Chapter 1: Introduction — What Does It Mean to Predict the Next Word?

Learning Objectives

After reading this chapter, the reader should be able to:

1. Explain why next-word prediction is the unifying principle that connects classical NLP, neural language models, and modern large language models.
2. Trace the historical development of language modeling from Shannon's information-theoretic experiments through statistical models to transformer-based LLMs.
3. Describe the four-part structure of the book and map each part to a phase in the evolution of language modeling.
4. Identify the mathematical, programming, and machine learning prerequisites needed for the remaining chapters.

1.1 The Prediction Paradigm

What is the simplest question we can ask about language, and why does answering it well require understanding almost everything about human communication?

Consider the following sentence: "The cat sat on the _____." If we ask a child, a linguist, a search engine, or a state-of-the-art language model to fill in the blank, each will produce a plausible answer — "mat," "floor," "bed," "couch" — and yet the processes underlying those answers differ enormously. The child draws on years of embodied experience, pattern recognition, and world knowledge absorbed from thousands of conversations and stories. The linguist might reason about collocational preferences and the statistical tendencies of English prepositional phrases. A simple statistical model counts how often each word has appeared after "The cat sat on the" in a large corpus of text. A modern neural language model, trained on hundreds of billions of words, computes a probability distribution over its entire vocabulary and selects or samples from the highest-ranked candidates. The prediction task is identical in each case. Only the machinery for solving it changes. This observation — that a single, concrete, measurable objective underlies the full arc of language technology — is the central thesis of this book, and we return to it in every chapter that follows.

1.1.1 What Is a Language Model?

A language model, in the most general and historically stable sense of the term, is any computational system that assigns a probability to a sequence of words. Given a sequence $w_1, w_2, \ldots, w_n$ drawn from a vocabulary V, a language model defines a probability distribution $P(w_1, w_2, \ldots, w_n)$ over all possible sequences of any length. This definition is deceptively simple: it encompasses a hand-crafted lookup table of trigram counts assembled in the 1980s, a 124-million-parameter neural network trained in 2019, and a 175-billion-parameter model that can converse, reason, and write code. All three are language models. They differ only in how they represent and compute the underlying probability distribution, and in how well they approximate the true statistical structure of human language.

A persistent and consequential misconception is that the term "language model" refers specifically

to large neural systems such as GPT or ChatGPT. This conflation obscures the intellectual history of the field and makes it harder to understand why modern systems work as well as they do. The concept of a language model predates deep learning by several decades. Frederick Jelinek and his colleagues at IBM developed n-gram language models for continuous speech recognition in the 1970s and 1980s — systems that assigned probabilities to word sequences using nothing more than frequency counts in text corpora. Those models were language models in precisely the same sense that GPT-3 is a language model: both assign probabilities to sequences of words. Understanding this continuity is essential to appreciating the trajectory of the field. We therefore adopt the broad, historically accurate definition throughout this book: a language model is a system that assigns probabilities to sequences of words, regardless of its internal architecture.

A particularly useful way to think about a language model is through the lens of prediction. Because the probability of a full sequence can always be decomposed — using the chain rule of probability — into a product of conditional probabilities, assigning probabilities to sequences is equivalent to predicting, at each position, the distribution over the next word given everything that came before. Formally, if we write $w_{<t}$ as shorthand for the sequence $(w_1, w_2, \ldots, w_{t-1})$, then any language model defines, for every position t in every sequence, a conditional distribution $P(w_t \mid w_{<t})$. This is the core object of study throughout this book: the probability $P(w_{\text{next}} \mid \text{context})$, which we can read as "the probability that the next word is w, given everything we have seen so far." The smartphone autocomplete feature that suggests words as we type a message is, in this precise sense, a language model: it ranks candidate next words by probability and presents the top three for the user to select. That ranking, however it is computed — by a lookup table, a recurrent network, or a transformer — constitutes a language model.

1.1.2 Prediction as the Common Thread

Why do radically different architectures — from simple count tables to billion-parameter neural networks — all converge on the same mathematical objective?

The unifying insight is that every major architecture in the history of language modeling can be understood as a different answer to the same question: given the context $w_{<t}$, what is $P(w_t \mid w_{<t})$? An n-gram model answers this question by consulting a table of frequencies, restricting the context to the last $n-1$ words and approximating the conditional probability by the relative frequency of the n-gram $w_{t-n+1}, \ldots, w_t$ in a training corpus. A recurrent neural network answers the same question by maintaining a hidden state vector that is updated at each time step, compressing the entire history into a fixed-dimensional representation and projecting it through a learned weight matrix to produce a probability distribution over V. A transformer answers the question by attending directly to all previous tokens in the context, weighting their contributions through a learned similarity function and aggregating the result to produce the next-word distribution. The input to each model is the same context; the output of each model is the same probability distribution; only the computational path between input and output differs.

Figure **??** illustrates this point concretely. Three model families — n-gram, RNN, and Transformer — are shown processing the same input context ("The cat sat on the") and producing distributions over the same next-word vocabulary. The differences are architectural: the n-gram model looks back at most $n-1$ tokens; the RNN summarizes the entire history into a hidden state; the Transformer attends to all previous tokens simultaneously. Despite these architectural differences, all three models are solving the same problem, and all three can be evaluated with the same metric: how well does the predicted probability distribution match the actual next word? This shared objective is

what makes it possible to compare models across architectural generations and to trace a coherent line of progress from the simplest statistical models to the most sophisticated neural systems.

