

# **Recurrent neural networks and LSTMs**

**NLP Week 6**

Thanks to Dan Jurafsky for most of the slides this week!

# Plan for today

1. Simple Recurrent Neural Networks (RNNs)
2. RNN architectures
  1. For language modeling
  2. For classification
3. Long Short Term Memory Units (LSTMs)
4. *Group exercises*

**We will build language models adding to each layer of their complexity:**

## LANGUAGE MODELS

- **N-gram language models** – Probabilistic models of language in context
- **Embeddings** – vector semantics
- **Recurrent neural language models, LSTMs** – neural networks for language
- **Transformers** – large language model (LLM) architectures
- **Encoder models** – scoring tasks
- **Encoder-decoder models** – generative tasks
- **Finetuning and in-context learning** – current foundation model paradigm

# Modeling Time in Neural Networks

**Language is inherently temporal.**

Yet the simple NLP classifiers we've seen (for example for sentiment analysis) mostly ignore time

- (Feedforward neural LMs (and the transformers we'll see later) use a "moving window" approach to time.)

Here we introduce a deep learning architecture with a different way of representing time

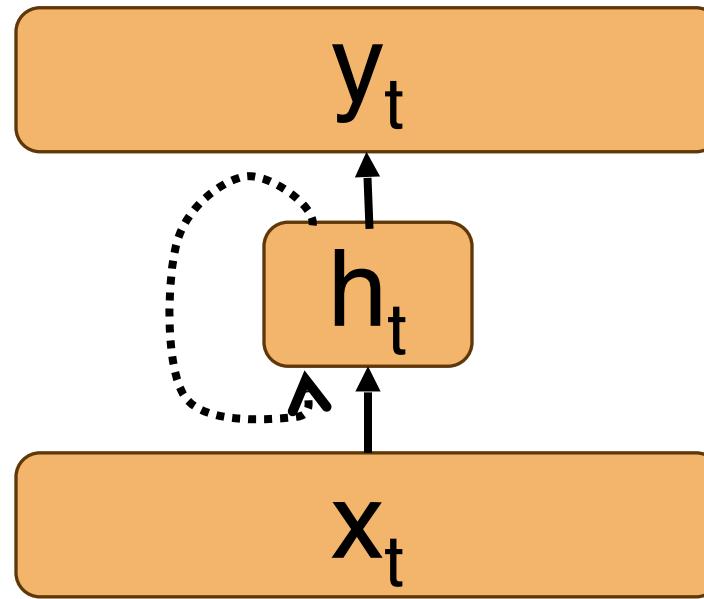
- **RNNs and their variants like LSTMs**

# **Recurrent Neural Networks (RNNs)**

**Any network that contains a cycle within its network connections.**

The value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input.

# Simple recurrent neural units (SRN or Elman Net)



The hidden layer has a recurrence as part of its input

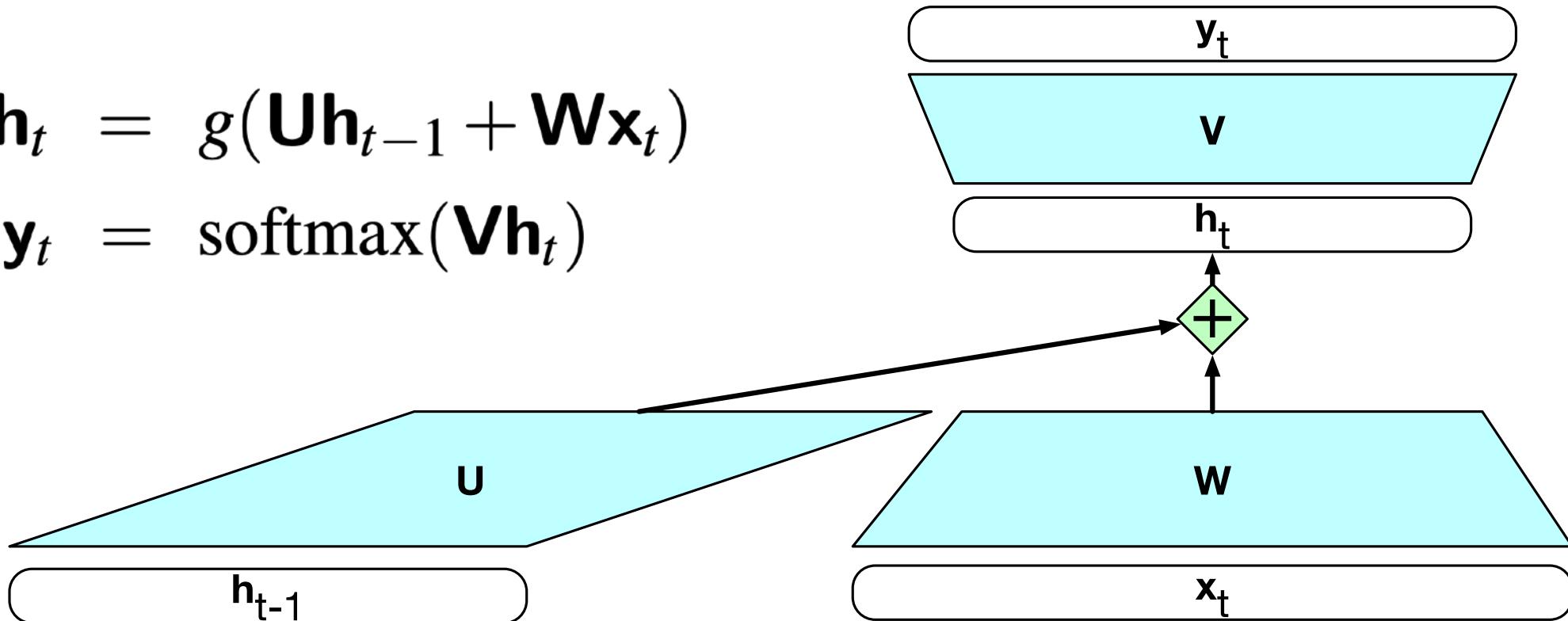
The activation value  $h_t$  depends on  $x_t$  but also  $h_{t-1}$ !

# Forward inference in simple RNNs

Very similar to the feedforward networks we've seen!

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t)$$

$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t)$$



# Inference has to be incremental

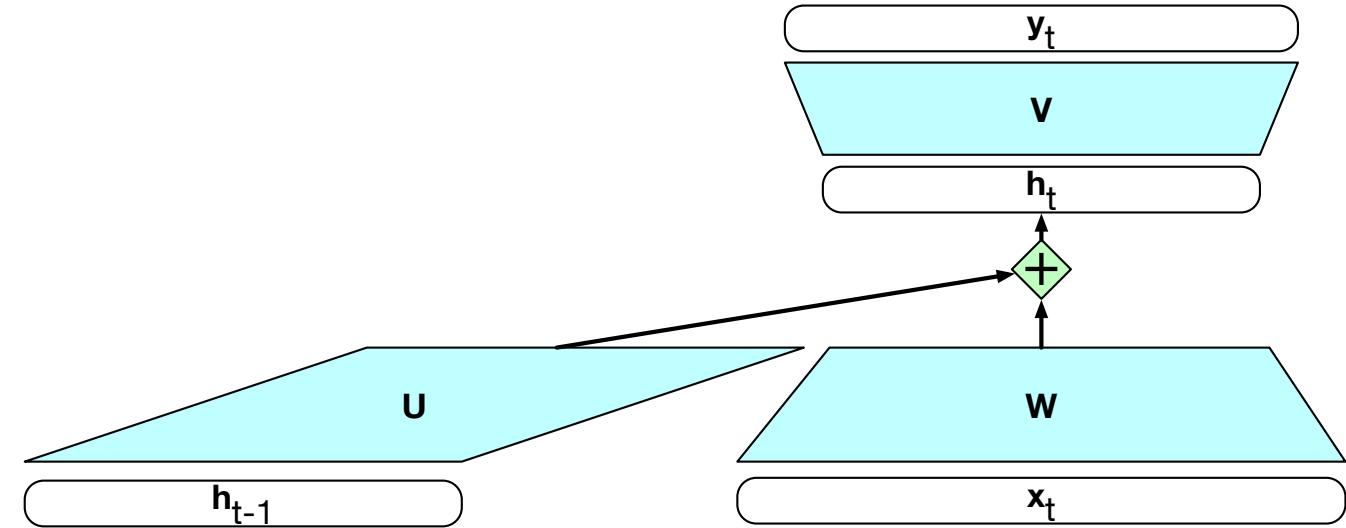
Computing  $h$  at time  $t$  requires that we first computed  $h$  at the previous time step!

```
function FORWARDRNN( $\mathbf{x}$ , network) returns output sequence  $\mathbf{y}$ 
     $\mathbf{h}_0 \leftarrow 0$ 
    for  $i \leftarrow 1$  to LENGTH( $\mathbf{x}$ ) do
         $\mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)$ 
         $\mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)$ 
    return  $\mathbf{y}$ 
```

# Training in simple RNNs

Just like feedforward training:

- training set,
- a loss function,
- backpropagation



Weights that need to be updated:

- $W$ , 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.

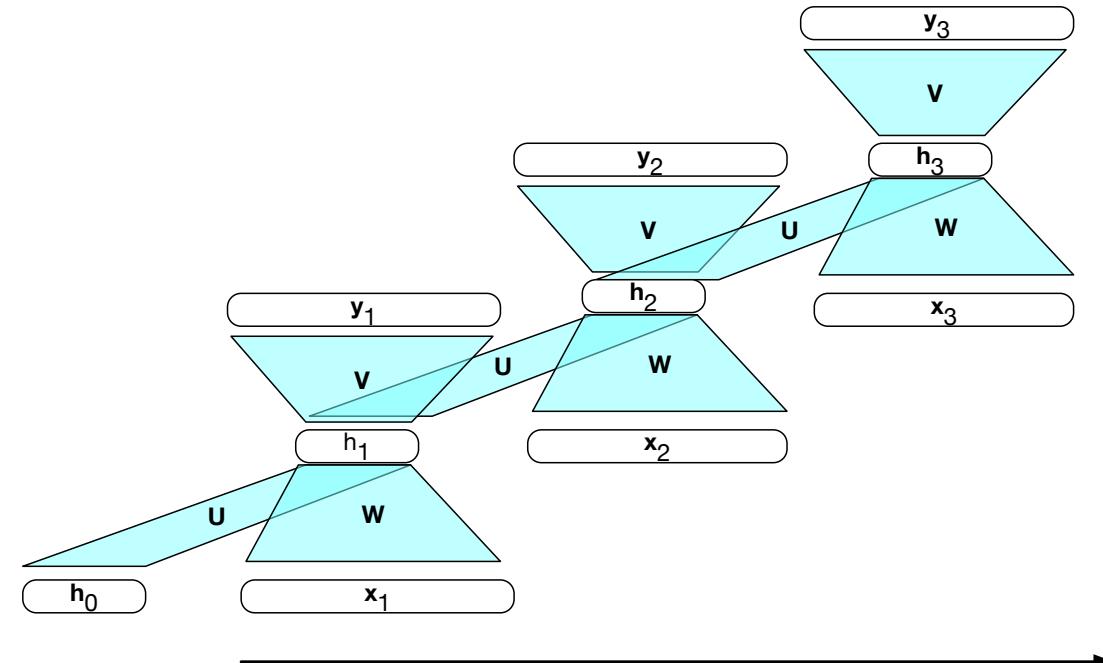
# Training in simple RNNs: unrolling in time

**Unlike feedforward networks:**

1. To compute loss function for the output at time  $t$  we need the hidden layer from time  $t - 1$ .
2. hidden layer at time  $t$  influences the output at time  $t$  and hidden layer at time  $t+1$  (and hence the output and loss at  $t+1$ ).

