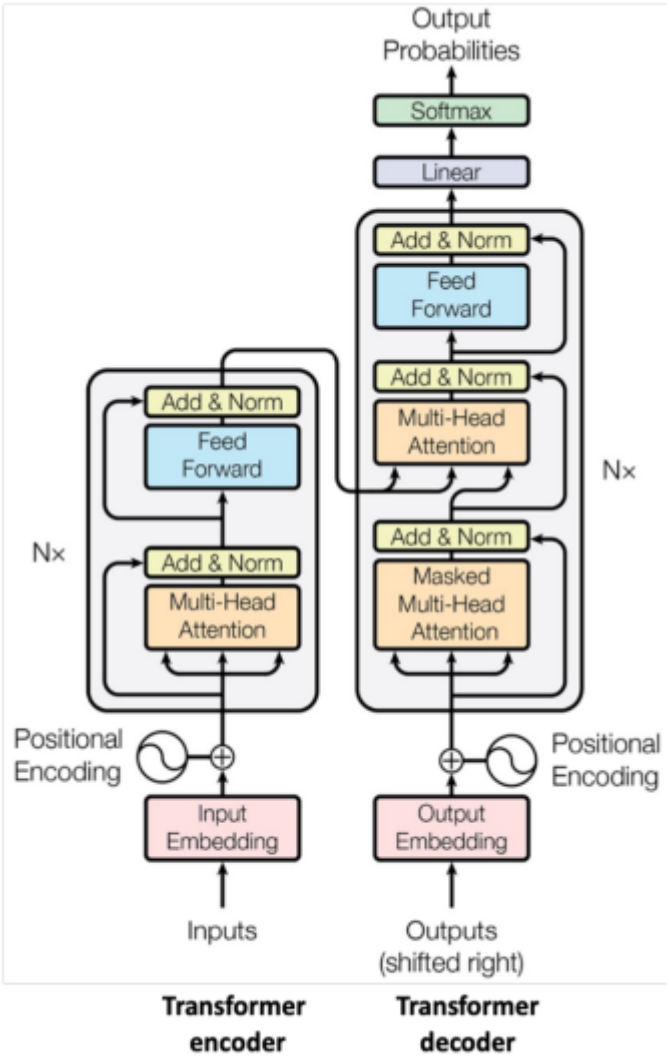
Note: revisit Neural Networks .Main key points are:

- a perceptron is a vector of numbers
- A “layer” is a matrix of numbers
- The neural network is a set of matrices, called “parameters” or “weights”

PART 1: Transformers

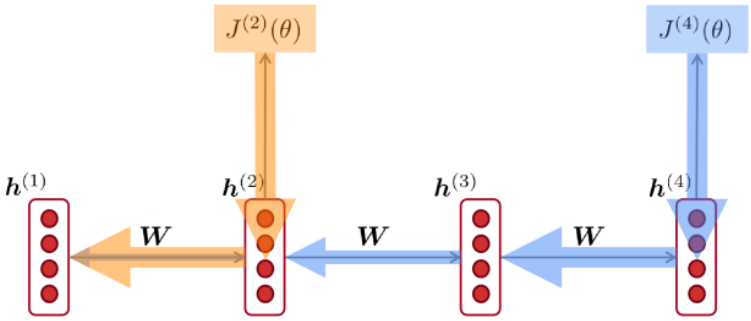
- **Basic diagram of Transformer**



For now it's enough t know that they capture how words relate to each other over long-distances and that they don't use any recurrent structures(unlike recurrent NNs)

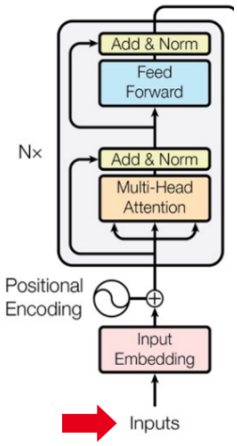
The problem with recurrent NNs is the difficulty the capture long-distance information
Vanishing gradients since the gradient signal from far away is weaker than signal close-by(Model

weights are only updated only with respect to near effects, not long-term effects.)



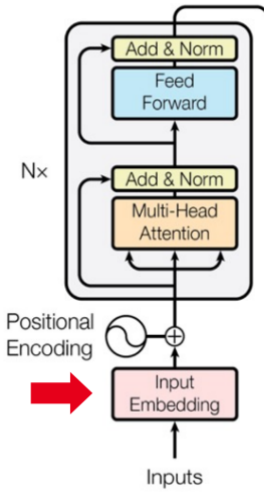
- RNNs also lack parallelizability (Forward and backward passes have O(sequence length) unparallelizable dependent operations) which hinders training on large datasets.
  Key Concepts of Transformers:
- Input embeddings
- Attention / self-attention / multi-head self-attention
- Positional encoding
- Feed forward layer
- Residual connections + layer normalization
- Transformer encoder vs. transformer decoder
  We will break these down in the next parts

1) Transformer inputs



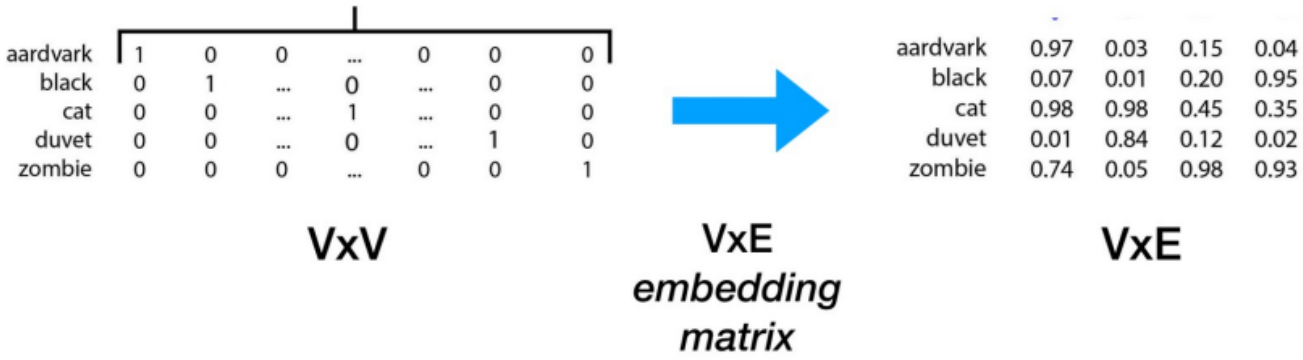
Inputs need to be vectors
Let’s start with original text: "A boy is eating an apple."
Step1: Turn into a sequence of tokens: - [,A, boy, is, eating, an, apple, .,]
Step2: Turn into vocabulary IDs: - [0, 1026, 338, 257, 4171, 37437, 601, 13, 1]
Step3: Obtain embeddings for each token Option1: Each ID can be represented by a one-hot vector (Lecture 5) - e.g. 3 -> [0, 0, 0, 1, 0, 0, 0, 0, ...]

