# Natural Language Processing

Lecture Notes

Master M2 — 2025–2026

*Yaë Ulrich Gaba*

*“The limits of my language mean the limits of my world.”*
*— Ludwig Wittgenstein*

March 25, 2026

# Contents

# Preface

## Course Objectives

*Natural Language Processing* (NLP) is one of the most dynamic areas of contemporary artificial intelligence. This graduate-level course (Master/PhD) covers classical mathematical foundations through to recent large language models.

The course has three primary objectives:

1. **Understand theoretical foundations** — probabilistic language modeling, information theory, linear algebra for vector representations.
2. **Master modern architectures** — recurrent networks, attention mechanisms, Transformers, pre-trained models (BERT, GPT, T5).
3. **Develop critical thinking** — rigorous evaluation, ethical considerations, algorithmic bias, generative system safety.

## Target Audience

This course is designed for Master's and PhD students in computer science, data science, or computational linguistics. The following prerequisites are assumed:

- **Mathematics**: linear algebra (vector spaces, matrix decompositions), multivariate calculus, probability and statistics.
- **Computer science**: Python programming, basic algorithms, familiarity with scientific libraries (NumPy, PyTorch or TensorFlow).
- **Machine learning**: regression, classification, gradient descent, basic neural networks.

## Course Organization

The course is organized into eleven chapters, following a logical progression from classical text representations to the most recent generative systems:

**Chapter 1 — Introduction to NLP.** Historical overview, fundamental tasks, challenges specific to natural language.

**Chapter 2 — Classical Text Representations.** Bag-of-words models, TF-IDF, $n$-grams, sparse representations.

**Chapter 3 — Word Embeddings.** Word2Vec (CBOW, Skip-gram), GloVe, FastText; geometric properties of embedding spaces.

**Chapter 4 — Sequence Models.** Recurrent neural networks (RNN), LSTM, GRU; vanishing and exploding gradient problems.

**Chapter 5 — Attention and Transformers.** Attention mechanism, self-attention, complete Transformer architecture.

**Chapter 6 — Pre-trained Models.** BERT, GPT, T5; the pre-training/fine-tuning paradigm.

**Chapter 7 — Fine-tuning and Instruction Tuning.** Adaptation strategies, LoRA, RLHF, alignment.

**Chapter 8 — Text Generation and Decoding.** Greedy decoding, beam search, sampling, top-$k$, top-$p$.

**Chapter 9 — Retrieval-Augmented Generation (RAG).** Passage retrieval, external knowledge integration, RAG pipelines.

**Chapter 10 — LLM Evaluation.** Automatic metrics, human evaluation, benchmarks, robustness.

**Chapter 11 — Ethics, Bias, and Safety.** Data and model bias, toxicity, privacy, regulation.

## Typographic Conventions

- Formal **definitions** are framed in `definition` environments.
- **Theorems**, **propositions**, and **lemmas** are numbered and, when relevant, accompanied by a proof.
- **Intuition** boxes provide qualitative understanding before formalization.
- **Warning** boxes highlight common mistakes.
- **Best practice** boxes summarize practical recommendations.
- Python code primarily uses the `transformers` (Hugging Face), `torch`, and `scikit-learn` libraries.

## Mathematical Notation

We adopt the following notation throughout the course:

| **Notation** | **Meaning** |
|---|---|
| $\mathbb{R}^d$ | Euclidean space of dimension $d$ |
| $\mathbb{N}$ | Set of natural numbers |
| $\mathbf{x}, \mathbf{W}$ | Vectors (bold lowercase), matrices (bold uppercase) |
| $\langle \mathbf{u}, \mathbf{v} \rangle$ | Inner product of $\mathbf{u}$ and $\mathbf{v}$ |
| $\|\|\mathbf{x}\|\|$ | Euclidean norm of $\mathbf{x}$ |
| $\|S\|$ | Cardinality of set $S$ |
| $\mathbb{E}[X]$ | Expectation of random variable $X$ |
| $P(A \mid B)$ | Conditional probability of $A$ given $B$ |
| $\mathrm{KL}(p\|\|q)$ | Kullback-Leibler divergence from $p$ to $q$ |
| $\sigma(\cdot)$ | Logistic sigmoid function |
| $\mathrm{softmax}(\cdot)$ | Softmax function |
| $\arg\max_x f(x)$ | Argument maximizing $f$ |
| $\arg\min_x f(x)$ | Argument minimizing $f$ |
| $\mathcal{V}$ | Vocabulary (finite set of tokens) |
| $V = \|\mathcal{V}\|$ | Vocabulary size |
| $\mathbf{E} \in \mathbb{R}^{V \times d}$ | Embedding matrix |
| $T$ | Sequence length |

## Software and Environment

Labs and code examples rely on the following ecosystem:

- **Python** $\geq 3.10$
- **PyTorch** $\geq 2.0$
- **Hugging Face Transformers** $\geq 4.35$
- **Hugging Face Datasets**
- **scikit-learn** for classical methods
- **NLTK** and **spaCy** for linguistic preprocessing
- **Matplotlib** / **seaborn** for visualization

**Reproducible environment**

We recommend using a virtual environment (`venv` or `conda`) and fixing random seeds (`torch.manual_seed`, `random.seed`, `numpy.random.seed`) to ensure experiment reproducibility.

## Historical Perspective

NLP has undergone several paradigm shifts. In the 1950s, early approaches were based on manually coded linguistic rules. The 1990s saw the rise of statistical methods (hidden Markov models, $n$-gram models). The 2010s were dominated by deep learning (word embeddings, recurrent networks). Since 2017, the Transformer architecture has ushered in the era of large language models (LLMs), radically transforming the landscape of the field.

**Why is language hard?**

Natural language is inherently ambiguous (polysemy, anaphora, ellipsis), contextual (meaning depends on pragmatic context), and compositional (sentence meaning depends on syntactic structure and constituent meanings). These properties make NLP fundamentally different from processing structured data.

## How to Use This Course

Each chapter is designed to be self-contained while building on concepts from previous chapters. We recommend sequential reading for a first pass, then targeted use as a reference.

Exercises at the end of each chapter are classified into three difficulty levels:

- $\star$ Comprehension and verification exercises.
- $\star\star$ Deepening exercises requiring mathematical reasoning or implementation.
- $\star\star\star$ Open research exercises that can serve as starting points for projects or theses.

## Acknowledgments

This course has benefited from the contributions of many colleagues, researchers, and students. We particularly thank the open-source communities of Hugging Face, PyTorch, and spaCy, whose tools make NLP teaching both rigorous and accessible.

We also thank students from previous cohorts whose questions and feedback have greatly improved the quality of this pedagogical material.

*[Author Name], March 2026*

# Chapter 1

# Introduction to NLP

In 1950, Alan Turing posed the founding question of artificial intelligence: can a machine "think"? And he proposed an operational criterion: if a human, conversing with a machine via text, cannot tell it apart from another human, then the machine "thinks" in a functional sense. This "Turing test" is fundamentally a *natural language processing* challenge: to pass it, a machine must understand syntax, semantics, context, irony, and subtext. Seventy-five years later, large language models (GPT, Claude, LLaMA) seem to be approaching this threshold — yet the fundamental challenges remain intact.

**Why study NLP?**

Language is the primary vehicle for human communication. Teaching machines to understand, produce, and reason about natural language is one of the fundamental challenges of artificial intelligence. Applications range from machine translation to information extraction, conversational agents, and text generation.

## 1.1 What is NLP?

**Definition 1.1** (Natural Language Processing)**.** *Natural Language Processing* (NLP) is the subfield of artificial intelligence that studies interactions between computers and human language. It encompasses the analysis, understanding, and generation of text and speech in natural languages.

NLP is distinguished from *computational linguistics* by its application-oriented focus, although both fields share a common theoretical foundation. Formally, we can view language as a generative system defined over a vocabulary $\mathcal{V}$:

$$\mathcal{L} \subseteq \mathcal{V}^* = \bigcup_{n=0}^{\infty} \mathcal{V}^n$$

where $\mathcal{V}^*$ denotes the set of all finite sequences over $\mathcal{V}$.

## 1.2 Levels of Linguistic Analysis

Natural language processing operates at several levels of analysis:

1. **Phonetics and phonology**: study of speech sounds (primarily for speech processing).
2. **Morphology**: internal structure of words (prefixes, suffixes, inflections). For example, "un-believe-able" decomposes into `un-` + `believe` + `-able`.
3. **Syntax**: sentence structure, grammatical relations.
4. **Semantics**: meaning of words and sentences.
5. **Pragmatics**: meaning in context, speaker intentions.
6. **Discourse**: coherence and structure beyond the sentence.

*Remark* 1.2. In modern NLP, deep models implicitly learn these levels of analysis from data, without explicit segmentation. However, understanding these levels remains essential for diagnosing errors and designing relevant evaluations.

## 1.3 Fundamental NLP Tasks

### 1.3.1 Text Classification

Text classification assigns a label $y \in \mathcal{Y}$ to a document $\mathbf{x} = (x_1, x_2, \ldots, x_T)$. The model learns a function $f : \mathcal{V}^* \to \mathcal{Y}$ from a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$.

Applications: sentiment analysis, spam detection, topic classification.

### 1.3.2 Sequence Labeling

Sequence labeling assigns a label $y_t \in \mathcal{Y}$ to each token $x_t$ in a sequence:

$$f : \mathcal{V}^T \to \mathcal{Y}^T$$

Applications: named entity recognition (NER), part-of-speech (POS) tagging.

**Example 1.3.** Consider the sentence "Mary lives in Paris" with the following NER labeling:

| Mary | lives | in | Paris |
|---|---|---|---|
| `B-PER` | `O` | `O` | `B-LOC` |

### 1.3.3 Machine Translation

Machine translation is a sequence-to-sequence transduction task:

$$f : \mathcal{V}^*_{\text{source}} \to \mathcal{V}^*_{\text{target}}$$

It is historically one of the driving tasks of NLP.

### 1.3.4 Question Answering

Given a context $C$ and a question $Q$, the system must extract or generate an answer $A$:

$$f : (C, Q) \to A$$

### 1.3.5 Automatic Summarization

Automatic summarization produces a condensed version $S$ of a document $D$, preserving essential information: $f : D \to S$ with $|S| \ll |D|$.

## 1.4 Probabilistic Foundations

Probabilistic language modeling is at the heart of NLP. A *language model* defines a probability distribution over token sequences.

**Definition 1.4** (Language Model)**.** A *language model* is a probability distribution $P$ over $\mathcal{V}^*$ such that:

$$P(\mathbf{x}) = P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1})$$

where the factorization follows from the chain rule of probability.

**Theorem 1.5** (Chain Rule)**.** *For any joint distribution $P(x_1, \ldots, x_T)$:*

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t})$$

*where $x_{<t} = (x_1, \ldots, x_{t-1})$ denotes the* context *at step $t$.*

*Proof.* By successive applications of the definition of conditional probability $P(A \mid B) = P(A, B)/P(B)$:

$$\begin{aligned} P(x_1, x_2, \ldots, x_T) &= P(x_1) \cdot P(x_2 \mid x_1) \cdot P(x_3 \mid x_1, x_2) \cdots \\ &= \prod_{t=1}^{T} P(x_t \mid x_{<t}). \end{aligned}$$ □

## 1.5 Information Theory and Language

**Definition 1.6** (Entropy)**.** The *entropy* of a discrete random variable $X$ taking values in $\mathcal{X}$ is:

$$H(X) = -\sum_{x \in \mathcal{X}} P(x) \log_2 P(x)$$

It measures the average uncertainty associated with $X$.

**Definition 1.7** (Cross-Entropy)**.** The *cross-entropy* between the true distribution $p$ and a model $q$ is:

$$H(p, q) = -\sum_{x} p(x) \log q(x) = H(p) + \mathrm{KL}(p \| q)$$

**Proposition 1.8** (Lower Bound)**.** For any model $q$, we have $H(p, q) \geq H(p)$, with equality if and only if $q = p$.

The *perplexity* of a language model $q$ is defined as:

$$\mathrm{PPL}(q) = 2^{H(p,q)}$$

**Key Information-Theoretic Metrics**

$$H(X) = -\sum_x P(x) \log P(x) \qquad \text{(entropy)} \qquad (1.1)$$

$$\mathrm{KL}(p\|q) = \sum_x p(x) \log \frac{p(x)}{q(x)} \qquad \text{(KL divergence)} \qquad (1.2)$$

$$\mathrm{PPL} = \exp\left(-\frac{1}{T}\sum_{t=1}^{T} \log q(x_t \mid x_{<t})\right) \qquad \text{(perplexity)} \qquad (1.3)$$

## 1.6 Classical NLP Pipeline

A classical NLP pipeline comprises the following stages:

1. **Data collection**: text corpora, web scraping, APIs.
2. **Preprocessing**:
   - Tokenization (splitting into lexical units)
   - Normalization (lowercasing, accent removal)
   - Stopword removal
   - Stemming or lemmatization
3. **Representation**: converting text to numerical vectors.
4. **Modeling**: training a model.
5. **Evaluation**: measuring performance on a test set.
6. **Deployment**: putting the model into production.

**Preprocessing and modern models**

With pre-trained models (BERT, GPT), preprocessing is generally minimal: the model's tokenizer (BPE, WordPiece) replaces classical stemming and stopword removal steps. Applying aggressive preprocessing can actually degrade performance.

## 1.7 Tokenization

**Definition 1.9** (Tokenization)**.** *Tokenization* is the process of splitting text into elementary units called *tokens*. A token can be a word, a subword, a character, or a byte.

### 1.7.1 Subword Tokenization

Modern subword tokenization algorithms (BPE, WordPiece, Unigram) offer a trade-off between word tokenization (very large vocabulary) and character tokenization (very long sequences).

**Byte Pair Encoding (BPE)**

```
1. Initialize the vocabulary with all individual characters.
2. Repeat $k$ times:
    (a) Count all adjacent symbol pairs in the corpus.
    (b) Merge the most frequent pair into a new symbol.
    (c) Add this symbol to the vocabulary.
3. Return the final vocabulary of size $|\mathcal{V}_0| + k$.
```

**Tokenization with Hugging Face**

```python
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
text = "Natural language processing is fascinating."
tokens = tokenizer.tokenize(text)
print(tokens)
# ['natural', 'language', 'processing', 'is', 'fascinating', '.']

ids = tokenizer.encode(text, return_tensors="pt")
print(ids)
# tensor([[ 101, 3019, 2653, 6364, 2003, 15281, 1012,  102]])
```

## 1.8 Brief History of NLP

| Period | Paradigm | Examples |
|---|---|---|
| 1950–1970 | Symbolic rules | ELIZA, SHRDLU |
| 1970–1990 | Formal grammars | ATN, LFG, HPSG |
| 1990–2010 | Statistical methods | HMM, CRF, $n$-grams |
| 2010–2017 | Deep learning | Word2Vec, LSTM, Seq2Seq |
| 2017– | Transformers & LLMs | BERT, GPT, T5, LLaMA |

*Remark* 1.10. The transition from the statistical paradigm to the neural paradigm does not represent a complete break: the concepts of cross-entropy, language modeling, and regularization remain at the heart of modern approaches.

