# Week 1: The Foundations of Text Segmentation

- **Word Tokenization:** This is the most common and intuitive method, where the text is segmented into individual words, typically splitting on spaces and punctuation. For example, the sentence "Machine learning is fascinating" becomes the list ["Machine", "learning", "is", "fascinating"]. This approach is effective for languages like English that use explicit word boundaries (spaces).
- **Character Tokenization:** This method segments the text into its smallest possible units: individual characters. The word "Machine" becomes ["M", "a", "c", "h", "i", "n", "e"]. This is a granular approach that is highly beneficial for tasks like spelling correction or for processing languages that do not have clear word boundaries, such as Chinese or Japanese.
- **Subword Tokenization:** This method strikes a balance between word and character tokenization. It breaks words into units that are larger than characters but often smaller than whole words, typically separating root words from common prefixes and suffixes. For example, "Chatbots" might become ["Chat", "bots"]. This is a powerful, modern technique for handling complex words and, as will be discussed, the problem of unknown words.

**Code-Switching**- For an NLP pipeline, code-switching presents a severe challenge. Most NLP models are trained on a large corpus of a single language, such as English. This training process builds a fixed vocabulary based on that English data. When a code-switched sentence like "Quiero jugar outside" is provided as input, a standard English tokenizer and model will encounter the words "Quiero" and "jugar." These words are not in the model's English vocabulary and will be classified as Out-of-Vocabulary (OOV) words

**White-Space Tokenization**

A common first attempt at tokenization is to use a simple "white-space tokenizer," which functions by splitting a string based on spaces

**Problem 1: Conjunctions and Contractions** Consider the sentences *"She's reading a book"* and *"I don't"*. A white-space tokenizer would produce and `["I", "don't"]`. This is problematic. The token *"She's" is ambiguous*; it could mean *"She is" or "She has*." Similarly, "don't" combines the tokens "do" and "not."

**Problem 2: Hyphenated Texts**

- **Case 1 (Single Token):** A word like "twenty-five" should almost always be considered a single token representing a single numerical concept.
- **Case 2 (Multiple Tokens):** A compound modifier like "California-based" is more ambiguous. Depending on the task, it might be best to treat it as one token ["California-based"] or split it ["California", "-", "based"].

**Problem 3: Punctuation**

Consider the text "I love coding!" or "apple, oranges, pears". A white-space split would produce ["I", "love", "coding!"] and ["apple,", "oranges,", "pears"]. The model would then build a vocabulary where "coding" and "coding!" are treated as two *completely different and unrelated words*.

**Stemming**

The goal of stemming is to reduce inflected or derived forms of a word to their common "root form," known as a "stem".

A stemmer operates by applying a set of simple, pre-defined rules to "chop off" or "remove" common prefixes and suffixes (affixes) from words. It does this without any regard for the word's linguistic context or meaning.

A key characteristic of stemming is that the resulting "stem" is often *not a real, valid dictionary word*.

- The words "running," "runner," and "ran" might all be stemmed to "run," which is a successful outcome.
- However, "arguing" and "argued" might be stemmed to "argu," which is not a word.
- "Caring" might be stemmed to "Car," which is a real word but has a completely different meaning.
- "university" and "universal" might both be stemmed to "univers".

**Lemmatization**

Its goal is to reduce a word not to a crude "stem," but to its true dictionary form, known as its "lemma".

Unlike stemming's simple "chopping" rules, lemmatization is a true linguistic process. To work correctly, a lemmatizer requires two key components:

1. **A Vocabulary:** A dictionary of the language to look up words and their valid base forms.
2. **Context (Part-of-Speech):** The word's Part-of-Speech (POS) tag—whether it is a noun, verb, adjective, etc..

The key feature of lemmatization is that its output is *always* a valid, meaningful dictionary word.

- **Examples:**
    - "running" (as a verb) becomes "run".
    - "studies" (as a noun) becomes "study".
    - "was" (as a verb) becomes "be".
    - "better" (as an adjective) becomes "good".

**Both stemming and lemmatization aim to solve the same core problem: reducing the inflectional variation in a text to a common base form. This normalization helps shrink the vocabulary size and mitigate the OOV problem. However, they achieve this goal in vastly different ways, presenting a classic NLP trade-off: Speed vs. Accuracy.**

| Feature | Stemming | Lemmatization |
|---|---|---|
| **Goal** | Reduce word to its "stem" (root form) | Reduce word to its "lemma" (dictionary form) |
| **Method** | Crude, rule-based heuristic (e.g., "chop off -ing") | Linguistic analysis (uses vocabulary, morphology, POS) |
| **Output** | May be a non-word (e.g., "argu", "univers") | Always a valid, real word (e.g., "argue", "university") |
| **Context-Aware?** | No. "studies" (noun) and "studies" (verb) are treated identically. | Yes. Requires Part-of-Speech (POS) tag for accuracy. |
| **Speed** | Fast, computationally cheap | Slow, computationally expensive |
| **Accuracy** | Low. Can be inaccurate (e.g., "Caring" -> "Car") | High. Linguistically precise. |
| **Example: "studies"** | `studi` | `study` |
| **Example: "better"** | `better` (no rule applies) | `good` |
| **Use Cases** | Search Engines, Information Retrieval, large-scale tasks | Chatbots, Machine Translation, Semantic Analysis |

# Modern Tokenization Strategies

**Byte-Pair Encoding (BPE) - Pre-tokenizers and Morphology**

**Stage 1: Pre-Tokenization** -Before the main subword algorithm (like BPE) is applied, the raw text is first passed through a **pre-tokenizer**. This component's job is to perform a simple, fast, rule-based split on the text to break it into "words". This pre-tokenization step is often based on splitting by whitespace and punctuation.

For example, a pre-tokenizer might take the string "Call 911!" and split it into a list of "words" and their offsets: [("Call", (0, 4)), ("911", (5, 8)), ("!", ( 8, 9))] (Note: some pre-tokenizers may split digits further, e.g., [("9", (5, 6)), ("1", (6, 7)), ("1", (7, 8))] ). The pre-tokenizer's role is critical: it defines the "barriers" or "boundaries" that the main tokenizer is *not allowed to cross*

**Stage 2: The Role of Morphology –**

**Morphology is the branch of linguistics that studies the *internal structure of words*. It is concerned** with morphemes, which are the smallest meaningful units of language. For example, the word "unhappiness" is not a single, indivisible unit. It is composed of three morphemes:

- un- (a prefix meaning "not")
- happy (the root word)
- -ness (a suffix that turns an adjective into a noun)

***Byte-Pair Encoding (BPE) - - -*** *>> BPE is a statistical tokenizer. It doesn't know what a "prefix" or a "root" is. It simply learns from the data that "un-", "happy", and "-ness" are highly frequent, re-usable character sequences, and it statistically learns to treat them as independent tokens.*

Algorithm that forms the basis for tokenizers used in many state-of-the-art models, including the GPT family and BERT.

The BPE algorithm has two distinct phases, as correctly identified in the study topics: the **Token Learner** (the training phase) and the **Token Segmenter** (the inference phase).

**Part 1: The Token Learner**

The **Token Learner** is the **training** phase. Its purpose is to take a large, raw text corpus and *build* two critical assets:

1. A final **vocabulary** of subword tokens.
2. A list of **"merge rules"** in priority order.

**Step-by-Step Algorithm:**

1. **Initialize:** The process begins with a "base vocabulary" consisting of every individual character present in the training corpus (e.g., l, o, w, e, r, n, s, t, i, d, _).
2. **Pre-tokenize & Format:** The training corpus is pre-tokenized into words (as described in 2.1). A special end-of-word symbol, such as _ or </w>, is appended to each word. This is a crucial step that allows the algorithm to distinguish between a character sequence *inside* a word (like "er") and one at the *end* of a word (like "er_"). The words are then split into their constituent characters.
    - Example Corpus : (low: 5), (lower: 2), (newest: 6), (widest: 3)
    - **Formatted Corpus:** (l o w _: 5), (l o w e r _: 2), (n e w e s t _: 6), (w i d e s t _: 3)
3. **Count Pairs:** The algorithm counts the frequency of every *adjacent pair* of symbols in the entire corpus.
4. **Merge:** The *most frequent* pair is identified (e.g., e and s appear 6 + 3 = 9 times) and is merged into a *new single symbol* (e.g., es).
5. **Update:** This new symbol (es) is added to the vocabulary , and this merge (e + s -> es) is recorded as the first merge rule. The corpus is updated by replacing all instances of "e" "s" with "es".
6. **Iterate:** Steps 3, 4, and 5 are repeated for a pre-defined number of merges (e.g., 10,000 times). This number of merges determines the final vocabulary size.

| Iteration | Most Frequent Pair | Count | New Symbol | Vocabulary Adds | Corpus Becomes... |
|---|---|---|---|---|---|
| 0 | - | - | - | *(base vocab)* | (l o w _: 5), (l o w e r _: 2), (n e w e s t _: 6), (w i d e s t _: 3) |
| 1 | e s | 9 (from newest, widest) | es | es | (l o w _: 5), (l o w e r _: 2), (n e w es t _: 6), (w i d es t _: 3) |
| 2 | es t | 9 (from newest, widest) | est | est | (l o w _: 5), (l o w e r _: 2), (n e w est _: 6), (w i d est _: 3) |
| 3 | est _ | 9 (from newest, widest) | est_ | est_ | (l o w _: 5), (l o w e r _: 2), (n e w est_: 6), (w i d est_: 3) |
| 4 | l o | 7 (from low, lower) | lo | lo | (lo w _: 5), (lo w e r _: 2), (n e w est_: 6), (w i d est_: 3) |
| 5 | lo w | 7 (from low, lower) | low | low | (low _: 5), (low e r _: 2), (n e w est_: 6), (w i d est_: 3) |
| 6 | n e | 6 (from newest) | ne | ne | (low _: 5), (low e r _: 2), (ne w est_: 6), (w i d est_: 3) |
| ... | ... | ... | ... | ... | *...and so on, until the merge limit is reached.* |

**Part 2: The Token Segmenter**

1. Take a new word to be tokenized, for example, the OOV word "lowest".
2. Pre-tokenize it using the same rules as the learner (add _): l o w e s t _.
3. Greedily apply the *merge rules* that were learned, in the *exact priority order* they were learned in.
4. **Example with "lowest_":**
    - Split into characters: l o w e s t _
    - Apply Learner's Merge Rules (from Table 2):
        - Rule 1 (e + s -> es): Yes. Word becomes: l o w es t _
        - Rule 2 (es + t -> est): Yes. Word becomes: l o w est _
        - Rule 3 (est + _ -> est_): Yes. Word becomes: l o w est_
        - Rule 4 (l + o -> lo): Yes. Word becomes: lo w est_
        - Rule 5 (lo + w -> low): Yes. Word becomes: low est_
        - Rule 6 (n + e -> ne): No.
    - No more learned merges can be applied.
5. The final tokenization is the list of remaining symbols: ["low", "est_"].

When the segmenter encounters a true OOV word that it has no relevant merge rules for (e.g., a random string like "FLIBBER"), it will simply fail to apply any merges. In this "worst-case" scenario, the tokenization "degrades gracefully" and the word is simply broken down into the individual characters it started with: ``.

# Statistical Language Modeling with N-Grams

A language model (LM) is a system that understands the statistical properties of a language. The most foundational type, which provides the theoretical basis for many advanced concepts, is the **N-Gram model**.

A **language model** is a probabilistic model that computes the probability of a sequence of words, $P(W)$.[48] It answers the fundamental question: "How likely is the sentence 'the black cat' versus 'cat black the' in English?" By doing so, it captures the underlying statistical patterns of a language.

The building block for the simplest and most traditional language models is the **N-Gram**. An N-Gram is a **contiguous sequence of N items** (e.g., words or characters) extracted from a text.[49] The "N" simply refers to the length of the sequence.

- **Example Text:** "the black cat sat on the mat"
- Unigrams (N=1): These are the individual words.
  ["the", "black", "cat", "sat", "on", "the", "mat"]
- Bigrams (N=2): These are all pairs of consecutive words.
  ["the black", "black cat", "cat sat", "sat on", "on the", "the mat"]
- Trigrams (N=3): These are all triplets of consecutive words.
  ["the black cat", "black cat sat", "cat sat on", "sat on the", "on the mat"]

**The Probabilistic Mechanics of N-Grams**

The ultimate goal of a language model is to calculate the joint probability of an entire sentence, $P(W)$, or $P(w_1, w_2, ..., w_n)$.

**The "Proper" Way: The Chain Rule of Probability** From probability theory, the **Chain Rule** provides an exact way to decompose this joint probability into a product of conditional probabilities. The probability of a sequence $P(A, B, C)$ is $P(A) \times P(B|A) \times P(C|A, B)$. For a sentence like "its water is so," the *true* probability is:

$$P(\text{"its water is so"}) = P(\text{"its"}) \times P(\text{"water"}|\text{"its"}) \times P(\text{"is"}|\text{"its water"}) \times P(\text{"so"}|\text{"its water is"})$$

**The Problem: Sparsity** This formula is mathematically correct but computationally impossible. To calculate the final term, $P(\text{"so"}|\text{"its water is"})$, we would need to count how many times we've seen the exact context "its water is" in our training data and then see how often "so" followed it. This context is a "long unique sequence". In any real-world corpus, we will *never* have seen this exact three-word history, or we will have seen it so rarely that we cannot derive a reliable probability. This is known as the **curse of dimensionality**, and it leads to an extreme **data sparsity** problem.

**The "Practical" Way: The Markov Assumption** To make this problem tractable, N-gram models make a powerful and simplifying assumption: the **Markov Assumption**. This assumption states that the probability of the next word depends *only* on a fixed, short history of the *previous N-1 words*, and not the entire history.

- **Bigram (N=2) Assumption:** We assume the probability of a word depends only on the *one* preceding word. We approximate the true probability $P(w_i|w_1...w_{i-1})$ with $P(w_i|w_{i-1})$.
  - $P(\text{"so"}|\text{"its water is"}) \approx P(\text{"so"}|\text{"is"})$
- **Trigram (N=3) Assumption:** We assume the probability depends only on the *two* preceding words.
  - $P(\text{"so"}|\text{"its water is"}) \approx P(\text{"so"}|\text{"water is"})$

**Calculating Probabilities (Maximum Likelihood Estimation - MLE)** Once this assumption is made, calculating these probabilities becomes a simple task of "counting and dividing" from our training corpus. This "count and divide" method is called **Maximum Likelihood Estimation (MLE)**.

For a bigram model, the formula is:

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

For example, to find the probability of "cat" following "black":

$$P(\text{"cat"}|\text{"black"}) = \frac{Count(\text{"black cat"})}{Count(\text{"black"})}$$

If "black cat" appears 100 times and "black" appears 1000 times in the corpus, the probability is $100/1000 = 0.1$.

**The Inevitable Problem: Sparsity in N-Gram Models**

The Markov assumption *reduces* the sparsity problem, but it does not *eliminate* it. Even with a bigram model, we will still encounter N-grams that *never* appeared in our training data.

Consider a (perfectly valid) new phrase, "black catapult." If our training corpus, however large, never happened to contain this exact bigram, our model would calculate the following:

- $Count(\text{"black catapult"}) = 0$
- $Count(\text{"black"}) = 1000$ (as before)
- $P(\text{"catapult"}|\text{"black"}) = \frac{0}{1000} = 0$

The Markov assumption *reduces* the sparsity problem, but it does not *eliminate* it. Even with a bigram model, we will still encounter N-grams that *never* appeared in our training data.

Consider a (perfectly valid) new phrase, "black catapult." If our training corpus, however large, never happened to contain this exact bigram, our model would calculate the following:

- $Count(\text{"black catapult"}) = 0$
- $Count(\text{"black"}) = 1000$ (as before)
- $P(\text{"catapult"}|\text{"black"}) = \frac{0}{1000} = 0$

This leads to the **Zero-Probability Catastrophe**. The model now believes that the phrase "black catapult" is *linguistically impossible*. This is a disaster because of the chain rule. If we try to calculate the probability of a sentence containing this phrase, such as "The black catapult...", the entire calculation will be:

$$P(\text{"The black catapult"}) = P(\text{"The"}) \times P(\text{"black"}|\text{"The"}) \times P(\text{"catapult"}|\text{"black"})$$
$$P(...) = (0.01) \times (0.05) \times (0)$$
$$P(...) = 0$$

Because *one* N-gram in the sequence has a probability of zero, the *entire sentence* is assigned a probability of zero. A language model that assigns zero probability to valid sentences is fundamentally broken and unusable.

**The Solution: Smoothing**

The solution to the zero-probability problem is **smoothing**, also known as "discounting". Smoothing is a family of statistical techniques that adjust the MLE probabilities to ensure that *no* N-gram is ever assigned a probability of exactly zero.

The core idea is to "shave off a little bit of probability mass" from the N-grams we *did* see (the "rich" events like "black cat") and *redistribute* that tiny, "stolen" amount of probability to all the N-grams we *didn't* see (the "poor" events like "black catapult").

**Laplace (Add-One) Smoothing** The simplest and most illustrative smoothing technique is **Laplace (Add-One) Smoothing**. The name describes the method perfectly: it "adds one" to every count, pretending we have seen every possible N-gram one extra time.

The formulas are adjusted as follows:

The formulas are adjusted as follows:

- **Original MLE:** $P(w_i|w_{i-1}) = \frac{Count(w_{i-1},w_i)}{Count(w_{i-1})}$
- **Laplace (Add-One):** $P_{Laplace}(w_i|w_{i-1}) = \frac{Count(w_{i-1},w_i)+1}{Count(w_{i-1})+V}$

While simple and effective at removing zeros, Laplace smoothing is actually a poor method in practice. For a large vocabulary (e.g., V = 50,000), it "moves too much probability mass". It steals a huge amount of probability from common, observed events just to give a tiny-but-non-zero probability to 50,000 unseen events. This often makes the model's predictions worse. For this reason, more advanced (but complex) techniques like Kneser-Ney smoothing are preferred in practice, though Add-One remains the foundational concept.

# Model Evaluation and Generation

After building and smoothing a language model, two questions remain:

1. How do we know if the model is any good? (Evaluation)
2. How do we use it to create new text? (Generation)

**Perplexity (PPL)** is the standard, intrinsic metric for evaluating the quality of a language model.

**Intuitive Explanation:** Perplexity is a measure of the model's **"uncertainty," "surprise," or "confusion"** when it is asked to predict a (previously unseen) test sentence. A good model should not be "perplexed" when it sees a well-formed sentence; it should find it probable and predictable.

- **The Rule: A lower perplexity is better**.
- **What the Number Means:** A perplexity score of $K$ can be interpreted as the model being, on average, as "confused" at each step as if it had to choose uniformly and randomly from $K$ possible options.
  - **Bad Model (e.g., PPL = 100):** The model is highly uncertain. At every word, it's as if it's choosing from 100 different possibilities.
  - **Good Model (e.g., PPL = 10):** The model is much more confident. At every word, it has effectively narrowed its choices down to only 10.

**Mathematical Intuition:** Perplexity is mathematically defined as the *inverse probability* of the test set, normalized by the number of words. This normalization is critical, as it prevents longer sentences from being unfairly penalized (since their total probability, a product of many fractions, will always be lower).

The most important mathematical relationship, especially for deep learning, is that perplexity is the exponentiated **cross-entropy** of the model:

$$PPL(W) = 2^{H(W)}$$

where $H(W)$ is the cross-entropy.

This mathematical link to cross-entropy is particularly relevant for students of deep learning. When a modern neural language model (like a Transformer) is trained, it is trained to minimize a **loss function**. That loss function is almost universally **cross-entropy**. Therefore, the act of training a deep learning model (minimizing cross-entropy loss) is *mathematically identical* to the act of *minimizing the model's perplexity*. This concept forms a direct bridge between the statistical N-gram models of this section and the advanced deep learning models of future study.

# WEEK 3: Word representation in NLP.

**The Bag-of-Words (BoW)** -
model is the simplest and most foundational feature extraction technique. It is built on a powerful simplifying assumption: text is modeled as an "unordered collection (a 'bag') of words".

**Corpus:**

- **D1:** "Welcome to Great Learning, Now start learning"
- **D2:** "Learning is a good practice"

**Step 1: Preprocessing (Lowercase, Punctuation/Stopword Removal)**

- **D1:** "welcome great learning now start learning"
- **D2:** "learning good practice"

**Step 2: Vocabulary Creation** The model scans the preprocessed text to build a vocabulary of all unique words.

- **Vocabulary:** {welcome, great, learning, now, start, good, practice}
- The vocabulary has 7 unique words, so our vectors will be 7-dimensional.

**Step 3: Vectorization (Scoring by Frequency)** Each document is converted into a 7-dimensional vector by counting the occurrences of each vocabulary word.

| Document | welcome | great | learning | now | start | good | practice |
|---|---|---|---|---|---|---|---|
| D1 | 1 | 1 | 2 | 1 | 1 | 0 | 0 |
| D2 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |

PROBLEMS

1. Loss of Syntax and Order
2. Loss of Semantics and Context
3. Sparsity and High-Dimensionality:

**TF-IDF**

It is a weighting scheme that "goes a step further" than BoW by replacing raw word counts with a score that reflects a word's *importance* or *saliency* within a document relative to the entire corpus.

The TF-IDF score for a term is the product of two statistics :

The TF-IDF score for a term is the product of two statistics :

1. **Term Frequency (TF):** Measures how frequently a term $t$ appears *in a specific document* $d$. This is often normalized to prevent a bias towards longer documents.
   - $TF(t, d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$
2. **Inverse Document Frequency (IDF):** Measures the term's *rarity* across the *entire corpus* $D$. This is the "logarithmically scaled inverse fraction of the documents that contain the word".
   - $IDF(t, D) = \log\left(\frac{N}{n_t}\right)$
   - Where $N$ is the total number of documents in the corpus, and $n_t$ is the number of documents containing the term $t$.

The final score is $TF - IDF(t, d, D) = TF(t, d) \times IDF(t, D)$.

