Large Language Models: An Illustrated Guide

(Work in progress)

llmbook.queirozf.com

Felipe Q. B. Almeida

February 2026

173130535b5ada6fc6f7356b474c8a69d9d7f8f8

Contents

Hello!

What this book IS

- A book focused on examples and figures to explain how Language Modeling started and how we got to where we are today.
- A technical book aimed at people with *some* mathematical and *some* programming knowledge.

What this book is NOT

- A reference for how to implement an LLM at the code level. Read Build a Large Language Model From Scratch by Sebastian Raschka instead;
- A book with extensive mathematical proofs of the statistical and algebraic components of LLMs (not sure there is any good source here other than the original papers themselves);
- A book on tools and frameworks to work with LLMs (Pytorch, Tensorflow, Huggingface, etc).
- A general book about NLP. Read Speech and Language Processing by Dan Jurafsky and James H. Martin.
- A book on how to deal with problems introduced by LLMs: hallucinations, jailbreaking, bias, existential risks, etc.
- A book on the empirical techniques and hacks to make LLMs work better at scale.
- A book on the technicals of putting LLM-enabled systems to use in products and organizations.
- A comprehensive reference on how Reinforcement Learning is applied to fine-tune LLMs. Read the RLHF Book by Nathan Lambert and watch Lectures 1-10 on [Stanford CS224R Deep Reinforcement Learning].

Acknowledgements

- My family and my dog Zulu: For support throughout my life.
- Alan Watts: The person who's had the biggest spiritual influence on my life. Buy his books on Amazon
- Aaron Clarey: Thanks for all the advice and hard truths delivered. Buy Aaron's books on Amazon and subscribe to his Youtube channel

1. Language Models: What and Why

This chapter covers

- Introducing Language Models (LMs); what they are, how they work, and basic use cases
- Showing why and how neural LMs have become key to modern NLP—and the role of representations
- Explaining how Transformers have upended the field in recent years
- Aligning LMs to human preferences, as is the case with ChatGPT

Anyone who hasn't been living under a rock in the last few years (late 2010s to early 2020s) has been bombarded with examples of seemingly magical content produced by new Artificial Intelligence (AI) models. Full novels written by AI; poems specifically generated to copy a specific author's style; Chatbots that act indistinguishably from a human being. The list goes on.

The names of such models may sound familiar to those with a technical bent: GPT, GPT-2, BERT, GPT-3, ChatGPT, and so on.

These are called **Large Language Models (LLMs)** and they have only grown more powerful and more expressive over time, being trained on ever larger amounts of data and applying techniques discovered by researchers in academia and in companies such as Google, Meta, and Microsoft.

In this chapter, we will give a brief introduction to language models (large and otherwise) and related technologies, providing a foundation for the rest of the book. It's called "What and Why" because we show *what* LMs are but also *why* they are now the base layer for modern Natural Language Processing (NLP).

In Section 1.1 we explain at a basic level what language models (LMs) are and how one can use them. Section 1.2 introduces the main types of LMs, namely statistical and neural language models. In Section 1.3 we show how Attention mechanisms and the Transformer architecture help LMs better keep state and use a word's context. In Section 1.4 we explain why LMs play a pivotal role in modern NLP and, finally, in Section 1.5 we show what alignment means in the context of LMs and how it is used to create models such as ChatGPT.

1.1. What are Language Models?

As the name suggests, a Language Model (LM) *models* how a given language works.

Modeling a language means assigning scores to arbitrary sequences of words, such that the higher the score for a sequence of words, the more likely it is a meaningful *sentence* in a language such as English. This is shown in Figure 1.1:

> *Large* language models
>
> The title of this book specifically mentions *Large* Language Models (LLMs). The term is not precisely defined but our working definition is: Large Language Models are LMs that **(a)** employ large, deep neural nets (billions of parameters) and **(b)** have been trained on massive amounts of text data (hundreds of billions of tokens).

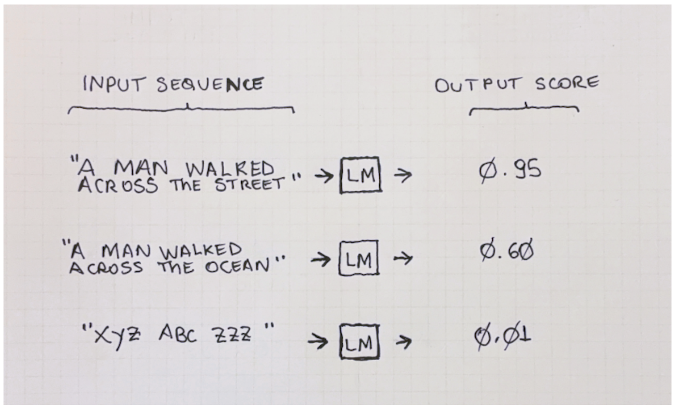

INPUT SEQUENCE
OUTPUT SCORE
"A MAN WALKED ACROSS THE STREET"
LM
Ø.95
"A MAN WALKED ACROSS THE OCEAN"
LM
Ø.6Ø
"XYZ ABC ZZZ"
LM
Ø.Ø1

Figure 1.1.: At its most basic, a Language Model is a *function* that takes in a sequence of words as input and outputs a score. High scores mean the input is *likely* in a language such as English. Low scores mean it's probably gibberish. An in-depth overview of language modeling is given in Chapter 2.

The LMs we will focus on in this book learn from *data* (as opposed to rules manually crafted by linguistics experts), mostly using Machine Learning (ML) tooling.

A perfect picture of a language would require us to have access to every document that was ever written in it. This is clearly impossible, so we settle for using as large a dataset as we can. An unlabelled natural language dataset is called a *corpus*.

The corpus is thus the set of documents that represent the language we are trying to model, such as English. The corpus is the data LMs are trained on. After they are trained they can then be used, just like any other ML model. Here, "using" a trained LM means feeding it word sequences as input and obtaining a likelihood score as output, as is shown in Figure 1.1.

Corpus and Corpora

A *corpus* is a set of unlabelled documents used to train a Language Model. *Corpora* is the plural form of *corpus*.

The distinction between **training-time** and **inference-time** is crucial to understanding how LMs work.[1]

Training-time refers to the stage where the model processes the corpus and learns the characteristics of the target language. This step is usually time-consuming (hours or days). On the other hand, *inference-time* is the stage in which the language model is *used*, for example in calculating a sentence's likelihood score; Inference is usually fast (milliseconds or seconds) and takes place after training. This will be explained in more detail in Chapter 2 where we dive deep into language modeling.

In the sections below we will have a better look at how LMs can be used in a real-world setting and what are the main types used in practice.

1.1.1. LM use cases

As we saw in the previous section, LMs need to be trained on a corpus of documents. After they have been trained, they hold some idea of what the language in question looks like, and only *then* can we use them for practical tasks.

The most basic for a language model is to output likelihood scores for word sequences, as we stated previously. But there is an additional use for them: *predicting* the next word in a sentence.

The two ways to use—or perform inference with—a trained language model are therefore: **(a)** calculate the likelihood score for an arbitrary input sequence and **(b)** predict the most likely next word in a sentence, based on the previous words. These are two simple and seemingly uninteresting applications, but they will unlock surprising outcomes as we'll see in the next sections. Figure 1.2 shows examples for both uses, side-by-side:

It may not be immediately obvious but these use cases are two sides of the same coin: if you have a trained LM trained to calculate the likelihood score, it is easy to use it to predict or guess what the next word in a sentence will be. This is because you can brute force your way into the solution by scoring every possible word in the vocabulary and picking the option with the highest score! (see Figure 1.2, right side).

Let's now have a brief look at the main types of language models, what they have in common, and how they differ.

[1]"In this book, training-time refers to the phase when parameters are learned; inference-time refers to using the trained model to make predictions. When we mean duration, we say training duration or inference latency."

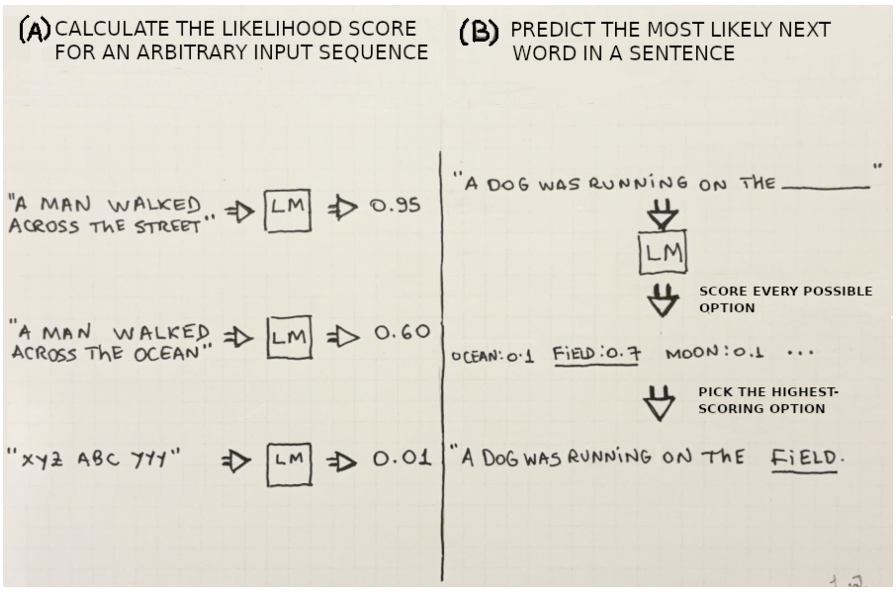

(A) CALCULATE THE LIKELIHOOD SCORE FOR AN ARBITRARY INPUT SEQUENCE
(B) PREDICT THE MOST LIKELY NEXT WORD IN A SENTENCE
"A MAN WALKED ACROSS THE STREET"
LM
0.95
"A MAN WALKED ACROSS THE OCEAN"
LM
0.60
"XYZ ABC YYY"
LM
0.01
"A DOG WAS RUNNING ON THE ______"
LM
SCORE EVERY POSSIBLE OPTION
OCEAN: 0.1
FIELD: 0.7
MOON: 0.1
...
PICK THE HIGHEST-SCORING OPTION
"A DOG WAS RUNNING ON THE FIELD.

Figure 1.2.: The two basic uses for Language Models: (a) calculate the likelihood score for a given word sequence and (b) predict the next word in the sequence. Note that the left side of this figure is just a rehash of Figure 1.1

1.2. Types of Language Models

In practice, the two most common ways to implement language models are either via **statistics** (counting word frequencies) or with the aid of **neural networks**. For both the general structure stays the same, just like we explained in Section 1.1: the language model is trained on some corpus and, once trained, it can be used as a function to measure how likely a given word sequence is *or* to predict the next word in a sequence.

Figure 1.3 shows the main types of language models, namely statistical and neural language models, with subclassifications. These are discussed in sections 1.2.1 and 1.2.2. Also, Chapter 2 and **?@sec-ch-neural-language-models-and-self-supervision** will provide a thorough analysis of these models.

Let's now explore the characteristics of statistical and neural language models.

1.2.1. Statistical Language Models

One simple way to model a language is to use probability distributions. We can posit that language is defined as the probability distribution of every word in that language.

In statistical language models, the probability of a word sequence is defined as the joint probability of the words in the sequence.

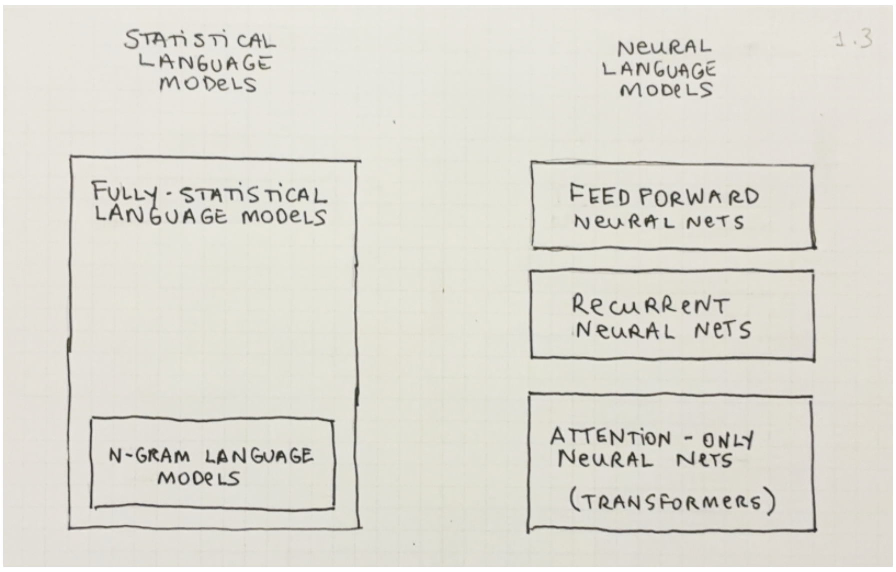

1.3
STATISTICAL LANGUAGE MODELS
NEURAL LANGUAGE MODELS
FULLY-STATISTICAL LANGUAGE MODELS
N-GRAM LANGUAGE MODELS
FEED FORWARD NEURAL NETS
RECURRENT NEURAL NETS
ATTENTION - ONLY NEURAL NETS (TRANSFORMERS)

Figure 1.3.: Types of Language models

It's possible to decompose the joint probability into a product of conditional probabilities, using the chain rule of probability. This enables one to make the necessary calculations and arrive at a likelihood *score* for a given sequence.

