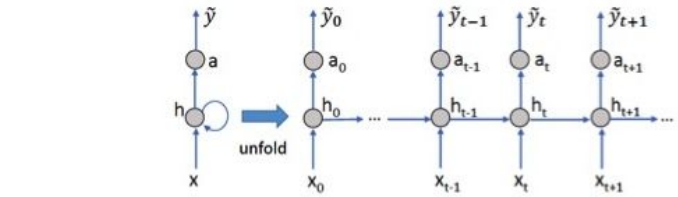
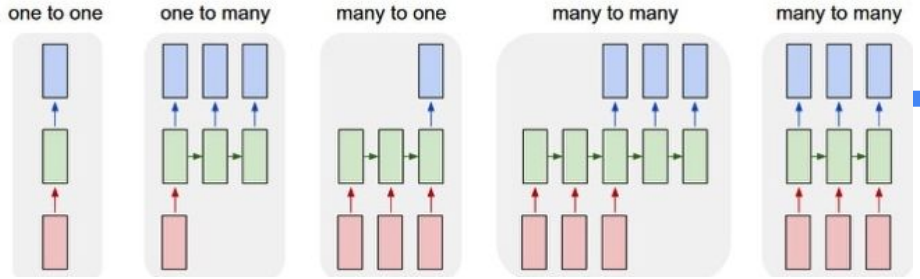
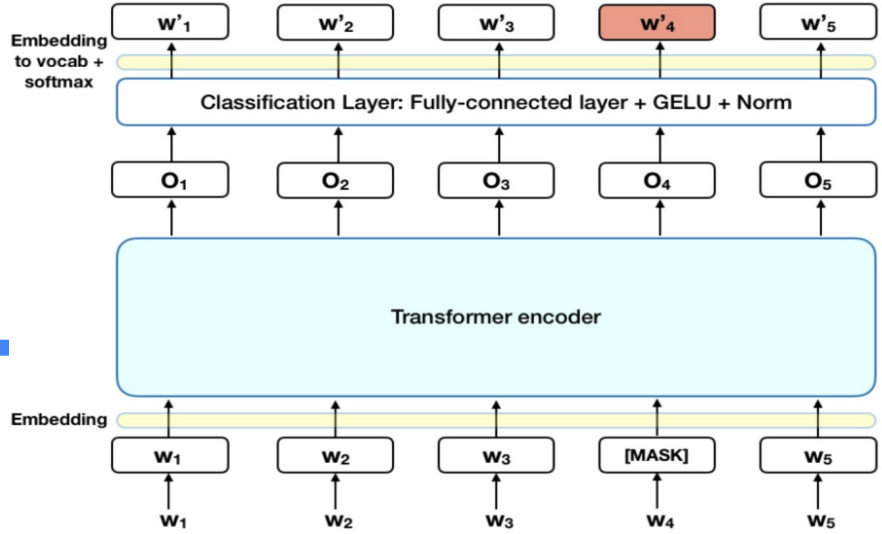
Recurrent Neural Nets & Transformers: Intro







Michael(Mike) Erlihson

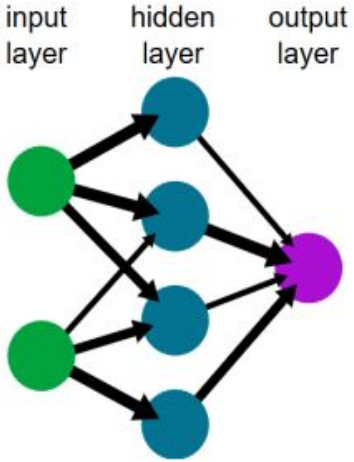
February, 2021

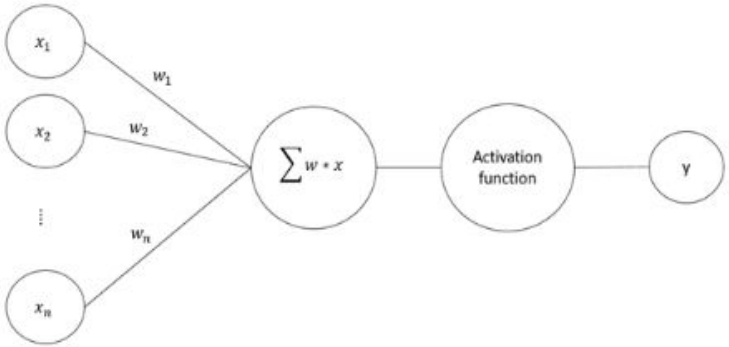
Agenda

- What is Recurrent Neural Network (RNN)
- Basic RNN Architecture
- More advanced RNN architectures
- Attention Layer in RNN
- Transformer: Self-Attention
- Transformer: Positional Encoding
- Transformer: Encoder and Decoder Architecture

Neural Network: Recap

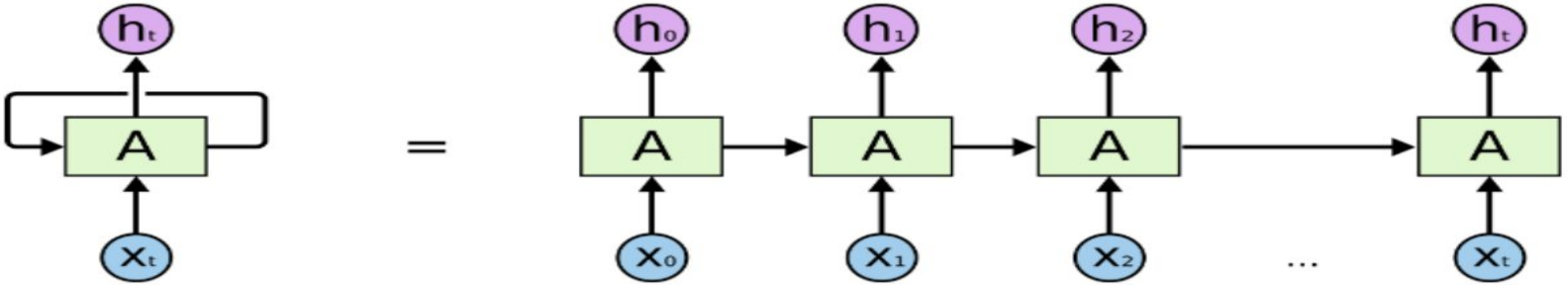
- Neural Network is a directed graph data structure
- Neural network consists of weights and nodes
- Hidden layers and output layer store a calculated value based on weights and previous layer node values
- Weights and value calculations can be represented as matrices
- Activation function is applied on the sum of weights*input values





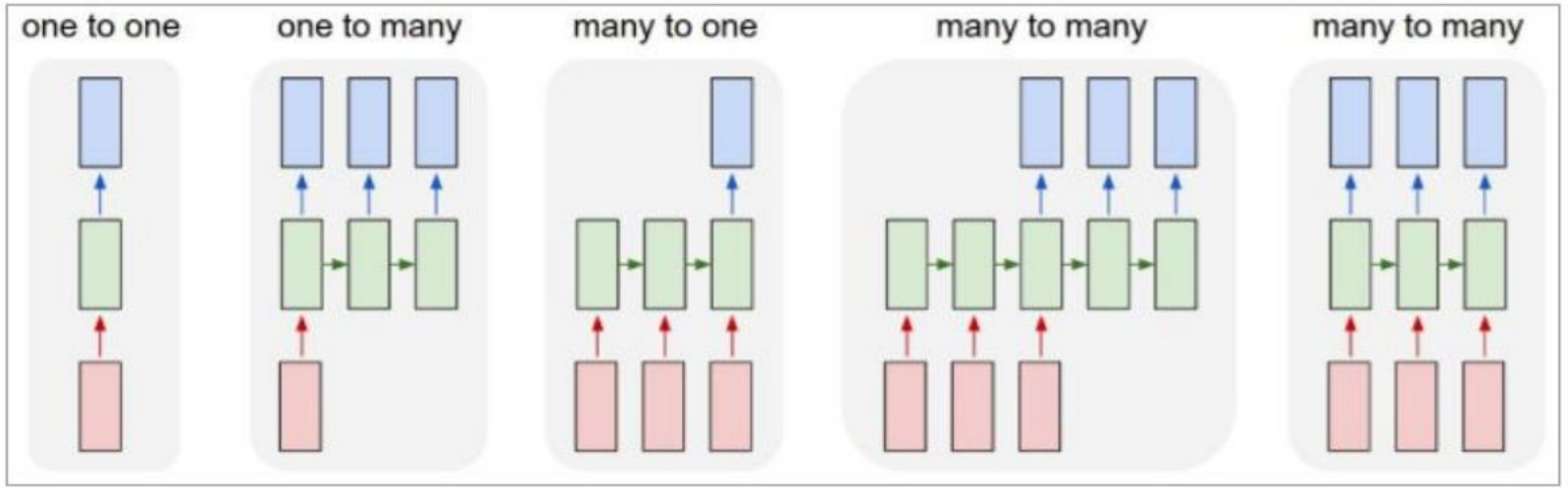
What is RNN?

- RNN is designated to tackle sequential data
  - Predict next word in a sentence, next vowel in speech, next frame in video…
- RNNs are called **recurrent** because they:
  - Perform the same task for every element of a sequence, with the output being depended on the previous computations
  - Have a "memory" capturing information about what has been calculated so far



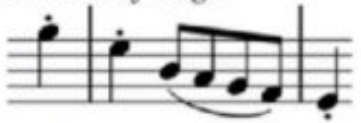
Motivation behind RNN: Why bother?

- RNNs allow us to operate over ***sequences*** of vectors: **sequences** in the input, the output, or in the most general case **both**



RNN: Vast variety of possible tasks

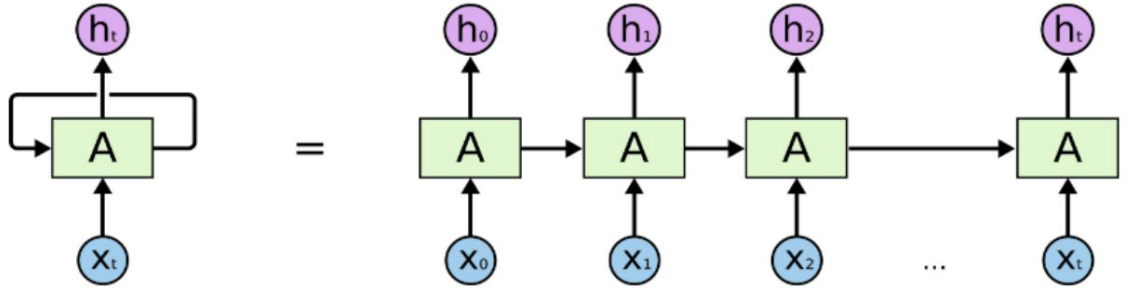
Examples of sequence data

Speech recognition		→	"The quick brown fox jumped over the lazy dog."
Music generation	∅	→	
Sentiment classification	"There is nothing to like in this movie."	→	★☆☆☆☆
DNA sequence analysis	AGCCCCTGTGAGGAACTAG	→	AGCCCCTGTGAGGAACTAG
Machine translation	Voulez-vous chanter avec moi?	→	Do you want to sing with me?
Video activity recognition		→	Running
Name entity recognition	Yesterday, Harry Potter met Hermione Granger.	→	Yesterday, Harry Potter met Hermione Granger.

RNN: Basic Unit

- **Definition:** An RNN cell, in the most abstract setting, is anything that has a state and performs some operation that takes a matrix of inputs
- RNN cells distinguish themselves from the regular neurons in the sense that they have a state and thus can remember information from the past
- The state of the cell helps it remember the past sequence and combine that information with the current input to provide an output

”Unrolled RNN”



RNN: Hidden State

Recurrent Neural Network

Previous hidden state

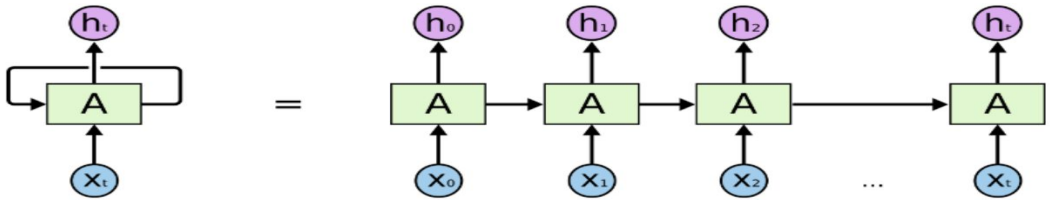
Input

$$h_t = f_W(h_{t-1}, x_t)$$

Sometimes a sequence in a vector

New hidden state

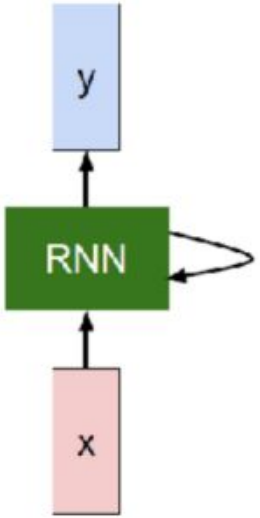
Activation function with parameters - W



Vanilla (Basic) RNN Scheme

(Vanilla) Recurrent Neural Network

The state consists of a single *"hidden"* vector **h**:



$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

In training process we are calculating the different weights

Weights for every new input

Weights for updating the new hidden state

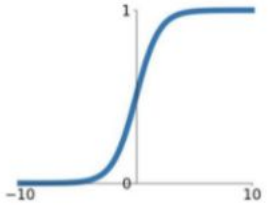
Weights for the output

Reminder: Activation Functions

Activation Functions

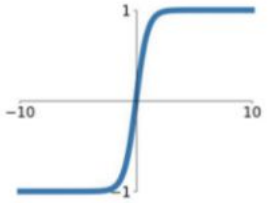
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



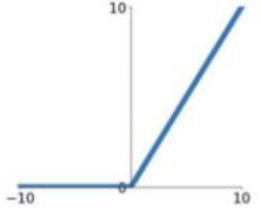
tanh

$$\tanh(x)$$



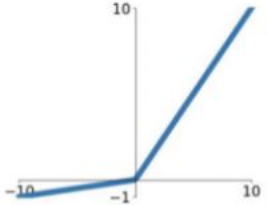
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

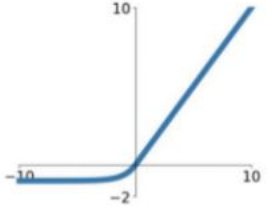


Maxout

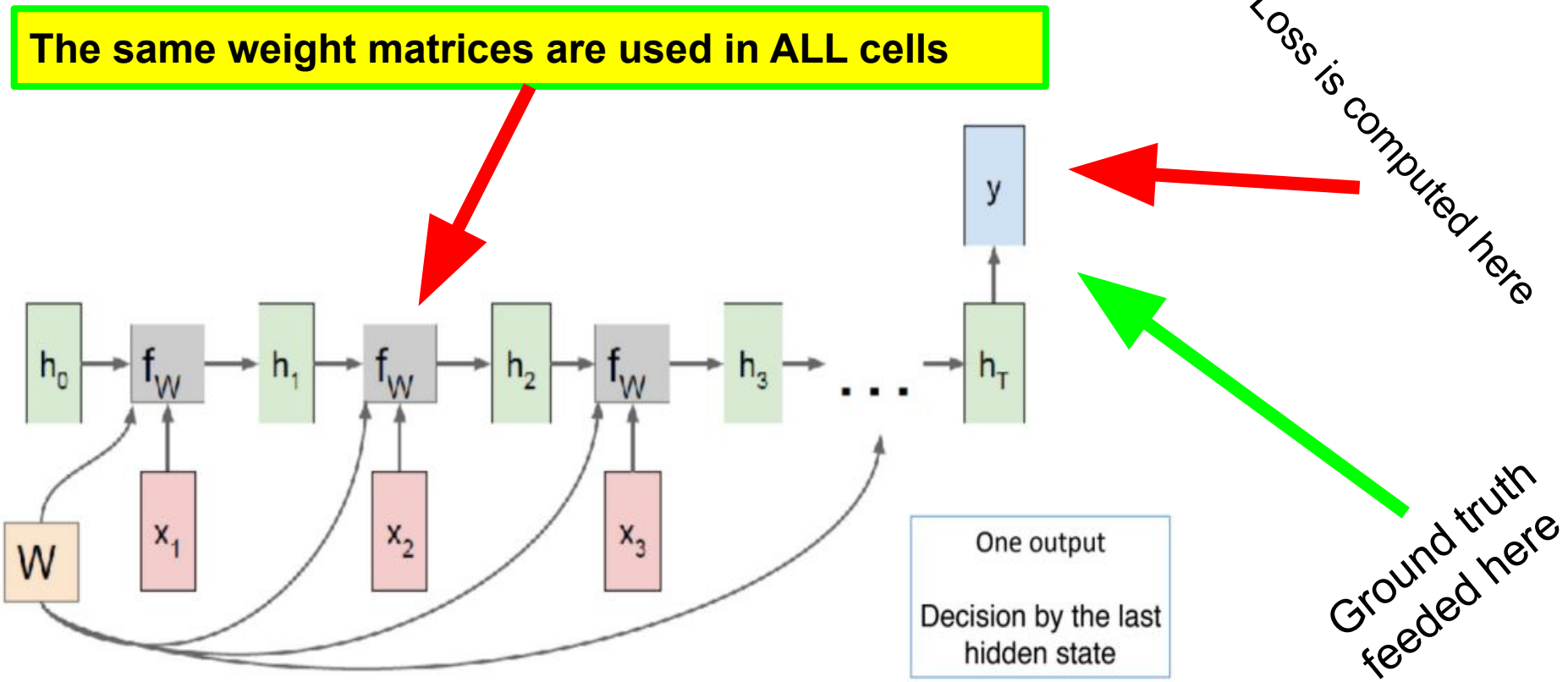
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

