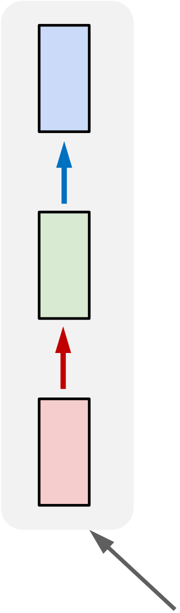
Lecture 10

Recurrent Neural Networks

"Vanilla" Neural Network

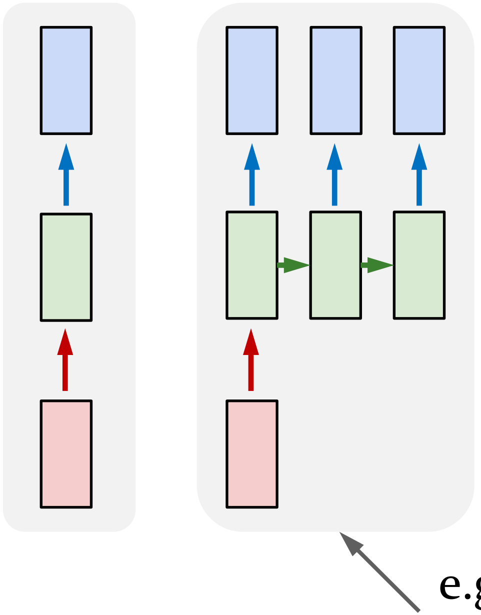
one to one



Vanilla Neural Networks

Recurrent Neural Networks: Process Sequences

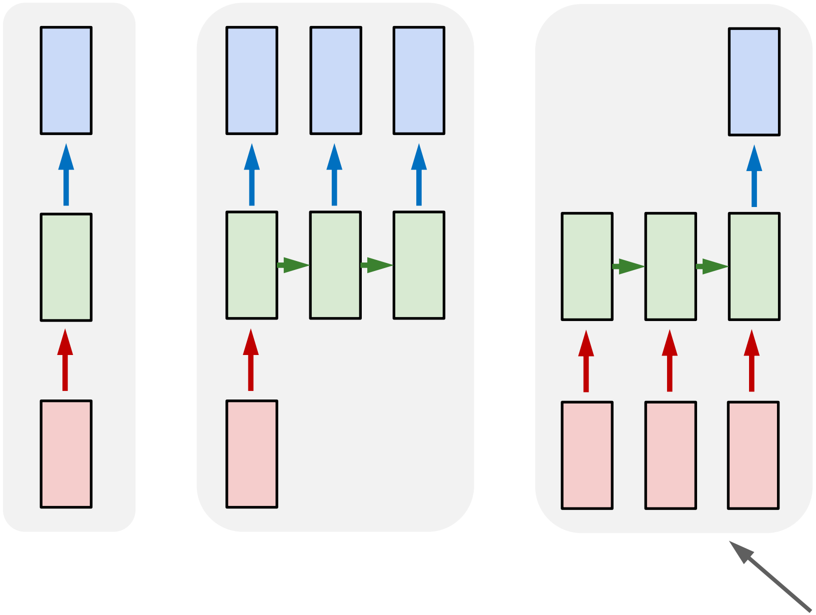
one to one one to many



e.g. Image Captioning
image -> sequence of words

Recurrent Neural Networks: Process Sequences

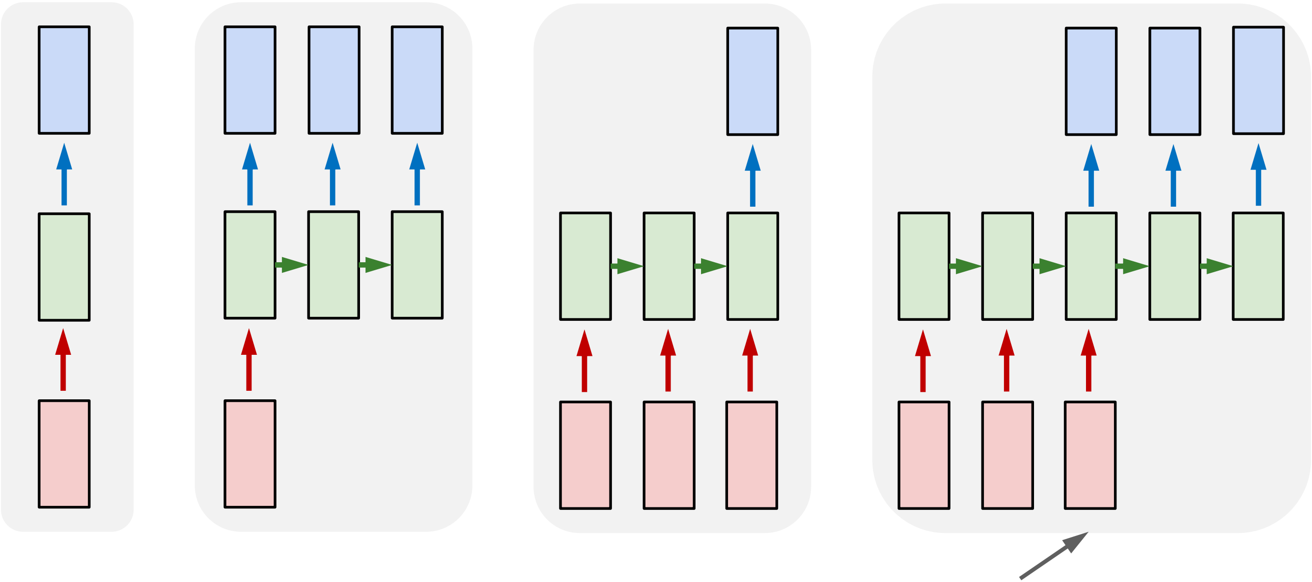
one to one one to many many to one



e.g. action prediction
sequence of video frames -> action class

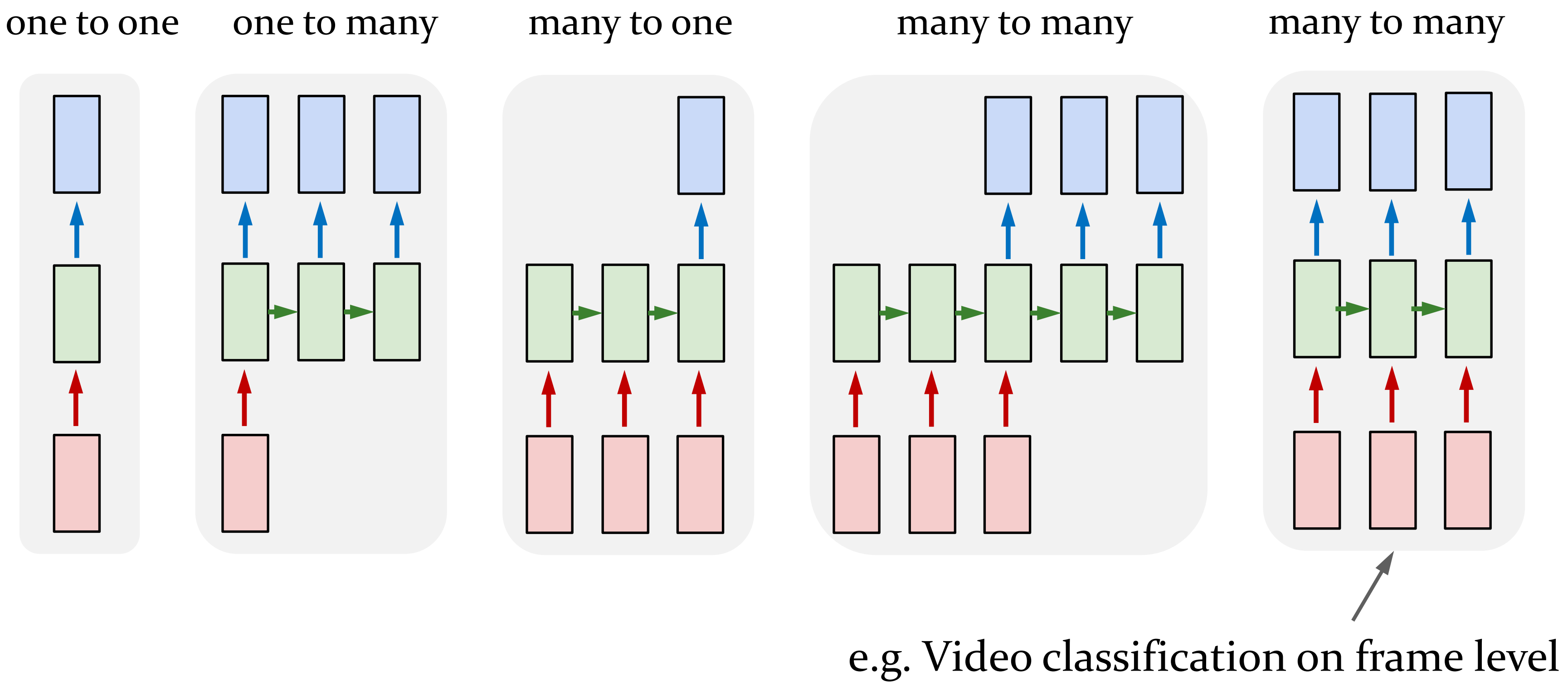
Recurrent Neural Networks: Process Sequences

one to one one to many many to one many to many

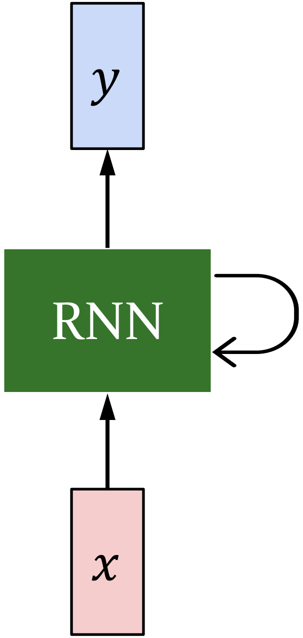


e.g. Video Captioning
Sequence of video frames -> caption

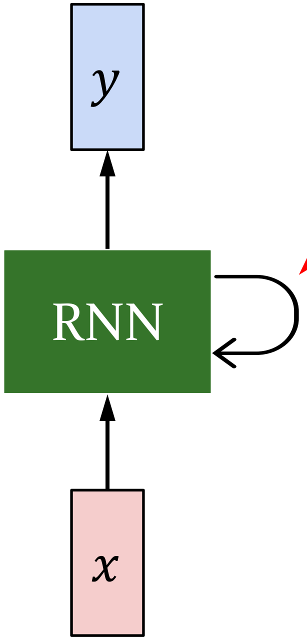
Recurrent Neural Networks: Process Sequences



Recurrent Neural Network

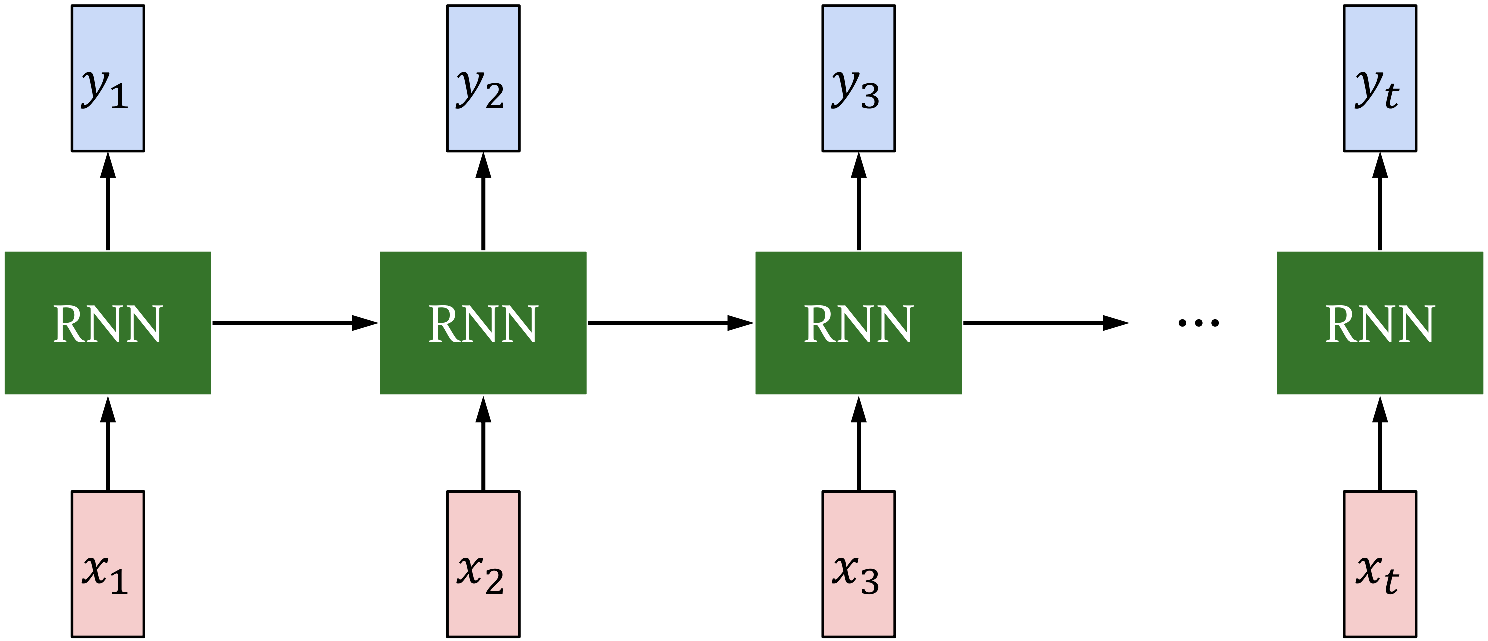


Recurrent Neural Network



Key idea: RNNs have an "**internal state**" that is updated as a sequence is processed

