Recurrent neural networks

- Dates back to (Rumelhart *et al.*, 1986)
- A family of neural networks for handling sequential data, which involves variable length inputs or outputs

- Especially, for natural language processing (NLP)

Sequential data

- Each data point: A sequence of vectors $x^{(t)}$, for $1 \leq t \leq \tau$
- Batch data: many sequences with different lengths τ
- Label: can be a scalar, a vector, or even a sequence

- Example
  - Sentiment analysis
  - Machine translation

Example: machine translation

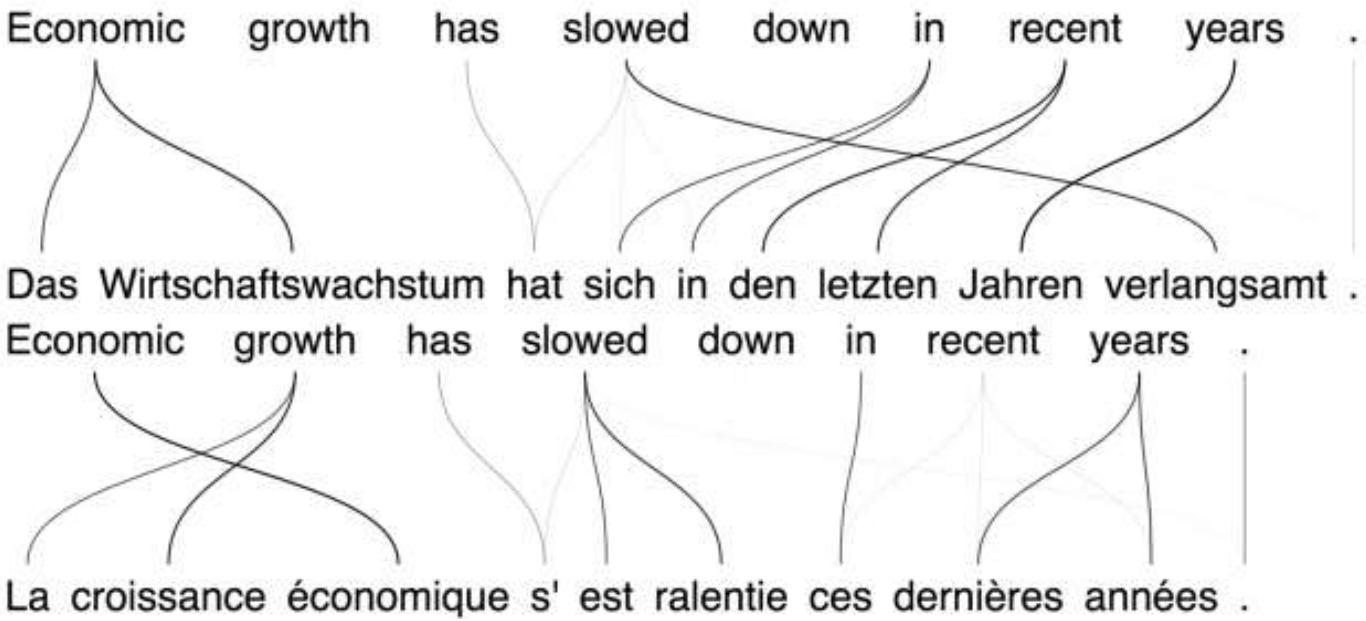


Figure from: devblogs.nvidia.com

More complicated sequential data

- Data point: two dimensional sequences like images
- Label: different type of sequences like text sentences

- Example: image captioning

Image captioning

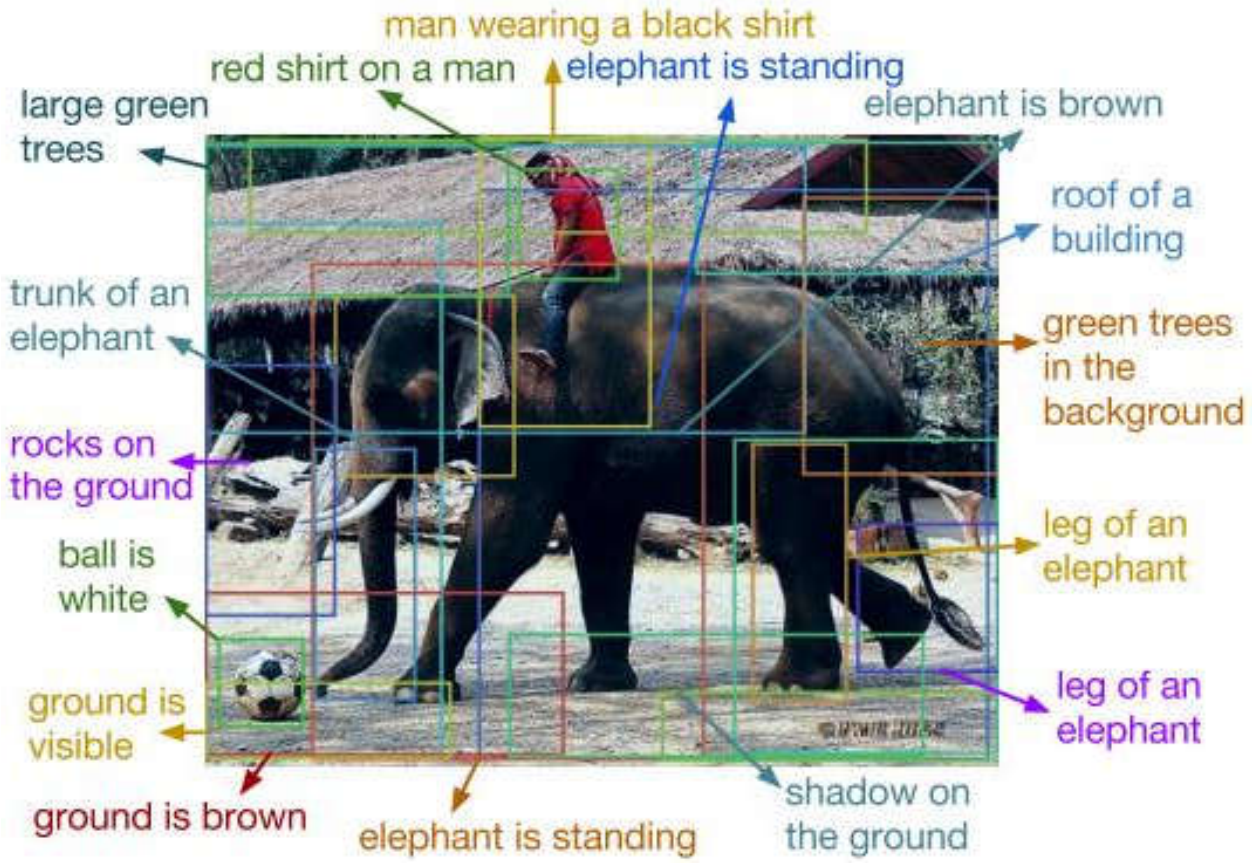
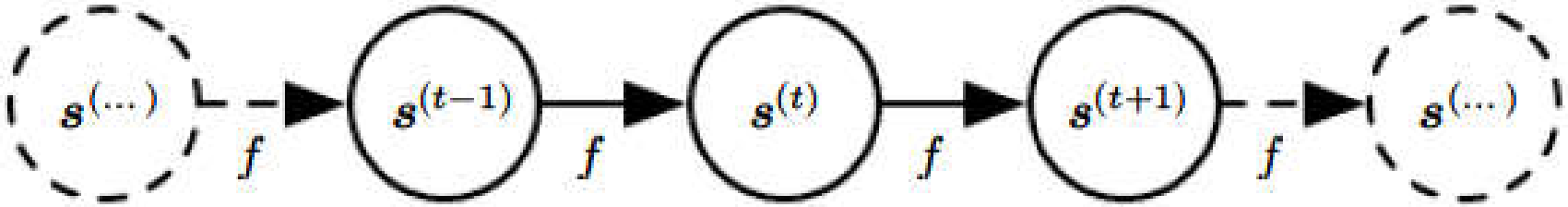


Figure from the paper “DenseCap: Fully Convolutional Localization Networks for Dense Captioning”, by Justin Johnson, Andrej Karpathy, Li Fei-Fei

Computational graphs

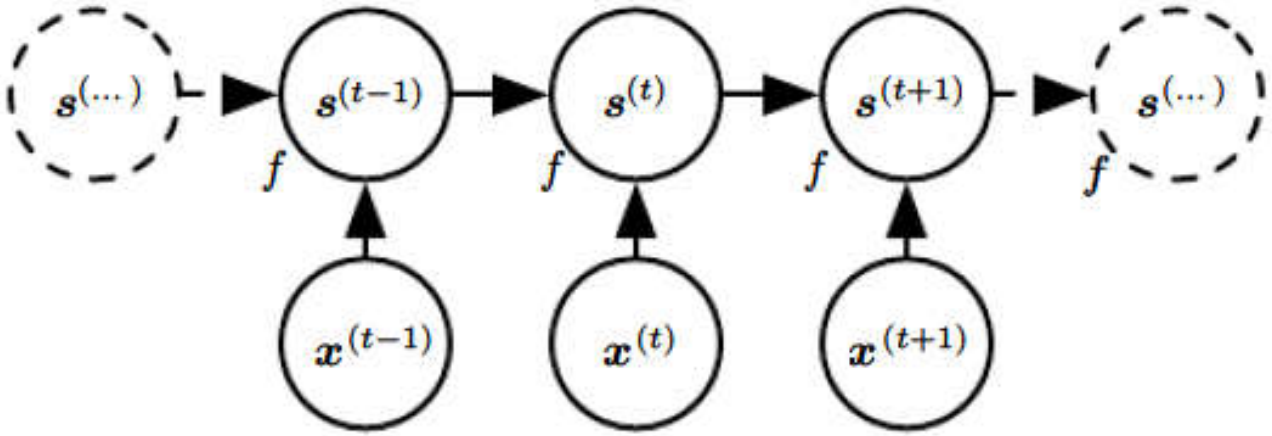
A typical dynamic system



$$s^{(t+1)} = f(s^{(t)}; \theta)$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

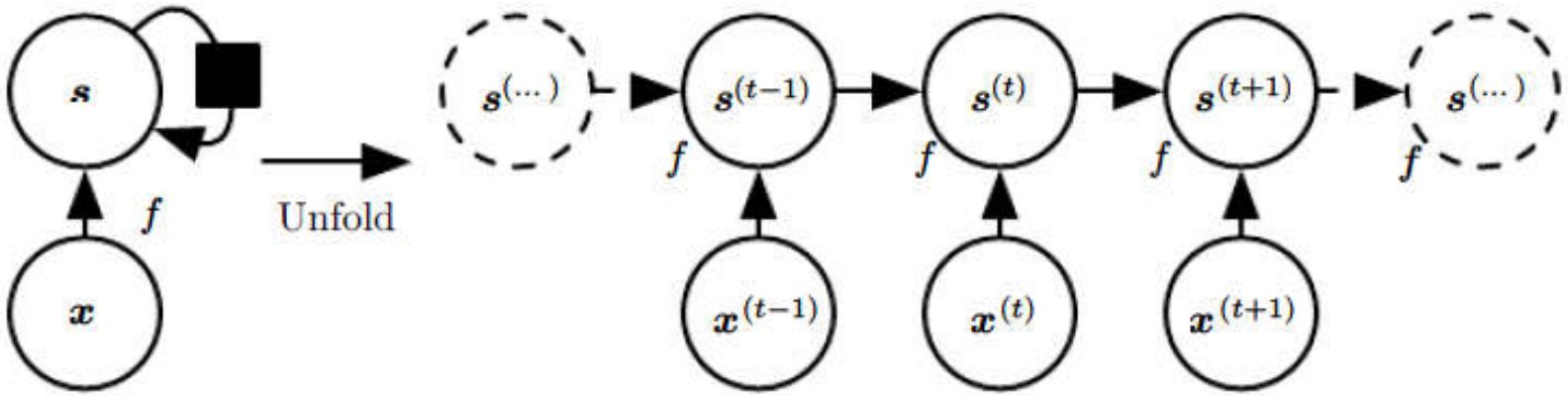
A system driven by external data



$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Compact view

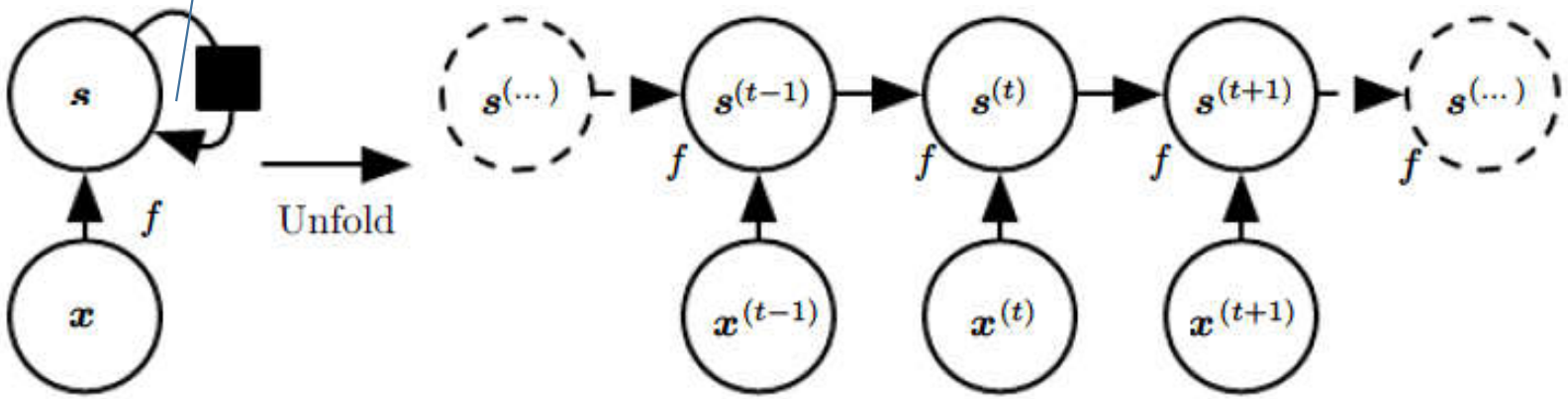


$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Compact view

square: one step time delay



$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Key: the same f and θ for all time steps

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Recurrent neural networks (RNN)

Recurrent neural networks

- Use the same computational function and parameters across different time steps of the sequence
- Each time step: takes the input entry and the previous hidden state to compute the output entry
- Loss: typically computed every time step

