Transformer: Neural machine translation without recurrence

- It is possible to replace recurrence with **self-attention** within the encoder and decoder, as in the **transformer architecture**

$$z_m^{(i)} = \sum_{n=1}^{M^{(s)}} \alpha_{m \to n}^{(i)} \left(\Theta_v \boldsymbol{h}_n^{(i-1)} \right)$$

$$\boldsymbol{h}_m^{(i)} = \Theta_2 \mathrm{ReLU}\left(\Theta_1 \boldsymbol{z}_m^{(i)} + \boldsymbol{b}_1 \right) + \boldsymbol{b}_2$$

For each token m at level i, we compute self-attention over the entire source sentence. The keys, values, and queries are all projections of the vector $\boldsymbol{h}^{(i-1)}$.

- The attention scores $\alpha_{m \to n}^{(i)}$ are computed using a scaled form of softmax attention,

$$\alpha_{m \to n} \propto \exp(\psi_\alpha(m, n)/M)$$

"Self-attention"

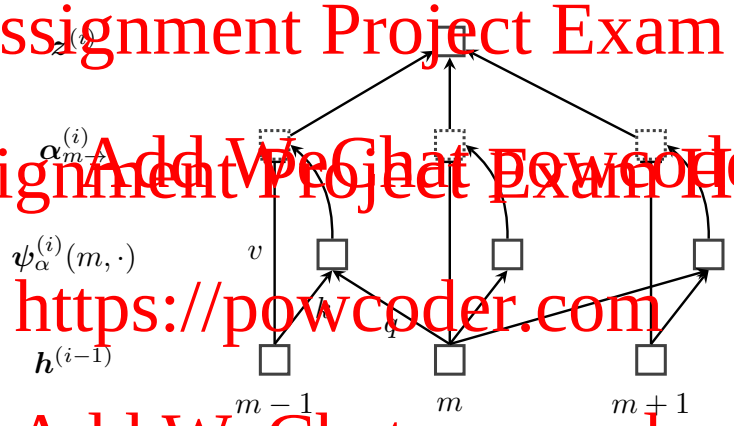
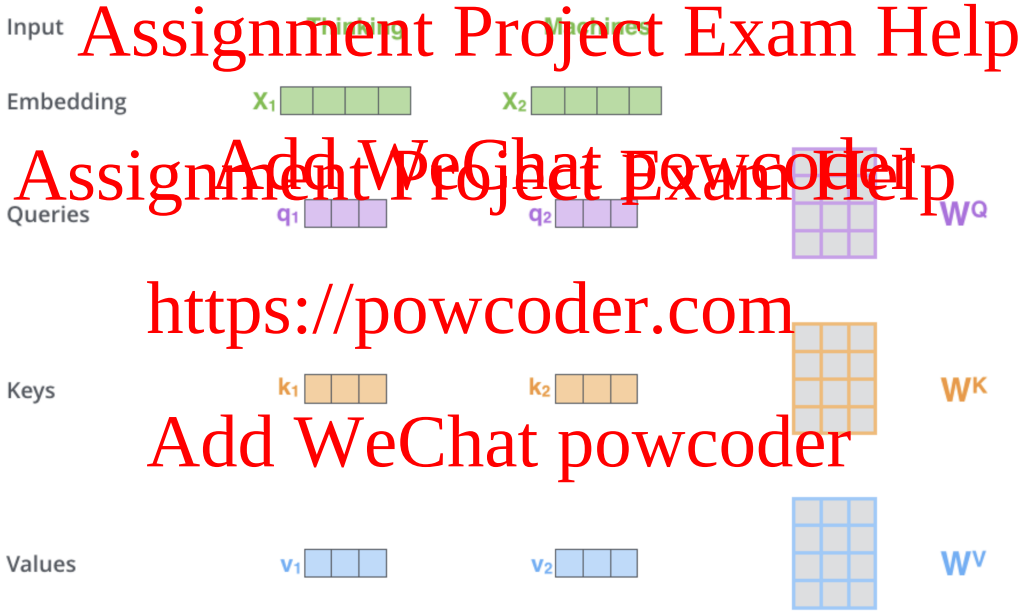


