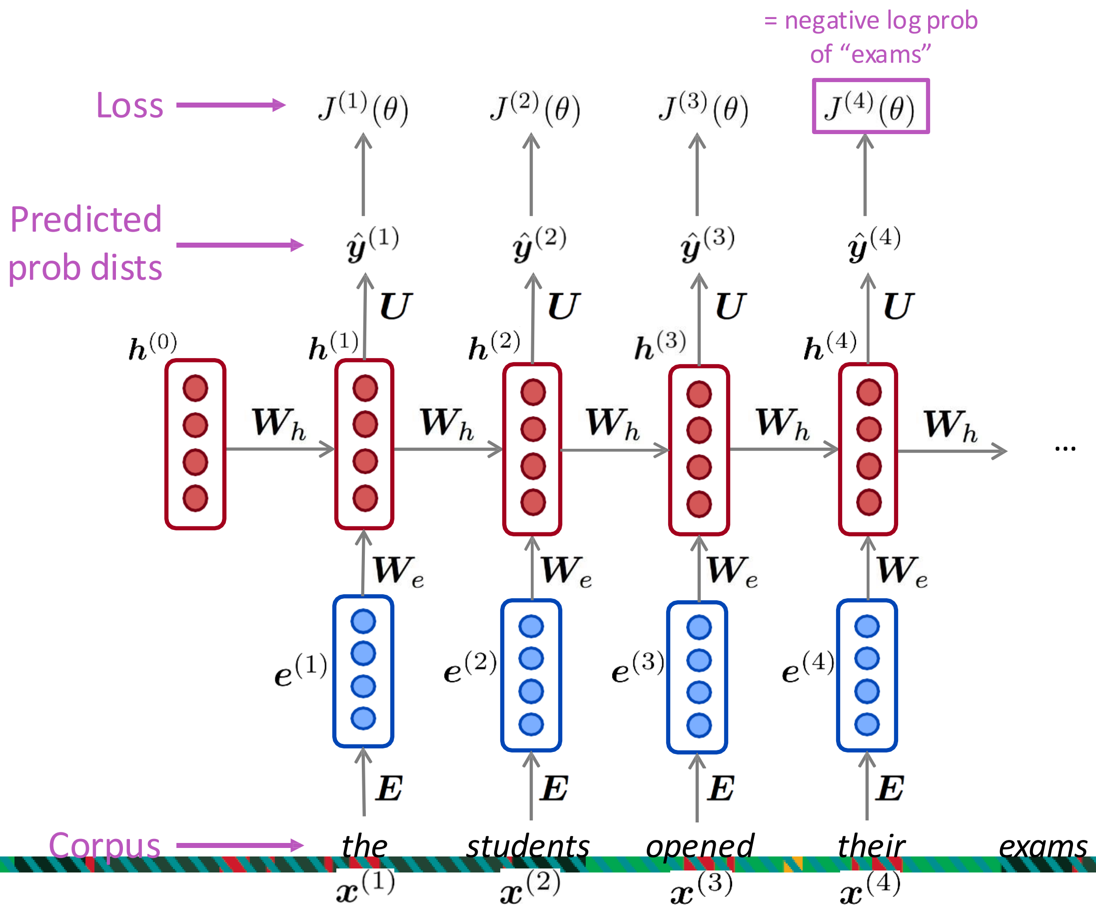
Training a RNN Language Model



Training a RNN Language Model



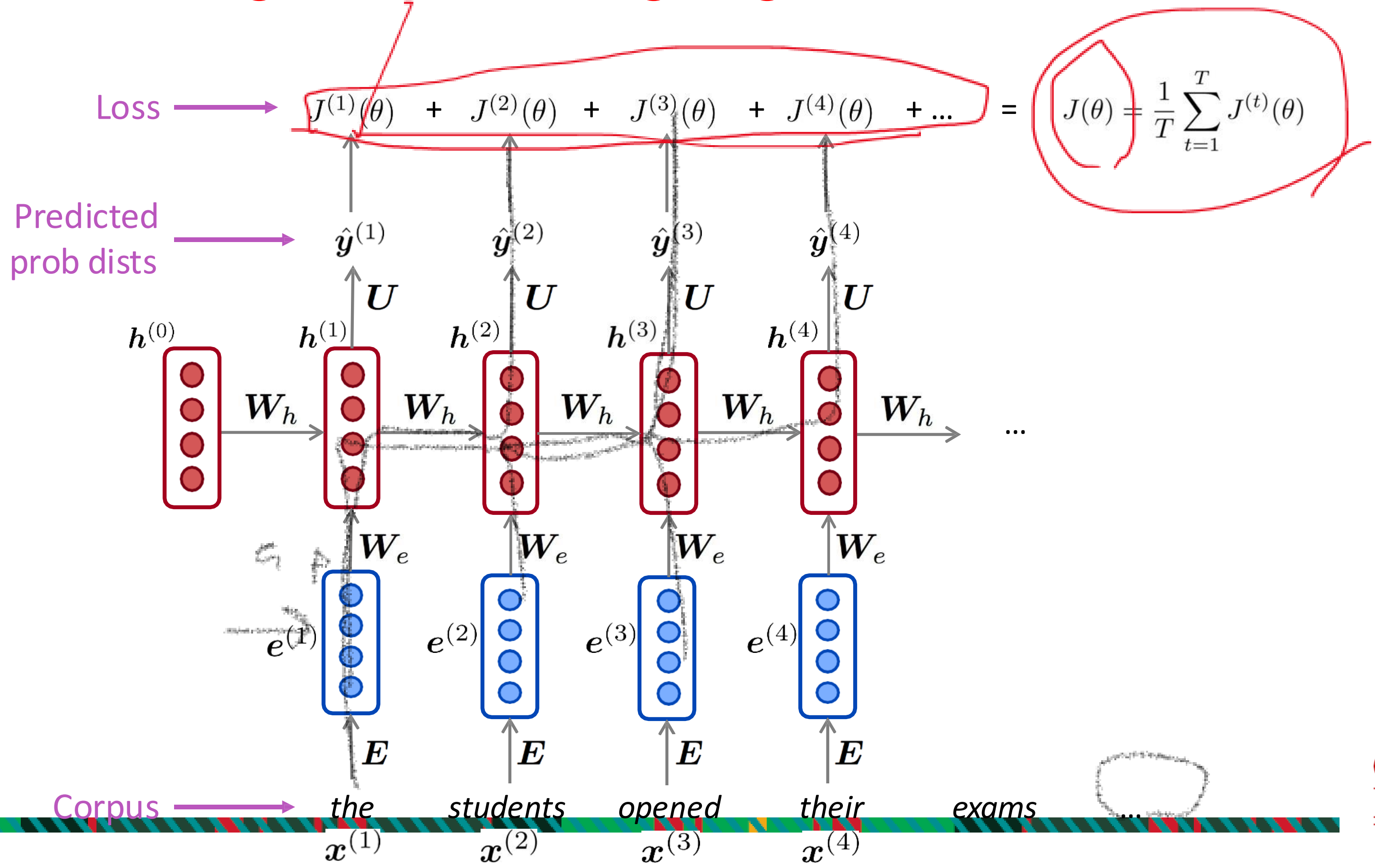
Training a RNN Language Model



Carnegie Mellon University

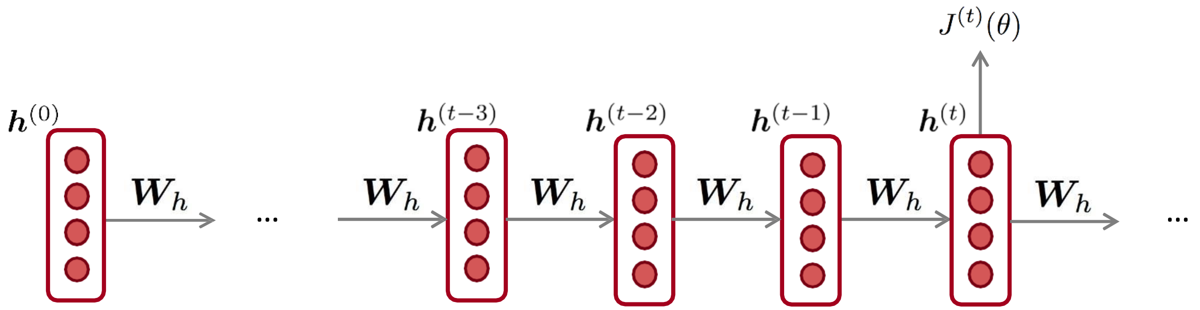
Abigail See

Training a RNN Language Model



Carnegie Mellon University

Backpropagation for RNNs

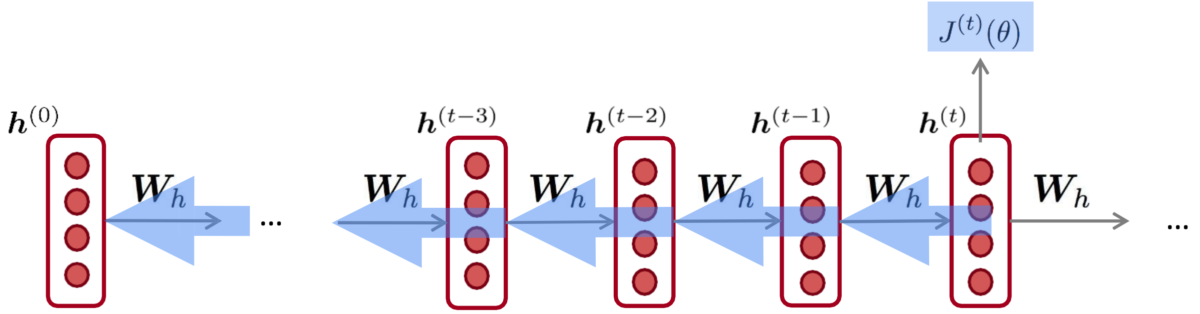


Question: What's the derivative of $J^{(t)}(\theta)$ w.r.t. the repeated weight matrix W_h ?

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

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

Backpropagation for RNNs



