Amity School of Engineering and Technology

B.Tech. Computer Science and Engineering

Semester VI

Generative Artificial Intelligence (Generative AI)

[AIML 303]

Faculty: Dr Rinki Gupta

Module IV (20%)

Use of Recurrent Neural Networks in Generative AI

- Recurrent Neural Networks (RNNs)

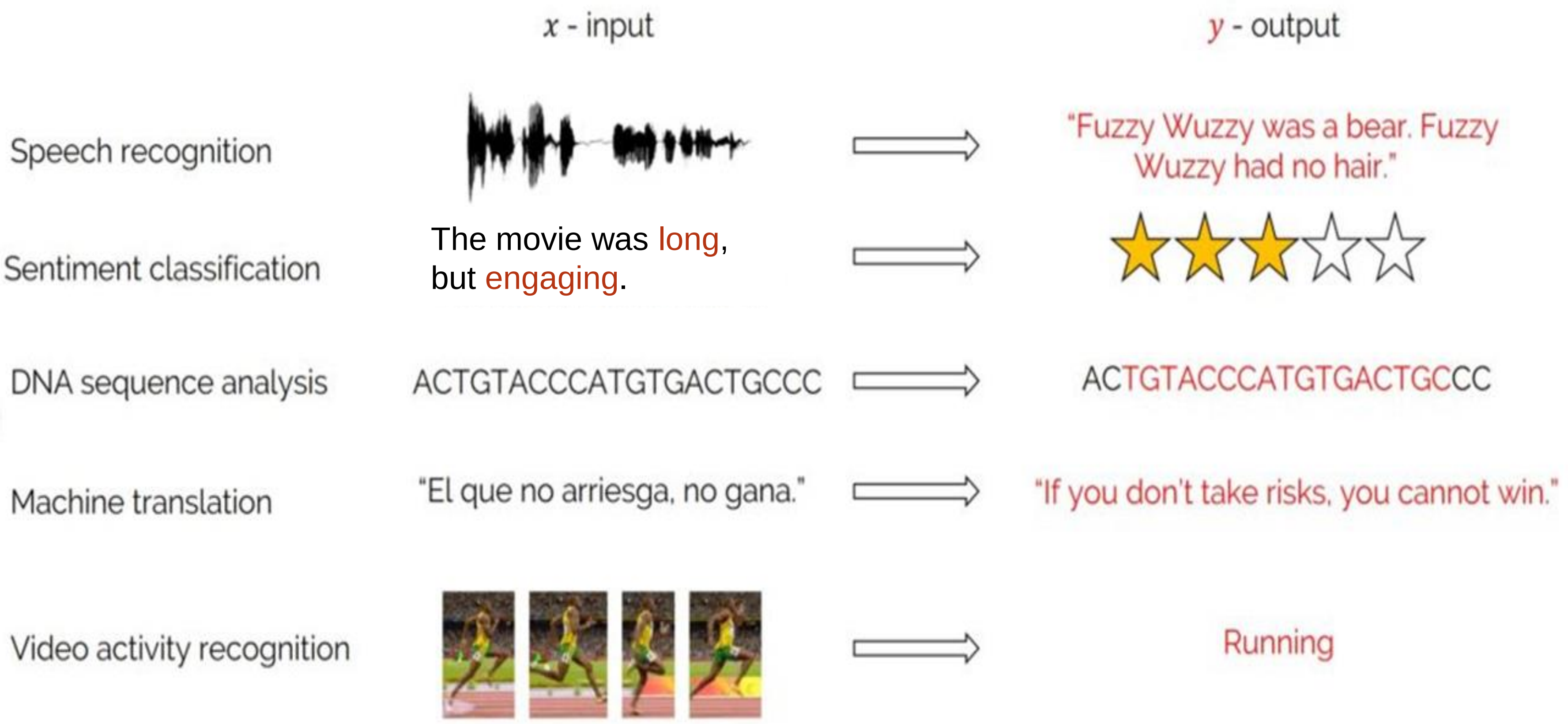
Ref: https://www.shiksha.com/online-courses/articles/introduction-to-recurrent-neural-network/

- Sequential models for solving Long-term dependency issues
- Applications of RNNs in Generative AI:
  - Text generation with RNNs
  - Music generation using RNNs
  - Speech synthesis and recognition

Sequence Learning

Data having a sequential order that needs to be followed to understand it
Inputs are related to each other

- I/P Audio and O/P transcript, both are sequences
- I/P sequence of text => o/p sentiment (positive, negative, angry, etc.). or rating
- I/P DNA sequence, predict which part belongs to which protein
- Sentence in one language -> another language
- I/p sequence of frames -> predict the activity



Can we use ANN/CNN for ***Sequential Modeling***?

Can we use ANN/CNN for ***Sequential Modeling***?

No ❌

Reasons:

- Fixed input size

Example : image size

32x32

Reasons:

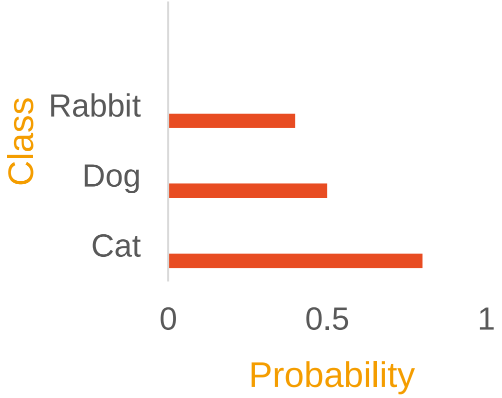
- Fixed input size

  Example : image size

- Fixed output size

  Example : probabilities of different classes

32x32



Reasons:

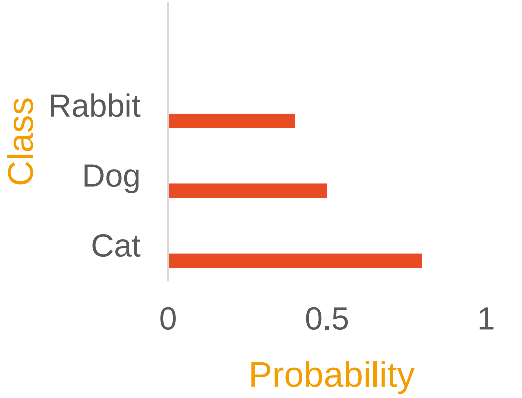
- Fixed input size

  Example : image size

32x32

- Fixed output size

  Example : probabilities of different classes



- Fixed computational steps

  Example : number of layers in the model

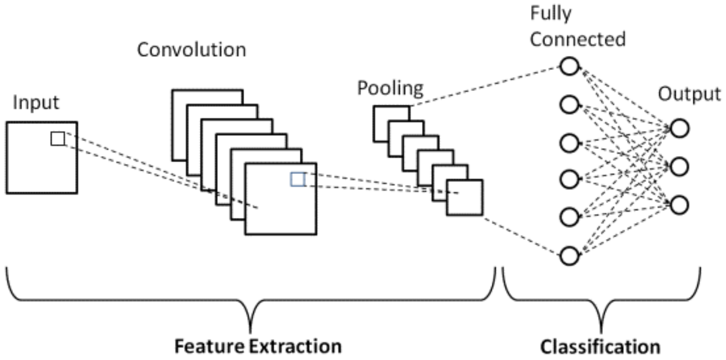


Image source: https://medium.com/techiepedia/binary-image-classifier-cnn-using-tensorflow-a3f5d6746697

Reasons:

- Words learned or approximated at a later position may change the approximation of a previous word.

  Example :

  - Blue dresses are looking good.
  - Blue dress is looking good.

- Parameter sharing is not done in conventional ANNs.

Reasons:

- Words learned or approximated at a later position may change the approximation of a previous word.

  Example :

  - Blue dresses are looking good.
  - Blue dress is looking good.

- Parameter sharing is not done in conventional ANNs.

There comes ***Recurrent Neural Network!***

Reasons for Using Recurrent Neural Network (RNN)

- Can handle inputs and outputs of **varying lengths**.

Reasons for Using Recurrent Neural Network (RNN)

- Can handle inputs and outputs of **varying lengths**.
- It involves **directed cycles** to recognize sequential characteristics of a data.

Reasons for Using Recurrent Neural Network (RNN)

- Can handle inputs and outputs of **varying lengths**.
- It involves **directed cycles** to recognize sequential characteristics of a data.
- **Shares parameters** across different parts of the network.

Reasons for Using Recurrent Neural Network (RNN)

- Can handle inputs and outputs of **varying lengths**.
- It involves **directed cycles** to recognize sequential characteristics of a data.
- **Shares parameters** across different parts of the network.
- Track **long-term** dependencies.

Reasons for Using Recurrent Neural Network (RNN)

- Can handle inputs and outputs of **varying lengths**.
- It involves **directed cycles** to recognize sequential characteristics of a data.
- **Shares parameters** across different parts of the network.
- Track **long-term** dependencies.
- Maintain information about **order**.

When to use RNN?

"Whenever there is a sequence of data and the temporal dynamics that connects the data is more important than the spatial content of each individual frame."

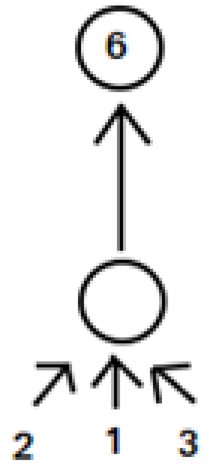
– Lex Fridman (MIT)

Image source:
https://commons.wikimedia.org/wiki/File:Lex_Fridman_teaching_at_MIT_in_2018.png

Why Recurrent Connection for Sequence Learning?

- Feed-forward neural network trained to predict sum of three numbers
  - Fixed input dimension

Input	Output
2+3+1	6

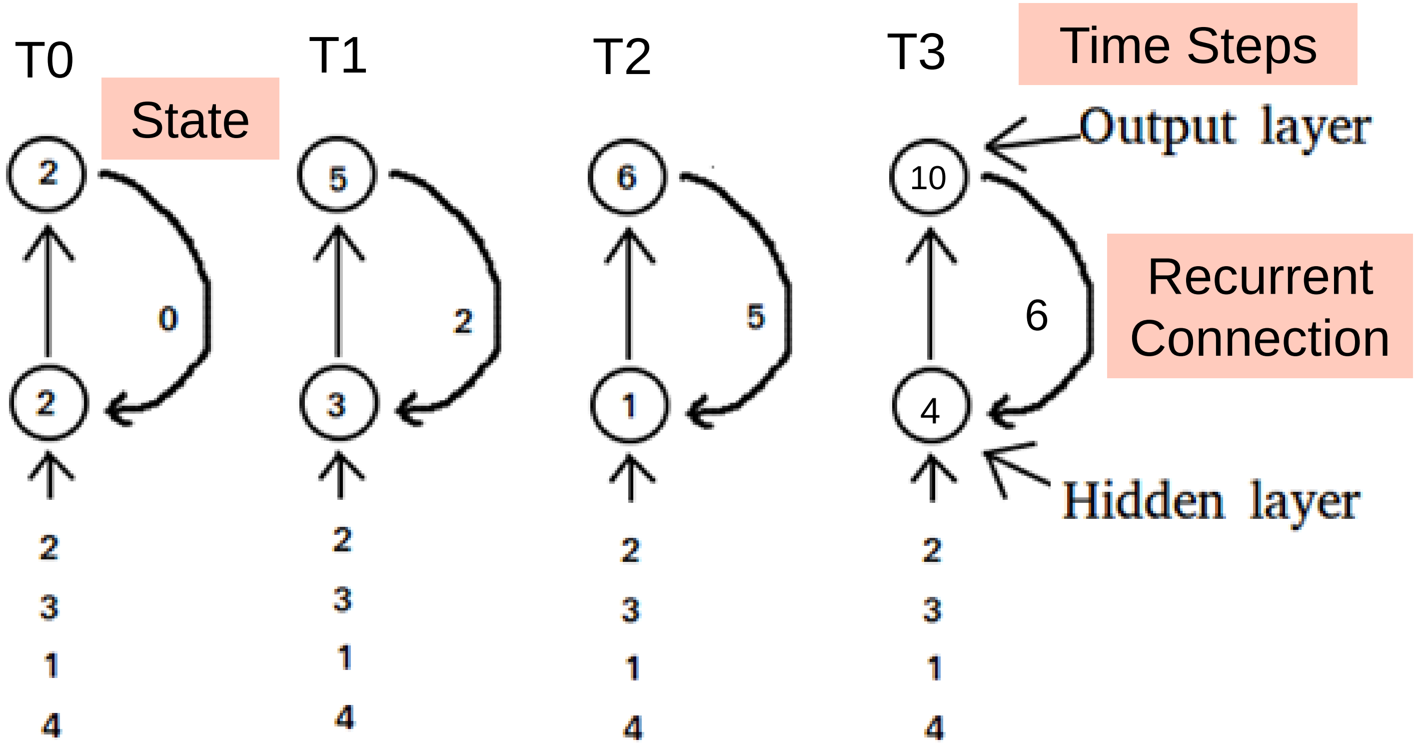


- What if Now we want to add 4 numbers? 2+3+1+4 = 10

Why Recurrent Connection for Sequence Learning?

- Recurrent neural network
  - Can be generalized to addition of more inputs
  - Can handle varying input length

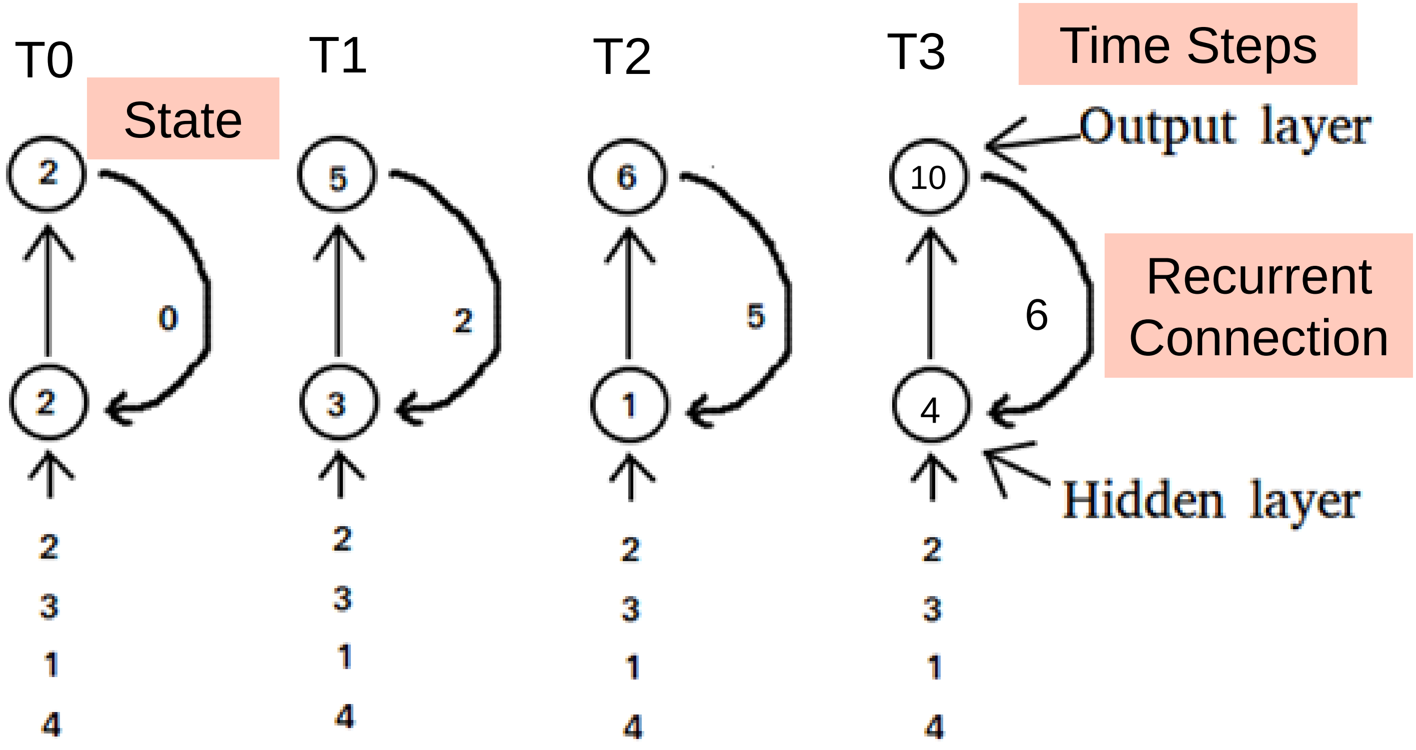
Input	Output
0+2	2
2+3	5
5+1	6
6+4	10



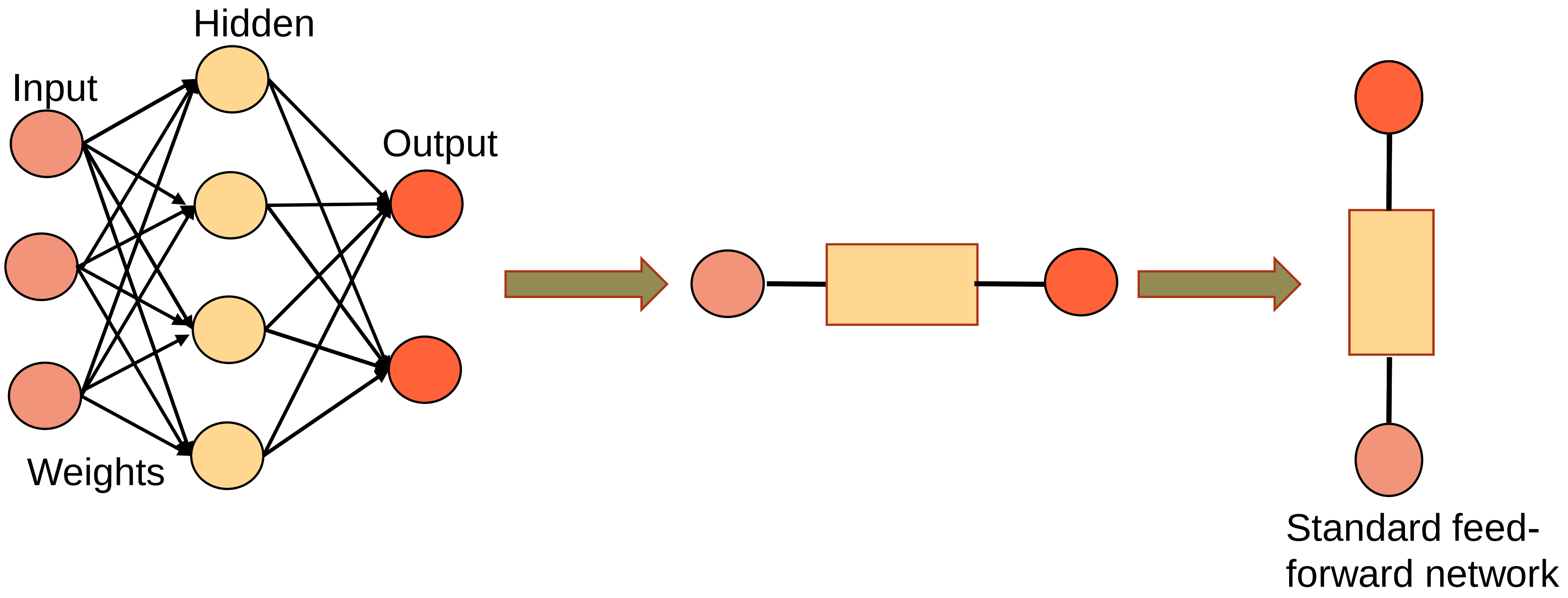
Why Recurrent Connection for Sequence Learning?

- RNNs operate on a sequence that contains vectors $x(t)$ with time step index $t \in \{0, \tau\}$
- Cycles represent the influence of the present value of a variable on its own value at a future time step
- Outputs from previous time steps are fed as input to the current time step ->

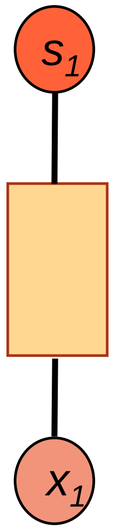
  Neural Network with Memory



Neural Network: Simplified



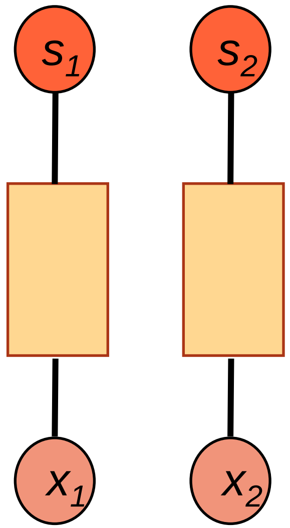
Handling Individual Time Steps



We have seen a feed-forward network for single time step.

How can we handle a sequence of inputs using this network?

