

Attention Approaches



Introduction To Attention Mechanisms

- Traditional Machine Translation systems typically rely on sophisticated feature engineering based on the statistical properties of text.
- In short, these systems are complex, and a lot of engineering effort goes into building them. Neural Machine Translation systems work a bit differently.
- In Neural machine translation (NMT), we map the meaning of a sentence into a fixed-length vector representation and then generate a translation based on that vector.
- NMT systems are much easier to build and train, and they don't require any manual feature engineering.
- Most NMT systems work by encoding the source sentence (e.g. a German sentence) into a vector using a Recurrent Neural Network, and then decoding an English sentence based on that vector, also using a RNN.

Why Attention

- **Attention** is a recent and **important component of the success** of modern neural networks.
- We want neural nets that **automatically weigh the relevance** of the input and use these weights to perform a task.

Main advantages:

- Performance Gain - Improves accuracy by focusing on relevant parts.
- None Or Few Parameters - Adds little computational cost.
- Fast (Easy To Parallelize) - Unlike traditional RNNs, attention can process data more efficiently.
- Drop-in Implementation - Can be easily added to existing models.
- Tool For "Interpreting" Predictions

Example

Task: Hotel location

you get what you pay for . not the cleanest rooms but bed was clean and so was bathroom . bring your own towels though as very thin . service was excellent , let us book in at 8:30am ! for location and price , this ca n't be beaten , but it is cheap for a reason . if you come expecting the hilton , then book the hilton ! for uk travellers , think of a blackpool b&b.

Task: Hotel cleanliness

you get what you pay for . not the cleanest rooms but bed was clean and so was bathroom . bring your own towels though as very thin . service was excellent , let us book in at 8:30am ! for location and price , this ca n't be beaten , but it is cheap for a reason . if you come expecting the hilton , then book the hilton ! for uk travellers , think of a blackpool b&b.

Task: Hotel service

you get what you pay for . not the cleanest rooms but bed was clean and so was bathroom . bring your own towels though as very thin . service was excellent , let us book in at 8:30am ! for location and price , this ca n't be beaten , but it is cheap for a reason . if you come expecting the hilton , then book the hilton ! for uk travellers , think of a blackpool b&b.

Example

What is the woman looking at?



tv



computer

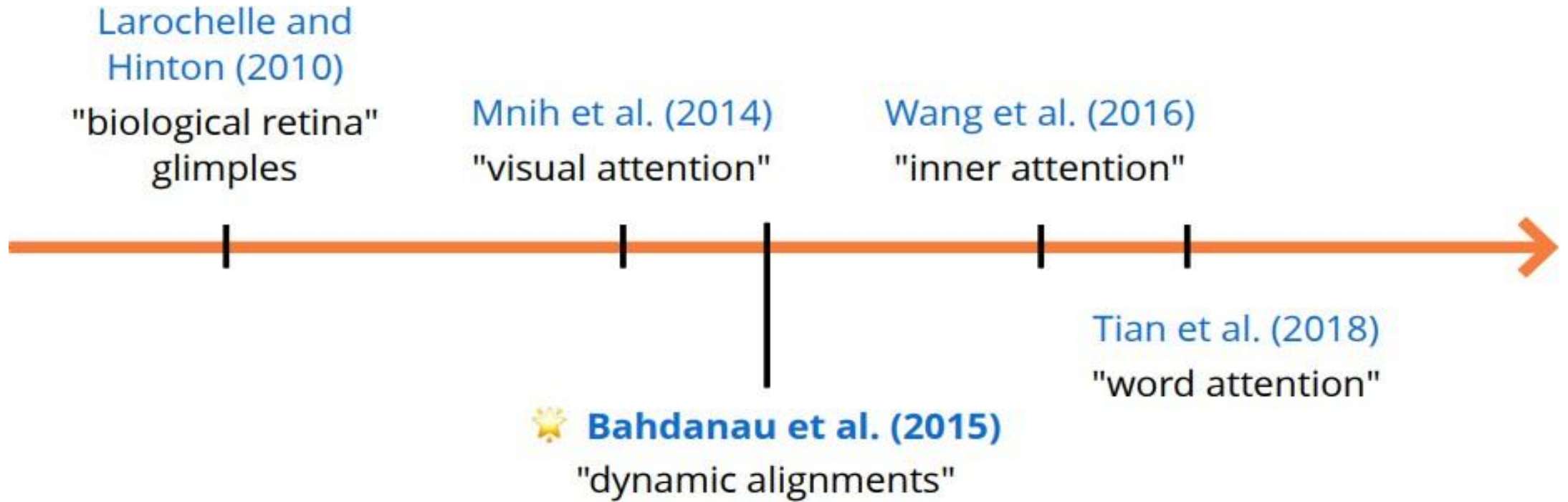


computer



Brief History

"first" introduced in NLP for Machine Translation by Bahdanau et al. (2015)



Brief History

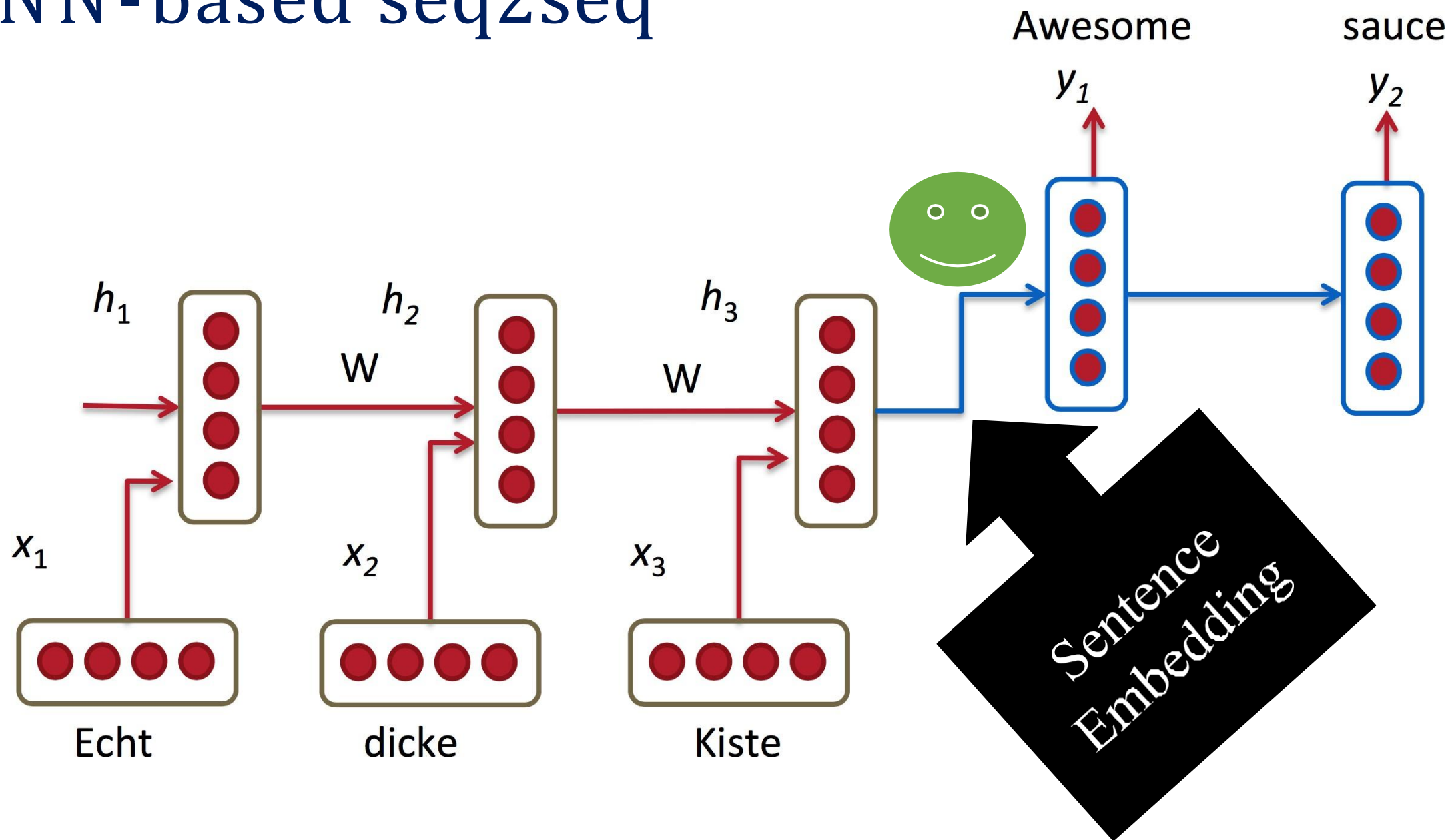
"first" introduced in NLP for Machine Translation by Bahdanau et al. (2015)



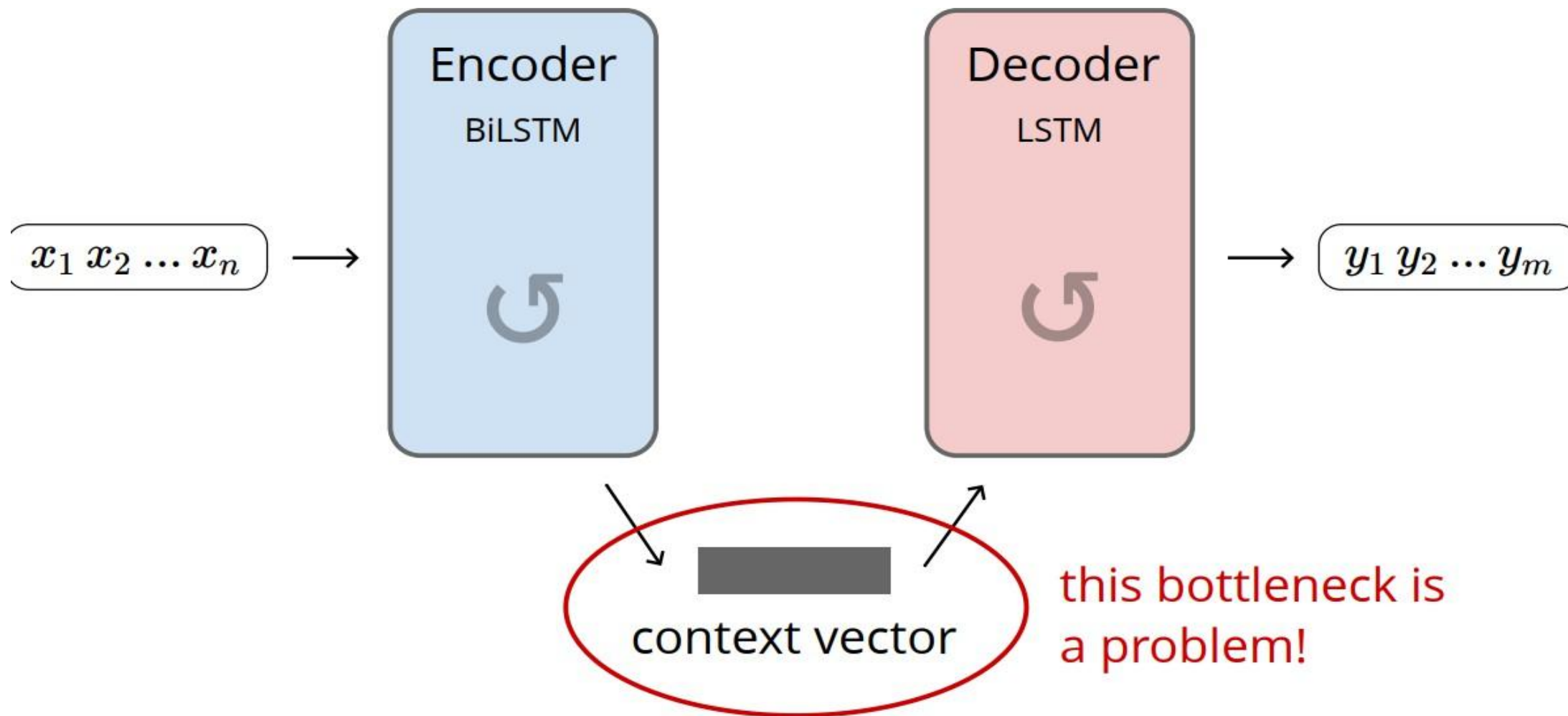
Attention in NLP

- Machine Translation
- Text Classification
- Text Summarization
- Language Modelling
- Question Answering
- Morphology
- Multimodal task
- Information Extraction
- Semantic
- Sentiment Analysis

