Recurrent Neural Networks

Natalie Parde, Ph.D.

Department of Computer Science

University of Illinois at Chicago

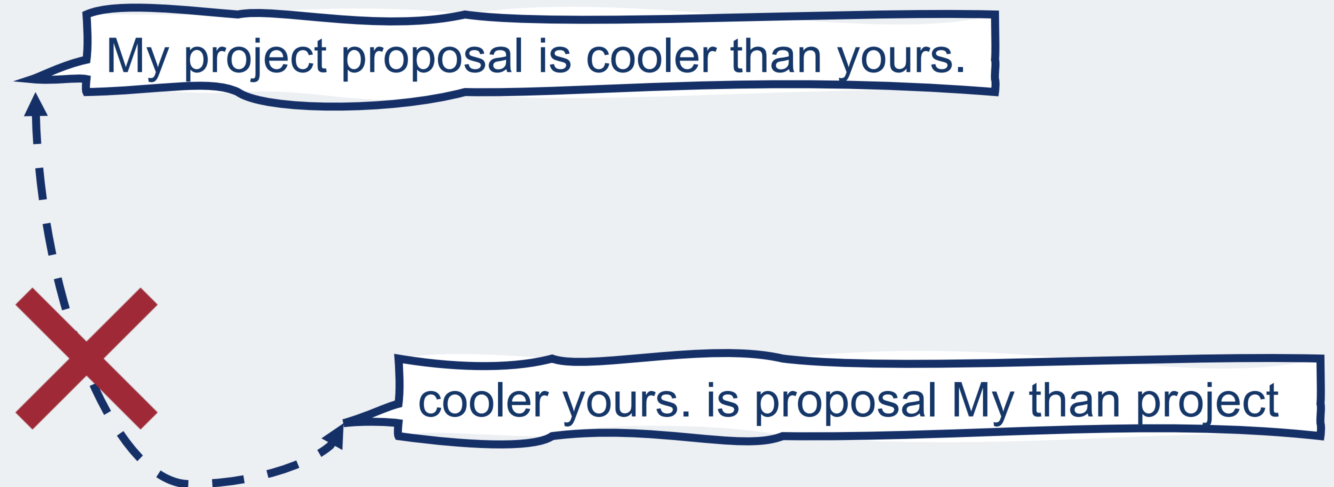
CS 521: Statistical Natural Language Processing

Spring 2020

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

What are recurrent neural networks?

- Neural networks that exploit the **temporal** nature of language!
- Also allow variable-length inputs



Language is inherently temporal.

- **Continuous input streams** of **indefinite length** that unfold over **time**
- Even clear from the metaphors we use to describe language:
  - Conversation flow
  - News feed
  - Twitter stream

To capture this phenomenon computationally, we can use recurrent neural networks to perform sequence processing.

- **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)

Sequence processing is particularly useful for some tasks!

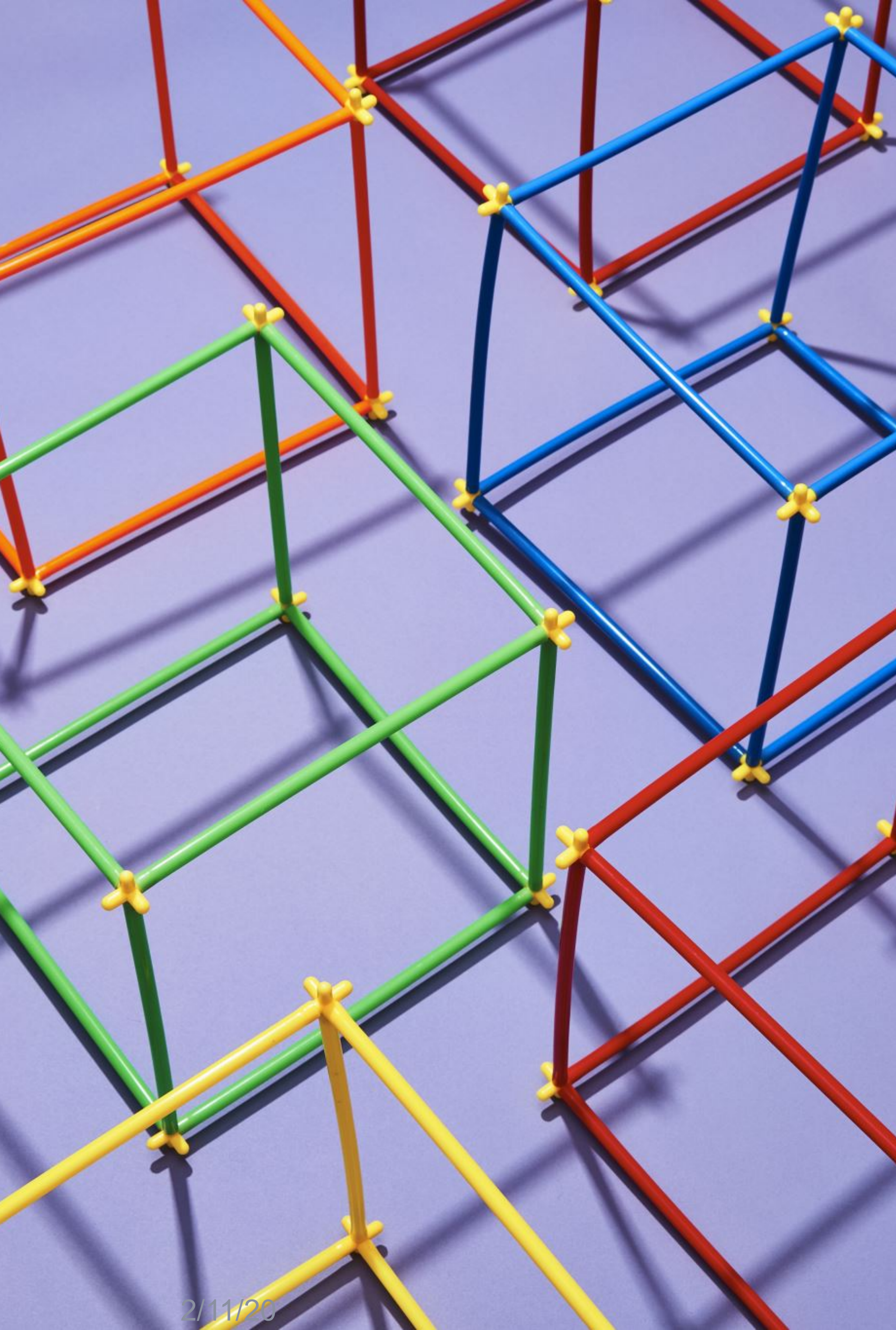
- Syntactic parsing
- Part of speech tagging
- Language modeling

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

verb? noun? adjective?

Natalie's tweet had a **like** within thirty seconds of posting it.

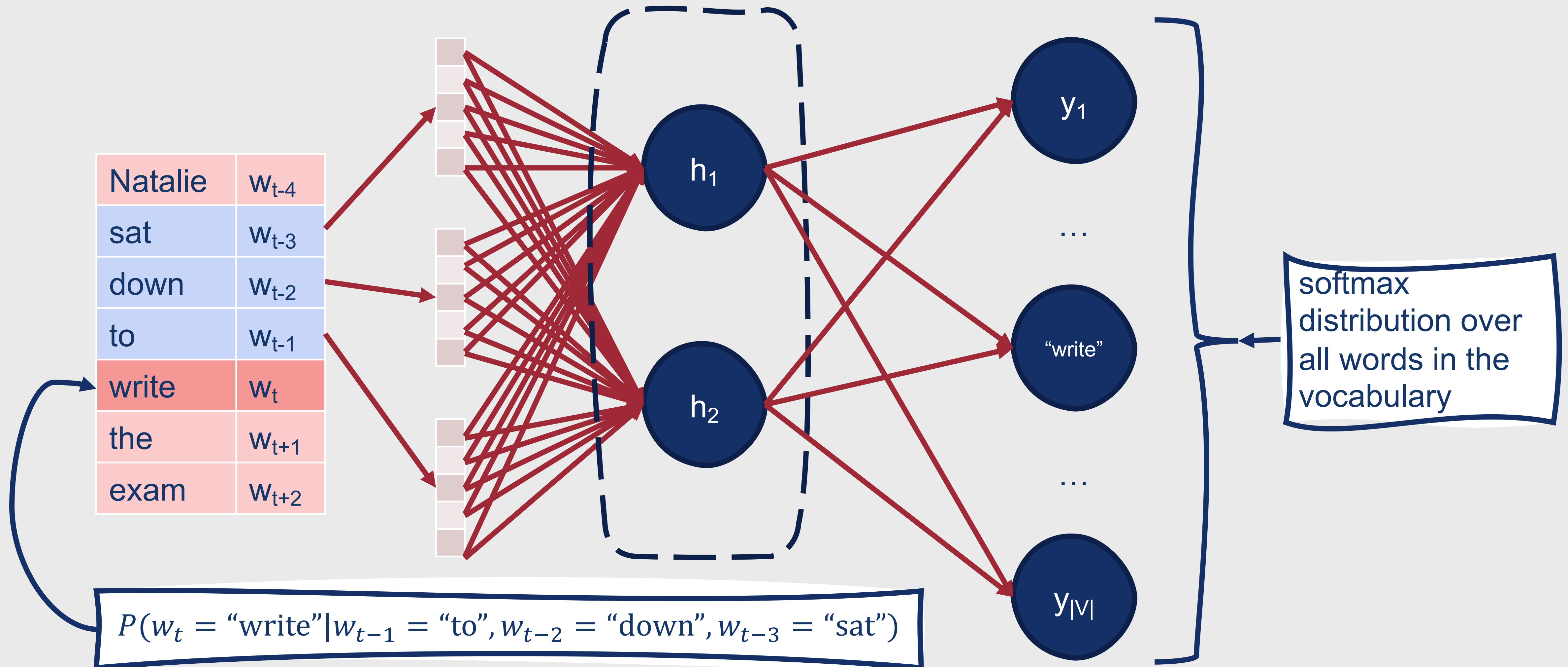
verb? noun? adjective?



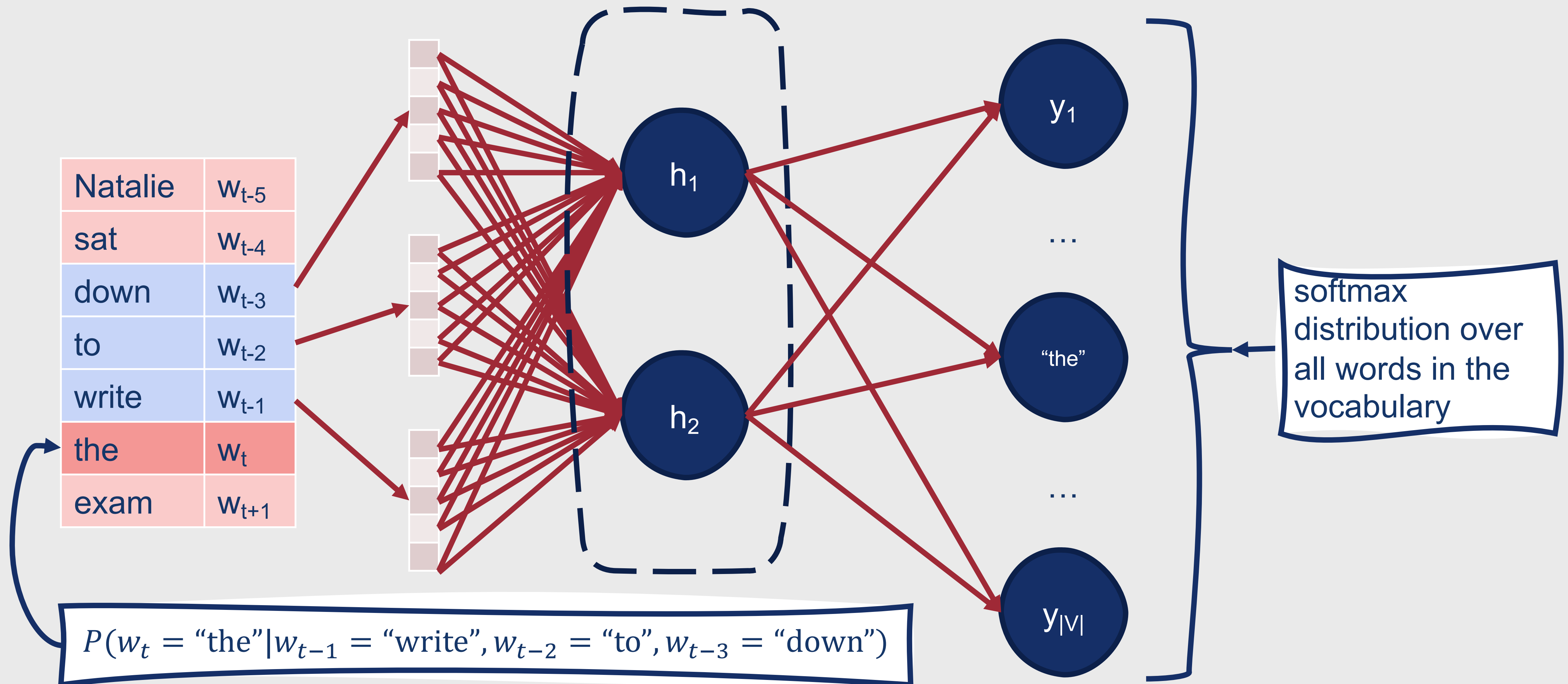
Aren't other neural network models (e.g., feedforward networks) already able to capture temporal information?

