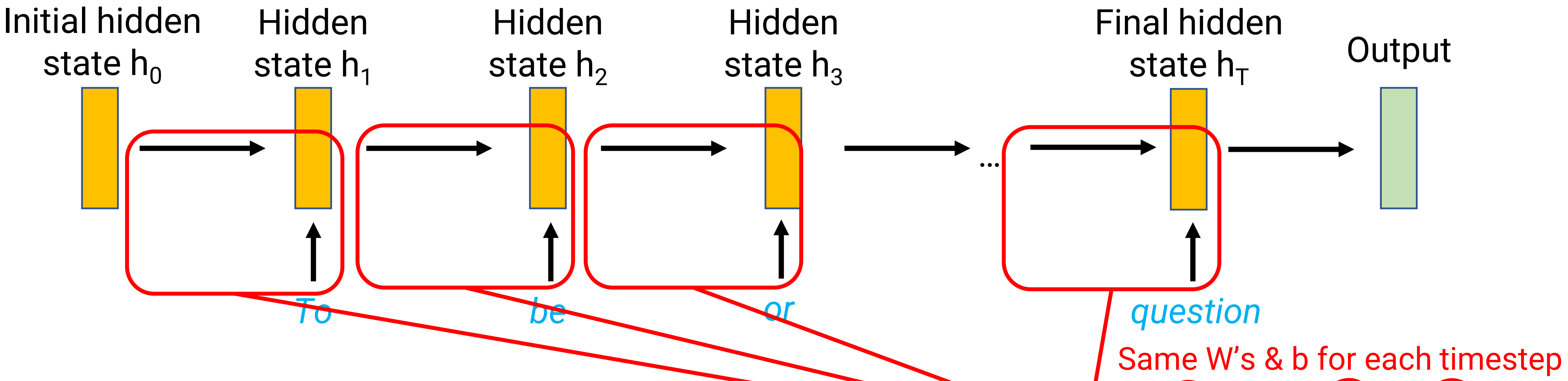
Deep Learning for Language: GRUs/LSTMs, Attention

Robin Jia
USC CSCI 467, Spring 2024
February 29, 2024

Review: “Vanilla”/”Elman” RNN



$$h_t = \tanh\left(\underbrace{W_h h_{t-1}}_{\text{Linear function of prev. hidden state}} + \underbrace{W_x x_t}_{\text{Linear function of current word vector}} + b\right)$$

- 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

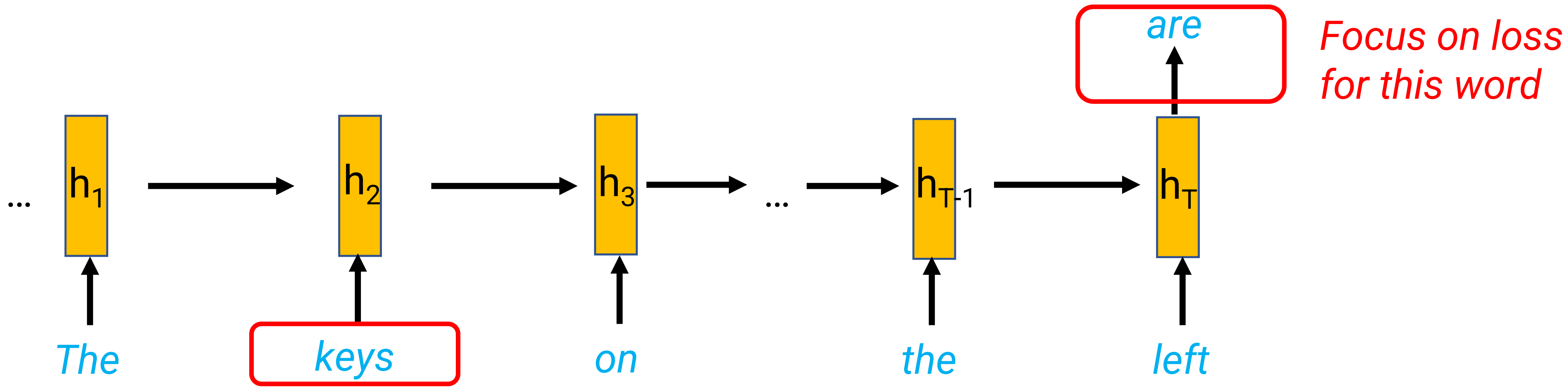
Review: Long-Range Dependencies

- 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)*

Review: Vanishing Gradient Problem

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



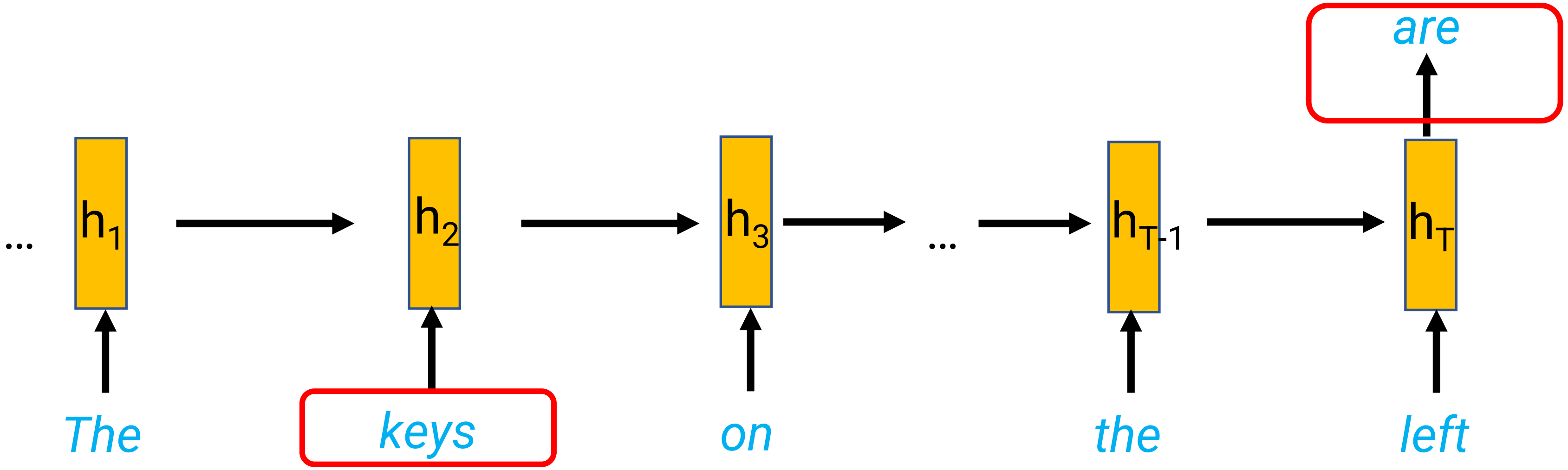
Outline

- Reducing the effect of vanishing gradients
- Sequence-to-sequence learning
- Attention

Avoiding Vanishing Gradients

Where did we go wrong?

$$h_t = \tanh\left(\underbrace{W_h h_{t-1}}_{\text{Multiplicative relationship between previous state and next state}} + W_x x_t + b\right), \quad \frac{\delta h_t}{\delta h_{t-1}} = \tanh'\left(W_h h_{t-1} + W_x x_t + b\right) \underbrace{\cdot\, W_h}_{\text{Leads to repeated multiplication by } W_h}$$



