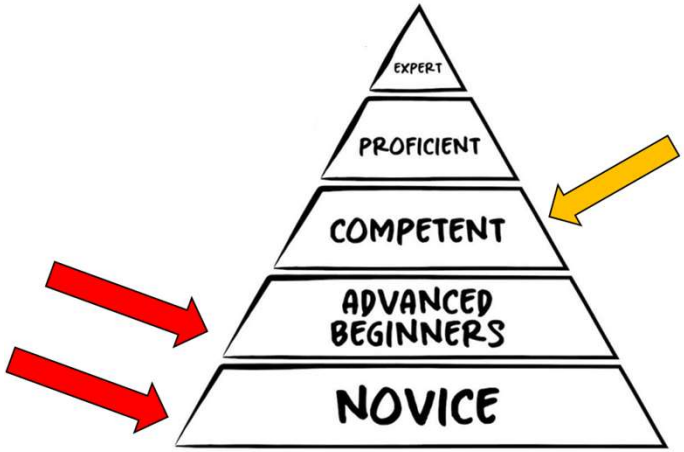
MTech DSML (PES)

Arun Sharma K

WHAT TO EXPECT

Basics of Text processing + How ML models process test + What are the SOTA DL models + What are LLMs and RAGs + Utilizing LLMs and RAGs = Foundation



Day Plan

Need and Applications

Workflow

NLP fundamentals

Hands on

Typical questions

Q&A

Participate

Sessions

1

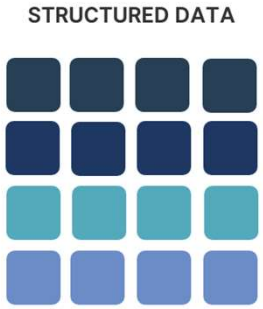
- Unstructured Vs structured
- Need for NLP
- Typical applications
- Aspects of NLP
- Workflow
- Clean-up
- EDA
- Hands-on

2

- Feature engineering
- Frequency based
- Sematic based
- Hands-on

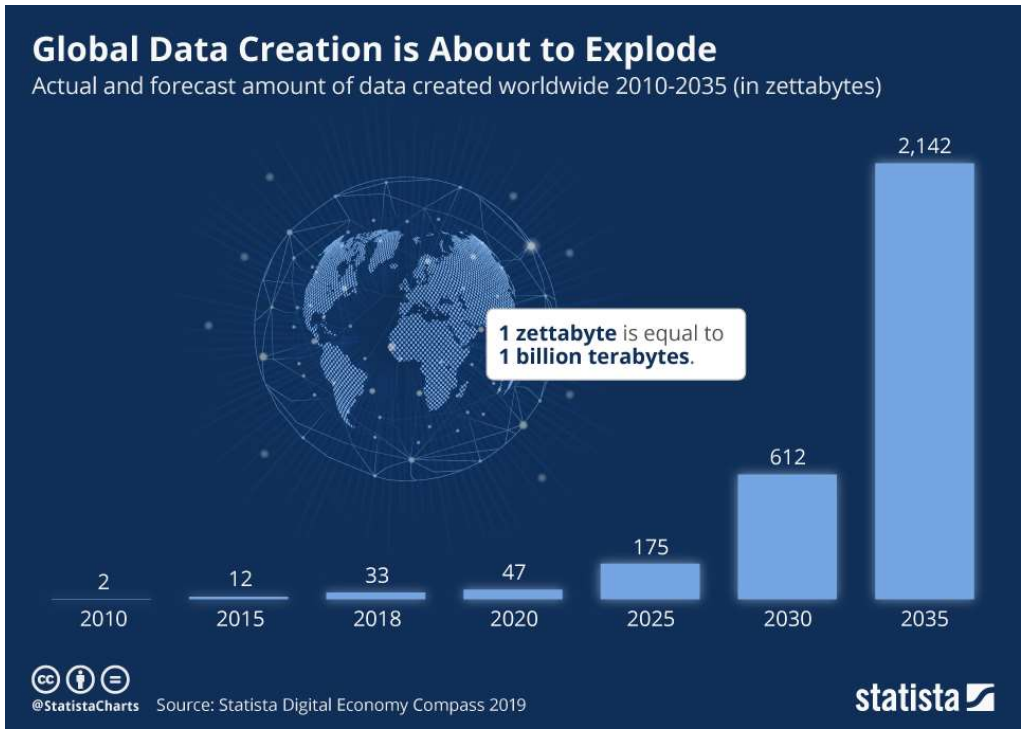
NLP:Session-1

Structured Vs Unstructured



Fits neatly into a database	Lacks a clear structure
Easy to retrieve and analyze	Needs special tools to be useful
Goes into a data warehouse	Requires more complex storage
Enables quick decision-making	Takes longer to analyze
Uses basic tools like spreadsheets, SQL	Requires advanced tools like NLP, ML

Need for NLP



Growing volume of text, streaming audio, video, clickstream, sensor and log data→ ~90% of worldwide data will be unstructured by 2025 [IBM]

Need for NLP

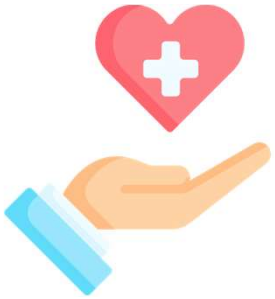
Huge emphasize from Organizations to focus on develop robust strategies for effectively

- Managing
- Storing
- Analyzing

Harness the data for better, data-driven decisions

Value

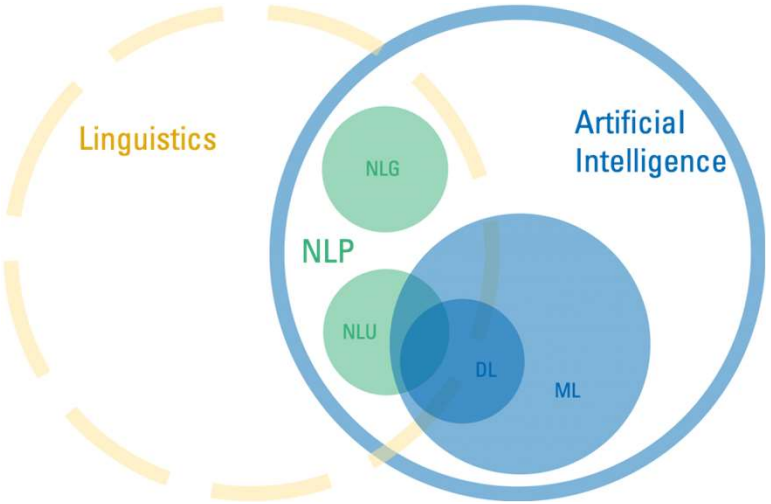
- 25% possible annual savings
- 31% reduction in 30-day readmission for certain patients
- USD 500,000 savings per year

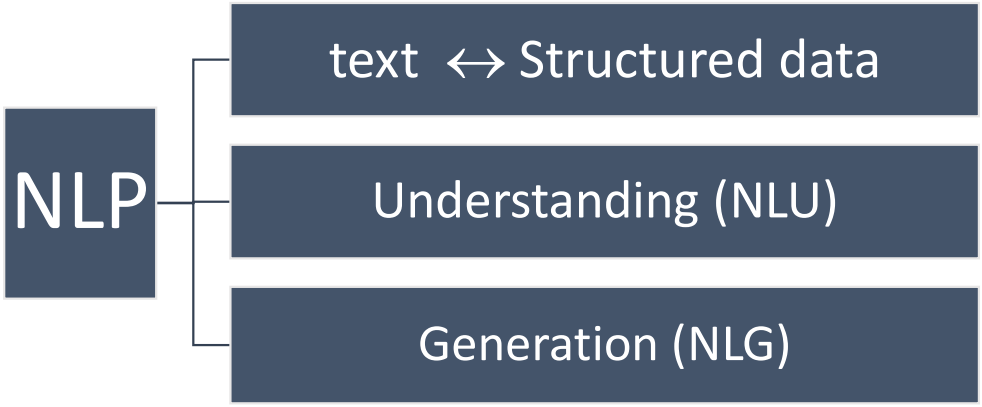


- 30% fewer fraud incidents
- Nearly USD 4 million reduction in expenditure
- 90% quicker time-to-value for big data analytics

https://www.ibm.com/think/insights/managing-unstructured-data#_edn2

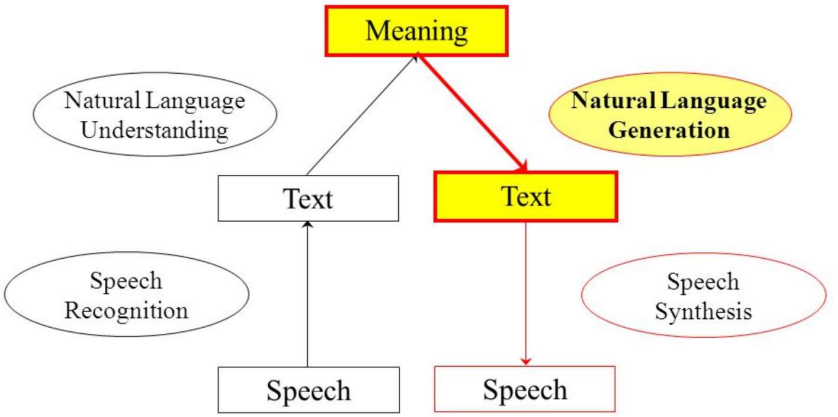
What is NLP


Linguistics
Artificial Intelligence
NLG
NLP
NLU
DL
ML


NLP
text ↔ Structured data
Understanding (NLU)
Generation (NLG)

NLU Vs NLG

- NLG is the inverse of NLU
- NLG maps from meaning to text, while NLU maps from text to meaning
- NLG is easier than NLU because a NLU system cannot control the complexity of the language structure it receives as input while NLG links the complexity of the structure of its output.



NLU
Ambiguity in input
ill-formed input
Under-specification

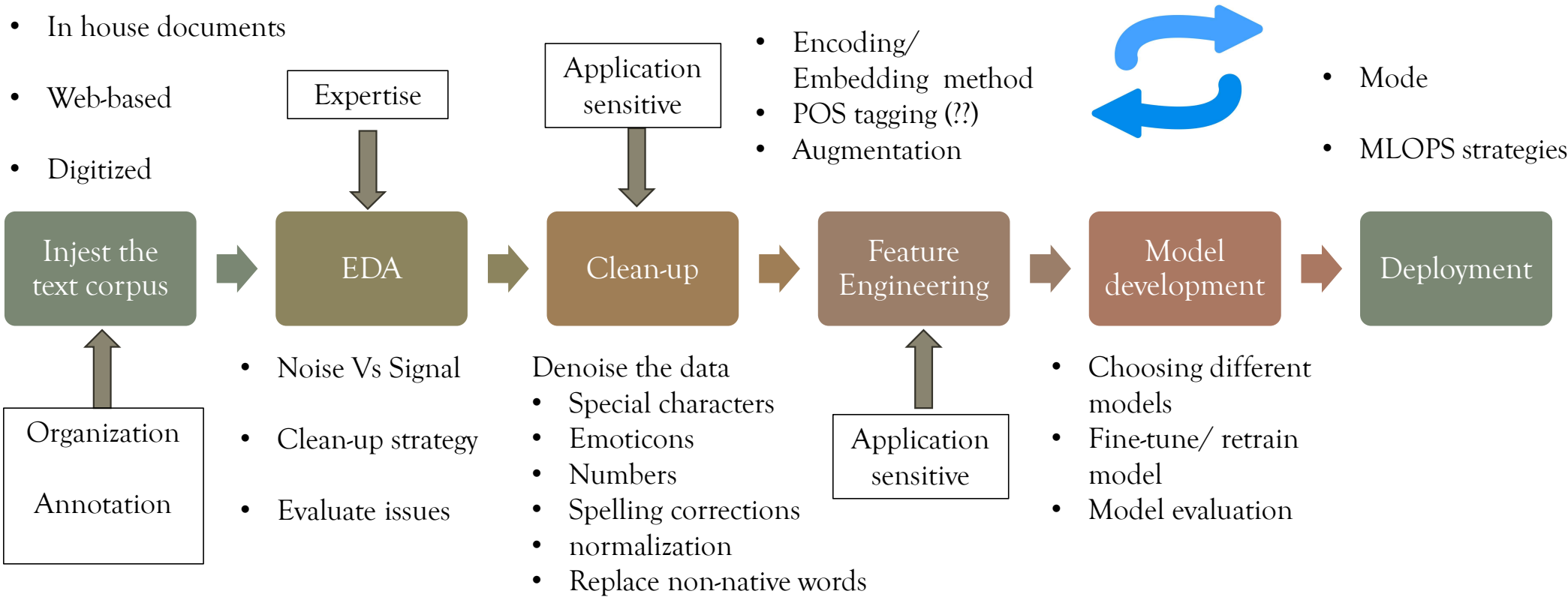
NLG
Relatively Unambiguous
Well-formed
Well-specified

http://www.lacsc.org/papers/PaperA6.pdf

Typical Applications of NLP

- Spell Checking
- Text Classification
- Sentiment Analysis/opinion mining
- Question Answering
- Automatic Summarization
- Text suggestion
- Machine Translation(statistical machine translation)
- Speech Recognition

Workflow


In house documents
Web-based
Digitized
Expertise
Application sensitive
Encoding/ Embedding method
POS tagging (??)
Augmentation
Mode
MLOPS strategies
Injest the text corpus
EDA
Clean-up
Feature Engineering
Model development
Deployment
Organization
Annotation
Noise Vs Signal
Clean-up strategy
Evaluate issues
Denoise the data
Special characters
Emoticons
Numbers
Spelling corrections
normalization
Replace non-native words
Application sensitive
Choosing different models
Fine-tune/ retrain model
Model evaluation

Steps in preprocessing

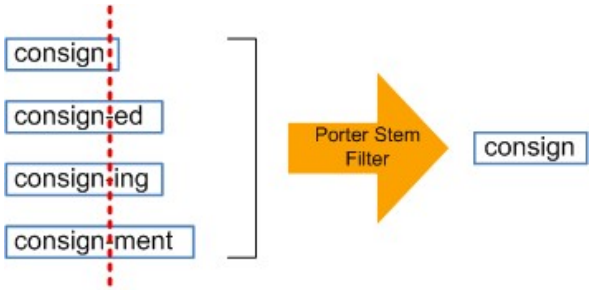
- Removing punctuations
- Tokenization
- Remove stop words
- Stemming
- Lemmatizing
- POS tagging
- Vectorizing Data
  - Bag of words/CountVectorizer
  - N-Grams
  - TF-IDF

Tokenization

- Sentence tokenization→ split collection of sentences to individual sentences
- Word tokenization→ split sentence into words
- Stop words→ words that do not hold much importance in processing compared to keywords
- Nltk stop words
- Adding new stop words

Stemming

- Stemming: heuristically removing the affixes of a word, to get its stem (root) word
- Stem (root) is the part of the word to which you add inflectional (changing/deriving) affixes such as (-ed,-ize, -s,-de,mis)
- Stemming :reducing inflection in words to their root forms such as mapping a group of words to the same stem even if the stem itself is not a valid word in the Language



https://www.datacamp.com/community/tutorials/stemming-lemmatization-python

Lemmatization

- Lemmatization: unlike Stemming, reduces the inflected words properly ensuring that the root word belongs to the language. In Lemmatization root word is called Lemma.

- A lemma is the dictionary form, or citation form of a set of words.

- *runs, running, ran* are all forms of the word *run*, therefore *run* is the lemma of all these words

Hands-on link

NLP S1 and S2 content and notebooks

REGEX

Regex

Sequences of characters that form search patterns

Regex	Description	Example Matches
.	Any single character except newline	a.b → "acb", "a_b"
\d	Any digit (0-9)	\d → "5", "123"
\D	Any non-digit character	\D → "a", "@"
\w	Any word character (a-z, A-Z, 0-9, _)	\w → "a", "1", "_"
\W	Any non-word character	\W → "!", " "
\s	Any whitespace (space, tab, newline)	\s → " ", "\t"
\S	Any non-whitespace character	\S → "a", "9"
^	Start of a string	^Hello → "Hello World"
$	End of a string	end$ → "The end"
*	0 or more repetitions	a* → "", "aaaa"
+	1 or more repetitions	a+ → "a", "aaa"
{n}	Exactly n repetitions	a{3} → "aaa"
()	Grouping	(abc)+ → "abcabc"
[]	Character set	[a-z] → "a", "z"

Examples

Regex to extract email

```
^[a-zA-Z0-9._]+@[a-zA-Z0-9]+.[a-zA-Z]{2,}$
```

- ^ asserts the start of the string.
- [a-zA-Z0-9._]+ matches the username (1+ alphanumeric, dots, underscores).
- @ matches the literal "@" character.
- [a-zA-Z0-9]+ matches the domain name.
- .[a-zA-Z]{2,} matches the top-level domain (e.g., ".com").
- $ asserts the end of the string.

Regex to extract phone#

```
^\d{10}$
```

- ^ asserts the start.
- \d{10} matches exactly 10 digits.
- $ asserts the end.

Feature engineering

Encoding and Embedding types

Unstructured to structured

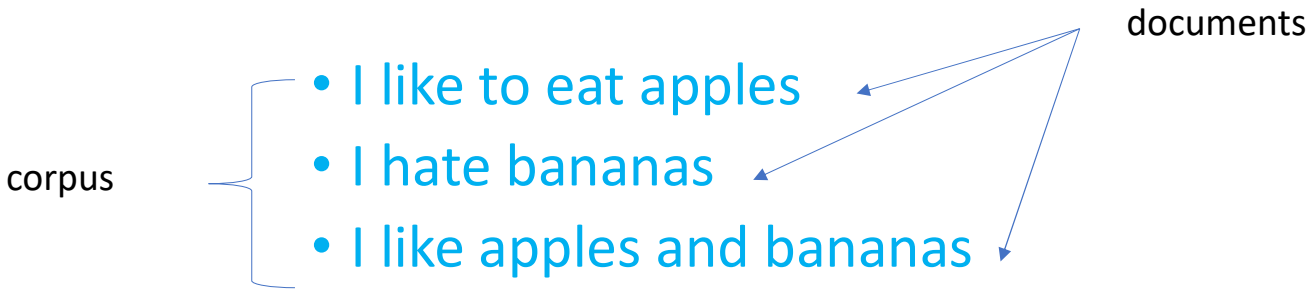
Frequency based

- Count vectorization
- TF-IDF vectorization
- N-gram

Semantic based

- word2Vec
- Glove
- ELMO
- BERT, SBERT
- GPT
- Custom embedding

Bag of words



Vocabulary ={and, apples, bananas, eat, hate, I, like, to}

	And	Apples	bananas	eat	hate	I	like	to	TARGET
1	0	1	0	1	0	1	1	1	1
2	0	0	1	0	1	1	0	0	0
3	1	1	1	0	0	1	1	0	1

I like to eat apples → [0,1,0,1,0,1,1,1]

TF-IDF

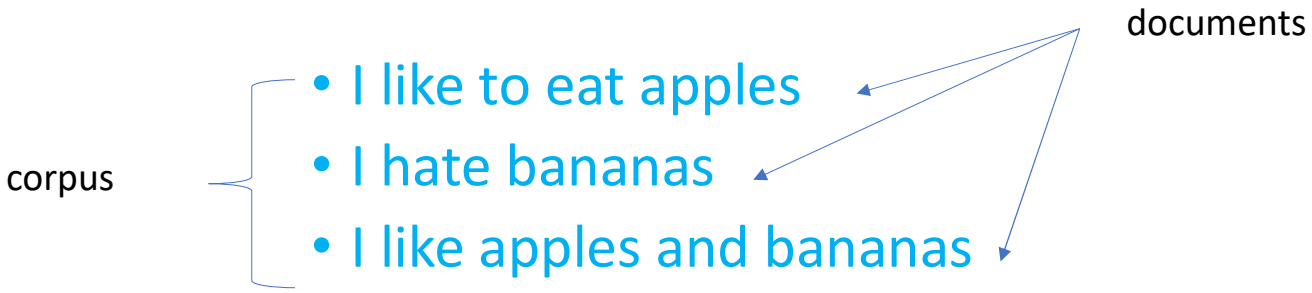
Statistical measure used to evaluate how important a word is to a document in a collection or corpus.

