Recurrent Neural Networks (RNNs)

AI with Deep Learning

EE4016

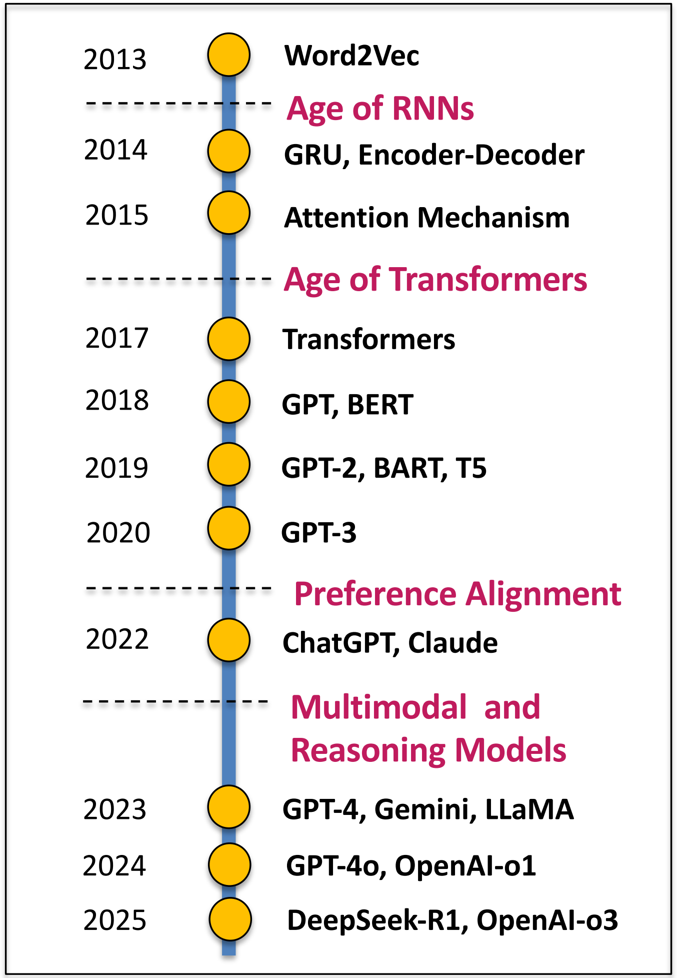
Prof. Lai-Man Po

Department of Electrical Engineering
City University of Hong Kong

https://medium.com/@lmpo/a-journey-through-rnn-lstm-gru-and-beyond-080cf1f13b41

LLMs: From Word2Vec to DeepSeek-R1 (2012-2025)

1. Tokenization and Word2Vec: BPE, CBOW, Skip-Gram
2. RNNs, LSTM, GRU, Seq-to-Seq (Encoder-Decoder) and Attention Mechanisms
3. Transformers with Self-Attention
4. Lager Language Models (LLMs): BERT, GPT, BART, T5
5. Preference Alignment by SFT and RLHF: ChatGPT, Claude, LLaMA
6. Multimodal Models: GPT-4, GPT-4o, Gemine, LLaVA
7. Reasoning Models: OpenAI-o1, DeepSeek-R1



2013 Word2Vec

Age of RNNs

2014 GRU, Encoder-Decoder

2015 Attention Mechanism

Age of Transformers

2017 Transformers

2018 GPT, BERT

2019 GPT-2, BART, T5

2020 GPT-3

Preference Alignment

2022 ChatGPT, Claude

Multimodal and Reasoning Models

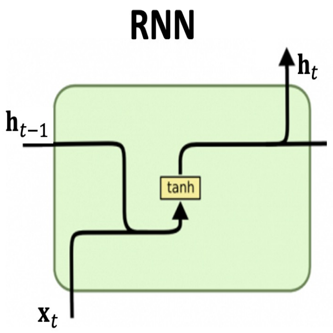
2023 GPT-4, Gemini, LLaMA

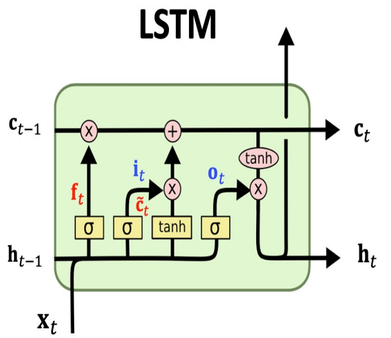
2024 GPT-4o, OpenAI-o1

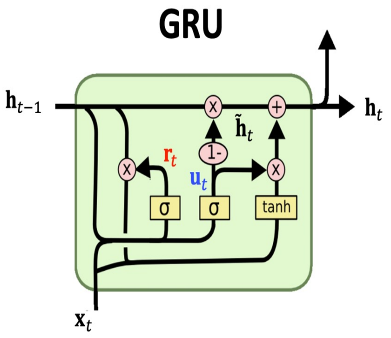
2025 DeepSeek-R1, OpenAI-o3

Content

1. Recurrent Neural Network (RNN) Architecture (1982)
2. Long-Short Term Memory (LSTM, 1997)
3. Gated Recurrent Unit (GRU, 2014)
4. Seq2Seq RNN and Attention Mechanisms
   - Seq2Seq RNN (Encoder-Decoder, 2014)
   - Bahdanau Attention (2014)
   - Luong Attention (2015)





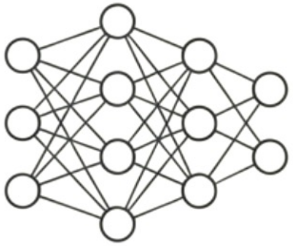


Recurrent Neural Networks (RNNs, 1982)

Neural Network Architecture & Inductive Bias

Selecting the right hypothesis space for the data

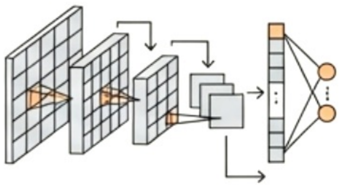
MLP (Multi-Layer Perceptron)



Inductive Bias:
Independence

Use Case:
Tabular Data

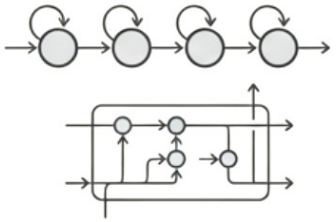
CNN (Convolutional Network)



Inductive Bias:
Spatial Locality & Invariance

Use Case:
Image Data

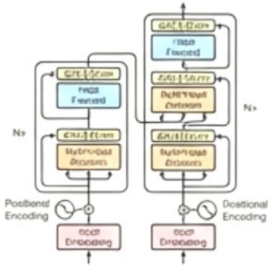
RNN / LSTM



Inductive Bias:
Sequentiality

Use Case:
Time-series, Sequence Data

Transformer

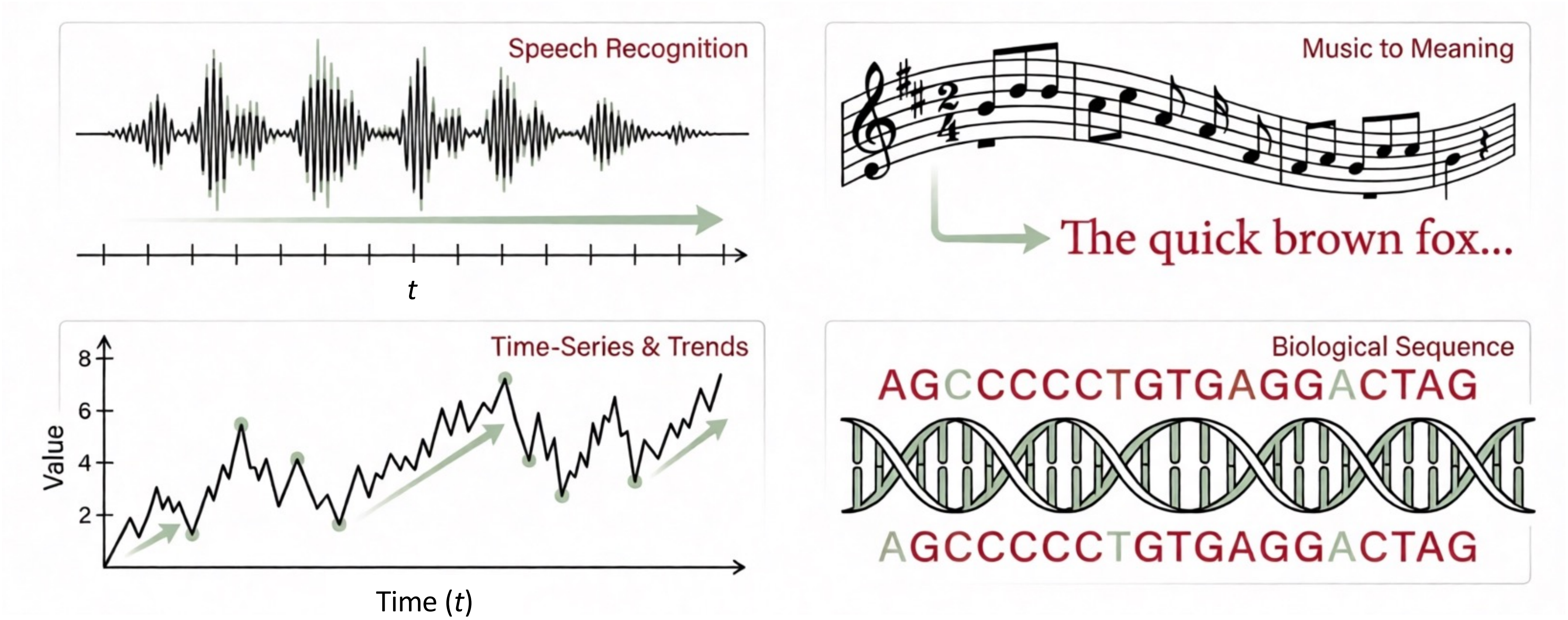


Inductive Bias:
Global Context (Self-Attention)

Use Case:
NLP, Seq2Seq, Vision

The Unique Nature of Sequential Data

Order creates meaning: data with a temporal dimension (*t*).

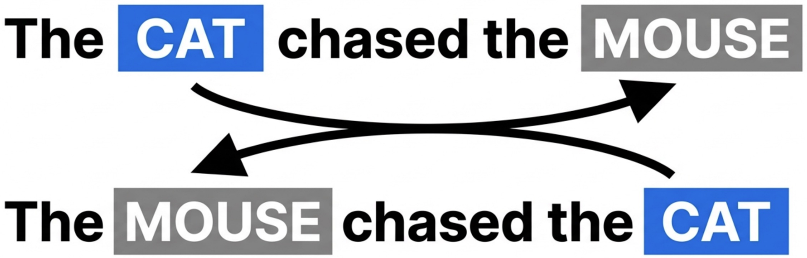


In Sequential Data, Order is Meaning

Text Analysis

Unlike static data, the position of elements in a sequence dictates semantic value.

Subject/Object Flip



Modifier Shift

A: She **only** called her friend yesterday. => Timing Emphasis

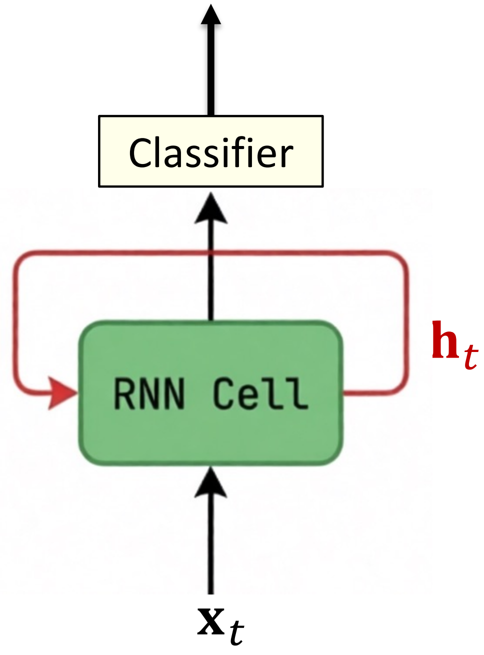
B: She called her **only** friend yesterday. => Uniqueness/Sadness Emphasis

In sequential data, moving an element changes the reality.

Recurrent Neural Networks (RNNs, 1982)

Handling variable-length sequences via parameter sharing

$$\hat{\mathbf{y}}_t = g\left(\mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y\right)$$



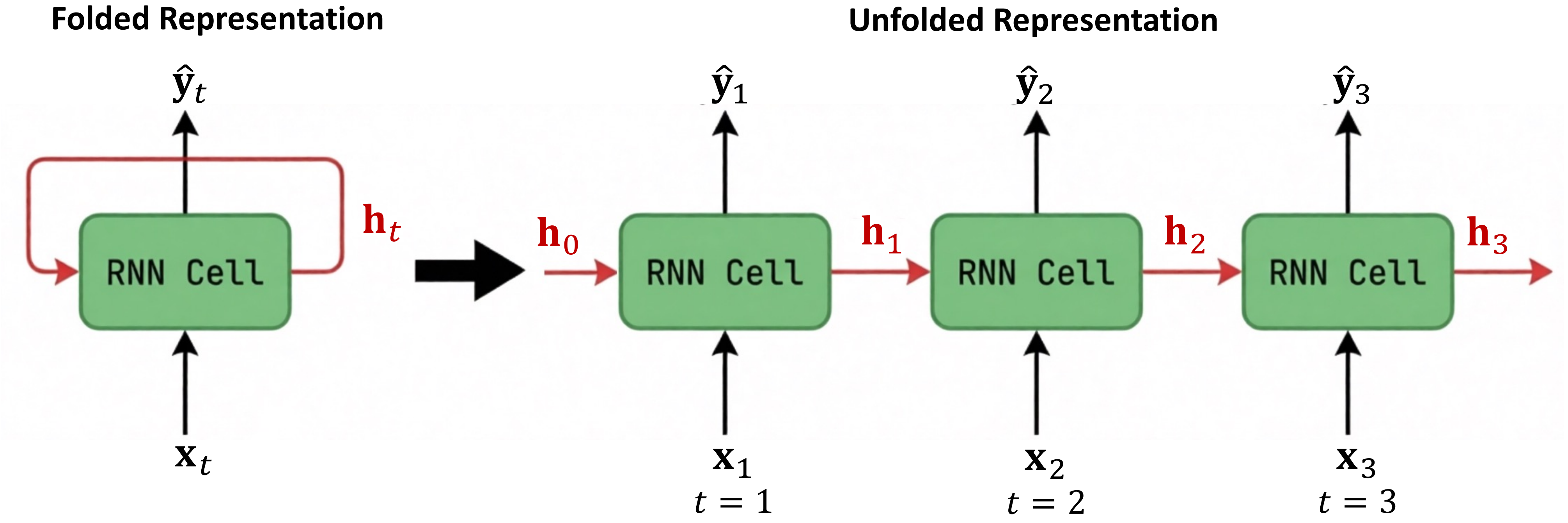
Shared Weights and biases

$$\mathbf{h}_t = \tanh(\mathbf{W}_h \cdot \underbrace{[\mathbf{x}_t, \mathbf{h}_{t-1}]}_{\text{Current Input + Previous Memory}} + \mathbf{b}_h)$$

Activation Function

RNN exhibits a **recurrent nature** by consistently applying the same function to each input while generating the output based on the previous computation.

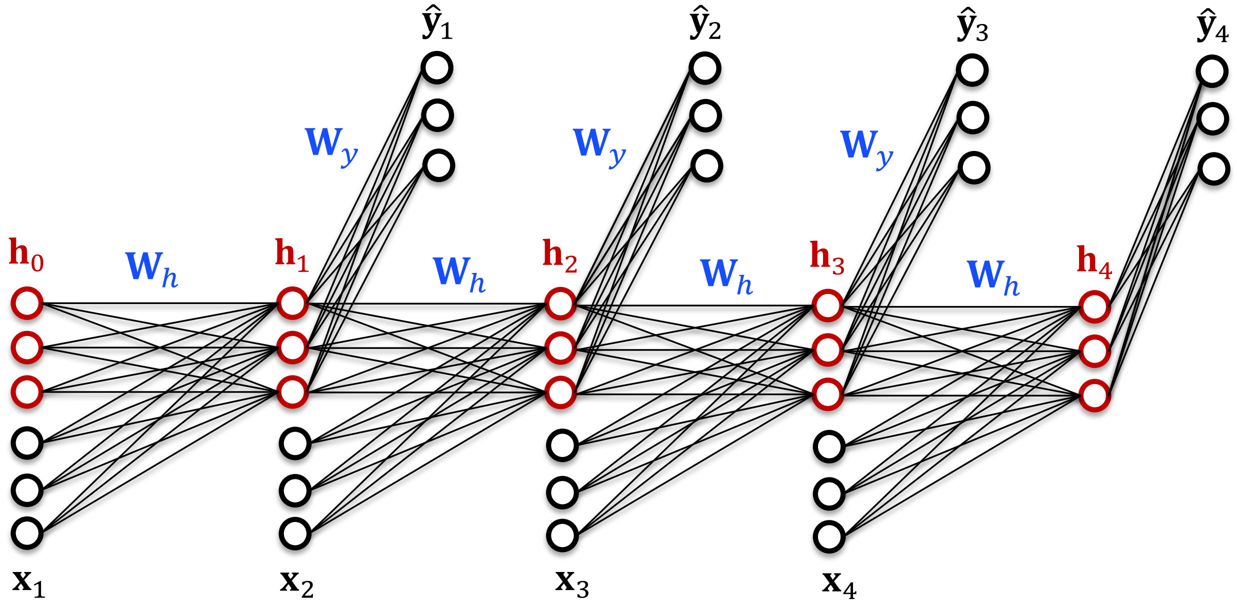
Unfolded Representation of RNNs



- The Hidden State $\mathbf{h}_t$ acts as a short-term memory, passing context from the past to the future.
- The "Loop" mechanism allows information to persist.

Unfolded RNN Architecture

When unrolled through time, a RNN resembles a deep feedforward network whose layers correspond to successive time steps — with the critical distinction that **all time-step layers share the same weight matrices and bias vectors**, enforcing temporal consistency in processing sequential data.



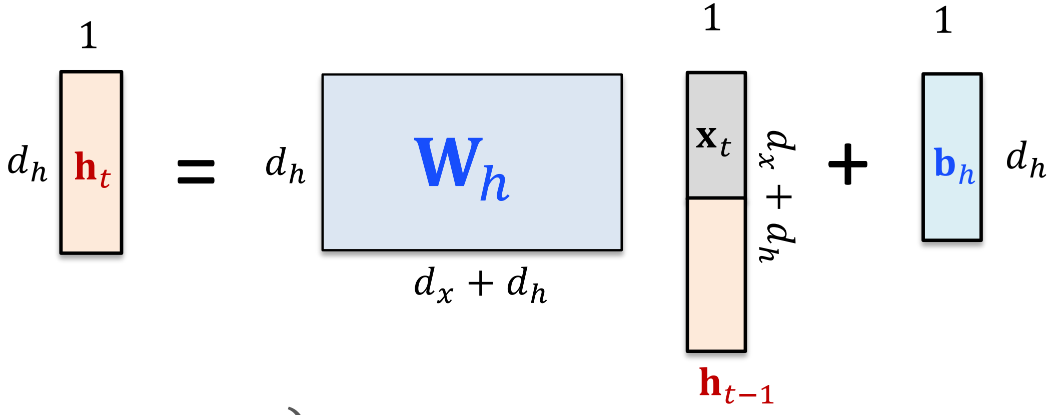
$$\hat{\mathbf{y}}_t = \text{softmax}\left(\mathbf{W}_y \mathbf{h}_t\right)$$

$$\mathbf{h}_t = \tanh(\mathbf{W}_h \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}])$$

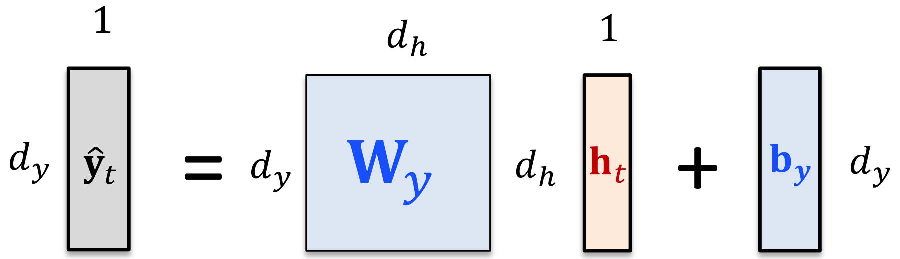
For simplicity, the unfolded RNN diagram omits the bias terms.

Matrix Representations of RNN Computation

- $\mathbf{h}_t = \tanh(\mathbf{W}_h \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_h)$



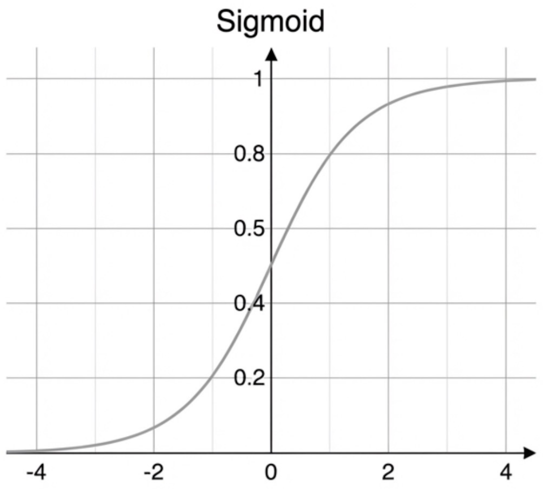
- $\hat{\mathbf{y}}_t = g(\mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y)$

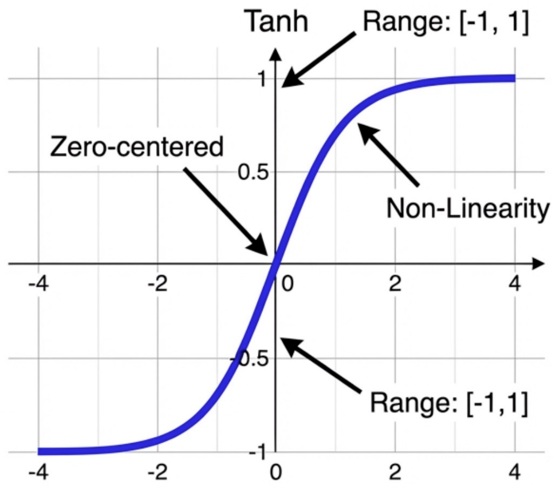


Why tanh Activation Function?

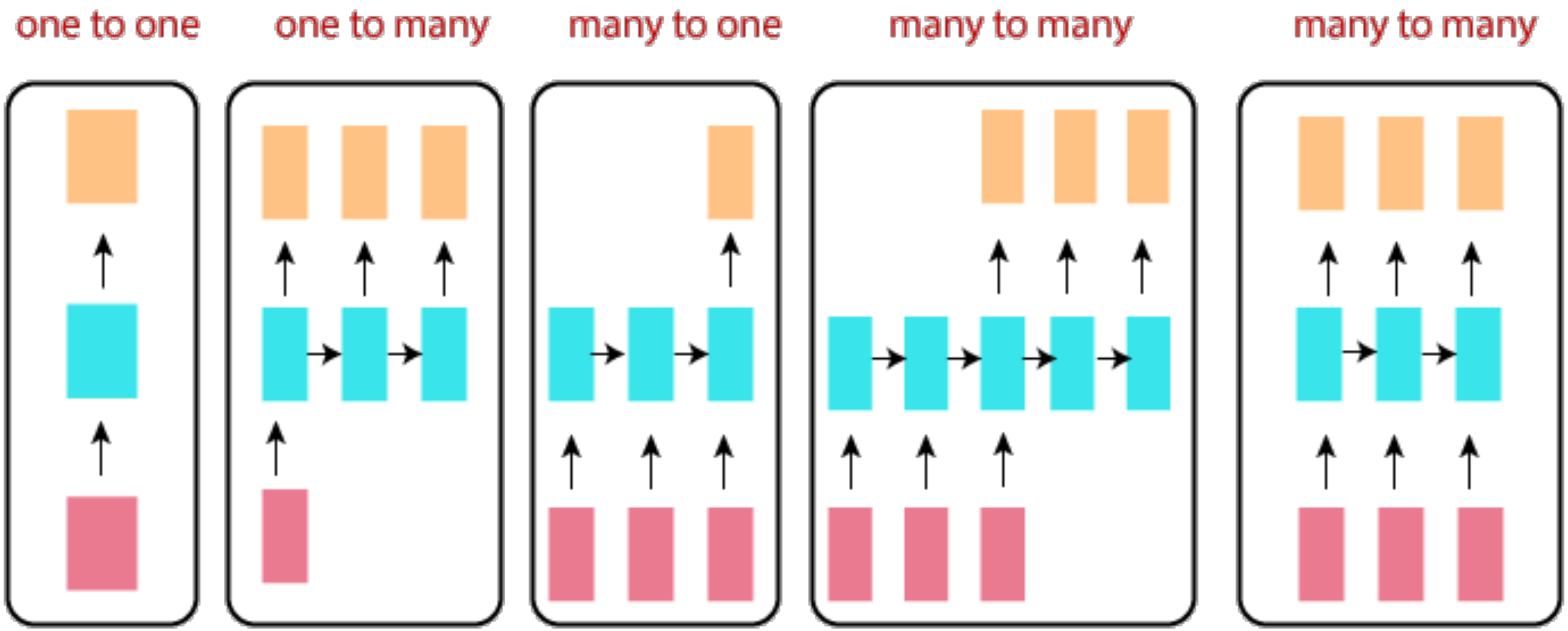
$$\mathbf{h}_t = \tanh(\mathbf{W}_h \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_h)$$

- The hyperbolic tangent function maps inputs to a range of [-1, 1].
- This handles negative values better than a sigmoid and keeps activations normalized, preventing values from exploding immediately.