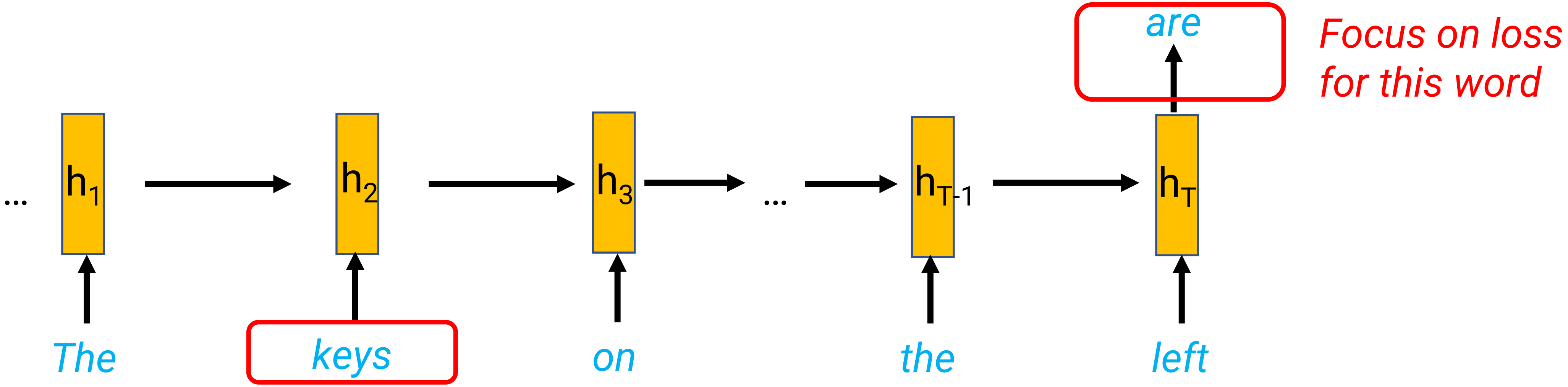
Focus on loss for this word

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

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

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

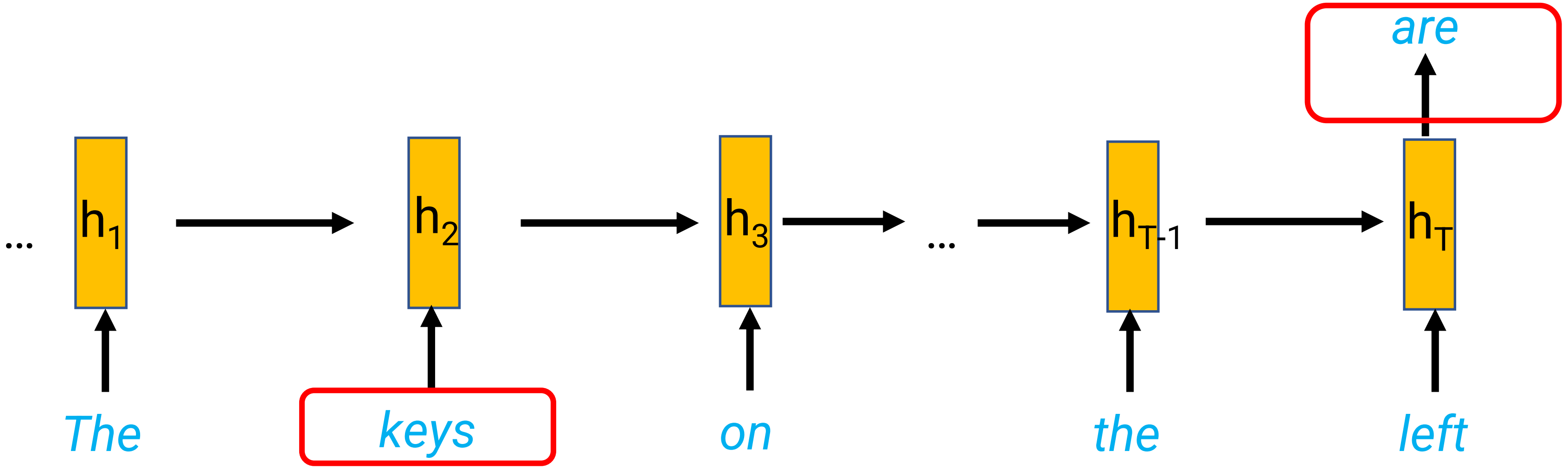


Avoiding Vanishing Gradients

- 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)

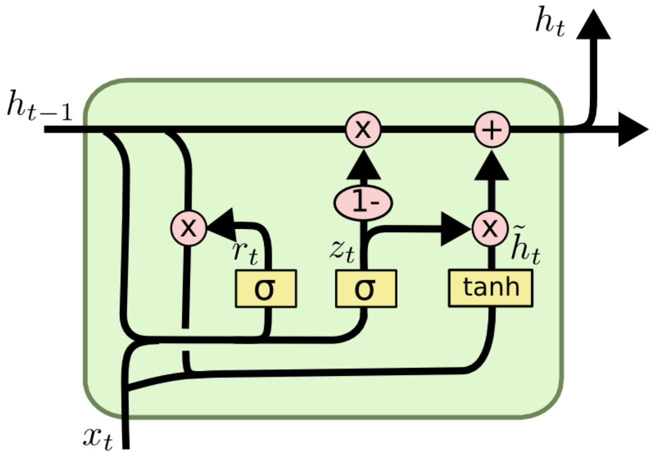
- One type of gated RNN
  - Here z_t is the "forget gate" vector
  - If z_{ti} = 1:
    - Forget the *i*-th neuron
    - Allow updating its value to $\tilde{h}_{ti}$, computed from r_{ti}
  - If z_{ti} = 0:
    - Don't forget the *i*-th neuron
    - Do not allow updating its value
    - Additive relationship between h_{t-1} and h_t
  - Parameters: W_z, W_r, W

Forget gate $$z_t = \sigma\overbrace{(W_z \cdot [h_{t-1}, x_t])}^{\text{Sigmoid ensures gate is between 0 and 1}}$$

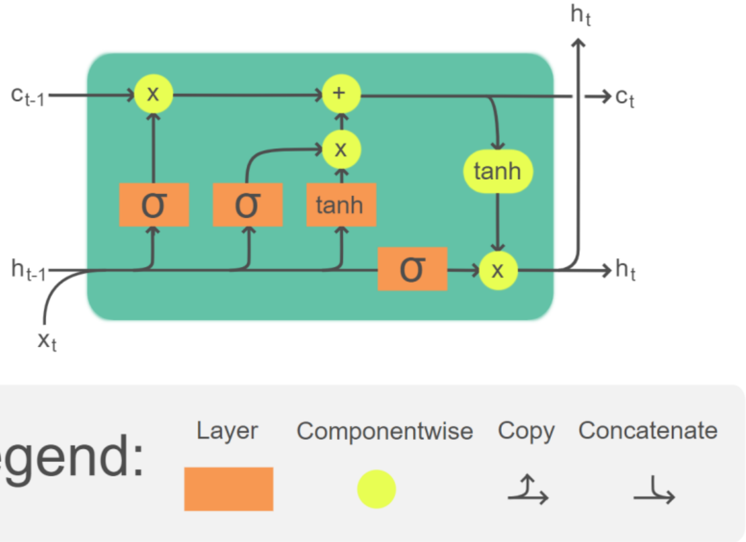
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

Planned update to h_t $$\tilde{h}_t = \tanh(W \cdot [r_t \odot h_{t-1}, x_t])$$

Actual update to h_t $$h_t = \underbrace{(1 - z_t) \odot h_{t-1}}_{\text{Forget parts of } h_{t-1}} + \underbrace{z_t \odot \tilde{h}_t}_{\text{Add update to parts that were forgotten}}$$



Long Short-Term Memory (LSTM)