- TF(t,d) = count of ‘t’ in d / number of words in d
- IDF(t) = log(N/(df(t) + 1))
- df(t) = occurrence of t in documents
- N= total number of documents in the corpus
- **TF-IDF(t, d) = tf(t, d) * log(N/(df + 1))**

Term Frequency
This measures the frequency of a word in a document. This highly depends on the length of the document and the generality of word

Inverse Document Frequency
This measures the importance of the document in the whole corpus. DF is the count of occurrences of term t in the document set N

Bag of words



Vocabulary ={and, apples, bananas, eat, hate, I, like, to}

	And	Apples	bananas	eat	hate	I	like	to	TARGET
1	0	1	0	1	0	1	1	1	1
2	0	0	1	0	1	1	0	0	0
3	1	1	1	0	0	1	1	0	1

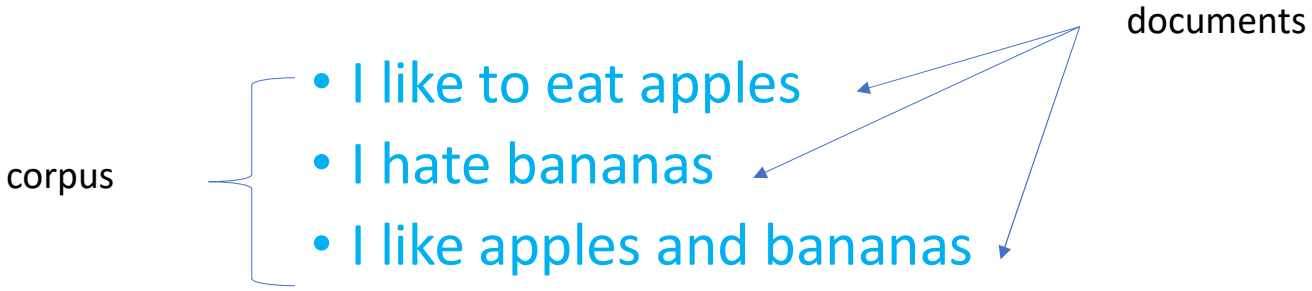
TF (I)= 1/5 =0.2
IDF (I)= ln(3/(3+1))=-0.28
TF-IDF(I)=0.2*-0.28=-0.05754

TF (Apples)= 1/5 =0.2
IDF (Apples)= ln(3/(2+1))=0
TF-IDF(Apple)=0.2*0=0

TF (to)= 1/5 =0.2
IDF (to)= ln(3/(1+1))=0.4
TF-IDF(to)=0.2*0.4=0.08

I like to eat apples → [0, 0, 0, 0.08, 0, -0.05754, 0, 0.08]

Bag of words-N grams



Vocabulary ={and, apples, bananas, eat, hate, I, like, to, I like, like to, to eat, eat apples, I hate, hate bananas, like apples, apples and, and bananas }

#	And	Apples	bananas	eat	hate	I	like	to	I like	like to	to eat	eat apples	I hate	hate bananas	like apples	apples and	and bananas	TARGET
1	0	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	1
2	0	0	1	0	1	1	0	0	0	0	0	0	1	1	0	0	0	0
3	1	1	1	0	0	1	1	0	1	0	0	0	0	0	1	1	1	1

I like to eat apples → [0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]

NLP: Text embeddings

Session-2

Word embedding

- **Basic CountVectors→ limitations**
  - The product is good→ [1,1,1,1]
  - Is the product good→ [1,1,1,1]
  - Vocabulary →[good, is, product, this]
- **Count-Based Vector Space Model**
- Arrangement of same words in a different order (syntactic) could change the meaning. Which is not captured by CountVectors
- Sequencing issue

Word embedding

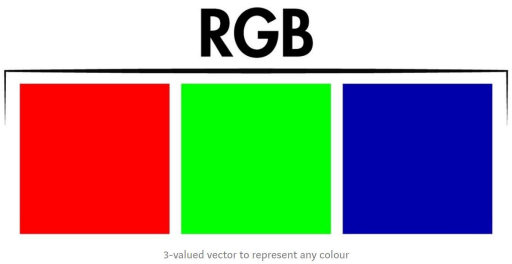
- Have a great time→ [0,1,1,1]
- Have a good time→ [1,0,1,1]
- Vocabulary→[good, great, have, time]
- Semantically similar sentences.
- Different representations by CountVectors/OHE
- Each word is treated as an orthogonal dimension and similarity between words is not captured
- This technique leads to a high dimensional sparse matrix

Word embedding: the idea

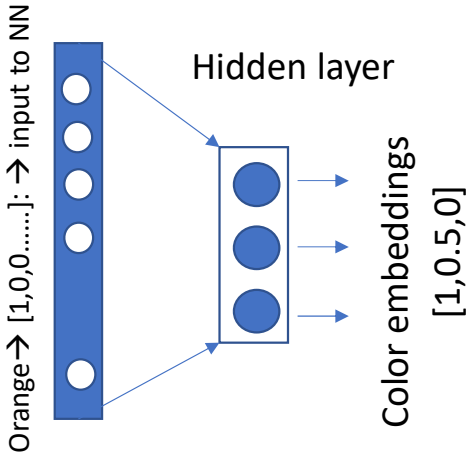
16 colors

Each location in the vector represents a different colour

Orange→ [1,0,0……]: the OHE way



Orange→ [R,G,B]:[1,0.5,0] the feature vector way



- LD representation
- Captures similarity→
  - compare orange and Blue → [1,0.5,0]-[0,0,1]→ large difference
  - compare orange and red → [1,0.5,0]-[1,0,0]→ less difference
- Essentially a FA type DR method

Context matters

- We watched a **monkey** swing effortlessly between the trees.
- My son tends to **monkey** around with the settings on the phone, causing unintended changes.

- he is a **rich** man
- he is **rich** manuel

Adjacent words add to/ modify the meaning of the word

Word embedding

- Word embeddings are word representation as vectors in a multi-dimensional space, that allows words with similar meaning to have a similar representation
  - leads to a low dimensional dense matrix
  - Each word is represented by a real-valued vector, often tens or hundreds of dimensions.
  - This is contrasted to the thousands or millions of dimensions required for sparse word representations
  - Convert input OHE matrix (HDS) to real-valued vector(LDD)
  - **semantic-Based Vector Space Model**

Distributional semantics

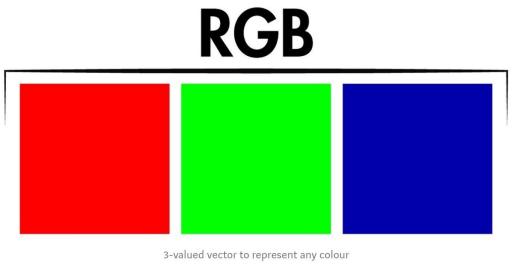
Distributional hypothesis: linguistic items with similar distributions have similar meanings.

Word embedding: the idea

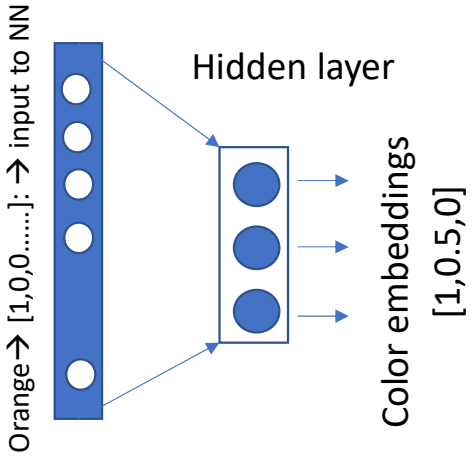
16 colors

Each location in the vector represents a different colour

Orange→ [1,0,0……]: the OHE way



Orange→ [R,G,B]:[1,0.5,0] the feature vector way



- LD representation
- Captures similarity→
  - compare orange and Blue → [1,0.5,0]-[0,0,1]→ large difference
  - compare orange and red → [1,0.5,0]-[1,0,0]→ less difference
- Essentially a FA type DR method

Word embedding→ A DR technique applied to text data

Word embedding: the idea

Vocabulary:
Man, woman, boy, girl, prince, princess, queen, king, monarch

Each word gets a 1x9 vector representation

	1	2	3	4	5	6	7	8	9
man	1	0	0	0	0	0	0	0	0
woman	0	1	0	0	0	0	0	0	0
boy	0	0	1	0	0	0	0	0	0
girl	0	0	0	1	0	0	0	0	0
prince	0	0	0	0	1	0	0	0	0
princess	0	0	0	0	0	1	0	0	0
queen	0	0	0	0	0	0	1	0	0
king	0	0	0	0	0	0	0	1	0
monarch	0	0	0	0	0	0	0	0	1

lower dimensional embedding

	Femininity	Youth	Royalty
Man			
Woman			
Boy			
Girl			
Prince			
Princess			
Queen			
King			
Monarch			

	Femininity	Youth	Royalty
Man	0	0	0
Woman	1	0	0
Boy	0	1	0
Girl	1	1	0
Prince	0	1	1
Princess	1	1	1
Queen	1	0	1
King	0	0	1
Monarch	0.5	0.5	1

Each word gets a 1x3 vector

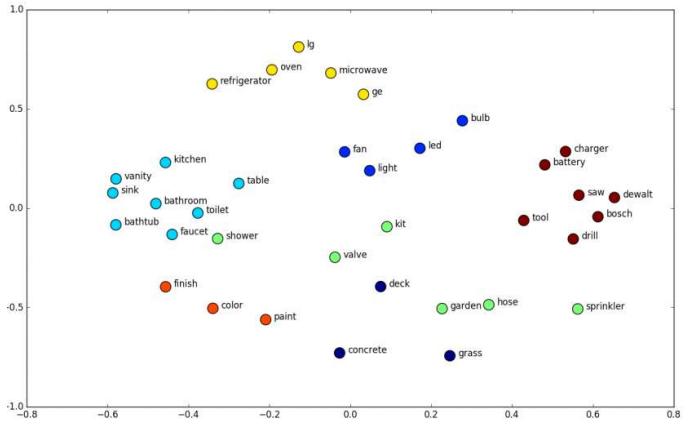
Similar words... similar vectors

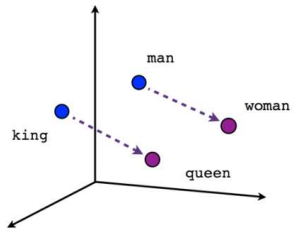
1. Each word is represented with a lower-dimensional vector (3 instead of 9).
2. Similar words have similar vectors here. There's a smaller distance between the embeddings for "girl" and "princess", than from "girl" to "prince". In this case, distance is defined by Euclidian distance.
3. The embedding matrix is much less sparse (less empty space),
4. We could add more words to the vocabulary without increasing the dimensionality. For instance, the word "child" might be represented with [0.5, 1, 0].
5. Relationships between words are captured and maintained, e.g. the movement from king to queen, is the same as the movement from boy to girl, and could be represented by [+1, 0, 0].

https://www.shanelynn.ie/get-busy-with-word-embeddings-introduction/

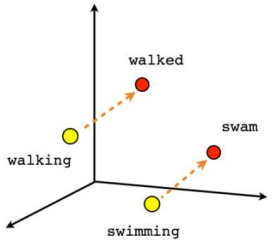
Outcomes of WE

- Recognizes words that are similar,
- Naturally captures the relationships between words as we use them

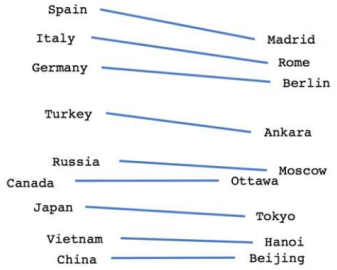




Male-Female



Verb tense



Country-Capital

- The relationships between words in the embedding space lend themselves to unusual word algebra, allowing words to be added and subtracted, and the results actually making sense. For instance, in a well-defined word embedding model, calculations such as (where [[x]] denotes the vector for the word 'x')

[[king]] – [[man]] + [[woman]] = [[queen]]

[[Paris]] – [[France]] + [[Germany]] = [[Berlin]]

Extracting word embedding

- The focus is to optimize the embeddings such that the core meanings and the relationships between words is maintained

- The central idea of word embedding training is that similar words are typically surrounded by the same “context” words in normal use

Different Embedding approaches

1. Word2Vec
2. Glove
3. FastText
4. ELMO (Sequential model-based)
5. BERT (Transformer-based)

$$\left[\begin{bmatrix}0.1\\0.3\\0.8\\0.9\\|\\|\\|\\0\end{bmatrix},\begin{bmatrix}0\\0.4\\0.6\\0.2\\|\\|\\|\\0.5\end{bmatrix},\begin{bmatrix}0.2\\0.3\\0.4\\0.7\\|\\|\\|\\0.1\end{bmatrix},\begin{bmatrix}0\\0.1\\0\\0.7\\|\\|\\|\\0.2\end{bmatrix}\right]$$

n × m Tensor for ‘m’ words/tokens

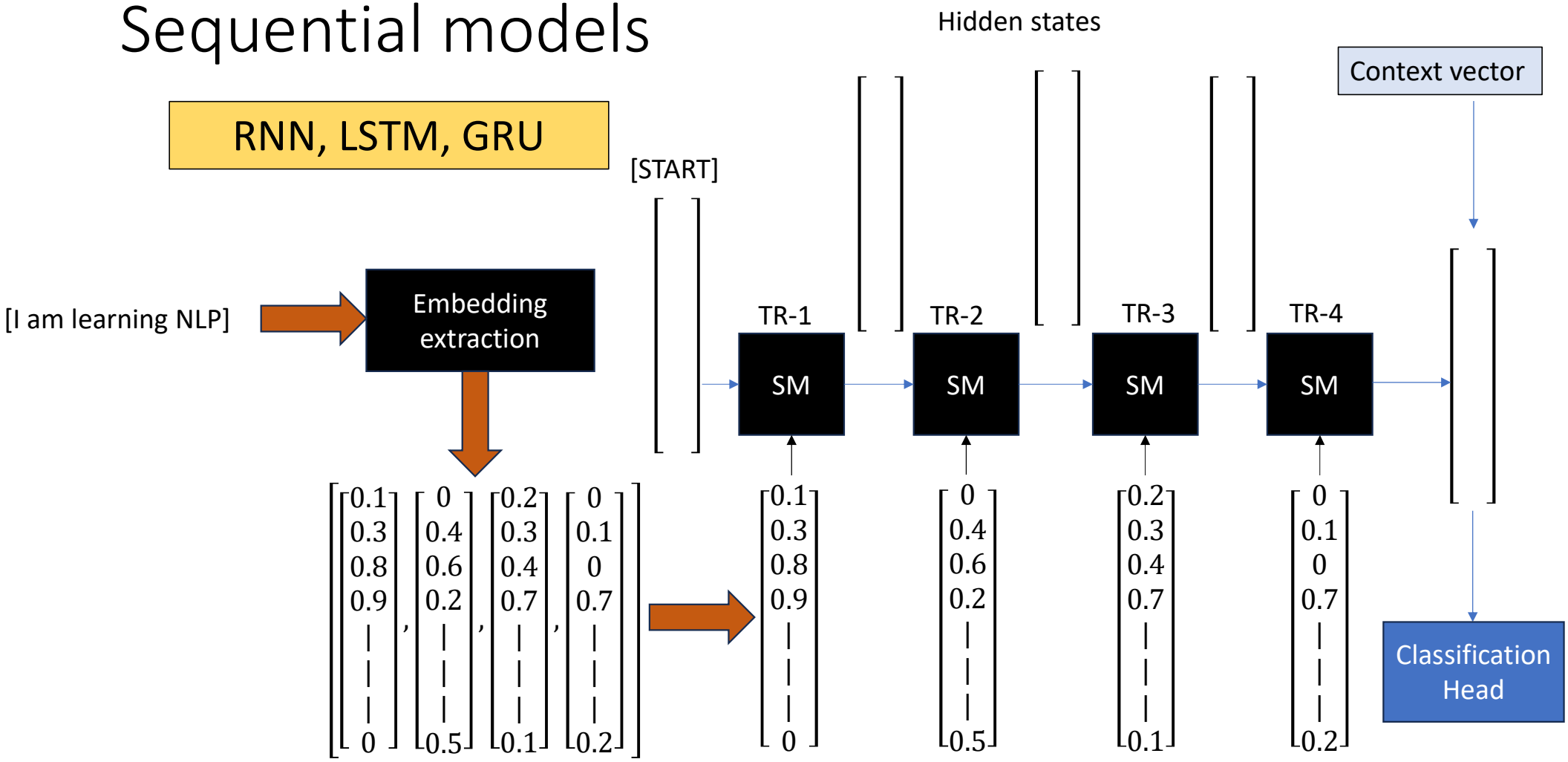
‘n’ dimensional matrix for each word

How to use this Tensor as input?

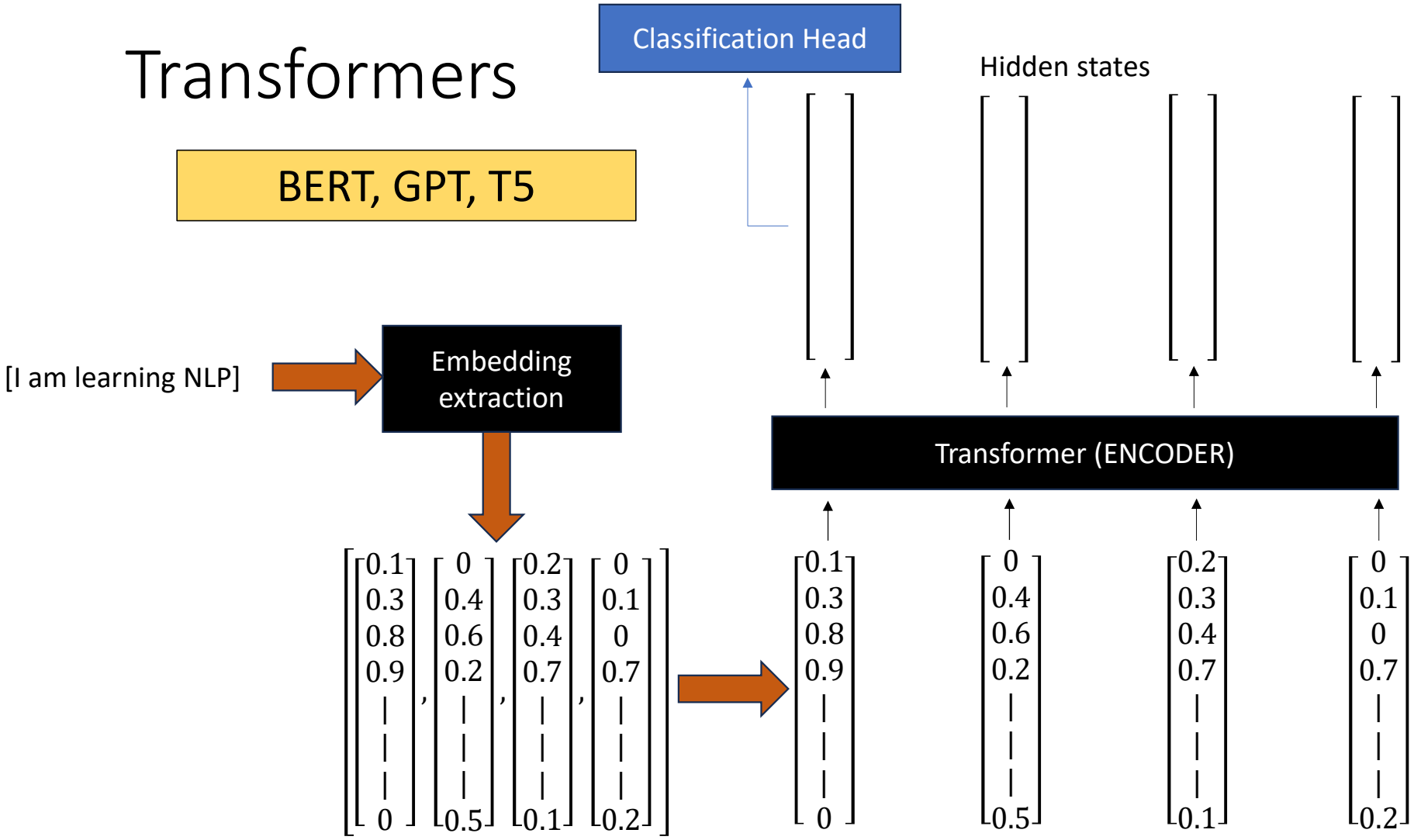
1) Aggregation→ Not a smart idea
2) Sequential models
3) Transformers

Google News is a news aggregator app developed by Google. It presents a continuous, customizable flow of articles organized from thousands of publishers and magazines. Google News is available as an app on Android, iOS, and the Web. Google released a beta version in September 2002 and the official app in January 2006.’

Sequential models



Transformers



word2vec

- Developed by Google
- Word2vec NNet models to get word embedding
- Not a Deep NNet.
- Has only one hidden layer
- Two approaches
  - CBOW
  - Skip-gram
- terms → ‘target’ and ‘context’

The following example demonstrates multiple pairs of target and context words as training samples, generated by a 5-word window sliding along the sentence.

