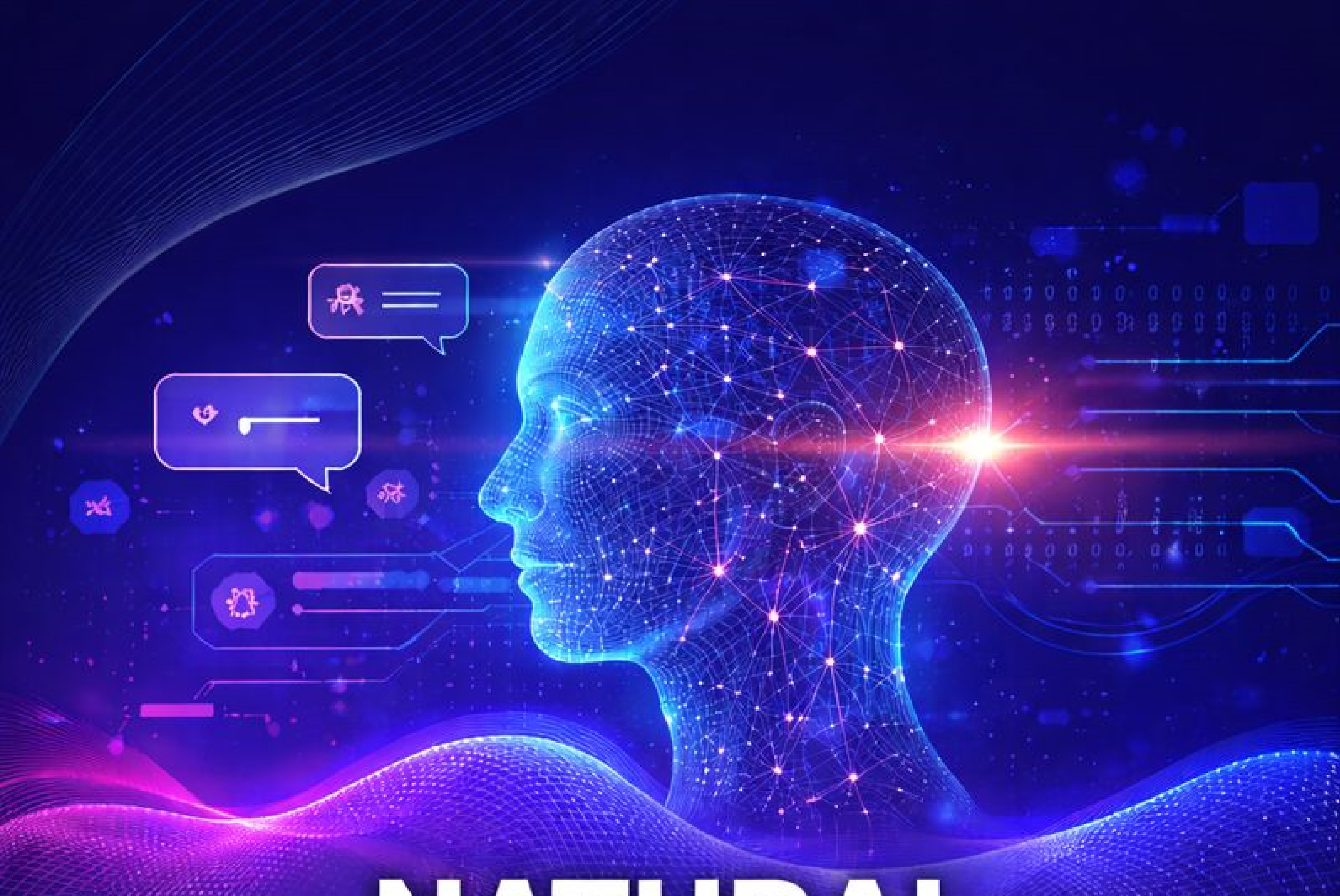
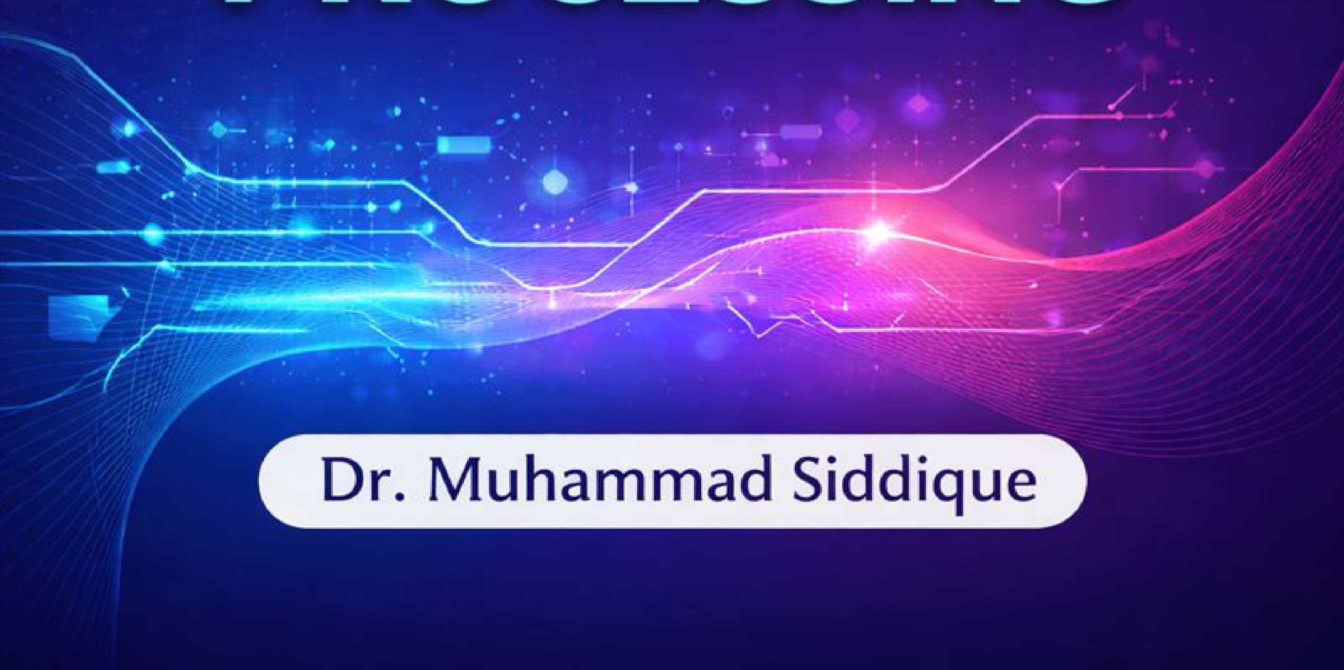


NATURAL LANGUAGE PROCESSING



Dr. Muhammad Siddique

Natural Language Processing

January 30, 2026

Copyright

Published by Muhammad Siddique
Email: msiddique@nfciet.edu.pk

Contents

List of Figures

List of Tables

Chapter 1

Natural Language Processing

Defining NLP

Natural Language Processing (NLP) is the interdisciplinary frontier of Artificial Intelligence that empowers computational systems to bridge the gap between human communication and machine understanding. It involves the automated processing of text and speech to extract meaning, generate responses, and facilitate seamless human-computer interaction.

Human language is perhaps the most complex data format in the AI ecosystem. Unlike structured databases, language is *unstructured, high-dimensional, and inherently ambiguous.* The transition from raw text to machine-interpretable vectors is the core challenge of this field. The Interdisciplinary Nexus NLP does not exist in a vacuum. It represents a sophisticated convergence of four major pillars:

- **Linguistics:** Understanding the rules of grammar, syntax, and meaning.
- **Computer Science:** Efficient algorithms and data structures for processing strings.
- **Probability & Statistics:** Modeling the likelihood of word sequences and context.
- **Machine Learning:** Learning patterns from massive corpora of data.

1.1 The Challenge: Why NLP is "AI-Hard"

While a calculator can process 10^{10} operations per second without error, it struggles to understand a simple sarcastic tweet. This is because natural language is governed by:

1. Structural and Semantic Ambiguity

A single sentence can yield multiple parse trees.

> *"I saw the man with the telescope."*

Does the man have the telescope, or did you use the telescope to see him? This **Syntactic Ambiguity** is a major hurdle for dependency parsing models.

2. Polysemy and Context

Words like "bank" (river bank vs. financial bank) or "charge" (electrical vs. legal) require **Contextual Embeddings** (like BERT or GPT) to resolve their meaning.