- In a sense, yes
- How?
  - **Sliding window approach**

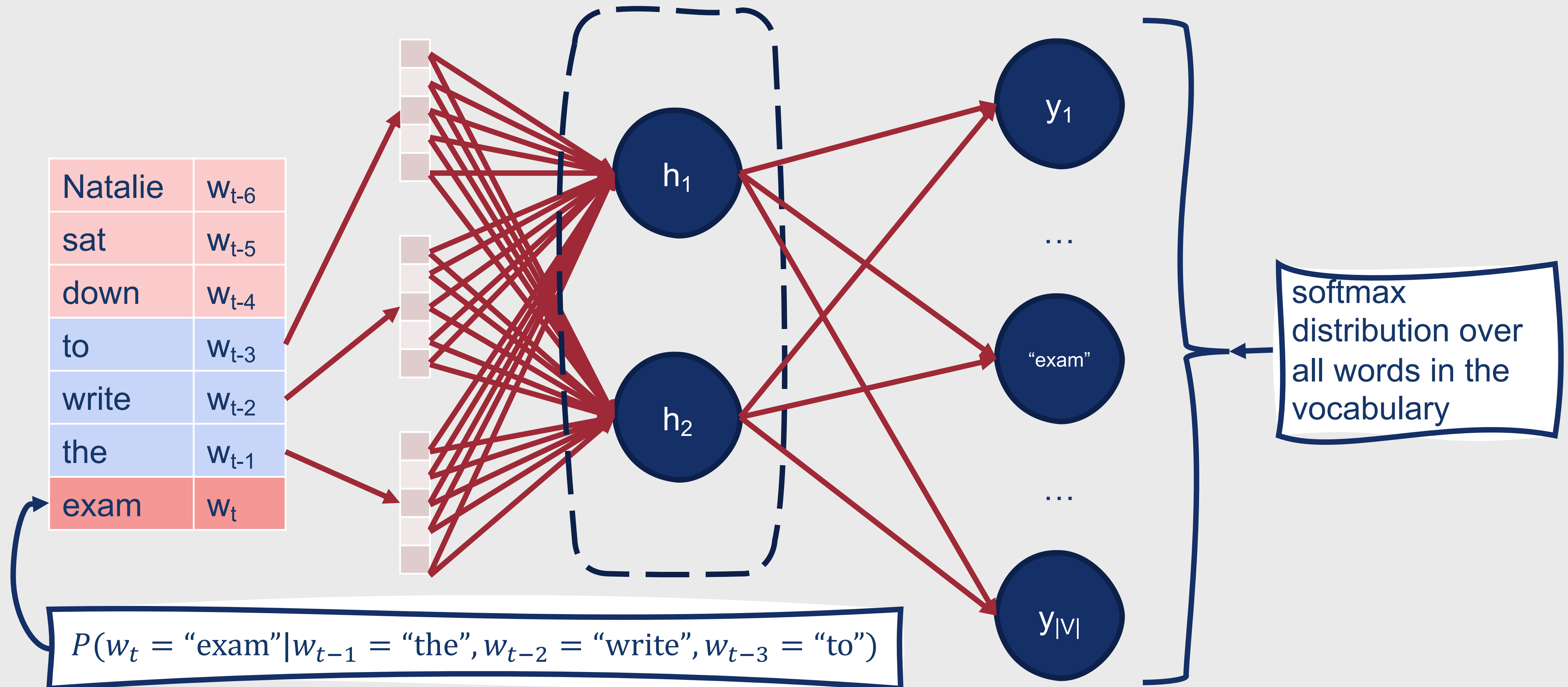
Sliding Window Approach



Sliding Window Approach

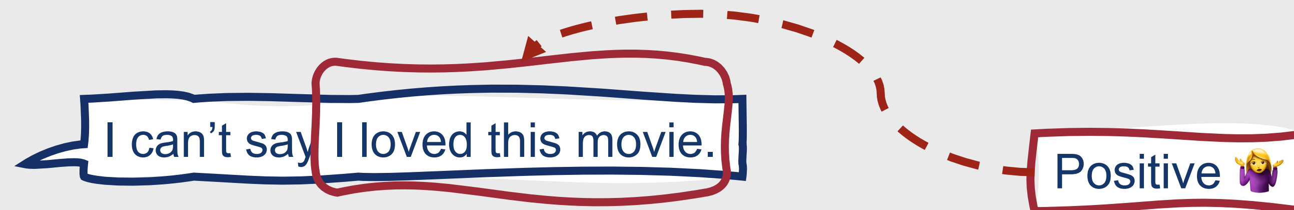


Sliding Window Approach



However, this method has some limitations.

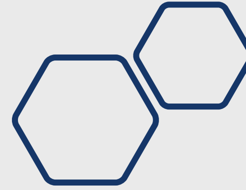
- **Constrains the context** from which information can be extracted
  - Only items within the predetermined context window can impact the model's decision
- Makes it difficult to learn **systematic patterns**
  - Particularly problematic when learning grammatical information (e.g., constituent parses)



However, this method has some limitations.

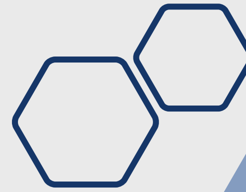
- **Constrains the context** from which information can be extracted
  - Only items within the predetermined context window can impact the model's decision
- Makes it difficult to learn **systematic patterns**
  - Particularly problematic when learning grammatical information (e.g., constituent parses)

Recurrent neural networks (RNNs) are designed to overcome these limitations.



- Built-in capacity to handle temporal information
- Can accept variable length inputs without the use of fixed-size windows

Recurrent Neural Networks

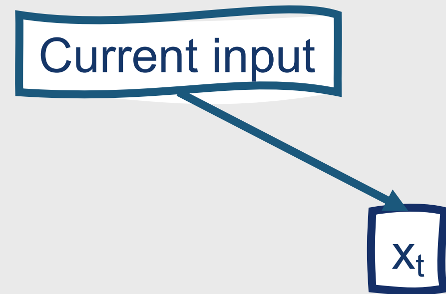


- Contain cycles within their connections
  - The value of a unit is dependent upon outputs from previous timesteps
- Many varieties exist
  - "Vanilla" RNNs
  - Long short-term memory networks (LSTMs)
  - Gated recurrent units (GRUs)

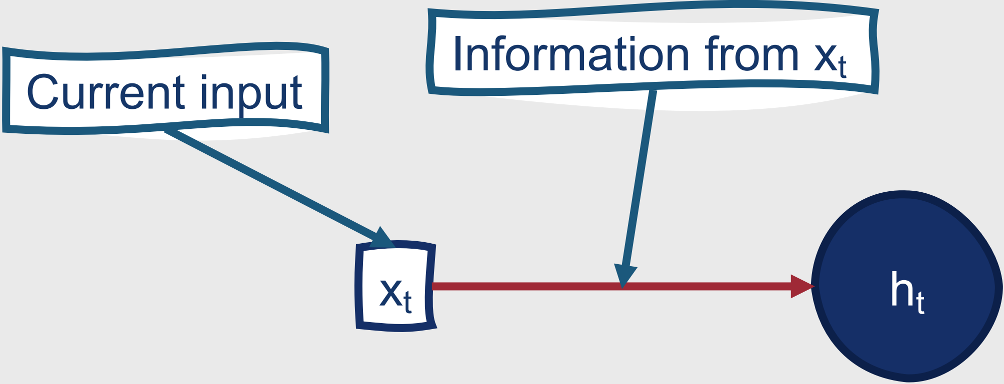
How do RNNs differ from standard feedforward neural networks?

- 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
- New hidden states are determined as a function of the existing hidden state and the new input at the current timestep

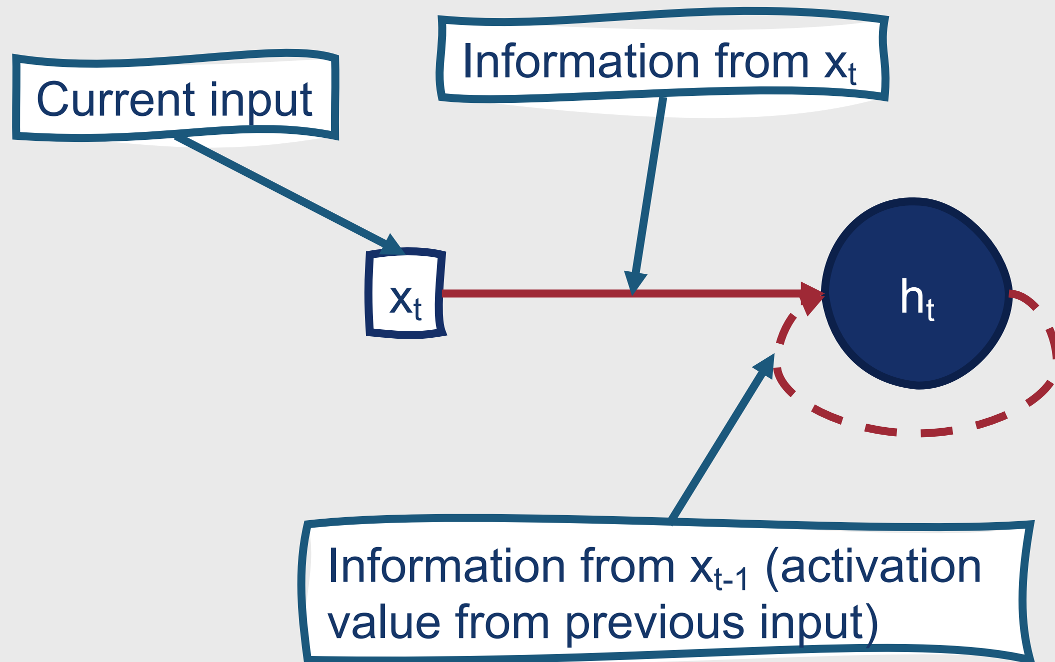
Vanilla RNN Layer



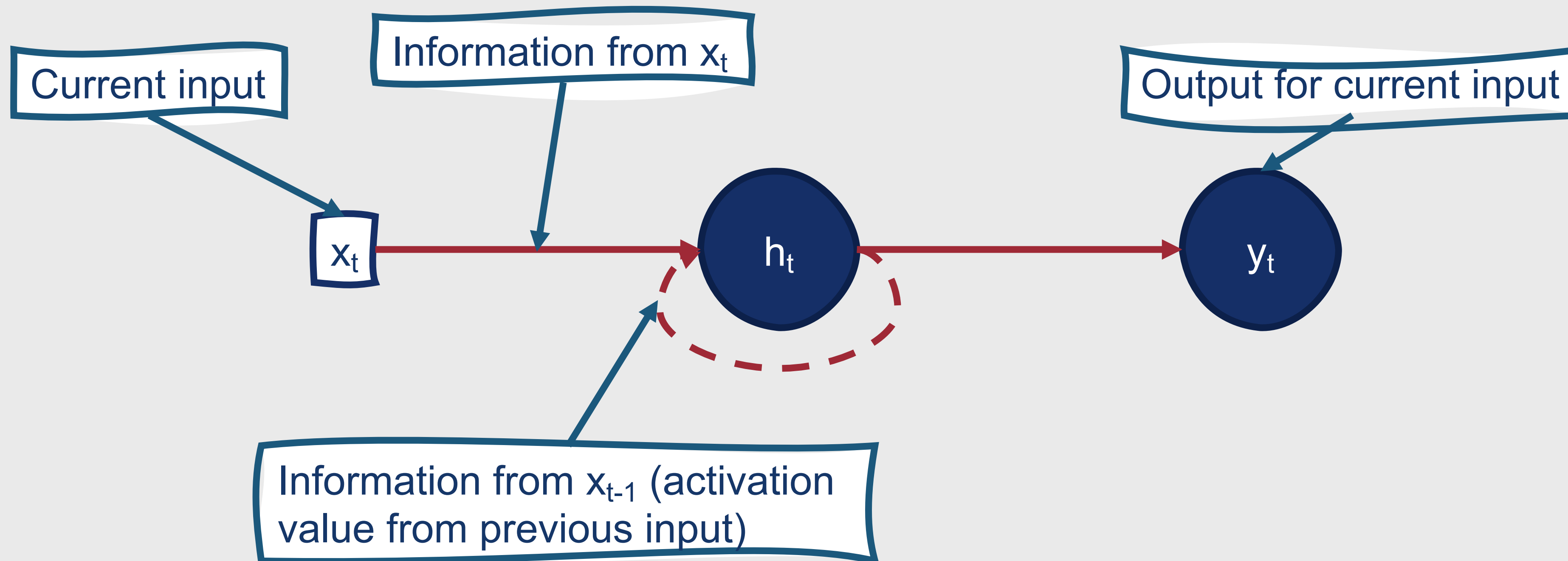
Vanilla RNN Layer



Vanilla RNN Layer



Vanilla RNN Layer



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

Outputs from earlier timesteps serve as additional context

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

Can include information extending all the way 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 from the previous timestep

Compute:

- Weighted sum of inputs

Most Significant Change

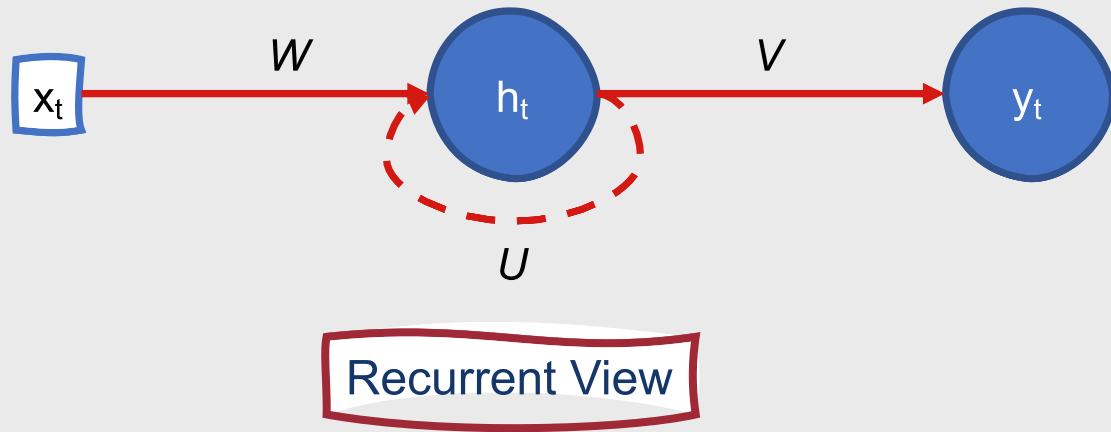
- New set of weights, *U*, that connect the hidden layer from the previous timestep to the current hidden layer
- These weights determine how the network should make use of prior context

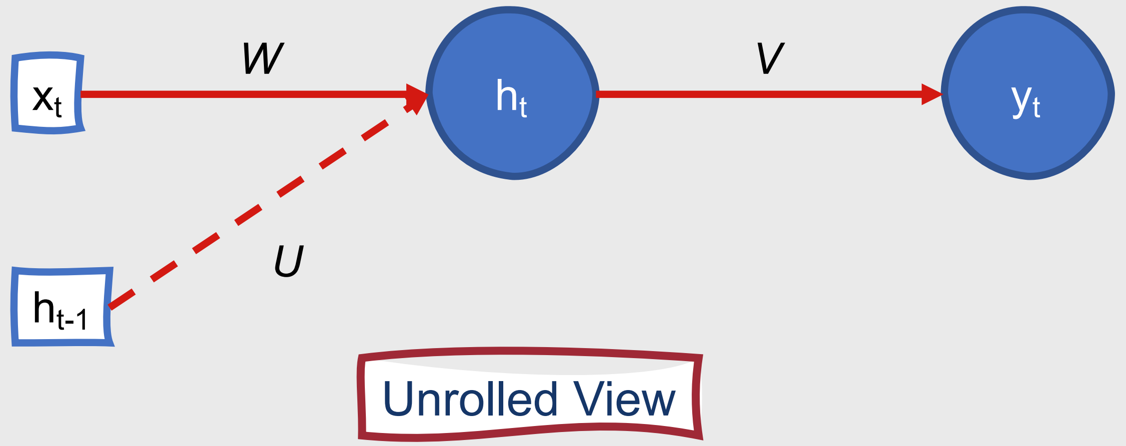
Formal Equations

- 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 Equations

- Just add (weights X activation values from previous timestep) product to the current (weights X inputs) product
  - $\mathbf{h} = \sigma(W\mathbf{x}_\mathrm{t} + U\mathbf{h}_{\mathrm{t}-1} + \mathbf{b})$
  - $\mathbf{z} = V\mathbf{h}_\mathrm{t}$
  - $y = \mathrm{softmax}(\mathbf{z})$
- *W*, *U*, and *V* are shared across all timesteps





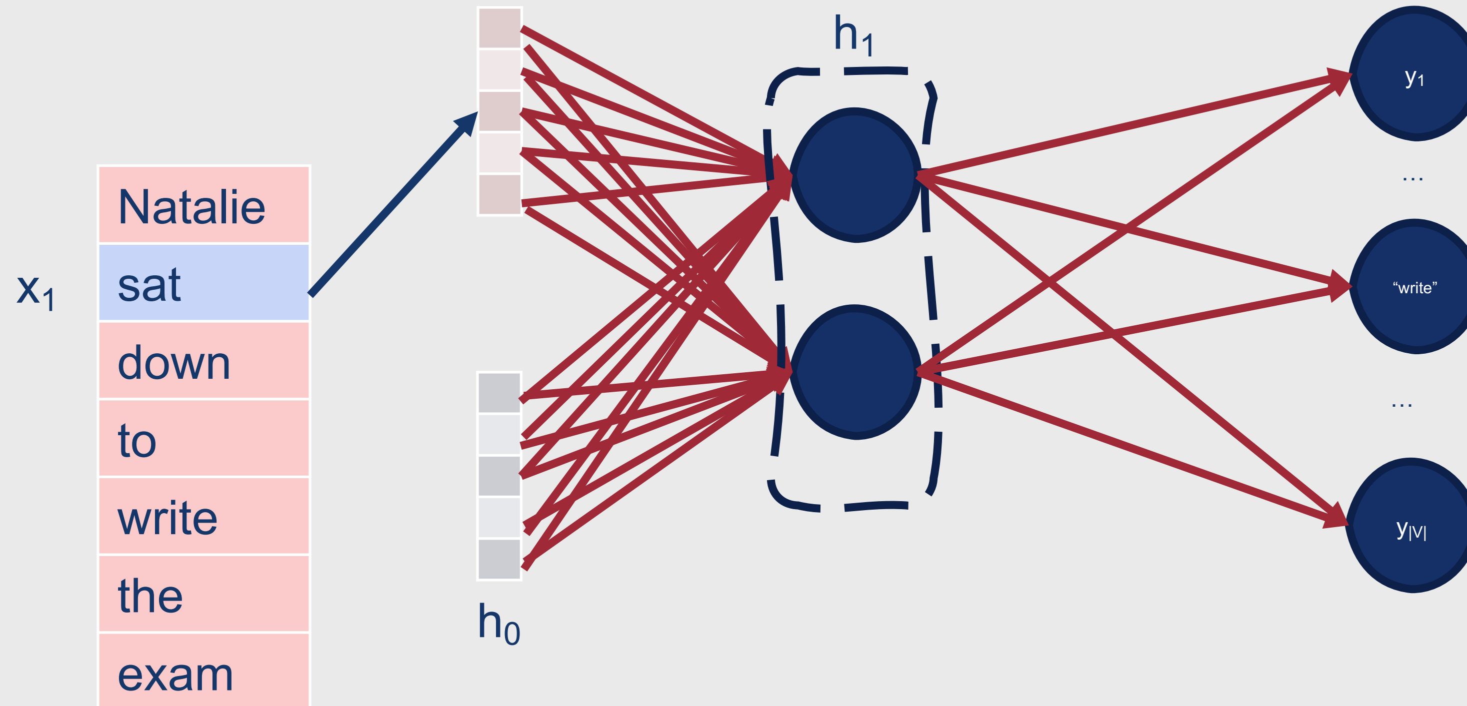
What does this look like when unrolled?

Formal Algorithm

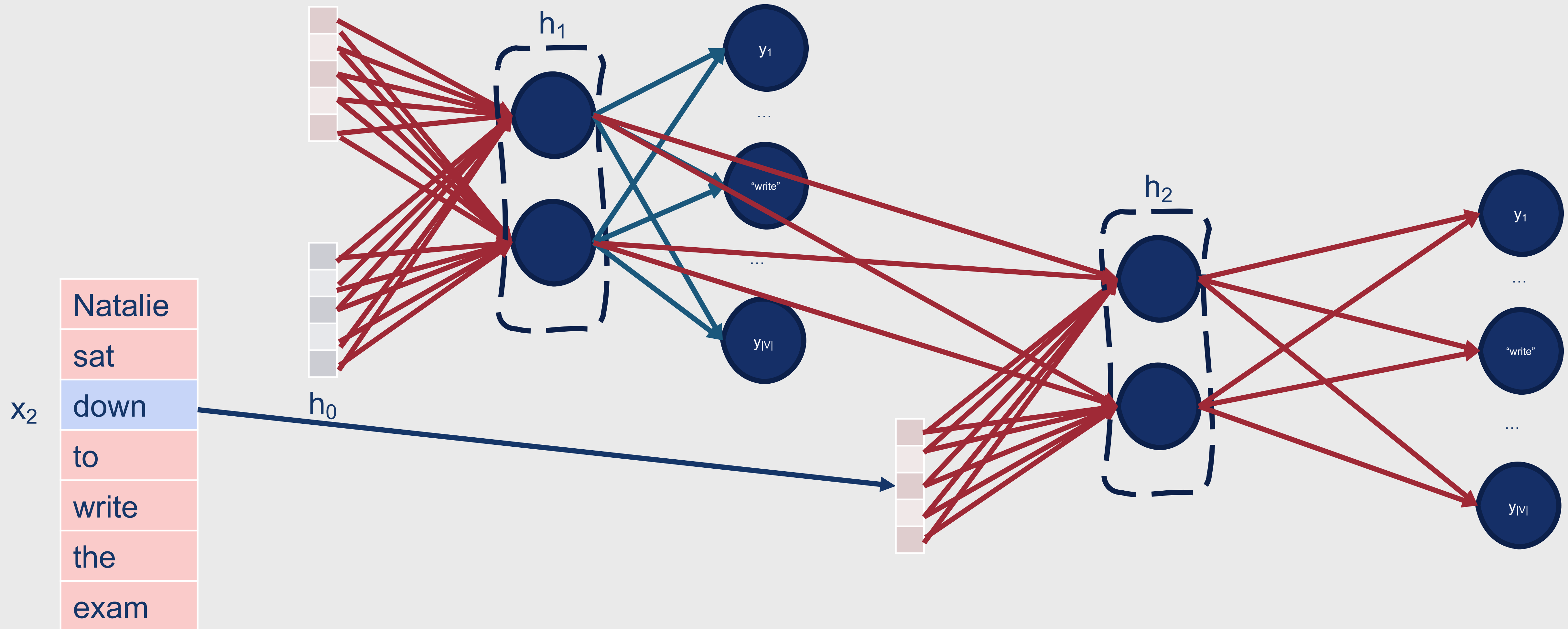
```
h_0 ← 0  # Initialize activations from the hidden layer to 0

# Iterate through each input element in temporal order
for i ← 1 to length(x) do:
      h_i ← g(U h_{i-1} + W x_i + b)   # Bias vector is optional
      y_i ← f(V h_i)
```

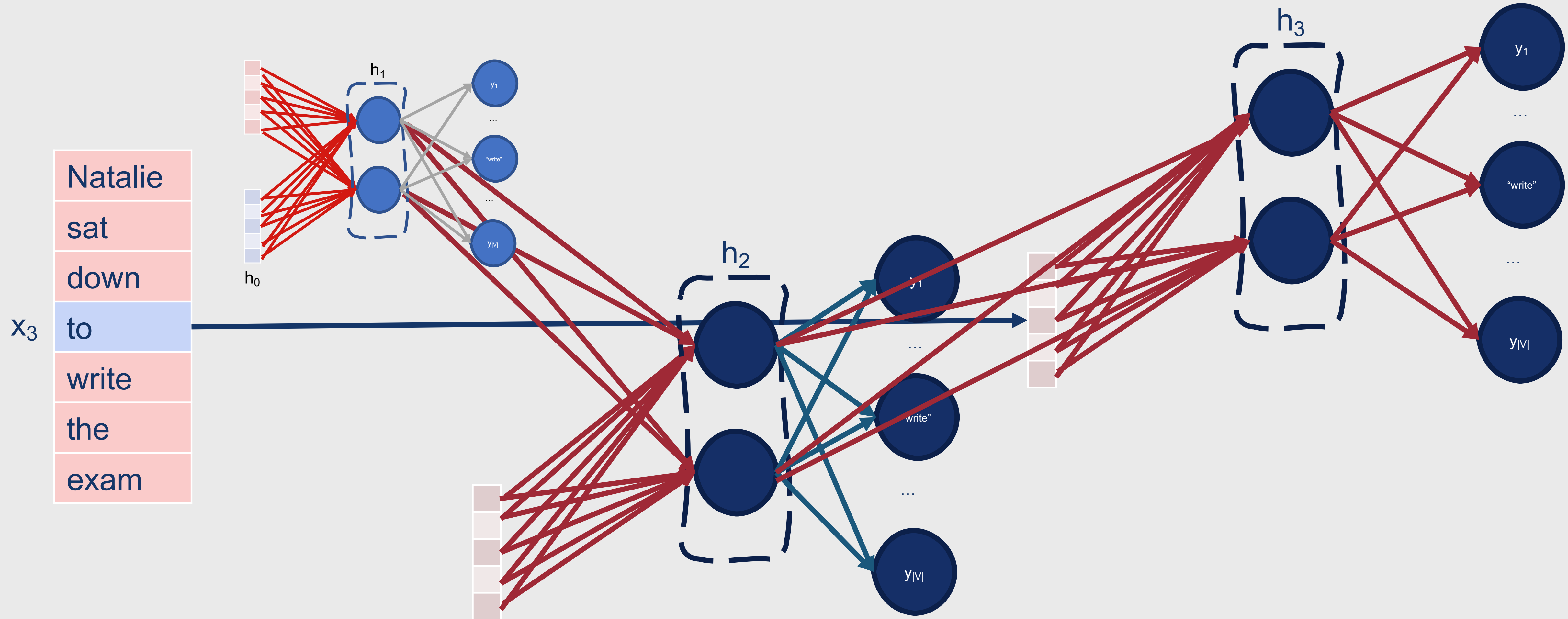
Earlier Example: RNN Edition



Earlier Example: RNN Edition



Earlier Example: RNN Edition



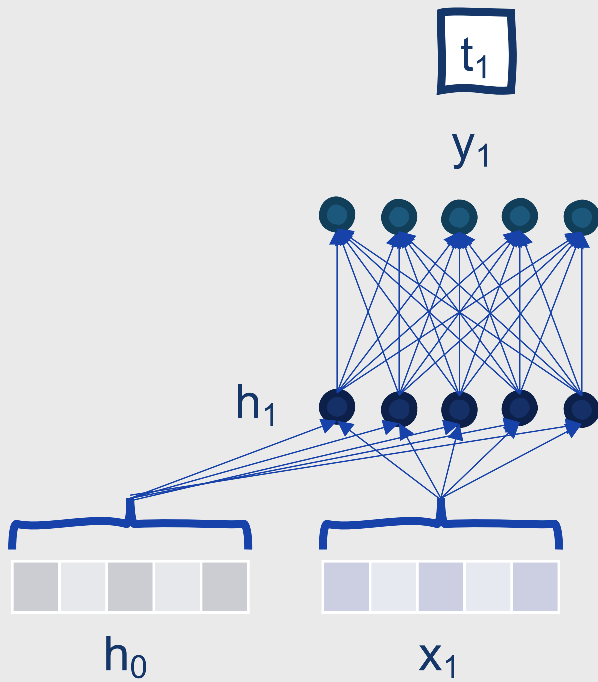
Training RNNs

- Same core elements:
  - Loss function
  - Optimization function
  - Backpropagation
- One extra set of weights to update
  - Hidden layer from *t-1* to current hidden layer at *t*

