Deep Learning in Natural Language Processing

Kietikul Jearanaitanakij

Department of Computer Engineering, KMITL

Representing Word with Numeric Values

1. **One-hot encoding**

Let n be the number of words in dictionary.

We use the n-dimension one-hot vector to represent each word.

Example:

Dict = { apple, orange, mango, durian }

apple = < 1, 0, 0, 0 >

orange = < 0, 1, 0, 0 >

mango = < 0, 0, 1, 0 >

durian = < 0, 0, 0, 1 >

Representing Word with Numeric Values

2. TF-IDF encoding

TF – Term frequency

$$\frac{No.\,of\ times\ the\ term\ occurs\ in\ the\ document}{Total\ No.\,of\ terms\ in\ the\ document}$$

IDF – Inverse Documet Frequency

$$\log\left(\frac{Total\ No.\,of\ documents}{No.of\ documents\ that\ a\ term\ occurs}\right)$$

Example: sentence1 = < I like orange >
sentence2 = < I like banana >
sentence3 = < I don’t know fresh figs >

Let sentence = document, word = term.

In sentence1, TF(I) = TF(like) = TF(orange) = 1/3 = 0.33

In sentence2, TF(I) = TF(like) = TF(banana) = 1/3 = 0.33

In sentence3, TF(I) = TF(don’t) = TF(know) = TF(fresh) = TF(figs) =1/5 = 0.2

IDF(I) = log(3/3) = 0 ; So, ‘I’ is not important term. IDF(like) = log(3/2) = 0.17

IDF(orange) = log(3/1) = log(3) = 0.47 ; Important term!
= IDF(banana),IDF(don’t),IDF(know),IDF(fresh),IDF(figs)

Representing Word with Numeric Values

TF-IDF encoding (cont.)

sentence1 = < I like orange > , TF of each word = 0.33

sentence2 = < I like banana >, TF of each word = 0.33

sentence3 = < I don’t know Figs and Plum>, TF of each word = 0.2

Note: *TF of each word in a sentence is identical by incidental.*

	banana	don’t	figs	fresh	know	like	orange
IDF	0.47	0.47	0.47	0.47	0.47	0.17	0.47

Note: *IDF of each word has only a single value. (‘I’ was removed since its IDF=0)*

Word positions in vector (sorted): [**banana**, **don’t, figs, fresh, know, like**, **orange**]

Word vector for each sentence is given below:

Sentence1: [**0**, **0, 0, 0, 0**, **0.33*0.17**, **0.33*0.47**]

Sentence2: [**0.33*0.47**, **0, 0, 0, 0, 0**, **0.33*0.17**, **0**]

Sentence3: [**0**, **0.17*0.47, 0.17*0.47, 0.17*0.47, 0.17*0.47,** **0**, **0**]

Warning! Result from Scikit-learn may different because it applies some advanced techniques, e.g. smoothing, log base e.

Representing Word with Numeric Values

3. Word Vector

- The most popular word embedding.
- Unlike one-hot and tf-idf encodings, word vector can capture the meaning, semantic similarity, and relationship of a word and its surrounding words.
- Each word is encoded as a multi-dimensional vector.

