Deep Learning for Language: Recurrent Neural Networks

Robin Jia
USC CSCI 467, Spring 2025
March 4, 2025

Peculiarities of language data

- Peculiarity #1: Text is not a numerical format
  - Feature vector = list of numbers
  - Image = 3xWxH grid of pixel brightness values
  - Text = sequence of words, not numbers
- Peculiarity #2: Text is variable sized
  - Feature vectors are always the same size for different examples
  - Images can be cropped/rescaled to be the same size for all examples
  - Text: Different examples have different # of words

Feeding Words to a Neural Network

- Peculiarity #1: Words are not numerical
- Solution: Learn word vectors, feed word vector of each word to model!
- Original input: T words
- Vector input: T vectors, each of size d

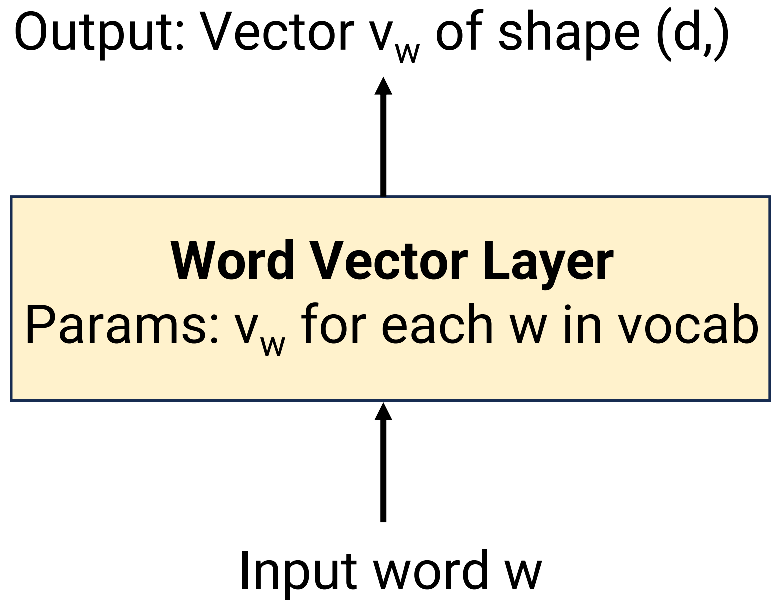
Text input:	*A*	*zoo*	*elephant*
Vector input:	-0.4	2.1	2.1
	1.4	-1.4	-1.3
	-1.2	3.2	0.3

Word *w*	Vector v_w
A	[-0.4, 1.4, -1.2]
Aardvark	[2.2, -1.8, 0.6]
Airport	[0.7, 0.3, 3.1]
…	
Elephant	[2.1, -1.3, 0.3
…	
Zoo	[2.1, -1.4, 3.2]

RNN “Building Blocks”

(6) Word Vector Layer

- Input w: A word (from our vocabulary)
  - Can also input list of words
- Output: A vector of length d
  - If input is many words, output is list of vectors for each word
- Formula: Return `word_vecs[w]`
- Parameters:
  - For each word w in vocabulary, there is a word vector parameter v_w of shape d
  - Think of this as a dictionary called `word_vecs`, where the keys are words & values are learned parameter vectors
  - Can initialize using word2vec, or randomly
  - Train them further with gradient descent to help final task
- In pytorch: nn.Embedding()

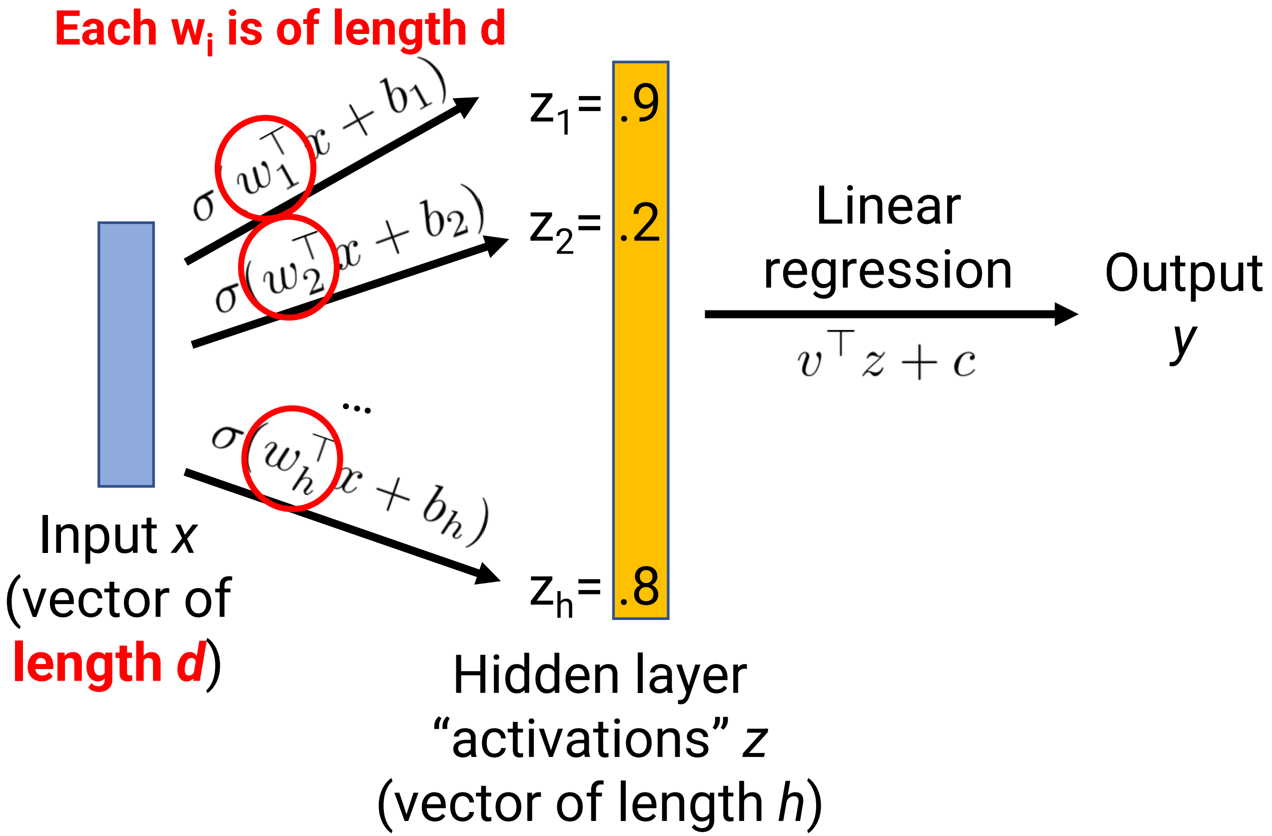


Outline

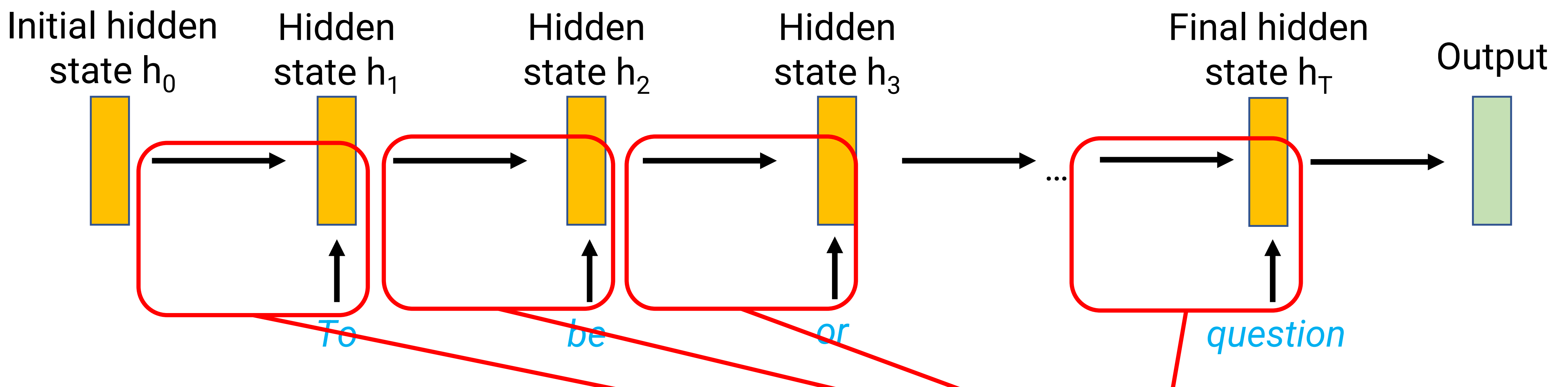
- Recurrent Neural Networks (RNNs) for sequential data
- Language modeling and Long-range dependencies
- Vanishing gradients and Gated RNNs

Handling variable length

- Peculiarity #2: Documents have different numbers of words
  - Example 1: *Amazing!*
  - Example 2: *There are many issues with this movie, such as…*
- Problem: In previous models, number of parameters depends on size of inputs
- Challenge: How can we use the **same** set of model parameters to handle inputs of any size?