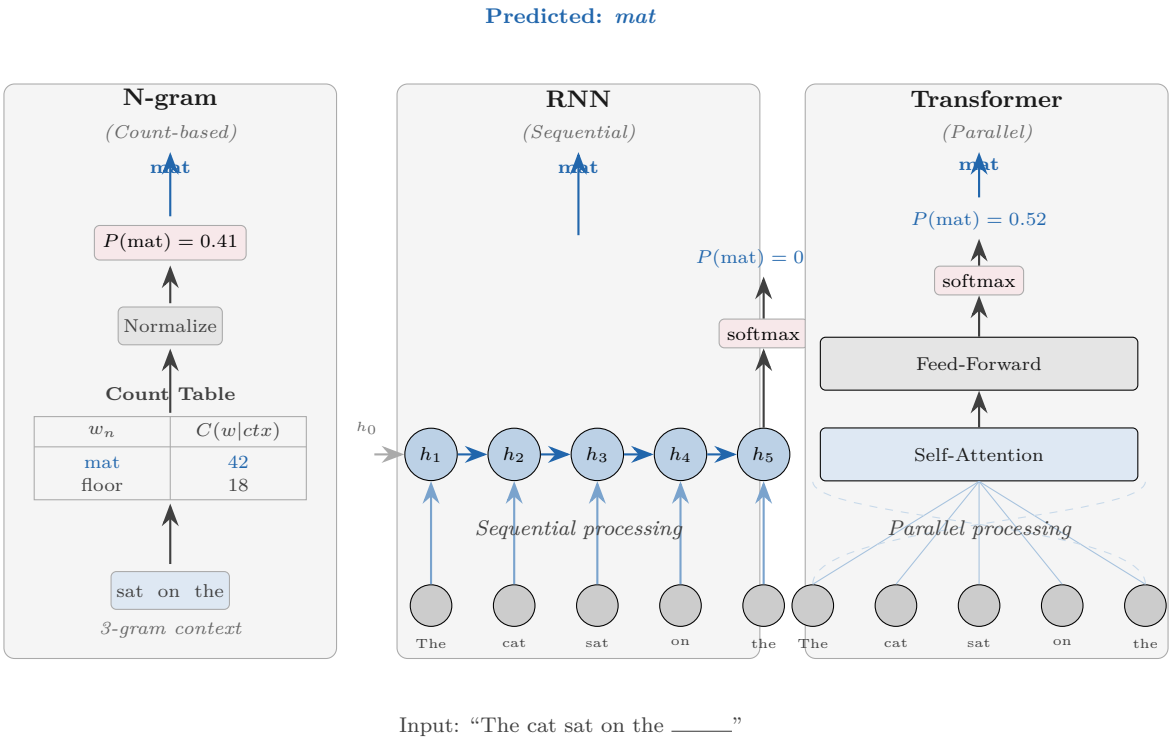

Figure 1: Figure 1.1 – Three Model Architectures for Next-Word Prediction

The thermometer analogy is illuminating here. Mercury thermometers, electronic thermometers, and infrared thermometers are all thermometers: they all measure temperature, they all return a reading in degrees, and they can all be evaluated against a ground-truth temperature standard. They differ in their sensing mechanisms, their accuracy, their speed, and their range of applicability. The progression from mercury to electronic to infrared did not change what a thermometer does; it changed how well and in what circumstances it does it. The analogy breaks down in one respect: thermometers measure a physical quantity that exists independently, while language models estimate a probability distribution that is, in some sense, defined by the model itself. Nevertheless, the structural parallel holds for language models. The progression from n-grams to neural networks to transformers did not change what a language model does — it has always been estimating $P(w_t \mid w_{<t})$ — but it dramatically improved how accurately and in what contexts that estimation can be performed.

Code Example 2 makes this abstract discussion concrete. By loading a pre-trained GPT-2 model and running a single forward pass on the prompt "The cat sat on the," we can examine the probability distribution over all 50,257 tokens in GPT-2's vocabulary and inspect the five highest-ranked candidates.

```
import torch
from transformers import GPT2LMHeadModel, GPT2Tokenizer
```

```
# Load pre-trained GPT-2 and its tokenizer
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
model = GPT2LMHeadModel.from_pretrained("gpt2")

# Encode the prompt and get the model's predictions
prompt = "The cat sat on the"
inputs = tokenizer(prompt, return_tensors="pt")
with torch.no_grad():
    logits = model(**inputs).logits

# Extract probabilities for the next token
next_token_probs = torch.softmax(logits[0, -1, :], dim=0)
top5 = torch.topk(next_token_probs, 5)

# Display the top 5 predicted next words with their probabilities
for prob, idx in zip(top5.values, top5.indices):
    token = tokenizer.decode(idx)
    print(f"  {token.strip():>12s}  {prob.item():.4f}")
```

Running this code produces a ranked list in which "floor," "mat," "bed," "couch," and "table" receive the highest probabilities, with the exact ranking depending on the model version. What we observe is not a single answer but a probability distribution: the model does not claim that the next word *is* "floor"; it claims that "floor" is somewhat more probable than "mat," which is somewhat more probable than "couch." This distributional perspective is fundamental to everything that follows. A language model is not a function that maps contexts to words; it is a function that maps contexts to probability distributions over words. All of the evaluation metrics, training objectives, and generation strategies discussed in later chapters flow from this basic framing.

1.1.3 From Probabilities to Applications

If a language model is just a system for assigning probabilities to sequences, why does it matter so much — and how does a single mathematical object give rise to such a wide range of applications?

The remarkable fact about the prediction objective is how many apparently different NLP tasks it subsumes. Text generation is the most direct application: we sample from $P(w_t \mid w_{<t})$ repeatedly, feeding each predicted token back into the context and sampling again, until we have produced a sequence of the desired length. The result — as the following code example demonstrates — is fluent, coherent text that continues naturally from any starting prompt.

```
from transformers import pipeline

# Load a pre-trained language model for text generation
generator = pipeline("text-generation", model="gpt2")

# Generate text by predicting one token at a time
prompt = "The future of artificial intelligence"
output = generator(prompt, max_new_tokens=50, do_sample=True, temperature=0.8)
```

```
# The model generates fluent text solely by predicting the next token repeatedly
print(output[0]["generated_text"])
```

Machine translation, which might appear to be a fundamentally different problem — transforming a sentence from one language to another — can also be framed as next-word prediction. We simply condition on the source sentence in addition to the previously generated target words, predicting $P(y_t \mid y_{<t}, x_{1:n})$ where $y_{1:T}$ is the target sequence and $x_{1:n}$ is the source sequence. Every word the model generates is the most probable next word in the target language given both the source and the partial translation produced so far. Automatic speech recognition follows the same pattern: the model predicts the most probable word sequence given the acoustic signal, which amounts to finding the sequence that maximizes $P(w_{1:n})$ times the acoustic likelihood — a classical application of the noisy channel model that we discuss further in Section 1.2.1. Text summarization condenses a long document into a short one by conditioning the next-word predictions on the source document, and dialogue systems respond to conversational history by treating the entire conversation as context for the next utterance.

Students often think that text generation and text classification are fundamentally different tasks. In the large language model setting, this distinction dissolves. Classifying a movie review as "positive" or "negative" can be accomplished by a language model that predicts whether the token "positive" or the token "negative" is more probable given the review text as context. Answering a factual question such as "What is the capital of France?" requires the model to predict that "Paris" is the most probable next word given the question. Every NLP task, under this framing, reduces to computing $P(w_{\text{next}} \mid \text{context})$ for an appropriately defined context. This does not mean all tasks are equally easy — the context may be very long, the vocabulary may contain many plausible candidates, and the probability distribution may be highly uncertain — but it does mean that a sufficiently capable language model is, in principle, a general-purpose NLP system.

The analogy to chess is instructive. A chess engine that can accurately predict the highest-probability next move — given the current board position and all moves played so far — can, by applying that capability repeatedly, play entire games, analyze arbitrary positions, generate opening theory, and evaluate endgame scenarios. From a single predictive capability, an enormous range of chess-related tasks become tractable. Language modeling works the same way. From the single capability of estimating $P(w_{\text{next}} \mid \text{context})$, text generation, translation, summarization, question answering, code completion, and dialogue all become instances of the same computational task. This is why next-word prediction is not merely a convenient framing device but the genuine mathematical foundation of modern NLP. The prediction framing is so general that one might worry it explains everything and therefore nothing — but the empirical evidence suggests otherwise. We have organized this entire book around this insight: every chapter presents a different model or technique, but every model and technique is ultimately evaluated on how well it estimates $P(w_{\text{next}} \mid \text{context})$, and improvements in that estimation are what drive progress across all NLP applications.

Section 1.1 has introduced the language modeling objective $P(w_{\text{next}} \mid w_{<t})$, clarified that this definition encompasses models from simple n-gram tables to large neural systems, and shown how a wide range of NLP applications reduce to next-word prediction under appropriate conditioning. We now trace how this objective has been pursued across seven decades of research, from Shannon's information-theoretic foundations to the large language models of the present day.

1.2 A Brief History of Language Modeling

How did a theoretical question about the statistical structure of language — posed in the late 1940s — give rise to one of the most transformative technologies of the early twenty-first century?

The history of language modeling is not a history of disconnected ideas. It is a single continuous thread of inquiry into the question that Section 1.1 identified as the core of the field: how do we estimate $P(w_t \mid w_{<t})$? Each generation of researchers inherited the answer of the previous generation, identified its principal limitation, and developed a new approach to overcome it. Understanding this progression is not merely historical interest; it explains why the field looks the way it does today and prepares the reader for the detailed treatment of each approach in later chapters.

