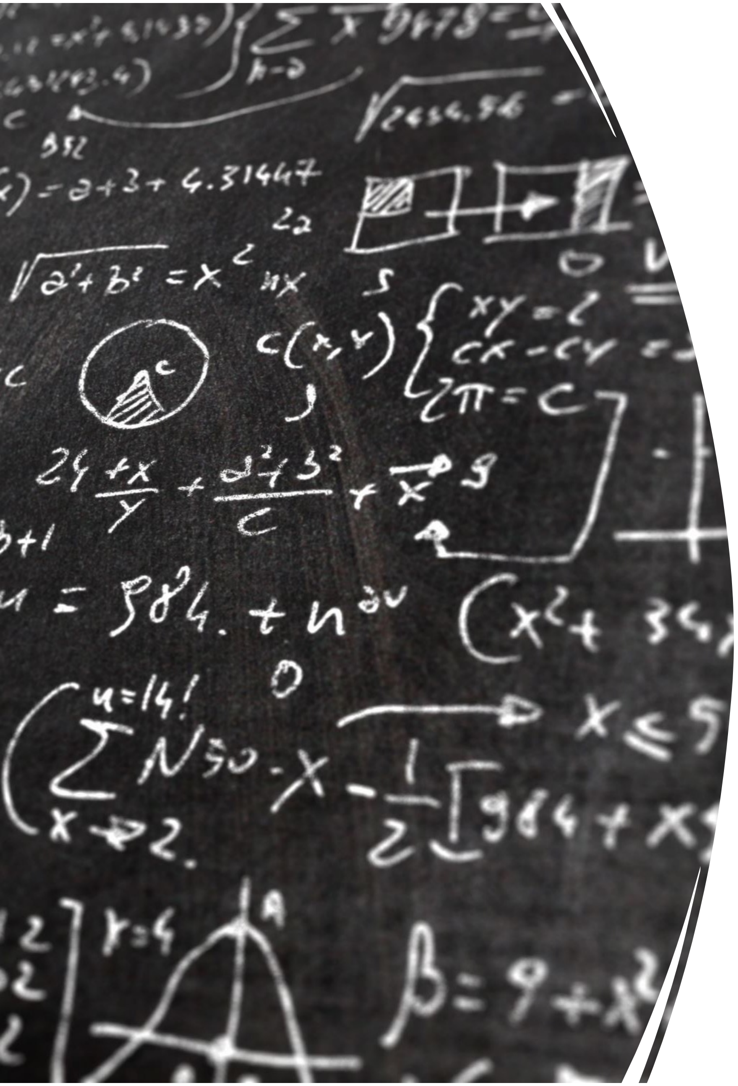


CS 383 – Computational Text Analysis

Lecture 13
Recurrent Neural Networks

Adam Poliak

03/01/2023

Slides adapted from Jordan Boyd-Graber, Daniel Khashabi

Announcements

- HW04
  - Due Friday (shorter than the previous ones)
- Reading 05
  - CTA/TADA/CSS papers using Word Embeddings
  - Look at piazza for deadline – Wednesday after spring break
- Reading 06
  - Will be back to Mondays
- Office hours this week:
  - Email me to schedule this week
- Final Project Ideation

  250 write up – what idea do you have, who are you working with

  Due before Spring break

Outline

Recap - Deep Averaging Neural Network

RNNs

Machine Learning in a nutshell

In a ML model, what are we training?

- **Parameters!**

How do we train parameters in supervised learning?

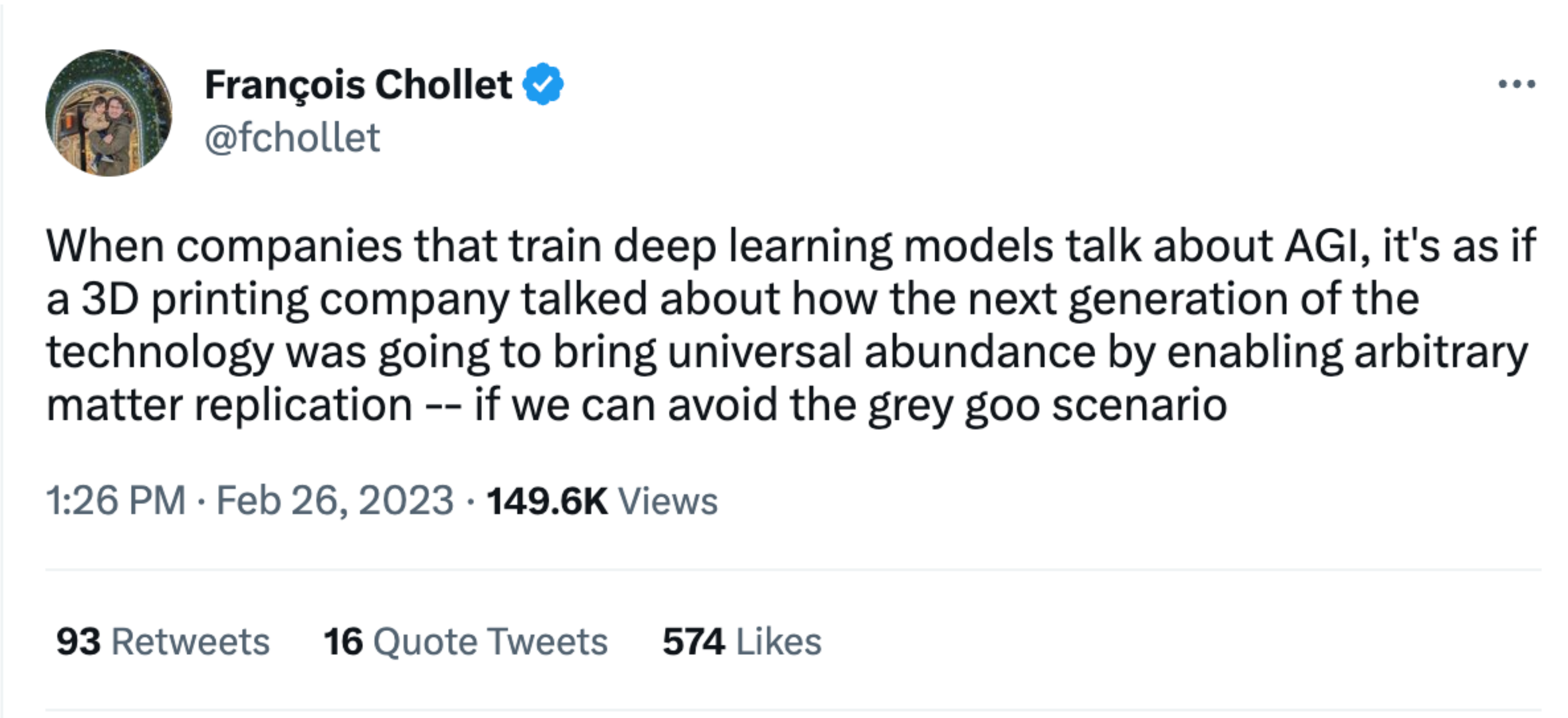
train parameters == figure out values for the parameters

- Update weights by using them to make predictions and seeing **how far off our predictions** are
  - **Loss function!**

Algorithm to learn weights?

- **SGD**
- Others exist but not covering them

Classify a tweet as viral or not



François Chollet
@fchollet

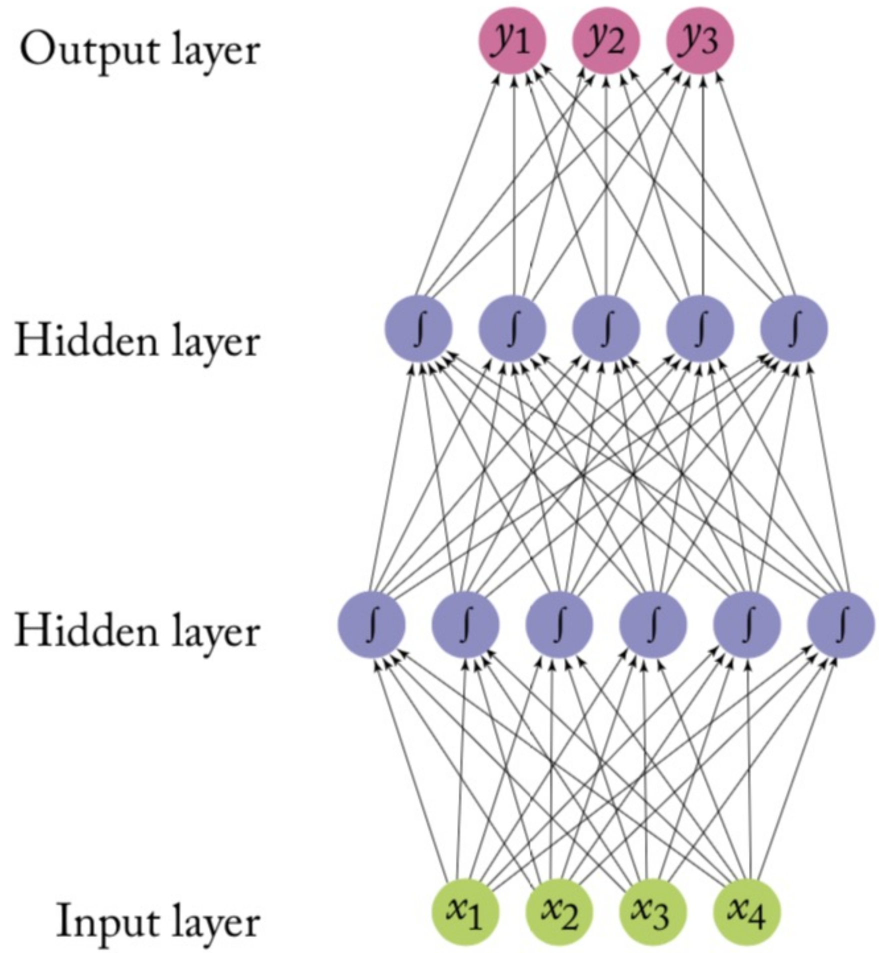
When companies that train deep learning models talk about AGI, it's as if a 3D printing company talked about how the next generation of the technology was going to bring universal abundance by enabling arbitrary matter replication -- if we can avoid the grey goo scenario

1:26 PM · Feb 26, 2023 · **149.6K** Views

93 Retweets **16** Quote Tweets **574** Likes

Classify a tweet as viral or not

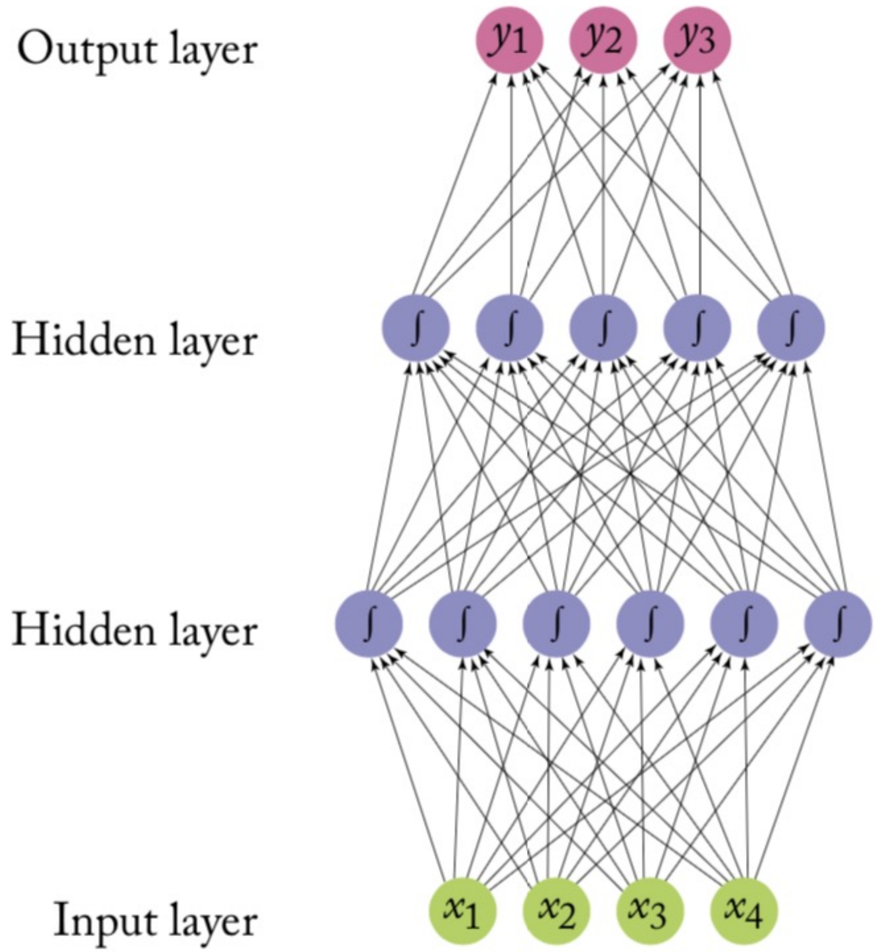
Classify a tweet as viral or not

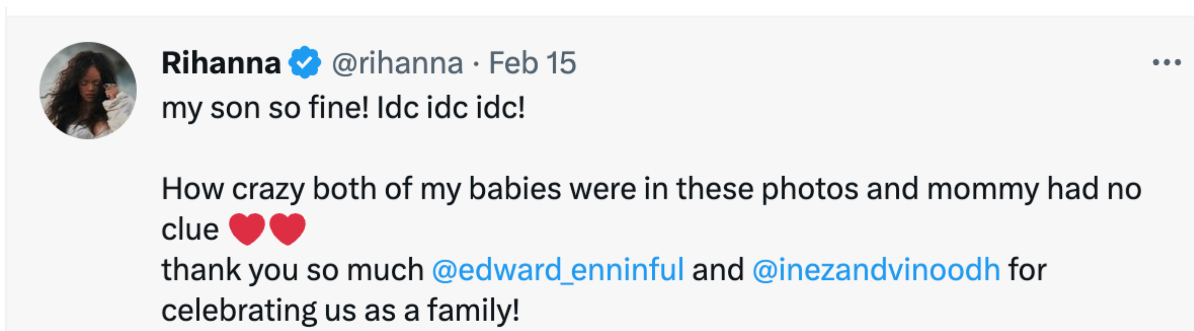


Taylor Swift @taylorswift13 · Jan 27

The Lavender Haze video is out now. There is lots of lavender. There is lots of haze. There is my incredible costar @laith_ashley who I absolutely adored working with.