**D1: “He played video games”**

Token count = 4 terms

(You wrote 6, but there are only **4 words**)

**D2: “She played him the videos”**

Token count = 5 terms

So $N = 2$ documents

Unique words across both documents:

- He , played , video, games, she, him, the, videos

Total = **8 unique terms**

$$IDF(t) = \log\left(\frac{2}{n_t}\right)$$

**TF(t, D1)**

"He played video games" → 4 words

- he = 1/4 = **0.25**
- played = 1/4 = **0.25**
- video = 1/4 = **0.25**
- games = 1/4 = **0.25**
- she = 0
- him = 0
- the = 0
- videos = 0

**TF(t, D2)**

"She played him the videos" → 5 words

- she = 1/5 = **0.20**
- played = 1/5 = **0.20**
- him = 1/5 = **0.20**
- the = 1/5 = **0.20**
- videos = 1/5 = **0.20**
- he = 0
- video = 0
- games = 0

Compute doc counts:

| Term | Appears In | n_t | IDF |
|---|---|---|---|
| he | D1 | 1 | log(2/1)= **0.301** |
| she | D2 | 1 | **0.301** |
| played | D1, D2 | 2 | log(2/2)= **0** |
| video | D1 | 1 | **0.301** |
| videos | D2 | 1 | **0.301** |
| games | D1 | 1 | **0.301** |
| him | D2 | 1 | **0.301** |
| the | D2 | 1 | **0.301** |

**For D1:**

| Term | TF | IDF | TF-IDF |
|---|---|---|---|
| he | 0.25 | 0.301 | **0.0753** |
| played | 0.25 | 0 | **0** |
| video | 0.25 | 0.301 | **0.0753** |
| games | 0.25 | 0.301 | **0.0753** |
| she | 0 | – | 0 |
| him | 0 | – | 0 |
| the | 0 | – | 0 |
| videos | 0 | – | 0 |

**For D2:**

| Term | TF | IDF | TF-IDF |
|---|---|---|---|
| she | 0.20 | 0.301 | **0.0602** |
| played | 0.20 | 0 | **0** |
| him | 0.20 | 0.301 | **0.0602** |
| the | 0.20 | 0.301 | **0.0602** |
| videos | 0.20 | 0.301 | **0.0602** |
| he | 0 | – | 0 |
| video | 0 | – | 0 |
| games | 0 | – | 0 |

<u>PROBLEMS SOLVED</u>

Moving from Frequency to Importance

The primary problem solved by TF-IDF is the major weakness of BoW, where "meaningless words like 'a' and 'the' can gain too much influence" simply because they are frequent.

The IDF component "filter[s] out common terms" by assigning them a low score. A term that appears in *all* documents (e.g., "the") would have an $n_t = N$, resulting in an $IDF = \log(N/N) = \log(1) = 0$. Thus, its final TF-IDF score is 0, effectively eliminating its influence.

**PROBLEMS**

TF-IDF is, at its core, still a Bag-of-Words model. It inherits all of BoW's other critical limitations.

- It "does not capture position in text, semantics, co-occurrences".
- It "computes document similarity directly in the word-count space".
- The vectors are still sparse, high-dimensional, and fail to capture any semantic relationships between words. "cat" and "dog" are no more similar to each other than they are to "automobile."

### Word2Vec Dense Embeddings

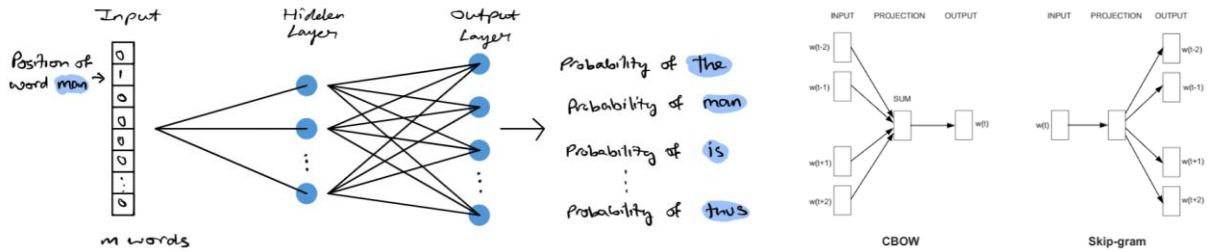
The core idea is no longer based on document-level co-occurrence, but on local context: "words which appear in similar contexts are mapped to vectors which are nearby" in the vector space. This approach allows the model to capture "precise syntactic and semantic word relationships", famously enabling vector arithmetic like `vector('King') - vector('Man') + vector('Woman')` ≈ `vector('Queen')`.

The *real* product of Word2Vec is **not** the output of the network (the predictions). The *real* product is the **hidden layer weight matrix**.
**"Predict the target word using the surrounding context words."**
It is literally a **fill-in-the-blank model**.

1. If your sentence is:
   "India wins the **world** cup next year."
   Then the context around **world** is:
   ["India", "wins", "cup", "next"]
   CBOW takes these four words as **input** and tries to guess the missing word (**world**) as **output**.
2. Input and Output
   **Input = context words**
   Example window size = 2
   So for each target word, the context = words on the left + right:
   [word-2, word-1, word+1, word+2]
   **Output = target word**
   The network predicts the **one** missing word in the center.
3. How Inputs Are Combined
   CBOW does **not** treat context words sequentially.
   It treats them as a "bag" → order does not matter.
   **Step 1: Convert each context word to its vector**
   Example: India → vector(India)
   wins → vector(wins)
   ...
   **Step 2: Combine them using average or sum**
   CBOW often uses **mean**:

$$v_{\text{context}} = \frac{v_{w-2} + v_{w-1} + v_{w+1} + v_{w+2}}{4}$$

   This creates **one input vector** representing all context words.
4. Hidden Layer
   This averaged context vector is passed into the hidden layer (usually linear).
   Hidden layer size = embedding dimension (e.g., 300).
   No activation (identity function).
   Weights are the **embeddings** we want to learn.
5. Output Layer
   The network predicts a probability distribution over the **entire vocabulary**.
   Softmax is used to choose the **most probable target word**.




The **problem is the denominator**, the normalization term.[36] This term requires calculating the dot product score for the target word against *every single other word in the vocabulary (V)*, just to calculate the probability for *one* training pair.
If the vocabulary size $V$ is 1,000,000, this means 1,000,000 calculations are required for *every single training pair* (e.g., for ["India", "wins", "cup", "next"]). The computational complexity is $O(V)$ per step.[36] This is not just "expensive"; it is "intractable" and makes training on large vocabularies impossible.[39]

CBOW outputs **ONE target word** from a vocabulary of maybe **1,000,000 words**.

Standard softmax requires computing:

$$P(\text{world}) = \frac{e^{W_{world}}}{\sum_{i=1}^{V} e^{W_i}}$$

This denominator sums over **all V words**.

If V = 1M → **1 million computations per prediction** → too slow.

## Negative Sampling-

The original Skip-Gram objective tries to predict the correct context word out of the entire vocabulary. That requires computing a softmax over all vocabulary words, which is expensive when the vocabulary is large.
Negative Sampling avoids this by **reframing the task**.
Instead of asking: - **"Which one of the 1,000,000 words is the correct context word?"**
it asks a much simpler question: -**"Is this pair (target, context) real or fake?"**
This turns the problem into **binary classification**. – from softmax to sigmoid.

**1. Take a real word pair (positive sample)**
From the sentence:
India wins the world cup next year
Assume your context window gives you a real pair:
(target = world, context = cup)
This is a **positive sample**.
You want the model to output:
Real pair → label = 1

---

**2. Generate k negative samples**
You now create "fake" pairs by keeping the same target word but pairing it with random words from the vocabulary.
For example, if k = 5:
(world, cup) → real
(world, apple) → fake
(world, school) → fake
(world, train) → fake
(world, banana) → fake
(world, system) → fake
Each fake pair gets label = 0.
These negative words are sampled using a special probability distribution (typically proportional to word frequency raised to the power 0.75).

---

**3. Train the model on these 1 + k pairs**
You update the model's weights only for:

- the target word
- the context word
- the k sampled negative words

**Why This Works**
Because Skip-Gram embeddings learn **which words occur together**.
By telling the model:

- "world" and "cup" occur together → strengthen similarity
- "world" and "banana" do not occur together → weaken similarity

the embeddings gradually converge so that:

- true context words are close in vector space
- random words are far away

This produces high-quality embeddings without ever computing a softmax.

## Hierarchical Softmax (HS)

Hierarchical Softmax is an optimization used in Word2Vec to make prediction faster by **replacing the full softmax with a binary tree**.
In normal softmax, predicting the target word requires computing a probability for **all words in the vocabulary**, which is too slow.
Hierarchical Softmax solves this by:

- building a **binary Huffman tree**
- assigning each word to a **leaf node**
- predicting a **path** from the root to a leaf instead of choosing 1 word out of V

**Word2Vec's Unsolved Issues**

Despite its revolutionary success, Word2Vec has two critical, unsolved issues.

1. **Out-of-Vocabulary (OOV) Words:** Word2Vec learns one vector for each word *in its training vocabulary*. If a word (e.g., a new name, a typo, a new piece of slang) is not in that vocabulary, the model "cannot provide embeddings" for it. This is a critical failure for real-world applications.
2. **Morphological Ignorance:** The model treats words as "atomic units". This means the words "run," "running," "ran," and "runner" are seen as four *completely unrelated* items. The model learns a separate vector for each, failing to capture the obvious linguistic and semantic link between them.
3. **The Polysemy Problem (Persists):** The original BoW problem of "bat" (animal) vs. "bat" (sport) *is still not solved*. Word2Vec learns only *one* static vector for the word "bat," which is effectively an *average* of all the contexts it appears in (both animal and sport).

## FastText and Subword Embeddings

FastText (by Facebook AI) is an extension of Word2Vec that improves embeddings by representing **each word as a combination of character-level n-grams**.This makes FastText far better at handling rare words and OOV (out-of-vocabulary) words.

**1. Core Idea - -** Word2Vec treats a word as an atomic symbol:

- "playing"
- "played"
- "player"

All three are unrelated unless the model learns them from context.

FastText breaks each word into **subword units** (character n-grams), for example:

For word: "playing"

With **n-gram size 3 to 6, FastText produces:**

- "<pl", "pla", "lay", "ayi", "yin", "ing", "ng>" (3-grams)
- "play", "layi", "ying" (4-grams)

FastText also includes the whole word as a feature.

This means:

word_vector("playing") = sum( vectors of all its n-grams )

**2. How It Trains (Skip-Gram or CBOW)**

FastText uses the same architecture as Word2Vec:

- **CBOW:** predict target word from surrounding context and **Skip-Gram:** predict context words from target word

But instead of using the word vector, it uses the **sum of n-gram vectors**.

Example (Skip-Gram): Target word: "playing" Context word: "football"

Before predicting, FastText does:

h = sum(embeddings of n-grams of "playing")

This h is used to predict the context word "football". - - - **** --- Training updates the embeddings of the n-grams, not just the word.

**3. Why This Helps**

**a) Morphology is learned**

Words like:

- play
- playing
- played
- player

share many n-grams:

- pla
- play
- lay

This makes their vectors automatically similar.

**b) Handles OOV (Out-of-Vocabulary) words**

If a new word appears, e.g.:

"playfulness"

Even if FastText has never seen this word during training, it can still compute its embedding because it has seen many of its n-grams:

- play
- lay
- ful
- ness
- etc.

Word2Vec **fails** here (OOV problem).

---

**c) Better for rare words**

Even if a rare word appears once, its n-grams appear in many other words, so its vector becomes stable.

---

**4. Prediction Architecture (Detailed)**

**Step 1: Break the input word into n-grams**

Using n=3 to 6 by default.

**Step 2: Get embeddings for all n-grams**

These are trainable vectors.

**Step 3: Combine them (usually sum or average)**

v(word) = Σ(embeddings of n-grams) + embedding(word)

**Step 4: Use this in a Skip-Gram or CBOW objective**

Loss is optimized using:

- hierarchical softmax, or
- negative sampling (default)

---

**5. Example: "India wins the world cup next year"**

Take the word “India”.

Its 3-grams:

- <In
- Ind
- ndi
- dia
- ia>

FastText vector for "India" = sum of vectors of these n-grams.

When training:

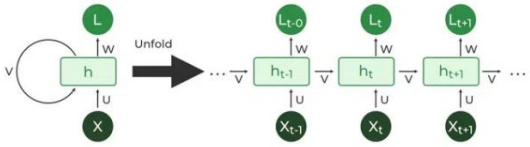
Context: "wins", "the", "world", "cup"

FastText uses this composite vector of "India" to predict context words.

This leads to better representations for:

- names
- countries
- rare words

# WEEK 4: Recurring Neural networks – RNN & LSTMS




### - - - RNN

A **Recurrent Neural Network (RNN)** is a neural architecture designed for **sequential data**, where the output at time *t* depends on both:

1. **Current input**
2. **Hidden state (memory) from previous time steps**

Formally, at each time step *t*:

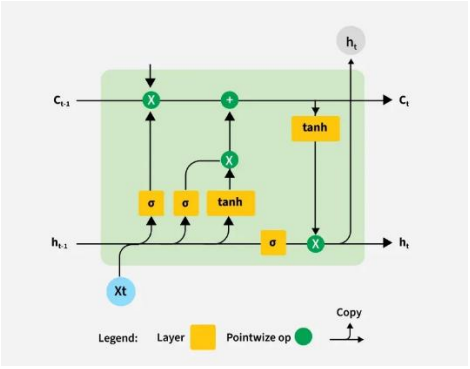
$$h_t = f(W_x x_t + W_h h_{t-1} + b)$$

Where:

- $x_t$ = input at time *t*
- $h_{t-1}$ = previous hidden state (memory)
- $W_x, W_h$ = learnable weights
- $f$ = activation function (usually tanh or ReLU)

The hidden state $h_t$ carries information through the sequence.

Training an RNN involves a variation of backpropagation known as **Backpropagation Through Time (BPTT)**. Conceptually, the RNN is "unrolled" across time steps, creating a deep feedforward network with T layers, where T is the sequence length.[1] The error calculated at the output is propagated backward through time to update the weights.
However, standard RNNs suffer profoundly from the **vanishing gradient problem**. Because the gradient is multiplied by the weight matrix W-hh at each time step during backward propagation, if the eigenvalues of W_{hh} are less than 1, the gradient decays exponentially as it travels back in time. It makes model un-convergable




## - - - LSTM

An **LSTM is an advanced RNN** architecture designed to handle **long-term dependencies** by using a special internal structure called **gates**.
**The LSTM cell has 3 gates:**

1. **Forget Gate ($f_t$):** Decides what past information to erase.
2. **Input Gate ($i_t$) & Candidate Memory (\tilde{C}_t): Decides what new information to add.**
3. **Output Gate ($o_t$):** -Controls what part of the memory to output.

S**equence** in which data flows through an LSTM cell at each time step (very clear, step-by-step):

1. The first step is to decide **what information to remove** from the old cell state.

$$f_t = \sigma(W_f[x_t, h_{t-1}] + b_f)$$

This produces values between **0 (forget)** and **1 (keep)**.

2. Next, the LSTM decides what new information to add. -Two calculations happen:

**a.** Input Gate Activation

i. $i_t = \sigma(W_i[x_t, h_{t-1}] + b_i)$ — Controls how much new info to write.

**b.** Candidate Memory

i. $\tilde{C}_t = \tanh(W_C[x_t, h_{t-1}] + b_C)$ - This is the new content that could be added.

3. **Update Cell State -** The old memory and new memory combine:
   a. $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$

      i. Forget some of the old state
      ii. Add new candidate information
      iii. This is the **core memory update**.\

4. **Output Gate -** Decides what part of the memory becomes the output hidden state.
   **a.** $o_t = \sigma(W_o[x_t, h_{t-1}] + b_o)$

5. **Compute New Hidden State -Finally, the hidden state is computed:**
   a. $h_t = o_t \odot \tanh(C_t)$ -This is the **output of the LSTM** for time step t.

## The Technical Reason LSTM Prevents Vanishing Gradients

**1. Additive Updates (not multiplicative like RNNs)**

**Standard RNNs repeatedly multiply hidden states → gradients shrink.**
**LSTMs use addition:**

$$C_t = f_t C_{t-1} + \dots$$

**Additive paths prevent exponential decay of gradients.**

**2. Linear Cell State (No repeated tanh)**
**The memory cell has a linear path:**

$$C_{t-1} \rightarrow C_t$$

**This path does *not* repeatedly apply nonlinear functions (like tanh), so gradients don't get squashed.**

## Use cases of Recurring NN models- Sequence Labeling: The Many-to-Many Paradigm-

The objective is to assign a categorical label to every element (token) in the sequence based on both its internal features and its context.
Common Applications:

- Part-of-Speech (POS) Tagging: Identifying words as Nouns, Verbs, Adjectives, etc.
- Chunking: Identifying constituents like Noun Phrases (NP) or Verb Phrases (VP).
- Sentiment Analysis (Fine-grained): Assigning sentiment polarity to specific words or phrases within a sentence.

✅ **1. What is happening during training?**
You have:

- An input sequence:
  **X = ($x_1$, $x_2$, …, $x_t$)**
- A label for each word:
  **Y = ($y_1$, $y_2$, …, $y_t$)**

The model must *predict one label for each word* — like POS tags, NER tags, etc.
✅ **2. The probability idea (simplified)**
The model says:
"If I know the hidden state at time t ($h_t$), I can predict the label for word t."
So the probability of the full sequence is assumed to be:

$$P(Y \mid X) = \prod_{t=1}^{T} P(y_t \mid h_t)$$

Meaning:
**The model predicts each label independently, one by one.**
(It does NOT look at label $y_{t+1}$ when predicting $y_t$ unless you use a CRF.)

✅ **3. Loss function (very simple)**
We take the **negative log probability** of the correct label at each time step.
This is just **cross-entropy** summed over all words:

$$L = -\sum_{t=1}^{T} \log\left(P(y_t^{true})\right)$$

Think of it like this:

- If the model is confident and correct → loss is small
- If the model is wrong → loss is large

This teaches the model to increase the probability of correct labels.

✅ **4. Backpropagation (simplified)**
**Forward pass:**

- Each word goes through the BiRNN/LSTM.
- Each word gets a predicted label (like Noun, Verb, etc.)

✅ **5. Inference (prediction time)**
At test time, for each word, the model simply picks:

$$\hat{y}_t = \arg\max P(y_t \mid x)$$

"Which label has the highest probability? Pick that one."
This is called **greedy decoding**.
Because it makes decisions for each step independently.

- It doesn't consider whether the sequence "Noun Noun Noun Noun" makes sense.
- It only picks the best label **for that specific word**.

**WHY BI DIRECTIONAL RNN/LSTM ARE USED???**

**Unidirectional RNN**

The translator hears one word and must translate immediately.

- First word: “bank”
- They guess: “financial institution”
- Later hear: “river” → they were wrong
  But it's too late.

**Bidirectional RNN**

The translator magically sees the entire sentence beforehand.

They know:

- The muddy **bank** collapsed **during the flood**

So they translate correctly.

**Sequence labeling**

The translator labels each word with tags like:

- Noun
- Verb
- Adjective

With greedy decoding, they must write the tag immediately once they feel 51% confident — even if committing early makes the whole sentence sound odd later.

## Use cases of Recurring NN models - Named Entity Recognition (NER) and BIO Tagging

Named Entity Recognition (NER) is a specialized sub-class of sequence labeling focused on identifying and categorizing real-world objects such as Persons (PER), Organizations (ORG), Locations (LOC), and Time expressions.
While standard sequence labeling (like POS tagging) deals with single-word concepts, NER must handle multi-token entities. For example, "New York City" is a single logical entity spanning three tokens. If a model simply classified "New" as LOC, "York" as LOC, and "City" as LOC, it might be ambiguous whether this is one city or three separate locations listed consecutively.

**Why do we need BIO or BILOU tags?**

To convert the multi-word entity problem into a **token-by-token classification**, we use a tagging scheme.

**BIO**

- **B** = Beginning of entity
- **I** = Inside entity
- **O** = Outside entity

Example:
Sentence: "Albert Einstein visited New York."

| Token | Tag |
|---|---|
| Albert | B-PER |
| Einstein | I-PER |
| visited | O |
| New | B-LOC |
| York | I-LOC |

Without BIO, the model wouldn’t know if “New York City” is one city or three separate words.
BIO fixes entity boundaries.

**Why variants exist (BIOES / BILOU)?**

BIOES adds more detail:

- **E** = End
- **S** = Single-word entity

**Problem with simple models (BiLSTM + Softmax)?**

**"Barack Obama met Angela Merkel."**

We expect BIO tags like:

| Token | Correct Tag |
|---|---|
| Barack | B-PER |
| Obama | I-PER |
| met | O |
| Angela | B-PER |
| Merkel | I-PER |

❌ **Example of what BiLSTM + Softmax might predict (INVALID)**

**Case 1: Predicts I-PER at the start of the sentence**

| Token | Predicted |
|---|---|
| Barack | I-PER (**INVALID**) |
| Obama | I-PER |
| met | O |
| Angela | I-PER (**INVALID — should start with B-PER**) |
| Merkel | I-PER |

**Why invalid?**

BIO rule: A sequence **cannot start with I-XXX** — you must begin with B-PER.

---

**Case 2: Predicts wrong transitions**

| Token | Predicted |
|---|---|
| Barack | B-PER |
| Obama | I-PER |
| met | O |
| Angela | B-PER |
| Merkel | I-LOC (**INVALID transition**) |

**Why invalid?**

If previous tag is **B-PER**, the next must be:

- I-PER OR
- O OR
- B-xxx

But this model outputs **I-LOC**, which breaks BIO logic.

---

**Case 3: Predicts two I-PER tokens but they belong to different entities**

| Token | Predicted |
|---|---|
| Barack | I-PER (**Invalid start**) |
| Obama | I-PER |
| met | O |
| Angela | I-PER (**Invalid, should be B-PER**) |
| Merkel | I-PER |

This **looks like one long entity: "Barack Obama met Angela Merkel"**, which is completely wrong.

---

❗ **Why does this happen?**

Because **Softmax makes each decision independently**:

- It only looks at *local* evidence
- It does **not** understand sequence constraints
- It does **not** remember BIO rules
- It does **not** enforce valid transitions

So it might think:

- "Barack looks like a person → give I-PER"
- "Angela also looks like a person → give I-PER"
- "Merkel looks like a place → give I-LOC"

Each prediction is **independent**, so it can break structural rules.
This is the **constraint problem**.

CRF (Conditional Random Field) solves this problem by:

**A. Enforcing tag rules**

CRF learns which tag transitions are allowed:
Examples:

- B-PER → I-PER (allowed)
- O → I-PER (not allowed)
- B-LOC → I-LOC (allowed)

CRF "knows" these rules because it learns a **transition matrix**.

**B. Predicting the entire sequence together**

Instead of predicting each word independently:

- LSTM gives scores for each tag (emission scores)
- CRF decides the **best sequence** considering all transitions

CRF = a global decision maker.

Once the BiLSTM has produced **emission scores** (logits for each tag at each time step), the CRF layer does **not** simply pick the highest-scoring tag for each word.
Instead, it finds the **best global tag sequence** for the entire sentence.
This is done using the **Viterbi algorithm**.

## Use cases of Recurring NN models - Sequence-to-Sequence (Seq2Seq) Models

- **Machine Translation:** "How are you?" (3 words) → "Comment allez-vous?" (3 words, but different internal structure).
- **Summarization:** Input article (500 words) → Summary (50 words).
- **Dialogue Systems:** User query → Chatbot response.

**Architecture: Encoder–Decoder**

**1. Encoder (Reads input)**

- It is usually a **BiLSTM or GRU**.
- It reads the full input sequence one token at a time.
- It **does not generate any output labels**.
- It keeps updating its hidden state.
- After reading last token, the final hidden state **h_T** becomes the **Context Vector**.