## 1.9 Open Challenges

Despite spectacular progress, many challenges remain open:

- **Real understanding vs. statistical correlations**: do LLMs capture *meaning* or merely surface regularities?

- **Low-resource languages**: the majority of resources are available in English; approximately 7,000 languages are spoken worldwide.
- **Robustness**: sensitivity to adversarial perturbations, neologisms, out-of-distribution domains.
- **Efficiency**: LLMs require considerable computational resources.
- **Ethics**: bias, toxicity, privacy, misinformation.

## 1.10 Exercises

**Exercise 1.1** ($\star$)**.** Compute the entropy of a uniform language model over a vocabulary of size $V = 50{,}000$. What is the corresponding perplexity?

**Exercise 1.2** ($\star$)**.** Consider the sentence "The cat eats the mouse." Apply the chain rule to express $P(\text{The, cat, eats, the, mouse})$ as a product of conditional probabilities.

**Exercise 1.3** ($\star\star$)**.** Implement a simple BPE tokenizer in Python. Start with a small corpus of 10 sentences and show the vocabulary evolution after 20 merges.

**Exercise 1.4** ($\star\star$)**.** Show that the Kullback-Leibler divergence $\mathrm{KL}(p\|q)$ is always non-negative (Gibbs' inequality). *Hint:* use the inequality $\log x \leq x - 1$.

**Exercise 1.5** ($\star\star\star$)**.** Compare the perplexity of an $n$-gram model (for $n = 1, 2, 3$) and a neural model (GPT-2) on a corpus of your choice. Analyze the results as a function of training corpus size.

# Chapter 2

# Classical Text Representations

Here is the fundamental problem of natural language processing: computers operate on numbers, but humans communicate with words. Bridging this gap—turning text into mathematical objects that algorithms can manipulate—is the challenge of *text representation*, and the history of NLP can be read as a series of increasingly sophisticated answers to this single question.

The earliest approach, dating back to the 1950s, was brutally simple: represent each document as a vector of word counts, ignoring grammar, word order, and meaning entirely. This "bag of words" model is laughably crude, yet it powered information retrieval systems for decades. The refinement came with TF-IDF weighting, which recognised that not all words are equally informative: "the" appears everywhere and tells you nothing, while "mitochondria" is rare and highly diagnostic. From there, the quest for richer representations led to latent semantic analysis, topic models, and ultimately to the word embeddings of Mikolov's Word2Vec (2013)—the breakthrough that showed meaning could be captured by geometry. But before we reach those heights, we must understand the classical foundations.

**From text to vectors**

Machine learning algorithms operate on numerical vectors, not raw text. This chapter presents classical methods for transforming textual documents into vector representations that can be used by classifiers or search engines.

## 2.1 Bag-of-Words Representation

**Definition 2.1** (Bag of Words)**.** The *bag-of-words* (BoW) model represents a document $d$ as a vector $\mathbf{x}_d \in \mathbb{N}^V$ where the $j$-th component is the count of term $t_j$ in $d$:

$$x_{d,j} = \text{count}(t_j, d)$$

This model completely ignores word order. The sentences "the cat eats the mouse" and "the mouse eats the cat" have the same representation.

*Remark* 2.2. Despite its simplicity, the bag-of-words model remains useful as a baseline and in applications where word order is less important (topic classification, document filtering).

## 2.2 TF-IDF Weighting

Raw frequency does not reflect the discriminative importance of a term. Very frequent words (“the”, “of”, “and”) dominate the representation without providing semantic information.

**Definition 2.3** (TF – Term Frequency)**.** The *term frequency* of $t$ in document $d$ is:

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

**Definition 2.4** (IDF – Inverse Document Frequency)**.** The *inverse document frequency* of term $t$ in a corpus $\mathcal{D}$ of $N$ documents is:

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

**Definition 2.5** (TF-IDF)**.** The *TF-IDF* weighting combines both measures:

$$\text{tf-idf}(t,d,\mathcal{D}) = \text{tf}(t,d) \times \text{idf}(t,\mathcal{D})$$

**Theorem 2.6** (Properties of TF-IDF)**.** *Let $\mathbf{v}_d \in \mathbb{R}^V$ be the TF-IDF vector of document $d$. Then:*

1. *tf-idf$(t,d) = 0$ if $t \notin d$;*
2. *tf-idf$(t,d)$ is high if $t$ is frequent in $d$ but rare in the corpus;*
3. *idf$(t) = 0$ if $t$ appears in all documents.*

**TF-IDF Variants**

$$\text{tf}_{\text{log}}(t,d) = 1 + \log(\text{count}(t,d)) \qquad \text{(log tf)} \qquad (2.1)$$

$$\text{idf}_{\text{smooth}}(t) = \log \frac{N+1}{\text{df}(t)+1} + 1 \qquad \text{(smoothed idf)} \qquad (2.2)$$

$$\text{tf-idf}_{\text{norm}} = \frac{\text{tf-idf}(t,d)}{\|\mathbf{v}_d\|} \qquad (L_2 \text{ normalized}) \qquad (2.3)$$

**TF-IDF with scikit-learn**

```python
from sklearn.feature_extraction.text import TfidfVectorizer

corpus = [
    "The cat sleeps on the mat",
    "The dog runs in the garden",
    "The cat and the dog play together"
]

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)
print(f"Matrix shape: {X.shape}")
print(f"Vocabulary: {vectorizer.get_feature_names_out()}")
print(f"TF-IDF matrix (dense):\n{X.toarray().round(2)}")
```

## 2.3 $N$-gram Models

**Definition 2.7** ($N$-gram)**.** An *$n$-gram* is a contiguous subsequence of $n$ tokens extracted from a text. For a sequence $(x_1, \ldots, x_T)$, the $n$-grams are:

$$\{(x_t, x_{t+1}, \ldots, x_{t+n-1}) : t = 1, \ldots, T - n + 1\}$$

### 2.3.1 $N$-gram Language Model

An $n$-gram language model approximates the conditional probability using a Markov assumption of order $n - 1$:

**Definition 2.8** ($N$-gram Language Model)**.**

$$P(x_t \mid x_1, \ldots, x_{t-1}) \approx P(x_t \mid x_{t-n+1}, \ldots, x_{t-1})$$

estimated by maximum likelihood:

$$\hat{P}(x_t \mid x_{t-n+1}^{t-1}) = \frac{\text{count}(x_{t-n+1}, \ldots, x_t)}{\text{count}(x_{t-n+1}, \ldots, x_{t-1})}$$

**Zero probability problem**

If an $n$-gram never appears in the training corpus, its estimated probability is zero, which makes the perplexity infinite. Smoothing techniques (Laplace, Kneser-Ney) are essential.

### 2.3.2 Laplace Smoothing (Add-$\alpha$)

**Definition 2.9** (Laplace Smoothing)**.** Laplace smoothing adds a pseudo-count $\alpha > 0$:

$$\hat{P}_\alpha(x_t \mid x_{t-n+1}^{t-1}) = \frac{\text{count}(x_{t-n+1}^{t}) + \alpha}{\text{count}(x_{t-n+1}^{t-1}) + \alpha V}$$

where $V = |\mathcal{V}|$.

### 2.3.3 Kneser-Ney Smoothing

**Definition 2.10** (Kneser-Ney Smoothing)**.** Kneser-Ney smoothing uses an absolute discount $\delta$ and a continuation distribution:

$$P_{\text{KN}}(w \mid h) = \frac{\max(\text{count}(h, w) - \delta, 0)}{\text{count}(h)} + \lambda(h) \cdot P_{\text{cont}}(w)$$

where $P_{\text{cont}}(w) \propto |\{h' : \text{count}(h', w) > 0\}|$ is the continuation probability and $\lambda(h)$ is a normalization factor.

## 2.4 Document Similarity

**Definition 2.11** (Cosine Similarity)**.** The *cosine similarity* between two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^V$ is:

$$\cos(\mathbf{u}, \mathbf{v}) = \frac{\langle \mathbf{u}, \mathbf{v} \rangle}{\|\mathbf{u}\| \cdot \|\mathbf{v}\|}$$

**Proposition 2.12** (Properties of Cosine Similarity)**.** 1. $\cos(\mathbf{u}, \mathbf{v}) \in [-1, 1]$

2. $\cos(\mathbf{u}, \mathbf{v}) = 1 \iff \mathbf{u} = \lambda \mathbf{v}$ for $\lambda > 0$

3. For TF-IDF vectors (non-negative components): $\cos(\mathbf{u}, \mathbf{v}) \in [0, 1]$

**Definition 2.13** (Jaccard Distance)**.** For two term sets $A$ and $B$:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}, \qquad d_J(A, B) = 1 - J(A, B)$$

## 2.5 Document-Term Matrix and SVD

The document-term matrix $\mathbf{M} \in \mathbb{R}^{N \times V}$ is generally very sparse ($>$ 99% zeros). The singular value decomposition (SVD) enables dimensionality reduction.

**Definition 2.14** (Latent Semantic Analysis (LSA))**.** Latent Semantic Analysis (LSA) applies truncated SVD to the TF-IDF matrix:

$$\mathbf{M} \approx \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^\top$$

where $k \ll \min(N, V)$. Documents are then represented in $\mathbb{R}^k$ by the rows of $\mathbf{U}_k \mathbf{\Sigma}_k$.

**Theorem 2.15** (Optimality of Truncated SVD (Eckart-Young))**.** *The rank-$k$ matrix that minimizes the Frobenius error $\left\|\mathbf{M} - \hat{\mathbf{M}}\right\|_F$ is given by the truncated SVD $\hat{\mathbf{M}} = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^\top$.*

**LSA with scikit-learn**

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD

vectorizer = TfidfVectorizer(max_features=10000)
X_tfidf = vectorizer.fit_transform(corpus)

svd = TruncatedSVD(n_components=100, random_state=42)
X_lsa = svd.fit_transform(X_tfidf)
print(f"Explained variance: {svd.explained_variance_ratio_.sum():.2%}")
print(f"Reduced dimension: {X_lsa.shape}")
```

## 2.6 Character $N$-gram Representations

Character $n$-grams capture morphology and are robust to spelling errors:

**Example 2.16.** The word "hello" with $n = 3$ produces the trigrams: `#he`, `hel`, `ell`, `llo`, `lo#` (where `#` marks word boundaries).

## 2.7 BM25 Model

**Definition 2.17** (BM25)**.** The BM25 score of document $d$ for query $q = \{q_1, \ldots, q_m\}$ is:

$$\text{BM25}(d, q) = \sum_{i=1}^{m} \text{idf}(q_i) \cdot \frac{\text{tf}(q_i, d) \cdot (k_1 + 1)}{\text{tf}(q_i, d) + k_1 \cdot \left(1 - b + b \cdot \frac{|d|}{\text{avgdl}}\right)}$$

where $k_1$ and $b$ are hyperparameters (typically $k_1 = 1.2$, $b = 0.75$) and avgdl is the average document length.

**BM25 in practice**

BM25 remains a reference method for information retrieval. Even modern RAG systems often use BM25 as a first retrieval stage before neural reranking.

## 2.8 Limitations of Sparse Representations

The methods presented in this chapter suffer from several limitations:

1. **High dimensionality**: $V$ can reach $10^5$ to $10^6$.
2. **Sparsity**: vectors are very sparse, making distance measures unreliable.
3. **Lack of semantics**: synonymous words have orthogonal components (“car” $\perp$ “automobile”).
4. **Context independence**: each word has a fixed representation independent of context.

These limitations motivate dense embeddings (Chapter 3), which represent words in a low-dimensional space where geometric proximity reflects semantic similarity.

## 2.9 Application: Naive Bayes Classification

**Definition 2.18** (Multinomial Naive Bayes Classifier)**.** Let a document be represented by its word counts $\mathbf{x} = (x_1, \ldots, x_V)$. The multinomial naive Bayes classifier predicts:

$$\hat{y} = \arg\max_{c \in \mathcal{Y}} \left[ \log P(c) + \sum_{j=1}^{V} x_j \log P(t_j \mid c) \right]$$

with $P(t_j \mid c) = \frac{\text{count}(t_j, c) + \alpha}{\sum_{j'} \text{count}(t_{j'}, c) + \alpha V}$.

**Naive Bayes classification**

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline
from sklearn.datasets import fetch_20newsgroups
```

```
data = fetch_20newsgroups(
    subset='train',
    categories=['sci.space', 'rec.sport.baseball']
)
pipeline = Pipeline([
    ('tfidf', TfidfVectorizer(max_features=10000)),
    ('clf', MultinomialNB(alpha=0.1))
])
pipeline.fit(data.data, data.target)
print(f"Accuracy (train): {pipeline.score(data.data, data.target):.2%}")
```

## 2.10 Exercises

**Exercise 2.1** ($\star$)**.** Compute the TF-IDF vectors for the following corpus: $d_1$ = "the cat sleeps", $d_2$ = "the dog sleeps", $d_3$ = "the cat and the dog".

**Exercise 2.2** ($\star$)**.** Show that cosine similarity is invariant under positive scaling: $\cos(\lambda\mathbf{u}, \mu\mathbf{v}) = \cos(\mathbf{u}, \mathbf{v})$ for $\lambda, \mu > 0$.

**Exercise 2.3** ($\star\star$)**.** Implement a bigram language model with Laplace smoothing. Compute the perplexity on a test corpus.

**Exercise 2.4** ($\star\star$)**.** Show that the Eckart-Young theorem implies that the truncated SVD gives the best rank-$k$ approximation in the Frobenius norm.

**Exercise 2.5** ($\star\star\star$)**.** Compare the performance of TF-IDF + logistic regression, TF-IDF + SVM, and LSA + logistic regression on sentiment classification (IMDB dataset). Analyze the impact of the LSA dimension $k$.

# Chapter 3

# Word Embeddings

In 2013, Tomas Mikolov and his team at Google published a paper that would transform natural language processing: *Efficient Estimation of Word Representations in Vector Space*. The idea was to train a shallow neural network to predict words from their context (or vice versa) and use the learned weights as vector representations of words. The result, Word2Vec, revealed a stunning geometric structure: semantic analogies become *vector operations*. The vector "king" − "man" + "woman" points to "queen." Geometry captures meaning.

But Word2Vec did not emerge from nothing. It belongs to a long tradition founded on the *distributional hypothesis* of J.R. Firth (1957): "you shall know a word by the company it keeps." This chapter traces the path from co-occurrence matrices to dense embeddings, through latent semantic analysis and GloVe.

**The distributional hypothesis**

"You shall know a word by the company it keeps" (J.R. Firth, 1957). Word embeddings are dense vector representations that capture semantic similarity: words appearing in similar contexts have nearby vectors.