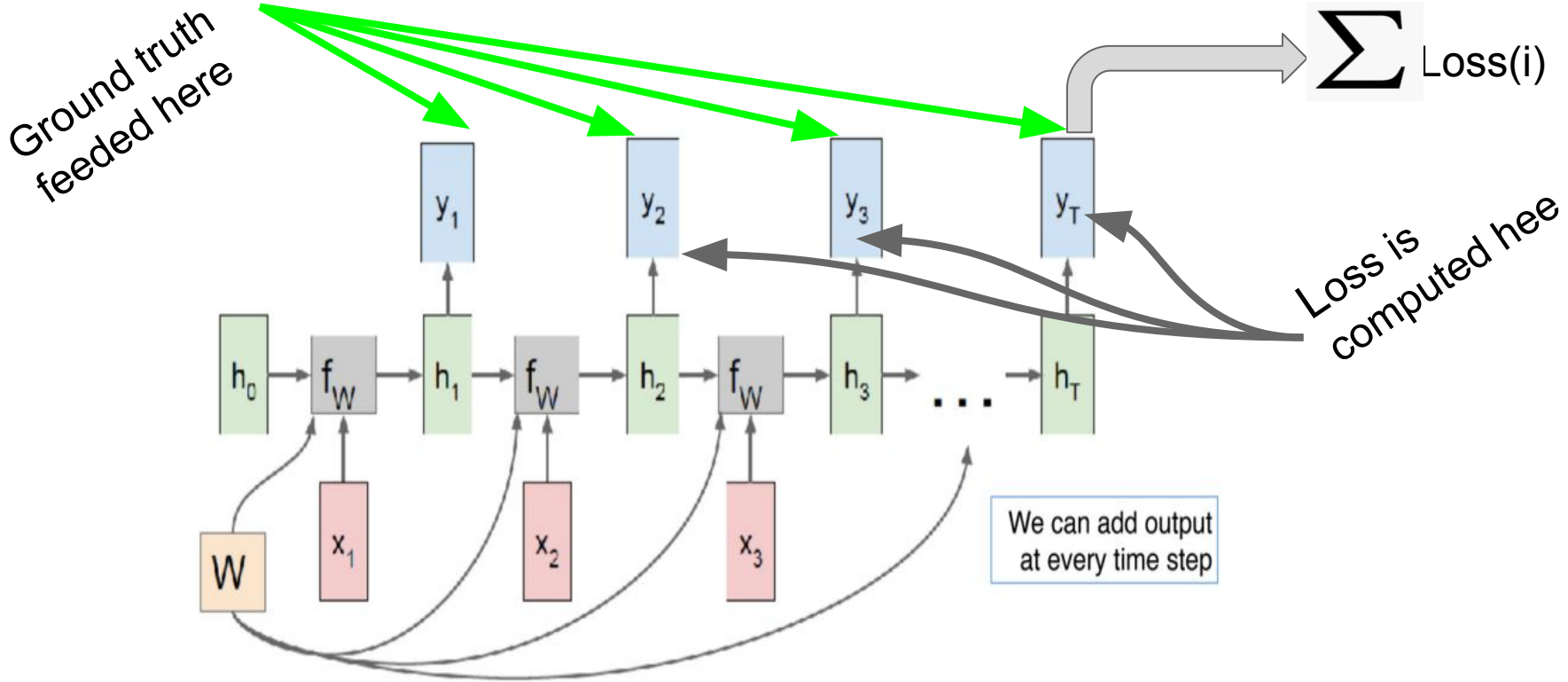
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



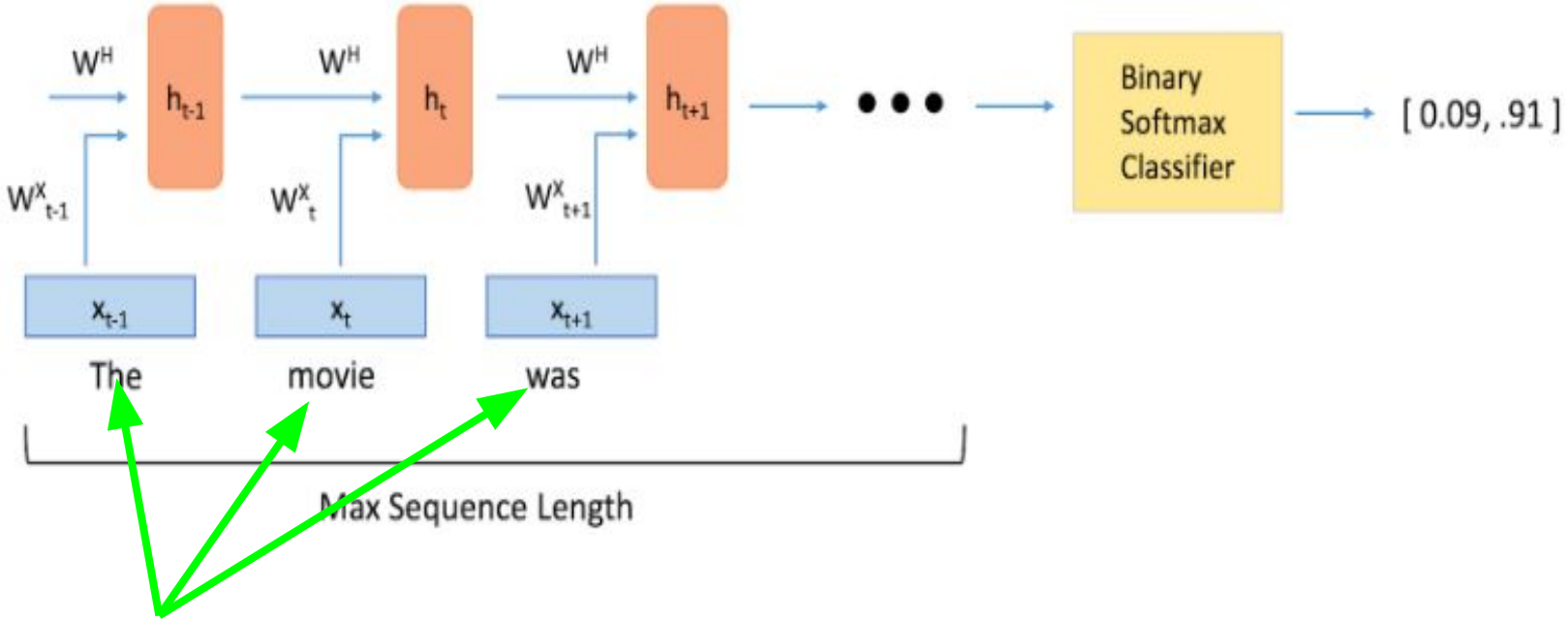
RNN: Training Procedure (Many To One)



RNN: Training Procedure (Many To Many)

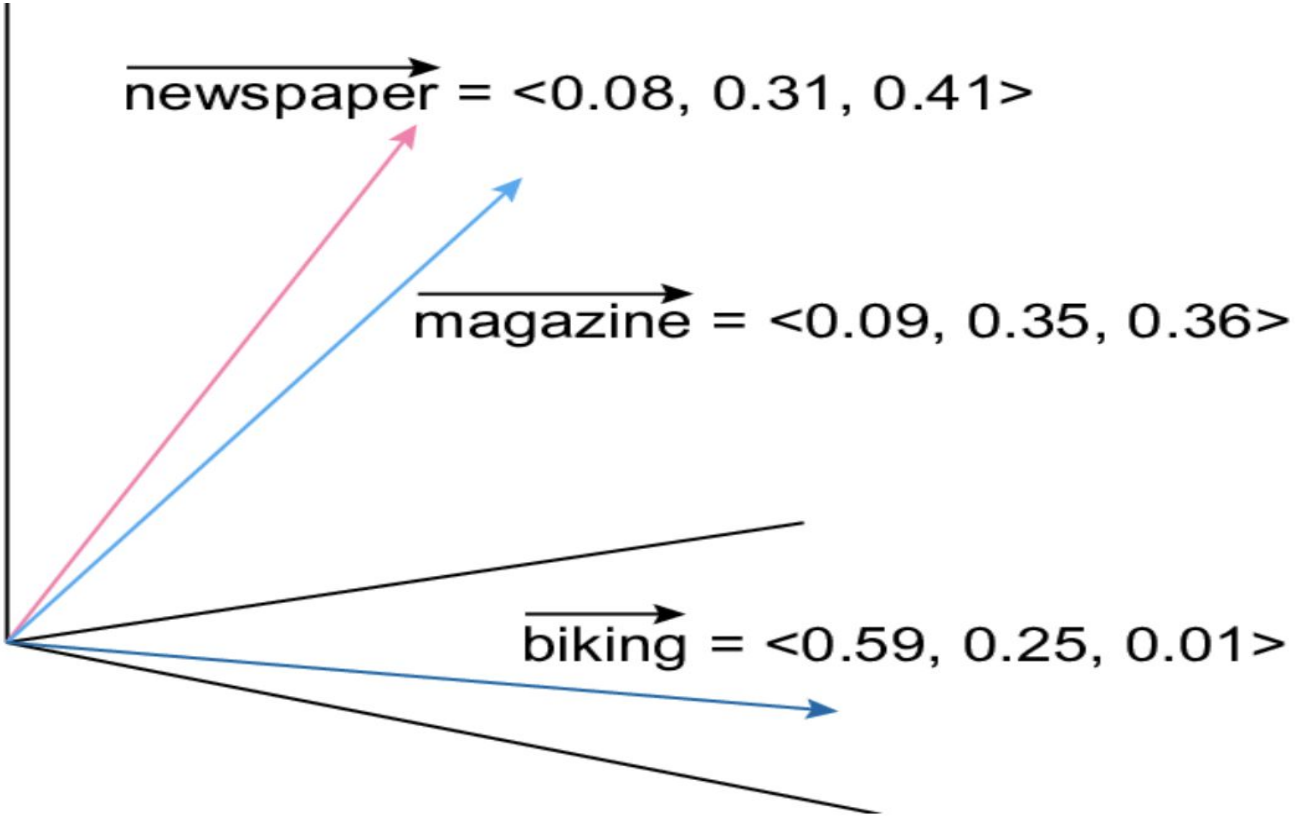


RNN Application: Sentiment Analysis



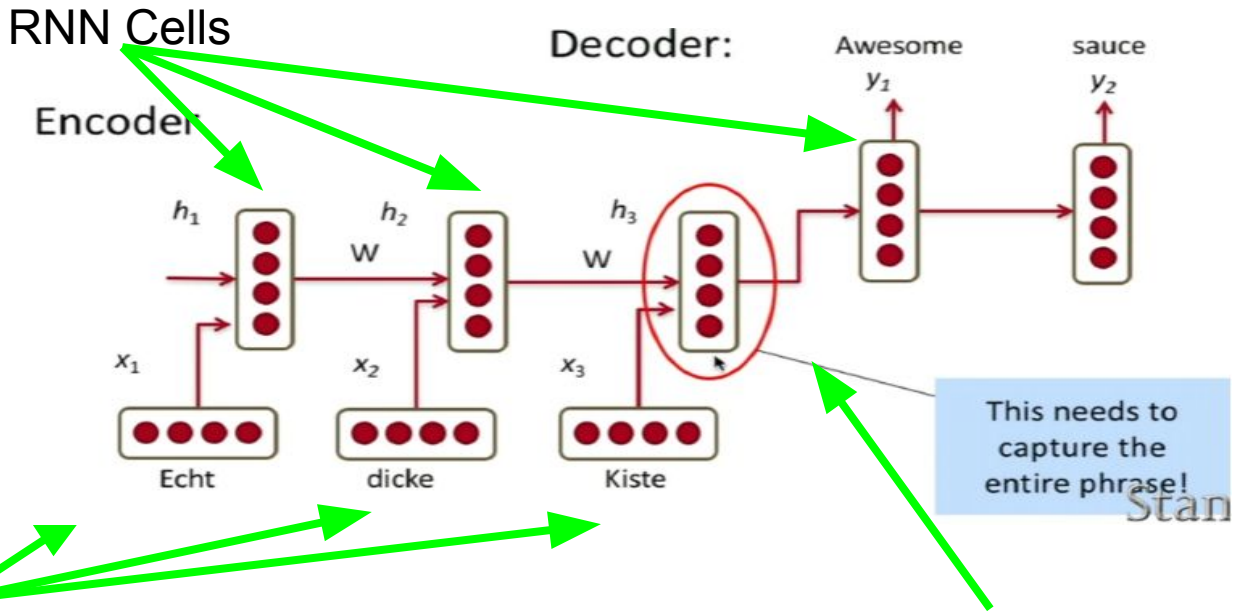
Word Embeddings

Reminder: Word Embedding



RNN Application: Machine translation

Main Task: Predict next word

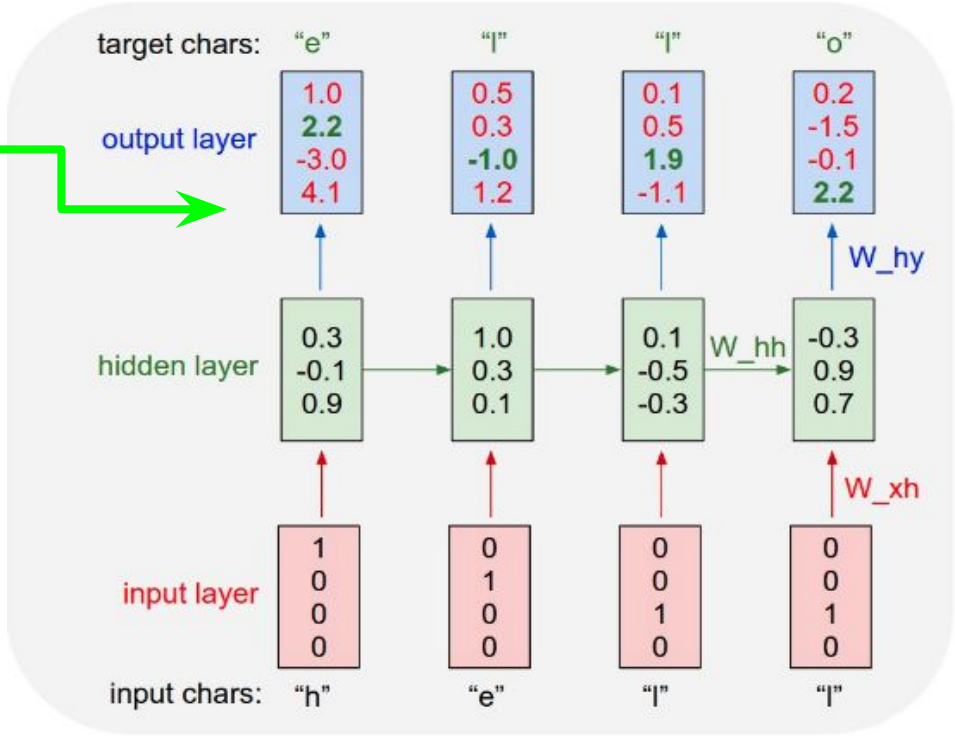


RNN: Character-level Language Model

Tries to predict the next chars in the sequence

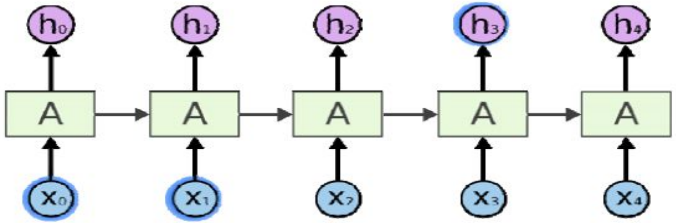
Vocabulary = [h, e, l, o]