Forget gate $f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

$$o_t = \sigma(W_o x_t + Uoh_{t-1} + b_o)$$

Planned update to c_t $\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$

Cell state $c_t = f_t \odot c_t + i_t \odot \tilde{c}_t$

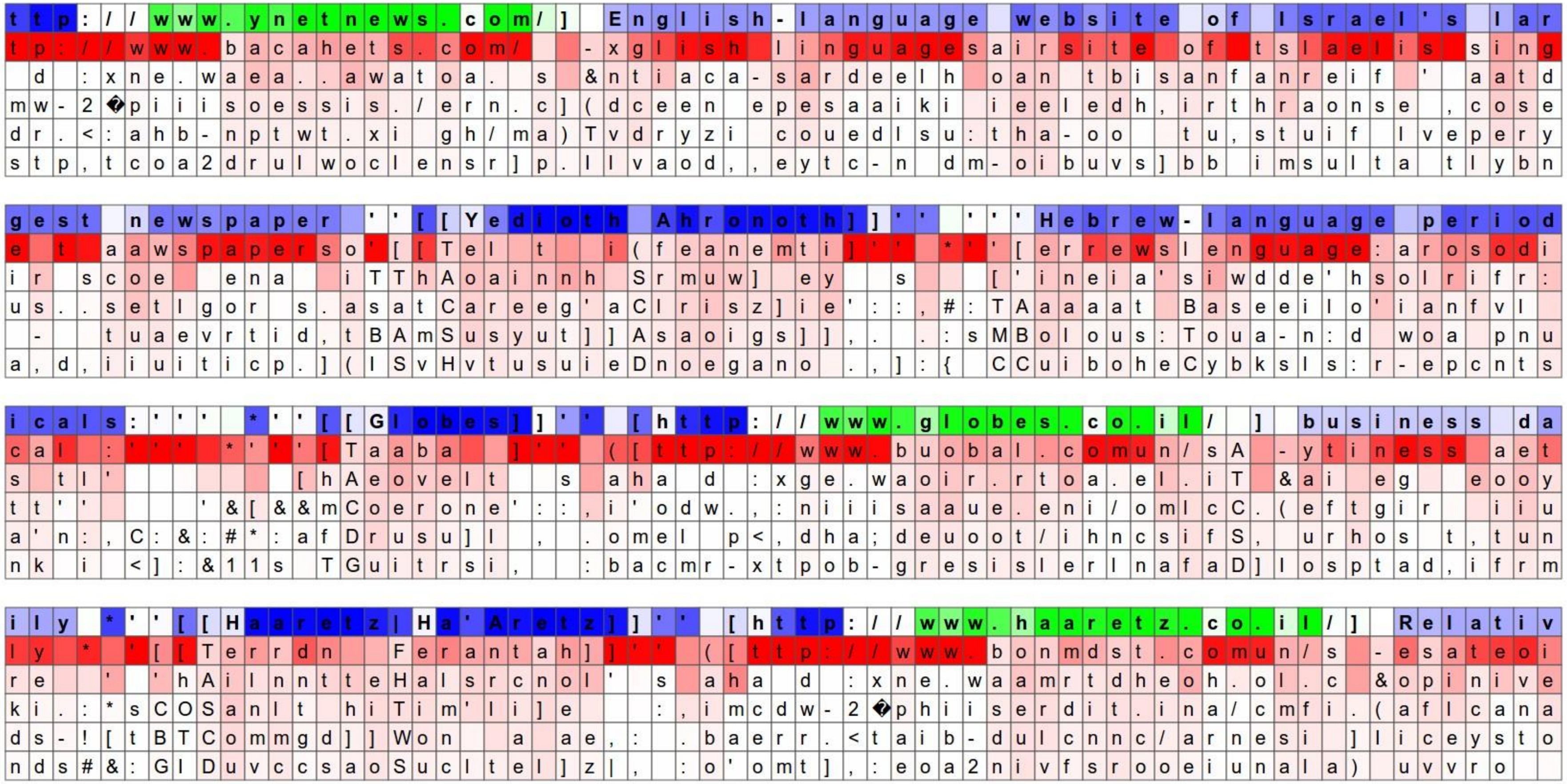
Hidden state $h_t = o_t \odot \tanh(c_t)$

Add the previous cell state * forget gate

- Another, more complicated gated RNN
- Commonly used in practice
- Overall idea:
  - Split the hidden state into normal hidden state h_t and “cell” state c_t
  - Cell state uses gated recurrence with forget gate f_t
  - Hidden state is gated function of cell state
  - Also has input and output gates i_t & o_t

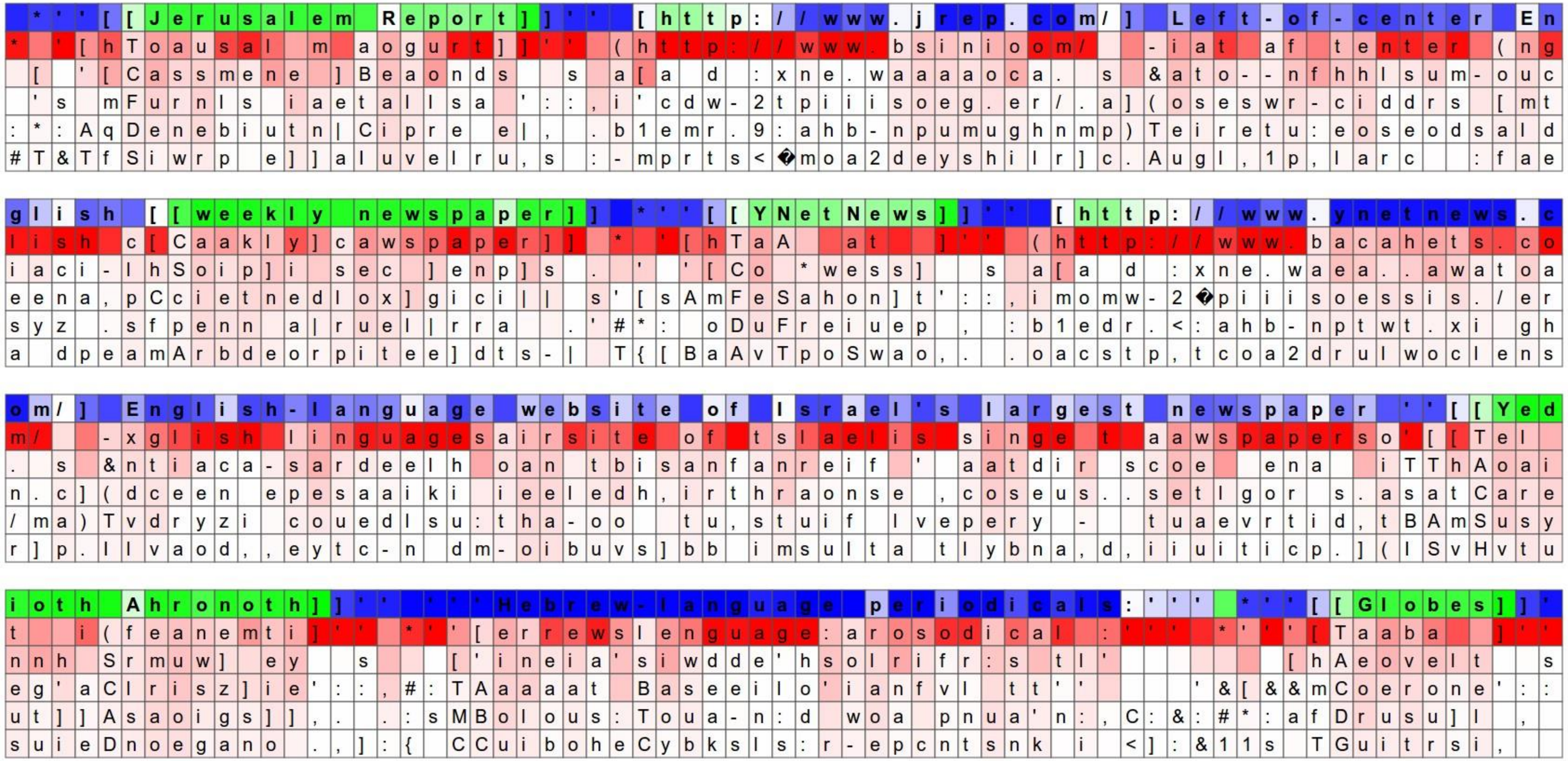
What do LSTMs learn?

- 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?

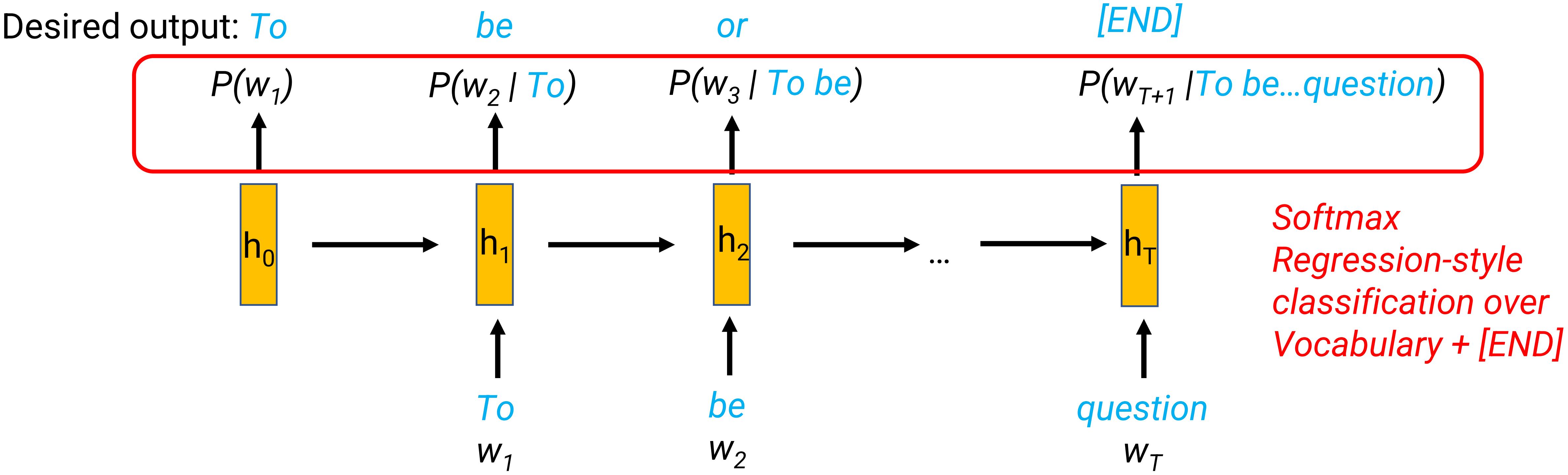
- Here: a character-level LSTM (not word-level)
- Blue/Green: Low/high values of 1 neuron
- Below: Top-5 predictions for next character
- This neuron fires only inside a Markdown [[link]] (so it knows when to close the square brackets?)