RNN-based seq2seq



RNN-based seq2seq



Challenges

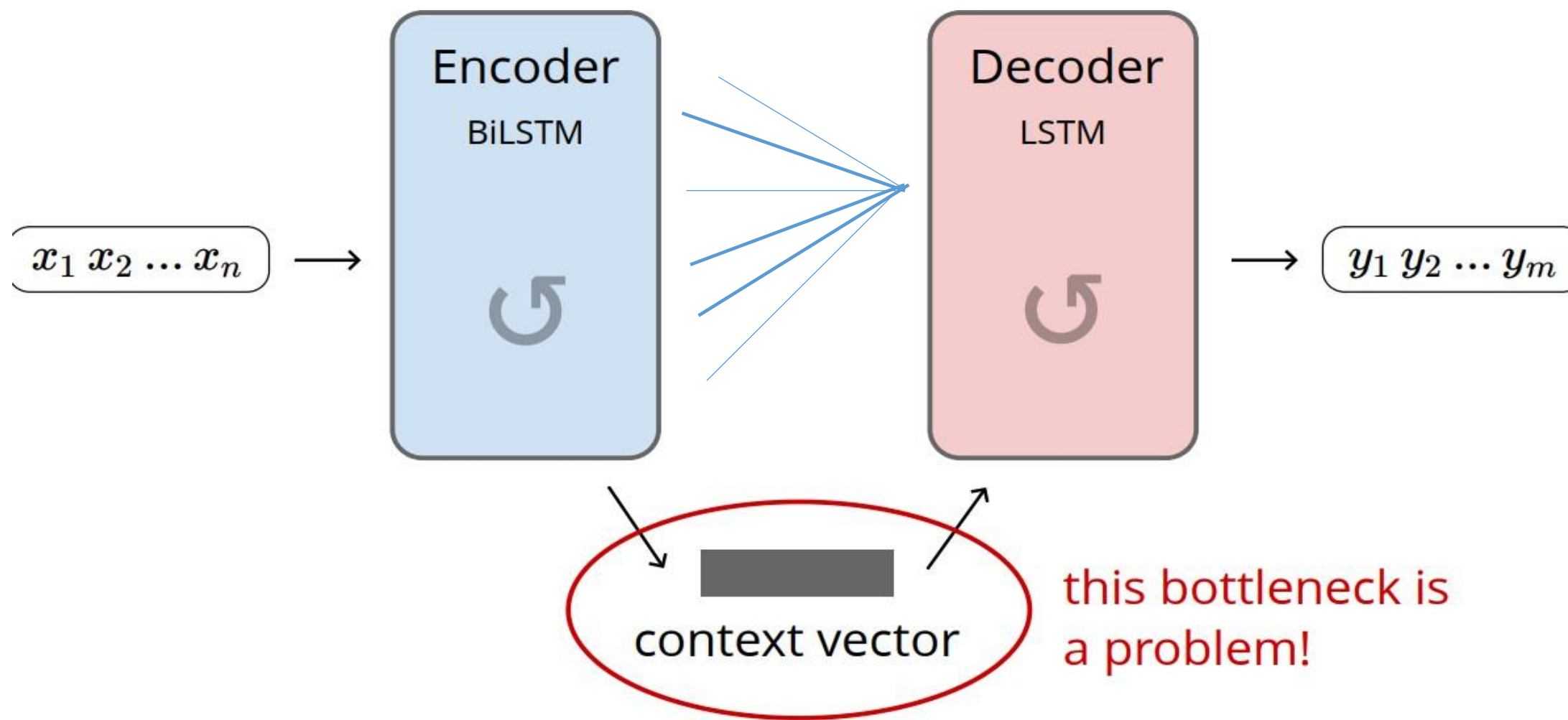
- Encoding all information about a **potentially very long sentence** into a single vector and then have the decoder produce a good translation based on only that.
- For example, assume the sentence is of **100 words long**, and the first word of the English **translation is probably highly correlated** with the first word of the source sentence.
- But that means the decoder has to consider information from **100 steps ago**, and that information needs to be somehow encoded in the vector.
- Recurrent Neural Networks are known to have problems **dealing with such long-range dependencies**.
- **LSTMs** should be able to deal with this, but in **practice long-range dependencies** are still problematic.

Solutions

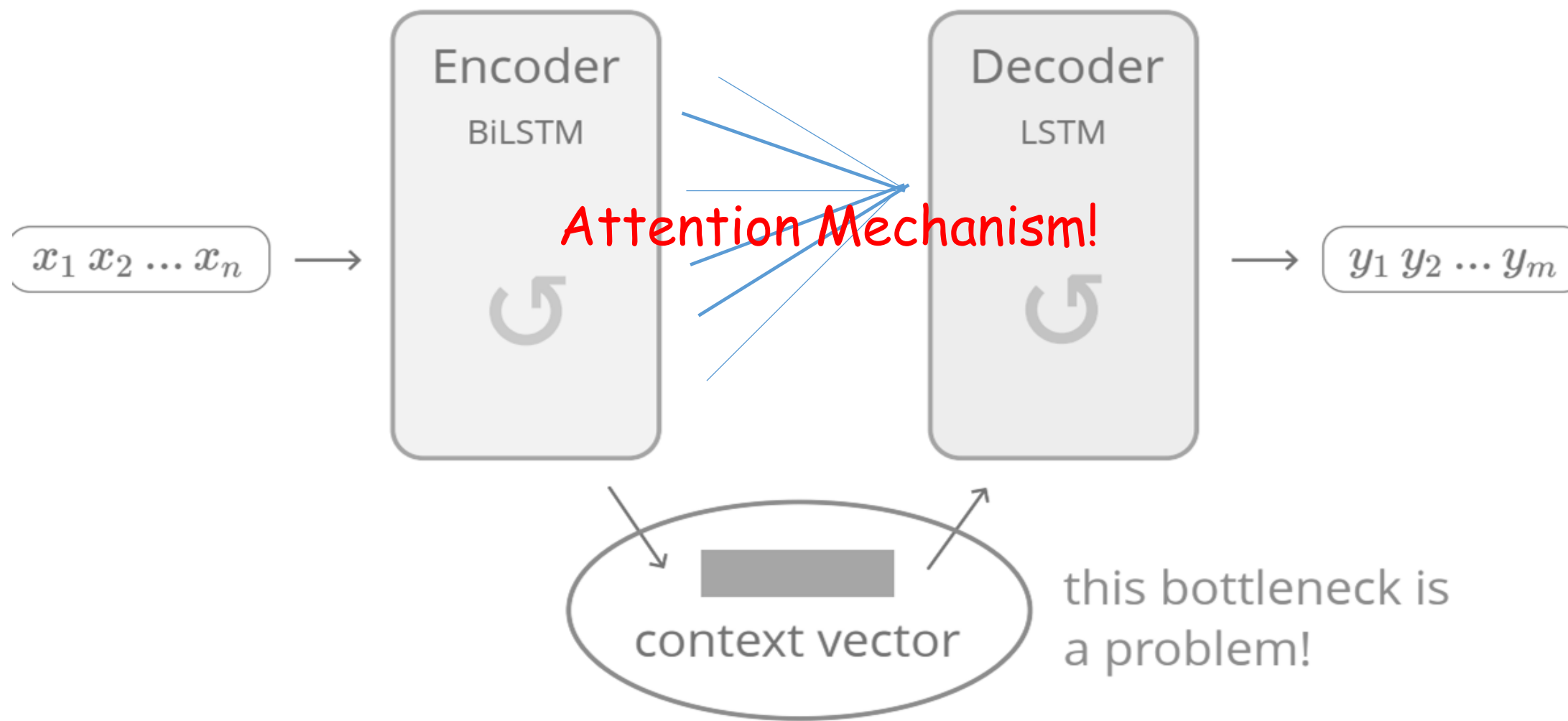
- Reversing the source sequence (feeding it backward into the encoder) produces significantly better results because it shortens the path from the decoder to the relevant parts of the encoder.
- Similarly, feeding an input sequence twice also seems to help a network to better memorize things.
- In a few cases(German/French, this technique might work)
- In a few other cases, there are languages in which the last word might be the root cause to predict the first word in the translation
- In that case, reversing the input would make things worse