2)Input Embeddings



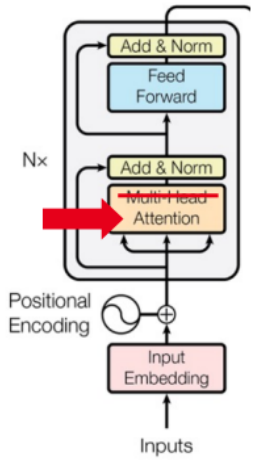
One-hot vectors are poor representations of words or tokens

- Very high dimension → huge weight matrix
- Semantic relations between words are not encoded
  E.g., distance between “dog” and “cat” is the same as between “orange” and “apple”
  -> We need an **embedding matrix**

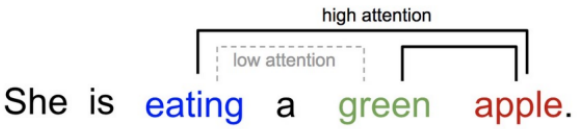


The conversion from the initial input representation to the embedding matrix is basically what happens in the red rectangle from the transformer architecture diagram.

3) Attention



Intuition: for a given token in the output sequence, only one or a few tokens in the input sequence are most important



- Assume that we have a set of values ($\mathbf{v}_1, \ldots, \mathbf{v}_n \in \mathbb{R}^{d_v}$) and a query vector $\mathbf{q} \in \mathbb{R}^{d_q}$

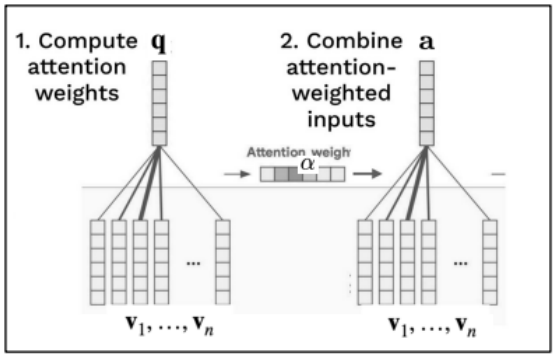
  Steps of calculating attention

  - Computing the **attention scores** $\mathbf{e} = g(\mathbf{q}, \mathbf{v}_i) \in \mathbb{R}^n$
  - Taking softmax to get **attention distribution** α:

$$\alpha = \text{softmax}(\mathbf{e}) \in \mathbb{R}^n$$

  - Using attention distribution to take **weighted sum** of values:

$$\mathbf{a} = \sum_{i=1}^{n} \alpha_i \mathbf{v}_i \in \mathbb{R}^{d_v}$$



3.a)Self-Attention

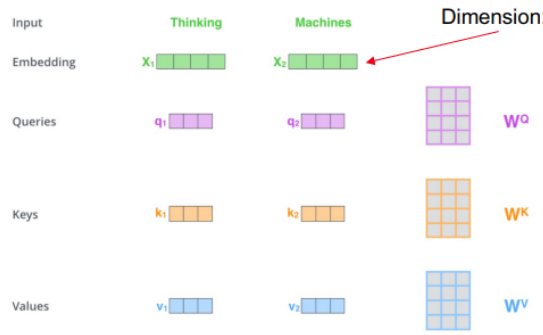
Attention from the sequence to itself (attention of each word to all other words)
Self-attention: each word in a sequence as the query, and all words in the sequence as keys. and values.

Dimension: 64

Step #1: Transform each input vector into three vectors: query, key, and value vectors

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q \in \mathbb{R}^{d_q} \qquad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K \in \mathbb{R}^{d_k} \qquad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V \in \mathbb{R}^{d_v}$$

$$\mathbf{W}^Q \in \mathbb{R}^{d_1 \times d_q} \qquad \mathbf{W}^K \in \mathbb{R}^{d_1 \times d_k} \qquad \mathbf{W}^V \in \mathbb{R}^{d_1 \times d_v}$$



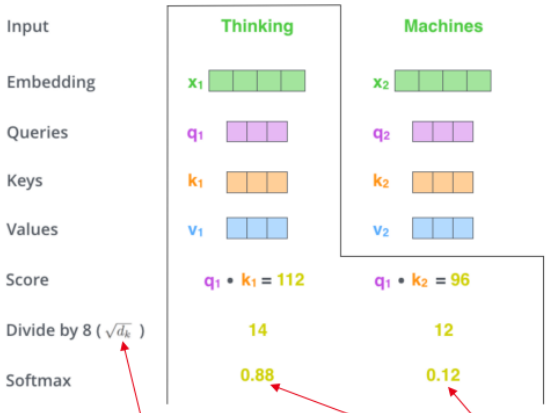
Note that we use row vectors here;
It is also common to write
$\mathbf{q}_i = \mathbf{W}^Q \mathbf{x}_i \in \mathbb{R}^{d_q}$
for $\mathbf{x}_i$ = a column vector

Step #2: Compute pairwise similarities between keys and queries; normalize with softmax

For each $\mathbf{q}_i$, compute attention scores and attention distribution:

$$\alpha_{i,j} = \text{softmax}(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}})$$

aka. "scaled dot product"
It must be $d_q = d_k$ in this case



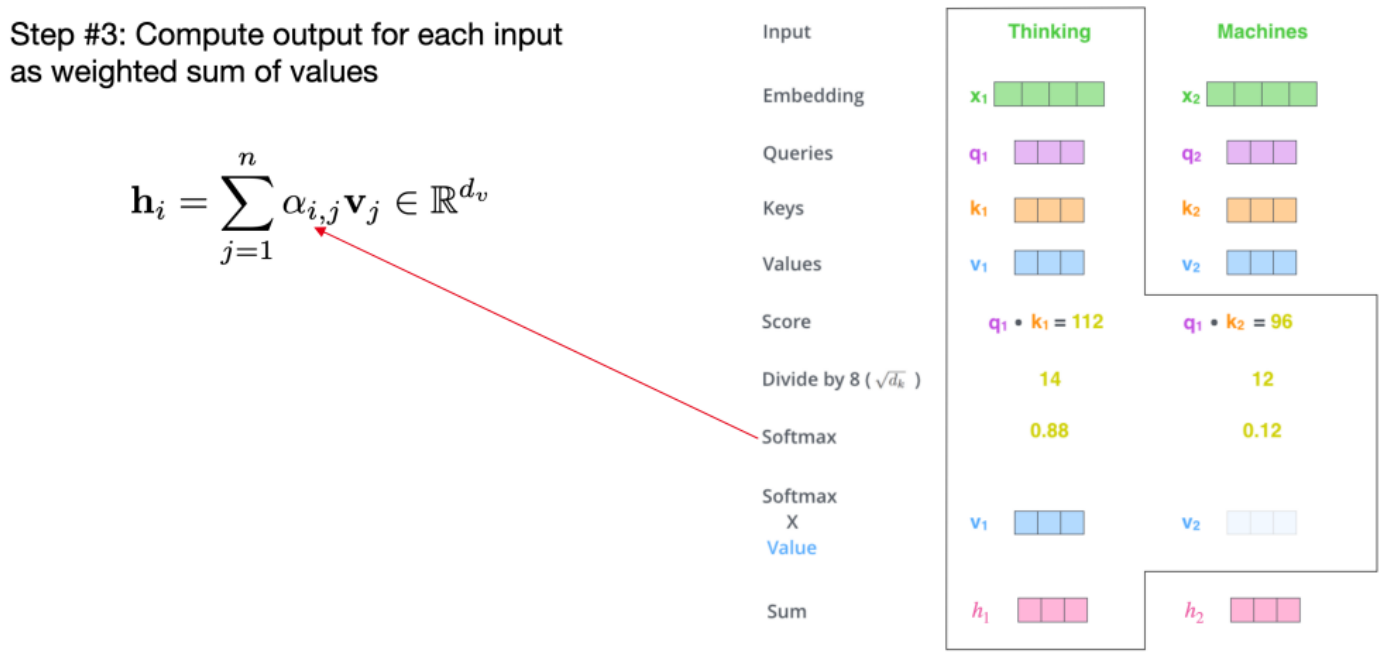
Q. Why scaled dot product?

