IMPERIAL

Attention

04/12/2025

Shamsuddeen Muhammad
Google DeepMind Academic Fellow,
Imperial College London
https://shmuhammadd.github.io/

Idris Abdulmumin
Postdoctoral Research Fellow,
DSFSI, University of Pretoria
https://abumafrim.github.io/

Reference

These slides are based on the course material by Daniel Jurafsky :

Chapter 08: https://web.stanford.edu/~jurafsky/slp3/

Bi-directional RNN

Traditional RNNs process sequences in a single direction, typically from past to future.

This is fine for **language modeling**, where predicting the next word only needs left context.

While this approach captures historical dependencies, it may overlook future context, which can be crucial for tasks where understanding both past and future information is essential.

For instance, in speech recognition, accurately interpreting a word can depend on both its preceding and succeeding words.

But for many NLP tasks (e.g., NER, POS tagging, translation, QA), future context is also important.

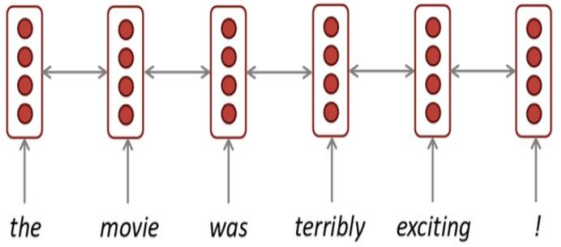
Bi-directional RNN

Bidirectional RNNs extend standard RNNs by processing a sequence in **two directions**:

(1) **Forward** (left → right) and
(2) **Backward** (right → left).

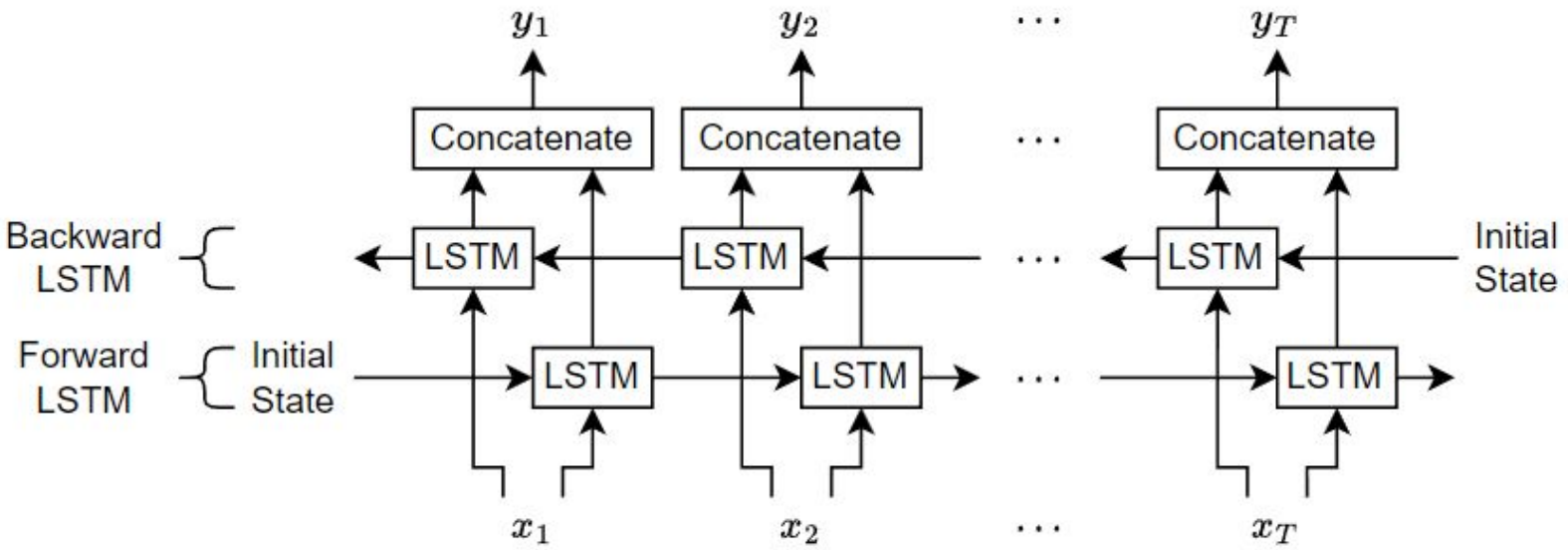
At each time step, the model combines information from both directions, producing a context-rich representation that incorporates **past and future** dependencies.

This contrasts with a unidirectional RNN, which only has access to past tokens.



The two-way arrows indicate bidirectionality and the depicted hidden states are assumed to be the concatenated forwards+backwards states

Bi-directional RNN



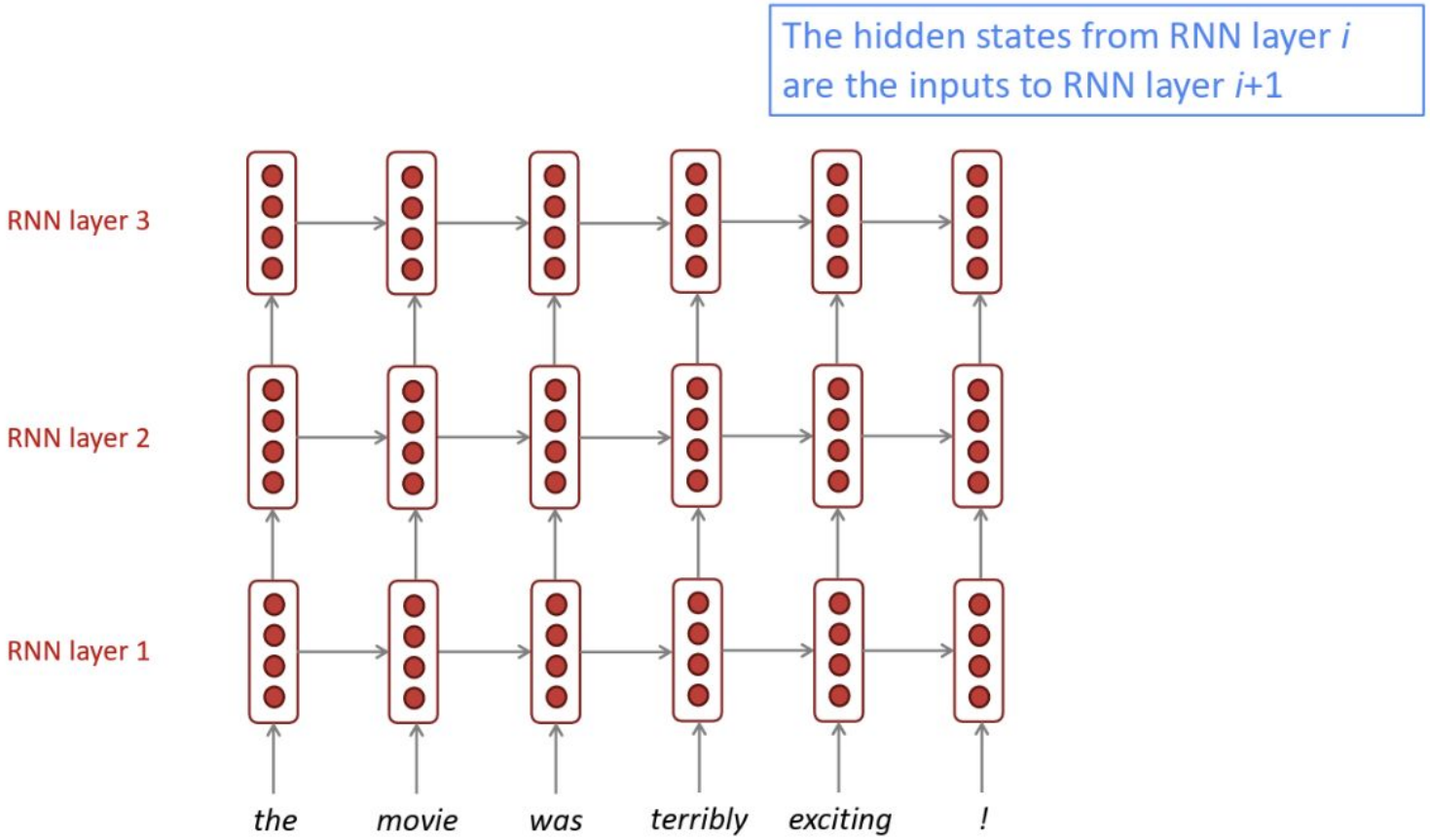
BiLSTM: Bidirectional Long Short-Term Memory

BiGRU: Bidirectional Gated Recurrent Unit

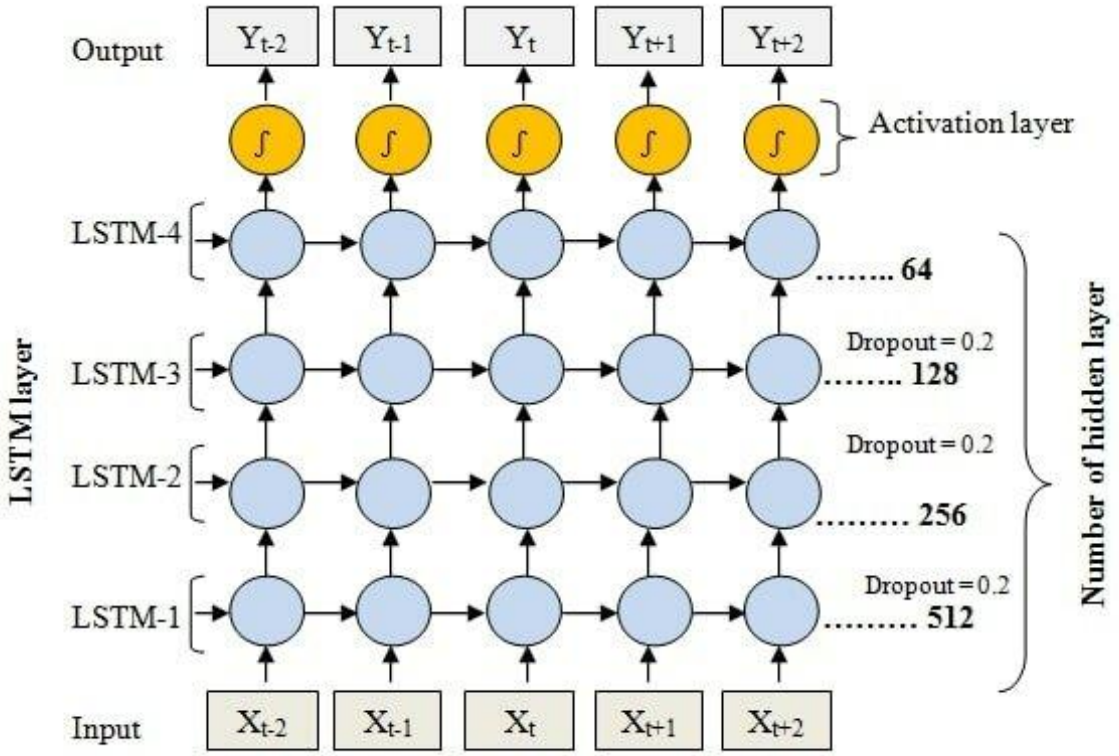
Capture Long-range dependencies

- RNNs are already “deep” on one dimension (they unroll over many timesteps)
- We can also make them “deep” in another dimension by applying multiple RNNs – this is a multi-layer RNN.
- This allows the network to compute more complex representations
  - The lower RNNs should compute lower-level features and the higher RNNs should compute higher-level features.
- Multi-layer RNNs are also called *stacked RNNs*.

Capture Long-range dependencies



Capture Long-range dependencies



Capture Long-range dependencies

Multi-layer or stacked RNNs allow a network to compute more complex representations – they work better than just have one layer of high-dimensional encodings!

- The lower RNNs should compute lower-level features and the higher RNNs should compute higher-level features.

High-performing RNNs are usually multi-layer (but aren't as deep as convolutional or feed-forward networks)

Transformer-based networks (e.g., BERT) are usually deeper, like 12 or 24 layers.

LSTMs: real-world success in

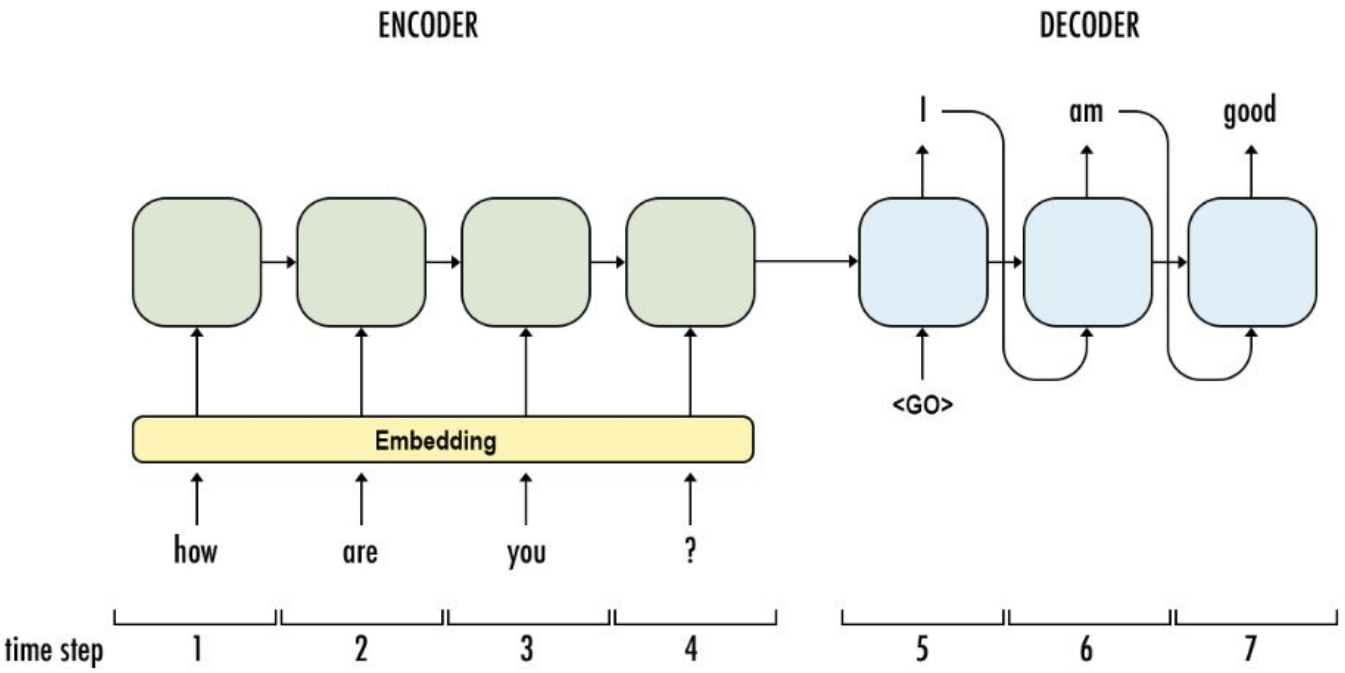
In 2013–2015, LSTMs started achieving state-of-the-art results

- Successful tasks include handwriting recognition, speech recognition, machine translation, parsing, and image captioning, as well as language models
- LSTMs became the dominant approach for most NLP tasks

Now (2019–2024), Transformers have become dominant for all tasks

- For example, in **WMT** (a Machine Translation conference + competition):
  - In WMT 2014, there were 0 neural machine translation systems (!)
  - In WMT 2016, the summary report contains "RNN" 44 times (and these systems won)
  - In WMT 2019: "RNN" 7 times, "Transformer" 105 times

Sequence-to-Sequence Modeling



Sequence-to-Sequence Modeling

Sequence-to-sequence (Seq2Seq) model is fundamentally built on **two components**:

Encoder: An RNN (often LSTM or GRU) reads the input sequence and produces hidden states.

Decoder: another RNN (again LSTM or GRU) uses the encoder's final state to generate the output sequence step-by-step.

Modern Seq2Seq

Later models introduced improvements:

- BiLSTMs in the encoder to capture both left and right context
- Attention mechanism (Bahdanau et al., 2015) to overcome long-range dependency issues
- Transformer-based Seq2Seq (Vaswani et al., 2017) without any RNNs

Seq2Seq Applications

- Machine Translation
- Summarization
- Dialogue Systems
- Speech Recognition
- Image Captioning

Machine Translation

Machine Translation (MT) is the task of translating a sentence *x* from one language (the source language) to a sentence *y* in another language (the target language).

x: *L'homme est né libre, et partout il est dans les fers*

y: *Man is born free, but everywhere he is in chains*

Machine Translation: 1950s

- Machine translation research began in the **early 1950s** on machines less powerful than high school calculators (before the term “A.I.” was coined!).
- MT heavily funded by military, but basically just simple rule-based systems doing word substitution.
- Human language is more complicated than that, and varies more across languages!
- Little understanding of natural language syntax, semantics, pragmatics.

Machine Translation: 1950s

- Rule-Based MT (RBMT)
- Statistical MT (SMT)
- Phrase-Based SMT

Limitations: rigid rules, feature engineering, weak generalization

2014 Neural Machine Translation

The 2014 work on **Neural Machine Translation (NMT)** marks a pivotal shift in machine translation (MT), introducing end-to-end neural architectures that replaced traditional statistical MT systems.