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

Question: How do we calculate this?

Answer: Backpropagate over timesteps *i=t*,…,0, summing gradients as you go.
This algorithm is called **"backpropagation through time"**

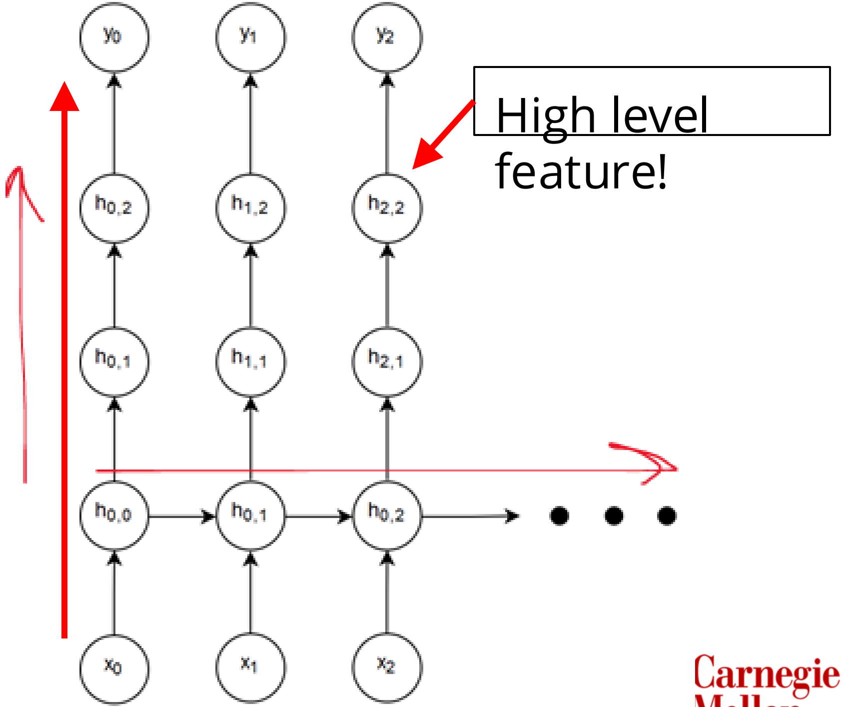
Carnegie Mellon University

Option 1: Feedforward Depth (d_f)

Notation: $h_{0,1}$ ⇒ time step 0, neuron #1

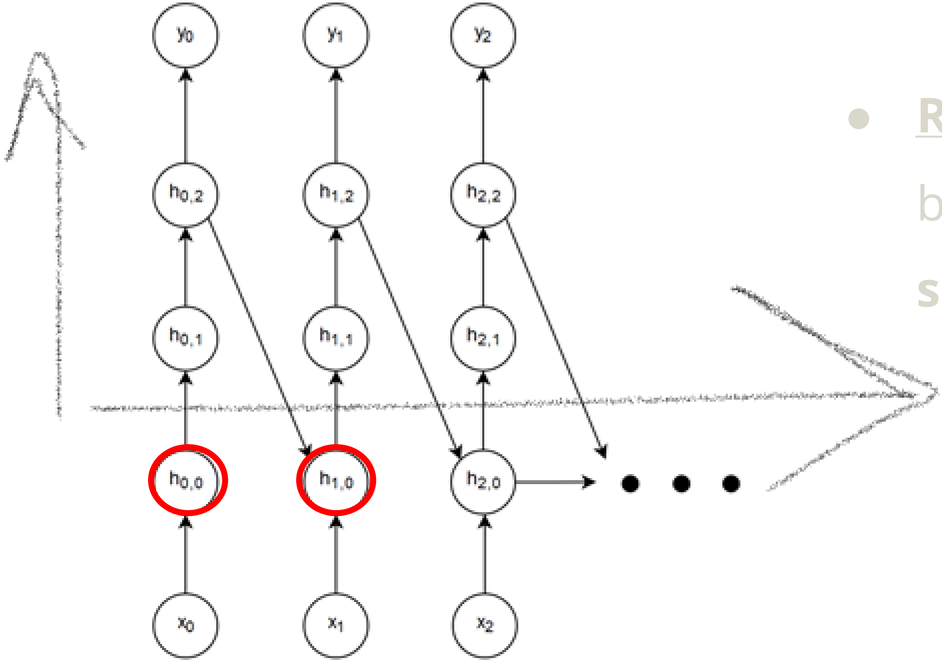
<u>FEEDFORWARD DEPTH</u>: **LONGEST** PATH BETWEEN AN INPUT AND OUTPUT AT THE **SAME TIMESTEP**

Feedforward depth = 4



Carnegie Mellon University

Option 2: Recurrent Depth (d_r)