RNN-based seq2seq



RNN-based seq2seq

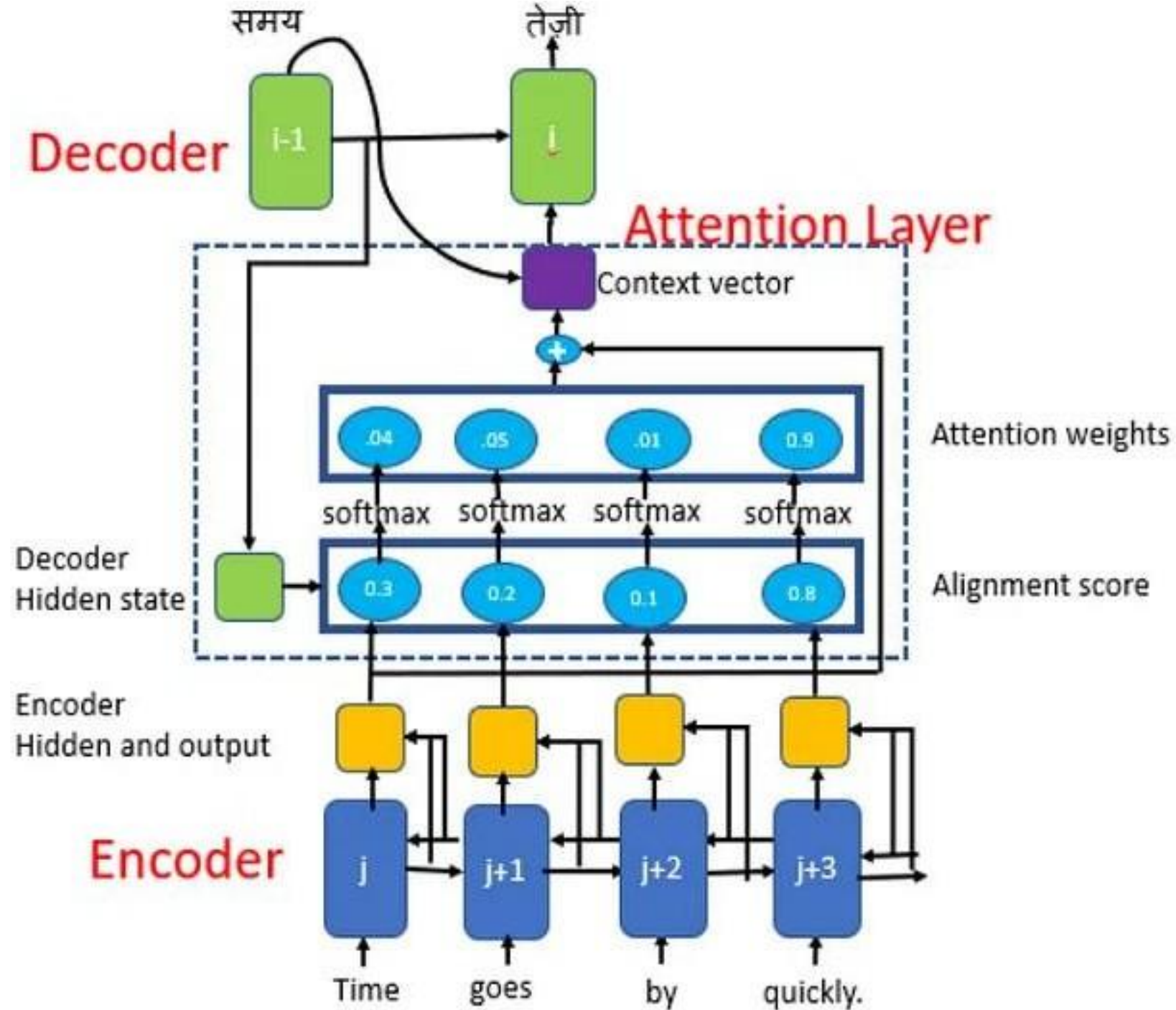


Attention Mechanism

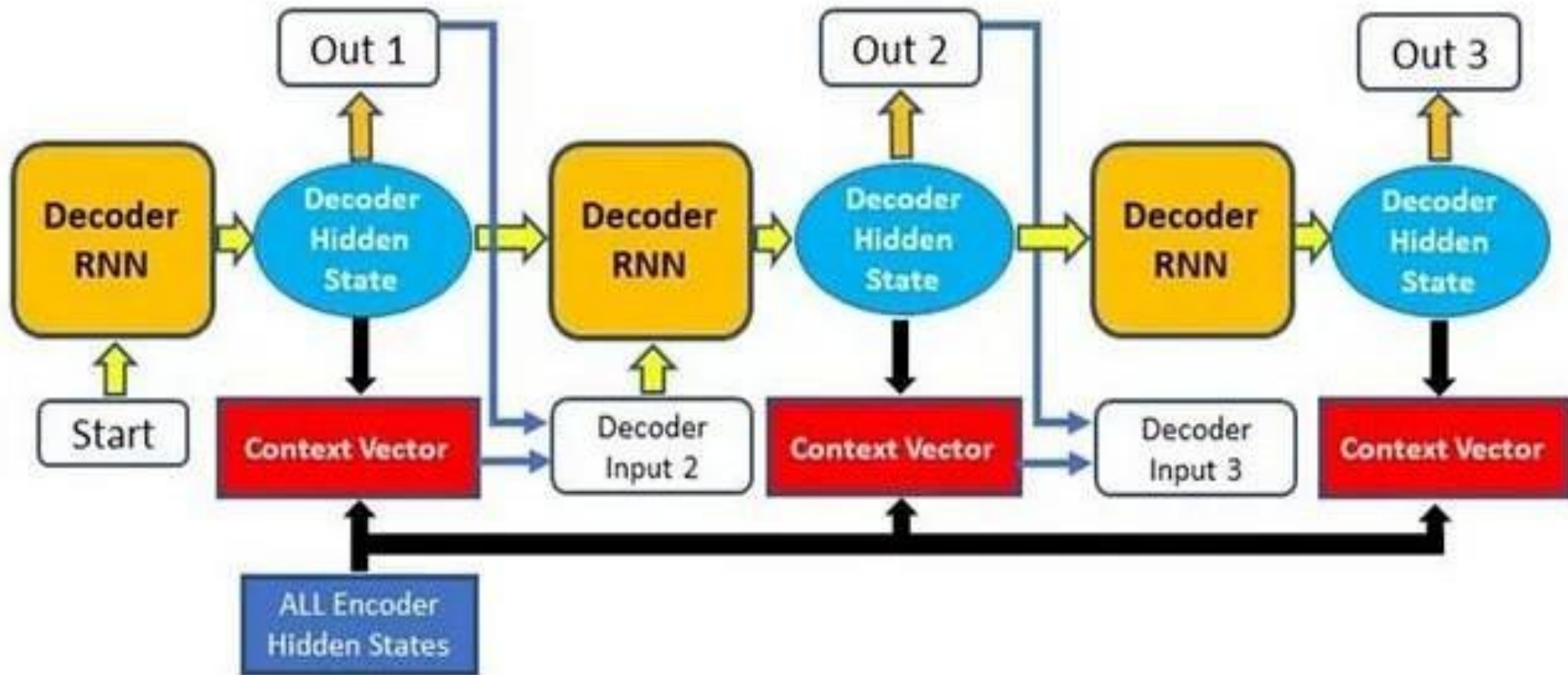
- The attention mechanism was introduced to **improve the performance of the encoder-decoder** model for **machine translation**.
- With an attention mechanism there is **no need to encode the full source sentence** into a **fixed-length vector**.
- Rather, we allow the decoder to **"attend"** to different **parts of the source sentence** at each step of the **output generation**.
- Importantly, we let the model **learn** what to attend to **based on the input sentence** and what it has produced so far.
- The attention mechanism helps the decoder **focus on the most important parts of the input sequence**. Instead of treating all input words equally, it **assigns different importance (weights) to each word**. The more relevant a word is to the current step, the higher its weight, allowing the decoder to make better predictions.

Attention Mechanism Cont'd

- The attention mechanism was introduced by Bahdanau et al. (2014) to address the bottleneck problem that arises with the use of a fixed-length encoding vector, where the decoder would have limited access to the information provided by the input
- Bahdanau et al.'s attention mechanism is divided into the step-by-step computations of the
 - *Alignment scores,*
 - *Weights, and*
 - *Context vector*
- also known as Additive attention as it performs a linear combination of encoder states and the decoder states

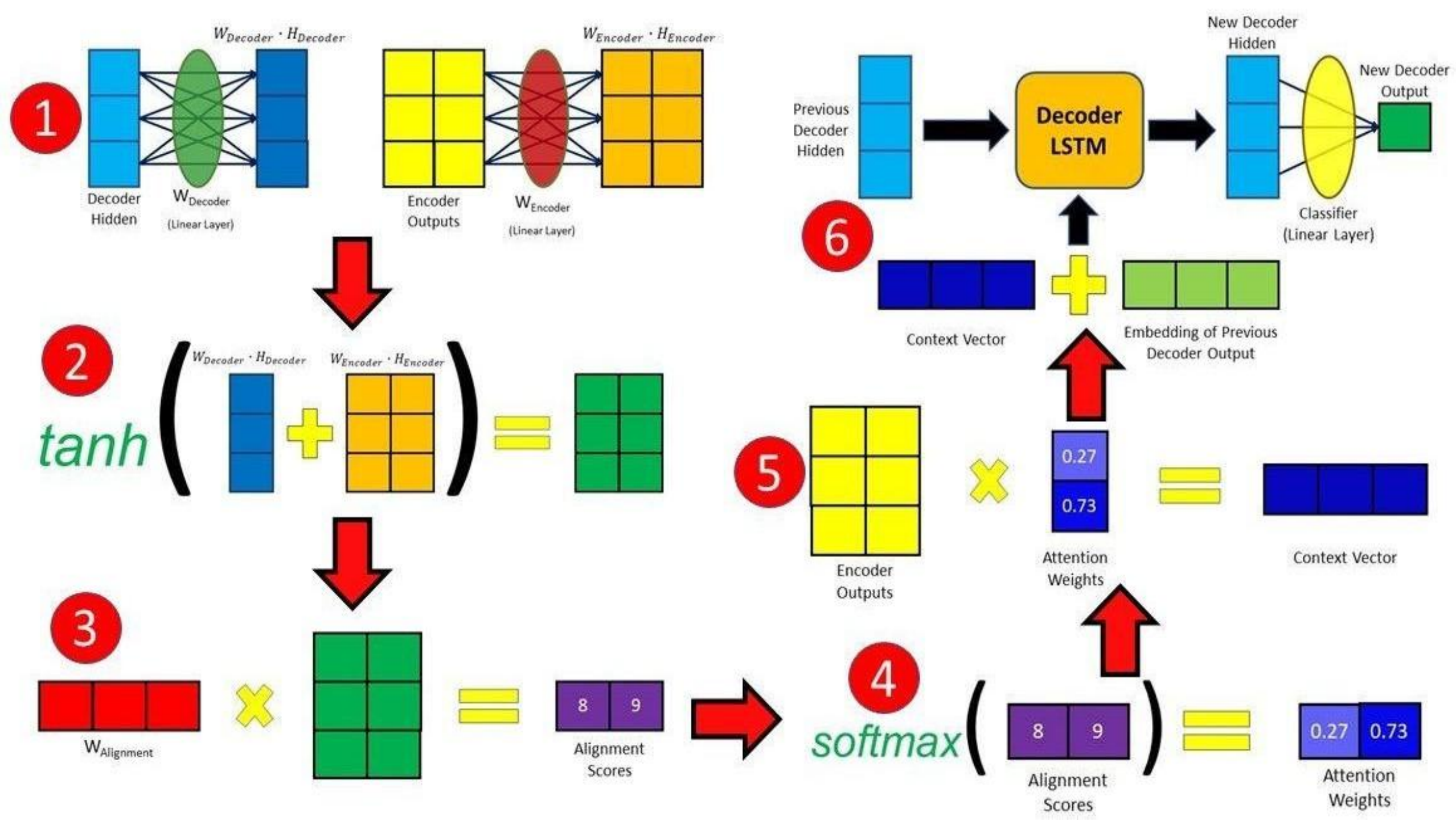


1. **Producing the Encoder Hidden States** - Encoder produces hidden states of **each** element in the input sequence
2. **Calculating Alignment Scores** between the **previous decoder hidden state** and each of **the encoder's hidden states** are calculated (Note: *The last encoder hidden state can be used as the first hidden state in the decoder*)
3. **Softmaxing the Alignment Scores** - The alignment scores for each encoder hidden state are combined and represented in a single **vector** and subsequently **softmaxed**
4. **Calculating the Context Vector** - the encoder hidden states and their respective alignment scores are *multiplied* to form the **context vector**
5. **Decoding the Output** - the context vector is *concatenated* with the previous decoder output and fed into the **Decoder RNN** for that time step along with the previous decoder hidden state to produce a **new output**
6. **The process (steps 2-5) repeats** itself for each time step of the decoder until an token is produced or output is past the specified maximum length



- All the **encoder hidden states**, along with the **decoder hidden state** are used to generate the **Context vector**

