Session 2
Deep Learning and Transformers

Palacode Narayana Iyer Anantharaman

16th Aug 2024

Transformer Visualization Demo

- https://poloclub.github.io/transformer-explainer/

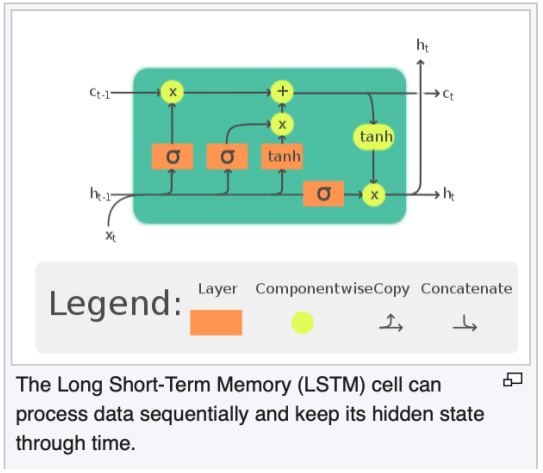
How LLMs are trained?

- Pretraining: The goal is to learn language patterns, grammatical constructs, common phrases, facts, some bit of reasoning, etc.

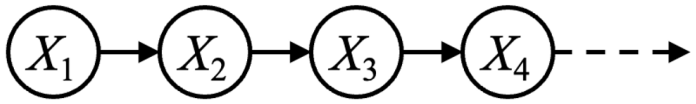
- Finetuning: The goal is to specialize the base pretrained model on specific tasks, in particular instruction following behavior

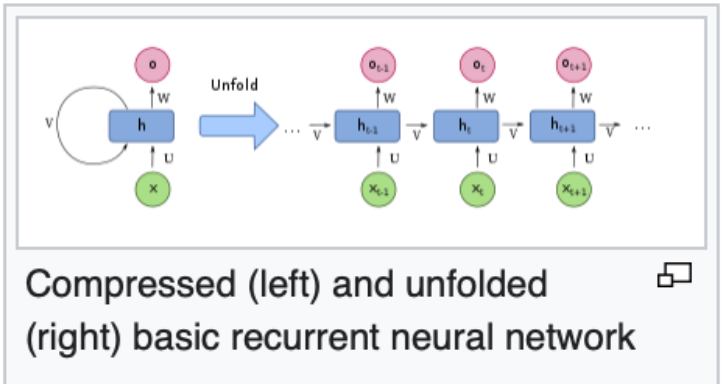
- (Optional) RLHF: Build a human compatible responses, avoid toxic content, enable honest, bias-less and factual answers

Key Foundational Technologies

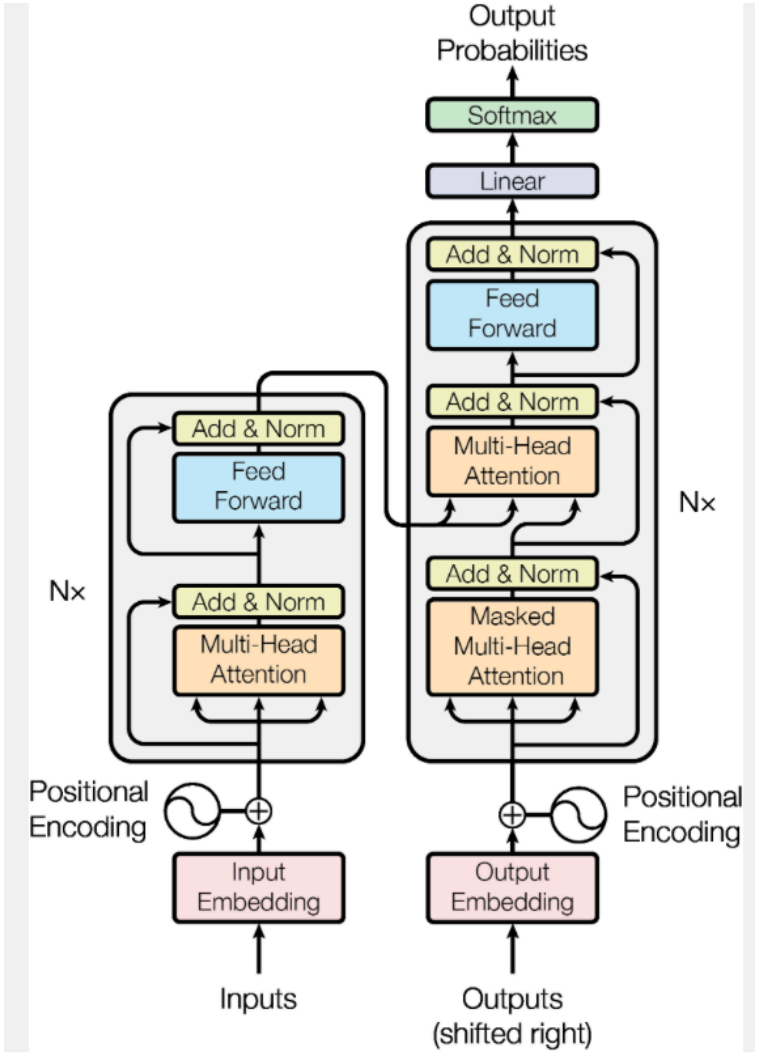


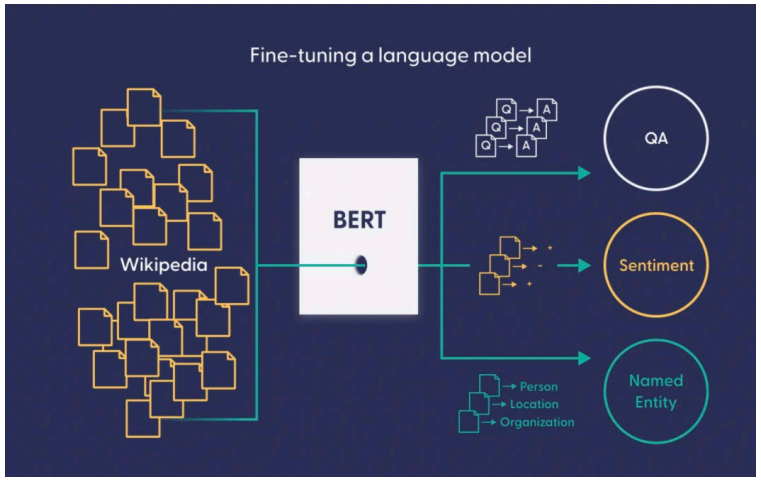
The Long Short-Term Memory (LSTM) cell can process data sequentially and keep its hidden state through time.

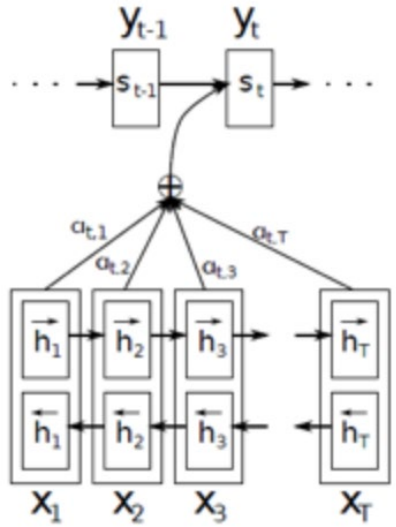




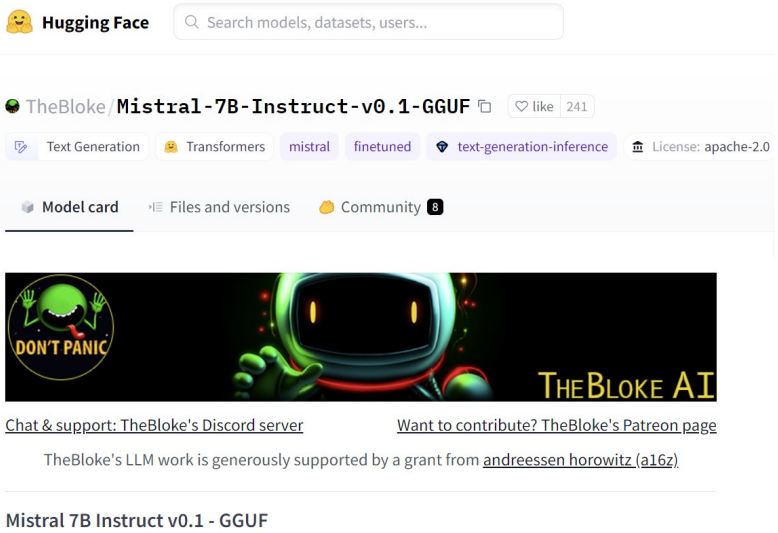
Compressed (left) and unfolded (right) basic recurrent neural network

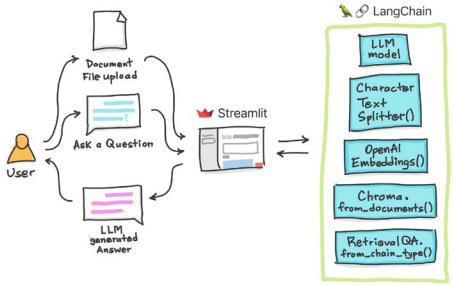


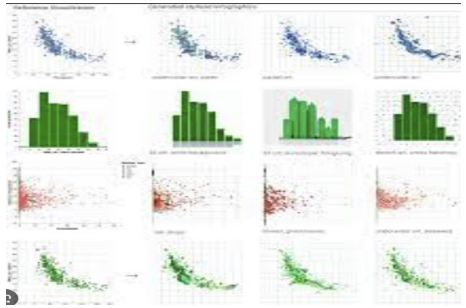


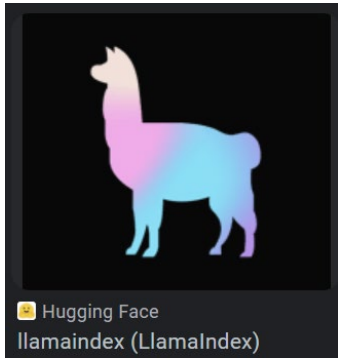


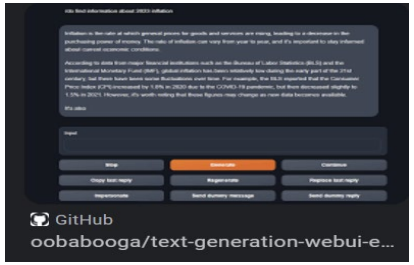
Models, Tools and Technologies

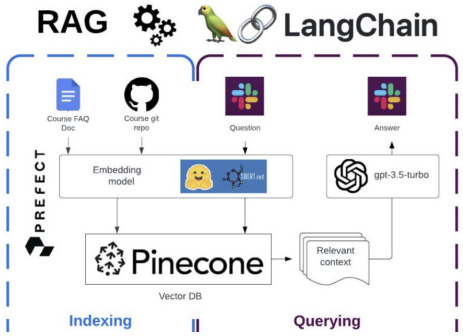






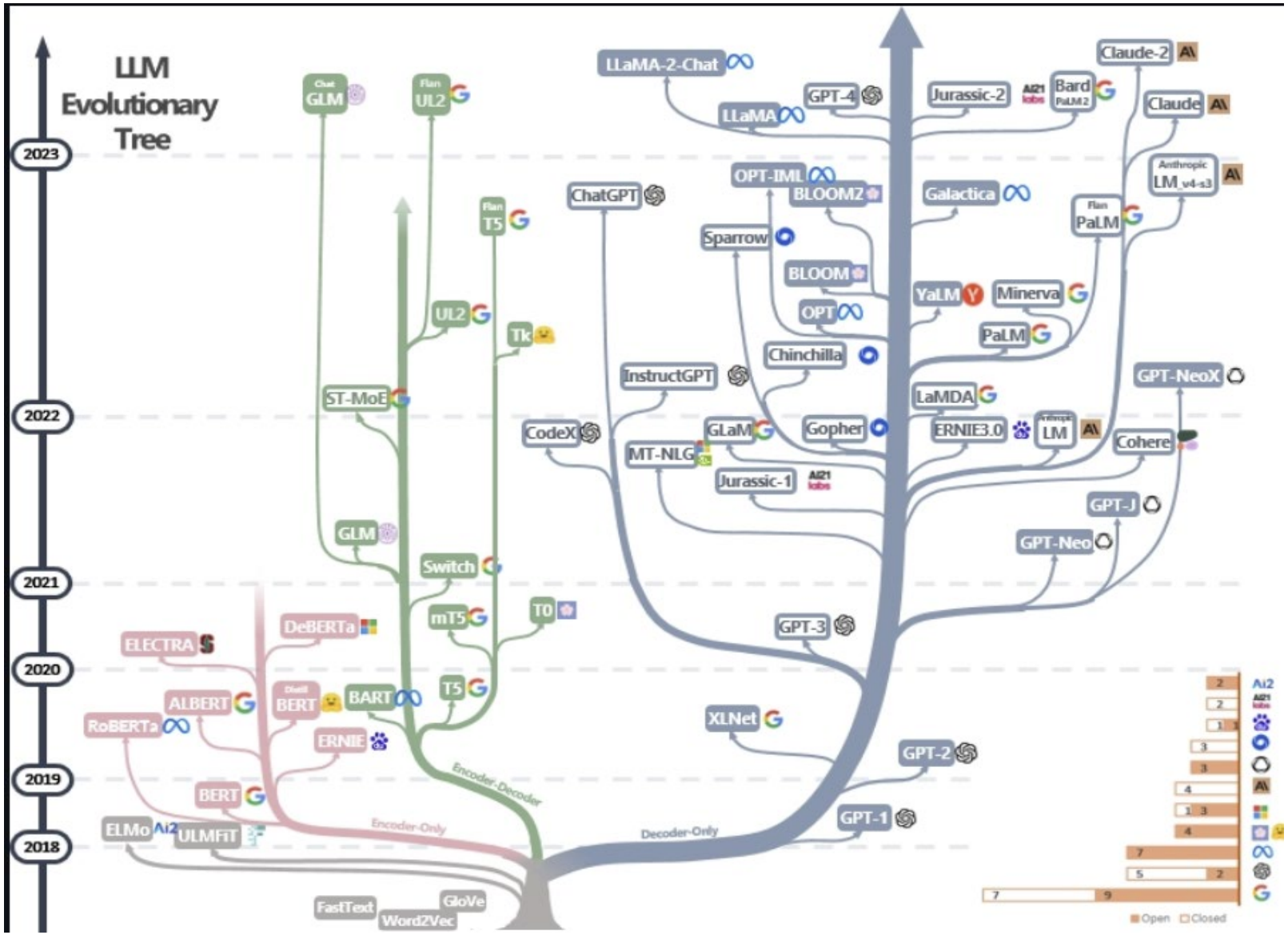






LLM is a huge **Transformer** (Encoder | Decoder | Both) Model that is **trained** with massive amount of data for one or more **NLP tasks**.

Transformers/LLM Timeline



GenAI products can be:

- Proprietary Models
- Open Source Models

Open source models are mostly **open source weights**.

The data used to train the models or the source code of model and training algorithms may not be public.

Licenses may permit usage only for research purposes

Some models permit more permissible **commercial usage.** (Llama 2, Mistral 7B, Llama 3)

Finetuned models derived from foundation models specialize the functionality

Key Topic #1 : Language Models

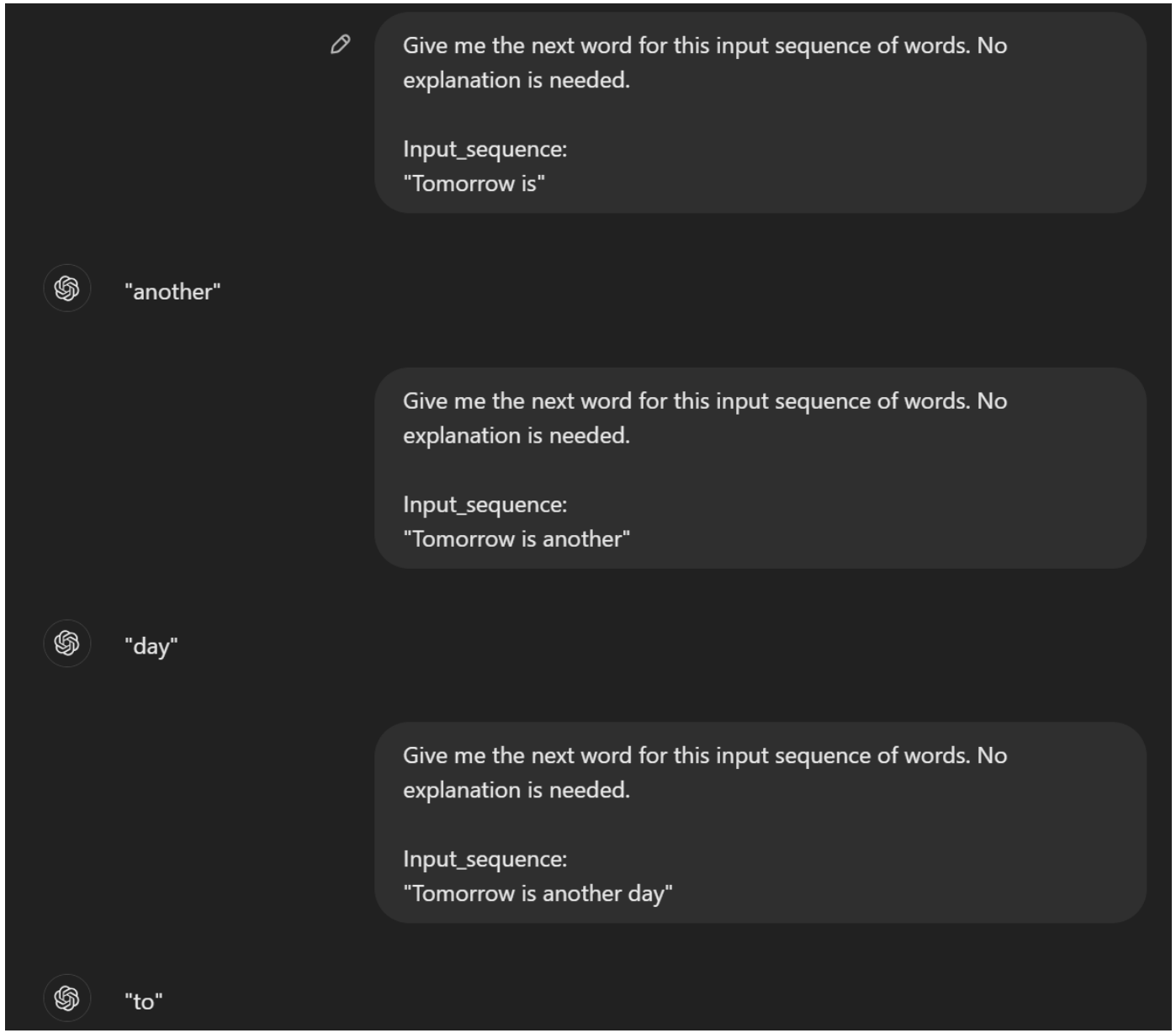
LMs and LLMs

- Language Model (LM) is a term that has its roots in Natural Language Processing

- LM computes the probability of the next token, given context consisting of current and previous tokens.
  - E.g. Given two context tokens (“bread”, “and”), most of us may predict the next word to be “butter” as the phrase “bread and butter” is quite common.
  - Similarly, one can complete phrases: `cout <<` , `SELECT *`, etc

- LMs can generate text. That is, given the context tokens, an LM can generate future words. **This is called an autoregressive model.**
  - GPTs are autoregressive

Illustration of an Autoregressive generation (GPT)



- Autoregressive models are trained on a huge corpus of data, say all Wikipedia contents, web scraped text, public documents etc.
- The training data is formed by “tokenizing” sentences to words (or sub words) to form different sub sequences.
- For each subsequence, the next word in that subsequence constitutes the target for the purpose of training.
- This is a form of unsupervised training, sometimes also termed as semi supervised.
- This training data can be used to train a sequence model (such as a RNN, LSTM or a Transformer)

Semi Supervised Learning

- Training a LM is a semi supervised learning task
- Start with a huge corpus of text data. Tokenize each sequence (say, a sentence) and further tokenize in to basic tokens (sub words).
- Form a dataset such that the word w_{i+1} is the target for the subsequence $w_0 \ldots w_i$
- Example: A special flight was arranged after the players were stranded in a hotel for three days after the victory.
- The above example has 19 words excluding special tokens like SOS, EOS. We can form as many sequences and define targets.
- Inputs: [“A”, “A special”, “A special flight”, “A special flight was”, …]
- Targets: [“special”, “flight”, “was”, “arranged”, …]

Large Language Models (LLM)

- LLMs are based on similar principle as LMs – that is predict a word, given context words.
- LLMs are termed “large” due to the size of the model in terms of number of parameters and also the number of tokens that are used to train the model.
- An LLM like ChatGPT takes in a sequence as input and produces a sequence of tokens as output. Thus, LLMs are sequence to sequence models like transformers.
- GPT stands for Generative Pretrained Transformer, where the pretraining is done on LM objective.
- A “small” LLM like Tiny Vicuna 1B model has 1B parameters and trained using hundreds of billions or even trillions of tokens, while a model like GPT-4 is rumored to have 1.76 Trillion parameters.

