6CS012 – Artificial Intelligence and Machine Learning.

Lecture – 09

Introduction to Natural Language Processing.

Sequence to Sequence Learning.

Siman Giri {Module Leader – 6CS012}

What we Built?

Natural Language Understanding.

TXT → **Text – Preprocessing.** + **Text – Embedding.** + **ML Models** → Class – 1 OR Class – 0

Step	Description	Output Example
1	Original Text	"I Love Machine Learning", "Machine Learning … , … "
2	Tokenization (word → Tokens→ index)	Word Index – {"i":1, "love":2, "machine":3, "learning":4} **Tokenized Sequence**: [[1, 2, 3, 4], [… …], […]]
3	Embedding (index → vector)	**Embedding Vectors** → [[0.01, −0.05, …, 0.08], [, …,][…]]

Average Embedding Vector: (Word Averaging):
"Embedding vectors are averaged to create a fixed-size input for machine learning models, allowing sequences of varying lengths to be handled uniformly"

- What is word averaging, and
    - what are its semantic implications and
        - associated challenges in the context of text representation?

Challenge – 1 – Text Data.

- **Texts Data are Inherently Sequential:**
  - The **meaning of a sentence** depends not only on **the individual words** used but also on **the specific order** in which **they appear**.
  - Language has **grammar, structure, and context**, all of which rely **on this sequence**.
  - **For example:**
    - The sentences: **"The dog chased the cat."** and **"The cat chased the dog."** contain the **same words** but the **meaning** is **completely different** due to the **change in word order**.
    - words like **"not"** or **"but"** can **flip or contrast** the sentiment of a sentence depending on where they **appear**.
  - In natural language or Text data , **context accumulates word by word i.e.**
    - **what we read or hear next often depends on what came before.**
  - This makes **text a temporal or ordered data type** much like **time series or audio signals**.
- Thus, preserving and modelling the sequential nature of text is essential for understanding meaning, emotion, intent and other linguistic feature.
- **How does ML models handle this? Do they even consider this?**

How do ML handle this? Word Averaging.

- Idea of Word Averaging or Embedded Vector Averaging:
  - **Word averaging** is a simple way to represent a full sentence (**sentence level embedding**) or document as a single vector by:
    - Looking up the **embedding vector** (e.g., from Word2Vec) for **each word** in the sentence.
    - Taking the **average (mean)** of all those vectors.
    - Using that **average vector** as the **feature representation for the whole sentence**.
- Example: For sentence – **"I Love NLP."**
  - Let's assume following is **3-D word2vec representation** (**just for an example**):

Word	Embedding
i	[0.1, 0.0, 0.3]
love	[0.8, 0.6, 0.2]
NLP	[0.4, 0.7, 0.9]

- $\textbf{Average Vector} = \left(\left[\frac{0.1+0.8+0.4}{3} + \frac{0.0+0.6+0.7}{3} + \frac{0.3+0.2+0.9}{3}\right]\right)$
- $\textbf{Sentence Vector Representations} = [0.433, 0.433, 0.467]$
- **This averaged vector is then fed into your classifier (e.g. Logistic Regression).**

Pros and Limitations of word averaging …

Pros

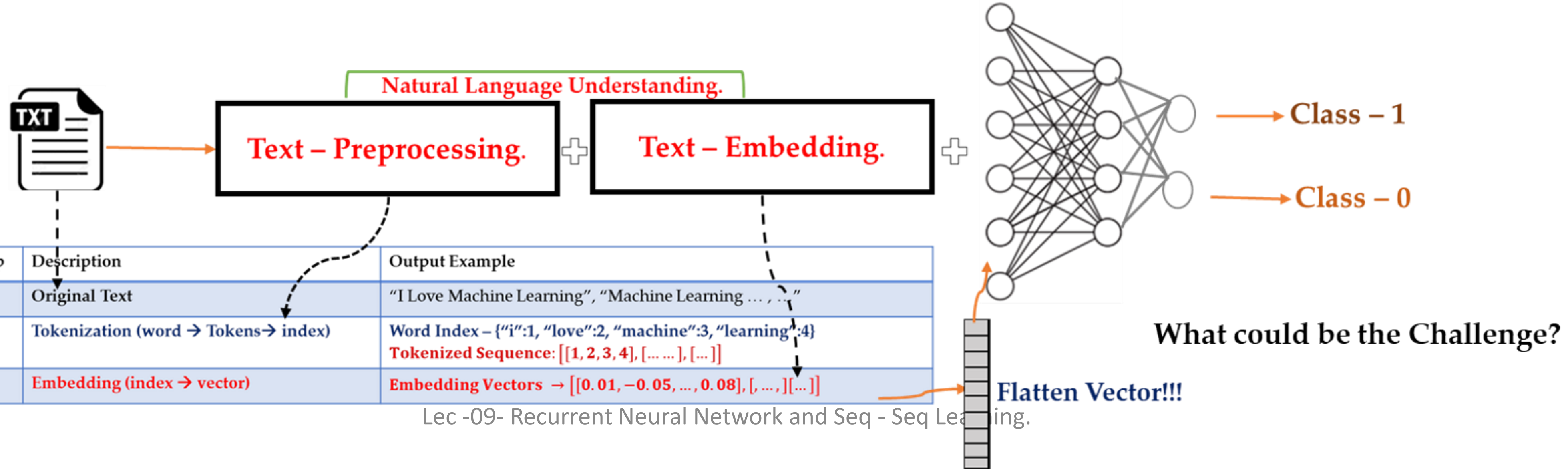
- **Simple and fast** to implement.
- Works reasonably well with **small datasets.**
- Reduces **variable-length text to a fixed-size vector**.

Limitations

- **Losses word order:**
  - "*I love NLP*" and "*NLP loves I*" produce the same average:
    - **no difference in meaning.**
- **Ignores important words:**
  - Every word contributes equally no attention to negation to sentiment heavy words
- **No context sensitivity:**
  - Words are used in isolation, the meaning of "bank" in "river bank" vs. "money bank" is the same.

Alternate to word averaging …

- We can conclude that:
  - Preserving and modeling the sequential nature of text is essential for understanding meaning, emotion, intent, and other linguistic features
  - Something that traditional machine learning models struggle with unless we manually craft features to capture such order.
- Way forward:
  - Can we design a Neural Networks such that it can handle sequence property, making them highly effective for text-based tasks like sentiment analysis, translation and question answering.



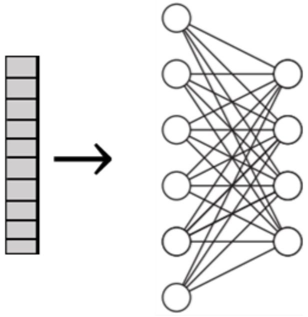
Step	Description	Output Example
1	Original Text	"I Love Machine Learning", "Machine Learning … , … "
2	Tokenization (word → Tokens→ index)	Word Index – {"i":1, "love":2, "machine":3, "learning":4} Tokenized Sequence: [[1, 2, 3, 4], [… …], […]]
3	Embedding (index → vector)	Embedding Vectors → [[0.01, −0.05, …, 0.08], [, …,][…]]

Limitations of FCN for Text.

- **No word order awareness:**
  - **No memory or context –** Can not retain previous words ,
    - Treats input as a flat vector.
  - Misses long – term dependencies (e.g. subject – verb agreement)
  - Thus, can not handle the sequential nature of text data.
  - Fails to distinguish: "cat chased mouse" vs. "mouse chased cat".

- **Fixed Input Size – (Manual Features Extraction) :**
  - Depends on handcrafted feature inputs like word averaging.
    - Loses structure and syntax
  - Requires padding or truncations to manage variable input size.
  - Can lose or distort important information.
- **Ignores word position:**
  - Process all words equally
  - May miss nuances like "not good" vs "good not".

$[\text{ cat, chased, mouse}] - \textbf{vector representation} \rightarrow [[\mathbf{v}_{\text{cat}}], [\mathbf{v}_{\text{chased}}], [\mathbf{v}_{\text{mouse}}]] \rightarrow$ **Word Average** $\rightarrow$



Overcoming FCN limitations

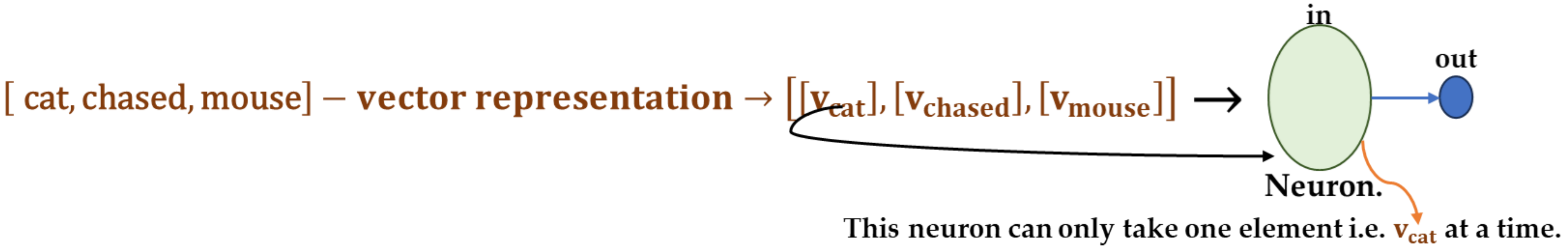
- To address these challenges, we need to redesign the neural network architecture to:
  - Handle variable length input sequences
  - Preserve the order of words in text
  - Eliminate the need for input averaging
  - Capture contextual and sequential dependencies
- This leads to sequence models like:
  - RNNs, LSTMs, GRUs and modern Transformer.

1. Neural Network for Sequential Input.

{ "The Model for Text Data." }

1.1 Neural Network for Sequential Input.

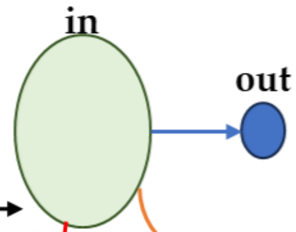
- Sequential Input means input must be in order i.e.
  - ["cat", "chased", "mouse"] ≠ ["mouse", "chased", "cat"]
- What changes can we make in this architecture so that it is able to use previous information?



1.1 Neural Network for Sequential Input.

- Sequential Input means input must be in order i.e.
  - ["cat", "chased", "mouse"] ≠ ["mouse", "chased", "cat"]
- What changes can we make in this architecture so that it is able to use previous information?

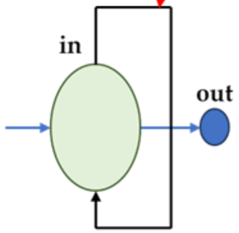
[cat, chased, mouse] – **vector representation** → $[[v_{cat}], [v_{chased}], [v_{mouse}]]$ →



Neuron.