Outline

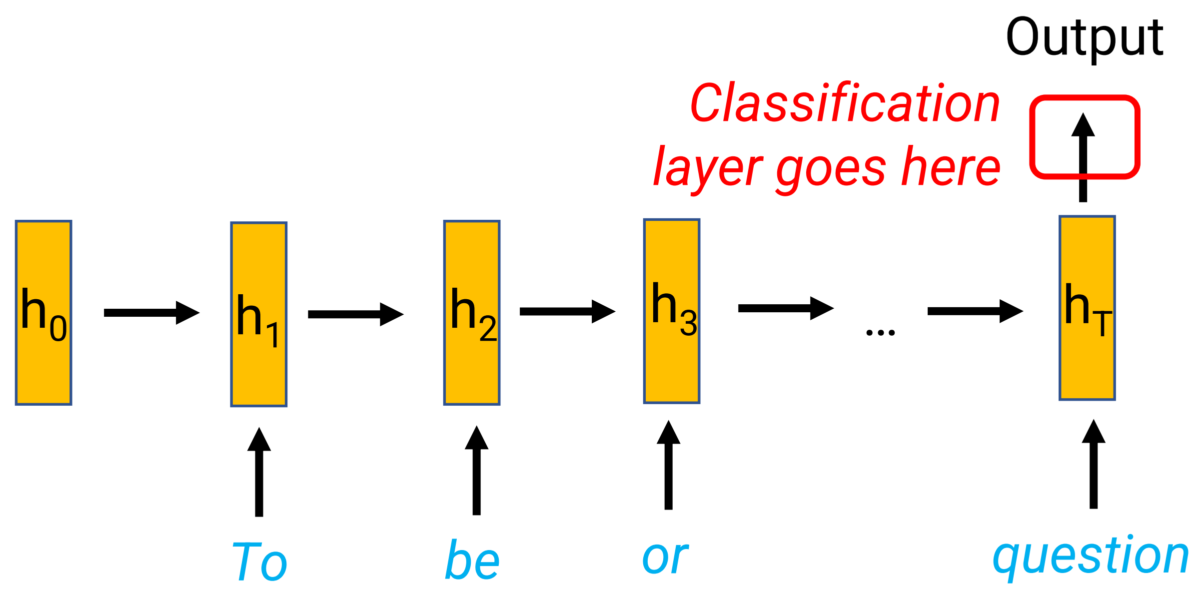
- Reducing the effect of vanishing gradients
- Sequence-to-sequence learning
- Attention

Review: Autoregressive Language Modeling



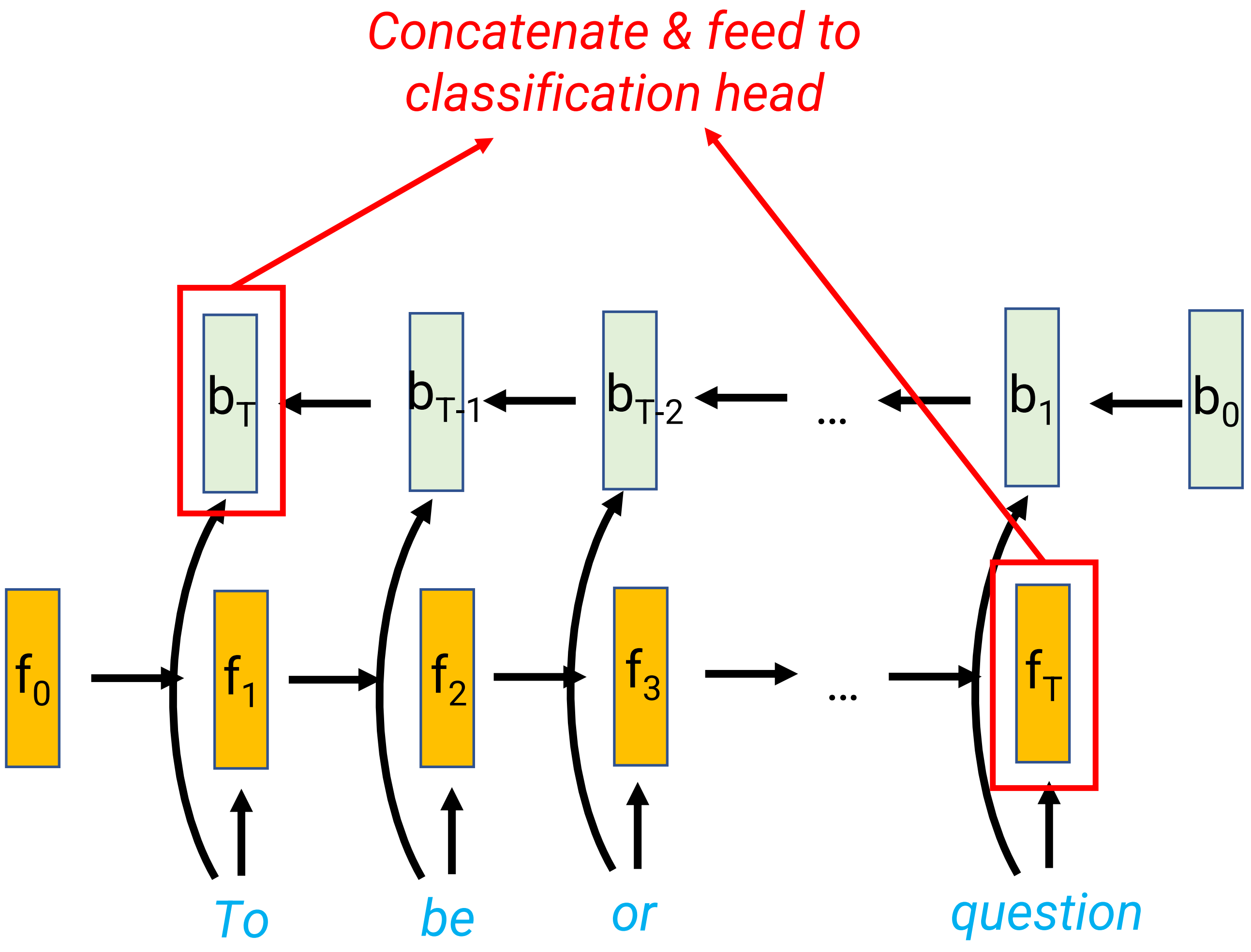
- At each step, probabilistically predict the next word given current hidden state
- One step’s desired output is the next step’s input (“autoregressive”)
- To mark end of sequence, model should predict the *[END]* token
- Called a “Decoder”: Looks at the hidden state and “decodes” next word

Text classification ("Encoder only")



- First run an RNN over text
- Use the final hidden state as an "encoding" of the entire sequence
- Use this as features, train a classifier on top
- Downside: Later words processed better than early words (long range dependency issues)