Unrolled RNN



RNN hidden state update

- We can process a sequence of vectors $\boldsymbol{x}$ by applying a **recurrence formula** at every time step:

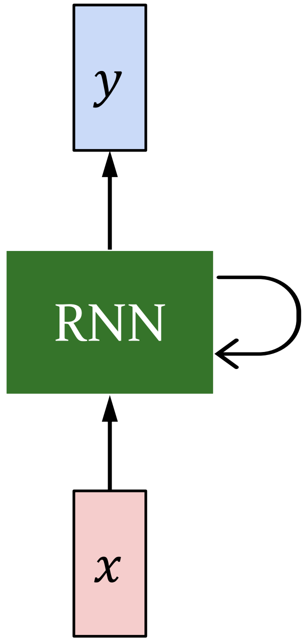
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

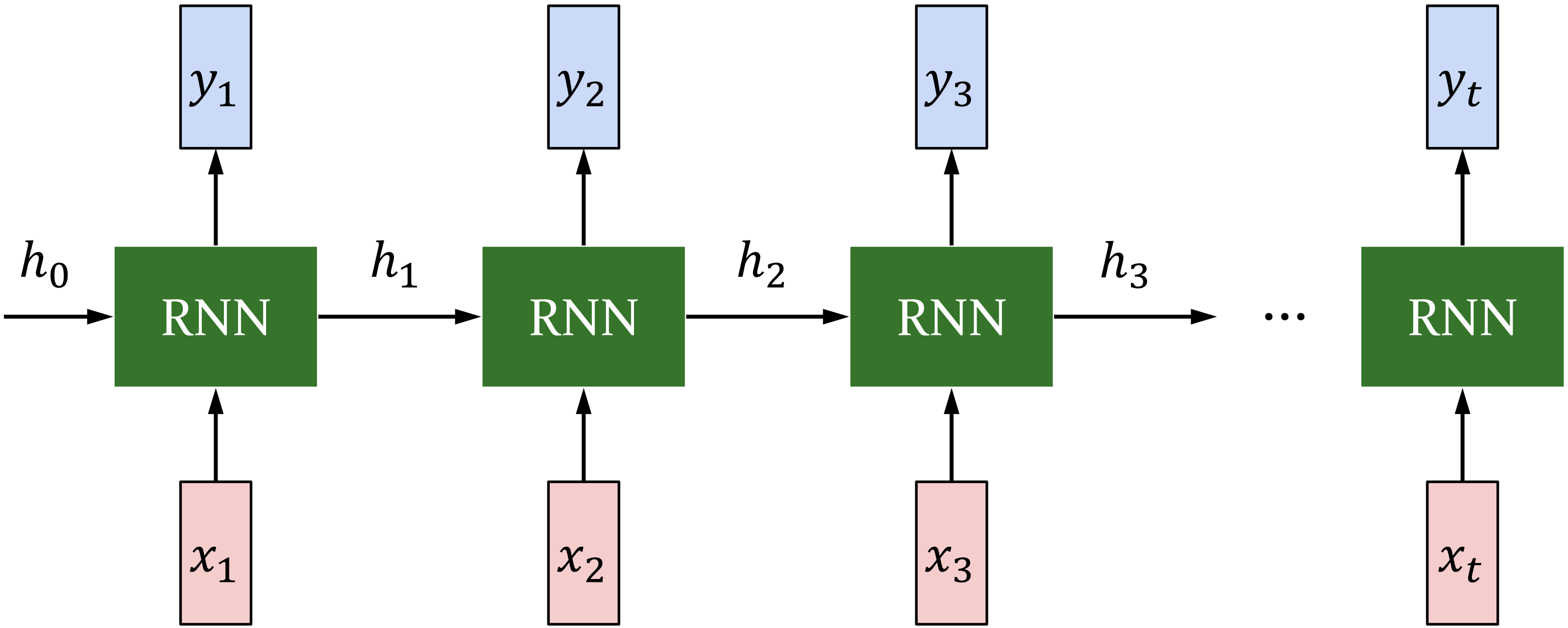
some function with parameter W

old state

input vector at time step t



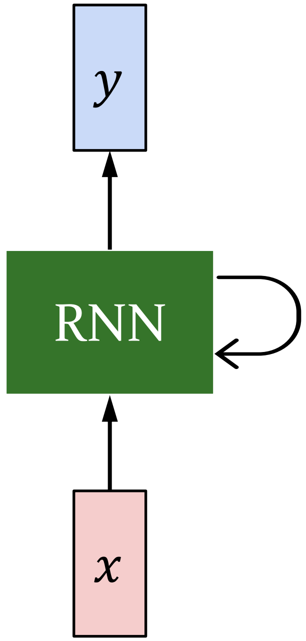
Recurrent Neural Network



Recurrent Neural Network

- We can process a sequence of vectors ***x*** by applying a **recurrence formula** at every time step:

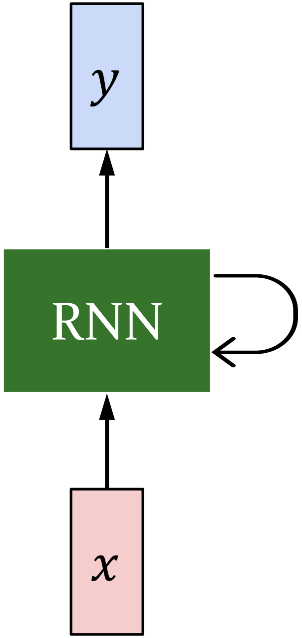
$$h_t = f_W(h_{t-1}, x_t)$$



Notice: the same function and the same set of parameters are used at every time step.

(Vanilla) Recurrent Neural Network

The state consists of a single "hidden" vector ***h***:



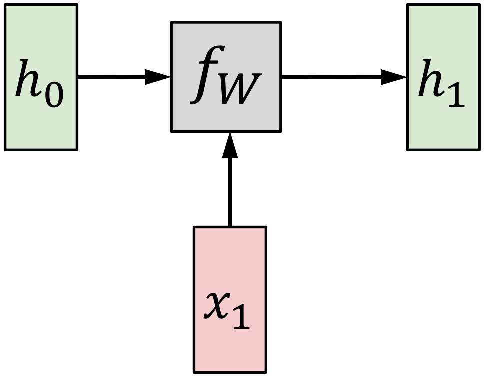
$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

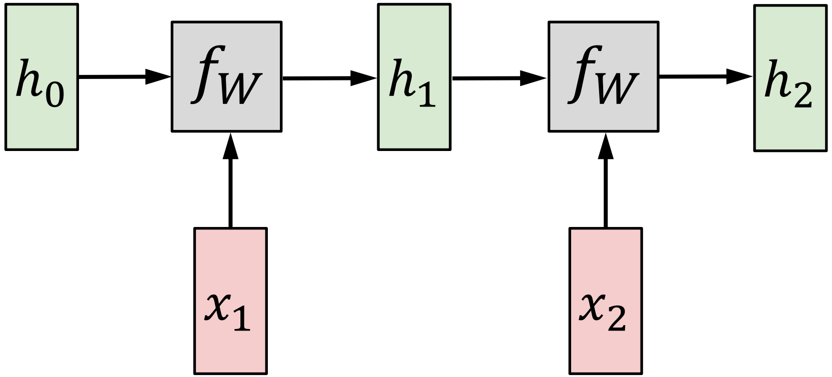
$$y_t = W_{hy}h_t$$

$$\boxed{\begin{gathered}\tanh x = \frac{e^{2x}-1}{e^{2x}+1} \\ \frac{d}{dx}\tanh x = 1 - \tanh^2 x\end{gathered}}$$

