Natural Language Processing

Sivieri Chiara

2025

Contents

1 Introduction **4**
1.1 What is NLP . . . 4
1.2 Morphology . . . 5
1.3 Lexical Analysis . . . 5
1.4 Syntax . . . 6
1.5 Ambiguities Explode Combinatorially . . . 6
1.6 Semantics . . . 7
1.6.1 Scope Ambiguities . . . 7
1.7 Pragmatics . . . 7
1.8 Discourse . . . 7
1.9 Difficulties . . . 7
1.10 Ambiguity . . . 8
1.10.1 Complexity of Linguistic Representations . . . 8
1.11 Recap . . . 8

2 Words and Tokens **10**
2.1 Regular Expressions . . . 10
2.2 Disjunctions . . . 10
2.2.1 Optional Elements and Wildcards . . . 10
2.2.2 Anchors . . . 11
2.2.3 More Regex Operators . . . 11
2.2.4 Substitution and Capture Groups . . . 11
2.3 Tokenization . . . 11
2.3.1 Issues in Tokenization . . . 12
2.4 Data-Driven Tokenization Algorithms . . . 12
2.4.1 Byte-Pair Encoding (BPE) . . . 12
2.5 Normalization, Lemmatization and Stemming . . . 13
2.5.1 Normalization . . . 13
2.5.2 Lemmatization . . . 13
2.5.3 Stemming . . . 13
2.5.4 Sentence Segmentation . . . 13
2.6 Datasets and Tools . . . 13
2.7 Edit Distance . . . 13
2.7.1 Levenshtein Distance . . . 14
2.7.2 Finding the Distance with Dynamic Programming . . . 14
2.8 Spelling Correction . . . 14
2.8.1 A Simple Application of Edit Distance . . . 14
2.8.2 The Noisy Channel Model . . . 14
2.8.3 The Noisy Channel Spelling Method . . . 14
2.9 Recap . . . 15

3 Language Models **16**
3.1 Introduction to Language Models . . . 16
3.2 Probabilistic Language Models . . . 16
3.2.1 Reminder: The Chain Rule . . . 16
3.3 The Markov Assumption . . . 16
3.4 Unigram Model . . . 17
3.5 Bigram Model . . . 17
3.6 N-Gram Models . . . 18
3.7 Estimating N-gram Probabilities . . . 18
3.7.1 Maximum Likelihood Estimate (MLE) . . . 18
3.8 Evaluating Language Models . . . 18
3.8.1 Extrinsic Evaluation . . . 19
3.8.2 Intrinsic Evaluation and Perplexity . . . 19
3.9 Generality, Sparsity, and Overfitting . . . 19
3.10 Smoothing . . . 20
3.10.1 Laplace (Add-1) Smoothing . . . 20
3.10.2 Add-k Smoothing . . . 20
3.10.3 Backoff . . . 20
3.10.4 Interpolation . . . 20
3.10.5 Kneser-Ney Smoothing . . . 20
3.11 Recap . . . 21

4 Text classification with linear models **22**
4.1 Sentiment Classification . . . 22
4.2 The Classification Task . . . 23
4.2.1 Approaches to Classification . . . 23
4.2.2 Supervised Approaches: Generative vs. Discriminative . . . 23
4.3 Naive Bayes Classifiers . . . 23
4.3.1 Summary: Naive Bayes is Not So Naive . . . 24
4.4 Logistic Regression . . . 25
4.4.1 How Logistic Regression Works . . . 25
4.4.2 Training Logistic Regression . . . 26
4.5 Evaluation Methods . . . 26
4.5.1 Metrics for Binary Classification . . . 26
4.5.2 Test Sets and Cross-Validation . . . 27
4.6 Lexicons for Sentiment, Affect, and Connotation . . . 27
4.6.1 Affective Meaning of Text . . . 27
4.6.2 Theories of Emotion . . . 27
4.7 Recap . . . 28

5 Vector Semantics and Sparse Embeddings **29**
5.1 Lexical Semantics . . . 29
5.1.1 Lemmas and Senses . . . 29
5.1.2 WordNet . . . 29
5.1.3 Word Similarity and Relatedness . . . 30
5.1.4 Connotation and Vector Origins . . . 30
5.2 Distributional Semantics . . . 30
5.3 Embeddings . . . 30
5.3.1 Two Common Models . . . 31
5.3.2 Sparse vs. Dense . . . 31
5.4 Word Vectors . . . 32
5.4.1 Co-occurrence Matrix . . . 32
5.4.2 Vector Comparison: Cosine Similarity . . . 32
5.4.3 A Frequency Paradox . . . 32
5.5 Reweighting . . . 33
5.5.1 TF-IDF (Term Frequency - Inverse Document Frequency) . . . 33
5.6 PPMI (Positive Pointwise Mutual Information) . . . 33
5.6.1 Issues with PPMI . . . 33

5.6.2 Summary of Reweighting . . . 33
5.7 Recap . . . 34

6 Neural language modeling and dense embeddings 35
6.1 From Sparse to Dense Representations . . . 35
6.2 Neural Language Modeling . . . 35
6.2.1 Feedforward Neural Network . . . 35
6.3 Word2vec . . . 36
6.3.1 Model Variants . . . 37
6.4 Evaluating Embeddings . . . 38
6.4.1 Intrinsic Evaluation . . . 38
6.4.2 Extrinsic Evaluation . . . 38
6.4.3 Window Size . . . 38
6.5 Other Embedding Models . . . 39
6.5.1 fastText . . . 39
6.5.2 GloVe (Global Vectors) . . . 39
6.6 Analysing Embeddings: Bias and History . . . 39
6.6.1 Embeddings Reflect Cultural Bias . . . 39
6.7 Recap . . . 39

7 Sequence processing with recurrent networks 41
7.1 Recurrent Neural Networks (RNNs) . . . 41
7.1.1 Simple Recurrent Neural Networks (Elman Networks) . . . 41
7.1.2 Neural Language Modeling with RNNs . . . 42
7.1.3 Training . . . 42
7.1.4 Backpropagation Through Time (BPTT) . . . 43
7.2 Applications . . . 43
7.3 Fancier Architectures . . . 44
7.4 Vanishing Gradient . . . 45
7.5 Advanced Gated Architectures . . . 45
7.5.1 Long Short-Term Memory (LSTM) . . . 45
7.5.2 Gated Recurrent Units (GRU) . . . 45
7.6 Other Solutions . . . 46
7.6.1 Skip Connections (Residual Connections) . . . 46
7.7 Recap . . . 46

8 Seq2seq, CNN, and Transformers 47
8.0.1 The Seq2seq Model . . . 47
8.1 Attention Mechanism . . . 48
8.2 Convolutional Networks for NLP . . . 48
8.2.1 Comparison: RNN vs CNN for Sentence Encoding . . . 49
8.3 Transformers . . . 49
8.3.1 Self-Attention . . . 50
8.3.2 Positional Encoding . . . 50
8.3.3 The Residual Stream . . . 51
8.3.4 Language Modeling Head and Decoder-Only Architecture . . . 51
8.4 Pretrained Language Models . . . 51
8.5 Recap . . . 51

9 Large Language Models (LLMs) 52
9.0.1 Autoregressive Text Completion . . . 52
9.1 Decoding Strategies . . . 53
9.2 Pretraining Large Language Models . . . 54
9.3 Finetuning . . . 54
9.4 Recap . . . 55

10 Masked Language Models **56**
10.1 Bidirectional Transformer Encoders . . . 56
10.2 Training Bidirectional Encoders . . . 57
10.2.1 Masked Language Modeling (MLM) . . . 57
10.3 Attention Mask . . . 58
10.4 Contextual Embeddings . . . 58
10.5 Fine-Tuning for Classification . . . 59
10.6 Fine-Tuning for Sequence Labeling . . . 59
10.7 Recap . . . 59

11 Post-training: Instruction Tuning, Model Alignment, and Test-Time Compute **61**
11.1 Instruction Tuning (Supervised Fine-Tuning - SFT) . . . 61
11.2 Learning from Preferences (Alignment) . . . 61
11.3 In-Context Learning (ICL) and Prompt Engineering . . . 62
11.4 Recap . . . 63

12 Small Language Models **64**
12.1 The Cost of Progress . . . 64
12.2 Approaches to Reducing Model Size . . . 64
12.3 Knowledge Distillation . . . 64
12.4 Vocabulary Transfer . . . 65
12.4.1 Fast Vocabulary Transfer (FVT) . . . 65
12.5 Recap . . . 66

13 Information Retrieval and Question Answering **67**
13.1 Information Retrieval (IR) . . . 67
13.1.1 Sparse Retrieval (TF-IDF & BM25) . . . 67
13.2 Dense Retrieval . . . 67
13.3 Recap . . . 68

1 Introduction

1.1 What is NLP

Natural Language Processing (NLP) is fundamentally about automating the analysis, generation, and acquisition of human (or "natural") language.

The practical applications and motivations for studying NLP are vast and impact many areas of technology and daily life. NLP enables us to:
Answer questions by querying the Web or large collections of documents., translate documents from one language to another, **perform library research and automatically summarize long texts,** estimate public opinion by analyzing large volumes of text and making predictions, follow directions given by a user in natural language, **fix** spelling or grammar **mistakes automatically,** manage messages intelligently for instance by flagging fake news or hate speech, write creative content like poems or novels, listen to user speech and provide advice, write computer code from natural language specifications, create interactive systems to help people learn, assist disabled people, help refugees or disaster victims, and to document or reinvigorate indigenous languages.

To better define the core tasks of NLP, we can introduce the central concept of **representation**. This is a formal structure that captures the information contained in the raw language.

- **Analysis** ("understanding" or "processing"): The input is raw language, and the output is some form of **representation** that supports a useful action.
- **Generation**: The input is a formal **representation**, and the output is natural language.
- **Acquisition**: This is the process of obtaining the **representation** and the necessary algorithms, using both pre-existing knowledge and data.

The crucial question that drives much of NLP is: what does this **representation** look like?

Levels of Linguistic Representation

To understand and process language, we typically break it down into a hierarchy of interconnected levels. The structure flows from raw signal (speech or text) to high-level meaning.

- The highest levels deal with meaning in context:
  - **Discourse**: Analyzing units of language larger than a single sentence, like paragraphs or conversations.
  - **Pragmatics**: Understanding the intended meaning and context that goes beyond the literal words.
- The middle levels deal with sentence-level meaning and structure:
  - **Semantics**: Understanding the literal meaning of words and sentences.
  - **Syntax**: Analyzing the grammatical structure of a sentence.
  - **Lexemes**: Analyzing individual words and their forms (lemmas).
- The foundational level is **Morphology**, the study of the internal structure of words. From morphology, the analysis splits into two paths depending on the input modality:
  - For spoken language: **Phonology** (sound systems) → **Phonetics** (physical properties of sounds) → **Speech**.
  - For written language: **Orthography** (writing systems and spelling) → **Text**.

1.2 Morphology

Morphology is the analysis of words into their meaningful components (morphemes). The complexity of this task varies significantly across different languages.

- **Analytic or Isolating languages** (e.g., English, Chinese) have a relatively simple morphology, where words tend to be single, unchangeable units.
- **Synthetic languages** (e.g., Turkish, Finnish, Hebrew) have a complex morphology, where a single word can contain many morphemes that express complex grammatical relationships.

Examples

- *evlerindekilerin* (Turkish): A single word that can mean "of those that are in their houses."
- in Chinese: "No one needs spaces to separate words," an example of an isolating language.
- *unfriend*, *Obamacare*, *Bill's*: English examples showing prefixes, compounding, and possessives.

1.3 Lexical Analysis

This level deals with individual words, or *lexemes*. Key tasks include:

- Normalizing and disambiguating words.
- Handling words with multiple meanings (polysemy), such as "bank" (a financial institution or a river edge) or "mean" (to signify, or average). An extra challenge comes from domain-specific meanings.
- Identifying multi-word expressions that function as a single unit, like "make a decision," "take out," or "make up."
- For English, a very common form of lexical analysis is **part-of-speech (POS) tagging**, which involves assigning a grammatical category (noun, verb, adjective, etc.) to each word.

1.4 Syntax

Syntax focuses on how to transform a sequence of symbols (words) into a hierarchical or compositional structure that represents its grammar. It is closely related to linguistic theories about what makes a sentence "well-formed."

Example of Well-formed vs. Ill-formed Sentences

- I want a flight to Tokyo (Well-formed)
- I want to fly to Tokyo (Well-formed)
- I found a flight to Tokyo (Well-formed)
- I found to fly to Tokyo (Ill-formed)

Ambiguities in Syntax

A core challenge in syntax is that a single sentence can often have multiple valid grammatical structures, leading to different interpretations. This is called ambiguity.

Prepositional Phrase Attachment Ambiguity

This occurs when it's unclear which word a prepositional phrase (like "with a knife" or "from space") is modifying.

- *"San Jose cops kill man **with knife**"*: Did the cops use a knife to kill the man, or was the man holding a knife when he was killed?
- *"Scientists count whales **from space**"*: Are the scientists located in space while counting, or are they counting whales that are in space?

Verb Phrase Attachment Ambiguity

This is similar, but involves ambiguity in what a verb phrase is modifying.

- *"Mutilated body washes up on Rio beach **to be used for Olympics beach volleyball**"*: Is the beach going to be used for volleyball, or is the body?

Coordination Scope Ambiguity This happens when it is unclear how parts of a sentence joined by a conjunction (like "and" or a comma) are grouped.

- *"Doctor: No heart, cognitive issues"*: Does this mean "(No heart) and (cognitive issues)" or "No (heart disease and cognitive issues)"?

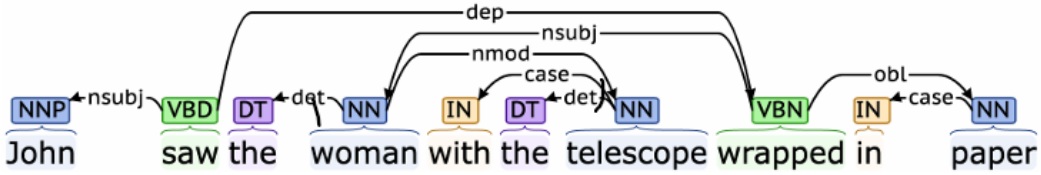
1.5 Ambiguities Explode Combinatorially

The number of possible interpretations for a sentence can grow extremely large as more ambiguous phrases are added. **Example**

Consider the sentence: *"John saw the woman with the telescope wrapped in paper."* This sentence has multiple possible syntactic analyses:

- John used a telescope to see a woman who was wrapped in paper.
- John saw a woman who was holding a telescope that was wrapped in paper.
- John saw a woman who was holding a telescope, and the entire scene was wrapped in paper (less likely, but syntactically possible).

Resolving this requires creating a dependency parse graph, which links words to show their grammatical relationships (e.g., subject, object, modifier) and select the most plausible structure.



1.6 Semantics

Semantics is the study of meaning. In NLP, this involves mapping natural language sentences into formal domain representations that a computer can work with.

- These representations could be a robot command, a database query, or a formal logic expression.
- *A famous quote by Groucho Marx illustrates semantic ambiguity: "In this country a woman gives birth every 15 minutes. Our job is to find that woman, and stop her."*

1.6.1 Scope Ambiguities

This is a key challenge in semantics, where the scope of quantifiers like "every" or "a" is unclear.

- *"A seat is available to every customer."* Does this mean there is one single seat that all customers must share, or that for each customer, there is a seat available?
- *"A web site is available to every customer."* Here, the context makes it clear there is one website for all customers.

A major goal of AI is to build systems that can go beyond specific domains and handle these ambiguities more generally.

1.7 Pragmatics

Pragmatics deals with any *non-local* meaning phenomena—that is, meaning that depends on context, shared knowledge, or conversational goals rather than just the literal words.

- *"Can you pass the salt?"* The literal meaning is a question about ability, but the pragmatic meaning is a request.
- *"Are you 18?" "Yes, I'm 25."* The answer "Yes, I'm 25" pragmatically implies that being 25 satisfies the condition of being over 18.

1.8 Discourse

Discourse analysis is concerned with the structures and effects found in related sequences of sentences, such as texts, dialogues, and multi-party conversations. It examines how sentences connect to form a coherent whole, using concepts like cohesion and coherence. The famous "I have a dream" speech by Martin Luther King, Jr. is a powerful example of structured discourse.

1.9 Difficulties

NLP is a challenging field for several reasons:

- **Noisy Input**: Real-world language data is often messy, with typos, grammatical errors, and slang.
- **Theorized Constructs**: Linguistic representations (like syntax trees) are theoretical models; they are not something we can directly observe in the data.
- **Data Scarcity**: It can be difficult to obtain enough high-quality training data for all the different aspects of language.
- **Complex Mappings**: The relationships and transformations between the different linguistic levels are extremely complex.
- **Application-Dependent Representations**: The best way to represent language depends heavily on the final application.

1.10 Ambiguity

Ambiguity is the single biggest challenge in NLP.

- Each string of words can have many possible interpretations at every linguistic level (syntax, semantics, pragmatics).
- The correct resolution of ambiguity depends on the **intended meaning**, which is often only inferable from context.
- Humans are exceptionally good at resolving linguistic ambiguity using world knowledge and context.
- Computers are not, because this resolution often requires true *understanding*.

1.10.1 Complexity of Linguistic Representations

The nature of language itself contributes to the difficulty.

- **Richness**: There are countless ways to express the same meaning, and an immeasurable number of meanings to express. Language is full of words and phrases with subtle differences.
- **Interactions**: Each linguistic level interacts with the others. A syntactic choice can affect semantic interpretation, which can be clarified by pragmatic context.
- **Diversity**: There is a tremendous diversity in human languages.
  - Languages express the same kind of meaning in very different ways (e.g., via word order, morphology, etc.).
  - Some languages can express certain meanings more readily or more often than others.

1.11 Recap

This chapter sets the stage for the entire field of Natural Language Processing (NLP). The central mission of NLP is to create systems that can **analyze, generate, and acquire human language**. Think of these three tasks as a cycle: a system must first *analyze* (or understand) language to create a meaningful internal **representation**. It can then use this representation to *generate* a response. The process of learning how to create these representations from data is called *acquisition*. The concept of "representation" is the cornerstone of NLP; it's the bridge between messy, ambiguous human language and the structured, logical world of computers.

To build these representations, NLP breaks language down into a **hierarchy of linguistic levels**. This is like a ladder of abstraction. At the very bottom, you have the raw signal: either text (**orthography**) or sound (**phonetics**). The first step up is understanding how words are formed from smaller pieces (**morphology**). Next, we analyze individual words and their roles (**lexical analysis**). From there, we figure out how words combine to form grammatically correct sentences (**syntax**). Once we have the structure, we can determine the literal meaning (**semantics**). Finally, at the top, we consider the context, intent, and conversational flow (**pragmatics** and **discourse**). Each level builds upon the one below it, and crucially, they all interact.

The single greatest obstacle that NLP faces at every single one of these levels is **ambiguity**. A sentence that seems perfectly clear to a human can have dozens of valid interpretations for a computer. The examples of syntactic ambiguity—like *"kill man with knife"*—show that grammar alone isn't enough to determine meaning. We need real-world knowledge. Semantic ambiguity, like scope (*"a seat for every customer"*), further complicates things. This is why NLP is considered a hard AI problem: resolving ambiguity requires a form of **understanding** that goes far beyond simple pattern matching.

Despite these challenges, NLP is incredibly valuable. Its applications are everywhere, from machine translation and search engines to sentiment analysis and generative AI like ChatGPT. The course will equip you with the tools to tackle these problems, starting with foundational techniques for syntax and semantics (like POS tagging and embeddings) and moving all the way up to the modern architectures (**Transformers** and **Large Language Models**) that power the current AI revolution. Understanding

this introductory chapter is key, as it provides the roadmap and the fundamental "why" behind all the techniques you will learn next.

2 Words and Tokens

2.1 Regular Expressions

Regular expressions (regex) are a foundational tool in NLP. They are a formal language used for specifying text strings and patterns. The classic NLP program ELIZA, which simulated a psychotherapist, used simple regular expressions to transform user input into responses, for example, by turning statements with "MY" into questions with "YOUR".

Why use regular expressions?
Regular expressions are essential for three main tasks in NLP:

1. **To describe text patterns**: They allow us to define complex search queries, like finding all instances of a particular word, regardless of capitalization.
2. **To extract information from text**: We can use them to find and pull out specific pieces of data, such as product names followed by prices (e.g., "Stuffed Animals & Toys under $10").
3. **To convert text into a standard form**: This process, known as **text normalization**, involves tasks like tokenization (splitting text into words), lemmatization/stemming (reducing words to their root form), and sentence segmentation.

For instance, to find all variations of the word "woodchuck" (e.g., woodchuck, woodchucks, Woodchuck, Woodchucks), we need a way to specify this pattern concisely. Regular expressions provide that mechanism.

2.2 Disjunctions

Disjunction allows us to specify alternatives in a pattern.

