Introduction to Transformers
Yogesh Kulkarni

Background

Background
till word2vec

In search of meaning . . .

Meaning, meaning?

Definition: meaning (Webster dictionary)

- the idea that is represented by a word, phrase, etc.
- the idea that a person wants to express by using words, signs, etc.
- the idea that is expressed in a work of writing, art, etc.

In a computer?

How do we represent the meaning of a word in a computer?

Common solution: Use e.g. WordNet, a thesaurus containing lists of **synonym sets** and **hypernyms** ("is a" relationships).

e.g. synonym sets containing "good":

```
from nltk.corpus import wordnet as wn
poses = { 'n':'noun', 'v':'verb', 's':'adj (s)', 'a':'adj', 'r':'adv'}
for synset in wn.synsets("good"):
    print("{}: {}".format(poses[synset.pos()],
        ", ".join([l.name() for l in synset.lemmas()])))
```

```
noun: good
noun: good, goodness
noun: good, goodness
noun: commodity, trade_good, good
adj: good
adj (sat): full, good  adj:
good
adj (sat): estimable, good, honorable, respectable  adj (sat):
beneficial, good
adj (sat): good
adj (sat): good, just, upright
...
adverb: well, good
adverb: thoroughly, soundly, good
```

e.g. hypernyms of "panda":

```
from nltk.corpus import wordnet as wn
panda = wn.synset("panda.01")  hyper =
lambda s: s.hypernyms()
list(panda.closure(hyper))
```

```
[Synset('procyonid.n.01'),
Synset('carnivore.n.01'),
Synset('placental.n.01'),
Synset('mammal.n.01'),
Synset('vertebrate.n.01'),
Synset('chordate.n.01'),
Synset('animal.n.01'),
Synset('organism.n.01'),
Synset('living_thing.n.01'),
Synset('whole.n.02'),
Synset('object.n.01'),
Synset('physical_entity.n.01'),
Synset('entity.n.01')]
```

Problems of WordNet

- Great as a resource but missing nuance: e.g. "proficient" is listed as a synonym for "good". This is only correct in some contexts.
- Missing new meanings of words: e.g., wicked, badass, nifty, wizard, genius, ninja, bombest. Impossible to keep up-to-date!
- Subjective
- Requires human labor to create and adapt
- Can't compute accurate word similarity

Discrete Symbols

Representing words as discrete symbols

In traditional NLP, we regard words as discrete symbols:
hotel, conference, motel - a localist representation

Means one 1, the rest 0s

Words can be represented by one-hot vectors:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

Vector dimension = number of words in vocab (e.g. 500,000)

Discrete Symbols

Example: in web search, if user searches for "Seattle motel", we would like to match documents containing "Seattle hotel".

But:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

These two vectors are orthogonal.
There is no natural notion of **similarity** for one-hot vectors!

Solution:

- Could try to rely on WordNet's list of synonyms to get similarity?
  - But it is well-known to fail badly: incompleteness, etc.
- **Instead: learn to encode similarity in the vectors themselves**

Context

Representing words by their context

- Distributional semantics: **A word's meaning is given by the words that frequently appear close-by**
  - *"You shall know a word by the company it keeps"* (J. R. Firth 1957)
  - One of the most successful ideas of modern statistical NLP!
- When a word *w* appears in a text, its **context** is the set of words that appear nearby (within a fixed-size window).
- Use the many contexts of *w* to build up a representation of *w*

…government debt problems turning into ***banking*** *crises as happened in 2009…*
…saying that Europe needs unified ***banking*** *regulation to replace the hodgepodge…*
…India has just given its ***banking*** *system a shot in the arm…*

These context words will represent ***banking***

(Ref: CS224n: Natural Language Processing with Deep Learning - Christopher Manning)

Fill in the blanks

What can we learn from reconstructing input?

Stanford University is located in −− , California.

(Ref: Language & Machine Learning - John Hewitt)

Fill in the blanks

I went to the ocean to see the fish, turtles, seals, and −−.

Fill in the blanks

Overall, the value I got from the two hours watching it was the sum total of the popcorn and the drink. The movie was −−.

Fill in the blanks

I was thinking about the sequence that goes 1, 1, 2, 3, 5, 8, 13, 21, − − − − − − − − − −.

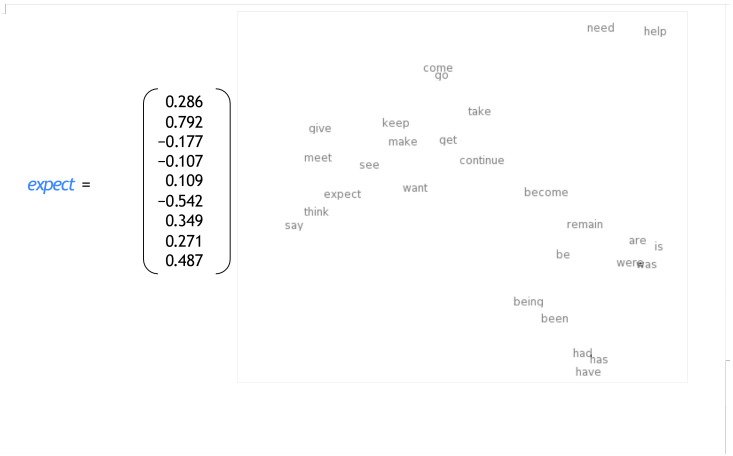
Word vectors

- Dense vector for each word
- Called distributed representation, word embeddings or word representations
- Test: similar to vectors of words that appear in similar contexts

$$\textit{banking} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$$

Visualization

Word meaning as a neural word vector



Word2Vec Overview

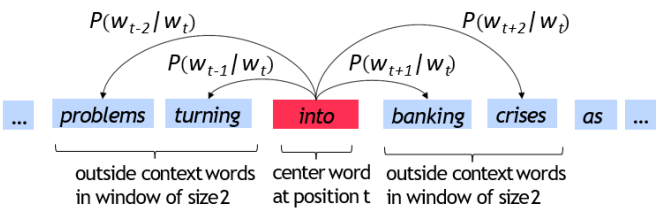
Tim Mikolov et al. 2013

Idea:

- Input: large corpus of text
- Output: Dense, fixed-custom sized vector
- Process (CBOW, Skip-Gram): Go through each word t in the text, thats center word c and context words-window as neighbors as o.
- Use the similarity of the word vectors for c and o to calculate the probability of o given c (or vice versa)
- Keep adjusting the word vectors to maximize this probability

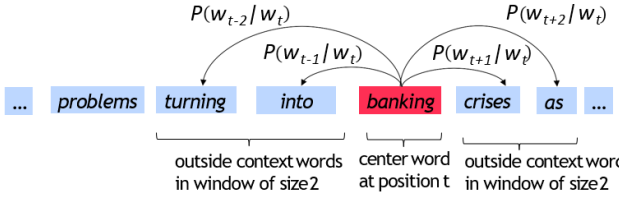
Word2Vec Overview

Example windows and process for computing $P(w_{t+j}|w_t)$



Word2Vec Overview

Example windows and process for computing $P(w_{t+j}|w_t)$



Sequence Recurrence Models for Sentence Vectors RNNs and LSTMs

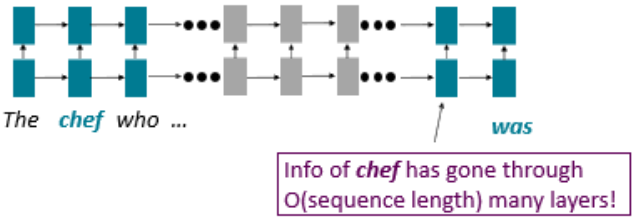
Sequence-to-Sequence (seq2seq)

See any issues with this traditional seq2seq paradigm?

