Mathematics in Modern Natural Language Modeling

Richard Xu

December 14, 2025

1 A few words

To start this topic, I will first discuss the mathematics in modern natural language processing. By the way, Natural Language Processing (NLP) is one of the most important/exciting applications of artificial intelligence, machine learning and data mining. Contrary to computer vision, NLP will influence the work of many people in the future.

Although neural networks play an important role in NLP. However, we will cover neural networks later in this topic. Therefore, we temporarily avoid talking about N-N.

Also note that while this topic is about techniques in NLP, these techniques can also be applied to other machine learning and data science settings.

2 word embedding

Words are symbols: one may not able to perform arithmetic operations on them. They are suppose to be norminal attributes. (this is something I have discussed in the data mining subject)

However, in many NLP tasks, we need to operate on them as if they are vectors. So we must turn each word into a vector, e.g., the word "machine" → [2.4, 1.2, 1.9 ...]

Once we do so, we can measure how similar or dissimilar between them. We can even perform "arithmetic" on them. A classic example from the the original word2vec paper would be:

$$\text{vec(King)} - \text{vec(man)} + \text{vec(woman)} = \text{vec(Queen)} \tag{1}$$

Mikolov et. al., (2013) "Linguistic Regularities in Continuous Space Word Representations" [1]

2.1 first attempt: one-hot encoding/embedding

a very naive and simple approach is to just use one-hot embedding, after all, students of MATH3836 (subject at last semester) should be very familiar with one-hot vector by now:

$$\begin{bmatrix} \text{``}a\text{''} & 1 & 0 & \cdots & 0 \\ \text{``}abbreviate\text{''} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \text{``}zoology\text{''} & 0 & 0 & \cdots & 1 \end{bmatrix} \tag{2}$$

however, the structure is huge and sparse. Think about the length of those vectors, they are as long as the total number of words in the vocabulary. In English, there is estimated of $40,000$ words.

In addition to the storage waste, one other disadvantage is that every pair of words are exactly $\sqrt{2}$ apart. So you can't really measure the similarity or dissimilarity between them.

The question is, can we do better?

2.2 second attempt: word2vec algorithm

The only data required are paragraphs of words, i.e., no extra human labeling is required. Unlike LDA models, with word-embeddings, we don't even need to know which word appear in which document.

2.2.1 conditional density using (target $\leftrightarrow$ context)

Although the paragraphs of words are the only information for the word2vec algorithm, however, there is still implicit labeling, i.e., we have the position of each word in the document. Therefore, we will use this information alone to construct the learning task:

1. Word2Vec algorithm leverage ("target", "context") relationships to maximize the conditional probabilities.
2. which way should the condition be? the answer is both. Therefore it offers two approaches, i.e., to maximize one of the two conditional densities:
   (a) skip-gram: Pr("context"|"target")
   (b) continuous bag of words (CBOW): Pr("target"|"context")

So we need a methodology to construct context and target:

1. pick window size (odd number)
2. extract all tokens based on this chosen window size
3. remove middle word in each window; this becomes your target word, rest are context

An important question to consider is how to define this probability? Taking Skip-gram as an example (for CBOW, you can calculate it similarly), it is Pr("context" = "a specific word"|"target"), so this discrete probability should be defined on every word in the vocabulary. So it should return a probability vector as large as the vocabulary size!

What function should we consider to construct such a probability vector? you guessed right! Since we need a discrete probability, we can use the Softmax function. Let us generically define these probabilities in terms of the "center" (c) and the "output" word (o):

$$\frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{o' \in \mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)} \tag{3}$$

and the entire softmax vectors can be represented as:

$$\left(\frac{\exp(\mathbf{u}_1^\top \mathbf{v}_c)}{\sum_{o' \in \mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)}, \ldots, \frac{\exp(\mathbf{u}_{o^\star}^\top \mathbf{v}_c)}{\sum_{o' \in \mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)}, \ldots, \frac{\exp(\mathbf{u}_{|\mathcal{V}|}^\top \mathbf{v}_c)}{\sum_{o' \in \mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)} \right) \tag{4}$$

and the index $o^\star$ is the one we are interested in.

2.2.2 skip-gram example: building training set

let's try Skip-gram of a window size 3, from the following sentence:

"the cat sit on the mat"

1. firstly, we generate all possible windows:
   - (a) "the", "cat", "sit", target: cat
   - (b) "cat", "sit", "on" , target: sit
   - (c) "sit", "on ", "the", target: on
   - (d) "on" , "the", "mat", target: the

   as far as this subject is concerned, we assume that for each target word, there are all context words on both sides. Therefore the word "the" will not be a context word

2. from each of the windows, the algorithm generate the input and output pairs for maximizing the conditional probabilities

   $$(\mathbf{x}, y) = \ (\text{target}, \text{context}) \tag{5}$$

   therefore we obtained:

   $$\begin{aligned} \text{“the”} &\leftarrow \text{“cat”} \\ \text{“sit”} &\leftarrow \text{“cat”} \\ \text{“cat”} &\leftarrow \text{“sit”} \\ \text{“on”} &\leftarrow \text{“sit”} \\ \text{“sit”} &\leftarrow \text{“on”} \\ \text{“the”} &\leftarrow \text{“on”} \\ \text{“on”} &\leftarrow \text{“the”} \\ \text{“mat”} &\leftarrow \text{“the”} \end{aligned} \tag{6}$$

3. Using some initialization values, it is possible to take each context, target pair and maximize their conditional probability. Note that we need to maximize a number of conditions, each word has a chance to be a context and target word:

   $$\begin{aligned} &\Pr\left(\text{“the”}|\text{“cat”}\right) \times \Pr\left(\text{“sit”}|\text{“cat”}\right) \times \Pr\left(\text{“cat”}|\text{“sit”}\right) \times \Pr\left(\text{“on”}|\text{“sit”}\right) \times \\ &\Pr\left(\text{“sit”}|\text{“on”}\right) \times \Pr\left(\text{“the”}|\text{“on”}\right) \times \Pr\left(\text{“on”}|\text{“the”}\right) \times \Pr\left(\text{“mat”}|\text{“the”}\right) \end{aligned} \tag{7}$$

   we can express the above mathematically as, when choosing a window size $2m + 1$:

$$\begin{aligned}\mathcal{L} &\equiv \prod_{n=1}^{N} \prod_{-m\leq j\leq m, j\neq 0} \Pr\left(w_{n+j}|w_n\right) \\ \log(\mathcal{L}) &= \sum_{n=1}^{N} \sum_{-m\leq j\leq m, j\neq 0} \log\Big(\Pr\left(w_{n+j}|w_n\right)\Big)\end{aligned} \tag{8}$$

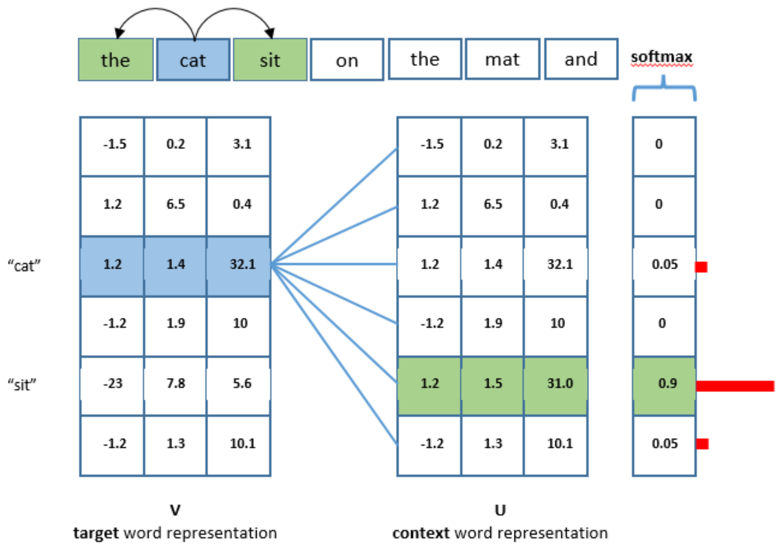
the outer sum tells us that each word n will get to be the "target" word once. The inner product is all the conditional densities using that target word.

4. For each word w_n, it has 2 representations $\mathbf{u}_n$ and $\mathbf{v}_n$, one for output (o) and one for context (c), and of course their size is much smaller than using one-hot!

For example, looking at a particular (i.e, the 2nd term in the product in Eq.(7):

$$\Pr\big(o = \text{“sit”} |\ c = \text{“cat”}\big) \tag{9}$$

we need to maximize the conditional density of $o =$ context word given a $c =$ target word. We illustrate the above using the diagram below:



2.2.3 optimizing Skip-Gram objective function

we need to maximize the probability of $c =$ context word given a $t =$ target word.

In order to compute:

$$\underset{\{\mathbf{v}\},\{\mathbf{u}\}}{\arg\max} \left\{\mathcal{L} \equiv \sum_{n=1}^{N} \sum_{-m\leq j\leq m, j\neq 0} \log\Big(\Pr\left(w_{n+j}|w_n\right)\Big)\right\} \tag{10}$$

We need to calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{v}_t}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{u}_c}$, $\forall \mathbf{v}_t, \mathbf{u}_c \in \mathcal{V}$. However, due to symmetry, we only look at $\arg\max_{\mathbf{v}_t}$. Also, let's look at the derivative of a single term in the sum, where we usually write:

$$\Pr\left(w_{n+j} | w_n\right) \equiv \Pr(o|c) \equiv \Pr(\mathbf{u}_o | \mathbf{v}_c) \tag{11}$$

we can just perform sum of all the derivatives together later:

$$\begin{aligned}
\log(\Pr(o|c)) &= \log\left(\frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{o' \in \mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)}\right) \\
\frac{\partial \log(\Pr(o|c))}{\partial \mathbf{v}_c} &= \frac{\partial \mathbf{u}_o^\top \mathbf{v}_c}{\partial \mathbf{v}_c} - \frac{\partial \log\left(\sum_{o' \in \mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)\right)}{\partial \mathbf{v}_c} \\
&= \mathbf{u}_o - \left(\frac{\partial}{\partial \mathbf{v}_c} \log\left(\sum_{o' \in \mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)\right)\right) \\
&= \mathbf{u}_o - \left(\frac{1}{\sum_{o' \in \mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)} \frac{\partial}{\partial \mathbf{v}_c}\left(\sum_{o' \in \mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)\right)\right) \\
&= \mathbf{u}_o - \left(\frac{1}{\sum_{o' \in \mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)}\left(\sum_{o' \in \mathcal{V}} \frac{\partial}{\partial \mathbf{v}_c} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)\right)\right) \\
&= \mathbf{u}_o - \frac{1}{\sum_{o' \in \mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)}\left(\sum_{o' \in \mathcal{V}} \mathbf{u}_{o'} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)\right) \\
&= \mathbf{u}_o - \frac{\sum_{o' \in \mathcal{V}} \mathbf{u}_{o'} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)}{\sum_{o' \in \mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)} \\
&= \mathbf{u}_o - \sum_{o' \in \mathcal{V}} \frac{\exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)}{\sum_{o'' \in \mathcal{V}} \exp(\mathbf{u}_{o''}^\top \mathbf{v}_c)} \mathbf{u}_{o'} \\
&= \mathbf{u}_o - \sum_{o' \in \mathcal{V}} \Pr(o'|c) \mathbf{u}_{o'} \\
&= \mathbf{u}_o - \mathbb{E}_{o' \sim \Pr(o'|c)}[\mathbf{u}_{o'}]
\end{aligned} \tag{12}$$

Obviously, when $|\mathcal{V}|$ is too large, the computational cost of computing the sum can be too high, there are many NLP mechanisms that can help us reduce the amount of computation. We will discuss them in the subsequent sections.

2.3 Simple CBoW example

very similar to to predict target word given multiple context words, where $\mathbf{u}_c$ become the average of context vectors.

3 Negative sampling

Let's use Skip-Gram model to demonstrate negative sampling. It is optimizing different objective, let $\theta = [\mathbf{u}, \mathbf{v}]$.

We let $\bar{w}$ to denote negative samples, meaning data coming from a negative population $\bar{D}$. In NLP, there could be many example of negative samples. For example, if one wishes to model skip-gram alike model, where words appear in the order of the text. Negative samples may mean words of any random ordering.

$$\begin{aligned}
\theta &= \arg\max_\theta \prod_{(w,c)\in D} \Pr(D=1|w,c,\theta) \prod_{(\bar{w},c)\in \bar{D}} \Pr(D=0|\bar{w},c,\theta) \\
&= \arg\max_\theta \prod_{(w,c)\in D} \Pr(D=1|w,c,\theta) \prod_{(\bar{w},c)\in \bar{D}} (1-\Pr(D=1|\bar{w},c,\theta)) \\
&= \arg\max_\theta \sum_{(w,c)\in D} \log\left(\Pr(D=1|w,c,\theta)\right) + \sum_{(\bar{w},c)\in \bar{D}} \log\left(1-\Pr(D=1|\bar{w},c,\theta)\right) \\
&= \arg\max_\theta \sum_{(w,c)\in D} \log \frac{1}{1+\exp\left[-\mathbf{u}_w^\top \mathbf{v}_c\right]} + \sum_{(\bar{w},c)\in \bar{D}} \log\left(1-\frac{1}{1+\exp\left[-\mathbf{u}_{\bar{w}}^\top \mathbf{v}_c\right]}\right) \qquad \log(\cdot) \quad \text{is monotone} \\
&= \arg\max_\theta \sum_{(w,c)\in D} \log\sigma(-\mathbf{u}_w^\top \mathbf{v}_c) + \sum_{(\bar{w},c)\in \bar{D}} \log\left(\frac{1}{1+\exp\left[\mathbf{u}_{\bar{w}}^\top \mathbf{v}_c\right]}\right) \\
&= \arg\max_\theta \sum_{(w,c)\in D} \log\sigma(\mathbf{u}_w^\top \mathbf{v}_c) + \sum_{(\bar{w},c)\in \bar{D}} \log\sigma(-\mathbf{u}_{\bar{w}}^\top \mathbf{v}_c) \\
&= \arg\max_\theta \sum_{(w,c)\in D} \log\sigma(\mathbf{u}_w^\top \mathbf{v}_c) + \sum_{(\bar{w},c)\in \bar{D}} \log\left(1-\sigma(\mathbf{u}_{\bar{w}}^\top \mathbf{v}_c)\right)
\end{aligned} \tag{13}$$

for the last line we apply the property:

$$\sigma(-t) = 1-\sigma(t) \tag{14}$$

negative sampling based on Skip-Gram model, it is optimizing different objective, let $\theta = [\mathbf{u}, \mathbf{v}]$:

$$\theta = \arg\max_\theta \sum_{(w,c)\in D} \log\sigma(\mathbf{u}_w^\top \mathbf{v}_c) + \sum_{(\bar{w},c)\in \bar{D}} \log\sigma(-\mathbf{u}_{\bar{w}}^\top \mathbf{v}_c) \tag{15}$$

it still has a huge sum term $\sum_{(\bar{w},c)\in\bar{D}}(.)$, so we change to:

$$\theta = \arg\max_\theta \log\sigma(\mathbf{u}_w^\top \mathbf{v}_c) + \sum_{\bar{w}=1}^{k} \mathbb{E}_{\bar{w}\sim P(w)} \log\sigma(-\mathbf{u}_{\bar{w}}^\top \mathbf{v}_c) \tag{16}$$

sample a fraction of negative samples in second terms: $\{\bar{w}\}$ instead of going for $\forall(\bar{w} \neq w) \in \mathcal{V}$. We will discuss more about negative sampling when we talk about generative models.

4 Other interesting word representation

4.1 what is FastText?

Becamse popular in 2016, a library created by Facebook (now called Meta!) research team for efficient learning of word representations by Enriching Word Vectors with Subword Information.

for example, $n = 3$, i.e., 3-grams:

- word: “where”,
- sub-words: “wh”, “whe”, “her”, “ere”, “re”

So how is it different from Word2Vec? Instead of words, we now have ngrams of subwords, what is its advantage?

