Sequence Processing with Recurrent Networks

Natalie Parde, Ph.D.

Department of Computer Science

University of Illinois at Chicago

CS 421: Natural Language Processing

Fall 2019

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

What is sequence processing?

- Automated processing of **sequential** items (e.g., words in a sentence) while taking into account **temporal** information (e.g., w_1 occurs before w_2)

Language is inherently temporal.

- **Continuous input streams** of **indefinite length**
- These sequences unfold over **time**

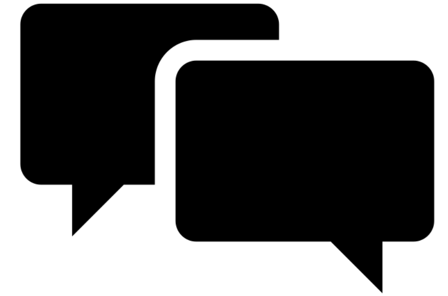
I hope there are no more midterms between now and the end of the semester.

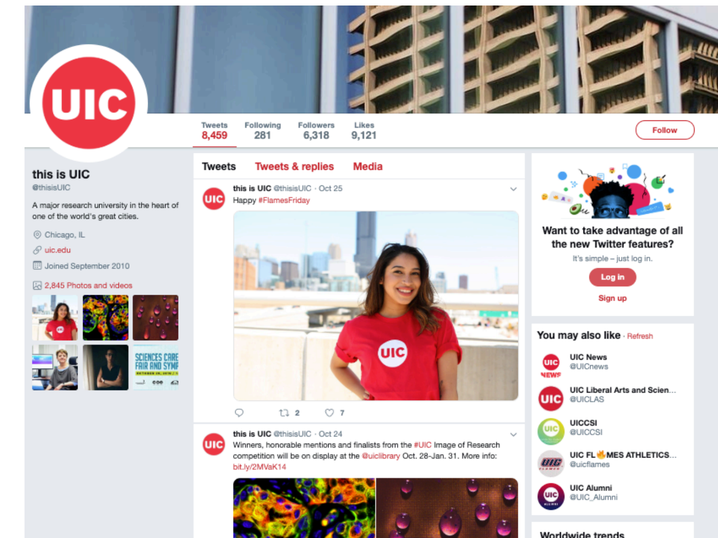
≠

no I hope there are more midterms between now and the end of the semester.

This is even evident in the way we talk about language!

- Conversation flow
- News feed
- Twitter stream





We've already looked at a few applications of sequence processing….

- Syntactic parsing
- Part of speech tagging
- Viterbi algorithm

Natalie did not **like** social events so she politely declined the party invitation

verb? noun? adjective?

…and many applications that do not incorporate temporal information.

Training	
Document	**Class**
Natalie was soooo thrilled that Usman had a famous new poem.	Sarcastic
She was totally 100% not annoyed that it had surpassed her poem on the bestseller list.	Sarcastic
Usman was happy that his poem about Thanksgiving was so successful.	Not Sarcastic
He congratulated Natalie for getting #2 on the bestseller list.	Not Sarcastic
Test	
Document	**Class**
Natalie told Usman she was soooo totally happy for him.	?

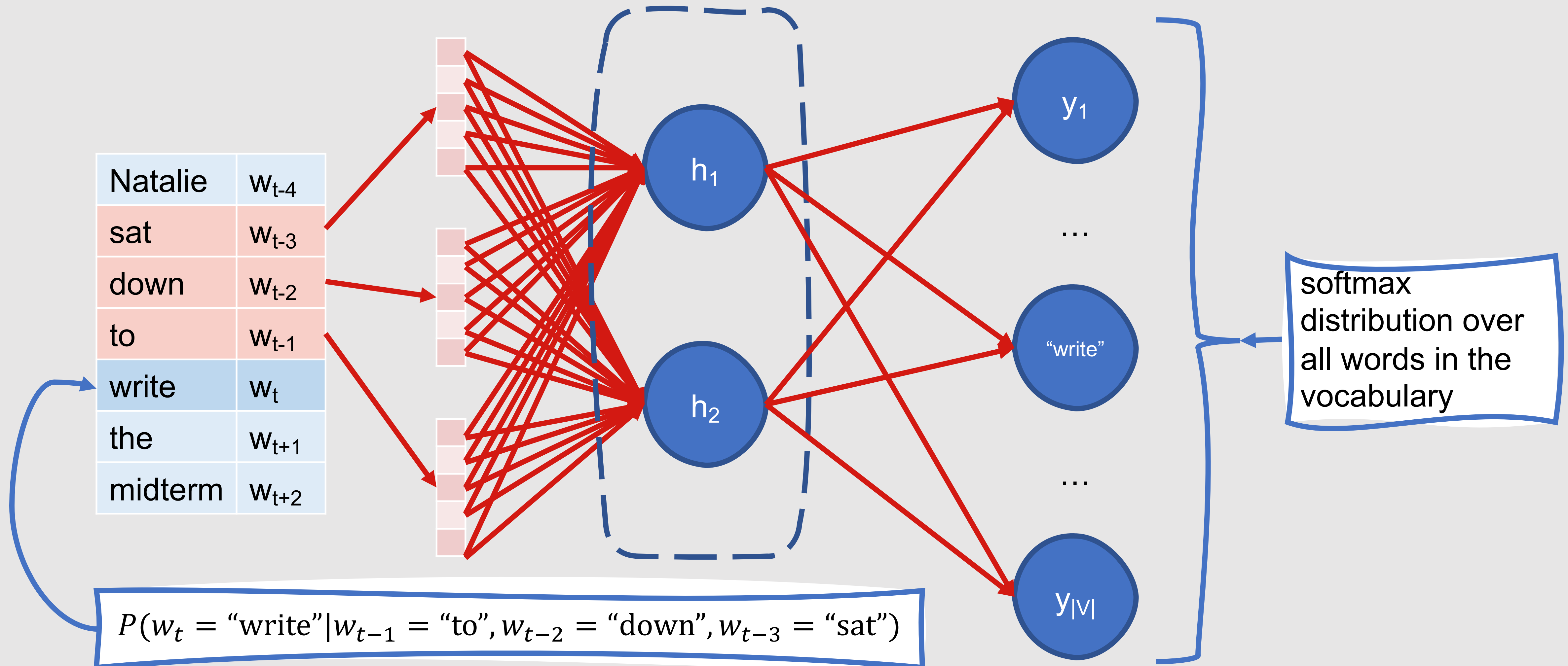
Word	P(Word|Sarcastic)	P(Word|Not Sarcastic)
Natalie	0.033	0.036
Usman	0.033	0.036
soooo	0.033	0.018
totally	0.033	0.018
happy	0.016	0.036

P(Sarcastic) = 0.5
P(Not Sarcastic) = 0.5

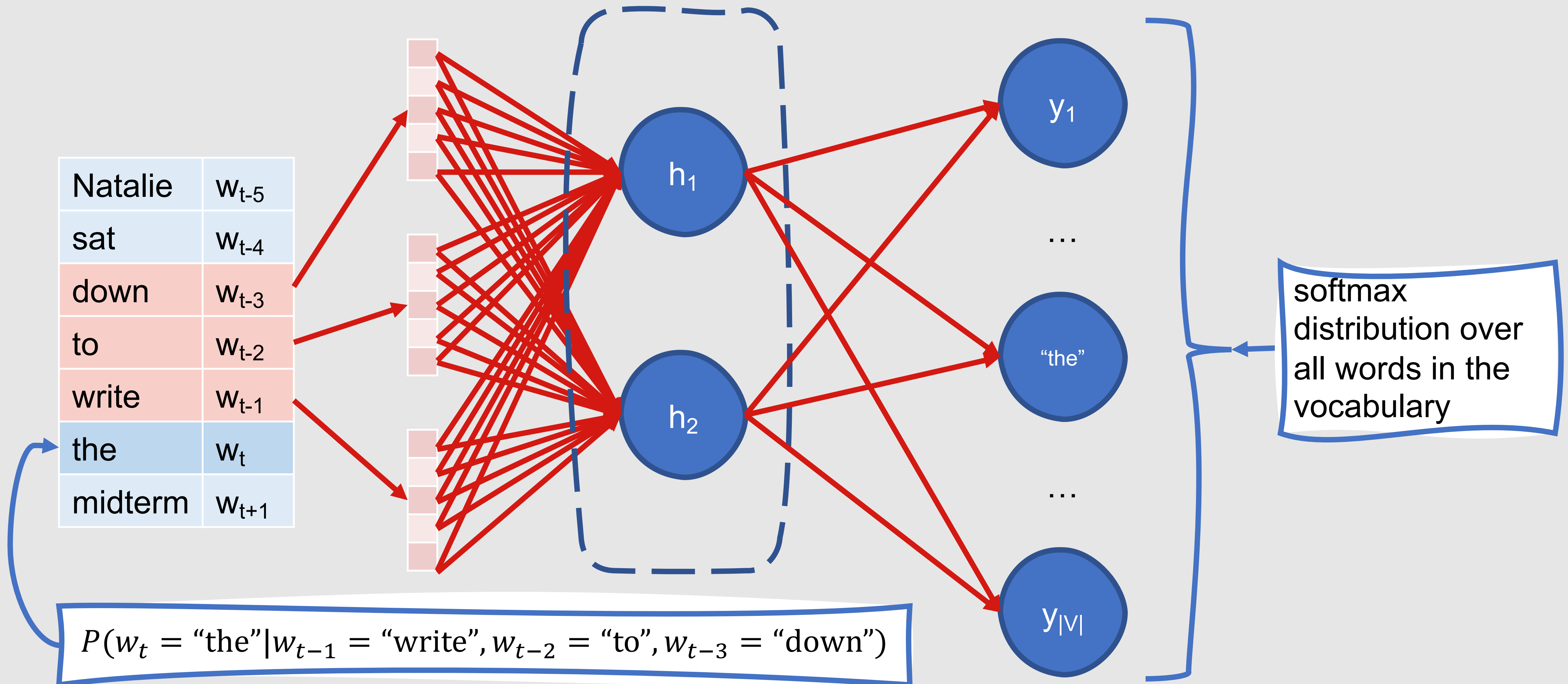
Natalie told Usman she was soooo totally happy for him.

Sarcastic

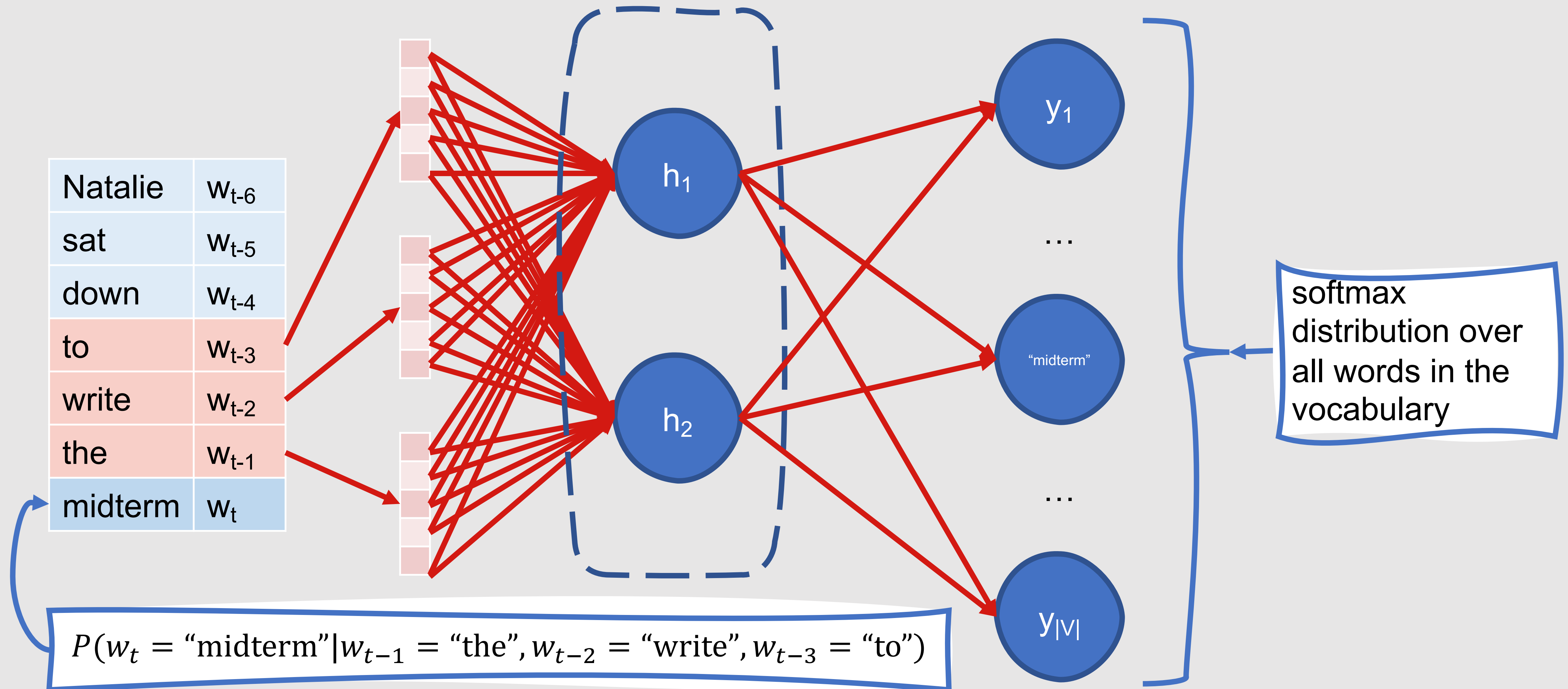
How can we adapt non-sequential models to make use of temporal information?



