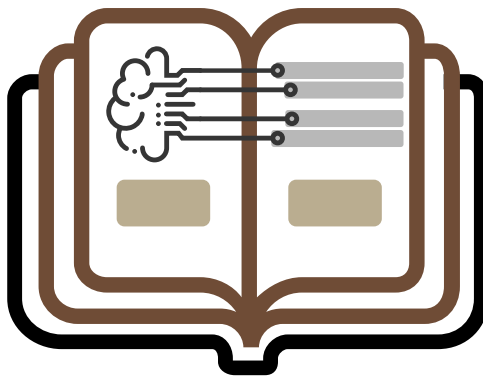# Natural Language Processing

Understanding Language Models

Han Cheol Moon
School of Computer Science and Engineering
Nanyang Technological University
Singapore
hancheol001@e.ntu.edu.sg

October 11, 2025

Contents

8 Encoder-Decoder 42

IV Post-Training 43

9 Preference Optimization 44

V Advanced 49

10 Reasoning Models 50

11 Agentic Models 51

Part I

Introduction

Chapter 1

Introduction

1.1 Linguistics

- **Phonetics**: the study and systematic classification of the sounds made in spoken utterance.

- **Morphology**: the study of the internal structure of words and how they are formed from smaller units of meaning (*i.e., morphemes*).

    - Example: the word *unhappiness* consists of three morphemes:
        1. *un-* (prefix meaning "not"),
        2. *happy* (root meaning "feeling good/pleased"),
        3. *-ness* (suffix turning an adjective into a noun).

- **Syntax**: the way in which linguistic elements (such as words) are put together to form constituents (such as phrases, clauses, grammar, and word order).

    - **Syntactic ambiguity**: occurs when a sentence can be interpreted in more than one way due to its structure.
    - *There are an old man and woman*: This can mean either:
        1. There is an old man and an old woman.
        2. There is an old man and (any) woman.

- **Semantics**: the study of meaning in language; how words, phrases, and sentences correspond to objects, actions, and ideas in the real or imagined world.

    - Includes logical relations like **entailment**: if sentence A is true, sentence B must also be true. Example: *John killed the wasp ⇒ The wasp is dead.*

- **Pragmatics**: the study of how context influences the interpretation of meaning, including speaker intention, implied meaning, and conversational inference.

    - *Can you pass the salt?*: Semantically a question about ability, but pragmatically a request.

1.2   Language Models

1.3   Pre-Training

This achievement is largely motivated by pre-training: we separate common components from many neural network-based systems, and then train them on huge amounts of unlabeled data using self-supervision. These pre-trained models serve as foundation models that can be easily adapted to different tasks via fine-tuning or prompting. As a result, the paradigm of NLP has been enormously changed. In many cases, large-scale supervised learning for specific tasks is no longer required, and instead, we only need to adapt pre-trained foundation models.

The discussion of pre-training issues in NLP typically involves two types of problems: sequence modeling (or sequence encoding) and sequence generation.

Optimizing θ on a pre-training task. Unlike standard learning problems in NLP, pre-training does not assume specific downstream tasks to which the model will be applied. Instead, the goal is to train a model that can generalize across various tasks.

Applying the pre-trained model to downstream tasks. To adapt the model to these tasks, we need to adjust the parameters slightly using labeled data or prompt the model with task descriptions.

Part II

Classical NLP

Chapter 2

Introduction

2.1 Evaluation Metrics

In the context of a bigram model, likelihood refers to the probability of observing a sequence of words in a corpus based on the bigram model's parameters. The bigram model assumes that the probability of a word in a sequence depends only on the immediately preceding word, making it a **Markov model of order 1**.

A bigram model represents the probability of a word sequence $W = (w_1, w_2, \ldots, w_n)$ as a product of conditional probabilities:

$$P(W) = P(w_1) \prod_{i=2}^{n} P(w_i | w_{i-1})$$

Here, $P(w_i | w_{i-1})$ is the conditional probability of word w_i given the preceding word w_{i-1}.

The **likelihood** of a sequence of words (data) $W = (w_1, w_2, \ldots, w_n)$ under the bigram model is:

$$L(\theta | W) = P(W | \theta) = P(w_1 | \theta) \prod_{i=2}^{n} P(w_i | w_{i-1}, \theta)$$

Here, θ represents the model parameters, specifically the probabilities $P(w_i | w_{i-1})$.

Maximum Likelihood Estimation (MLE) in the Bigram Model   To find the parameters $P(w_i | w_{i-1})$ that maximize the likelihood of the observed data, the **maximum likelihood estimation (MLE)** approach is commonly used.

The bigram probabilities $P(w_i | w_{i-1})$ are estimated from the frequency counts in the training corpus:

$$P(w_i | w_{i-1}) = \frac{\text{Count}(w_{i-1}, w_i)}{\text{Count}(w_{i-1})}$$

where:

- $\text{Count}(w_{i-1}, w_i)$ is the number of times the bigram (w_{i-1}, w_i) appears in the corpus.

- $\text{Count}(w_{i-1})$ is the number of times the word w_{i-1} appears.

To simplify computations, the log-likelihood of the sequence is used:

$$\log L(\theta|W) = \log P(w_1|\theta) + \sum_{i=2}^{n} \log P(w_i|w_{i-1}, \theta)$$

- Recall: TP/(TP+FN). Find all relevant cases whithin a dataset.

- Precision: TP/(TP+FP): While recall expresses the ability to find all relevant instances in a dataset, precision expresses the proportion of the data points our model says was relevant actually were relevant.

- The F1 score is the harmonic mean of precision and recall taking both metrics into account in the following equation:

2.1.1 Perplexity

Intuitively, perplexity can be understood as a *measure of uncertainty*. The perplexity of a language model can be seen as the level of perplexity. Consider a language model with an entropy of three bits, in which each bit encodes two possible outcomes of equal probability. This means that when predicting a symbol, that language model has to choose among $2^3 = 8$ possible options. Thus, we can argue that this language model has a perplexity of 8.

It can be modeled as $2^H(P, Q)$:

$$PPL(W) = P(w_1, \cdots, w_N)^{-\frac{1}{N}}$$

$$\approx \left(\prod_{i=1}^{N} P(w_i|w_{<i}) \right)^{-\frac{1}{N}}$$

$$= \sqrt[n]{\frac{1}{\prod_{i=1}^{N} P(w_i|w_{<i})}}$$

Let's derive it from a cross-entropy. We want to optimize P_θ instead of the true distribution P:

$$\mathcal{L}_{CE} = -\mathbb{E}_{w \sim P}[P_\theta(w_i|w_{<i})] \tag{2.1}$$

$$\approx -\frac{1}{N} \sum_{i=1}^{N} \log P_\theta(w_i|w_{<i}) \tag{2.2}$$

$$= -\frac{1}{N} \log \prod_{i=1}^{N} P_\theta(w_i|w_{<i}) \tag{2.3}$$

$$= \log \left(\prod_{i=1}^{N} P_\theta(w_i|w_{<i}) \right)^{-\frac{1}{N}} \tag{2.4}$$

$$= \log \sqrt[N]{\frac{1}{\prod_{i=1}^{N} P_\theta(w_i|w_{<i})}} \tag{2.5}$$

$$\tag{2.6}$$

Thus, $PPL(W) = \exp\left(\mathcal{L}_{CE}\right)$.

2.1.2 Cross-Entropy and Perplexity

$$
\begin{aligned}
H(P, Q) &= -\sum_x P(x) \log Q(x) \\
&= -\sum_x P(x)[\log P(x) + \log Q(x) - \log P(x)] \\
&= -\sum_x P(x)\left[\log P(x) + \log \frac{Q(x)}{P(x)}\right] \\
&= H(P) + D_{KL}(P||Q)
\end{aligned}
$$

It should be noted that since the empirical entropy $H(P)$ is unoptimizable, when we train a language model with the objective of minimizing the cross entropy loss, the true objective is to minimize the KL-divergence of the distribution, which was learned by our language model from the empirical distribution of the language.

Chapter 3

Classical NLP Techniques

3.1 Edit Distance

Edit distance is a method used in spell correction to determine how similar two words are by calculating the minimum number of operations (insertions, deletions, or substitutions) required to transform one word into another. The smaller the edit distance, the more similar the two words are.

For example, let's say we have the following dictionary of valid words: "cat", "car", "cart", "care", "cards", "cast". If the input word is "carr", we calculate the edit distance between "carr" and each word in the dictionary as follows:

- cat: 3 (insert "r" and "r", then delete "t")

- car: 1 (replace second "r" with "t")

- cart: 2 (insert "t" and delete second "r")

The smallest edit distance is 1, between "carr" and "car", so "car" would be selected as the corrected word.

The edit distance method can be improved by using techniques such as weighting the importance of each operation (insertion, deletion, substitution) or using a more sophisticated algorithm such as *Levenshtein distance*. Additionally, the method can be combined with other methods such as language modeling or phonetic analysis to further improve spell correction accuracy.

3.2 Point-wise Mutual Information

Point-wise Mutual Information (PMI) is a statistical measure to calculate the association between two words in a given corpus. PMI is calculated by comparing the probability of the co-occurrence of two words with their individual probabilities of occurrence.

Formally, it is a quantity which is closely related to the mutual information is the point-wise

mutual information. For two events (not random variables) x and y, this is defined as

$$\text{PMI}[x, y] \triangleq \log \frac{p(x, y)}{p(x)p(y)} \tag{3.1}$$

$$= \frac{p(x|y)}{p(x)} = \frac{p(y|x)}{p(y)} \tag{3.2}$$

This measures the discrepancy between these events occurring together compared to what would be expected by chance.

- x and y are two words being considered,

- $P(x)$ is the probability of the occurrence of word x in the corpus,

- $P(y)$ is the probability of the occurrence of word y in the corpus, and

- $P(x, y)$ is the probability of the co-occurrence of words x and y in the corpus. In other words, x and y are adjacent

For terms with three words, the formula becomes:

$$\text{PMI}[x, y, z] = \log \frac{p(x, y, z)}{p(x)p(y)p(z)}$$

PMI values can range from $-\infty$ to ∞. Positive PMI values indicate that the words have a strong association, while negative values indicate that the words are unlikely to appear together.

For example, consider a small corpus of text:

- "The cat sat on the mat. The dog sat on the mat."

- The matrix below is 6×6 considering all possible combination of the "forward" co-occurrences.

$$\begin{bmatrix} & the & cat & dog & sat & on & mat \\ the & 0 & 1 & 1 & 0 & 0 & 2 \\ cat & 0 & 0 & 0 & 1 & 0 & 0 \\ dog & 0 & 0 & 0 & 1 & 0 & 0 \\ sat & 0 & 0 & 0 & 0 & 2 & 0 \\ on & 2 & 0 & 0 & 0 & 0 & 0 \\ mat & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

3.2.1 Remove Stopwords prior to PMI

In the above example we have not removed stopwords, so some of you might be wondering if we need to remove stopwords prior to PMI. It depends on the problem statement but if your objective is to find the related words, you should remove stopwords prior to calculating PMI.

3.3 TF-IDF

The term TF stands for term frequency, and the term IDF stands for inverse document frequency.

The TF-IDF representation takes into account the importance of each word in a document. In the bag-of-words model, each word is assumed to be equally important, which is obviously a less accurate assumption.