- **Square Brackets [ ]**: To match any single character from a set. For example, '/[wW]oodchuck/' will match both "woodchuck" and "Woodchuck".
- **Ranges [A-Z]**: Inside square brackets, a dash specifies a range of characters. '/[A-Z]/' matches any uppercase letter, while '/[0-9]/' matches any digit.
- **Negation [^]**: A caret '^' at the beginning of a character set negates it, matching any character that is *not* in the set. For example, '/[^A-Z]/' matches any character that is not an uppercase letter. If the caret is not the first character, it is treated as a literal '^'.
- **Pipe Symbol —**: To match one of several complete expressions. For example, '/groundhog—woodchuck/' will match either the full string "groundhog" or the full string "woodchuck".

2.2.1 Optional Elements and Wildcards

These symbols are used to match patterns of variable length.

- **?**: The question mark makes the preceding character optional (it can appear zero or one time). '/colou?r/' matches both "color" and "colour".
- *** (Kleene Star)**: Matches the preceding character zero or more times.
- **+ (Kleene Plus)**: Matches the preceding character one or more times.
- **.**: The period is a wildcard that matches any single character. '/beg.n/' matches "begin", "beg'n", and "begun".

2.2.2 Anchors

Anchors are special characters that match a position rather than a character.

- ^: Matches the start of a line. '/^The/' matches "The" only if it's at the beginning of a line.
- **$**: Matches the end of a line. '/dog./' matches a line ending with "dog.".
- **\b**: Matches a word boundary (the position between a word character and a non-word character). '/the/' matches the word "the" but not "other".
- **\B**: Matches a non-word boundary.

2.2.3 More Regex Operators

- **Common Character Sets**: There are convenient aliases for common character sets. **\d** matches any digit, '**\w** matches any alphanumeric character (plus underscore), and '**\s**:' matches any whitespace character. Their uppercase counterparts (' D', ' W', '§') match the negation of these sets.
- **Counting**: Curly braces '' are used to specify the number of occurrences. 'n' means exactly n times, 'n,m' means from n to m times, and 'n,' means at least n times.
- **Escape Symbol (\)**: A backslash is used to escape a special character and treat it as a literal. To match a literal period, you must use '.'.

2.2.4 Substitution and Capture Groups

Importantly, regular expressions are widely used in substitutions to find and replace text.

- **Capture Groups ()**: Parentheses around part of a regex create a capture group. The substring that matches this part is stored and can be referenced later.
- **Backreferences**: In the replacement string, **\1**, **\2** etc., refer to the content of the first, second, etc., capture group. For example, '/the (.*)er they were, the **\1**er they will be/' could turn "the smarter they were" into "the smarter they will be".
- **Tools**: Regex functionalities are built into most programming languages (like Python's 're' module) and shell utilities like 'sed'.

2.3 Tokenization

Nearly every NLP task begins with text normalization, which involves three key steps:

1. Segmenting or **tokenizing** [1]words in running text.
2. Normalizing word formats (e.g., converting to lowercase).
3. Segmenting sentences in running text.

Before tokenizing, we must define what a "word" is.

- **Lemma vs. Wordform**: A **lemma** is the base or dictionary form of a word (e.g., 'cat'). A **wordform** is the fully inflected surface form (e.g., 'cats'). 'cat' and 'cats' are different wordforms but share the same lemma.
- **Type vs. Token**: A **type** is a unique word in the vocabulary. A **token** is an instance of a type in running text. In "they lay back on the grass", the word "the" is one type, but it appears twice as two tokens.
- We use **N** to denote the total number of tokens and $|V|$ for the vocabulary size (the number of types). **Herdan's Law** describes the relationship between them: $|V| = kN^{\beta}$, showing that the vocabulary grows much slower than the total number of tokens.

[1] A **token** is an instance of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing. It represents a single occurrence of a linguistic unit (like a word or punctuation mark) in running text. Ideally, it is distinguished from a **type**, which is the abstract class representing a unique word in the vocabulary. For example, in the sentence "rose is a rose", there are 4 tokens but only 3 types.

2.3.1 Issues in Tokenization

Tokenization is not as simple as splitting on spaces. Many cases are ambiguous:

- **Punctuation**: How to handle possessives (‘Finland’s‘), clitic contractions (‘what’re‘, ‘I’m‘), and hyphens in compounds (‘state-of-the-art‘)?
- **Ambiguous Symbols**: Periods can mark the end of a sentence or be part of an abbreviation (‘m.p.h.‘, ‘Ph.D.‘).
- **Other Constructs**: Numbers (‘45.50‘)
- **Multi-word Expressions**: Some phrases like ”San Francisco” act as a single unit and might need to be treated as a single token.

Standard and Language-Specific Tokenization
For speed, standard tokenizers often use deterministic algorithms based on regular expressions compiled into efficient finite state automata. However, tokenization rules are often language-specific.

- **French**: Should ‘L’ensemble‘ be one token or two (‘L’‘ and ‘ensemble‘)?
- **German**: Nouns are often compounded into very long words (e.g., ‘Lebensversicherungsgesellschaft-sangestellter‘), which may need to be split by a ”compound splitter” for tasks like information retrieval.
- **Chinese**: Words are composed of characters (hanzi) and are not separated by spaces. Deciding what constitutes a ”word” is complex, and for many tasks, it is better to use characters as the basic input units.

2.4 Data-Driven Tokenization Algorithms

Modern NLP systems often use data-driven tokenization as an alternative to hand-crafted rules. This approach is especially effective for handling unknown words and large vocabularies. The core idea is to break words down into smaller, meaning-bearing units called **subwords**.

- **The Process**: These algorithms use a two-fold scheme. A **token learner** analyzes a large corpus to induce a vocabulary of subwords. Then, a **token segmenter** uses this vocabulary to tokenize new text.
- **Common Algorithms**: The most common algorithms are **Byte-Pair Encoding (BPE)**, **WordPiece**, and **Unigram Language Modeling**.

2.4.1 Byte-Pair Encoding (BPE)

Originally a data compression algorithm, BPE is adapted for word segmentation.

- **Token Learner**: The BPE learner starts with a vocabulary of all individual characters in the corpus. It then iteratively finds the most frequent pair of adjacent tokens (a character ngram) and merges them into a single new token, adding it to the vocabulary. This process continues until a target vocabulary size is reached. For example, ’e’ and ’s’ might be merged into ’es’, and later ’es’ and ’t’ into ’est’.
- **Token Segmenter**: Once the vocabulary and merge rules are learned, the segmenter tokenizes new text by first splitting it into characters and then applying the learned merge rules iteratively. If a token is not found in the vocabulary, it can be represented by the special ‘[UNK]‘ token or left as a sequence of smaller subwords.
- **Multilingual Issues**: BPE tokenizers trained on multilingual data dominated by English tend to dedicate most of their vocabulary to English subwords. This leads to worse segmentation for other languages, resulting in longer token sequences (higher ”fertility”) and lower efficiency.

2.5 Normalization, Lemmatization and Stemming

2.5.1 Normalization

Word normalization is the task of putting words and tokens into a standard format. A common type is **case folding**, which maps all text to lowercase. This is useful for tasks like information retrieval (so "US" and "us" match), but can be detrimental for others like sentiment analysis or machine translation where case can carry important meaning. Another example is standardizing abbreviations, such as ensuring 'U.S.A.' and 'USA' are treated as the same token.

2.5.2 Lemmatization

Lemmatization is the task of reducing inflected or variant forms of a word to its base form, or **lemma**. This is a full morphological analysis.

- Examples: 'am', 'are', 'is' → 'be'; 'cars', 'car's' → 'car'.
- This process allows systems to treat different forms of a word similarly (e.g., a search for "restaurant" should also find documents containing "restaurants").
- Accurate lemmatization requires parsing words into their core meaning-bearing units (**stems**) and the bits and pieces that adhere to them (**affixes**).

2.5.3 Stemming

Stemming is a simpler, cruder method for normalization. It is a naive version of morphological analysis that simply reduces terms to their stems by crudely chopping off affixes.

- This process is language-dependent and rule-based.
- Example: 'automate(s)', 'automatic', 'automation' → 'automat'.
- The **Porter Stemmer** is a widely used algorithm for English that applies cascading rewrite rules (e.g., 'ATIONAL' → 'ATE', 'ING' → ϵ).

2.5.4 Sentence Segmentation

This task, also known as sentence splitting, involves identifying sentence boundaries. The most useful cues are punctuation. Question marks and exclamation points are relatively unambiguous, but periods are more difficult as they can also denote abbreviations. Rule-based systems, like the one in Stanford CoreNLP, are commonly used for this.

2.6 Datasets and Tools

Practical NLP work relies on toolkits and datasets. The **Natural Language Toolkit (NLTK)** for Python is a foundational library that provides tools for tokenization, stemming, lemmatization, and more, as well as access to numerous corpora (text datasets). A common dataset for sentiment analysis is the **IMDB Movie Reviews Dataset**, which contains thousands of positive and negative reviews for training and testing classification models.

2.7 Edit Distance

How similar are two strings?

We often need to measure the similarity between two strings. This is crucial for:

- **Spelling correction**: Is the misspelled word "graffe" closer to "giraffe" or "aardvark"?
- **Coreference resolution**: Are "Università di Bologna" and "University of Bologna" referring to the same entity?

Minimum edit distance

Is a metric that quantifies this similarity. It is defined as the minimum number of edit operations (insertion, deletion, substitution) needed to transform one string into another. This can be viewed as an **alignment problem**.

2.7.1 Levenshtein Distance

The Levenshtein distance is a common implementation of minimum edit distance. A cost is assigned to each operation:

- Insertion: cost 1
- Deletion: cost 1
- Substitution: cost 2 (or 0 if the characters are the same)

2.7.2 Finding the Distance with Dynamic Programming

Calculating the minimum edit distance is a search problem to find the shortest path of transformations between two strings. Because the search space is huge but many paths lead to the same intermediate states, we use **dynamic programming**. The algorithm works by building a matrix where each cell 'D(i,j)' stores the minimum distance between the first 'i' characters of the source string and the first 'j' characters of the target string. The final value in the bottom-right corner of the matrix is the minimum edit distance between the two full strings. By storing back-pointers during the computation, we can also reconstruct the optimal alignment of characters.

2.8 Spelling Correction

2.8.1 A Simple Application of Edit Distance

Spelling correction has two main perspectives:

- **Non-word spelling correction**: The user types a word not found in the dictionary (e.g., "graffe"). We can detect this, generate a ranked list of candidate corrections (e.g., "giraffe"), and rank them based on a distance metric like edit distance.
- **Real-word spelling correction**: The user types a valid word that is incorrect in the given context (e.g., "there" instead of "three"). This is harder to detect, as any word could potentially be an error.

2.8.2 The Noisy Channel Model

A powerful framework for spelling correction is the **Noisy Channel Model**. We imagine that the user's intended correct word, 'w', is passed through a "noisy channel" (the act of typing) and is distorted into the observed (misspelled) word, 'x'. The goal is to find the most probable intended word 'w' given the observed word 'x'. Using Bayesian inference, we want to find the 'w' that maximizes $P(w|x)$. By Bayes' rule, this is equivalent to maximizing:

$$\hat{w} = \underset{w \in V}{\text{argmax}}\ P(x|w)P(w)$$

This breaks the problem into two components:

- $P(w)$ **- The Prior Probability (Language Model)**: How likely is the word 'w' in the language? This is typically estimated from its frequency in a large corpus. More common words get a higher score.
- $P(x|w)$ **- The Channel Model (Error Model)**: What is the probability that the user typed 'x' when they meant to type 'w'? This can be estimated based on edit distance. Errors with fewer edits are more likely. We can refine this using a **confusion matrix** that tracks the frequency of specific typos (e.g., substituting 'e' for 'o').

2.8.3 The Noisy Channel Spelling Method

The complete process is:

1. **Generate Candidates**: For a given input word, find all words in the dictionary with a small edit distance (e.g., Damerau-Levenshtein distance of 1, which also includes transpositions).

2. **Score Candidates**: For each candidate, calculate its score by multiplying its language model probability $P(w)$ with its channel model probability $P(x|w)$.

3. **Choose the Best**: The candidate with the highest score is the suggested correction.

This model can be improved by using a more sophisticated language model that considers the surrounding context (e.g., bigram probabilities like $P(w_i|w_{i-1})$) instead of just unigram frequency.

2.9 Recap

This chapter, "Words and Tokens," dives into the fundamental first steps of any NLP pipeline: taking raw, unstructured text and breaking it down into clean, manageable units. The journey starts with **Regular Expressions**, the essential tool for pattern matching. They are the low-level "scissors and glue" we use to find, extract, and manipulate strings, forming the basis for more complex tools.

The first major task is **Tokenization**, the process of segmenting text into words or "tokens." We quickly learn this is not as simple as splitting on whitespace. The chapter highlights numerous challenges, from handling punctuation and contractions in English to dealing with compound nouns in German and the absence of spaces in Chinese. This complexity motivates the shift from simple rule-based tokenizers to modern, **data-driven subword tokenization** methods like **Byte-Pair Encoding (BPE)**. BPE learns to merge frequent character sequences from a corpus, creating a vocabulary that can efficiently represent any word, even those not seen during training. This is a crucial innovation for modern large language models.

Once we have tokens, we need to standardize them, a process broadly called **Normalization**. The chapter presents a spectrum of techniques. On one end is simple **case folding**. A more linguistically motivated approach is **Lemmatization**, which uses morphological analysis to find a word's dictionary root (e.g., 'are' $\rightarrow$ 'be'). On the other, cruder end is **Stemming**, a heuristic approach that simply chops off word endings. The choice between them is a classic trade-off: lemmatization offers accuracy at the cost of complexity, while stemming provides a fast but less precise alternative.

Finally, the chapter addresses the reality of noisy text by introducing a way to measure string similarity: **Minimum Edit Distance** (specifically Levenshtein distance). This algorithm, powered by dynamic programming, formally calculates how many edits (insertions, deletions, substitutions) are needed to transform one string into another. This metric becomes the core of a powerful application: **Spelling Correction**. The chapter introduces the **Noisy Channel Model**, a foundational probabilistic framework in NLP. It elegantly reframes spelling correction as finding the most likely intended word by combining two probabilities: the **Language Model** ($P(w)$, how common is a word?) and the **Channel/Error Model** ($P(x|w)$, how likely is this specific typo?). This model illustrates a key theme in NLP: combining linguistic knowledge (what words are likely) with real-world data (what errors people make) to solve complex problems.

3 Language Models

3.1 Introduction to Language Models

The primary goal of a language model is to define the probability of a word occurring in a given context, which we can denote as $P(w|c)$. We have already seen a potential use for this in spelling correction, where the language model component $P(w|c)$ helps us decide which candidate correction is more likely.

Language modeling is, in fact, a fundamental concept in NLP with many intuitive applications, including:

1. Speech recognition
2. Augmentative and Alternative Communication (AAC) Systems
3. Natural language generation
4. Text classification

3.2 Probabilistic Language Models

In most cases, the context 'c' is a sequence of preceding words or tokens. Therefore, a language model allows us to compute the probability of a word given a sequence of prior words, such as $P(w_5|w_1, w_2, w_3, w_4)$.

Another way of putting this is that a language model can compute the probability of an entire sentence or sequence of words, $P(W) = P(w_1, w_2, ..., w_n)$. Any model that computes either the probability of the next word or the probability of a whole sequence is called a **language model**.

3.2.1 Reminder: The Chain Rule

To calculate the probability of a sequence of words, we rely on the chain rule of probability. Using the notation w_1^n to indicate a sequence of n words $(w_1, w_2, ..., w_n)$, the chain rule states:

$$P(w_1^n) = P(w_1)P(w_2|w_1)P(w_3|w_1^2)...P(w_n|w_1^{n-1})$$

This can be written more compactly as:

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

For example, the probability of the sentence "its water is so transparent" would be calculated as:

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

How to Estimate these Probabilities

A naive approach might be to just count and divide, using the Maximum Likelihood Estimate (MLE):

$$P(\text{the}|\text{its water is so transparent that}) = \frac{\text{Count(its water is so transparent that the)}}{\text{Count(its water is so transparent that)}}$$

However, this is not feasible. The number of possible sentences in a language is infinite or practically boundless. We will never see enough data to reliably estimate the probabilities of these long sequences. This is the problem of **data sparsity**.

3.3 The Markov Assumption

To overcome the problem of data sparsity, we introduce a simplifying assumption known as the **Markov assumption**[2]. The assumption is that the probability of the next word depends not on the entire history of preceding words, but only on a fixed-size window of the last k words.

[2]Named after Andrey Markov. The assumption simplifies modeling by stating that the future depends only on the present state (or a limited history), not on the entire past sequence.

- For example, we could assume the probability of "the" only depends on the previous word, "that" (a window of $k = 1$):

$$P(\text{the}|\text{its water is so transparent that}) \approx P(\text{the}|\text{that})$$

- Or perhaps it depends on the last two words (a window of $k = 2$):

$$P(\text{the}|\text{its water is so transparent that}) \approx P(\text{the}|\text{transparent that})$$

In general, for $k \geq 1$, the assumption is:

$$P(w_i|w_1^{i-1}) \approx P(w_i|w_{i-k}^{i-1})$$

This allows us to approximate the probability of an entire sentence as:

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

This type of model is called an **N-gram model**, where N refers to the size of the sequence being considered (N = k+1).

3.4 Unigram Model

The simplest case is the unigram model, where we assume the history doesn't matter at all ($k = 0$). The probability of a word is independent of any context.

$$P(w_1^n) \approx \prod_{i} P(w_i)$$

What knowledge does this model embody?

A unigram model only captures the frequency of individual words. It has no knowledge of grammar or word order. We can visualize the knowledge it embodies through sampling. By repeatedly picking words based on their individual probabilities, we can generate "sentences". For example, frequent words like "the" and "of" will occupy large intervals on a probability number line, making them much more likely to be chosen than rare words.

Example Generated Sentences

Sentences generated from a unigram model are typically incoherent collections of words:

- *fifth an of futures the an incorporated a a the inflation most dollars quarter in is mass*
- *thrift did eighty said hard 'm july bullish*

3.5 Bigram Model

In a bigram model, we make the Markov assumption that only the previous word matters ($k = 1$). The probability of a sentence is the product of conditional probabilities of each word given the one that came before it.

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

Example Generated Sentences

Sentences generated from a bigram model show some local structure and are more coherent than unigram sentences, but they still lack long-term consistency:

- *texaco rose one in this issue is pursuing growth in a boiler house said mr. gurria mexico 's motion control proposal without permission from five hundred fifty five yen*
- *outside new car parking lot of the agreement reached*

3.6 N-Gram Models

We can extend this idea to trigrams (k=2), 4-grams (k=3), and so on. However, in general, even these models are insufficient to fully capture the complexity of language. Their main weakness is their inability to handle **long-range dependencies**.

- Consider the sentence: *The computer(s) which I had just put into the machine room on the fifth floor is (are) crashing.*
- The choice between "is" and "are" depends on "computer(s)", which is many words away. An N-gram model with a small N would not be able to capture this relationship.

Despite this limitation, we can often get away with using N-gram models for many practical tasks.

3.7 Estimating N-gram Probabilities

3.7.1 Maximum Likelihood Estimate (MLE)

The standard way to estimate N-gram probabilities is with the Maximum Likelihood Estimate (MLE), which is simply a matter of counting and dividing. For a bigram model:

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

Example: Berkeley Restaurant Project

Consider a corpus of user queries to a restaurant dialogue system. We can compute unigram counts (e.g., 'i': 2533, 'want': 927) and bigram counts from this data. By dividing the raw bigram counts by the corresponding unigram counts of the preceding word, we can obtain the bigram probabilities.

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

What Kinds of Knowledge?

These simple bigram probabilities capture various linguistic phenomena:

- **Cultural/Domain-specific**: $P(\text{chinese}|\text{want}) = .0065$ is higher than $P(\text{english}|\text{want}) = .0011$, reflecting that users in this context are more likely to ask for Chinese food.
- **Syntactic**: $P(\text{to}|\text{want}) = .66$ is very high, capturing the common verb phrase "want to". Similarly, $P(\text{eat}|\text{to}) = .28$ is high. In contrast, ungrammatical sequences like "food to" or "want spend" have zero probability in the data ($P(\text{food}|\text{to}) = 0$).

3.8 Evaluating Language Models

How do we know if our model is good? A good language model should prefer good sentences to bad ones, meaning it should assign a higher probability to "real" or frequently observed sentences than to ungrammatical or rare ones.

- We train the model's parameters on a **training set**.
- We test its performance on an unseen **test set** (also called a held-out set). We must never peek at the test set during development.
- A **development set (devset)** is another held-out set used for tuning hyperparameters.
- A model is considered better if it assigns a higher total probability to the test set.

There are two main approaches to evaluation: extrinsic and intrinsic.

3.8.1 Extrinsic Evaluation

This is also known as end-to-end, in-vivo, or downstream evaluation.

- The only true way to know if an improvement to a component (like our language model) is useful is to see if it helps the final task.
- The process involves putting the language model into a full NLP system (e.g., a spelling corrector or speech recognizer), running the task, and measuring the system's accuracy. We then compare the accuracy of a system with model A versus model B.
- This is the best and most definitive evaluation method, but it is often very costly and time-consuming.

3.8.2 Intrinsic Evaluation and Perplexity

Intrinsic evaluation measures the quality of a component independently of any application. For language models, the most common intrinsic metric is **perplexity**.