To avoid the dot product to become too large for larger d_k; scaling the dot product by $\frac{1}{\sqrt{d_k}}$ is easier for optimization

Remember D_k=64

Determine how much each word will be expressed at this position

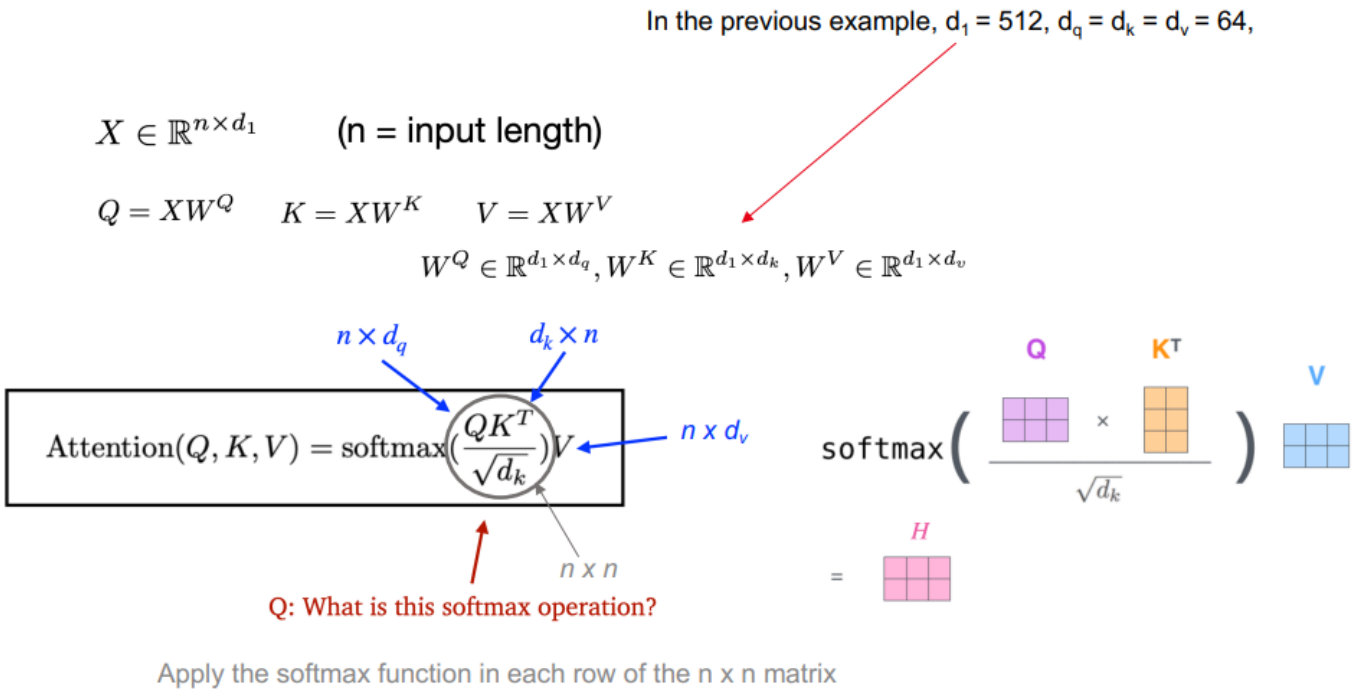
(https://jalammar.github.io/illustrated-transformer/)



(https://jalammar.github.io/illustrated-transformer/)

The result here is the output vector for the word "Thinking" and it is obviously: $0.88v_1 + 0.12v_2$

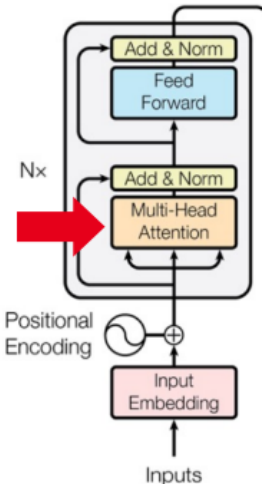
Calculating Attention in Matrix form



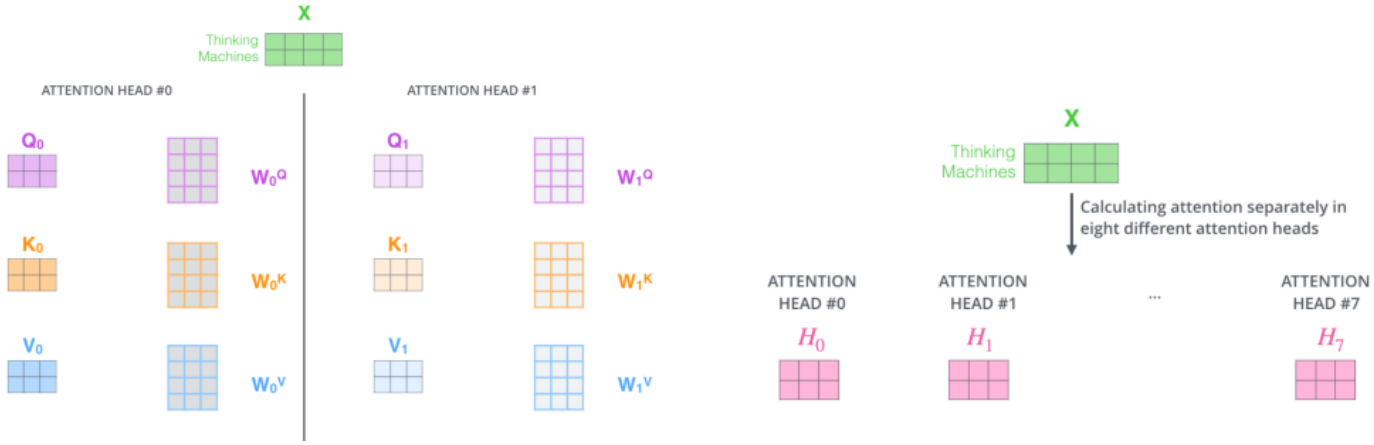
Apply the softmax function in each row of the n x n matrix

3.b)Multi-Head Self-Attention

The previous part was a basic explanation of Self-Attention concept. But what our transformer architecture actually uses is more advanced and it is called **Multi-Head Self-Attention**

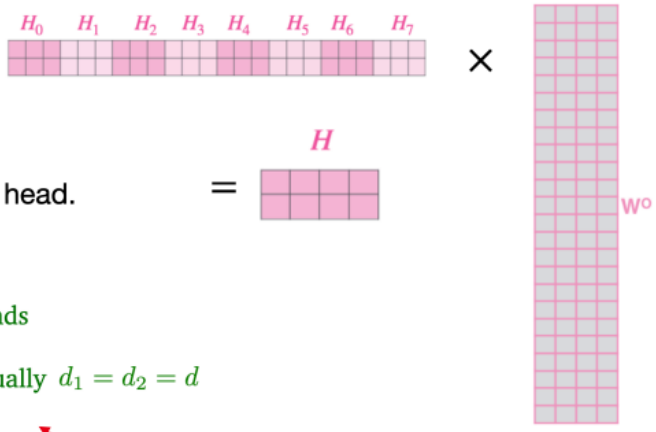


- It is better to use multiple attention functions instead of one!
  - Each attention function ("head") can focus on different positions.