RNN Architectures



Classification

Image Capture Generation

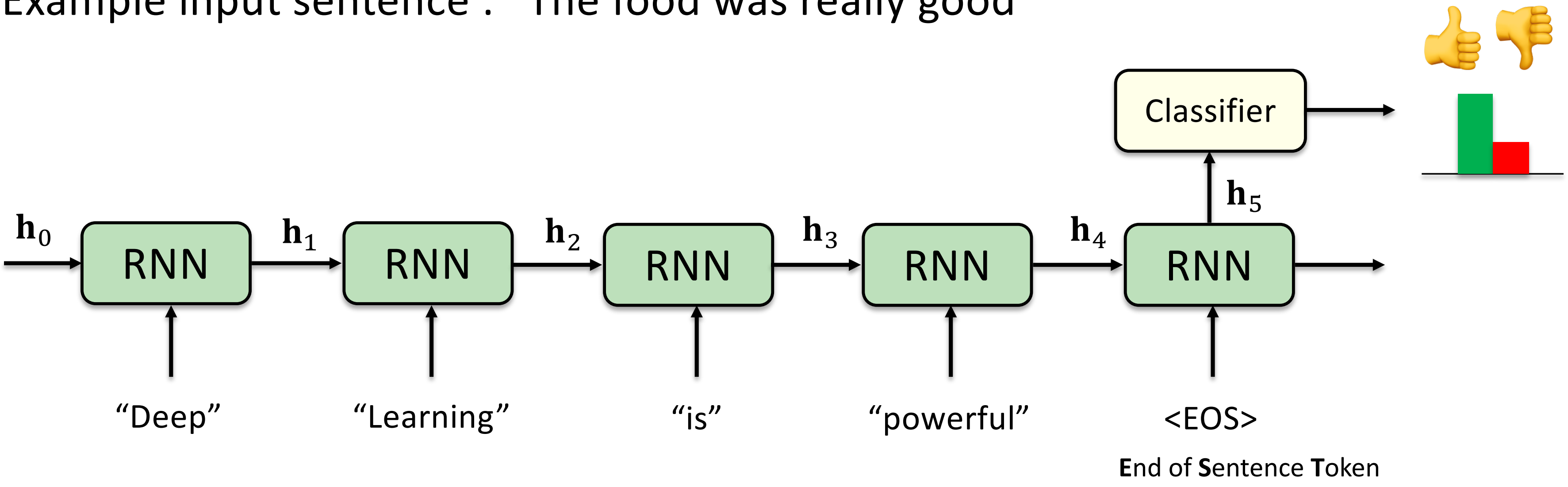
Sentiment Analysis

Machine Translation

Name Entity Recognition

RNN-based Sentiment Analysis (Many to One)

- Classify a restaurant or movie or product review as positive or negative
  - “Deep Learning is very powerful” ➔ Positive
  - “The vacuum cleaner broke within two weeks”. ➔ Negative
  - “The movie had slow parts, but overall was worth watching” ➔ Positive
- The input sentence is variable length while **the output is fixed length**.
- Example input sentence : “The food was really good”



RNN Sentiment Analysis Example

Consider a Vanilla RNN for sentiment analysis processing the input sequence: “**Joyful moments**”.

The RNN uses the following recurrence relation to compute hidden states:

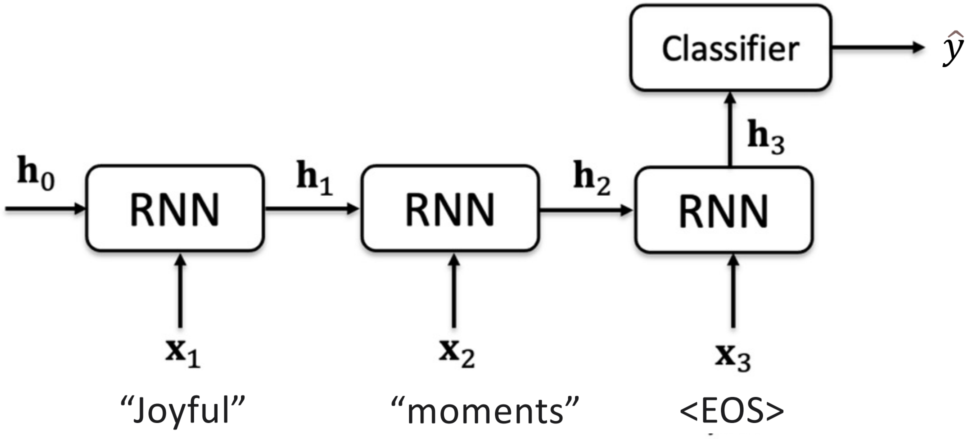
$$\mathbf{h}_t = \tanh(\mathbf{W}_h \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_h)$$

where

- $\mathbf{h}_0 = [0, \quad 0]^T$ is the initial hidden state,
- $\mathbf{W}_h$ is the weight matrix for hidden-to-hidden transitions:

$$\mathbf{W}_h = \begin{bmatrix} 0.3 & 0.7 & 0.1 & 0.7 \\ 0.9 & 0.4 & 0.6 & 0.2 \end{bmatrix}$$

- $\mathbf{b}_h$ is the bias vector: $\mathbf{b}_h = [-0.7, \quad 0.4]^T$
- Word embeddings are given as:
  - ○ “Joyful” → $\mathbf{x}_1 = [0.7, 0.2]^T$
  - ○ “moments” → $\mathbf{x}_2 = [0.1, 0.5]^T$
  - ○ <EOS> → $\mathbf{x}_3 = [0.9, 0.2]^T$



$$\hat{y} = \sigma(\mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y)$$

Negative “0” or Positive “1”

Solution (a)

Compute the hidden states $\mathbf{h}_1$, $\mathbf{h}_2$ and $\mathbf{h}_3$ after processing each word in sequence. Show all intermediate steps.

Given that $\mathbf{h}_0 = [0, \quad 0]^T$

$$\mathbf{h}_1 = \tanh(\mathbf{W}_h \cdot [\mathbf{x}_1, \mathbf{h}_0] + \mathbf{b}_h) = \tanh\left(\begin{bmatrix} 0.3 & 0.7 & 0.1 & 0.7 \\ 0.9 & 0.4 & 0.6 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} 0.7 \\ 0.2 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -0.7 \\ 0.4 \end{bmatrix}\right) = \text{tahh}\left(\begin{bmatrix} -0.35 \\ 1.111 \end{bmatrix}\right) = \begin{bmatrix} -0.337 \\ 0.800 \end{bmatrix}$$

$$\mathbf{h}_2 = \tanh(\mathbf{W}_h \cdot [\mathbf{x}_2, \mathbf{h}_1] + \mathbf{b}_h) = \tanh\left(\begin{bmatrix} 0.3 & 0.7 & 0.1 & 0.7 \\ 0.9 & 0.4 & 0.6 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} 0.1 \\ 0.5 \\ -0.337 \\ 0.800 \end{bmatrix} + \begin{bmatrix} -0.7 \\ 0.4 \end{bmatrix}\right) = \text{tahh}\left(\begin{bmatrix} 0.2063 \\ 0.6478 \end{bmatrix}\right) = \begin{bmatrix} 0.203 \\ 0.570 \end{bmatrix}$$

$$\mathbf{h}_3 = \tanh(\mathbf{W}_h \cdot [\mathbf{3}, \mathbf{h}_2] + \mathbf{b}_h) = \tanh\left(\begin{bmatrix} 0.3 & 0.7 & 0.1 & 0.7 \\ 0.9 & 0.4 & 0.6 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} 0.9 \\ 0.2 \\ 0.203 \\ 0.570 \end{bmatrix} + \begin{bmatrix} -0.7 \\ 0.4 \end{bmatrix}\right) = \text{tahh}\left(\begin{bmatrix} -0.1293 \\ 1.5258 \end{bmatrix}\right) = \begin{bmatrix} 0.128 \\ 0.909 \end{bmatrix}$$

Solution (b)

The final sentiment prediction is computed using a binary classifier with a sigmoid activation function:

$$\hat{y} = \sigma\left(\mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y\right)$$

where $\mathbf{W}_y = [0.4, \quad 0.9]$ and $\mathbf{b}_y = [0.6]$

Compute the predicted output $\hat{y}$ and determine whether the sentiment is classified as **Negative (0) or Positive (1) based on the standard threshold of 0.5.**

==

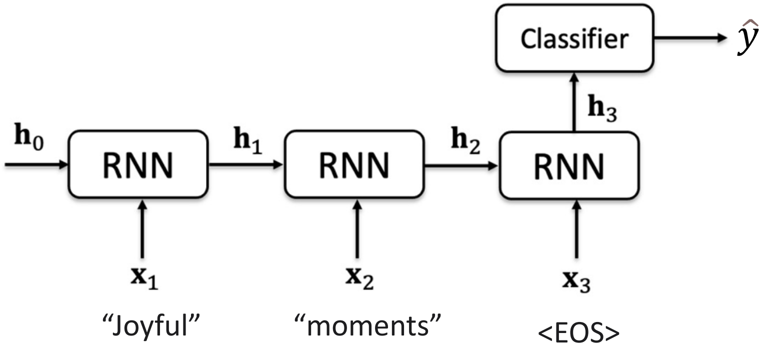
Compute predicted output $\hat{y}$ with Final classifier

$$\hat{y} = \sigma\left(\mathbf{W}_y \mathbf{h}_3 + \mathbf{b}_y\right) = \sigma\left(\begin{bmatrix}0.4 & 0.9\end{bmatrix}\begin{bmatrix}0.128\\0.909\end{bmatrix} + [0.6]\right) = \sigma(1.4693)$$

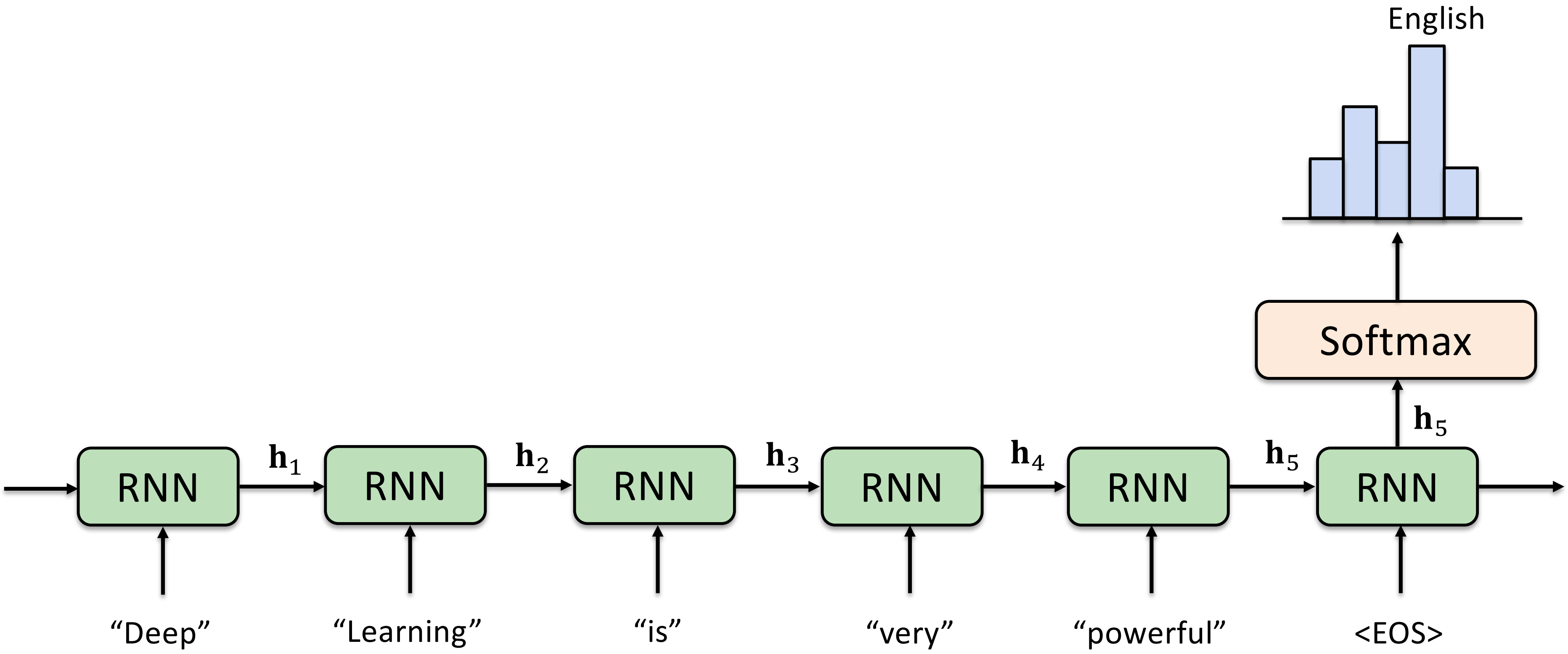
$$= \frac{1}{1 + e^{-1.4593}} = 0.813$$

Predicted output: $\hat{y} = 0.813$.

Since $\hat{y} = 0.813 > 0.5$, **classify as Positive (1).**

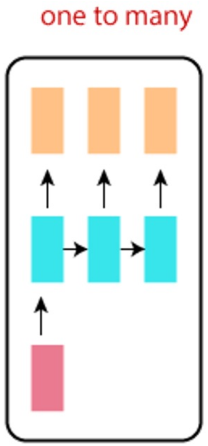


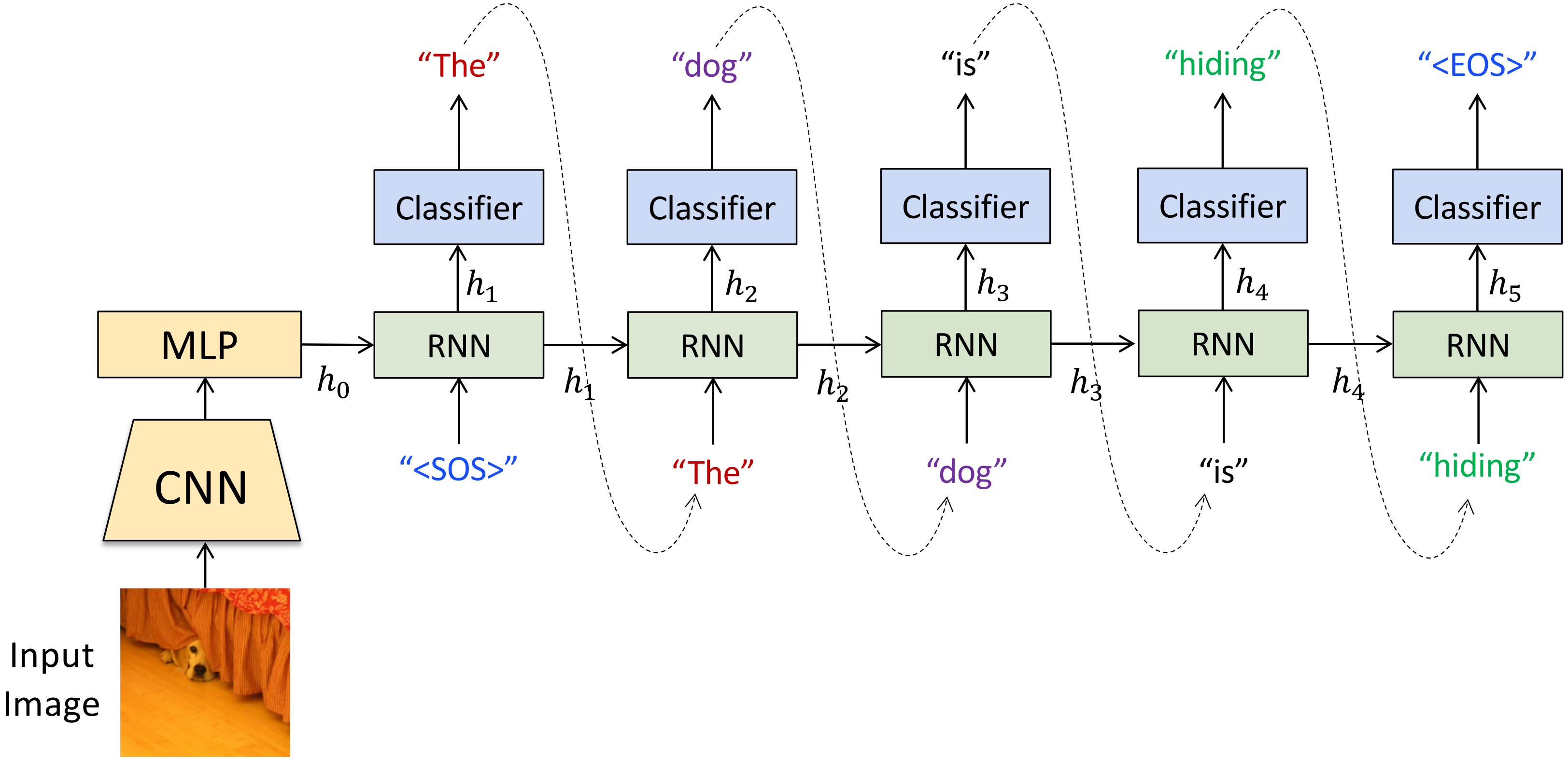
Language Detection (Many to One)



RNN-based Image Captioning

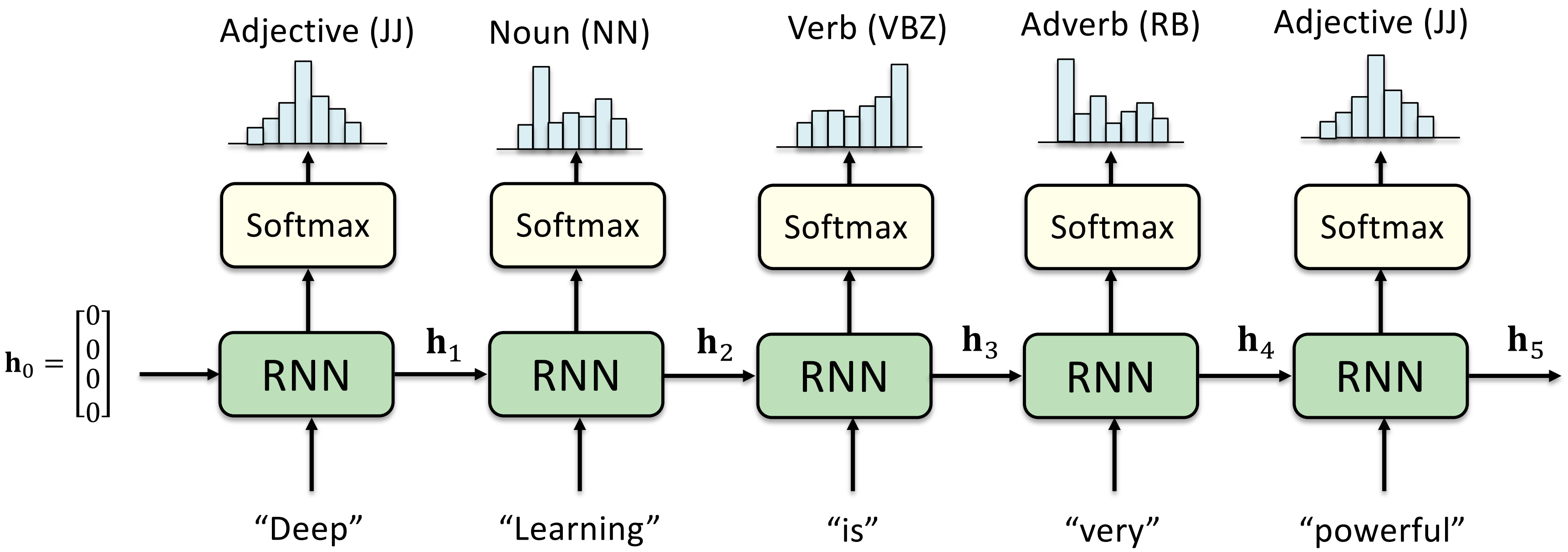
Input is a single image, and output is a sequence of text in variable length.



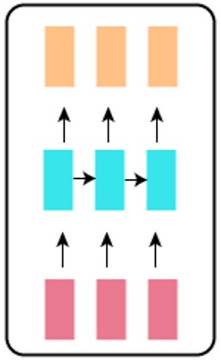


Part-of-Speech (POS) Tagging

POS tagging assigns grammatical labels (e.g., noun, verb) to words in a sentence. RNNs excel at this task by capturing syntactic relationships through sequential processing. Their hidden state encodes grammatical context, enabling accurate POS tag predictions for each word.

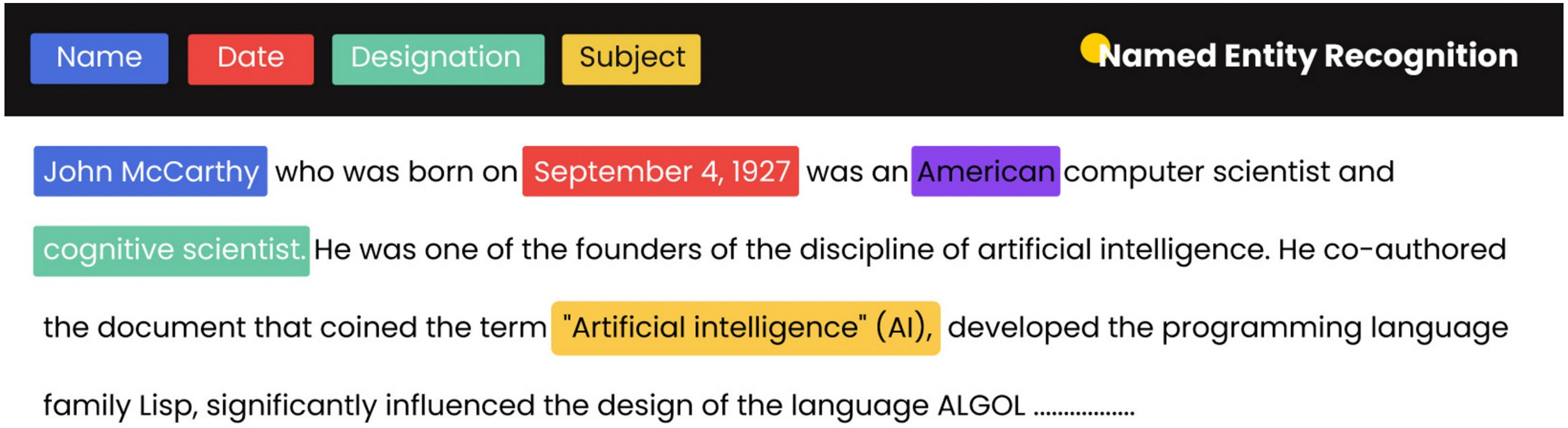


many to many



Named Entity Recognition (NER)

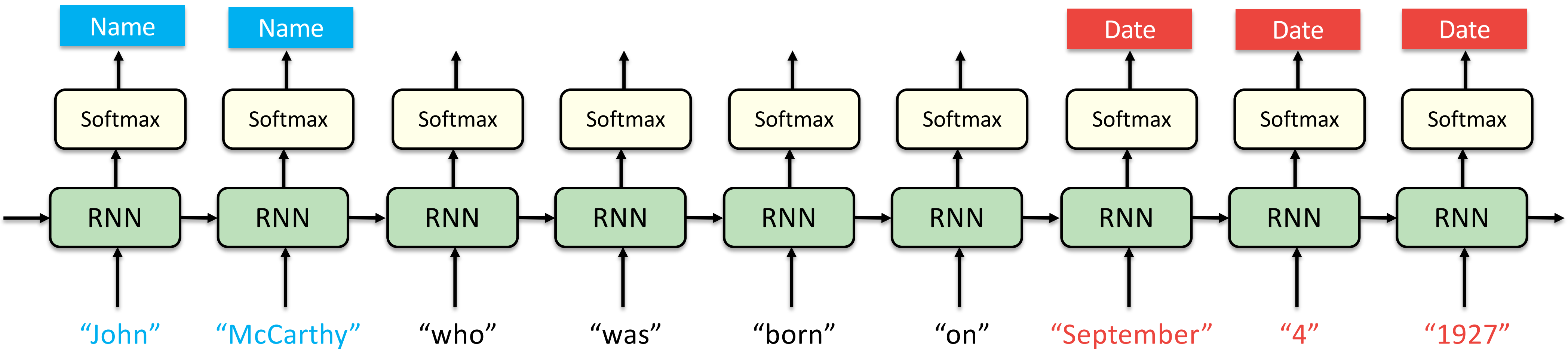
- **Named Entity Recognition (NER)** is an NLP technique that **identifies and classifies key entities in text**, such as names of people, dates, organizations, locations, designation, subject, and other predefined categories.



John McCarthy who was born on September 4, 1927 was an American computer scientist and cognitive scientist. He was one of the founders of the discipline of artificial intelligence. He co-authored the document that coined the term "Artificial intelligence" (AI), developed the programming language family Lisp, significantly influenced the design of the language ALGOL ....