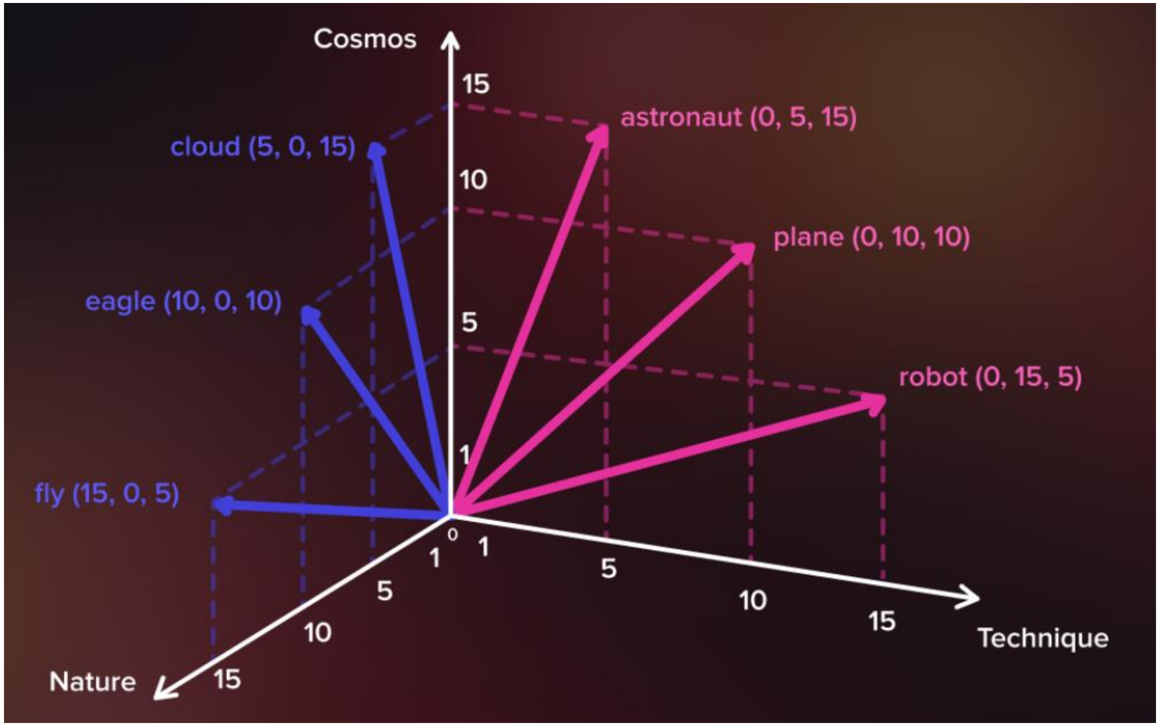


Figure credit: https://serokell.io/blog/word2vec

Representing Word with Numeric Values

- Word2Vec uses a **shallow neural network** (have one or two hidden layers between the input and output) to learn the meaning of words from a large corpus of texts.
- The shallow neural network of Word2Vec can quickly recognize semantic similarities and identify synonymous words using **logistic regression classifier**.
- To represent a particular word as an input vector in multidimensional space, simply use a bag-of-word encoding.
- The training algorithm is categorized into two modes: **continuous bag of words (CBOW)** or **skip-gram**.

Representing Word with Numeric Values

- **Continuous bag of words (CBOW)** => **Tom __?__ Goong**

  Predict the target word ‘Yum’ from the context words (Tom, Goong)

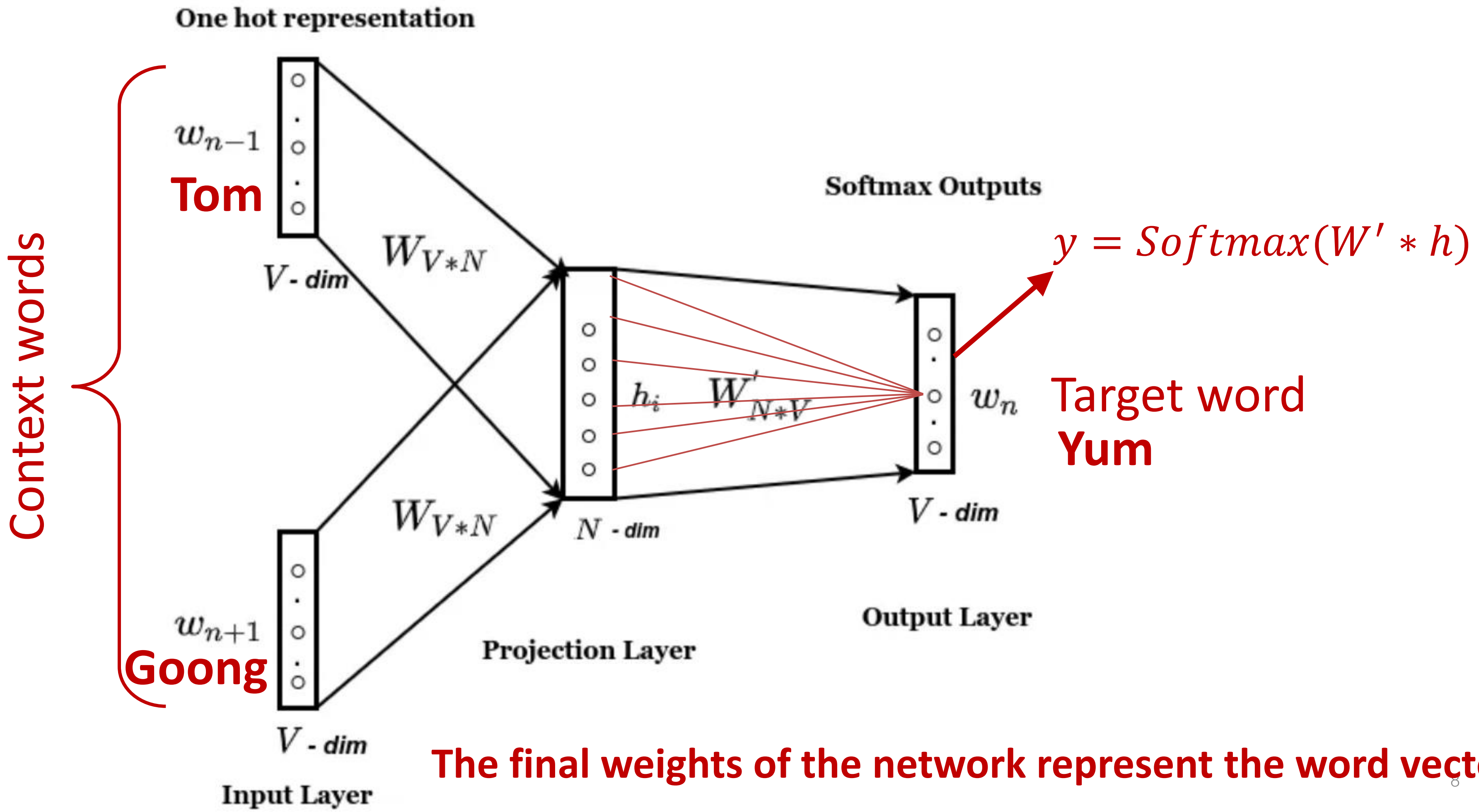
  - The **input layer** is represented by **a one-hot encoded vector**, where each element in the vector corresponds to a specific word in the vocabulary. For example, if the vocabulary contains 10,000 words, the input layer will have 10,000 elements.
  - The **hidden layer** is where the word embeddings are learned. The number of neurons in the hidden layer can be varied depending upon the dimension of the word vector.
  - The **output layer** is also a dense layer, with each neuron representing a specific word in the vocabulary. The number of neurons in the output layer is the same as the number of words in the vocabulary.

