RNN, Memory, and Attention

Yue Qiu[⋆], Kaixin Wang[†], **Tian Xie**[‡], Tao Zeng [†], and Xiangzhong Zheng[‡]

[⋆]Shanghai University of International Business and Economics
[†]Zhejiang University
[‡]**Shanghai University of Finance and Economics**

November 15, 2023 @ **LLM Workshop Series**

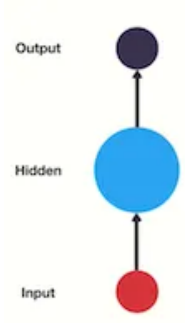
Recurrent Neural Network Model

Recurrent Neural Network Model

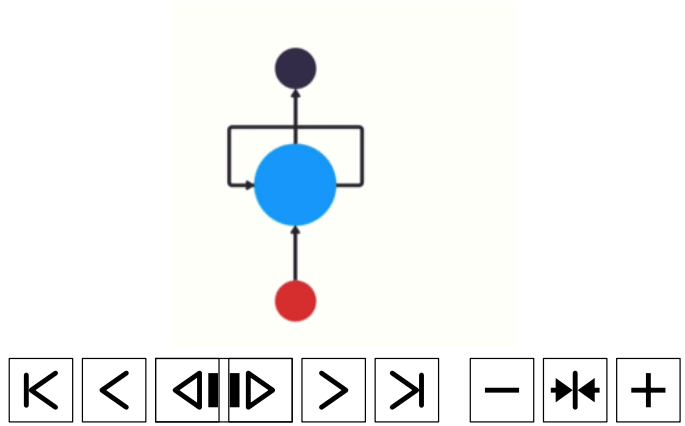
- RNNs are a class of neural networks designed for sequence data, where the **order** of elements matters.
- Dynamic architecture with **recurrent connections** that allow information to be passed between time steps.
- Contains hidden states that store information from **previous** time steps.
  - Hidden states are similar to "memory", but can be quite limited, especially for vanilla RNN.

RNN vs. FNN

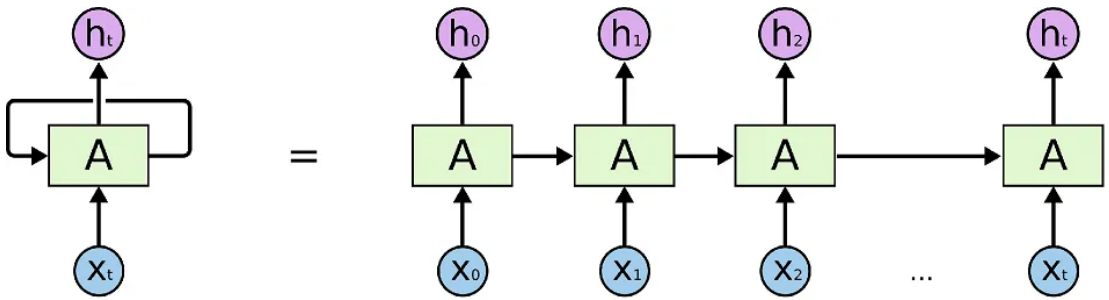
- A simple FNN has the input coming in, **red** dot, to the hidden layer, **blue** dot, which results in **black** dot output.

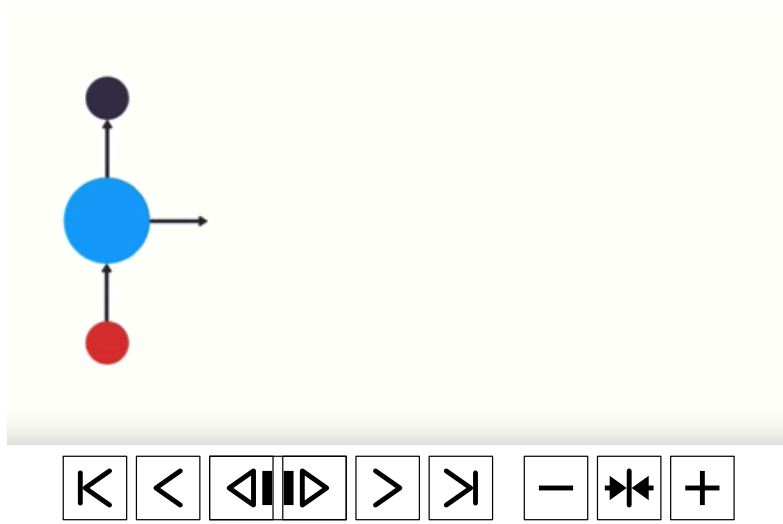


- An RNN feeds it's output to itself at **next time-step**, forming a **loop**, passing down much needed information.



- To better understand the **flow**, look at the unrolled version below, where each RNN has **different** input (token in a sequence) and output at each time step.