Issues with recurrent models

Linear interaction distance. O(sequence length) steps for distant word pairs to interact means:

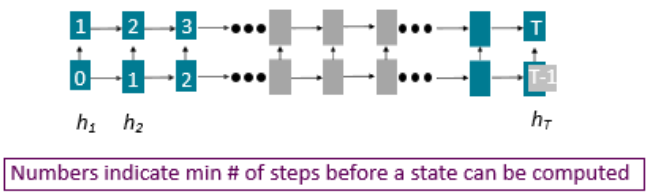
- Hard to learn long-distance dependencies (because gradient problems!)
- Linear order of words is "baked in"; not necessarily the right way to think about sentences . . . Meaning sentence structure of one language may not be correspondingly same as order in the other language.



Issues with recurrent models

Lack of parallelizability. Forward and backward passes have O(sequence length) unparallelizable operations

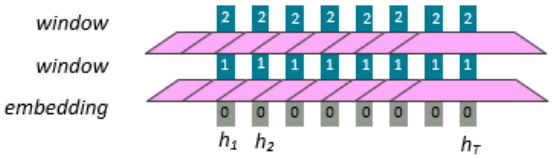
- GPUs can perform a bunch of independent computations at once!
- But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
- Inhibits training on very large datasets!



Then?

If not recurrence, then what? How about word windows? Word window models aggregate local contexts

- Also known as 1D convolution
- Number of unparallelizable operations not tied to sequence length!

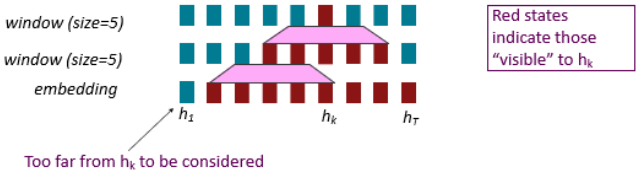


Numbers indicate min # of steps before a state can be computed

Then?

What about long-distance dependencies?

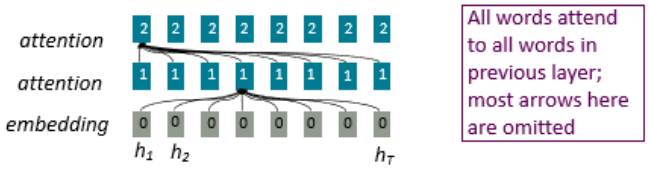
- Stacking word window layers allows interaction between farther words
- But if your sequences are too long, you'll just ignore long-distance context



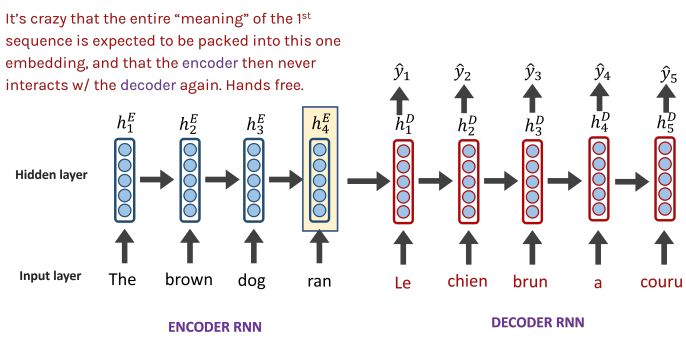
Attention

If not recurrence, then what? How about attention?

- Attention treats each word's representation as a query to access and incorporate information from a set of values.
- We saw attention from the decoder to the encoder; today we'll think about attention within a single sentence.
- If attention gives us access to any state... maybe we can just use attention and don't need the RNN?
- Number of unparallelizable operations not tied to sequence length.
- All words interact at every layer!



Sequence-to-Sequence (seq2seq)

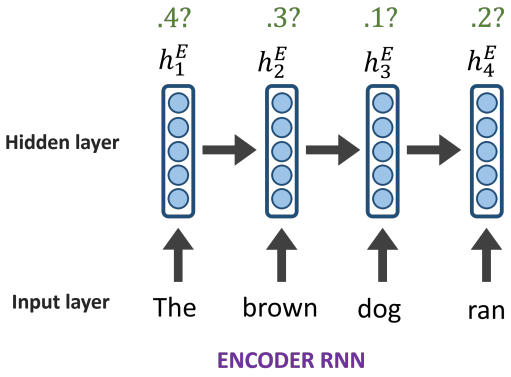


Sequence-to-Sequence (seq2seq)

Instead, what if the decoder, at each step, pays **attention** to a distribution of all of the encoder's hidden states?

seq2seq + Attention

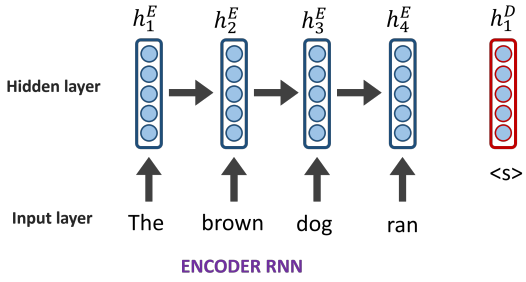
Q: How do we determine how much to pay attention to each of the encoder's hidden layers?



seq2seq + Attention

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

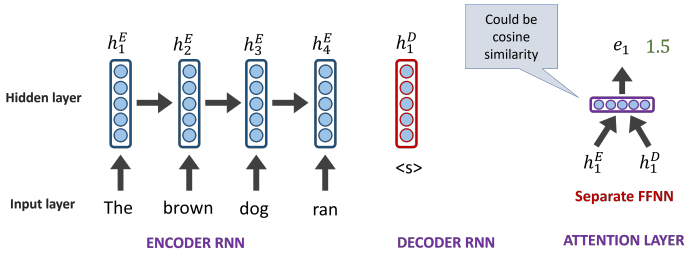
A: Let's base it on our decoder's previous hidden state (our latest representation of meaning) and all of the encoder's hidden layers!