Reference: https://medium.com/@codethulo/understanding-the-continuous-bag-of-words-cbow-model-architecture-working-mechanism-and-math-78c7284a8d5a

Representing Word with Numeric Values

- Continuous bag of words (CBOW)

Tom __?__ Goong

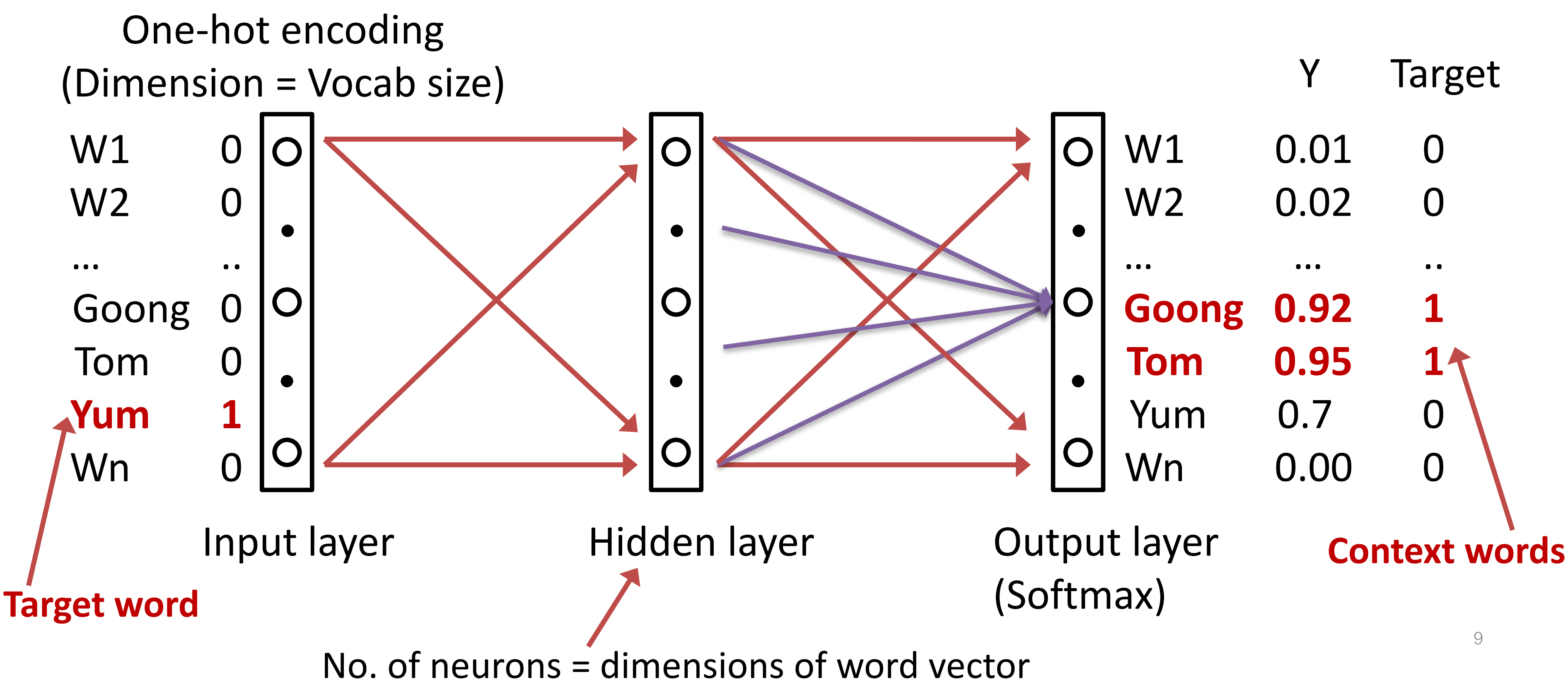


The final weights of the network represent the word vectors.

Representing Word with Numeric Values

- **Skip-gram** => ___?___ **Yum** ___?___.