Example training sequence:
“hello”

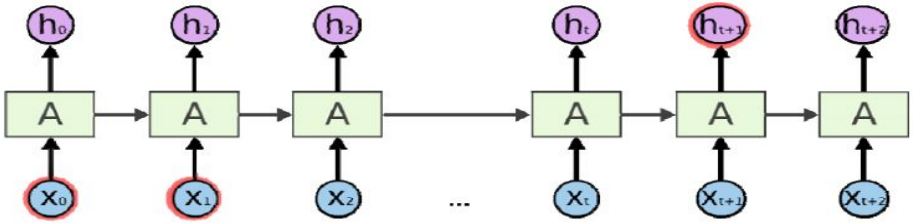


Limitations of Vanilla RNN

- Vanilla RNN works well for a small time step (remembers two-three previous hidden states)
- However, the sensitivity of the input values decays over time in a standard RNN



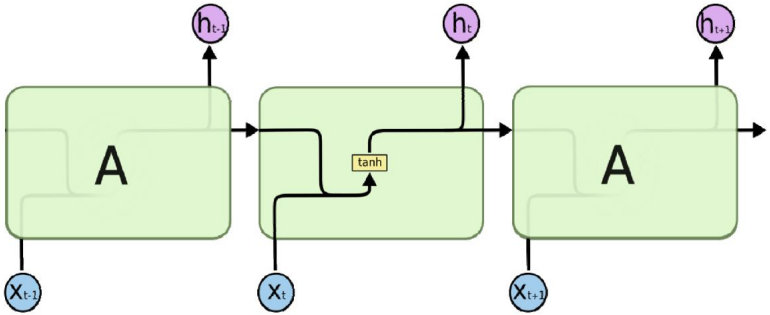
"the clouds are in the sky"



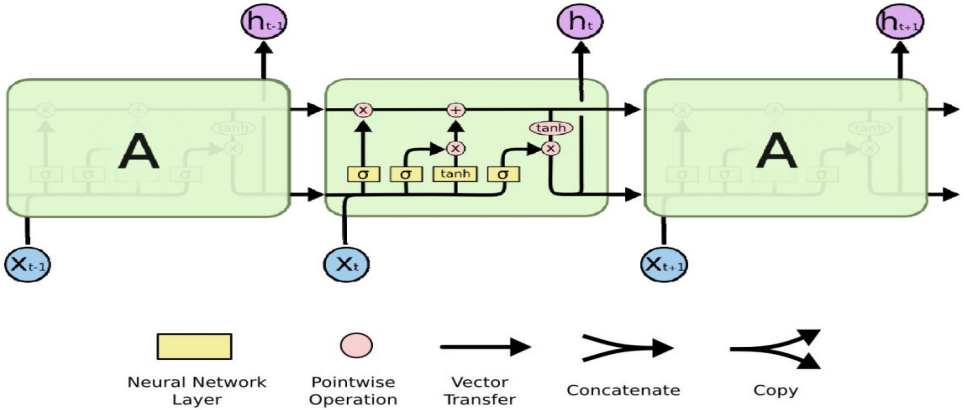
"I grew up in France
…
I speak fluent French."

LSTM: long-short term memory

A **standard RNN** contains a single layer in the repeating module



LSTM: A special kind of RNN for learning long-term dependencies

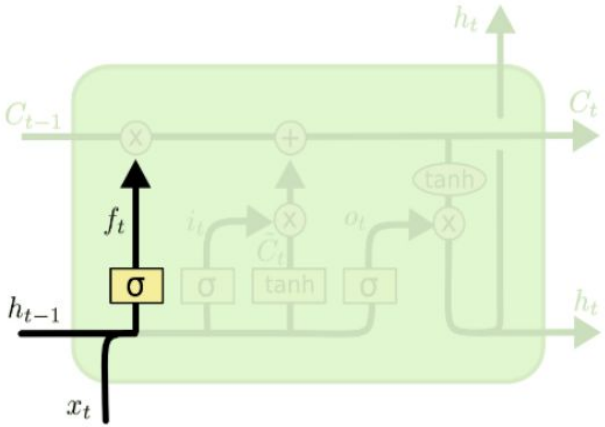


Hmm, looks complicated…. Don't worry, we are going to figure it out

LSTM: Main Principle (in simple words)

- **LSTM vs RNN**: similar control flow as a recurrent neural network
- **LSTM vs RNN**: both process data passing on as it propagates forward
- **LSTM vs RNN:** differences are the **operations** within the LSTM's cell
- LSTM cell operations are used to allow it **to keep or forget** information
- LSTM's core concept: **cell state(memory, C_t)** and its various gates.

LSTM, Step by Step: Keep Gate



"How much" of the previous cell state we want to keep/forget

N x (N+M)

N × 1

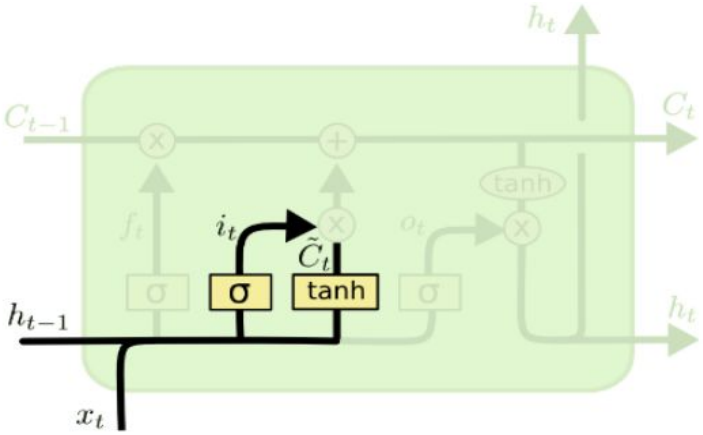
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

N × 1 M × 1

N × 1

f close to **0** means that we want to **forget most** of the previous state

LSTM, Step by Step: Write Gate



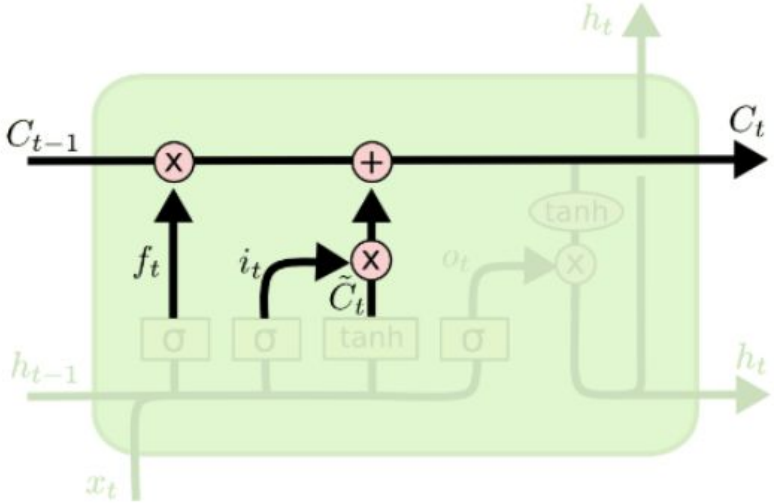
"How much" of the cell state "candidate" we want to keep/forget

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

Cell state candidate

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

LSTM, Step by Step: Update gate



New cell state

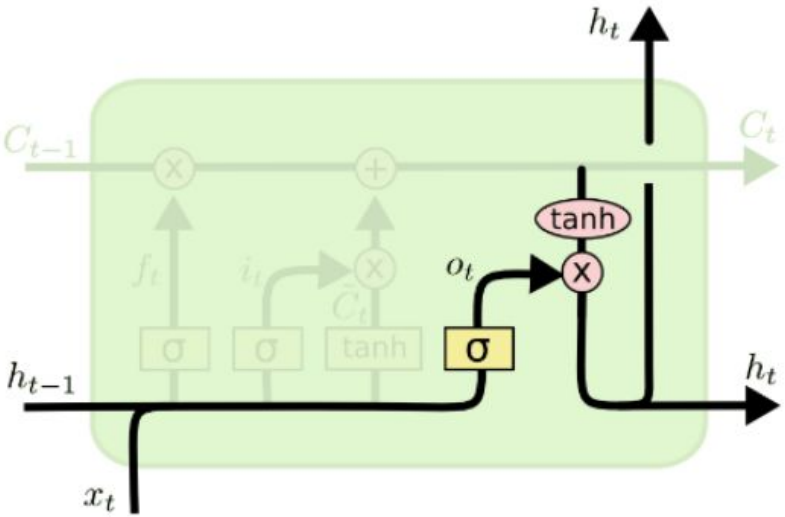
Previous cell state

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

Combine “portion” of previous cell state and “portion” candidate cell state:
Long-Short Memory

Cell state candidate

LSTM, Step by Step: Read(Output) gate



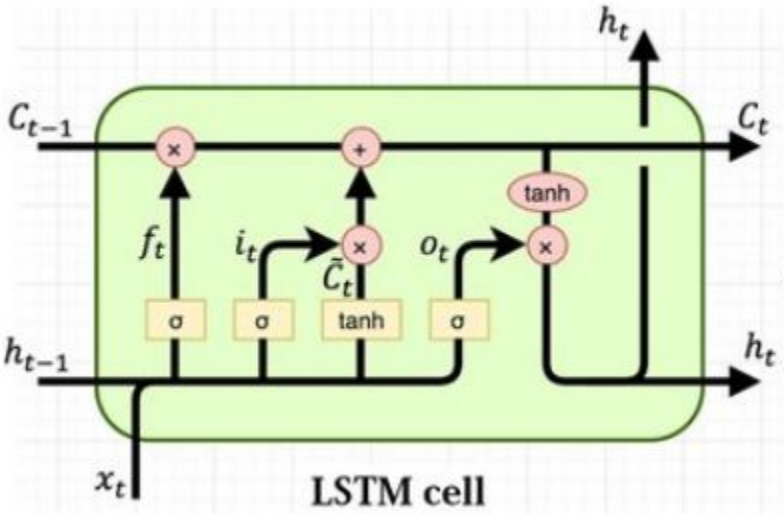
Standard RNN Output Computation

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

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

Use cell state to compute new hidden state

LSTM: All pieces together

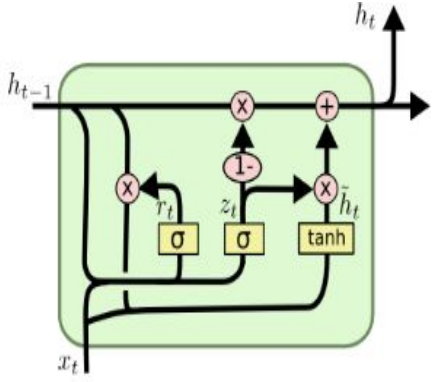


$$i_t = \sigma\left(x_t U^i + h_{t-1} W^i\right)$$
$$f_t = \sigma\left(x_t U^f + h_{t-1} W^f\right)$$
$$o_t = \sigma\left(x_t U^o + h_{t-1} W^o\right)$$
$$\tilde{C}_t = \tanh\left(x_t U^g + h_{t-1} W^g\right)$$
$$C_t = \sigma\left(f_t * C_{t-1} + i_t * \tilde{C}_t\right)$$
$$h_t = \tanh(C_t) * o_t$$

Not so complicated, right?

Gated Recurrent Unit(GRU): Simplified LSTM

- Introduced by Cho, et al. (2014).
- Got rid of the cell state and used the hidden state to transfer information.
- Has only two gates, a reset gate and update gate

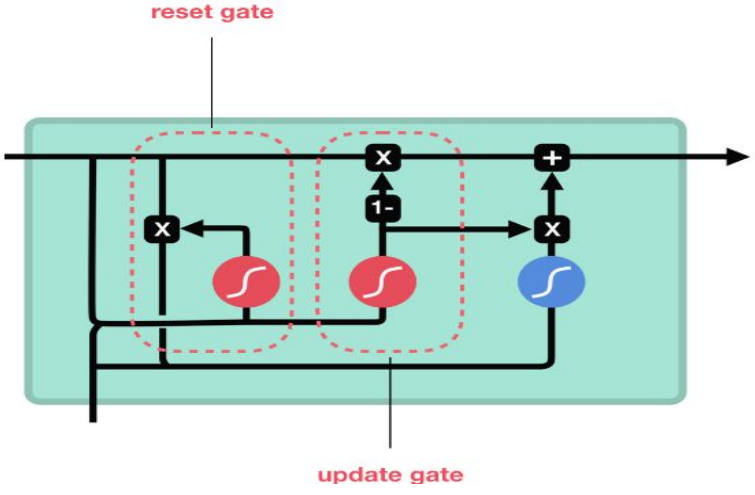


$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

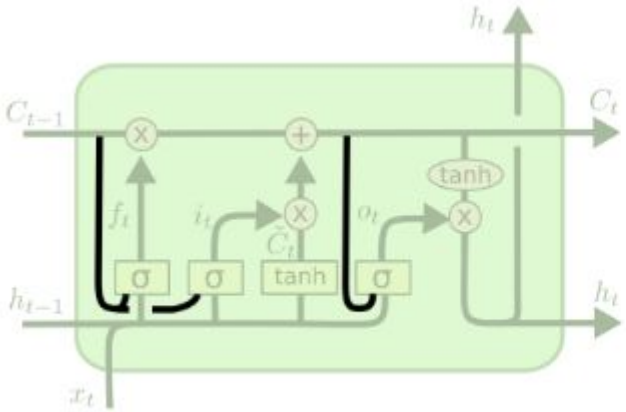
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$



LSTM with Peephole Connection

- Adds “peepholes” to all the gates
- Let the gate layers look at the cell state
- May use some peepholes and not others.



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

BiDirectional RNNs

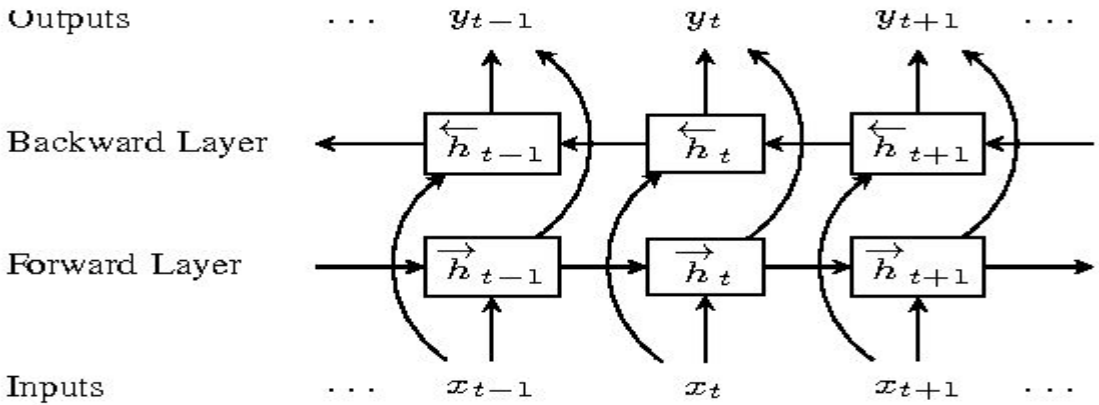
Let's say that we want to do name entity recognition. Following two examples are included in the training set; 'He said, "Teddy bears are on sale!"', and 'He said, "Teddy Roosevelt was a great President!"'.