## 3.1 From Sparse to Dense Representations

Recall that a one-hot representation encodes each word $w_i$ as a vector $\mathbf{e}_i \in \mathbb{R}^V$ with $(\mathbf{e}_i)_j = \delta_{ij}$. This representation has two major drawbacks:

1. **Dimensionality**: $V$ can reach $10^5$ to $10^6$.
2. **Orthogonality**: $\langle \mathbf{e}_i, \mathbf{e}_j \rangle = 0$ for $i \neq j$, even for synonyms.

The goal of embeddings is to learn a function $\phi : \mathcal{V} \to \mathbb{R}^d$ with $d \ll V$ (typically $d \in [100, 300]$) such that geometric proximity reflects semantic similarity.

## 3.2 Word2Vec

### 3.2.1 CBOW Architecture

**Definition 3.1** (Continuous Bag of Words (CBOW))**.** The CBOW model predicts the center word $w_t$ from its context $C_t = \{w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}\}$ where $c$ is the window size. The objective is to maximize:

$$\mathcal{L}_{\text{CBOW}} = \sum_{t=1}^{T} \log P(w_t \mid C_t)$$

with:

$$P(w_t \mid C_t) = \frac{\exp(\langle \mathbf{v}_{w_t}, \bar{\mathbf{u}}_{C_t} \rangle)}{\sum_{w \in \mathcal{V}} \exp(\langle \mathbf{v}_w, \bar{\mathbf{u}}_{C_t} \rangle)}$$

where $\bar{\mathbf{u}}_{C_t} = \frac{1}{2c} \sum_{w \in C_t} \mathbf{u}_w$ is the average of context vectors.

### 3.2.2 Skip-gram Architecture

**Definition 3.2** (Skip-gram)**.** The Skip-gram model predicts context words from the center word. The objective is to maximize:

$$\mathcal{L}_{\text{SG}} = \sum_{t=1}^{T} \sum_{\substack{j=-c \\ j \neq 0}}^{c} \log P(w_{t+j} \mid w_t)$$

with:

$$P(w_o \mid w_i) = \frac{\exp(\langle \mathbf{v}_{w_o}, \mathbf{u}_{w_i} \rangle)}{\sum_{w \in \mathcal{V}} \exp(\langle \mathbf{v}_w, \mathbf{u}_{w_i} \rangle)}$$

**Softmax cost**

The softmax denominator requires a sum over the entire vocabulary $V$, which is prohibitive. Two common approximations are used: negative sampling and hierarchical softmax.

### 3.2.3 Negative Sampling (NEG)

**Definition 3.3** (Negative Sampling)**.** The negative sampling objective replaces the softmax with:

$$\mathcal{L}_{\text{NEG}} = \log \sigma(\langle \mathbf{v}_{w_o}, \mathbf{u}_{w_i} \rangle) + \sum_{k=1}^{K} \mathbb{E}_{w_k \sim P_n} \left[ \log \sigma(-\langle \mathbf{v}_{w_k}, \mathbf{u}_{w_i} \rangle) \right]$$

where $K$ is the number of negative examples and $P_n(w) \propto \text{count}(w)^{3/4}$.

**Theorem 3.4** (NEG-PMI Equivalence)**.** *Levy and Goldberg (2014) showed that Skip-gram with negative sampling implicitly factorizes a shifted PMI matrix:*

$$\mathbf{u}_w^\top \mathbf{v}_c = PMI(w, c) - \log K$$

*where $PMI(w, c) = \log \frac{P(w,c)}{P(w)P(c)}$ is the pointwise mutual information.*

## 3.3 GloVe

**Definition 3.5** (GloVe)**.** The GloVe model (Pennington et al., 2014) learns embeddings by minimizing the weighted squared error on the co-occurrence matrix:

$$\mathcal{L}_{\text{GloVe}} = \sum_{i,j=1}^{V} f(X_{ij}) \left(\langle \mathbf{u}_i, \mathbf{v}_j \rangle + b_i + c_j - \log X_{ij}\right)^2$$

where $X_{ij}$ is the co-occurrence count, $b_i, c_j$ are biases, and $f$ is a weighting function:

$$f(x) = \begin{cases} (x/x_{\max})^{\alpha} & \text{if } x < x_{\max} \\ 1 & \text{otherwise} \end{cases}$$

with typically $x_{\max} = 100$ and $\alpha = 0.75$.

*Remark* 3.6. GloVe combines the advantages of global methods (matrix factorization) and local methods (context window). The GloVe objective is equivalent to a factorization of the log-co-occurrence matrix.

## 3.4 FastText

**Definition 3.7** (FastText)**.** FastText (Bojanowski et al., 2017) enriches Skip-gram by representing each word as the sum of its character $n$-grams. For a word $w$ with $n$-gram set $\mathcal{G}_w$:

$$\mathbf{u}_w = \sum_{g \in \mathcal{G}_w} \mathbf{z}_g$$

where $\mathbf{z}_g \in \mathbb{R}^d$ is the vector of $n$-gram $g$.

**FastText advantages**

- Handles out-of-vocabulary (OOV) words via $n$-gram decomposition.
- Captures morphology (prefixes, suffixes).
- Particularly effective for morphologically rich languages (Turkish, Finnish, Arabic).

## 3.5 Geometric Properties

### 3.5.1 Vector Analogies

One of the most remarkable properties of embeddings is their ability to capture semantic relations through arithmetic operations:

**Example 3.8.**

$$\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{queen}}$$

Formally, to solve the analogy $a : b :: c :?$, we seek:

$$d^* = \operatorname*{arg\,max}_{d \in \mathcal{V} \setminus \{a,b,c\}} \cos(\mathbf{v}_d, \mathbf{v}_b - \mathbf{v}_a + \mathbf{v}_c)$$

**Theorem 3.9** (Linear Structure of Analogies)**.** *Under the assumption that embeddings factorize a PMI matrix (Theorem 3.4), regular semantic relations manifest as linear directions in embedding space:*

$$\mathbf{v}_b - \mathbf{v}_a \approx \mathbf{v}_{b'} - \mathbf{v}_{a'}$$

*for all pairs* $(a, b)$ *and* $(a', b')$ *sharing the same relation.*

### 3.5.2 t-SNE Visualization

**Embedding visualization**

```python
import numpy as np
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
from gensim.models import KeyedVectors

# Load pre-trained embeddings
wv = KeyedVectors.load_word2vec_format('GoogleNews-vectors.bin',
↪  binary=True)
words = ['king', 'queen', 'man', 'woman', 'prince', 'princess',
         'paris', 'france', 'berlin', 'germany', 'rome', 'italy']
vectors = np.array([wv[w] for w in words])

tsne = TSNE(n_components=2, random_state=42, perplexity=5)
coords = tsne.fit_transform(vectors)

plt.figure(figsize=(10, 8))
for i, word in enumerate(words):
    plt.scatter(coords[i, 0], coords[i, 1])
    plt.annotate(word, (coords[i, 0]+0.5, coords[i, 1]+0.5))
plt.title("t-SNE Visualization of Word2Vec Embeddings")
plt.show()
```

## 3.6 Embedding Evaluation

### 3.6.1 Intrinsic Evaluation

- **Analogies**: accuracy on analogy benchmarks (Google analogy dataset, BATS).
- **Similarity**: Spearman correlation between cosine similarity and human judgments (SimLex-999, WordSim-353).

### 3.6.2 Extrinsic Evaluation

Performance of embeddings as features in downstream tasks: sentiment classification, NER, word sense disambiguation.

**Embedding evaluation metrics**

$$\text{Analogy accuracy} = \frac{\text{correct analogies}}{\text{total analogies}} \tag{3.1}$$

$$\rho_{\text{Spearman}} = 1 - \frac{6\sum_i d_i^2}{n(n^2-1)} \qquad \text{(rank correlation)} \tag{3.2}$$

## 3.7 Bias in Embeddings

Embeddings learned from large text corpora inherit societal biases present in the data.

**Example 3.10.** Bolukbasi et al. (2016) showed that:

$$\mathbf{v}_{\text{computer programmer}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{homemaker}}$$

This result reflects gender stereotypes present in the training corpora.

**Definition 3.11** (Debiasing by Projection)**.** The debiasing method of Bolukbasi et al. consists of:

1. Identifying the bias direction $\mathbf{g}$ (via PCA on gender pairs).
2. Projecting neutral words orthogonally to $\mathbf{g}$: $\mathbf{v}'_w = \mathbf{v}_w - \langle \mathbf{v}_w, \mathbf{g} \rangle \mathbf{g}$.
3. Equalizing definitional pairs.

## 3.8 Contextual vs. Static Embeddings

Static embeddings (Word2Vec, GloVe, FastText) assign a single vector to each word, regardless of context. This is problematic for polysemous words:

**Example 3.12.** The word "bank" can refer to a financial institution or the side of a river. A static embedding represents both senses with a single vector, which is a compromise between the two.

Contextual embeddings (ELMo, BERT, GPT) solve this problem by producing different representations depending on context. They will be studied in Chapter 6.

## 3.9 Complete Implementation

**Skip-gram training with PyTorch**

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

class SkipGram(nn.Module):
    def __init__(self, vocab_size, embed_dim):
        super().__init__()
        self.target_embed = nn.Embedding(vocab_size, embed_dim)
        self.context_embed = nn.Embedding(vocab_size, embed_dim)

    def forward(self, target, context, negatives):
        # target: (B,), context: (B,), negatives: (B, K)
        t = self.target_embed(target)          # (B, d)
        c = self.context_embed(context)        # (B, d)
        n = self.context_embed(negatives)      # (B, K, d)

        # Positive score
        pos_score = torch.sum(t * c, dim=1)    # (B,)
        pos_loss = -torch.log(torch.sigmoid(pos_score) + 1e-10)

        # Negative scores
        neg_score = torch.bmm(n, t.unsqueeze(2)).squeeze(2)  # (B, K)
        neg_loss = -torch.log(torch.sigmoid(-neg_score) + 1e-10).sum(1)

        return (pos_loss + neg_loss).mean()

model = SkipGram(vocab_size=50000, embed_dim=300)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

## 3.10 Exercises

**Exercise 3.1** (⋆)**.** Show that the cosine similarity between two one-hot vectors is zero for different words.

**Exercise 3.2** (⋆⋆)**.** Derive the gradient of the Skip-gram objective with negative sampling with respect to vectors $\mathbf{u}_{w_i}$ and $\mathbf{v}_{w_o}$.

**Exercise 3.3** (⋆⋆)**.** Show that the GloVe objective is equivalent to a weighted matrix factorization of the matrix $\log X_{ij}$.

**Exercise 3.4** (⋆⋆)**.** Train Word2Vec embeddings (using `gensim`) on an English corpus (Wikipedia). Evaluate the analogies "king – man + woman = ?" and "Paris – France + Germany = ?".

**Exercise 3.5** (⋆⋆⋆)**.** Implement the debiasing method of Bolukbasi et al. (2016). Measure gender bias before and after correction on a set of profession-related words.

# Chapter 4

# Sequence Models

The bag of words throws order out the window: "the cat eats the mouse" and "the mouse eats the cat" receive the same representation. To capture meaning, one must respect the sequence. Recurrent neural networks (RNNs), introduced by Jeffrey Elman in 1990, offer an elegant solution: a hidden state that propagates from one word to the next, accumulating contextual information. But simple RNNs suffer from the *vanishing gradient* problem. The solution came from LSTM cells (Hochreiter and Schmidhuber, 1997) and GRU cells (Cho et al., 2014), which introduce gating mechanisms to control the flow of information.

**Modeling word order**

Unlike the bag-of-words model, sequence models treat text as an ordered sequence of tokens. Recurrent neural networks (RNNs) maintain a hidden state that encodes the sequence history, allowing them to capture temporal dependencies.

## 4.1 Recurrent Neural Networks (RNN)

**Definition 4.1** (RNN – Elman Network)**.** A simple recurrent network (Elman, 1990) processes a sequence $(\mathbf{x}_1, \dots, \mathbf{x}_T)$ by maintaining a hidden state $\mathbf{h}_t \in \mathbb{R}^h$:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h) \tag{4.1}$$
$$\mathbf{y}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y \tag{4.2}$$

where $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$, $\mathbf{W}_{xh} \in \mathbb{R}^{h\times d}$, $\mathbf{W}_{hy} \in \mathbb{R}^{o\times h}$ are weight matrices.

*Remark* 4.2. The hidden state $\mathbf{h}_t$ acts as a "compressed memory" of the entire history $(\mathbf{x}_1, \dots, \mathbf{x}_t)$. In theory, an RNN can model dependencies of arbitrary length; in practice, this ability is severely limited.

## 4.2 Vanishing and Exploding Gradient Problem

**Theorem 4.3** (Vanishing Gradient)**.** *During backpropagation through time (BPTT), the gradient of the loss $\mathcal{L}$ with respect to $\mathbf{h}_k$ $(k \ll t)$ satisfies:*

$$\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_k} = \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}}$$

*where each Jacobian* $\frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} = diag(\tanh'(\cdot))\mathbf{W}_{hh}$*.*

*If* $\|\mathbf{W}_{hh}\| < 1/\gamma$ *(where* $\gamma = \max|\tanh'(\cdot)|$*), the product of Jacobians decays exponentially, making it impossible to learn long-range dependencies.*

*Proof.* We have $\left\|\frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}}\right\| \leq \gamma \|\mathbf{W}_{hh}\|$ where $\gamma \leq 1$ for tanh. Therefore:

$$\left\| \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \right\| \leq (\gamma \|\mathbf{W}_{hh}\|)^{t-k}$$

If $\gamma \|\mathbf{W}_{hh}\| < 1$, this quantity tends to 0 exponentially fast as $t - k \to \infty$. □

**Gradient clipping**

To counter the exploding gradient, *gradient clipping* is used:

$$\hat{\mathbf{g}} = \begin{cases} \mathbf{g} & \text{if } \|\mathbf{g}\| \leq \theta \\ \theta \cdot \frac{\mathbf{g}}{\|\mathbf{g}\|} & \text{otherwise} \end{cases}$$

where $\theta$ is the clipping threshold (typically $\theta \in [1, 5]$).

## 4.3 Long Short-Term Memory (LSTM)

**Definition 4.4** (LSTM)**.** The LSTM network (Hochreiter & Schmidhuber, 1997) introduces a memory cell $\mathbf{c}_t$ and three gates to control the flow of information:

$$\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) && \text{(forget gate)} && (4.3)\\
\mathbf{i}_t &= \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) && \text{(input gate)} && (4.4)\\
\tilde{\mathbf{c}}_t &= \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c) && \text{(candidate)} && (4.5)\\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t && \text{(cell update)} && (4.6)\\
\mathbf{o}_t &= \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) && \text{(output gate)} && (4.7)\\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t) && \text{(hidden state)} && (4.8)
\end{aligned}$$

where $\odot$ denotes the Hadamard (element-wise) product.