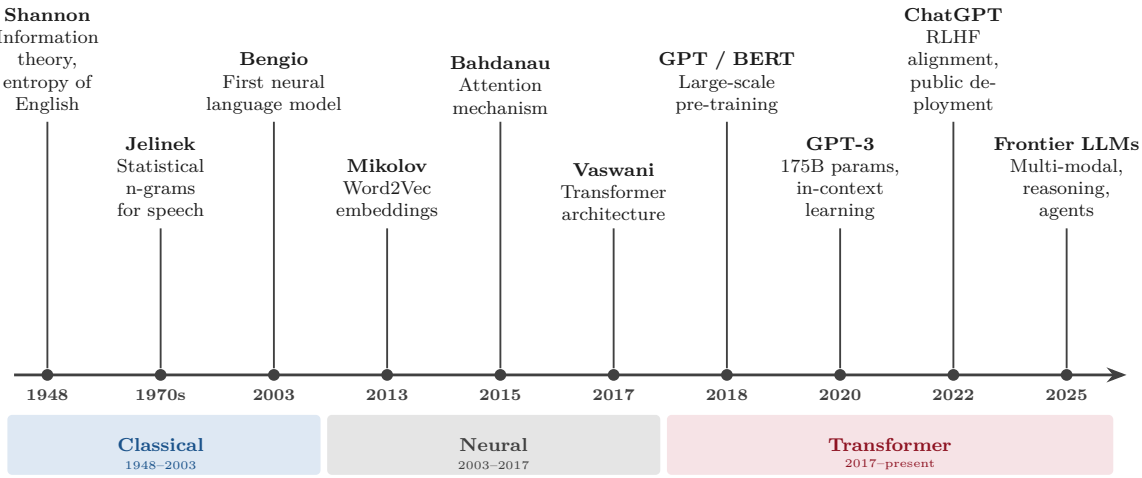

Figure 2: Figure 1.2 – Seven Decades of Language Modeling Evolution

1.2.1 Shannon and Information Theory

Claude Shannon published "A Mathematical Theory of Communication" in the Bell System Technical Journal in 1948, working at Bell Labs on the problem of reliable transmission over noisy telephone wires. In founding the field of information theory, he did something that he may not have fully anticipated: he defined the language modeling problem [Shannon, 1948]. The paper's central contribution was a precise definition of information: the information content of an event is inversely proportional to its probability, and the average information content of a random variable — its entropy — is $H(P) = -\sum_x P(x) \log_2 P(x)$. Shannon applied this framework to the problem of communication over noisy channels, proving fundamental theorems about the limits of reliable communication. But embedded in the same paper was a question about language: how much information does each symbol in a message actually carry? If English text were perfectly random, each letter would carry $\log_2 27 \approx 4.75$ bits (for 26 letters plus space). Shannon suspected — correctly — that English was far from random: letters are constrained by spelling, words are constrained by grammar, and sentences are constrained by discourse. The actual entropy of English should therefore be far below the theoretical maximum.

To estimate this entropy, Shannon designed an experiment, described in his 1951 follow-up paper, that is worth recounting in detail because it establishes a direct and explicit connection between

information theory and language modeling [Shannon, 1951]. He presented human subjects with printed text from a biography of Thomas Jefferson and asked them to predict the next character one at a time, making repeated guesses until they guessed correctly. When the subject guessed correctly on the first attempt, that character was highly predictable — it carried little information. When it required many guesses, the character was surprising — it carried more information. By averaging the number of guesses required across many characters, Shannon was able to place upper and lower bounds on the true entropy of printed English. His estimates ranged from 0.6 to 1.5 bits per character, with the most reliable estimates around 1.0 to 1.3 bits per character. This was far below the theoretical maximum of 4.75 bits, confirming that English is massively redundant and highly predictable. The experiment was, in retrospect, the first empirical language modeling study: Shannon's human subjects were acting as language models, and their guessing accuracy was a measure of how well a language model could predict English text.

Sidebar 1: Shannon's Guessing Game

In 1951, Claude Shannon devised an experiment to estimate the entropy of printed English. He presented human subjects with text from a Jefferson biography, asking them to predict the next character one at a time. When they guessed correctly on the first try, that character carried little information — it was highly predictable. When multiple guesses were required, the character was surprising — it carried more information. By averaging the number of guesses over many characters and applying information-theoretic bounds, Shannon estimated that English carries between 0.6 and 1.5 bits per character, far below the theoretical maximum of 4.75 bits for 27 equiprobable symbols. This demonstrated that human language is massively redundant and highly predictable, establishing the fundamental connection between prediction ability and information content that motivates the entire field of language modeling. Every metric we use to evaluate language models in this book — perplexity, cross-entropy, bits per character — descends directly from Shannon's guessing game. Source: [Shannon, 1951].

The significance of Shannon's work for language modeling cannot be overstated. Shannon's framework established three foundational ideas that persist through every subsequent chapter. First, it established that the quality of a language model can be measured objectively: a model that assigns higher probability to text that actually occurs is a better model, and this quality can be quantified in bits. Second, it established that human language has deep statistical regularities that a sufficiently sophisticated model should be able to capture. Third, it established a conceptual connection between the source model in a communication system — the statistical process that generates the text — and the predictive model that a receiver uses to decode it. In Shannon's framework, a language model is the receiver's best guess about the source. Every language model trained in every chapter of this book is, at some level of abstraction, an attempt to build a better receiver for the Shannon source that is human language.

It is tempting to assume that information theory is primarily a discipline about electronics and communication engineering, with only tenuous connections to natural language. Shannon himself was deeply interested in the statistical structure of English text and devoted significant effort to understanding it. The connection between information theory and language is not incidental but constitutive: the mathematical definition of a language model — a probability distribution over sequences — is an information-theoretic object, and the evaluation metrics we use to compare models are information-theoretic measures. We develop these connections formally in Chapter 2.

1.2.2 Statistical Language Models: The n-gram Era

How did a practical engineering problem at IBM and Bell Labs transform the theoretical insights of information theory into the first generation of working language models?

The path from Shannon's theoretical framework to the first practical language models ran through the speech recognition laboratories of IBM and AT&T Bell Labs in the 1970s and 1980s. The driving application was continuous speech recognition: given an acoustic signal, find the most probable word sequence. Frederick Jelinek and his colleagues at IBM recognized that this problem could be decomposed into two components — an acoustic model that estimates the probability of acoustic observations given a word, and a language model that estimates the prior probability of word sequences [Jelinek, 1976]. The language model component was exactly Shannon's source model: a distribution over sequences of words that could be estimated from large text corpora and used to prefer linguistically plausible transcriptions over phonetically similar but ungrammatical alternatives.