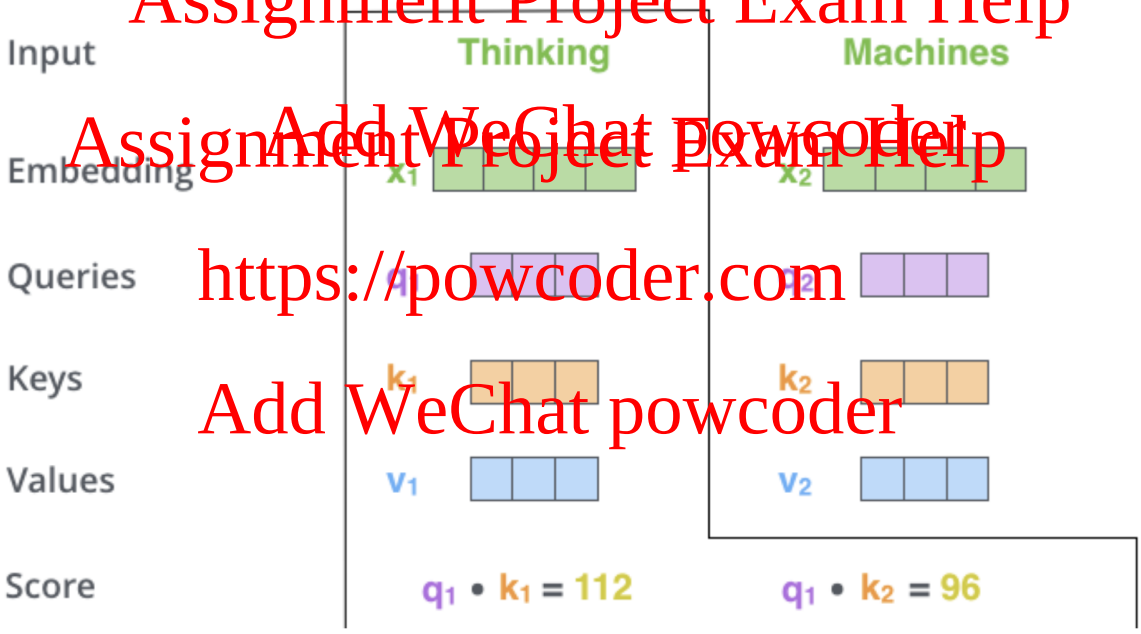
Figure 18.7: The transformer encoder's computation of $\boldsymbol{z}_m^{(i)}$ from $\boldsymbol{h}^{(i-1)}$. The **k**ey, **v**alue, and **q**uery are shown for token $m-1$. For example, $\psi_{\alpha}^{(i)}(m, m-1)$ is computed from the key $\Theta_k \boldsymbol{h}_{m-1}^{(i-1)}$ and the query $\Theta_q \boldsymbol{h}_m^{(i-1)}$, and the gate $\alpha_{m\rightarrow m-1}^{(i)}$ operates on the value $\Theta_v \boldsymbol{h}_{m-1}^{(i-1)}$. The figure shows a minimal version of the architecture, with a single attention head. With multiple heads, it is possible to attend to different properties of multiple words.

“Self-attention”