**Proposition 4.5** (Gradient in LSTMs)**.** In an LSTM, the gradient of the loss with respect to the memory cell $\mathbf{c}_k$ contains a term:

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_k} = \prod_{i=k+1}^{t} (\text{diag}(\mathbf{f}_i) + \ldots)$$

The forget gate $\mathbf{f}_i$ allows the gradient to propagate without attenuation when $\mathbf{f}_i \approx \mathbf{1}$, partially solving the vanishing gradient problem.

## 4.4 Gated Recurrent Unit (GRU)

**Definition 4.6** (GRU)**.** The GRU (Cho et al., 2014) simplifies the LSTM by merging the memory cell and hidden state, using only two gates:

$$\mathbf{z}_t = \sigma(\mathbf{W}_z[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_z) \qquad \text{(update gate)} \tag{4.9}$$

$$\mathbf{r}_t = \sigma(\mathbf{W}_r[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_r) \qquad \text{(reset gate)} \tag{4.10}$$

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_h[\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_h) \qquad \text{(candidate)} \tag{4.11}$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t \qquad \text{(interpolation)} \tag{4.12}$$

*Remark* 4.7. The GRU has fewer parameters than the LSTM (3 gate matrices vs. 4) and offers comparable performance in many tasks. The choice between LSTM and GRU is often empirical.

## 4.5 Bidirectional RNNs

**Definition 4.8** (BiRNN)**.** A bidirectional RNN processes the sequence in both directions and concatenates the hidden states:

$$\mathbf{h}_t = [\overrightarrow{\mathbf{h}}_t; \overleftarrow{\mathbf{h}}_t] \in \mathbb{R}^{2h}$$

where $\overrightarrow{\mathbf{h}}_t$ is the left-to-right state and $\overleftarrow{\mathbf{h}}_t$ is the right-to-left state.

## 4.6 Encoder-Decoder Architecture (Seq2Seq)

**Definition 4.9** (Seq2Seq)**.** The encoder-decoder architecture (Sutskever et al., 2014) uses an encoder RNN to compress the source sequence into a context vector $\mathbf{c} = \mathbf{h}_T^{\text{enc}}$, and a decoder RNN to generate the target sequence:

$$\text{Encoder:} \quad \mathbf{h}_t^{\text{enc}} = f_{\text{enc}}(\mathbf{x}_t, \mathbf{h}_{t-1}^{\text{enc}}) \tag{4.13}$$

$$\text{Decoder:} \quad \mathbf{h}_t^{\text{dec}} = f_{\text{dec}}(\mathbf{y}_{t-1}, \mathbf{h}_{t-1}^{\text{dec}}) \tag{4.14}$$

$$P(y_t \mid y_{<t}, \mathbf{x}) = \text{softmax}(\mathbf{W}_o \mathbf{h}_t^{\text{dec}}) \tag{4.15}$$

**Information bottleneck**

Compressing the entire source sequence into a single vector **c** creates an information bottleneck. This problem is solved by the attention mechanism (Chapter 5).

## 4.7 Implementation with PyTorch

**LSTM for text classification**

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

class LSTMClassifier(nn.Module):
```

*CHAPTER 4. SEQUENCE MODELS*

```
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_classes,
                 num_layers=2, dropout=0.3, bidirectional=True):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim,
        ↪   padding_idx=0)
        self.lstm = nn.LSTM(
            embed_dim, hidden_dim, num_layers=num_layers,
            batch_first=True, dropout=dropout,
            bidirectional=bidirectional
        )
        direction_factor = 2 if bidirectional else 1
        self.classifier = nn.Sequential(
            nn.Dropout(dropout),
            nn.Linear(hidden_dim * direction_factor, num_classes)
        )

    def forward(self, input_ids, lengths):
        embeds = self.embedding(input_ids)       # (B, T, d)
        packed = nn.utils.rnn.pack_padded_sequence(
            embeds, lengths.cpu(), batch_first=True, enforce_sorted=False
        )
        _, (hidden, _) = self.lstm(packed)       # hidden: (layers*dirs,
        ↪   B, h)
        # Concatenate last forward and backward hidden states
        hidden = torch.cat([hidden[-2], hidden[-1]], dim=1)  # (B, 2h)
        return self.classifier(hidden)           # (B, num_classes)

model = LSTMClassifier(
    vocab_size=30000, embed_dim=256,
    hidden_dim=128, num_classes=2
)
```

**Seq2Seq encoder-decoder**

```
class Seq2SeqEncoder(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers=2):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.rnn = nn.GRU(embed_dim, hidden_dim, num_layers,
                          batch_first=True, bidirectional=True)
        self.fc = nn.Linear(hidden_dim * 2, hidden_dim)

    def forward(self, src):
        embedded = self.embedding(src)
        outputs, hidden = self.rnn(embedded)
        # Combine bidirectional hidden states
        hidden = torch.cat([hidden[-2], hidden[-1]], dim=1)
        hidden = torch.tanh(self.fc(hidden)).unsqueeze(0)
        return outputs, hidden
```

```python
class Seq2SeqDecoder(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers=1):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.rnn = nn.GRU(embed_dim, hidden_dim, num_layers,
                          batch_first=True)
        self.fc_out = nn.Linear(hidden_dim, vocab_size)

    def forward(self, input_token, hidden):
        embedded = self.embedding(input_token).unsqueeze(1)
        output, hidden = self.rnn(embedded, hidden)
        prediction = self.fc_out(output.squeeze(1))
        return prediction, hidden
```

## 4.8 Teacher Forcing and Scheduled Sampling

**Definition 4.10** (Teacher Forcing)**.** During training, *teacher forcing* consists of providing the decoder with the ground-truth token $y_{t-1}$ rather than its own prediction $\hat{y}_{t-1}$. This accelerates convergence but creates a mismatch (*exposure bias*) between training and inference.

**Definition 4.11** (Scheduled Sampling)**.** *Scheduled sampling* (Bengio et al., 2015) mitigates exposure bias by randomly alternating between teacher forcing and autoregressive generation:

$$\text{input}_t = \begin{cases} y_{t-1} & \text{with probability } \epsilon_t \\ \hat{y}_{t-1} & \text{with probability } 1-\epsilon_t \end{cases}$$

where $\epsilon_t$ gradually decreases from 1 to 0.

## 4.9 Exercises

**Exercise 4.1** ($\star$)**.** Count the total number of parameters in an LSTM with $d = 256$ (input) and $h = 512$ (hidden state).

**Exercise 4.2** ($\star$)**.** Explain why a bidirectional RNN cannot be used for autoregressive language modeling.

**Exercise 4.3** ($\star\star$)**.** Prove that the gradient in a simple RNN decays exponentially with temporal distance under the conditions of the vanishing gradient theorem.

**Exercise 4.4** ($\star\star$)**.** Implement a character-level language model with a 2-layer LSTM. Train it on a literary corpus and generate text by sampling.

**Exercise 4.5** ($\star\star\star$)**.** Compare the performance of a GRU and an LSTM on a NER task (CoNLL-2003 dataset). Analyze convergence speed and ability to capture long-range dependencies.

28

# Chapter 5

# Attention and Transformers

"Attention Is All You Need." This title, chosen by Vaswani et al. for their 2017 paper, has become one of the most famous in the history of artificial intelligence. The *Transformer*, the architecture they proposed, entirely abandons recurrence in favour of an *attention* mechanism that allows each word in the sequence to "look" directly at every other word, without distance restriction. The result: massive parallelism during training, the ability to capture long-range dependencies, and performance that quickly surpassed all existing models. The Transformer is the building block of GPT, BERT, and the entire large language model revolution.

**Attention: looking where it matters**

The attention mechanism allows the model to focus on relevant parts of the input at each generation step, eliminating the encoder-decoder bottleneck. The Transformer architecture, based entirely on attention, has revolutionized NLP since 2017.

## 5.1 Attention in Seq2Seq Models

**Definition 5.1** (Bahdanau Attention)**.** Additive attention (Bahdanau et al., 2015) computes a context vector $\mathbf{c}_t$ as a weighted average of encoder states:

$$e_{t,s} = \mathbf{v}_a^\top \tanh(\mathbf{W}_a \mathbf{h}_s^{\text{enc}} + \mathbf{U}_a \mathbf{h}_{t-1}^{\text{dec}}) \tag{5.1}$$

$$\alpha_{t,s} = \frac{\exp(e_{t,s})}{\sum_{s'=1}^{S} \exp(e_{t,s'})} \tag{5.2}$$

$$\mathbf{c}_t = \sum_{s=1}^{S} \alpha_{t,s} \mathbf{h}_s^{\text{enc}} \tag{5.3}$$

where $\alpha_{t,s}$ is the attention weight on source position $s$ at decoding step $t$.

**Definition 5.2** (Luong Attention)**.** Multiplicative attention (Luong et al., 2015) uses a simpler score:

$$e_{t,s} = (\mathbf{h}_t^{\text{dec}})^\top \mathbf{W}_a \mathbf{h}_s^{\text{enc}}$$

Variants: *dot* ($e_{t,s} = (\mathbf{h}_t^{\text{dec}})^\top \mathbf{h}_s^{\text{enc}}$), *general*, *concat*.

*CHAPTER 5. ATTENTION AND TRANSFORMERS*

## 5.2 Self-Attention

**Definition 5.3** (Scaled Dot-Product Attention)**.** Self-attention (*scaled dot-product attention*) operates on query $\mathbf{Q}$, key $\mathbf{K}$, and value $\mathbf{V}$ vectors:

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

where $d_k$ is the key dimension and the factor $1/\sqrt{d_k}$ stabilizes gradients.

**Theorem 5.4** (Justification of the Scaling Factor)**.** *Let* $\mathbf{q}, \mathbf{k} \in \mathbb{R}^{d_k}$ *be random vectors with i.i.d. components of mean* $0$ *and variance* $1$*. Then:*

$$\mathbb{E}[\langle \mathbf{q}, \mathbf{k} \rangle] = 0, \qquad \mathit{Var}[\langle \mathbf{q}, \mathbf{k} \rangle] = d_k$$

*Without the factor* $1/\sqrt{d_k}$*, the variance of the dot product grows with* $d_k$*, pushing the softmax into saturated regions where gradients are near zero.*

*Proof.* $\langle \mathbf{q}, \mathbf{k} \rangle = \sum_{i=1}^{d_k} q_i k_i$. Since components are independent with zero mean: $\mathbb{E}[q_i k_i] = \mathbb{E}[q_i]\mathbb{E}[k_i] = 0$ and $\text{Var}[q_i k_i] = \mathbb{E}[q_i^2]\mathbb{E}[k_i^2] = 1$. By independence: $\text{Var}[\langle \mathbf{q}, \mathbf{k} \rangle] = d_k$. □

## 5.3 Multi-Head Attention

**Definition 5.5** (Multi-Head Attention)**.** Multi-head attention projects $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$ into $H$ different subspaces and concatenates the results:

$$\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \tag{5.4}$$

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = [\text{head}_1; \dots; \text{head}_H]\mathbf{W}^O \tag{5.5}$$

where $\mathbf{W}_i^Q \in \mathbb{R}^{d \times d_k}$, $\mathbf{W}_i^K \in \mathbb{R}^{d \times d_k}$, $\mathbf{W}_i^V \in \mathbb{R}^{d \times d_v}$, $\mathbf{W}^O \in \mathbb{R}^{Hd_v \times d}$, with $d_k = d_v = d/H$.

*Remark* 5.6. Each attention head can learn to focus on different types of relationships: syntactic, semantic, positional, etc.

## 5.4 Transformer Architecture

**Definition 5.7** (Transformer Block (Encoder))**.** A Transformer encoder block consists of:

1. Multi-head self-attention with residual connection and layer normalization:

$$\mathbf{Z} = \text{LayerNorm}(\mathbf{X} + \text{MultiHead}(\mathbf{X}, \mathbf{X}, \mathbf{X}))$$

2. Position-wise feed-forward network with residual connection:

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

$$\mathbf{H} = \text{LayerNorm}(\mathbf{Z} + \text{FFN}(\mathbf{Z}))$$

**Definition 5.8** (Transformer Block (Decoder))**.** A Transformer decoder block adds a cross-attention layer between masked self-attention and the FFN:

1. Masked (causal) self-attention
2. Cross-attention: $\text{MultiHead}(\mathbf{H}^{\text{dec}}, \mathbf{H}^{\text{enc}}, \mathbf{H}^{\text{enc}})$
3. Feed-forward network

Each sub-layer is followed by a residual connection and layer normalization.

## 5.5 Positional Encoding

Self-attention is permutation-invariant. To inject positional information, a positional encoding is added to the input embeddings.

**Definition 5.9** (Sinusoidal Positional Encoding)**.** The positional encoding of Vaswani et al. (2017):

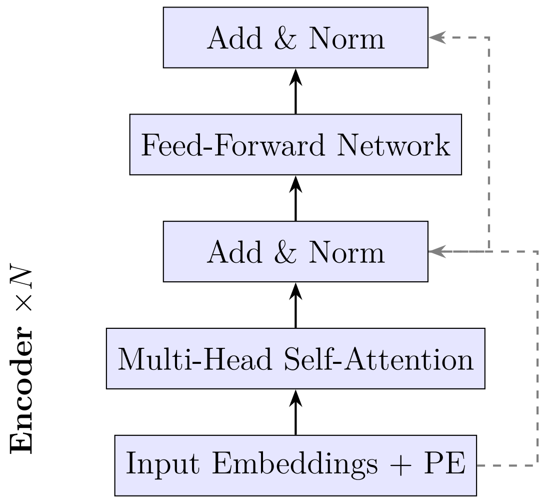
$$\mathrm{PE}(\mathrm{pos}, 2i) = \sin\left(\frac{\mathrm{pos}}{10000^{2i/d}}\right) \tag{5.6}$$

$$\mathrm{PE}(\mathrm{pos}, 2i+1) = \cos\left(\frac{\mathrm{pos}}{10000^{2i/d}}\right) \tag{5.7}$$

where pos is the position and $i$ is the dimension index.

**Proposition 5.10** (Translation Property)**.** Sinusoidal encoding allows representing position shifts as linear transformations: there exists a matrix $\mathbf{M}_k$ such that $\mathrm{PE}(\mathrm{pos}+k) = \mathbf{M}_k \cdot \mathrm{PE}(\mathrm{pos})$.

## 5.6 Transformer Architecture Diagram



## 5.7 Computational Complexity

**Proposition 5.11** (Self-Attention Complexity)**.** For a sequence of length $T$ and dimension $d$:

- **Self-attention**: $O(T^2 d)$ time, $O(T^2 + Td)$ memory.
- **RNN**: $O(Td^2)$ time (sequential).
- **FFN**: $O(Td^2)$ time (parallelizable).

Self-attention is advantageous when $T < d$ but becomes prohibitive for long sequences ($T > 4096$).

CHAPTER 5. ATTENTION AND TRANSFORMERS

**Transformer parameter count**

For a Transformer with $N$ layers, dimension $d$, $H$ heads, FFN dimension $d_{\text{ff}}$:

$$\text{Params per layer} = 4d^2 + 2d \cdot d_{\text{ff}} + \text{biases}$$

$$\text{Total} \approx N(4d^2 + 2d \cdot d_{\text{ff}}) + V \cdot d$$

## 5.8 Efficient Attention Variants

To address quadratic complexity, several variants have been proposed:

- **Sparse attention** (Longformer, BigBird): local + global attention, $O(T\sqrt{T})$.
- **Linear attention** (Performers): kernel approximation, $O(Td^2)$.
- **Flash Attention** (Dao et al., 2022): hardware-level optimization, exact but with optimized memory access patterns.

## 5.9 Attention Visualization

**Attention weight visualization with Hugging Face**