How can we adapt non-sequential models to make use of temporal information?



How can we adapt non-sequential models to make use of temporal information?



Disadvantages of the Sliding Window Approach

Limits the context from which information can be extracted

- Items outside the predetermined context window cannot impact the model's decision
  - What if a task requires information that can be arbitrarily distant from the point at which processing is occurring?

Makes it difficult to learn systematic patterns

- Particularly problematic when trying to learn grammatical elements, e.g., constituent parses

The solution?

- **Recurrent Neural Networks**
  - Neural networks designed specifically to handle temporal information
  - Can accept variable length inputs without the use of fixed-size windows

Recurrent Neural Networks (RNNs)

Networks that contain cycles within their connections

- The value of a unit is dependent on outputs from previous time steps as input

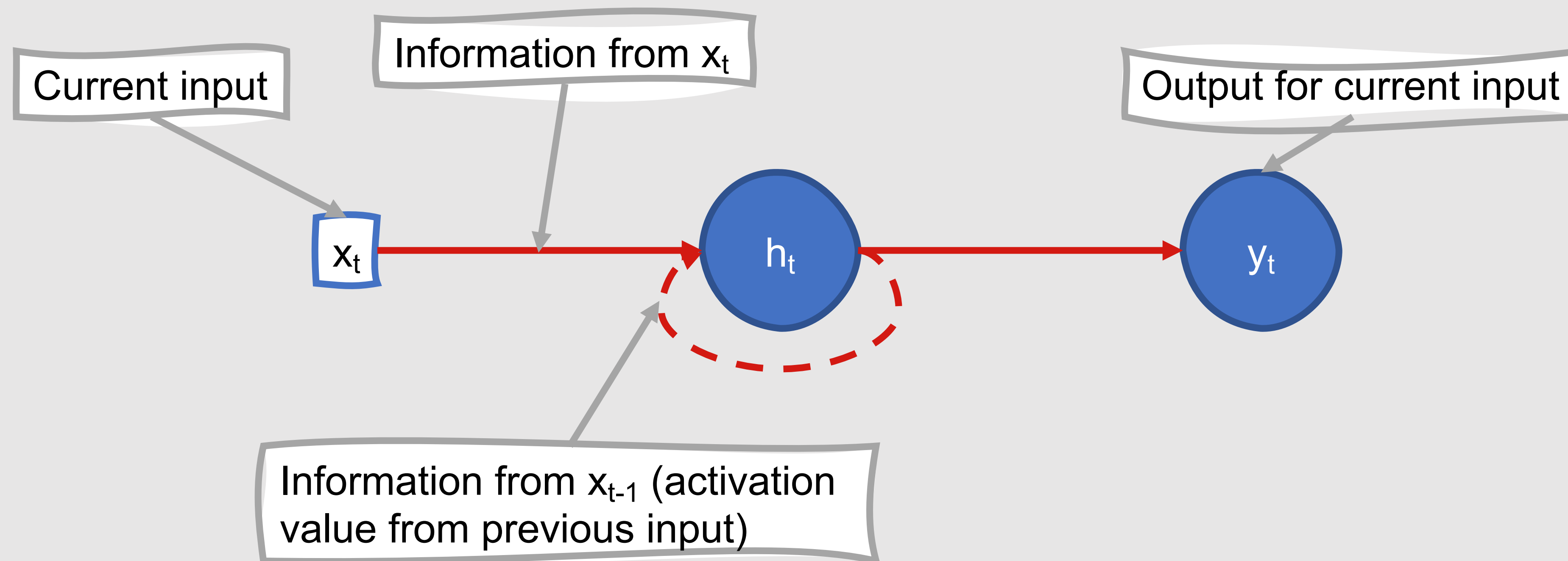
Many different variations among RNNs

- Long short-term memory network (**LSTM**)
- Bidirectional LSTM (**BiLSTM**)
- Gated recurrent unit (**GRU**)

How do RNNs differ from standard feedforward neural networks?

- Memory!
  - Loops in the network allow information to persist over time
  - Information is stored between timesteps using an internal **hidden state**, and fed back into the model the next time it reads an input
    - Some type of output is also predicted at each timestep
- New hidden states are determined as a function of the existing hidden state and the new input at the current timestep
  - This function remains the same across timesteps

Simple RNN



Thus, hidden layers in RNNs are more complex than in feedforward networks.

Capable of encoding outputs from earlier timesteps, which can thereby serve as additional context

Makes decisions based on both current input and outputs from prior timesteps

Does not impose a fixed-length limit on this prior context (can include information extending back to the beginning of the sequence)

However, computation units still perform the same core actions.

Given:

- Input vector
- (New!) activation values for the hidden layer for the previous time step

Compute:

- Weighted sum of inputs

Most Significant Change

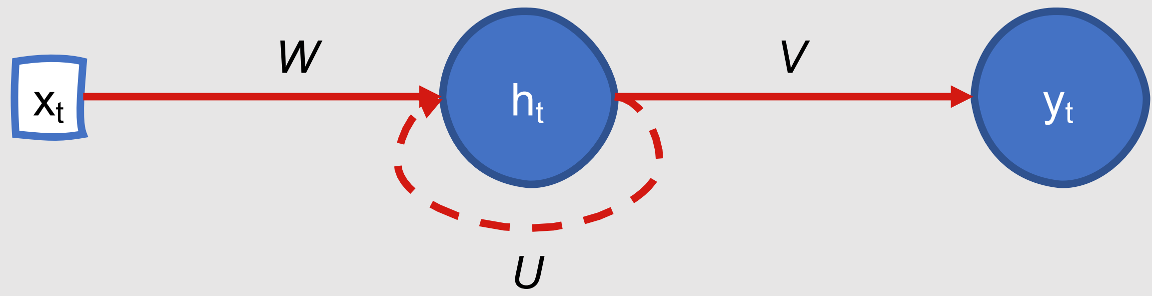
- New set of weights, *U*, that connect the hidden layer from the previous time step to the current hidden layer
- These weights determine how the network should make use of prior context in determining the output for the current input
- Just like with feedforward networks, weights are trained using backpropagation

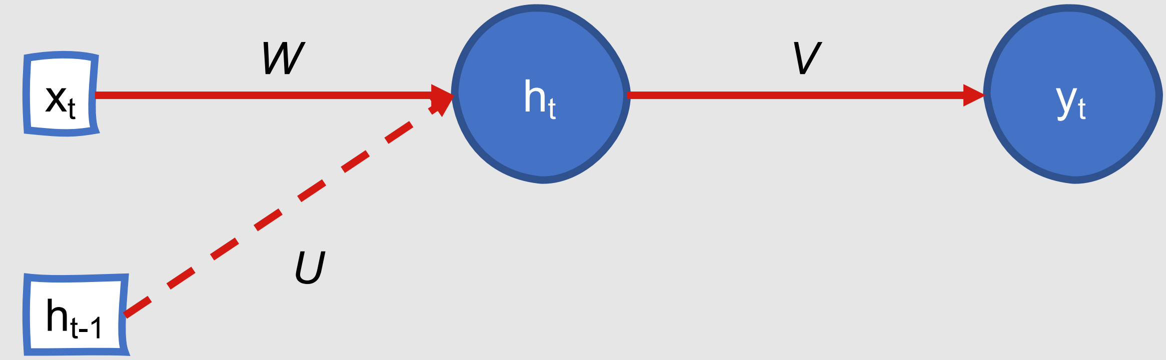
Formal Definition

- **Forward inference** (mapping a sequence of inputs to a sequence of outputs) is quite similar to what we've seen with feedforward networks!
- Recall the basic set of equations for a feedforward neural network:
  - $\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b})$
  - $\mathbf{z} = U\mathbf{h}$
  - $y = \text{softmax}(\mathbf{z})$

Formal Definition

- The only change we need to make to the original set of equations is to add the additional (weights X activation values from previous timestep) product to the current (weights X inputs) product
  - $\mathbf{h} = \sigma(W\mathbf{x_t} + U\mathbf{h_{t-1}} + \mathbf{b})$
  - $\mathbf{z} = V\mathbf{h_t}$
  - $y = \text{softmax}(\mathbf{z})$
- Note that the weight matrices *W*, *U*, and *V* (the renamed weight matrix for the output layer) are shared across all timesteps





How would this process look, illustrated as a feedforward network?

Formal Algorithm

```
h0 ← 0  # Initialize activations from the hidden layer to 0
for i ← 1 to length(x) do:  # Iterate through each input element in temporal order
    hi ← g(Uhi-1 + Wxi + b)  # Bias vector is optional
    yi ← f(Vhi)
```

$h_0 \leftarrow 0$ # Initialize activations from the hidden layer to 0

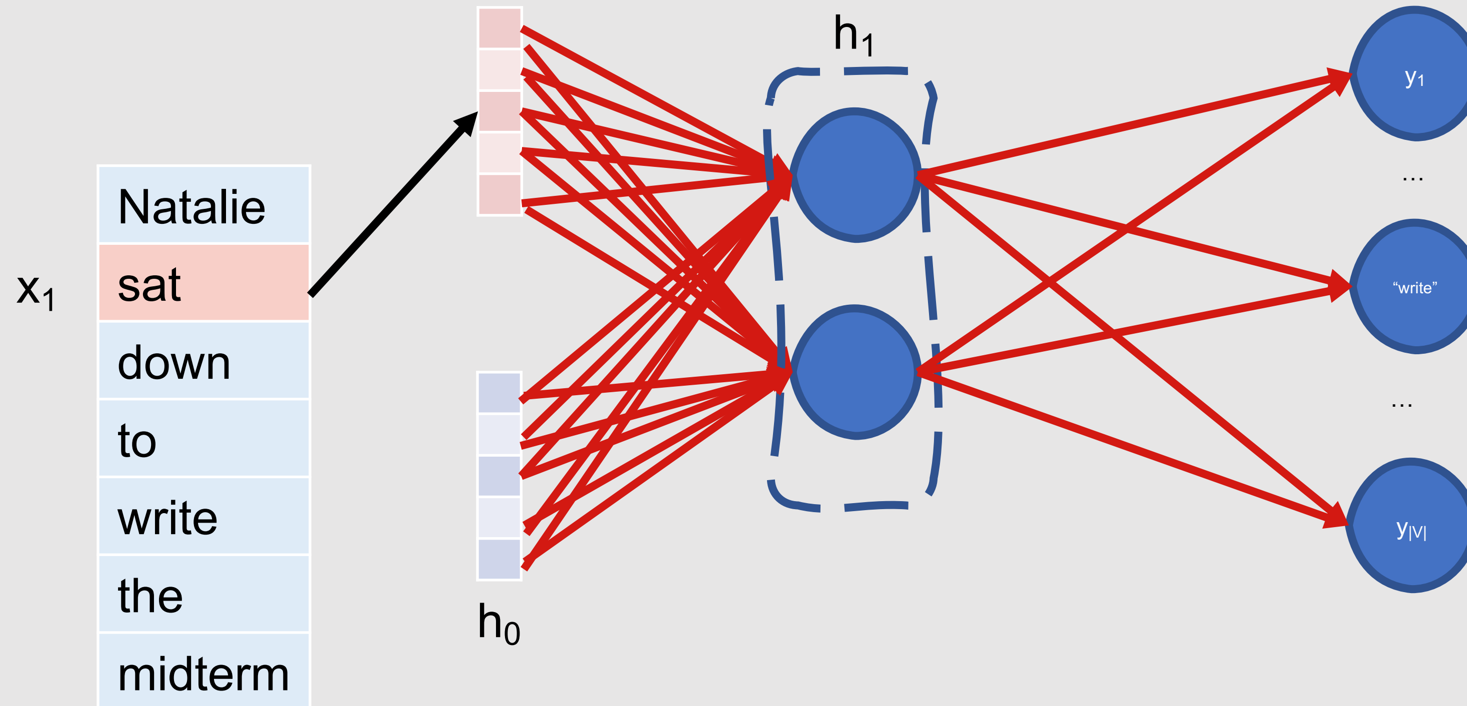
for i ← 1 to length(x) do: # Iterate through each input element in temporal order

$\mathbf{h}_i \leftarrow g(U\mathbf{h}_{i-1} + W\mathbf{x}_i + \mathbf{b})$ # Bias vector is optional