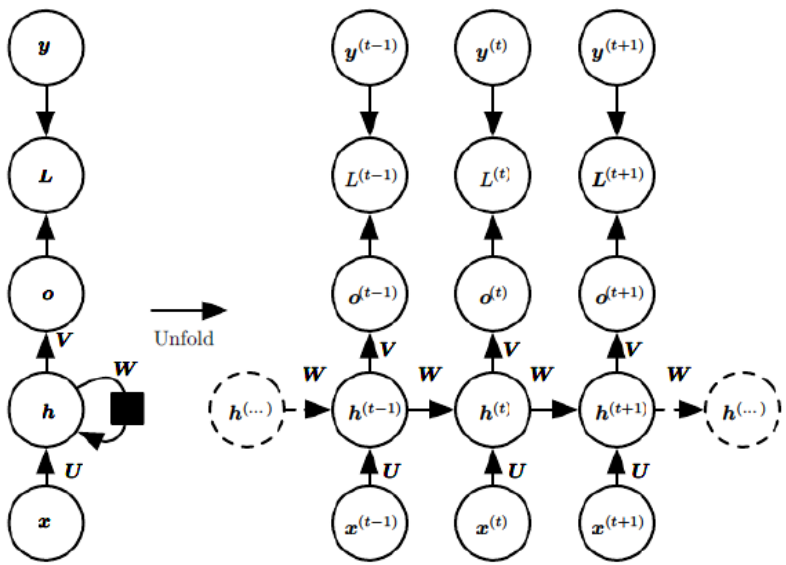


Formal Definition by Goodfellow et al. (2015)

- Many recurrent neural networks use the following equation or a similar equation to define the values of their hidden units.

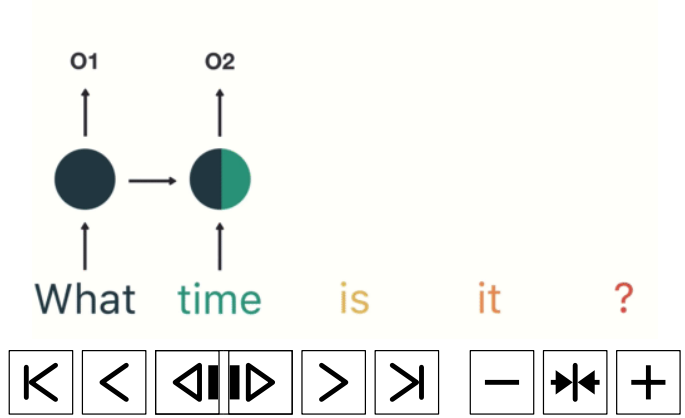
$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \boldsymbol{\theta}).$$

where the variable $\mathbf{h}$ represents the state, $\mathbf{x}$ is the input, and $\boldsymbol{\theta}$ is the parameter vector.

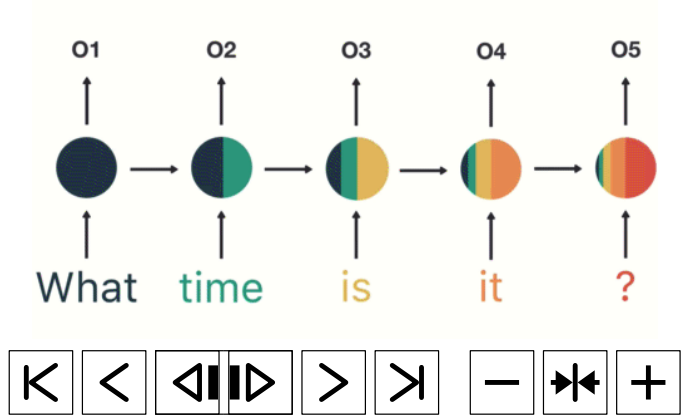


Example: Text Recognition

- We use RNN to analyze the text "What time is it?"
- The RNN first breaks the **sequence** into **tokens**.



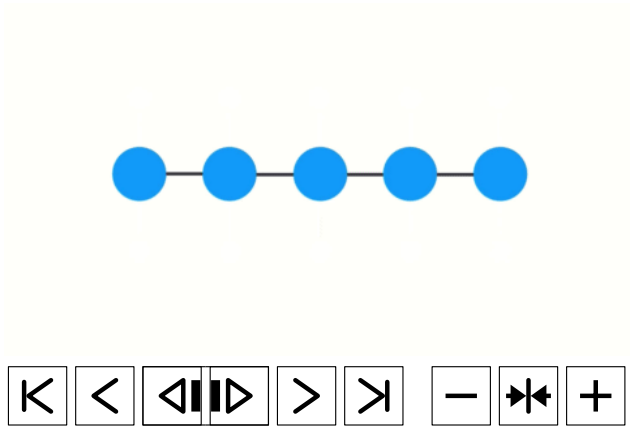
- Then, RNN retrieves the **final query** as the processed **outcome**.