Predict the context words (‘Tom’, ‘Goong’) from the given word ‘Yum’.



Representing Word with Numeric Values

According to the original paper, Mikolov et al., it is found that

- **Skip-Gram** works well with small datasets, and can better represent less frequent words (e.g. pulchritudinous).

- **CBOW** is found to train faster than Skip-Gram, and can better represent more frequent words (e.g. beautiful, nice).

Representing Word with Numeric Values

Interesting facts about Word2Vec

- Word vectors are meaningful as you can find their relationships.

  Vector(King) – Vector(Man) + Vector(Woman) = Vector(Queen)
- **Gensim**[1] provides an implementation of **Word2Vec** that you can train the model from scratch.
- PyThaiNLP[2] provides **Thai2Vec** which contains 51,556 Thai word embeddings each of 300 dimensions.
- Another well-known word encoding is **GloVe** (Global Vectors for Word Representation) from Stanford University which was trained on different method but give similar results.

1. https://radimrehurek.com/gensim/models/word2vec.html
2. https://pythainlp.github.io/tutorials/notebooks/word2vec_examples.html

Representing Sentence with Numeric Values

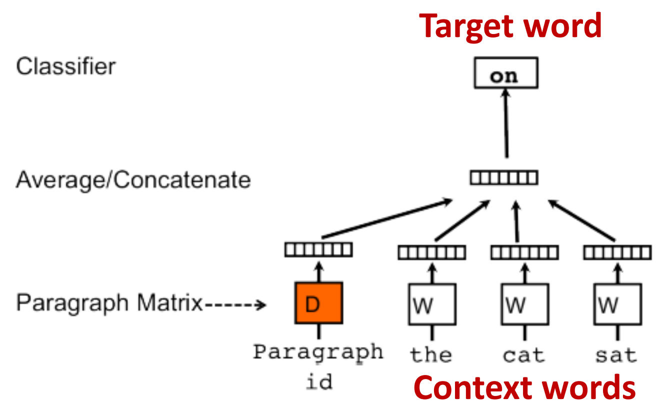
- The simple strategy to find the sentence vector is to **average all vectors of words** in that sentence.
- Although this simple averaging works in most task, it performs poorly for sentiment analysis tasks, because it "loses the word order in the same way as the standard bag-of-words models do“.
- Other advanced methods to find a sentence vector can be found in:
  - https://nlp.stanford.edu/~socherr/EMNLP2013_RNTN.pdf
  - https://arxiv.org/abs/1506.06726
  - https://aclanthology.org/P16-1089.pdf
  - https://arxiv.org/pdf/1908.10084.pdf

Representing Document with Numeric Values

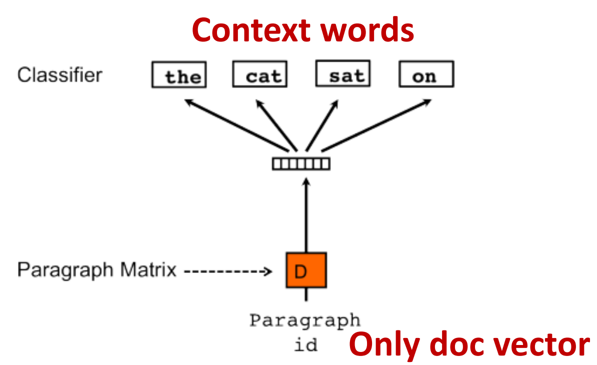
- **Doc2Vec** has a similar training process to Word2Vec. Two Doc2Vec training algorithms: PV-DM and PV-DBOW. Unseen document is inferred its vector by the average of word vectors.

- **Distributed memory (PV-DM)** : instead of using just words to predict the target word, we also added another feature vector, which is document-unique (paragraph id). The target word is predicted. (PV-DM consumes more memory, but has better result than PV-DBOW)

- **Distributed bag of words (PV-DBOW)** : Only the document vector (or paragraph id) is fed into the hidden layer. The context words in the document are predicted. (PV-DBOW is faster than PV-DM and works very well on shortish-docs)
- Once words are represented as vectors of numeric values, we can apply them to any deep learning models.



PV-DM



PV-DBOW

Transformer

- **Transformer** is a deep learning architecture that uses **Attention Mechanism** to significantly improve the performance of NLP translation models.
- It was first introduced in the paper "**Attention is all you need**"[1]
- Numerous projects including Google's BERT and OpenAI's GPT series have built on this foundation
- Transformer takes a text sequence as input and produce another text sequence as output.

Happy birthday → Transformer → **สุขสันต์วันเกิด**

1. https://arxiv.org/abs/1706.03762

Transformer

- It contains a stack of **Encoder layers** and **Decoder layers.**
- Each layer in Encoder and Decoder layers has its own set of weights.
- These weights can learn relationship among each word and other words in the sentence.
- Finally, there is an Output layer to generate the final output.