RNN-based NER

- NER can be implemented by many-to-many RNN architecture with **aligned sequential pairs**

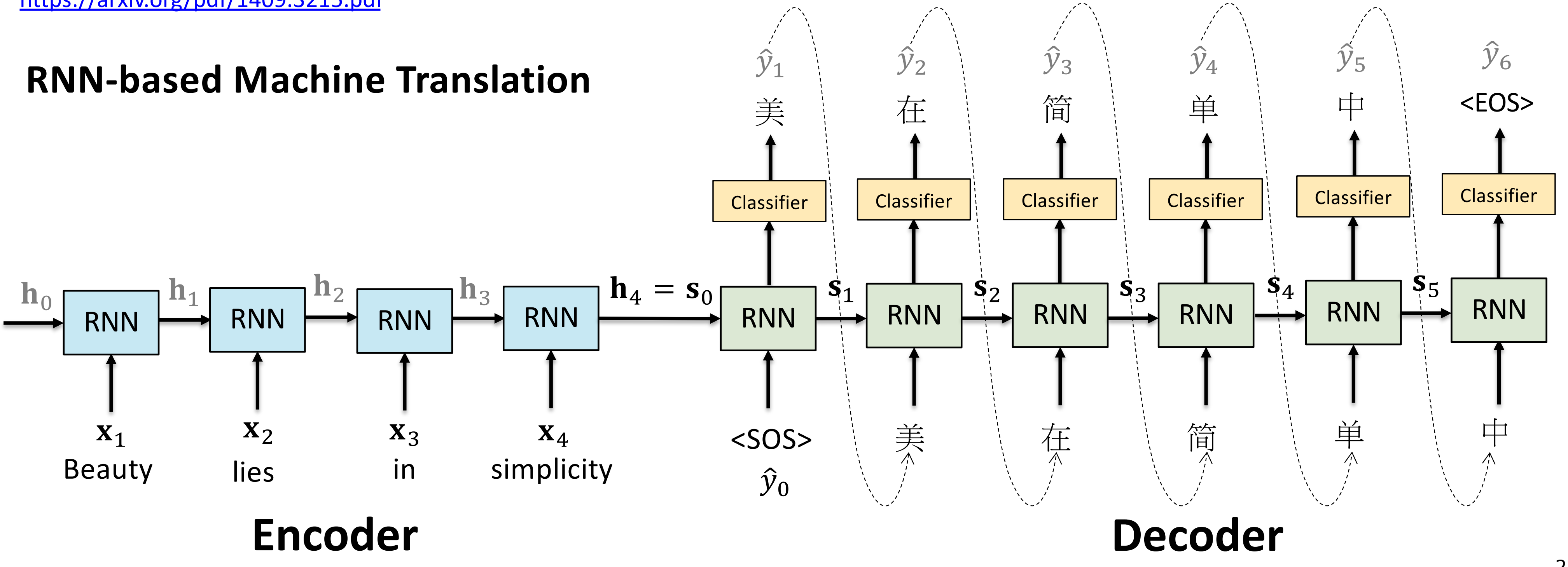


Many-to-Many Unaligned Sequential Paris

RNNs are used in language modeling to **predict the next word's probability distribution based on previous words**. By updating their hidden state, they capture sentence context, aiding tasks like autocomplete by suggesting likely next words from user input history.

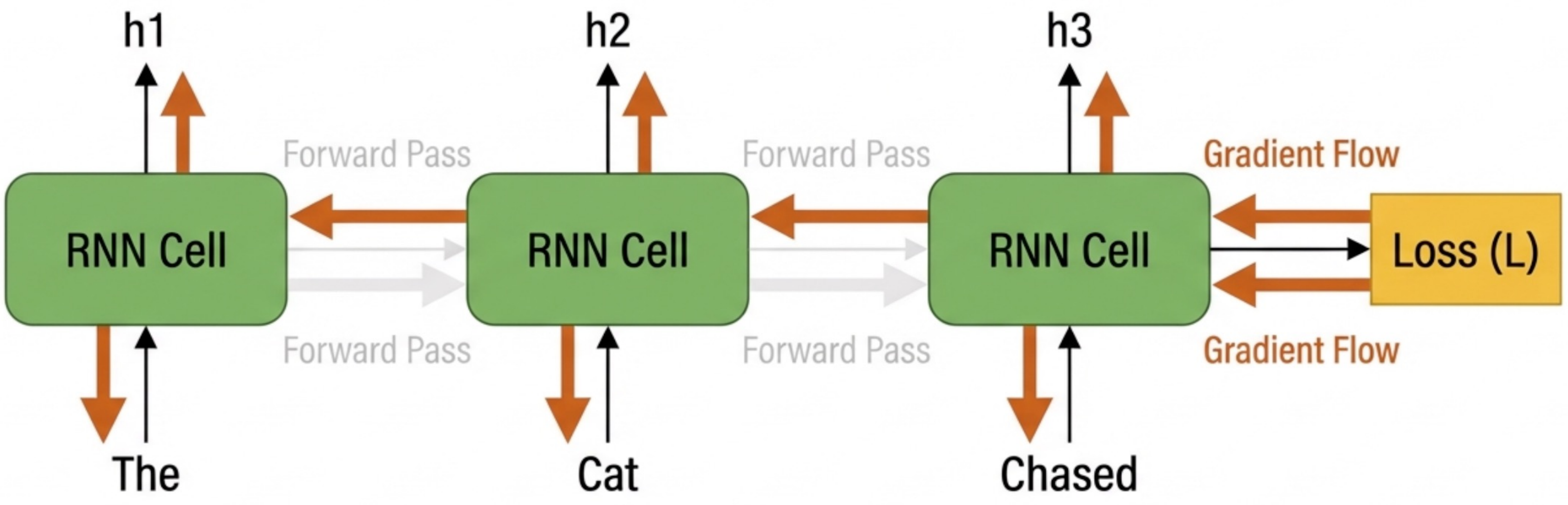
https://arxiv.org/pdf/1409.3215.pdf

RNN-based Machine Translation



RNN Training:
Back-Propagation Through Time
(BPTT)

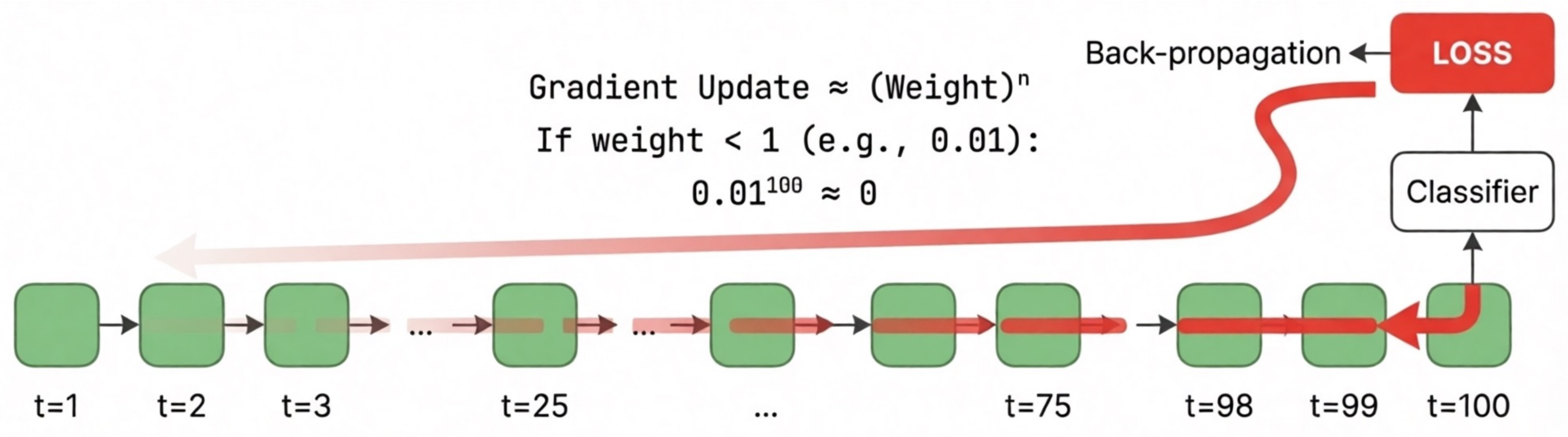
RNN Training via Back-Propagation Through Time (BPTT)



Gradients must travel backward through every time step to update the first weights.

The Fatal Flaw: Vanishing Gradients

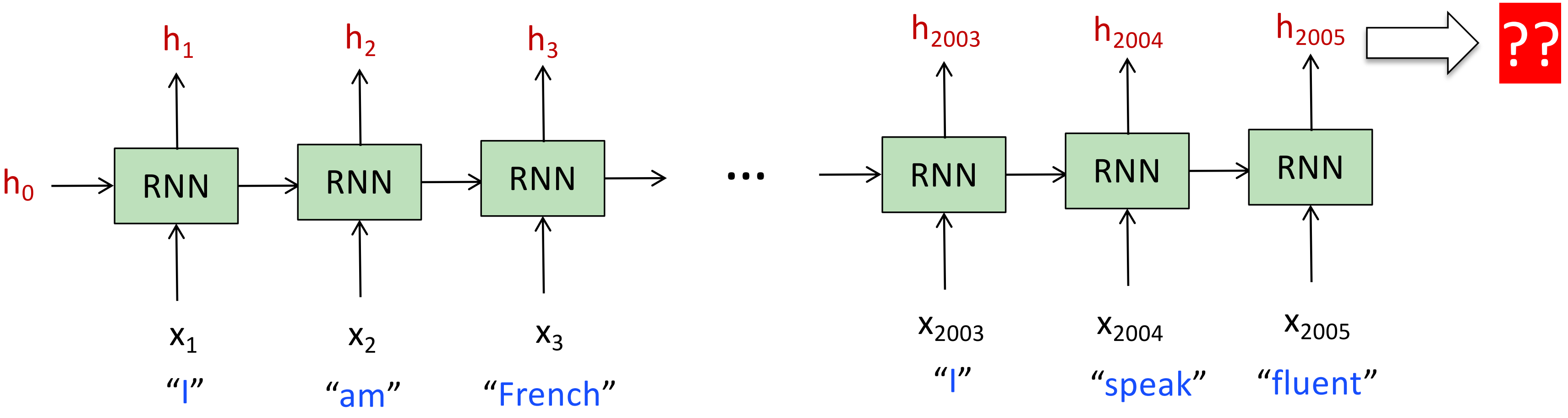
Why vanilla RNNs cannot learn long-term dependencies.



As the gradient **shrinks to zero**, the network stops learning from the beginning of the sequence. It effectively suffers from amnesia.

Long-Range Dependencies Problem

- If we want to predict the last word in a **short sentence** "The grass is green", that's **totally doable using simple RNN**.
- But if we want to predict the last word in a **long sentence** "I am French (2000 words later) I speak fluent French". We need to be able to remember **long range dependencies**. RNN's are bad at this. They forget the long term past easily.



RNN is **difficult to capture very long-term dependencies** due to the Gradient exponentially decays as it backpropagated.

Exploding Gradient Problem

What if weights are high?

- Could lead to the exploding gradients problem
- This, however, is not much of a problem.
  - Will show up as **NaN** during implementation
    - "NaN" stands for "Not a Number."
    - It is a special floating-point value used to represent undefined or unrepresentable results of mathematical operations, particularly when calculations involve invalid or infinite values.
  - **Gradient clipping works!**

How to Tackle these Weakness of RNNs?

- **Change activation functions:**
  - The problem can be alleviated somewhat but not eliminated
  - May affect the learning performance
- **Better Solutions: Advanced RNN Architectures**
  - **LSTM (Long Short-Term Memory) - 1997**
  - **GRU (Gated Recurrent Units) - 2015**

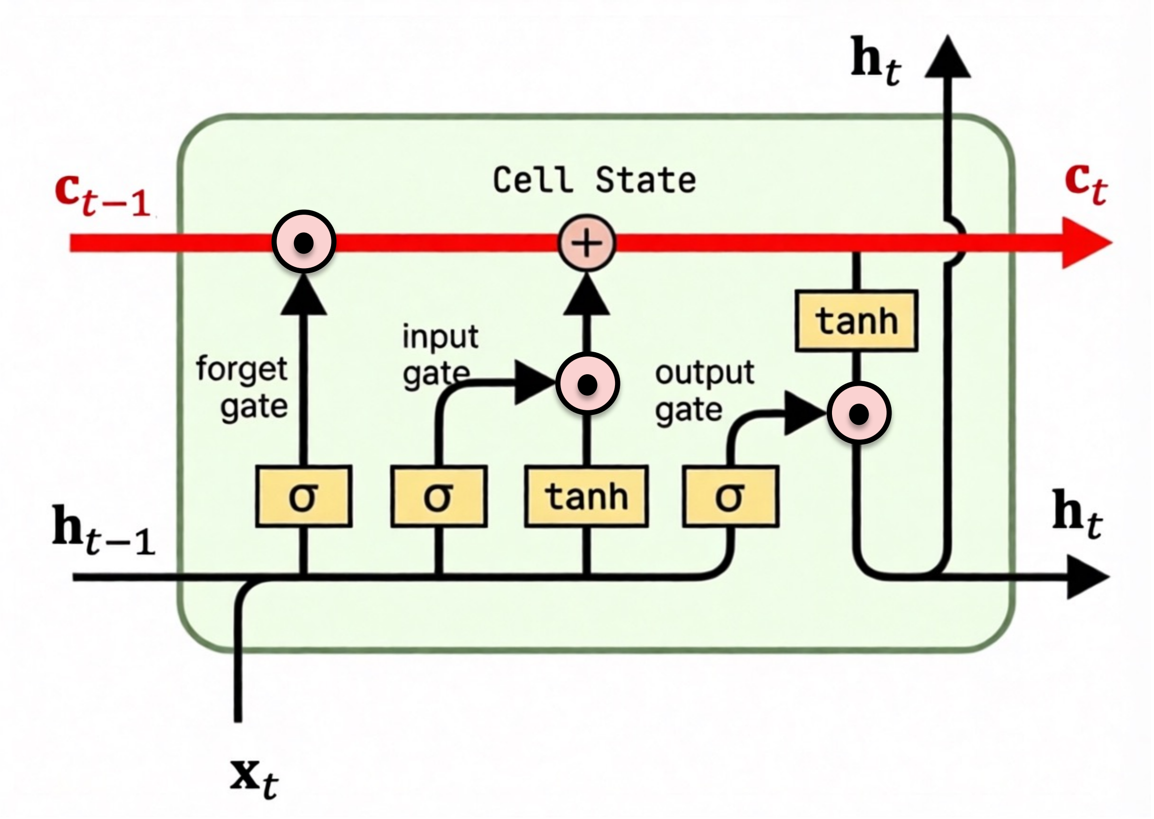
Long Short-Term Memory (LSTM, 1997)

The Long Short-Term Memory (LSTM) recurrent neural network architecture was introduced in 1997 in the paper "Long Short-Term Memory" by Sepp Hochreiter and Jürgen Schmidhuber.

Memory Upgrade: Long Short-Term Memory (LSTM)

Solving the Vanishing Gradient with a "Superhighway" for memory.

- Proposed in 1997 to solve the vanishing gradient problem.
- The innovation is the **"Cell State"**—a superhighway that allows information to flow largel unchanged.



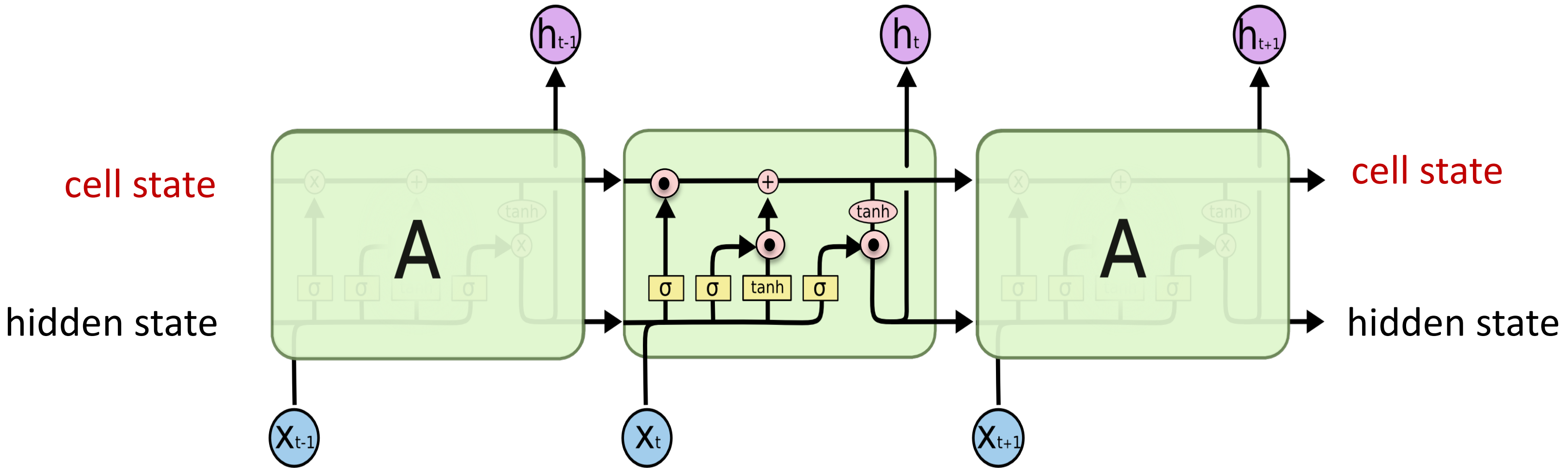
Cell State $\mathbf{c}_t$ vs Hidden State $\mathbf{h}_t$

- The cell state $\mathbf{c}_t$ **contains essential information regarding context and historical patterns, basically memory**. It runs through the cell and can be adjusted by several so-called gates with linear interactions.
- It's common to mix up the cell state $\mathbf{c}_t$ and hidden state $\mathbf{h}_t$, but generally:
  - The **cell state** is designed to hold the network's **long-term** memory.
  - The **hidden state** captures **short-term** dependencies and mainly retains recent information. It is also utilized in the cell's output for prediction.

https://towardsdatascience.com/long-short-term-memory-lstm-improving-rnns-40323d1c05f8

LSTM: Architecture

- The crucial aspect of LSTMs is the **cell state $\mathbf{c}_t$**, which **traverses from the input to the output of a cell**, acting as the long-term memory component of the LSTM.



cell state

hidden state

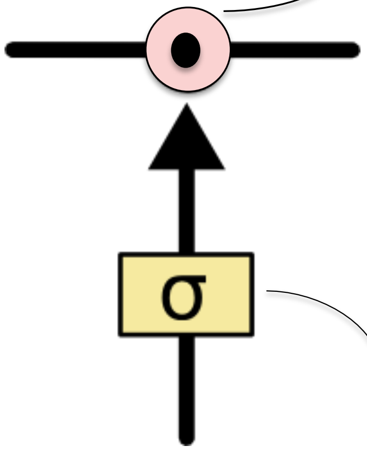
cell state

hidden state

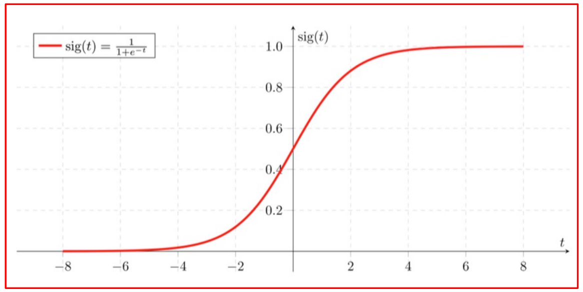
S. Hochreiter and J. Schmidhuber, Long short-term memory, Neural Computation 9 (8), pp. 1735–1780, 1997

Gates

- A neural network σ
- Output range [0, 1]
- Pointwise multiplication
- Filtering function

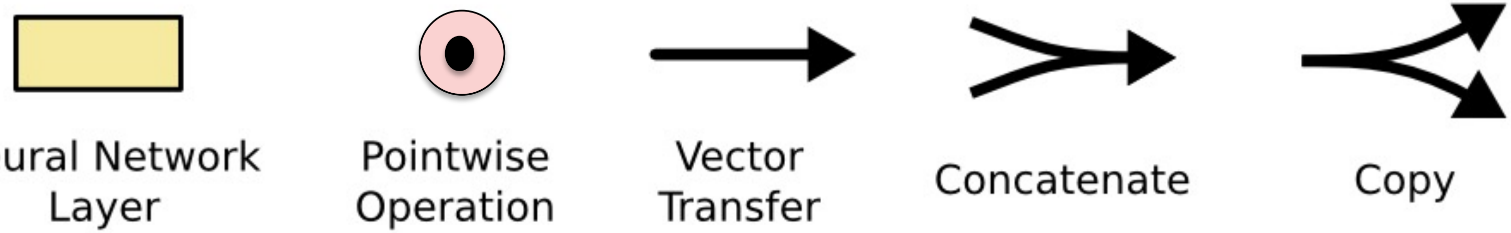


Pointwise multiplication of 2 vectors

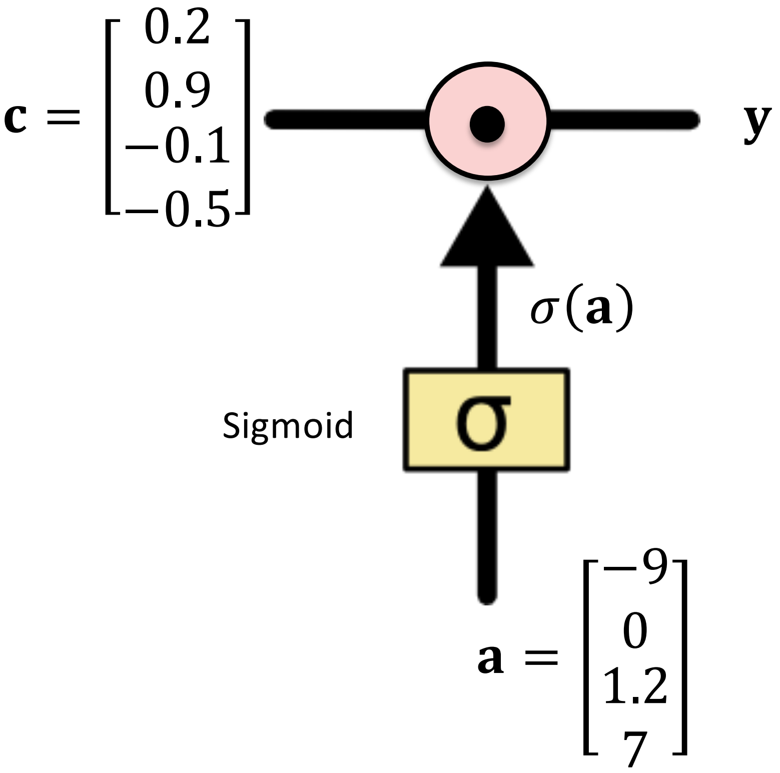


Sigmoid function: between 0 and 1.

$\odot$



Sigmoid and Pointwise Multiplication


$\mathbf{c} = \begin{bmatrix} 0.2 \\ 0.9 \\ -0.1 \\ -0.5 \end{bmatrix}$
$\mathbf{y}$
$\sigma(\mathbf{a})$
Sigmoid
σ
$\mathbf{a} = \begin{bmatrix} -9 \\ 0 \\ 1.2 \\ 7 \end{bmatrix}$

$$\sigma(\mathbf{a}) = \sigma\left(\begin{bmatrix} -9 \\ 0 \\ 1.2 \\ 7 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0.5 \\ 0.8 \\ 1 \end{bmatrix}$$

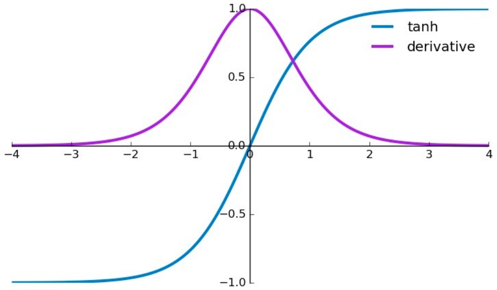
$$\mathbf{y} = \mathbf{c} \odot \sigma(\mathbf{a}) = \begin{bmatrix} 0.2 \\ 0.9 \\ -0.1 \\ -0.5 \end{bmatrix} \odot \begin{bmatrix} 0 \\ 0.5 \\ 0.8 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.45 \\ -0.08 \\ -0.5 \end{bmatrix}$$

The sigmoid layer outputs numbers between 0-1 determine how much each component should be let through using the **x gate** as pointwise multiplication.

Sigmoid vs. Tanh

- **Why sigmoid?** σ
  - Sigmoid can output 0 to 1, it can be used to forget or remember the information.