Finally, we just concatenate all the heads and apply an output projection matrix.

$$\text{MultiHead}(Q,K,V) = \text{Concat}(\text{head}_1,...,\text{head}_h)W^O$$
$$\text{head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V)$$



- In practice, we use a *reduced* dimension for each head.

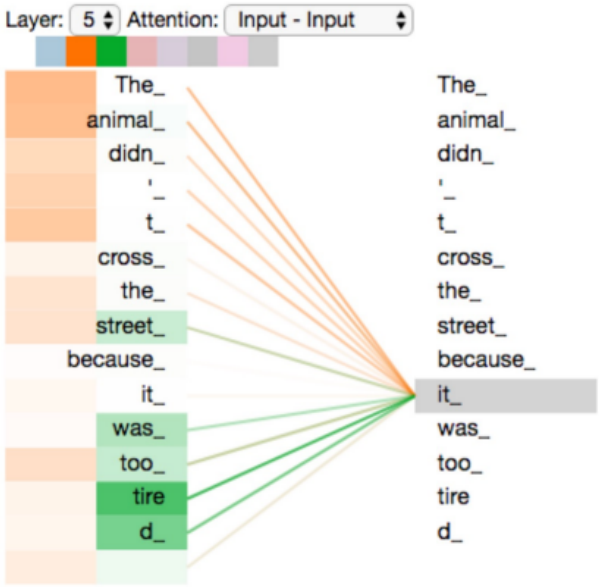
$$W_i^Q \in \mathbb{R}^{d_1 \times d_q}, W_i^K \in \mathbb{R}^{d_1 \times d_k}, W_i^V \in \mathbb{R}^{d_1 \times d_v}$$

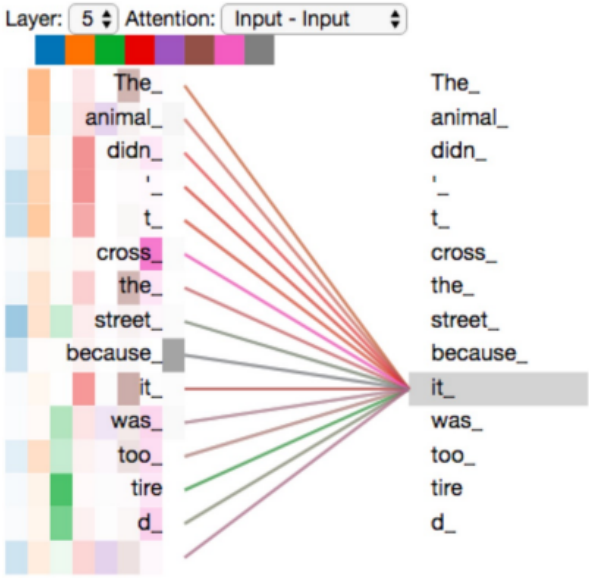
$$d_q = d_k = d_v = d/m \quad d = \text{hidden size}, m = \#\text{ of heads}$$

$$W^O \in \mathbb{R}^{d \times d_2} \quad \text{If we stack multiple layers, usually } d_1 = d_2 = d$$

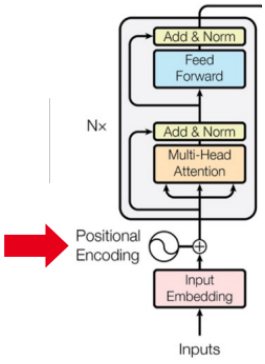
- The total computational cost is similar to that of single-head attention with full dimensionality.

In the previous example, $d_1 = 512$, $d_q = d_k = d_v = 64$, $m = 8$





4)Positional encoding

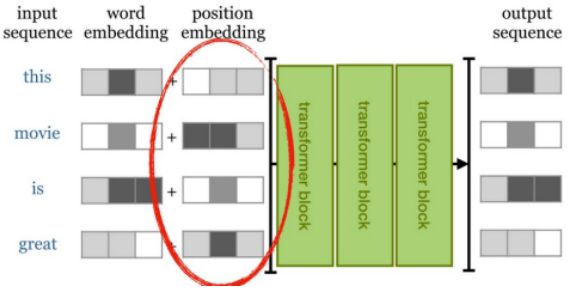


Self-Attention is **position-invariant** :

E.g., [this, movie, is, great] is the same as [is, this, movie, great]

-> Solution: we add position-encoding vectors to embedding vectors

$$\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{p}_i$$

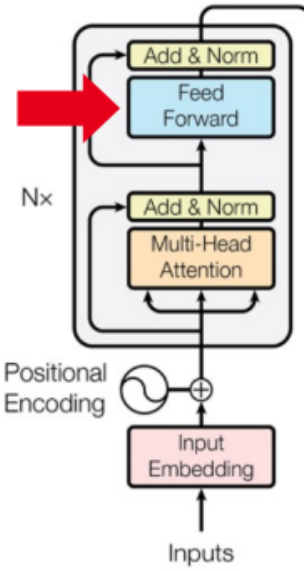


Learned absolute position encoding: let all $\mathbf{p}_i$ be learnable parameters

- $P \in \mathbb{R}^{d \times L}$ for L = max sequence length

- **Pros**: each position gets to be learned to fit the data
- **Cons**: can't extrapolate to indices outside of max sequence length L
- Most systems use this!

5)Feed-Forward layer(Lecture about neural networks)



- **Problem:** There are no elementwise nonlinearities in self-attention
- **Solution:** add a feed-forward network to post-process each output vector

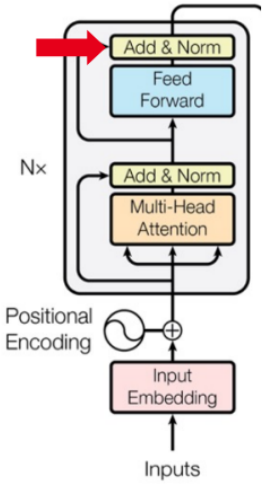
$$\text{FFN}(\mathbf{x}_i) = \text{ReLU}(\mathbf{x}_i\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

$$\mathbf{W}_1 \in \mathbb{R}^{d \times d_{ff}}, \mathbf{b}_1 \in \mathbb{R}^{d_{ff}}$$

$$\mathbf{W}_2 \in \mathbb{R}^{d_{ff} \times d}, \mathbf{b}_2 \in \mathbb{R}^{d}$$

In practice, they use $d_{ff} = 4d$

6)Residual Connection and Layer Normalization



Add & Norm: $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$

- Residual connection
  - output = module(input) + input
  - Allows gradient to flow from the loss function all the way to the first layer

- Layer Normalization
  - Neural net modules perform best when input vectors have uniform mean and std in each dimension.
  - As inputs flow through the network, means and std's get blown out.
  - Layer Normalization is a hack to reset things to where we want them in between layers.
  - Idea: normalize the hidden vector values to unit mean and stand deviation within each layer