Handling Individual Time Steps

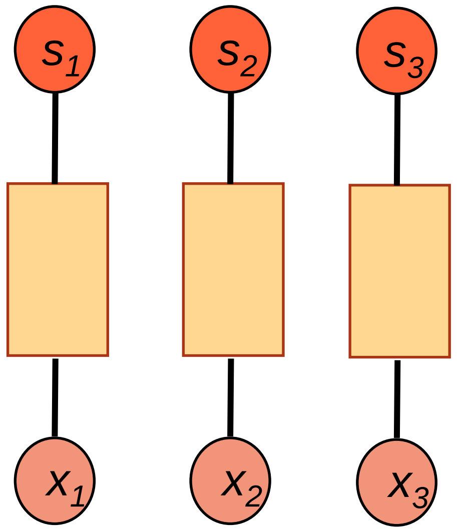


We have seen a feed-forward network for single time step.

How can we handle a sequence of inputs using this network?

We can copy this network and repeat the operations multiple times to try to handle inputs that are fed in corresponding to different times.

Handling Individual Time Steps

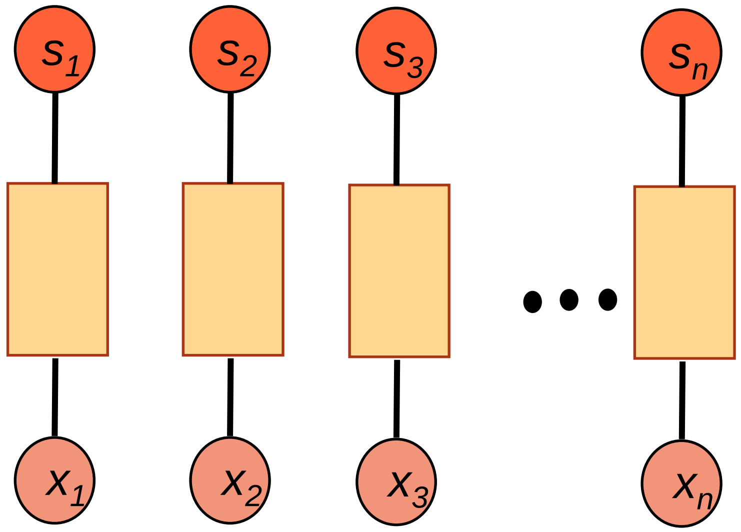


We have seen a feed-forward network for single time step.

How can we handle a sequence of inputs using this network?

We can copy this network and repeat the operations multiple times to try to handle inputs that are fed in corresponding to different times.

Handling Individual Time Steps

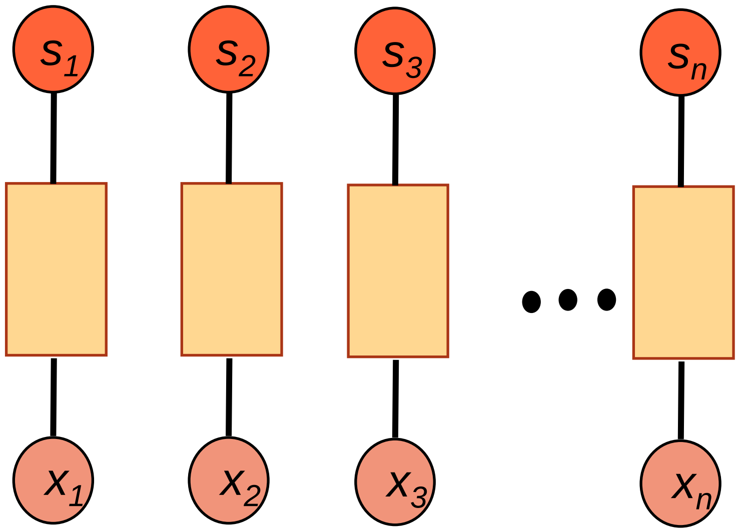


We have seen a feed-forward network for single time step.

How can we handle a sequence of inputs using this network?

We can copy this network and repeat the operations multiple times to try to handle inputs that are fed in corresponding to different times.

Handling Individual Time Steps



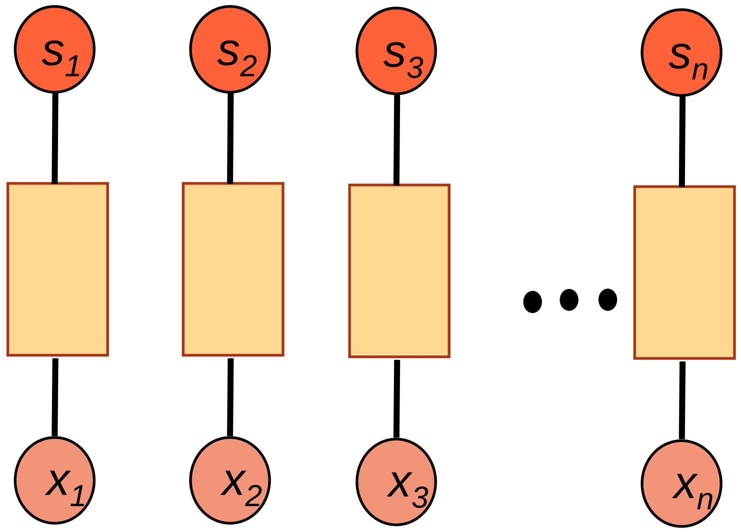
In general,

$$s_n = f(x_n)$$

n = time step

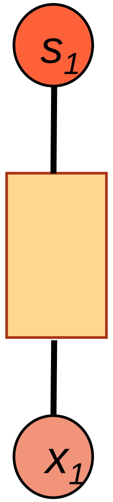
- Same function is used
- Replicate network any number of times
- Ensure parameter sharing
- Number of timestep does not matter

Handling Individual Time Steps



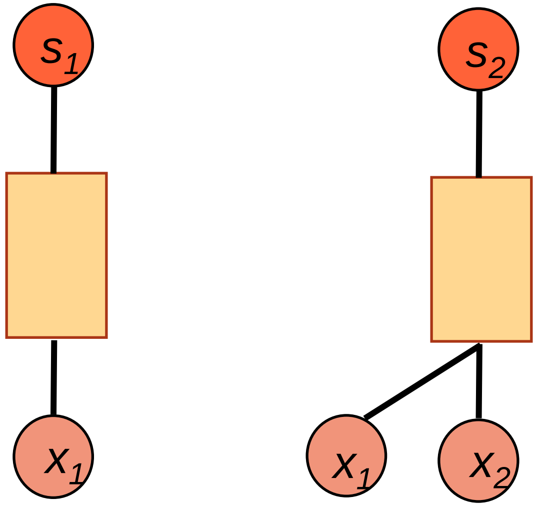
How to maintain the interdependency between input?

- In the sequential problems, the outputs predicted at the latter stage is dependent on the previous input.
- However, the independent replicas will not ensure the temporal dependencies between the inputs.
- How do we consider the interdependency between inputs?



A Simple Approach

Let’s consider one approach

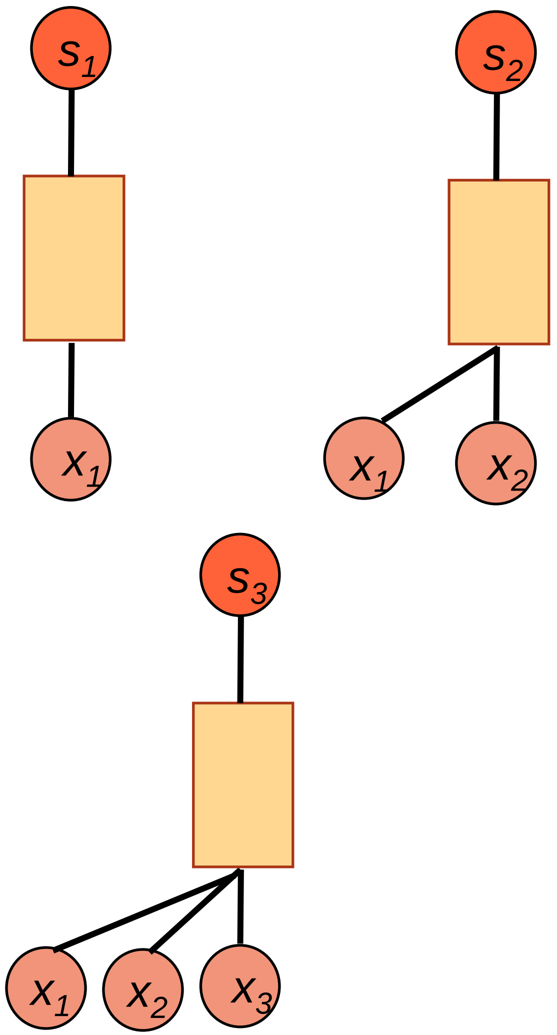


A Simple Approach

Let's consider one approach

We can feed all the previous inputs to the network at each timestep

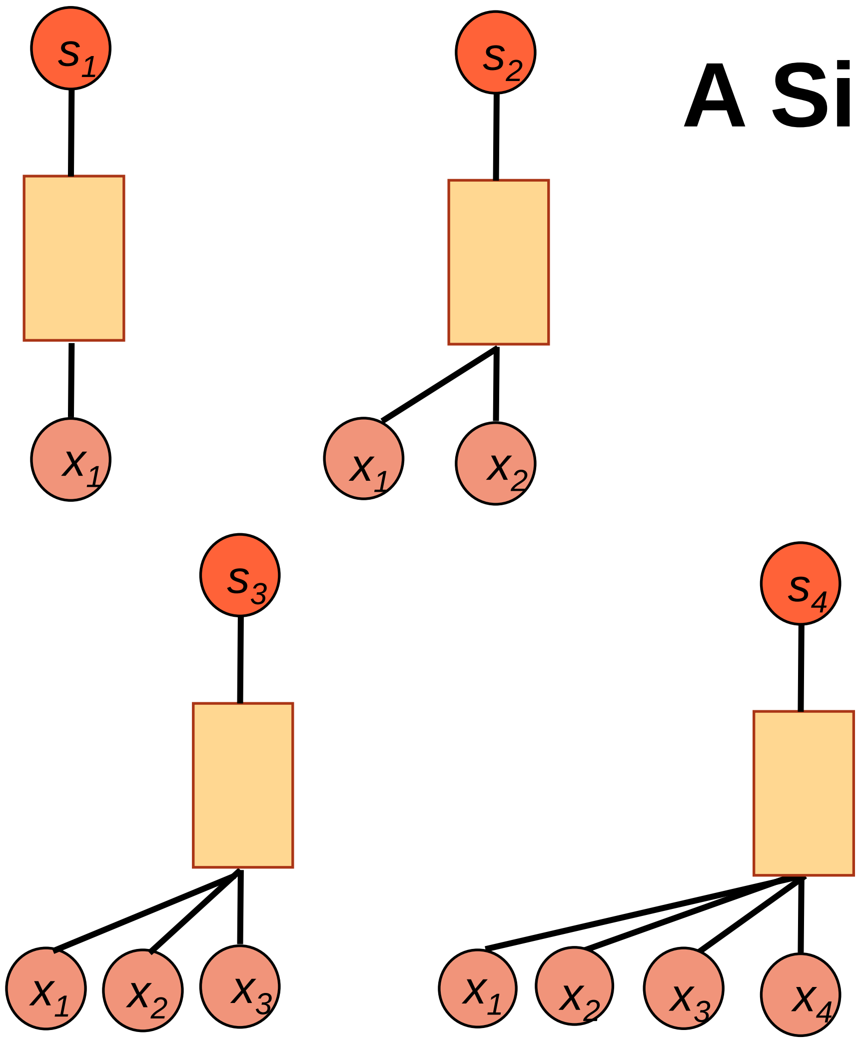
A Simple Approach



Let’s consider one approach

We can feed all the previous inputs to the network at each timestep

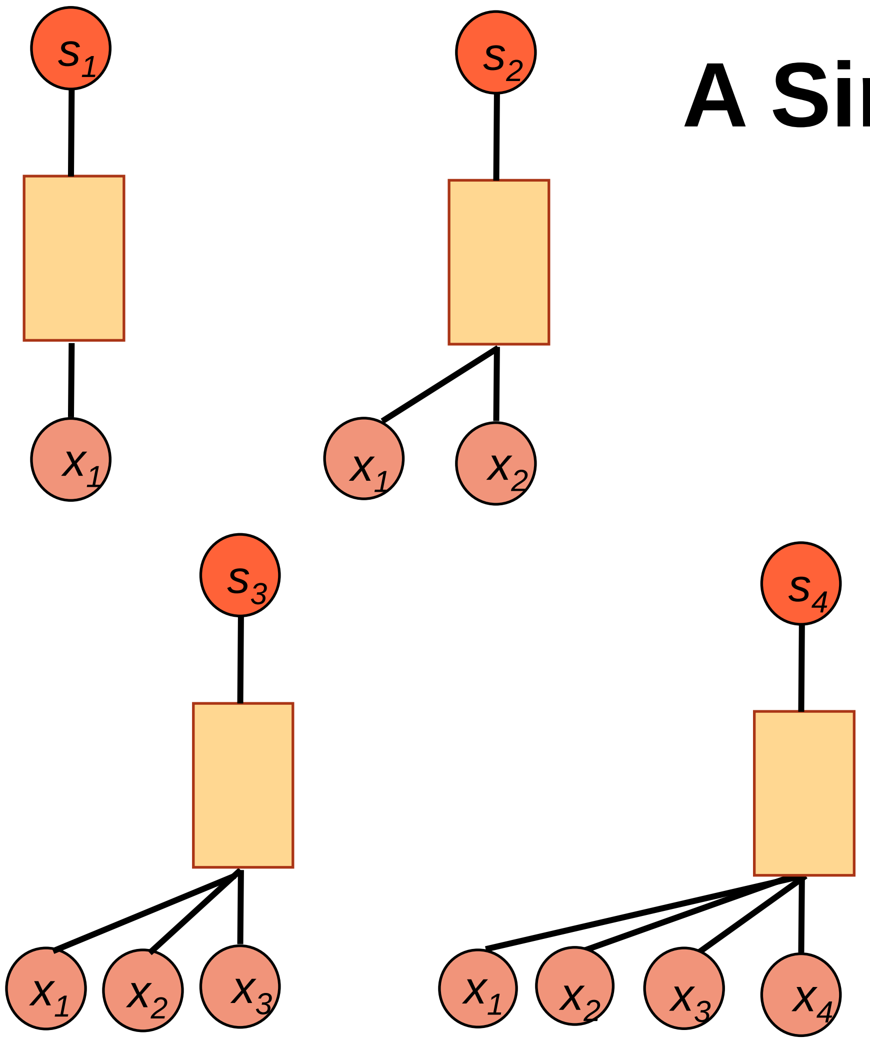
A Simple Approach



Let’s consider one approach

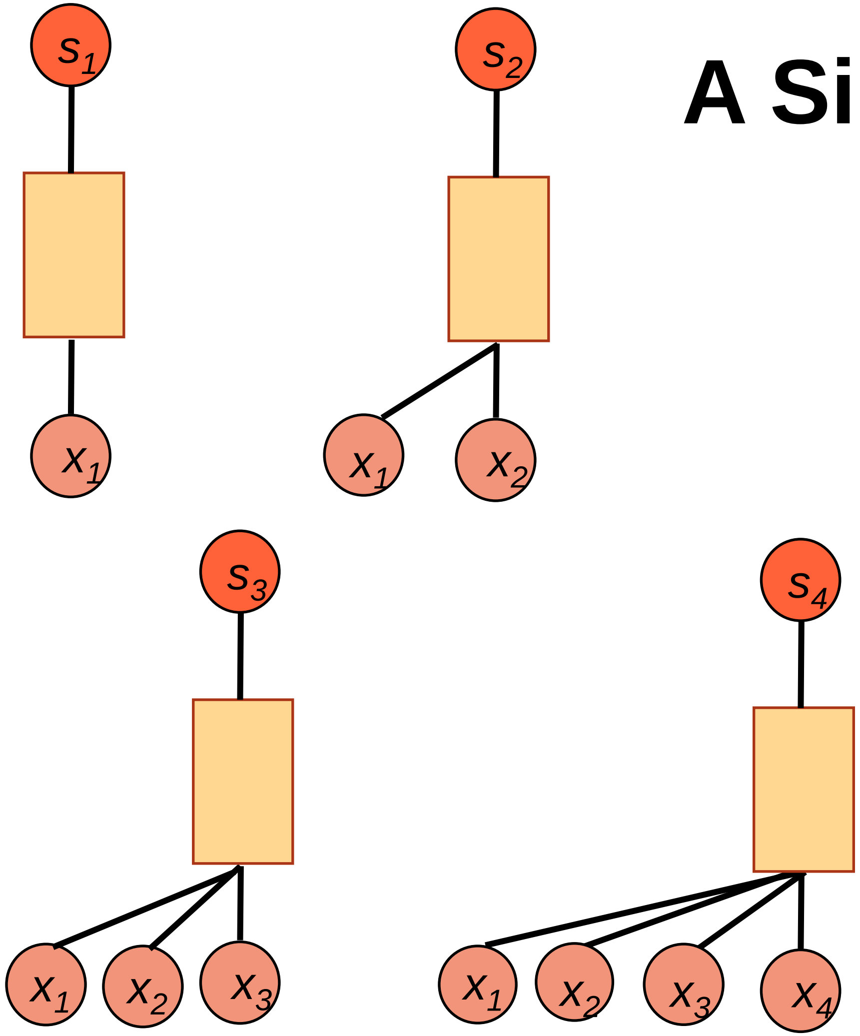
We can feed all the previous inputs to the network at each timestep

A Simple Approach



Will this approach work?

A Simple Approach

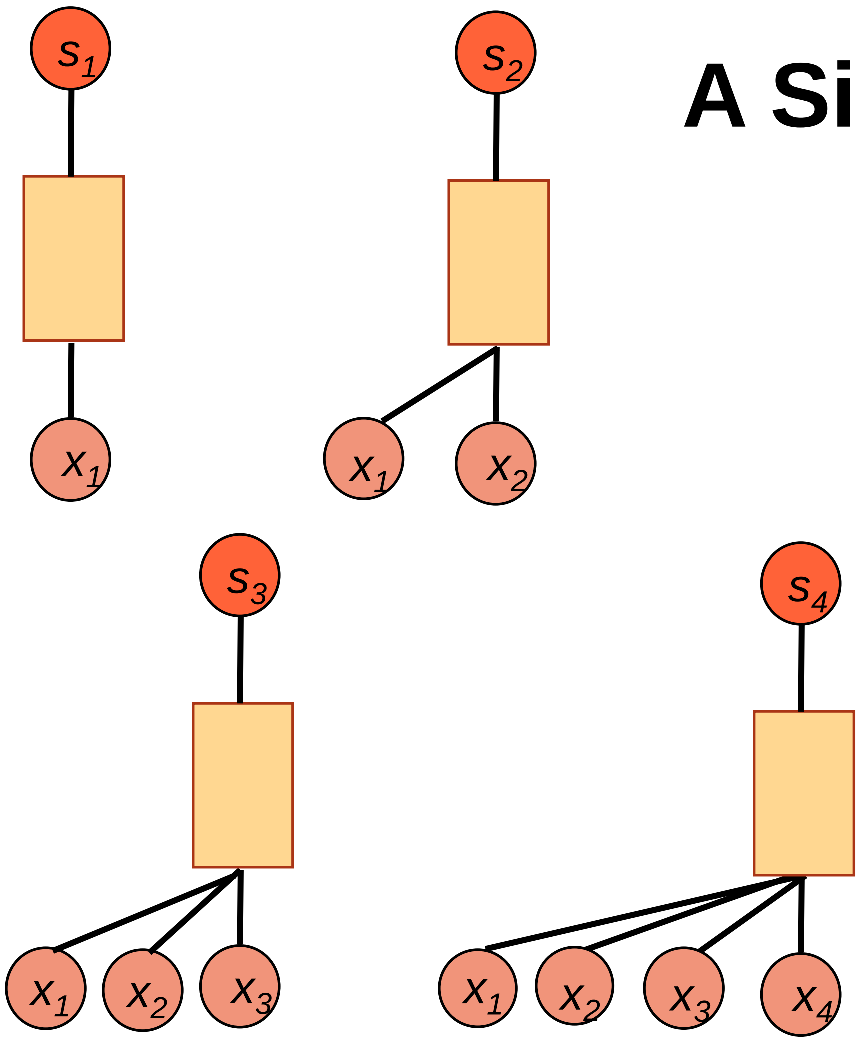


Problem

- Different function for different time-step

$s_1 = f_1(x_1)$
$s_2 = f_2(x_1,x_2)$
$s_3 = f_3(x_1,x_2,x_3)$ ……

A Simple Approach



Problem

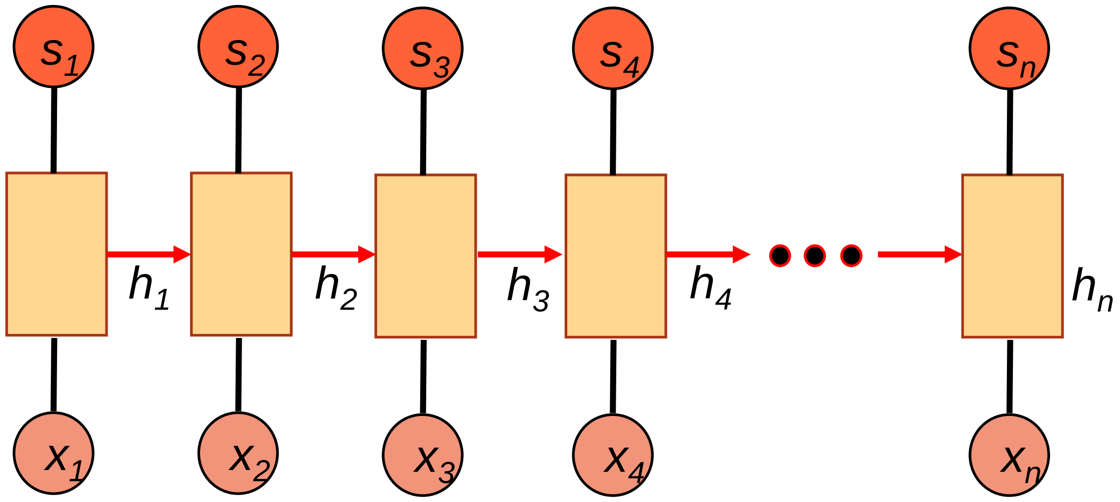
- Different function for different time-step