3. Evolution and Slang

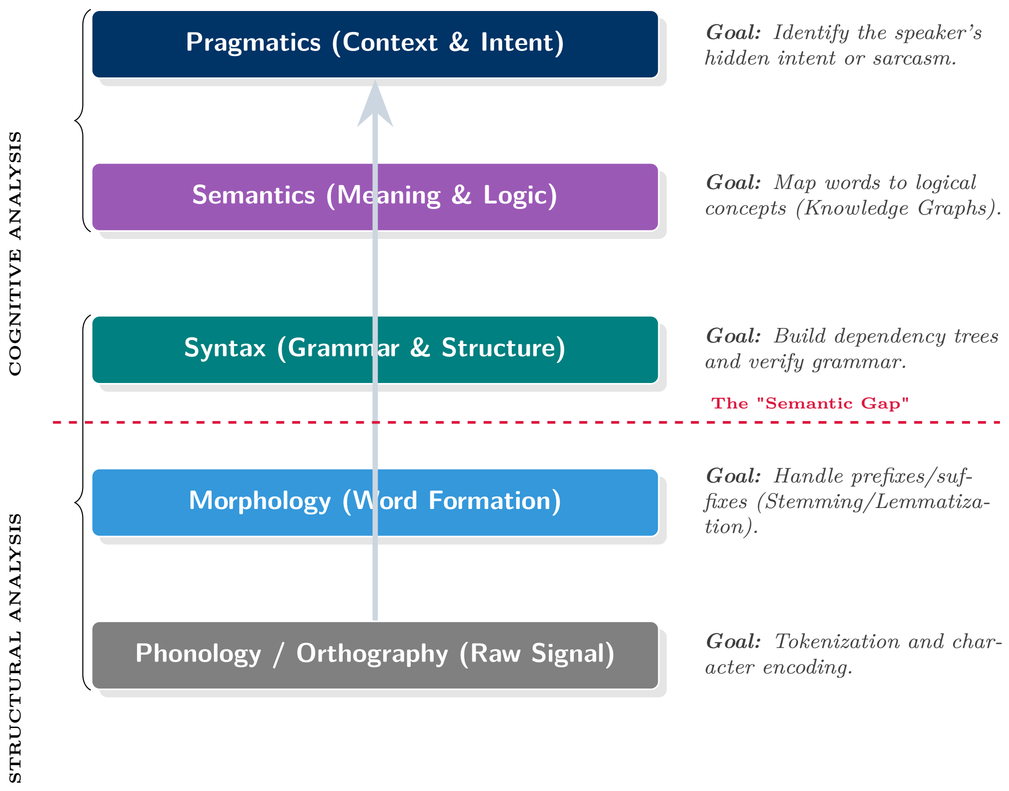
Language is a living organism. New tokens emerge (e.g., "generative AI", "fin-tech"), and meanings shift over time. Static dictionaries are insufficient; models must learn from dynamic data streams.

Core Research Question

How can we transform high-dimensional, ambiguous, and discrete symbolic language into continuous mathematical representations that preserve semantic intent?

1.2 The Architecture of Language Understanding

To process language, we must decompose it into hierarchical layers. For an AI agent to "understand" a sentence, it must navigate through several levels of linguistic abstraction:

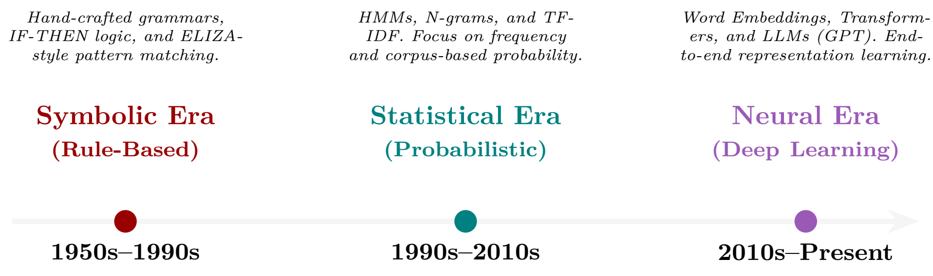

COGNITIVE ANALYSIS
STRUCTURAL ANALYSIS
Pragmatics (Context & Intent)
Goal: Identify the speaker's hidden intent or sarcasm.
Semantics (Meaning & Logic)
Goal: Map words to logical concepts (Knowledge Graphs).
Syntax (Grammar & Structure)
Goal: Build dependency trees and verify grammar.
The "Semantic Gap"
Morphology (Word Formation)
Goal: Handle prefixes/suffixes (Stemming/Lemmatization).
Phonology / Orthography (Raw Signal)
Goal: Tokenization and character encoding.

1.3 The Evolution and Taxonomy of NLP

The journey of NLP is a reflection of the broader history of Artificial Intelligence=moving from rigid logic to fluid, data-driven intuition.

Historical Evolution: From Rules to Reasoning

The transformation of NLP can be categorized into three pivotal eras, each defined by a shift in how machines "perceive" human symbols.


Hand-crafted grammars, IF-THEN logic, and ELIZA-style pattern matching.
HMMs, N-grams, and TF-IDF. Focus on frequency and corpus-based probability.
Word Embeddings, Transformers, and LLMs (GPT). End-to-end representation learning.
Symbolic Era (Rule-Based)
Statistical Era (Probabilistic)
Neural Era (Deep Learning)
1950s–1990s
1990s–2010s
2010s–Present

- **Symbolic Phase:** Relying on expert-defined linguistic rules. While precise, these systems were brittle and failed to capture the "fuzzy" nature of real-world language.

- **Statistical Phase:** Introduced robustness. By treating language as a stochastic process, models could handle noise, though they lacked deep semantic "understanding."

- **Neural Phase:** The current paradigm. Utilizing *Attention Mechanisms* and *Transformers*, models now learn hierarchical representations directly from raw text, achieving near-human performance in many domains.

—

Core Tasks: The NLP Functional Landscape

Modern NLP is categorized into two main branches: **NLU** (Understanding) and **NLG** (Generation). As a AI student, one will encounter these core tasks throughout:

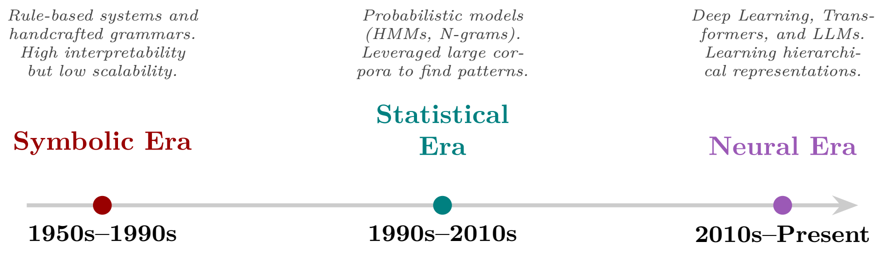
Task Category	Description	Typical Algorithms
Classification	Determining the sentiment or category of a text.	BERT, SVM, Naive Bayes
Sequence Labeling	Identifying entities (NER) or parts of speech (POS).	Bi-LSTM, CRF, Transformers
Translation	Mapping a sequence from Source to Target language.	Seq2Seq, Attention, T5
Summarization	Condensing long documents into key points.	GPT-4, BART, Pegasus
Reasoning	Answering questions based on a given context.	RAG Systems, RoBERTa

Note for AI Practitioners

The success of these complex tasks is rarely achieved by raw data alone. It requires a meticulous pipeline of **Preprocessing** and **Vectorization**=turning human symbols into the high-dimensional tensors that neural networks crave.

Historical Evolution of NLP

The evolution of NLP is a journey from rigid symbolic logic to the fluid, high-dimensional reasoning of modern Neural Networks. This progression is generally categorized into three transformative eras:


Rule-based systems and handcrafted grammars. High interpretability but low scalability.
Probabilistic models (HMMs, N-grams). Leveraged large corpora to find patterns.
Deep Learning, Transformers, and LLMs. Learning hierarchical representations.
Symbolic Era
Statistical Era
Neural Era
1950s–1990s
1990s–2010s
2010s–Present

The NLP Pipeline: Architectural Overview

In modern AI, NLP is treated as a sequence of modular transformations. The goal is to convert unstructured strings into structured tensors that a machine can "understand."

The Intersection of NLP and AI

NLP serves as the **cognitive interface** between human logic and machine computation. It is an indispensable component of the AI ecosystem for three primary reasons:

- **Interaction:** It enables *Human-Computer Interaction (HCI)*, allowing agents to interpret intent from natural speech or text.

- **Multimodality:** Modern AI combines NLP with Vision (CV). For example, in Visual Question Answering (VQA), the model must "understand" a question to "see" the answer in an image.
- **Knowledge Acquisition:** Because most human knowledge is stored in text, NLP techniques are used to build *Knowledge Graphs* and perform large-scale data mining.

Key takeaway

The ultimate challenge of NLP is the **Semantic Gap**=the distance between a word's mathematical representation (a vector) and its true human meaning (context, nuance, and intent).

1.4 Text Preprocessing: Optimizing the Signal-to-Noise Ratio

Text preprocessing is a foundational stage in Natural Language Processing that directly influences the performance, robustness, and interpretability of downstream models. Raw text=harvested from social media, web crawls, or digital documents=is inherently **noisy, inconsistent, and unstructured**.