1. Helpful for finding representations for rare words
2. Give vector representations for words not present in dictionary

4.1.1 computation involving sub-words

We then represent a word $\mathbf{w}$ by the sum of the vector representations of all its n-grams. Concretely, in order to compute an un-normalised score $u(\cdot, \cdot)$ between $\mathbf{w}$ with center word $\mathbf{v}_c$:

Given a word $\mathbf{w}$ (e.g., “where”), $ns(\mathbf{w})$ is the set of n-grams appearing in $\mathbf{w}$, (e.g., “wh”, “whe”, “her”, “ere”, “re”), and $\{ \mathbf{z}_s \}$ is the representation to each individual n-gram $s \in ns(w)$ where its representation is simply the sum:

$$\mathbf{w} \equiv \sum_{s \in w} \mathbf{z}_s \tag{17}$$

therefore, now that the un-normalized inner product between center word $\mathbf{v}_c$ and $\mathbf{w}$ is:

$$u(\mathbf{w}, \mathbf{v}_c) = \exp\left[\sum_{s \in ns(w)} \mathbf{z}_s^\top \mathbf{v}_c\right] \tag{18}$$

4.2 Global Vectors for Word Representation(GloVe)

using the fact that co-occurrence probabilities of words are useful, GloVe learns word vectors through word co-occurrences.

I uses a co-occurrence matrix P where each entry P_{ij} describes how often word i appears in the context of word j. It also allows fast training and scalable to huge corpora. The objective function is:

$$\theta^* = \arg\min_{\theta} \left(J(\theta) \equiv \frac{1}{2} \sum_{\mathbf{u}_i \mathbf{v}_j \in \mathcal{V}} f(P_{ij})(\mathbf{u}_i^\top \mathbf{v}_j - \log P_{ij})^2 \right) \tag{19}$$

it tries to minimize difference:

$$\left(\mathbf{u}_i^\top \mathbf{v}_j - \log P_{ij}\right) \tag{20}$$

therefore, the more frequently two words appear together,more similar their vector representation should be, in terms of their inner product. $f(.)$ is weighting function to "prevent" certain scenarios, for example:

$$P_{ij} = 0 \implies \log P_{ij} = -\infty \implies f(0) = 0 \tag{21}$$

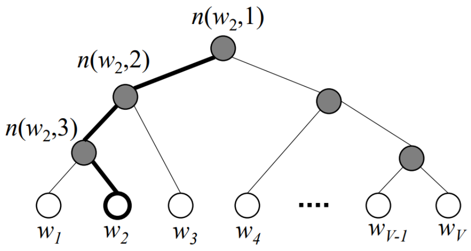
4.3 Hierarchical Softmax

This work was proposed in [2]

for word2vec algorithm, we compute explicitly the word representation of $\mathbf{u}_o$ as before using the softmax function, i.e., Eq.(4):

$$\left(\frac{\exp(\mathbf{u}_1^\top \mathbf{v}_c)}{\sum_{o'\in\mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)}, \ldots, \frac{\exp(\mathbf{u}_{o^\star}^\top \mathbf{v}_c)}{\sum_{o'\in\mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)}, \ldots, \frac{\exp(\mathbf{u}_{|\mathcal{V}|}^\top \mathbf{v}_c)}{\sum_{o'\in\mathcal{V}} \exp(\mathbf{u}_{o'}^\top \mathbf{v}_c)}\right)$$

However, the normalization part is computationally expensive. So the idea is, can we design a word2vec alike model that encapsulating all the information for every single word in the corpus, and then we can compute probabilities without the huge normalization term? Well, one way of doing so is is to represent the entire $\Pr(\mathbf{w}|c) \equiv \Pr(\mathbf{w}|\mathbf{u}_c)$ by:



Xin Rong, word2vec Parameter Learning Explained - it shows the corresponding path of $\mathbf{w}_2$

Note that $\mathbf{w}$ is fixed and $\mathbf{u}_c$ is the input variable. super advantage: $\Pr(\mathbf{w}|\mathbf{v}_c)$ is already a probability by multiplying all probabilities of path, no need to normalize!

4.3.1 definition

1. Each word in this softmax tree, i.e., $\mathbf{w}_i$ has a unique (pre-defined) path, which performs a left or right turn from nodes: $n(w_i, 1), n(w_i, 2), n(w_i, 3), \ldots$ note $n(\cdot)$. these are purely symbolic.
2. the route is defined in such a way that, each child node is from a (LEFT/RIGHT) "channel" of its parent: i.e., $n(w, j+1) = \mathrm{ch}(n(w, j))$, for example:

$$\begin{aligned} n(w_2,2) &= \text{LEFT } (n(w_2,1)) \\ n(w_2,3) &= \text{LEFT } (n(w_2,2)) \\ \underbrace{n(w_2,4)}_{w_2} &= \text{RIGHT}(n(w_2,3)) \end{aligned} \tag{22}$$

3. there are V words in leaf (white node), there are $V-1$ inner (non-leaf) nodes (grey node)
4. each node associates with a vector $\mathbf{v}'$ which is shared among all words going through this node. note that we use $\mathbf{v}'_n$ instead of $\mathbf{v}_n$ to indicate that this is not word representation

4.3.2 compute the probabilities

$$\text{we define:} \quad \text{branch}[.] = \begin{cases} 1: & \text{LEFT} \\ -1: & \text{RIGHT} \end{cases} \tag{23}$$

now that we substitute a context $\mathbf{u}_c$ (skip-gram or CBOW), we can obtain its probability as:

$$\Pr(w|c) = \prod_{j=1}^{L(w)-1} \sigma\Big(\underbrace{\text{branch}\big[n(w,j+1) = \text{ch}(n(w,j))\big]}_{\text{the sign}} \mathbf{v}^{\top}_{n(w,j)} \mathbf{u}_c \Big) \tag{24}$$

note that $\text{branch}\big[n(w,j+1) = \text{ch}(n(w,j))\big]$ is given, i.e., this is determined form the configuration of the tree itself. If one adds up all the probabilities of the leaf node, one can see that, for any input context $\mathbf{u}_c$:

$$\sum_{\mathbf{w} \in \mathcal{V}} p(\mathbf{w}|\mathbf{u}_c) = 1 \tag{25}$$

4.3.3 example of $\Pr(\mathbf{w}_2|\mathbf{u}_c)$ and $\Pr(\mathbf{w}_3|\mathbf{u}_c)$

- at root level, i.e., $n(\cdot,1)$, $n(w_2,1) = n(w_3,1)$ in fact, the values in the set $\big\{n(w_i,1)\big\}_{i=1}^{|\mathcal{V}|}$ all equal
- $n(w_2,2) = n(w_3,2)$ as both w_2 and w_3 share the same path from 1 to 2

  therefore, we can compute the probabilities as:

$$\begin{aligned} \Pr(w_2|c) &= p\big(n(w_2,1), \text{LEFT}\big) p\big(n(w_2,2), \text{LEFT}\big) p\big(n(w_2,3), \text{RIGHT}\big) \\ &= \sigma\Big({\mathbf{v}'_{n(w_2,1)}}^{\top} \mathbf{u}_c \Big) \sigma\Big({\mathbf{v}'_{n(w_2,2)}}^{\top} \mathbf{u}_c \Big) \sigma\Big(-\mathbf{v}'^{\top}_{n(w_2,3)} \mathbf{u}_c \Big) \\ \Pr(w_3|c) &= p\big(n(w_3,1), \text{LEFT}\big) p\big(n(w_3,2), \text{RIGHT}\big) \\ &= \sigma\Big({\mathbf{v}'_{n(w_3,1)}}^{\top} \mathbf{u}_c \Big) \sigma\Big(-{\mathbf{v}'_{n(w_3,2)}}^{\top} \mathbf{u}_c \Big) \end{aligned} \tag{26}$$

5 Overview about Natural Language Processing Tasks

5.1 major NLP tasks

As mentioned earlier, there are too many NLP tasks. Therefore, we list some areas of active research fields:

- machine translation: encoder to decoder

  automatically translate text from one human language to another, for example, English to Chinese. Since 2014, Neural Machine Translation (NMT) dominates!

- text summerization: context to decoder

  1. Extraction-based summarization extracts objects (part-sentences or words) form the long document without modification, i.e., pick the important bits
  2. abstraction-based summarization
     involves paraphrasing sections of the source document

- Q and A: encoder to decoder given context

  the above three (3) may share a design architecture/elements

5.2 down-stream tasks

Too many of applications, and I list a few example works done by my former students.

- natural language generation by learning document corpus
  generate natural language from a machine representation, or for machine to generate semantically-similar texts given a training corpus
- chatbot
  enable human and machine to communicate using natural language
- natural language to cross-domain translation
  1. NLP to image
  2. NLP to animation
- topic modeling
  this is unsupervised learning, tries to assign each document in the document corpus a latent distribution of topics

In the rest of the topics, let me describe some of the modern "classic" NLP tasks:

6 Beam search

Let's talk a little about natural language generation, or NLG. or In Deep NLP terms, the Decoder generates words jointly, so how we may compute:

$$\{\widehat{y}_1, \dots, \widehat{y}_T\} = \arg\max_{y_1,\dots,y_T} \left[\Pr(y_1, \dots, y_T|\mathbf{x}) \equiv \Pr(y_1|\mathbf{x}) \Pr(y_2|y_1, \mathbf{x}), \dots, \Pr(y_T|y_1, \dots, y_{T-1}, \mathbf{x}) \right] \quad (27)$$

but as the depth of the tree, i.e., T became large, compute probability for each and every single chain became unmanageable. Let's look at the following two extreme scenarios:

1. select all: tree-width $= N$, so we get answer to be:

$$\{\widehat{y}_1, \dots, \widehat{y}_T\} = \arg\max_{y_1,\dots,y_T} \Big[\Pr(y_1|\mathbf{x}) \Pr(y_2|y_1, \mathbf{x}) \Pr(y_3|y_1, y_2, \mathbf{x}) \dots, \Pr(y_T|y_1, \dots, y_{T-1}, \mathbf{x}) \Big] \tag{28}$$

   in each depth, keep (select) full width N, until its full depth T, before select a best path (accurate, but computationally infeasible): N^T paths!

2. select one: tree-width $= 1$, greedy algorithm (def. making locally optimal choice at each stage, to "approximately" a global optimum)

   select best word at each depth t: choose one branch in a depth, and discard rest of sibling branches
   (fast & storage efficient, but accuracy-wise bad)

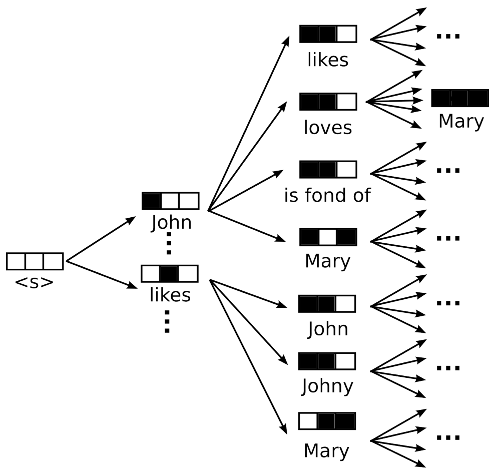


image credit: internet

$$\{\widehat{y}_1, \dots, \widehat{y}_T\} \approx \{\tilde{y}_1, \dots, \tilde{y}_T\}$$
$$= \Big\{ \tilde{y}_1 \equiv \arg\max_{y_1} \Pr(y_1|\mathbf{x}),\ \tilde{y}_2 = \arg\max_{y_2} \Pr(y_2|\tilde{y}_1, \mathbf{x}), \dots, \tilde{y}_T \equiv \arg\max_{y_T} \Pr(y_T|red\tilde{y}_1, \dots, \tilde{y}_{T-1}, \mathbf{x}) \Big\} \tag{29}$$

6.1 trade off?

Obviously, having tree width of N and 1 are both not ideal, so we go for a comprise:
easy , why don't we select tree width W, such that

$$1 < W < N \tag{30}$$

loop from 1 to T, at each depth t:

1. use W most probable branches chosen at the previous depth
2. extend each W branch by depth $+1$, and to generate $W \times N$ candidate branches
3. choose the W most probable branches, and go for next iteration

6.2 beam-search: normalization

1. problem: of beam-search is that the words are generated from:

$$\Pr(y_1, \ldots, y_T|\mathbf{x}) = \Pr(y_1|\mathbf{x}) \Pr(y_2|y_1, \mathbf{x}) \Pr(y_3|y_1, y_2, \mathbf{x}) \Pr(y_T|y_1, \ldots, y_{T-1}, \mathbf{x}) \tag{31}$$

   therefore, shorter the word, higher the probability (less to multiply)

2. solution: beam-search normalization

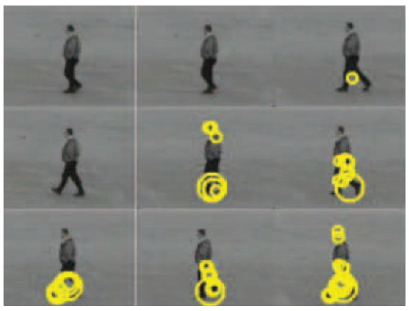
   one simple example: Andrew Ng's (2017) DL course:

$$\begin{aligned} \{\widehat{y}_1, \ldots, \widehat{y}_T\} &= \arg\max_{y_1,\ldots,y_T} \Pr(y_1, \ldots, y_T|\mathbf{x}) = \Pr(y_1|\mathbf{x}) \Pr(y_2|y_1, \mathbf{x}), \ldots, \Pr(y_T|y_1, \ldots, y_{T-1}, \mathbf{x}) \\ &= \arg\max_{y_1,\ldots,y_T} \left(\frac{1}{T^\alpha} \sum_{t=1}^{T} \log \Pr(y_t|y_1, \ldots, y_{t-1}, \mathbf{x}) \right) \end{aligned} \tag{32}$$

the method is not new, it's called geometry mean:

$$\left(\prod_{i=1}^{T} p_i \right)^{\frac{1}{T}} = \exp \left[\frac{1}{T} \sum_{i=1}^{T} \log p_i \right] \tag{33}$$

I also used geometry mean to address problem of variable number of features at each image in a sequence


HMM-MIO: An enhanced hidden Markov model for action recognition [3]

6.3 more sophisticated normalization

Based on the paper, Wu et. al., (2016), "Google's Neural Machine Translation System: Bridging the Gap". The aim is to maximize scores generated by the model:

$$s(Y, X) = \frac{1}{\text{lp}(Y)} \log P(Y|X) + \text{cp}(X, Y) \tag{34}$$

where X is the input sentence and the Y is the output sentence. We use $|Y|$ to indicate the length of Y. Notice that there are two penalties used, the first one is "length penalty" or lp, the second is "coverage penalty" or cp.

6.3.1 Length penalty lp(Y)

Since we try to maximize $s(Y, X)$. Therefore, you would like lp(Y) to be as small as possible. It is inversely proportional to $s(Y, X)$. In words, it tries encourage the length of output Y do not become too long.

the most obvious way of defining lp is $\text{lp}^* = |Y|^\alpha$, similar to the previous method, i.e., Eq.(32), where α can be used to tune rate of increase.

