LSTMs, GRUs, Encoder-Decoder Models, and Attention

Natalie Parde, Ph.D.

Department of Computer Science

University of Illinois at Chicago

CS 521: Statistical Natural Language Processing

Spring 2020

Many slides adapted from Jurafsky and Martin (https://web.stanford.edu/~jurafsky/slp3/).

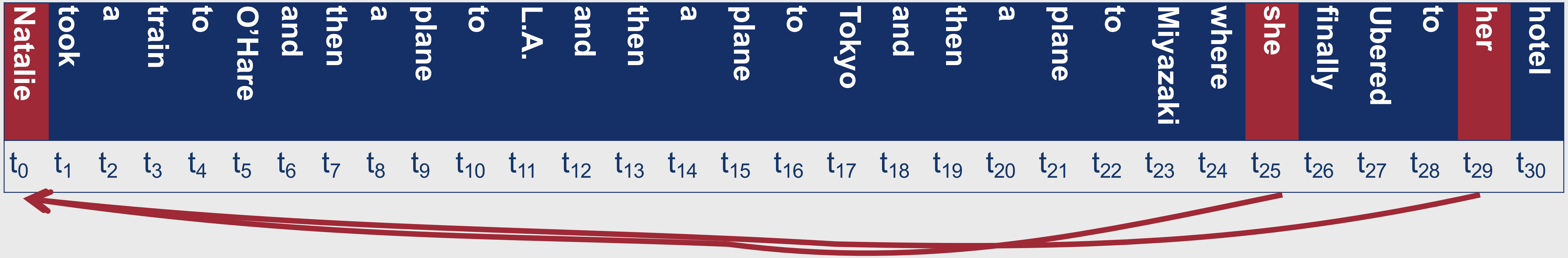
“Vanilla” RNNs hold many advantages over feedforward networks for NLP tasks.

- Temporal context
- Variable-length input

- However …they’re not perfect (no networks are!)

In particular, RNNs may struggle with managing context.

- In a simple RNN, the final state tends to reflect more information about **recent items** than those at the beginning of the sequence
- **Distant timesteps → less information**



This long-distance information can be critical to many tasks!

Why is it so hard to maintain long-distance context?

- Hidden layers must perform two tasks simultaneously:
  - Provide information useful for the current decision (input at *t*)
  - Update and carry forward information required for future decisions (input at time *t*+1 and beyond)
- These tasks may not always be perfectly aligned with one another

There's also the issue of "vanishing gradients"….

- When small derivatives are repeatedly multiplied together, the products can become extremely small
- This means that when backpropagating through time for a long sequence, gradients can become so close to zero that they are no longer effective for model training!

How can we address this?

- Design more complex RNNs that learn to:
  - **Forget** information that is no longer needed
  - **Remember** information still required for future decisions

Long Short-Term Memory Networks (LSTMs)

- **Remove information** no longer needed from the context, and **add information** likely to be needed later
- Do this by:
  - Adding an **explicit context layer** to the architecture
  - This layer controls the flow of information into and out of network layers using specialized neural units called **gates**

LSTM Gates

- **Feedforward layer** + **sigmoid activation** + **pointwise multiplication** with the layer being gated
- Combination of sigmoid activation and pointwise multiplication essentially creates a **binary mask**
  - Values near 1 in the mask are passed through **nearly unchanged**
  - Values near 0 are **nearly erased**

LSTM Gates

- Three main gates:
  - **Forget gate:** Should we erase this existing information from the context?
  - **Add gate:** Should we write this new information to the context?
  - **Output gate:** What information should be revealed as output for the current hidden state?

Forget Gate

- Goal: Delete information from the context that is no longer needed
  - $f_t = \sigma(\underbrace{U_f h_{t-1} + W_f x_t})$
  - $k_t = c_{t-1} \odot f_t$

Weighted sum of:
- Hidden layer at the previous timestep
- Current input

Forget Gate

- Goal: Delete information from the context that is no longer needed
  - $f_t = \sigma(U_f h_{t-1} + W_f x_t)$
  - $k_t = \underbrace{c_{t-1}}_{\text{Context vector from the previous timestep}} \odot f_t$

Add Gate

- Goal: Select the information to add to the current context
  - $g_t = \underbrace{\tanh(U_g h_{t-1} + W_g x_t)}_{\text{Regular RNN computation}}$
  - $i_t = \sigma(U_i h_{t-1} + W_i x_t)$
  - $j_t = g_t \odot i_t$
  - $c_t = j_t + k_t$

Add Gate

- Goal: Select the information to add to the current context
  - $g_t = \tanh(U_g h_{t-1} + W_g x_t)$
  - $i_t = \sigma(U_i h_{t-1} + W_i x_t)$
  - $j_t = g_t \odot i_t$
  - $c_t = j_t + k_t$

Weighted sum of:
- Hidden layer at the previous timestep
- Current input

Add Gate

- Goal: Select the information to add to the current context
  - $g_t = \tanh(U_g h_{t-1} + W_g x_t)$
  - $i_t = \sigma(U_i h_{t-1} + W_i x_t)$
  - $j_t = \underbrace{g_t \odot i_t}$
  - $c_t = j_t + k_t$

New information to be added

Add Gate

- Goal: Select the information to add to the current context
  - $g_t = \tanh(U_g h_{t-1} + W_g x_t)$
  - $i_t = \sigma(U_i h_{t-1} + W_i x_t)$
  - $j_t = g_t \odot i_t$
  - $c_t = \underbrace{j_t + k_t}$

Updated context vector contains:
- New information to be added
- Existing information from context vector that was not removed by the forget gate

Output Gate

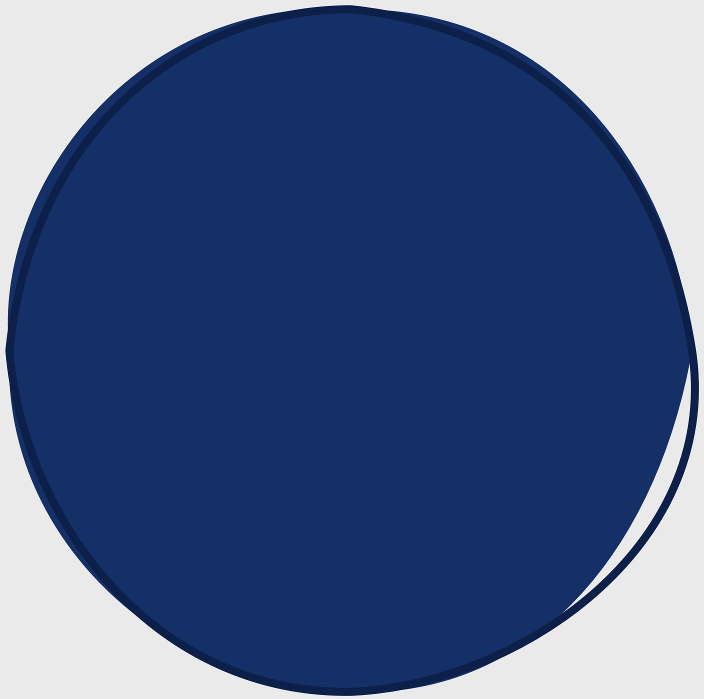
- Goal: Decide what information is required for the *current* hidden state
  - $o_t = \sigma(U_o h_{t-1} + W_o x_t)$
  - $h_t = o_t \odot \tanh(c_t)$

Weighted sum of:
- Hidden layer at the previous timestep
- Current input

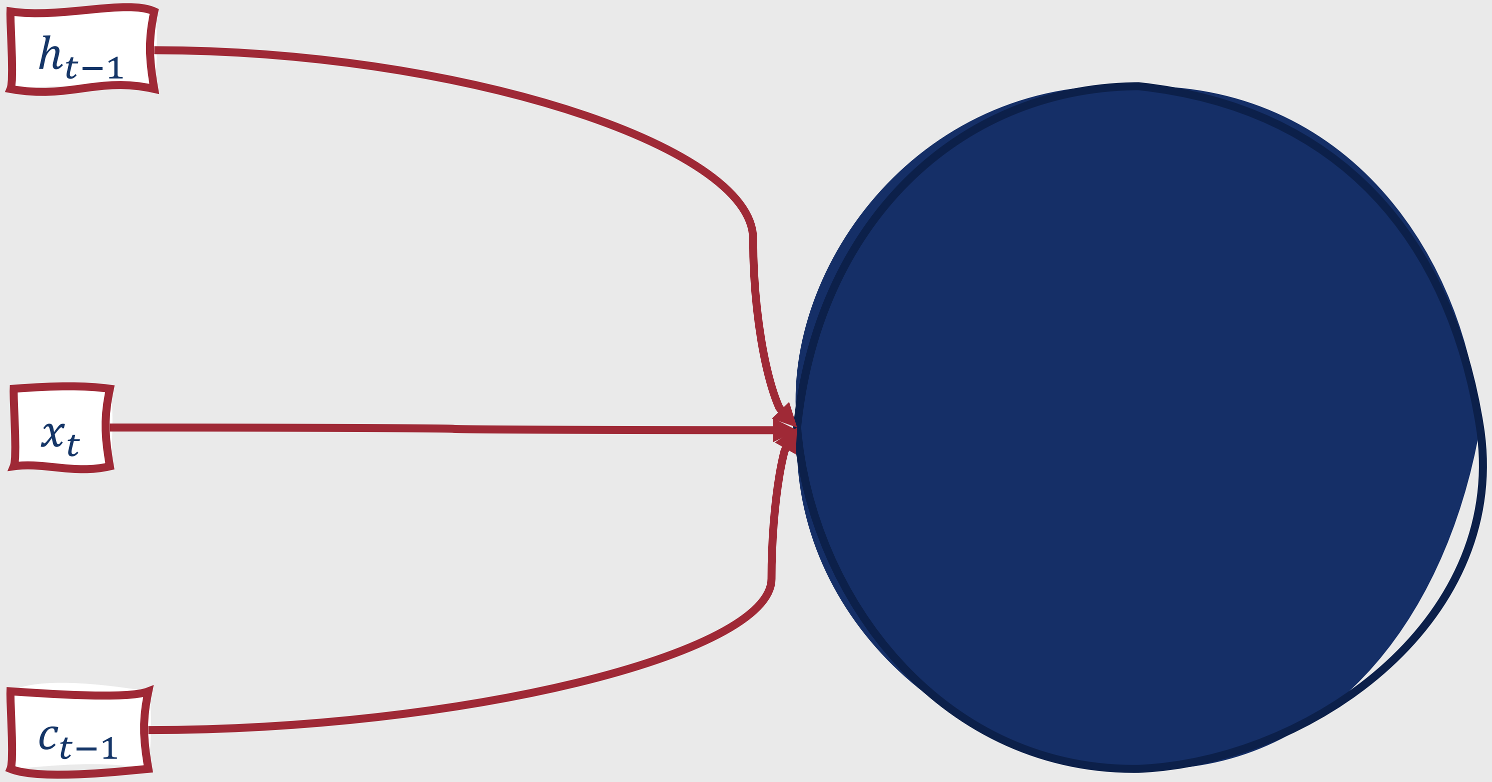
Output Gate

- Goal: Decide what information is required for the *current* hidden state
  - $o_t = \sigma(U_o h_{t-1} + W_o x_t)$
  - $h_t = \underbrace{o_t \odot \tanh(c_t)}_{\text{Updated hidden layer output}}$

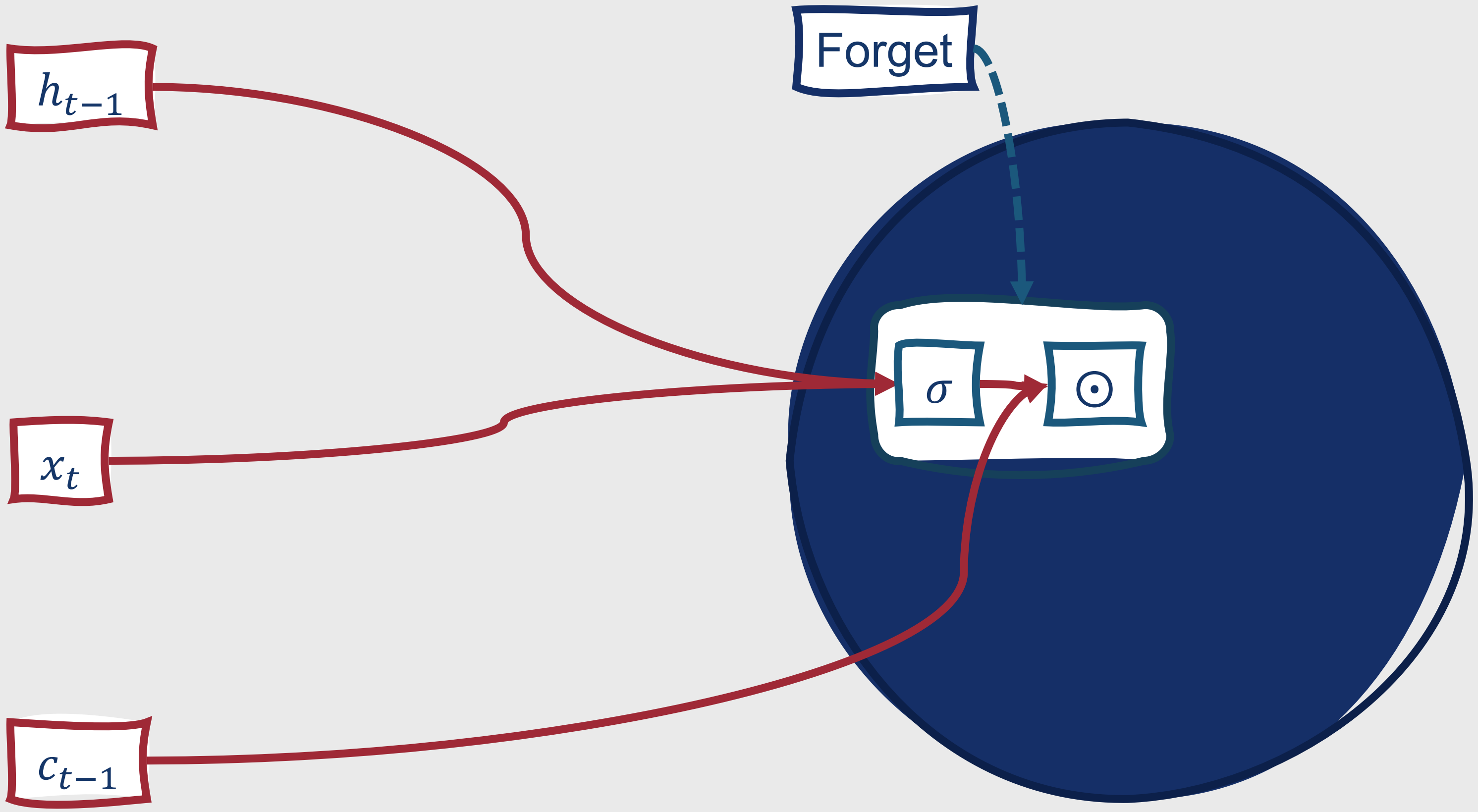
What does this process look like in a single LSTM unit?