If the information flows only from left to right, our model has no way to learn that "Teddy" in the second example is a name.

Solution: Bidirectional RNN

BiDirectional RNNs, Cont'

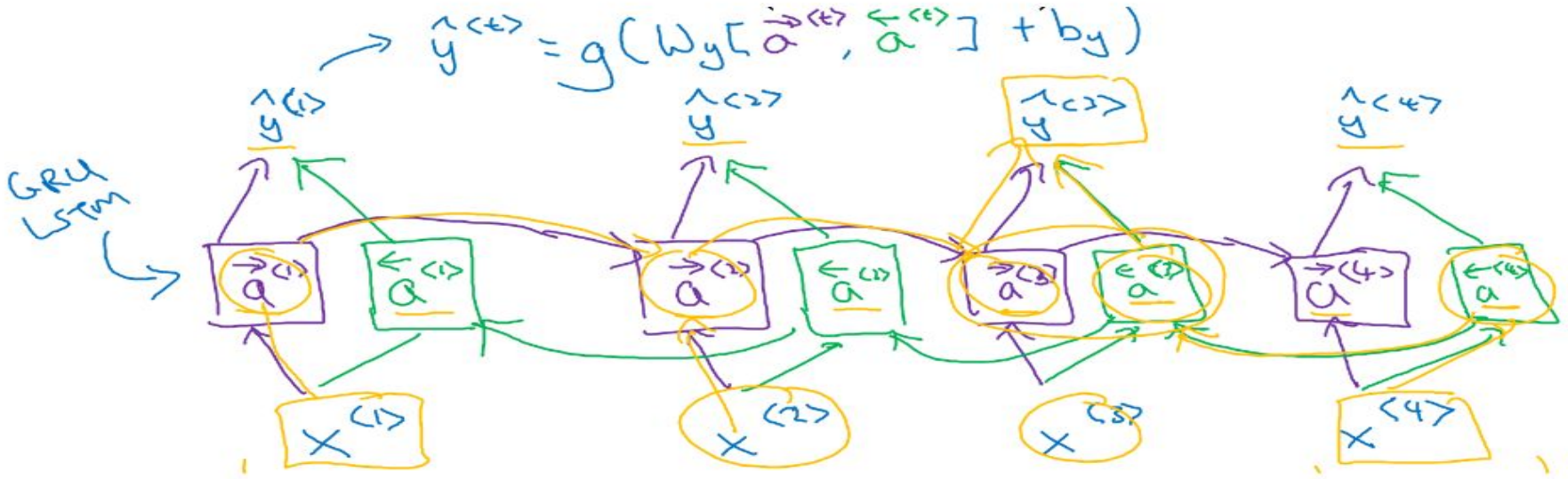
- 2 forward propagations; left to right and right to left
- For each layer combine activations from two forward props to output prediction.



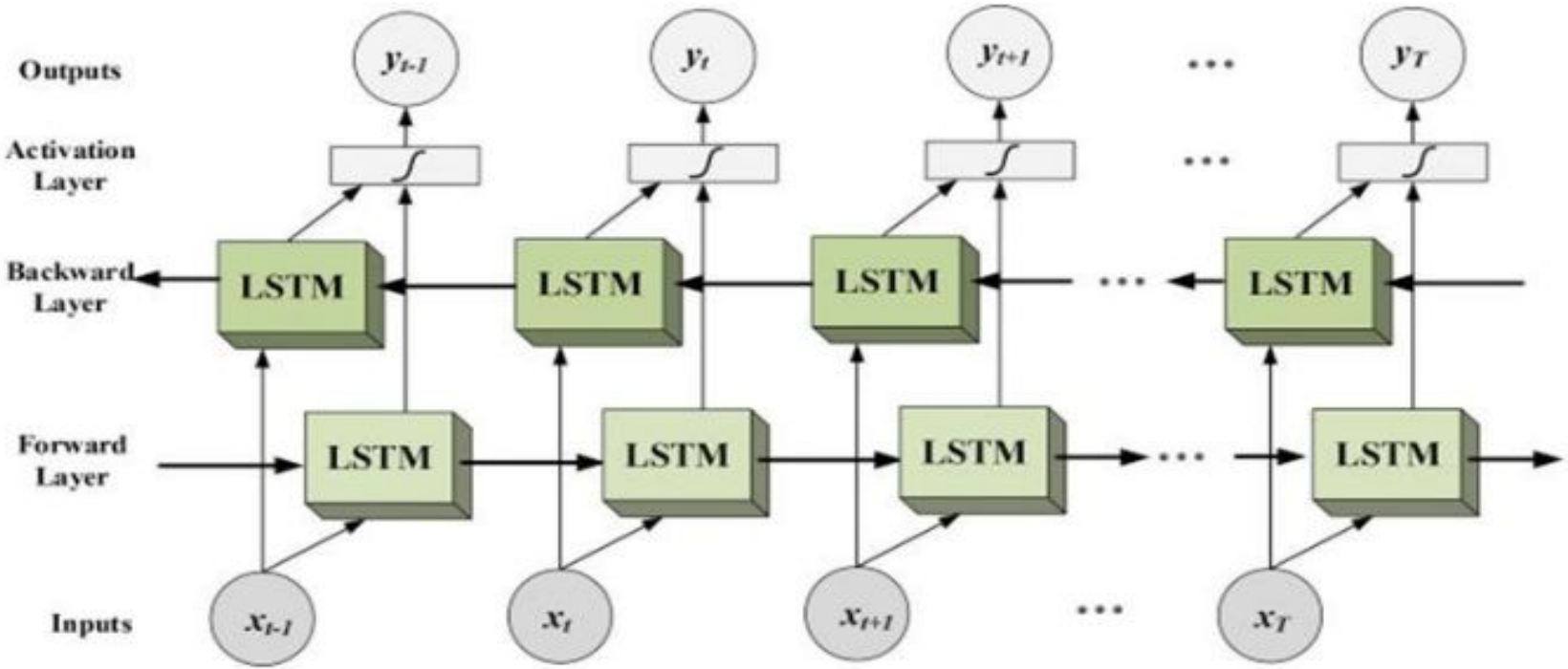
Bi Directional RNN for word sequence

BiDirectional RNNs, more detailed view

$$\hat{y}^{<t>} = g(W_y[\overrightarrow{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y)$$



BiDirectional RNNs with LSTM



Attention in RNN: Why?

- Even LSTMs struggle to effectively encode(and decode) long sequences

- NN has to find connections between long input and output sentences (dozens of words) - machine translation, sentence similarity etc

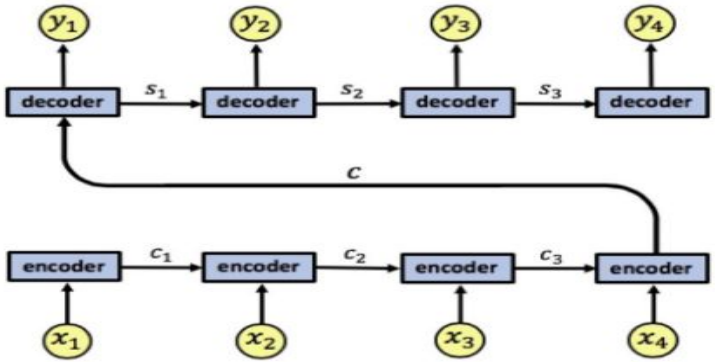
- **Attention** (in RNN) is a mechanism allowing it to focus on certain parts of the input sequence when predicting a certain part of the output sequence

- **Combination of attention mechanisms** enabled improved performance in many tasks making it an integral part of modern RNN networks

Attention in RNN: Machine Translation Issue

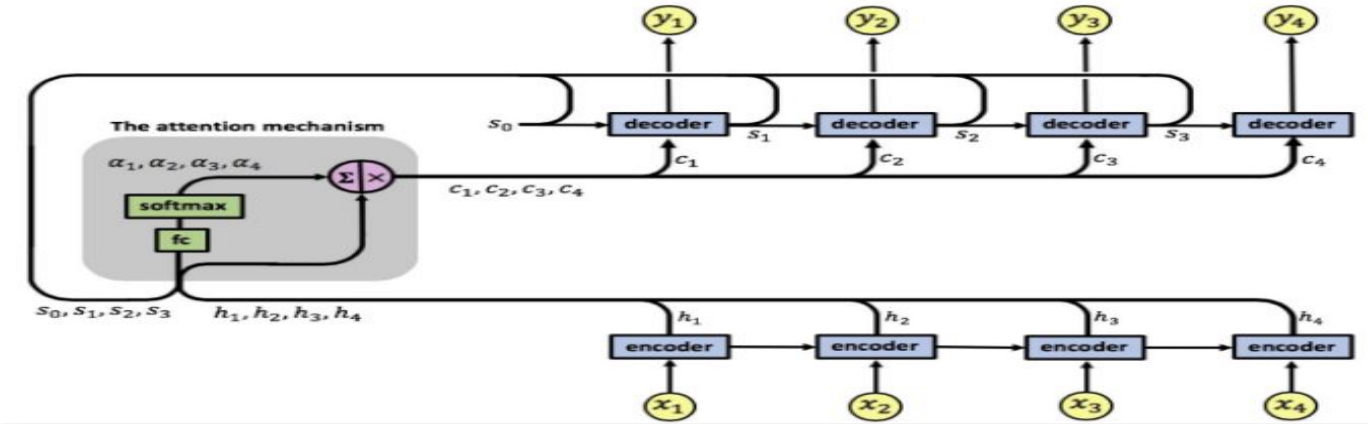
- Encoder needs to represent the entire input sequence as a single vector, which can cause information loss

- Decoder needs to decipher the passed information from this single vector, a complex task in itself.



A potential issue with this encoder–decoder approach is that a neural network needs to be able to compress all the necessary information of a source sentence into a fixed-length vector. This may make it difficult for the neural network to cope with long sentences, especially those that are longer than the sentences in the training corpus.

Attention mechanism:



- Attention mechanism is located between encoder and decoder

- Its input is composed of the encoder's output vectors h1, h2, h3, h4 and the decoders' states s0, s1, s2, s3, the attention's output is a sequence of vectors called context vectors(c1, c2, c3, c4)

Attention: Context vector

- The context vectors enable the decoder to focus on certain parts of the input when predicting its output.
- Each context vector is a weighted sum of the encoder's output vectors h1, h2, h3, h4
- The vectors h1, h2, h3, h4 are scaled by weights αij capturing the degree of relevance of input xj to output at time i, yi.

Context vector

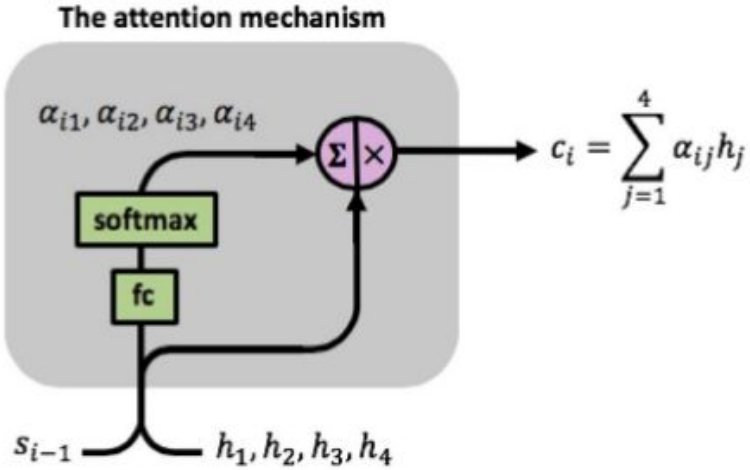
$$c_i = \sum_{j=1}^{4} \alpha_{ij} h_j$$

Attention weights

- Attention weights are learned using an additional FC shallow network
- FC NN receives the concatenation of vectors [si-1,hi] as input at time step i and has a single fully-connected layer

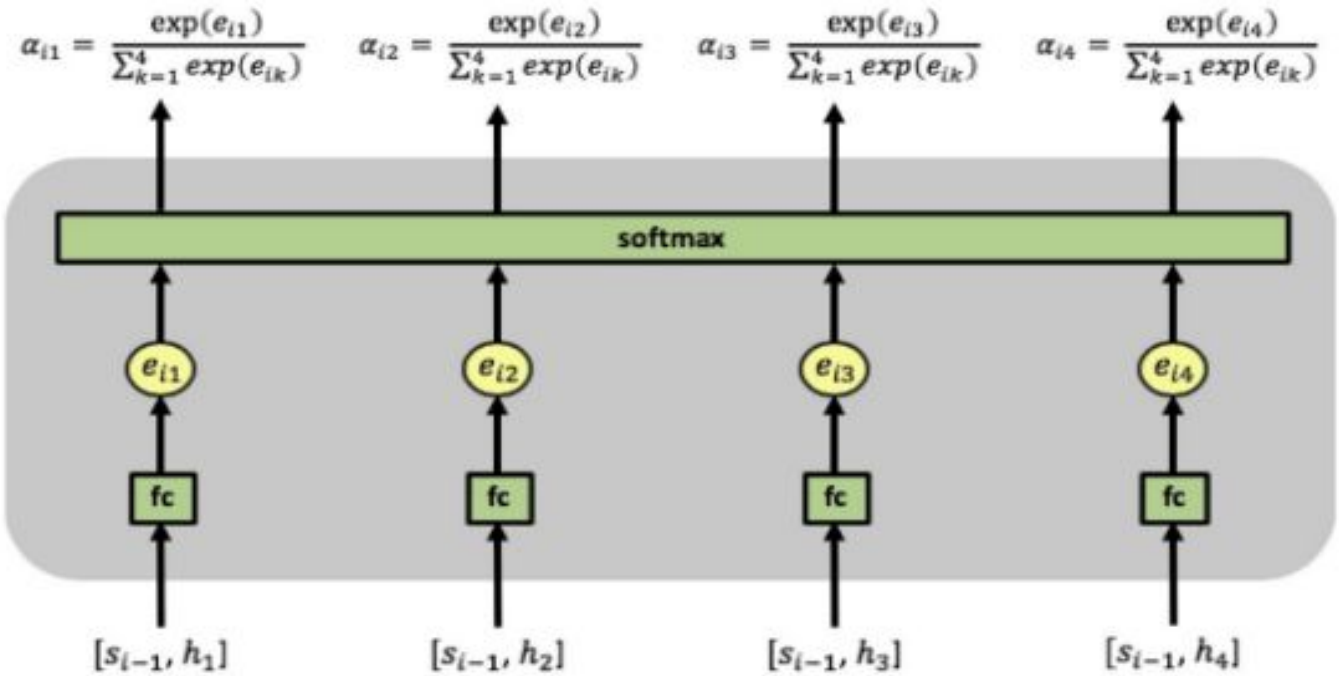
$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{4} exp(e_{ik})}$$