The method to calculate the TF-IDF weights of a term in a document is given by the following formula:

- **Term Frequency** (TF), $tf(t, d)$, is the relative frequency of term t within document d,

$$tf(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}},$$

  where $f_{t,d}$ is the raw count of a term in a document, *i.e.,*, the number of times that term t occurs in document d. Note the denominator is simply the total number of terms in document d (counting each occurrence of the same term separately). There are various other ways to define term frequency:

  - The raw count itself: $tf(t, d) = f_{t,d}$
  - Boolean "frequencies":

  $$tf(t, d) = \begin{cases} 1, & \text{if } t \text{ occurs in } d \\ 0 & \text{otherwise} \end{cases}$$

  - Logarithmically scaled frequency: $tf(t, d) = \log(1 + ft, d)$
  - Augmented frequency, to prevent a bias towards longer documents, *e.g.,*. raw frequency divided by the raw frequency of the most frequently occurring term in the document:

  $$tf(t, d) = 0.5 + 0.5 \frac{f_{t,d}}{\max\{f_{t',d} : t' \in d\}}$$

- **Inverse document frequency (IDF)**: *The IDF is a measure of how much information the word provides, i.e.,, if it is common or rare across all documents.* It is the logarithmically scaled inverse fraction of the documents that contain the word, which is obtained by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of that quotient:

$$idf(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

  - N total number of documents in the corpus $(= |D|)$.
  - $|\{d \in D : t \in d\}|$: number of documents where the term t appears. If the term is not in the corpus, this will lead to a division-by-zero. It is therefore common to adjust the numerator $1 + N$ and denominator to $1 + |\{d \in D : t \in d\}|$.
  - Similarly, IDF can also be expressed

  $$idf(t, D) = \log \frac{N - n + 0.5}{n + 0.5} + 1,$$

  where n is the number of documents containing the term t.

3.3.1 Term frequency–inverse document frequency

TF-IDF is calculated as a multiplication of $tf(t, D)$ and $idf(t, D)$.

- A high weight in TF–IDF is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents; the weights hence tend to filter out common terms.

- Since the ratio inside the IDF's log function is always greater than or equal to 1, the value of IDF (and TF–IDF) is greater than or equal to 0.

- As a term appears in more documents, the ratio inside the logarithm approaches 1, bringing the IDF and TF–IDF closer to 0.

In short, TF

- This measures the frequency of a word in a document. If a word appears more times in a document, it is likely more important to the document.

- number of times term "word" appears in a document / Total number of terms in the document

- Consider a document containing 100 words wherein the word 'cat' appears 3 times, then TF is 0.03

IDF:

- This measures the importance of a word in the entire corpus. If a word appears in many documents, it is likely less important to any individual document.

- $\log_e$ total number of documents/ Number of documents with term "word" in it.

- If we have 10 million documents, and 'cat' appears in 1,000 of these. Then, IDF is $\log 10,000,000/1,000 = 4$

Finally, TF-IDF is $0.03 \times 4 = 0.12$ The higher the TF-IDF score, the rarer the term and vice versa.

3.3.2 Link with Information Theory

This expression shows that summing the TF–IDF of all possible terms and documents recovers the mutual information between documents and term taking into account all the specificities of their joint distribution. Each TF–IDF hence carries the "bit of information" attached to a term x document pair.

3.4 Best Match 25 Ranking Algorithm

Best Match 25 ranking algorithm or BM25 is a bag-of-words retrieval function that ranks a set of documents based on the query terms appearing in each document, regardless of their proximity

within the document. It is a family of scoring functions with slightly different components and parameters. One of the most prominent instantiations of the function is as follows.

Given a query Q, containing keywords $q_1, \ldots, q_n$, the BM25 score of a document D is given by

$$\text{score}(D, Q) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgd1})},$$

where $f(q_i, D)$ is the number of times that the keyword q_i occurs in the document, D, $|D|$ is the length of the document D in words, and $avgd1$ is the average document length in a collection of documents. k_1 and b are free parameters, usually chosen, in absence of an advanced optimization, as $k_1 \in [1.2, 2.0]$ and $b = 0.75$.

3.5   Reciprocal Rank Fusion

Reciprocal Rank Fusion (RRF), a simple method for combining the document rankings from multiple information retrieval systems.

$$RRF(d \in D) = \sum_{r \in R} \frac{1}{k + r(d)}$$

- k is a constant that helps to balance between high and low ranking. Typically set at 60.

- $r(d)$ is the rank/position of the document.

3.6   PageRank and TextRank

3.7   Label Smoothing

For each training example x, our model computes the probability of each label $k \in \{1, ..., K\}$, $p(k|x) = \frac{\exp(z_k)}{\sum_i \exp(z_i)}$. Here z_k are the logits.

Label smoothing is a mechanism to regularize the classifier layer by estimating the marginalized effect of label-dropout during training.

Vanila corss-entropy can cause two problem:

- First, it may result in over-fitting: if the model learns to assign full probability to the ground-truth label for each training example, it is not guaranteed to generalize.

- Second, it encourages the differences between the largest logit and all others to become large, and this, combined with the bounded gradient $\frac{\partial \ell}{\partial z_k}$, reduces the ability of the model to adapt. Intuitively, this happens because the model becomes too confident about its predictions.

We propose a mechanism for encouraging the model to be less confident. While this may not be desired if the goal is to maximize the log-likelihood of training labels, it does regularize the model and makes it more adaptable. The method is very simple. Consider a distribution over labels $u(k)$, independent of the training example x, and a smoothing parameter ϵ. For a training example with ground-truth label y, we replace the label distribution $q(k|x) = \delta_{k,y}$ with

$$q'(k|x) = (1 - \epsilon)\delta_{k,y} + \epsilon u(k)$$

which is a mixture of the original ground-truth distribution $q(k|x)$ and the fixed distribution $u(k)$, with weights $1 - \epsilon$ and ϵ, respectively. This can be seen as the distribution of the label k obtained as follows:

$$q'(k|x) = (1 - \epsilon)\delta_{k,y} + \frac{\epsilon}{K}$$

We refer to this change in ground-truth label distribution as label-smoothing regularization, or LSR.

3.7.1   Another Interpretation

Instead of using one-hot encoded vector, we introduce noise distribution $u(y|x)$. Our new ground truth label for data (x_i, y_i) would be

$$
\begin{aligned}
p'(y|x_i) &= (1 - \varepsilon)p(y|x_i) + \varepsilon u(y|x_i) \\
&= \begin{cases} 1 - \varepsilon + \varepsilon u(y|x_i) & \text{if } y = y_i \\ \varepsilon u(y|x_i) & \text{otherwise} \end{cases}
\end{aligned}
$$

Where ε is a weight factor, $\varepsilon \in [0, 1]$, and note that $\sum_{y=1}^{K} p'(y|x_i) = 1$.

Chapter 4

POS Tagging

4.1 Introduction

Part-of-speech (POS) tagging is the process of labeling words in a text with their corresponding parts of speech in natural language processing (NLP). It helps algorithms understand the grammatical structure and meaning of a text.

Part III

Transformers

Chapter 5

Attention Mechanism

TLDR

- *Attention is a communication mechanism*, which can be seen as nodes in a directed graph looking at each other and aggregating information with a weighted sum from all nodes that point to them, with data-dependent weights.

- There is no notion of space. Attention simply acts over a set of vectors. This is why we need to positionally encode tokens.

- Each example across batch dimension is of course processed completely independently and never "talk" to each other

- In an "encoder" attention block just delete the single line that does masking with 'tril', allowing all tokens to communicate. This block here is called a "decoder" attention block because it has triangular masking, and is usually used in autoregressive settings, like language modeling.

- "self-attention" just means that the keys and values are produced from the same source as queries. In "cross-attention", the queries still get produced from x, but the keys and values come from some other, external source (*e.g.,* an encoder module)

- Scaled attention divides the pre-softmax scores QK by its head size. If Q and K have unit variance, this makes the scores roughly unit variance, so the softmax stays diffuse and doesn't saturate.

5.1 Attention

The attention mechanism assigns *attention scores* (*i.e., weights*) to different parts of the input sequence, indicating how much each part contributes to the current output. In essence, the model **pays attention** to certain parts of the input more than others while processing each token in the sequence.

Assume the encoder produces 3 hidden states (each a 2-dimensional vector):

$$h_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad h_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad h_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$
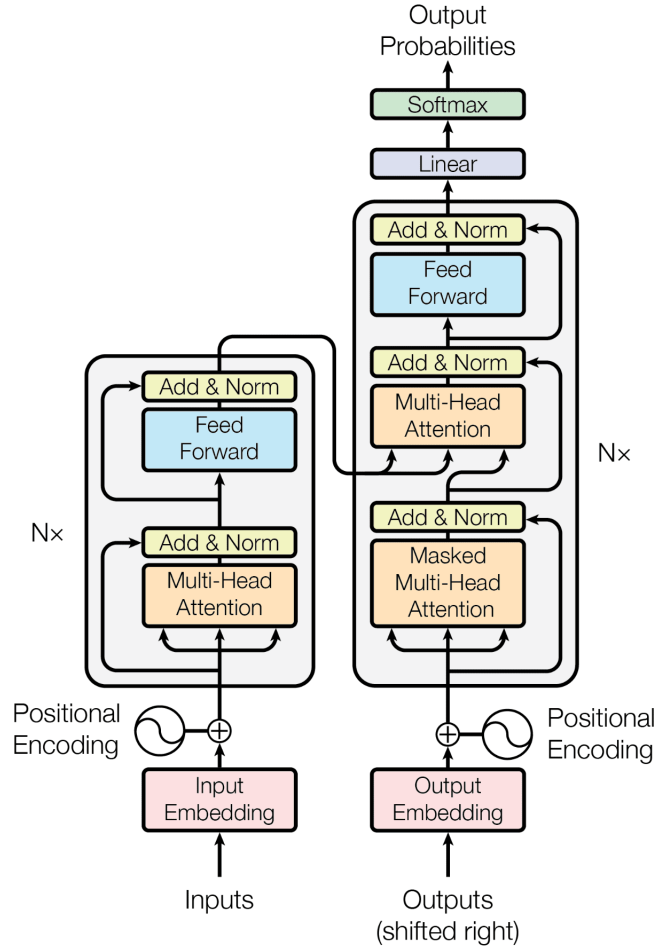
Figure 5.1: An illustration of Transformer architecture.

Let the decoder's previous hidden state at time $t - 1$ be:

$$s_{t-1} = \begin{bmatrix} 0.5 \\ 0.2 \end{bmatrix}.$$

Using the *dot product* as our score function, the alignment score for each encoder hidden state is:

$$e_{tj} = s_{t-1}^{\top} h_j.$$

Compute each:

1. For h_1:

$$e_{t1} = [0.5 \quad 0.2] \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 0.5.$$

2. For h_2:

$$e_{t2} = [0.5 \quad 0.2] \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 0.2.$$

3. For h_3:

$$e_{t3} = [0.5 \quad 0.2] \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 0.7.$$

The attention weight for each encoder time step is given by:

$$\alpha_{tj} = \frac{\exp(e_{tj})}{\sum_{k=1}^{3} \exp(e_{tk})}.$$

Calculate the exponentials:

- $\exp(0.5) \approx 1.6487$,

- $\exp(0.2) \approx 1.2214$,

- $\exp(0.7) \approx 2.0138$.