However, a different and more sophisticated normalization lp(Y) can be defined:

$$\text{lp}(Y) = \frac{(5 + |Y|)^\alpha}{(5 + 1)^\alpha} \tag{35}$$

Let's plot them both so we can see their difference:

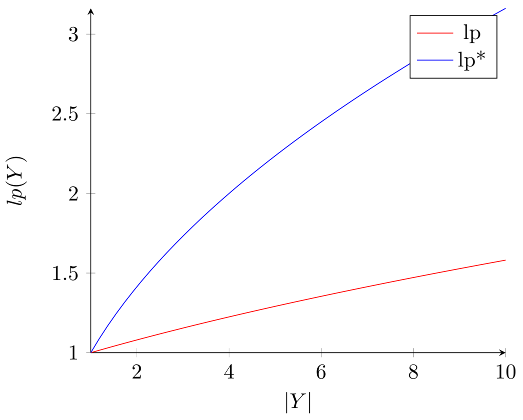


Figure 1: compare and constrast the two length panelties, $\text{lp}(Y) = \frac{(5+|Y|)^\alpha}{(5+1)^\alpha}$ and $\text{lp}^*(Y) = |Y|^\alpha$

When $|Y|$ is small, the difference between lp(Y) and lp$^*(Y)$ is not as significant. However, for lp(Y), the penalty increases "slowly" as $|Y|$ increases, while lp$^*(Y)$ increases much faster. In addition, the effect of having the value 5 in it (based on empirical studies presumably). This can be seen as:

6.3.2 coverage penalty

The second part of the penalty is the coverage penalty or cp, where the objective function is also maximized:

$$s(Y,X) = \frac{1}{\mathrm{lp}(Y)} \log P(Y|X) + \mathrm{cp}(X,Y) \tag{36}$$

it needs also to maximize coverage penalty:

$$\mathrm{cp}(X,Y) = \beta \sum_{j=1}^{|X|} \log \Big(\min \big(\sum_{i=1}^{|Y|} a_{i,j}, 1.0 \big) \Big) \tag{37}$$

- where $a_{i,j}$ is attention probability of i-th target decoder word on j-th source encoder word (technically via their "states", but we use word in here). Sorry to bring this up in here before we formally talk about it in section (7). If we can portray attention as a matrix, then:

$$\mathbf{a} = \begin{bmatrix} a_{i=1,j=1} & a_{i=1,j=2} & a_{i=1,j=3} \\ a_{i=2,j=1} & a_{i=2,j=2} & a_{i=2,j=3} \\ a_{i=3,j=1} & a_{i=3,j=2} & a_{i=3,j=3} \end{bmatrix} \tag{38}$$

- we know that:

$$\sum_{j=1}^{|X|} a_{i,j} = 1 \qquad \text{and} \qquad \sum_{i=1}^{|Y|} a_{i,j} \neq 1 \text{ in general} \tag{39}$$

  i.e, each row of **a** needs to add-up to 1 but each column of **a** does not need to add-up to 1

- think **a** as a matrix of size $|Y| \times |X|$, which we distribute a total mass of $|Y|$ in value among all of its elements, for example, $|X| = |Y| = 3$:

$$\underbrace{\mathbf{a} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}}_{\text{minimized cp}(X,Y)} \qquad \underbrace{\mathbf{a} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{maximized cp}(X,Y)} \tag{40}$$

- favor translations that fully cover source sentence according to the attention module, i.e., there is one corresponding encoder word for each decoder word.. (jokingly, the reverse may correspond how I translate Cantonese to English/Mandarin at the moment)

6.4 More sophisticated normalization

- lastly, one may encourage the decoder to be longer than the encoder (which is fixed as we do know the input): opennmt.net/OpenNMT/translation/beam_search/. It is natural to assume that the longer the sentence from the input, the longer the sentence of the output.

$$\mathrm{ep}(X, Y) = \gamma \frac{|Y|}{|X|} \tag{41}$$

7 Sequence to Sequence with Attention

this work was proposed in [4] or [5]. The pure RNN-base seq2seq was described in [6].

- encoders have hidden states: $\{\mathbf{z}_1, \ldots, \mathbf{z}_m\} \in \mathbb{R}^d$. This is the latent representation of words feed into the encoder. (language to be translated from)
- decoders have hidden states: $\{\mathbf{h}_1, \ldots, \mathbf{h}_n\} \in \mathbb{R}^d$. This is the latent representation of words of the decoder. (language to be translated to)
- compute dot-product: $\mathbf{h}_i^\top \mathbf{z}_j$. this is how aligned (in the latent representation sense) between i^{th} state in the encoder and j^{th} state in the decoder.
- attentions between i^{th} decoder state and j^{th} encoder state is:

$$a_{ij} \equiv a_{i \leftarrow j} = \frac{\exp\left(\mathbf{h}_i^\top \mathbf{z}_j\right)}{\sum_{j'=1}^m \exp\left(\mathbf{h}_i^\top \mathbf{z}_{j'}\right)} \tag{42}$$

  this is normalized (in terms of probability). Showing how much does i^{th} word in the decoder receives from j^{th} word in the encoder. In terms of translation, it means in translating i^{th} word, how much attention (what proportion) does it receive from the j^{th} word from the original language. Obviously:

$$\sum_{j=1}^m a_{ij} = 1 \tag{43}$$

- each i^{th} decoder has $\mathbf{a}_i = \{a_{i,1}, \ldots, a_{i,m}\}$ attention weights/probability vector of the encoder
- condition vector $\mathbf{c}_i$ for each decode word i:

$$\mathbf{c}_i = \sum_{j=1}^m a_{i,j} \mathbf{z}_j \tag{44}$$

  it is a convex combination of encoder states each weighted by its contribution/attention to decoder state i. Therefore $\mathbf{c}_i$ contains the attention information.
- new augmented decoder state $\tilde{\mathbf{h}}_t$:

$$\tilde{\mathbf{h}}_t = [\mathbf{c}_i; \mathbf{h}_t] \in \mathbb{R}^{2d} \tag{45}$$

7.0.1 some new variations

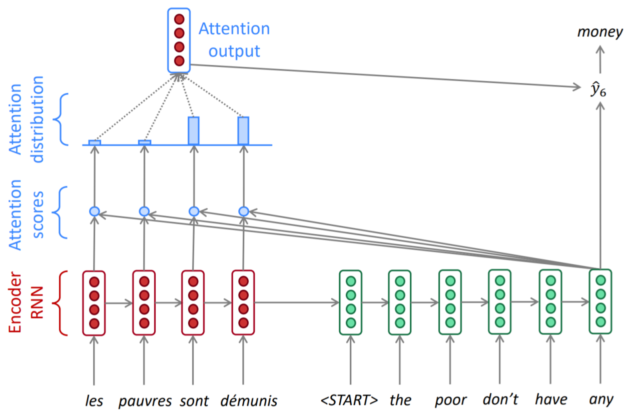


image from internet

In recent NLP literature, many version exist for dot-product: $\mathbf{h}_i^\top \mathbf{z}_j$

1. enable h and z have different dimensionality

$$\mathbf{h}_i^\top \mathbf{W} \mathbf{z}_j \tag{46}$$

2. alternatively:

$$\mathbf{v}_i^\top \tanh(\mathbf{W_1} \mathbf{h}_i + \mathbf{W_2} \mathbf{z}_j) \tag{47}$$

$(\mathbf{v}_i, \mathbf{W_1}, \mathbf{W_2})$ are parameters of this dot-product. Pointer Networks uses this!

7.1 Improvements

- issue one: decoder sometimes repeat themselves (e.g. "machine learning machine learning ..."). It happens when the attention probably vector for $\{\mathbf{h}_t\}$ are similar

  solution we will use [7]

  - coverage vector Sum of attention distributions so far, say up to word t, we have:

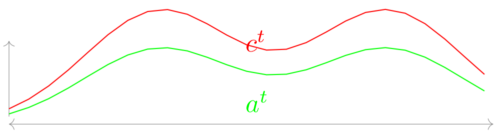
$$\mathbf{c}^t = \sum_{t'=0}^{t-1} \mathbf{a}^{t'} \tag{48}$$

  - penalize overlap between coverage vector $\mathbf{c}^t$ and new attention distribution $\mathbf{a}^t$:

$$\text{covloss}_t = \sum_i \min(\mathbf{a}_i^t, \mathbf{c}_i^t) \tag{49}$$

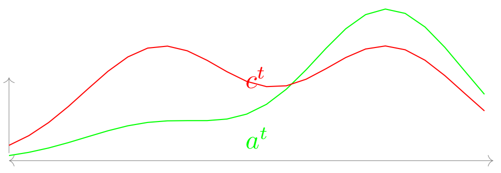
- the above equation can be understood as follows. (although we are using a continuous representation):

  imagine $c_i^t \geq a_i^t \forall i$, then $\text{covloss}_t = 1$, which is its maximum. this happens when covloss_t is a multilicative envelop of a^t:

  $\text{covloss}_t = \sum_i a_i^t = 1$ at maximum

  this is the situation that you want to avoid, as the current attention is $\mathbf{a}^t$ has the same shape as all previous attentions $\mathbf{c}^t$. So it is more likely to generate repeating words. alternatively, the following is desirable:

  $\text{covloss}_t < 1$

- in essence, minimizing covloss_t makes $\mathbf{a}^t$ distributed differently to $\mathbf{c}^t$.
- not otherwise stated, however, I believe $\mathbf{c}^t$ may also be calculated as:

$$\mathbf{c}^t = \frac{1}{t} \sum_{t'=0}^{t-1} \mathbf{a}^{t'} \tag{50}$$

7.2 Improvement

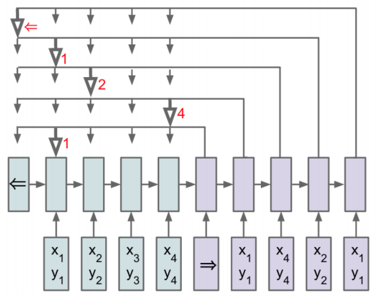
- issue two decoder may not able to translate "out-of-vocabulary words" such as names of a company

- suppose to have the following text summerization task:
  - original text:
    "The ShatinCo has made all reasonable efforts to ensure that this material has been reproduced with the consent of TaiwaiCo"
  - summerized text:
    "TaiwaiCo allowed ShatinCo to reuse its content"
  - some of the word should appear as is it
- RNN-based summarization may replace "Mary" with "Jane" and "Sydney" with "Melbourne" since these word embedding tend to cluster (and hence their dot product are similar!)
- solution "Pointer Networks" may be handy to comes to help!

7.2.1 What is Pointer Networks anyway?

In Pointer Networks [8] described as follows:

For pointer networks, the input is a given set of data points. Note that the raw input data has no sequence. they form purely a set. By designing a network stricture that looks like this:

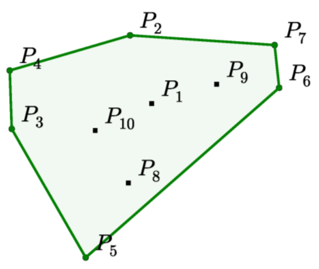


It will output a series of positions relative to the original data. In a way it reorders the original "arbitrary" input sequence (since the original data does not form any order). Now that the new order becomes useful.

So for training data, the "correct order" of data is its output y.

7.2.2 use for mathematics

it could solve combinatorial geometry problems:



we compute:

$$\Pr(C_i|C_1, \ldots, C_{i-1}, \mathcal{P}) \tag{51}$$

where $\mathcal{P} = \{P_1, \ldots\}$

Question for the students: how do you get the training data to solve the above problem? By the way, this is an example of how one can use neural networks to solve difficult mathematics problems.

7.2.3 back to NLP

- "Seq2Seq with attention" is to predict content of next word
- "Pointer Networks" is to predict next position of encoding sequence

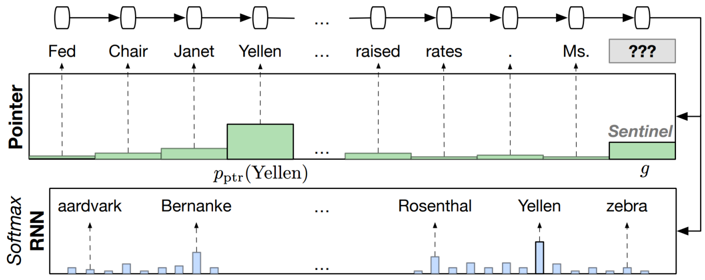
- uses a different "inner product " function: $v_i^\top \tanh(\mathbf{W_1} h_i + \mathbf{W_2} z_j)$

$$a_{ij} = \frac{\exp\left(\mathbf{v}_i^\top \tanh(\mathbf{W_1}\mathbf{h}_i + \mathbf{W_2}\mathbf{z}_j)\right)}{\sum_{j'=1}^{m} \exp\left(\mathbf{v}_i^\top \tanh(\mathbf{W_1}\mathbf{h}_i + \mathbf{W_2}\mathbf{z}_{j'})\right)} \tag{52}$$

- now that we apply Pointer Network to "copy" rare words from encoder to decoder, what about generating words that don't appear in the encoder?
- the answer is a mixture model that does both copy (extraction) and generation (abstraction)

7.2.4 Pointer Sentinel Mixture Models

- Pointer sentinel mixture models [9]
- combines abstraction and extraction together



$$p(\text{"}Yellen\text{"}) = g \times p_{\text{vocab}}(\text{"}Yellen\text{"}) + (1-g) \times p_{\text{ptr}}(\text{"}Yellen\text{"}) \tag{53}$$

- g is mixture gate, uses sentinel to dictate how much probability mass to give to vocabulary
- note that PSMM paper doesn't discuss seq2seq, instead it is about generate $\Pr(y_N | w_1, \ldots, w_{N-1})$

8 Transformer Architecture

8.1 Dot-Product Attention (DPA)

8.1.1 single query $\mathbf{q}$

The very famous paper, "Attention Is All You Need" (Google) [10] introduced the concept of attention, which is a key component of the Transformer architecture

We are given a single query vector $\mathbf{q}$, and matrix of keys $\mathbf{K}$ and values $\mathbf{V}$:

$$\mathbf{q} \equiv \underbrace{\begin{bmatrix} - & \mathbf{q} & - \end{bmatrix}}_{\in \mathbb{R}^{1\times d_k}} \qquad \mathbf{K} \equiv \underbrace{\begin{bmatrix} - & \mathbf{k}_1 & - \\ - & \dots & - \\ - & \mathbf{k}_m & - \end{bmatrix}}_{\in \mathbb{R}^{m\times d_k}} \qquad \mathbf{V} \equiv \underbrace{\begin{bmatrix} - & \mathbf{v}_1 & - \\ - & \dots & - \\ - & \mathbf{v}_m & - \end{bmatrix}}_{\in \mathbb{R}^{m\times d_v}} \tag{54}$$

and let $(\mathbf{q}, \mathbf{K}, \mathbf{V})$ be tuples. Bear in mind that in Eq.(54), $\mathbf{q}$ is a row vector. This is to conform with Eq.(60), where all the $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ have each element expressed as a row vector.

the Dot-Product Attention (DPA) is defined as:

$$\begin{aligned}
\implies \mathbf{q}\mathbf{K}^\top &= \begin{bmatrix} \underbrace{\mathbf{q}\mathbf{k}_1^\top}_{\in\mathbb{R}} & \dots & \underbrace{\mathbf{q}\mathbf{k}_m^\top}_{\in\mathbb{R}} \end{bmatrix} \\
\implies A(\mathbf{q}, \mathbf{K}, \mathbf{V}) &\equiv \text{softmax}(\mathbf{q}\mathbf{K}^\top)\mathbf{V} \\
&= \text{softmax}\big(\begin{bmatrix} \mathbf{q}\mathbf{k}_1^\top & \dots & \mathbf{q}\mathbf{k}_m^\top \end{bmatrix} \big) \begin{bmatrix} - & \mathbf{v}_1 & - \\ - & \dots & - \\ - & \mathbf{v}_m & - \end{bmatrix} \\
&= \sum_{i=1}^{m} \underbrace{\frac{\exp[\mathbf{q}\mathbf{k}_i^\top]}{\sum_j \exp[\mathbf{q}\mathbf{k}_j^\top]}}_{\mathbf{a}_i} \mathbf{v}_i \\
&\in \mathbb{R}^{d_\mathbf{v}}
\end{aligned} \tag{55}$$

8.1.2 Scaled Dot-Product Attention (SDPA)

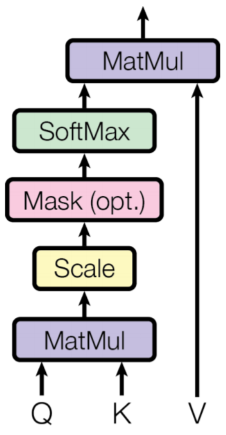
the problem starts as d_k becomes larger, variance of $\mathbf{q}\mathbf{k}^\top$ increases. hen as a result, some dot product values gets very large, with exp(.), softmax $\mathbf{p}$ gets peaky! Remember derivative of cross-entropy between softmax p and y is:

$$\begin{aligned}
\mathbf{C}(\mathbf{z}) &= -\sum_{k=1}^{K} y_k \left[\log\left(p_k\right)\right] = -\sum_{k=1}^{K} y_k \left[\log\left(\frac{\exp^{z_k}}{\sum_l \exp^{z_l}}\right)\right] \\
\implies \frac{\mathbf{C}(\mathbf{z})}{\partial \mathbf{z}} &= (\mathbf{p} - \mathbf{y})
\end{aligned} \tag{56}$$

with a peaky softmax, lots of element in gradient vector $\frac{\mathbf{C}(\mathbf{z})}{\partial \mathbf{z}}$ are zero!