Classify a tweet as viral or not





FFN’s issues

Input size is fixed, but the length of text (or a document) is variable

Solutions:

1. Create a fixed length representation
2. Recurrent Neural Networks

Deep Averaging Network

Represent each document as a continuous bag of words, averaging the word embeddings

$$x = w_1, w_2, \ldots w_n$$

$$z_0 = CBOW(w_1, w_2, \ldots w_n). CBOW = \sum_i E[w_i]$$

$$\hat{y} = MLP(z_o)$$

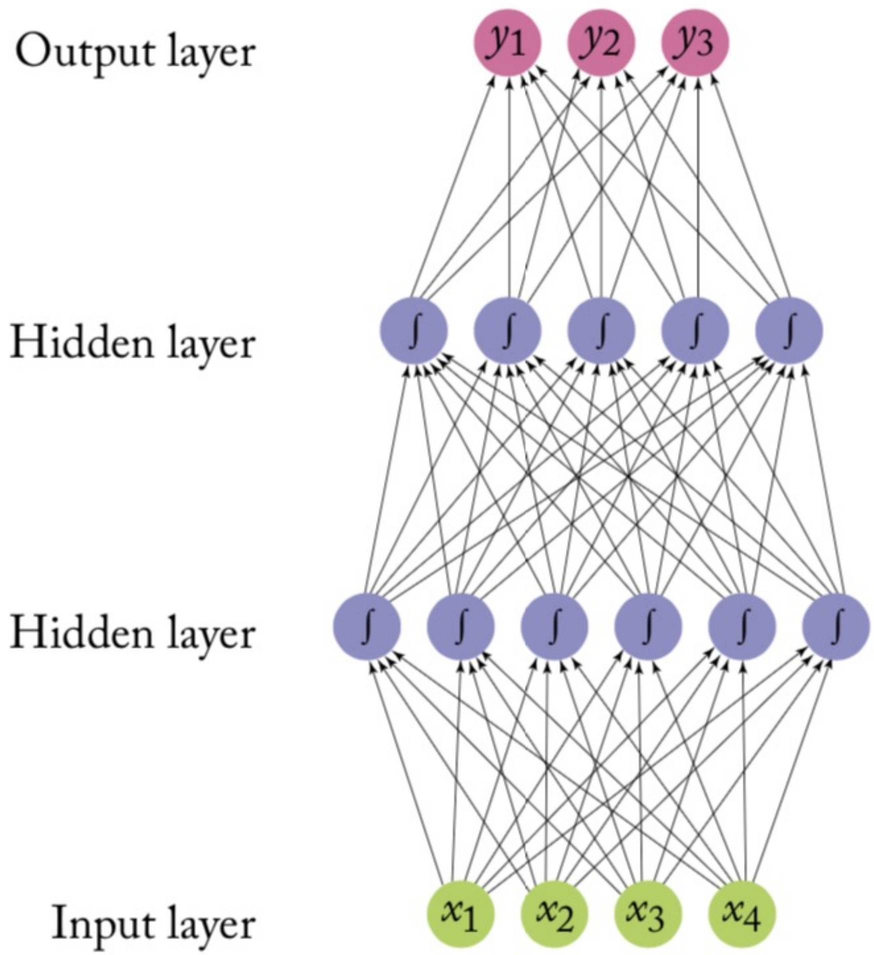
Multilayer Perceptron

Feed-forward NN

$$MLP_1 = g(xW_1 + b_1)W_2 + b_2$$

$$MLP_2 = g(g(xW_1 + b_1)W_2 + b_2)\, W_3 + b_3$$

MLP_2



Output layer
y_1 y_2 y_3
Hidden layer
Hidden layer
Input layer
x_1 x_2 x_3 x_4

Deep Averaging Network

Represent each document as a continous bag of words, i.e. averaging the word embeddings

$x = w_1, w_2, \dots w_n$

$z_0 = CBOW(w_1, w_2, \dots w_n). CBOW = \sum_i E[w_i]$

$\hat{y} = MLP(z_o)$

Homework after spring break

FFN’s issues

Input size is fixed, but the length of text (or a document) is variable

Solutions:

1. Create a fixed length representation
2. **Recurrent Neural Networks**

RNN - motivation

How can we model a **long** (possibly infinite) context using a finite **model?**

Recursion

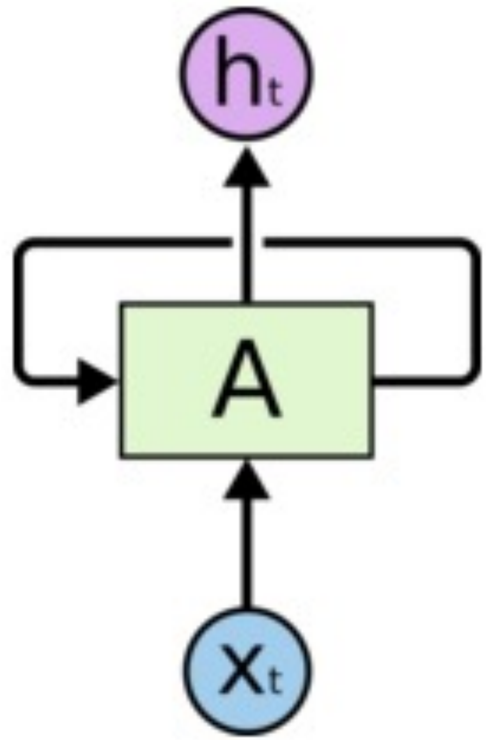
Recurrent Neural Networks are a family of NNs that learn sequential data via **recursive dynamics**

Recurrent Neural Network (RNN)

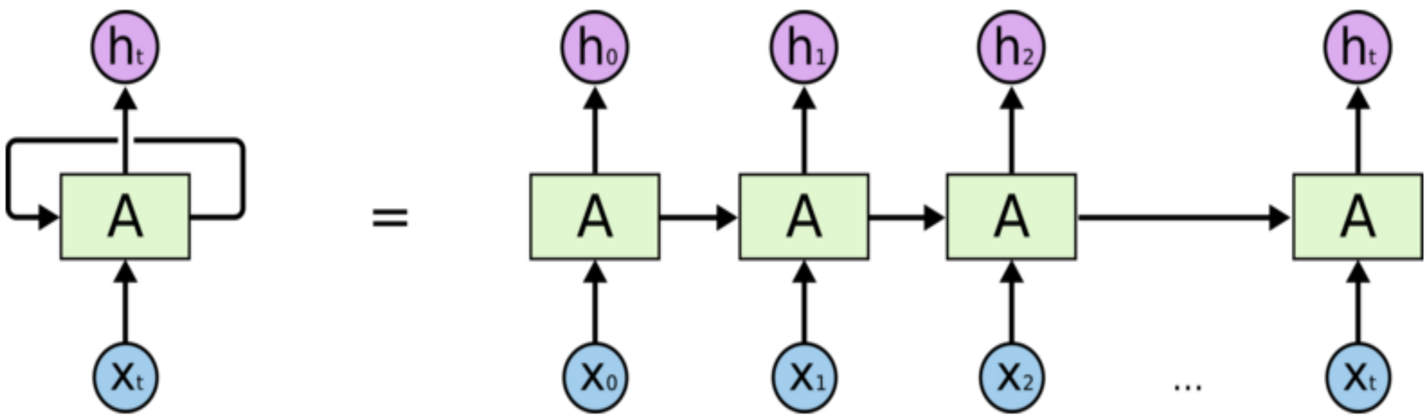
$h_t = f(h_{t-1}, x_t)$

In the diagram, $f(\ldots)$ looks at some input x_t and its previous hidden state h_{t-1} and outputs a revised state h_t.

A loop allows information to be passed from one step of the network to the next.



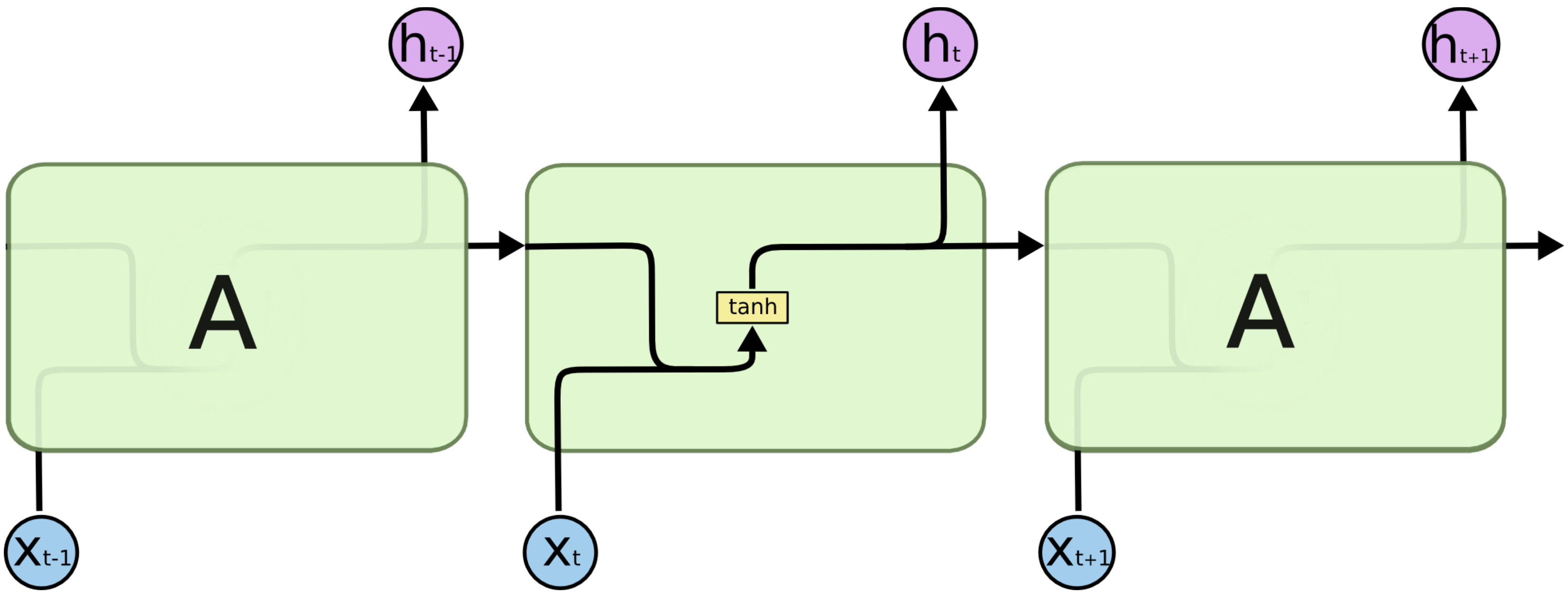
Unrolling an RNN



A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.

RNN internal

$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$



Updating weights in a RNN

$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$

$$\frac{\partial \tan(x)}{\partial x} = 1 - \tan(x)^2$$

So what variables do we need to update?

Our weights and biases

Updating weights in a RNN

$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$

$$\frac{\partial \tan(x)}{\partial x} = 1 - \tan(x)^2$$

$$\frac{\partial L}{\partial \theta} = \begin{bmatrix} \frac{\partial L}{\partial W_{ih}} \\ \frac{\partial L}{\partial b_{ih}} \\ \frac{\partial L}{\partial W_{ih}} \\ \frac{\partial L}{\partial b_{ih}} \end{bmatrix}$$

Updating weights in a RNN

$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$

Lets let this be ***g***

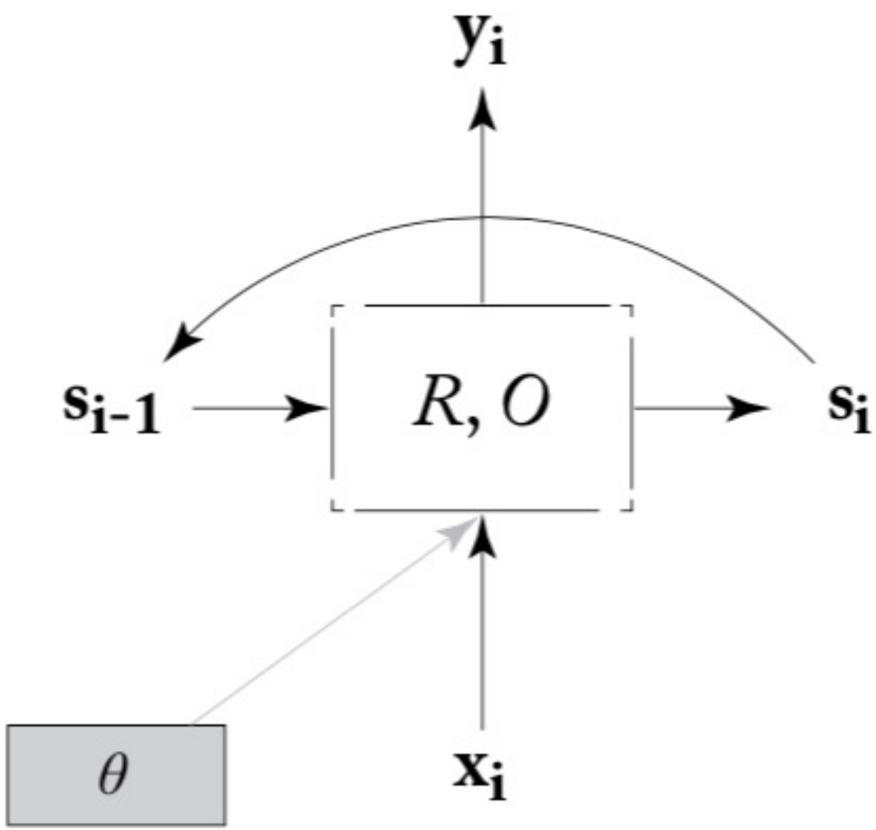
$$\frac{\partial L}{\partial \theta} = \begin{bmatrix} \frac{\partial L}{\partial W_{ih}} \\ \frac{\partial L}{\partial b_{ih}} \\ \frac{\partial L}{\partial W_{ih}} \\ \frac{\partial L}{\partial b_{ih}} \end{bmatrix} = \begin{bmatrix} 1 - \tan(g)\, x_t \\ 1 - \tan(g) \\ 1 - \tan(g)\, h_{t-1} \\ 1 - \tan(g)\, x_t \end{bmatrix}$$