You can think of this Context Vector as:
A compressed meaning of the entire input sentence.

**2. Decoder (Generates output)**

- It is another RNN (LSTM/GRU).
- It starts with the Encoder's final hidden state.
- It generates the output **one word at a time**, autoregressively:
  - Generate $y_1$
  - Use $y_1$ to generate $y_2$
  - Use $y_2$ to generate $y_3$
  - ...
- Stops at the <EOS> token.

So:
Input: 7 words
Output: maybe 5 words or 20 words.
No fixed relationship.

**Training: Teacher Forcing vs. Free Running**

**The Problem: Exposure Bias**

During training, the decoder must be given its previous output to predict the next word.
But early in training, the model's predictions are **terrible**.

**Solution: Teacher Forcing**

During training:

- At step t, instead of feeding the model's *predicted* word,
- We feed the **true target word**.

Example:

- Predicted $y_1$ = "Apple" (wrong)
- Teacher Forcing: feed ground truth "The" into Step 2

This stabilizes training and helps the model learn correct patterns.

---

**Scheduled Sampling**

Teacher Forcing causes a mismatch:

- **Training time**: inputs are perfect (ground truth)
- **Inference time**: inputs are model-generated, often wrong

Scheduled Sampling reduces Teacher Forcing gradually:

- Start: 100% teacher forcing
- Later: 50% teacher forcing, 50% model prediction
- End: mostly model prediction

This teaches the model to **recover from its own mistakes**.
de

**4.3 Inference: Beam Search**

During testing:

- We don't have ground truth
- Decoder must use its own predictions

Beam width = K

(typically 3, 5, or 10)

Beam Search keeps **top K best partial sequences** instead of 1.

Algorithm:

1. Step 1: get top K words
   e.g., "The", "A", "It"
2. Step 2: expand each → K×V candidates
3. Score them using cumulative probability
4. Keep top K
5. Repeat until all reach <EOS>

This gives:

- Longer-term planning
- Higher quality sequence
- Avoids greedy mistakes

Beam Search ≈ multiple parallel futures

We keep only the best ones.

## ATTENTION MECHANISM IN SEQ2SEQ ENCODER DECODER

Seq2Seq architecture, primarily built upon Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, encountered a critical structural limitation. As sequence lengths increased, model performance degraded precipitously. This phenomenon, identified as the "information bottleneck," arose from the architectural constraint that required the encoder to compress the entirety of a source input—regardless of its complexity or length—into a single, fixed-dimensional vector.

**2.1 The Classical Seq2Seq Architecture**

The sequence-to-sequence model is designed to estimate the conditional probability of a target sequence $Y = \{y_1, y_2, \ldots, y_{T_y}\}$ given a source sequence $X = \{x_1, x_2, \ldots, x_{T_x}\}$. In its canonical form, this architecture comprises two distinct recurrent neural networks: an encoder and a decoder.

The encoder processes the input sequence sequentially. At each time step $t$, the hidden state $h_t$ is updated based on the current input token embedding $x_t$ and the previous hidden state $h_{t-1}$:

$$h_t = f(x_t, h_{t-1})$$

where $f$ is a non-linear activation function, typically realized by an LSTM or Gated Recurrent Unit (GRU) to mitigate the vanishing gradient problem inherent in standard RNNs.

Crucially, after the encoder processes the final token $x_{T_x}$, the resulting hidden state $h_{T_x}$ is designated as the **Context Vector** ($c$). This vector acts as the sole conduit of information between the source and target languages.

$$c = h_{T_x}$$

The decoder is then initialized, often using $c$ as its initial hidden state. It generates the output sequence autoregressively, where the probability of the next token $y_t$ is conditioned on the context vector, the decoder's current hidden state $s_t$, and the previously generated token $y_{t-1}$:

$$P(y_t|y_1, \ldots, y_{t-1}, X) = g(y_{t-1}, s_t, c)$$

**The Fixed-Width Constraint:** The dimensionality of the context vector (e.g., $\mathbb{R}^{256}$, $\mathbb{R}^{512}$, or $\mathbb{R}^{1024}$) is a hyperparameter fixed prior to training. This implies that the model must compress the semantic content of the input sequence into this static vector space, regardless of whether the input is a three-word greeting or a fifty-word legal clause. As noted in research, the encoder is forced to treat the intermediate state as a "sufficient statistic" for the entire input string.

**Compression Loss and Vanishing Gradients:** As the sequence length $T_x$ grows, the encoder struggles to preserve high-frequency information from the beginning of the sequence. By the time the recurrent network reaches the final token $x_{T_x}$, the contribution of $x_1$ to the final state $h_{T_x}$ has likely been attenuated through repeated matrix multiplications and non-linear activations. While LSTMs alleviate this via gating mechanisms, they do not eliminate the structural incapacity to store infinite information in finite space.

SEQ2SEQ with - ATTENTION (Bahdanau) —

Standard RNNs read sequences in chronological order (left to right). However, in translation, the context of a word often depends on future words (e.g., adjective-noun agreement in Romance languages). To address this, Bahdanau attention utilizes a **Bidirectional RNN (BiRNN)**.

The encoder consists of two separate RNNs:

1. **Forward RNN:** Reads the input sequence from $x_1$ to $x_{T_x}$, calculating a sequence of forward hidden states $(h_1, \dots, h_{T_x})$.
2. **Backward RNN:** Reads the sequence in reverse from $x_{T_x}$ to $x_1$, resulting in backward hidden states $(\overleftarrow{h}_1, \dots, \overleftarrow{h}_{T_x})$.

**Annotation Concatenation:** The final representation for each word $x_j$, denoted as the **annotation** $h_j$, is the concatenation of the forward and backward hidden states.

$$h_j = \begin{bmatrix} h_j \\ \overleftarrow{h}_j \end{bmatrix}$$

If the hidden units of each RNN have dimension $n$, the annotation $h_j$ has dimension $2n$. This ensures that $h_j$ contains a focused summary of the word $x_j$ along with its surrounding context from the entire sentence.

2. How relevant is each source word

   At decoding step **t**:

   - The decoder has a hidden state $\mathbf{s_{t-1}}$ → represents everything it has translated so far.
   - The encoder has produced annotations $\mathbf{h_1, h_2, \dots, h\_J}$ (one for each source word).

   For each source word $\mathbf{h_j}$, we compute a score:

   **How compatible is (the next thing I want to write) with (source word j)?**

   Mathematically:

$$e_{tj} = v_a^\top \tanh(W_a s_{t-1} + U_a h_j)$$

   **What this means in simple terms:**

   - $\mathbf{s_{t-1}}$ = what I have translated so far
   - $\mathbf{h_j}$ = meaning of source word *j*
   - Put them inside a small feedforward network
   - Output a number (score) telling how important source word **j** is right now

   This is why attention is data-driven:

   - If the decoder is about to output a verb, it may attend strongly to the verb in the source.
   - If it is about to output a noun, it attends to the corresponding noun.

   **Why this uses the "previous translations"?**

   Because $\mathbf{s_{t-1}}$ contains the entire translation history.

   So the question becomes:

   > "Given what I have already translated ($s_{t-1}$), which encoder word matters now ($h_j$)?"

   This is exactly what "alignment" means.

3. The scores $e_{tj}$ are unnormalized. They can be any real number. - To use them as weights, we convert them to probabilities:

$$\alpha_{tj} = \frac{\exp(e_{tj})}{\sum_k \exp(e_{tk})}$$

   $\alpha_{tj}$ ≈ "How much attention should I give to source word j right now?"

   All $\alpha_{tj}$ sum to **1**, so they form a probability distribution.

   Examples:

   - $\alpha_{t3} = 0.70$ → source word 3 is very important
   - $\alpha_{t1} = 0.02$ → ignore source word 1

   This is why attention is often visualized as a heatmap.

4. Now that the model knows how important each source word is ($\alpha_{tj}$), it builds the final **context vector**:

$$c_t = \sum_j \alpha_{tj} h_j$$

   The decoder creates a **weighted average** of all encoder states.

   If:

   - $\alpha_{tj}$ is high → $h_j$ influences $c_t$ strongly
   - $\alpha_{tj}$ is low → $h_j$ is almost ignored

**Why this solves the bottleneck problem**

The decoder no longer depends on only the final encoder state.

Instead:

At each decoding step, the model re-reads the entire source sentence in a smart, weighted way.

Every output step gets its own custom context vector:

- $c_1$: focus on first word(s)
- $c_2$: maybe shift focus
- $c_3$: focus on verb
- … and so on

5. **Decoding With Context (Generating the Output Token)**

Now the decoder has:

- previous state $\mathbf{s_{t-1}}$
- previous output word's embedding $\mathbf{y_{t-1}}$
- context vector $\mathbf{c_t}$

It combines them:

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

**What is happening:**

- $c_t$ tells "what to pay attention to in the input"
- $y_{t-1}$ tells "what I wrote previously"
- $s_t$ becomes the updated decoder state
- From this, the model predicts the next word:

$$P(y_t \mid y_{<t}, X) = \text{softmax}(g(y_{t-1}, s_t, c_t))$$

The model then outputs:

- the next word $\mathbf{y_t}$
- moves to the next time step (**t+1**)
- computes new alignment scores, attention weights, a new context vector, etc.

This repeats until <EOS>.

SEQ2SEQ with - ATTENTION (Luo ng Attention (Multiplicative))

Luong attention answers the same question as Bahdanau:

"Which encoder hidden states should the decoder look at to generate the next word?"

The difference is how the model scores encoder states: Luong uses a multiplicative (dot / bilinear) score instead of a small FFNN.

**Alignment model: How we score each source word**

At decoding step $t$we have:

- Decoder hidden state: $s_t$(note: Luong uses the *current* decoder state, not $s_{t-1}$as Bahdanau did in the original paper)
- Encoder annotations: $h_1, h_2, \ldots, h_{T_x}$

**Raw score (energy)** options in Luong:

1. **Dot product (simplest):**

$$e_{tj} = s_t^\top h_j$$

2. **General (learned weight matrix):**

$$e_{tj} = s_t^\top W_a h_j$$

(You can think of the general form as projecting one side before doing the dot product.)

**Simple intuition: <u>AS DOT PRODUCTS IS LOGIC TO CALCULATE THE SIMILARITIES B/W 2 VECTORS</u>**

If $s_t$and $h_j$point in similar directions (high dot product), the source word $j$is likely relevant for generating $y_t$.

<u>Bahdanau (additive) attention usually performs slightly better on small/medium models because it can learn more flexible alignments.</u>
<u>Luong (multiplicative) attention is faster, simpler, and performs better on large models due to efficiency.</u>

| | | | |
|---|---|---|---|
| Small encoder/decoder (≤ 512 dims) | Bahdanau | Real-time inference required | Luong |
| Large encoder/decoder (> 512 dims) | Luong | Best alignment accuracy (NER, MT alignment) | Bahdanau |
| | | Training speed matters | Luong |

# WEEK 5: TRANSFORMERS

A Transformer is a neural network architecture based on stacked encoder and/or decoder layers that rely entirely on self-attention and feed-forward networks—without any recurrence or convolution—to model dependencies between sequence elements.
It uses multi-head self-attention mechanisms to compute context-aware representations of tokens by weighting their pairwise interactions, and incorporates positional encodings to preserve order information.
Transformers are optimized end-to-end using gradient descent and are highly parallelizable, enabling efficient training on very large datasets for tasks such as language modeling, translation, classification, and generation.

## Self-Attention Mechanism -- Trainable weights but not FFNN

Self-attention helps every word "look at" other words and decide **how important** they are for understanding the sentence.
We need **self-attention** because earlier sequence models (RNNs, LSTMs, GRUs, CNNs) **could not capture long-range dependencies efficiently, accurately, or in parallel**.

1. To learn relationships between any two words—no matter how far apart
   RNNs read tokens one-by-one.
   If two important words are far apart:
   "The report that the auditor submitted last month was fraudulent."
   The model must remember "report" across many steps → memory fades, information decays.
   Self-attention directly links every word to every other word in one step.
   So "report" and "fraudulent" immediately influence each other.
   No memory bottleneck. No forgetting.

2. To remove dependence on sequential processing (parallelization)
   RNNs must process words one after another:
   word1 → word2 → word3 → word4 → … (slow)
   This destroys training speed.
   Self-attention processes all words simultaneously:
   word1 ↔ word2 ↔ word3 ↔ word4 ↔ … (parallel)
   This gives massive GPU parallelization, making modern LLMs possible.

3. To create contextual embeddings instead of static embeddings
   Word2Vec, GloVe give one embedding per word:
   - "bank" (financial)
   - "bank" (river)

   Same vector → ambiguous.
   Self-attention lets the embedding change depending on context:
   "I deposited money in the bank." → financial
   "He sat by the bank of the river." → water body
   Because attention helps the word pull information from the right neighbors.

4. To eliminate the RNN "bottleneck"
   In RNN-based encoder–decoder architectures:
   - the whole sentence gets compressed into a single vector
   - decoder relies on that one vector → **information loss**

   Self-attention avoids bottlenecks:
   - every word directly communicates with every other
   - the decoder can look at all encoder outputs at once

   More expressive. Less information lost.

5. learn richer patterns with Multi-Head Attention
   One attention head learns one type of relationship:

- subject–verb
- adjective–noun
- coreference
- long-term dependencies
- semantic roles

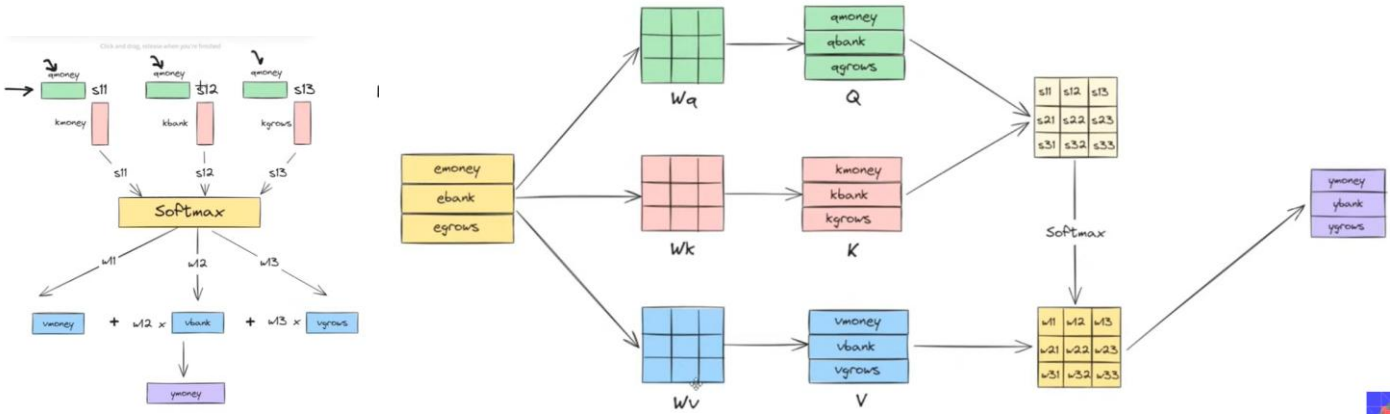
  Multi-head attention lets the model learn **many patterns in parallel**.

HOW??? – To comprehend the efficacy of the Transformer, one must first understand the limitations of static word embeddings. Traditional approaches, such as Word2Vec or GloVe, assign a fixed vector to each word in the vocabulary. In such a model, the word "bank" possesses the exact same vector representation whether it appears in the context of a "river bank" or a "financial bank." This lack of contextual flexibility limits the model's ability to resolve ambiguity. Self-attention addresses this by generating **contextual embeddings**, where the representation of a token is dynamically constructed based on its relationship to every other token in the sequence

Self-attention uses the **dot product** because it is one of the simplest and most powerful ways to measure **similarity** between two vectors. Dot-product attention lets every word ask every other word- How important are you for my meaning?

The fundamental innovation of the self-attention mechanism is the decomposition of the input vector into three distinct functional vectors: the **Query ($Q$)**, the **Key ($K$)**, and the **Value ($V$)** `[8, 9]`. These vectors are not inherent to the raw input data; rather, they are derived through learned linear transformations. For every input token embedding $x$, the model learns three distinct weight matrices—$W^Q$, $W^K$, and $W^V$—which project the input into three separate geometric subspaces `[1, 10]`.

- **The Query ($Q$): The Search Intent.** The Query vector represents the token's "needs" or "questions" regarding its surroundings. It encodes what the token is looking for within the sequence to clarify its own meaning. For instance, in the sentence "The dog plays fetch," the token "plays" acts as a query. To fully define the action, it must "search" for a subject (who is playing?) and an object (what is being played?) `[6]`. The Query vector contains the features necessary to identify these missing pieces of information.
- **The Key ($K$): The Index or Label.** The Key vector acts as the descriptor or label for a token. It serves as the matching mechanism against the Query. Returning to the library analogy, if the Query is a specific research topic (e.g., "Renaissance Art"), the Key represents the metadata, keywords, or tags on the spine of a book (e.g., "Art," "History," "1500s") `[12]`. The Key exposes the features of a token that other tokens might find relevant.
- **The Value ($V$): The Content.** The Value vector contains the actual informational content or semantic meaning of the token that will be retrieved. Once a match is established between a Query and a Key (indicating relevance), the model retrieves the information stored in the corresponding Value vector. It is the text inside the book `[5, 12]`.




The mechanism determines "relevance" through the **Dot Product** of the Query and Key vectors. Geometrically, the dot product measures the projection of one vector onto another; a high dot product indicates that the vectors are pointing in similar directions within the high-dimensional space, signifying strong alignment or compatibility `[5, 13]`.

For a specific query $q$ and a key $k$, the raw attention score is calculated as:

$$\text{Score} = q \cdot k^T$$

While the dot product provides an intuitive measure of similarity, applying it directly within a deep neural network introduces severe statistical instability, specifically regarding the variance of the resulting scores. This necessitates the defining characteristic of the Transformer's attention mechanism: **Scaling**.

The complete formula for Scaled Dot-Product Attention is:

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

Here, $d_k$ represents the dimensionality of the Key vectors. The term $\frac{1}{\sqrt{d_k}}$ is the scaling factor. To appreciate the necessity of this term, we must perform a rigorous statistical analysis of high-dimensional vector operations `[14, 15]`.

Once the attention scores are computed and normalized via the softmax function, they act as a set of weights summing to 1.0. These weights represent the relevance of each token in the sequence to the current query token. The final step of the self-attention block is the aggregation of information using these weights [5].

**Multi-Head Self-Attention: Diversity of Representation**

While the single-head scaled dot-product attention is powerful, it possesses a theoretical limitation: it must compress all types of relationships into a single probability distribution. To address this, the Transformer architecture employs **Multi-Head Attention**. Instead of performing a single attention function with one set of Q, K, V matrices, the model operates multiple attention "heads" in parallel

"Attention Is All You Need" paper, $d_{model} = 512$ and $h = 8$. This results in a dimension per head of $d_k = 64$ `[24]`.

Once all $h$ heads have computed their respective output vectors (each of dimension $d_k$), the model must combine them back into a unified representation. This is achieved through concatenation followed by a linear projection.

1. **Concatenation:** The output vectors from all heads are concatenated side-by-side to form a matrix of dimension $h \times d_k$. Since $h \times d_k = d_{model}$, the concatenated width matches the original model dimension [13].

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)$$

2. **Linear Projection ($W^O$):** The concatenated matrix is passed through a final linear weight matrix $W^O$.

$$\text{Output} = \text{Concat}(\ldots)W^O$$

The role of $W^O$ is to mix the information from the different subspaces. It aggregates the syntactic insights from Head 1, the semantic insights from Head 2, and the positional insights from Head 3 into a single, rich feature vector that is passed to the subsequent layers of the network [13, 26]. This ensures that the final embedding encapsulates a holistic view of the token's

Here is a sentence that is much harder because it contains words with **multiple meanings (polysemy)** and **long-distance connections**.

**The Complex Sentence**

**"The server crashed because the heavy traffic overloaded the port."**

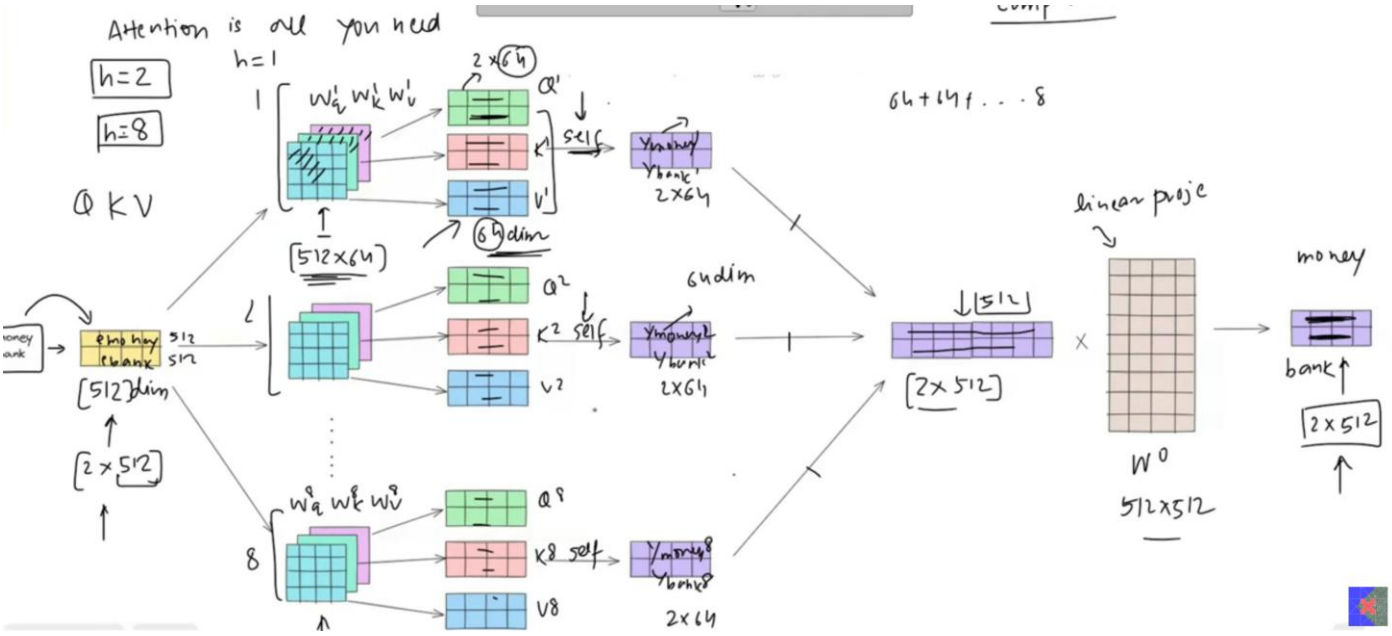
To a human, this is obviously about computers. But to a machine, every noun here is a trap because they all have double meanings:

1. **Server:** A computer? Or a waiter at a restaurant?
2. **Traffic:** Cars on a road? Or data packets?
3. **Port:** A harbor for ships? A sweet wine? Or a network connection?