Sum of exponentials:
$$S = 1.6487 + 1.2214 + 2.0138 \approx 4.8839.$$

Now compute each attention weight:

1. For α_{t1}:
$$\alpha_{t1} = \frac{1.6487}{4.8839} \approx 0.3374.$$

2. For α_{t2}:
$$\alpha_{t2} = \frac{1.2214}{4.8839} \approx 0.2501.$$

3. For α_{t3}:
$$\alpha_{t3} = \frac{2.0138}{4.8839} \approx 0.4125.$$

These weights sum to 1 (up to rounding):

$$0.3374 + 0.2501 + 0.4125 \approx 1.0000.$$

The context vector c_t is the weighted sum of the encoder hidden states:

$$c_t = \alpha_{t1}h_1 + \alpha_{t2}h_2 + \alpha_{t3}h_3.$$

Substitute in the values:

$$\begin{aligned}
c_t &= 0.3374 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 0.2501 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 0.4125 \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 0.3374 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.2501 \end{bmatrix} + \begin{bmatrix} 0.4125 \\ 0.4125 \end{bmatrix} \\
&= \begin{bmatrix} 0.3374 + 0.4125 \\ 0 + 0.2501 + 0.4125 \end{bmatrix} \\
&= \begin{bmatrix} 0.7499 \\ 0.6626 \end{bmatrix}.
\end{aligned}$$

So, the context vector is approximately:

$$c_t \approx \begin{bmatrix} 0.75 \\ 0.66 \end{bmatrix}.$$

In a typical decoder, the context vector c_t is combined with the previous hidden state and possibly the previously generated output to update the current hidden state. For example, an update could be:

$$s_t = f(s_{t-1}, y_{t-1}, c_t),$$

or, if you are using a simple formulation with a combined input, it might be:

$$s_t = \tanh(W[s_{t-1}; c_t]),$$

where $[s_{t-1}; c_t]$ denotes the concatenation of s_{t-1} and c_t, and W is a learnable weight matrix. The updated state s_t would then be used to predict the next output token.

Let's use a machine translation task (English to French) as an example. Suppose we are translating the sentence "I am learning" into French.

- Input: Sequence of words in English: '[I, am, learning]'

- Output: Sequence of words in French: '[Je, suis, en, train, d'apprendre]'

Instead of compressing all the input information into a fixed-size context vector (like in traditional encoder-decoder models), the attention mechanism allows the decoder to look at different parts of the input sentence at each step of the decoding process.

	I	am	learning
Je	0.7	0.2	0.1
suis	0.1	0.8	0.1
en	0.05	0.15	0.8

This matrix shows that "Je" strongly attends to "I", "suis" attends mostly to "am", and "en" attends primarily to "learning".

5.1.1 Self-Attention

- n: the number of tokens in the sequence.

- d_{model}: the dimension of the input embeddings.

- d_k: the dimension of the query and key vectors.

- d_v: the dimension of the value vectors (often $d_k = d_v$, but they need not be equal).

Assume we have an input sequence of n tokens. For each token i (with $1 \leq i \leq n$) we start with an embedding vector $\mathbf{x}_i \in \mathbb{R}^{d_{\text{model}}}$. In self-attention, we first linearly project these embeddings into three different spaces to obtain the *query*, *key*, and *value* vectors:

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

where

- $W^Q, W^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ are the query and key projection matrices,

- $W^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ is the value projection matrix,

- d_k (and sometimes d_v) is a chosen dimensionality for these spaces.

For a given token i, we compute its output representation as a weighted sum of the value vectors of all tokens. The weights are determined by the similarity between the query $\mathbf{q}_i$ and the keys $\mathbf{k}_j$ of all tokens j in the sequence.

First, compute the *dot-product scores* between the query for token i and every key:

$$s_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j, \quad \text{for } j = 1, 2, \ldots, n.$$

To prevent the dot products from growing too large in magnitude (especially when d_k is large), we *scale the scores* by $\sqrt{d_k}$:

$$\tilde{s}_{ij} = \frac{s_{ij}}{\sqrt{d_k}}.$$

Next, we apply the softmax function over the scaled scores for token i to obtain the attention weights α_{ij}:

$$\alpha_{ij} = \frac{\exp(\tilde{s}_{ij})}{\sum_{l=1}^{n} \exp(\tilde{s}_{il})}.$$

These weights satisfy $\sum_{j=1}^{n} \alpha_{ij} = 1$.

Finally, the output for token i, denoted by $\mathbf{z}_i$, is the weighted sum of the value vectors:

$$\mathbf{z}_i = \sum_{j=1}^{n} \alpha_{ij} \, \mathbf{v}_j.$$

In matrix form for all tokens, if we define matrices Q, K, and V whose rows are the vectors $\mathbf{q}_i$, $\mathbf{k}_i$, and $\mathbf{v}_i$ respectively, the self-attention operation is:
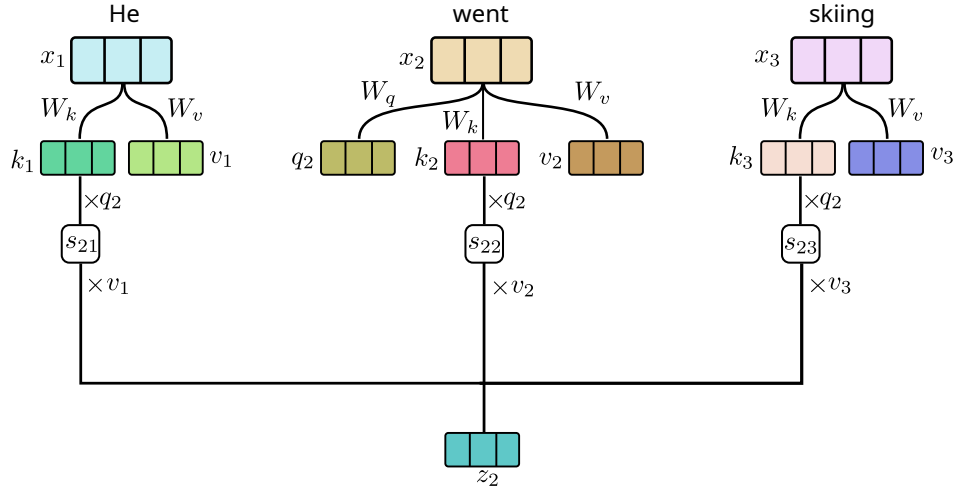
Figure 5.2: An illustration of the self-attention. The final output is the weighted sum of the value vectors.

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

Here, $Q, K, V \in \mathbb{R}^{n \times d_{model}}$, QK^T's time complexity is $O(N^2 d)$. This quadratic cost is massive for long input-sequences such as documents to be summarized or character-level inputs.

Input Embeddings   Each token i is represented by an embedding:

$$\mathbf{x}_i \in \mathbb{R}^{d_{\text{model}}}.$$

You can think of all the tokens put together as a matrix:

$$X \in \mathbb{R}^{n \times d_{\text{model}}}.$$

Projection Matrices   To obtain the queries, keys, and values, we use learned projection matrices:

$$W^Q \in \mathbb{R}^{d_{\text{model}} \times d_k},$$
$$W^K \in \mathbb{R}^{d_{\text{model}} \times d_k},$$
$$W^V \in \mathbb{R}^{d_{\text{model}} \times d_v}.$$

Projected Matrices   Multiplying the input X by these weight matrices gives:

$$Q = X W^Q \in \mathbb{R}^{n \times d_k},$$
$$K = X W^K \in \mathbb{R}^{n \times d_k},$$
$$V = X W^V \in \mathbb{R}^{n \times d_v}.$$

Here, each row of Q (or K, or V) corresponds to the query (or key, or value) of one token.

Dot-Product Attention Scores The attention scores between tokens are computed using:

$$QK^\top \in \mathbb{R}^{n \times n}.$$

In this product:

- Q is $n \times d_k$
- $K^\top$ is $d_k \times n$

Therefore, the result is an $n \times n$ matrix where each entry (i, j) represents the (unnormalized) similarity between token i and token j.

Scaled Dot-Product and Softmax Before applying the softmax, the scores are scaled by $\sqrt{d_k}$:

$$\frac{QK^\top}{\sqrt{d_k}} \in \mathbb{R}^{n \times n}.$$

Then, applying the softmax function row-wise produces an attention weight matrix $A \in \mathbb{R}^{n \times n}$:

$$A = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right).$$

You can also view the scaling factor as a **temperature**:

$$\text{softmax}\left(\frac{QK^\top}{\tau}\right).$$

- Bigger τ: softer (more uniform) distribution.
- Bigger τ: sharper distribution.

Final Output Finally, the output of the self-attention layer is computed as:

$$\text{Attention}(Q, K, V) = AV.$$

Since:

- A is $n \times n$,
- V is $n \times d_v$,

The final output is:

$$\text{Attention}(Q, K, V) \in \mathbb{R}^{n \times d_v}.$$

5.1.2 Masked Attention

In autoregressive tasks (*e.g.,* language modeling), it is essential that when predicting a token at position i, the model does not peek at any tokens at positions $j > i$. *Masked attention* ensures that each token only attends to tokens at the same or earlier positions. The masked attention is often referred to *cross-attention*. This is just a self-attention in decoder.

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

where M

$$M_{ij} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases}$$

Note that $-\infty$ will make *exp* term to be zero.

5.1.3 Multi-Head Attention

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. **Rather than computing a single attention function with full-dimensional queries, keys, and values, the mechanism splits them into multiple "heads" and computes attention in parallel**.

Suppose:

- The input embeddings (or previous layer outputs) form the matrix $X \in \mathbb{R}^{n \times d_{\text{model}}}$,

- d_{model} is the model (or embedding) dimension,

- We decide to use h attention heads.

Each head will work with lower-dimensional projections of the input. In particular, we typically set:

$$d'_k = d'_v = \frac{d_{\text{model}}}{h},$$

so that each head processes queries, keys, and values of dimensions d'_k and d'_v, and the total computation remains efficient.