$$\frac{\partial \tan(x)}{\partial x} = 1 - \tan(x)^2$$

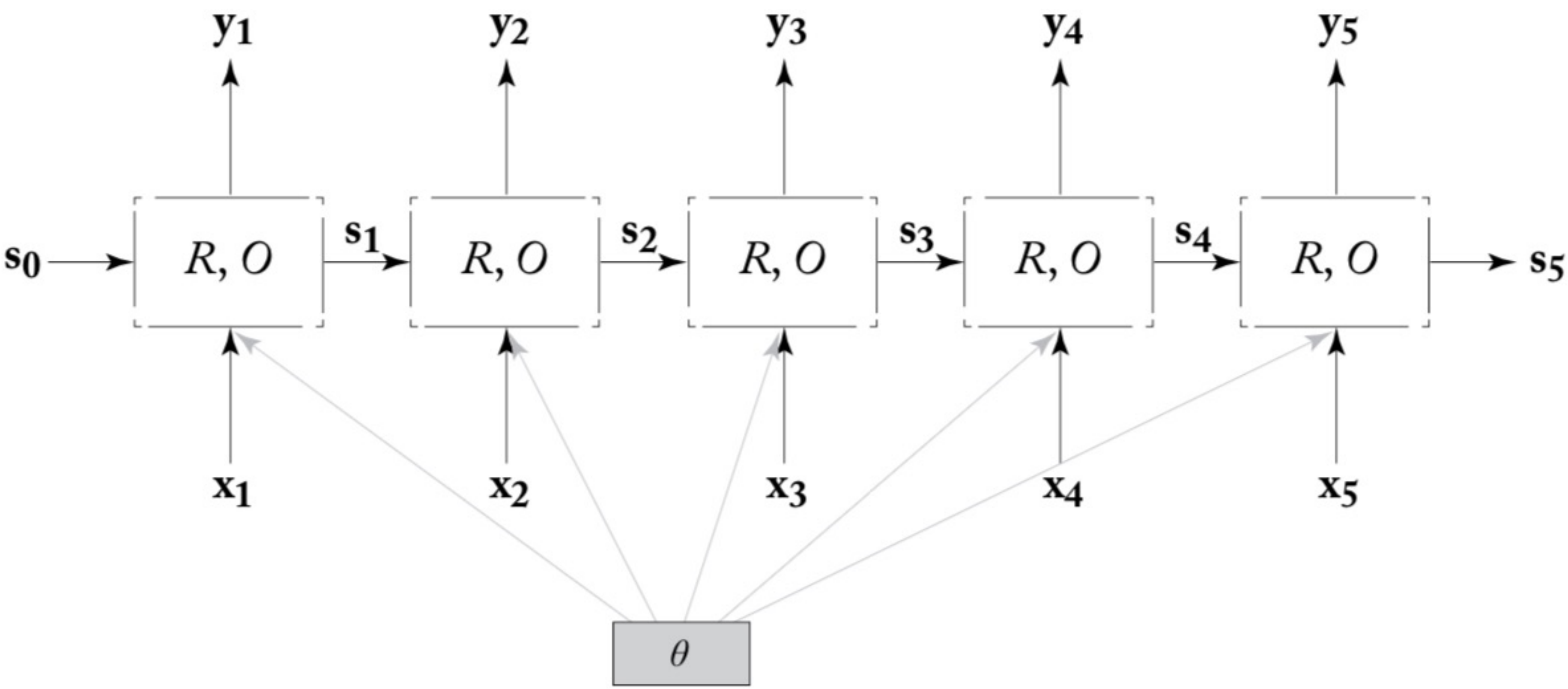
RNN cell

$$s_i = R(x_i, s_{i-1}, \theta)$$

$$\widehat{y_i} = O(s_i)$$

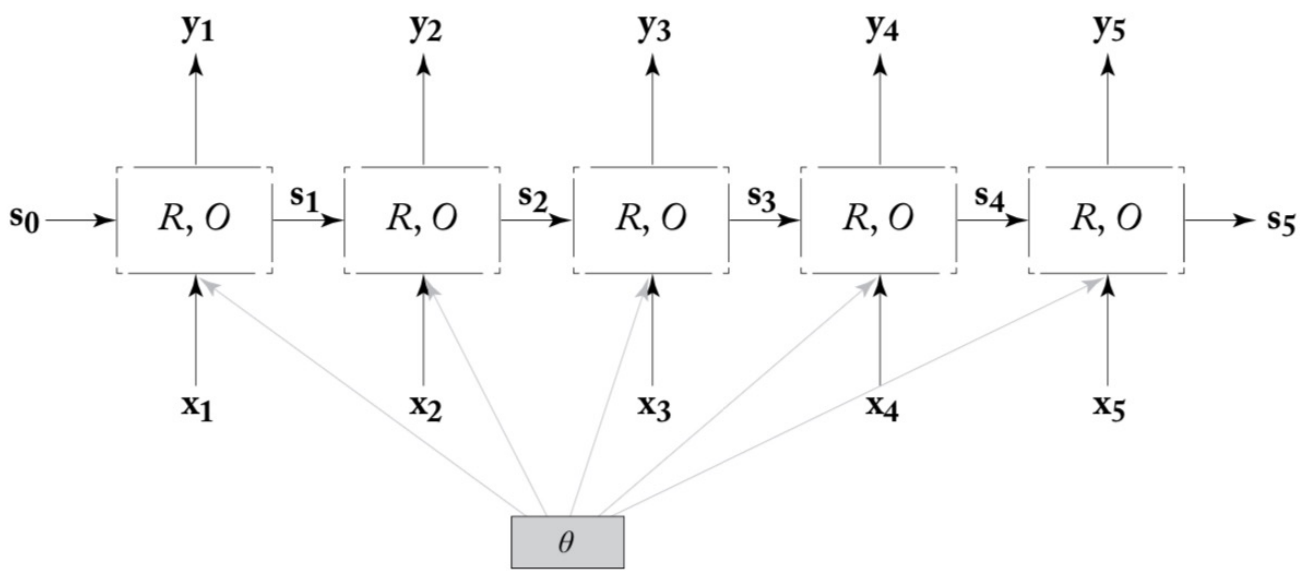


Unrolling RNN



Revisiting LM

$$P(x_t|x_{t-1}, x_{t-2}, \dots x_1)$$

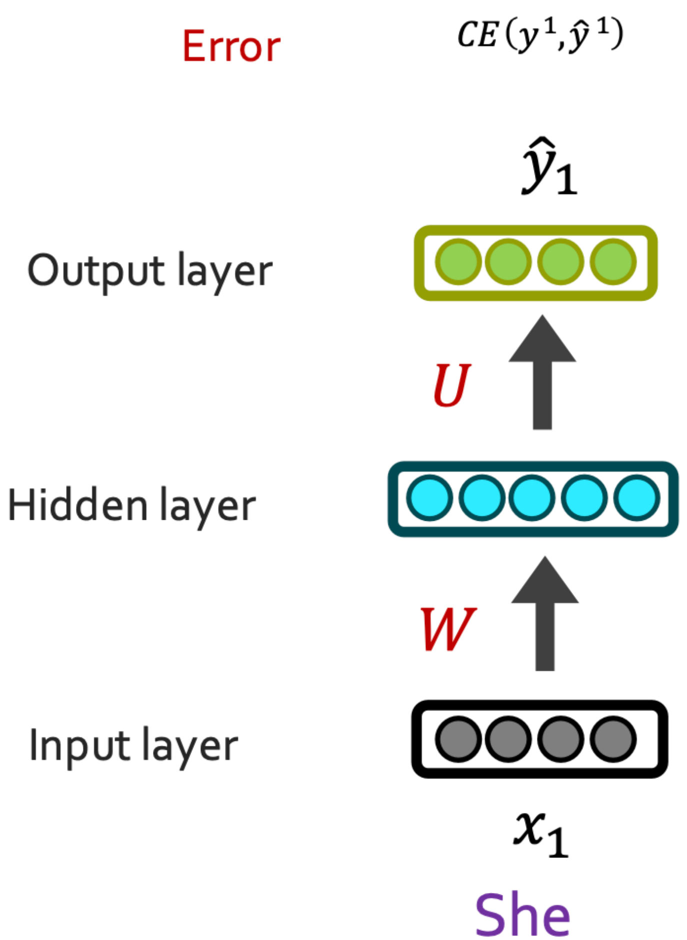


Pass in one word at a time

Compute probability over entire vocab by applying predictive head to last output

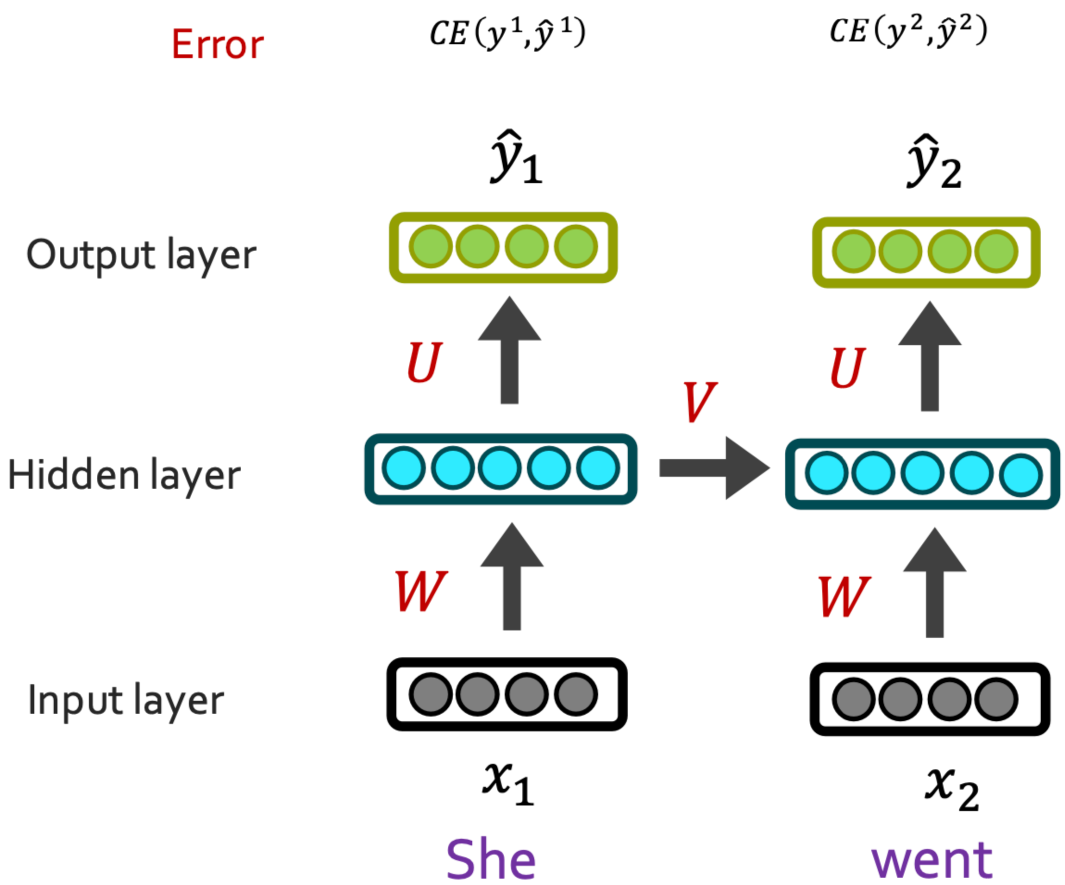
RNN: Forward

$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$



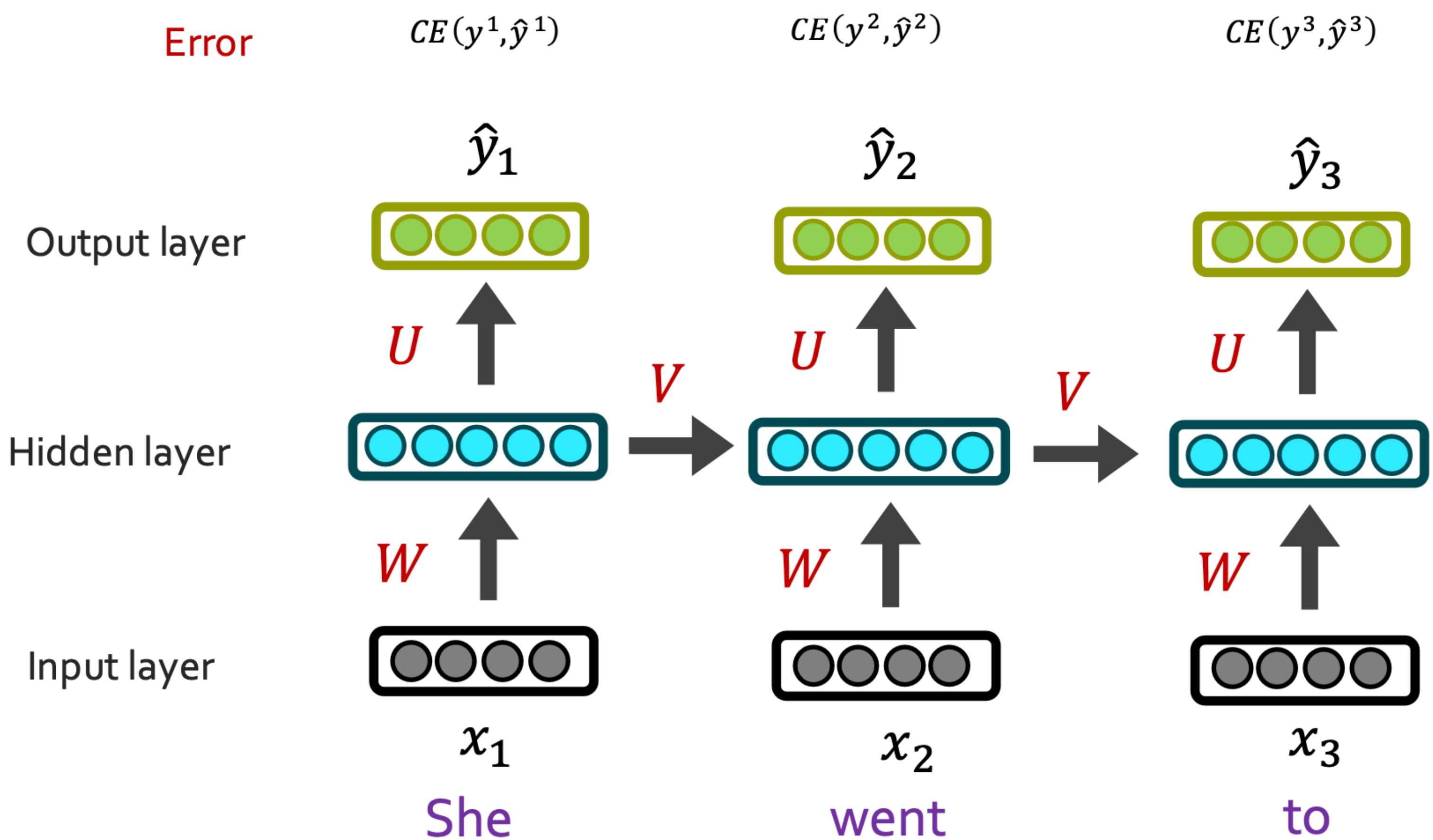
RNN: Forward