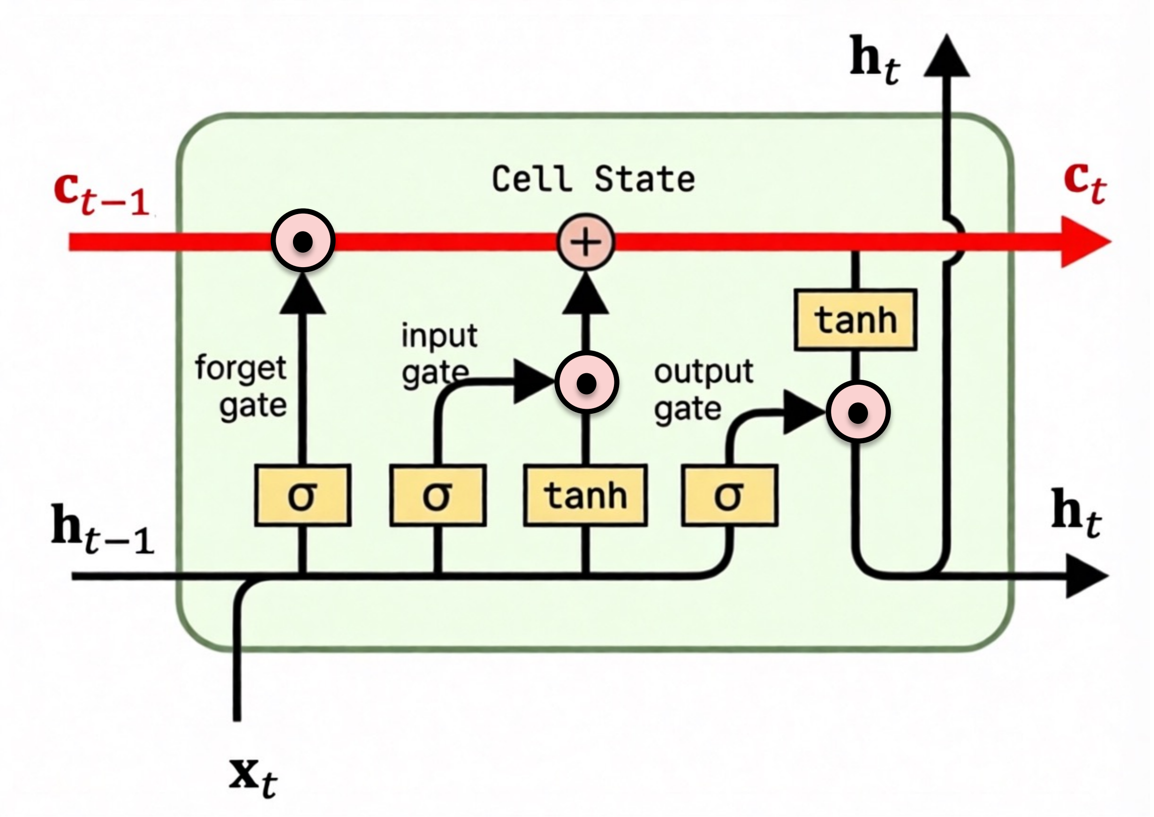
- **Why tanh?** tanh
  - To overcome the vanishing gradient problem.
  - tanh’s second derivative can sustain for a long range before going to zero.



Large tanh activations will give small gradients

LSTM Cell

- The cell state $\mathbf{c}_t$ enables information to flow throughout the entire LSTM chain with minimal linear operations.
- Three gates (**forget gate**, **input gate**, and **output gate**) in the LSTM act as filters.
    - These gates control the flow of information and decide which information is retained or discarded.

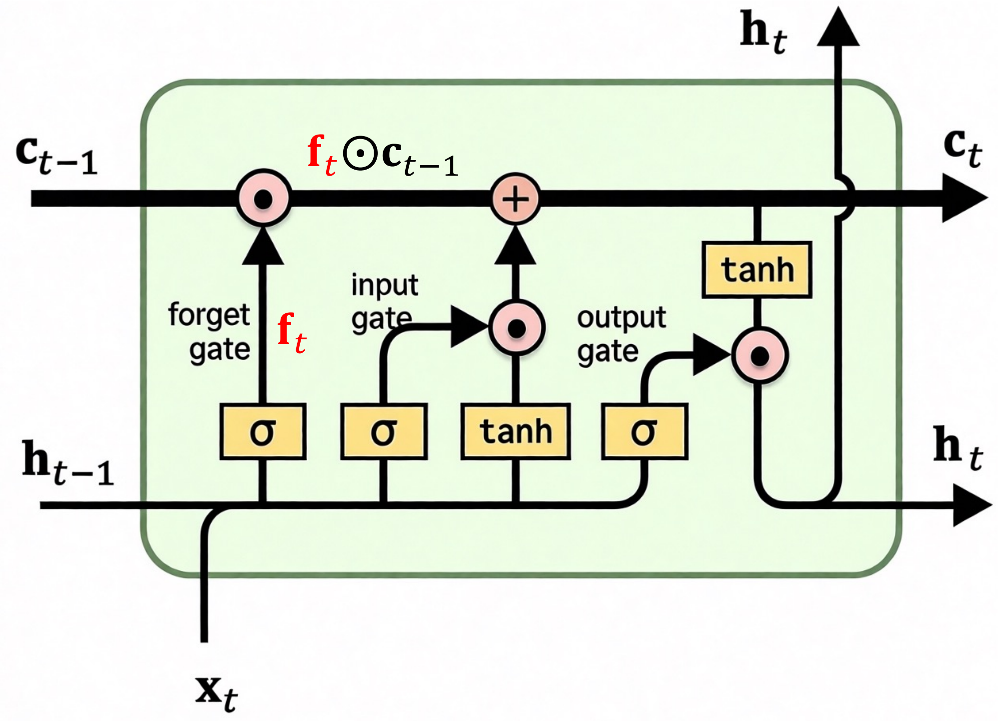


LSTM: Forget Gates

Decides what information to discard (0) or keep (1)

$$\mathbf{f}_t = \sigma\left(\mathbf{W}_f \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_f\right)$$

- $\mathbf{f}_t$ outputs a number between 0 and 1 for each number in the cell state $\mathbf{c}_{t-1}$.
- Zero to completely forget and one to keep all information from $\mathbf{c}_{t-1}$.



LSTM: Input Gates

Decides what new information to store in the cell state.

$$\mathbf{i}_t = \sigma(\mathbf{W}_i \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_i)$$

Sigmoid layer decides which values are updated.

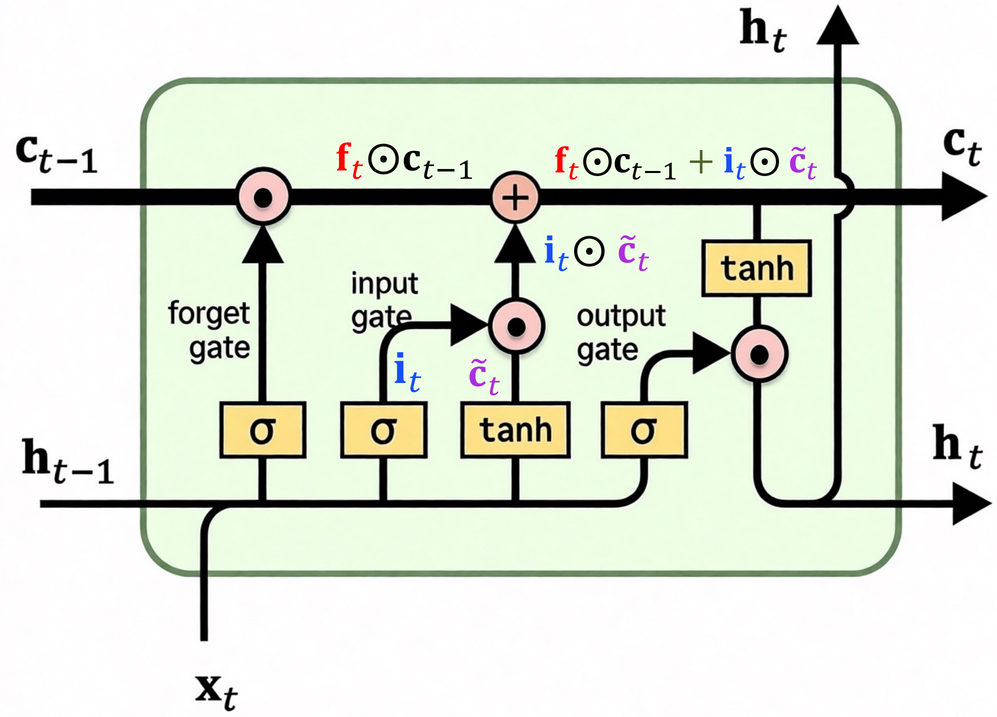
$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_c)$$

tanh layer gives weights to the values to be added to the state.

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$

Cell State

Internal Cell State



LSTM: Output Gates

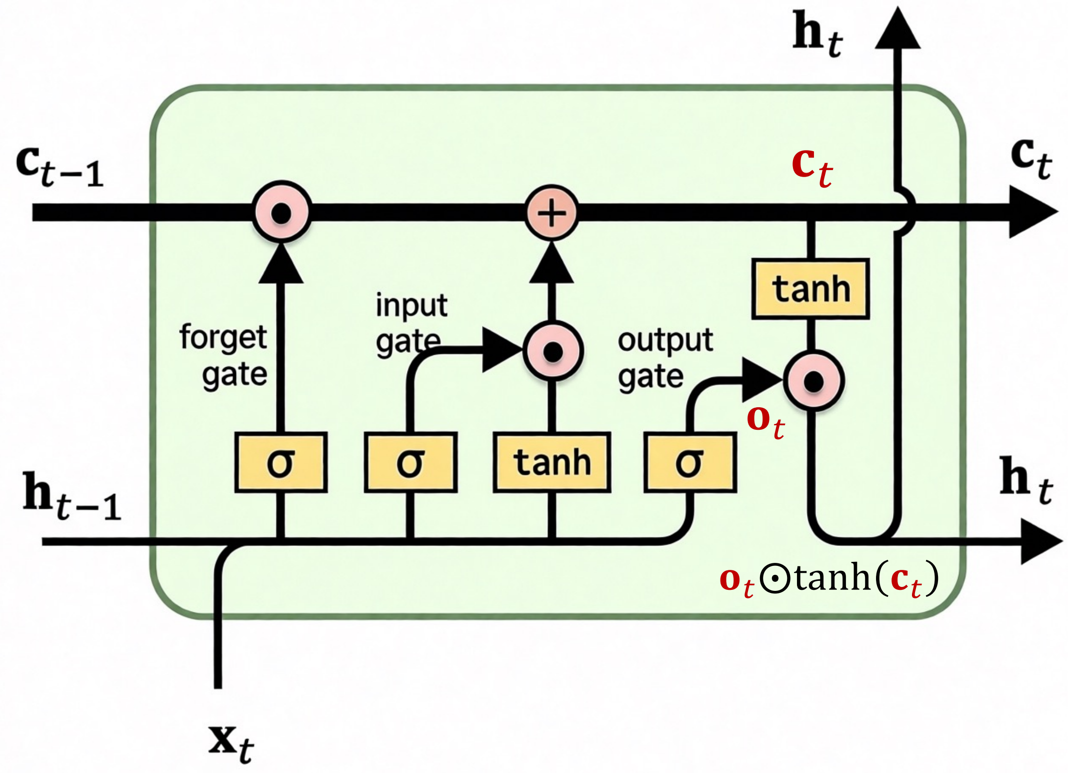
Filters the cell state to produce the final hidden state output.

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_o)$$

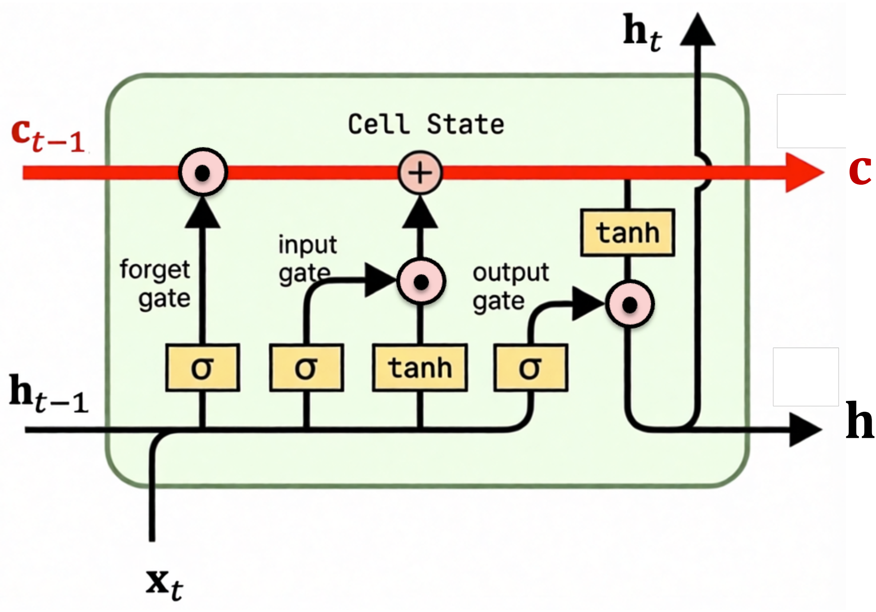
Sigmoid layer decides which part of cell state is selected for output.

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

Tanh layer gives weights to the values (-1 to 1]



LSTM Cell



Forget Gate

$$\mathbf{f}_t = \sigma\left(\mathbf{W}_f \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_f\right)$$

Input Gate

$$\mathbf{i}_t = \sigma(\mathbf{W}_i \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_i)$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \underbrace{\tanh(\mathbf{W}_c \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_c)}_{\tilde{\mathbf{c}}_t}$$

Internal Cell State

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_o)$$

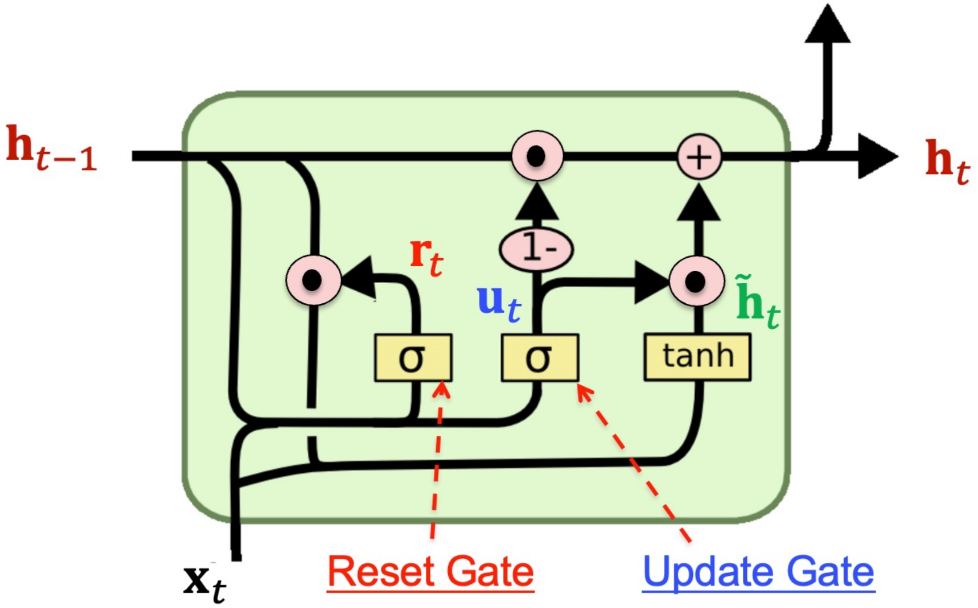
Output Gate

Gated Recurrent Unit (GRU, 2014)

The Gated Recurrent Unit (GRU) was introduced in 2014 in the paper "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation" by Kyunghyun Cho et al.

The Gated Recurrent Unit (GRU, 2014)

Introduced in 2014, the GRU simplifies the LSTM by combining gates and merging the cell/hidden states. It is faster to compute with often comparable performance.



https://towardsdatascience.com/a-brief-introduction-to-recurrent-neural-networks-638f64a61ff4

Gated Recurrent Units (GRU)

- The **Reset Gate** is responsible for the short-term memory as it decides how much past information is kept and disregarded.

$$\mathbf{r}_t = \sigma(\mathbf{W}_r \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_r)$$

- The **Update Gate**, in contrast, is responsible for the long-term memory and is comparable to the LSTM’s forget gate.

$$\mathbf{u}_t = \sigma(\mathbf{W}_u \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_u)$$

- **The influence of the previous hidden state on the candidate hidden state is controlled by the reset gate**

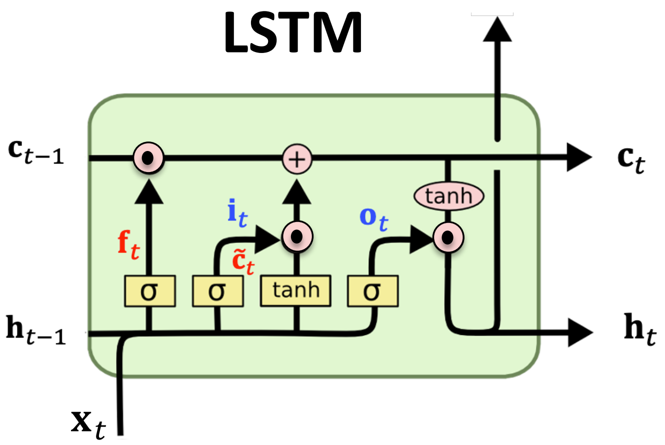
$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_h \cdot [\mathbf{x}_t, \mathbf{r}_t \odot \mathbf{h}_{t-1}] + \mathbf{b}_h)$$

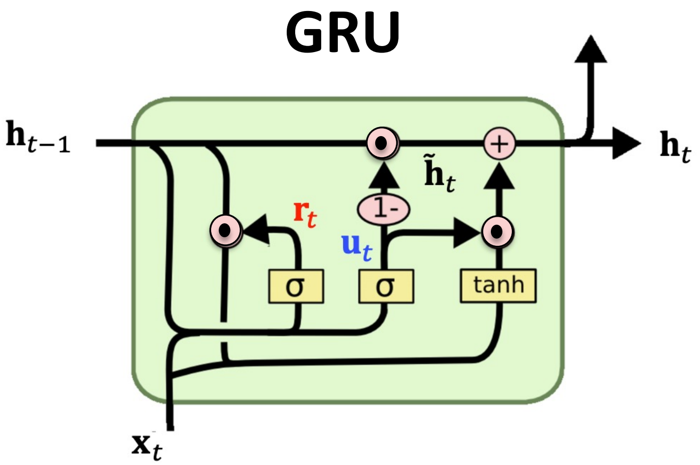
- The previous hidden state $\mathbf{h}_{t-1}$ and the candidate hidden state $\tilde{\mathbf{h}}_t$ **are combined** by the update gate.

$$\mathbf{h}_t = \mathbf{u}_t \odot \tilde{\mathbf{h}}_t + (1 - \mathbf{u}_t) \odot \mathbf{h}_{t-1}$$

LSTM vs GRU

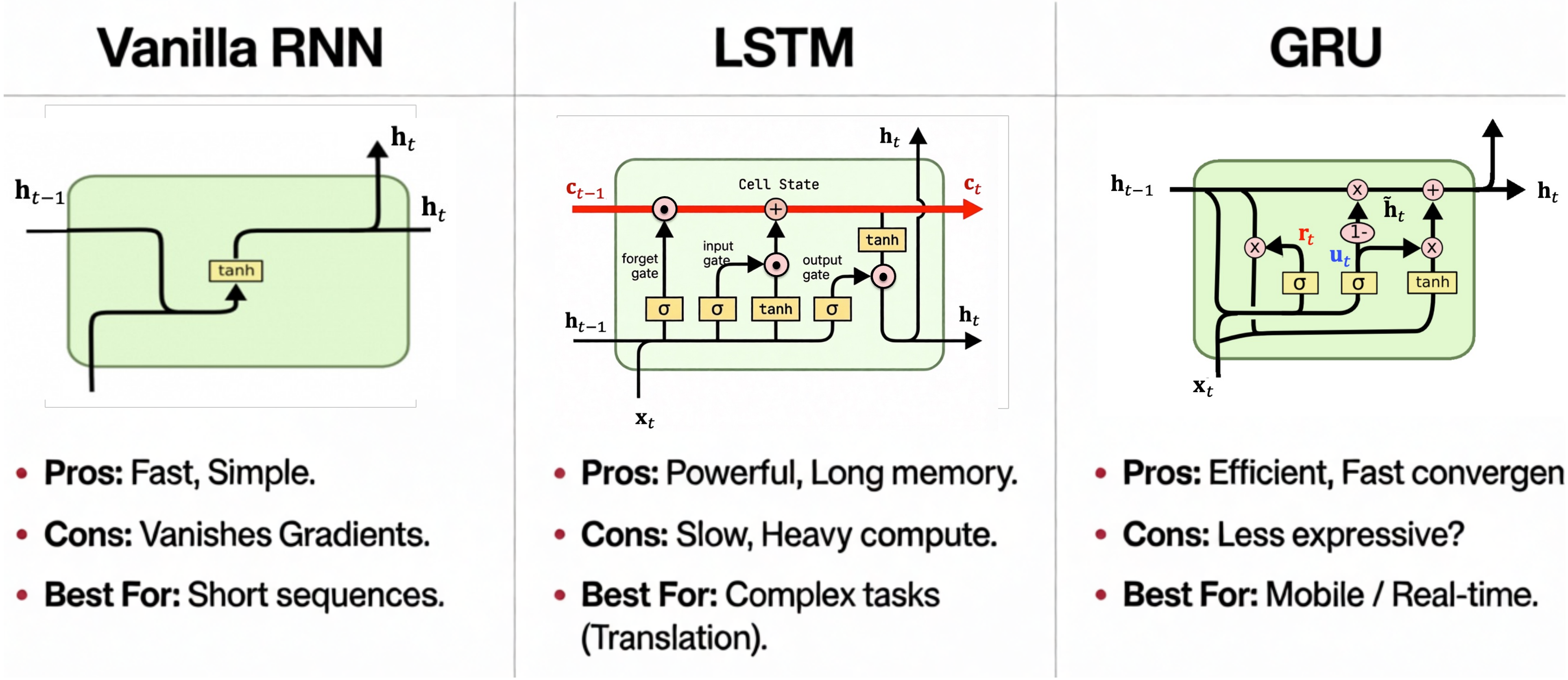
- GRUs also takes $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ as inputs. They perform some calculations and then pass along $\mathbf{h}_t$. What makes them different from LSTMs is that GRUs don't need the cell layer to pass values along. The calculations within each iteration ensure that the $\mathbf{h}_t$ values being passed along **either retain a high amount of old information or are jump-started with a high amount of new information**.





https://towardsdatascience.com/a-brief-introduction-to-recurrent-neural-networks-638f64a61ff4

Architecture Showdown: RNN vs. LSTM vs. GRU



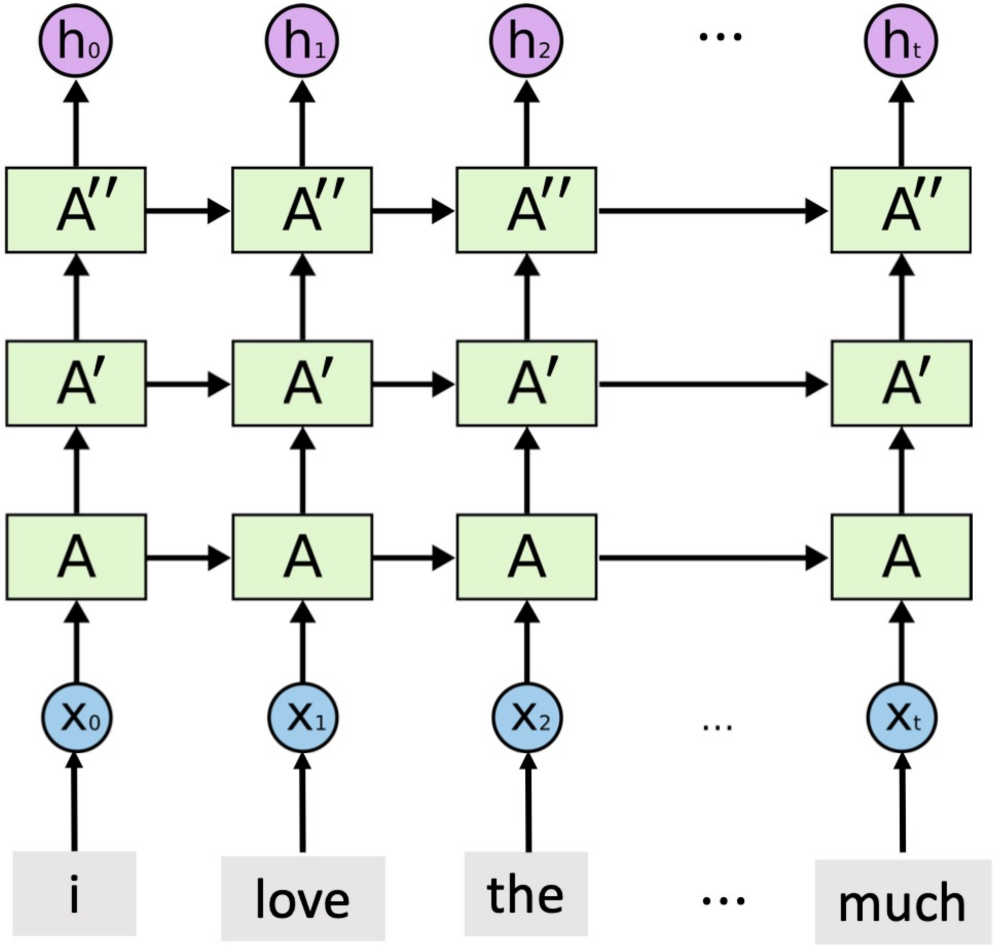
Vanilla RNN	LSTM	GRU
Pros: Fast, Simple.	**Pros:** Powerful, Long memory.	**Pros:** Efficient, Fast convergen
Cons: Vanishes Gradients.	**Cons:** Slow, Heavy compute.	**Cons:** Less expressive?
Best For: Short sequences.	**Best For:** Complex tasks (Translation).	**Best For:** Mobile / Real-time.