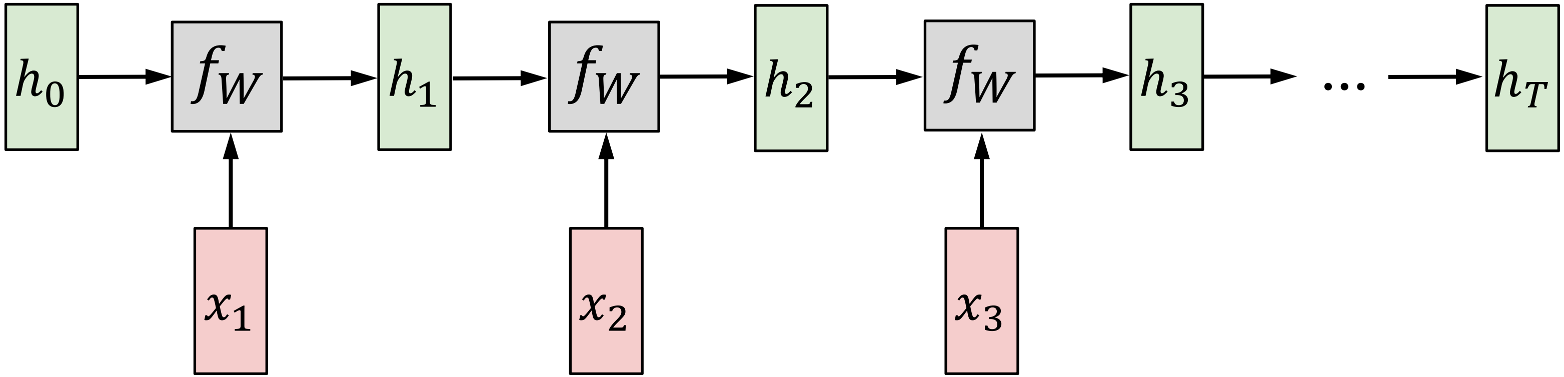
RNN: Computational Graph



RNN: Computational Graph

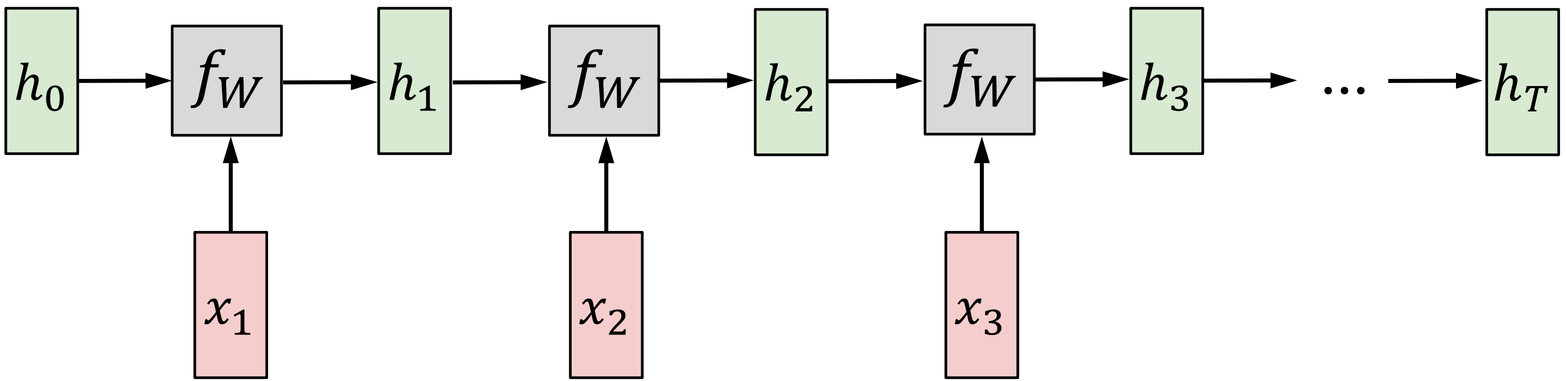


RNN: Computational Graph

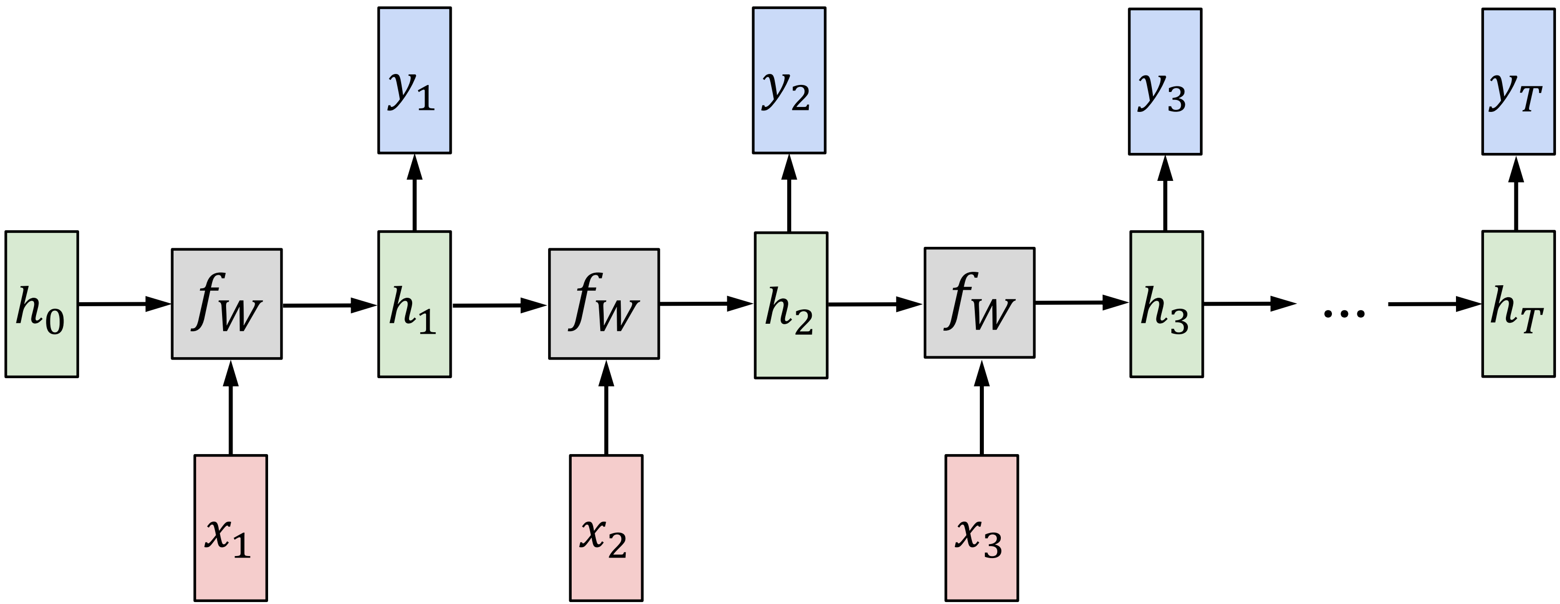


RNN: Computational Graph

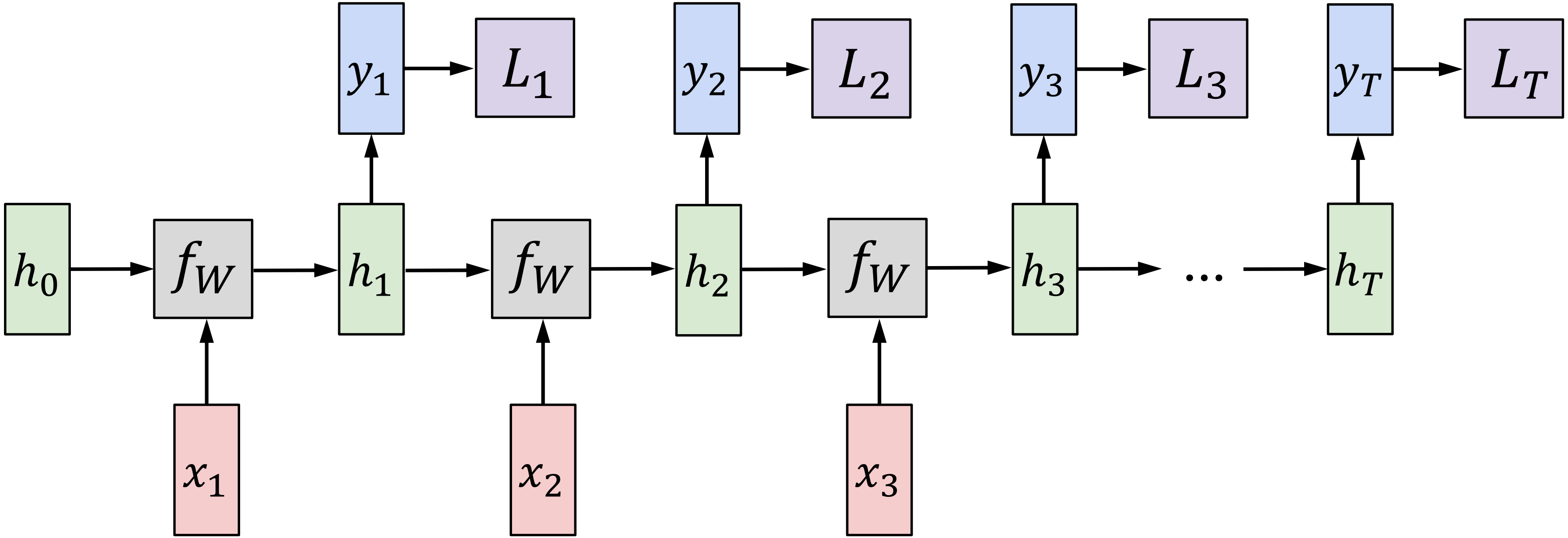
Re-use the same weight matrix at every time-step