From an algorithmic perspective, preprocessing bridges the gap between human symbolic language and numerical computation. Because machine learning models operate on high-dimensional tensors rather than raw strings, this stage serves as a critical filter that shapes the linguistic information available for learning.

The AI Practitioner's Goal

The objective of preprocessing is to reduce **superficial variability** (noise) while maximizing the **semantic signal**. Effective preprocessing improves convergence speed and model generalization.

1.4.1 The Anatomy of Raw Text Noise

Raw textual data is heterogeneous and linguistically redundant. one must account for:

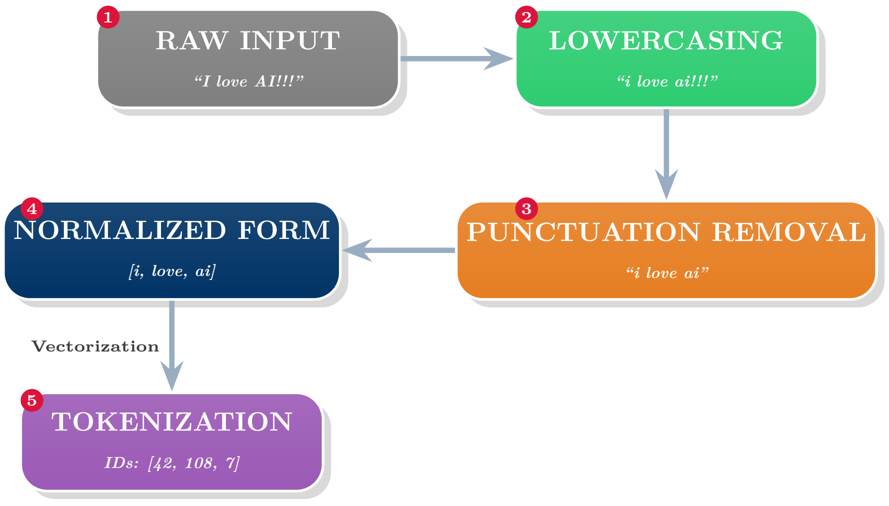
- **Orthographic Variance:** "NLP", "nlp", and "N.L.P." represent the same concept but different byte sequences.
- **Syntactic Noise:** Punctuations, URLs, emojis, and HTML tags that may not contribute to specific tasks like topic modeling.

- **Morphological Redundancy:** Words like "running", "runs", and "ran" share a common root (run).

—

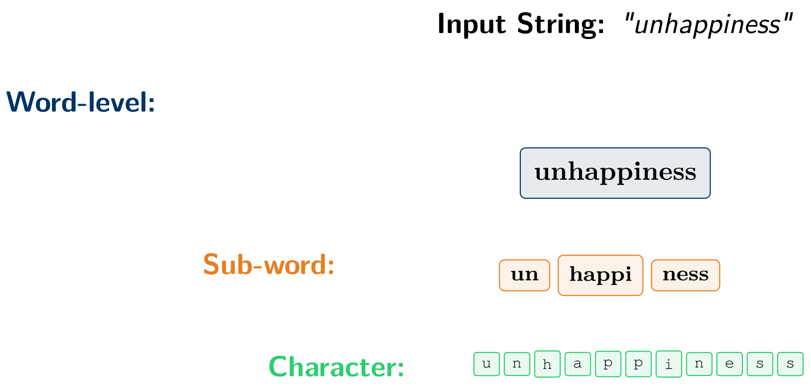
1.4.2 Standardization via Text Normalization

Normalization transforms text into a canonical form. This reduces the **vocabulary sparsity** problem, ensuring that the model does not treat "Data" and "data" as two entirely different features in the vector space.



1.4.3 Tokenization: Defining the Atomic Units

Tokenization is the process of atomizing a continuous stream of text into discrete units called *tokens*. In the AI workflow, this step is critical because it defines the granularity of the model's alphabet. The choice of tokenization directly impacts the vocabulary size, the handling of rare words, and the computational complexity of the model.



As illustrated above, different strategies produce varying numbers of tokens for the same input. The trade-offs are summarized in the following table:

Granularity	Advantages	Limitations
Word-level	High semantic density per token.	Large Vocab; Out-of-Vocabulary (OOV) errors.
Character-level	Fixed, tiny vocabulary; zero OOV issues.	Long sequences; high compute cost; low semantics.
Sub-word (BPE)	**Modern Standard (LLMs)**. Handles rare words efficiently.	More complex preprocessing logic.

Table 1.1: Comparison of Tokenization Granularity

Why Sub-word Tokenization Wins

By using algorithms like **Byte Pair Encoding (BPE)** or **WordPiece**, models can represent the word *"unhappiness"* as a combination of common prefixes (*un-*), roots (*happi*), and suffixes (*-ness*). This allows the AI to "guess" the meaning of words it has never seen before by analyzing their sub-components.

Heuristic Note

While **lowercasing** is standard for Sentiment Analysis, it is often avoided in **Named Entity Recognition (NER)**, where capitalization (e.g., "Apple" the company vs. "apple" the fruit) provides vital semantic cues.

1.4.4 Filtering and Normalization Strategies

Stop Word Removal: The Noise vs. Signal Trade-off

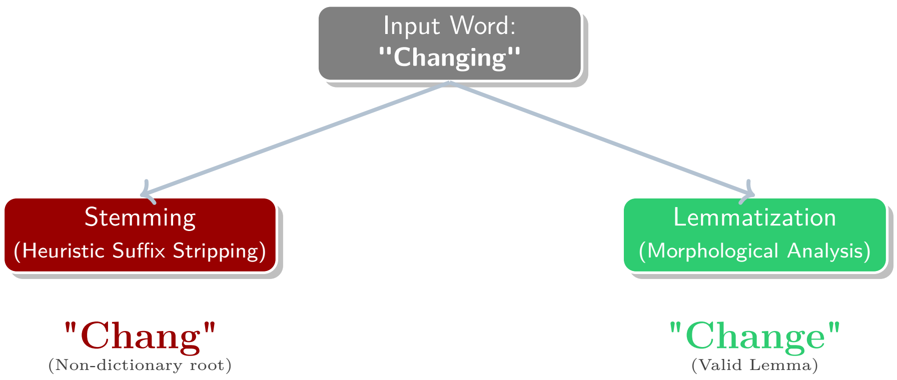
Stop words are high-frequency tokens (e.g., *"the"*, *"is"*, *"and"*, *"in"*) that often carry minimal semantic weight in classical IR (Information Retrieval) tasks.

- **Classical Utility:** In Bag-of-Words (BoW) or TF-IDF models, removing stop words reduces the dimensionality of the feature space and focuses on "content-bearing" terms.

- **Modern Constraint:** In Deep Learning and Transformer-based models, stop words are rarely removed. They provide essential **syntactic cues** and structural context necessary for the *Attention Mechanism* to function correctly.

Stemming vs. Lemmatization

These techniques aim to reduce morphological variants to a common base. However, their algorithmic approaches differ significantly:


Input Word: "Changing"
Stemming (Heuristic Suffix Stripping)
Lemmatization (Morphological Analysis)
"Chang"
(Non-dictionary root)
"Change"
(Valid Lemma)

- **Stemming:** A "chopping" heuristic (e.g., Porter Stemmer). Fast but imprecise; often leads to *over-stemming*.
- **Lemmatization:** Uses a vocabulary and morphological analysis (e.g., WordNet) to return the *Lemma*. It requires Part-of-Speech (POS) context for high accuracy.

1.4.5 Preprocessing as an Integrated Pipeline

In production AI systems, preprocessing is a stateful pipeline. The order of operations is non-trivial; for example, tokenization usually precedes lemmatization because lemmas depend on word-level context.

Design Pattern: "Task-Aware" Preprocessing

There is no "one-size-fits-all" pipeline. For **Social Media AI**, noise handling (emoji mapping/slang correction) is paramount. For **Legal AI**, preserving capitalization and specific punctuation is critical for capturing jurisdictional nuances.

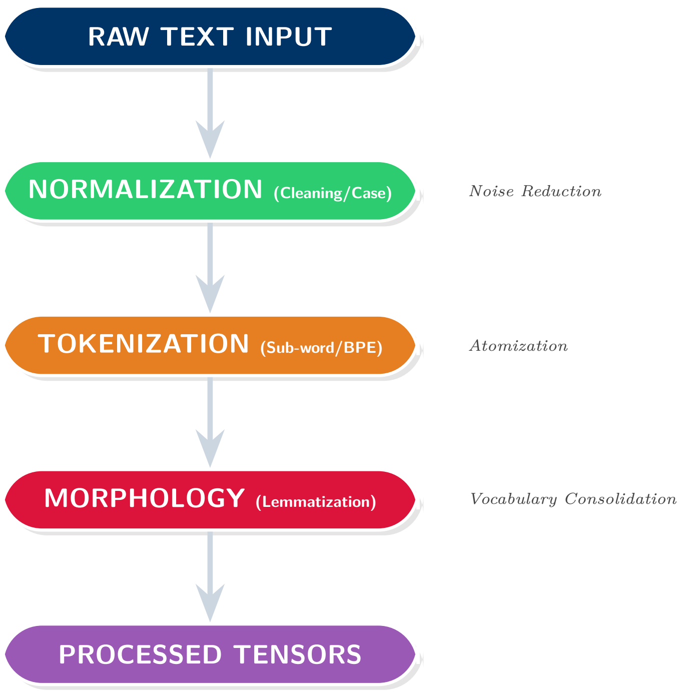

RAW TEXT INPUT
NORMALIZATION (Cleaning/Case)
Noise Reduction
TOKENIZATION (Sub-word/BPE)
Atomization
MORPHOLOGY (Lemmatization)
Vocabulary Consolidation
PROCESSED TENSORS

Figure 1.1: The transformation hierarchy from unstructured text to machine-ready representations.