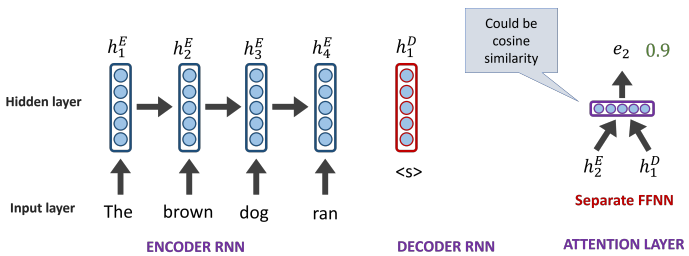
seq2seq + Attention

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

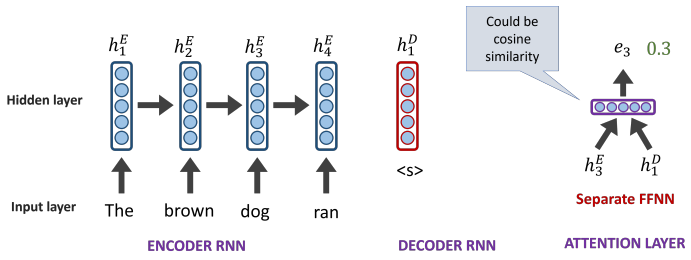
A: Let's base it on our decoder's previous hidden state (our latest representation of meaning) and all of the encoder's hidden layers! We want to measure similarity between decoder hidden state and encoder hidden states in some ways.



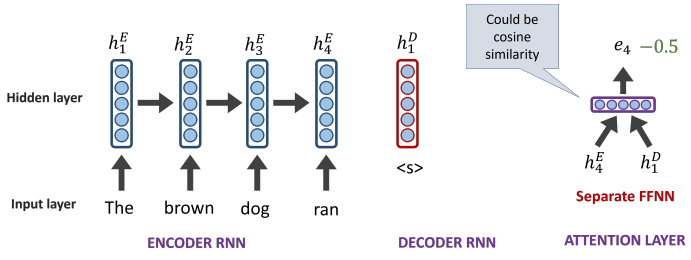
seq2seq + Attention



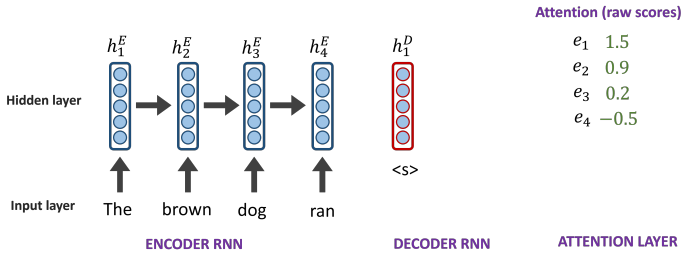
seq2seq + Attention



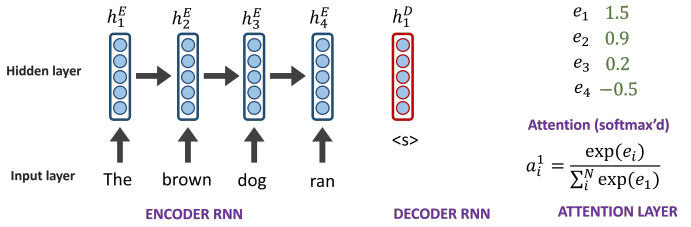
seq2seq + Attention



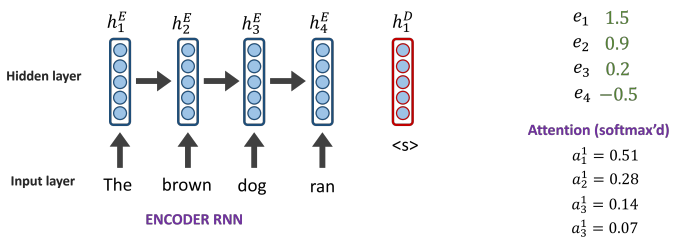
seq2seq + Attention



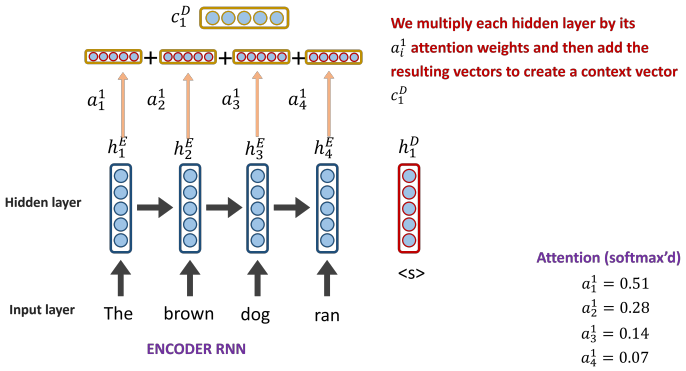
seq2seq + Attention



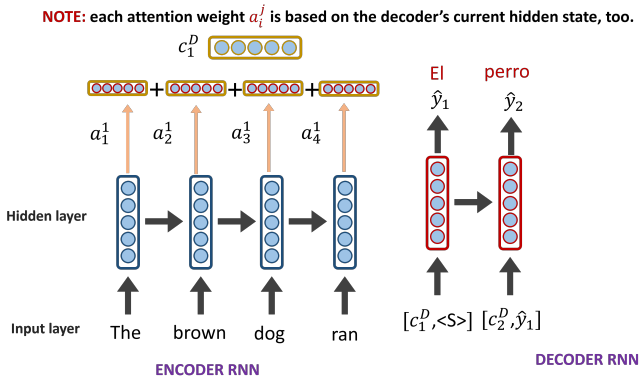
seq2seq + Attention



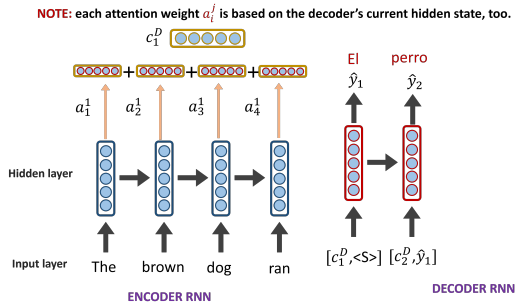
seq2seq + Attention



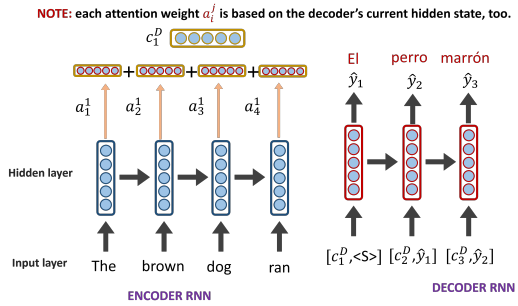
seq2seq + Attention



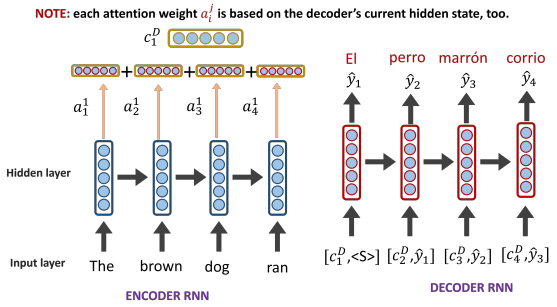
seq2seq + Attention



seq2seq + Attention



seq2seq + Attention



seq2seq + Attention

Attention:

- greatly improves seq2seq results
- allows us to visualize the contribution each word gave during each step of the decoder

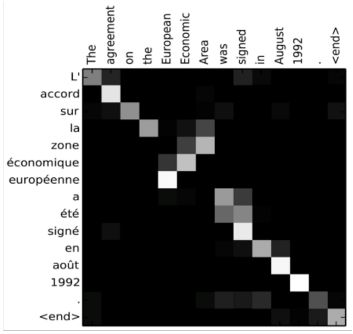


Image source: Fig 3 in Bahdanau et al., 2015

Self-Attention

- Attention operates on **queries**, **keys**, and **values**.
  - We have some **queries** $q_1, q_2, \ldots, q_T$. Each query is $q_i \in \mathbb{R}^d$
  - We have some **keys** $k_1, k_2, \ldots, k_T$. Each key is $k_i \in \mathbb{R}^d$
  - We have some **values** $v_1, v_2, \ldots, v_T$. Each value is $v_i \in \mathbb{R}^d$

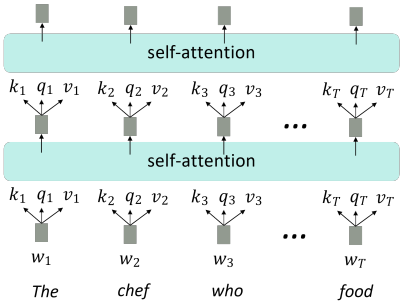
  The number of queries can differ from the number of keys and values in practice.
- In **self-attention**, the queries, keys, and values are drawn from the same source.
  - For example, if the output of the previous layer is $x_1, \ldots, x_T$, (one vec per word) we could let $v_i = k_i = q_i = x_i$ (that is, use the same vectors for all of them!)
- The (dot product) self-attention operation is as follows:

$$e_{ij} = q_i^T k_j \qquad \alpha_i = \frac{\exp(e_{ij})}{\Sigma_{j'} \exp(e_{ij'})} \qquad \text{output}_i = \Sigma_j \alpha_{ij} v_j$$

Compute **key-query** affinities | Compute attention weights from affinities (softmax) | Compute outputs as weighted sum of **values**

Self-Attention

- In the diagram at the right, we have stacked self-attention blocks, like we might stack LSTM layers.
- Can self-attention be a drop-in replacement for recurrence?
- No. It has a few issues, which we'll go through.
- First, self-attention is an operation on **sets**. It has no inherent notion of order.



Self-attention doesn't know the order of its inputs.

Self-Attention

Barriers and solutions for Self-Attention as a building block

Barriers

- Doesn't have an inherent notion of order! →

Solutions

Self-Attention

Fixing the first self-attention problem: Sequence order

- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each **sequence index** as a **vector**

  $p_i \in \mathbb{R}^d$, for $i \in \{1, 2, \ldots, T\}$ are position vectors

- Don't worry about what the p_i are made of yet!
- Easy to incorporate this info into our self-attention block: just add the p_i to our inputs!
- Let v_i', k_i', q_i' be our old values, keys, and queries.

$$v_i = v_i' + p_i$$
$$q_i = q_i' + p_i$$
$$k_i = k_i' + p_i$$

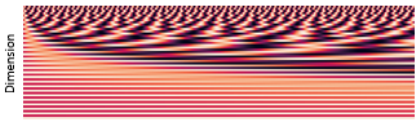
In deep self-attention networks, we do this at the first layer! You could concatenate them as well, but people mostly just add...

Self-Attention

Position representation vectors through sinusoids

- **Sinusoidal position representations:** concatenate sinusoidal functions of varying periods:

$$p_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$



- Pros:
  - Periodicity indicates that maybe "absolute position" isn't as important
  - Maybe can extrapolate to longer sequences as periods restart!
- Cons:
  - Not learnable; also the extrapolation doesn't really work!

Image: https://timodenk.com/blog/linear-relationships-in-the-transformers-positional-encoding/

Self-Attention

Barriers and solutions for Self-Attention as a building block

Barriers

- Doesn't have an inherent notion of order! →
- No nonlinearities for deep learning! It's all just weighted averages →

Solutions

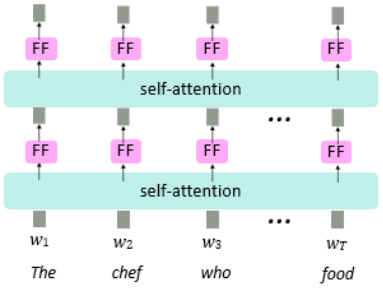
- Add position representations to the inputs

Self-Attention

Adding nonlinearities in self-attention

- Note that there are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages **value** vectors
- Easy fix: add a **feed-forward network** to post-process each output vector.

$$m_i = MLP(\text{output}_i)$$
$$= W_2 * \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2$$



Intuition: the FF network processes the result of attention

Self-Attention

Barriers and solutions for Self-Attention as a building block

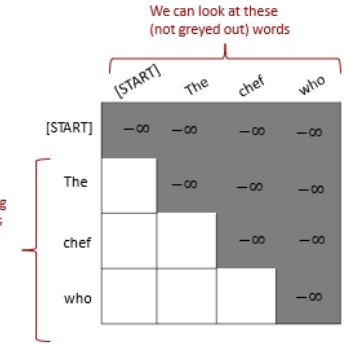
Barriers	Solutions
Doesn't have an inherent notion of order!	Add position representations to the inputs
No nonlinearities for deep learning magic! It's all just weighted averages	Easy fix: apply the same feedforward network to each self-attention output.
Need to ensure we don't "look at the future" when predicting a sequence • Like in machine translation • Or language modeling	

Self-Attention

Masking the future in self-attention

- To use self-attention in **decoders**, we need to ensure we can't peek at the future.
- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)
- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

$$e_{ij} = \begin{cases} q^T k_j, j < i \\ -\infty, j \geq i \end{cases}$$



Self-Attention

Barriers and solutions for Self-Attention as a building block

Barriers	• Solutions
Doesn't have an inherent notion of order!	Add position representations to the inputs
No nonlinearities for deep learning magic! It's all just weighted averages	Easy fix: apply the same feedforward network to each self- attention output.
Need to ensure we don't "look at the future" when predicting a sequence • Like in machine translation • Or language modeling	Mask out the future by artificially setting attention weights to 0!

Self-Attention

Necessities for a self-attention building block:

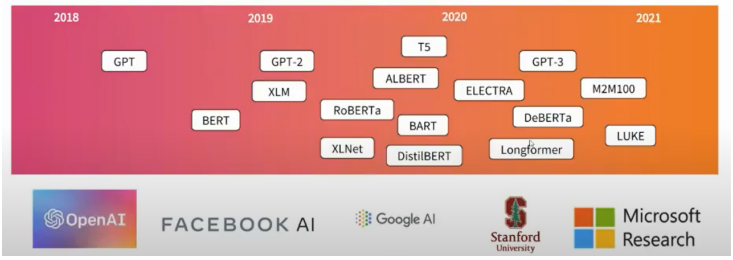
- Self-attention: the basis of the method.
- Position representations: Specify the sequence order, since self-attention is an unordered function of its inputs.
- Nonlinearities:
  - At the output of the self-attention block
  - Frequently implemented as a simple feed-forward network.
- Masking:
  - In order to parallelize operations while not looking at the future.
  - Keeps information about the future from "leaking" to the past.
- That's it! But this is not the Transformer model we've been hearing about.