$$where \qquad e_{ij} = \mathrm{fc}(s_{i-1}, h_j)$$

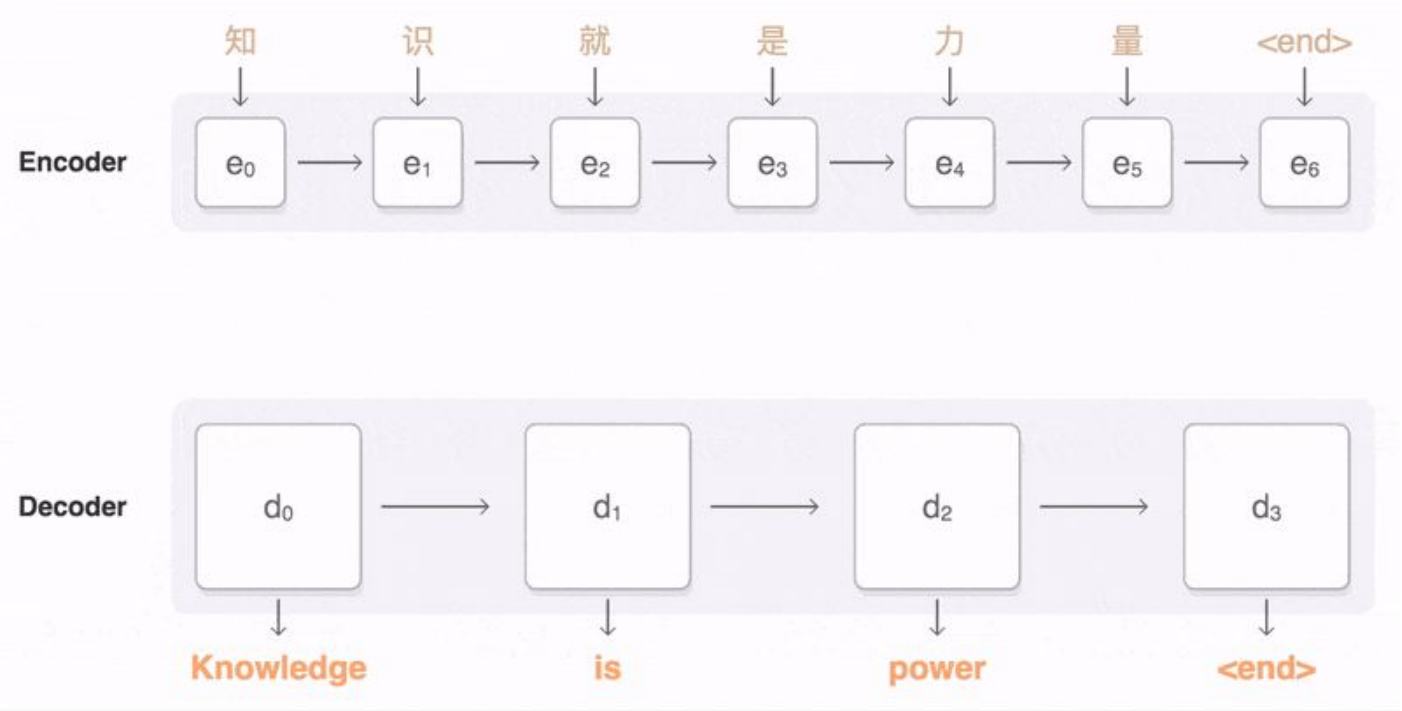


** Attention weights computation can be performed with more complex mechanisms*

Attention Mechanism: All pieces together



Neural Translation with Attention



Attention Mechanism: General Scheme

Transformers: Intro

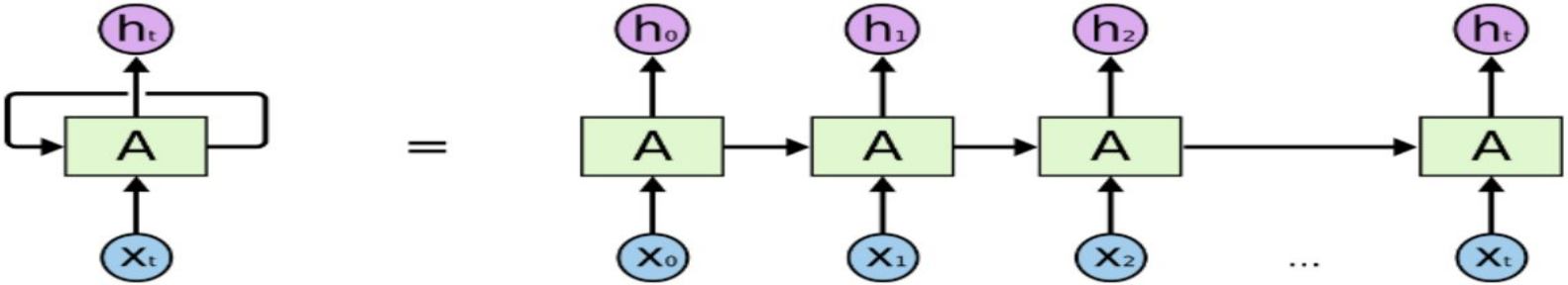
Agenda

- RNNs Recap
- Transformers: General Intuition
- Transformer: High-level Architecture
- Attention: Intuition and How it works
- Positional Encoding
- Encoder and Decoder Architecture
- Transformer Usage Modes

RNNs Recap

- RNN is designated to tackle sequential data
  - Predict next word in a sentence, next vowel in speech, next frame in video…

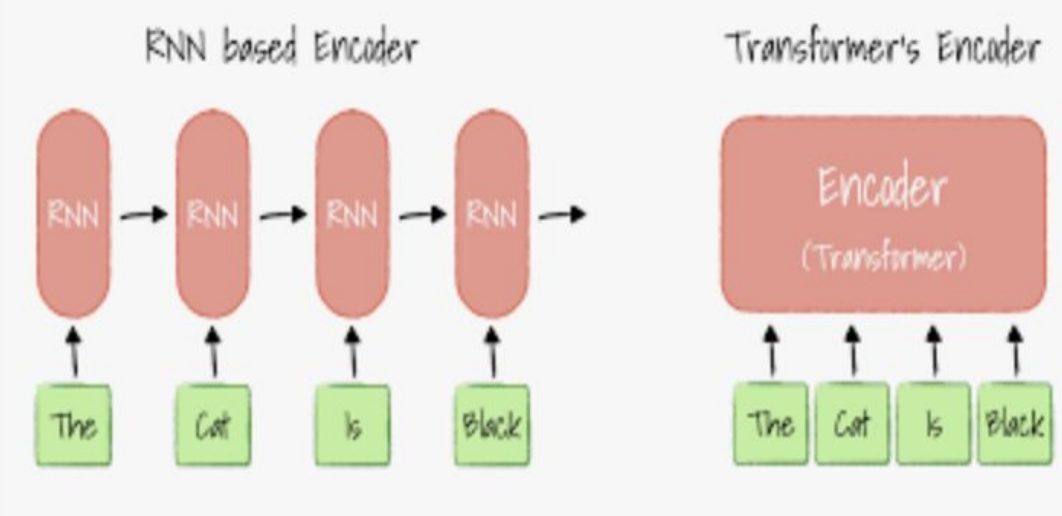
- RNNs are called **recurrent** because they*:*
  - Perform the same task for every sequence element, with the output being depended on the previous(hidden) outputs
  - Have a “memory” capturing information about what has been calculated so far

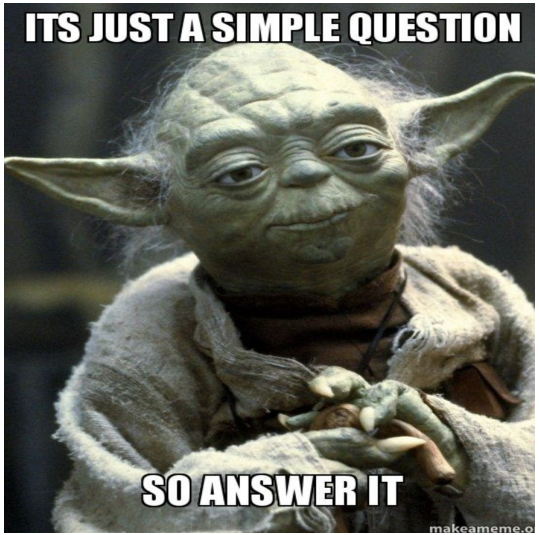


Transformers Revolution: Just One Simple Question

Why don’t we feed the entire input sequence?

No dependencies between hidden states! That might be cool!





But there is a problem here...

Instead of a sequence of elements, we have a set now (**order is irrelevant - !!!**)

"Hello I love you" ==

"Hello", "I", "love", "you"

"love", "Hello", "I", "you"

"I", "love", "Hello", "you"

.....

But the texts have an order!!
The Transformer needs to know it to "understand" the text meaning

Solution: Positional Encodings!!

- **Positional encoding**: set of small values, added to the word embedding vector before the first Transformer layer
- **Positional encoding**: The same word representation is dependent on its position in the input sentence


LET'S GET OUR HANDS DIRTY

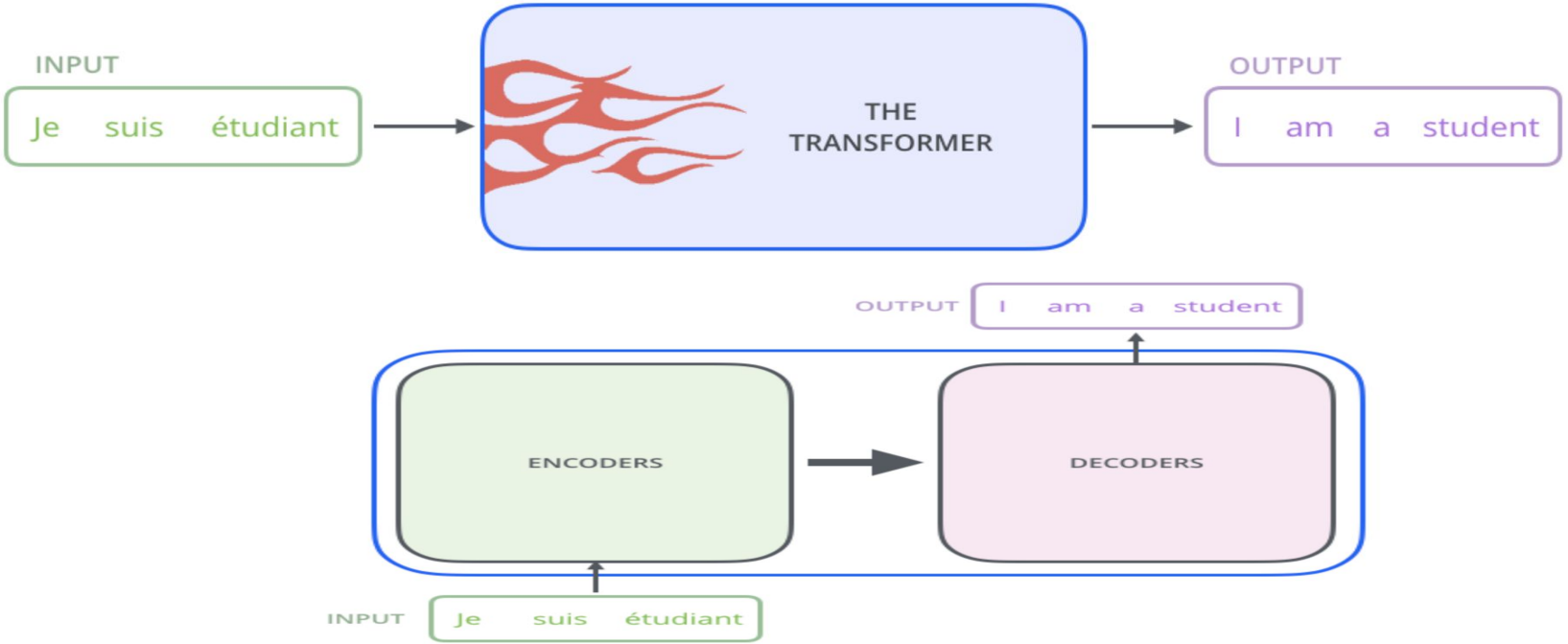
Transformer: A New Emperor of the NLP World

- NLP models until 2018 used different flavours of RNN for most types of tasks
- But then… came “**Attention is all you need”** and “**BERT: …**” papers were published and took the NLP world by storm!
- They introduced new network architecture called **Transformer** relying on attention mechanisms
- Since then **ALMOST ALL** NLP models are based on Transformers/BERT (have BERT/Former inside the name RoBERT, AlBERT, LinFormer, Reformer, PerFormer etc)