The Transformer combines these insights:

1. Head 3 says "Port" definitely means **Network**, not Wine.
2. Because "Port" is Network, Head 1 updates: "Traffic" must mean **Data**, not Cars.
3. Because "Traffic" is Data, "Server" must mean **Computer**, not Waiter.

By running these heads in parallel, the model resolves the ambiguity instantly, rather than getting stuck on the first definition of "Server" it finds.




**Positional Encoding in Transformers**

### 3.1 Mathematical Formulation and the Frequency Spectrum

Instead of learning a vector for every position (which would fail to generalize beyond the training set size), Vaswani et al. proposed calculating the positional vector using sinusoidal functions of varying frequencies. For a model with embedding dimension $d_{model}$, the encoding for a position $pos$ and dimension $i$ is defined as:

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

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

inusoidal encoding is essentially a "soft," continuous version of this binary counter in the floating-point domain.[9]

- **High Frequencies (Low dimensions):** These behave like the Least Significant Bits. They oscillate rapidly, changing values significantly between position $t$ and $t+1$. This provides the model with high-resolution information, enabling it to distinguish between immediate neighbors effectively.
- **Low Frequencies (High dimensions):** These behave like the Most Significant Bits. They oscillate very slowly, ensuring that the pattern does not repeat over long distances. This effectively provides a unique "global timestamp" for the token, preventing aliasing where two distant tokens might otherwise look identical if only high frequencies were used.[10]

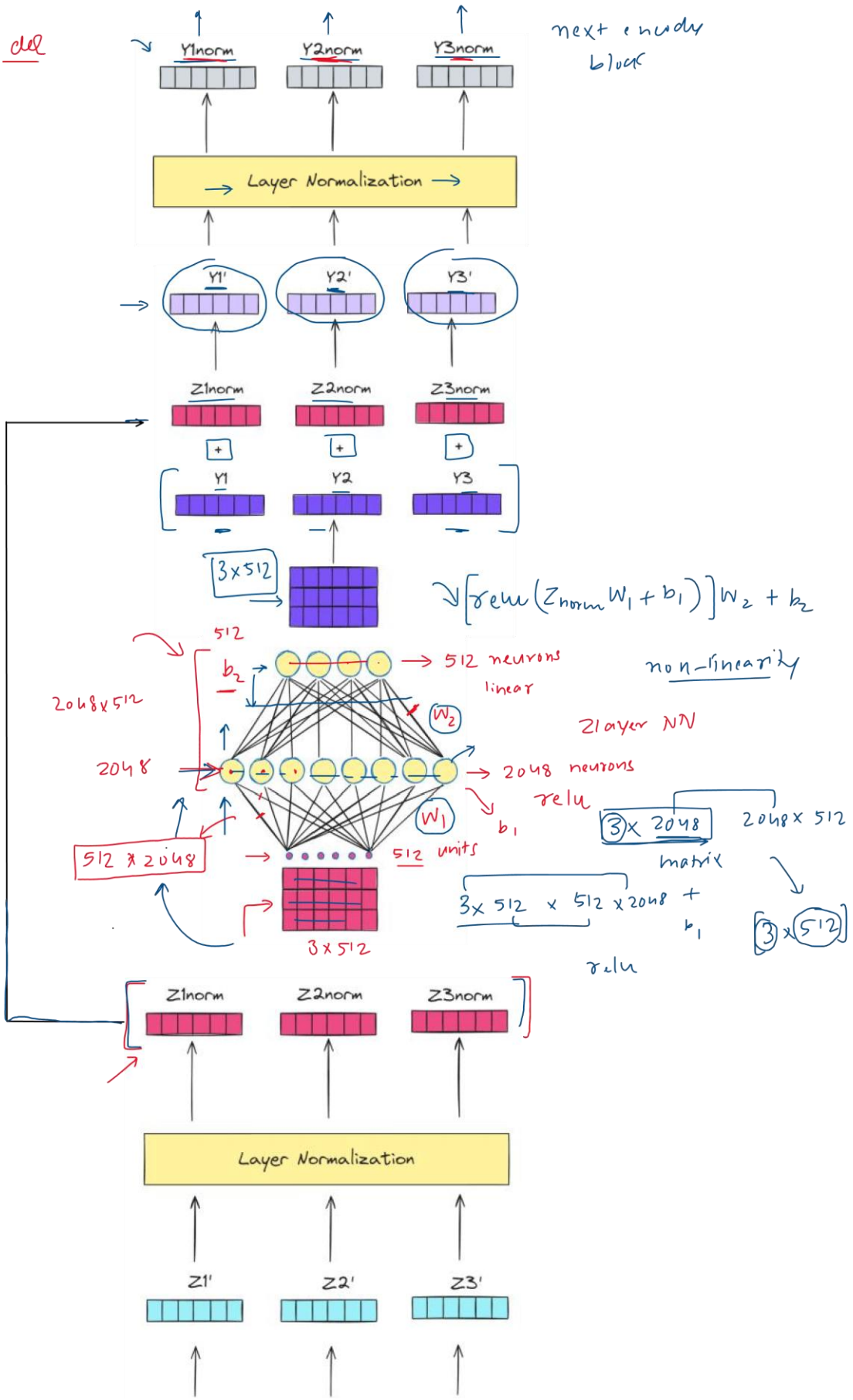
How can we combine both the semantic information (from the word embedding) and the positional information (from the positional encoding) into a single vector without "overwriting"?
Answer: When you add the embedding vector to the positional encoding vector, you're combining two different kinds of information, but the values are added element-wise (i.e., each component of the positional vector is added to the corresponding component of the embedding vector). Since this is element-wise addition, both pieces of information (semantic from embedding and positional from encoding) are blended together without losing the distinctiveness of either. The reason positional encoding doesn't overwrite the embedding's meaning is that each dimension of the Embedding vector captures some small part of the word's meaning. Positional vectors often use small magnitudes compared to the embedding vector, meaning the positional information acts like a fine adjustment to the embeddings, instead of dominating them

*The progression follows a clear trend toward **inductive biases** that better reflect the underlying structure of reality.*

1. ***Absolute (Sinusoidal/Learned):** Assumed position is a fixed coordinate. Effective, but brittle to length changes.*
2. ***Relative (T5/Shaw):** Acknowledged that distance matters more than location. Improved generalization but introduced computational overhead.*
3. ***Rotational (RoPE):** Synthesized the two using complex analysis. Encodes position absolutely (rotation) but recovers interaction relatively (dot product phase shift). This represents the current state-of-the-art, balancing efficiency with theoretical elegance.*
4. ***Bias-Based (ALiBi):** Prioritizes extrapolation by hard-coding locality, trading off flexible global attention for infinite length stability.*

**Benefits of Normalization in Deep Learning**

- Improved Training Stability:
  - Normalization helps to stabilize and accelerate the training process by reducing the likelihood of extreme values that can cause gradients to explode or vanish.
- Faster Convergence:
  - By normalizing inputs or activations, models can converge more quickly because the gradients have more consistent magnitudes. This allows for more stable updates during backpropagation.
- Mitigating Internal Covariate Shift:
  - Internal covariate shift refers to the change in the distribution of layer inputs during training. Normalization techniques, like batch normalization, help to reduce this shift, making the training process more robust.
- Regularization Effect:
  - Some normalization techniques, like batch normalization, introduce a slight regularizing effect by adding noise to the mini-batches during training. This can help to reduce overfitting.


del
Y1norm
Y2norm
Y3norm
next encoder block
Layer Normalization
Y1'
Y2'
Y3'
Z1norm
Z2norm
Z3norm
Y1
Y2
Y3
3x512
relu(Z_norm W1 + b1) W2 + b2
512
b2
512 neurons
linear
non-linearity
2048x512
W2
2 layer NN
2048
2048 neurons
relu
W1
b1
3x 2048
2048x 512
matrix
512 x 2048
512 units
3x 512 x 512 x 2048 +
b1
3x512
relu
Z1norm
Z2norm
Z3norm
Layer Normalization
Z1'
Z2'
Z3'

## Masked self-attention

the transformer's decoder is auto regressive at the inference time and non auto regressive at the training time-
During **training**, we already have the entire target sentence:
Example target:I love deep learning .
Teacher forcing means:

- At training step, the decoder sees **all previous true tokens**.

If the decoder was strictly autoregressive during training, we would feed tokens **one by one** → training would be extremely slow.
But transformers want to train with **parallelism** — process the full sequence at once.
→ We need a trick so that:

- **Model can run in parallel during training**, -- **But still behave autoregressively in its logic**.

**The Issue:** In standard Self-Attention, every word looks at *every other word*.
**The Cheat:** If we don't stop it, the word "The" will look at the word "cat" (which is sitting right next to it in the input) and just copy it. The model learns nothing about language; it just learns to copy the next word in the list.

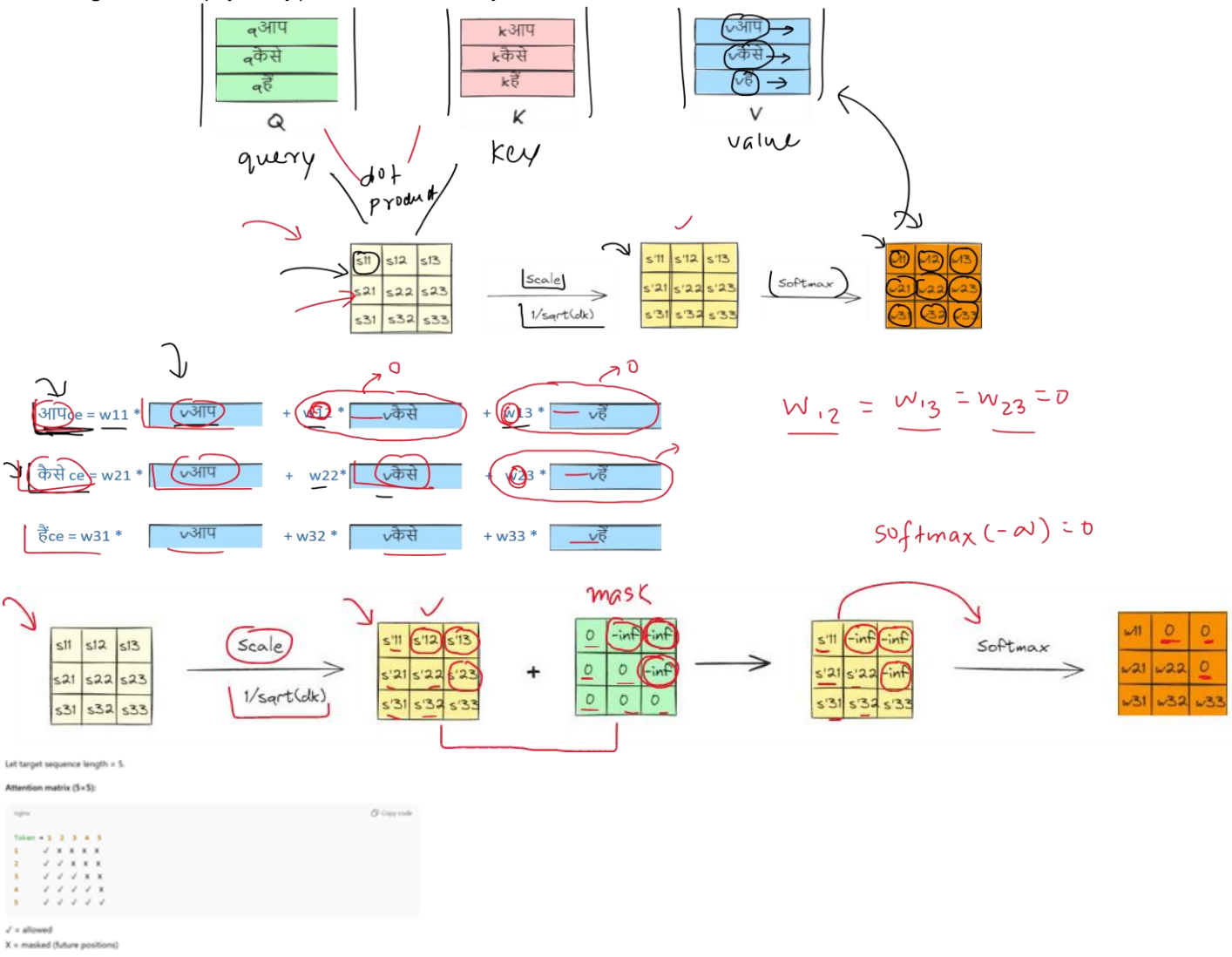
**2. The Solution: The Look-Ahead Mask**
To stop this cheating, we apply a **Mask** to the Self-Attention layer. Think of this mask as a set of "blinders" or a curtain.
The mask ensures that:

- The 1st word can only look at the 1st position.
- The 2nd word can look at the 1st and 2nd positions.
- The 3rd word can look at the 1st, 2nd, and 3rd positions.
- **No word can see the future.**

**3. How it enables Parallelism (The "Fast" Part)**
This is the clever part. Because we masked the future, we don't need to loop through the sentence one word at a time (like we do in inference).
We can feed the **entire huge matrix** of words into the GPU at once. The GPU calculates the attention scores for *every word simultaneously*.

- It calculates the prediction for "The → "cat"
- **AND** the prediction for "cat" →"sat"
- **AND** the prediction for "sat" →"on"

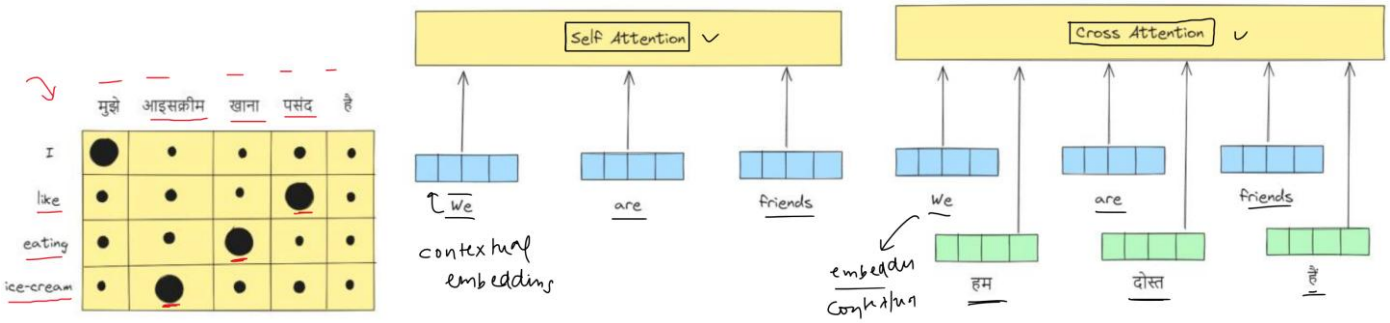
All of these happen in a single millisecond. The mask guarantees that the calculation for "The" didn't accidentally use data from "sat", even though "sat" was physically present in the memory.



## Cross-attention

Cross-attention = **Decoder tokens (Queries)-asks questions** to **Encoder outputs (Keys and Values).**
Think of it like this:

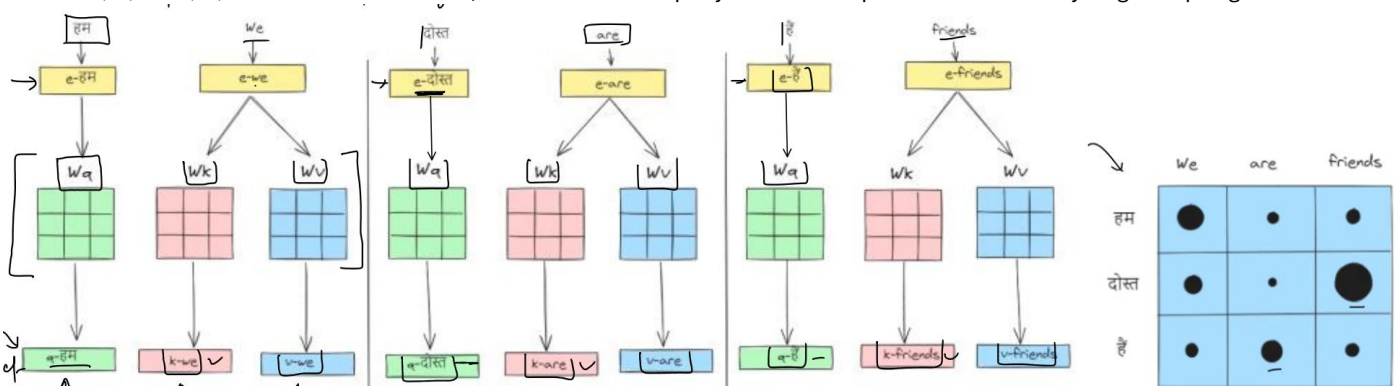
- The **encoder** has already read the entire input sentence and stored a “summary” of each word in its hidden states.
- During output generation, the **decoder** needs to look back at those encoded representations.

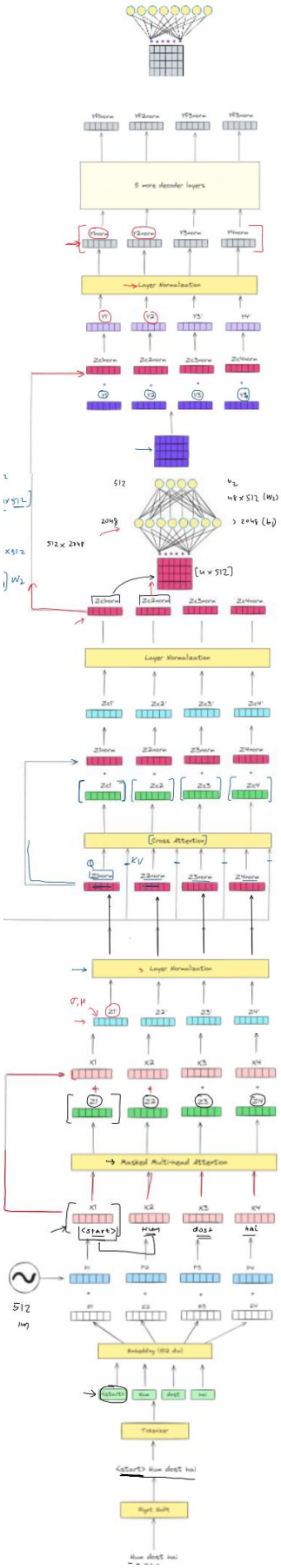


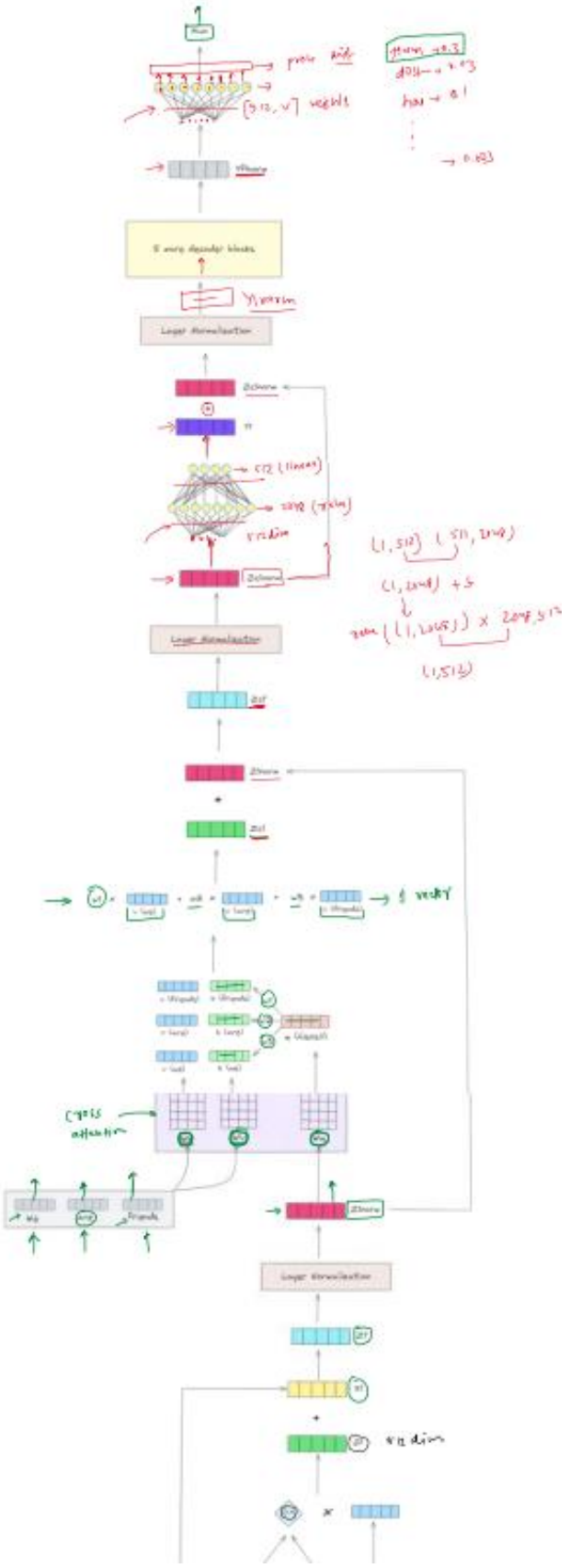

Just like Self-Attention, we use Query (Q), Key (K), and Value (V). But here, they come from different places!

- **Query (Q):** Comes from the **Decoder**.
  - *Meaning:* "I am currently generating the 3rd word. What information do I need from the source text?"
- **Key (K) & Value (V):** Come from the **Encoder**.
  - *Meaning:* These represent the original input sentence (e.g., the English sentence we are translating).

Because $K$ and $V$ come from the Encoder, the Decoder can "query" the entire input sentence at every single step of generation.




5 more decoder layers
Layer Normalization
Layer Normalization
Cross Attention
Layer Normalization
Masked Multi-Head Attention
<start>
Hum
dost
hai
Tokenizer
<start> Hum dost hai
Right Shift
Hum dost hai



# WEEK 6: PRE-TRAINING – FINE-TUNING

**Pretraining** is when a model (like BERT, GPT, T5, etc.) is trained on a **huge amount of generic text data** *without any specific downstream task in mind*. The model reads a massive amount of text (like the entire internet) to learn the basics: grammar, logic, facts about the world, and how to complete a sentence. It learns *how* to learn, but it doesn't have a specific job yet. This part is incredibly expensive and time-consuming.

Finetuning = Taking the pretrained model and **training it a little more** on a **specific task like** Sentiment classification, Named Entity Recognition, Machine Translation, Question Answering, Fraud detection

**ELMo is a contextual word embedding model-** It came before BERT and was revolutionary for one reason- it gives different embeddings for the same word depending on the sentence.