the solution then becomes that of scaling by length of d_k:

$$A(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V} \tag{57}$$



"mask (opt.)" is only used at decoder during training

8.1.3 in the case of "seq2seq with attention"

we have:

$$\begin{aligned} \mathbf{q} &\equiv \mathbf{h}_i \\ \mathbf{k}_i &= \mathbf{v}_i = \mathbf{z}_i \\ \implies A(\mathbf{q}, \mathbf{K}, \mathbf{V}) &\equiv A({\color{blue}\mathbf{h}_i}, {\color{red}\mathbf{Z}}, {\color{red}\mathbf{Z}}) = c_i \end{aligned} \tag{58}$$

where is our conditional or context vector. In order to confirm the notation in Eq.(54), we have assumed $\mathbf{h}_i$ to be a row vector, so instead of $\mathbf{h}_i^\top \mathbf{z}_j$, we express it as $\mathbf{h}_i \mathbf{z}_j^\top$:

$$a_{ij} = \frac{\exp\left(e_{i,j}\right)}{\sum_{t=1}^{m} \exp\left(e_{i,t}\right)} = \frac{\exp\left(\mathbf{h}_i \mathbf{z}_j^\top\right)}{\sum_{t=1}^{m} \exp\left(\mathbf{h}_i \mathbf{z}_t^\top\right)} \tag{59}$$

8.1.4 multiple queries

now we have multiple $\mathbf{Q} = \{\mathbf{q}_i\}$, e.g., N words in the decoder, we now have $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$ expressed as:

$$\mathbf{Q} \equiv \underbrace{\begin{bmatrix} - & \mathbf{q}_1 & - \\ - & \dots & - \\ - & \mathbf{q}_n & - \end{bmatrix}}_{n \times d_k} \qquad \mathbf{K} \equiv \underbrace{\begin{bmatrix} - & \mathbf{k}_1 & - \\ - & \dots & - \\ - & \mathbf{k}_m & - \end{bmatrix}}_{m \times d_k} \qquad \mathbf{V} \equiv \underbrace{\begin{bmatrix} - & \mathbf{v}_1 & - \\ - & \dots & - \\ - & \mathbf{v}_m & - \end{bmatrix}}_{m \times d_v} \tag{60}$$

$A(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\mathbf{Q}\mathbf{K}^\top)\mathbf{V}$ can be expressed as follows:

$$\implies \mathbf{Q}\mathbf{K}^\top = \begin{bmatrix} \mathbf{q}_1\mathbf{k}_1^\top & \dots & \mathbf{q}_1\mathbf{k}_m^\top \\ \dots & \dots & \dots \\ \mathbf{q}_n\mathbf{k}_1^\top & \dots & \mathbf{q}_n\mathbf{k}_m^\top \end{bmatrix} \tag{61}$$

in the case of self attention, where $m = n$, then $\mathbf{QK}^\top$ is a square matrix of $n \times n$.

$$\begin{aligned} \implies \mathbf{O} &\equiv \text{softmax}(\mathbf{QK}^\top)\mathbf{V} \\ &= \mathbf{AV} \\ &= \underbrace{\begin{bmatrix} \text{softmax}\big(\begin{bmatrix}\mathbf{q}_1\mathbf{k}_1^\top & \dots & \mathbf{q}_1\mathbf{k}_m^\top\end{bmatrix} \big) \\ \vdots \\ \text{softmax}\big(\begin{bmatrix}\mathbf{q}_n\mathbf{k}_1^\top & \dots & \mathbf{q}_n\mathbf{k}_m^\top\end{bmatrix} \big) \end{bmatrix}}_{n\times m} \underbrace{\begin{bmatrix} - & \mathbf{v}_1 & - \\ - & \dots & - \\ - & \mathbf{v}_m & - \end{bmatrix}}_{m\times d_v} \\ &= \underbrace{\begin{bmatrix} \underbrace{\sum_{i=1}^{m} \frac{\exp[\mathbf{q}_1\mathbf{k}_i^\top]}{\sum_j \exp[\mathbf{q}_1\mathbf{k}_j^\top]}\mathbf{v}_i} & \text{row vector 1 in } \mathbb{R}^{d_v} \\ & \vdots \\ \underbrace{\sum_{i=1}^{m} \frac{\exp[\mathbf{q}_n\mathbf{k}_i^\top]}{\sum_j \exp[\mathbf{q}_n\mathbf{k}_j^\top]}\mathbf{v}_i} & \text{row vector } n \text{ in } \mathbb{R}^{d_v} \end{bmatrix}}_{n\times d_v} \end{aligned} \tag{62}$$

looking at the i-th row of $\mathbf{A}$, i.e., $\mathbf{a}_i$, it can be expressed as a weighted sum of the rows of $\mathbf{V}$.

$$\mathbf{a}_i\mathbf{V} = \begin{bmatrix} a_{i,1} & \dots & a_{i,m} \end{bmatrix} \begin{bmatrix} - & \mathbf{v}_1 & - \\ \vdots & \ddots & \vdots \\ - & \mathbf{v}_m & - \end{bmatrix} = \sum_{i=1}^{m} a_{i,j}\mathbf{v}_i \tag{63}$$

The output $\mathbf{O}$'s dimension has number of row to be the same as $\mathbf{Q}$, and column dimension to be d_V. This means that it has the number of elements of $\mathbf{Q}$, and each element is the same dimension as $\mathbf{v}_i$.

8.2 Self-attention and Multi-head Attention

The Transformer architecture uses multi-head attention, which is a key component of the Transformer architecture.Generally, input vectors are $(\mathbf{Q}, \mathbf{K}, \mathbf{V})$

When we let:

$$\mathbf{Q} = \mathbf{XW}_Q, \quad \mathbf{K} = \mathbf{XW}_K, \quad \mathbf{V} = \mathbf{XW}_V \tag{64}$$

and let:

$$m = n = T \tag{65}$$

we achieved self-attention, in summary:

1. for each head i, linear transform $\mathbf{X}$ into several lower dimensional spaces via projection matrices $\mathbf{W}_i^q, \mathbf{KW}_i^k, \mathbf{VW}_i^v$ to obtain:

$$\mathbf{O}^{(i)} = \text{SDPA}(\mathbf{XW}_Q^{(i)}, \mathbf{XW}_K^{(i)}, \mathbf{XW}_V^{(i)}) \tag{66}$$

2. each iteration i correspond to one "layer" of the ["linear", "Scaled Dot-Product Attention"] on the figure 2

3. then concatenate to produce output:

   with multi-head attention, we have multiple $\mathbf{O}_i$s therefore, we need to mix them together:

$$\mathbf{Y} = \underbrace{\text{concat}(\mathbf{O}_1, \ldots, \mathbf{O}_h)}_{T \times d_{\text{model}}} \underbrace{\mathbf{W}_O}_{d_{\text{model}} \times d_{\text{model}}} \tag{67}$$

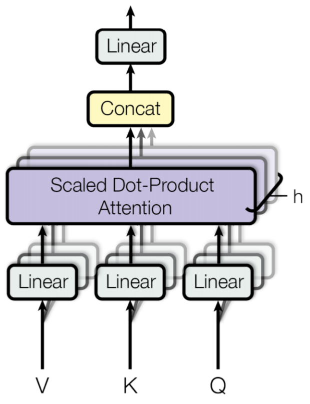


Figure 2: Multi-head Attention

8.3 Detailed implementation

8.3.1 class CausalSelfAttention(nn.Module)

```
class CausalSelfAttention(nn.Module):
```

this class defines the causal self-attention mechanism, the smallest unit of the Transformer architecture. We know that if we have h heads, then when we put multiple heads together, we have:

$$\begin{aligned}
\mathbf{Q} &= \underbrace{\begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \dots & - \\ - & \mathbf{x}_T & - \end{bmatrix}}_{T \times d_{\text{model}}} \underbrace{\begin{bmatrix} \underbrace{\mathbf{W}_Q{}^{(1)}}_{d_{\text{model}} \times d_k} & \dots & \underbrace{\mathbf{W}_Q{}^{(h)}}_{d_{\text{model}} \times d_k} \end{bmatrix}}_{d_{\text{model}} \times d_{\text{model}}} = \underbrace{\begin{bmatrix} \underbrace{\mathbf{Q}^{(1)}}_{T \times d_k} & \dots & \underbrace{\mathbf{Q}^{(h)}}_{T \times d_k} \end{bmatrix}}_{T \times d_{\text{model}}} \\
\mathbf{K} &= \underbrace{\begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \dots & - \\ - & \mathbf{x}_T & - \end{bmatrix}}_{T \times d_{\text{model}}} \underbrace{\begin{bmatrix} \underbrace{\mathbf{W}_K{}^{(1)}}_{d_{\text{model}} \times d_k} & \dots & \underbrace{\mathbf{W}_K{}^{(h)}}_{d_{\text{model}} \times d_k} \end{bmatrix}}_{d_{\text{model}} \times d_{\text{model}}} = \underbrace{\begin{bmatrix} \underbrace{\mathbf{K}^{(1)}}_{T \times d_k} & \dots & \underbrace{\mathbf{K}^{(h)}}_{T \times d_k} \end{bmatrix}}_{T \times d_{\text{model}}}
\end{aligned} \tag{68}$$

note that this also correpsonds to the B, T, C format (after adding the batch dimension B). where,

$$\begin{aligned}
h &\equiv n_{\text{heads}} \\
d_k &= \mathrm{d}_v \\
d_k \times h &\equiv \mathrm{d}_{\text{model}} \\
d_v \times h &\equiv \mathrm{d}_{\text{model}}
\end{aligned} \tag{69}$$

and the code has a few definitions:

```
def __init__(self, d_model: int, n_heads: int, dropout: float, max_seq_len: int)
    :
        super().__init__()
        assert d_model % n_heads == 0
        self.n_heads = n_heads
        self.head_dim = d_model // n_heads  # per-head dimensionality

        # Single projection that produces query, key, and value in one matmul
        self.qkv = nn.Linear(d_model, 3 * d_model)
        # Final projection after concatenating heads
        self.proj = nn.Linear(d_model, d_model)
        self.attn_dropout = nn.Dropout(dropout)
        self.resid_dropout = nn.Dropout(dropout)
```

Precompute variable mask matrix and store lower-triangular mask of shape $(1, 1, T, T)$, so it can be broadcasted over (B, h, T, T):

for example, for $T = 5$, we have:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{70}$$

```

mask = torch.tril(torch.ones(max_seq_len, max_seq_len))
mask = mask.view(1, 1, max_seq_len, max_seq_len)
```

8.3.2 Step of each Transformer Layer

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
```

1. At the conclusion of each Transformer layer, we have the output $\mathbf{Y}$ of shape (B, T, C), where it looks like (assume $B = 1$). See Eq.(82).

$$\left[\underbrace{\begin{bmatrix} - & \mathbf{y}_1 & - \\ & \vdots & \\ - & \mathbf{y}_T & - \end{bmatrix}}_{T \times d_{\text{model}}}\right] \tag{71}$$

   The column dimesion is $h \times d_v$, which is also the size of the input vector $\mathbf{x}_t$. It can be thought as the "essential" form of the qkv matrix.

   Now, we need to perform:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V \tag{72}$$

   to obtain the following single large matrix before splitting into $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ matrices, we still assume $B = 1$.

$$\left[\underbrace{\begin{bmatrix} -\mathbf{q}_1^{(1)}- & \dots & -\mathbf{q}_1^{(h)}- & -\mathbf{k}_1^{(1)}- & \dots & -\mathbf{k}_1^{(h)}- & -\mathbf{v}_1^{(1)}- & \dots & -\mathbf{v}_1^{(h)}- \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{q}_T^{(1)}- & \dots & -\mathbf{q}_T^{(h)}- & -\mathbf{k}_T^{(1)}- & \dots & -\mathbf{k}_T^{(h)}- & -\mathbf{v}_T^{(1)}- & \dots & -\mathbf{v}_T^{(h)}- \end{bmatrix}}_{T \times (d_{\text{model}} \times 3)}\right] \tag{73}$$

```
        B, T, C = x.size()  # batch, time, channels
        qkv = self.qkv(x)
```

   after running the following code, we split the (B, T, C) format into (B, T, h, d_v), i.e., the big multi-head matrix containing all $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ into separate $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ matrices:

$$\left[\underbrace{\begin{bmatrix} - & \mathbf{q}_1^{(1)} & - & \dots & - & \mathbf{q}_1^{(h)} & - \\ & \vdots & & \ddots & & \vdots & \\ - & \mathbf{q}_T^{(1)} & - & \dots & - & \mathbf{q}_T^{(h)} & - \end{bmatrix}}_{T \times d_{\text{model}}} \underbrace{\begin{bmatrix} - & \mathbf{k}_1^{(1)} & - & \dots & - & \mathbf{k}_1^{(h)} & - \\ & \vdots & & \ddots & & \vdots & \\ - & \mathbf{k}_T^{(1)} & - & \dots & - & \mathbf{k}_T^{(h)} & - \end{bmatrix}}_{T \times d_{\text{model}}} \underbrace{\begin{bmatrix} - & \mathbf{v}_1^{(1)} & - & \dots & - & \mathbf{v}_1^{(h)} & - \\ & \vdots & & \ddots & & \vdots & \\ - & \mathbf{v}_T^{(1)} & - & \dots & - & \mathbf{v}_T^{(h)} & - \end{bmatrix}}_{T \times d_{\text{model}}}\right] \tag{74}$$

```
        q, k, v = qkv.split(C, dim=2)  # each of shape (B, T, C)
```

2. now we split the **QKV** single large matrix into each separate **Q**, **K**, **V** matrices:

```
def reshape_heads(t: torch.Tensor) -> torch.Tensor:
        # Reshape to (B, n_heads, T, head_dim) for parallel head
            computation
        return t.view(B, T, self.n_heads, self.head_dim).transpose(1, 2)

# (B, n_heads, T, head_dim)
q = reshape_heads(q)
k = reshape_heads(k)
v = reshape_heads(v)
```

after running the above code, it has split up **Q**, **K** and **V**, so that each will be in the form of for simplicity let $B = 1$, and we would like to have the structure to look like:

$$\mathbf{Q} = \underbrace{\begin{bmatrix} \underbrace{\begin{bmatrix} - & \mathbf{q}_1 & - \\ - & \dots & - \\ - & \mathbf{q}_T & - \end{bmatrix}}_{\mathbf{Q}_{(1)}:\, T\times d_k} & \dots & \underbrace{\begin{bmatrix} - & \mathbf{q}_1 & - \\ - & \dots & - \\ - & \mathbf{q}_T & - \end{bmatrix}}_{\mathbf{Q}_{(h)}:\, T\times d_k} \end{bmatrix}}_{T\times d_{\text{model}}} \mathbf{K} = \underbrace{\begin{bmatrix} \underbrace{\begin{bmatrix} - & \mathbf{k}_1 & - \\ - & \dots & - \\ - & \mathbf{k}_T & - \end{bmatrix}}_{\mathbf{K}_{(1)}:\, T\times d_k} & \dots & \underbrace{\begin{bmatrix} - & \mathbf{k}_1 & - \\ - & \dots & - \\ - & \mathbf{k}_T & - \end{bmatrix}}_{\mathbf{K}_{(h)}:\, T\times d_k} \end{bmatrix}}_{T\times d_{\text{model}}}$$
$$\mathbf{V} = \underbrace{\begin{bmatrix} \underbrace{\begin{bmatrix} - & \mathbf{v}_1 & - \\ - & \dots & - \\ - & \mathbf{v}_T & - \end{bmatrix}}_{\mathbf{V}_{(1)}:\, T\times d_v} & \dots & \underbrace{\begin{bmatrix} - & \mathbf{v}_1 & - \\ - & \dots & - \\ - & \mathbf{v}_T & - \end{bmatrix}}_{\mathbf{V}_{(h)}:\, T\times d_v} \end{bmatrix}}_{T\times d_{\text{model}}} \tag{75}$$

where:

$$\begin{aligned} h &\equiv n_{\text{heads}} \\ d_k \times h &\equiv \mathrm{d}_{\text{model}} \\ d_v \times h &\equiv \mathrm{d}_{\text{model}} \end{aligned} \tag{76}$$

3. we then apply $\mathbf{QK}^\top$:

```
att = (q @ k.transpose(-2, -1)) # (B, n_heads, T, T)
```

note the above code does not perform $\mathbf{QK}^\top$, after **Q** and **K** are concatenated from all the heads, but rather it performs $\mathbf{QK}^\top$ for each head, i.e., broadcasted matrix multiplication:

$$\begin{bmatrix} \mathbf{Q}_{(1)}\mathbf{K}_{(1)}^\top & \dots & \mathbf{Q}_{(h)}\mathbf{K}_{(h)}^\top \end{bmatrix} \tag{77}$$

4. we then scale by $1/\sqrt{d_k}$:

```
att = att / (self.head_dim ** 0.5)
```

where we obtained:

$$\begin{bmatrix} \frac{\mathbf{Q}_{(1)}\mathbf{K}_{(1)}^\top}{\sqrt{d_k}} & \dots & \frac{\mathbf{Q}_{(h)}\mathbf{K}_{(h)}^\top}{\sqrt{d_k}} \end{bmatrix} = \begin{bmatrix} \underbrace{\begin{bmatrix} \frac{\mathbf{q}_1^{(1)}\mathbf{k}_1^{(1)\top}}{\sqrt{d_k}} & \dots & \frac{\mathbf{q}_1^{(1)}\mathbf{k}_T^{(1)\top}}{\sqrt{d_k}} \\ \vdots & \ddots & \vdots \\ \frac{\mathbf{q}_T^{(1)}\mathbf{k}_1^{(1)\top}}{\sqrt{d_k}} & \dots & \frac{\mathbf{q}_T^{(1)}\mathbf{k}_T^{(1)\top}}{\sqrt{d_k}} \end{bmatrix}}_{\frac{\mathbf{Q}^{(1)}\mathbf{K}^{(1)\top}}{\sqrt{d_k}}} & \dots & \underbrace{\begin{bmatrix} \frac{\mathbf{q}_1^{(h)}\mathbf{k}_1^{(h)\top}}{\sqrt{d_k}} & \dots & \frac{\mathbf{q}_1^{(h)}\mathbf{k}_T^{(h)\top}}{\sqrt{d_k}} \\ \vdots & \ddots & \vdots \\ \frac{\mathbf{q}_T^{(h)}\mathbf{k}_1^{(h)\top}}{\sqrt{d_k}} & \dots & \frac{\mathbf{q}_T^{(h)}\mathbf{k}_T^{(h)\top}}{\sqrt{d_k}} \end{bmatrix}}_{\mathbf{Q}^{(h)}\mathbf{K}^{(h)\top}} \end{bmatrix} \tag{78}$$

5. we apply a causal mask to the attention matrix, so that each word can only attend to the previous words:

```
# Apply causal mask so position t cannot attend to positions > t
att = att.masked_fill(self.mask[:, :, :T, :T] == 0, float('-inf'))
```

$$\begin{bmatrix} \begin{bmatrix} \frac{\mathbf{q}_1^{(1)}\mathbf{k}_1^{(1)\top}}{\sqrt{d_k}} & \dots & -\infty \\ \vdots & \ddots & \vdots \\ \frac{\mathbf{q}_T^{(1)}\mathbf{k}_1^{(1)\top}}{\sqrt{d_k}} & \dots & \frac{\mathbf{q}_T^{(1)}\mathbf{k}_T^{(1)\top}}{\sqrt{d_k}} \end{bmatrix} & \dots & \begin{bmatrix} \frac{\mathbf{q}_1^{(h)}\mathbf{k}_1^{(h)\top}}{\sqrt{d_k}} & \dots & -\infty \\ \vdots & \ddots & \vdots \\ \frac{\mathbf{q}_T^{(h)}\mathbf{k}_1^{(h)\top}}{\sqrt{d_k}} & \dots & \frac{\mathbf{q}_T^{(h)}\mathbf{k}_T^{(h)\top}}{\sqrt{d_k}} \end{bmatrix} \end{bmatrix} \tag{79}$$

6. apply row-wise softmax to the attention matrix:

```
att = F.softmax(att, dim=-1)
att = self.attn_dropout(att)
```

$$\begin{bmatrix} \underbrace{\begin{bmatrix} \text{softmax}\left(\begin{bmatrix} \frac{\mathbf{q}_1^{(1)}\mathbf{k}_1^{(1)\top}}{\sqrt{d_k}} & \dots & -\infty \end{bmatrix}\right) \\ \vdots \\ \text{softmax}\left(\begin{bmatrix} \frac{\mathbf{q}_T^{(1)}\mathbf{k}_1^{(1)\top}}{\sqrt{d_k}} & \dots & \frac{\mathbf{q}_T^{(1)}\mathbf{k}_T^{(1)\top}}{\sqrt{d_k}} \end{bmatrix}\right) \end{bmatrix}}_{\text{softmax}\left(\frac{\mathbf{Q}^{(1)}\mathbf{K}^{(1)\top}}{\sqrt{d_k}}\right)} & \dots & \underbrace{\begin{bmatrix} \text{softmax}\left(\begin{bmatrix} \frac{\mathbf{q}_1^{(h)}\mathbf{k}_1^{(h)\top}}{\sqrt{d_k}} & \dots & -\infty \end{bmatrix}\right) \\ \vdots \\ \text{softmax}\left(\begin{bmatrix} \frac{\mathbf{q}_T^{(h)}\mathbf{k}_1^{(h)\top}}{\sqrt{d_k}} & \dots & \frac{\mathbf{q}_T^{(h)}\mathbf{k}_T^{(h)\top}}{\sqrt{d_k}} \end{bmatrix}\right) \end{bmatrix}}_{\text{softmax}\left(\frac{\mathbf{Q}^{(h)}\mathbf{K}^{(h)\top}}{\sqrt{d_k}}\right)} \end{bmatrix} \tag{80}$$

7. apply weighted sum to the values:

```
# Weighted sum of values
y = att @ v  # (B, n_heads, T, head_dim)
```

where we obtained:

$$\underbrace{\left[\underbrace{\text{softmax}\left(\frac{\mathbf{Q}^{(1)}\mathbf{K}^{(1)}{}^{\top}}{\sqrt{d_k}}\right)\mathbf{V}^{(1)}}_{T\times d_v} \quad \dots \quad \underbrace{\text{softmax}\left(\frac{\mathbf{Q}^{(h)}\mathbf{K}^{(h)}{}^{\top}}{\sqrt{d_k}}\right)\mathbf{V}^{(h)}}_{T\times d_v}\right]}_{T\times d_{\text{model}}} \tag{81}$$

8. transform the output back to the original dimension B, T, C, where contiguous() is a method that returns a contiguous tensor.

```
# (B, n_heads, T, head_dim) -> (B, T, n_heads, head_dim)
y = y.transpose(1, 2)
y = y.contiguous()
# Reassemble heads: (B, T, C)
y = y.view(B, T, C)
```

know that:

$$\begin{aligned}\underbrace{\begin{bmatrix} - & \mathbf{y}_1 & - \\ & \vdots & \\ - & \mathbf{y}_T & - \end{bmatrix}}_{T\times d_{\text{model}}} &= \underbrace{\left[\underbrace{\begin{bmatrix} - & \mathbf{o}_1^{(1)} & - \\ - & \dots & - \\ - & \mathbf{o}_T^{(1)} & - \end{bmatrix}}_{\mathbf{O}_{(1)}:\, T\times d_v} \quad \dots \quad \underbrace{\begin{bmatrix} - & \mathbf{o}_1^{(h)} & - \\ - & \dots & - \\ - & \mathbf{o}_T^{(h)} & - \end{bmatrix}}_{\mathbf{O}_{(h)}:\, T\times d_v}\right]}_{T\times d_{\text{model}}} \underbrace{\mathbf{W}_O}_{d_{\text{model}}\times d_{\text{model}}} \\ &= \underbrace{\begin{bmatrix} - & \mathbf{o}_1^{(1)} & - & \dots & - & \mathbf{o}_1^{(h)} & - \\ & \vdots & & \ddots & & \vdots & \\ - & \mathbf{o}_T^{(1)} & - & \dots & - & \mathbf{o}_T^{(h)} & - \end{bmatrix}}_{T\times d_{\text{model}}} \underbrace{\mathbf{W}_O}_{d_{\text{model}}\times d_{\text{model}}}\end{aligned} \tag{82}$$

each row of $\mathbf{y}_i$ can be determined independently and in parallel.

9. we then perform a final linear projection on the concatenated heads:

```
# Final linear projection on the concatenated heads
y = self.resid_dropout(self.proj(y)) # resid_dropout is the residual
    dropout
return y
```

one can see that the final output is a tensor of shape B, T, d_{model}.

8.3.3 class TransformerBlock(nn.Module)

The following is the code for Pre-norm Transformer block with attention followed by an MLP, where the CausalSelfAttention() is from the previous section:

```
class TransformerBlock(nn.Module):
    def __init__(self, d_model: int, n_heads: int, d_ff: int, dropout: float,
        max_seq_len: int):
        super().__init__()
        # Pre-layernorm improves optimization stability in autoregressive LMs
        self.ln1 = nn.LayerNorm(d_model)
        self.attn = CausalSelfAttention(d_model, n_heads, dropout, max_seq_len)
        self.ln2 = nn.LayerNorm(d_model)
        # Feed-forward network (position-wise MLP)
        self.mlp = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model),
            nn.Dropout(dropout),
        )
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # Attention sub-layer with residual connection
        x = x + self.attn(self.ln1(x))
        # MLP sub-layer with residual connection
        x = x + self.mlp(self.ln2(x))
        return x
```

$x = x + f(x)$ is the residual connection. and nn.GELU() is the activation function.

8.3.4 class TransformerLM(nn.Module)

```
class TransformerLM(nn.Module):
    """Causal language model built from Transformer blocks."""

    def __init__(self, config: ModelConfig):
        super().__init__()
        self.config = config
        # Token and positional embeddings are summed to form the input stream
        self.token_emb = nn.Embedding(config.vocab_size, config.d_model)
        self.pos_emb = nn.Embedding(config.max_seq_len, config.d_model)
        self.drop = nn.Dropout(config.dropout)
        # Stack of identical Transformer blocks
        self.blocks = nn.ModuleList([
            TransformerBlock(config.d_model, config.n_heads, config.d_ff, config
                .dropout, config.max_seq_len)
            for _ in range(config.n_layers)
        ])
        # Final layernorm before projecting to vocabulary logits
        self.ln_f = nn.LayerNorm(config.d_model)
        # LM head ties to embedding size but does not share weights here
        self.lm_head = nn.Linear(config.d_model, config.vocab_size, bias=False)

    def forward(self, idx: torch.Tensor, targets: Optional[torch.Tensor] = None)
        :
```

```
        """Forward pass producing next-token logits and optional cross-entropy
            loss.

        Parameters
        ----------
        idx : torch.Tensor
            Token indices of shape (B, T).
        targets : Optional[torch.Tensor]
            Target indices of shape (B, T). If provided, a token-level
            cross-entropy loss is computed and returned.

        Returns
        -------
        Tuple[torch.Tensor, Optional[torch.Tensor]]
            - logits: (B, T, vocab_size)
            - loss: scalar loss if targets is provided, else None
        """
        B, T = idx.shape
        # Ensure sequence length fits within the configured context window
        assert T <= self.config.max_seq_len

        # Positions 0..T-1 for current sequence length
        pos = torch.arange(0, T, dtype=torch.long, device=idx.device).unsqueeze
            (0)
```

this is to create a tensor of shape $(1, T)$ with values from 0 to T-1. We call the unsqueeze(0) to add a batch dimension, so that it can be broadcasted over the batch dimension when adding to the token embeddings.

$$\begin{bmatrix}\begin{bmatrix}0 & 1 & 2 & \dots & T-1\end{bmatrix}\end{bmatrix} \tag{83}$$

```

        # Embed tokens and positions, then apply dropout
        x = self.token_emb(idx) + self.pos_emb(pos)
        x = self.drop(x)

        # Pass through the stack of Transformer blocks
        for blk in self.blocks:
            x = blk(x)

        # Final normalization and projection to vocabulary
        x = self.ln_f(x)
        logits = self.lm_head(x)

        # Compute token-level cross-entropy if targets are given
        loss = None
        if targets is not None:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.
                view(-1))
        return logits, loss
```

```
    @torch.no_grad()
    def generate(self, idx: torch.Tensor, max_new_tokens: int) -> torch.Tensor:
        """Greedy-free sampling generation from the language model.

        Iteratively samples `max_new_tokens` next tokens by:
        - conditioning on the last `max_seq_len` tokens
        - computing logits for the next position
        - sampling from the softmax distribution
        - appending the sampled token to the context

        Parameters
        ----------
        idx : torch.Tensor
            Initial context token indices of shape (B, T0).
        max_new_tokens : int
            Number of new tokens to sample and append.