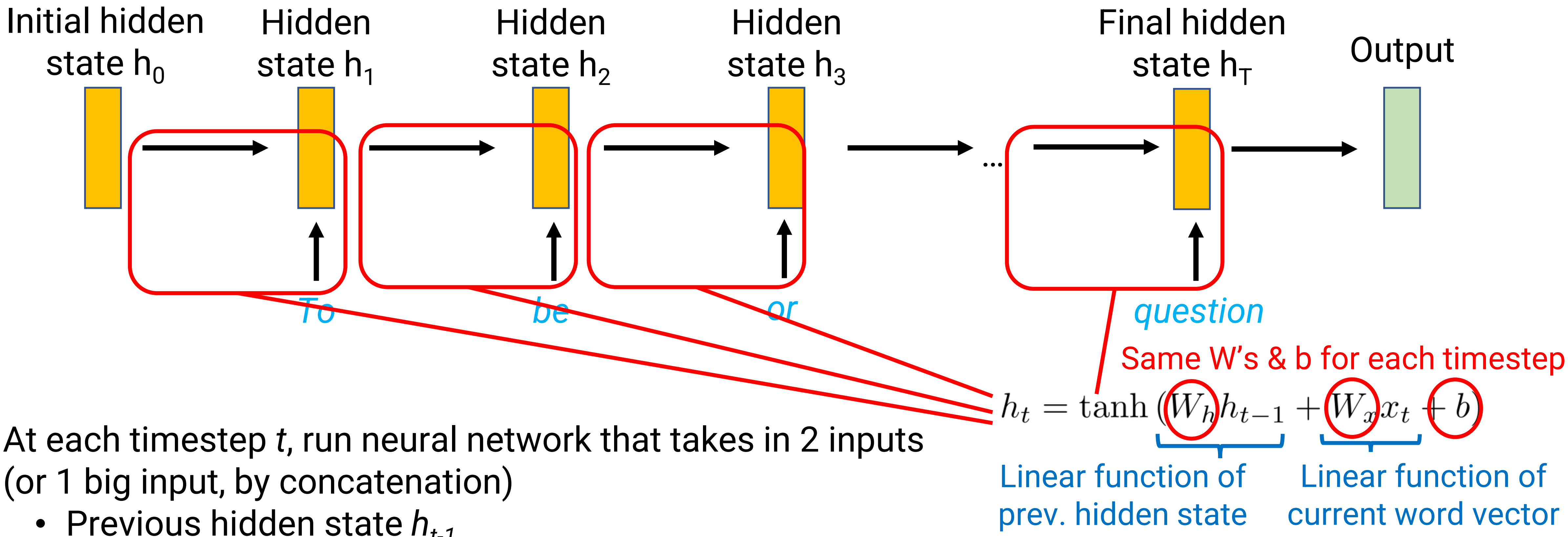


Recurrent Neural Networks (RNNs)



- Idea: Recurrence!
  - “Read” the input one word at a time
  - At each step, update the hidden state of the network
  - **Model parameters to do this update are same for each step**

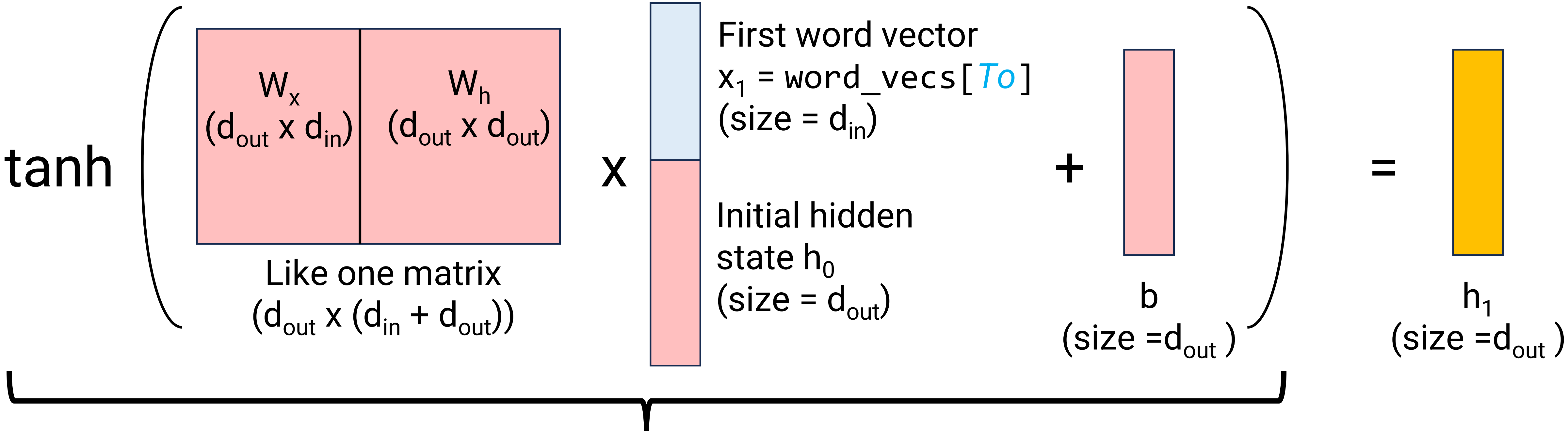
A “Vanilla”/”Elman” RNN



$$h_t = \tanh\left(\underbrace{W_h h_{t-1}} + \underbrace{W_x x_t} + b\right)$$

- At each timestep *t*, run neural network that takes in 2 inputs (or 1 big input, by concatenation)
  - Previous hidden state h_{t-1}
  - Vector for current word x_t
- Learn linear function of both inputs, add bias, apply non-linearity
- Parameters: Recurrence params (W_h, W_x, b), initial hidden state h_0, word vectors

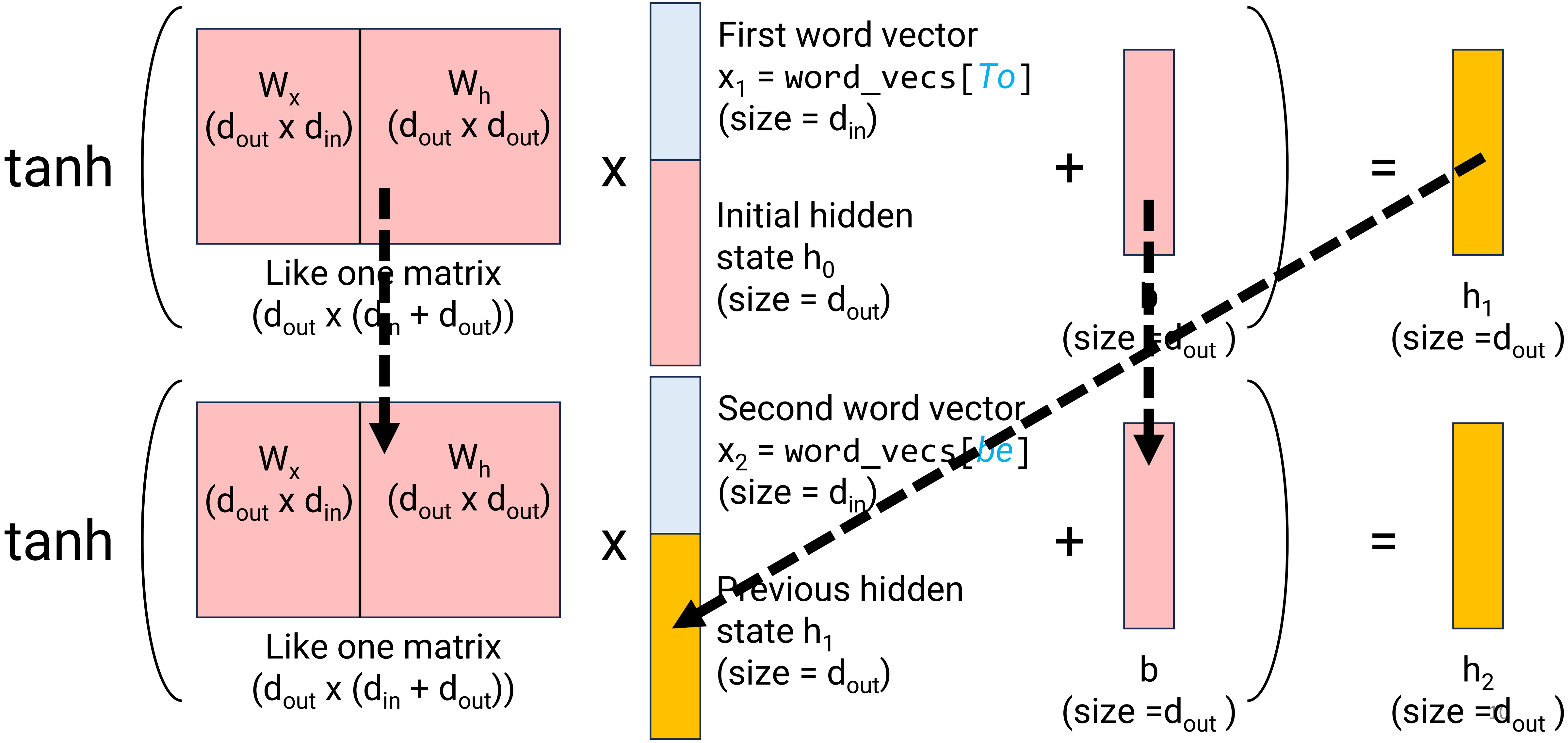
RNN as Repeated Application of 1 MLP layer



Linear transformation of input (with bias), then nonlinearity
AKA One MLP layer!