Stacked RNN

- **Multi-layer RNN architecture** with multiple hidden layers
- **Forward Pass:**
  - For a stack of L Layers:

$$\mathbf{h}_t^{(l)} = \mathrm{RNN}\left(\mathbf{h}_t^{(l-1)}, \mathbf{h}_{t-1}^{(l)}\right)$$

where $h_t^{(l)}$ is the hidden state at layer l and time step t.



https://towardsdatascience.com/understanding-bidirectional-rnn-in-pytorch-5bd25a5dd66

Bidirectional RNN

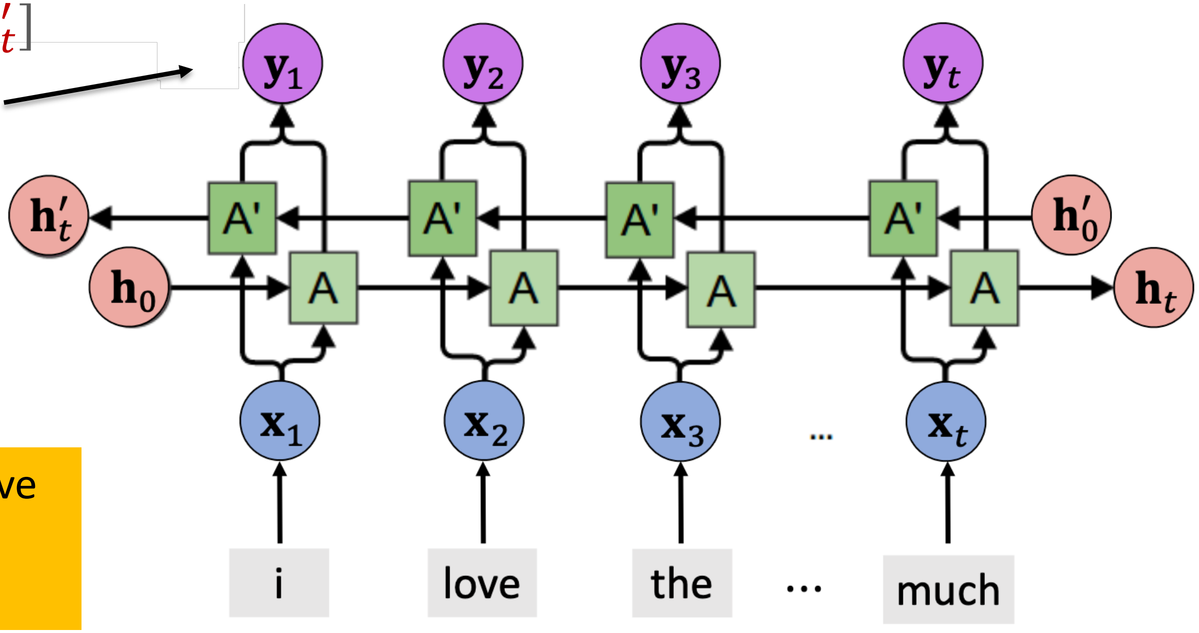
- RNNs can process the input sequence in forward and in the reverse directions
  - **Concatenate the 2 states** $[\mathbf{h}_t, \mathbf{h}'_t]$
  - Passed to the upper RNN layer

$$\mathbf{h}_t = \tanh(\mathbf{W}_h \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_h)$$

$$\mathbf{h}'_t = \tanh(\mathbf{W}'_h \cdot [\mathbf{x}_t, \mathbf{h}'_{t+1}] + \mathbf{b}'_h)$$

$$\mathbf{y}_t = g\big(\mathbf{W}_y[\mathbf{h}_t, \mathbf{h}'_t] + \mathbf{b}_y\big)$$

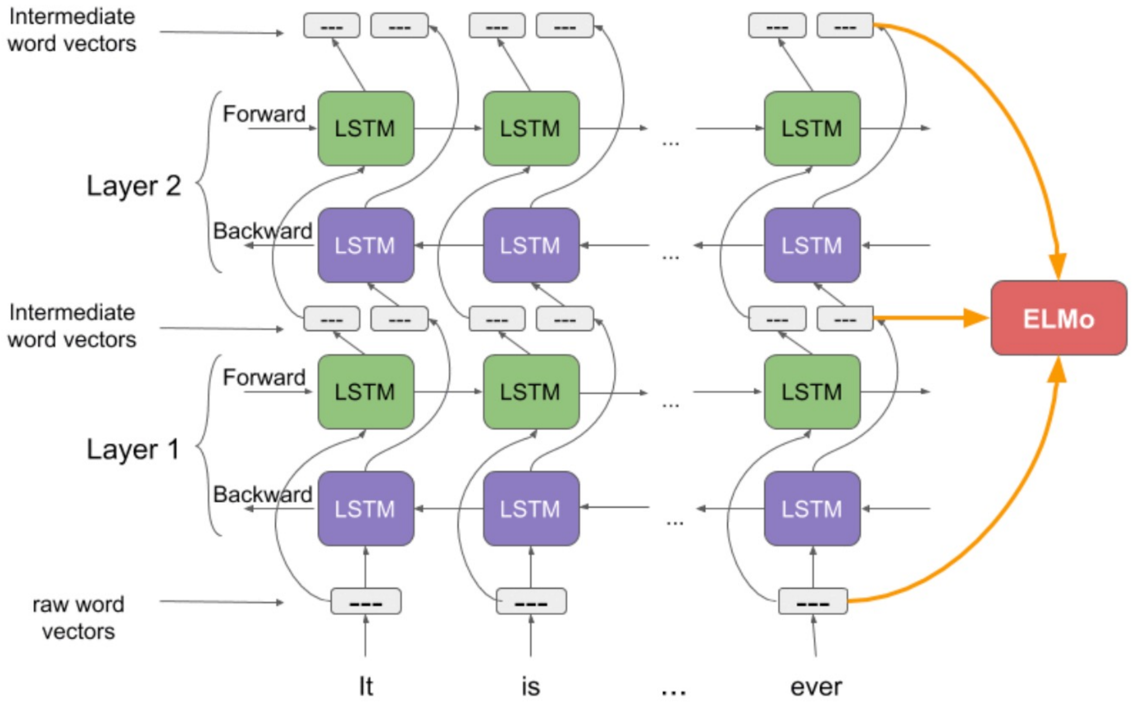
This structure allows the networks to have both backward and forward information about the sequence at every time step.



https://towardsdatascience.com/understanding-bidirectional-rnn-in-pytorch-5bd25a5dd66

Scaling Up: Deep and Bidirectional Architectures

Going deeper and processing contexts in both directions.



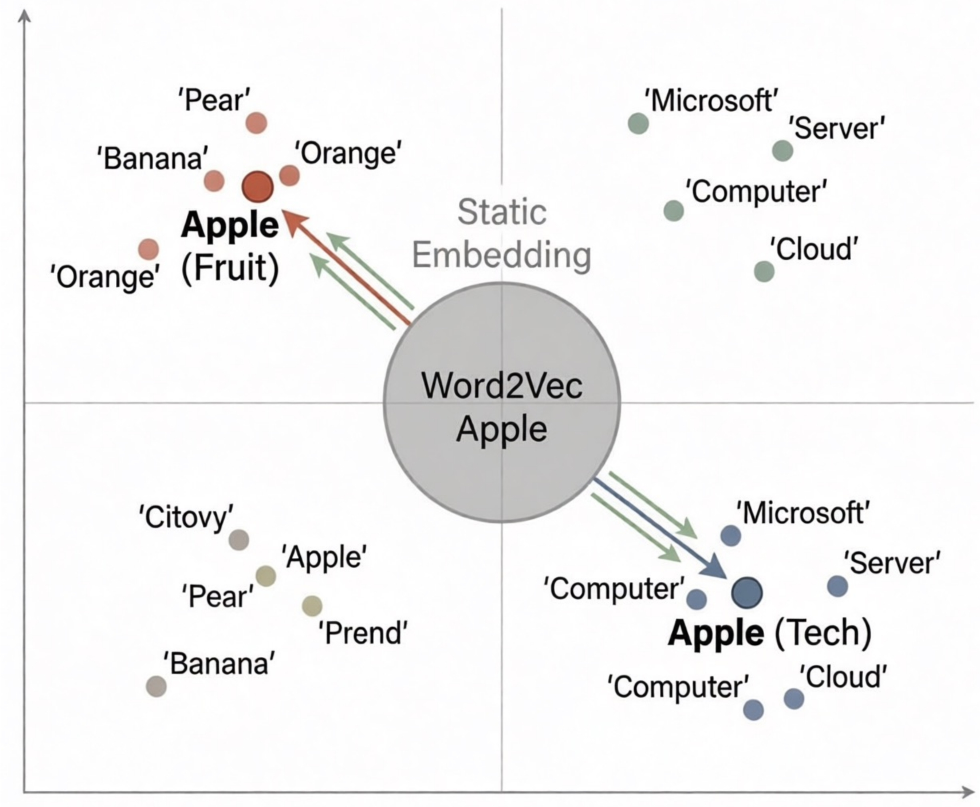
The ELMo Architecture

Processing sequence in both directions allows the model to understand that 'bank' means something different in 'river bank' vs 'bank deposit.

Contextualized Embeddings: ELMo

Embeddings from Language Models (2018): Solving **Polysemy**.

ELMo
of Sesame Street



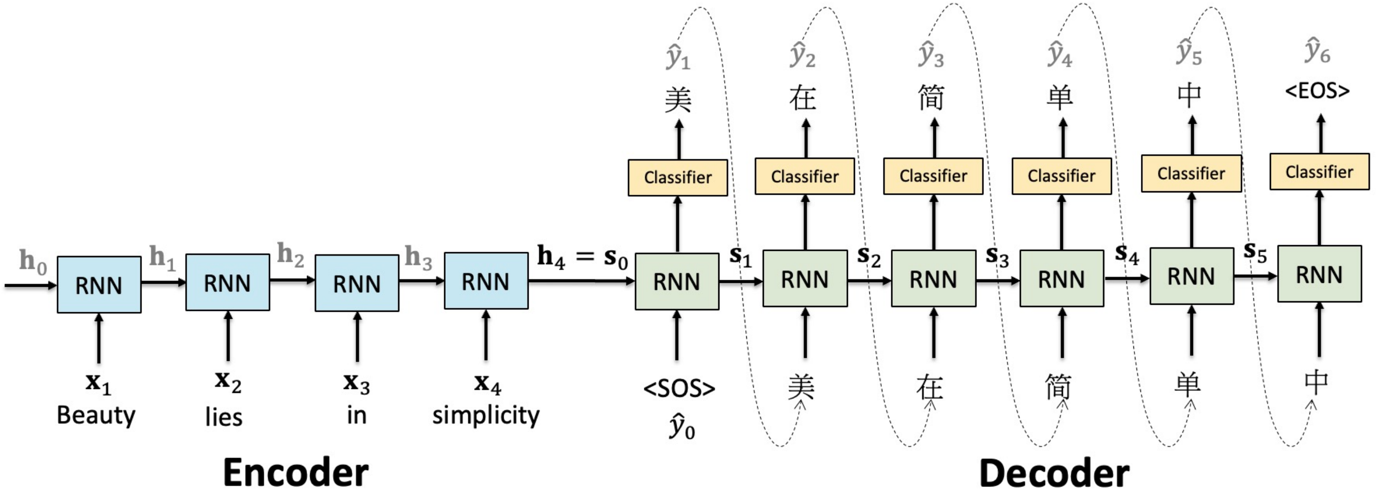
Peters et al, Deep Contextualized Word Representations", in NACCL-HLT, 2018

The Translation Challenge: Sequence-to-Sequence (Seq2Seq)

The Encoder-Decoder Architecture (Sutskever et al., 2014)

Era of Sequence-to-Sequence RNN (2014)

- In 2014, **Seq2Seq** models with RNNs pioneered modern neural translation in the paper "**Sequence to Sequence Learning with Neural Networks**" by Sutskever, Vinyals, and Le. This mapped source to target sequences using:
    - **Encoder RNN** (often LSTM for vanishing gradients): Compresses input into **a fixed context vector**.
    - **Decoder RNN**: Generates output word by word from the vector.



Anatomy of a Vanilla Encoder-Decoder

The Encoder:

$$\mathbf{h}_j = \mathrm{RNN}_e(\mathbf{x}_j, \mathbf{h}_{j-1})$$

Input Token

Previous Hidden State

The Bottleneck:

$$\mathbf{c} = \mathbf{h}_T = \mathbf{s}_0$$

Context Vector

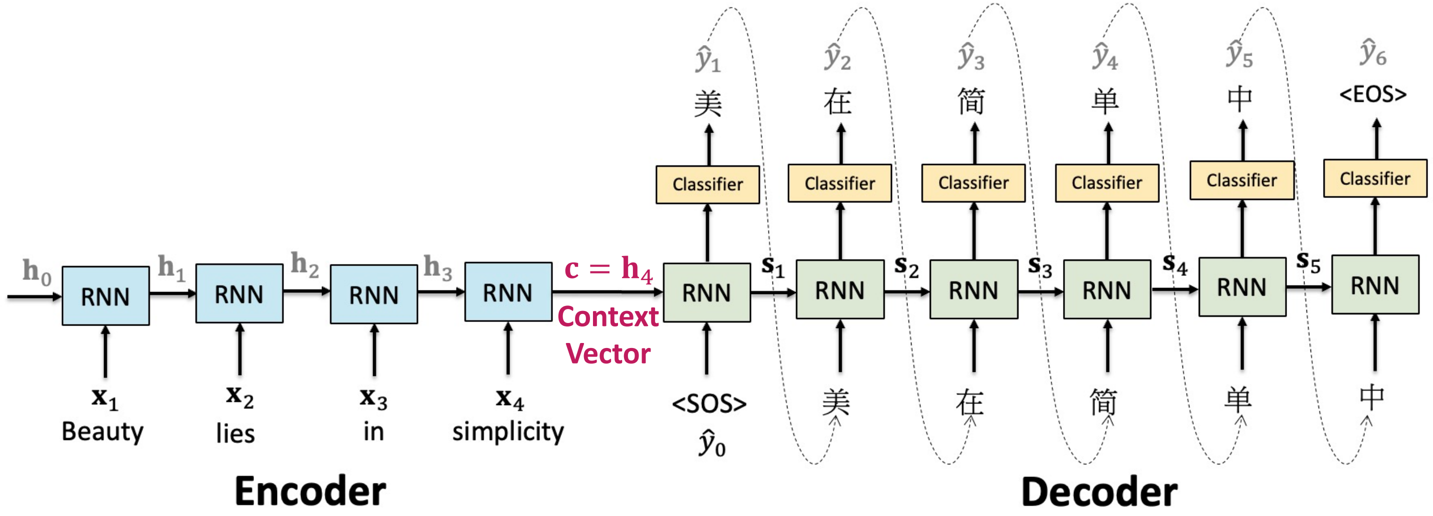
The final hidden state of the encoder ($\mathbf{h}_T$) becomes the specific Context Vector ($\mathbf{c}$). This vector initializes the decoder.

The Decoder Step:

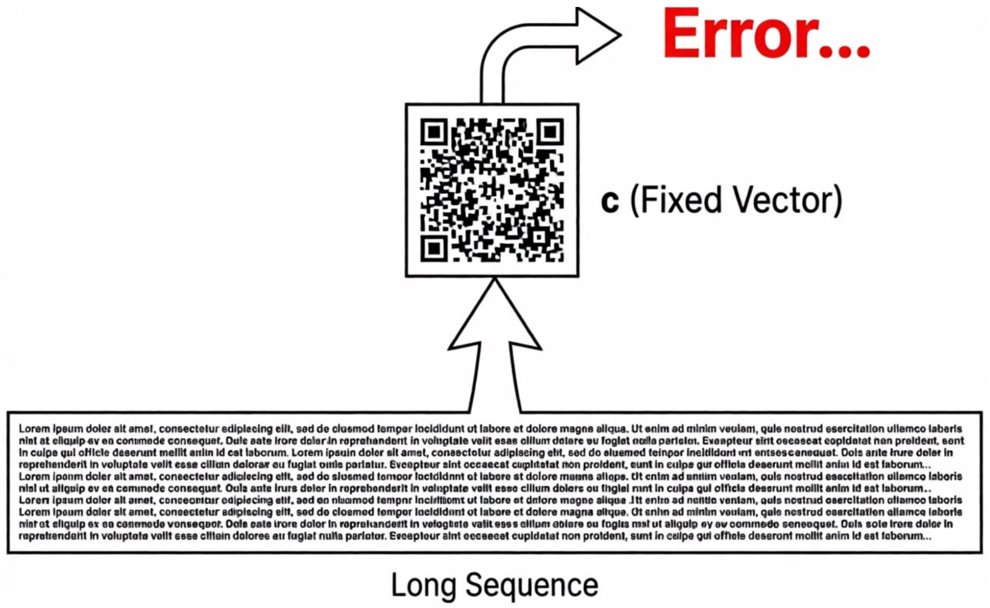
$$\mathbf{s}_t = \mathrm{RNN}_d(\hat{y}_{t-1}, \mathbf{s}_{t-1})$$

$$\hat{y}_t = \mathrm{softmax}(\mathbf{W}_y \mathbf{s}_t + \mathbf{b}_y)$$

The decoder uses the context to generate tokens step-by-step.



The Fatal Flaw: The Information Bottleneck



Compression Loss: Information discarded in fixed vector

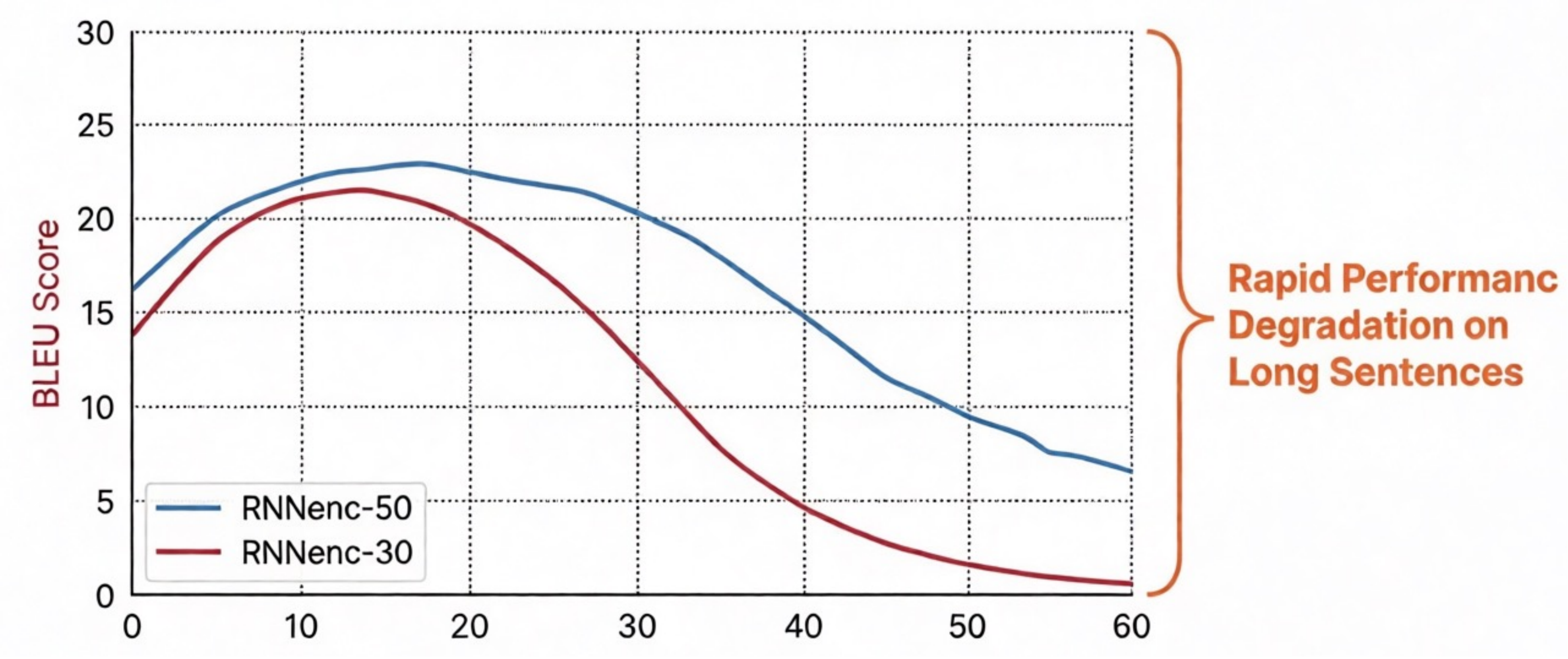
The Compressions Problem:

The model is forced to compress an arbitrarily long input sequence (X) into a fixed-length vector (**c**)

The Forgetting Phenomenon:

Because RNNs process sequentially, information from the beginning of the sentence is diluted by the time the encoder reaches the end. The vector '**c** ' is heavily biased toward the most recent tokens, causing the model to "forget" the start of the sentence.

The Performance Cliff



Empirical evidence showed that standard Seq2Seq models could not effectively translate sentences longer than 30 words, blocking real-world adoption.

The Attention Mechanism
A Game-Changer (2014–2015)

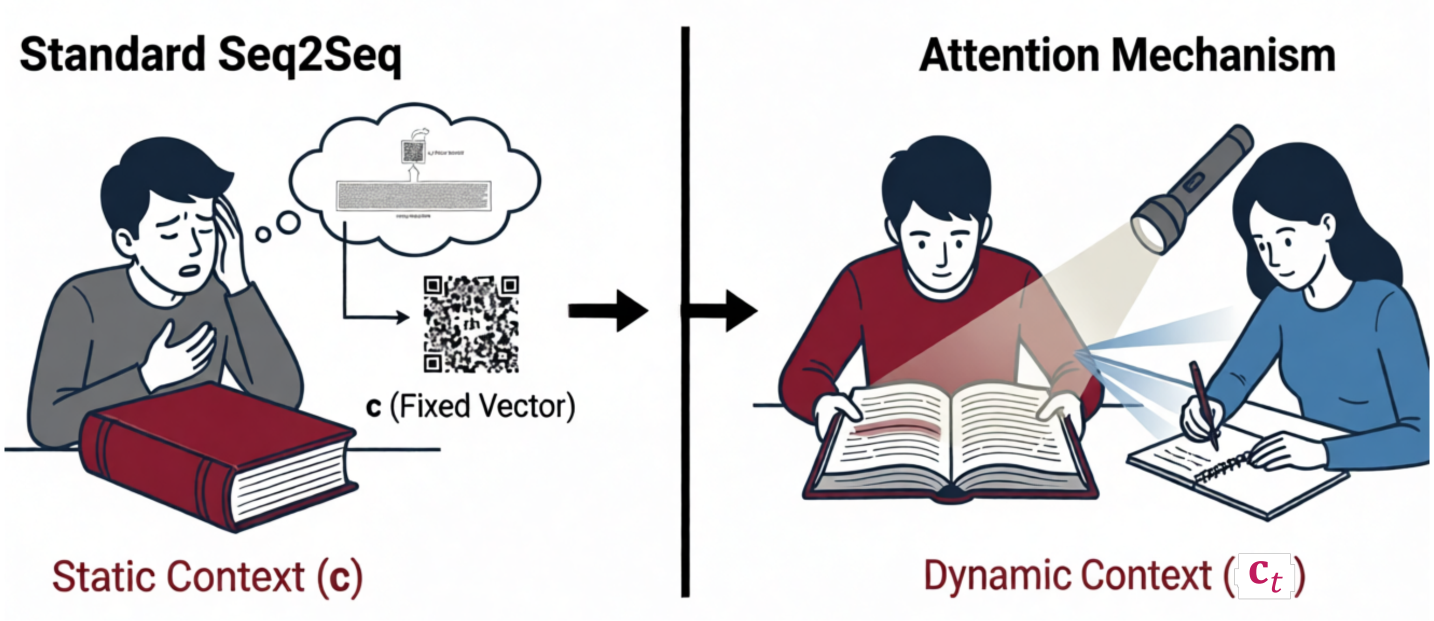
To resolve the fixed-length context vector bottleneck, attention mechanisms were introduced in two papers

“Neural Machine Translation by Jointly Learning to Align and Translate” (2014)

and

“Effective Approaches to Attention-based Neural Machine Translation” (2015).