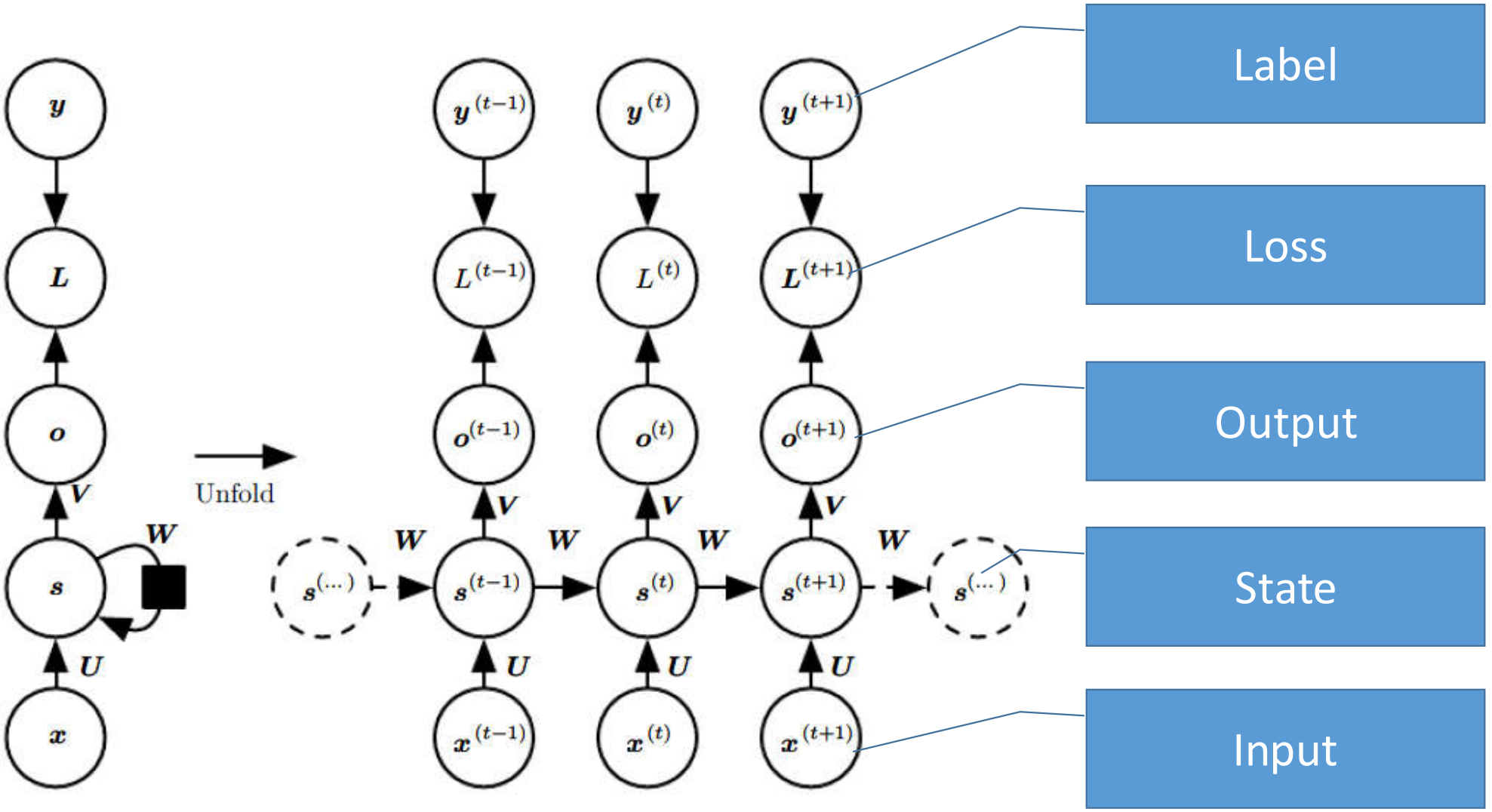
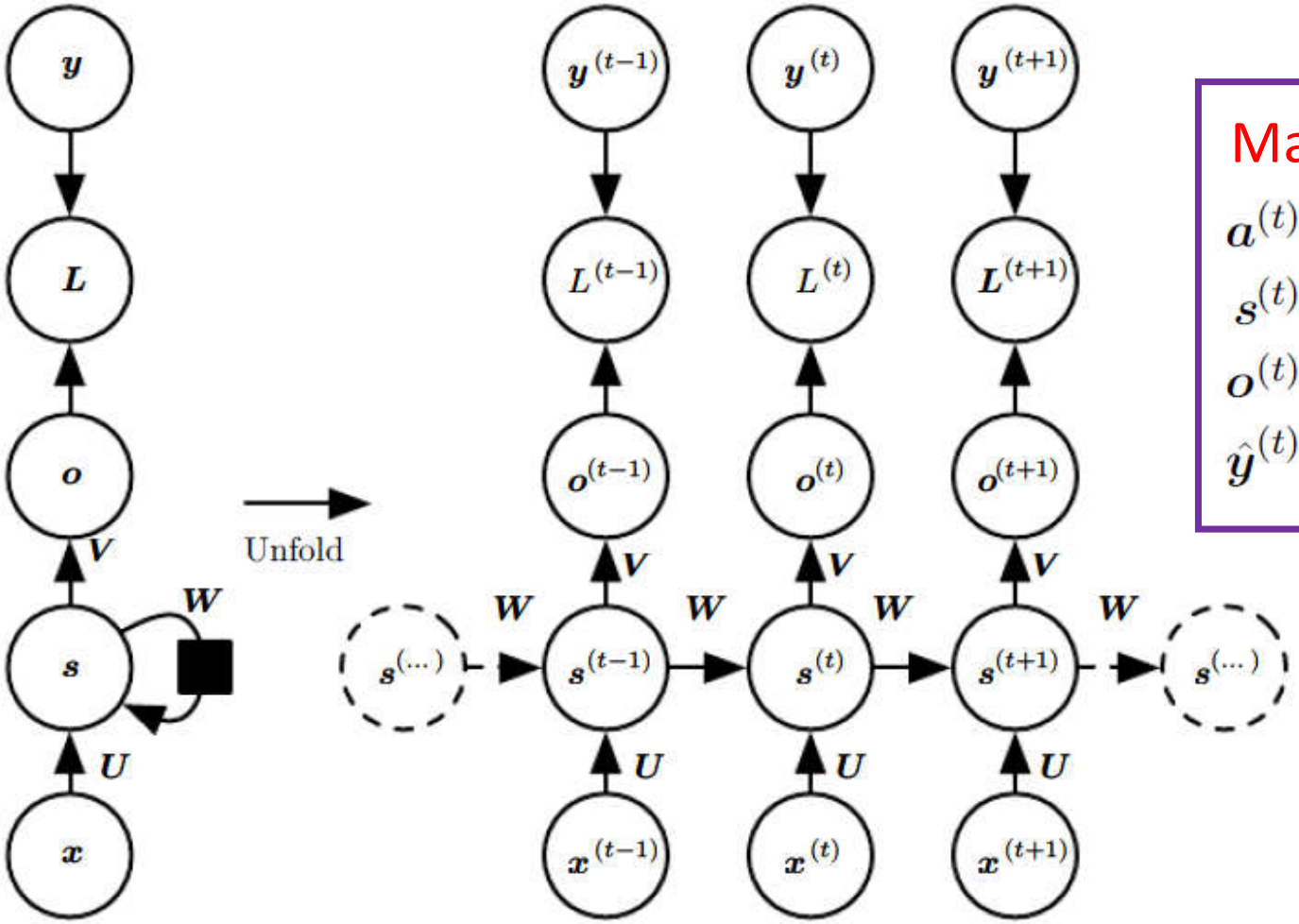


Figure from *Deep Learning*, by Goodfellow, Bengio and Courville



Math formula:

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{s}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{s}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{s}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Advantage

- Hidden state: a lossy summary of the past
- Shared functions and parameters: greatly reduce the capacity and good for generalization in learning
- Explicitly use the prior knowledge that the sequential data can be processed by in the same way at different time step (e.g., NLP)

Advantage

- Hidden state: a lossy summary of the past
- Shared functions and parameters: greatly reduce the capacity and good for generalization in learning
- Explicitly use the prior knowledge that the sequential data can be processed by in the same way at different time step (e.g., NLP)

- Yet still powerful (actually universal): any function computable by a Turing machine can be computed by such a recurrent network of a finite size (see, e.g., Siegelmann and Sontag (1995))

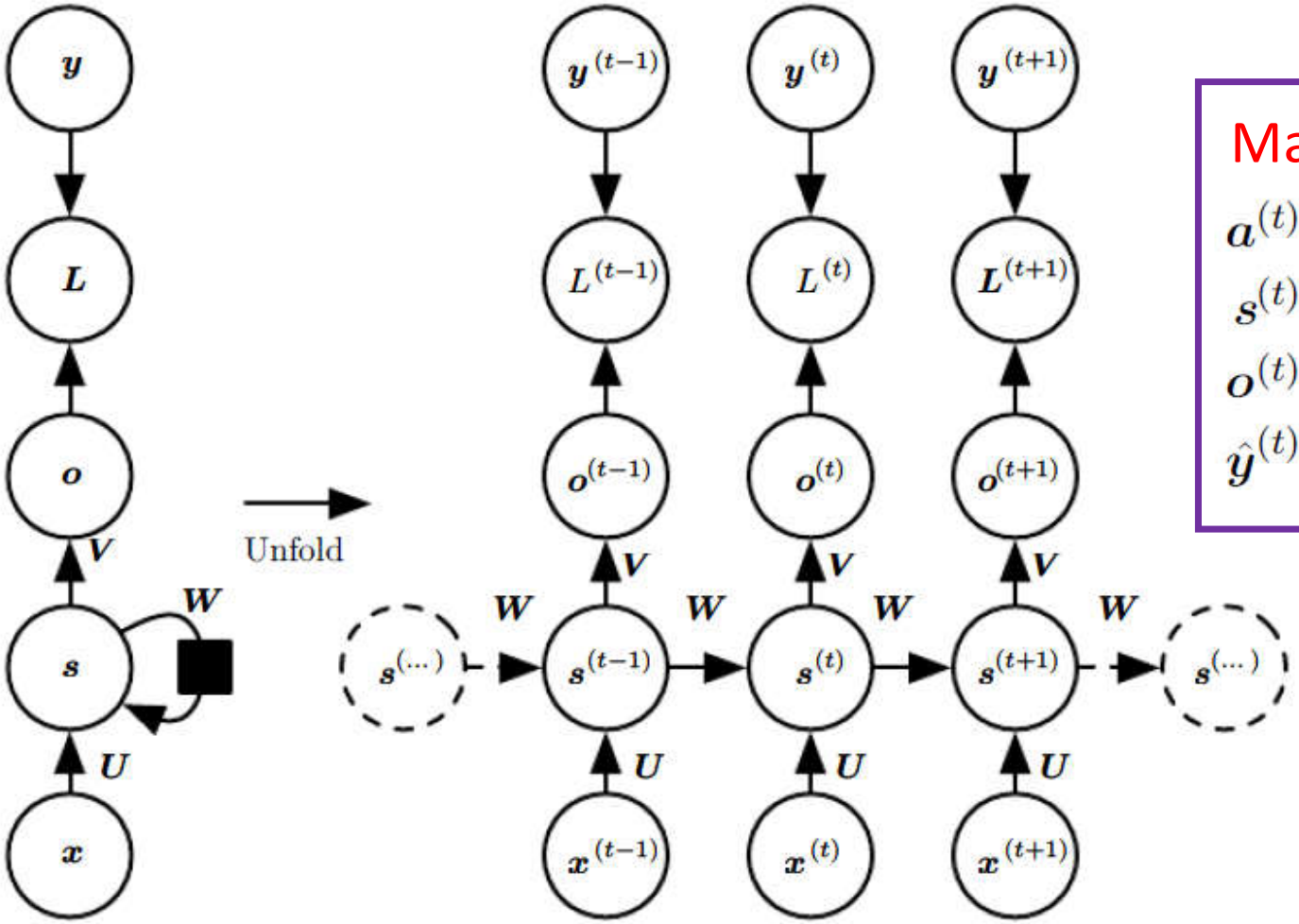
Training RNN

- Principle: unfold the computational graph, and use backpropagation
- Called back-propagation through time (BPTT) algorithm
- Can then apply any general-purpose gradient-based techniques

Training RNN

- Principle: unfold the computational graph, and use backpropagation
- Called back-propagation through time (BPTT) algorithm
- Can then apply any general-purpose gradient-based techniques

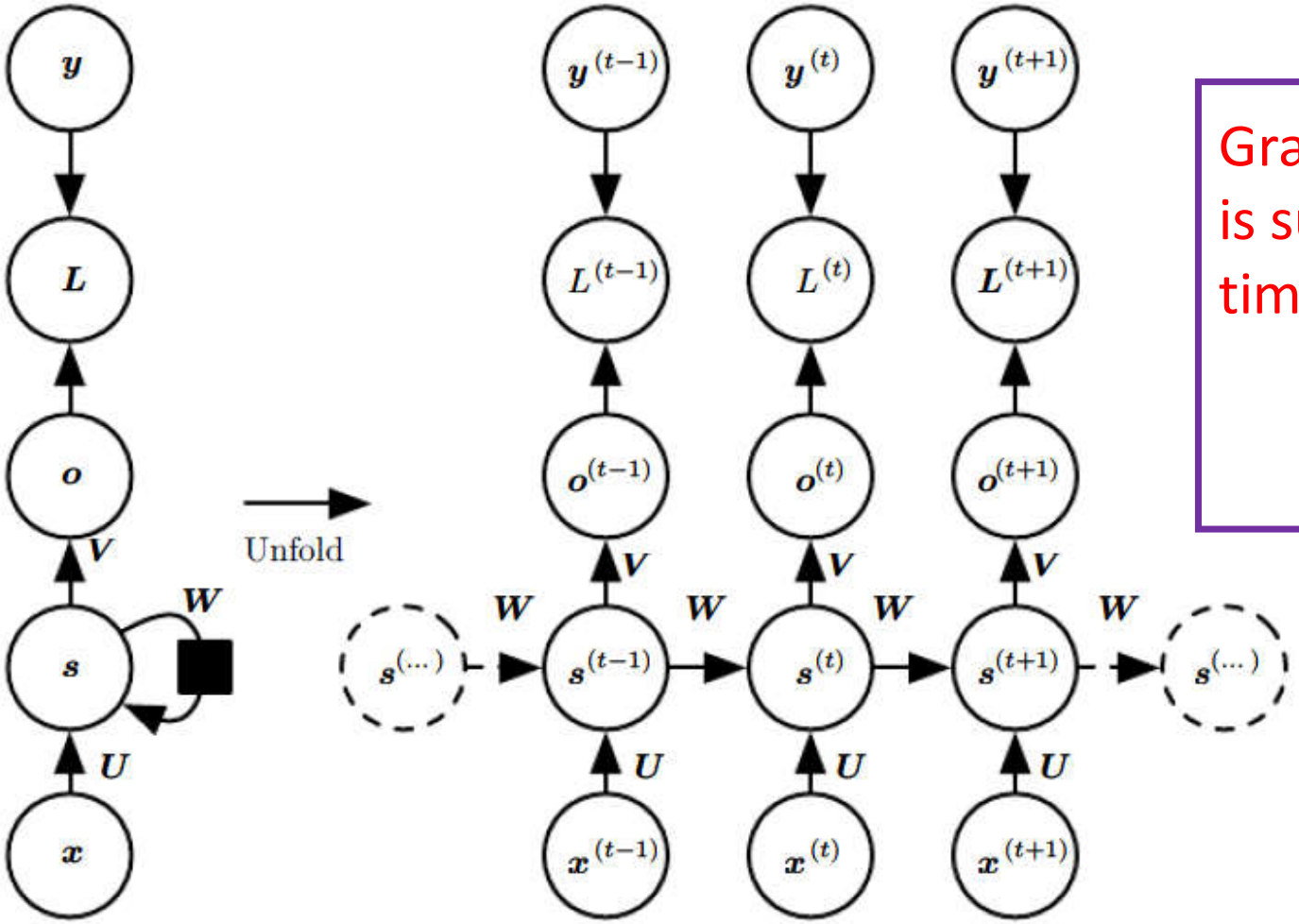
- Conceptually: first compute the gradients of the internal nodes, then compute the gradients of the parameters



Math formula:

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{s}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{s}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{s}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

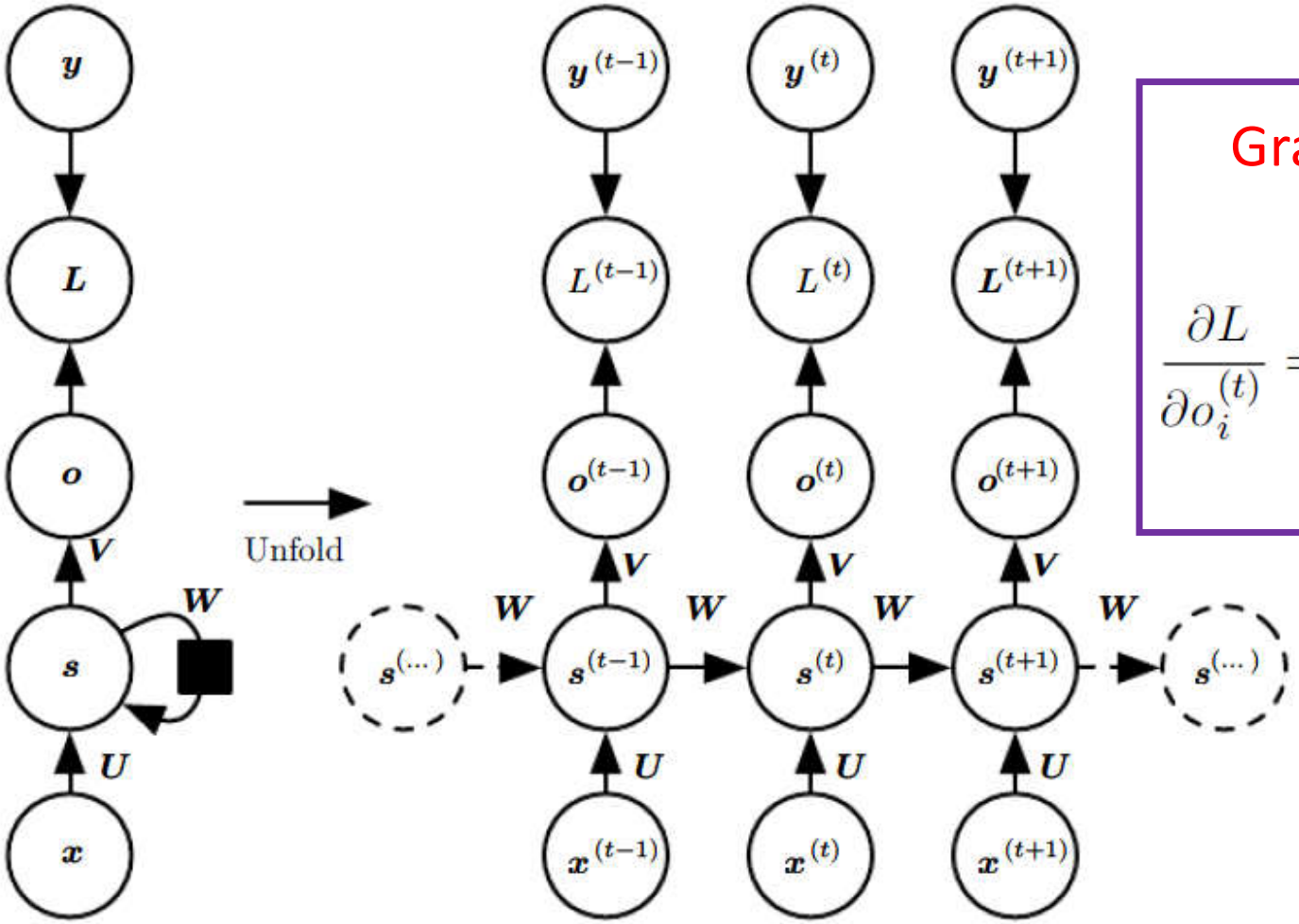
Figure from *Deep Learning*, Goodfellow, Bengio and Courville