        Returns
        -------
        torch.Tensor
            Extended token indices of shape (B, T0 + max_new_tokens).
        """
        self.eval()
        for _ in range(max_new_tokens):
            # Restrict conditioning length to model's maximum context window
            # only the last self.config.max_seq_len tokens are used to condition
                the generation, as they are the most recent tokens
            idx_cond = idx[:, -self.config.max_seq_len:]
            logits, _ = self(idx_cond)
            # Select logits for the last time step (the next token prediction)
            logits = logits[:, -1, :]
            probs = F.softmax(logits, dim=-1)
            # Multinomial sampling allows for non-greedy, stochastic generation
            next_id = torch.multinomial(probs, num_samples=1)
            # Append sampled token and continue
            idx = torch.cat((idx, next_id), dim=1)
        return idx
```

8.4 Cross-Attention

If we were to perform cross-attention, **K** and **V** must be the same thing. For example,

1. Text-to-image Cross-Attention:

$$\begin{aligned} \mathbf{K} \text{ and } \mathbf{V} \text{ are text embedding} \\ \mathbf{Q} \text{ is image embedding} \end{aligned} \tag{84}$$

2. Image-to-text Cross-Attention:

$$\begin{aligned}\mathbf{K} \text{ and } \mathbf{V} \text{ are image embedding}\\ \mathbf{Q} \text{ is text embedding}\end{aligned} \tag{85}$$

8.5 $\mathbf{K}-\mathbf{V}$ caching

looking at $\mathbf{QK}^\top$, imagine in a self-attention scenario, where the current senetence length is T, and $m = n = T$, then $\mathbf{QK}^\top$ is a square matrix of $T \times T$, where we have:

$$\mathbf{QK}^\top = \begin{bmatrix} \mathbf{q}_1\mathbf{k}_1^\top & \dots & \mathbf{q}_1\mathbf{k}_T^\top \\ \vdots & \ddots & \vdots \\ \mathbf{q}_T\mathbf{k}_1^\top & \dots & \mathbf{q}_T\mathbf{k}_T^\top \end{bmatrix} \tag{86}$$

let's say we have a new word w_{T+1} to be added to the sentence, then we have:

$$\begin{bmatrix} \mathbf{q}_1\mathbf{k}_1^\top & \dots & \mathbf{q}_1\mathbf{k}_T^\top & \mathbf{q}_1\mathbf{k}_{T+1}^\top \\ \vdots & \ddots & \vdots & \\ \mathbf{q}_T\mathbf{k}_1^\top & \dots & \mathbf{q}_T\mathbf{k}_T^\top & \mathbf{q}_T\mathbf{k}_{T+1}^\top \\ \mathbf{q}_{T+1}\mathbf{k}_1^\top & \dots & \mathbf{q}_{T+1}\mathbf{k}_T^\top & \mathbf{q}_{T+1}\mathbf{k}_{T+1}^\top \end{bmatrix} = \begin{bmatrix} \mathbf{QK}^\top & \mathbf{Qk}_{T+1}^\top \\ \mathbf{q}_{T+1}\mathbf{K}^\top & \mathbf{q}_{T+1}\mathbf{k}_{T+1}^\top \end{bmatrix} \tag{87}$$

From here, one sees that in generic matrix multiplication, one needs to cache both $\mathbf{K}$ and $\mathbf{Q}$. However, the matrix masks out the upper triangular part of the matrix, i.e., it looks like this:

$$\begin{bmatrix} \mathbf{q}_1\mathbf{k}_1^\top & - & - & - \\ \vdots & \ddots & \vdots & \vdots \\ \mathbf{q}_T\mathbf{k}_1^\top & \dots & \mathbf{q}_T\mathbf{k}_T^\top & - \\ \mathbf{q}_{T+1}\mathbf{k}_1^\top & \dots & \mathbf{q}_{T+1}\mathbf{k}_T^\top & \mathbf{q}_{T+1}\mathbf{k}_{T+1}^\top \end{bmatrix} = \begin{bmatrix} \underbrace{\mathbf{QK}^\top}_{\text{lower triangular}} & - \\ \mathbf{q}_{T+1}\mathbf{K}^\top & \mathbf{q}_{T+1}\mathbf{k}_{T+1}^\top \end{bmatrix} \tag{88}$$

One can see the newly added row (no new column is added) in the above matrix, requires $\mathbf{q}_{T+1}$, $\mathbf{k}_{T+1}$ and $\mathbf{K}$. This means that $\mathbf{K}$ needs to be cached.

Again looking at Eq.(62):

$$\begin{bmatrix} \frac{\exp(\mathbf{q}_1\mathbf{k}_1^\top)}{\exp(\mathbf{q}_1\mathbf{k}_1^\top)}\mathbf{v}_1 \\ \frac{\exp(\mathbf{q}_2\mathbf{k}_1^\top)}{\exp(\mathbf{q}_2\mathbf{k}_1^\top)+\exp(\mathbf{q}_2\mathbf{k}_2^\top)}\mathbf{v}_1 + \frac{\exp(\mathbf{q}_2\mathbf{k}_2^\top)}{\exp(\mathbf{q}_2\mathbf{k}_1^\top)+\exp(\mathbf{q}_2\mathbf{k}_2^\top)}\mathbf{v}_2 \\ \frac{\exp(\mathbf{q}_3\mathbf{k}_1^\top)}{\exp(\mathbf{q}_3\mathbf{k}_1^\top)+\exp(\mathbf{q}_3\mathbf{k}_2^\top)+\exp(\mathbf{q}_3\mathbf{k}_3^\top)}\mathbf{v}_1 + \frac{\exp(\mathbf{q}_3\mathbf{k}_2^\top)}{\exp(\mathbf{q}_3\mathbf{k}_1^\top)+\exp(\mathbf{q}_3\mathbf{k}_2^\top)+\exp(\mathbf{q}_3\mathbf{k}_3^\top)}\mathbf{v}_2 + \frac{\exp(\mathbf{q}_3\mathbf{k}_3^\top)}{\exp(\mathbf{q}_3\mathbf{k}_1^\top)+\exp(\mathbf{q}_3\mathbf{k}_2^\top)+\exp(\mathbf{q}_3\mathbf{k}_3^\top)}\mathbf{v}_3 \\ \vdots \\ \sum_{i=1}^{T} \frac{\exp(\mathbf{q}_T\mathbf{k}_i^\top)}{\sum_{j=1}^{T}\exp(\mathbf{q}_T\mathbf{k}_j^\top)}\mathbf{v}_i \\ \sum_{i=1}^{T+1} \frac{\exp(\mathbf{q}_{T+1}\mathbf{k}_i^\top)}{\sum_{j=1}^{T+1}\exp(\mathbf{q}_{T+1}\mathbf{k}_j^\top)}\mathbf{v}_i \end{bmatrix} \tag{89}$$

the last row in the above matrix contains the newly added row, where all $\mathbf{v}_i$ are needed for computation and hence they need to be cached.

8.5.1 Multi-head latent attention

say we have sentence of length T, and we have each word represented by a $d = 7168$ dimensional vector. Then we can have:

$$\begin{aligned}\mathbf{L}_{K,V} &= \mathbf{X}\mathbf{W}_{\downarrow K,V} \\ &= \underbrace{\begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \dots & - \\ - & \mathbf{x}_T & - \end{bmatrix}}_{T\times d_{\text{model}}} \underbrace{\begin{bmatrix} \mathbf{w}_{1,1} & \cdots & \mathbf{w}_{1,576} \\ \vdots & \ddots & \vdots \\ \mathbf{w}_{7168,1} & \cdots & \mathbf{w}_{7168,576} \end{bmatrix}}_{d_{\text{model}}\times 576}\end{aligned} \tag{90}$$

where $\mathbf{L}_{K,V}$ is the "common" part of the latent representation of the sentence for both $\mathbf{K}$ and $\mathbf{V}$.

normally, in the case of self-attention, one should have, $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$, $\mathbf{K} = \mathbf{X}\mathbf{W}_K$, $\mathbf{V} = \mathbf{X}\mathbf{W}_V$, but in here, we are having:

$$\begin{array}{lcl} \text{traditional} & & \text{multi-head latent attention} \\ \mathbf{Q} = \mathbf{X}\mathbf{W}_Q & \rightarrow & \underbrace{\mathbf{Q}}_{T\times 128} = \mathbf{X} \underbrace{\mathbf{W}_Q}_{7168\times 128} \\ \mathbf{K} = \mathbf{X}\mathbf{W}_K & \rightarrow & \underbrace{\mathbf{K}}_{T\times 128} = \mathbf{X} \underbrace{\mathbf{W}_{\downarrow K,V}}_{d_{\text{model}}\times 576} \underbrace{\mathbf{W}_{\uparrow K}}_{576\times 128} = \underbrace{\mathbf{L}_{K,V}}_{T\times 576} \underbrace{\mathbf{W}_{\uparrow K}}_{576\times 128} \\ \mathbf{V} = \mathbf{X}\mathbf{W}_V & \rightarrow & \underbrace{\mathbf{V}}_{T\times 128} = \mathbf{X} \underbrace{\mathbf{W}_{\downarrow K,V}}_{d_{\text{model}}\times 576} \underbrace{\mathbf{W}_{\uparrow V}}_{576\times 128} = \underbrace{\mathbf{L}_{K,V}}_{T\times 576} \underbrace{\mathbf{W}_{\uparrow V}}_{576\times 128} \end{array} \tag{91}$$

making $\mathbf{W}_{\downarrow K,V}$ to be common for all heads for both $\mathbf{K}$ and $\mathbf{V}$. Therefore only a smaller $\mathbf{W}_{\uparrow K}$ and $\mathbf{W}_{\uparrow V}$ need to be trained per head for both $\mathbf{K}$ and $\mathbf{V}$. This means that:

$$\begin{aligned} \mathbf{W}_K &\longrightarrow \mathbf{W}_{\downarrow K,V}\ \mathbf{W}_{\uparrow K} \\ \mathbf{K} = \mathbf{W}_K &\longrightarrow \mathbf{X}\mathbf{W}_{\downarrow K,V}\ \mathbf{W}_{\uparrow K} \end{aligned} \tag{92}$$

substitute:

$$\begin{aligned} \mathbf{Q}\mathbf{K}^\top &= \mathbf{X}\mathbf{W}_Q(\mathbf{X}\ \mathbf{W}_{\downarrow K,V}\ \mathbf{W}_{\uparrow K})^\top \\ &= \mathbf{X}\mathbf{W}_Q\big((\mathbf{X}\ \mathbf{W}_{\downarrow K,V})\ \mathbf{W}_{\uparrow K})\big)^\top \\ &= \underbrace{\mathbf{X}}_{T\times d_{\text{model}}} \underbrace{(\mathbf{W}_Q\mathbf{W}_{\uparrow K}^\top)}_{d_{\text{model}}\times 576} \underbrace{(\mathbf{X}\ \mathbf{W}_{\downarrow K,V})^\top}_{576\times T} \end{aligned} \tag{93}$$

$$
\begin{aligned}
\mathbf{Q}\mathbf{K}^\top &= \mathbf{X}\mathbf{W}_Q(\mathbf{X}\ \mathbf{W}_{\downarrow K,V}\ \mathbf{W}_{\uparrow K})^\top \\
&= \mathbf{X}\mathbf{W}_Q\big((\mathbf{X}\ \mathbf{W}_{\downarrow K,V})\ \mathbf{W}_{\uparrow K})\big)^\top \\
&= \underbrace{\mathbf{X}}_{T\times d_{\text{model}}}\ \underbrace{(\mathbf{W}_Q\mathbf{W}_{\uparrow K}^\top)}_{d_{\text{model}}\times 576}\underbrace{(\mathbf{X}\ \mathbf{W}_{\downarrow K,V})^\top}_{576\times T}
\end{aligned}
\tag{94}
$$

- Remember that although we have different $\mathbf{W}_{\uparrow K}$ for each head, $(\mathbf{W}_Q\mathbf{W}_{\uparrow K}^\top)$ can be computed first only once, before they multiple with $\mathbf{X}$ (and $(\mathbf{X}\ \mathbf{W}_{\downarrow K,V})$) during inference time for each variable length $\mathbf{X}$. This means that we do not need to store $\mathbf{W}_Q$ and $\mathbf{W}_{\uparrow K}$ separately, but rather we can store $\mathbf{W}_Q\mathbf{W}_{\uparrow K}^\top$ once. It's like having a new $\mathbf{W}_Q$

In terms of caching, assumed that at the lenth of T words, we have cached $\mathbf{X}(\mathbf{W}_Q\mathbf{W}_{\uparrow K}^\top)$ and $\mathbf{X}\mathbf{W}_{\downarrow K,V}$. Then after adding a new word, i.e., $\mathbf{X}_{T+1}$, we can just compute one more row of the matrix $\mathbf{X}_{T+1}\mathbf{W}_Q\mathbf{W}_{\uparrow K}$ and $\mathbf{X}_{T+1}\mathbf{W}_{\downarrow K,V}$ and append it to the cache, i.e.,

$$
\begin{aligned}
&\begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \dots & - \\ - & \mathbf{x}_T & - \end{bmatrix}\big[(\mathbf{W}_Q\mathbf{W}_{\uparrow K}^\top)\big] = \text{cache} \\
\Longrightarrow &\begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \dots & - \\ - & \mathbf{x}_T & - \\ - & \mathbf{x}_{T+1} & - \end{bmatrix}\big[(\mathbf{W}_Q\mathbf{W}_{\uparrow K}^\top)\big] = \begin{bmatrix} \text{cache} \\ \mathbf{x}_{T+1}(\mathbf{W}_Q\mathbf{W}_{\uparrow K}^\top) \end{bmatrix}
\end{aligned}
\tag{95}
$$

the same applies to $\mathbf{X}\mathbf{W}_{\downarrow K,V}$, i.e.,

$$
\begin{aligned}
&\begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \dots & - \\ - & \mathbf{x}_T & - \end{bmatrix}\big[\mathbf{W}_{\downarrow K,V}\big] = \text{cache} \\
\Longrightarrow &\begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \dots & - \\ - & \mathbf{x}_T & - \\ - & \mathbf{x}_{T+1} & - \end{bmatrix}\big[\mathbf{W}_{\downarrow K,V}\big] = \begin{bmatrix} \text{cache} \\ \mathbf{x}_{T+1}\mathbf{W}_{\downarrow K,V} \end{bmatrix}
\end{aligned}
\tag{96}
$$

Now, let's look at the output of the multi-head attention:

$$
\begin{aligned}
\underbrace{\mathbf{Y} = \begin{bmatrix} - & \mathbf{y}_1 & - \\ & \vdots & \\ - & \mathbf{y}_T & - \end{bmatrix}}_{T\times d_{\text{model}}} &= \underbrace{\text{concat}(\mathbf{O}_1, \dots, \mathbf{O}_h)}_{T\times(d_v\times h)} \underbrace{\mathbf{W}_O}_{d_{\text{model}}\times d_{\text{model}}} \\
&= \left[\text{softmax}\left(\frac{\mathbf{Q}_{(1)}\mathbf{K}_{(1)}{}^\top}{\sqrt{d_k}}\right)\mathbf{V}_{(1)} \quad \dots \quad \text{softmax}\left(\frac{\mathbf{Q}_{(h)}\mathbf{K}_{(h)}{}^\top}{\sqrt{d_k}}\right)\mathbf{V}_{(h)}\right]\mathbf{W}_O \\
&= \left[\mathbf{O}_{(1)}\mathbf{V}_{(1)} \quad \dots \quad \mathbf{O}_{(h)}\mathbf{V}_{(h)}\right]\mathbf{W}_O \\
&= \begin{bmatrix} \underbrace{\mathbf{O}_{(1)}}_{T\times d_v} & \dots & \underbrace{\mathbf{O}_{(h)}}_{T\times d_v} \end{bmatrix} \begin{bmatrix} \underbrace{\mathbf{V}_{(1)}}_{d_v\times d_v} & \dots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{0} & \dots & \underbrace{\mathbf{V}_{(h)}}_{d_v\times d_v} \end{bmatrix} \begin{bmatrix} \underbrace{\mathbf{W}_O^{1:d_v,1:d_{\text{model}}}}_{d_v\times d_{\text{model}}} \\ \vdots \\ \underbrace{\mathbf{W}_O^{d_v\times(h-1)+1:d_{\text{model}},1:d_{\text{model}}}}_{d_v\times d_{\text{model}}} \end{bmatrix} \\
&= \underbrace{\left[\mathbf{O}_{(1)} \quad \dots \quad \mathbf{O}_{(h)}\right]}_{T\times d_{\text{model}}} \underbrace{\begin{bmatrix} \underbrace{\mathbf{V}_{(1)}\mathbf{W}_O^{1:d_v,1:d_{\text{model}}}}_{d_v\times d_{\text{model}}} \\ \vdots \\ \underbrace{\mathbf{V}_{(h)}\mathbf{W}_O^{d_v\times(h-1)+1:d_{\text{model}},1:d_{\text{model}}}}_{d_v\times d_{\text{model}}} \end{bmatrix}}_{d_{\text{model}}\times d_{\text{model}}}
\end{aligned}
\tag{97}
$$

now that we replaced $\mathbf{W}_V$ with $\mathbf{W}_{\downarrow K,V}\mathbf{W}_{\uparrow V}$, we can have:

$$
\mathbf{V} = \underbrace{\mathbf{X}}_{T\times d_{\text{model}}} \underbrace{\mathbf{W}_{\downarrow K,V}}_{d_{\text{model}}\times 576} \underbrace{\mathbf{W}_{\uparrow V}}_{576\times d_v} \tag{98}
$$

therefore, we can have each $\mathbf{V}$, for example $\mathbf{V}_{(1)}$ as

$$
\mathbf{V}_{(1)}\mathbf{W}_O^{1:d_v,1:d_{\text{model}}} = \underbrace{\mathbf{X}}_{T\times d_{\text{model}}} \underbrace{\mathbf{W}_{\downarrow K,V}}_{d_{\text{model}}\times 576} \underbrace{\mathbf{W}_{\uparrow V}}_{576\times d_v} \underbrace{\mathbf{W}_O^{1:d_v,1:d_{\text{model}}}}_{d_v\times d_{\text{model}}} \tag{99}
$$

we already computed $\mathbf{X}\mathbf{W}_{\downarrow K,V}$ as $\mathbf{L}_{K,V}$, and we can also pre-compute $\mathbf{W}_{\uparrow V}\mathbf{W}_O^{1:d_v,1:d_{\text{model}}}$ as single matrix, instead of storing $\mathbf{W}_{\uparrow V}$ and $\mathbf{W}_O^{1:d_v,1:d_{\text{model}}}$ separately.

Finally, we have $\mathbf{Y}$ as the output of the multi-head attention layer.

8.6 RoPE

Here is the RoPE formula:

```
class RotaryEmbedding(nn.Module):
    def __init__(self, head_dim=64, rope_theta=100000):
        super().__init__()
        self.head_dim = head_dim
        self.rope_theta = rope_theta

    def forward(self, x):
        # x shape: [batch, num_heads, seq_len, head_dim]
```

```
        # Compute frequencies
        i = torch.arange(0, self.head_dim, 2)
        freqs = 1.0 / (self.rope_theta ** (i / self.head_dim))
```

$$\begin{aligned} i &= \begin{bmatrix} 0 & 2 & 4 & \dots & d_v - 2 \end{bmatrix} \\ \text{freqs} &= \underbrace{\begin{bmatrix} \frac{1}{\theta^{0/d_v}} & \frac{1}{\theta^{2/d_v}} & \frac{1}{\theta^{4/d_v}} & \dots & \frac{1}{\theta^{(d_v-2)/d_v}} \end{bmatrix}}_{1 \times (d_v/2)} \end{aligned} \tag{100}$$

to avoid notational clutter, we write θ^{t/d_v} as θ^t for short.

$$\text{freqs} = \underbrace{\begin{bmatrix} \frac{1}{\theta^{0}} & \frac{1}{\theta^{2}} & \frac{1}{\theta^{4}} & \dots & \frac{1}{\theta^{d_v-2}} \end{bmatrix}}_{1 \times (d_v/2)} \tag{101}$$

this is in the decreasing frequency order.