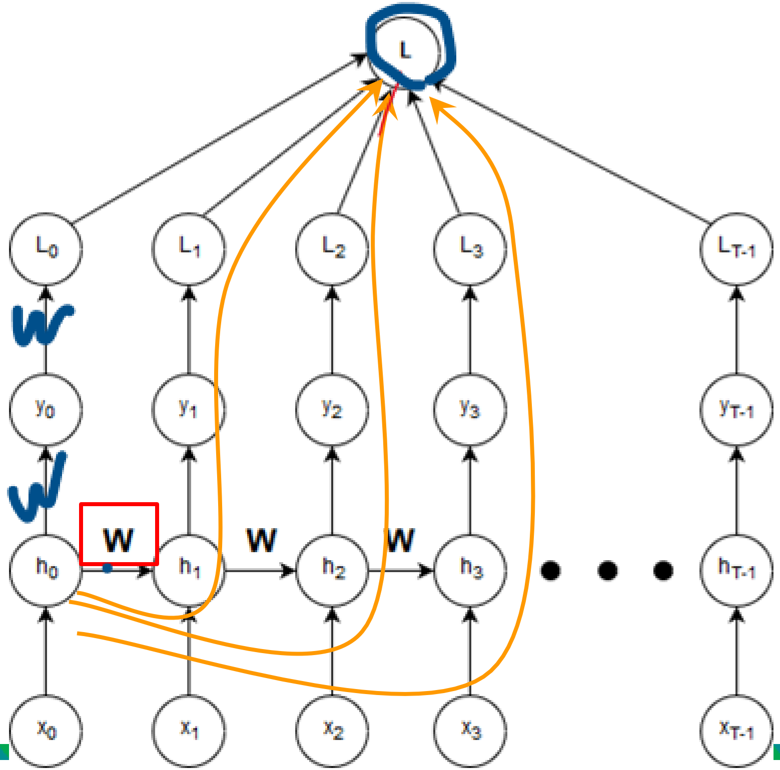


Recurrent depth = 3

- **Recurrent depth**: Longest path between **same hidden state** in **successive timesteps**

Carnegie Mellon University

Backpropagation Through Time (BPTT)



OBJECTIVE IS TO UPDATE THE WEIGHT MATRIX:

$$\mathbf{W} \to \mathbf{W} - \alpha \frac{\partial L}{\partial \mathbf{W}}$$

ISSUE: **W** OCCURS EACH TIMESTEP

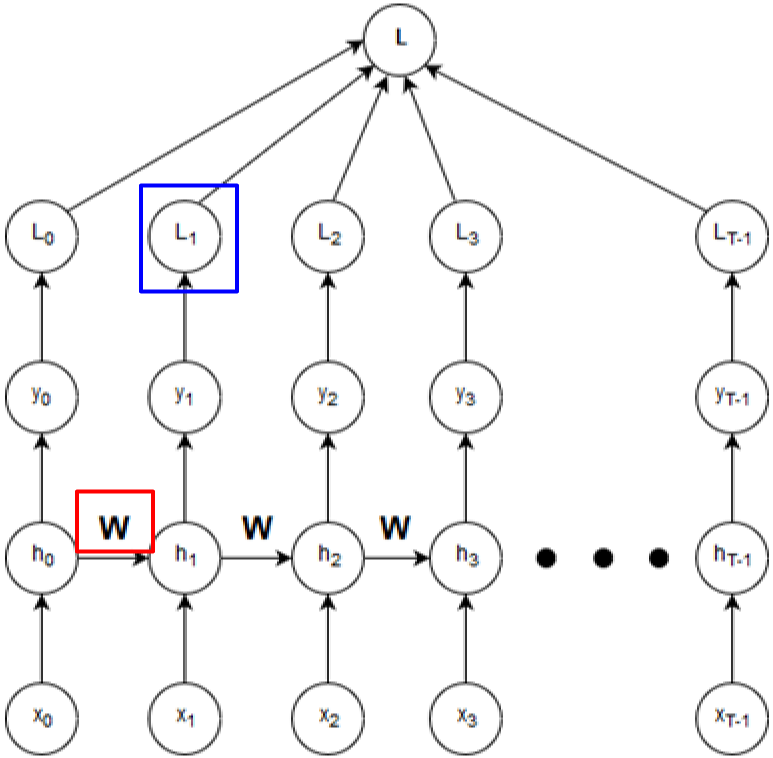
EVERY PATH FROM **W** TO L IS ONE DEPENDENCY

FIND ALL PATHS FROM **W** TO L!

(note: dropping subscript h from $\mathbf{W_h}$ for brevity)

Carnegie Mellon University

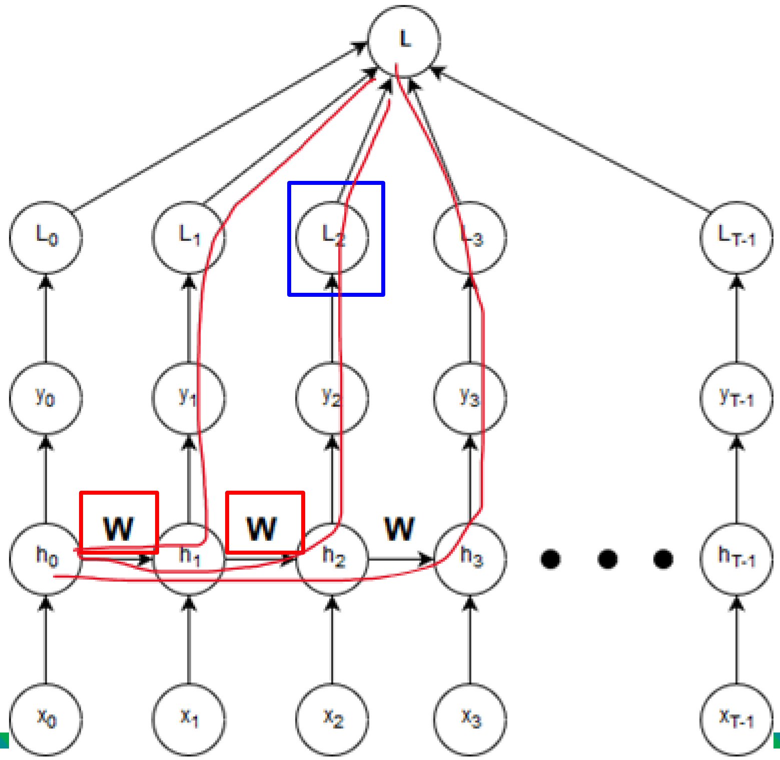
Systematically Finding All Paths



How many paths exist from W to L *through* L_1?

Just 1. Originating at h_0.

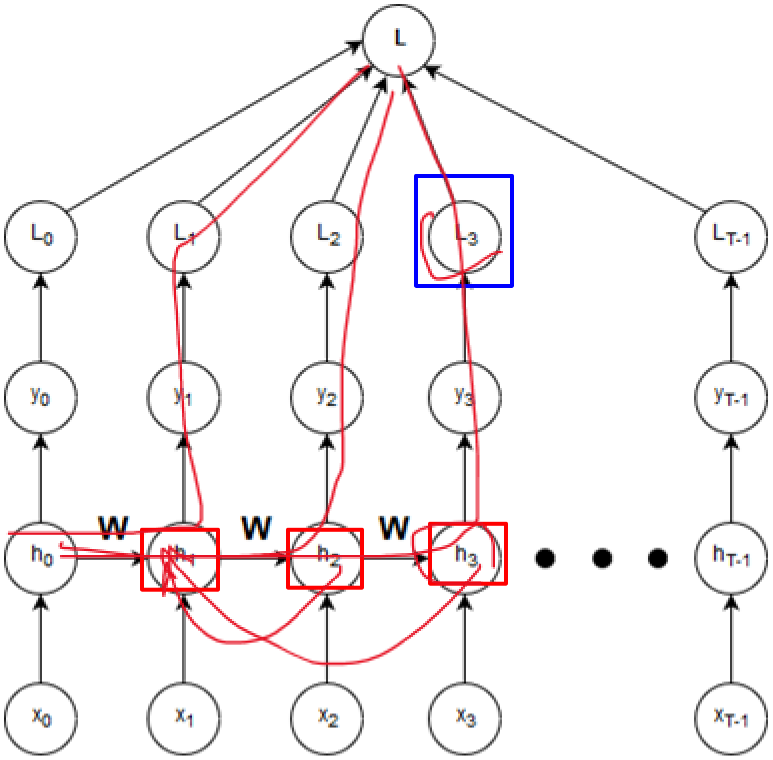
Systematically Finding All Paths



How many paths from **W** to L *through L_2*?