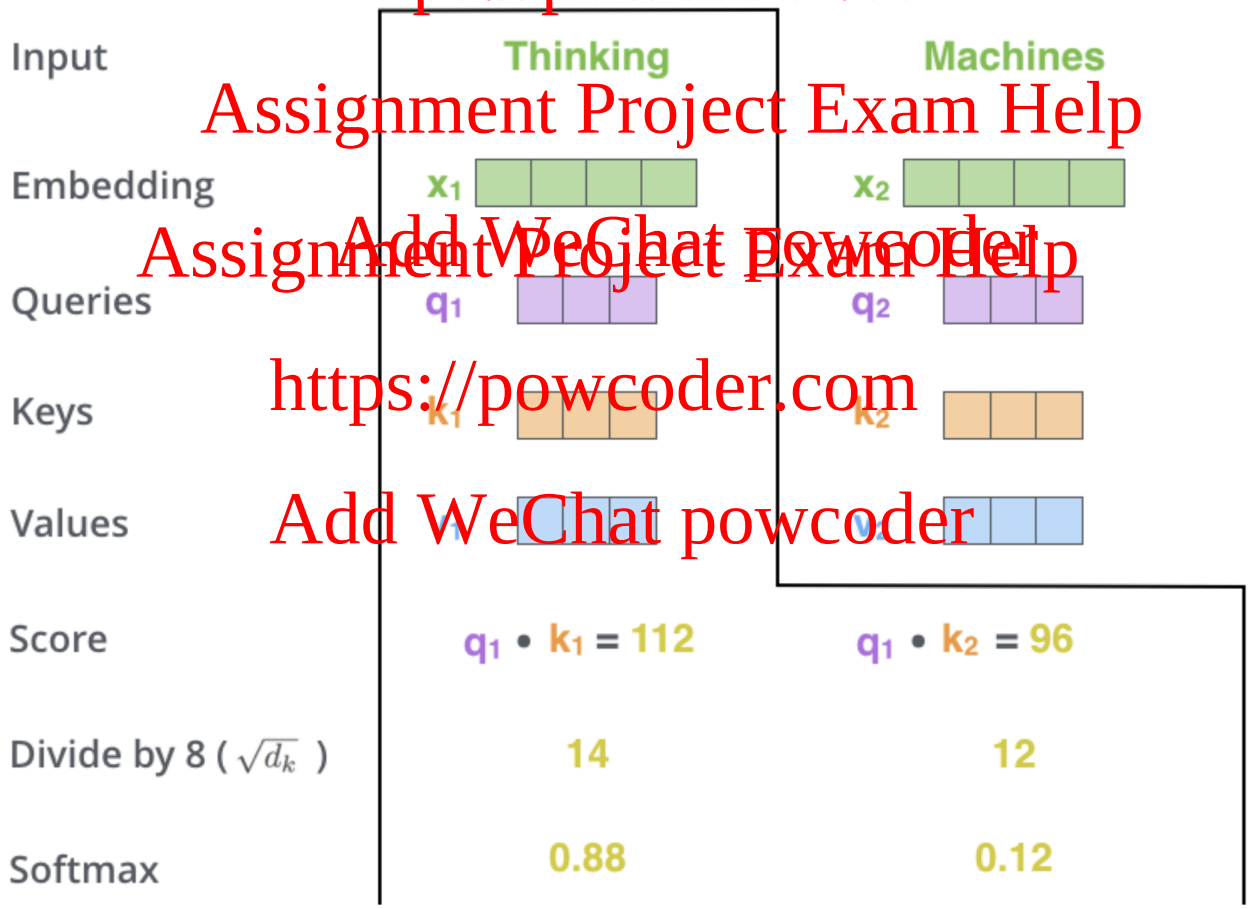


Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

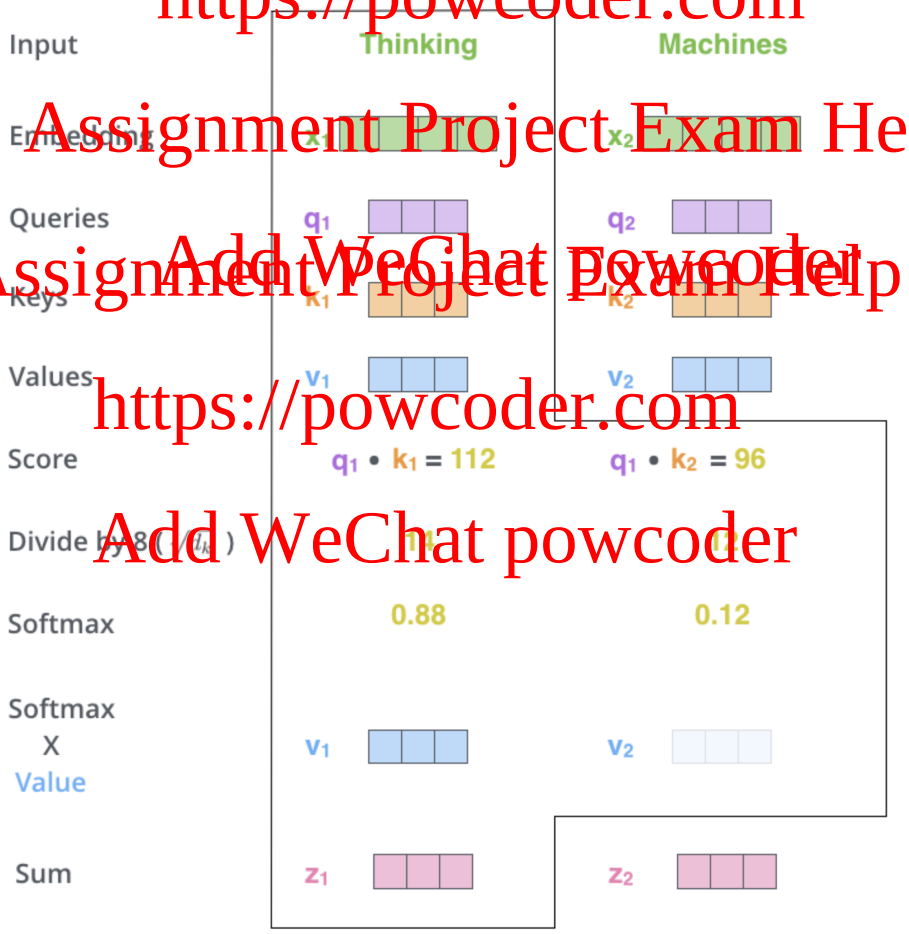
"Self-attention"



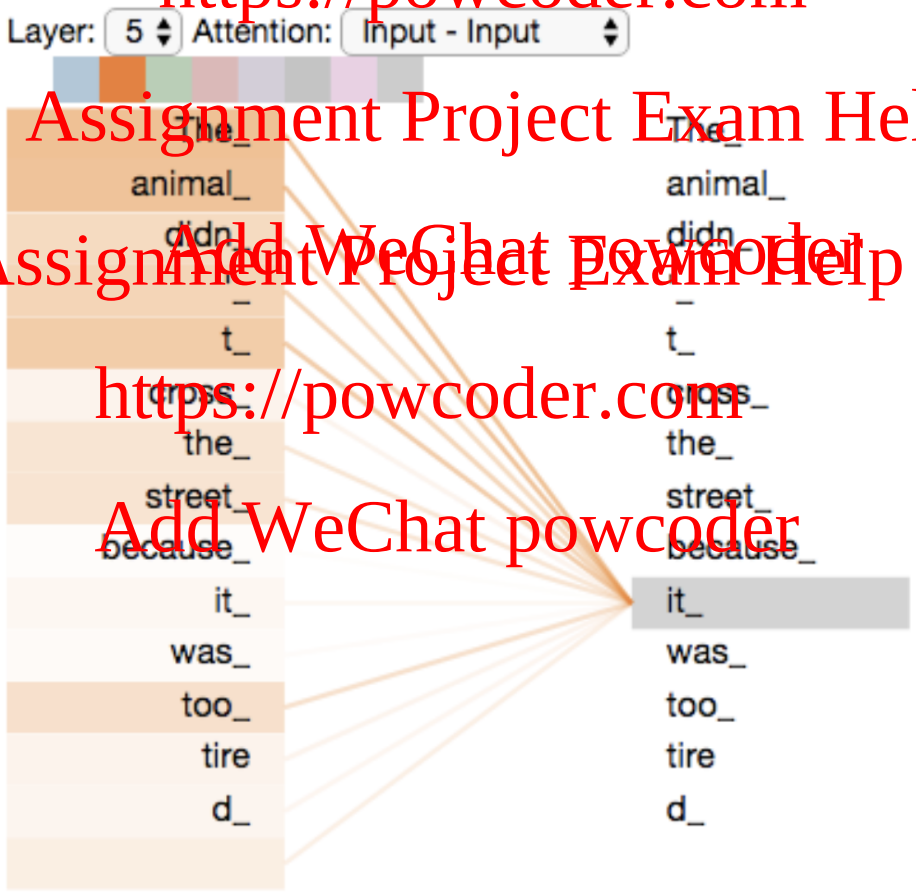
“Self-attention”



“Self-attention”