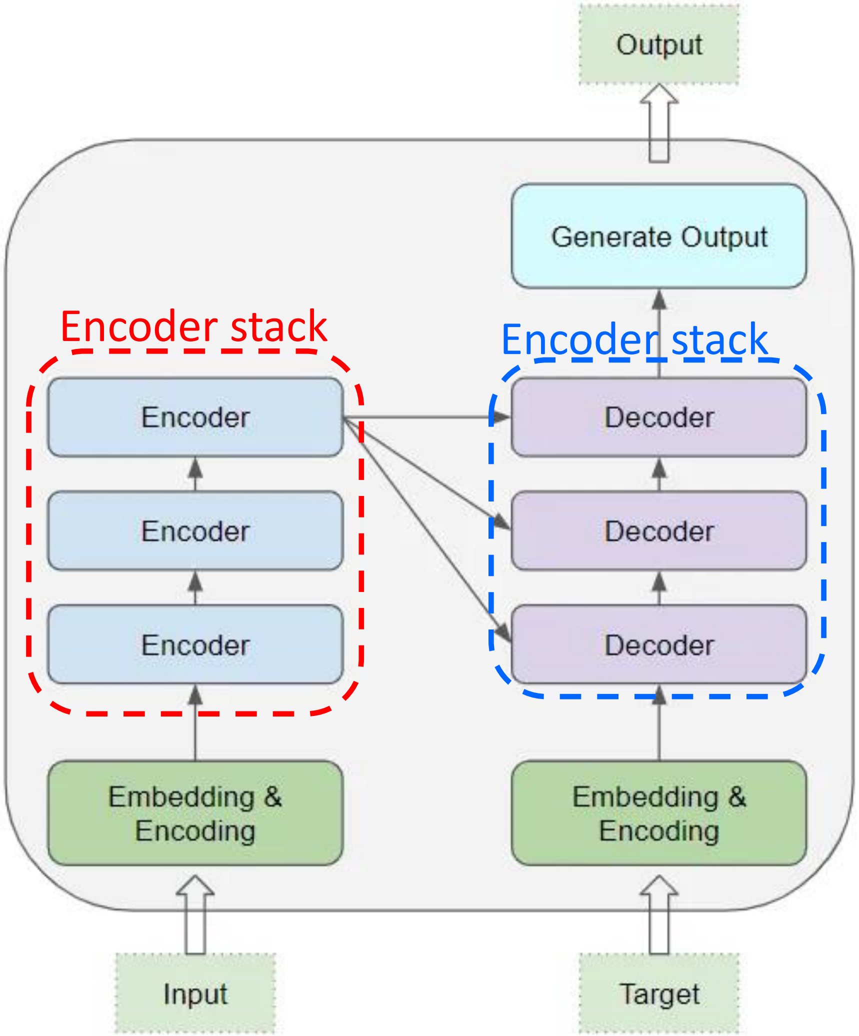
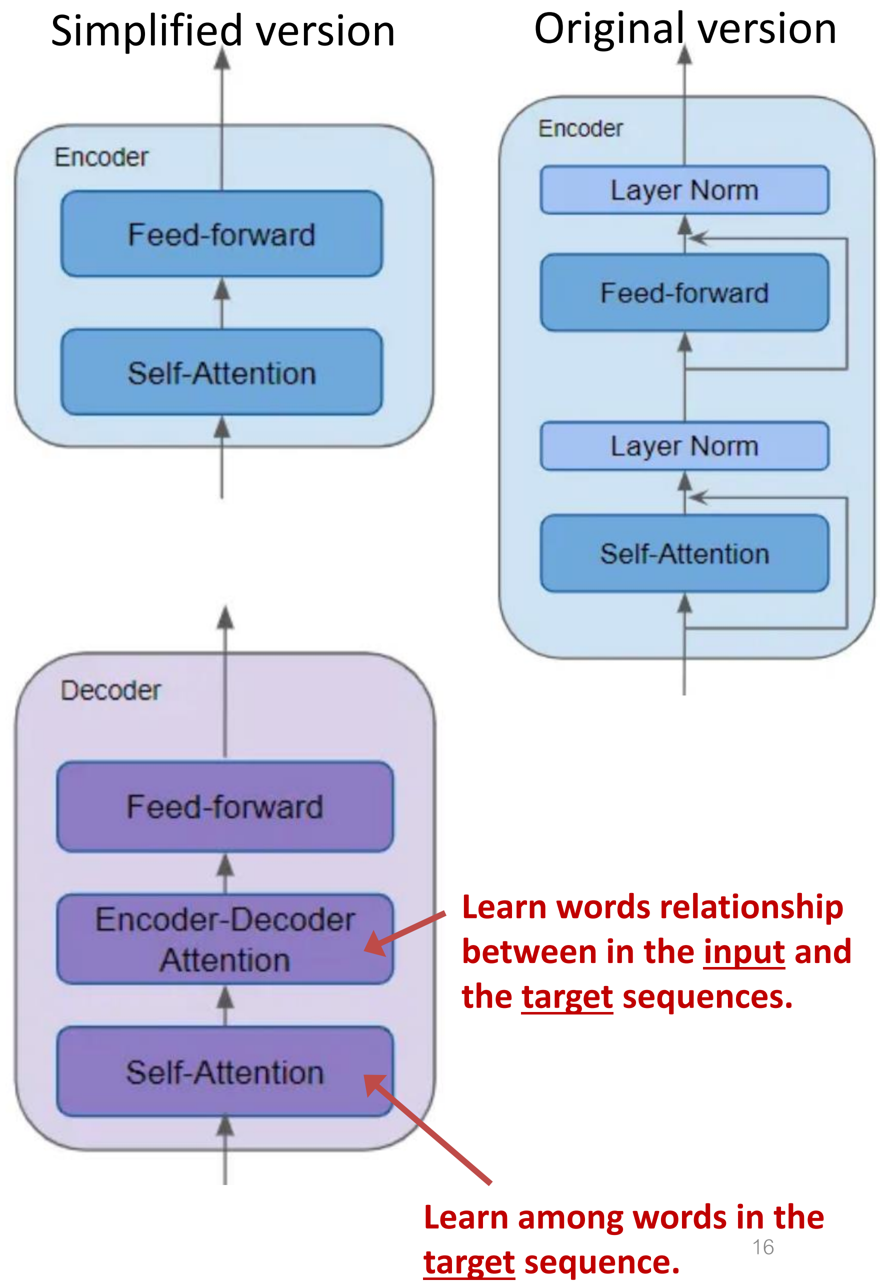