2. Originating at h_0 and h_1.

Systematically Finding All Paths



And 3 in this case.

Origin of path = basis for Σ

$$\frac{\partial L}{\partial \mathbf{W}}$$

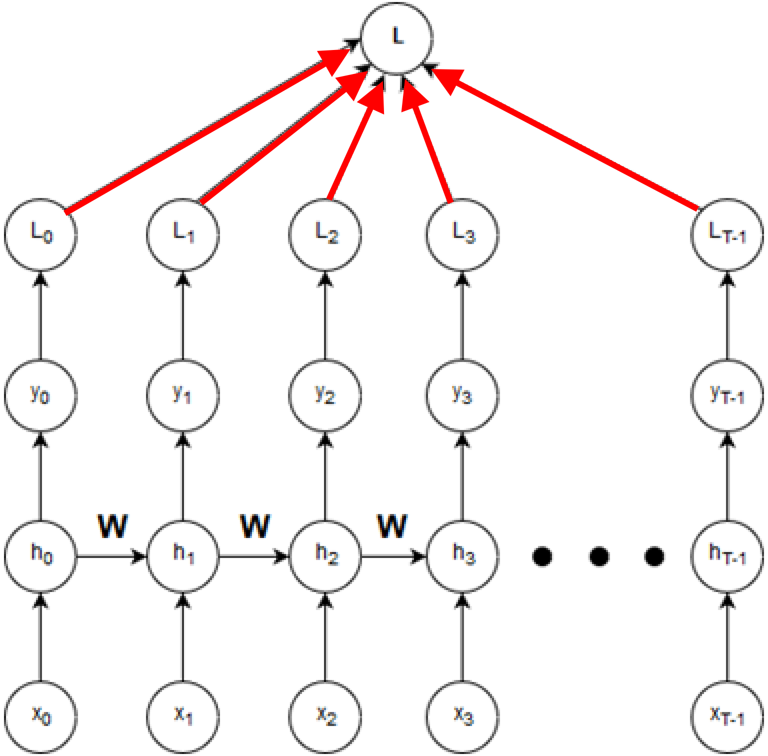
The gradient has two summations:

1: Over L_j

2: Over h_k

To skip proof, click here.

Backpropagation as two summations

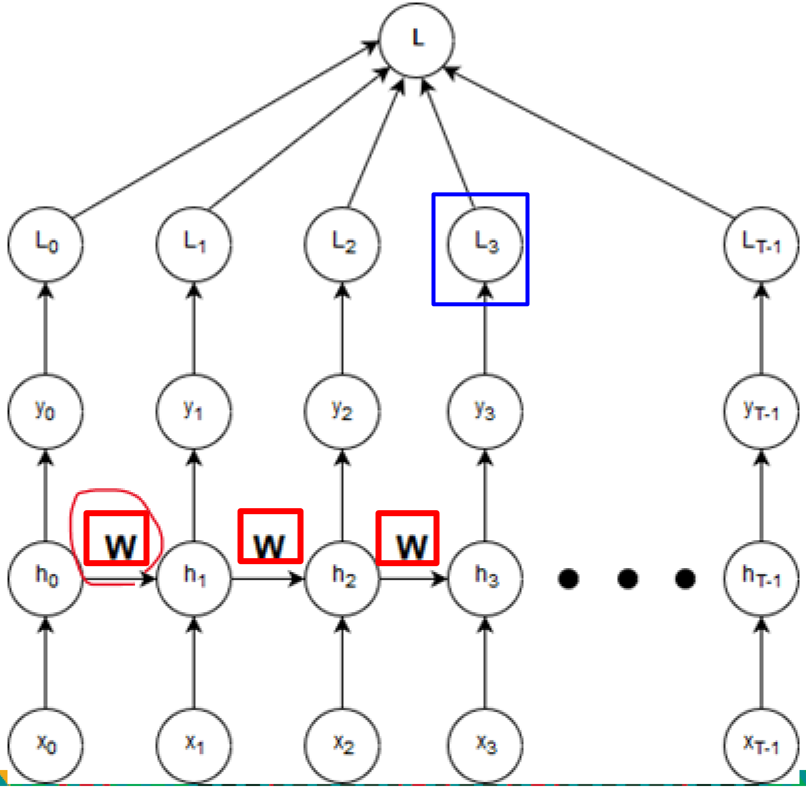


First summation over L

$$\frac{\partial L}{\partial \mathbf{W}} = \sum_{j=0}^{T-1} \frac{\partial L_j}{\partial \mathbf{W}}$$

Carnegie Mellon University

Backpropagation as two summations

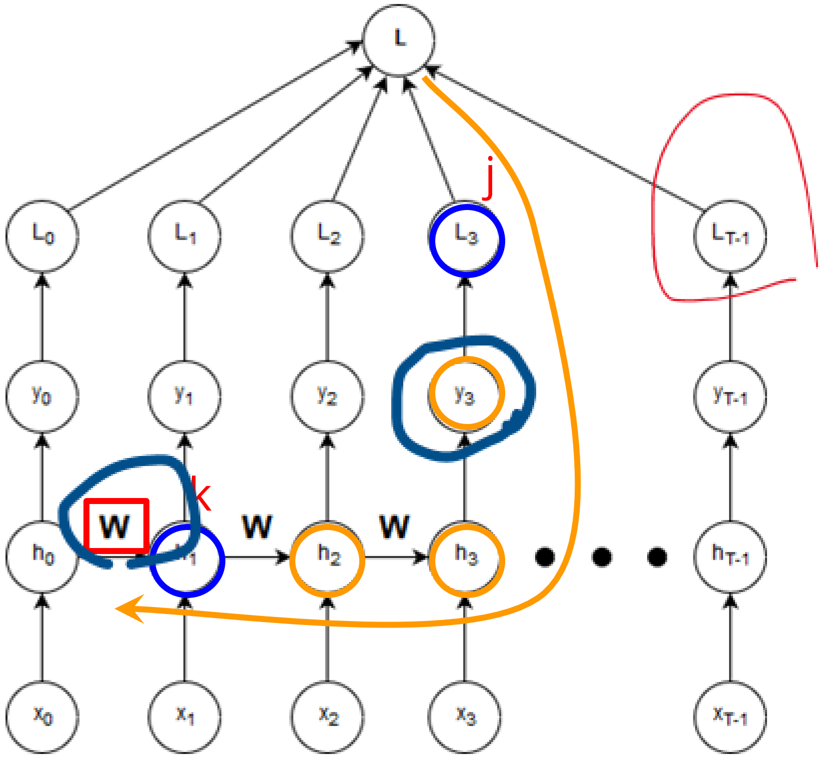


- **Second summation over h:** Each $\mathbf{L_j}$ depends on the weight matrices *before it*

$$\frac{\partial L_j}{\partial \mathbf{W}} = \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

L_j depends on all h_k before it.