Legend:
Word vectors
Other learned parameters
Hidden states

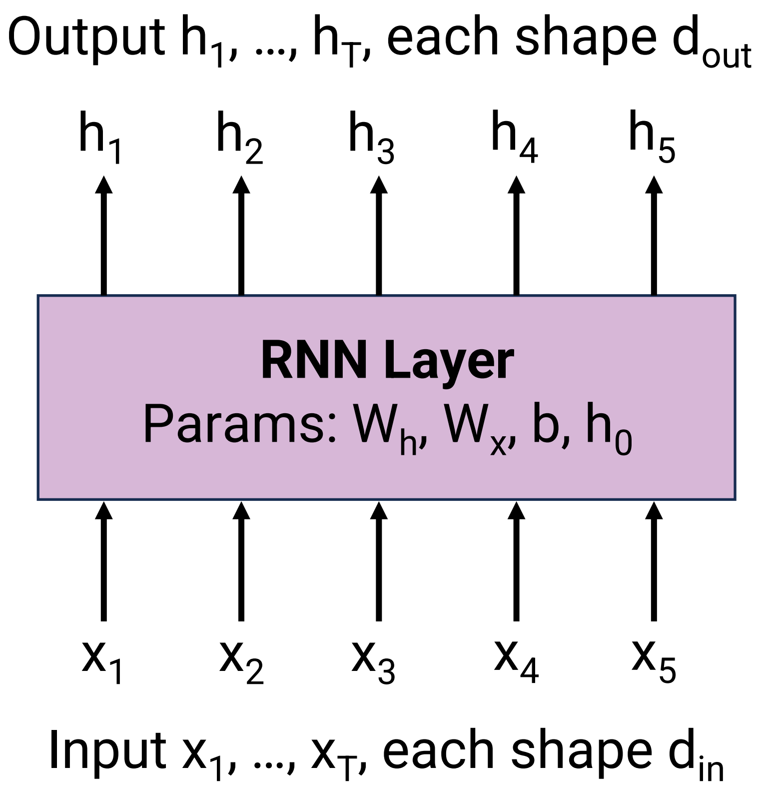
RNN as Repeated Application of 1 MLP layer



RNN "Building Blocks"

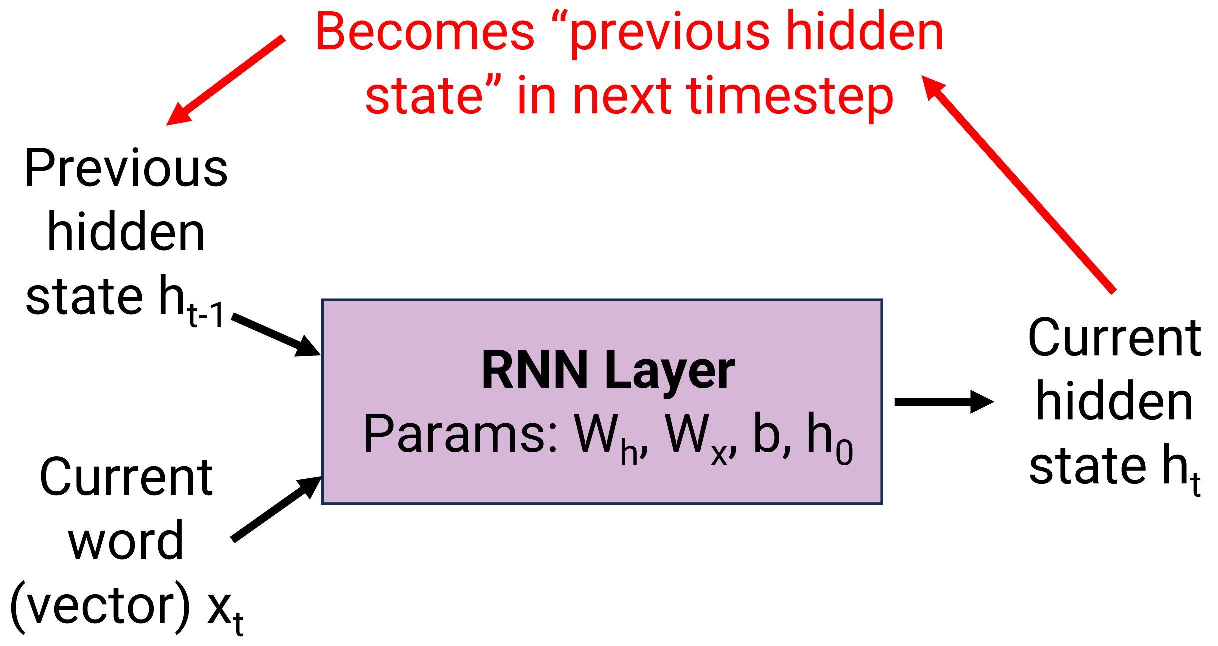
(7) RNN Layer

- Input: List of vectors $x_1, \ldots, x_T$, each of size d_{in}
  - E.g., x_t is word vector for t-th word in sentence
  - Equivalent to a T x d_{in} matrix
- Output: List of vectors $h_1, \ldots, h_t$, each of size d_{out}
  - d_{out}: Dimension of hidden state
  - Equivalent to a T x d_{out} matrix
- Formula (Elman RNN): $h_t = \tanh\left(W_h h_{t-1} + W_x x_t + b\right)$
- Parameters:
  - W_h: Matrix of shape (d_{out}, d_{out})
  - W_x: Matrix of shape (d_{out}, d_{in})
  - b: Vector of shape (d_{out},)
  - h_0: Vector of shape (d_{out},)
- In pytorch: nn.RNN(), etc.

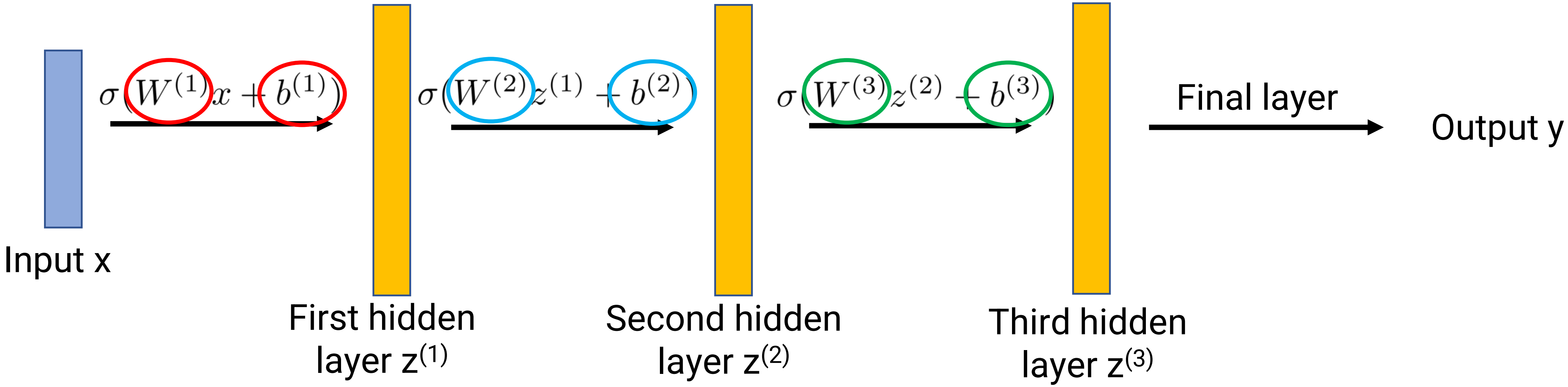


Recurrent Neural Network Diagrams

- Can also visualize RNN's with a diagram that has a cycle ("recurrence")
  - RNN layer is just a neural network that takes in two vectors and produces a third vector
  - New vector gets fed back in next timestep
  - Previous slide is the "unrolled" diagram