- Perplexity is a probability-based metric, but it can be a bad approximation of real-world performance unless the test data looks very similar to the training data.
- For this reason, it is generally only useful in pilot experiments. Any improvements in perplexity should be corroborated by a downstream evaluation before drawing firm conclusions.

Intuition of Perplexity (The Shannon Game): The intuition behind perplexity is how well a model can predict the next word. A better model is one that assigns a higher probability to the word that actually occurs. A unigram model would be terrible at this game because it has no sense of context.

Definition: Perplexity is the inverse probability of the test set, normalized by the number of words (N).

$$PP(W) = P(w_1^N)^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(w_1^N)}} = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}}$$

Perplexity as Branching Factor: Perplexity can be thought of as the weighted average branching factor. If a model for random digits assigns a probability of 1/10 to each digit, its perplexity is 10. This means at each position, the model is as "perplexed" as if it had to choose uniformly among 10 options.

Lower Perplexity = Better Model: Minimizing perplexity is equivalent to maximizing the probability of the test set. On a task with the WSJ corpus, a unigram model might have a perplexity of 962, a bigram 170, and a trigram 109, showing that more context improves the model's predictive power.

3.9 Generality, Sparsity, and Overfitting

Perplexity is always measured on a specific test set. This raises the question of how general these language models are.

- An N-gram model trained on Shakespeare will generate text that looks like Shakespeare. A model trained on the Wall Street Journal will generate financial news text. The models are highly dependent on their training corpus.
- As N increases, N-gram models get better at modeling their specific training corpus.

The Perils of Overfitting

We need robust models that **generalize** well to new, unseen data. Models perform best when the test corpus is similar in genre and dialect to the training corpus. The main problem is **sparsity**:

- N-grams that were not seen in the training corpus are assigned a probability of zero by the MLE.
- However, these n-grams might perfectly well occur in the test set (e.g., "denied the offer" might appear in the test set even if only "denied the allegations" was seen in training).
- If any n-gram in the test set has zero probability, the entire sentence probability becomes zero, and the **perplexity becomes infinite**.

Handling Unknown (OOV) Words

- **Closed vocabulary** systems assume all possible words are known.
- **Open vocabulary** systems must handle unknown or Out-Of-Vocabulary (OOV) words. The standard solution is to create a pseudo-word, '¡UNK¿'. During training, we choose a vocabulary and replace all words not in it with '¡UNK¿'. Alternatively, we can replace all words below a certain frequency threshold.
- This turns an open vocabulary problem into a closed one.
- **Warning**: Using a small vocabulary will artificially reduce perplexity (by making the prediction task easier). Perplexities should only be compared between models that use the exact same vocabulary.

3.10 Smoothing

To solve the zero-probability problem, we must prevent the model from assigning a probability of 0 to unseen events. The core idea is called **smoothing** or **discounting**: we "shave off" a bit of probability mass from the more frequent, seen events and redistribute it to the unseen events. There are various methods to do this.

3.10.1 Laplace (Add-1) Smoothing

The simplest method is to add one to all n-gram counts before normalizing.

- **Formula**: $P^*_{\text{Laplace}}(w_i|w_{i-1}) = \frac{\text{Count}(w_{i-1}w_i)+1}{\text{Count}(w_{i-1})+V}$, where V is the vocabulary size.
- **Problem**: This method moves a very significant share of the probability mass to all the zero-count events, dramatically and often poorly altering the probability distribution. It gives too much probability to things that were never seen.

3.10.2 Add-k Smoothing

A simple extension is to add a small value k (where $k < 1$) instead of 1. The value of k (e.g., 0.5, 0.05) is a hyperparameter that needs to be optimized on a devset. This is useful for some tasks like text classification but does not work very well for language modeling.

3.10.3 Backoff

The intuition behind backoff is that if we have no evidence for a particular n-gram (e.g., a trigram), we should "back off" and use a less-contextual, lower-order n-gram (e.g., a bigram or even a unigram). Using less context can help with generalization.

$$P_{\text{Backoff}}(w_i|w_{i-n+1}^{i-1}) = \begin{cases} P^*(w_i|w_{i-n+1}^{i-1}) & \text{if Count}(w_{i-n+1}^{i}) > 0 \\ \alpha(w_{i-n+1}^{i-1})P_{\text{Backoff}}(w_i|w_{i-n+2}^{i-1}) & \text{otherwise} \end{cases}$$

To make this a valid probability distribution, we need to **discount** the higher-order n-grams to save some probability mass for the lower-order ones.

3.10.4 Interpolation

Instead of choosing just one n-gram level, interpolation mixes the probability estimates from all n-gram estimators (e.g., trigram, bigram, and unigram), weighted by coefficients (λ) that sum to 1.

$$\hat{P}(w_i|w_{i-2}w_{i-1}) = \lambda_1 P(w_i|w_{i-2}w_{i-1}) + \lambda_2 P(w_i|w_{i-1}) + \lambda_3 P(w_i)$$

3.10.5 Kneser-Ney Smoothing

This is the most commonly used and best-performing of the classic smoothing methods. It uses a more sophisticated form of discounting called **absolute discounting**. The core idea is that the amount of probability mass to discount from higher-frequency counts can be estimated by looking at the counts of those same n-grams on a held-out set.

Stupid Backoff

With the advent of web-scale corpora (like the Google N-grams dataset, with over 1 trillion tokens), simpler methods can be effective. **Stupid backoff** is a backoff model that does not include the complex discounting needed to make it a true probability distribution. It simply backs off to the lower-order n-gram if the higher-order one has a zero count. While it's no longer a probability distribution (it's a score), it works surprisingly well on very large datasets.

3.11 Recap

This chapter introduces one of the most fundamental concepts in all of NLP: the **Language Model**. A language model's core function is to answer the question: "Given a sequence of words, what is the probability of the next word?" or, more broadly, "What is the probability of this entire sentence?". This ability is crucial for countless applications, from predicting the next word in your phone's keyboard to guiding a speech recognition system.

The theoretical foundation for calculating sentence probability is the **Chain Rule**. However, applying it directly is impossible due to **data sparsity** we'll never see most long word sequences in any finite amount of training data. The classic solution to this problem is the **Markov Assumption**: we simplify the problem by assuming the probability of a word only depends on a small, fixed number of preceding words. This gives rise to **N-gram models** (unigrams, bigrams, trigrams, etc.), which estimate these probabilities by simply counting occurrences in a large text corpus (Maximum Likelihood Estimation).

Once a model is built, we need to know if it's any good. The chapter presents two evaluation strategies. **Extrinsic evaluation** is the gold standard: you plug the model into a real-world application and see if it improves performance. This is accurate but slow. For faster iteration, we use **intrinsic evaluation**, with **perplexity** being the standard metric. Perplexity can be intuitively understood as the model's "branching factor" or confusion when predicting the next word. A lower perplexity means the model is less "surprised" by the test data and is therefore better at predicting it.

However, a major pitfall of simple N-gram models is **overfitting**. They learn their training data too well, leading to the critical **zero-probability problem**: any n-gram not seen during training is assigned a probability of zero. This is not only incorrect but also makes perplexity infinite if such an n-gram appears in the test set. To solve this, we must use **smoothing** (or discounting). The core idea of smoothing is to take a small amount of probability mass from the n-grams we've seen and redistribute it to those we haven't. The chapter outlines a progression of techniques, from the simple but flawed **Laplace (add-one) smoothing** to more sophisticated and effective methods like **Backoff**, **Interpolation**, and the historically dominant **Kneser-Ney smoothing**. These methods are essential for building robust N-gram models that can generalize beyond their training data.

4 Text classification with linear models

Text classification, also known as text categorization, is the fundamental task of assigning a predefined label or category to an entire text or document.

4.1 Sentiment Classification

A prominent example of text classification is **sentiment analysis**, which focuses on the detection of attitudes. An attitude can be defined as an enduring, affectively colored belief or disposition towards an object or person. A complete sentiment analysis considers several components:

- **Holder (source)**: Who is expressing the attitude.
- **Target (aspect)**: What the attitude is about.
- **Type of attitude**: This can be a specific emotion like *love*, *hate*, *desire*, etc. More commonly, it is simplified to a weighted **polarity** (*positive*, *negative*, *neutral*) along with its strength.
- **Text**: The sentence or document containing the attitude.

Applications of Sentiment Classification

Sentiment analysis has many important applications across various domains:

- **Commercial**: To understand what people think about a new product or brand.
- **Policy-making**: To gauge public sentiment towards a public health measure or a new law.
- **Prediction**: To forecast election outcomes or market trends based on public opinion.

In its simplest form, sentiment classification is a **binary** task (positive vs. negative). Simple **lexical features**, the words themselves, can offer very informative cues. For example, words like "amazing," "great," and "awesome" strongly suggest positive sentiment, while "bad," "scream," and "refuse" suggest negative sentiment.

However, sentiment classification is a hard problem. Simple word-spotting is often insufficient due to linguistic phenomena like sarcasm, irony, and negation. Consider these examples:

- *"If you are reading this because it is your darling fragrance, please wear it at home exclusively, and tape the windows shut."* (Sarcastic negative review)
- *"This film should be brilliant... However, it can't hold up."* (Starts positive but the final sentiment is negative)

Other Text Classification Tasks

Besides sentiment analysis, text classification encompasses many other tasks:

- **Spam detection**: Classifying emails as spam or not spam.
- **Language identification**: Determining the language of a text, often a crucial first step in any NLP pipeline.
- **Authorship attribution**: Identifying the author of a text, which has applications in digital humanities and forensic linguistics.
- **Subject category classification**: Assigning a topic to a document (e.g., politics, sports, technology). This was the original motivation for the invention of the Naive Bayes algorithm in 1961.
- **Toxic language detection**: Identifying hate speech, misogyny, or other forms of toxic content.

In fact, many NLP tasks can be framed as classification problems, including period disambiguation, tokenization, language modeling, and part-of-speech tagging.

4.2 The Classification Task

Formally, the text classification task is defined as:

- Given an input x (often denoted d for a document).
- And a fixed set of output classes $Y = \{y_1, ..., y_M\}$ (often denoted c for a class).
- The goal is to return a predicted class $y \in Y$.

4.2.1 Approaches to Classification

There are three main approaches to solving text classification tasks:

- **Rule-based**: Involves handwritten rules, such as blacklists/whitelists or regular expressions (e.g., contains "$" and "LOTTERY"). These systems can be highly accurate if carefully refined by an expert, but they are often expensive to build and maintain.
- **In-context learning**: Uses a decoder Large Language Model (LLM) and a *prompt* that describes the task and classes. The prompt may include a few examples (*few-shot*) or none (*zero-shot*).
- **Supervised machine learning**: Uses a training set of N labeled documents (d_i, c_i) to learn a model. A **probabilistic classifier** from this family also provides the probability of an observation belonging to a class, which can be useful downstream.

4.2.2 Supervised Approaches: Generative vs. Discriminative

Supervised classifiers can be further divided into two categories:

- **Generative approaches**: These models build a model of how a class could generate some data. To classify a new data point, the system compares it with the models for each class to find the best fit. **Naive Bayes** is a classic example.
- **Discriminative approaches**: These models learn what features are most useful for distinguishing between classes, without necessarily learning about the underlying data distribution of the classes themselves. They directly learn a decision boundary. **Logistic Regression** is a prime example.

4.3 Naive Bayes Classifiers

Representation: Bag-of-Words
The Multinomial Naive Bayes classifier begins by representing a document as a **bag-of-words**. This assumption means that the position and order of words are ignored; we only make use of the frequency of each word. The document is treated as an unordered collection of its words.

The Naive Bayes Classifier
Naive Bayes is a probabilistic classifier. Given a document d, it returns the class $\hat{c}$ that has the maximum posterior probability $P(c|d)$.

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \; P(c|d)$$

Using Bayes' rule, we can rewrite this as:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \; \frac{P(d|c)P(c)}{P(d)}$$

Since $P(d)$ is the same for all classes, we can ignore it, simplifying the problem to:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \; \underbrace{P(d|c)}_{\text{likelihood}} \underbrace{P(c)}_{\text{prior}}$$

If we represent the document as a set of features $f_1, ..., f_n$ (the words), this becomes:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \; P(f_1, ..., f_n|c)P(c)$$

The "Naive" Assumption

The "naive" part of the name comes from a strong simplifying assumption: that all features (words) are **conditionally independent** of each other, given the class. This allows us to break down the likelihood term into a product of individual probabilities:

$$P(f_1, ..., f_n|c) \approx \prod_i P(f_i|c)$$

The final Naive Bayes classification rule is therefore:

$$c_{NB} = \underset{c \in C}{\text{argmax}}\ P(c) \prod_i P(w_i|c)$$

Estimating Probabilities

The prior and likelihood probabilities are learned from frequencies in the training data using Maximum Likelihood Estimates (MLE):

- **Prior** $P(c)$: The probability of a class is the fraction of documents in the training set that belong to that class.
$$\hat{P}(c) = \frac{N_c}{N_{doc}}$$
- **Likelihood** $P(w_i|c)$: The probability of a word given a class is the fraction of the total word count in that class that is accounted for by that word.
$$\hat{P}(w_i|c) = \frac{\text{Count}(w_i, c)}{\sum_{v \in V} \text{Count}(v, c)}$$

For practical reasons (to avoid floating-point underflow), these calculations are typically done in logspace:

$$c_{NB} = \underset{c \in C}{\text{argmax}}\ \log P(c) + \sum_i \log P(w_i|c)$$

Zero-count and OOV words

A major issue arises if a word in the test document does not appear in any training document of a particular class c. Its likelihood $P(w_i|c)$ will be 0, which zeroes out the entire probability for that class. To prevent this, **Laplace (add-1) smoothing** is commonly used in Naive Bayes categorization.

$$\hat{P}_{\text{Laplace}}(w_i|c) = \frac{\text{Count}(w_i, c) + 1}{(\sum_{v \in V} \text{Count}(v, c)) + |V|}$$

Optimizing for Sentiment Analysis

- **Binary NB**: For sentiment analysis, word occurrence often matters more than its frequency. A **binary Naive Bayes** model only considers the presence or absence of a word in a document, not its count.
- **Negation**: To handle negation, a common technique is to prepend a `NOT_` tag to every word between a negation word (like "not", "didn't") and the next punctuation mark.
- **Sentiment Lexicons**: When training data is insufficient, specialized sentiment lexicons (lists of positive and negative words) can be used as features, for example by adding a feature that counts how many positive or negative words from the lexicon appear in the document.

4.3.1 Summary: Naive Bayes is Not So Naive

- **Strengths**: It is very fast, requires low storage, and is robust to irrelevant features. It is very good in domains with many equally important features.
- **Weaknesses**: It makes an overly strong independence assumption, which can cause it to overestimate evidence from correlated features.
- **Role**: It serves as a good, dependable **baseline** for text classification, but other classifiers often give better accuracy.

4.4 Logistic Regression

Discriminative Classification

Like Naive Bayes, Logistic Regression (LR) is a probabilistic classifier that uses supervised machine learning. However, unlike Naive Bayes, LR is a **discriminative model**. It does not attempt to build a model of how a class generates data ($P(d|c)$); instead, it directly computes the posterior probability $P(c|d)$.

A probabilistic ML classifier generally has four components:

1. A feature representation of the input.
2. A classification function that computes the prediction via $p(y|x)$.
3. An objective function (or loss function) that measures error.
4. An optimization algorithm to minimize that error.

4.4.1 How Logistic Regression Works

LR calculates a weighted sum of evidence for a class based on input features x_i, weights w_i, and a bias term b:

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b = w \cdot x + b$$

This sum z can be any real number. To force the output to be a legal probability (between 0 and 1), it is passed through the **logistic function** (or sigmoid function):

$$P(y = 1) = \sigma(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-(w \cdot x + b)}}$$

The decision boundary is typically at 0.5 probability.

Features for Logistic Regression

Unlike Naive Bayes, which usually relies on bag-of-words, LR allows for more sophisticated **feature engineering**. Features can be designed based on linguistic intuitions. For sentiment analysis, features could include:

- x_1: count of positive lexicon words in the document.
- x_2: count of negative lexicon words in the document.
- x_3: 1 if the word "no" is present, 0 otherwise.
- x_6: log of the total word count of the document.

General Remarks and Comparison to Naive Bayes

- **Feature Interactions**: In both LR and NB, interactions between features must be designed by hand. The recent trend is towards **representation learning**, where features are learned automatically.
- **Correlated Features**: LR is robust to correlated features, whereas NB is not (due to its independence assumption).
- **Performance**: LR generally works better on larger datasets, while NB can work extremely well on small datasets.
- **Training Speed**: NB is much faster to train as it's a simple counting process with no optimization step.

4.4.2 Training Logistic Regression

The goal is to learn the parameters (weights w and bias b) that maximize the probability of the correct labels in the training data. This is achieved by minimizing a **loss function**, which measures the distance between the model's prediction and the true label. The standard loss function for logistic regression is the **cross-entropy loss**.

- **Optimization**: Because this loss function is convex, we can use optimization algorithms like **Stochastic Gradient Descent (SGD)** to find the parameter values that minimize the loss.
- **Regularization**: To avoid overfitting, a regularization term (like L1 or L2) is added to the loss function to penalize large weights.

Multinomial Logistic Regression

For tasks with more than two classes, LR can be extended to **multinomial logistic regression** (also known as softmax regression or maxent classifier). It uses the **softmax function** to normalize the output into a probability distribution over all possible classes.

4.5 Evaluation Methods

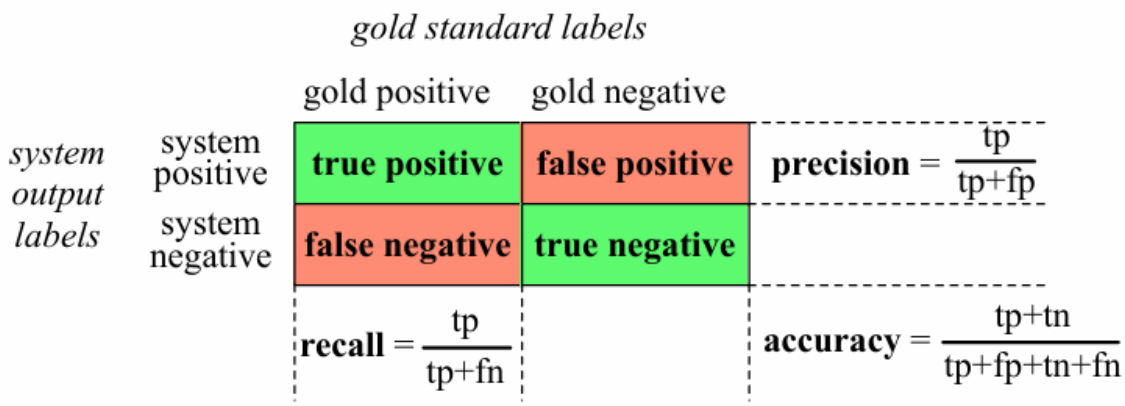
4.5.1 Metrics for Binary Classification

To evaluate a classifier, we compare its predictions against a **gold standard**[3] (human-annotated labels) using a **contingency table**, which breaks down predictions into:

- **True Positives (tp)**: Correctly identified as positive.
- **False Positives (fp)**: Incorrectly identified as positive.
- **True Negatives (tn)**: Correctly identified as negative.
- **False Negatives (fn)**: Incorrectly identified as negative.

From this, we can compute key metrics:

- **Precision**: $\frac{tp}{tp+fp}$ (How accurate are the positive predictions?)
- **Recall**: $\frac{tp}{tp+fn}$ (What fraction of the actual positives were found?)
- **Accuracy**: $\frac{tp+tn}{tp+fp+tn+fn}$. Accuracy is only a good metric when classes are balanced.
- **F1-score**: The harmonic mean of precision and recall, $F_1 = \frac{2PR}{P+R}$. This is the most commonly used metric for binary classification tasks.



[3] Also known as 'ground truth'. It refers to data that has been manually annotated by humans and is assumed to be 100% correct for the purpose of evaluation.

Metrics with Multiple Classes

For multi-class tasks, we use a **confusion matrix** to see where the errors are being made. We can then calculate precision and recall for each class individually. To get a single performance number, we can average these scores:

- **Microaverage**: Collect all decisions into a single contingency table and compute the metric once. This is dominated by the most frequent class.
- **Macroaverage**: Compute the performance for each class, then average these scores. This is more appropriate when all classes are equally important.

4.5.2 Test Sets and Cross-Validation

If a dataset is not large enough to create a representative test set, we use **10-fold cross-validation**. The data is split into 10 parts; the model is trained 10 times on 9 parts and tested on the remaining 1 part. The final error rate is the average of the 10 runs.

Statistical Significance Testing

When comparing two classifiers, A and B, we need to know if A's better performance is real or just due to chance on a particular test set. We use statistical significance testing to reject the **null hypothesis** (that there is no difference between the models). If the **p-value** is small (e.g., ¡ 0.05), we can confidently say that A is truly better than B.