Enter the Attention Mechanism

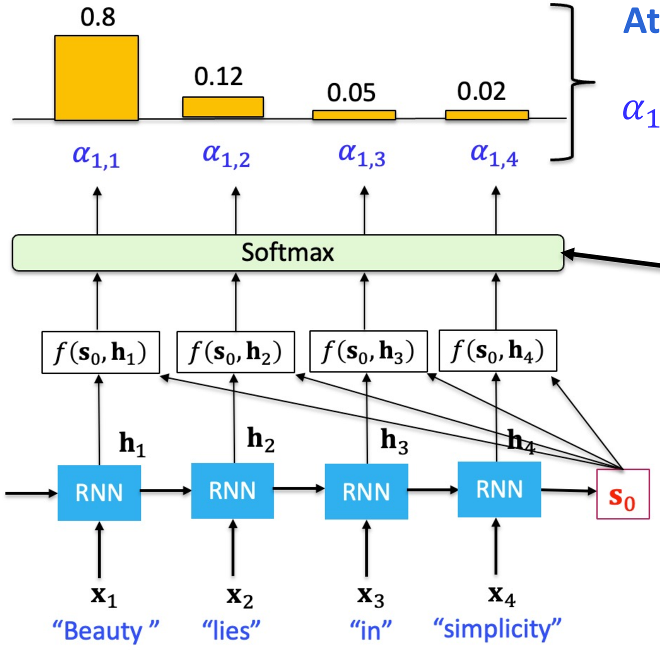


- **The Paradigm Shift**: Instead of relying on a single static vector for the whole translation, Attention allows the decoder to "look back" at the source sentence at every step.
- **Key Concept**: Computation of a **dynamic context vector** $\mathbf{c}_t$ for every decoder time step t, based on **Attention Weights** ($\alpha_{i,j}$).

Solution A: Bahdanau Attention (2014)

Neural Machine Translation by Jointly Learning to Align and Translate

- **Core Mechanism**: Uses a small feed-forward neural network to calculate alignment scores.
- **Key Characteristic:** The context vector is computed before the decoder updates its state.

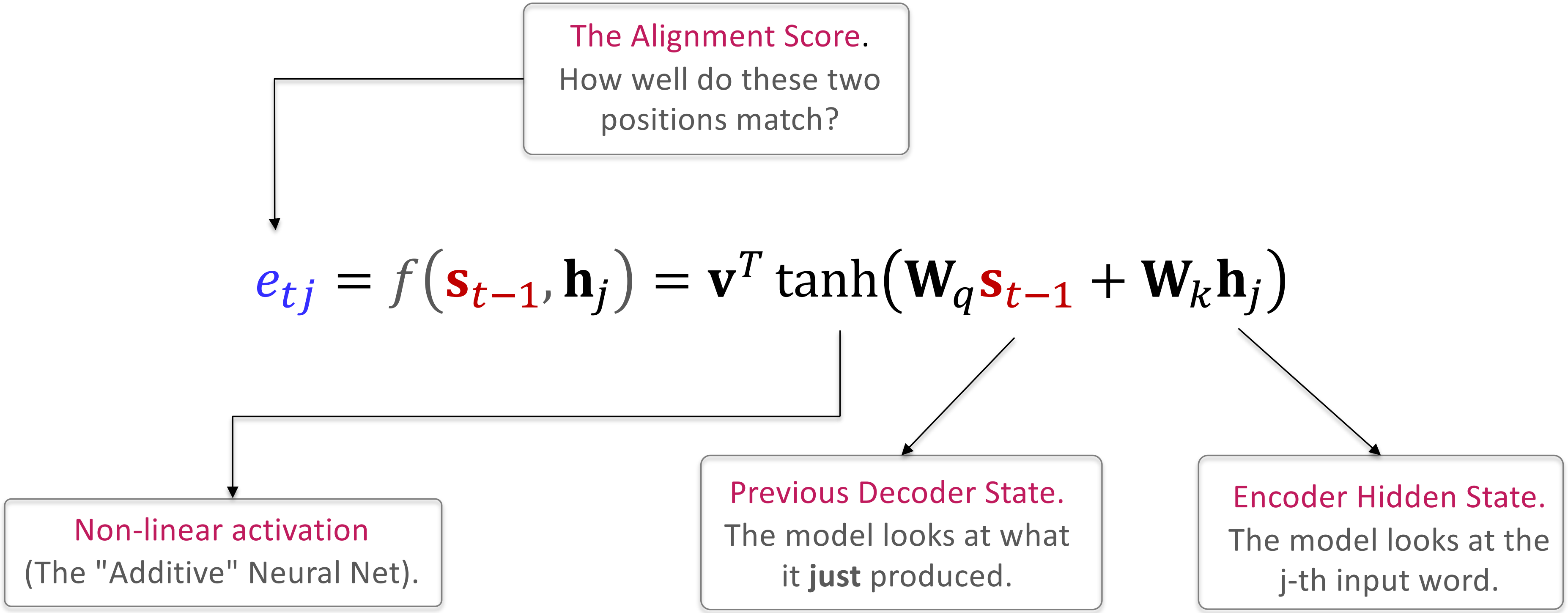


Attention Weights:

$$\alpha_{1,j} = \mathrm{softmax}\left(f\left(\mathbf{s}_0, \mathbf{h}_j\right)\right) = \frac{\exp\left(f\left(\mathbf{s}_0, \mathbf{h}_j\right)\right)}{\sum_{k=1}^{T} \exp\left(f\left(\mathbf{s}_0, \mathbf{h}_k\right)\right)}$$

We apply a softmax function to the alignment scores to obtain the normalized attention weights $\alpha_{1,j}$, which will be used as weights applying to encoder's hidden state $\mathbf{h}_j$ to create the context vector $\mathbf{c}_1$

Bahdanau Step 1: Alignment Scores



The Alignment Score. How well do these two positions match?

$$e_{tj} = f\left(\mathbf{s}_{t-1}, \mathbf{h}_j\right) = \mathbf{v}^T \tanh\left(\mathbf{W}_q \mathbf{s}_{t-1} + \mathbf{W}_k \mathbf{h}_j\right)$$

Non-linear activation (The "Additive" Neural Net).

Previous Decoder State. The model looks at what it **just** produced.

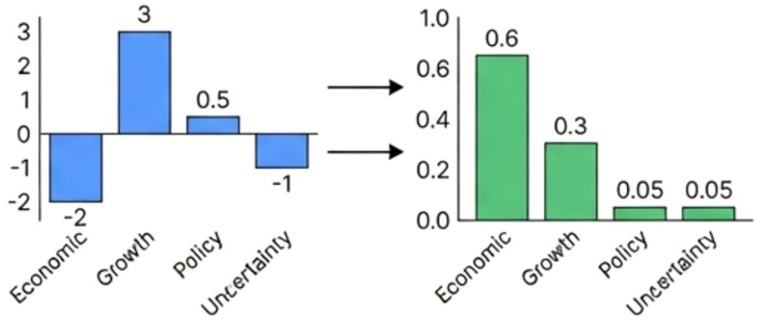
Encoder Hidden State. The model looks at the j-th input word.

Bahdanau's critical design choice: the alignment score uses an additive combination of transformed vectors: query $\mathbf{s}_{t-1}$ and key $\mathbf{h}_j$ are projected via learnable matrices $\mathbf{W}_q$ and $\mathbf{W}_k$, added, passed through tanh, and dotted with vector $\mathbf{v}$.

Bahdanau Step 2 & 3: Weights & Context

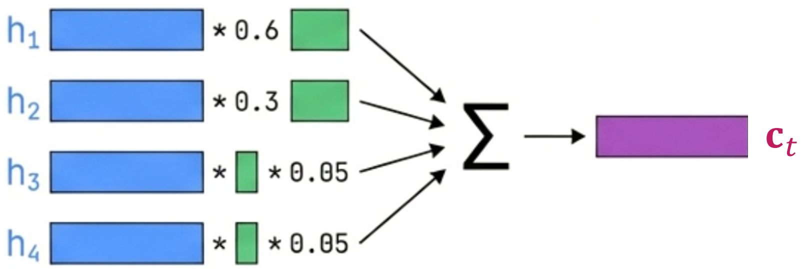
- Softmax Normalization

$$\alpha_{t,j} = \text{softmax}(e_{tj})$$



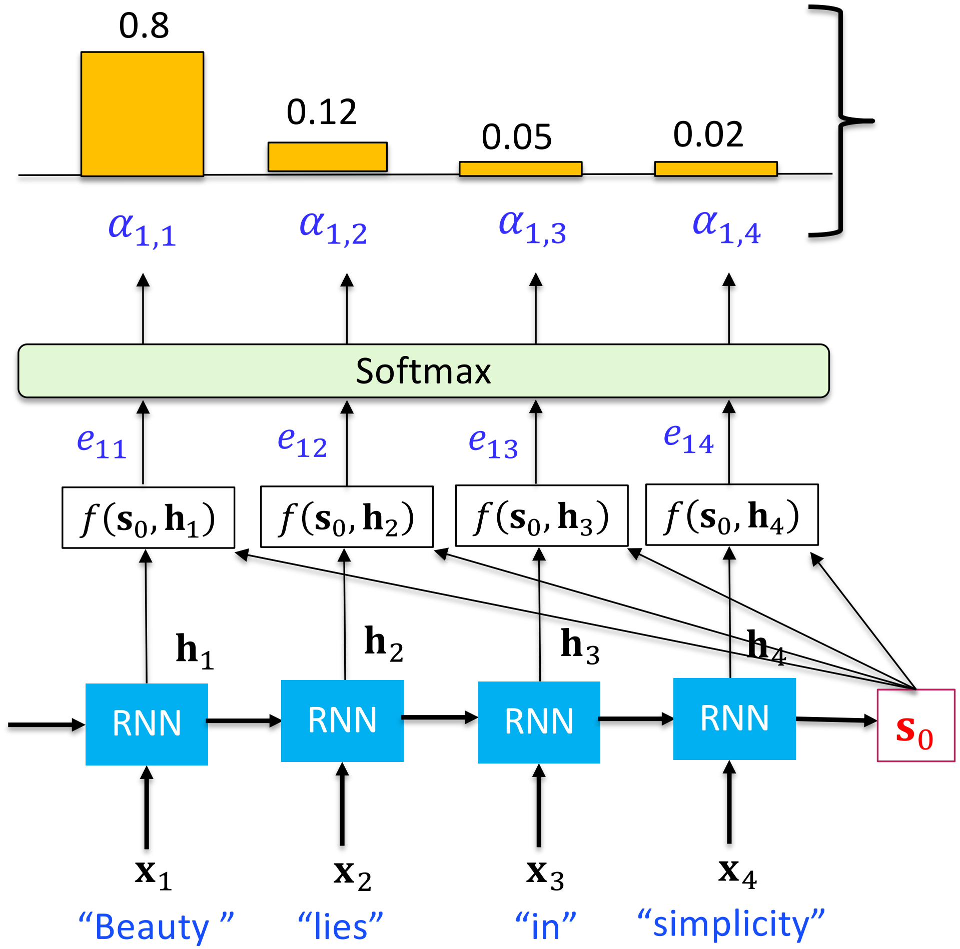
- Context Vector Calculation

$$\mathbf{c}_t = \sum_j \alpha_{t,j} \mathbf{h}_j$$



The Context Vector c_t is a weighted sum. If the model is 90% focused on "Economic", the resulting vector is 90% "Economic".

Attention Weights



Attention Weights:

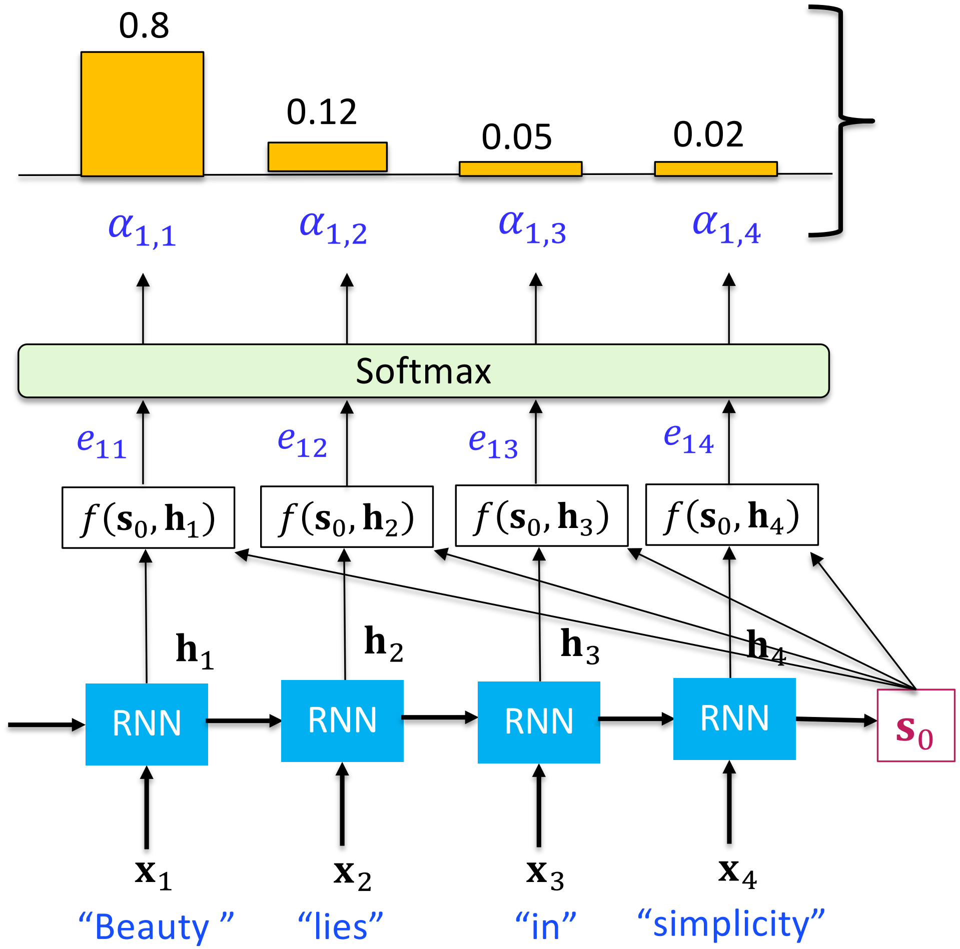
$$\alpha_{1,j} = \text{softmax}\left(f\left(\mathbf{s}_0, \mathbf{h}_j\right)\right)$$

$$= \frac{\exp\left(f\left(\mathbf{s}_0, \mathbf{h}_j\right)\right)}{\sum_{k=1}^{T} \exp\left(f\left(\mathbf{s}_0, \mathbf{h}_k\right)\right)}$$

where $m = 4$ in this example

- We apply a softmax activation function to the alignment scores to obtain the normalized attention weights $\alpha_{1,j}$, which will be used as weights applying to encoder’s hidden state $\mathbf{h}_j$ to create the context vector $\mathbf{c}_1$

Attention Weights

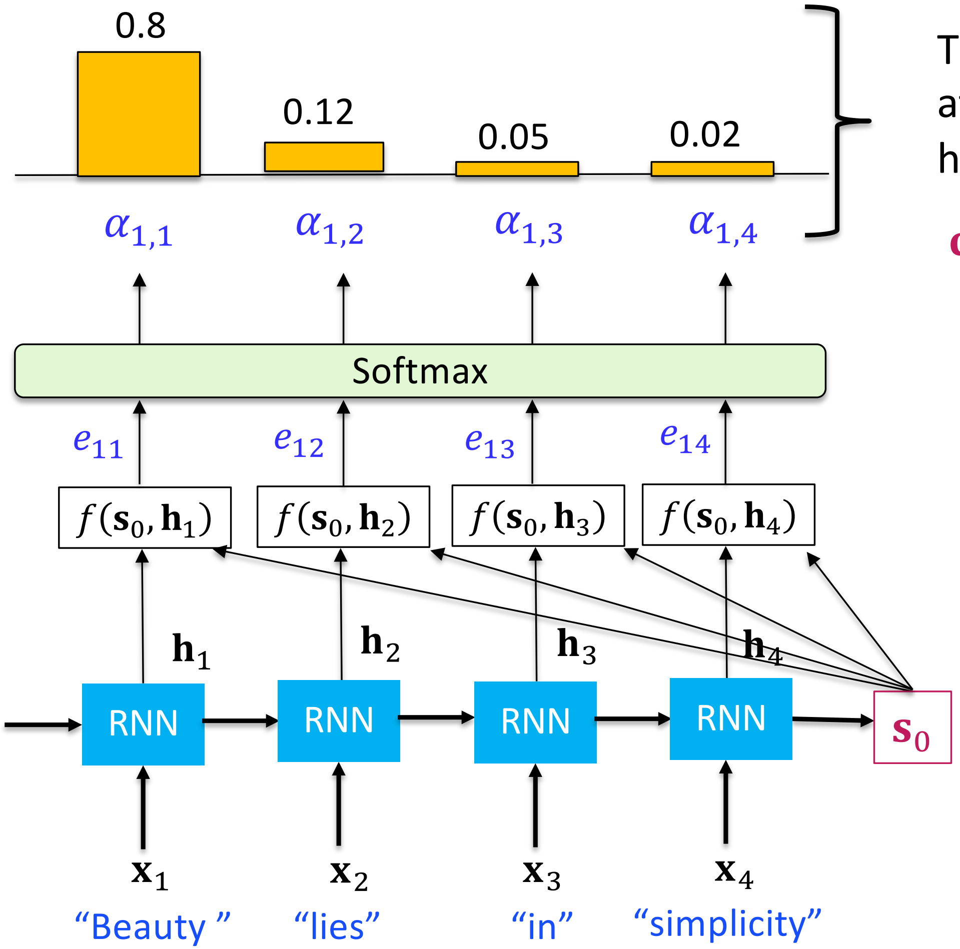


- In general, the normalized attention weights $\alpha_{t,j}$ for the output at position “t” can be calculated as

$$\alpha_{t,j} = \text{softmax}\left(f\left(\mathbf{s}_{t-1}, \mathbf{h}_j\right)\right)$$

$$= \frac{\exp\left(f\left(\mathbf{s}_{t-1}, \mathbf{h}_j\right)\right)}{\sum_{k=1}^{T} \exp\left(f\left(\mathbf{s}_{t-1}, \mathbf{h}_k\right)\right)}$$

Context Vector with Attention



The context vector $\mathbf{c}_1$ is the weighted sum of each attention weight with its corresponding encoder hidden state $(\mathbf{h}_k)$

$$\mathbf{c}_1 = \alpha_{1,1}\mathbf{h}_1 + \alpha_{1,2}\mathbf{h}_2 + \alpha_{1,3}\mathbf{h}_3 + \alpha_{1,4}\mathbf{h}_4$$

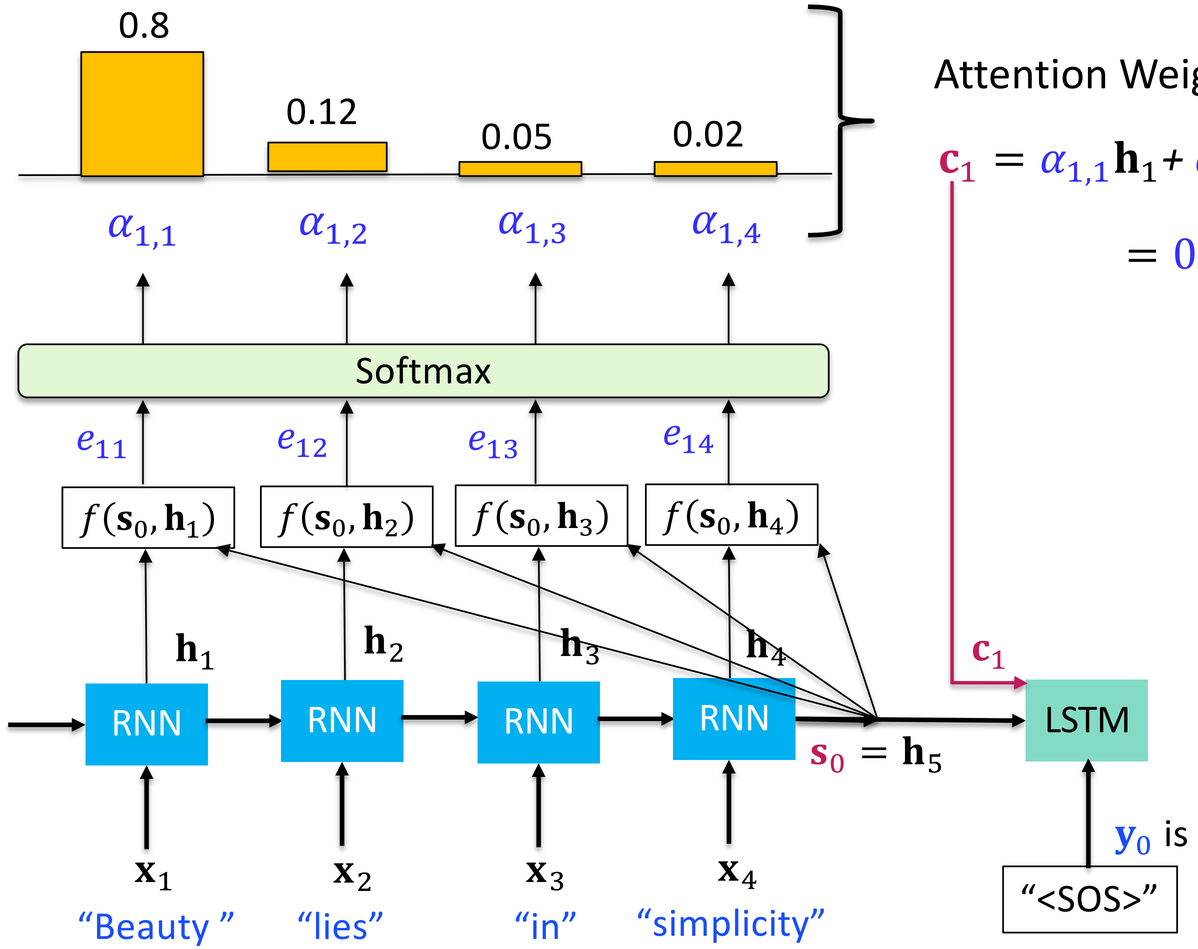
$$= 0.8\mathbf{h}_1 + 0.12\mathbf{h}_2 + 0.05\mathbf{h}_3 + 0.02\mathbf{h}_4$$

Attention Weights $\alpha_{t,j} = \dfrac{\exp\left(f(\mathbf{s}_{t-1}, \mathbf{h}_j)\right)}{\sum_{k=1}^{T} \exp\left(f(\mathbf{s}_{t-1}, \mathbf{h}_k)\right)}$

$$\mathbf{c}_t = \sum_j \alpha_{t,j}\mathbf{h}_j$$

Context Vector

Attention Mechanism: $\mathbf{c}_1$



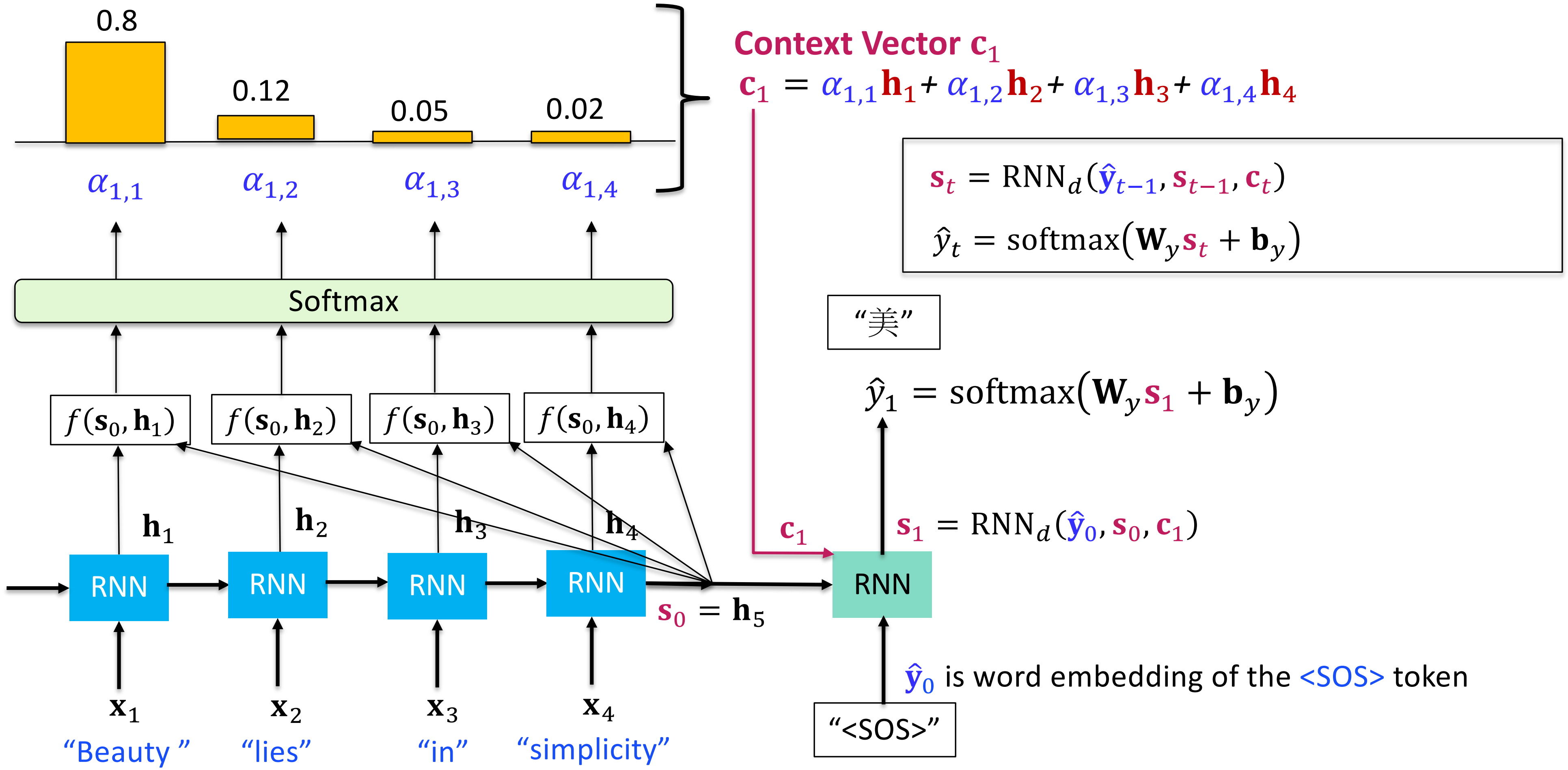
Attention Weights $\alpha_{1,j}$

$$\mathbf{c}_1 = \alpha_{1,1}\mathbf{h}_1 + \alpha_{1,2}\mathbf{h}_2 + \alpha_{1,3}\mathbf{h}_3 + \alpha_{1,4}\mathbf{h}_4$$

$$= 0.8\mathbf{h}_1 + 0.12\mathbf{h}_2 + 0.05\mathbf{h}_3 + 0.02\mathbf{h}_4$$

$\mathbf{y}_0$ is word embedding of the <SOS> token

Attention Mechanism: $\mathbf{c}_1$



Context Vector $\mathbf{c}_1$

$$\mathbf{c}_1 = \alpha_{1,1}\mathbf{h}_1 + \alpha_{1,2}\mathbf{h}_2 + \alpha_{1,3}\mathbf{h}_3 + \alpha_{1,4}\mathbf{h}_4$$

$$\mathbf{s}_t = \mathrm{RNN}_d(\hat{\mathbf{y}}_{t-1}, \mathbf{s}_{t-1}, \mathbf{c}_t)$$

$$\hat{y}_t = \mathrm{softmax}\big(\mathbf{W}_y\mathbf{s}_t + \mathbf{b}_y\big)$$