1.4.6 Practical Implementation: The Preprocessing Logic

For an AI practitioner, the preprocessing pipeline is often the first "layer" of the model. The following implementation uses the industry-standard `NLTK` library to demonstrate a sequential transformation.

Python Implementation: Preprocessing Pipeline

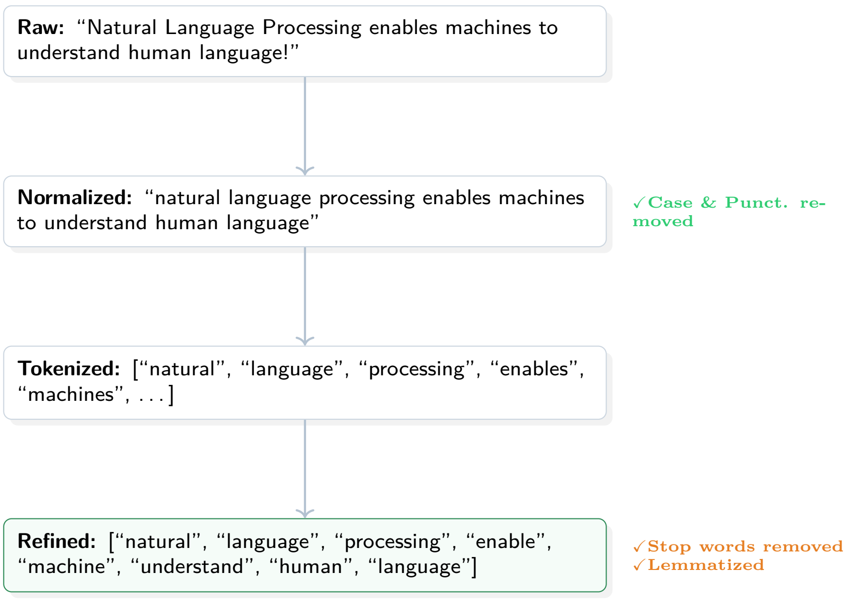
```python
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
# Loading linguistic resources
nltk.download('stopwords')
nltk.download('wordnet')

raw_text = "Natural Language Processing enables
    machines to understand human language!"
```

Python Implementation: Preprocessing Pipeline

```python
# 1. Lowercasing & Noise Removal
clean_text = raw_text.lower()
clean_text = re.sub(r'[^\w\s]', '', clean_text)

# 2. Tokenization (Splitting into atomic units)
tokens = clean_text.split()

# 3. Stop Word Filtering & Lemmatization
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()

processed = [lemmatizer.lemmatize(w) for w in tokens if w not in stop_words]
print(f"Final Representation: {processed}")
```

Visualizing the Data State Change

As a BS AI student, it is important to visualize how the variable `text` evolves in memory:


Raw: “Natural Language Processing enables machines to understand human language!”
Normalized: “natural language processing enables machines to understand human language”
✓Case & Punct. removed
Tokenized: [“natural”, “language”, “processing”, “enables”, “machines”, . . .]
Refined: [“natural”, “language”, “processing”, “enable”, “machine”, “understand”, “human”, “language”]
✓Stop words removed
✓Lemmatized

1.4.7 Preprocessing in the Era of Deep Learning

In modern Deep Learning (DL), the philosophy of preprocessing has shifted. While classical models required aggressive "cleaning" to manage high-dimensional sparse matrices, modern architectures like **Transformers** utilize:

- **Sub-word Tokenization:** Moving away from simple splits to algorithms like *WordPiece*, which handle out-of-vocabulary (OOV) tokens gracefully.
- **Contextual Preservation:** Keeping stop words and punctuation to allow the **Self-Attention** mechanism to calculate the relationship between every token in a sentence.

Pedagogical Summary

Mastering preprocessing provides the foundational intuition for **Linguistic Abstraction**. It is the bridge where the ambiguity of human thought meets the precision of machine computation. By choosing the right preprocessing strategy, you are not just cleaning data=you are *engineering the feature space* in which your AI will live.

1.5 Language Modeling: The Probabilistic Engine of AI

Language models (LMs) constitute the intellectual core of modern Natural Language Processing. Beyond simple string manipulation, an LM is a probabilistic framework that assigns likelihoods to sequences of tokens, enabling machines to reason about linguistic structure, coherence, and intent.

At its core, language modeling addresses a fundamental predictive challenge: *Given a partial sequence of words, what is the most plausible continuation?* This capability provides machines with a form of "computational intuition," allowing them to distinguish between natural human expression and incoherent noise.

The LM Objective

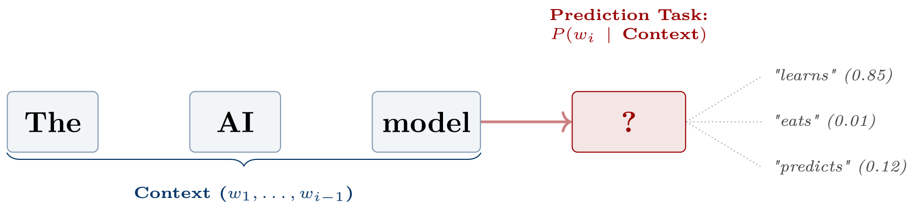
A Language Model seeks to estimate the probability distribution over a sequence of tokens, effectively modeling the statistical structure of human thought as expressed through text.

1.5.1 Foundations: From Sequences to Probabilities

Formally, a language model calculates the joint probability of a sequence of n tokens, denoted as $P(w_1, w_2, \ldots, w_n)$. To make this computationally accessible, we employ the **Chain Rule of Probability**.

$$P(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} P(w_i \mid w_1, \ldots, w_{i-1}) \tag{1.1}$$

This decomposition reveals that the probability of a sentence is the product of the probability of each word given all preceding words.



Perspective

As an AI student, view the Chain Rule not just as a formula, but as a recursive process. The complexity of modern LLMs like GPT-4 arises from their ability to approximate this probability across billions of parameters and trillions of tokens.

1.5.2 The Evolution of the Modeling Paradigm

The history of language modeling is defined by how we represent and process the "context" $(w_1, \ldots, w_{i-1})$:

- **N-gram Models (Classical):** These models assume a *Markov property*, where the prediction only depends on the previous $n - 1$ words. While efficient, they suffer from the "sparsity problem" and cannot capture long-range dependencies.
- **Neural Language Models:** Using word embeddings and hidden states, these models represent context as continuous vectors, allowing for better generalization between similar words.
- **Large Language Models (Transformers):** Modern architectures use *Self-Attention* to weigh the importance of every word in the sequence simultaneously, regardless of distance.

Statistical Language Models: The N-gram Era

Before the advent of deep learning, language modeling was primarily a counting problem. The most influential of these approaches are N-gram models, which simplify the chain rule by making a *Markov Assumption*: that the probability of a word depends only on the immediate $n-1$ preceding words.

The Markov Approximation

A Bigram model ($n = 2$) approximates the probability of a word based solely on its predecessor:

$$P(w_i \mid w_1, \ldots, w_{i-1}) \approx P(w_i \mid w_{i-1})$$

A Trigram model ($n = 3$) extends this window to two preceding words:

$$P(w_i \mid w_1, \ldots, w_{i-1}) \approx P(w_i \mid w_{i-2}, w_{i-1})$$

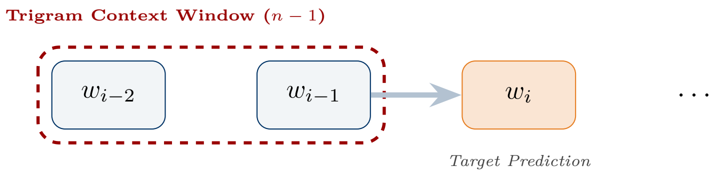


Figure 1.2: The sliding context window in an N-gram model ($n = 3$).

Despite their efficiency, N-grams suffer from the **Data Sparsity** problem: if a specific sequence was never seen during training, the model assigns it a probability of zero, even if it is grammatically correct.

—

1.5.3 Neural Language Models: Learning Continuous Representations

Neural Language Models (NLMs) revolutionized the field by moving from discrete frequency counts to Distributed Representations. Instead of treating words as isolated symbols, NLMs map words into a continuous, high-dimensional vector space (*Embeddings*).