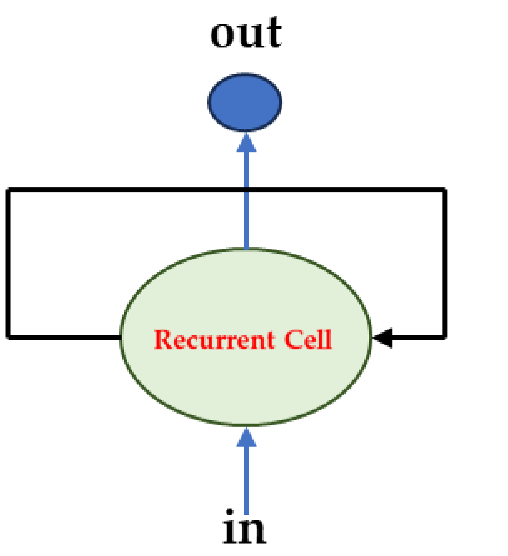
This neuron can only take one element i.e. v_{cat} at a time.

What happens if we add a loop, that can pass prior information,

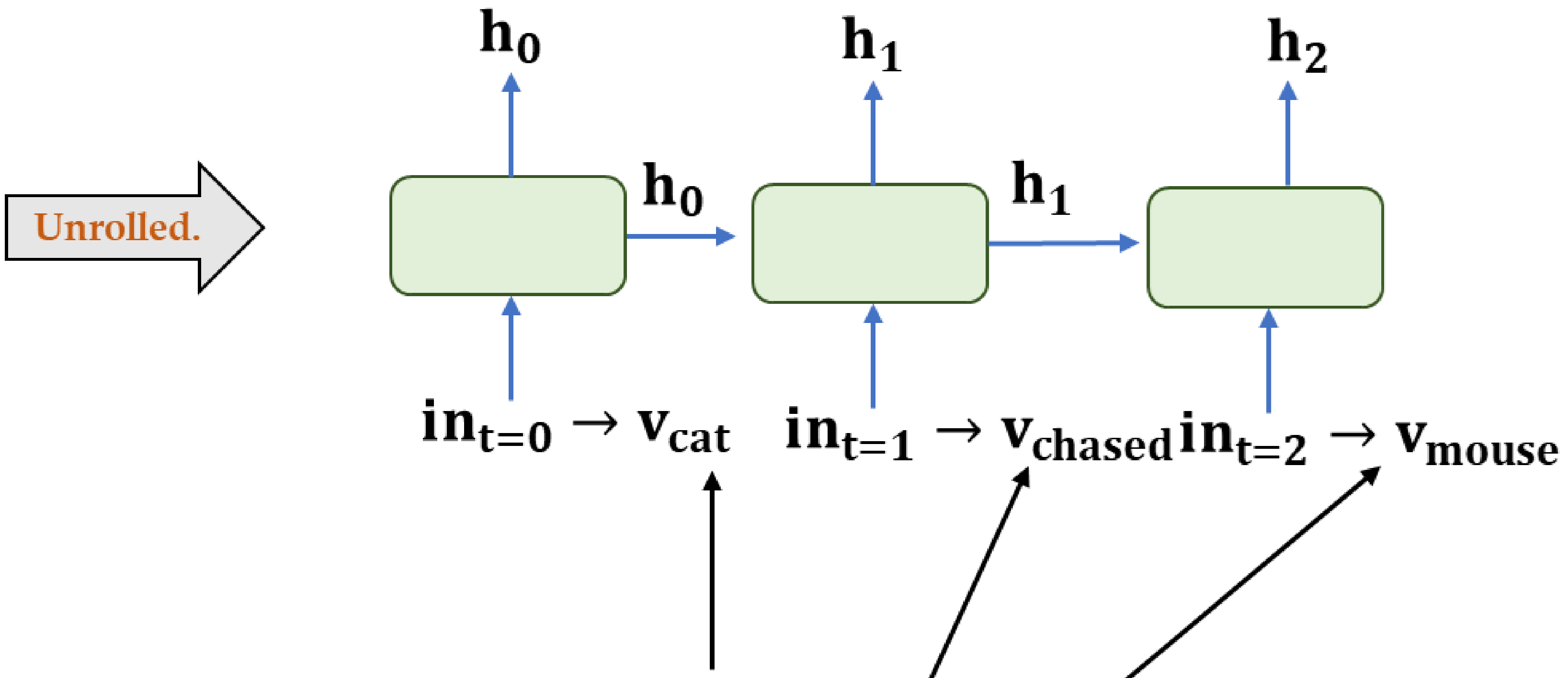


Looks Complicated … Let's Simplify.

1.2 Sequential Input: Neurons with Recurrence.



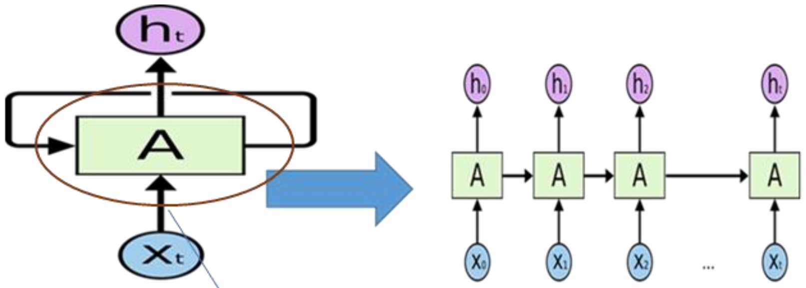
Neurons with Recurrence.



[cat, chased, mouse] – vector representation → $[[v_{cat}], [v_{chased}], [v_{mouse}]]$

1.3 Recurrent Neural Network: Introduction.

- A **Recurrent Neural Network (RNN)** is a class of artificial neural networks designed to process sequential data by maintaining a hidden state that captures information about previous elements in the sequence.
- At each time step , an RNN takes an input vector (word embedding) and combines it with the previous hidden state to produce a new hidden state:
  - $\mathbf{h_t = f_W(h_{t-1}, x_t)}$
    - $\mathbf{x_t}$ → input at time step t.
    - $\mathbf{h_{t-1}}$ → previous hidden state old state.
    - $\mathbf{w}$ → learned weights or parameters.
    - $\mathbf{f_W}$ → **some** mapping function with parameters $\mathbf{w}$: $\mathbf{x_t \rightarrow h_t}$.



Recurrent Neural Network with loops

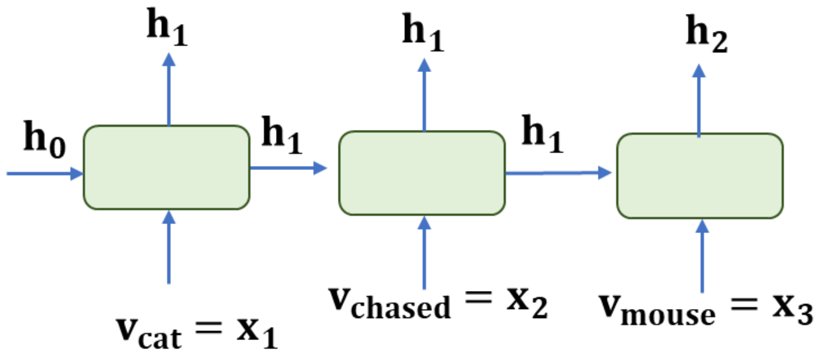
Unrolled recurrent neural network

Image from Christopher Olah's blog.

known as Recurrence Cell or Hidden Cell, where all the action happens.

1.4 A "vanilla" RNN.

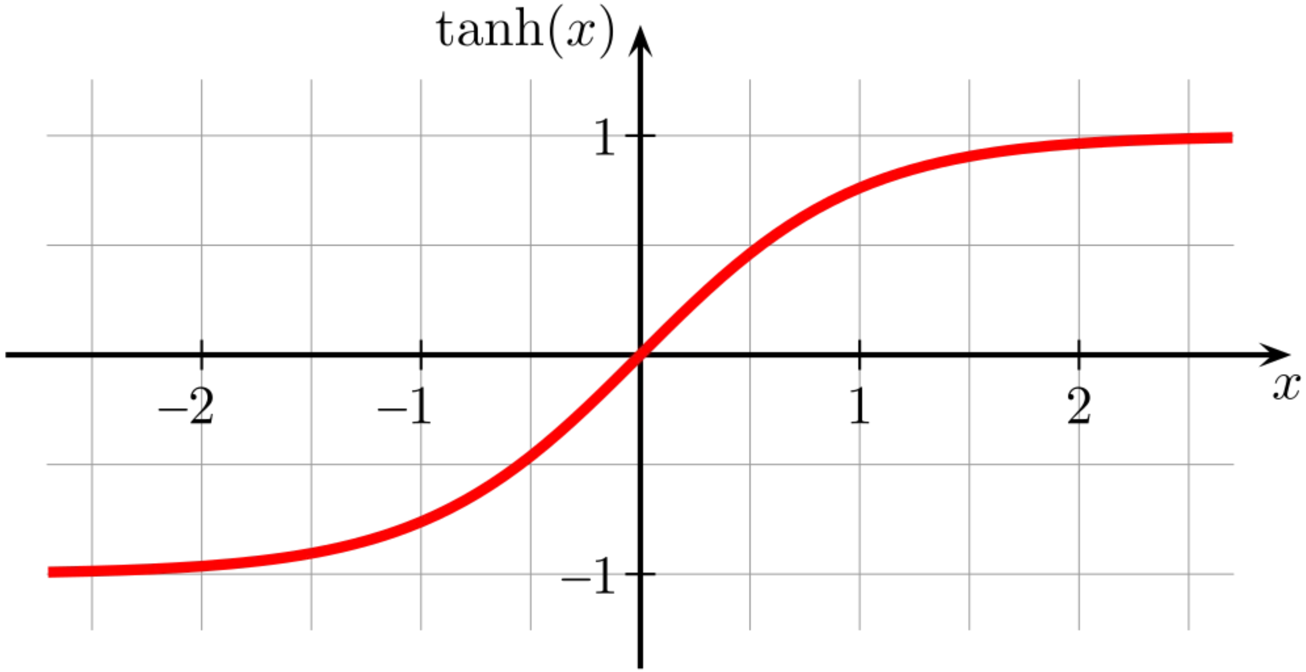
- aka "**simple**" RNN or "**Elman**" RNN after Jefferey Elman.
  - A vanilla RNN, updates its hidden state by combining the current words embedding and the previous hidden state using tanh as an activation function.
- For our Example – **["cat", "chased","mouse"];** Let's assume:
  - **Input:** $\mathbf{x_1 \rightarrow v_{cat}; x_2 \rightarrow v_{chased}, ; x_3 \rightarrow v_{mouse}}$ **{Here: v_ → vector represetation of input tokens}**
  - *Each word is fed one at a time into the* **RNN cell.**
    - ***At time step – 1:***
      - $\mathbf{h_1 = \tanh(w_x \cdot x_1 + w_h \cdot h_0 + b)}$ **(input → cat)**
    - ***At time step – 2:***
      - $\mathbf{h_2 = \tanh(w_x \cdot x_2 + w_h \cdot h_1 + b)}$ **(input → chased)**
    - ***At time step – 3:***
      - $\mathbf{h_3 = \tanh(w_x \cdot x_3 + w_h \cdot h_2 + b)}$ **(input → mouse)**



- **Here:**
  - $\mathbf{h_0}$ **→ usually initialized to zeros.**
  - $\mathbf{h_t}$ **→ each $\mathbf{h_t}$ carries a summary of all previous words upto time t.**
  - **So , by the end , $\mathbf{h_3}$ knows something about "cat chased mouse" as a whole - in order.**