Note

Section 2.3 includes detailed explanation and examples of how to calculate a likelihood score with statistical LMs.

This approach, however, quickly becomes impractical with real-world-size data, as it is computationally expensive to calculate the terms for large text corpora—the number of possible combinations grows exponentially with the size of the context and the vocabulary.

In addition to being inefficient, statistical LMs aren't able to generalize calculations if we need to score a word sequence that is not present in the corpus—they would assign a score of zero to every sequence not seen in the train set.

Therefore, these models aren't often used in practice—but they provide a foundation for N-gram models, as we'll see next.

1.2.1.1. N-gram language models

N-gram language models are an optimization on top of statistical language models. But what *are* N-grams?

N-grams are an abstraction of words. N-grams where $N = 1$ are called unigrams and are just another name for a word. If $N = 2$, they are called bigrams and they represent ordered pairs of words. If $N = 3$, they're called trigrams and—you guessed it—they represent ordered triples of words. Figure 1.4 shows an example of what a sentence looks like when it's split into unigrams, bigrams, and trigrams:

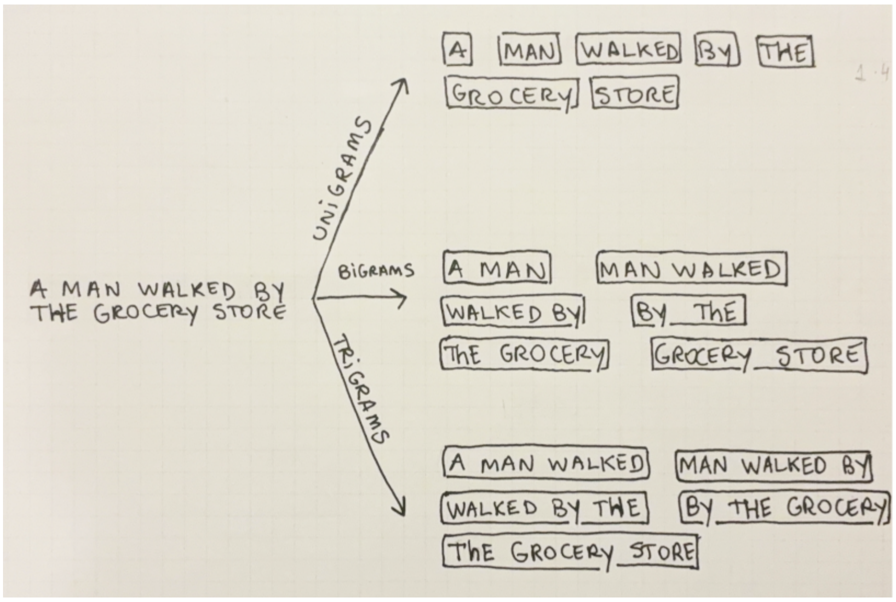


Figure 1.4.: N-grams: Representing a sentence with N-grams: In this example, the sentence *"A man walked by the grocery store"* can be represented with unigrams, bigrams, trigrams, etc. Note that a unigram is just another name for a word.

But *how* do N-grams make statistical LMs better?

N-grams enable us to prune the number of context words when calculating conditional probabilities. Instead of considering all previous words in the context, we approximate it by using the last N words. This reduces the space and the number of computations needed as compared with fully statistical LMs and helps address the curse of dimensionality related to rare combinations of words.

N-gram models are no panacea, however; they still suffer from the inability to generalize calculations to unseen sequences; and deliberately ignoring contexts beyond $N - 1$ words hinders the capacity of the model to consider longer dependencies. N-gram models will be discussed in more detail in Chapter 2.

1.2.2. Neural Language Models

As interest in neural nets picked up again in the early 2000s, researchers (starting with Bengio et al. (2003)) began to experiment with applying neural nets to the task of building a language model, using the well-known and trusted backpropagation algorithm. They found out that not only it was possible, but it worked better than any other language model seen so far—and it solved a key problem faced by statistical language models: not being able to generalize into unseen words.

The training strategy relies on *self-supervised learning*: training a neural network where the features are the words in the context and the target is the next word. One can simply build a training set like that and then train it in a supervised way like any other neural net.[2]

In addition to being a good way to train language models (in the sense that they are good at predicting how likely a piece of text is), it turns out that there remains an interesting by-product after training a neural LM: *learned* representations for words—**word embeddings**.

> Note
>
> **?@sec-ch-representing-words-and-documents-as-vectors** will focus specifically on text representations and embeddings will be discussed in depth.

Even though the first model introduced by Bengio et al. (2003) was a relatively simple feedforward, shallow neural net, it proved that this strategy worked, and it set the path forward for many other developments.

With time, neural LMs evolved by using ever more complex neural nets, trained on increasingly larger datasets. Deep neural nets, convolutional neural nets, recurring connections, the encoder-decoder architecture, attention, and, finally, transformers, are just some examples of the technologies used in these models. Figure 1.5 shows a selected timeline with some of the key technological breakthroughs and milestones related to neural LMs:

Most modern language models are neural LMs. This is unsurprising because **(1)** neural nets can handle a lot of complexity and **(2)** neural LMs can be trained on massive amounts of data, with no need for labeling. The only constraints are the available computing power and one's budget.

Research (from both academia and industry) on neural nets has advanced a lot in the last decades so it was a match made in heaven: as the magnitude of text on the Web grew larger, there appeared new and more efficient ways to train neural nets: better algorithms on the software side and purpose-built hardware on the hardware side.

[2]See **?@sec-self-supervision-and-language-modeling** for more details on self-supervision applied to language modeling.

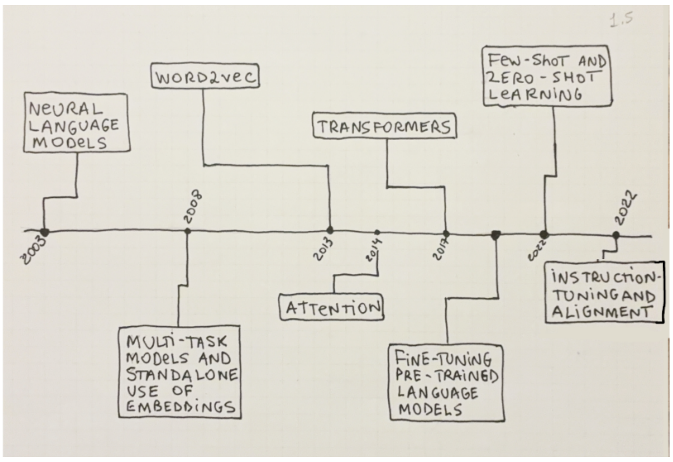

1.5
Neural Language Models
Word2vec
Transformers
Few-Shot and Zero-Shot Learning
2003
2008
2013
2014
2017
2020
2022
Attention
Instruction-Tuning and Alignment
Multi-task Models and Standalone Use of Embeddings
Fine-Tuning Pre-Trained Language Models

Figure 1.5.: Selected timeline with key milestones related to neural language models, from both academia and industry.

Let's now explain the role a word's context plays in neural LMs and how they help us train better models.

1.2.2.1. The need for memory in Neural LMs

The basic building block of text is a word, but a word on its own doesn't tell us much. We need to know its **context**—the other words around it—to fully understand what a word means. This is seen in *polysemous*[3] words: the word *"cap"* in English can mean a head cover, a hard limit for something (a spending cap), or even a verb. Without context, it's impossible to know what the word means.

As a human is better able to understand a word when its context is available, so are LMs. In the case of language modeling, this means having some kind of *memory* or *state* in the model—so that it can consider past words as it predicts the next ones.

A word's *context*

In LM parlance, the *context* of a word W refers to the accompanying words around W. For example, if we focus on the word *"running"* in the sentence *"A dog is running on the field"*, the context is made up of *"A dog is ..."* on the left side and *"... on the field"* on the right side.

While the neural language models we have seen so far do take *some* context into account when training, there are key limitations: They use small contexts (5-10 words only) and

[3]Polysemous words are those that have multiple meanings.

the context size needs to be fixed *a priori* for the whole model[4]. Recurrent neural nets can work around this limitation, as we'll see next.

1.2.2.2. Recurrent Neural Networks

The standard way to incorporate *state* in neural nets (to address the memory problem explained above) has for some time been Recurrent Neural Networks (RNNs). RNNs use the output from the previous time step as additional features to produce an output for the current time step. This enables RNNs to take past data into account. The basic differences between regular (i.e. feedforward) and recurrent neural nets are shown in Figure 1.6:

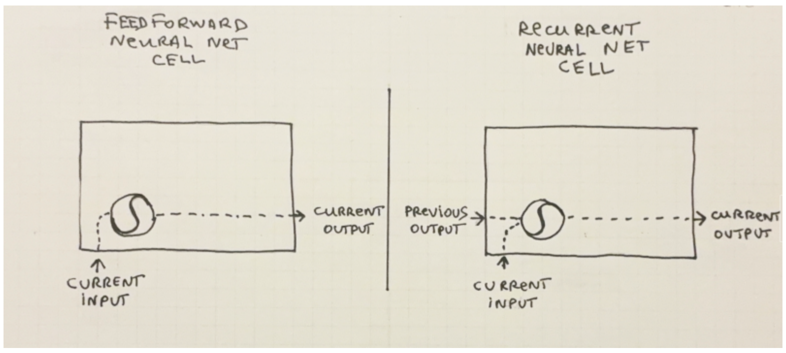


Figure 1.6.: feedforward neural nets only use features from the current time step to calculate the output, whereas recurrent neural nets use features from the current time stamp but also use the output from the previous time step to calculate the output. The dotted lines represent the flow of information and the circles represent nonlinear operations on vectors, such as the sigmoid function.

The simplest way to train RNNs is to use an algorithm called Backpropagation Through Time (BPTT). It's similar to the normal backpropagation algorithm but for each iteration, the network is first *unrolled* so that the recurrent connections can be treated as if they were normal connections.

There are three issues with BPTT for RNNs however: Firstly, it's **computationally expensive** to execute especially as one increases the number of time steps one wants to look at (in the case of NLP, this means the size of the *context*). Secondly, it's not easy to **parallelize** training for RNNs, as many operations must be executed sequentially. Finally, running backpropagation over such large distances causes gradients to **explode** or **vanish**, which precludes the training of networks using larger contexts.

[4]*Feedforward* neural nets cannot natively deal with variable-length input.

1.2.2.3. Better memory: LSTMs

It turns out one can be a little more clever when propagating past information in RNNs, to enable storing longer contexts while avoiding the problems of vanishing/exploding gradients.

One can better control how past information is passed along with so-called *memory cells*. One commonly used type of memory cell is the LSTM (Long Short-term Memory).

LSTMs were introduced some time ago by Hochreiter and Schmidhuber (1997) and they work by propagating an internal state and applying nonlinear operations to the inputs (from the current and previous time steps) and gates to control what should be input, output or forgotten[5]. In vanilla RNN cells, no such operations are applied, and no state is propagated explicitly. These 3 gates are the 3 solid blocks labeled "F", "I" and "O", shown in Figure 1.7, right side.