Linear Projections for Each Head For each head $i \in \{1, \ldots, h\}$, we define learned projection matrices:

$$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d'_k},$$
$$W_i^K \in \mathbb{R}^{d_{\text{model}} \times d'_k},$$
$$W_i^V \in \mathbb{R}^{d_{\text{model}} \times d'_v}.$$

We then project the input X to obtain:

$$Q_i = XW_i^Q \in \mathbb{R}^{n \times d'_k},$$
$$K_i = XW_i^K \in \mathbb{R}^{n \times d'_k},$$
$$V_i = XW_i^V \in \mathbb{R}^{n \times d'_v}.$$
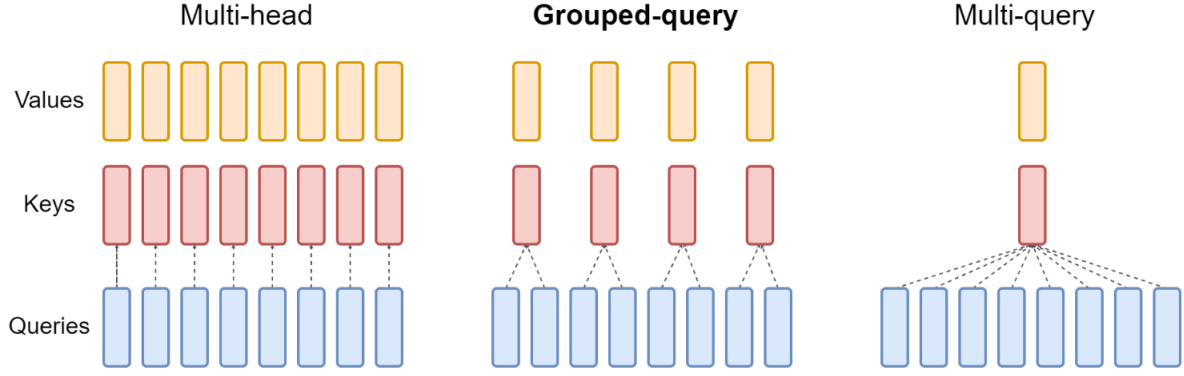
Compute Scaled (Masked) Dot-Product Attention for Each Head For each head i, compute:
$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i),$$

where the attention function is defined as:

$$\text{head}_i = \text{softmax}\left(\frac{Q_i K_i^\top + M}{\sqrt{d_k'}}\right) V_i.$$

- If masking is not required (*e.g.,* in the encoder or in *non-autoregressive* settings), simply set $M = 0$.

- *For decoder self-attention in autoregressive models*, M is defined as in the Masked Attention section above.

Concatenate the Heads and Project Once all heads are computed, we concatenate their outputs:
$$\text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h) \in \mathbb{R}^{n \times (h \cdot d_v')}.$$

Finally, we apply a learned linear projection:

$$W^O \in \mathbb{R}^{(h \cdot d_v') \times d_{\text{model}}},$$

to obtain the final multi-head attention output:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \in \mathbb{R}^{n \times d_{\text{model}}}.$$

5.2 Various Attention Mechanisms

If we use a single value and a single key over all queries, we can significantly reduce the memory usage for *KV caching*. This is the basic idea of the *multi-query attention* (MQA) and the *grouped query attention* (GQA). However, MQA tends to lower the output quality as the model size increases. GQA achieved a balance between them.

Chapter 6

Inference of Autoregressive Model

6.1 Overview of Inference in Autoregressive Models

In an autoregressive transformer (*e.g.,* GPT), tokens are generated one at a time. During inference, the model must compute attention for the next token based on all previously generated tokens. However, recomputing the entire attention from scratch at each time step would be inefficient. Thus, these models often use *caching* to store intermediate results (the keys and values) from previous time steps.

- **Training**: The model can process the entire sequence in parallel using masked self-attention. A mask (typically a triangular matrix) prevents tokens from "seeing" future tokens.

- **Inference**: The model generates one token at a time. At time step $t+1$, it uses the cached tokens $[x_1, x_2, \ldots, x_t]$ to compute the probability distribution for the next token.

To avoid recomputing keys and values for tokens $x_1, \ldots, x_t$ at every step, the model stores them (usually for each layer). When a new token is generated, only its query needs to be computed, and then the cached keys and values are used to compute the attention.

The technique of storing and reusing the computed keys and values from previous tokens during autoregressive generation is commonly called *KV-caching* (short for Key-Value Caching).

6.2 KV-Caching

Step 1: Previous Tokens and Cached Representations   Assume that by time step t the model has generated tokens:
$$x_1, \ x_2, \ \ldots, \ x_t.$$

For a given transformer layer, let the cached key and value matrices be:
$$K_{\leq t} \in \mathbb{R}^{t \times d_k},$$
$$V_{\leq t} \in \mathbb{R}^{t \times d_v},$$
where d_k and d_v are the key and the value dimensions, respectively. They are often set to be the same.

Step 2: Compute the Query for the New Token When generating the next token x_{t+1}, its input (often the embedding of the previously generated token or a special start symbol) is used to compute a query vector for each layer:

$$q_{t+1} \in \mathbb{R}^{d_k}.$$

This is computed by a linear projection:

$$q_{t+1} = x_{t+1} W^Q,$$

where $W^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$ is the learned projection matrix.

Step 3: Compute the Attention Scores The new query q_{t+1} is compared with all the cached keys. In a single attention head, the (scaled) dot-product attention scores are computed as:

$$s_{t+1,j} = \frac{q_{t+1} \cdot k_j}{\sqrt{d_k}} \quad \text{for } j = 1, 2, \ldots, t,$$

and often, for implementation convenience, the new token's own key k_{t+1} is also computed and appended to the cache. In that case, you would have:

$$s_{t+1,j} = \frac{q_{t+1} \cdot k_j}{\sqrt{d_k}} \quad \text{for } j = 1, 2, \ldots, t+1.$$

Since the model is autoregressive, the mask is implicit. There are no "future" tokens beyond $t+1$ at inference time. (If you do compute for all $t+1$ positions, a mask would ensure that token $t+1$ only attends to tokens 1 through $t+1$.)

Step 4: Apply the Softmax to Get Attention Weights The scores are then normalized with the softmax function to obtain the attention weights:

$$\alpha_{t+1,j} = \frac{\exp(s_{t+1,j})}{\sum_{j'=1}^{t+1} \exp(s_{t+1,j'})}, \quad j = 1, \ldots, t+1.$$

These weights determine how much the new token attends to each of the previous tokens (and its own representation, if included).

Step 5: Compute the Weighted Sum of Values The output of the attention layer for the new token is then computed as:

$$z_{t+1} = \sum_{j=1}^{t+1} \alpha_{t+1,j} \, v_j,$$

where each v_j is the value vector from the cache (or computed for the new token in the case of $j = t+1$):

$$v_j = x_j W^V, \quad j = 1, \ldots, t+1.$$

This z_{t+1} is then passed on through the rest of the transformer layer (including feed-forward sub-layers, layer normalization, etc.) to eventually produce logits over the vocabulary.

Step 6: Generate the Next Token and Update the Cache

- The model uses the final output (after all transformer layers) to compute a probability distribution over the vocabulary.

- A token is chosen (*e.g.,* via sampling or greedy decoding) and appended to the sequence.

- The new token's key and value vectors (from each layer) are computed and added to the cache so that future tokens can attend to it.

6.2.1 Inference without Caching

Let's take a look at the following example:

- Number of tokens so far: $t = 3$. We have already generated tokens $\{x_1, x_2, x_3\}$. We now want to generate token x_4.

- Model dimensionality: To keep it simple, let's say each token embedding is 2-dimensional ($d_{\text{model}} = 2$) and the attention uses a single head with key/query dimension $d_k = 2$ and value dimension $d_v = 2$.

- Token embeddings (just made-up numbers):

$$x_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad x_2 = \begin{bmatrix} 3 \\ 4 \end{bmatrix}, \quad x_3 = \begin{bmatrix} 5 \\ 6 \end{bmatrix}, \quad x_4 = \begin{bmatrix} ? \\ ? \end{bmatrix}$$

  The forth one is the one we want to generate. We will compute attention for tokens x_1, x_2, x_3, x_4 all at once.

- Projection matrices (W^Q, W^K, W^V) are each 2×2 for this example. For instance, we have the following matrices:

$$W^Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad W^K = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}, \quad W^V = \begin{bmatrix} 0.5 & -0.5 \\ 1.0 & 0.5 \end{bmatrix}.$$

- Keys:

$$k_i = x_i \, W^K$$

  For $i = 1, 2, 3, 4$:

  - $k_1 = \begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \end{bmatrix}$

  - $k_2 = \begin{bmatrix} 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 10 \end{bmatrix}$

  - $k_3 = \begin{bmatrix} 5 & 6 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 16 \end{bmatrix}$

  - $k_4 = \begin{bmatrix} ?, & ? \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} ? & ? \end{bmatrix}$

- Values:

$$v_i = x_i \, W^V$$

  For $i = 1, 2, 3, 4$:

$$- v_1 = \begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 0.5 & -0.5 \\ 1.0 & 0.5 \end{bmatrix} = \begin{bmatrix} 2.5 & 0.5 \end{bmatrix}$$

$$- v_2 = \begin{bmatrix} 3 & 4 \end{bmatrix} \begin{bmatrix} 0.5 & -0.5 \\ 1.0 & 0.5 \end{bmatrix} = \begin{bmatrix} 5.5 & 0.5 \end{bmatrix}$$

$$- v_3 = \begin{bmatrix} 5 & 6 \end{bmatrix} \begin{bmatrix} 0.5 & -0.5 \\ 1.0 & 0.5 \end{bmatrix} = \begin{bmatrix} 8.5 & 0.5 \end{bmatrix}$$

$$- v_4 = \begin{bmatrix} ?, & ? \end{bmatrix} \begin{bmatrix} 0.5 & -0.5 \\ 1.0 & 0.5 \end{bmatrix} = \begin{bmatrix} ? & ? \end{bmatrix}$$

All of the above must be computed at the current time step if we do not use caching.

With KV caching, you would *not* recalculate k_1, k_2, k_3 and v_1, v_2, v_3. Instead:

- You already have $\{k_1, k_2, k_3\}$ and $\{v_1, v_2, v_3\}$ stored from the previous steps.
- You only compute:
  - q_4 (the query for the new token),
  - k_4, v_4 (the new key and value to add to the cache).

In other words, you skip re-projecting and re-computing every key and value from tokens $\{1, 2, 3\}$. This saves a substantial amount of computation when generating long sequences, especially in large transformers (like GPT).

Note that storing data in the cache uses up memory space. Systems with limited memory resources may struggle to accommodate this additional memory overhead, potentially resulting in out-of-memory errors. This is especially the case when long inputs need to be processed, as the memory required for the cache grows linearly with the input length and the batch size.