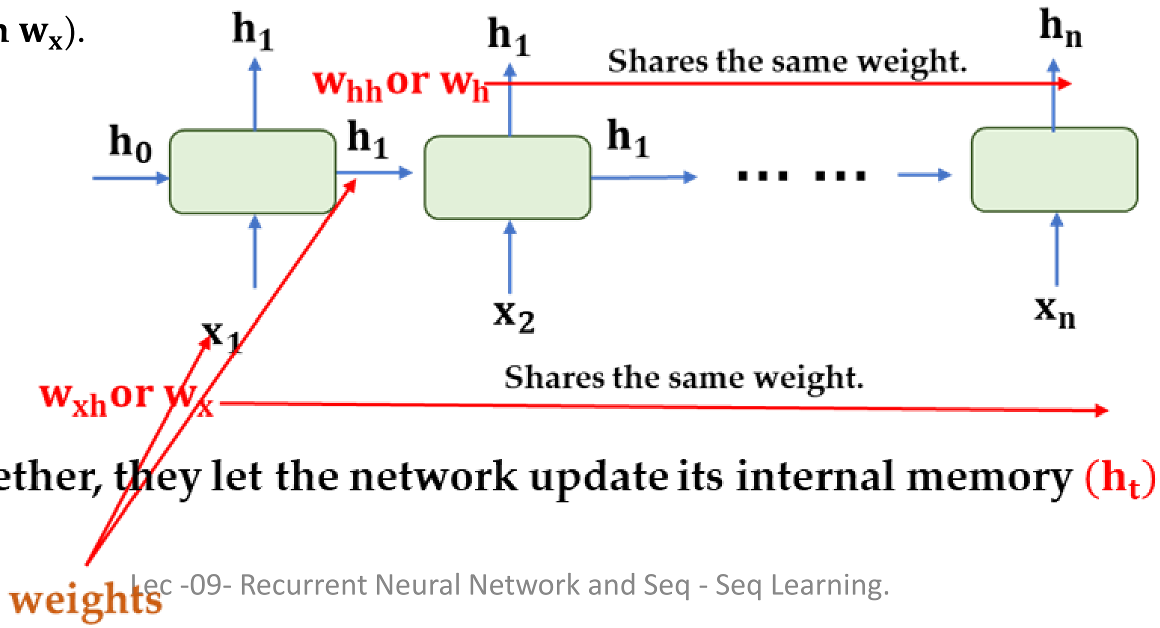
Remember tanh!

- Mathematical Representations:
  - $\mathbf{f(x) = tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}}$
- **Output range: (-1, 1)**



1.4.1 Parameters (w) in simple RNN.

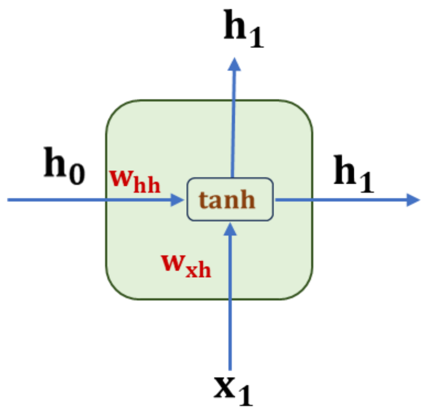
- **w_x aka w_{xh}:**
  - Purpose: Transforms the current *input* x_t
    - i.e. project weight to hidden state (connects the input layer to hidden layer)
  - **Shape: (input_dim, hidden_dim)**
    - input_dim = size of input vectors of tokens i.e. embedding dimensions.
    - hidden_dim = number of neurons in RNN.
  - What it sees now (**through w_x**).
- **w_h aka w_{hh}:**
  - Purpose: Transforms the previous hidden *state* h_{t-1}
    - i.e. project weight to another hidden state h_t (connects the hidden layer to itself over time)
  - **Shape : (hidden_dim, hidden_dim)**
  - What it sees now (**through w_h**).



Together, they let the network update its internal memory (h_t) at each time step.

1.4.2 w in Practice.

- A vanilla RNN cell has following computations:

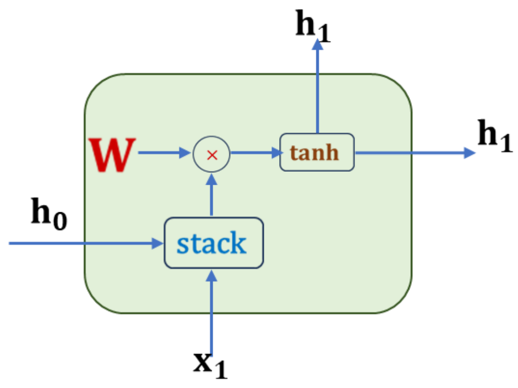


At every time step:

$$h_t = \tanh(w_{hh} \cdot h_{t-1} + w_{xh} \cdot x_t) \; - [1]$$

This involves **two separate matrix multiplications** which makes computation long and repeated,

- Efficient Computation via vectorization:



To simplify and speed up computation, we concatenate the previous hidden state and current input:

$$z_t = \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}_{hidden\_dim+input\_dim}$$

Then, we apply a single weight matrix $W_{hidden\_dim\times(hidden\_dim+input\_dim)}$:

$$h_t = \tanh\big((w_{hh}, w_{hx})(z_t)\big) == \tanh\big(W(z_t)\big) \; - [2]$$

Here:

$$W \in \mathbb{R}^{hidden\_dim\times(hidden\_dim+input\_dim)}$$

This is equivalent to eq [1], but in practice makes eq [2] efficient and requires only one matrix multiplication.

2. Building RNN for Application.

{Text Classification or Sentiment Analysis.}

2.1 RNN Architecture for Text Classification.

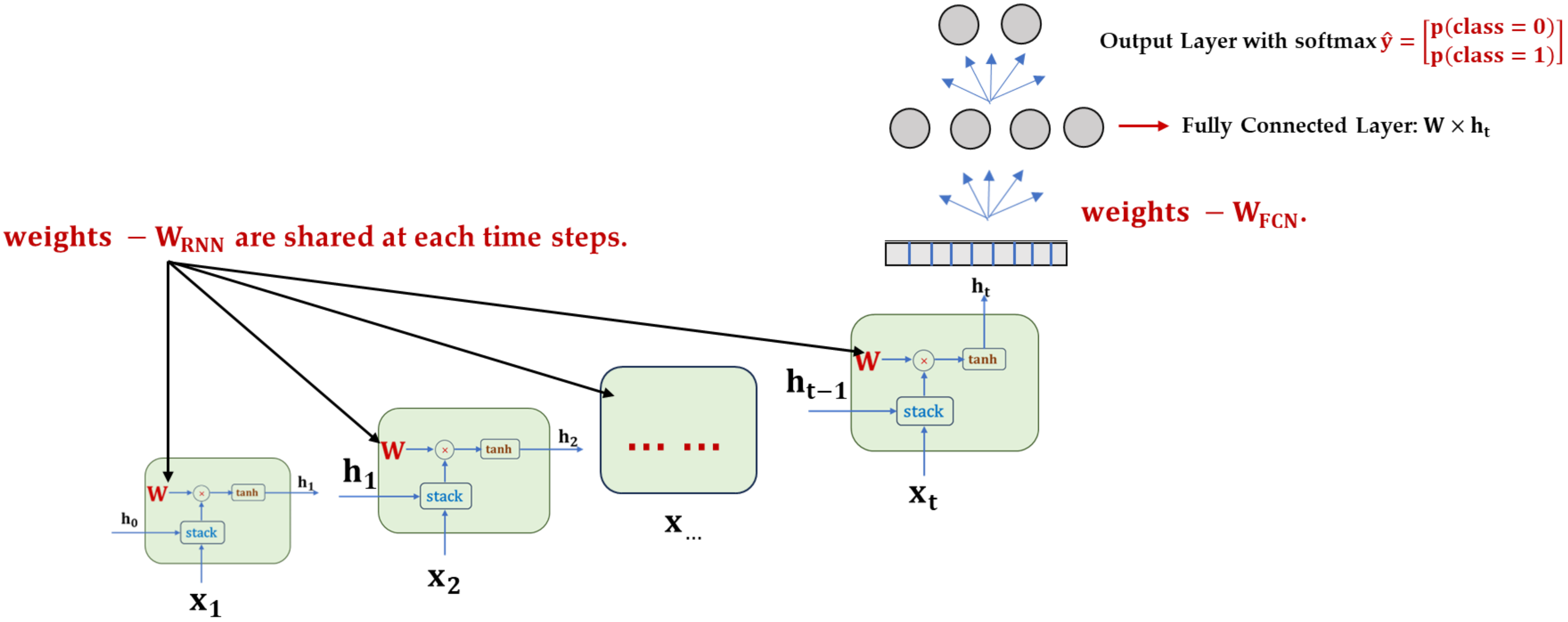
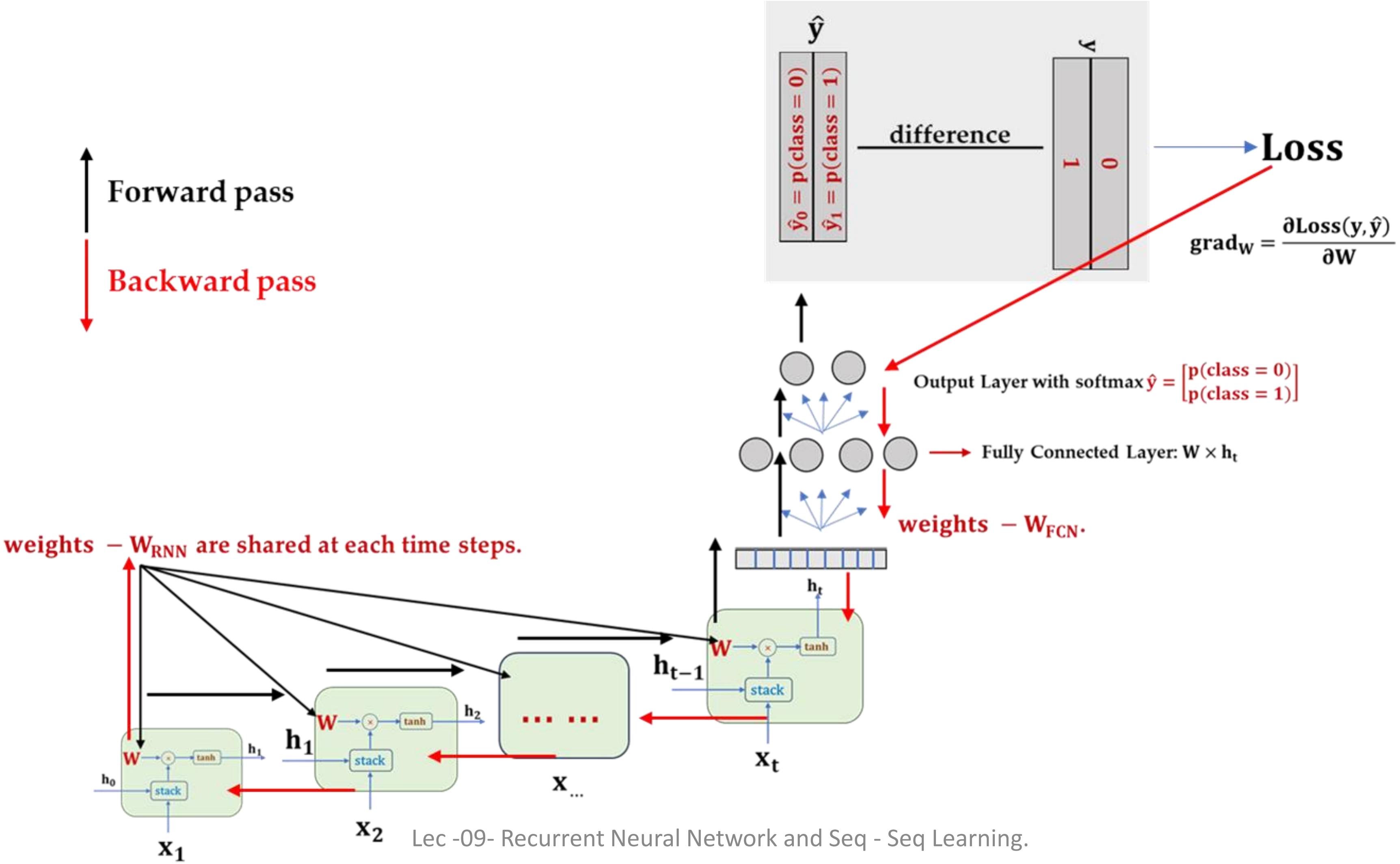