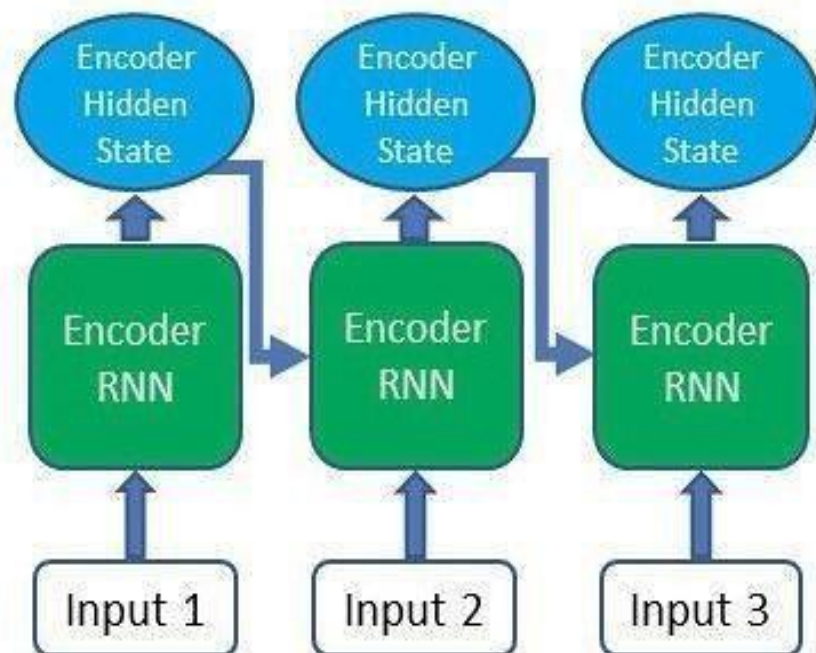
$$\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_s) = \mathbf{v}_a^\top \tanh(\mathbf{W}_1 \mathbf{h}_t + \mathbf{W}_2 \bar{\mathbf{h}}_s) \quad [\text{Bahdanau's additive style}]$$



- 1. Encode the Input Sentence (Blue & Yellow):** The encoder processes the input sequence and generates hidden states (Yellow). The decoder hidden state (Blue) will be used for attention computation.
- 2. Compute Alignment Scores (Green, Blue & Yellow):** The decoder hidden state (Blue) and each encoder hidden state (Yellow) are combined using a learned transformation. The tanh activation (Green) is applied to capture complex relationships.
- 3. Apply Weight Matrix (Red & Green):** A weight matrix $W_{\text{alignment}}$ (Red) is used to transform the scores (Green), producing alignment scores.
- 4. Compute Attention Weights (Purple & Blue):** The alignment scores (Purple) are passed through softmax to generate attention weights (Blue), representing how much focus each encoder hidden state should receive.
- 5. Generate Context Vector (Blue & Yellow):** The encoder hidden states (Yellow) are weighted by the attention scores (Blue), producing the context vector (Blue).
- 6. Decode the Output (Blue, Green & Yellow):** The context vector (Blue) and previous decoder output (Green) are passed into the decoder LSTM (Yellow) to generate the next word in the output sequence.

Step 1: Encode the Input Sentence

- The input sentence is passed through an **encoder**, which generates a **sequence of hidden states** H_{encoder} .
- Instead of using only the hidden state at the final time step, we'll be carrying forward all the hidden states produced by the encoder to the next step.
- These hidden states **represent the meaning** of each word in the input.



Step 2: Compute Alignment Scores

- The **decoder's current hidden state** H_{decoder} is compared with **each encoder hidden state** H_{encoder} .
- The comparison is done using an **alignment score function**, which determines **how much focus** the decoder should give to each input word.

Concat Attention (Bahdanau's Additive Method):

$$\text{score}_{\text{alignment}} = W \cdot \tanh(W_{\text{combined}}(H_{\text{encoder}} + H_{\text{decoder}}))$$

- The encoder and decoder hidden states are **concatenated**.
- They are **transformed by a weight matrix** W_{combined} .
- A **tanh activation** is applied, followed by another transformation using W .
- This method allows the model to **learn complex dependencies** between the encoder and decoder states.

Step 3: Compute Attention Weights

- The alignment scores are **normalized** using **Softmax** to form **attention weights**.
- These weights determine the **importance of each word in the input** for generating the next word in the output.

Step 4: Create the Context Vector

- The attention weights are used to compute a **weighted sum** of encoder hidden states.
- This weighted sum is called the **context vector**, which contains **important information** for the next decoding step.

Step 5: Decode the Output

- The **context vector** and the **previous decoder output** are **fed into the decoder**.
- The decoder **combines this information** to generate the next word in the output sentence.

Luong's Attention

- Luong's attention mechanism, an extension Bahdanau attention mechanism, is applied at the top layer of the encoder, unlike Bahdanau, which applies attention across all hidden states.

Key Differences Between Bahdanau and Luong Attention

- **Bahdanau Attention (Additive Attention)** computes alignment scores using a feedforward neural network.
- **Luong Attention (Multiplicative Attention)** uses dot product and other simple scoring functions, making it computationally efficient.

Steps in Luong Attention Mechanism

1. Encoding the Input Sequence

- The encoder processes the input sequence and generates a set of hidden representations, denoted as $H = \{h_i\}_{i=1}^T$, where T is the length of the input sequence.

2. Computing the Current Decoder Hidden State

- The decoder generates its hidden state at time t using the previous hidden state and the last output: $s_t = \text{RNN}_{\text{decoder}}(s_{t-1}, y_{t-1})$

3. Computing Alignment Scores

- The alignment score measures how well each encoder hidden representation h_i aligns with the current decoder state s_t : $e_{t,i} = a(s_t, h_i)$

Dot Product Attention (Luong's Dot Method):

$$\text{score}_{\text{alignment}} = H_{\text{encoder}} \cdot H_{\text{decoder}}$$

The encoder and decoder hidden states are directly multiplied.

General Attention (Luong's General Method):

$$\text{score}_{\text{alignment}} = W(H_{\text{encoder}} \cdot H_{\text{decoder}})$$

A weight matrix W is added to transform the encoder hidden state before computing the dot product.

Concat Attention (Bahdanau's Additive Method):

$$\text{score}_{\text{alignment}} = W \cdot \tanh(W_{\text{combined}}(H_{\text{encoder}} + H_{\text{decoder}}))$$

4. Applying Softmax to Compute Attention Weights

- The alignment scores are converted into attention weights using the softmax

$$\text{function: } \alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{j=1}^T \exp(e_{t,j})}$$

5. Computing the Context Vector

- The context vector is a weighted sum of the encoder hidden representations:

$$c_t = \sum_{i=1}^T \alpha_{t,i} h_i$$

6. Generating the Attentional Hidden State

- The context vector and the decoder hidden state are combined and passed

$$\text{through a transformation: } \tilde{s}_t = \tanh(W_c[c_t; s_t])$$

7. Generating the Final Output

- The final output is obtained by applying a softmax function over the transformed

$$\text{hidden state: } y_t = \text{softmax}(W_y \tilde{s}_t)$$

Bahdanau Attention

2. Computing Alignment Scores

Alignment scores are computed using the previous decoder hidden state and all encoder hidden states.

5. Using Context Vector in Decoding

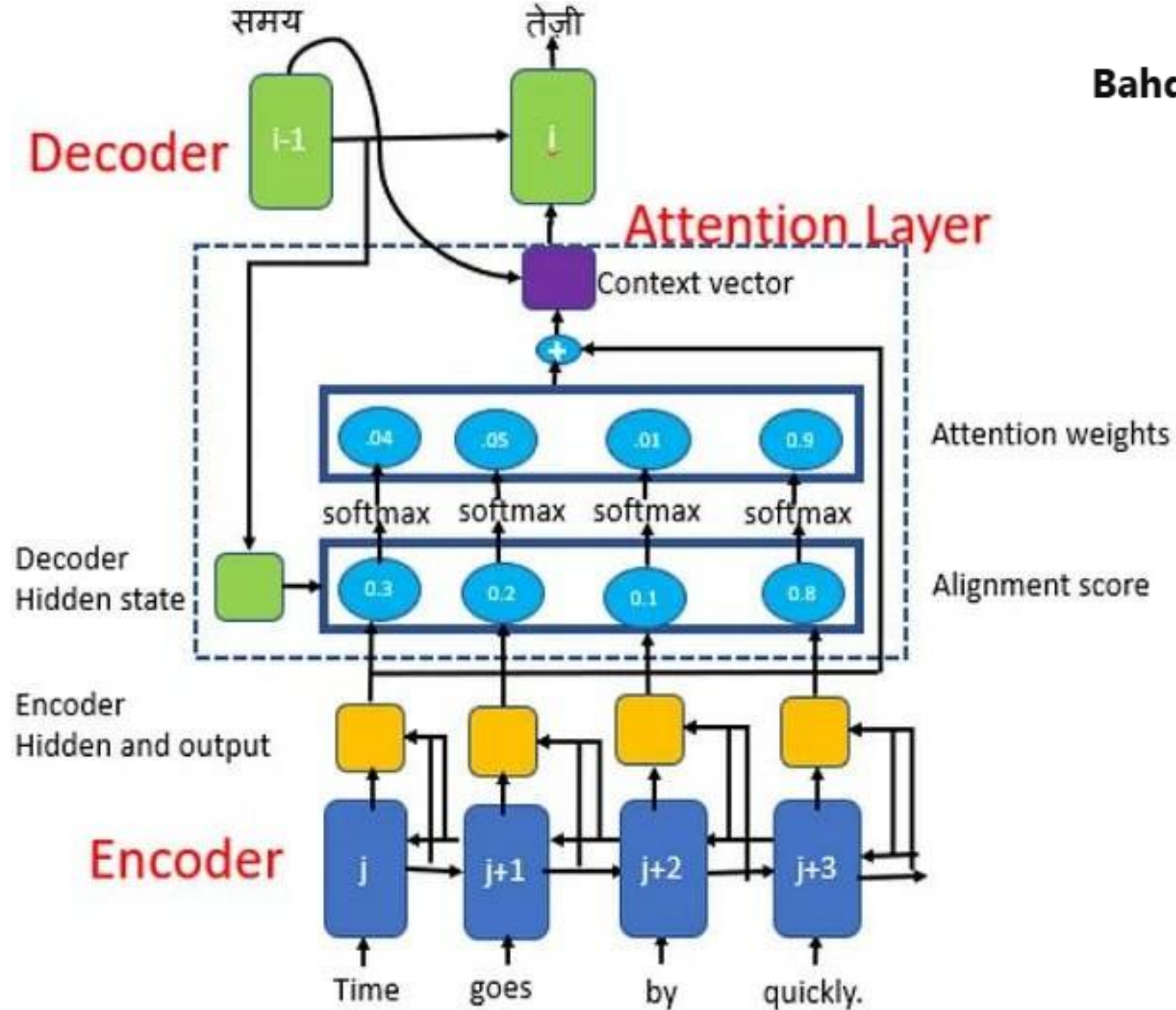
The context vector is concatenated with the previous decoder output and fed into the decoder RNN.