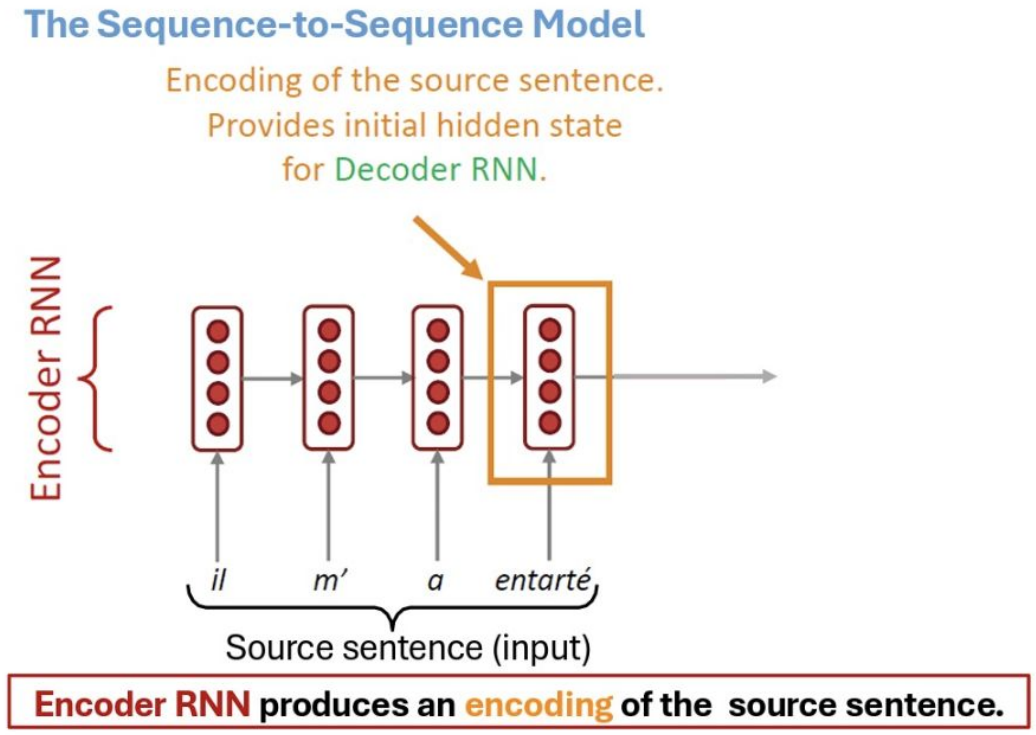
Two landmark papers introduced foundational ideas:

Sequence-to-Sequence Learning with Neural Networks (Sutskever et al., 2014, NIPS): *Demonstrated that a single neural network could be trained end-to-end for translation.*

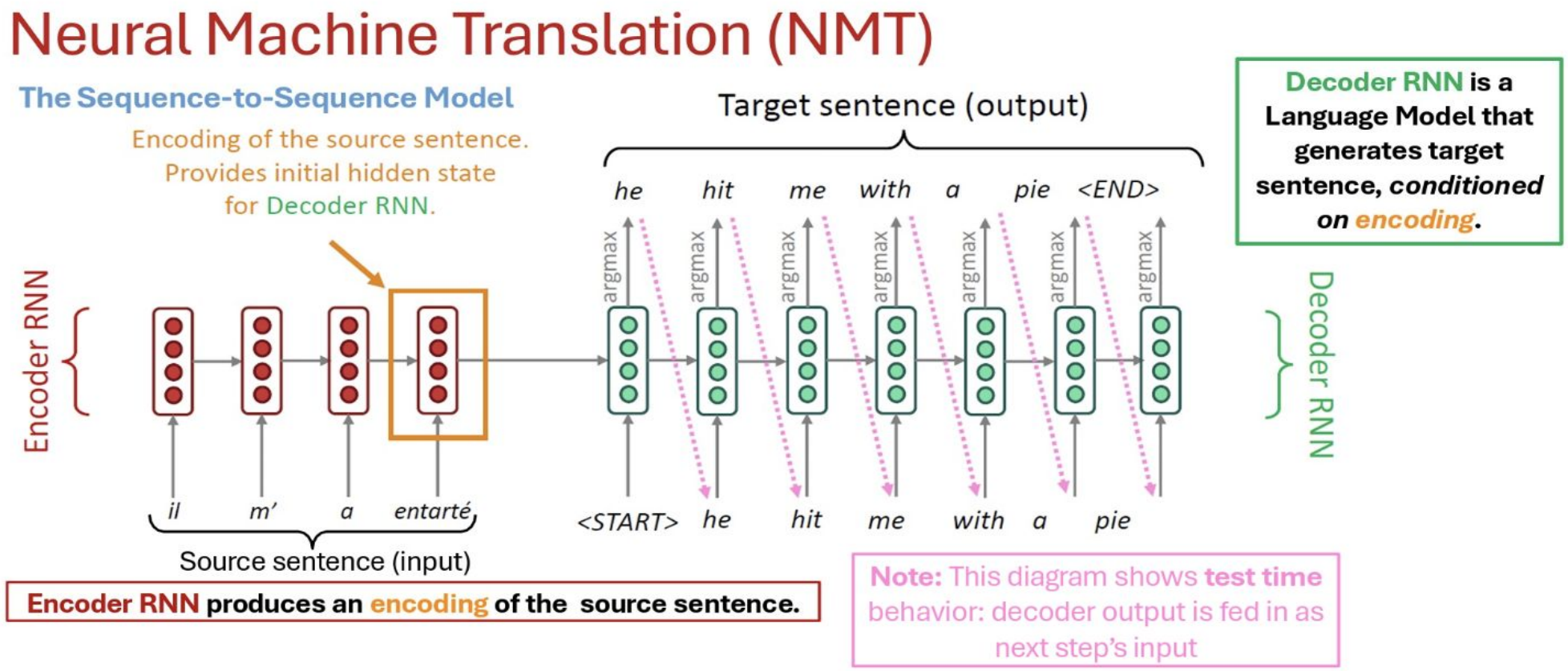
Neural Machine Translation by Jointly Learning to Align and Translate (Bahdanau et al., 2014, ICLR 2015)
Introduced the attention mechanism, allowing the decoder to dynamically focus on different parts of the input sequence at each output step.

Neural Machine Translation

Neural Machine Translation



Neural Machine Translation



Neural Machine Translation

- The general notion here is an encoder-decoder model
  - One neural network takes input and produces a neural representation
  - Another network produces output based on that neural representation
  - If the input and output are sequences, we call it a seq2seq model

Neural Machine Translation

- The sequence-to-sequence model is an example of a **Conditional Language Model**
  - **Language Model** because the decoder is predicting the next word of the target sentence *y*
  - **Conditional** because its predictions are *also* conditioned on the source sentence *x*

- NMT directly calculates $P(y|x)$:

$$P(y|x) = P(y_1|x)\, P(y_2|y_1, x)\, P(y_3|y_1, y_2, x) \ldots \underbrace{P(y_T|y_1, \ldots, y_{T-1}, x)}$$

Probability of next target word, given target words so far and source sentence *x*

- **Question:** How to train an NMT system?
- **(Easy) Answer:** Get a big parallel corpus…
  - But there is now exciting work on "unsupervised NMT", data augmentation, etc.

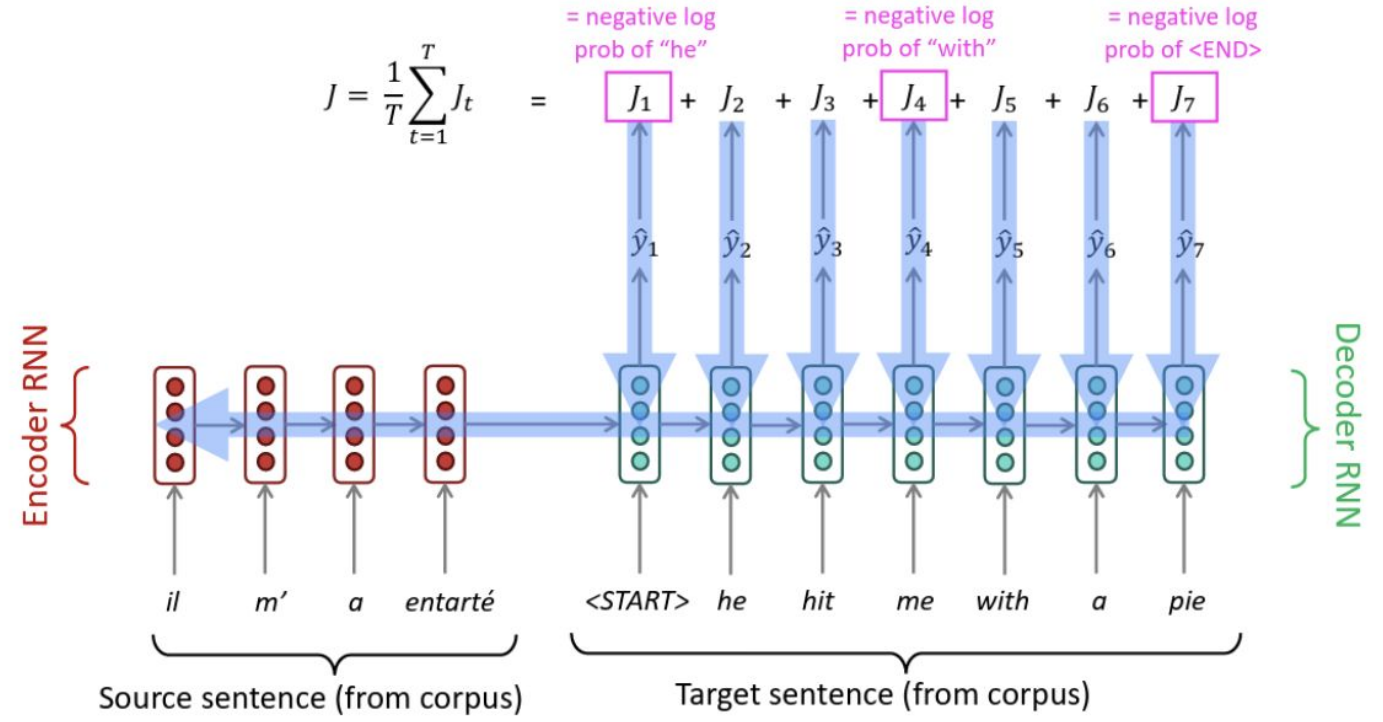
Example of Data Augmentation

Round-Trip Translation (Dual Learning): Translate source to target and then back to source.

Cross-Lingual Word Substitution: Replace words with synonyms or transliterations from another language using bilingual dictionaries or word embeddings.

Back-translation is a method where monolingual data from the target language is translated *back* into the source language using a reverse translation model (i.e., target → source). The resulting synthetic parallel data is then used to train the source → target NMT system.

Training a Neural Machine Translation



Seq2seq is optimized as a **single system.** Backpropagation operates "*end-to-end*".

Decoding

After training a sequence-to-sequence model (e.g., for machine translation), **decoding** refers to how we **generate the output sequence** (e.g., translated sentence) from the model.

At each step, the decoder:

- Takes in previous generated words
- Outputs a probability distribution over the vocabulary
- Selects the next word

The choice of decoding strategy significantly affects translation quality.

Decoding Strategies

- Greedy Decoding
- Beam Search Decoding
- Sampling-Based Decoding

NMT: the first big success story of NLP Deep Learning

Neural Machine Translation went from a fringe research attempt in **2014** to the leading standard method in **2016**

- **2014**: First seq2seq paper published [Sutskever et al. 2014]
- **2016**: Google Translate switches from SMT to NMT – and by 2018 everyone had
  - https://www.nytimes.com/2016/12/14/magazine/the-great-ai-awakening.html

- This was amazing!
  - **SMT** systems, built by hundreds of engineers over many years, were outperformed by NMT systems trained by small groups of engineers in a few months

Problems with RNNs (again)

- Linear interaction distance
- Bottleneck problem
- Lack of parallelizability

Linear Interact Distance

Linear Interaction Distance: RNNs model sequences in a strictly linear, step-by-step fashion. To relate information from distant positions (e.g., the first and last word of a long sentence), the signal must traverse each intermediate timestep.

Consequence: Long-range dependencies are difficult to learn effectively because of **gradient vanishing/exploding** and limited memory capacity.

Lack of Parallelizability

Lack of Parallelizability: RNNs must process inputs sequentially — each hidden state depends on the previous one.

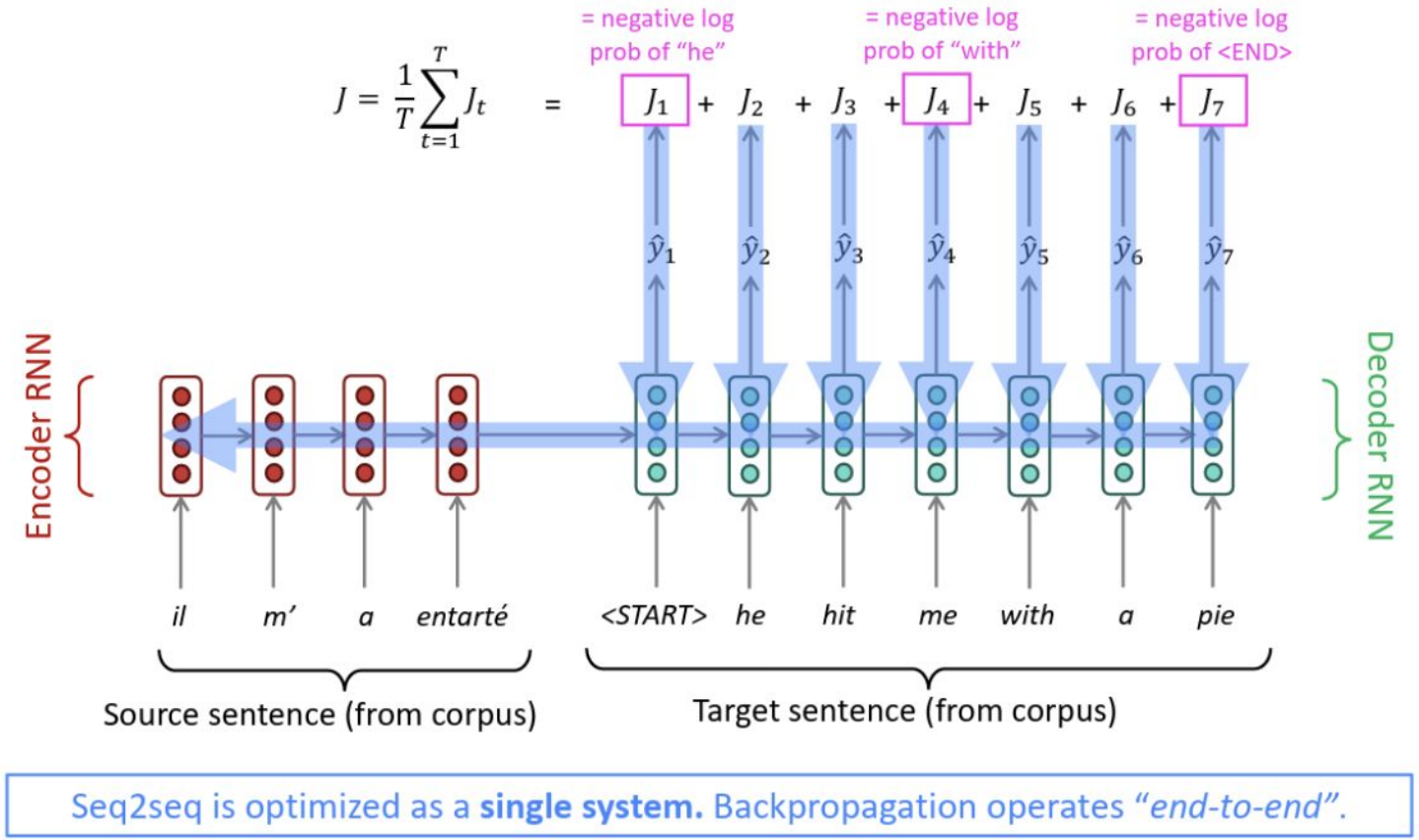
Consequence: Unlike CNNs or Transformers, RNNs **cannot be parallelized across time steps**, making training and inference **slow** and less scalable on modern hardware (e.g., GPUs/TPUs).

Bottleneck Problem

Bottleneck Problem Issue: In encoder-decoder RNNs (e.g., Seq2Seq), the entire input sequence is compressed into a fixed-size **context vector**.

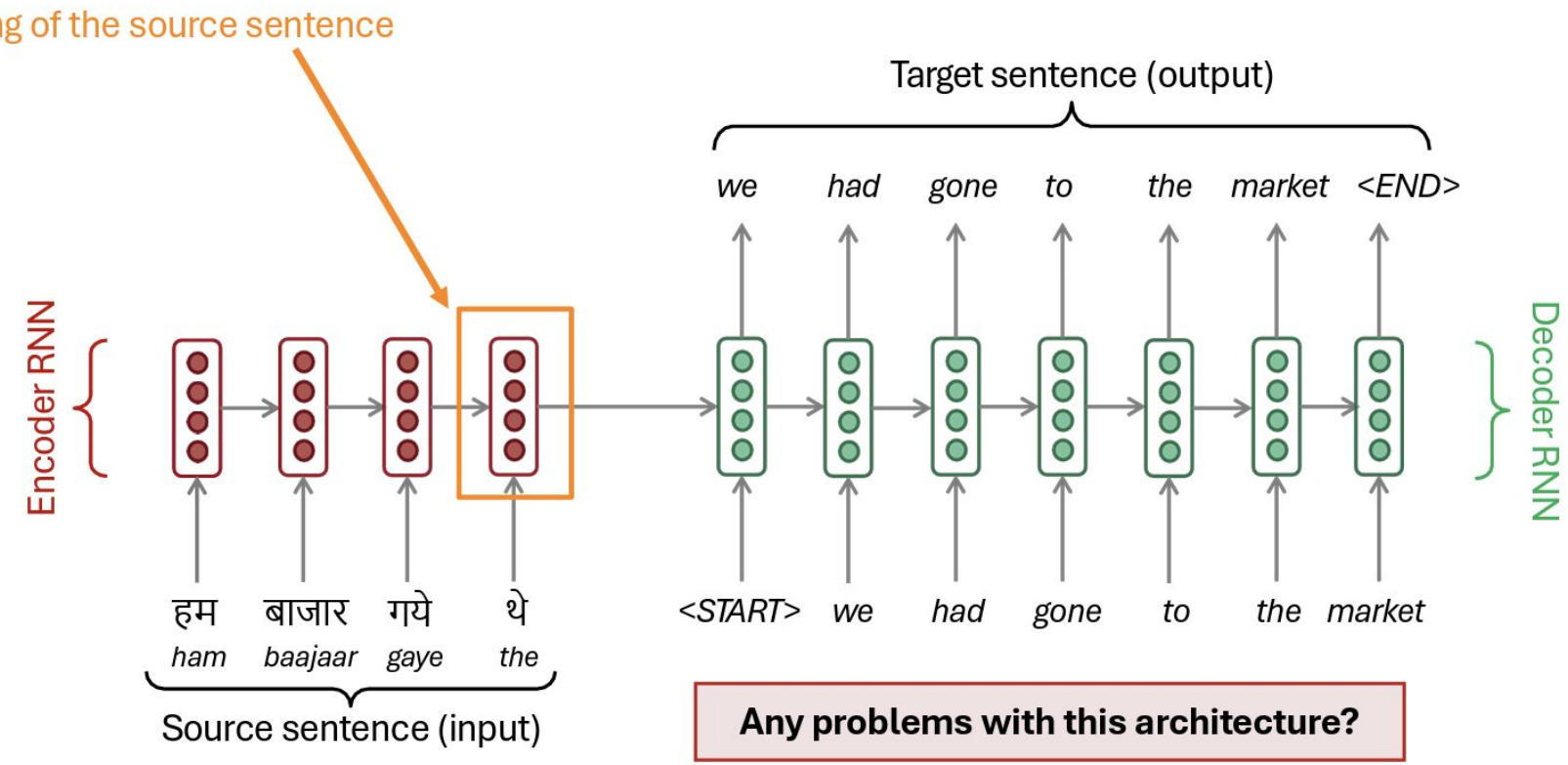
Consequence: This "bottleneck" limits the amount of information passed from encoder to decoder, especially for long input sequences, leading to poor performance in tasks like translation or summarization.