> “The man who passes the sentence should swing the sword.” – Ned Stark

Sliding window (size = 5)	Target word	Context
[The man who]	the	man, who
[The man who passes]	man	the, who, passes
[The man who passes the]	who	the, man, passes, the
[man who passes the sentence]	passes	man, who, the, sentence
…	…	…
[sentence should swing the sword]	swing	sentence, should, the, sword
[should swing the sword]	the	should, swing, sword
[swing the sword]	sword	swing, the

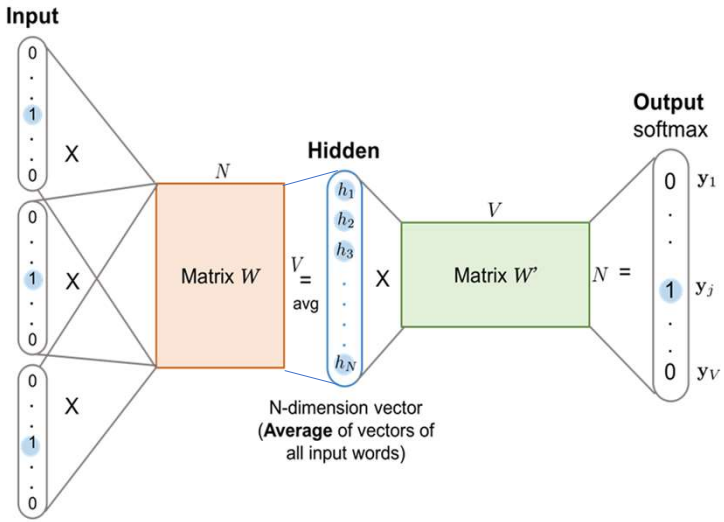
Each context-target pair is treated as a new observation in the data. For example, the target word “swing” in the above case produces four training samples: (“swing”, “sentence”), (“swing”, “should”), (“swing”, “the”), and (“swing”, “sword”).

The central idea of word embedding training is that similar words are typically surrounded by the same “context” words in normal use.

https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html

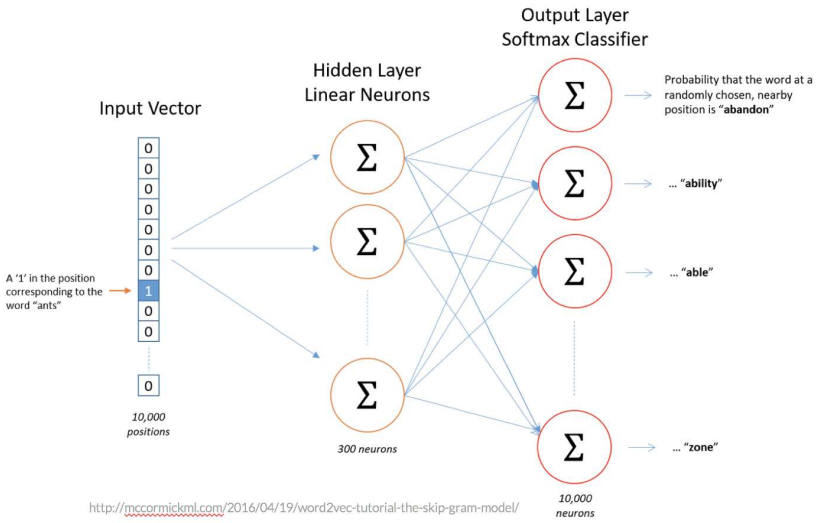
CBOW

- Uses a neural network to predict a target word, given a context of words
- Create embeddings for the target word
- Has a projection/ averaging layer
- No activation function for the hidden layer
- SoftMax AF for the output layer.
- Weights are learnt for vocabulary classification (given context words, predict the probability of target words)
- Once trained on a corpus, the OL can be discarded since we are interested only in the EL/HL output



Skip-gram

- Uses a neural network to predict context words, given a target word
- Create embeddings for the target word
- No activation function for the hidden layer
- SoftMax AF for the output layer.
- Once trained on a corpus, the OL can be discarded since we are interested only in the EL/HL output



CBOW Vs. Skip-gram

- **Skip-gram:** works well with small amount of the training data, represents well even rare words or phrases.
- **CBOW:** several times faster to train than the skip-gram, slightly better accuracy for the frequent words.

Applications of W2V

- **Analyzing Survey Responses**
- **Recommendation systems**

GloVe

- **Global Vectors for Word Representation**
- Developed by Stanford University
- Glove is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space (https://nlp.stanford.edu/projects/glove/)
- The word vector representation is in terms of a ratio of probability

GloVe

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

- I love programing.
- I love Math.
- I tolerate Biology

	I	love	Programming	Math	tolerate	Biology
I	0	2	0	0	1	0
love	2	0	1	1	0	0
Programming	0	1	0	0	0	0
Math	0	1	0	0	0	0
tolerate	1	0	0	0	0	1
Biology	0	0	0	0	1	0

$P_{love\text{-}I}=2/3$

$P_{tolerate\text{-}I}=1/3$

Create co-occurrence matrix

- Calculate co-occurrence probabilities
- Calculate word vectors such that

$$J = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2 ,$$

A new weighted least squares regression model

GloVe Vs word2vec

- No clear winner
- GloVe does better in word analogy task
- We have better options
- *contextualized* word-embeddings

The problem: polysemy

- will you read this article
- Can you read this article
- I heard that you read this article **(same word different tenses)**

- to sanction is to permit
- put sanction on means not permit **(same word different meaning)**

- W2V and Glove lead to same vector for the words read/sanction in these the sentences

- Issue with polysemous words

Embeddings from Language Models (ELMo)

- Unlike word2vec and GLoVe, the ELMo assigns a vector to a token/word that is a function of the entire sentence containing that word.
- ELMo word representations take the entire input sentence into equation for calculating the word embeddings
- Hence, the same word can have different vectors under different contexts

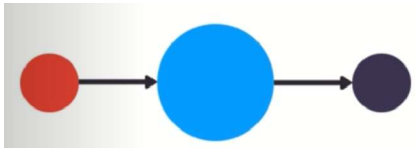
More advanced Embedding techniques

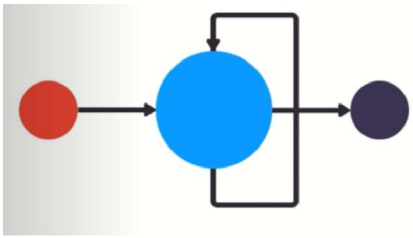
- BERT
- GPT-*

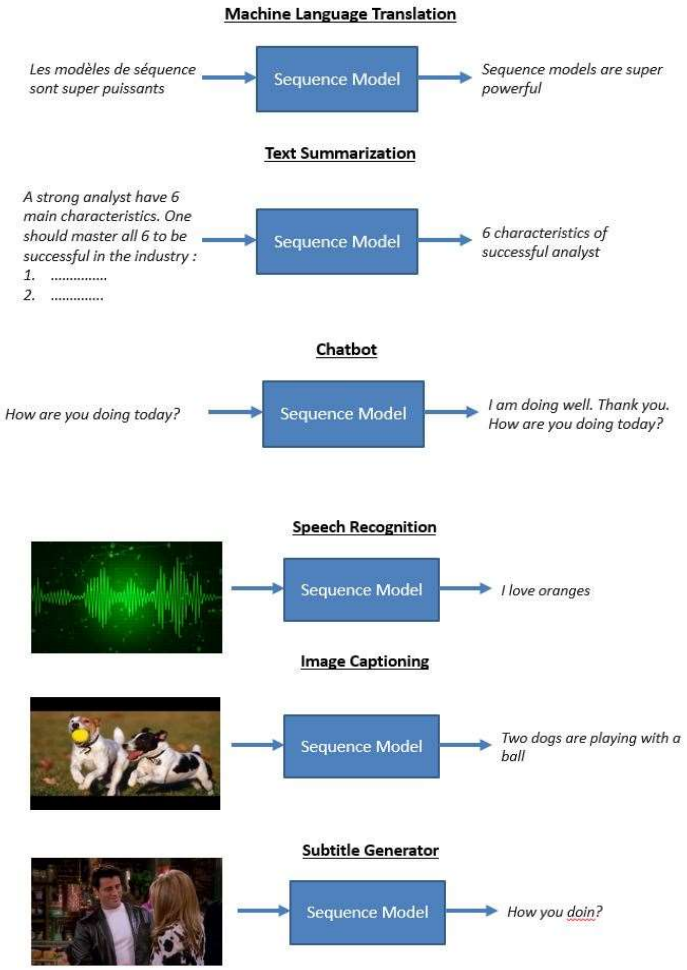
Sequential modeling

Sequential modeling

- Conventional prediction
- $Y = f(x_i : \{i=1..k\})$
- Sequential models
- $Y_n = f(x_{n-m} : \{m=1..j\})$

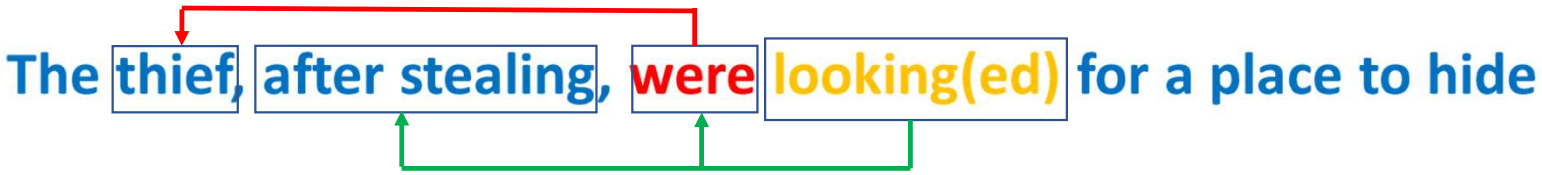






Sequential models

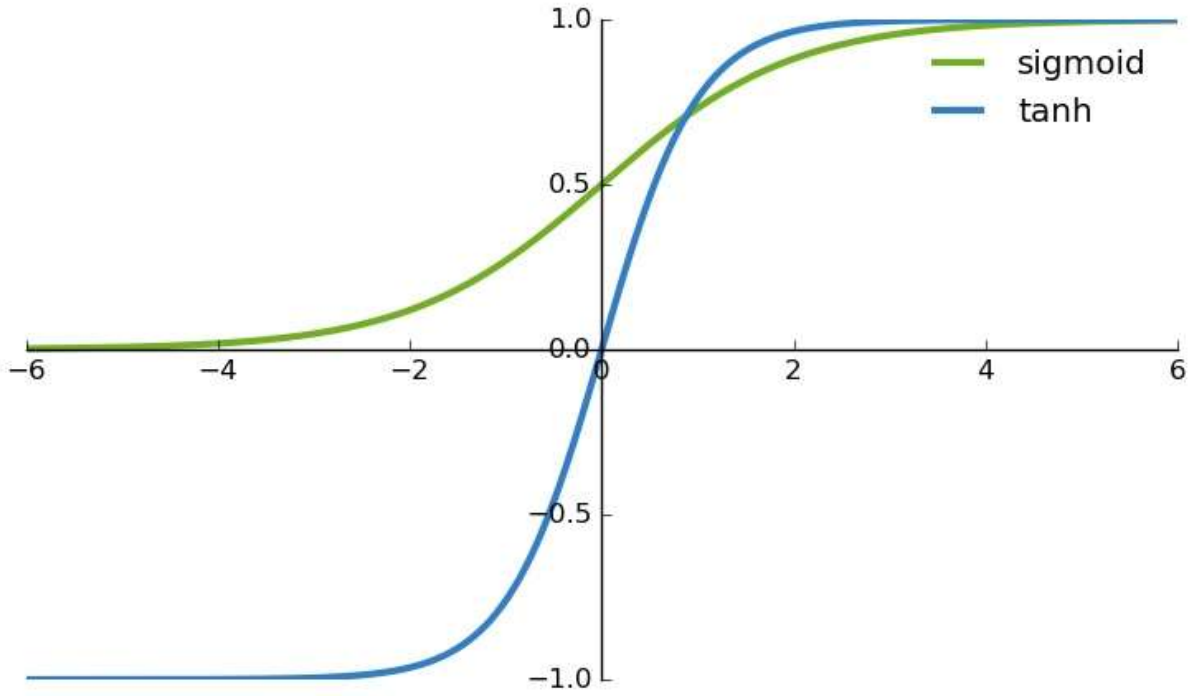
- **The thief, after stealing, was looking(ed) for a place to hide**
- **The thief, after stealing, were looking(ed) for a place to hide**
- To use the correct ***state verb***, we need to **'remember'** if the subject is singular or plural
- To use correct action verb tense, we need to **'remember'** the state verb
- **Sequential models use internal memory and memory manipulation for this**



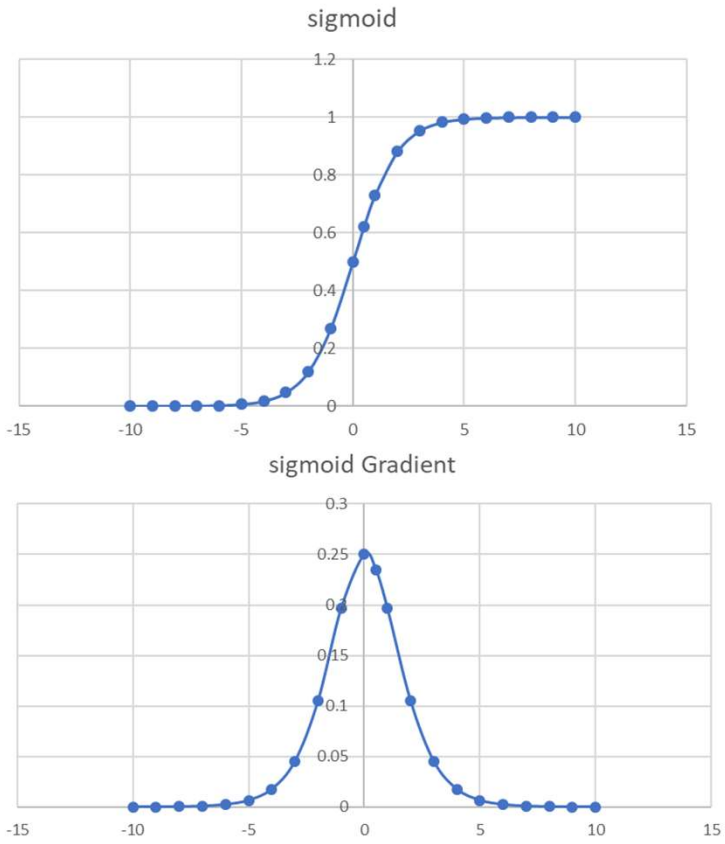
Non-Sequential Vs. Sequential

- Relational
  - In non-sequential, all the inputs are independent of each other
  - In sequence, all the inputs are related
- Input/output size
  - Fixed in NS
  - Varies in S

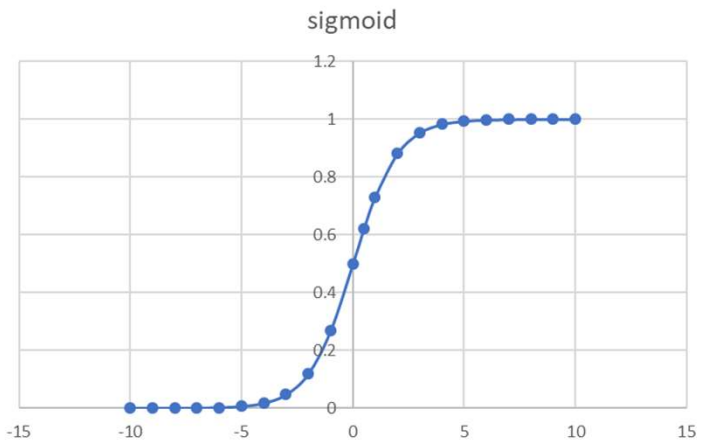
Activation Functions in RNNs

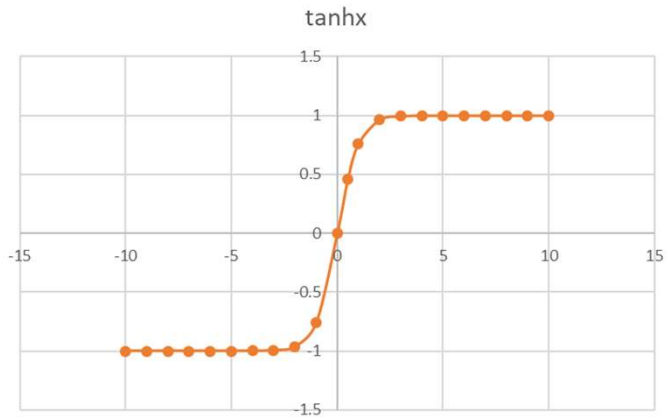


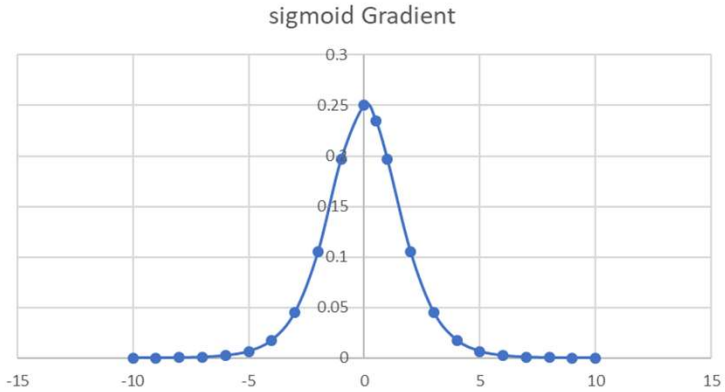
Sigmoid

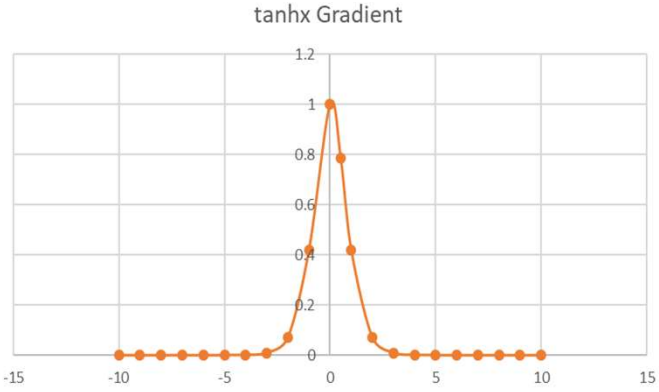

sigmoid
1.2
1
0.8
0.6
0.4
0.2
0
-15
-10
-5
0
5
10
15
sigmoid Gradient
0.3
0.25
0.2
0.15
0.1
0.05
0
-15
-10
-5
0
5
10
15