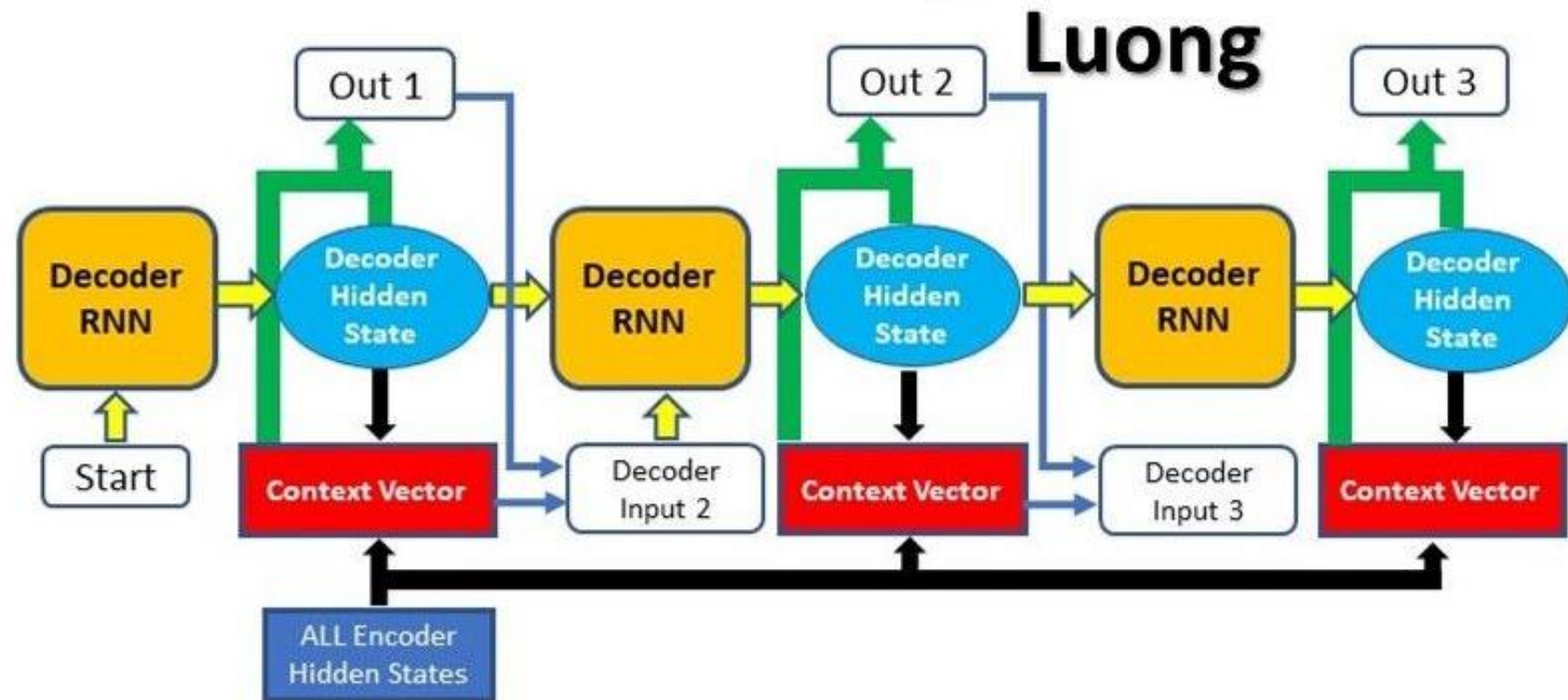
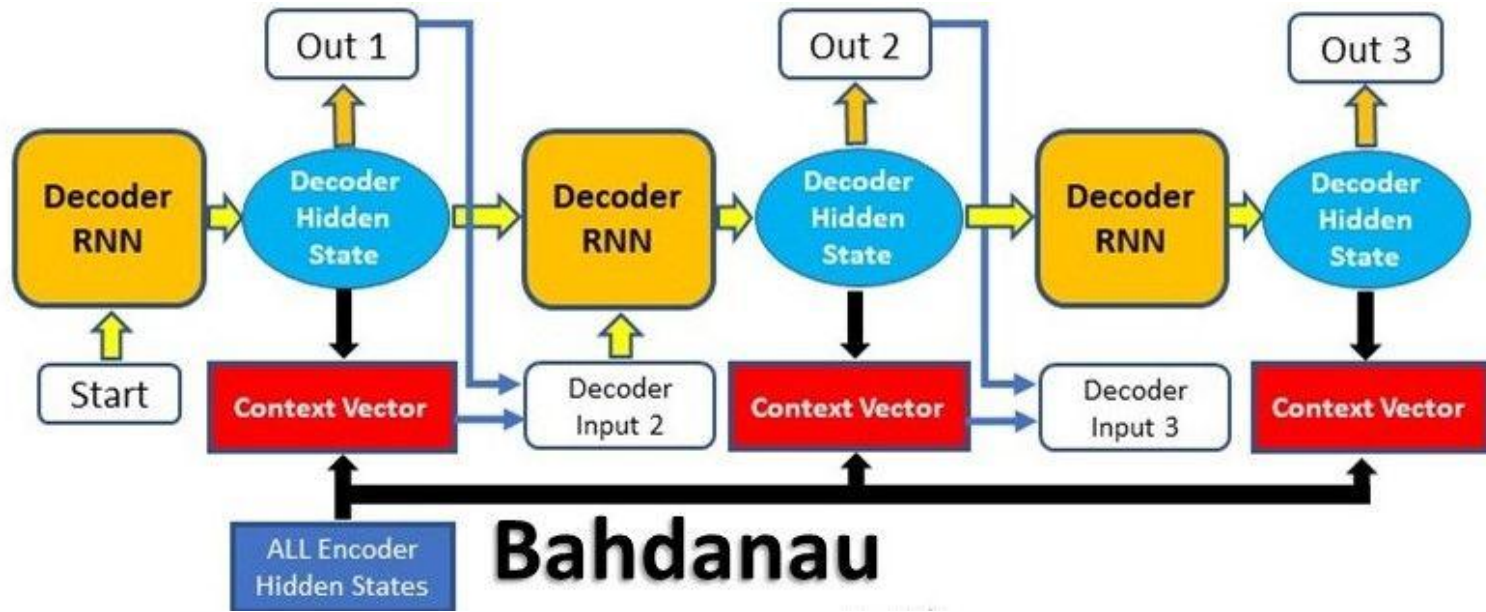
Luong Attention

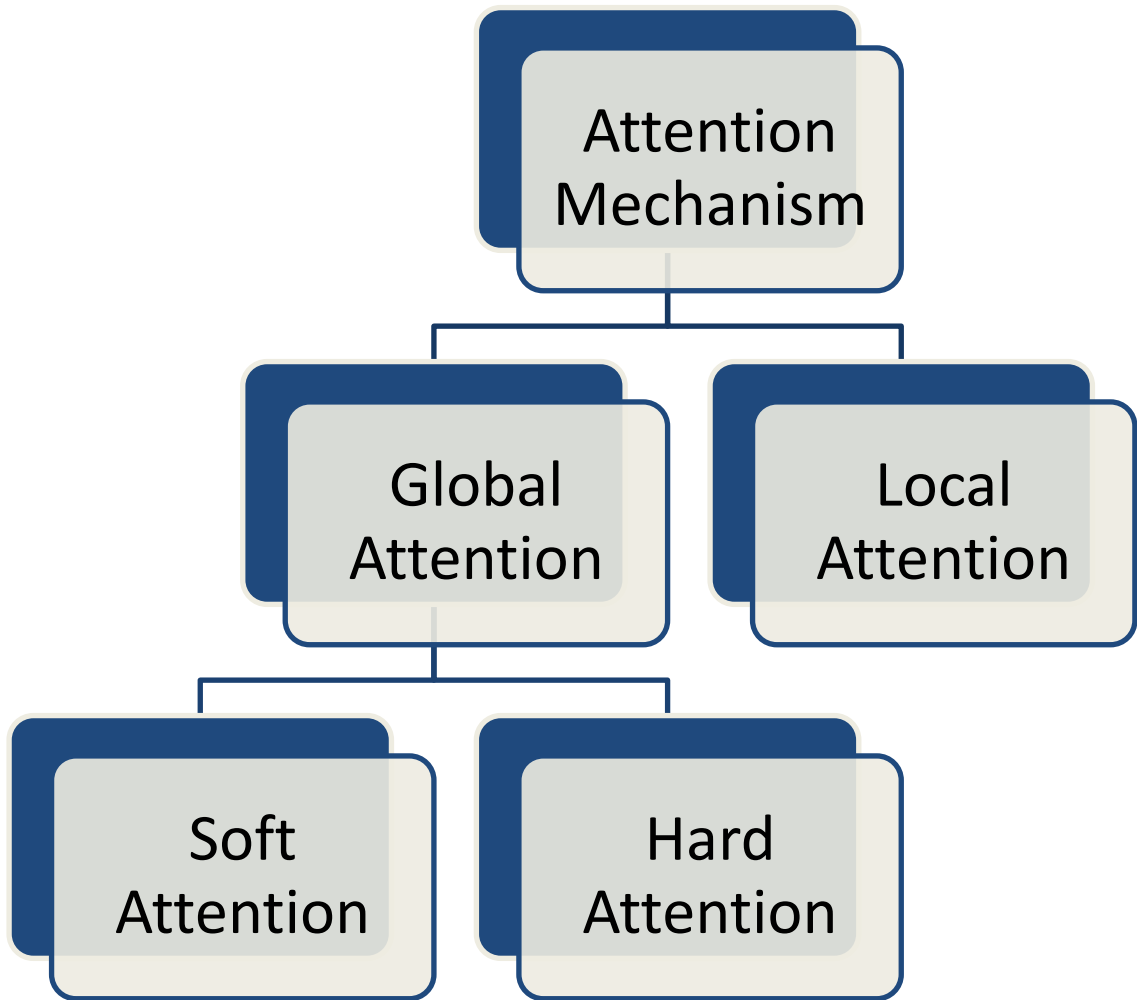
Alignment scores are computed using the current decoder hidden state and all encoder hidden states.

The context vector is concatenated with the current decoder hidden state and transformed via a feedforward layer before generating the output.