Computational Cost In sum, the vanilla self-attention on n tokens,

- Each token needs to look at all n tokens to get attention scores
- Thus, the cost is $O(n^2)$

For instance, if we given a sentence with three tokens,

1. First token: Look at 1 token (cost: $O(1^2)$)
2. Second token: Look at 2 tokens (cost: $O(2^2)$)
3. Third token: Look at 3 tokens (cost: $O(3^2)$)

If we add up all these costs for generating a sequence of length n, we get:

$$O(1^2 + 2^2 + 3^2 + \cdots + n^2) \approx O(n^3)$$
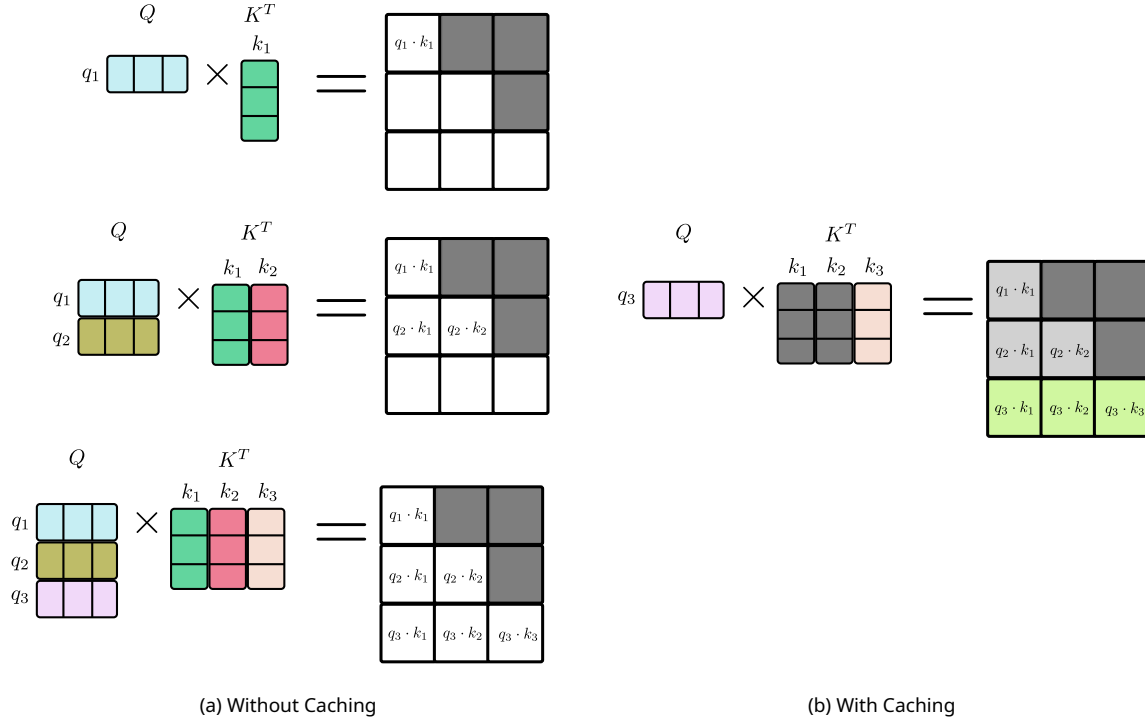
With caching,

Figure 6.1: An example of KV caching

1. Process 1 token cost $O(1)$

2. Process 1 new token + look at 1 cached token cost $O(2)$

3. Process 1 new token + look at 2 cached tokens cost $O(3)$

Adding these up:

$$O(1 + 2 + 3 + \cdots + n) = O(n^2)$$

Chapter 7

Positional Embedding

The self-attention mechanism in transformers treats all tokens in a sequence in parallel without an inherent notion of order. This means that, by itself, self-attention is invariant to the order of input tokens. Positional encoding is introduced to inject order information so that the model can differentiate between tokens based on their positions.

- **Absolute Positional Embeddings**: Each position in the sequence is assigned a unique vector. Although straightforward, these embeddings don't scale well to longer sequences and fail to capture the nuances of relative positions between tokens.

- **Relative Positional Embeddings**: These embeddings focus on the distance between tokens, which can improve the model's understanding of token relationships. However, they typically introduce additional complexity into the model architecture.

7.1 Permutation Invariance of Self-Attention

Without positional embeddings, the transformer's self-attention would treat the input as a bag of tokens, ignoring the order entirely. The added positional embeddings break this permutation invariance by encoding the position directly into the token representation. As a result, even if the same tokens are present, the model can infer their relative order and roles in the sentence.

Let's look at an example:

```python
import torch
import torch.nn as nn
from transformers import AutoTokenizer, AutoModel

# Small, public model (no auth required)
model_id = "prajjwal1/bert-tiny"
tok = AutoTokenizer.from_pretrained(model_id)
model = AutoModel.from_pretrained(model_id)

text = "The dog chased another dog"
tokens = tok(text, return_tensors="pt")["input_ids"]          # [batch, seq]
embeddings = model.get_input_embeddings()(tokens)              # [batch, seq,
    hidden]
hdim = embeddings.shape[-1]

# Randomly initialized MHA
```

```
16  W_q = nn.Linear(hdim, hdim, bias=False)
17  W_k = nn.Linear(hdim, hdim, bias=False)
18  W_v = nn.Linear(hdim, hdim, bias=False)
19  mha = nn.MultiheadAttention(embed_dim=hdim, num_heads=4, batch_first=True)
20
21  with torch.no_grad():
22      for param in mha.parameters():
23          nn.init.normal_(param, std=0.1)
24
25  output, _ = mha(W_q(embeddings), W_k(embeddings), W_v(embeddings))
26
27  # With BERT tokenization, sequence is:
28  # [CLS], "the", "dog", "chased", "another", "dog", [SEP]
29  dog1_out = output[0, 2]
30  dog2_out = output[0, 5]
31  breakpoint()
32  print(f"Dog output identical?: {torch.allclose(dog1_out, dog2_out, atol=1e-6)}"
        )
```

We use raw token embeddings (no positional encodings). Then, identical tokens (*i.e.,* "dog") at different positions produce identical attention outputs under our randomly initialized MHA.

If you simply pass this sentence into a transformer, the model wouldn't know the order of the words. The same set of token (*i.e.,* word) embeddings could represent a sentence like "chased dog the dog another" if no ordering information were provided.

We formulate this as follows: With a permutation matrix P of shape (n, n), the input tokens can be permuted as
$$X' = PX.$$

Let's follow the same self-attention process for X':

$$Q' = X'W_q = PXW_q = PQ,$$

$$K' = X'W_k = PXW_k = PK,$$

$$V' = X'W_v = PXW_v = PV.$$

Then, compute the scores using Q' and K':

$$S' = \frac{Q'(K')^T}{\sqrt{d_k}} = \frac{(PQ)(PK)^T}{\sqrt{d_k}}.$$

Note that

$$(PK)^T = K^T P^T,$$

so

$$S' = \frac{PQK^T P^T}{\sqrt{d_k}} = P\,S\,P^T.$$

The softmax is applied row-wise.
$$A' = \text{Softmax}(S').$$

Since P and P^T are just reordering rows and columns, respectively, the attention weights A are simply permuted like below:
$$A' = P\,A\,P^T.$$

Finally, the output for the permuted input is:

$$
\begin{aligned}
\text{Attention}(X') &= A'V' \\
&= (PAP^T)(PV) = PAV \quad \text{, since } PP^T = I \\
&= P\,\text{Attention}(X).
\end{aligned}
$$

As you can see the self-attention mechanism is *equivariant* to permutations. This means that if you permute the input tokens, the output is permuted in the same way. There is no mechanism in the equations above that distinguishes one ordering from another; the operations treat all tokens symmetrically.

Thus, without additional positional encodings, if you were to shuffle the tokens, the model would compute the same set of pairwise interactions—just in a different order. The structure of the equations does not provide any mechanism for the model to know that one token came before or after another.

7.2   Sinusoidal (Absolute) Positional Encoding

To encode positional information to tokens, there are several desirable properties:

1. Unique encoding for each position

2. Linear relation between two encoded positions: If we know the encoding for position p, it should be straightforward to compute the encoding for position $p + k$, making it easier for the model to learn positional patterns.

3. Generalizes to longer sequences: it should generalize outside their training distribution.

4. Generated by a deterministic process the model can learn: This allows the model to learn the mechanism

5. Extensible to multiple dimensions: To handle multimodality.

In Transformer, positional encoding vectors are added to the token (word) embeddings before the input is fed into the self-attention layers. This addition gives the model a sense of the order in the sequence, enabling it to capture the sequential relationships between tokens despite processing them in parallel. The absolute positional encoding is given by

- $\text{PE}(pos, 2i) = \sin\left(\dfrac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$

- $\text{PE}(pos, 2i + 1) = \cos\left(\dfrac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right),$

where:

- *pos*: the position of the token in the sequence (starting at 0),

- i: the index along the embedding dimension,

- d_{model}: the total dimension of the model's embeddings.
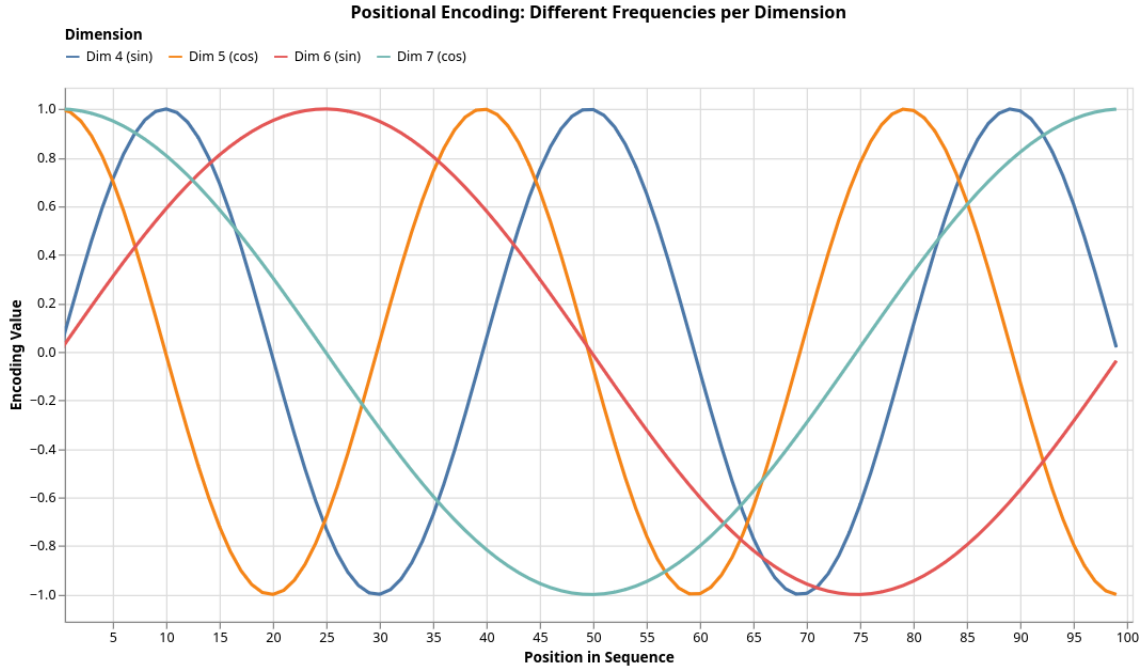
Figure 7.1: The visualization of absolute positional embedding. Each dimension oscillates at a different frequency and gets a unique positional encoding.

7.2.1 Example

Let's work through a concrete example with a very small embedding dimension:

- Assume $d_{\text{model}} = 4$.

- We'll compute the positional encoding for two positions: $pos = 0$ and $pos = 1$.