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

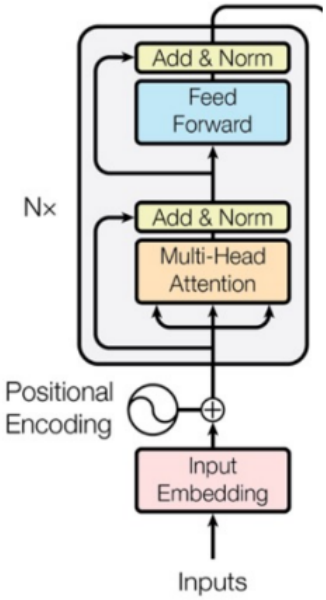
$\gamma, \beta \in \mathbb{R}^d$ are learnable parameters

Layer normalization (Ba et al., 2016)

Tranformer Encoder(1+2+3+4+5+6)

From the bottom to the top:

- Input embedding
- Positional encoding
- A stack of Transformer encoder layers

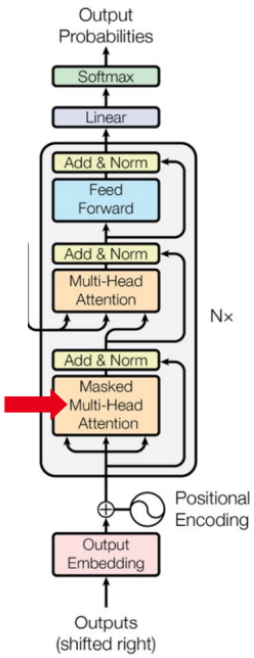


Transformer encoder is a stack of N layers, which consists of two sub-layers:

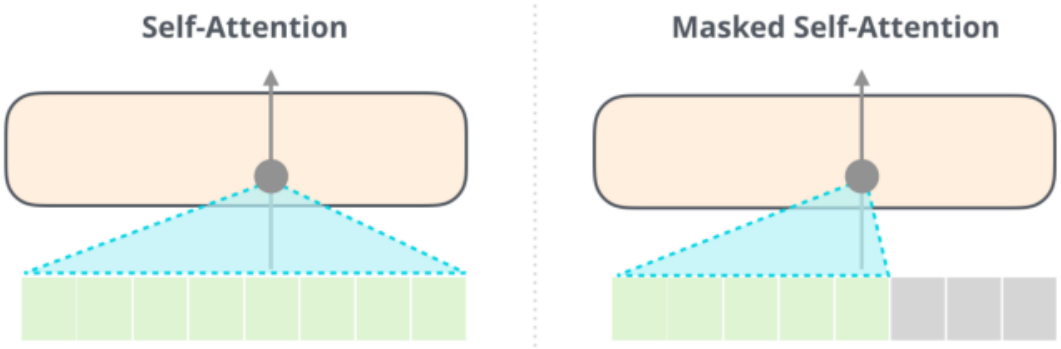
- Multi-head attention layer
- Feed-forward layer

$$\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^{d_1} \xrightarrow{\text{Transformer Encoder}} \mathbf{h}_1, \ldots, \mathbf{h}_n \in \mathbb{R}^{d_2}$$

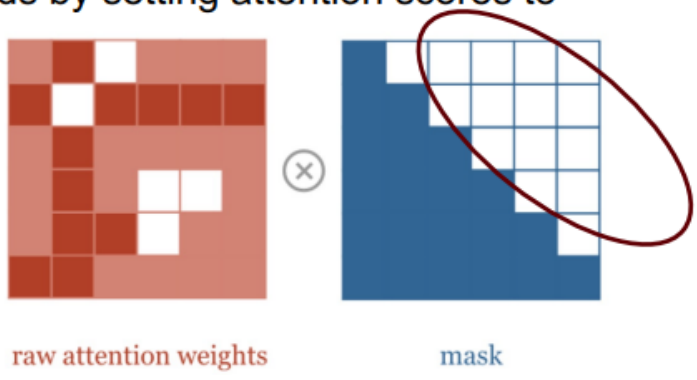
7)Masked Self-Attention



- We can't attend to the future text for the decoder!
- Solution: for every q_i, only attend to the previous context $\{(\mathbf{k}_j, \mathbf{v}_j)\}, j \leq i$



- Masked multi-head attention: compute attention as we normally do, mask out attention to future words by setting attention scores to $-\infty$



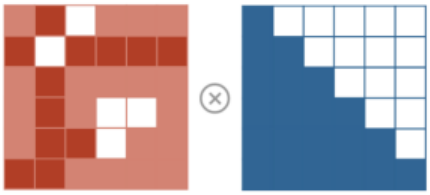
- Masked multi-head attention: compute attention as we normally do, mask out attention to future words by setting attention scores to $-\infty$

- **Quiz**

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q, \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K, \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V$$

$$e_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \forall j = 1, \ldots, n$$

$$\alpha_i = \text{softmax}(\mathbf{e}_i)$$



raw attention weights mask

The following matrix denotes the values of $\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$ for $1 \leq i \leq n, 1 \leq j \leq n$ $(n = 4)$

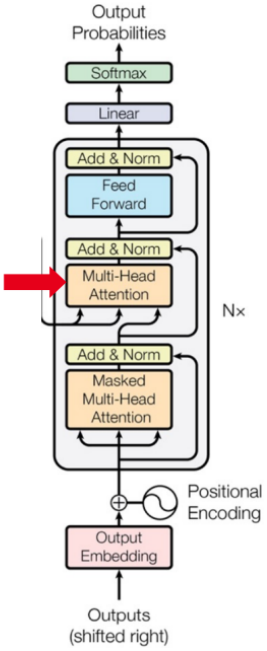
1	0	-1	-1
1	1	-1	0
0	1	1	-1
-1	-1	2	1

What should be the value of $\alpha_{2,2}$ in masked attention?

(A) 0
(B) 0.5
(C) $\frac{e}{2e + e^{-1} + 1}$
(D) 1

Note: quiz inserted as a small exercise

8) Multi-head Cross-attention



Self-attention:

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^Q, \mathbf{k}_i = \mathbf{x}_i\mathbf{W}^K, \mathbf{v}_i = \mathbf{x}_i\mathbf{W}^V$$

$$e_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \forall j = 1, \ldots, n$$

$$\alpha_i = \text{softmax}(\mathbf{e}_i)$$

$$\mathbf{h}_i = \sum_{j=1}^{n} \alpha_{i,j}\mathbf{v}_j$$

Cross-attention: (always from the top layer)

$\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_m$: hidden states from encoder