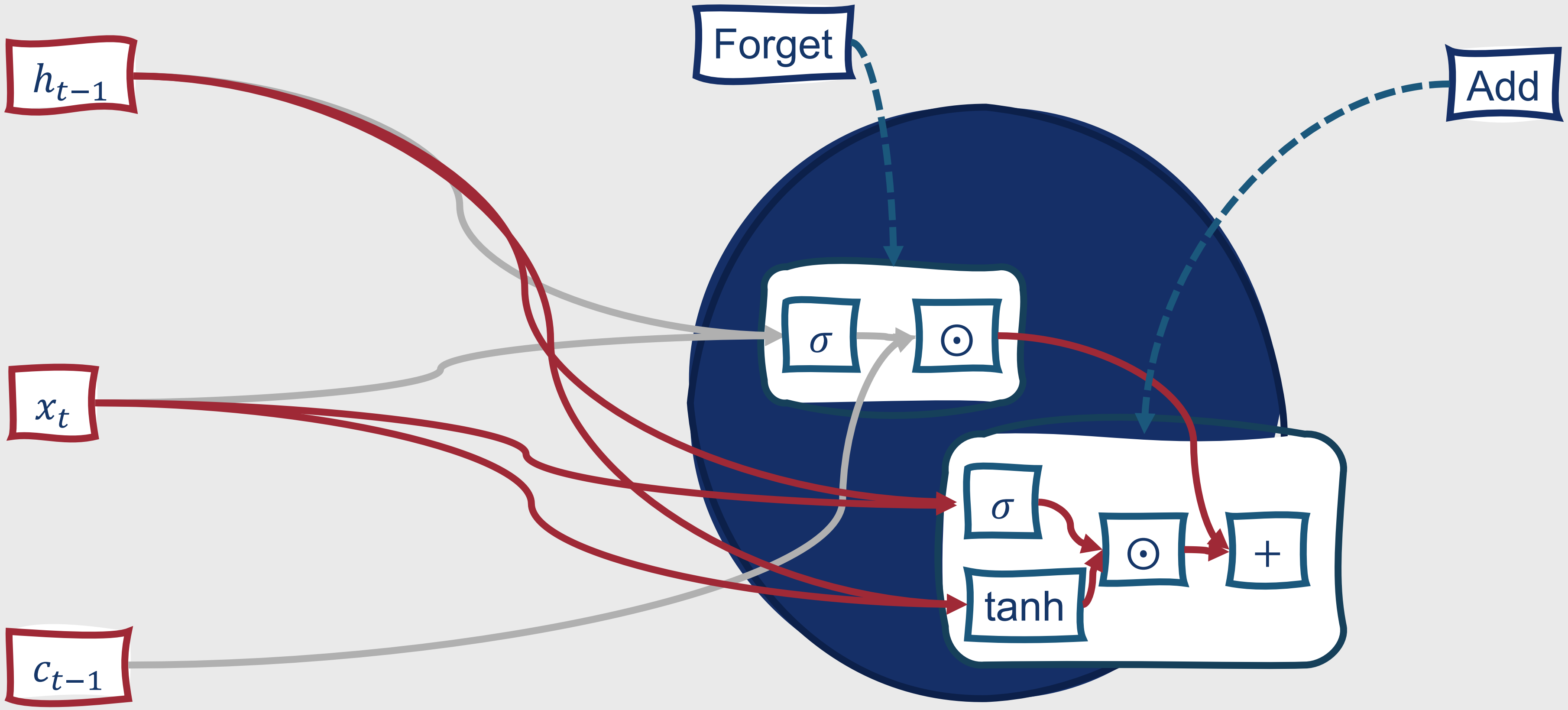
What does this process look like in a single LSTM unit?



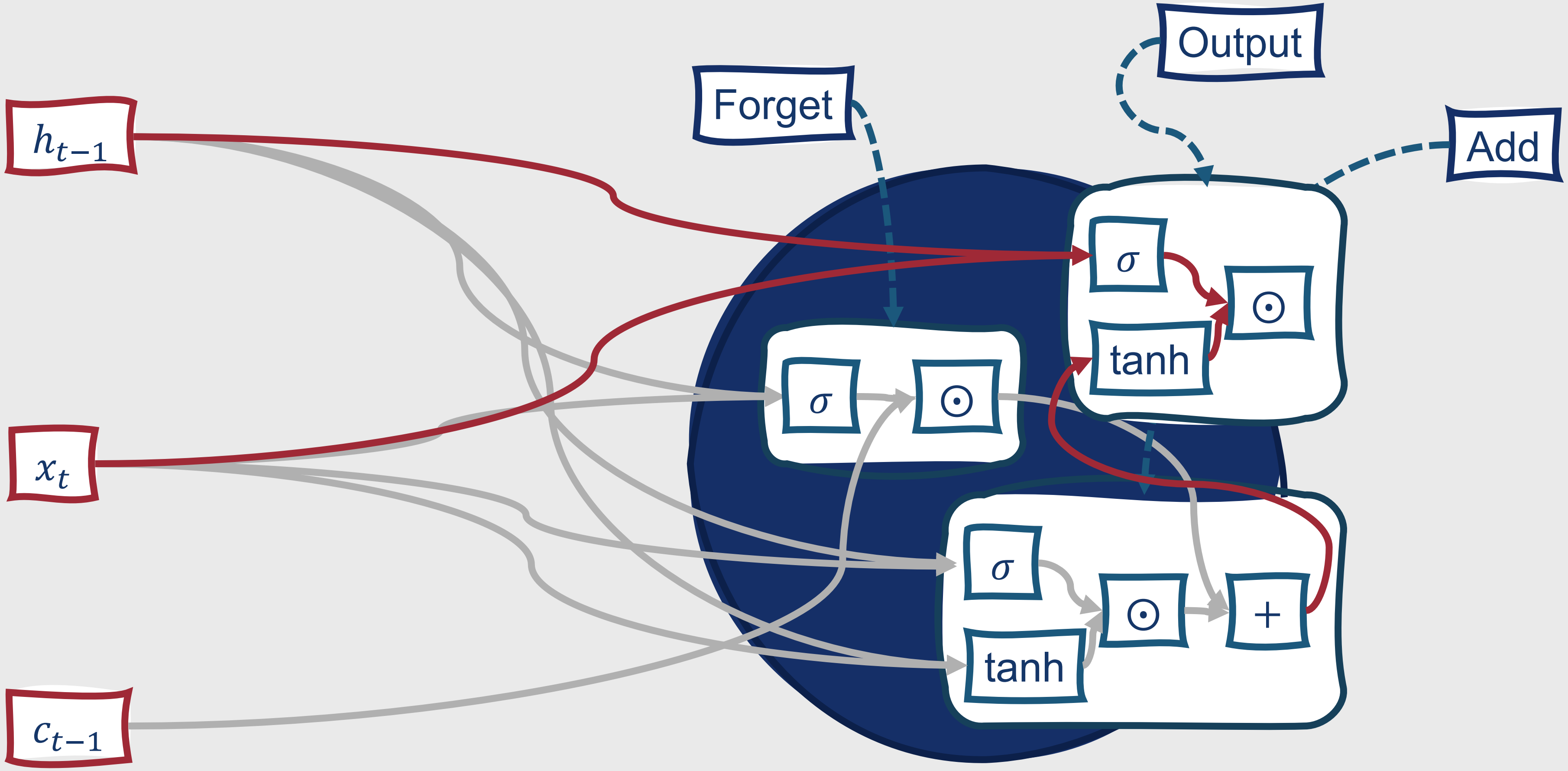
What does this process look like in a single LSTM unit?



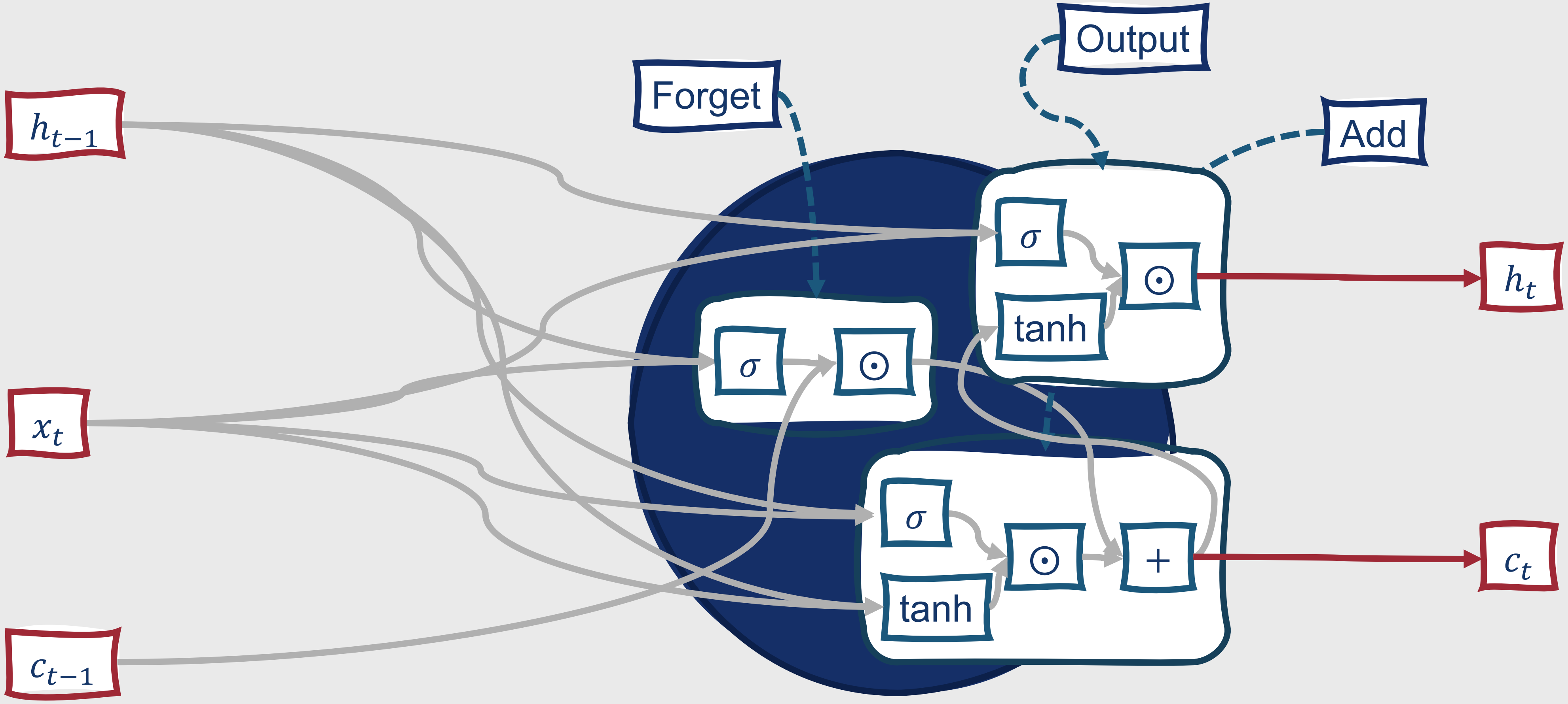
What does this process look like in a single LSTM unit?



What does this process look like in a single LSTM unit?



What does this process look like in a single LSTM unit?



Long Short-Term Memory Networks (LSTMs)

- LSTMs thus accept as input:
  - **Context layer**
  - **Hidden outputs** from previous timestep
  - **Current input vector**
- They return as output:
  - **Context layer**
  - **Hidden outputs** from the current timestep
- The output of the hidden layer can be used as input to subsequent layers in a stacked RNN, or to the network's output layer

Gated Recurrent Units (GRUs)

- Also manage the context that is passed through to the next timestep, but do so by utilizing a simpler architecture than LSTMs
  - No separate context vector
  - Only two gates
    - **Reset** gate
    - **Update** gate
- Gates still use a similar design to that seen in LSTMs
  - **Feedforward layer + sigmoid activation + pointwise multiplication** with the layer being gated, resulting in a **binary-like mask**

Reset Gate

- Goal: Decide which aspects of the previous hidden state are relevant to the current context
  - $r_t = \sigma(U_r h_{t-1} + W_r x_t)$
  - $\tilde{h}_t = \tanh(U(r_t \odot h_{t-1}) + W x_t)$

Weighted sum of:
- Hidden layer at the previous timestep
- Current input

Reset Gate

- Goal: Decide which aspects of the previous hidden state are relevant to the current context
  - $r_t = \sigma(U_r h_{t-1} + W_r x_t)$
  - $\underbrace{\widetilde{h_t} = \tanh(U(r_t \odot h_{t-1}) + W x_t)}$

$\boxed{\text{Intermediate representation for } h_t}$

Update Gate

- Goal: Decide which aspects of the intermediate hidden state and which aspects of the previous hidden state need to be preserved for future use
  - $z_t = \sigma(U_z h_{t-1} + W_z x_t)$
  - $h_t = (1 - z_t)h_{t-1} + z_t \tilde{h}_t$

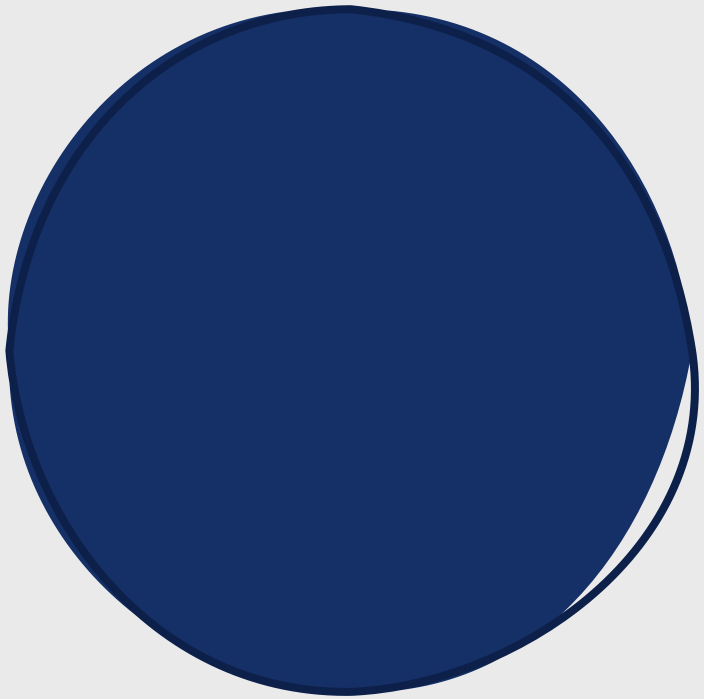
Weighted sum of:
- Hidden layer at the previous timestep
- Current input