4.6 Lexicons for Sentiment, Affect, and Connotation

4.6.1 Affective Meaning of Text

Beyond simple positive/negative polarity, text can convey a wide range of affective states:

- **Emotion**: A brief, intense response to a specific event (e.g., angry, joyful).
- **Mood**: A longer-lasting, diffuse state (e.g., cheerful, gloomy).
- **Attitude**: An enduring disposition towards something (e.g., liking, hating).

This deeper linguistic analysis, closely related to **subjectivity**, has many applications, from social media analysis to improving conversational agents. The core idea is to use **lexicons** lists of words with associated affective scores,instead of using every word as a feature.

4.6.2 Theories of Emotion

There are two influential theories for modeling emotion:

1. **Basic Emotions**: Proposes that there are a fixed set of atomic emotions (e.g., Plutchik's 8: joy, sadness, anger, fear, etc.).
2. **Dimensional Model**: Describes emotions as a continuum in a 2/3-dimensional space:
   - **Valence**: The pleasantness of the stimulus.
   - **Arousal**: The intensity of the emotion.
   - **Dominance**: The degree of control exerted by the stimulus.

   The **valence** dimension is often used directly as a measure of sentiment.

Sentiment and Emotion Lexicons

Various lexicons have been created to capture these affective meanings:

- **NRC Valence, Arousal, and Dominance (VAD) Lexicon**: Assigns scores on all three dimensions to 20,000 words.
- **NRC Word-Emotion Association Lexicon (EmoLex)**: Labels 14,000 words with Plutchik's 8 basic emotions.

These lexicons are created through methods like expert annotation, crowdsourcing (e.g., Amazon Mechanical Turk), and semi-supervised or supervised learning from data like online reviews.

4.7 Recap

This chapter moves from modeling sequences of words to a core NLP application: **Text Classification**, the task of assigning a category to a document. Using sentiment analysis as a running example, we explore how to build systems that can label text as positive or negative.

The chapter introduces two foundational linear models, highlighting the critical distinction between **generative** and **discriminative** approaches. The first is **Naive Bayes**, a generative classifier. Its elegance lies in its simplicity: it uses Bayes' rule to find the most probable class for a document. It relies on a "bag-of-words" representation and, crucially, a "naive" assumption that all words are conditionally independent. This makes it incredibly fast and a surprisingly effective baseline, especially on smaller datasets.

In contrast, we have **Logistic Regression**, a discriminative classifier. Instead of modeling how the data for each class is generated, it directly learns a decision boundary that best separates the classes. It computes a weighted sum of evidence from various input features and uses the logistic (sigmoid) function to map this sum to a probability. Unlike Naive Bayes, it is robust to correlated features and often performs better on larger datasets, but requires an iterative optimization process (like Gradient Descent) to learn its weights.

A significant portion of the chapter is dedicated to the crucial practice of **Evaluation**. We learn that simple accuracy is often misleading, especially for unbalanced datasets. Instead, the standard metrics are **Precision**, **Recall**, and their harmonic mean, the **F1-score**. For multi-class problems, the **confusion matrix** helps diagnose where a model is making errors, and macro/micro averaging provide ways to summarize performance. The chapter also covers essential methodologies like **cross-validation** for small datasets and **statistical significance testing** to ensure that a model's superior performance is not just a fluke.

Finally, the chapter broadens the scope from simple positive/negative sentiment to a richer understanding of **affective meaning**, including emotions and moods. It introduces the idea of using manually or automatically created **Lexicons** dictionaries of words annotated with sentiment polarity, emotion, or dimensional scores (Valence, Arousal, Dominance). These lexicons serve as powerful feature sets for machine learning models or can be used directly in rule-based systems, providing a valuable tool for understanding the nuances of human expression in text.

5 Vector Semantics and Sparse Embeddings

5.1 Lexical Semantics

The Concept of Representation

We begin by questioning the nature of representation itself. René Magritte's famous painting of a pipe features the caption "Ceci n'est pas une pipe" ("This is not a pipe"). This highlights a fundamental truth in NLP: the label or the word is just a representation, not the object itself. In lexical semantics, we study how these "labels" (words) relate to meaning.

5.1.1 Lemmas and Senses

When we look at a dictionary, such as Merriam-Webster, we encounter specific terminology:

- **Lemma**[4]: This is the citation form of a word (e.g., *pipe*). It is the headword that covers all inflected forms (like *pipes*, or archaic forms like *pype*). Dictionaries typically do not have separate definitions for every inflected form.
- **Word Sense (or Concept)**: This refers to a specific meaning component of a word. Since lemmas can be **polysemous** (having multiple meanings), a crucial task in NLP is **Word Sense Disambiguation** determining which correct sense is intended in a specific context.

Relations Between Senses

We define meaning largely through the relationships between different words:

- **Synonymy**: A relation of identity (or near identity) between two senses. For example, *couch/sofa*, *water/*H_2O. Two words are synonyms if they can be substituted for one another without changing the truth condition of the sentence (propositional meaning). However, the **Principle of Contrast** suggests that perfect synonyms do not exist; a difference in form almost always implies a difference in nuance, politeness, slang, register, or genre.
- **Antonymy**: Senses that are opposites with respect to one feature of meaning, while being very similar otherwise. Examples include *dark/light* (binary opposition), *short/long* (ends of a scale), or *rise/fall* (reversives).
- **Taxonomic Relations**:
  - **Hyponymy (IS-A)**: Subordination. E.g., *car* is a subordinate of *vehicle*.
  - **Hypernymy**: Superordination. E.g., *furniture* is a superordinate of *lamp*.
  - **Meronymy**: Part-whole relation. E.g., *leg* is a part of *table*.

5.1.2 WordNet

These relationships form structured graphs. **WordNet** is a massive lexical database that organizes words into such a graph.

- It groups words into sets of synonyms called **synsets**.
- It defines relationships like hypernyms, hyponyms, and meronyms for nouns, and specific relations like entailment (*snore* implies *sleep*) for verbs.
- It uses **Supersenses** as coarse semantic categories (e.g., defining "dog" under the category ANIMAL, "car" under ARTIFACT).

WordNet allows us to perform **Word Sense Disambiguation** by mapping input words to their specific graph nodes (senses).

[4] A Lemma is the canonical form, dictionary form, or citation form of a set of words (headword). For example, 'run' is the lemma for 'runs', 'running', 'ran'

5.1.3 Word Similarity and Relatedness

While synonyms are rare, words often share **similarity** (e.g., *cat/dog* are not synonyms but are similar as they are both animals). Shifting focus from precise sense relations to word similarity simplifies many semantic tasks.

- **Relatedness (Association)**: This is broader than similarity. *Coffee* and *cup* are related but not similar. *Scalpel* and *surgeon* are related. This concept connects to **Semantic Fields** (words covering a specific domain like "hospital" or "restaurant") and **Topic Models** (like Latent Dirichlet Allocation).
- **Semantic Frames**: These denote a perspective or event type. For example, a "commercial transaction" frame involves participants like *buyer*, *seller*, *goods*, and *money*.

5.1.4 Connotation and Vector Origins

Words also carry **affective meaning** or sentiment. Osgood et al. (1957) proposed a revolutionary idea: meaning could be plotted in a multidimensional space using three axes:

1. **Valence** (positive/negative)
2. **Arousal** (active/passive)
3. **Dominance** (powerful/weak)

By assigning numbers to words on these scales (e.g., *heartbreak*: [2.45, 5.65, 3.58]), they created the first expression of **vector semantics models**: representing a word as a point in space.

5.2 Distributional Semantics

The Problem of Definition

Defining meaning rigidly is difficult. Wittgenstein (1945) argued that categories like "game" have no single common feature, but rather a network of "family resemblances." He famously stated: *"The meaning of a word is its use in the language."*

Vector Semantics and the Distributional Hypothesis

This leads to the approach of the **Linguistic Distributionalists** (Harris, Firth). They define a word by its environment—the company it keeps.

- **Distribution**: The set of contexts (neighboring words) in which a word occurs.
- **Hypothesis**: If words A and B share similar distributions (contexts), they likely share similar meanings.

Example: What is Ongchoi?

Imagine you don't know the word "Ongchoi." You read sentences like:

- "Ongchoi is delicious sautéed with garlic."
- "Ongchoi is superb over rice."
- "...spinach sautéed with garlic over rice."

Even without a definition, the context tells you that "Ongchoi" is likely a leafy green similar to spinach or chard. This demonstrates how context defines meaning.

5.3 Embeddings

We combine the distributional intuition (counting neighbors) with the vector intuition (points in space). Each word becomes a vector, or an **embedding**.

- It is called an embedding because the word is embedded into a mathematical space.
- Similar words fall nearby in this space.
- This allows us to compute **similarity** mathematically.
- Embeddings can be learned automatically from text without complex supervision.

5.3.1 Two Common Models

1. **TF-IDF**: A baseline model using **sparse** vectors. Words are functions of counts of nearby words.
2. **Word2vec**: Uses **dense** vectors. Representations are created by training a classifier to distinguish nearby words from far-away words.

5.3.2 Sparse vs. Dense

Example 1: TF-IDF (Sparse)

In this example using `scikit-learn`, the output is a matrix where each column represents a word. Notice how many zeros are present (sparsity) because a document typically contains only a small subset of the entire vocabulary.

```
from sklearn.feature_extraction.text import TfidfVectorizer

corpus = [
    "the cat sat on the mat",
    "the dog sat on the log",
    "cats and dogs are enemies"
]

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)

# View the feature names (the vocabulary)
print(vectorizer.get_feature_names_out())
# ['and' 'are' 'cat' 'cats' 'dog' 'dogs' 'enemies' 'log' 'mat' 'on' 'sat' 'the
    ']

# View the sparse matrix for the first document
# Only words present in the doc have non-zero values
print(X.toarray()[0])
# [0.    0.    0.42  0.    0.    0.    0.    0.    0.42  0.32  0.32  0.64]
```

Example 2: Word2vec (Dense)

In this example using `Gensim`, the word "king" is represented by a dense vector. Unlike TF-IDF, these numbers are not counts; they are learned weights that position the word in a semantic space. Even if the vocabulary has 10,000 words, this vector will still have a fixed, small size (e.g., 100 dimensions).

```
from gensim.models import Word2Vec

# A toy corpus (tokenized)
sentences = [
    ["cat", "say", "meow"],
    ["dog", "say", "woof"],
    ["cat", "eats", "mouse"],
    ["dog", "eats", "meat"]
]

# Train a Word2Vec model
# vector_size=5 means we compress meaning into just 5 dimensions
model = Word2Vec(sentences, vector_size=5, min_count=1)

# Get the dense vector for "cat"
vector = model.wv['cat']
print(vector)
# Output: [-0.024, 0.045, -0.012, 0.088, -0.091]
# (Note: Real vectors are typically 100-300 dimensions)

# Find similarity
print(model.wv.similarity('cat', 'dog'))
# Output: 0.98 (High similarity because they share context "say", "eats")
```

5.4 Word Vectors

5.4.1 Co-occurrence Matrix

The foundation of vector semantics is the **co-occurrence matrix**, which records how often words occur together. There are many design choices involved: matrix design, reweighting, dimensionality reduction, and vector comparison methods.

Term-Document Matrix
Originally used in Information Retrieval.

- Dimensions: $|V| \times |D|$ (Vocabulary size $\times$ Number of documents).
- Each column is a document vector; each row is a word vector.
- **Example**: In a matrix of Shakespeare plays, the word "battle" might appear 1 time in *As You Like It* and 13 times in *Henry V*. The word "fool" might appear 36 times in *As You Like It* (a comedy) and almost never in history plays.
- Two documents are similar if their vectors are similar. Two words are similar if they appear in similar documents.

Word-Word Matrix (Term-Term)
This is more common for general semantic representation.

- Dimensions: $|V| \times |V|$.
- Rows are target words, columns are context words.
- Context is usually a window (e.g., ± 4 words).
- **Example**: "Strawberry" and "Cherry" will both frequently co-occur with context words like "pie" and "sugar." "Digital" will co-occur with "computer." Thus, the vectors for strawberry and cherry will be geometrically closer to each other than to "digital."

Window Size and Scaling

- **Window size**: Smaller windows (e.g., 1-3 words) capture syntactic information (collocations). Larger windows capture broader semantic information (topic).
- **Scaling**: We can use flat scaling (all neighbors count equal) or scaled (neighbors closer to the target word count more).

5.4.2 Vector Comparison: Cosine Similarity

To compare vectors, the **dot product** is a starting point, but it is sensitive to vector length (frequency). High frequency words would have high dot products just because they are frequent. To solve this, we normalize by the length of the vectors. This results in the **Cosine Similarity**:

$$\text{cosine}(\mathbf{w}, \mathbf{v}) = \frac{\mathbf{w} \cdot \mathbf{v}}{|\mathbf{w}||\mathbf{v}|}$$

This measures the cosine of the angle between the vectors.

- 1: Perfect similarity (vectors point in the same direction).
- 0: No similarity (vectors are orthogonal).
- -1: Opposite (though counts are usually non-negative).

5.4.3 A Frequency Paradox

Raw frequency counts are problematic. Extremely frequent words (like stopwords "the", "it") co-occur with everything but are not informative. We need **reweighting** to handle this skew.

5.5 Reweighting

Goals of Reweighting

We want to amplify trustworthy, unusual counts and deemphasize mundane, frequent ones. We prefer mathematical methods over creating manual "stopword lists."

5.5.1 TF-IDF (Term Frequency - Inverse Document Frequency)

This is the standard for Term-Document matrices. It combines two terms:

1. **TF (Term Frequency)**: How often does term t occur in document d? Usually log-transformed to dampen the effect of very high counts:

$$tf_{t,d} = \begin{cases} 1 + \log_{10} \text{Count}(t, d) & \text{if Count} > 0 \\ 0 & \text{otherwise} \end{cases}$$

2. **IDF (Inverse Document Frequency)**: How rare is the term across all documents?

$$idf_t = \log_{10} \left(\frac{N}{df_t} \right)$$

where N is the total number of documents and df_t is the number of documents containing term t. Words like "good" or "the" have a very low IDF (near 0).

The final weight is the product: $w_{t,d} = tf_{t,d} \times idf_t$. This creates discriminative vectors. For example, in Shakespeare, "Romeo" has a high IDF (very informative), while "sweet" has a textual IDF of zero (appears in all plays).

Using TF-IDF vectors, we can compare documents by computing the centroid (mean) of the vectors of all words in the document.

5.6 PPMI (Positive Pointwise Mutual Information)

For Word-Word matrices, TF-IDF is less suitable. Instead, we use **Pointwise Mutual Information (PMI)**. PMI asks: "How much more do words w and c co-occur than we would expect by pure chance?"

$$PMI(w, c) = \log_2 \frac{P(w, c)}{P(w)P(c)}$$

- $P(w, c)$ is the actual probability of co-occurrence.
- $P(w)P(c)$ is the probability of co-occurrence if the words were independent.

Since negative PMI (co-occurring less than chance) is unreliable and problematic for distance metrics, we typically use **Positive PMI (PPMI)**, replacing all negative values with 0:

$$PPMI(w, c) = \max(PMI(w, c), 0)$$

5.6.1 Issues with PPMI

PMI is biased towards infrequent events. Very rare words often have artificially high PMI scores. To fix this, we can:

- Use **Laplace smoothing** (adding k to counts).
- **Context Distribution Smoothing**: Raise the context probability $P(c)$ to the power of α (typically $\alpha = 0.75$). This increases the probability of rare contexts, thereby lowering their potentially inflated PMI score.

5.6.2 Summary of Reweighting

- Raw counts are often noise.
- **TF-IDF**: Good for document matrices. Punishes words appearing in many documents.
- **PPMI**: Good for word-word matrices. Amplifies counts that are high relative to independence.
- Both create **sparse** representations (many zeros).

5.7 Recap

This chapter introduces a paradigm shift in how we represent meaning in NLP, moving from discrete symbols (like definitions in a dictionary) to continuous mathematics (vectors in space).

It starts with **Lexical Semantics**, the traditional study of word meaning. We looked at structured resources like **WordNet**, which organize words into a graph of relations (synonyms, hypernyms, etc.). While useful, these resources are rigid and don't capture the nuance that "meaning is use."

This leads to **Distributional Semantics**, based on the idea that you can know a word by the company it keeps (context). If we count the neighbors of a word, we can create a **co-occurrence matrix**.

- If we look at which words appear in which *documents*, we get a **Term-Document matrix** (useful for Information Retrieval).
- If we look at which words appear near *other words*, we get a **Term-Term (Word-Word) matrix**.

These matrices turn words into **Sparse Vectors**. A word becomes a long list of numbers. The beauty of this is that we can now use geometry to measure meaning. The **Cosine Similarity** (the angle between two vectors) becomes our proxy for semantic similarity. If the vectors for "strawberry" and "cherry" point in the same direction, they mean similar things.

However, raw counts of words are misleading because frequent words (like "the") dominate. We need **Reweighting** to find the true signal in the noise:

1. **TF-IDF**: Used for documents. It says, "A word is important to a document if it appears a lot here (High TF), but is rare elsewhere (High IDF)."
2. **PPMI**: Used for word associations. It says, "Two words are strongly associated if they appear together much more often than random chance would predict."

By the end of this chapter, we have a way to turn words into meaningful, albeit very large and sparse, numerical vectors. This sets the stage for the next evolution in NLP: **Dense Embeddings** (like Word2vec), which we will likely discuss next, solving the problem of these vectors being too large and mostly empty.

6 Neural language modeling and dense embeddings

6.1 From Sparse to Dense Representations

We previously explored vector representations like TF-IDF and PPMI. These are characterized by being **long** (the length is equal to the vocabulary size $|V|$, typically 20,000 to 50,000) and **sparse** (most elements are zero). We now introduce an alternative: **dense vectors**. These are:

- **Short**: The length is typically between 50 and 1,000 dimensions.
- **Dense**: Most elements are non-zero real numbers.

Why go Dense?

There are several reasons to prefer dense vectors over sparse ones:

- **Machine Learning features**: Short vectors are easier to use as features in ML models because there are fewer weights to tune.
- **Generalization**: Dense vectors generalize better. The idea is to compress the "most" important information into fewer dimensions.
- **Synonymy**: They capture synonymy much better. In a sparse model, "car" and "automobile" are distinct dimensions with no inherent connection. If a classifier learns that "car" is a valid subject for "drive", it won't know that "automobile" is too. In a dense space, these words will be neighbors, allowing the model to generalize what it learns about one to the other.

Popular dense embedding models include **Word2vec**, **GloVe**, and **fastText**. These build upon the ideas of neural language modeling.

6.2 Neural Language Modeling

The Problem with Statistical Models

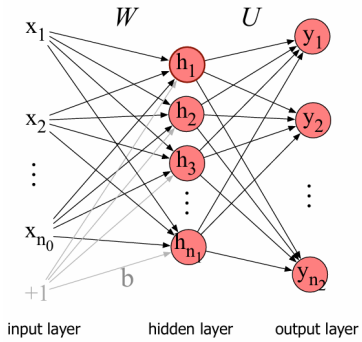
Traditional statistical language models (like N-grams) suffer from the **curse of dimensionality**[5]. Word sequences in a test set are likely to be different from those seen during training. If a specific sequence was not seen, it gets a zero probability (without smoothing). For example, if the training corpus contains "The cat is walking in the bedroom", a good model should generalize to understand that "A dog was running in a room" is almost as likely, even if those exact n-grams were never observed. Neural models achieve this by learning distributed representations.

6.2.1 Feedforward Neural Network

The core idea is to use a neural network to **predict** the next word rather than simply counting occurrences. A simple Feedforward Network consists of:

- An **input layer** x (representing the context words).
- A **hidden layer** h (forming a representation of the input).
- An **output layer** y (probabilities over the vocabulary).

The parameters are the weight matrices **W**, **U** and the bias vector **b**. The network uses activation functions like tanh for the hidden layer and softmax for the output to produce a valid probability distribution.



[5] The **Curse of dimensionality** refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces. In language modeling, it refers to the fact that with a vocabulary size $|V|$, the number of possible sequences grows exponentially, making it impossible to observe all possible n-grams during training, thus leading to data sparsity.

The Setup

To predict the next word given a sequence w_1^n:

1. We associate each word $w \in V$ with a feature vector (embedding) $e(w) \in \mathbb{R}^d$. These are stored in an **embedding matrix E**.
2. We represent input words using **one-hot encoding**[6]. Multiplying the matrix **E** by a one-hot vector effectively selects the column corresponding to that word.
3. We concatenate the vectors of the input context words to form the input layer x.
4. We pass this through the network to get a softmax distribution over V.