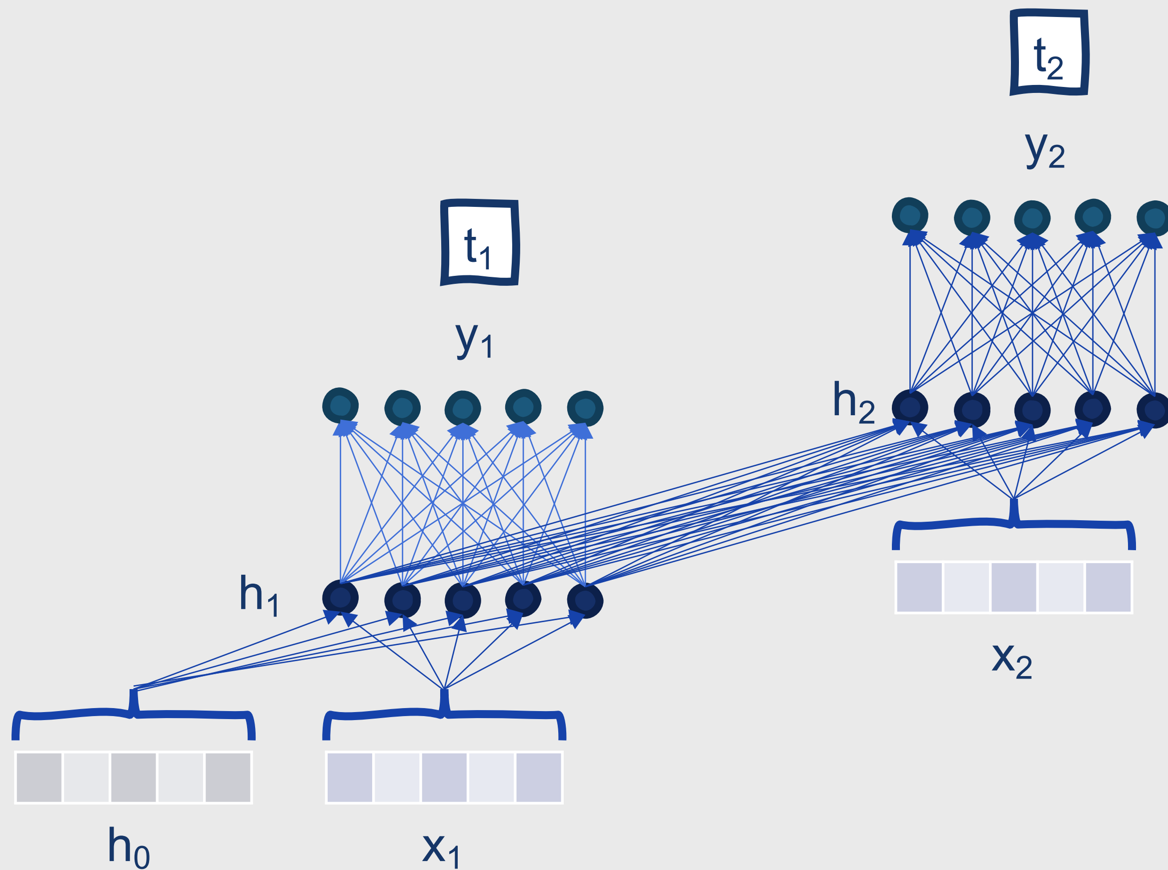
Forward Inference

- Compute h_t and y_t at **each step in time**
- Compute the loss at **each step in time**