**So:** to measure error accruing to  $h_t$ ,

we need to know its influence on both the current output as well as the ones that follow.

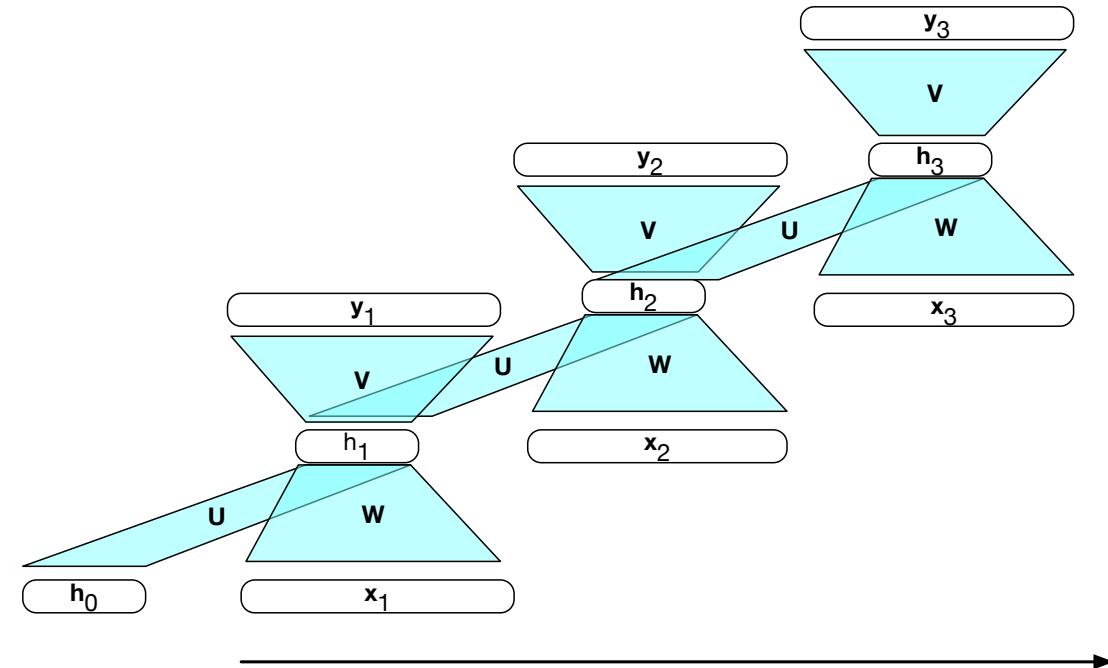


# Training in simple RNNs: unrolling in time

We unroll the RNN into a feedforward computational graph eliminating recurrence!

**Given an input sequence:**

1. Generate an unrolled feedforward network specific to input
2. Use graph to train weights directly via ordinary backprop



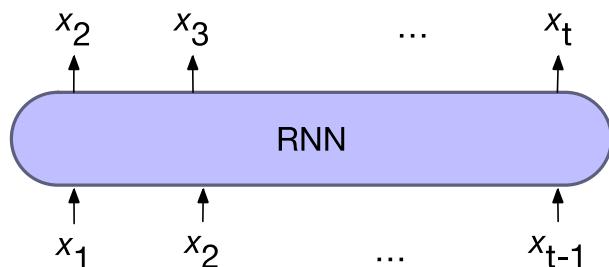
# RNN Architectures

We will see three types of RNN-based architectures that can be used for different tasks:

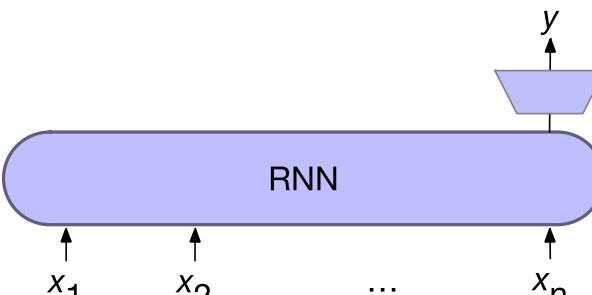
**1. RNNs for language modeling**

**2. RNNs for classification**

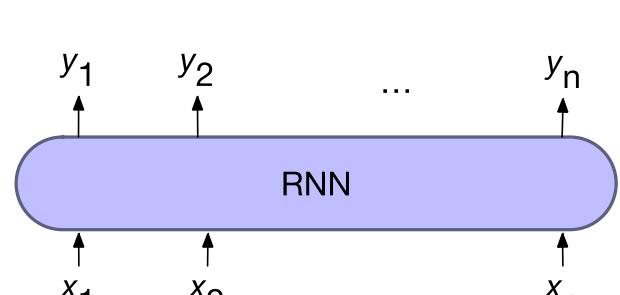
**3. RNNs for sequences**



c) language modeling



b) sequence classification



a) sequence labeling

# Reminder: Language Modeling

**Modeling the probability of words in context.**

**Task:** predict next word  $w_t$

given prior words  $w_{t-1}, w_{t-2}, w_{t-3}, \dots$

**Problem with N-gram and feedforward NNs:** Dealing with sequences of arbitrary length.

**Previous solution:** Sliding windows (of fixed length)

$$P(w_t | w_{t-N+1}^{t-1})$$

**What about RNNs ?** No limit on context size! All prior words count.

$$P(w_t | w_1^{t-1})$$

# **The size of the conditioning context for different LMs**

## **The n-gram LM:**

Context size is the  $n - 1$  prior words we condition on.

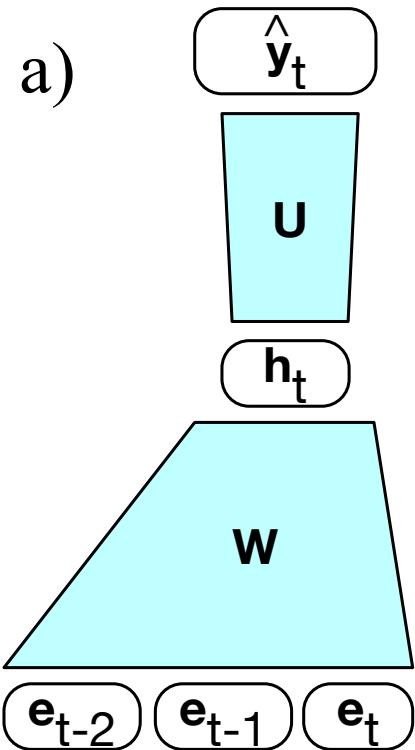
## **The feedforward LM:**

Context is the window size.