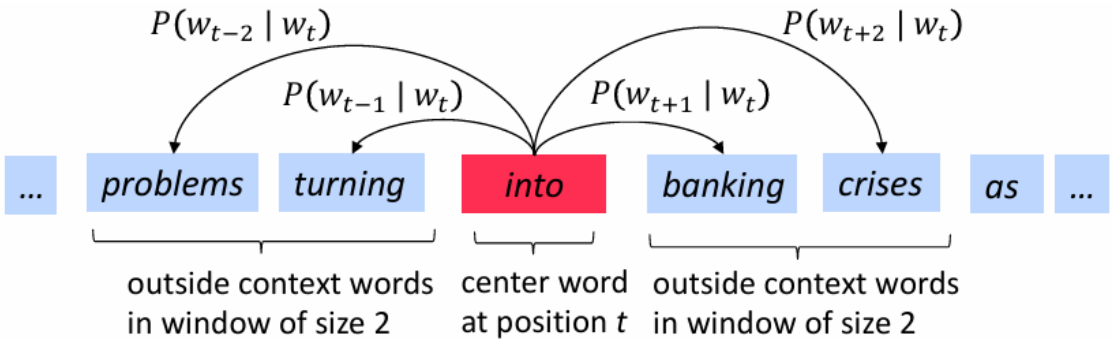
Crucially, all parameters, including the embedding matrix **E**, are learned simultaneously. This works because similar words will learn similar feature vectors (embeddings). Since the probability function is smooth, a small change in the feature vector (swapping "cat" for "dog") results in a small change in probability, enabling generalization.

Training

Training involves initializing weights randomly and processing a large text corpus. The model predicts one word at a time, calculating the **Cross-Entropy Loss** (negative log likelihood) against the actual next word. Errors are backpropagated to update the weights ($\mathbf{E}, \mathbf{W}, \mathbf{U}, \mathbf{b}$). This is a form of **self-supervision**[7]: the data itself provides the labels (the next word), allowing us to learn from vast amounts of unlabeled text.

6.3 Word2vec

Word2vec (Mikolov et al., 2013) is a popular embedding method that is simpler and faster to train than the full neural language model described above. Instead of a full language modeling task (predicting the next word exactly), it learns a simpler **binary classification** task: *Is word w likely to show up near target word c?*



Intuition and Objective

The intuition is to treat the target word and its neighboring context words as **positive examples**, and randomly sampled words from the lexicon as **negative examples**. A logistic regression classifier is trained to distinguish these cases. We don't care about the classification task itself; we care about the weights learned, which become our embeddings.

More specifically:

- We iterate through the corpus.
- For a center word c and an outside context word o, we use the similarity of their vectors (dot product) to calculate the probability of o given c.

[6] **One-hot encoding** is a representation where a categorical variable (like a word) is represented as a binary vector. All elements are 0 except for the index corresponding to the word, which is 1. For a vocabulary of size $|V|$, the vector length is $|V|$.

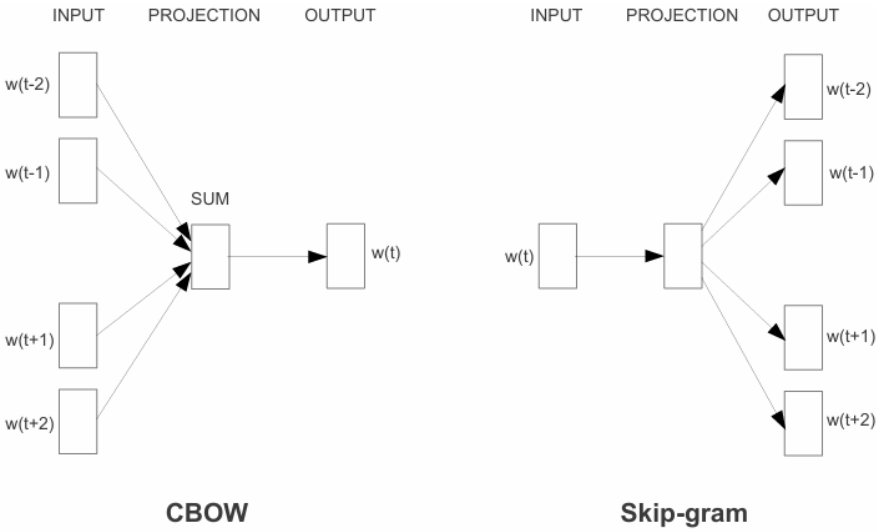
[7] **Self-supervision** is a learning technique where the data provides the supervision. Instead of explicit human labels, the model predicts part of the input from other parts (e.g., predicting the next word given the previous ones). It turns unsupervised data into a supervised task.

- We define $P(+|w, c) = \sigma(c \cdot w)$, where σ is the sigmoid function. This turns the dot product into a probability.
- We use two vectors per word: one when it is a center word (v_w) and one when it is an outside word (u_w). This simplifies optimization.

6.3.1 Model Variants

There are two main architectures in Word2vec:

1. **CBOW (Continuous Bag Of Words)**: Predicts the center word from the bag of outside context words.
2. **Skip-gram**: Predicts the outside context words (position independent) given the center word.



Negative Sampling

Computing the full softmax over the entire vocabulary is too expensive because of the normalization term. Word2vec uses **Negative Sampling** (SGNS - Skip-Gram with Negative Sampling). For each positive pair (center word, context word), we create K noise pairs (center word, random word). The objective is to maximize the probability of the true pair and minimize the probability of the noise pairs using binary logistic regression.

$$J(\theta) = -\log \sigma(u_o^\top v_c) - \sum_{k=1}^{K} \log \sigma(-u_k^\top v_c)$$

Code Example: Word2Vec and Bias

This is how we can train a simple Word2Vec model and explore semantic relationships, including analogies and potential biases.

```
from gensim.models import Word2Vec

# Toy corpus
sentences = [["king", "is", "a", "man"], ["queen", "is", "a", "woman"]]

# Train Skip-gram model (sg=1)
model = Word2Vec(sentences, vector_size=10, window=2, min_count=1, sg=1)

# Check analogy: "king" - "man" + "woman"
result = model.wv.most_similar(positive=['woman', 'king'], negative=['man'])
print(result)
```

```
# Ideally prints [('queen', 0.9...)]

# Checking bias (conceptually)
# If vectors reflect culture, we might find:
# vector('doctor') - vector('man') + vector('woman') approx vector('nurse')
# This reveals gender stereotypes embedded in the corpus.
```

The Algorithm Visualized

The Skip-gram algorithm starts with random vectors. In each step of gradient descent, it tries to shift embeddings so that the target vector (e.g., "apricot") moves *closer* to the context vector (e.g., "jam") and *further* from the noise vector (e.g., "matrix").

6.4 Evaluating Embeddings

6.4.1 Intrinsic Evaluation

This involves comparing embedding scores to human judgments.

- **Word Similarity**: Using datasets like SimLex-999 or WordSim-353, we calculate the cosine similarity between vector pairs and correlate it with human similarity ratings.
- **Analogy Tasks**: Testing the model's ability to solve puzzles like "a is to b as a* is to b*".

6.4.2 Extrinsic Evaluation

Using the embeddings as input features for a downstream task (like Named Entity Recognition or Sentiment Analysis) and measuring the performance improvement.

Properties of Embeddings

6.4.3 Window Size

The size of the context window C affects what the embeddings capture:

- **Small window ($C = \pm 2$)**: Captures syntactic and functional information. Nearest words to "Hogwarts" might be "Sunnydale" or "Evernight" (other locations).
- **Large window ($C = \pm 5$)**: Captures broader topical and semantic information. Nearest words to "Hogwarts" might be "Dumbledore" or "Malfoy".

Types of Association

The relationship between words captured by embeddings can be categorized into two types:

- **First-order co-occurrence (Syntagmatic association)**: This refers to words that typically appear *near* each other in a sentence (e.g., "wrote" and "poem").
- **Second-order co-occurrence (Paradigmatic association)**: This refers to words that have similar neighbors but might not appear together. They are essentially substitutable (e.g., "wrote" and "said" both appear near "book" or "words").

Embeddings are generally good at capturing second-order co-occurrence (similarity).

Analogy and Relational Meaning

Embeddings capture relational meaning via vector arithmetic. The most famous example is:

$$\text{vector(king)} - \text{vector(man)} + \text{vector(woman)} \approx \text{vector(queen)}$$

This works for various relations (pluralization, capital cities, etc.). **Caveat**: The closest vector to the result is often just the input word itself (e.g., "king"). We usually have to exclude the input words to find the answer. Also, simple arithmetic ("parallelogram method") is too simple to model human cognition fully.

6.5 Other Embedding Models

6.5.1 fastText

An extension of Word2vec that addresses the problem of **Out-Of-Vocabulary (OOV)** words and morphology.

- Instead of one vector per word, it represents a word as a bag of character **n-grams** (subwords).
- For example, "where" might be represented by the sum of embeddings for '¡wh', 'whe', 'her', 'ere', 're¿', and the word '¡where¿' itself.
- This allows the model to build vectors for unseen words (like "apple-like") by summing the vectors of their parts.

6.5.2 GloVe (Global Vectors)

While Word2vec relies on local prediction windows, GloVe builds on **global co-occurrence counts**. It attempts to combine the benefits of count-based matrix factorization (like LSA) and window-based learning. It uses ratios of co-occurrence probabilities to encode meaning components.

6.6 Analysing Embeddings: Bias and History

Embeddings as a Window Onto History
We can train embeddings on corpora from different time periods (diachronic embeddings) to study language change.

- **Semantic Shift**: We can visualize how words move in the vector space over time. For example, "gay" shifted from "cheerful" to "homosexual"; "broadcast" shifted from agriculture (casting seeds) to media.
- **Sentiment Evolution**: We can track how the sentiment of specific words changes. For instance, "terrific" changed from negative (terror-inducing) to positive over 150 years.

6.6.1 Embeddings Reflect Cultural Bias

Embeddings learn from human text, and thus they learn human biases (gender, ethnicity, etc.).

- **Gender Bias**: Analogies like 'father : doctor :: mother : x' might result in 'x = nurse'. Vector projections show that occupations like "homemaker", "nurse", "receptionist" are closer to "she", while "architect", "captain", "philosopher" are closer to "he".
- **Ethnic Bias**: Studies replicating the Implicit Association Test (IAT) on embeddings found that African-American names (e.g., "Leroy") had higher cosine similarity with unpleasant words compared to European-American names.
- **Historical Stereotypes**: Embeddings trained on historical data reflect the prejudices of that time. For example, adjectives associated with Chinese people in English corpora shifted from "barbaric" and "monstrous" in 1910 to "inhibited" and "passive" in 1990.

This highlights a critical issue: embeddings reflect and replicate **representational harm**.

6.7 Recap

This chapter marks the transition to **Neural NLP**. We moved from sparse, high-dimensional vectors (TF-IDF) to **dense, low-dimensional embeddings**. The motivation is clear: dense vectors are efficient, generalize better, and naturally capture semantic similarity (synonyms are neighbors).

We started with the general **Neural Language Model**. This architecture uses a Feedforward Neural Network to predict the next word based on a window of previous words. Crucially, it learns an **embedding matrix E** as part of the training process. The input words are converted to vectors via lookup (one-hot multiplication), concatenated, and passed through hidden layers to a softmax output.

This is a **self-supervised** task, meaning we can train on infinite amounts of unlabeled text.

However, full language modeling is computationally expensive. This led to **Word2vec**, a simplified framework focused solely on learning high-quality embeddings. It replaces the prediction task with a binary classification task: "Is this context word real or noise?". We explored two architectures: **CBOW** (context predicts center) and **Skip-gram** (center predicts context). To make training feasible, we use **Negative Sampling**, which approximates the expensive softmax by contrasting the true context with a few random noise words.

The resulting embeddings have fascinating properties. They capture **relational meaning** through vector arithmetic (King - Man + Woman = Queen). We also looked at **fastText**, which improves upon Word2vec by using subword information (character n-grams) to handle unknown words and morphology, and **GloVe**, which leverages global co-occurrence statistics.

Finally, we analyzed the sociological implications of these models. Since embeddings are trained on human text, they act as a **mirror of history and culture**. They allow us to track how word meanings and sentiments evolve over centuries. However, they also learn and amplify human **biases**. We saw evidence of gender and ethnic stereotypes embedded deep within the vector space (e.g., associating women with "soft" adjectives or specific ethnicities with negative traits). This awareness is vital for building ethical AI systems.

7 Sequence processing with recurrent networks

Introduction: The Temporal Nature of Language

Language is inherently sequential. Whether it's the flow of a conversation, a news feed, or a Twitter stream, language happens over time. The machine learning approaches we have seen so far, like Logistic Regression (LR) or Multi-Layer Perceptrons (MLP), treat the input as a fixed block. They access all aspects of the input simultaneously (e.g., a bag-of-words vector). They have no inherent concept of time or sequence order.

To use these static models for language, we usually resort to a work-around: the **sliding window**. We take a fixed-size window of tokens as input (e.g., the last 2 words) to predict the next one. However, this approach has significant limitations:

- **Fixed context size**: We can't look back further than the window allows. Enlarging the window increases the model size and complexity.
- **No symmetry**: The model learns separate weights for the word at position $t-1$ and the word at position $t-2$. It doesn't inherently know that "cat" means the same thing regardless of where it appears in the window.

We need a neural architecture that can process inputs of any length and capture the true temporal nature of language.

7.1 Recurrent Neural Networks (RNNs)

The core idea behind Recurrent Neural Networks is to apply the **same weights repeatedly**. Instead of learning different weights for each position in a fixed window, we use one set of weights and reuse it at every time step.

An RNN is a network containing cycles. The value of a unit depends on its own earlier outputs as an input. This creates a form of **memory**.

- **Symmetry**: The same weights are applied at every time step.
- **Flexibility**: The model size doesn't increase for longer inputs; we just run the cycle more times.

7.1.1 Simple Recurrent Neural Networks (Elman Networks)

The standard or "Vanilla" RNN (proposed by Elman in 1990) processes sequences one element at a time. At each time step t:

- The network receives an input vector x_t.
- It computes a **hidden state** h_t, which acts as the memory.
- Crucially, h_t depends on both the current input x_t AND the previous hidden state h_{t-1}.

The equations governing this are:

$$h_t = g(Wh_{t-1} + Ux_t + b)$$
$$y_t = f(Vh_t)$$

Where:

- W, U, and V are weight matrices shared across all time steps.
- g is a non-linear activation function (usually tanh or ReLU).
- f is the output activation function (usually softmax for classification[8]).

Unrolled Network

To understand how an RNN works (and how to train it), it is helpful to visualize it unrolled in time. We can draw copies of the network for each time step $t = 1, t = 2, t = 3...$ connected in a chain. This clearly shows that while the values (x_t, h_t, y_t) change at every step, the weights (W, U, V) remain identical.

[8] **Soft classification** refers to a classification task where the output is a probability distribution over classes (using softmax), rather than a hard decision (0 or 1).

Code Example: A Simple RNN Cell in PyTorch

Vanilla RNN

```python
import torch
import torch.nn as nn

class SimpleRNNCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(SimpleRNNCell, self).__init__()
        # Weight matrices for input (W_xh) and hidden state (W_hh)
        self.W_xh = nn.Linear(input_size, hidden_size)
        self.W_hh = nn.Linear(hidden_size, hidden_size)
        self.activation = nn.Tanh()

    def forward(self, x, h_prev):
        # Equation: h_t = tanh(W_xh * x_t + W_hh * h_{t-1} + bias)
        h_next = self.activation(self.W_xh(x) + self.W_hh(h_prev))
        return h_next

# Example usage
input_size = 10
hidden_size = 20
rnn = SimpleRNNCell(input_size, hidden_size)

x_t = torch.randn(1, input_size)       # Input at time t
h_prev = torch.randn(1, hidden_size)   # Hidden state from t-1

h_t = rnn(x_t, h_prev)                 # New hidden state
```

7.1.2 Neural Language Modeling with RNNs

We can use an RNN to build a language model (RNN-LM).

1. **Input**: A sequence of words represented as one-hot vectors.
2. **Embeddings**: These are multiplied by an embedding matrix E to get dense vectors $e^{(t)}$.
3. **Hidden Layer**: The RNN computes the hidden state $h^{(t)}$ based on the previous state $h^{(t-1)}$ and the current embedding $e^{(t)}$.
4. **Output**: A softmax layer predicts the probability distribution for the next word $y^{(t)}$ over the entire vocabulary $|V|$.

7.1.3 Training

To train an RNN-LM:

- We take a large corpus of text.
- We feed the sequence into the RNN.
- At every step t, we compute the output distribution $\hat{y}^{(t)}$ and compare it to the actual next word $y^{(t)}$ (the ground truth).
- The loss function is the **Cross-Entropy Loss** at each step: $J^{(t)}(\theta) = -\log \hat{y}^{(t)}_{x_{t+1}}$.
- The total loss is the average loss over all time steps.

A crucial technique during training is **Teacher Forcing**[9]. Instead of feeding the model its own prediction from the previous step (which might be wrong), we always feed it the correct actual word from the corpus.

[9] **Teacher Forcing** is a training strategy for recurrent neural networks where the model is fed the *ground truth* from a prior time step as input, rather than its own generated output. This stabilizes training and helps the model converge faster.

7.1.4 Backpropagation Through Time (BPTT)

Since the RNN is unrolled over time, we backpropagate errors not just through layers, but back through time steps. The gradient with respect to a repeated weight (like W) is the sum of the gradients at each time step it was used. This process is called Backpropagation Through Time.

7.2 Applications

RNNs are versatile and can be applied to many NLP tasks beyond language modeling.

Language Generation

Once trained, an RNN-LM can generate text. We feed it an initial token (like '¡s¿'), sample a word from the output distribution, and then feed that sampled word back in as the input for the next step. This is called autoregressive generation.

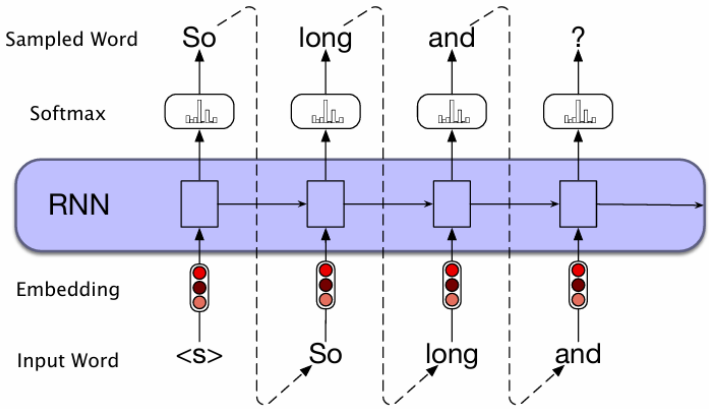


Figure 1: Just like an n-gram Language Model,you can use an RNN Language Model to generate text by repeated sampling.

Sequence Labeling

Tasks like POS-tagging or Named Entity Recognition. Here, we want to assign a label to every element in the sequence. The input is a sequence of words, and the output at each step is a probability distribution over tags (e.g., Noun, Verb).

Sequence Classification

Tasks like Sentiment Analysis or Spam Detection. We process the entire sequence of words, but we only care about the final output. Typically, we use the final hidden state h_n (or an average/max of all hidden states) as a representation of the whole sentence. This "sentence encoding" is then fed into a standard classifier (like a Feedforward Network) to make a final decision (Positive/Negative).

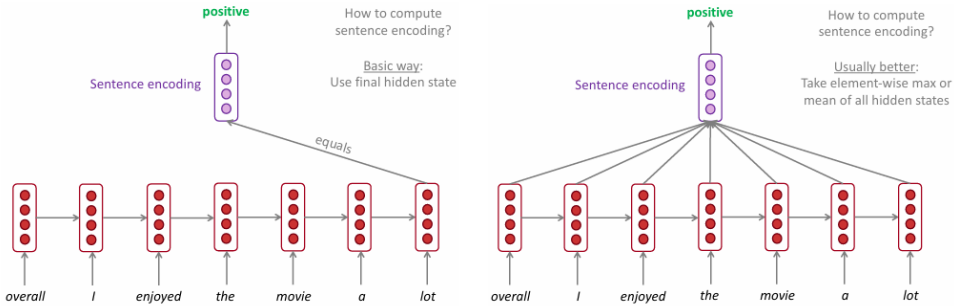


Figure 2: Sentence encoding could be computed as the final hidden state or as element-wise max or mean of all hidden states

Encoder-Decoder Models

Tasks like Machine Translation or Question Answering. One RNN acts as an Encoder: it processes the input sequence (e.g., a question) and compresses it into a hidden state vector. This vector is then used by another component (the Decoder) to generate the output (e.g., the answer).

7.3 Fancier Architectures

Stacked (Multi-Layer) RNNs

Just as deep feedforward networks are more powerful than shallow ones, we can stack RNNs. The sequence of outputs from RNN 1 serves as the input sequence for RNN 2. This allows the model to learn representations at different levels of abstraction. However, training costs rise steeply.

Bidirectional RNNs

In standard RNNs, the output at time t only depends on the past (words 1 to t). But for many tasks (like POS tagging or translation), knowing the *future* context is helpful. A Bidirectional RNN consists of two independent RNNs:

- One processes the sequence forward (left-to-right).
- One processes the sequence backward (right-to-left).

The final representation at each step is the concatenation of the forward and backward hidden states. Note: This cannot be used for standard Language Modeling (predicting the next word) because it would "see" the answer.

Character-Level Embeddings

A powerful extension of RNNs involves looking inside the words. Instead of treating words as atomic units, we can use a Bidirectional RNN to read the sequence of characters that make up a word.

- The final hidden states of this character-level RNN act as a **character-based embedding** of the word.
- This embedding is then concatenated with the standard pre-trained word embedding.
- This approach allows the model to handle **Out-Of-Vocabulary (OOV)** words and capture morphological clues (like prefixes and suffixes) that standard word embeddings might miss.