  $s_1 = f_1(x_1)$

  $s_2 = f_2(x_1,x_2)$

  $s_3 = f_3(x_1,x_2,x_3)$ ……
- Depends on input length

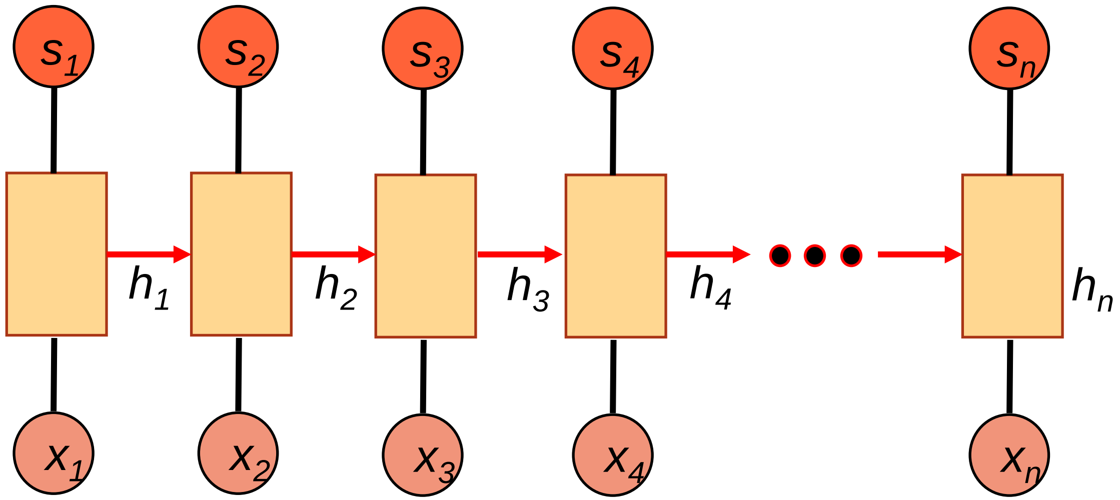
Recurrent Neural Network



Solution

Add recurrent connection

Recurrent Neural Network



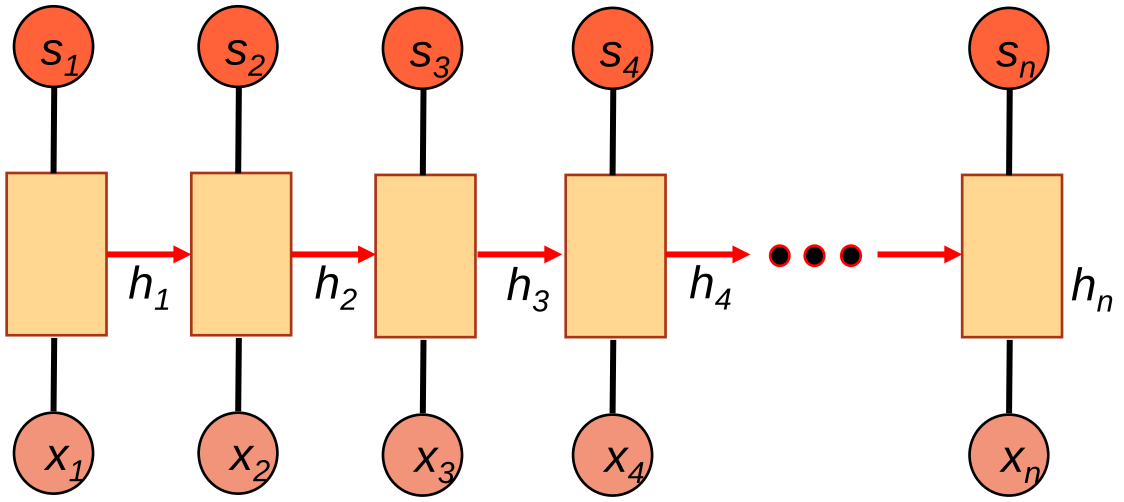
Solution

Add recurrent connection

$$s_n = f(x_n, h_{n-1})$$

- the **same function** is executed at each timestep

Recurrent Neural Network



Solution

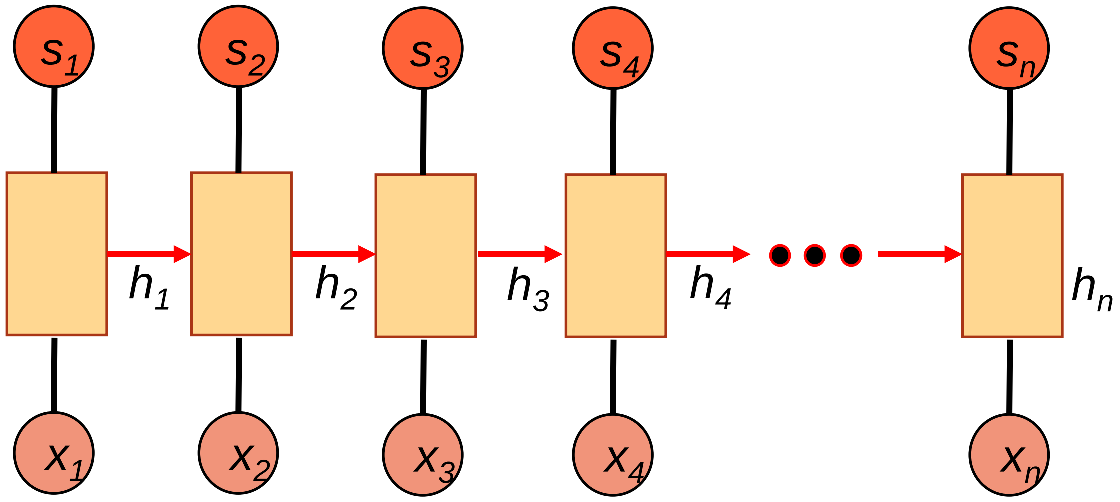
Add recurrent connection

$$s_n = f(x_n, h_{n-1})$$

input

- the **same function** is executed at each timestep
- New input x_n at each time step

Recurrent Neural Network



Solution

Add recurrent connection

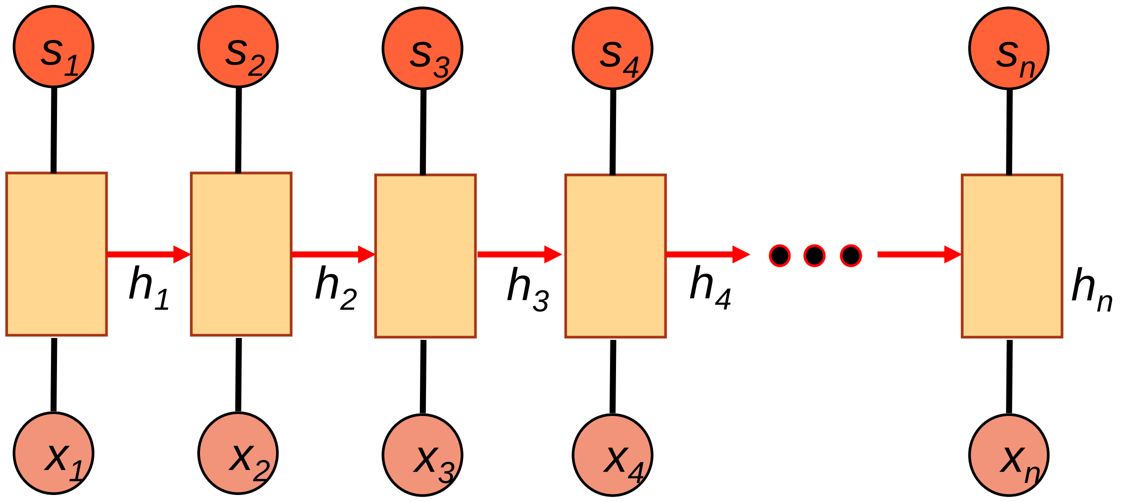
$$s_n = f(x_n, h_{n-1})$$

output

input

- the **same function** is executed at each timestep
- New input x_n at each time step
- Output s_n is generated by applying **same function** *f* at each time step

Recurrent Neural Network



Solution

Add recurrent connection

$$s_n = f(x_n, h_{n-1})$$

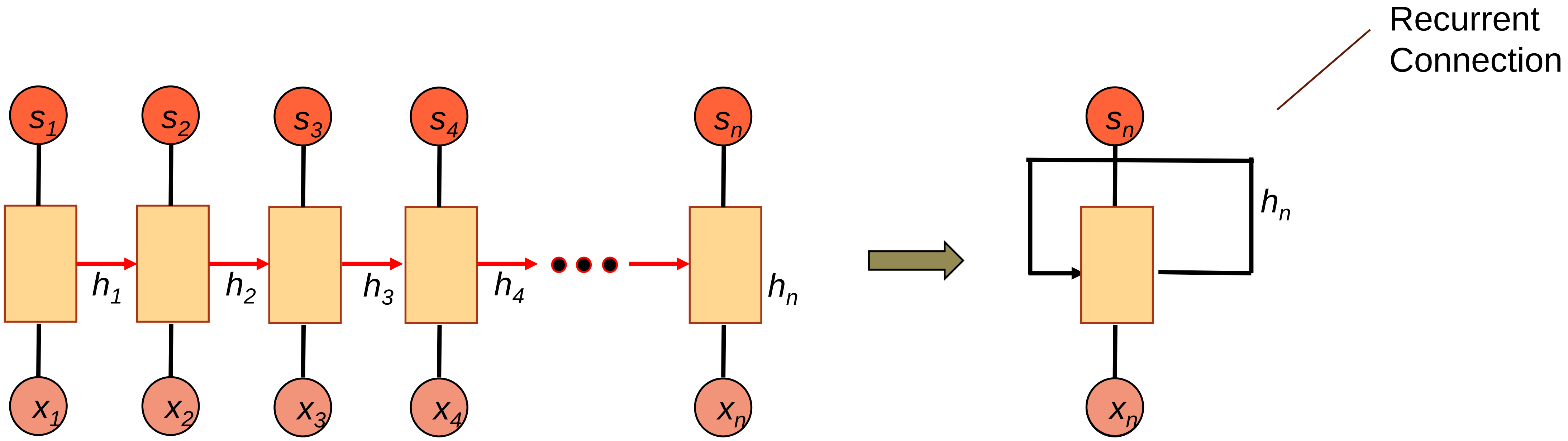
output

input

past memory

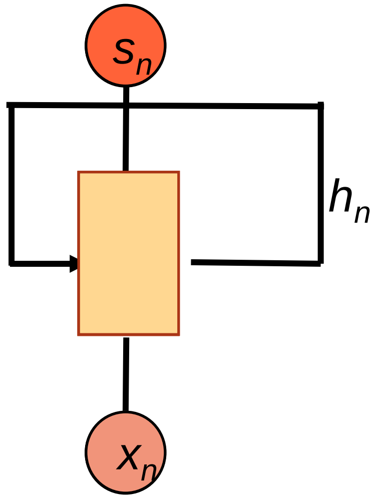
- the **same function** is executed at each timestep
- New input x_n at each time step
- Output s_n is generated by applying **same function** *f* at each time step
- At each time step, h_n is updated as a sequence of input is processed

Recurrent Neural Network



Can be represented more compactly

Recurrent Neural Network



A sequence of vectors is processed by applying a recurrence formula at each time step.

$$h_n = f_W(x_n, h_{n-1})$$

Hidden state

Function with weight W

Input

Old hidden state

- Same function is used
- Ensure parameter sharing
- Handles the temporal dependency between sequence

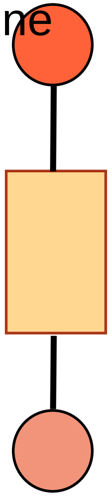
Recurrent Neural Network Architectures

RNN architectures are named based on the number of inputs and outputs.

Standard neural network go from input to output in one direction and they are not able to maintain information about the previous events in a sequence of events.

Recurrent Neural Network Architectures

one to one

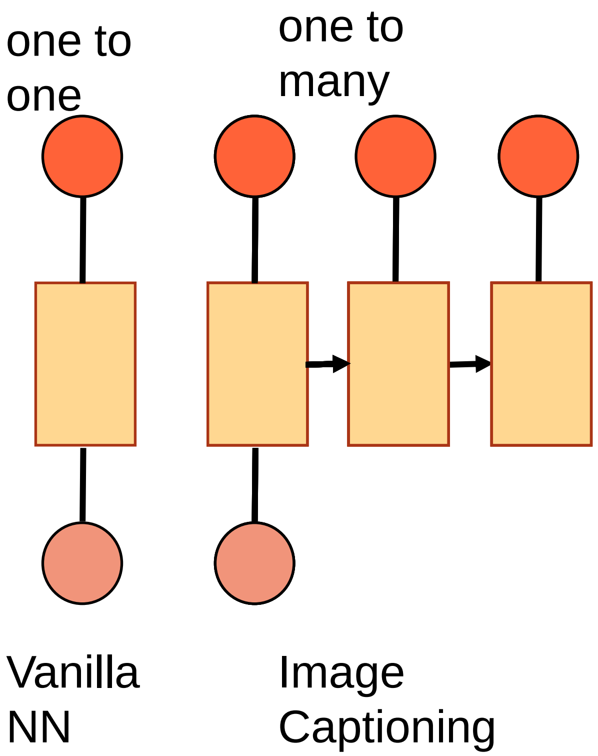


Vanilla NN

RNN architectures are named based on the number of inputs and outputs.

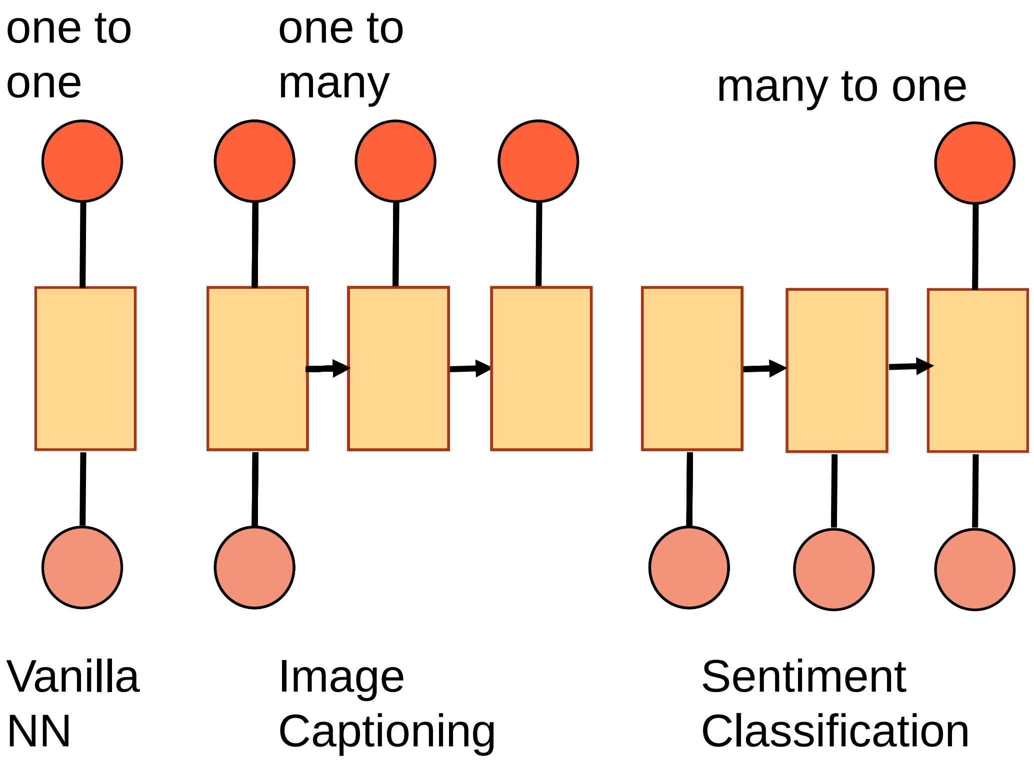
First is one to one architecture. Also known as Vanilla NN. It is used in machine learning problems such as regression and classification problems.

Recurrent Neural Network Architectures



Next is one to many RNN.
It is used in various applications such as image captioning.

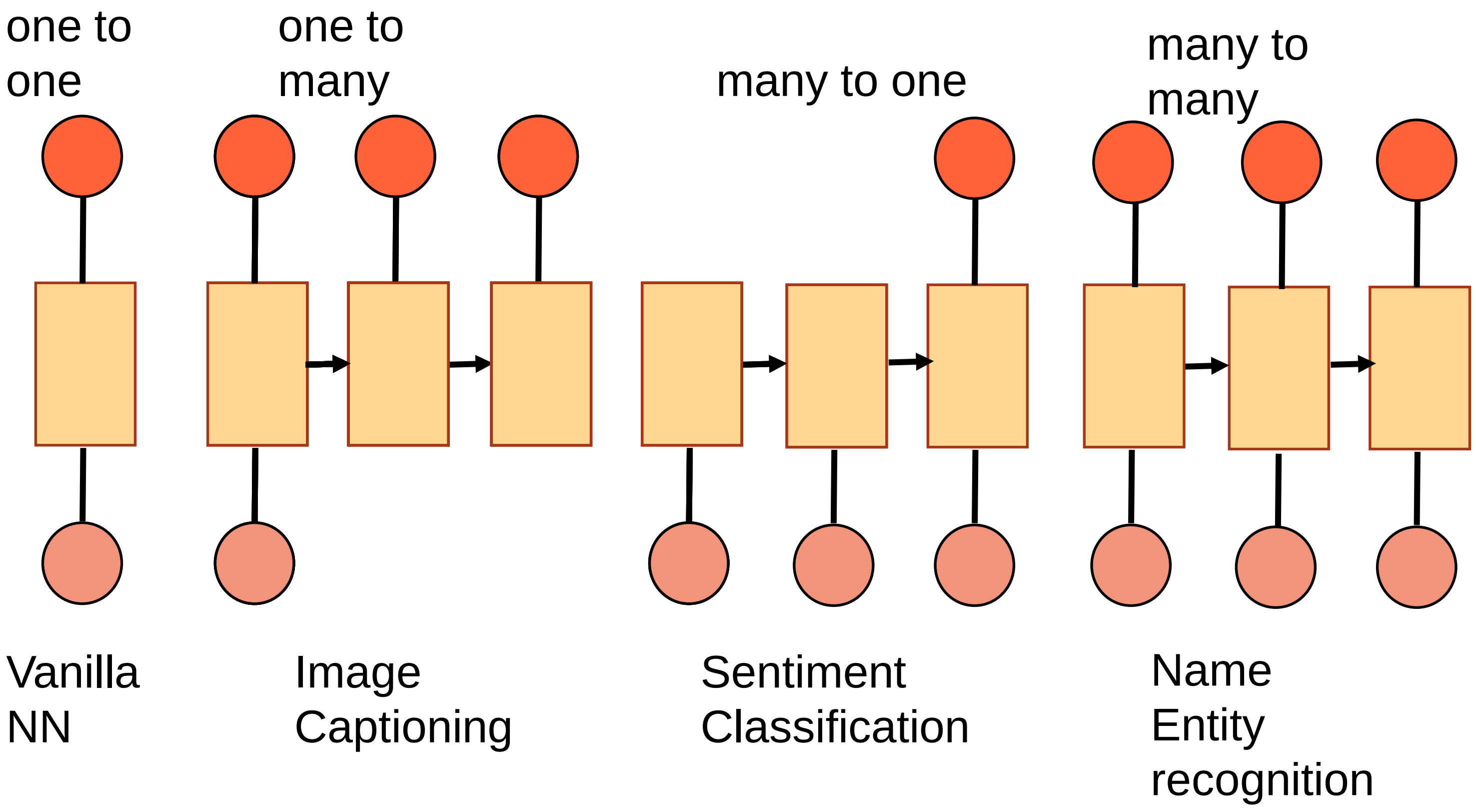
Recurrent Neural Network Architectures



Next is many to one RNN.