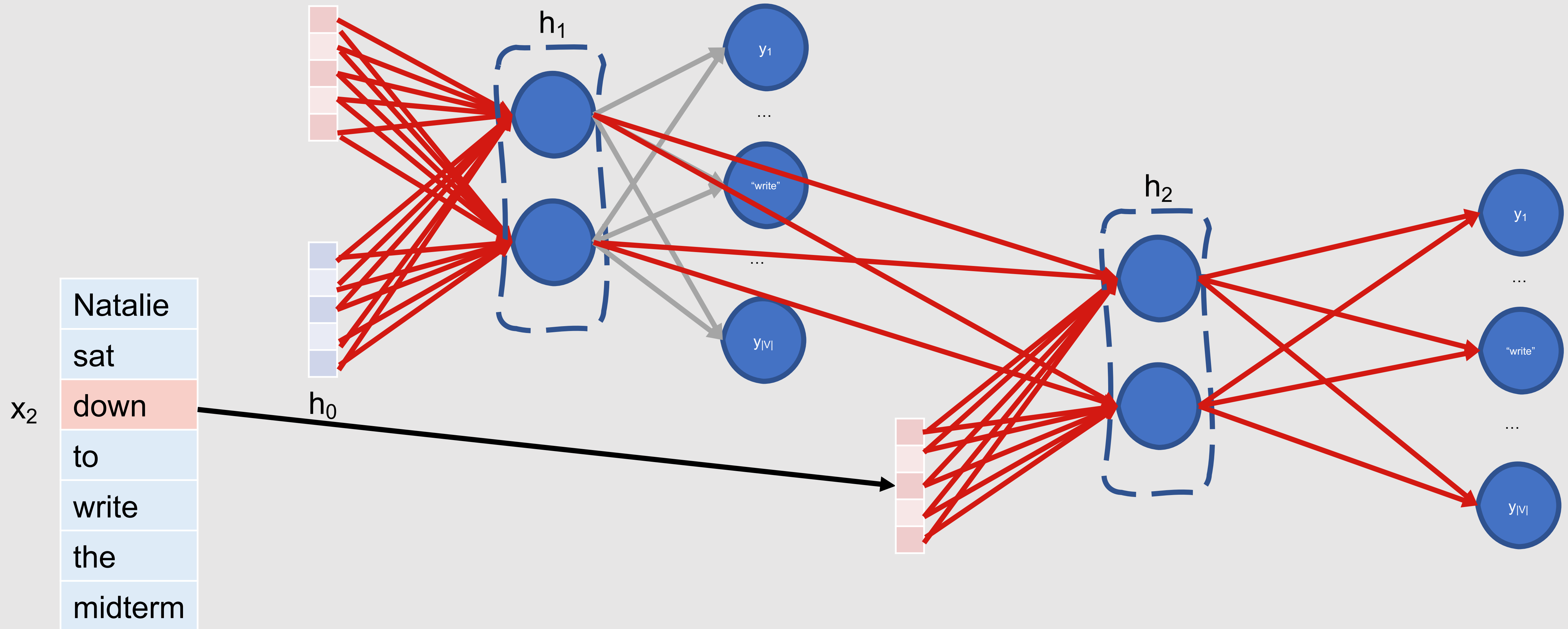
$y_i \leftarrow f(V\mathbf{h}_i)$

New values for h and y are calculated with each time step!

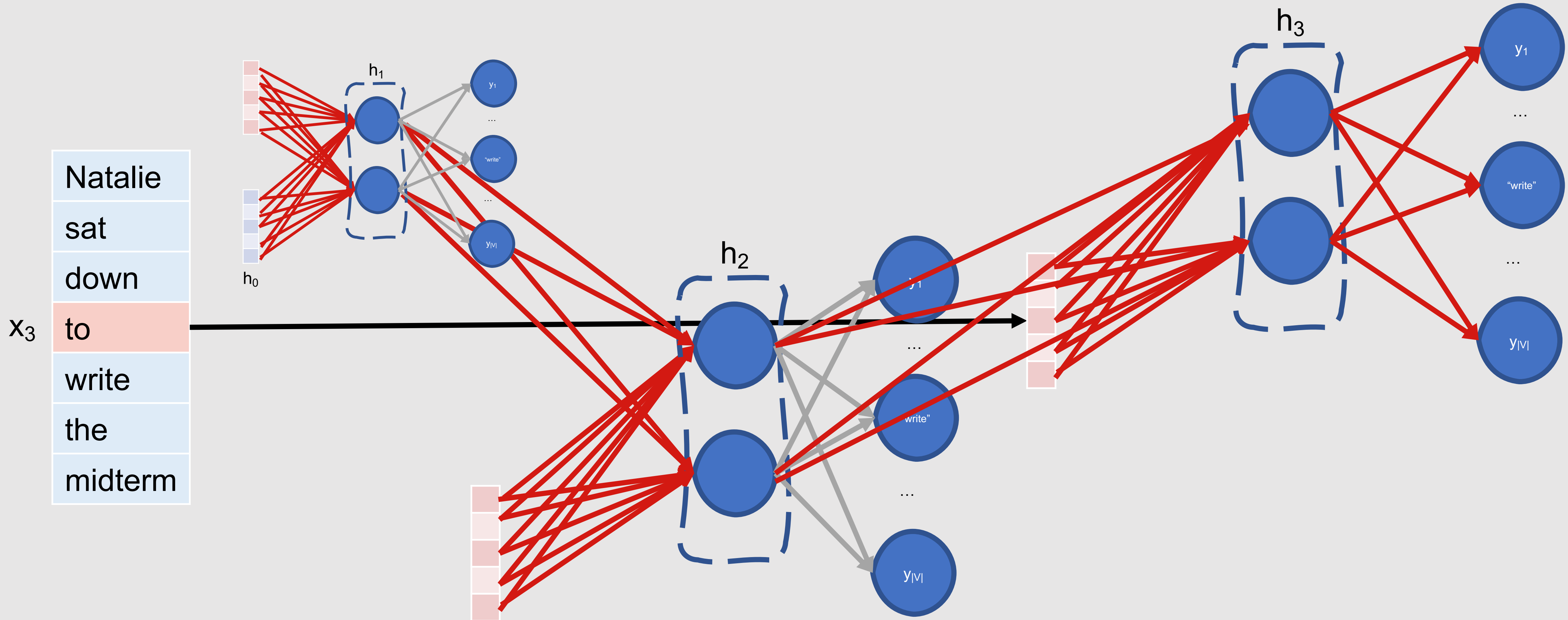
Unrolling a simple RNN….



Unrolling a simple RNN….



Unrolling a simple RNN….



Training RNNs

- Same core elements:
  - Loss function
  - Optimization function
  - Backpropagation
- One extra set of weights to update
  - Previous hidden layer to current hidden layer (U)

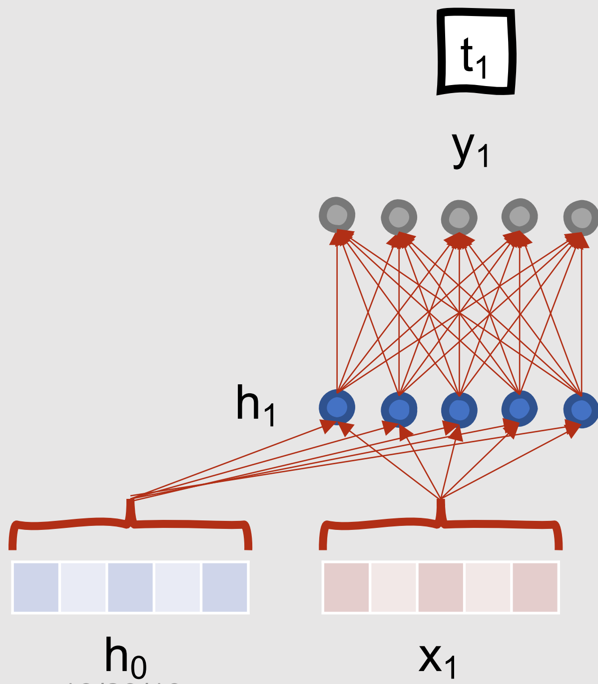
Additional Considerations for Computing Loss

- To compute the loss at time *t* we need the hidden layer from time *t*-1
- To assess the overall error accruing to $\mathbf{h}_t$, we need the current output as well as outputs that will follow

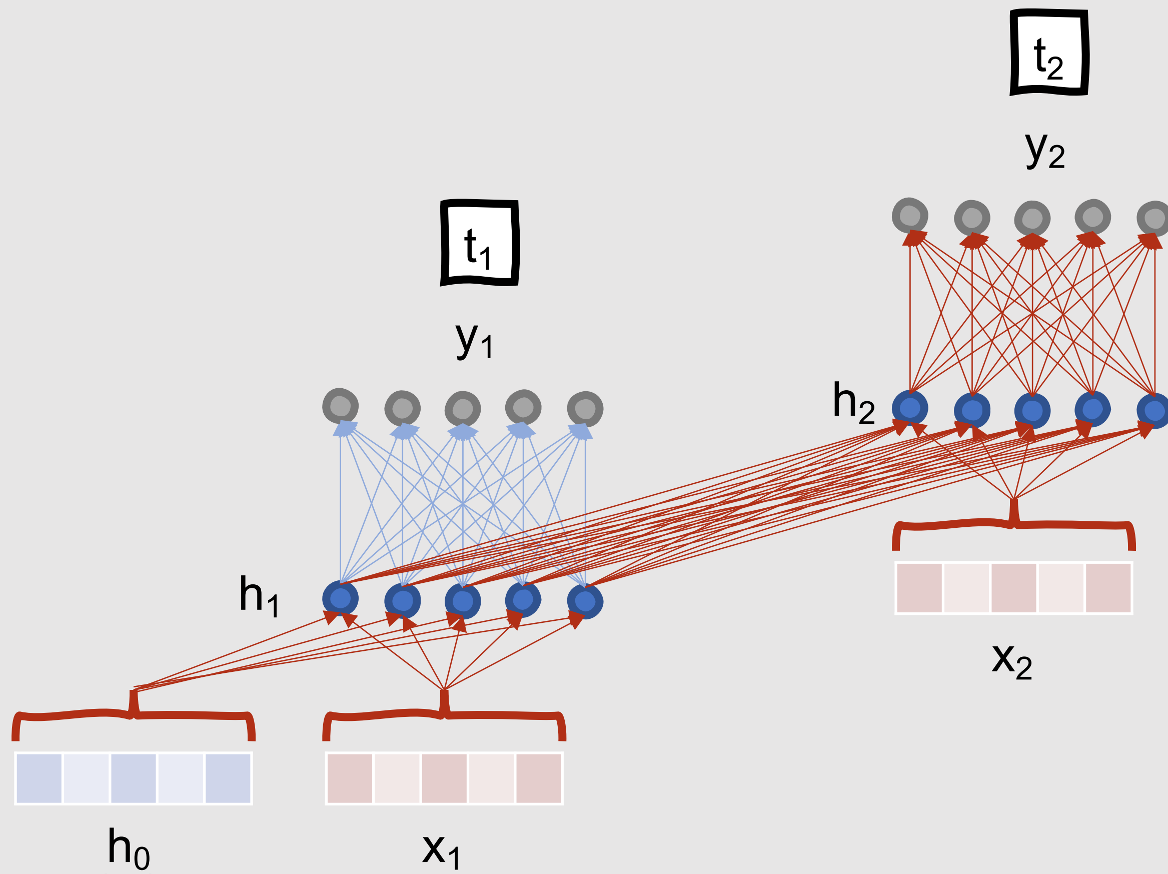
Backpropagation Through Time

- Two-pass algorithm for training RNNs
- First pass: **Perform forward inference**
  - Compute h_t and y_t at each step in time
  - Compute the loss at each step in time
- Second pass: **Process the sequence in reverse**
  - Compute the required error gradients at each step backward in time