ELMo uses a **bidirectional 2-layer LSTM language model**.
**Steps:**

1. Input sentence → characters (ELMo is character-based, not token-based)
2. Characters go into a CNN → produce word-level embeddings
3. Go into a **2-layer BiLSTM language model**
    - Forward LM: predicts next word
    - Backward LM: predicts previous word
4. Combine outputs from both directions
5. Weighted sum → Final contextual ELMo embedding

# *The BERT Architecture: An Encoder-Only Topology*

The core of BERT is a multi-layer bidirectional Transformer encoder. This architecture is designed to map input sequences to a sequence of continuous representations, which are then fed into task-specific output layers.

BERT utilizes WordPiece tokenization, a sub-word segmentation algorithm that strikes a balance between character-level and word-level representations. The vocabulary size is fixed at 30,522 tokens. Common words (e.g., "the", "play") exist as whole tokens, while rarer words are decomposed into meaningful subunits. For example, the word "playing" might be tokenized as ["play", "##ing"], where ## denotes a suffix.

**Segment and Position Embeddings -** BERT injects **Position Embeddings**. Unlike the sinusoidal functions used in the original Transformer paper, ***BERT uses learned absolute position embeddings***, supporting sequence lengths up to 512 tokens.[3] This means the model learns a unique vector for "position 1," "position 2," and so on.

Furthermore, to support sentence-pair tasks (like Question Answering), BERT includes **Segment Embeddings**. A learned vector $E_A$ is added to every token in the first sentence, and a vector $E_B$ is added to every token in the second sentence. This allows the model to easily distinguish between a Question and a Passage, or a Premise and a Hypothesis.[5]

**Special Tokens**

| Token | Used In | Purpose |
|---|---|---|
| [CLS] | Every input | Sequence representation (classification) |
| [SEP] | Single or paired sentences | Separates segments / marks end |
| [PAD] | Batch training | Makes sequences equal length; ignored using attention mask |
| [MASK] | Pretraining only | MLM prediction token |

**Pre-Training Objectives: The Engine of Understanding**- BERT's capability to generalize across domains is a result of its rigorous pre-training on massive unlabeled corpora (specifically, the BooksCorpus and English Wikipedia). Unlike supervised learning, which requires labeled data, BERT employs self-supervised learning via **two objectives: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP).**

**Masked Language Modeling (MLM) -** "bidirectional" nature creates a tricky problem for training.

Standard models (like GPT) are trained to predict the next word.

- Input: "The man went to the..." - - - - - >> Target: "store"

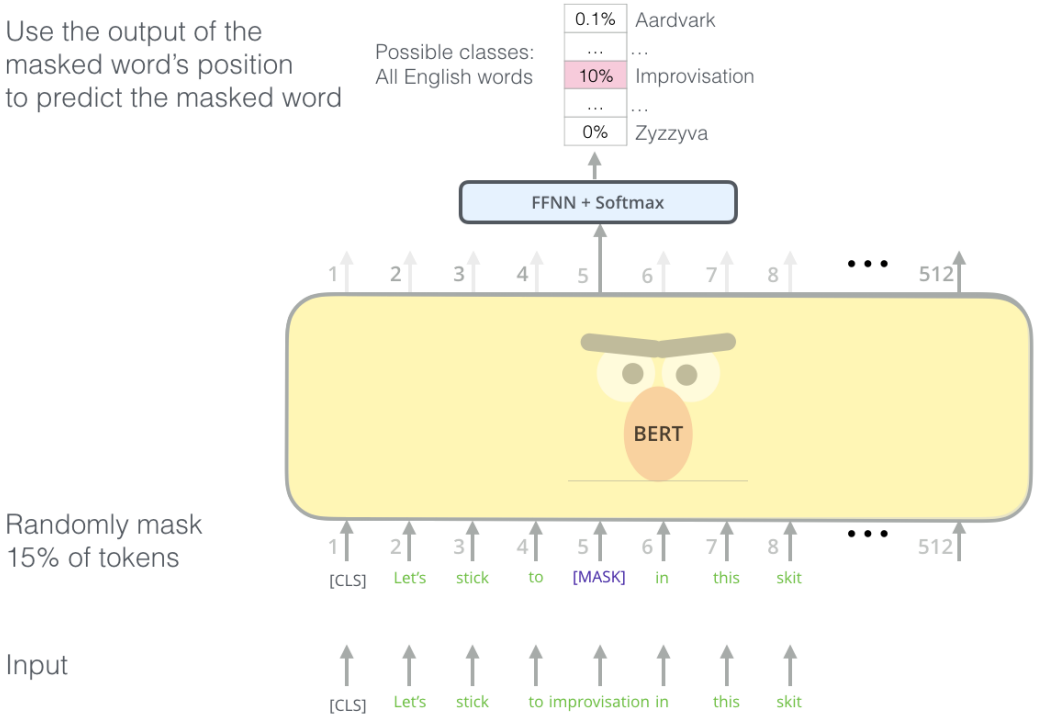
**Since BERT sees the *entire* sentence at once (both left and right context)**,- if we simply gave it the sentence "The man went to the store" and asked it to predict the word "store" - the model can see the answer in the input, it just copies it and learns nothing about language structure. Since we can't predict the *next* word (because of the bidirectional cheating), we hide words *inside* the sentence.

- **The Trick:** We replace 15% of the words with a special [MASK] token.
- **The Goal:** The model must guess the original word based on the context from **both** directions.

**Example:** Input: The man went to the [MASK] to buy milk. Target: store

1. **80% of the time:** The token is replaced with the [MASK] token.
    - *Rationale:* This forces the model to rely on the contextual embeddings of the surrounding words to predict the missing concept.
2. **10% of the time:** The token is replaced with a **random word** from the vocabulary.
    - *Rationale:* This forces the model to keep its contextual representation robust. The model must determine that the random word is semantically incoherent with the sentence and predict the correct original word. This prevents the model from simply copying the input to the output.
3. **10% of the time:** The token remains **unchanged**.
    - *Rationale:* This biases the representation towards the actual observed word. Since the model does not know *which* words are targets and which are inputs, it is forced to maintain a distributional representation of the correct token at every position, even when the word is visible. This bridges the gap between pre-training and fine-tuning.

BUT BECAUSE OF MASKS- BERT KNEW WHICH ALL TOKENS WERE MANIPULATED-



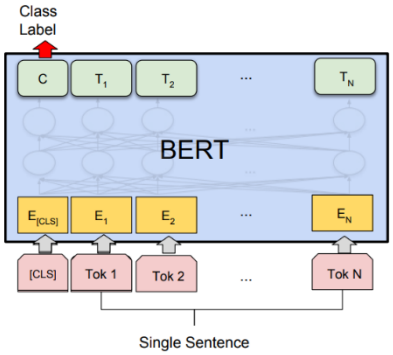

**Next Sentence Prediction (NSP) -**Understanding individual words isn't enough. We also need to understand how sentences relate to each other. **The Goal:** The model has to predict: Did Sentence B actually follow Sentence A, or is it just a random sentence from a different document?

- **Input**: A pair of sentences, A and B.
- **Positive Samples** (IsNext): 50% of the time, B is the actual sentence that immediately follows A in the training corpus.
- **Negative Samples** (NotNext): 50% of the time, B is a random sentence from the corpus.

The RoBERTa (Robustly Optimized BERT Pretraining Approach) study demonstrated that the NSP task might actually be detrimental or redundant when training on sufficiently large datasets. By training on contiguous sequences of full documents (without the NSP interruption) and using dynamic masking (changing the mask pattern every epoch), RoBERTa achieved superior results without the NSP head. DeBERTa (Decoding-enhanced BERT with disentangled attention) introduces a structural change to the attention mechanism. DeBERTa keeps token embedding and position embedding separate and computes attention scores using disentangled matrices

**Fine-Tuning Architectures: Adapting the Heads-**

**Sequence Classification with BERT-**Sequence classification tasks aim to assign a single label to an entire text input (e.g., sentiment, grammatical correctness, or semantic equivalence).BERT handles these tasks by using the contextualized representation of the [CLS] token.




For a single sentence classification task:
[CLS] Sentence [SEP]

- The entire sequence passes through the BERT encoder.
- Due to self-attention, each token attends to every other token.
- The [CLS] token acts as a learnable "summary" vector for the whole input.

Let the final hidden state corresponding to the [CLS] token be:

$$C \in \mathbb{R}^H$$

- $H = 768$ for BERT-Base
- $H = 1024$ for BERT-Large

Self-attention ensures $C$captures information from all tokens, making it suitable for sequence-level predictions.
Classification Head Architecture -A task-specific classification layer is added on top of $C$:

$$P = \text{softmax}(CW^T)$$

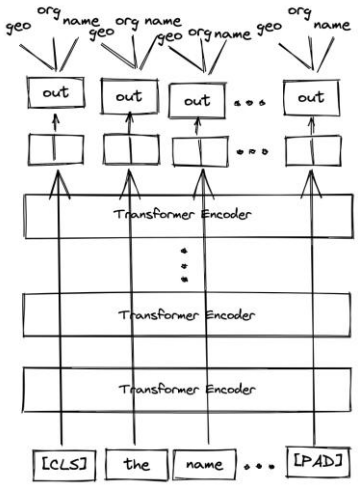
Where:

- $W \in \mathbb{R}^{K \times H}$is a trainable parameter matrix
- $K$= number of classes
  - $K = 2$for binary sentiment
  - $K = 3$for entailment tasks (e.g., MNLI)

This is the entire fine-tuning head for sequence classification.
Loss Function- A standard cross-entropy loss is used and gradients flow back through:

$$\mathcal{L} = -\sum_{i=1}^{K} y_i \log P_i$$

**Token Classification (NER) Using BERT**




Barack Obama was born in Hawaii.
After WordPiece tokenization:
['Barack', 'Obama', 'was', 'born', 'in', 'Hawaii', '.'] - - [CLS] Barack Obama was born in Hawaii . [SEP]
For each token in the input, BERT produces a final hidden vector:

$$T_i \in \mathbb{R}^H$$

Where:

- **i** = token index
- **H** = hidden dimensionality
  (BERT-Base: 768, BERT-Large: 1024)

These T_i vectors already encode:

- semantics
- context from left and right (bidirectional)
- syntactic relations

This makes them perfect for token classification.

**3. Classification Head (One Linear Layer per Token) -** A simple linear classifier is added on top of each token:

$$P_i = \text{softmax}(T_i W^T)$$

Where:

- **W** = weight matrix $\mathbb{R}^{E \times H}$
- **E** = number of entity tag classes - (Example: 9 for CoNLL-2003: PER, ORG, LOC, MISC, O, etc.)
- For each token → softmax gives a probability distribution over all entity classes.

BERT uses **WordPiece** tokenization. - A single word may become several tokens:
Example: - Washington → ["Wash", "##ing", "##ton"] - But NER labels are assigned **per word**, not per subword.
**Industry-standard solution:**

- Compute predictions for ALL subwords.
- **Only use the first subword's prediction** for the final label. AND Ignore the remaining subwords during loss computation.

**How ignored subwords are handled:** They are assigned: label_id = -100 -
**If a token is marked with label -100, loss is skipped for that position.**

**Span-Based Question Answering (SQuAD) with BERT**

In extractive QA, the model must extract a continuous span (subsequence) from the passage that answers the question.

Question: “Where was Obama born?”

Passage: “Barack Obama was born in *Hawaii*.”

Correct span: "Hawaii"

(start index = 5, end index = 5)

**1. Input Format**

BERT receives**: [CLS] Question Tokens [SEP] Passage Tokens [SEP]**

Token embeddings → Transformer → final contextual representations

$$T_1, T_2, \ldots, T_n \in \mathbb{R}^H$$

Each $T_i$contains:

- question–passage interactions (via cross-attention internally)
- full bidirectional context

These contextualized vectors enable span prediction.

**2. BERT’s QA Head: Two Trainable Vectors** For SQuAD, BERT adds only **two** new learnable parameters:

- **Start vector** $S \in \mathbb{R}^H$
- **End vector** $E \in \mathbb{R}^H$

These are **not** per-token weights — they are **global vectors** shared across all positions.

🔹 **3. Computing Start and End Probabilities -** For every token position $i$, compute:

✓ **Start score:** $\text{score}_i^{start} = S \cdot T_i$

✓ **End score:** $\text{score}_i^{end} = E \cdot T_i$

Then apply softmax across all tokens:

**Start probability:**

$$P_i^{start} = \frac{e^{S \cdot T_i}}{\sum_j \; e^{S \cdot T_j}}$$

**End probability:**

$$P_i^{end} = \frac{e^{E \cdot T_i}}{\sum_j \; e^{E \cdot T_j}}$$

🔹 **4. Training Objective (Cross-Entropy)**

During training:

- Ground truth = (start_index, end_index)
- Loss = sum of two cross-entropies:

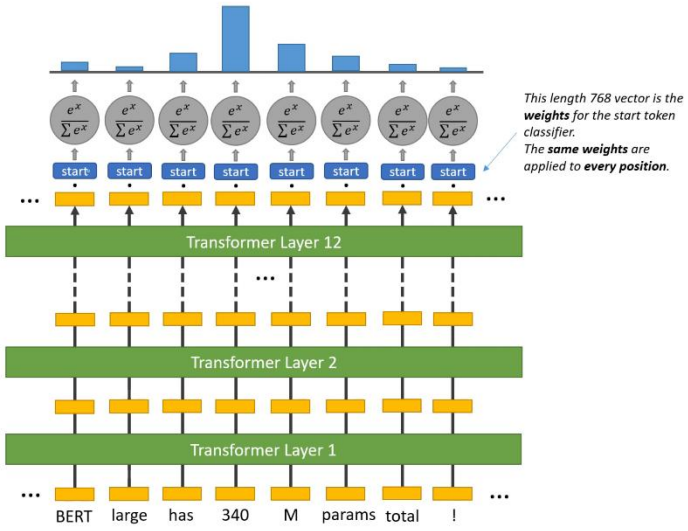
$$\mathcal{L} = CE(P^{start}, y^{start}) + CE(P^{end}, y^{end})$$

This trains the model to predict the correct boundaries.

🔹 **5. Inference (Selecting Best Span)**

At inference, choose span $(i, j)$:

$$(i^*, j^*) = \arg\max_{j \geq i} (S \cdot T_i + E \cdot T_j)$$




Explanation:

- Add start score + end score
- Ensure answer is not reversed (end must be ≥ start)
- Apply optional max span length constraint (e.g., ≤ 30 tokens)

This yields the best plausible answer span.

# *An Encoder-Decoder – T5*

*T*raditional transfer learning paradigms, models like BERT required specific architectural modifications—"heads"—to adapt to different downstream tasks. A classification task might require a linear layer on top of the `CLS` token; a question-answering task might require a span-prediction head. This complicated the transfer learning.

T5 eliminates this complexity by treating every NLP problem as a text generation task. In this framework, the input is always a text string, and the output is always a text string.

- **Translation:** The model receives "translate English to German: That is good." and is trained to output "Das ist gut."
- **Classification:** For the CoLA (Corpus of Linguistic Acceptability) task, the input "cola sentence: The course is jumping well." prompts the output "unacceptable."
- **Regression (STS-B):** Even regression tasks are cast as text generation. For the Semantic Textual Similarity Benchmark, the input "stsb sentence1: The cat sat. sentence2: The cat lay." prompts the model to generate the string "4.8".

This unification allows T5 to use a single loss function—standard maximum likelihood using cross-entropy—and a single set of hyperparameters across all tasks. It relies heavily on **task prefixes** (textual prompts prepended to the input) to prime the model for the specific operation required.

Training uses **one loss**:

$\mathcal{L} = -\sum_t \log p(y_t \mid y_{<t}, x)$ (standard cross-entropy)

**No task-specific heads** , **No architectural changes** , **Shared hyperparameters** across all tasks

Seamless multi-task training via text prompts

### Architectural: Relative Positional Embeddings in T5

T5 retains the overall Transformer backbone but **fundamentally departs** from earlier models in how it encodes *token order*. This design choice—**Relative Positional Bias**—is one of the major reasons T5 generalizes so well, especially on long sequences.

Vanilla Transformers (Vaswani et al., 2017) rely on **absolute positional encodings**, either: **Sinusoidal** (fixed functions) OR **Learned absolute vectors** , These are **added** to token embeddings. **However, absolute encodings have a fatal weakness:**

If the model is trained on sequences of length ≤ N, it learns embeddings only up to position N.

For positions > N (e.g., 513, 600, or 2048), there are **no pre-trained positional vectors**.

This leads to:

- Poor length generalization OR Degraded performance on longer contexts OR Inability to extrapolate beyond training lengths OR Failure in tasks like long-document summarization, long QA, or multi-hop reasoning

This is because absolute positions bind the model to a **fixed coordinate system** that it has not seen beyond training maximum length.

**2. T5's Solution: Relative Positional Bias (a scalar added to attention logits)** T5 abandons absolute positions entirely and adopts a **relative positional encoding** scheme.

**The Key Insight**

Rather than telling the model:

*"This token is at position 238."*

T5 tells it:

*"This token is 5 places to the right of the current token."*

This shift from **absolute** to **relative** position allows T5 to generalize to unseen sequence lengths naturally.

---

**3. How T5 Implements Relative Position: The T5 Bias**

In the self-attention mechanism, attention weights are computed as $\text{Attention}(i,j) = \frac{Q_i \cdot K_j}{\sqrt{d_k}}$

T5 modifies this by adding a learnable scalar bias based on the *relative distance*: $\text{Attention}(i,j) = \frac{Q_i \cdot K_j}{\sqrt{d_k}} + b_{\text{rel}(i,j)}$

Where:

- $i$= position of the query token
- $j$= position of the key token
- $b_{\text{rel}(i,j)}$= scalar learned for the bucket representing (i – j)

Important:

- No positional vectors are added to embeddings
- Only scalar biases are added to attention logits

Bucketing Mechanism: Efficient Generalization to Long Distances - Why do we need bucketing? Relative position means: distance $= j - i$ But sequences can be arbitrarily long (10 tokens or 10,000 tokens).

Small distances get their own bucket. Large distances get grouped together in logarithmic ranges. – *distance ka bucketing hua h*

buckets. This reflects the linguistic intuition that precise relative position matters most for local syntactic dependencies, while broader relative position suffices for long-range semantic relationships.

Implication: This design allows T5 to generalize zero-shot to sequence lengths longer than those seen during training, a capability that is particularly advantageous for tasks like long-document summarization where the input length varies wildly.

**The "Span Corruption" Pre-training Objective (T5)**
"Artificial intelligence is transforming industries across the world by enabling automation, insights, and enhanced decision-making."
Tokenize it (simplified):
Artificial | intelligence | is | transforming | industries | across | the | world | by | enabling | automation | insights | and | enhanced | decision-making .
Total tokens = 16. T5 corrupts 15% ≈ 2–3 tokens but as *spans*, not single tokens.
Step 1 — Select Tokens to Corrupt
Assume random selection chooses: "transforming" "world" "automation insights and" (these were adjacent)
**Step 2 — Merge into Spans** Now merge adjacent corrupted tokens:
We get **three spans**:

1. **Span 0:** [transforming]
2. **Span 1:** [world]
3. **Span 2:** [automation insights and] (multi-token span)

**This is crucial — T5 does not mask ka ek-ek word like BERT — it masks phrases.**
**Step 3 — Replace Each Span With a Unique Sentinel Token -** Now the corrupted input becomes:
Artificial intelligence is <extra_id_0> industries across the <extra_id_1> by enabling <extra_id_2> enhanced decision-making .
Breakdown:

- <extra_id_0> replaces "transforming"
- <extra_id_1> replaces "world"
- <extra_id_2> replaces "automation insights and"

**Sentinel tokens are placeholders** that mark exactly where each missing chunk should be reconstructed.
**Step 4 — Target Output Contains Only the Missing Spans -** The decoder's target is simply:
<extra_id_0> transforming <extra_id_1> world <extra_id_2> automation insights and <Z>
All spans and In correct order and With their sentinel markers and Ends with <Z> (EOS)

Why this is a Sequence-to-Sequence Learning Problem
The encoder sees a damaged version of the text. The decoder must regenerate only the missing parts, not the whole text.
This forces deep understanding:

- syntax (word order)
- semantics (meaning relations)
- discourse structure (how parts connect)

# *An Encoder-Decoder - BART*

BART does not use sinusoidal positional encodings. - It uses learned positional embeddings, just like RoBERTa and GPT-2.
Instead of just masking tokens like BERT or masking spans like T5, **BART corrupts the input in *many* possible ways and trains the model to reconstruct the original document.** - - its called **Arbitrary Noising Functions → Universal Denoising Autoencoder**
**The Noise Manifold — BART's Five Types of Corruption**

---

**3.1 Token Masking (BERT-style)**
Example:
Original - - The economy is slowing due to inflation.
Corrupted - - The economy is <mask> due to inflation.
**3.2 Token Deletion (harder than masking)**
Tokens are *removed*, not replaced:
Original - - The economy is slowing due to inflation.
Corrupted - - The economy slowing inflation.
Why is this harder?

- No placeholder to indicate *something is missing*
- Model must infer **where** and **what** was deleted
- Strengthens syntactic rhythm & grammar detection

**3.3 Text Infilling (Poisson spans)**
**Most powerful and important objective.**
Span lengths ~ Poisson(λ = 3).
Original - -Artificial intelligence is transforming global industries at scale.
Span mask: "transforming global industries"
Corrupted - -Artificial intelligence is <mask> at scale.
The model must reconstruct: - -transforming global industries
Why Poisson?

- 0 → insert mask (BART creates missing segments)
- 3 → typical linguistic phrase length
- variable-length spans = more realistic corruption

This trains:

- phrase-level semantics
- long-range coherence
- variable-length sequence reconstruction

**3.4 Sentence Permutation (shuffle sentences)**

Document:

1. "The storm hit last night."
2. "Several houses were damaged."
3. "Rescue teams arrived early morning."

Corrupted (shuffled):

Several houses were damaged.

The storm hit last night.

Rescue teams arrived early morning.

Model must reorder them.

Trains:

- discourse understanding
- causal flow
- narrative structure
- logical ordering

Critical for:

- long-document summarization
- story generation
- explanation ordering

**3.5 Document Rotation**

Choose a random token and rotate the document.

Original:

[A] The stock market crashed yesterday. [B] Investors panicked. [C] Trading was halted.

If rotation starts at "Investors":

Corrupted:

Investors panicked. Trading was halted. The stock market crashed yesterday.

Model must figure out:

- where the true beginning is
- typical structure of introductions
- paragraph boundaries

This enhances long-range global coherence.

**Why BART's Noising Is Better Than BERT or T5 Alone**

**✓ BERT's MLM:**

- Only masks tokens → too easy
- Sequence length stays same → decoder impossible
- No generation abilities

**✓ T5's Span Corruption:**

- Only spans → not general enough
- No permutations, no deletion, no rotation

**✓ BART generalizes ALL of them:**

- token-level
- phrase-level
- sentence-level
- structural-level
- document-level

# *GPT: Scaling the Transformer Decoder*

Structural Techniques
1. Pre-LayerNorm (LayerNorm BEFORE attention & FFN)-A technique that normalizes activations so training stays stable.
🔶 Two ways to place it:

1. Post-LayerNorm (original Transformer): LN comes after each residual block.
2. Pre-LayerNorm (GPT-2 and GPT-3): LN comes before self-attention and FFN blocks.