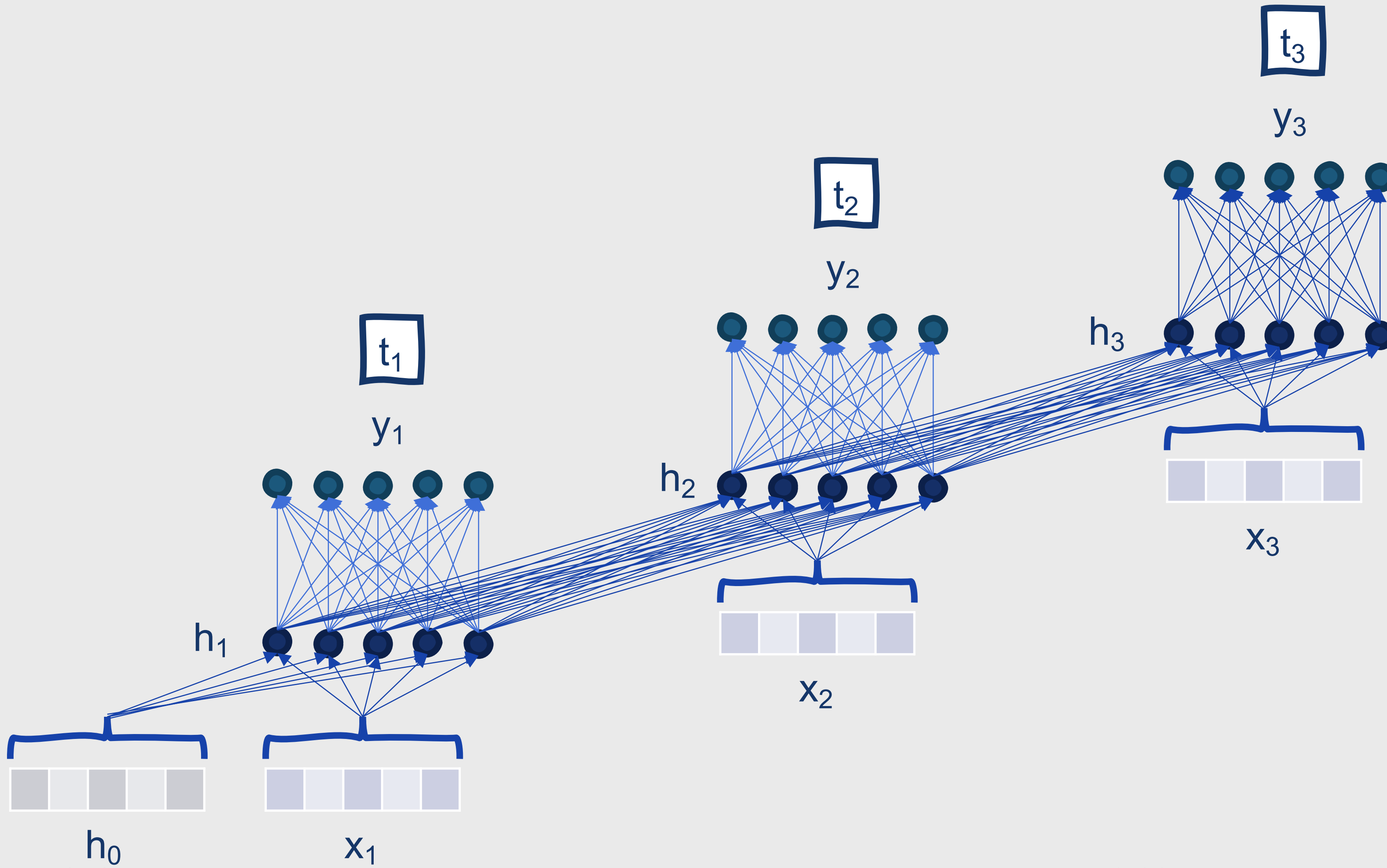
Forward Pass



Forward Pass



Forward Pass

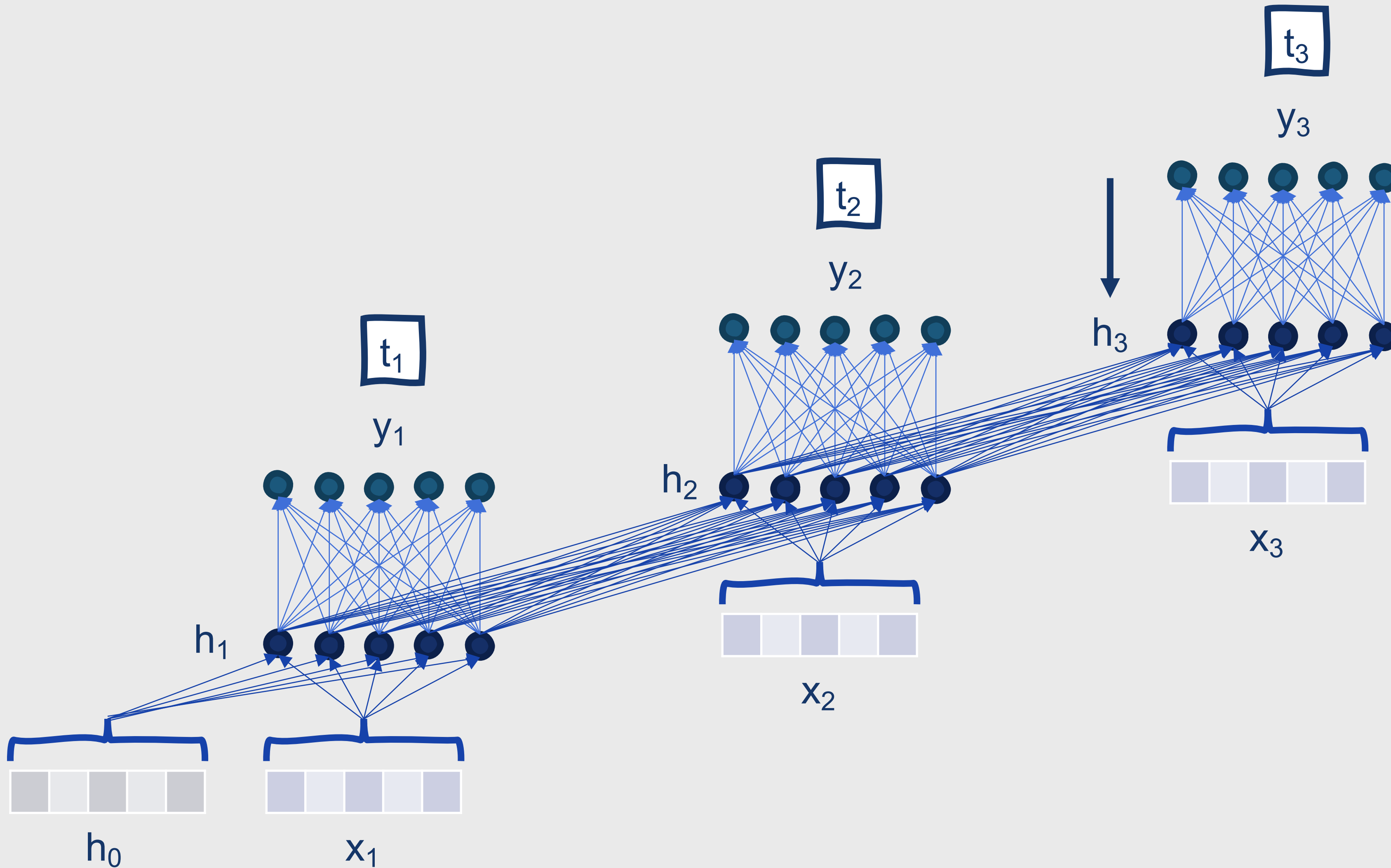


Backpropagation Through Time

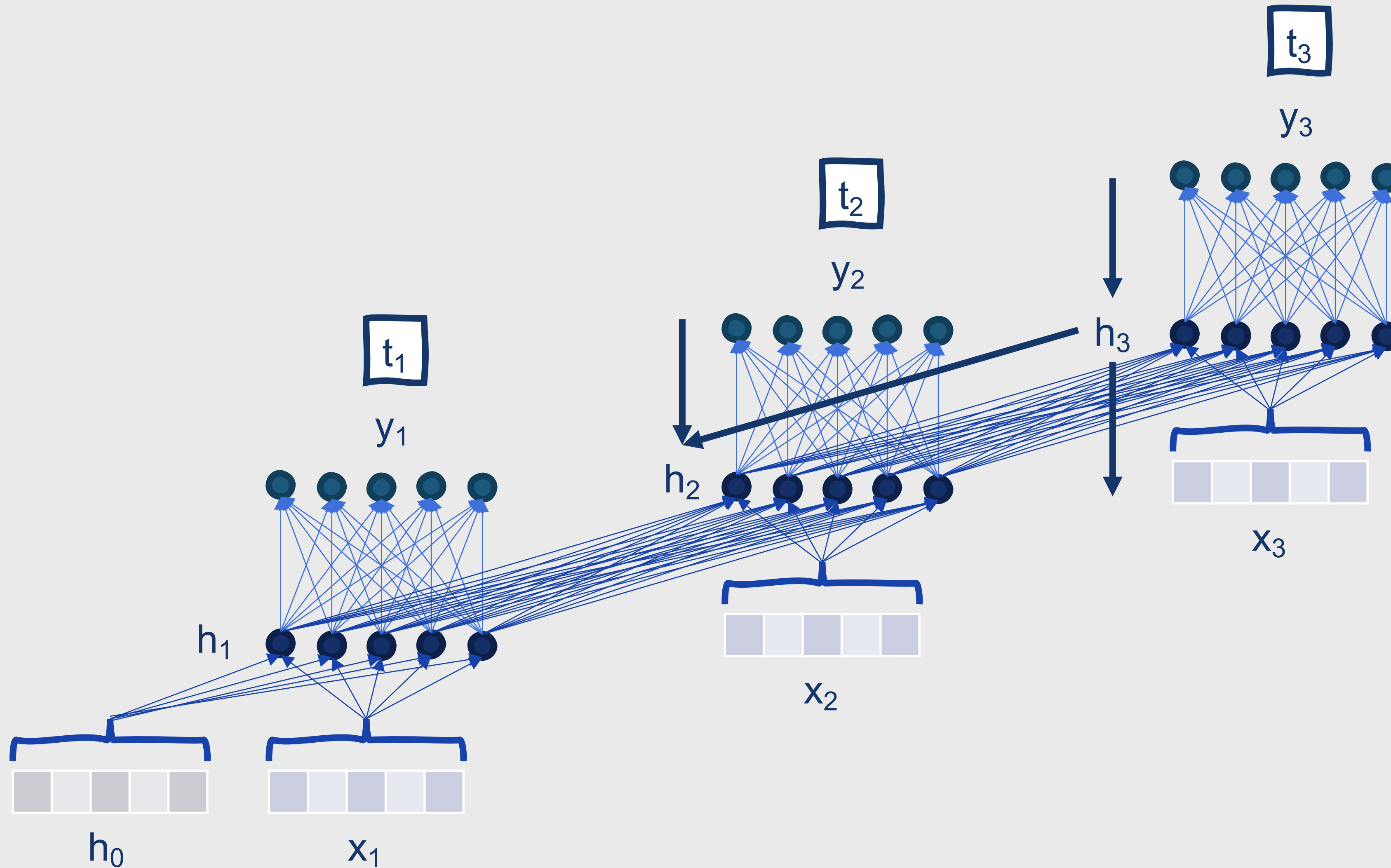
- Process the sequence in reverse
- Compute the required error gradients at **each step backward in time**

Updated from feedforward networks!

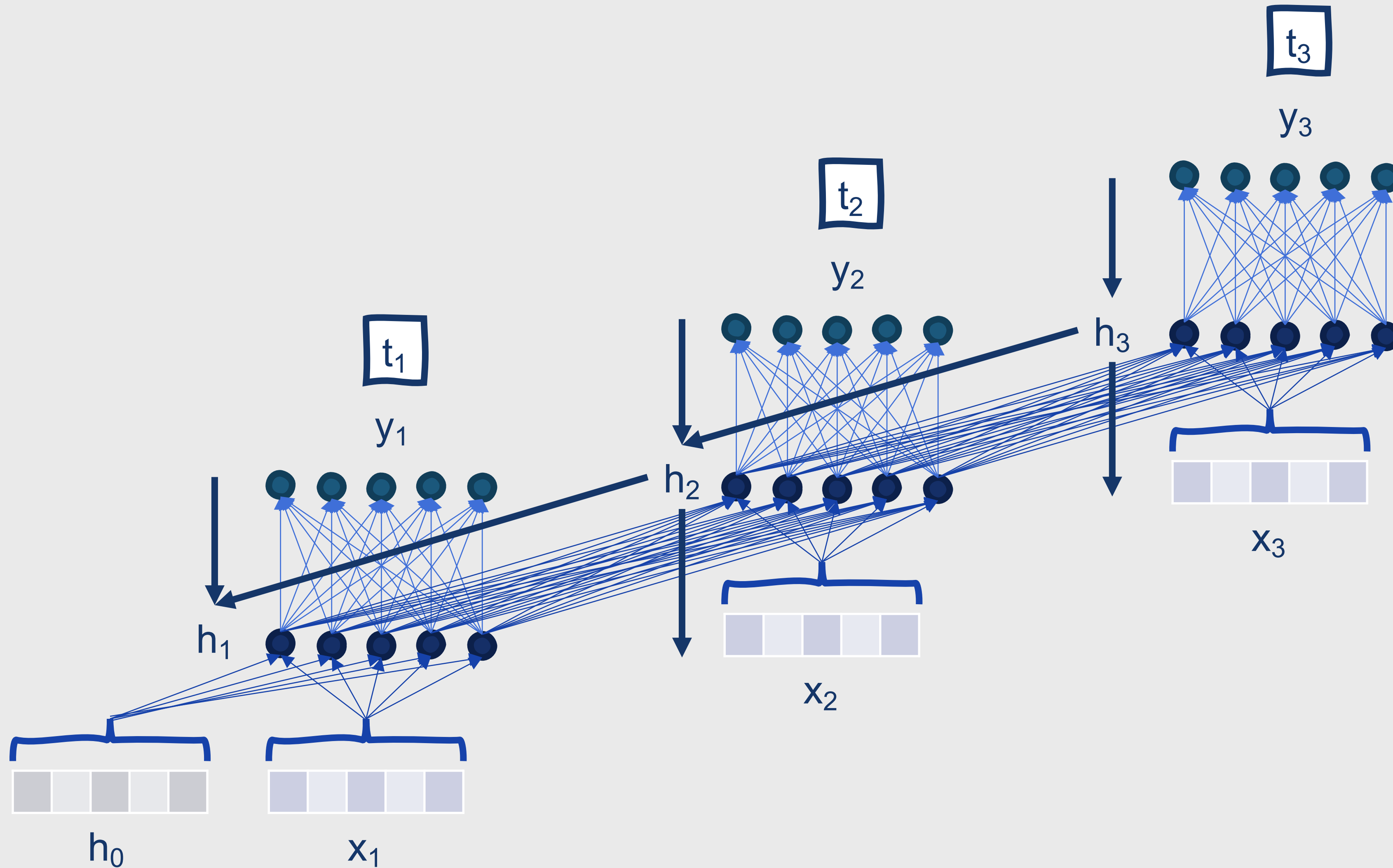
Backward Pass



Backward Pass



Backward Pass



Updated Backpropagation Equations

- Now we have three sets of weights we need to update:
  - *W*, the weights from the input layer to the hidden layer
  - *U*, the weights from the previous hidden layer to the current hidden layer
  - *V*, the weights from the hidden layer to the output layer

Updated Backpropagation Equations

- Now we have three sets of weights we need to update:
  - *W*, the weights from the input layer to the hidden layer
  - *U*, the weights from the previous hidden layer to the current hidden layer
  - *V*, the weights from the hidden layer to the output layer

t_3 y_3 t_2 y_2 h_3 t_1 y_1 h_2 x_3 h_1 x_2 h_0 x_1 U

Updated Backpropagation Equations

- Now we have three sets of weights we need to update:
  - *W*, the weights from the input layer to the hidden layer
  - *U*, the weights from the previous hidden layer to the current hidden layer
  - *V*, the weights from the hidden layer to the output layer

t_3

y_3

t_2

y_2

h_3

t_1

y_1

h_2

x_3

h_1

x_2

h_0

x_1

V

Updating the weights for *V* works no differently from feedforward networks.

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial V}$$

Updating the weights for *V* works no differently from feedforward networks.

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial V}$$

Reduce the first two terms to an error term, δ_t

Activation value of the hidden layer at the current timestep, h_t

Updating the weights for *V* works no differently from feedforward networks.

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial V}$$

$$\frac{\partial L}{\partial V} = \delta_t h_t$$

Updating the weights for *W* and *U* works a little bit differently.

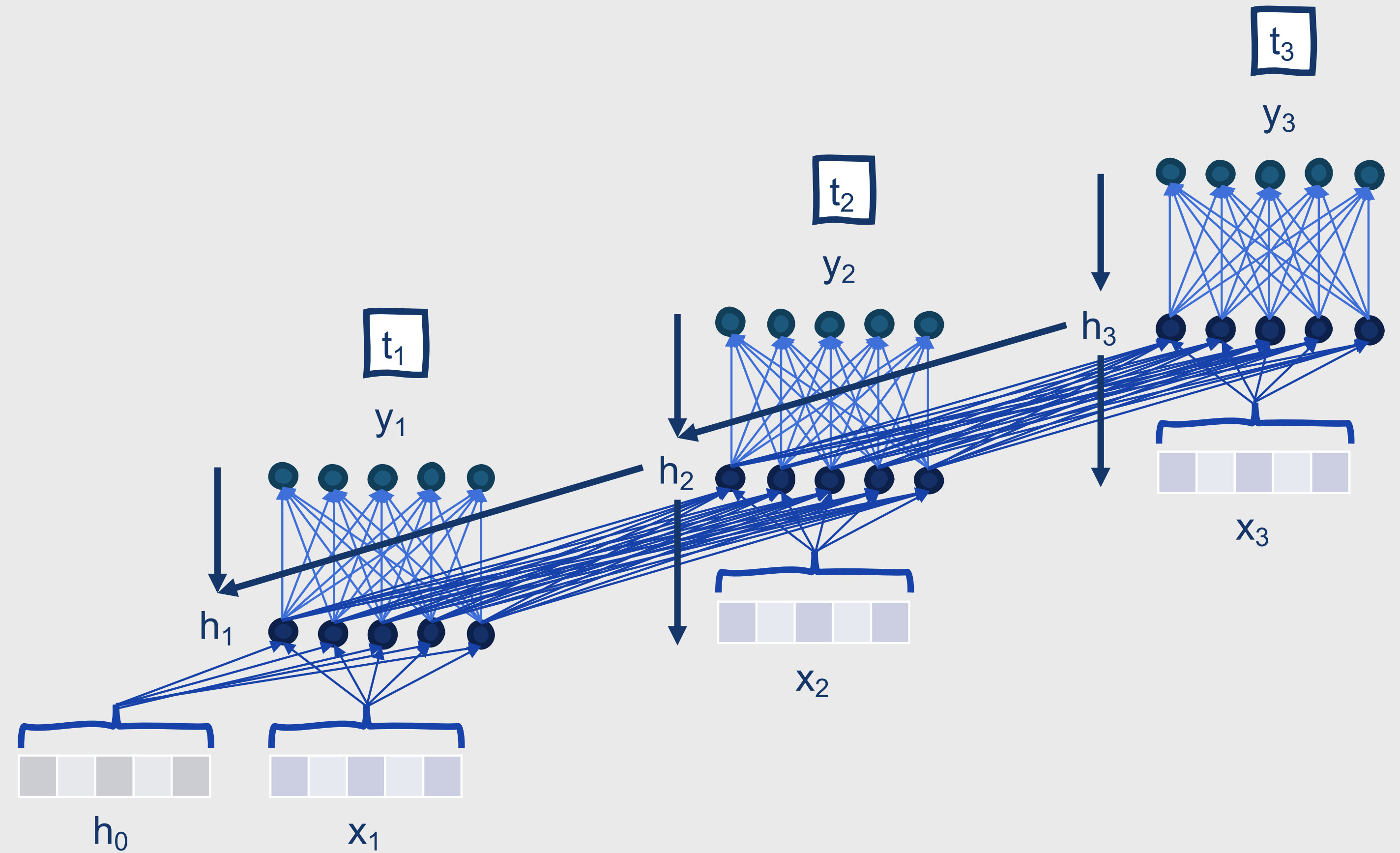
- Error term for a hidden layer, δ_h, must be the sum of the error term from the current output and the error term from the next timestep
  - $\delta_h = g'(z)V\delta_t + \delta_{t+1}$

Once we have this updated error term for the hidden layer, we can proceed as usual to compute the gradients for *U* and *W*.

- $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial a}\frac{\partial a}{\partial W} = \delta_h x_t$
- $\frac{\partial L}{\partial U} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial a}\frac{\partial a}{\partial U} = \delta_h h_{t-1}$

One remaining step….

- Backpropagate the error from δ_h to h_{t-1} based on the weights in U
  - $\delta_{t+1} = g'(z)U\delta_h$
- At this point, we have all of the necessary gradients to update U, V, and W!



Applications of Recurrent Neural Networks

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

At this point, we've seen a few types of language models.

- N-gram language models
- Feedforward neural network language models
  - In our earlier example today, a sliding window variation of this

These models attempt to predict the next word in a sequence given a prior context of fixed length.

- What's challenging about this approach?
  - 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

- 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

- 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

- 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

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.