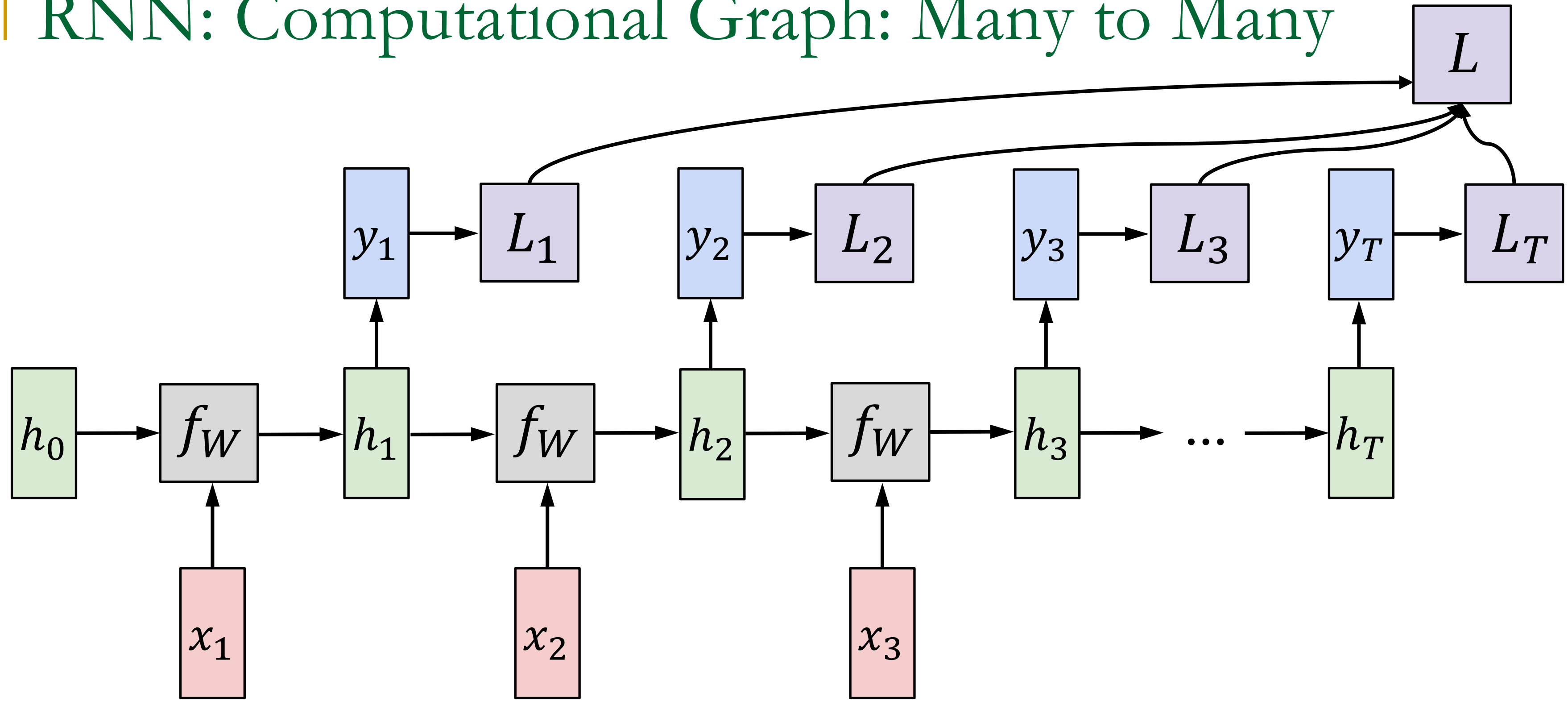
RNN: Computational Graph: Many to Many



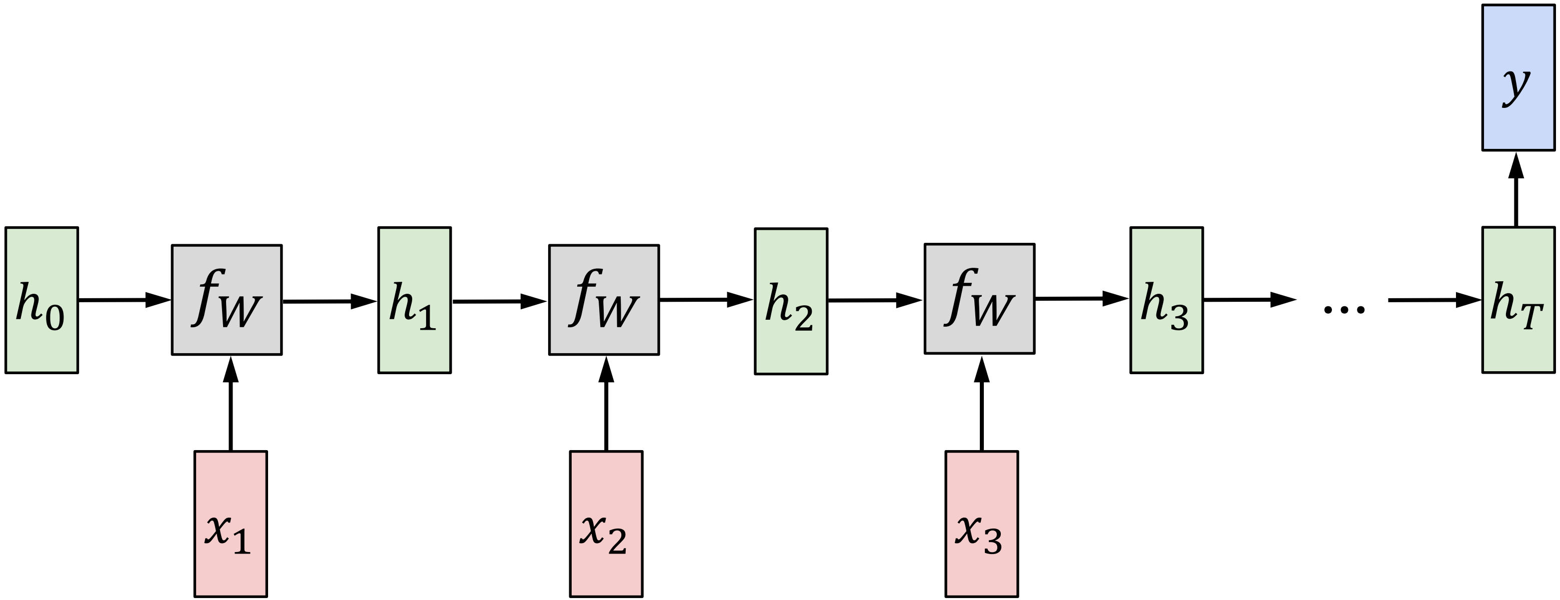
RNN: Computational Graph: Many to Many



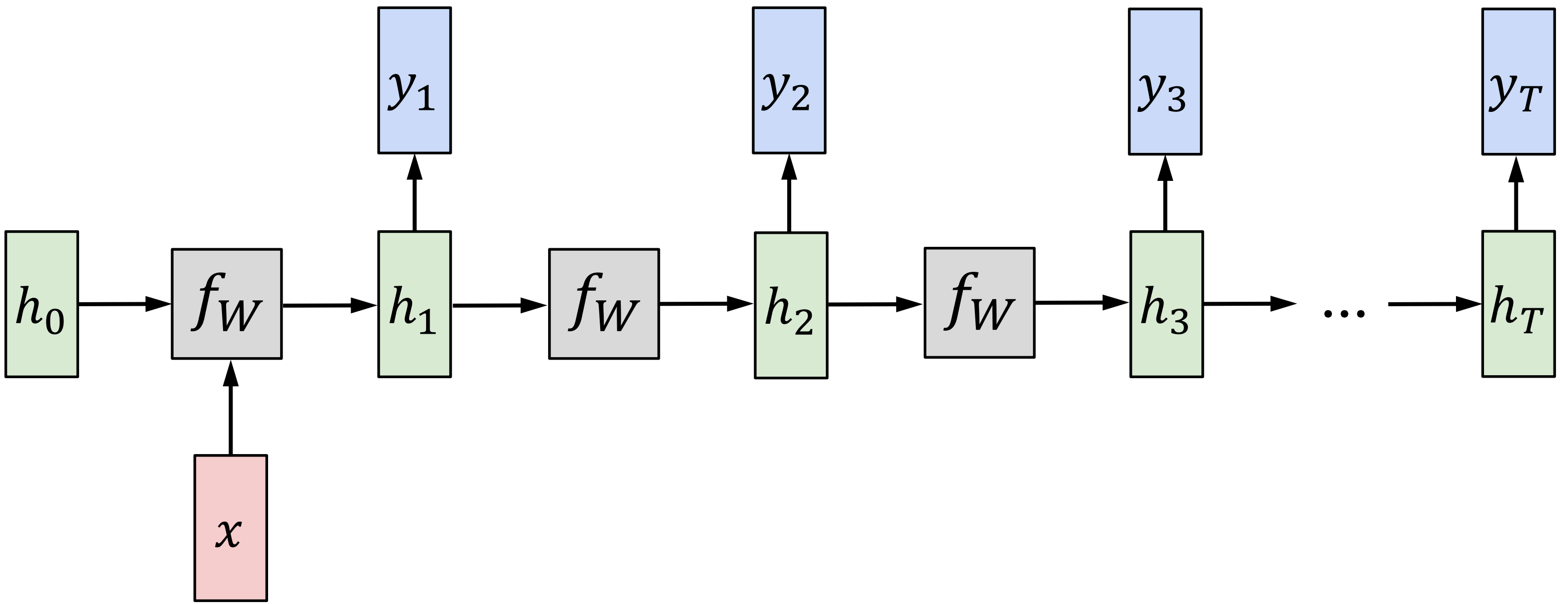
RNN: Computational Graph: Many to Many



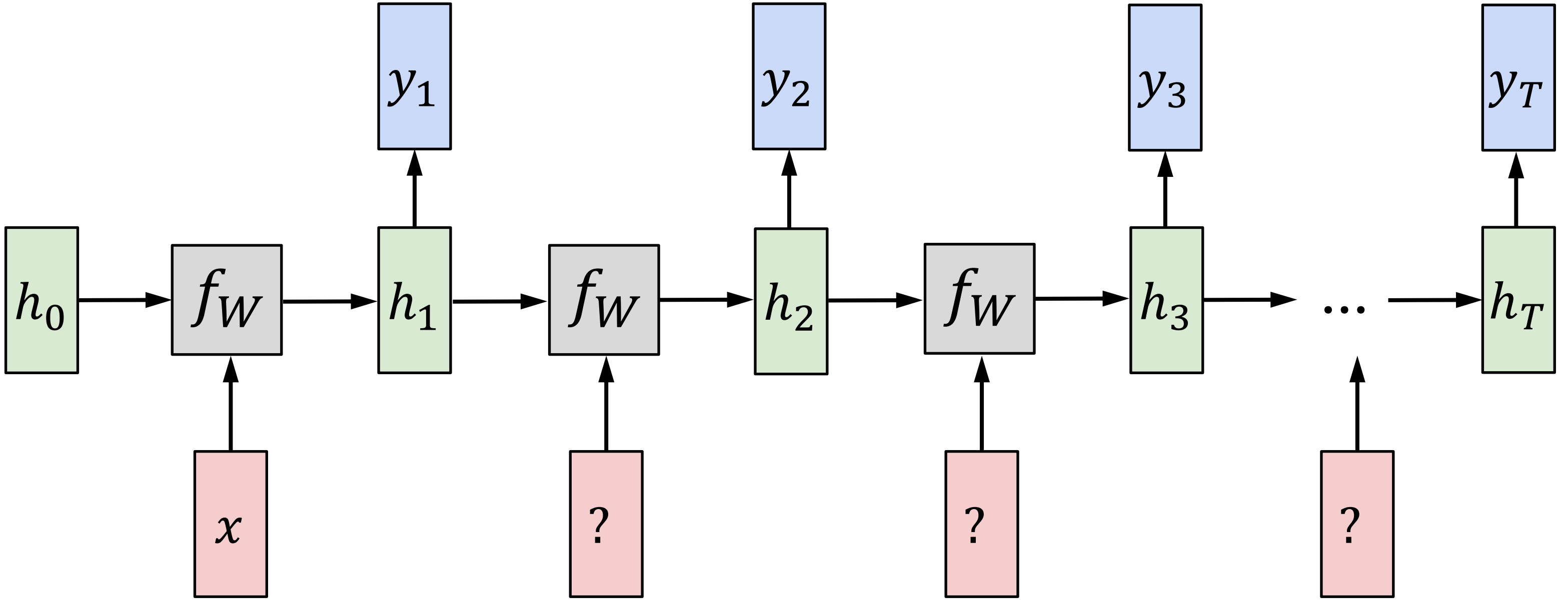
RNN: Computational Graph: Many to One



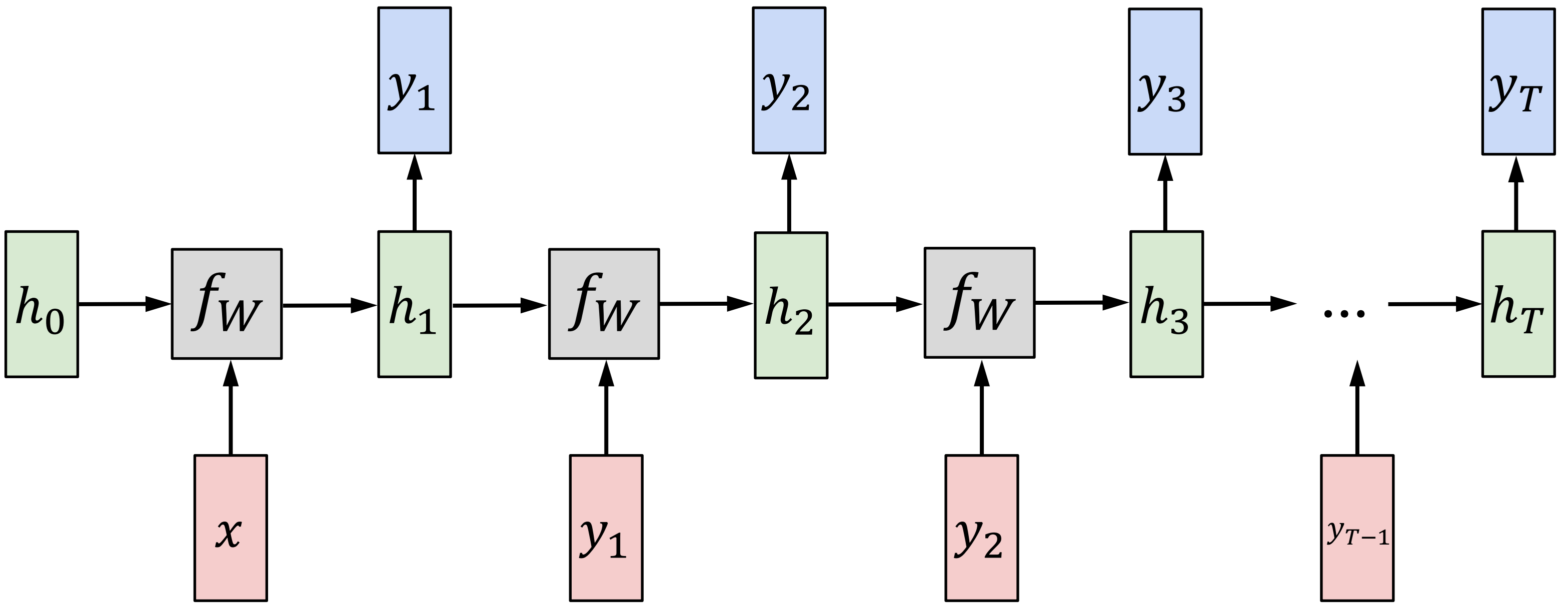
RNN: Computational Graph: One to Many



RNN: Computational Graph: One to Many



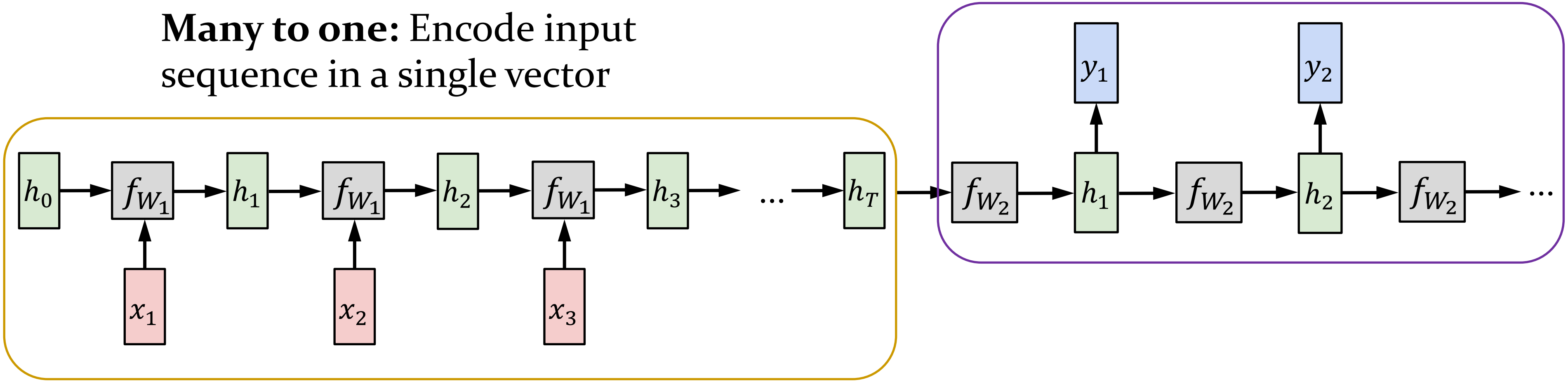
RNN: Computational Graph: One to Many



Sequence to Sequence: Many-to-one + one-to-many

One to many: Produce output sequence from single input vector

Many to one: Encode input sequence in a single vector

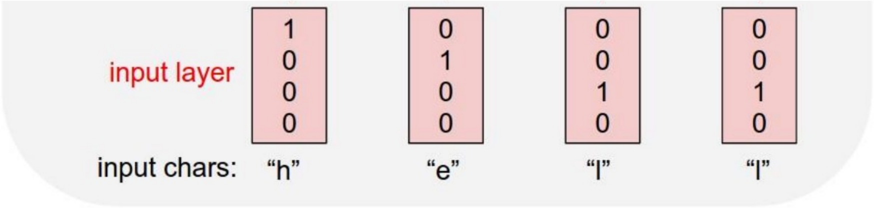


Sutskever et al, "Sequence to Sequence Learning with Neural Networks", NIPS 2014

Example: Character-level Language Model

Vocabulary: [h,e,l,o]

Example training sequence: “**hello**”

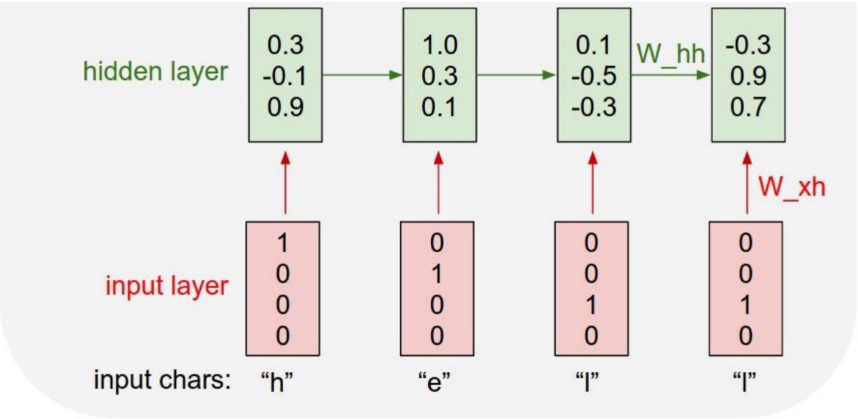


Example: Character-level Language Model

Vocabulary: [h,e,l,o]

Example training sequence: "**hello**"