The practical solution Jelinek's group developed was the n-gram model. The key insight was a simplifying assumption: instead of conditioning on the entire history of words up to position t, approximate the conditional probability $P(w_t \mid w_{<t})$ by conditioning only on the last $n-1$ words. This is the Markov assumption, applied to language: the current word depends only on the most recent context. For a trigram model ($n = 3$), we approximate $P(w_t \mid w_{<t}) \approx P(w_t \mid w_{t-2}, w_{t-1})$, and we estimate this conditional probability by counting how often the trigram w_{t-2}, w_{t-1}, w_t appears in a training corpus and dividing by the count of the bigram w_{t-2}, w_{t-1}. The result is a simple, interpretable, and computationally efficient model that can be trained on any sufficiently large text collection. The training process requires nothing more than counting and dividing.

n-gram models proved remarkably effective for their era. Properly tuned trigram models, with sophisticated smoothing techniques such as Kneser-Ney interpolation to handle unseen n-grams, powered Google Translate's first decade of operation, underpinned every major commercial speech recognition system through the 2000s, and remain competitive for many tasks even today. It would be a serious mistake — one we explicitly caution against — to dismiss n-gram models as "trivial" or "outdated." They remain pedagogically essential because they embody, in their simplest possible form, the core ideas of corpus-based probability estimation, the Markov assumption, and the trade-off between model complexity and data sparsity. Every neural model in this book is, in one sense, an attempt to overcome the limitations of the n-gram approximation: the restriction to a fixed context window of $n-1$ words, the inability to generalize across semantically similar words ("dog" and "puppy" appear in completely different n-gram entries), and the data sparsity that makes reliable estimation of long n-grams impractical. Understanding these limitations is the motivation for everything that follows in Parts II and III. We provide a thorough treatment of n-gram models, including their estimation, smoothing, and evaluation, in Chapter 3.

1.2.3 The Neural Turn

What limitation of n-gram models motivated the development of neural language models, and how did Yoshua Bengio's 2003 paper change the trajectory of the entire field?

The fundamental limitation of n-gram models is what Bengio and colleagues identified as the *curse of dimensionality* in language modeling: the number of possible word sequences grows exponentially with sequence length, and any language model that represents each sequence explicitly will face catastrophic data sparsity [Bengio et al., 2003]. More concretely, an n-gram model has no mechanism

for generalization: it cannot infer that the sentence "The dog sat on the mat" is similar to "The puppy sat on the mat," because it has no representation of the semantic similarity between "dog" and "puppy." If the trigram "puppy sat on" was never seen in training, the model assigns it zero probability, regardless of how many times "dog sat on" appeared. This zero-probability problem was the central obstacle to n-gram model quality, and despite sophisticated smoothing methods, it remained an intrinsic limitation of the count-based approach.

Bengio, Ducharme, Vincent, and Jauvin proposed a radical solution in their 2003 paper "A Neural Probabilistic Language Model" [Bengio et al., 2003]. Instead of representing words as atomic, discrete symbols, they proposed learning a *distributed representation* — a real-valued vector — for each word in the vocabulary. Similar words would be mapped to nearby vectors in this embedding space, enabling the model to generalize: what the model learns about "dog" transfers automatically to "puppy" because the two words are close in embedding space. The model architecture consisted of an embedding lookup that mapped each context word to its vector, a feedforward network that processed the concatenated embeddings, and a softmax output layer that produced a probability distribution over the vocabulary. The entire system — embeddings and network weights — was trained jointly to minimize the negative log-likelihood of the training corpus. The paper demonstrated that this neural language model could outperform smoothed n-gram models on the Brown corpus benchmark, establishing for the first time that neural representations could benefit language modeling. We recall the excitement when Bengio's results first appeared — the idea that a neural network could learn word similarity and use it for prediction was not new in principle, but seeing it work in practice changed the conversation.

The decade following Bengio's paper saw rapid development along two lines. The first was the development of *word embeddings* as standalone representations. Mikolov and colleagues introduced Word2Vec in 2013, a highly efficient method for learning word embeddings from raw text using a simplified prediction objective [Mikolov et al., 2013]. Word2Vec embeddings captured striking semantic regularities — the famous "king $-$ man $+$ woman $\approx$ queen" arithmetic — and became one of the most widely adopted tools in NLP. We address an important misconception here: Word2Vec is not itself a language model. It is a method for learning word representations that can serve as inputs to language models. The language model is the system that uses those representations, along with a sequence model, to predict the next word. The second line of development was the application of recurrent neural networks (RNNs) and Long Short-Term Memory networks (LSTMs) to the language modeling task. Unlike feedforward models, which required a fixed-length context window, recurrent models maintained a hidden state vector that was updated at each time step, theoretically allowing the model to condition on arbitrarily long histories. The combination of learned embeddings and recurrent processing produced language models that substantially outperformed n-gram baselines on standard benchmarks, and by the mid-2010s, LSTM-based language models had become the dominant paradigm in NLP research.

1.2.4 Attention and the Transformer

Why did recurrent neural networks, despite their theoretical ability to model long-range dependencies, struggle in practice — and what architectural insight resolved this limitation?

Recurrent neural networks face a structural challenge that becomes acute as sequences grow longer: the hidden state at any time step must simultaneously serve as the model's memory of everything that occurred earlier and as the input representation for the current prediction. Because this state has a fixed dimensionality — typically 512 or 1024 in practical models — it inevitably loses

information about earlier parts of the sequence as new inputs arrive. For short sequences this limitation is manageable. For sequences spanning hundreds of words, the hidden state at the end of the sequence may retain little trace of information from the beginning. The technical manifestation of this problem is the vanishing gradient: when training RNNs through backpropagation over long sequences, the gradient signal that propagates back to earlier time steps diminishes exponentially, making it difficult for the model to learn dependencies between tokens that are far apart. LSTMs partially ameliorated this problem through gated memory cells, but the fundamental bottleneck of a fixed-dimensional state vector remained.

Bahdanau, Cho, and Bengio proposed a direct solution in their 2015 paper on neural machine translation: an *attention mechanism* that allows the model to look back at all encoder hidden states when generating each decoder output [Bahdanau et al., 2015]. Rather than compressing the entire source sequence into a single fixed-dimensional vector, the attention mechanism computes a weighted combination of all encoder states at each decoding step, with the weights determined by the similarity between the current decoder state and each encoder state. This "soft" alignment allowed the model to focus on the most relevant parts of the source when generating each target word, directly addressing the bottleneck of fixed-size representations. The attention mechanism was immediately successful, improving translation quality substantially and introducing a new inductive bias — that generating a word should involve attending to the relevant parts of the input — that proved broadly applicable.

The Transformer architecture, introduced by Vaswani and colleagues in the landmark paper "Attention Is All You Need" in 2017, extended this insight radically: why not build an entire sequence model from attention, without any recurrence at all [Vaswani et al., 2017]? The Transformer replaced the sequential, step-by-step processing of RNNs with *self-attention*: at each layer, every token attends to every other token in the sequence, computing representations that incorporate information from arbitrary positions simultaneously. This architectural choice had two major consequences. First, it eliminated the vanishing gradient problem for long-range dependencies, because information from any position could flow directly to any other position in a single layer. Second, and equally important, it enabled full parallelization over the sequence length during training, because all attention computations can be performed simultaneously rather than sequentially. On modern graphics processing units (GPUs), this parallelization translated to dramatically faster training, enabling the field to scale to training sets and model sizes that were impractical with recurrent architectures. The Transformer became, within a year of its introduction, the dominant architecture for virtually every NLP task. We provide a complete technical exposition of the Transformer in Chapter 8.

You might expect that the key innovation of the Transformer was the attention mechanism itself, which had already appeared in earlier work. The key innovation was the demonstration that self-attention alone — without any recurrence or convolution — is sufficient to build a high-quality sequence model, and that this purely attentional architecture can be scaled to produce models of previously impossible quality. Recurrence was not merely reduced; it was eliminated entirely, and the resulting architecture could be trained orders of magnitude faster.

1.2.5 The Large Language Model Era

The Transformer's training efficiency made scaling possible in a way that recurrent architectures could not support. The first sign of what scale could produce came from GPT-2, introduced by Radford and colleagues at OpenAI in 2019 [Radford et al., 2019]. With 1.5 billion parameters and

training on 40 gigabytes of web text, GPT-2 generated text of a quality that surprised even its creators: coherent multi-paragraph essays, plausible news articles, and stylistically varied prose that, at first glance, was difficult to distinguish from human writing. The same year, Devlin and colleagues at Google introduced BERT (Bidirectional Encoder Representations from Transformers), which used a masked language modeling objective — predicting randomly masked tokens using bidirectional context — to produce representations that, after fine-tuning, achieved state of the art on virtually every NLP benchmark [Devlin et al., 2019].