Sigmoid and tanh

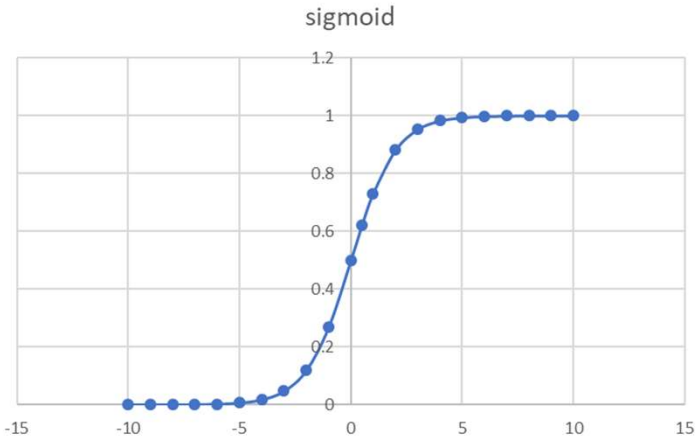


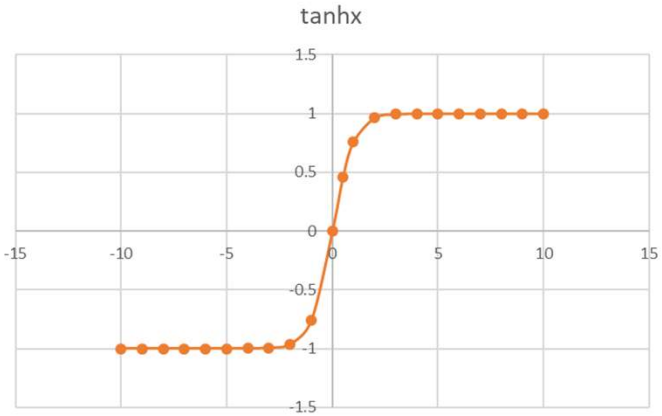


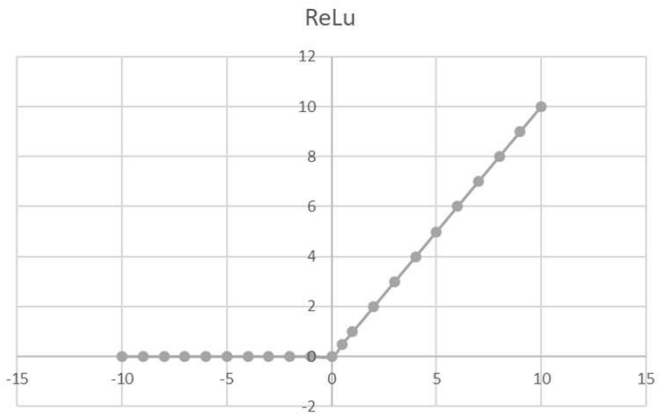


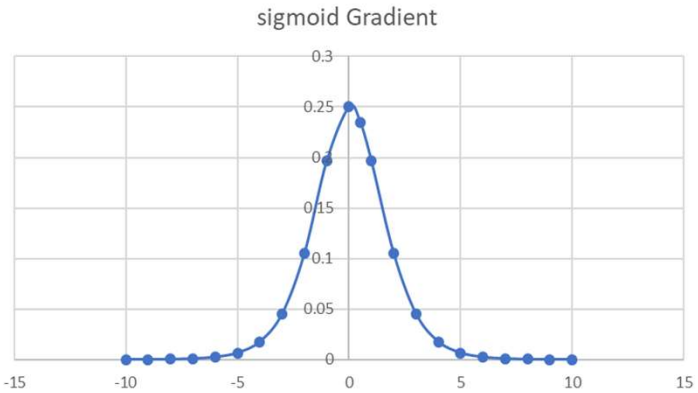


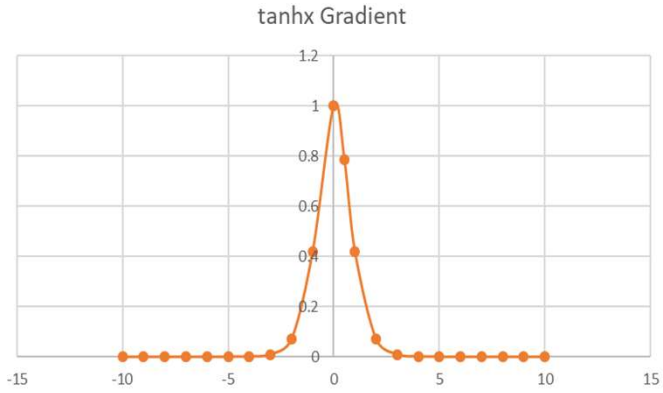
Sigmoid, tanh, ReLu

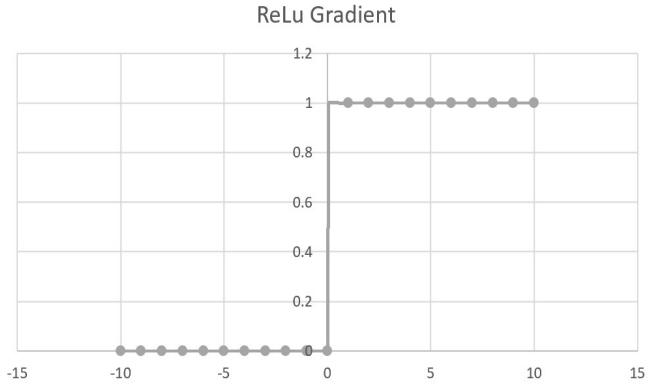




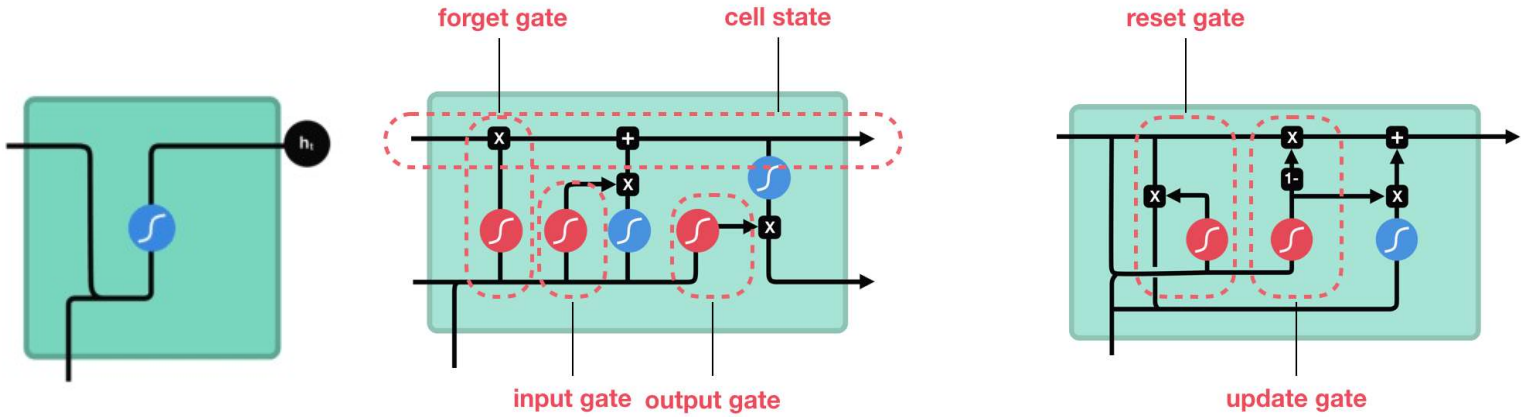






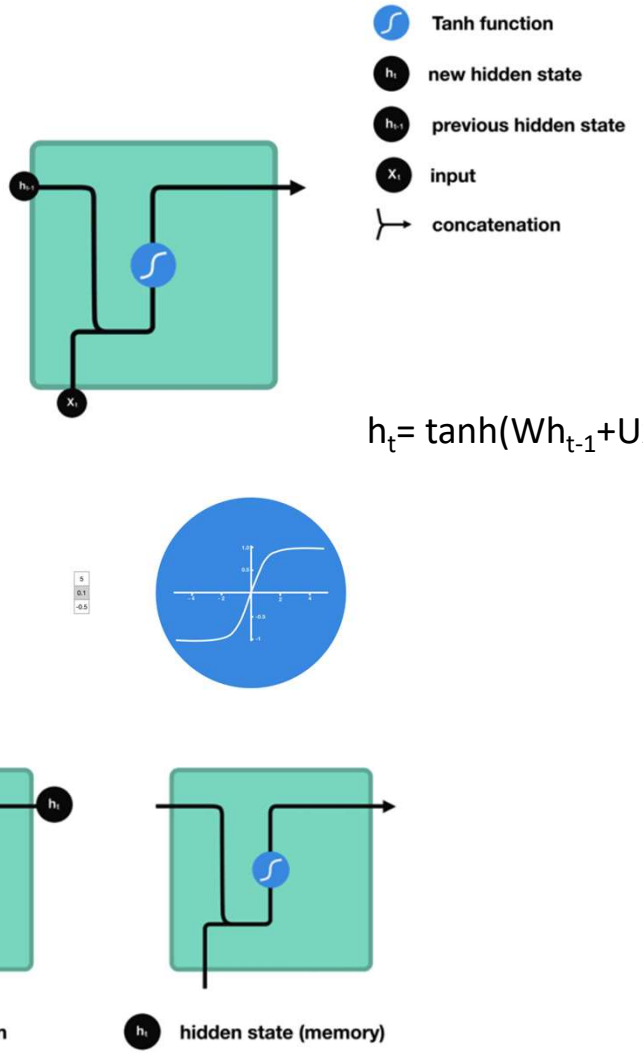


Sequential model ANNs

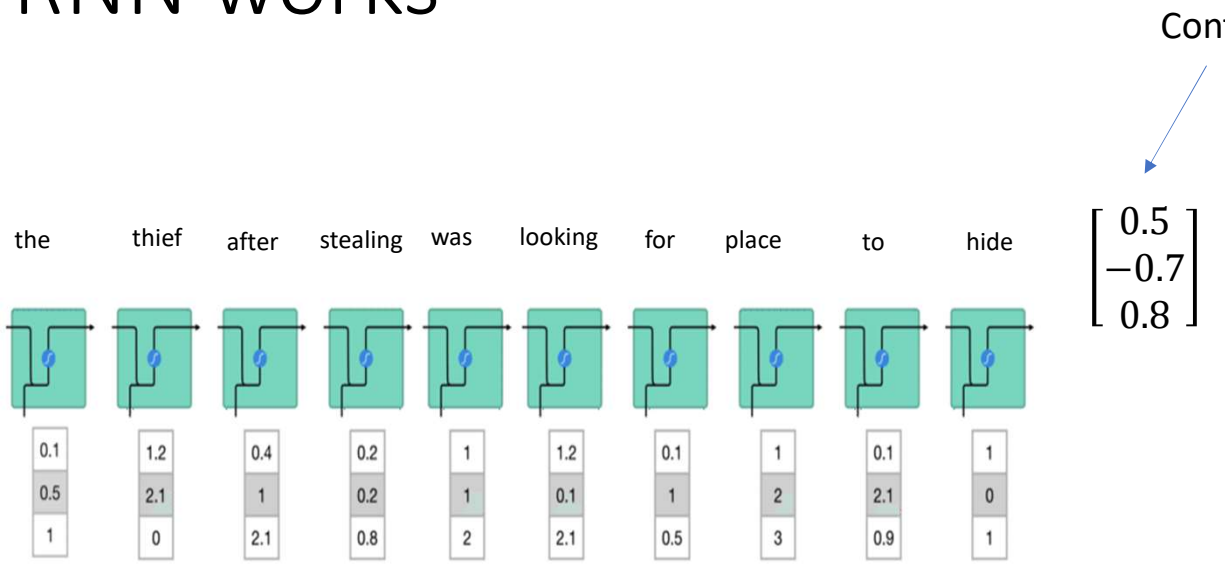


How vanilla RNN works

- Words get transformed into machine-readable vectors(embeddings)
- RNN processes the sequence of vectors one by one
- While processing, it passes the previous hidden state to the next step of the sequence
- **The hidden state acts as the neural networks memory**



How vanilla RNN works




```
from keras.layers import SimpleRNN
rnn = SimpleRNN(10, return_sequences=True)
output = rnn(input_data)
# 'output' will contain hidden states for all time steps
```

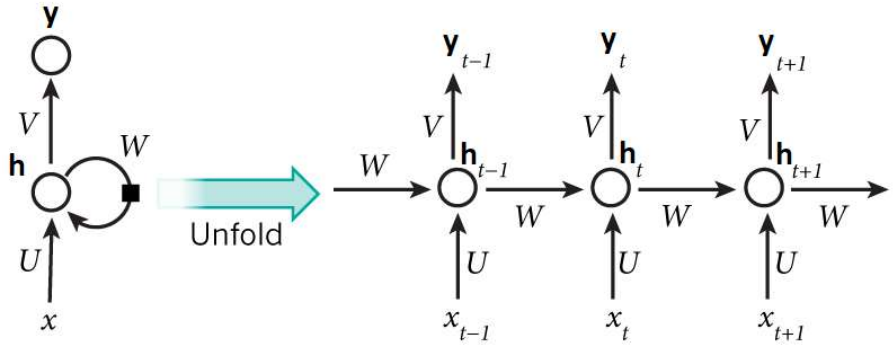
(batch_size,time_steps,10)

```
from keras.layers import SimpleRNN
rnn = SimpleRNN(10, return_sequences=False)
output = rnn(input_data)
# 'output' will contain only the last hidden state
```

(batch_size, 10)

Vanilla RNN

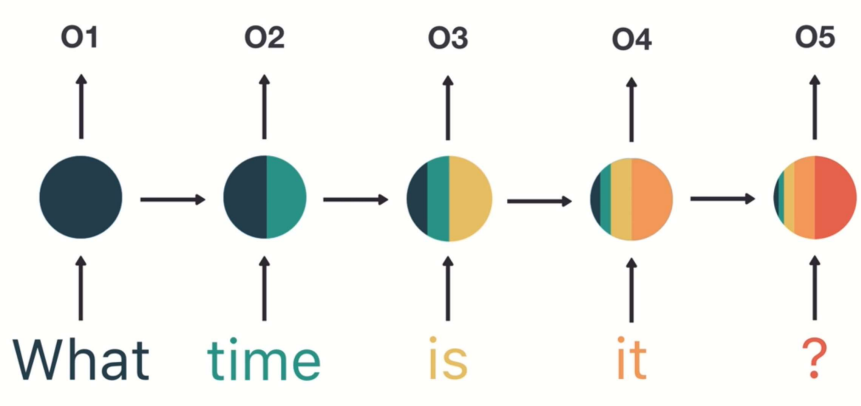
- Current state $h_t=f(h_{t-1},x_t)$
- With activation function: $h_t= \tanh(Wh_{t-1}+Ux_t+b_h)$
- *Output state* $y_t= g(\mathbf{V}\, h_t+b_o)$
  - **W** is *the weight at previous hidden state,*
  - **U** is the *weight at current input state,*
  - **V** is the *weight at the output state*



http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

Vanilla RNN

- Limitation:
  - **can capture only short-term relations**
  - **Looses sight of long-term dependencies due to multiple operations (vanishing gradient)**
- How to solve this problem
  - **A way to decide which part of the sequence to remember**
  - **A way to decide what aspect of the current input to remember**
- LSTM solves this problem through “**gate(-ing)**” approach



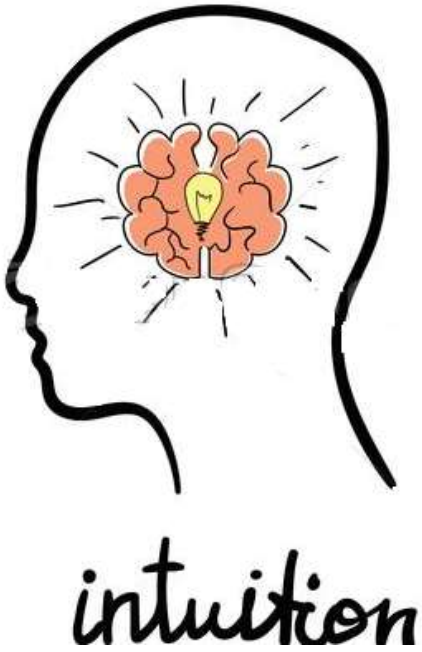
https://www.youtube.com/watch?v=LHXXI4-IEns

LSTM

- The intuition:

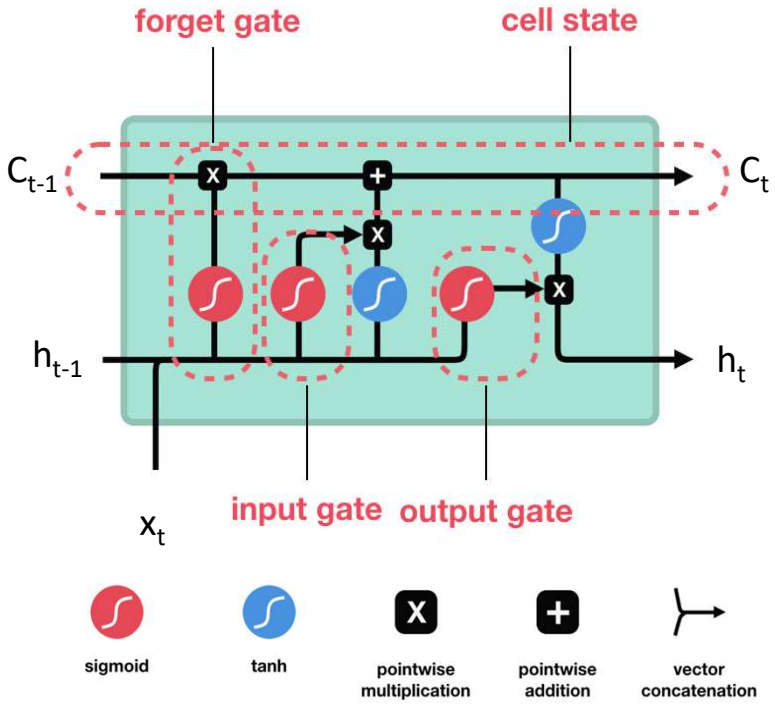
- *Amazing, the movie is outstanding. I have watched it twice already and will watch it again.*

- ***Amazing***, ~~*the movie is*~~ ***outstanding***. ~~*I have*~~ ***watched*** ~~*it*~~ ***twice*** ~~*already and will*~~ ***watch*** ~~*it*~~ ***again***

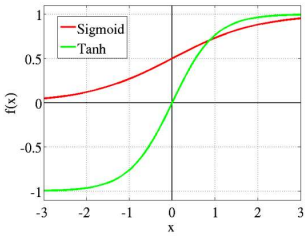


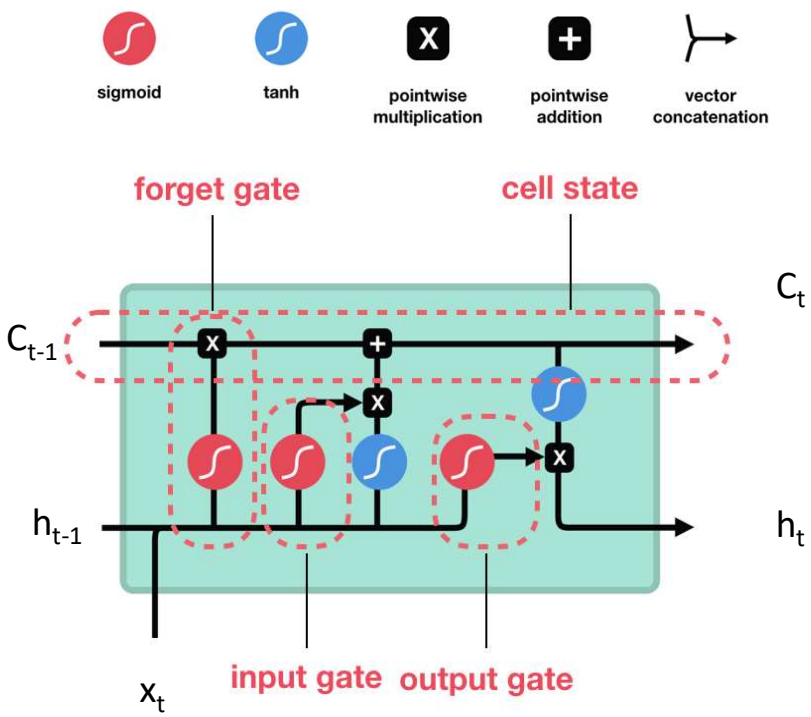
LSTM

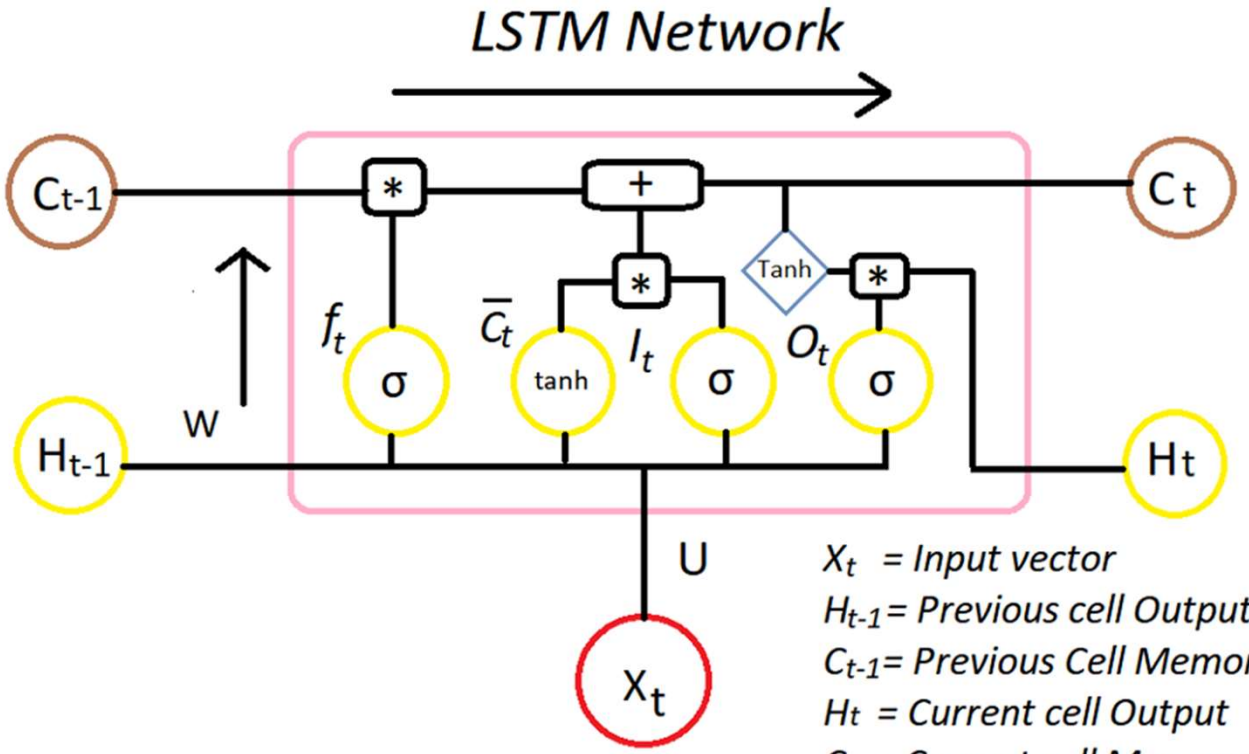
- The **Forget gate** decides what is relevant to keep from prior steps.
- The **input gate** decides what information is relevant to add from the current step.
- The **output gate** determines what the next hidden state should be.