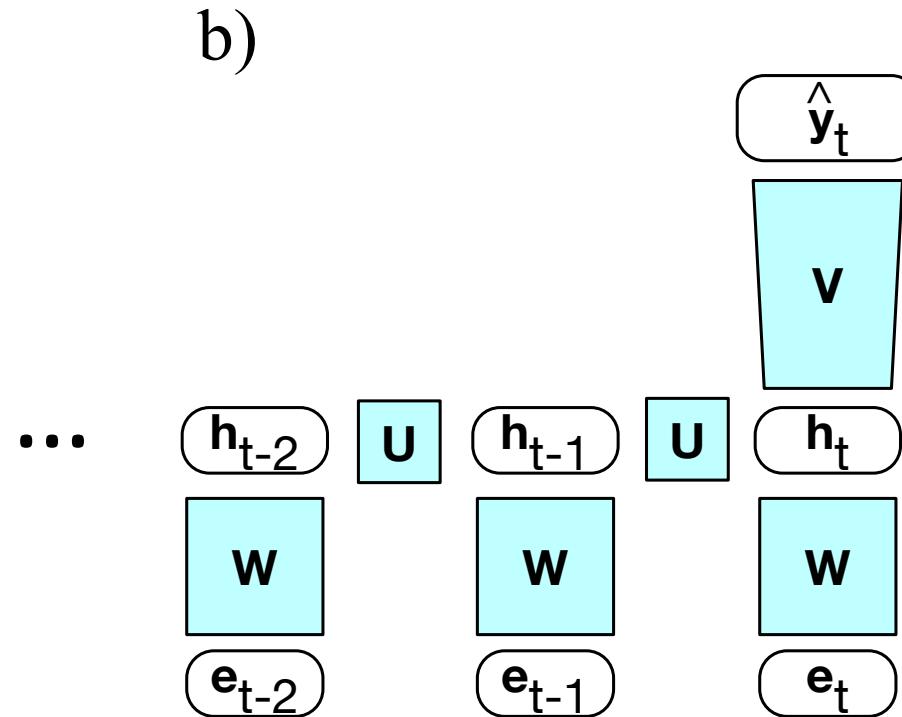
## **The RNN LM:**

No fixed context size;  $h_{t-1}$  represents entire history

# Feed forward LMs vs RNN LMs



FFN



RNN

# Forward inference in the RNN LM

Given input  $X$  of  $N$  tokens

$$\mathbf{X} = [\mathbf{x}_1; \dots; \mathbf{x}_t; \dots; \mathbf{x}_N]$$

Use embedding matrix to get the embedding for current token  $x_t$

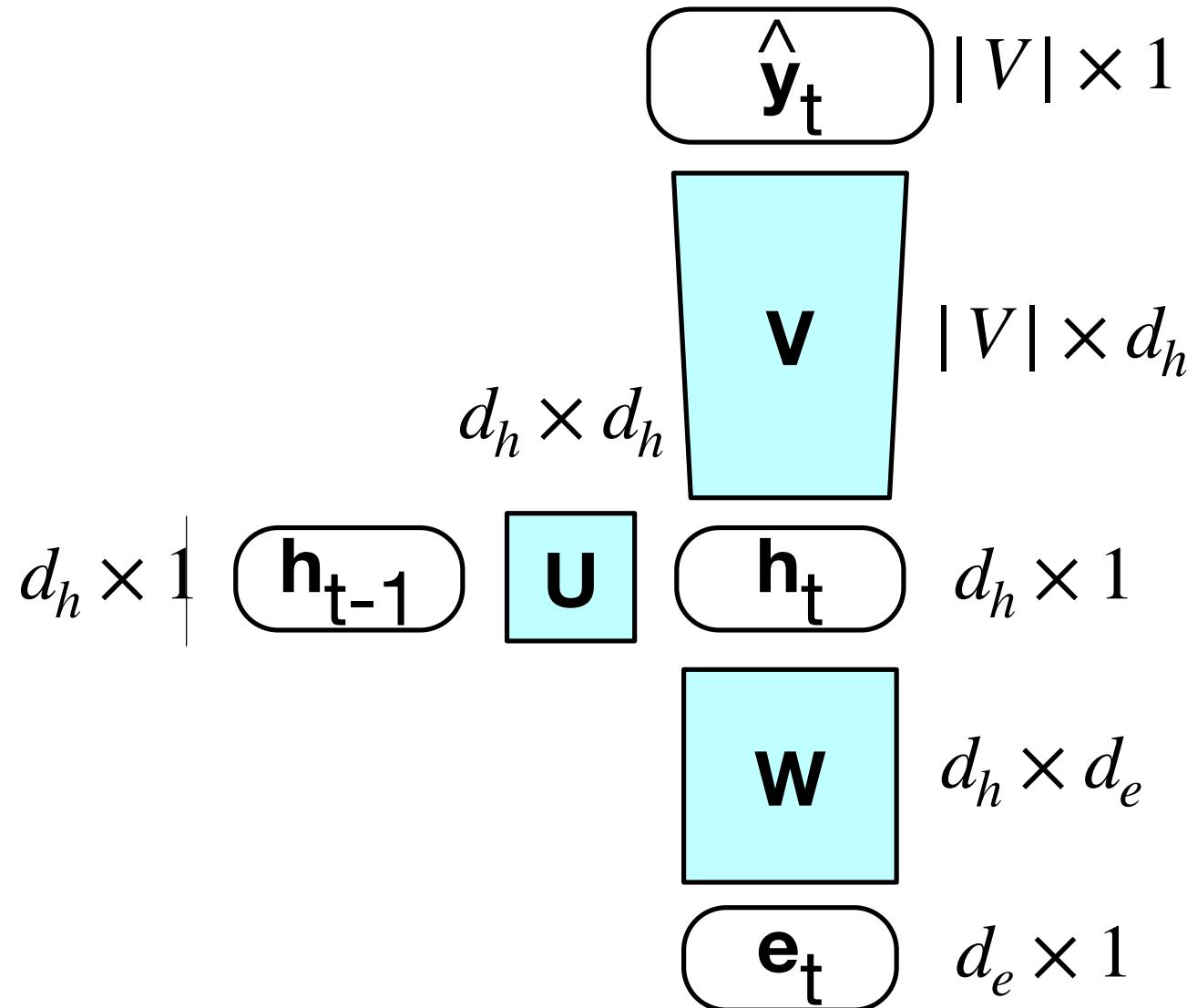
Combine ...

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t$$

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t)$$

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t)$$

# Shapes



# **Computing the probability that the next word is word k**

$$P(w_{t+1} = k | w_1, \dots, w_t) = \hat{\mathbf{y}}_t[k]$$

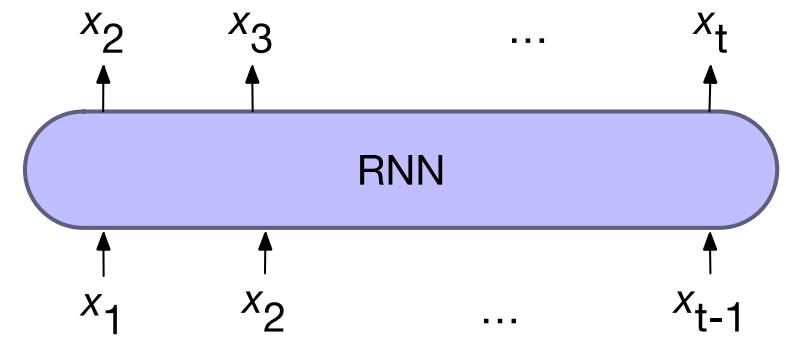
# **Computing the probability, or scoring, a sentence**

$$P(w_{1:n}) = \prod_{i=1}^n P(w_i | w_{1:i-1})$$

$$= \prod_{i=1}^n \hat{\mathbf{y}}_i[w_i]$$

# Training RNN LM

- **Self-supervision (like with Feed forward LM)**
  - take a corpus of text as training material
  - ask the model to predict the next word
  - (Unlike feed-forward) at each time step  $t$  !
- **Why called self-supervised ?** we don't need human labels; the text is its own supervision signal
- **We train the model to :**
  - minimize the error in predicting the true next word in the training sequence, using cross-entropy as the loss function.



c) language modeling

# Cross-entropy loss

Minimizes the difference between (1) a predicted probability distribution and (2) the correct distribution.

$$L_{CE} = - \sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w]$$

**CE loss for LMs can actually be simplified since:**

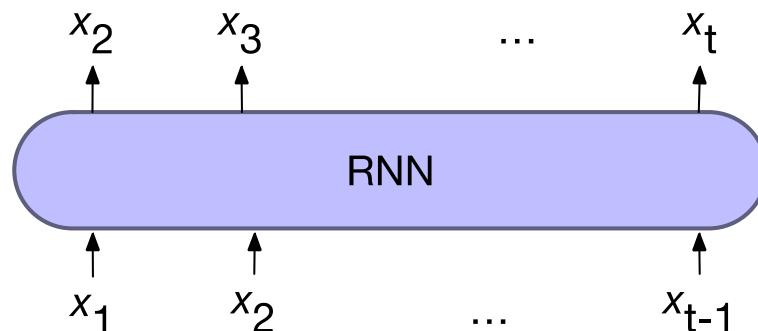
- Correct distribution  $\mathbf{y}_t$  is a one-hot vector over vocab (i.e. where prob of actual next word is 1, and all the other entries are 0.)
- CE loss for LMs is only determined by the probability of next word.
- So at time t, CE loss is:

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = - \log \hat{\mathbf{y}}_t[w_{t+1}]$$

# Teacher forcing

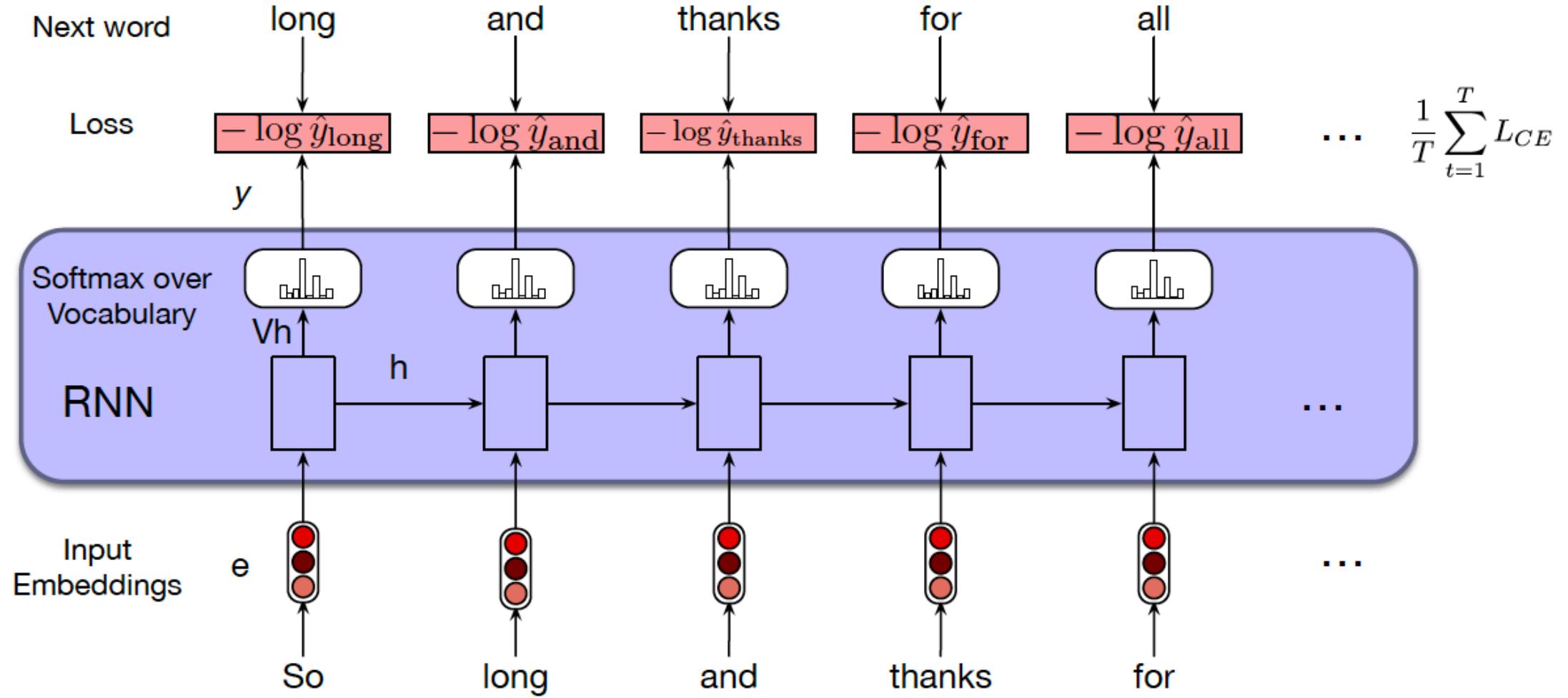
We always give the model the correct history to predict the next word (rather than feeding the model the previous time step next word prediction — which could be wrong).

This is called **teacher forcing** (in training we **force** the context to be correct based on the gold words).



c) language modeling

# Summary: Training RNN for language modeling



# Weight tying

An optional architectural modification when  $d_e = d_h$ .