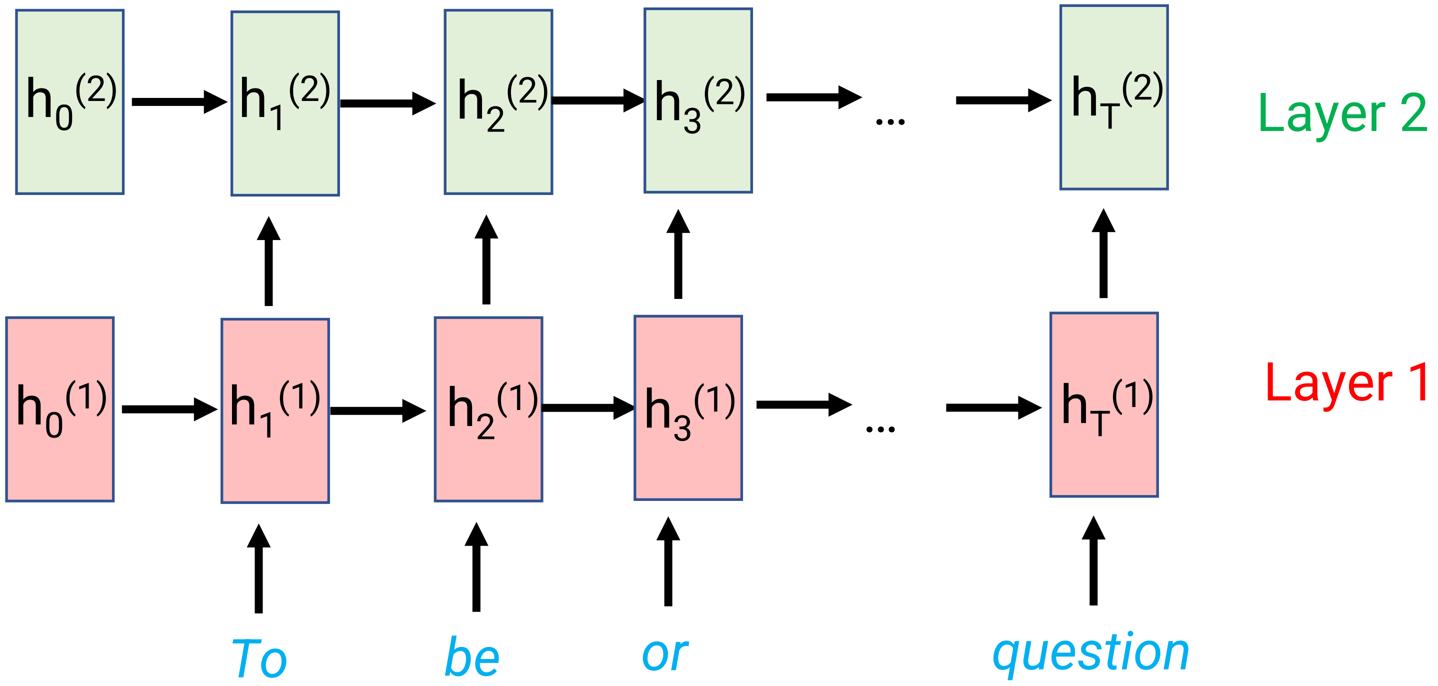


Recurrence vs. Depth



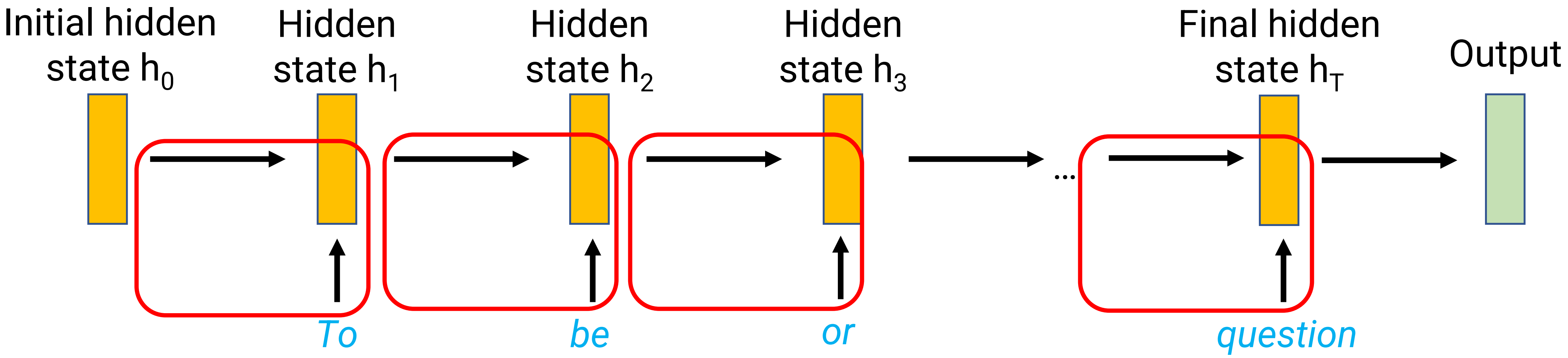
- Deep networks (i.e., adding more layers)
  - Computation graph becomes longer
  - Number of parameters also grows; each step uses new parameters
- Recurrent neural networks
  - Computation graph becomes longer
  - Number of parameters **fixed**; each step uses **same parameters**

Recurrence and Depth



- You can have multiple layers of recurrence too!
  - Left-to-right axis ("time dimension"): Length is size of input, same parameters in each step
  - Top-to-bottom axis ("depth dimension"): Length is depth of network, different parameters in each row

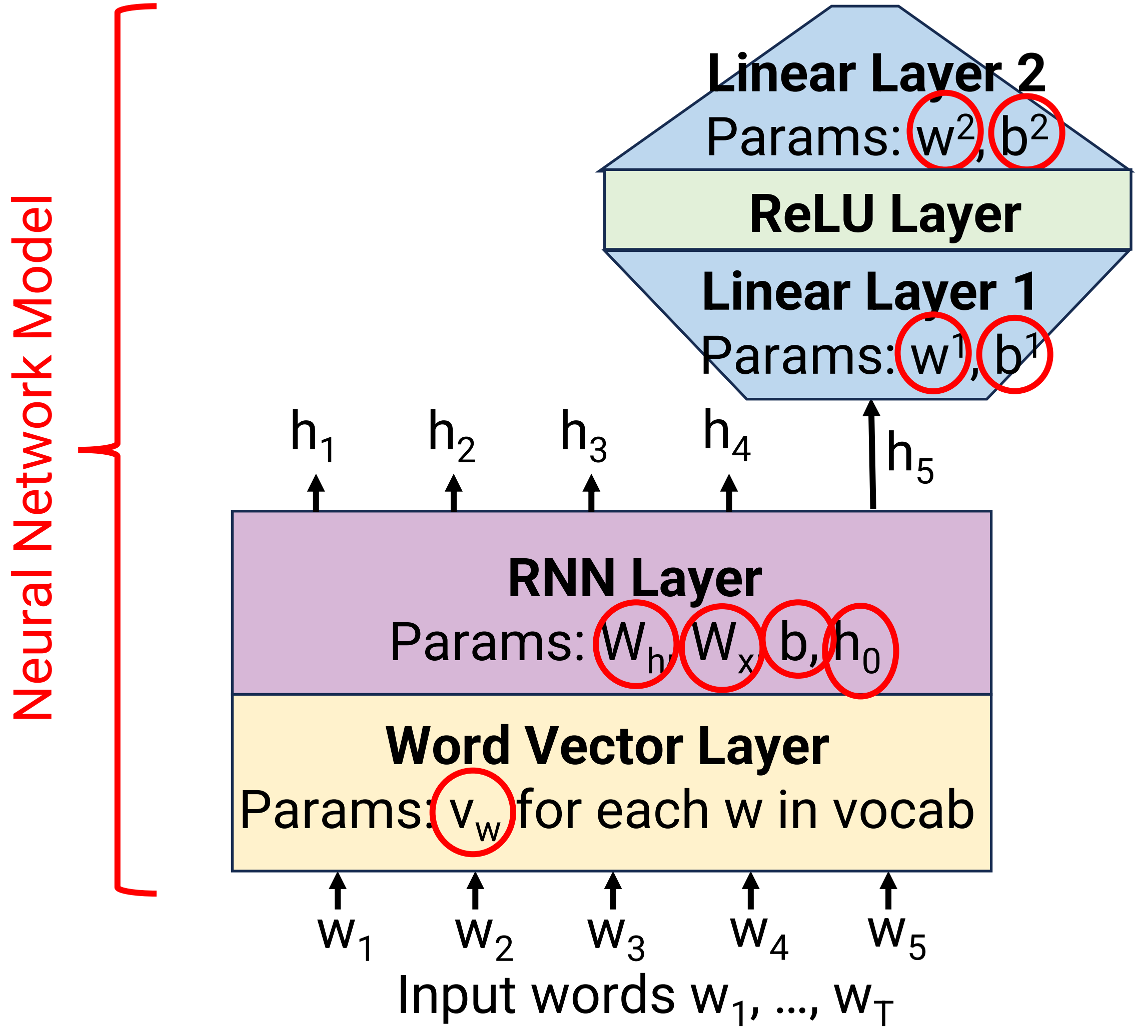
Training an RNN



- Basic usage of RNN: Make prediction based on final hidden state
- Same recipe: Backpropagation to compute gradients + gradient descent
- Must backpropagate through whole computation graph
  - "Backpropagation through time"

Building an RNN encoder model

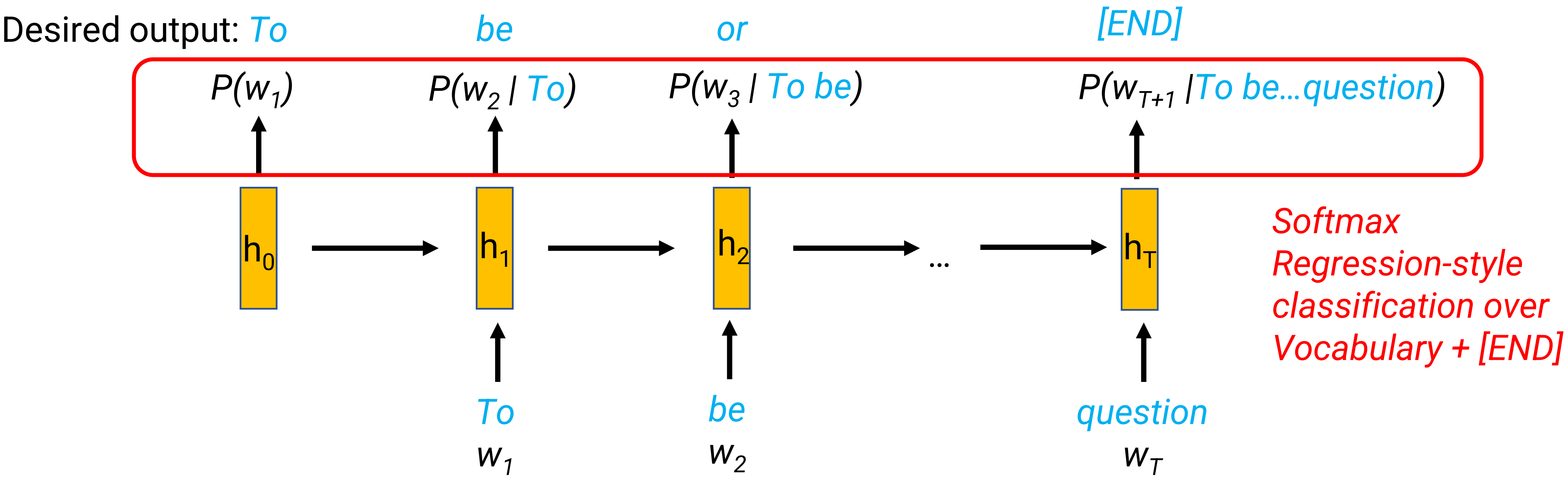
- A generic RNN architecture
  - Map each word to a vector
  - Feed word vectors to RNN to generate list of hidden states
  - Feed final hidden state to MLP to make final prediction (e.g., document classification)
- Basic steps are still all the same
  - Backpropagation still works
- Gradient descent needed to update all parameters