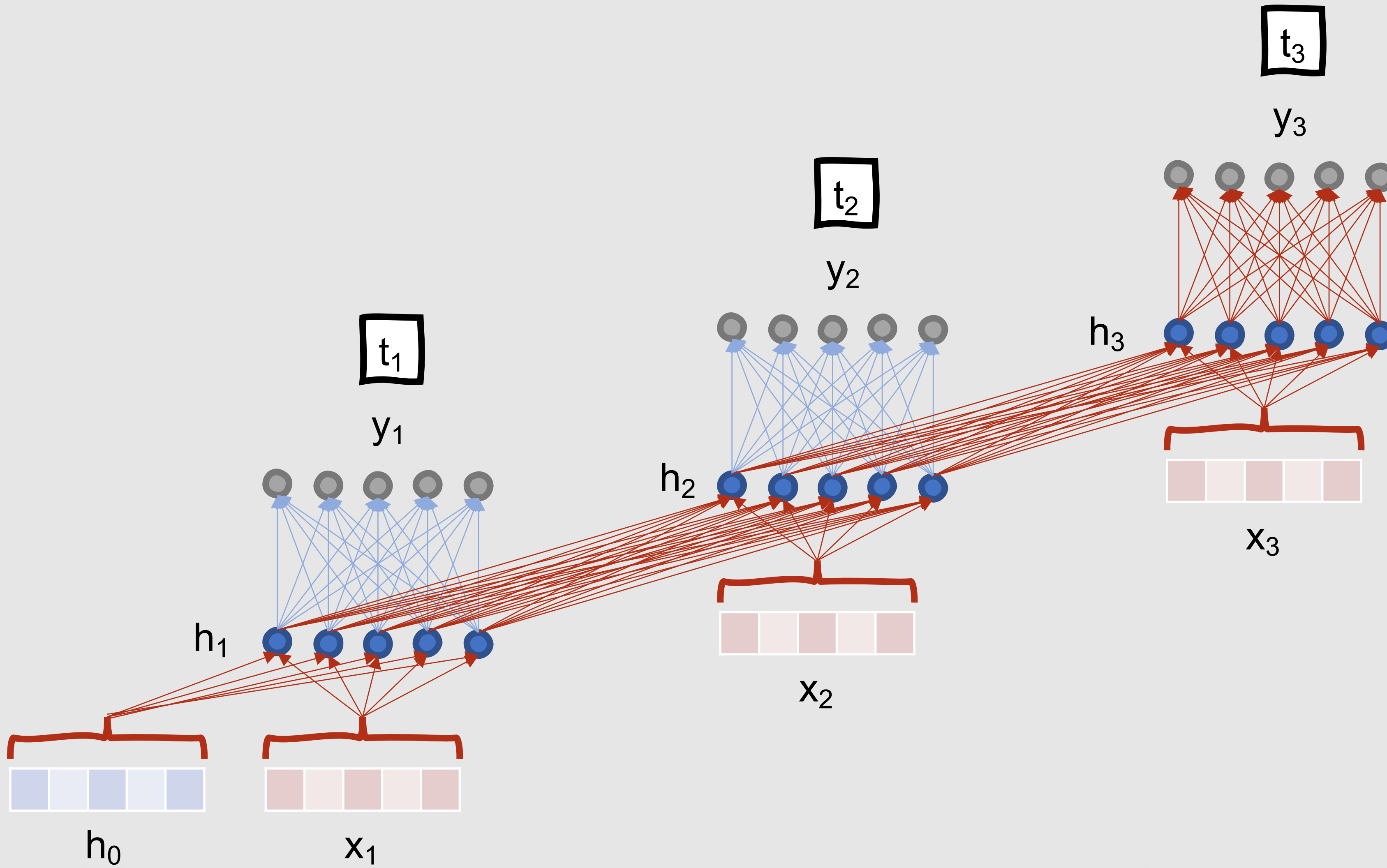
Forward Pass



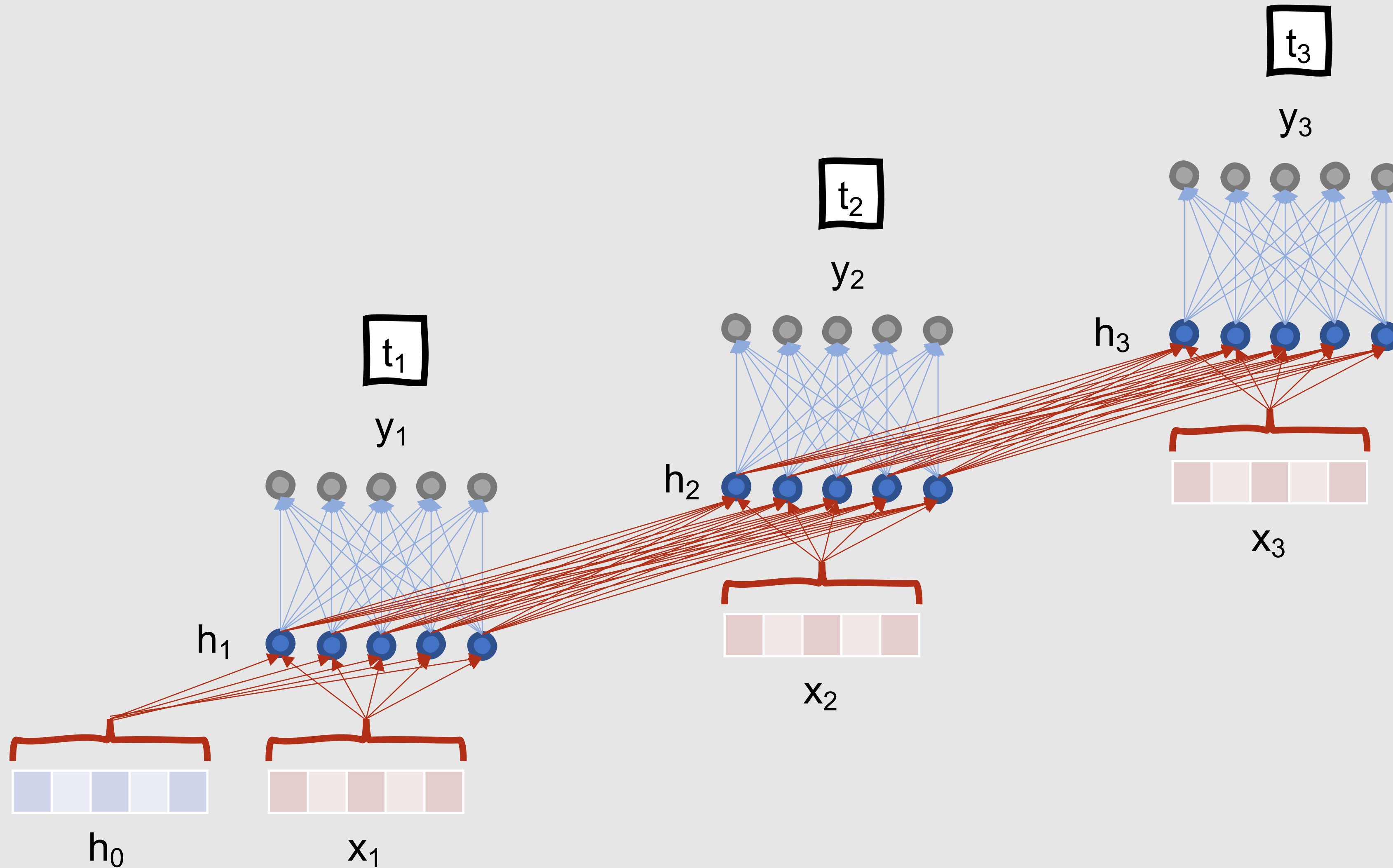
Forward Pass



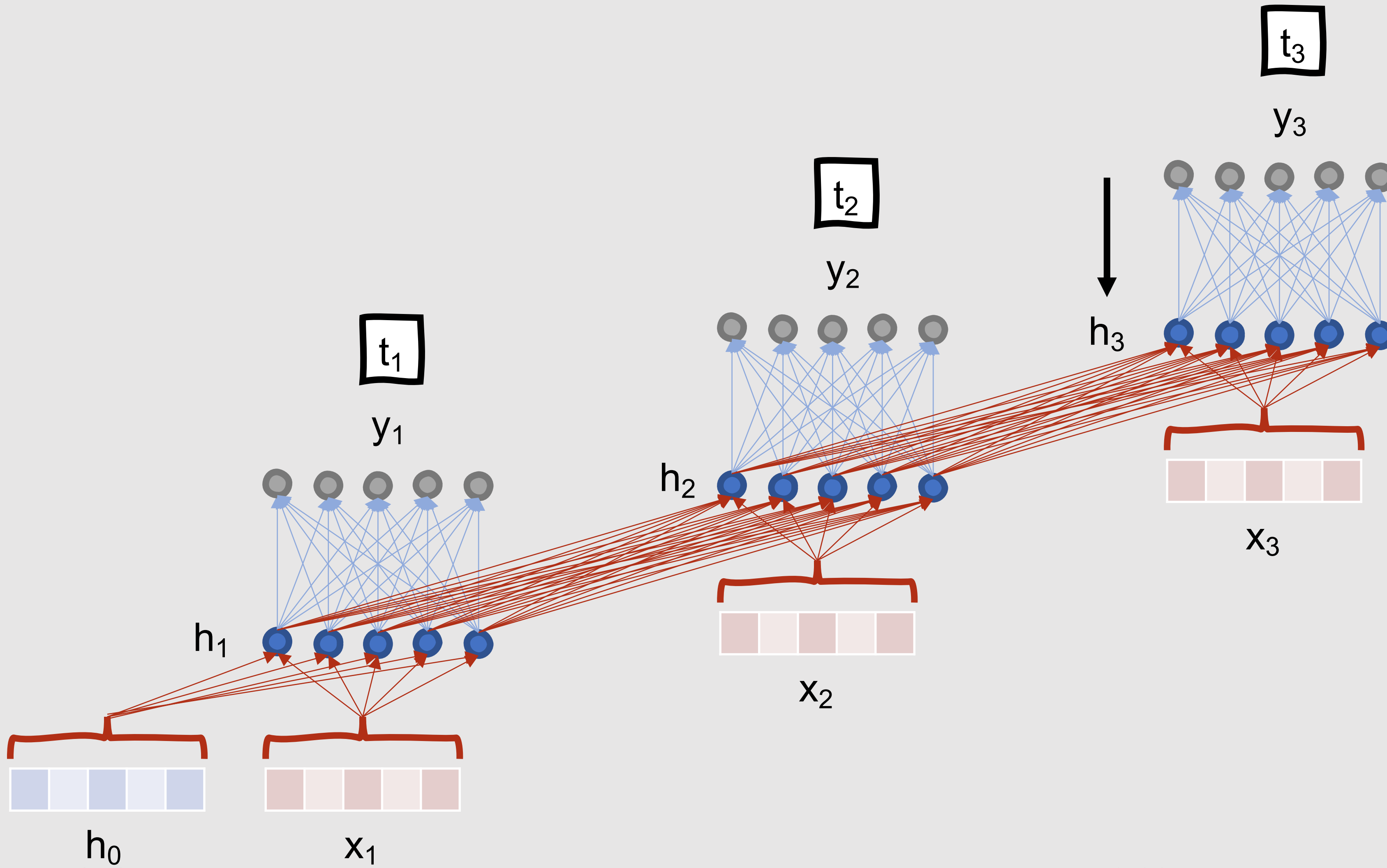
Forward Pass



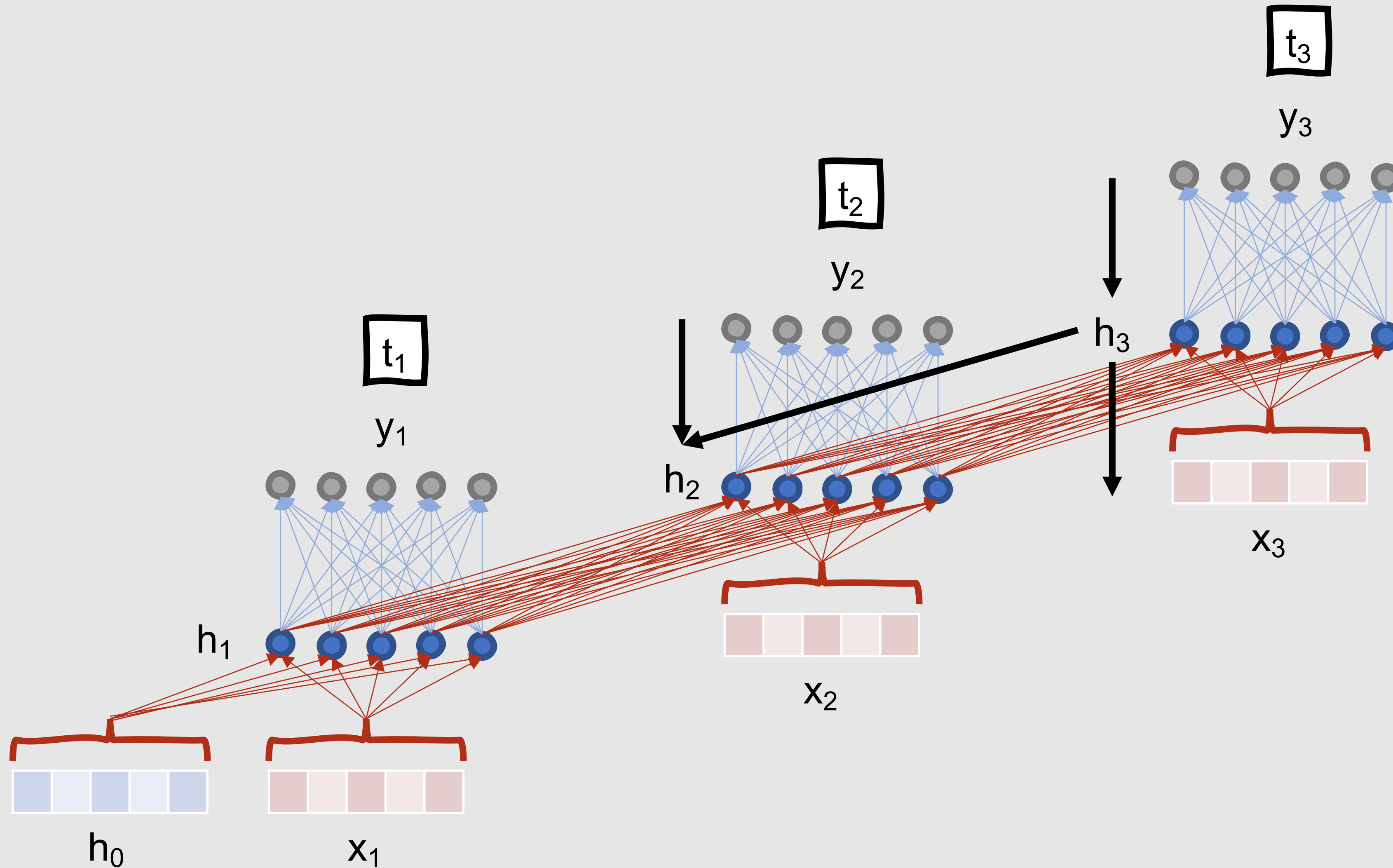
...and now...