When the embedding dimension and hidden dimension are the same, then the embedding matrix  $\mathbf{E}$  and the final layer matrix  $\mathbf{V}$  are of similar shape:  $\mathbf{E}$  is  $d \times |\mathcal{V}|$  while  $\mathbf{V}$  is  $|\mathcal{V}| \times d$

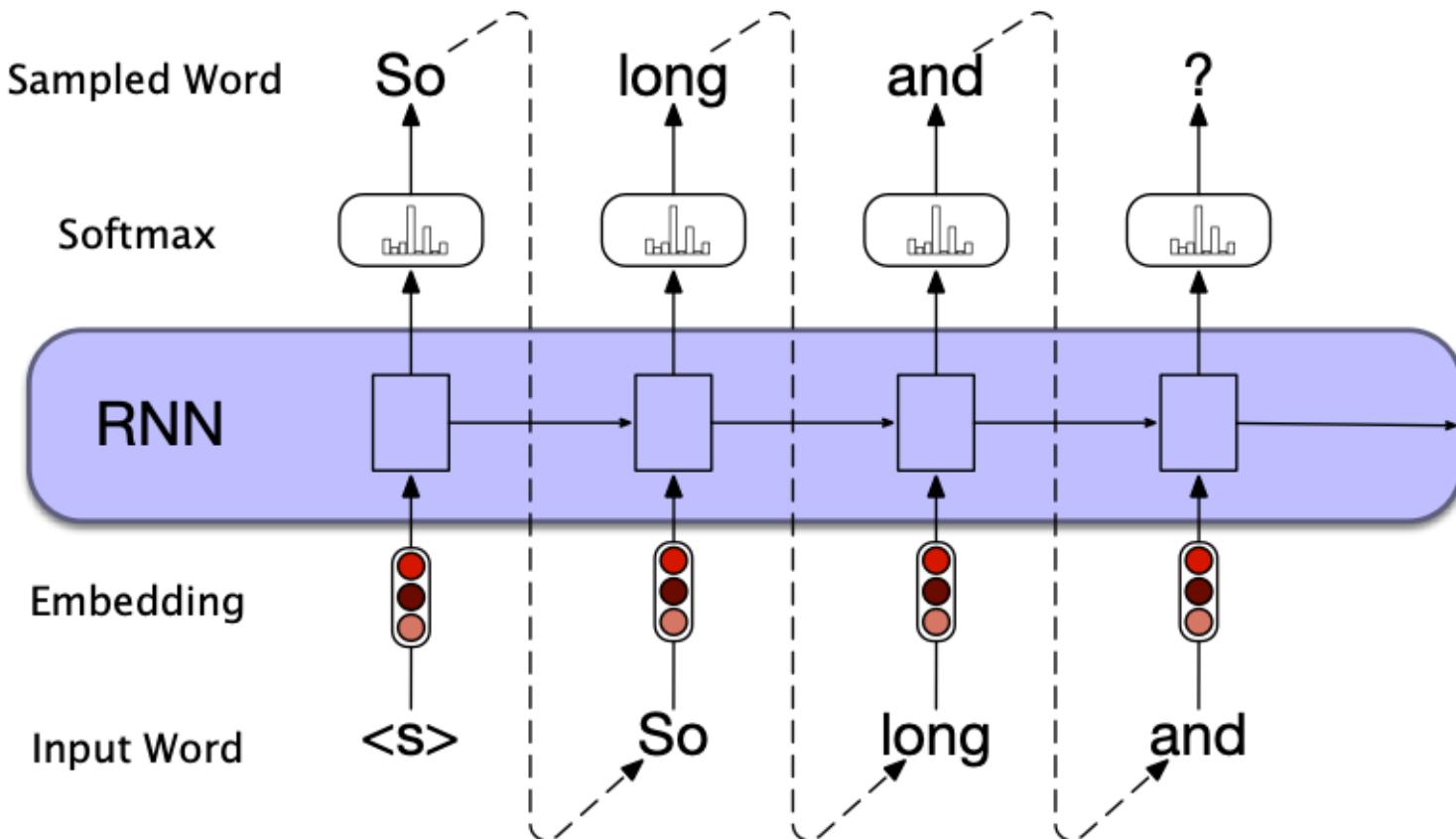
Instead of having separate matrices, we just tie them together, transposing  $\mathbf{E}^\top$  instead of  $\mathbf{V}$ , so embeddings appear twice:

$$\mathbf{e}_t = \mathbf{Ex}_t$$

$$\mathbf{h}_t = g(\mathbf{Uh}_{t-1} + \mathbf{We}_t)$$

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{E}^\top \mathbf{h}_t)$$

# Autoregressive generation from a RNN LM



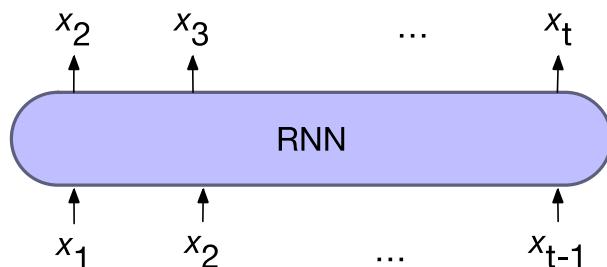
# RNN Architectures

We will see three types of RNN-based architectures that can be used for different tasks:

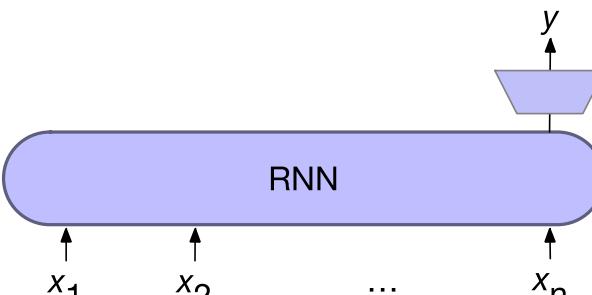
**1. RNNs for language modeling**

**2. RNNs for classification**

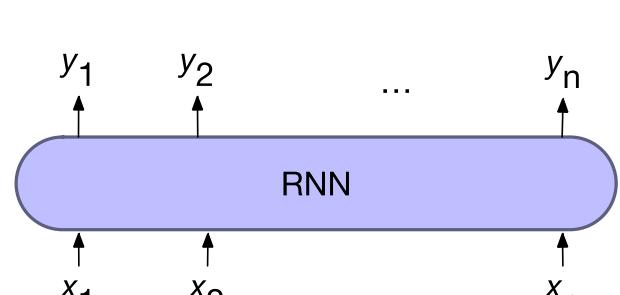
**3. RNNs for sequences**



c) language modeling



b) sequence classification

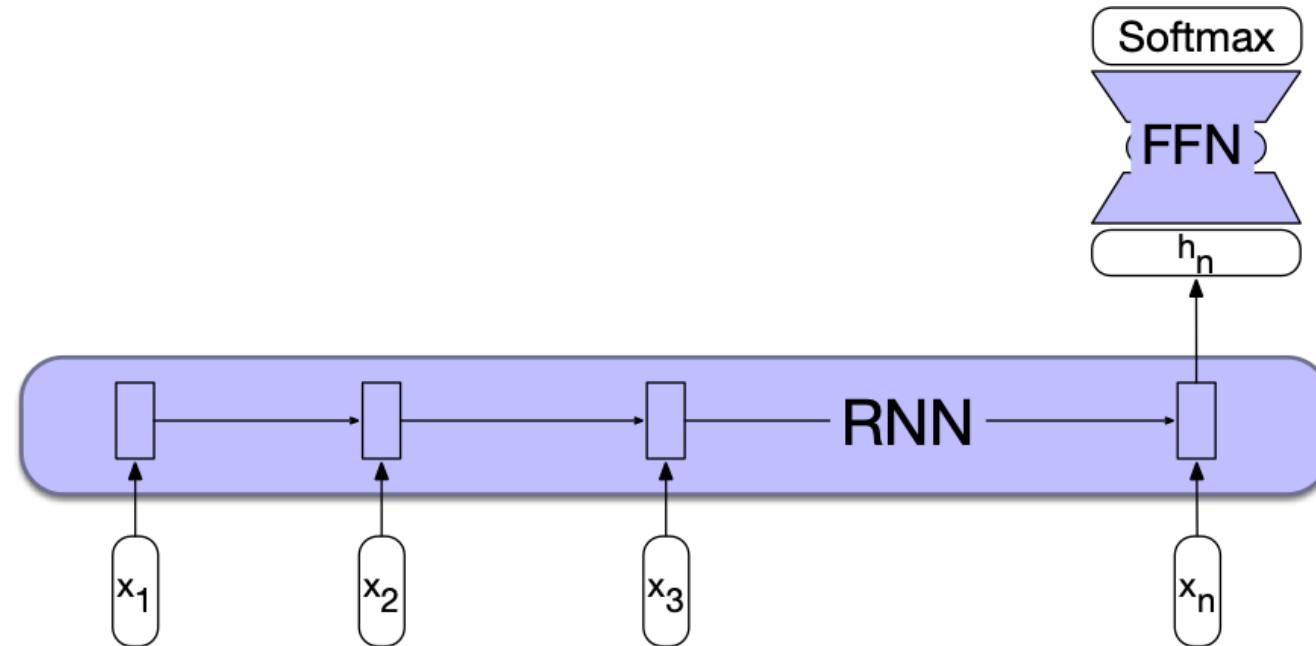


a) sequence labeling

# RNNs for text classification

There is only one final output prediction, rather than an output at each state.

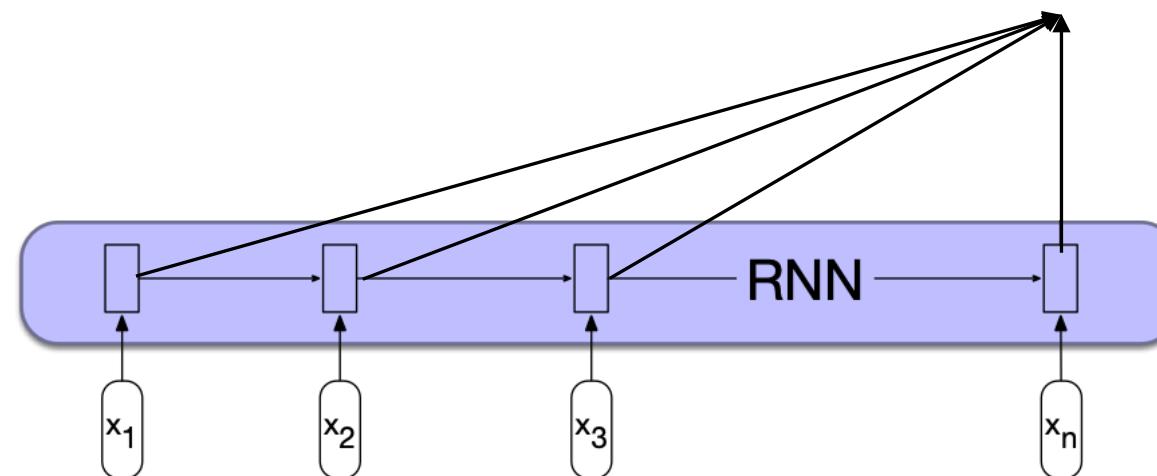
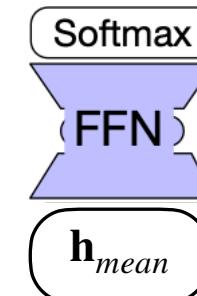
We can use just the final hidden state as a sequence encoding that gets fed to a regular feedforward NN for classification.