Advantages of Neural Approaches:

- **Generalization:** Because similar words have similar embeddings, the model can generalize context. If it has seen "the cat sat," it can reasonably predict "the dog sat."
- **Increased Context:** Unlike N-grams, which explode in size as n increases, neural models can capture longer dependencies without an exponential increase in parameters.

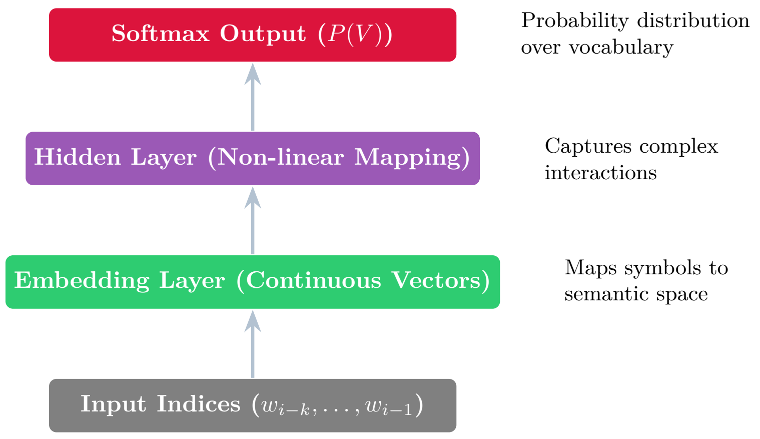

Softmax Output ($P(V)$)
Probability distribution over vocabulary
Hidden Layer (Non-linear Mapping)
Captures complex interactions
Embedding Layer (Continuous Vectors)
Maps symbols to semantic space
Input Indices ($w_{i-k}, \ldots, w_{i-1}$)

Figure 1.3: Conceptual architecture of a Feed-forward Neural Language Model.

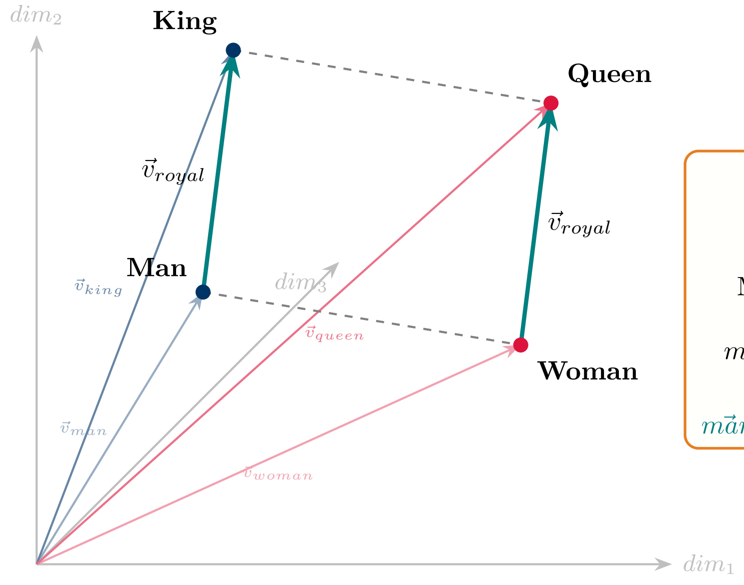
- **End-to-End Learning:** Probability tables are replaced by weights optimized via backpropagation, allowing the model to "discover" linguistic features automatically.

Important Insight

The shift from Statistical to Neural models represents a shift from **Lookup Tables** to **Function Approximation**. We no longer "look up" a probability; we "calculate" it through a learned non-linear mapping.

1.5.4 Word Embeddings: The Geometry of Meaning

A transformative innovation in neural language modeling is the move from *sparse, high-dimensional* representations to **dense, low-dimensional word embeddings**. Words are mapped into a continuous vector space where semantic similarity is encoded as geometric proximity, satisfying the *Distributional Hypothesis*: words that appear in similar contexts share similar meanings.



Vector Arithmetic:

$$\vec{king} - \vec{man} \approx \vec{v}_{royalty}$$
$$\vec{man} + \vec{v}_{royalty} = \vec{king}$$

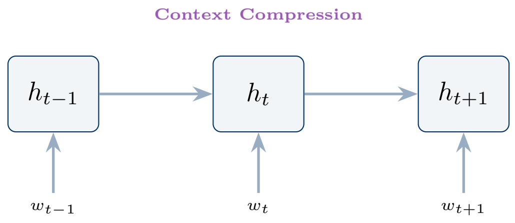
Moving from $\vec{man}$ by the "royalty" vector ($\vec{king} - \vec{man}$) lands you near $\vec{queen}$ if starting from $\vec{woman}$

$$\vec{man} - \vec{woman} \approx \vec{king} - \vec{queen}$$

This geometric structure allows models to generalize. If a model learns that "Paris" is the capital of "France," the underlying vector relationship allows it to infer the capital of "Italy" without explicit instruction.

1.5.5 Recurrent Neural Networks (RNNs): Modeling Temporal Flow

While standard neural networks assume independent inputs, language is inherently sequential. **Recurrent Neural Networks (RNNs)** address this by maintaining a *hidden state* (h_t), acting as a "memory" of all preceding words in a sequence.



The Vanishing Gradient Problem

Traditional RNNs struggle with long-range dependencies (e.g., a subject at the start of a long paragraph affecting a verb at the end). This led to the development of **LSTMs** and **GRUs**, which use "gates" to selectively remember or forget information across long time steps.

1.6 Transformer Architectures: The End of Recurrence

The introduction of the Transformer architecture (Vaswani et al., 2017) marked a definitive shift in language modeling. By abandoning sequential recurrence in favor of Self-Attention mechanisms, Transformers can capture relationships between all tokens in a sequence *simultaneously*. This design enables massive parallelism and mitigates the "bottleneck" issues inherent in RNNs.

The Self-Attention Concept

Unlike RNNs that process text word-by-word, Self-Attention allows each token to "attend" to every other token in the sequence. The model dynamically calculates weight scores to determine which words are most relevant to the current context, regardless of their distance.

1.6.1 Working of the Transformer Architecture

Figure **??** illustrates the working of a decoder-style Transformer used for sentence generation, as in GPT models. The architecture transforms an input sequence through self-attention, feed-forward processing, and autoregressive feedback to predict the next token.

Input Embeddings and Positional Encoding

Each input token w_t is mapped to a dense embedding vector. Since Transformers lack an inherent notion of order, positional encoding is added:

$$\mathbf{X}_t = \text{Embedding}(w_t) + \text{PositionalEncoding}(t).$$

This representation captures both semantic meaning and token position.

Multi-Head Self-Attention

The core of the Transformer is multi-head self-attention. From the input $\mathbf{X}$, queries, keys, and values are computed:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}^K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}^V.$$

Attention weights are obtained using scaled dot-product attention:

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

Multiple heads operate in parallel, capturing different contextual relationships, and their outputs are combined.