$$CE(y,\hat{y}) = -\sum_{w\in V} y_w \log \widehat{y_w}$$



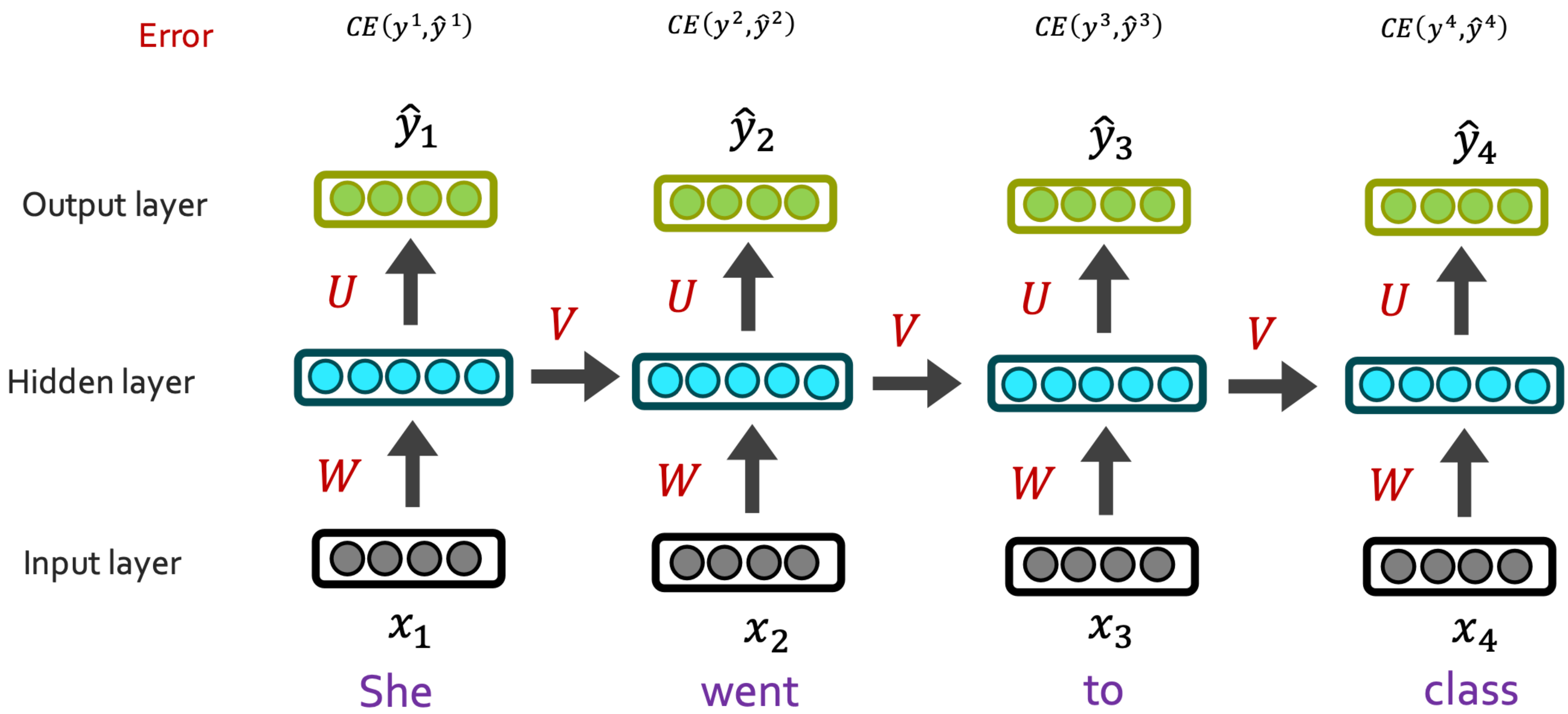
RNN: Forward

$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$



RNN: Forward

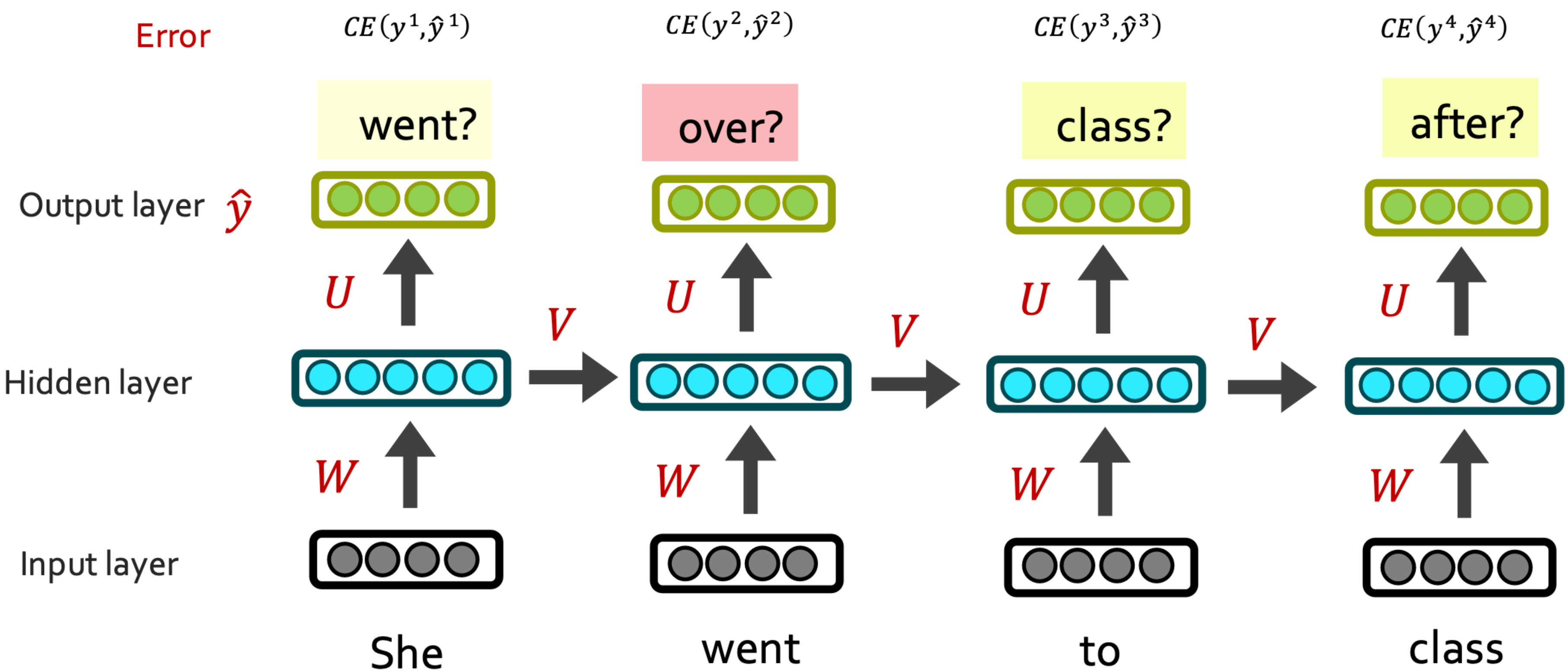
$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$



RNN: Forward

$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$

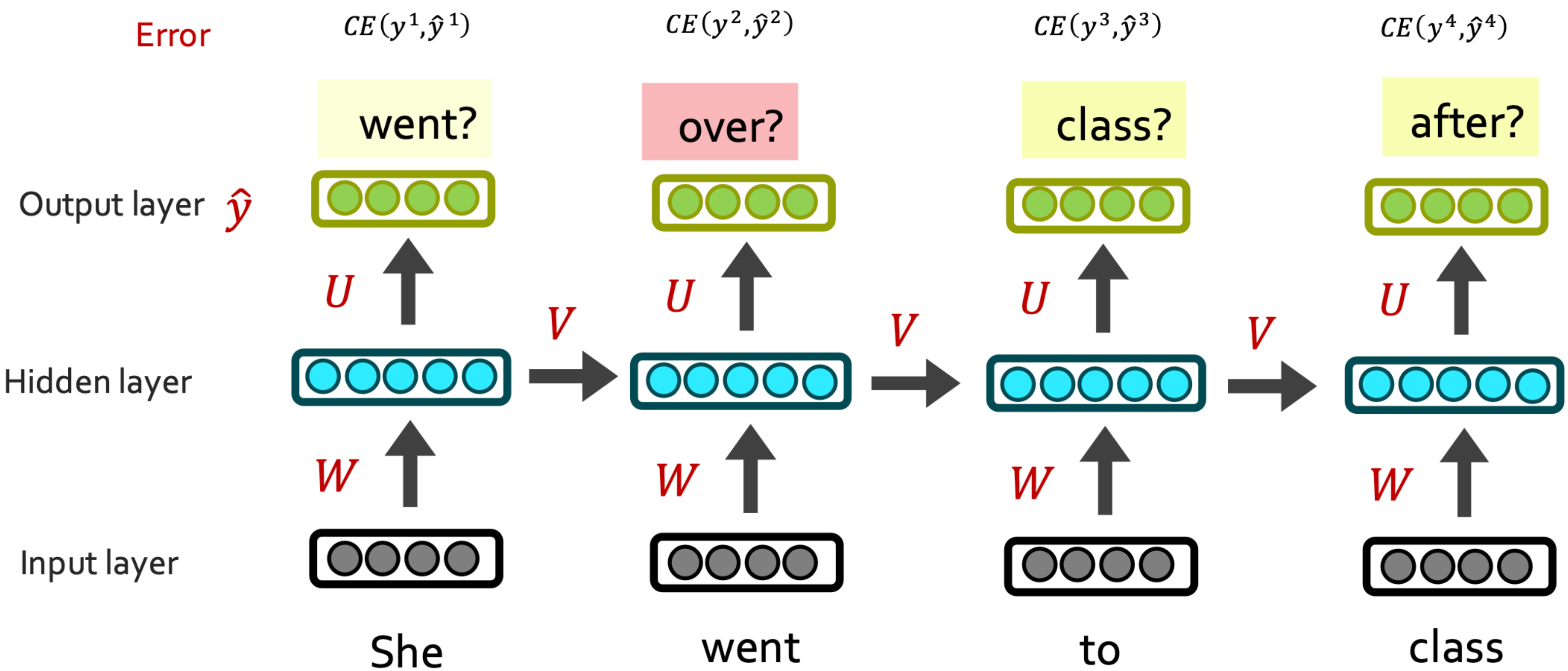
Loss is just averaging Cross-Entropy all predictions



RNN: Backwards

$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$

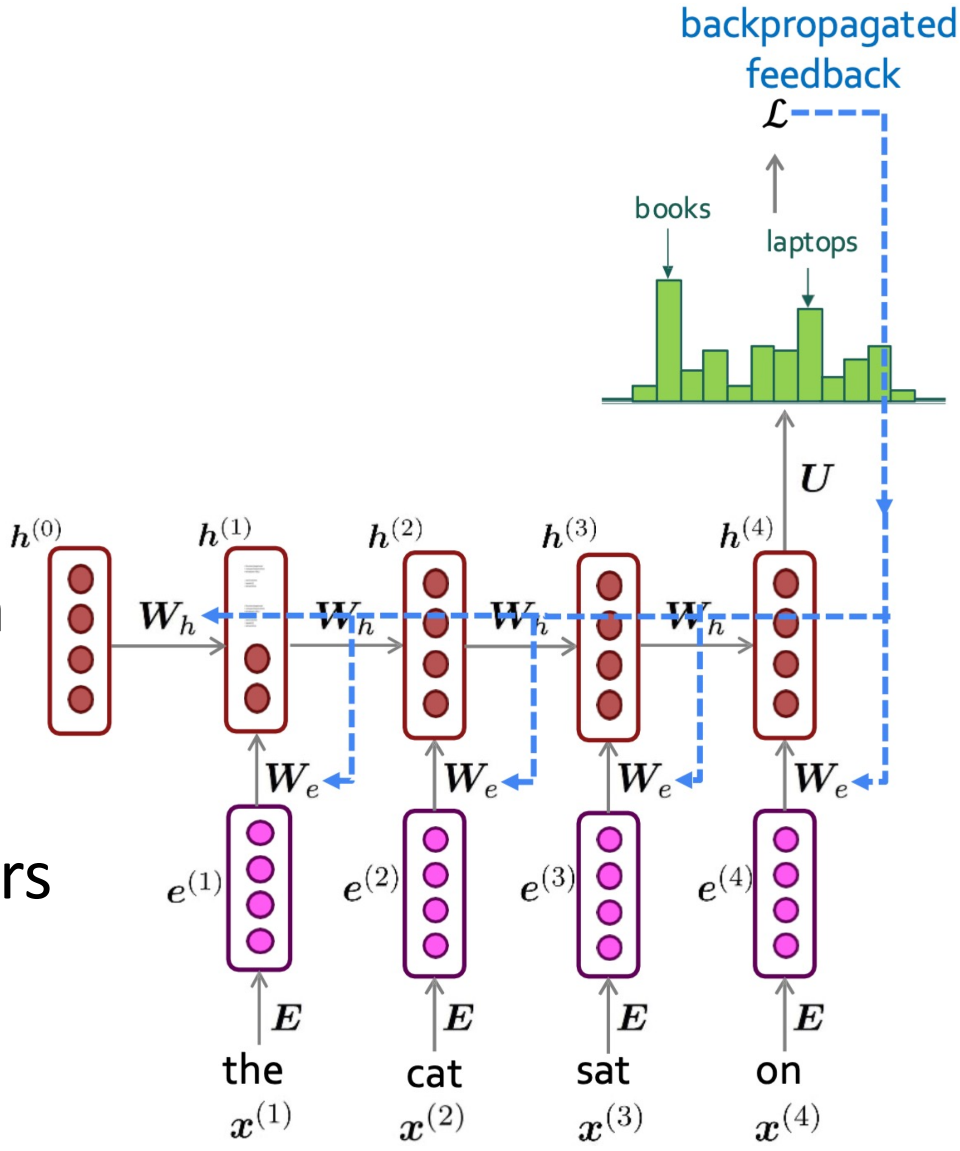
Compute the loss at the end, then propagate derivative of loss back to update the parameters