# RNNs for text classification

Alternatively, Instead of taking the last hidden state only, we can use some pooling function of all the hidden states, like **mean pooling**.

$$\mathbf{h}_{mean} = \frac{1}{n} \sum_{i=1}^n \mathbf{h}_i$$



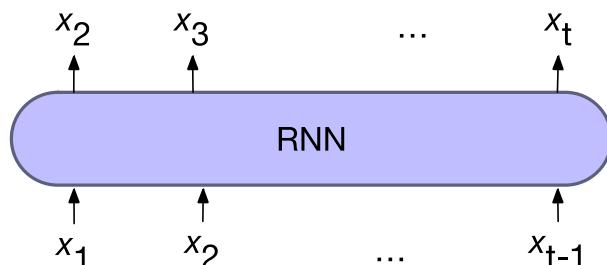
# RNN Architectures

We will see three types of RNN-based architectures that can be used for different tasks:

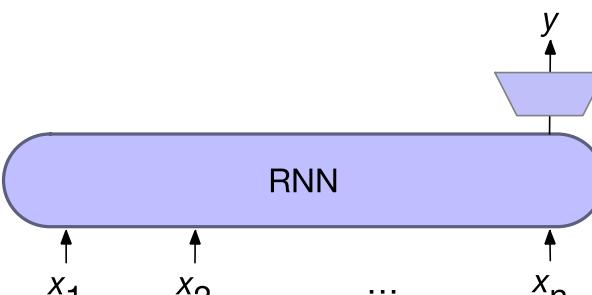
**1. RNNs for language modeling**

**2. RNNs for classification**

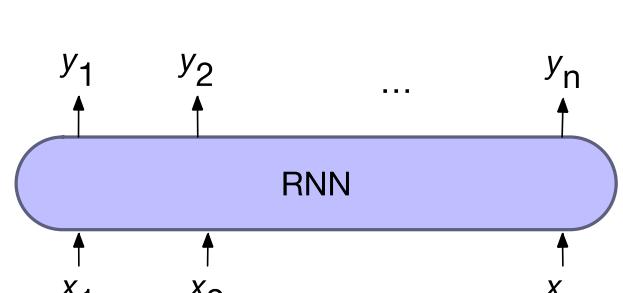
**3. RNNs for sequences**



c) language modeling



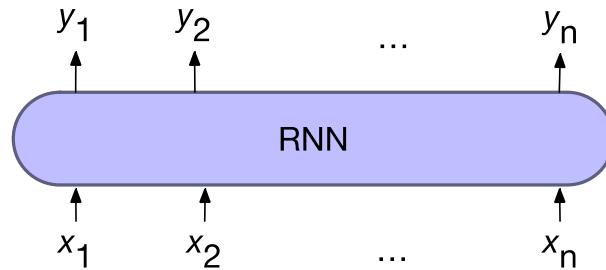
b) sequence classification



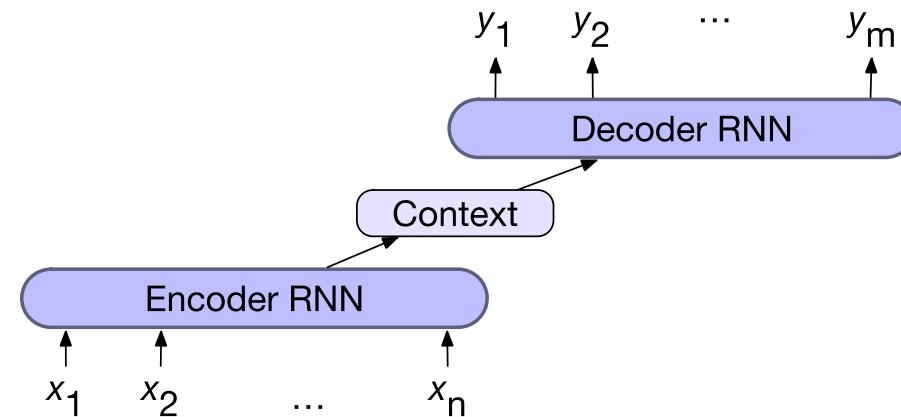
a) sequence labeling

# Two types of sequence prediction tasks

1. **sequence labeling** : When output sequence length matches input
2. **seq-to-seq or encoder-decoder** : When output and input sequence are of different lengths (Week 10 of this course)



a) sequence labeling

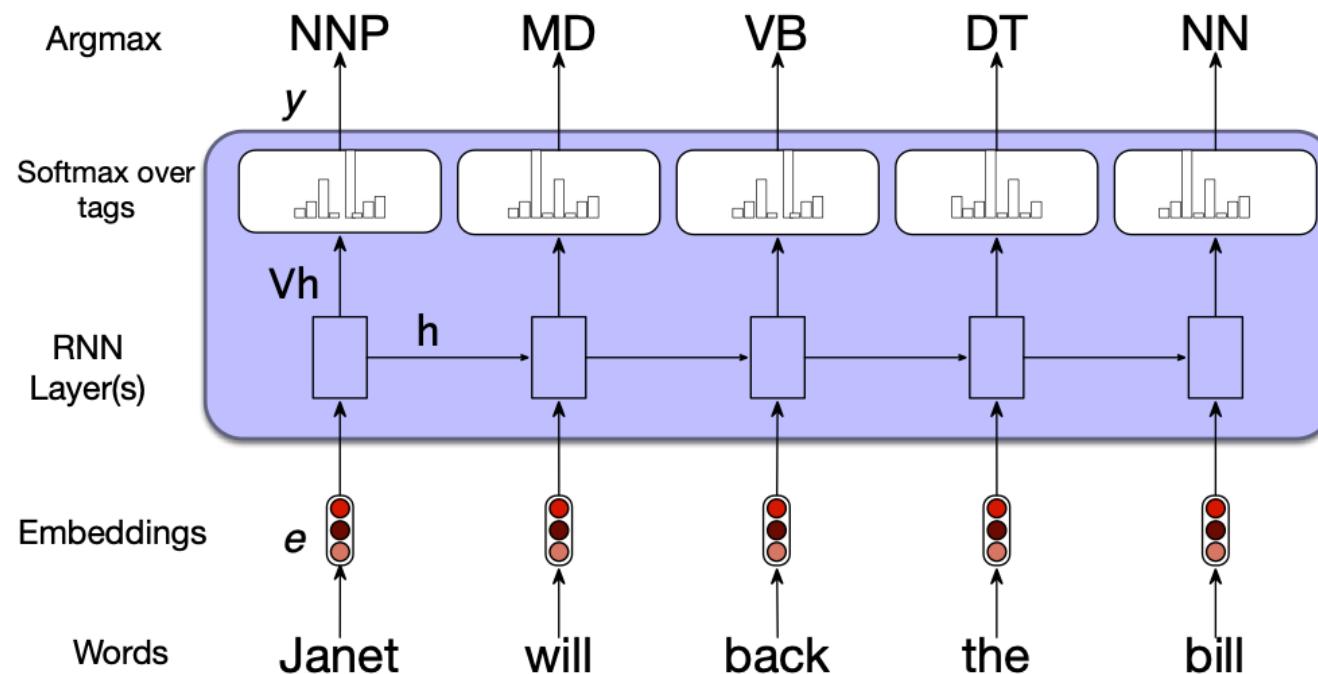


d) encoder-decoder

# RNNs for sequence labeling

Assign a label to each element of a sequence (eg. POS tagging):

- The main difference from LM architecture and classification architecture is that the output dimension for output layer is over labels (not vocab), and there is an output prediction at each time step (not just the last one).



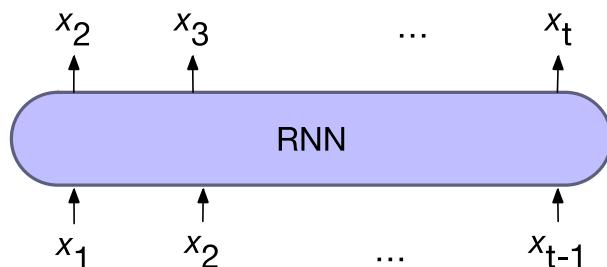
# RNN Architectures

We will see three types of RNN-based architectures that can be used for different tasks:

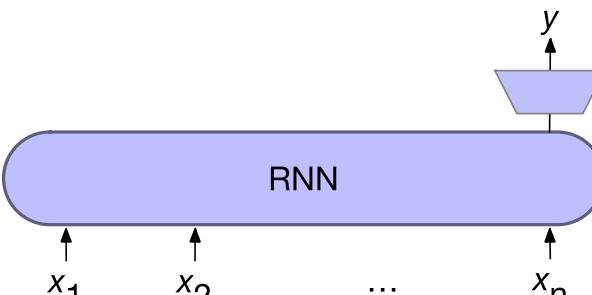
**1. RNNs for language modeling**

**2. RNNs for classification**

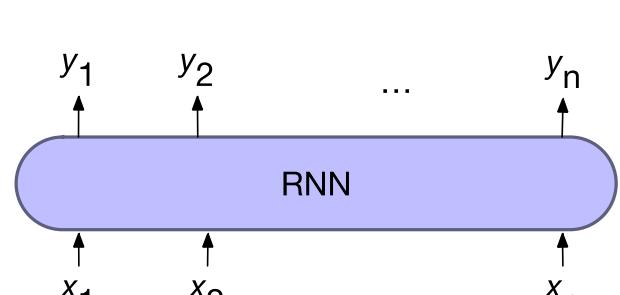
**3. RNNs for sequences**



c) language modeling



b) sequence classification



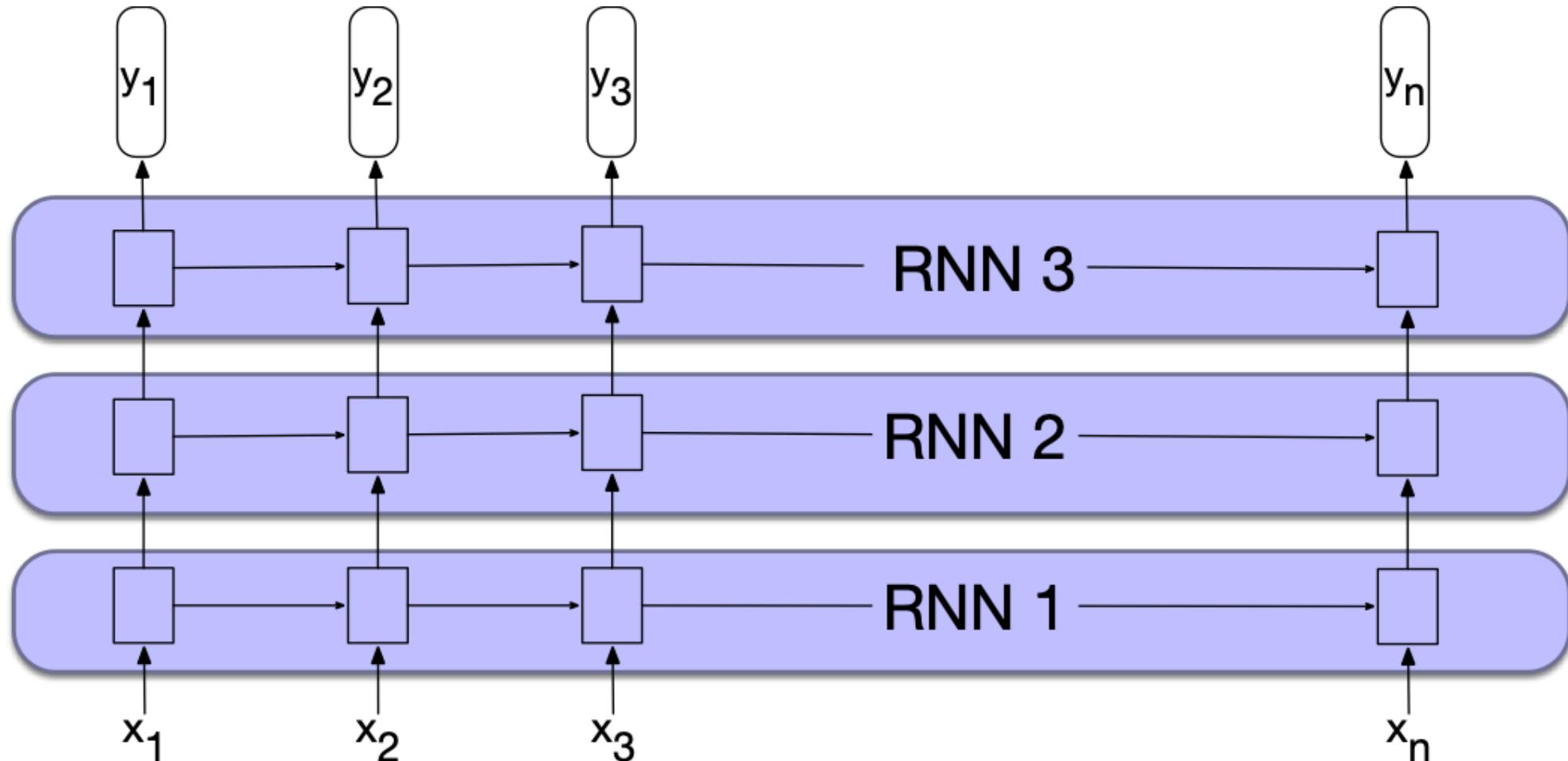
a) sequence labeling

# Other architectural modifications

We have seen different ways to use **simple single layered RNN in a unidirectional manner**. In all use cases we can additionally modify our architectures to include:

- 1. Multiple layers or Stacked RNNs**
- 2. Bidirectional encoding**

# Stacked RNNs

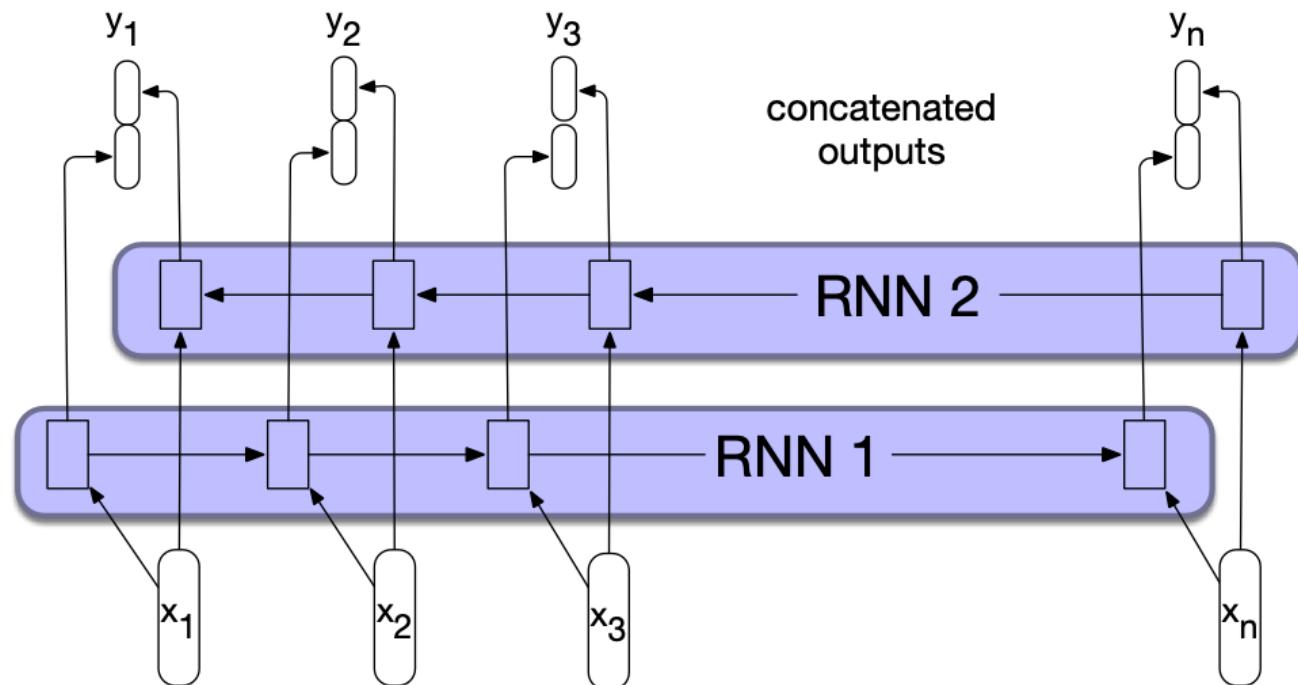


# Bidirectional RNNs

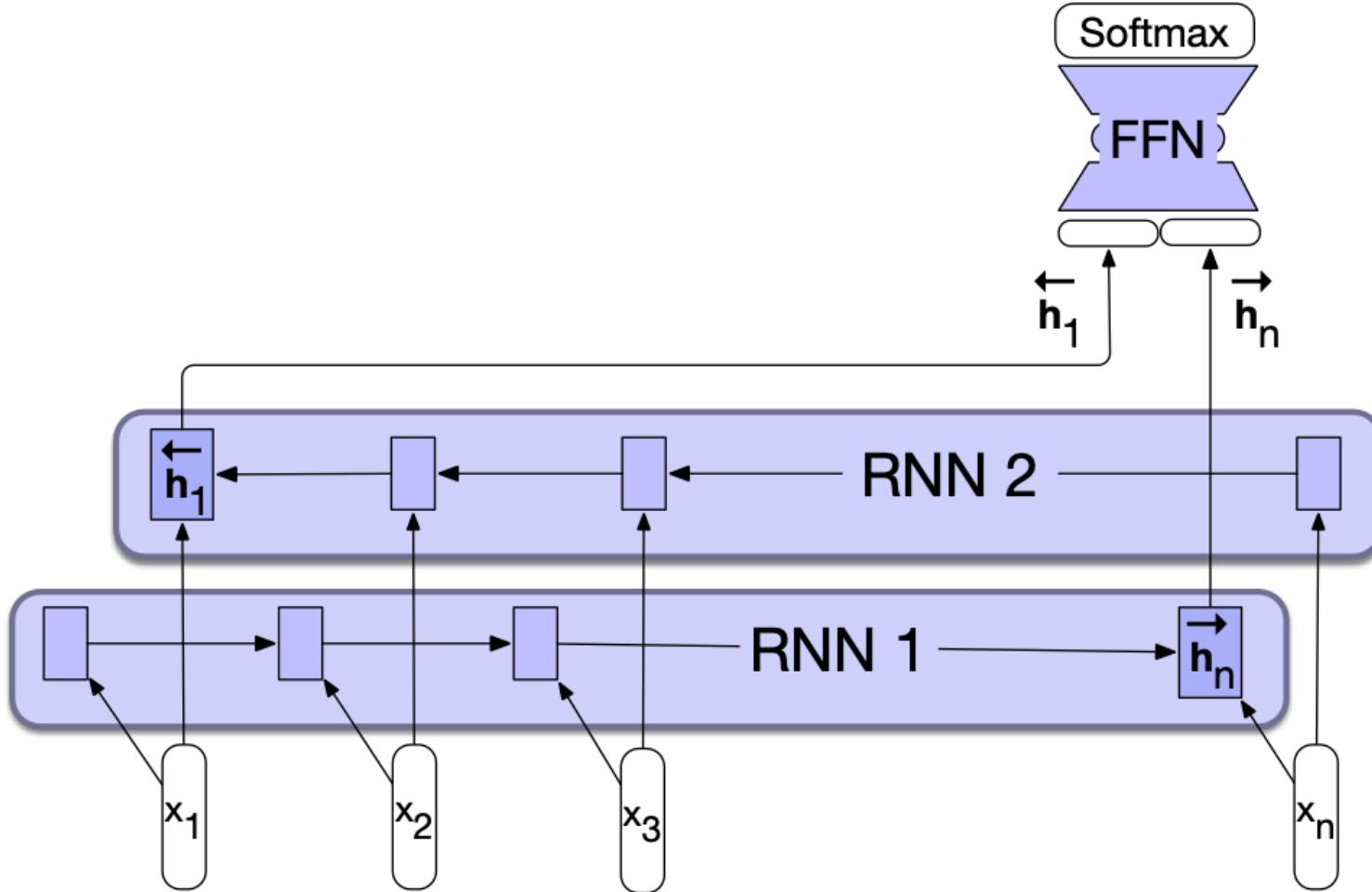
$$\mathbf{h}_t^f = \text{RNN}_{\text{forward}}(\mathbf{x}_1, \dots, \mathbf{x}_t)$$

$$\mathbf{h}_t^b = \text{RNN}_{\text{backward}}(\mathbf{x}_t, \dots, \mathbf{x}_n)$$

$$\begin{aligned}\mathbf{h}_t &= [\mathbf{h}_t^f ; \mathbf{h}_t^b] \\ &= \mathbf{h}_t^f \oplus \mathbf{h}_t^b\end{aligned}$$



# Bidirectional RNNs for classification



# **Other types of recurrent units**

**Recall:** RNN is any network that contains a cycle within its network connections, i.e. the value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input.