LSTM


Sigmoid
Tanh
f(x)
x
1
0.5
0
-0.5
-1
-3
-2
-1
0
1
2
3


sigmoid
tanh
pointwise multiplication
pointwise addition
vector concatenation
forget gate
cell state
C_{t-1}
C_t
h_{t-1}
h_t
x_t
input gate
output gate



X_t = Input vector
H_{t-1} = Previous cell Output
C_{t-1} = Previous Cell Memory
H_t = Current cell Output
C_t = Current cell Memory

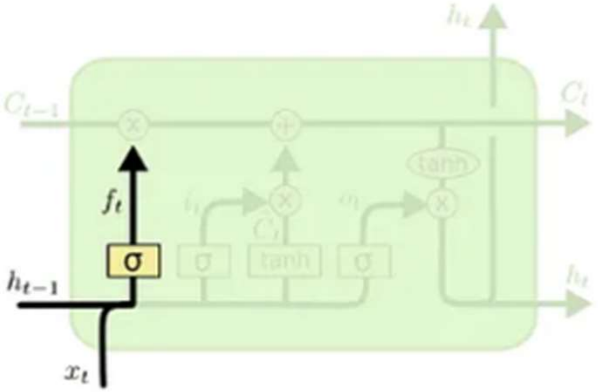
(*) = Element-wise multiplication
(+) = Element-wise addition

$$f_t = \sigma (X_t * U_f + H_{t-1} * W_f)$$
$$\bar{C}_t = \tanh (X_t * U_c + H_{t-1} * W_c)$$
$$I_t = \sigma (X_t * U_i + H_{t-1} * W_i)$$
$$O_t = \sigma (X_t * U_o + H_{t-1} * W_o)$$

$$C_t = f_t * C_{t-1} + I_t * \bar{C}_t$$
$$H_t = O_t * \tanh (C_t)$$

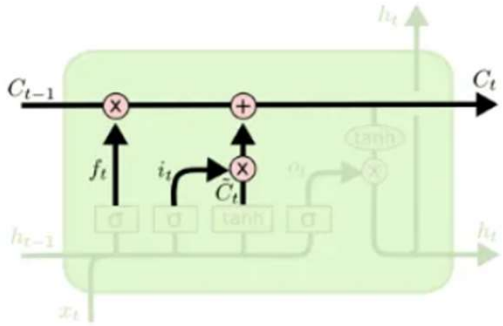
W, U = weight vectors for forget gate (f), candidate (c), i/p gate (I) and o/p gate (O)

Note : These are different weights for different gates, for simpicity's sake, I mentioned W and U



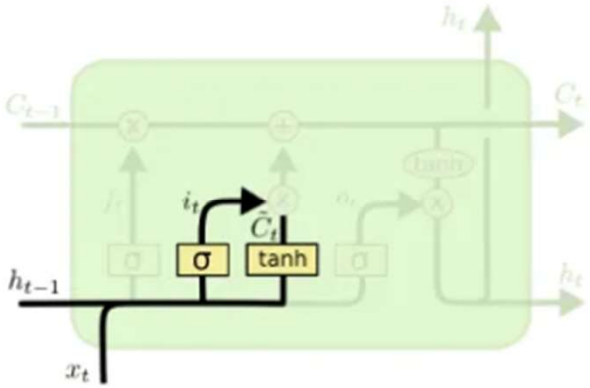
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

A forget gate



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

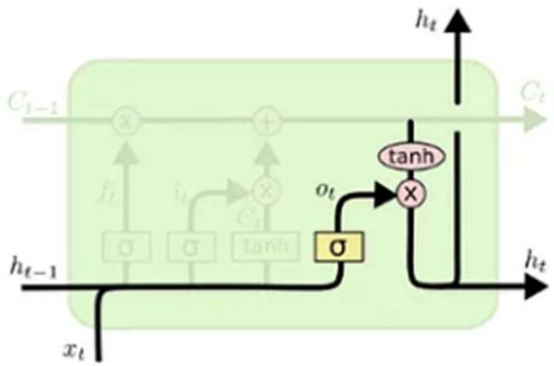
Updating Cell State



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

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

An Input Gate

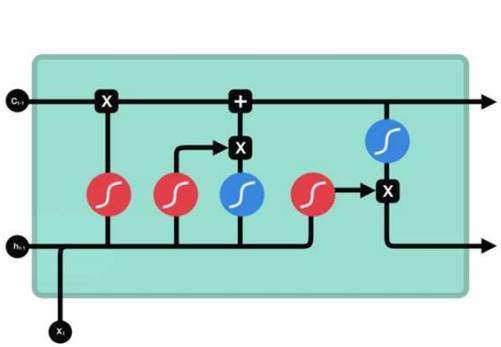


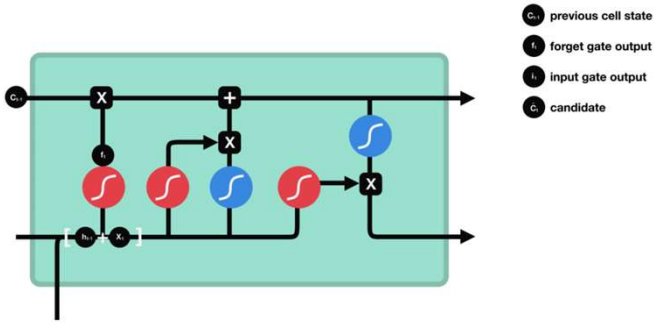
$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

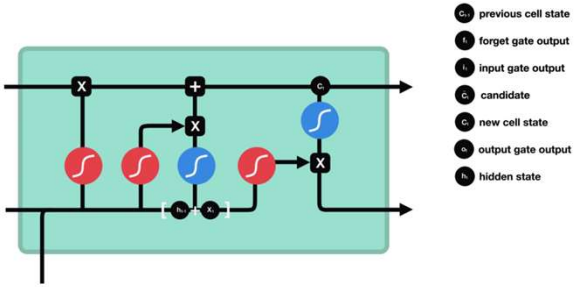
$$h_t = o_t * \tanh\left(C_t\right)$$

Output Gate

LSTM







Forget gate

- This gate decides what information should be forgotten or remembered.
- Information from the previous hidden state and information from the current input is passed through the sigmoid function.
- Values come out between 0 and 1.
- The closer to 0 means to forget, and the closer to 1 means to remember

Input Gate

- To update the cell state, we have the input gate.
- First, pass the previous hidden state and current input into a sigmoid function.
- That decides which values will be updated by transforming the values to be between 0 and 1.
- Also pass the hidden state and current input into the tanh function
- Multiply the tanh output with the sigmoid output.
- The sigmoid output will decide which information is important to keep from the tanh output

Output Gate

The output gate decides what the next hidden state should be.
Remember that the hidden state contains information on previous inputs.
The hidden state is also used for predictions.
First, pass the previous hidden state and the current input into a sigmoid function. Then we pass the newly modified cell state to the tanh function.
Multiply the tanh output with the sigmoid output to decide what information the hidden state should carry. The output is the hidden state

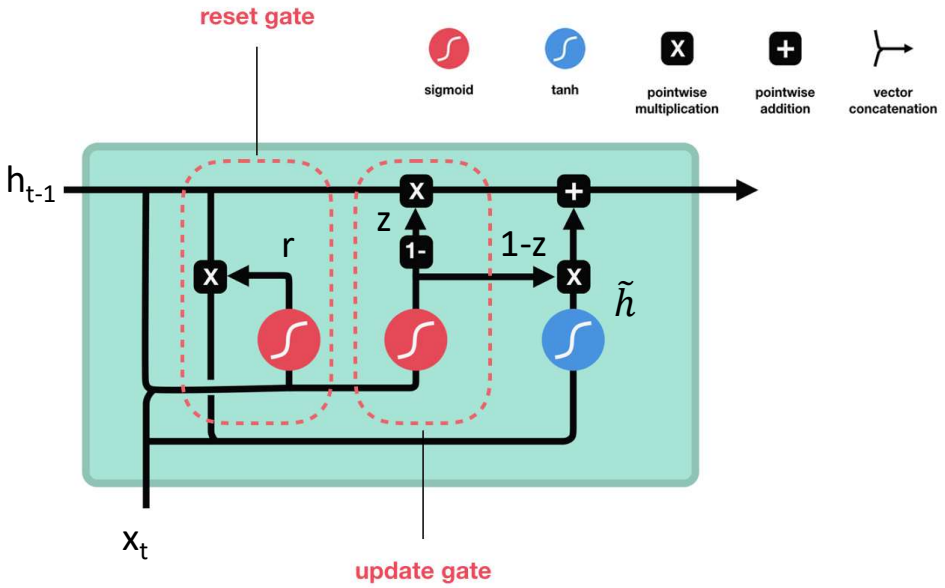
- First, the cell state gets pointwise multiplied by the forget vector.
- This has a possibility of dropping values in the cell state if it gets multiplied by values near 0.
- Then we take the output from the input gate and do a pointwise addition which updates the cell state to new values that the neural network finds relevant. That gives us our new cell state

GRU

- Gated Recurrent Unit
- Uses two gates to decide
- How much past information to remember
- what new information to add

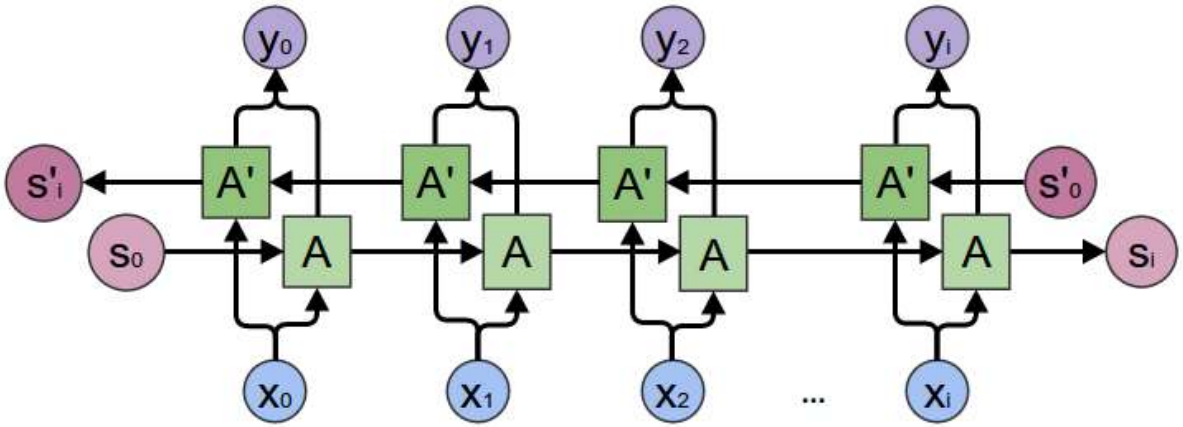
$$z = \sigma(W_z \cdot x_t + U_z \cdot h_{(t-1)} + b_z)$$
$$r = \sigma(W_r \cdot x_t + U_r \cdot h_{(t-1)} + b_r)$$
$$\tilde{h} = tanh(W_h \cdot x_t + r * U_h \cdot h_{(t-1)} + b_h)$$
$$h = z * h_{(t-1)} + (1 - z) * \tilde{h}$$

The reset gate is used to decide how much past information to forget



The update gate acts similar to the forget and input gate of an LSTM.
It decides what new information to add

Bidirectional LSTM Vs Transformer



As opposed to directional models, which read the text input sequentially (left-to-right or right-to-left), the Transformer encoder reads the entire sequence of words at once. Therefore it is considered bidirectional, though it would be more accurate to say that it's non-directional. This characteristic allows the model to learn the context of a word based on all of its surroundings (left and right of the word).

References

- https://youtu.be/8HyCNIVRbSU
- http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21
- https://towardsdatascience.com/sequence-models-by-andrew-ng-11-lessons-learned-c62fb1d3485b

Transformers

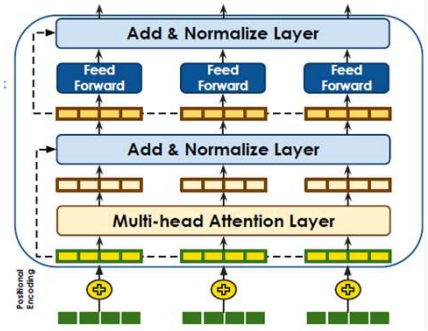
Elements of Transformers

Transformers

Encoder

- Word embedding
- Position encoding
- Multi-headed self-Attention
- Skip Connections and Layer Normalization
- Feed forward

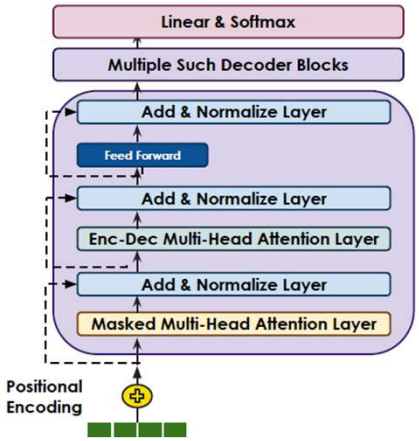
Processes the input sequence and encodes its representation



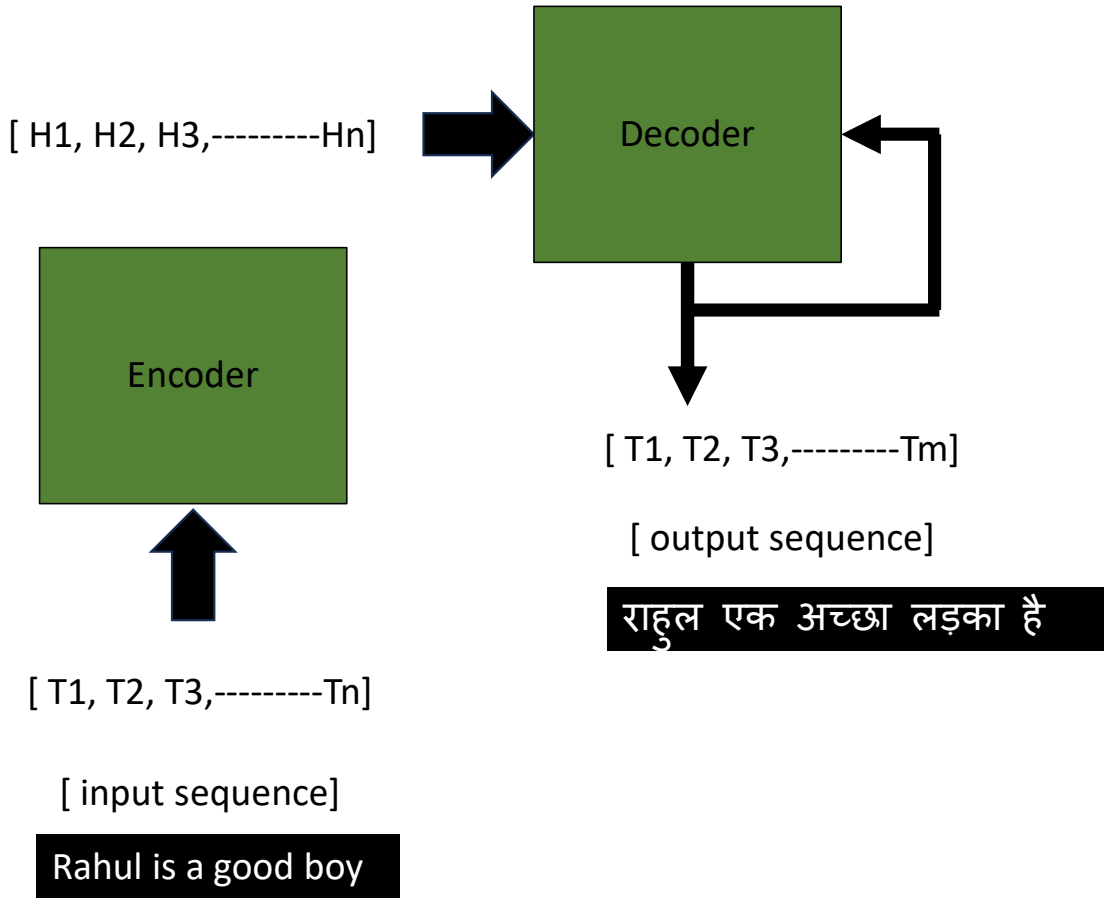
Decoder

- Position encoding
- Multi-headed self-Attention
- Masked Multi-headed attention
- Encoder-Decoder attention
- Skip Connections and Layer Normalization
- Linear and SoftMax

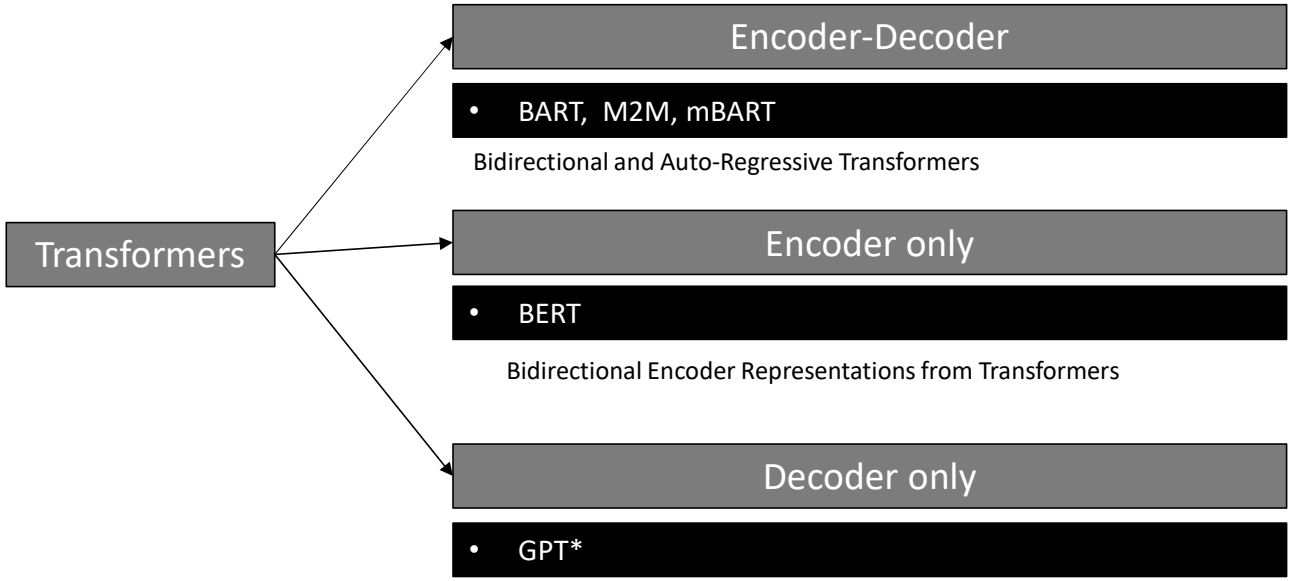
Generates the output sequence (such as the translated sentence) by attending to both the encoded input and the previously generated output