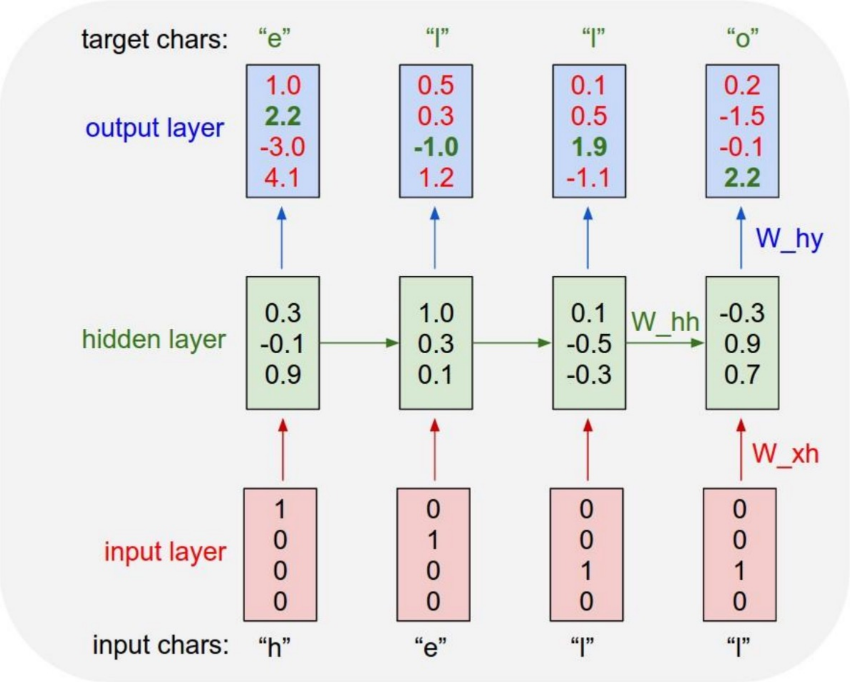
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$



Example: Character-level Language Model

Vocabulary: [h,e,l,o]

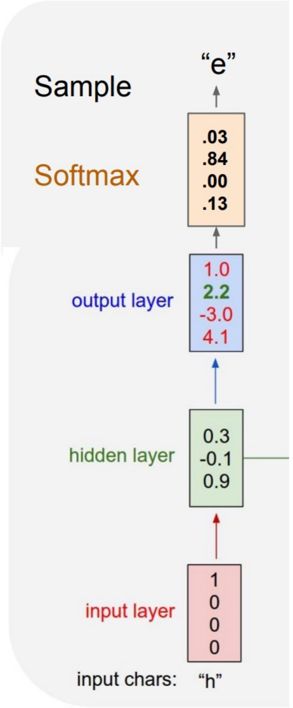
Example training sequence: “**hello**”



Example: Character-level Language Model

Vocabulary: [h,e,l,o]

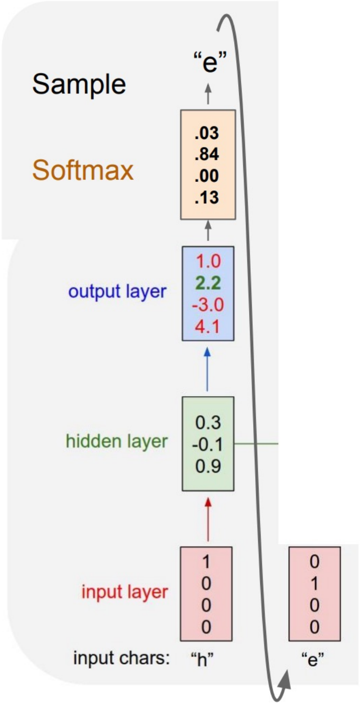
At test-time sample characters one at a time, feed back to model



Example: Character-level Language Model

Vocabulary: [h,e,l,o]

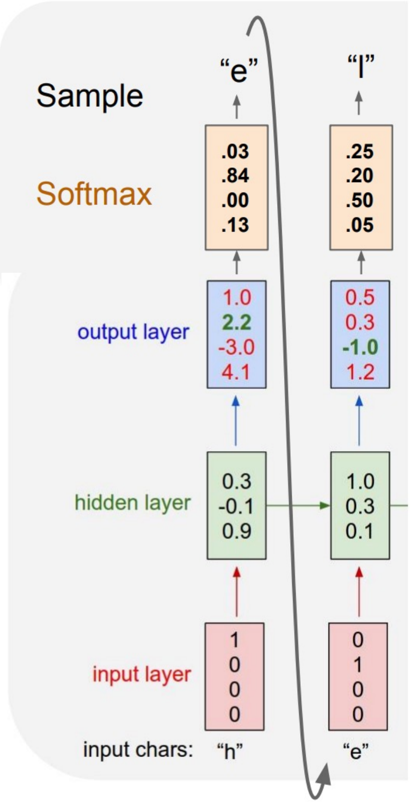
At test-time sample characters one at a time, feed back to model



Example: Character-level Language Model

Vocabulary: [h,e,l,o]

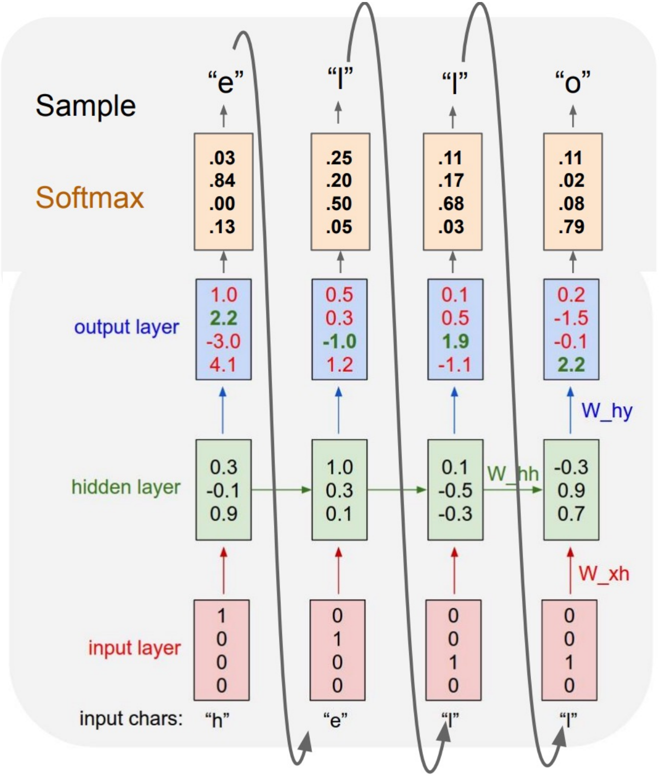
At test-time sample characters one at a time, feed back to model



Example: Character-level Language Model

Vocabulary: [h,e,l,o]

At test-time sample characters one at a time, feed back to model

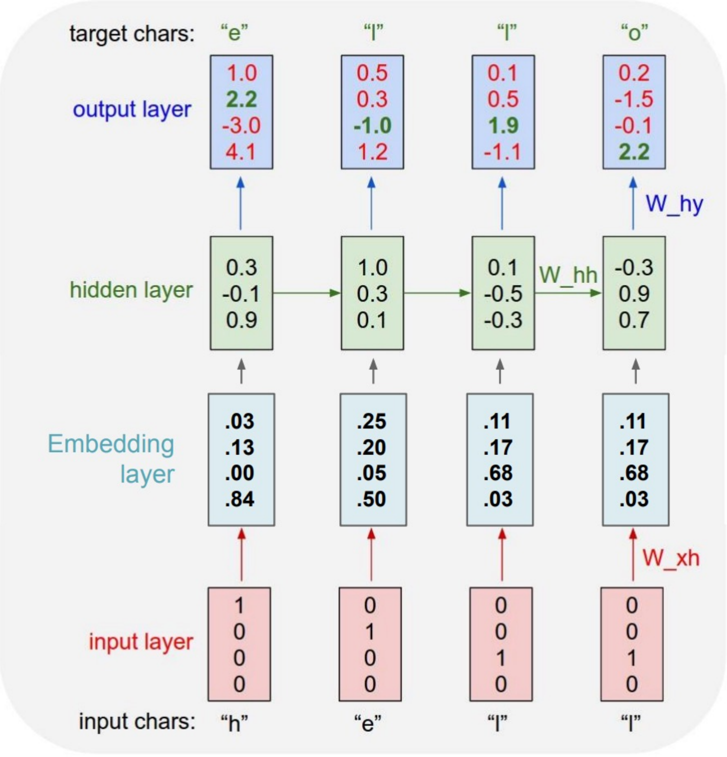


Example: Character-level Language Model

Vocabulary: [h,e,l,o]

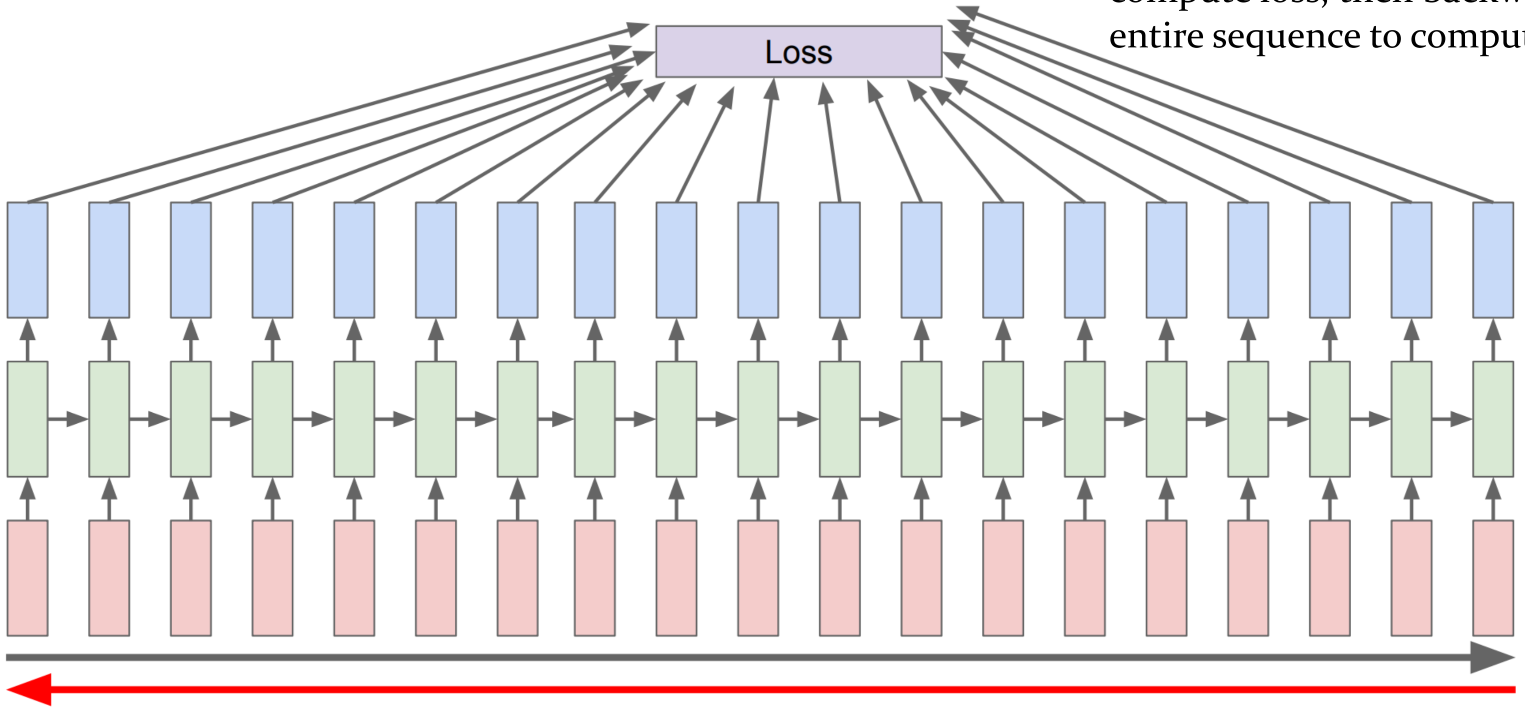
Matrix multiply with a one-hot vector just extracts a column from the weight matrix. We often put a separate embedding layer between input and hidden layers.

$$\begin{matrix}[w_{11}\ w_{12}\ w_{13}\ w_{14}] \\ [w_{21}\ w_{22}\ w_{23}\ w_{14}] \\ [w_{31}\ w_{32}\ w_{33}\ w_{14}] \\ \end{matrix} \begin{matrix}[1] \\ [0] \\ [0] \\ [0]\end{matrix} = \begin{matrix}[w_{11}] \\ [w_{21}] \\ [w_{31}] \\ \end{matrix}$$



Backpropagation through time (BPTT)