Bottleneck Problem

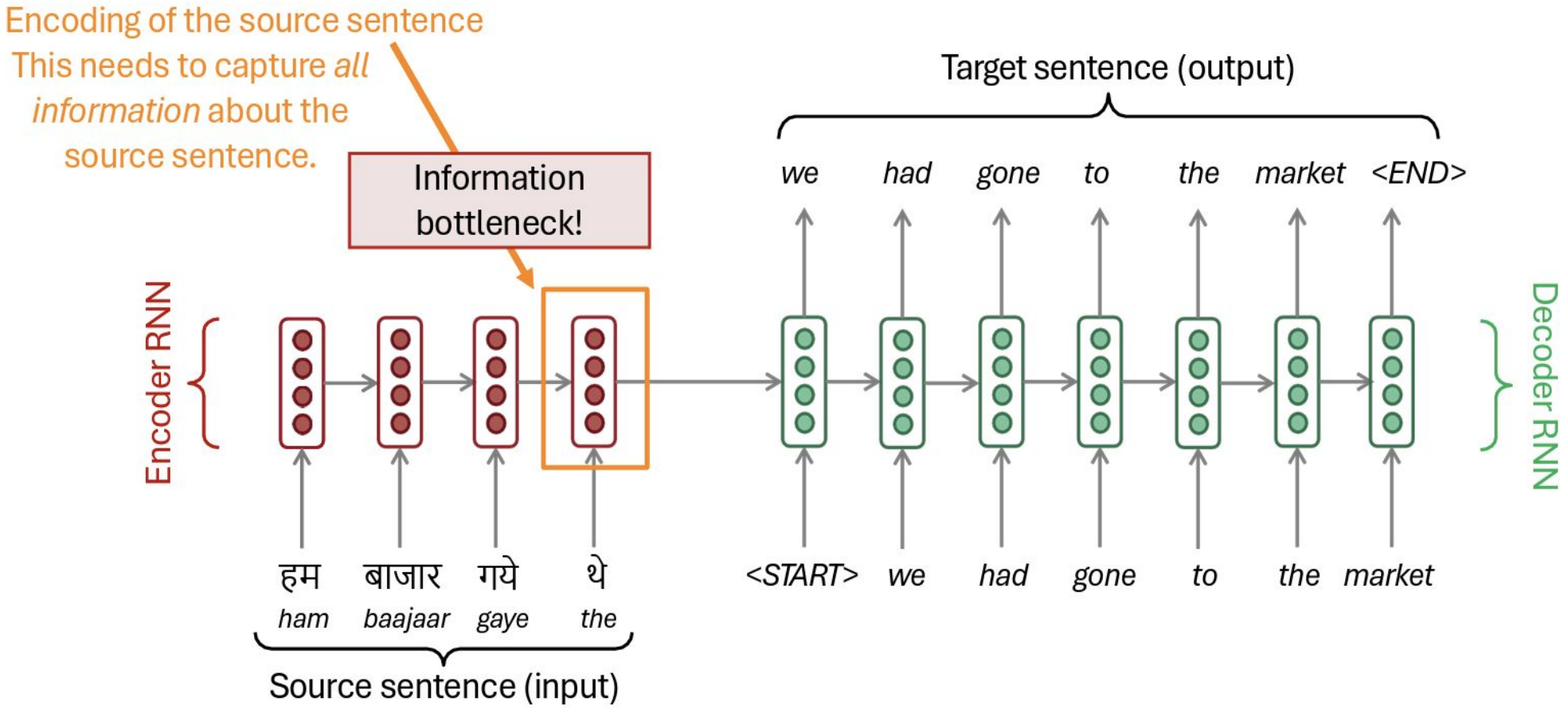


Seq2seq is optimized as a **single system.** Backpropagation operates "*end-to-end*".

Sequence-to-Sequence: The Bottleneck Problem



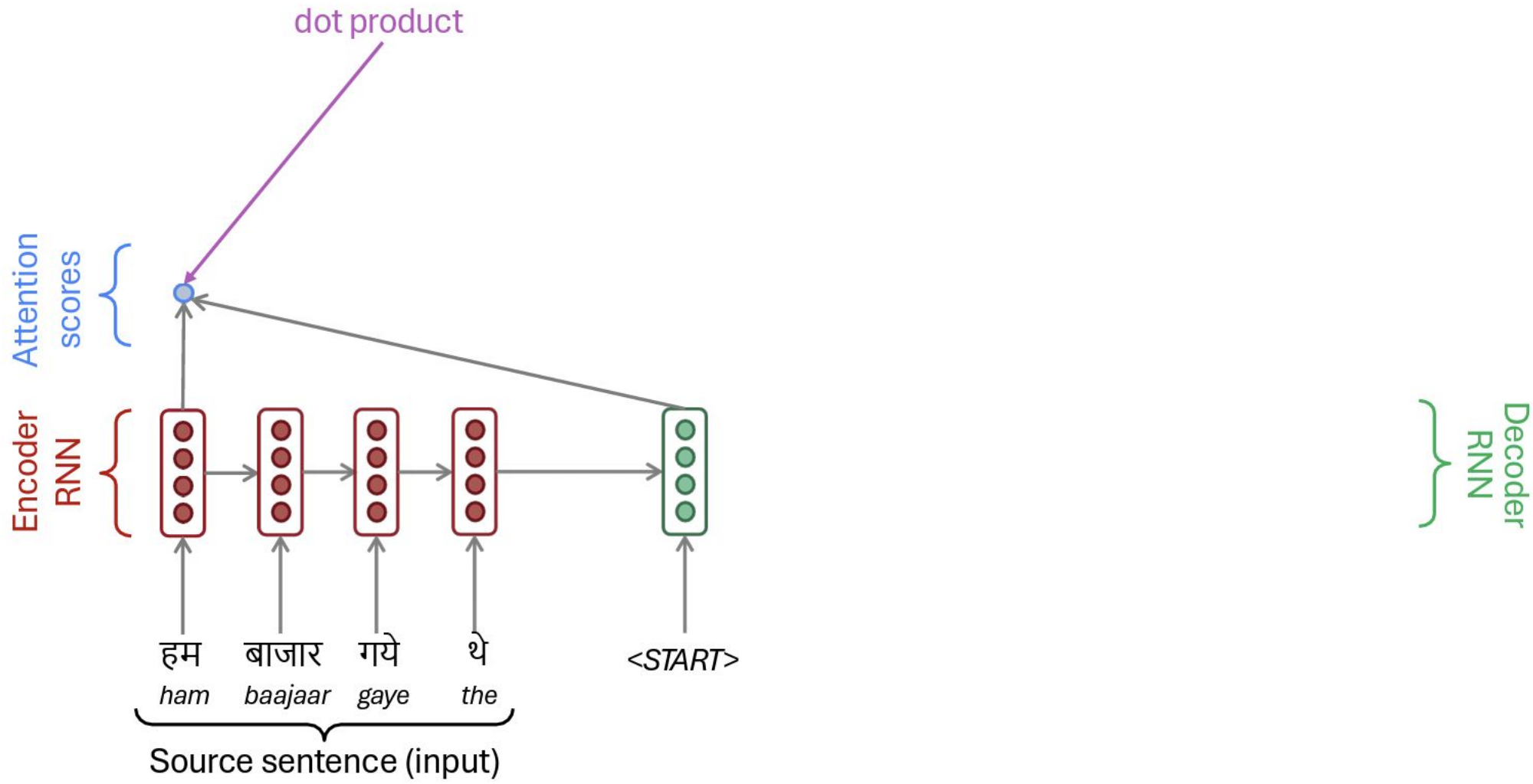
Sequence-to-Sequence: The Bottleneck Problem



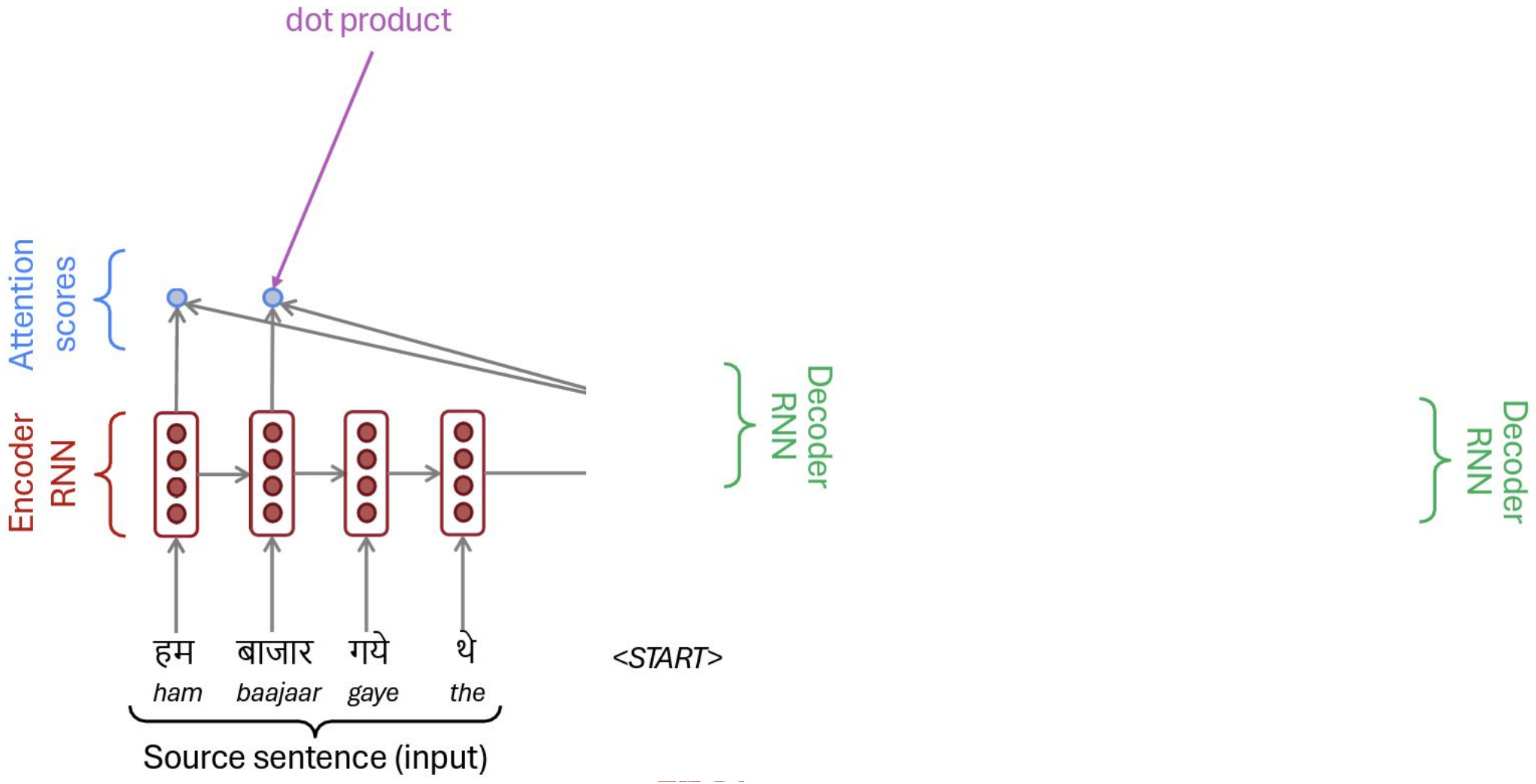
Attention

- **Attention** provides a solution to the bottleneck problem.
- **Core idea**: on each step of the decoder, use direct connection to the encoder to focus on a particular part of the source sequence
- Let's start with the visualization of the attention mechanism.