In 2020, Brown et al. demonstrated what scale alone could produce: GPT-3, with 175 billion parameters trained on hundreds of billions of tokens, exhibited a phenomenon that the field had not anticipated [Brown et al., 2020]. The few-shot learning demonstrations were, to put it mildly, unexpected. GPT-3 demonstrated *in-context learning*. Given a few examples of a task in the prompt — without any modification of the model's parameters — GPT-3 could perform that task on new examples with competitive accuracy. The model appeared to "learn" from the prompt examples at inference time, even though no gradient updates were made. This capability, which emerged from scale rather than from any deliberate architectural or training innovation, suggested that very large language models develop general-purpose reasoning capabilities that extend far beyond simple pattern matching. Scaling curves across dozens of benchmarks showed smooth, consistent improvement with model size, data volume, and training compute, with no clear saturation point in sight.

The period from 2022 to the present has seen the emergence of *large language models* that combine the scaling insights of GPT-3 with instruction tuning and human feedback. ChatGPT (2022) demonstrated that a GPT-class language model, trained with reinforcement learning from human feedback (RLHF) to follow instructions helpfully and safely, could serve as a general-purpose conversational assistant capable of writing, reasoning, coding, and answering questions across virtually every domain. Subsequent frontier models — GPT-4, Claude, Gemini, LLaMA, and their successors — have pushed capabilities further still, incorporating multimodal inputs, extended context windows, improved reasoning, and tool use. We examine the technical underpinnings of instruction tuning and RLHF in Chapter 12.

Sidebar 2: The Unreasonable Effectiveness of Language Models

Training a model to predict the next token in internet text — with no explicit programming of grammar, logical rules, or world knowledge — produces a system that can write essays, solve mathematics problems, translate between dozens of languages, and generate working code in multiple programming languages. This phenomenon, which echoes Wigner's famous observation about the "unreasonable effectiveness of mathematics" in physics, suggests that next-word prediction is a far richer learning signal than it first appears. To predict the next word accurately across diverse text, a model must implicitly learn spelling and morphology (from predictability within words), syntax (from the grammatical constraints on word order), semantics (from the collocational and topical patterns that govern word choice), common-sense reasoning (from the inferential patterns that govern plausible continuations), and domain-specific factual knowledge (from the statistical regularities of specialized text). The observation raises deep questions about the relationship between prediction and understanding that the field has not yet resolved, and it provides the philosophical motivation for organizing an entire textbook around the prediction paradigm. Sources: [Radford et al., 2019], [Brown et al., 2020].

We note an important distinction that this book maintains throughout. Large language models are,

at their core, next-word predictors: systems trained to minimize the negative log-likelihood of a text corpus, which is exactly the language modeling objective we defined in Section 1.1. Their emergent capabilities — reasoning, in-context learning, instruction following — arise from the interaction of scale, data diversity, and the richness of the prediction objective. Whether these capabilities constitute genuine "understanding" in a philosophically meaningful sense is an open question that we return to briefly in Chapter 15. We do not pretend to have resolved this question, and the honest answer is that no one has. The book's position is agnostic: we can study and build language models rigorously without resolving the philosophical question, just as we can study and build thermometers without resolving the metaphysical question of what temperature really is. What matters for the purpose of this book is that next-word prediction, pursued to sufficient scale and sophistication, produces systems of extraordinary capability, and that understanding those systems requires understanding the prediction objective at their foundation.

Section 1.2 has traced the evolution of language modeling from Shannon's information-theoretic foundations through the n-gram era, the neural turn, the Transformer revolution, and the large language model era. This historical arc — from simple count tables to systems trained on trillions of tokens — is the narrative that organizes the four parts of this book, which we describe in the next section.

1.3 How This Book Is Organized

How does the conceptual dependency among the key ideas in language modeling determine the order in which we should learn them?

The organization of this book follows the conceptual dependency structure of language modeling rather than strict chronological order. Each part builds on the foundations laid by the previous part, and the transition between parts corresponds to a fundamental change in the type of model being studied. We have spent considerable time debating the ordering of these chapters, and the current sequence reflects hard-won convictions about what must come before what. Whether we have chosen the right ordering is ultimately for the reader to judge. We have designed the book to support multiple reading strategies: a linear first-to-last reading provides the most complete understanding, but several alternative paths are described in Section 1.3.5 for readers with specific backgrounds or goals.

1.3.1 Part I: Foundations (Chapters 1–3)

Why do we need three chapters of mathematical and conceptual groundwork before we can study a neural language model?

Part I establishes the mathematical machinery and the classical algorithmic baseline without which no later chapter can be fully understood. Chapter 1 — the present chapter — provides the conceptual framing and motivates the prediction paradigm as the book's organizing principle. Chapter 2 develops the probability theory, information theory, and optimization foundations that pervade the entire book: the chain rule decomposition of sequence probability, entropy and cross-entropy as measures of model quality, perplexity as the standard language model evaluation metric, and the gradient-based optimization methods used to train all neural models. Chapter 3 implements the first concrete language model — the n-gram model — using the mathematical tools of Chapter 2, providing a performance baseline against which every neural model in later chapters is compared.

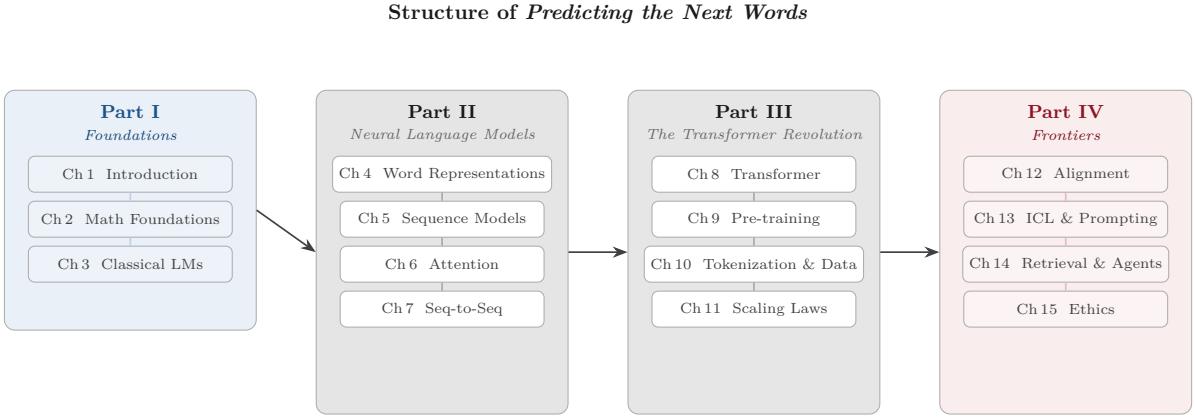

Figure 3: Figure 1.3 – Structure of "Predicting the Next Words" by Part and Chapter

A common impulse among students approaching this book is to skip directly to Part III, where the most modern and exciting models live. We strongly discourage this. Perplexity, cross-entropy, and the chain rule decomposition are not historical curiosities; they are used in every chapter that follows. The evaluation of every neural language model — whether a simple feedforward network, an LSTM, or a 175-billion-parameter transformer — uses exactly the metrics introduced in Chapter 2. The n-gram model of Chapter 3 is not merely a historical artifact; it is the performance floor against which neural improvements are measured, and it remains competitive for many practical applications. Understanding Part I is a prerequisite for appreciating what Parts II, III, and IV actually accomplish. The structural engineering analogy applies: before building a skyscraper, one must understand load-bearing principles and traditional construction. The sophistication of the later chapters depends entirely on the solidity of the foundation laid here.