Encode-Decoder



Transformer types



machine translation, summarization, and text generation

Semantic analysis, NER, SA, Semantic Search, classification

Text generation, Chatbot, Code generation

Attention types

Attention

Self-attention

- Understand the relevance of one word to another to capture the meaning

Focus on input sequence only

Masked Self-attention

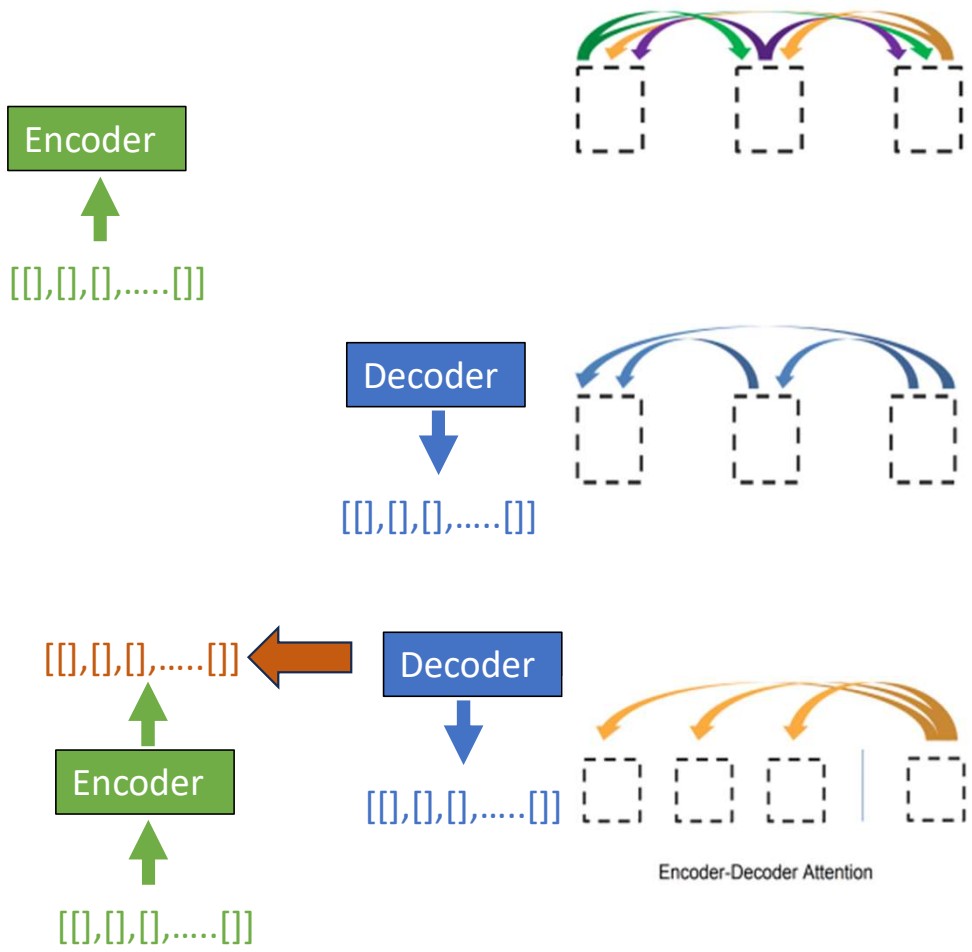
- Hide the future words from the decoder during training
- Mask padding (add high negative value)

Focus on output sequence only

Encoder-Decoder

- Helps the decoder to focus on specific words will predicting the next word

decoder focuses on relevant parts of the input sequence, produced by the encoder, when generating each token

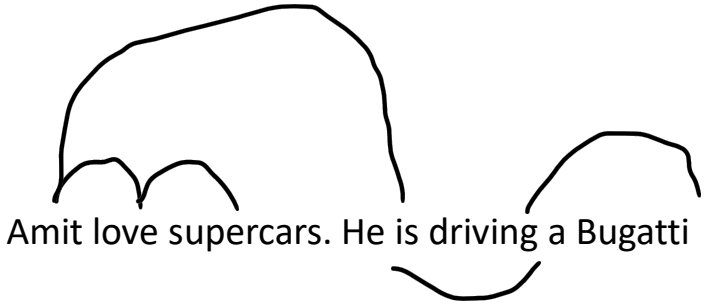


Multi-headed self-attention

The meaning of the word depends on words that may or may not be in the immediate neighbourhood

Self-Attention is a way to determine this dependency

One of the core elements of Transformers is Multi-headed Self-attention



Each word will look at what word is relevant to it

As part of this exercise, multiple relationships might stem out

Each Self-attention captures one of such relations

Multi-headed attention captures multiple relations

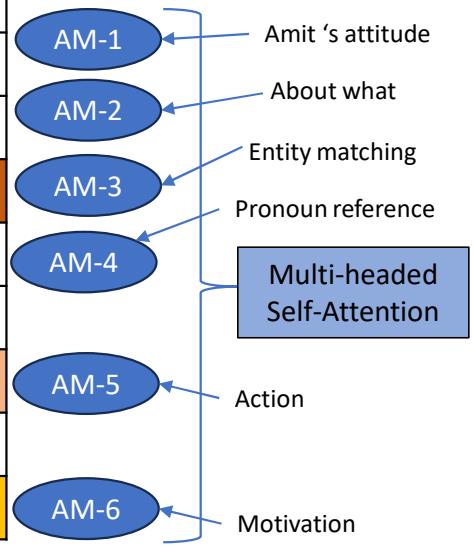
Multi headed self-attention

The meaning of the word depends on words that may or may not be in the immediate neighbourhood

Self-Attention is a way to determine this dependency

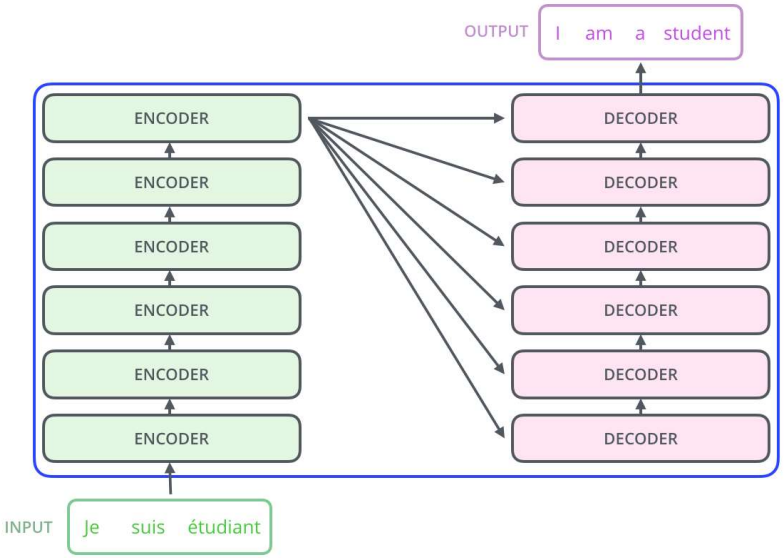
One of the core elements of Transformers is Multi-headed Self-attention

	amit	loves	supercars	he	is	driving	a	bugatti
amit								
loves								
supercars								
he								
is								
driving								
a								
bugatti								



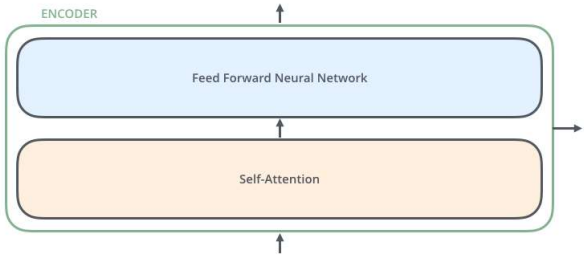
Amit is driving a Bugatti since he loves supercars

Transformers

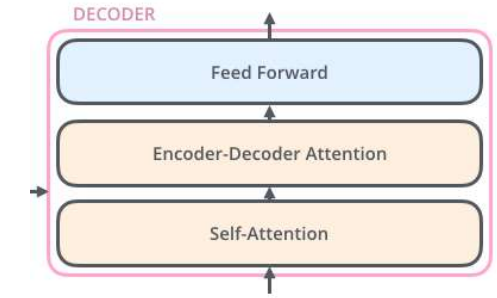


All encoders have the same architecture.

All decoders have the same architecture.

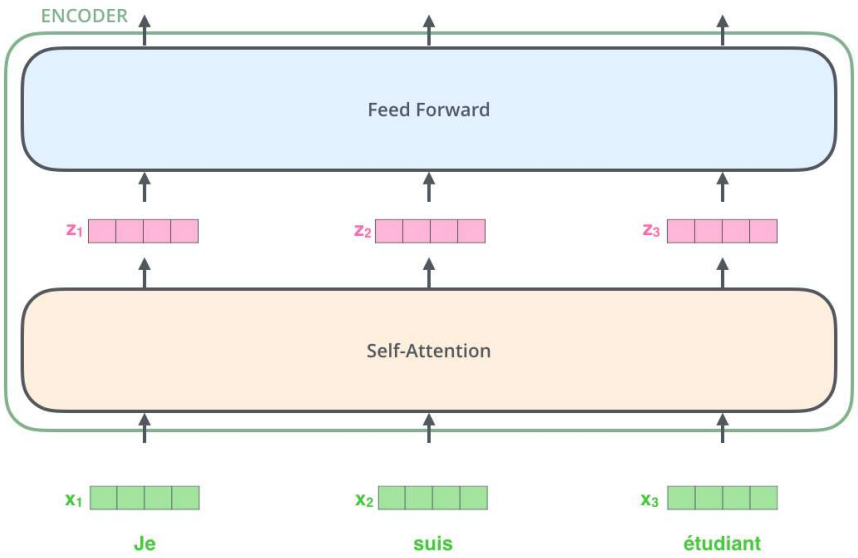


The encoder's inputs first flow through a **self-attention** layer. It helps the encoder look at other words in the input sentence as it encodes a specific word



The decoder has SA and FF and also , a layer between them: an attention layer that helps the decoder focus on relevant parts of the input sentence

Self Attention



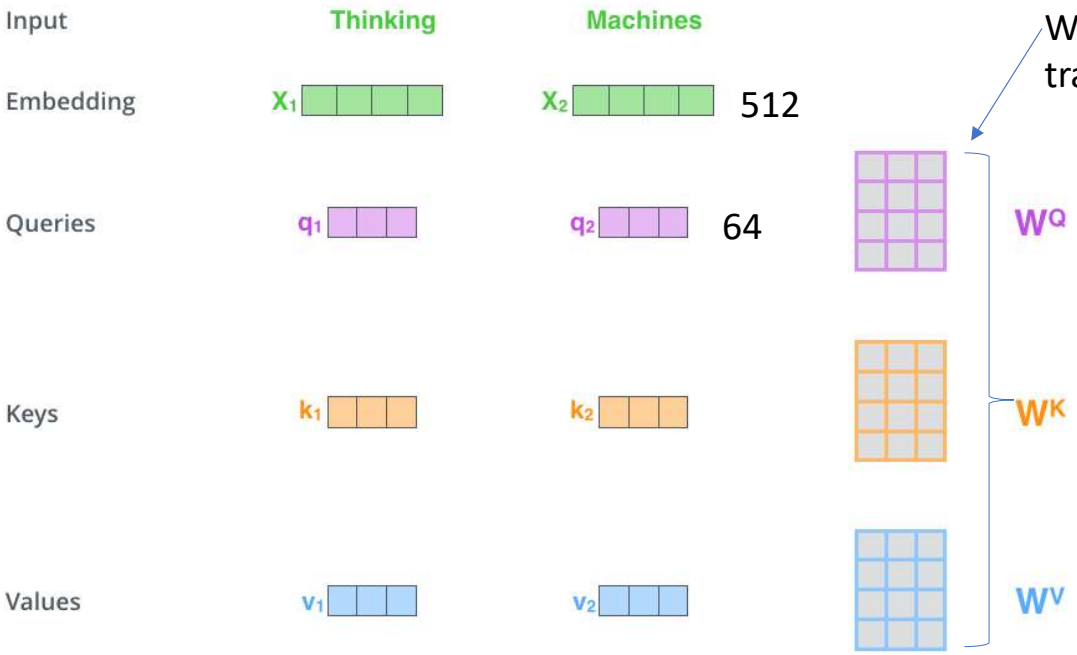
Each word is embedded into a vector of size 512

One key property of the Transformer, which is that the word in each position flows through its own path in the encoder {positional encoding}, dependencies captured only in SA.

Steps in SA:

- Create three vectors from each of the encoder's input vectors. 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

Self Attention



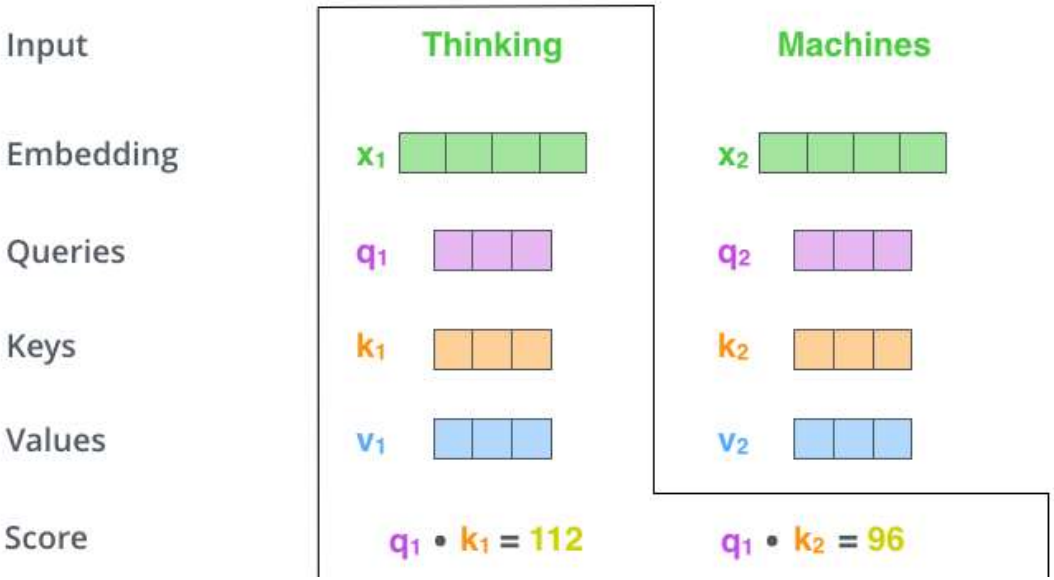
Weight matrices obtained during training

Steps in SA:

STEP-1

- Create three vectors from each of the encoder's input vectors. 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

Self Attention



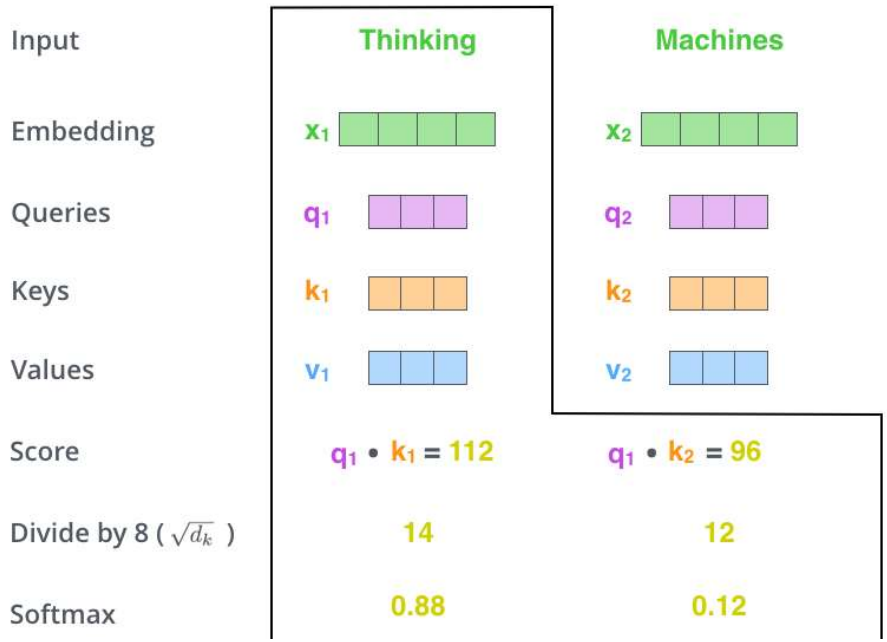
Steps in SA:

STEP-2

- Calculate scores for each word input against the other words. 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

Self Attention



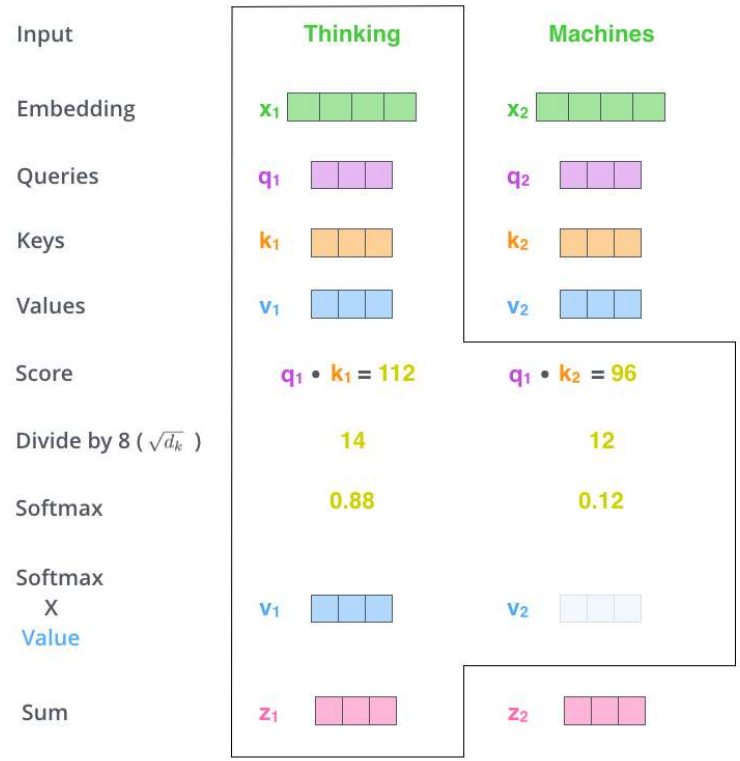
Steps in SA:

<u>STEP-3 and 4</u>

- Divide by 8
- Pass the result through a SoftMax operation. SoftMax normalizes the scores so they're all positive and add up to 1

This SoftMax score determines how much each word will be expressed at this position. Clearly the word at its position will have the highest SoftMax score, but it's useful to attend to another word that is relevant to the current word

Self Attention



Steps in SA:

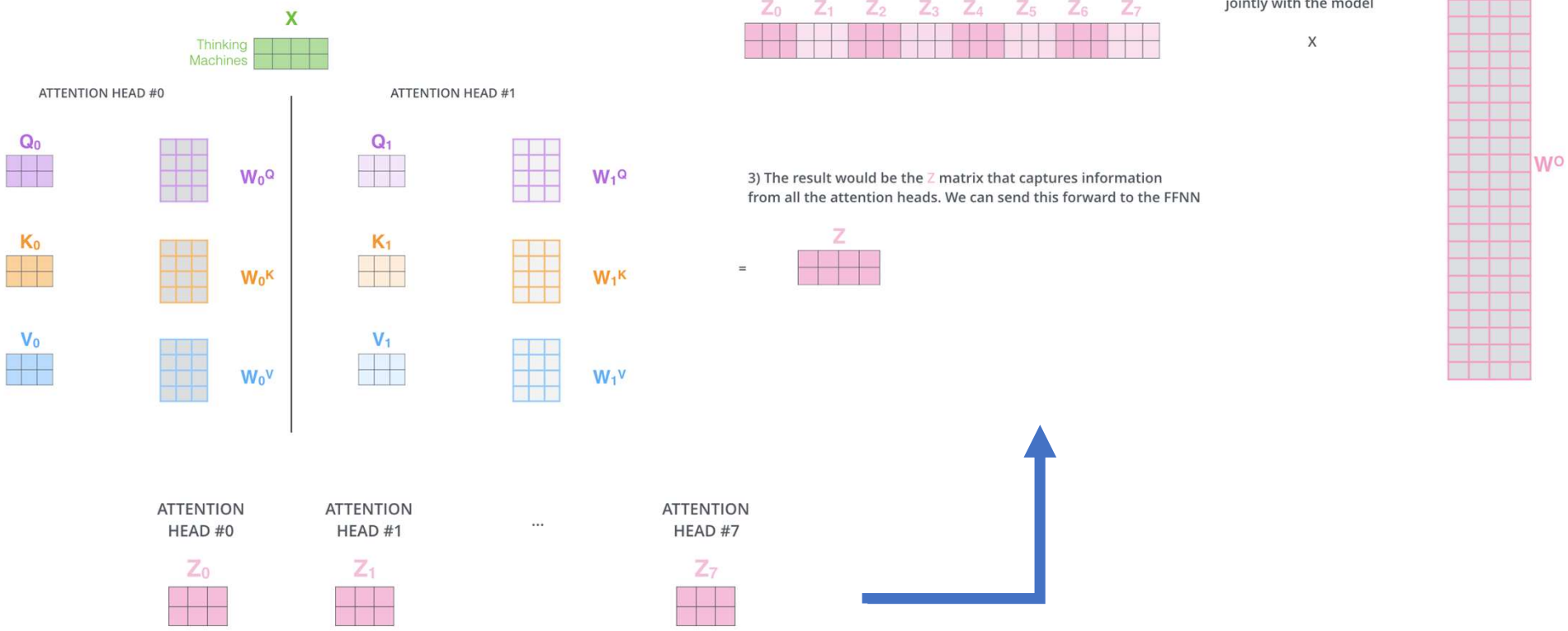
STEP-5 and 6

- multiply each value vector by the SoftMax score. The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words