The Language Model Problem

- The LM problem is to assign a probability to a word sequence
  - $P(w_1, w_2, \ldots, w_n)$
  - Example: P(tomorrow, is, a, holiday)

- This may also be stated as a problem of computing the probability of each word given its preceding context.
  - $P(w_i \mid w_1, w_2 \ldots w_{i-1})$
  - Example: P(holiday | tomorrow, is, a)

- Language Models can generate text (Auto regressive)

The key question is, how do we implement a machine that takes the context and provides a meaningful completion?

Language Model Implementation

- Lexical Based: n-gram models
  - Limited Context (Bigram, Trigram)
- Neural Networks based (Bengio, 2003)
  - A small window of context (e.g. 3 words of context)
- Deep networks based (RNN, LSTM)
  - A larger context window (tens of tokens)
- **Transformers based**
  - Much bigger context (typically 4K or above), captures correlations between tokens more effectively. BERT and GPT are the 2 popular transformer based architectures.

N-gram Model

e.g. Have a good day!

- Unigram: Words in the given sequence are independent of each other

- Bigram: A word at position t depends only on the one which is at position t-1 and given this, it is independent of all other words in the sequence. (First order Markov assumption)

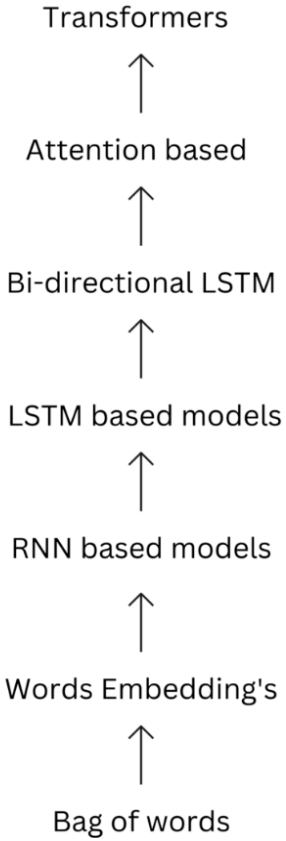
- Trigram: A word at position t depends only on the one which is at position t-1 and the one at position t-2 and given these, it is independent of all other words in the sequence. (Second order Markov assumption)

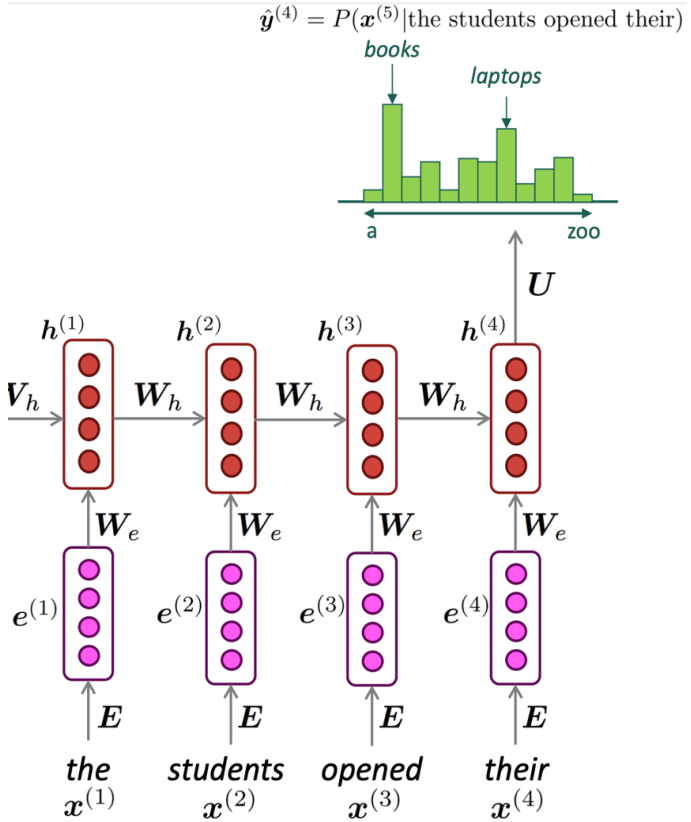
- These assumptions are needed to make LM computation tractable

LM Implementation using DL Models

- Language Models can be implemented using deep learning networks, such as RNN, LSTM, Transformer networks

- These are covered later in this presentation





Language Models and Embedding

- Embedding provides a numerical representation of a token, typically as a vector.
- A token could be a word, sub word, special words, or even characters. For images, a token may represent a patch m x n of an image.
- Typically, this embedding is obtained by training a deep learning model on a LM objective.
- Thus, embedding is one of the applications of Language Modeling
- Other applications of LM include speech recognition, text completion, NLP tasks like Machine Translation and so on.

Transformers Based Embedding

- The State Of The Art (SOTA) embedding models are Transformer based.
- BERT and GPT architectures created the initial breakthroughs
- The success of such models were key to the evolution leading to ChatGPT
- We cover the technical architecture of BERT/GPT after discussing Transformers

Tokenization and Embedding (takeaways are highlighted in gray)

- Natural Language text is represented in lexical form by characters, words, sentences, paragraphs etc.
- We could view these lexical units as an ordered sequence of tokens.
  - Tokens could be characters, sub words, words, etc.
- **The process of splitting a natural language text in to tokens is called tokenization.**
- **Embedding Model transforms each token to an embedding vector**, so that they can be further processed by a DL Model.
- Tokenization is not an unique process, usually it depends on the model we are using. E.g. BERT token set is different from that of GPT 4

HuggingFace Example

- While we focus mostly on text, the data creation process for any modality (image, video, audio) have some common traits.
- HuggingFace provides a set of classes to handle this.

- Text, use a Tokenizer to convert text into a sequence of tokens, create a numerical representation of the tokens, and assemble them into tensors.
- Speech and audio, use a Feature extractor to extract sequential features from audio waveforms and convert them into tensors.
- Image inputs use a ImageProcessor to convert images into tensors.
- Multimodal inputs, use a Processor to combine a tokenizer and a feature extractor or image processor.

Autotokenizer (Ref: https://huggingface.co/docs/transformers/en/autoclass_tutorial)

```
>>> from transformers import AutoTokenizer

>>> tokenizer = AutoTokenizer.from_pretrained("google-bert/bert-base-uncased")
```

Then tokenize your input as shown below:

```
>>> sequence = "In a hole in the ground there lived a hobbit."
>>> print(tokenizer(sequence))
{'input_ids': [101, 1999, 1037, 4920, 1999, 1996, 2598, 2045, 2973, 1037, 7570, 10322, 4183, 1012, 102],
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

Demo: BERT Tokenizer, Embedding

```
[ ]: import torch
```

```
[ ]: # from pytorch_pretrained_bert import BertTokenizer, BertModel
from transformers import BertTokenizer, BertModel
```

```
[ ]: tokenizer = BertTokenizer.from_pretrained('bert-base-multilingual-cased')  # ("bert-base-uncased")
```

```
[ ]: txt = "I love Adobe Photoshop"
txt1 = "[CLS] " + txt + " [SEP]"
tokenized_text = tokenizer.tokenize(txt1)
```

```
[ ]: with open("vocab.txt", "w", encoding="utf-8") as f:
  for token in tokenizer.vocab.keys():
    f.write(token + "\n")
```

```
[ ]: list(tokenizer.vocab.keys())[48947]
```

```
[ ]: # Map the token strings to their vocabulary indeces.
indexed_tokens = tokenizer.convert_tokens_to_ids(tokenized_text)

# Display the words with their indeces.
for tup in zip(tokenized_text, indexed_tokens):
    print('{:<12} {:>6,}'.format(tup[0], tup[1]))
```

```
[ ]: # Mark each of the tokens as belonging to sentence "1".
segments_ids = [1] * len(tokenized_text)

print (segments_ids)
```

```
[ ]: # Convert inputs to PyTorch tensors
tokens_tensor = torch.tensor([indexed_tokens])
segments_tensors = torch.tensor([segments_ids])
```

```
[ ]: # Load pre-trained model (weights)
model = BertModel.from_pretrained('bert-base-uncased',
                                  output_hidden_states = True, # Whether the model returns all hidden-states.
                                  )