Fig. credit: https://towardsdatascience.com/transformers-explained-visually-part-1-overview-of-functionality-95a6dd460452

Transformer

- The **Encoder** contains the all-important **Self-attention layer** that computes the <u>relationship between different words in the sequence</u>, as well as a Feed-forward layer.

- The **Decoder** contains the <u>Self-attention layer</u> and the <u>Feed-forward layer</u>, as well as a second <u>Encoder-Decoder attention layer</u>.



Transformer

- The most important function in Transformer is its **Attention Mechanism**.

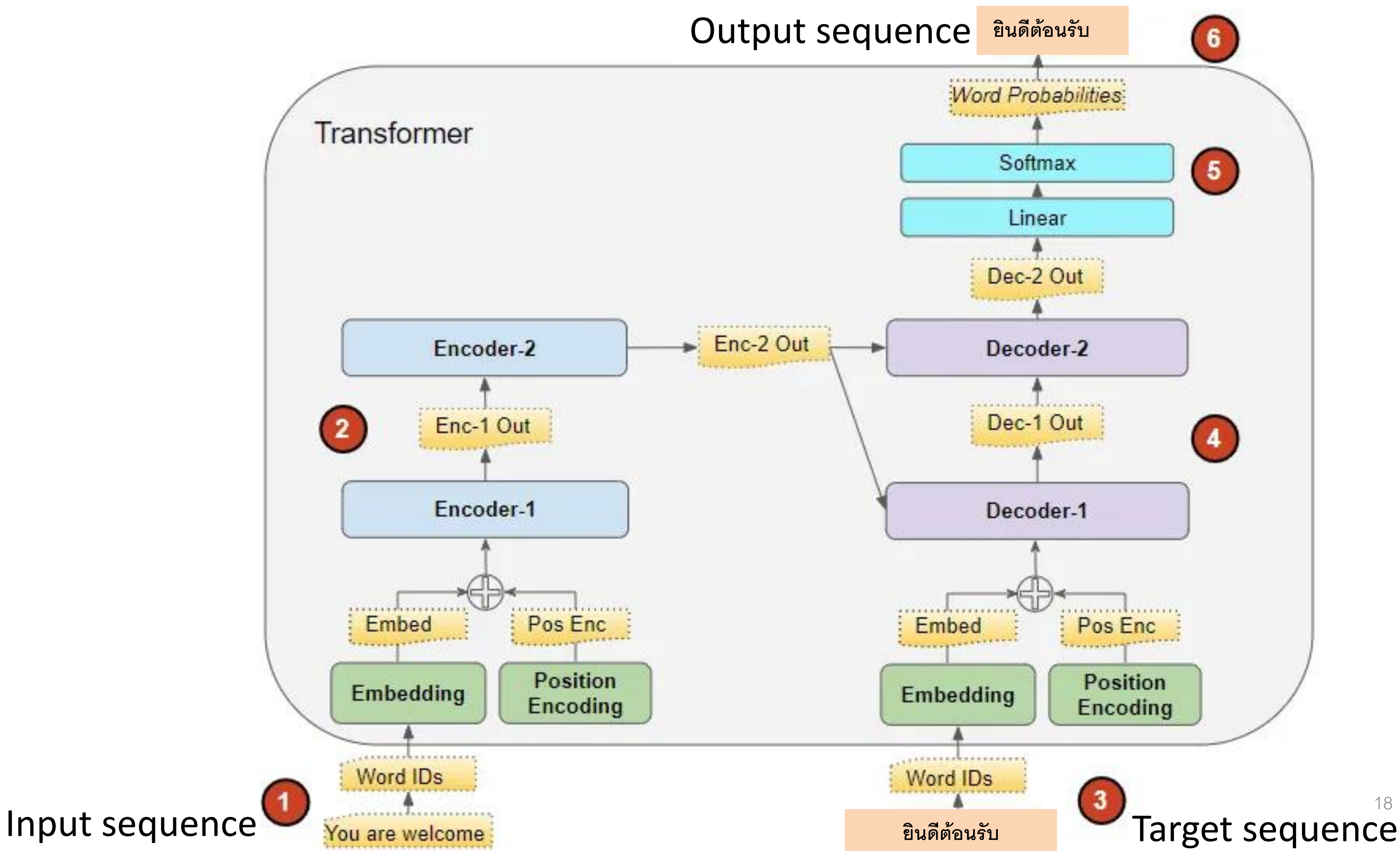
- While processing a word, Attention enables the model to focus on other words in the input that are closely related to that word.