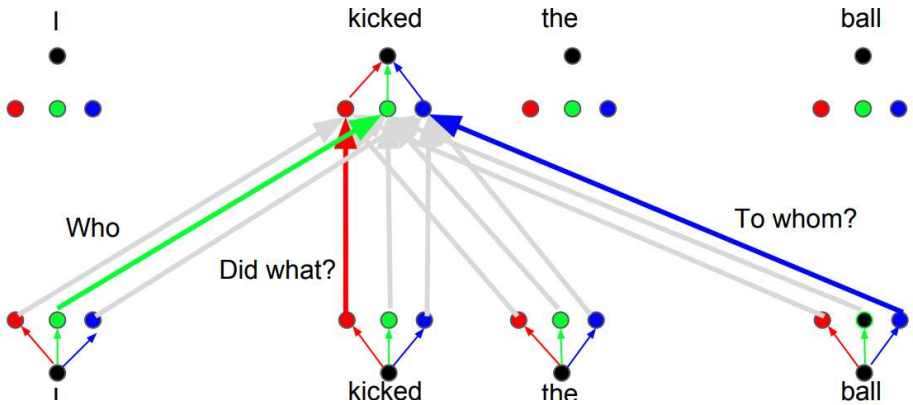
- Sum up the weighted value vectors. This produces the output of the self-attention layer at this position

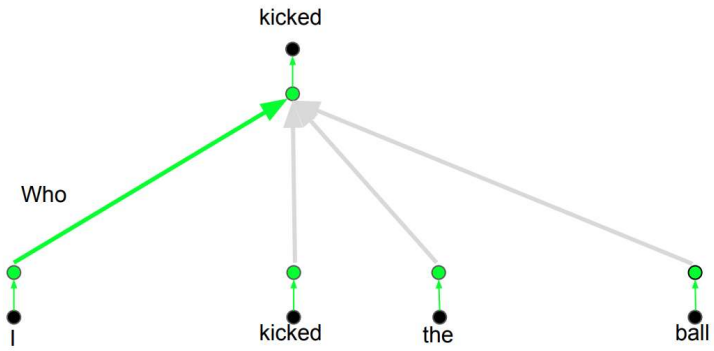
This SoftMax score determines how much each word will be expressed at a position. Clearly the word at its own position will have the highest SoftMax score, but it's useful to attend to another word that is relevant to the current word

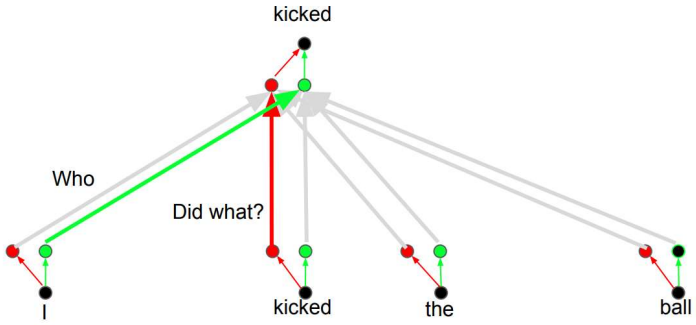
Multiheaded attention



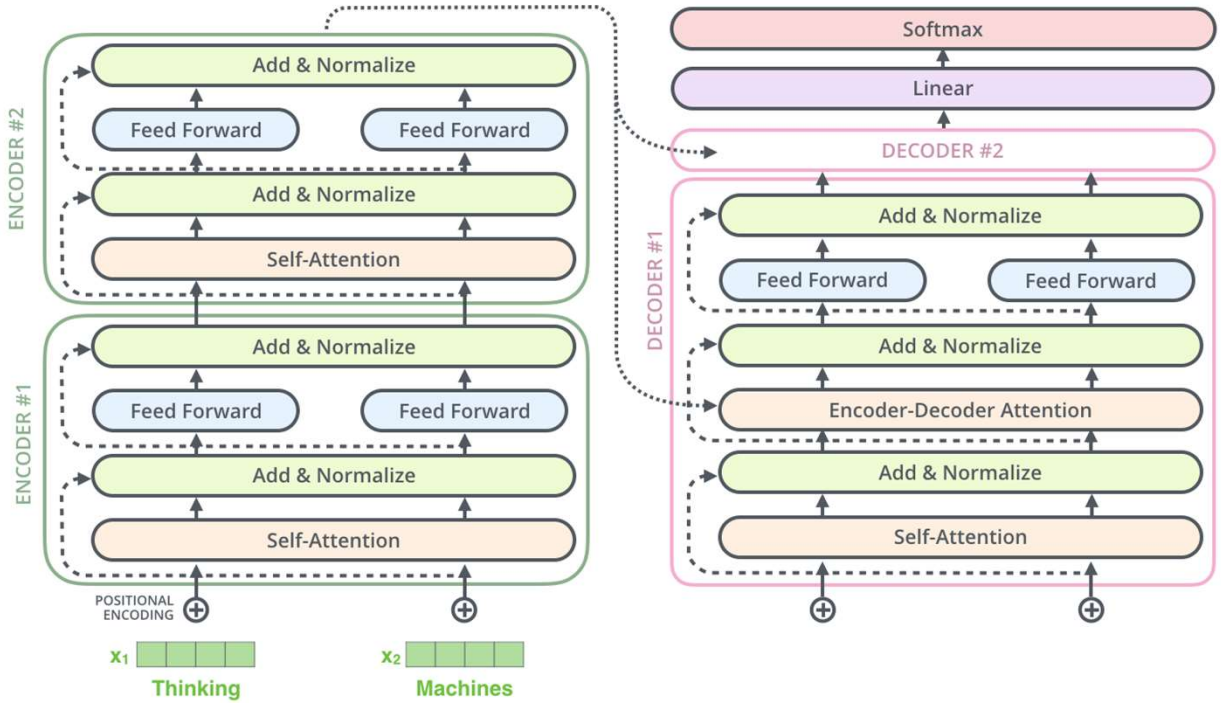
Multi-head attention



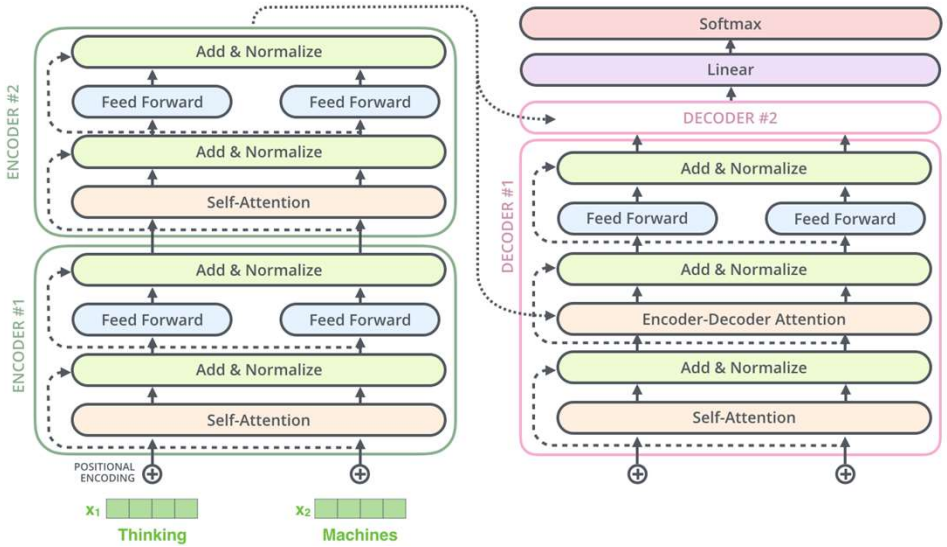




Transformer: more details



Transformer: positional encoding



- https://d2l.ai/chapter_attention-mechanisms/self-attention-and-positional-encoding.html#subsec-positional-encoding

10.6.3. Positional Encoding

Unlike RNNs that recurrently process tokens of a sequence one by one, self-attention ditches sequential operations in favor of parallel computation. To use the sequence order information, we can inject absolute or relative positional information by adding *positional encoding* to the input representations. Positional encodings can be either learned or fixed. In the following, we describe a fixed positional encoding based on sine and cosine functions [Vaswani et al., 2017].

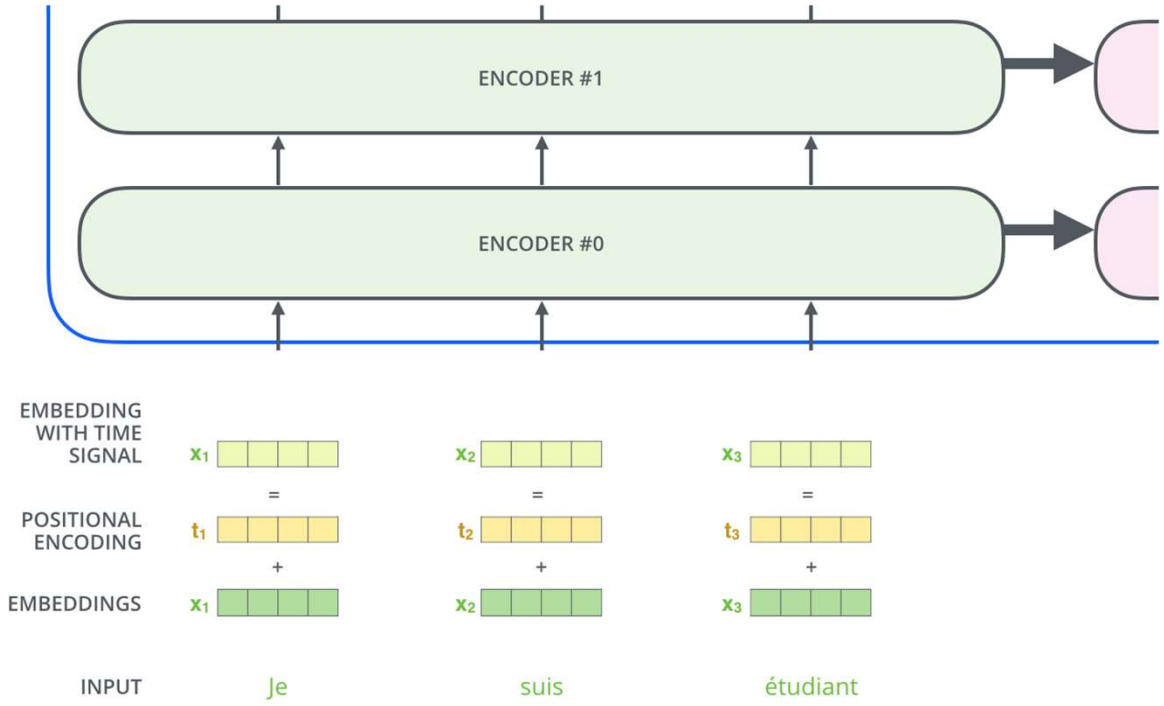
Suppose that the input representation $\mathbf{X} \in \mathbb{R}^{n \times d}$ contains the d-dimensional embeddings for n tokens of a sequence. The positional encoding outputs $\mathbf{X} + \mathbf{P}$ using a positional embedding matrix $\mathbf{P} \in \mathbb{R}^{n \times d}$ of the same shape, whose element on the i^{th} row and the $(2j)^{\text{th}}$ or the $(2j+1)^{\text{th}}$ column is

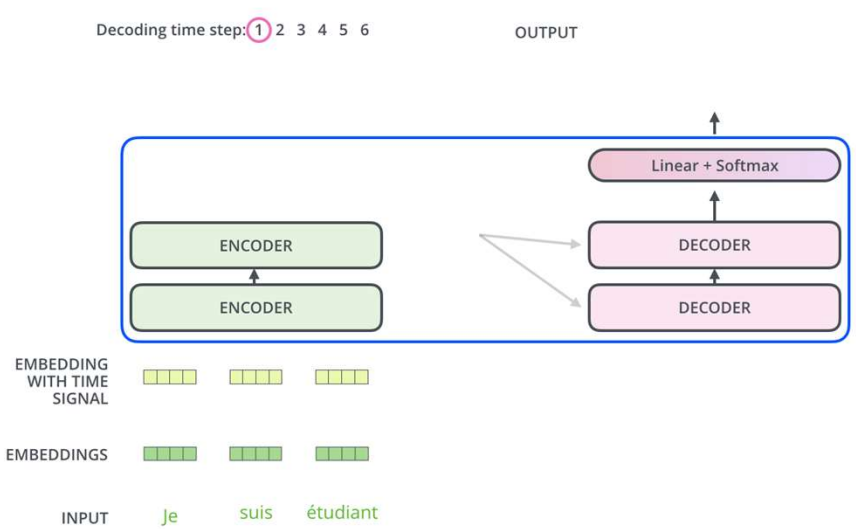
$$p_{i,2j} = \sin\left(\frac{i}{10000^{2j/d}}\right), \qquad (10.6.2)$$
$$p_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right).$$

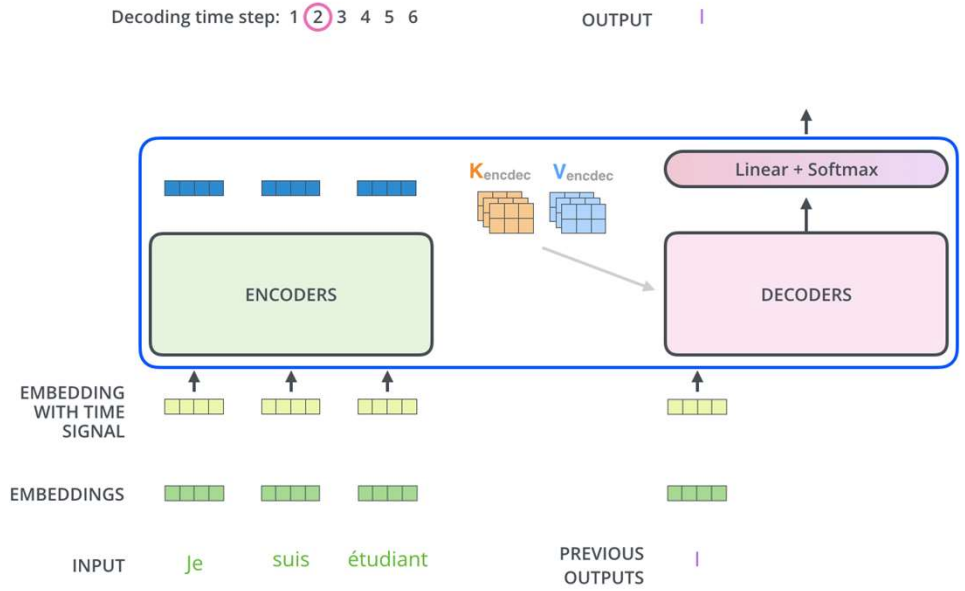
At first glance, this trigonometric-function design looks weird. Before explanations of this design, let us first implement it in the following `PositionalEncoding` class.

Besides capturing absolute positional information, the above positional encoding also allows a model to easily learn to attend by relative positions

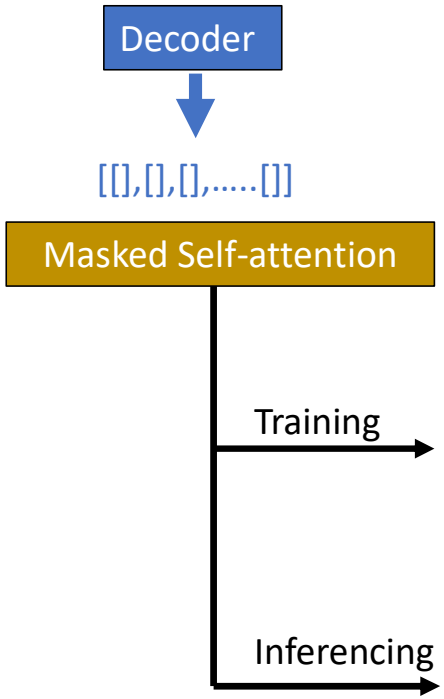
Transformer: positional encoding




Decoding time step: 1 2 3 4 5 6
OUTPUT
Linear + Softmax
ENCODER
ENCODER
DECODER
DECODER
EMBEDDING WITH TIME SIGNAL
EMBEDDINGS
INPUT
Je
suis
étudiant


Decoding time step: 1 2 3 4 5 6
OUTPUT
I
Kencdec
Vencdec
Linear + Softmax
ENCODERS
DECODERS
EMBEDDING WITH TIME SIGNAL
EMBEDDINGS
INPUT
Je
suis
étudiant
PREVIOUS OUTPUTS
I

Masked Multi headed self-attention



Causal mask (autoregressive mask)

- **Use:** Autoregressive models (Sequential prediction)→ One token at a time.
- **Need**: Capture rich contextual information by mapping dependencies between past and currently predicted tokens
- **Work**: Similar to Self-attention with masking to tokens

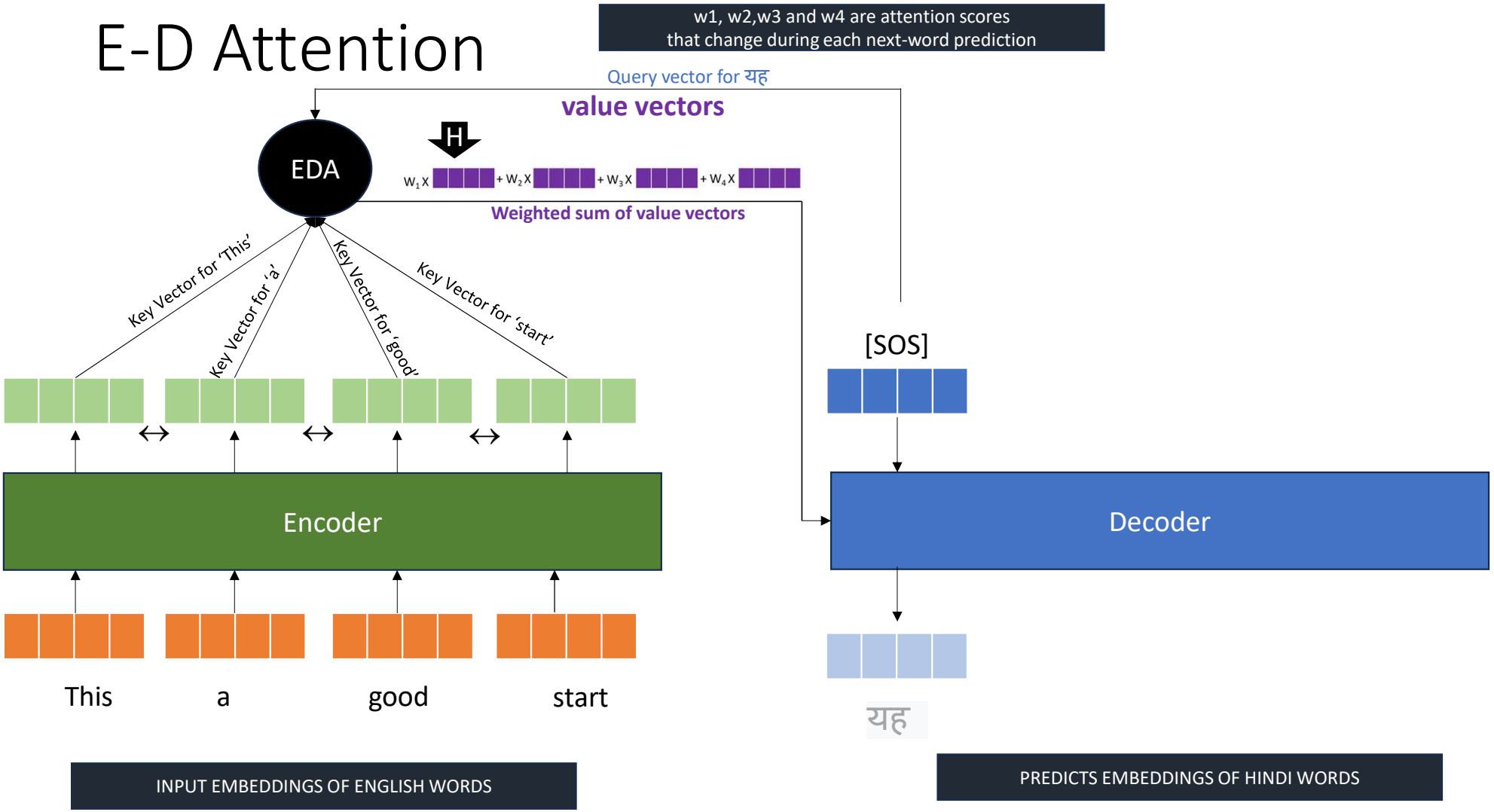
- The causal mask is applied to ensure that each token can only "see" the previous tokens since the entire sequence is available during training, This keeps the model from cheating
- Masking is turned off in the last layer. The other layers still see masking to maintain consistency between training and inferencing