Update Gate

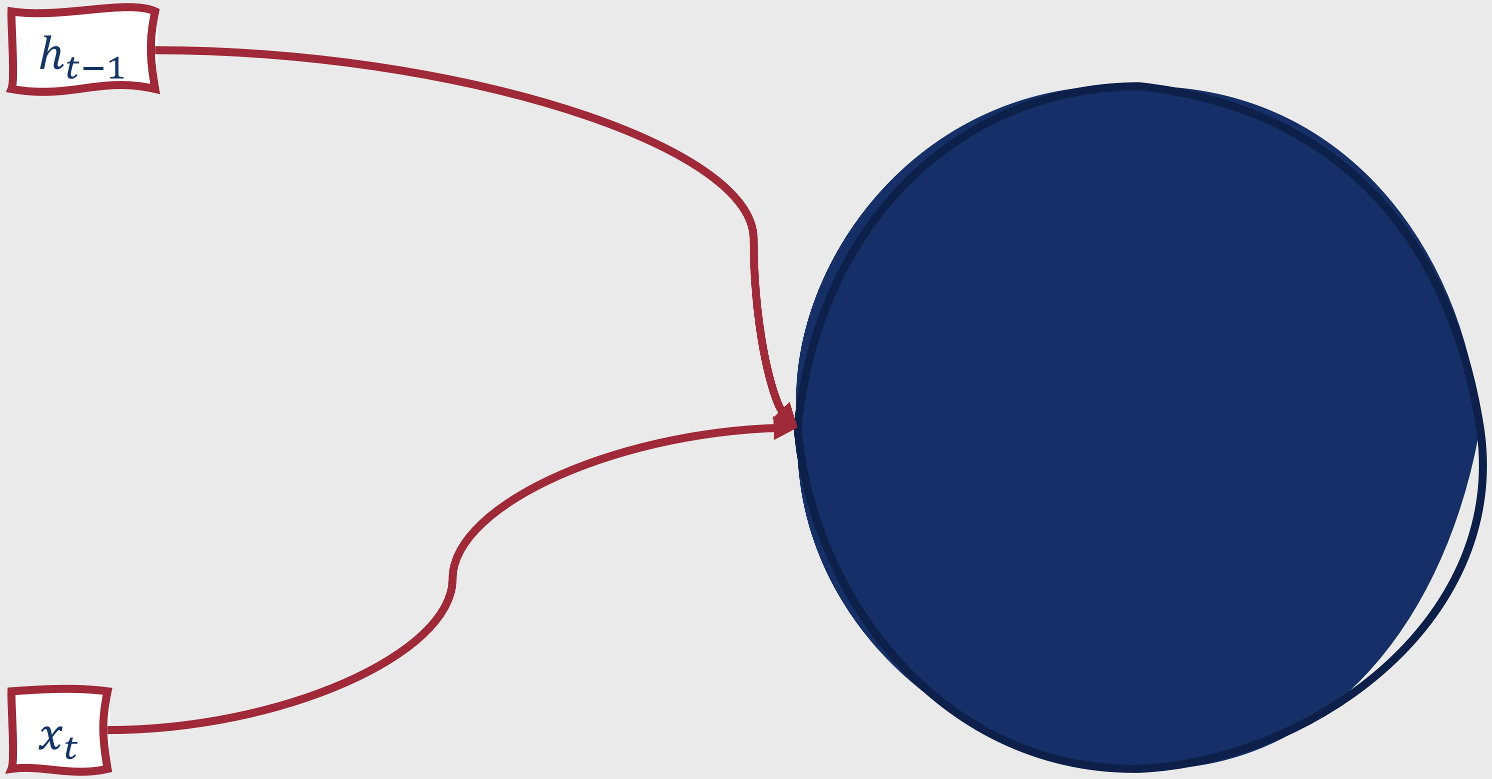
- Goal: Decide which aspects of the intermediate hidden state and which aspects of the previous hidden state need to be preserved for future use
  - $z_t = \sigma(U_z h_{t-1} + W_z x_t)$
  - $h_t = \underbrace{(1 - z_t)h_{t-1} + z_t \tilde{h}_t}$

Updated hidden layer output

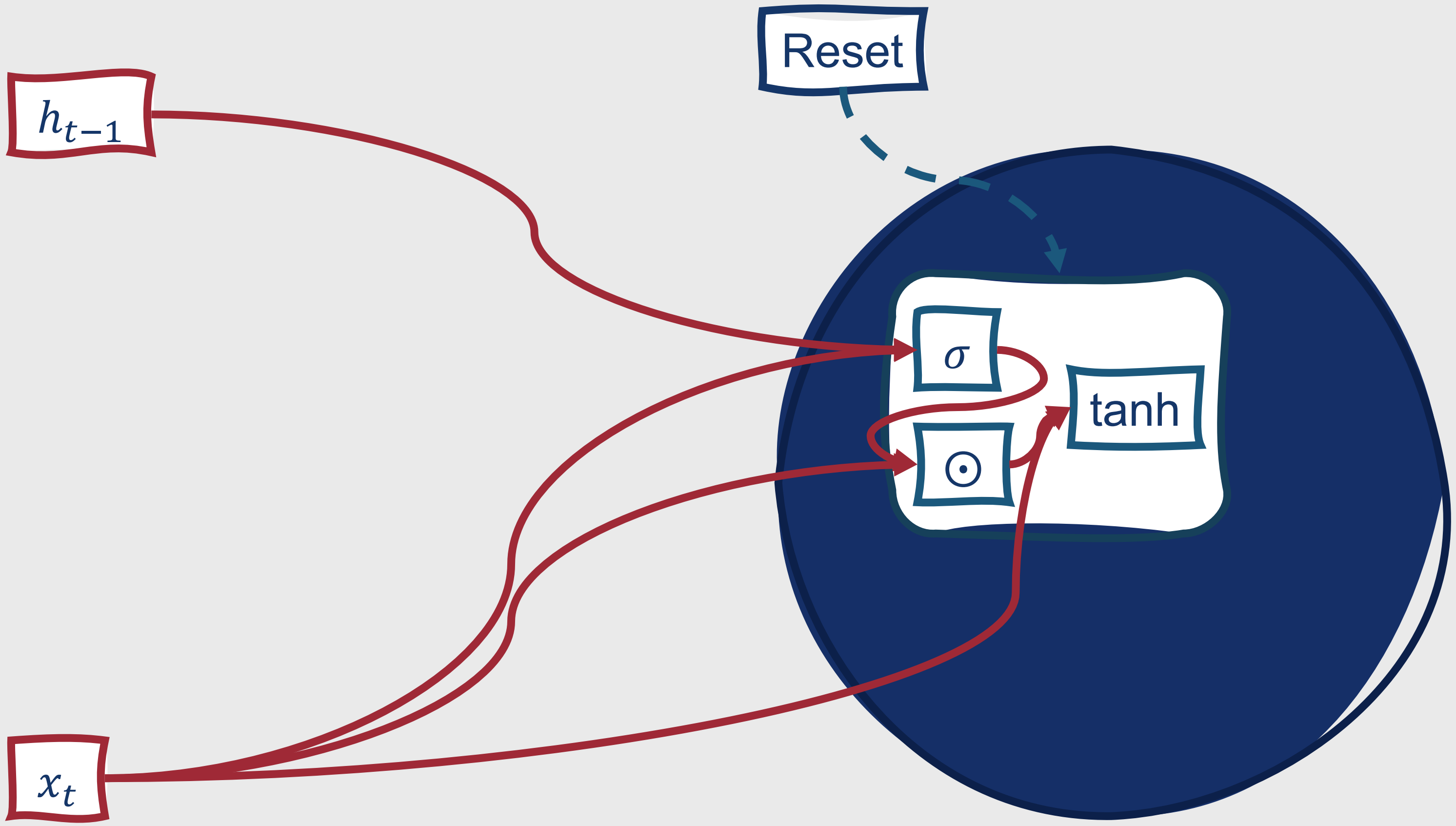
What does this process look like in a single GRU unit?



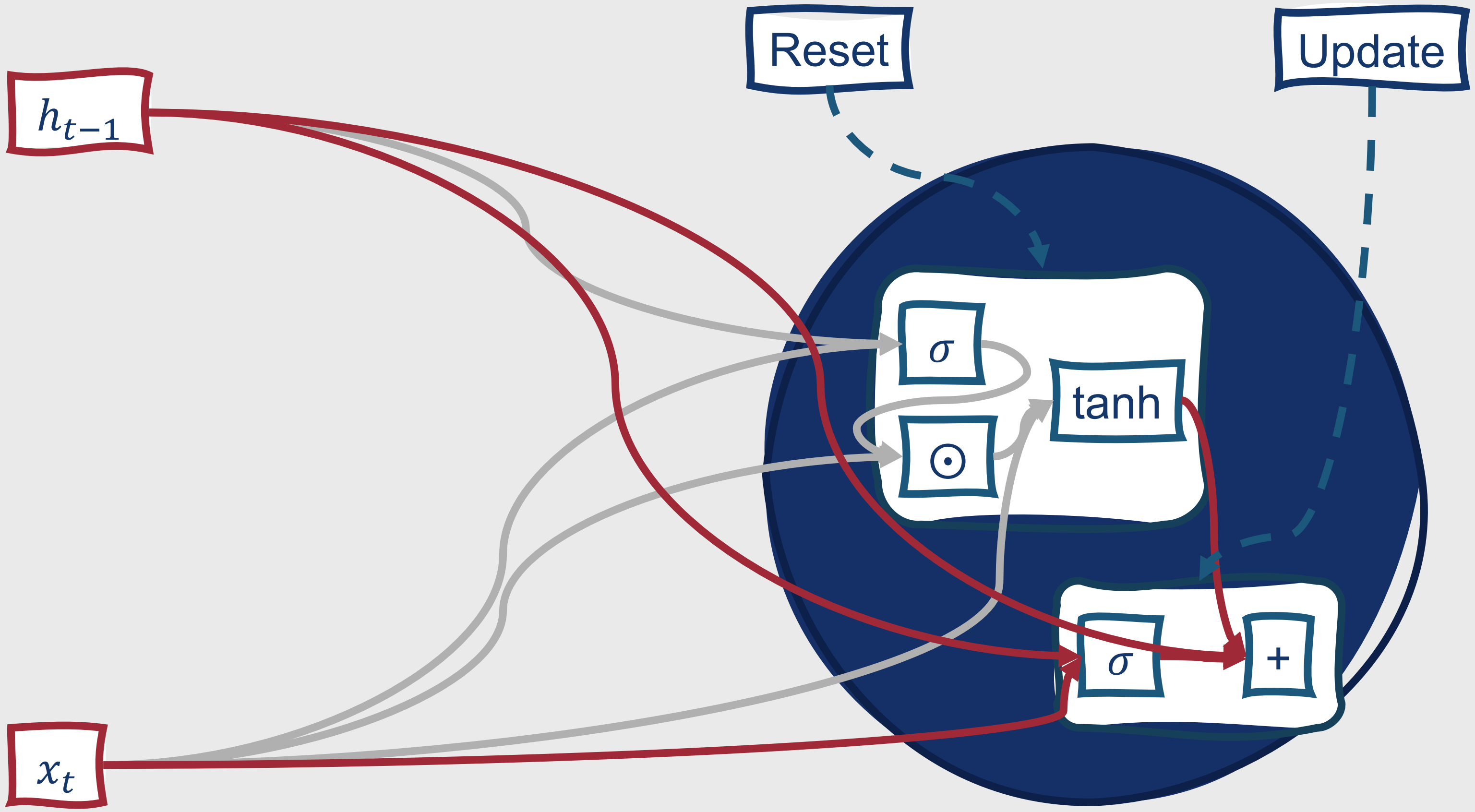
What does this process look like in a single GRU unit?



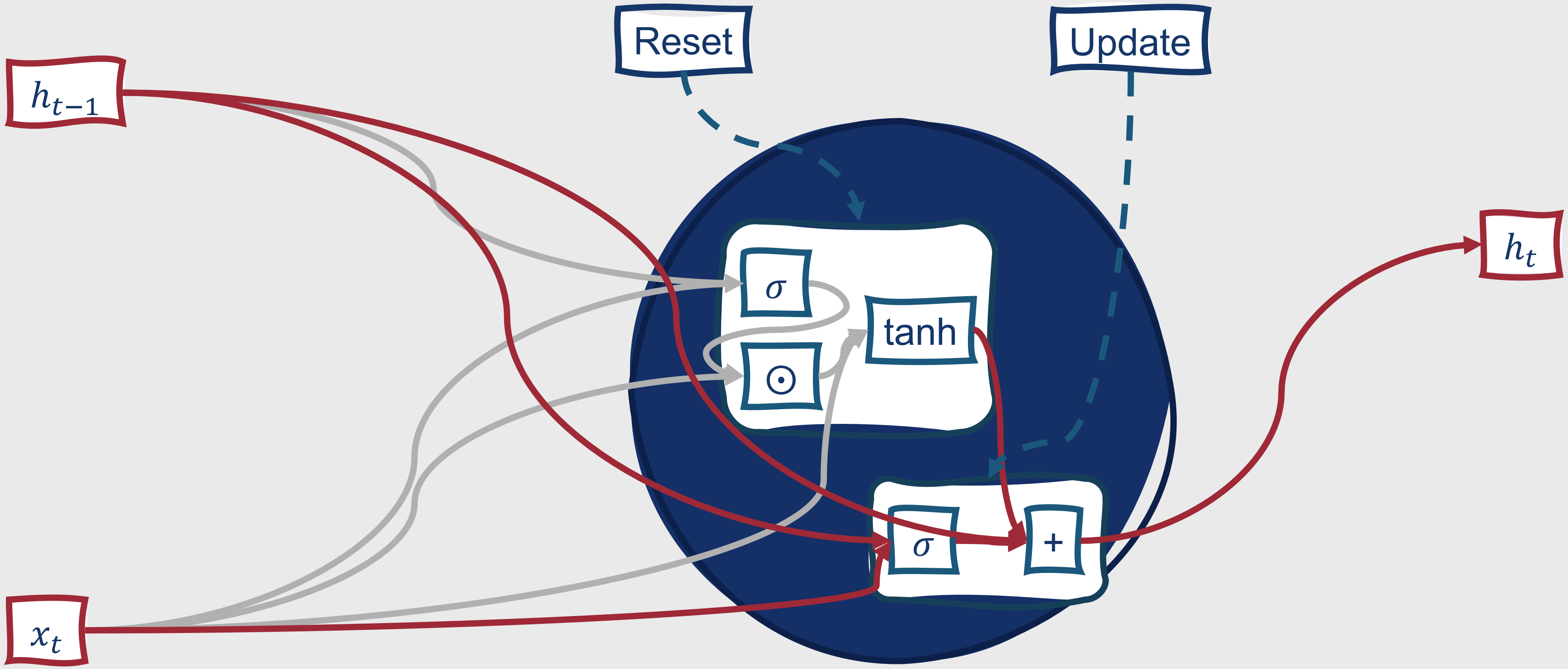
What does this process look like in a single GRU unit?



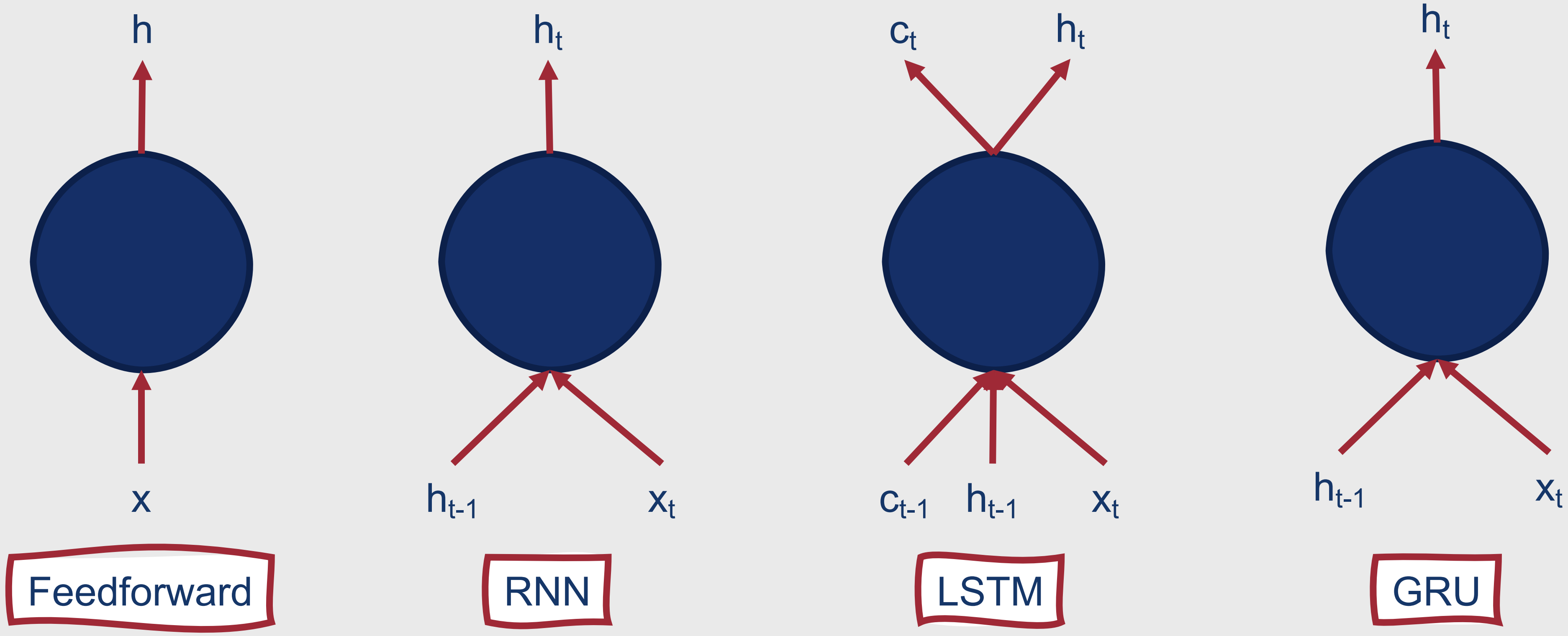
What does this process look like in a single GRU unit?