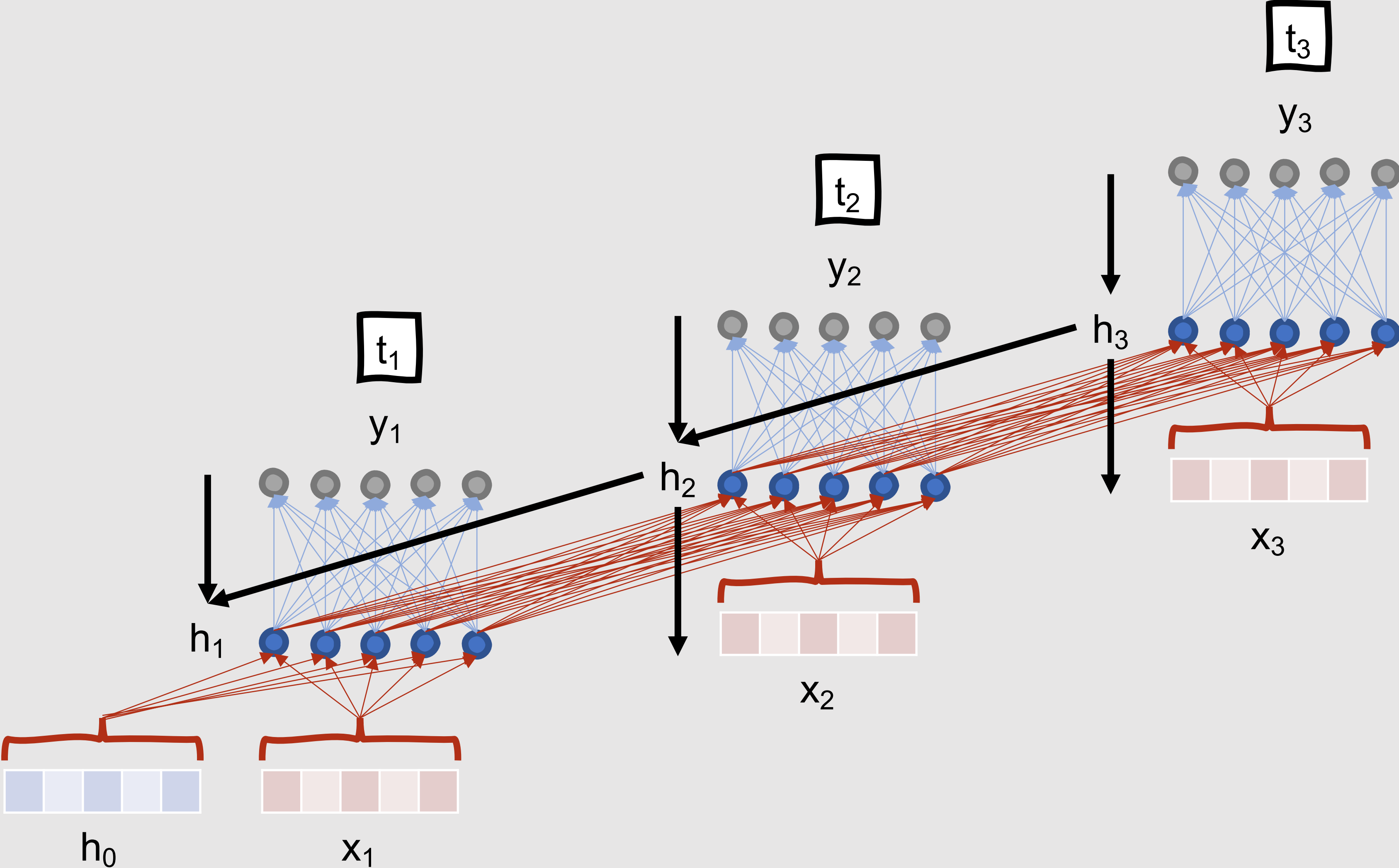
Backward Pass



Backward Pass



Backward Pass

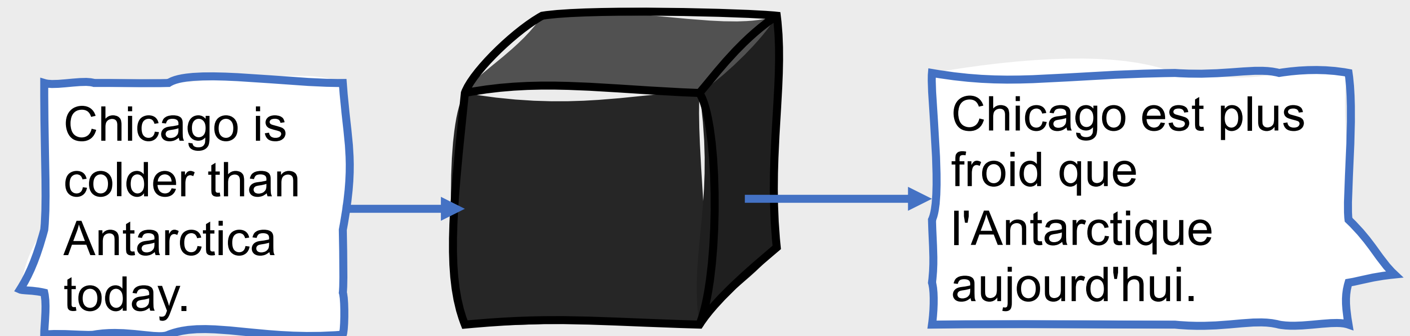


Applications of Recurrent Neural Networks

- Language modeling
- Part-of-speech tagging
- Sequence classification tasks

RNNs also form the basis for sequence-to-sequence approaches.

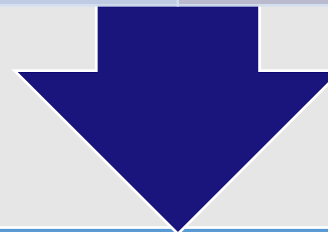
- **Sequence-to-sequence (seq2seq):** Model input and output are both sequences
- Useful for:
  - Text summarization
  - Machine translation
  - Question answering



Recurrent Neural Language Models

What we've seen so far:

N-gram language models	Feedforward networks with sliding windows



Both of these attempt to predict the next word in a sequence given a prior context of fixed length

The problem with *n*-gram and feedforward language models?

In both approaches, model quality is dependent on context size

Anything outside the fixed context window has no impact on the model's decision!

Recurrent Neural Language Models

- Recurrent neural language models process sequences one word at a time, as seen in the previous slides
- This means that they **avoid constraining the context size**
- The **hidden state embodies information about all of the preceding words**, all the way back to the beginning of the sequence

Recurrent Neural Language Models

- At each timestep:
  1. **Retrieve an embedding** for the current input word
  2. **Combine the weighted sums** of (a) the input embedding values and (b) the activations of the hidden layer from the previous step, to compute a new set of activation values from the hidden layer
  3. **Generate a set of outputs** based on the activations from the hidden layer
  4. **Pass the outputs through a softmax function** to generate a probability distribution over the entire vocabulary

Recurrent Neural Language Models

- Recurrent models can be trained using the same data used to train *n*-gram and feedforward language models
  - Collection of representative text
- Correct class → still the word that actually comes next in the data
- Task: Predict the next word in a sequence given all previous words, rather than only those in a context window of size *n*

How can we generate text with neural language models?

Model Completion (Machine-Written, 10 Tries): The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

Pérez and the others then ventured further into the valley. "By the time we reached the top of one peak, the water looked blue, with some crystals on top," said Pérez.

Pérez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them – they were so close they could touch their horns.

While examining these bizarre creatures the scientists discovered that the creatures also spoke some fairly regular English. Pérez stated, "We can see, for example, that they have a common 'language,' something like a dialect or dialectic."

Dr. Pérez believes that the unicorns may have originated in Argentina, where the animals were believed to be descendants of a lost race of people who lived there before the arrival of humans in those parts of South America.

While their origins are still unclear, some believe that perhaps the creatures were created when a human and a unicorn met each other in a time before human civilization. According to Pérez, "In South America, such incidents seem to be quite common."

However, Pérez also pointed out that it is likely that the only way of knowing for sure if unicorns are indeed the descendants of a lost alien race is through DNA. "But they seem to be able to communicate in English quite well, which I believe is a sign of evolution, or at least a change in social organization," said the scientist.

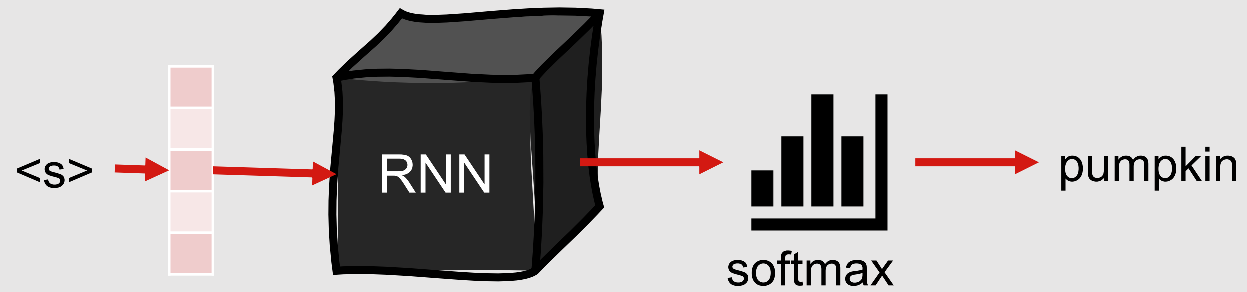
Generation with Neural Language Models

1. Sample the first word in the output from the softmax distribution that results from using the **beginning of sentence marker** (<s>) as input
2. Get the embedding for that word
3. Use it as input to the network at the next time step, and sample the following word as in (1)
4. Repeat until the **end of sentence marker** (</s>) is sampled, or a fixed length limit is reached

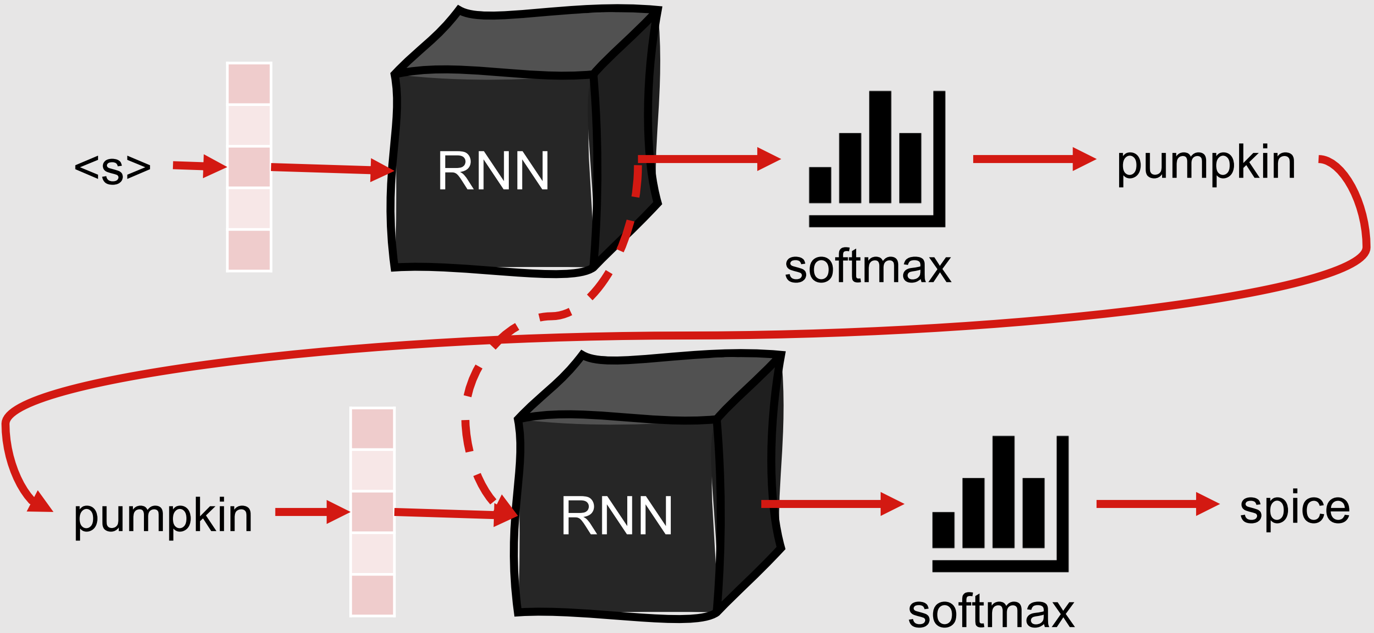
Autoregressive Generation

- This technique is referred to as **autoregressive generation**
  - Word generated at each timestep is conditioned on the word generated previously by the model
- Key to successful autoregressive generation?
  - Prime the generation component with **appropriate context** (e.g., something more useful than <s>)