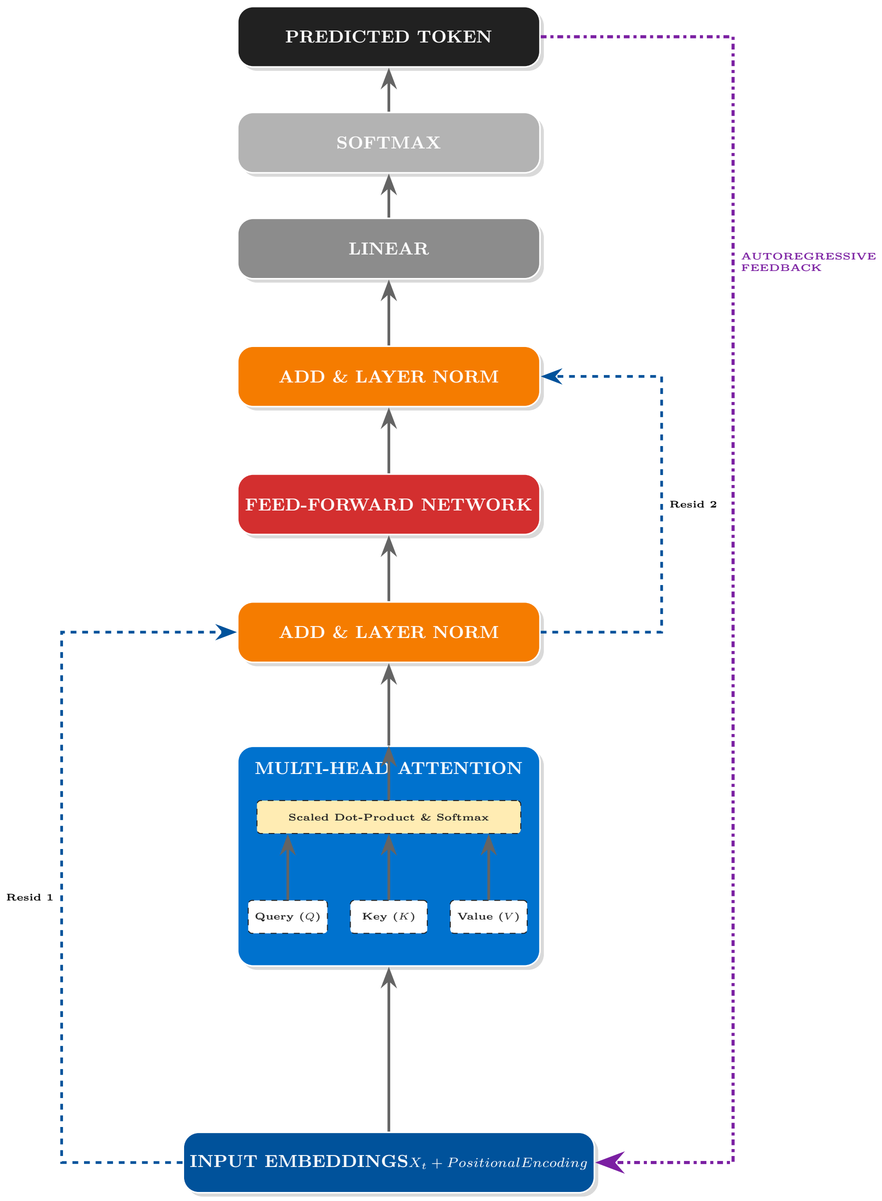


Figure 1.4: Detailed diagram of the Attention Mechanism.

Add & Layer Normalization

Residual connections and layer normalization stabilize training and preserve information:

$$\mathbf{Z}_1 = \text{LayerNorm}(\mathbf{X} + \text{Attention}(\mathbf{X})).$$

Position-Wise Feed-Forward Network

A position-wise feed-forward network applies non-linear transformations to each token independently:

$$\text{FFN}(\mathbf{x}) = \max(0, \mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2.$$

Second Add & Layer Normalization

A second residual connection further refines the representations:

$$\mathbf{Z}_2 = \text{LayerNorm}(\mathbf{Z}_1 + \text{FFN}(\mathbf{Z}_1)).$$

Output Prediction and Autoregression

The final representation is projected to the vocabulary space and normalized using softmax:

$$P(y_{t+1} \mid \mathbf{X}) = \text{Softmax}(\mathbf{Z}_2\mathbf{W}_{\text{vocab}}).$$

The predicted token y_{t+1} is fed back as input for the next time step, enabling autoregressive text generation.

The Transformer combines self-attention, feed-forward networks, and residual normalization to model long-range dependencies efficiently, forming the foundation of modern NLP systems.

1.7 Transformer Case Study: Self-Attention with Positional Encoding

Transformers represent a fundamental shift in how sequential data is processed. Unlike traditional models such as **Recurrent Neural Networks (RNNs)** and **Convolutional Neural Networks (CNNs)**, Transformers process all tokens *in parallel*. While this parallelism makes Transformers extremely fast and scalable, it introduces a critical challenge:

Transformers do not inherently understand word order.

RNNs process tokens sequentially, so order is naturally preserved. CNNs partially capture order using sliding windows. Transformers, however, treat input tokens as a *set*, not a sequence.

To overcome this limitation, Transformers use **Positional Encoding (PE)**, which explicitly injects positional (order) information into token embeddings.

This section presents a detailed Example illustrating how positional encoding is incorporated before the self-attention mechanism.

Problem Setup

Consider the short sentence:

"Deep Learning Models"

The sentence contains three tokens, which are processed simultaneously by the Transformer.

- Sequence length: $L = 3$
- Embedding dimension: $d = 3$

A small embedding dimension is intentionally chosen so that every computation can be clearly followed.

Step 1: Token Embeddings (Semantic Information Only)

Each word is first mapped to a numerical vector called a **word embedding**. These embeddings encode semantic meaning but **do not contain positional information**.

Let the input embedding matrix be:

$$X = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad \begin{array}{l} \leftarrow \text{Deep} \\ \leftarrow \text{Learning} \\ \leftarrow \text{Models} \end{array} \tag{1.2}$$

Each row corresponds to a word, and each column represents one embedding feature.

Key Observation

At this stage, the Transformer cannot distinguish between *"Deep Learning Models"* and *"Models Learning Deep"*. Reordering the rows of X does not change the model's behavior.

This demonstrates why explicit positional information is necessary.

Step 2: Motivation for Positional Encoding

Natural language meaning depends heavily on word order. Sentences containing the same words in different orders often convey entirely different meanings.

Since Transformers rely on matrix multiplications and dot products, positional information must be encoded numerically. This is achieved using positional encodings.

Step 3: Positional Encoding Matrix

Let the positional encoding matrix be $P \in \mathbb{R}^{3\times 3}$:

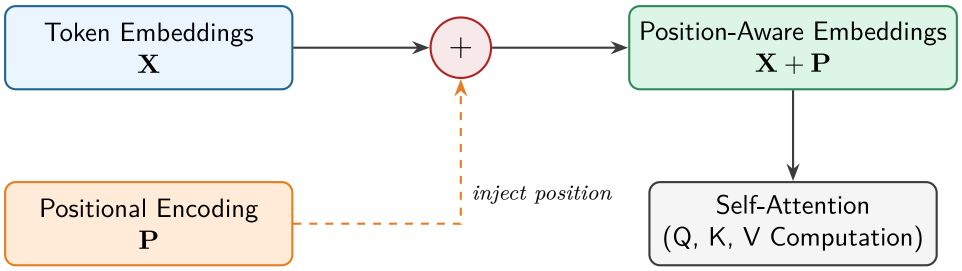


Figure 1.5: Injection of positional encoding into token embeddings.

$$P = \begin{bmatrix} 0.0 & 1.0 & 0.0 \\ 0.5 & 0.5 & 0.5 \\ 1.0 & 0.0 & 1.0 \end{bmatrix} \tag{1.3}$$

Each row corresponds to a unique position in the sequence:

- Row 1: Position 1 (“Deep”)
- Row 2: Position 2 (“Learning”)
- Row 3: Position 3 (“Models”)

Interpretation

Positional encoding assigns each token a unique positional identity, allowing the Transformer to differentiate tokens based on their order in the sequence.

Step 4: Position-Aware Embeddings

The final input to the Transformer is obtained by adding the positional encoding to the word embeddings:

$$X' = X + P = \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 0.5 & 1.5 & 1.5 \\ 2.0 & 1.0 & 1.0 \end{bmatrix} \tag{1.4}$$

This element-wise addition preserves semantic meaning while injecting order information.

Key Insight

Each token embedding now contains:

- Semantic information (what the word means)
- Positional information (where the word appears)

This is the only point at which sequence order enters a Transformer model.

At this stage, the input is fully prepared for the self-attention mechanism, which will be discussed next.

Once the position-aware embedding matrix X' has been formed, the Transformer applies the **self-attention mechanism**. This mechanism allows each token to interact with every other token in the sequence while respecting positional information.

Step 5: Linear Projections to Queries, Keys, and Values

Self-attention begins by projecting the position-aware embeddings X' into three different spaces called:

- Queries (Q)
- Keys (K)
- Values (V)

These projections are learned using weight matrices:

$$W^Q,\ W^K,\ W^V \in \mathbb{R}^{3\times 3}$$

Let the projection matrices be:

$$W^Q = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}, \quad W^K = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}, \quad W^V = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{1.5}$$

The projected matrices are computed as:

$$Q = X'W^Q, \quad K = X'W^K, \quad V = X'W^V \tag{1.6}$$

The core innovation of Self-Attention lies in its use of **three distinct linear projections** of the same input embedding matrix. These projections create three complementary views of each token, enabling context-aware reasoning.

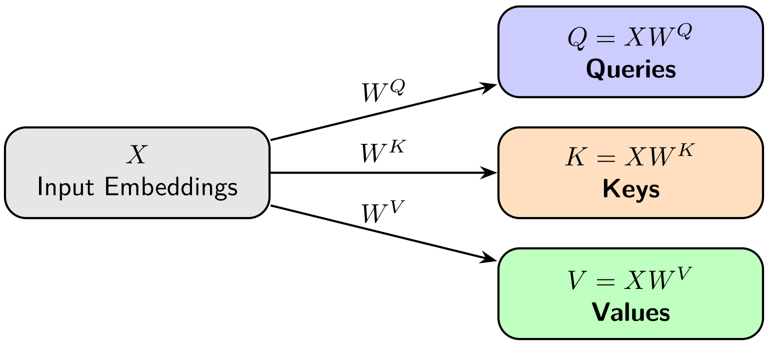


Figure 1.6: The same input embeddings are projected into three different functional spaces using learned weight matrices.