1.3.2 Part II: Neural Language Models (Chapters 4–7)

How does the field progress from static word representations to sequential models to attention-based architectures — and why does each step follow necessarily from the limitations of its predecessor?

Part II traces the development of neural language models from the first distributed word representations through recurrent architectures and attention mechanisms to sequence-to-sequence models for conditional generation. Chapter 4 introduces word embeddings: dense vector representations of words learned from context that capture semantic and syntactic regularities. We explain the Word2Vec training objective, the geometry of embedding spaces, and the practical considerations involved in using pre-trained embeddings. Chapter 5 introduces recurrent neural networks and LSTMs as sequence models for language, covering the forward and backward passes, the vanishing gradient problem, and the practical training of deep recurrent language models on standard benchmarks. Chapter 6 introduces the attention mechanism, explaining how learned soft alignments between sequence positions overcome the bottleneck of fixed-dimensional hidden states. Chapter 7 integrates these components into encoder-decoder architectures for conditional generation tasks such as machine translation and summarization, covering teacher forcing, beam search, and the evaluation of generation quality with BLEU and related metrics.

Part II presents these architectures in the order in which they were historically developed, and this

order is not coincidental. Each chapter presents an architecture that solves a specific limitation of the architecture from the previous chapter: neural embeddings solve the generalization failure of n-grams, recurrent models extend the context window beyond the fixed $n-1$ tokens of n-gram models, attention addresses the fixed-capacity bottleneck of recurrent hidden states, and encoder-decoder architectures extend the conditional generation framework to tasks where input and output sequences have different lengths. Students who understand why each chapter's architecture improves on the previous one will be well prepared to understand the Transformer of Chapter 8, which is best viewed as the culmination of this design trajectory rather than as a discontinuous breakthrough. A natural but incorrect intuition is that Part II is "outdated" because transformers have superseded RNNs; this should be firmly resisted. Understanding attention as a separate concept from the full Transformer architecture, and understanding why recurrent processing is useful and where it falls short, is necessary for a deep understanding of why the Transformer works as well as it does.

1.3.3 Part III: The Transformer Revolution (Chapters 8–11)

What technical insights enable a single architecture to scale from millions to hundreds of billions of parameters and achieve state-of-the-art performance across every NLP task?

Part III presents the Transformer architecture and the pre-training paradigm that transformed language modeling from a specialized technique into a general-purpose approach to NLP. Chapter 8 provides a thorough, component-by-component exposition of the Transformer: multi-head self-attention, positional encodings, layer normalization, feedforward sublayers, and the residual connections that make very deep Transformers trainable. We derive the attention mechanism from first principles and explain the computational and memory complexity of each component. Chapter 9 covers pre-training and fine-tuning: the causal language modeling objective used by GPT-family models, the masked language modeling objective used by BERT-family models, and the practical considerations involved in adapting a pre-trained model to a downstream task through fine-tuning or parameter-efficient methods such as LoRA. Chapter 10 addresses subword tokenization — the Byte Pair Encoding and SentencePiece algorithms that segment text into subword units — which is a technical prerequisite that is often underappreciated but has important consequences for model vocabulary, handling of rare words, and multilingual generalization. Chapter 11 discusses scaling laws: empirical relationships between model size, training data volume, compute budget, and final model quality that govern the design of large language model training runs and provide a principled basis for resource allocation.

A natural but incorrect intuition is that the Transformer is a monolithic black box whose internal workings are inaccessible to analysis. We systematically decompose the architecture into understandable components and provide mathematical derivations and intuitions for each one. By the end of Part III, the reader should be able to read and understand the source code of a Transformer implementation, trace the flow of information through the architecture from input tokens to output logits, and reason about why specific design choices — such as the scaling factor $1/\sqrt{d_k}$ in the attention dot product — matter for training stability and performance.

1.3.4 Part IV: Frontiers (Chapters 12–15)

What does it take to turn a language model that predicts the next token into a system that is helpful, safe, and capable of acting in the world?

Part IV addresses the techniques and considerations that transform a pre-trained language model into a deployed AI system. Chapter 12 covers alignment and instruction tuning: the supervised

fine-tuning on human demonstrations and reinforcement learning from human feedback (RLHF) that convert a next-token predictor into a conversational assistant that follows instructions, avoids harmful outputs, and produces responses that users find helpful and informative. We treat RLHF as a constrained optimization problem and explain the reward model, policy gradient methods, and the Kullback-Leibler divergence penalty that prevents the fine-tuned model from diverging too far from the pre-trained baseline. Chapter 13 covers in-context learning and prompting: the phenomenon by which large language models can perform new tasks from examples in the prompt without parameter updates, and the engineering discipline of prompt design that has emerged around this capability. Chapter 14 introduces retrieval-augmented generation (RAG) and language model agents: systems that extend the language model's capabilities by grounding generation in retrieved documents and allowing the model to take actions — executing code, querying databases, browsing the web — in pursuit of user-specified goals. Chapter 15 closes the book with a discussion of safety, bias, and societal impact: the technical and ethical challenges of deploying language models responsibly, including the measurement and mitigation of demographic biases, the detection of generated text, and the open questions about long-term societal consequences.

A critical misconception that Part IV addresses directly is that alignment is optional — a "nice to have" that can be added after the core model is developed. The chapter sequence of Part IV reflects the view that alignment, safety, and ethics are not afterthoughts but integral components of the language model development process. A language model that predicts the next token accurately but that produces harmful, biased, or factually incorrect outputs when deployed is not a successful language model; it is a failure mode. Understanding why alignment is hard, and what current techniques can and cannot accomplish, is as important for a complete understanding of the field as understanding the transformer architecture itself.

1.3.5 Suggested Reading Paths

Not all readers need to follow the same path. We recognize that readers bring different backgrounds and goals to this book and have designed it to support several reading strategies. The *complete linear path* — reading all fifteen chapters in order — is recommended for students who are approaching language modeling as a new subject and want the most thorough preparation. The chapter ordering follows conceptual dependencies, and each chapter assumes familiarity with the chapters that precede it. This is the path we recommend for readers using the book as a primary course text.

The *foundations-first path* is appropriate for readers with strong ML backgrounds who want to understand language models from mathematical first principles before engaging with modern architectures. This path covers Chapters 1 through 3 thoroughly, then proceeds directly to Chapters 8 through 11 (Part III), using Part II (Chapters 4 through 7) as optional supplementary reading on the historical development of neural architectures. The *applied-first path* is designed for practitioners who want to use pre-trained language models effectively and understand how to fine-tune, prompt, and deploy them, but who do not need a full derivation of each architectural component. This path covers Chapters 1, 8, 9, 12, 13, and 14, with Chapter 2 recommended as a reference for metric definitions. Finally, the *theory-deep path* is for readers focused on the theoretical aspects of scaling, evaluation, and the information-theoretic foundations of language modeling. This path prioritizes Chapters 1, 2, 3, 8, 11, and 15, treating the intermediate architectural chapters as background reading. Regardless of reading path, Chapter 2 (mathematical foundations) and Chapter 8 (the Transformer) are the two most technically dense chapters and merit close attention from all readers.