Training RNNs

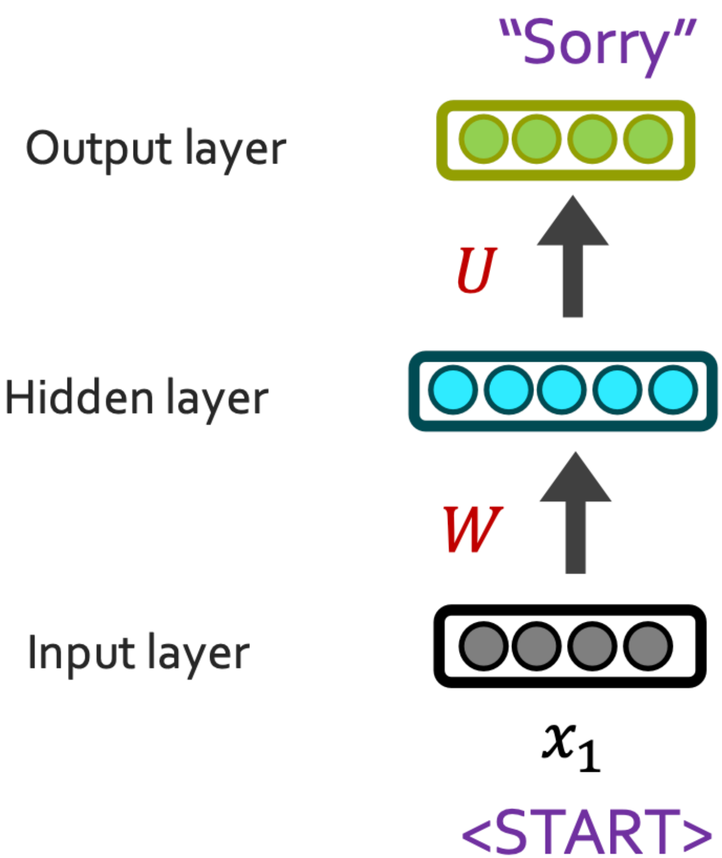
"Backprop over time"

1. Compute $\mathcal{L}$ for a batch of sentences
2. Compute gradients of $\mathcal{L}$ in respect to parameters
3. Repeat



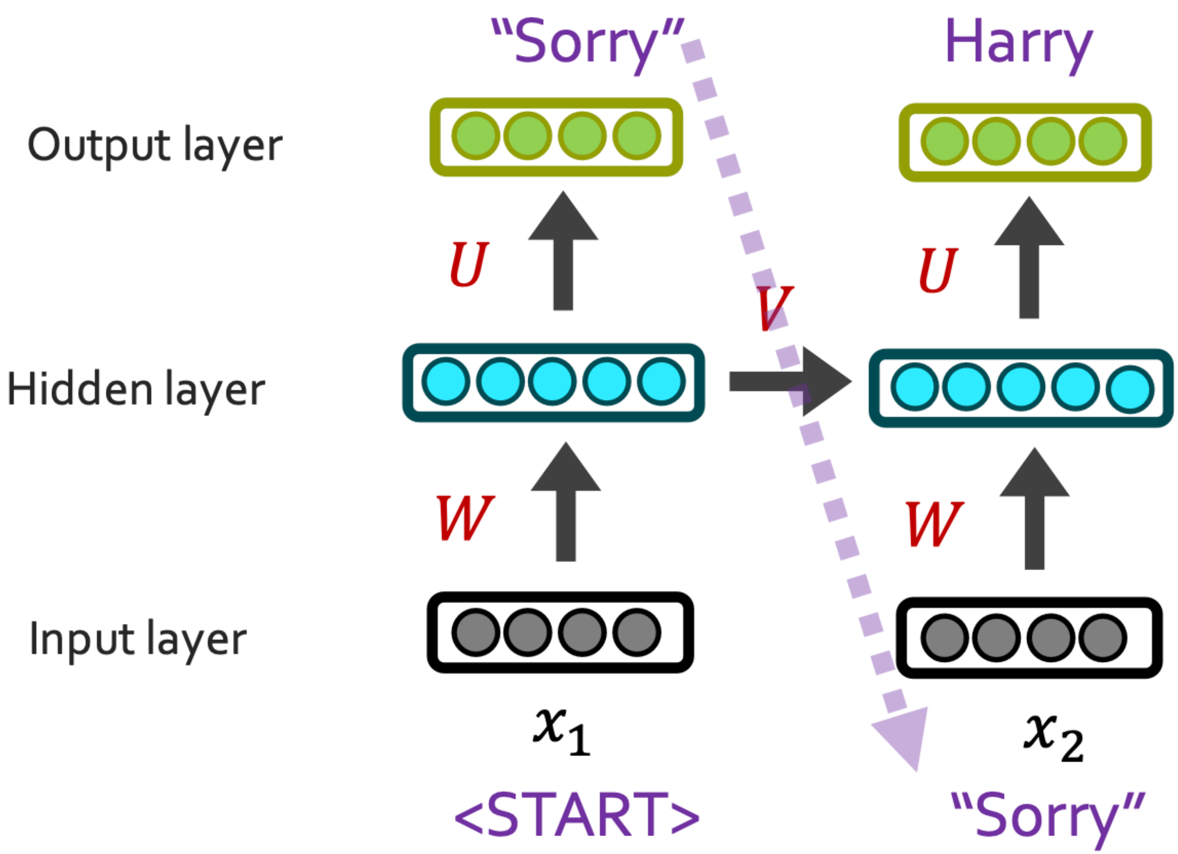
Generating with RNNs

Until we see a </s>, generate the most likely next word by sampling from previously predicted word



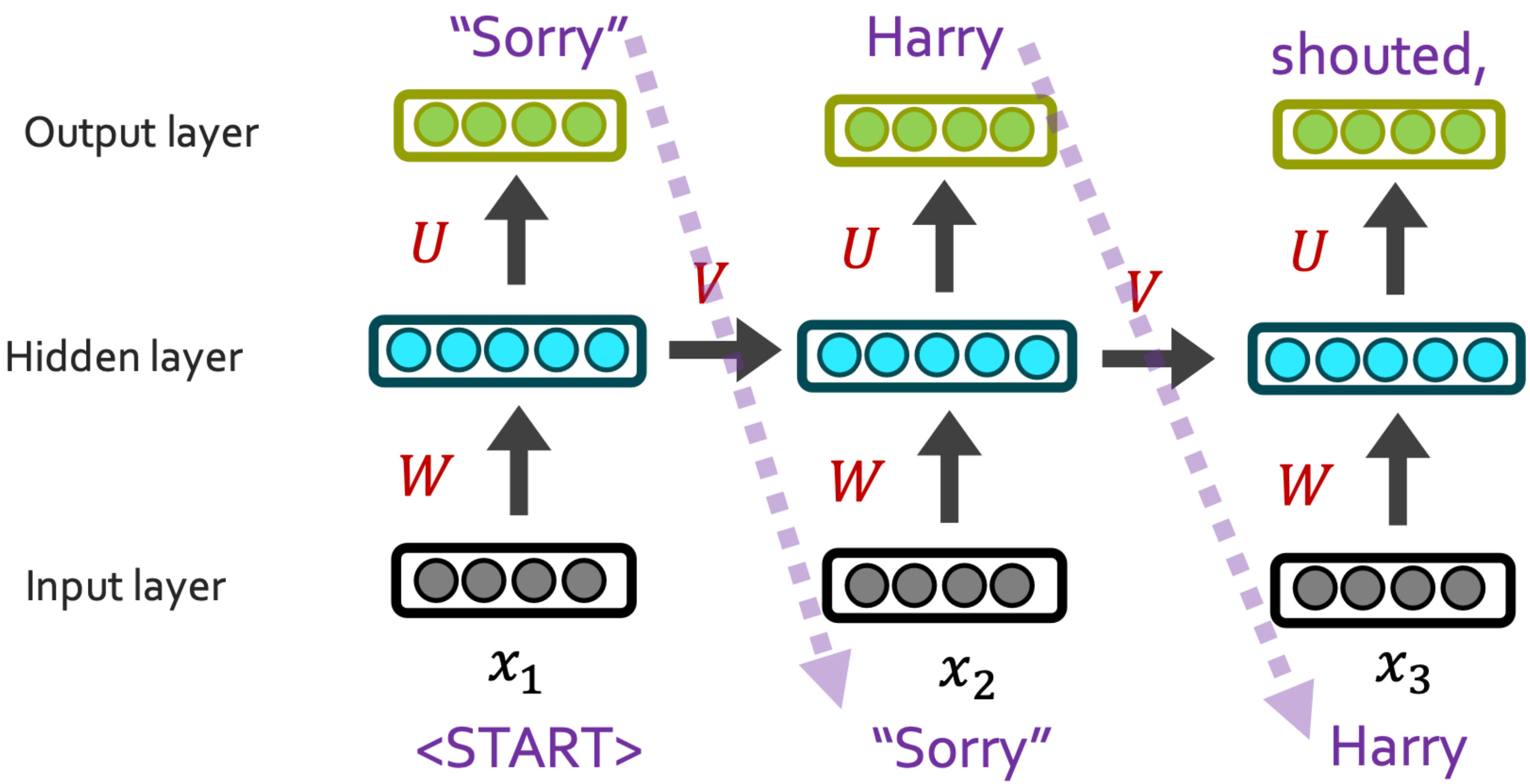
Generating with RNNs

Until we see a </s>, generate the most likely next word by sampling from previously predicted word



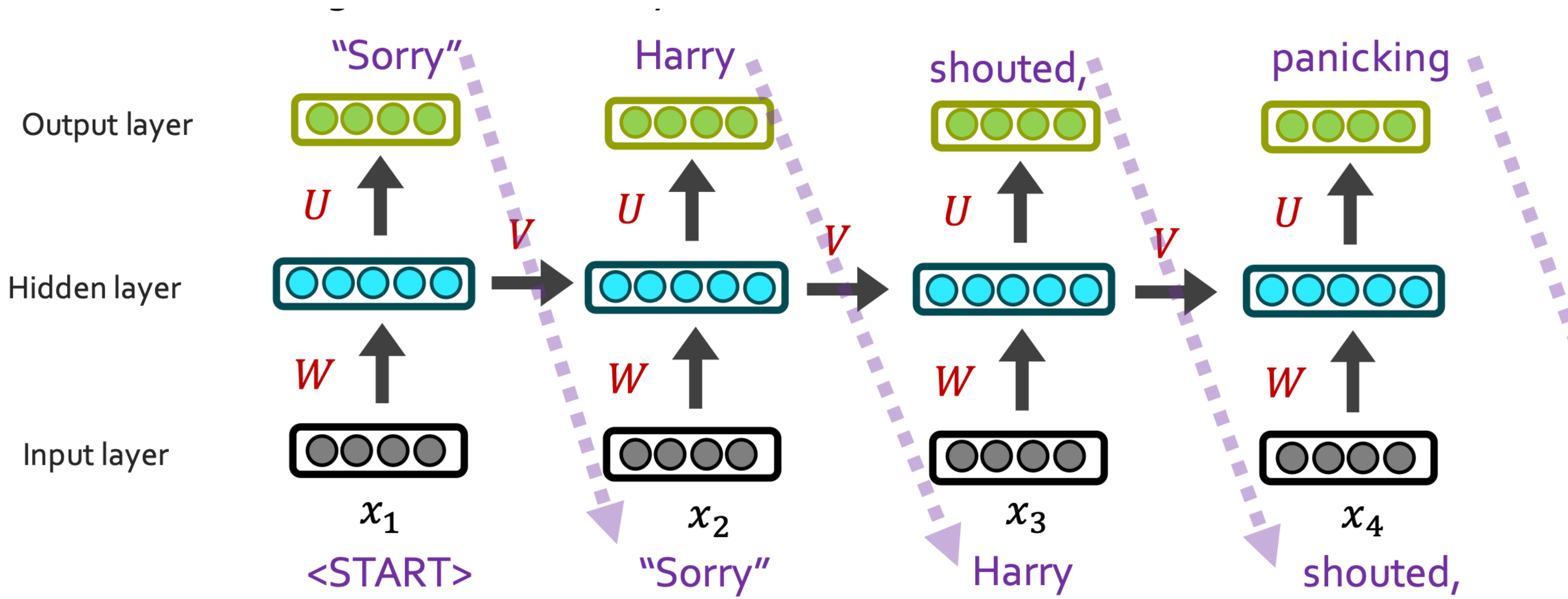
Generating with RNNs

Until we see a </s>, generate the most likely next word by sampling from previously predicted word



Generating with RNNs

Until we see a </s>, generate the most likely next word by sampling from previously predicted word



RNN’s: Pros and Cons

Pros:

- Model size doesn’t increase for longer inputs.
  - Reusing same parameters
- Computation can use information from many previous steps

Cons:

- Slow computation
- Can forget longer history/context
- Vanishing/exploding gradients