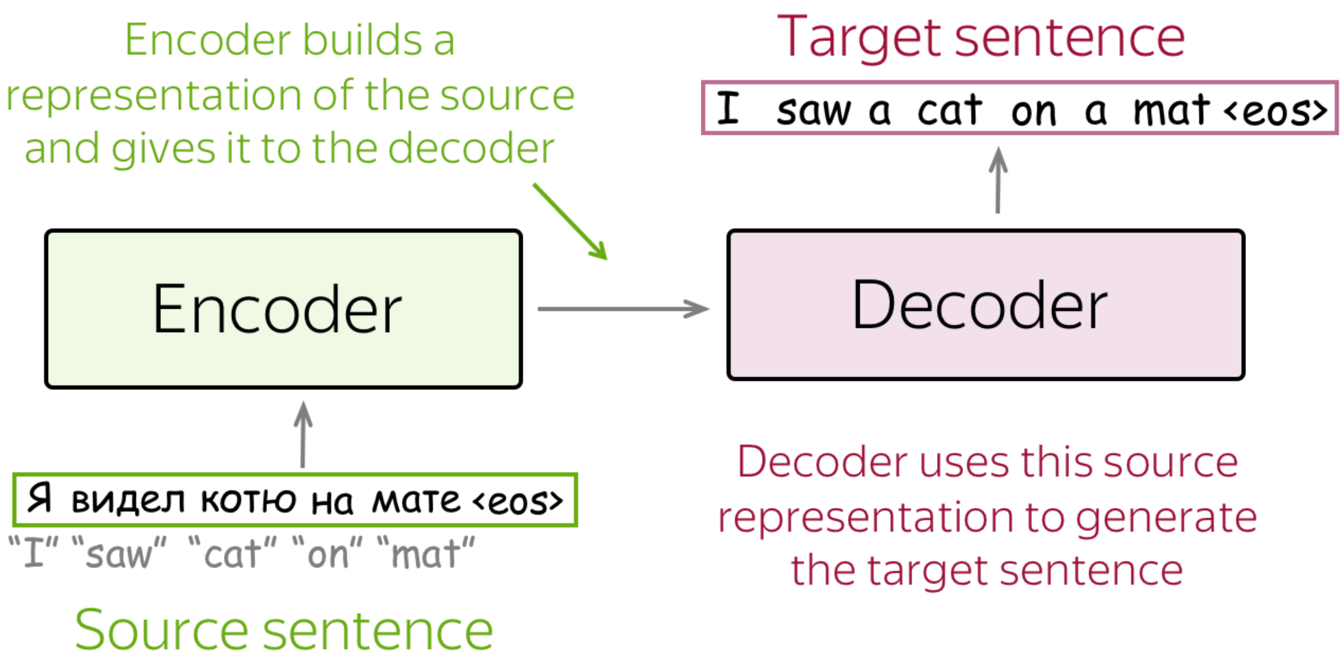
Bi-directional encoders



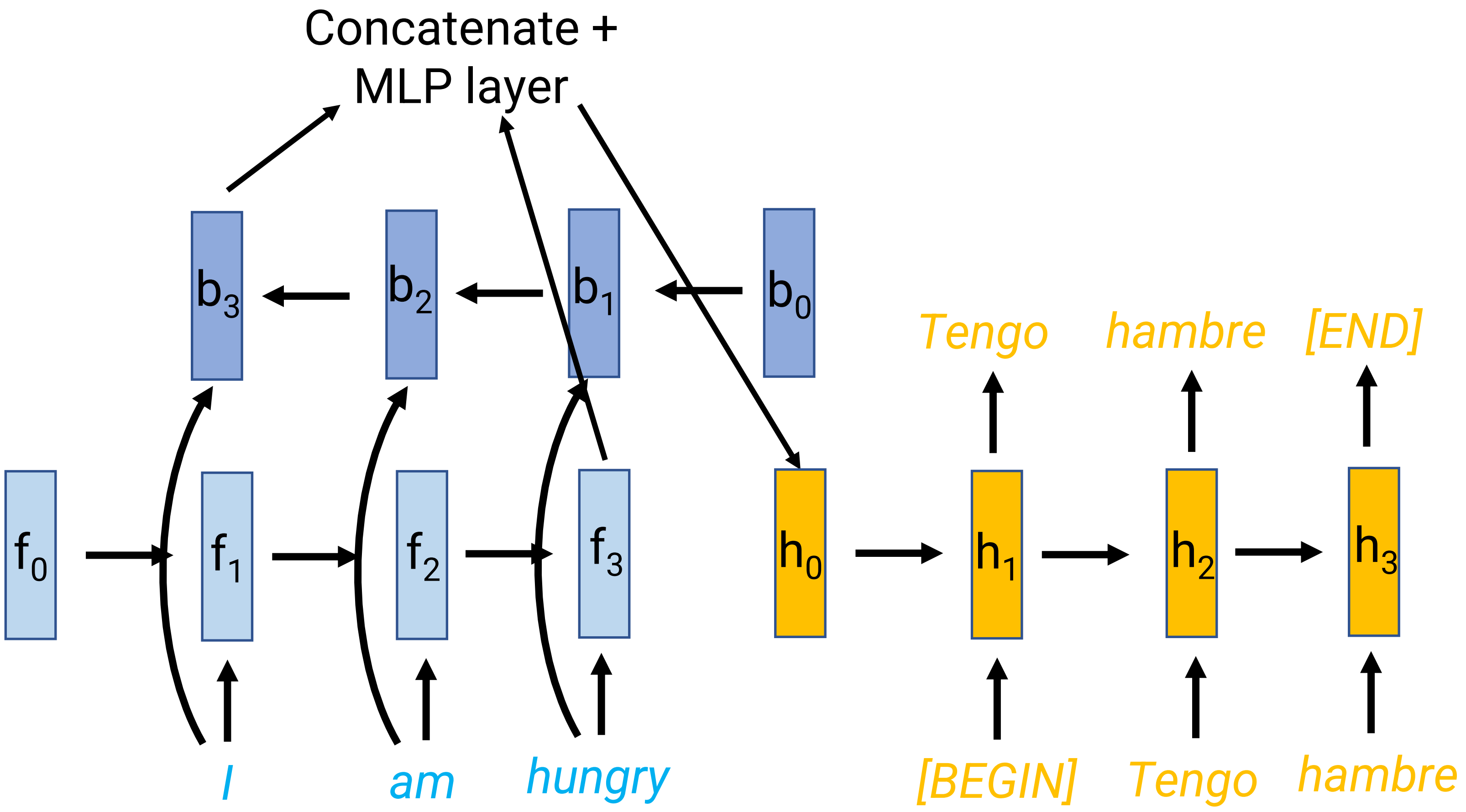
- Run one RNN left-to-right, and another one right-to-left
    - (I'll call forward-direction hidden states f_t, backward-direction hidden states b_t)
- Concatenate the 2 final hidden states as final representation
    - Note: This encoding is twice as large now—we've doubled the number of features passed to the final classifier

Sequence-to-sequence Tasks

- **Sequence-to-sequence** tasks
  - Machine translation (Russian -> English)
  - Summarization (Document -> Summary)
  - Personal Assistants (Command -> Action)
- Encoder: "Reads" the input sentence, produces a feature vector summarizing the input
- Decoder: Uses that vector as its initial state, predicts output tokens one at a time



Encoder-decoder model



- Example: Machine Translation
  - Input = English text
  - Output = Spanish text
- Encoder: Process English sentence into vector
  - E.g. Bidirectional encoder + MLP layer to generate decoder's initial state
- Decoder: Use vector as initial hidden state and start doing language modeling in Spanish
- Vector space acts as a "shared language"

The Power of Building Blocks

- We now know about a lot of components
- We can assemble in any way we think makes sense, given the input and desired output
- We only have to think about the forward pass!
- Code to learn parameters is always the same:
  - Get a batch of training examples
  - Compute the loss (forward pass)
  - Run backpropagation to get gradient of loss w.r.t. parameters
  - Gradient descent to update parameters