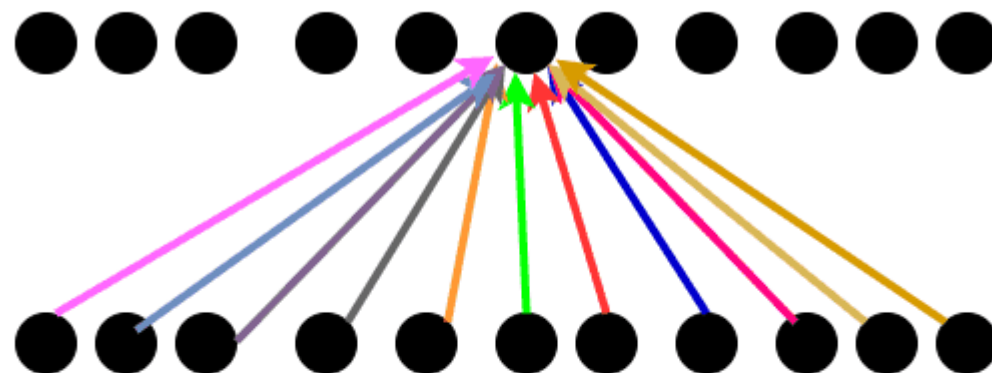
Bahdanau Attention



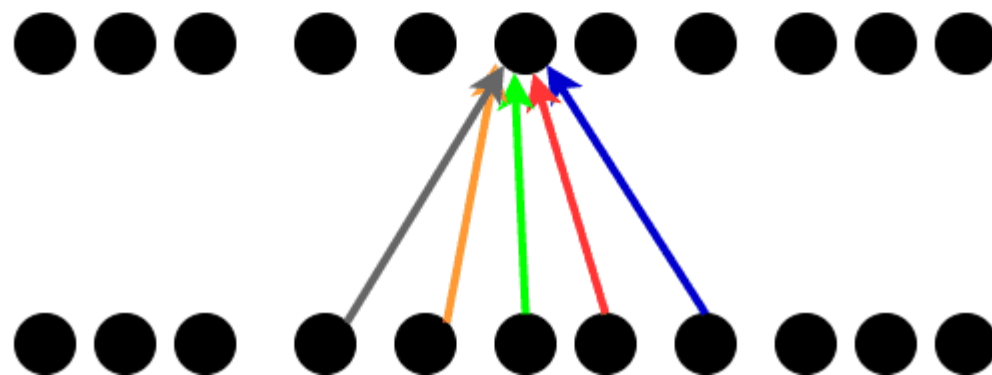




Global attention



Local attention



Global Attention

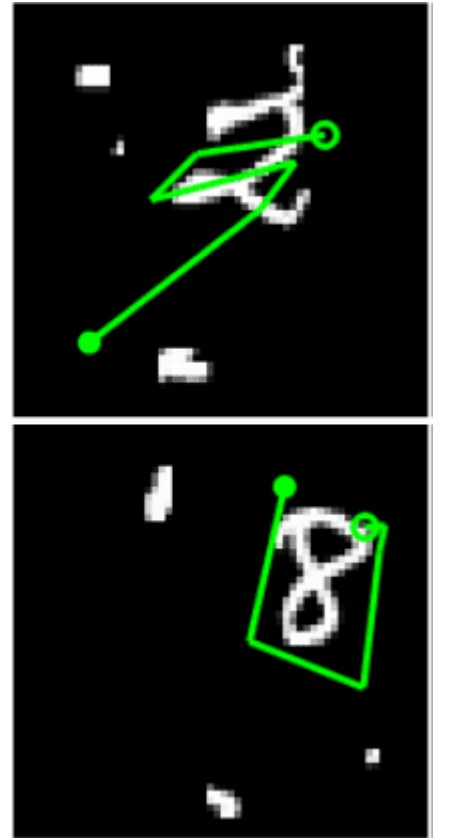
- Originally, Global Attention (defined by Luong et al 2015) had a few subtle differences from the Attention concept we discussed previously.
- The differentiation is that it considers all the hidden states of both the encoder LSTM and decoder LSTM to calculate a "variable-length context vector ct ", whereas Bahdanau et al. used the previous hidden state of the unidirectional decoder LSTM and all the hidden states of the encoder LSTM to calculate the context vector.
- The term "global" Attention is appropriate because all the inputs are given importance.

Global Attention Cont'd

- When a “global” Attention layer is applied, a lot of computation is incurred.
- This is because all the hidden states must be taken into consideration, concatenated into a matrix, and multiplied with a weight matrix of correct dimensions to get the final layer of the feedforward connection.
- To solve this we can prefer local attention
- **Soft Attention** is the global Attention where all image patches are given some weight; but in **hard Attention**, only one image patch is considered at a time.

Global Attention Cont'd

- **Hard means** that it can be described by discrete variables while **soft attention** is described by continuous variables.
- In other words, **hard attention** replaces a deterministic method with a stochastic sampling model.
- **starting** from a random location in the image tries to find the "important pixels" for classification
- Roughly, the algorithm has to choose a direction to go inside the image, during training.
- Cannot use SGD.

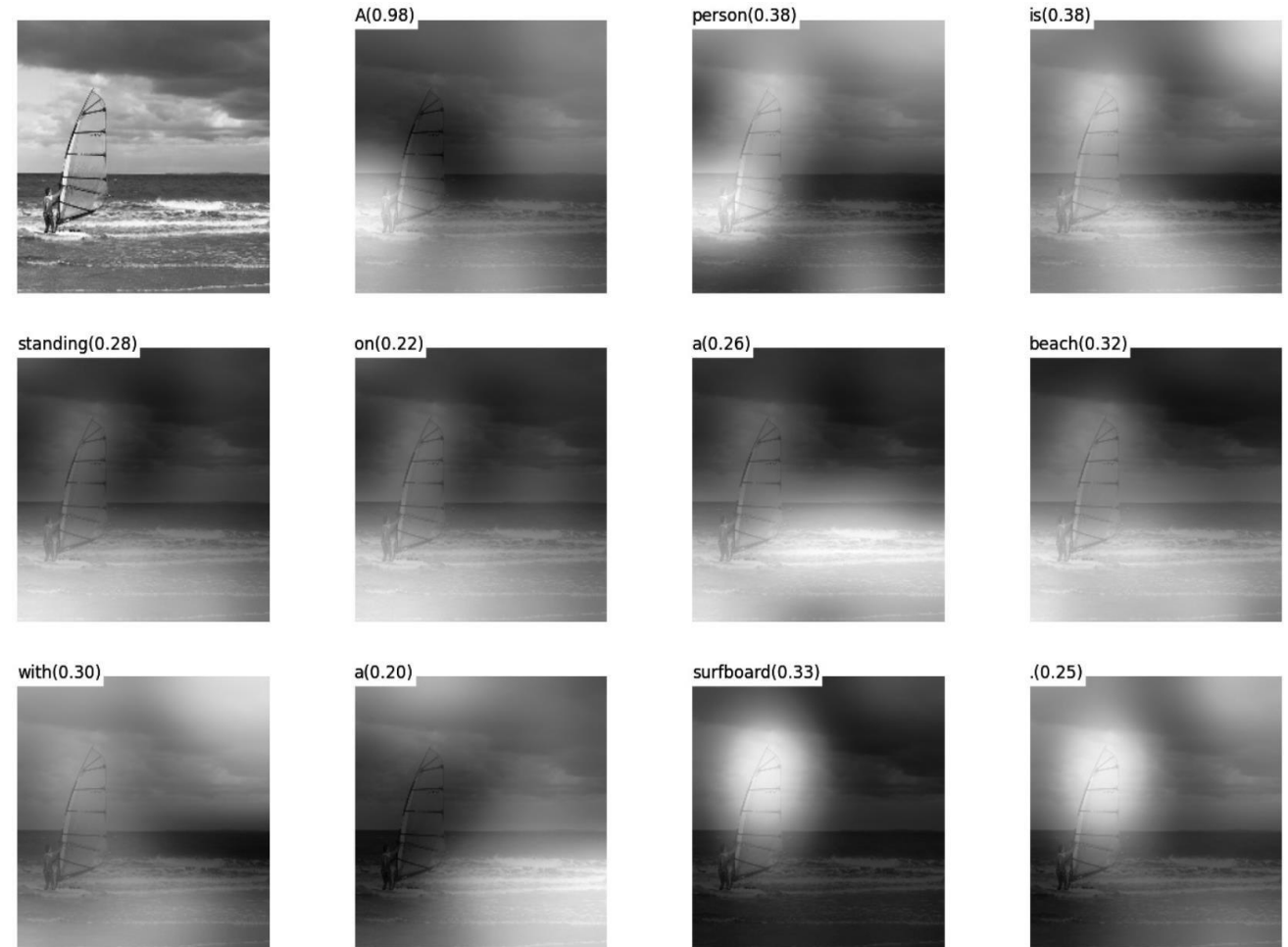


Difference Between Global vs Local Attention

- Computing attention over the entire input sequence as in **Global Attention** is sometimes unnecessary because despite its simplicity it can be **computationally expensive**. Thus Local Attention resulted as a solution for this as Local Attention considers only a subset of the input units/tokens.
- Comparatively Local Attention is seen as hard attention since we need to take hard decisions at first, to exclude some input points.
- In global attention, we require as many weights as the source sentence length whereas in Local attention it is less because attention is paced over only a few source states.

Application

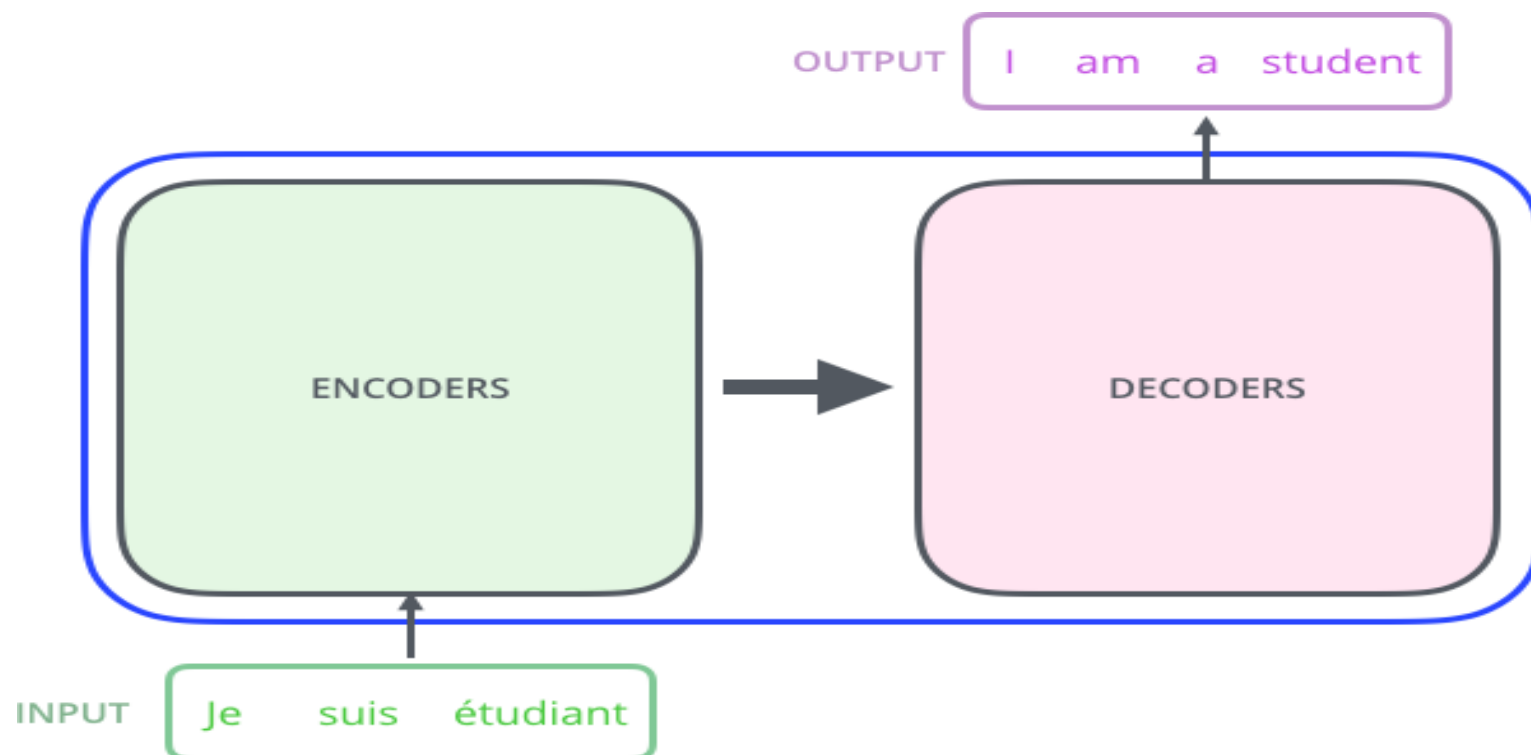
- They use a Convolutional Neural Network to “encode” the image, and a Recurrent Neural Network with attention mechanisms to generate a description. By visualizing the attention weights (just like in the translation example), we interpret what the model is looking at while generating a word:

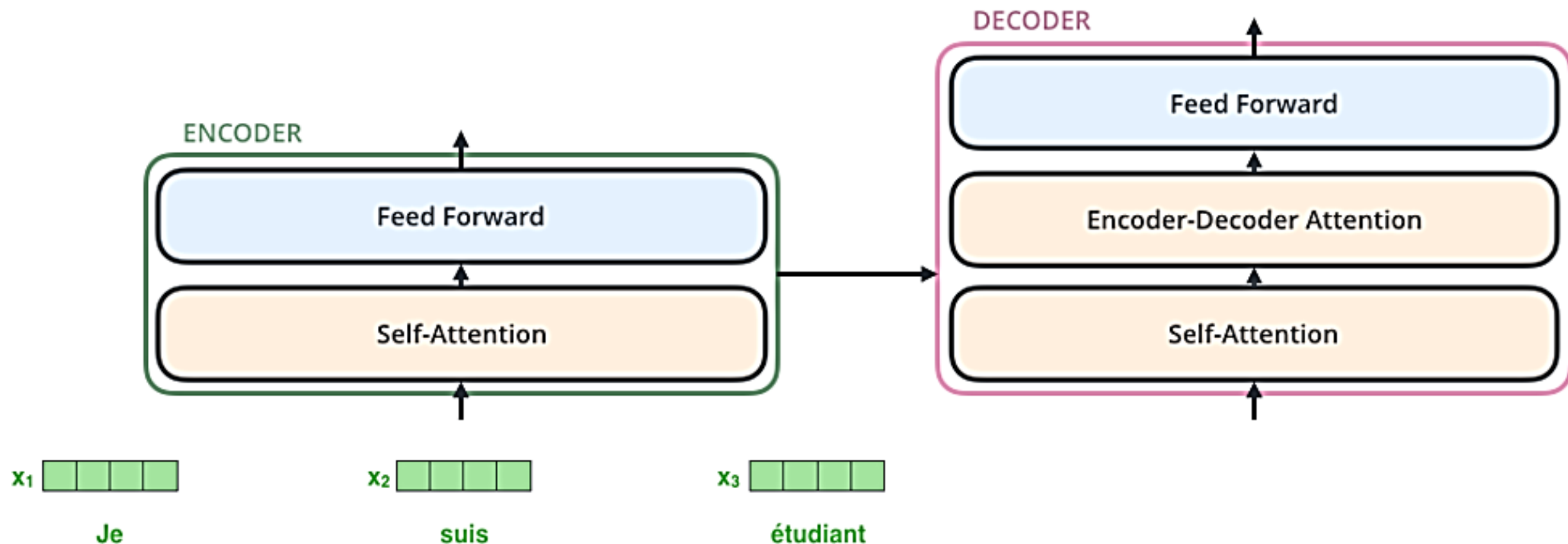


(b) A person is standing on a beach with a surfboard.

Self Attention

- The **Transformer** utilizes self-attention to **weigh the importance of different words in a sequence relative to each other**. This allows it to **capture long-range dependencies** efficiently.
- Unlike RNNs, Transformers **process the entire sequence simultaneously** rather than sequentially, making them **highly parallelizable and efficient**.
- Outperforms Google Neural Machine Translation (GNMT) model.
- Optimized for parallel computation, leading to **faster training times**.
- **Recommended by Google Cloud** for tasks requiring large-scale sequence modeling.





"love" \rightarrow $[0.72, -0.11, 0.50, \dots, 0.01]$ (e.g., 300 dimensions)

"like" \rightarrow $[0.70, -0.10, 0.52, \dots, 0.00]$ \leftarrow almost similar!

"banana" \rightarrow $[0.02, 0.83, -0.40, \dots, 0.92]$ \leftarrow very different

1. The Encoder's Function

Self-Attention Layer

- The **encoder** receives an input sentence (e.g., "Je suis étudiant").
- Before encoding a word, the encoder **looks at all the words** in the sentence using **self-attention**.
- **Why?** Because words depend on each other (e.g., in "New York City," "New" is important for "York City").
- The **self-attention mechanism** helps capture these dependencies by computing relationships between all words.

Feed-Forward Network

- The **output** of the self-attention layer is sent to a **feed-forward neural network (FFN)**.
- This FFN applies a **non-linear transformation** independently to each word.
- **Why independently?** Because self-attention already captured dependencies, so the FFN just refines word representations.

2. The Decoder's Function

Self-Attention Layer in Decoder

- The decoder first applies **self-attention** to its own input (translated words generated so far).
- However, it is **masked** to prevent looking at future words (during training).

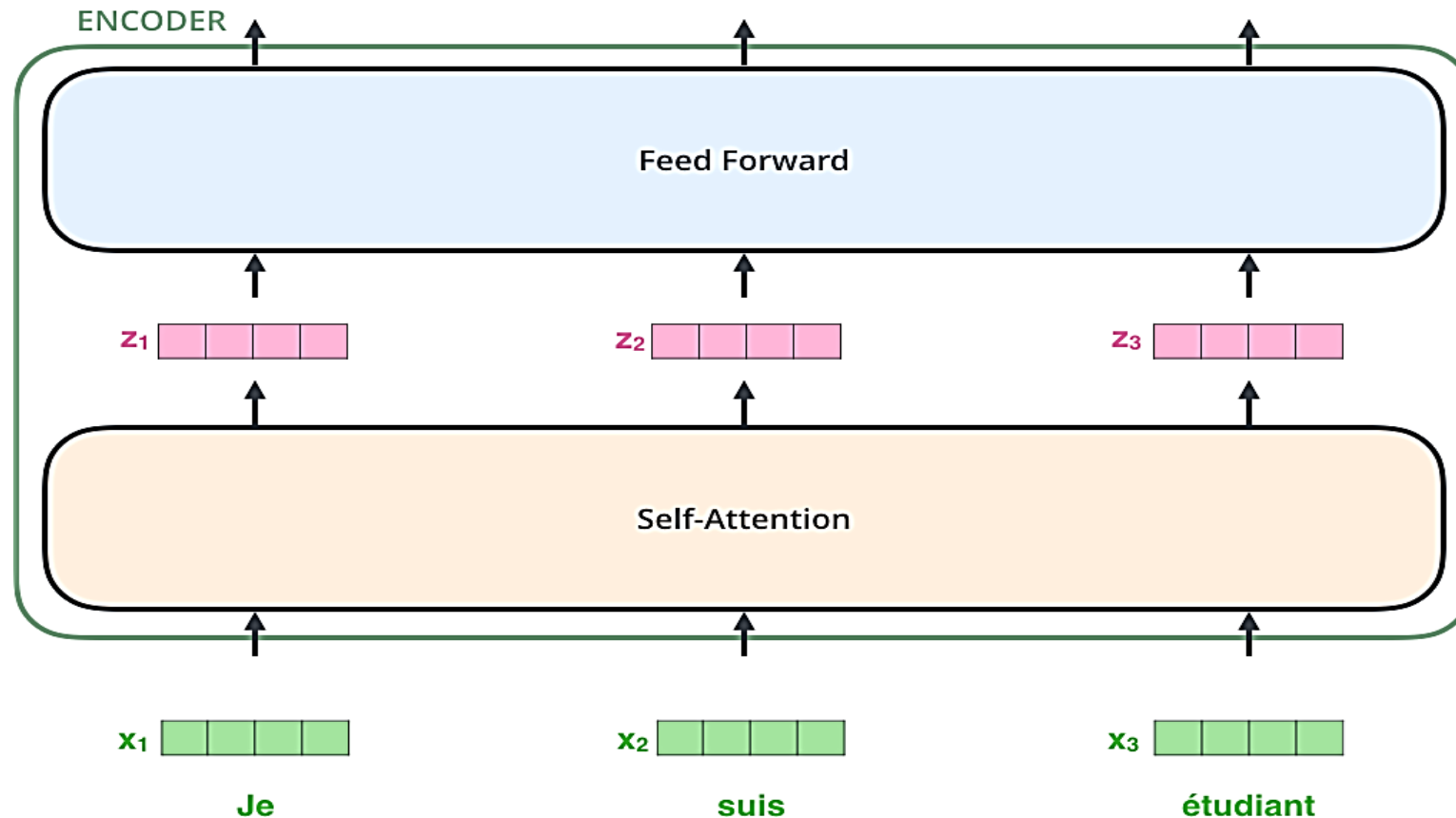
Encoder-Decoder Attention

- This is the new **attention layer** between self-attention and the feed-forward layer.
- It allows the **decoder to focus on relevant words from the encoder's output**.
- **Example:** If translating "Je suis étudiant" to "I am a student", the decoder needs to align:
 - "Je" → "I"
 - "suis" → "am"
 - "étudiant" → "student"

Feed-Forward Layer

- The output of the encoder-decoder attention layer is processed by an FFN (like in the encoder).

- The embedding only happens in the **bottom-most encoder**.
- The abstraction that is common to all the encoders is that they receive a list of vectors each of the size **512** -
- In the bottom encoder that would be the **word embeddings**, but in other encoders, it would be the output of the **encoder that's directly below**.