$\mathbf{x}_1, \ldots, \mathbf{x}_n$: hidden states from decoder

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^Q \quad i = 1,2,\ldots,n$$

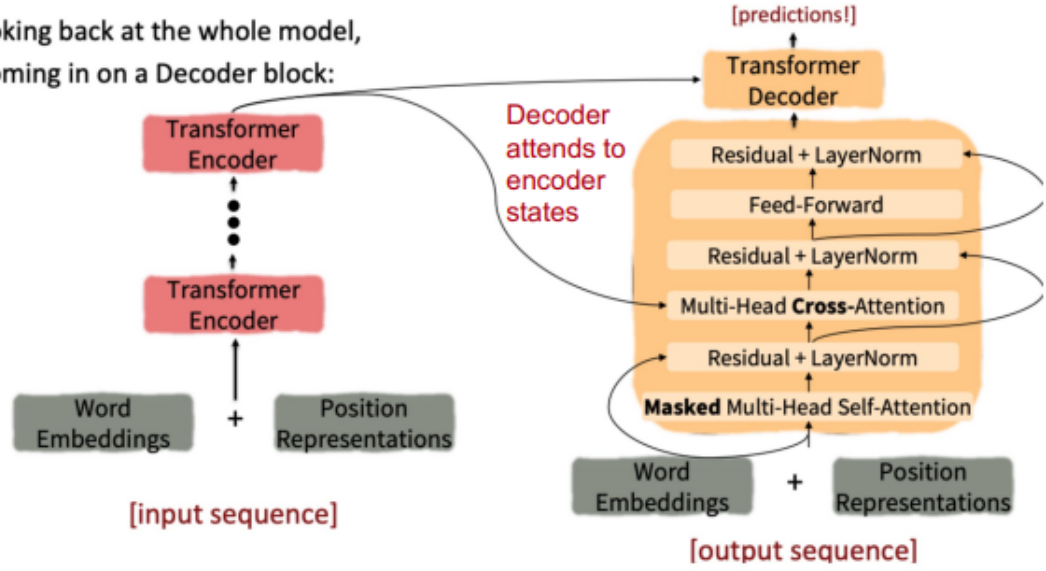
$$\mathbf{k}_j = \tilde{\mathbf{x}}_j\mathbf{W}^K, \mathbf{v}_j = \tilde{\mathbf{x}}_j\mathbf{W}^V \quad \forall j = 1,2,\ldots,m$$

$$e_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \forall j = 1, \ldots, m$$

$$\alpha_i = \text{softmax}(\mathbf{e}_i)$$

$$\mathbf{h}_i = \sum_{j=1}^{m} \alpha_{i,j}\mathbf{v}_j$$

Looking back at the whole model, zooming in on a Decoder block:



Transformer Decoder

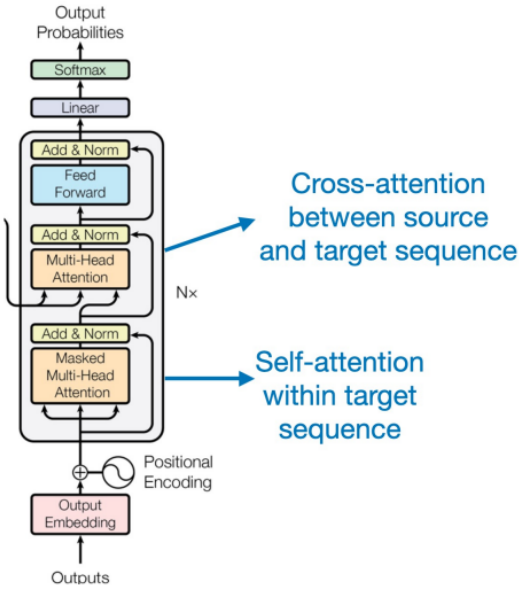
Transformer Decoder

From the bottom to the top:

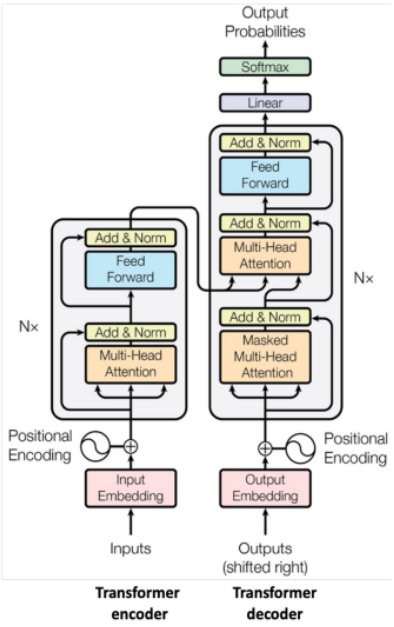
- Output embedding
- Positional encoding
- A stack of Transformer decoder layers
- Linear + softmax

Transformer decoder is a stack of N layers, which consists of three sub-layers:

- Masked multi-head attention
- Multi-head cross-attention
- Feed-forward layer
- (W/ Add & Norm between sub-layers)



We finally discussed all the Transformer architecture and here is how it looks like



Now let's end this section by discussing some pros and cons of transformers

Pros:

- Easier to capture long-range dependencies: we draw attention between every pair of words!
- Easier to parallelize:

$$Q = XW^Q \qquad K = XW^K \qquad V = XW^V$$

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

Cons

- Are positional encodings enough to capture positional information? self-attention is an unordered function of its input
- Quadratic computation in self-attention: can become very slow when the sequence length is large

Transformer parameters

The main transformer layer is stacked many times (That is what the Nx in the Transformer Diagram refers to)
The overall hyperparameters are:

- Number of layers
- Number of attention heads
- Embedding dimension

PART2: Contextualized Representations and Transformer-based LLMs

Limitations of Word2Vec :
Static word embeddings: one vector for each token (Embedding (bank) = [0.224, 0.168, -0.309, 0.160])
But Polysemous words

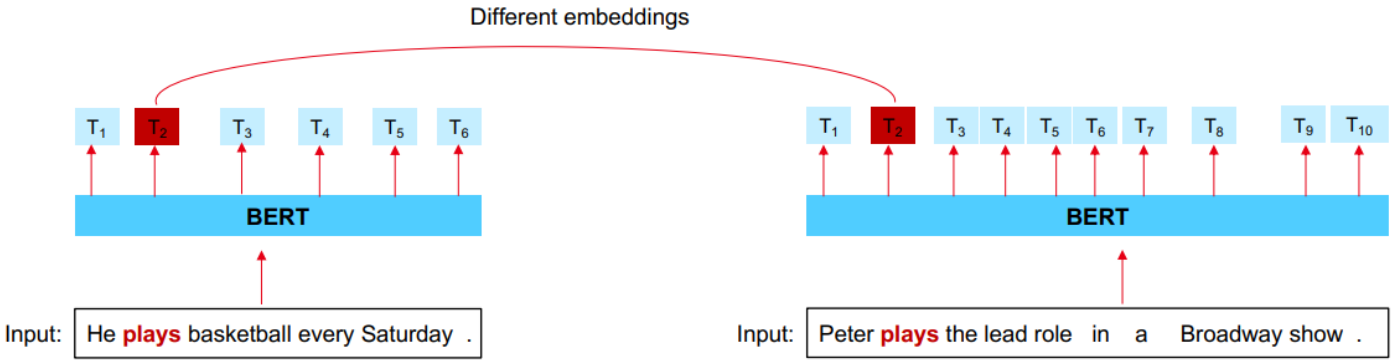
bank^1 : ...a *bank* can hold the investments in a custodial account ...
bank^2 : ...as agriculture burgeons on the east *bank*, the river ...