Outline

- Recurrent Neural Networks (RNNs) for sequential data
- Language modeling and Long-range dependencies
- Vanishing gradients and Gated RNNs

Autoregressive Language Modeling



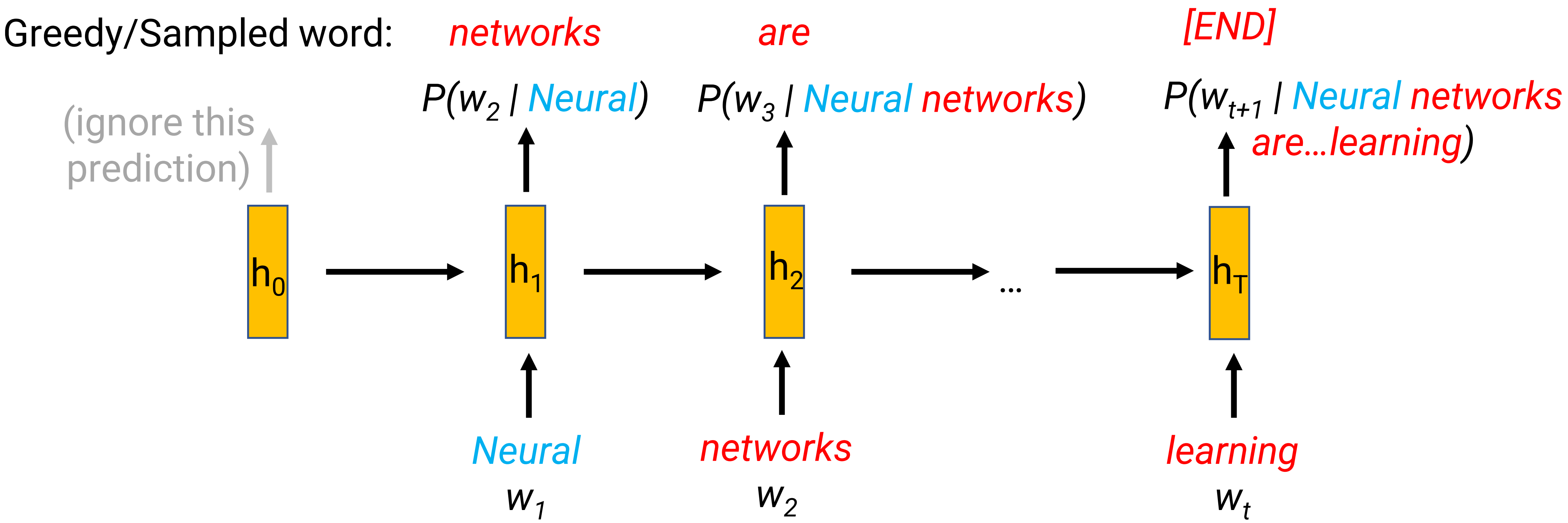
- At each step, probabilistically predict the next word given current hidden state
- One step's desired output is the next step's input ("autoregressive")
- To mark end of sequence, model should predict the *[END]* token
- Called a "Decoder": Looks at the hidden state and "decodes" next word

Autoregressive Language Model Training

- Training example: “Convolutional neural networks are good for image classification”
- Want to maximize P(“Convolutional neural networks are good for image classification”)
- MLE: Take log and decompose by chain rule:

  log P(“Convolutional”)
  + log P(“neural” | “Convolutional”)
  + log P(“networks” | “Convolutional neural”)
  + log P(“are” | “Convolutional neural networks”) + …
- Decomposes into a bunch of **next-word-classification** problems
- Backpropagation + gradient descent to minimize loss
  - Update RNN parameters
  - Update word vectors
  - Update final layer classifier over vocabulary

Generating text with LM's



- Test time: Given some prefix, "autocomplete" the rest of the sentence
- First, feed prefix as input to model
- At each timestep, choose next word based on model's predictions
  - Greedy: Choose the most likely word
  - Sampling: Sample from the model's probability distribution over words
- Feed the model's generated word back as the next word, stop if *[END]*

Long-Range Dependencies

- Every step, you update the hidden state with the current word
- Over time, information from many words ago can easily get lost!
- This means RNNs can struggle to model **long-range dependencies**

The keys to the cabinet ___ (on the table)

plural singular

Long-Range Dependencies

- Every step, you update the hidden state with the current word
- Over time, information from many words ago can easily get lost!
- This means RNNs can struggle to model **long-range dependencies**

*The keys to the cabinet **are** (on the table)*

plural singular

Long-Range Dependencies

- Every step, you update the hidden state with the current word
- Over time, information from many words ago can easily get lost!
- This means RNNs can struggle to model **long-range dependencies**

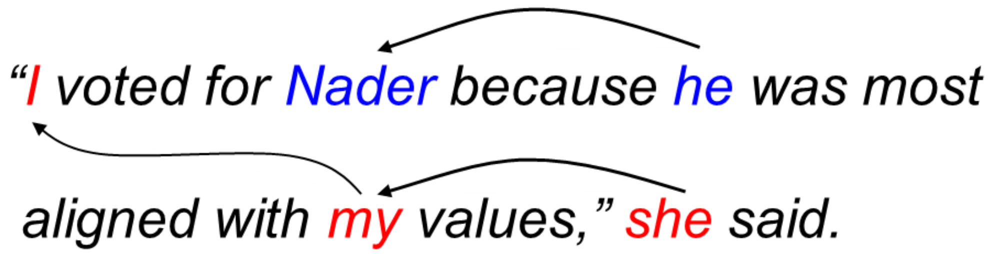
The keys to the cabinet by the door ***are*** *(on the table)*

Long-Range Dependencies

- Every step, you update the hidden state with the current word
- Over time, information from many words ago can easily get lost!
- This means RNNs can struggle to model **long-range dependencies**

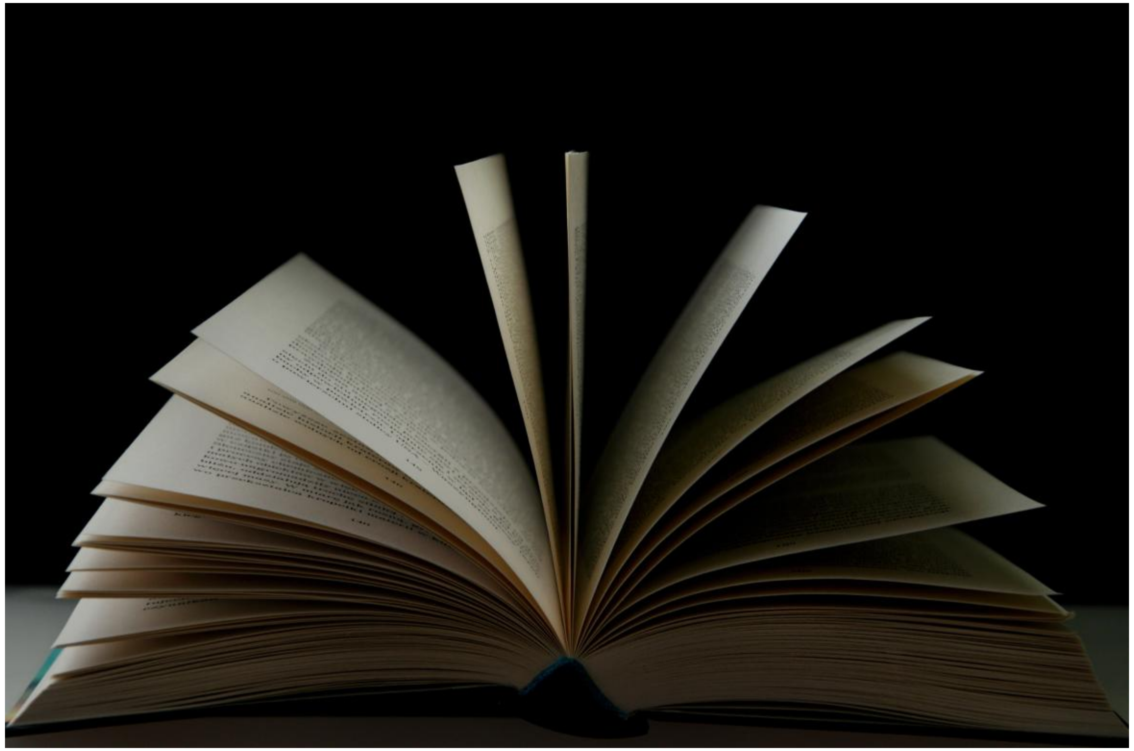
*The keys to the cabinet by the door on the left **are** (on the table)*

Long-Range Dependencies



- "Coreference": When two words refer to the same underlying person/place/thing
  - Pronouns typically ***corefer*** to an ***antecedent*** (something mentioned earlier in the text)
- Coreference relationships can even span multiple sentences

Even longer-range dependencies



- Imagine trying to generate a novel…
  - Same set of characters
  - Characters have to behave in consistent ways
  - Sensible ordering of events