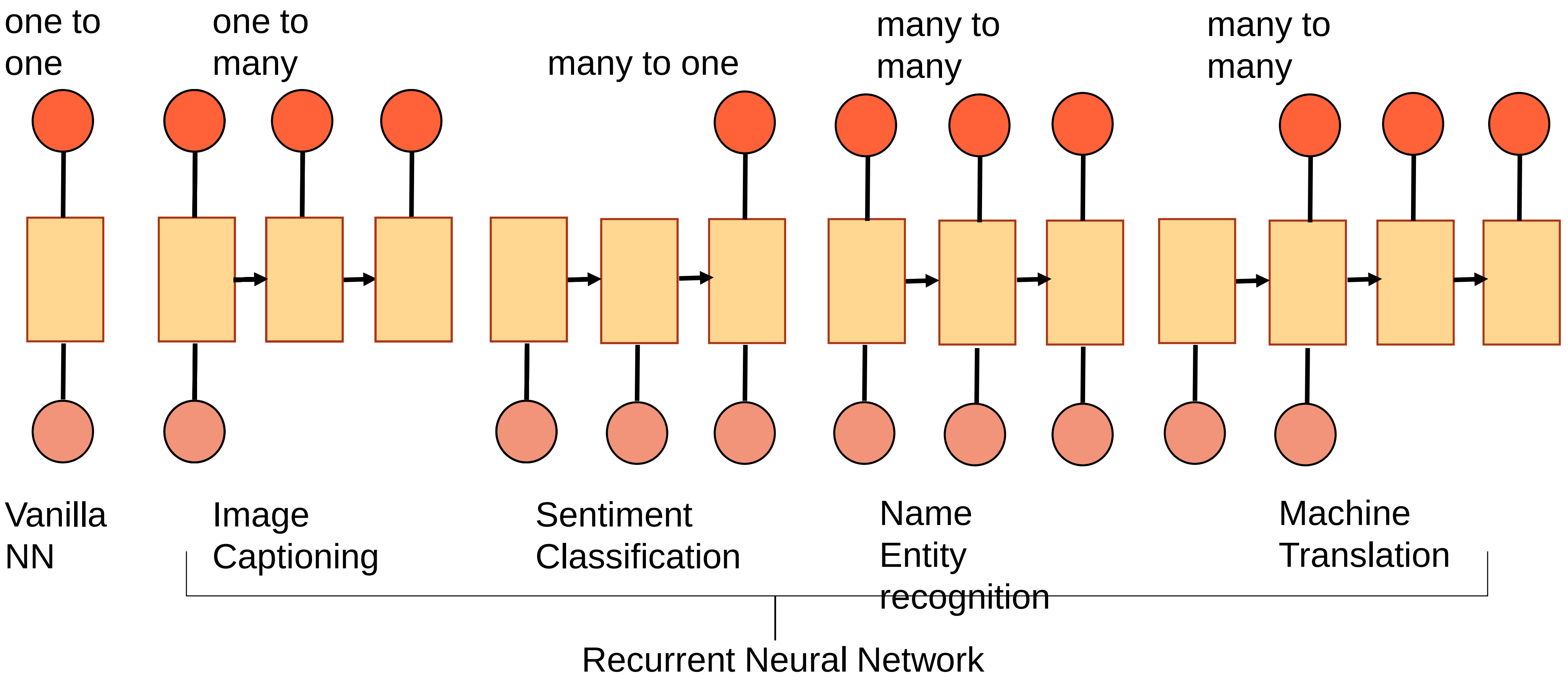
It is used in sentiment analysis.

Recurrent Neural Network Architectures



Next is many to many RNN. It is used for name entity recognition and machine translation.

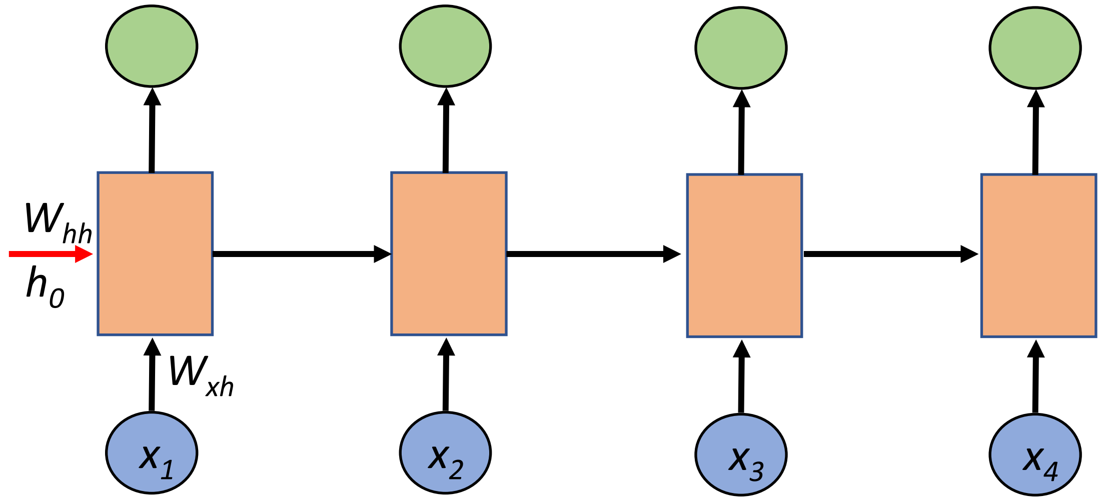
Recurrent Neural Network Architectures (with application examples)



RNN Training

1. Forward Pass
2. Backward Propagation Through Time (BPTT)

RNN: Forward Propagation



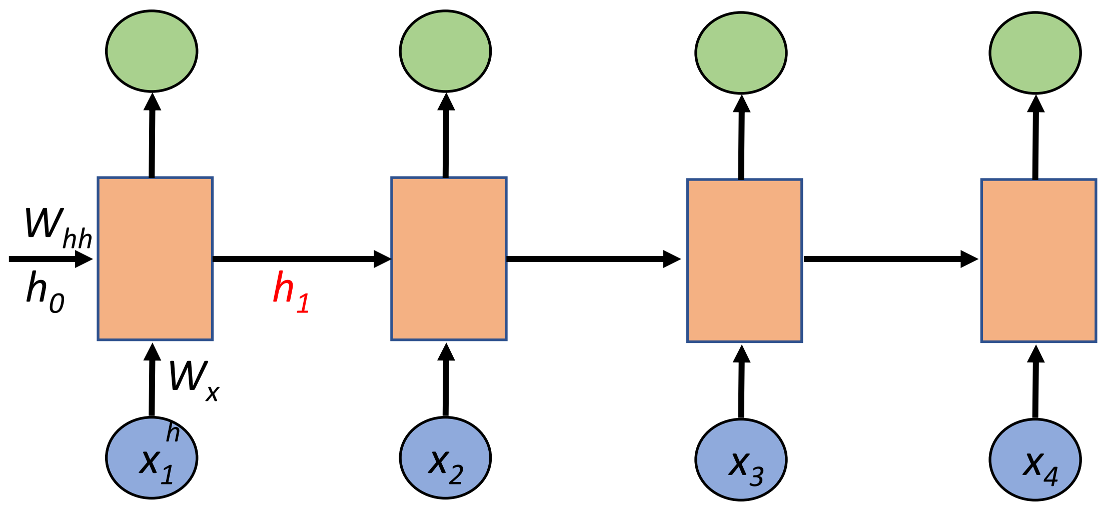
$$W_{xh}.x_1 + W_{hh}.h_0$$

First multiply input x_1 with corresponding weight matrix W_{xh}

Add previous hidden state h_0 weighted with weight matrix W_{hh}.

Since we don't have the values of h_0, we will use zero matrix initially

RNN: Forward Propagation



$$h_1 = g(W_{xh}.x_1 + W_{hh}.h_0 + b)$$

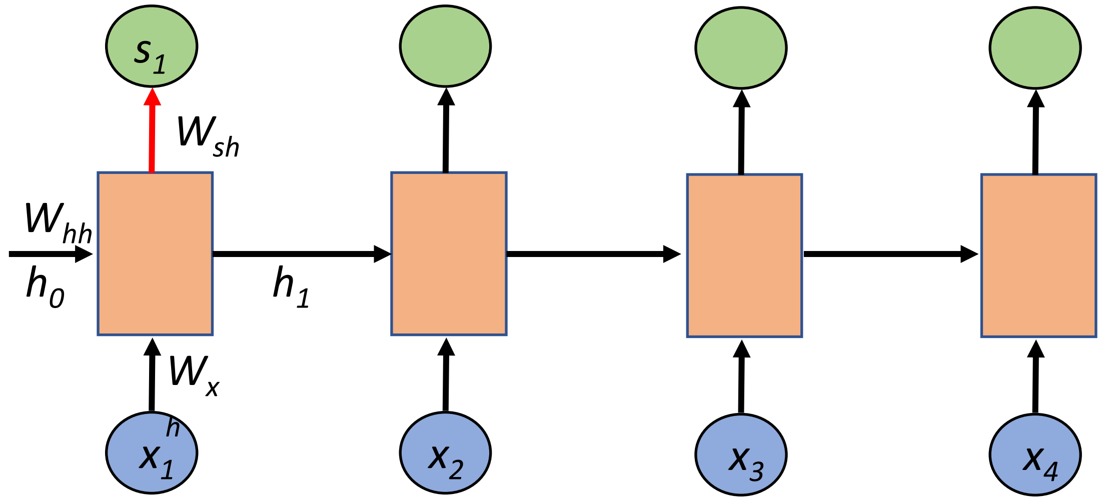
Activation function

Bias

Apply some activation function g (preferably tanh) will give us new hidden state h_1.

RNN: Forward Propagation

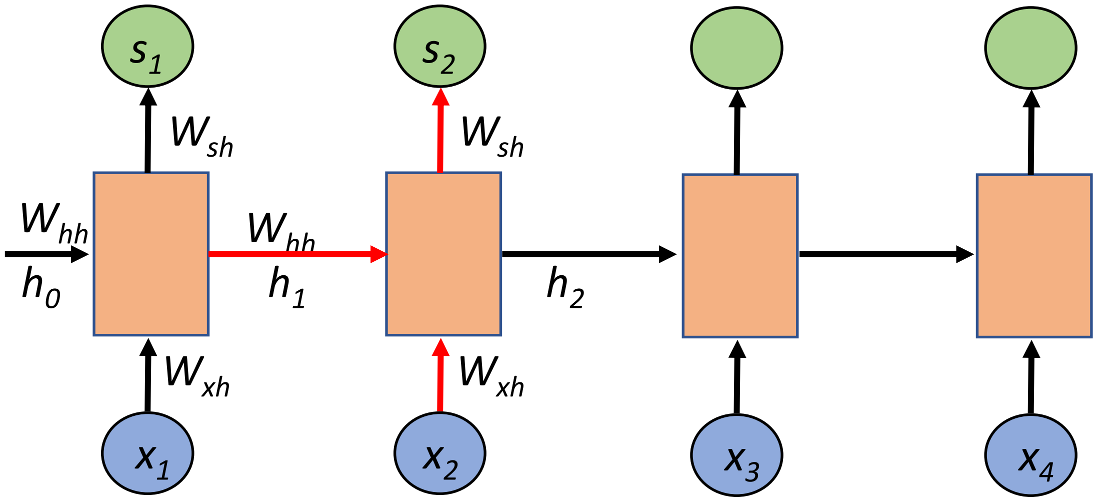
$$h_1 = g(W_{xh}.x_1 + W_{hh}.h_0 + b)$$
$$s_1 = \emptyset(W_{sh}.h_1 + c)$$



Again, multiply hidden state with another weight matrix W_{sh} and pass through a new activation function $\emptyset$ will result in a new output state s_1 for that particular time step.

x_1	Input
h_0	Previous hidden state
h_1	Current hidden state
W_{xh}, W_{hh}, W_{sh}	Shared parameters
$g, \emptyset$	Activation functions
b,c	Biases
s_1	Output state

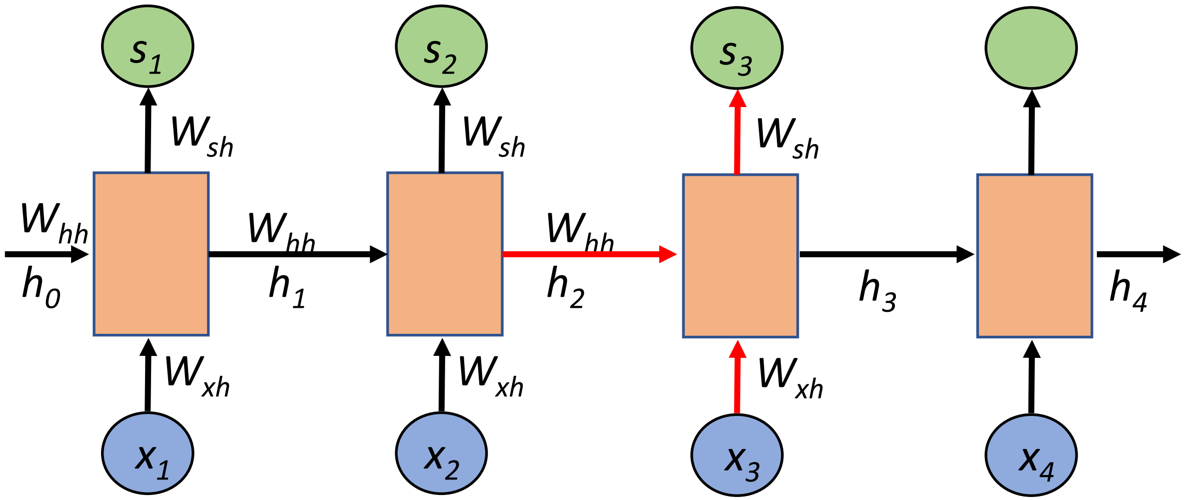
RNN: Forward Propagation



$$h_1 = g(W_{xh}.x_1 + W_{hh}.h_0 + b)$$
$$s_1 = \emptyset(W_{sh}.h_1 + c)$$
$$h_2 = g(W_{xh}.x_2 + W_{hh}.h_1 + b)$$
$$s_2 = \emptyset(W_{sh}.h_2 + c)$$

This process will repeat for all time steps.

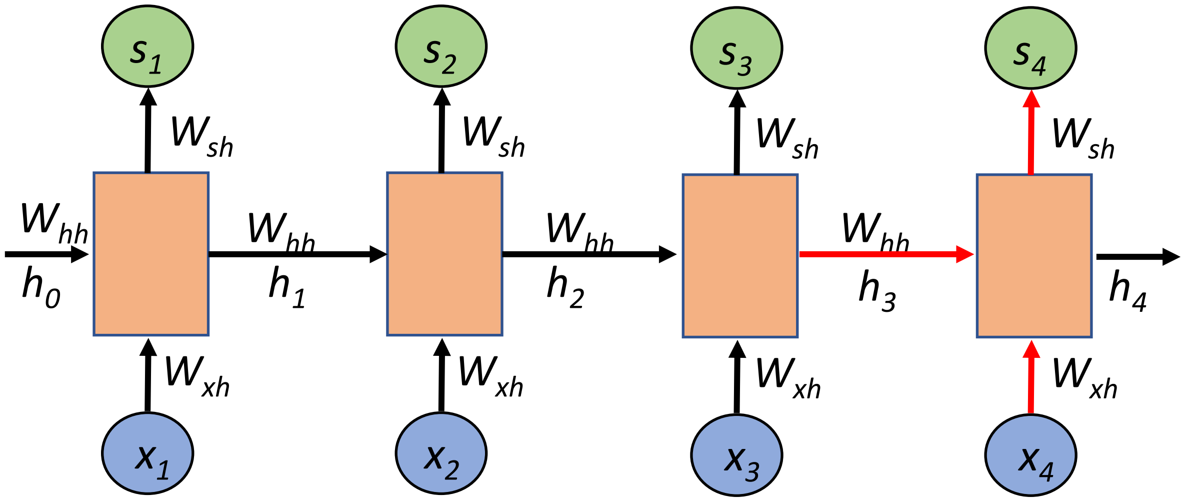
RNN: Forward Propagation



$$h_1 = g(W_{xh}.x_1 + W_{hh}.h_0 + b)$$
$$s_1 = \emptyset(W_{sh}.h_1 + c)$$
$$h_2 = g(W_{xh}.x_2 + W_{hh}.h_1 + b)$$
$$s_2 = \emptyset(W_{sh}.h_2 + c)$$
$$h_3 = g(W_{xh}.x_3 + W_{hh}.h_2 + b)$$
$$s_3 = \emptyset(W_{sh}.h_3 + c)$$

This process will repeat for all time steps.

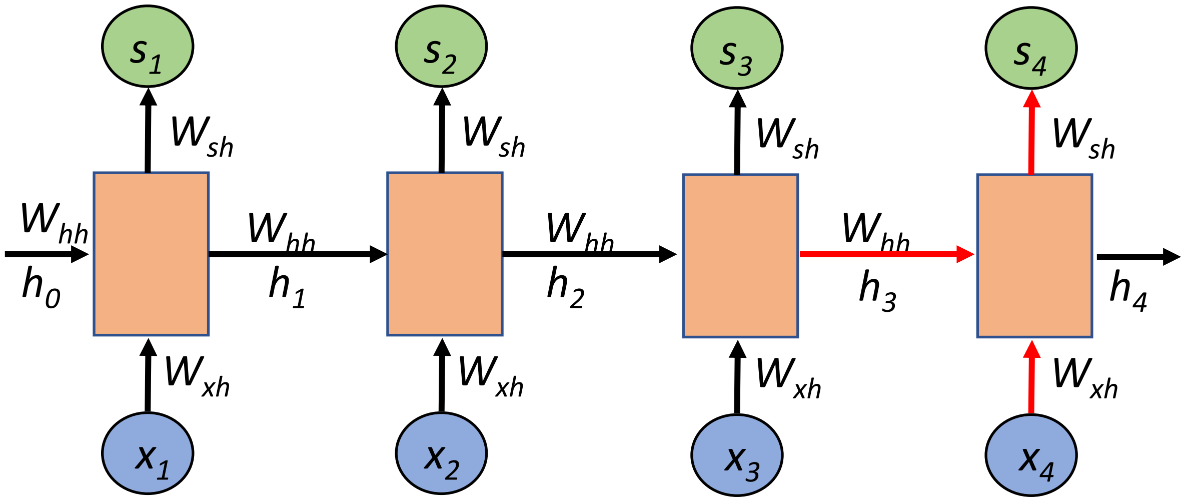
RNN: Forward Propagation



$$h_1 = g(W_{xh}.x_1 + W_{hh}.h_0 + b)$$
$$s_1 = \emptyset(W_{sh}.h_1 + c)$$
$$h_2 = g(W_{xh}.x_2 + W_{hh}.h_1 + b)$$
$$s_2 = \emptyset(W_{sh}.h_2 + c)$$
$$h_3 = g(W_{xh}.x_3 + W_{hh}.h_2 + b)$$
$$s_3 = \emptyset(W_{sh}.h_3 + c)$$
$$h_4 = g(W_{xh}.x_4 + W_{hh}.h_3 + b)$$
$$s_4 = \emptyset(W_{sh}.h_4 + c)$$

Weight matrix W_{xh} , W_{hh} and W_{sh} remains same throughout the forward propagation, thus ensuring parameter sharing

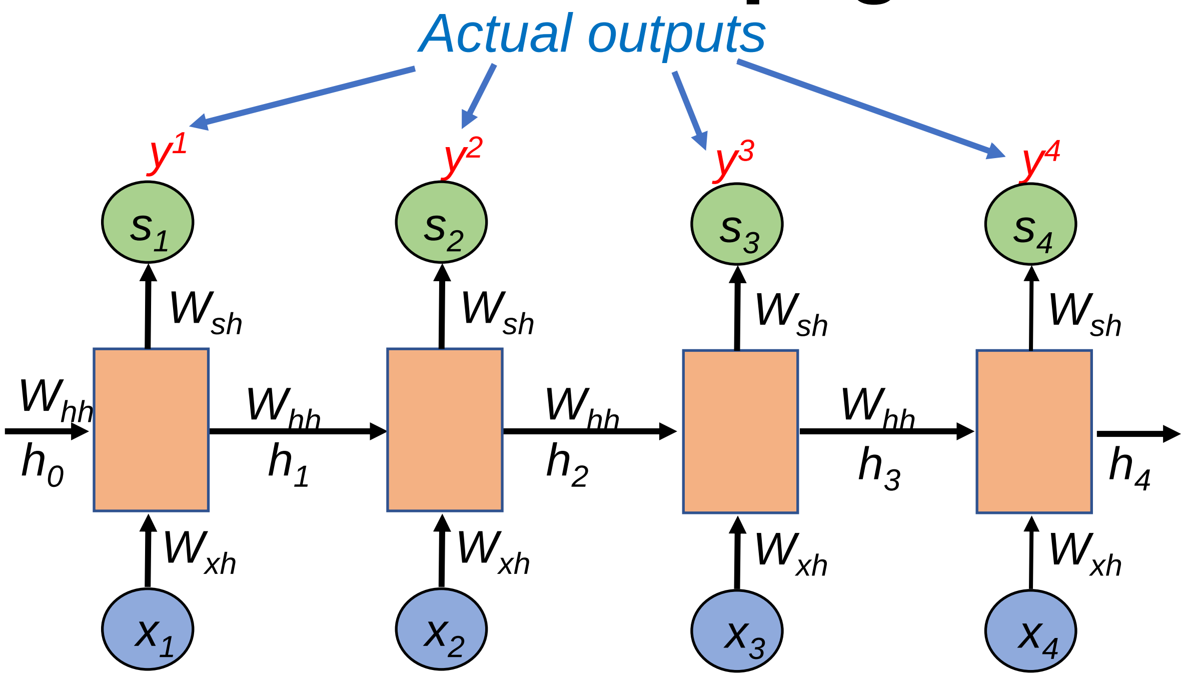
RNN: Forward Propagation



$$h_1 = g(W_{xh}.x_1 + W_{hh}.h_0 + b)$$
$$s_1 = \emptyset(W_{sh}.h_1 + c)$$
$$h_2 = g(W_{xh}.x_2 + W_{hh}.h_1 + b)$$
$$s_2 = \emptyset(W_{sh}.h_2 + c)$$
$$h_3 = g(W_{xh}.x_3 + W_{hh}.h_2 + b)$$
$$s_3 = \emptyset(W_{sh}.h_3 + c)$$
$$h_4 = g(W_{xh}.x_4 + W_{hh}.h_3 + b)$$
$$s_4 = \emptyset(W_{sh}.h_4 + c)$$

This parameter sharing also ensures that the network becomes agnostic to the length of the input. Because now whether we have 10 inputs or 20 inputs it does not matter, because at every time step the same function is executed. That is why it is important that at every time step we have the same function.