Vanishing/exploding gradient

Backpropagated loss multiplied at each layer

If |loss|> 1,

exponential growth -> ∞

If loss > 0 and <1

exponential decay -> 0

Solution – Gradient Clipping

If the gradient is greater than some threshold, scale it before updating weights

Pascan u et al. 2013
http://proceedings.mlr.press/v28/pascanu13.pdf

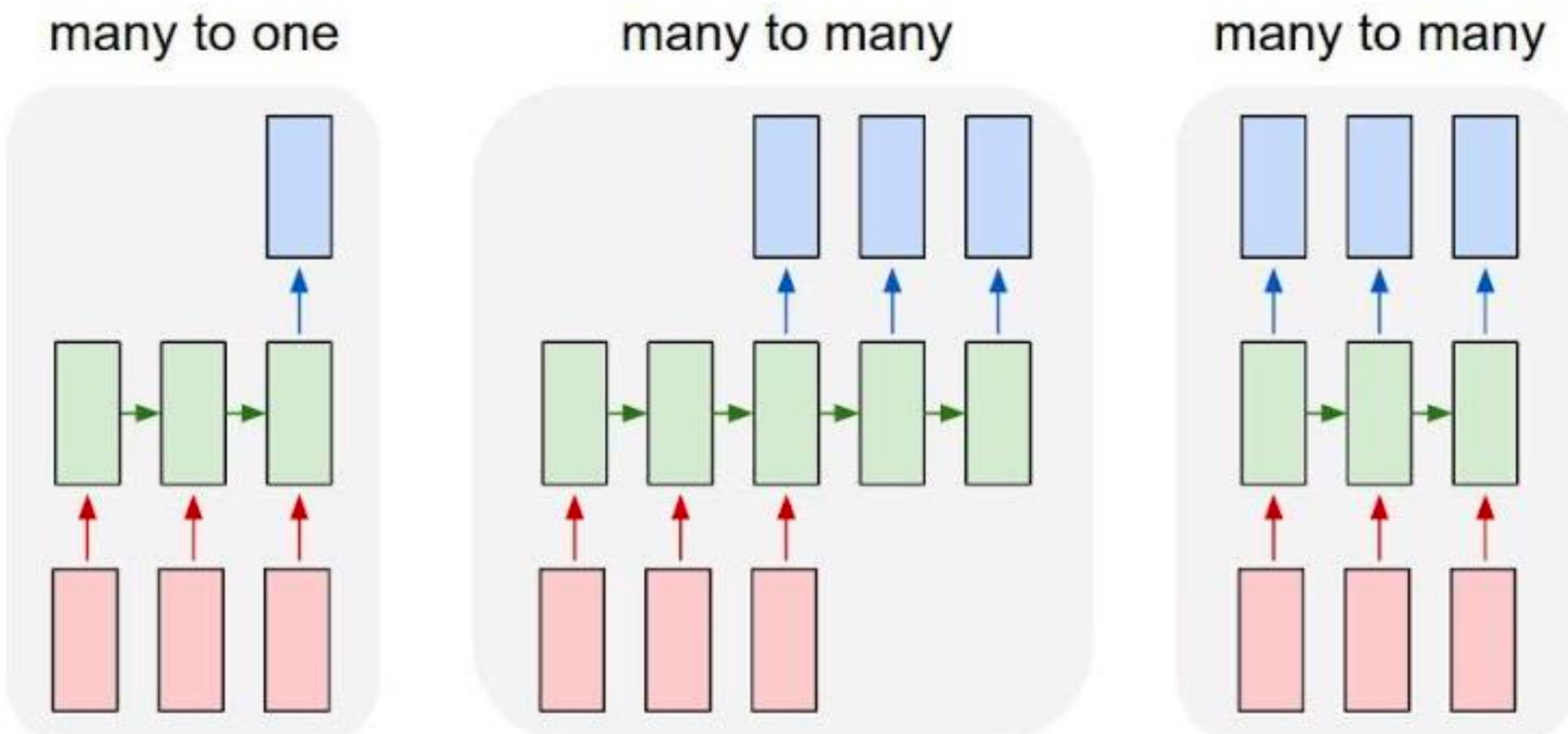
Intuition:

Take a step in the same direction, but smaller

Algorithm 1 Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
$\quad \hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$
end if

RNNs applied to other tasks



Text Classification

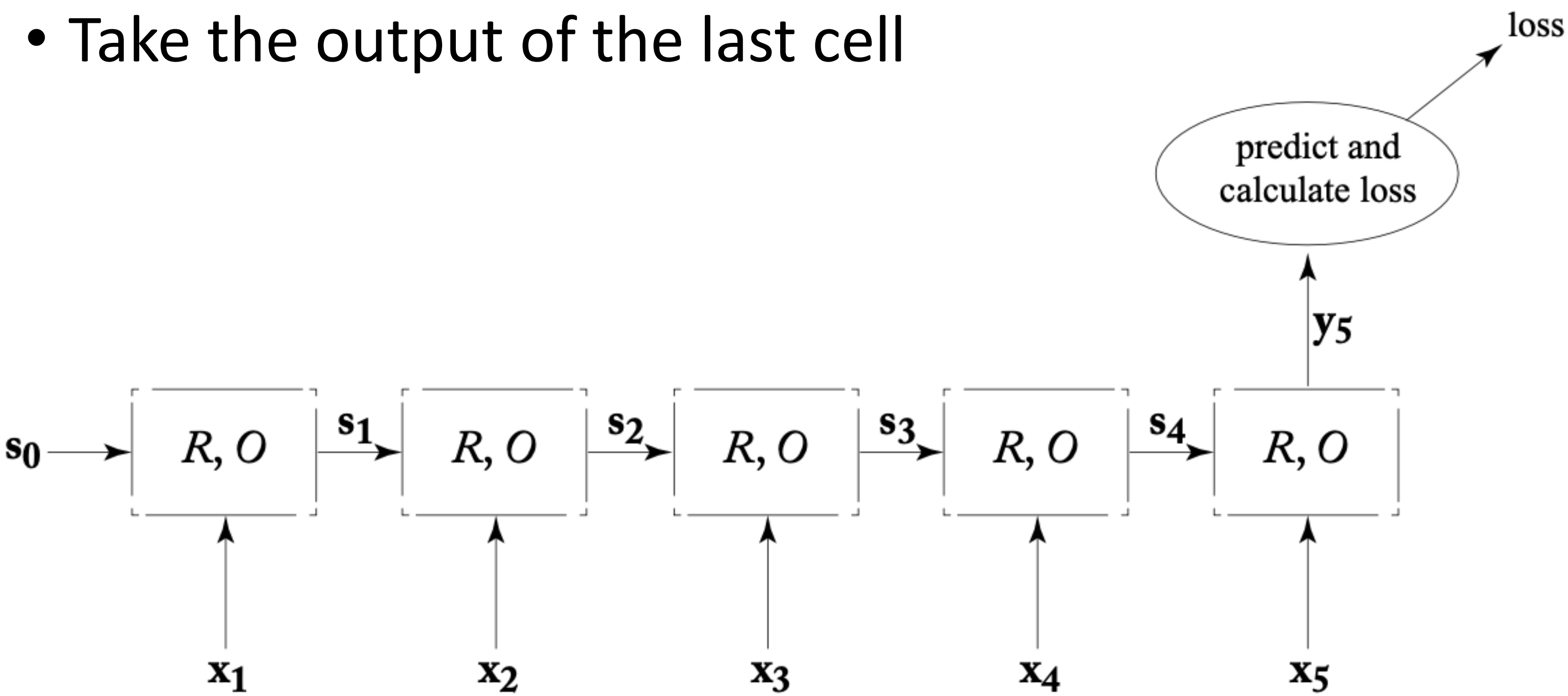
Language Modeling

POS Tags

Extracting representation from RNN layer

Acceptor

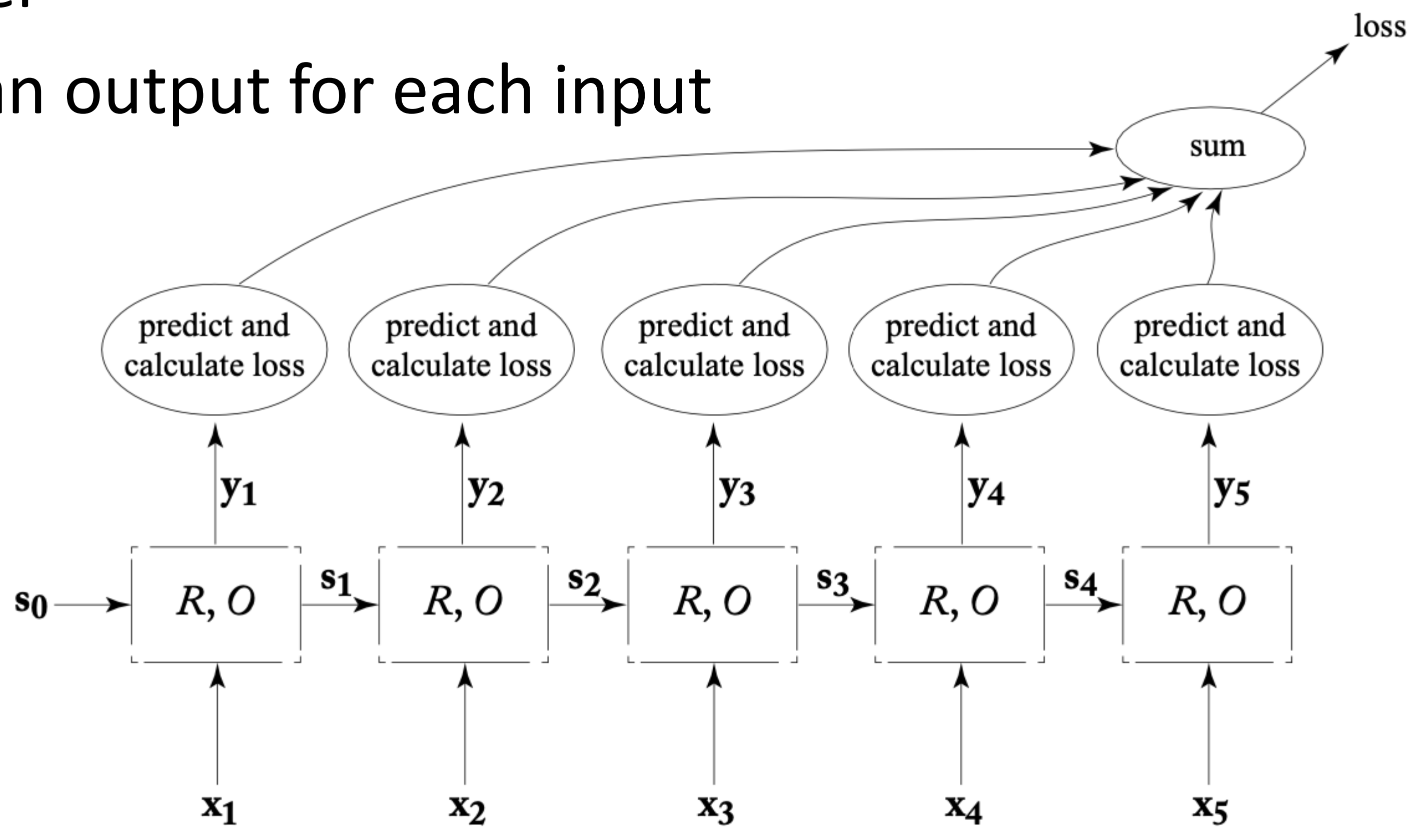
- Take the output of the last cell



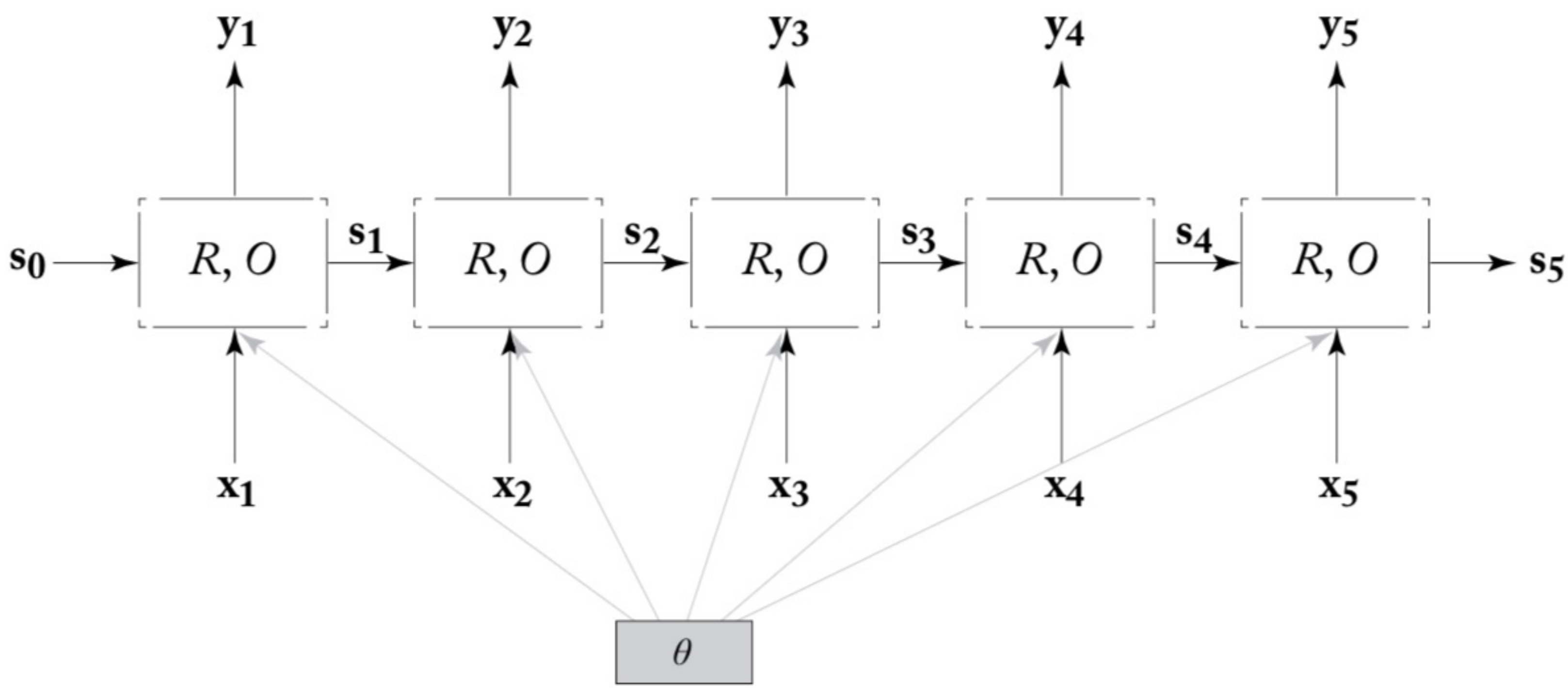
Extracting representation from RNN layer