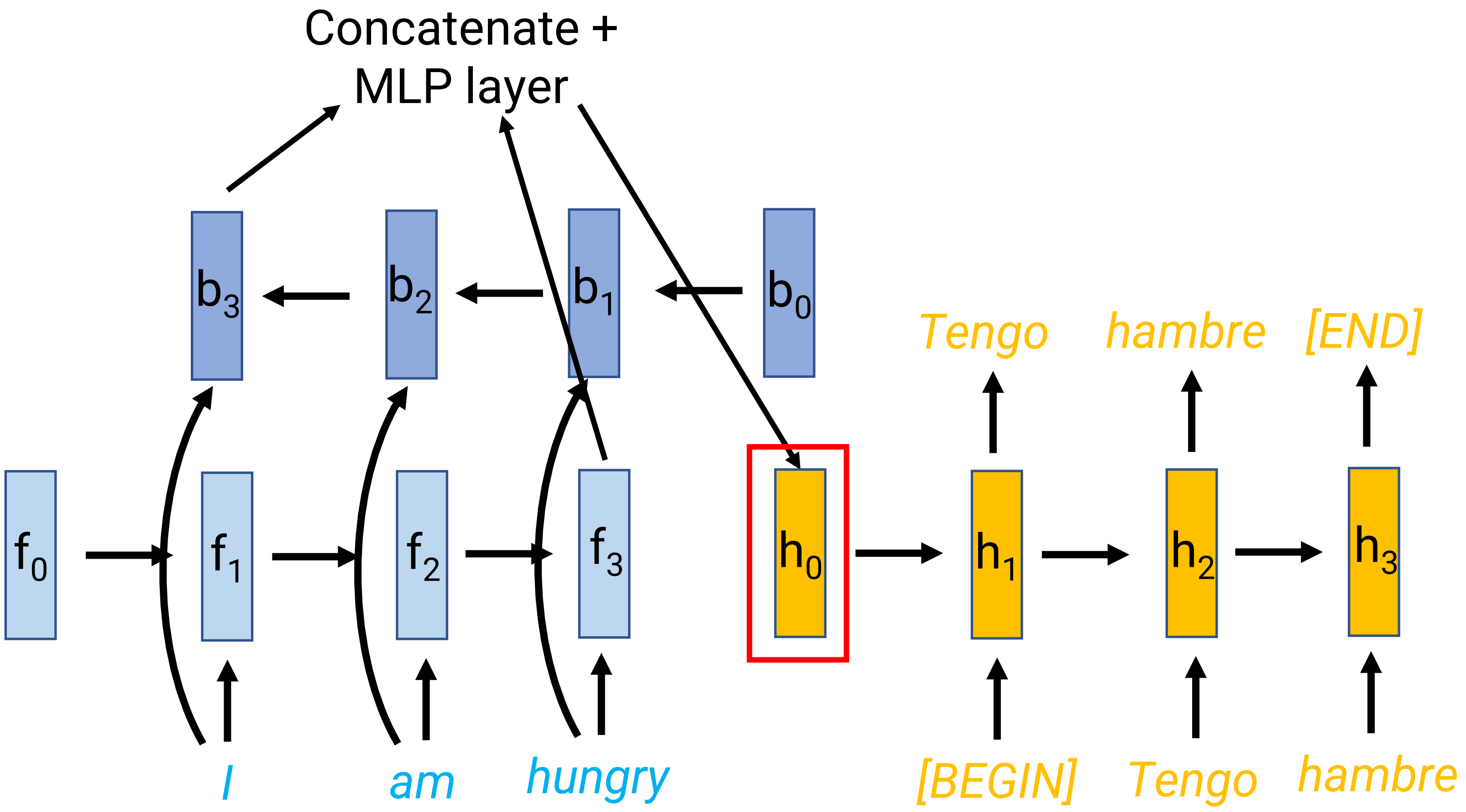
Announcements

- HW2 due today @ 11:59pm
- Section Friday: Midterm Review (practice exam + questions)
- Midterm exam: Thursday March 7, SLH 100
  - Practice exams released on website
  - Everything through end of today's lecture is fair game
  - Will post spreadsheet of lecture video links on Piazza

Outline

- Reducing the effect of vanishing gradients
- Sequence-to-sequence learning
- Attention

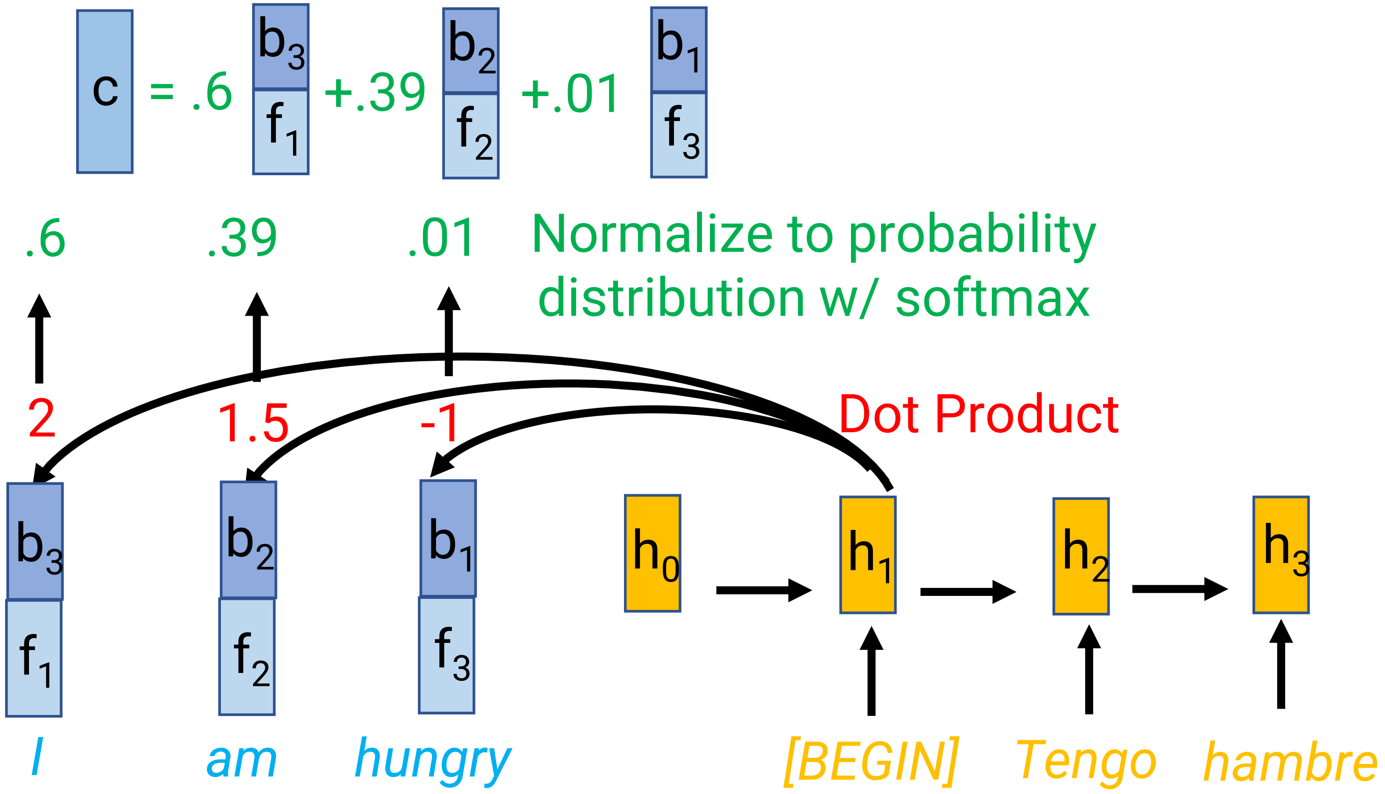
What’s missing? Alignment



- Challenge: The single encoder output has to store information about the entire sentence in a single vector
- Better strategy: Look for the next input word to translate, then translate that word
- Traditional MT: Alignment between input & output sentences
- Can we get a neural network to model alignments?