'**Ball**' is closely related to '**blue**' and '**holding**'. On the other hand, 'blue' is not related to 'boy'

- Transformers include multiple attention scores for each word.
- Training data consists of two parts:
  1. The source or input sequence (e.g. "You are welcome" in English, for a translation problem)
  2. The destination or target sequence (e.g. "**ยินดีต้อนรับ**" in Thai)

Transformer

- The Transformer's goal is to learn how to output the target sequence, by using both the input and target sequences.



Transformer

Training steps in Transformer.

1. The **input sequence** is **encoded with the position of each word**. Then the encoded signal is fed to the Encoder.
2. **Encoder stack** processes this signal with **self attention**.
3. The **target sequence** is **prepended** with a start-of-sentence token, converted into target embeddings (with positional encoding), and fed to the Decoder.
4. The **stack of Decoders** processes the target sequence embedding with **self attention**.
5. The outputs from the **Decoder stack** and the **Encoder stack** are brought into the **attention** calculation to find the relationship between them. (Encoder-Decoder attention)
6. The Output layer converts it into word probabilities (softmax function) and the final **output sequence**.
7. Loss value is calculated and the gradient is back-propagated.

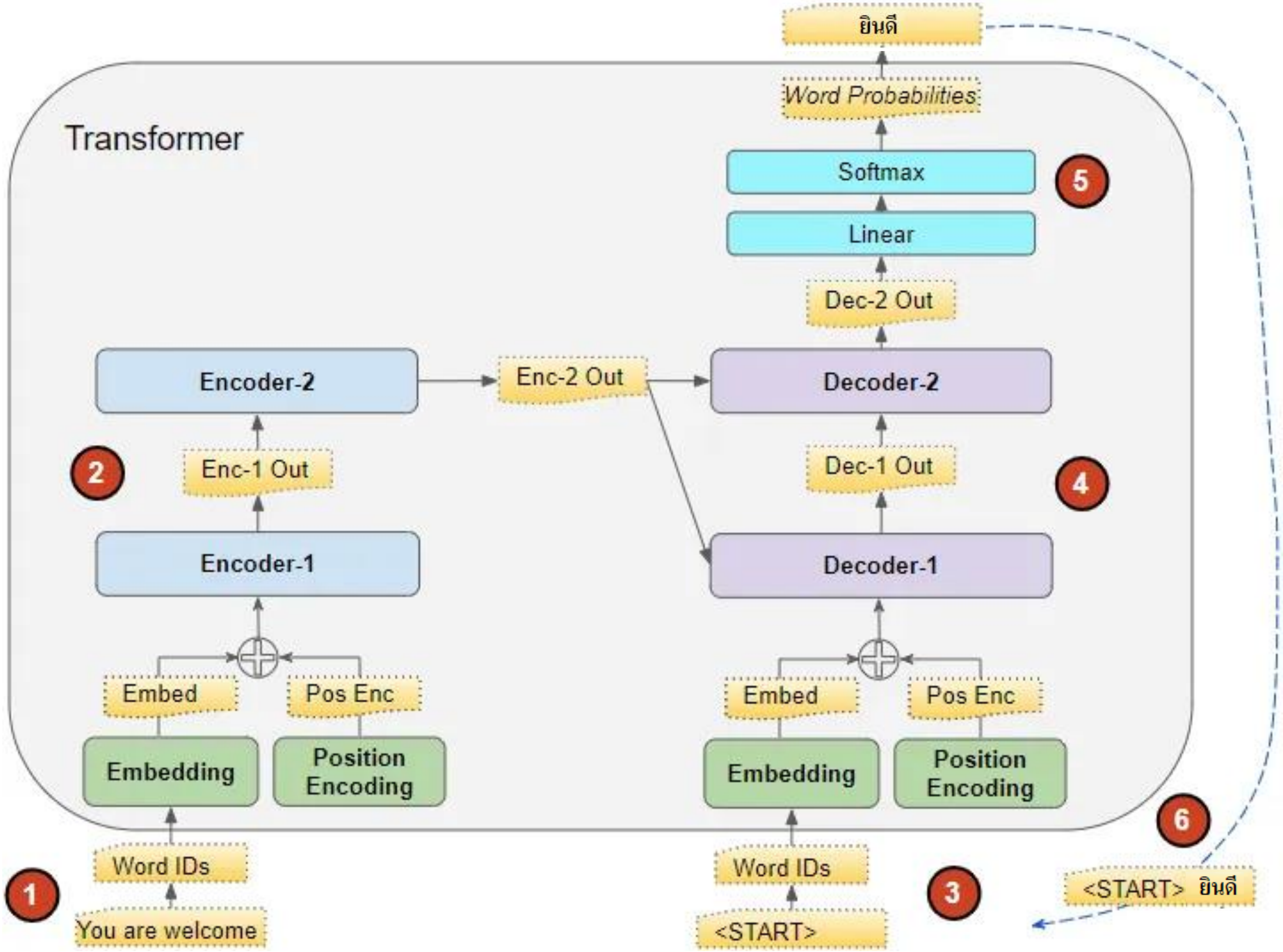
Inference in Transformer

- During **Inference**, we have only the **input sequence** and don't have the target sequence to pass as input to the Decoder.

- The goal of the Transformer is to produce the target sequence (at the output layer) from the input sequence alone.