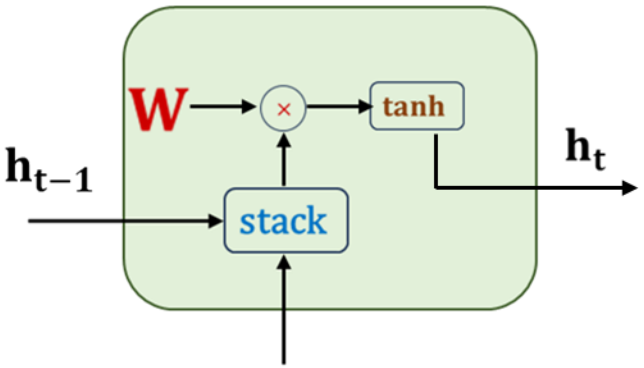
Fig: This particular architecture is called "Many to One" and mostly use for the task of Text Classification or Sentiment Analysis.

2.2 Training an RNN.



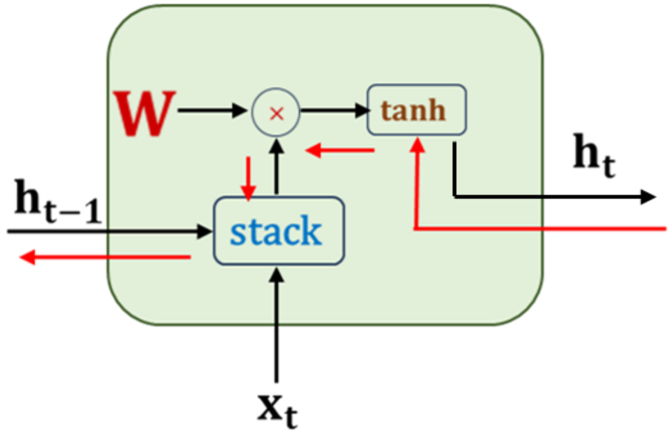
2.3 Gradient Computation at RNN Cell.

Forward Computation.



$$h_t = \tanh(w_{hh} \cdot h_{t-1} + w_{xh} \cdot x_t) = \tanh\left(W \cdot \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}\right)$$

Backward Propagation

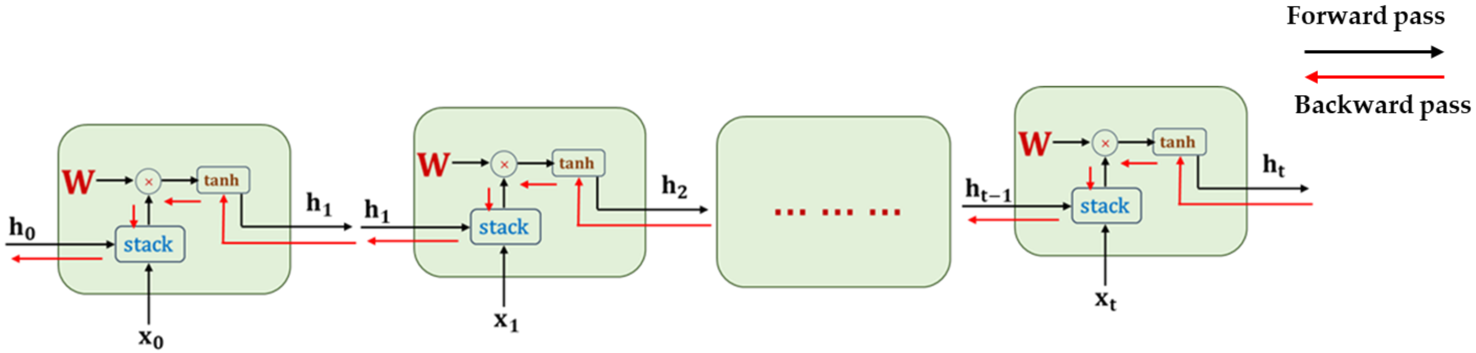


$$\frac{\partial C_t}{\partial h_t} = \left(\frac{\partial C}{\partial y}\right)\left(\frac{\partial y}{\partial h_t}\right)\left(\frac{\partial h_t}{\partial h_{t-1}}\right)\cdots\left(\frac{\partial h_2}{\partial h_1}\right)$$

Cautions: Backpropagation Starts at Output Layer and Follows back through FCN, which we already have discussed.

2.4 Challenges of Training RNN.

- Standard Gradient Flow at RNN:

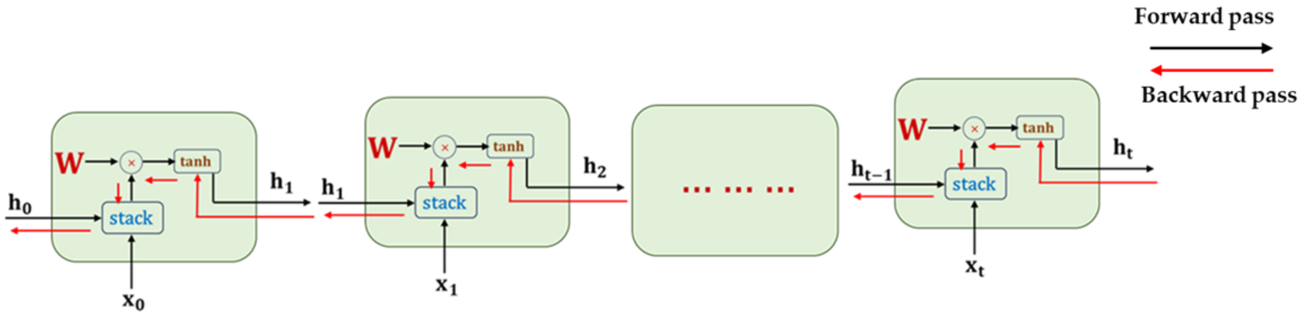


- Computing the gradient **w** w.r.to $\mathbf{h_t}$ involves
  - repeated gradient computations across time steps
  - and repeated tanh.
- This may lead to problem of **Exploding and Vanishing gradient**:

2.4.1 Challenges of Training RNN.

- **Exploding Gradient:**
  - **Challenges:**
    - Many values are more than 1
    - multiplied repeatedly in forward and backward computations –
    - values will grow exponentially:
  - **Effect:**
    - Model becomes unstable, loss starts to diverge may tend to over or underfitting.
  - **Solution:**
    - Gradient Clipping i.e. clipped big gradients to some threshold e.g. [-5, 5].

- **Vanishing Gradient:**
  - **Challenges:**
    - Many values are less than 1
    - are repeatedly multiplied in forward and back propagation
    - values many shrink to zero.
  - **Solution:**
    - Use ReLU,
    - or apply initialization techniques (Xavier or He) for better weight scaling
    - **or Can we design better neuron architecture with memory.**



2.4.2 Challenge of Training: Long –Term Dependencies.

- With the longer sequences, gradient computation also increases and might cause a vanishing gradient.
- It may not be able to hold information for longer sentences.

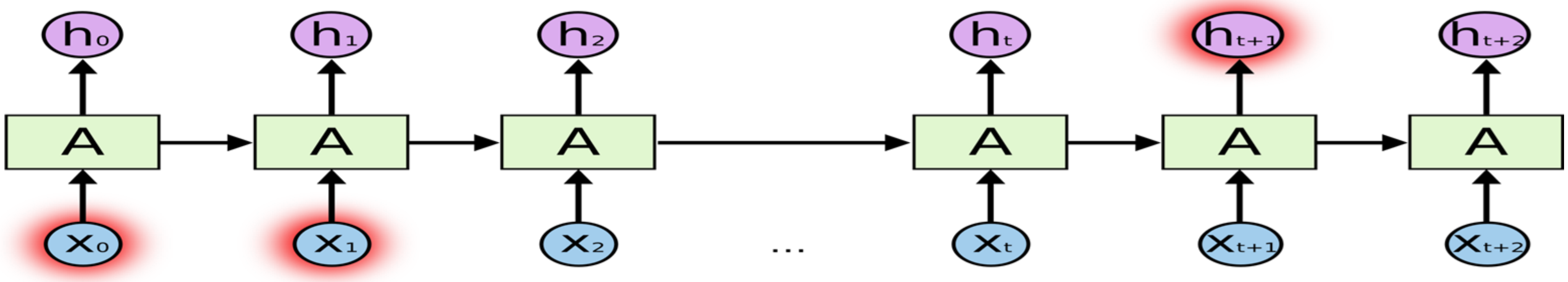
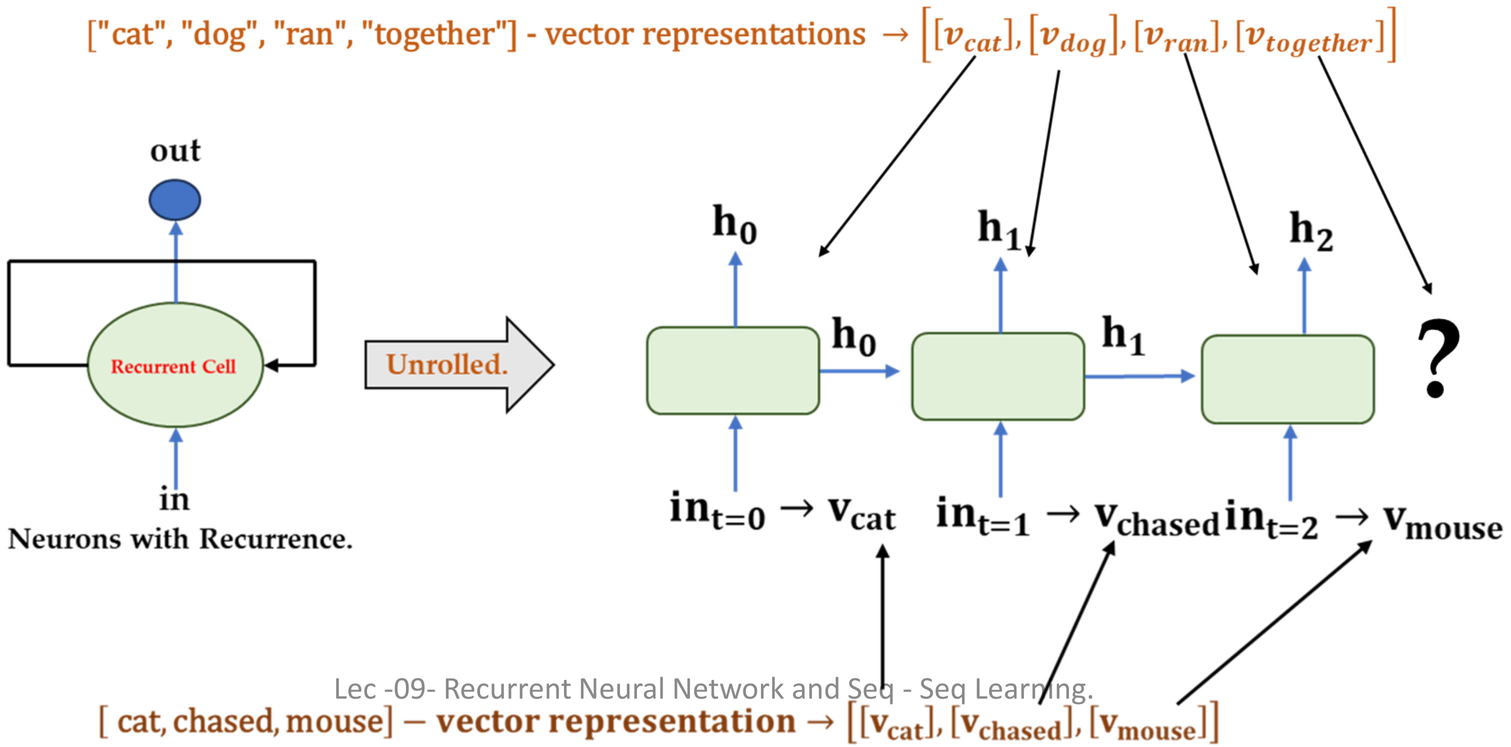