```
        # Compute angles for all positions
        pos = torch.arange(x.shape[2])
        angles = torch.outer(pos, freqs)
```

since x shape is (batch, num_heads, T, head_dim), therefore x.shape[2] is the seq_len, i.e., T. Therefore, we have:

$$\begin{aligned} \text{angles} &= \begin{bmatrix} 0 \times \frac{1}{\theta^0} & 0 \times \frac{1}{\theta^2} & 0 \times \frac{1}{\theta^4} & \dots & 0 \times \frac{1}{\theta^{d_v-2}} \\ 1 \times \frac{1}{\theta^0} & 1 \times \frac{1}{\theta^2} & 1 \times \frac{1}{\theta^4} & \dots & 1 \times \frac{1}{\theta^{d_v-2}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (T-1) \times \frac{1}{\theta^0} & (T-1) \times \frac{1}{\theta^2} & (T-1) \times \frac{1}{\theta^4} & \dots & (T-1) \times \frac{1}{\theta^{d_v-2}} \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} \frac{0}{\theta^0} & \frac{0}{\theta^2} & \frac{0}{\theta^4} & \dots & \frac{0}{\theta^{d_v-2}} \\ \frac{1}{\theta^0} & \frac{1}{\theta^2} & \frac{1}{\theta^4} & \dots & \frac{1}{\theta^{d_v-2}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{T-1}{\theta^0} & \frac{T-1}{\theta^2} & \frac{T-1}{\theta^4} & \dots & \frac{T-1}{\theta^{d_v-2}} \end{bmatrix}}_{T \times (d_v/2)} \end{aligned} \tag{102}$$

for each t^{th} token, there is a corresponding frequency for each feature, i.e., $\frac{t}{\theta^0}, \frac{t}{\theta^2}, \frac{t}{\theta^4}, \dots, \frac{t}{\theta^{d_v-2}}$

```
        # Precompute sin/cos
        cos = torch.cos(angles)
        sin = torch.sin(angles)
        cos = cos.unsqueeze(0).unsqueeze(0)
        sin = sin.unsqueeze(0).unsqueeze(0)
```

$$\cos = \left[\left[\begin{bmatrix} \cos(\frac{0}{\theta^0}) & \cos(\frac{0}{\theta^2}) & \cos(\frac{0}{\theta^4}) & \dots & \cos(\frac{0}{\theta^{d_v-2}}) \\ \cos(\frac{1}{\theta^0}) & \cos(\frac{1}{\theta^2}) & \cos(\frac{1}{\theta^4}) & \dots & \cos(\frac{1}{\theta^{d_v-2}}) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \cos(\frac{T-1}{\theta^0}) & \cos(\frac{T-1}{\theta^2}) & \cos(\frac{T-1}{\theta^4}) & \dots & \cos(\frac{T-1}{\theta^{d_v-2}}) \end{bmatrix}\right]\right] \tag{103}$$

$$\sin = \left[\left[\begin{bmatrix} \sin(\frac{0}{\theta^0}) & \sin(\frac{0}{\theta^2}) & \sin(\frac{0}{\theta^4}) & \dots & \sin(\frac{0}{\theta^{d_v-2}}) \\ \sin(\frac{1}{\theta^0}) & \sin(\frac{1}{\theta^2}) & \sin(\frac{1}{\theta^4}) & \dots & \sin(\frac{1}{\theta^{d_v-2}}) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sin(\frac{T-1}{\theta^0}) & \sin(\frac{T-1}{\theta^2}) & \sin(\frac{T-1}{\theta^4}) & \dots & \sin(\frac{T-1}{\theta^{d_v-2}}) \end{bmatrix}\right]\right] \tag{104}$$

```
        # Split into pairs
        x_even = x[..., 0::2]
        x_odd = x[..., 1::2]
```

for example, if x is $\mathbf{Q}$, then we have:

$$\mathbf{Q}_{even} = \begin{bmatrix} q_{0,0} & q_{0,2} & q_{0,4} & \cdots & q_{0,d_v-2} \\ q_{1,0} & q_{1,2} & q_{1,4} & \cdots & q_{1,d_v-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ q_{T-1,0} & q_{T-1,2} & q_{T-1,4} & \cdots & q_{T-1,d_v-2} \end{bmatrix} \mathbf{Q}_{odd} = \begin{bmatrix} q_{0,1} & q_{0,3} & q_{0,5} & \cdots & q_{0,d_v-1} \\ q_{1,1} & q_{1,3} & q_{1,5} & \cdots & q_{1,d_v-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ q_{T-1,1} & q_{T-1,3} & q_{T-1,5} & \cdots & q_{T-1,d_v-1} \end{bmatrix} \tag{105}$$

```
        # Apply rotation (broadcasting cos/sin across batch and heads)
        x_even_new = x_even * cos - x_odd * sin
        x_odd_new = x_even * sin + x_odd * cos
```

$$\mathbf{Q}'_{even} = \underbrace{\begin{bmatrix} q_{0,0}\cos(\frac{0}{\theta^0}) - q_{0,1}\sin(\frac{0}{\theta^0}) & q_{0,2}\cos(\frac{0}{\theta^2}) - q_{0,3}\sin(\frac{0}{\theta^2}) & \dots & q_{0,d_v-2}\cos(\frac{0}{\theta^{d_v-2}}) - q_{0,d_v-1}\sin(\frac{0}{\theta^{d_v-2}}) \\ q_{1,0}\cos(\frac{1}{\theta^0}) - q_{1,1}\sin(\frac{1}{\theta^0}) & q_{1,2}\cos(\frac{1}{\theta^2}) - q_{1,3}\sin(\frac{1}{\theta^2}) & \dots & q_{1,d_v-2}\cos(\frac{1}{\theta^{d_v-2}}) - q_{1,d_v-1}\sin(\frac{1}{\theta^{d_v-2}}) \\ \vdots & \vdots & \vdots & \ddots \\ q_{T-1,0}\cos(\frac{T-1}{\theta^0}) - q_{T-1,1}\sin(\frac{T-1}{\theta^0}) & q_{T-1,2}\cos(\frac{T-1}{\theta^2}) - q_{T-1,3}\sin(\frac{T-1}{\theta^2}) & \dots & q_{T-1,d_v-2}\cos(\frac{T-1}{\theta^{d_v-2}}) - q_{T-1,d_v-1}\sin(\frac{T-1}{\theta^{d_v-}} \\ q_{T,0}\cos(\frac{T}{\theta^0}) - q_{T,1}\sin(\frac{T}{\theta^0}) & q_{T,2}\cos(\frac{T}{\theta^2}) - q_{T,3}\sin(\frac{T}{\theta^2}) & \dots & q_{T,d_v-2}\cos(\frac{T}{\theta^{d_v-2}}) - q_{T,d_v-1}\sin(\frac{T}{\theta^{d_v-2}}) \end{bmatrix}}_{T\times(d_v/2)} \tag{106}$$

$$
\mathbf{Q}'_{odd} =
$$

$$
\underbrace{\left[\begin{array}{cccc}
q_{0,0}\sin(\frac{0}{\theta^0}) + q_{0,1}\cos(\frac{0}{\theta^0}) & q_{0,2}\sin(\frac{0}{\theta^2}) + q_{0,3}\cos(\frac{0}{\theta^2}) & \dots & q_{0,d_v-2}\sin(\frac{0}{\theta^{d_v-2}}) + q_{0,d_v-1}\cos(\frac{0}{\theta^{d_v-2}}) \\
q_{1,0}\sin(\frac{1}{\theta^0}) + q_{1,1}\cos(\frac{1}{\theta^0}) & q_{1,2}\sin(\frac{1}{\theta^2}) + q_{1,3}\cos(\frac{1}{\theta^2}) & \dots & q_{1,d_v-2}\sin(\frac{1}{\theta^{d_v-2}}) + q_{1,d_v-1}\cos(\frac{1}{\theta^{d_v-2}}) \\
\vdots & \vdots & \vdots & \ddots \\
q_{T-1,0}\sin(\frac{T-1}{\theta^0}) + q_{T-1,1}\cos(\frac{T-1}{\theta^0}) & q_{T-1,2}\sin(\frac{T-1}{\theta^2}) + q_{T-1,3}\cos(\frac{T-1}{\theta^2}) & \dots & q_{T-1,d_v-2}\sin(\frac{T-1}{\theta^{d_v-2}}) + q_{T-1,d_v-1}\cos(\frac{T-1}{\theta^{d_v-}} \\
q_{T,0}\sin(\frac{T}{\theta^0}) + q_{T,1}\cos(\frac{T}{\theta^0}) & q_{T,2}\sin(\frac{T}{\theta^2}) + q_{T,3}\cos(\frac{T}{\theta^2}) & \dots & q_{T,d_v-2}\sin(\frac{T}{\theta^{d_v-2}}) + q_{T,d_v-1}\cos(\frac{T}{\theta^{d_v-2}})
\end{array}\right.}_{T\times(d_v/2)}
\tag{107}
$$

One can see that the addition of the corresponding even and odd elements will be a $2-d$ rotation by each corresponding θ with different frequency. For example, the addtion of the two blue terms will be a $2-d$ rotation by θ^2 with frequency 1, i.e., by $\frac{1}{\theta^2}$ radians, and the addtion of the two red terms will be a $2-d$ rotation by $\frac{1}{\theta^2}$ radians.

$$
\begin{bmatrix} q'_{1,2} \\ q'_{1,3} \end{bmatrix} = \begin{bmatrix} \cos(\frac{1}{\theta^2}) & -\sin(\frac{1}{\theta^2}) \\ \sin(\frac{1}{\theta^2}) & \cos(\frac{1}{\theta^2}) \end{bmatrix} \begin{bmatrix} q_{1,2} \\ q_{1,3} \end{bmatrix} = \begin{bmatrix} q_{1,2}\cos(\frac{1}{\theta^2}) - q_{1,3}\sin(\frac{1}{\theta^2}) \\ q_{1,2}\sin(\frac{1}{\theta^2}) + q_{1,3}\cos(\frac{1}{\theta^2}) \end{bmatrix} \tag{108}
$$

or more generally, for each token t, and each feature block $(2i, 2i+1)$ (i.e., the i^{th} feature block), we have:

$$
\begin{aligned}
\begin{bmatrix} q'_{t,2i} & q'_{t,2i+1} \end{bmatrix} &= \begin{bmatrix} q_{t,2i}\cos(\frac{t}{\theta^2}) - q_{t,2i+1}\sin(\frac{t}{\theta^2}) & q_{t,2i}\sin(\frac{t}{\theta^2}) + q_{t,2i+1}\cos(\frac{t}{\theta^2}) \end{bmatrix} \\
&= \begin{bmatrix} q_{t,2i} & q_{t,2i+1} \end{bmatrix} R(\frac{t}{\theta^2})
\end{aligned} \tag{109}
$$