Transformer

Transformer Attention is all you need

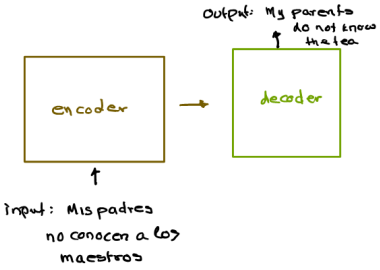
Popularity

Masking the future in self-attention



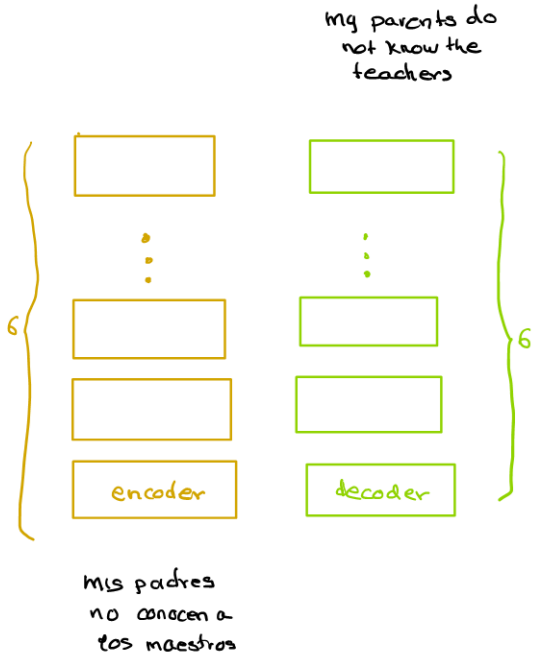
Transformers

- The Transformer is a model that uses attention to boost the speed with which seq2seq with attention models can be trained. The biggest benefit, however, comes from how The Transformer lends itself to parallelization. We will break it apart and look at how it functions.
- In its heart it contains an encoding component, a decoding component, and connections between them.



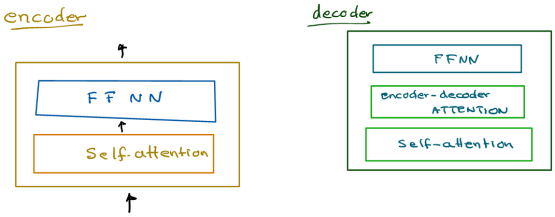
Transformers

- The encoding is a stack of encoders.
- The original paper stacks six of them on top of each other – there's nothing magical about the number six, one can definitely experiment with other arrangements).
- The decoding is a stack of decoders of the same number.

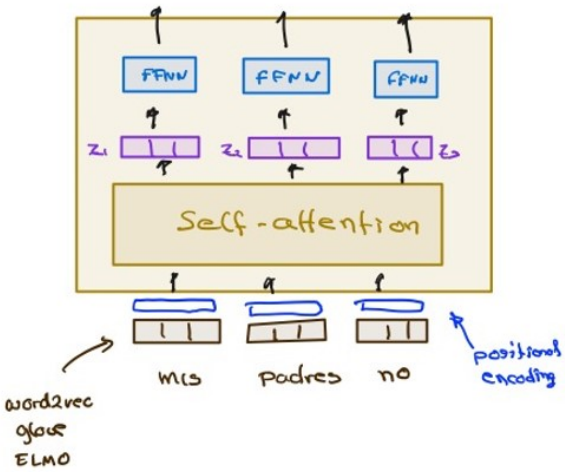


Transformers

The encoder's inputs goes through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word. The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence (similar what attention does in seq2seq models).



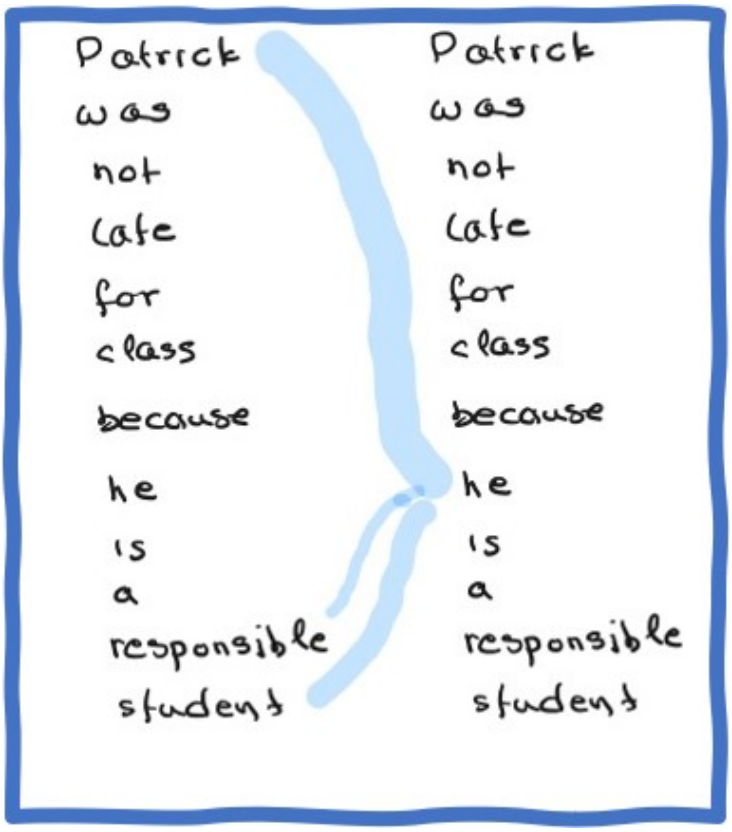
Transformers



- A key property of the Transformer, which is that the word in each position flows through its own path in the encoder. There are dependencies between these paths in the self-attention layer.
- The feed-forward layer does not have those dependencies, Therefore, the various paths can be executed in parallel while flowing through the feed-forward layer.

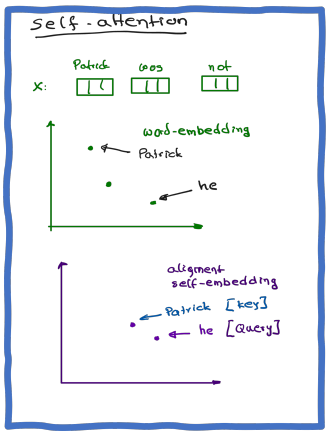
Transformers

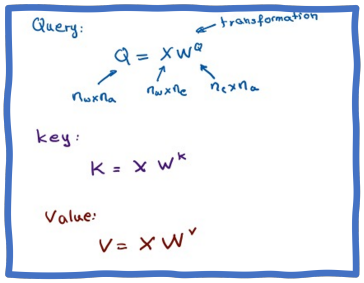
Patrick was not late for class because **he** is a responsible student

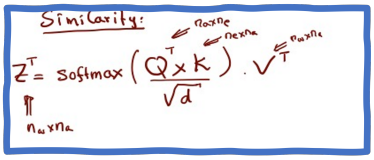


- What does "he" in this sentence refer to? Is it referring to the class or to the student? It's a simple question to a human, but not as simple to an algorithm.
- When the model is processing the word "he", self-attention allows it to associate "he" with "Patrick".