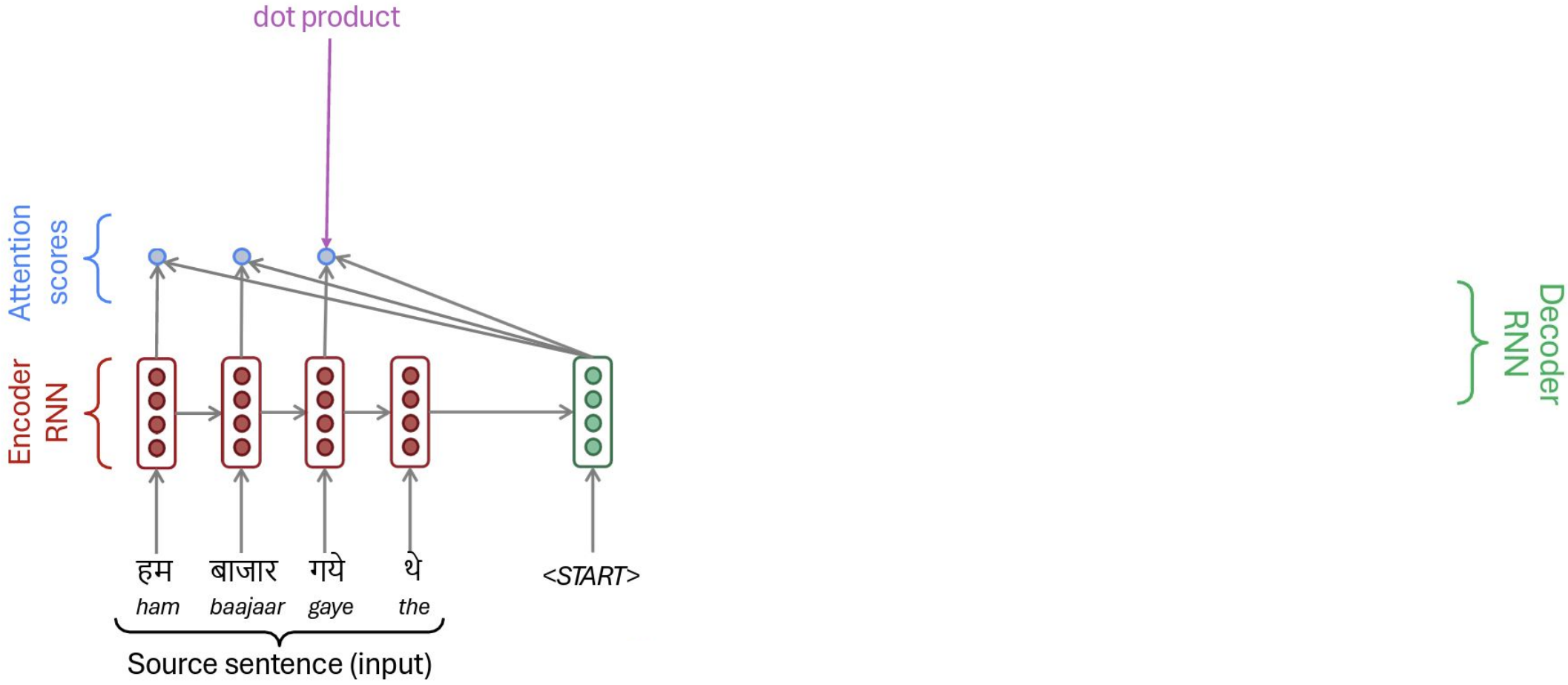
Sequence-to-Sequence With Attention



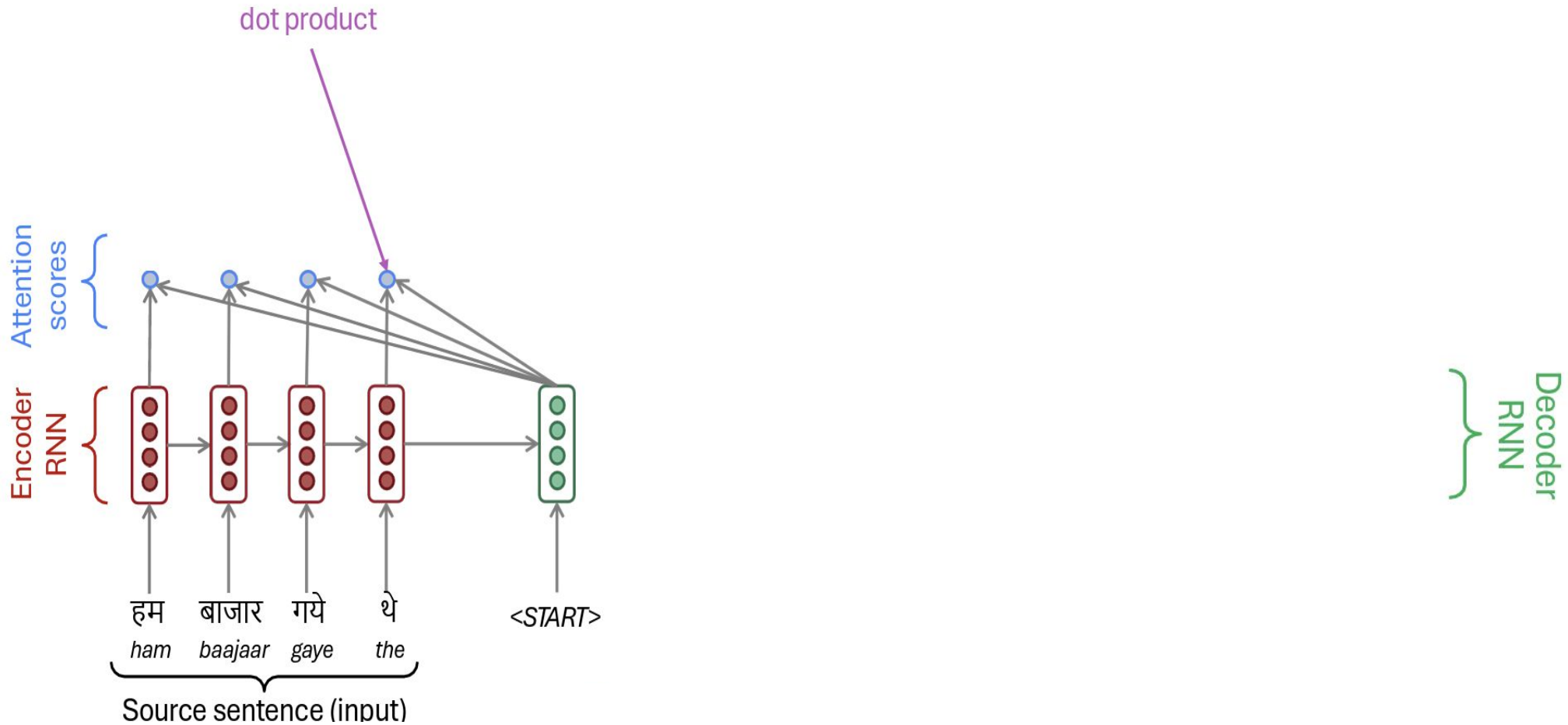
Sequence-to-Sequence With Attention



Sequence-to-Sequence With Attention

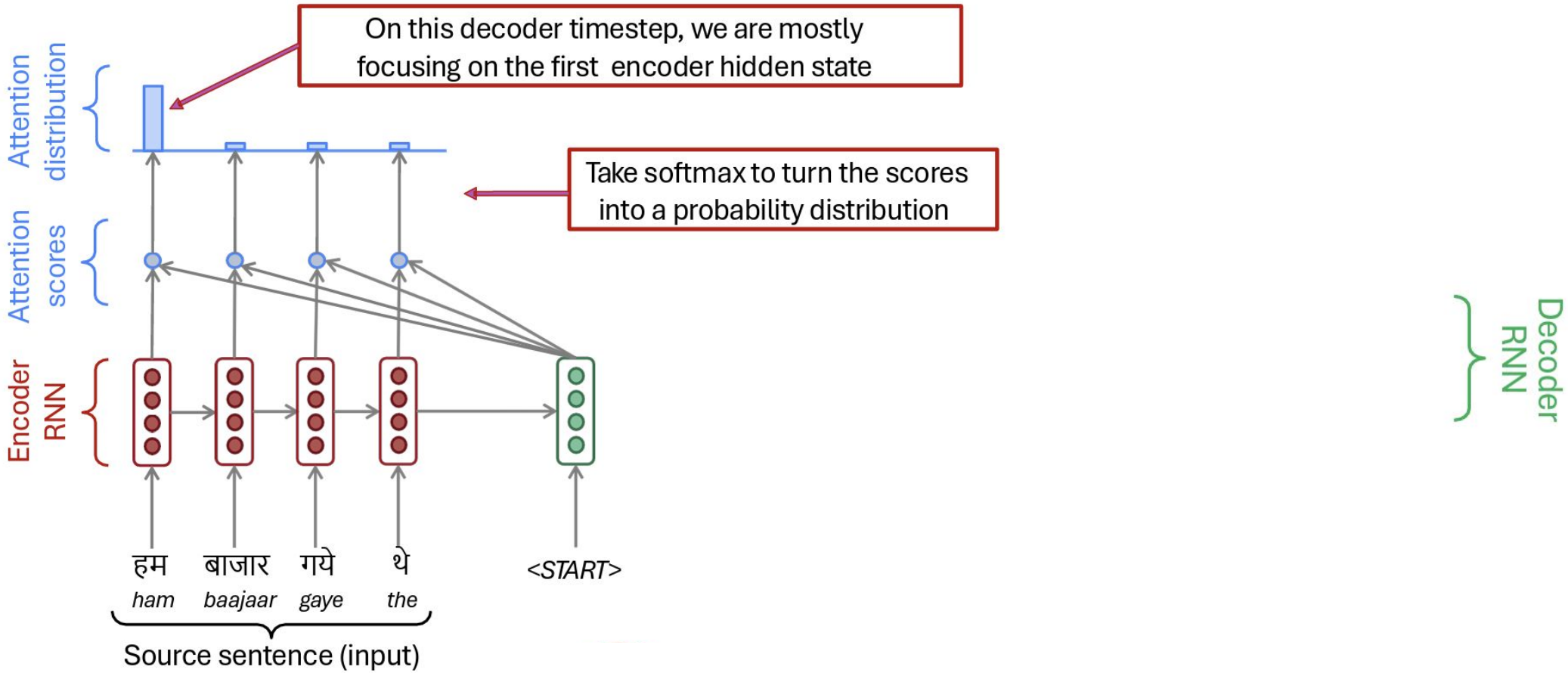


Sequence-to-Sequence With Attention



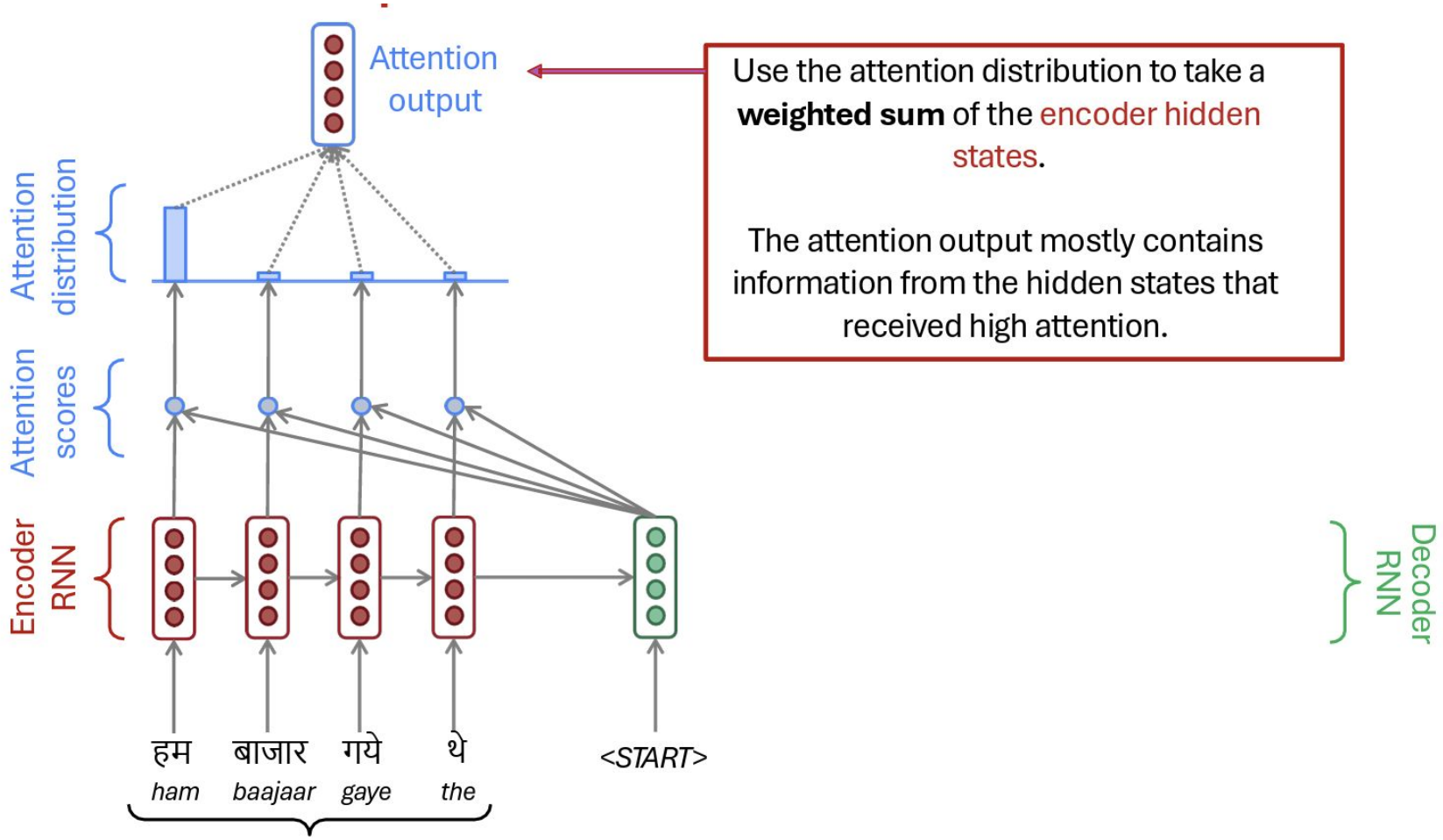
Sequence-to-Sequence With Attention

Attention vector



Assign attention weight to each word, to know how much "attention" the model should pay to each word (i.e., for each word, the network learns a "context")

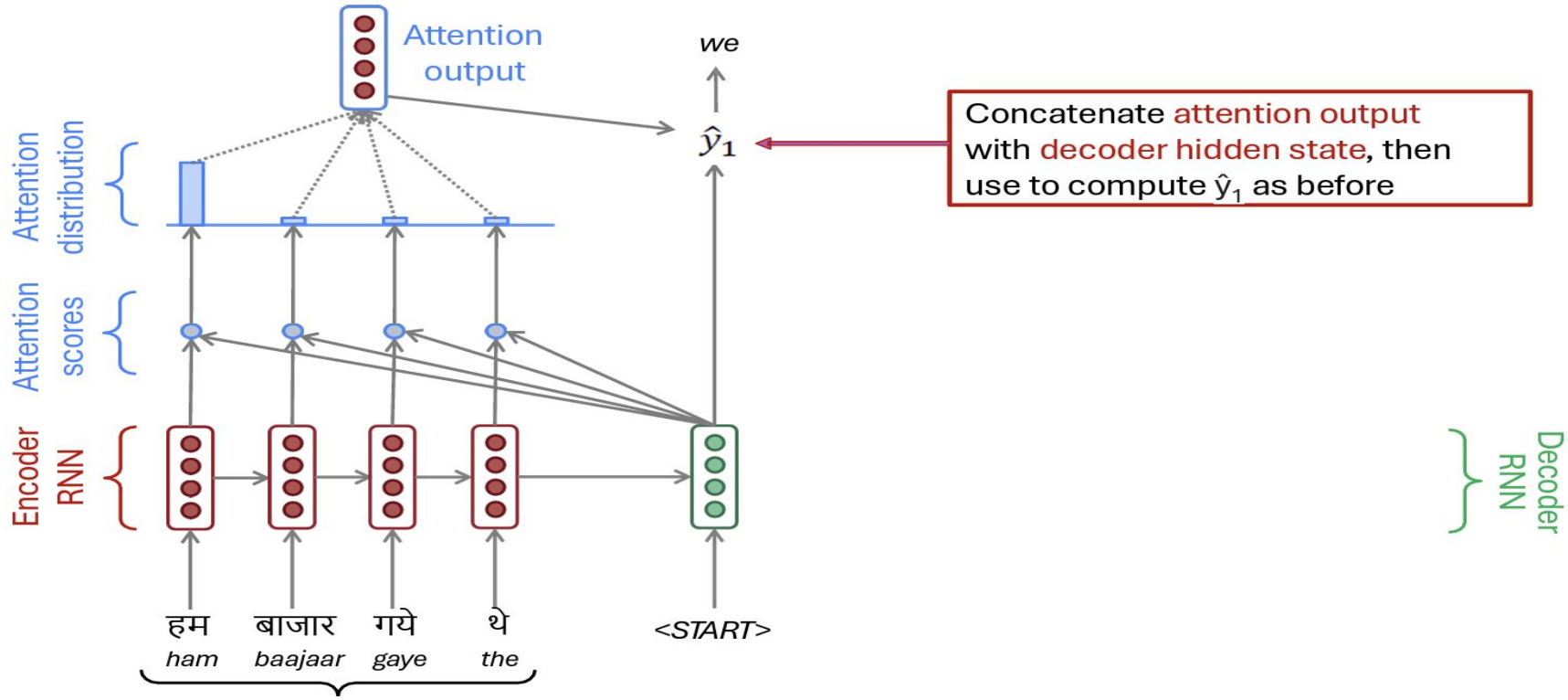
Sequence-to-Sequence With Attention



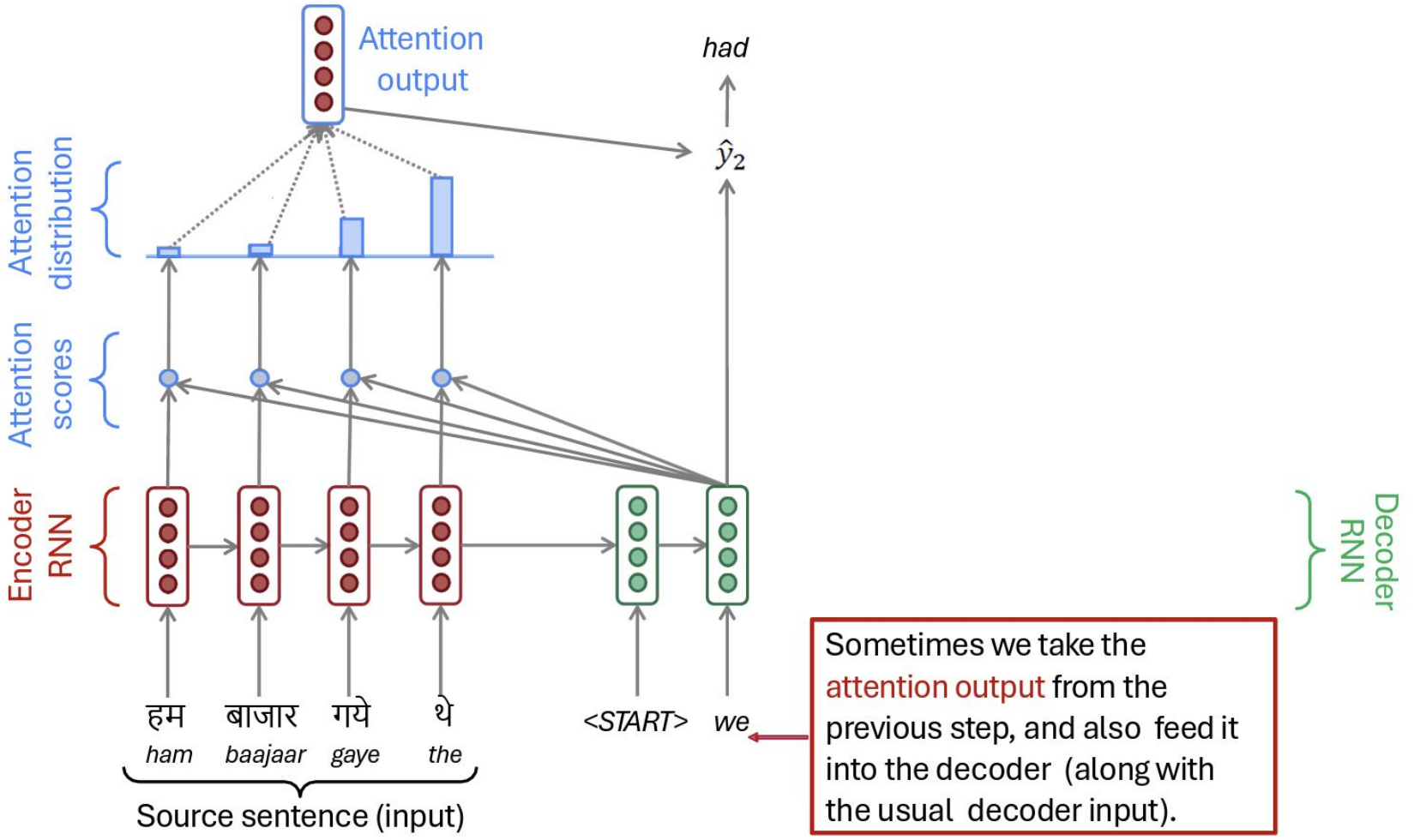
Use the attention distribution to take a **weighted sum** of the encoder hidden states.

The attention output mostly contains information from the hidden states that received high attention.