Autoregressive Generation



Autoregressive Generation



Autoregressive Generation

<s>
RNN
softmax
pumpkin

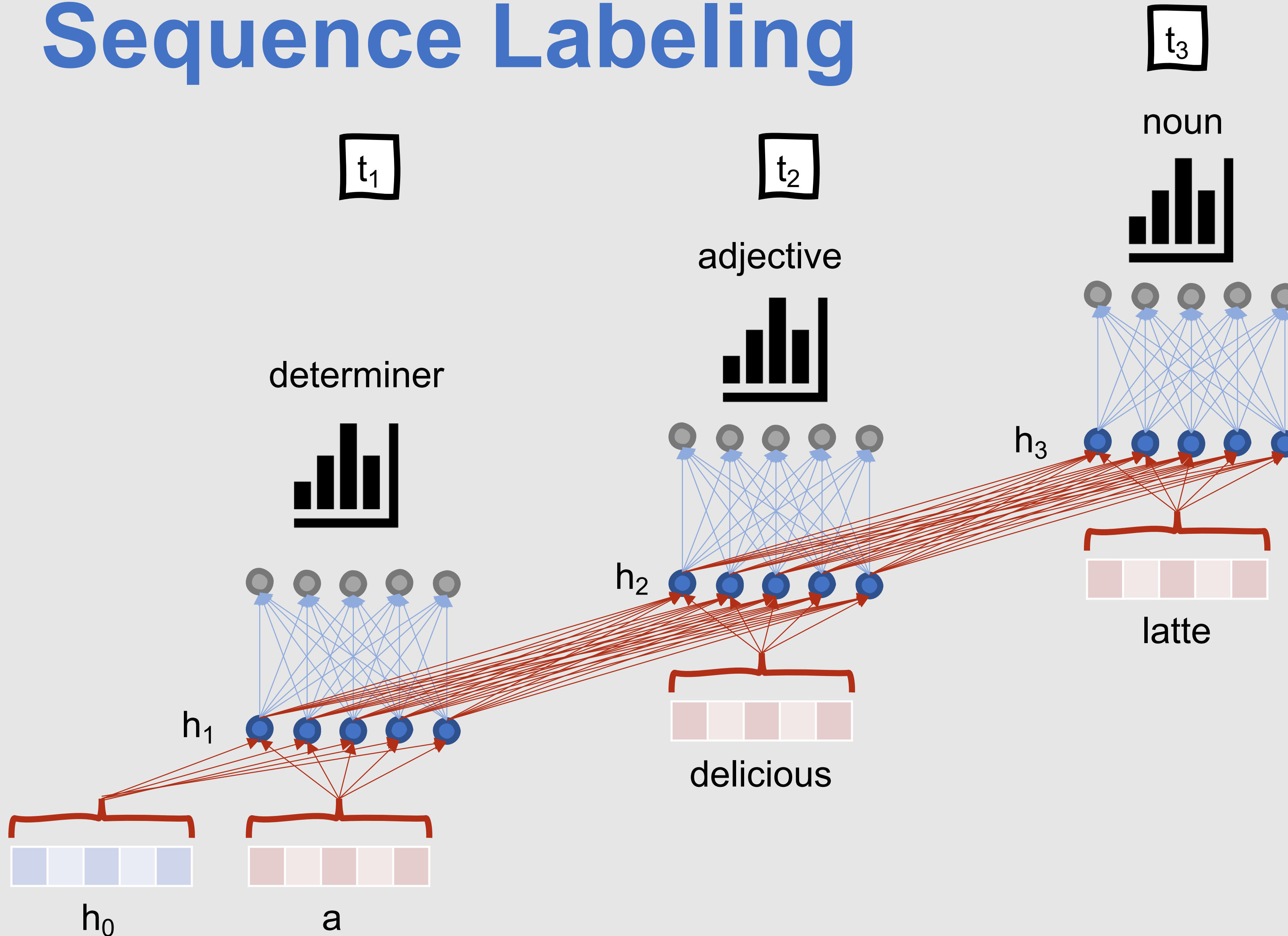
pumpkin
RNN
softmax
spice

spice
RNN
softmax
latte

Sequence Labeling

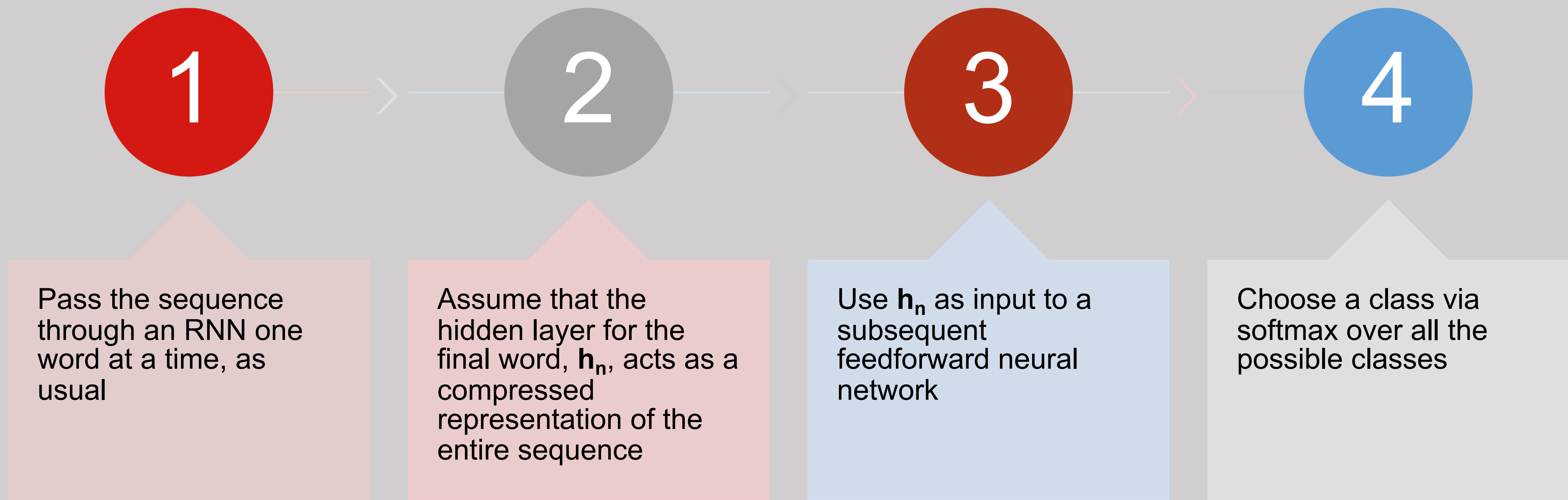
- Task: Given a fixed set of labels, assign a label to each element of a sequence
  - Example: Part-of-speech tagging
- In an RNN:
  - Inputs → word embeddings
  - Outputs → label probabilities generated by the softmax (or other activation) function over the set of all labels

Sequence Labeling



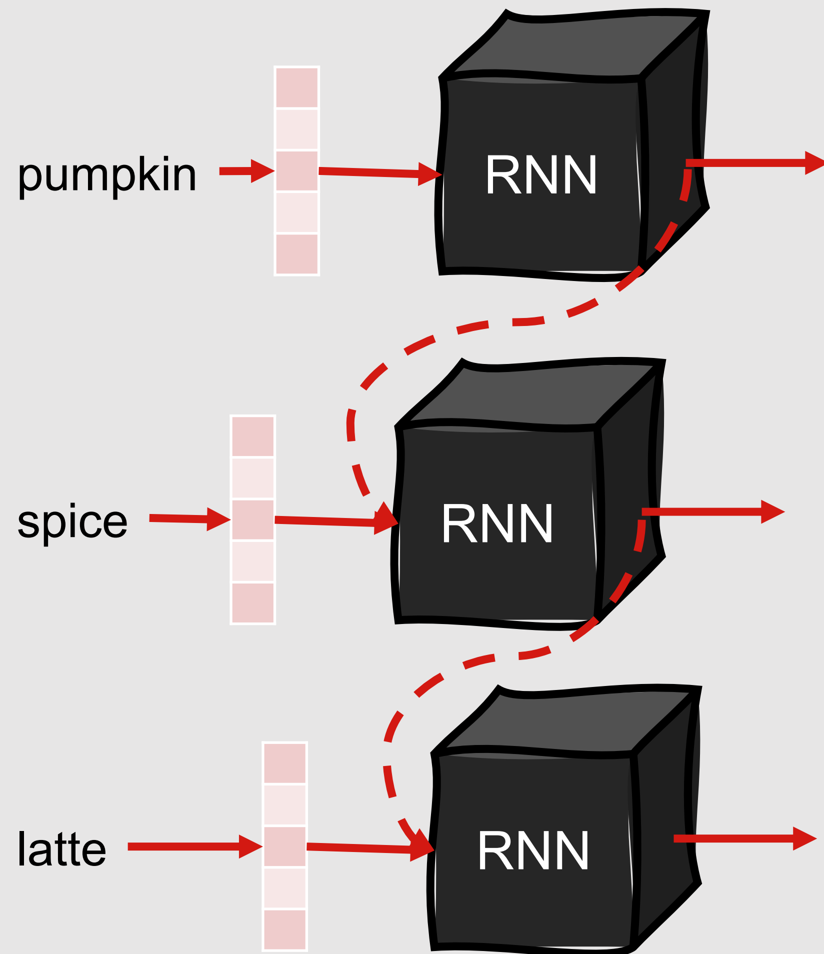
Sequence Classification

- Task: Given an input sequence, **assign the entire sequence to a class** (rather than the individual tokens within it)
- Useful for:
  - Document-level topic classification
  - Spam detection
  - Message routing
  - Deception detection
  - Any other applications for which sequences of text are classified as belonging to one of a small number of categories!

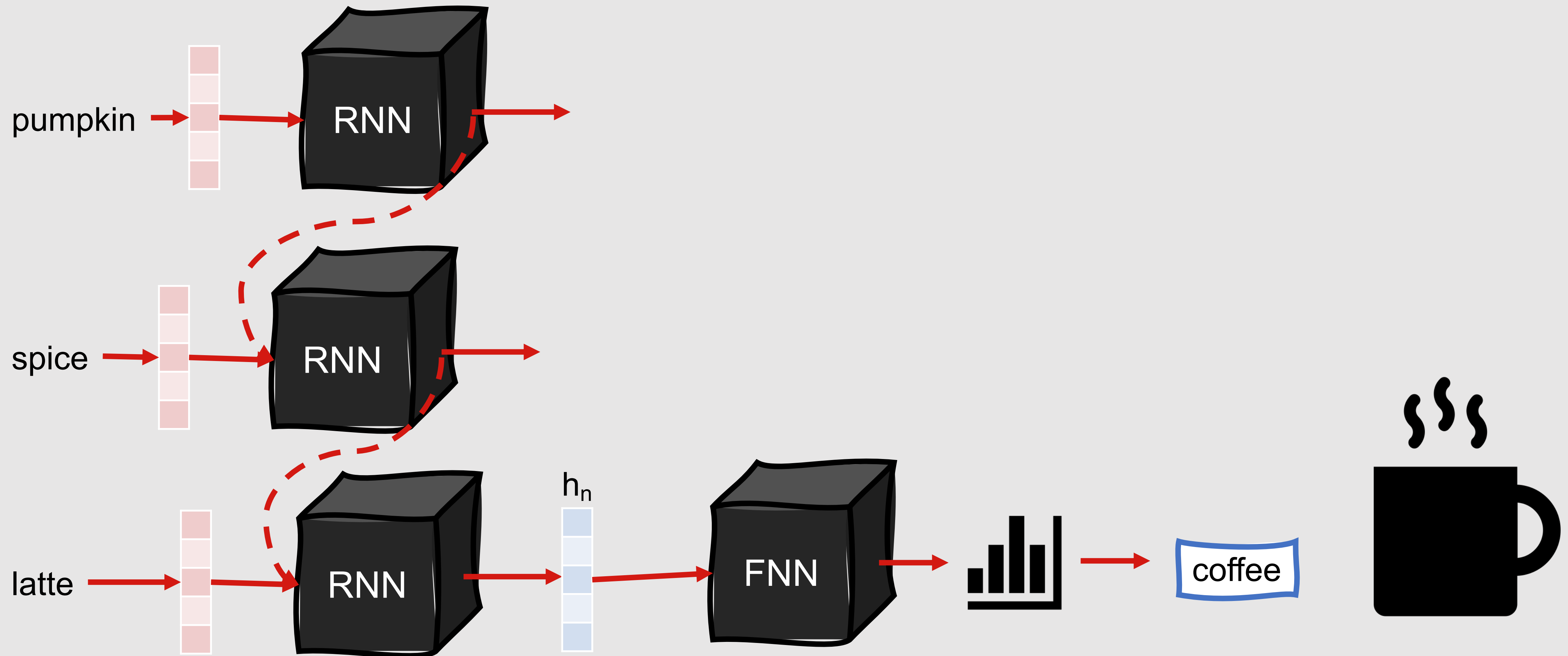


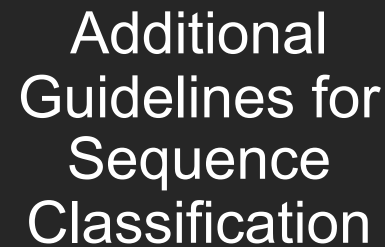
How to use RNNs for sequence classification?