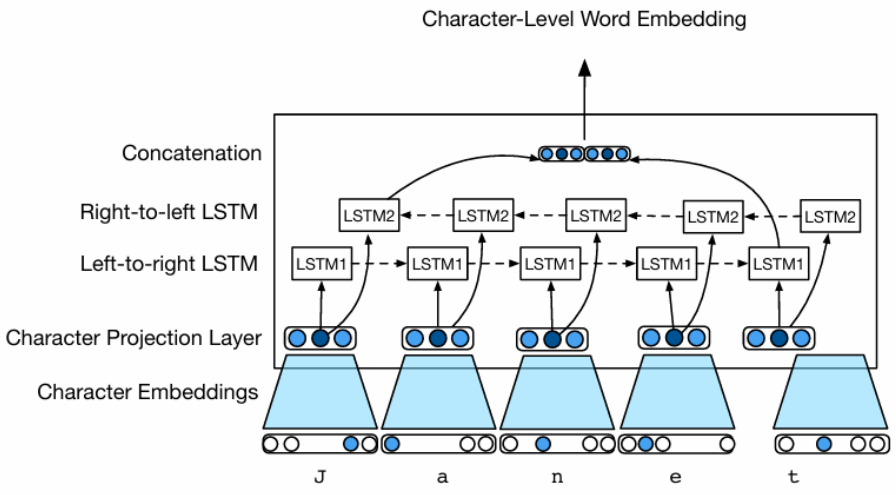


Figure 3: Bi-RNN accepts word character sequences and emits embeddings derived from a forward and backward passover the sequence. The network itself is trained in the context of a largerend-application where the loss is propagated all the way through to the character vector embeddings.

7.4 Vanishing Gradient

Training standard RNNs on long sequences is notoriously difficult due to the **Vanishing Gradient problem**[10]. When we backpropagate errors over many time steps, the gradient signal gets multiplied repeatedly by the weight matrix. If the weights are small, the gradient shrinks exponentially and "vanishes" before reaching the early parts of the sequence.

- **Consequence:** The model weights are updated only based on near-term effects. The model fails to learn long-term dependencies (e.g., connecting a subject at the start of a long sentence to a verb at the end).
- **Syntactic vs. Sequential Recency:** The model might learn to predict based on the most recent word (sequential recency) rather than the grammatically correct distant word (syntactic recency).

7.5 Advanced Gated Architectures

To solve the vanishing gradient problem, more complex cell architectures were introduced. They introduce "gates" to control the flow of information.

7.5.1 Long Short-Term Memory (LSTM)

Proposed by Hochreiter and Schmidhuber (1997). The core idea is to separate the cell state c_t (long-term memory) from the hidden state h_t (short-term output). LSTMs have three gates (vectors of values between 0 and 1) that protect and control the cell state:

1. Forget gate (f_t): What information to throw away from the previous cell state.
2. Input gate (i_t): What new information to store in the cell state.
3. Output gate (o_t): What parts of the cell state to output as the hidden state h_t.

This architecture provides a direct path (the "highway") for gradients to flow backwards through time without vanishing, allowing the model to learn very long-distance dependencies.

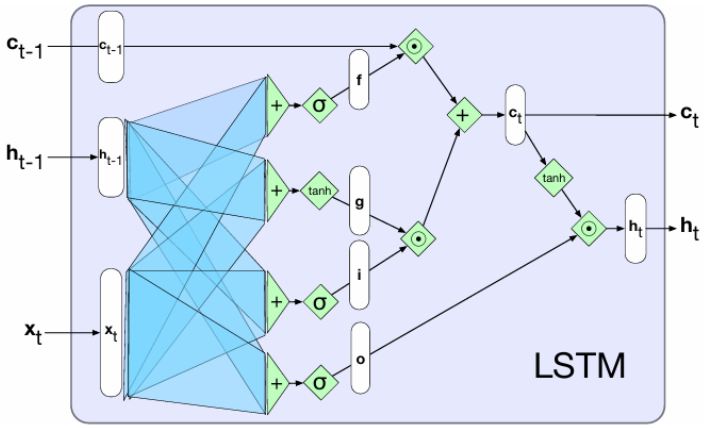


Figure 4: A single LSTM unit displayed as a computation graph. The inputs to each unit consist of the current input x, the previous hidden state h_{t-1}, and the previous context c_{t-1}. The outputs are a new hidden state h_t and an updated context c_t.

7.5.2 Gated Recurrent Units (GRU)

Proposed by Cho et al. (2014) as a simpler alternative to LSTM.

- It merges the cell state and hidden state into a single state h_t.

[10] The **Vanishing Gradient Problem** occurs during the training of deep neural networks (like RNNs) when gradients are backpropagated. Because gradients are multiplied at each layer (chain rule), if they are small (< 1), they shrink exponentially as they go back in time, effectively preventing the network from learning long-term dependencies.

- It has only two gates: an Update gate (combines forget and input functions) and a Reset gate.
- GRUs are faster to compute than LSTMs and often achieve comparable performance.

7.6 Other Solutions

7.6.1 Skip Connections (Residual Connections)

The vanishing gradient problem isn't unique to RNNs; deep feedforward networks suffer from it too. A general solution is the Skip Connection (or residual connection). Instead of just passing x through a layer to get $F(x)$, we add the original input back to the output: $y = F(x) + x$. This creates a direct path for the gradient to flow through the network, acting like a "highway". This is the core idea behind ResNets and DenseNets.

7.7 Recap

This chapter explores how we handle the fundamental nature of human language: **time**. Unlike images which are static, text is a sequence. Previous models tried to force text into fixed boxes (windows), losing the ability to see the full picture or understand long-distance relationships.

The solution is the **Recurrent Neural Network (RNN)**. The genius of the RNN is **parameter sharing**: it uses the exact same "brain" (weights) to process every word in the sequence, one by one. As it reads, it maintains a **hidden state**, a vector that acts as a short-term memory, carrying information from the past to the present.

We can use RNNs for everything:

- **Language Modeling**: Predicting the next word (which lets us generate text like Shakespeare or Obama).
- **Sequence Labeling**: Tagging every word (like POS tagging).
- **Classification**: Reading a whole sentence to decide its sentiment.

However, standard "Vanilla" RNNs have a fatal flaw: the Vanishing Gradient. When trying to learn from long sequences, the error signal gets weaker and weaker as it travels back in time. The network effectively "forgets" what happened more than a few steps ago. It focuses on **sequential recency** (what words are close) rather than true **syntactic dependency**.

This led to the invention of Gated Architectures like LSTMs and GRUs. These are specialized RNNs equipped with "gates", learnable mechanisms that decide what to remember, what to ignore, and what to forget. They act like managers of the network's memory, protecting valuable long-term information from being overwritten. These architectures dominated NLP for years (roughly 2013-2017) until the arrival of Transformers, proving that managing memory and information flow is the key to mastering sequence processing.

8 Seq2seq, CNN, and Transformers

Sequence to Sequence Networks (Seq2seq)

A **Sequence to Sequence** network (often called seq2seq or Encoder-Decoder network) is a model designed to map an input sequence to an output sequence. It consists of two main components, typically Recurrent Neural Networks (RNNs):

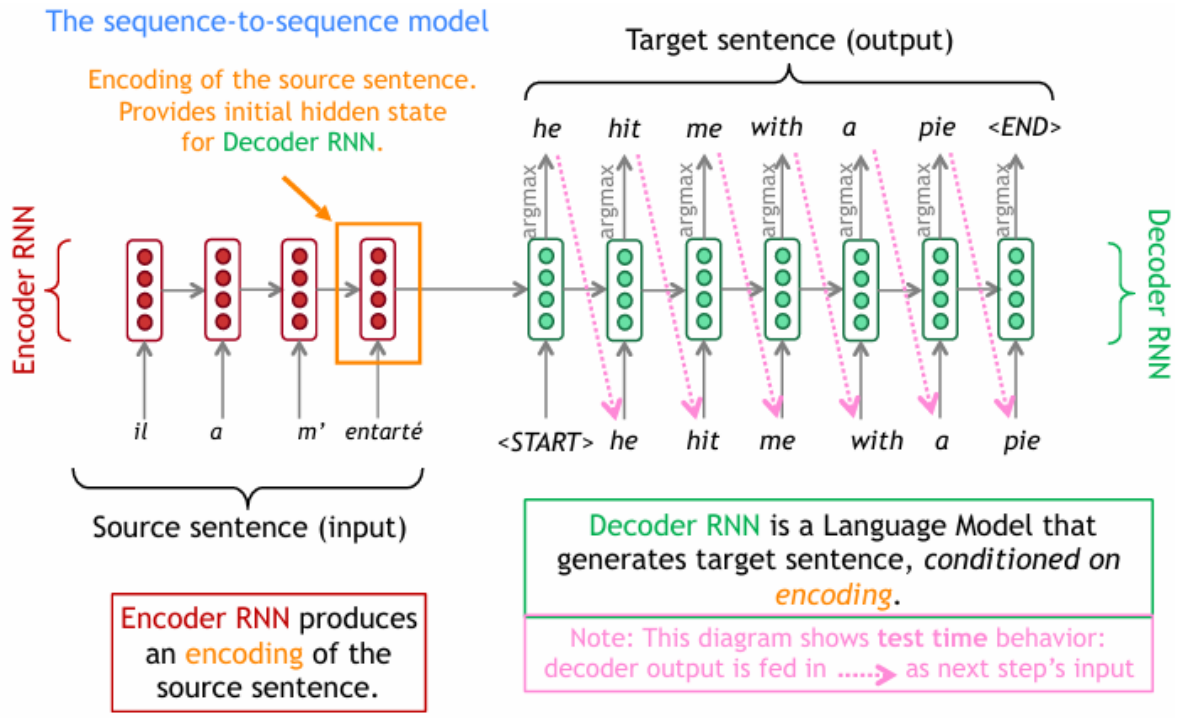
- **Encoder**: Reads the input sequence (source) token by token and compresses it into a single context vector (the final hidden state).
- **Decoder**: Takes that context vector as its initial state and generates the output sequence (target) one token at a time.

This architecture is the foundation of **Neural Machine Translation (NMT)**.

8.0.1 The Seq2seq Model

Consider the task of translating "il a m'entarté" (French) to "he hit me with a pie" (English).

1. The **Encoder RNN** reads the source sentence. At each step, it updates its hidden state. The final hidden state is crucial: it is the "encoding" of the entire source sentence.
2. The **Decoder RNN** is essentially a conditional Language Model. It is initialized with the encoder's final state.
3. At each step, the decoder generates a word based on its current hidden state and the previously generated word.
4. *Test time behavior*: The output word predicted at step t is fed as input for step $t+1$.



Training Seq2seq

Training is done "end-to-end". We feed a large corpus of source-target sentence pairs.

- The system calculates the loss at every step of the decoder (negative log probability of the correct target word).

- The total loss is the sum of losses over all timesteps: $J = \frac{1}{T}\sum_{t=1}^{T} J_t$.
- Backpropagation flows from the decoder, through the "bottleneck" context vector, all the way back to the encoder.

8.1 Attention Mechanism

The standard seq2seq model has a bottleneck: the Encoder must compress the *entire* source sentence into a single fixed-length vector. This is hard for long sentences. The **Attention Mechanism** solves this by allowing the Decoder to "look back" at the source sentence at every step.

- Instead of relying only on the final hidden state, the Decoder can access **all** the Encoder's hidden states $h_1, ..., h_N$.
- At each decoding step t, the model computes **attention scores** to decide which source words are relevant for the current target word.
- These scores create an **attention distribution** α^t (via softmax).
- The model computes a weighted sum of the encoder hidden states (the **context vector**).
- This context vector is concatenated with the decoder's state to make a prediction.

This allows the model to **focus** on specific parts of the input (e.g., focusing on "entarté" when generating "pie").

8.2 Convolutional Networks for NLP

While RNNs are designed for sequences, **Convolutional Neural Networks (CNNs)**[11] can also be effective for text.

1D Convolution

The idea is to apply a **filter** (or kernel) of a certain size (e.g., 3) that slides over the sentence.

- It computes a vector for every possible subsequence of length 3 (trigrams).
- Example: "tentative deal reached to keep government open". A filter of size 3 would process "tentative deal reached", then "deal reached to", and so on.
- This operation is position-invariant: it detects features (like "not good") regardless of where they appear in the sentence.

Typically, we use multiple filters of different sizes (e.g., 2, 3, 4) to capture different n-gram patterns. After convolution, we typically use **Max-over-time pooling**: we take the maximum value from each feature map. This captures the most important feature detected by that filter anywhere in the sentence and produces a fixed-size output vector regardless of sentence length.

Character-Aware Neural Language Models

A powerful application of CNNs in NLP is to build word embeddings from characters. Instead of storing a massive embedding matrix for every word in the vocabulary (which is memory-intensive and fails on OOV words), we can compute the embedding on the fly.

- The input is the sequence of characters of a word.
- A CNN processes this character sequence to extract features (e.g., identifying prefixes, suffixes, stems).
- Max-pooling over time produces a fixed-size vector for the word.
- This vector is then fed into an RNN (LSTM) language model.

This approach significantly reduces model parameters and achieves state-of-the-art results by leveraging subword information.

[11] **Convolutional Neural Networks (CNNs)** were originally designed for computer vision to capture spatial hierarchies in images. In NLP, they are adapted (1D convolution) to capture local patterns in sequences, such as n-grams, regardless of their position in the sentence.

8.2.1 Comparison: RNN vs CNN for Sentence Encoding

- **RNNs**: Cognitively plausible (read left-to-right). Great for sequence tagging and language modeling. Slow to train (sequential).
- **CNNs**: Good for classification. Easy to parallelize on GPUs (efficient). Harder to interpret linguistically but effective at spotting local key phrases.

8.3 Transformers

The **Transformer** is the modern standard architecture for NLP, overcoming the limitations of both RNNs (sequential, slow) and CNNs (local context only). It relies entirely on the **Attention mechanism** to draw global dependencies between input and output.

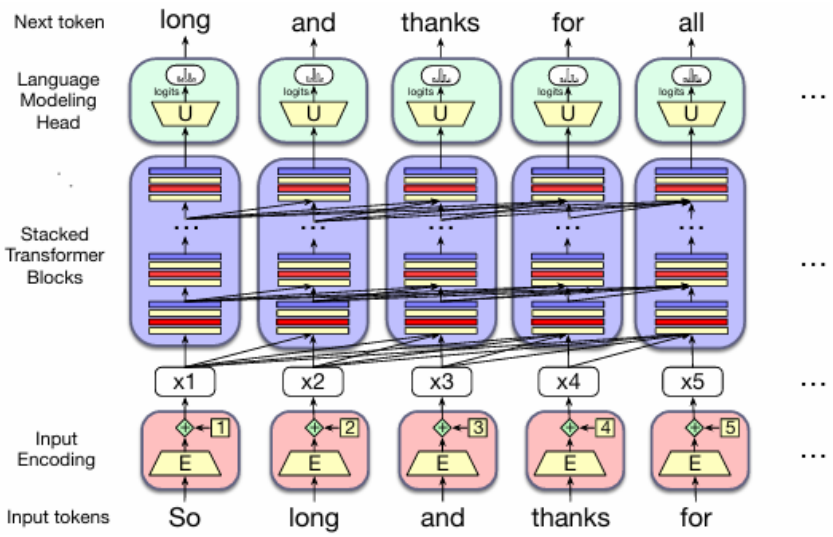
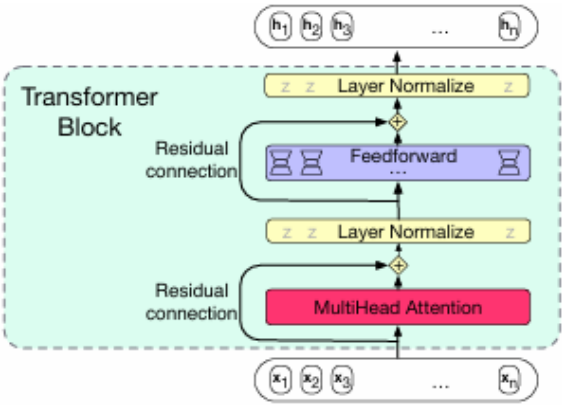


Figure 5: The architecture of a (left-to right) transformer, show inghoweachinputtoken get encoded, passed through a set of stacked transformer blocks, and then a language model head that predicts the next token.

Architecture Overview

A Transformer is composed of a stack of **blocks**. Each block maps a sequence of input vectors $(x_1, \ldots, x_n)$ to a sequence of output vectors $(z_1, \ldots, z_n)$ of the same length and dimensionality d. The key components inside a block are:

1. **Multi-Head Self-Attention**: Allows the network to directly look at other words in the sequence.
2. **Feedforward Layer**: A position-wise fully connected network.
3. **Residual Connections**[12] and **Layer Normalization**.



A transformer block showing all the layers.

[12] **Residual Connections** (or skip connections) allow the input of a layer to be added directly to its output ($y = f(x)+x$). This helps gradients flow through deep networks during backpropagation, mitigating the vanishing gradient problem.

Code Example: Self-Attention Mechanism

This snippet demonstrates the core calculation of scaled dot-product attention.

```python
import torch
import torch.nn.functional as F
import math

def scaled_dot_product_attention(query, key, value):
    # query, key, value shape: (batch_size, seq_len, d_k)
    d_k = query.size(-1)

    # 1. Compute scores: (Q @ K^T) / sqrt(d_k)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)

    # 2. Apply Softmax to get attention weights
    attention_weights = F.softmax(scores, dim=-1)

    # 3. Multiply weights by Value: weights @ V
    output = torch.matmul(attention_weights, value)

    return output, attention_weights

# Example usage
d_model = 64
seq_len = 10
x = torch.randn(1, seq_len, d_model) # Input sequence

# In self-attention, Q, K, V usually come from the same source x
output, weights = scaled_dot_product_attention(x, x, x)
print(output.shape) # (1, 10, 64)
```

8.3.1 Self-Attention

Self-attention allows a word to "look at" other words in the same sentence to build a better representation of itself.

- **Comparison**: We compare pairwise all elements. To compute the representation for word x_3, we compare x_3 with x_1, x_2, x_3 (and prior words).
- **Roles**: Each input embedding x_i plays three roles, projected into three vectors:
  - **Query** (q_i): The current focus of attention.
  - **Key** (k_i): What the query compares against.
  - **Value** (v_i): The content to be extracted.
- **Calculation**:

$$\text{score}(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$$

$$\alpha_{ij} = \text{softmax}(\text{score})$$

$$\text{output}_i = \sum_j \alpha_{ij} v_j$$

- **Multi-Head Attention**: Instead of one set of Q, K, V, we use multiple "heads". Each head can learn different relationships (e.g., one head focuses on syntax, another on pronouns). The outputs are concatenated.

8.3.2 Positional Encoding

Since the Transformer has no recurrence or convolution, it has no inherent sense of order (it treats the sentence as a set). To fix this, we inject information about the position of tokens. We add a **Positional Embedding** vector to the Input Embedding vector. $X = \text{WordEmbeddings} + \text{PositionalEmbeddings}$.

8.3.3 The Residual Stream

An intuitive way to think about the Transformer is the **residual stream**. The input vector x_i travels through the network, and each layer (Attention, Feedforward) reads from the stream, processes information, and *adds* the result back to the stream (via residual connections). The vector evolves as it moves up, but retains its identity.

8.3.4 Language Modeling Head and Decoder-Only Architecture

The final layer of a Transformer used for language modeling (like in GPT) is the **Language Modeling Head**.

- It takes the final output vector of a token (e.g., h_N^L) from the residual stream.
- It projects it back to the vocabulary size using an "unembedding layer" (often the transpose of the input embedding matrix).
- A softmax function produces probabilities for the next token.
- For text generation, we stack multiple decoder blocks and map the input token w_i to predict w_{i+1}.

8.4 Pretrained Language Models

Transformers enabled the era of **Large Language Models (LLMs)**. Instead of training from scratch for every task, we use **Pretraining**: 1. Train a massive model on vast amounts of text using self-supervision (e.g., predict the next word). This learns general knowledge about language and the world. 2. Use this pretrained model as a **foundation** for downstream applications (fine-tuning). This approach overcomes the data scarcity problem for specific tasks and leverages the parallel processing power of Transformers.

8.5 Recap

This chapter covers the evolution of neural architectures for processing text sequences. We started with **Seq2seq (Encoder-Decoder)**, the architecture that revolutionized Machine Translation. It uses two RNNs: one to consume the input into a context vector, and another to generate the output.

The crucial upgrade to Seq2seq was **Attention**[13]. Instead of forcing the encoder to compress a whole sentence into one vector (a bottleneck), Attention allows the decoder to search through the entire source sentence at every step, focusing on the words most relevant to the current prediction.

We briefly looked at **CNNs** for NLP. While born for images, they are surprisingly effective for text classification. They act like "n-gram detectors", scanning the text with filters to find local patterns (like key phrases) regardless of their position. They are fast and efficient but struggle with long-distance dependencies compared to attention. We also saw how they can be used to create **character-aware** embeddings.