What does this process look like in a single GRU unit?



Overall, comparing inputs and outputs for some different types of neural units….



When to use LSTMs vs. GRUs?

Why use GRUs instead of LSTMs?

- **Computational efficiency:** Good for scenarios in which you need to train your model quickly and don't have access to high-performance computing resources

Why use LSTMs instead of GRUs?

- **Performance:** LSTMs generally outperform GRUs at the same tasks

So far, we've looked at a variety of sequential networks.

- Recurrent neural networks
- LSTMs
- GRUs
- Stacked RNNs (LSTMs, GRUs)
- Bidirectional RNNs (LSTMs, GRUs)

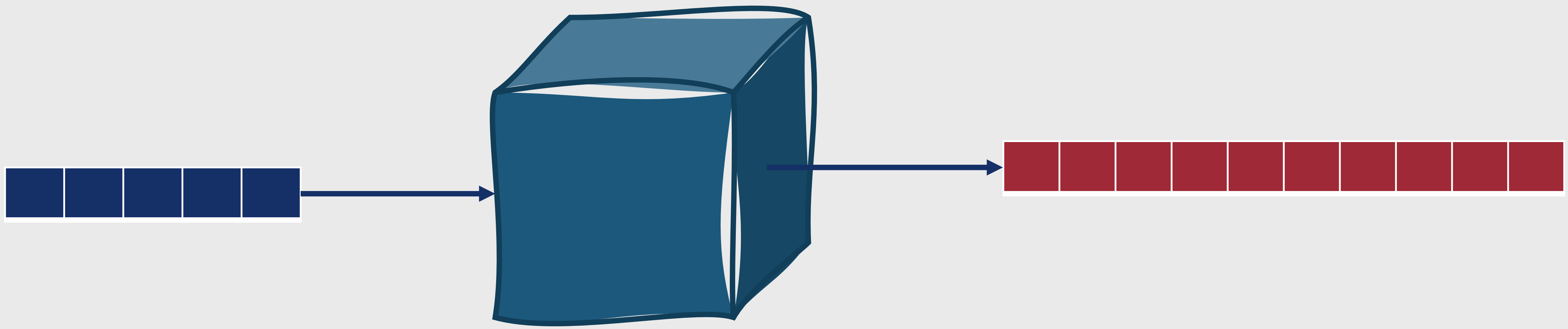
So far, we've looked at a variety of sequential networks.

- Recurrent neural networks
- LSTMs
- GRUs
- Stacked RNNs (LSTMs, GRUs)
- Bidirectional RNNs (LSTMs, GRUs)

All transform input sequences to output sequences in a one-to-one fashion

What if we don't need (or want) a one-to-one correspondence between input and output?

- Encoder-decoder networks
- Also called sequence-to-sequence (seq2seq) models



- Generate **contextually-appropriate**, **arbitrary-length** output sequences
- Particularly useful for:
  - Machine translation
  - Summarization
  - Question answering
  - Dialogue modeling

Encoder-Decoder Models

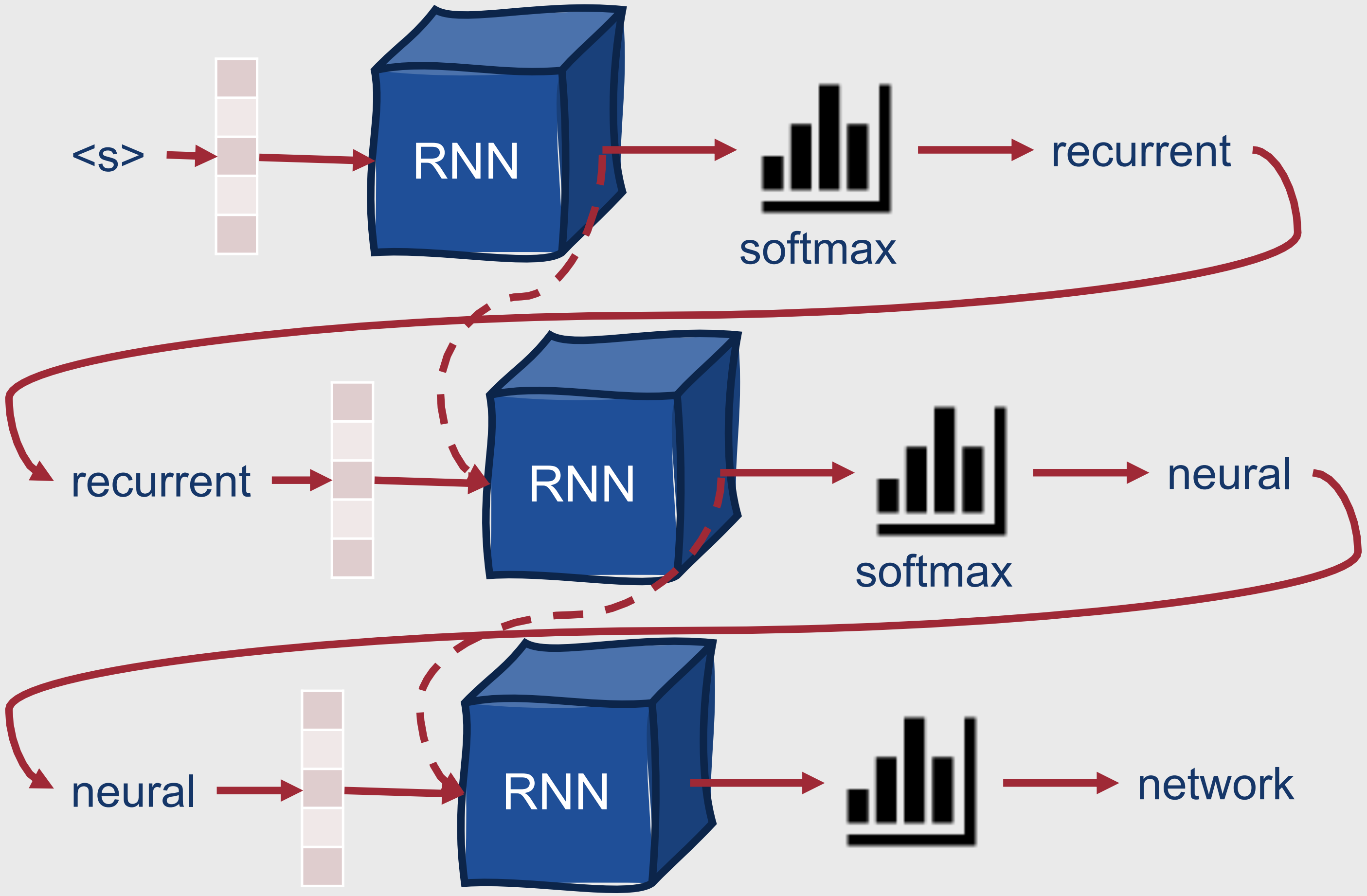
Encoder-Decoder Models

- Basic premise:
  - Use a neural network to **encode an input to an internal representation**
  - Pass that internal representation as input to a second neural network
  - Use that neural network to **decode the internal representation to a task-specific output sequence**
- Usually, the encoder and decoder are both some type of **RNN**
- This method allows networks to be trained in an **end-to-end** fashion

Where did this idea come from?

Recall our discussion of autoregressive generation:

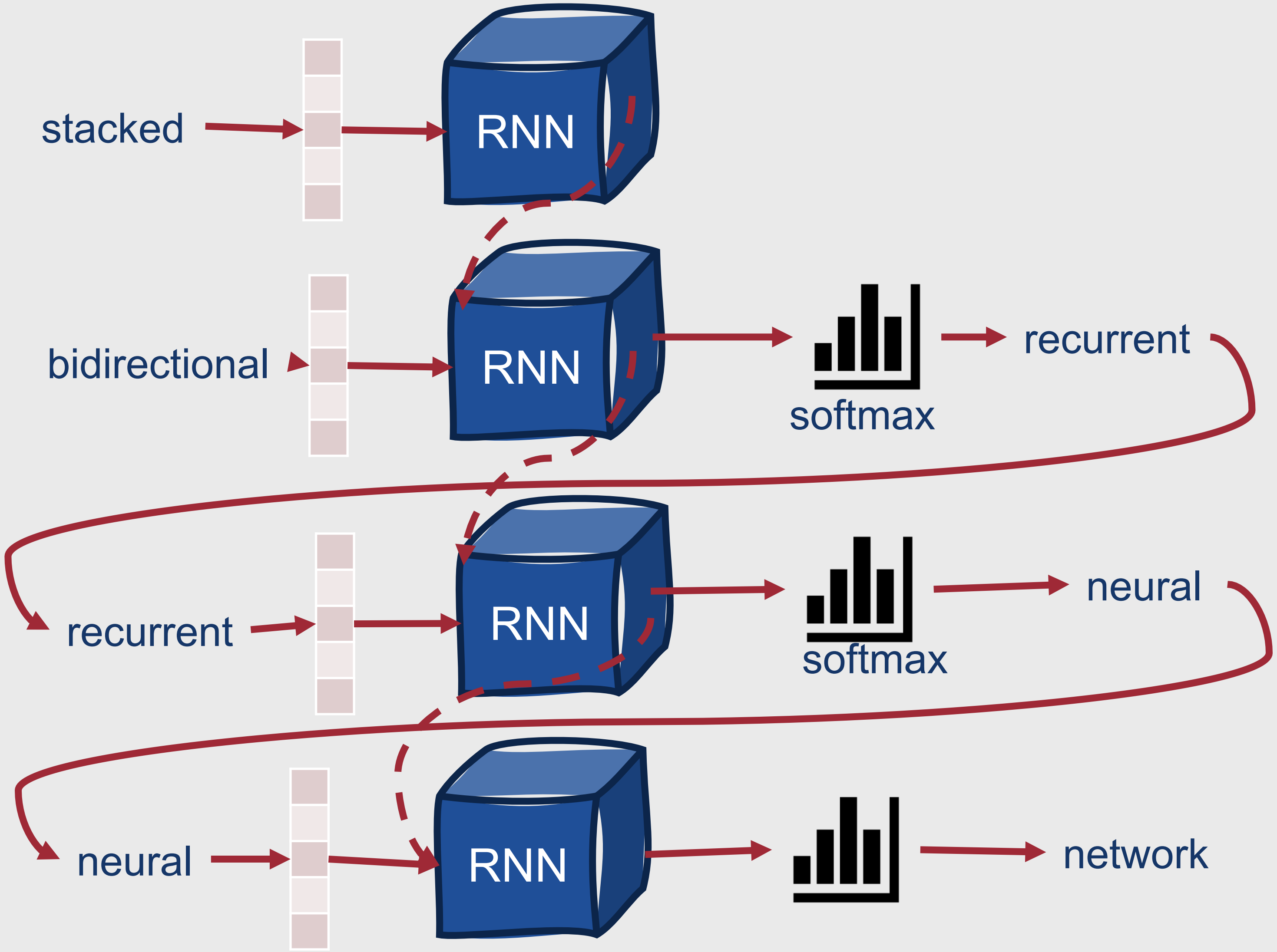
- Start with a seed token (e.g., <s>)
- Predict the most likely next word in the sequence
- Use that word as input at the next timestep
- Repeat until an end token (or max length) is reached



Slight variation to this idea….

- Rather than generating a sentence from scratch, the model can generate a sentence given a prefix
  - Pass the specified prefix through the language model, in sequence
  - End with the hidden state corresponding to the last word of the prefix
  - Start the autoregressive process at that point
- **Goal: Output sequence should be a reasonable completion of the prefix**

Updated Autoregressive Generation



We can build upon this idea to transform one type of sequence to another.

- Machine translation example:
  1. Take a **sequence of text from Language #1**
  2. Take the **translation of that text from Language #2**
  3. **Concatenate the two sequences**, separated by a marker
  4. Use these concatenated sequences to **train the autoregressive model**
  5. Test the model by **passing in only the first part of a concatenated sequence** (text from Language #1) and checking to see what the generated completion (text from Language #2) looks like

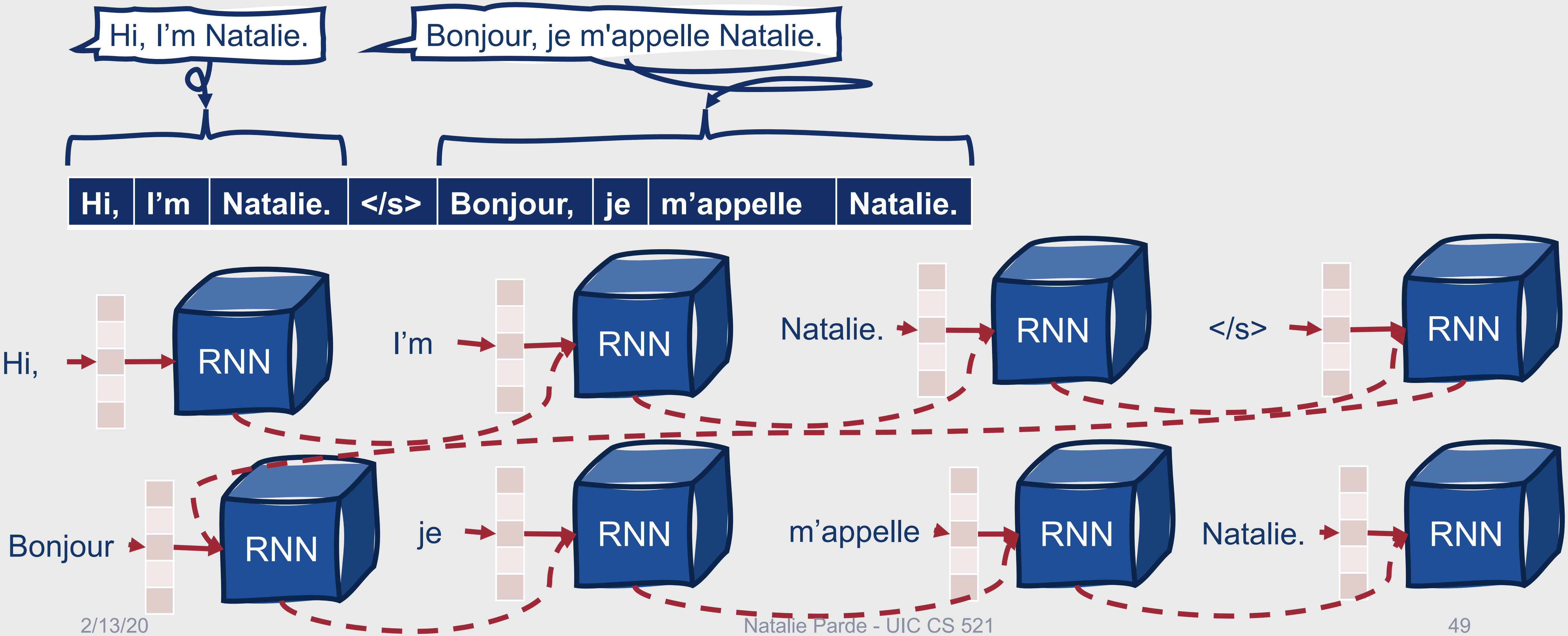
Intuition: Machine Translation

Hi, I’m Natalie.