**All the architectures we have just seen can be implemented with ANY type of RNN.**

We saw simple recurrent neural units (SRN). Two other common types are gated recurrent units (GRU) and **long-short-term memory units (LSTM)**

# Motivating the LSTM: dealing with distance

**Hidden layers in RNN are forced to do two things:**

- a) Provide information useful for the current decision,
- b) Update and carry forward information required for future decisions.

**Leads to two problems :**

1. It's hard to assign probabilities accurately when context is very far away:  
*'The flights the airline was canceling were full'*  
this requires knowing and prioritizing information from far away prior states over closer info.
2. During backprop, we have to repeatedly multiply gradients through time and many hidden states which can lead to **the "vanishing gradient" problem**

# **LSTM: Long short-term memory network**

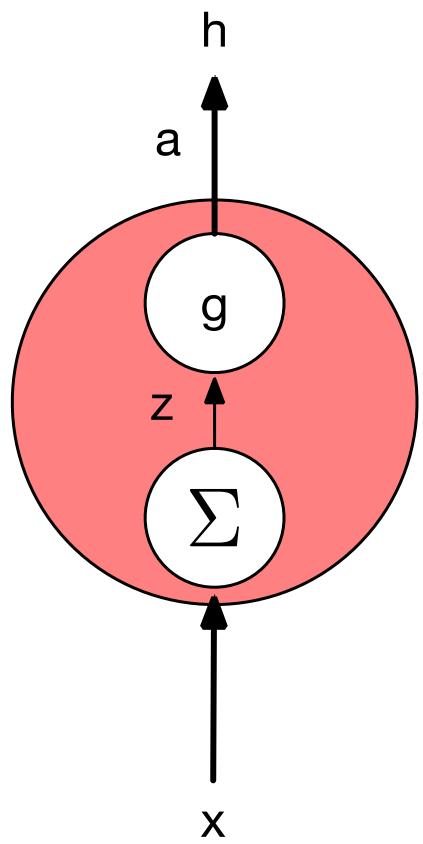
**LSTMs divide the context management problem into two subproblems:**

1. removing information no longer needed from the context,
2. adding information likely to be needed for later decision making

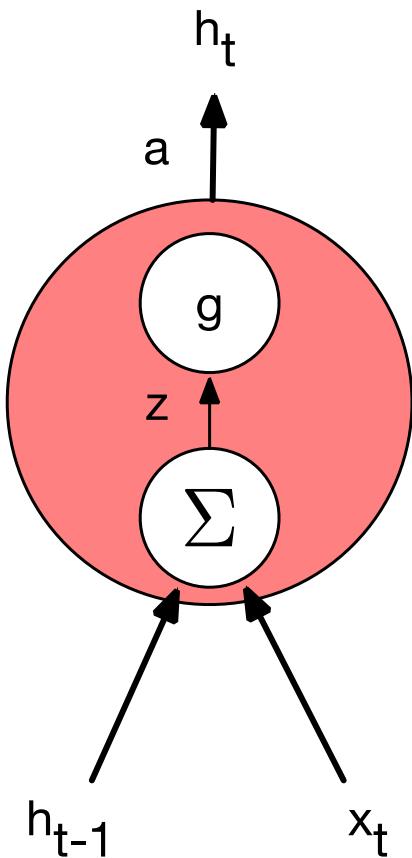
**LSTMs add:**

- explicit context layer
- Neural circuits with **gates** to control information flow