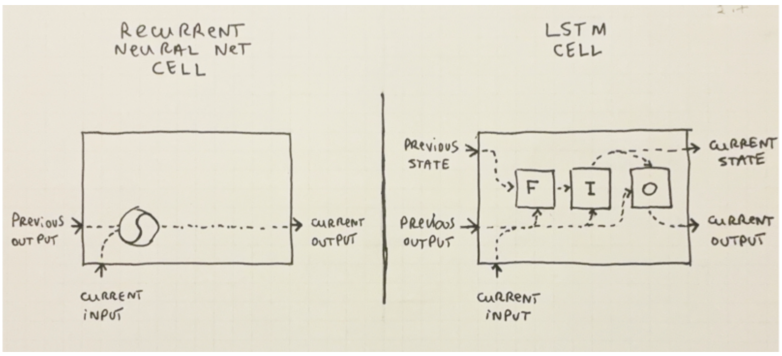


Figure 1.7.: RNN cells (left) take the output from the previous time step and also the current input and apply a nonlinear operation (circles) to produce the current output. LSTM cells (right) also propagate an internal state, to which several nonlinear operations are applied—forget gates, input gates, and output gates, represented by the letters F, I, and O, respectively. The dotted lines represent the flow of information and the circles represent nonlinear operation on vectors, such as the sigmoid function.

Crucially, LSTMs don't address the computational costs of training recurrent neural nets, because the recurrent connections are still present. They do help avoid the problem of vanishing/exploding gradients and they also help in storing longer-range dependencies than would be possible in a vanilla RNN, but the scaling issues when training remain. We will cover RNNs and LSTM cells in more detail in Chapter 04.

In Section 1.2 we saw the main types of language models and we showed how neural nets enable better training of LMs. We also saw how important it is for LMs to be able to keep state and how RNNs can be used for that, but training these is costly and they don't

[5]The usual implementation of an LSTM includes a "forget-gate" as introduced by Gers et al. (1999).

work as well as we'd expect. Attention mechanisms and the Transformer architecture address these points, as we'll see next.

1.3. Attention and the Transformer revolution

If you are interested in modern NLP, you will have heard the terms *Attention* and *Transformers* being thrown around recently. You might not understand *exactly* what they are, but you picked up a few hints and you have a feeling Transformers are a significant part of modern LLMs—and that they have something to do with Attention.

You'd be right on both counts. Let's see what Attention is, how it enables Transformers, and why they matter. These two topics will be covered in more detail in Part II.

1.3.1. Enter Attention

The problem of how to propagate past information to the present efficiently and accurately also occupied the minds of researchers and practitioners working on a different language task: Machine Translation.

The traditional way to handle machine translation and other *sequence-to-sequence* (Seq2Seq) problems is using a recurrent neural network architecture called the *encoder-decoder*. This architecture consists of *encoding* input sequences into a single, fixed-length vector and then *decoding* it back again to generate the output. See the upper part of Figure 1.8 for a visual representation.

Note

We'll cover Sequence-to-sequence (Seq2seq) learning in more detail in **?@sec-ch-sequence-learning-and-the-encoder-decoder-architecture**.

Soon after the introduction of these encoder-decoder networks, other researchers (Bahdanau et al. (2014)) proposed a subtle but impactful enhancement: instead of encoding the input sequences into a *single* fixed-length vector as an intermediary representation, they are encoded into multiple so-called *annotation vectors* instead. Then, at decoding time, an *attention mechanism* learns which annotations it should use—or attend to. This can be seen before the decoder in Figure 1.8, below.

More specifically, the attention mechanism inside the decoder contains another small feedforward neural network with learnable parameters. This is the so-called alignment model and its task is precisely to learn, over time, which of the vectors generated by the encoder best fit the output it is trying to generate. This is represented in Figure 1.8 on the bottom part:

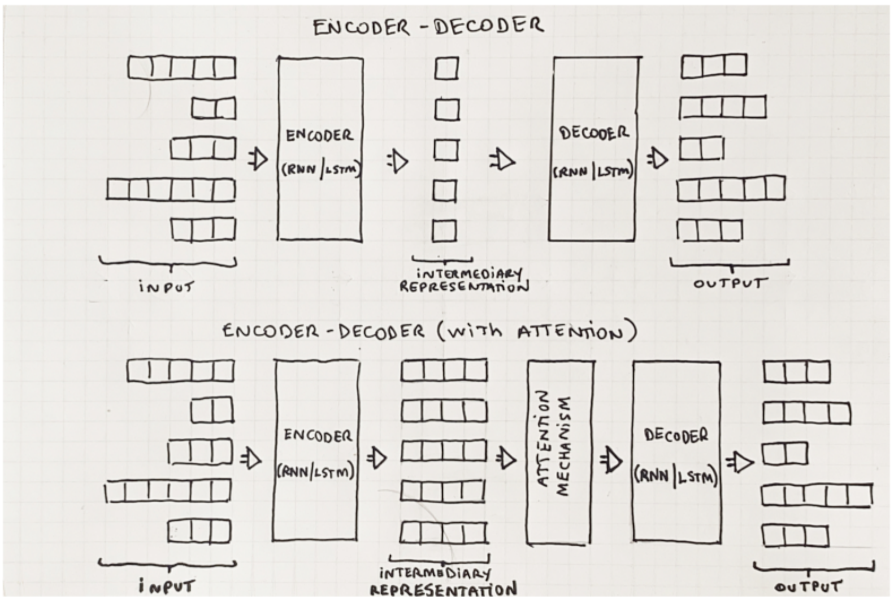


Figure 1.8.: Differences between regular (above) and attention-enabled (below) encoder-decoder networks. The structure is similar but the bottom network uses multiple vectors for the intermediary representation and there is an extra component before the decoder—the attention mechanism.

Attention as Information Retrieval

A common way to think about Attention is by framing it as an *information retrieval* problem with *query*, *key*, and *value* vectors.
In a translation task, for example, each output word (in the target language) can be seen as a *query* and each input word (in the source language) is a combination of keys and values, which will be *searched over* to find the best input word. This will be explained in more detail in **?@sec-ch-attention-and-transformers**.

While adding Attention cells to encoder-decoder networks does allow for more precise models, **they still use recurrent connections**, which make them computationally expensive to train and hard to parallelize. This is where Transformers come in, as we'll see next.

1.3.2. Transformers

Vaswani et al. (2017) introduced an alternative version of the encoder-decoder architecture, along with several engineering tricks to make training such networks much faster. It was called the **Transformer**, and it has been the architecture of choice for most large NLP models since then.

The seminal Transformer article was called "Attention is all you need", for good reason: The proposed architecture ditched RNN layers altogether, replacing them with Attention layers (while keeping the encoder-decoder structure). This is shown in detail in Figure 1.9: The encoder and decoder components are there but recurrent connections are nowhere to be seen—only attention layers.

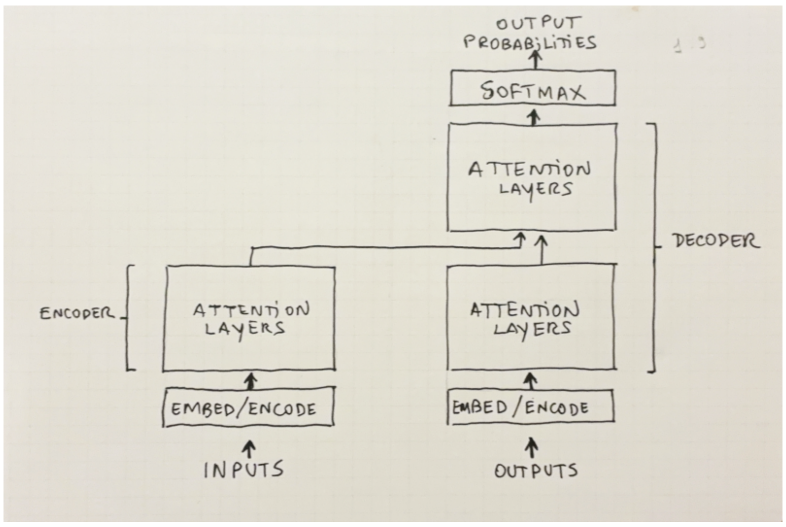


Figure 1.9.: The original Transformer model for Seq2Seq learning. It is still an encoder-decoder architecture (akin to Figure 1.8), but there are no RNNs or any other recurrent connections—only attention layers. Adapted from Vaswani et al. (2017)

The removal of recurrent connections was crucial. The main problem with RNNs and LSTMs was precisely the presence of these connections. As we saw earlier, these precluded parallel training using GPUs, TPUs, and other purpose-built hardware and, therefore, severely limited the amount of data models could be trained on.

Without RNNs or any recurrent connections, the original Transformer model was able to match or even surpass the then-current state of the art in machine translation, at a fraction of the cost (100 to 1000 times more efficiently).

The key advancements introduced by Transformers were **(1)** using *self-attention* instead of recurrent connections both in the encoder and the decoder **(2)** encoding words with *positional embeddings* to keep track of word position and **(3)** introducing *multi-head attention* as a way to add more expressivity while enabling more parallelization in the architecture.

Let's see how and why Transformers are used for language modeling.

1.3.3. Transformer-based Language models

Now we know what Transformers are, but we saw that they were created for Seq2Seq learning, not for language modeling.

We can, however, *repurpose* encoder-decoder Transformers to build language models—we can just use the decoder part of the network, in a self-supervised training setting, just like the original LMs we saw in previous sections.

Such language models are now called *encoder* Transformers or *decoder* Transformers, depending on which part of the original Transformer they use. The first truly large LM based on the Transformer architecture was the OpenAI GPT-1 model by Radford et al. (2018), a decoder Transformer. Figure 1.10 shows a timeline of released transformer-based LLMs, starting with GPT-1, soon after the seminal paper was published:

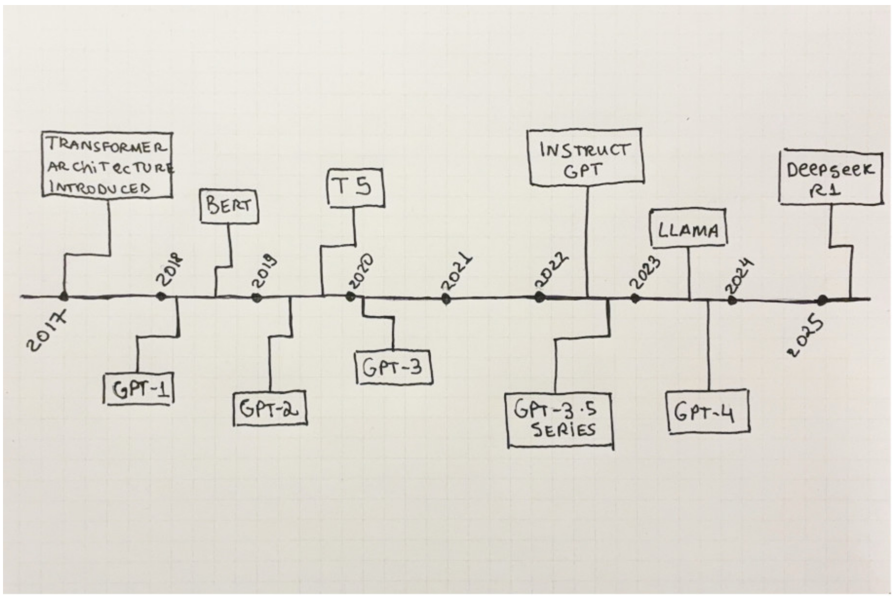


Figure 1.10.: Selected timeline with the main transformer-based LLMs released as of this writing

ℹ Note

?@sec-ch-selected-models-explained contains a description of the most important Transformer-based LMs.

We are still missing one part of the puzzle: why exactly are language models (especially large LMs) so important for NLP?

1.4. Why Language models? LMs as the building blocks of modern NLP

We saw in the previous sections that one can use Neural Nets to train Language Models—and that this works surprisingly well. We also saw how using Transformers enables us to train massively larger and more powerful LMs.

You are probably wondering why we talk so much about Language Models if their uses are relatively limited and seemingly uninteresting (predicting the next word in a sentence doesn't seem all that sexy, right?).

The short answer is threefold: **(1)** LMs can be used to build *representations* for downstream NLP tasks **(2)** LMs can be trained on huge amounts of data because they're unsupervised **(3)** we can plug language models into *any* NLP task.

We'll explain each of these 3 points in detail in sections 1.4.2, 1.4.3 and 1.4.4 but first, let's quickly see what we mean by NLP.

1.4.1. NLP is all around us

NLP stands for Natural Language Processing, an admittedly vague term. In this book, we will take it to mean any sort of Machine Learning (ML) task that involves natural language — text as written by humans. This includes all physical text ever written and, most importantly, all text on the Web.