```python
from transformers import AutoTokenizer, AutoModel
import torch
import matplotlib.pyplot as plt
import seaborn as sns

model_name = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name, output_attentions=True)

text = "The cat sat on the mat because it was tired"
inputs = tokenizer(text, return_tensors="pt")
with torch.no_grad():
    outputs = model(**inputs)

# Attention weights: tuple of (num_layers) tensors of shape (B, H, T, T)
attention = outputs.attentions
layer, head = 6, 3
tokens = tokenizer.convert_ids_to_tokens(inputs["input_ids"][0])
attn_weights = attention[layer][0, head].numpy()

plt.figure(figsize=(10, 8))
sns.heatmap(attn_weights, xticklabels=tokens, yticklabels=tokens,
↪   cmap="viridis")
plt.title(f"Attention - Layer {layer}, Head {head}")
plt.tight_layout()
plt.show()
```

5.10. COMPLETE TRANSFORMER BLOCK IMPLEMENTATION

## 5.10 Complete Transformer Block Implementation

Transformer encoder block in PyTorch

```
import torch
import torch.nn as nn
import math

class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        assert d_model % num_heads == 0
        self.d_k = d_model // num_heads
        self.num_heads = num_heads
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def forward(self, Q, K, V, mask=None):
        B, T, d = Q.shape
        q = self.W_q(Q).view(B, T, self.num_heads, self.d_k).transpose(1,
        ↪  2)
        k = self.W_k(K).view(B, -1, self.num_heads,
        ↪  self.d_k).transpose(1, 2)
        v = self.W_v(V).view(B, -1, self.num_heads,
        ↪  self.d_k).transpose(1, 2)

        scores = torch.matmul(q, k.transpose(-2, -1)) /
        ↪  math.sqrt(self.d_k)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))
        attn = torch.softmax(scores, dim=-1)
        out = torch.matmul(attn, v)
        out = out.transpose(1, 2).contiguous().view(B, T, -1)
        return self.W_o(out)

class TransformerEncoderBlock(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()
        self.attn = MultiHeadAttention(d_model, num_heads)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff), nn.ReLU(),
            nn.Linear(d_ff, d_model)
        )
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        x = self.norm1(x + self.dropout(self.attn(x, x, x, mask)))
```

*CHAPTER 5. ATTENTION AND TRANSFORMERS*

```
        x = self.norm2(x + self.dropout(self.ffn(x)))
        return x
```

## 5.11 Exercises

**Exercise 5.1** ($\star$)**.** Verify that $\mathrm{Var}[\langle \mathbf{q}, \mathbf{k} \rangle] = d_k$ when the components are i.i.d. $\mathcal{N}(0, 1)$.

**Exercise 5.2** ($\star$)**.** Calculate the number of parameters in a Transformer encoder block with $d = 512$, $H = 8$, $d_{\mathrm{ff}} = 2048$.

**Exercise 5.3** ($\star\star$)**.** Show that scaled dot-product attention without the scaling factor can lead to near-zero softmax gradients for large values of $d_k$.

**Exercise 5.4** ($\star\star$)**.** Implement a causal mask for decoder self-attention and verify that it prevents the model from "seeing the future."

**Exercise 5.5** ($\star\star\star$)**.** Implement a complete encoder-decoder Transformer for French-to-English translation. Compare with an LSTM-based Seq2Seq model with Bahdanau attention on the same dataset.

# Chapter 6

# Pre-trained Models

In 2018, three papers transformed NLP: ELMo (Peters et al.), GPT (Radford et al.), and above all BERT (Devlin et al.). Their shared idea: pre-train a large language model on billions of unlabelled words, then fine-tune it on the target task with very little supervised data. This "pre-training / fine-tuning" paradigm revolutionised the field: the learned representations capture syntax, semantics, and even factual knowledge about the world, rendering previous specialised approaches largely obsolete. This was the beginning of the *foundation model* era.

**The pre-training / fine-tuning paradigm**

Rather than training a model from scratch for each task, we pre-train a large model on unlabeled data, then adapt (fine-tune) it on task-specific data. This paradigm, inaugurated by ELMo (2018) and popularized by BERT (2018), has transformed NLP.

## 6.1 Learning Contextual Representations

**Definition 6.1** (Contextual Representation)**.** A *contextual representation* is a function $\phi : \mathcal{V}^T \times \{1, \ldots, T\} \to \mathbb{R}^d$ that maps each token $x_t$ to a vector $\mathbf{h}_t$ dependent on the context $(x_1, \ldots, x_T)$:

$$\mathbf{h}_t = \phi(\mathbf{x}, t) \neq \phi(\mathbf{x}', t) \quad \text{if } \mathbf{x} \neq \mathbf{x}'$$

even when $x_t = x'_t$.

This contrasts with static embeddings where $\phi(w)$ is context-independent.

## 6.2 ELMo: Context via Bidirectional RNN

**Definition 6.2** (ELMo)**.** ELMo (Embeddings from Language Models, Peters et al., 2018) produces contextual representations by combining layers of a BiLSTM pre-trained as a bidirectional language model:

$$\text{ELMo}_t = \gamma \sum_{\ell=0}^{L} s_\ell \mathbf{h}_t^\ell$$

where $\mathbf{h}_t^\ell$ is the hidden state of layer $\ell$ at position $t$, $s_\ell$ are task-specific learned weights, and $\gamma$ is a scaling factor.

## 6.3 BERT: Bidirectional Encoder Representations from Transformers

### 6.3.1 Architecture

BERT uses a Transformer encoder (bidirectional self-attention) with the following configurations:

| | **BERT-base** | **BERT-large** |
|---|---|---|
| Layers $N$ | 12 | 24 |
| Dimension $d$ | 768 | 1024 |
| Heads $H$ | 12 | 16 |
| Parameters | 110M | 340M |

### 6.3.2 Pre-training Objectives

**Definition 6.3** (Masked Language Model (MLM))**.** The MLM objective randomly masks 15% of input tokens and trains the model to predict them:

$$\mathcal{L}_{\text{MLM}} = -\sum_{t\in\mathcal{M}} \log P(x_t \mid \mathbf{x}_{\setminus\mathcal{M}})$$

where $\mathcal{M}$ is the set of masked positions. Among selected tokens: 80% are replaced with `[MASK]`, 10% with a random token, 10% unchanged.

**Definition 6.4** (Next Sentence Prediction (NSP))**.** The NSP objective trains the model to predict whether sentence $B$ follows sentence $A$ in the original corpus:

$$\mathcal{L}_{\text{NSP}} = -\log P(\text{IsNext} \mid \texttt{[CLS]}, A, \texttt{[SEP]}, B)$$

**MLM limitations**

Masking creates a mismatch between pre-training (`[MASK]` tokens present) and inference (no `[MASK]`). Subsequent work (RoBERTa, ALBERT) showed that NSP is unnecessary and that training benefits from longer sequences and more data.

### 6.3.3 WordPiece Tokenization

BERT uses WordPiece, a BPE variant that maximizes likelihood:

$$\text{score}(a, b) = \frac{\text{count}(ab)}{\text{count}(a)\cdot\text{count}(b)}$$

## 6.4 GPT: Generative Pre-trained Transformer

**Definition 6.5** (GPT)**.** GPT (Radford et al., 2018) uses a Transformer decoder (causal self-attention) pre-trained with an autoregressive objective:

$$\mathcal{L}_{\text{GPT}} = -\sum_{t=1}^{T} \log P(x_t \mid x_1, \ldots, x_{t-1})$$

| Model | Parameters | Layers | Context |
|---|---|---|---|
| GPT-1 | 117M | 12 | 512 |
| GPT-2 | 1.5B | 48 | 1024 |
| GPT-3 | 175B | 96 | 2048 |
| GPT-4 | undisclosed | — | 128K |

*Remark* 6.6. GPT-3 demonstrated emergent *few-shot learning* capabilities: the ability to solve new tasks from a few examples presented in the prompt, without weight updates.

## 6.5 T5: Text-to-Text Transfer Transformer

**Definition 6.7** (T5)**.** T5 (Raffel et al., 2020) reformulates all NLP tasks as text-to-text problems. The model is an encoder-decoder Transformer pre-trained with a denoising (*span corruption*) objective: contiguous token spans are replaced with sentinels, and the decoder must reconstruct the masked spans.

**Architecture comparison**

| | **BERT** | **GPT** | **T5** |
|---|---|---|---|
| Architecture | Encoder | Decoder | Enc-Dec |
| Attention | Bidirectional | Causal | Bid. + Causal |
| Objective | MLM + NSP | Autoregressive | Span corruption |
| Use case | Understanding | Generation | Both |

## 6.6 Scaling Laws

**Theorem 6.8** (Scaling Laws (Kaplan et al.))**.** *The test loss $L$ of a language model follows power laws with respect to the number of parameters $N$, dataset size $D$, and compute budget $C$:*

$$L(N) \propto N^{-\alpha_N} \qquad \alpha_N \approx 0.076 \tag{6.1}$$
$$L(D) \propto D^{-\alpha_D} \qquad \alpha_D \approx 0.095 \tag{6.2}$$
$$L(C) \propto C^{-\alpha_C} \qquad \alpha_C \approx 0.050 \tag{6.3}$$

*(for GPT-type models on English data).*

*Remark* 6.9. The Chinchilla scaling laws (Hoffmann et al., 2022) show that optimal compute budget allocation follows $N \propto D$: model size and data size should be increased at the same rate.

## 6.7 Using Hugging Face

BERT for text classification

```python
from transformers import AutoTokenizer,
↪  AutoModelForSequenceClassification
from transformers import pipeline

# Sentiment pipeline
clf = pipeline("sentiment-analysis",
               model="nlptown/bert-base-multilingual-uncased-sentiment")
result = clf("This NLP course is truly excellent!")
print(result)
# [{'label': '5 stars', 'score': 0.73}]

# Direct usage
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
model = AutoModelForSequenceClassification.from_pretrained(
    "bert-base-uncased", num_labels=2
)
inputs = tokenizer("This is a great course!", return_tensors="pt")
outputs = model(**inputs)
logits = outputs.logits  # (1, 2)
```

GPT-2 for text generation

```python
from transformers import pipeline

generator = pipeline("text-generation", model="gpt2")
text = generator(
    "Natural language processing is",
    max_length=50,
    num_return_sequences=1,
    temperature=0.7
)
print(text[0]["generated_text"])
```

T5 for summarization

```python
from transformers import pipeline

summarizer = pipeline("summarization", model="t5-base")
article = """
Natural language processing (NLP) is a subfield of linguistics,
computer science, and artificial intelligence concerned with the
interactions between computers and human language. The goal is to
enable computers to understand, interpret, and generate human language.
"""
summary = summarizer(article, max_length=30, min_length=10)
```

```python
print(summary[0]["summary_text"])
```

## 6.8 Variants and Evolution

**RoBERTa** (Liu et al., 2019): removes NSP, trains longer with more data and longer sequences.

**ALBERT** (Lan et al., 2020): embedding factorization and cross-layer parameter sharing to reduce size.

**DeBERTa** (He et al., 2021): disentangled attention with relative position.

**LLaMA** (Touvron et al., 2023): Meta's open model family with RMSNorm, RoPE, and SwiGLU.

**Mistral / Mixtral** (2023–2024): grouped-query attention (GQA), mixture of experts (MoE).

## 6.9 Feature Extraction and Probing

**Definition 6.10** (Probing Classifiers)**.** *Probing classifiers* are simple classifiers (logistic regression) trained on frozen representations from a pre-trained model to evaluate what linguistic information is captured by each layer.

Studies have shown that lower layers capture morphology and syntax, while upper layers encode more semantic information.

## 6.10 Exercises

**Exercise 6.1** ($\star$)**.** Compare the parameter counts of BERT-base and GPT-2 (117M). What architectural differences explain the gaps?

**Exercise 6.2** ($\star$)**.** Explain why BERT cannot be directly used for autoregressive text generation.

**Exercise 6.3** ($\star\star$)**.** Implement a probing classifier to evaluate whether BERT layer-6 representations encode POS tagging information. Use the Universal Dependencies dataset.

**Exercise 6.4** ($\star\star$)**.** Reproduce BERT's masking experiment: compute the pseudo-log-likelihood perplexity $\mathrm{PLL}(\mathbf{x}) = \sum_{t=1}^{T} \log P(x_t \mid \mathbf{x}_{\setminus t})$ on a set of sentences.

**Exercise 6.5** ($\star\star\star$)**.** Study the impact of scaling laws: train Transformer language models of increasing sizes (1M, 10M, 100M parameters) on the same corpus and plot the loss vs. parameters curve on a log-log scale.

# Chapter 7

# Fine-tuning and Instruction Tuning

In 2018, GPT and BERT demonstrated that a model pre-trained on massive text corpora acquires a general understanding of language, transferable to specific tasks. But how does one go from generalist to specialist? *Fine-tuning* — adjusting the model's weights on a targeted dataset — is the classic answer. Yet as models grew to billions of parameters, full fine-tuning became prohibitive. Two revolutions followed in quick succession: first, parameter-efficient methods like LoRA (Hu et al., 2022), which modify only a tiny fraction of weights; then *instruction tuning* and RLHF (Reinforcement Learning from Human Feedback), which align the model not to a task but to *human preferences*. It is this last ingredient that transforms a language model into a conversational assistant. This chapter traces that trajectory, from classical fine-tuning to alignment through reinforcement.

**Adapting a generalist model**

A pre-trained model is a language "generalist." Fine-tuning and instruction tuning transform it into a specialist for a specific task or desired behavior, often with limited labeled data.

## 7.1 Classical Fine-tuning

**Definition 7.1** (Fine-tuning)**.** *Fine-tuning* consists of re-training a pre-trained model $\theta_{\text{pre}}$ on a task-specific dataset $\mathcal{D}_{\text{task}} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$:

$$\theta^* = \arg\min_\theta \sum_{i=1}^N \mathcal{L}(f_\theta(\mathbf{x}_i), y_i), \quad \theta_0 = \theta_{\text{pre}}$$

**Fine-tuning strategies**

- **Learning rate**: use a small learning rate ($2 \times 10^{-5}$ to $5 \times 10^{-5}$ for BERT).
- **Warmup**: gradual learning rate increase over the first iterations.
- **Epochs**: 2–4 epochs are generally sufficient.
- **Partial freezing**: freeze lower layers and only fine-tune upper layers for small datasets.

**Fine-tuning BERT with Hugging Face Trainer**