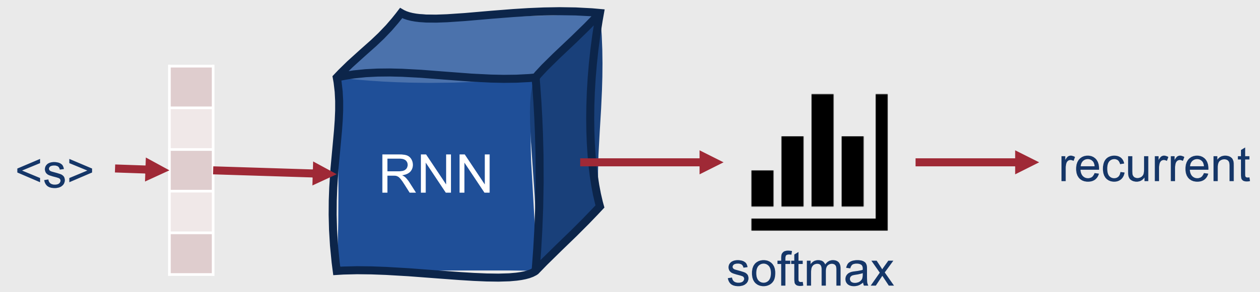
Generation with Neural Language Models

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

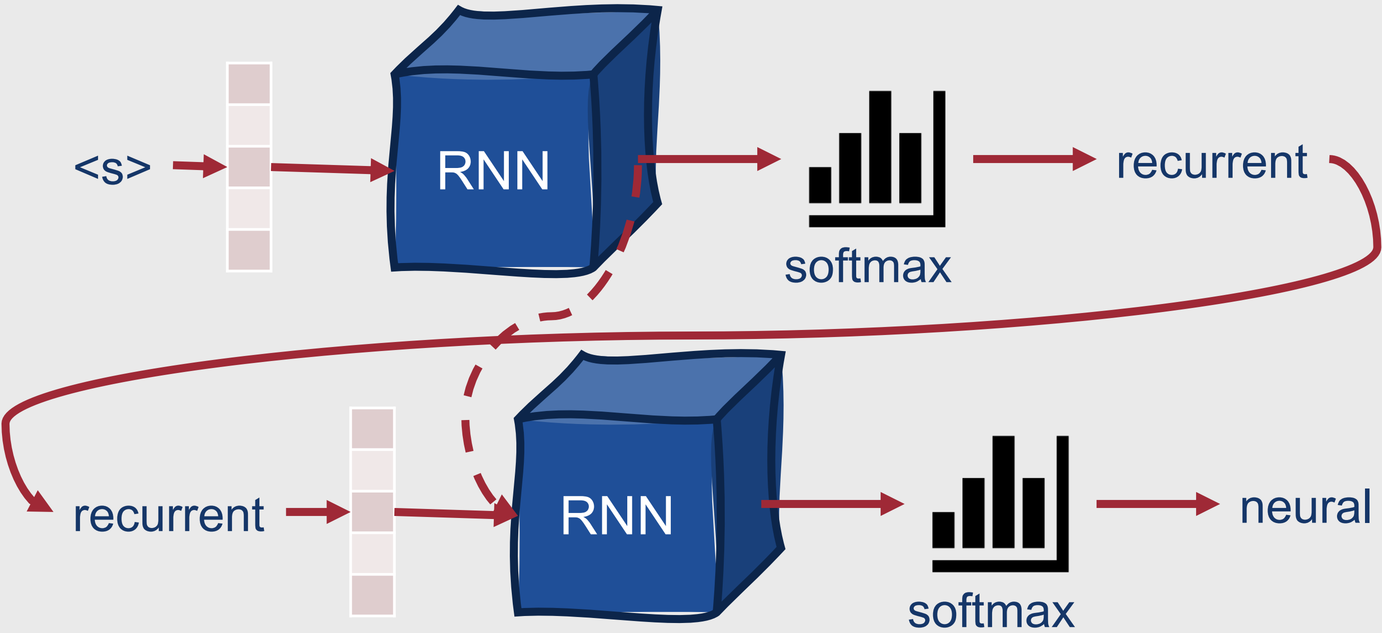
Autoregressive Generation

- This technique is referred to as **autoregressive generation**
  - Word generated at each timestep is conditioned on the word generated previously by the model

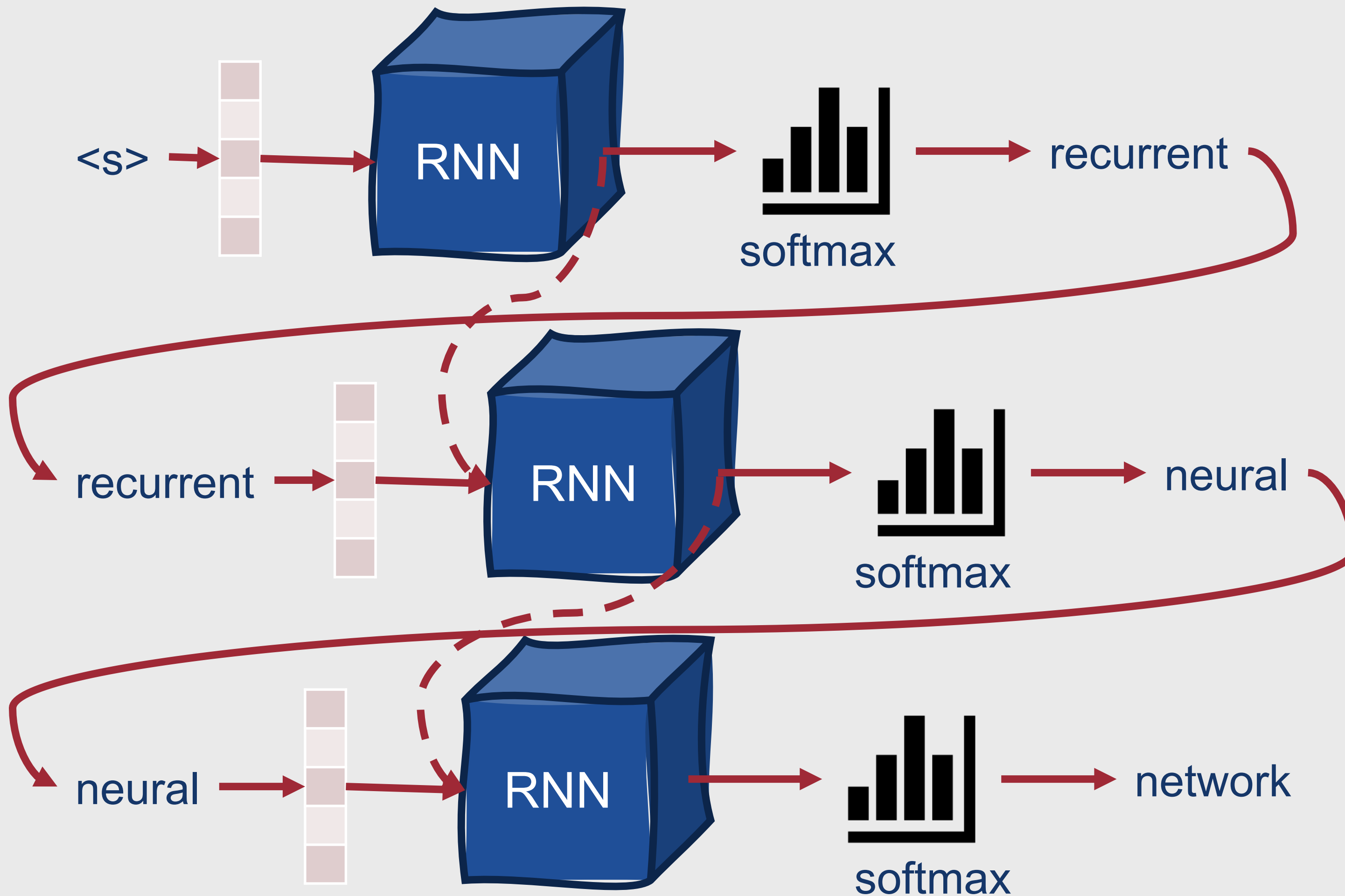
Autoregressive Generation



Autoregressive Generation



Autoregressive Generation



Autoregressive Generation

<s> RNN softmax recurrent

recurrent RNN softmax neural

neural RNN softmax network

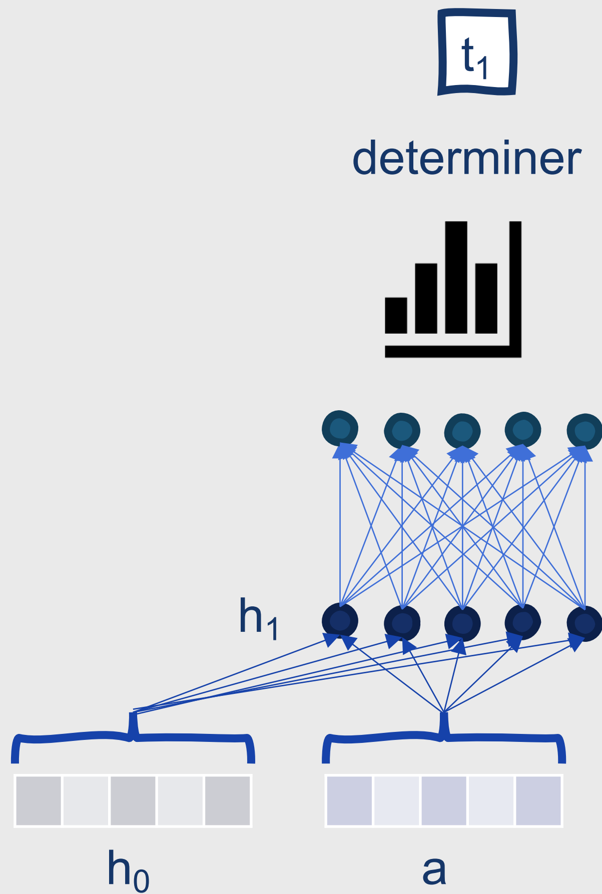
Key to successful autoregressive generation?

Prime the generation component with **appropriate context** (e.g., something more useful than <s>)

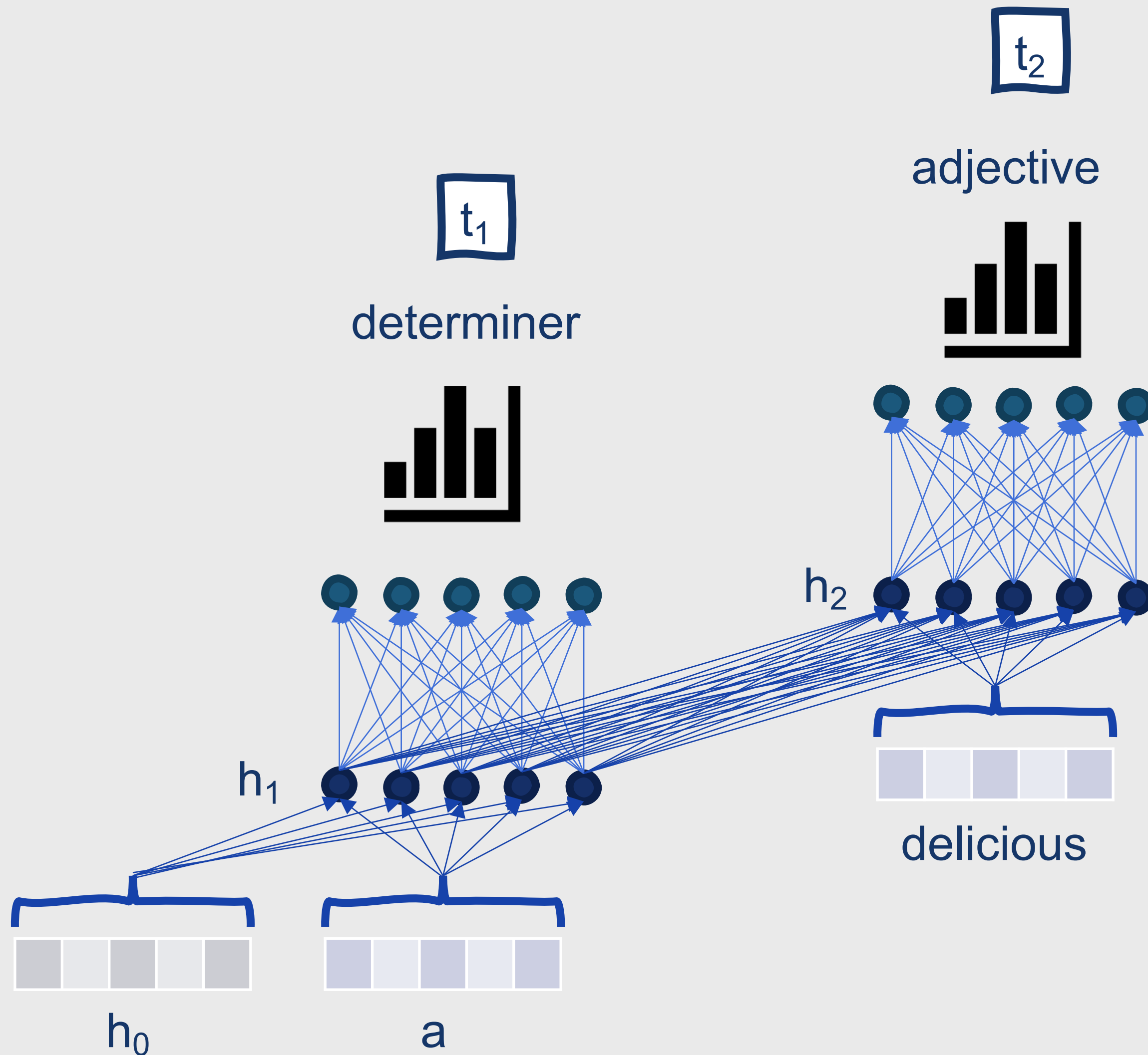
RNNs are also highly useful for sequence labeling.