# Put the model in "evaluation" mode, meaning feed-forward operation.
model.eval()
```

Hands On: Autotokenizer

- Task#2 Objective: Familiarize yourself with Tokenizers

(a) Use the Autotokenizer from HuggingFace and create an instance of Llama 3 tokenizer.

(b) Tokenize using tokenizer.tokenize() method for the text:

“Amoxicillin is used to treat a wide variety of bacterial infections. This medication is a penicillin-type antibiotic.”

(c) Determine the following by writing a suitable program:

- Vocabulary size of Llama 3 tokenizer

- Special tokens used in a given sentence e.g. in BERT [CLS] and [SEP] are used.

- Dimensions of the embedding matrix

Encoder Decoder Design

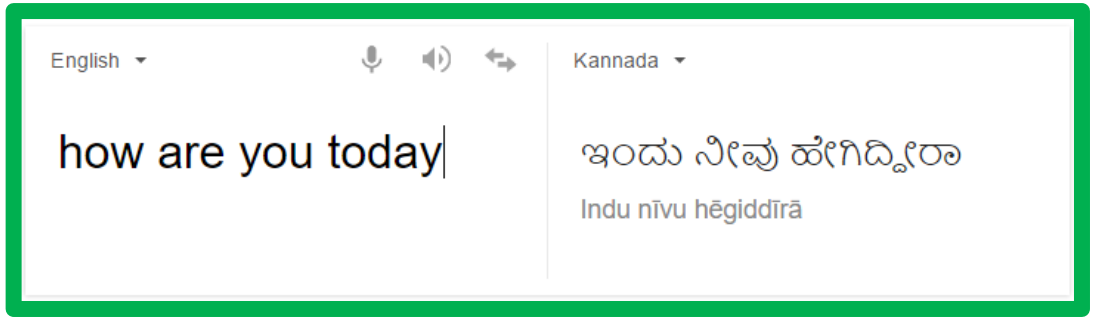
- Use 2 RNN's, one for encoding and the other decoding
  - Example: Machine Translation

- The activations of the final stage of the encoder is fed to the decoder

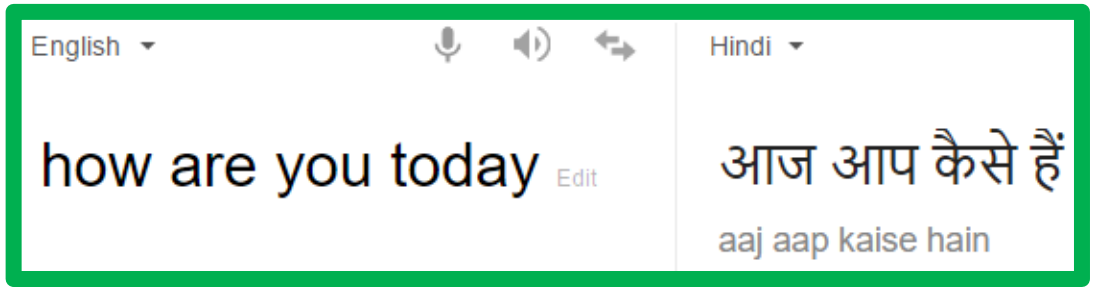
- Useful when the output sequence is of variable length and the entire input sequence can be processed before generating the output

What are attention networks?

- Suppose we have a sequence of input and we are to produce a sequence of outputs for this.
  - E.g. The inputs could be a sequence of words, could also be image segments

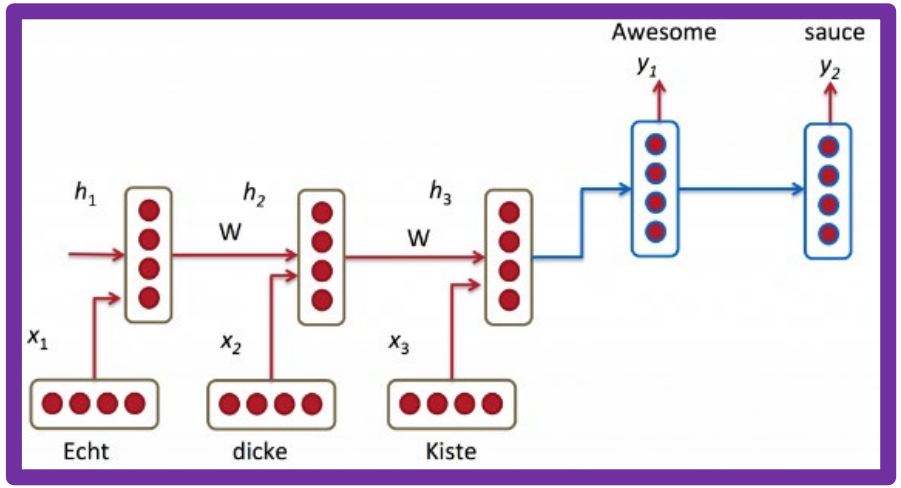


- Attention mechanisms focus on a narrow region of the input when making a classification decision for the output sequence.
  - E.g. In a machine translation, when outputting a word, the attention network might give higher weightage to certain words in the input.



Neural Machine Translation

- The source sentence is encoded as a single fixed length vector which is then translated in to the target language sentence.
- The encoding process is equivalent to learning a higher level meaning of the sentence, which can't be effectively captured using techniques like n-grams
- The length of input sequence is arbitrary, it is not always effective to encode sentence with same length vector.
  - Suppose you ask an LLM to summarize a big paragraph, is it efficient to use a single vector as the context?
  - Given a large paragraph and a question as shown in the figure on the right, can the model write the required answer?
- Attention mechanisms address the issue by using the hidden states of every step in the encoder to compute an attention vector for the decoder.



The Tribhuvan Museum contains artifacts related to the King Tribhuvan (1906–1955). It has a variety of pieces including his personal belongings, letters and papers, memorabilia related to events he was involved in and a rare collection of photos and paintings of Royal family members. The Mahendra Museum is dedicated to king Mahendra of Nepal (1920–1972). Like the Tribhuvan Museum, it includes his personal belongings such as decorations, stamps, coins and personal notes and manuscripts, but it also has structural reconstructions of his cabinet room and office chamber. The Hanumandhoka Palace, a lavish medieval palace complex in the Durbar, contains three separate museums of historic importance. These museums include the Birendra museum, which contains items related to the second-last monarch, Birendra of Nepal.

When did Tribhuvan die?

Attention Mechanism

- When using the attention mechanism we make use of the hidden vectors computed at each time step of the encoder and hence the decoding doesn't need to depend on one single hidden vector

- The decoder "attends" to different parts of the source sentence at each step of the output generation.

- The model learns what to attend to based on the input sentence and what it has produced so far.
  - E.g. In the example of English to Kannada translation, the last word "today" translates in to the first word "indu" in Kannada. The attention model learns to focus on "today" when generating "indu"

Attention Mechanism

- X_i are the input words, processed by the encoder. Y_i are the outputs, the translated words
- The output Y_t depends not only on the previous hidden state value but also on the weighted combination of all input states
- α‘s are weights that determine how much of a given input word should be considered in computing the output at a given time step
- α‘s are normalized to 1 using a softmax

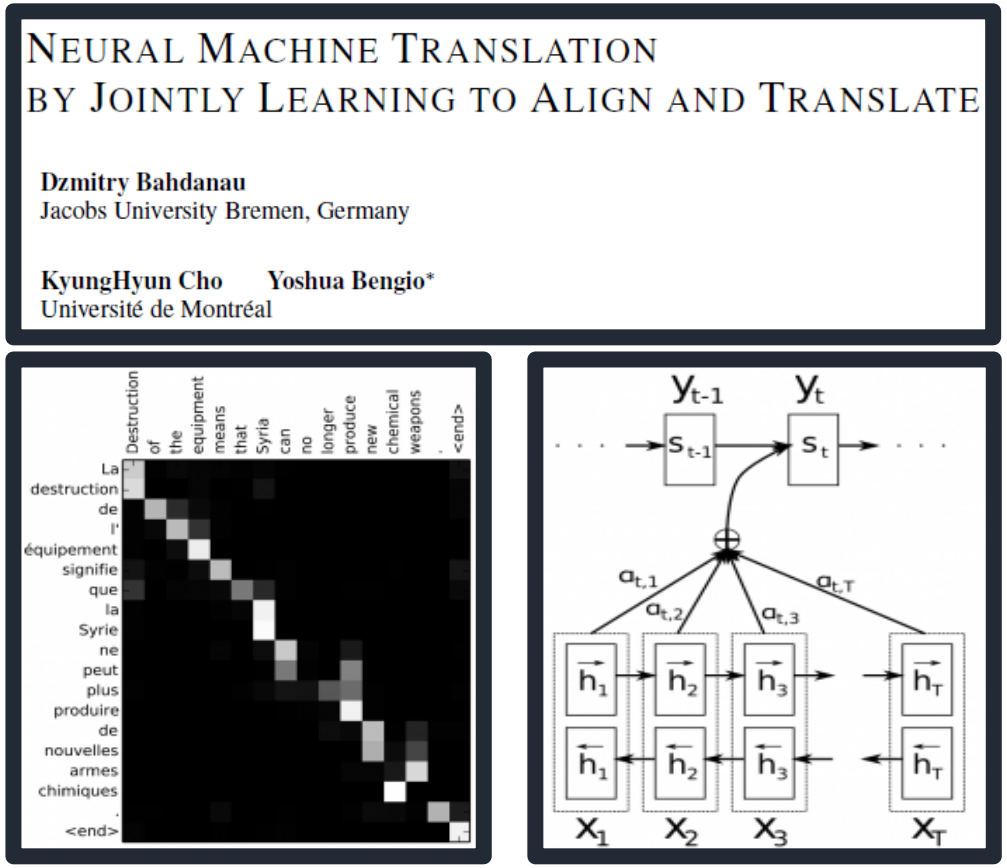
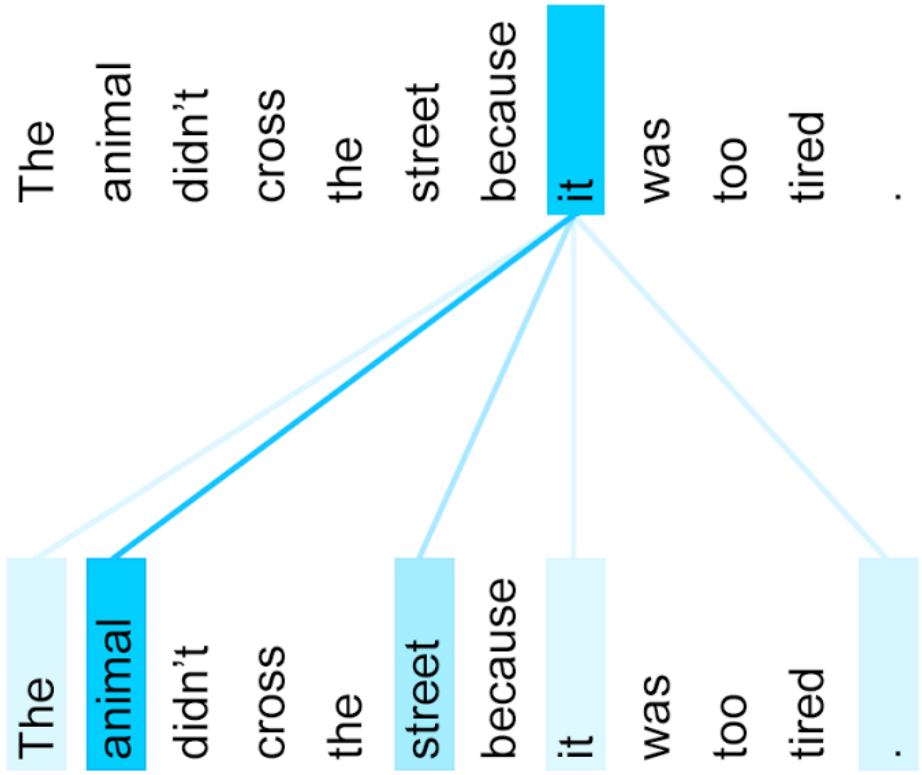
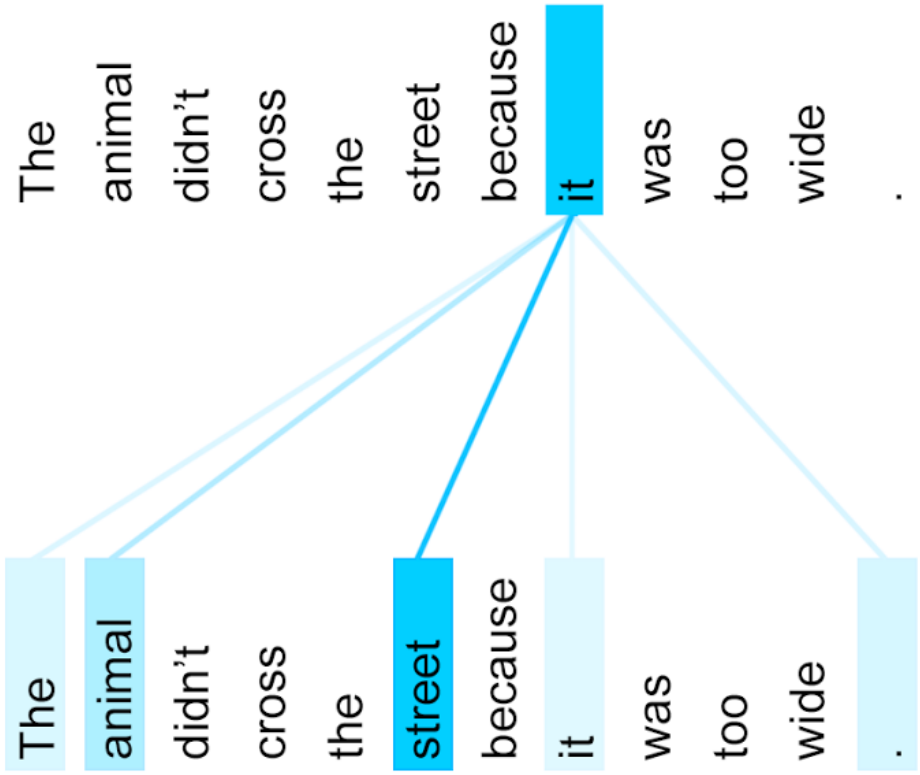


Illustration of Self Attention – Coreference Resolution





Self Attention for NLP Tasks

- Coreference Resolution
  - Sachin played well. The legend got the “man-of-the-match” award.

- Anaphora/Cataphora resolution
  - Sachin played well and he was awarded with the trophy.
  - As he was preparing to face the first ball from the fiery fast bowler Dennis Lillie, Sunil Gavaskar appeared an embodiment of concentration.
  - When he wears that blue jersey he will be a different Hardik Pandya because we know he can score those runs and take those wickets

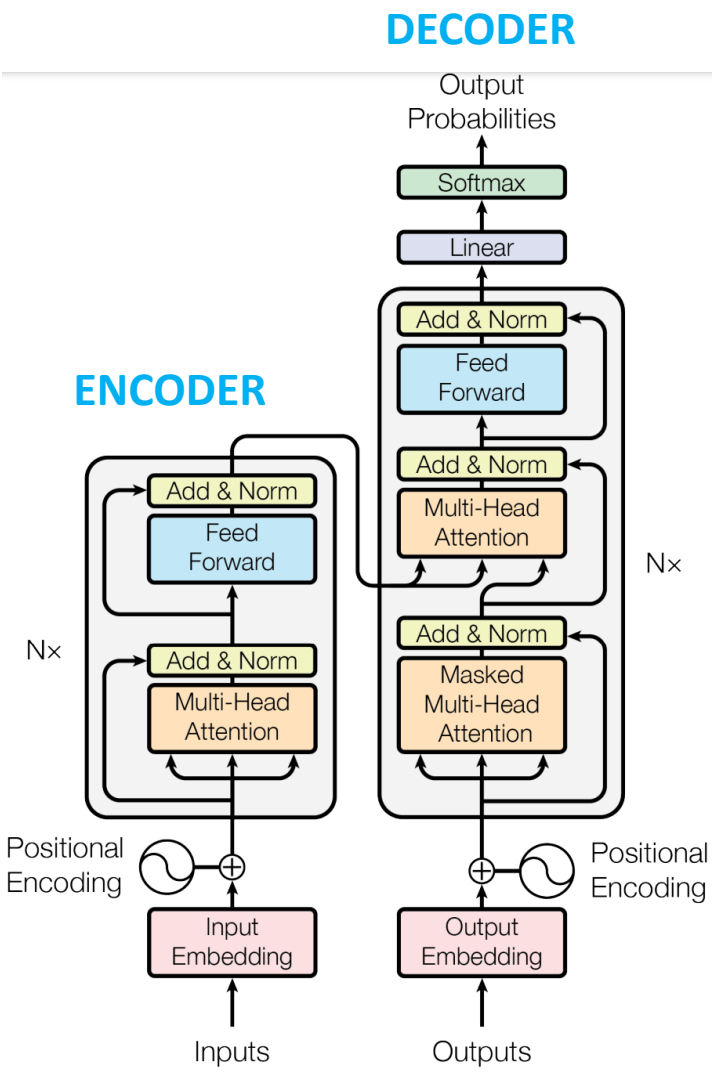
- Word sense disambiguation
  - Ganga was furious today as there were heavy rains due to the cyclone.

Part 2: Transformers - A Swiss Army Knife for Language Tasks

Transformers are transformative

- Transformer Networks are excellent at capturing patterns in long sequences
  - Self Attention capturing pairwise influences between tokens in a sequence, mitigation of vanishing gradient issues
- They are scalable to huge datasets
  - Model capacity due to a multilayer architecture, Ability to process tokens in parallel
- Extends beyond NLP applications
  - CNNs capture correlation of pixels in a small kernel. Vision Transformers capture long range dependencies. Imagine the ability to take in to account the influence of a pixel at the top left of an image with those at the bottom right.
- ImageNet moment for NLP: You can download a pretrained model and finetune with custom data.
  - Several papers published with corresponding pretrained models that can be finetuned.
- Model hubs like Hugging Face's enabled easy and consistent access to models and datasets.
  - Number of transformer based models published in Hugging Face exploded.

Architecture



- Transformer is an implementation of encoder-decoder architectural pattern.
- The encoder maps an input sequence of symbol representations (x_1, ..., x_n) to a sequence of continuous representations z = (z_1, ..., z_n).
- Given z, the decoder then generates an output sequence (y_1, ..., y_m) of symbols one element at a time.
- At each step the model is auto-regressive consuming the previously generated symbols as additional input when generating the next.

Building Blocks

- Input Embedding
- Positional Encoding
- Encoder Stack (A list of Encoders)
  - Encoder
    - Multi Head Attention (h-heads)
      - Scaled dot product attention
    - Feed Forward Network
    - Layer Norm

- Output Embedding
- Positional Encoding
- Decoder Stack (A list of decoders)
  - Decoder
    - Masked Multi Head Attention (h-heads)
      - Scaled dot product attention
    - Multi Head Attention (h-heads)
      - Scaled dot product attention
    - Feed Forward Network
    - Layer Norm
- Output Stage
  - Linear Layer (logits)
  - Softmax

Dimensions and Terminology

- Inputs to the transformer are a sequence of **tokens** (say, words or sub words from a piece of text) that are in a symbolic form (lexical)
  - Text => Tokenize => sub_words
- Tokens are to be represented using an **embedding**
  - Sub_word => d_{model} dimensional vector
- We denote the dimension of embedding same as model dimension: d_{model}
- We view the notion of Attention as querying a piece of memory organized as key-value pairs: (**Query, Keys, Values**). So, we need to derive these Q, K, V vectors from input vector by projecting on to d_k, d_k, d_v space. E.g. from 512 dimensions to 64 dimensions.
- We denote the dimensions of Query, Key, Value to be: d_k, d_k, d_v
- In the original paper, $d_{model} = 512$, $d_k = 64$, $d_v = 64$ and number of heads in the multi head = 8.
- The number of layers in the Encoder (also Decoder) stack = 6

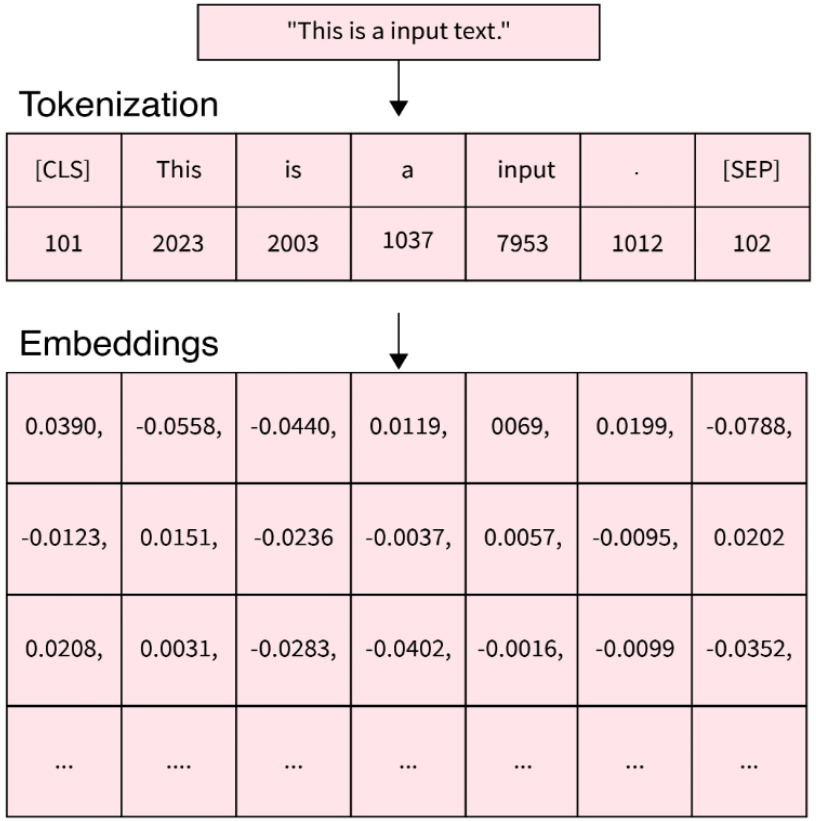
Embedding

- Embedding associated each token with a vector, termed embedding
- A good embedding should preserve the meaning of the token that should adapt well to the context in which it is used
- A key hyperparameter is the embedding dimension, we set it to $d_{model} = 512$

Enter text: Creating embeddings from tokens is the first step for transformers
{'input_ids': [101, 4526, 7861, 8270, 4667, 2015, 2013, 19204, 2015, 2003, 1996, 2034, 3357, 2005, 19081, 102], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
['[CLS]', 'creating', 'em', '##bed', '##ding', '##s', 'from', 'token', '##s', 'is', 'the', 'first', 'step', 'for', 'transformers', '[SEP]']

Vocabulary Size = 30522

Embedding Illustrated



Embedding

```python
class Embedder(torch.nn.Module):

    def __init__(self, vocab_size, d_model):
        super().__init__()
        self.embed = torch.nn.Embedding(vocab_size, d_model)

    def forward(self, x):
        # [123, 0, 23, 5] -> [[..512..], [...512...], ...]
        return self.embed(x)
```

POS Encoding

- Transformers do not have implicit position information, as the input tokens can be processed in parallel. At the same time, we need to know their position in order to preserve the sequence.
- Hence we encode the position explicitly
- Positional encoding needs to represent a pattern that can be learned by the model.
- POS encoder returns the position as a vector. We need the length of the vector to be d_{model}

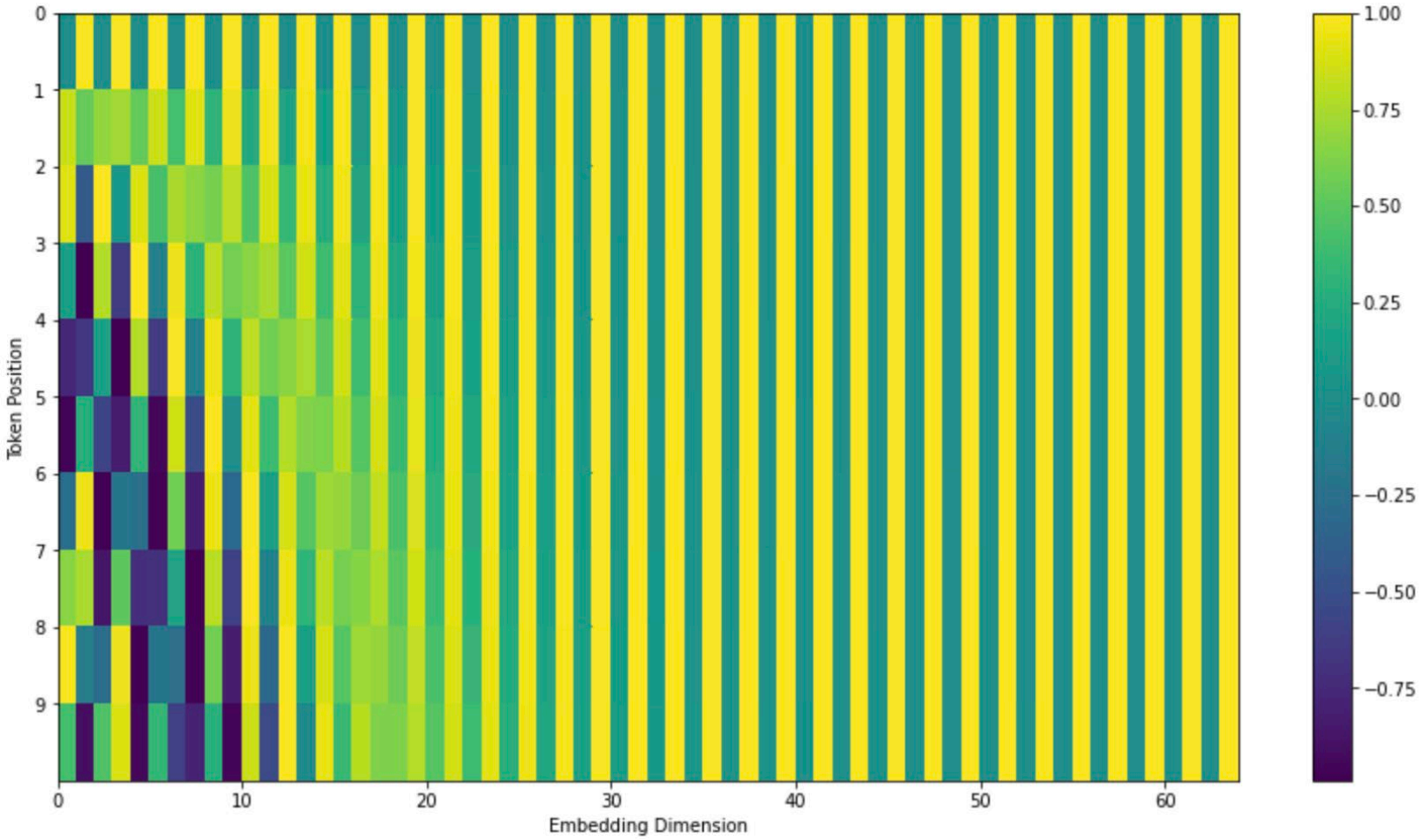
Positional encoding

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

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

```python
import math

class PositionalEncoder(torch.nn.Module):

    def __init__(self, d_model, max_seq_len=80):
        super().__init__()
        self.d_model = d_model

        # create constant positional encoding matrix
        pe_matrix = torch.zeros(max_seq_len, d_model)

        for pos in range(max_seq_len):
            for i in range(0, d_model, 2):
                pe_matrix[pos, i] = math.sin(pos/10000**(2*i/d_model))
                pe_matrix[pos, i+1] = math.cos(pos/10000**(2*i/d_model))
        pe_matrix = pe_matrix.unsqueeze(0)     # Add one dimension for batch size
        self.register_buffer('pe', pe_matrix)  # Register as persistent buffer