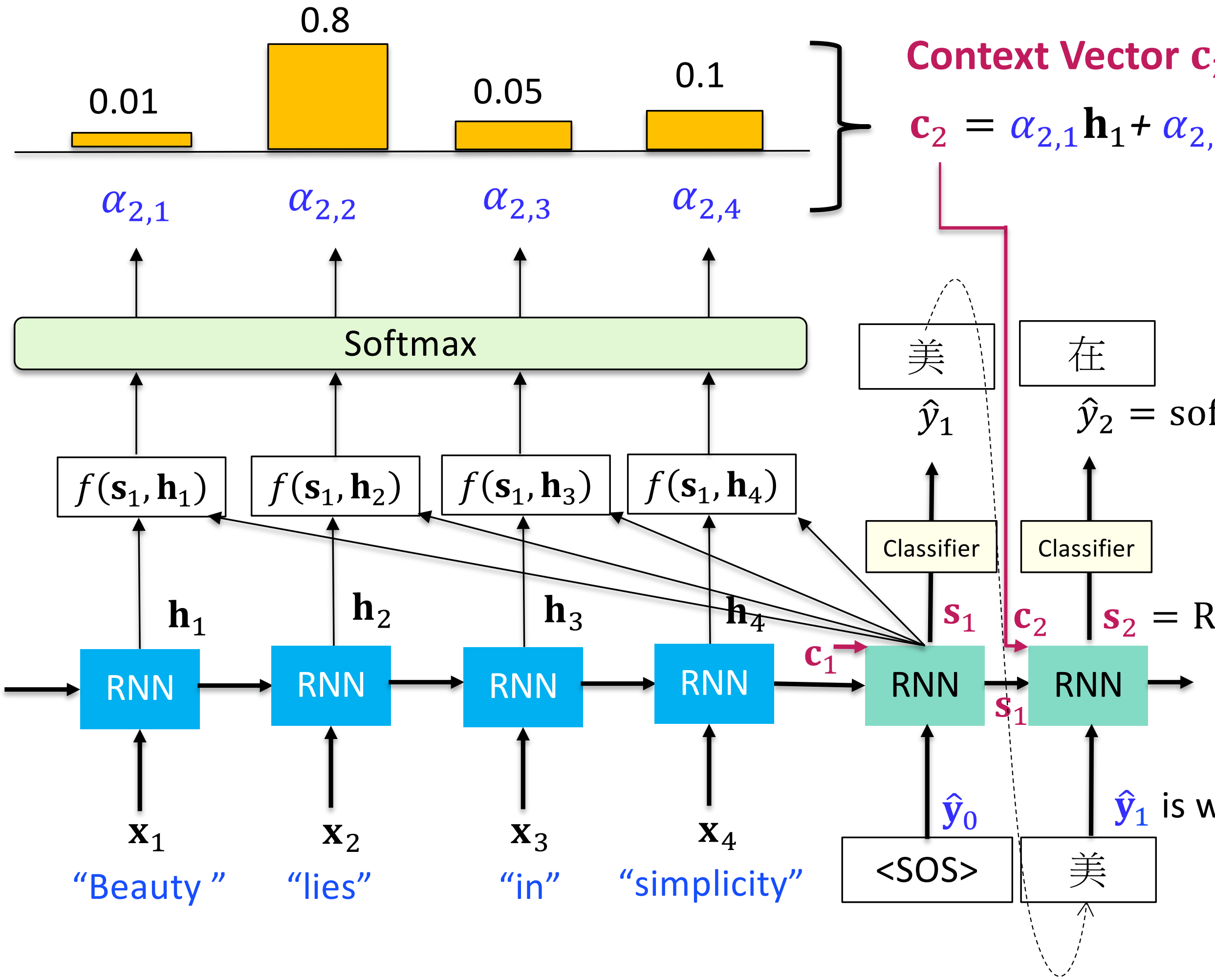
$$\hat{y}_1 = \mathrm{softmax}\big(\mathbf{W}_y\mathbf{s}_1 + \mathbf{b}_y\big)$$

$$\mathbf{s}_1 = \mathrm{RNN}_d(\hat{\mathbf{y}}_0, \mathbf{s}_0, \mathbf{c}_1)$$

$$\mathbf{s}_0 = \mathbf{h}_5$$

$\hat{\mathbf{y}}_0$ is word embedding of the <SOS> token

Attention Mechanism: $\mathbf{c}_2$



Context Vector $\mathbf{c}_2$

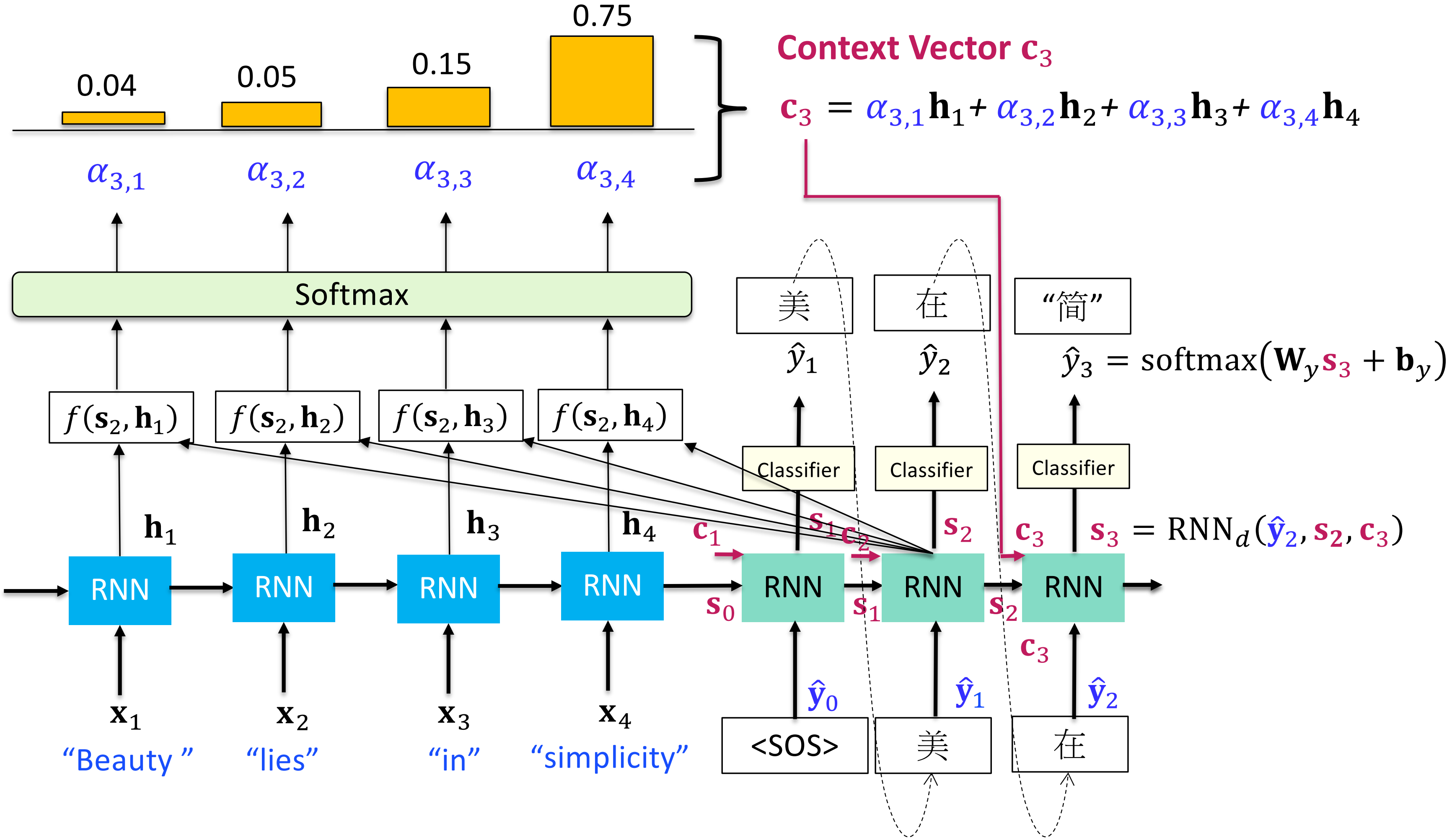
$$\mathbf{c}_2 = \alpha_{2,1}\mathbf{h}_1 + \alpha_{2,2}\mathbf{h}_2 + \alpha_{2,3}\mathbf{h}_3 + \alpha_{2,4}\mathbf{h}_4$$

$$\hat{y}_2 = \mathrm{softmax}\big(\mathbf{W}_y \mathbf{s}_2 + \mathbf{b}_y\big)$$

$$\mathbf{s}_2 = \mathrm{RNN}_d(\hat{\mathbf{y}}_1, \mathbf{s}_1, \mathbf{c}_2)$$

$\hat{\mathbf{y}}_1$ is word embedding of the $\hat{y}_1$ token

Attention Mechanism: $\mathbf{c}_3$



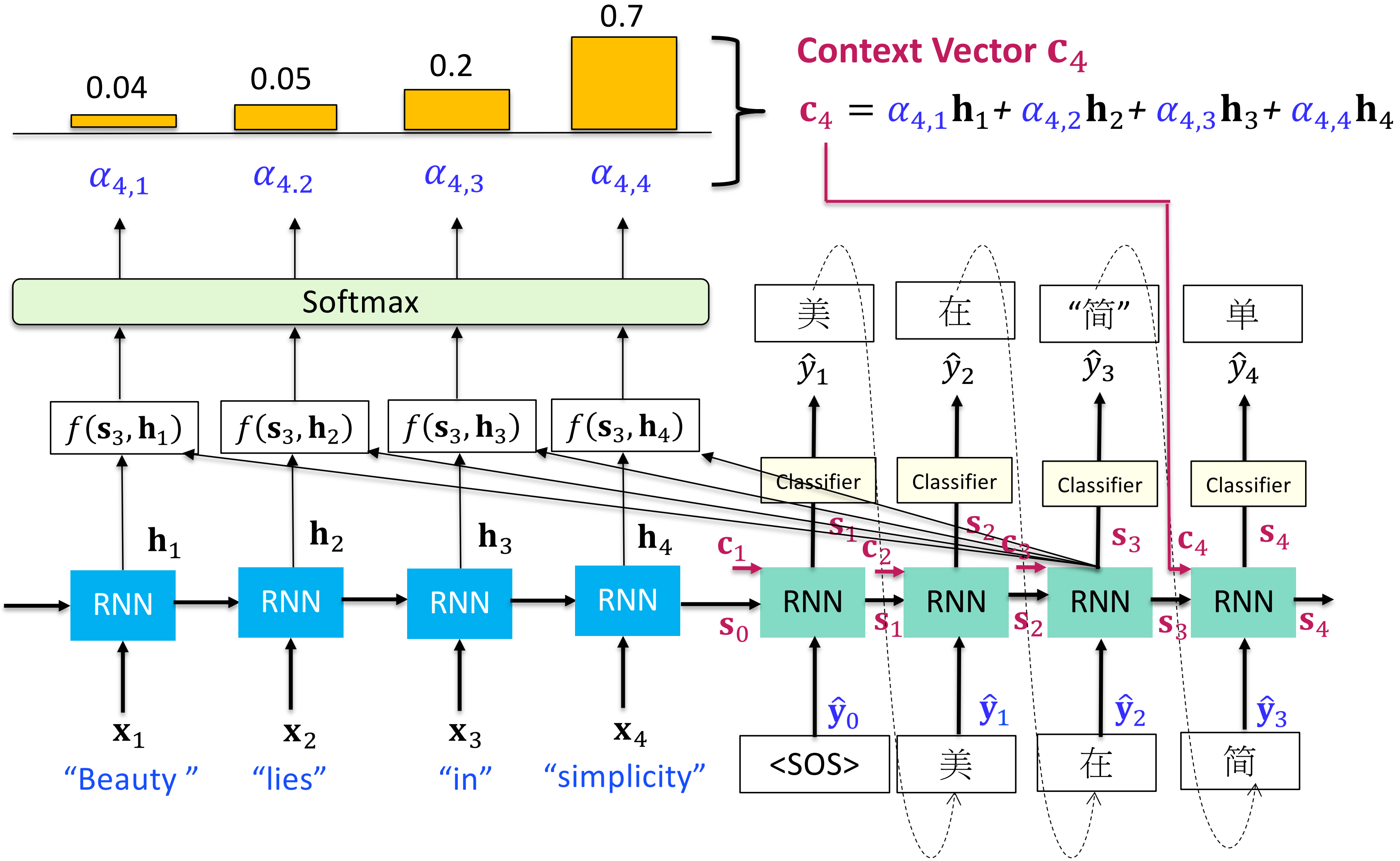
Context Vector $\mathbf{c}_3$

$$\mathbf{c}_3 = \alpha_{3,1}\mathbf{h}_1 + \alpha_{3,2}\mathbf{h}_2 + \alpha_{3,3}\mathbf{h}_3 + \alpha_{3,4}\mathbf{h}_4$$

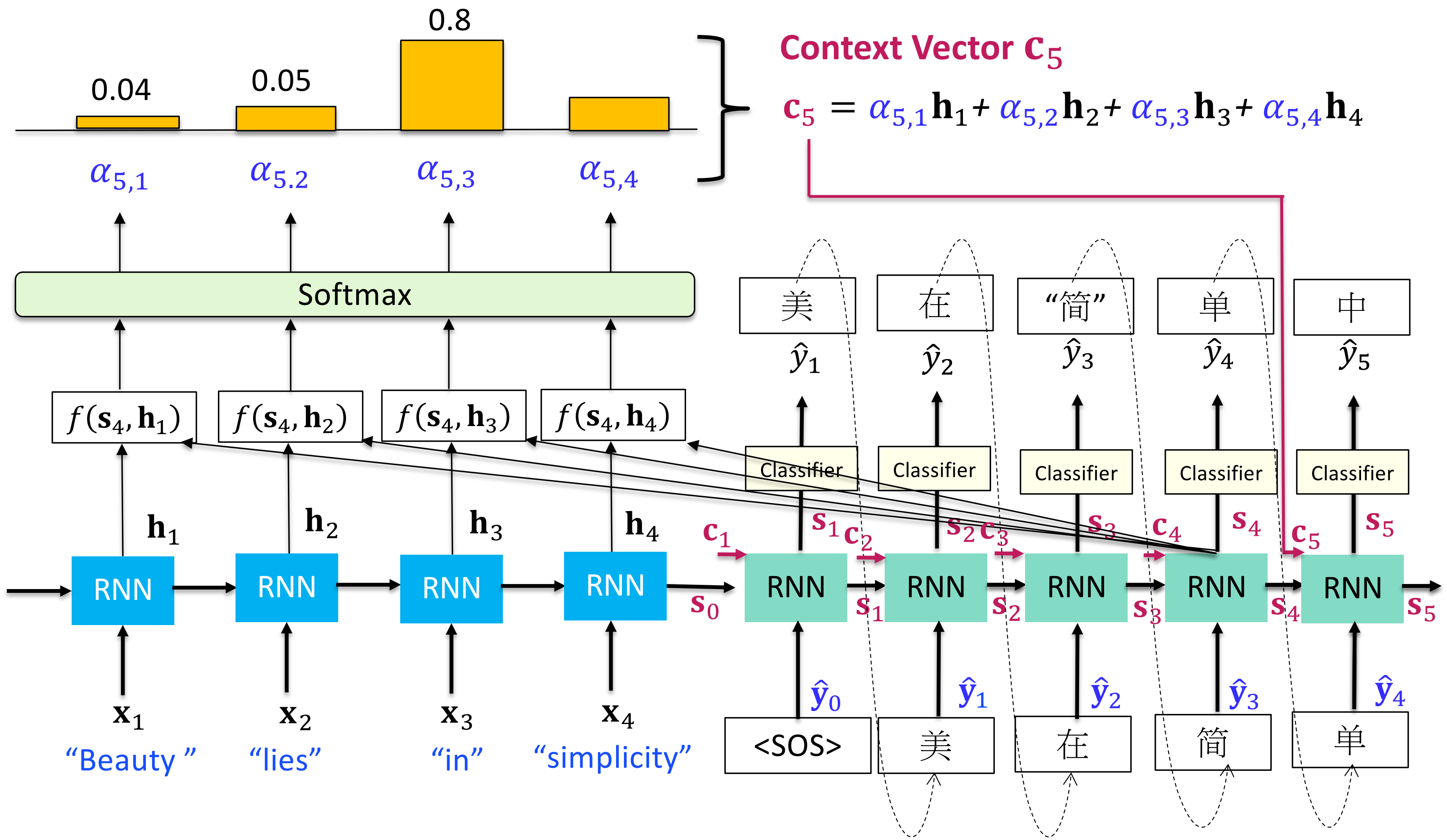
$$\hat{y}_3 = \text{softmax}(\mathbf{W}_y\mathbf{s}_3 + \mathbf{b}_y)$$

$$\mathbf{s}_3 = \text{RNN}_d(\hat{\mathbf{y}}_2, \mathbf{s}_2, \mathbf{c}_3)$$

Attention Mechanism: $\mathbf{c}_4$



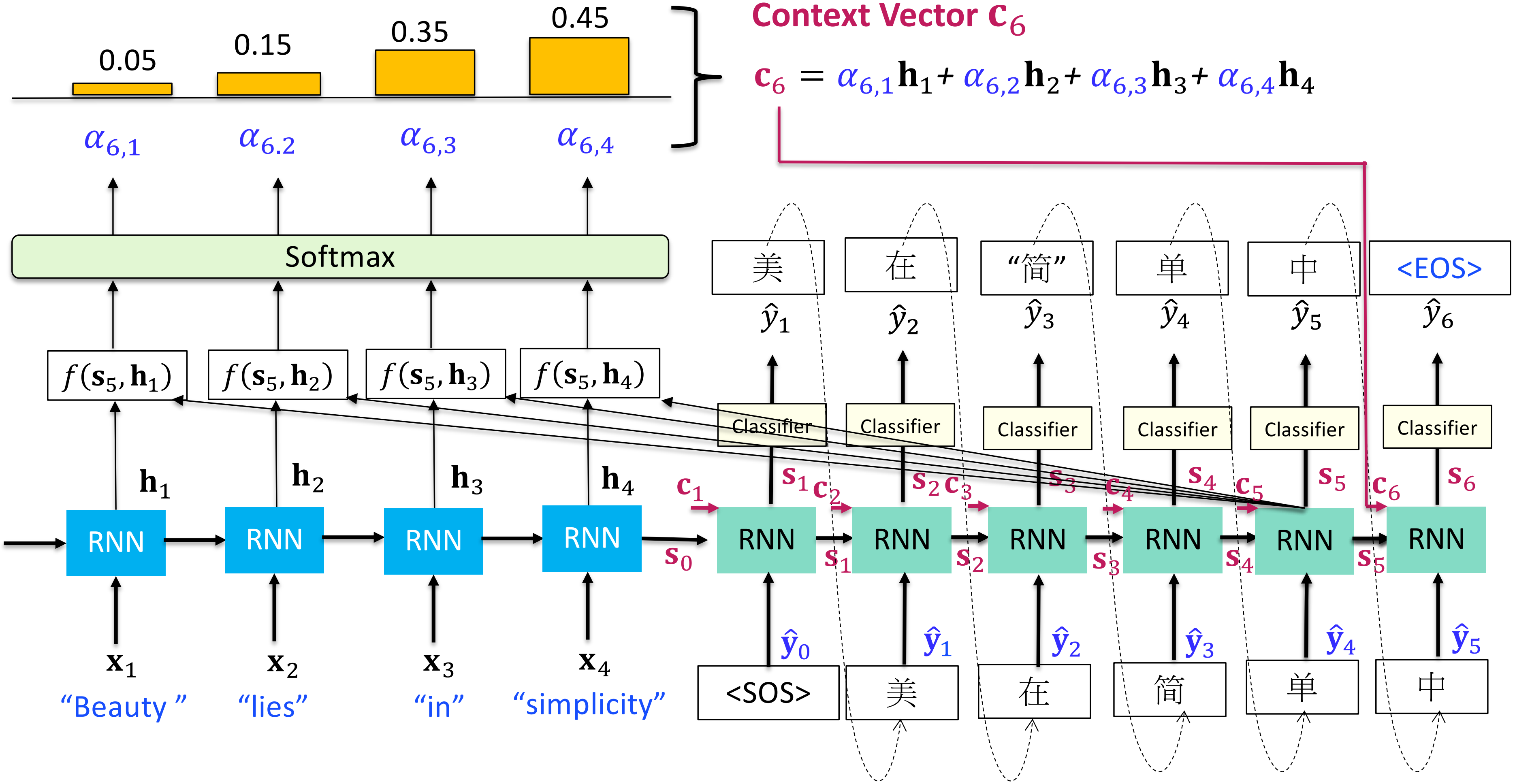
Attention Mechanism: $\mathbf{c}_5$



Context Vector $\mathbf{c}_5$

$$\mathbf{c}_5 = \alpha_{5,1}\mathbf{h}_1 + \alpha_{5,2}\mathbf{h}_2 + \alpha_{5,3}\mathbf{h}_3 + \alpha_{5,4}\mathbf{h}_4$$

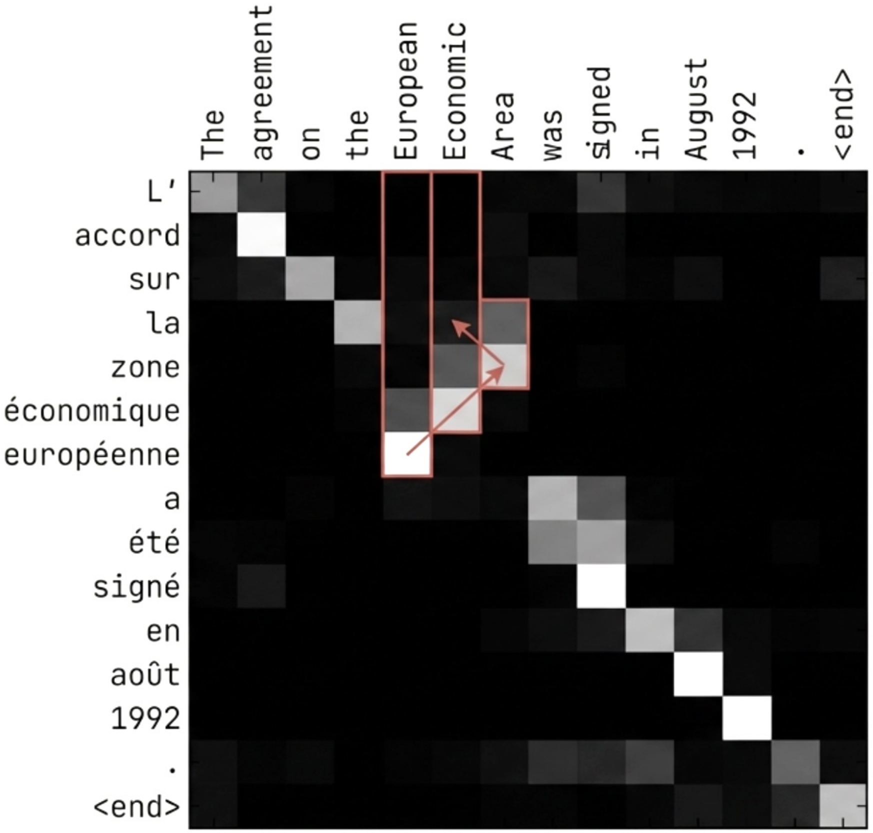
Attention Mechanism: $\mathbf{c}_6$


0.05
0.15
0.35
0.45
$\alpha_{6,1}$
$\alpha_{6.2}$
$\alpha_{6,3}$
$\alpha_{6,4}$
Softmax
$f(\mathbf{s}_5, \mathbf{h}_1)$
$f(\mathbf{s}_5, \mathbf{h}_2)$
$f(\mathbf{s}_5, \mathbf{h}_3)$
$f(\mathbf{s}_5, \mathbf{h}_4)$
$\mathbf{h}_1$
$\mathbf{h}_2$
$\mathbf{h}_3$
$\mathbf{h}_4$
RNN
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_3$
$\mathbf{x}_4$
“Beauty ”
“lies”
“in”
“simplicity”
$\mathbf{s}_0$
$\mathbf{c}_1$ $\mathbf{c}_2$ $\mathbf{c}_3$ $\mathbf{c}_4$ $\mathbf{c}_5$ $\mathbf{c}_6$
$\mathbf{s}_1$ $\mathbf{s}_2$ $\mathbf{s}_3$ $\mathbf{s}_4$ $\mathbf{s}_5$ $\mathbf{s}_6$
Classifier
美
在
“简”
单
中
<EOS>
$\hat{y}_1$ $\hat{y}_2$ $\hat{y}_3$ $\hat{y}_4$ $\hat{y}_5$ $\hat{y}_6$
$\hat{\mathbf{y}}_0$ $\hat{\mathbf{y}}_1$ $\hat{\mathbf{y}}_2$ $\hat{\mathbf{y}}_3$ $\hat{\mathbf{y}}_4$ $\hat{\mathbf{y}}_5$
<SOS>
美
在
简
单
中