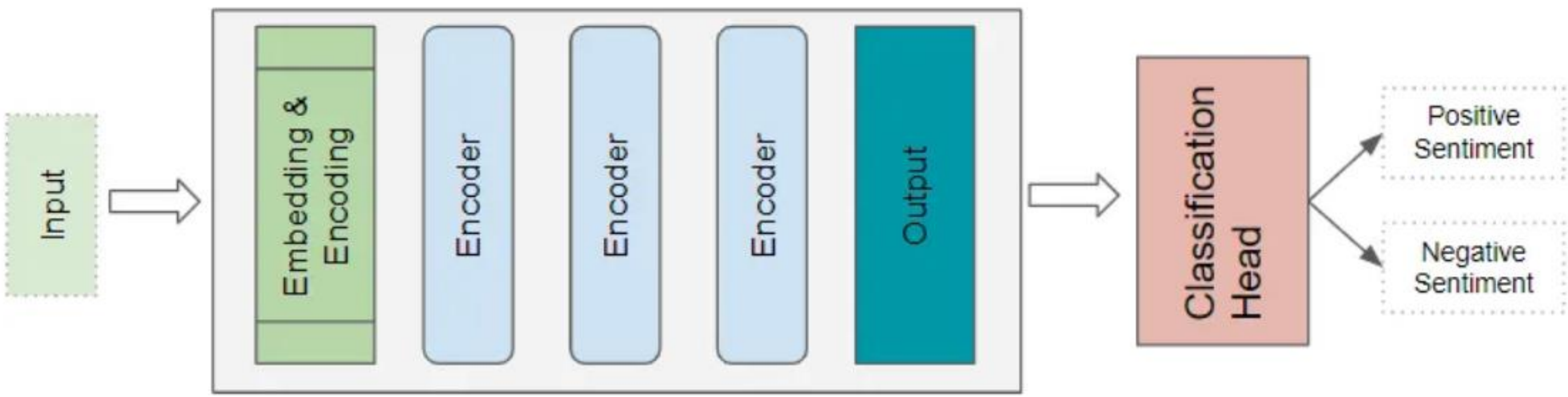
- We **generate the output in a loop :** feed the **entire** output sequence from the previous timestep to the Decoder in the next timestep until we reach an end-of-sentence token <eos> at the output layer.

Inference in Transformer



Applications of Transformer

- Transformer is frequently used in sequence-to-sequence models for applications such as **Machine Translation**, **Text Summarization**, **Question-Answering**, **Named Entity Recognition**, and **Speech Recognition**.
- The basic Encoder Layer is used as a common building block with different application-specific '**heads**' depending on the problem being solved.



Sample application of transformer: Sentiment analysis

Large Language Model (LLM)

- **LLM** is a very large deep learning models that are pre-trained on vast amounts of data.
- LLM can perform completely different tasks such as answering questions, summarizing documents, translating languages, sentiment analysis, and text generation, etc.
- Once trained, LLMs can be finetuned to perform specific task by using small data.

Three common learning models exist:

- Zero-shot learning; Base LLMs can respond to a broad range of requests without explicit training or prompts.

  Ex: Query “What is the capital of France?”, Answer: “Paris”
- Few-shot learning: By providing a few relevant prompt examples, base model performance significantly improves in that specific area.

  Ex: “France: Paris, Germany: Berlin, Canada: Ottawa”,   Query: “Thailand”
- Fine-tuning: This is an extension of few-shot learning in that we train a base model with a specific task’s dataset.