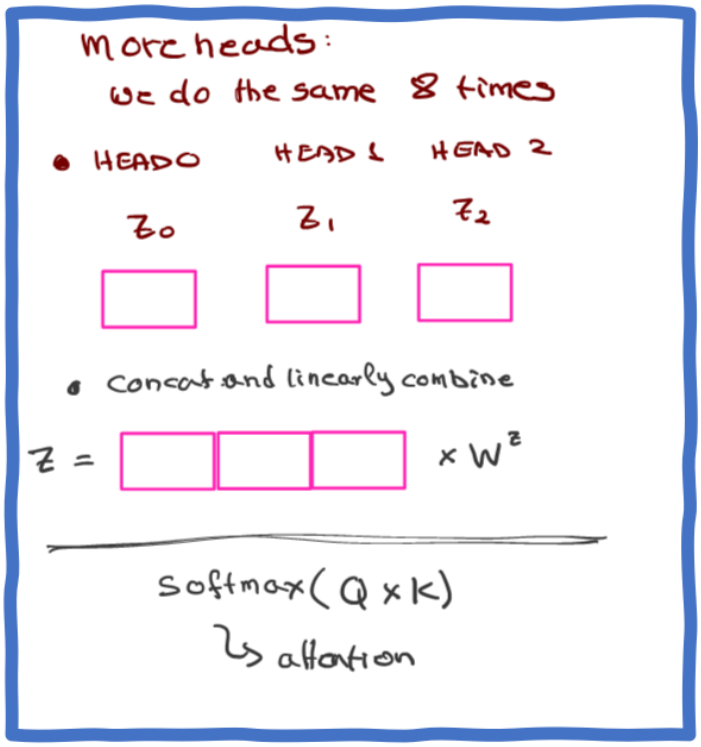
Transformers







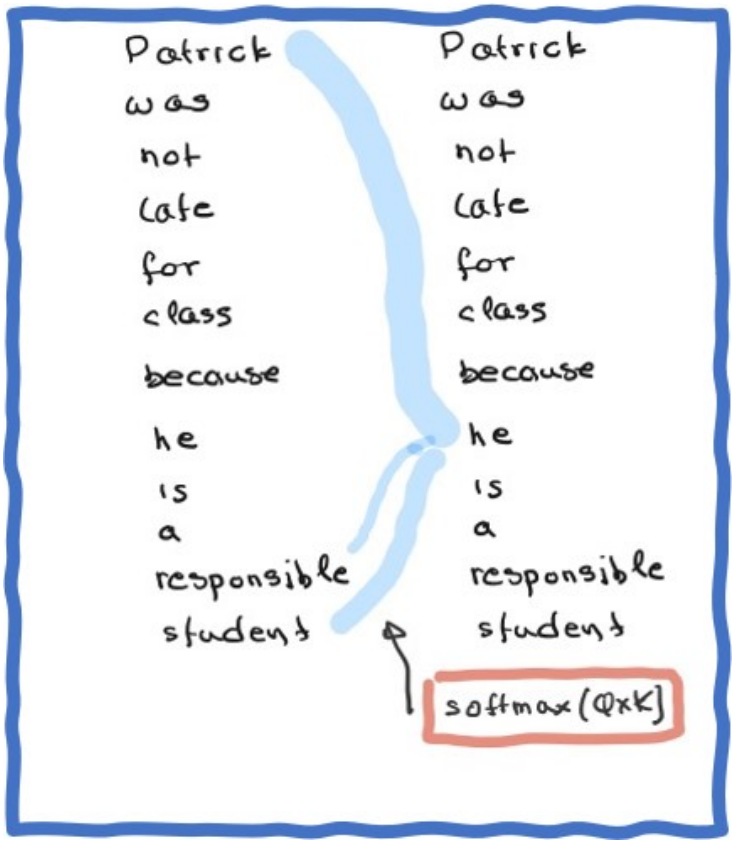
Transformers



In the same fashion as CNN that we need more than one filter, transformers add a mechanism called "multi-headed" attention. This improves the performance of the attention layer in two ways:

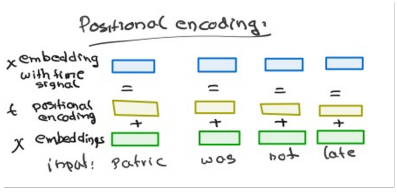
- It expands the model's ability to focus on different positions.
- It gives the attention layer multiple "representation subspaces"

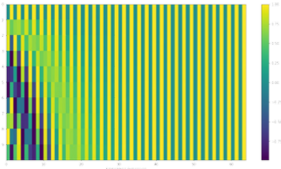
Transformers

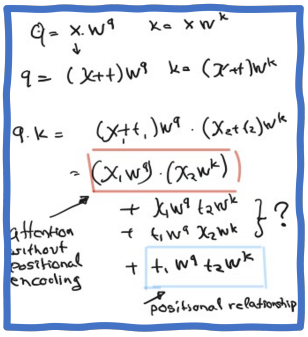


As we encode the word "he", one attention head is focusing most on "Patrick", while another is focusing on "student" – in a sense, the model's representation of the word "he" combines the representations of both "Patrick" and "stduent".

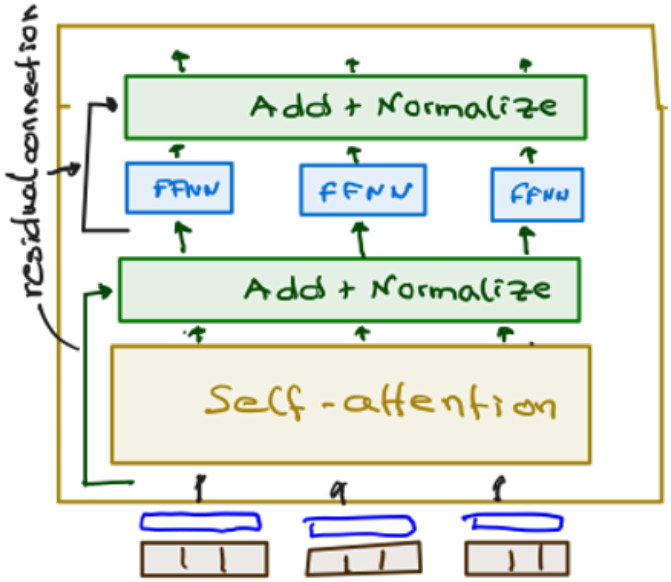
Transformers







Transformers



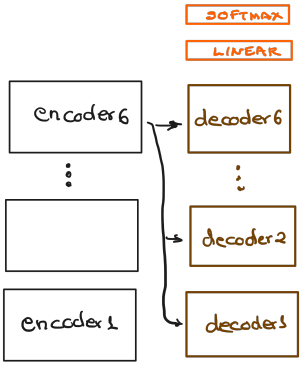
- Details in the architecture of the encoder:
- Each sub-layer in each encoder has a residual connection around it
- And a layer-normalization step.