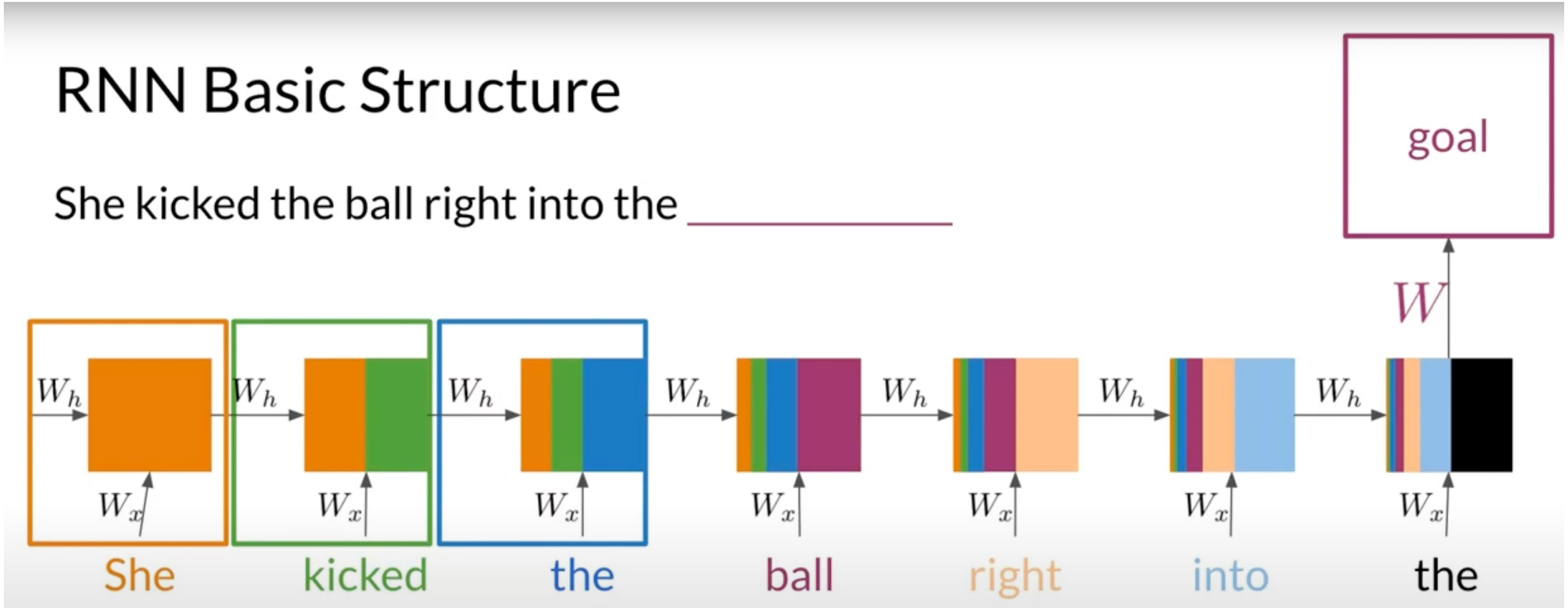
Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient



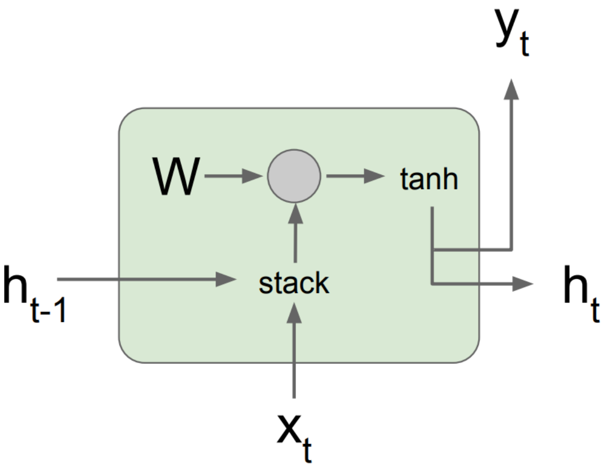
RNN tradeoffs

- RNN Advantages:
  - Can process any length input
  - Computation for step t can (in theory) use information from many steps back
  - Model size doesn't increase for longer input
  - Same weights applied on every timestep, so there is symmetry in how inputs are processed.
- RNN Disadvantages:
  - Recurrent computation is slow
  - In practice, difficult to access information from many steps back

Vanilla RNN Vanishing Gradients

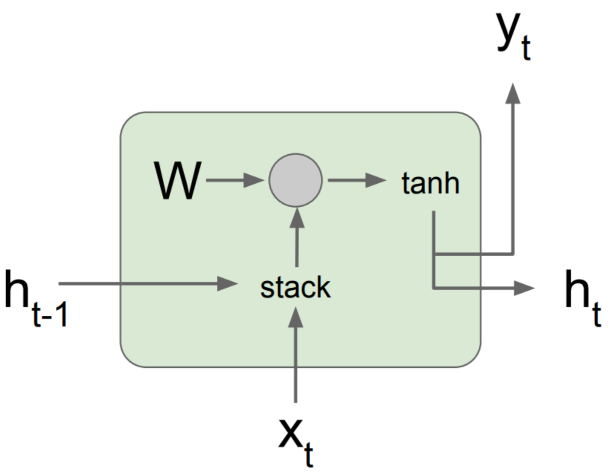


Vanilla RNN Gradient Flow



$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

Vanilla RNN Gradient Flow

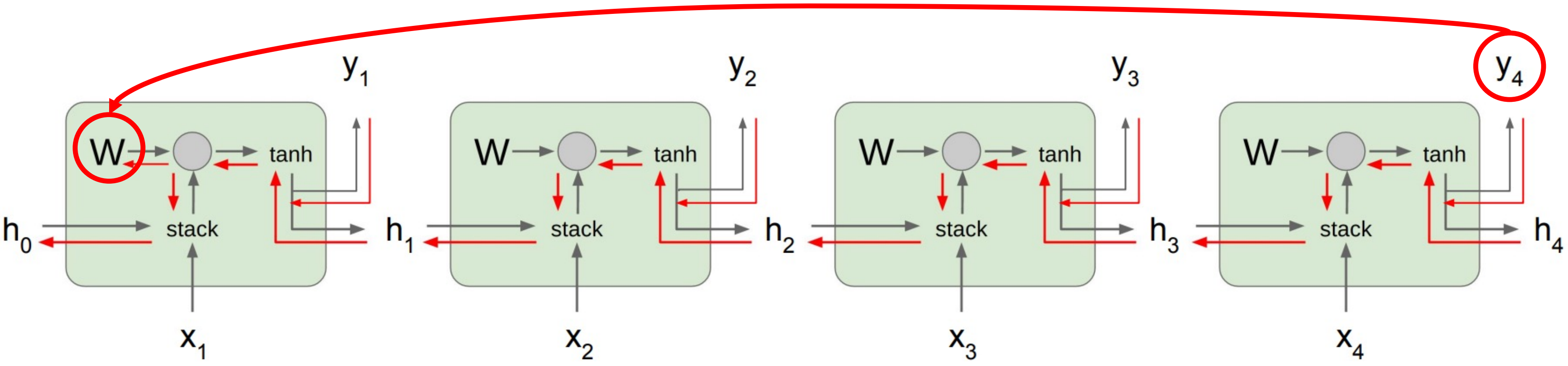


$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix}\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

$$
\frac{\partial h_t}{\partial h_{t-1}} = \tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}
$$

Vanilla RNN Gradient Flow

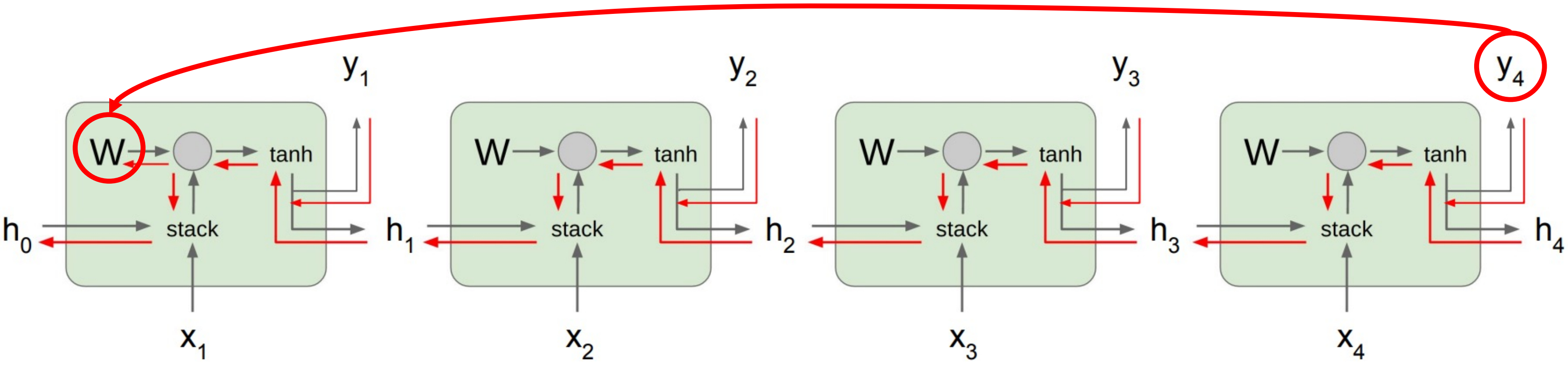
Gradients over multiple time steps:



$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W}$$

Vanilla RNN Gradient Flow

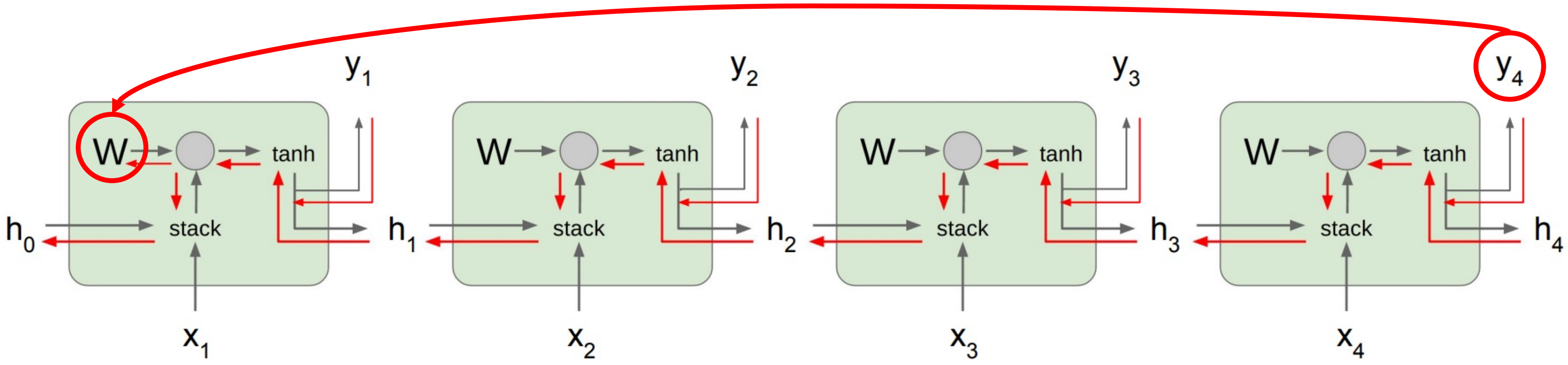
Gradients over multiple time steps:



$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T} \left(\prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial h_1}{\partial W}$$

Vanilla RNN Gradient Flow

Gradients over multiple time steps:

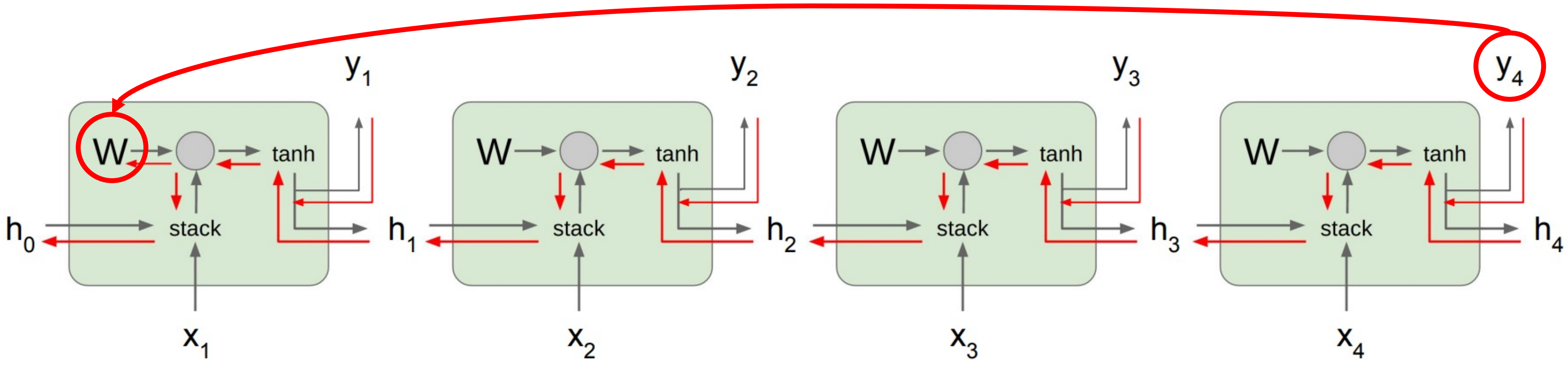


$$\boxed{\frac{\partial h_t}{\partial h_{t-1}} = \tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T}\frac{\partial h_t}{\partial h_{t-1}}\cdots\frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T}\boxed{\frac{\partial h_t}{\partial h_{t-1}}}\right)\frac{\partial h_1}{\partial W}$$