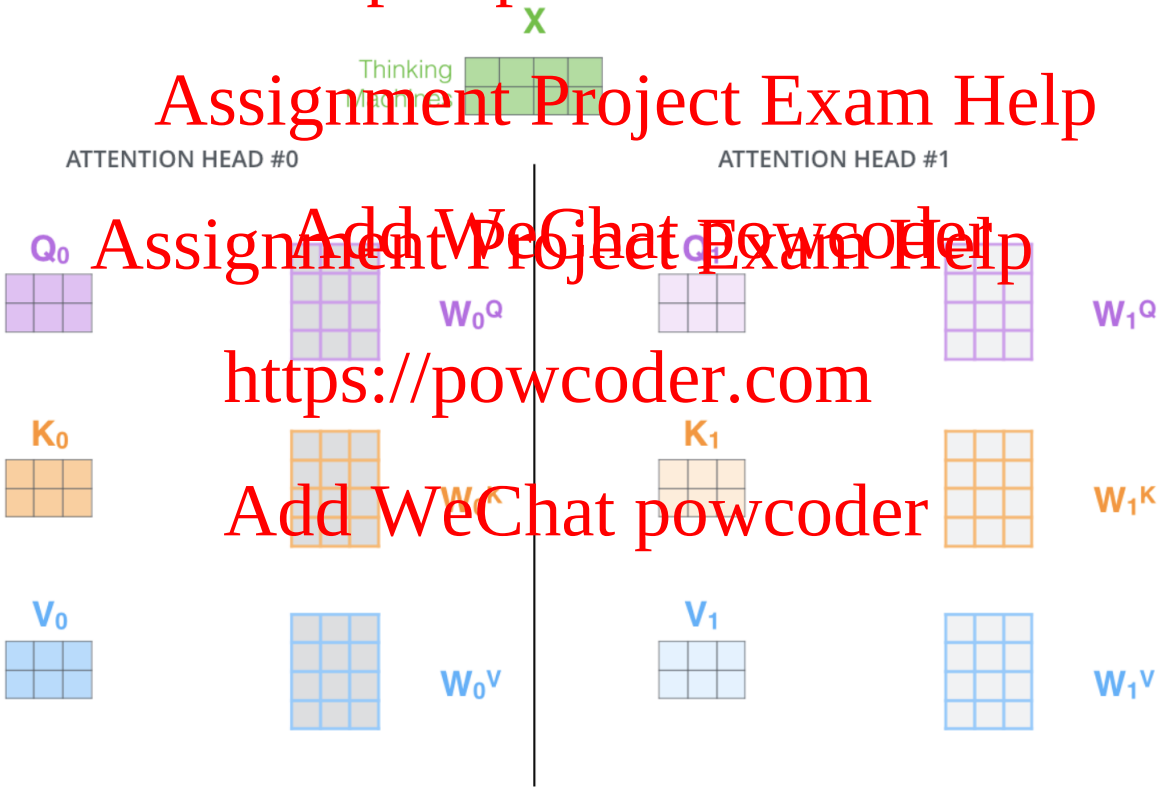
"Self-attention"



"Self-attention"

- Attention is an effective mechanism to model dependencies.
- By varying the keys, values, and queries, we get different variants of attention. We get self-attention by making the key, value and query to be projections of tokens in the same sentence.
- Intuitively self-attention contextualizes a word token in a sentence (or any sequence of word tokens), which is useful in interpreting the word token, which will in turns helps other tasks that build on the understanding of words.

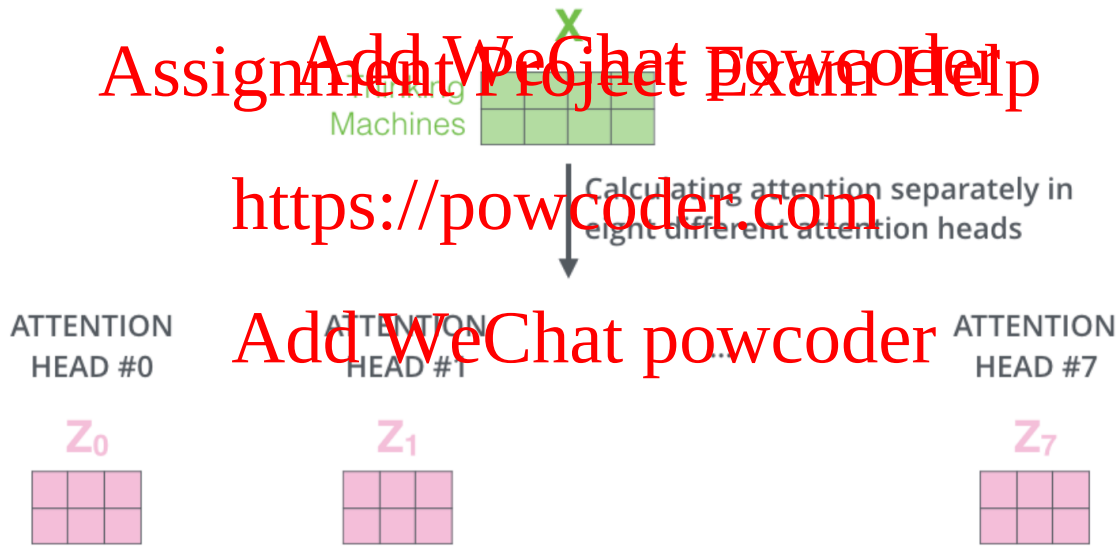
Multiple Attention "heads"



With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.