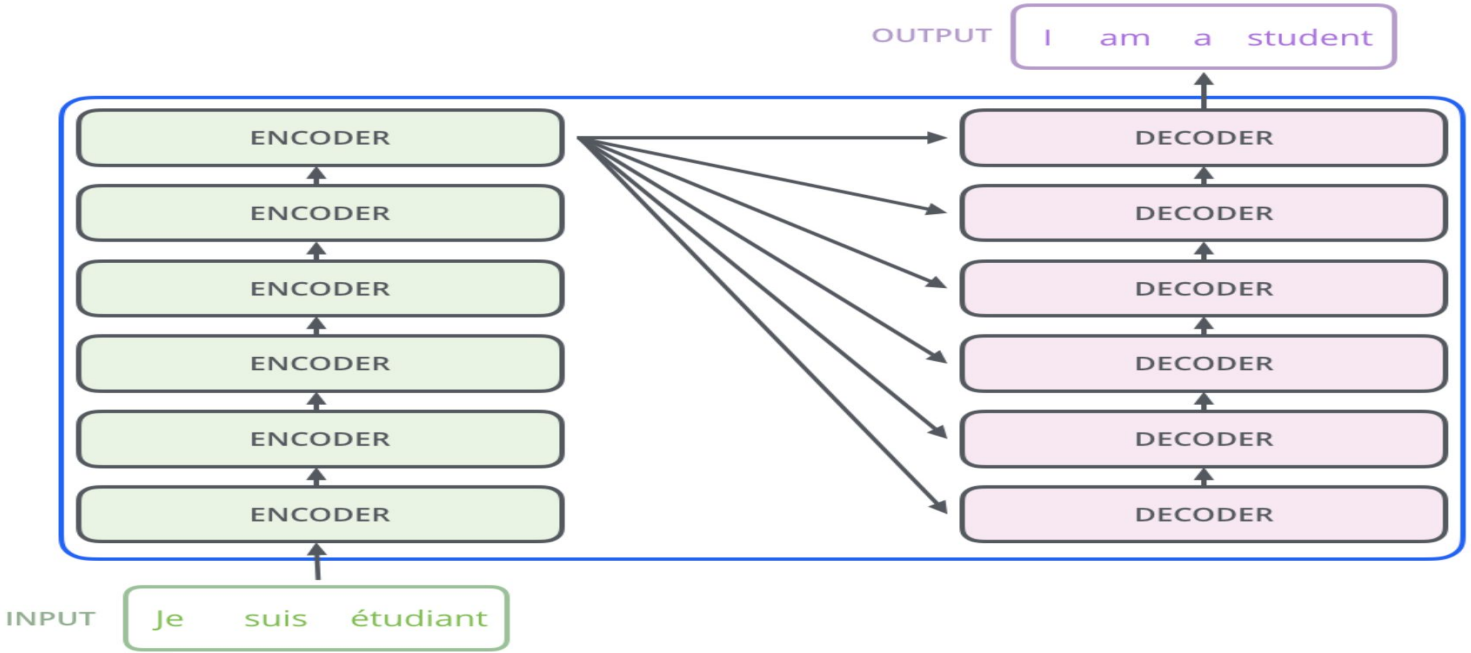
Transformers: High-Level Look

Sentence in Language A

Sentence in Language B

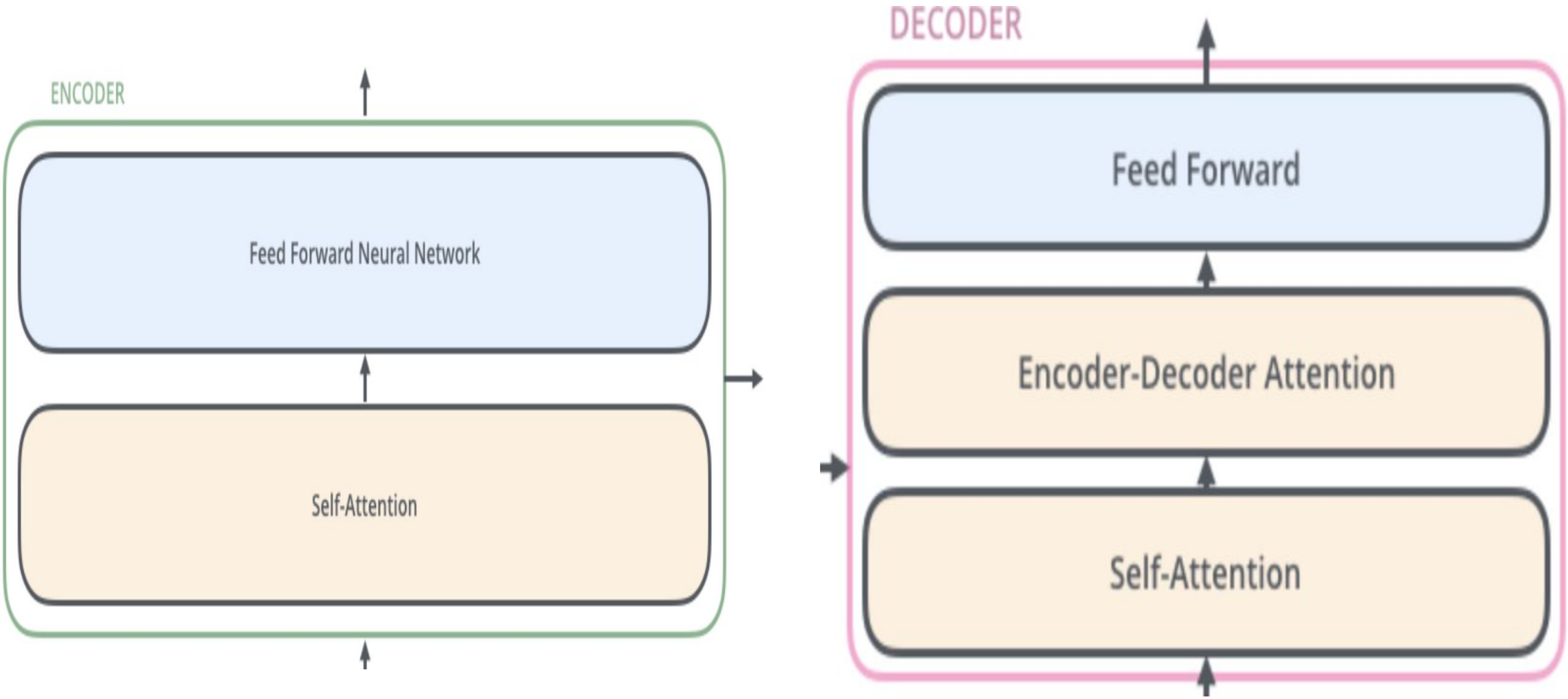


Encoder-Decoder: High-Level Look

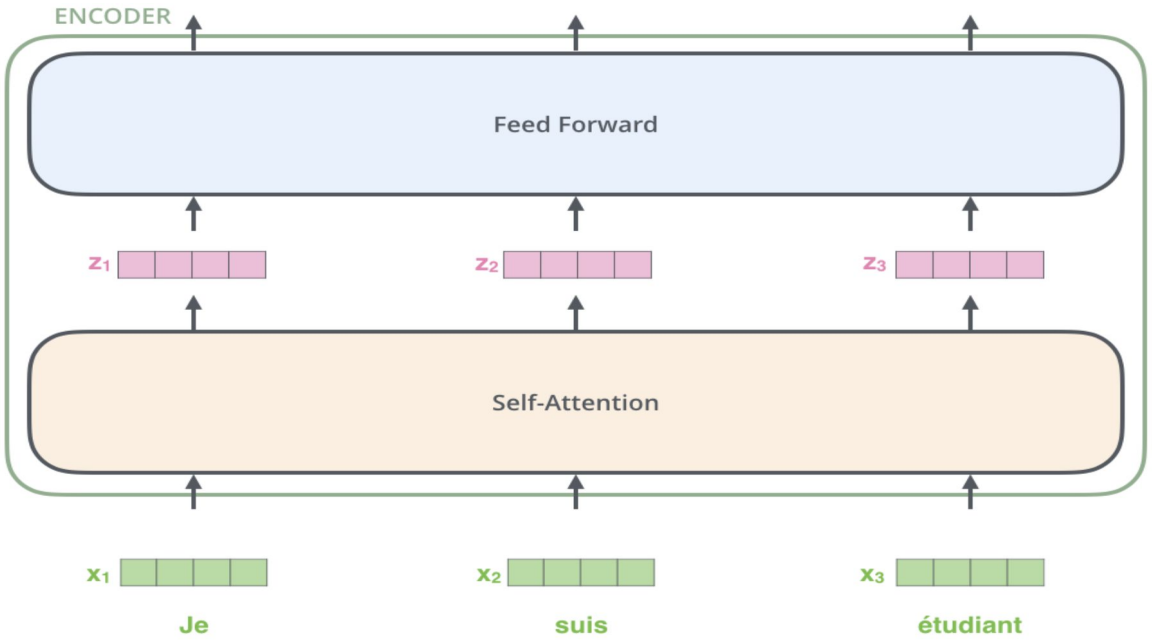


- All encoders have the same architecture
- All decoders have the same architecture

Encoder & Decoder: Main Blocks



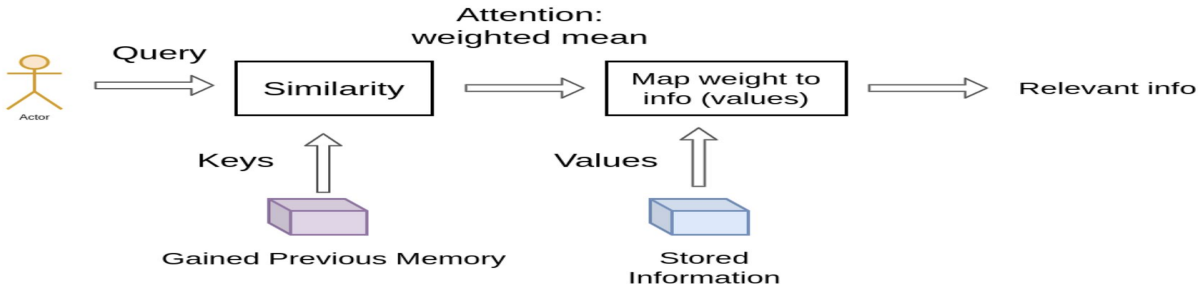
How Encoder Works: Self-Attention + Standard Layers



Wait, but what is this SELF-ATTENTION?

Attention Mechanism: An origin

- Key-value-query concepts originate from information retrieval systems
- **Video search (query)**: content/feature-based lookup
  - Search engine maps the query against a set of keys (video title, description, tags etc.) associated with different stored videos.
  - Algorithm present the best-matched videos (values)
  - **Attention** is the choice of the video with a maximum relevance score



Attention Mechanism: Transformer

But what are the “query”, “key”, and “value” vectors now?

Abstractions useful for computing/thinking about ATTENTION

But what does it really mean?

Transformer Attention: An Essence

Instead of choosing **where** to look according to the position in a sequence,

we now choose the **content that we wanna concentrate at**

- Split the data into key-value pairs (query preserves its original meaning)
- Keys define the attention weights to look at the data
- Values is the information that we will actually get
- Determine similarity metrics (cosine similarity = normalized inner product)

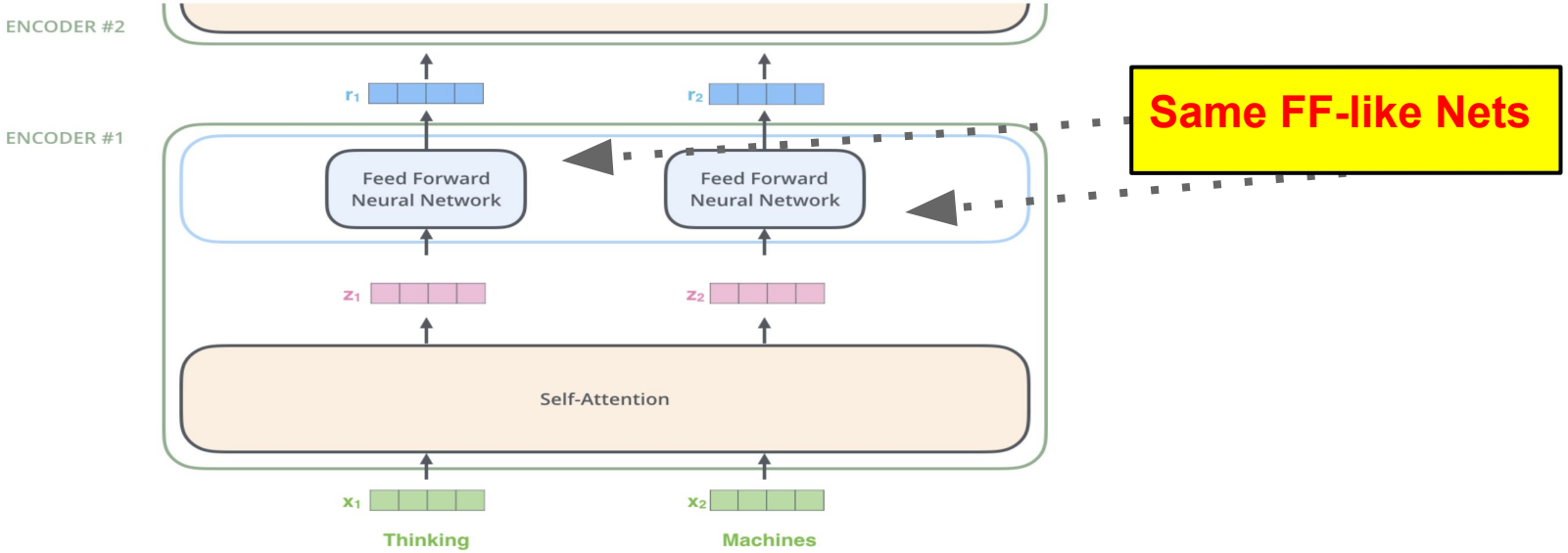
Self-Attention: Let's be More Specific

> "Self-attention, sometimes called intra-attention, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence." ~ Ashish Vaswani et al. [2] from Google Brain.

- Detects correlations between input words indicating the syntactic and contextual sentence structure
- These words share subject-verb-object relationship - that's what self-attention captures

	Hello	I	love	you
Hello	0.8	0.1	0.05	0.05
I	0.1	0.6	0.2	0.1
love	0.05	0.2	0.65	0.1
you	0.2	0.1	0.1	0.6

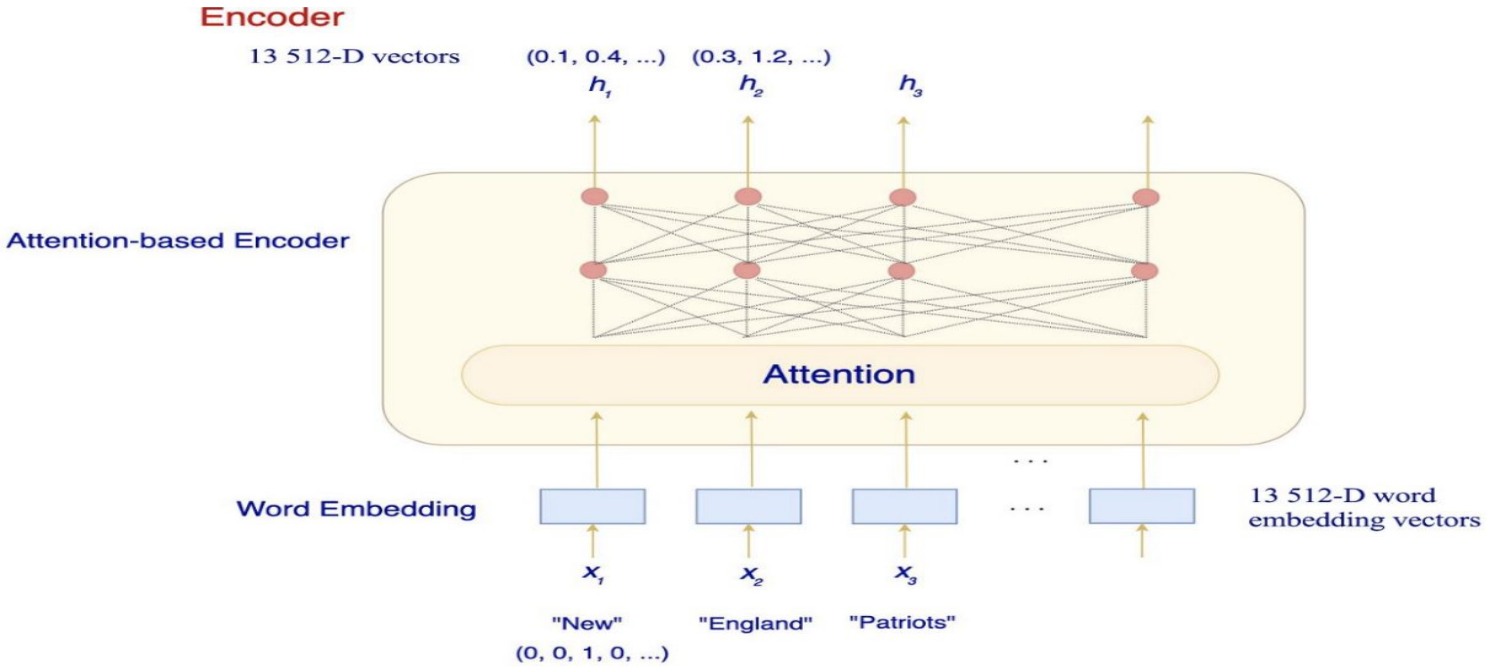
Encoder: Self-Attention + Standard Layers



Self-Attention: word flows through its **own path in the encoder**; dependencies between these paths

FF layer: various paths can be executed **in parallel**

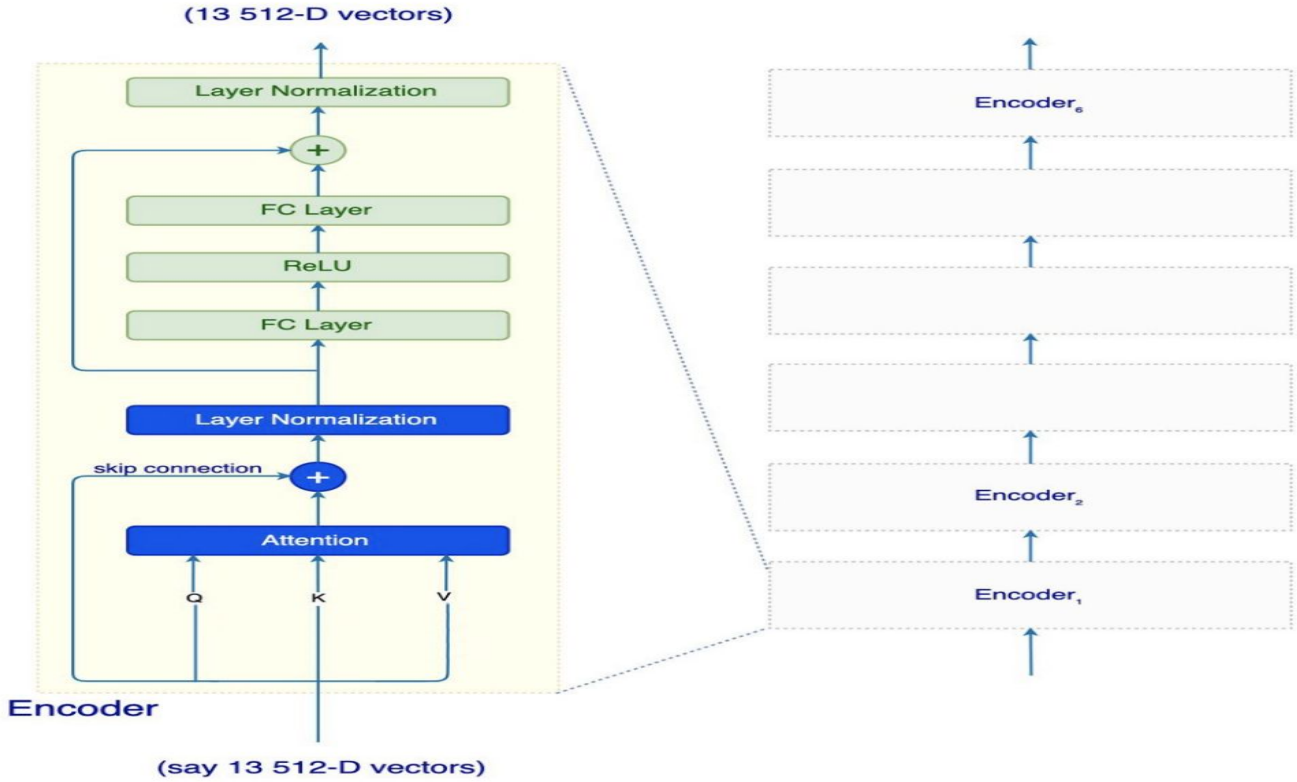
Encoder: Contextualized Token Embeddings