The Limitation of Vanilla RNN

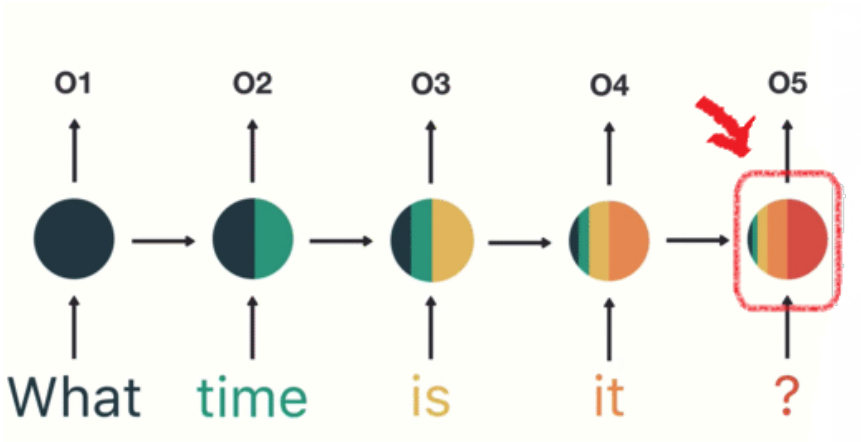
- The limitation of vanilla RNNs in capturing long-term dependencies is primarily due to the **vanishing gradient problem**: given the loss function (L_t) and the network's parameters (θ), during back-propagation, we have

$$\lim_{t \to \infty} \frac{\partial L_t}{\partial \theta} \approx 0$$



- As backpropagation through time spans multiple steps, $\nabla \to \mathbf{0}$, having **minimal impact** on weight updates.

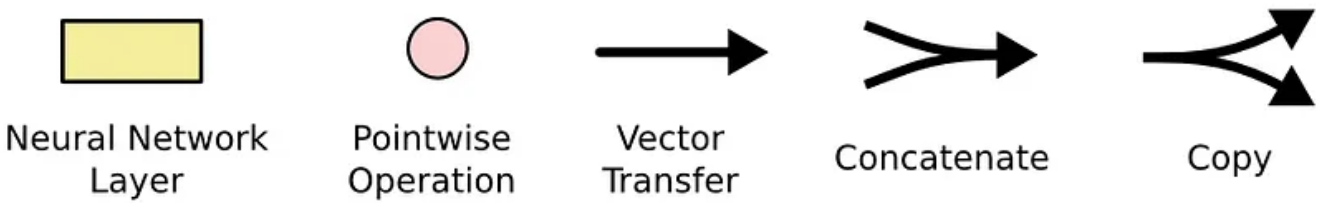
Any "Forest Green" Left in the Last Cell?



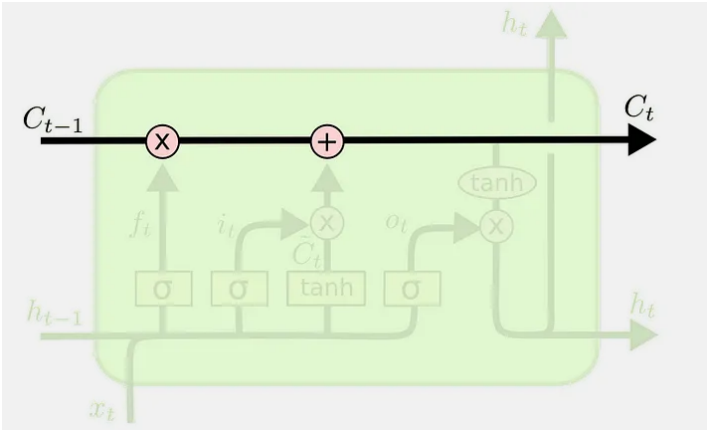
- This phenomenon is **particularly problematic** for learning **long-term dependencies** in sequential data.

Long Short-Term Memory Model

- LSTM is a specialized version of RNNs with **enhanced memory** capabilities.
- It is designed to address the **vanishing gradient problem** in vanilla RNNs, allowing for better learning of long-range dependencies in sequences.
- Some notations:

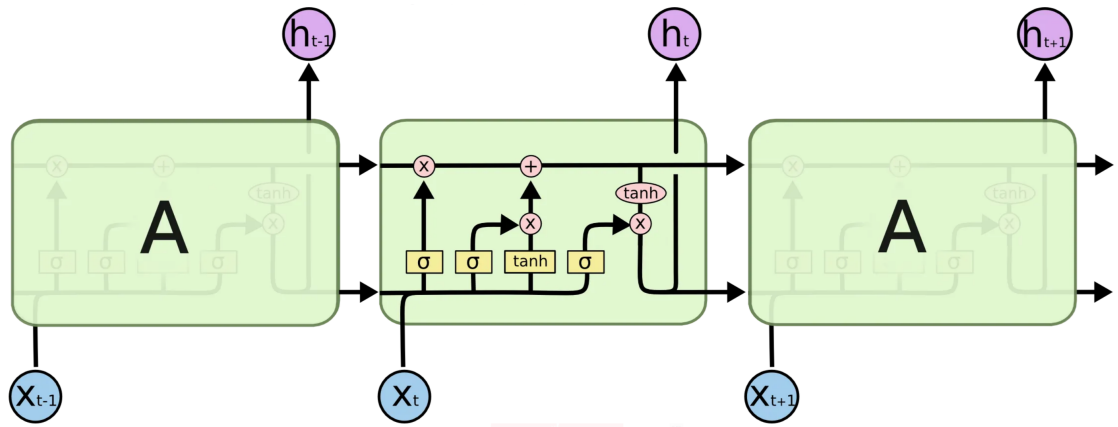


- LSTMs shine with a dedicated **cell state** (C_t) for each node, **passed down** the chain.



- We can **modify** the cell state with **linear mathematical interactions** regulated by other **inner components**.
  - These inner components, known as **gates** with activation functions, can add or remove information to the cell state.
- A simple RNN has a simple NN in itself acting as a **sole gate** for some data manipulations.

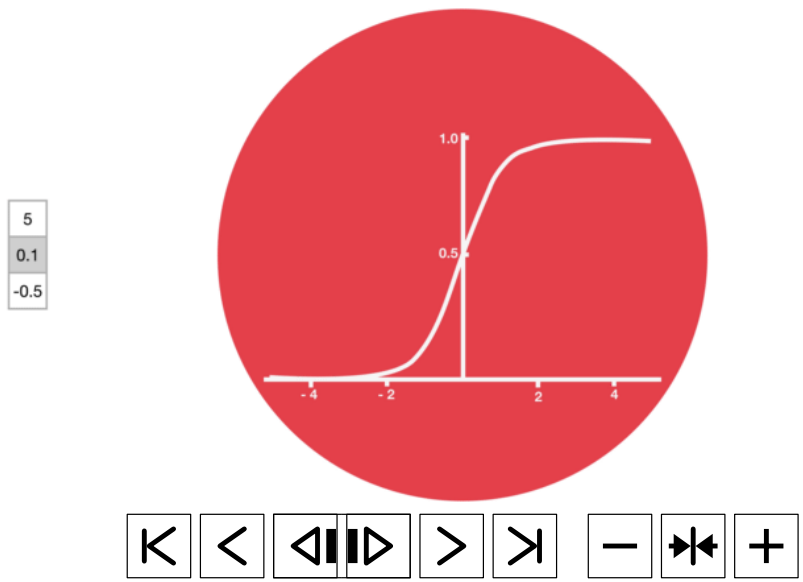
- LSTM has **four gates**, which grant its ability to remove or add information to the cell state.
- Here is a LSTM chain of 3 nodes and internal structure depiction



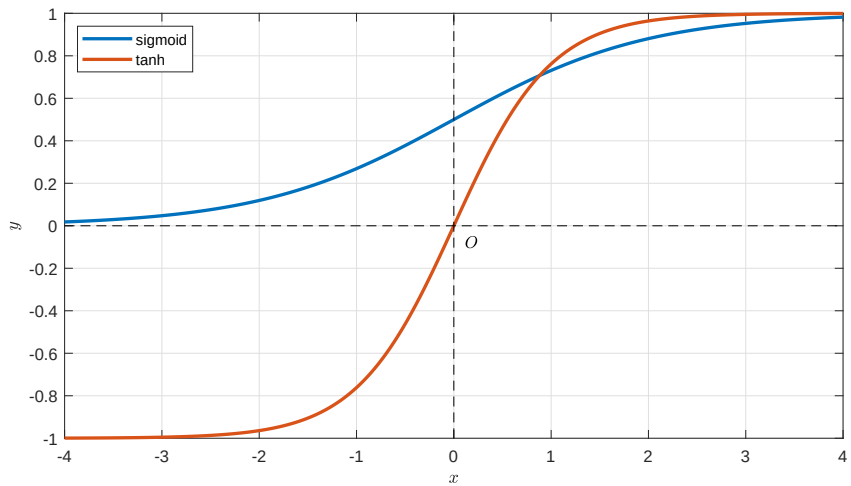
- The gates allow information to be optionally altered and let through. For this **sigmoid** and **tanh** gates are used internally.

sigmoid and tanh Activation

- The sigmoid helps to squash the incoming values between 0 and 1 ($[0, 1]$). This is used in conjugation of other component to stop(if 0) or allow(1) incoming information.