Multiple Attention "heads"

If we do the same self-attention calculation we outlined above, just eight different times with different weight matrices, we end up with eight different Z matrices

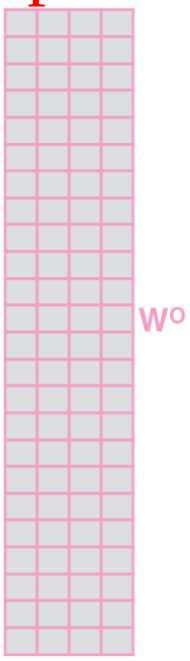


Multiple Attention "heads"

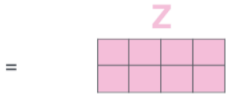
How do we do that? We concat the matrices then multiple them by an additional weights matrix WO.

1) Concatenate all the attention heads

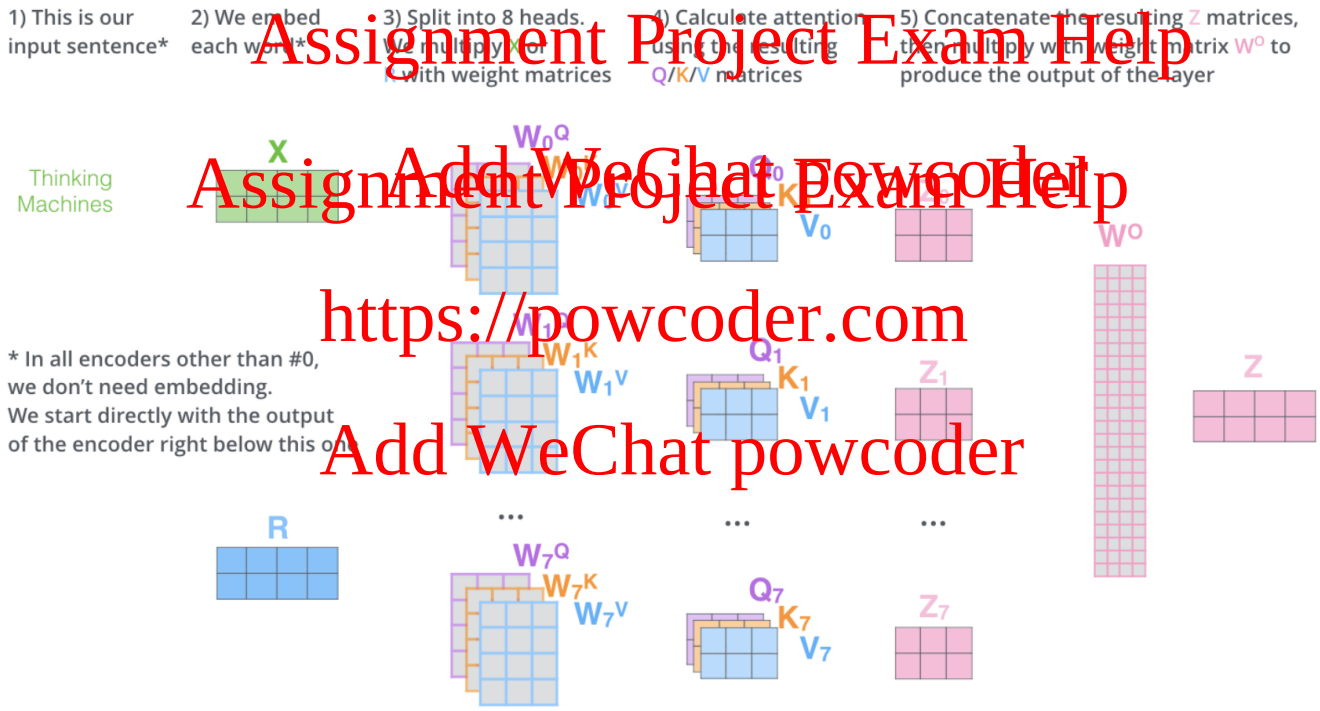
2) Multiply with a weight matrix W^O that was trained jointly with the model



3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN



Multiple Attention "heads"



What does self-attention do?

- Contextualization
- why is contextualization a good thing?
  - "Work out the **solution** in your head."
  - "Heat the **solution** to 75° Celsius."
- Having the same embedding for both instances of "solution" doesn't make sense

The Transformer architecture in (almost) its entirety

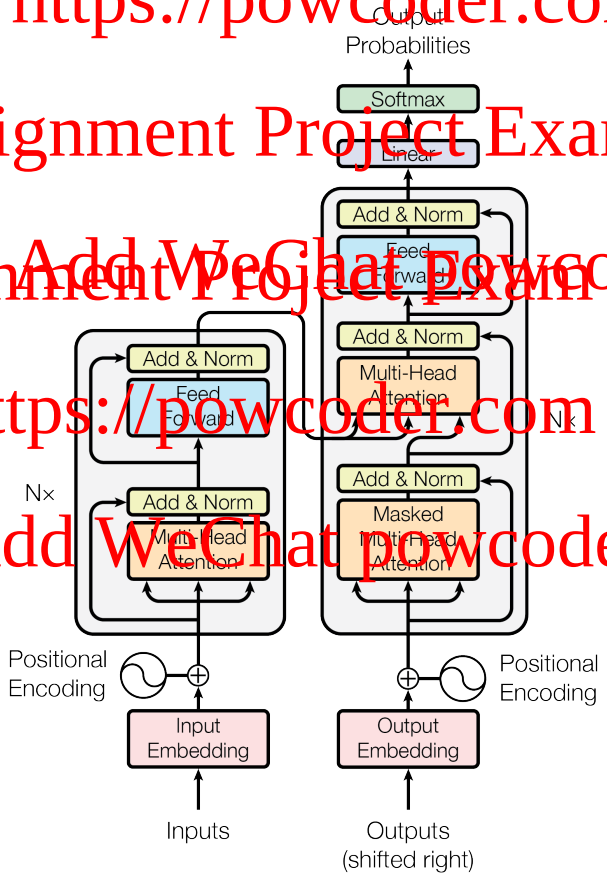


Figure 1: The Transformer - model architecture.