Section 1.3 has described the four-part structure of the book and the conceptual rationale for the

chapter ordering. Before proceeding to the first technical chapter, we summarize the mathematical and programming background that the book assumes, and introduce the notation conventions that we use throughout.

1.4 Prerequisites and Notation

What does a reader need to know before engaging with this book, and how can we ensure that notation never becomes an obstacle to understanding?

Every technical book faces a tension between self-containedness and tractability: the more prerequisites we review, the more time readers spend on material they may already know; the fewer we review, the more readers without that background will struggle. We resolve this tension by being explicit about prerequisites, pointing readers to suitable resources for topics we do not cover, and developing within the book all of the specialized knowledge — information theory, subword tokenization, attention mechanisms — that is not typically covered in standard undergraduate curricula.

1.4.1 Mathematical Prerequisites

The book requires solid undergraduate-level competence in three mathematical domains: linear algebra, probability theory, and calculus. In linear algebra, the reader should be comfortable with vectors and matrices as mathematical objects, matrix-vector multiplication, the geometric interpretation of dot products and cosine similarity, and the concepts of rank, invertibility, and eigendecomposition at a conceptual level. Vectors are ubiquitous in neural language models: words are represented as vectors, hidden states are vectors, attention weights are vectors, and the transformation between layers is a matrix-vector product. A reader who does not understand why the dot product $\mathbf{x}^\top \mathbf{y}$ measures the similarity between vectors $\mathbf{x}$ and $\mathbf{y}$ will find Chapter 8's treatment of self-attention opaque. In our experience, the most common reason students struggle with later chapters is not insufficient mathematics but insufficient comfort with the notation — which is why we take the time to establish it carefully here.

In probability theory, the reader should understand probability distributions, conditional probability, Bayes' theorem, expectations, and the distinction between probability mass functions (discrete distributions over vocabularies) and probability density functions (continuous distributions used in latent variable models). The chain rule of probability — $P(w_1, \ldots, w_n) = \prod_{t=1}^{n} P(w_t \mid w_{<t})$ — is used in every chapter and should be understood intuitively as well as formally. Familiarity with the multinomial distribution, the softmax function, and basic maximum likelihood estimation is assumed from Chapter 2 onward. In calculus, the reader needs partial derivatives and the chain rule of differentiation, because training all neural models involves computing gradients of a scalar loss function with respect to millions of model parameters through backpropagation. An intuitive understanding of gradient descent — that we move parameters in the direction that most rapidly reduces the loss — is sufficient; a rigorous treatment of convergence and convexity is not required.

The book does not require measure theory, functional analysis, real analysis beyond undergraduate calculus, abstract algebra, or any form of continuous optimization more advanced than gradient descent. Information theory and deep-learning-specific mathematics — including attention derivations, positional encoding schemes, and scaling law analysis — are developed within the book in the chapters where they first appear. A reader who is comfortable with the contents of a standard

undergraduate linear algebra course (such as Strang's *Introduction to Linear Algebra*), a one-semester probability and statistics course, and single and multi-variable calculus has all the mathematical background this book requires. Readers who feel uncertain about any of these areas are encouraged to review the relevant sections of Goodfellow, Bengio, and Courville's *Deep Learning* [Goodfellow et al., 2016], which provides accessible reviews of all three domains in its early chapters.

1.4.2 Programming Prerequisites

What programming experience does the reader need to engage with the code examples and exercises in this book?

The programming portions of the book require Python competence at the intermediate level. Specifically, the reader should be comfortable defining functions and classes, working with lists and dictionaries, importing and using third-party libraries, reading and writing files, and understanding basic object-oriented programming. All code in the book is written in Python 3.10 or later and is self-contained: each code example includes all necessary imports and can be run directly from the companion repository without modification.

The primary computational framework is PyTorch, which we use for all neural model implementations from Chapter 4 onward. The reader does not need prior experience with PyTorch, but should be able to read Python code fluently and understand basic programming concepts such as loops, conditionals, and function calls. The Hugging Face Transformers library is used extensively from Chapter 8 onward for loading pre-trained models, tokenizing text, and running inference; again, no prior experience is required, as the relevant API calls are introduced and explained when first used. Secondary tools that appear in the book include NumPy for array operations, Matplotlib for visualization, and the `datasets` library from Hugging Face for accessing standard NLP benchmark datasets.

We emphasize that the code in this book is pedagogical rather than production-grade. We prioritize clarity over performance: we avoid complex optimizations, device-specific code paths, and engineering abstractions that would obscure the algorithm being illustrated. A reader who wants to build a production language model system will need to engage with additional engineering literature; the code in this book is designed to make the algorithms transparent and the concepts concrete. All inline code examples are extracts from full Jupyter notebooks in the companion repository, where complete, runnable implementations with expected outputs and dependency specifications are provided.

1.4.3 Notation Conventions

Why does notation matter, and what conventions do we use to ensure mathematical precision without sacrificing readability?

Inconsistent notation is one of the most persistent obstacles to learning in technical fields, particularly in machine learning and NLP, where different research communities have developed different conventions that frequently conflict. We adopt throughout this book the notation of Goodfellow, Bengio, and Courville's *Deep Learning* [Goodfellow et al., 2016], with NLP-specific extensions for vocabulary, sequences, and language model quantities.

The typographic conventions are as follows. Scalars are denoted by lowercase italic letters: x, w, d, η. Vectors are denoted by bold lowercase letters: $\mathbf{x}$, $\mathbf{h}$, $\mathbf{e}$. Matrices are denoted by bold uppercase letters: $\mathbf{W}$, $\mathbf{Q}$, $\mathbf{K}$. Sets are denoted by calligraphic letters: $\mathcal{V}$ for the vocabulary, $\mathcal{D}$ for a

dataset. Probability distributions over discrete random variables use capital P, as in $P(w \mid \text{context})$; probability densities over continuous variables use lowercase p, as in $p(\mathbf{z})$ for a latent variable distribution. The specific symbols used in this chapter and throughout the book are collected in Table **??**, and each symbol is re-introduced at its point of first use in later chapters.

Several notation conventions require particular attention. The token at position t in a sequence is denoted w_t, and the subsequence of all tokens preceding position t is written $w_{<t}$ as shorthand for $(w_1, w_2, \ldots, w_{t-1})$. The vocabulary — the set of all distinct tokens that the model can predict — is denoted $\mathcal{V}$, and its cardinality is $|\mathcal{V}|$. In practice, $|\mathcal{V}|$ ranges from around 10,000 for word-level models to 50,000 to 100,000 for subword models. The model parameters are collected into a vector θ, and the loss function that training minimizes is $\mathcal{L}(\theta)$. For the language modeling objective specifically, the loss is the average negative log-likelihood, $\mathcal{L}(\theta) = -\frac{1}{T}\sum_{t=1}^{T} \log P(w_t \mid w_{<t}; \theta)$. We note that the symbol $\mathbf{V}$ (bold uppercase) is used in Chapter 8 for the value matrix in multi-head attention, distinct from the vocabulary set $\mathcal{V}$; context always disambiguates between these two uses.

1.4.4 The Companion Repository

Where can the reader find runnable code, datasets, and additional resources to complement the material in each chapter?

Every code example in this book has a corresponding complete, runnable Jupyter notebook in the companion GitHub repository. The repository is organized by chapter, with one directory per chapter containing all notebooks, data files, and utility scripts relevant to that chapter. Each notebook specifies its Python and library dependencies with pinned version numbers and includes the expected output of each code cell, enabling the reader to verify that their environment is configured correctly before working through the exercises. The companion repository also contains implementations of all major algorithms presented in the book — n-gram models, feedforward language models, LSTMs, Transformers — with detailed inline comments that cross-reference the relevant equations and sections in the book.