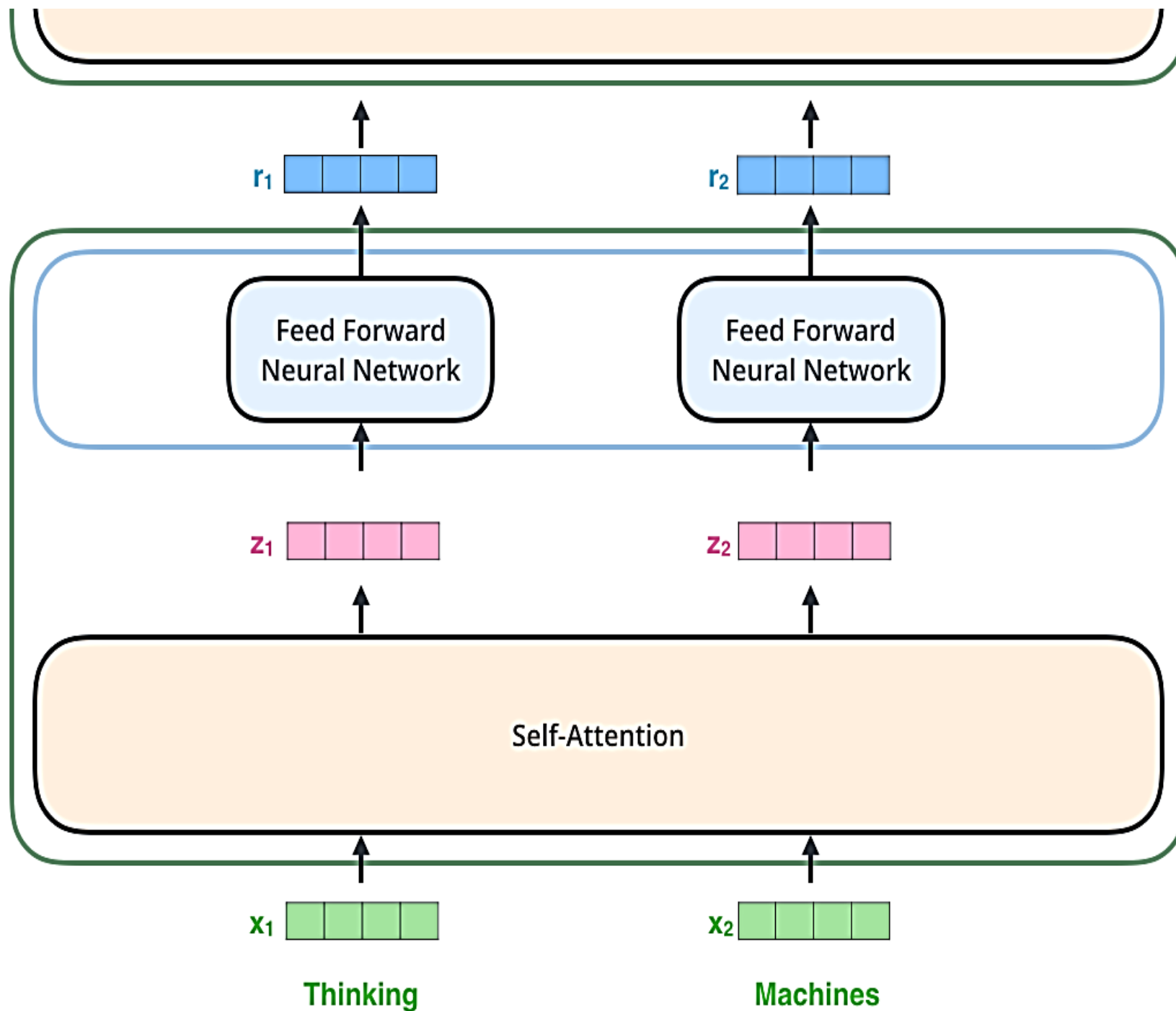


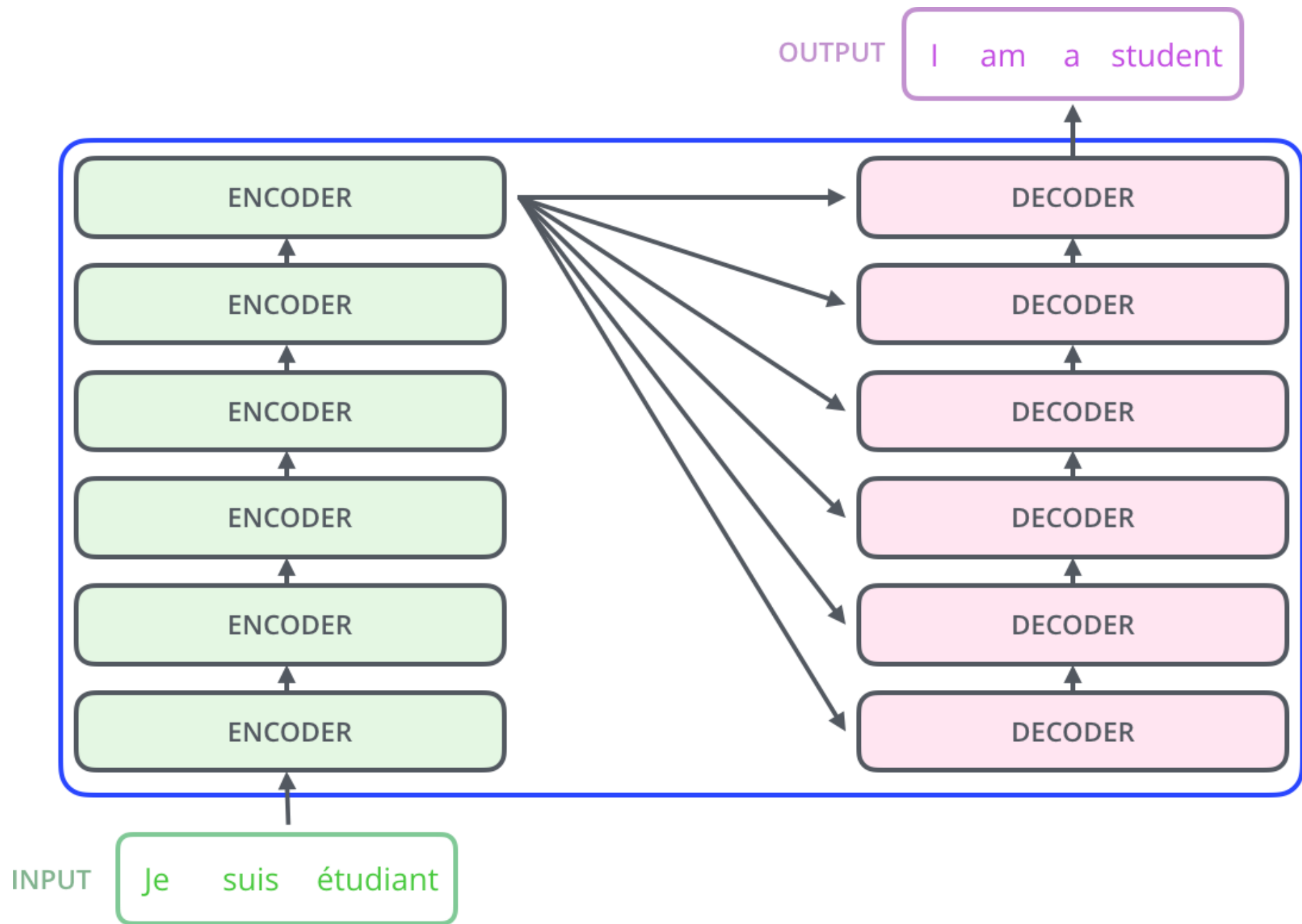
Self Attention Cont'd

- An encoder receives a **list of vectors as input**.
- It processes this list by passing these vectors into a '**self-attention**' layer, then into a feed-forward neural network, then sends out the output upwards to **the next encoder**.
- One key **property of the Transformer** is that **the word in each position flows** through its own path in the encoder.
- There are **dependencies between these paths in the self-attention layer**.
- The **feed-forward layer does not have those dependencies**, however, and thus the various paths can be **executed in parallel** while flowing through the feed-forward layer.

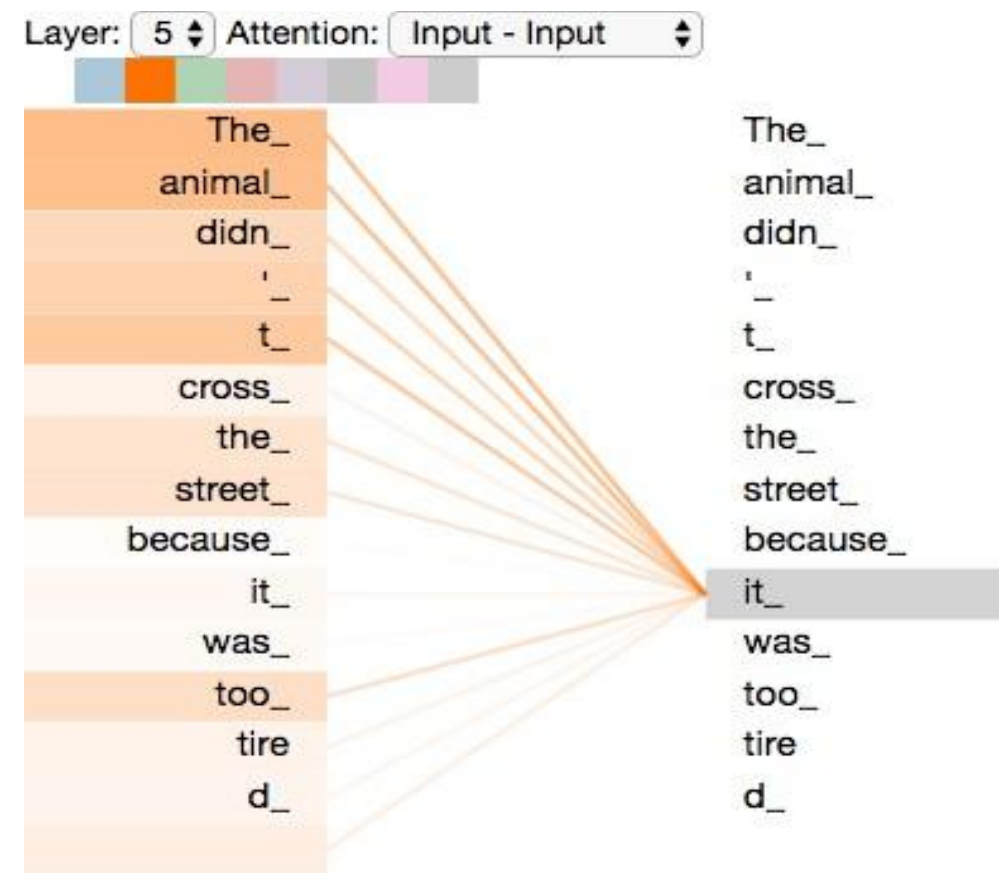
ENCODER #2

ENCODER #1





- The animal didn't cross the street because **it** was too tired
- As the model processes each word (each position in the input sequence), **self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.**
- **Maintaining a hidden state** allows an RNN to incorporate its representation of previous words/vectors it has processed with the current one it's processing.
- Self-attention is the method the Transformer uses to bake the "understanding" of other relevant words into the one we're currently processing.



Self Attention Cont'd

- The **first step** in calculating self-attention is to create **three vectors** from each of the encoder's input vectors (in this case, the embedding of each word).
- So for each word, we create a **Query vector**, a **Key vector**, and a **Value vector**.
- These vectors are created by multiplying the embedding by **three matrices** that we trained during the training process.

Input

Thinking

Machines

Embedding

X_1 

X_2 

Queries

q_1 

q_2 



W^Q

Keys

k_1 

k_2 



W^K

Values

v_1 

v_2 



W^V

Self Attention Cont'd

- The **second step in calculating self-attention** is to **calculate a score**.
- Calculating the self-attention for the first word in this example, "Thinking".
- Compute score for each word of the input sentence against this word.
- The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.
- The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring.
- So if we're processing the self-attention for the word in position #1, the first score would be the dot product of q_1 and k_1 .
- The second score would be the dot product of q_1 and k_2 .

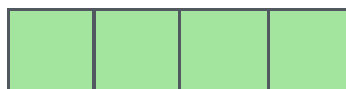
Input

Thinking

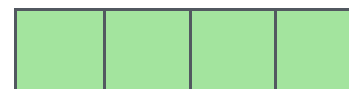
Machines

Embedding

x_1

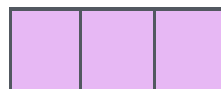


x_2

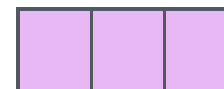


Queries

q_1



q_2

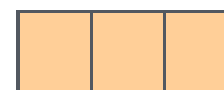


Keys

k_1



k_2



Values

v_1



v_2



Score

$$q_1 \cdot k_1 = 112$$

$$q_1 \cdot k_2 = 96$$

Self Attention Cont'd

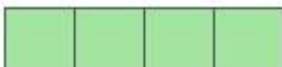
- The third and fourth steps are to divide the scores by 8 (the square root of the dimension of the key vectors used).
- This leads to having more stable gradients.
- There could be other possible values here, but this is the default), then pass the result through a softmax operation.
- Softmax normalizes the scores so they're all positive and add up to 1.

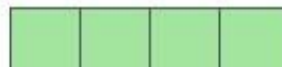
Input

Thinking

Machines

Embedding

x_1 

x_2 

Queries

q_1 

q_2 

Keys

k_1 

k_2 

Values

v_1 

v_2 

Score

$$q_1 \cdot k_1 = 112$$

$$q_1 \cdot k_2 = 96$$

Divide by 8 ($\sqrt{d_k}$)

14

12

Softmax

0.88

0.12

Self Attention Cont'd

- The **fifth step** is to multiply each value vector by the softmax score (in preparation to sum them up).
- The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).
- The **sixth step** is to sum up the weighted value vectors.
- This produces the output of the self-attention layer at this position (for the first word).

Final result

Input

Embedding

Queries

Keys

Values

Score

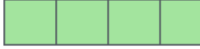
Divide by 8 ($\sqrt{d_k}$)

Softmax

Softmax
X
Value

Sum

Thinking

x_1 

q_1 

k_1 

v_1 

$q_1 \cdot k_1 = 112$

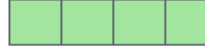
14

0.88

v_1 

z_1 

Machines

x_2 

q_2 

k_2 

v_2 

$q_2 \cdot k_2 = 96$

12

0.12

v_2 

z_2 