Position Embedding

The original Transformer includes an absolute, precomputed position embedding.

- For word w at position $pos \in [0, L-1]$ in the sequence $\boldsymbol{w} = \{w_0, \cdots, w_{L-1}\}$, with a 4-dimensional embedding e_w, and $d_{model} = 4$, the operation would be:

$$
\begin{aligned}
e_{w'} &= e_w + \left[\sin\left(\frac{pos}{10000^0}\right), \cos\left(\frac{pos}{10000^0}\right), \sin\left(\frac{pos}{10000^{2/4}}\right), \cos\left(\frac{pos}{10000^{2/4}}\right)\right] \\
&= e_w + \left[\sin\left(pos\right), \cos\left(pos\right), \sin\left(\frac{pos}{100}\right), \cos\left(\frac{pos}{100}\right)\right]
\end{aligned}
$$

- The formula to calculate the position embedding is:

$$
\begin{aligned}
PE(pos, 2i) &= \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \\
PE(pos, 2i+1) &= \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)
\end{aligned}
$$

for $i \in [0, d_{model}/2 - 1]$

Position-wise feedforward neural network

- A feed-forward network is applied to each position separately and identically, i.e., with shared weights across positions in the sequence

$$FFN(\boldsymbol{x}) = \boldsymbol{O}_2(ReLU(\boldsymbol{O}_1\boldsymbol{x} + \boldsymbol{b}_1)) + \boldsymbol{b}_2$$

- This is equivalent to applying two one-dimensional convolutional network with one kernel to the sequence.
- Question to think about: why not just use a one-dimensional convolution?

Layer normalization

- Given a minibatch of input of size m, $B = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_m\}$, after applying layer normalization to each sample in the batch, we get a transformed minibatch $B' = \{\boldsymbol{y}_1, \boldsymbol{y}_2, \cdots, \boldsymbol{y}_m\}$, where $\boldsymbol{y}_i = LN_{\gamma,\beta}(\boldsymbol{x}_i)$
- Specifically, layer normalization needs the following steps:
  - Compute the mean and variance of each sample in the batch:

$$\mu_i = \frac{1}{K}\sum_{k=1}^{K} x_{i,k} \qquad \sigma_i^2 = \sum_{k=1}^{K}(x_{i,k} - \mu_i)^2$$

  - Normalize each sample such that the elements in the sample have zero mean and unit variance:

$$\hat{x}_{i,k} = \frac{x_{i,k} - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$$

  - Finally scaling and shifting with γ and β:

$$\boldsymbol{y}_i = \boldsymbol{\gamma} \odot \hat{\boldsymbol{x}}_i + \boldsymbol{\beta} \triangleq LN_{\gamma,\beta}(\boldsymbol{x}_i)$$

Reducing the output space with BPE

BPE is a simple data compression technique

- Initialize the vocabulary with the character vocabulary, and segment each words into a sequence of characters,plus a end-of-word symbol '·'.

- Iteratively count all symbol pairs, and replace each most frequent pair ['A', 'B'] with a new symbol 'AB'. Each merge produces a new symbol which represents a character ngram. Frequent character ngrams are eventually merged into a single symbol.

- The final vocabulary size equals to the initial vocabulary size plus the number of merge operations
- The number of merge operations is a hyperparameter

Python implementation

In [13]:
```
import re, collections, timeit
```

In [14]:
```
def get_stats(vocab):
    pairs = collections.defaultdict(int)
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols)-1):
            pairs[symbols[i],symbols[i+1]] += freq
    return pairs
```

In [15]:
```
def merge_vocab(pair, v_in):
    v_out = {}
    bigram = re.escape(' '.join(pair))
    p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
    for word in v_in:
        w_out = p.sub(''.join(pair),word)
        v_out[w_out] = v_in[word]
    return v_out
```

Python implementation

In [18]:
```
vocab = {' l o w </w>':5, 'l o w e r </w>':2, 'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
    pairs = get_stats(vocab)
    best = max(pairs, key=pairs.get)
    vocab = merge_vocab(best, vocab)
    print(best)
    print (vocab)
```

```
('e', 's')
{' l o w </w>': 5, 'l o w e r </w>': 2, 'n e w es t </w>': 6, 'w i d es t </w>': 3}
('es', 't')
{' l o w </w>': 5, 'l o w e r </w>': 2, 'n e w est </w>': 6, 'w i d est </w>': 3}
('est', '</w>')
{' l o w </w>': 5, 'l o w e r </w>': 2, 'n e w est</w>': 6, 'w i d est</w>': 3}
('l', 'o')
{' lo w </w>': 5, 'lo w e r </w>': 2, 'n e w est</w>': 6, 'w i d est</w>': 3}
('lo', 'w')
{' low </w>': 5, 'low e r </w>': 2, 'n e w est</w>': 6, 'w i d est</w>': 3}
('n', 'e')
{' low </w>': 5, 'low e r </w>': 2, 'ne w est</w>': 6, 'w i d est</w>': 3}
('ne', 'w')
{' low </w>': 5, 'low e r </w>': 2, 'new est</w>': 6, 'w i d est</w>': 3}
('new', 'est</w>')
{' low </w>': 5, 'low e r </w>': 2, 'newest</w>': 6, 'w i d est</w>': 3}
('low', '</w>')
{' low</w>': 5, 'low e r </w>': 2, 'newest</w>': 6, 'w i d est</w>': 3}
('w', 'i')
{' low</w>': 5, 'low e r </w>': 2, 'newest</w>': 6, 'wi d est</w>': 3}
```