The inline code examples that appear in the text of each chapter are extracts from these full notebooks. They are designed to be legible on their own and to illustrate the algorithmic concept being discussed, but they are not standalone scripts: they may omit boilerplate imports, data loading code, and output formatting that would distract from the pedagogical point. Readers who wish to run the examples should use the full notebooks from the repository rather than copy-pasting the inline extracts. The companion repository additionally includes a collection of supplementary exercises beyond those printed in each chapter, with automated test cases that allow readers to verify the correctness of their solutions, and a set of curated reading lists for each chapter pointing to the original research papers, review articles, and textbook chapters most relevant to deepening understanding of the chapter's topic.

Section 1.4 has established the mathematical and programming prerequisites for the book, introduced the notation conventions that are used throughout, and described the companion repository. We have now laid all of the conceptual groundwork that the remaining chapters depend on. Before proceeding, we collect the eight exercises that accompany this chapter.

Exercises

Exercise 1 (Theory, Basic)

Explain in three to five sentences why predicting the next word might be sufficient for a model to learn grammar, semantics, and even some world knowledge. Use the example of predicting the word following "The capital of France is _____" to ground your explanation.

Hint: Think carefully about what the model must "know" — about geography, about the grammatical category of the word that follows "is," about the typical topics of sentences that begin with "The capital of" — in order to assign high probability to the correct next word. The model's ability to make this prediction accurately requires implicitly encoding information that looks very much like grammatical knowledge and factual knowledge about the world.

Exercise 2 (Theory, Basic)

List three real-world applications of language models that go beyond text generation and explain how each one can be framed as a next-word prediction problem. Your answer should specify, for each application, what constitutes the "context" and what constitutes the "next word."

Hint: Consider machine translation as predicting target-language words conditioned on source-language context. What is the analogous formulation for automatic speech recognition? For question answering?

Exercise 3 (Theory, Intermediate)

Shannon estimated the entropy of printed English at approximately 1.0 to 1.5 bits per character. If English had instead 27 equally likely characters (26 letters and one space), what would the entropy be? What does the gap between this maximum entropy value and Shannon's estimate tell us about the predictability of natural language?

Hint: The maximum entropy of a uniform distribution over k equally probable outcomes is $\log_2 k$. Shannon's estimate was obtained experimentally by asking human subjects to predict characters in real English text. What does the ratio between the two values tell you about the redundancy of English?

Exercise 4 (Programming, Basic)

Using the Hugging Face Transformers library, load GPT-2 and generate 10 different continuations of the prompt "Once upon a time" with temperature 1.0. Observe the variation across the 10 outputs and write two to three sentences discussing why the outputs differ despite the identical prompt.

Hint: Use `pipeline("text-generation", model="gpt2")` with `do_sample=True` and `temperature=1.0`. Set different random seeds for each of the 10 generations, or call the generator 10 times without setting seeds. The variation arises from stochastic sampling from the probability distribution rather than deterministic selection.

Exercise 5 (Programming, Basic)

Write a Python function `top_k_predictions(prompt, model, tokenizer, k=10)` that takes a prompt string, a pre-loaded Hugging Face language model and tokenizer, and a value k, and returns the k most probable next tokens with their probabilities as a list of `(token_string, probability)` tuples. Test your function on three different prompts and discuss any patterns you observe in the predictions.

Hint: Perform a single forward pass through the model, extract the logits at the last token position, apply `torch.softmax` to obtain probabilities, and use `torch.topk` to extract the k largest values. Decode each token index using `tokenizer.decode`.

Exercise 6 (Programming, Intermediate)

Compare the top-5 next-token predictions of GPT-2-small (124 million parameters, model identifier `"gpt2"`) and GPT-2-large (774 million parameters, model identifier `"gpt2-large"`) on the same set of five prompts of your choosing. For each prompt, identify a "target" word that a human would consider the most natural or expected continuation, and report the probability that each model assigns to that target word. Does the larger model consistently assign higher probability to the expected word?

Hint: Load both models using `GPT2LMHeadModel.from_pretrained`. Run identical forward passes and compare the softmax probability at the index of the target token. Design your prompts to test different types of knowledge: factual, grammatical, and stylistic.

Exercise 7 (Programming, Intermediate)

Modify the text generation code to use greedy decoding: at each step, always select the single most probable next token rather than sampling. Generate 100 tokens from GPT-2 starting from the prompt "The best way to learn programming is to" using greedy decoding. Observe the output and write two to three sentences explaining why greedy decoding often produces repetitive or degenerate text.

Hint: Set `do_sample=False` in `model.generate()`. With no randomness in the selection process, the model's predictions become deterministic, and any tendency to repeat high-probability patterns is self-reinforcing: generating a word increases its prominence in the context, which may increase its probability at subsequent positions.

Exercise 8 (Programming, Advanced)

Build a simple unigram language model from word frequencies in a publicly available text file (for example, a chapter of a Project Gutenberg novel). Your implementation should: (a) tokenize the text by splitting on whitespace and converting to lowercase; (b) count the frequency of each word type; (c) normalize the counts to produce a probability distribution over the vocabulary; and (d) generate 50 words by sampling from this distribution. Compare the output qualitatively to a 50-word continuation generated by GPT-2 from a related prompt.

Hint: Use `collections.Counter` for frequency counting and `numpy.random.choice` with a `p` argument for sampling. The unigram model will produce "word salad" — a grammatically incoherent sequence — because it ignores all context. This contrast with GPT-2 output illustrates concretely why conditioning on context matters for language modeling quality.

Looking Ahead

We have argued in this chapter that language modeling — estimating $P(w_{\text{next}} \mid w_{<t})$ — is the single most important idea in NLP, connecting Shannon's information-theoretic experiments in 1948 to the frontier language models of 2026. We have traced the historical development from count-based n-gram models to neural architectures to the Transformer and the large language model era, mapped the four-part structure of the book, and stated the mathematical and programming prerequisites for what follows.

But how do we make the prediction objective mathematically precise? How do we decompose the probability of an entire sentence into a product of next-word predictions, and how do we estimate

the parameters of a model that assigns those probabilities? How do we measure whether one model predicts better than another? These are mathematical questions, and they require mathematical tools. In the next chapter, we develop the probability theory, information theory, and optimization foundations that every subsequent chapter depends on: the chain rule decomposition, entropy and cross-entropy, perplexity as the canonical language model evaluation metric, and the maximum likelihood training objective that unifies all of the models we study.

References

[Shannon, 1948] Shannon, C. E. (1948). A Mathematical Theory of Communication. *Bell System Technical Journal*, 27(3), 379–423.

[Shannon, 1951] Shannon, C. E. (1951). Prediction and Entropy of Printed English. *Bell System Technical Journal*, 30(1), 50–64.

[Jelinek, 1976] Jelinek, F. (1976). Continuous Speech Recognition by Statistical Methods. *Proceedings of the IEEE*, 64(4), 532–556.

[Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, 3, 1137–1155.

[Mikolov et al., 2013] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *Proceedings of ICLR*.

[Bahdanau et al., 2015] Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural Machine Translation by Jointly Learning to Align and Translate. *Proceedings of ICLR*.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. *Advances in Neural Information Processing Systems*, 30, 5998–6008.

[Devlin et al., 2019] Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Proceedings of NAACL-HLT*, 4171–4186.

[Radford et al., 2019] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners. OpenAI Technical Report.

[Brown et al., 2020] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., et al. (2020). Language Models Are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 33, 1877–1901.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.