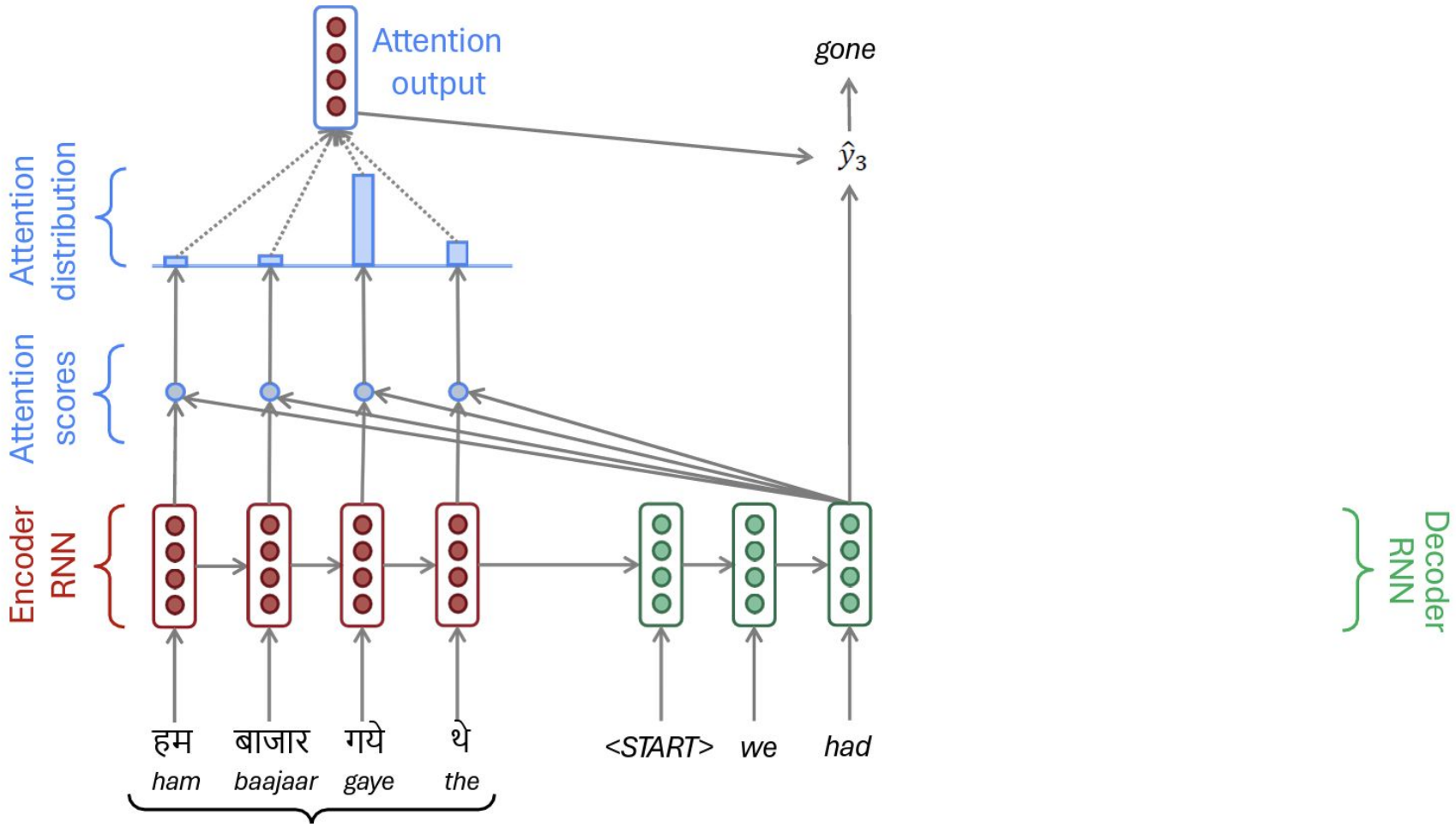
Sequence-to-Sequence With Attention



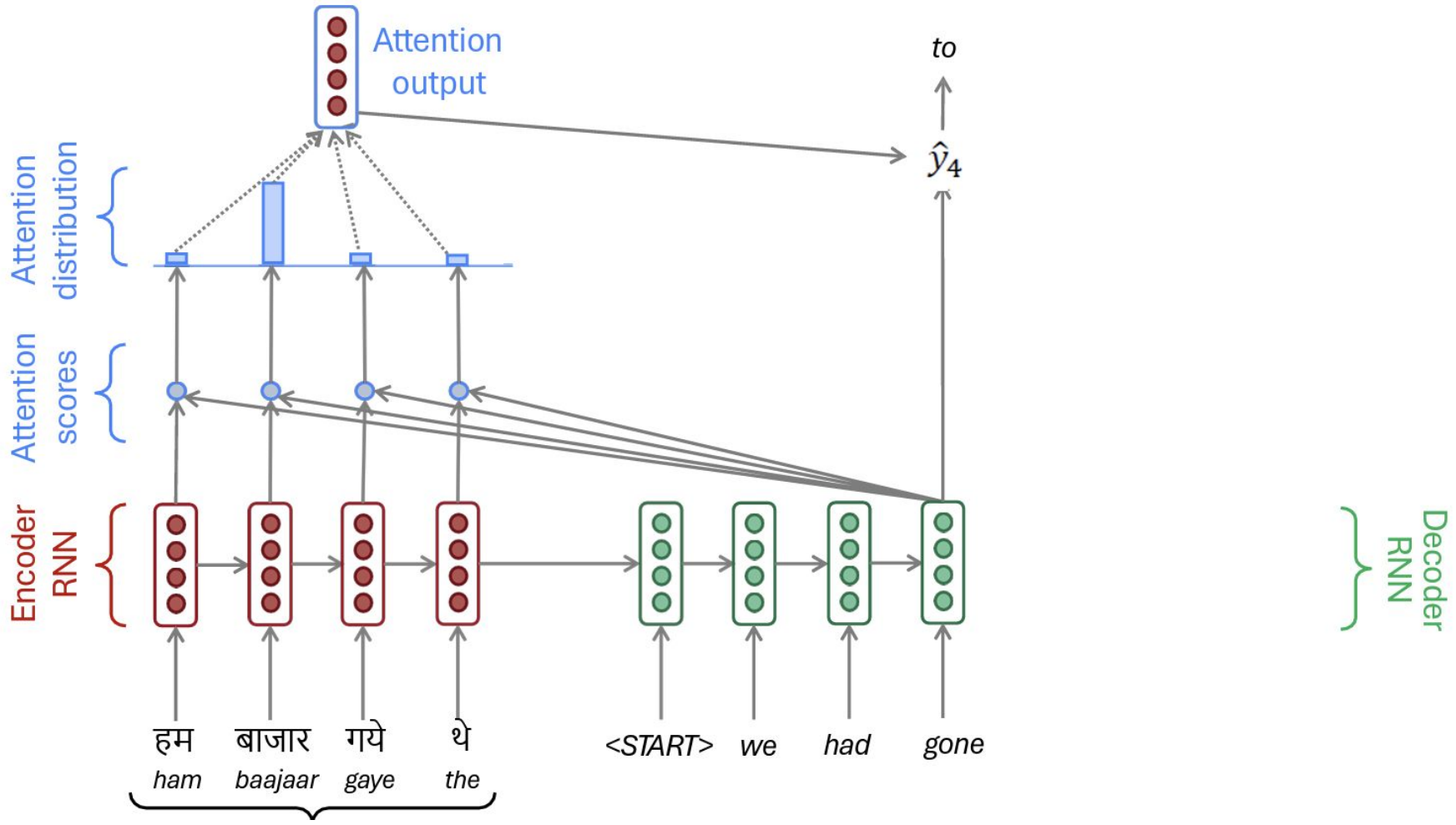
Sequence-to-Sequence With Attention



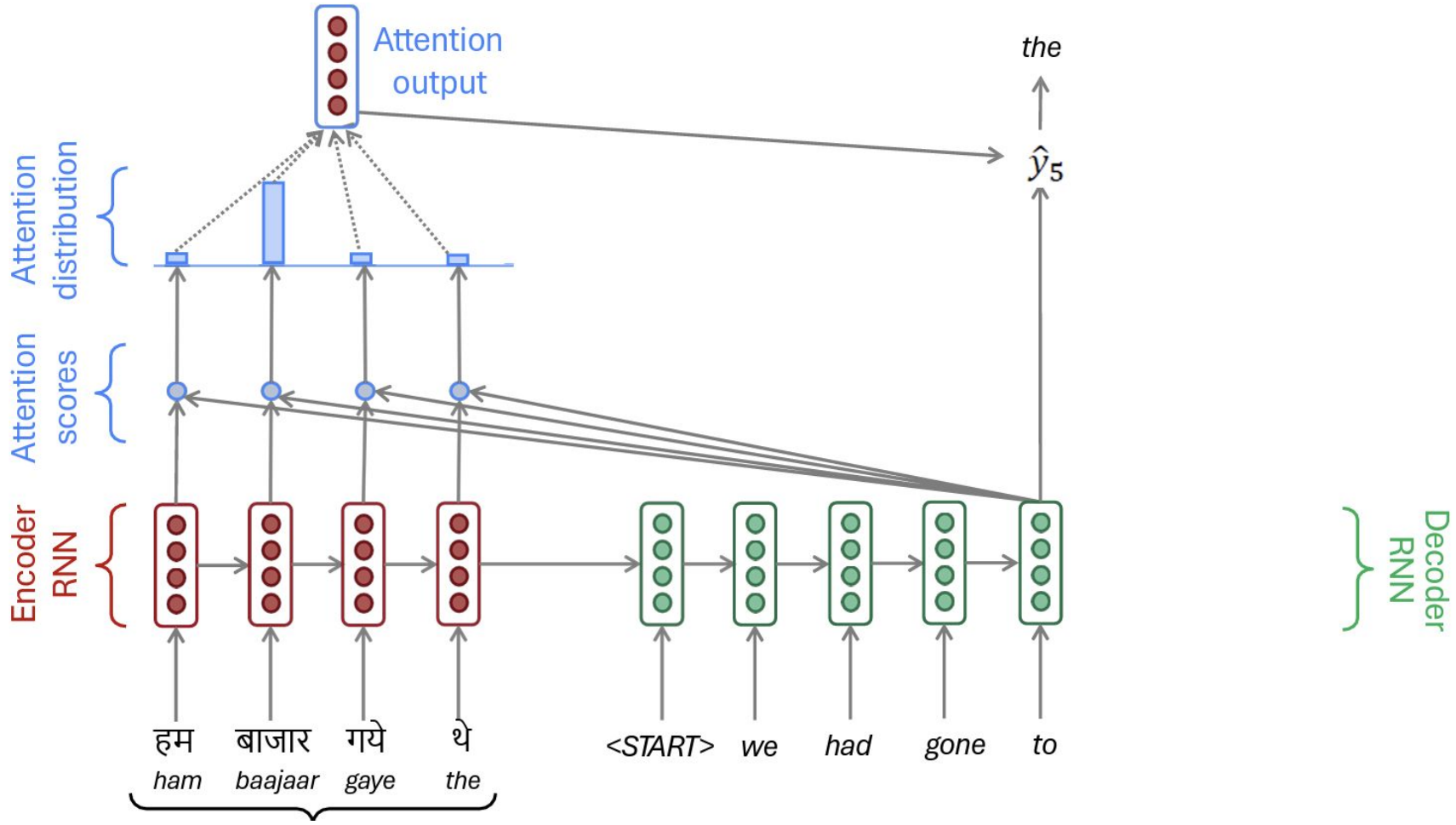
Sequence-to-Sequence With Attention



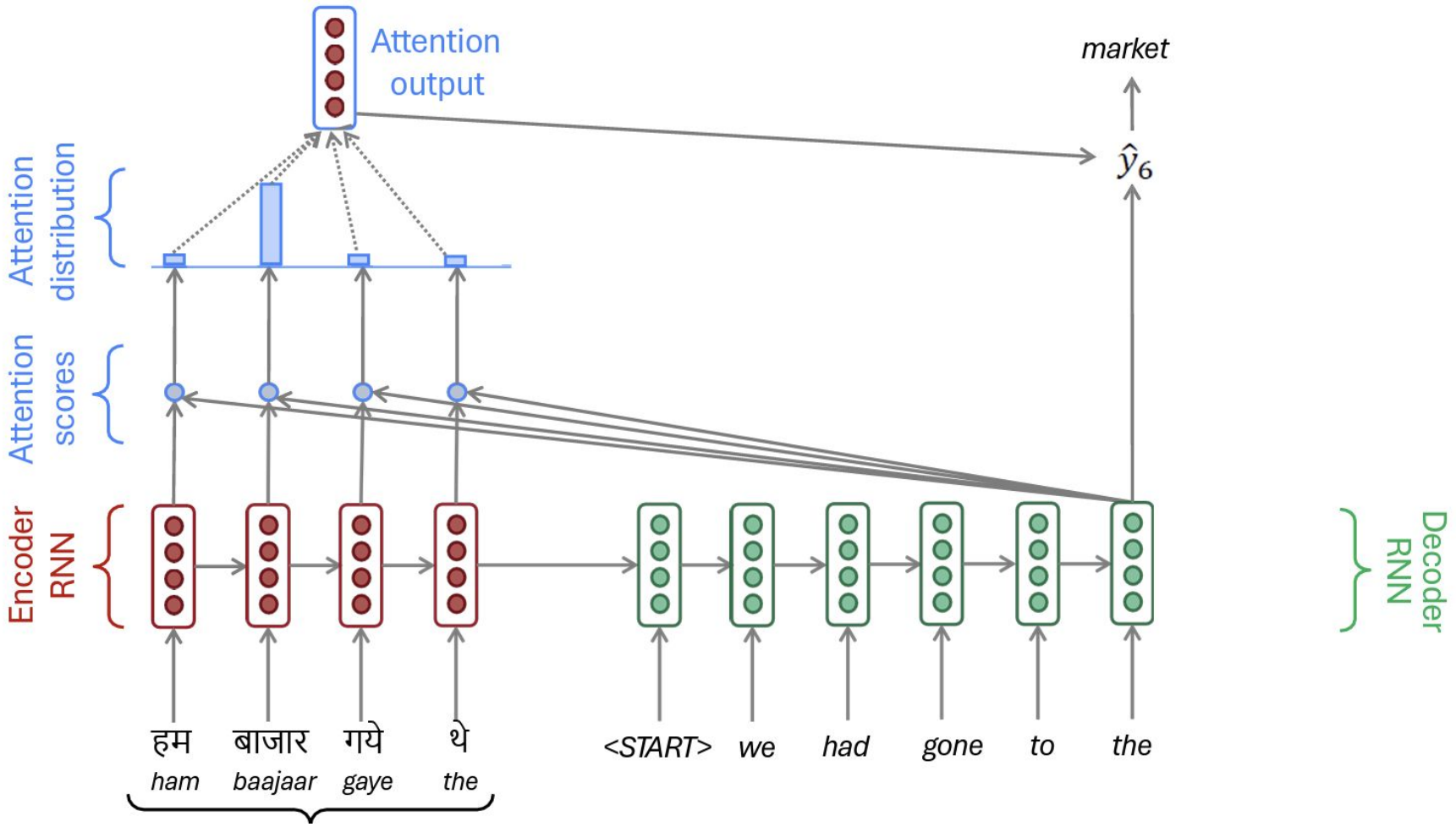
Sequence-to-Sequence With Attention



Sequence-to-Sequence With Attention

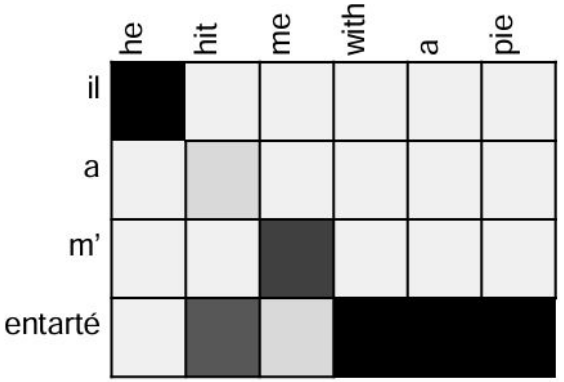


Sequence-to-Sequence With Attention



Attention is Great

- Attention significantly improves NMT performance
  - It's very useful to allow decoder to focus on certain parts of the source
- Attention solves the bottleneck problem
  - Attention allows decoder to look directly at source; bypass bottleneck
- Attention helps with vanishing gradient problem
  - Provides shortcut to faraway states
- Attention provides some interpretability
  - By inspecting attention distribution, we can see what the decoder was focusing on
  - We get (soft) alignment for free!
  - This is cool because we never explicitly trained an alignment system
  - The network just learned alignment by itself



Attention Mechanism

- Originally developed for language translation:
  Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. https://arxiv.org/abs/1409.0473

"... allowing a model to automatically (soft-)search for parts of a source sentence that are relevant to predicting a target word ..."

"traditional" encoder+decoder RNN

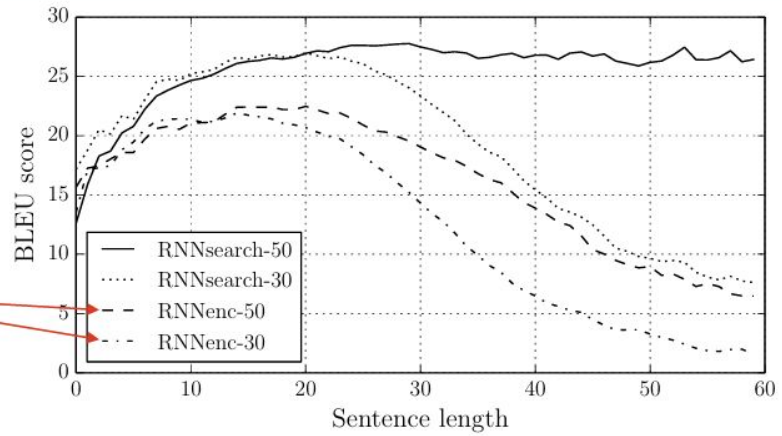


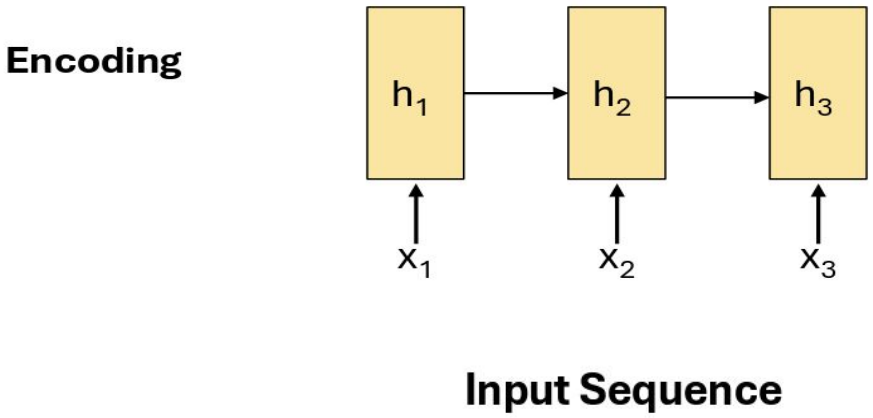
Figure 2: The BLEU scores of the generated translations on the test set with respect to the lengths of the sentences. The results are on the full test set which includes sentences having unknown words to the models.

Attention is a General Deep Learning Technique

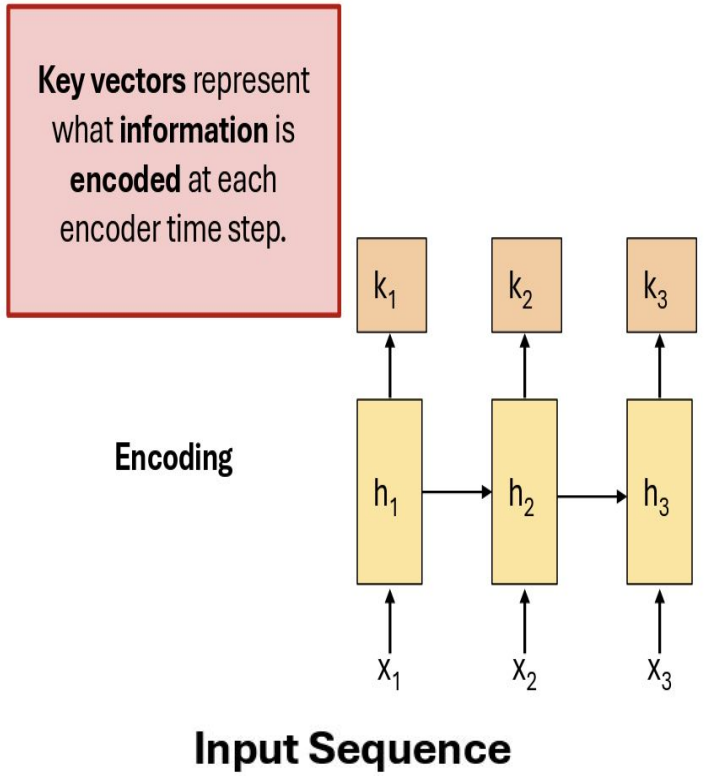
- We've seen that attention is a great way to improve the sequence-to-sequence model for Machine Translation.
- However: You can use attention in many architectures (not just seq2seq) and many tasks (not just MT)
- More general definition of attention:
  - Given a set of vector *values*, and a vector *query*, attention is a technique to compute a weighted sum of the values, dependent on the query.
- We sometimes say that the query *attends to* the values.
- For example, in the seq2seq + attention model, each decoder hidden state (query) *attends to* all the encoder hidden states (values).
- **Intuition**:
  - The weighted sum is a *selective summary* of the information contained in the values, where the query determines which values to focus on.
  - Attention is a way to obtain a *fixed-size representation of an arbitrary set of representations* (the values), dependent on some other representation (the query).