Since $d_{\text{model}} = 4$, we have 4 dimensions indexed $0, 1, 2, 3$. The formulas split the dimensions into even and odd indices:

For $pos = 0$

- Dimension 0 (even index, $i = 0$):

$$\text{PE}(0, 0) = \sin\left(\frac{0}{10000^{\frac{2 \cdot 0}{4}}}\right) = \sin\left(\frac{0}{10000^0}\right) = \sin(0) = 0.$$

- Dimension 1 (odd index, $i = 0$):

$$\text{PE}(0, 1) = \cos\left(\frac{0}{10000^{\frac{2 \cdot 0}{4}}}\right) = \cos(0) = 1.$$

- Dimension 2 (even index, $i = 1$):

$$\text{PE}(0, 2) = \sin\left(\frac{0}{10000^{\frac{2 \cdot 1}{4}}}\right) = \sin\left(\frac{0}{10000^{0.5}}\right) = \sin(0) = 0.$$

- Dimension 3 (odd index, $i = 1$):

$$\text{PE}(0, 3) = \cos\left(\frac{0}{10000^{\frac{2\cdot1}{4}}}\right) = \cos(0) = 1.$$

- So, the positional encoding for $pos = 0$ is:

$$[0,\ 1,\ 0,\ 1].$$

For $pos = 1$

- Dimension 0 (even index, $i = 0$):

$$\text{PE}(1, 0) = \sin\left(\frac{1}{10000^{\frac{2\cdot0}{4}}}\right) = \sin\left(\frac{1}{10000^0}\right) = \sin(1) \approx 0.84147.$$

- Dimension 1 (odd index, $i = 0$):

$$\text{PE}(1, 1) = \cos\left(\frac{1}{10000^{\frac{2\cdot0}{4}}}\right) = \cos(1) \approx 0.54030.$$

- Dimension 2 (even index, $i = 1$):

$$\text{PE}(1, 2) = \sin\left(\frac{1}{10000^{\frac{2\cdot1}{4}}}\right) = \sin\left(\frac{1}{10000^{0.5}}\right) = \sin\left(\frac{1}{100}\right) = \sin(0.01) \approx 0.00999983.$$

- Dimension 3 (odd index, $i = 1$):

$$\text{PE}(1, 3) = \cos\left(\frac{1}{10000^{\frac{2\cdot1}{4}}}\right) = \cos(0.01) \approx 0.99995.$$

- Thus, the positional encoding for $pos = 1$ is approximately:

$$[0.84147,\ 0.54030,\ 0.00999983,\ 0.99995].$$

7.2.2 Rotation Matrix Aspect

For each frequency ω_i, the sinusoidal PE at position p uses a pair

$$\left[\sin(\omega_i p),\ \cos(\omega_i p)\right].$$

Think of this as a 2D vector on the unit circle with angle $\theta = \omega_i p$.

If you move from position p to $p + k$, the angle increases by $\omega_i k$. The new pair is obtained by a 2×2 rotation:

$$\begin{bmatrix} \sin(\omega_i(p+k)) \\ \cos(\omega_i(p+k)) \end{bmatrix} = \underbrace{\begin{bmatrix} \cos(\omega_i k) & -\sin(\omega_i k) \\ \sin(\omega_i k) & \cos(\omega_i k) \end{bmatrix}}_{\text{rotation by angle } \omega_i k} \begin{bmatrix} \sin(\omega_i p) \\ \cos(\omega_i p) \end{bmatrix}.$$

This is just the angle-addition identity written as a matrix multiply. The key is a relative shift k acts as a linear rotation in each $\sin, \cos$ 2D subspace.

The full PE concatenates many such pairs for different ω_i. Collect them into a vector

$$\text{PE}(p) = \begin{bmatrix} \sin(\omega_1 p), \cos(\omega_1 p), & \sin(\omega_2 p), \cos(\omega_2 p), & \ldots \end{bmatrix}.$$

A shift by k is then a **block-diagonal rotation**:

$$\text{PE}(p + k) = \underbrace{\text{diag}\big(R(\omega_1 k), R(\omega_2 k), \ldots\big)}_{=:\ R(k)}\ \text{PE}(p),$$

where each block $R(\omega_i k)$ is the 2×2 rotation above.

- **Relative structure**: Because $\text{PE}(p+k) = R(k)\text{PE}(p)$, relative offsets are linearly encoded.

- Extrapolation: No learned parameters; positions outside training range still lie on circles $\rightarrow$ better length extrapolation than learned embeddings.

- **RoPE connection**: RoPE takes the rotation view further by rotating Q and K in these 2D subspaces, making attention depend on relative positions more cleanly than simple addition.

- **Dot-product behavior** (vanilla additive PE): In a standard Transformer, we add PE to token embeddings: $x_p = e_p + \text{PE}(p)$. Attention logits involve

$$x_p^\top x_q = e_p^\top e_q + e_p^\top \text{PE}(q) + \text{PE}(p)^\top e_q + \text{PE}(p)^\top \text{PE}(q).$$

  The PE–PE term depends on $(cos(w_i(q - p)))$ (via angle differences), which injects relative info, but cross terms mix with content.

Example   Pick one frequency $\omega = \frac{2\pi}{T}$ (period T). If $T = 8 \Rightarrow \omega = \frac{\pi}{4}$, $(p = 2)$, $(k = 3)$:

$$\text{PE}(2) = \begin{bmatrix} \sin(\tfrac{\pi}{2}), \cos(\tfrac{\pi}{2}) \end{bmatrix} = [1, 0].$$

Rotation by $\omega k = \frac{3\pi}{4}$:

$$R = \begin{bmatrix} \cos \frac{3\pi}{4} & -\sin \frac{3\pi}{4} \\ \sin \frac{3\pi}{4} & \cos \frac{3\pi}{4} \end{bmatrix} = \begin{bmatrix} -\frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{bmatrix}.$$

Then $R[1, 0]^\top = \begin{bmatrix} -\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2} \end{bmatrix}^\top$, which equals $\begin{bmatrix} \sin(\omega(p + k)), \cos(\omega(p + k)) \end{bmatrix}$ as expected, where $\omega(p + k) = 5\pi/4$.

7.2.3   Issues of APE

- Poor length extrapolation

  - **Models see a finite range of absolute indices during training**. At inference, new indices (longer contexts) map to unseen p_i—distribution shift.
  - Sinusoidal APE helps a bit (positions are computed, not learned), but periodicity can cause aliasing at very long lengths and still doesn't give a clean relative bias.

- Translation non-equivariance

  – If you shift the entire sequence by $+k$, absolute indices change, so the same local
    pattern appears under different p_i. The model must relearn invariances that are
    naturally relative (*e.g.,* the next token depends on the previous few tokens, regardless
    of where they are).

- Polluting the semantic information of token embeddings by adding PE.

- No direct encoding of relative distance

  – With APE, attention scores don't decompose nicely into a term that is a function of
    $(i - j)$. The model can learn to approximate relative behavior, but it's indirect and
    fragile.

- Fixed or awkward max length

  – Learned APE needs a table up to some L_{max}. Going beyond it requires interpolation
    or ad-hoc extension; both can degrade quality.

  – Even sinusoidal forms often require careful frequency choices and still degrade for
    out-of-distribution lengths.

- Streaming / chunking pain:

  – For long-form or streaming inference, resetting positions per chunk changes absolute
    indices and can cause mismatches when stitching attention across chunks. Extra
    engineering (position remapping) is needed.

7.3   Rotary Position Embedding (RoPE)

Rather than adding a positional vector to the token embeddings, **RoPE rotates the embed-
dings by a position-specific angle.** Think of it as twisting the embedding in space based on
its position. For instance, if you have a simple two-dimensional embedding for the word "dog",
you can imagine its vector being rotated by an angle θ if it's the first word, 2θ if it's the second
word, and so on.

For a high-dimensional embedding $\mathbf{q}$ (*e.g.,* in $\mathbb{R}^d$) at position p, we create a block diagonal matrix
$\text{diag}\big(R(\omega_1 p), R(\omega_2 p), \ldots, R(\omega_{d/2} p)\big)$. Then, we can rotate each dimension of $\mathbf{q}$:

$$
\begin{bmatrix}
R_1 & \mathbf{0} & \cdots & \mathbf{0} \\
\mathbf{0} & R_2 & \cdots & \vdots \\
\vdots & \ddots & \ddots & \mathbf{0} \\
\mathbf{0} & \cdots & \mathbf{0} & R_{d/2}
\end{bmatrix}
\begin{bmatrix}
q_1 \\
q_2 \\
\vdots \\
q_d
\end{bmatrix}
$$

As you can see, RoPE divides the vector into $d/2$ pairs (or 2D subspaces). For a token at position
i, denote its embedding by:

$$
\mathbf{x}_i \in \mathbb{R}^d.
$$

We partition $\mathbf{x}_i$ into pairs:

$$
\mathbf{x}_i^{(k)} = \begin{pmatrix} x_{i,2k} \\ x_{i,2k+1} \end{pmatrix}, \quad k = 0, 1, \ldots, \frac{d}{2} - 1.
$$

Subsequently, for each 2D subspace indexed by k, RoPE defines a rotation angle:

$$\theta_{i,k} = i \cdot \alpha_k,$$

where α_k is a scaling factor that typically depends on the dimension k. A popular choice is:

$$\alpha_k = \frac{1}{10000^{\frac{2k}{d}}}.$$

This scaling mimics the frequency scaling in sinusoidal embeddings (*i.e.*, positional embedding), ensuring that different subspaces capture positional information at different granularities.

In two-dimensional geometry, any rotation by an angle θ can be represented by a *rotation matrix* $R(\theta)$. This matrix is a standard tool in linear algebra and has the form:

$$R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}.$$

We compute its rotated version:

$$\text{RoPE}_i\big(\mathbf{x}_i^{(k)}\big) = R(\theta_{i,k})\,\mathbf{x}_i^{(k)} = \begin{pmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{pmatrix} \begin{pmatrix} x_{i,2k} \\ x_{i,2k+1} \end{pmatrix}.$$

This yields the updated coordinates:

$$\tilde{x}_{i,2k} = x_{i,2k}\cos(\theta_{i,k}) - x_{i,2k+1}\sin(\theta_{i,k}),$$

$$\tilde{x}_{i,2k+1} = x_{i,2k+1}\cos(\theta_{i,k}) + x_{i,2k}\sin(\theta_{i,k}).$$

After processing all $d/2$ subspaces, we concatenate the results back into a full d-dimensional vector $\tilde{\mathbf{x}}_i$.

In transformer architectures, **RoPE is applied to both the query and the key vectors** except the value vecotrs:

$$\tilde{\mathbf{q}}_i = \text{RoPE}_i(\mathbf{q}_i),$$
$$\tilde{\mathbf{k}}_j = \text{RoPE}_j(\mathbf{k}_j).$$

Then, the attention score between positions i and j is calculated as:

$$\text{Attention}(i,j) = \frac{\tilde{\mathbf{q}}_i \cdot \tilde{\mathbf{k}}_j}{\sqrt{d}}.$$

A key property of RoPE is that the dot product between the rotated vectors can be reinterpreted to show how relative positions are encoded. In particular, one can derive that:

$$\tilde{\mathbf{q}}_i^\top \tilde{\mathbf{k}}_j = \mathbf{q}_i^\top \mathbf{M}(i,j)\,\mathbf{k}_j,$$

where $\mathbf{M}(i,j)$ is a block diagonal matrix that encapsulates the effect of the relative positional difference $j - i$.

Focus on one 2D subspace (indexed by k):

- The query subvector at position i is rotated by $\theta_{i,k} = i\alpha_k$.

- The key subvector at position j is rotated by $\theta_{j,k} = j\alpha_k$.