RNN: Back Propagation Through Time (BPTT)



After computing the output state for all time step. We need to find the loss with respect to the actual output.

Loss calculation:

Loss $L = \mathcal{L}(y_1, s_1) + \mathcal{L}(y_2, s_2) + \mathcal{L}(y_3, s_3) + \mathcal{L}(y_4, s_4)$

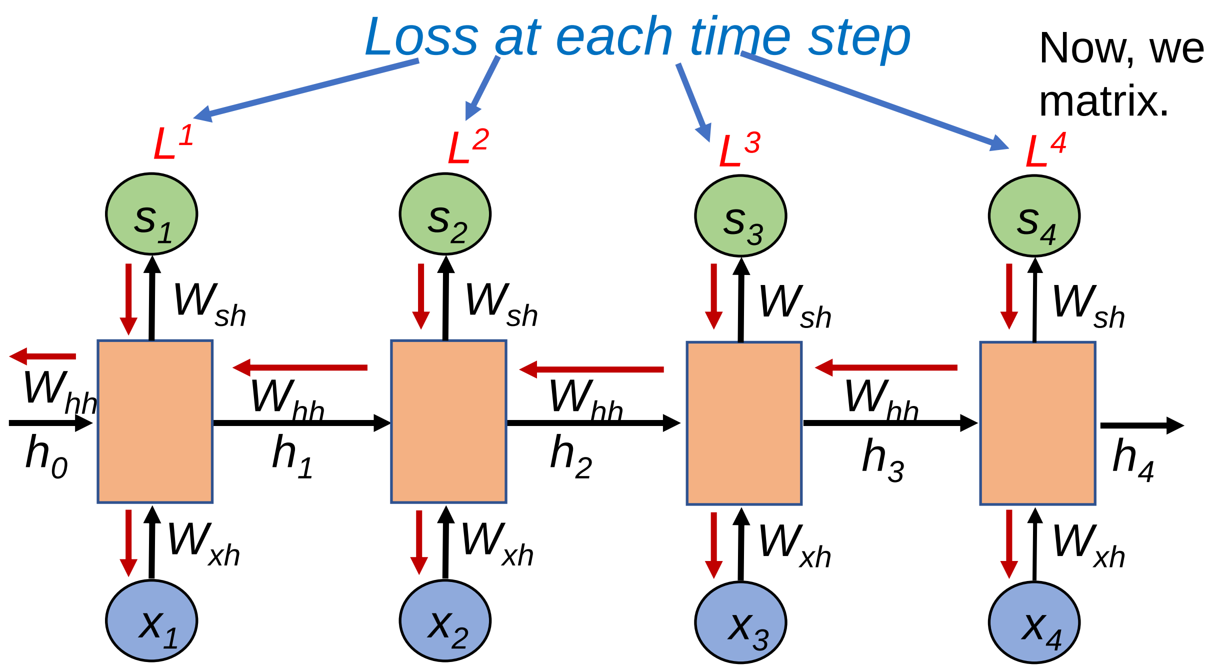
$\mathcal{L}$ = *Loss function*

In general:

$L = \sum_{i=1}^{n} \mathcal{L}(y_i, s_i)$

- Here, we can use any loss function according to our problem as well as the network design
- The loss of each time step will be computed separately.
- Finally, we will add all loses to generate a final loss L.

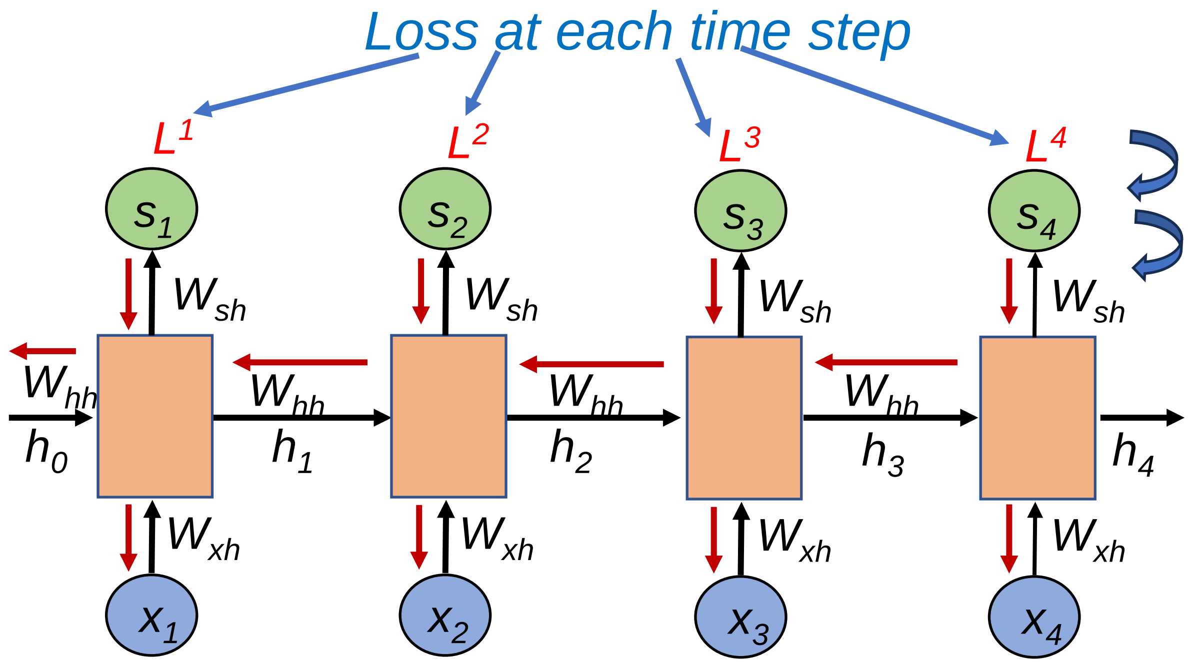
RNN: Back Propagation Through Time (BPTT)



Now, we will calculate gradients with respect to each weight matrix.

Gradient calculation wrt W_{sh} *:*

RNN: Back Propagation Through Time (BPTT)



Gradient calculation wrt W_{sh} :

$$\frac{\partial L^4}{\partial W_{sh}} = \frac{\partial L^4}{\partial s_4} \cdot \frac{\partial s_4}{\partial W_{sh}}$$

$$= -(y_4 - s_4).h_4$$

Weight updation wrt W_{sh} :

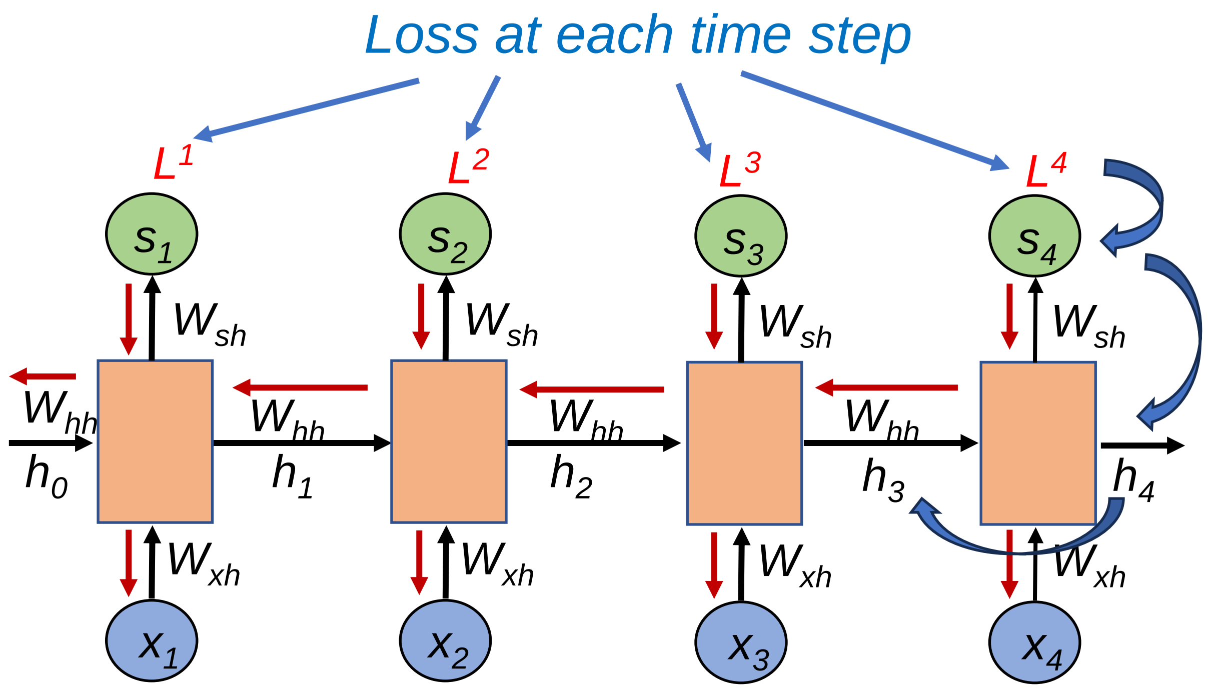
$$W_{sh} = W_{sh} - \sum_{i=1}^{n}(y_i - s_i).h_i$$

Note:

least square function $\mathcal{L} = \frac{1}{2}(y_4 - s_4)^2$

$$s_4 = W_{sh}.h_4$$

RNN: Back Propagation Through Time (BPTT)



Gradient calculation wrt W_{hh} :

$$\frac{\partial L^4}{\partial W_{hh}} = \frac{\partial L^4}{\partial s_4} \cdot \frac{\partial s_4}{\partial h_4} \cdot \frac{\partial h_4}{\partial W_{hh}}$$

$$= -(y_4 - s_4). W_{sh}. \frac{\partial h_4}{\partial W_{hh}}$$

Now, $h_4 = g(W_{xh}.x_4 + W_{hh}.h_3)$

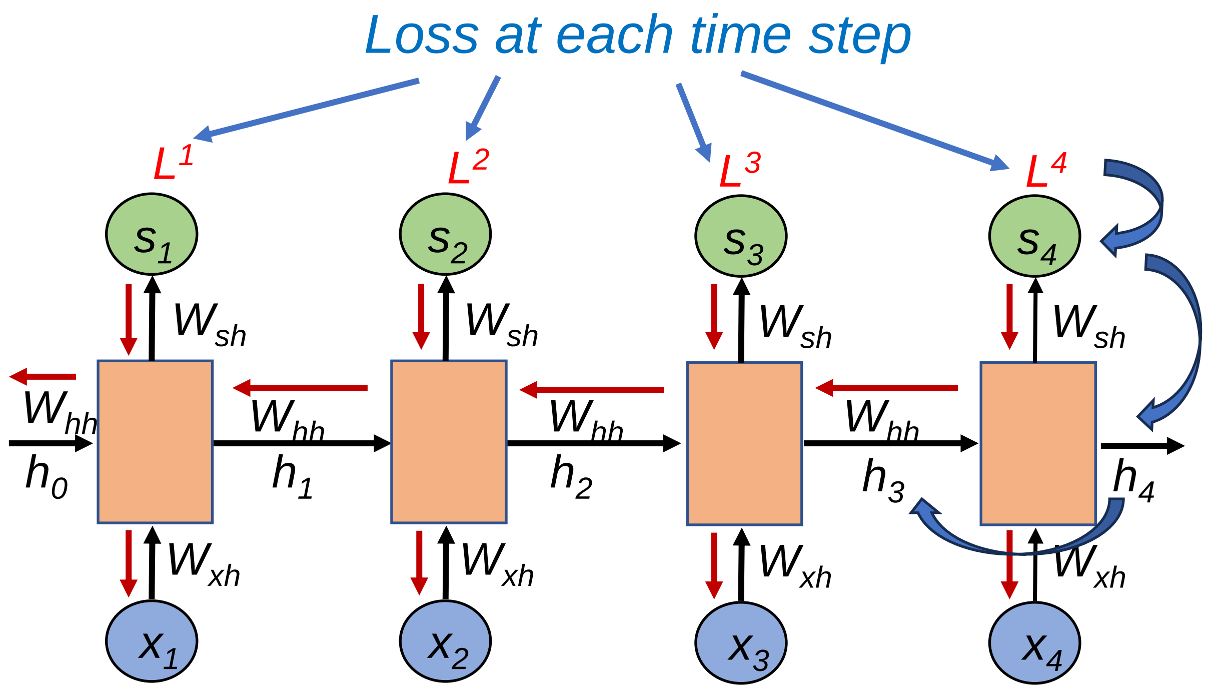
Then, $\frac{\partial h_4}{\partial W_{hh}} = g'.[h_3 + \frac{\partial h_3}{\partial W_{hh}}]$

Note:

least square function $\mathcal{L} = \frac{1}{2}(y_4 - s_4)^2$

$$s_4 = W_{sh}.h_4$$
$$h_4 = g(W_{xh}.x_4 + W_{hh}.h_3)$$

RNN: Back Propagation Through Time (BPTT)



Gradient calculation wrt W_{hh} :

$$\frac{\partial L^4}{\partial W_{hh}} = \frac{\partial L^4}{\partial s_4} \cdot \frac{\partial s_4}{\partial h_4} \cdot \frac{\partial h_4}{\partial W_{hh}}$$

$$= -(y_4 - s_4).W_{sh}.\frac{\partial h_4}{\partial W_{hh}}$$

Now, $h_4 = g(W_{xh}.x_4 + W_{hh}.h_3)$

$h_3 = g(W_{xh}.x_3 + W_{hh}.h_2)$

$h_2 = g(W_{xh}.x_2 + W_{hh}.h_1)$

Then, $\frac{\partial h_4}{\partial W_{hh}} = g'.[h_3 + \frac{\partial h_3}{\partial W_{hh}}]$

$$= g'.[h_3 + h_2 + \frac{\partial h_2}{\partial W_{hh}}]$$

$$= g'.[h_3 + h_2 + \cdots + \frac{\partial h_0}{\partial W_{hh}}]$$

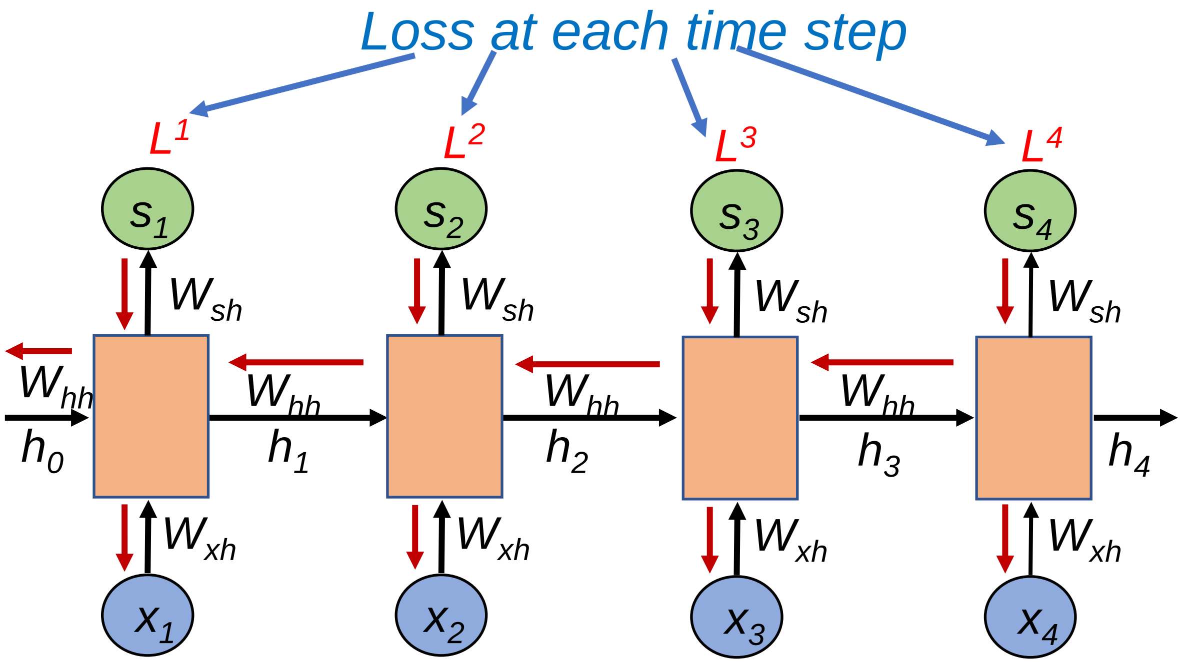
Note:

least square function $\mathcal{L} = \frac{1}{2}(y_4 - s_4)^2$

$s_4 = W_{sh}.h_4$

$h_4 = g(W_{xh}.x_4 + W_{hh}.h_3)$

RNN: Back Propagation Through Time (BPTT)



Gradient calculation wrt W_{hh} *:*

$$\frac{\partial L^4}{\partial W_{hh}} = -(y_4 - s_4).W_{sh}.g'.[h_3 + h_2 + \cdots + \frac{\partial h_0}{\partial W_{hh}}]$$

Weight updation wrt W_{hh} *:*

$$W_{hh} = W_{hh} - \sum_{i=1}^{n} \frac{\partial L^i}{\partial W_{hh}}$$

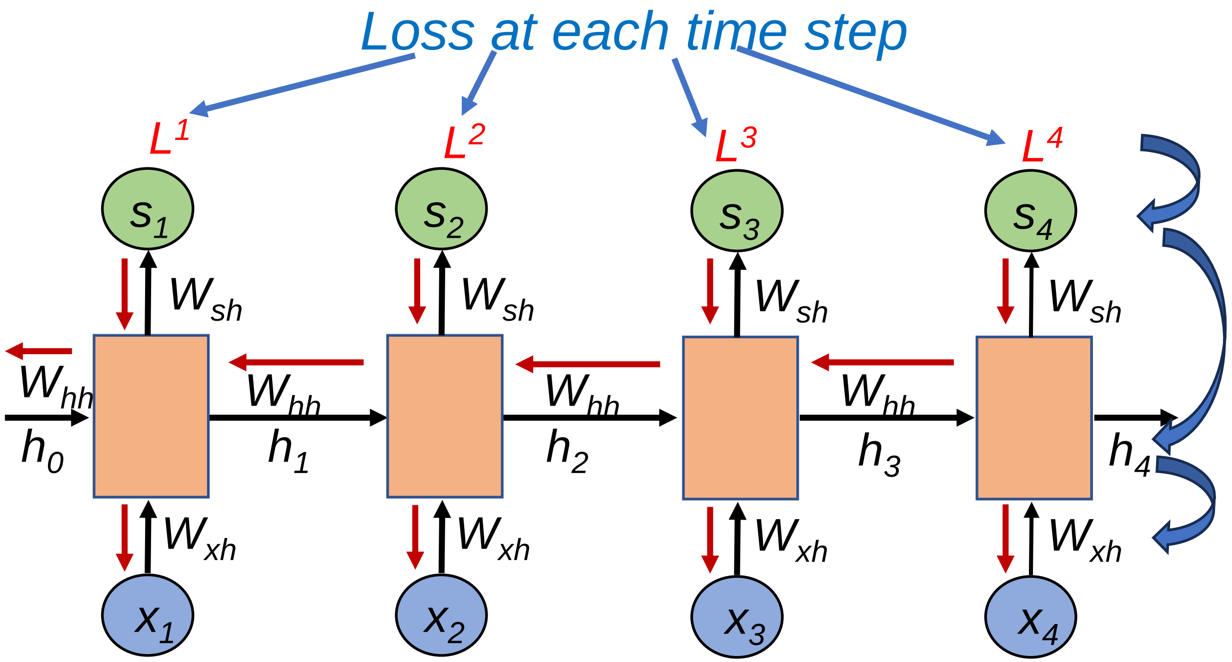
Note:

least square function $\mathcal{L} = \frac{1}{2}(y_4 - s_4)^2$

$$s_4 = W_{sh}.h_4$$

$$h_4 = g(W_{xh}.x_4 + W_{hh}.h_3)$$

RNN: Back Propagation Through Time (BPTT)