```
from transformers import (AutoTokenizer,
↪  AutoModelForSequenceClassification,
                          Trainer, TrainingArguments)
from datasets import load_dataset

dataset = load_dataset("imdb")
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

def tokenize_fn(examples):
    return tokenizer(examples["text"], truncation=True,
    ↪  padding="max_length",
                     max_length=256)

tokenized = dataset.map(tokenize_fn, batched=True)
model = AutoModelForSequenceClassification.from_pretrained(
    "bert-base-uncased", num_labels=2
)

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=3,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=32,
    learning_rate=2e-5,
    warmup_steps=500,
    weight_decay=0.01,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized["train"],
    eval_dataset=tokenized["test"],
)
trainer.train()
```

## 7.2 Parameter-Efficient Fine-Tuning (PEFT)

Full fine-tuning of large models is expensive in memory and compute. PEFT methods update only a small fraction of parameters.

### 7.2.1 LoRA: Low-Rank Adaptation

**Definition 7.2** (LoRA)**.** LoRA (Hu et al., 2022) injects low-rank matrices into attention layers. For a weight matrix $\mathbf{W}_0 \in \mathbb{R}^{d\times d}$, the update is:

$$\mathbf{W} = \mathbf{W}_0 + \Delta\mathbf{W} = \mathbf{W}_0 + \mathbf{B}\mathbf{A}$$

where $\mathbf{B} \in \mathbb{R}^{d\times r}$, $\mathbf{A} \in \mathbb{R}^{r\times d}$ with $r \ll d$ (typically $r \in [4, 64]$). Only $\mathbf{A}$ and $\mathbf{B}$ are trained; $\mathbf{W}_0$ is frozen.

**Proposition 7.3** (LoRA Parameter Complexity)**.** The number of trainable parameters per layer is $2rd$ instead of $d^2$. The compression ratio is:

$$\frac{2rd}{d^2} = \frac{2r}{d}$$

For $d = 4096$ and $r = 16$: 0.78% of the original parameters.

**LoRA with PEFT**

```python
from peft import LoraConfig, get_peft_model, TaskType
from transformers import AutoModelForCausalLM

model = AutoModelForCausalLM.from_pretrained("meta-llama/Llama-2-7b-hf")

lora_config = LoraConfig(
    task_type=TaskType.CAUSAL_LM,
    r=16,
    lora_alpha=32,
    lora_dropout=0.1,
    target_modules=["q_proj", "v_proj", "k_proj", "o_proj"],
)

model = get_peft_model(model, lora_config)
model.print_trainable_parameters()
# trainable params: 4,194,304 || all params: 6,742,609,920 || trainable%:
↪  0.062
```

### 7.2.2 QLoRA

**Definition 7.4** (QLoRA)**.** QLoRA (Dettmers et al., 2023) combines LoRA with 4-bit quantization of the base model, enabling fine-tuning of 65B parameter models on a single 48 GB GPU.

### 7.2.3 Other PEFT Methods

- **Prefix tuning** (Li & Liang, 2021): adds learnable vectors to the beginning of attention keys and values.
- **Adapters** (Houlsby et al., 2019): inserts small feed-forward networks between Transformer layers.
- **IA$^3$** (Liu et al., 2022): learned scaling vectors for keys, values, and FFN.

## 7.3 Instruction Tuning

**Definition 7.5** (Instruction Tuning)**.** *Instruction tuning* trains a language model on a set of natural language instructions paired with expected responses:

$$\mathcal{D}_{\text{inst}} = \{(\text{instruction}_i, \text{response}_i)\}_{i=1}^{N}$$

The model learns to follow general instructions rather than solve a single task.

Examples of models: FLAN-T5 (Chung et al., 2022), InstructGPT (Ouyang et al., 2022).

## 7.4 RLHF: Reinforcement Learning from Human Feedback

**Definition 7.6** (RLHF)**.** RLHF (*Reinforcement Learning from Human Feedback*) aligns a language model with human preferences in three stages:

1. **SFT** (Supervised Fine-Tuning): supervised fine-tuning on human demonstrations.
2. **Reward model**: train a model $r_\phi$ that predicts human preference between two responses:
$$\mathcal{L}_{\text{RM}} = -\mathbb{E}_{(y_w, y_l)}\left[\log \sigma(r_\phi(y_w) - r_\phi(y_l))\right]$$
where $y_w$ is the preferred response and $y_l$ the less preferred.
3. **PPO optimization**: optimize the policy $\pi_\theta$ via proximal policy optimization:
$$\mathcal{L}_{\text{PPO}} = \mathbb{E}\left[r_\phi(y) - \beta \text{KL}(\pi_\theta \| \pi_{\text{ref}})\right]$$
where the KL term prevents drift from the reference model $\pi_{\text{ref}}$.

**DPO (Direct Preference Optimization) Objective**

DPO (Rafailov et al., 2023) eliminates the explicit reward model:

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}_{(x, y_w, y_l)}\left[\log \sigma\left(\beta \log \frac{\pi_\theta(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \beta \log \frac{\pi_\theta(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)}\right)\right]$$

## 7.5 Alignment and Safety

**Definition 7.7** (Aligned Model)**.** A language model is said to be *aligned* if it satisfies the **HHH** criteria (Helpful, Honest, Harmless):

- **Helpful**: responds to requests in an informative manner.
- **Honest**: does not fabricate information, expresses uncertainty.
- **Harmless**: refuses dangerous requests, avoids toxic content.

**Red teaming and jailbreaking**

*Red teaming* consists of systematically testing the limits of an aligned model. *Jailbreaking* refers to techniques for evading guardrails (prompt injection, system role manipulation). Alignment is a continuous process, not a final state.

## 7.6 Transfer Learning and Domain Adaptation

**Definition 7.8** (Domain Adaptation). *Domain adaptation* consists of further pre-training a model on target-domain data before task-specific fine-tuning. Examples: BioBERT (biomedical), SciBERT (scientific), FinBERT (finance).

**Theorem 7.9** (Generalization Bound in Transfer Learning). *Let $\epsilon_S(\theta)$ be the source domain error and $\epsilon_T(\theta)$ be the target domain error. Under bounded divergence assumptions $d_{\mathcal{H}}(\mathcal{D}_S, \mathcal{D}_T)$:*

$$\epsilon_T(\theta) \leq \epsilon_S(\theta) + d_{\mathcal{H}}(\mathcal{D}_S, \mathcal{D}_T) + \lambda$$

*where $\lambda$ is the sum of optimal errors on both domains.*

## 7.7 Exercises

**Exercise 7.1** ($\star$). Calculate the number of trainable parameters with LoRA ($r = 8$) applied to all 4 attention projections of a model with $d = 4096$ and $N = 32$ layers.

**Exercise 7.2** ($\star\star$). Implement BERT fine-tuning for sentiment classification on IMDB with Hugging Face `Trainer`. Compare performance with full freezing, partial freezing (first 6 layers), and full fine-tuning.

**Exercise 7.3** ($\star\star$). Derive the DPO objective from the RLHF objective by showing that the optimal policy under KL constraint has an analytical form.

**Exercise 7.4** ($\star\star$). Compare LoRA and full fine-tuning in terms of performance and training time on a NER task.

**Exercise 7.5** ($\star\star\star$). Implement a simplified RLHF pipeline: (1) SFT on a small instruction dataset, (2) reward model via binary preferences, (3) PPO optimization (use `trl`).

*CHAPTER 7. FINE-TUNING AND INSTRUCTION TUNING*

# Chapter 8

# Text Generation and Decoding

A language model that predicts the next word is, quite literally, a text generator in waiting. But between the distribution $P(x_t \mid x_{<t})$ and coherent, fluent, informative text lies a chasm — that of the *decoding strategy*. Systematically choosing the most probable word (greedy decoding) produces repetitive, dull text. Naïvely sampling from the distribution produces incoherent text. The solutions — beam search, top-$k$, nucleus sampling (top-$p$), temperature — are all trade-offs between *quality* and *diversity*. Ari Holtzman and collaborators, in their paper "The Curious Case of Neural Text Degeneration" (2020), showed that nucleus sampling elegantly solves the degeneration problem. This chapter explores these strategies and their probabilistic foundations.

**From language model to generator**

An autoregressive language model defines $P(x_t \mid x_{<t})$. Generating text amounts to sequentially sampling from this distribution. The choice of decoding strategy radically influences the quality, diversity, and coherence of the produced text.

## 8.1 Greedy Decoding

**Definition 8.1** (Greedy Decoding)**.** Greedy decoding selects the most probable token at each step:

$$x_t = \arg\max_{w \in \mathcal{V}} P(w \mid x_{<t})$$

**Limitations of greedy decoding**

Greedy decoding does not guarantee the globally optimal sequence. It can produce repetitions and lacks diversity. The most probable sequence is not always the most natural one.

## 8.2 Beam Search

**Definition 8.2** (Beam Search)**.** Beam search maintains the $B$ most probable partial sequences (*beams*) at each decoding step:

$$\mathcal{B}_t = \text{top-}B\left\{(s \oplus w, \text{score}(s) + \log P(w \mid s)) : s \in \mathcal{B}_{t-1}, w \in \mathcal{V}\right\}$$

where $B$ is the beam width.

CHAPTER 8. TEXT GENERATION AND DECODING

**Proposition 8.3** (Properties of Beam Search)**.**

1. For $B = 1$: equivalent to greedy decoding.
2. For $B = |\mathcal{V}|^T$: equivalent to exhaustive search.
3. In practice, $B \in [4, 10]$ offers a good trade-off.

### 8.2.1 Length Normalization

Beam search favors shorter sequences (more $\log P < 0$ terms for longer ones). We normalize the score by length:

$$\text{score}_{\text{norm}}(s) = \frac{1}{|s|^\alpha} \sum_{t=1}^{|s|} \log P(x_t \mid x_{<t})$$

where $\alpha \in [0.6, 1.0]$ is a hyperparameter (Wu et al., 2016).

## 8.3 Stochastic Sampling

**Definition 8.4** (Temperature Sampling)**.** Temperature sampling with $\tau > 0$ modifies the distribution:

$$P_\tau(w \mid x_{<t}) = \frac{\exp(\text{logit}_w/\tau)}{\sum_{w'} \exp(\text{logit}_{w'}/\tau)}$$

- $\tau \to 0$: converges to greedy decoding.
- $\tau = 1$: original distribution.
- $\tau > 1$: more uniform distribution (more diversity).

**Definition 8.5** (Top-$k$ Sampling)**.** Top-$k$ sampling (Fan et al., 2018) restricts sampling to the $k$ most probable tokens:

$$P_{\text{top-}k}(w \mid x_{<t}) = \begin{cases} \frac{P(w|x_{<t})}{\sum_{w' \in \mathcal{V}_k} P(w'|x_{<t})} & \text{if } w \in \mathcal{V}_k \\ 0 & \text{otherwise} \end{cases}$$

where $\mathcal{V}_k$ contains the $k$ most probable tokens.

**Definition 8.6** (Top-$p$ Sampling (Nucleus))**.** Top-$p$ sampling (Holtzman et al., 2020) selects the smallest set $\mathcal{V}_p$ such that:

$$\sum_{w \in \mathcal{V}_p} P(w \mid x_{<t}) \geq p$$

then samples from $\mathcal{V}_p$ (after renormalization).

**Combining strategies**

In practice, temperature, top-$k$, and top-$p$ are often combined. For example, $\tau = 0.7$, $k = 50$, $p = 0.9$ offers a good balance between coherence and diversity for creative generation.

## 8.4 Repetition Penalties

**Definition 8.7** (Repetition Penalty)**.** To avoid repetitions, already-generated tokens are penalized:

$$\text{logit}'_w = \begin{cases} \text{logit}_w/\theta & \text{if } w \in x_{<t} \text{ and } \text{logit}_w > 0 \\ \text{logit}_w \cdot \theta & \text{if } w \in x_{<t} \text{ and } \text{logit}_w \leq 0 \end{cases}$$

where $\theta > 1$ is the penalty factor.

## 8.5 Contrastive Decoding

**Definition 8.8** (Contrastive Decoding)**.** Contrastive decoding (Li et al., 2023) selects tokens that maximize the log-probability difference between an expert model $\pi_{\text{exp}}$ and an amateur model $\pi_{\text{ama}}$:

$$x_t = \arg\max_{w \in \mathcal{V}_p} [\log \pi_{\text{exp}}(w \mid x_{<t}) - \alpha \log \pi_{\text{ama}}(w \mid x_{<t})]$$

## 8.6 Speculative Decoding

**Definition 8.9** (Speculative Decoding)**.** Speculative decoding (Leviathan et al., 2023) uses a small fast model (*draft model*) to propose $K$ tokens, then a large model verifies them in parallel. Accepted tokens are those whose probability under the large model is sufficient, accelerating generation without changing the output distribution.

## 8.7 Generation Quality Metrics

**Generation metrics**

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \quad (n\text{-gram precision}) \tag{8.1}$$

$$\text{ROUGE-L} = F_1(\text{LCS}) \quad (\text{longest common subsequence}) \tag{8.2}$$

$$\text{METEOR} = F_1 \cdot (1 - \text{Penalty}) \quad (\text{flexible alignment}) \tag{8.3}$$

$$\text{BERTScore} = F_1(\text{BERT cosine sim}) \quad (\text{contextual similarity}) \tag{8.4}$$

## 8.8 Implementation

**Decoding strategies with Hugging Face**

```
from transformers import AutoTokenizer, AutoModelForCausalLM

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

CHAPTER 8. TEXT GENERATION AND DECODING

```python
prompt = "The future of artificial intelligence"
input_ids = tokenizer.encode(prompt, return_tensors="pt")

# Greedy decoding
greedy = model.generate(input_ids, max_length=50)

# Beam search
beam = model.generate(input_ids, max_length=50, num_beams=5,
                      length_penalty=0.8, no_repeat_ngram_size=3)

# Top-k + Top-p sampling
sampled = model.generate(input_ids, max_length=50, do_sample=True,
                         temperature=0.7, top_k=50, top_p=0.9,
                         repetition_penalty=1.2)

for name, ids in [("Greedy", greedy), ("Beam", beam), ("Sampled",
↪   sampled)]:
    print(f"{name}: {tokenizer.decode(ids[0],
    ↪   skip_special_tokens=True)}")
```

**Step-by-step decoding**

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

def generate_top_p(model, input_ids, max_new_tokens=50, temperature=0.8,
                   top_p=0.9):
    generated = input_ids.clone()
    for _ in range(max_new_tokens):
        with torch.no_grad():
            outputs = model(generated)
            logits = outputs.logits[:, -1, :] / temperature