Table 1.1 shows a selected list of NLP tasks that have been addressed both by researchers in academia and practitioners in the industry:

Table 1.1.: Selected examples of NLP tasks

Task	**Description/Example**
Language Modeling	Capture the distribution of words in a language. Also, score a given word sequence to measure its likelihood or predict the next word in a sentence.
Machine Translation	Translate a piece of text between languages, keeping the semantics the same.
Natural Language Inference (NLI)	Establish the relationship between two pieces of text (e.g., do the texts *imply* one another? Do they *contradict* one another?). Also known as Textual Entailment.
Question Answering (Q&A)	Given a question and a document, retrieve the correct answer to the question (or conclude that it doesn't exist).

Task	Description/Example
Sentiment Analysis	Infer the *sentiment* expressed by text. Examples of sentiments include: "positive", "negative" and "neutral".
Summarization	Given a large piece of text, extract the most relevant parts thereof (extractive summarization) or generate a shorter text with the most important message (abstractive summarization).

All of these problems can be framed as normal machine learning tasks, be they supervised or unsupervised, classification or regression, pointwise predictions or sequence learning, binary or multiclass, discrete or real-valued. They can be modeled using any of the default ML algorithms at our disposal (neural nets, tree-based models, linear models, etc).

The one difference between text-based ML—that is, NLP— and other forms of ML tasks is that text data must be *encoded* in some way before it can be fed to traditional ML algorithms. This is because ML algorithms cannot deal natively with text, only numerical data. How to *represent* text data is crucial in NLP, as we'll see next.

1.4.2. It's all about representation

As explained above, text data must be *encoded* as numbers before we can apply ML to it. Therefore all NLP tasks must begin by building representations for the text we want to operate on. The way we represent data in ML is usually via numeric vectors.

The traditional form of representing text is the so-called *bag-of-words* (BOW) schema. As the name implies, this means representing text as an unordered set (i.e. a bag) of words. The simplest way to represent one word is to use a *one-hot encoded* (OHE) vector. An OHE vector only has one of its elements "turned on" with a 1. All other elements are 0. See Figure 1.11 (top part) for an example.

TF-IDF

You may have heard of TF-IDF as a common way to represent text data. We don't include it in this section because TF-IDF vectors are used to represent a *document*, not a single word. Again, refer back to **?@sec-ch-representing-words-and-documents-as-vectors** where we'll explain these concepts in detail.

Although simple, BOW encoding works reasonably well in practice, for many NLP tasks—they are usually *combined* with some form of weighting such as TF-IDF (see callout above).

Now for the problems. Firstly, OHE vectors are *sparse* (only one element is "on" and all others are "off") and *large* (their length must be the size of the vocabulary). This means they are memory- and compute-intensive to work with. Secondly, OHE vectors encode no *semantic* information at all. The OHE vector for the word *"cow"* is just as geometrically "distant" from the word *"bull"* as it is from the word *"spacecraft"*.

We mentioned learned representations in Section 1.2.2 when we said that one of the by-products of training a neural LM was the creation of word embeddings: fixed-length representation vectors for each word.

Embeddings look very different from OHE vectors, as can be seen in Figure 1.11. They are smaller in length; they are denser and they encode semantic information about the word. This opens up a whole new avenue for making NLP more accurate.

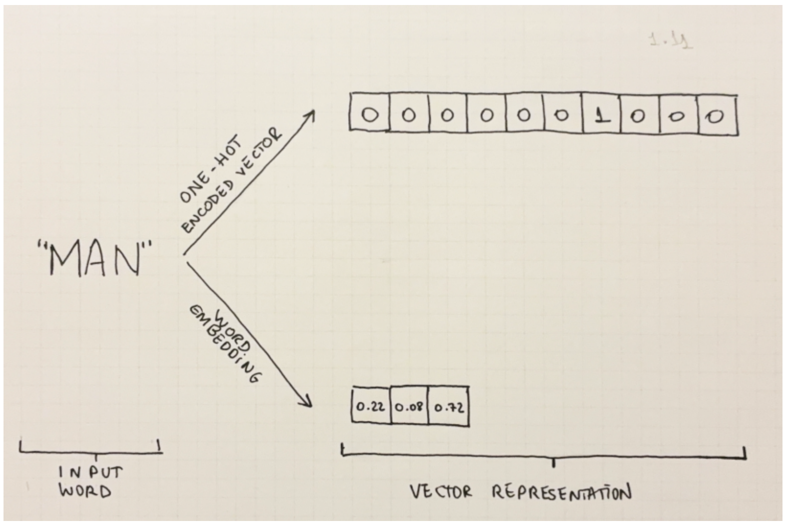


Figure 1.11.: The word "man", represented in two ways: as a one-hot encoded vector (top) and as a word embedding (bottom). Word embeddings are shorter and denser than OHE vectors.

Another advantage of embeddings is that they get continually more accurate as the LMs they are trained on get larger and more powerful. See Table 1.2 for a summarized comparison between OHE vectors and word embeddings:

Table 1.2.: Differences between One-hot encoded vectors and word embeddings

	OHE Vectors	**Word Embeddings**
Density	Sparse	Dense
Discrete vs Continuous	Discrete	Continuous
Length	Long (as long as the vocabulary size)	Short (Fixed-length)

	OHE Vectors	**Word Embeddings**
Encoded Semantics	No semantic information encoded	Encode semantic information (similar words are closer together)

Word2vec (Mikolov et al. (2013)) was one of the first LMs trained exclusively to produce embeddings. It showed that a relatively simple architecture (a shallow, linear neural net) trained on more data beats more complex models by far.

The embeddings produced by Word2vec were so good that one could even perform arithmetic on them and arrive at reasonable results. Figure 1.12 shows an example of this: the country-capital relationship can be represented as a vector addition. If you add the vector that represents the country-capital relationship to the vector that represents a country, you will arrive close enough to the vector that represents its capital city!

Word embeddings were an immediate boon to NLP tasks: they could immediately be used as a drop-in replacement for OHE vectors.

1.4.3. Unsupervised training for the win

Training language models does not require labeled data; you just need natural language datasets to either calculate word co-occurrence statistics or train a neural network in a self-supervised manner, as we explained in Section 1.2.

Unlabeled data is much more widely available and cheaper to obtain, as labeling is usually done by humans. This greatly increases the amount of data LMs can be trained on, and thus the amount of knowledge such they can hold.

This makes LMs a great base layer that could, in theory, encode *all* knowledge that exists in text form, including all text in books but, most importantly, all text that's available in digital form.

With the explosion of the amount of text data on the web, one could create *corpora* in the order of trillions of tokens. This, together with advancements in model architectures and purpose-built hardware, has enabled the creation of very large and capable models. Access to computing resources becomes the only real bottleneck to ever larger models.

As models reach nearly unlimited capacity, the question then becomes: "How do we put these *world models* to use?". This is what we will see now, as we draw the final connection between LMs and NLP at large and show why LMs potentialize *all* NLP tasks.

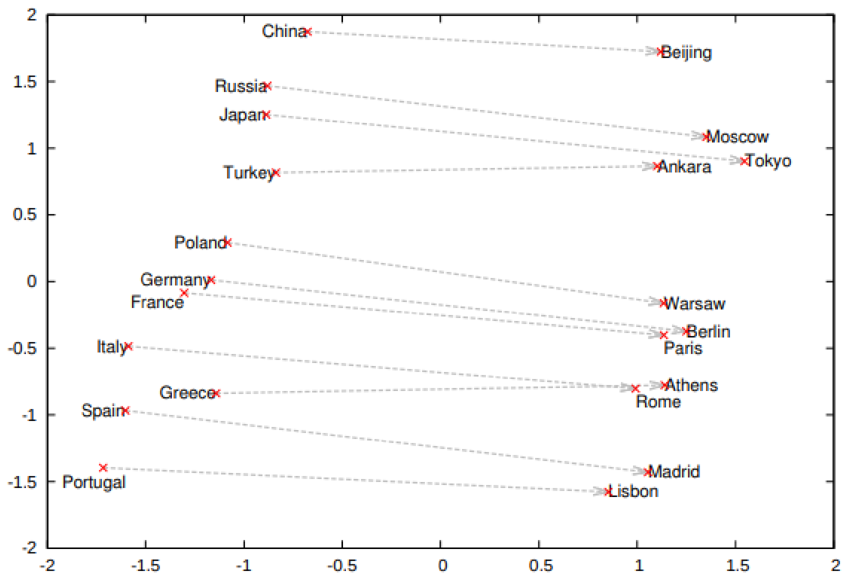


Figure 1.12.: Word2vec embedding vectors for countries and capitals plotted on a 2D chart. They are so accurate that one can visually and geometrically identify country-capital relationships over several pairs. Source: Mikolov et al. (2013)

1.4.4. The de facto base layer for NLP tasks

The main reason why language models can be trained on such large datasets (in the order of trillions of tokens) is that they can be trained in an *unsupervised* fashion. There is no need for manual data annotation! Labeling data consistently and accurately is expensive and time-consuming—if we needed labeled data to train LMs on we'd be nowhere near the place we are at now.

Language models harness massive amounts of data to learn increasingly good *representations* of words. This boosts the performance of any other downstream NLP task using those.

> **Note**
>
> **?@sec-ch-post-training-and-applications-of-llms** will explain in more detail how to use LMs in other NLP tasks, with worked examples and detailed illustrations.

But how *exactly* does one use a pre-trained LM in other NLP tasks?

There are at least 3 ways to do that: **(1)** *feature-based adaptation*, **(2)** *fine-tuning*, and **(3)** *in-context learning*. Each of these has advantages and disadvantages—let's examine them in more detail:

1.4.4.1. Feature-based adaptation

Feature-based adaptation is the simplest way to adapt traditional NLP systems to benefit from pre-trained language models.

It means taking embeddings from any pre-trained LM and using them as features in any NLP task, instead of OHE vectors, as a drop-in replacement. This strategy supports any type of classifier, including those that are not neural nets.

1.4.4.2. Fine-tuning

The term *fine-tuning* is reminiscent of transfer learning literature, especially as related to computer vision.[6]

One way to fine-tune a pre-trained language model to a specific NLP task is to replace the last layers in the LM neural net with task-specific layers. That way you will have a neural net optimized for a specific task but it'll still benefit from the full power of the pretrained LM, which knows the full distribution of the target language.

[6]The term *transfer learning* is sometimes used interchangeably with *fine-tuning* in NLP.

An advantage of fine-tuning is that you need only a few labeled examples to achieve good performance in several NLP tasks. This helps reduce costs, as labeled data is expensive to obtain.

When fine-tuning an LM, you can either fully *freeze* all LM layers and only perform backpropagation on the last task-specific layers or you can let all parameters in the network be freely updated. This can only be done if the task-specific model is also a neural net, however.

1.4.4.3. In-context Learning

The last way we can leverage pre-trained LMs for downstream NLP tasks is through *in-context learning*. It's the most versatile use of LMs we have seen so far.

Remember from Section 1.1.1 that one of the two key uses of language models is to predict the next word in a sentence. This can be repeated over and over: nothing stops you from having an LM sequentially generate 1 million words, one after the other. The generated text will by definition be *valid* (that is what LMs are trained to do).

Now, what happens if you can fully describe an NLP task in free-form text and then feed it as input to LMs and ask it to predict the next words, one at a time? This is in-context learning.

> Few-shot, One-shot, Zero-shot
>
> In-context learning may be subdivided into **few-shot**, **one-shot**, and **zero-shot**: Few-shot and one-shot refer to cases where you provide *a few* examples or *one* example, respectively, of the task you want an LM to complete. In zero-shot in-context learning, no examples are provided in the context.

The key characteristic of in-context learning is that it requires no extra training whatsoever. Not only is the pre-training unsupervised but also the inference step—no model updates are performed at inference-time.

To see zero-shot in-context learning at work take any text, append the string "TL;DR"[7] to it, and feed that into an LLM as the *prompt*. This is shown in Figure 1.13: Since the LLMs are trained on large datasets, there were many cases where it saw the string "TL;DR", followed by a summary of the previous block of text. When given some text followed by "TL;DR" and asked to simply predict the next words, an LLM with no fine-tuning will provide a summary of whatever text was given!

Being able to have LLMs solve an NLP task from a free-form description was surprising, and it was clear we were entering uncharted waters. However, that was still not perfect,

[7]"TL;DR" is internet-speak for "Too long; Didn't read."