Transformers

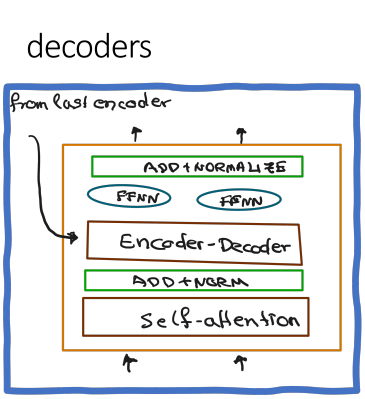
Transformer is stacked encoders and decoders.

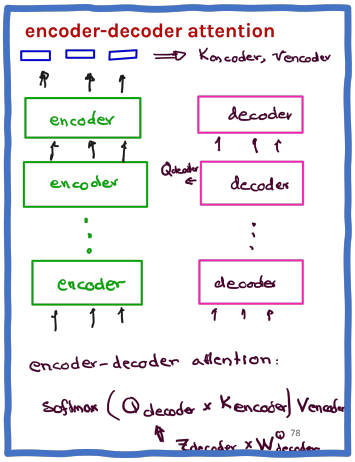
So far:

Positional encoding ☑
Encoder ☑
Self-attention ☑

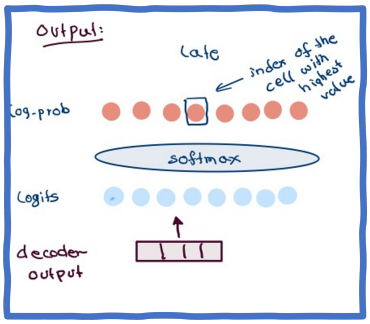


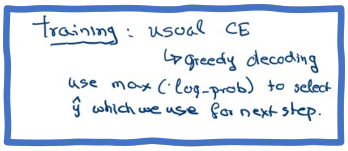
Transformers

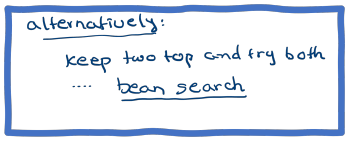




The Final Linear and Softmax Layer







Application

Applications

Sentence Transformers

Introduction

- a Python framework for sentence, text and image embeddings, for more than 100 languages
- embeddings can then be compared e.g. with cosine-similarity to find sentences with a similar meaning, semantic textual similar, semantic search, or paraphrase mining.
- Installation `pip install -U sentence-transformers`

Sentence Embedding

- to embed sentences into a vector space
- a must for using text in any machine learning algorithm

```
from sentence_transformers import SentenceTransformer,util
model = SentenceTransformer('all-MiniLM-L6-v2')

sentences = ['This framework generates embeddings for each
     input sentence',
    'Sentences are passed as a list of string.']

embeddings = model.encode(sentences)

for sentence, embedding in zip(sentences, embeddings):
    print("Sentence:", sentence)
    print("Embedding:", embedding)
    print("")
```

Sentence Similarity

- to compute the similarity between two sentences.
- more semantic the embedding, similarity becomes semantic

```
from sentence_transformers import SentenceTransformer,util
model = SentenceTransformer('all-MiniLM-L6-v2')

emb1 = model.encode("I am eating Apple")
emb2 = model.encode("I like fruits")
cos_sim = util.cos_sim(emb1, emb2)
print("Cosine-Similarity:", cos_sim)
```

Semantic Search

- Query-Response model
- for tasks such as question answering, where you must find documents containing answers to a given question

```
from sentence_transformers import SentenceTransformer, util
model = SentenceTransformer('clips/mfaq')

question = "How many models can I host on HuggingFace?"
answer_1 = "All plans come with unlimited private models and
     datasets."
answer_2 = "AutoNLP is an automatic way to train and deploy
     state-of-the-art NLP models, seamlessly integrated with
     the Hugging Face ecosystem."
answer_3 = "Based on how much training data and model
     variants are created, we send you a compute cost and
     payment link - as low as $10 per job."

query_embedding = model.encode(question)
corpus_embeddings = model.encode([answer_1, answer_2,
     answer_3])

print(util.semantic_search(query_embedding,
     corpus_embeddings))
```

Clustering

Grouping similar content

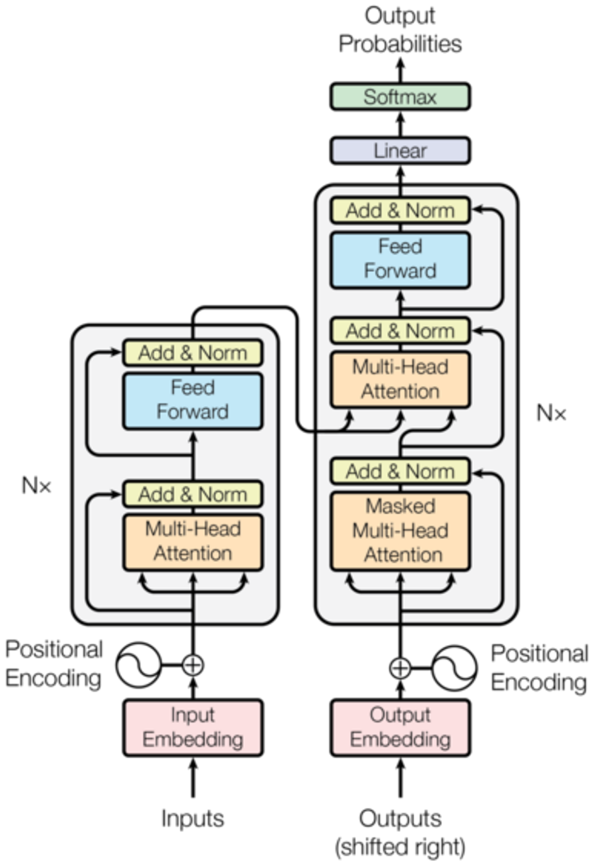
```
from sklearn.cluster import KMeans
import numpy as np
embedder = SentenceTransformer('all-MiniLM-L6-v2')
corpus = ['A man is eating food.','A man is eating a piece
     of bread.', 'Horse is eating grass.','A man is eating
     pasta.' ...]
corpus_embeddings = embedder.encode(corpus)
# Normalize the embeddings to unit length
corpus_embeddings = corpus_embeddings /
     np.linalg.norm(corpus_embeddings, axis=1, keepdims=True)
clustering_model = KMeans(n_clusters=4)
clustering_model.fit(corpus_embeddings)
cluster_assignment = clustering_model.labels_
print(cluster_assignment)
clustered_sentences = {}
for sentence_id, cluster_id in enumerate(cluster_assignment):
    if cluster_id not in clustered_sentences:
        clustered_sentences[cluster_id] = []

    clustered_sentences[cluster_id].append(corpus[sentence_id])
print(clustered_sentences)
```