Transducer

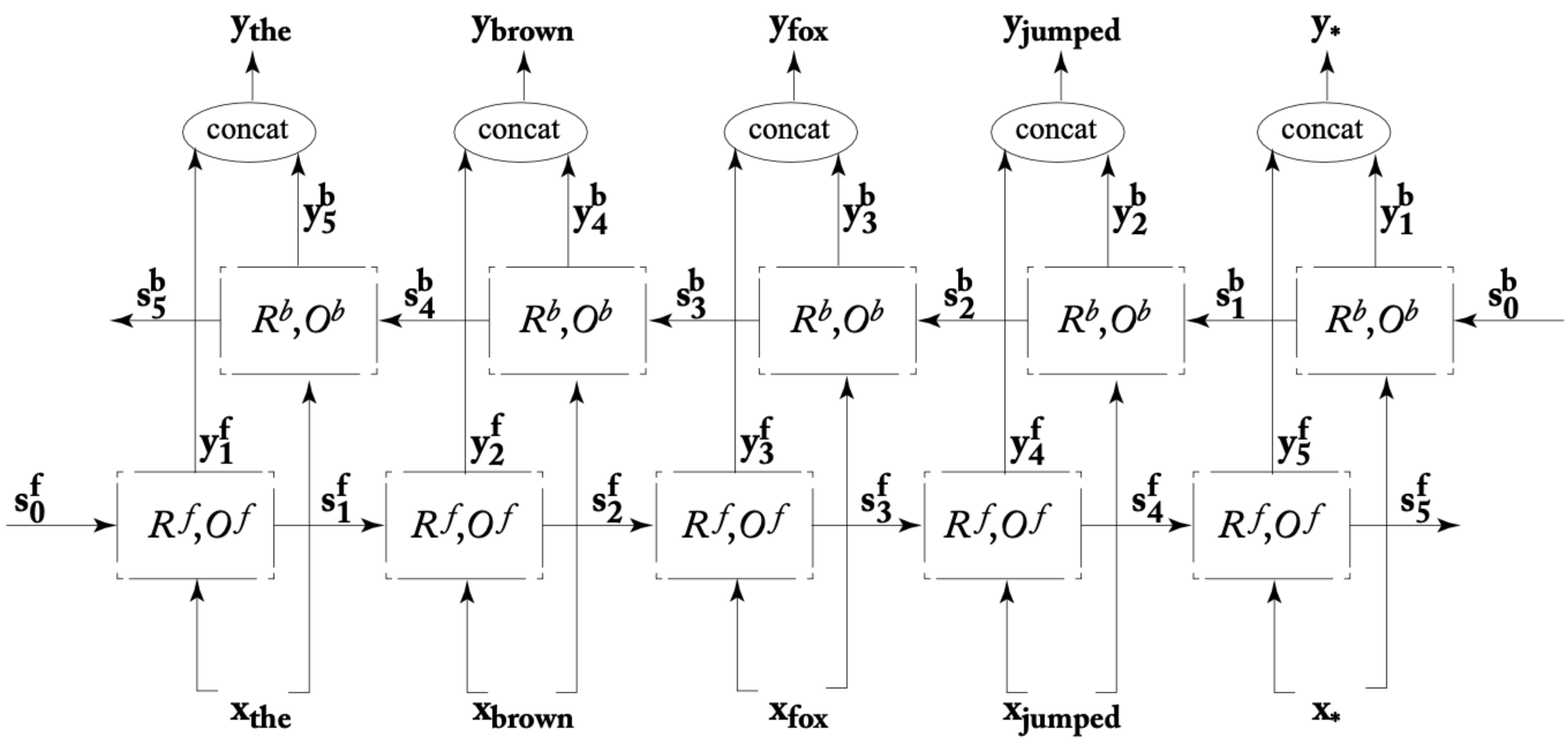
- Create an output for each input



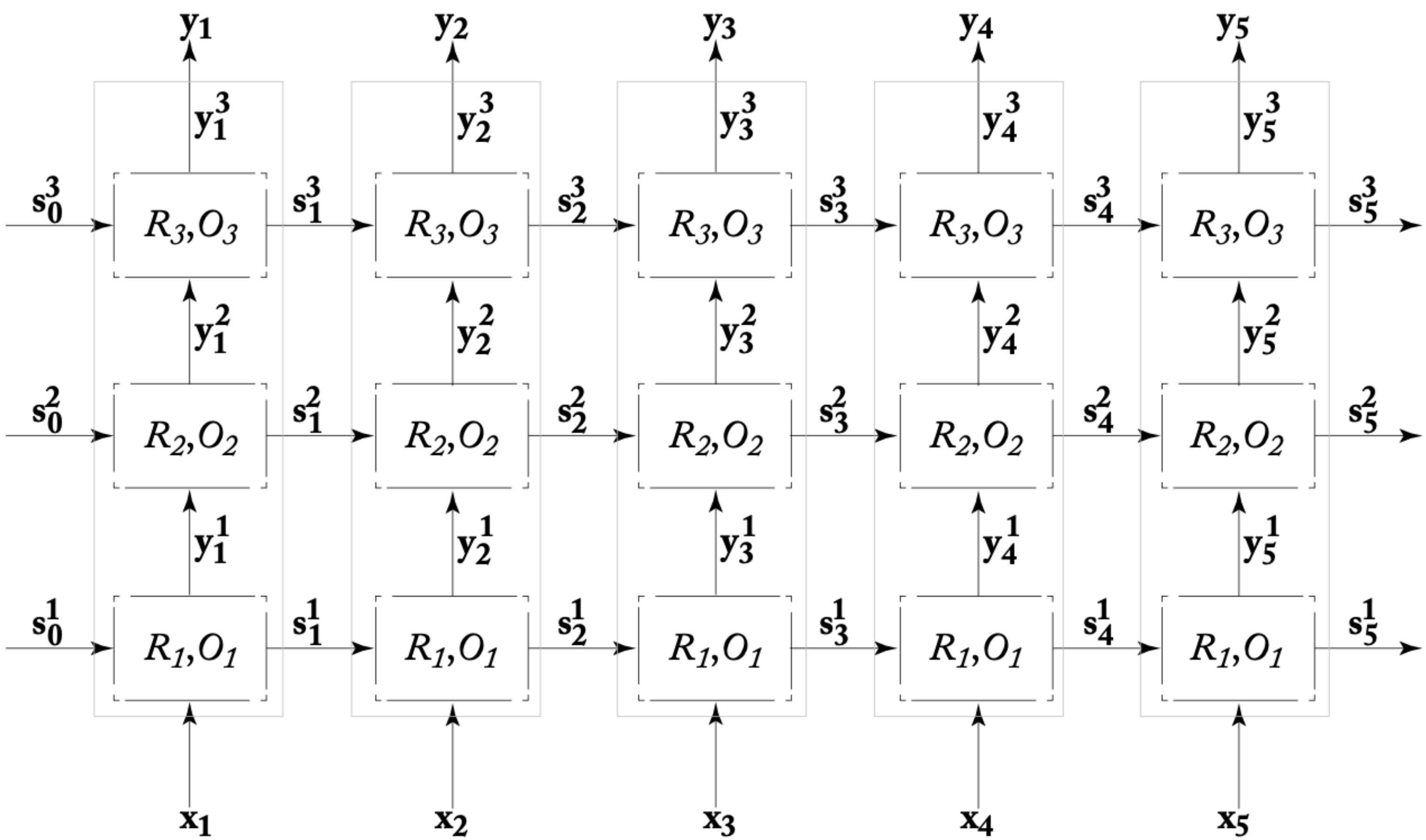
How else can we expand this?



Bi-directional



Stack more layers



Pytorch - nn.RNN

Parameters:

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional RNN. Default: `False`

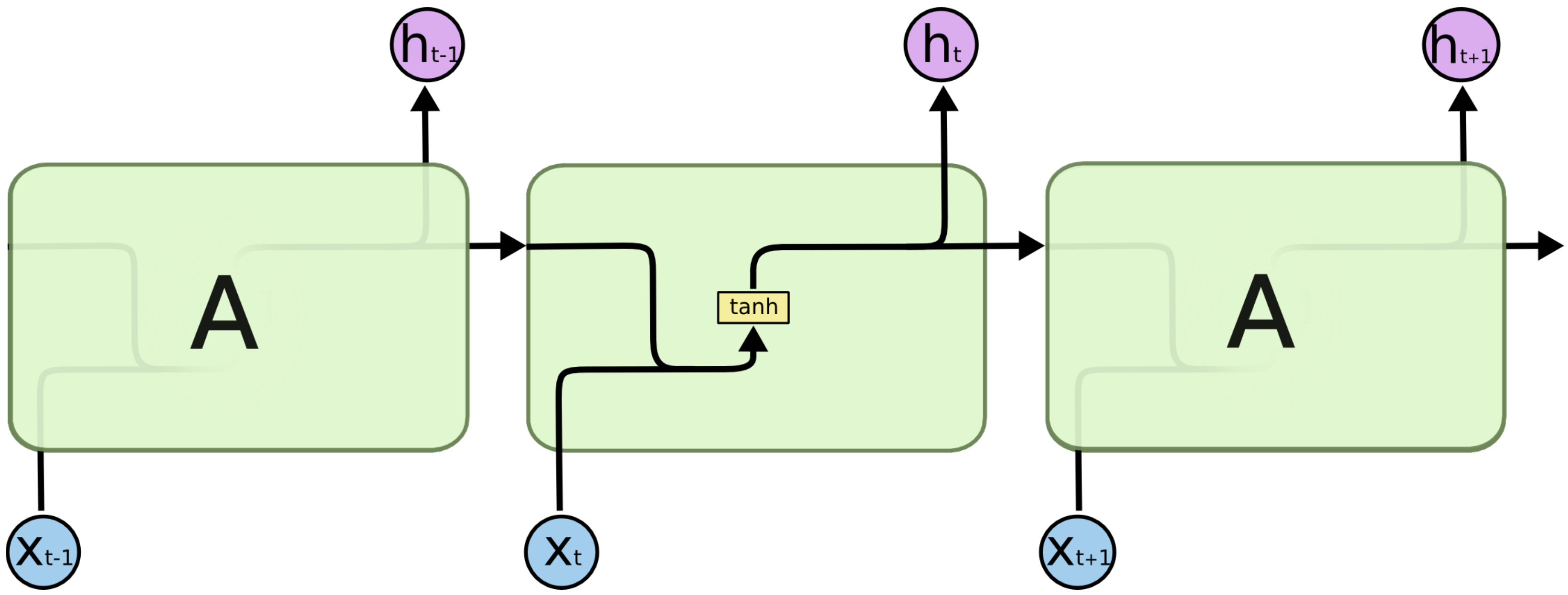
RNNs – long input

RNNs can remember anything (in theory)

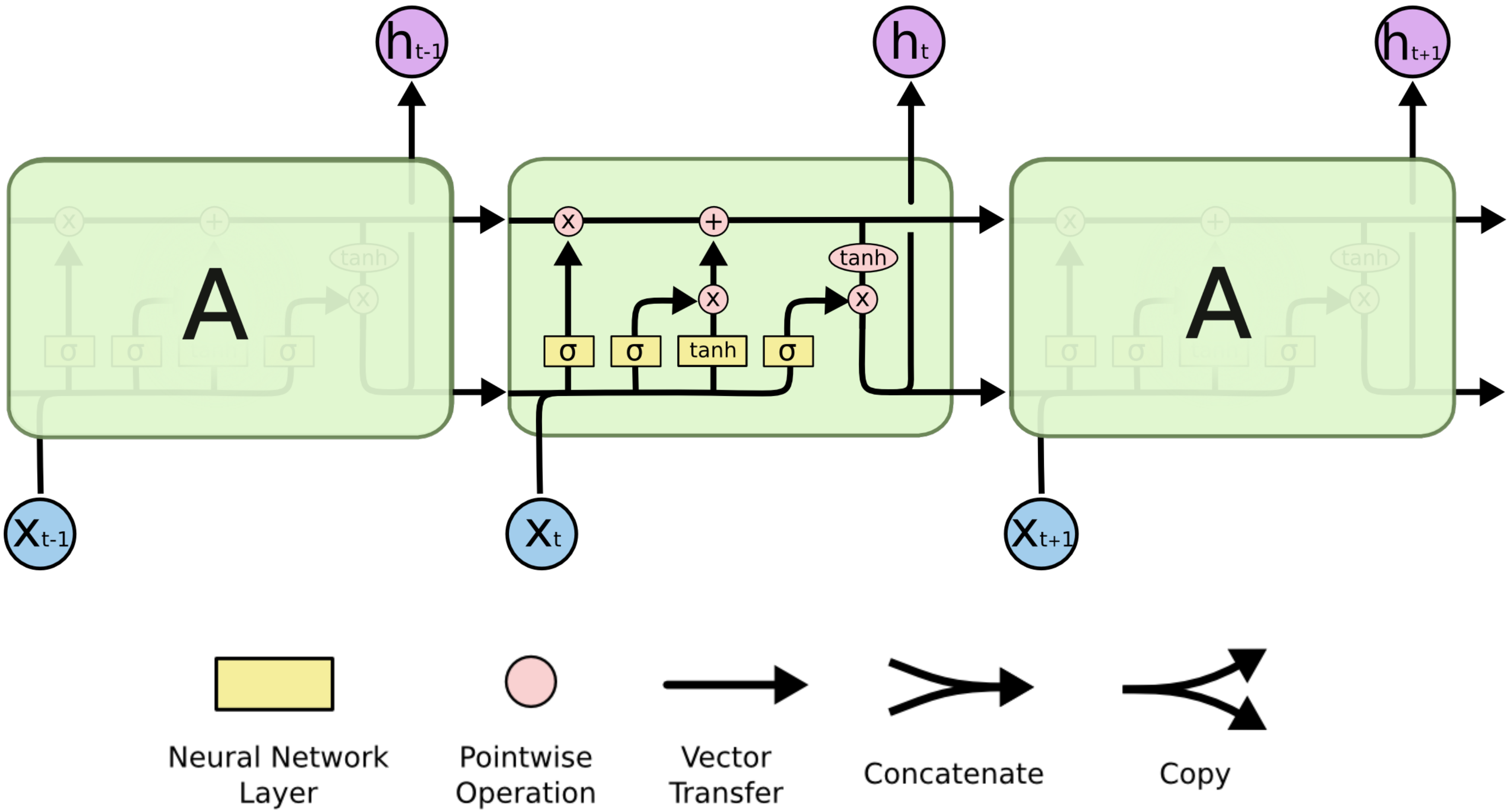
Sometimes its important to forget

Solution: Long-Short Term Memory (LSTM)