Gradient calculation wrt W_{xh} :

$$\frac{\partial L^4}{\partial W_{xh}} = \frac{\partial L^4}{\partial s_4} \cdot \frac{\partial s_4}{\partial h_4} \cdot \frac{\partial h_4}{\partial W_{xh}}$$

$$= -(y_4 - s_4).W_{sh} . \frac{\partial h_4}{\partial W_{xh}}$$

Now, $h_4 = g(W_{xh}.x_4 + W_{hh}.h_3)$

Then, $\frac{\partial h_4}{\partial W_{xh}} = g'.[x_4 + \frac{\partial W_{hh}.h_3}{\partial W_{xh}}]$

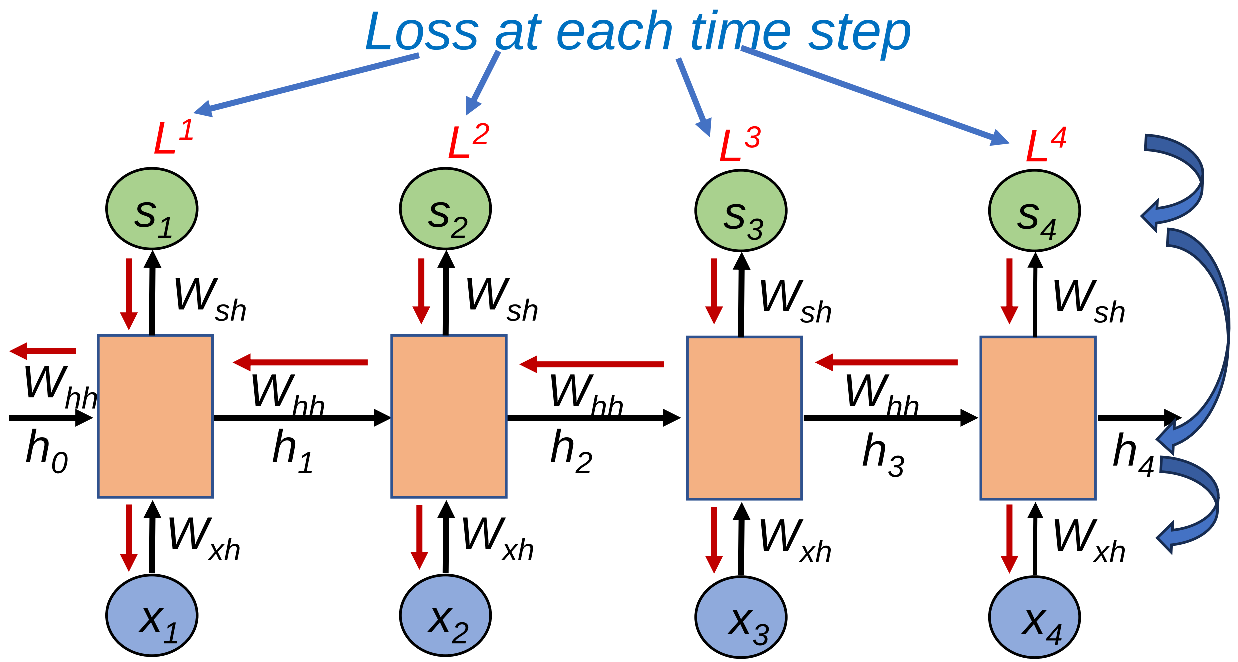
$$= g'.[x_4 + W_{hh}.g'(z_2).\frac{\partial z_2}{\partial W_{xh}}]$$

Note:

least square function $\mathcal{L} = \frac{1}{2}(y_4 - s_4)^2$

$$s_4 = W_{sh}.h_4$$
$$h_4 = g(W_{xh}.x_4 + W_{hh}.h_3)$$

RNN: Back Propagation Through Time (BPTT)



Gradient calculation wrt W_{xh} :

$$\frac{\partial L^4}{\partial W_{xh}} = \frac{\partial L^4}{\partial s_4} \cdot \frac{\partial s_4}{\partial h_4} \cdot \frac{\partial h_4}{\partial W_{xh}}$$

$$= -(y_4 - s_4).W_{sh}.\frac{\partial h_4}{\partial W_{xh}}$$

Now, $h_4 = g(W_{xh}.x_4 + W_{hh}.h_3)$

$$h_3 = g(\underbrace{W_{xh}.x_3 + W_{hh}.h_2}_{z_2})$$

Then, $\frac{\partial h_4}{\partial W_{xh}} = g'.[x_4 + \frac{\partial W_{hh}.h_3}{\partial W_{xh}}]$

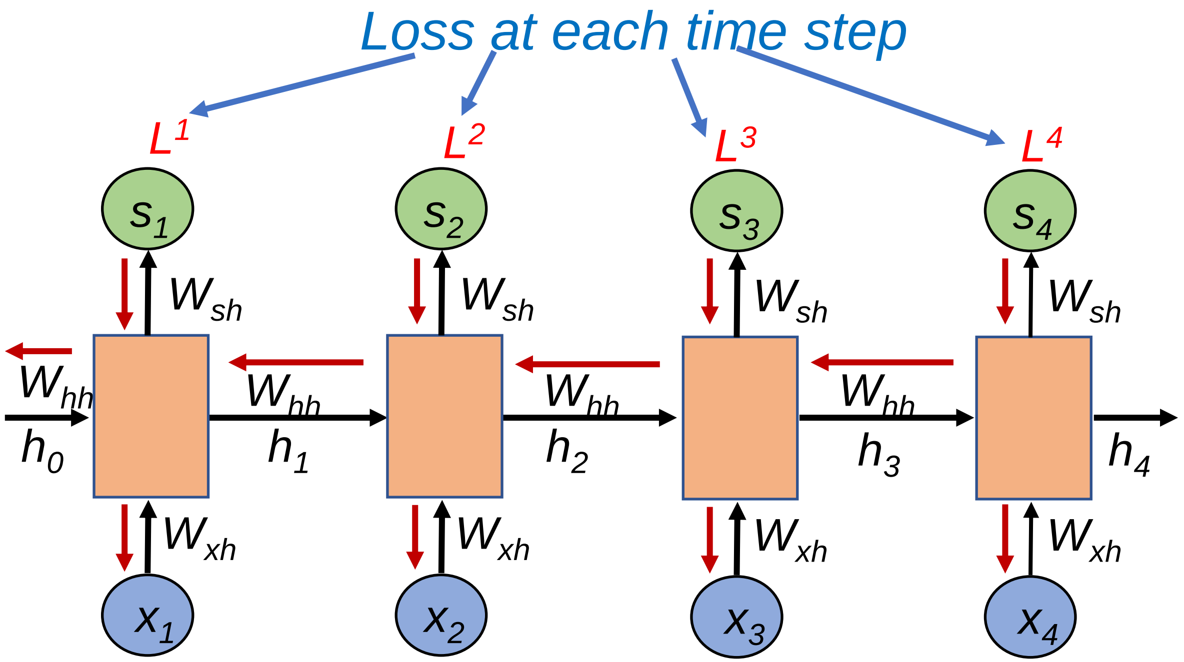
$$= g'.[x_4 + W_{hh}.g'(z_2).\frac{\partial z_2}{\partial W_{xh}}]$$

Note:

least square function $\mathcal{L} = \frac{1}{2}(y_4 - s_4)^2$

$$s_4 = W_{sh}.h_4$$
$$h_4 = g(W_{xh}.x_4 + W_{hh}.h_3)$$

RNN: Back Propagation Through Time (BPTT)



Gradient calculation wrt W_{xh} :

$$\frac{\partial L^4}{\partial W_{xh}}$$

$$= -(y_4 - s_4)W_{sh}.\, g'.\,[x_4 + W_{hh}.\, g'(z_2).\frac{\partial z_2}{\partial W_{xh}} \dots.]$$

Weight updation wrt W_{xh} :

$$W_{xh} = W_{xh} - \sum_{i=1}^{n} \frac{\partial L^i}{\partial W_{xh}}$$

Note:

least square function $\mathcal{L} = \frac{1}{2}(y_4 - s_4)^2$

$$s_4 = W_{sh}.\, h_4$$

$$h_4 = g(W_{xh}.\, x_4 + W_{hh}.\, h_3)$$

Limitations of RNN

- Gradient calculation involves many factors of weights and contribution of activation function
- This may lead to:

Weight Updation

$$W \leftarrow W - \eta \frac{\partial L}{\partial W}$$

Exploding Gradient

$$\frac{\partial L}{\partial W} \gg 1 \rightarrow resulting\ in\ NAN\ values$$

Vanishing Gradient

$$\frac{\partial L}{\partial W} << 1 \rightarrow resulting\ in\ 0\ values$$

If this weight Matrix W is very big this can result in the exploding gradient problem and it may not possible to train the network stably.

Conversely we can have the instance where the weight matrices are very small -> very small value at the end as a result of these repeated weight Matrix computations ->Vanishing gradient where now your gradient has just dropped down close to zero and again network cannot be trained stably.

Limitations of RNN

- Gradient calculation involves many factors of weights and contribution of activation function.
- This may lead to:

Exploding Gradient

- make learning unstable

Vanishing Gradient

- Short term dependencies

"the stars shine in the ?" → sky ***(RNN works good here)***

- Long term dependencies

"I grew up in Spain…..............................
I speak fluent Spanish". ***(Difficult for RNN to remember as gap increases)***

Possible Solutions

Exploding Gradient

- ❑ Gradient clipping

```
# inside the optimizer we are doing clipping
optimizer=tf.keras.optimizers.SGD(clipvalue=0.5)
```

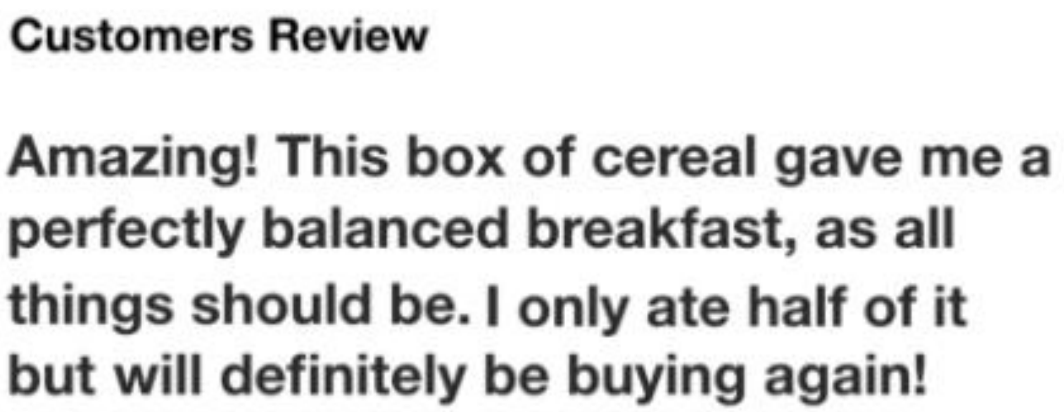
Vanishing Gradient

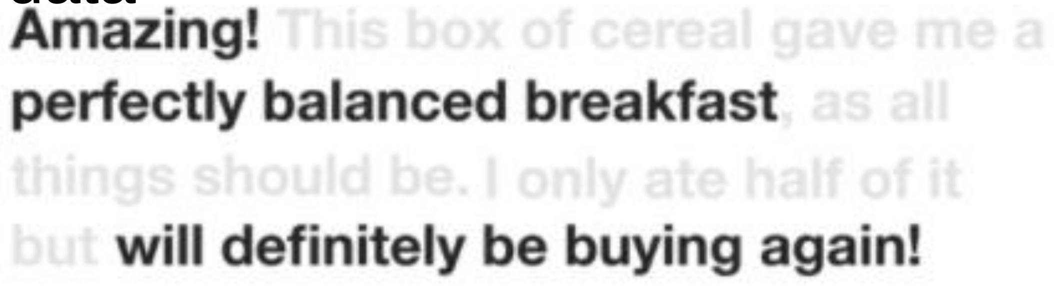
- ❑ Activation function (ReLu)
- ❑ Weight initialization (identity matrix)
- ❑ Gated cells (LSTM,GRU,etc)

Long Short Term Memory networks (LSTM)

- Introduced by Hochreiter & Schmidhuber (1997)
- LSTMs are a type of RNN capable of learning long-term dependencies

LSTMs can learn to keep only relevant information to make predictions, and forget non relevant data

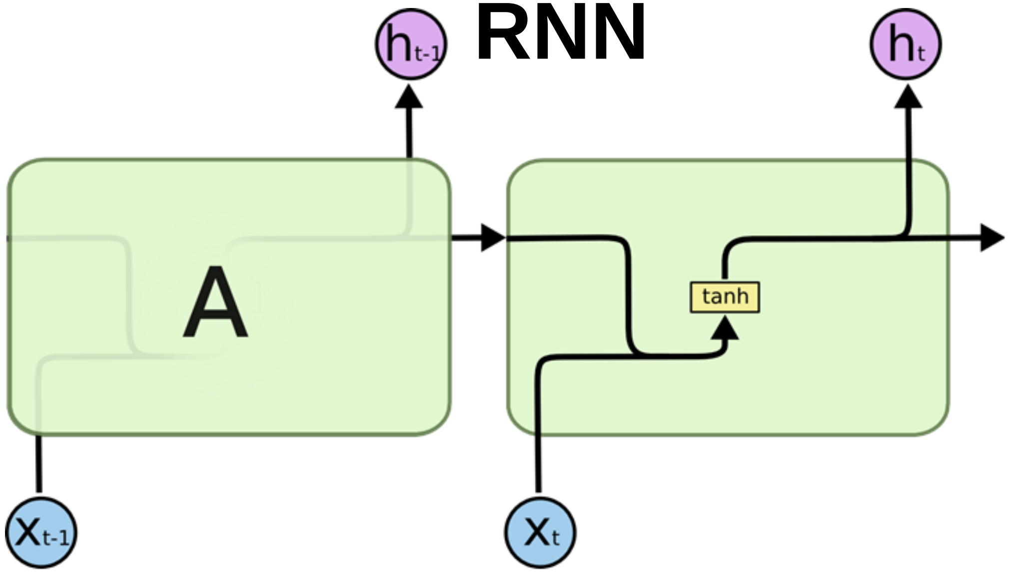


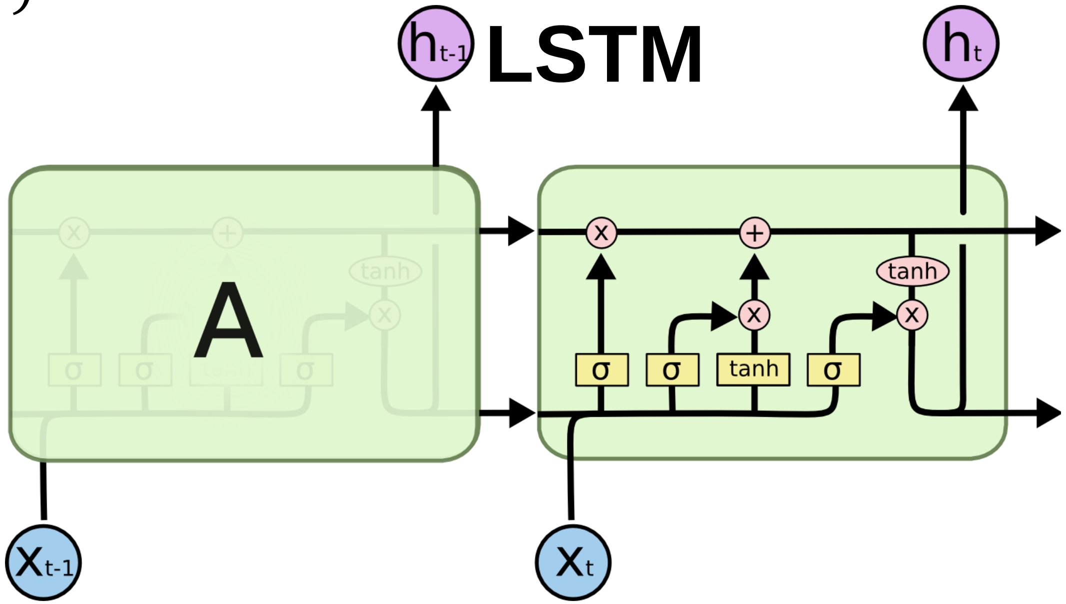


Long Short Term Memory networks (LSTM)

- Introduced by Hochreiter & Schmidhuber (1997)
- LSTMs learn long-term dependencies using Cell State and Gates

$$h_t = f(h_{t-1} \cdot W, x_t \cdot U)$$



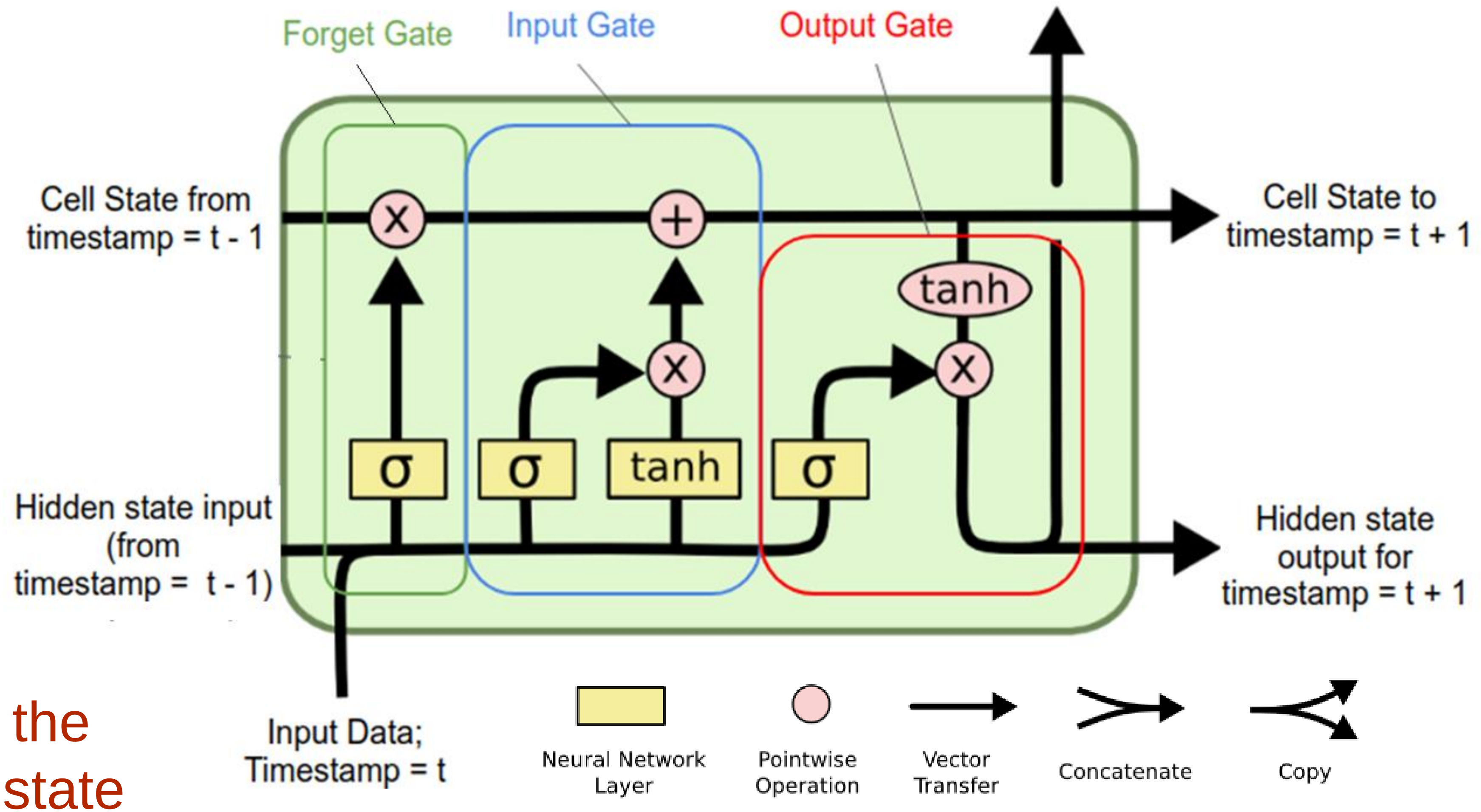


Reference: https://builtin.com/artificial-intelligence/transformer-neural-network