```
        # Interleave back
        x_rotated = torch.stack([x_even_new, x_odd_new], dim=-1)
        x_rotated = x_rotated.flatten(-2, -1)
        return x_rotated
```

torch.stack([], dim=-1) will stack the $\mathbf{Q}'_{even}$, of size $(T, (d_v/2))$, and $\mathbf{Q}'_{odd}$, of size $(T, (d_v/2))$, along the last dimension (hence creating a new dimension of size 2) so that the result is a tensor of shape $(T, (d_v/2), 2)$, and then the flatten(-2, -1) will flatten the last two dimensions, so that the result is a tensor of shape (T, d_v), which is the same as the original $\mathbf{Q}$ tensor.

8.7 Decoupled RoPE

we have, given h heads:

$$
\begin{aligned}
\mathbf{Y}^{(i)} &= \text{softmax}\left(\frac{1}{\sqrt{d_k}}\underbrace{\mathbf{X}}_{T\times d_{\text{model}}}\underbrace{\mathbf{W}_Q^{(i)}}_{d_{\text{model}}\times 128}\underbrace{{\mathbf{W}^{(i)}}_{\uparrow K}^{\top}}_{128\times 576}\underbrace{(\mathbf{X}\,\mathbf{W}_{\downarrow K,V})^{\top}}_{576\times T}\right)\underbrace{\mathbf{X}\,\mathbf{W}_{\downarrow K,V}}_{T\times 576}\underbrace{\mathbf{W}^{(i)}{}_{\uparrow V}}_{576\times d_v}\underbrace{\mathbf{W}_O^{(i-1)\times d_v+1:i\times d_v,1:d_{\text{model}}}}_{d_v\times d_{\text{model}}}\\
\mathbf{Y} &= \sum_{i=1}^{h}\mathbf{Y}^{(i)}
\end{aligned}
\tag{110}
$$

when we have multiple heads, the output is the same size, except there are multiple of $\mathbf{Y}^{(i)}$ added together.

Here is something about RoPE, let's take the terms inside the softmax function, and remove the $\frac{1}{\sqrt{d_k}}$ term, we have:

$$
\underbrace{\mathbf{X}}_{T\times d_{\text{model}}}\underbrace{\mathbf{W}_Q^{(i)}}_{d_{\text{model}}\times 128}\underbrace{{\mathbf{W}^{(i)}}_{\uparrow K}^{\top}}_{128\times 576}\underbrace{(\mathbf{X}\,\mathbf{W}_{\downarrow K,V})^{\top}}_{576\times T}
\tag{111}
$$

Since RoPE is applied to the $\mathbf{Q}$ and $\mathbf{K}$ matrices, where there is not a single RoPE matrix for all the tokens, i.e., we do not have something like:

$$
\underbrace{\mathbf{X}}_{T\times d_{\text{model}}}\underbrace{\mathbf{W}_Q^{(i)}}_{d_{\text{model}}\times 128}\mathbf{R}^{(i)}{\mathbf{R}^{(i)}}^{T}\underbrace{{\mathbf{W}^{(i)}}_{\uparrow K}^{\top}}_{128\times 576}\underbrace{(\mathbf{X}\,\mathbf{W}_{\downarrow K,V})^{\top}}_{576\times T}
\tag{112}
$$

For this reason, we may consider a decoupled RoPE mechanism, where we have two separate part of each row of $\mathbf{Q}$ and $\mathbf{K}$ matrices, i.e., we break up $\mathbf{q}_t$ and $\mathbf{k}_t$ into two parts, one is the part that does not involve RoPE (of a larger dimension), and the other is the part that involves RoPE (of a smaller dimension).

For each head i, and each token t, we have:

$$
\boldsymbol{k}_t^C = \boldsymbol{W}_{\uparrow K}^{(i)}\,\boldsymbol{c}_t^{KV}, \quad \boldsymbol{v}_t^C = \boldsymbol{W}_{\uparrow V}^{(i)}\,\boldsymbol{c}_t^{KV}
\tag{113}
$$

for the t-th token, we have the following concatenation of outputs:

$$
\boldsymbol{o}_t = \begin{bmatrix}\boldsymbol{o}_{t,1} & \boldsymbol{o}_{t,2} & \dots & \boldsymbol{o}_{t,h}\end{bmatrix}
\tag{114}
$$

which is the weighted sum of $\boldsymbol{v}_j^C$ for $j = 1, 2, \dots, t$, responding to a specific query $\boldsymbol{q}_t$, we now drop the head index i for simplicy:

$$
\boldsymbol{o}_t = \sum_{j=1}^{t}\text{Softmax}\left(\frac{\boldsymbol{q}_t^{\top}\boldsymbol{k}_j}{\sqrt{d_h + d_h^R}}\right)\boldsymbol{v}_j^C
\tag{115}
$$

by letting: $\boldsymbol{c}_t^Q = \boldsymbol{W}_{\downarrow Q}\,\boldsymbol{h}_t, \quad \boldsymbol{c}_t^Q \in \mathbb{R}^{d_{\mathrm{C}}}$, we have:

$$
\begin{aligned}
\boldsymbol{q}_t &= \left[(\underbrace{\boldsymbol{W}_{\uparrow Q}}_{d_h \times d_C} \underbrace{\boldsymbol{W}_{\downarrow Q}}_{d_C \times d_{\text{model}}} \boldsymbol{h}_t) \quad \text{RoPE}(\underbrace{\boldsymbol{W}_Q^R}_{d_h^R \times d_C} \underbrace{\boldsymbol{W}_{\downarrow Q}}_{d_C \times d_{\text{model}}} \boldsymbol{h}_t) \right] \\
&= \underbrace{\left[\boldsymbol{W}_{\uparrow Q}\, \boldsymbol{c}_t^Q \quad \text{RoPE}(\boldsymbol{W}_Q^R\, \boldsymbol{c}_t^Q) \right]}_{(d_h + d_h^R) \times 1}
\end{aligned} \tag{116}
$$

you see that $\mathbf{q}_t$ has two parts, each projecting the hidden state $\mathbf{h}_t$ into a different space by matrices $\boldsymbol{W}_{\uparrow Q}$ and $\boldsymbol{W}_Q^R$. Similiarly, by letting: $\boldsymbol{c}_t^{KV} = \boldsymbol{W}_{\downarrow K}\, \boldsymbol{h}_t, \quad \boldsymbol{c}_t^{KV} \in \mathbb{R}^{d_C}$, we have:

$$
\begin{aligned}
\boldsymbol{k}_j &= \left[(\underbrace{\boldsymbol{W}_{\uparrow K}}_{d_h \times d_C} \underbrace{\boldsymbol{W}_{\downarrow K}}_{d_C \times d_{\text{model}}} \boldsymbol{h}_j) \quad \text{RoPE}(\underbrace{\boldsymbol{W}_K^R}_{d_h^R \times d_C} \underbrace{\boldsymbol{W}_{\downarrow K}}_{d_C \times d_{\text{model}}} \boldsymbol{h}_j) \right] \\
&= \underbrace{\left[\boldsymbol{W}_{\uparrow K}\, \boldsymbol{c}_j^{KV} \quad \text{RoPE}(\boldsymbol{W}_K^R\, \boldsymbol{c}_j^{KV}) \right]}_{(d_h + d_h^R) \times 1}
\end{aligned} \tag{117}
$$

$$
\begin{aligned}
\boldsymbol{k}_t = \left[\boldsymbol{W}_{\uparrow K}\, \boldsymbol{c}_t^{KV} \quad \text{RoPE}(\boldsymbol{W}_K^R\, \boldsymbol{c}_t^{KV}) \right] \in \mathbb{R}^{d_h + d_h^R} \\
\text{where} \quad \boldsymbol{W}_{\uparrow K} \in \mathbb{R}^{d_h \times d_C}, \quad \boldsymbol{W}_K^R \in \mathbb{R}^{d_h^R \times d_C}
\end{aligned} \tag{118}
$$

now we look at the value $\boldsymbol{v}_t^C$, we have:

$$
\begin{aligned}
\boldsymbol{v}_t^C &= \underbrace{\boldsymbol{W}_{\uparrow V}}_{d_h \times d_C} \underbrace{\boldsymbol{W}_{\downarrow KV}}_{d_C \times d_{\text{model}}} \underbrace{\boldsymbol{h}_t}_{d_{\text{model}} \times 1} \\
&= \underbrace{\boldsymbol{W}_{\uparrow V}}_{d_h \times d_C} \underbrace{\boldsymbol{c}_t^{KV}}_{d_C \times 1}
\end{aligned} \tag{119}
$$

during inference, we have the following steps:

- for the t-th token, we compute:

$$
\boldsymbol{q}_t = \left[\boldsymbol{W}_{\uparrow Q}\, \boldsymbol{c}_t^Q \quad \text{RoPE}(\boldsymbol{W}_Q^R\, \boldsymbol{c}_t^Q) \right] \tag{120}
$$

  here, nothing needs to be cached.

- and for each of the j-th token, $\forall j \in \{1, 2, \ldots, t-1\}$, we have:

$$
\boldsymbol{k}_j = \left[\boldsymbol{W}_{\uparrow K}\, \boldsymbol{c}_j^{KV} \quad \text{RoPE}(\boldsymbol{W}_K^R\, \boldsymbol{c}_j^{KV}) \right] \tag{121}
$$

  the things in the cache are:

  – $\boldsymbol{c}_t^{KV} = \boldsymbol{W}_{\downarrow KV}\, \boldsymbol{h}_t$
  – $\boldsymbol{k}_t^R = \text{RoPE}(\boldsymbol{W}_K^R\, \boldsymbol{c}_t^{KV})$

8.8 Flash Attention

8.8.1 traditional softmax algorithm

The algorithm for computing softmax is as follows, given there is a logit set $\{x_i\}_{i=1}^N$, where each $x_i \in \mathbb{R}$. We need three separate loops to compute the softmax output:

1. loop 1: compute the maximum value in the logit set:

$$\begin{aligned} & m_0 = -\infty \\ & \text{for } 1 \leq i \leq N \text{ do} \\ & \quad m_i = \max(m_{i-1}, x_i) \\ & \text{return } m_N \end{aligned} \tag{122}$$

2. loop 2: compute the sum of the exponential of the logits after we obtained m_N which is the maximum value in the logit set:

$$\begin{aligned} & d_0 = 0 \\ & \text{for } 1 \leq i \leq N \text{ do} \\ & \quad d_i = d_{i-1} + \exp^{x_i - m_N} \\ & \text{return } d_N \end{aligned} \tag{123}$$

3. loop 3: normalize the logits:

$$\begin{aligned} & \text{for } 1 \leq i \leq N \text{ do} \\ & \quad a_i = \frac{\exp^{x_i - m_N}}{d_N} \\ & \text{return } \{a_i\}_{i=1}^N \end{aligned} \tag{124}$$

8.8.2 online softmax algorithm

now looking at the second loop, in essence, at the i-th iteration, it is computing the sum of the exponential of the logits from 1 to i:

$$\begin{aligned} d_i &= \sum_{j=1}^{i} \exp^{x_j - m_N} \\ &= \underbrace{\sum_{j=1}^{i-1} \exp^{x_j - m_N}}_{d_{i-1}} + \exp^{x_i - m_N} \\ &= d_{i-1} + \exp^{x_i - m_N} \end{aligned} \tag{125}$$

with the initial condition $d_0 = 0$, the above can also be expressed as:

$$d_0 = 0 \quad \text{and} \quad d_i = d_{i-1} + \exp^{x_i - m_N} \tag{126}$$

However, the above cannot be combined with the first loop, because of the m_N term. Therefore, instead of subtracting a “global” maximum value m_N, we subtract a “intermediate” maximum value m_i up to the i-th iteration:

$$d'_i = \sum_{j=1}^{i} \exp^{x_j - m_i} \tag{127}$$

at the completion of the entire loop, we have $d'_N = d_N$, even though $d_i \neq d'_i \quad \forall i \neq N$ in general. However, if we do the same trick as Eq.(127), however, the first term is not equal to $d'_{i-1} = \sum_{j=1}^{i-1} \exp^{x_j - m_{i-1}}$, i.e., you cannot write the sum as, (a term only contain sum up to $i-1$ terms d_{i-1}) + (a term only contain x_i term).

$$d'_i = \underbrace{\sum_{j=1}^{i-1} \exp^{x_j - m_i}}_{\neq d'_{i-1}} + \exp^{x_i - m_i} \tag{128}$$

but we can make it so, by adding and subtracting a dummy variable:

$$\begin{aligned}
d'_i &= \sum_{j=1}^{i-1} \exp^{x_j - m_i} + \exp^{x_i - m_i} \\
&= \Big(\sum_{j=1}^{i-1} \exp^{x_j - m_i}\Big) \underbrace{\exp^{m_{i-1}} \exp^{-m_{i-1}}}_{\text{dummy variable}} + \exp^{x_i - m_i} && \text{perform} \times \exp^{m_{i-1}} \exp^{-m_{i-1}} \\
&= \Big(\sum_{j=1}^{i-1} \exp^{x_j - m_{i-1}}\Big) \exp^{m_{i-1}} \exp^{-m_i} + \exp^{x_i - m_i} && \text{sawp the red and blue terms} \\
&= \underbrace{\Big(\sum_{j=1}^{i-1} \exp^{x_j - m_{i-1}}\Big)}_{d'_{i-1}} \exp^{m_{i-1} - m_i} + \exp^{x_i - m_i} \\
&= d'_{i-1} \exp^{m_{i-1} - m_i} + \exp^{x_i - m_i}
\end{aligned} \tag{129}$$

therefore, we can have the algorithm to be:

1. loop 1: compute the maximum value in the logit set:

$$\begin{aligned}
&m_0 = -\infty \\
&d_0 = 0 \\
&\text{for } 1 \leq i \leq N \text{ do} \\
&\quad m_i = \max(m_{i-1}, x_i) \\
&\quad d_i = d_{i-1}\exp^{m_{i-1}-m_i} + \exp^{x_i - m_i} \\
&\text{return } m_N, d_N
\end{aligned} \tag{130}$$

we also changed $d_i' \to d_i$ to simplify the notation.

2. loop 2: same as loop 3 in the traditional softmax algorithm:

$$\begin{aligned}
&\text{for } 1 \leq i \leq N \text{ do} \\
&\quad a_i = \frac{\exp^{x_i - m_N}}{d_N} \\
&\text{return } \{a_i\}_{i=1}^{N}
\end{aligned} \tag{131}$$

   since we are not perform "reduce" operation, such as sum, max, etc., as in here, we compute each a_i individually, we cannot combine it with the first loop.

8.8.3 online attention

looking at the single $\mathbf{q}$ row matrix form, where we have:

$$\begin{aligned}
\mathbf{q}\mathbf{K}^\top &= \begin{bmatrix}\mathbf{q}\mathbf{k}_1^\top & \dots & \mathbf{q}\mathbf{k}_m^\top\end{bmatrix} \\
\implies \mathbf{o} &= \text{softmax}(\mathbf{q}\mathbf{K}^\top)\mathbf{V} \\
&= \begin{bmatrix}\text{softmax}\big(\begin{bmatrix}\mathbf{q}\mathbf{k}_1^\top & \dots & \mathbf{q}\mathbf{k}_m^\top\end{bmatrix}\big)\end{bmatrix} \begin{bmatrix} - & \mathbf{v}_1 & - \\ - & \dots & - \\ - & \mathbf{v}_m & - \end{bmatrix} \\
&= \sum_{i=1}^{N} \frac{\exp[\mathbf{q}\mathbf{k}_i^\top]}{\sum_j \exp[\mathbf{q}\mathbf{k}_j^\top]}\mathbf{v}_i \\
&= \sum_{i=1}^{N} a_i \mathbf{v}_i
\end{aligned} \tag{132}$$

taking a particular row of the above, $\mathbf{x} = \mathbf{q}\mathbf{K}^\top$, where $x_i = \mathbf{q}\mathbf{k}_i^\top$ is an element of $\mathbf{x}$:

1. loop 1: compute the maximum value in the logit set:

$$\begin{aligned}
&m_0 = -\infty \\
&d_0 = 0 \\
&\text{for } 1 \leq i \leq N \text{ do} \\
&\quad x_i = q_i \mathbf{k}_i^\top \\
&\quad m_i = \max(m_{i-1}, x_i) \\
&\quad d_i = d_{i-1} \exp^{m_{i-1}-m_i} + \exp^{x_i - m_i} \\
&\text{return } m_N, d_N
\end{aligned} \tag{133}$$

2. loop 2: adding $\mathbf{o} = \sum_{i=1}^{N} a_i \mathbf{v}_i$ to the second loop:

$$\begin{aligned}
&\mathbf{o}_0 = \mathbf{0} \\
&\text{for } 1 \leq i \leq N \text{ do} \\
&\quad a_i = \frac{\exp^{x_i - m_N}}{d_N} \\
&\quad \mathbf{o}_i = \mathbf{o}_{i-1} + a_i \mathbf{v}_i
\end{aligned}
\qquad
\begin{aligned}
&\mathbf{o}_0 = \mathbf{0} \\
&\text{for } 1 \leq i \leq N \text{ do} \\
&\quad \mathbf{o}_i = \mathbf{o}_{i-1} + \frac{\exp^{x_i - m_N}}{d_N} \mathbf{v}_i
\end{aligned} \tag{134}$$

note that we can apply the same trick as Eq.(127) to the second loop:

$$\begin{aligned}
\mathbf{o}_i &= \sum_{i=1}^{N} \frac{\exp^{x_i - m_N}}{d'_N} \mathbf{v}_i \\
\mathbf{o}'_i &= \sum_{j=1}^{i} \frac{\exp^{x_j - m_i}}{d'_i} \mathbf{v}_j
\end{aligned} \tag{135}$$

where we have $\mathbf{o}_N = \mathbf{o}'_N$. Apply the same track as Eq.(127), we have:

$$\begin{aligned}
\mathbf{o}'_i &= \sum_{j=1}^{i} \frac{\exp^{x_j - m_i}}{d'_i} \mathbf{v}_j \\
&= \Big(\sum_{j=1}^{i-1} \frac{\exp^{x_j - m_i}}{d'_i} \mathbf{v}_j\Big) + \frac{\exp^{x_i - m_i}}{d'_i} \mathbf{v}_i \\
&= \underbrace{\Big(\sum_{j=1}^{i-1} \frac{\exp^{x_j - m_i}}{d'_i} \mathbf{v}_j\Big)}_{\neq \mathbf{o}_{i-1}} \frac{d'_{i-1}}{d'_{i-1}} \frac{\exp^{m_{i-1}}}{\exp^{m_{i-1}}} + \frac{\exp^{x_i - m_i}}{d'_i} \mathbf{v}_i \\
&= \underbrace{\Big(\sum_{j=1}^{i-1} \frac{\exp^{x_j - m_{i-1}}}{d'_{i-1}} \mathbf{v}_j\Big)}_{= \mathbf{o}_{i-1}} \frac{d'_{i-1}}{d'_i} \frac{\exp^{m_i}}{\exp^{m_{i-1}}} + \frac{\exp^{x_i - m_i}}{d'_i} \mathbf{v}_i \\
&= \mathbf{o}_{i-1} \frac{d'_{i-1}}{d'_i} \frac{\exp^{m_i}}{\exp^{m_{i-1}}} + \frac{\exp^{x_i - m_i}}{d'_i} \mathbf{v}_i
\end{aligned} \tag{136}$$

8.8.4 final algorithm

everything comes down to just a single loop:

$$
\begin{aligned}
&m_0 = -\infty \\
&d_0 = 0 \\
&\mathbf{o}_0 = \mathbf{0} \\
&\text{for } 1 \leq i \leq N \text{ do} \\
&\quad {\color{red} x_i = q_i \mathbf{k}_i^\top} \\
&\quad m_i = \max(m_{i-1}, x_i) \\
&\quad d_i = d_{i-1} \exp^{m_{i-1} - m_i} + \exp^{x_i - m_i} \\
&\quad \mathbf{o}_i = \mathbf{o}_{i-1} \frac{d_{i-1}}{d_i} \frac{\exp^{m_i}}{\exp^{m_{i-1}}} + \frac{\exp^{x_i - m_i}}{d_i} \mathbf{v}_i \\
&\text{return } \mathbf{o}_N
\end{aligned}
\tag{137}
$$

again, we changed $d'_i \to d_i$ and $\mathbf{o}'_i \to \mathbf{o}_i$ to simplify the notation.

References

[1] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig, "Linguistic regularities in continuous space word representations," in Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies, 2013, pp. 746–751.

[2] Andrea Frome, Greg S Corrado, Jon Shlens, Samy Bengio, Jeff Dean, Marc'Aurelio Ranzato, and Tomas Mikolov, "Devise: A deep visual-semantic embedding model," Advances in neural information processing systems, vol. 26, 2013.

[3] Oscar Perez Concha, Richard Yi Da Xu, Zia Moghaddam, and Massimo Piccardi, "Hmm-mio: an enhanced hidden markov model for action recognition," in CVPR 2011 WORKSHOPS. IEEE, 2011, pp. 62–69.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio, "Neural machine translation by jointly learning to align and translate," arXiv preprint arXiv:1409.0473, 2014.

[5] Minh-Thang Luong, Hieu Pham, and Christopher D Manning, "Effective approaches to attention-based neural machine translation," arXiv preprint arXiv:1508.04025, 2015.

[6] Ilya Sutskever, Oriol Vinyals, and Quoc V Le, "Sequence to sequence learning with neural networks," Advances in neural information processing systems, vol. 27, 2014.

[7] Zhaopeng Tu, Zhengdong Lu, Yang Liu, Xiaohua Liu, and Hang Li, "Modeling coverage for neural machine translation," arXiv preprint arXiv:1601.04811, 2016.

[8] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly, "Pointer networks," Advances in neural information processing systems, vol. 28, 2015.

[9] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher, "Pointer sentinel mixture models," arXiv preprint arXiv:1609.07843, 2016.

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin, "Attention is all you need," Advances in neural information processing systems, vol. 30, 2017.