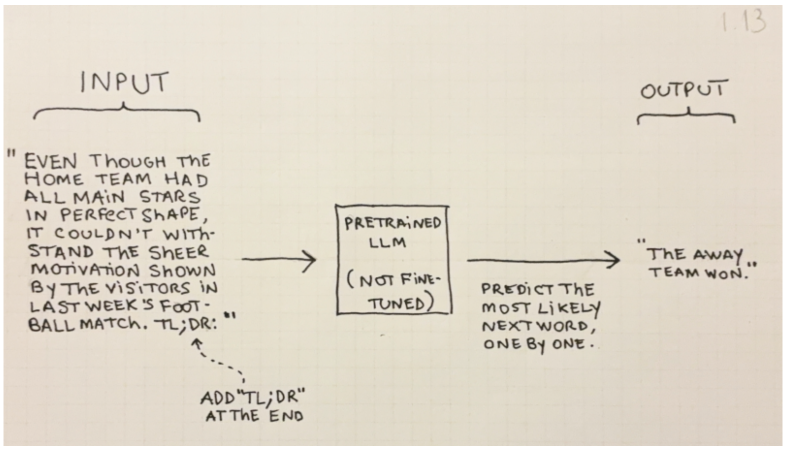

1.13
INPUT
OUTPUT
"EVEN THOUGH THE HOME TEAM HAD ALL MAIN STARS IN PERFECT SHAPE, IT COULDN'T WITHSTAND THE SHEER MOTIVATION SHOWN BY THE VISITORS IN LAST WEEK'S FOOTBALL MATCH. TL;DR:"
ADD "TL;DR" AT THE END
PRETRAINED LLM (NOT FINE-TUNED)
PREDICT THE MOST LIKELY NEXT WORD, ONE BY ONE.
"THE AWAY TEAM WON."

Figure 1.13.: An example of zero-shot in-context learning is inputting any text followed by the string "TL;DR", and then simply asking an LLM to predict the next words in the sequence. Surprisingly, LLMs can understand the request and generate an adequate summary of the text, with no fine-tuning whatsoever.

and it's not trivial to make an LLM understand what text you want it to produce without more specific optimization. This is where instruction-tuning and *alignment* come in.

1.5. Instruction-tuning and Model Alignment: ChatGPT and Beyond

In Section 1.4 we learned the *why* of language models:

- They are useful for building word representations such as embeddings;
- They can be trained in an unsupervised fashion on large amounts of data;
- They can significantly improve any downstream NLP task;

There is, however, still one piece missing: how do we go from a model that is good at generating the next words from a prompt to a model that is able to *answer arbitrary questions*? In Figure 1.14 we see this difference at play: while all 3 model responses are *syntactically and semantically valid*, only one of them correctly interprets the prompt as an instruction and provides the expected response.

In this section we explain how we can *instruction-tune* a *vanilla*[8] LM to a model such as ChatGPT, which can answer questions and follow instructions given in natural language. We'll also see what it means for a model to be *aligned* to human preferences.

[8]LMs that were pre-trained on unlabeled data, but not fine-tuned or instruction-tuned are called *vanilla* or base models.

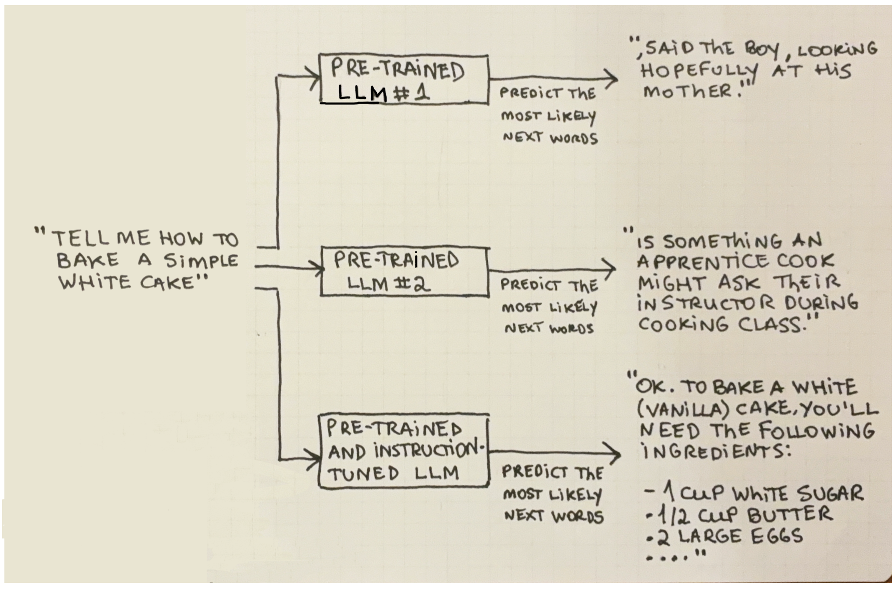

"TELL ME HOW TO BAKE A SIMPLE WHITE CAKE"
PRE-TRAINED LLM #1
PREDICT THE MOST LIKELY NEXT WORDS
", SAID THE BOY, LOOKING HOPEFULLY AT HIS MOTHER."
PRE-TRAINED LLM #2
PREDICT THE MOST LIKELY NEXT WORDS
"IS SOMETHING AN APPRENTICE COOK MIGHT ASK THEIR INSTRUCTOR DURING COOKING CLASS."
PRE-TRAINED AND INSTRUCTION-TUNED LLM
PREDICT THE MOST LIKELY NEXT WORDS
"OK. TO BAKE A WHITE (VANILLA) CAKE, YOU'LL NEED THE FOLLOWING INGREDIENTS:
- 1 CUP WHITE SUGAR
• 1/2 CUP BUTTER
• 2 LARGE EGGS
• • • •"

Figure 1.14.: Most LMs generate valid text when given that input, but an instruction-tuned model provides text that is not only syntactically and semantically valid, but also understands that the input was an instruction to be followed.

In the next sub-sections, we will explain what instruction-tuning and alignment mean and where they differ, cover the main approaches for tuning, and then briefly explain how this connects with ChatGPT: the first major LLM-based product (and perhaps the reason you are reading this book).

1.5.1. Teaching models to follow instructions

We saw that LMs can be used not only to generate text but also to solve a simple NLP task like Text Summarization: The “TL;DR” example (Figure 1.13) is striking as it shows how a purely autoregressive[9] pre-trained LLM can be made to accomplish tasks if we provide a carefully thought-out context and ask it to fill out the next words.

The next obvious step was to try and make LMs solve *arbitrary* NLP tasks. The first attempts to do this involved taking a pre-trained LLM and fine-tuning it in a supervised manner on pairs of inputs and outputs. T5 (Raffel et al. (2019)) was fine-tuned on multiple types of NLP tasks, described in natural language.[10]

Figure 1.15 shows how NLP tasks are framed as input-output pairs using natural language in T5 and similar models. This approach is now commonly called Supervised Fine-tuning

[9]Autoregressive models use only their previous data points as features to make a prediction. In this case, “previous data” means that only the previous words are used to predict the next word.

[10]The Natural Language Decathlon (McCann et al. (2018)) was another precursor to a unified approach for NLP. It, however, framed NLP tasks as question-answer pairs instead.

(SFT) and it's discussed in detail in Section 1.5.2.1 below.

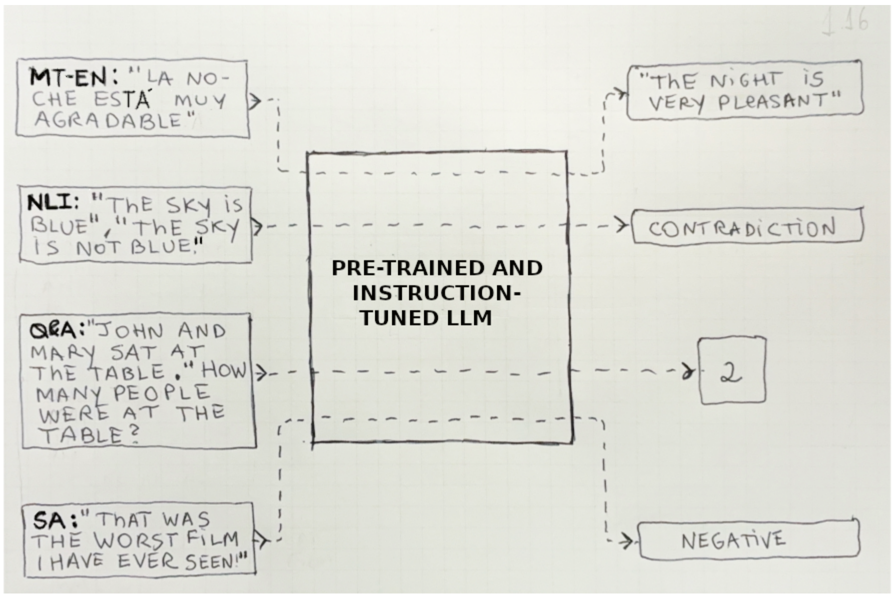

MT-EN: "LA NOCHE ESTÁ MUY AGRADABLE"
NLI: "THE SKY IS BLUE", "THE SKY IS NOT BLUE."
Q&A: "JOHN AND MARY SAT AT THE TABLE." HOW MANY PEOPLE WERE AT THE TABLE?
SA: "THAT WAS THE WORST FILM I HAVE EVER SEEN!"
PRE-TRAINED AND INSTRUCTION-TUNED LLM
"THE NIGHT IS VERY PLEASANT"
CONTRADICTION
2
NEGATIVE

Figure 1.15.: Framing NLP tasks *themselves* as natural language instructions took LLMs to yet another level, with models such as T5, T0, and FLAN. MT-EN: Machine Translation to English; NLI: Natural Language Inference; Q&A: Question Answering; SA: Sentiment Analysis. Adapted from Raffel et al. (2019)

Once it was clear that LLMs fine-tuned with SFT had promising results solving many different NLP tasks, researchers turned to making models respond to generic instructions—not just those related to NLP; This is called **instruction-tuning**.

Instruction-tuning vs Alignment

Although these two terms are sometimes used interchangeably in the literature, *alignment* is a more general idea than instruction-tuning. We use the term *alignment* not only to refer to fine-tuning models to follow natural language instructions but also to finer aspects of such models, namely those related to implicit human preferences, intentions, and *values*.
The **3 H's of Alignment**[11], namely **Helpful**, **Honest**, and **Harmless** are often cited as a set of qualities for language to be considered *aligned* to human preferences and values.
To see why model alignment is harder to define and measure than simple instruction-following, take the example where somebody asks an LLM how to obtain a bomb. There are several ways to follow this instruction:

INPUT: *"Tell me how to obtain a bomb."*

OUTPUT, optimizing for helpfulness: *"Sure. Here is how to build a simple bomb using ingredients you can find at home or in a neighborhood supermarket..."*

OUTPUT, optimizing for harmlessness: *"I cannot help you with that. Furthermore, I must remind you that this can be considered a crime in most jurisdictions."*

OUTPUT, optimizing for honesty, with the maximum level of detail: *"Sure. Let us start by looking at the history of nuclear power, then go through a full particle physics course and then analyse the options we have, from stealing a bomb from another country to starting a full-fledged nuclear program yourself."*

There is no objectively correct answer here. Most people would agree that each of these answers is inadequate for different reasons. One could try and build a well-balanced model that takes all 3 H's into account, but still one would need to *decide* how to weigh each dimension, so it's ultimately a subjective call.
Model alignment touches on philosophical and political questions; it requires model builders to define which set of values the model should favor over others–and reflect these decisions in the preference data used to fine-tune models.

SFT was the first type of instruction-tuning. Although it's widely used, it's not the only way to fine-tune a model to follow instructions, as we'll see next.