Image from Christopher Olah's blog.

2.4.3 Challenge of Training: Handling Variable Length Sequence.

- Real world datasets contain sentences of varying lengths. For example:
  - **sentence – 1** – ["cat", "chased", "mouse"]
  - **sentence – 2** – ["cat", "dog", "ran", "together"]
- Problem:
  - If we design our RNN architecture to unroll for only 3-time steps, it will fail to process sentence 2 which has 4 tokens.



2.4.3.1 Handling Variable Length.

- Solution – **Padding to Max Sequence Length**.
  - To ensure uniform input lengths:
  - Determine **maximum sentence length**, say **n** across the dataset.
  - Pad all shorter sentences with a special token **(e.g. <“pad”>)** to match **n**.
    - **<“pad”> will have a special vector representations.**

Sentence	Original	After Padding (n = 4)
Sentence 1	[“cat”, “chased”, “mouse”]	[“<pad>”, “cat”, “chased”, “mouse”]
Sentence 2	[“cat”, “dog”, “ran”, “together”]	[“cat”, “dog”, “ran”, “together”]

2.5 Final Tips on Training RNN.

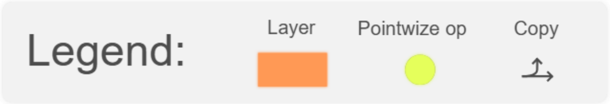
- Following Design Criteria most be thought of while building and training RNNs:
  - RNNs must meet following criteria to model sequential data:
    - **Handle variable-length sequences.**
    - **Track long-term dependencies.**
    - **Maintain information about order.**
    - **Share parameters across the sequences.**

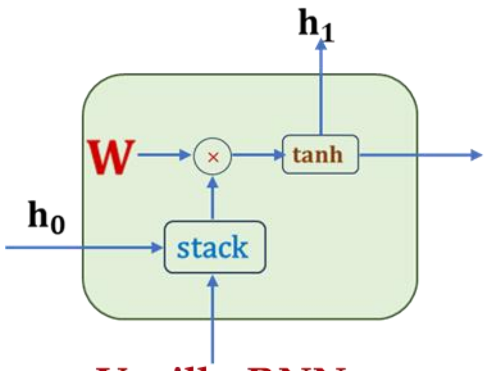
3. Adding Memory to "vanilla" recurrent neuron.

{LSTM – Long Short-Term Memory}

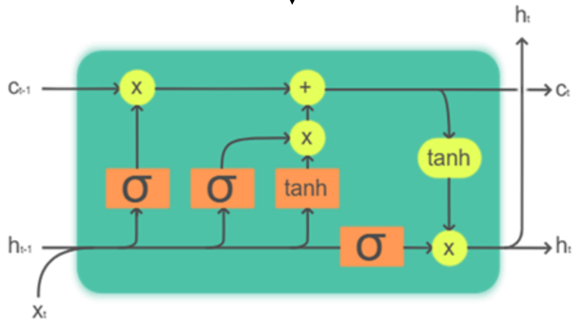
3.1 “vanilla” to “LSTM” neuron.

- So far, we have seen only a simple recurrence formula for the Vanilla RNN.
  - In practice, we actually will rarely ever use Vanilla RNN formula.
- Instead, we will use what we call a
  - **Long-Short Term Memory (LSTM) RNN:**
    - This help us to overcome the problem of
    - Vanishing Gradient and Problem of Long Short-Term Dependencies.
      - Idea: Insert a memory in Network….
        - How:
          - Use **gates** to **selectively add** or **remove information** within each recurrent unit.
          - Gates are created using





Vanilla RNN

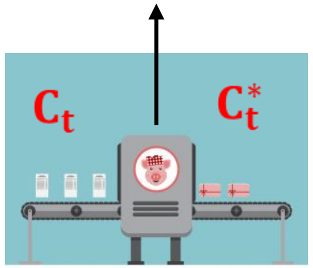


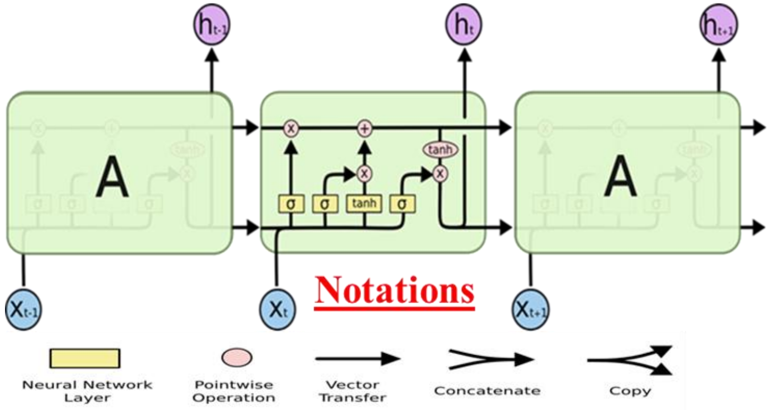
LSTM Neuron

3.2 Core Idea Behind LSTMs

- The **cell state C_t** is the core component that allows LSTMs (**Long Short-Term Memory networks**) to **retain long-term dependencies** over **sequences.**
  - Think of the cell state as a **conveyor belt** running through all **LSTM cells** in the sequence.
- It provides a path for information to flow with **minimal modification**, thus avoiding the vanishing gradient problem seen in traditional RNNs.
  - Information in the cell state is **modified slightly** via **multiplicative gates**.
  - These gates decide what information to **keep**, **update**, or **forget** at each time step.

Slight modification with gates.





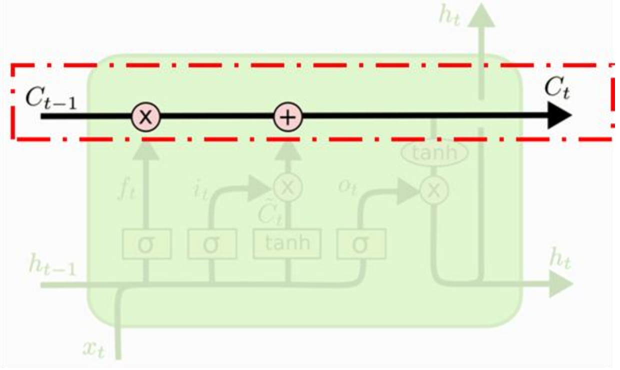
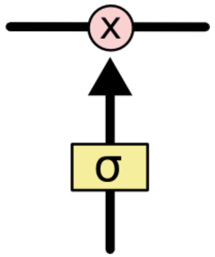


Image from Christopher Olah's blog.

3.3 Gates in LSTM: Controlling Information Flow.

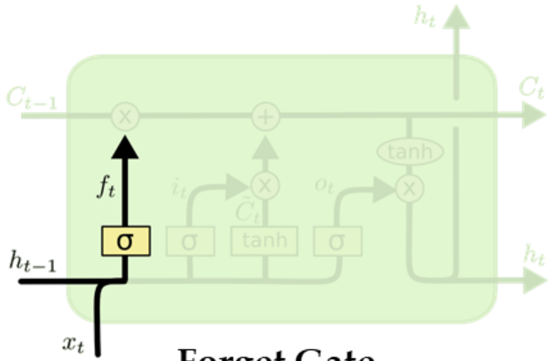
- Gates are **special mechanisms** in LSTMs that **control the flow of information** through the **cell state**.
- They act like **valves**, deciding which information to **keep**, **update**, or **discard**.
- Each **Gate** is made up of:



  - A **sigmoid activation layer**: outputs **values between 0 and 1 describes** how much of each component should be let through.
    - **A value of 0 means let nothing through**
    - **A value of 1 means let everything through**
  - A **pointwise multiplication**: scales information based on gate's sigmoid output.
- **Together**, this mechanism allows LSTMs to **learn what to forget, remember, and output** at each time step.
- An LSTM has three type of Gates: **an input gate; an output gate; and a forget gate;**

3.3.1 Gate – 1 – forget Gate.

- **Purpose:**
  - Decides what information to discard from the previous cell state $\{\mathbf{C_{t-1}}\}$.
- **Input:**
  - Previous hidden state h_{t-1} and current input x_t.
- **Operation:**
  - $\mathbf{f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)}$
- The output $\mathbf{f_t}$ (values between 0 and 1) determines how much cell **state $\mathbf{C_{t-1}}$** should be retained at current cell.
- **Effect:** cell state is updated as:
  - $\mathbf{C_t = f_t * C_{t-1} + \cdots}$

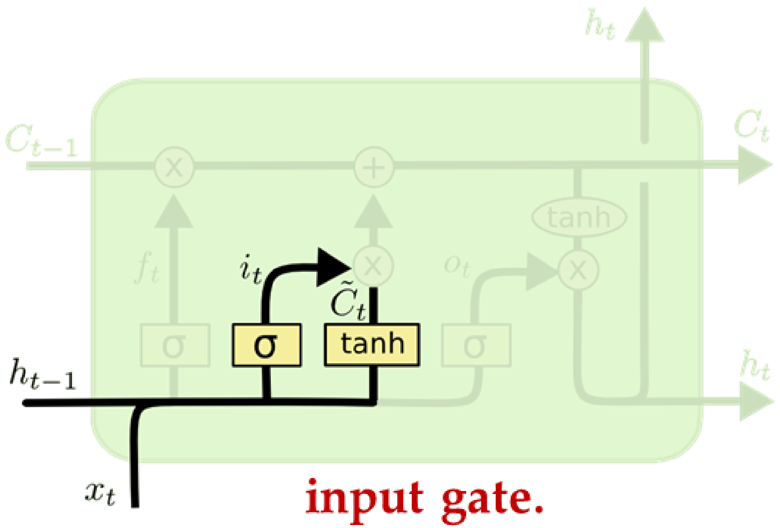


Forget Gate.