Tutorials

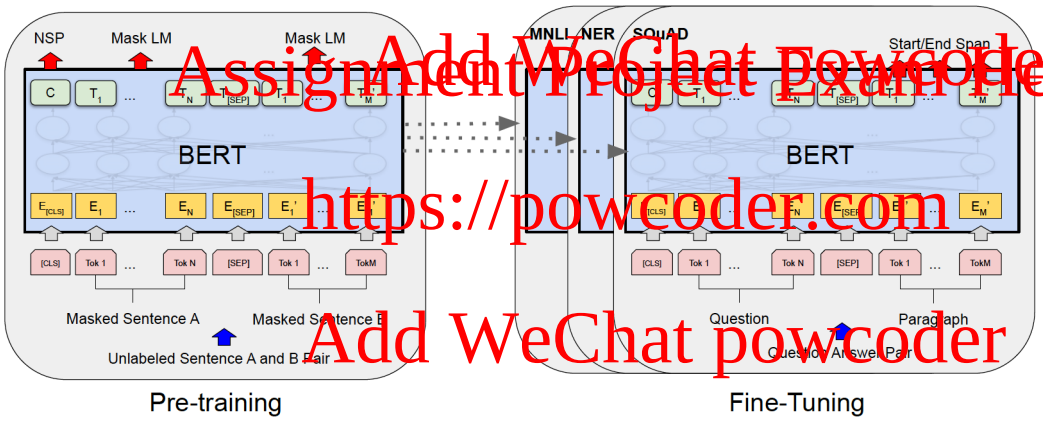
- NMT with Tensorflow: https://github.com/tensorflow/nmt
- Attention visualization: https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/
- Illustrated transformer: http://jalammar.github.io/illustrated-transformer/

BERT: Bidirectional Encoder Representations from Transformers

Input Representations:

- The input can be both a single sentence or a pair of sentences
- "Sentence" here just means a sequence of words.
- Sentences are broken down into WordPieces (token vocab size= 30K).
- The first token of the input sequence is a special symbol [CLS]
- Sentence pars are packed together into one sentence and separated with a special token [SEP]
- Separate token embeddings for the first sentence and the second sentence in a sentence pair.