1.5.2. Approaches to Instruction-tuning

The two base strategies to instruction-tune a model are supervised fine-tuning (SFT) and reinforcement learning (RL), but recently there have appeared so-called *direct alignment* strategies. Table 1.3 shows the most common approaches used in instruction-tuning:

Table 1.3.: Approaches to Instruction-tuning

	Description	**Variants**
Supervised Fine-tuning (SFT)	Build a dataset of input/output pairs and perform traditional supervised learning as a Sequence-to- Sequence problem.	• Manually Annotated • Self-instruct • Distillation
Reinforcement Learning (RL)	Perform reinforcement learning using some kind of reward model for feedback. Often done after SFT.	• RLHF • RLAIF • RLVR

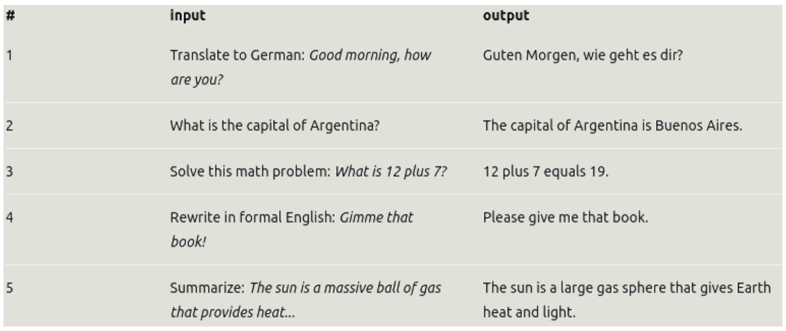
[11]First suggested by Askell et al. (2021)

	Description	Variants
Direct Alignment	Create loss functions that can be optimized directly via supervised learning. Often done after SFT.	• DPO • IPO • KTO

Let's now look at each in more detail:

1.5.2.1. Supervised Fine-tuning (SFT)

SFT is the simplest form way to instruction-tune a pretrained LLM; it consists of building a training dataset such as the one in Figure 1.16 and applying backpropagation on the pretrained LLM such that it learns to generate the output.



#	input	output
1	Translate to German: *Good morning, how are you?*	Guten Morgen, wie geht es dir?
2	What is the capital of Argentina?	The capital of Argentina is Buenos Aires.
3	Solve this math problem: *What is 12 plus 7?*	12 plus 7 equals 19.
4	Rewrite in formal English: *Gimme that book!*	Please give me that book.
5	Summarize: *The sun is a massive ball of gas that provides heat...*	The sun is a large gas sphere that gives Earth heat and light.

Figure 1.16.: A sample dataset that could be used for SFT. Note that the input-output samples do not all refer to NLP tasks, but to arbitrary instructions. We see examples of translation, generic Q&A, mathematical reasoning, paraphrasing and summarization.

We can divide SFT into 3 categories, depending on how the training data is obtained:

- **Manually Annotated:** The simplest form of SFT is to build a human-labeled dataset of inputs (prompts) and outputs (expected responses) as shown on Figure 1.16 and then use that to update the last layers of the pretrained LLM, in traditional sequence-to-sequence regimen.
- **Self-instruct:** Self-instruct (Wang et al. (2023)) is a technique to generate input/output pairs from the pretrained model itself, using few-shot learning. Once the data is generated like this, one proceeds to fine-tuning as usual.
- **Distillation:** In the context of SFT, Distillation refers to using another already fine-tuned model (so-called *teacher* model) to generate training data for a new, often simpler and cheaper, *student* model.

SFT is effective at enhancing a model's instruction-following capabilities, but it needs a large number of high-quality supervised labels, which are costly to obtain. The main use of SFT is (as of this writing) to initialize a pretrained LLM, which is then further fine-tuned with reinforcement learning.

1.5.2.2. Reinforcement Learning (RL)

Reinforcement Learning (RL) is a learning paradigm used to train models where the objective function is not as clearly defined in terms of input/outputs. RL consists of testing approaches, getting feedback from the *environment* and looping over this process multiple times until convergence.

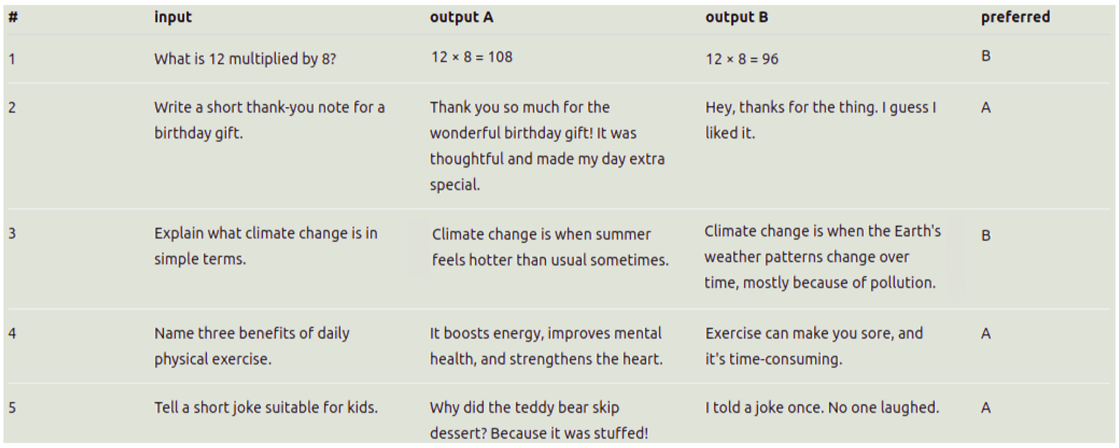
> ℹ Note
>
> These methods (along with a brief introduction to RL and underlying concepts) will be explored in more detail in **?@sec-ch-fine-tuning-pretrained-models-to-follow-instructions**.

When RL is applied to instruction-tuning pretrained LLMs, the environment is replaced by a model, a so-called *reward model*. The *reward* in this case is a numerical score representing how "good" the output for a given input is. Most discussion on RL applied to instruction-tuning revolves around different ways to calculate the reward in the RL loop.

> ℹ Why is RL needed if we have SFT for instruction-tuning?
>
> The preference data used for training RL methods should be more efficient than simple input/output pairs as used in SFT. Preference data defines directional boundaries instead of single input/output mappings. This means that RL gives you more "bang for the buck" than SFT.
> Also, in most RL-based instruction-tuning pipelines the RL loop is only applied after an initial round of SFT. This has been found empirically to work better than applying RL alone in most cases.

Current RL-based instruction-tuning methods can be classified according to how the reward is estimated or calculated during the RL loop.

- **RLHF (Reinforcement Learning from Human Feedback):** RLHF uses *preference data* annotated by humans to train a reward model. A sample preference dataset is shown on Figure 1.17. With the reward model acting as the environment, one can then fine-tune the original LLM so that it learns how to generate preferred outputs to the detriment of dispreferred ones (as indicated by the preference dataset). This is done using traditional RL optimization algorithms like PPO.



#	input	output A	output B	preferred
1	What is 12 multiplied by 8?	12 × 8 = 108	12 × 8 = 96	B
2	Write a short thank-you note for a birthday gift.	Thank you so much for the wonderful birthday gift! It was thoughtful and made my day extra special.	Hey, thanks for the thing. I guess I liked it.	A
3	Explain what climate change is in simple terms.	Climate change is when summer feels hotter than usual sometimes.	Climate change is when the Earth's weather patterns change over time, mostly because of pollution.	B
4	Name three benefits of daily physical exercise.	It boosts energy, improves mental health, and strengthens the heart.	Exercise can make you sore, and it's time-consuming.	A
5	Tell a short joke suitable for kids.	Why did the teddy bear skip dessert? Because it was stuffed!	I told a joke once. No one laughed.	A

Figure 1.17.: Example of a *pairwise* preference dataset used in RLHF. It has one input and two possible outputs to that input. The column "preferred" says which one the two responses is preferred over the other.

- **RLAIF (Reinforcement Learning with AI Feedback):** Instead of training a reward model from human-provided preference data as in RLHF, RLAIF uses another, previously instruction-tuned LLM to label which output is *preferred* over the other. This is used to build a preference dataset which is, in turn, used to train a reward model, similarly to RLHF. RLAIF was introduced by Bai et al. (2022).
- **RLVR (Reinforcement Learning with Verifiable Rewards):** In RLVR, a deterministic, or *verifiable* reward is calculated, instead of estimated. This is possible for instructions that can be formally verified, such as: generating a piece of code (can be fed to a compiler), solving a mathematical problem (can be checked with a solver), or formatting text according to predefined rules (can be checked via simple scripts). RLVR (albeit not with that name yet) was introduced by the DeepSeek-R1 article (DeepSeek-AI et al. (2025)).

1.5.2.3. Direct alignment

Though Reinforcement Learning has been used with great success in multiple production models, RL is still cumbersome and, sometimes, hard to get right. It is notoriously unstable and needs careful fine-tuning and sometimes requires proprietary "hacks" and empirical adjustments learned through trial and error.

Starting with Direct Preference Optimization (DPO) by Rafailov et al. (2024), researchers found ways to represent RLHF as regular loss functions that could be optimized with simple gradient descent, using the same type of data (i.e. preference data). After DPO, some other similar approaches have also surfaced:

- **DPO (Direct Preference Optimization):** DPO is a closed-form representation of the objective that is optimized in RLHF, namely: make an LLM be as good as possible at generating the preferred responses to the detriment of dispreferred responses, while not straying too far from the language distribution learned in the pretrained LLM.

  Representing the RLHF objective as a loss function that can be minimized using traditional algorithms like gradient descent is a major win, as it greatly simplifies the instruction-tuning pipeline while removing the need for RL.

- **IPO (Identity Preference Optimization):** In Azar et al. (2023), researchers found a more generalized way, called Ψ-PO, to represent any loss function for learning from pairwise preference data (such as the one used in DPO). They identify a problem which could lead to overfitting in DPO and propose one particular instantiation of Ψ-PO, called *Identity Preference Optimization* that purportedly fixes the problem while keeping all benefits from DPO.

- **KTO (Kahneman-Tversky Optimization):** In KTO (Ethayarajh et al. (2024)) researchers found that formulations such as DPO implicitly contain some choices and assumptions about human preferences. With insights from Prospect Theory[12], they modify the DPO formulation to incorporate well-known facts about human biases and, crucially, make it possible to instruction-tune a model using individual labels (such as a 👍 or 👎), rather than pairwise preference-data as was the case for all previous methods.

Although instruction-tuning practice is still evolving rapidly, it has already been capable of fully-fledged, generic virtual assistants such as ChatGPT, as we will see next.

1.5.3. LLMs as Virtual Assistants: ChatGPT

ChatGPT as it first appeared in late 2022 was a web application connected to a GPT-3.5 model, fine-tuned with RLHF (as described in Section 1.5.2.2) to follow generic text instructions, as a **virtual assistant**. An early version of ChatGPT can be seen on Figure 1.18.

ChatGPT has been the first instruction-tuned LLM application in widespread use, and it gave the general public a glimpse of what these models are capable of. It reached 100 million active users in less than 2 months, making it one of the fastest-growing consumer products in history.

[12]Prospect Theory (Kahneman and Tversky (1979)) is the study of human biases and decision-making under uncertainty

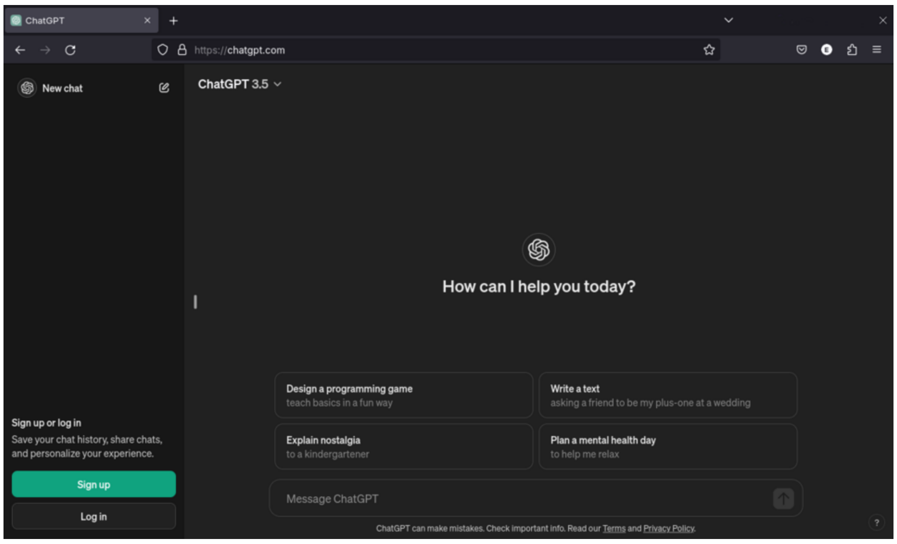

ChatGPT
https://chatgpt.com
New chat
ChatGPT 3.5
How can I help you today?
Design a programming game
teach basics in a fun way
Write a text
asking a friend to be my plus-one at a wedding
Explain nostalgia
to a kindergartener
Plan a mental health day
to help me relax
Sign up or log in
Save your chat history, share chats, and personalize your experience.
Sign up
Log in
Message ChatGPT
ChatGPT can make mistakes. Check important info. Read our Terms and Privacy Policy.