The dot product in this subspace is:

$$\tilde{\mathbf{q}}_i^{(k)\top}\tilde{\mathbf{k}}_j^{(k)} = \left(\mathbf{q}_i^{(k)}\right)^\top R(\theta_{i,k})^\top R(\theta_{j,k})\,\mathbf{k}_j^{(k)}.$$

Since the transpose of a rotation matrix is its inverse (i.e., $R(\theta)^\top = R(-\theta)$), we have:

$$R(\theta_{i,k})^\top R(\theta_{j,k}) = R(-\theta_{i,k})R(\theta_{j,k}) = R\big(\theta_{j,k} - \theta_{i,k}\big).$$

Because $\theta_{j,k} - \theta_{i,k} = (j-i)\alpha_k$, the transformation becomes:

$$R\big((j-i)\alpha_k\big).$$

Repeating this for each 2D subspace results in a block diagonal matrix:

$$\mathbf{M}(i,j) = \mathrm{diag}\Big(R\big((j-i)\alpha_0\big),\ R\big((j-i)\alpha_1\big),\ \ldots,\ R\big((j-i)\alpha_{\frac{d}{2}-1}\big)\Big).$$

Each block is the 2×2 rotation matrix:

$$R\big((j-i)\alpha_k\big) = \begin{pmatrix} \cos\big((j-i)\alpha_k\big) & -\sin\big((j-i)\alpha_k\big) \\ \sin\big((j-i)\alpha_k\big) & \cos\big((j-i)\alpha_k\big) \end{pmatrix}.$$

- Relative Positional Bias: The matrix $\mathbf{M}(i,j)$ adjusts the dot product between queries and keys based on their relative positions $(j-i)$. Thus, the value vectors are not modified. Instead of explicitly adding a relative position embedding, the rotation inherently modulates the interaction between tokens.

- Unified Encoding: Since $\mathbf{M}(i,j)$ is built from standard rotation matrices $R(\theta)$, it seamlessly encodes the relative positional difference across all 2D subspaces. This results in a unified treatment where both absolute and relative positional cues are embedded into the attention calculation.

- Elegant Mathematical Foundation: The use of $R(\theta)$ comes directly from classical geometry and linear algebra. It leverages the well-known properties of rotations in 2D—specifically, that rotations preserve vector norms and that the composition of rotations is itself a rotation (with the angle being the sum or difference of the individual angles). This mathematical elegance translates into an efficient and effective mechanism for positional encoding.

- Geometric Interpretation:
  - The rotation matrix $R(\theta)$ rotates any 2D vector by the angle θ (in the counterclockwise direction) while preserving its magnitude.
  - This property makes it ideal for encoding positional shifts—rotating a vector **does not change its content (its norm) but changes its direction**, thereby encoding positional information.

- Inherent Relative Encoding: When different positions correspond to different rotation angles, **the relationship between any two positions can be captured by the difference in their rotation angles**. This leads to a natural encoding of relative position without having to explicitly compute or store separate relative position vectors.

7.4   Position Interpolation (PI)

If a model is trained with a maximum context length (L) but we want to read sequences up to $L' = sL$, where $s > 1$. PI *compresses* absolute indices so that positions never exceed the range seen in training. Concretely, for a token at position i (0-based), PI uses a *scaled* index

$$\hat{i} = \frac{i}{s}.$$

RoPE angles are then computed with $\hat{i}$ instead of i. Using the notation from above, each 2D subspace k is rotated by

$$\hat{\theta}_{i,k} = \hat{i}\alpha_k = \frac{i}{s}\alpha_k, \quad \text{where} \quad \alpha_k = \frac{1}{10000^{2k/d}}$$

Thus the PI-rotated query/key components become

$$\widetilde{\mathbf{q}}_i^{(k)} = R\left(\frac{i}{s}\alpha_k\right)\mathbf{q}_i^{(k)}, \qquad \widetilde{\mathbf{k}}_j^{(k)} = R\left(\frac{j}{s}\alpha_k\right)\mathbf{k}_j^{(k)}.$$

The per-subspace attention contribution depends on the *scaled* relative offset:

$$\left(\widetilde{\mathbf{q}}_i^{(k)}\right)^\top \left(\widetilde{\mathbf{k}}_j^{(k)}\right) = \left(\mathbf{q}_i^{(k)}\right)^\top R\left(\frac{j-i}{s}, \alpha_k\right)\mathbf{k}_j^{(k)}.$$

Equivalently, PI replaces $j - i$ with $(j - i)/s$ in the block-diagonal matrix $\mathbf{M}(i, j)$. In other words, **PI uniformly stretches the wavelength** in all RoPE planes by the same factor s.

In sum, PI squeezes the new sequence inside the original context window. Desptie its simplicity, PI has several limitations:

- It normally requires finetuning on about billion tokens.

- the perplexity slightly increases

7.5   NTK-Aware Scaled RoPE

Recall that RoPE encodes positions by rotating each query/key vector by angles that grow with token index. Those angles come from a bank of sinusoidal frequencies $\theta_i = b^{-2i/d}$, where b is the base, typically $10,000$, d is the head dimension, and i indexes each 2D subspace.

During the pre-training, the inputs are squeezed into chunks of sequences of equal amounts of tokens L. After the pre-training, it tends to perform badly on sequences longer than L. Instead of training again on longer chunks, *NTK-aware scaling* changes the base of RoPE. Empirically this lets you extend context (often 2–4×) with small perplexity hit, even without fine-tuning; with light fine-tuning it goes further.

Let s be the *extension factor* (*e.g.,* going $4k \to 16k$ means $s = 4$), and let $|D| = d$ be the per-head dimension. NTK-aware scaling replaces the RoPE base b by a larger base b' by

$$b' = b \cdot s^{\frac{d}{d-2}}$$

Then, you recompute RoPE frequencies as usual:

$$\theta_i' = b'^{-2i/d} \quad i = 0 \ldots \frac{d}{2} - 1.$$

This is the *Adjusted Base Frequency (ABF)* view derived from NTK theory; it leaves token indices m unchanged and only modifies the frequency bank.

In short, the trick is to use a larger RoPE base while keeping token indices m unchanged.

7.5.1   Intuition

Increasing b slows the phase growth of rotations at high i (*i.e.,* high frequencies), so relative angles between nearby tokens remain similar even when absolute positions are much larger. In the NTK view, that keeps the model's kernel (and thus its inductive bias) closer to what it saw in training, mitigating the usual long-range decay/aliasing of RoPE. Empirical and theoretical works tie context length capability tightly to the RoPE base; too small a base leads to superficial long-context behavior (low perplexity but poor retrieval).

```
# d_head = per-head dimension (even)
# base = 10000.0 by default
# s = target_context / train_context (e.g., 16_384/4_096 -> 4.0)

def ntk_aware_inv_freq(d_head: int, base: float, s: float):
    import math, torch
    b_prime = base * (s ** (d_head / (d_head - 2)))
    idx = torch.arange(0, d_head, 2.0)
    inv_freq = (b_prime ** (-idx / d_head))  # shape [d_head/2]
    return inv_freq  # use to build cos/sin angles as in standard RoPE
```

7.6   YaRN: Yet another RoPE extensioN

Extend context length *without* over-warping local geometry. YaRN does this by (*i*) **interpolating frequencies per dimension** instead of applying a single uniform scale, and (*ii*) optionally applying a mild **attention-temperature** adjustment; an *inference-only* dynamic variant is also possible.

7.6.1   Dimension-Aware ("by-parts") Interpolation

We define λ_k as the wavelength of the RoPE embedding at d-th hidden dimension:

$$\lambda_k = \frac{2\pi}{\alpha_k},$$

where $k = 0, 1, \ldots, d/2 - 1$. The wavelength describes the length of tokens needed in order for the RoPE embedding at dimension k to perform a full rotation (2π). Let L be the training context and $s = L'/L$ the desired extension factor. We can define a *usage ratio*, which counts how many cycles the k-th RoPE plane completes across the training window L:

$$r_k \triangleq \frac{L}{\lambda_k} = \frac{L\alpha_k}{2\pi}.$$

- Small r_k means $\lambda_k \gg L$ (low frequency / long wave). Across the whole training window, this plane barely completes a fraction of a cycle

- Large r_k means $\lambda_k \ll L$ (high frequency / short wave). It oscillates many times inside L.

Choose thresholds $0 < \alpha < \beta$ and a ramp

$$\gamma(r) = \begin{cases} 0, & r < \alpha \\ \frac{r-\alpha}{\beta-\alpha} & \alpha \le r \le \beta \\ 1, & r > \beta. \end{cases}$$

YaRN **interpolates** each plane's frequency between "no change" and a PI-like scaling:

$$\alpha_k^{(\text{yarn})} = \underbrace{(1 - \gamma(r_k))}_{\text{keep low-freq}} \cdot \alpha_k + \underbrace{\gamma(r_k)}_{\text{scale high-freq}} \cdot \frac{\alpha_k}{s}.$$

- Leaves low frequencies alone

- Scales high frequencies strongly

Equivalently, the angle at position i becomes

$$\theta_{i,k}^{(\text{yarn})} = i \cdot \alpha_k^{(\text{yarn})} = i \cdot \left[(1 - \gamma(r_k))\alpha_k + \gamma(r_k)\frac{\alpha_k}{s} \right].$$

Long-wave (low-frequency) planes preserve their original geometry (good for coarse structure), while short-wave planes receive stronger stretching (good for avoiding aliasing at long absolute positions). Mid bands are smoothly blended. Compared with PI, this limits over-compression of frequencies that were already gentle at the training window.

7.6.2 Attention-Temperature Adjustment

YaRN also introduces a softmax temperature t to stabilize attention over longer spans:

$$\text{Attention}(i,j) = \text{softmax}\left(\frac{\widetilde{\mathbf{q}}_i^\top \widetilde{\mathbf{k}}_j}{t\sqrt{d}} \right).$$

The original paper suggests to use (for $s \ge 1$):

$$\sqrt{\frac{1}{t}} = 0.1 \ln s + 1.$$

The t decreases smoothly as the window grows. This mitigates over-sharp attention when many positions are competing at extended lengths.

7.6.3 Comparison Summary

- **PI:** $i \to i/s$ (uniform across planes).

- **NTK-aware base change:** keep i but reduce all frequencies by increasing the RoPE base (global shift).

- **YaRN:** *per-dimension* blending between NTK-aware and PI-like scaling with mild attention temperature.

Chapter 8

Encoder-Decoder

8.1 Encoder-Decoder

8.1.1 Encoder

8.1.2 Decoder

The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did. And just like we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word.

```python
def forward(self, tgt: Tensor, memory: Tensor) -> Tensor:
    seq_len, dimension = tgt.size(1), tgt.size(2)
    tgt += position_encoding(seq_len, dimension)
    for layer in self.layers:
        tgt = layer(tgt, memory)
    return torch.softmax(self.linear(tgt), dim=-1)
```

Part IV

Post-Training

Chapter 9

Preference Optimization

9.1 Introduction

To make a large language model (LLM) actually useful for answering questions, we typically start from a base model and apply post-training (often called fine-tuning). Unlike massive web corpora used for pre-training, post-training data are smaller, curated, and purpose-built. Post-training is used to align the model's behavior (helpful, truthful, harmless) and to amplify capabilities acquired during pre-training. Common approaches include supervised fine-tuning (SFT) and reinforcement learning from human feedback (RLHF). Following DeepSeek R1, reinforcement learning with verifiable rewards (RLVR) has gained traction for improving reasoning and coding.

Verifiable rewards are signals that can be checked by a simple, trustworthy procedure, so the agent cannot easily game them. Rather than rewarding "doing a good job" in the abstract, you only reward outputs that pass an external test (a verifier).

A basic RL objective for post-training is

$$\max_{\pi} \ \mathbb{E}_{y \sim \pi(\cdot|x)}\big[r(x,y)\big] \ - \ \beta \, \mathrm{KL}(\pi(\cdot \mid x) \, \| \, \pi_0(\cdot \mid x)),$$

where

- r: For a prompt x and model output y, we can define a scalar reward $r(x,y)$ that measures how desirable y is.