LSTM

- Cell State: encode information from *earlier* time steps, reducing the effect of memory loss
- Hidden State: working memory is usually called the hidden state
- Input at time step t

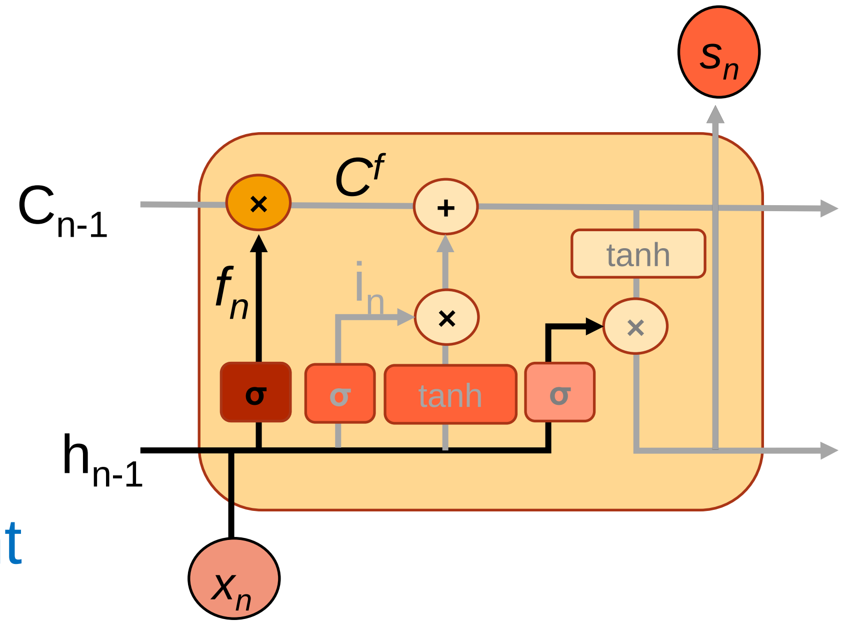
Regular RNNs have just the hidden state and no cell state



Long Short-Term Memory (LSTM)

How do LSTMs work?

a) **Forget**
b) Input
c) Update
d) Output



Forget gate gets rid of irrelevant information

Decides which information to forget from previous cell state (C_{n-1})

$$f_n = \sigma(W_{hf} h_{n-1} + W_{xf} x_n + b_f)$$

$$C^f = f_n * C_{n-1}$$

Output of Sigmoid Activation (σ):
0 => Forget,
1 => Keep

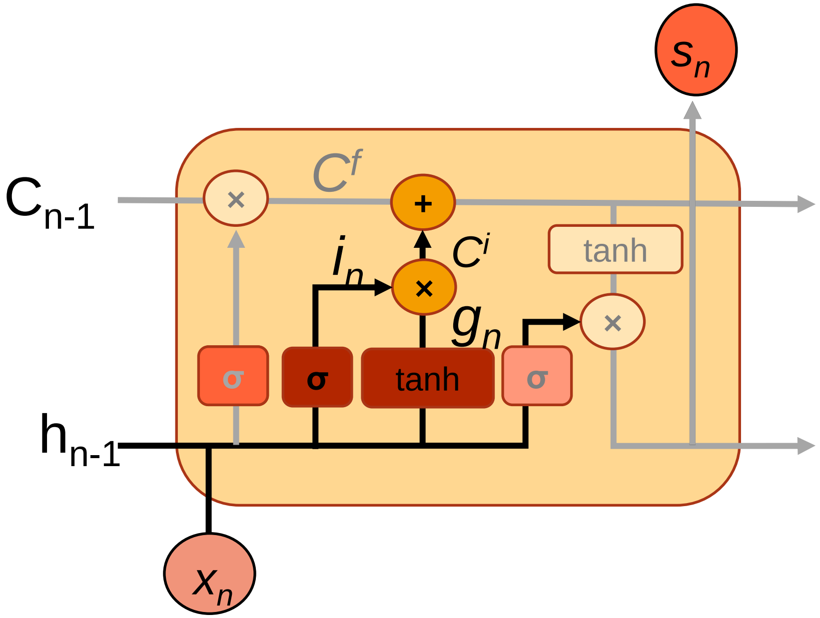
Long Short-Term Memory (LSTM)

How do LSTMs work?

a) Forget
b) Input
c) Update
d) Output

Input gate stores relevant information from current input

Decides which new information to be saved to the current cell state in



$$i_n = \sigma(W_{hi}h_{n-1} + W_{xi}x_n + b_i)$$
$$g_n = tanh(W_{hg}h_{n-1} + W_{xg}x_n + b_g)$$
$$C^i = (i_n * g_n)$$

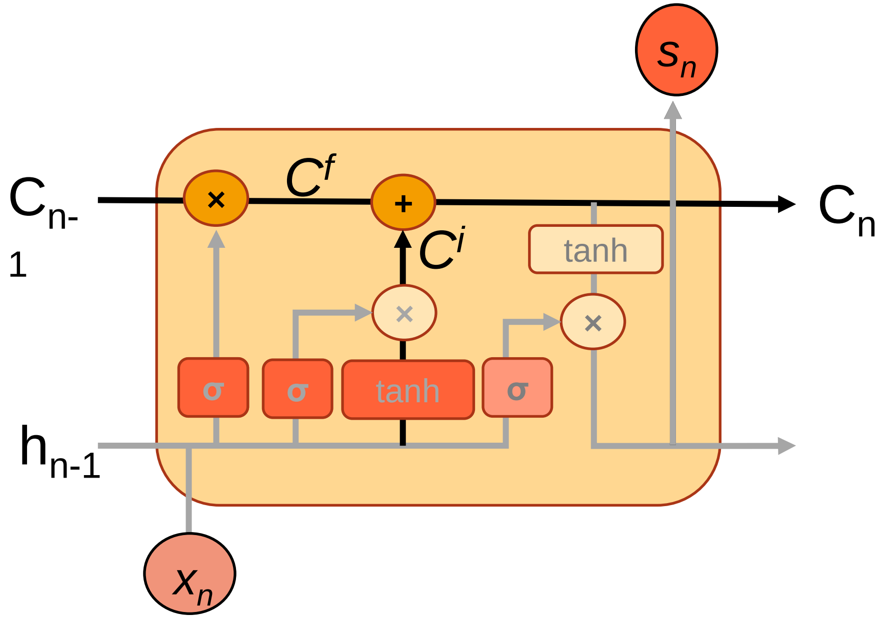
Output of Sigmoid Activation (σ):
0 => Forget,
1 => Keep

Long Short-Term Memory (LSTM)

How do LSTMs work?

a) Forget
b) Input
c) Update
d) Output

Selectively update cell state value



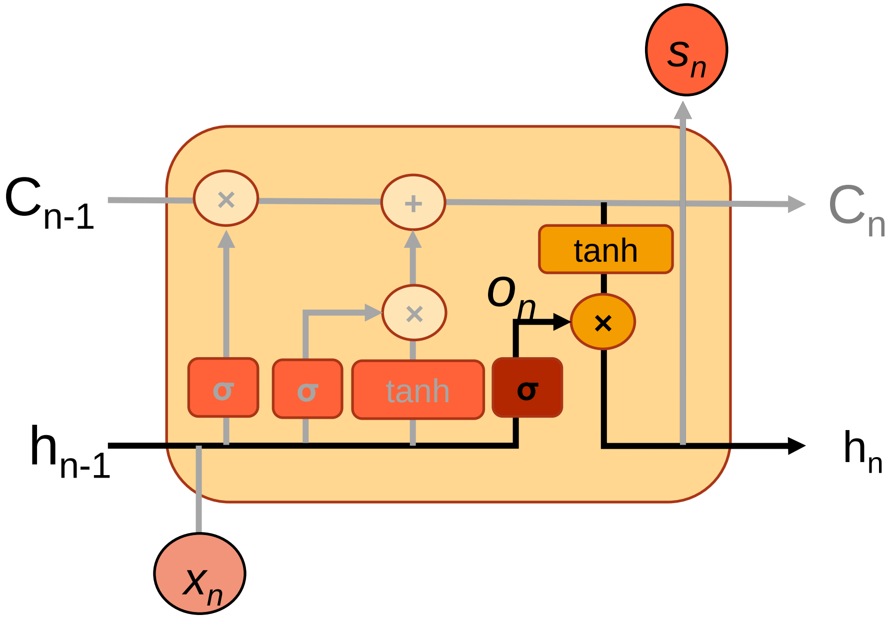
$$C_n = C^i + C^f$$

Long Short-Term Memory (LSTM)

Output of Sigmoid Activation (σ):
0 => Forget,
1 => Keep

How do LSTMs work?
a) Forget
b) Input
c) Update
d) Output

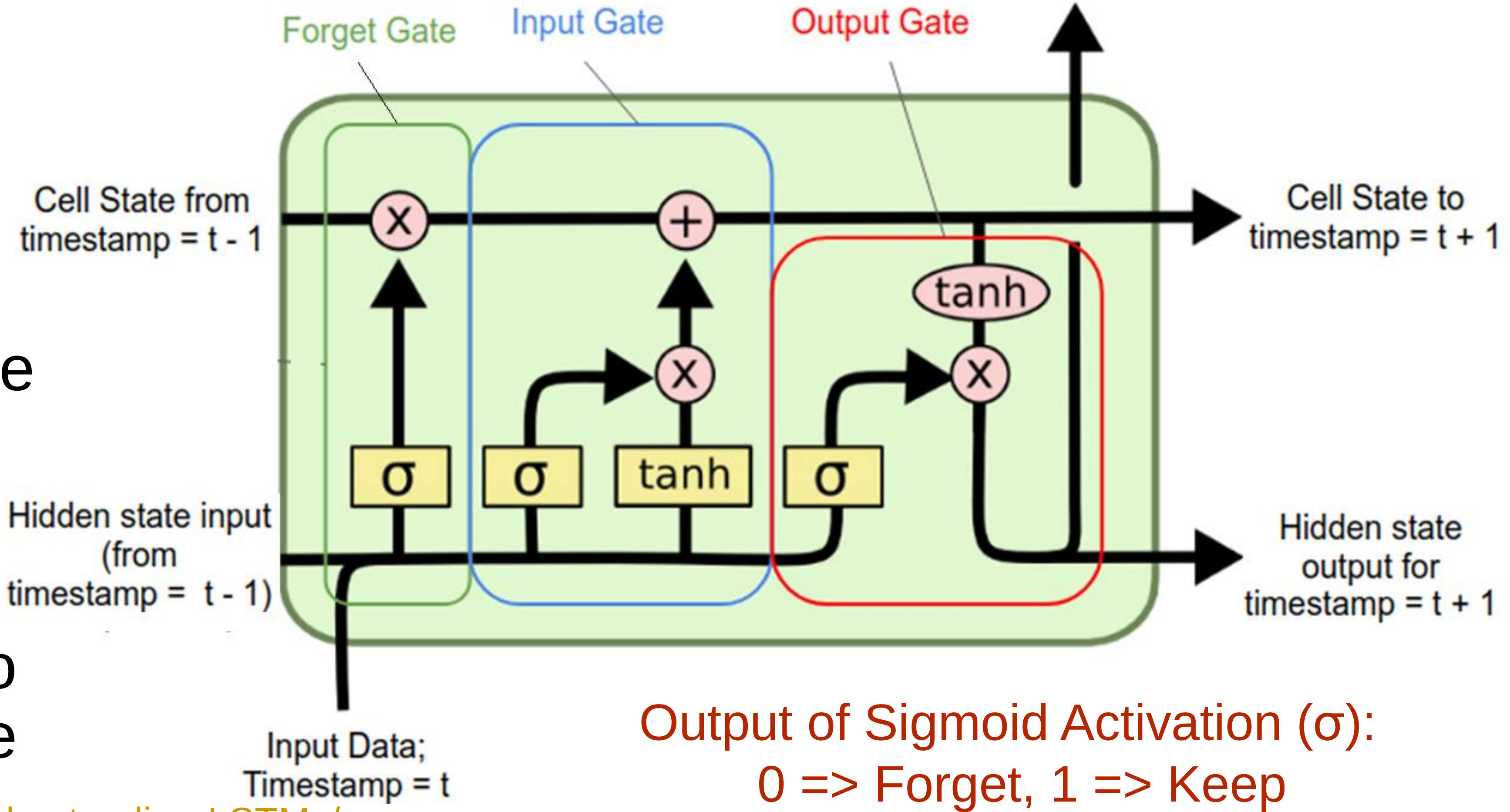
Output gate returns a filtered version of the cell state
Decides which information to give to the next hidden state



$$o_n = \sigma(W_{ho}h_{n-1} + W_{xo}x_n + b_o)$$
$$h_n = o_n * \tanh(C_n)$$

LSTM has three gates

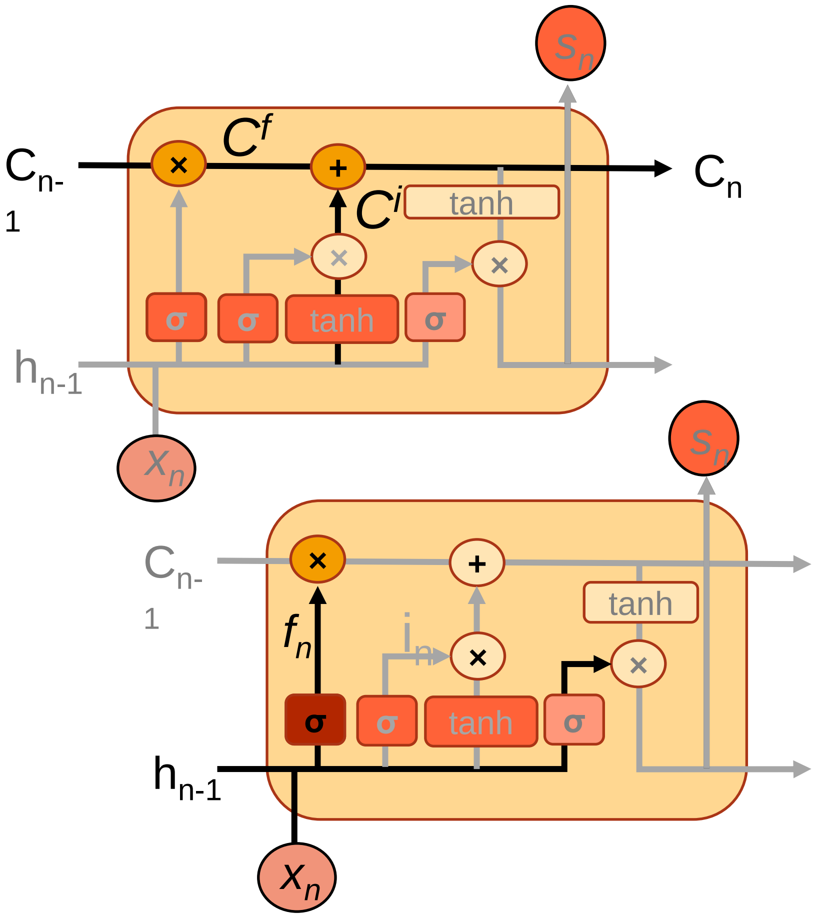
- Forget Gate: Which information to forget from previous cell state (Ct-1)
- Input Gate: Which new information to be saved to the current cell state Ct
- Output Gate: which information to give to the next hidden state



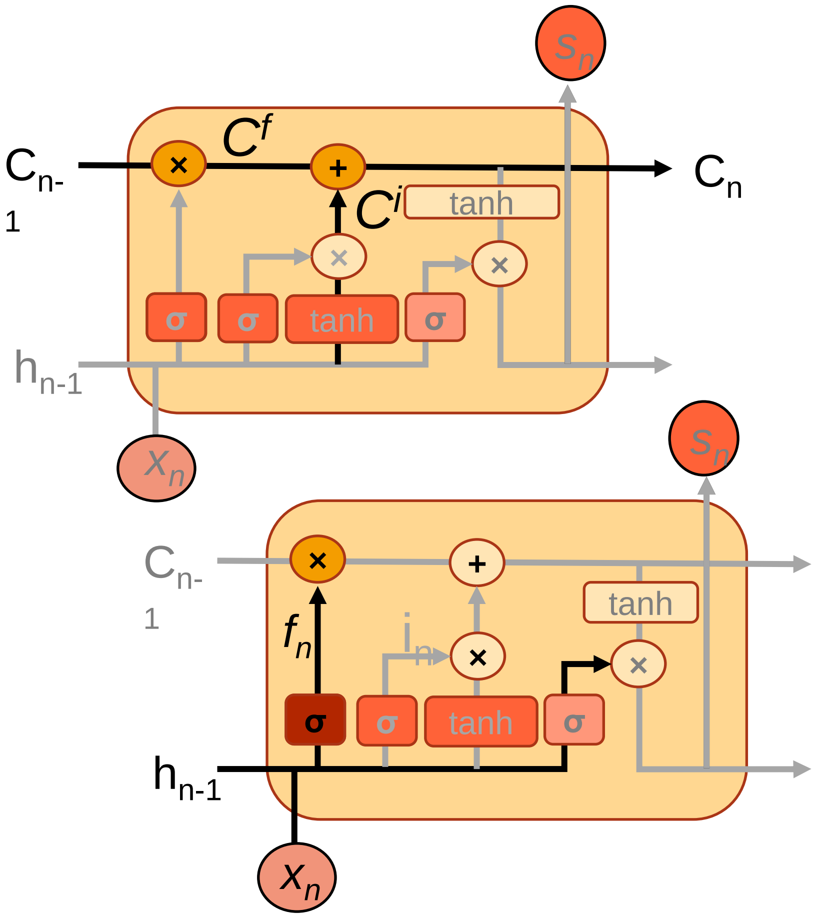
Output of Sigmoid Activation (σ):
0 => Forget, 1 => Keep

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM: How it solves the vanishing gradient problem?



LSTM: How it solves the vanishing gradient problem?