Why Pre-LayerNorm is better?
Problem with Post-LN in deep models:
If you have many layers (48 in GPT-2, 96 in GPT-3), the gradients often:

- Explode , vanish , cause instability, fail to converge

Pre-LN solves this:

- Gradients flow smoothly backward,No explosion/vanishing,Model becomes trainable even at huge depths

This change made GPT-3's 96-layer stack possible.
2. GeLU (Gaussian Error Linear Unit) instead of ReLU
What is ReLU? ReLU(x) = max(0, x) Sharp, piecewise-linear, zero for negative values.
What is GeLU? GeLU(x) = x * Φ(x) (where Φ is the CDF of a normal distribution) - It is a smooth, probabilistic ReLU.
3. Byte-Level BPE Tokenizer
This is one of the most important GPT-2 innovations.
4.What is BPE? Byte Pair Encoding — breaks text into subword pieces.
**Positional Encoding: Learned Positional Embeddings (NOT Sinusoidal)**
Both **GPT-2** and **GPT-3** use **learned (trainable) positional embeddings**, not sinusoidal encodings.

The primary source of tokens for GPT-3 was **Common Crawl**, a massive, open repository of web crawl data. While Common Crawl provides the necessary volume (petabytes of text), it is inherently noisy, containing significant amounts of spam, unintelligible text, and low-quality content. To address this, the researchers implemented an aggressive filtering pipeline.

1. **Quality Classifier:** A logistic regression classifier was trained to distinguish between "high-quality" content and standard Common Crawl noise. The positive examples for this classifier were drawn from **WebText2**, a curated dataset of web pages linked from Reddit posts with high engagement (karma), which served as a proxy for human-validated quality. Common Crawl documents were then scored, and only those passing a certain quality threshold were retained.
2. **Fuzzy Deduplication:** To prevent the model from overfitting to redundant text—and to ensure the integrity of the validation split—extensive fuzzy deduplication was performed at the document level using MinHashLSH. This process removed documents that were significantly similar to each other or to the test set, preventing "data contamination" where the model might memorize test questions present in the training data.

**Example:**

| Text | Classifier score | Kept? |
|---|---|---|
| Wikipedia paragraph | 0.98 | ✔ Yes |
| StackOverflow answer | 0.92 | ✔ Yes |
| Broken HTML snippet | 0.12 | ✗ No |
| SEO spam like "Buy shoes cheap!!" | 0.05 | ✗ No |

| Dataset | Raw Quantity (Tokens) | Weight in Training Mix | Effective Epochs (300B Training) |
|---|---|---|---|
| Common Crawl (filtered) | 410 Billion | 60% | 0.44 |
| WebText2 | 19 Billion | 22% | 2.9 |
| Books1 | 12 Billion | 8% | 1.9 |
| Books2 | 55 Billion | 8% | 0.43 |
| Wikipedia | 3 Billion | 3% | 3.4 |

**In-Context Learning (ICL)**
**ICL = learning inside the prompt without updating weights.**
Before GPT-3, to teach a model sentiment analysis or translation, you had to **fine-tune**: change the weights via backpropagation.
GPT-3 showed something shocking:
*If the model is large enough, it can learn a task on the fly, inside the prompt, purely by reading examples — without updating any weights.*
This happens through two loops:
**Outer Loop — Pretraining (Weight Learning)**
This is where the **weights** learn:

- Grammar & facts & reasoning patterns & "how to detect patterns from examples"

During pretraining, the model sees **300 billion tokens** and learns general abilities - This is where GPT-3 learns **how to learn**.
**Inner Loop — In-Context Learning (Task Learning)**
This happens **during inference only**, using the context (prompt).
Example:
Dog → Chien
Cat → Chat
Mouse → ?
The model **adapts its internal activations**, not its weights.
The "learning" exists **only in the attention states and KV-cache**, and disappears after the context clears.
**Two components:**
🟡 **1. Previous Token Head**
It attends to the token just before the current one.
Example: At position t seeing "Potter", it links to position t−1 "Harry".
This builds the sequence representation.

🟠 **2. Induction Head**

This head does something amazing:

1. It sees the current token A.
2. Searches the prompt for a **previous occurrence** of A.
3. Jumps to the token that came **after** the previous A.
4. Boosts the probability of that token.

**Prompting Strategies in GPT-3 — Zero-Shot, One-Shot, Few-Shot (Explained in Simple + Deep Way)**

Large language models like GPT-3 don't update weights during inference.

So the only way to "teach" them a task at query time is through **prompting**.

**1. Zero-Shot Prompting**

**Definition** You give the model **only instructions**, no examples.

**Example** Translate English to French: cheese => ??

**How it works** The model must rely **only on its pre-trained knowledge**:

- What is translation?
- What does "=>" mean?
- What is the French word for cheese?

It uses prior knowledge stored during pretraining.

**Performance**

Zero-shot is surprisingly strong but:

- Highly sensitive to wording
- Sensitive to punctuation
- Sensitive to format

**Intuition**

Zero-shot → The model is guessing the task *from scratch using general intelligence*.

**2. One-Shot Prompting**

**Definition**

You give **one example** of how the task should be done.

**Example**

Translate English to French:
sea otter => loutre de mer
cheese =>

**Why one example helps**

The example acts like a **task anchor**:

- It shows the format
- It clarifies ambiguity (translation? definition? synonym?)
- It pulls the model's internal state toward the right "subspace"

This is because attention heads recognize repetition patterns.

**3. Few-Shot Prompting**

**Definition**

You give **many examples** (10–100), filling the context window.

**Example**

Translate English to French:
sea otter => loutre de mer
peppermint => menthe poivrée
plush giraffe => girafe peluche
(to many more)
cheese =>

**Why few-shot is powerful**

Many examples activate **induction heads**, a special mechanism inside transformers that:

1. Detect repeated patterns
2. Predict the continuation of the pattern
3. Apply the correct mapping to new inputs

The model effectively performs "meta-learning" inside the context.

**Effect**

Few-shot prompting behaves like **temporary fine-tuning**,
but **without any weight updates**.

**Performance** Few-shot > One-shot > Zero-shot

**Failures and Weaknesses**

Despite these successes, GPT-3 struggled significantly on tasks requiring "bidirectional" reasoning or complex logical deduction.

- **ANLI (Adversarial Natural Language Inference):** On this dataset, which involves determining if a hypothesis follows from a premise, GPT-3 performed poorly, barely beating random chance in some categories.
- **Comparisons:** On tasks like **DROP** (Discrete Reasoning Over Paragraphs), which requires reading a passage and performing arithmetic on numbers found within it, GPT-3 lagged significantly behind fine-tuned models. The autoregressive nature of GPT-3 makes it difficult to perform "multi-hop" reasoning where the model must read the end of a text, hold a number in memory, look back to the beginning, and perform an operation.

How GPT Is Trained: Feeding Ground Truth & Computing Loss
GPT training is just next-token prediction.
For a given training sentence:
"The cat sat on the mat"
We tokenize:

| Position | Token |
|---|---|
| 1 | The |
| 2 | cat |
| 3 | sat |
| 4 | on |
| 5 | the |
| 6 | mat |

---

🔹 Step 1: Input Sequence (Left-Shifting Trick)
GPT receives the input tokens, shifted by one position:
Input to model:
[The, cat, sat, on, the]
Target labels (ground truth):
[cat, sat, on, the, mat]
So:

- Model sees "The" and must predict "cat"
- Model sees "The cat" and must predict "sat"
- Model sees "The cat sat" and must predict "on"

This continues for all tokens.

---

🔹 Step 2: Apply Causal Mask
At each position $t$, the model can attend only to positions ≤ t.
Example:
At token 3 "sat":

- It can attend to: "The", "cat", "sat"
- It cannot attend to: "on", "the", "mat"

This ensures the model cannot “cheat” by looking at future tokens.

---

🔹 Step 3: Forward Pass → Predict Next Token Distribution
For each input position t, the model outputs a probability distribution over the entire vocabulary V (e.g., 50,000 tokens):
Example output distribution (fake numbers):
At position 1:
P(cat)=0.91
P(mat)=0.01
P(dog)=0.02
...
At position 2:
P(sat)=0.87
P(on)=0.03
...

---

🔹 Step 4: Compare Output With Ground Truth With Cross-Entropy Loss
GPT’s training loss is simply:

$$\text{Loss} = \sum_{t} \text{CrossEntropy}(P_{\text{model}}(x_t), x_{t+1})$$

Meaning:

- If the model predicted "cat" with 91% probability, loss is low
- If it predicted something else, loss is high

Loss is averaged across all positions of all sequences in the batch.

---

🔹 Step 5: Backpropagation Updates All Weights
The cross-entropy loss is backpropagated through:

- token embeddings
- Q/K/V projections
- attention heads
- feed-forward layers
- layer norms
- output softmax

This updates weights so next time the model is slightly better at predicting the next token.

# *Domain-Adaptive Pretraining (DAPT)*

Domain-Adaptive Pretraining (DAPT), where a generalist model is further trained on domain corpora, and Domain-Specific Pretraining from Scratch, where the model is initialized randomly and trained exclusively on domain data.The core premise of DAPT is that the syntactic and semantic structures learned from massive general corpora (like Wikipedia or BookCorpus) are transferable to specialized domains. For instance, the grammatical structure of a sentence in a biomedical paper largely mirrors standard English, even if the lexicon differs. DAPT exploits this by taking a converged model and subjecting it to further gradient updates on a domain-specific dataset.

Continual Learning: How DAPT Works Internally
DAPT is literally continuing gradient descent from a converged model.
If a general model learned: $\theta_{\text{general}}$
Then DAPT modifies it to: $\theta_{\text{domain}} = \theta_{\text{general}} + \Delta\theta$
This update Δθ encodes domain-specific statistics.
Catastrophic Forgetting & Alignment Drift

This is the central problem of DAPT.
Catastrophic Forgetting As the model optimizes the domain loss: - - - $L_{\text{domain}}$
its weights shift toward P(domain) and away from P(general).
General abilities decay.
Symptoms:

- BioBERT struggles with everyday reasoning
- Llama-2-Chat loses instruction-following
- Financially trained models forget common-sense metaphors

Because ALL weights are updated, a small gradient shift can degrade abilities acquired over millions of steps.
Alignment Drift Chat models are aligned through RLHF – Reinforcement Learning Through Human Feedback
DAPT destroys RLHF priors unless as alignment is *re-run* after DAPT.

LoRA reduces forgetting - LoRA trains only low-rank matrices, freezing the backbone.
This reduces—*not eliminates*—forgetting.

The Vocabulary-Lock Problem: Why DAPT Can Fail DAPT inherits the tokenizer of the base model.
Example - - - "Acetylcholinesterase" → "Acetyl" + "##cholin" + "##ester" + "##ase"
This causes: Attention dilution - One concept becomes 4 tokens → attention must reconstruct meaning.
Longer sequences - - - More tokens → less room in the context window.
Wrong semantics - - - "ester" (from polyester) ≠ biochemical "ester".
Thus embedding space is misaligned with domain. -- - - This is the key structural flaw of DAPT.

---

6. PubMedBERT: A Better Alternative (Training From Scratch)
PubMedBERT discarded the DAPT approach:
New vocabulary built entirely from PubMed
No fragmentation ,No wrong semantics , Dense representations of domain-specific terms
Trained from scratch on domain-only data and Avoids negative transfer from general web data.
📌 Result
PubMedBERT > BioBERT across all biomedical tasks (BLURB benchmark).
Reason:
Correct vocabulary + in-domain training beats general-model DAPT.

# *Multilingual Pretraining*

Multilingual LLMs like **mBERT, XLM-R, BLOOM, mT5** try to pack **hundreds of languages** into one shared set of parameters.
This creates two major problems:

1. **Capacity Dilution** (too many languages, not enough parameters)
2. **Gradient Interference** (languages fighting each other during training

**Temperature-Based Sampling (Multilingual Pretraining)**
**Goal:** Make sure **low-resource languages (LRLs)** don't get ignored and **high-resource languages (HRLs)** don't dominate training.
**Core Problem** Real-world multilingual corpora are *extremely* imbalanced:

| Language | Tokens |
|---|---|
| English | 150B |
| Chinese | 100B |
| Spanish | 40B |
| Swahili | 20M |
| Amharic | 10M |

Difference between English and Swahili:
**150,000,000,000 vs 20,000,000 → 7500× more**
If we sample batches proportional to real data (the "true distribution") →
**the model never sees the low-resource languages.**
This causes: - catastrophic forgetting for LRLs , poor embeddings for rare scripts ,degraded cross-lingual transfer
**The Solution: Temperature-Based Sampling**

Instead of sampling languages using their *true* proportions, we *smoothe* the distribution using: $p_i = \frac{q_i^\alpha}{\sum_j \ q_j^\alpha}$

$q_i$= real distribution of language $\alpha$= **temperature / smoothing factor**

3️⃣ **Effect of the Temperature α**
**If 0 < α < 1 (Flattening)** The distribution becomes **flatter**: HRLs get **downsampled** LRLs get **oversampled**
This is how mBERT and XLM-R are trained.
**If α → 0**
All languages become **equally likely**, regardless of corpus size.
(Extreme oversampling of LRLs)
This may overfit LRLs (dangerous if small corpus).

4️⃣ **Numerical Example (Very Intuitive)**

| Language | Real proportion (q) |
|---|---|
| English | 90% |
| Swahili | 10% |

---

**Case 1: α = 1.0 → No smoothing**

$$p_{\text{English}} = 0.9, p_{\text{Swahili}} = 0.1$$

Model sees English 9× more.

---

**Case 2: α = 0.5**
Compute:

$$0.9^{0.5} = 0.949, 0.1^{0.5} = 0.316$$

Normalize:

- English: 0.949 / (0.949 + 0.316) = **0.75**
- Swahili: 0.316 / (0.949 + 0.316) = **0.25**

Now English is only 3× more frequent
**➞ LRLs get more exposure**

**XLM-R** Uses **α = 0.3** → aggressive smoothing -
Results: +15% gains on Swahili, Urdu, Tamil **and** HRLs still reasonable - XLM-R's approach is more principled and effective.
**mBERT** Uses a simpler heuristic but same idea: Downsample big languages, upsample small ones.

# *Uptraining: Architectural Metamorphosis - - - TBD!!!!*

**Uptraining** = upgrading an existing pretrained model **without training from scratch**, usually by **changing architecture or capabilities** and then doing *a small amount* of continued training to recover performance.
This is different from:

| Term | Meaning |
|---|---|
| **Fine-tuning** | Task-specific, small dataset, small weight updates |
| **Continued Pretraining (CPT / DAPT)** | More general training on a large corpus, no architecture changes |
| **Uptraining** | **Architecture changes** + small retraining to recover model quality |

Uptraining lets you "reuse the sunk cost" of pretraining and make legacy models behave like new ones.

# WEEK 7: APPLICATIONS

## *Question answering*

The primary architectural driver for the Retriever-Reader split is the "inference bottleneck." Modern Large Language Models (LLMs) and Machine Reading Comprehension (MRC) models—such as BERT, RoBERTa, T5, and GPT-4—are computationally expensive. Their complexity, often measured in billions of parameters, precludes their application to every document in a corpus of millions. It is computationally infeasible to feed the entire English Wikipedia into a Transformer model to answer a single question like "When was the Eiffel Tower constructed?".

**Why Retriever–Reader Systems Still Exist -** (Even though GPT-4/5 seem very powerful)
LLMs like GPT are amazing, but they have **structural limitations** that make them **incapable of replacing retrieval pipelines** when it comes to factual accuracy, up-to-date knowledge, or large-scale information access.

1️⃣ **Context Window Limitation**
Even with huge context windows (GPT-4: 128k, GPT-5: up to millions), GPT **CANNOT** load or process entire databases or the whole internet at inference.
Why? **A Transformer processes *all tokens at once*** - - Complexity = **$O(n^2)$** over the sequence length.
Meaning:

- 100k tokens → manageable
- 1M tokens → expensive
- 10M tokens → nearly impossible
- The entire Wikipedia (6B tokens) → absolutely impossible

So a GPT cannot:
❌ "Read all Wikipedia pages every time you ask a question"
❌ Load 10 million documents into context
❌ Perform brute-force search across a corpus
That's why we use a **Retriever**:
✓ Retrieves only ~20 to 100 most relevant documents
✓ Cheap and fast (sub-millisecond search via BM25 / FAISS / HNSW / ColBERT)
✓ Avoids giving GPT a massive context that is mathematically intractable

---

2️⃣ **GPT's Knowledge Is Frozen (Static)**
LLMs learn during **pretraining** and then get "frozen."
This means the model:

- ❌ Knows nothing after its cutoff date
- ❌ Can't see new events
- ❌ Can't access private data
- ❌ Can't query your sales database
- ❌ Can't fetch today's news
- ❌ Can't read a PDF unless you provide it

LLMs operate like this:
"I know everything that was in my training data up to a certain year. If it's new, I have no idea."
A **Retriever** solves this: ✓ Fetches new documents ✓ Pulls data from your own files ✓ Keeps models up-to-date
✓ Enables RAG (Retrieval-Augmented Generation) ✓ Lets GPT answer questions on documents it never saw during training
This is why all enterprise AI uses retrieval pipelines.

---

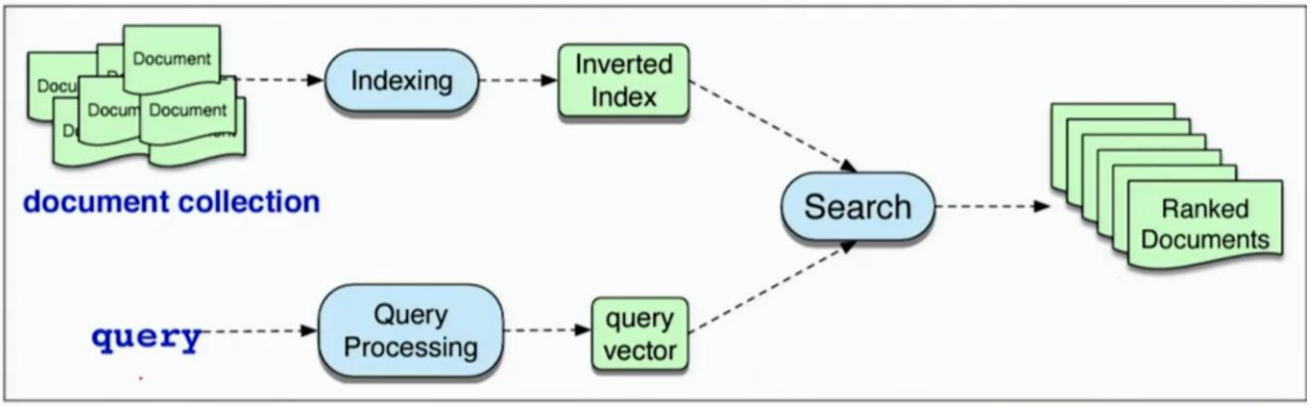
Therefore, the problem is decomposed:

1. **The Retriever:** A high-speed, high-recall module responsible for narrowing down the search space from millions of documents to a manageable subset of candidates (typically $k=20$ to $100$). This component relies on efficient indexing structures—inverted indices for sparse methods or Maximum Inner Product Search (MIPS) for dense methods.[5]
2. **The Reader:** A computationally intensive, high-precision module that processes the retrieved passages to extract the exact answer span or generate a natural language response. The reader assumes that the retriever has successfully included the correct answer within the candidate set.[1]

## Theoretical Foundations of **Sparse Retrieval**

Sparse retrieval is the classical family of information retrieval (IR) methods used long before neural networks.

TF-IDF




*term frequency-inverse document frequency (tf-idf)*

$$tf_{t,d} = log_{10}(1 + count(t,d))$$

$$idf_t = log_{10}\frac{N}{df_t}$$

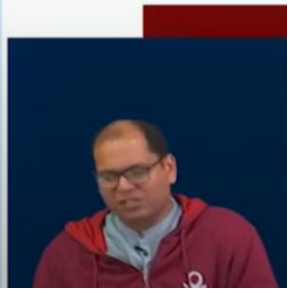
$$tf - idf(t,d) = tf_{t,d} \cdot idf_t$$

| Query | | | | | | |
|---|---|---|---|---|---|---|
| **word** | **cnt** | **tf** | **df** | **idf** | **tf-idf** | **n'lized** = tf-idf/$\|q\|$ |
| sweet | 1 | 1 | 3 | 0.125 | 0.125 | 0.383 |
| nurse | 0 | 0 | 2 | 0.301 | 0 | 0 |
| love | 1 | 1 | 2 | 0.301 | 0.301 | 0.924 |
| how | 0 | 0 | 1 | 0.602 | 0 | 0 |
| sorrow | 0 | 0 | 1 | 0.602 | 0 | 0 |
| is | 0 | 0 | 1 | 0.602 | 0 | 0 |

$|q| = \sqrt{.125^2 + .301^2} = .326$

**Query**: sweet love
**Doc 1**: Sweet sweet nurse! Love?
**Doc 2**: Sweet sorrow
**Doc 3**: How sweet is love?
**Doc 4**: Nurse!

| | Document 1 | | | | | Document 2 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **word** | **cnt** | **tf** | **tf-idf** | **n'lized** | **× q** | **cnt** | **tf** | **tf-idf** | **n'lized** | **×q** |
| sweet | 2 | 1.301 | 0.163 | 0.357 | **0.137** | 1 | 1.000 | 0.125 | 0.203 | **0.0779** |
| nurse | 1 | 1.000 | 0.301 | 0.661 | 0 | 0 | 0 | 0 | 0 | 0 |
| love | 1 | 1.000 | 0.301 | 0.661 | **0.610** | 0 | 0 | 0 | 0 | 0 |
| how | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| sorrow | 0 | 0 | 0 | 0 | 0 | 1 | 1.000 | 0.602 | 0.979 | 0 |
| is | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $\|d_1\| = \sqrt{.163^2 + .301^2 + .301^2} = .456$ | | | | | | $\|d_2\| = \sqrt{.125^2 + .602^2} = .615$ | | | | |
| Cosine: $\sum$ of column: **0.747** | | | | | | Cosine: $\sum$ of column: **0.0779** | | | | |



While revolutionary at its inception, TF-IDF suffers from critical limitations that render it insufficient for high-precision QA tasks:

1. **Lack of Saturation:** The Term Frequency component in standard TF-IDF is linear. If a word appears 20 times, it is weighted 20 times higher than if it appears once. However, in natural language, the marginal information gain of a word decreases as it is repeated. A document mentioning "computer" 100 times is not necessarily 100 times more relevant than one mentioning it 10 times; it may simply be a longer document or a repetitive one. This linearity can lead to relevance distortion.
2. **Length Bias:** TF-IDF naturally favors longer documents because they have higher term counts merely by virtue of their length. Without normalization, long, verbose documents dominate the ranking, potentially burying concise, relevant passages.
3. **Unbounded IDF:** For extremely rare terms, the IDF value can be disproportionately high, leading to ranking instability where a single occurrence of a rare typo might outweigh multiple occurrences of relevant terms.