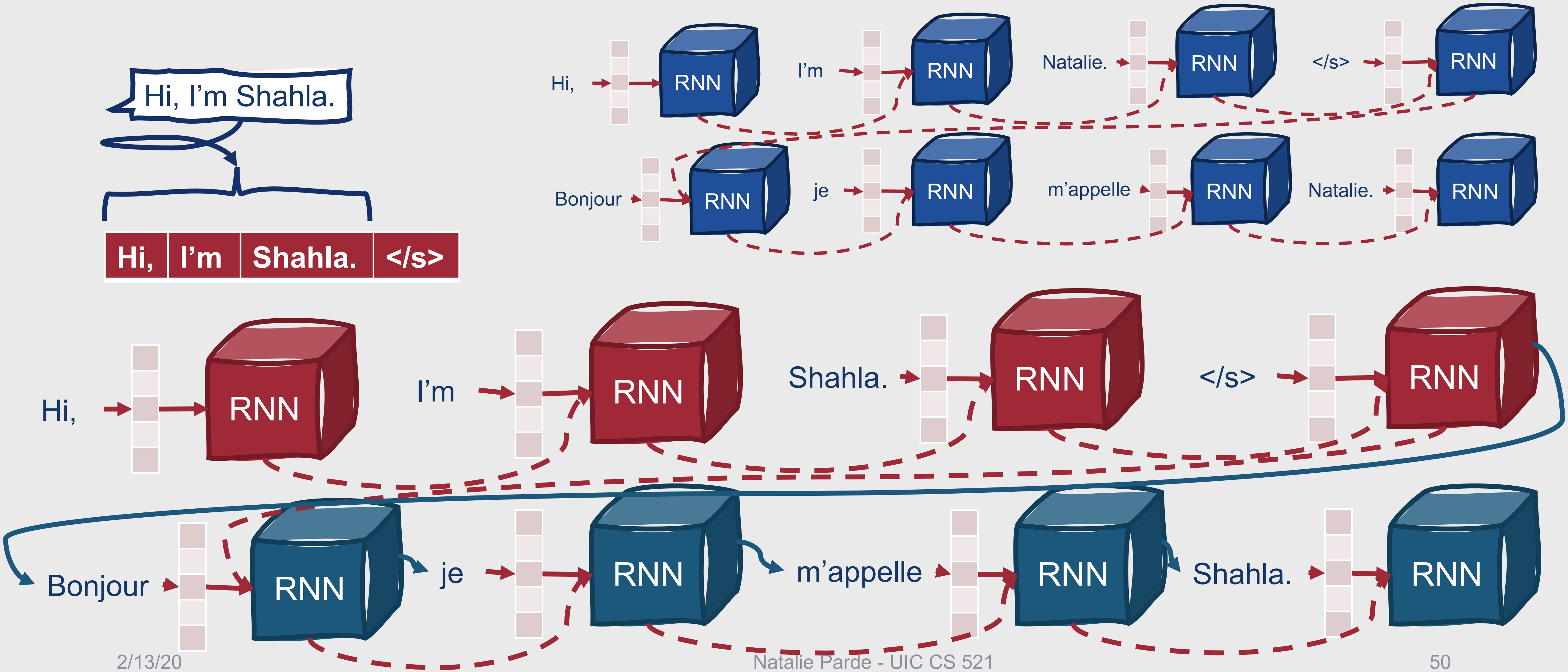
Bonjour, je m'appelle Natalie.

Intuition: Machine Translation

Intuition: Machine Translation



Intuition: Machine Translation



This intuition forms the basis of encoder-decoder networks.

- Key elements of an encoder-decoder network:
  - **Encoder:** Generates a contextualized representation of the input
  - **Decoder:** Takes the contextualized representation and autoregressively generates a sequence of outputs

More formally….

- **Encoder**
  - Accepts an input sequence, x_1^n
  - Generates a sequence of contextualized representations, h_1^n
- **Context vector**
  - A function, c, of h_1^n that conveys the basic meaning of x_1^n to the decoder
- **Decoder**
  - Accepts c as input
  - Generates an arbitrary-length sequence of hidden states, h_1^m, from which a corresponding sequence of output states y_1^m can be obtained

Encoders

- Can be any type of neural network
  - Feedforward network
  - CNN
  - RNN
  - LSTM
  - GRU

More common

- These networks can be stacked on top of one another
  - Very common: Stacked Bi-LSTMs

Decoders

- Need to perform autoregressive generation to produce the output sequence
- Can be any type of recurrent network
  - RNN
  - LSTM
  - GRU

Decoders

- Formally….
  - $c = h_n^e$
  - $h_0^d = c$

  - $h_t^d = g(\widehat{y_{t-1}}, h_{t-1}^d)$
  - $z_t = f(h_t^d)$
  - $y_t = \text{softmax}(z_t)$

Decoders

- Formally….
  - $c = h_n^e$
  - $h_0^d = c$
  - $h_t^d = g(\widehat{y_{t-1}}, h_{t-1}^d)$
  - $z_t = f(h_t^d)$
  - $y_t = \text{softmax}(z_t)$

Final hidden state of the encoder

Decoders

- Formally….
  - $c = h_n^e$
  - $h_0^d = c$

First hidden state of the decoder

  - $h_t^d = g(\widehat{y_{t-1}}, h_{t-1}^d)$
  - $z_t = f(h_t^d)$
  - $y_t = \text{softmax}(z_t)$

Decoders

- Formally….
  - $c = h_n^e$
  - $h_0^d = c$
  - $h_t^d = \underbrace{g(\overbrace{\widehat{y_{t-1}}}^{\text{Embedding for the output sampled from the previous step}}, h_{t-1}^d)}_{\text{Some type of RNN}}$
  - $z_t = f(h_t^d)$
  - $y_t = \text{softmax}(z_t)$

Embedding for the output sampled from the previous step

Some type of RNN

Decoders

- Formally….
  - $c = h_n^e$
  - $h_0^d = c$

  - $h_t^d = g(\widehat{y_{t-1}}, h_{t-1}^d)$
  - $z_t = f(h_t^d)$
  - $y_t = \text{softmax}(z_t)$

Regular ending steps (activation function applied to hidden state outputs, and softmax applied to activation outputs)

A couple useful extensions….

- Formally….
  - $c = h_n^e$
  - $h_0^d = c$

  - $h_t^d = g(\widehat{y_{t-1}}, h_{t-1}^d) \rightarrow \boxed{h_t^d = g(\widehat{y_{t-1}}, h_{t-1}^d, c)}$
  - $z_t = f(h_t^d)$
  - $y_t = \text{softmax}(z_t)$

Make the context vector available at each timestep when decoding, so that its influence doesn't diminish over time

A couple useful extensions….

- Formally….
  - $c = h_n^e$
  - $h_0^d = c$

  - $h_t^d = g(\widehat{y_{t-1}}, h_{t-1}^d) \rightarrow h_t^d = g(\widehat{y_{t-1}}, h_{t-1}^d, c)$
  - $z_t = f(h_t^d)$
  - $y_t = \text{softmax}(z_t) \rightarrow y_t = \boxed{\text{softmax}(\widehat{y_{t-1}}, z_t, c)}$

Condition output on not only the hidden state, but the previous output and encoder context (easier to keep track of what's been generated already)

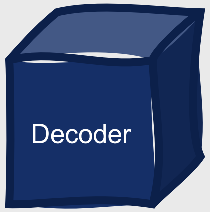
What other ways can we improve the decoder's output quality?

- **Beam search**
- Improved **context vector**
  - Final hidden state tends to be more focused on the end of the input sequence
  - Can be addressed by using bidirectional RNNs, summing the encoder hidden states, or averaging the encoder hidden states

Beam Search

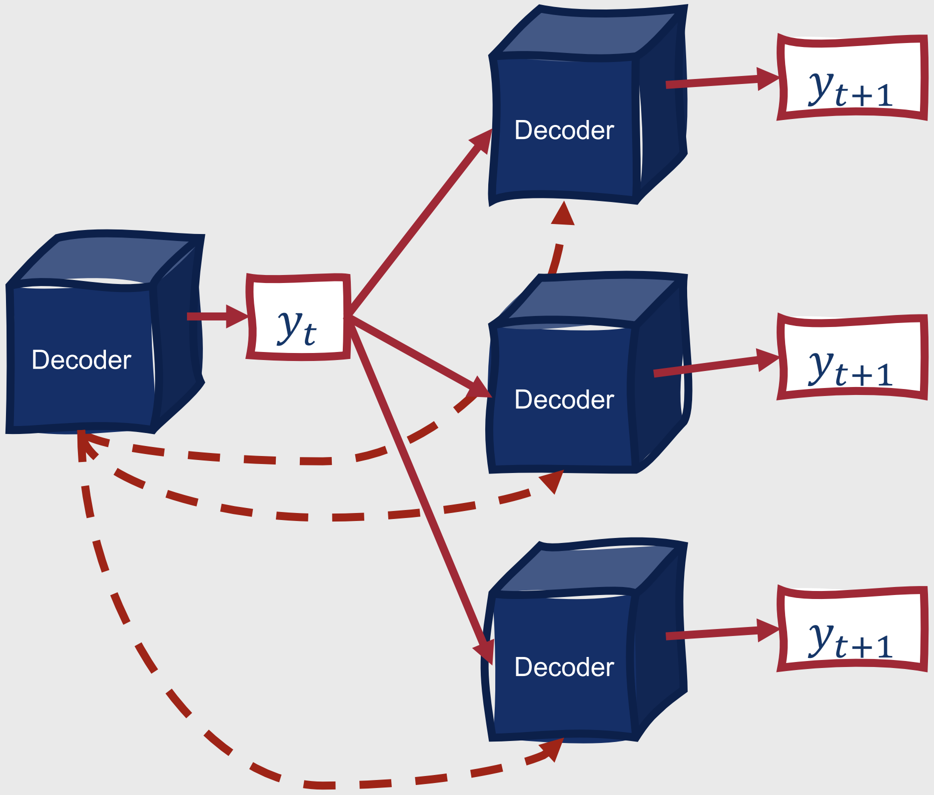
- Selects from multiple possible outputs by framing the task as a **state space search**
- Combines **breadth-first search** with a **heuristic filter**
  - Continually prunes search space to stay a fixed size (**beam width**)
- Results in a set of *b* **hypotheses**, where *b* is the beam width

How does beam search work?



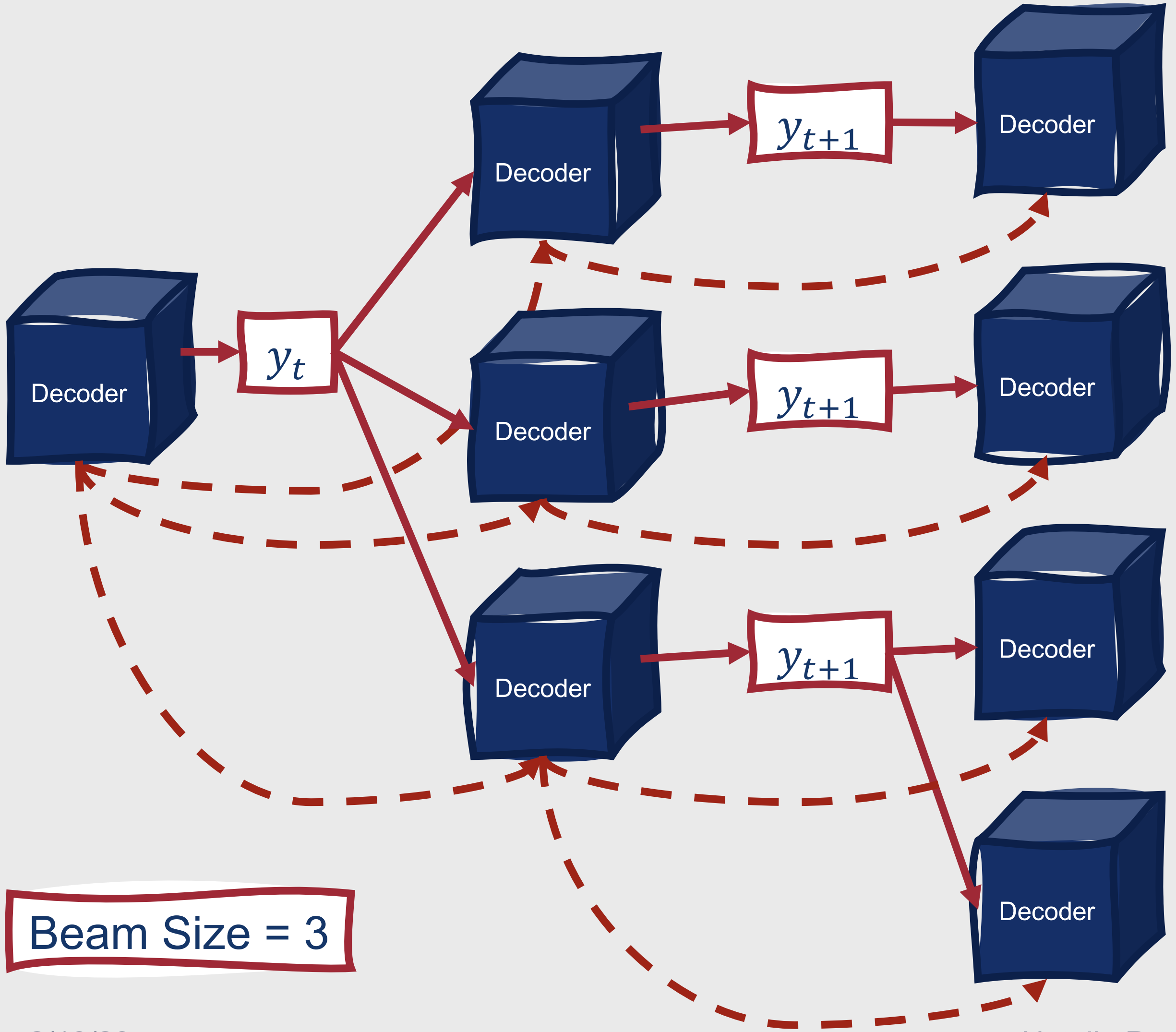
Beam Size = 3

How does beam search work?