Announcements

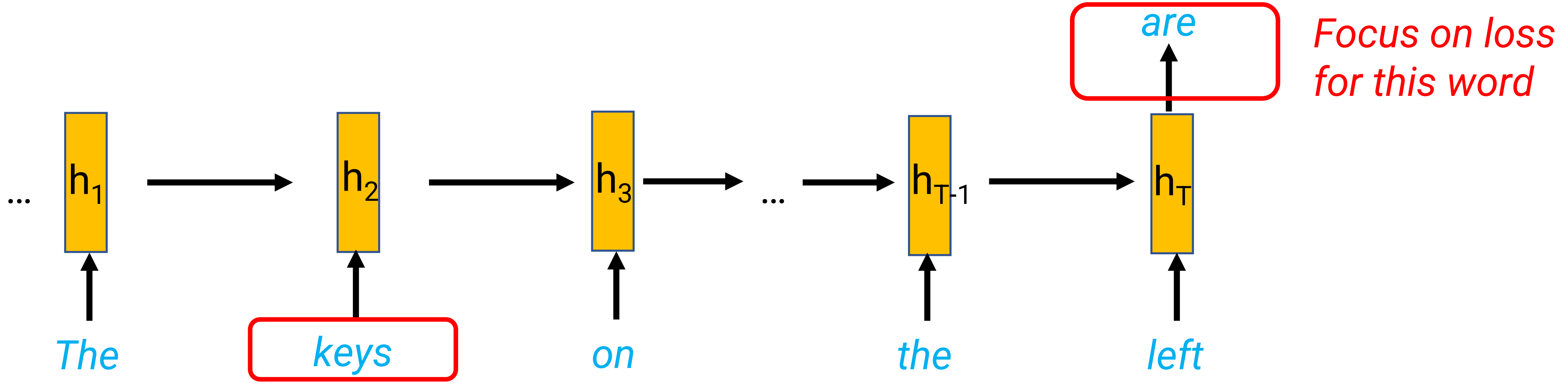
- HW2 due this Thursday, March 6
- Section Friday: Midterm preparation
- Midterm exam: Thursday March 13
  - Refer to Piazza for logistics including which room to go to
  - Practice exams released on website
  - Midterm will cover all topics through end of this week
  - Please write in pen

Outline

- Recurrent Neural Networks (RNNs) for sequential data
- Language modeling and Long-range dependencies
- Vanishing gradients and Gated RNNs

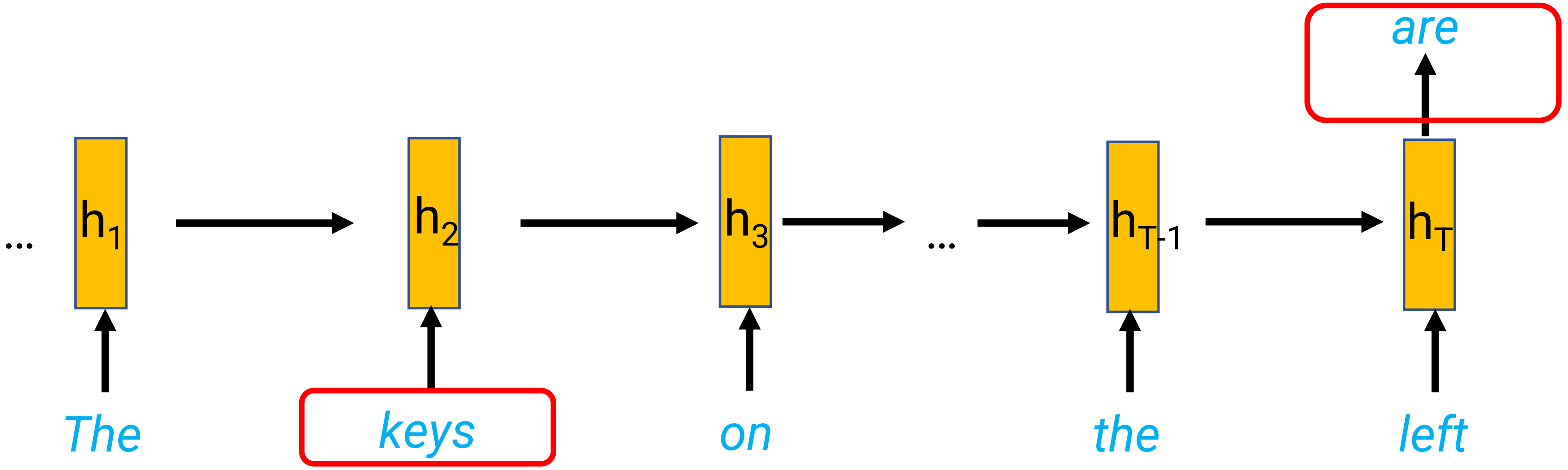
Backpropagation through time, revisited

- Model needs to know that the correct word is *are* because of the word *keys*!
- Let's backpropagate the loss on generating *are* to the word vector parameters for *keys*
  - For simplicity, let's assume all the hidden states are just 1-dimensional
  - Step 1: Compute $\delta Loss/\delta(h_T)$
  - Step 2: Compute $\delta Loss/\delta(h_{T-1}) = \delta Loss/\delta(h_T) * \delta(h_T)/\delta(h_{T-1})$
  - Step 3: Compute $\delta Loss/\delta(h_{T-2}) = \delta Loss/\delta(h_T) * \delta(h_T)/\delta(h_{T-1}) * \delta(h_{T-1})/\delta(h_{T-2})$
  - …
  - Gradient through "*keys*" hidden state: $\delta Loss/\delta(h_T) * \delta(h_T)/\delta(h_{T-1}) * \delta(h_{T-1})/\delta(h_{T-2}) * \ldots * \delta(h_3)/\delta(h_2)$
  - Gradient through "*keys*" word vector: $\delta Loss/\delta(h_T) * \delta(h_T)/\delta(h_{T-1}) * \delta(h_{T-1})/\delta(h_{T-2}) * \ldots * \delta(h_3)/\delta(h_2) * \delta(h_2)/\delta(x_2)$



The Vanishing Gradient Problem

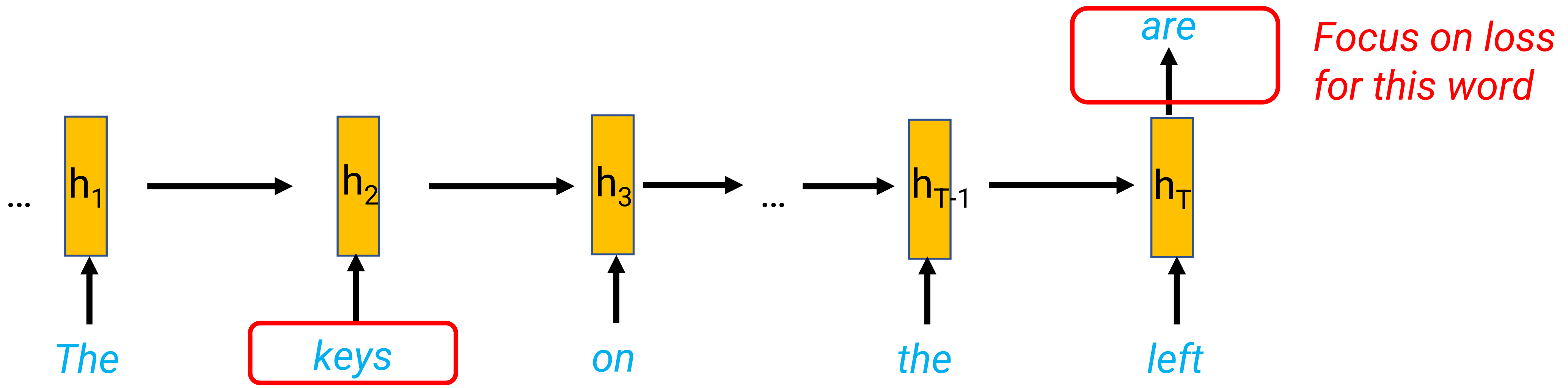
- Gradient through "*keys*" word vector: δLoss/δ(h_T) * δ(h_T)/δ(h_{T-1}) * δ(h_{T-1})/δ(h_{T-2}) * … * δ(h_3)/δ(h_2) * δ(h_2)/δ(x_2)
  - What is each individual δ(h_t)/δ(h_{t-1}) term ?
  - Elman network: $h_t = \tanh\left(W_h h_{t-1} + W_x x_t + b\right),\quad \frac{\delta h_t}{\delta h_{t-1}} = \underbrace{\tanh'\left(W_h h_{t-1} + W_x x_t + b\right)}_{\text{Ignore for now}} \cdot \underbrace{W_h}_{\text{The same parameter over and over!}}$
  - After *t* timesteps, have a factor of *(W_h)^t* (to the *t*-th power)!
  - If W_h << 1, this quickly becomes 0 ("vanishes")



Focus on loss for this word

The Vanishing Gradient Problem

- Vanishing Gradients: Updates to one word/hidden state not influenced by loss on words many steps in the future
  - Illustrated only for 1-dimensional hidden states, but same thing happens when states are vectors/parameters are matrices
- Result: Hard for model to learn long-range dependencies!