Gradient at $L^{(t)}$: (total loss is sum of those at different time steps)

$$\frac{\partial L}{\partial L^{(t)}} = 1.$$

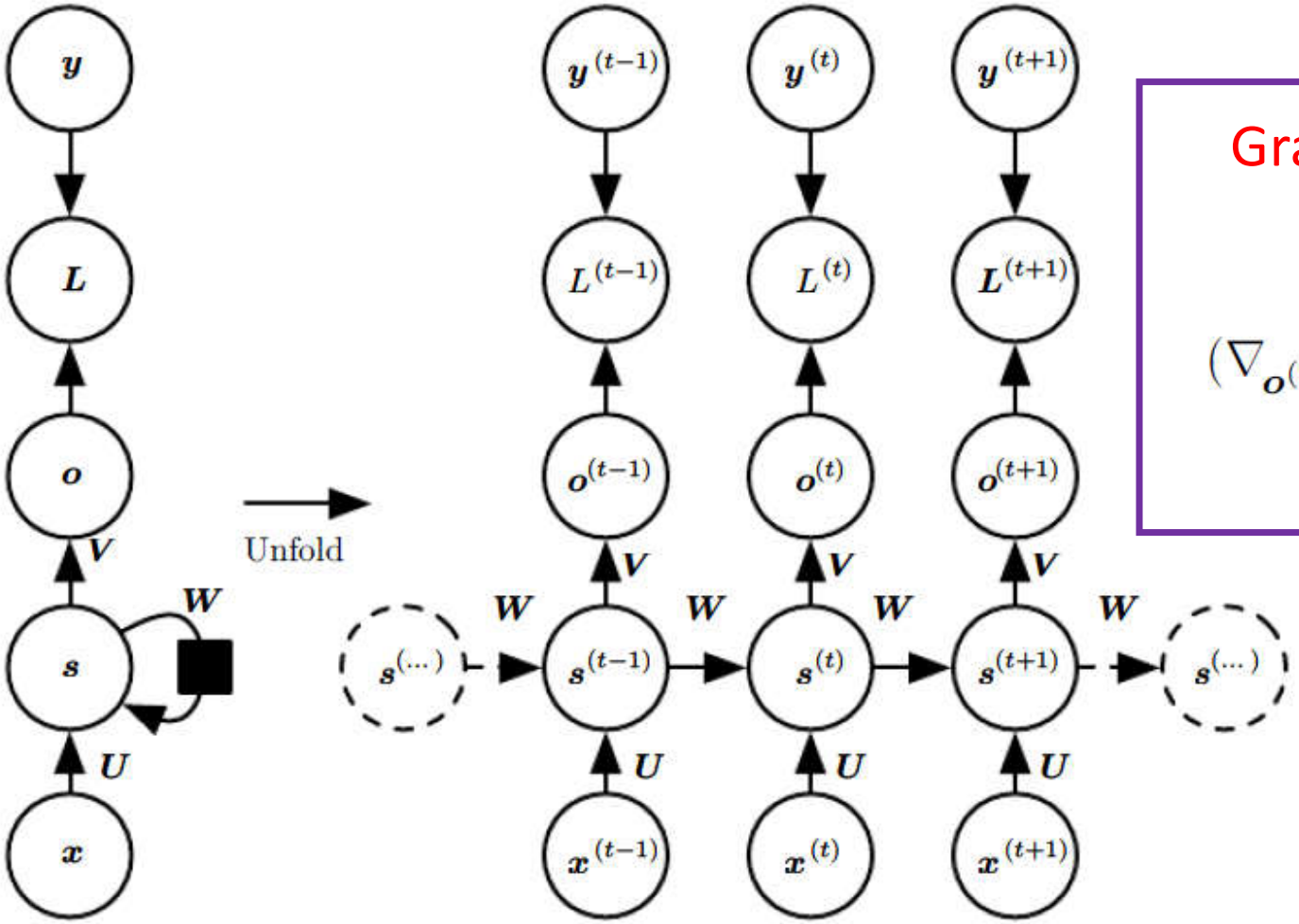
Figure from *Deep Learning*, Goodfellow, Bengio and Courville



Gradient at $o^{(t)}$:

$$\frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - 1_{i,y^{(t)}}$$

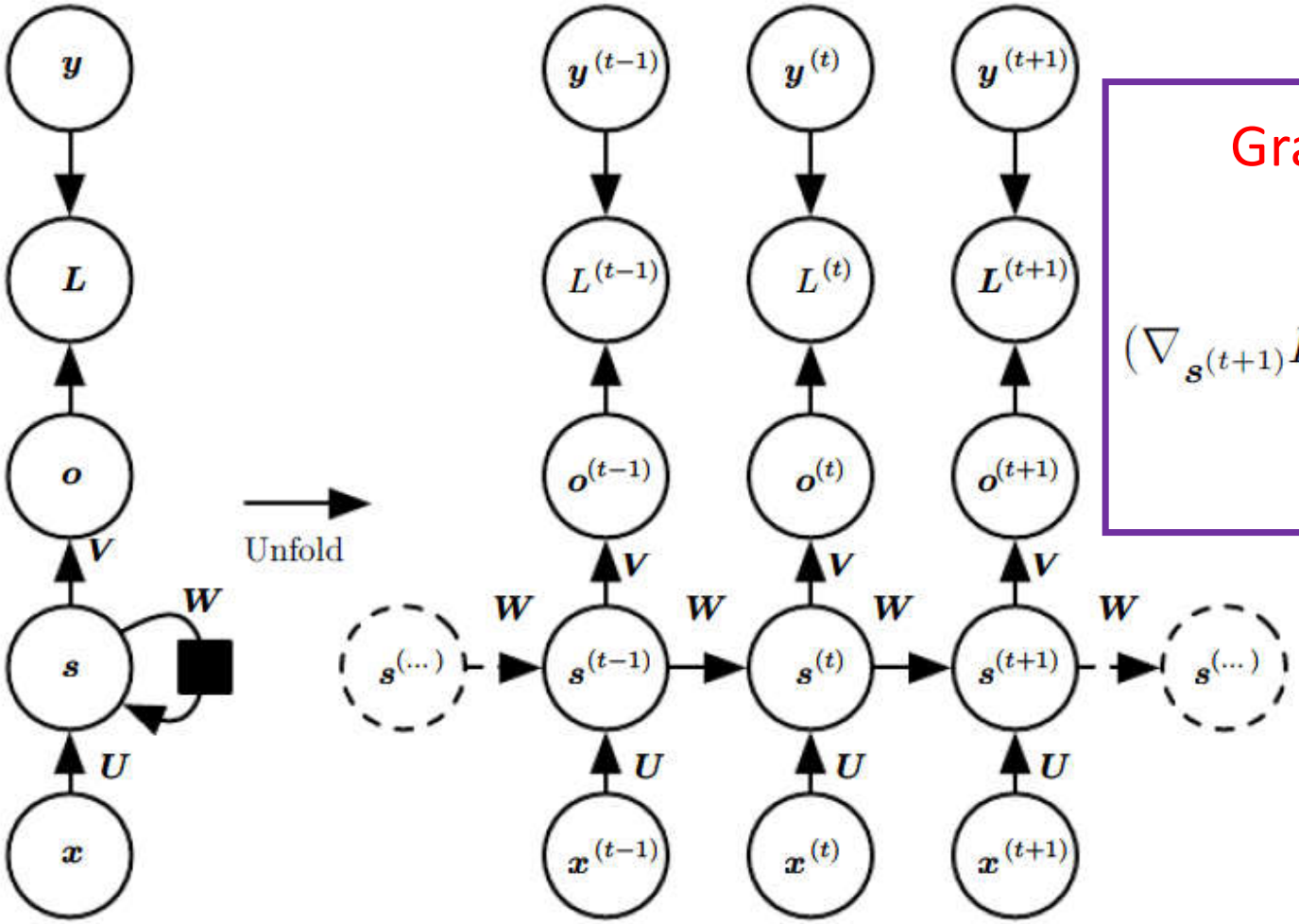
Figure from *Deep Learning*, Goodfellow, Bengio and Courville



Gradient at $s^{(\tau)}$:

$$(\nabla_{\boldsymbol{o}^{(\tau)}} L)\,\frac{\partial \boldsymbol{o}^{(\tau)}}{\partial \boldsymbol{s}^{(\tau)}} = (\nabla_{\boldsymbol{o}^{(\tau)}} L)\,\boldsymbol{V}$$

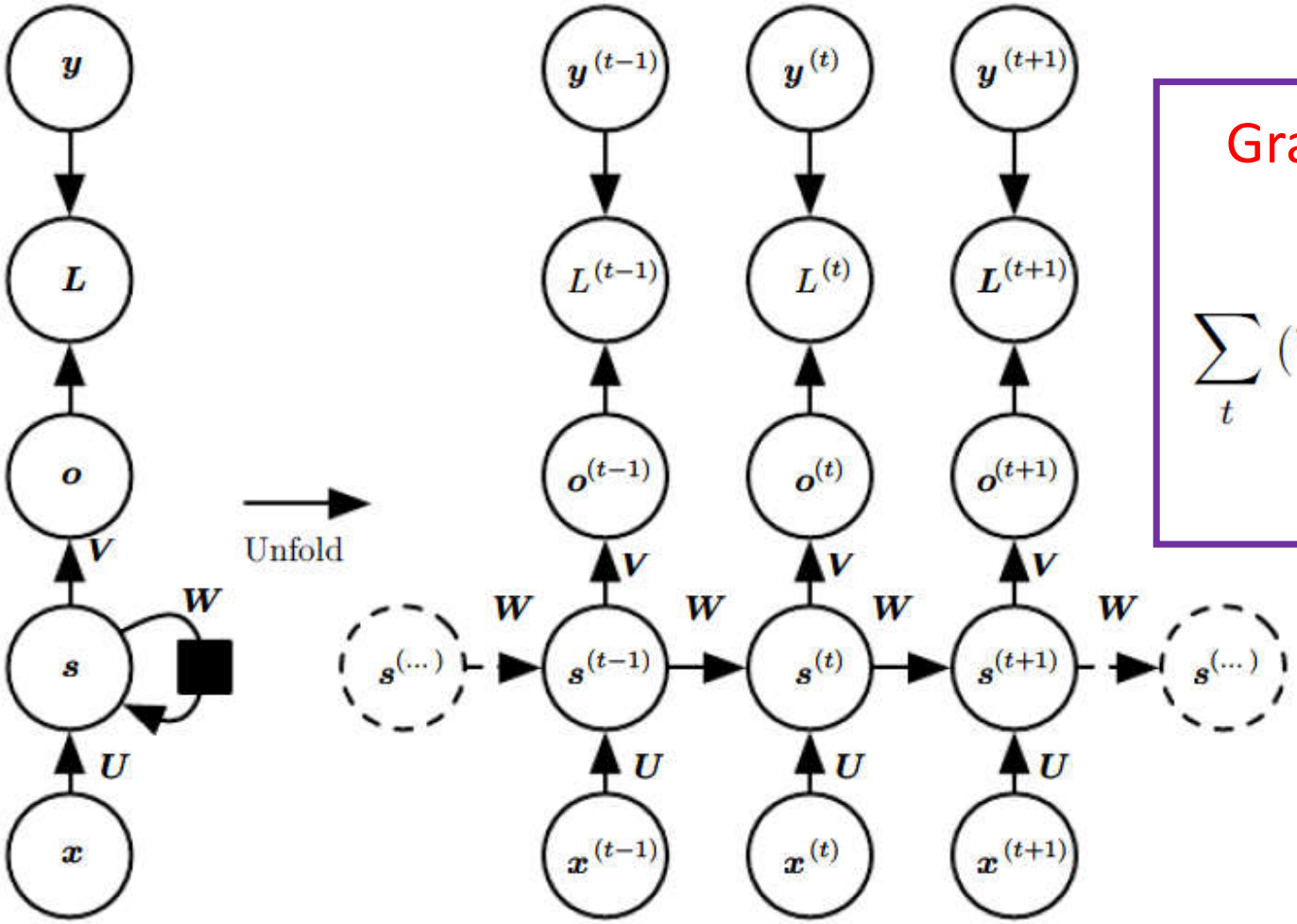
Figure from *Deep Learning*, Goodfellow, Bengio and Courville



Gradient at $s^{(t)}$:

$$(\nabla_{\boldsymbol{s}^{(t+1)}} L)\,\frac{\partial \boldsymbol{s}^{(t+1)}}{\partial \boldsymbol{s}^{(t)}} + (\nabla_{\boldsymbol{o}^{(t)}} L)\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{s}^{(t)}}$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville



Gradient at parameter V:

$$\sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{V}} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{s}^{(t)\top}$$

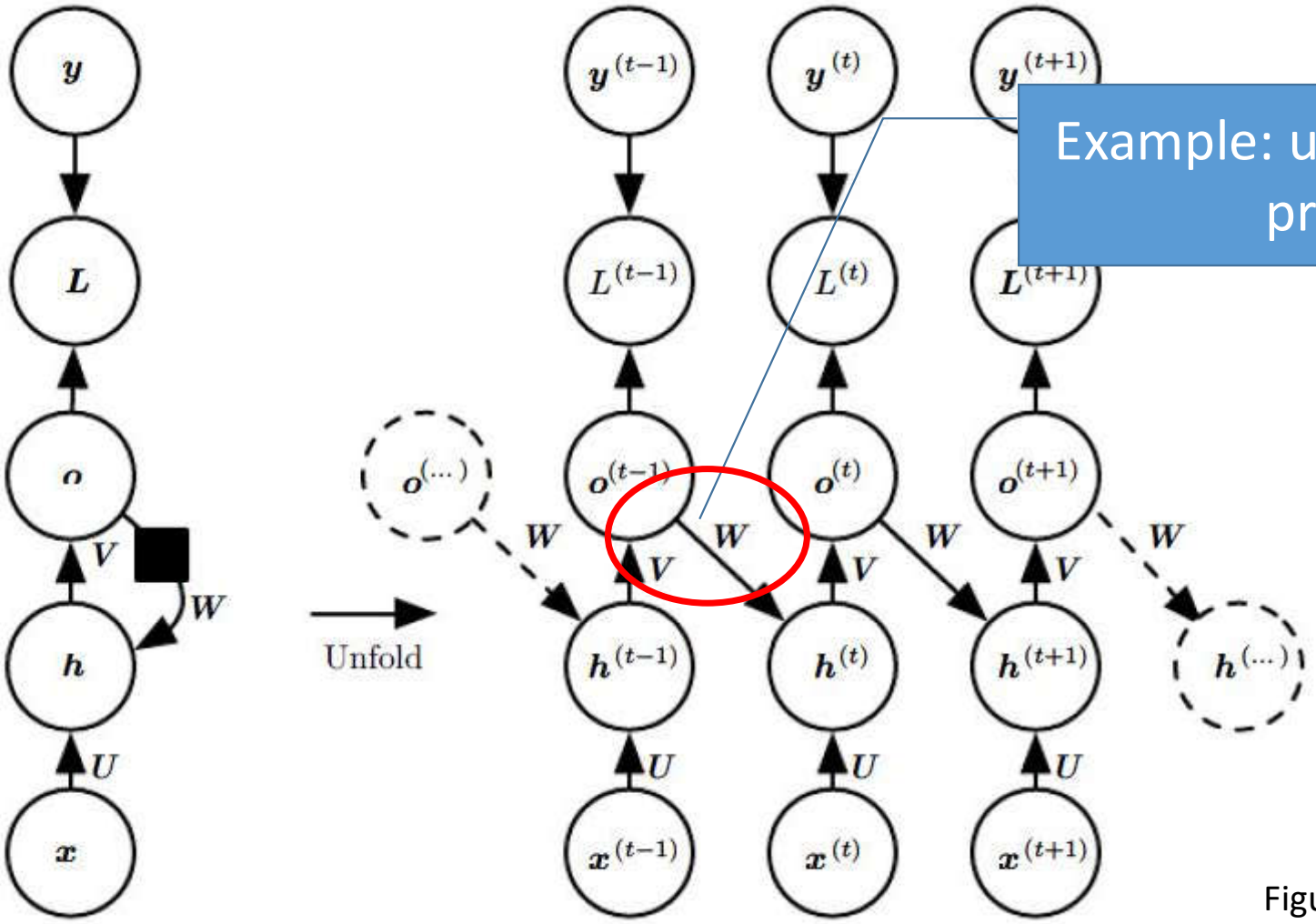
Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Variants of RNN