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

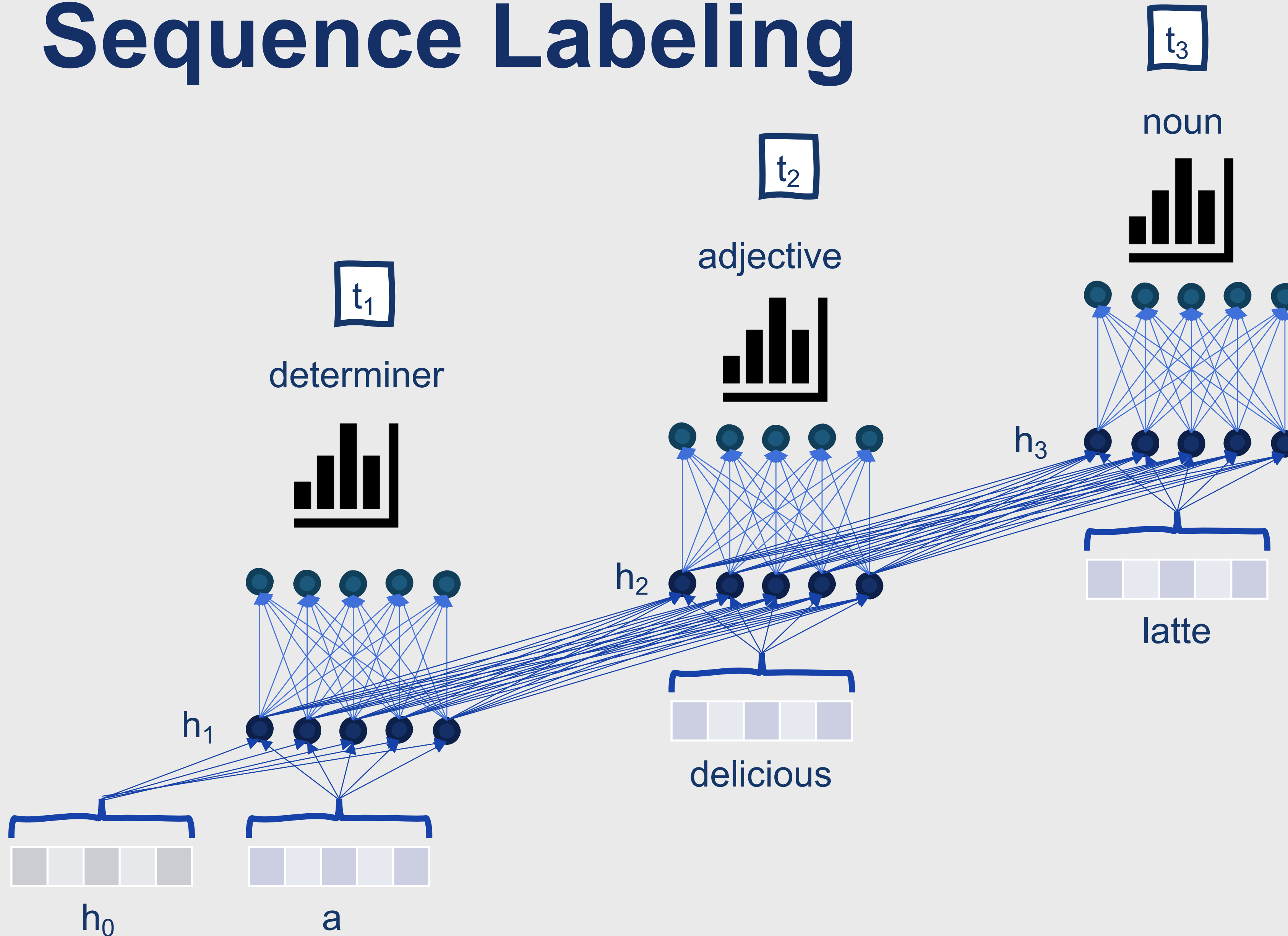
Sequence Labeling



Sequence Labeling



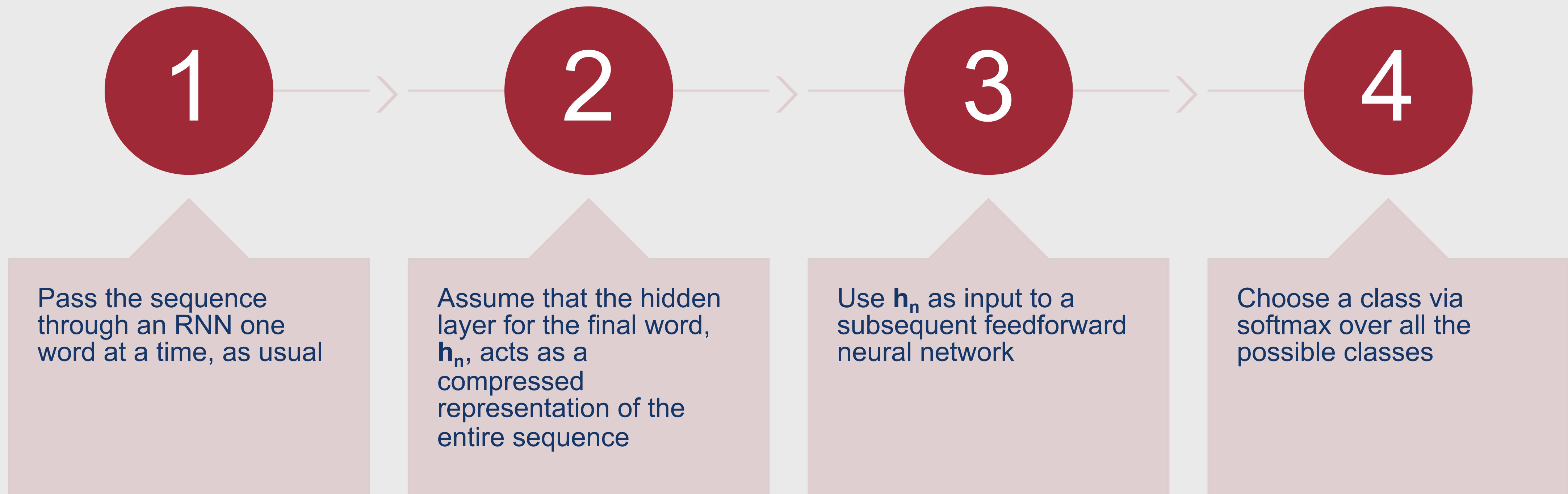
Sequence Labeling



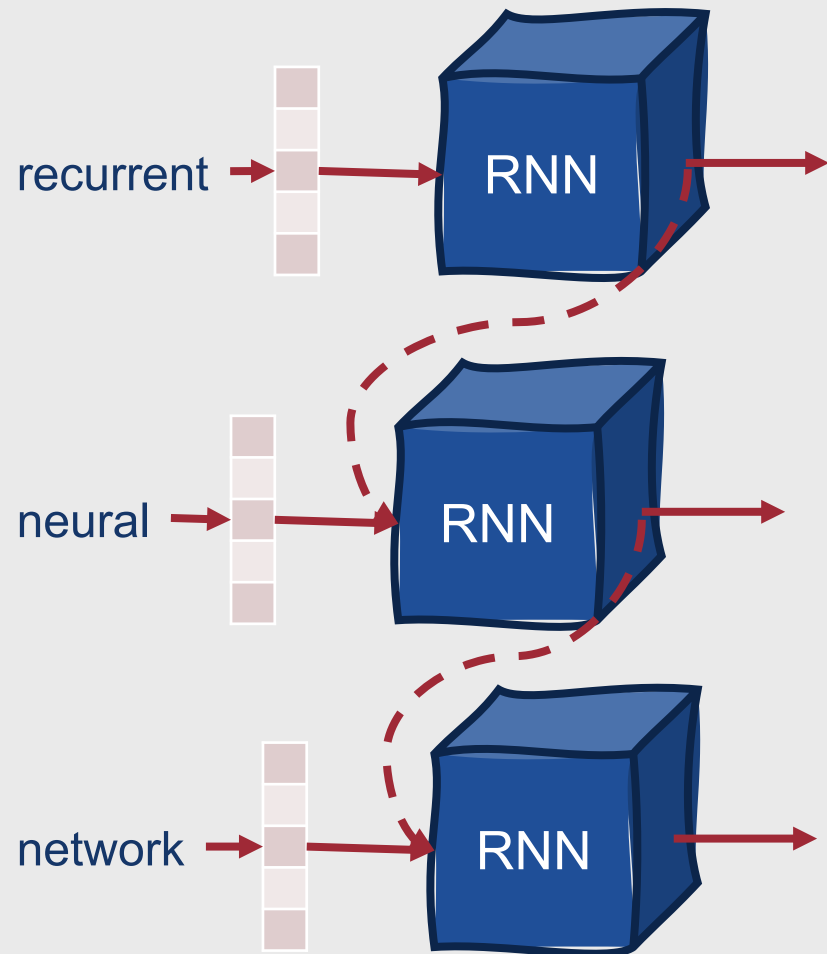
They're also useful for sequence classification!

- Task: Given an input sequence, **assign the entire sequence to a class** (rather than the individual tokens within it)

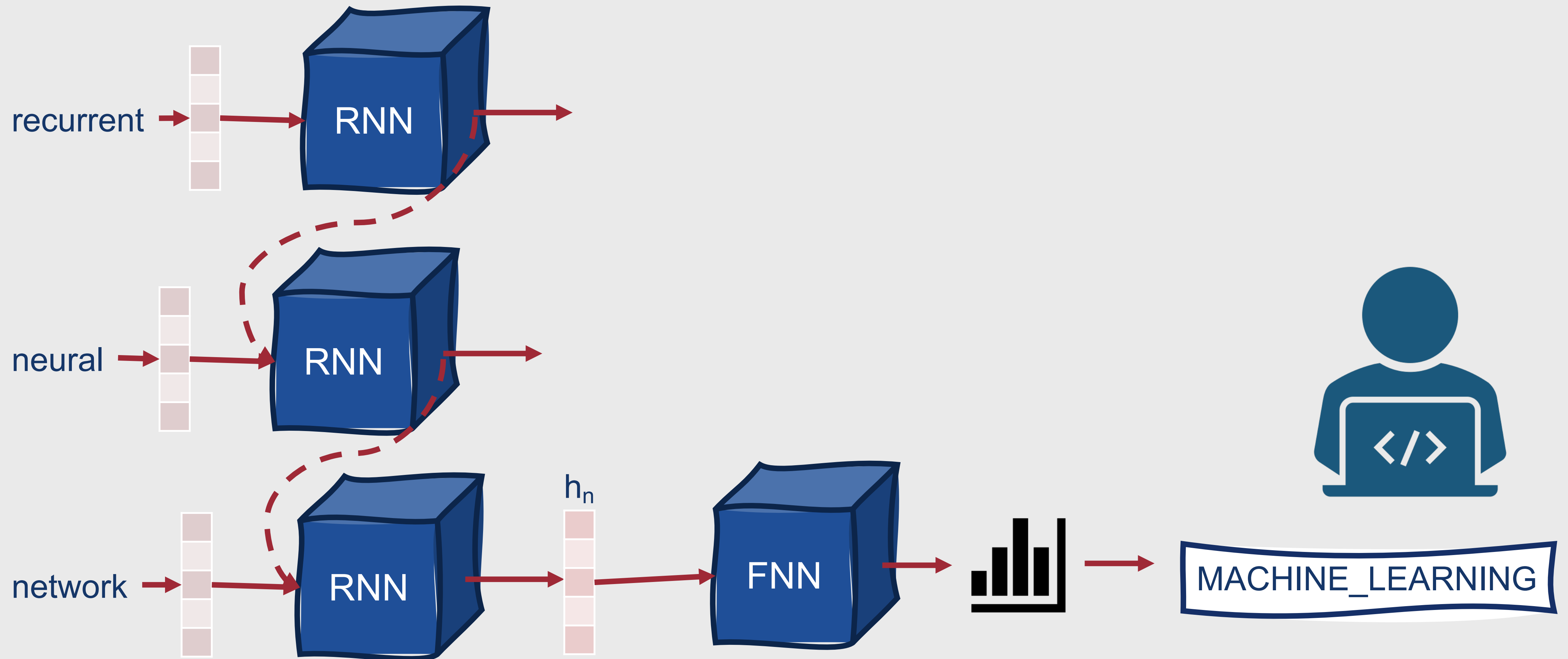
How to use RNNs for sequence classification?



Sequence Classification



Sequence Classification



Notes about Sequence Classification

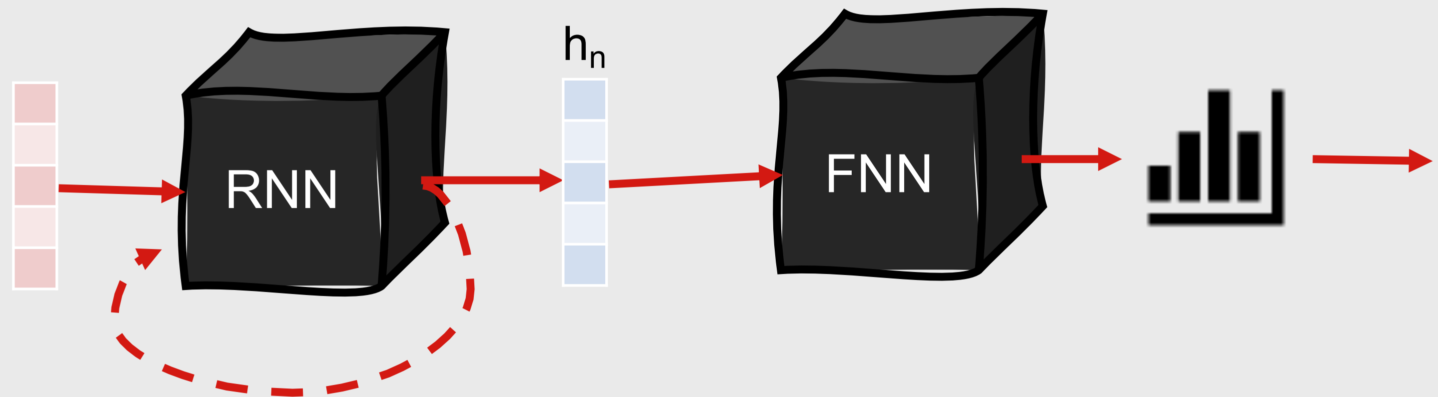
- No loss associated with intermediate outputs
- Loss function is based entirely on the final classification task!
- Errors are still backpropagated all the way through the RNN
- The process of adjusting weights the entire way through the network based on the loss from a downstream application is often referred to as **end-to-end training**

Where do we go from here?