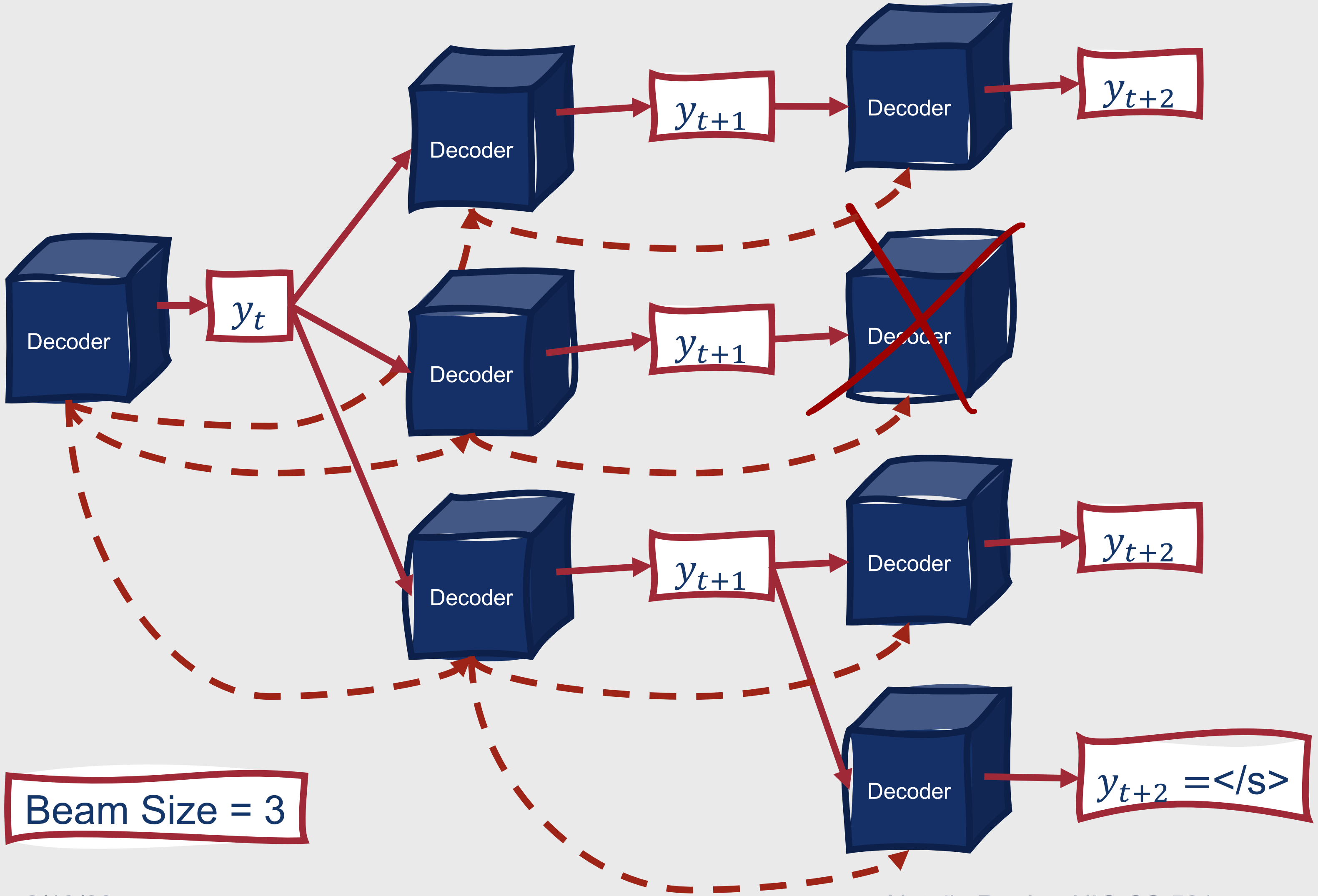


Beam Size = 3

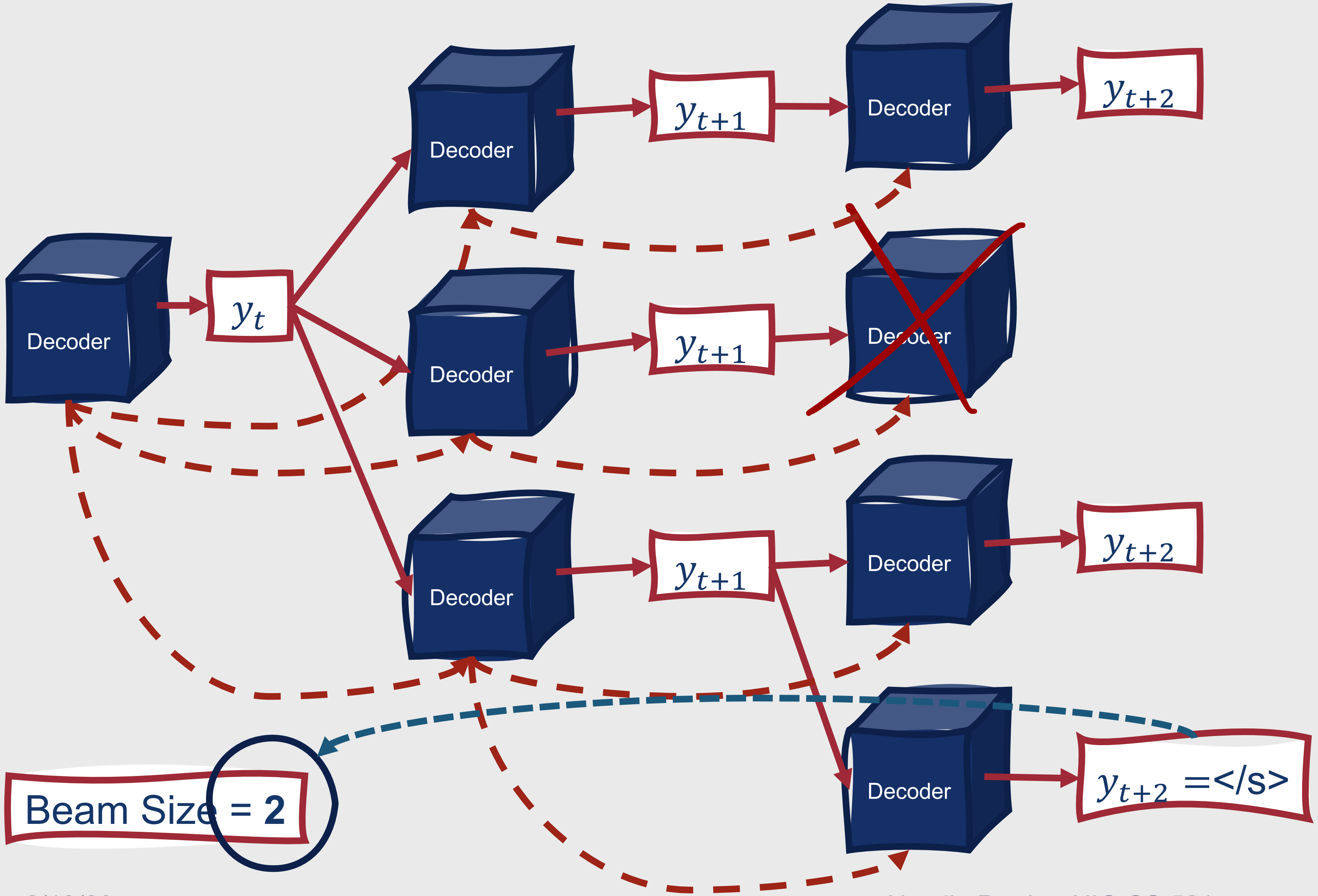
How does beam search work?



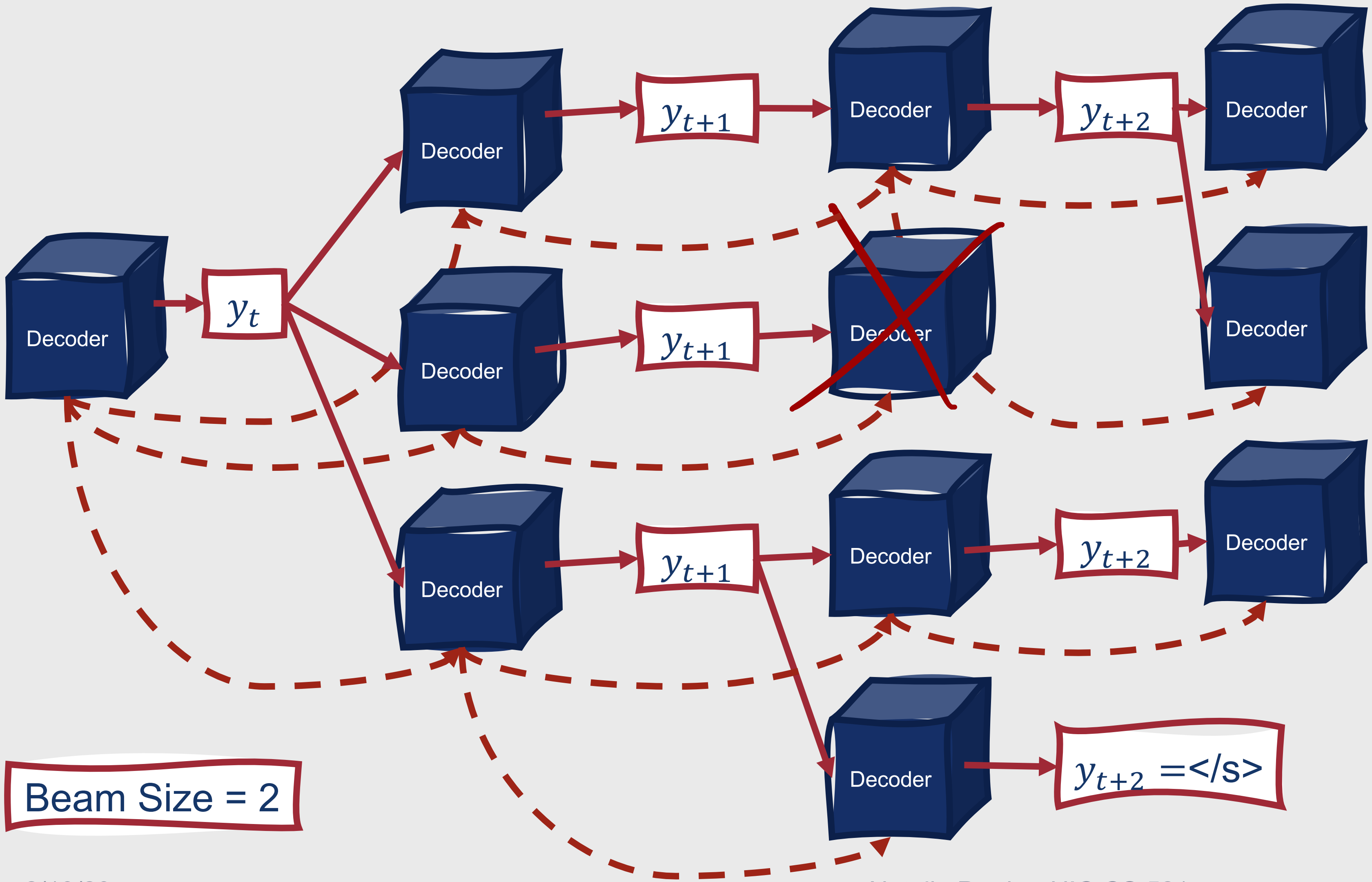
How does beam search work?



How does beam search work?

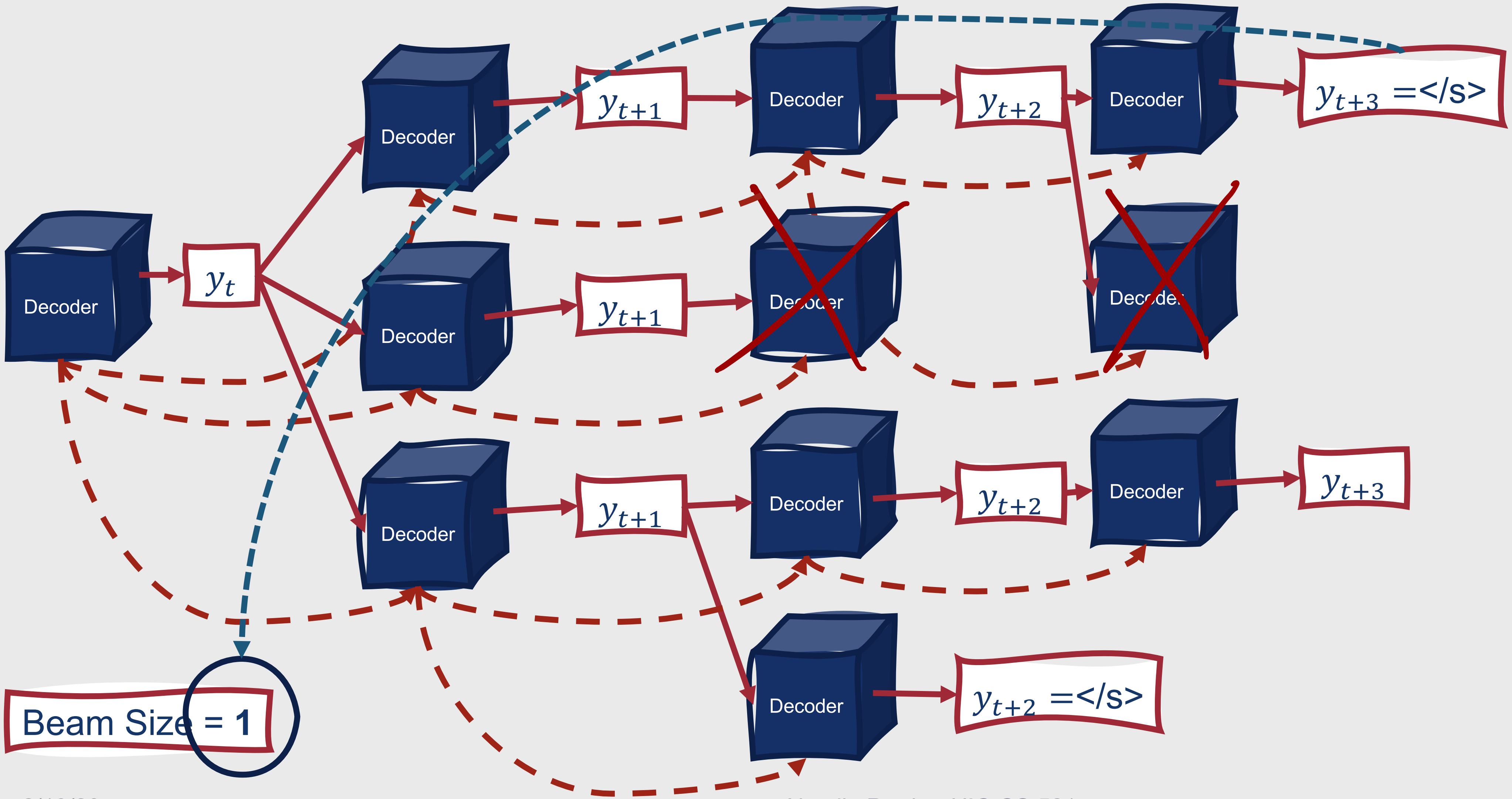


How does beam search work?