RNN

- Use the same computational function and parameters across different time steps of the sequence
- Each time step: takes the input entry and the previous hidden state to compute the output entry
- Loss: typically computed every time step

- Many variants
  - Information about the past can be in many other forms
  - Only output at the end of the sequence



Example: use the output at the previous step

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Example: only output at the end

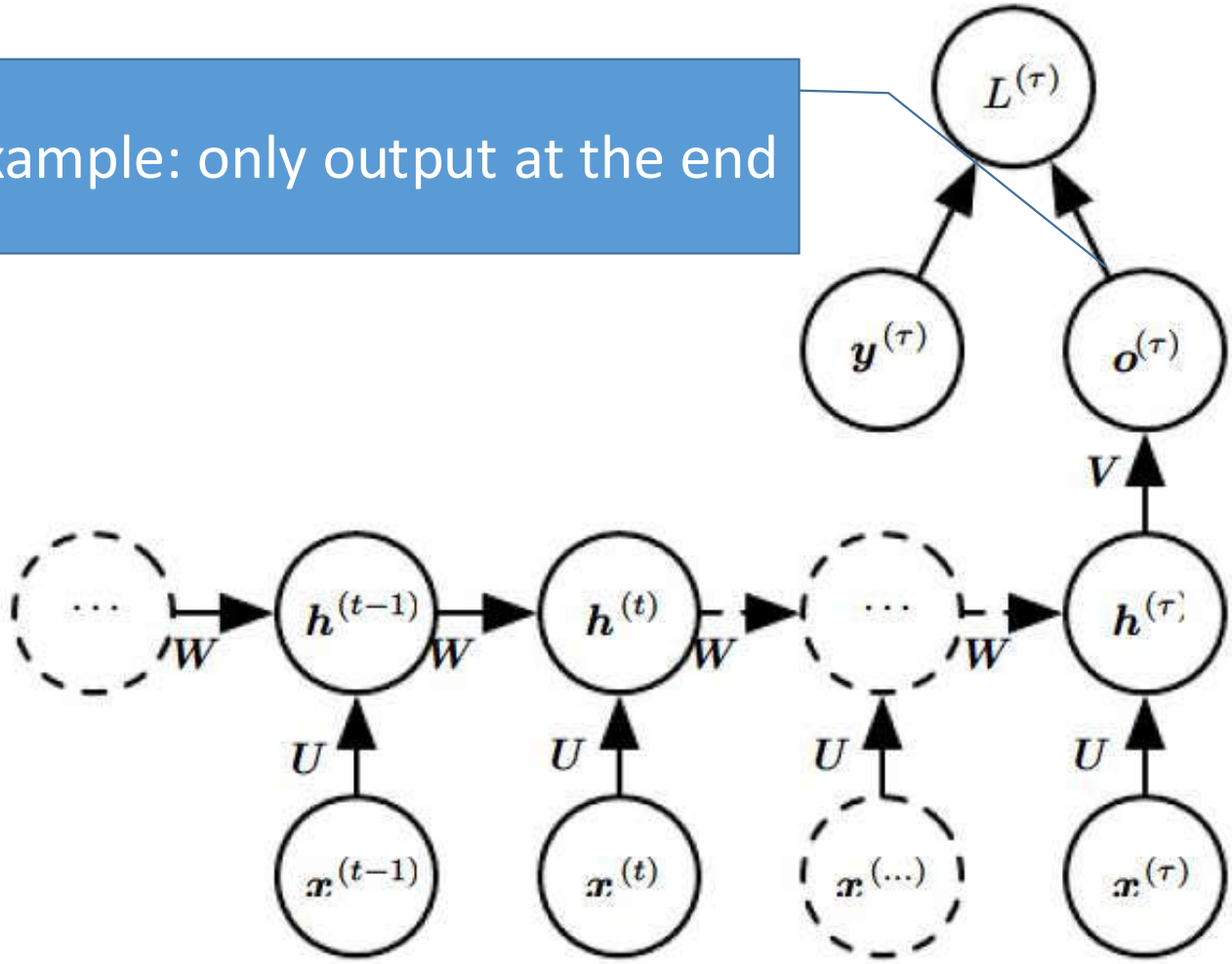


Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Bidirectional RNNs

- Many applications: output at time t may depend on the whole input sequence
- Example in speech recognition: correct interpretation of the current sound may depend on the next few phonemes, potentially even the next few words

- Bidirectional RNNs are introduced to address this

BiRNNs

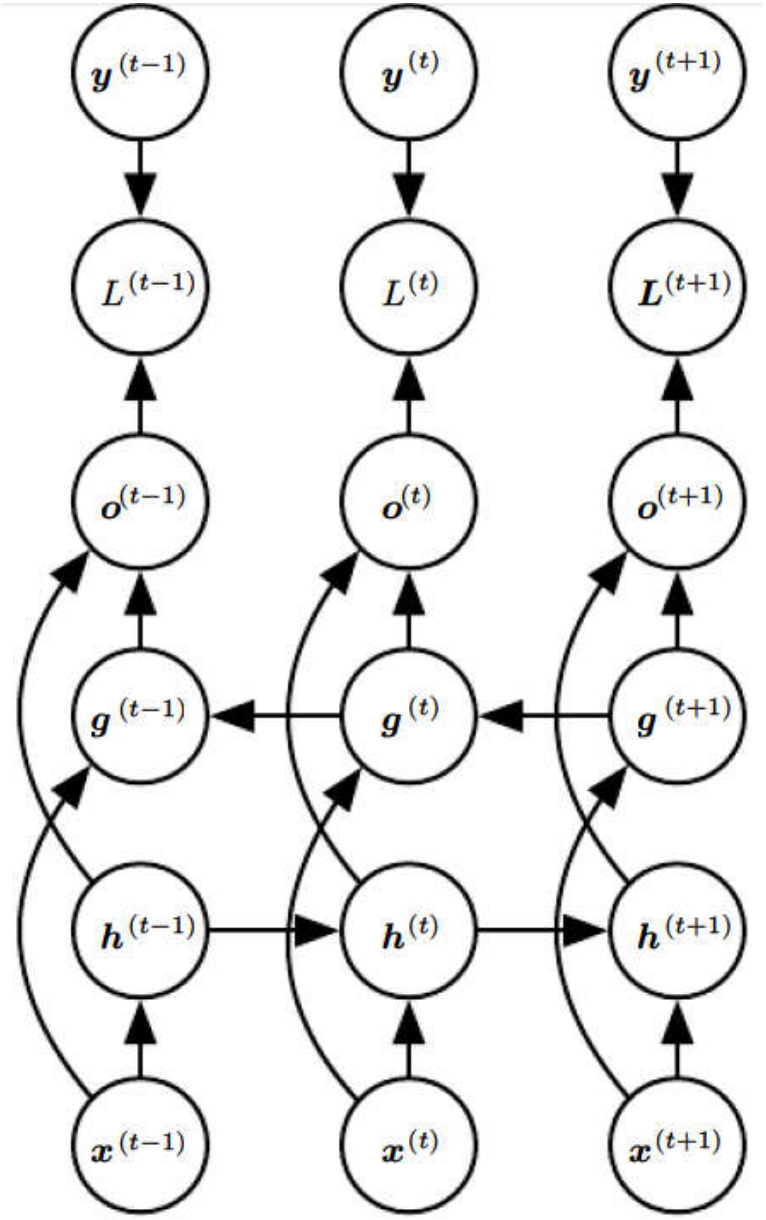


Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Encoder-decoder RNNs

- RNNs: can map sequence to one vector; or to sequence of same length
- What about mapping sequence to sequence of different length?
- Example: speech recognition, machine translation, question answering, etc

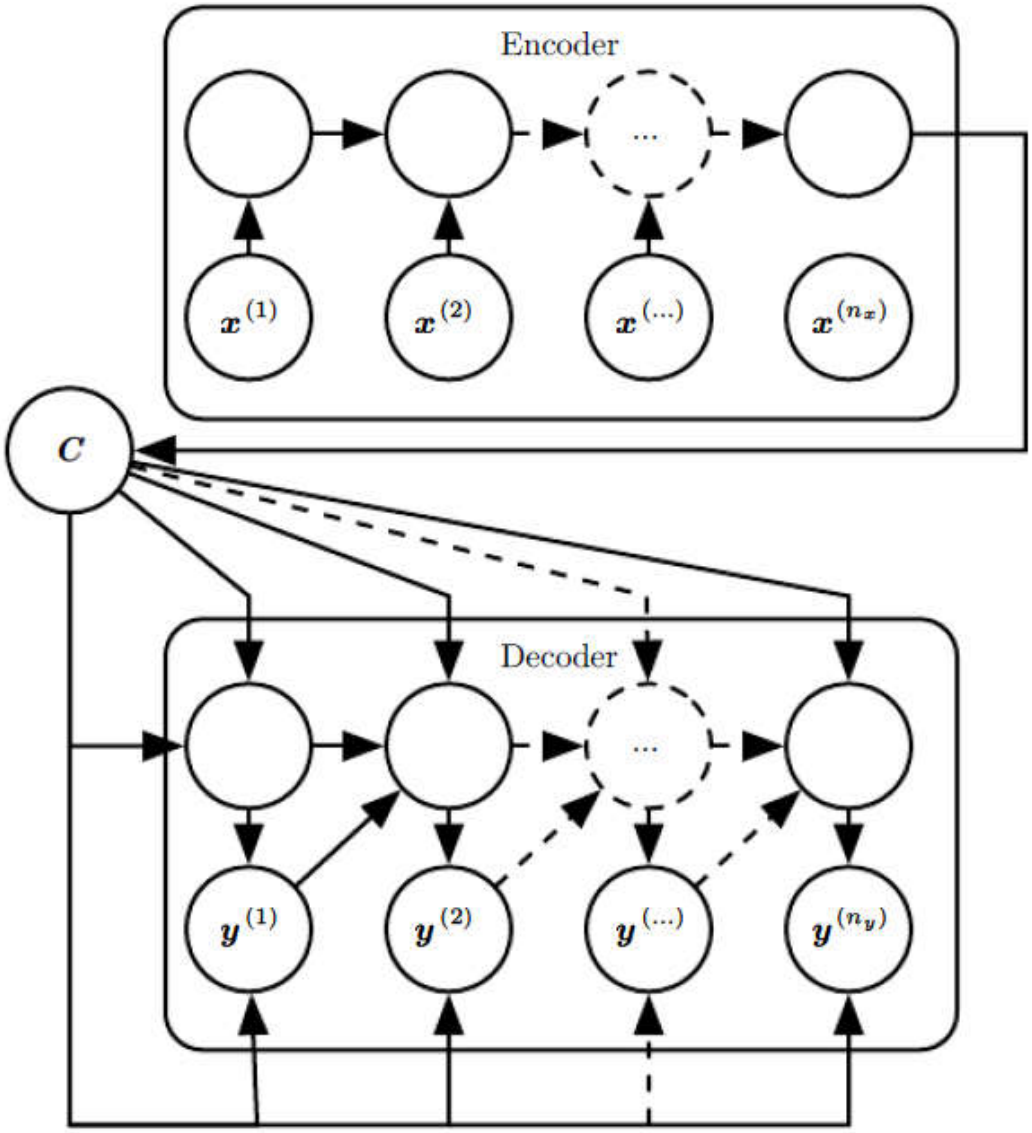


Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Deep Learning

DR. PRAVEEN BLESSINGTON T
Professor
Email: praveentblessington@gmail.com
Mobile: 9730562120

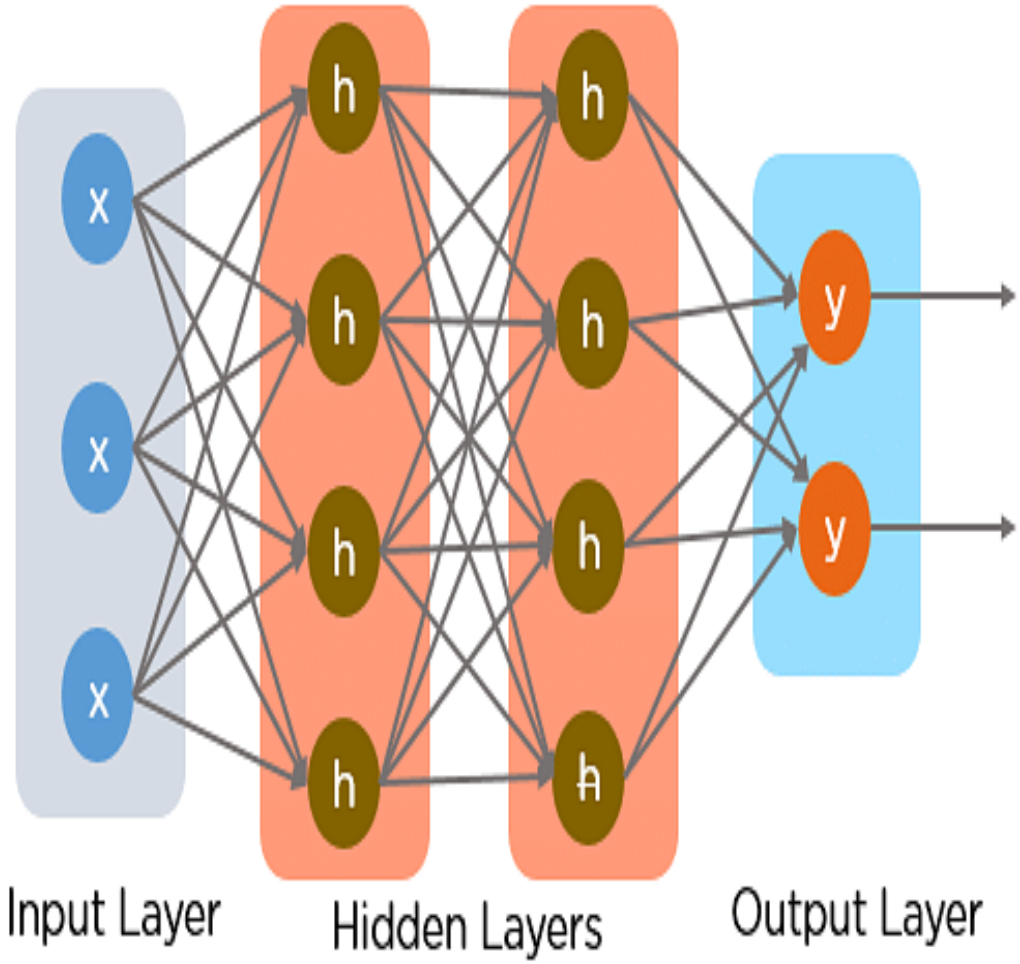
Recurrent Neural Networks

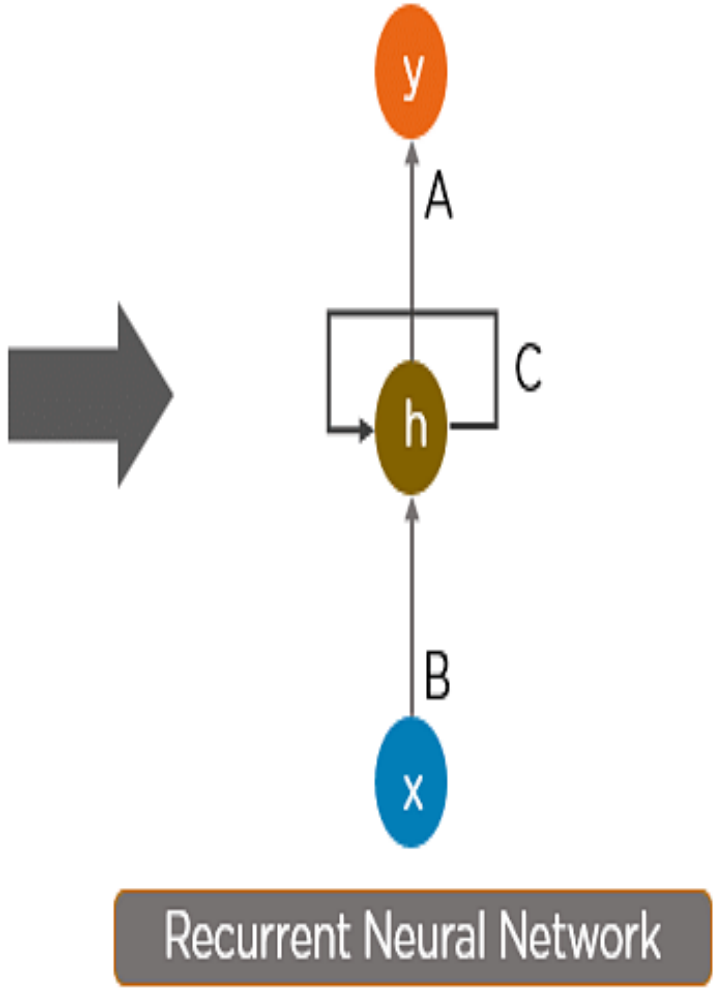
- Recurrent Neural Networks (RNNs) are a type of neural network that can process sequential data such as time series, speech, and text by maintaining a hidden state that captures information about the past inputs.
- RNNs are used in deep learning and in the development of models that simulate neuron activity in the human brain.
- RNNs have feedback connections that allow them to retain information from previous time steps.