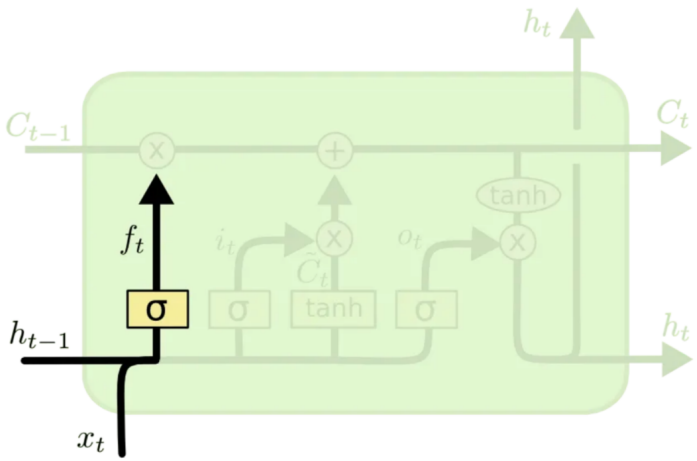
- Tanh on other hand helps to compress the values in range -1 to $+1$.



- These activation functions also help to curb the outputs which otherwise would just explode after successive multiplications along the chain.

Inside the LSTM Layer: Step 1

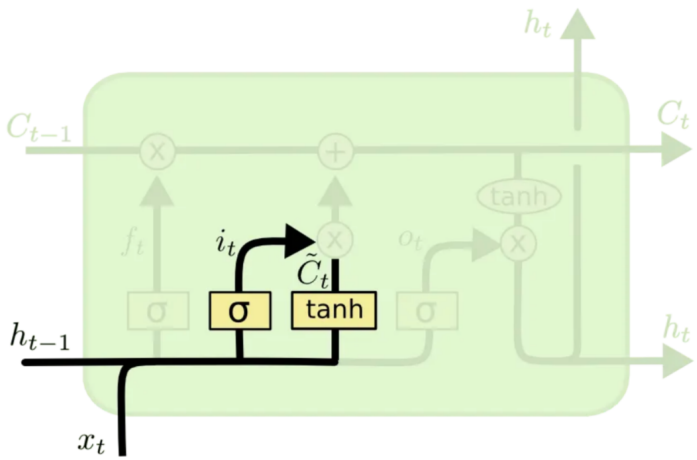
- First step is to decide what all should be **forgotten** from the cell state.
  - A Sigmoid is used in forget gate layer.
- The forget gate looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1} .



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

Inside the LSTM Layer: Step 2

- Second step involves deciding what new data should be **added back** to the cell state.
- This involves 2 parts and are used in conjunction.
  - sigmoid gate layer deciding what exactly to change (with 0 to 1 range)
  - tanh gate layer creating candidate values that could be added to the cell state (with -1 to $+1$ range)

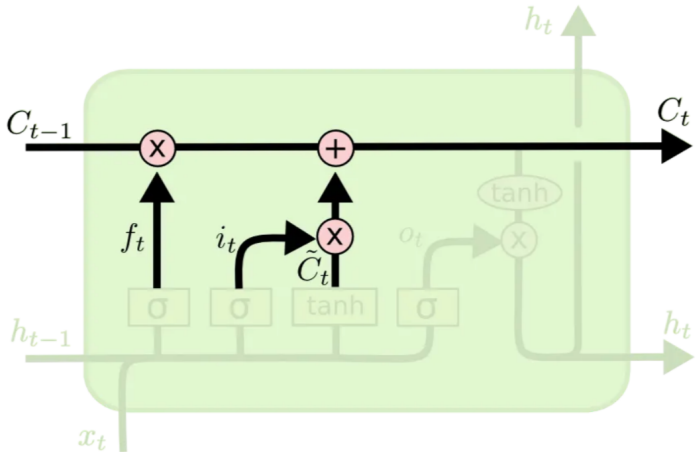


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

Inside the LSTM Layer: Step 3

- This step includes **manipulating the incoming** cell state C_{t-1} to reflect the **decisions** from forget (**step 1**) and input gate (**step 2**) layers with help of point-wise multiplication and addition respectively.

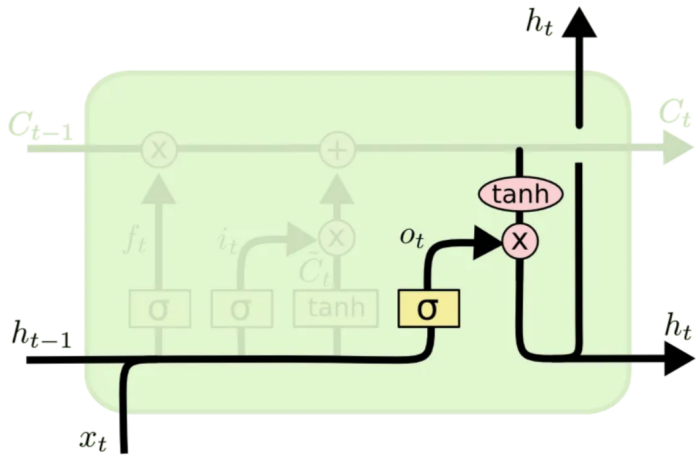


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- Here f_t is for point-wise multiplication with C_{t-1} to forget information from context vector and $i_t * C_t$ is new candidate values scaled by how much they needs to be updated.