->Cannot capture the nuanced semantics of a word in different contexts

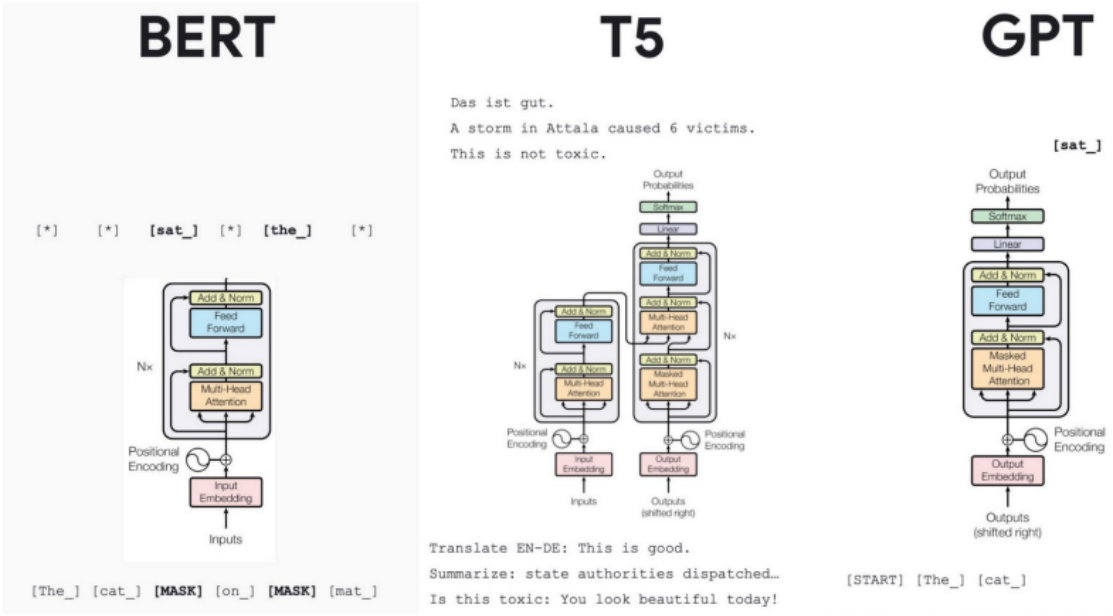
- He **plays** basketball every Saturday.
- Peter **plays** the lead role in a Broadway show.
- Weather **plays** a big role in shaping our daily activities.

1)Contextualized Representations

Transformer-based pretrained LMs (e.g., BERT, GPT) can generate contextualized representations for a word/token according to its surrounding context:



2)Transformer based LLMS



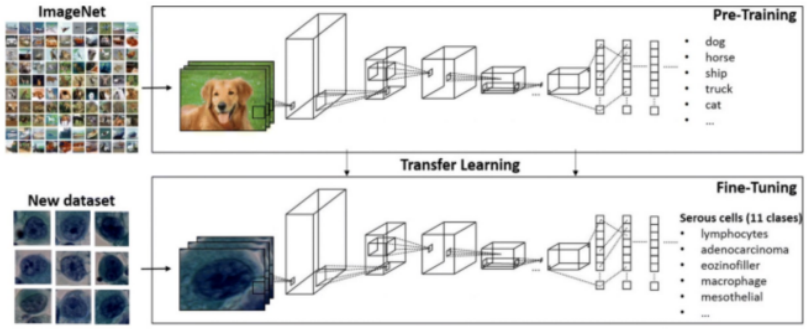
A good example to understand the utility of each architecture

1) Pretraining and Fine-Tuning (BERT)

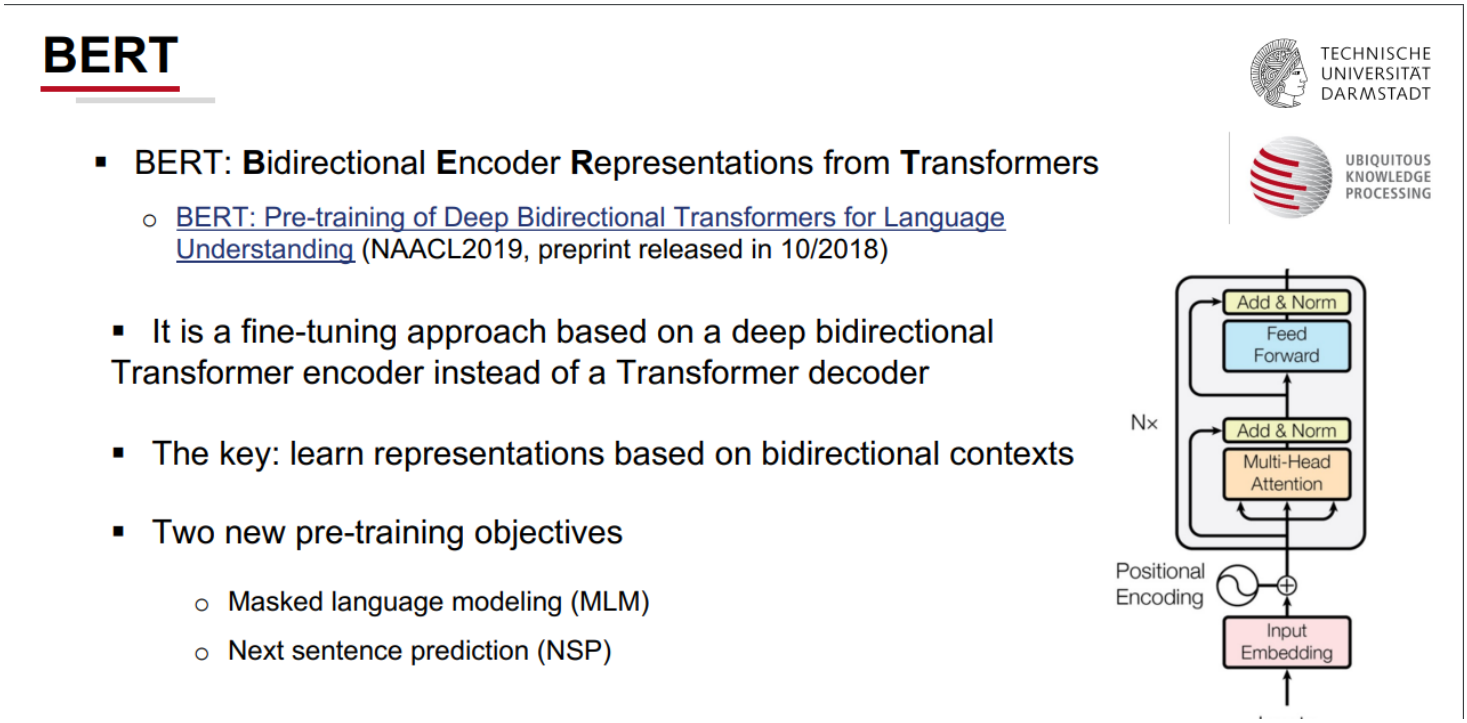
"Pre-train" a model on a large dataset for task X, then **"fine-tune"** it on a dataset for task Y
Key idea: X is somewhat related to Y, so a model that can do X will have some good neural representations for Y as well
Example: Image-Net pretraining

- Image-Net pretraining

BERT- NLP's Image-Net Moment



We will use BERT for this part as an example to explain Pretraining and Fine-Tuning