Masked Multi headed self-attention

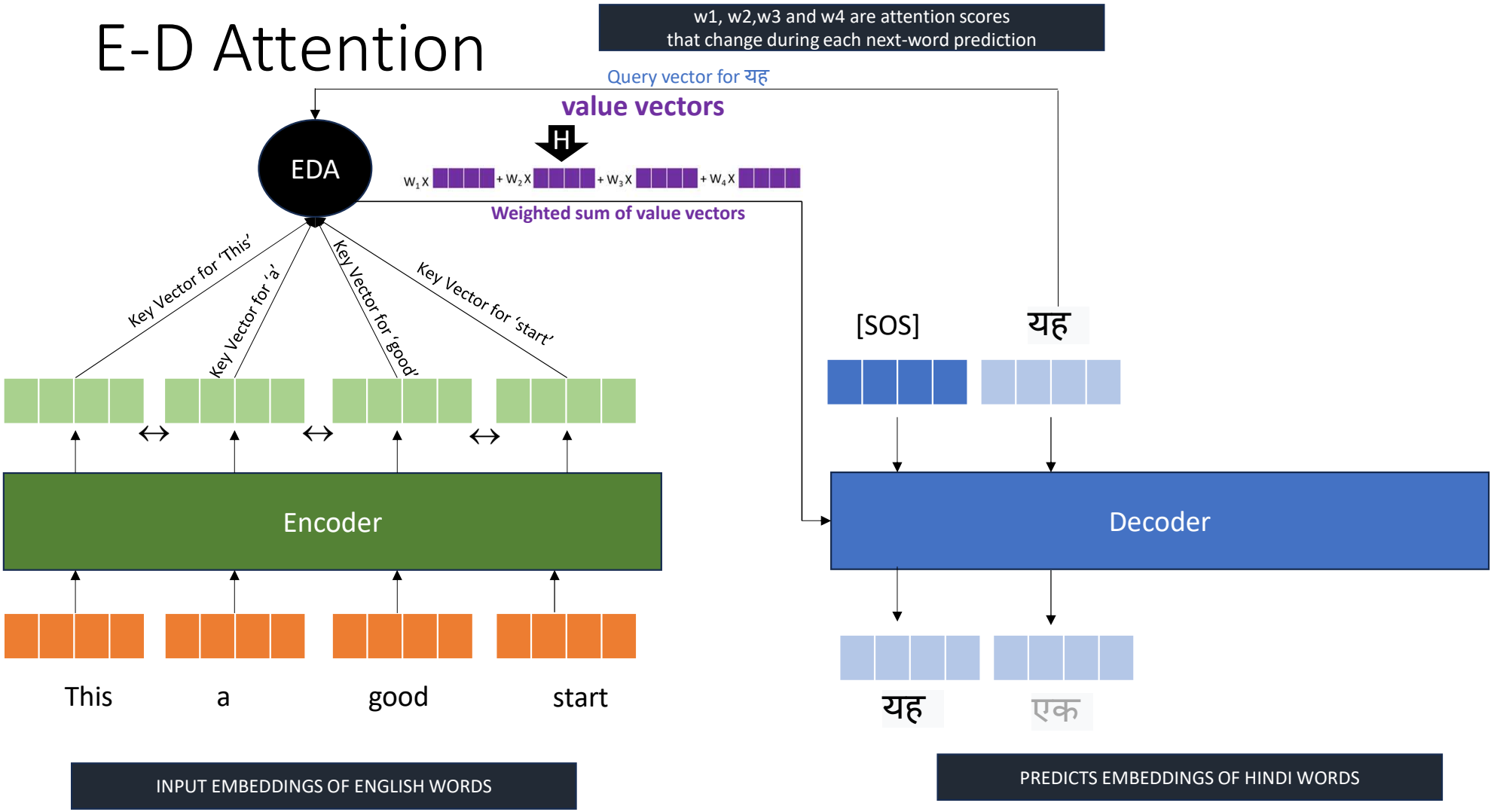
Applying mask:

“It is going to rain heavily today”	Token to be trained as a response
“It is”	Currently predicted tokens
[x.xx, y.yy, -10^8, -10^8, -10^8, -10^8, -10^8]	Masked matrix for predicting the next word (“going”)→ Q.K output
[0.65, 0.35, 0, 0, 0, 0, 0]	Attention scores

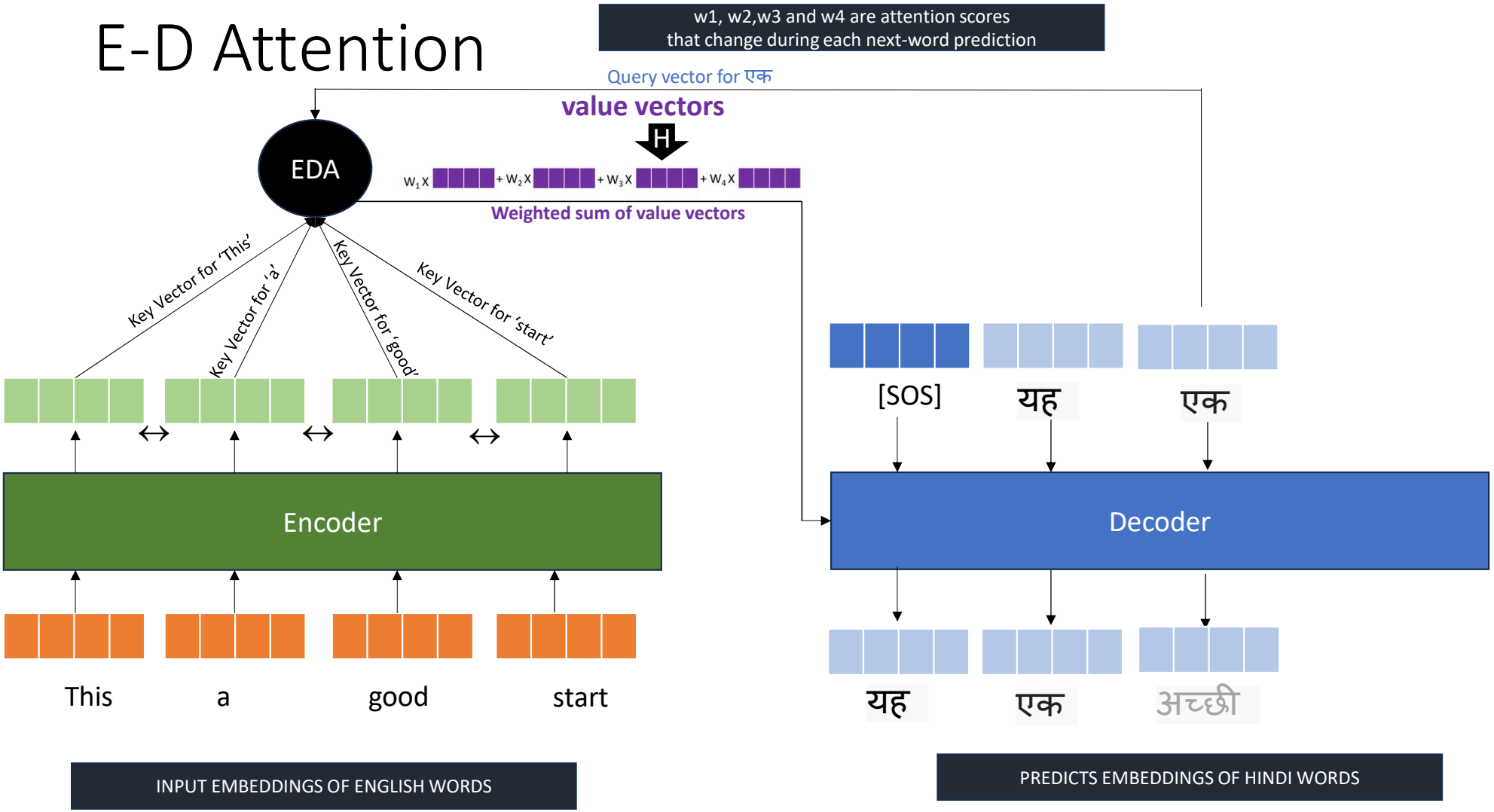
E-D Attention



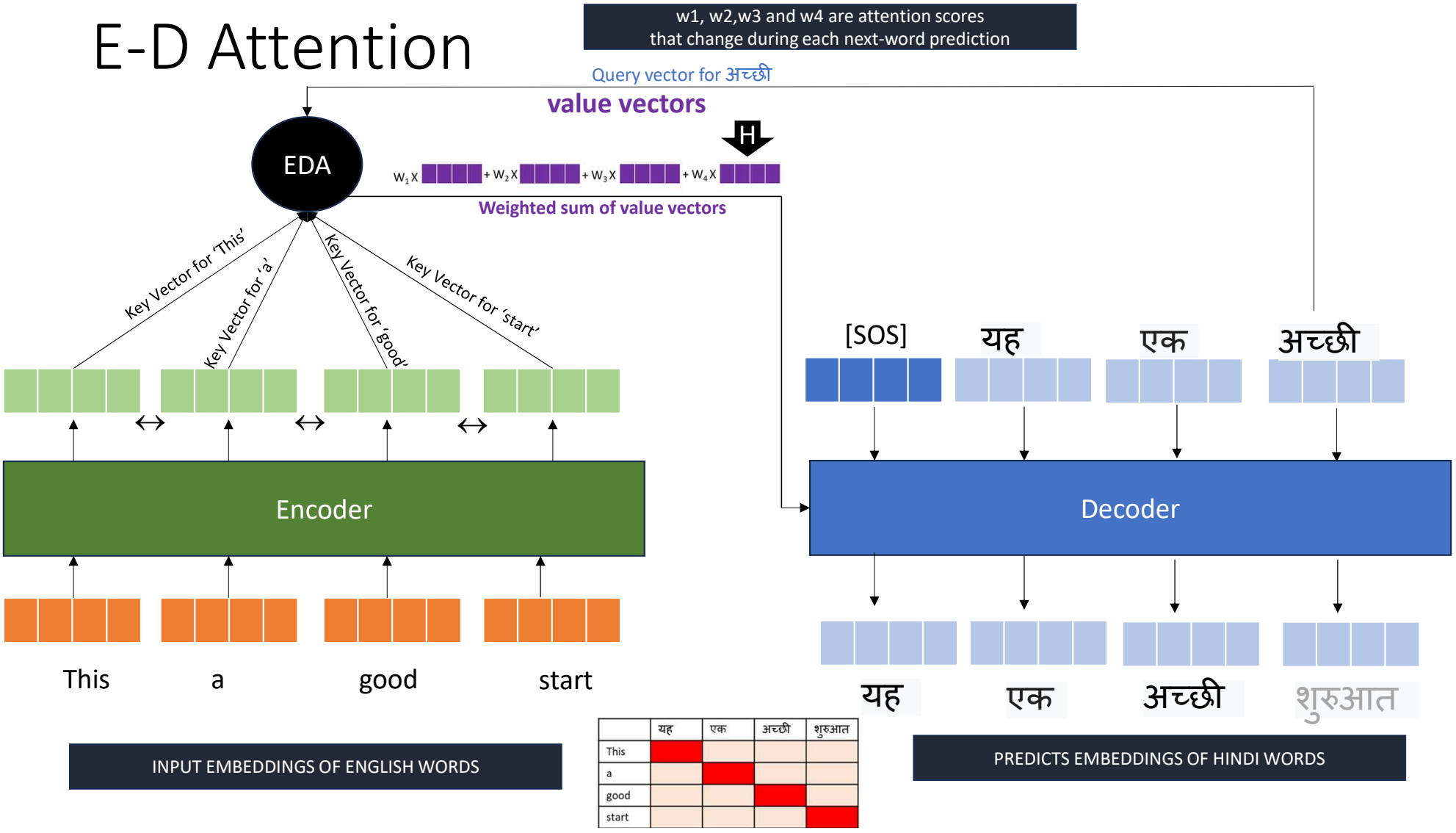
E-D Attention



E-D Attention



E-D Attention



	यह	एक	अच्छी	शुरुआत
This				
a				
good				
start				

BERT: Just an intro

- Bidirectional Encoder Representations from Transformers

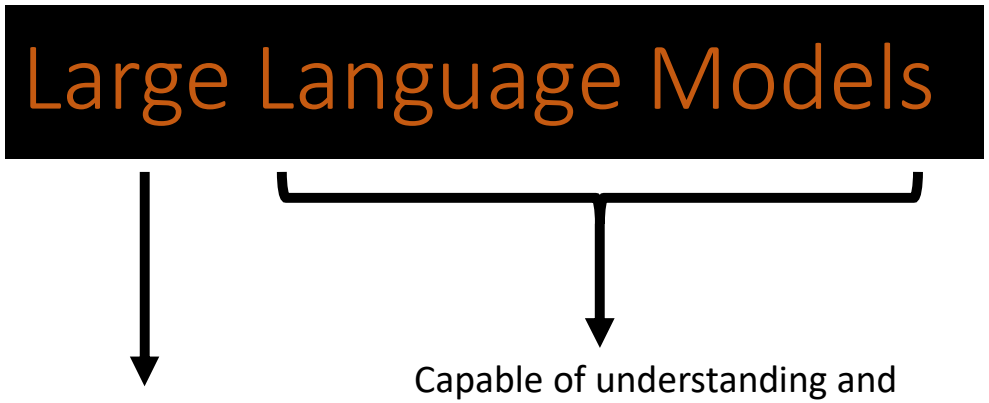
- BERT architecture is a multi-layer bidirectional Transformer encoder. We have two versions of BERT: BERT base and BERT large.
- BERT base has 12 Encoders with 12 bidirectional self-attention heads and 110 million parameters
- BERT large has 24 Encoders with 24 bidirectional self-attention heads and 340 million parameters

- When training the BERT model, Masked LM and Next Sentence Prediction are trained together, with the goal of minimizing the combined loss function of the two strategies

- Masked LM (MLM): Before feeding word sequences into BERT, 15% of the words in each sequence are replaced with a [MASK] token. The model then attempts to predict the original value of the masked words, based on the context provided by the other, non-masked, words in the sequence

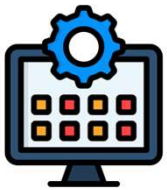
- Next Sentence Prediction(NSP): During training, 50% of the inputs are a pair in which the second sentence is the subsequent sentence in the original document, while in the other 50% a random sentence from the corpus is chosen as the second sentence. The assumption is that the random sentence will be disconnected from the first sentence.

Large Language Models



- Pre-trained on a large corpus of text data
- Learn the complex patterns and rules of human language

Capable of understanding and generating human language



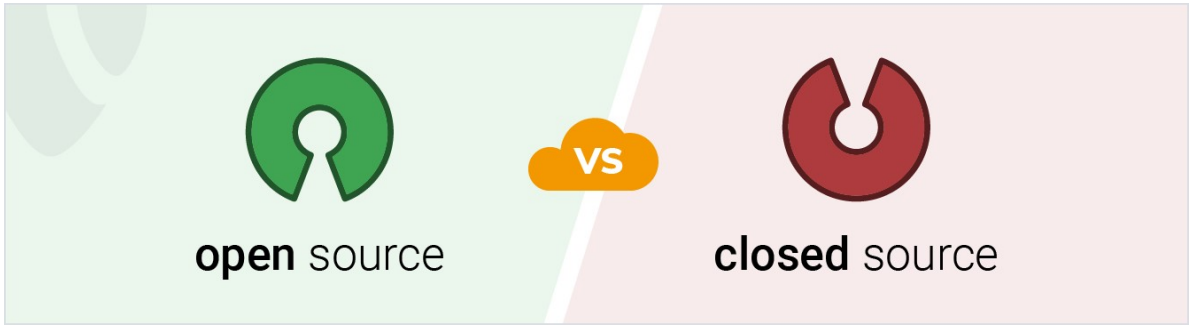
task automation, customer service chatbots, subject research, document summarization and content generation (text, images, audio, video, code)

Open-source proprietary

pricing, parameter size, context window, customization options, deployability

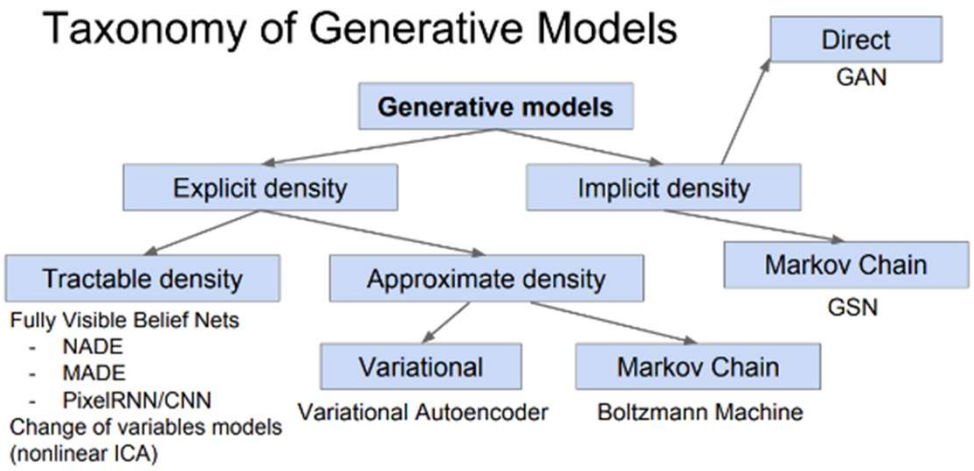
Web Interface, API Interface, Third-Party Platforms

Popular LLMs

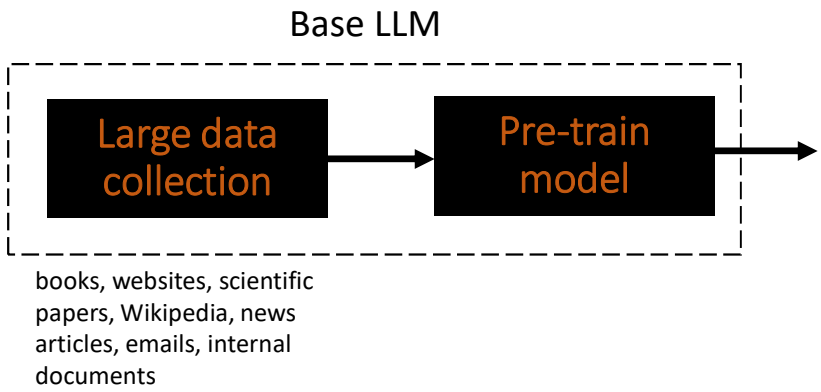


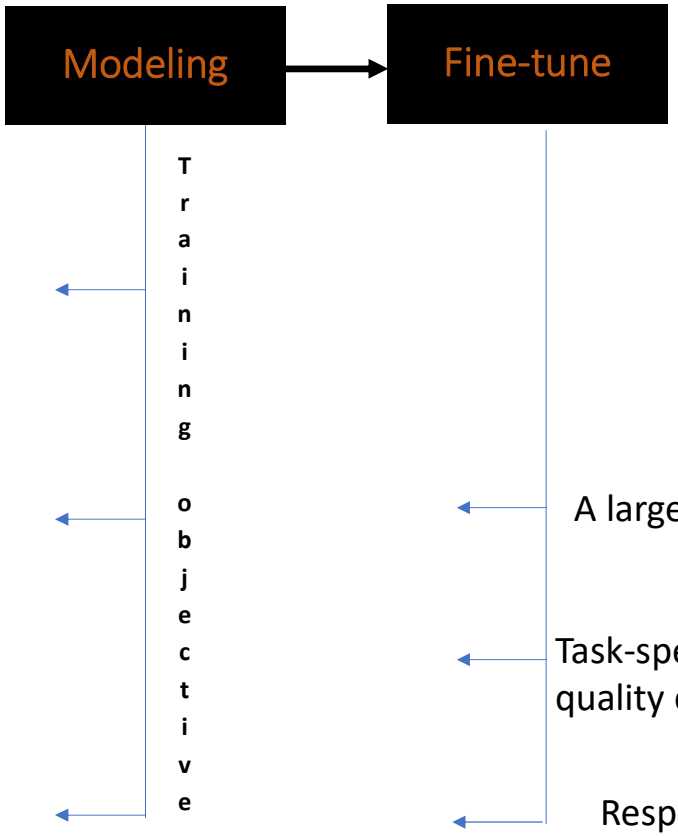
1.GPT-NeoX
2.BLOOM
3.LLaMA 2
4.BERT
5.XGen-7B
6.Falcon-180B
7.Vicuna-33B
8.Dolly 2.0
9.CodeGen
10.latypus 2

LLM Evaluation


Taxonomy of Generative Models
Generative models
Direct
GAN
Explicit density
Implicit density
Tractable density
Approximate density
Markov Chain
GSN
Fully Visible Belief Nets
- NADE
- MADE
- PixelRNN/CNN
Change of variables models
(nonlinear ICA)
Variational
Variational Autoencoder
Markov Chain
Boltzmann Machine

Training of Large Language Models





Autoregressive Modeling (Models like GPT):
Train the model to predict the next token given the previous tokens.

Masked Language Modeling (BERT-like Models):
Randomly mask some tokens in the input and train the model to predict the masked tokens. This forces the model to learn the context and relationships between words.

"Sequence-to-Sequence Modeling (T5, BART):
Encode the input sequence and then generate a target sequence, useful for translation, summarization, or text-to-text tasks.

https://towardsdatascience.com/transformers-141e32e69591

http://jalammar.github.io/illustrated-transformer/