Backpropagation as two summations



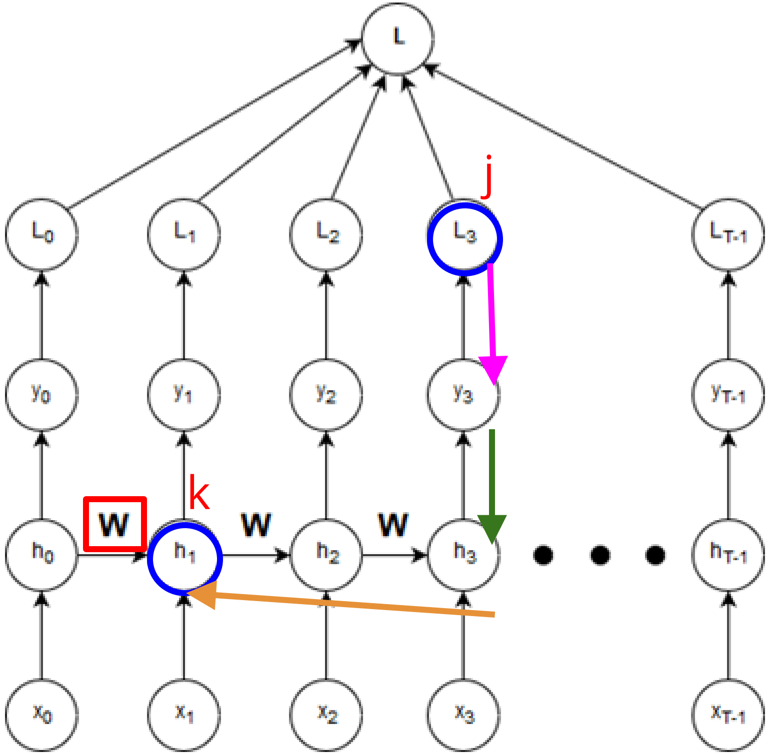
$$\frac{\partial L_j}{\partial W} = \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

- No explicit of L_j on h_k
- Use chain rule to fill missing steps

$\frac{\partial L_3}{\partial y_3}$

$$\frac{\partial L_j}{\partial \mathbf{W}} = \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \frac{\partial h_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

Backpropagation as two summations

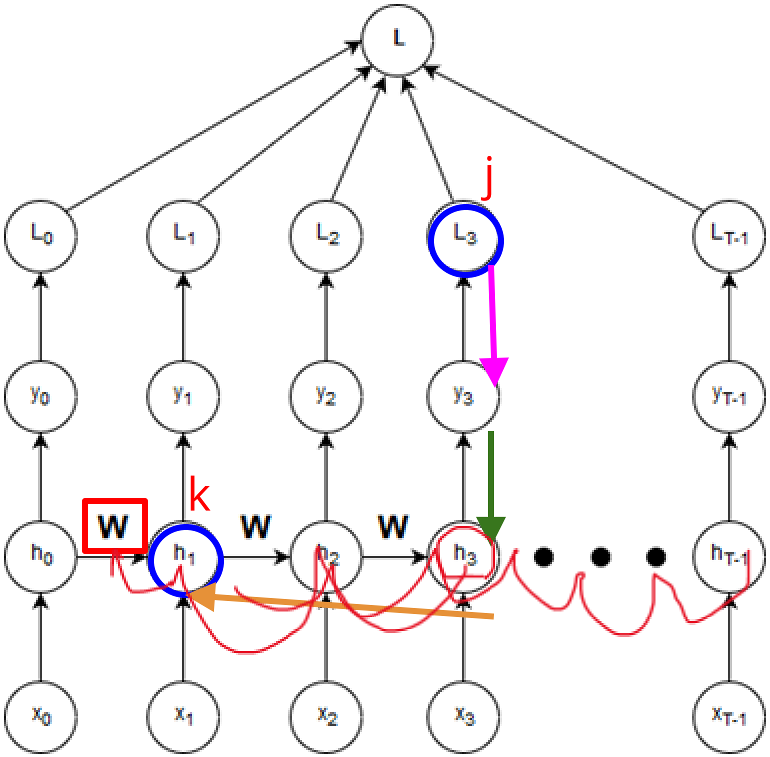


$$\frac{\partial L_j}{\partial W} = \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

- No explicit of L_j on h_k
- Use chain rule to fill missing steps

$$\frac{\partial L_j}{\partial \mathbf{W}} = \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial y_j}} \boxed{\frac{\partial y_j}{\partial h_j}} \boxed{\frac{\partial h_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

The Jacobian



$$\frac{\partial L_j}{\partial \mathbf{W}} = \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial y_j}} \boxed{\frac{\partial y_j}{\partial h_j}} \boxed{\frac{\partial h_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

Indirect dependency. One final use of the chain rule gives:

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

$$\frac{\partial h_3}{\partial h_1} = \frac{\partial h_3}{\partial h_2} \times \frac{\partial h_2}{\partial h_1}$$

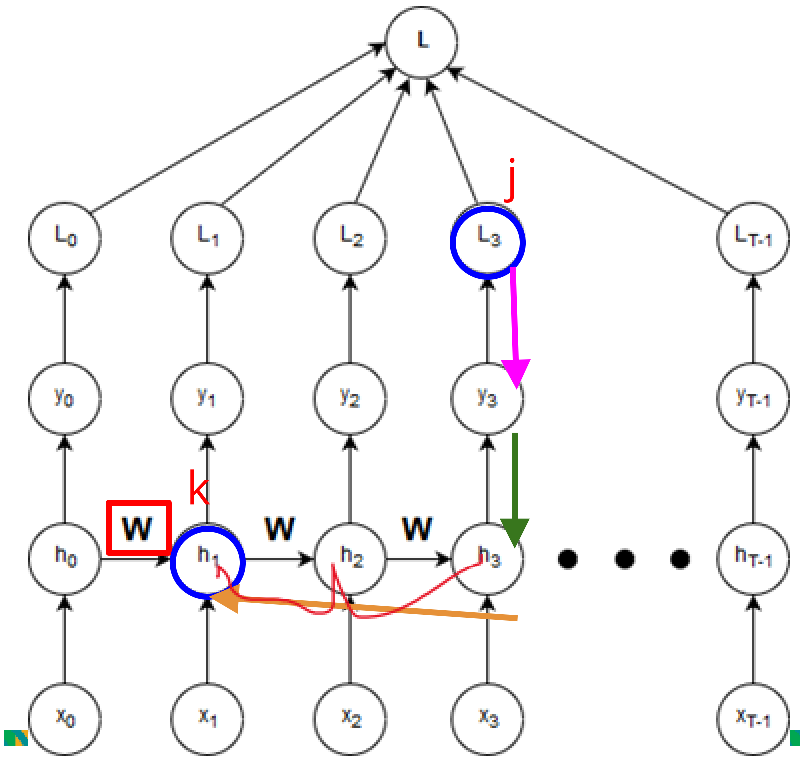
$$\frac{\partial h_3}{\partial h_2} \, \frac{\partial h_2}{\partial h_1}$$

"The Jacobian"

The Final Backpropagation Equation

$$\frac{\partial L}{\partial \mathbf{W_h}} = \sum_{j=0}^{T-1} \sum_{k=1}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \left(\prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}} \right) \frac{\partial h_k}{\partial \mathbf{W_h}}$$