Vanilla RNN Gradient Flow

Gradients over multiple time steps:

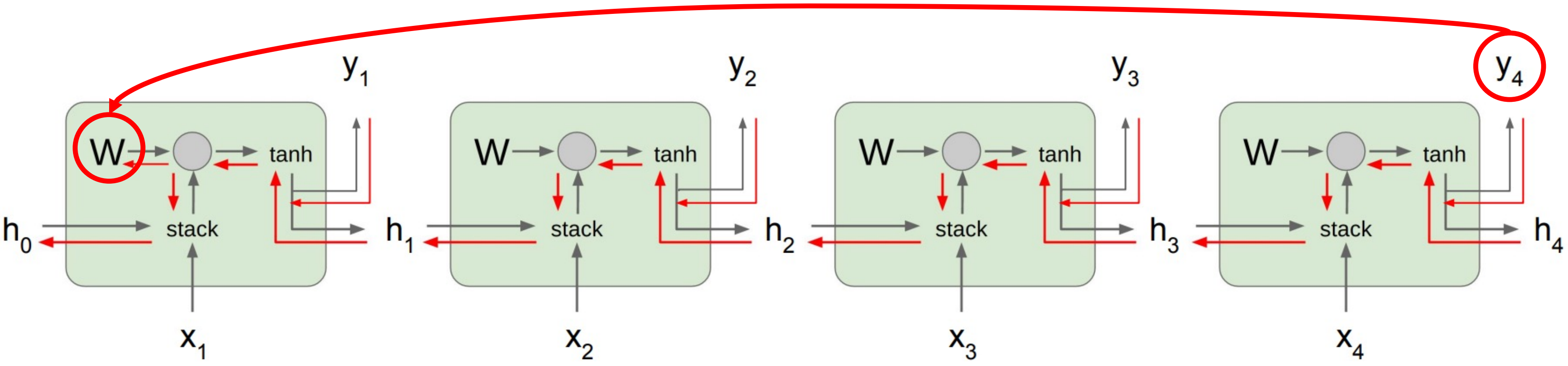


Almost always < 1
Vanishing gradients

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} \boxed{\tanh'(W_{hh}h_{t-1} + W_{xh}x_t)}\right) W_{hh}^{T-1} \frac{\partial h_1}{\partial W}$$

Vanilla RNN Gradient Flow

Gradients over multiple time steps:



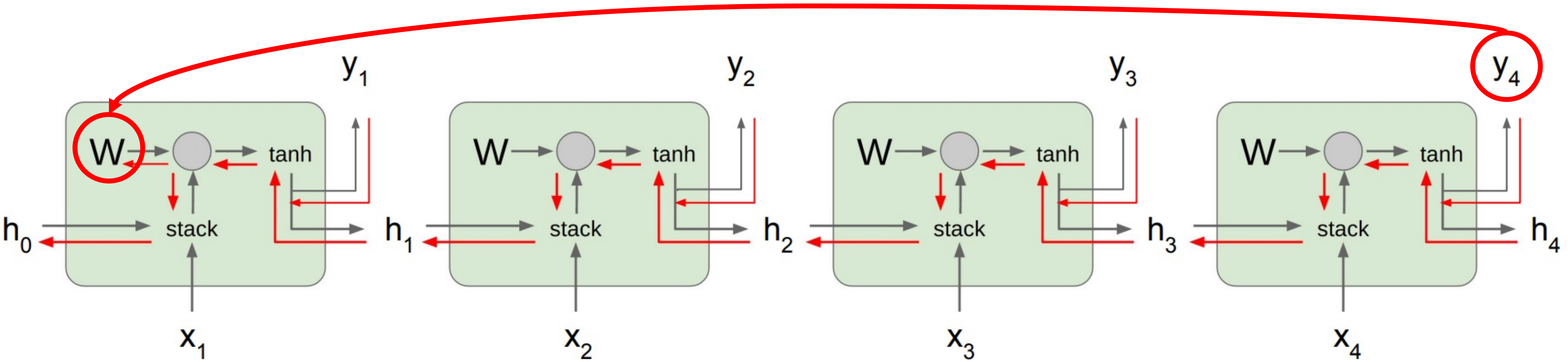
Largest singular value > 1: **Exploding gradients**

Largest singular value < 1: **Vanishing gradients**

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} \tanh'(W_{hh}h_{t-1} + W_{xh}x_t)\right)\boxed{W_{hh}^{T-1}}\frac{\partial h_1}{\partial W}$$

Vanilla RNN Gradient Flow

Gradients over multiple time steps:



Largest singular value > 1: **Exploding gradients** → **Gradient clipping:** Scale gradient if its norm is too big

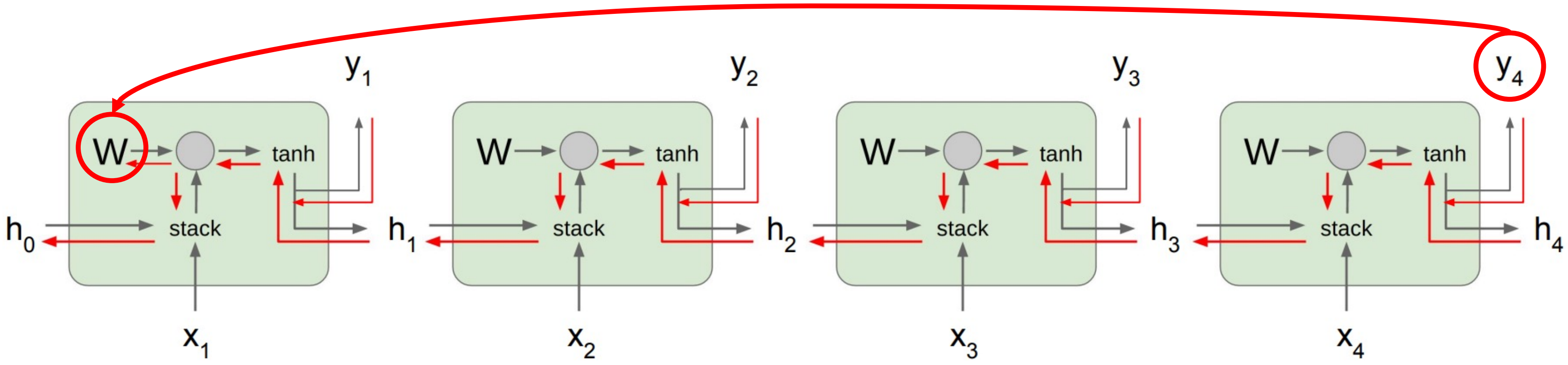
Largest singular value < 1: **Vanishing gradients**

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} \tanh'(W_{hh}h_{t-1} + W_{xh}x_t)\right)\boxed{W_{hh}^{T-1}}\frac{\partial h_1}{\partial W}$$

```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
  grad *= (threshold / grad_norm)
```

Vanilla RNN Gradient Flow

Gradients over multiple time steps:



Largest singular value > 1: **Exploding gradients**

Largest singular value < 1: **Vanishing gradients** → Change RNN architecture

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} \tanh'(W_{hh}h_{t-1} + W_{xh}x_t)\right)\boxed{W_{hh}^{T-1}}\frac{\partial h_1}{\partial W}$$

Long Short Term Memory (LSTM)

Vanilla RNN

$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix}\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

LSTM

$$
\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}
$$

$$
c_t = f \odot c_{t-1} + i \odot g
$$

$$
h_t = o \odot \tanh(c_t)
$$

Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

Long Short Term Memory (LSTM)

Vanilla RNN

$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)
\end{aligned}
$$

LSTM

Four gates

$$
\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}
$$

Cell state

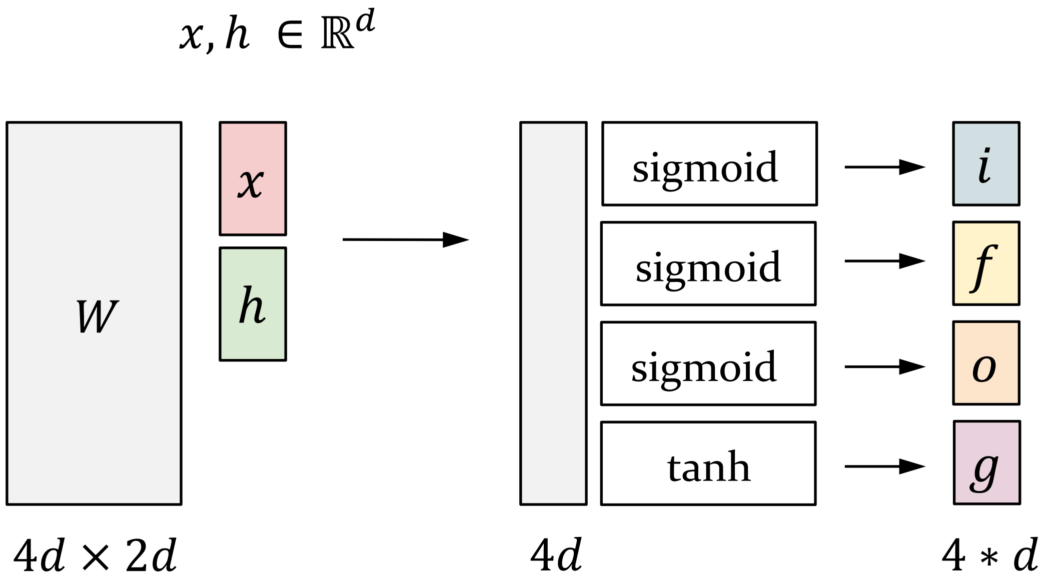
$$c_t = f \odot c_{t-1} + i \odot g$$

Hidden state

$$h_t = o \odot \tanh(c_t)$$

Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

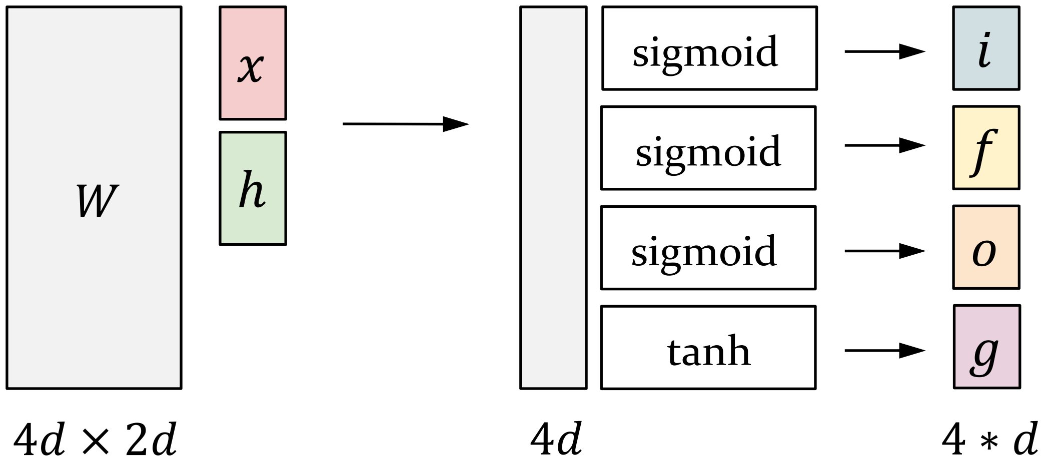
Long Short Term Memory (LSTM)



Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

Long Short Term Memory (LSTM)