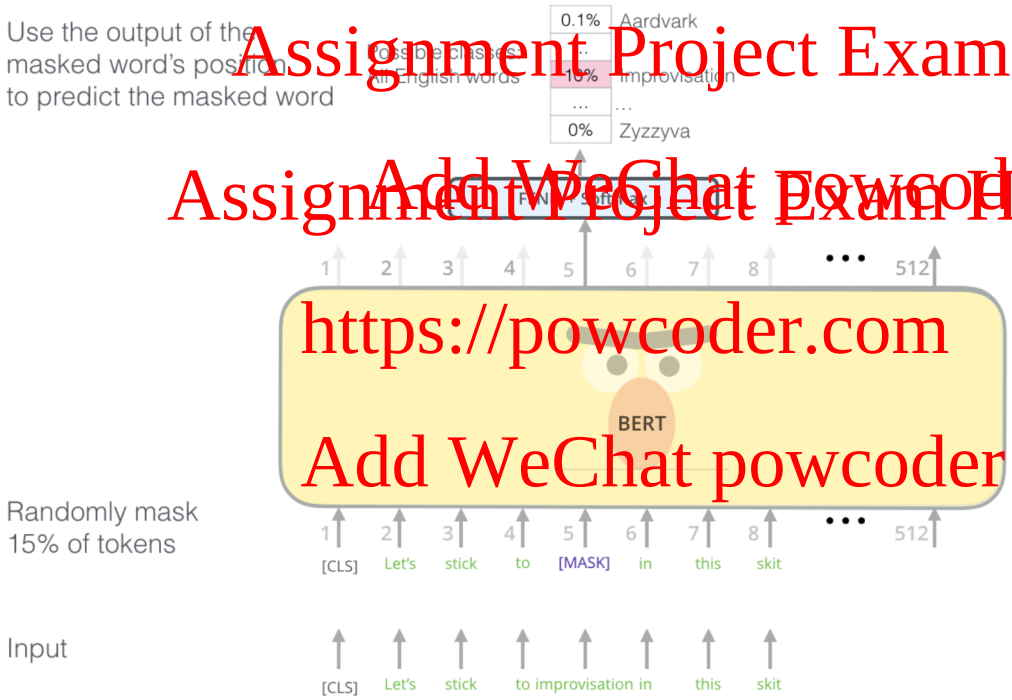
The pretrain-fine tune paradigm



Pre-Training BERT

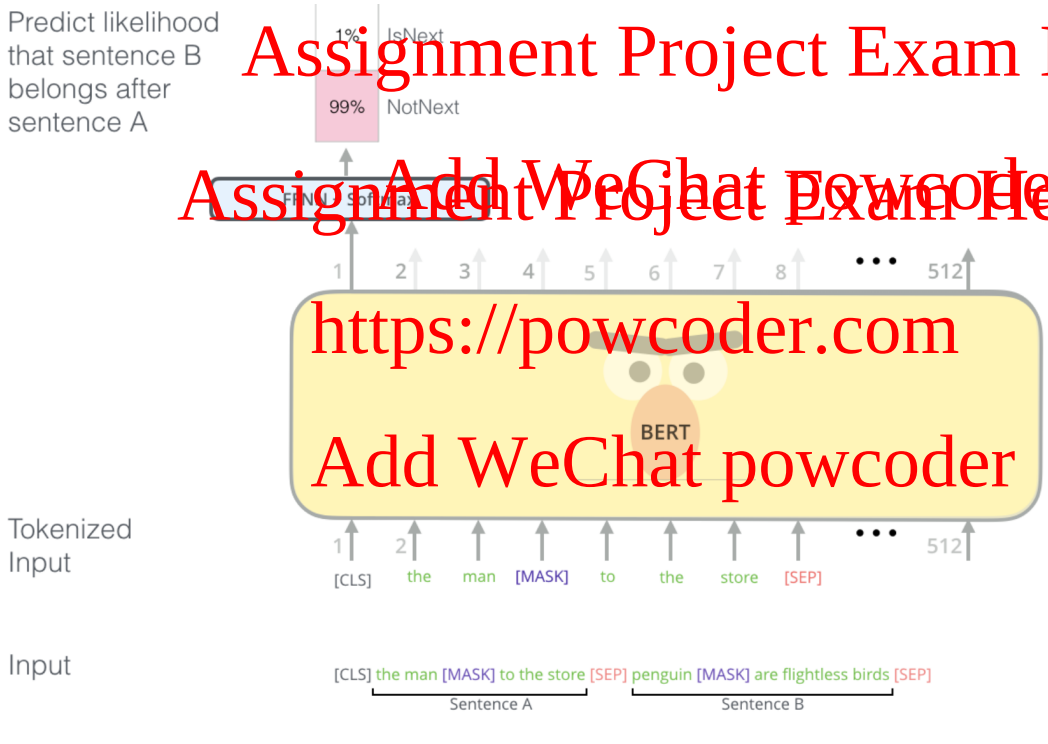
- The Transformer is trained with a Masked Language Model (MLM) task, and a next sentence prediction (NSP) task for sentence pairs.
- The MLM task (or the Cloze task) is trained by predicting 15% of the tokens that are randomly masked.
- The final hidden vectors corresponding to the masked tokens are fed into a softmax over the token vocabulary.

BERT Pretraining with MLM

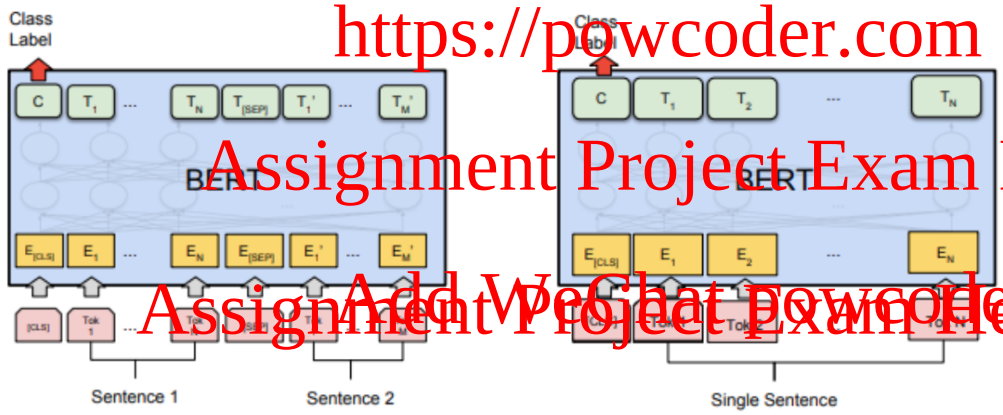


BERT's clever language modeling task masks 15% of words in the input and asks the model to predict the missing word.

BERT next sentence prediction

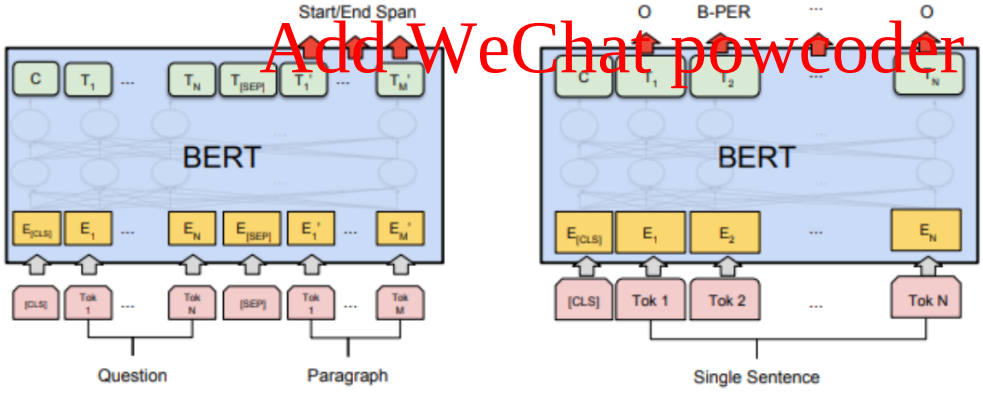


BERT: fine tune to specific tasks



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

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



(c) Question Answering Tasks: SQuAD v1.1

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