Is Attention All we need?

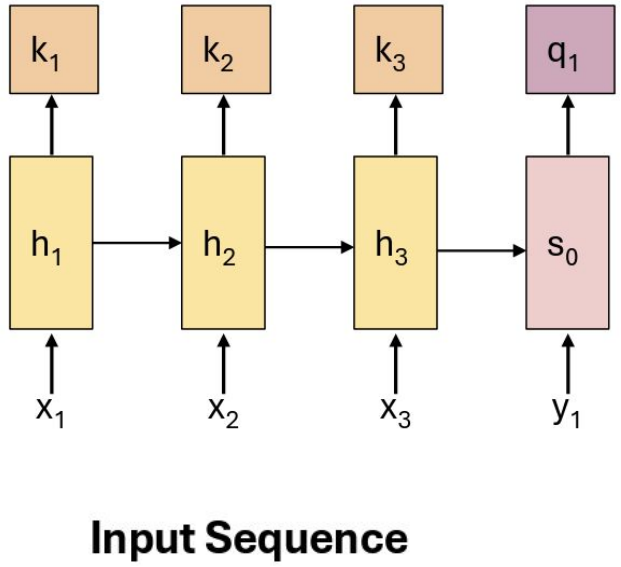
Attention



Attention

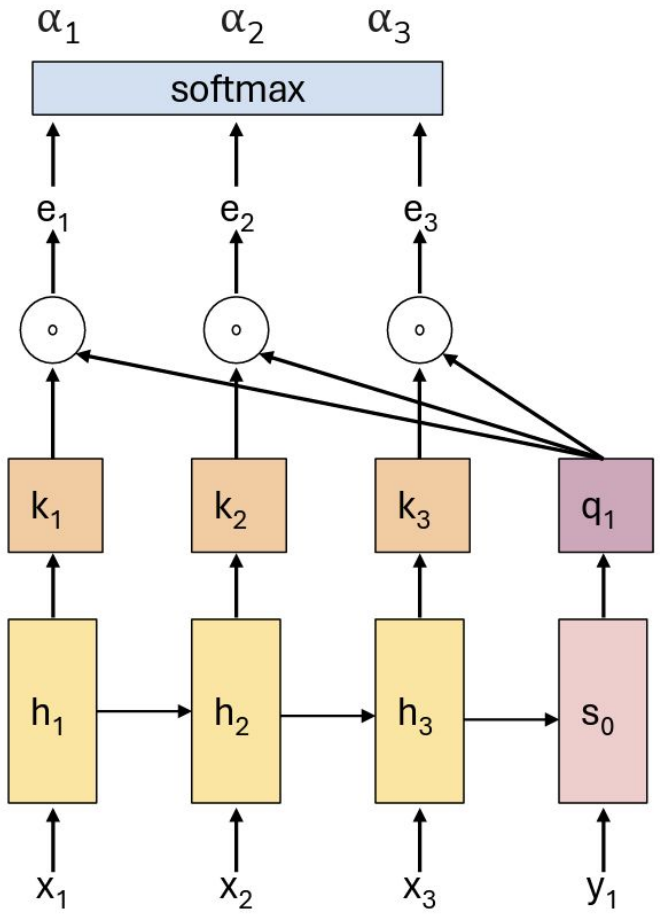

Key vectors represent what information is encoded at each encoder time step.
Encoding
k_1
k_2
k_3
h_1
h_2
h_3
x_1
x_2
x_3
Input Sequence

Attention



Query vectors represent what information we are **looking for** at each decoder time step.

Attention

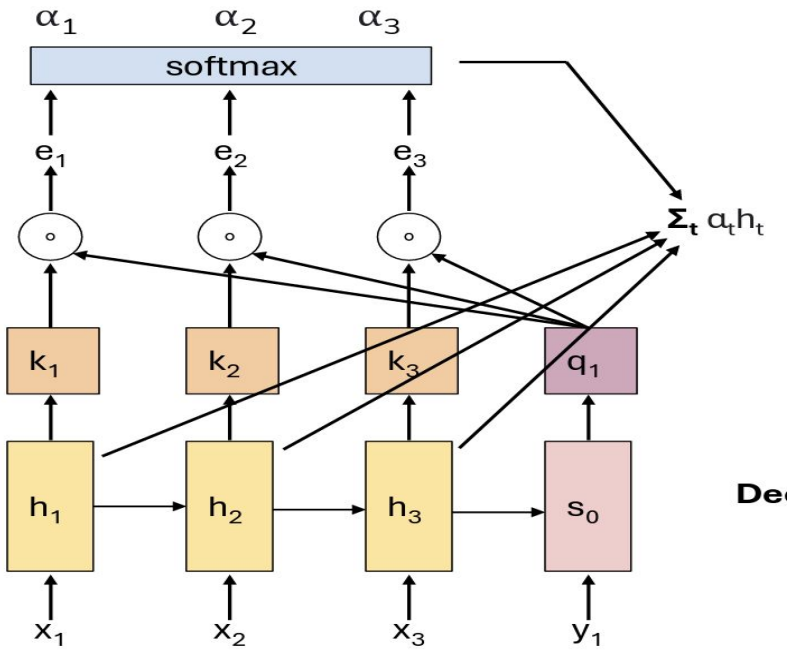


Softmax converts the similarity scores into a **probability distribution**.

Dot product between query vector and every key vector gives **similarity score**.

Decoding

Attention



The output of attention mechanism is the **weighted sum** of hidden vectors.

Instead of simply summing up the hidden vectors, we can transform them using a learned function to generate **value vectors** and then compute a weighted sum.

Decoding

Variants of Attention

Original formulation: $a(\mathbf{q}, \mathbf{k}) = w_2^T \tanh(W_1[\mathbf{q}; \mathbf{k}])$

Bilinear product: $a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^T W \mathbf{k}$ Luong et al., 2015

Dot product: $a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^T \mathbf{k}$ Luong et al., 2015

Scaled dot product: $a(\mathbf{q}, \mathbf{k}) = \frac{\mathbf{q}^T \mathbf{k}}{\sqrt{|\mathbf{k}|}}$ Vaswani et al., 2017

More information:
"Deep Learning for NLP Best Practices", Ruder, 2017. http://ruder.io/deep-learning-nlp-best-practices/index.html#attention
"Massive Exploration of Neural Machine Translation Architectures", Britz et al, 2017, https://arxiv.org/pdf/1703.03906.pdf

Variants of Attention

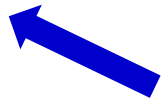
Original formulation: $a(\mathbf{q}, \mathbf{k}) = w_2^T \tanh(W_1[\mathbf{q}; \mathbf{k}])$

Bilinear product: $a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^T W \mathbf{k}$ Luong et al., 2015

Dot product: $a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^T \mathbf{k}$ Luong et al., 2015

Scaled dot product: $a(\mathbf{q}, \mathbf{k}) = \frac{\mathbf{q}^T \mathbf{k}}{\sqrt{|\mathbf{k}|}}$ Vaswani et al., 2017

Variant	Expressivity	Learnable Params	Efficiency	Reference
Additive	High	Yes	Moderate	Bahdanau et al., 2015
Bilinear	Medium	Yes	Low	Luong et al., 2015
Dot Product	Low	No	Very Low	Luong et al., 2015
Scaled Dot Product	Medium	No	Very Low	Vaswani et al., 2017

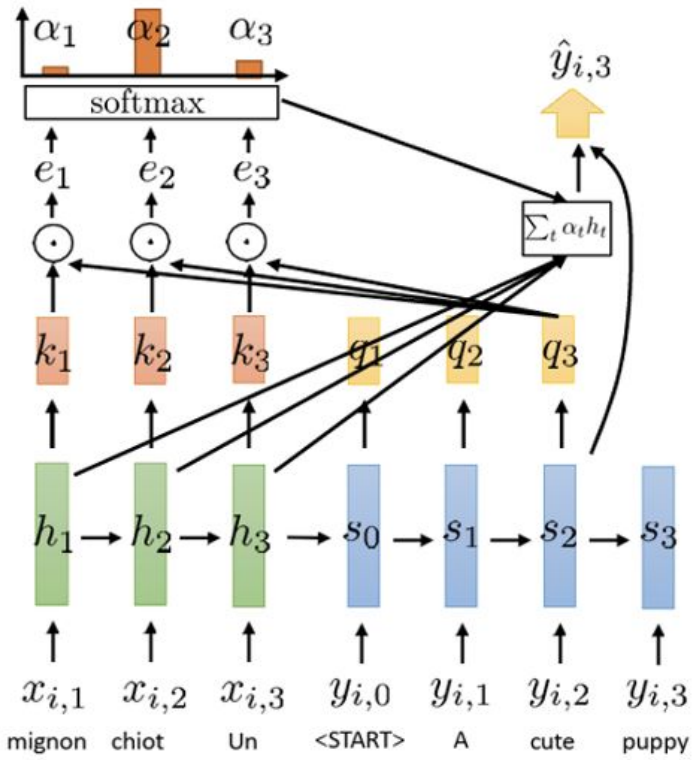


We just demonstrate Dot product attention

The Transformer paper used scaled dot product

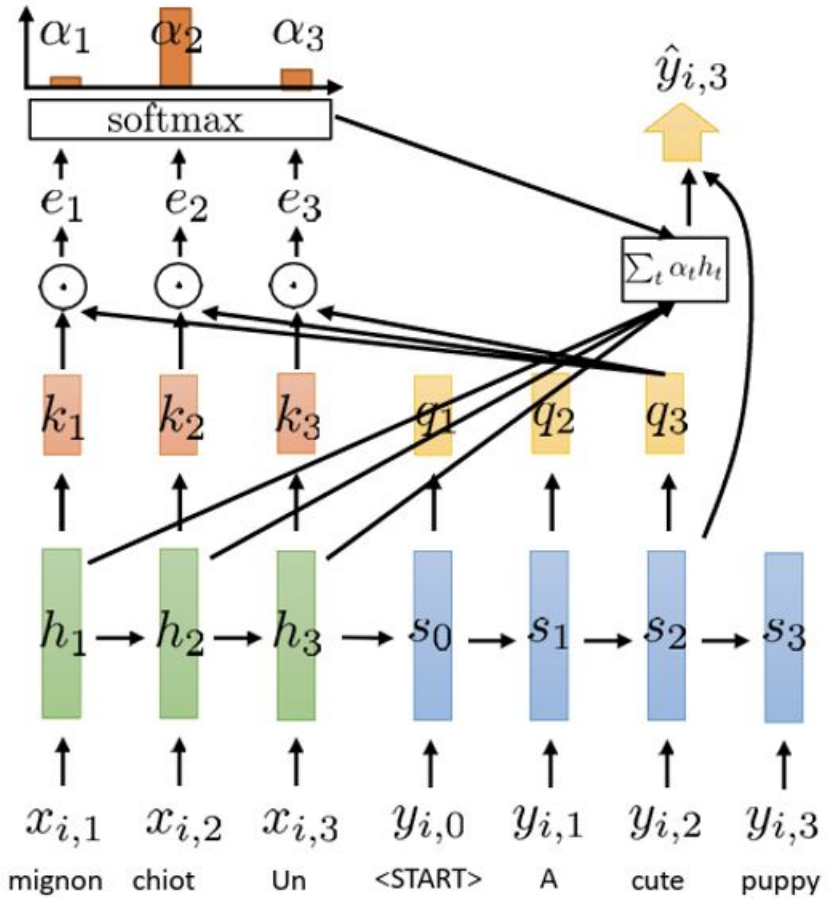
More information:
"Deep Learning for NLP Best Practices", Ruder, 2017. http://ruder.io/deep-learning-nlp-best-practices/index.html#attention
"Massive Exploration of Neural Machine Translation Architectures", Britz et al, 2017, https://arxiv.org/pdf/1703.03906.pdf

Do we need recurrence if we have attention?



- If we have **attention**, do we even need recurrent connections?
- Can we transform our RNN into a **purely attention-based model**?
- Attention can access all time steps simultaneously, potentially doing everything that recurrence can, and even more. However, this approach presents some challenges:

Do we need recurrent if we have attention?



- If we have **attention**, do we even need recurrent connections?
- Can we transform our RNN into a **purely attention-based model**?
- Attention can access all time steps simultaneously, potentially doing everything that recurrence can, and even more. However, this approach presents some challenges:

The encoder lacks temporal dependencies at all!

Self-Attention

Self-attention enables a model to compute contextualized representations of tokens by attending to other tokens in the sequence. It relies on three core components:

- **Query (q)**
- **Key (k)**
- **Value (v)**

Self-Attention

Let the input sequence be $X = [\mathbf{x}_1, \ldots, \mathbf{x}_n] \in \mathbb{R}^{n \times d}$, where n is the sequence length and d is the embedding dimension.

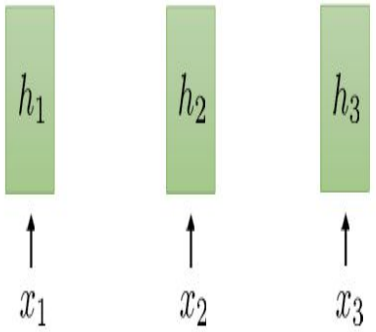
$$\mathbf{q}_i = \mathbf{W}^Q \mathbf{x}_i \qquad \text{(Query)}$$

$$\mathbf{k}_j = \mathbf{W}^K \mathbf{x}_j \qquad \text{(Key)}$$

$$\mathbf{v}_j = \mathbf{W}^V \mathbf{x}_j \qquad \text{(Value)}$$

- $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V \in \mathbb{R}^{d \times d}$ are learnable projection matrices.

Self-Attention



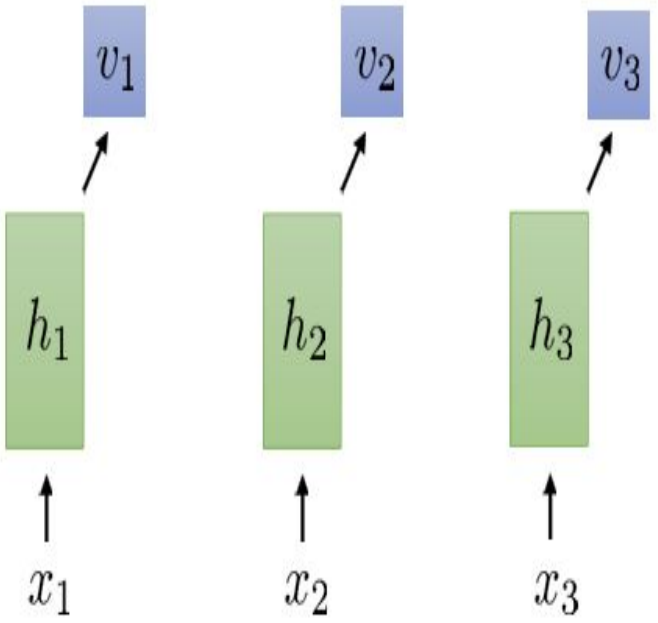
this is *not* a recurrent model!
but still weight sharing:

$$h_t = \sigma(W x_t + b)$$

shared weights at all time steps

(or any other nonlinear function)

Self-Attention



$v_t = v(h_t)$ before just had $v(h_t) = h_t$, now e.g. $v(h_t) = W_v h_t$

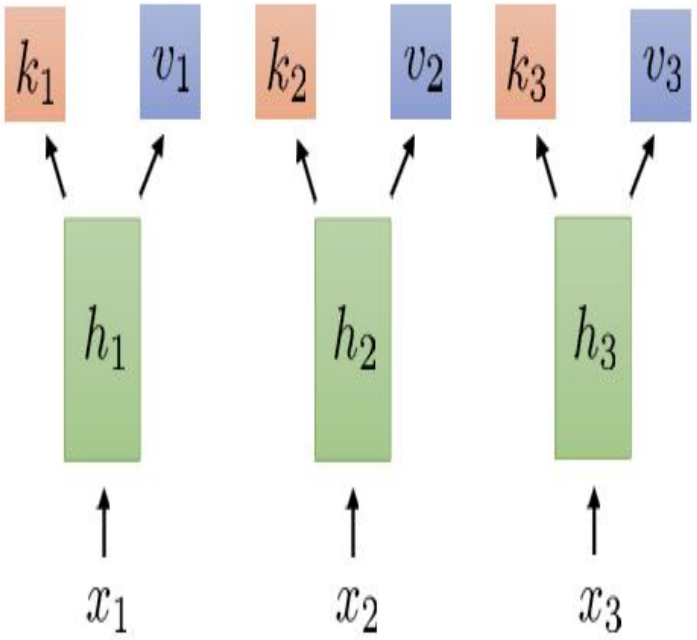
this is *not* a recurrent model!
but still weight sharing:

$$h_t = \sigma(W x_t + b)$$

shared weights at all time steps

(or any other nonlinear function)

Self-Attention



$v_t = v(h_t)$ before just had $v(h_t) = h_t$, now e.g. $v(h_t) = W_v h_t$
$k_t = k(h_t)$ (just like before) e.g., $k_t = W_k h_t$