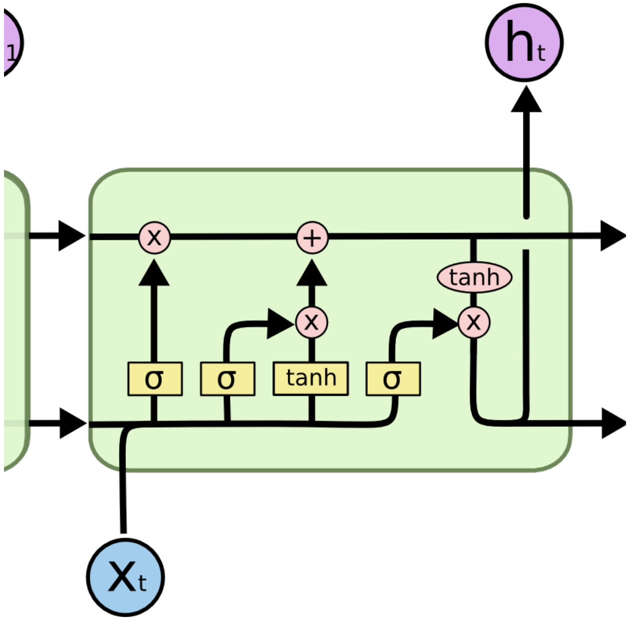
RNN internal



LSTM internal



LSTM internal



$$s_j = R_{\text{LSTM}}(s_{j-1}, x_j) = [c_j ; h_j]$$

$$c_j = f \odot c_{j-1} + i \odot z$$

$$h_j = o \odot \tanh(c_j)$$

$$i = \sigma(x_j W^{xi} + h_{j-1} W^{hi})$$

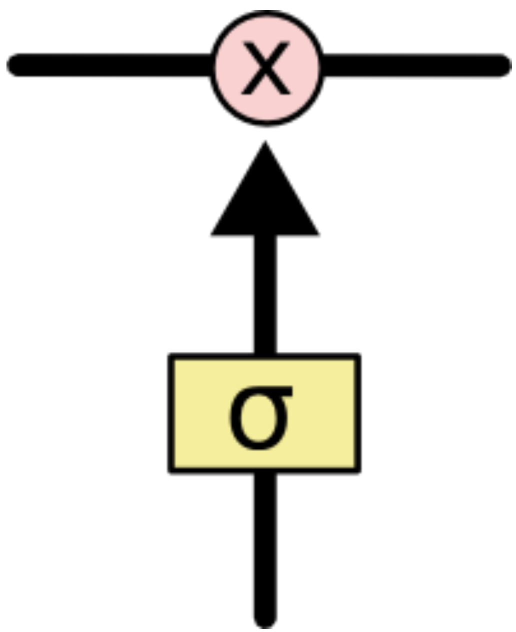
$$f = \sigma(x_j W^{xf} + h_{j-1} W^{hf})$$

$$o = \sigma(x_j W^{xo} + h_{j-1} W^{ho})$$

$$z = \tanh(x_j W^{xz} + h_{j-1} W^{hz})$$

$$y_j = O_{\text{LSTM}}(s_j) = h_j$$

LSTM’s rely on gates

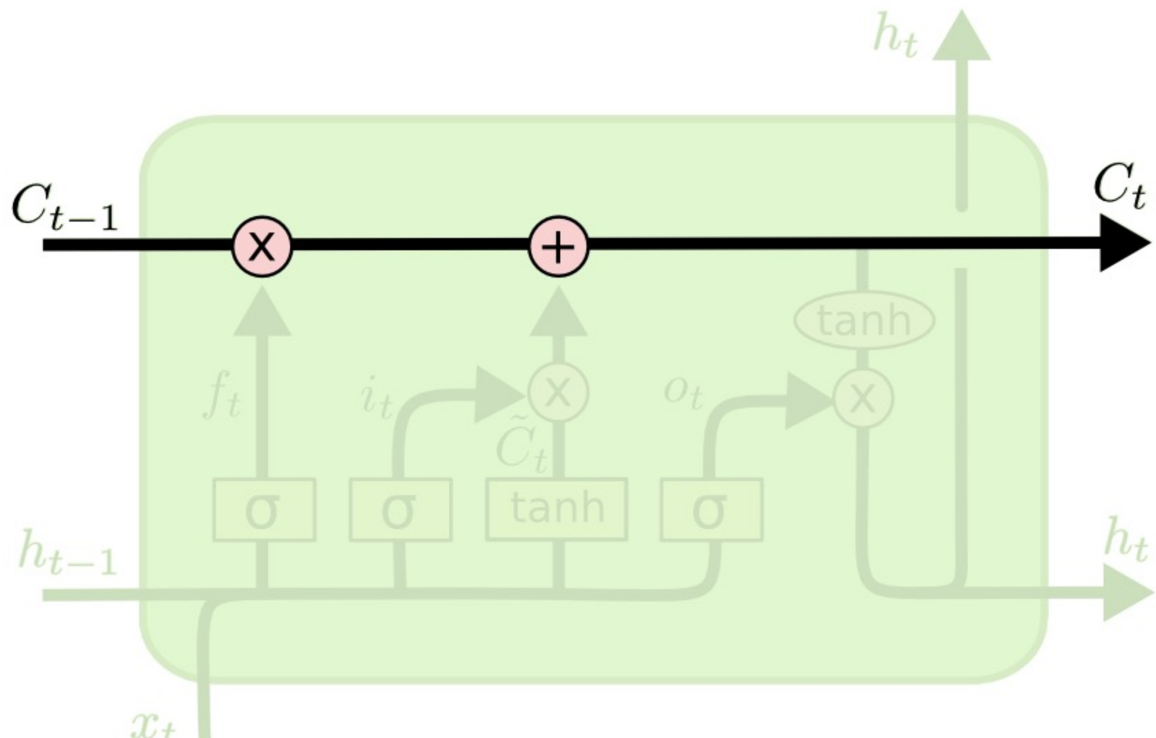


- Multiply input by value in 0,1]
- Zero means forget everything
- 1 means carry everything through (unchanged)

- 4 gates used in LSTM

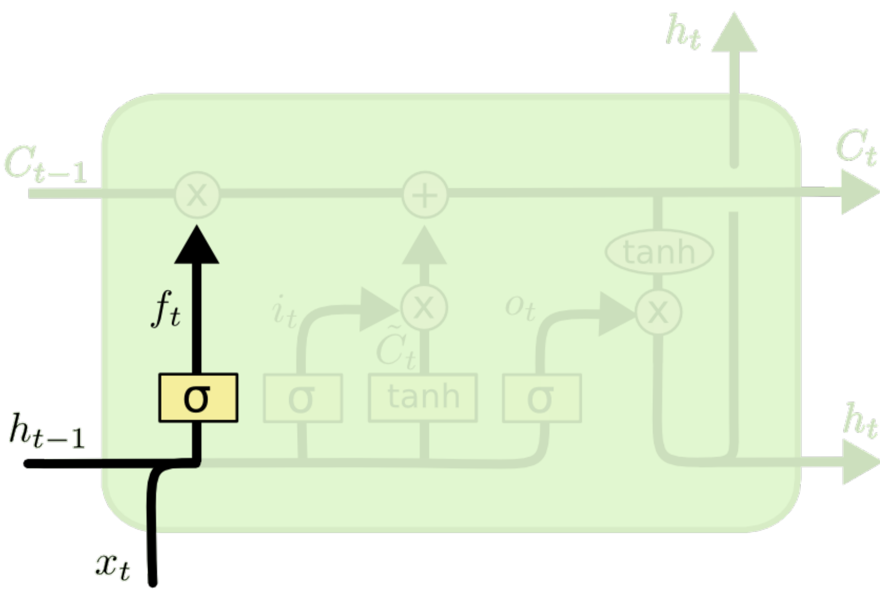
LSTM gates: cell state

- Passes the memory through the cell



LSTM gates: forget

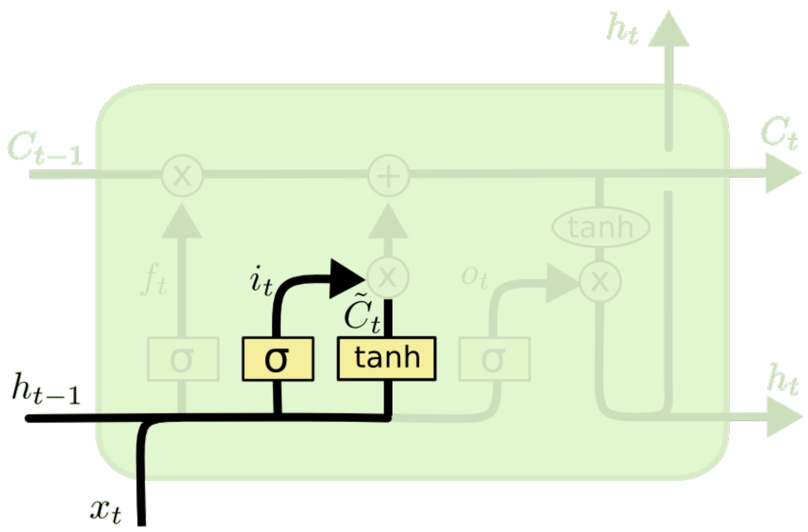
- Can decide to forget the previous state h_{t-1}



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

LSTM gates: update

- Compute new contribution to cell state based on hidden state and input.

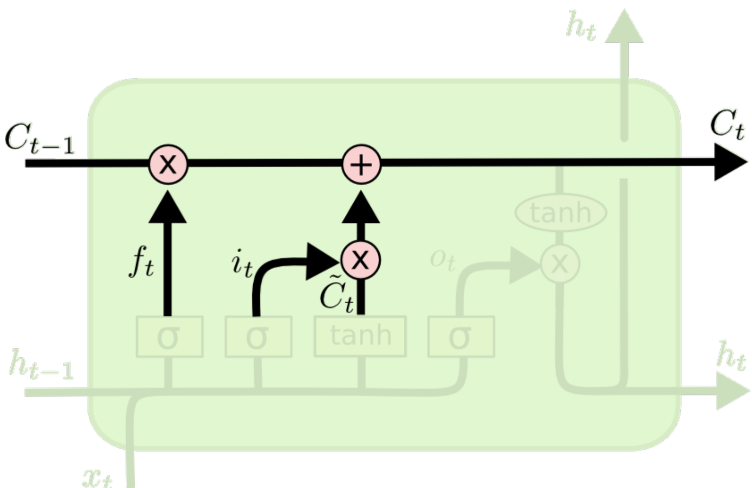


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

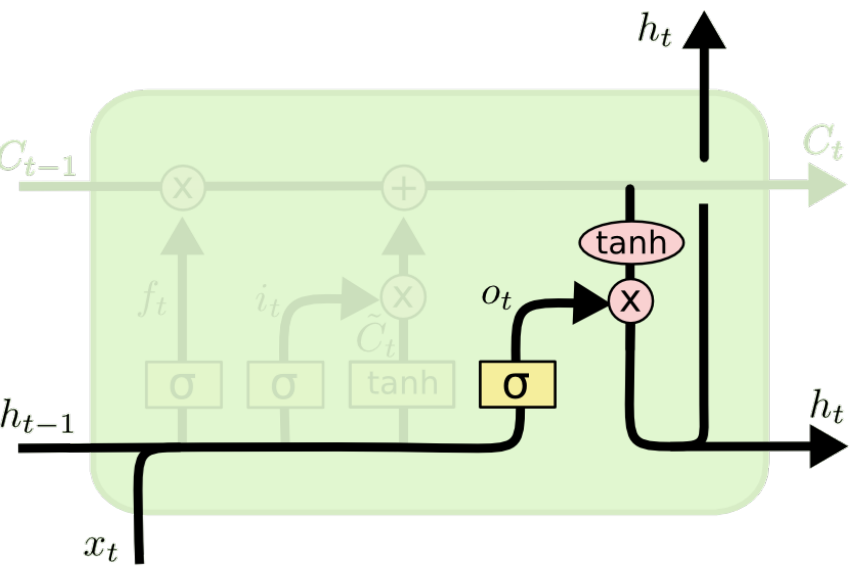
LSTM gates: update (interpolate)

- Can decide to forget the previous state h_{t-1}



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

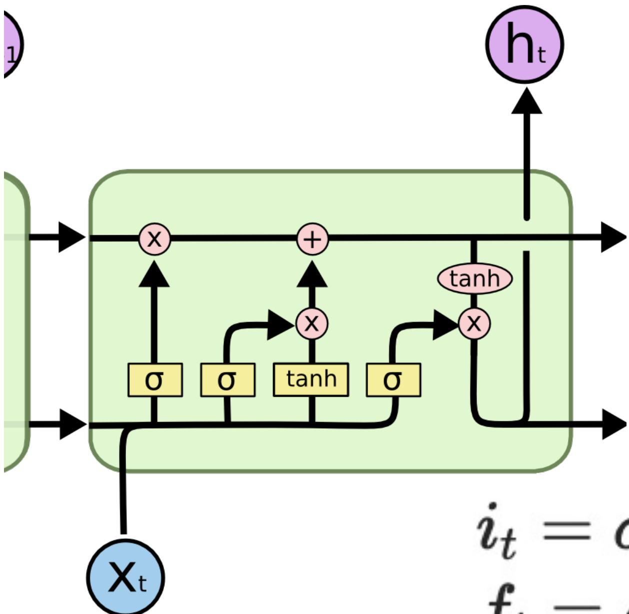
LSTM output (hidden)



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

LSTM internal



$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

Pytorch - nn.LSTM

Parameters:

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`
- **proj_size** – If `> 0`, will use LSTM with projections of corresponding size. Default: 0