Inside the LSTM Layer: Step 4

- The output is a modified version of the cell state. For this sigmoid decides what part needs to be modified, which is multiplied with output after tanh (used for scaling), resulting in **next period** hidden state output **(h_t)**.

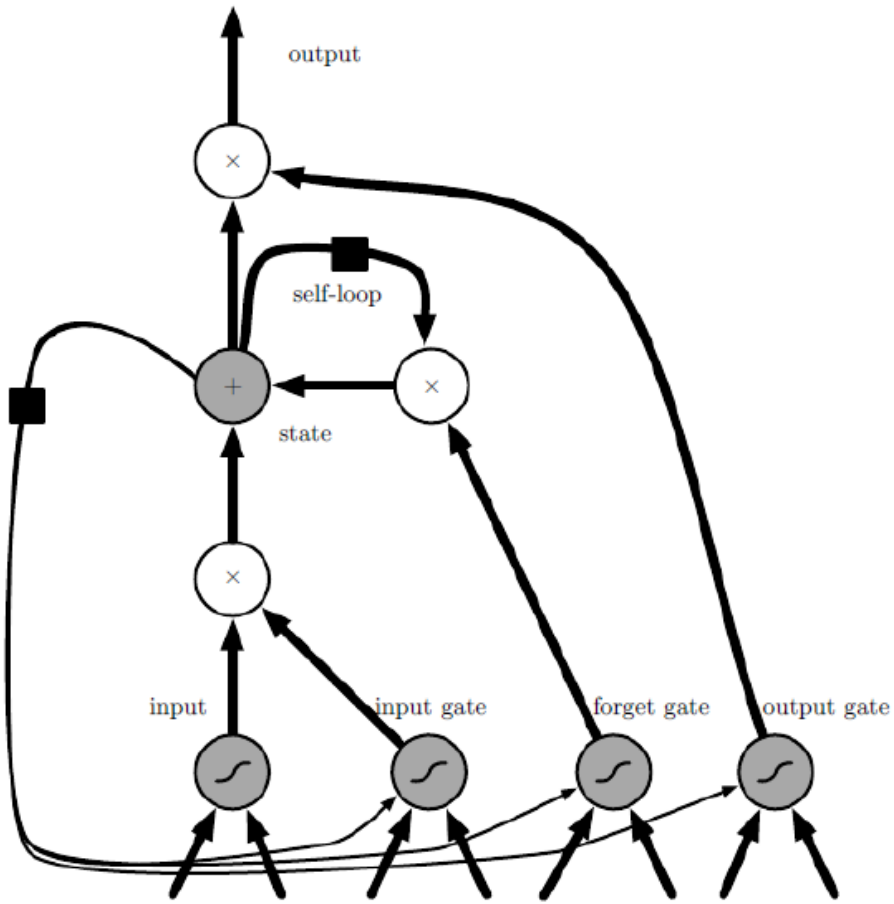


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

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

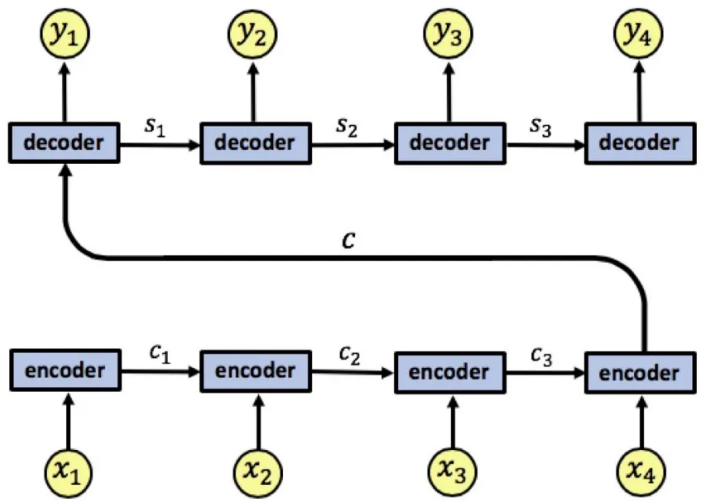
Formal Definition by Goodfellow et al. (2015)

- The core components of an LSTM cell include:
  - **Cell State (c_t)**: The long-term memory that can store information over long sequences.
  - **Hidden State (h_t)**: The short-term memory or output that carries information to the next time step.
  - Three gates:
    - **Forget Gate (f_t)**: Determines what information from the cell state should be discarded or kept.
    - **Input Gate (i_t)**: Determines what new information should be stored in the cell state.
    - **Output Gate (o_t)**: Determines the next hidden state based on the cell state.
- The LSTM architecture allows it to selectively update and access the cell state, making it powerful for capturing and retaining information over extended sequences.