$*\rightarrow$ **Elementwise Matrix Multiplication.**

3.3.2 Gate – 1 – input Gate.

- **Purpose:**
  - Decides what new information to store in the cell state.
- **Operation:**
  - Operation – 1 – Input gate layer (sigmoid):
    - $\mathbf{i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)}$
  - Operation – 2 – Candidate values:
    - $\mathbf{\tilde{C}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c)}$
- **Effect**:
  - Current cell state's status is updated as:
    - $\mathbf{C_t = f_t * C_{t-1} + i_t * \tilde{C}_t}$

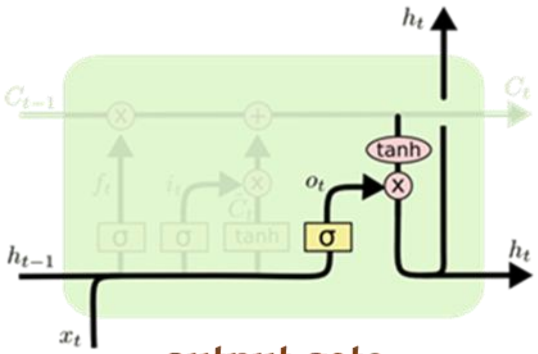


input gate.

$*\rightarrow$ **Elementwise Matrix Multiplication.**

3.3.3 Gate – 1 – output Gate.

- **Purpose:**
  - Determines the next hidden state h_t, which is also the output of the LSTM cell.
- **Operations:**
  - Operation – 1 – sigmoid layer:
    - Decides which parts of the cell state will influence the output.
      - $\mathbf{o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)}$
  - Operation – 2 – Hidden state calculation:
    - Pass the updated cell state through tanh and multiply element wise **with $\mathbf{o_t}$** to get the final **hidden state $\mathbf{h_t}$**:
      - $\mathbf{h_t = o_t * tanh(C_t)}$



output gate.

$*\rightarrow$ **Elementwise Matrix Multiplication.**

3.4 LSTM - Summary

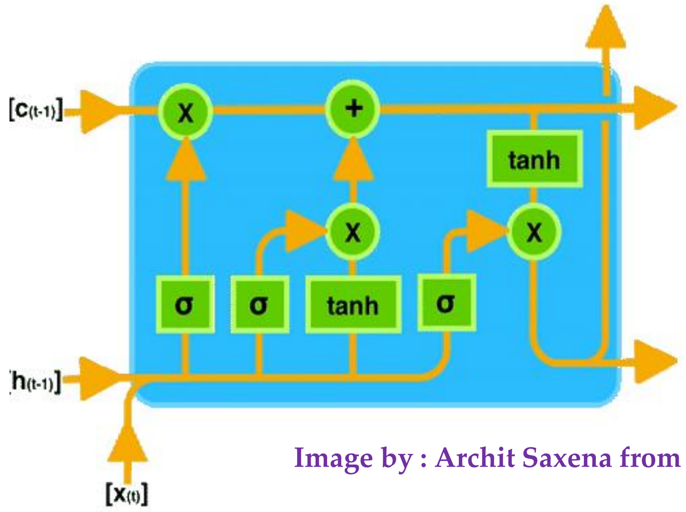
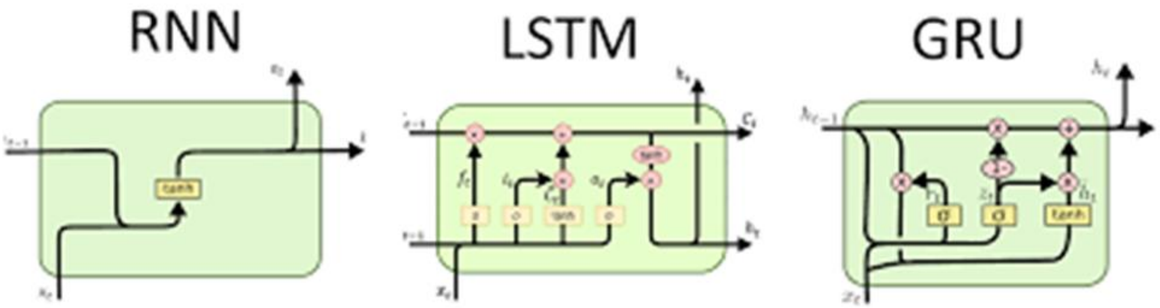


Image by : Archit Saxena from medium p

- Maintain a cell state.
- Use gate to control the flow of information.
  - Forget gate gets rid of irrelevant information.
  - Selectively update cell state.
  - Output gate return a filtered version of the cell state.
- Backpropagation through time with partially uninterrupted gradient flow.
- Also, There is similar architecture called **Gated Recurrence Unit**, :
  - The **Gated Recurrent Unit (GRU)** is a recurrent neural network architecture introduced by **Cho et al. in 2014**.
  - It is a **simplified variant of LSTM**, designed to solve the **vanishing gradient problem** with fewer parameters and faster training.

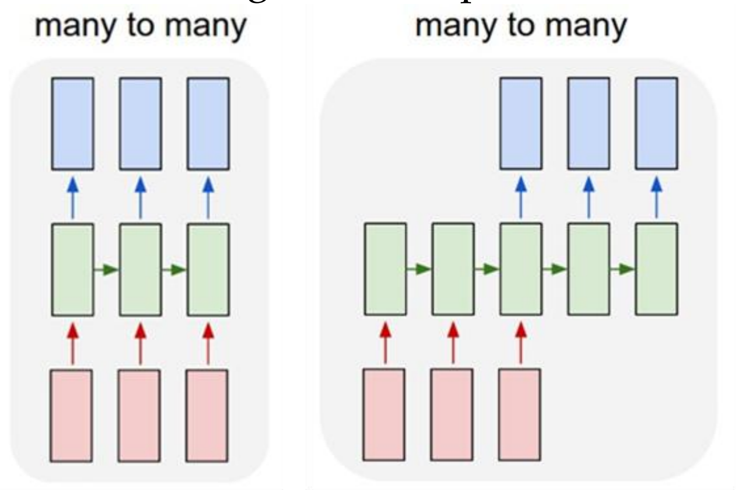


4. Seq to Seq Learning with RNN.

{ An Example of Machine Translation Task.}

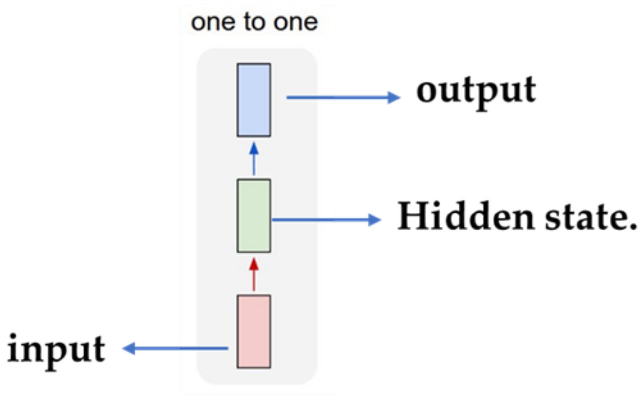
4.1 Sequence-to-Sequence (Seq2Seq) model

- Developed by Google in 2018 for use in machine translation.
- What is "Seq – 2 – Seq"?
  - **Goal:**
    - Converts one sequence into another, such as a sentence in English to a sentence in French.
  - **Powered by:**
    - RNNs, but more effectively LSTMs or GRUs to handle long – term dependencies and prevent the vanishing gradient problem.
- **Applications:**
  - Machine Translation.
  - Text Summarization.
  - Chatbots.
  - Speech Recognition.

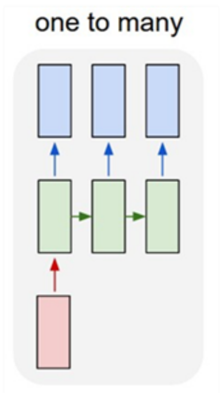


Input: Sequence.
Output: Sequence.
Example: Machine Translation.

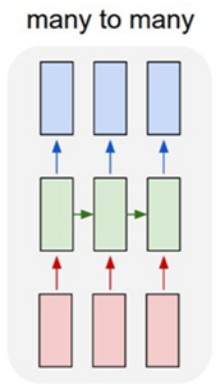
4.2 Sequences in "input" or in "output" – Example.



Input: No Sequence.
Output: No Sequence.
Example: Standard Classification/Regression Problems.



Input: No Sequence.
Output: Sequence.
Example: Image Captions Generator.



Input: Sequence.
Output: Sequence.
Example: Machine Translation.

Fig: Architecture other than "Seq to Seq" Model.

4.3 Training "Sequences" in Output.

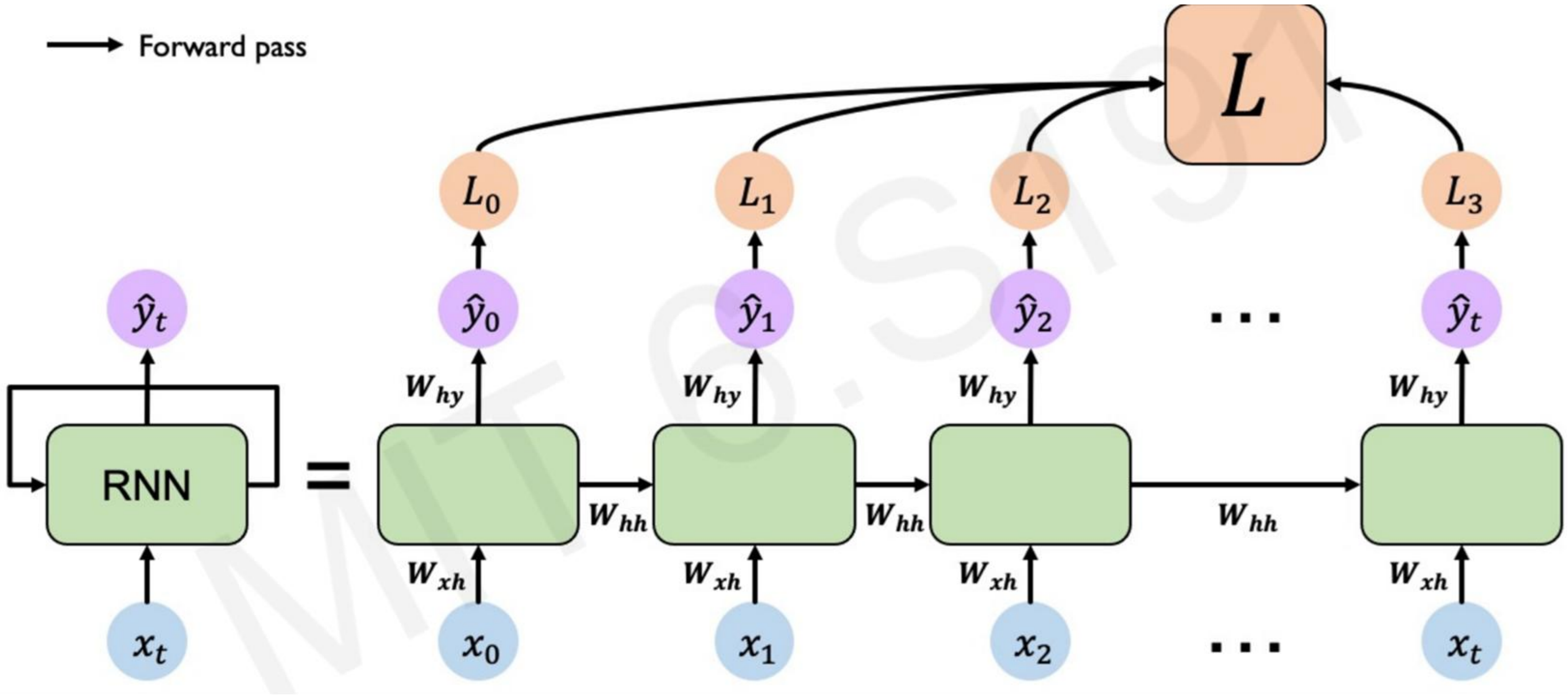


Image from MIT 6.S19 Deep Learning.