Okapi BM25 — The Probabilistic Refinement of TF-IDF
Okapi BM25 was developed to **fix the weaknesses of TF-IDF** (linearity + length bias).
It is based on the **Probabilistic Retrieval Framework (PRF)**, especially the **Binary Independence Model (BIM)**

**BM25 solves two fundamental TF-IDF flaws:**
**1. Term Frequency Saturation — controlled by $k_1$**
TF-IDF treats term frequency as **linear**:

- 10 occurrences = 10× importance
- which is unrealistic

BM25 uses a **non-linear saturation curve**: $y = \frac{tf\,(k_1+1)}{tf+k_1}$

As $tf \to \infty$, the score approaches $k_1 + 1$.
Meaning:

- The first occurrence of a term gives huge information
- Later repetitions provide **diminishing returns**
- Realistic representation of human relevance judgment

---

**2. Length Normalization — controlled by $b$**
The term: $1 - b + b \cdot \frac{|D|}{avgdl}$

penalizes long documents.

- If document is **longer than average**, denominator ↑ → score ↓
- If **shorter**, denominator ↓ → score ↑

Meaning:

- Long documents mentioning "computer" 3 times may be noisy
- Short documents mentioning it 3 times are likely *focused*
-

**Limits of Lexical Matching (Why BM25 is Not Enough)**
Despite BM25's probabilistic power, it has fatal limitations due to **lexical gap**:
**1. Strict string matching -** If words differ even slightly, matching fails:

- "canine nutrition"
- "dog food"

BM25 → **score = 0**
Dense models → find semantic similarity.

---

**2. No synonym understanding**

- "doctor" ≠ "physician"
- "purchase" ≠ "buy"
- "AI" ≠ "artificial intelligence"

Keyword-based systems simply cannot bridge vocabulary differences.

---

**3. No polysemy understanding**
Same word, different meaning:

- "Apple" (company)
- "apple" (fruit)

BM25 scores both equally.

---

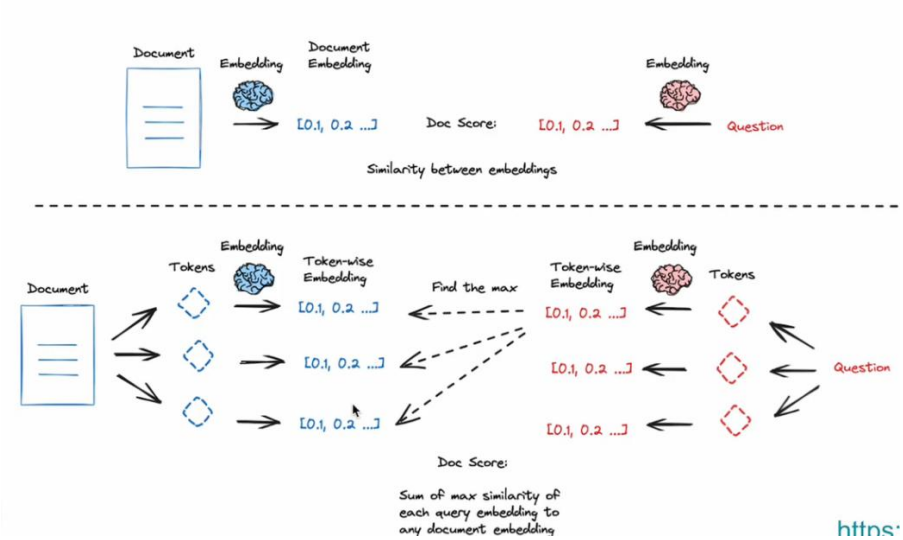
**4. No contextual understanding**
TF-IDF / BM25 treat documents as **bags of words**:

- Order ignored
- Meaning ignored
- Grammar ignored

Thus BM25 cannot understand:

- Paraphrases
- Conceptual similarity
- Multi-hop reasoning

ColBERT




The main idea: Do all the expensive BERT computation upfront (offline), but delay query–document interaction until the very last step.
And crucially:
DPR — Compress everything → A document = 1 vector
ColBERT — Keep everything → A document = 100–200 token vectors
ColBERT represents a document as a *cloud* of token vectors.
This lets ColBERT preserve → Names, numbers, dates, entities, verbs, negations, word order cues
Architecture: Two Encoders (Like DPR), Completely Different Outputs
ColBERT still uses a query encoder and document encoder, but:

- They output many vectors, not one.
- Each vector corresponds to a token, not the whole passage.

Query Encoder $E_Q(q) \rightarrow \{v_1, v_2, \dots, v_{N_q}\}$

Where each $v_i$= contextualized query token embedding.
Document Encoder $E_D(d) \rightarrow \{u_1, u_2, \dots, u_{N_d}\}$

Each $u_j$= contextualized document token embedding.
To reduce storage:

- A linear projection reduces dimension (e.g., 768 → 128).
- All vectors are L2-normalized.

The MaxSim Operator - The heart of ColBERT is the MaxSim scoring function:

$$S_{q,d} = \sum_{i \in q} \max_{j \in d} (v_i \cdot u_j)$$

Breakdown:
1 Token-to-Token Similarity - Compute similarity of every query token with every document token:
Similarity matrix: $N_q \times N_d$
2 Max-Pooling For each query token $v_i$:
Find the single best-matching token in the document.
This is the "soft exact match" effect:

- "bank" (finance sense) → matches "institution", "loan", "credit"
- "bank" (nature sense) → matches "river", "shore"

3 Summation Add the best-match similarity for all query tokens.
This yields the final document score.

**Training ColBERT via Cross-Encoder Distillation**
A Cross-Encoder is a model that takes the query and the document together as a *single input sequence* to the Transformer, letting all tokens interact with each other.
Example input: [CLS] query tokens … [SEP] document tokens … [SEP]
Then the Transformer processes everything jointly.
It outputs a single relevance score (usually from the CLS token).

Step 1 — Prepare Query + Candidate Documents

For each query q, we collect:

- Positive passages (contain answer)
- Hard negatives (similar but wrong)
- Easy negatives (random wrong docs)

Example for query: *"When was the Eiffel Tower built?"*

Candidate passages:

- P1: "The Eiffel Tower was constructed in 1887–1889." → positive
- P2: "The tower in Paris is famous…" → hard negative
- P3: "The Great Wall of China…" → easy negative

Step 2 — Teacher Cross-Encoder Computes Gold Scores

For each pair (q, P): The cross-encoder jointly encodes them: BERT([CLS] q [SEP] P [SEP])

It outputs a relevance score $S_t$. These scores are extremely high-quality because BERT sees full interactions between q tokens and passage tokens.

| Passage | Teacher score $S_t$ |
|---|---|
| P1 | 10.2 |
| P2 | 3.1 |
| P3 | -0.8 |

These are dense, expressive semantic labels.

Step 3 — Student ColBERT Computes Its Own Scores

ColBERT encodes:

- Query token embeddings → Q = {$q_1$, $q_2$, …}
- Document token embeddings → D = {$d_1$, $d_2$, …}

Then scoring = MaxSim:

$$Score(q, P) = \sum_i \max_j (q_i \cdot d_j)$$

This gives a ColBERT score $S_s$ for each passage.

| Passage | Student score $S_s$ |
|---|---|
| P1 (correct) | 8.7 |
| P2 | 2.5 |
| P3 | -0.2 |

Step 4 — Softmax Over Teacher Scores The teacher's scores are normalized:

$$T_i = softmax(S_{\text{teacher}})$$

Teacher softmax
P1: 0.92
P2: 0.07
P3: 0.01
This is a probability distribution of relevance.

Step 5 — Softmax Over ColBERT Scores

Same over student: $S_i = softmax(S_{\text{student}})$

Student softmax
P1: 0.88
P2: 0.10
P3: 0.02

Step 6 — KL-Divergence Loss

The goal: Make ColBERT's distribution match teacher's distribution.

Loss: $\mathcal{L} = KL(T \parallel S)$

Which expands to: $\sum_i \quad T_i \cdot \log \frac{T_i}{S_i}$

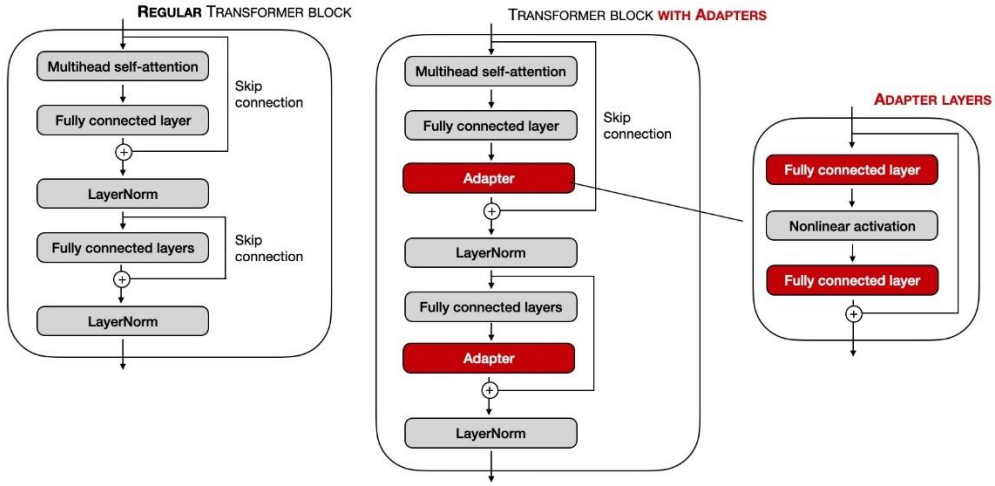
This penalizes ColBERT when it disagrees with teacher.

# WEEK 10: PEFT

For a model of the scale of GPT-3 (175 billion parameters) or LLaMA-65B, FULL FINE-TUNING (FFT) is not merely expensive; it is computationally prohibitive for the vast majority of research and industrial entities. Training a 65 billion parameter model requires storing the model weights, the gradients for each weight, and the optimizer states (e.g., momentum and variance in AdamW), which can necessitate over 780 GB of GPU memory merely to load the training state, far exceeding the capacity of standard high-performance accelerators like the NVIDIA A100 (80GB). Furthermore, the operational logistics of FULL FINE-TUNING (FFT) are daunting. If an enterprise wishes to deploy a model for fifty distinct downstream tasks—ranging from code generation to medical summarization—FULL FINE-TUNING (FFT) would necessitate the storage and management of fifty distinct copies of the massive model weights. This creates a linear scaling of storage costs and introduces significant friction in deployment pipelines, where swapping multi-gigabyte models into VRAM is slow and resource-intensive

PEFT - By freezing the vast majority of the pre-trained weights and targeting updates to a minuscule subset of parameters—or by injecting a small number of new trainable parameters—PEFT methods can achieve performance parity with FFT while reducing the trainable parameter count by up to 10,000 times.

**What an adapter *is* (intuitively)**




An adapter is a very small, trainable neural module that you physically insert *inside* each Transformer layer. Instead of changing the huge pre-trained weights of the model (MHA / FFN), you leave them frozen and only train the small adapter modules. That gives you fine-tuning power while keeping most of the model unchanged.

The adapter's structure (the math)

If the Transformer layer has hidden dimension $d$(e.g., 1,024 or 4,096), the adapter uses a bottleneck of much smaller dimension $r$(e.g., 16, 32, 64). For an input hidden vector $h \in \mathbb{R}^d$:

1. Down-projection: $W_{down} \in \mathbb{R}^{d \times r}$maps $h$into the small space: $z = W_{down}^{\top} h$(notation varies; your excerpt used $W_{down} \cdot h$).
2. Nonlinearity: apply $\sigma$(ReLU/GELU) to $z$.
3. Up-projection: $W_{up} \in \mathbb{R}^{r \times d}$maps back to $d$.
4. Residual add: add that output to the original $h$.

Compactly: $A(h) = W_{up}\, \sigma(W_{down}\, h) + h$.

Because $r \ll d$, the number of trainable parameters is small (roughly $2dr$per adapter), so only a few percent of the full model's parameters are updated.

Where adapters are placed — two common choices

Both placements insert adapters inside the Transformer block but differ in *how many* and *where*.

- Houlsby (two adapters per layer)
  - One after the Multi-Head Attention (MHA) projection and one after the FFN.
  - More expressive (two intervention points) → often slightly better adaptation capability.
  - But doubles the number of adapter blocks (e.g., 24-layer model → 48 adapters) — larger parameter budget and more sequential operations.
- Pfeiffer (one adapter per layer)
  - Single adapter per layer (commonly after the FFN, before final layernorm).
  - Roughly half the extra parameters of Houlsby, with similar empirical performance on many benchmarks.
  - Preferred when you want efficiency with near-parity performance.

Modern frameworks (AdapterHub, UniPELT, etc.) can try both positions or learn which to activate per-layer.

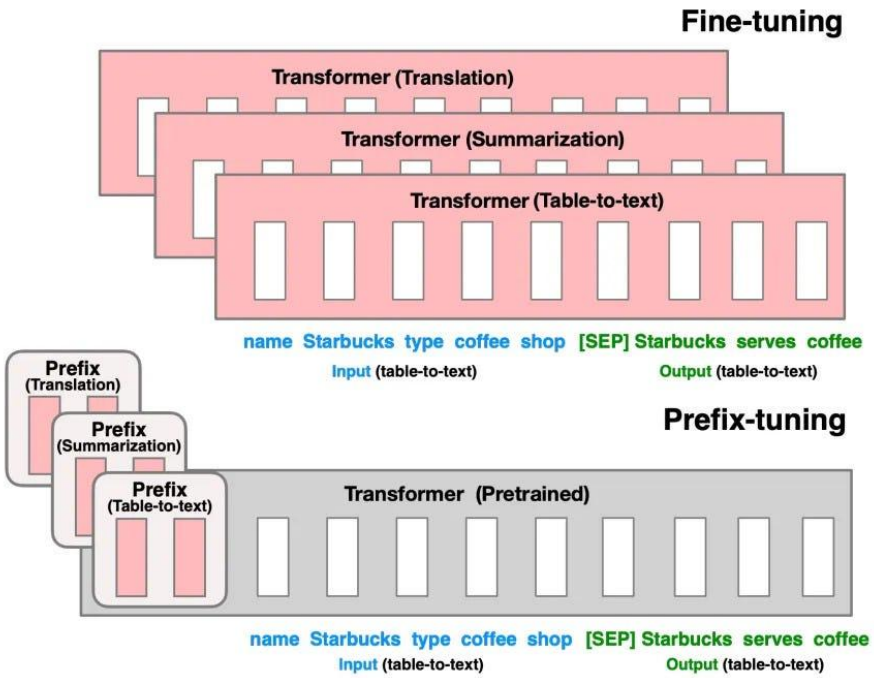
Training dynamics and memory

- Parameter memory (optimizer state): small. Only adapter weights (and maybe LayerNorm) are updated → optimizer states are tiny → you can fine-tune on much smaller GPUs.
- Activation memory: not much reduced. The frozen backbone still needs forward activations to compute gradients that flow back into adapters, so you still store activations for the full model unless you use activation checkpointing.
- Compute/time: adapters make the computational graph deeper (extra forward/backward ops), so each epoch can be slightly slower than prompt-only methods which don't change the network depth.

Inference behavior — the big drawback

- Adapters are additive — they are actual extra layers in the model graph. During inference, inputs must pass through those extra computations, which:
  - Cannot be trivially fused with existing MHA/FFN ops.
  - Launch additional small kernels, which is expensive for small batches (online inference).
- Cumulative latency: one adapter is tiny, but across many layers the added latency accumulates. Measured slowdowns vary (roughly 2–30% depending on model depth, batch size, sequence length, hardware, and kernel-launch overhead).
- Contrast with LoRA: LoRA-style reparameterizations learn low-rank updates that *can be merged* into the base weights before inference, so they don't add runtime layers or latency.

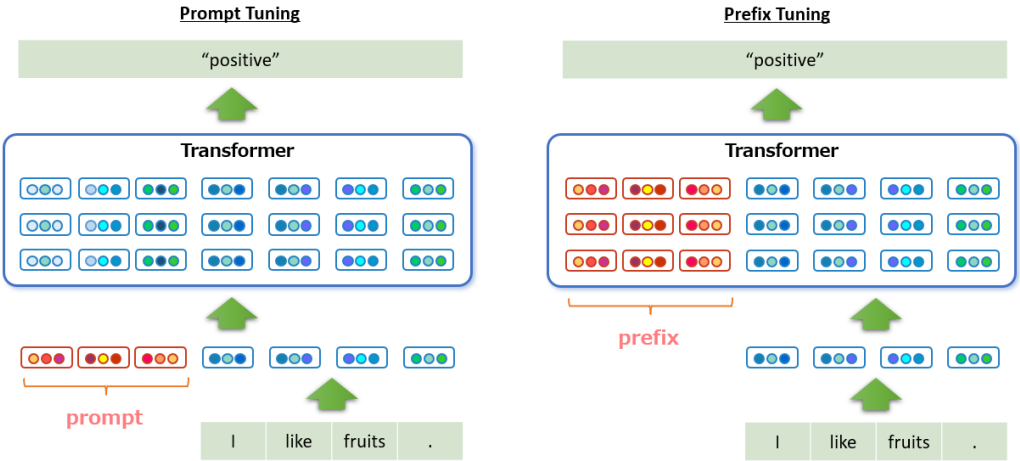
**What an Prompt Tuning & Prefix Tuning**



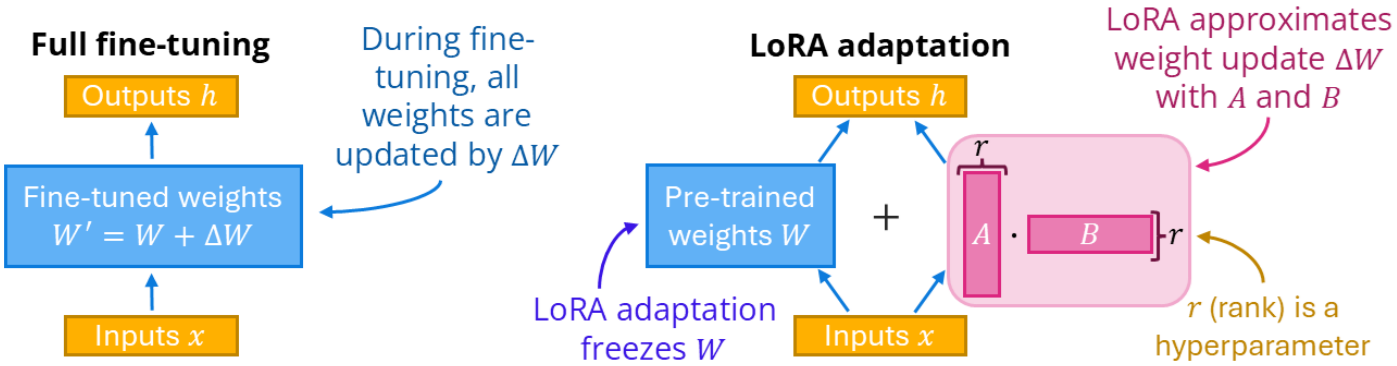

**Prompt tuning** sticks a few trainable *virtual tokens* at the *input embedding* layer. **Prefix tuning** sticks trainable *KV vectors* into the attention mechanism at *every layer*. Both keep the pre-trained model frozen and only learn small extra parameters — but they trade off expressivity, memory, and inference complexity very differently.

**Limitations & operational costs**

1. **Context-window reduction.** Prefix tokens occupy the model's finite context. Example: 4096 token limit − 100 prefix tokens = 3996 tokens available for the user. That subtraction is permanent while using that prefix length.
2. **KV cache explosion.** During autoregressive inference, the KV cache must hold the prefixes for every layer and for every request (since prefixes are layer-specific). This increases memory per request.
3. **Batching & serving complexity.** If different users/tasks use different prefixes, the server can't reuse the same KV cache or easily batch efficiently — could fragment the cache and reduce throughput.
4. **Storage** - Prefixes for all layers must be stored per task; for very long prefixes or many tasks, storage grows



LoRA




**LoRA (Low-Rank Adaptation)** replaces a full weight update $\Delta W$ with a low-rank factorization $BA$. You train only $A, B$ (tiny), keep the backbone frozen, and — critically — can **merge** $BA$ into the base weights before inference so there is **no runtime latency** penalty.
LoRA is applied **only to linear projection matrices**, NOT to the entire layer.
In a Transformer block, these are the main linear layers:
**A. Self-Attention (Multi-Head Attention) - -** Each attention block has four projection matrices:

1. **Wq** — Query projection
2. **Wk** — Key projection
3. **Wv** — Value projection
4. **Wo** — Output projection (post-attention)

These are the matrices actually used in attention computation.
**LoRA is usually attached to - - Wq (Queries)** OR **Wk (Keys)** OR **Sometimes Wv or Wo**
**Wq and Wk give the best results**, so most implementations (HuggingFace, PEFT, LLaMA-LoRA papers) attach LoRA *primarily* to Q & K.
**Why attach LoRA to Q and K specifically?**

- Q and K control **attention patterns** — i.e., *what the model looks at*.
- Small rank updates here can drastically change behavior with very few parameters.

V (values) changes the *content* of representations → less efficient.
O (output) affects all heads → too diffuse.
So for best compute–performance balance:
MATH:
Let $W_0 \in \mathbb{R}^{d\times k}$ be a frozen weight (e.g., the Query projection $W_q$). Instead of learning a full $\Delta W$, LoRA models: $\Delta W = B\,A$
With $B \in \mathbb{R}^{d\times r}, A \in \mathbb{R}^{r\times k}, r \ll \min(d,k)$.
A forward pass for that linear layer becomes: $h = W_0 x + \Delta W x = W_0 x + B\,(Ax)$. A scalar scaling factor $\alpha/r$ is commonly applied so the final contribution is $\frac{\alpha}{r} BA$. $\alpha$ is a tunable hyperparameter used to keep update magnitudes consistent across different $r$.

**3) Initialization & stability**
Practical, stable init used in the LoRA paper (and common practice):

- Initialize $A$ with small Gaussian noise.
- Initialize $B$ to **zero**.
  Zeroing $B$ makes $\Delta W(0) = BA = 0$ at step 0, so the model starts exactly as the pre-trained model — avoids destructive initial perturbations.