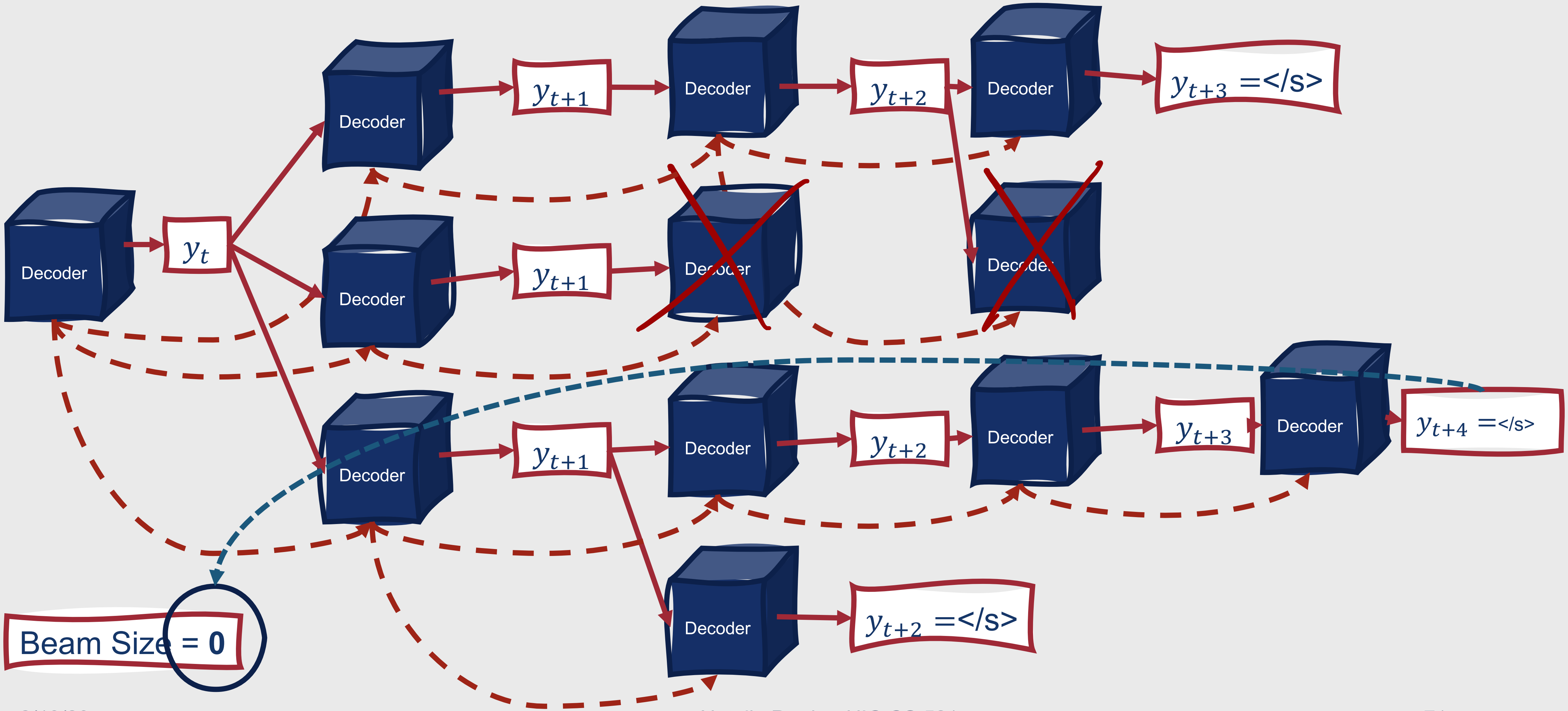


Beam Size = 2

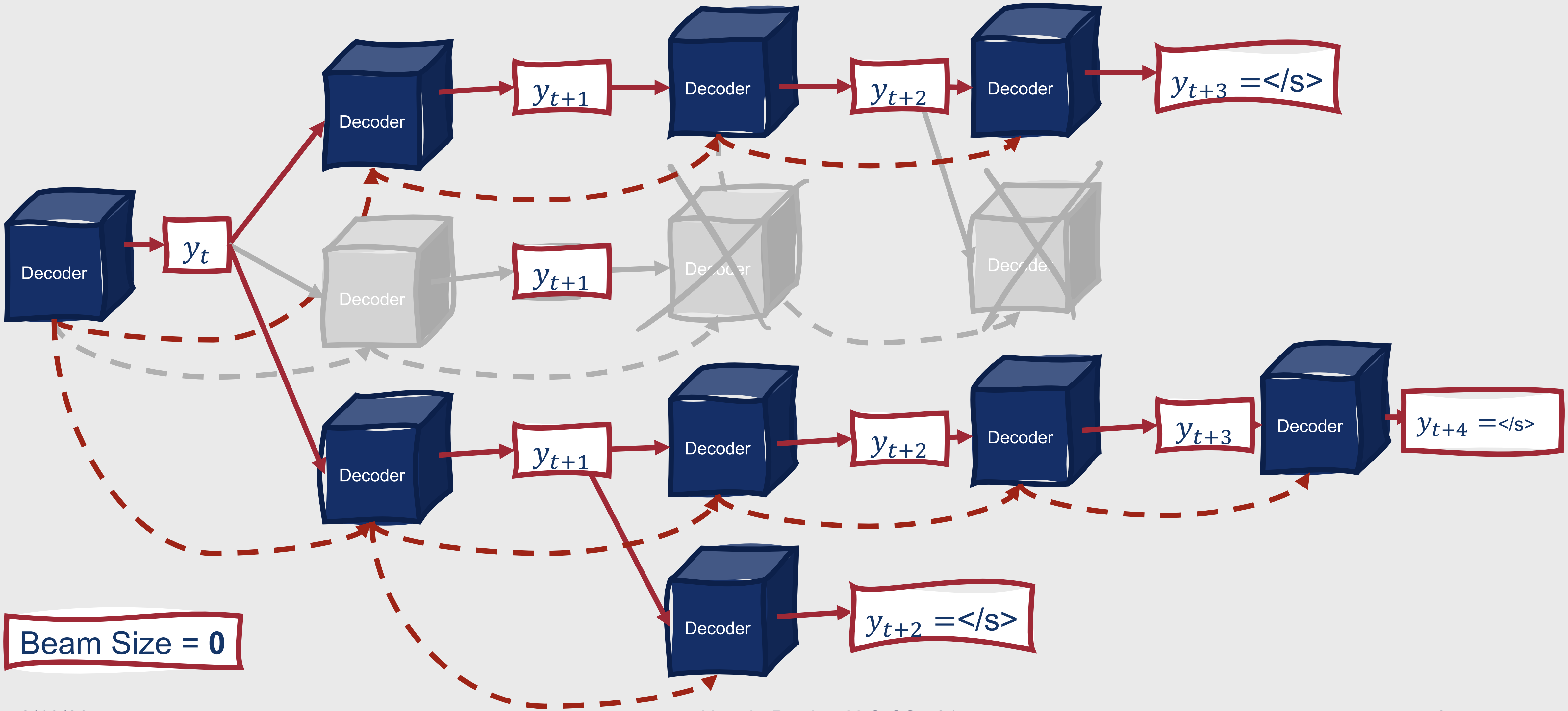
How does beam search work?



How does beam search work?



How does beam search work?



How do we choose a best hypothesis?

- Probabilistic scoring scheme
- Pass all or a subset of hypotheses to a downstream application

So far, the encoder context vectors we've seen have been simple and static.

- Can we do better?
  - Yes 🙂

Attention Mechanism

- Takes entire encoder context into account
- Dynamically updates during the course of decoding
- Can be embodied in a fixed-size vector

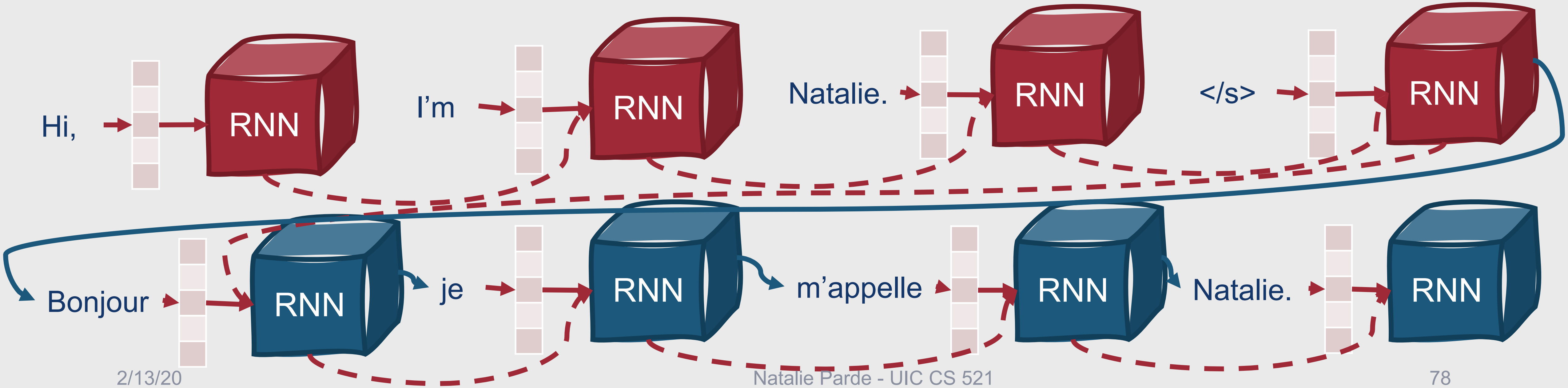
Recall....

- We've already made our context vector available at each timestep when decoding
  - $h_t^d = g(\widehat{y_{t-1}}, h_{t-1}^d, c)$
- The first step in creating our attention mechanism is to update our hidden state such that it is conditioned on an updated context vector with each decoding step
  - $h_t^d = g(\widehat{y_{t-1}}, h_{t-1}^d, c_t)$

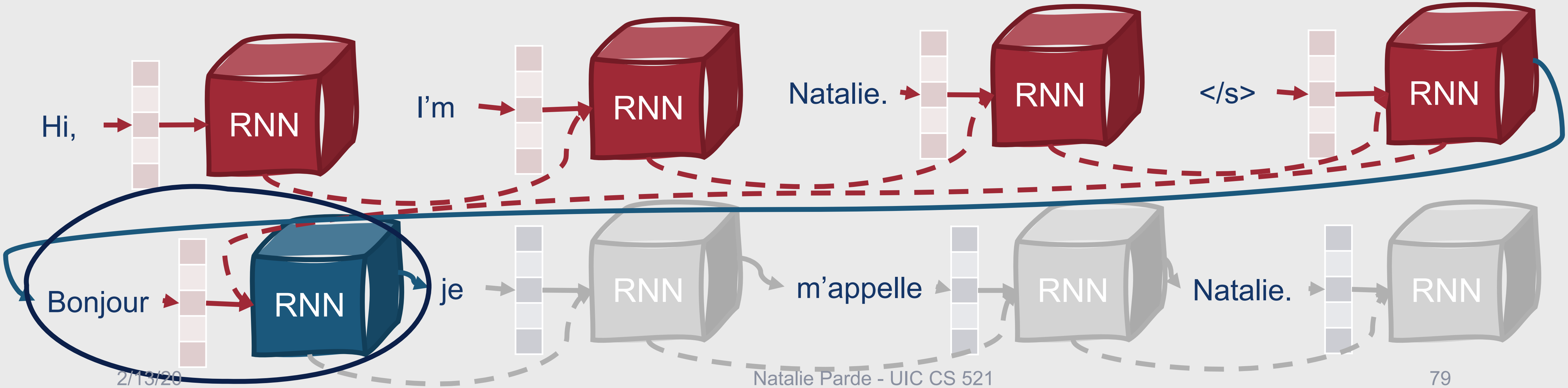
How do we dynamically create a new context vector at each step?

- Compute a vector of scores that capture the relevance of each encoder hidden state to the decoder hidden state, h_{t-1}^{d}
  - $score(h_{i-1}^{d}, h_{j}^{e}) = h_{i-1}^{d} \cdot h_{j}^{e}$