Continues....

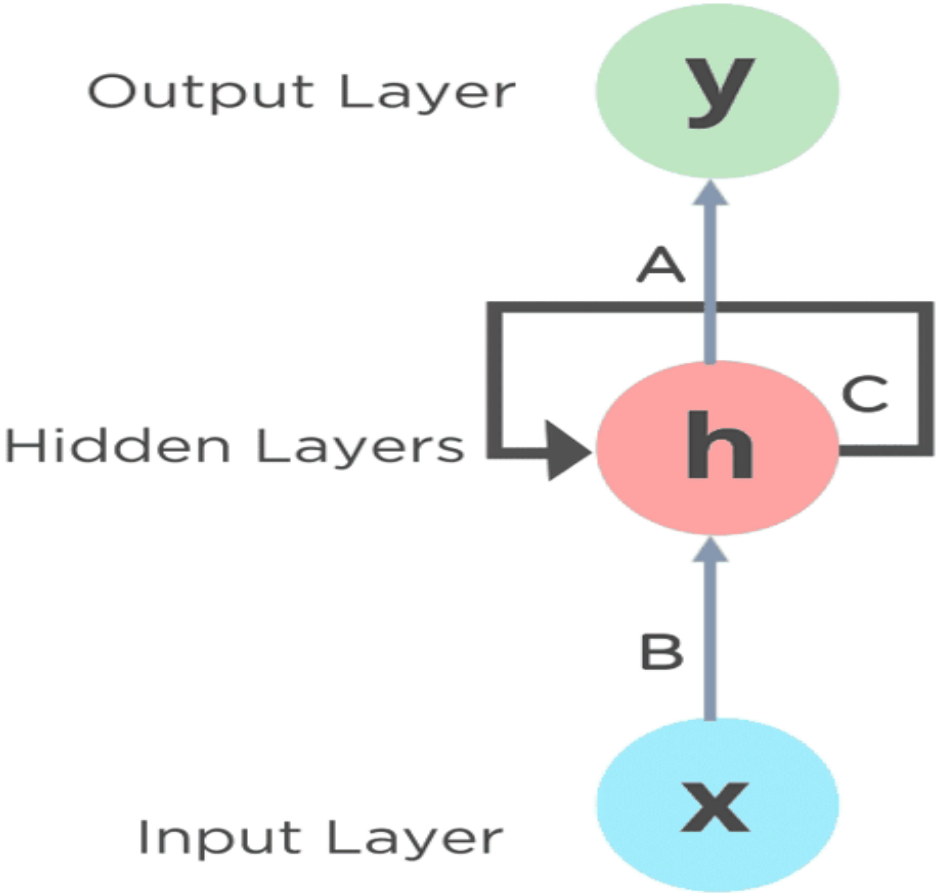
- This enables them to capture temporal dependencies. RNNs recognize data's sequential characteristics and use patterns to predict the next likely scenario.
- RNNs are often better to use for tasks that involve sequential inputs, such as speech and language. RNN models are mostly used in the fields of natural language processing and speech recognition.
- RNNs are derived from feedforward networks. They exhibit similar behavior to how human brains function.

Continues....


x
x
x
h
h
h
h
h
h
h
h
y
y
Input Layer
Hidden Layers
Output Layer


y
A
C
h
B
x
Recurrent Neural Network

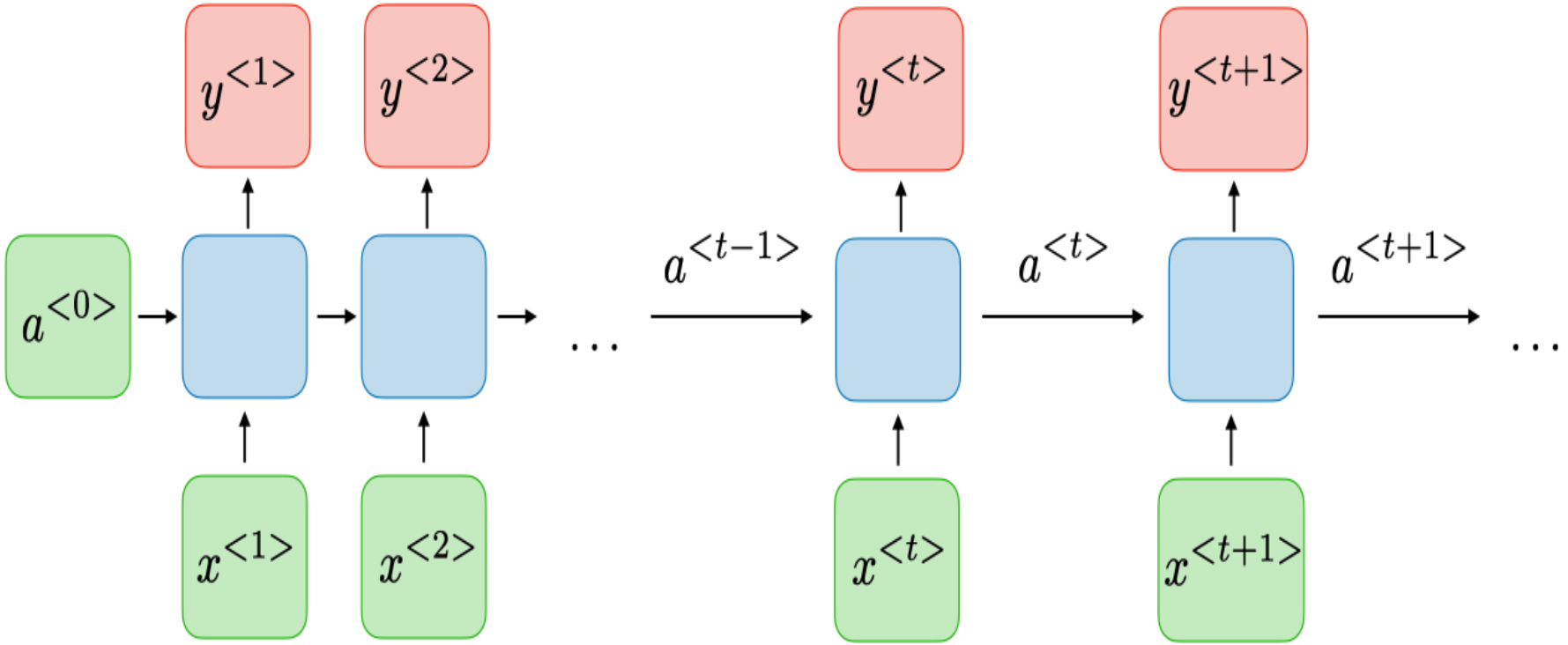
Continues....



A, B and C are the parameters

Architecture of a Traditional RNN

- RNNs are a type of neural network that has hidden states and allows past outputs to be used as inputs.

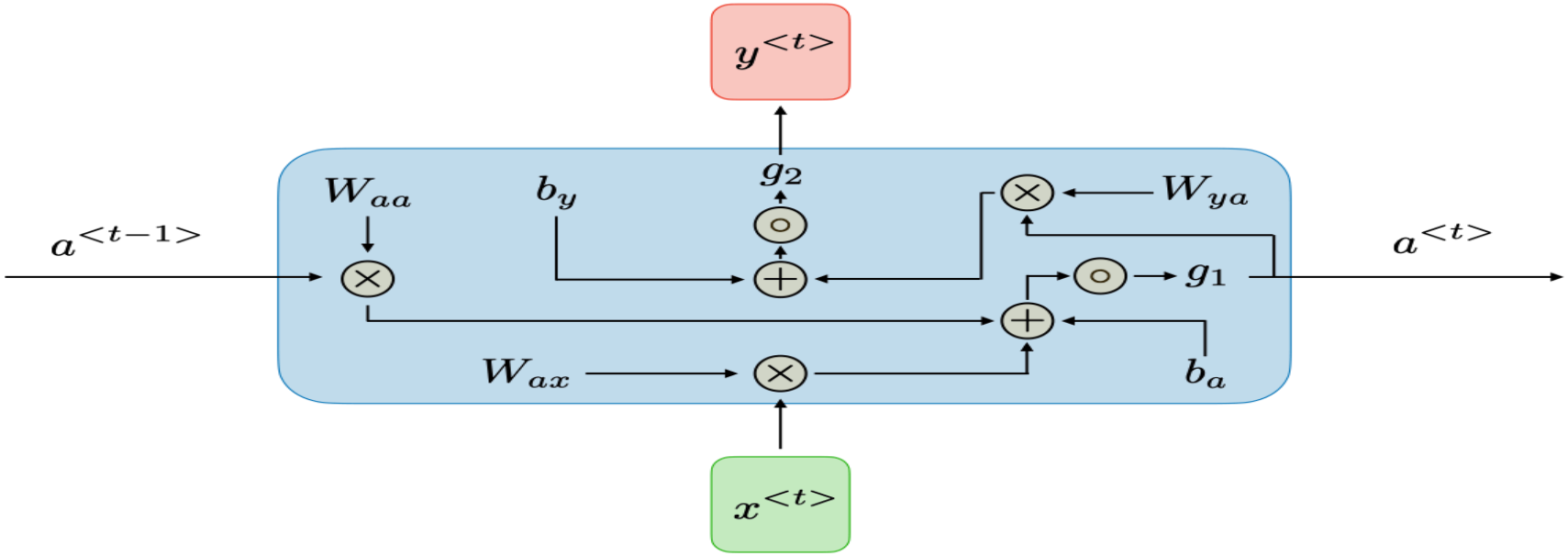


Continues...

For each timestep t, the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows:

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a) \quad \text{and} \quad y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

where $W_{ax}, W_{aa}, W_{ya}, b_a, b_y$ are coefficients that are shared temporally and g_1, g_2 activation functions.



How RNN works

- The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit.
- This hidden state signifies the past knowledge that the network currently holds at a given time step.
- This hidden state is updated at every time step to signify the change in the knowledge of the network about the past.
- The hidden state is updated using the following recurrence relation:-

Continues....

The formula for calculating the current state:

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

where:

h_t -> current state
h_{t-1} -> previous state
x_t -> input state

Formula for applying Activation function(tanh):

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

where:

w_{hh} -> weight at recurrent neuron
w_{xh} -> weight at input neuron

Continues....

The formula for calculating output:

$$y_t = W_{hy} h_t$$

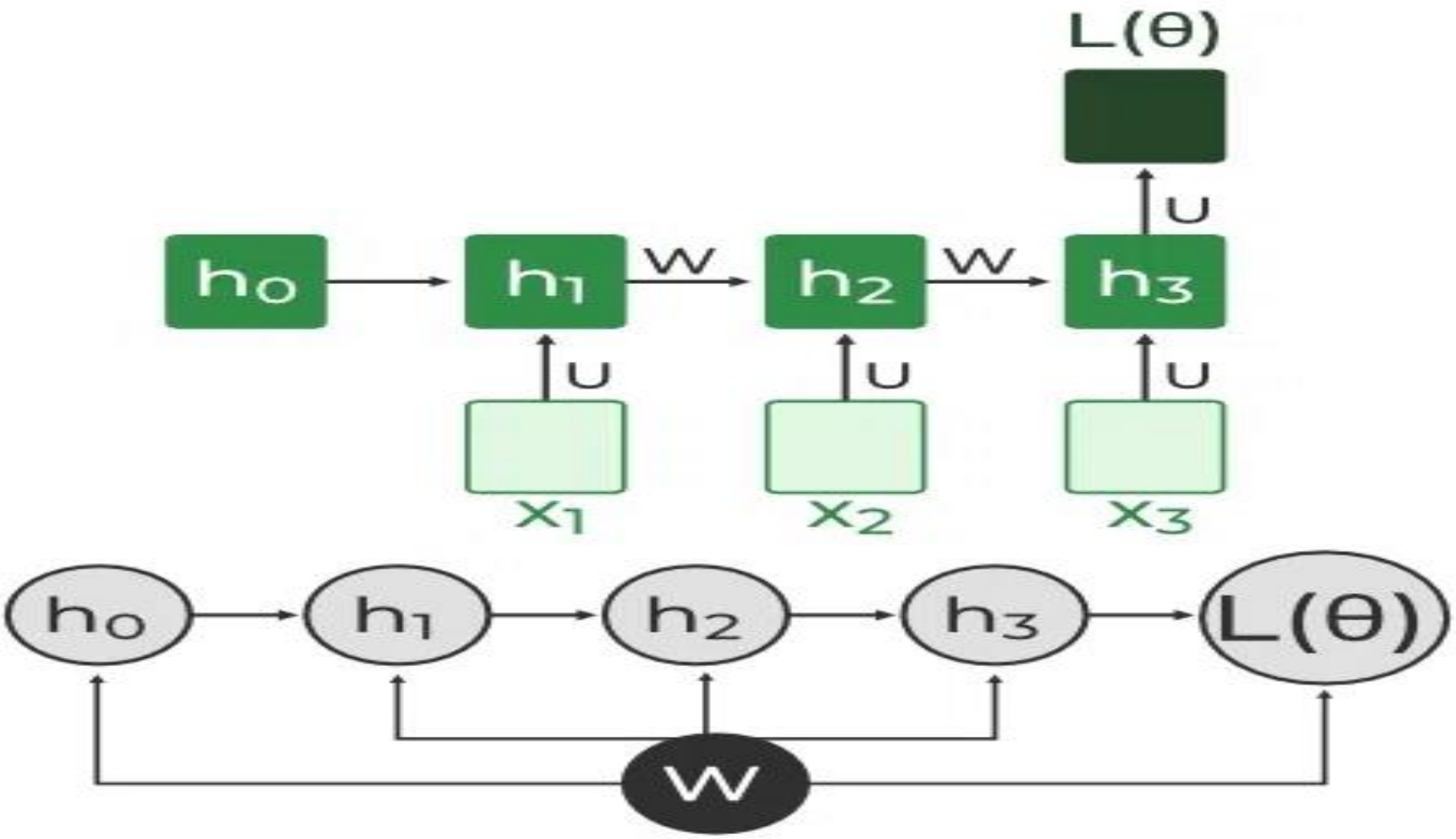
```
Y_t -> output
W_hy -> weight at output layer
```

These parameters are updated using Backpropagation. However, since RNN works on sequential data here we use an updated backpropagation which is known as Backpropagation through time.

Backpropagation Through Time (BPTT)

- In RNN the neural network is in an ordered fashion and since in the ordered network each variable is computed one at a time in a specified order like first h1 then h2 then h3 so on.
- Hence we will apply backpropagation throughout all these hidden time states sequentially.

Continues....



Continues....

- L(θ)(loss function) depends on h3
- h3 in turn depends on h2 and W
- h2 in turn depends on h1 and W
- h1 in turn depends on ho and W
- where ho is a constant starting state.

$$\frac{\partial \mathbf{L}(\theta)}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathbf{L}(\theta)}{\partial W}$$

For simplicity of this equation, we will apply backpropagation on only one row

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \frac{\partial h_3}{\partial W}$$

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \sum_{k=1}^{3} \frac{\partial h_3}{\partial h_k} \cdot \frac{\partial h_k}{\partial W}$$

This algorithm is called backpropagation through time (BPTT) as we backpropagate over all previous time steps

Training through RNN

- A single-time step of the input is provided to the network.
- Then calculate its current state using a set of current input and the previous state.
- The current h_t becomes h_{t-1} for the next time step.
- One can go as many time steps as possible according to the problem and join the information from all the previous states.

Continues….

- Once all the time steps are completed the final current state is used to calculate the output.
- The output is then compared to the actual output i.e the target output and the error is generated.
- The error is then back-propagated to the network to update the weights and hence the network (RNN) is trained using Backpropagation through time.

Advantages

1. An RNN remembers each and every piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short-Term Memory(LSTM).
2. Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.

Disadvantages

1. Gradient vanishing and exploding problems.
2. Training an RNN is a very difficult task.
3. It cannot process very long sequences if using tanh or relu as an activation function.