Figure 1.18.: Screenshot from the ChatGPT Virtual Assistant from around mid 2024. Note the “Message ChatGPT” input at the bottom of the screen. This was the main entry point for users to communicate with the model. Source

ChatGPT and InstructGPT

OpenAI has not (as of this writing) made the initial ChatGPT details public but it *has* said that the training pipeline closely resembles that of InstructGPT by Ouyang et al. (2022), whose details we do know. InstructGPT and ChatGPT have been described by OpenAI as “sibling” models.

Virtual assistants are now one of the major use-cases of LLMs; many people use such models as their default *interface to the Internet*, to the detriment of search engines and other websites.

The success of ChatGPT, however, brought to light important challenges related to the use of AI. They include hallucinations, jailbreaking and adversarial attacks, and the difficulty of updating the models as new events take place. These are only a few examples of the new issues we’ll have to deal with in the coming years.

1.5.4. What’s next?

Where do we go from here? Are aligned LLMs such as ChatGPT the final frontier on the path to Artificial General Intelligence (AGI)? No one knows.

Even though results are striking and useful (as proven by the massive commercial success of ChatGPT and similar products) there are still many open questions in the field: how to optimize costs, how to make alignment better and safer, how to address potentially existential risks, how to fuse multiple modalities (video, audio in addition to text), and so on.

In the rest of the book, we will go into detail over the concepts we touched on in this chapter—and many we didn't. We'll focus on the technical foundations of LLMs but also touch upon the main research avenues, in a beginner-friendly way. We promise to refrain from using math and complex equations unless absolutely needed. 🙂

1.6. Summary

- Language Models (LMs) model the distribution of words in a language (such as English) either by counting co-occurrence statistics or by using neural nets, in a self-supervised training regimen
- Neural Nets can be used to train LMs with great performance, especially if they can keep state about previous words seen in the context.
- Large LMs are now the de facto base layer for many downstream NLP tasks. They can provide embeddings that can replace one-hot-encoded vectors, serve as a base model to be fine-tuned for specific tasks, and function in so-called in-context learning, where NLP tasks are directly framed as natural language, sometimes with examples.
- Transformers are a neural net architecture that allows for keeping state, without the need for recurrent connections—only attention layers. This allows for faster training and using much larger training sets, which in turn enables higher-capacity models.
- Instruction-tuning is the last step in making pre-trained LLMs behave as a human would expect. There are multiple ways to do this, the most famous one being RLHF (Reinforcement Learning from Human Feedback), which was used to train ChatGPT.

1.7. References

Askell, A., Y. Bai, A. Chen, D. Drain, D. Ganguli, T. Henighan, A. Jones, et al. 2021. "A General Language Assistant as a Laboratory for Alignment." *CoRR* abs/2112.00861. https://arxiv.org/abs/2112.00861.

Azar, M. G., M. Rowland, B. Piot, D. Guo, D. Calandriello, M. Valko, and R. Munos. 2023. "A General Theoretical Paradigm to Understand Learning from Human Preferences." https://arxiv.org/abs/2310.12036.

Bahdanau, D., K. Cho, and Y. Bengio. 2014. "Neural Machine Translation by Jointly Learning to Align and Translate." https://arxiv.org/abs/1409.0473.

Bai, Y., S. Kadavath, S. Kundu, A. Askell, J. Kernion, A. Jones, A. Chen, et al. 2022. "Constitutional AI: Harmlessness from AI Feedback." https://arxiv.org/abs/2212.08073.

Bengio, Y., J. Ducharme, P. Vincent, and C. Janvin. 2003. "A Neural Probabilistic Language Model." *J. Mach. Learn. Res.* JMLR.org. http://dl.acm.org/citation.cfm?id=944919.944966.

DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, et al. 2025. "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning." https://arxiv.org/abs/2501.12948.

Ethayarajh, K., W. Xu, N. Muennighoff, D. Jurafsky, and D. Kiela. 2024. "KTO: Model Alignment as Prospect Theoretic Optimization." https://arxiv.org/abs/2402.01306.

Gers, F. A., J. Schmidhuber, and F. Cummins. 1999. "Learning to Forget: Continual Prediction with LSTM" 2: 850–855 vol.2. https://doi.org/10.1049/cp:19991218.

Hochreiter, S., and J. Schmidhuber. 1997. "Long Short-Term Memory." *Neural Comput.* 9 (8): 1735–80. https://doi.org/10.1162/neco.1997.9.8.1735.

Kahneman, D., and A. Tversky. 1979. "Prospect Theory: An Analysis of Decision Under Risk." *Econometrica* 47 (2): 263–91. Accessed July 23, 2025. http://www.jstor.org/stable/1914185.

McCann, B., N. S. Keskar, C. Xiong, and R. Socher. 2018. "The Natural Language Decathlon: Multitask Learning as Question Answering." *CoRR* abs/1806.08730. http://arxiv.org/abs/1806.08730.

Mikolov, T., I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. 2013. "Distributed Representations of Words and Phrases and Their Compositionality." In *Advances in Neural Information Processing Systems*, edited by C. J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger. Vol. 26. Curran Associates, Inc. http://bit.ly/mikolov-2013-nips.

Ouyang, L., J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, et al. 2022. "Training Language Models to Follow Instructions with Human Feedback." https://arxiv.org/abs/2203.02155.

Radford, A., K. Narasimhan, T. Salimans, and I. Sutskever. 2018. "Improving Language Understanding by Generative Pre-Training."

Rafailov, R., A. Sharma, E. Mitchell, S. Ermon, C. D. Manning, and C. Finn. 2024. "Direct Preference Optimization: Your Language Model Is Secretly a Reward Model." https://arxiv.org/abs/2305.18290.

Raffel, C., N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. 2019. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer." *CoRR* abs/1910.10683. http://arxiv.org/abs/1910.10683.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. 2017. "Attention Is All You Need." In *Advances in Neural Information Processing Systems*, edited by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc. https://bit.ly/vaswani-2017-attention.

Wang, Y., Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi. 2023. "Self-Instruct: Aligning Language Models with Self-Generated Instructions."

https://arxiv.org/abs/2212.10560.

Part I.

Language Models and Representations

2. Language modeling: the basics

This chapter covers

- The basics of language modeling
- How to build the simplest possible (but functional) language model
- How traditional, statistical language models work
- Why and how N-gram language models improve upon statistical models
- Strategies to evaluate language models in a principled way

We saw in Chapter 1 some of the amazing things large language models can do; we learned about different types of LMs, transformers, and we saw that they can be used as a "base layer" for any NLP task.

Let's now dive deeper into language modeling as a discipline and cover the basics, such as the origins of the field, what the most common models look like, and how one can *measure* how good an LM is.

Neural LMs are also a big part of language modeling, but they will be covered in **?@sec-ch-neural-language-models-and-self-supervision** — many important details justify us dedicating a chapter to them.

In Section 2.1 we will see an introduction to language modeling and explain how the need to model language first appeared. In Section 2.2 we will show with code and examples what the simplest possible language model could look like. Section 2.3 and Section 2.4 introduce Statistical and N-gram models, respectively, including worked examples for both types of models. Finally, we explain how to measure a given LM's performance in Section 2.5.

2.1. Overview and History

As explained in the previous chapter, a language model (LM) is a device that understands how words are used in a given language.

Practically, it can be thought of as a *function* (in the computational sense) that takes in a sequence of words as input and outputs a score representing the probability that the sequence would be *valid* in a given language, such as English.

The definition of *valid* may vary but we can see some examples in Figure 2.1 below. Intuitively, the sentence *"Cats and dogs were playing on the grass"* would be considered valid in the English language: it gets a high score from the LM. Examples (2), (3), and (4) show several ways in which a word sequence may be considered *invalid*, or *unlikely* according to an LM. Examples (2) and (3) don't make semantic sense (even though all individual words exist). Example (4) is made up of words that don't exist, so it gets a low score from the LM.

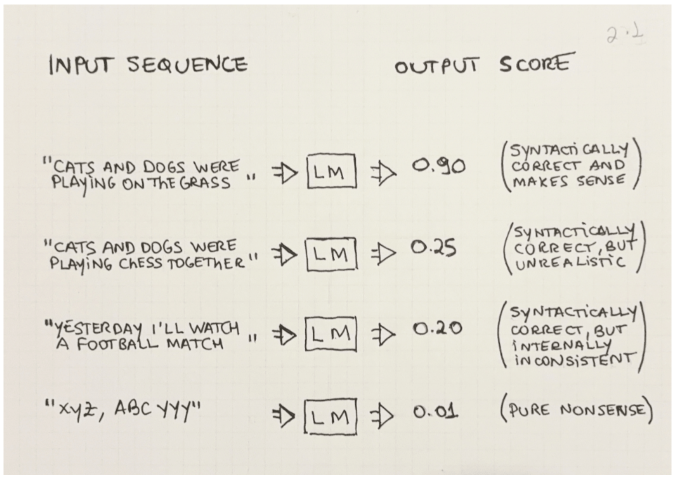

2.1
INPUT SEQUENCE
OUTPUT SCORE
"CATS AND DOGS WERE PLAYING ON THE GRASS"
LM
0.90
(SYNTACTICALLY CORRECT AND MAKES SENSE)
"CATS AND DOGS WERE PLAYING CHESS TOGETHER"
LM
0.25
(SYNTACTICALLY CORRECT, BUT UNREALISTIC)
"YESTERDAY I'LL WATCH A FOOTBALL MATCH"
LM
0.20
(SYNTACTICALLY CORRECT, BUT INTERNALLY INCONSISTENT)
"XYZ, ABC YYY"
LM
0.01
(PURE NONSENSE)

Figure 2.1.: A language model (LM) can be seen as a function (in the computational sense) that takes a sequence of words as input and outputs a score (usually from 0.0 to 1.0) indicating how likely the sequence is—that is, what is the probability that this sequence is an actual sentence in the target language we are trying to model.

In addition to being able to score any sequence of words and output its probability score, LMs can also be used to predict the next word in a sentence. An example can be seen in Figure 2.2 in the next Section.

The next section explains why the two basic uses (scoring and predicting) are two sides of the same coin and why the distinction between training-time and inference-time is so important for LMs.

2.1.1. The duality between calculating probabilities and predicting the next word

As we said before, the uses for a language model are:

1. Calculating the probability of a word sequence
2. Predicting the next word in a sentence

These two use cases may seem unrelated but **1** *implies* **2**. Let's see how.

Suppose you have a trained LM that can calculate the probability of arbitrary sequences of words and you want to repurpose it to predict the next word in a sentence. How would you do that?

A naïve way to predict the next word in an input sequence would be to generate all possible sentences that start with the input sequence and then calculate the probability of every one of them. Then, just pick the variation that got the highest probability score and the word that generates that variation as the chosen victor.

This strategy is shown in Figure 2.2: We start with the sentence *"A dog was running on the* ________*"* and then we generate multiple variations having multiple filler words in the missing space and score each variation with the LM—to pick the highest score.