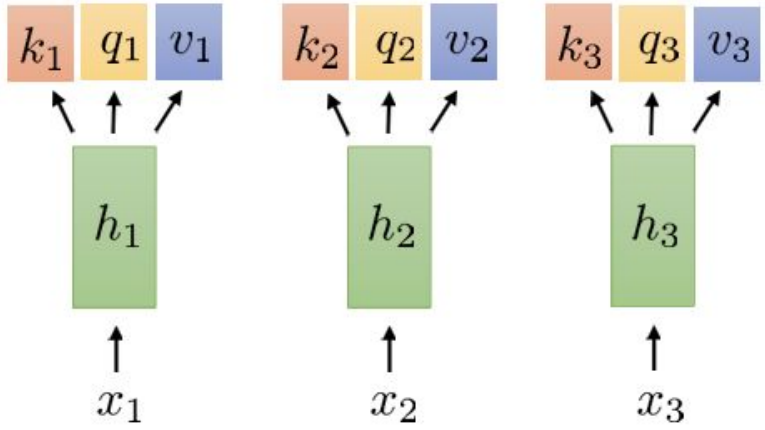
this is *not* a recurrent model!
but still weight sharing:

$$h_t = \sigma(W x_t + b)$$

shared weights at all time steps

(or any other nonlinear function)

Self-Attention



$v_t = v(h_t)$ before just had $v(h_t) = h_t$, now e.g. $v(h_t) = W_v h_t$
$k_t = k(h_t)$ (just like before) e.g., $k_t = W_k h_t$

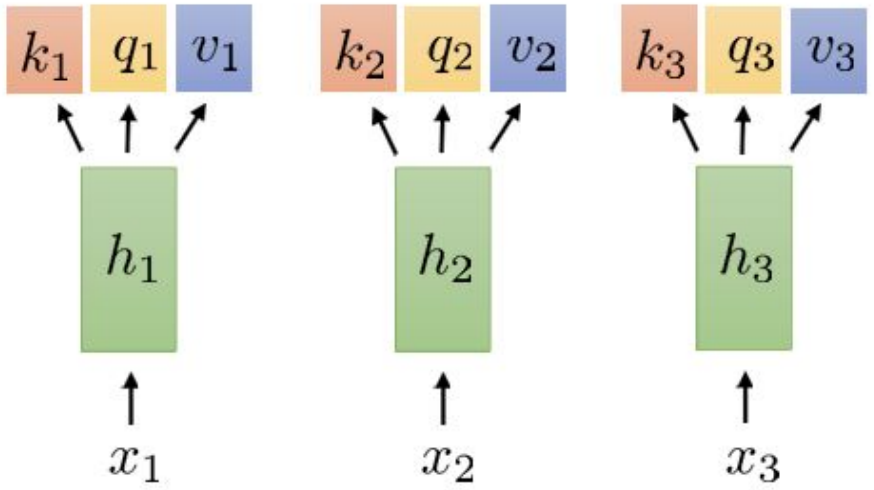
this is *not* a recurrent model!
but still weight sharing:

$$h_t = \sigma(W x_t + b)$$

shared weights at all time steps

(or any other nonlinear function)

Self-Attention



$v_t = v(h_t)$ before just had $v(h_t) = h_t$, now e.g. $v(h_t) = W_v h_t$

$k_t = k(h_t)$ (just like before) e.g., $k_t = W_k h_t$

$q_t = q(h_t)$ e.g., $q_t = W_q h_t$

this is *not* a recurrent model!

but still weight sharing:

$$h_t = \sigma(W x_t + b)$$

shared weights at all time steps

(or any other nonlinear function)

Self-Attention

Self-attention answers a simple question at every position in a sequence:

"Which other tokens are relevant to me, and how should their information influence my representation?"

To do this, each token is linearly projected into **three vectors**:

Query (Q) — *"What am I looking for?"*

A **query** represents the current token's information need. It encodes what feature or context this token wants to retrieve from the others.

Example: If the token is *"he"*, its query may encode the need to find who *"he"* refers to.

Self-Attention

Self-attention answers a simple question at every position in a sequence:

"Which other tokens are relevant to me, and how should their information influence my representation?"

To do this, each token is linearly projected into **three vectors**:

2. Key (K) — "What do I contain?"

key represents the properties of a token that make it relevant or irrelevant to others.

- Keys are compared with queries.
- The similarity **Q·K** determines how much attention one token should give to another.

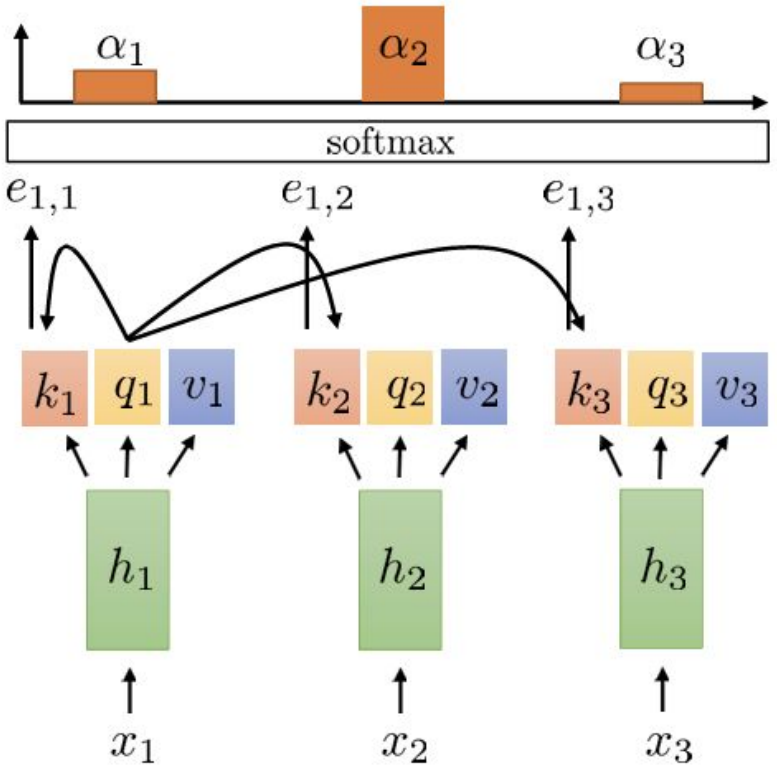
Self-Attention

3. Value (V) — *"What information do I pass along if selected?"*

A value stores the token's actual content to be aggregated (e.g., semantic features).

- Attention weights (from Q·K) determine how much each value contributes to the final representation of a token

Self-Attention



$e_{l,t} = q_l \cdot k_t$

$v_t = v(h_t)$ before just had $v(h_t) = h_t$, now e.g. $v(h_t) = W_v h_t$

$k_t = k(h_t)$ (just like before) e.g., $k_t = W_k h_t$

$q_t = q(h_t)$ e.g., $q_t = W_q h_t$

this is *not* a recurrent model!

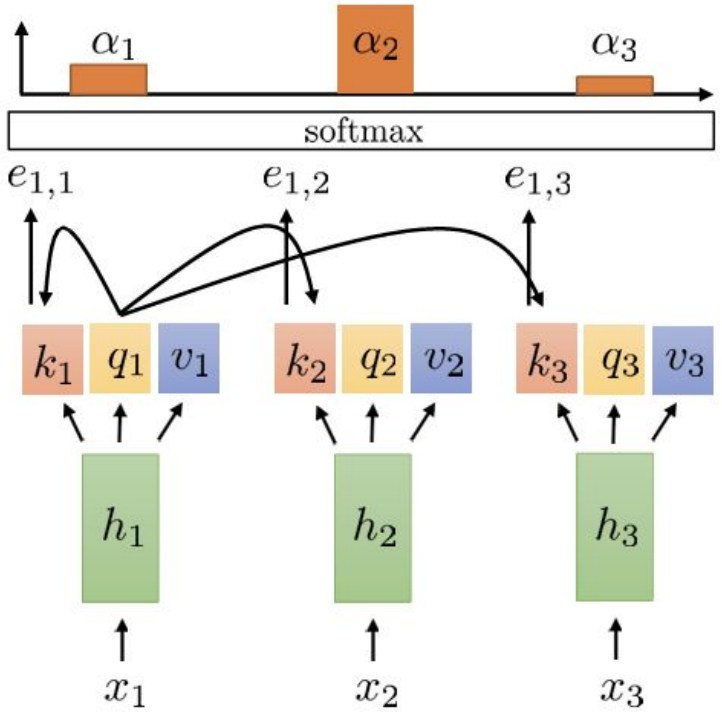
but still weight sharing:

$$h_t = \sigma(W x_t + b)$$

shared weights at all time steps

(or any other nonlinear function)

Self-Attention



$$\alpha_{l,t} = \exp(e_{l,t}) / \sum_{t'} \exp(e_{l,t'})$$

$$e_{l,t} = q_l \cdot k_t$$

$v_t = v(h_t)$ before just had $v(h_t) = h_t$, now e.g. $v(h_t) = W_v h_t$

$k_t = k(h_t)$ (just like before) e.g., $k_t = W_k h_t$

$q_t = q(h_t)$ e.g., $q_t = W_q h_t$

this is *not* a recurrent model!

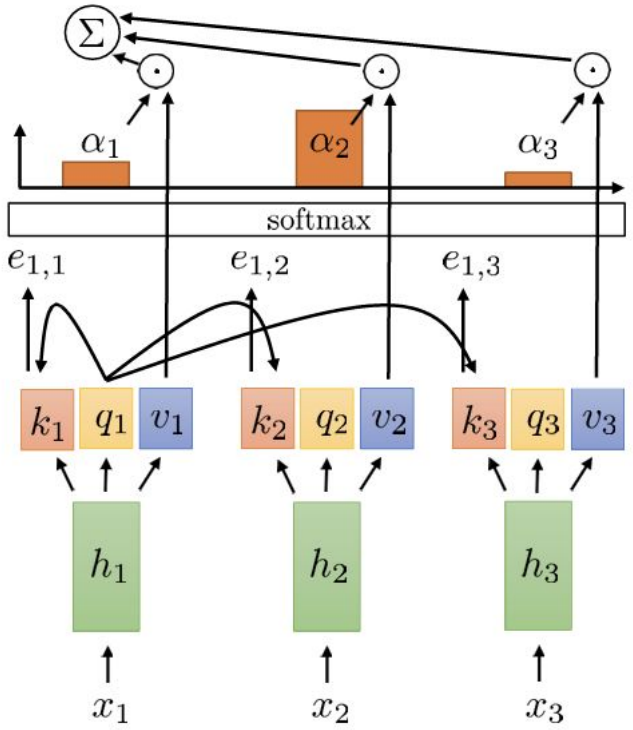
but still weight sharing:

$$h_t = \sigma(W x_t + b)$$

shared weights at all time steps

(or any other nonlinear function)

Self-Attention



$$\alpha_{l,t} = \exp(e_{l,t}) / \sum_{t'} \exp(e_{l,t'})$$

$$e_{l,t} = q_l \cdot k_t$$

$v_t = v(h_t)$ before just had $v(h_t) = h_t$, now e.g. $v(h_t) = W_v h_t$

$k_t = k(h_t)$ (just like before) e.g., $k_t = W_k h_t$

$q_t = q(h_t)$ e.g., $q_t = W_q h_t$

this is *not* a recurrent model!

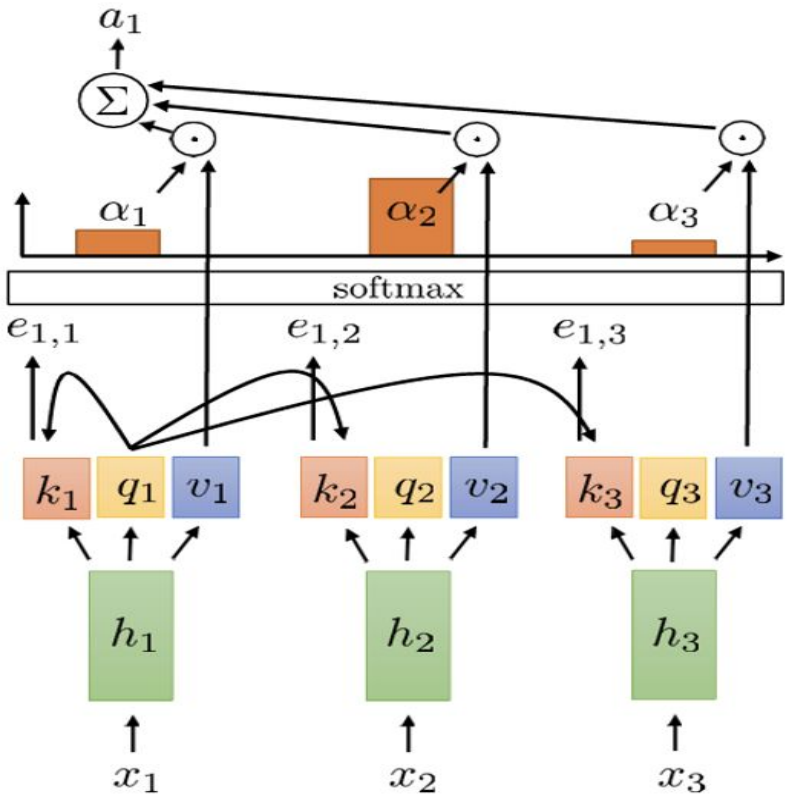
but still weight sharing:

$$h_t = \sigma(W x_t + b)$$

shared weights at all time steps

(or any other nonlinear function)

Self-Attention



$$a_l = \sum_t \alpha_{l,t} v_t$$

$$\alpha_{l,t} = \exp(e_{l,t}) / \sum_{t'} \exp(e_{l,t'})$$

$$e_{l,t} = q_l \cdot k_t$$

$v_t = v(h_t)$ before just had $v(h_t) = h_t$, now e.g. $v(h_t) = W_v h_t$

$k_t = k(h_t)$ (just like before) e.g., $k_t = W_k h_t$

$q_t = q(h_t)$ e.g., $q_t = W_q h_t$

this is *not* a recurrent model!

but still weight sharing:

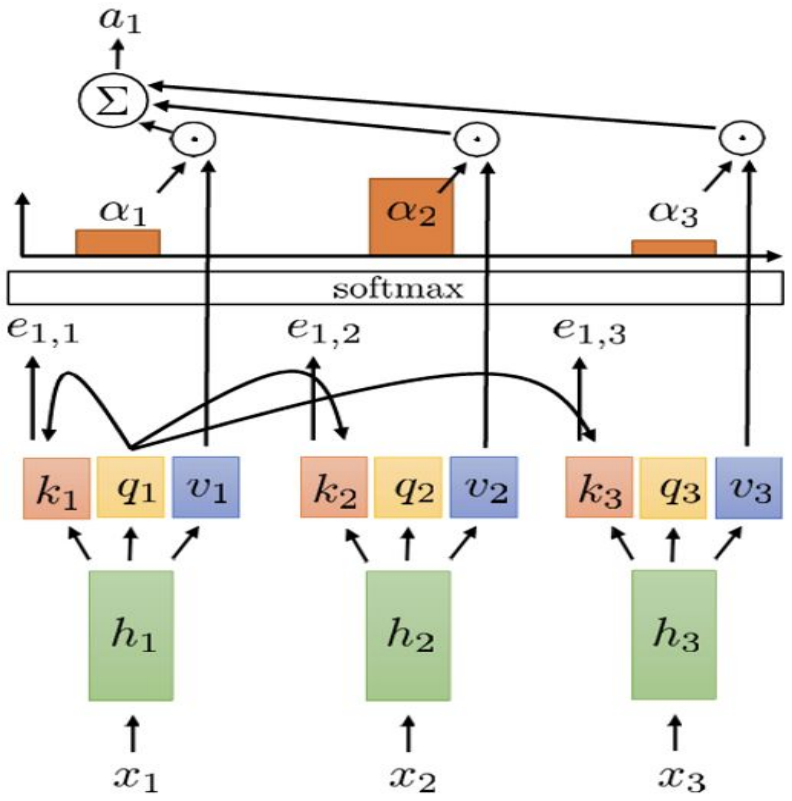
$$h_t = \sigma(W x_t + b)$$

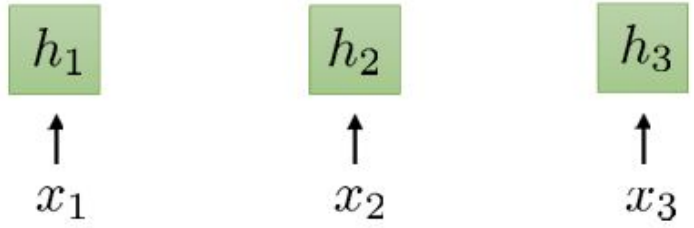
shared weights at all time steps

(or any other nonlinear function)