Applications

- Language Modelling and Generating Text
- Speech Recognition
- Machine Translation
- Image Recognition, Face detection
- Time series Forecasting

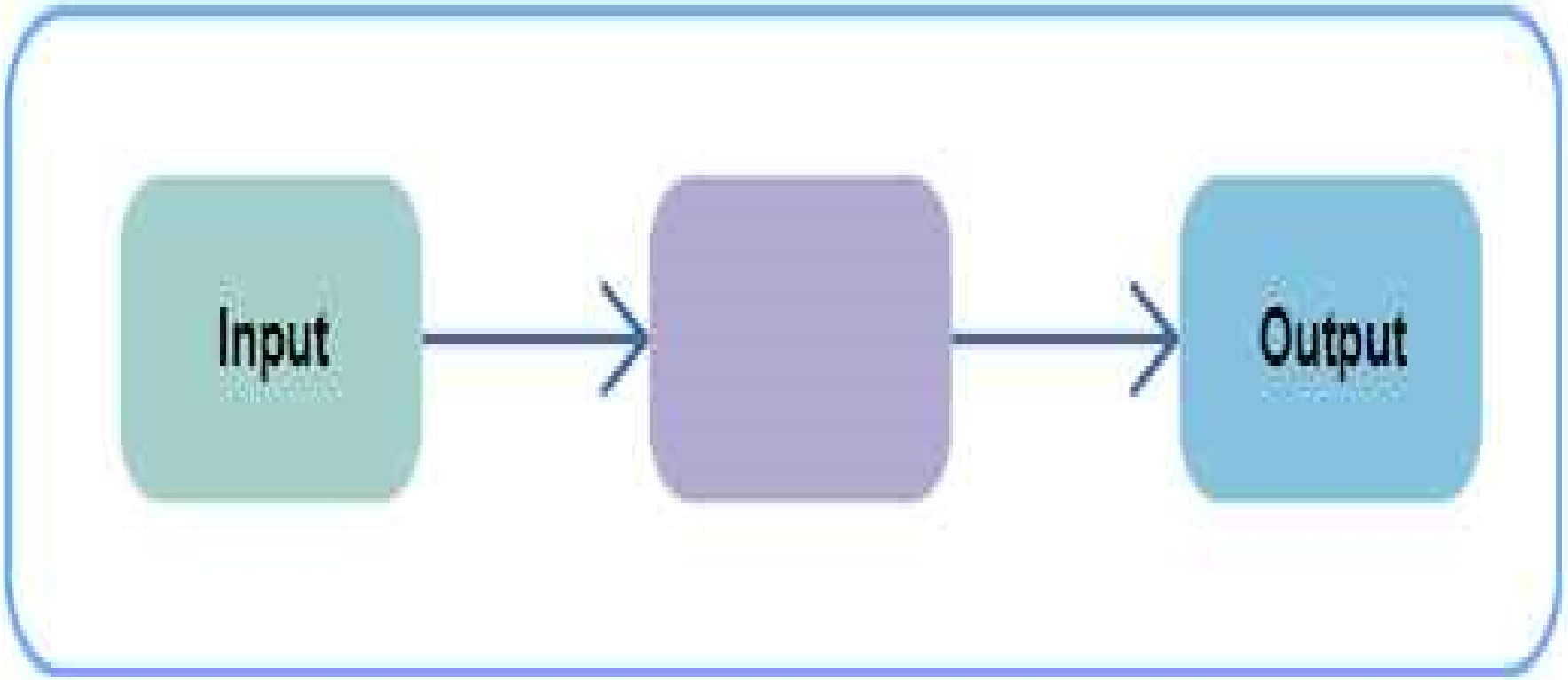
Types Of RNN

- There are four types of RNNs based on the number of inputs and outputs in the network.
  1. One to One
  2. One to Many
  3. Many to One
  4. Many to Many

1. One to One

- The simplest type of RNN is One-to-One, which allows a single input and a single output. It has fixed input and output sizes and acts as a traditional neural network.
- The One-to-One application can be found in Image Classification.

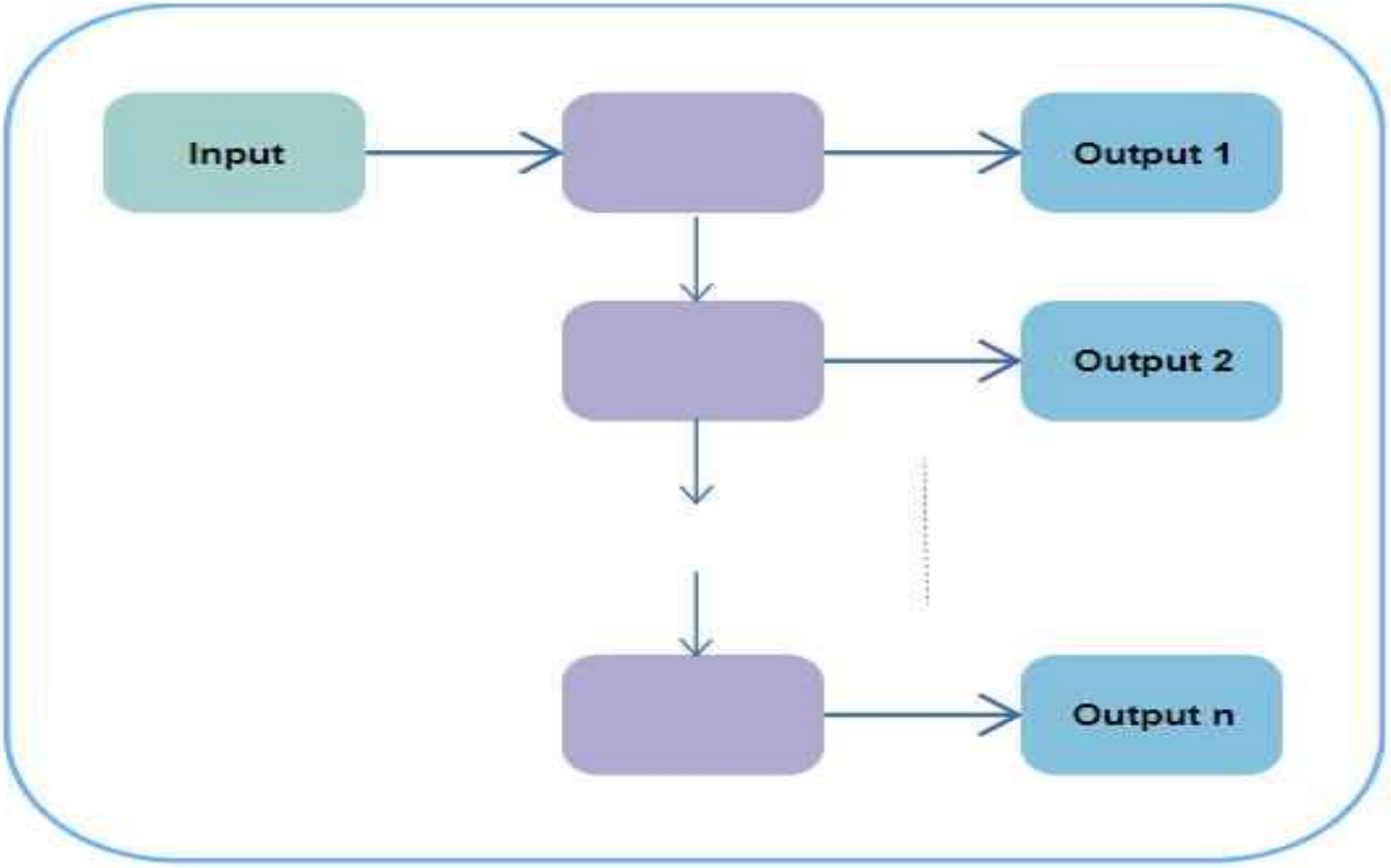
Continues....



2. One to Many

- One-to-Many is a type of RNN that gives multiple outputs when given a single input.
- It takes a fixed input size and gives a sequence of data outputs.
- Its applications can be found in Music Generation and Image Captioning.

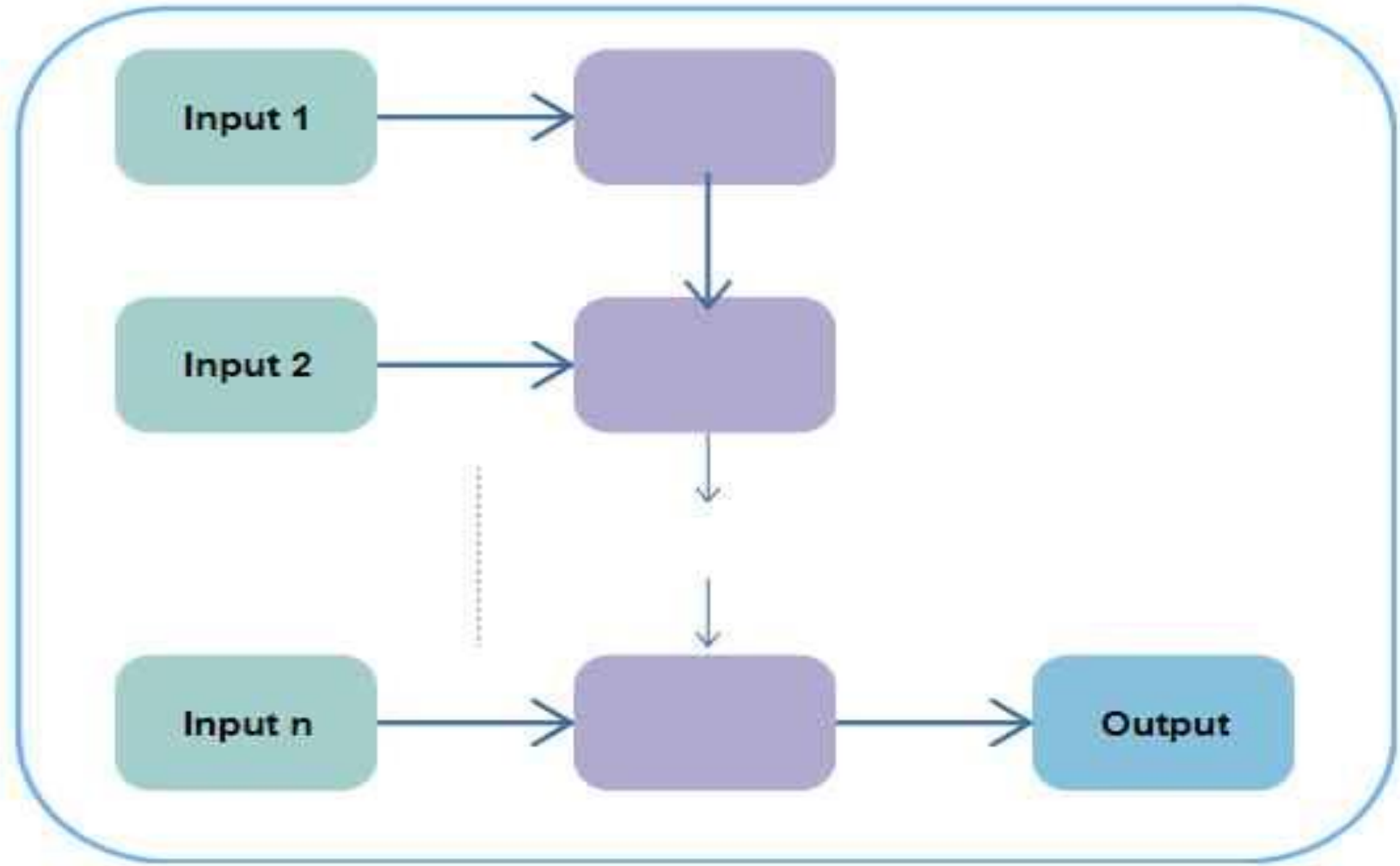
Continues....



3. Many to One

- Many-to-One is used when a single output is required from multiple input units or a sequence of them.
- It takes a sequence of inputs to display a fixed output.
- Sentiment Analysis is a common example of this type of Recurrent Neural Network.

Continues....

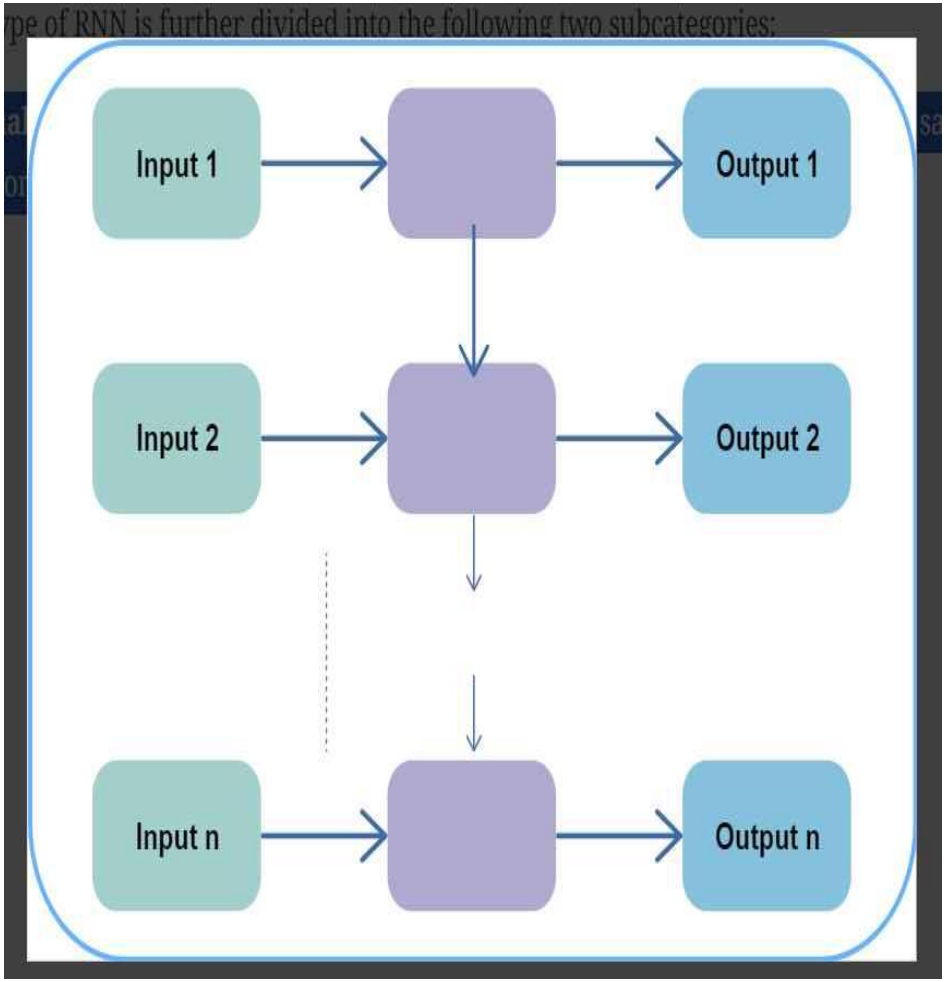


4. Many to Many

- Many-to-Many is used to generate a sequence of output data from a sequence of input units.
  - This type of RNN is further divided into the following two subcategories:
    - **Equal Unit Size**
    - **Unequal Unit Size**

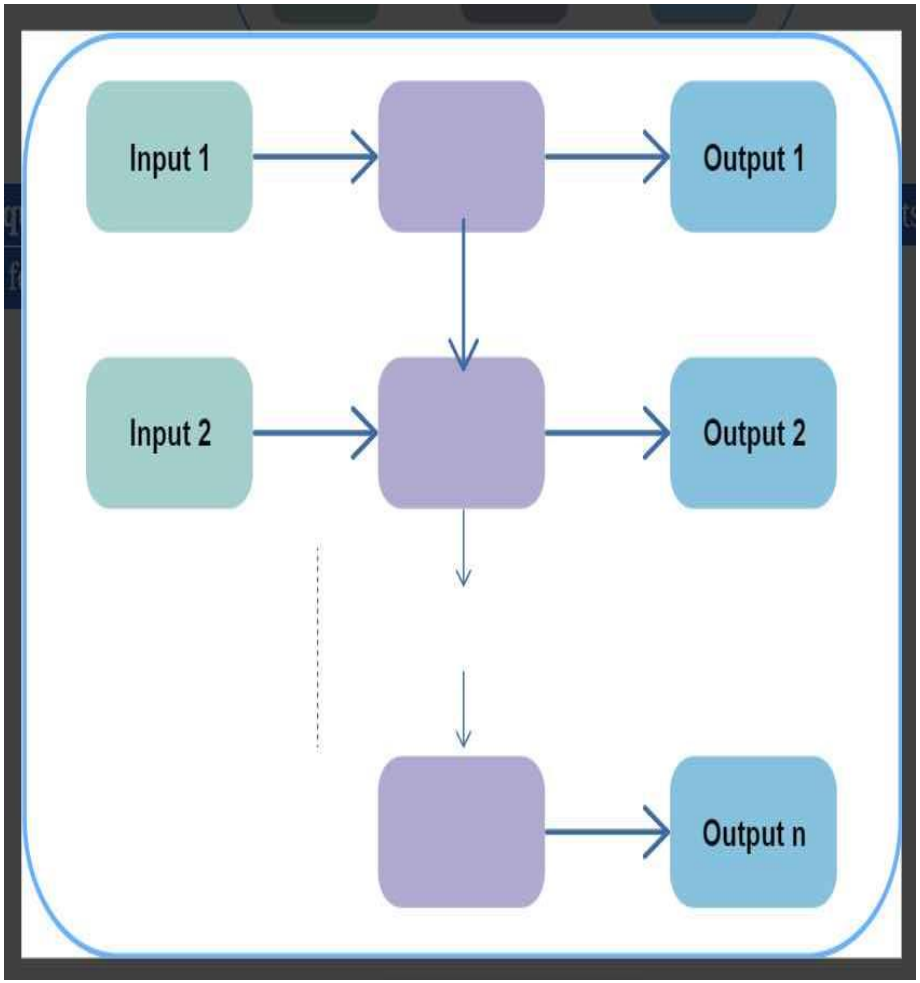
Continues....