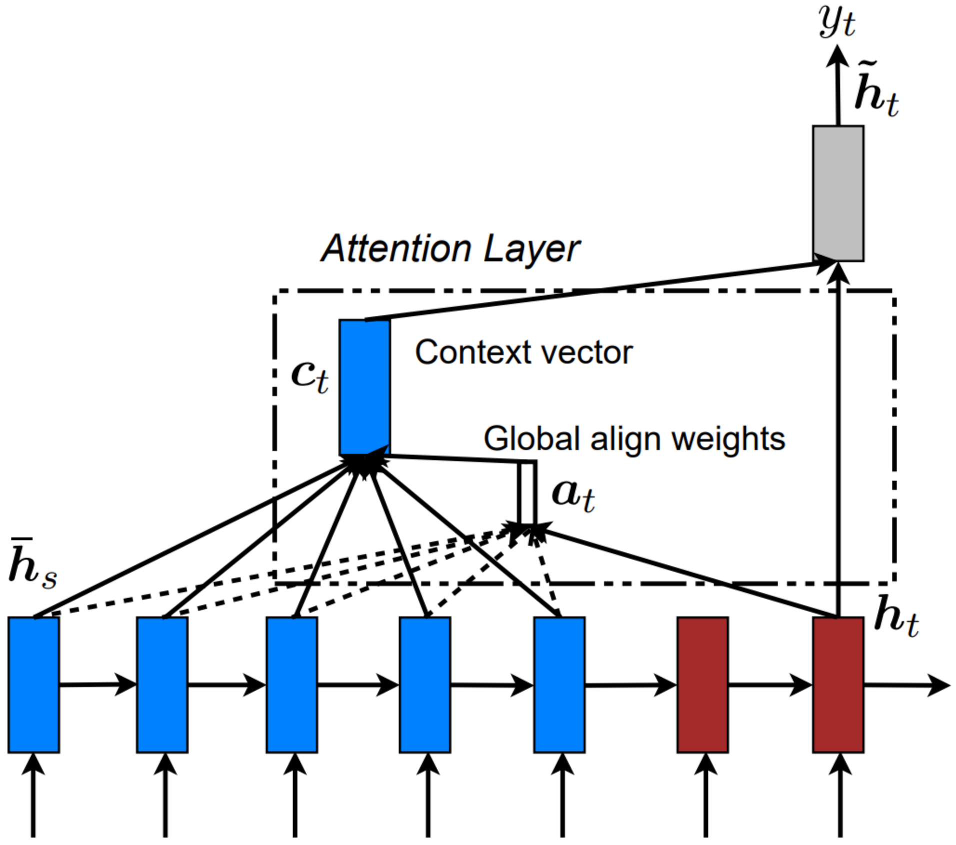
Attention

- Compute similarity between decoder hidden state and each encoder hidden state
  - E.g., dot product, if same size
- Normalize similarities to probability distribution with softmax
- Output: "Context" vector *c* = weighted average of encoder states based on the probabilities
  - No new parameters (like ReLU/max pool)
- Use *c* when computing decoder outputs or transitions
- Intuition
  - Step 1: Find similar input words
  - Step 2: Grab the encoder representation of those words
  - Step 3: Tell the decoder that this is relevant



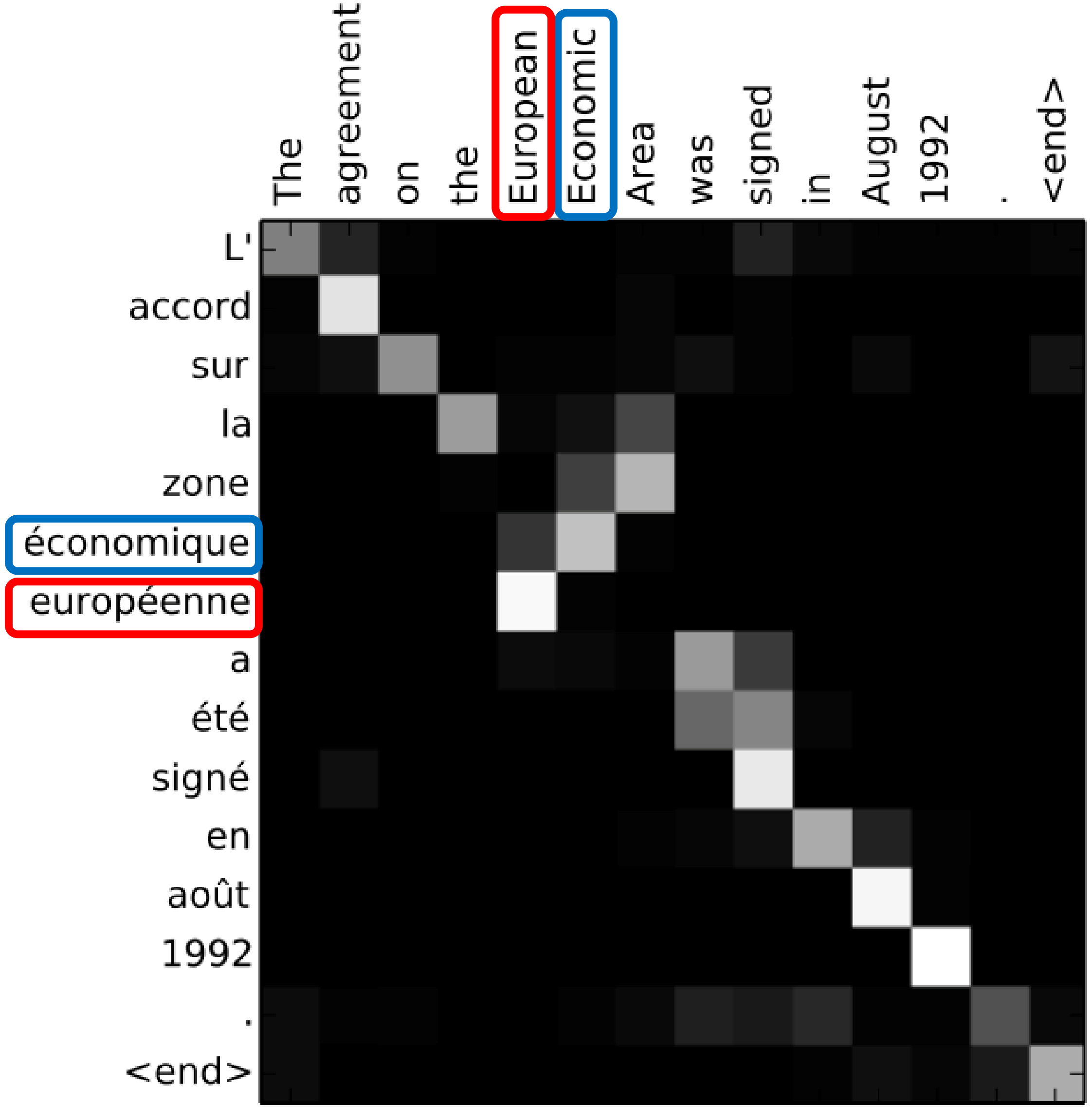
Using Attention in Seq-to-seq model

- Many similar ways one could implement an attention mechanism
- Example from a well-known 2015 paper by Luong et al. on machine translation
  - Blue = encoder states
  - Red = decoder states
  - Note: Encoder was unidirectional here
- Dot-product decoder state h_t with encoder states, then apply softmax to produce weights a_t
- Weighted sum of encoder states yields context vector c_t
- Context vector c_t concatenated with decoder state h_t, fed through 1 MLP layer to generate $\tilde{h}_t$
- $\tilde{h}_t$ used to make prediction y_t



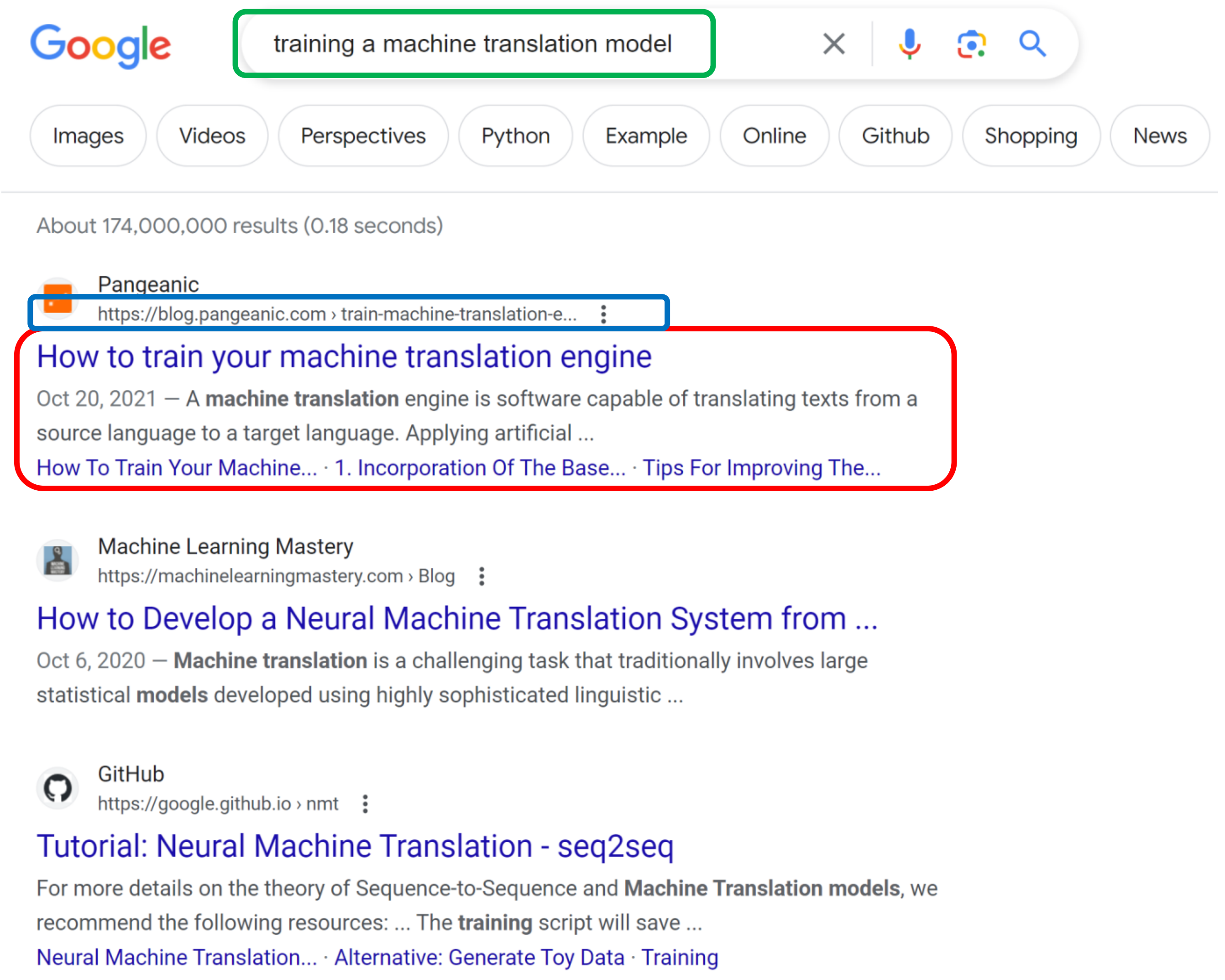
Luong, Pham, and Manning. Effective Approaches to Attention-based Neural Machine Translation. EMNLP 2015.