4.3.1 Training "Sequences" in Output.

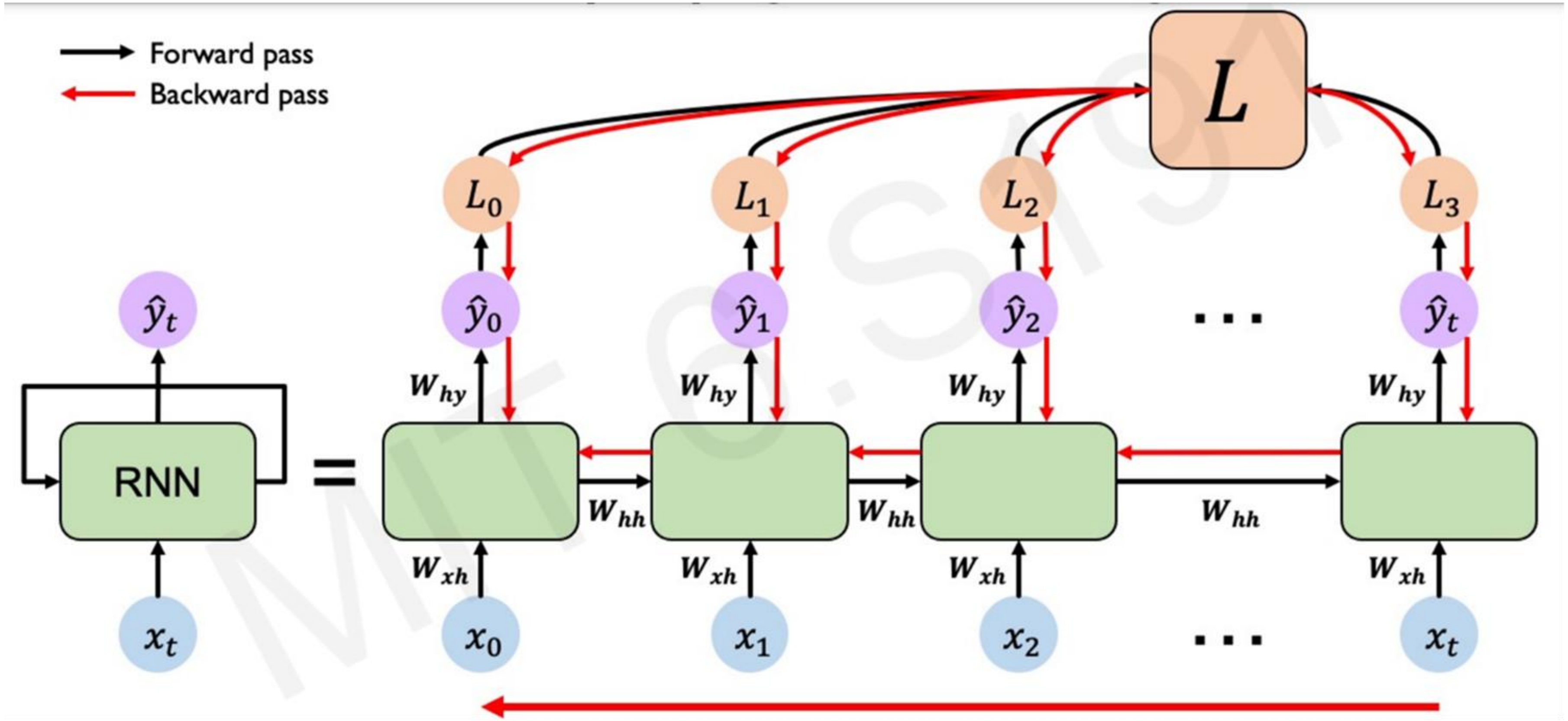
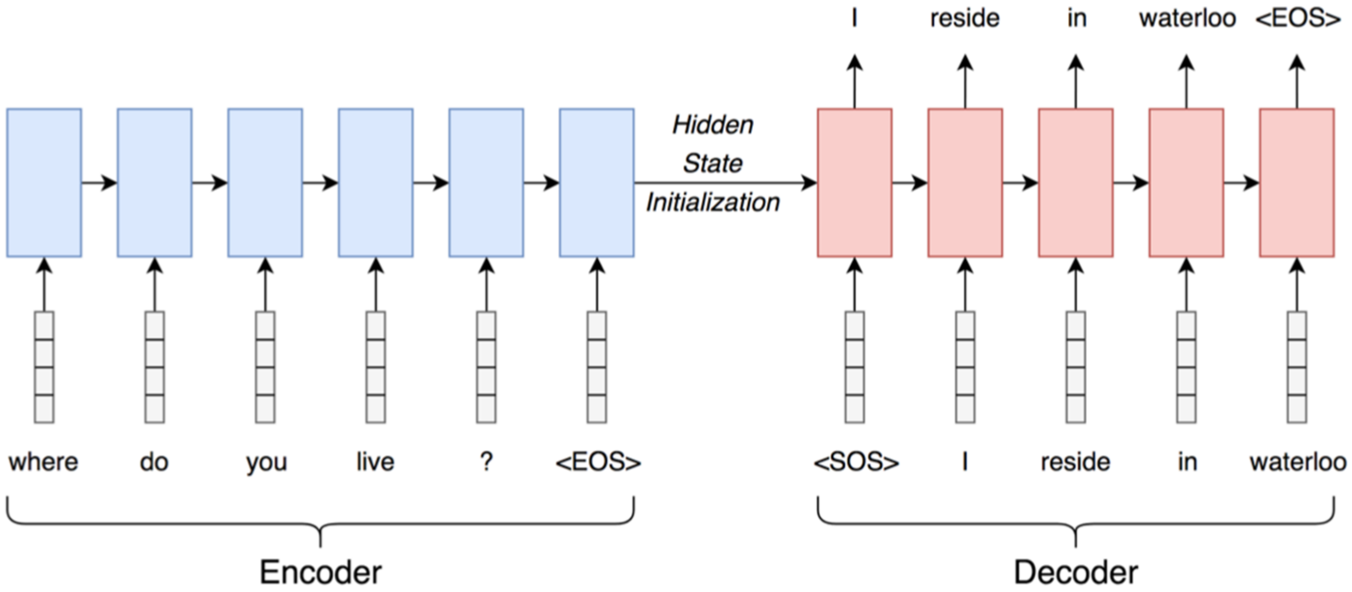


Image from MIT 6.S19 Deep Learning.

- Back-propagation Through Time:

Seq – Seq Model for Question and Answering: An Example.

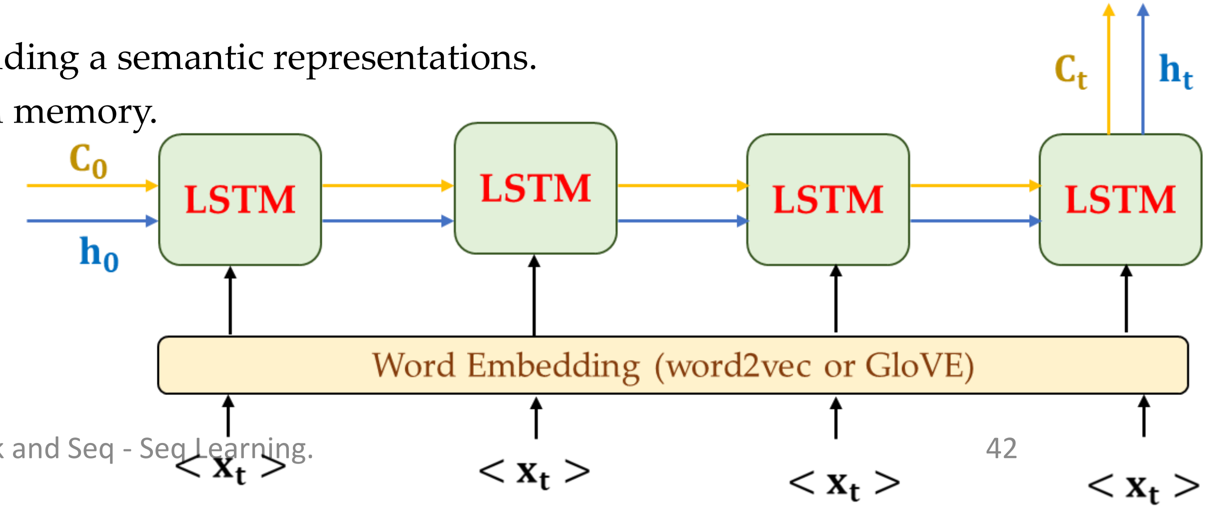


- In a vanilla **Seq2Seq model**, the **final hidden state (and cell state)** of the **Encoder** becomes the **initial hidden state (and cell state)** of the **Decoder**.
- This is how the **context of the input sentence** is transferred to the **decoder** for **output generation**.