$x, h \in \mathbb{R}^d$



g: <u>Gate gate</u>, how much to write to cell

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \boxed{\tanh} \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot \boxed{g}$$

$$h_t = o \odot \tanh(c_t)$$

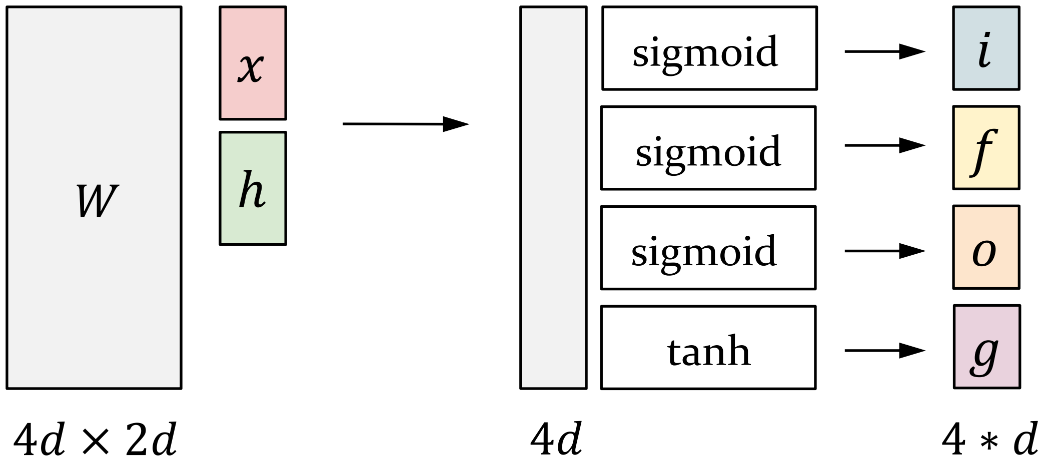
Hochreiter and Schmidhuber, “Long Short Term Memory”, Neural Computation 1997

Long Short Term Memory (LSTM)

i: <u>Input gate</u>, whether to write to cell

$x, h \in \mathbb{R}^d$

g: <u>Gate gate</u>, how much to write to cell



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \boxed{\sigma} \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + \boxed{i} \odot g$$

$$h_t = o \odot \tanh(c_t)$$

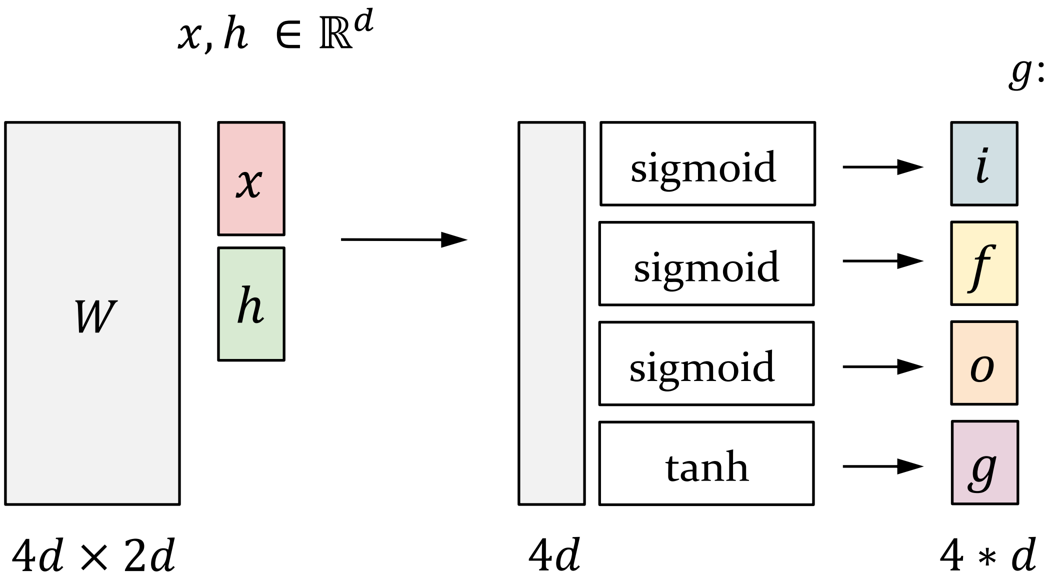
Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

Long Short Term Memory (LSTM)

i: Input gate, whether to write to cell

$\boxed{f\text{: Forget gate, whether to erase cell}}$

g: Gate gate, how much to write to cell



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \boxed{\sigma} \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = \boxed{f} \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

Long Short Term Memory (LSTM)

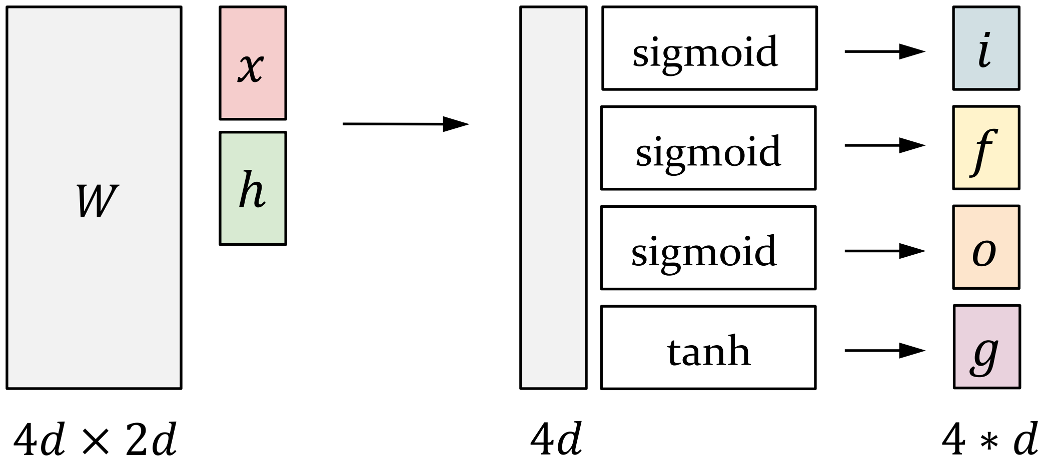
i: Input gate, whether to write to cell

f: Forget gate, whether to erase cell

o: Output gate, how much to reveal cell

g: Gate gate, how much to write to cell

$x, h \in \mathbb{R}^d$



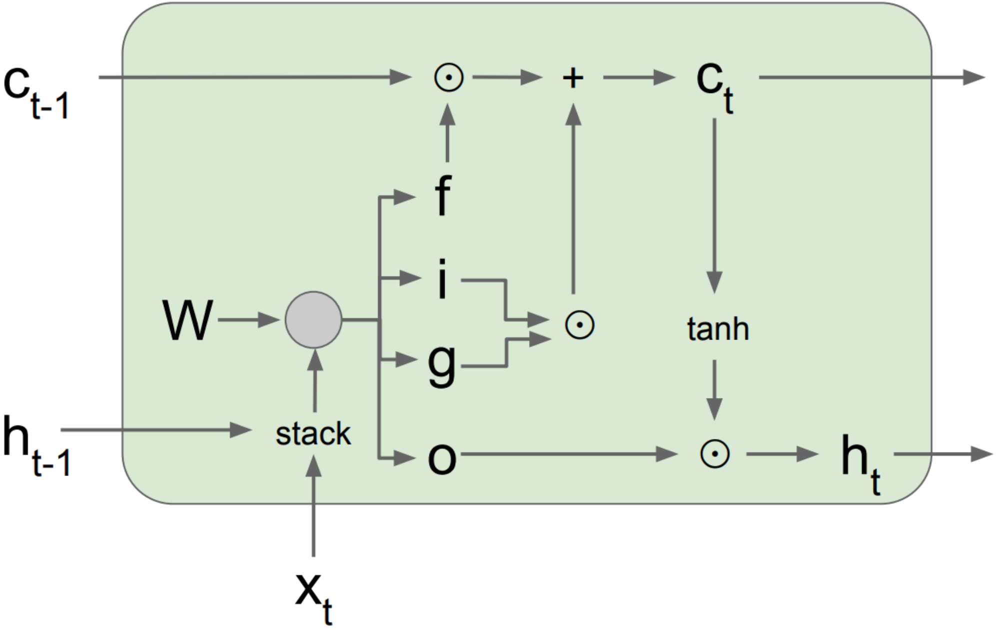
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \boxed{\sigma} \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = \boxed{o} \odot \tanh(c_t)$$

Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

Long Short Term Memory (LSTM)



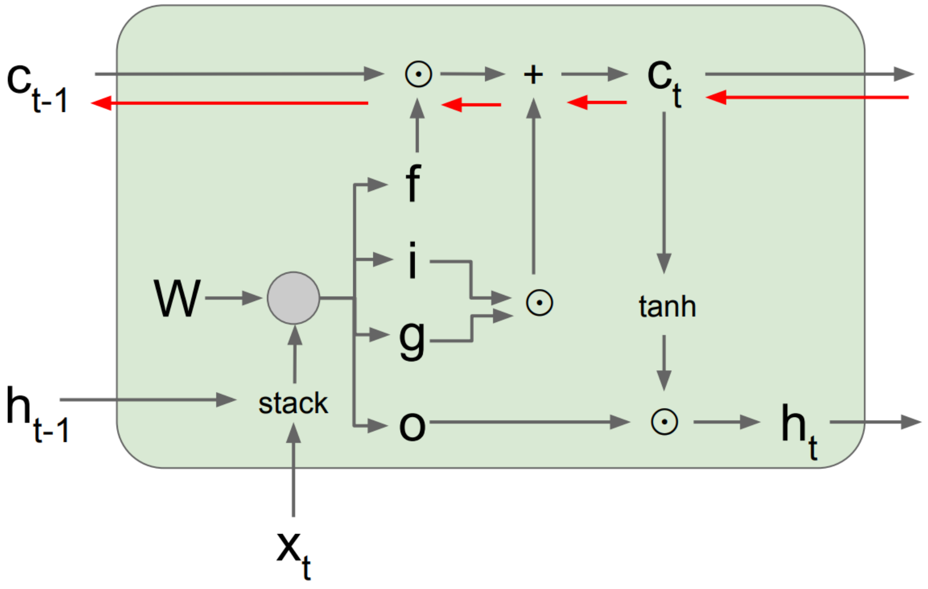
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

Long Short Term Memory (LSTM) : Gradient Flow



Backpropagation from c_t to c_{t-1} only elementwise multiplication by f, no matrix multiply by W

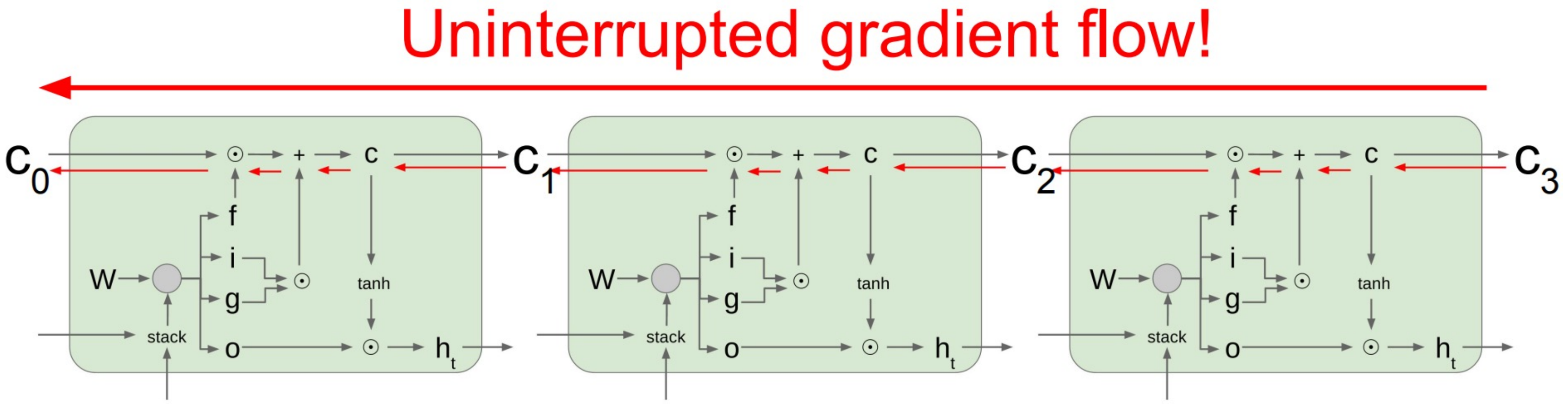
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

Long Short Term Memory (LSTM) : Gradient Flow



Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

Do LSTMs solve the vanishing gradient problem?

- The LSTM architecture makes it easier for the RNN to preserve information over many timesteps
  - e.g. if the $\boldsymbol{f} = \mathbf{1}$ and the $\boldsymbol{i} = \mathbf{0}$, then the information of that cell is preserved indefinitely.
  - By contrast, it's harder for vanilla RNN to learn a recurrent weight matrix Wh that preserves info in hidden state
- LSTM doesn't guarantee that there is no vanishing/exploding gradient, but it does provide an easier way for the model to learn long-distance dependencies

Summary

- RNNs allow a lot of flexibility in architecture design
- Vanilla RNNs are simple but don't work very well
- Backward flow of gradients in RNN can explode or vanish.
- Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)
- Common to use LSTM or GRU: their additive interactions improve gradient flow
- Better/simpler architectures are a hot topic of current research, as well as new paradigms for reasoning over sequences
- Better understanding (both theoretical and empirical) is needed.