Vanishing and Exploding

- Vanishing gradient occurs because
  - Gradient w.r.t. words t steps in the past has $(W_h)^t$
  - And when $W_h << 1$ (e.g., at initialization time)
- What if $W_h > 1$?
  - Gradients get bigger as you go backwards in time: Exploding gradients!
  - Vanishing gradients more usual, but explosion can happen too
- Quick fix: Gradient Clipping
  - If gradient is super large, “clip” it to some maximum amount
    - Rescale the total vector to some maximum norm
    - Clip each entry to be within some minimum/maximum value
- Outside of RNNs, vanishing/exploding gradients can happen whenever you have long computation graphs with lots of multiplications

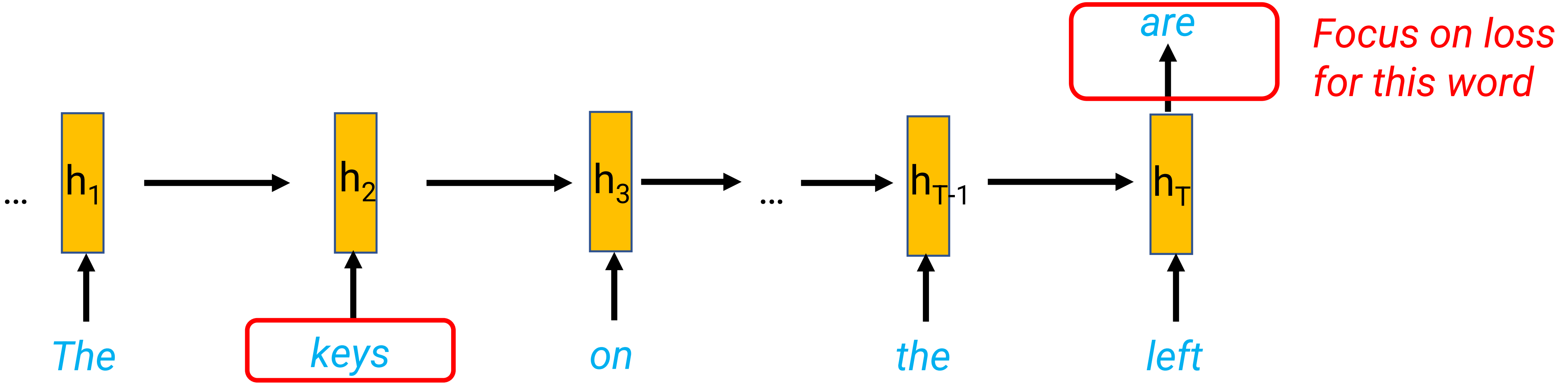
Avoiding Vanishing Gradients

- Where did we go wrong?

$$h_t = \tanh\left(\underbrace{W_h h_{t-1}} + W_x x_t + b\right), \quad \frac{\delta h_t}{\delta h_{t-1}} = \tanh'\left(W_h h_{t-1} + W_x x_t + b\right) \underbrace{\cdot W_h}$$

Multiplicative relationship between previous state and next state

Leads to repeated multiplication by W_h

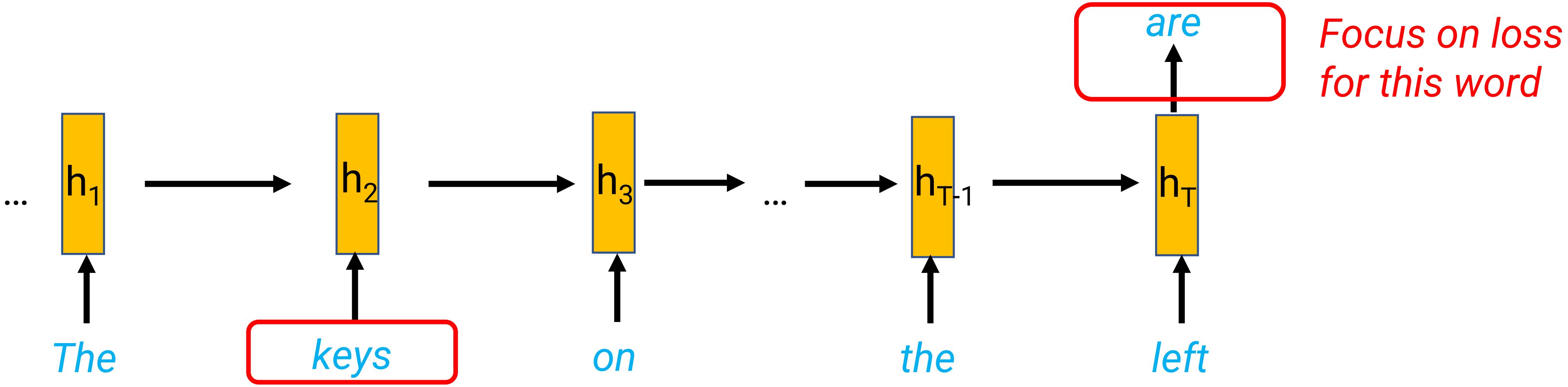


Avoiding Vanishing Gradients

- Extreme idea: A purely additive relationship
  - Pro: No vanishing gradients
  - Pro: Old hidden state carried through to all future times
  - Con: May be good to "forget" irrelevant information about old states

$$h_t = h_{t-1} + \underbrace{g(h_{t-1}, x_t)}_{\text{Additive relationship}},$$

$$\frac{\delta h_t}{\delta h_{t-1}} = \underbrace{1 + \frac{\delta}{\delta h_{t-1}} g(h_{t-1}, x_t)}_{\text{Gradients also add, not multiply}}$$

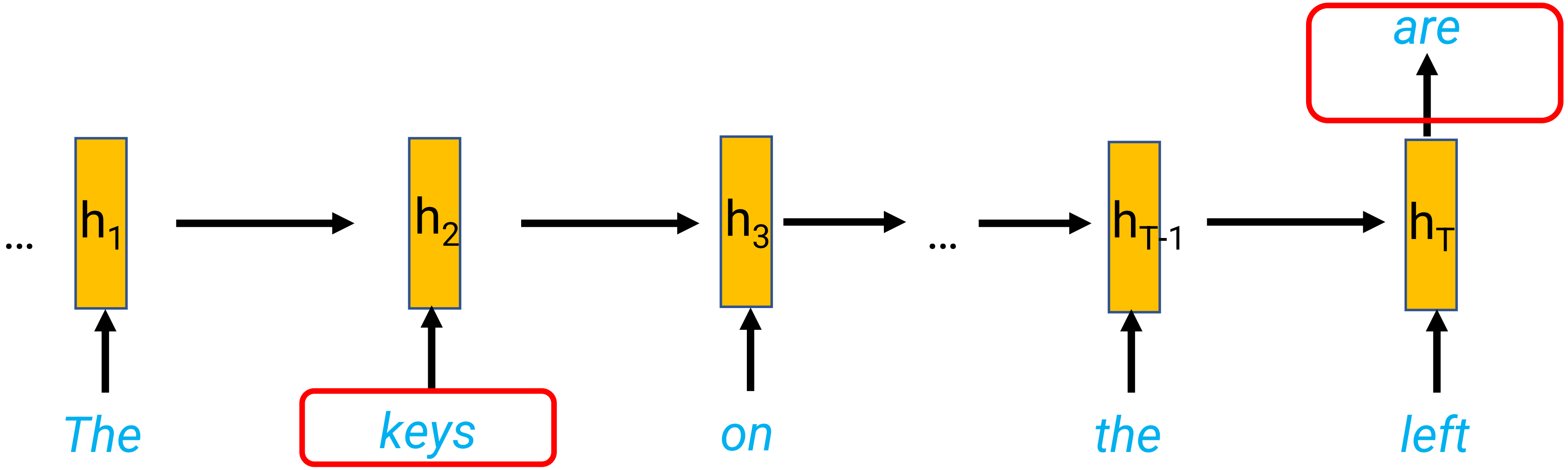


Avoiding Vanishing Gradients

- Middle-ground: **Gated** recurrence relationship
  - Additive component makes gradients add, not multiply = less vanishing gradients
  - Forget gate allows for selectively "forgetting" some neurons within hidden state
  - When forget gate is all 1's, becomes the purely additive model (no vanishing)

Elementwise multiplication

$$h_t = h_{t-1} \underbrace{\odot f(h_{t-1}, x_t)}_{\text{"forget gate" in [0, 1]}} + \underbrace{g(h_{t-1}, x_t)}_{\text{Additive relationship}}$$



Focus on loss for this word

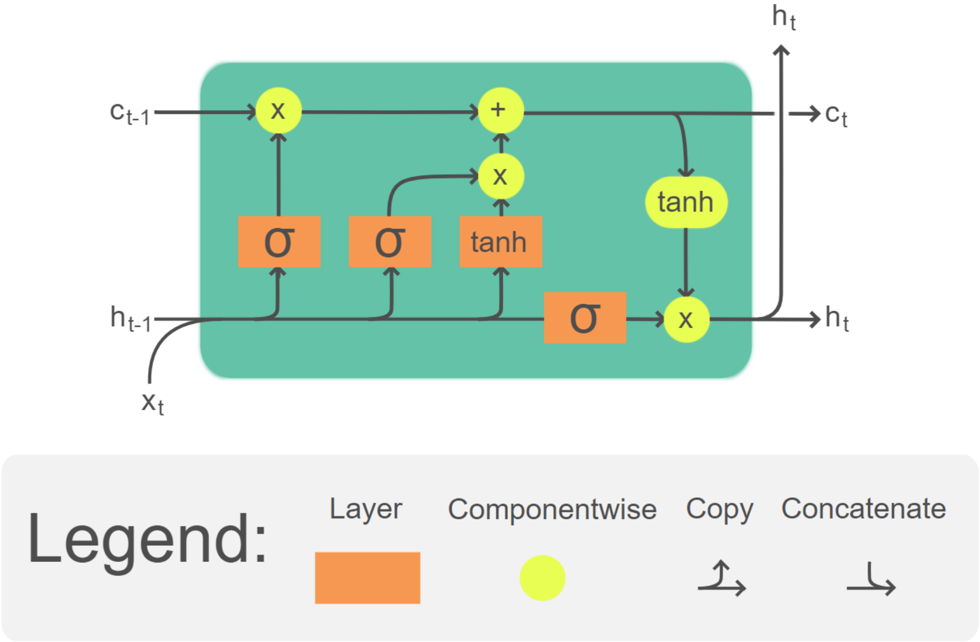
Gated Recurrent Units (GRUs)