- **Equal Unit Size:** In this case, the number of both the input and output units is the same.
- A common application can be found in Name-Entity Recognition.



Continues....

- **Unequal Unit Size:** In this case, inputs and outputs have different numbers of units.
- Its application can be found in Machine Translation.

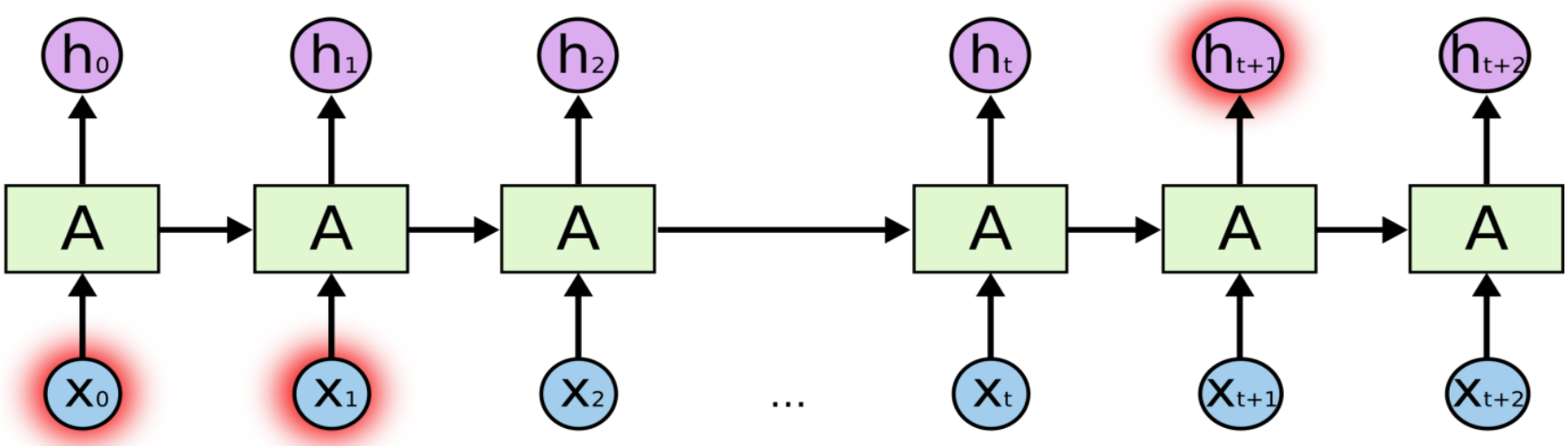


Vanishing/Exploding Gradient Problem

- Backpropagated errors multiply at each layer, resulting in exponential decay (if derivative is small) or growth (if derivative is large).
- Makes it very difficult train deep networks, or simple recurrent networks over many time steps.

Long Distance Dependencies

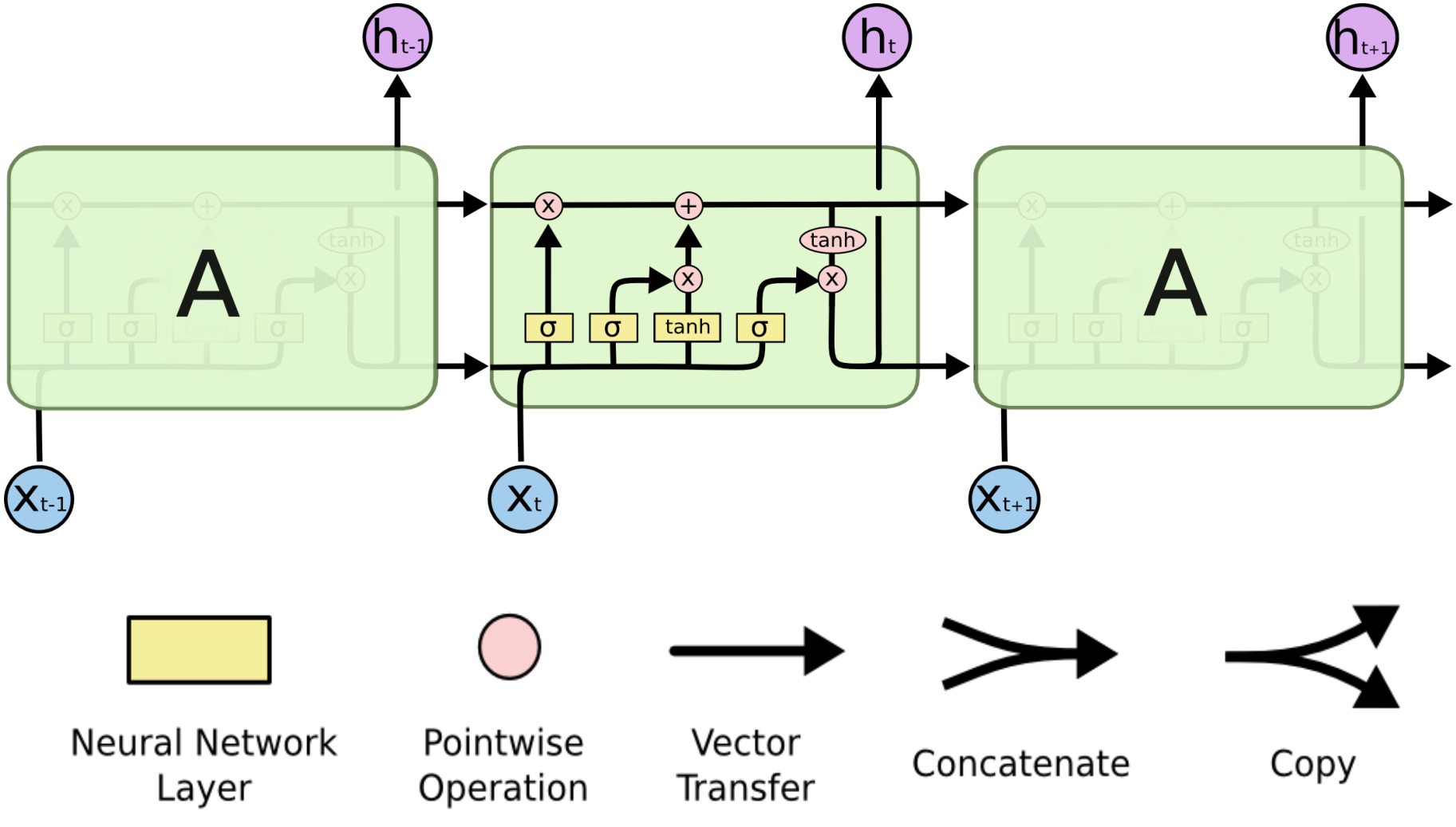
- It is very difficult to train SRNs to retain information over many time steps
- This make is very difficult to learn SRNs that handle long-distance dependencies, such as subject-verb agreement.



Long Short Term Memory(LSTM)

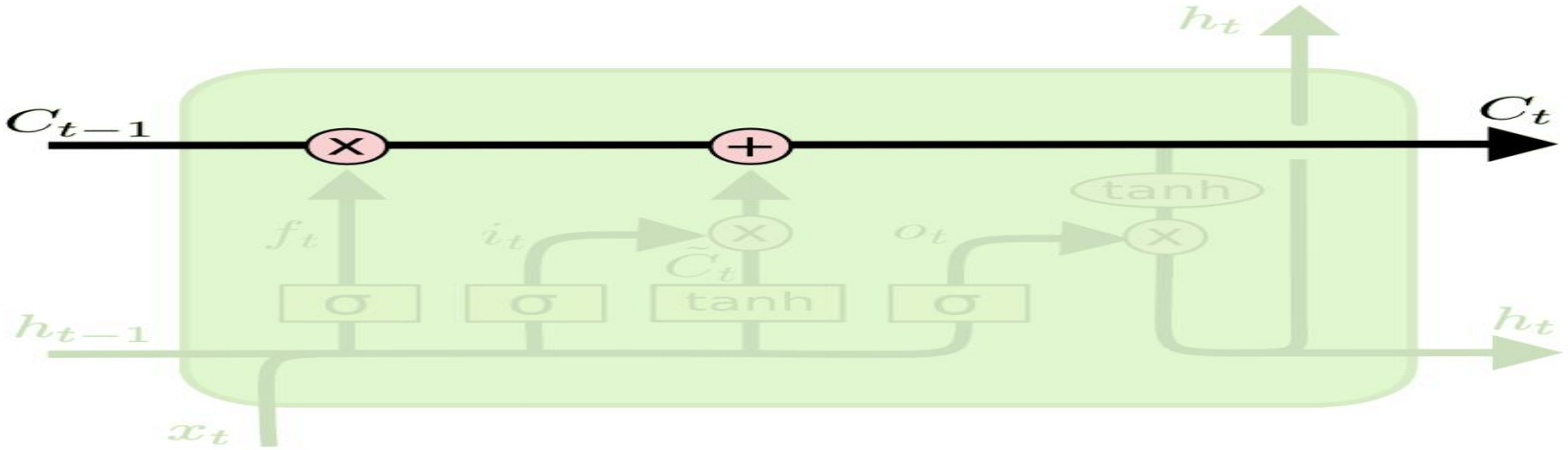
- LSTM networks, add additional gating units in each memory cell.
  - Forget gate
  - Input gate
  - Output gate
- Prevents vanishing/exploding gradient problem and allows network to retain state information over longer periods of time.

LSTM Network Architecture



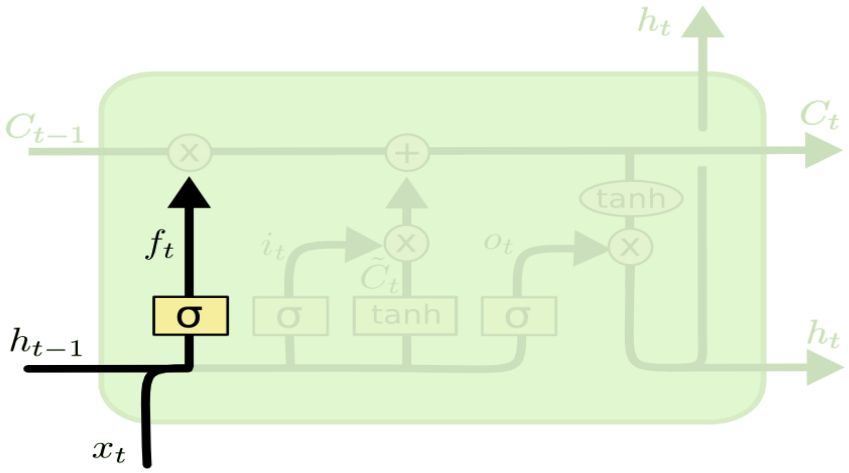
Cell State

- Maintains a vector C_t that is the same dimensionality as the hidden state, h_t
- Information can be added or deleted from this state vector via the forget and input gates.



Forget Gate

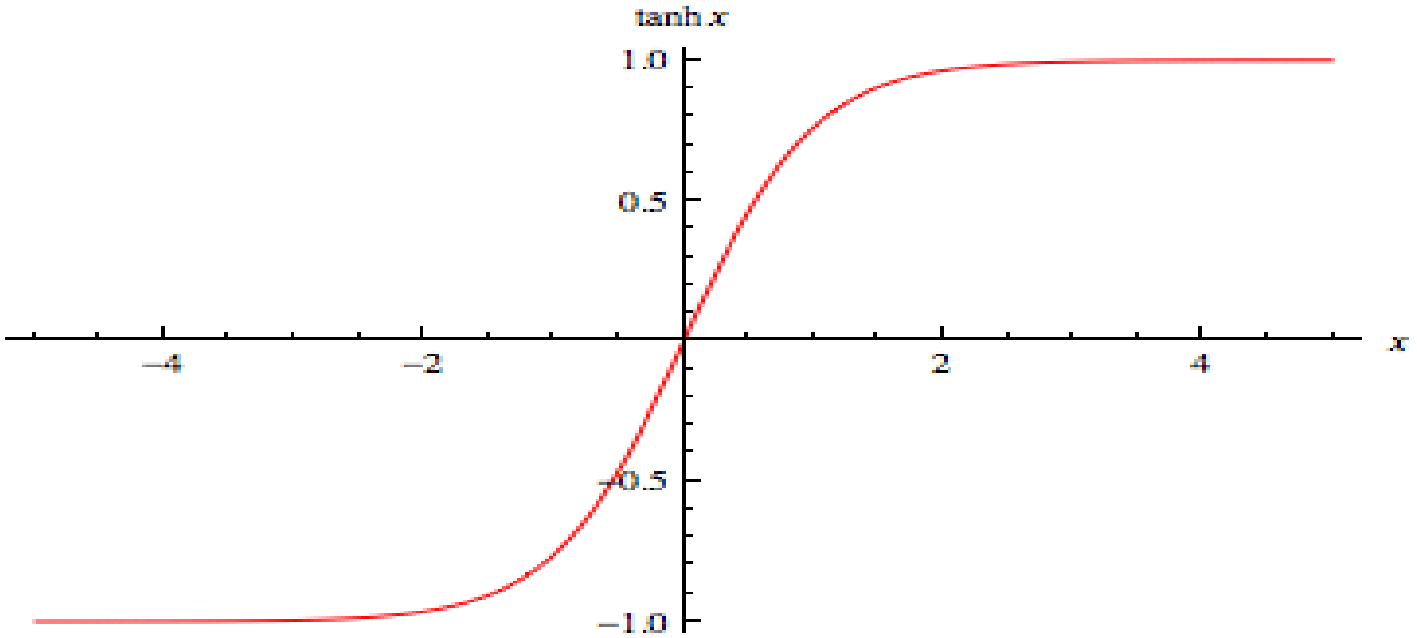
- Forget gate computes a 0-1 value using a logistic sigmoid output function from the input, x_t, and the current hidden state, h_t:
- Multiplicatively combined with cell state, "forgetting" information where the gate outputs something close to 0.



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

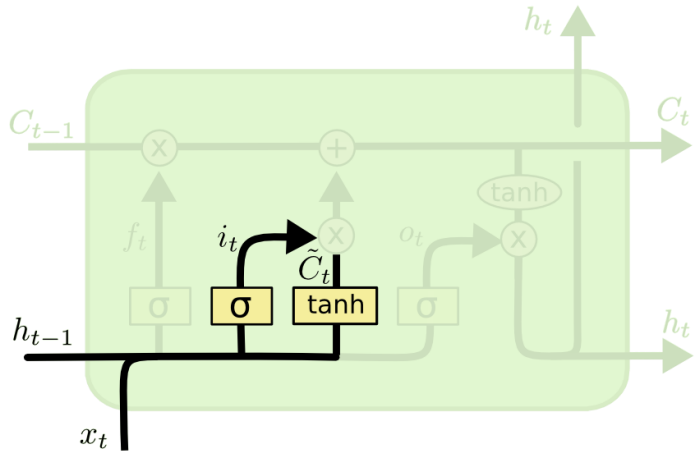
Hyperbolic Tangent Units

- Tanh can be used as an alternative nonlinear function to the sigmoid logistic (0-1) output function.
- Used to produce threshold output between –1 and 1.



Input Gate

- First, determine which entries in the cell state to update by computing 0-1 sigmoid output.
- Then determine what amount to add/subtract from these entries by computing a tanh output (valued –1 to 1) function of the input and hidden state.