Backpropagation as two summations



$$\frac{\partial L}{\partial \mathbf{W_h}} = \sum_{j=0}^{T-1} \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial y_j}} \boxed{\frac{\partial y_j}{\partial h_j}} \boxed{\left(\prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}} \right)} \frac{\partial h_k}{\partial \mathbf{W_h}}$$

- Often, to reduce memory requirement, we truncate the network
- Inner summation runs from *j-p* to *j* for some *p* ==> truncated BPTT

Expanding the Jacobian

$$\frac{\partial L}{\partial W} = \sum_{j=0}^{T-1} \sum_{k=1}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \boxed{\left(\prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}} \right)} \frac{\partial h_k}{\partial \mathbf{W}}$$

$$h_m = f(\mathbf{W}_h h_{m-1} + \mathbf{W}_x x_m)$$

$$\frac{\partial h_m}{\partial h_{m-1}} = \mathbf{W}_h^T \mathrm{diag}\left(f'(\mathbf{W}_h h_{m-1} + \mathbf{W}_x x_m)\right)$$

The Issue with the Jacobian

$$\frac{\partial h_j}{\partial h_k} = \prod_{m=k+1}^{j} \boxed{\mathbf{W}_h^T} \boxed{\mathrm{diag}\left(f'(\mathbf{W}_h h_{m-1} + \mathbf{W}_x x_m)\right)}$$

Weight Matrix

Derivative of activation function

REPEATED MATRIX MULTIPLICATIONS LEADS TO **VANISHING AND EXPLODING GRADIENTS.**
HOW? LET'S TAKE A SLIGHT DETOUR.

Eigenvalues and Stability

Vanishing gradients

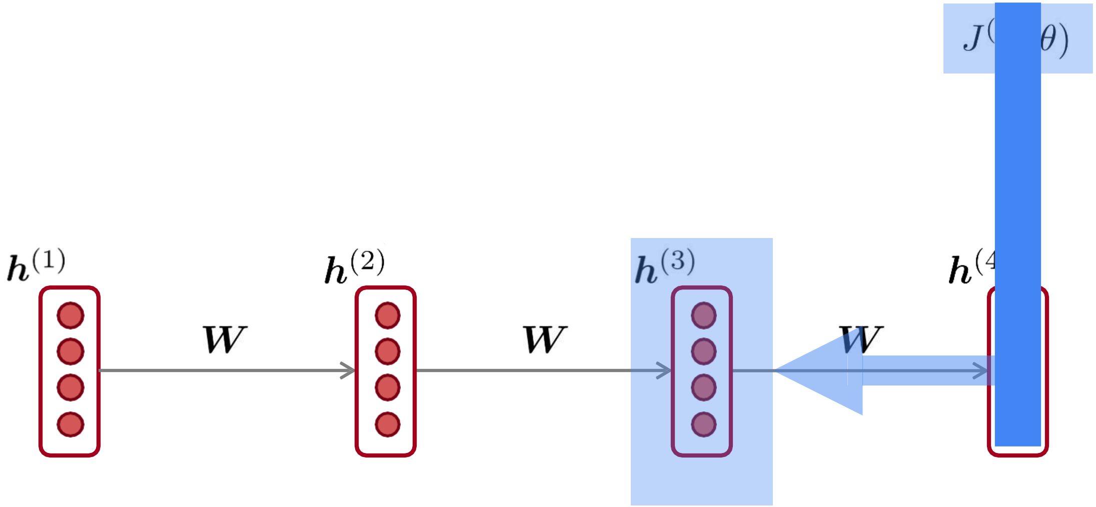
$$\Lambda = \begin{bmatrix} -0.6180 & 0 \\ 0 & 1.6180 \end{bmatrix} \longrightarrow \Lambda^{10} = \begin{bmatrix} \boxed{0.0081} & 0 \\ 0 & \boxed{122.9919} \end{bmatrix}$$

Exploding gradients

$$W_h^n = Q^{-1} * \Lambda^n * Q$$

Carnegie Mellon University

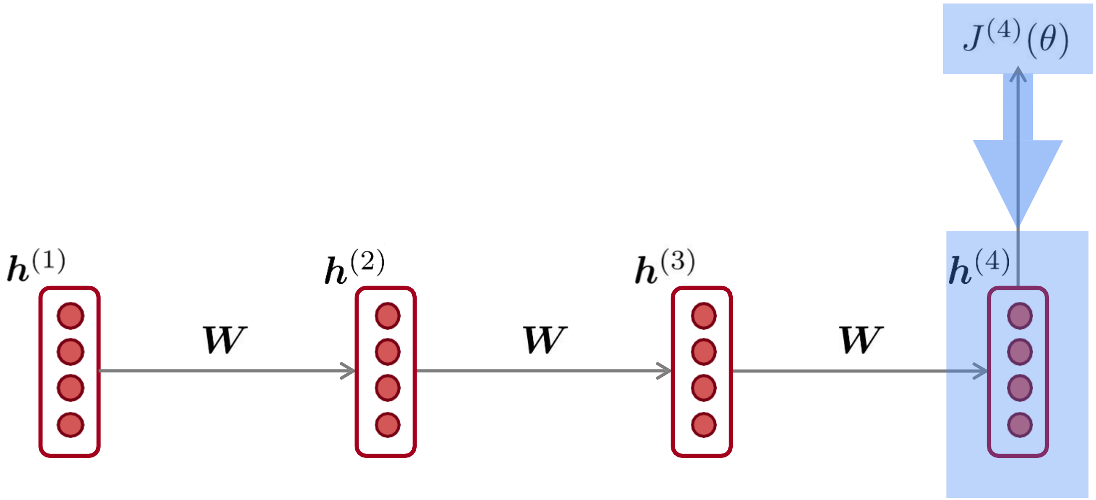
Vanishing gradient intuition



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

chain rule!

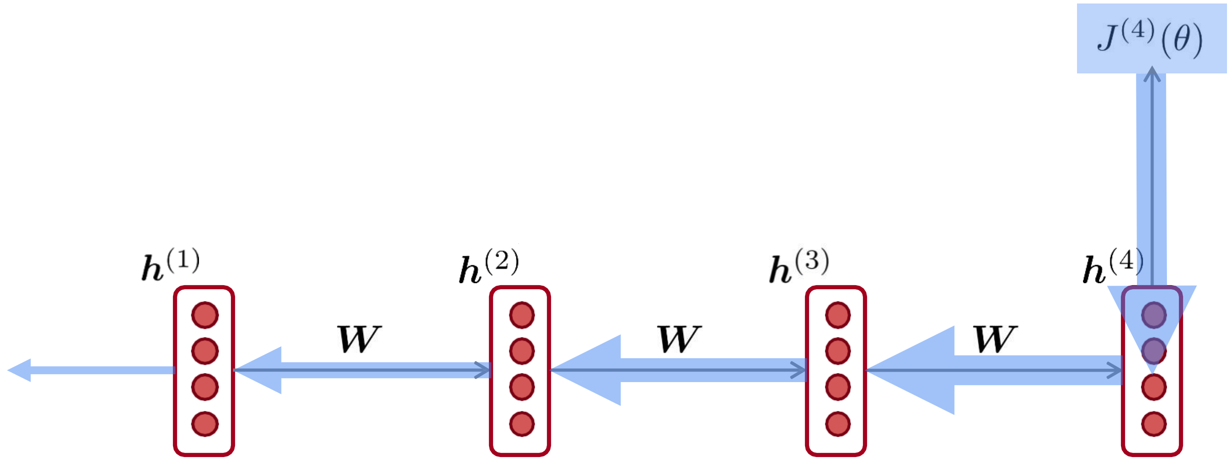
Vanishing gradient intuition



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

chain rule!