Context Vector $\mathbf{c}_6$

$$\mathbf{c}_6 = \alpha_{6,1}\mathbf{h}_1 + \alpha_{6,2}\mathbf{h}_2 + \alpha_{6,3}\mathbf{h}_3 + \alpha_{6,4}\mathbf{h}_4$$

The Result: Interpretability



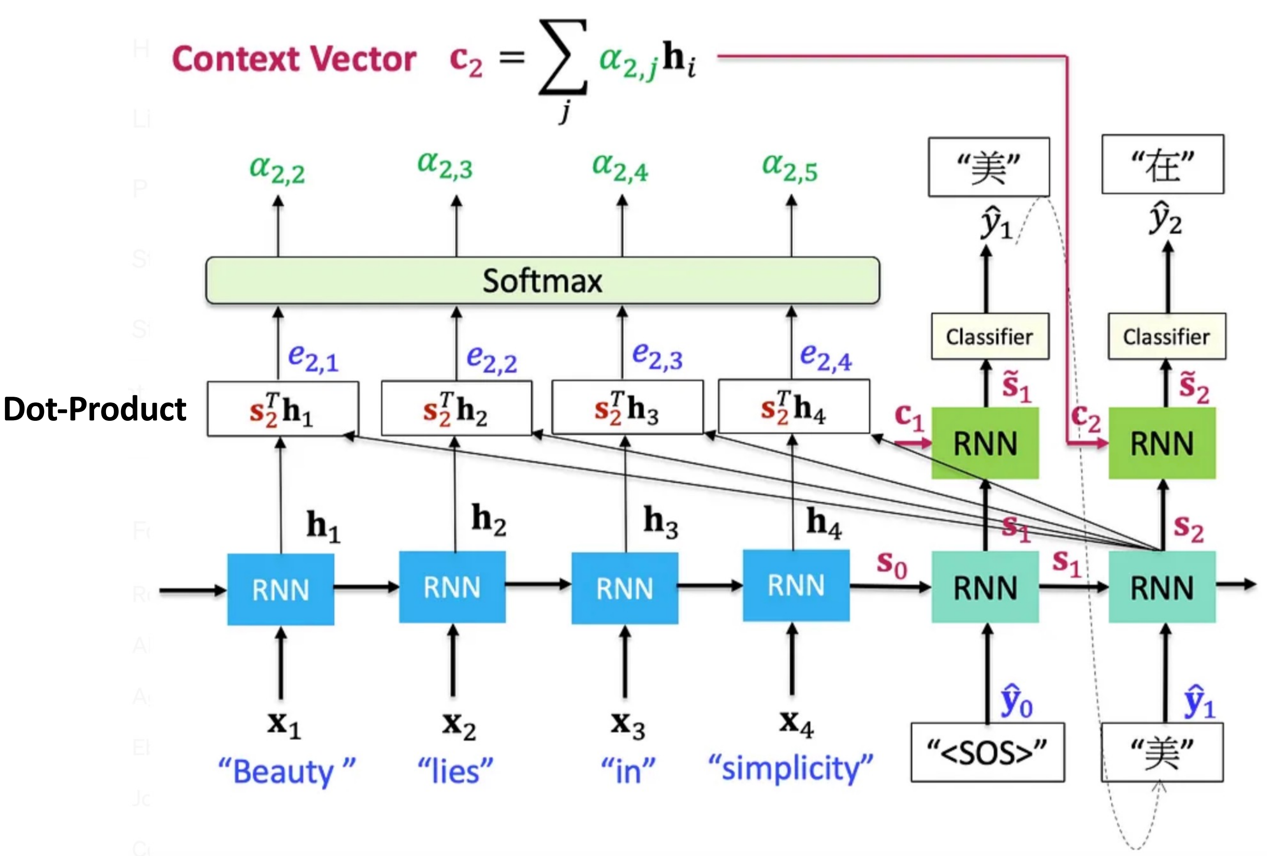
Seeing the Thinking:
Attention weights provide a heatmap of the model's focus.

The Insight:
In this example, the model successfully aligns English adjectives (pre-noun) with French adjectives (post-noun), jumping 'forward' and 'backward' in the source sentence effectively.

Image from ['Neural Machine Translation by jointly learning to align and translate'] by Bzmitry Bahdanau et al in 2014.

Solution B: Luong Attention (2015)

Effective Approaches to Attention-based Neural Machine Translation

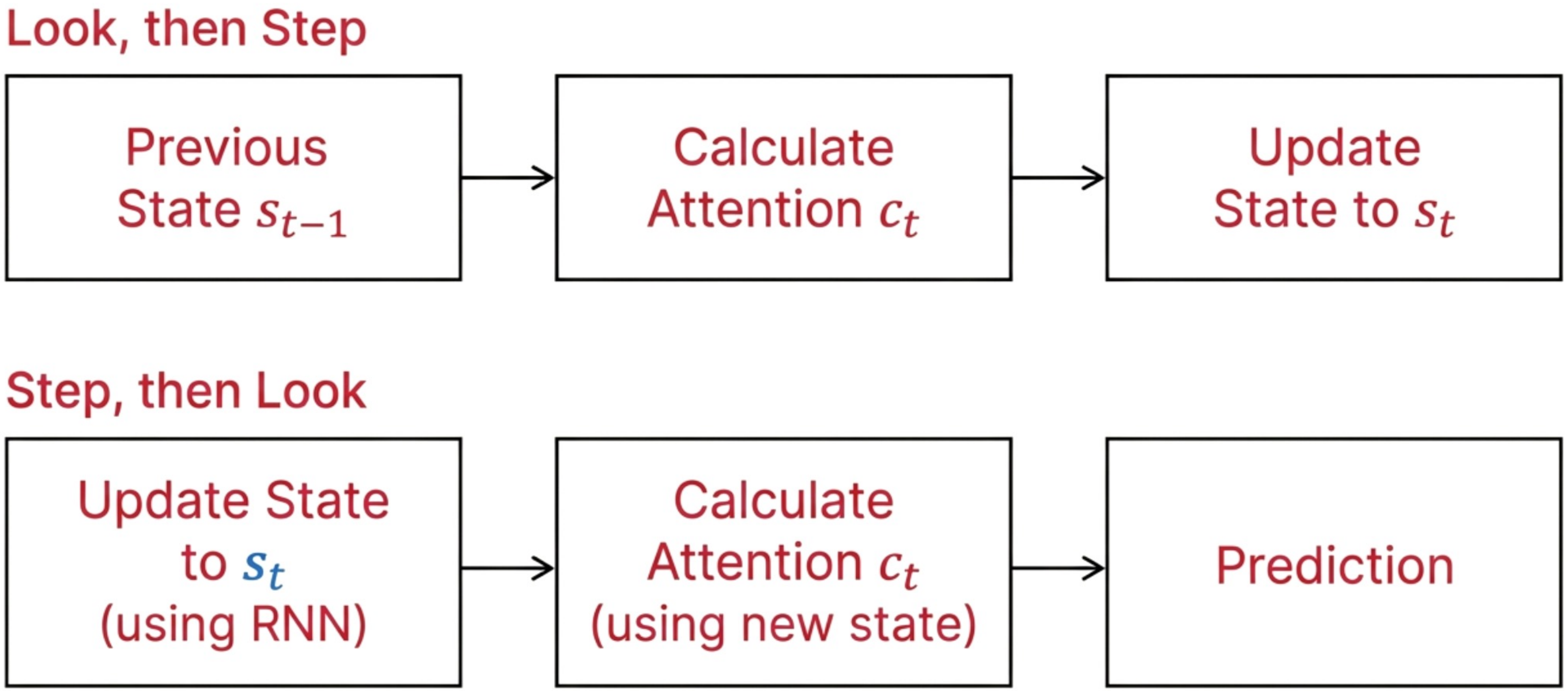


Classification:
Multiplicative Attention

Core Mechanism:
Simplifies scoring using matrix multiplication (**Dot Product**).

Key Characteristic: The attention score is computed using the current decoder state ($\mathbf{s}_t$), not the previous one.

Luong Step 1: The Timing Shift


Look, then Step
Previous State s_{t-1}
Calculate Attention c_t
Update State to s_t
Step, then Look
Update State to s_t (using RNN)
Calculate Attention c_t (using new state)
Prediction

Luong Step 2: Multiplicative Scoring

1. **Dot** (The most efficient):

$$e_{t,j} = \mathbf{s}_t^T \mathbf{h}_j$$

Pure dot product. Extremely fast matrix multiplication.

2. **General**:

$$e_{t,j} = \mathbf{s}_t^T \mathbf{W}_a \mathbf{h}_j$$

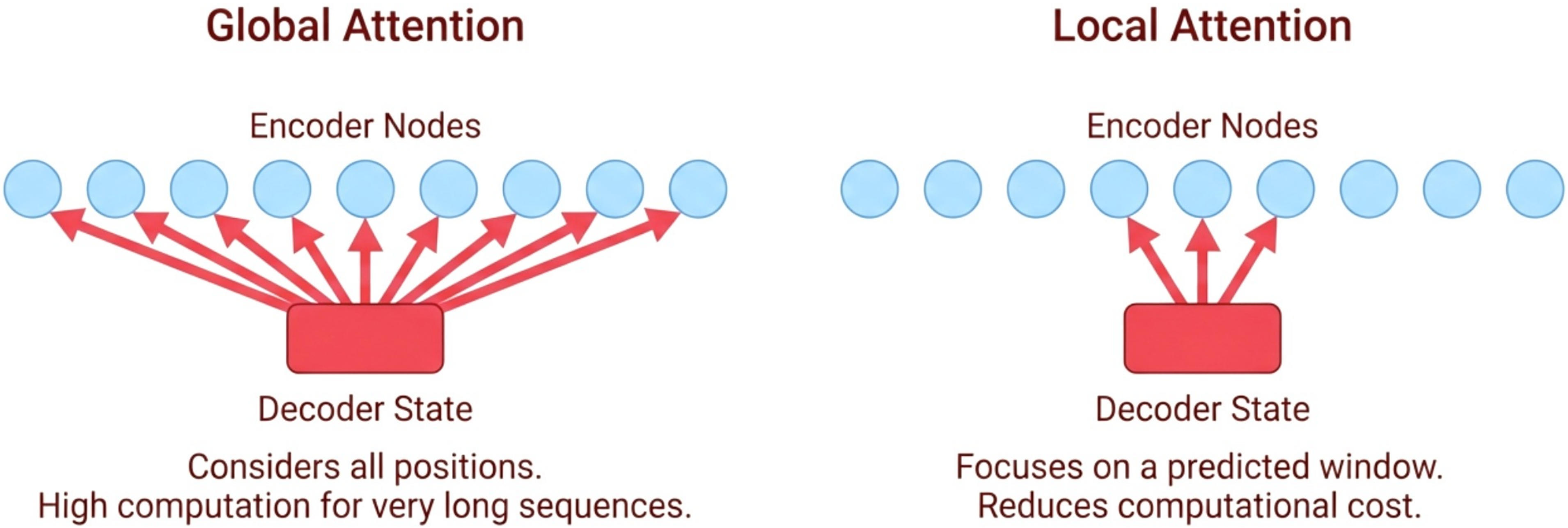
Includes a learnable weight matrix $\mathbf{W}_a$ for more flexibility.

3. **Concat**:

$$e_{t,j} = \mathbf{v}^T \tanh\left(\mathbf{W}_a \cdot \left[\mathbf{s}_t, \mathbf{h}_j\right]\right)$$

Similar to Bahdanau's additive approach, but uses current state st.

Global vs. Local Attention



Global Attention

Encoder Nodes

Decoder State

Considers all positions.
High computation for very long sequences.

Local Attention

Encoder Nodes

Decoder State

Focuses on a predicted window.
Reduces computational cost.

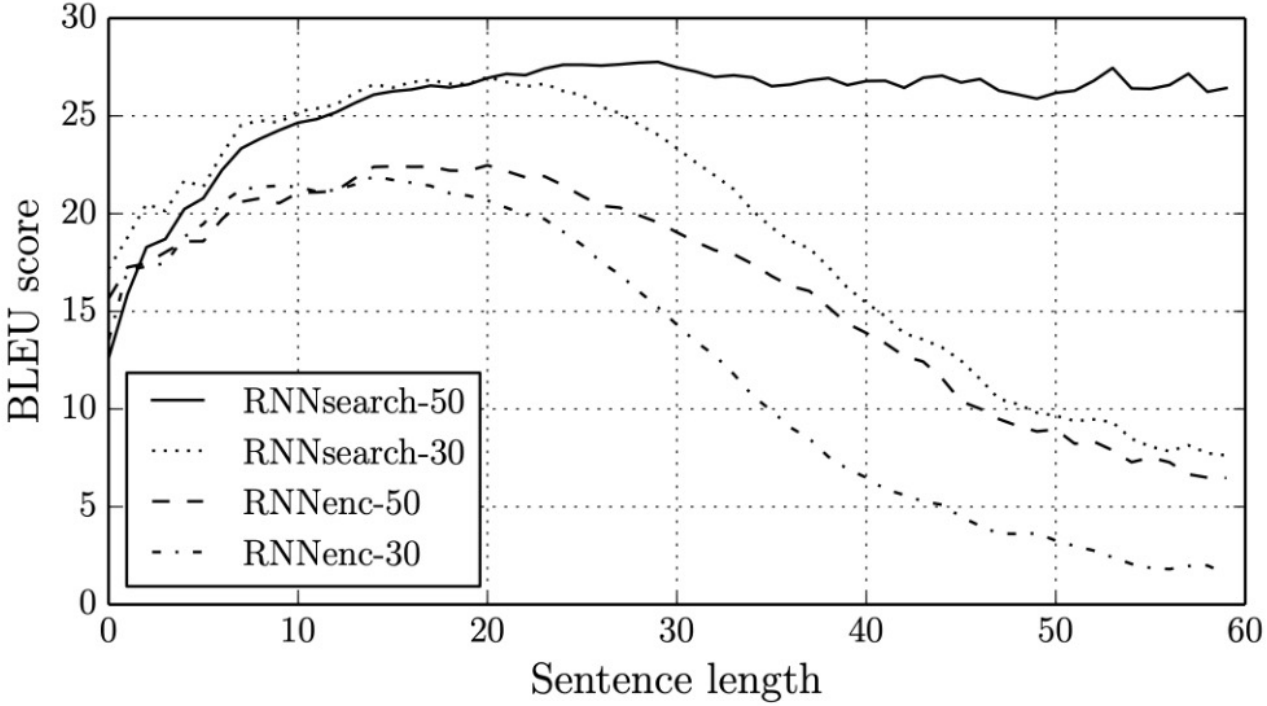
While Global Attention is the foundation of modern Transformers, Luong's Local Attention offered an optimization path for handling extremely long documents.

Head-to-Head: Bahdanau vs. Luong

	Bahdanau (Additive)	Luong (Multiplicative)
Score Calculation	Neural Network $\mathbf{v}^T \tanh\left(\mathbf{W}_q \mathbf{s}_{t-1} + \mathbf{W}_k \mathbf{h}_j\right)$	Matrix Multiplication $e_{t,j} = \mathbf{s}_t^T \mathbf{h}_j$
Input State	Previous Decoder State ($\mathbf{s}_{t-1}$)	Current Decoder State ($\mathbf{s}_t$)
Process Flow	$\mathbf{s}_{t-1} \rightarrow \text{Score} \rightarrow \mathbf{c}_t \rightarrow \mathbf{s}_t$	$\mathbf{s}_{t-1} \rightarrow \mathbf{s}_t \rightarrow \text{Score} \rightarrow \mathbf{c}_t$
Complexity	High	Low (optimized)

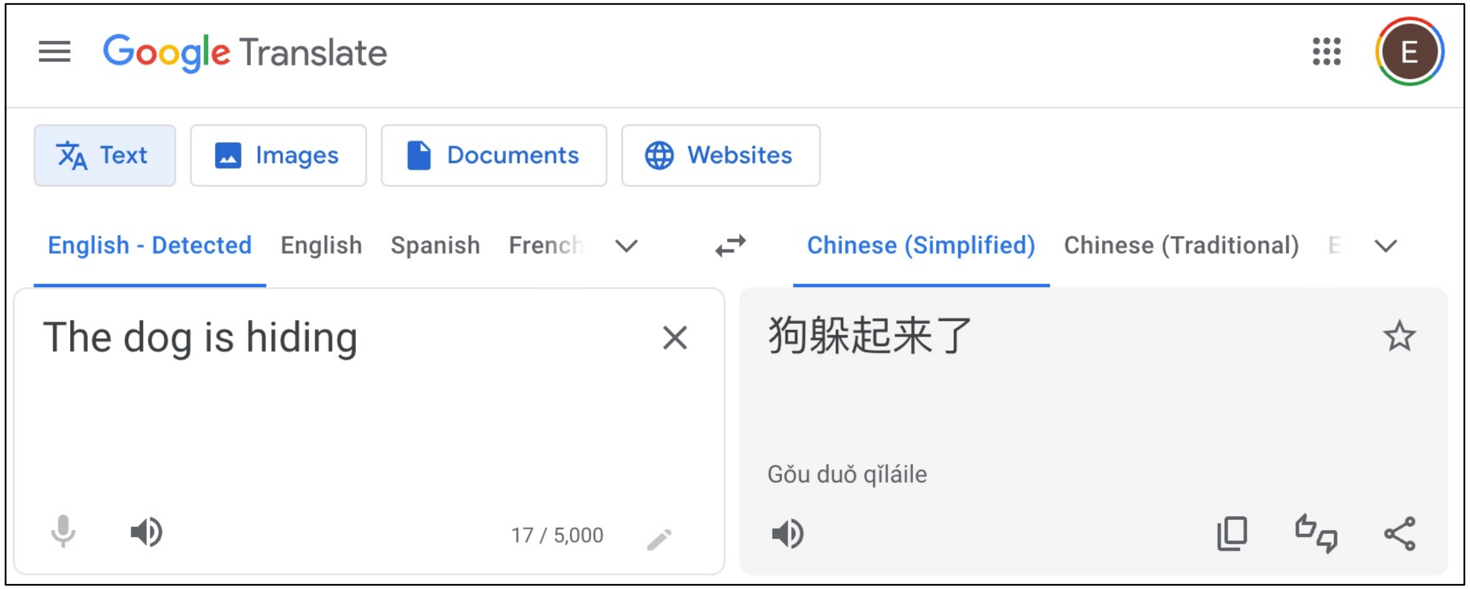
Seq2Seq with Attention vs Traditional Seq2Seq

The BLEU scores of the generated translations on the test set with respect to the lengths of the sentences. The results are on the full test set which includes sentences having unknown words to the models.



Google Neural Machine Translation (GNMT)

In 2016, Google Translate transitioned to a neural machine translation system, which markedly enhanced translation quality, fluency, and accuracy—cutting errors by as much as 60% relative to SMT for certain language pairs.

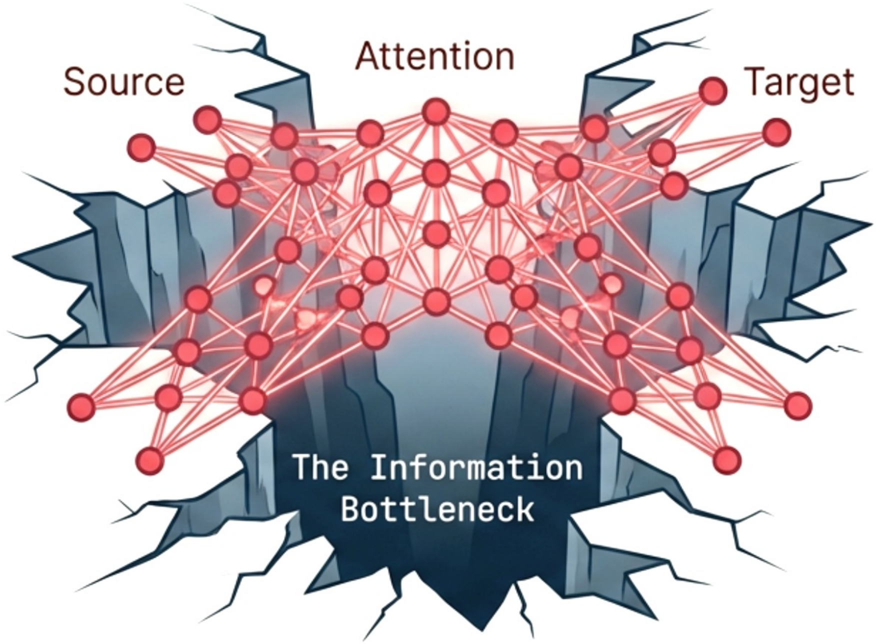


The dog is hiding

狗躲起来了

Gǒu duǒ qǐláile

Breaking the Bottleneck



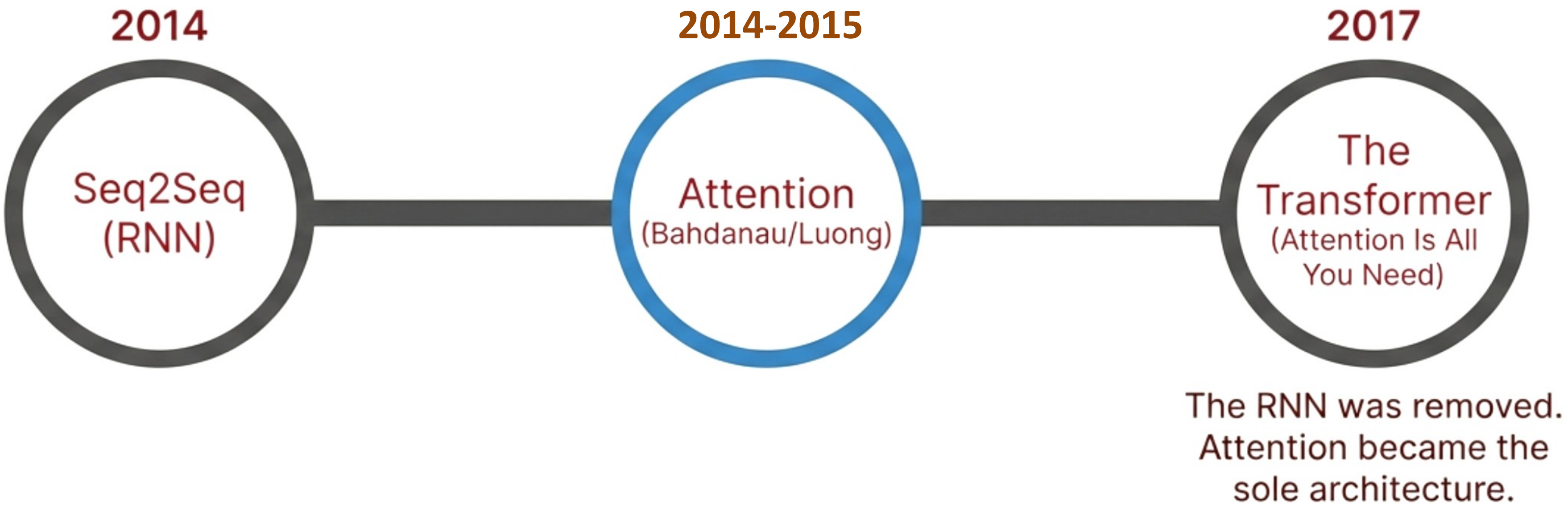
The Flaw: Fixed-length vectors caused catastrophic forgetting in long sequences.

The Fix: Attention mechanisms allowed the model to access the entire input history dynamically.

The Result: Dramatic improvements in translation quality (BLEU) and the ability to handle complex documents.

This concept of **dynamic context** was more than just an upgrade for RNNs; it was the essential steppingstone that directly inspired the self-attention mechanism.

The Path to Transformers



2014

Seq2Seq (RNN)

2014-2015

Attention (Bahdanau/Luong)

2017

The Transformer (Attention Is All You Need)

The RNN was removed. Attention became the sole architecture.

Bahdanau and Luong proved the power of attention. In 2017, researchers took this to its logical conclusion, discarding recurrence entirely to create the Transformer-the architecture behind GPT and modern LLMs.