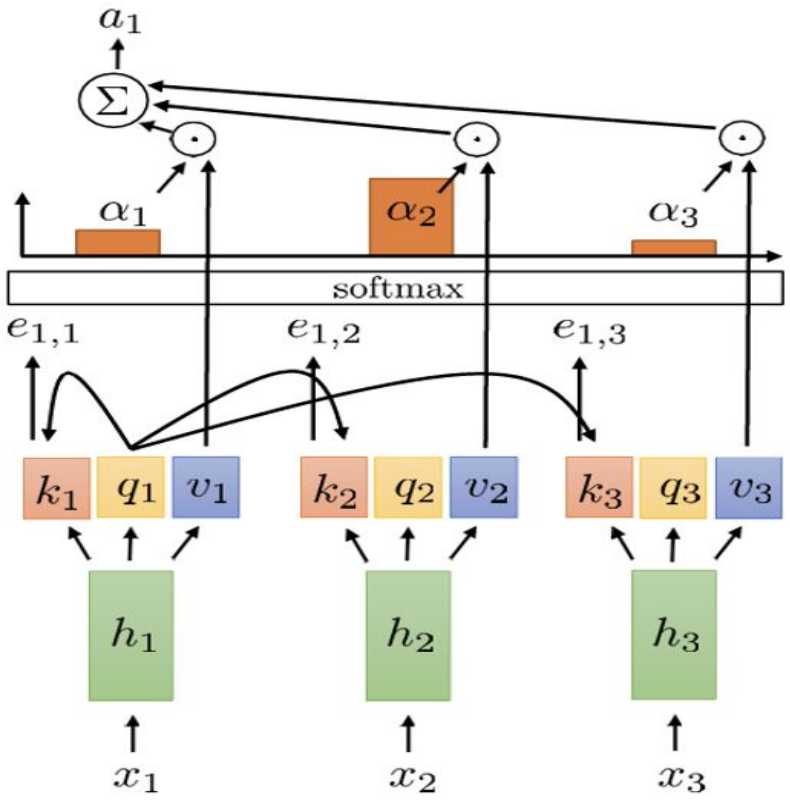
Self-Attention

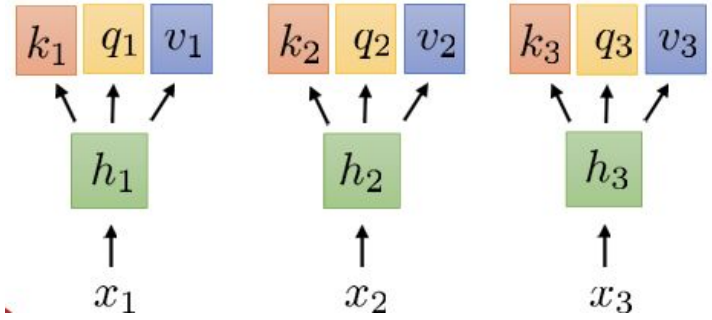
We can think of all these as a layer



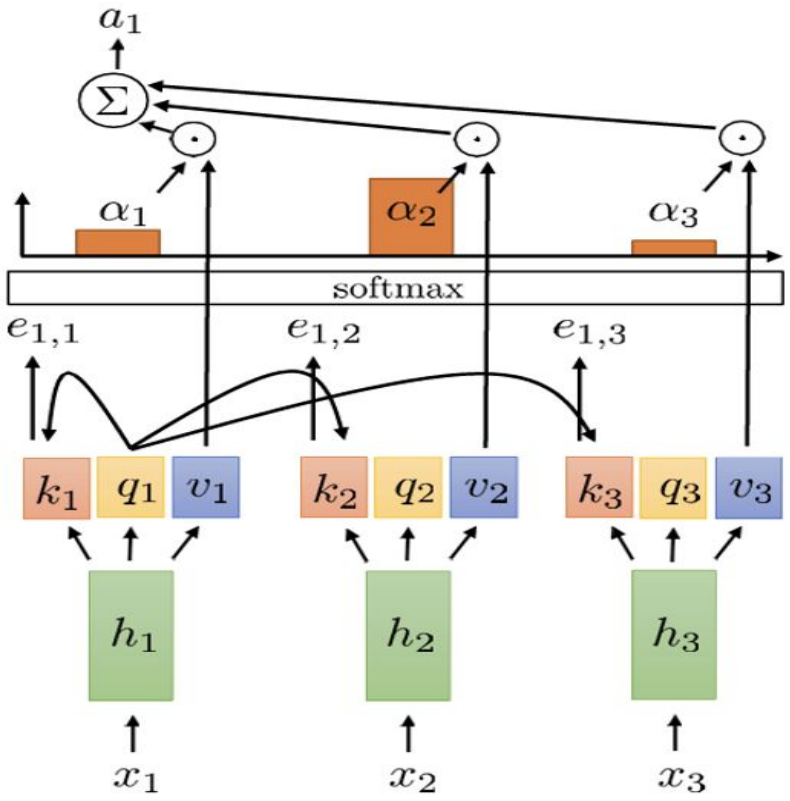


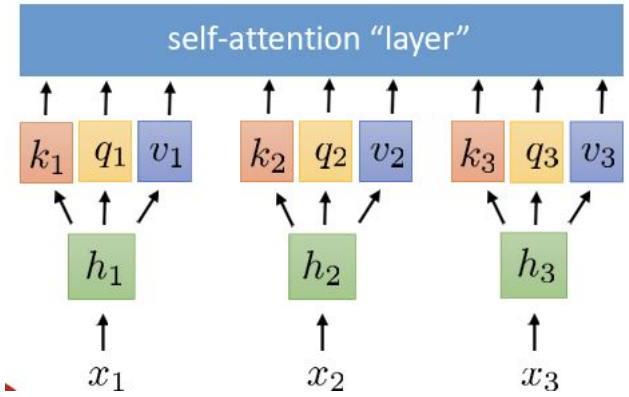
Self-Attention



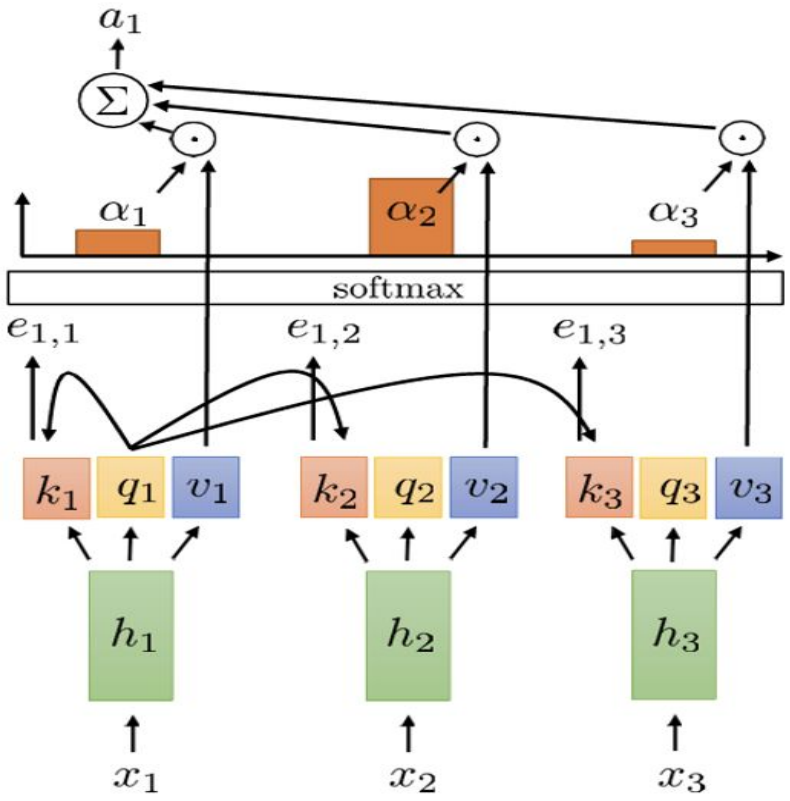


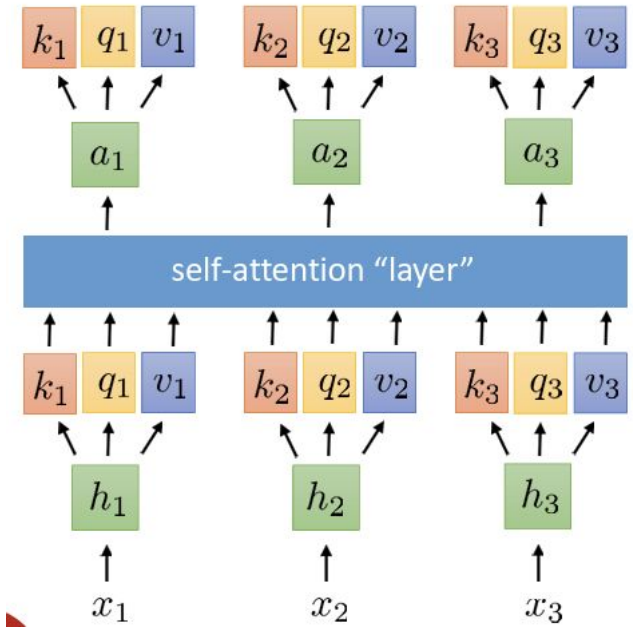
Self-Attention



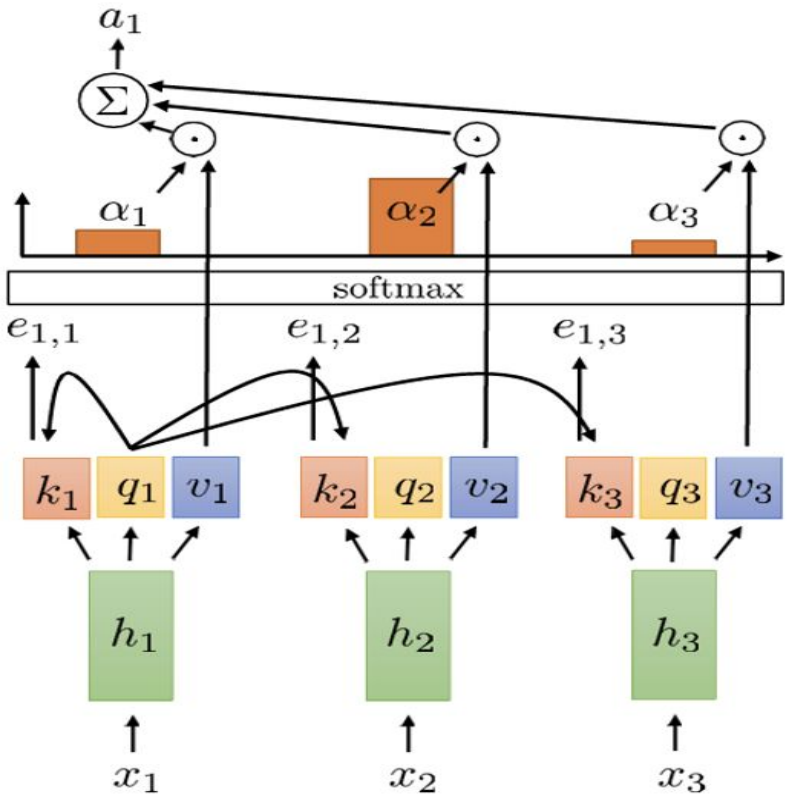


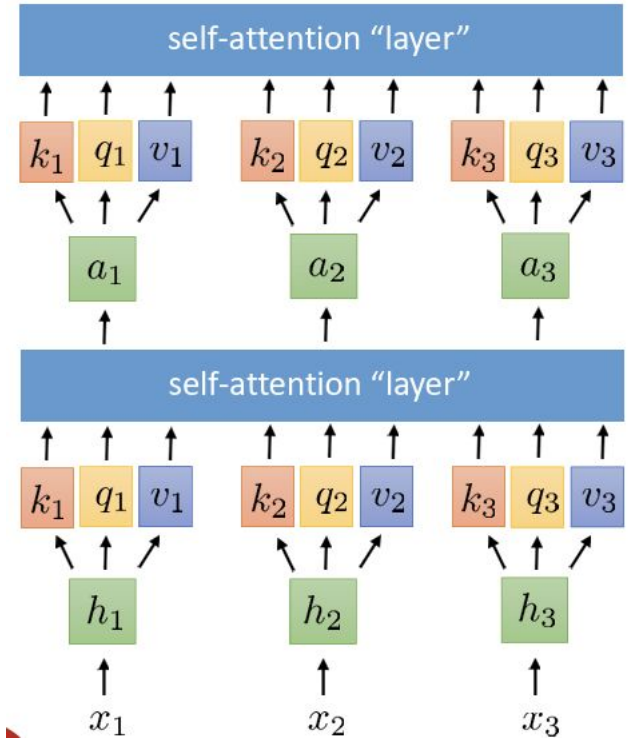
Self-Attention



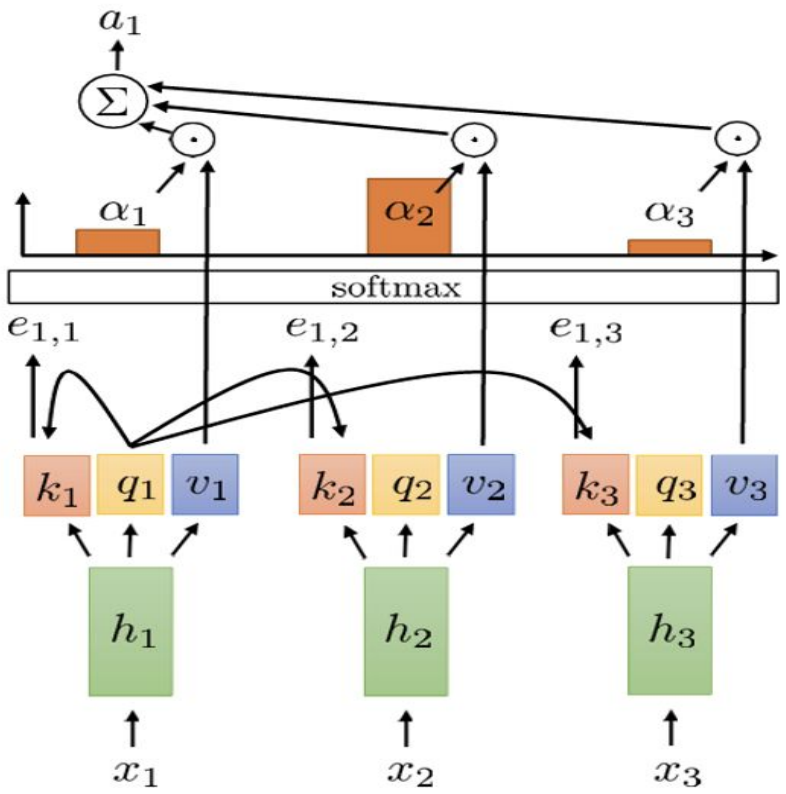


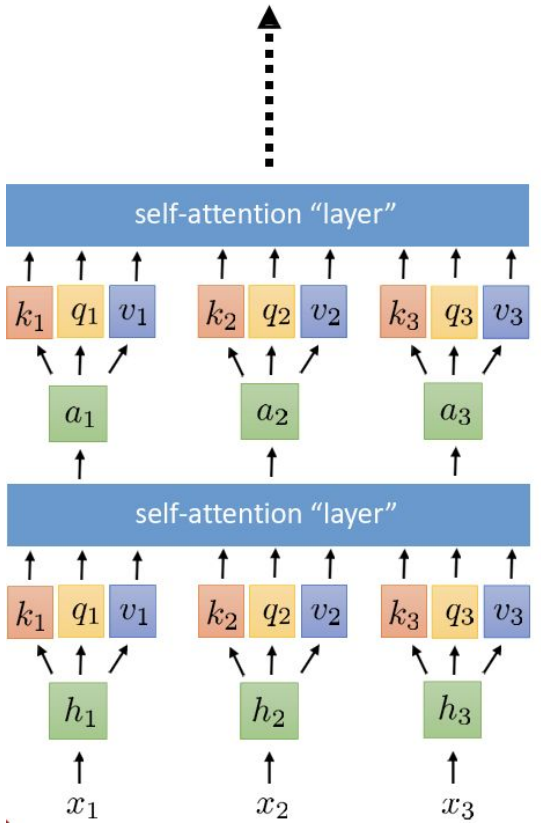
Self-Attention



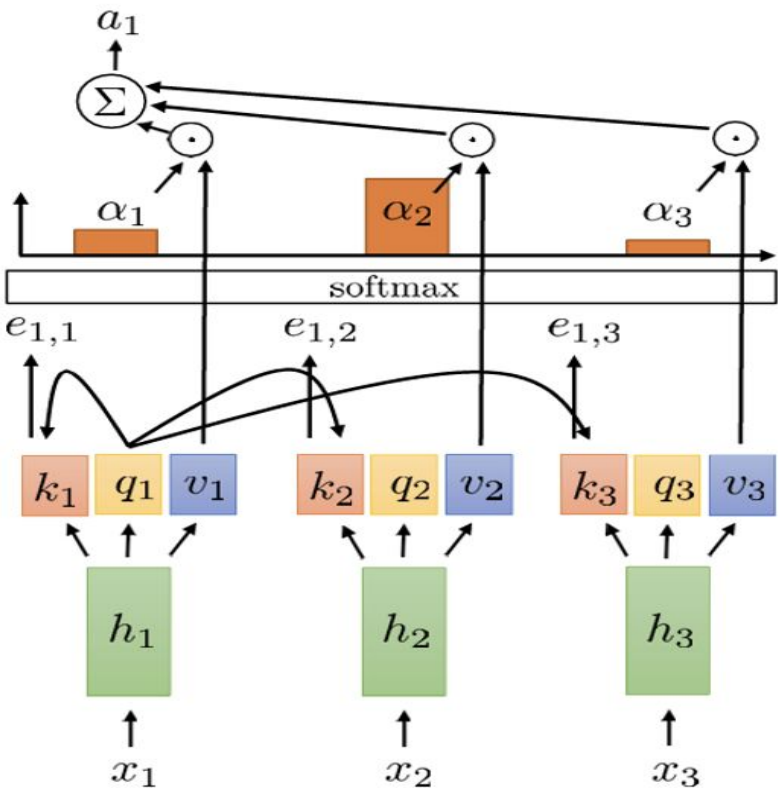


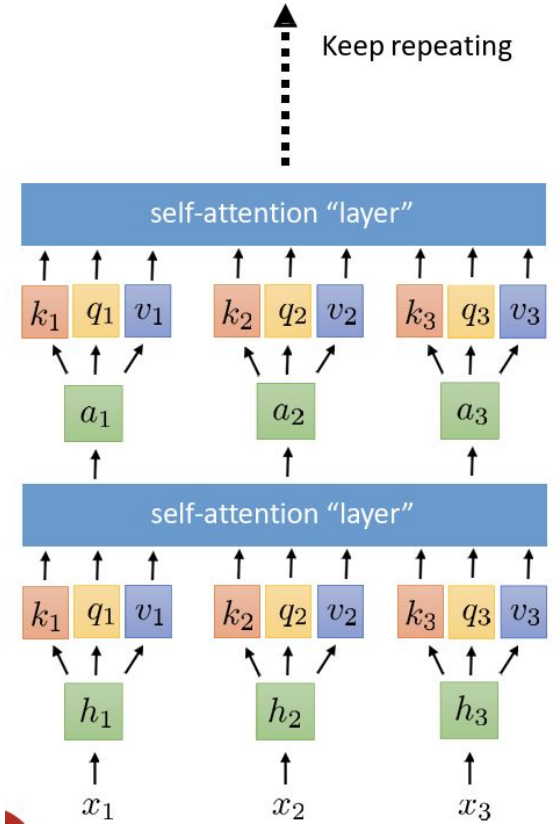
Self-Attention





Self-Attention





Self-Attention

Self-attention alone is insufficient because:

- It has no inherent notion of order → needs positional encoding
- One head cannot represent all relations → needs multi-head attention
- It remains linear without depth → needs nonlinear feed-forward layers
- It can see future tokens → needs masked decoding

These limitations motivate the full **Transformer architecture**.

From Self-Attention to Transformers

IMPERIAL

Q and A