    def forward(self, x):
        # x is a sentence after embedding with dim (batch, number of words, vector dimension)
        seq_len = x.size()[1]
        x = x + self.pe[:, :seq_len]
        return x
```

POS Encoding Illustrated



LayerNorm and BatchNorm

- **LayerNorm** normalizes across the features of a single input vector:

$$\text{Input:} \quad \mathbf{x}_i = [x_{i1}, x_{i2}, x_{i3}, \ldots, x_{id}]$$

$$\text{Normalization:} \quad \mu_i = \frac{1}{d}\sum_{j=1}^{d} x_{ij}, \quad \sigma_i^2 = \frac{1}{d}\sum_{j=1}^{d}(x_{ij} - \mu_i)^2$$

$$\text{Output:} \quad y_{ij} = \gamma \cdot \frac{x_{ij} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} + \beta$$

- **BatchNorm** normalizes across the batch for each feature:

$$\text{Input:} \quad \mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]$$

$$\text{Normalization:} \quad \mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{ij}, \quad \sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{ij} - \mu_j)^2$$

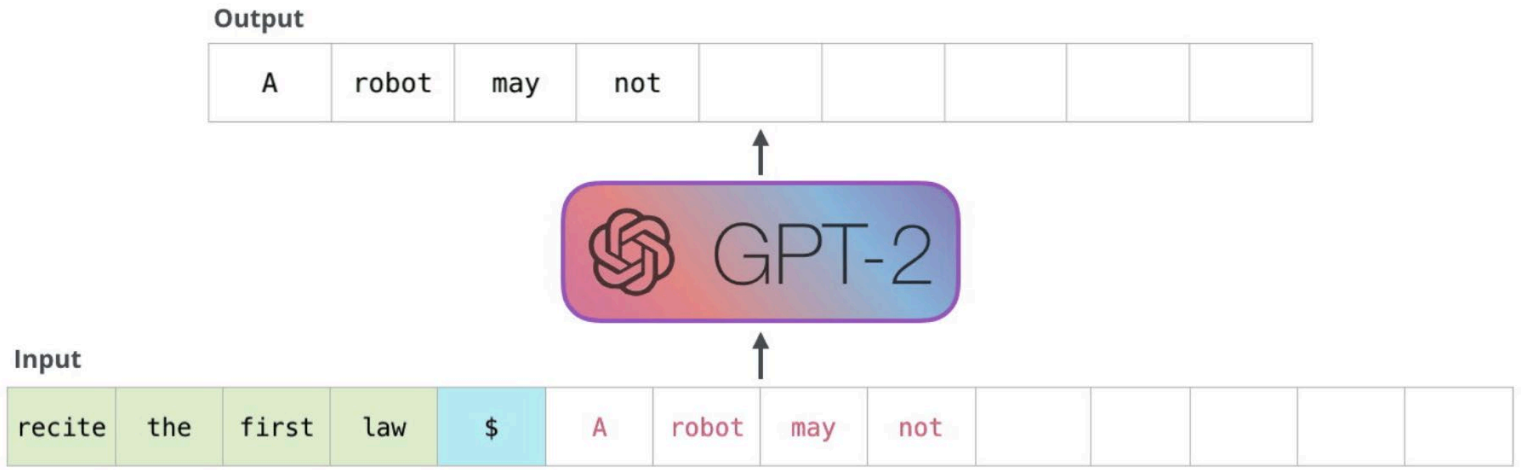
$$\text{Output:} \quad y_{ij} = \gamma \cdot \frac{x_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} + \beta$$

- There are different types of normalizations that are used when training a ML model.
  - Feature Normalization, Layer Normalization, Batch Normalization
- Transformers use Layer Normalization.
- Normalizations help improve the training stability and enable faster convergence

Example for illustrating Self Attention

Consider the text:

- First Law of robotics: A robot may not injure a human being or, through inaction, allow a human being to come to harm.
- Second law of robotics: A robot must obey the orders given to **it** by human beings except where **such orders** would conflict with the **First Law**.
- Example Prompt Instruction:



Example for illustrating Self Attention

- Correlations between tokens in a given sequence

- Multiple perspectives of looking at these correlations

- Representing the information in multiple levels of abstraction.

Consider the text:

- The constitution of India says: "India that is Bharat". Are Indians citizens of Bharat also?

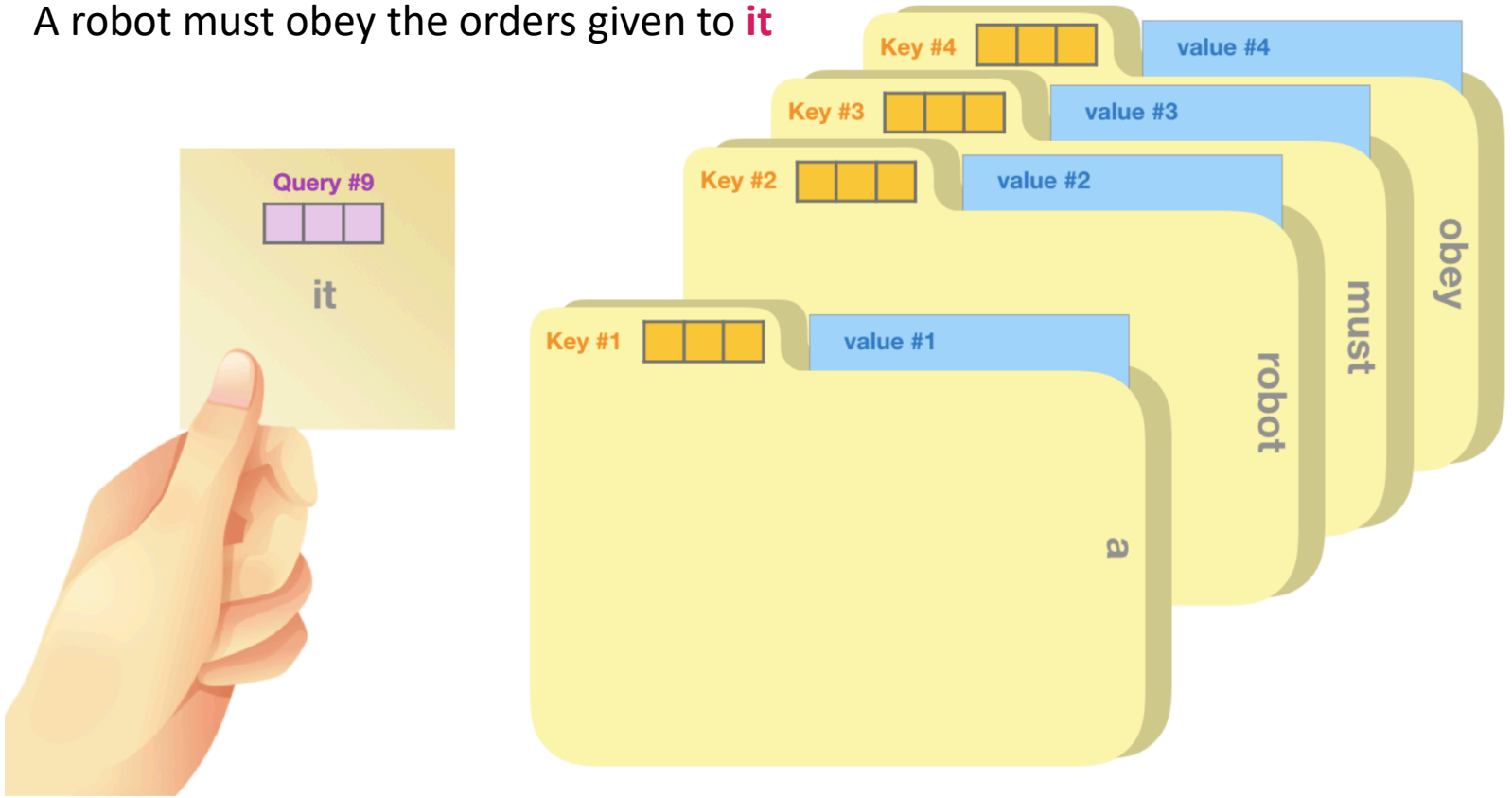
- To answer the above we need to understand multiple concepts, e.g. Indians are citizens of India, India and Bharat are interchangeable etc.

- How to learn and generalize across multiple concepts?

Query, Key, Value

- Query: The query is a representation of the current word used to score against all the other words (using their keys). We only care about the query of the token we're currently processing.
- Key: Key vectors are like labels for all the words in the segment. They're what we match against in our search for relevant words.
- Value: Value vectors are actual word representations, once we've scored how relevant each word is, these are the values we add up to represent the current word.

A robot must obey the orders given to **it**



Memory = {
$f_k(k_1)$: $f_v(v_1)$,
$f_k(k_2)$: $f_v(v_2)$,
....
}

Query = $f_q(k_i)$

Taking the dot product between query and memory, we can determine the weight of each token for the given query.

Feed Forward Network: Notice the linear projections

Feed Forward layer

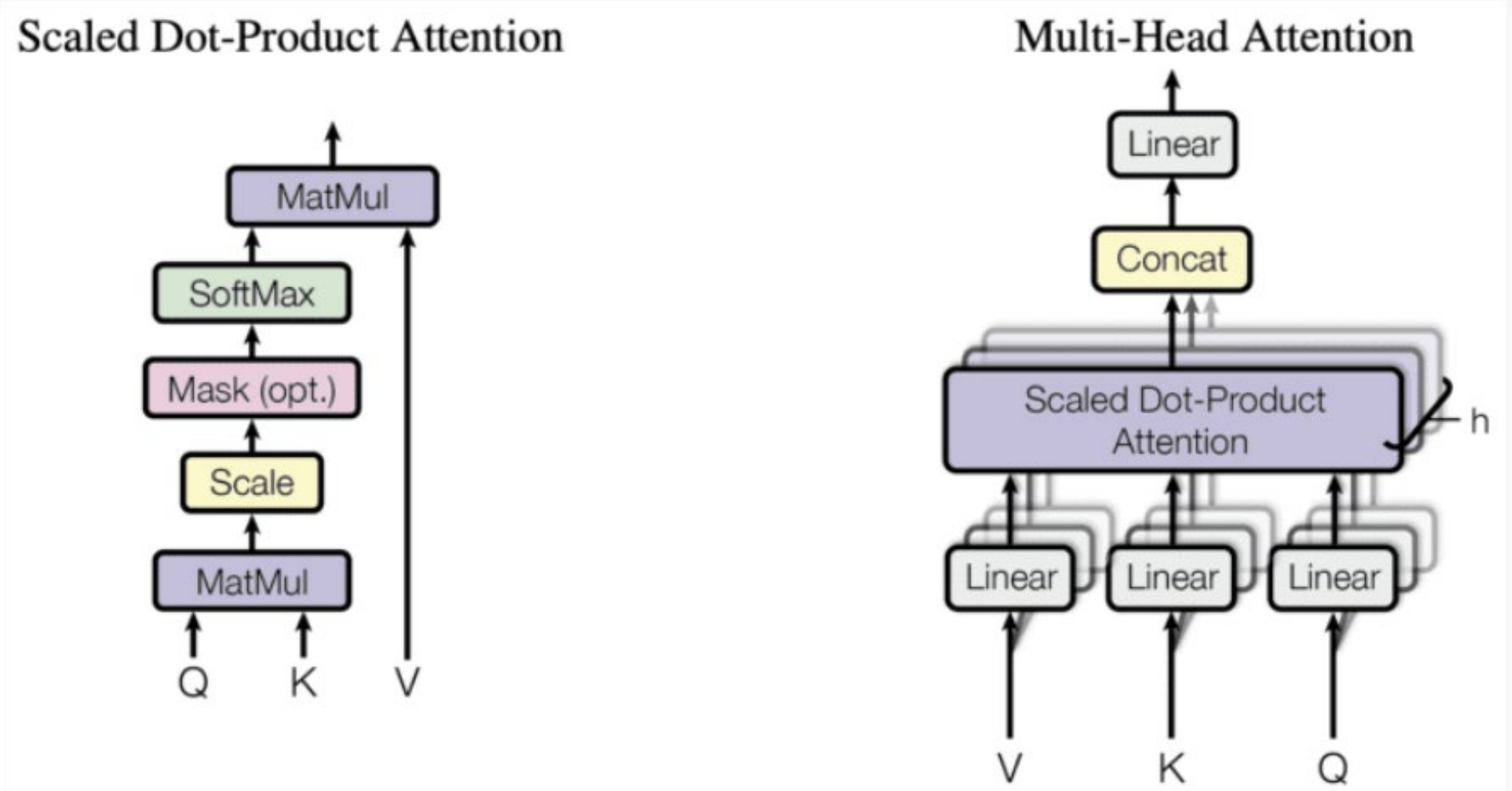
```
class FeedForward(torch.nn.Module):
    def __init__(self, d_model, d_ff=2048, dropout=0.1):
        super().__init__()

        self.linear_1 = torch.nn.Linear(d_model, d_ff)
        self.dropout = torch.nn.Dropout(dropout)
        self.linear_2 = torch.nn.Linear(d_ff, d_model)

    def forward(self, x):
        x = self.dropout(F.relu(self.linear_1(x)))
        x = self.linear_2(x)
        return x
```

Computing Self Attention

Q/K	hello	how	are	you
hello				
how				
are				
you				

- Our goal is to capture the correlations between each token with every other token in the given sequence
- We take each token, project it on to a query plane, key plane, value plane: That is, provide 3 different representations
- We compute the strength of each query vector with each key vector and scale the value vectors using this strength

Scaled Dot Product Attention



What is Self-Attention?

Fig Credits: Umar Jamil

D_k is 64 in the original paper

$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Self-Attention allows the model to relate words to each other.

In this simple case we consider the sequence length **seq** = 6 and $\mathbf{d_{model}} = \mathbf{d_k} = 512$.

The matrices **Q**, **K** and **V** are just the input sentence.

softmax(**Q** (6, 512) X **K^T** (512, 6) / $\sqrt{512}$) =

	YOUR	CAT	IS	A	LOVELY	CAT	Σ
YOUR	0.268	0.119	0.134	0.148	0.179	0.152	1
CAT	0.124	0.278	0.201	0.128	0.154	0.115	1
IS	0.147	0.132	0.262	0.097	0.218	0.145	1
A	0.210	0.128	0.206	0.212	0.119	0.125	1
LOVELY	0.146	0.158	0.152	0.143	0.227	0.174	1
CAT	0.195	0.114	0.203	0.103	0.157	0.229	1

(6, 6)

* all values are random.

* for simplicity I considered only one head, which makes $d_{model} = d_k$.

How to compute Self-Attention?

Fig Credits: Umar Jamil

$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

	YOUR	CAT	IS	A	LOVELY	CAT
YOUR	0.268	0.119	0.134	0.148	0.179	0.152
CAT	0.124	0.278	0.201	0.128	0.154	0.115
IS	0.147	0.132	0.262	0.097	0.218	0.145
A	0.210	0.128	0.206	0.212	0.119	0.125
LOVELY	0.146	0.158	0.152	0.143	0.227	0.174
CAT	0.195	0.114	0.203	0.103	0.157	0.229

(6, 6)

X

V

(6, 512)

=

Attention

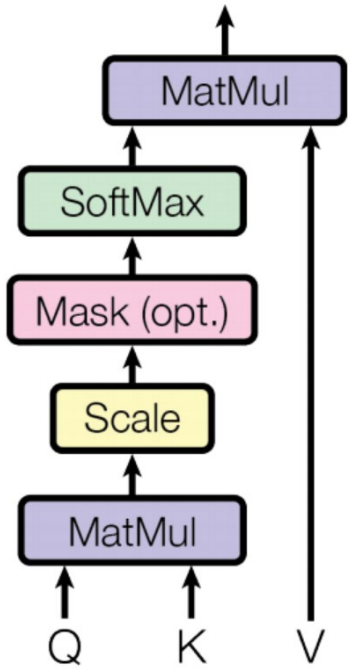
(6, 512)

Each row in this matrix captures not only the meaning (given by the embedding) or the position in the sentence (represented by the positional encodings) but also each word's interaction with other words.

Scaled Dot Product Attention

Model layers

Scaled Dot-Product Attention layer



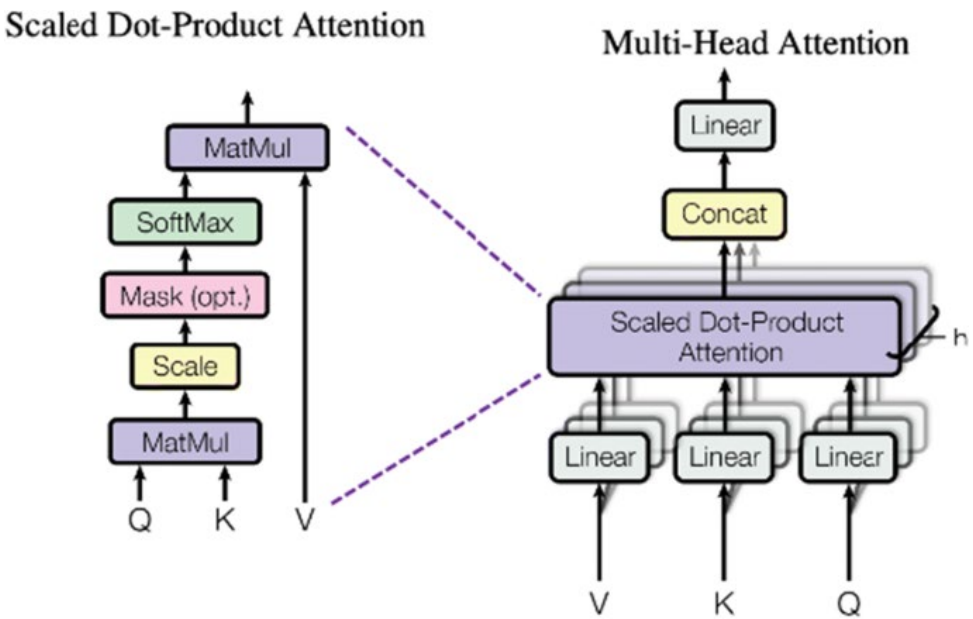

```python
import math
import torch.nn.functional as F

# Given Query, Key, Value, calculate the final weighted value
def scaled_dot_product_attention(q, k, v, mask=None, dropout=None):
    # Shape of q and k are the same, both are (batch_size, seq_len, d_k)
    # Shape of v is (batch_size, seq_len, d_v)
    attention_scores = torch.matmul(q, k.transpose(-2, -1))/math.sqrt(q.shape[-1])  # size (batch_size, seq_len, seq_len)

    # Apply mask to scores
    # <pad>
    if mask is not None:
        attention_scores = attention_scores.masked_fill(mask == 0, value=-1e9)

    # Softmax along the last dimension
    attention_weights = F.softmax(attention_scores, dim=-1)

    if dropout is not None:
        attention_weights = dropout(attention_weights)

    output = torch.matmul(attention_weights, v)
    return output
```

Multi-head Attention

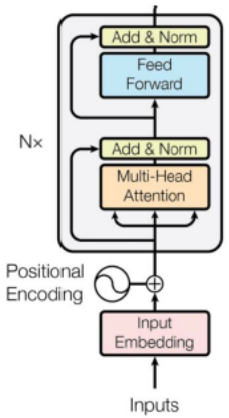
$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

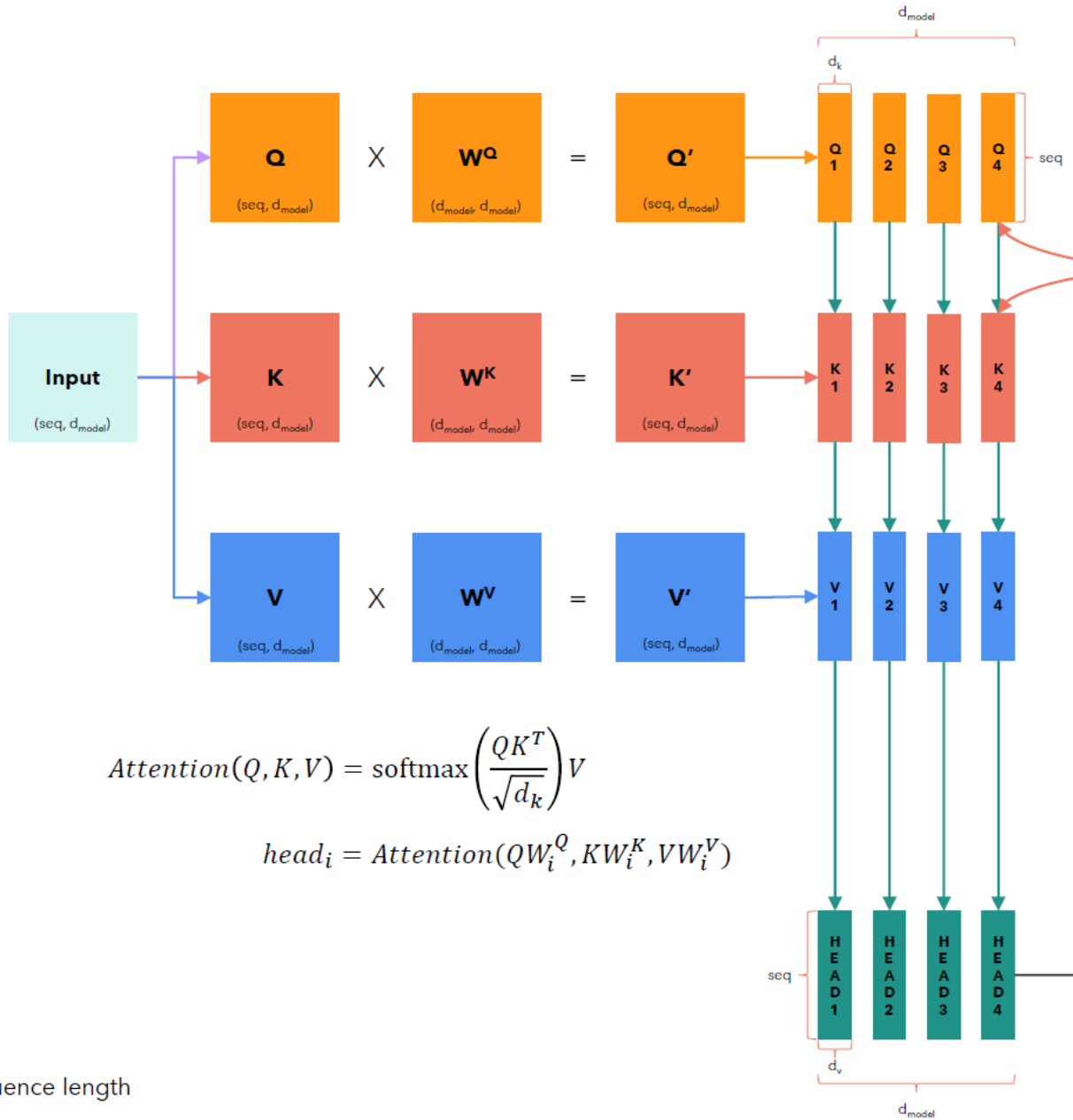
$$MultiHead(Q, K, V) = Concat(head_1 \ \ldots head_h)W^O$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

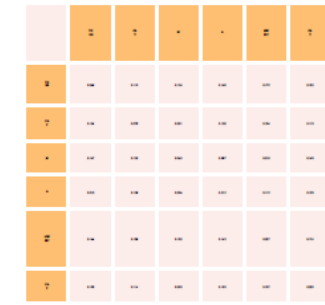