        # Top-p filtering
        sorted_logits, sorted_indices = torch.sort(logits,
        ↪   descending=True)
        cumulative_probs = torch.cumsum(
            F.softmax(sorted_logits, dim=-1), dim=-1
        )
        mask = cumulative_probs - F.softmax(sorted_logits, dim=-1) >=
        ↪   top_p
        sorted_logits[mask] = float('-inf')
        logits.scatter_(1, sorted_indices, sorted_logits)

        probs = F.softmax(logits, dim=-1)
        next_token = torch.multinomial(probs, num_samples=1)
        generated = torch.cat([generated, next_token], dim=-1)

        if next_token.item() == tokenizer.eos_token_id:
            break
    return generated
```

## 8.9 Controlled Generation

**Definition 8.10** (Controlled Generation)**.** *Controlled generation* aims to steer a language model's output according to specified attributes (tone, topic, style) without retraining. Approaches:

- **PPLM** (Dathathri et al., 2020): gradient perturbation.
- **Classifier-free guidance**: interpolation between conditional and unconditional distributions.
- **Constrained decoding**: hard lexical constraints.

## 8.10 Exercises

**Exercise 8.1** ($\star$)**.** Show that greedy decoding is a special case of beam search with $B = 1$.

**Exercise 8.2** ($\star$)**.** Compute the top-$p$ distribution for $p = 0.9$ given the probabilities $(0.4, 0.3, 0.15, 0.1, 0.05)$.

**Exercise 8.3** ($\star\star$)**.** Implement beam search with length normalization. Test with different values of $\alpha$ and $B$ on GPT-2.

**Exercise 8.4** ($\star\star$)**.** Quantitatively compare (BLEU, diversity) generation via greedy, beam search ($B = 5$), and nucleus sampling ($p = 0.9$) on a test corpus.

**Exercise 8.5** ($\star\star\star$)**.** Implement speculative decoding with a draft model (GPT-2 small) and a target model (GPT-2 large). Measure the speedup obtained and verify that the output distribution is preserved.

*CHAPTER 8. TEXT GENERATION AND DECODING*

# Chapter 9

# Retrieval-Augmented Generation

Large language models are remarkably capable, but they have two structural weaknesses: they "hallucinate" (generate plausible but false information) and their knowledge is frozen at training time. In 2020, Patrick Lewis and collaborators at Facebook AI Research proposed an elegant solution: *Retrieval-Augmented Generation* (RAG). The principle is to couple the generative model with a document retrieval system: before generating a response, the model retrieves relevant passages from an external knowledge base and uses them as context. This architecture decouples *memory* (the document base, easily updated) from *reasoning* (the language model). RAG is today at the heart of enterprise question-answering systems, specialized chatbots, and research assistants.

**Why augment with retrieval?**

Generative language models (LLMs) suffer from hallucinations and knowledge frozen at training time. Retrieval-Augmented Generation (RAG) addresses these problems by providing the model with relevant documents from an external knowledge base before generation. This grounds the output in verifiable facts and enables knowledge updates without retraining.

## 9.1 RAG Framework

**Definition 9.1** (Retrieval-Augmented Generation)**.** A **RAG** system combines a *retriever* and a *generator*. Given a query $q$ and a corpus $\mathcal{C}$:

1. **Retrieve**: extract the $k$ most relevant documents $\{d_1, \ldots, d_k\} = \text{Retrieve}(q, \mathcal{C})$
2. **Generate**: produce the answer $y = \text{Generate}(q, d_1, \ldots, d_k)$

**Probabilistic RAG Formulation**

$$P(y|q) = \sum_{d \in \mathcal{C}} P(d|q)\, P(y|q,d) \approx \sum_{i=1}^{k} P(d_i|q)\, P_{\text{LLM}}(y|q,d_i)$$

where $P(d|q)$ is the relevance score and $P_{\text{LLM}}(y|q,d)$ is the generative probability conditioned on the document.

**Proposition 9.2** (Benefits of RAG)**.** 1. **Up-to-date knowledge**: the knowledge base can be updated without retraining the model.

CHAPTER 9. RETRIEVAL-AUGMENTED GENERATION

2. **Traceability**: sources are identifiable (citations).
3. **Reduced hallucinations**: the model grounds its output in verifiable facts.
4. **Parameter efficiency**: a smaller model with retrieval can match the quality of a larger model.

## 9.2 Dense Retrieval

**Definition 9.3** (Dense Passage Retrieval (DPR))**.** DPR (Karpukhin et al., 2020) encodes queries and documents into a shared vector space:

$$\text{sim}(q, d) = \mathbf{e}_q^\top \mathbf{e}_d = E_Q(q)^\top E_D(d)$$

where $E_Q$ and $E_D$ are BERT encoders fine-tuned to maximize similarity between queries and relevant passages.

**Definition 9.4** (Contrastive Training Loss for DPR)**.** Training uses a contrastive loss over positive and negative passages:

$$\mathcal{L} = -\log \frac{\exp(\text{sim}(q, d^+))}{\exp(\text{sim}(q, d^+)) + \sum_{j=1}^{n} \exp(\text{sim}(q, d_j^-))}$$

where $d^+$ is the relevant passage and $\{d_j^-\}$ are negatives (in-batch negatives and hard negatives from BM25).

**Definition 9.5** (ColBERT)**.** ColBERT (Khattab et al., 2020) uses a *late interaction* between query and document tokens:

$$\text{sim}(q, d) = \sum_{i=1}^{|q|} \max_{j=1}^{|d|} \mathbf{q}_i^\top \mathbf{d}_j$$

where $\mathbf{q}_i$ and $\mathbf{d}_j$ are contextual token embeddings. This fine-grained interaction improves precision while remaining efficient through precomputed document embeddings.

*Remark* 9.6. Dense retrieval outperforms traditional sparse methods (BM25) on many benchmarks, especially for semantic matching. However, BM25 remains competitive for exact keyword matching and is often used as a first-stage retriever or for generating hard negatives.

## 9.3 Vector Databases

**Definition 9.7** (Vector Database)**.** A **vector database** stores embedding vectors and supports Approximate Nearest Neighbor (ANN) search in sub-linear time. Examples include FAISS, Pinecone, Weaviate, Qdrant, and ChromaDB.

**Proposition 9.8** (Indexing Methods)**.** The main ANN indexing methods are:

- **IVF** (Inverted File): partitions the space into Voronoi cells for coarse search.
- **HNSW** (Hierarchical Navigable Small World): a navigable graph structure offering excellent speed-accuracy trade-offs.
- **PQ** (Product Quantization): compresses vectors via quantization to reduce memory footprint.

## 9.4 Chunking Strategies

**Definition 9.9** (Document Chunking). **Chunking** divides documents into fragments suitable for the model's context window:

- **Fixed-size**: $n$ tokens with overlap of $m$ tokens.
- **Paragraph-based**: respects document structure.
- **Semantic**: splits at topic boundaries (using a segmentation model).
- **Recursive**: hierarchical splitting (title $\rightarrow$ section $\rightarrow$ paragraph).

**Chunk Size Trade-off**

Chunks that are too small lose context; chunks that are too large dilute relevant information and waste the model's token budget. In practice, 256–512 tokens with 50–100 overlap offers a good balance.

## 9.5 RAG Architectures

**Definition 9.10** (RAG-Sequence and RAG-Token). Lewis et al. (2020) propose two variants:

- **RAG-Sequence**: a single document is used for the entire generation: $P(y|q) \approx \sum_d P(d|q) \prod_t P(y_t|q, d, y_{<t})$
- **RAG-Token**: the document can change at each token: $P(y|q) \approx \prod_t \sum_d P(d|q)P(y_t|q, d, y_{<t})$

### 9.5.1 Advanced RAG

**Definition 9.11** (Self-RAG). Self-RAG (Asai et al., 2023) teaches the model to decide *when* to retrieve (via special reflection tokens) and to evaluate the relevance of retrieved documents before using them.

**Definition 9.12** (RAPTOR). RAPTOR builds a hierarchical tree of summaries: leaves are original chunks, internal nodes are cluster summaries. Retrieval traverses the tree from general to specific.

**Intuition**

Modern RAG pipelines are increasingly sophisticated: they include reranking of retrieved documents, query rewriting, iterative retrieval (the model can issue follow-up queries), and self-reflection on whether the retrieved context is sufficient.

## 9.6 Evaluating RAG Systems

**Definition 9.13** (RAG Evaluation Metrics). RAG evaluation covers three dimensions:

1. **Retrieval quality**: recall@$k$, precision@$k$, MRR (Mean Reciprocal Rank).
2. **Faithfulness**: is the answer consistent with the retrieved documents?
3. **Answer relevance**: does the answer correctly address the question?

*CHAPTER 9. RETRIEVAL-AUGMENTED GENERATION*

**Recall@$k$ and MRR**

$$\text{Recall@}k = \frac{|\{d \in \text{top-}k : d \in \mathcal{R}\}|}{|\mathcal{R}|} \qquad \text{MRR} = \frac{1}{|Q|}\sum_{q \in Q} \frac{1}{\text{rank}(d_q^+)}$$

where $\mathcal{R}$ is the set of relevant documents and $d_q^+$ is the first relevant document.

**Example 9.14** (RAGAS Framework)**.** The RAGAS framework automatically evaluates RAG systems on four metrics: faithfulness, answer relevance, context relevance, and context recall. It uses an LLM as a judge to assess each dimension.

## 9.7 Python Implementation

**Minimal RAG Pipeline**

```python
from sentence_transformers import SentenceTransformer
import numpy as np

class SimpleRAG:
    def __init__(self, docs, model_name="all-MiniLM-L6-v2"):
        self.encoder = SentenceTransformer(model_name)
        self.docs = docs
        self.embeddings = self.encoder.encode(docs)

    def retrieve(self, query, k=3):
        q_emb = self.encoder.encode([query])
        scores = (self.embeddings @ q_emb.T).squeeze()
        top_k = np.argsort(scores)[-k:][::-1]
        return [(self.docs[i], scores[i]) for i in top_k]

    def generate_prompt(self, query, k=3):
        results = self.retrieve(query, k)
        context = "\n\n".join([doc for doc, _ in results])
        return (f"Context:\n{context}\n\n"
                f"Question: {query}\nAnswer:")

# Usage
docs = ["Paris is the capital of France.",
        "Berlin is the capital of Germany.",
        "The Eiffel Tower is 330 meters tall."]
rag = SimpleRAG(docs)
prompt = rag.generate_prompt("What is the capital of France?")
```

## 9.8 Exercises

**Exercise 9.1** (RAG Pipeline)**.** Implement a complete RAG pipeline using ChromaDB as the vector store and a HuggingFace model for generation. Test on a corpus of Wikipedia articles.

**Exercise 9.2** (Chunking Strategies)**.** Compare three chunking strategies (fixed-size, paragraph-based, semantic) on a corpus of 100 documents. Measure recall@5 for a set of 50 questions.

**Exercise 9.3** (DPR vs BM25)**.** Compare DPR (dense retrieval) with BM25 (sparse retrieval) on the Natural Questions dataset. In which cases does dense retrieval outperform BM25? And vice versa?

**Exercise 9.4** (RAG Evaluation)**.** Design an evaluation protocol for a medical question-answering RAG system. Which metrics are critical? How should hallucinations be handled in a medical context?

*CHAPTER 9. RETRIEVAL-AUGMENTED GENERATION*

# Chapter 10

# LLM Evaluation

**Why evaluation is hard**

Evaluating NLP systems is fundamentally difficult because language is ambiguous, subjective, and context-dependent. A translation can be faithful but awkward, a summary can be fluent but unfaithful. Each task requires tailored metrics, and no automatic metric perfectly captures human-perceived quality.

## 10.1 Intrinsic vs Extrinsic Evaluation

**Definition 10.1** (Intrinsic Evaluation)**. Intrinsic evaluation** measures the quality of an isolated component, independently of its downstream application. Examples: perplexity of a language model, POS tagging accuracy.

**Definition 10.2** (Extrinsic Evaluation)**. Extrinsic evaluation** measures the impact of a component on a complete downstream task. Example: the effect of a better language model on dialogue system quality.

*Remark* 10.3. Intrinsic evaluation is reproducible and inexpensive but does not guarantee practical utility. Extrinsic evaluation is more relevant but costlier and harder to isolate.

## 10.2 Perplexity

**Definition 10.4** (Perplexity)**.** The **perplexity** of a language model $P$ on a test corpus $w_1, \ldots, w_N$ is:

$$\text{PPL} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_{<i})\right) = \exp(H_{\text{cross}})$$

where $H_{\text{cross}}$ is the cross-entropy between the true distribution and the model.

**Proposition 10.5** (Interpreting Perplexity)**.**

- PPL $= |\mathcal{V}|$: the model is uniform (no prediction).
- PPL $= 1$: perfect prediction.
- In practice, GPT-2 achieves PPL $\approx 22$ on WikiText-103.

**Limitations of Perplexity**

Perplexity only measures the model's predictive ability on the test corpus. It does not capture coherence, factual faithfulness, or stylistic quality. Two models with the same perplexity can generate text of very different qualities.

## 10.3 BLEU

**Definition 10.6** (BLEU)**.** BLEU (Bilingual Evaluation Understudy, Papineni et al., 2002) measures translation quality by comparing $n$-grams between a candidate $c$ and reference translations $\{r\}$:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} \frac{1}{N} \log p_n\right)$$

where $p_n$ is the modified $n$-gram precision and BP is the brevity penalty.

**BLEU Components**

**Modified $n$-gram precision**:

$$p_n = \frac{\sum_{n\text{-gram}\in c} \min(\text{count}(c), \max_r \text{count}(r))}{\sum_{n\text{-gram}\in c} \text{count}(c)}$$

**Brevity penalty**:

$$\text{BP} = \begin{cases} 1 & \text{if } |c| > |r| \\ \exp(1 - |r| \,/\, |c|) & \text{otherwise} \end{cases}$$

*Remark* 10.7. BLEU-4 (with $N = 4$) is the standard for machine translation. It correlates reasonably with human judgment at the corpus level but poorly at the sentence level.

## 10.4 ROUGE

**Definition 10.8** (ROUGE)**.** ROUGE (Recall-Oriented Understudy for Gisting Evaluation, Lin, 2004) measures summary quality with a focus on recall:

- **ROUGE-N**: $n$-gram recall: $\text{ROUGE-N} = \frac{\sum_{n\text{-gram}\in r} \min(\text{count}(c), \text{count}(r))}{\sum_{n\text{-gram}\in r} \text{count}(r)}$
- **ROUGE-L**: longest common subsequence (LCS) based F-measure.

**Proposition 10.9** (BLEU vs ROUGE)**.**

| | **BLEU** | **ROUGE** |
|---|---|---|
| Orientation | Precision | Recall |
| Primary use | Translation | Summarization |
| Granularity | Corpus | Document |

## 10.5 BERTScore

**Definition 10.10** (BERTScore)**.** BERTScore (Zhang et al., 2020) uses contextual embeddings from BERT to compute similarity between reference and candidate tokens:

$$P_{\text{BERT}} = \frac{1}{|c|} \sum_{c_i \in c} \max_{r_j \in r} \mathbf{c}_i^\top \mathbf{r}_j, \qquad R_{\text{BERT}} = \frac{1}{|r|} \sum_{r_j \in r} \max_{c_i \in c} \mathbf{r}_j^\top \mathbf{c}_i$$

$$F_{\text{BERT}} = 2 \cdot \frac{P_{\text{BERT}} \cdot R_{\text{BERT}}}{P_{\text{BERT}} + R_{\text{BERT}}}$$

*Remark* 10.11. BERTScore captures semantic similarity rather than exact word matching, making it more robust to paraphrases. It correlates better with human judgment than BLEU and ROUGE on many tasks.

## 10.6 Human Evaluation

**Definition 10.12** (Human Evaluation Protocols)**.** Human evaluation remains the gold standard. Common protocols include:

1. **Likert scale**: annotators rate from 1 to 5 on criteria (fluency, coherence, relevance).
2. **Pairwise comparison**: "Which response is better, A or B?" (more reliable than absolute scales).
3. **Ranking**: order $n$ outputs from best to worst.
4. **Elo rating**: dynamic ranking through iterative comparisons (used by Chatbot Arena).

**Definition 10.13** (Inter-Annotator Agreement)**. Cohen's kappa** measures agreement between two annotators corrected for chance:

$$\kappa = \frac{P_o - P_e}{1 - P_e}$$

where $P_o$ is observed agreement and $P_e$ is expected agreement by chance. $\kappa > 0.8$ indicates excellent agreement.

**Human Evaluation Biases**

Human evaluation suffers from biases: preference for longer responses, position bias (first option is often preferred), annotator fatigue, and cultural variability. Rigorous protocols (randomization, explicit criteria, calibration) are essential.

## 10.7 Benchmarks: GLUE, SuperGLUE, and Beyond

**Definition 10.14** (GLUE)**.** The **GLUE** benchmark (General Language Understanding Evaluation, Wang et al., 2018) comprises 9 language understanding tasks: textual entailment (MNLI, RTE, WNLI), similarity (STS-B, MRPC, QQP), sentiment (SST-2), acceptability (CoLA), and QA (QNLI).

**Definition 10.15** (SuperGLUE)**. SuperGLUE** (Wang et al., 2019) succeeded GLUE with harder tasks: BoolQ, CB, COPA, MultiRC, ReCoRD, RTE, WiC, WSC. Current models surpass human performance on SuperGLUE.

*Remark* 10.16. Since LLMs have saturated GLUE and SuperGLUE, more challenging benchmarks have emerged: MMLU (multi-domain knowledge), BIG-Bench (emergent tasks), HELM (holistic evaluation), and LMSYS Chatbot Arena (user-based evaluation).

## 10.8 LLM-as-a-Judge

**Definition 10.17** (LLM-as-a-Judge)**.** The **LLM-as-a-Judge** approach uses a powerful language model (GPT-4, Claude) to evaluate outputs from other models. The judge receives a structured prompt with evaluation criteria and returns a score or ranking.

**Leaderboard Pitfalls**

Leaderboards can be misleading:

1. **Benchmark overfitting**: models are optimized specifically for test benchmarks.
2. **Data contamination**: test data may appear in training corpora.
3. **Inappropriate metrics**: a single score masks strengths and weaknesses.
4. **Lack of diversity**: benchmarks underrepresent certain languages and cultures.

## 10.9 Python Implementation

**Computing BLEU and ROUGE**