- An SFT model (*i.e.,* policy model) π and a frozen reference model π_0.

- KL term regularizes π to stay close to π_0.

9.1.1 Reward Models in RLHF

The InstructGPT paper popularized training a reward model (RM) from human preference rankings, now central to RLHF. An RM predicts how closely a candidate response matches what humans would prefer for a given prompt. To collect preference data, we sample a prompt and several model responses; human raters rank the responses (best→worst). These pairwise/ordinal labels train the RM, which then supplies a scalar reward during RL fine-tuning.
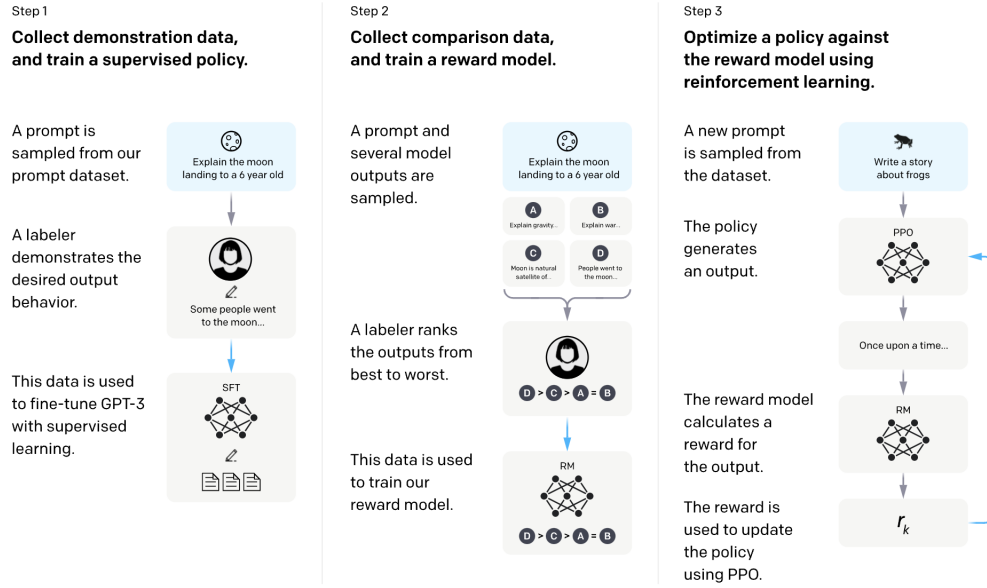
Preference Data Example

```
1 {
2 "prompt": [
3   {"system":"You are a helpful, honest assistant."},
4   {"role": "user", "content": "What color is the sky?"},
5 ],
6 "chosen": [{"role": "assistant", "content": "Washington, D.C."}],
7 "rejected": [{"role": "assistant", "content": "? The capital of the United
      States is Washington, D.C."}],
8 }
```

9.1.2   Training Rewards Model

A standard way to train an RM draws on the *Bradley–Terry model* for pairwise preferences. For two items i and j with positive scores p_i, p_j, the probability that i is preferred over j is

$$P(i \succ j) = \frac{p_i}{p_i + p_j}.$$

For LLM preferences, the two items are completions y_1, y_2 to the same prompt x. Let an RM $r_\theta(x, y)$ assign a scalar score to each completion. The pairwise preference probability can be written as

$$P\big(y_1 \succ y_2 \mid x\big) = \sigma\Big(r_\theta(x, y_1) - r_\theta(x, y_2)\Big),$$

where $\sigma(\cdot)$ is the logistic function. Training the RM then amounts to maximizing the likelihood of observed human preferences under this model.

9.2   Direct Preference Optimization

In RL, the goal is to train an agent to maximize a reward signal. However, in many real-world scenarios, the reward function is not explicitly known or is difficult to define. Instead, we often

have access to preference data, where humans provide comparisons between different outputs or trajectories (*e.g.,* "Output A is better than Output B").

Traditional approaches to this problem involve:

- Learning a reward function from preference data.

- Using the learned reward function to train a policy via RL.

DPO simplifies this process by directly optimizing the policy to align with the preference data, bypassing the need for an explicit reward function. DPO can be represented as follows:

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x,y_w,w_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right], \quad (9.1)$$

where

- π_{ref} represents a *reference policy* (*e.g.,* a pre-trained or baseline policy).

- π_θ represents the *learned policy* parameterized by θ.

Let's deep dive into this equation. DPO leverages the *Bradley-Terry model,* a probabilistic framework for pairwise comparisons, to directly optimize the policy. The key idea is to express the probability of one output being preferred over another in terms of the policy's action probabilities. This allows the policy to be trained directly on preference data.

Assume that we have a dataset $\mathcal{D} = \{(x_i, y_i^1, y_i^2, p_i)\}_{i=1}^N$, where:

- x_i: Input context.

- y_i^1, y_i^2: Two possible outputs (*e.g.,* text responses or actions).

- p_i: Preference label, where $p_i = 1$ if y_i^1 is preferred over y_i^2, and $p_i = 0$ otherwise.

The Bradley-Terry model defines the probability that y_i^1 is preferred over y_i^2 as:

$$P(y_i^1 \succ y_i^2 \mid x_i) = \frac{\exp(r(x_i, y_i^1))}{\exp(r(x_i, y_i^1)) + \exp(r(x_i, y_i^2))},$$

where $r(x, y)$ is a reward function.

In DPO, the reward function $r(x, y)$ is parameterized by the learned policy $\pi_\theta(y \mid x)$ and the reference policy π_{ref}:

$$r(x, y) = \beta\log\frac{\pi_\theta(y \mid x)}{\pi_{\text{ref}}(y \mid x)},$$

where:

- β: A temperature parameter controlling the sharpness of the policy.

- $\pi_\theta(y \mid x)$: The probability of output y under the learned policy.

- $\pi_{\text{ref}}(y \mid x)$: The probability of output y under the reference policy.

This formulation ensures that the reward is tied to how much the learned policy π_θ deviates from the reference policy π_{ref}.

```python
1  import torch
2  import torch.nn as nn
3  import torch.optim as optim
4  from torch.utils.data import DataLoader, Dataset
5  import numpy as np
6
7  # Set random seed for reproducibility
8  torch.manual_seed(42)
9  np.random.seed(42)
10
11 # Define a simple neural network for the policy
12 class PolicyNetwork(nn.Module):
13     def __init__(self, input_dim, output_dim):
14         super(PolicyNetwork, self).__init__()
15         self.fc = nn.Sequential(
16             nn.Linear(input_dim, 128),
17             nn.ReLU(),
18             nn.Linear(128, output_dim),
19             nn.Softmax(dim=-1)  # Output is a probability distribution
20         )
21
22     def forward(self, x):
23         return self.fc(x)
24
25 # Synthetic dataset for preference data
26 class PreferenceDataset(Dataset):
27     def __init__(self, num_samples, input_dim):
28         self.num_samples = num_samples
29         self.input_dim = input_dim
30         self.x = torch.randn(num_samples, input_dim)  # Random input contexts
31         self.y1 = torch.randint(0, 2, (num_samples,))  # Random output 1
32         self.y2 = torch.randint(0, 2, (num_samples,))  # Random output 2
33         self.p = torch.randint(0, 2, (num_samples,))  # Random preferences (0
    or 1)
34
35     def __len__(self):
36         return self.num_samples
37
38     def __getitem__(self, idx):
39         return self.x[idx], self.y1[idx], self.y2[idx], self.p[idx]
40
41 # DPO loss function
42 def dpo_loss(pi_theta, pi_ref, y1, y2, p, beta=1.0):
43     # Compute log probabilities under the learned and reference policies
44     log_pi_theta_y1 = torch.log(pi_theta.gather(1, y1.unsqueeze(1))).squeeze()
45     log_pi_theta_y2 = torch.log(pi_theta.gather(1, y2.unsqueeze(1))).squeeze()
46     log_pi_ref_y1 = torch.log(pi_ref.gather(1, y1.unsqueeze(1))).squeeze()
47     log_pi_ref_y2 = torch.log(pi_ref.gather(1, y2.unsqueeze(1))).squeeze()
48
49     # Compute the reward differences
50     r_y1 = beta * (log_pi_theta_y1 - log_pi_ref_y1)
51     r_y2 = beta * (log_pi_theta_y2 - log_pi_ref_y2)
52
53     # Compute the preference probability using the Bradley-Terry model
54     logits = r_y1 - r_y2
55     loss = -torch.mean(p * torch.log(torch.sigmoid(logits)) + (1 - p) * torch.
    log(torch.sigmoid(-logits)))
56     return loss
57
58 # Hyperparameters
59 input_dim = 10  # Dimension of input context
```

```python
60  output_dim = 2  # Number of possible outputs (binary for simplicity)
61  num_samples = 1000  # Number of preference pairs
62  batch_size = 32
63  learning_rate = 1e-3
64  num_epochs = 10
65  beta = 1.0  # Temperature parameter
66
67  # Create dataset and dataloader
68  dataset = PreferenceDataset(num_samples, input_dim)
69  dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
70
71  # Initialize policies
72  pi_theta = PolicyNetwork(input_dim, output_dim)  # Learned policy
73  pi_ref = PolicyNetwork(input_dim, output_dim)  # Reference policy (fixed)
74  pi_ref.eval()  # Freeze the reference policy
75
76  # Optimizer
77  optimizer = optim.Adam(pi_theta.parameters(), lr=learning_rate)
78
79  # Training loop
80  for epoch in range(num_epochs):
81      for x, y1, y2, p in dataloader:
82          # Forward pass: compute probabilities under the learned and reference
83      policies
83          pi_theta_probs = pi_theta(x)
84          with torch.no_grad():
85              pi_ref_probs = pi_ref(x)
86
87          # Compute DPO loss
88          loss = dpo_loss(pi_theta_probs, pi_ref_probs, y1, y2, p, beta)
89
90          # Backward pass and optimization
91          optimizer.zero_grad()
92          loss.backward()
93          optimizer.step()
94
95      print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}")
96
97  # Test the learned policy
98  test_x = torch.randn(1, input_dim)  # Random test input
99  pi_theta_probs = pi_theta(test_x)
100 print(f"Test input: {test_x}")
101 print(f"Learned policy probabilities: {pi_theta_probs}")
```

Part V

Advanced

Chapter 10

Reasoning Models

Compared to regular LLMs, reasoning LLMs tend to break down a problem into smaller steps (often called reasoning steps or thought processes) before answering a given question.

Chapter 11

Agentic Models

Bibliography

[1] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[2] Rachit Singh. . `https://rachitsingh.com/elbo_surgery/`, 2017. Online; accessed 29 January 2014.

[3] Hany Hassan, Anthony Aue, Chang Chen, Vishal Chowdhary, Jonathan Clark, Christian Federmann, Xuedong Huang, Marcin Junczys-Dowmunt, William Lewis, Mu Li, Shujie Liu, Tie-Yan Liu, Renqian Luo, Arul Menezes, Tao Qin, Frank Seide, Xu Tan, Fei Tian, Lijun Wu, Shuangzhi Wu, Yingce Xia, Dongdong Zhang, Zhirui Zhang, and Ming Zhou. Achieving human parity on automatic chinese to english news translation. *CoRR*, abs/1803.05567, 2018.