```
# Input: Sequence of token embeddings X
Q = XW^Q  # Queries
K = XW^K  # Keys
V = XW^V  # Values

# Attention scores
scores = Q @ K.T / sqrt(d_k)
attention_weights = softmax(scores)

# Weighted sum of values
context = attention_weights @ V

# Multi-head attention (multiple heads concatenated)
multi_head_output = concat([head1, head2, ..., headN]) @ W^O

# Feed-forward network
ffn_output = relu(multi_head_output @ W1 + b1) @ W2 + b2

# Add & Norm
output = LayerNorm(input + ffn_output)
```

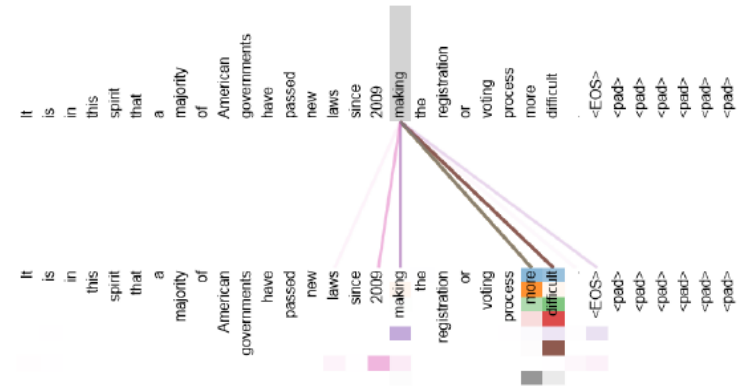




$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) =$$



Attention Visualizations



$$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

$$MultiHead(Q,K,V) = Concat(head_1 \ldots head_h)W^O$$

seq = sequence length

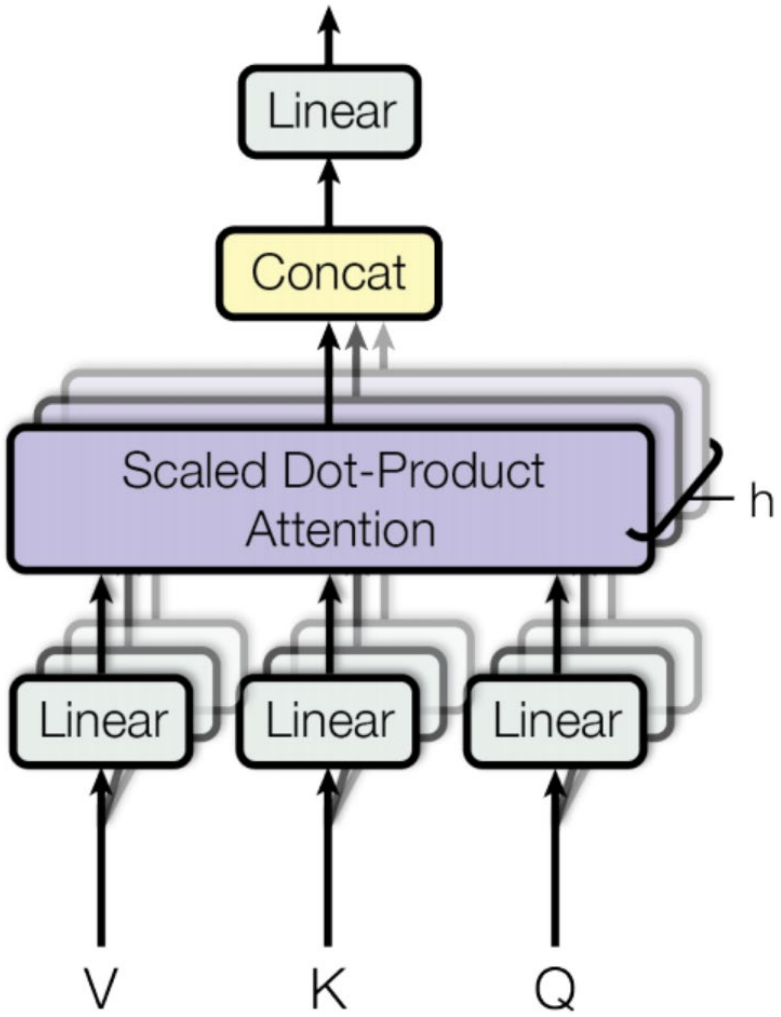
d_{model} = size of the embedding vector

h = number of heads

$d_k = d_v$ = d_{model} / h

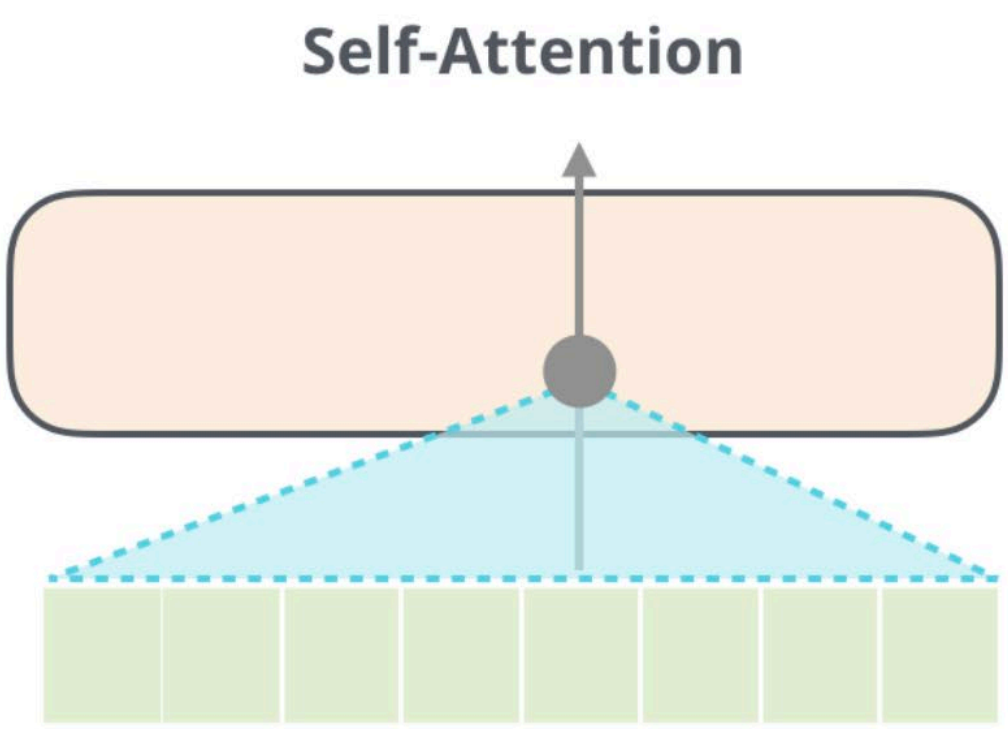
Umar Jamil – https://github.com/hkproj/transformer-from-scratch-notes

Multi Head Attention

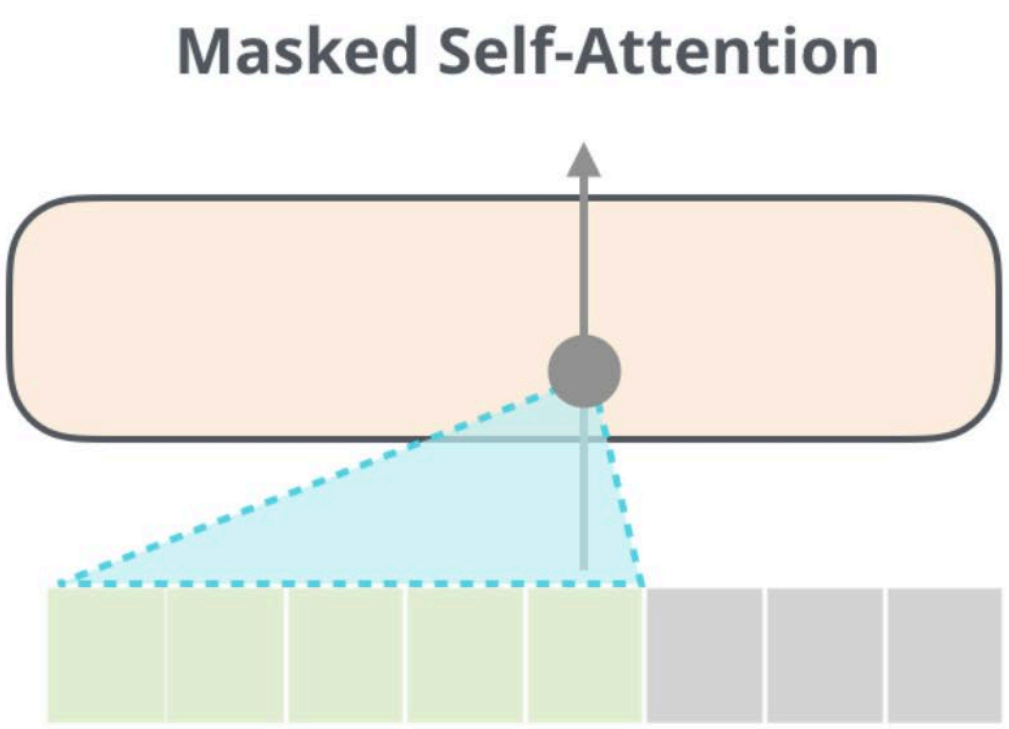



```python
class MultiHeadAttention(torch.nn.Module):
    def __init__(self, n_heads, d_model, dropout=0.1):
        super().__init__()

        self.n_heads = n_heads
        self.d_model = d_model
        self.d_k = self.d_v = d_model//n_heads

        # self attention linear layers
        # Linear layers for q, k, v vectors generation in different heads
        self.q_linear_layers = []
        self.k_linear_layers = []
        self.v_linear_layers = []
        for i in range(n_heads):
            self.q_linear_layers.append(torch.nn.Linear(d_model, self.d_k))
            self.k_linear_layers.append(torch.nn.Linear(d_model, self.d_k))
            self.v_linear_layers.append(torch.nn.Linear(d_model, self.d_v))

        self.dropout = torch.nn.Dropout(dropout)
        self.out = torch.nn.Linear(n_heads*self.d_v, d_model)

    def forward(self, q, k, v, mask=None):
        multi_head_attention_outputs = []
        for q_linear, k_linear, v_linear in zip(self.q_linear_layers,
                                                self.k_linear_layers,
                                                self.v_linear_layers):
            new_q = q_linear(q)  # size: (batch_size, seq_len, d_k)
            new_k = k_linear(k)  # size: (batch_size, seq_len, d_k)
            new_v = v_linear(v)  # size: (batch_size, seq_len, d_v)

            # Scaled Dot-Product attention
            head_v = scaled_dot_product_attention(new_q, new_k, new_v, mask, self.dropout)  # (batch_size, seq_len, d_v)
            multi_head_attention_outputs.append(head_v)

        # Concat
        #import pdb; pdb.set_trace()
        concat = torch.cat(multi_head_attention_outputs, -1)  # (batch_size, seq_len, n_heads*d_v)

        # Linear layer to recover to original shap
        output = self.out(concat)  # (batch_size, seq_len, d_model)

        return output
```

Masked Self Attention for Decoder



How are you today?



How are you today?
How are you today?
How are you today?

Feed Forward Network

Feed Forward layer

```
class FeedForward(torch.nn.Module):
    def __init__(self, d_model, d_ff=2048, dropout=0.1):
        super().__init__()

        self.linear_1 = torch.nn.Linear(d_model, d_ff)
        self.dropout = torch.nn.Dropout(dropout)
        self.linear_2 = torch.nn.Linear(d_ff, d_model)

    def forward(self, x):
        x = self.dropout(F.relu(self.linear_1(x)))
        x = self.linear_2(x)
        return x
```

Layer Normalization layer

Normalization

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2$$

$$\hat{Z}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

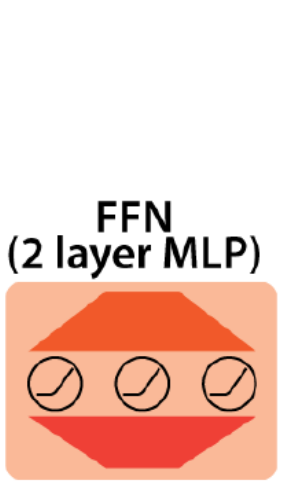
Add two learnable parameters

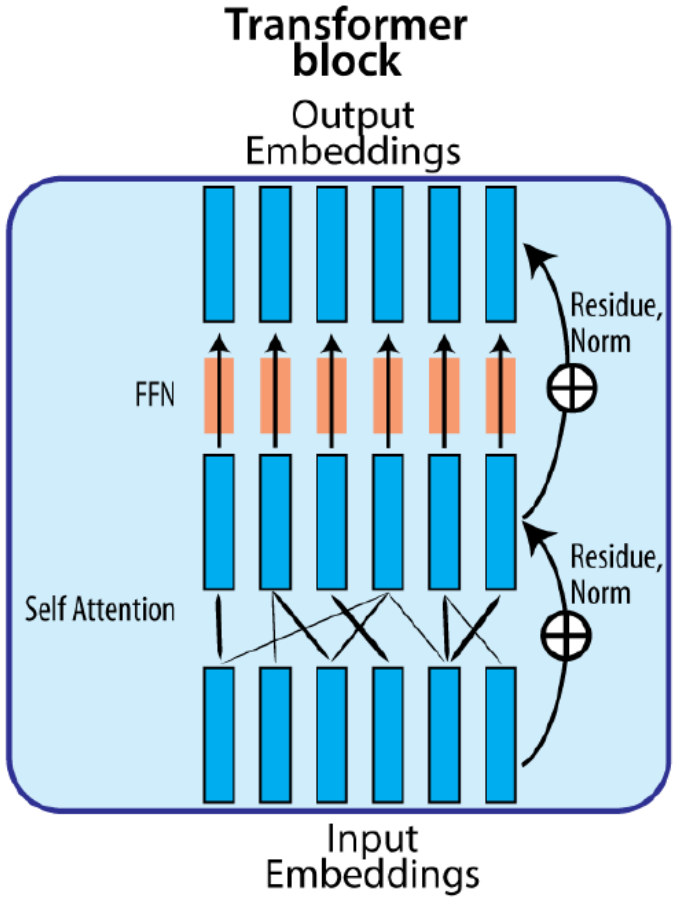
$$\tilde{Z}_i = \alpha_i * \hat{Z}_i + \beta_i$$

```
class LayerNorm(torch.nn.Module):
    def __init__(self, d_model, eps=1e-6):
        super().__init__()
        self.d_model = d_model
        self.alpha = torch.nn.Parameter(torch.ones(self.d_model))
        self.beta = torch.nn.Parameter(torch.zeros(self.d_model))
        self.eps = eps