Input: Raw Text

Output: Contextualized tokens embeddings

Encoder: More Detailed Layer Scheme



Self-Attention: Reminder

Dark colors represent higher attention (Image by Author)

Self-Attention in Detail

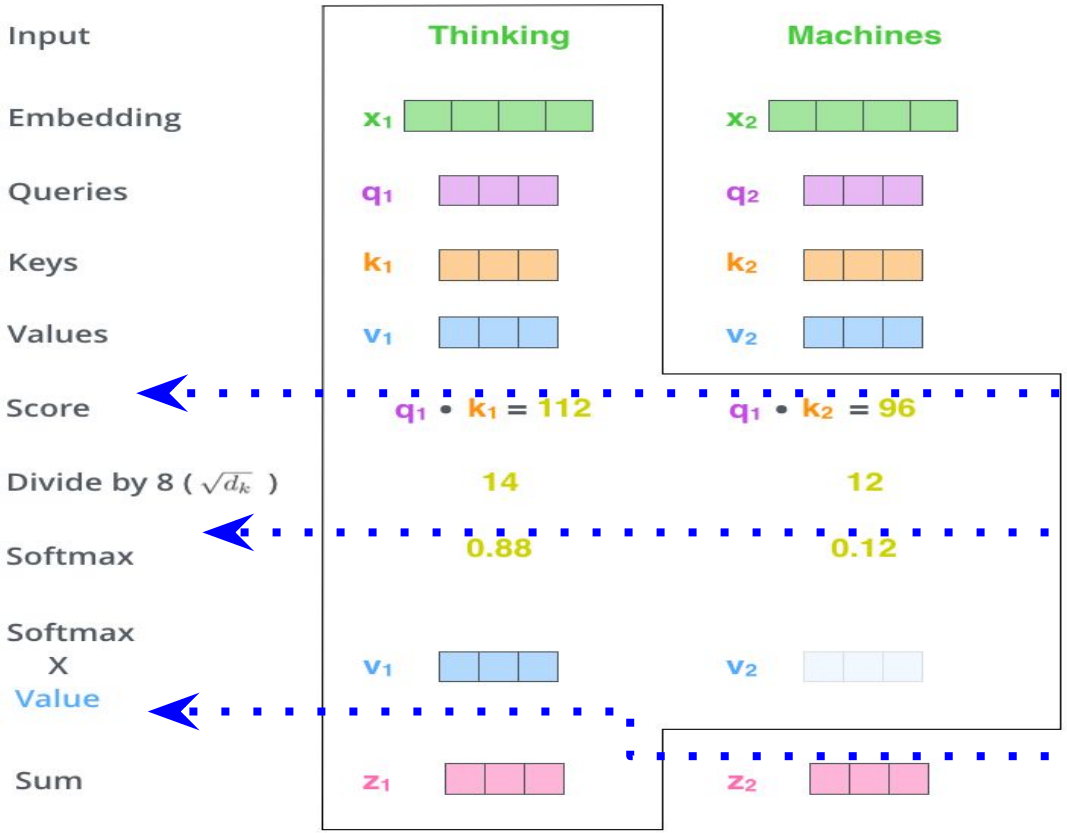
- Construct 3 vectors from each of the encoder's input vectors

- For each word, Query vector Q, a Key vector, K and a Value vector, V, are created

- K, Q and V are created by multiplying the embedding by 3 matrices **learned** during the training process.

- **Important:** these attention vectors can be of any dimension (depends on the dimension of the matrices multiplying embeddings)

Compute self-attention for each word

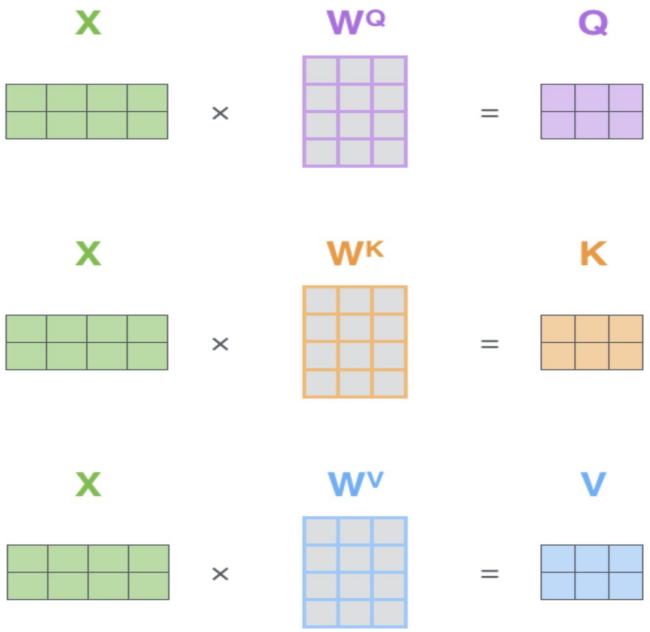


Score each word of the sentence against this word: determines how much focus on other parts of sentence while encoding this word

Normalize

Multiply each value vector by the softmax score: keep intact the values of the word(s) to focus on, and drown-out irrelevant words (multiplied by tiny numbers)

Self-Attention: General Scheme

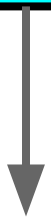


$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) V = Z$$

Every row in the X matrix corresponds to a word in the input sentence

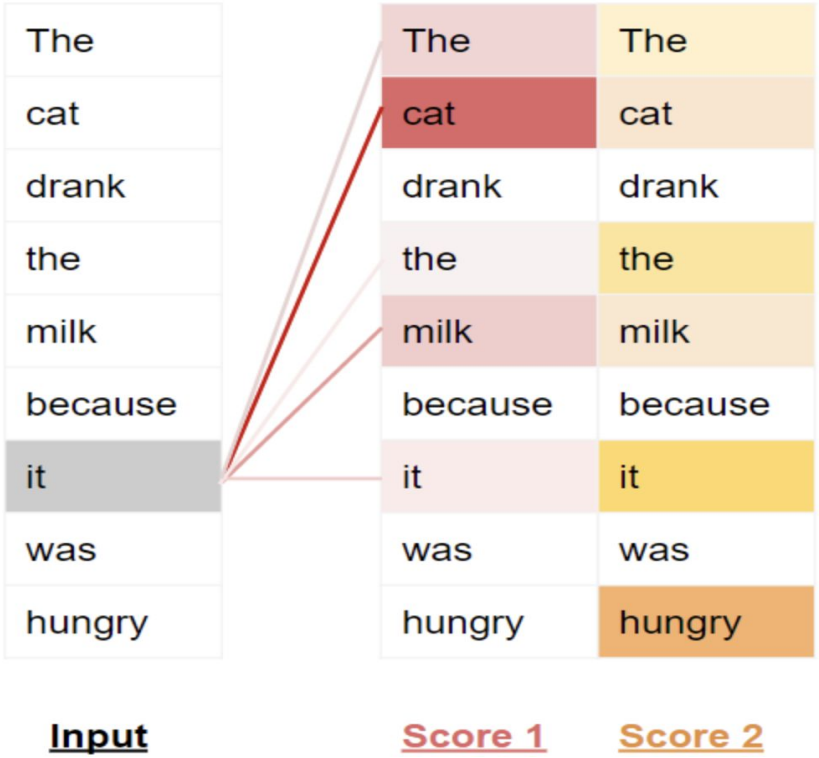
Core Idea: Multiple Attention Scores

To enable handling more nuances about intent/semantics of the sentence, Transformers include multiple attention scores for each word



Multi-Head Attention

Multiple Attention Scores: Why we need them?



- 'it' word: the 1st score highlights 'cat', while the 2nd highlights 'hungry'.
- Decoding 'it': (e.g. translation): incorporates some aspect of both 'cat' and 'hungry' into the translated word.

More powerful contextualized Embeddings!!!

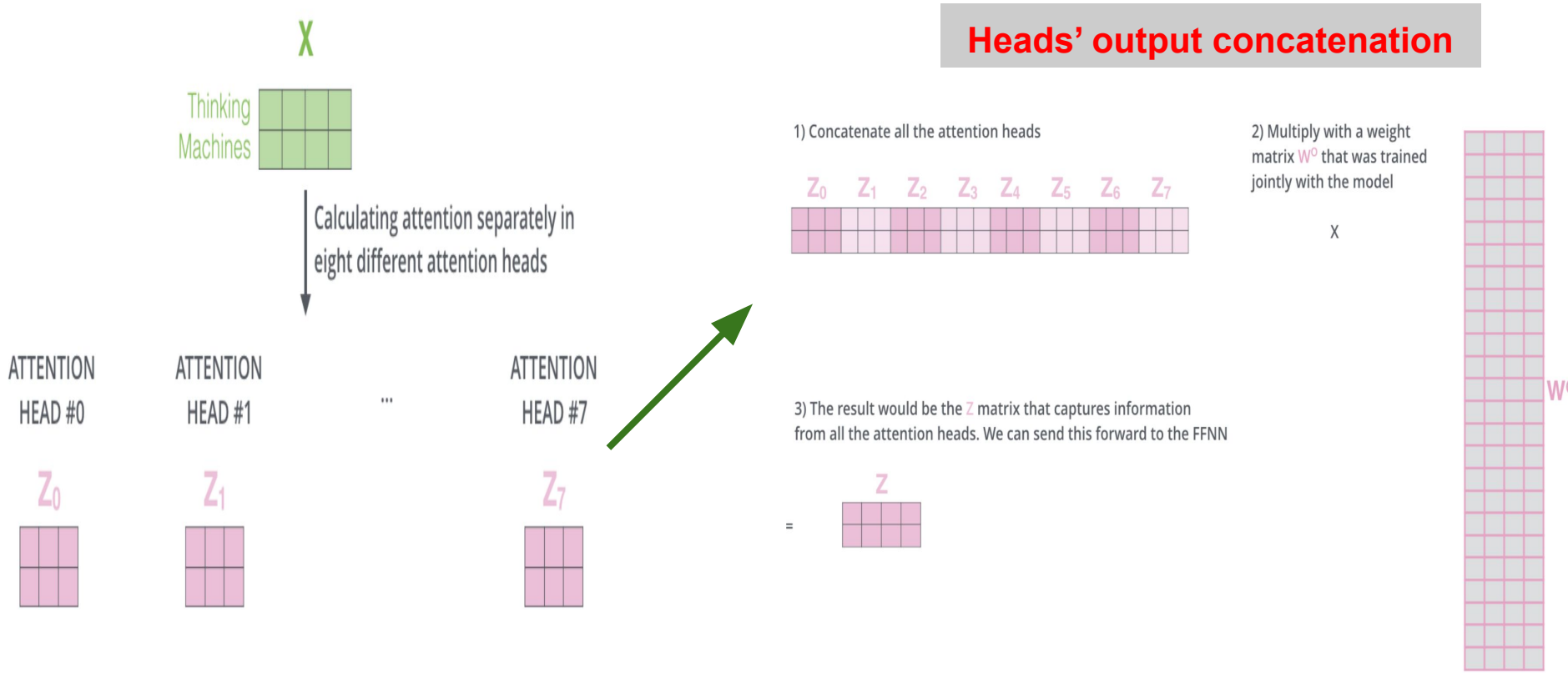
Multi-Head Attention

- Each head captures different contextual information by correlating words in a unique manner.

- Expands the model's ability to focus on different positions
  - In translating a sentence like "The **animal** didn't cross the street because it was too **tired**", we would want to know which word "it" refers to.

- Gives the attention layer multiple "representation subspaces".
  - Multiple sets of Query/Key/Value matrices (Transformer uses 8 heads).
  - After training, each set is used to project the input embeddings (or vectors from lower encoders) into a different representation subspaces

Multi-Head Architecture: All Pieces Together



Multi-Head Self-Attention: Detailed Scheme

1) This is our input sentence*

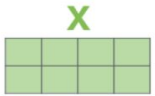
2) We embed each word*

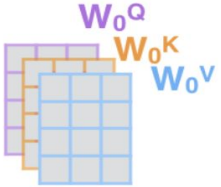
3) Split into 8 heads. We multiply X or R with weight matrices

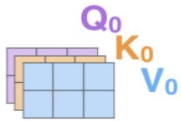
4) Calculate attention using the resulting Q/K/V matrices

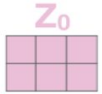
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

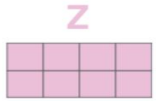
Thinking Machines

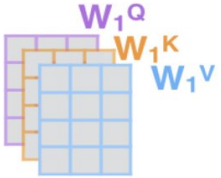


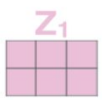




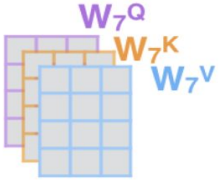


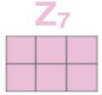






* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one





But what about the words' order?

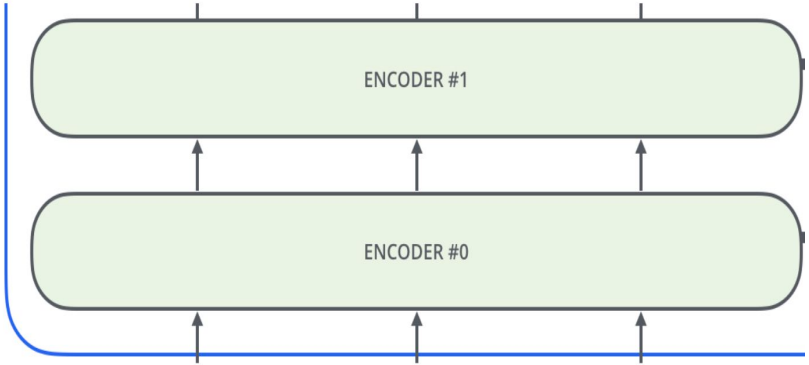
- Self-attention mechanism are agnostic to the words' order which is unacceptable for language model

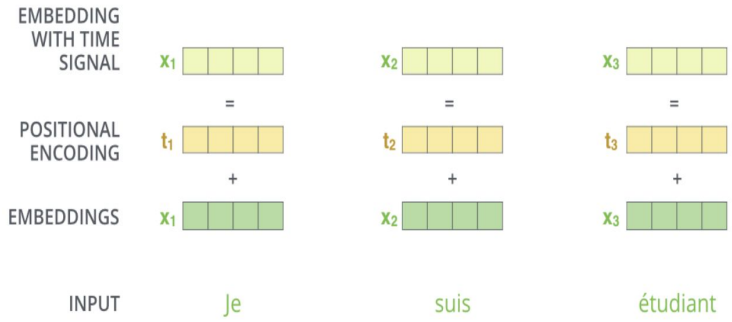
- **Question:** How we take words order into account?

- **Answer: Positional Encodings** - encode the absolute or relative positional information into the word embedding.

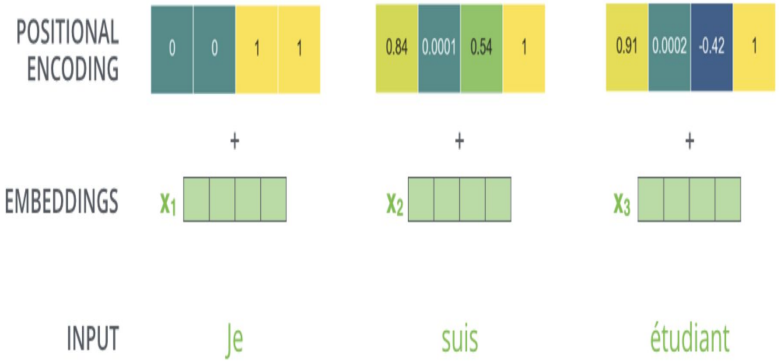
- Adds a position encoding with the same dimension to the word embedding

Positional Encoding: How it works?