Each projection emphasizes a *different semantic role*:

- Q asks: *What am I looking for?*
- K answers: *How relevant am I?*
- V provides: *What information do I contribute?*

Substituting X' gives:

$$Q = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 3 & 2 \\ 3 & 2 & 3 \end{bmatrix}, \quad K = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 3 & 3 \\ 3 & 3 & 3 \end{bmatrix}, \quad V = \begin{bmatrix} 2 & 3 & 4 \\ 2 & 4 & 4 \\ 3 & 3 & 6 \end{bmatrix} \tag{1.7}$$

The Mechanics and Intuition of Linear Projections

The transformation from input X to Q, K, V via weight matrices W^Q, W^K, W^V is a learned feature extraction process:

- **Queries (Q):** The "Search Term." Represents what a token is currently looking for.
- **Keys (K):** The "Index/Label." Represents what information a token can offer to others.
- **Values (V):** The "Content." The actual data that gets passed on once a match is found.

Concrete Example: The "Library" Analogy Imagine you are searching for a book in a library:

- **Query:** The text you type into the search bar: *"History of Rome."*
- **Key Baker:** The labels on the spines of all the books in the library: *"Ancient Civilizations," "Italian Geography," "Roman Empire."*
- **Value:** The actual pages and information inside the book you eventually pull off the shelf.

The Attention mechanism calculates the similarity between your **Query** and all available **Keys**, then returns a weighted sum of the corresponding **Values**.

Step 6: Attention Score Calculation

To determine how strongly each word should attend to every other word, the dot product between queries and keys is computed:

$$S = QK^\top \tag{1.8}$$

This results in the attention score matrix:

$$S = \begin{bmatrix} 12 & 16 & 18 \\ 16 & 22 & 24 \\ 18 & 24 & 27 \end{bmatrix} \tag{1.9}$$

Each row corresponds to a query token, and each column corresponds to a key token.

Interpretation of Scores

Higher values indicate stronger relevance. For example, the first word ("Deep") attends most strongly to the third word ("Models").

Step 7: Softmax Normalization

Mathematical Summary of Interaction

The relationship can be synthesized in the following operational flow:

$$\text{Attention}(Q, K, V) = \underbrace{\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)}_{\text{The Routing Logic}} \times \underbrace{V}_{\text{The Content}} \tag{1.10}$$

- **Similarity Stage (QK^T):** Determines how much information to "borrow" from each word.
- **Scaling Stage ($\sqrt{d_k}$):** Normalizes the variance of dot products to prevent gradient vanishing.
- **Aggregation Stage (αV):** Combines the retrieved "Values" into a new, context-rich vector.

The final output of self-attention is obtained by weighting the value vectors:

$$Z = \alpha V \tag{1.11}$$

The attention scores are normalized row-wise using the Softmax function:

$$\alpha = \text{Softmax}(S) \tag{1.12}$$

This produces attention weights:

$$\alpha = \begin{bmatrix} 0.02 & 0.24 & 0.74 \\ 0.01 & 0.27 & 0.72 \\ 0.01 & 0.18 & 0.81 \end{bmatrix} \tag{1.13}$$

Each row sums to 1, ensuring a valid probability distribution.

Key Observation

All tokens place the highest attention on the third position. This behavior emerges naturally due to positional encoding and projection geometry.

Step 8: Contextual Output Computation

This yields the contextualized representations:

$$Z = \begin{bmatrix} 2.77 & 3.06 & 5.34 \\ 2.75 & 3.07 & 5.32 \\ 2.83 & 3.00 & 5.49 \end{bmatrix} \tag{1.14}$$

Each row of Z represents a word embedding enriched with **global context**.

Conceptual Summary

Positional encoding introduces order only once=at the input level. From that point onward, self-attention propagates positional effects through dot products and weighted combinations, enabling the Transformer to reason about sequence structure without recurrence.

1.8 How Are W^Q, W^K, and W^V Learned in a GPT Transformer?

One of the most common misconceptions is that the matrices W^Q, W^K, and W^V are *hand-designed* or *manually chosen*. In reality, these matrices are **learned parameters**, discovered automatically through training on massive text corpora.

This section provides a **deep, numerical, step-by-step explanation** of how the values inside the matrices

$$W^Q,\ W^K,\ W^V \in \mathbb{R}^{3\times 3}$$

are formed in a GPT-style Transformer.

We deliberately use a **small 3-dimensional example** so that every multiplication, gradient, and update can be clearly understood.

Key Clarification: These Matrices Are NOT Attention

Before proceeding, it is critical to understand the role of these matrices.

Important Clarification

The matrices W^Q, W^K, and W^V do **not** store attention patterns. Instead, they define **how the model should look at embeddings**.

They are:

- Linear transformations
- Shared across all tokens
- Learned via gradient descent

Attention emerges **after** these projections.

How Learning Happens

GPT is trained using a **next-token prediction objective**.
Given a sequence:

"Deep Learning Models"

GPT learns to predict:

"are powerful"

More generally, GPT minimizes the loss:

$$\mathcal{L} = -\log P(x_{t+1} \mid x_1, x_2, \ldots, x_t)$$

This loss depends on:

- Attention scores
- Attention outputs
- Final logits

And attention depends on:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

Therefore, **the loss indirectly controls the values inside** W^Q, W^K, and W^V.

Step 1: Initialize the Weight Matrices

At the very beginning of training, the matrices are initialized randomly.

For illustration, assume GPT (Transformer) initializes:

$$W^Q_{\text{init}} = \begin{bmatrix} 0.01 & -0.02 & 0.03 \\ 0.00 & 0.02 & -0.01 \\ -0.01 & 0.01 & 0.02 \end{bmatrix} \tag{1.15}$$

$$W^K_{\text{init}} = \begin{bmatrix} -0.02 & 0.01 & 0.00 \\ 0.01 & 0.03 & -0.02 \\ 0.00 & -0.01 & 0.02 \end{bmatrix} \tag{1.16}$$

$$W^V_{\text{init}} = \begin{bmatrix} 0.02 & 0.01 & 0.00 \\ -0.01 & 0.02 & 0.01 \\ 0.01 & 0.00 & 0.03 \end{bmatrix} \tag{1.17}$$

Why Random Initialization?

If all values were zero, every token would behave identically. Randomness breaks symmetry and allows specialization.

These numbers are small because:

- Large values destabilize training
- Gradients must propagate smoothly

Step 2: Training Dynamics of W^Q, W^K, and W^V

In Step 1, we emphasized that the matrices

$$W^Q,\ W^K,\ W^V \in \mathbb{R}^{3\times 3}$$

are **not manually designed**, but are instead **learned from data**. In this section, we explain in detail *how training modifies these matrices* through repeated optimization steps.

This section answers the most important question:

How do random initial matrices become meaningful attention mechanisms?

We proceed slowly, following the exact sequence used during GPT training.

Step 2.1: Initialization of Trainable Parameters

At the beginning of training, GPT initializes all parameters randomly. This includes embeddings, attention weights, feed-forward weights, and output layers.

For our simplified Transformer, assume the following initial values:

$$W^Q_{(0)} = \begin{bmatrix} 0.01 & -0.02 & 0.03 \\ 0.00 & 0.02 & -0.01 \\ -0.01 & 0.01 & 0.02 \end{bmatrix} \tag{1.18}$$

$$W^K_{(0)} = \begin{bmatrix} -0.02 & 0.01 & 0.00 \\ 0.01 & 0.03 & -0.02 \\ 0.00 & -0.01 & 0.02 \end{bmatrix} \tag{1.19}$$

$$W^V_{(0)} = \begin{bmatrix} 0.02 & 0.01 & 0.00 \\ -0.01 & 0.02 & 0.01 \\ 0.01 & 0.00 & 0.03 \end{bmatrix} \tag{1.20}$$

Why Random Initialization?

If all parameters started with identical values (e.g., zeros), all tokens would behave identically, and the model would never learn diverse patterns. Random initialization breaks this symmetry and enables specialization.

Step 2.2: Forward Pass = Computing Q, K, and V

Let the position-aware embedding matrix be:

$$X' = \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 0.5 & 1.5 & 1.5 \\ 2.0 & 1.0 & 1.0 \end{bmatrix} \tag{1.21}$$

The Query matrix is computed as:

$$Q = X'W^Q$$

We compute this **row by row**.

Query for Token 1 ("Deep"):

$$q_1 = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} W^Q_{(0)}$$

$$q_1 = \begin{bmatrix} 0.01 + 0.00 - 0.01, & -0.02 + 0.02 + 0.01, & 0.03 - 0.01 + 0.02 \end{bmatrix} = \begin{bmatrix} 0.00 & 0.01 & 0.04 \end{bmatrix}$$

The same procedure is applied to obtain q_2 and q_3. Identical steps are used to compute K and V.

Important Insight

Every entry in Q, K, and V is a linear combination of the corresponding embedding features, weighted by trainable parameters.

Step 2.3: Attention Scores and Contextual Output

Once Q, K, and V are computed, attention scores are calculated:

$$S = QK^\top$$

These scores measure how strongly each token should attend to every other token. After applying Softmax, we obtain attention weights:

$$\alpha = \text{Softmax}(S)$$

The final attention output is:

$$Z = \alpha V$$

At this stage, the Transformer has produced a **contextual representation** for each token.