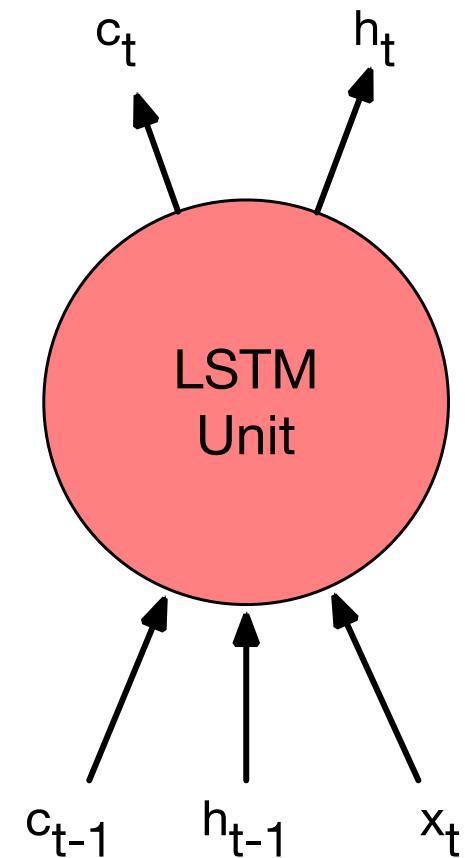
# Difference between neural units



(a)  
Feedforward unit

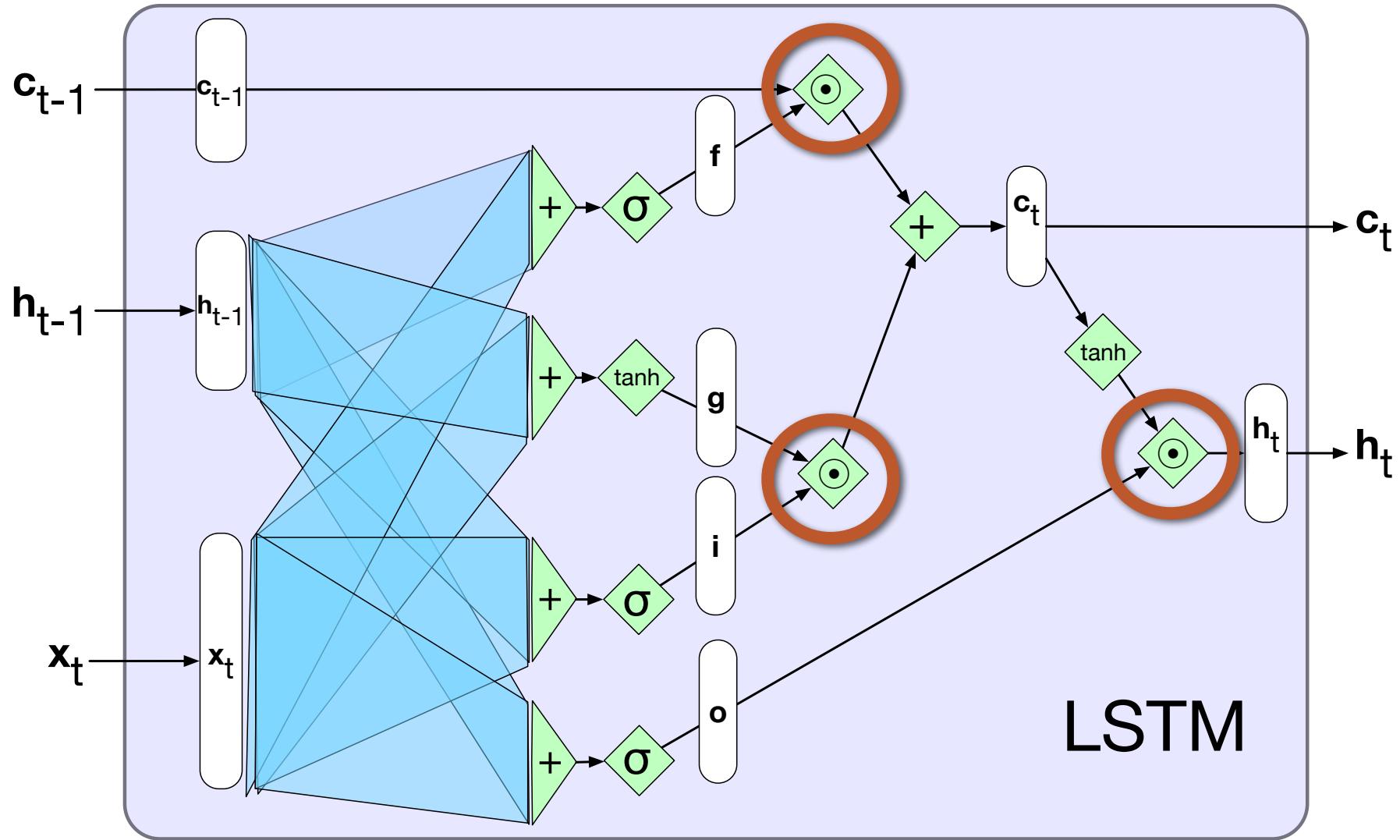


(b)  
Simple recurrent unit

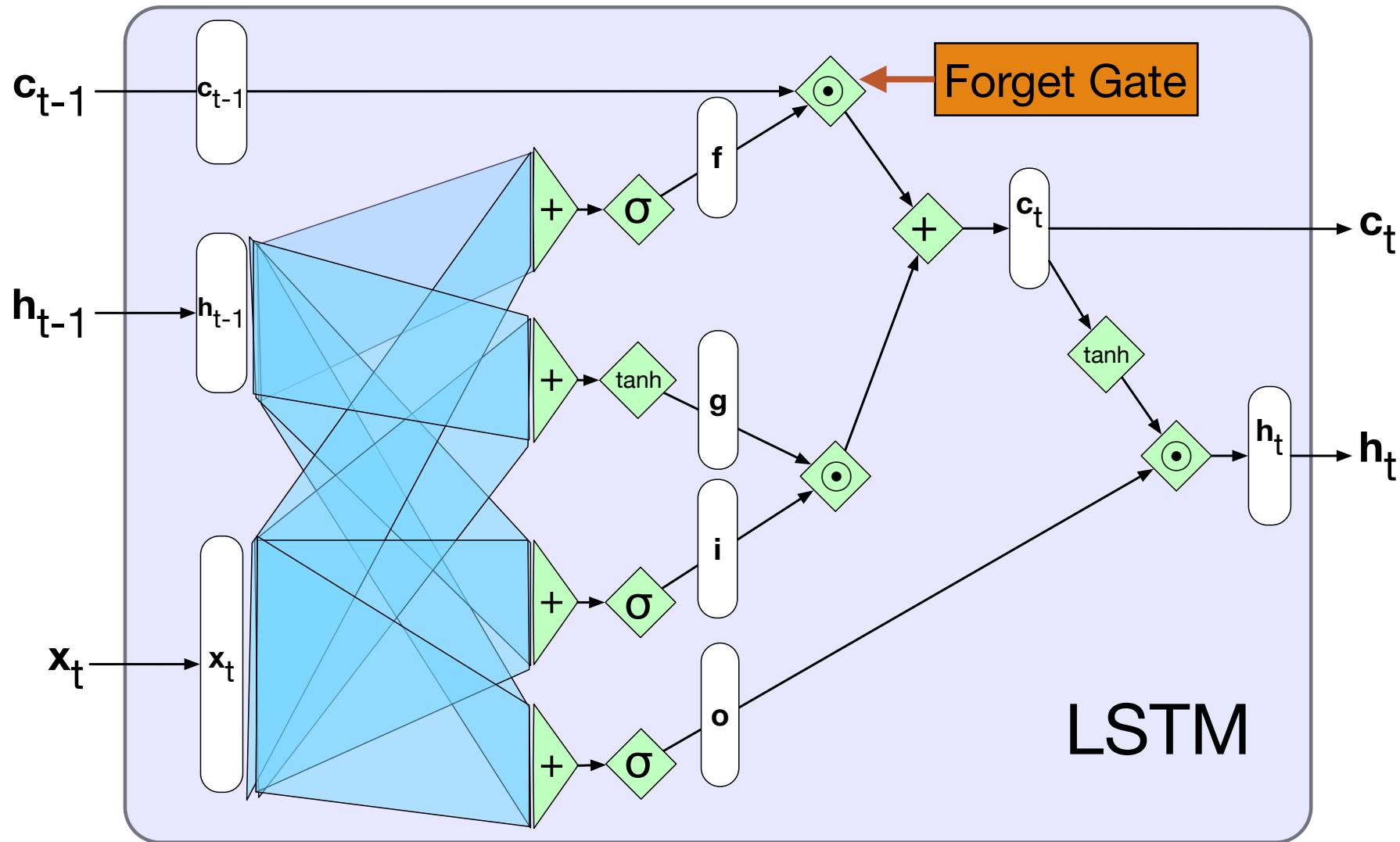


(c)  
LSTM unit

# The LSTM



# The LSTM



# Forget gate

**Its role :** delete information from the context that is no longer needed.

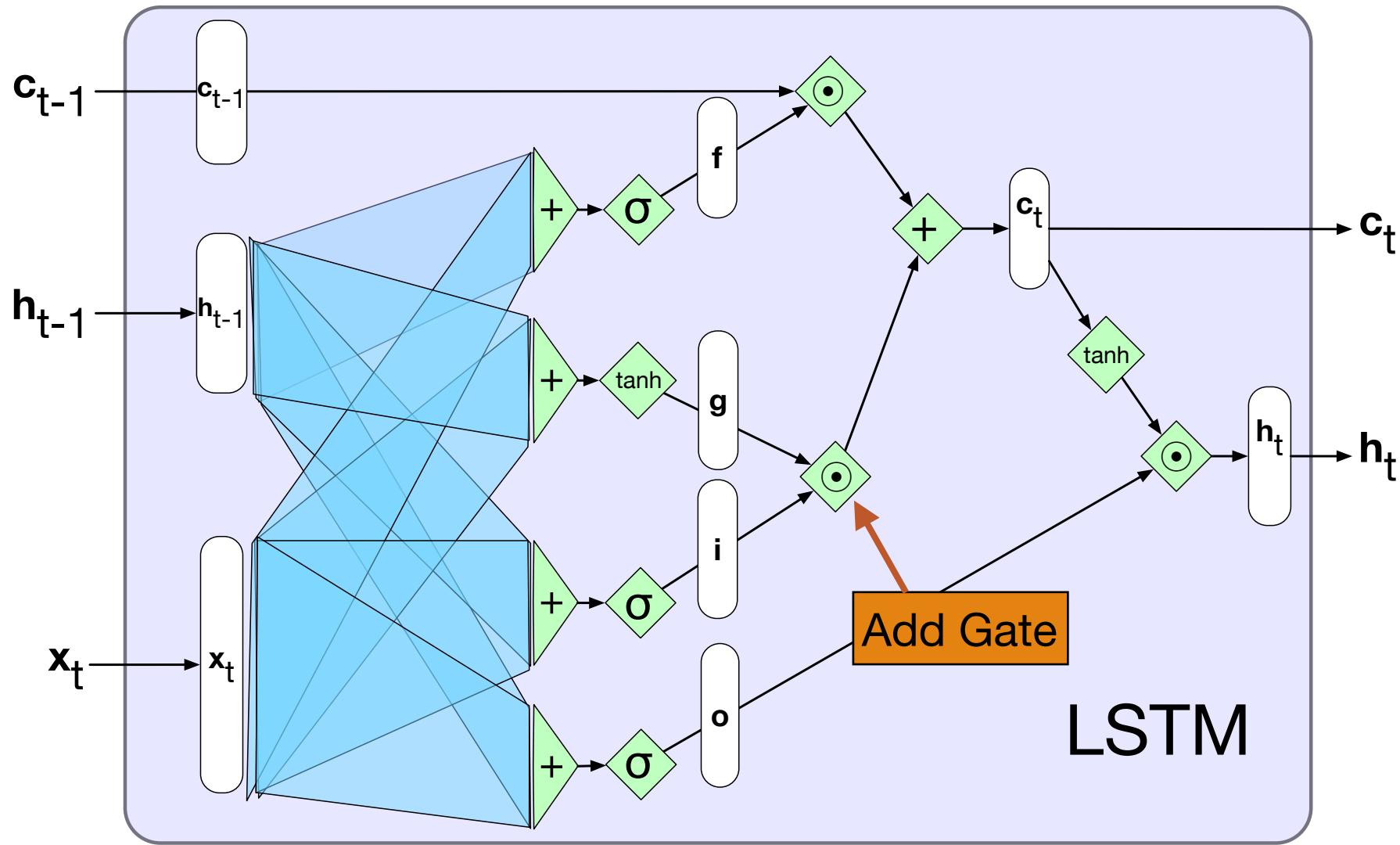
Computes weighted sum of the previous hidden states and current input and passes through sigmoid activation.

This mask is then multiplied element-wise with context vector to remove info from context that is no longer required.

$$\mathbf{f}_t = \sigma(\mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t)$$

$$\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t$$

# The LSTM



# Add gate

**Its role:** selecting information to add to current context.

First calculate regular information passing (as with SRN):

$$\mathbf{g}_t = \tanh(\mathbf{U}_g \mathbf{h}_{t-1} + \mathbf{W}_g \mathbf{x}_t)$$

Then, select new info to add to context by multiplying element wise with sigmoid output over current state output.

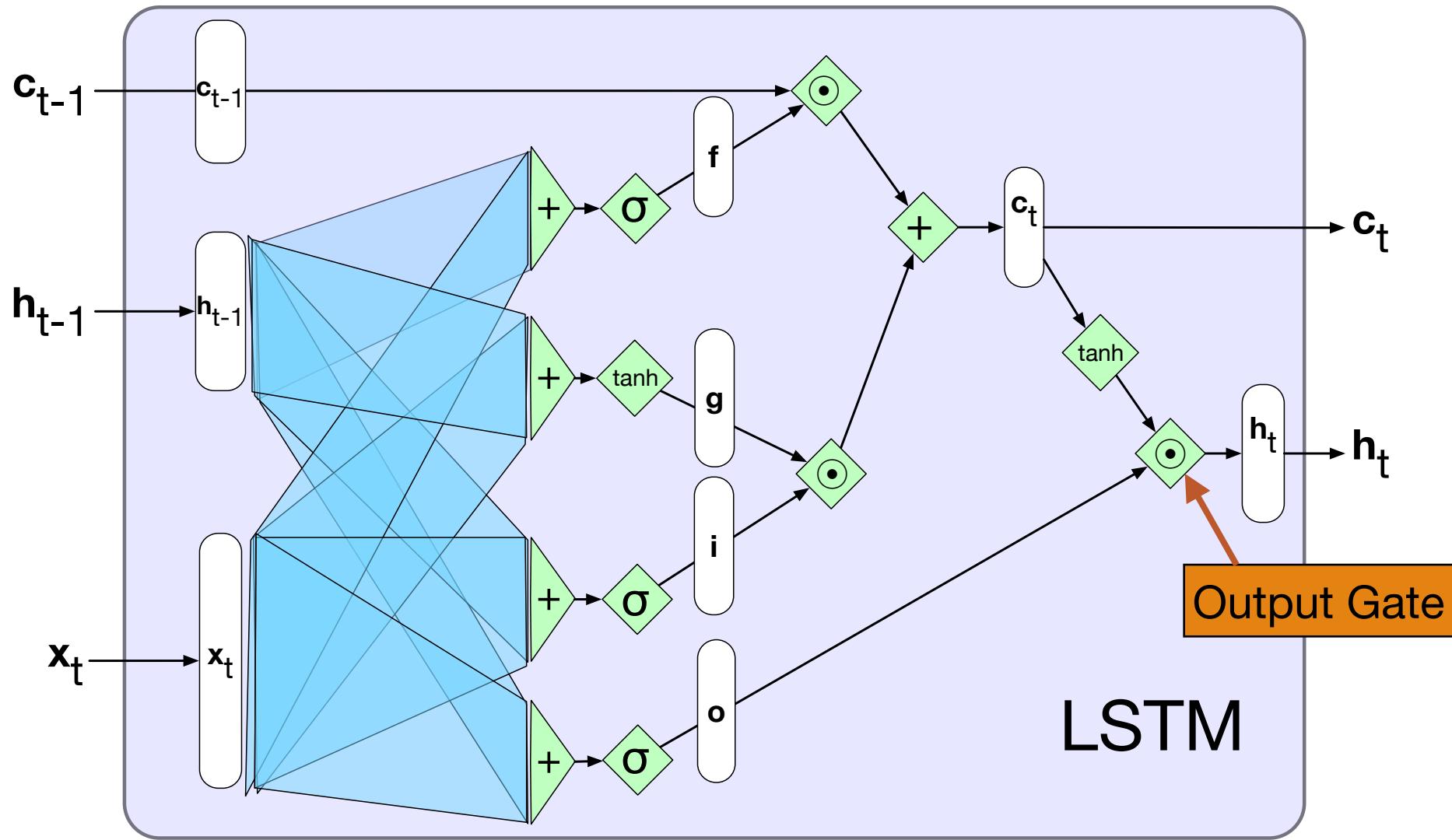
$$\mathbf{i}_t = \sigma(\mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{W}_i \mathbf{x}_t)$$

$$\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t$$

Add this to the modified context vector to get our new context vector.

$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t$$

# The LSTM



# Output gate

**Its role:** Decide what information is required for the current hidden state (as opposed to what information needs to be preserved for future decisions).

Multiplies element-wise the current state sigmoid output by new context.

$$\mathbf{o}_t = \sigma(\mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{W}_o \mathbf{x}_t)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

# LSTMs for natural language data

Because natural language has many long distance dependencies, LSTMs are the recurrent neural network of choice for language modeling.



**[15 minute break]**

# **Working with RNN models!**

**Team up!**

**Open exercises/week 6 in your course folder and start writing/running code!**