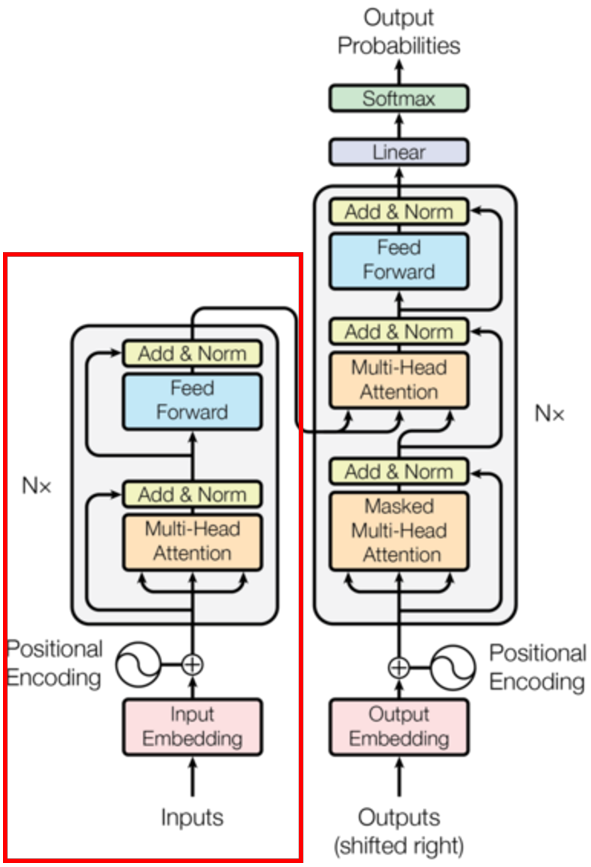
(ref. https://stackoverflow.com/questions/55619176/how-to-cluster-similar-sentences-using-bert)

Conclusion

Conclusion

Architecture



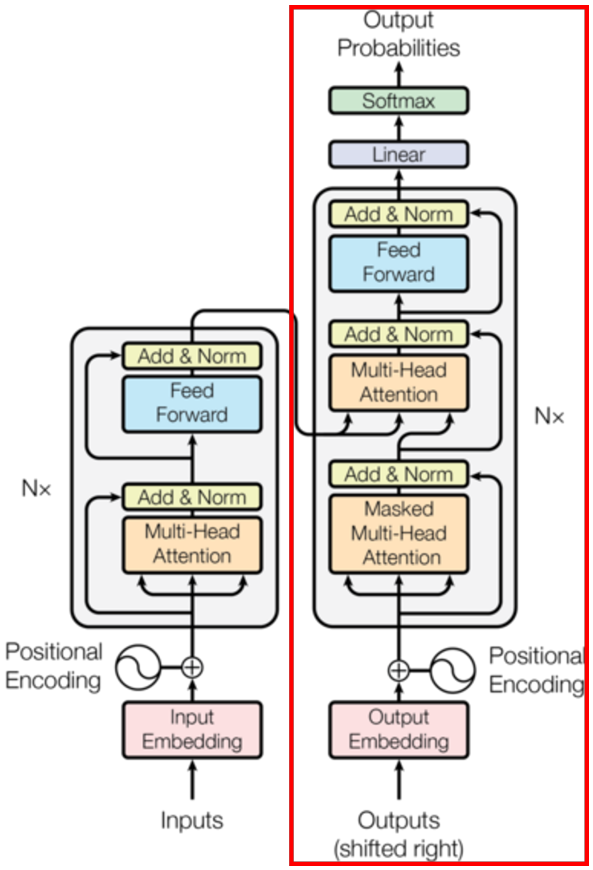


- Encoder-Decoder was there
- New?: Multi head attention
- Fine tuned with Custom Data
- Flavors:
  - Encoder only
  - Decoder only
  - Encoder-Decoder

Encoder only

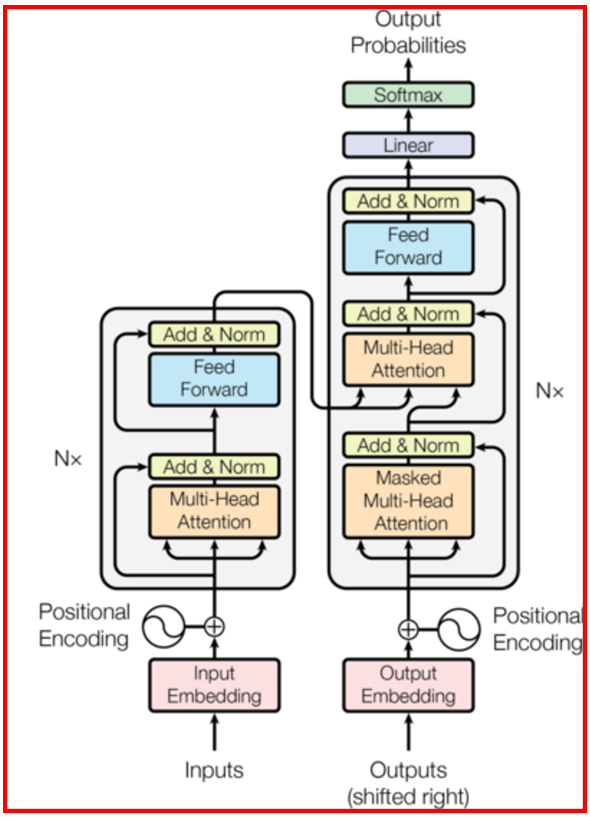
- If we are only interested in training a language model for the input for some other tasks, then we do not need the decoder of the transformer.
- Pre-trained by predicting masked word
- BERT, XLNet, DistillBERT, RoBERTA
- Usage:
  - Text Classification
  - Named Entity Recognition
  - Extractive Question Answering

Decoder only



- If we do not have input, we just want to model the "next word", we can get rid of the encoder side of a transformer and output "next word" one by one.
- Pre-trained by predicting next word
- GPT-*, Transformer XL
- Usage:
  - Text Generation

Encoder-Decoder, both



- Pre-trained by Seq2Seq manner
- T5, BART, PEGASUS
- Usage:
  - Text summarization
  - Machine Translation
  - SQL generation

References

- Course materials of Christopher Manning and John Hewitt
- "I never knew Sentence Transformers could be so useful!" - Pradip Nichite
- BERT: Human Language Technologies, Dipartimento di Informatica, Giuseppe Attardi
- BERT Explained: A Complete Guide with Theory and Tutorial – Samia Khalid
- BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding – Jacob Devlin
- AC295 Pavlos Protopapas, Inst of Applied Computational Science, Harvard
- The Annotated Transformer – Sasha Rush
- The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning) – Jay Alammar
- The Illustrated Transformer – Jay Alammar