- One type of gated RNN
  - Here z is the "forget gate" vector
  - Where z_i = 0:
    - Forget this neuron
    - Allow updating its value
  - Where z_i = 1:
    - Don't forget this neuron
    - Do not allow updating its value
- Parameters: *W*, *U*, plus parameters of *g*
  - (*g* has a slightly complicated form not shown, has its own parameters)

$$h_t = h_{t-1} \overbrace{\odot\, z}^{\text{"forget gate"}} + \overbrace{g(x_t, h_{t-1}) \odot (1 - z)}^{\text{Additive relationship}}$$

$$z = \underbrace{\sigma(Wx_t + Uh_{t-1})}_{\text{Sigmoid ensures gate is between 0 and 1}}$$

Long Short-Term Memory (LSTM)

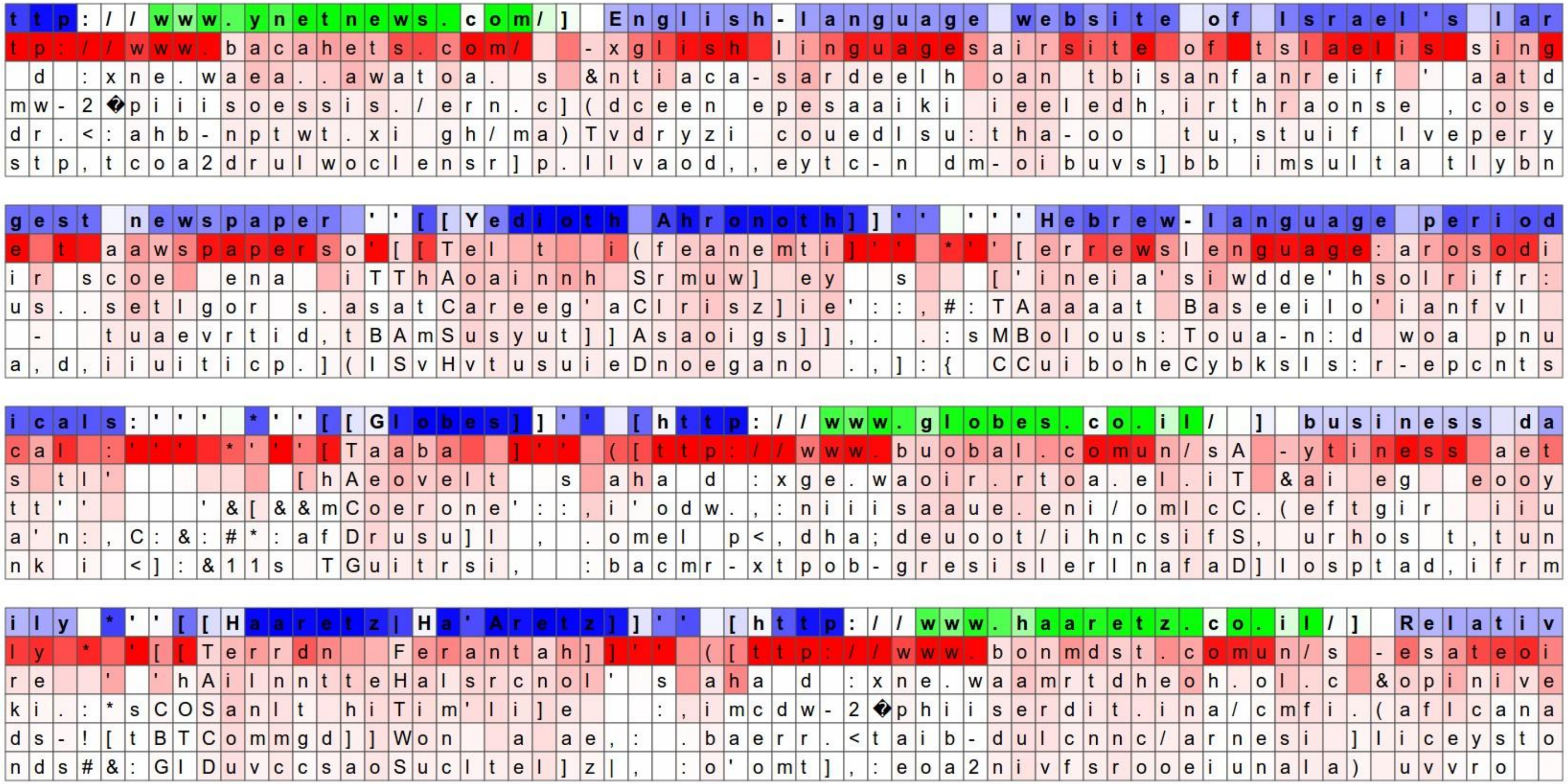


$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$

- Another, more complicated gated RNN
- Commonly used in practice
- What's the idea?
  - Split the hidden state into normal hidden state h_t and "cell" state c_t
  - Cell state uses gated recurrence
  - Hidden state is gated function of cell state

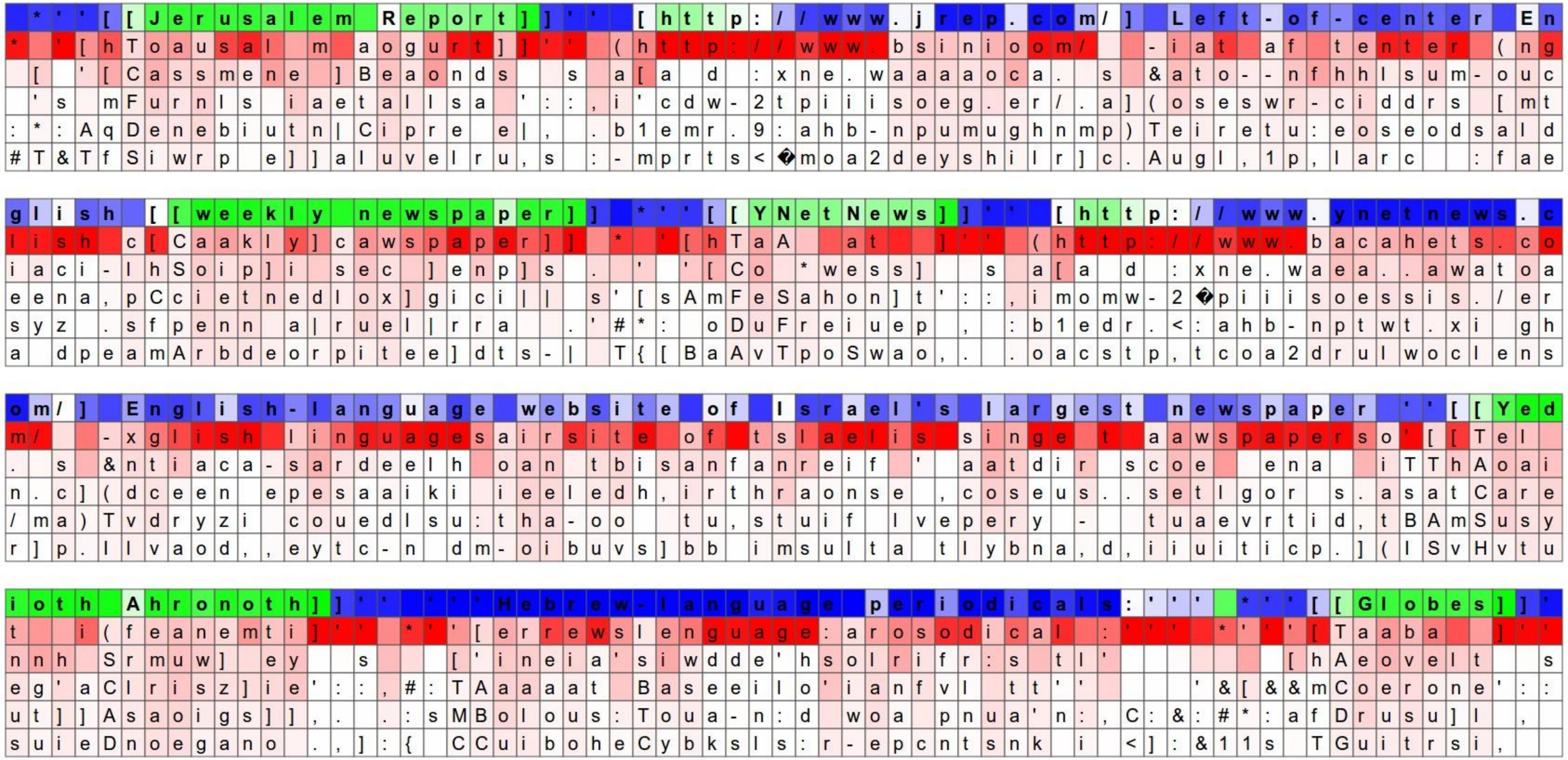
What do LSTMs learn?

- Here: a character-level LSTM (not word-level)
- Blue/Green: Low/high values of 1 neuron
- Below: Top-5 predictions for next character
- This neuron seems to detect whether we're inside a URL



What do LSTMs learn?

- Here: a character-level LSTM (not word-level)
- Blue/Green: Low/high values of 1 neuron
- Below: Top-5 predictions for next character
- This neuron fires only inside a Markdown [[link]] (so it knows when to close the square brackets?)



Conclusion

- Deep Learning for Language must deal with possibly long inputs
- RNNs handle arbitrarily long inputs with fixed number of parameters
- Need to handle long-range dependencies, but hard to learn due to vanishing gradients
- Gated RNNs (GRUs, LSTMs) can reduce vanishing gradient problems