    def forward(self, x):
        # x size: (batch_size, seq_len, d_model)
        x_hat = (x - x.mean(dim=-1, keepdim=True))/(x.std(dim=-1, keepdim=True) + self.eps)
        x_tilde = self.alpha*x_hat + self.beta
        return x_tilde
```

From Attention to **TransformerBlock**

- A **transformer block** has
  - Self Attention
    - information exchange ***between tokens***
  - Feed forward network
    - Information transform ***within tokens***
    - E.g. a multi-layer perceptron with 1 hidden layer
    - GeLU activation is commonly used.
  - Normalization (Layer normalization)
- A transformer model contains N x **transformer block**





GELU is an activation function: Gaussian Error Linear Unit

Rationale

- Why multiple encoder/decoder blocks?
- Why many attention heads?
- Why self attention?
- Why cross attention?

Variations from base architecture and performance

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
(A)				1	512	512				5.29	24.9	
				4	128	128				5.00	25.5	
				16	32	32				4.91	25.8	
				32	16	16				5.01	25.4	
(B)					16					5.16	25.1	58
					32					5.01	25.4	60
(C)	2									6.11	23.7	36
	4									5.19	25.3	50
	8									4.88	25.5	80
		256			32	32				5.75	24.5	28
		1024			128	128				4.66	26.0	168
			1024							5.12	25.4	53
			4096							4.75	26.2	90
(D)							0.0			5.77	24.6	
							0.2			4.95	25.5	
								0.0		4.67	25.3	
								0.2		5.47	25.7	
(E)	positional embedding instead of sinusoids									4.92	25.7	
big	6	1024	4096	16			0.3		300K	**4.33**	**26.4**	213

BERT, GPT Architectural Parameters

1. **BERT-Base:**
   - Number of layers: 12 (transformer layers)
   - Hidden size (dimension of the encoder layers): 768
   - Number of attention heads: 12
   - Total parameters: Approximately 110 million parameters
2. **BERT-Large:**
   - Number of layers: 24 (transformer layers)
   - Hidden size (dimension of the encoder layers): 1024
   - Number of attention heads: 16
   - Total parameters: Approximately 340 million parameters

	GPT-1	GPT-2	GPT-3
Parameters	117 Million	1.5 Billion	175 Billion
Decoder Layers	12	48	96
Context Token Size	512	1024	2048

Model specifications Our model largely follows the original transformer work [62]. We trained a 12-layer decoder-only transformer with masked self-attention heads (768 dimensional states and 12 attention heads). For the position-wise feed-forward networks, we used 3072 dimensional inner states. We used the Adam optimization scheme [27] with a max learning rate of 2.5e-4. The learning rate was increased linearly from zero over the first 2000 updates and annealed to 0 using a cosine schedule. We train for 100 epochs on minibatches of 64 randomly sampled, contiguous sequences of 512 tokens. Since layernorm [2] is used extensively throughout the model, a simple weight initialization of $N(0, 0.02)$ was sufficient. We used a bytepair encoding (BPE) vocabulary with 40,000 merges [53] and residual, embedding, and attention dropouts with a rate of 0.1 for regularization. We also employed a modified version of L2 regularization proposed in [37], with $w = 0.01$ on all non bias or gain weights. For the activation function, we used the Gaussian Error Linear Unit (GELU) [18]. We used learned position embeddings instead of the sinusoidal version proposed in the original work. We use the *ftfy* library[2] to clean the raw text in BooksCorpus, standardize some punctuation and whitespace, and use the *spaCy* tokenizer.[3]

Part 3: Transformers for Generative AI

BERT

- Bidirectional Encoder Representation from Transformers

- Two strategies to apply pretrained language representations to downstream tasks:
  - Feature-based
  - Finetuning

- BERT improves upon finetuning approaches using 2 objectives:
  - Masked Language Model (MLM)
  - Next Sentence Prediction (NSP)

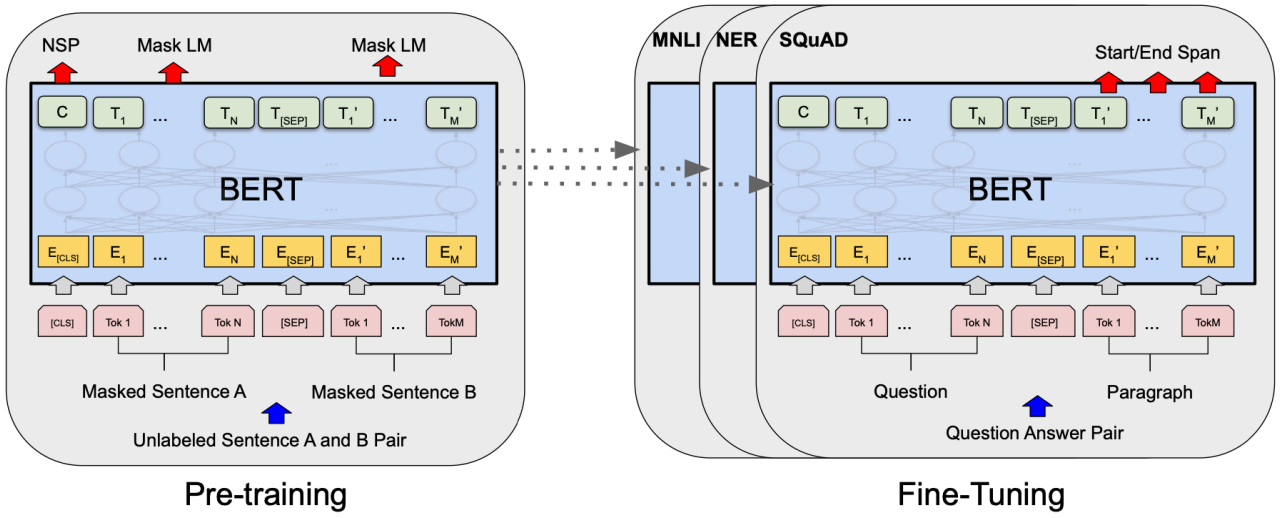


Figure 1: Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, all parameters are fine-tuned. `[CLS]` is a special symbol added in front of every input example, and `[SEP]` is a special separator token (e.g. separating questions/answers).

BERT: Vocabulary and Representation

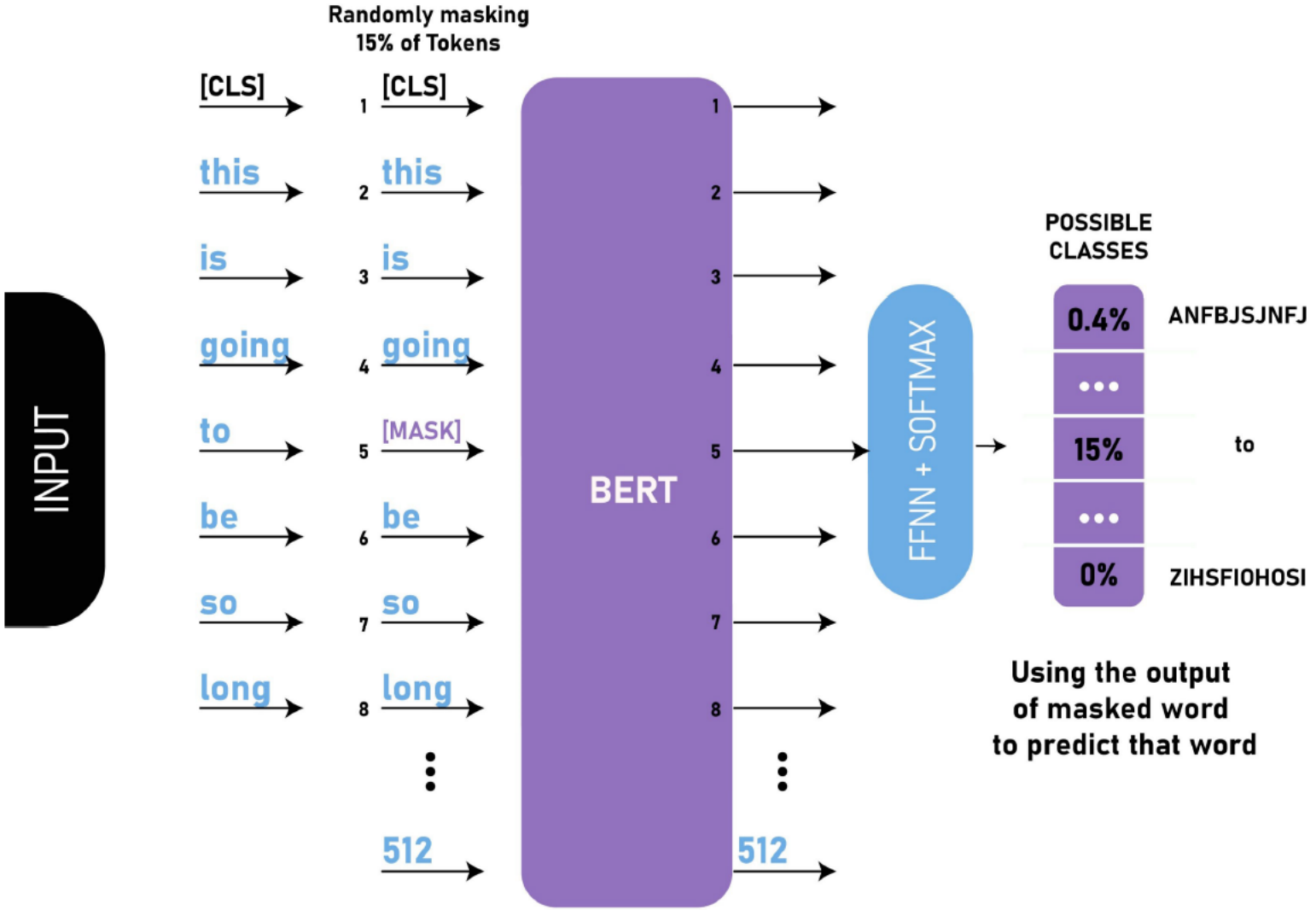
- BERT uses WordPiece embeddings (Wu et al., 2016) with a 30,000 token vocabulary.
- The first token of every sequence is always a special classification token ([CLS]).
- The final hidden state corresponding to this token is used as the aggregate sequence representation for classification tasks.
- Sentence pairs are packed together into a single sequence.
- Sentences are separated in two ways. First, with a special token ([SEP]). Second, using a learned embedding to every token indicating whether it belongs to sentence A or sentence B.

Masked word prediction / Cloze task:

BERT = Bidirectional Encoder Representations from Transformers

Fig Credits: Binxu Wang, Harvard

- Mask out 15% tokens and predict those [MASK].
- **Bidirectional:** Predict the mask token from context on both sides .
- Transformer **Encoder,** all-to-all self-attention
- Suitable for text **representation**



BERT Masked Language Model Training

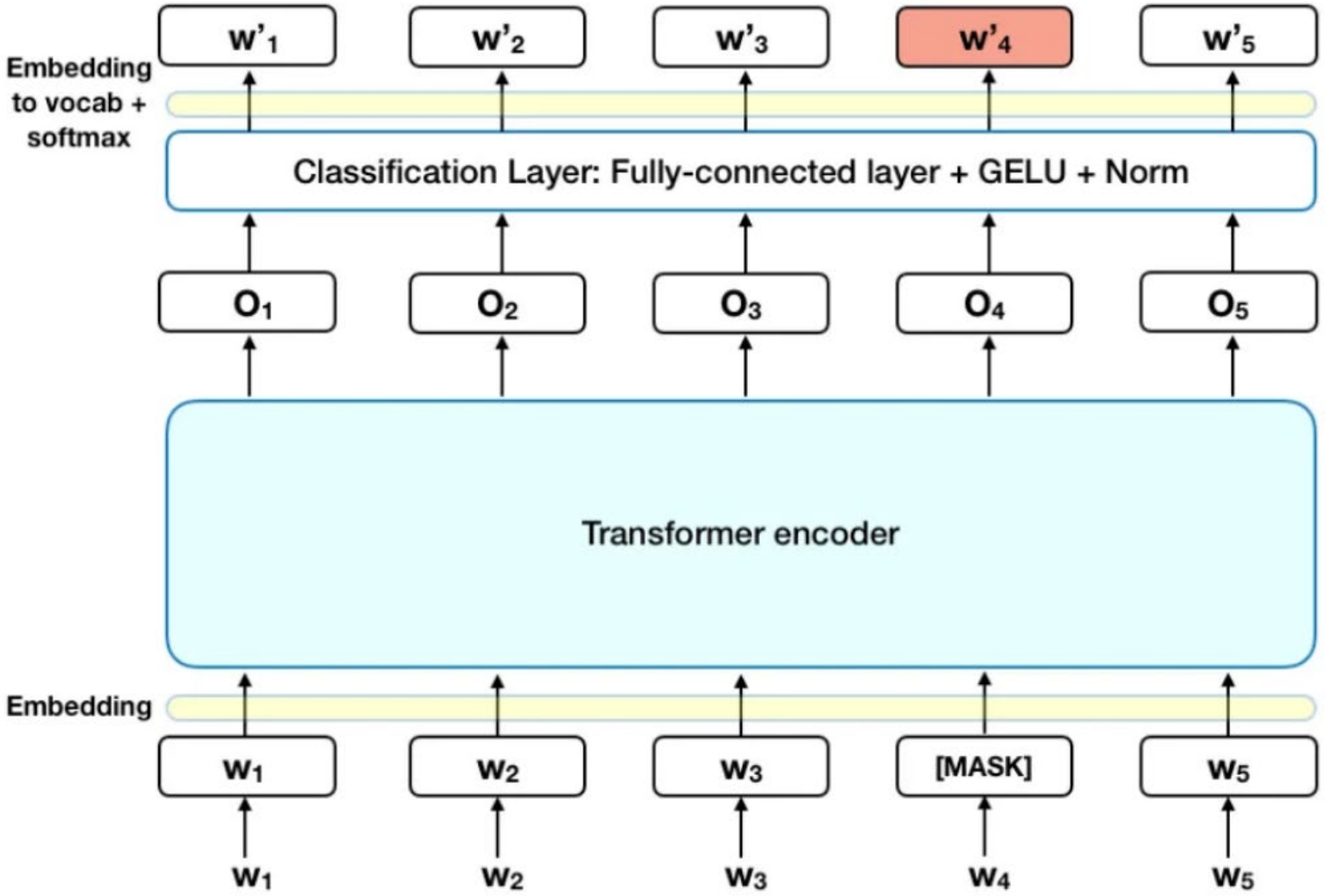


Fig Credits: Binxu Wang, Harvard

Next Sentence Prediction

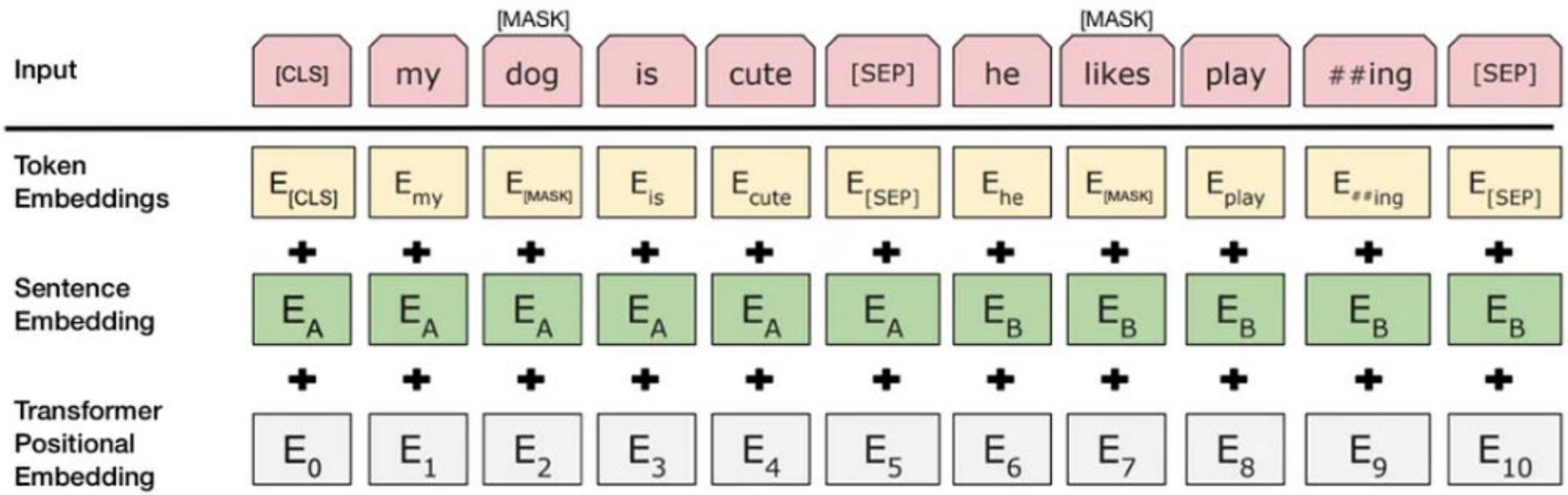
- In the BERT training process, the model receives pairs of sentences as input
- Model learns to predict if the second sentence in the pair is the subsequent sentence in the original document.
- During training, 50% of the inputs are a pair in which the second sentence is the subsequent sentence in the original document, while in the other 50% a random sentence from the corpus is chosen as the second sentence. We assume that the random sentence is not subsequent to the first sentence.

Next sentence Prediction

To help the model distinguish between the two sentences in training, the input is processed in the following way before entering the model:

1. A [CLS] token is inserted at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
2. A sentence embedding indicating Sentence A or Sentence B is added to each token. Sentence embeddings are similar in concept to token embeddings with a vocabulary of 2.
3. A positional embedding is added to each token to indicate its position in the sequence.

Next Sentence Prediction Architecture



Source: BERT [Devlin et al., 2018], with modifications

BERT Training

To predict if the second sentence is indeed connected to the first, the following steps are performed:

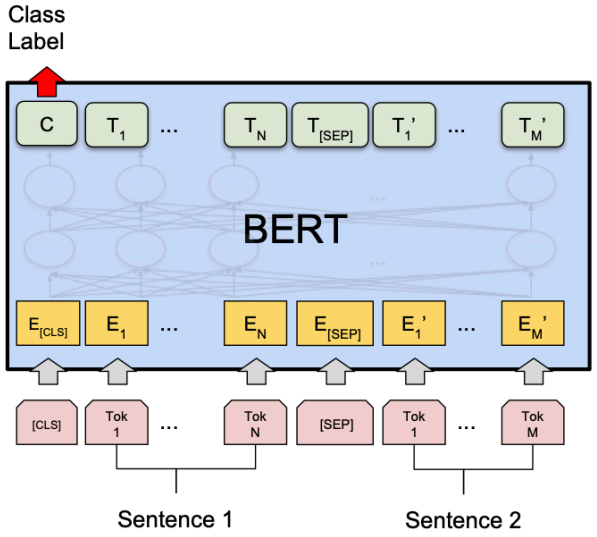
- The entire input sequence goes through the Transformer model.

- The output of the [CLS] token is transformed into a 2×1 shaped vector, using a simple classification layer (learned matrices of weights and biases).

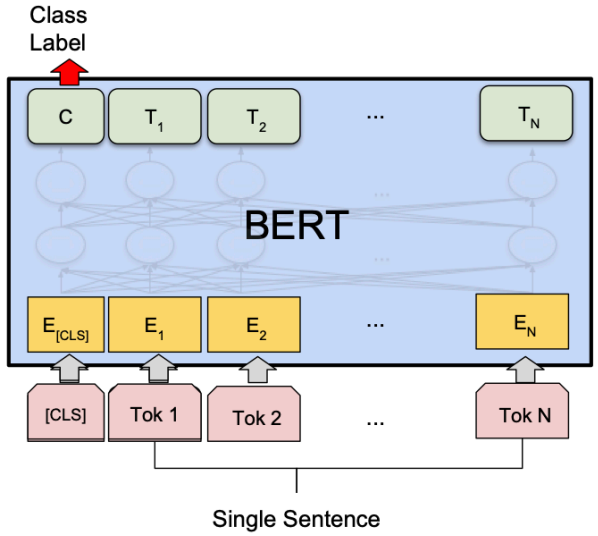
- Calculating the probability of IsNextSequence with softmax.

- 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.

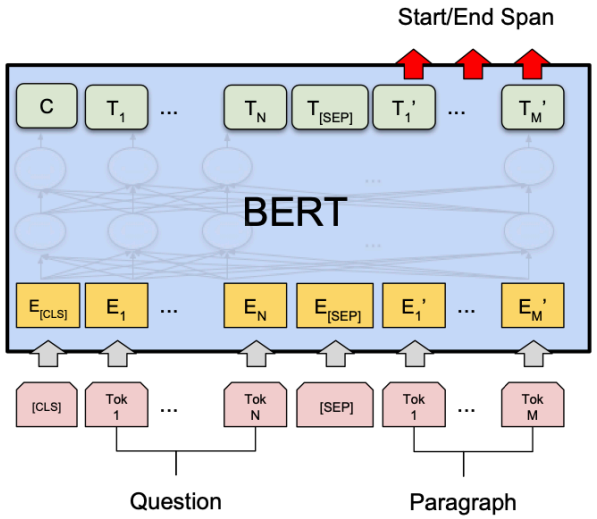
BERT Fine-tuning on different tasks

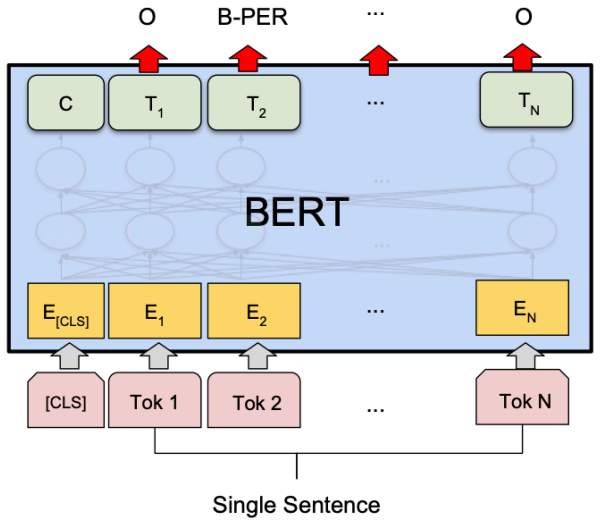


(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG



(b) Single Sentence Classification Tasks: SST-2, CoLA





MNLI: Multi Genre NL Inference
QQP: Quora Question Pair
QNLI: Question Natural Language Inference
SST-2: Stanford Sentiment Treebank
CoLA: Corpus of Linguistic Acceptability
STS-B: Semantic Textual Similarity Benchmark
MRPC: Microsoft Research Paraphrase Corpus
RTE: Recognizing Textual Entailment
WNLI: Winograd NLI
SQuAD: Stanford Question Answering dataset
CoNLL 2003 NER: Cornell University Named Entity Recognition Dataset

BERT Key Insights

- Unified architecture across multiple tasks
- Compact representation of vocabulary using Wordpiece representation
- Results have shown that finetuning is relatively inexpensive, transfer learning is shown to be very effective. (A few hours of GPU time)
- Larger models lead to a strict accuracy improvement across all the datasets used during training

It has long been known that increasing the model size will lead to continual improvements on large-scale tasks such as machine translation and language modeling, which is demonstrated by the LM perplexity of held-out training data shown in Table 6. However, we believe that this is the first work to demonstrate convincingly that scaling to extreme model sizes also leads to large improvements on very small scale tasks, provided that the model has been sufficiently pre-trained. Peters et al. (2018b) presented

Usage Examples

```
>>> from transformers import pipeline
>>> unmasker = pipeline('fill-mask', model='bert-base-uncased')
>>> unmasker("Hello I'm a [MASK] model.")