```python
from nltk.translate.bleu_score import sentence_bleu
from rouge_score import rouge_scorer

# BLEU
reference = [["the", "cat", "is", "on", "the", "mat"]]
candidate = ["the", "cat", "is", "sitting", "on", "the", "mat"]
bleu = sentence_bleu(reference, candidate)
print(f"BLEU: {bleu:.4f}")

# ROUGE
scorer = rouge_scorer.RougeScorer(
    ['rouge1', 'rouge2', 'rougeL'], use_stemmer=True)
ref = "the cat is on the mat"
hyp = "the cat is sitting on the mat"
scores = scorer.score(ref, hyp)
for key, val in scores.items():
    print(f"{key}: P={val.precision:.3f} R={val.recall:.3f} "
          f"F={val.fmeasure:.3f}")
```

BERTScore

```python
from bert_score import score as bert_score

refs = ["The cat is on the mat."]
hyps = ["The cat is sitting on the mat."]
P, R, F1 = bert_score(hyps, refs, lang="en")
print(f"BERTScore F1: {F1.mean():.4f}")
```

## 10.10 Exercises

**Exercise 10.1** (Manual BLEU)**.** Manually compute the BLEU-2 (bigram) score for the reference “the small black cat sleeps” and the candidate “the black cat sleeps well”. Detail the computation of $p_1$, $p_2$, and BP.

**Exercise 10.2** (BERTScore vs BLEU)**.** On a corpus of 100 (reference, candidate) pairs containing paraphrases, compare BLEU and BERTScore. Identify cases where the two metrics diverge and explain why.

**Exercise 10.3** (Human Evaluation Protocol)**.** Design a human evaluation protocol for a customer service chatbot. Define criteria, scale, number of annotators, and how to measure inter-annotator agreement.

**Exercise 10.4** (Benchmark Contamination)**.** Explain how test data contamination in an LLM’s training corpus can bias results on MMLU. Propose methods to detect and mitigate this problem.

# Chapter 11

# Ethics, Bias, and Safety

**Responsibility in NLP**

Language models are deployed at massive scale and influence consequential decisions: hiring, justice, healthcare, education. They reproduce and amplify biases present in their training data. Understanding, measuring, and mitigating these biases is a fundamental responsibility of the NLP practitioner.

## 11.1 Bias in Word Embeddings

**Definition 11.1** (Embedding Bias)**.** Word embeddings capture the statistical regularities of the corpus, including social stereotypes. For example, in Word2Vec trained on Google News:

$$\vec{\text{man}} - \vec{\text{woman}} \approx \text{computer }\vec{\text{programmer}} - \vec{\text{homemaker}}$$

**Theorem 11.2** (WEAT — Word Embedding Association Test)**.** *The WEAT test (Caliskan et al., 2017) measures the association between two sets of target words $X, Y$ and two sets of attribute words $A, B$:*

$$d(X, Y, A, B) = \sum_{x \in X} s(x, A, B) - \sum_{y \in Y} s(y, A, B)$$

*where $s(w, A, B) = \frac{1}{|A|} \sum_{a \in A} \cos(w, a) - \frac{1}{|B|} \sum_{b \in B} \cos(w, b)$. Statistical significance is assessed via a permutation test.*

**Example 11.3** (Gender Bias)**.** Applying WEAT with $X = \{\text{math, science, engineering}\}$, $Y = \{\text{arts, literature, humanities}\}$, $A =$ male names, $B =$ female names, reveals a significant association between science and male, reproducing a gender stereotype.

## 11.2 Bias in Language Models

**Definition 11.4** (LLM Bias)**.** LLMs exhibit bias at multiple levels:

1. **Representational**: certain groups are underrepresented or stereotyped.
2. **Allocative**: the model performs better for some groups (e.g., resume summaries favoring certain names).

CHAPTER 11. ETHICS, BIAS, AND SAFETY

3. **Generative**: producing toxic, discriminatory, or inaccurate content about certain groups.

**Proposition 11.5** (Sources of Bias)**.**

- **Training data**: the web contains biased, sexist, and racist content.
- **Annotation**: annotators bring their own cultural and linguistic biases.
- **Training objective**: maximizing likelihood reproduces dominant patterns.
- **Deployment**: feedback biases (users interact more with certain content) amplify existing biases.

## 11.3 Fairness Metrics

**Definition 11.6** (Equalized Odds)**.** A classifier satisfies **equalized odds** if its performance is identical across all demographic groups $g$:

$$P(\hat{Y} = 1|Y = y, G = g) = P(\hat{Y} = 1|Y = y) \quad \forall g, \forall y \in \{0, 1\}$$

**Definition 11.7** (Demographic Parity)**. Demographic parity** requires that the decision be independent of the group:

$$P(\hat{Y} = 1|G = g) = P(\hat{Y} = 1) \quad \forall g$$

**Incompatibility of Fairness Criteria**

It is generally impossible to simultaneously satisfy demographic parity, equalized odds, and calibration (impossibility theorems of Chouldechova, 2017, and Kleinberg et al., 2016). The choice of criterion depends on the application context.

**Definition 11.8** (Toxicity Bias)**. Toxicity bias** measures a model's propensity to generate toxic content (insults, threats, hate speech). The RealToxicityPrompts score (Gehman et al., 2020) measures the probability of generating toxic content from neutral prompts.

## 11.4 Debiasing Techniques

**Definition 11.9** (Embedding Debiasing)**.** The method of Bolukbasi et al. (2016) identifies the bias direction $\mathbf{b}$ (e.g., the man-woman direction) and projects embeddings to remove this component:

$$\mathbf{w}_{\text{debias}} = \mathbf{w} - (\mathbf{w} \cdot \mathbf{b})\mathbf{b}$$

for gender-neutral words (not definitionally gendered).

*Remark* 11.10. Geometric embedding debiasing has been criticized: Gonen and Gold (2019) show that biases persist in neighborhood structure even after projection. More recent approaches intervene at the data or training level.

**Definition 11.11** (LLM Debiasing Strategies)**.**

1. **Data curation**: filtering toxic content, balancing representation.

2. **Instruction tuning**: fine-tuning with explicit fairness instructions.

3. **RLHF/DPO**: training the model to prefer unbiased responses via human feedback.

4. **Prompting**: system instructions specifying equitable behavior.

5. **Controlled decoding**: modifying generation probabilities to reduce toxicity.

## 11.5 Toxicity Detection

**Definition 11.12** (Toxicity Classifier)**.** A **toxicity classifier** is a supervised model trained to detect offensive, hateful, or dangerous content. Examples: Perspective API (Google), OpenAI Moderation API.

**Proposition 11.13** (Challenges of Toxicity Detection)**.**

1. **Subjective definition**: toxicity depends on context, culture, and intent.

2. **Classifier bias**: toxicity models themselves can be biased (e.g., classifying African-American English as more toxic).

3. **Evasion**: users circumvent filters through creative rephrasing.

4. **Over-filtering**: aggressive filtering censors legitimate content.

## 11.6 Environmental Impact

**Definition 11.14** (Carbon Cost of Training)**.** Training large language models consumes substantial energy. Strubell et al. (2019) estimated the carbon footprint of training a Transformer at ∼284 tonnes of $CO_2$ (comparable to 5 times the lifetime emissions of an American car).

**Carbon Footprint Estimation**

$$CO_2 \approx \text{PUE} \times \text{kWh} \times \text{carbon intensity (g CO}_2\text{/kWh)}$$

where PUE (Power Usage Effectiveness) ≈ 1.1–1.4 for modern data centers.

**Proposition 11.15** (Reduction Strategies)**.**

- **Smaller models**: distillation, pruning, quantization.

- **Efficient training**: mixed precision, gradient checkpointing, architecture search.

- **Location**: train in regions with low-carbon electricity.

- **Reuse**: share pretrained models (HuggingFace Hub).

*CHAPTER 11. ETHICS, BIAS, AND SAFETY*

## 11.7 Responsible AI Practices

**Definition 11.16** (Model Cards)**.** A **model card** (Mitchell et al., 2019) documents a model: intended use, limitations, known biases, evaluation metrics, training data, and ethical considerations.

**Definition 11.17** (Datasheets for Datasets)**. Datasheets** (Gebru et al., 2021) document datasets: motivation, composition, collection process, preprocessing, recommended uses, and demographic distributions.

*Remark* 11.18. Regulations are evolving rapidly:

- **EU AI Act** (2024): risk-based classification of AI systems, transparency obligations.
- **US Executive Order on AI** (2023): safety and testing requirements for foundation models.
- Companies adopt *Responsible AI frameworks* integrating risk assessment, regular audits, and ethics committees.

## 11.8 Python Implementation

Bias Detection in Embeddings

```python
import numpy as np
from gensim.models import KeyedVectors

def weat_score(model, X, Y, A, B):
    """Compute WEAT effect size."""
    def s(w, A_set, B_set):
        a_sims = [model.similarity(w, a) for a in A_set
                  if a in model]
        b_sims = [model.similarity(w, b) for b in B_set
                  if b in model]
        return np.mean(a_sims) - np.mean(b_sims)

    x_scores = [s(x, A, B) for x in X if x in model]
    y_scores = [s(y, A, B) for y in Y if y in model]
    d = np.mean(x_scores) - np.mean(y_scores)
    std = np.std(x_scores + y_scores)
    return d / std if std > 0 else 0.0

# Example: gender bias in professions
X = ["programmer", "engineer", "scientist"]
Y = ["nurse", "teacher", "librarian"]
A = ["man", "male", "boy", "he"]
B = ["woman", "female", "girl", "she"]
# score = weat_score(model, X, Y, A, B)
```

**Geometric Debiasing**

```python
def debias_embedding(embedding, bias_direction):
    """Project out the bias direction from an embedding."""
    projection = (np.dot(embedding, bias_direction)
                  * bias_direction)
    return embedding - projection

# Compute bias direction (e.g., he - she)
bias_dir = model["he"] - model["she"]
bias_dir = bias_dir / np.linalg.norm(bias_dir)

# Debias a neutral word
debiased = debias_embedding(model["programmer"], bias_dir)
```

## 11.9 Exercises

**Exercise 11.1** (Bias Analysis)**.** Download pretrained Word2Vec or FastText embeddings. Compute the WEAT score for gender bias in professions. Compare results between embeddings trained on different corpora (Wikipedia vs Common Crawl).

**Exercise 11.2** (Debiasing)**.** Implement the Bolukbasi et al. debiasing method. Verify that biased analogies are reduced. Test whether debiasing affects performance on a semantic similarity task (e.g., STS-B).

**Exercise 11.3** (LLM Toxicity)**.** Use the OpenAI Moderation API (or a HuggingFace classifier) to evaluate the toxicity of 100 generations from a language model using neutral prompts. Which types of toxicity are most frequent?

**Exercise 11.4** (Model Card)**.** Write a complete model card for a sentiment classifier trained on English movie reviews. Include limitations, potential biases, and usage recommendations.

*CHAPTER 11. ETHICS, BIAS, AND SAFETY*

# Bibliography

[1] Jurafsky, D. and Martin, J.H., *Speech and Language Processing*, 3rd ed. draft, 2024.

[2] Vaswani, A. et al., Attention is All You Need, *NeurIPS*, 2017.

[3] Devlin, J. et al., BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, *NAACL*, 2019.

[4] Brown, T.B. et al., Language Models are Few-Shot Learners, *NeurIPS*, 2020.