Vanishing gradient intuition



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

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)

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

Vanishing gradient proof sketch

- Consider matrix L2 norms:

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

- 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

- 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

Carnegie Mellon University

Adapted from Abigail See

RNN Training Problem

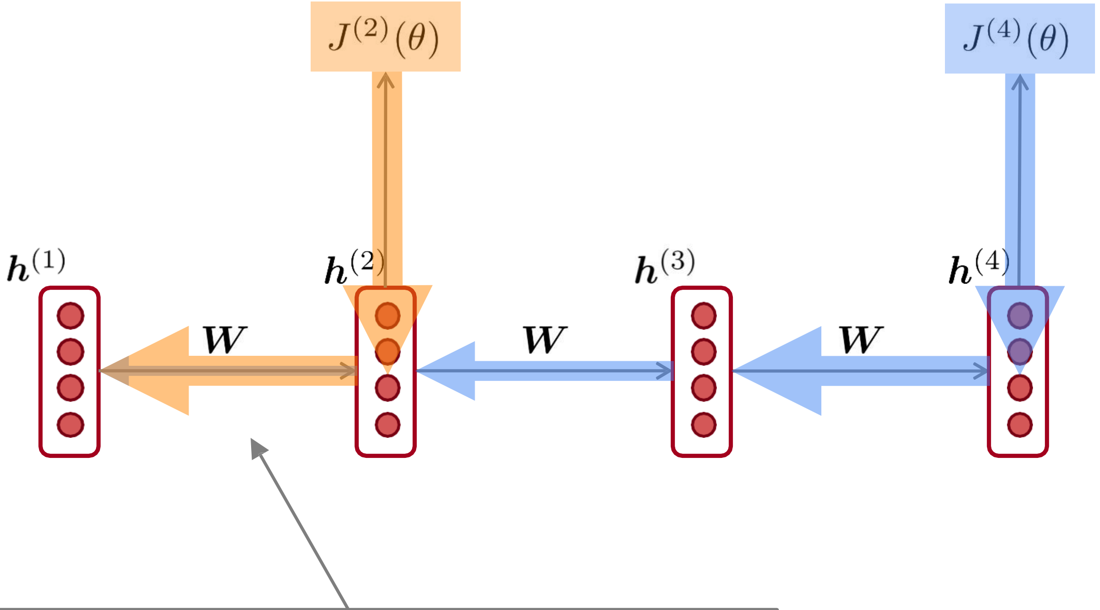
Training recurrent neural networks can be very difficult. Two common issues with training recurrent neural networks are vanishing gradients and exploding gradients. Exploding gradients can occur when the gradient becomes too large and error gradients accumulate, resulting in an unstable network. Vanishing gradients can happen when optimization gets stuck at a certain point because the gradient is too small to progress. Gradient clipping can prevent these issues in the gradients that mess up the parameters during training.

MachineLearningmastery.com

RNN Training Problem

Jane walked into the room. John walked in too. It was late in the day. Jane said hi to ___ .

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.

Abigail See

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 7^{th} 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

Carnegie Mellon University

Effect of vanishing gradient on RNN-LM

- **LM task:** *The writer of the books* ____ *is* / *are*

- **Correct answer**: *The writer of the books <u>is</u> planning a sequel*

- **Syntactic recency:** *The <u>writer</u> of the books <u>is</u>* (correct)

- **Sequential recency:** *The writer of the <u>books</u> <u>are</u>* (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:

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

- 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

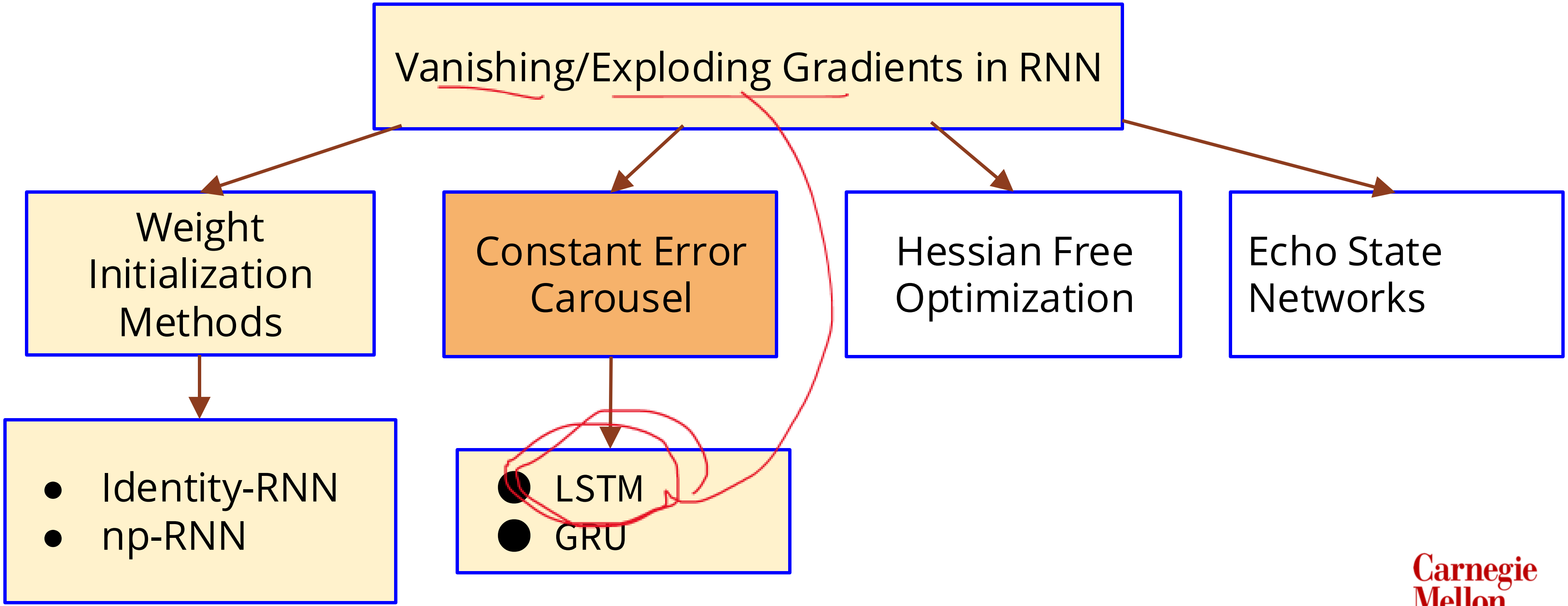
Algorithm 1 Pseudo-code for norm clipping

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

Learning Long Term Dependencies

Outline


Vanishing/Exploding Gradients in RNN
Weight Initialization Methods
Constant Error Carousel
Hessian Free Optimization
Echo State Networks
Identity-RNN
np-RNN
LSTM
GRU

Weight Initialization Method

$$\frac{\partial h_j}{\partial h_k} = \prod_{m=k+1}^{j} \mathbf{W}_h^T \mathrm{diag}\left(f'(\mathbf{W}_h h_{m-1} + \mathbf{W}_x x_m)\right)$$

Activation function : ReLU

$$\frac{\partial h_j}{\partial h_k} = (\mathbf{W}_h^T)^n \;\; = Q^{-1} * \Lambda^n * Q$$