[{'sequence': "[CLS] hello i'm a fashion model. [SEP]",
  'score': 0.1073106899857521,
  'token': 4827,
  'token_str': 'fashion'},
 {'sequence': "[CLS] hello i'm a role model. [SEP]",
  'score': 0.08774490654468536,
  'token': 2535,
  'token_str': 'role'},
 {'sequence': "[CLS] hello i'm a new model. [SEP]",
  'score': 0.05338378623127937,
  'token': 2047,
  'token_str': 'new'},
 {'sequence': "[CLS] hello i'm a super model. [SEP]",
  'score': 0.04667217284440994,
  'token': 3565,
  'token_str': 'super'},
 {'sequence': "[CLS] hello i'm a fine model. [SEP]",
  'score': 0.027095865458250046,
  'token': 2986,
  'token_str': 'fine'}]
```

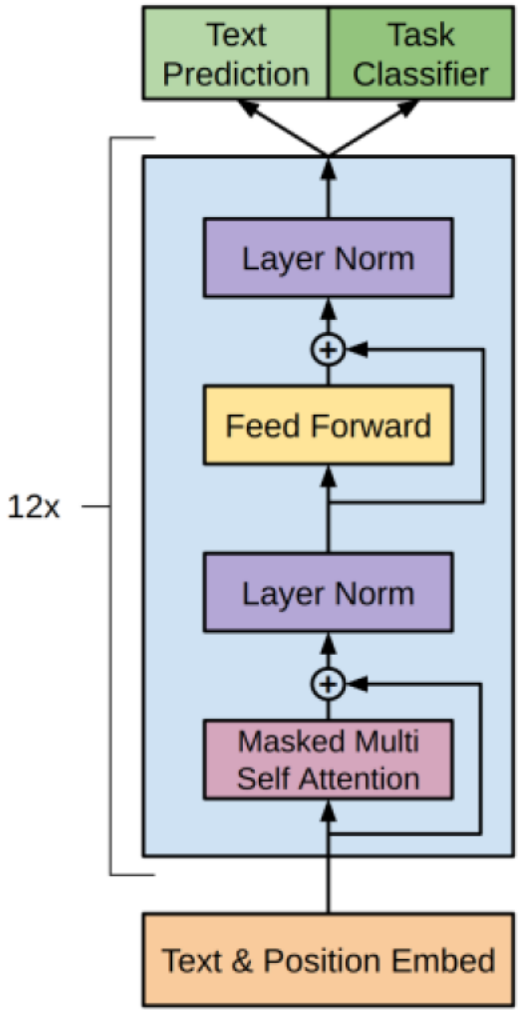
Here is how to use this model to get the features of a given text in PyTorch:

```
from transformers import BertTokenizer, BertModel
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained("bert-base-uncased")
text = "Replace me by any text you'd like."
encoded_input = tokenizer(text, return_tensors='pt')
output = model(**encoded_input)
```

By passing labels for different tasks, Hugging Face Transformer based models allow inferencing for various tasks

GPT architecture

- Learned text & position embedding
- Transformer block with causal masked self-attention
- *Optional cross attention* module for adding conditional info.

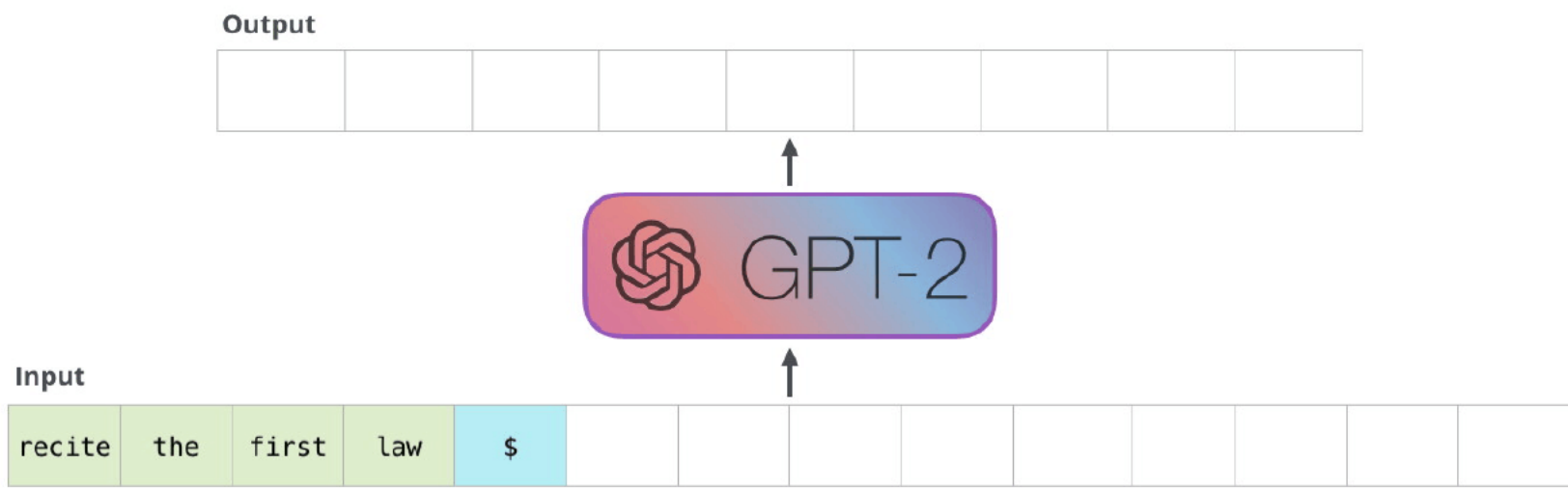


GPT1
Radford, A., Narasimhan, K., Salimans, T. and Sutskever, I., 2018. Improving language understanding by generative pre-training.

Next word prediction:
GPT = Generative Pre-trained Transformer

- **Pre-trained:** Each word predict the next word considering all previous words
  - Enforcing causal attention.
  - Enabling text generation

- **Generative:** Autoregressive model, suitable for text generation.



The Illustrated GPT-2

How does the decoding happen in a GPT?

- It is essential to understand how the instruction following GPT works.
- Input sequences (prompt) is processed, self attentions computed and abstract representations generated
- Till the prompt is processed and understood, the decoder doesn’t emit the response
- Once prompt is processed, decoder emits the response in an auto regressive way
- Some models use explicit special tokens to delimit prompt and the response, while ChatGPT or GPT 4 do this demarcation implicitly.

GPT Finetuning

- GPT training procedure consists of two stages:
  - The first stage is learning a high-capacity language model on a large corpus of text.
  - In the second fine-tuning stage, GPT is adapted to a discriminative task with labeled data.

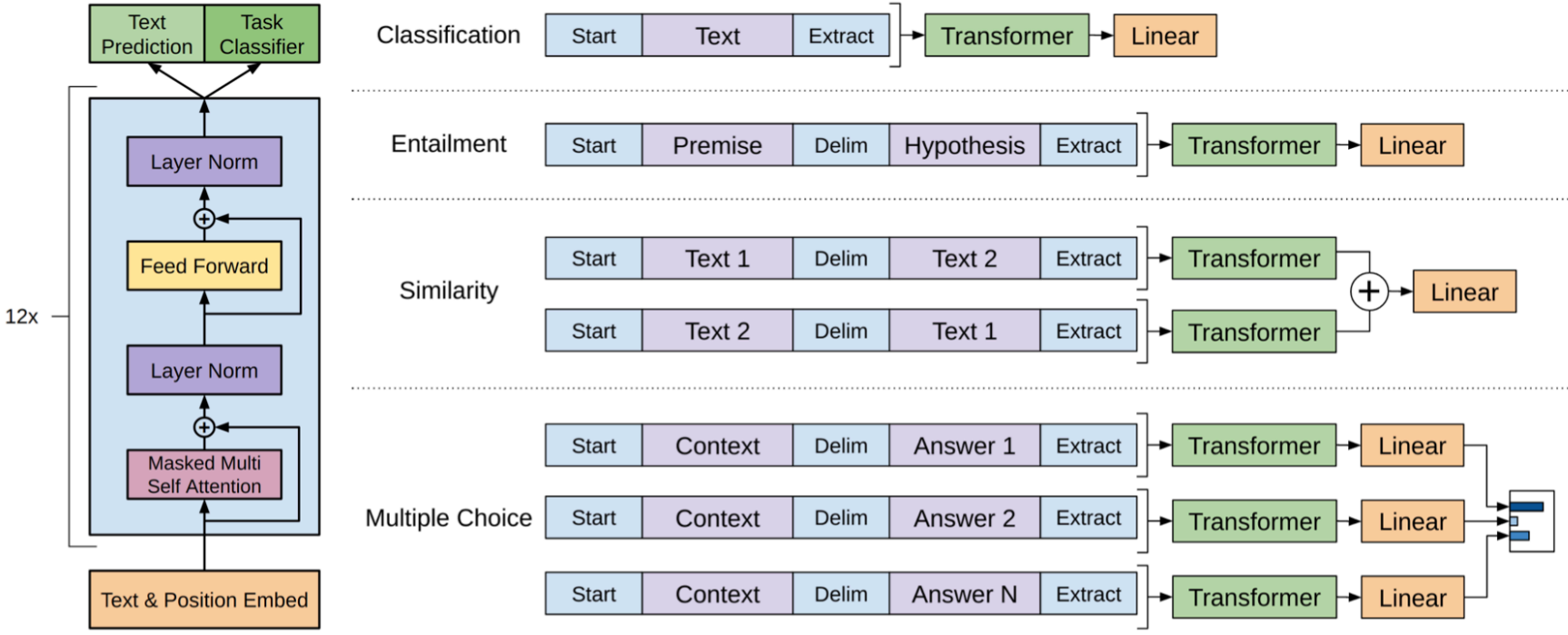
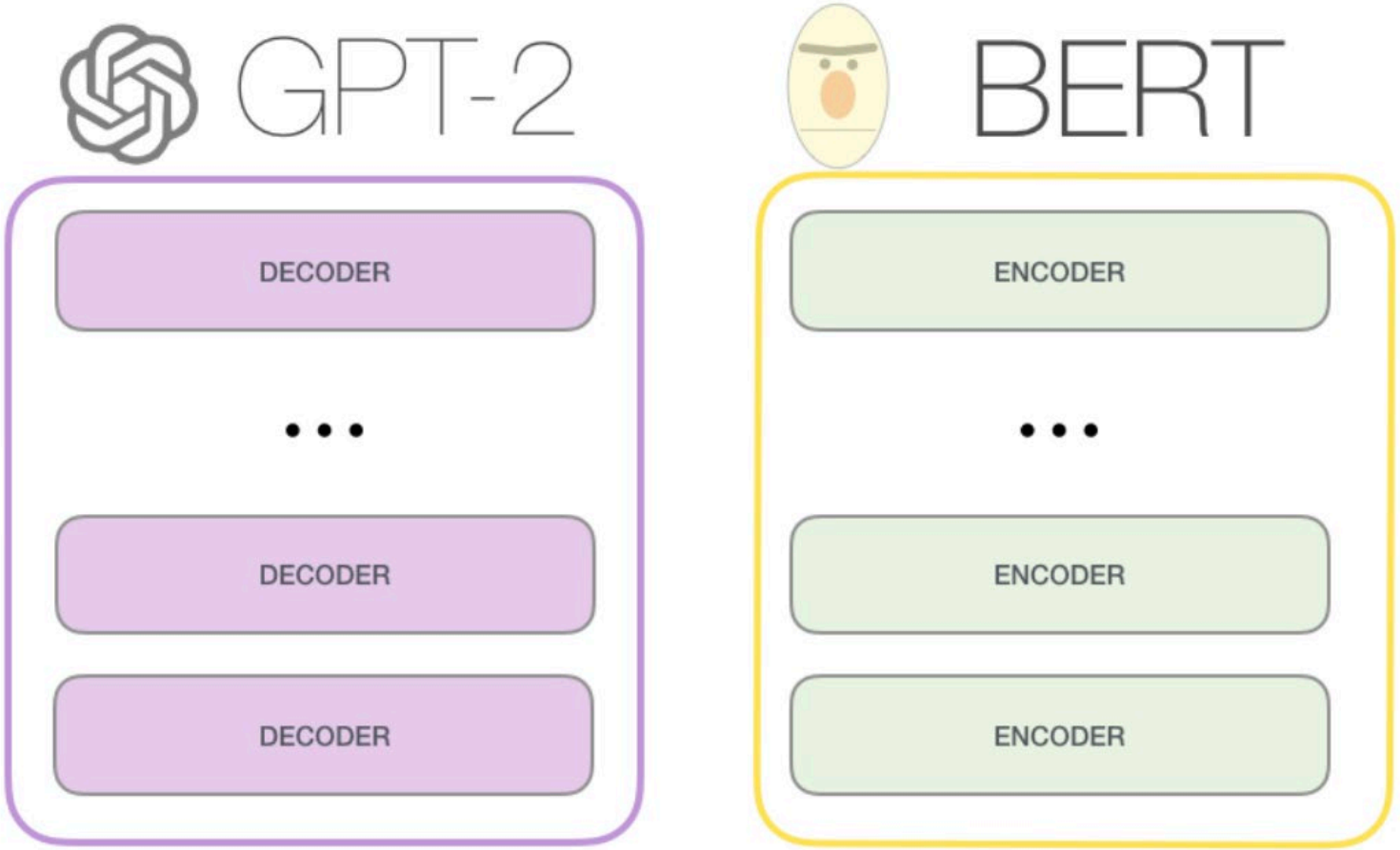


Figure 1: **(left)** Transformer architecture and training objectives used in this work. **(right)** Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

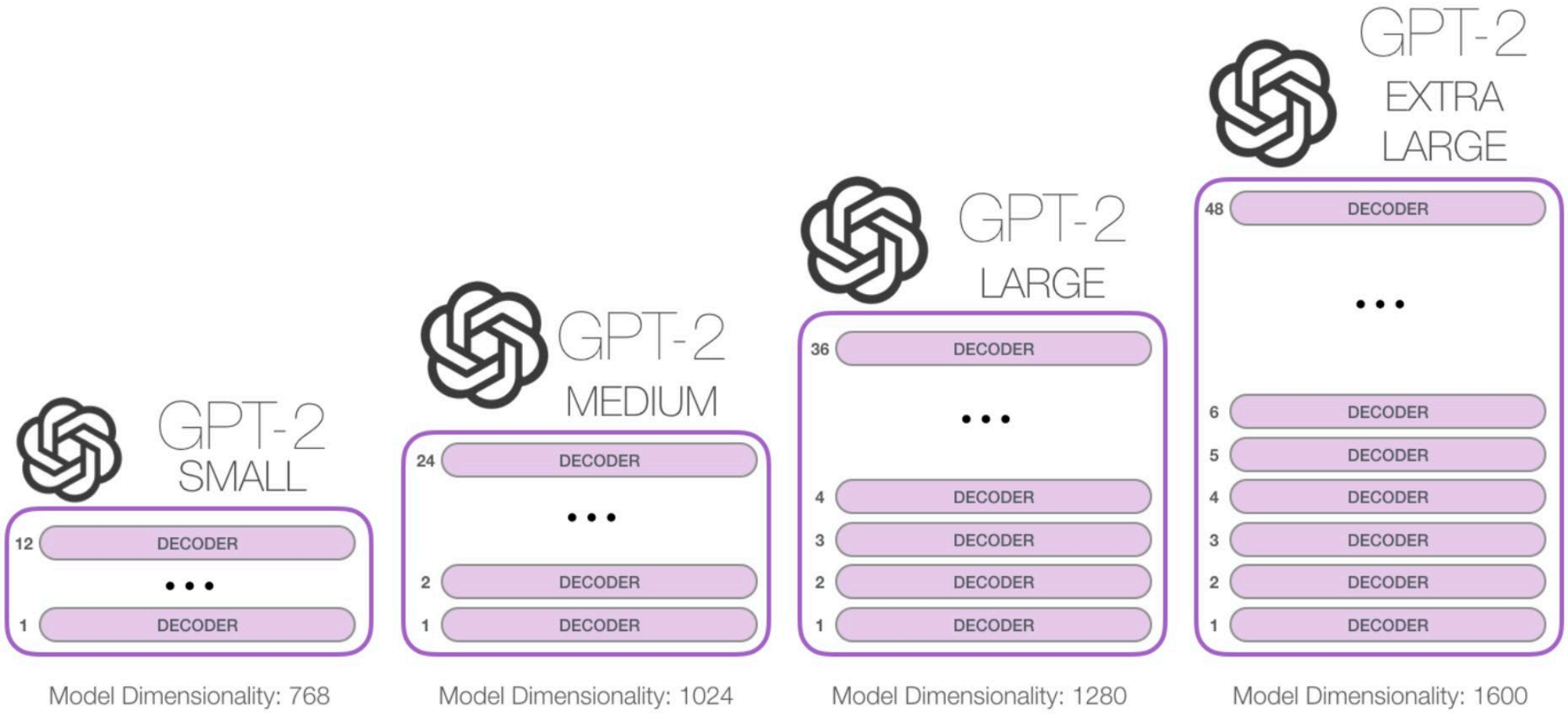
GPT versus BERT Approaches

Fig Credits: Jay Alammar

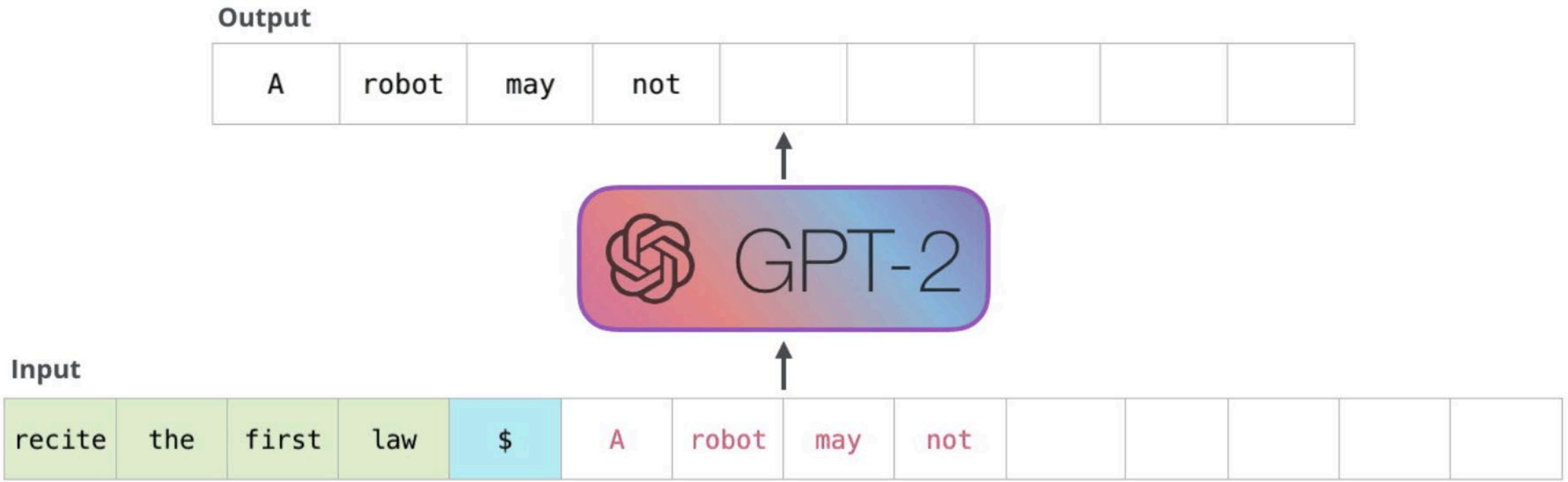


GPT-2 Variants

Fig Credits: Jay Alammar

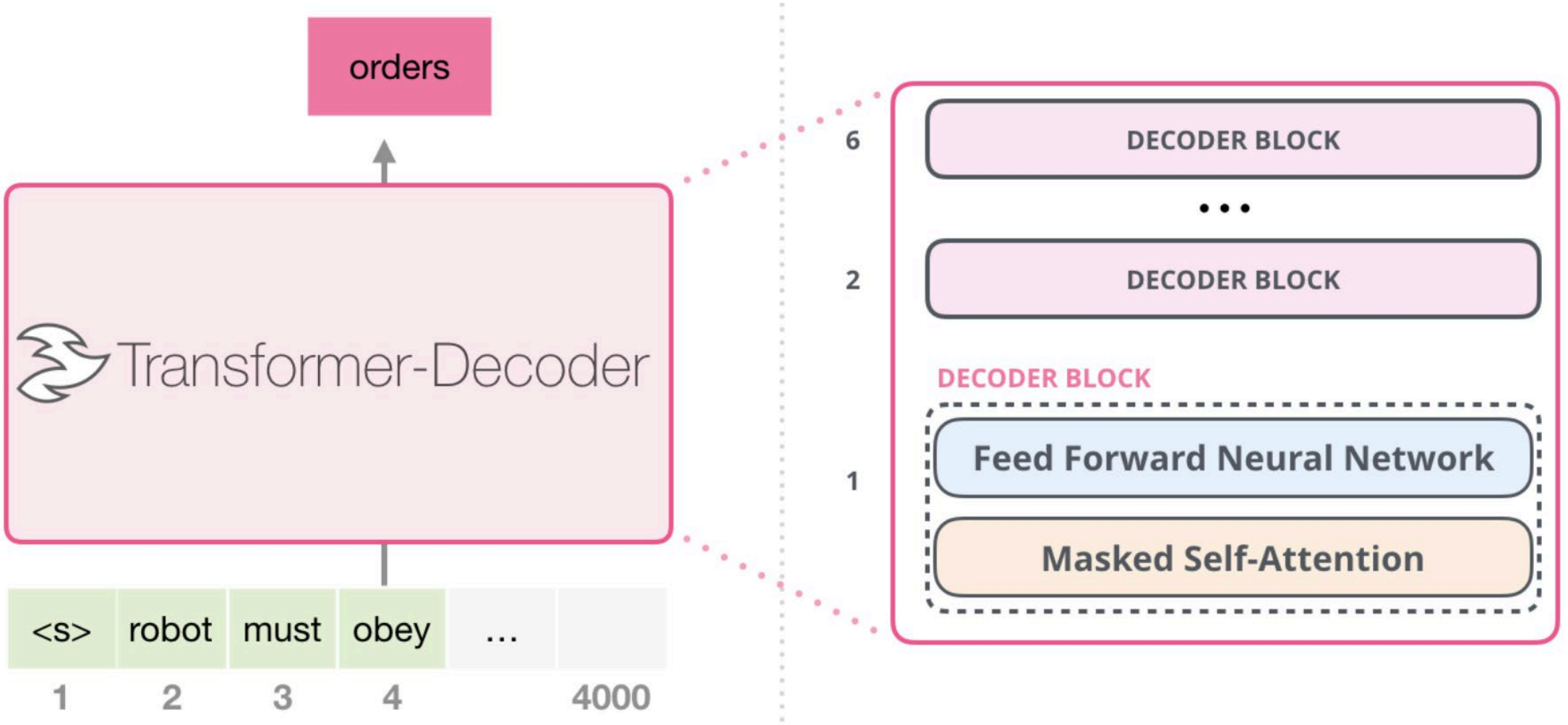


Decoding in GPT



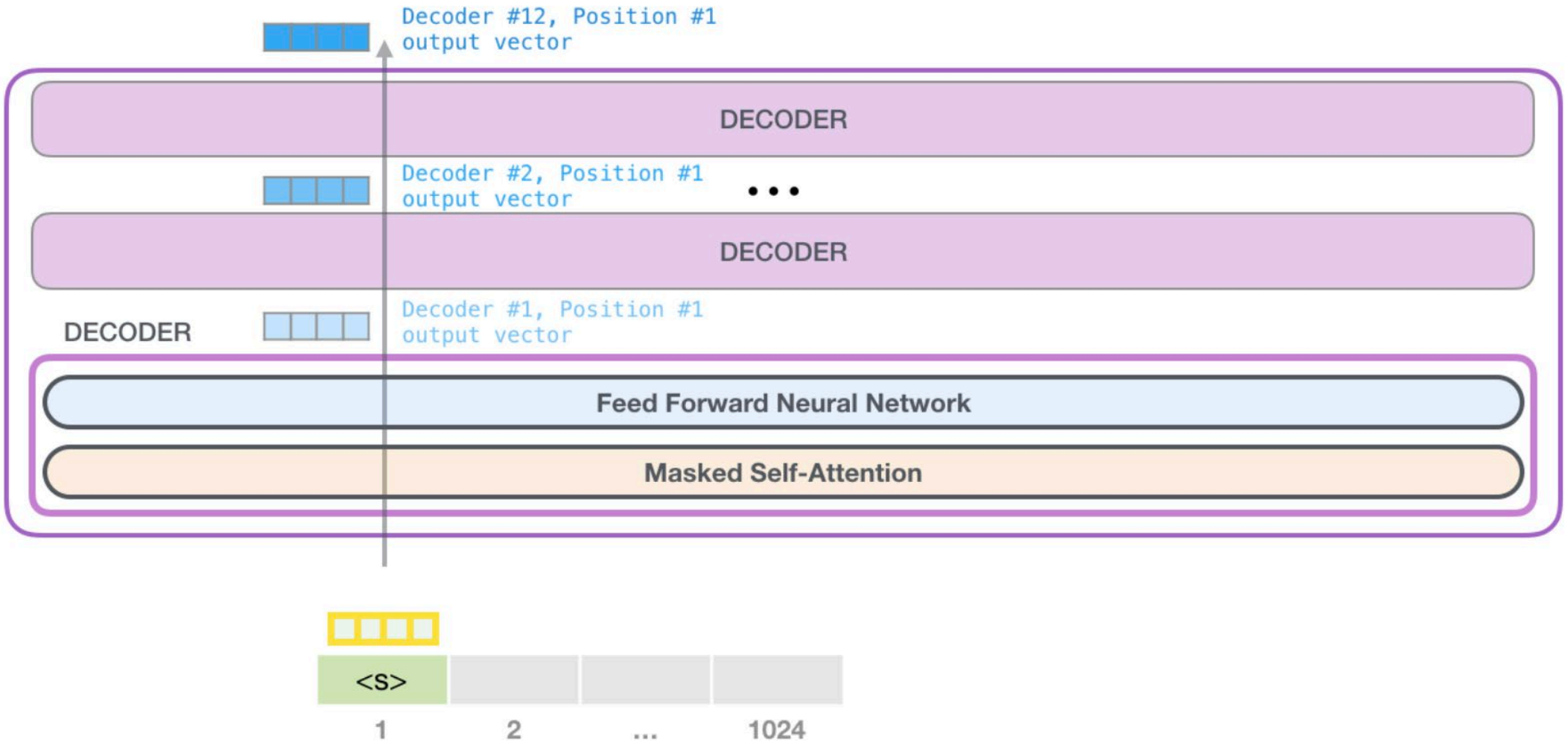
Decoder Only Model

Fig Credits: Jay Alammar



Processing through the decoder stack

Fig Credits: Jay Alammar



Self Attention Example

Fig Credits: Jay Alammar

Language heavily relies on context. For example, look at the second law:

> ***Second Law of Robotics***
> *A robot must obey the orders given **it** by human beings except where **such orders** would conflict with the **First Law**.*

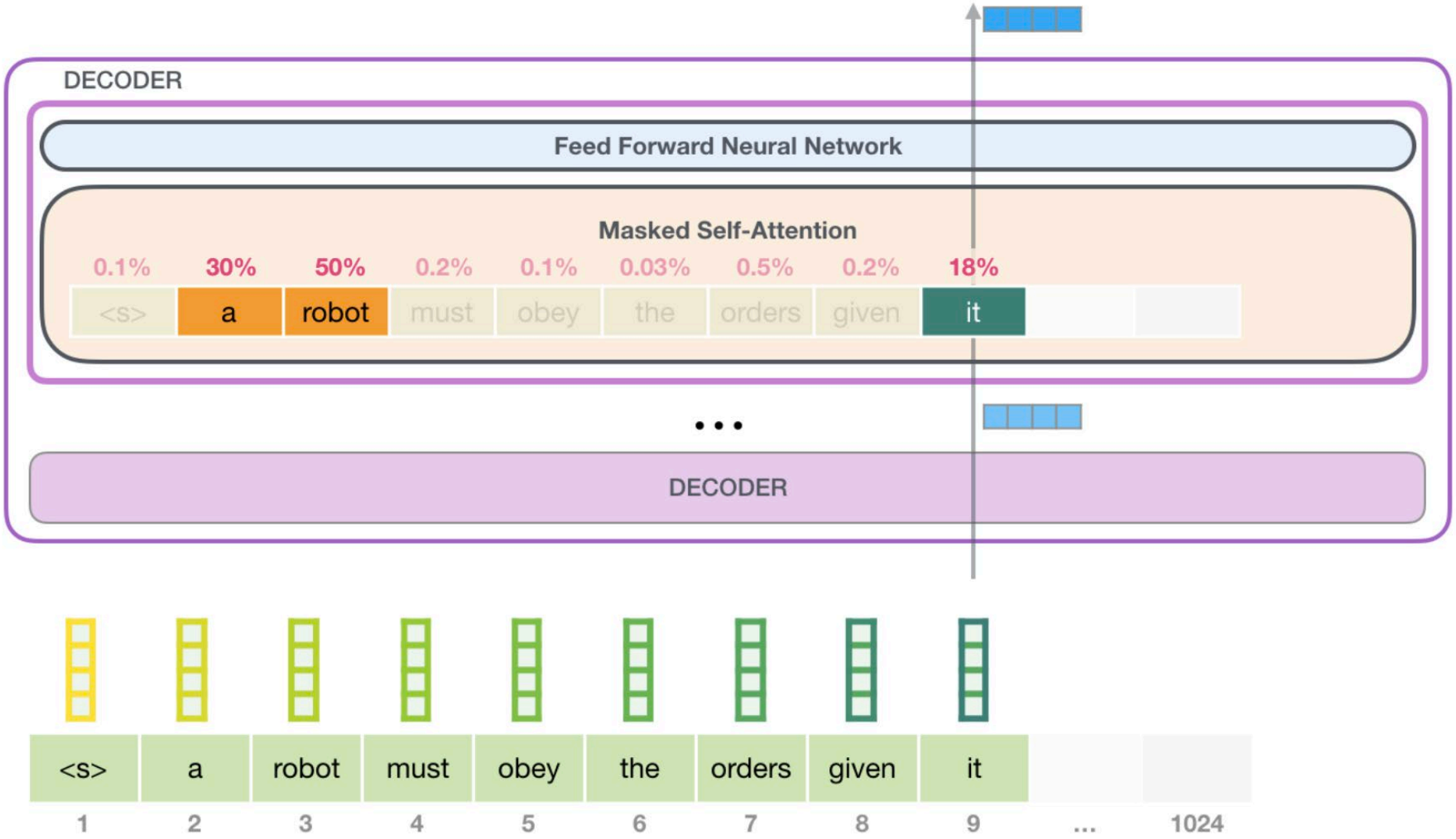
I have highlighted three places in the sentence where the words are referring to other words. There is no way to understand or process these words without incorporating the context they are referring to. When a model processes this sentence, it has to be able to know that:

- **it** refers to the robot
- **such orders** refers to the earlier part of the law, namely “the orders given it by human beings”
- **The First Law** refers to the entire First Law

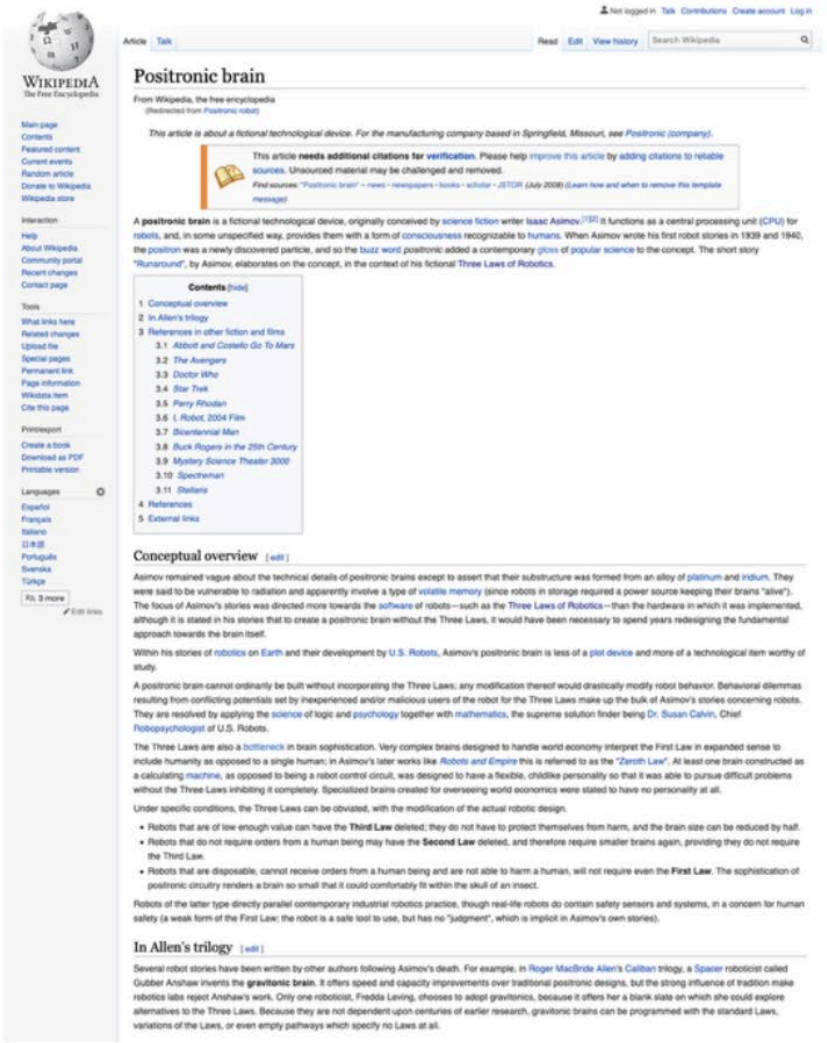
This is what self-attention does. It bakes in the model’s understanding of relevant and associated words that explain the context of a certain word before processing that word (passing it through a neural network). It does that by assigning scores to how relevant each word in the segment is, and adding up their vector representation.

As an example, this self-attention layer in the top block is paying attention to “a robot” when it processes the word “it”. The vector it will pass to its neural network is a sum of the vectors for each of the three words multiplied by their scores.

Self Attention in Decoder



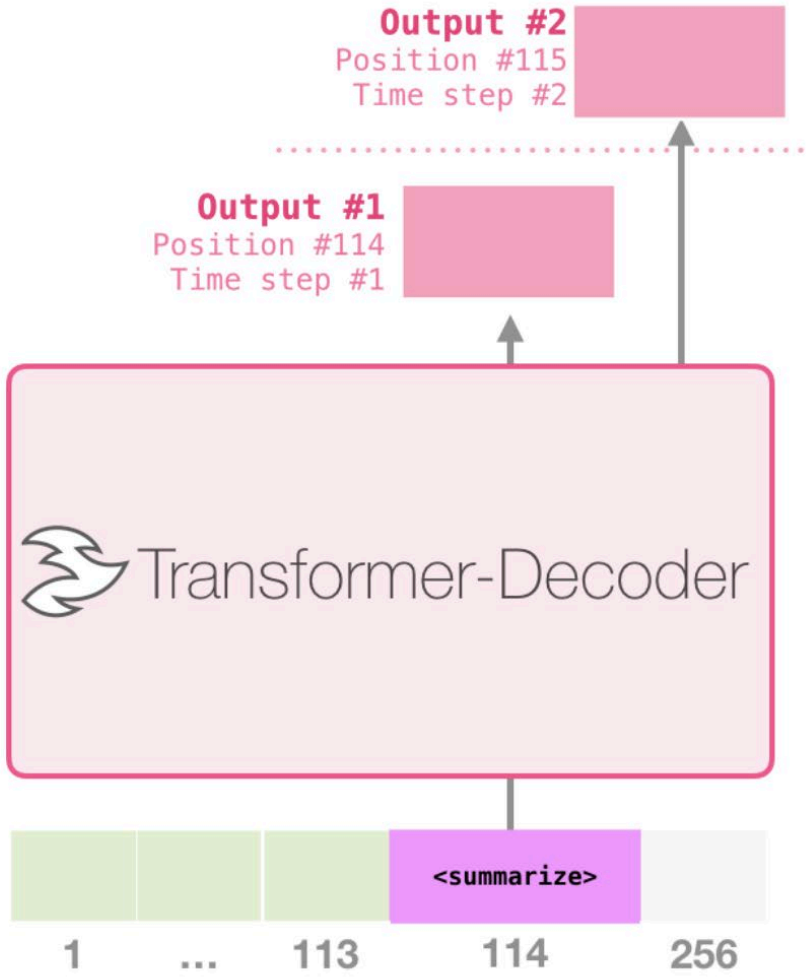
Summarization



Training for summarization

Training Dataset

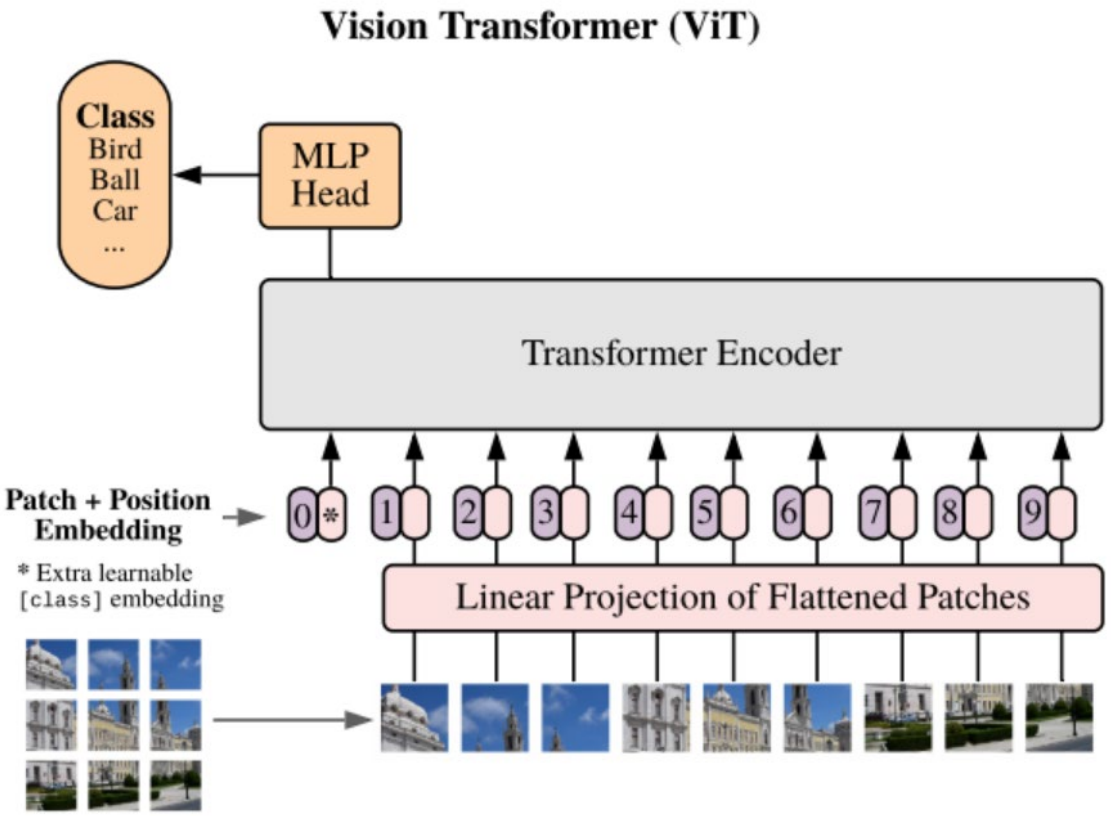
Article #1 tokens		<summarize>	Article #1 Summary
Article #2 tokens	<summarize>	Article #2 Summary	padding
Article #3 tokens		<summarize>	Article #3 Summary



Vision Transformers

Fig Credits: Binxu Wang, Harvard

- Transformers can be used in multimodal applications
- When the input is an image, patches of an image can be used as tokens
- For classification extra [CLS] token to be added.
- Vision transformers need a higher level of data and training effort.
- CNNs may provide better performance on models trained with less data, vision transformers are more scalable and a preferred technique for contemporary applications, particularly due to availability of pre-trained models



Summary

- Transformers brought in the era of transfer learning for NLP – the ImageNet moment
- Transformer based models like GPT are the basis for modern LLMs
- Transformers are versatile, they are getting to be the “go to” model for other data types like images.
- Transformers can be trained on multiple tasks at the same time and thus are foundation for LLMs. This also enables novel programming metaphors like prompt engineering

Thank You