To give the model a sense of the order of the words, we add positional encoding vectors -- the values of wh



A real example of positional encoding with a toy embedding size of 4

Positional Encoding: Properties

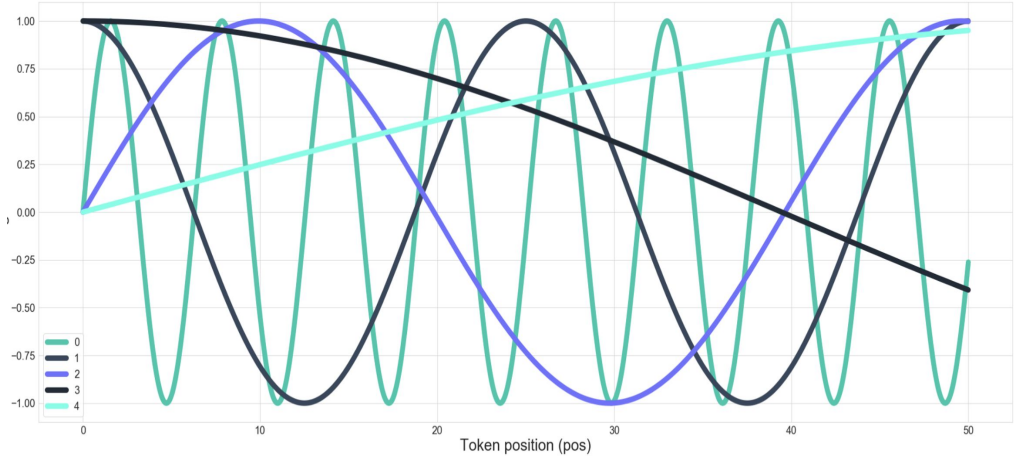
- Unique encoding for each time-step (word's position in a sentence)
- Distance between any two time-steps should be consistent across sentences with different lengths
- Generalizes to longer sentences without any efforts and bounded
- Deterministic

Positional Encoding: How it works?

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{\frac{2i}{d_{model}}}})$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{\frac{2i}{d_{model}}}})$$

Alternating positional encoding values. Using word position **pos**, embedding dimension **i**, and the number of embedding dimensions **d_model**.



Sinusoidal functions of the first five embedding indices using a total embedding dimensionality of 20. The embedding index positions are shown in the legend.

Periodic function allows us to embed relative word position information into the word embedding.

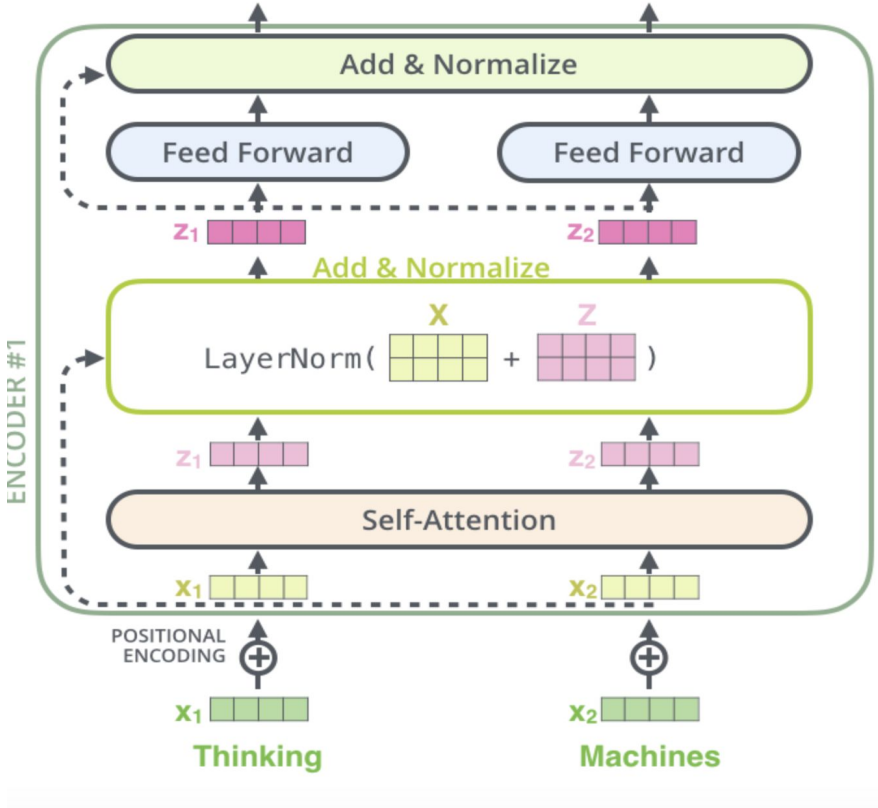
Encoder: Sum up

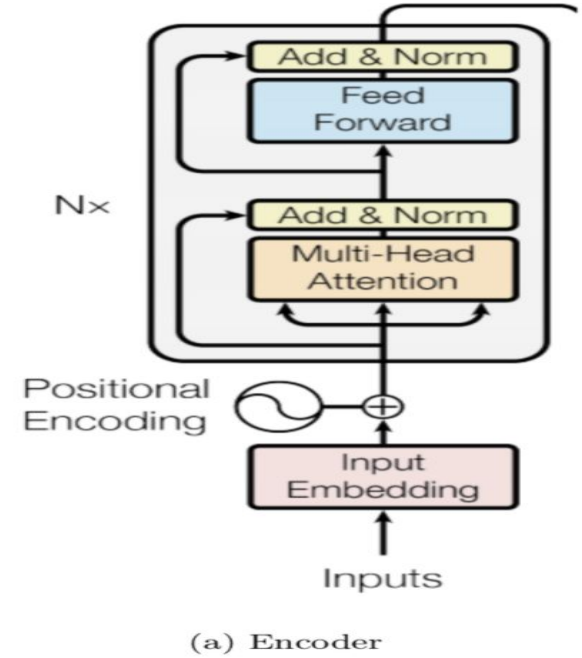
1. Word embeddings of the input sentence are computed simultaneously.

2. Positional encodings are then applied to each embedding resulting in word vectors that also include positional information.

3. The word vectors are passed to the first encoder block.

Full Encoder Architecture

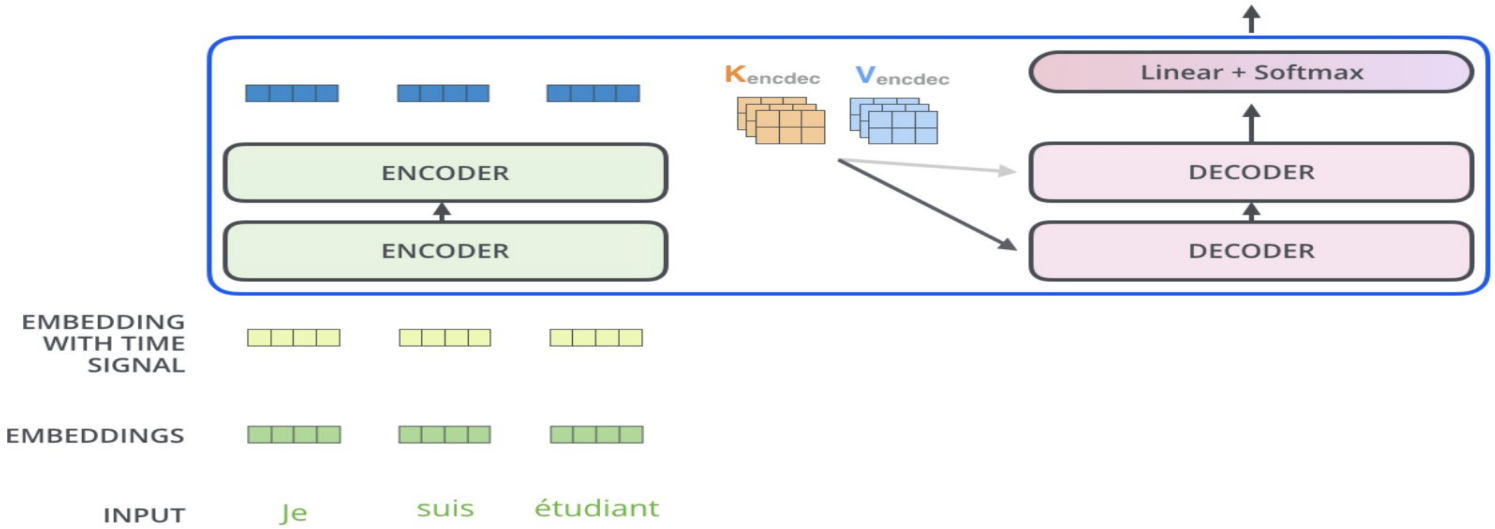




(a) Encoder

Original paper encoder architecture

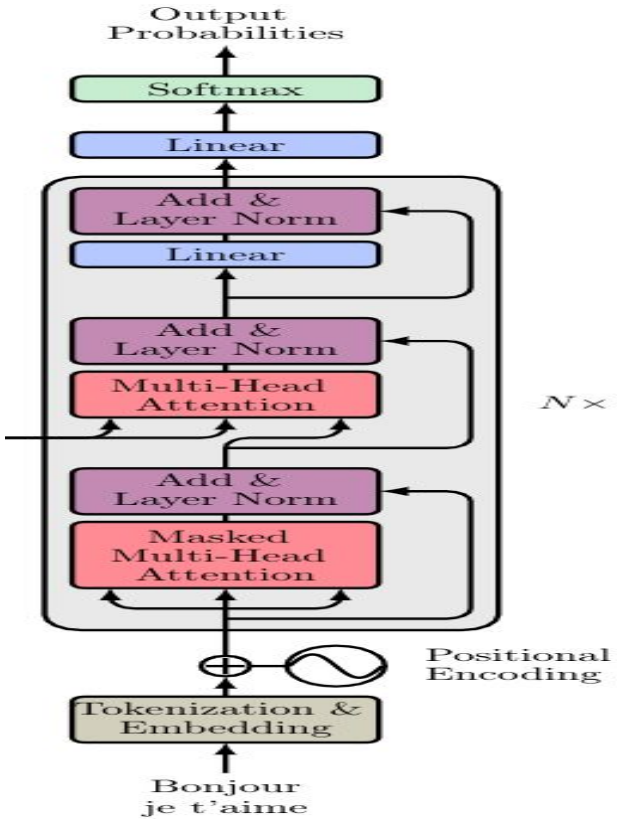
Decoder: Just a couple important notes



- The output of the top encoder is transformed into a set of attention vectors K and V

- They used by each decoder in its “encoder-decoder attention” layer helping the decoder focus on appropriate places in the input sequence

Decoder Architecture: Details



1. **Masked multi-head self-attention layer**
2. Normalization layer followed by a residual connection
3. **Multi-head attention layer (Encoder-Decoder attention)**
4. ….
5. The last layer predicts the next token in the output sentence(probabilities)

Most Important parts

Decoder-Decoder Attention Layer

- Combines input and output sentence

- Encoder's output constitutes input sentence embedding(database) - **used to produce Key and Value matrices**

- Masked Multi-head attention output contains so far generated sentence: represented as **Query matrix in attention layer** : "search" in the database.

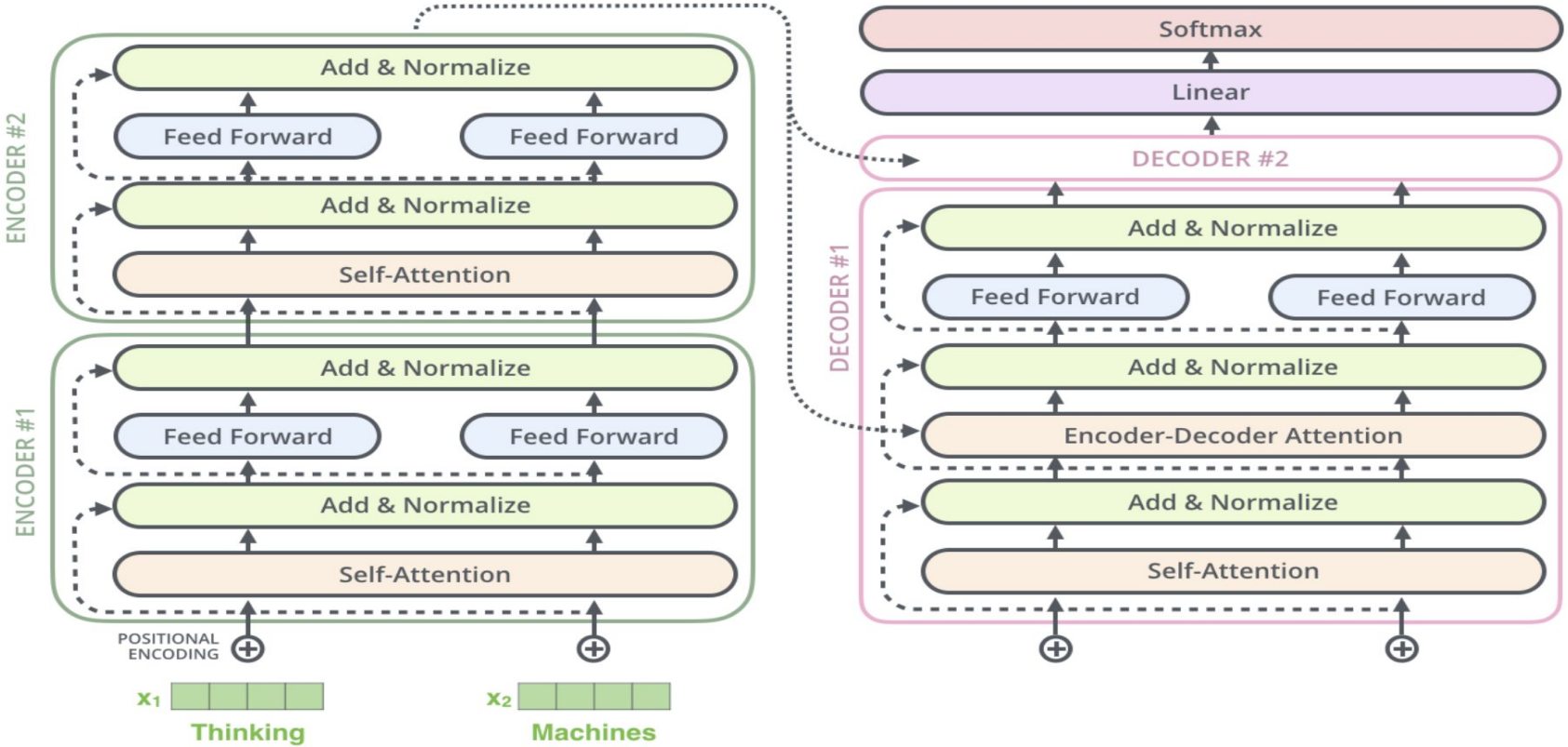
Decoder: Masked Self-Attention

- Predict one word (token) after another - sequential processing unavoidable(e.g. translation)
- **We don’t know the whole sentence because it hasn’t been produced yet**
- Decoder self-attention layer **is only allowed** to use **earlier positions** in the output sequence
- Mask future positions (set to -inf) in the self-attention calculation

$$\text{MaskedAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T + \mathbf{M}}{\sqrt{d_k}}\right)\mathbf{V}$$

M contains -infs and zeros ONLY

Full Transformer Architecture



Transformer Success Reasons:

A. Distributed and independent representations at each block
   a. Each transformer block has 8 contextualized representations capturing different input features

B. Meaning heavily depends on the context: self-attention
   a. Relationships between word representation are expressed by attention weights
   b. No locality notion model makes global associations

C. Multiple encoder and decoder blocks
   a. Builds more abstract representations with more layers.
   b. Similar to stacking recurrent/convolution blocks we can stack multiple transformer blocks (~receptive field in terms of pairs of distributed representations)

Transformer Usage Modes

- Encoder-only (e.g. sentiment analysis, topic modeling, NER)
- Decoder-only (e.g. language modeling)
- Encoder-decoder (e.g. machine translation, summarisation, question answering)

Topics Not Covered :(

- How Transformer is trained (masking words)
- Transformer adaption to perform a wide range of NLP tasks
- Main issues with Transformers: Quadratic Complexity!!
- Different Transformer architectures, improvements
- Transformers for images, time-series data and other domains

Thank You
For Your Attention