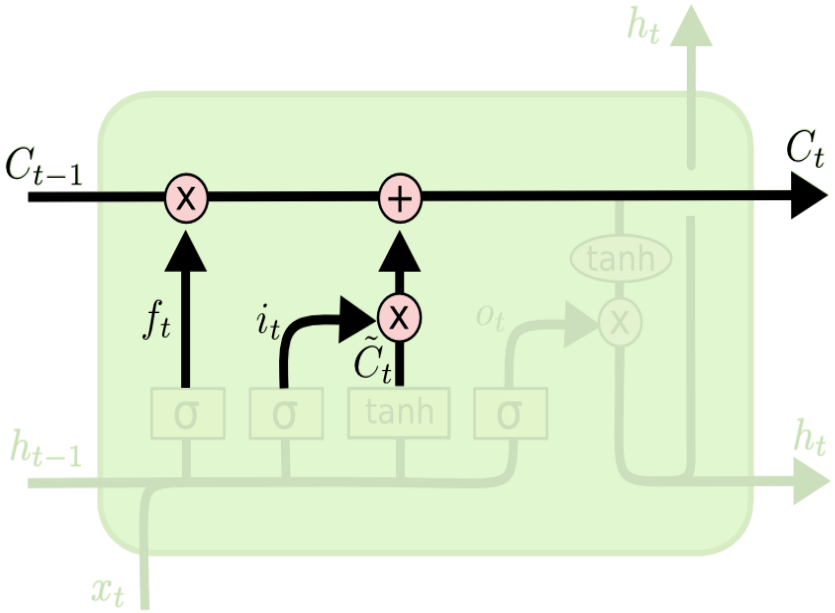


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

Updating the Cell State

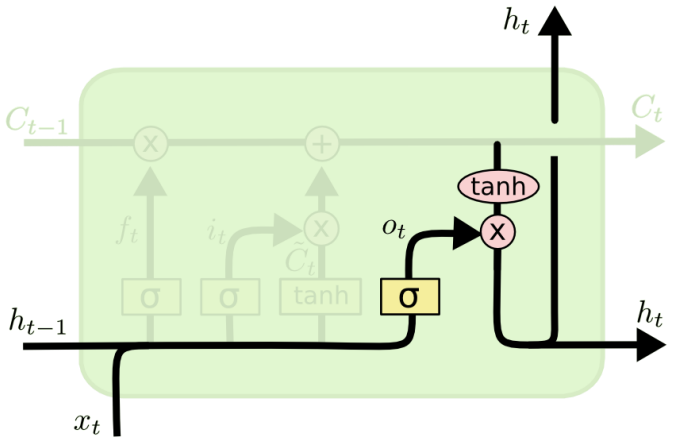
- Cell state is updated by using component-wise vector multiply to "forget" and vector addition to "input" new information.



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

Output Gate

- Hidden state is updated based on a "filtered" version of the cell state, scaled to –1 to 1 using tanh.
- Output gate computes a sigmoid function of the input and current hidden state to determine which elements of the cell state to "output".

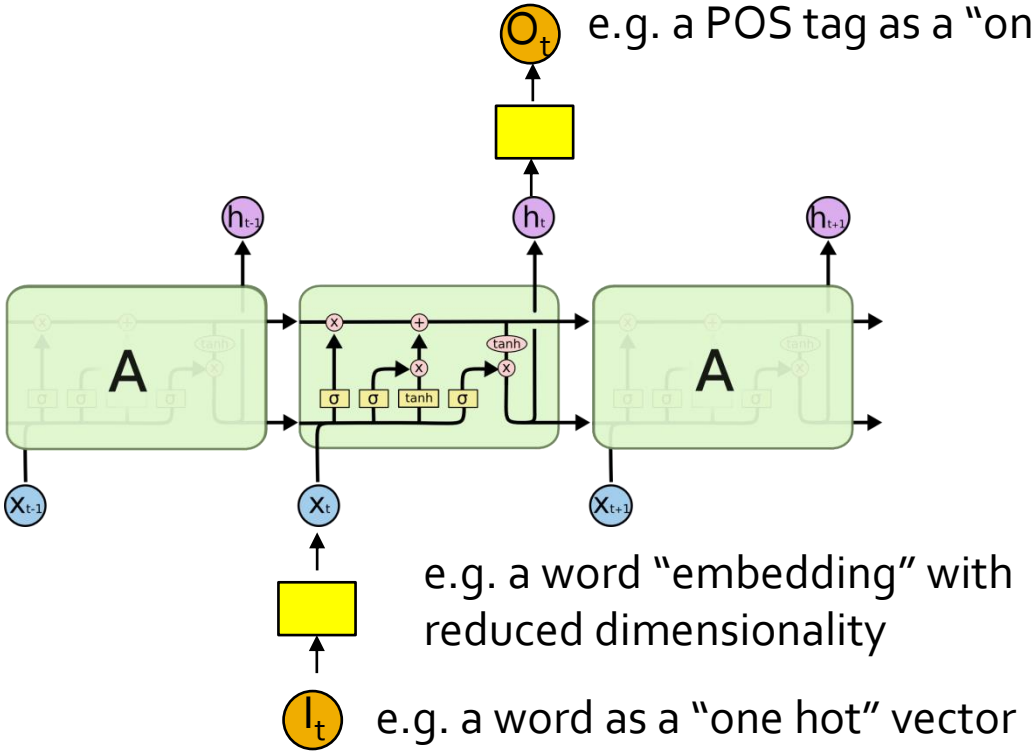


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

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

Continues….

- Single or multilayer networks can compute LSTM inputs from problem inputs and problem outputs from LSTM outputs.



LSTM Training

- Trainable with backprop derivatives such as:
  - Stochastic gradient descent (randomize order of examples in each epoch) with momentum (bias weight changes to continue in same direction as last update).
  - ADAM optimizer (Kingma & Ma, 2015)
- Each cell has many parameters (W_f, W_i, W_C, W_o)
  - Generally requires lots of training data.
  - Requires lots of compute time that exploits GPU clusters.

General Problems Solved with LSTMs

- Sequence labeling
  - Train with supervised output at each time step computed using a single or multilayer network that maps the hidden state (h_t) to an output vector (O_t).
- Language modeling
  - Train to predict next input ($O_t = I_{t+1}$)
- Sequence (e.g. text) classification
  - Train a single or multilayer network that maps the final hidden state (h_n) to an output vector (O).

LSTM Application Architectures

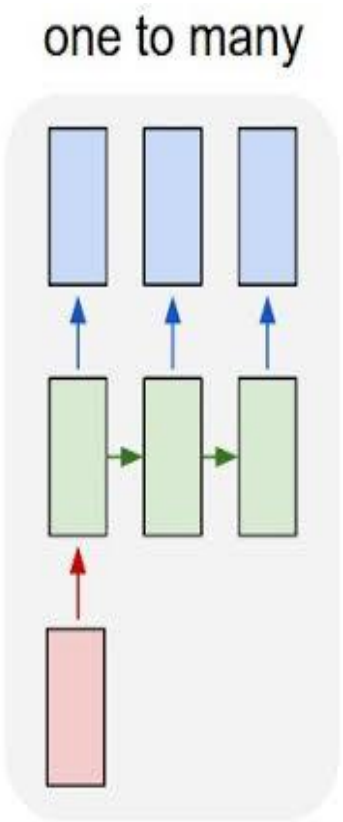
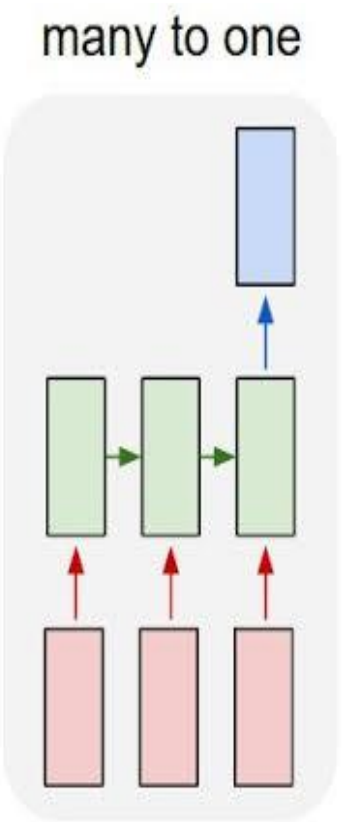
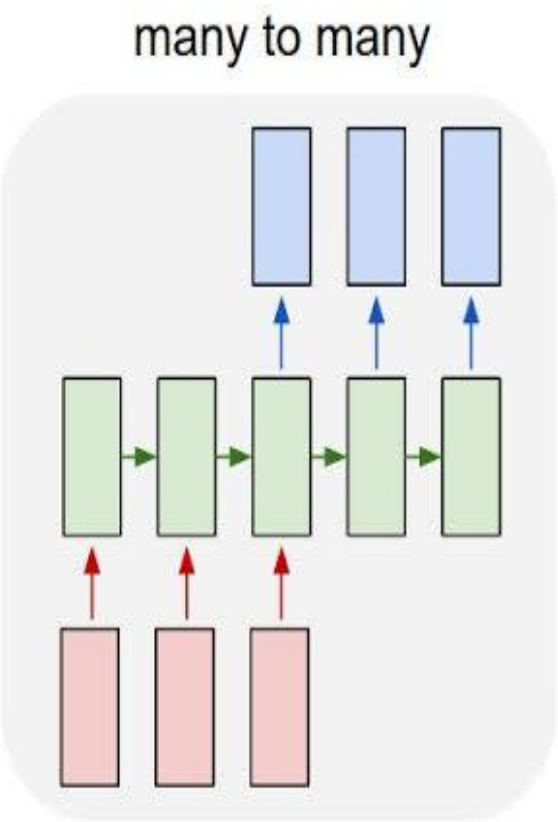


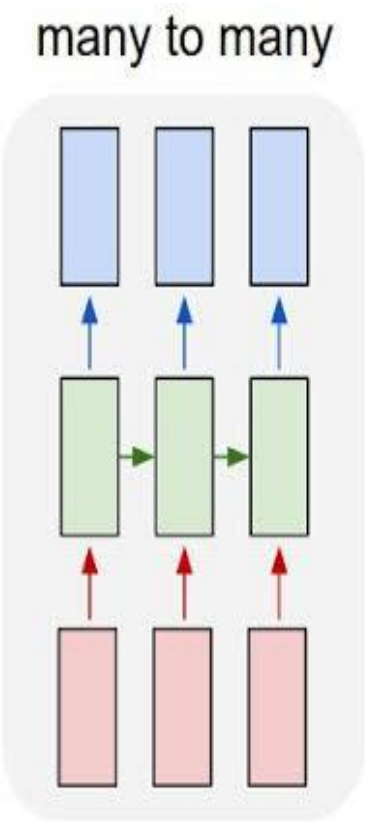
Image Captioning



Video Activity Recog.
Text Classification



Video Captioning
Machine Translation



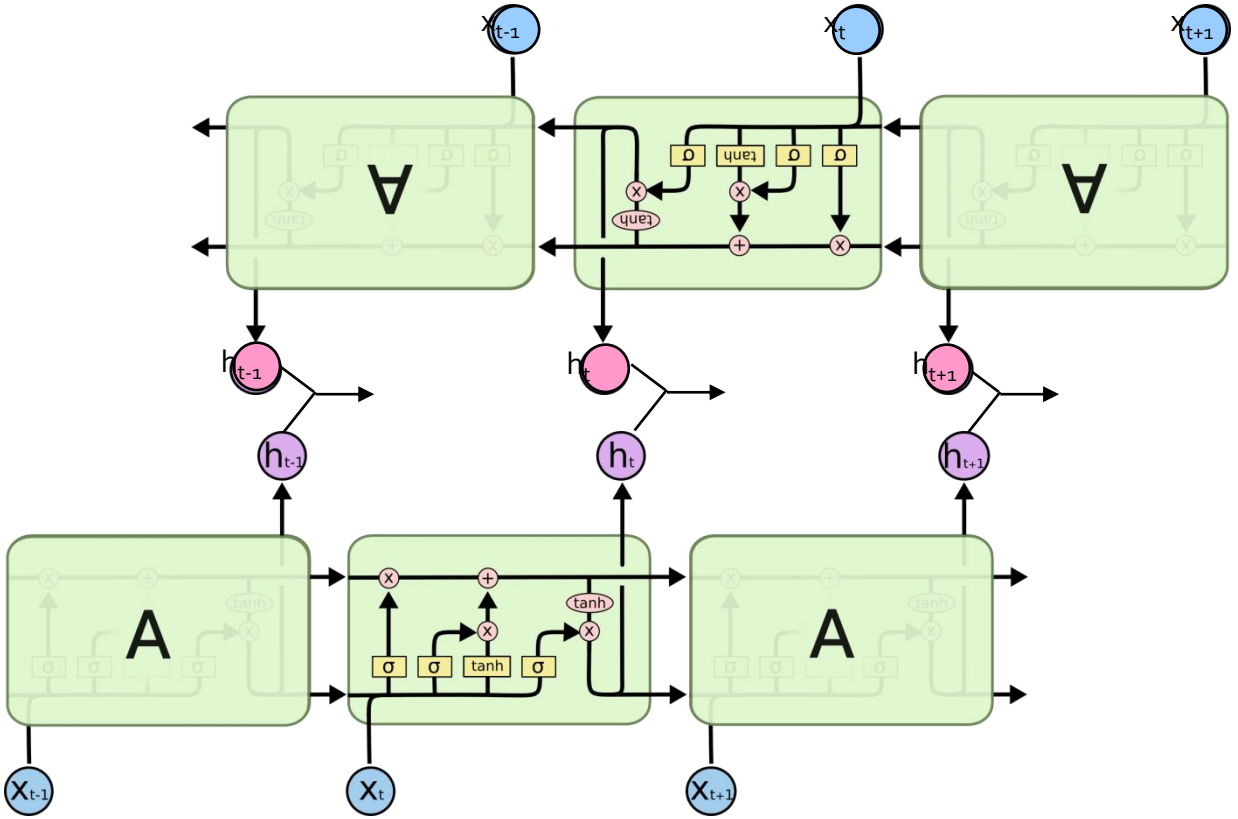
POS Tagging
Language Modeling

Applications of LSTMs

- Speech recognition: Language and acoustic modeling
- Sequence labeling
  - POS Tagging
  - NER
  - Phrase Chunking
- Neural syntactic and semantic parsing
- Image captioning: CNN output vector to sequence
- Sequence to Sequence
  - Machine Translation
  - Video Captioning (input sequence of CNN frame outputs)

Bi-directional LSTM (Bi-LSTM)

- Separate LSTMs process sequence forward and backward and hidden layers at each time step are concatenated to form the cell output.



Attention

- For many applications, it helps to add "attention" to RNNs.
- Allows network to learn to attend to different parts of the input at different time steps, shifting its attention to focus on different aspects during its processing.
- Used in image captioning to focus on different parts of an image when generating different parts of the output sentence.
- In MT, allows focusing attention on different parts of the source sentence when generating different parts of the translation.

Attention for Image Captioning

Figure 2. Attention over time. As the model generates each word, its attention changes to reflect the relevant parts of the image. "soft" (top row) vs "hard" (bottom row) attention. (Note that both models generated the same captions in this example.)

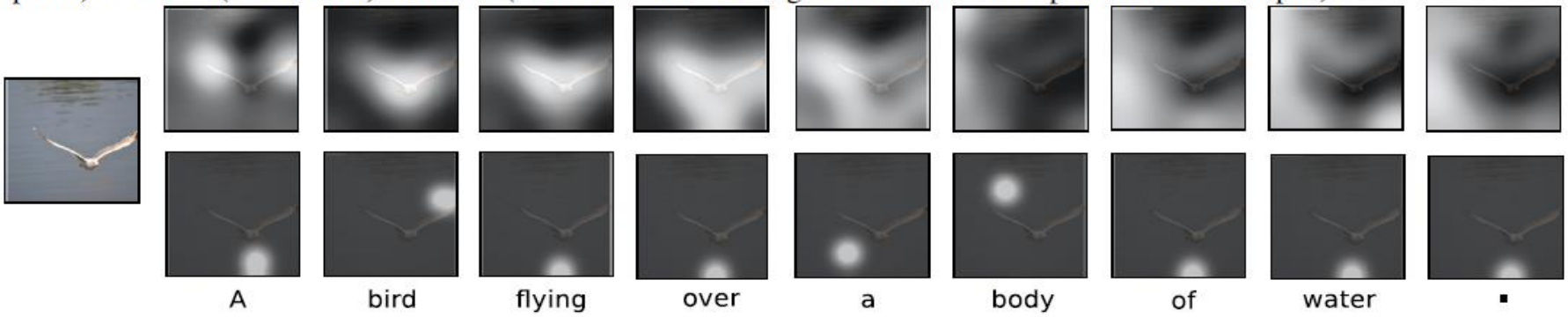


Figure 3. Examples of attending to the correct object (*white* indicates the attended regions, *underlines* indicated the corresponding word)

A woman is throwing a frisbee in a park.

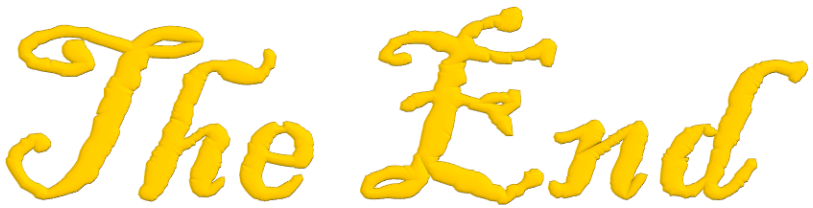
A dog is standing on a hardwood floor.

A stop sign is on a road with a mountain in the background.

A little girl sitting on a bed with a teddy bear.

A group of people sitting on a boat in the water.

A giraffe standing in a forest with trees in the background.


The End