- So far, we’ve discussed “vanilla” RNNs
- Many additional varieties exist!
- Extensions to the vanilla RNN model:
  - RNN + Feedforward layers
  - Stacked RNNs
  - Bidirectional RNNs

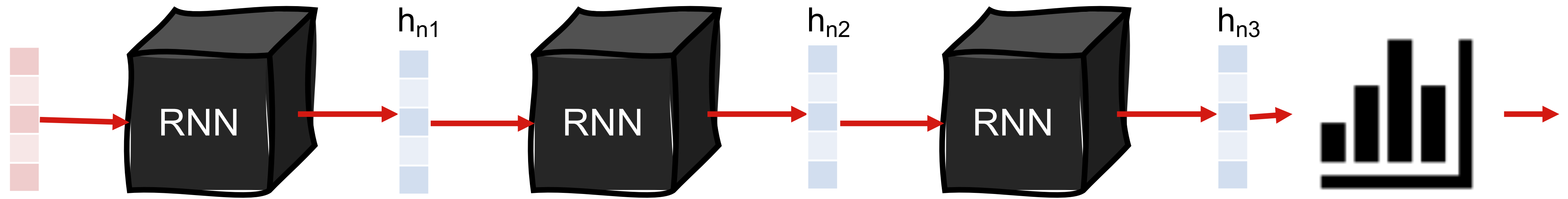
Where do we go from here?

- So far, we've discussed "vanilla" RNNs
- Many additional varieties exist!
- Extensions to the vanilla RNN model:
  - RNN + Feedforward layers
  - Stacked RNNs
  - Bidirectional RNNs



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

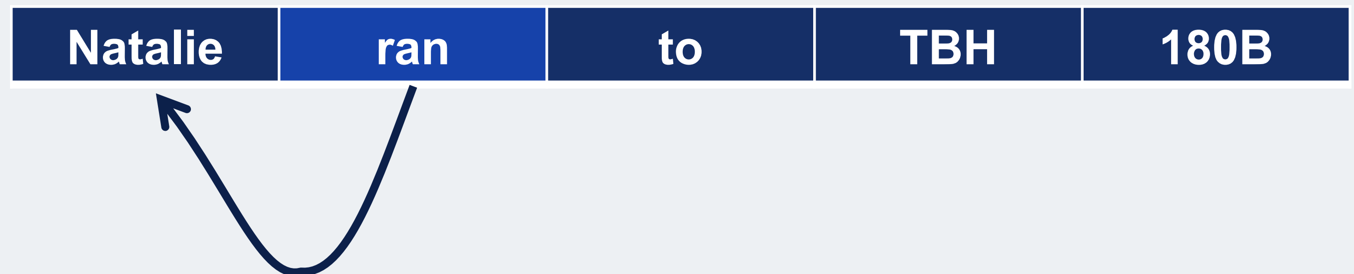


Stacked RNNs

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

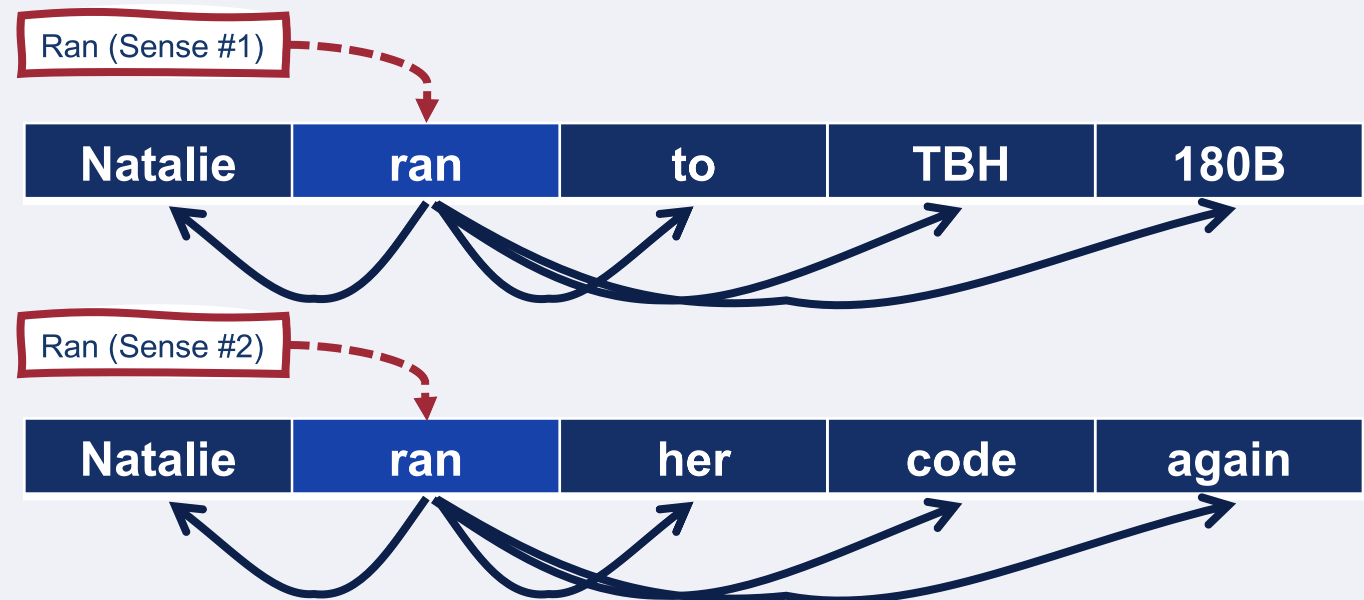
Bidirectional RNNs

- 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

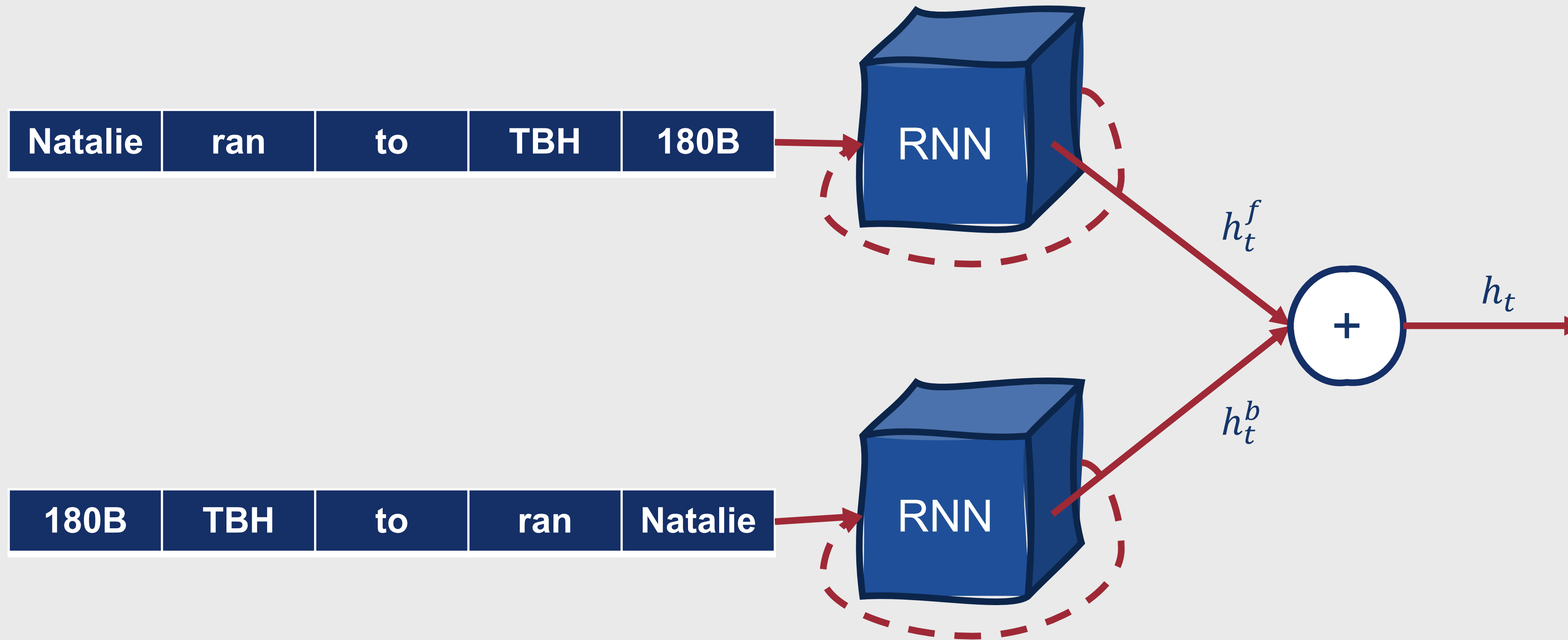
Bidirectional RNNs

- 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.

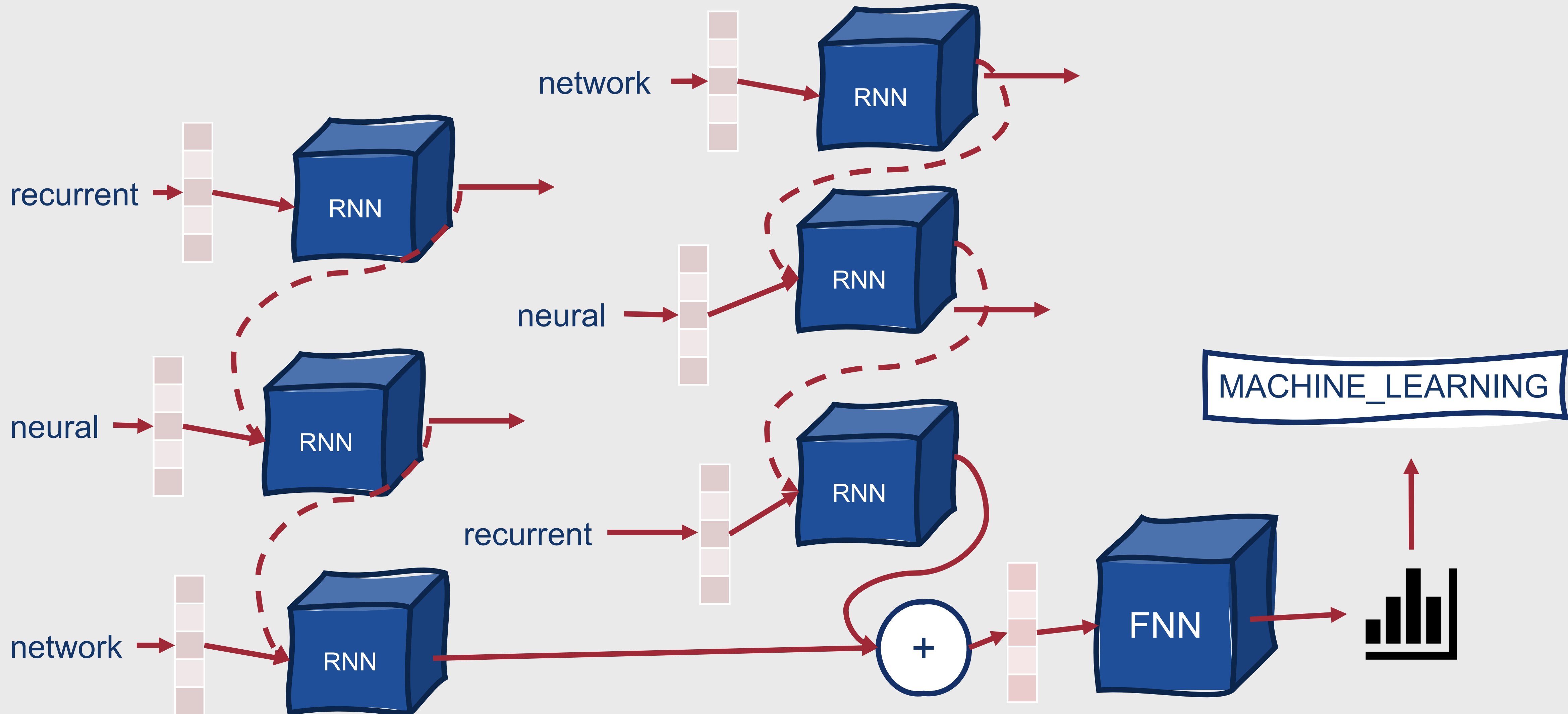
Bidirectional RNNs

Bidirectional RNNs

Bidirectional RNNs



Sequence Classification with a Bidirectional RNN



More advanced variations to come….

- Additional ways to combine RNNs
- Architectural modifications to allow better context management

Summary: Recurrent Neural Networks

- **Recurrent neural networks (RNNs)** are designed to make use of **temporal information** from input sequences
  - 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 **language modeling**, **text generation**, **sequence labeling**, and (when combined with a feedforward network) **sequence classification**
- More complex varieties of RNNs include:
  - **Stacked RNNs**
  - **Bidirectional RNNs**