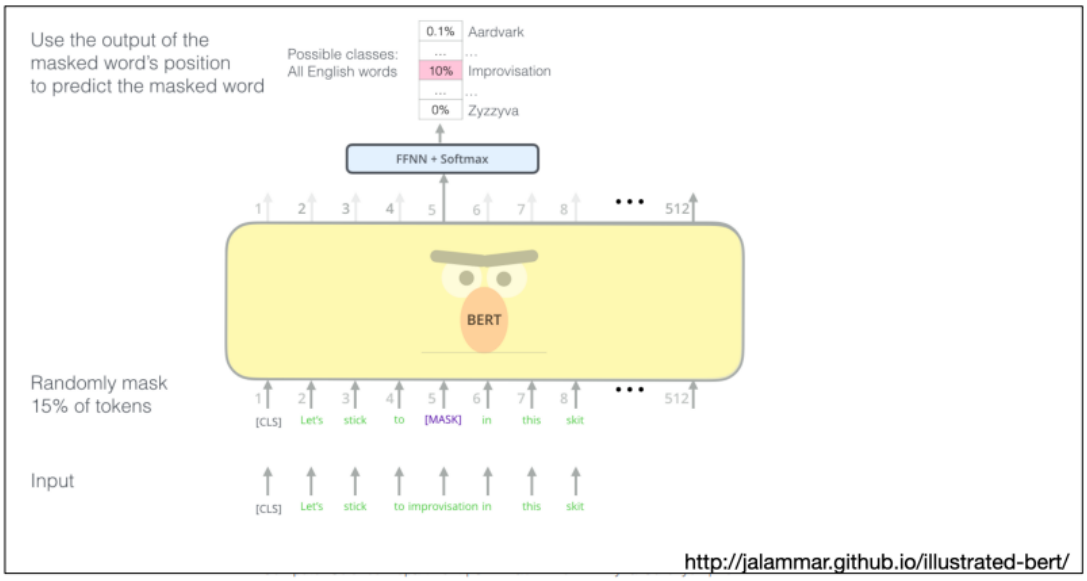
Masked Language Models (MLM)

- Idea: Masked out k% of the input words, and then predict the masked words

store gallon

the man went to [MASK] to buy a [MASK] of milk

k = 15% in practice



Next Sentence Prediction (NSP)

- Idea: Many NLP tasks require understanding the relationships between two sentences
- NSP is designed to reduce the gap between pre-training and fine-tuning

(Later work shows that NSP hurts performance though)

[CLS]: a special token always at the beginning

[SEP]: a special token used to separate two segments

Input = `[CLS] the man went to [MASK] store [SEP]`
`he bought a gallon [MASK] milk [SEP]`
Label = `IsNext`

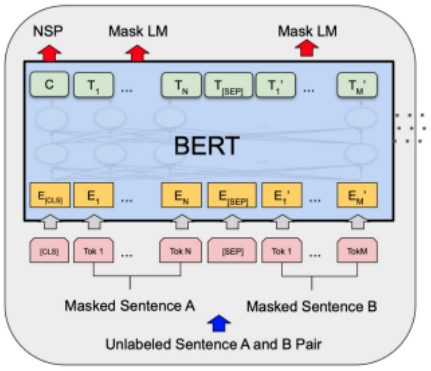
Input = `[CLS] the man [MASK] to the store [SEP]`
`penguin [MASK] are flight ##less birds [SEP]`
Label = `NotNext`

They sample two contiguous segments for 50% of the time and another random segment from the corpus for 50% of the time

BERT Pre-training and Fine-tuning

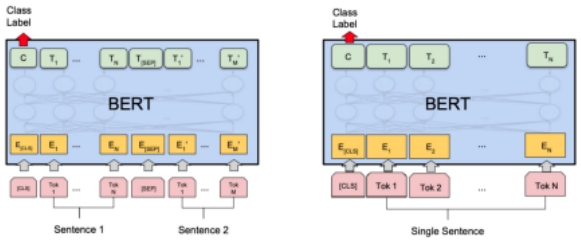
Pre-training

- MLM and NSP are trained together
- [CLS] is pre-trained for NSP
- Other token representations are trained for MLM



Fine-tuning

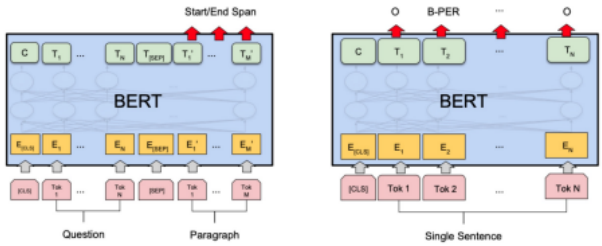
sentence-level tasks



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

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

token-level tasks

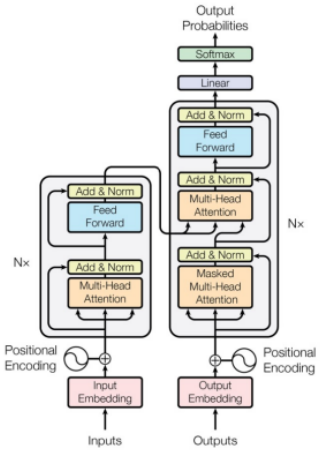


(c) Question Answering Tasks: SQuAD v1.1

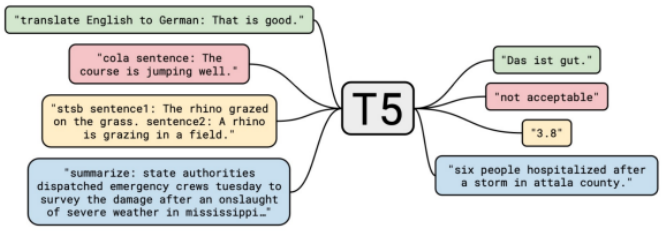
(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

2)T5

- Encoder-decoder architecture based on Transformers



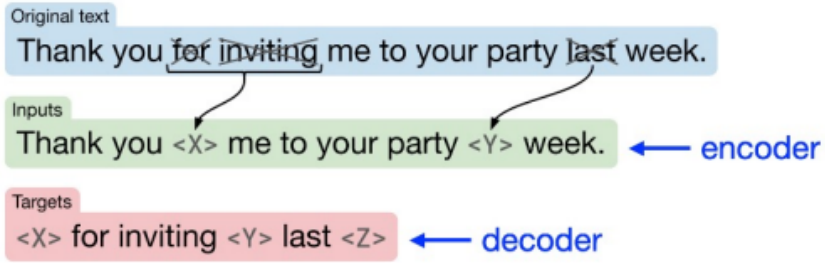
T5 = **T**ext-**t**o-**T**ext **T**ransfer **T**ransformer



T5 comes in different sizes:

- t5-small.
- t5-base.
- t5-large.
- t5-3b.
- t5-11b.

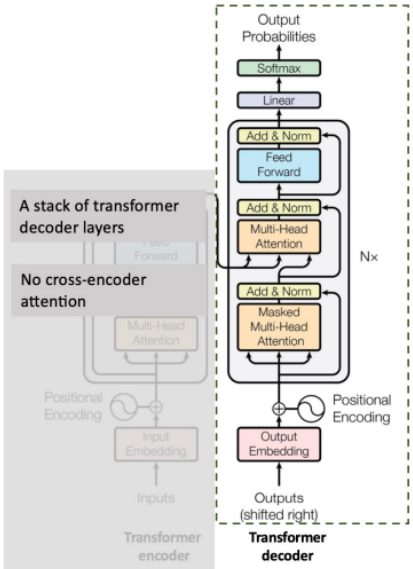
- Unsupervised pre-training: corrupting span prediction
  - Mean span length of 3 and corrupt 15% of the original sequences



3) GPT

GPT: Generative Pretrained Transform

- Based on the transformer decoder architecture
- Unsupervised pretraining: next token prediction given the previous context



Inference

- Autoregressive
- Sample the next token from the distribution, append it to the input, run through the model again