Vector of Context Scores

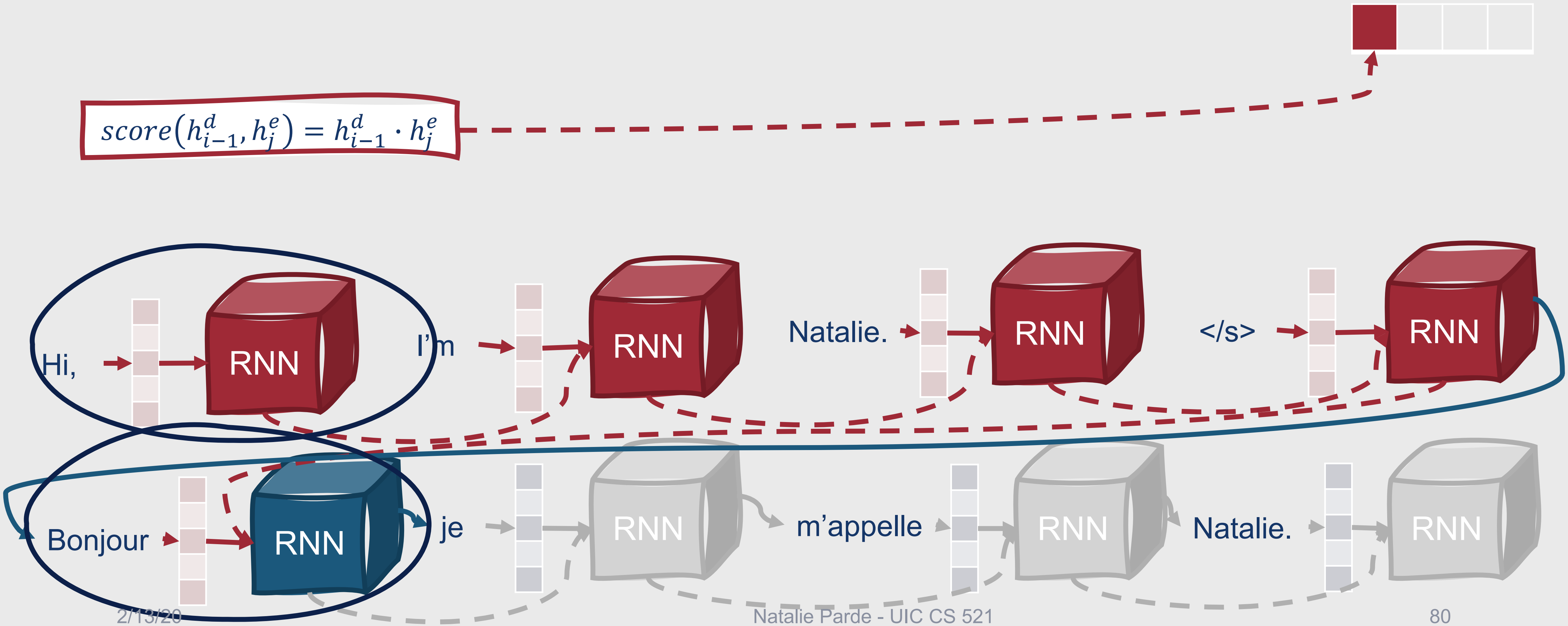


Vector of Context Scores

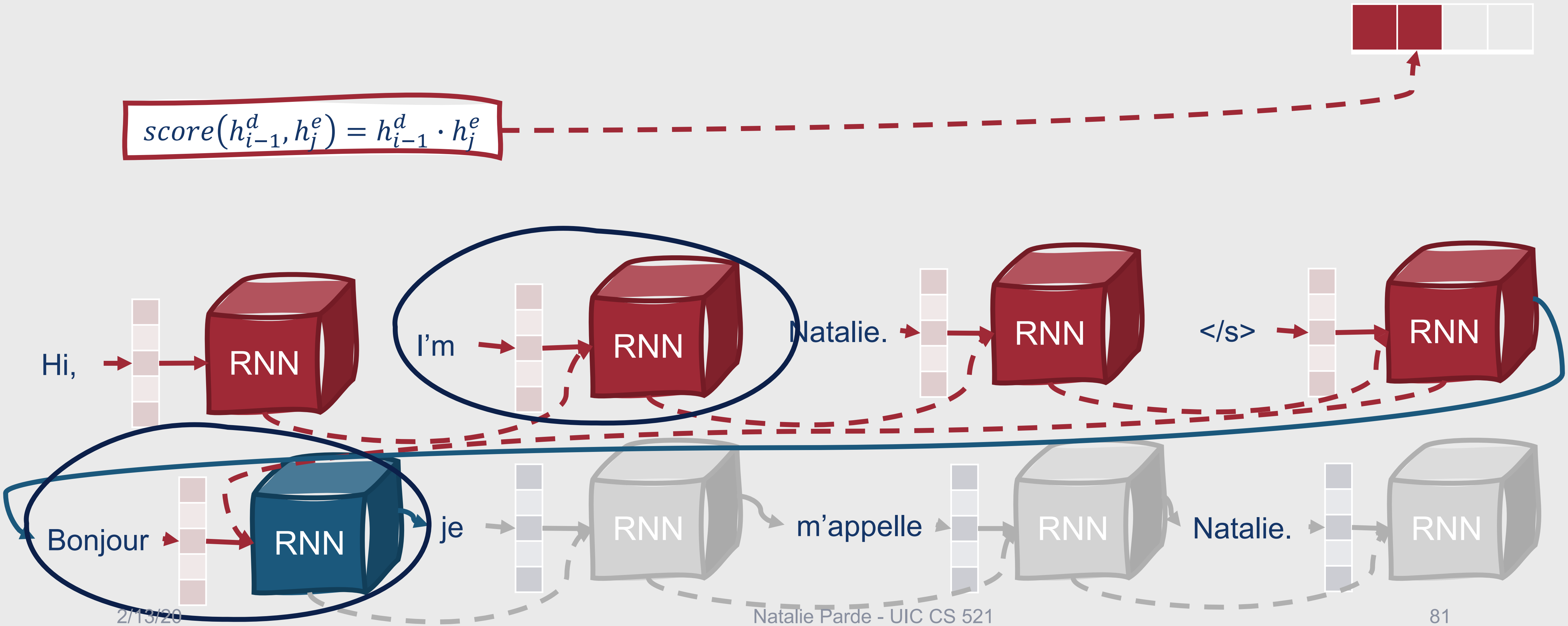


Vector of Context Scores

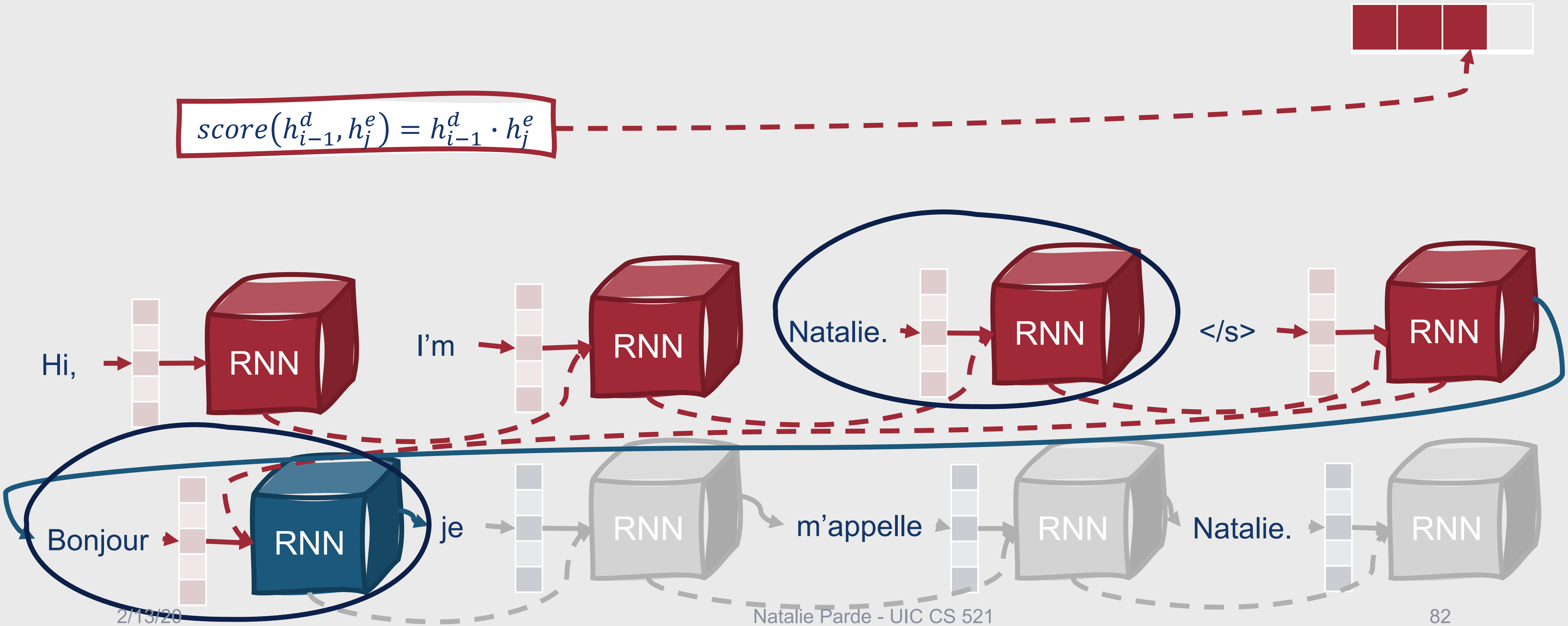
$$score(h_{i-1}^d, h_j^e) = h_{i-1}^d \cdot h_j^e$$



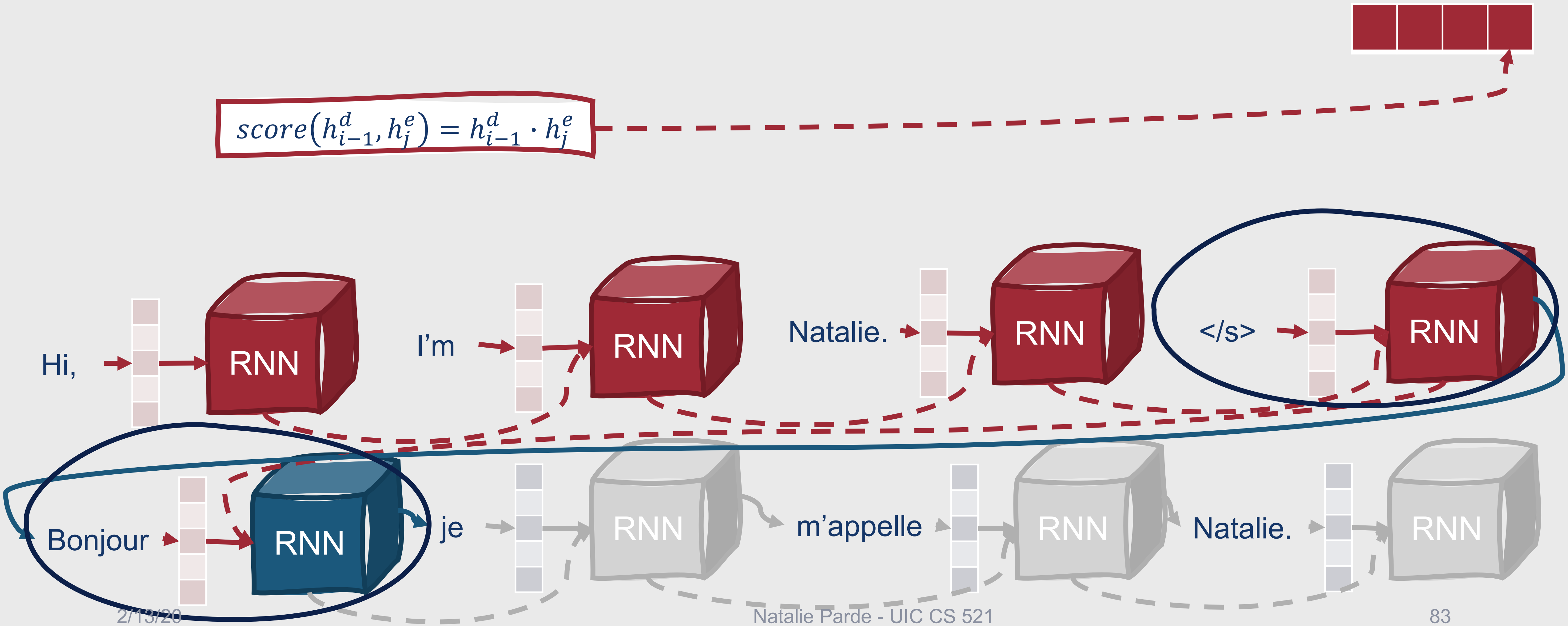
Vector of Context Scores



Vector of Context Scores



Vector of Context Scores



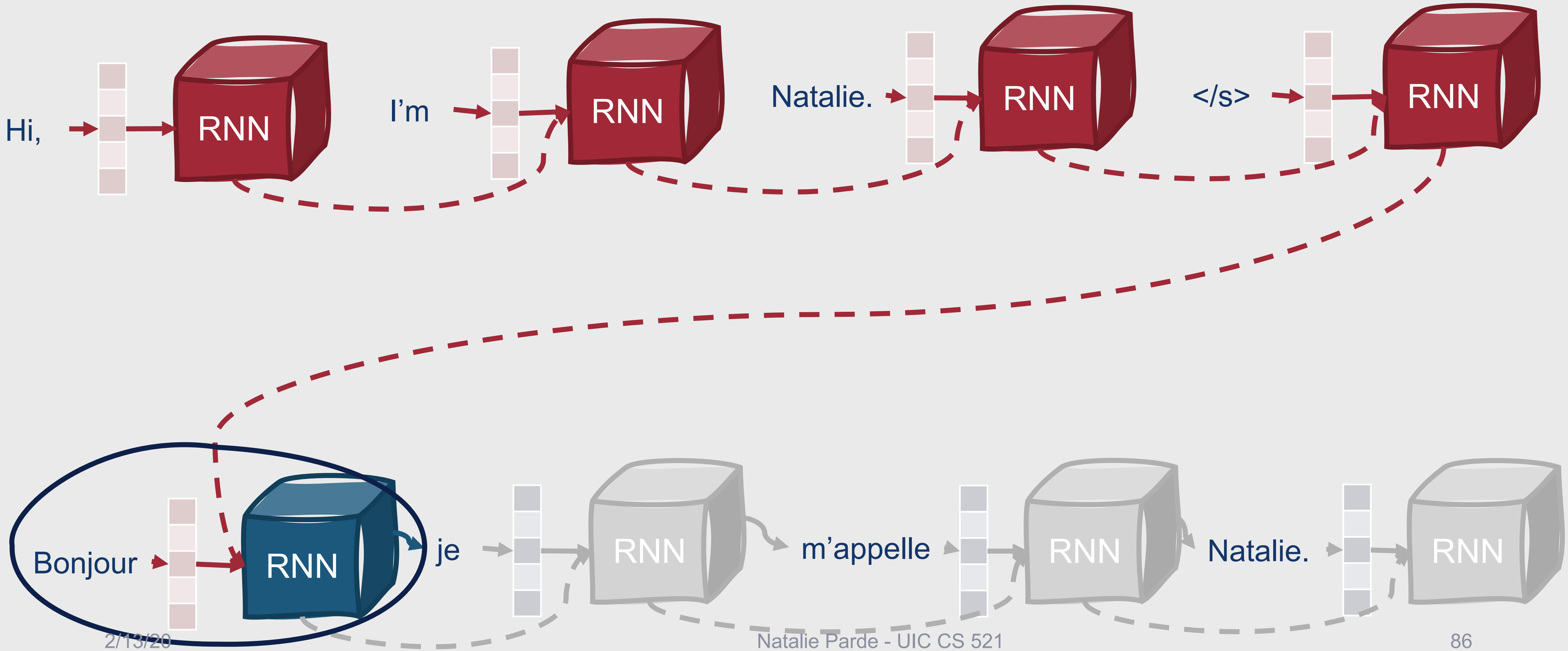
In practice, a simple dot product isn't the best similarity metric.

- Instead, parameterize the score with its own set of weights
  - $score\left(h_{i-1}^{d}, h_{j}^{e}\right) = h_{i-1}^{d} W_s h_{j}^{e}$
- This allows the model to learn which aspects of similarity between the encoder and decoder states are important

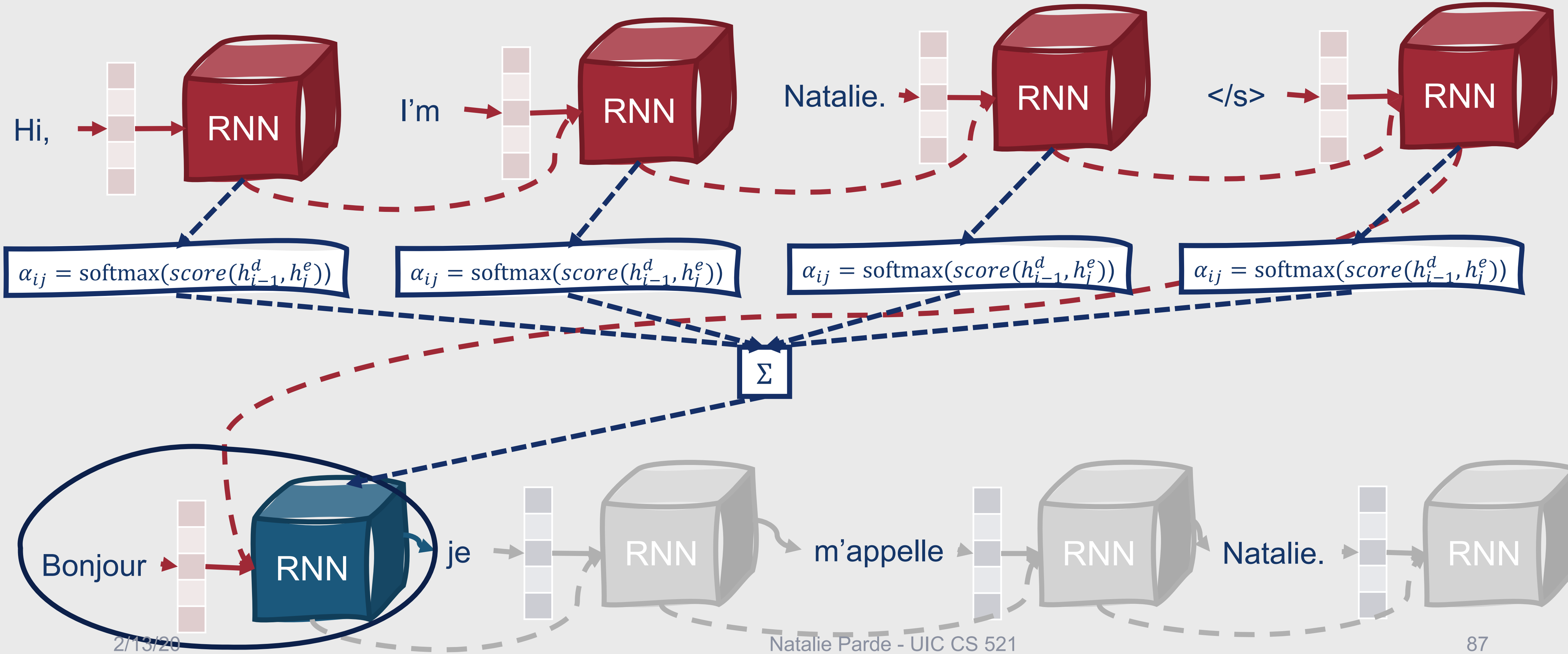
How do we make use of these context scores?

- **Normalize them** to create a vector of weights, α_{ij}
  - $\alpha_{ij} = \mathrm{softmax}(score(h_{i-1}^{d}, h_j^{e}) \forall j \in e)$
  - Provides the proportional relevance of each encoder hidden state j to the current decoder state i
- Finally, **take a weighted average over all the encoder hidden states** to create a fixed-length context vector for the current decoder state
  - $c_i = \sum_j \alpha_{ij} h_j^{e}$

Thus, we finally have an encoder-decoder model with attention!



Thus, we finally have an encoder-decoder model with attention!



Summary: LSTMs, GRUs, Encoder-Decoder Models, and Attention

- Although simple ("vanilla") RNNs hold many advantages over feedforward networks for a variety of NLP tasks, **they may struggle with managing context**
- **Long short-term memory networks (LSTMs)** and **gated recurrent units (GRUs)** address this issue by introducing gating mechanisms that learn which information to forget and pass forward at different timesteps
- In their base forms, **RNN models learn one-to-one correspondences** between input and output sequences
- To learn mappings between arbitrary-length sequences instead, **encoder-decoder models first encode input into an intermediate representation, and then decode that representation to a task-specific sequence**
- They do this by making use of techniques originating in **autoregressive generation**
- Output sequences from these models can be improved by performing **beam search** or incorporating improved mechanisms for passing context between encoder and decoder states
- One way to create improved context vectors is to use an **attention mechanism**
- Attention mechanisms take the **entire encoder context** into account, **dynamically update** during the course of decoding, and can be embodied in a **fixed-size vector**