Sequence Classification



Sequence Classification





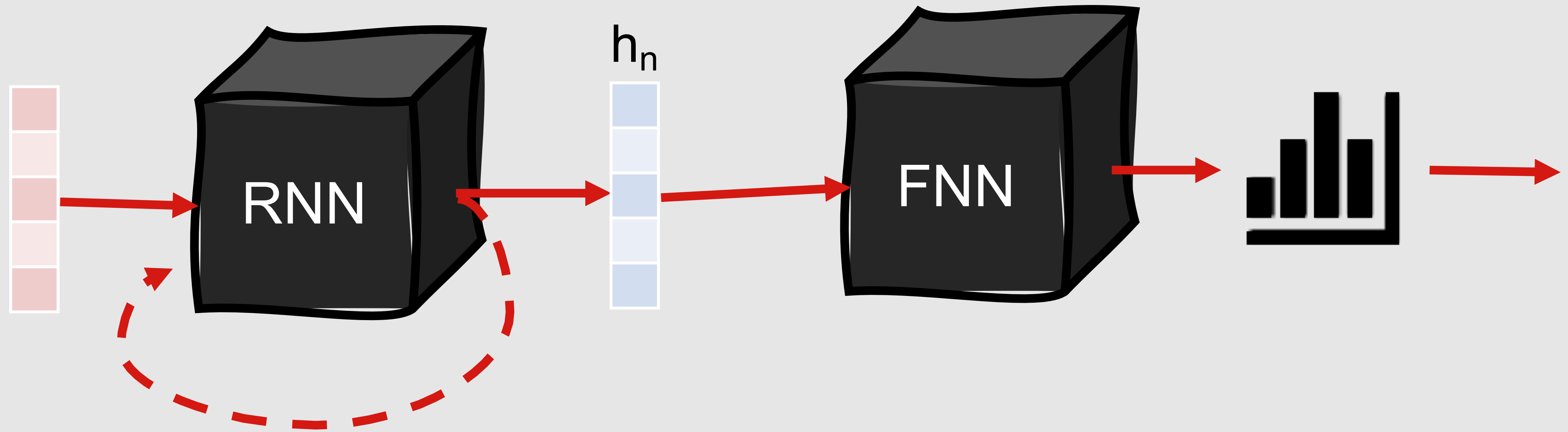
Additional Guidelines for Sequence Classification

No intermediate outputs (e.g., “pumpkin” = coffee) for words in the sequence that precede the last word → no loss terms associated with those elements

Loss function is based entirely on final classification task!

However, errors are still propagated backward all the way through the RNN

The process of adjusting weights the entire way through the network based on the loss from a downstream application is often referred to as **end-to-end training**

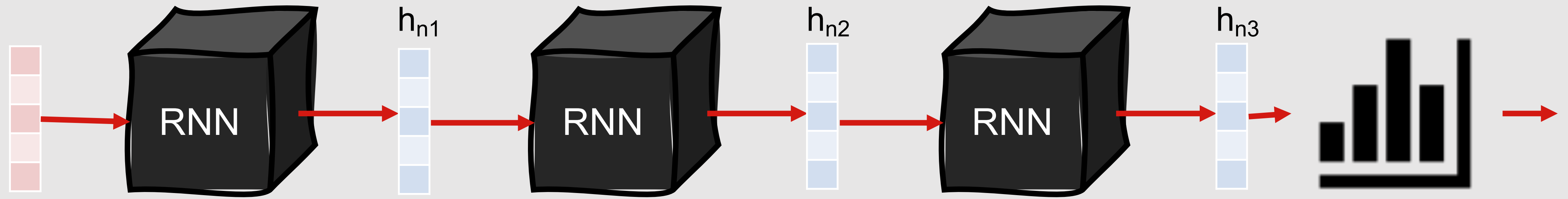


Deep Recurrent Neural Networks

- As demonstrated with sequence classification, it is possible to combine neural networks to form more complex architectures
  - RNN + Feedforward Neural Network
  - Two RNNs
  - Possibilities are (theoretically) endless!

Stacked RNNs

- Use the entire sequence of outputs from one RNN as the input sequence to another
- Capable of outperforming single-layer networks
- Why?
  - Having more layers allows the network to learn representations at differing levels of **abstraction** across layers
    - Early layers → more fundamental properties
    - Later layers → more meaningful groups of fundamental properties

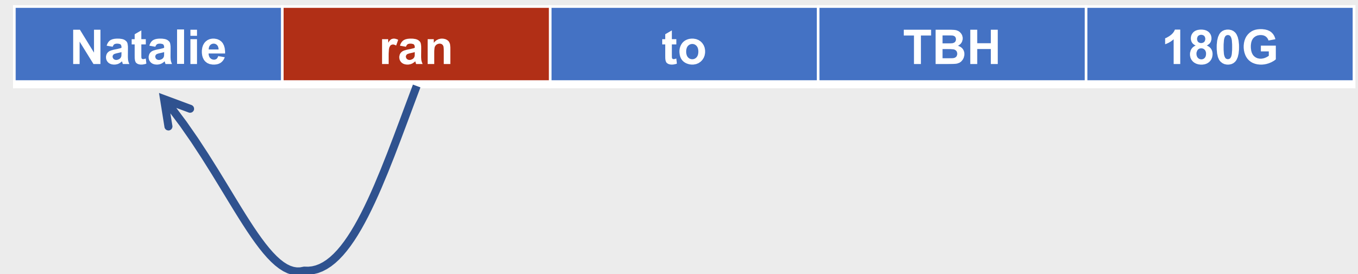


Stacked RNNs

- Optimal number of RNNs to stack together?
  - Depends on application and training set
- More RNNs in the stack → increased training costs

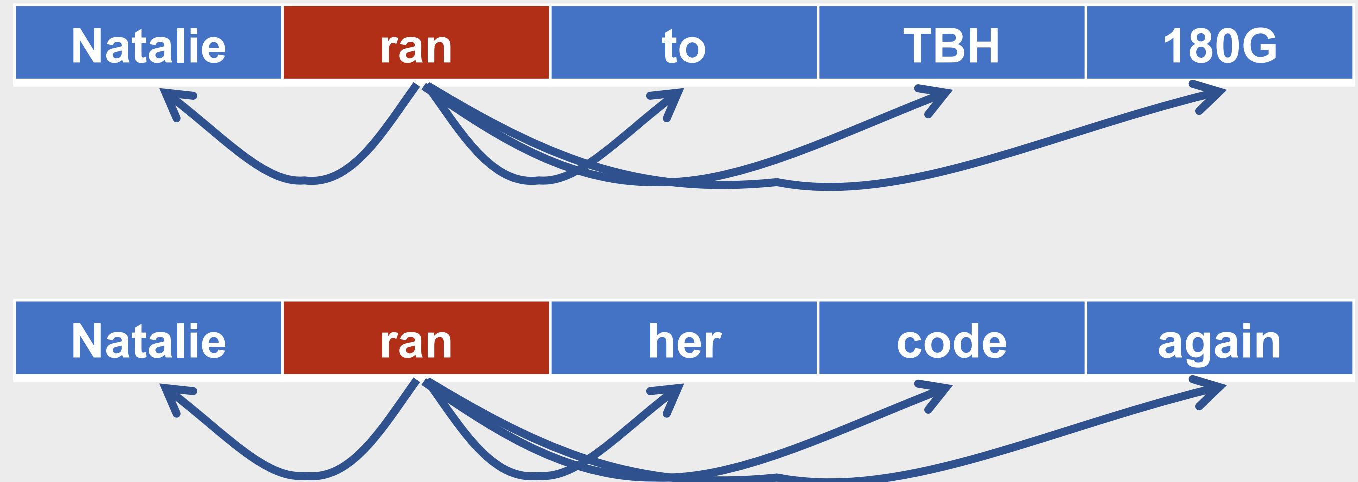
Bidirectional RNNs

- Simple RNNs only consider the information in a sequence leading up to the current timestep
  - $h_t^f = RNN_{forward}(x_1^t)$
    - h_t^f corresponds to the normal hidden state at time *t*
- This could be visualized as the context to the left of the current time



Bidirectional RNNs

- However, in many cases the context after the current timestep (to the right of the current time) could be useful as well!
- In many applications we have access to the entire input sequence at once anyway



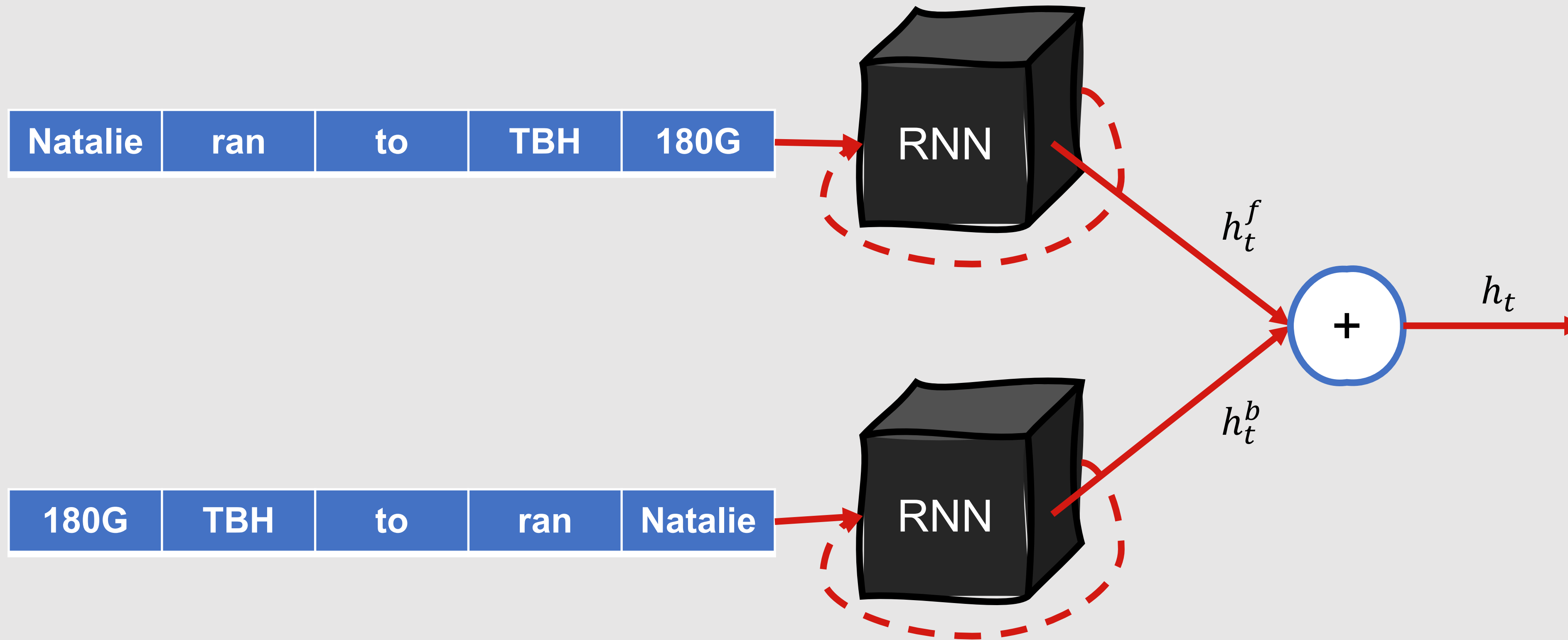
Bidirectional RNNs

- How can we make use of information from both sides of the current timestep?
- Simple solution:
  - Train an RNN on an input sequence in **reverse**
    - $h_t^b = RNN_{backward}(x_t^n)$
      - h_t^b corresponds to information from the current timestep to the end of the sequence
  - **Combine** the forward and backward networks

Bidirectional RNNs

- Two independent RNNs
  - One where the input is processed from start to end
  - One where the input is processed from end to start
- Outputs combined into a single representation that captures both the left and right contexts of an input at each timestep
  - $h_t = h_t^f \oplus h_t^b$
- How to combine the contexts?
  - Concatenation
  - Element-wise addition, multiplication, etc.