Visualizing attention



- Source is English, Target is French
- Each row is a probability distribution over the English text
- Alignment makes sense, overcomes word order differences
  - When generating "économique" attend to "Economic"
  - When generating "européenne" attend to "European"

Attention as Retrieval



- Consider a search engine:
  - Queries: What you are looking for
    - E.g., What you type into Google search
  - Keys: Summary of what information is there
    - E.g., Text from each webpage
  - Values: What to give the user
    - E.g., The URL of each webpage

General Form of Attention

(8) Attention Layer

- Inputs (all vectors of length *d*):
  - Query vector q
  - Key vectors $k_1, \ldots, k_T$
  - Value vectors $v_1, \ldots, v_T$
- Output (also vector of length *d*)

How well does the query match each key?

  - Dot product q with each key vector k_t to get score s_t:

$$s_t = q^\top k_t$$

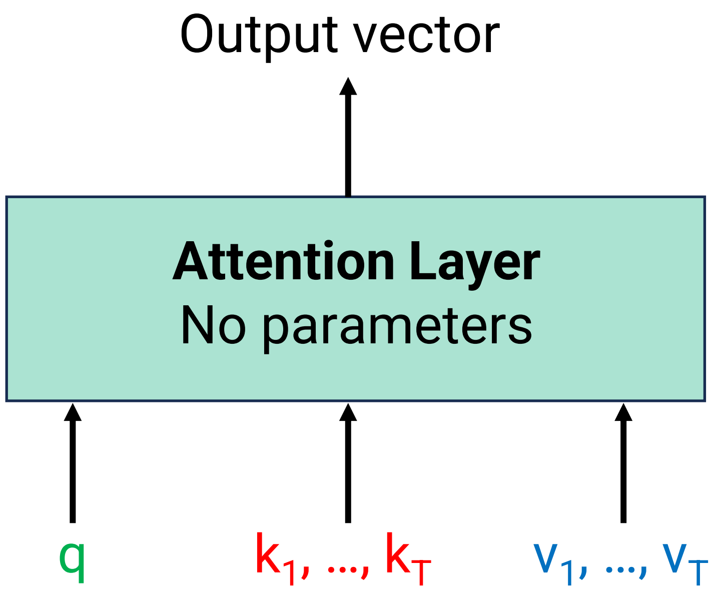
  - Softmax to get probability distribution $p_1, \ldots, p_T$:

$$p_t = \frac{e^{s_t}}{\sum_{j=1}^{T} e^{s_j}}$$

  - Return weighted average of value vectors:

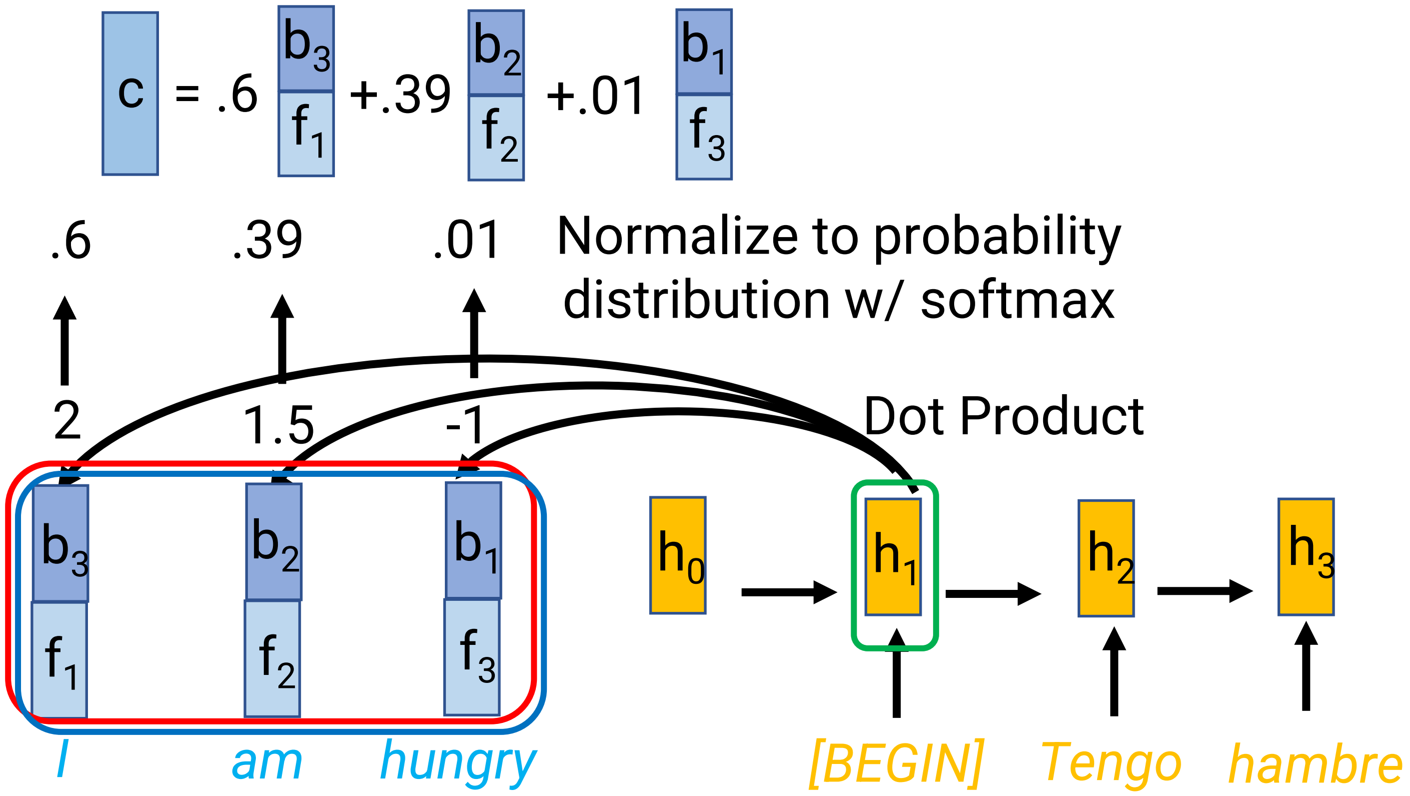
$$\sum_{t=1}^{T} p_t v_t$$

Dominated by the values corresponding to the "best-matching" keys



Attention in Seq-to-seq RNNs

- Applies a general attention layer where:
  - Query = Current decoder hidden state
  - Keys = Encoder hidden states
  - Values = Encoder hidden states (same as keys)



Conclusion

- GRUs, LSTMs: Add gates + additive connections to reduce vanishing gradients
- Ways to use RNNs
  - As a decoder: To generate text
  - As an encoder: To produce feature vectors for text
  - Sequence-to-sequence: Use 2 RNNs, one for each purpose
- Attention: Know which part of the input matters when generating each word of the output
  - After Spring Break: Can we get rid of RNN’s, and only use attention?