However, training does *not* stop here.

Step 2.4: Prediction and Loss Computation

GPT is trained to predict the next token. Let the true next token be represented by a one-hot vector:

$$y = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

Let the model's prediction be:

$$\hat{y} = \begin{bmatrix} 0.2 & 0.3 & 0.5 \end{bmatrix}$$

The training objective uses the cross-entropy loss:

$$\mathcal{L} = -\sum_i y_i \log(\hat{y}_i)$$

Substituting values:

$$\mathcal{L} = -\log(0.5) \approx 0.693$$

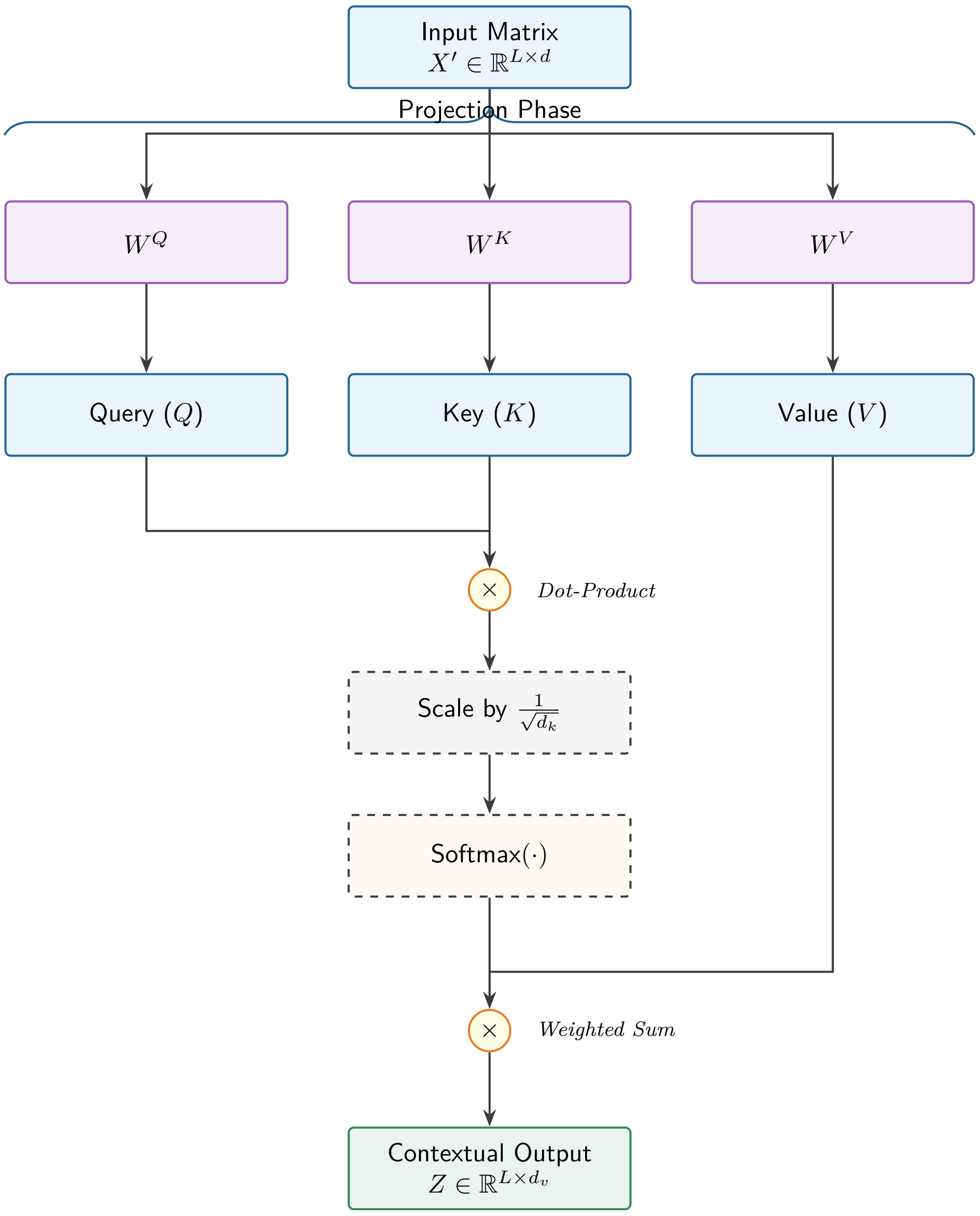


Figure 1.7: Detailed Computational Flow of Self-Attention. The path routing is adjusted to ensure no overlap between lines and computation blocks.

Why Loss Matters

The loss quantifies how wrong the model is. A high loss signals that parameters must change.

Step 2.5: Backpropagation = Computing Gradients

Training now enters the **backward pass**. Gradients of the loss with respect to all parameters are computed using the chain rule.

For the Query weights:

$$\frac{\partial \mathcal{L}}{\partial W^Q} = X'^{\top} \frac{\partial \mathcal{L}}{\partial Q}$$

Each gradient element answers:

How much does changing this number affect the loss?

Assume the computed gradient is:

$$\frac{\partial \mathcal{L}}{\partial W^Q} = \begin{bmatrix} 0.10 & -0.05 & 0.02 \\ 0.08 & -0.02 & 0.01 \\ 0.05 & -0.01 & 0.03 \end{bmatrix} \tag{1.22}$$

Similar gradients are computed for W^K and W^V.

Step 2.6: Gradient Descent Update Rule

Using a learning rate $\eta = 0.1$, parameters are updated:

$$W^Q_{(1)} = W^Q_{(0)} - \eta \frac{\partial \mathcal{L}}{\partial W^Q}$$

$$W^Q_{(1)} = \begin{bmatrix} 0.01 & -0.02 & 0.03 \\ 0.00 & 0.02 & -0.01 \\ -0.01 & 0.01 & 0.02 \end{bmatrix} - 0.1 \begin{bmatrix} 0.10 & -0.05 & 0.02 \\ 0.08 & -0.02 & 0.01 \\ 0.05 & -0.01 & 0.03 \end{bmatrix}$$

$$W^Q_{(1)} = \begin{bmatrix} 0.00 & -0.015 & 0.028 \\ -0.008 & 0.022 & -0.011 \\ -0.015 & 0.011 & 0.017 \end{bmatrix}$$

Key Learning Mechanism

Each update nudges the matrix in the direction that reduces prediction error. Over billions of steps, structure emerges.

Step 2.7: Iterative Refinement Over Training Steps

This process repeats for:

- Millions of sentences

- Billions of tokens
- Hundreds of epochs

Eventually:

- W^Q learns what to ask
- W^K learns what is relevant
- W^V learns what information to pass

At convergence, these matrices encode:

- Syntax
- Semantics
- Long-range dependencies

Step 2.8: Conceptual Summary

Big Picture

W^Q, W^K, and W^V begin as noise. Through loss-driven updates, they become the geometric machinery that enables attention, reasoning, and language understanding in GPT.

Step 3: Forward Pass with a Concrete Input

Consider the position-aware embedding matrix:

$$X' = \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 0.5 & 1.5 & 1.5 \\ 2.0 & 1.0 & 1.0 \end{bmatrix} \tag{1.23}$$

This matrix is passed into the Query projection:

$$Q = X'W^Q \tag{1.24}$$

Let us compute the **first row of Q explicitly**.

Query vector for "Deep":

$$q_1 = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0.01 & -0.02 & 0.03 \\ 0.00 & 0.02 & -0.01 \\ -0.01 & 0.01 & 0.02 \end{bmatrix}$$

$$q_1 = \Big[(1)(0.01) + (1)(0.00) + (1)(-0.01),\ (1)(-0.02) + (1)(0.02) + (1)(0.01),\ (1)(0.03) + (1)(-0$$

$$q_1 = \begin{bmatrix} 0.00 & 0.01 & 0.04 \end{bmatrix}$$

This is how a single row of W^Q numerically influences attention.

Key Insight

Each column of W^Q represents a **learned question pattern**.

Step 4: Why Values Change During Training

After computing attention and predicting the next token, the model computes the loss.

If the prediction is wrong, gradients flow backward:

$$\frac{\partial \mathcal{L}}{\partial W^Q}, \quad \frac{\partial \mathcal{L}}{\partial W^K}, \quad \frac{\partial \mathcal{L}}{\partial W^V}$$

Using gradient descent:

$$W^Q \leftarrow W^Q - \eta \frac{\partial \mathcal{L}}{\partial W^Q}$$

Why These Matrices Become Meaningful

After billions of updates, these matrices learn:

- Which features matter for context
- Which features signal relevance
- Which features should be propagated forward

Summary: Embedding Dynamics

- W^Q, W^K, W^V start as random noise.
- They are refined by prediction errors during backpropagation.
- Their numerical values encode complex linguistic structures.
- Attention patterns (relationships between tokens) emerge directly from these weights.