output
self-loop
state
input
input gate
forget gate
output gate

Example: Sequence-to-sequence Model

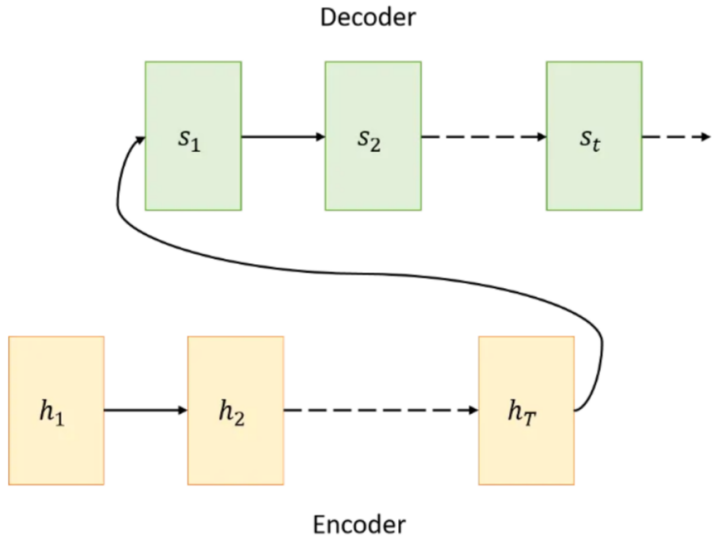
- Sequence-to-sequence (seq2seq) models in NLP are used to convert sequences of Type A to sequences of Type B.
  - Machine Translation, Text Summarization, Speech-to-Text, Chatbot Responses, Question Answering, Time Series Prediction, etc.
- The LSTM encoder has an input sequence x_1,..., x_4. We denote the encoder states by c_1, c_2, c_3. The encoder outputs a single output vector c which is passed as input to the decoder.
- The LSTM decoder states are denoted as by s_1, s_2, s_3 and the network's output by y_1, ..., y_4.



Attention Mechanism

Issues with Vanilla LSTM-based seq2seq Model

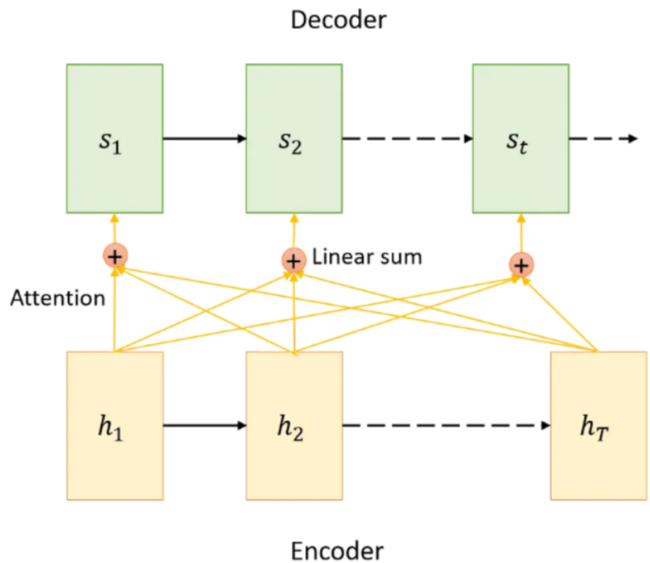
- Take a look at the seq2seq model example again:



- The encoding step compresses the entire input sequence into a single vector, risking potential **information loss**.
- The decoder faces a **complex task** extracting information from the compressed vector, limiting the model's understanding and performance.
- The encoder-decoder approach encounters difficulty handling **long input** sequences, especially those surpassing the length of sentences in the training data.