Finally, we arrived at the **Transformer**, the current state-of-the-art. The Transformer discards recurrence entirely ("Attention is All You Need").

- It processes the whole sequence in parallel (unlike RNNs).
- It uses **Self-Attention** to model relationships between all words in a sentence simultaneously, regardless of distance.
- It uses **Positional Encodings** to understand word order.
- It uses **Multi-Head Attention** to capture different types of relationships at once.

The Transformer's ability to handle long-range dependencies and train in parallel on massive datasets paved the way for **Pretrained Language Models** (like BERT and GPT), which learn general language representations from the web and serve as the foundation for modern AI.

[13] **Attention** is a mechanism that allows the model to focus on different parts of the input sequence when generating each part of the output, effectively bypassing the bottleneck of a single fixed-length context vector.

9 Large Language Models (LLMs)

Conditional Generation

Large Language Models (LLMs) have shifted the paradigm of NLP towards **conditional generation**. This is the task of generating text conditioned on an input piece of text, commonly referred to as a **prompt**. The key to effective conditional generation is **context**. Because modern LLMs (based on Transformers) can handle long contexts, they can "look back" into the prompt to understand the task, the style, or the specific information required.

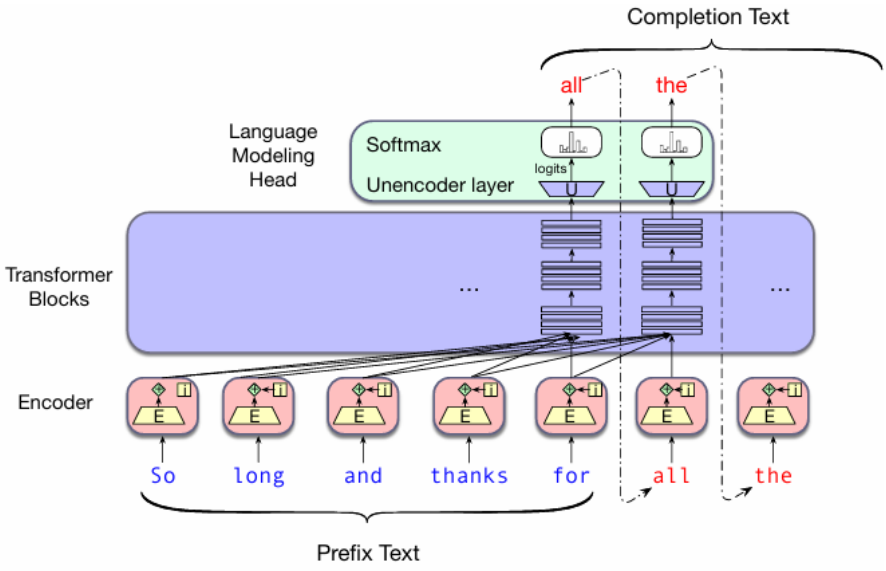
Conceptually, this is a simple task: **left-to-right text completion**. The model predicts the next word based on all previous words. However, by structuring the input (prompt) cleverly, we can solve many practical NLP tasks:

- **Sentiment Analysis**: Instead of a classifier, we ask the model to complete a sentence.
  - Input: "The sentiment of the sentence 'I like Jackie Chan' is..."
  - The model predicts the probability of the word "positive" vs "negative".
- **Question Answering** (Zero-shot[14]):
  - Input: "Q: Who wrote the book 'The Origin of Species'? A:"
  - The model completes it with: "Charles Darwin".
- **Summarization**:
  - Input: [Full Article Text] + "tl;dr:"
  - The model generates the summary.

9.0.1 Autoregressive Text Completion

LLMs like GPT use a **Decoder-only architecture**[15]. They are **autoregressive**, meaning:

1. The model takes the current sequence (prefix).
2. It generates the probability distribution for the next token.
3. A token is selected (sampled).
4. This new token is **appended** to the sequence, becoming part of the input for the next step.
5. This process repeats until a special '¡END¿' token is generated or a length limit is reached.



[14] **Zero-shot learning**: The ability of a model to perform a task without having seen any specific examples of that task during training, relying only on its general pre-training knowledge and the task description in the prompt.

[15] **Decoder-only architecture**: A type of Transformer model (like GPT) that uses only the decoder stack. Unlike the original Transformer which had an encoder for input and decoder for output, these models treat everything as a single sequence generation task using masked self-attention to prevent looking at future tokens.

9.1 Decoding Strategies

"Decoding" is the process of selecting the output tokens from the probability distributions predicted by the model. The choice of decoding strategy drastically affects the quality and creativity of the text.

Greedy Decoding

The simplest method. At each step, we simply select the token with the highest probability (argmax).

- **Pros**: Fast, deterministic.
- **Cons**: It often leads to repetitive or sub-optimal text. It suffers from the "garden path" problem: a choice that looks best *now* (locally optimal) might lead to a dead end or a bad sentence later. It misses the **globally** most probable sequence.

Beam Search

Instead of keeping just the single best token, we keep track of the k **most probable sequences** (hypotheses) at each step. k is the **beam width**.

- At each step, we expand all k hypotheses and keep the top k of the resulting combinations.
- We score hypotheses using the sum of log probabilities (to avoid numerical underflow with small probabilities).
- We normalize by length to prevent the model from preferring very short sentences (since adding more probabilities always lowers the total score).
- **Usage**: Standard in Machine Translation, where accuracy is paramount.

Sampling

For tasks like creative writing or dialogue, we want diversity, not just the single most likely (and boring) sentence. We use **random sampling**: picking the next word according to its probability distribution. However, pure random sampling can pick very rare/nonsensical words from the "tail" of the distribution. To fix this, we use:

1. **Top-k Sampling**: We truncate the distribution to the top k most likely words, renormalize the probabilities, and sample from this subset. This cuts off the unreliable tail.
2. **Top-p (Nucleus) Sampling**: Instead of a fixed number k, we pick the smallest set of words whose cumulative probability exceeds p (e.g., $p = 0.9$). This adapts the candidate pool size dynamically based on the model's confidence.
3. **Temperature Sampling**: We reshape the probability distribution using a hyperparameter τ (temperature)[16].
   - $\tau < 1$ (Low temp): Peaks become sharper. The model becomes more conservative and confident (approaches greedy decoding).
   - $\tau > 1$ (High temp): Distribution flattens. Rare words become more likely. The model becomes more creative/diverse but risks incoherence.

Code Example: Generating Text with Hugging Face

This snippet demonstrates how to load a model and generate text using different decoding strategies.

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model_name = "gpt2"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

input_text = "Once upon a time"
```

[16] **Temperature** (τ) is a hyperparameter used to control the randomness of predictions. Low temperature makes the model more confident and deterministic (peaks become sharper), while high temperature makes it more diverse and creative (distribution flattens).

```
input_ids = tokenizer.encode(input_text, return_tensors="pt")

# Greedy Decoding
greedy_output = model.generate(input_ids, max_length=50)

# Sampling with Top-k and Temperature
sample_output = model.generate(
    input_ids,
    max_length=50,
    do_sample=True,
    top_k=50,
    temperature=0.7
)

print("Greedy:", tokenizer.decode(greedy_output[0], skip_special_tokens=True))
print("Sampling:", tokenizer.decode(sample_output[0], skip_special_tokens=True)
    )
```

9.2 Pretraining Large Language Models

Pretraining is the phase where the model learns general knowledge about language and the world from vast amounts of text using **self-supervision** (predicting the next word).

Training Differences vs RNNs

- **Parallelism**: Unlike RNNs which process text sequentially, Transformers process the whole context window at once (masking future tokens). Loss calculation is not serial.
- **Context**: We fill the full context window (e.g., 2048 tokens), often packing multiple documents together separated by an ‘¡EOT¿‘ (End of Text) token.
- **Scale**: Batch sizes are massive (millions of tokens).

Training Corpora

Models are trained on hundreds of gigabytes or even terabytes of text scraped from the web.

- **Common Crawl / C4**: Massive web crawls (150B+ tokens).
- **The Pile**: A curated collection of diverse datasets (Wikipedia, PubMed, GitHub, Books, etc.).
- **Dolma**: 3 trillion tokens.

Issues: Data quality is crucial (”garbage in, garbage out”). We must filter for toxicity, remove boilerplate code, deduplicate text, and handle PII (Personally Identifiable Information). Most data is in English, creating a language bias.

Scaling Laws

Empirical laws describe the relationship between model performance (loss), model size (N), dataset size (D), and compute budget (C).

- Performance improves as a power-law with N and D.
- **Chinchilla Scaling**: To train a model optimally given a compute budget, you should scale parameters and data equally. A 70B parameter model (like Llama 3) trained on ”only” 200B tokens is undertrained; it should be trained on trillions of tokens to be optimal.

9.3 Finetuning

After pretraining, we have a ”generic” LLM. To specialize it for a specific domain (e.g., medical) or task (e.g., chat), we use **finetuning.**

Types of Finetuning

1. **Continued Pretraining**: Keep training on domain-specific text (unlabeled).
2. **Task-Specific Head**: Add a classifier layer on top and train (like with BERT).
3. **Supervised Finetuning (SFT)**: Train the model to follow instructions using a dataset of (prompt, response) pairs. This is crucial for Chatbots.

Parameter-Efficient Fine-Tuning (PEFT)
Finetuning a massive model (e.g., 175B parameters) is prohibitively expensive because you have to update all weights. **PEFT** methods freeze most of the pre-trained weights and only update a small subset of parameters.

- **LoRA (Low-Rank Adaptation)**: The most popular method. Instead of updating a large weight matrix W directly ($W+\Delta W$), we approximate the update ΔW with two smaller, low-rank matrices A and B such that $\Delta W = A \times B$.
- We only train A and B. This reduces the number of trainable parameters by orders of magnitude, allowing finetuning of huge models on consumer hardware.

9.4 Recap

This chapter explores the giants of modern NLP: **Large Language Models (LLMs)**. The core concept is **Conditional Generation**: give the model a prompt (context), and it generates the continuation. This simple mechanism, when scaled up, can solve almost any NLP task (translation, summarization, QA) without task-specific architecture changes.

We examined the **Decoding** process—how the model chooses words.

- **Greedy decoding** is simple but shortsighted.
- **Beam search** explores multiple futures to find the best overall sequence (great for translation).
- **Sampling** (Top-k, Top-p, Temperature) introduces controlled randomness to generate diverse and creative text (essential for storytelling and chatbots).

The power of LLMs comes from **Pretraining** on massive web-scale corpora (like Common Crawl or The Pile). We learned about **Scaling Laws**, which tell us that performance predictably improves with more data and larger models, provided they are balanced correctly (Chinchilla optimal).

Finally, we discussed how to adapt these giants. **Finetuning** specializes a model for a task. Since full finetuning is expensive, techniques like **LoRA (Low-Rank Adaptation)** have become standard. LoRA allows us to inject a small number of trainable parameters into a frozen model, adapting it effectively without the massive compute cost of full training. This democratization of finetuning is a key trend in current AI.

10 Masked Language Models

10.1 Bidirectional Transformer Encoders

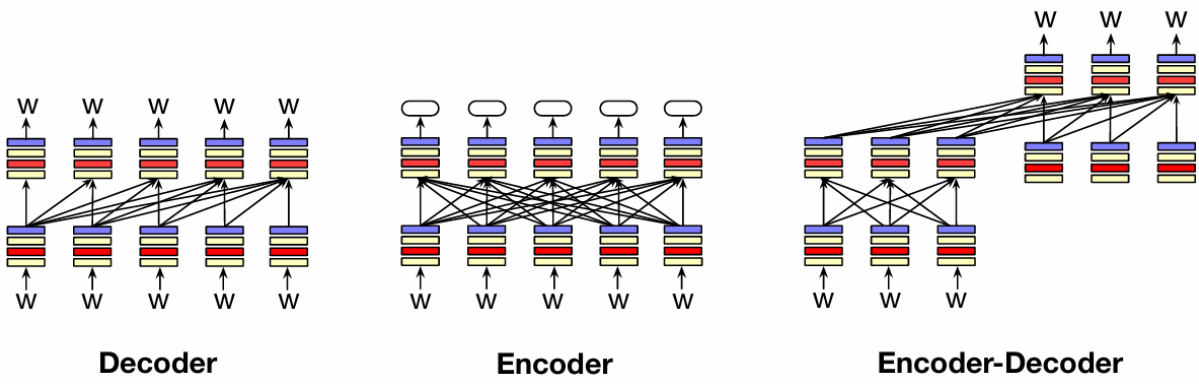
So far, we have focused on generative transformer architectures for language modeling (like GPT). These rely on **causal attention**, meaning they can only look at the past (left-to-right) to predict the next token. However, many NLP tasks benefit from **bi-directional context**. Just as we saw with bi-RNNs, knowing what comes *after* a word is often as important as knowing what came before, especially for interpretative tasks.

Architectures that consider context from both sides are called **Encoders**. They produce **contextual embeddings**[17]: representations of words that depend on the entire sentence they appear in.

Three Architectures for Language Models

- **Decoder-only (e.g., GPT)**: Autoregressive, left-to-right. Good for generation.
- **Encoder-only (e.g., BERT)**: Bidirectional attention. Good for understanding/analysis (classification, tagging).
- **Encoder-Decoder (e.g., T5, BART)**[18]: Combines both. Good for sequence-to-sequence tasks like translation.

The first widely-used transformer-based encoder model was **BERT** (Bidirectional Encoder Representations from Transformers).



Architecture Differences

The main differences between an Encoder model (BERT) and a Decoder model (GPT) are:

1. **Attention is not causal**: In a bidirectional self-attention layer, the attention mechanism can look at the entire input sequence at once. To implement this, we simply **remove the attention mask** (the upper-triangular mask of $-\infty$) that prevented looking ahead in the decoder.
2. **Training is different**: Since we can see the future, we can't use the standard next-token prediction objective (otherwise the model would just "cheat" by looking at the next word).

Examples

- **BERT**: Uses WordPiece[19] tokenization, $|V| = 30k$, 12 layers, 768 hidden dimension. Trained on English Wikipedia and BooksCorpus.

[17] **Contextual Embeddings**: Unlike static embeddings (Word2Vec) where "bank" has the same vector in "river bank" and "bank account", contextual embeddings (BERT) generate a different vector for "bank" depending on the surrounding words, capturing its specific meaning in that sentence.

[18] **Encoder-Decoder**: Architecture like T5 or BART. The encoder sees the full input (bidirectional), and the decoder generates the output autoregressively. Suitable for tasks like translation or summarization.

[19] **WordPiece**: A subword tokenization algorithm similar to BPE. It starts with characters and iteratively merges symbols that maximize the likelihood of the training data, rather than just frequency. Used by BERT.

- **XLM-RoBERTa**: A multilingual version trained on 100 languages with a larger vocabulary ($250k$).

Note: Masked Language Models (MLMs) are usually much smaller than Causal LMs (e.g., 500M parameters vs 400B+ for Llama 3).

10.2 Training Bidirectional Encoders

10.2.1 Masked Language Modeling (MLM)

To train a bidirectional model, we use a **cloze task**[20] (fill-in-the-blank). This is a form of **denoising**. Instead of predicting the next word, the model is asked to predict a missing item given the rest of the sentence (both left and right context).

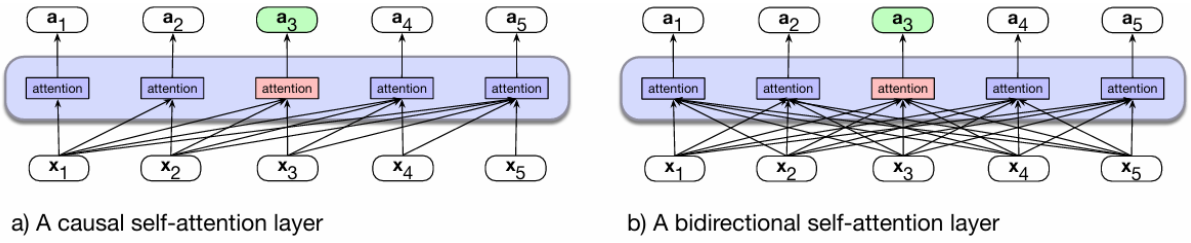


Figure 6: Information flow in a bidirectional attention model. In process in each token, the model attends to all inputs, both before and after the current one. So attention for token 3 can draw on information from following tokens.

The Protocol:

1. A random sample of tokens (typically 15%) from the training sequence is selected.
2. These chosen tokens are manipulated:
   - 80% of the time: Replaced with the special token '[MASK]'.
   - 10% of the time: Replaced with a random word (to teach the model that not all inputs are correct).
   - 10% of the time: Left unchanged (to bias the representation towards the actual word).
3. The model must predict the original token for these masked positions.
4. The loss (Cross-Entropy) is calculated **only** on the masked tokens.

This process is somewhat inefficient compared to autoregressive training (where every token is a target), but it creates powerful bidirectional representations.

Training Issues

- **Multilinguality**: Training on multilingual data (like Common Crawl) helps low-resource languages but suffers from the "curse of multilinguality" (capacity dilution).
- **Tokenizer Fertility**: Measures how many tokens are produced per word. High fertility (many tokens for one word) slows down inference.

[20] **Cloze task**: A linguistic test where a participant is asked to supply words that have been removed from a passage. In NLP, this inspired the Masked Language Modeling objective.

10.3 Attention Mask

How do we practically tell the model whether it can look at future tokens or not? We use an **Attention Mask**.[21] Recall that the core of the attention mechanism involves computing the dot product of queries and keys to get raw scores: QK^T. This results in an $N \times N$ matrix where cell (i, j) represents the score between token i and token j.

- **Causal Masking (Decoders)**: To prevent the model from looking into the future, we add a mask matrix M to the scores before applying softmax. In this mask, all entries where $j > i$ (future tokens) are set to $-\infty$. When the softmax is applied, these values become 0.

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)$$

- **No Masking (Encoders)**: To implement bidirectional attention, we simply **do not apply this mask**. This allows the softmax to distribute probability mass over the entire sequence, letting token i attend to token j regardless of their positions.

This simple matrix operation switches the architecture from a GPT-style decoder to a BERT-style encoder.

10.4 Contextual Embeddings

The output of a BERT-style model is a sequence of vectors h_i^L. Unlike static embeddings (Word2Vec), these are **contextual**. The vector for "die" in "The German article 'die'" will be very different from the vector for "die" in "single person dies", because the self-attention mechanism aggregates information from the specific context. Visualizing these embeddings (e.g., with UMAP) shows that different senses of polysemous words form distinct clusters.

Code Example: Contextual Embeddings with BERT

This code shows how the same word "bank" gets different vectors depending on the sentence.

```python
from transformers import BertTokenizer, BertModel
import torch
from scipy.spatial.distance import cosine

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

text1 = "I went to the bank to deposit money."
text2 = "We sat on the river bank."