**4) Parameter savings — numeric example (step-by-step)**
Example for a $4096 \times 4096$ matrix:

- Full parameters to update: $4096 \times 4096 = 16{,}777{,}216$.
- LoRA with rank $r = 8$ uses $(4096 \times 8) + (8 \times 4096)$.
  - $4096 \times 8 = 32{,}768$.
  - So total LoRA params = $32{,}768 + 32{,}768 = 65{,}536$.
- Fraction of parameters updated:

$$\frac{65{,}536}{16{,}777{,}216} = 0.00390625 = 0.390625\%.$$

- Compression factor:

$$\frac{16{,}777{,}216}{65{,}536} = 256,$$

i.e., **~256× fewer parameters** to train for that matrix.

**5) Inference: why LoRA is special**

Because $\Delta W = BA$ has the *same shape* as $W_0$, you can **merge** the learned update into the base weight:

$$W_{\text{merged}} = W_0 + \frac{\alpha}{r} BA.$$

Once merged:

- The model graph is identical to the original pre-trained model.
- **No extra layers, no additional kernel launches → zero inference latency overhead**.
  This makes LoRA highly attractive for production, latency-sensitive systems.

Two deployment strategies:

1. **Merge before inference** — minimal latency; each task becomes its own model copy (cheap if you can store many merged copies).
2. **Dynamic LoRA serving (preferred in multi-task serving)** — keep $W_0$ in VRAM and **load tiny LoRA tensors** $A, B$ per request (or swap them). Because LoRA tensors are small (megabytes), swapping is fast and allows one GPU to serve many task variants.

**6) Where to apply LoRA in practice**

Common targets:

- Attention projection matrices (Query $W_q$, Value $W_v$; sometimes Key $W_k$ or Output $W_o$).
- Occasionally apply to FFN projection matrices (input/output of FFN) for tasks needing more representational shift.

Typical hyperparameters:

- $r$ (rank): commonly **4, 8, 16, 32** (8 or 16 are popular defaults).
- $\alpha$: scaling constant — set so $\alpha/r$ gives reasonable magnitude; some choose $\alpha = r$ (so scaling ≈1) or tune $\alpha \in \{8,16,32\}$.
- Learning rate: often **higher** than full fine-tuning because few params; try grid around 1e-3 → 1e-4 with AdamW variants.

**7) Training dynamics**

- Only $A, B$ (and perhaps a small LayerNorm or bias) receive gradients → optimizer state is tiny → fits on smaller GPUs.
- Gradients propagate through the frozen $W_0$ only to update $A, B$; training is memory-efficient in terms of optimizer state (but activations still need to be stored unless you use checkpointing).

**KronA (Kronecker Adaptation)**

**KronA** uses Kronecker products to build a huge, expressive weight update from two much smaller matrices. That gives a *much higher effective rank* than LoRA for the same (or smaller) parameter budget, and — like LoRA — the update can be merged into the base weights so inference has zero extra latency. The tradeoff is extra implementation and math complexity during training.

**2) Why this matters vs LoRA (intuition + numbers)**

**LoRA**: $\Delta W = BA$ with $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times d}$. The maximum rank of $\Delta W$ is $r$. Parameter count ≈ $2dr$.

**KronA**: take $d = s^2$. Let $A_k, B_k \in \mathbb{R}^{s\times s}$. Then $\Delta W = A_k \otimes B_k$ is $d \times d$ but stored with only $2s^2$ parameters (since each factor is $s \times s$). Note $s^2 = d$, so parameters ≈ $2d$.

Concrete numeric example (digit-by-digit):

- Let $d = 4096$. Then $\sqrt{d} = 64$.
- **LoRA** with rank $r = 8$:
  - $d \times r = 4096 \times 8 = 32{,}768$.
  - Two matrices ⇒ $2 \times 32{,}768 = 65{,}536$ parameters.
- **KronA** using two $64 \times 64$ factors:
  - Each factor has $64 \times 64 = 4{,}096$ parameters.
  - Two factors ⇒ $2 \times 4{,}096 = 8{,}192$ parameters.
- Ratio: $65{,}536 \div 8{,}192 = 8$. So KronA here uses **8× fewer parameters** than LoRA with $r = 8$.

**Rank difference**

- If $A_k, B_k$ are full rank (rank 64 each), then $\text{rank}\,(\Delta W) = 64 \times 64 = 4096 = d$ → **full rank**.
- LoRA with $r = 8$ yields $\text{rank}\,(\Delta W) \leq 8$.

So KronA can give *far greater expressivity per parameter* in low-param regimes.

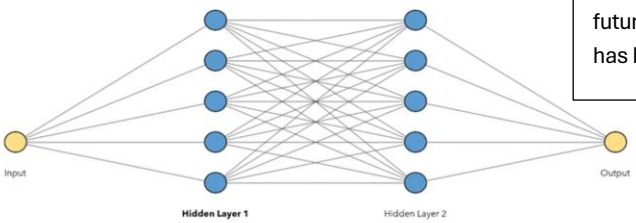
**5) Empirical trade-offs & where KronA shines**

- **When KronA is attractive**
  - Very tight trainable-parameter budgets (you need high expressivity with few params).
  - Tasks where low-rank LoRA updates are too restrictive and you need higher-frequency/complex updates.
  - Small- to medium-sized models where prompt/prefix methods underperform.
- **When LoRA is preferable**
  - Simplicity and existing ecosystem/tools: LoRA is widely supported (HuggingFace/PEFT tools).
  - If rank-tuning is straightforward and parameter budgets are not extremely tight.

**Quantization–** representing model values (weights, activations, sometimes gradients/embeddings) using **lower-precision numbers** (e.g., int8, int4, fp16) instead of 32-bit floating point (FP32).
Goals: reduce model size, memory bandwidth, cache pressure, and often inference latency and energy. The main cost is **accuracy loss** due to finite precision.

# Applying quantization: integer case

Step 4 – Pass to future layers – such that future layer doesn't even realize there has been quantization




**Step 3**

**Dequantize** → **Output** → $Y_q = XW + B$ ← **Perform all operations using integer arithmetic**

**Step 2**

How do we dequantize Y?
Y is the result of an operation, how can we dequantize it since we never calculated its scale (s) and zero (z) parameters?

We basically run inference on the model using a few inputs and "observe" the typical output to calculate the scale (s) and the zero (z). This process is called calibration. **We will see it later with Post-Training Quantization.**

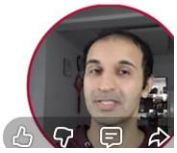
**Input**
Sometimes called "activation".
Can be quantized "on the fly" using a process called (**dynamic quantization**) or with observers.

**Weight**

**Bias**
Usually quantized as int32

**Step 1**

**Quantize**



| | | | | |
|---|---|---|---|---|
| 2484.8 | -323.89 | 0 | 1638.11 | 1999.61 |
| -275.59 | 58.62 | -85.98 | 1187.55 | 1022.13 |
| 2483.8 | 1055.71 | 59.72 | 338.39 | 1442.53 |
| 2021.12 | -322.89 | 2459.82 | 900.31 | 1844.46 |
| 1963.27 | 829.25 | 1352.48 | 567.54 | 1588.66 |

**Quantize**

[-127,127]

We usually *sacrifice* -128 to obtain a symmetric range

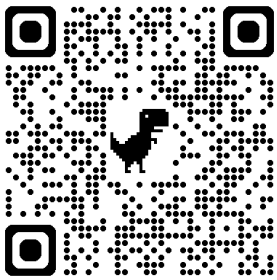
| | | | | |
|---|---|---|---|---|
| 127 | -17 | 0 | 84 | 102 |
| -14 | 3 | -4 | 61 | 52 |
| 127 | 54 | 3 | 17 | 74 |
| 103 | -17 | 126 | 46 | 94 |
| 100 | 42 | 69 | 29 | 81 |

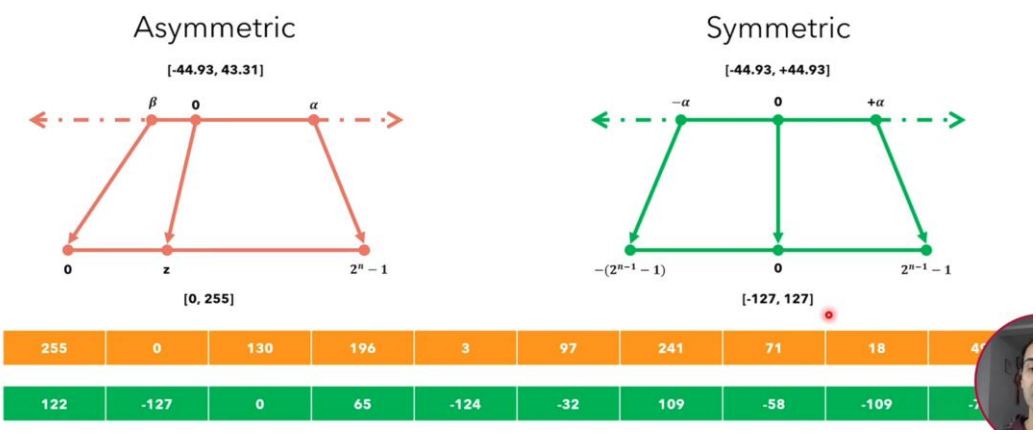
**Dequantize**

| | | | | |
|---|---|---|---|---|
| 2484.8 | -332.61 | 0 | 1643.49 | 1995.67 |
| -273.91 | 58.7 | -78.26 | 1193.49 | 1017.4 |
| 2484.8 | 1056.53 | 58.7 | 332.61 | 1447 |
| 2015.23 | -332.61 | 2465.23 | 900.01 | |
| 1956.54 | 821.74 | 1350.01 | 567.4 | |

# Asymmetric vs Symmetric quantization

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 43.31 | -44.93 | 0 | 22.99 | -43.93 | -11.35 | 38.48 | -20.49 | -38.61 | -28.02 |






| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 255 | 0 | 130 | 196 | 3 | 97 | 241 | 71 | 18 | 4 |
| 122 | -127 | 0 | 65 | -124 | -32 | 109 | -58 | -109 | -1 |

**What is "Activation" in Quantization?**

In the context of quantization:

**Activation = the intermediate output of a layer (the layer's output values).**

It **does NOT** mean:

- ReLU
- GELU
- Softmax
- Any nonlinear function

Instead, it refers to **the floating-point tensor produced by a layer before it goes into the next layer**.

🔶 **STEP 0 — You start with a trained FP32 model** - The model has:

- FP32 weights
- FP32 activations
- FP32 biases

Nothing is quantized yet.

---

🔶 **STEP 1 — (PTQ) Calibration Phase**

Before quantization begins, we must **estimate the dynamic range of activations**. **Why?** Because:

- Weights can be scanned directly → their min & max are known
- But **activations depend on actual data**
- So to know how to quantize each layer's outputs, we must **run sample inputs through the FP32 model** and "observe" min/max.

**What happens?** For each layer:

1. Collect activation values
2. Compute:
   - $x_{\min}$
   - $x_{\max}$
3. Create quantization parameters:
   - scale $s = \frac{x_{\max} - x_{\min}}{q_{\max} - q_{\min}}$
   - zero-point $z = \text{round}(-\frac{x_{\min}}{s})$

These become the **calibration stats**.

👉 After calibration, you know how to quantize:

- Inputs / activations
- Weights
- Biases

---

🔶 **STEP 2 — Quantize Weights & Biases (offline)**

Weights are quantized **once** (static quantization):

**For each weight:**

$$q_w = \text{round}(\frac{w}{s_w}) + z_w$$

Biases are quantized to **int32**:

$$b_q = \frac{b}{s_x \cdot s_w}$$

This keeps math correct.

---

🔶 **STEP 3 — Quantize Input Activation (per layer)**

Before passing into a quantized layer:

$$q_x = \text{round}(\frac{x}{s_x}) + z_x$$

Now both:

- **input** → int8
- **weights** → int8

---

🔶 **STEP 4 — Integer Matrix Multiplication**

This is the main computational saving step.

$$Y_q = X_q W_q + B_q$$

All operations inside are **integer ops** (int8 → int32 accumulator).
This is what makes quantization fast.

---

🔶 **STEP 5 — Dequantize the Result (or quantize output for next layer)**

After integer computation, the output is still integer.
We convert it back to FP32 if needed:

$$\hat{Y} = s_y(Y_q - z_y)$$

Or if next layer is also quantized:

- We DO NOT dequantize
- Instead we **requantize** into int8 for next layer

This keeps the computation chain fully integer.

---

🔶 **STEP 6 — Continue Through Layers**

Each layer repeats:
**Quantize → Integer MatMul → Requantize → Next Layer**
Until you reach the final layer.

---

🔶 **STEP 7 — Final Layer Output (FP32 or int8)**

You can choose:

- **Return FP32 output** → common for NLP
- **Return int8 output** → common for embedded vision models

**Activation quantization is hard** because:

1. Activations change for every input
2. Their distributions vary widely and include outliers
3. You need calibration data to determine scales
4. Errors propagate into all later layers
5. Activation ranges differ across batches/sequences
6. LLM activation outliers frequently break 8-bit quantization

**Range estimation & calibration techniques - used GRID SEARCH**

For good quantization you need reliable min/max or distributional estimates for activations.
Common methods:

- **Min-Max**: take min and max over calibration dataset. Simple but sensitive to outliers.
- **Percentile clipping**: use e.g. 99.9th percentile to ignore rare outliers; reduces dynamic range and quantization noise.
- **KL divergence (histogram search)**: used by TensorFlow for activations — choose clipping threshold that minimizes KL between original and quantized histograms.
- **MSE minimization**: pick clipping bound to minimize mean-squared quantization error.
- **Running statistics**: compute moving averages of min/max during calibration or training.
- **Layer-wise learned scales**: learned via QAT or LSQ (learned scale quantization).

Calibration dataset: small (e.g., 100–1000 samples) representative of real inputs; quality directly affects PTQ results.

| TYPE | TYPICAL BIT WIDTH | TYPICAL RANGE |
|---|---|---|
| Int8 | 1 byte | -127 to 127 |
| UInit8 | 1 byte | 0 to 255 |
| Int32 | 4 bytes | -2147483648 to 2147483647 |
| Unit32 | 4 bytes | 0 to 4294967295 |
| Int64 | 8 bytes | -9223372036854775808 to -9223372036854775807 |
| Unit64 | 8 bytes | 0 to 18446744073709551615 |
| Float | 4 bytes | 1.2E-38 to 3.4E+38 (~6 digits) |
| Double | 8 bytes | 2.3E-308 to 1.7E+308 (~15 digits) |

**affine quantization** Suppose our tensor is: x = [0.5, -1.2, 3.4, 2.1, -0.7, 1.8, 0.0, 4.6, -2.3, 1.1]

Compute: **xmin = -2.3** **xmax = 4.6**

**Step 2 — Compute Scale and Zero-Point** Using min–max affine quantization:

**Scale**

$$s = \frac{x_{\max} - x_{\min}}{q_{\max} - q_{\min}} = \frac{4.6 - (-2.3)}{127 - (-128)} = \frac{6.9}{255} = 0.0270588$$

**Zero-point**

$$z = \text{round}(-\frac{x_{\min}}{s}) = \text{round}(-\frac{-2.3}{0.0270588}) = \text{round}(84.99) = 85$$

So: **scale s = 0.02706** **zero-point z = 85**

✅ **Step 3 — Quantize Each Number**

Use: $q = \text{clip}(\text{round}(x/s) + z,\ -128, 127)$

Let's compute for each:

| x | x/s | round(x/s) | +z | q clipped |
|---|---|---|---|---|
| 0.5 | 18.47 | 18 | 85+18=103 | 103 |
| -1.2 | -44.34 | -44 | 41 | 41 |
| 3.4 | 125.64 | 126 | 211 → capped | **127** |
| 2.1 | 77.61 | 78 | 163 → capped | **127** |
| -0.7 | -25.88 | -26 | 59 | 59 |
| 1.8 | 66.50 | 66 | 151 → capped | **127** |
| 0.0 | 0 | 0 | 85 | 85 |
| 4.6 | 170.0 | 170 | 255 → capped | **127** |
| -2.3 | -85.0 | -85 | 0 | 0 |
| 1.1 | 40.66 | 41 | 126 | 126 |

**Final quantized int8 tensor:**
q = [103, 41, 127, 127, 59, 127, 85, 127, 0, 126]

---

✅ **Step 4 — Dequantization**
Use:

$$\hat{x} = s(q - z)$$

Compute a few:
**Example for q = 103:**

$$\hat{x} = 0.02706 \cdot (103 - 85) = 0.02706 \cdot 18 = 0.487$$

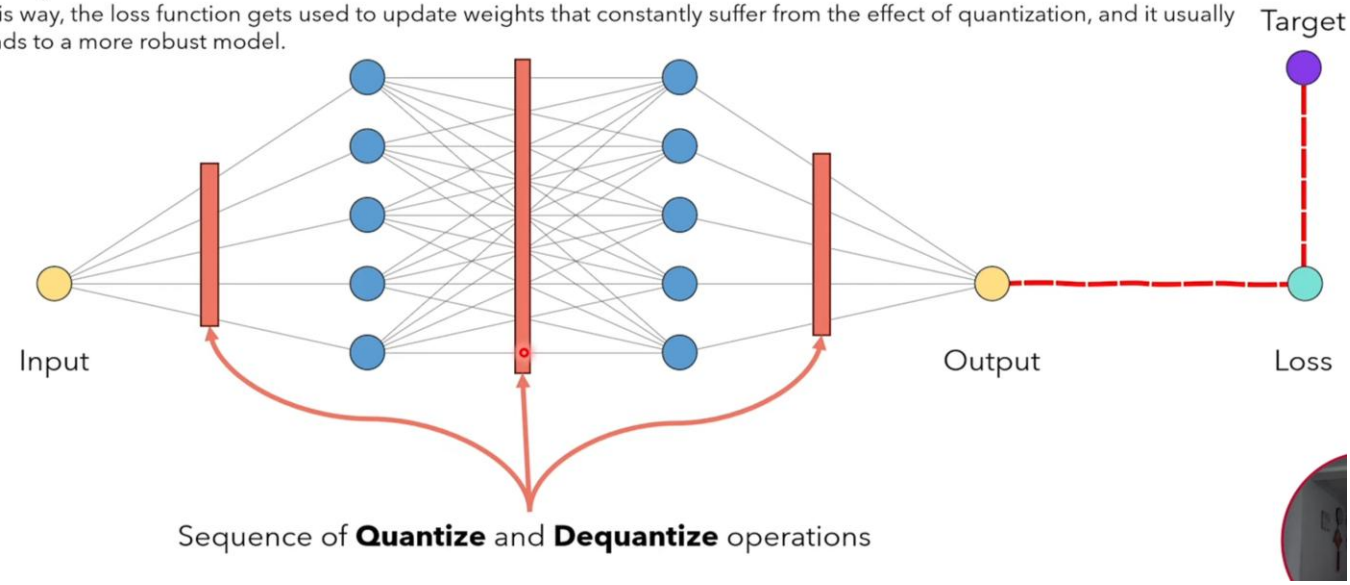
Close to original 0.5.
**For q = 41:**

$$\hat{x} = 0.02706 \cdot (41 - 85) = 0.02706 \cdot (-44) = -1.19$$

Close to -1.2.
**Final dequantized values (approx):**
x_hat ≈ [0.49, -1.19, 1.14, 1.14, -0.70, 1.14, 0.00, 1.14, -2.30, 1.11]
(Clipped values all map to ~1.14 because they saturate at INT8 range.)

# Quantization Aware Training (QAT)

We insert some fake modules in the computational graph of the model to simulate the effect of the quantization during training.
This way, the loss function gets used to update weights that constantly suffer from the effect of quantization, and it usually leads to a more robust model.




Quantization-Aware Training (QAT) is the *most accurate* form of quantization.
Unlike PTQ (Post-Training Quantization), QAT **simulates quantization during training** so that the model *learns to be robust* to the errors caused by low-precision arithmetic.

**PTQ Problems**

- The activations may have **very large dynamic ranges**, causing poor scale selection.
- Important small values may be **rounded to zero** (precision loss).
- Non-linear layers (attention, matmul) accumulate quantization noise.
- Sensitive layers (like attention projections in LLMs or first/last layers in CNNs) degrade sharply.
-

**How QAT Internally Works**
QAT inserts **FakeQuantize() modules** into the model:
**FakeQuantize Forward Pass**
x_fp32 → quantize to int8 range → dequantize back to fp32
The model continues using FP32 values — but with quantization noise injected.
**Quantization simulation:**

$$\text{FakeQuant}(x) = s \cdot (\text{clip}(\text{round}(\frac{x}{s}) + z, q_{\min}, q_{\max}) - z)$$

During forward pass:

- Quantization is simulated
- The model "feels" the rounding errors

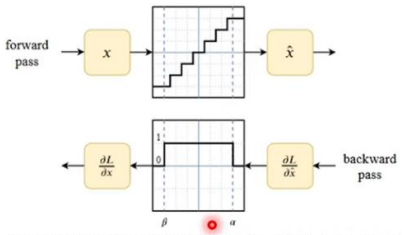
**FakeQuant Backward Pass**
Quantization is *not differentiable* (round, clip).
Therefore, QAT uses:
✓ **STE** — Straight-Through Estimator

- Treat round() as identity during backward
- Ignore clipping outside range (or pass 0 gradient)

Thus, gradients flow *as if quantization does not exist*.

# Quantization Aware Training (QAT): gradient

During backpropagation, the model needs to evaluate the gradient of the loss function w.r.t every weight and input. A problem arises: what is the derivative of the quantization operation we defined before?
A typical solution is to approximate the gradient with the STE (Straight-through Estimator) approximation.




The STE approximation results in 1 if the value being quantized is in the range $[\alpha, \beta]$, otherwise it is 0.

*INTEGER QUANTIZATION FOR DEEP LEARNING INFERENCE: PRINCIPLES AND EMPIRICAL EVALUATION*, Wu et al.

The STE is a hack used during **Quantization-Aware Training (QAT)** to allow backpropagation through *non-differentiable* quantization operations such as:

- **round()**
- **clip()**
- **sign()**
- **argmax()**

These functions have **zero gradients almost everywhere**, so normally **gradient descent would break**.
QAT solves this using STE so the quantizer can be used during forward pass, but gradients can still flow backward.

---

🔥 **Why STE is needed**

Quantization forward pass:

$$q = \text{round}(\frac{x}{s})$$

But:

- **round() derivative = 0** almost everywhere
- So gradient cannot flow to previous layers → network cannot train

**STE bypasses this problem by pretending the quantizer is the identity function during backward pass.**

**QAT Forward Pass**

1. Fake quantize weights (round + clip)
2. Fake quantize activations
3. Do GEMM / Conv using **integer-like values**

**QAT Backward Pass**

1. Pretend quantization didn't happen (STE)
2. Compute gradients normally
3. Update floating-point weights
4. Fake-quantize them again in next forward pass

This teaches the network to **adapt its real FP weights** so that the quantized version performs well.

---

🧩 **Why STE Works (Intuition)**

Quantization is a **projection** onto a discrete grid.
A small gradient step in FP space may completely change the quantized point.
STE allows the model to "see" how its quantized version behaves without blocking gradients.
The FP weights learn to move in a direction that makes the quantized version perform well.