Ans: Using Forget gate

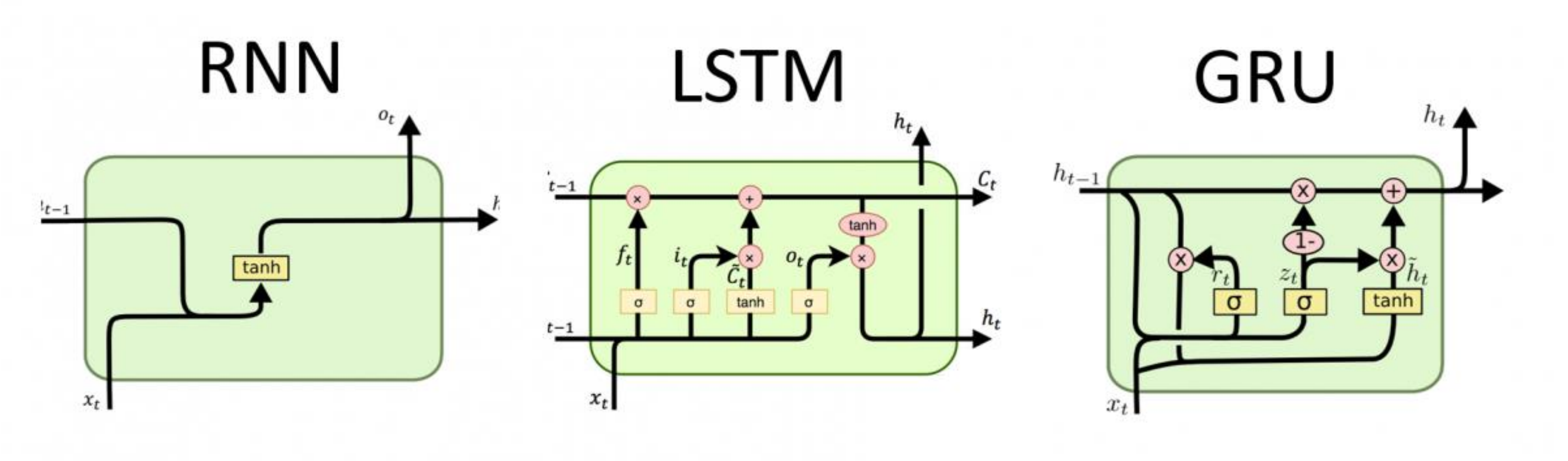
Gradient at C_n passed on to C_{n-1} is unaffected by any other operations, but the forget gate.

$$C^f = f_n * C_{n-1}$$

$$C_n = C^i + C^f$$

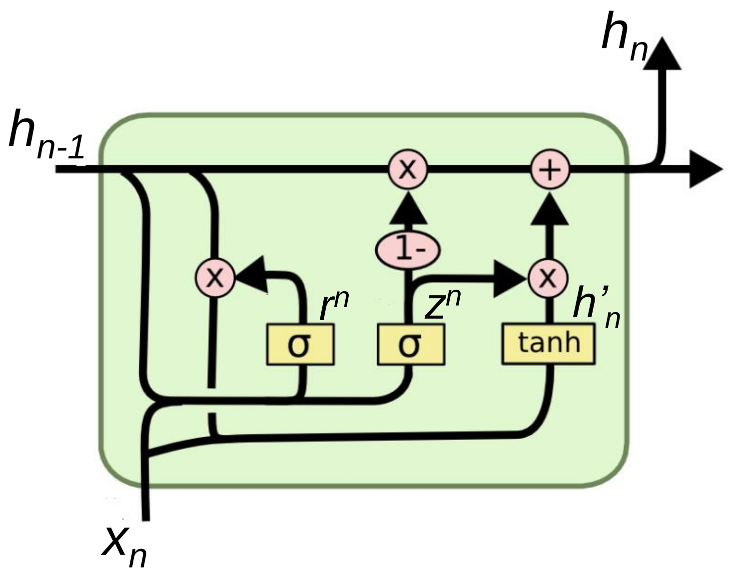
$$h_n = o_n * \tanh(C_n)$$

RNN, LSTM, GRU



Refer: http://dprogrammer.org/rnn-lstm-gru

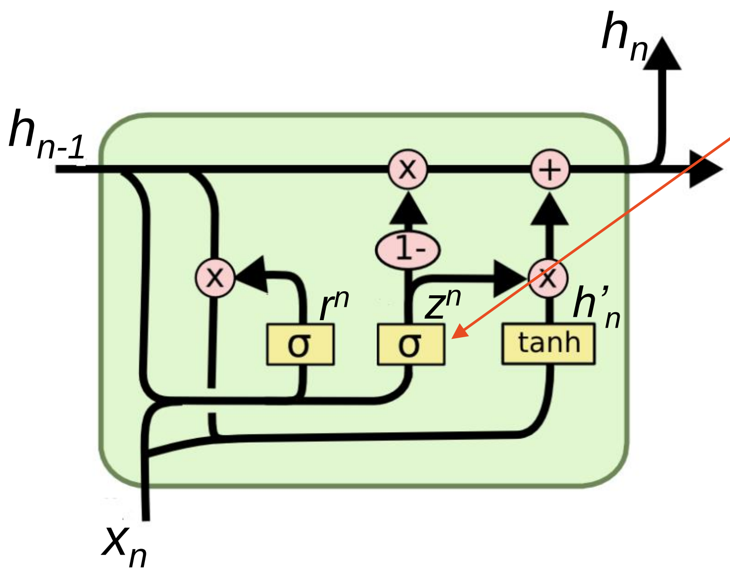
Gated Recurrent Unit (GRU)



- Proposed in 2014 as simpler alternative to LSTM
- Combines **forget** and **input** gates into a single **update** gate
- Merges cell state $\boldsymbol{C_n}$ and hidden state $\boldsymbol{h_n}$

Refer: http://dprogrammer.org/rnn-lstm-gru

Gated Recurrent Unit (GRU)



Update gate: controls what parts of hidden state are updated vs preserved

$$\boxed{z^n = \sigma(W_z * [h_{n-1}, x_n])}$$

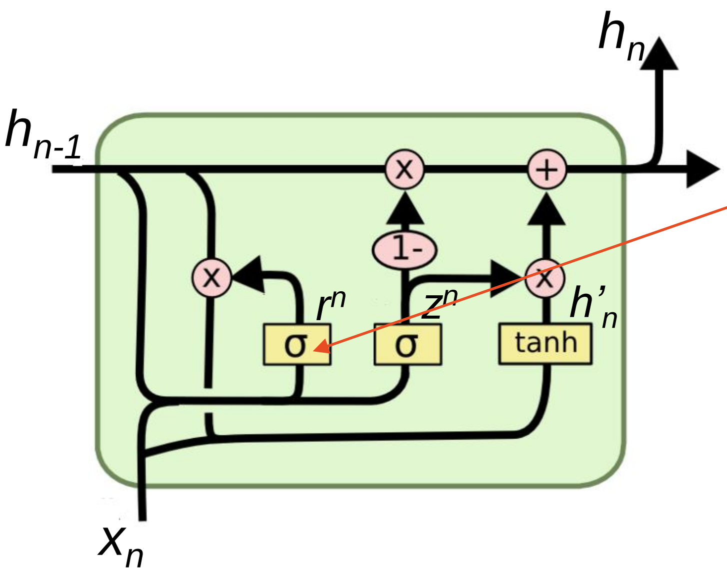
Reset gate: controls what parts of previous hidden state are used to compute new content

$$\boxed{r^n = \sigma(W_r * [h_{n-1}, x_n])}$$

New Hidden state content: selects useful parts of previous hidden state. Use this and current input to compute new hidden content $\boxed{h'_n = \tanh(W * [r^n * h_{n-1}, x_n])}$

Refer: http://dprogrammer.org/rnn-lstm-gru

Gated Recurrent Unit (GRU)



Update gate: controls what parts of hidden state are updated vs preserved

$$\boxed{z^n = \sigma(W_z * [h_{n-1}, x_n])}$$

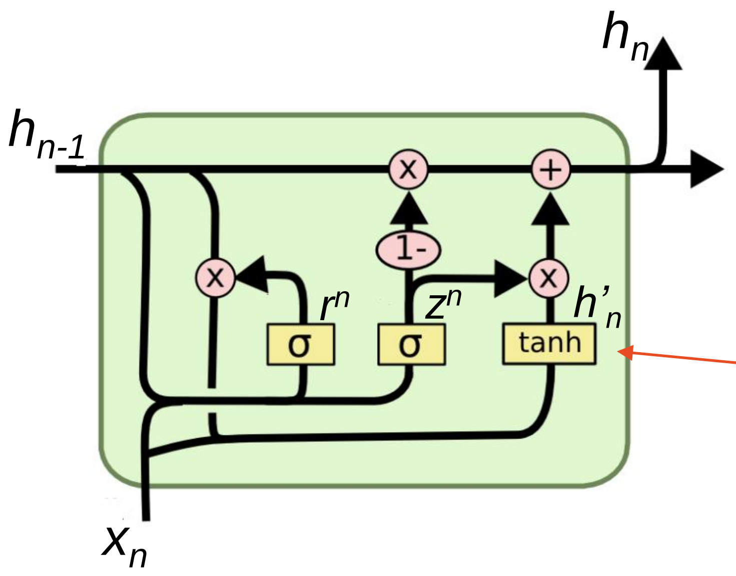
Reset gate: controls what parts of previous hidden state are used to compute new content

$$\boxed{r^n = \sigma(W_r * [h_{n-1}, x_n])}$$

New Hidden state content: selects useful parts of previous hidden state. Use this and current input to compute new hidden content $\boxed{h'_n = \tanh(W * [r^n * h_{n-1}, x_n])}$

Refer: http://dprogrammer.org/rnn-lstm-gru

Gated Recurrent Unit (GRU)



Update gate: controls what parts of hidden state are updated vs preserved

$$\boxed{z^n = \sigma(W_z * [h_{n-1}, x_n])}$$

Reset gate: controls what parts of previous hidden state are used to compute new content

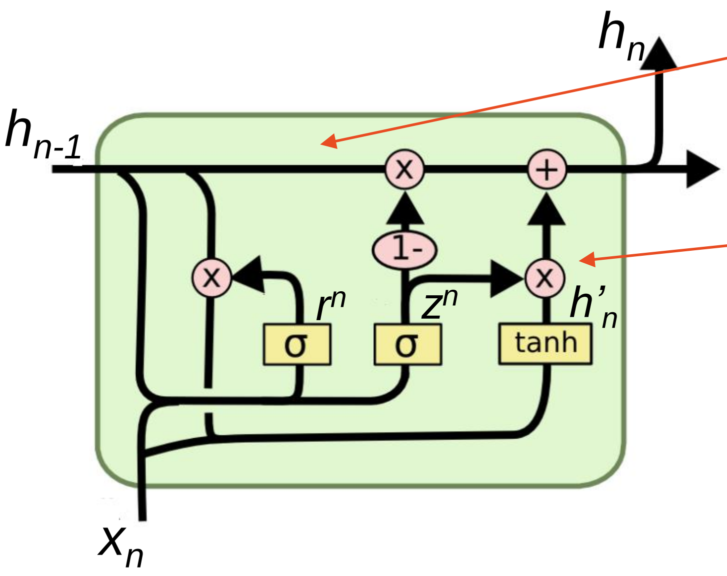
$$\boxed{r^n = \sigma(W_r * [h_{n-1}, x_n])}$$

New Hidden state content: selects useful parts of previous hidden state. Use this and current input to compute new hidden content

$$\boxed{h'_n = \tanh(W * [r^n * h_{n-1}, x_n])}$$

Refer: http://dprogrammer.org/rnn-lstm-gru

Gated Recurrent Unit (GRU)



$$h_n = (1 - z^n) * h_{n-1} + z^n * h'_n$$

Hidden state: simultaneously controls what is kept from previous hidden state, and what is updated to new hidden state content

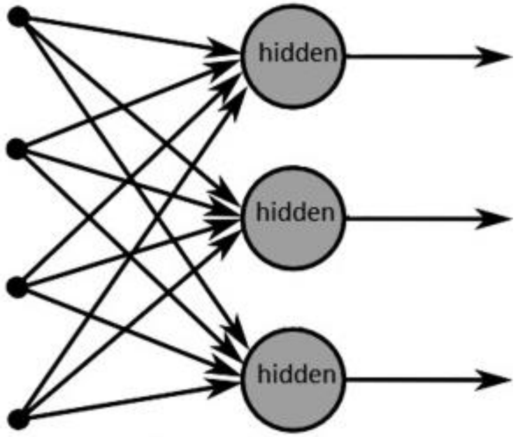
Refer: http://dprogrammer.org/rnn-lstm-gru

LSTM vs GRU

- Input and forget gates of LSTMs are coupled by an update gate in GRUs; reset gate in GRUs is applied directly to previous hidden state
- GRU has two gates, an LSTM has three gates. Lesser parameters to learn!
- In GRUs:
  - No internal memory (c_n) different from exposed hidden state
  - No output gate as in LSTMs
- LSTM is a good default choice (especially if data has long-range dependencies, or if training data is large)
- Switch to GRUs for speech and fewer parameters

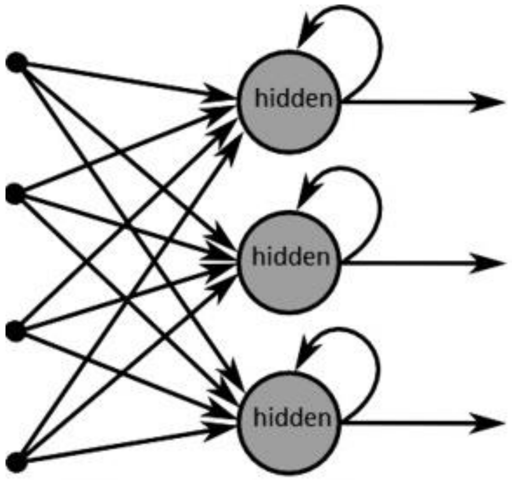
Feed-Forward Neural Network

- Network has no Memory
- Useful when one input is not related to another, eg. Images of cars, people, handwritten digits
-
- CNN, Autoencoders



Feed-back Neural Network

- Network has memory or feedback
- Useful when there is a sequence that needs to be recognized, eg. Video, speech, stock market analysis, etc.
- Recurrent Neural Network (RNN), LSTMs



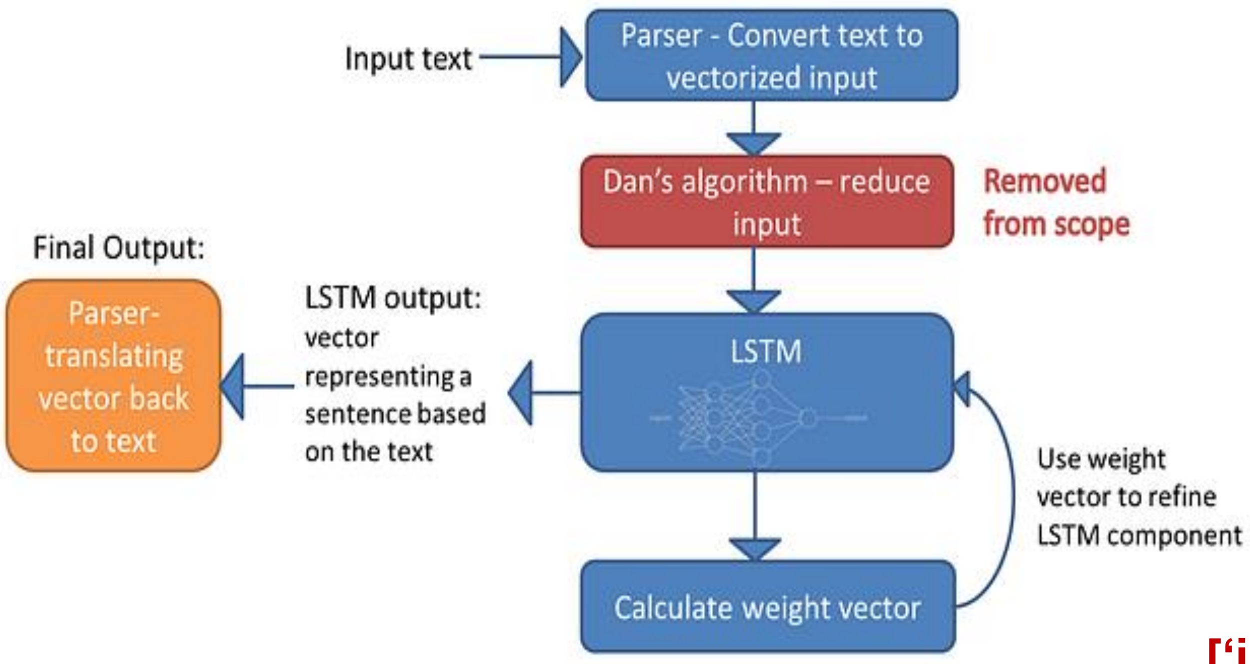
Recurrent Neural Network (RNN) vs. Feedforward neural network

- RNN is a generalization of feedforward neural network that has an internal memory
- RNN is recurrent in nature as it performs the same function for every input of data while the output of the current input depends on the past one computation.
- Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs.
- This makes them applicable to tasks such as unsegmented, connected handwriting recognition or speech recognition.
- In other neural networks, all the inputs are independent of each other. But in RNN, all the inputs are related to each other.

Applications of RNNs in Generative AI

- Text generation with RNNs
- Music generation using RNNs
- Speech Recognition using RNN

Text generation with RNNs



- Text converted to vectors
- Tokenization: separate a piece of text into smaller units called tokens

I love machine learning. I hate exams. -> [‘i’, ‘love’, ‘machine’, ‘learning’, ‘i’,...]

- Ignore unimportant words (eg. ‘so’, ‘to’, ‘from’)
- Vectorization: represent text using vectors (determine number of unique words)

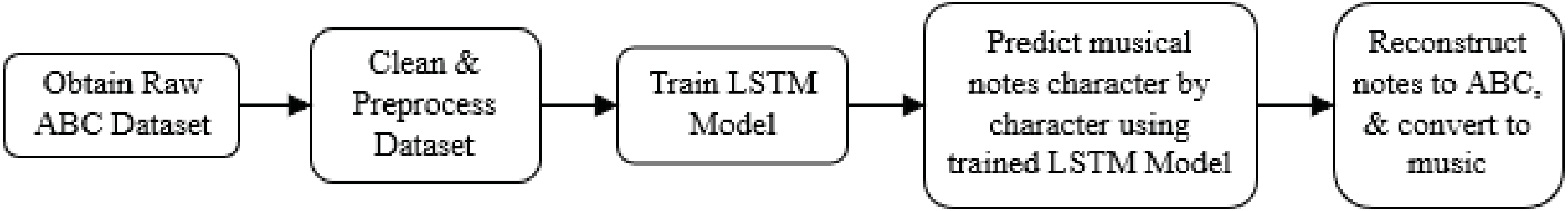
[‘i’, ‘love’, ‘machine’, ‘learning’, ‘i’,...] -> [10, 23,45,67,10]

- Train LSTM to predict next word, given a few previous words
- Eg. Autocompletion, lyrics generation

Ref: https://medium.com/@luigi.fiori.lf0303/text-generator-using-neural-network-6b11ec32b403

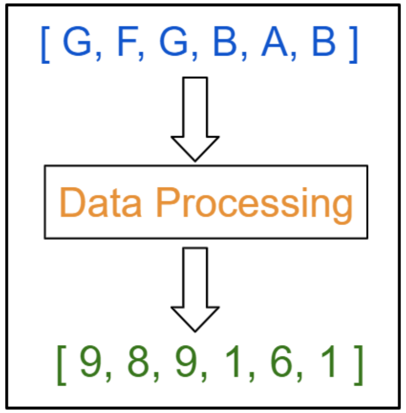
https://www.youtube.com/watch?v=un_SNEBH-0E

Music generation using RNNs



- Music representation: abc notation (Each note is written as a separate letter), MIDI format
- Eg of ABC notation `dff cee|def gfe|dff cee|dfe dBA|dff cee|`
- Vectorization: represent notes using vectors

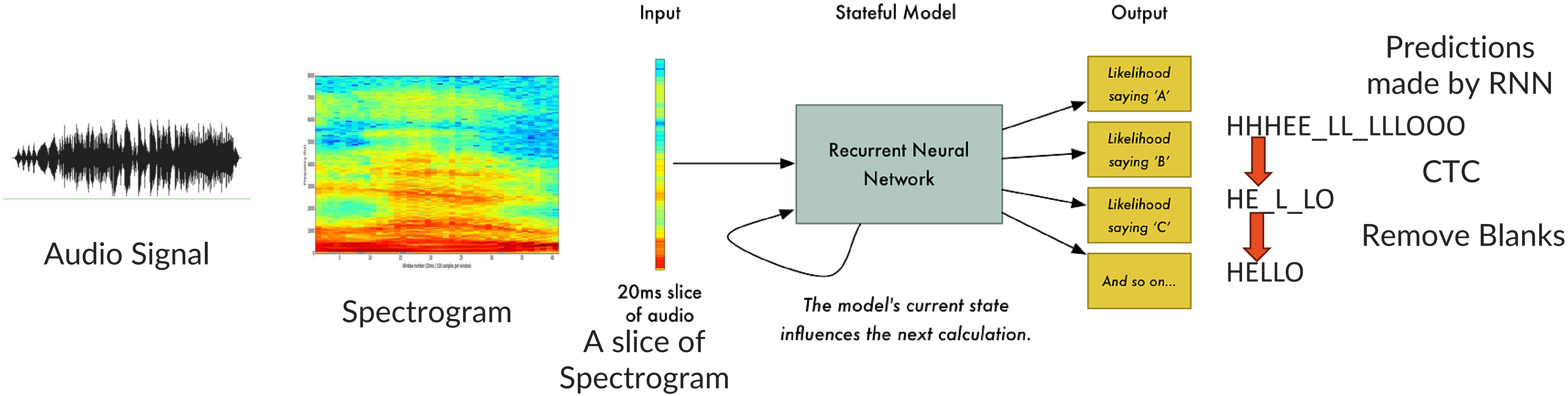
  (determine number of unique notes)
- Train LSTM to predict next note, given a few previous notes



Ref:
https://medium.com/analytics-vidhya/neural-networks-rnns-and-music-generation-a1838dfa6472

https://ieeexplore.ieee.org/document/8628712

Speech Recognition using RNN



- Speech audio signal is represented in form of spectrogram
- The likelihood of spoken letter is determined for short-time segments using RNN (RNN maps acoustic sequence to phonetic sequences)
- Connectionist Temporal Classification (CTC) decides whether to keep, emit any label, or put no label, at every timestep

Ref: https://medium.com/@ageitgey/machine-learning-is-fun-part-6-how-to-do-speech-recognition-with-deep-learning-28293c162f7a

Topics to cover

- Architecture of Recurrent Neural Networks, with example of applications
- Vanishing/Exploding gradient problems of RNN, and their solutions
- Working of LSTM. How LSTM solves the long term dependency issue of RNN
- LSTM vs. GRU
- Feedforward NN vs. RNN