4.4 Encoder – Decoder for Machine Translation.

Encoder:

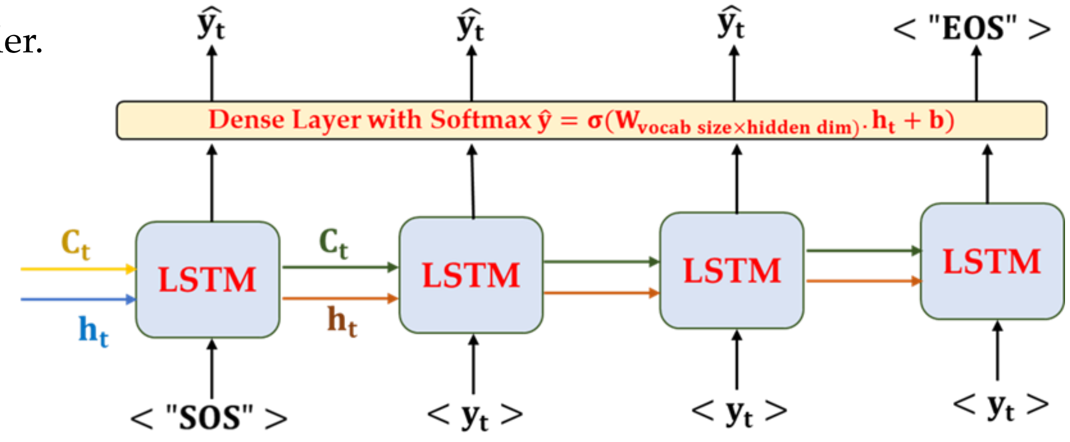
- Processes the input sentence word by word.
- Converts the entire sequence into a fixed – length context vector (hidden state + cell state for LSTM).
- Learns to represent semantics and context of the source sentence.
- **Current Input Word (x_t):**
  - This is the word/token at time step t from the input sentences. (“I”, “love”, “you”).
- **Output of Encoder – Hidden State and Cell State (h_t & C_t):**
  - $h_t, C_t = \text{LSTM\_enc}\,(x_t, h_{t-1}, C_{t-1})$
  - This holds the context of all previous words and helps in building a semantic representations.
  - In LSTM, it also includes the cell state C_t to carry long – term memory.



4.4 Encoder – Decoder for Machine Translation.

Decoder:

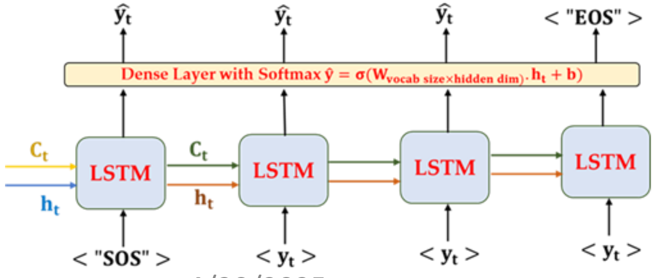
- Initialization:
  - Takes the context vector (final hidden and cell states) from the encoder.
    - $h_0^{dec} = h_T^{enc}; C_0^{dec} = C_T^{enc}$
- Function:
  - Generates the target sentence one word at a time.
    - At each time step t, it uses:
      - The previous output word y_{t-1}.
      - The previous hidden state h_{t-1}.
      - The previous cell state C_{t-1}.
  - Formal update at Time Step t:
    - $h_t, C_t = LSTM_{dec}(y_{t-1}, h_{t-1}, C_{t-1})$
  - Output Layer:
    - The decoder's **hidden state h_t** is passed through Dense layer followed by a softmax to predict the next word.
      - $\hat{y}_t = softmax(W.h_t + b)$



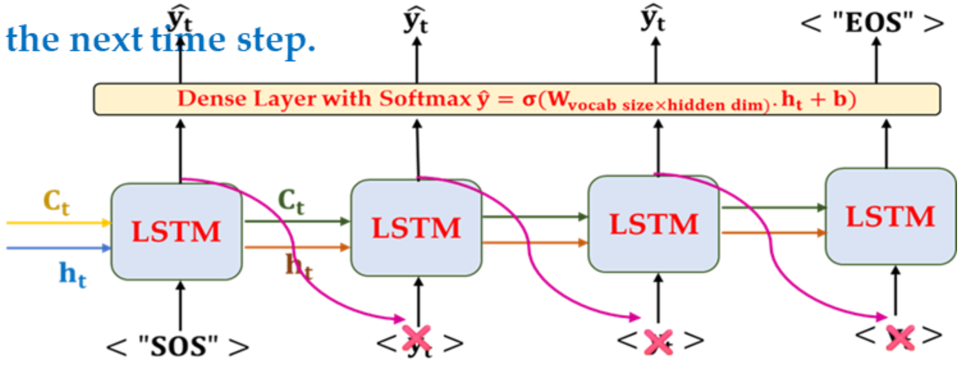
Decoder with Teacher Forcing.

4.4.1 What is Teacher Forcing?

- In **sequence-to-sequence models** (like vanilla machine translation), **teacher forcing** means:
- At each decoder time step **during training**, instead of feeding the ***previous predicted word***, you feed the **actual ground-truth word** from **the target sentence**.
- Why use Teacher Forcing?
  - It helps the model learn **faster and more accurately**, especially early in training.
- It avoids **cascading errors** — because if the decoder made a mistake in the previous step, feeding that into the next one could lead to even worse predictions.
- Teacher Forcing During Inference or Testing:
  - **Teacher Forcing ❌ is NOT used during Inference.**
  - There is no ground truth available so:
    - **We do not use teacher forcing.**
    - **The decoder must feed its own previous prediction back as input for the next time step.**
    - **This is also known as autoregressive decoding.**



Decoder with Teacher Forcing.



During Inference/ Testing

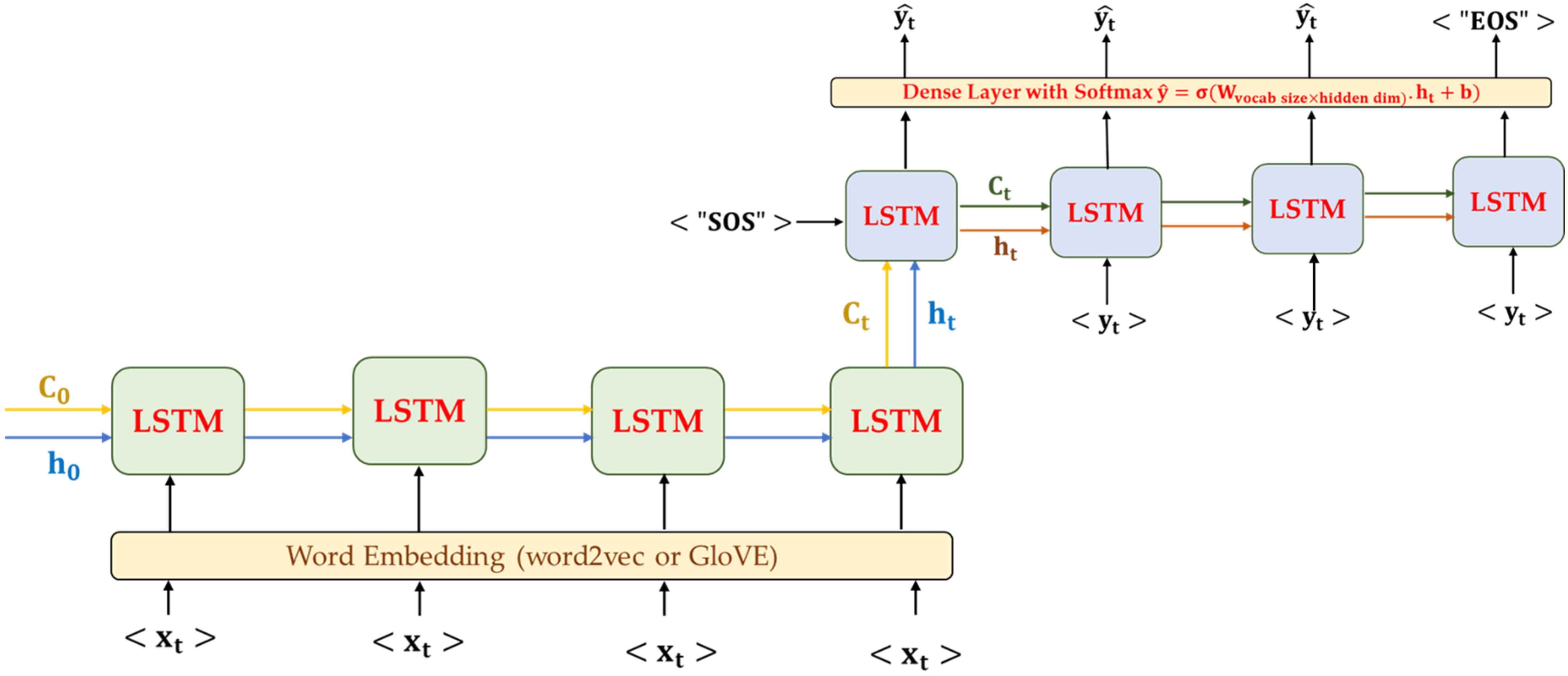
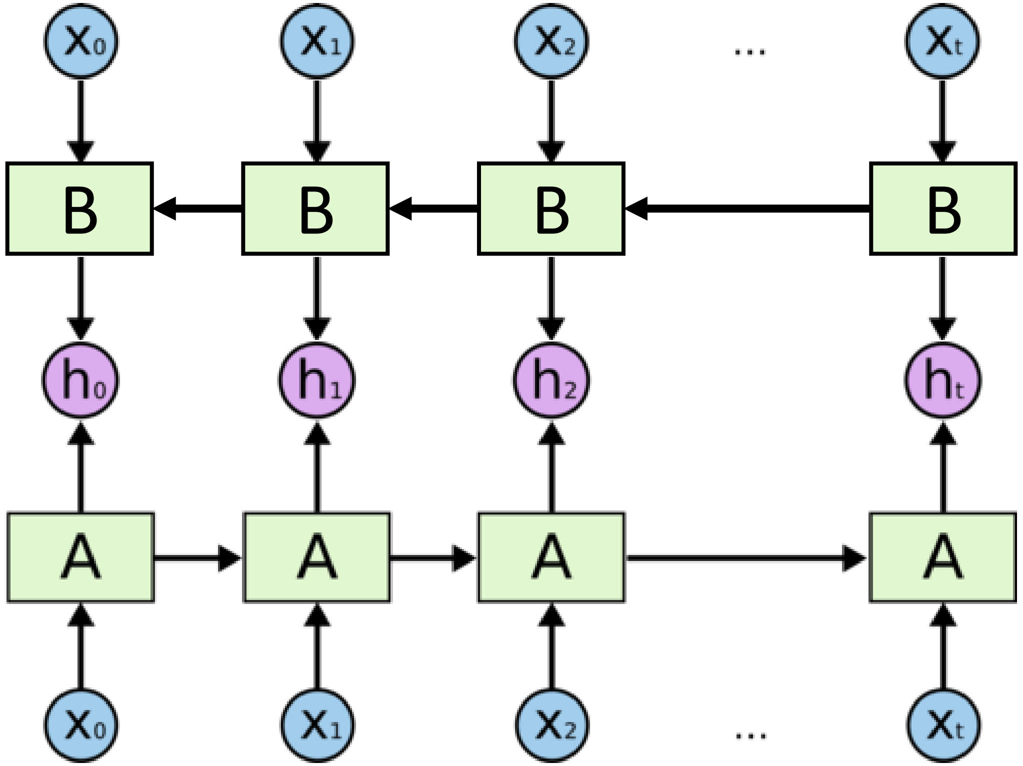


Fig: A Vanilla Encoder – Decoder for Machine Translation.

4.5 Stacking – Multiple RNN Layers

- **Bidirectional RNN**
  - Connects two recurrent units (synced many-to-many model) of opposite directions to the same output.
  - Captures forward and backward information from the input sequence
  - Apply to data whose current state (**e.g., h_0**) can be better determined when given future information **(e.g., $x_1, x_2, \ldots, x_t$)**
    - E.g., in the sentence "the bank is robbed," the semantics of "bank" can be determined given the verb "robbed."



Thank You.