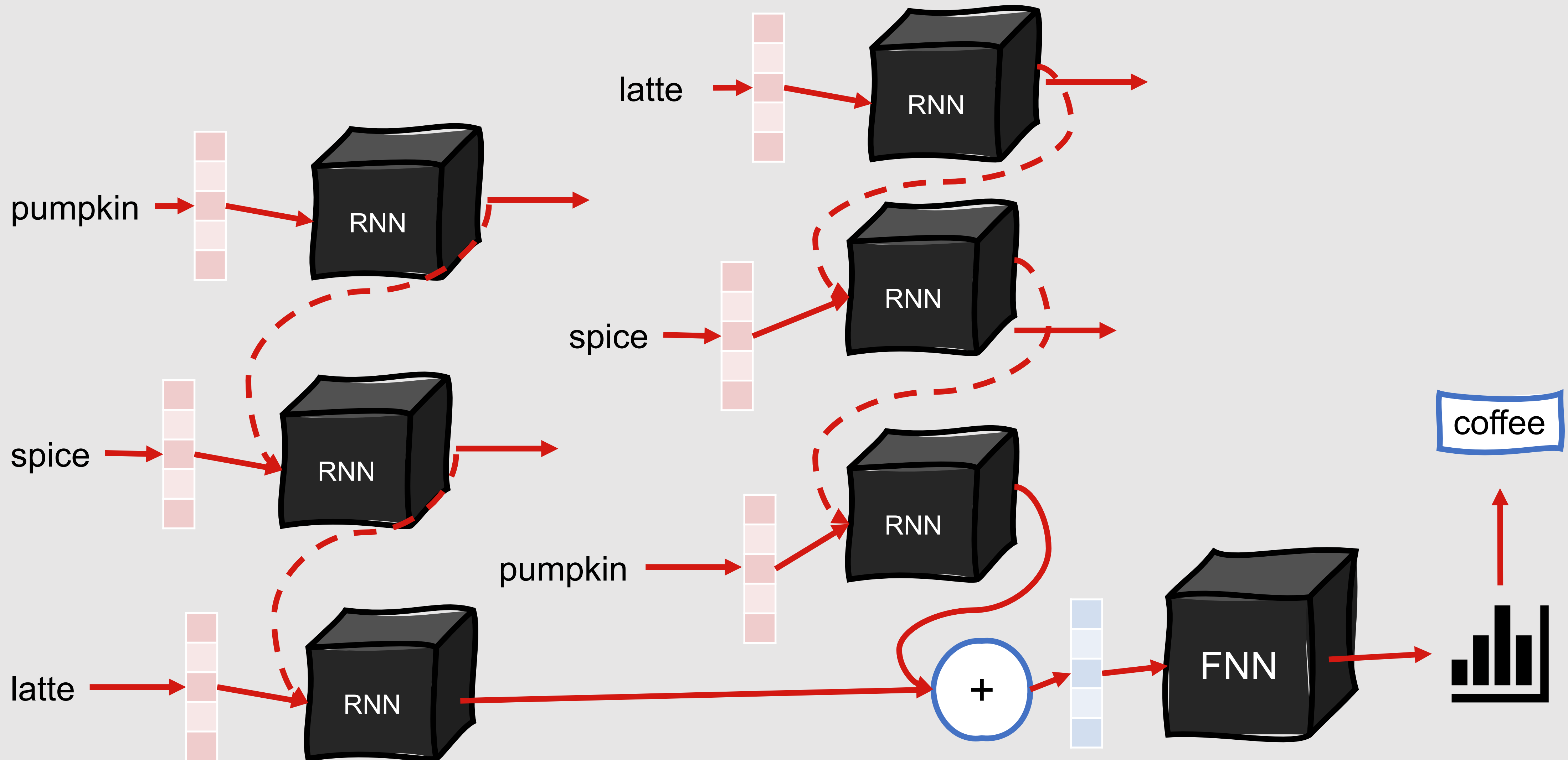
Bidirectional RNNs



Bidirectional RNNs are particularly useful for sequence classification.

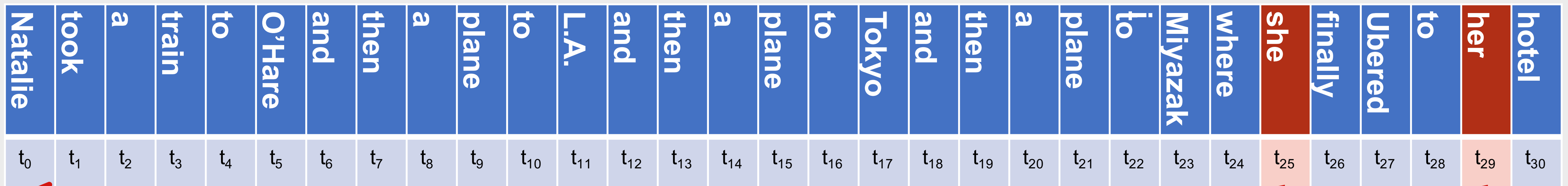
- Simple RNN → final state naturally reflects more information about the end of the sentence
- Bidirectional RNN → final state is a combination of the forward and backward passes

Sequence Classification with a Bidirectional RNN



Managing Context in RNNs

- 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
- However, long-distance information can be critical to many applications!



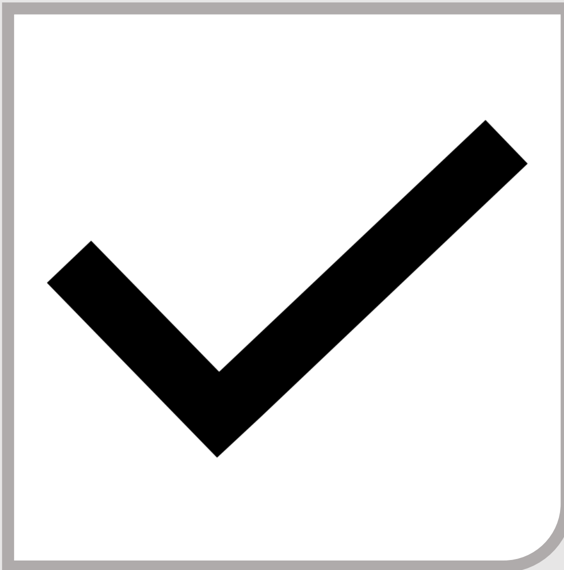
Why is it so hard to maintain long-distance context?

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)

Vanishing gradients

- When small derivatives are repeatedly multiplied together (as would happen when backpropagating through time for a long sequence), **gradients can become so close to zero at early layers that they are no longer effective for model training**

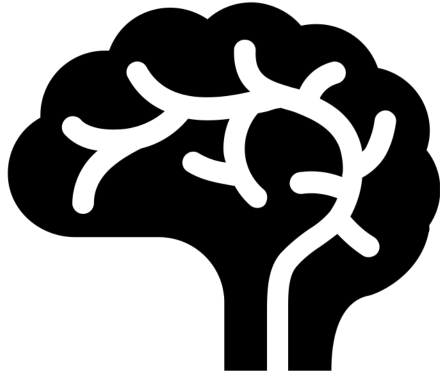


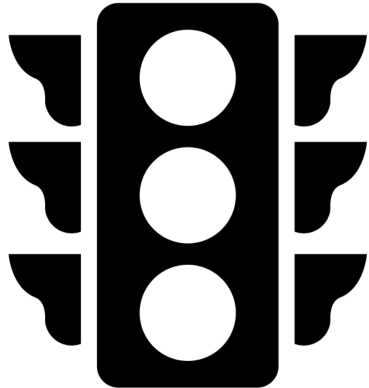
How can we address this?

- 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
  - Control 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?
- **Input gate:** Should we write this new information to the context?
- **Output gate:** What information should be revealed as output for the current hidden state?

Long Short-Term Memory Networks (LSTMs)

- LSTMs thus accept as input:
  - **Context layer**
  - **Hidden layer** from previous timestep
  - **Current input vector**
- 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

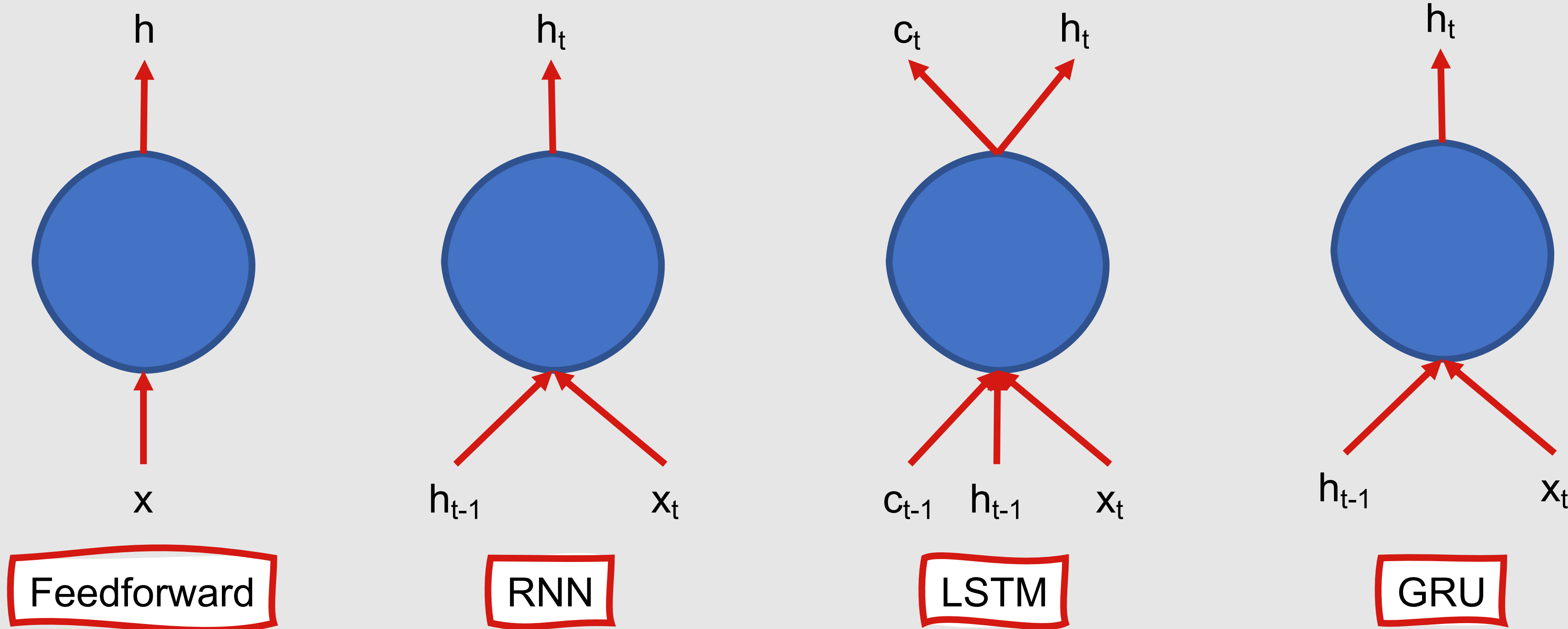
Gated Recurrent Units (GRUs)

- 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**

GRU Gates

- **Reset:** Which aspects of the previous hidden state are relevant to the current context?
  - What can be ignored?
- **Update:** Based on the intermediate representation produced by the reset gate, which aspects will be used directly in the new hidden state?
  - Which aspects of the previous state need to be preserved for future use?
- Recall that $\mathbf{h_t} = \sigma(W\mathbf{x_t} + U\mathbf{h_{t-1}} + \mathbf{b})$
- Letting $\mathbf{h_t}$' be an intermediate representation learned by the reset gate, and z be an intermediate representation produced by the update gate:
  - $\mathbf{h_t} = (1 - \mathbf{z_t})\mathbf{h_{t-1}} + \mathbf{z_t}\mathbf{h_t}'$

Comparing Inputs and Outputs for 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

Do inputs to RNNs need to be word embeddings?

- Nope
- In some cases, word embeddings might not be the best type of input:
  - Lexicon is so large that it is impractical to represent each possible word as an embedding
  - Dataset has many unknown words
  - Morphological information is critical to the task

Alternatives to Word Embeddings

Input character sequences	Input character sequences directly to the RNN
Use subword representations	Use subword representations rather than full word embeddings
Build input sequences	Build input sequences based on information from linguistic analyses

Input representations can also be combined….

- Particularly successful: Word embeddings enriched with character information
  - Learn embeddings from a bidirectional RNN that accepts character sequences for each word as input
  - Concatenate the learned character embeddings with ordinary word embeddings
- Opportunities are endless …currently a very active research area!

Summary: Sequence Processing with Recurrent Networks

- **Sequence processing** makes use of **temporal** information from input sequences
- **Recurrent neural networks (RNNs)** are neural networks specifically designed for sequence processing
  - Bonus: Can accept inputs of variable length
- RNNs base their decisions on both **current input** and **activation values from the previous timestep**
- RNNs are particularly useful for **text generation**, **sequence labeling**, and (when combined with a feedforward network) **sequence classification**
- More complex varieties of RNNs include:
  - **Stacked RNNs**
  - **Bidirectional RNNs**
  - **Long short-term memory networks (LSTMs)**
  - **Gated recurrent units (GRUs)**
- Although RNNs usually accept word embeddings as input, they can also use other types of input sequences (e.g., **character sequences** or **subword embeddings**)