# Get embeddings
def get_embedding(text):
    inputs = tokenizer(text, return_tensors="pt")
    outputs = model(inputs)
    # Get embedding for "bank" (index 5 in text1, index 5 in text2)
    return outputs.last_hidden_state[0][5].detach().numpy()

bank_money = get_embedding(text1)
bank_river = get_embedding(text2)

# Similarity should be lower than 1.0 because context differs
similarity = 1 - cosine(bank_money, bank_river)
print(f"Similarity: {similarity:.4f}")
```

[21] **Attention Mask**: A matrix used to control which tokens the model is allowed to "see". In practice, it adds negative infinity ($-\infty$) to the attention scores of forbidden tokens (like future words in GPT or padding tokens), forcing their probability to zero after the softmax operation.

Word Sense Disambiguation (WSD)

Because of this property, BERT embeddings are excellent for WSD. A simple Nearest-Neighbor approach works well: 1. Compute the "average" embedding for each sense of a word from a labeled corpus. 2. For a new target word, compute its contextual embedding. 3. Assign the sense whose average embedding is closest (cosine similarity).

10.5 Fine-Tuning for Classification

Once pretrained, BERT can be fine-tuned for downstream tasks.

Text Classification

To classify an entire sentence (e.g., Sentiment Analysis):

- We add a special token '[CLS]' at the beginning of the input.
- The output vector corresponding to '[CLS]' is treated as the representation of the entire sequence.
- We add a simple classifier head (Feedforward + Softmax) on top of this vector: $y = \text{softmax}(h^L_{[CLS]} W_C)$.
- We train the whole model (encoder + head) on the labeled dataset using Cross-Entropy loss.

Sequence-Pair Classification

Tasks like Paraphrase Detection or Natural Language Inference (NLI) require comparing two sentences.

- Input: '[CLS] Sentence A [SEP] Sentence B'.
- The '[SEP]' token acts as a separator.
- The model uses self-attention to compare words across both sentences.
- Classification is again done using the '[CLS]' token.

10.6 Fine-Tuning for Sequence Labeling

Tasks like **Named Entity Recognition (NER)** or POS tagging require a label for every token.

- We pass the output vector h^L_i of *each* token to a classifier.
- We learn a set of weights W_K to project the vector to the number of tags (e.g., B-PER, I-ORG, O).
- Alternatively, we can feed the outputs to a CRF (Conditional Random Field) layer to model valid tag transitions.

Evaluation

We evaluate these models on benchmarks like GLUE (General Language Understanding Evaluation), which aggregates multiple tasks (sentiment, similarity, inference) to test general linguistic knowledge. Performance is measured using metrics like Accuracy, F1-score, or Matthews Correlation, depending on the task balance.

10.7 Recap

This chapter completes our tour of Transformer architectures by focusing on **Masked Language Models (MLMs)**, or Encoders, with **BERT** as the prime example.

While GPT (Decoder) is built for generating text (predicting the future), BERT (Encoder) is built for *understanding* text. It uses bidirectional attention, meaning it can look at the whole sentence at once—left and right context simultaneously. This switch is implemented simply by removing the **Attention Mask** (the matrix of $-\infty$) that prevents decoders from peeking at future tokens.

Because it can "see the future", we can't train it by predicting the next word. Instead, we use the **Masked Language Modeling** objective: we hide some words (cloze task) and ask the model to fill in the blanks based on the surrounding context. This forces the model to learn deep, contextual relationships between words.

The result is **Contextual Embeddings**. Unlike Word2Vec, where a word has a static vector, BERT produces a dynamic vector that changes meaning based on context. This solves polysemy (e.g., "bank" of a river vs. "bank" for money).

We use these pretrained encoders by **Fine-Tuning** them:

- For **Classification** (e.g., sentiment), we use the special `[CLS]` token as a summary of the whole sentence.
- For **Sequence Labeling** (e.g., NER), we classify every token individually.
- For **Sentence Pairs** (e.g., entailment), we feed two sentences separated by `[SEP]` and let the attention mechanism cross-reference them.

This "Pretrain-then-Finetune" paradigm became the standard in NLP (until the recent rise of zero-shot generative models), allowing us to reach state-of-the-art performance on benchmarks like GLUE with relatively small amounts of labeled data.

11 Post-training: Instruction Tuning, Model Alignment, and Test-Time Compute

Why Pretraining Isn't Enough

"Foundation" models are trained with a simple objective: predict the next word on massive datasets. While this teaches them language and facts, it doesn't make them helpful assistants.

- **Misunderstanding intent**: If prompted with "Explain the moon landing...", a base model might continue with "...to a six year old in a few sentences" (completing the pattern) rather than actually answering the question.
- **Harmful outputs**: Pretraining data contains bias, toxicity, and falsehoods. Without alignment, models can generate dangerous or offensive content.

To fix this, we need Post-training steps: Instruction Tuning (to make models helpful) and Preference Alignment (to make them safe and aligned with human values).

11.1 Instruction Tuning (Supervised Fine-Tuning - SFT)

Instruction Tuning transforms a base model into a chat model. It is a form of supervised learning where the model is trained on pairs of '(instruction, response)'. The objective remains Cross-Entropy Loss (next-token prediction), but the data is different: it consists of explicit commands and the desired behavior.

Data Resources and Construction

Obtaining high-quality instruction data is challenging. The main strategies are:

1. Human Annotation: Hiring experts or crowdworkers to write prompts and perfect answers (e.g., the Aya dataset, with 204k examples in 65 languages).
2. Repurposing NLP Datasets: We can convert existing labeled datasets (like SQuAD for QA or MNLI for inference) into instruction format.
    - Crucially, we use Templates to vary the phrasing. A single task like "Is the sentiment positive?" can be turned into "How does the reviewer feel?", "Rate this movie", "Classify the sentiment", etc. This variety teaches the model to be robust to different user prompts.
3. Synthetic Generation (Model-in-the-loop): Using a strong LLM to generate training data for a weaker one.
    - Example: Bonito. This method takes unannotated text and uses a model to generate a synthetic instruction/question about that text, creating a new training pair '(generatedquestion, originaltext)'.

Instruction Tuning as Meta-Learning

By training on a wide variety of tasks (translation, summarization, logic) phrased as instructions, the model learns not just those specific tasks, but the *general ability to follow instructions*. This allows it to perform well on unseen tasks (Zero-shot generalization).

11.2 Learning from Preferences (Alignment)

Instruction tuning makes models helpful, but we also need to align them with nuanced human preferences. It is hard to write a mathematical loss function for abstract qualities like "helpfulness", "honesty", or "safety". Instead, we rely on human feedback.

Human Feedback Sources

- Annotator Judgments: Humans compare two model outputs and select the better one. Ranking ($A > B$) is faster and more reliable than giving absolute scores (Likert scale 1-5).
- Implicit Signals: Data from platforms like Reddit or StackExchange (upvotes/downvotes) can be treated as preference signals (highly upvoted reply > downvoted reply).

Reward Modeling

Since humans can't grade every single output during training, we train a Reward Model (RM) to act as a proxy.

- **Input**: A prompt x and a model output o.
- **Output**: A scalar score $r(x, o)$.
- **Training Data**: Pairs of outputs (o_w, o_l) where a human preferred o_w (winner) over o_l (loser).
- **Bradley-Terry Model**[22]: We model the probability of preference as a logistic function of the score difference:
$$P(o_w \succ o_l|x) = \sigma(r(x, o_w) - r(x, o_l))$$
- **Loss**: We minimize the negative log likelihood of the human preferences.

Reinforcement Learning from Human Feedback (RLHF)

Once we have a Reward Model, we use it to fine-tune the LLM using Reinforcement Learning (specifically PPO - Proximal Policy Optimization[23]).

- **Policy (π)**: The LLM is the agent. Its action is generating the next token.
- **Reward**: The score from the Reward Model at the end of the sequence.
- **KL Divergence Penalty**: A critical component. We add a penalty term to the reward equation proportional to the KL divergence between the current policy π and the original reference model π_{ref} (the SFT model).
$$R(x, o) = r_{RM}(x, o) - \beta \log \frac{\pi(o|x)}{\pi_{ref}(o|x)}$$
- **Why KL?**: This prevents Reward Hacking (where the model finds "loopholes" in the reward model, producing gibberish that gets a high score) and keeps the model fluent and close to its original training distribution.

11.3 In-Context Learning (ICL) and Prompt Engineering

Another way to improve performance without updating weights is Test-Time Compute: giving the model more time or information in the prompt to "think". This exploits In-Context Learning: the ability of LLMs to learn a pattern from the prompt itself without gradient updates. Larger models (e.g., 175B parameters) exhibit this "emergent ability" much more strongly than smaller ones.

Prompt Engineering Techniques

- Zero-shot: Just the task description. "Translate to French: cheese".
- One-shot / Few-shot: Providing one or a few examples (demonstrations) in the context. This dramatically improves performance by showing the model the expected format and style.
- Chain-of-Thought (CoT): Asking the model to "think step by step" before giving the answer. This elicits reasoning capabilities, especially for math and logic tasks where intermediate steps are crucial.

[22] **Bradley-Terry Model**: A probability model for predicting the outcome of a paired comparison. It assumes that the probability of item i beating item j depends on the difference between their latent scores or strengths.

[23] **Proximal Policy Optimization (PPO)**: A reinforcement learning algorithm used to fine-tune LLMs. It optimizes the policy (the model) to maximize the expected reward while ensuring the new policy doesn't deviate too drastically from the old one, ensuring training stability.

Code Example: Prompt Templates and Chain-of-Thought

This snippet shows how structured prompting works in practice.

```python
# 1. Basic Prompt Template (Zero-shot)
def sentiment_prompt(text):
    return f"Review: {text}\nSentiment (Positive/Negative):"

print(sentiment_prompt("I loved this movie!"))
# Output: Review: I loved this movie!
#         Sentiment (Positive/Negative):

# 2. Chain-of-Thought Prompting (Few-shot)
cot_prompt = """
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls.
   Each can has 3 tennis balls. How many tennis balls does he have now?
A: Roger started with 5 balls. 2 cans of 3 balls each is 6 balls.
   5 + 6 = 11. The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and
   bought 6 more, how many apples do they have?
A:
"""
# The model is expected to generate the reasoning step before the answer.
```

Mechanism: Induction Heads

How does ICL work? A key hypothesis is Induction Heads[24]. These attention heads learn a pattern-matching behavior: if they see sequence $[A][B]...[A]$, they predict $[B]$. A related benchmark for testing this capability is the Needle-in-the-Haystack[25] test.

Advanced Prompting

- Prefix Tuning / Soft Prompts: Instead of searching for discrete words to prompt the model, we can optimize a continuous vector (prefix) that is prepended to the input embeddings. This is a form of parameter-efficient fine-tuning where the "prompt" is learned by gradient descent.

11.4 Recap

This chapter focuses on what happens *after* the massive pretraining phase. A pretrained LLM is like a raw diamond: it has vast knowledge but lacks direction and polish.

Instruction Tuning (SFT) gives it direction. By training on curated datasets of instructions and answers (often created via templates or synthetic generation like Bonito), we turn the model into a helpful assistant that follows commands rather than just completing text.

Alignment (RLHF) gives it polish and safety. Since we can't easily write a loss function for "helpfulness" or "safety", we rely on human preferences. We train a Reward Model to act as a proxy for human judgment (using the Bradley-Terry model), and then use Reinforcement Learning (PPO) to optimize the LLM to maximize this reward. Crucially, we use KL Divergence to keep the model anchored to reality and prevent it from "gaming" the reward system.

Finally, we explored Test-Time Compute or Prompt Engineering. We don't always need to train the model to improve performance. By cleverly structuring the input—giving examples (Few-Shot) or asking for reasoning (Chain-of-Thought)—we can unlock capabilities that are latent in the model. This In-Context Learning is a surprising emergent property of large transformers, likely driven by mechanisms like Induction Heads.

[24] **Induction Heads**: A specific type of attention head in Transformers that implements a copy-paste mechanism. They look for previous occurrences of the current token (A) and attend to the token that followed it (B), predicting that B will follow A again. This is believed to be the mechanism behind in-context learning.

[25] **Needle-in-the-Haystack**: A benchmark task designed to test an LLM's ability to retrieve a specific piece of information (the "needle") hidden within a very long context window (the "haystack").

12 Small Language Models

12.1 The Cost of Progress

Since the advent of pre-training (e.g., BERT), NLP has seen rapid progress, surpassing human baselines on benchmarks like GLUE[26]. However, this performance comes at a steep price.

- Exponential Growth: Models have grown from millions to hundreds of billions of parameters.
- Compute Cost: Training models like BERT-Large requires vastly more FLOPs[27] than earlier models like ELMo. RoBERTa uses 16x more compute than BERT-Large.
- Barriers: This creates barriers for academia and smaller companies who cannot afford large GPU clusters.
- Environmental Impact: The carbon footprint of training and maintaining these models is significant.
- Inference Latency: Even fine-tuning or just running inference on massive models can be too slow for real-time applications.

We need solutions to reduce model size without sacrificing too much performance.

12.2 Approaches to Reducing Model Size

There are several strategies to create "Small" Language Models (SLMs) that are lighter and faster:

1. Parameter Sharing: Sharing weights between layers (e.g., ALBERT) to reduce the total number of unique parameters.
2. Pruning: Removing weights (setting them to zero) that have little impact on the loss (saliency). However, sparse matrices are often hard to optimize on modern hardware.
3. Mixed Precision / Quantization: Using fewer bits to represent weights (e.g., 16-bit float or 4-bit integer via QLoRA[28]).
4. Mixture of Experts (MoE): Using a router to activate only a subset of parameters (experts) for each token, keeping inference fast while having high total capacity (e.g., Mixtral).
5. Knowledge Distillation: Training a smaller "student" model to mimic a larger "teacher".

12.3 Knowledge Distillation

Introduced by Hinton et al. (2015), this is a general compression method. The core idea is that the "soft targets" (probability distributions) produced by a large Teacher model contain much more information (dark knowledge) than the hard class labels (0 or 1).

- Temperature (τ): We divide the logits by a temperature parameter $\tau > 1$ before softmax to "soften" the distribution, revealing relationships between incorrect classes (e.g., a picture of a BMW might have a small probability of being a truck, but near zero of being a cat).
- Loss Function: The Student model is trained to minimize a combination of:
  - Distillation Loss: Distance (KL Divergence) between Student's soft predictions and Teacher's soft predictions.
  - Student Loss: Standard cross-entropy against the ground truth labels.

Famous examples include DistilBERT, TinyBERT, and DistilRoBERTa.

[26] **GLUE (General Language Understanding Evaluation)**: A collection of resources for training, evaluating, and analyzing natural language understanding systems. It includes tasks like sentiment analysis, textual entailment, and question answering.

[27] **FLOPs (Floating Point Operations)**: A measure of computer performance, useful in fields of scientific computations that require floating-point calculations. In deep learning, it is used to estimate the computational cost of training a model.

[28] **Quantization**: The process of mapping input values from a large set (often continuous) to output values in a (smaller) finite set. In neural networks, it typically involves reducing the precision of the weights (e.g., from 32-bit float to 8-bit or 4-bit integer) to save memory and computation.

Code Example: Knowledge Distillation Loss

This snippet demonstrates how to implement the loss function for distilling knowledge from a teacher model to a student model.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

def distillation_loss(student_logits, teacher_logits, labels, T, alpha):
    """
    student_logits: Logits from the student model
    teacher_logits: Logits from the teacher model
    labels: True hard labels
    T: Temperature for softening probability distributions
    alpha: Weight for the distillation loss (vs classification loss)
    """
    # Soften probabilities
    soft_targets = F.softmax(teacher_logits / T, dim=1)
    soft_prob = F.log_softmax(student_logits / T, dim=1)

    # Distillation loss (KL Divergence between soft distributions)
    # Note: nn.KLDivLoss expects log-probabilities as input
    distill_loss = nn.KLDivLoss(reduction='batchmean')(soft_prob, soft_targets)

    # Student loss (Standard Cross-Entropy with hard labels)
    student_loss = F.cross_entropy(student_logits, labels)

    # Weighted sum
    # T^2 factor scales gradients to match magnitude of hard loss gradients
    return alpha * (T * T) * distill_loss + (1 - alpha) * student_loss
```

12.4 Vocabulary Transfer

Another way to reduce model size is focusing on the vocabulary. Pretrained models use general-purpose subword tokenizers (like WordPiece or BPE[29]) with huge vocabularies (30k-250k tokens). However, for a specific vertical domain (e.g., medicine), the language distribution shifts.

- Standard tokenizers might break domain-specific terms into many small fragments (e.g., "interferon" $\rightarrow$ "inter", "##fer", "##on"), increasing sequence length and inference time.
- Ad-hoc Tokenization: We can train a new tokenizer $\mathcal{T}_{in}$ specifically for the target domain $\mathcal{D}_{in}$. This reduces the average sequence length.

12.4.1 Fast Vocabulary Transfer (FVT)

The problem with a new tokenizer is that the pre-trained embeddings become useless because the tokens don't match. FVT solves this by transferring knowledge from the old general embeddings (E_{gen}) to the new domain embeddings (E_{in}).

- If a token t_i exists in both vocabularies, copy the embedding.
- If a new token t_i is composed of multiple old tokens t_j, we initialize $E_{in}(t_i)$ as the average of the embeddings $E_{gen}(t_j)$.
- This allows us to resize the embedding matrix (reducing parameters) and process text faster (shorter sequences).

This technique can be combined with Knowledge Distillation (distilling first on general data, then on in-domain data) for maximum efficiency.

[29] **Byte-Pair Encoding (BPE)**: A subword tokenization method that iteratively merges the most frequent pair of bytes (or characters) in a sequence. It allows models to handle rare words by breaking them down into common subword units.

12.5 Recap

This chapter addresses the "elephant in the room" of modern NLP: Efficiency. While scaling laws tell us that "bigger is better" for performance, real-world constraints (hardware, latency, cost, energy) force us to look for Small Language Models (SLMs).

We explored two main avenues to achieve this: 1. Compression via Knowledge Distillation: Instead of training a small model from scratch (which might not learn enough), we force it to mimic a large, smart "Teacher". By learning the teacher's probability distribution (soft targets), the small "Student" learns the teacher's generalization capabilities, not just the right answers. 2. Optimization via Vocabulary Transfer: We realized that general-purpose tokenizers are inefficient for specific domains. By creating a domain-specific vocabulary (e.g., treating "interferon" as one token instead of three) and using Fast Vocabulary Transfer (FVT) to initialize the new embeddings from the old ones, we can shrink the model's input layer and speed up processing without losing the pre-training knowledge.

These techniques prove that we don't always need 100B+ parameters. For specific applications, a well-optimized, distilled small model can be fast, cheap, and incredibly effective.

13 Information Retrieval and Question Answering

Introduction

Question Answering (QA) has traditionally focused on factoid questions[30]. While modern QA relies heavily on LLMs, they suffer from hallucinations (giving wrong answers with high confidence) and lack access to private or rapidly changing data. The mainstream solution is Retrieval: finding relevant context first, and then generating an answer based on it.

13.1 Information Retrieval (IR)

Ad-hoc retrieval is the task where a user poses a query to a system (like a search engine), which returns an ordered set of relevant documents from a collection.

13.1.1 Sparse Retrieval (TF-IDF & BM25)

The classical approach uses sparse vectors based on word frequencies.

- TF-IDF: Weights terms by how frequent they are in the document (TF) vs how rare they are in the whole collection (IDF).

$$\text{score}(q, d) = \sum_{t \in q} \frac{\text{tf-idf}(t, d)}{|d|}$$

- Okapi BM25: A more robust variant of TF-IDF that adds parameters to tune term saturation (k) and length normalization (b). It remains a very strong baseline.

To make search efficient, we use an Inverted Index[31], which maps each term to the list of documents containing it (postings list).

Evaluation Metrics

Since IR returns a ranked list, simple accuracy is not enough.

- Precision: Fraction of retrieved docs that are relevant.
- Recall: Fraction of relevant docs that were retrieved.
- Precision-Recall Curve: Plots precision at different recall levels.
- Mean Average Precision (MAP): Summarizes the precision-recall curve into a single number.

13.2 Dense Retrieval

Sparse methods suffer from the vocabulary mismatch problem (e.g., searching for "cat" won't find "feline"). Dense Retrieval solves this by using dense embeddings (like BERT) to capture semantic meaning. There are two main architectures:

1. Bi-Encoder: Encodes the query and the document separately into two vectors. Similarity is the dot product.

$$\text{score}(q, d) = \text{BERT}_Q(q) \cdot \text{BERT}_D(d)$$

   - **Pros**: Very fast. Document vectors can be pre-computed and stored in a vector database (searched via Faiss[32]).
   - **Cons**: Less accurate because query and document don't interact deeply.

2. Cross-Encoder: Feeds both query and document into a single BERT model: '[CLS] Query [SEP] Document'.

$$\text{score}(q, d) = \text{Linear}(\text{BERT}(q, d)_{[CLS]})$$

[30] **Factoid questions**: Questions that can be answered with simple facts, usually expressed in short texts or single entities (e.g., "Who invented the telephone?" -¿ "Alexander Graham Bell").

[31] **Inverted Index**: A data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents. It is the core component of search engine indexing algorithms.

[32] **Faiss (Facebook AI Similarity Search)**: A library for efficient similarity search and clustering of dense vectors. It contains algorithms that search in sets of vectors of any size, up to ones that possibly do not fit in RAM.

- **Pros**: Highly accurate (full attention between query and doc words).
- **Cons**: Very slow (must re-compute for every query-doc pair). Impractical for large collections.

Standard Pipeline: Use a Bi-Encoder (or BM25) to retrieve a top-k shortlist, then use a Cross-Encoder to re-rank them for high precision.

Code Example: Dense Retrieval with Sentence Transformers

This snippet shows how to encode queries and documents into dense vectors and find the most relevant document using cosine similarity.

```python
from sentence_transformers import SentenceTransformer, util

model = SentenceTransformer('all-MiniLM-L6-v2')

# Corpus of documents
docs = [
    "The capital of France is Paris",
    "The capital of Italy is Rome",
    "London is known for its red buses"
]

# Encode documents and query
doc_embeddings = model.encode(docs)
query_embedding = model.encode("What is the capital of Italy?")

# Compute cosine similarity
scores = util.cos_sim(query_embedding, doc_embeddings)

# Find best match
best_doc_idx = scores.argmax()
print(f"Best document: {docs[best_doc_idx]}")
# Output: The capital of Italy is Rome
```

13.3 Recap

This chapter bridges the gap between searching for documents and finding answers. We started with Information Retrieval (IR). The classic methods (TF-IDF, BM25) rely on keyword matching and inverted indexes. They are fast and explainable but fail when words don't match exactly (vocabulary mismatch). Dense Retrieval solves this by embedding queries and documents into a semantic vector space using Bi-Encoders. This allows finding documents that match the *meaning*, not just the words. Ideally, we combine speed and accuracy with a Retrieve-then-Rerank pipeline (Bi-Encoder for retrieval, Cross-Encoder for reranking).

This retrieval capability is the engine behind modern Question Answering. Instead of trying to memorize all world knowledge in its weights (which leads to hallucinations and stale data), an AI system uses the Retrieve-and-Read pattern.
In the past, "Reading" meant extracting a specific span of text (Extractive QA with BERT). Today, it means Retrieval-Augmented Generation (RAG): feeding the retrieved text into a generative LLM to synthesize a complete, fluent answer. RAG is currently the standard architecture for building reliable, knowledgeable AI assistants.