Carnegie Mellon University

Weight Initialization Method

RANDOM W_H INITIALIZATION OF RNN HAS NO CONSTRAINT ON EIGENVALUES

⇒ VANISHING OR EXPLODING GRADIENTS IN THE INITIAL EPOCH

Carnegie Mellon University

Weight Initialization Trick #1: IRNN

- W_H INITIALIZED TO IDENTITY

- ACTIVATION FUNCTION: RELU

Geoffrey et al, "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units"

Weight Initialization Trick #2: np-RNN

- W_h positive definite (+ve real eigenvalues)
- At least one eigenvalue is 1, others all less than equal to one
- Activation function: ReLU

Geoffrey et al, "Improving Performance of Recurrent Neural Network with ReLU nonlinearity""

Carnegie Mellon University

Weight Initialization Method

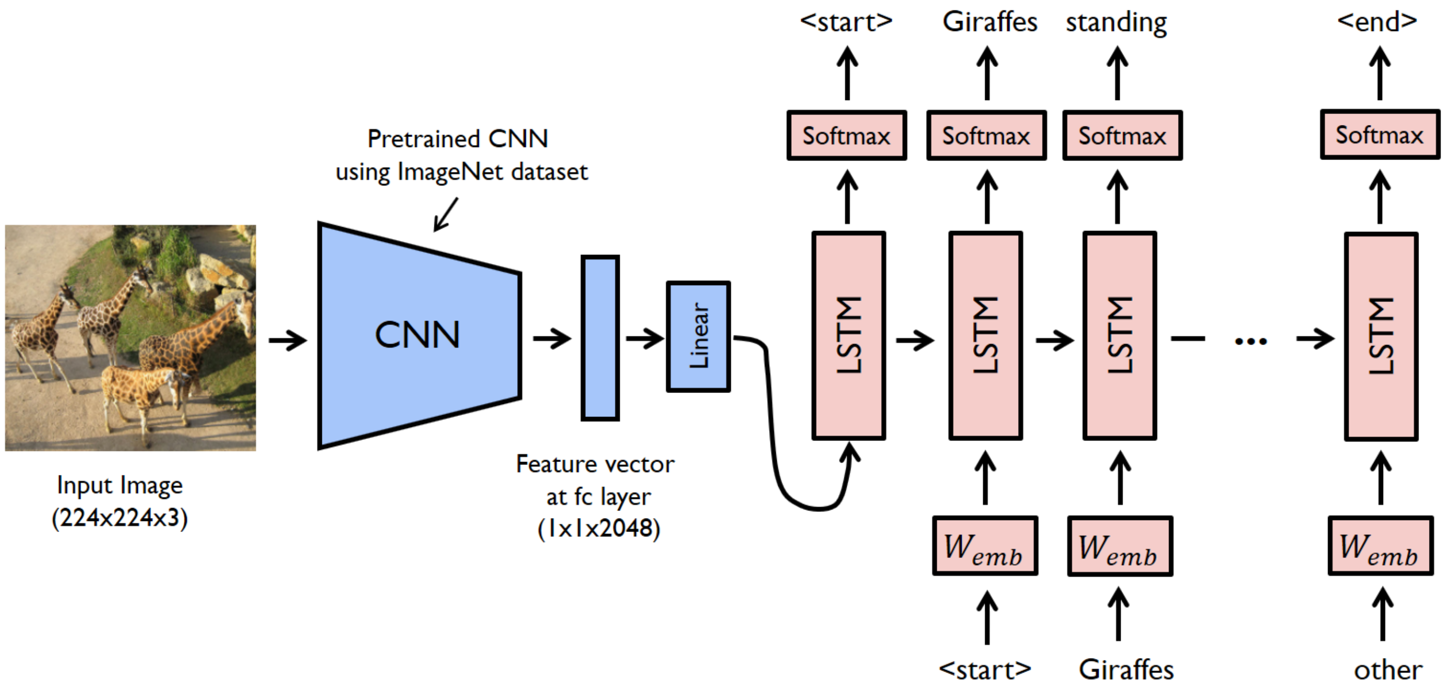
CAREFUL INITIALIZATION OF W_H WITH SUITABLE EIGENVALUES

⇒ ALLOWS THE RNN TO LEARN IN THE INITIAL EPOCHS

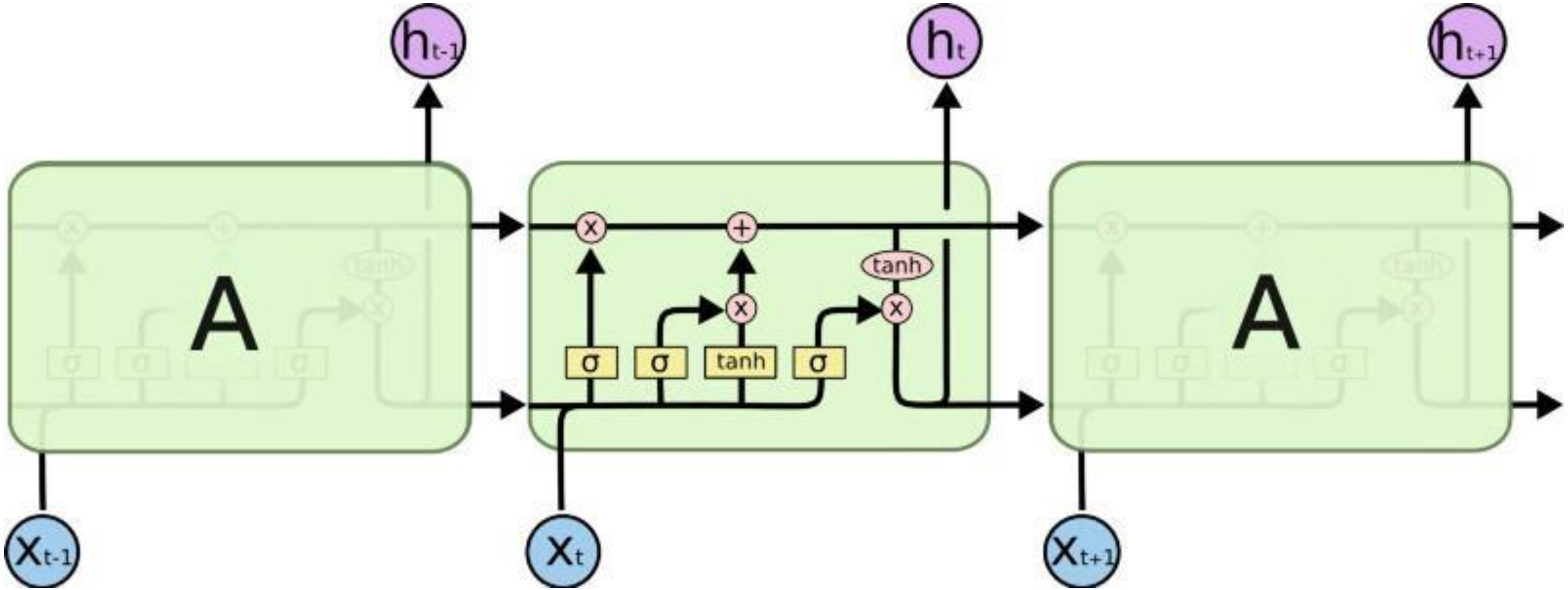
⇒ HENCE CAN GENERALIZE WELL FOR FURTHER ITERATIONS

Carnegie Mellon University

Motivation: RNNs are not new but combination with CNN



The LSTM Network



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