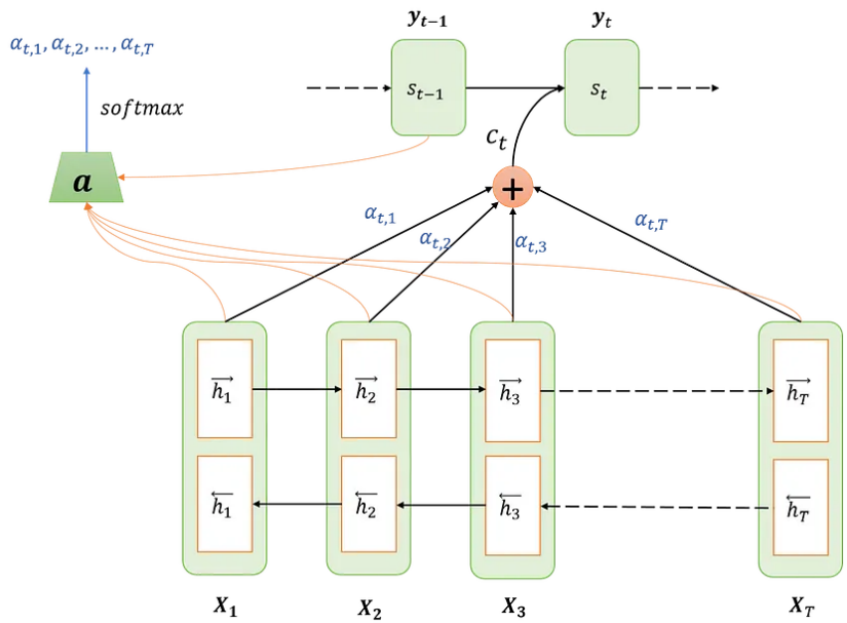
Attention Mechanism to the Rescue

- **Attention** based architecture attends every hidden state from each encoder node at every time step and then makes predictions after deciding which one is more informative.



- But how does it decide which states are more or less useful for every prediction at every time step of decoder?

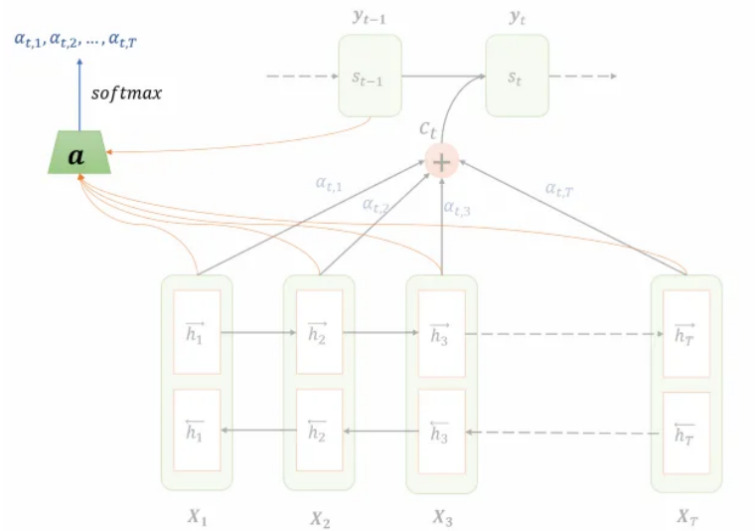
- **Solution**: Use a Neural Network to **learn** which hidden encoder states to **attend** to and by how much.
  1. Encoding
  2. Computing Attention weights
  3. Creating context vector
  4. Decoding / Translation



Calculate Attention Weights

- For each time step t at the decoder, the level/amount of attention to be paid to the hidden encoder unit h_j is denoted by α_{tj} and calculated as a function of both h_j and previous hidden state of decoder s_{t-1}.
- Many attention block uses a **Softmax** unit at the end to **normalize** the outgoing attention values:

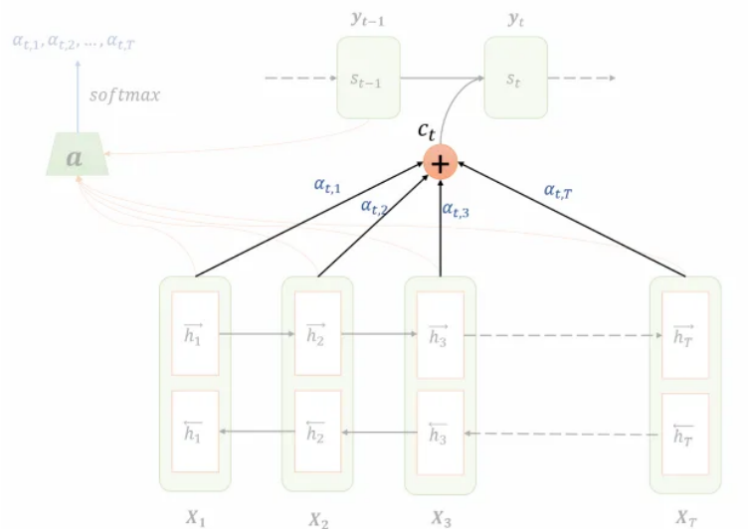
$$\begin{aligned} e_{tj} &= \alpha(h_j, s_{t-1}) \\ \alpha_{tj} &= \frac{\exp(e_{tj})}{\sum_{k=1}^{T} \exp(e_{tk})} \end{aligned} \qquad \forall j \in [1, T]$$



Creating Context Vector

- The context vector is a simple linear combination of the hidden values.

$$c_t = \sum_{j=1}^{T} \alpha_{tj} h_j$$



- Final step is the decoding step. c_t is used along with the previous hidden state of the decoder s_{t-1} to compute the new hidden state and output of the decoder: s_t and y_t.