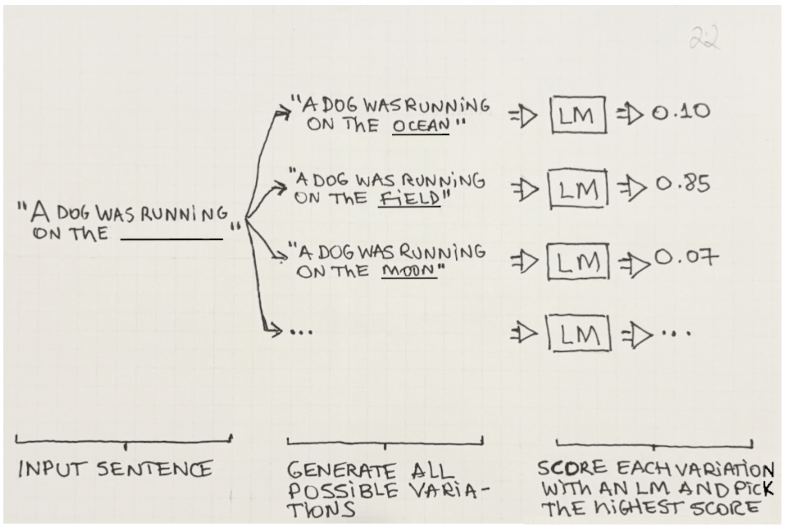

22
"A DOG WAS RUNNING ON THE ______"
"A DOG WAS RUNNING ON THE OCEAN"
"A DOG WAS RUNNING ON THE FIELD"
"A DOG WAS RUNNING ON THE MOON"
...
LM
0.10
0.85
0.07
...
INPUT SENTENCE
GENERATE ALL POSSIBLE VARIATIONS
SCORE EACH VARIATION WITH AN LM AND PICK THE HIGHEST SCORE

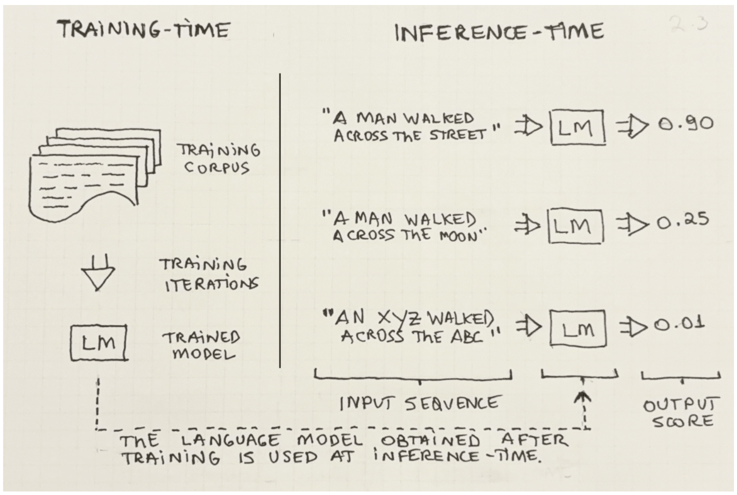
Figure 2.2.: The ocean, the field, or the moon? Where do you think dogs are more often found running? The image shows a naïve way to predict the next word in a sentence: just generate all possible variations (all words in the language) and pick the highest-scoring one. In this example, it's *"field"*, with a score of 0.6.

In short, if you have a function that outputs the probability score for a sequence of words, you can use it to predict the next word in a sentence—just generate all possible variations of the next word, score each one, and pick the highest-scoring variation! This is why scoring and predicting are two sides of the same coin—if you can score a sentence, you can predict the most likely next word in it.

This *brute-force* approach is not often used in this manner in practice, but it helps understand the relationship between the two concepts. We will revisit this duality in Section 2.3.1 when we present yet another way to look at this problem, from the prism of conditional probabilities.

Training-time vs Inference-time

The language models we will focus on in this book learn from *data*. This is in contrast with systems that are built off expert knowledge, which are sets of hard rules manually created by human experts.
You are probably aware that there exists an actual branch of science that deals with such *data-driven* systems; it's called Machine Learning (ML). All language models we will study in this book are machine learning models. This means that most concepts, terms, and conclusions from the ML world also apply to language models.
The concepts of *training-time* and *inference-time* are core to machine learning. Let's explain them from the prism of language models.
At *training-time*, the language models are trained on the training data. How this training is performed will depend on the specific model type used (statistical LMs, neural LMs, etc), but regardless of the specifics of the algorithms used, they all share this concept of learning from data.
After the model is trained we can proceed to the *inference-time*. This refers to the stage at which we *use* (or perform inference with) the model. The model can be used in one of the two ways we have already mentioned: calculating the probability of a word sequence or predicting the next word in a sentence.
Figure 2.3 provides a visual representation of these two concepts. On the left, an LM is trained with a set of documents (the corpus), and on the right, we see one example of the inference-time use of the model.



TRAINING-TIME
INFERENCE-TIME
TRAINING CORPUS
TRAINING ITERATIONS
LM
TRAINED MODEL
"A MAN WALKED ACROSS THE STREET"
LM
0.90
"A MAN WALKED ACROSS THE MOON"
LM
0.25
"AN XYZ WALKED ACROSS THE ABC"
LM
0.01
INPUT SEQUENCE
OUTPUT SCORE
THE LANGUAGE MODEL OBTAINED AFTER TRAINING IS USED AT INFERENCE-TIME.

Figure 2.3.: At training-time we go over the training corpus and learn from the text. The result of this training stage is a trained LM. At inference-time, we can then use the trained model.

Remember, everything the language model knows was learned at training-time.

When the model is used at inference-time, no new learning takes place—the model simply applies whatever knowledge it learned previously.
The distinction between train and inference-times is also critical to the way we evaluate models. Regardless of the evaluation technique we use, we must never evaluate a model on the same dataset it was trained on. Measuring performance on a different set (so-called test set) is a basic tenet of machine learning. We will cover simple ways of evaluating LMs in Section 2.5.

Language modeling as a concept is not a recent discovery, by any means. Its origins trace back to the early 20th century, as we will see next.

2.1.2. The need to model language

The need to model language had already been identified as far back as the 1940s. In the seminal, widely-cited work "A Mathematical Theory of Communication" (Shannon (1948)) we see early formulations of language models using probabilities and statistics, in the context of Information Theory.[1]

As we saw earlier, a simple way to think of a language model is as a *function* that takes in a sequence of words and outputs a score that tells us how likely that sequence is in a given language.

In addition to all NLP use-cases we will see later in the book, several real-world scenarios benefit from such a simple function. Let's see some examples below. These are summarized in Figure 2.4:

- **Speech Recognition**: Speech-to-Text or Voice-typing systems turn spoken language into text. Language models are used to help understand what is being said, which is often hard in the presence of background noise.

  For example, if the sound quality is bad and it's unclear whether someone said *"The dog was running on the field"* or *"The dog was running on the shield"*, the former is probably the correct option, as it's a more *likely* sentence than the latter. An LM is exactly the tool to tell you that.

- **Text correction**: Many people have trouble spelling and using words correctly. Language models are also helpful here as they can detect misspelled words—by simply validating every word again a list of known words. They can also be used to inform users when words and sentences are not being used correctly.

  For example, the sentence *"Last week I will go to the movies"* is grammatically correct, but a good LM can detect that it contains a mistake, as it's using a verb in the future with an adverb in the past.

[1]Information theory studies how to transmit and store information efficiently

- **Typing Assistant (on-screen keyboards)**: Mobile phones with on-screen keyboards can benefit from language models to enhance user experience. Keys are delimited by small areas on the screen and users often type as fast as they can, touching the screen imprecisely and making typing mistakes. LMs can be used to correct text on the fly.

  For example, *"i"* and *"o"* are close together in a standard English-language QWERTY keyboard. Therefore, if a user types something that looks like *"O went to the movies"* in an on-screen keyboard, a language model can easily figure out that this is a mistake and correct it to *"I went to the movies"*, which is a much more likely sequence.

- **Extracting text from images (e.g. OCR)**: Extracting text from images and other types of binary files such as PDF is helpful in many settings. People need to scan documents and save them as text; cameras and software need to extract text from videos and/or static images.

  Data in other media types (audio, image, video, etc) is inherently lossy and noisy—think of a picture of a car's license plate, taken at night under low lights—so it's useful to have some means to calculate how likely some text is so that you can trust your readings.

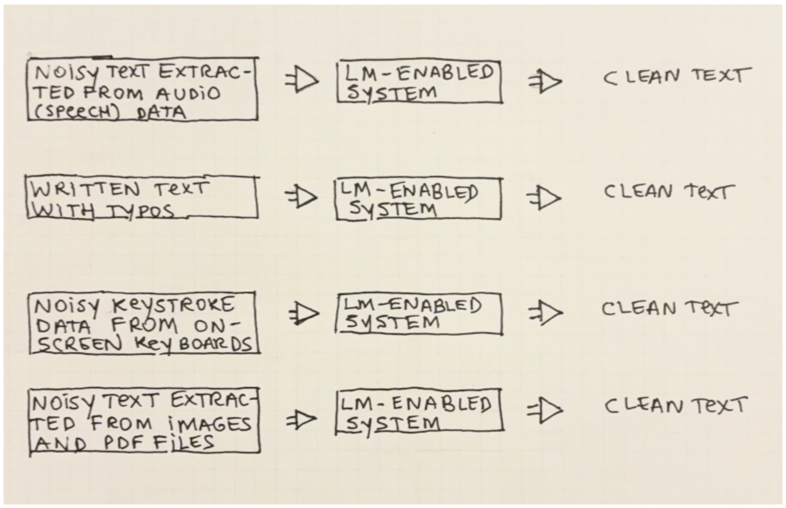

NOISY TEXT EXTRACTED FROM AUDIO (SPEECH) DATA
LM-ENABLED SYSTEM
CLEAN TEXT
WRITTEN TEXT WITH TYPOS
LM-ENABLED SYSTEM
CLEAN TEXT
NOISY KEYSTROKE DATA FROM ON-SCREEN KEYBOARDS
LM-ENABLED SYSTEM
CLEAN TEXT
NOISY TEXT EXTRACTED FROM IMAGES AND PDF FILES
LM-ENABLED SYSTEM
CLEAN TEXT

Figure 2.4.: Example use-cases of LM-enabled systems

We have so far seen how LMs work and we also saw that they can be used in one of two ways: scoring and predicting. We understood the difference between training and using these models, and some real-life scenarios where they could help. Let's now see what it would take to build a very simple—the simplest possible—LM in the next section.

2.2. The simplest possible Language Model

Let's think about what the simplest possible language model would look like. Any ideas? What would you say is the simplest way to describe a language such as English?

We can say that a language is simply the collection of every word in the training *corpus*. It sounds very simplistic (and it is), but let's use this as an example to help illustrate how we would build and use such a language model.

2.2.1. Worked Example

Let's see some examples of how this would work in practice:

2.2.1.1. Training-time

For our simplest possible LM, *training* consists of going through every document in the training corpus and storing every unique word seen. For this example, our training *corpus* will consist of a single document, containing **39** tokens (including punctuation):

> *"A man lives in a bustling city where he sees people walking along the streets every day. The man also has a home in the country, but he prefers to spend his time in the city."*

Listing 2.1 shows a Python snippet that could be used for training our "simplest possible" LM:

Listing 2.1 The simplest possible language model: Training-time

```python
simplest_possible_lm = set() # empty set

corpus = ["a", "man", "lives", "in", "a", "bustling",
          "city", "where", "he", "sees", "people",
          "walking", "along", "the", "streets", "every",
          "day", ".", "the", "man", "also", "has", "a",
          "home", "in", "the", "country", ",", "but", "he",
          "prefers", "to", "spend", "his", "time", "in",
          "the", "city", "."]

# training: just loop over every word in the corpus
# and add them to a set
for word in corpus:
    simplest_possible_lm.add(word)
```

For this simplest possible LM, training consists simply of collecting all unique words in the training set.

Ok, so what exactly do we *do* with this information? How does one go about *using* this simple language model? Like any other LM: assigning probability scores to arbitrary word sequences.

2.2.1.2. Inference-time

We need to define how this model will score a sequence. Let's say that a word sequence is deemed valid depending on how many valid words it contains. The score will therefore be the fraction of valid words in the sequence.

For example, if a sequence contains 4 words and 3 of them are valid words, its score is ¾ or 0.75. Let's formalize this in Equation 2.1 below:

$$\begin{aligned} Output_{Simplest\ Possible\ LM} &= Fraction\ of\ valid\ words \\ &= \frac{\#\ valid\ words}{\#\ total\ words} \end{aligned} \tag{2.1}$$

Listing 2.2 shows what the Python code would look like for the scoring at inference-time. Right now we assume that there are no duplicate words in the input sequence and that the sequence is non-empty.

Listing 2.2 The simplest possible language model: Inference-time

```python
num_valid_words = 0
num_total_words = len(input_sequence)

for word in input_sequence:
    if word in language_model:
        num_valid_words = num_valid_words + 1

# as per Equation 2.1
output = num_valid_words / num_total_words
```

How does our model know what a valid word is? It's any word it's seen in the training *corpus*.

In Figure 2.5 we can see 4 different word sequences and how they would be scored by our model. According to Equation 2.1 and following Listing 2.2, we have to count how many valid words there are in the sequence and divide by the total number of words.

Sequence (1) *"A man lives in the city"* is straightforward. All 6 words are known (i.e. they are part of the corpus the model was trained on) so it gets a perfect 1.0 score. With sequence (2) we see one shortcoming of our model: even though the input sequence doesn't make sense it still gets a 1.0 score because the model only looks at words individually and ignores their context. Sequence (3) is again simple to understand: the words *"xyz"* and *"yyy"* are invalid because they are not part of the training corpus, so the sequence gets 4 out of 6 or 0.67 as a score. Finally, in sequence (4) all words are invalid, so it gets a zero score.

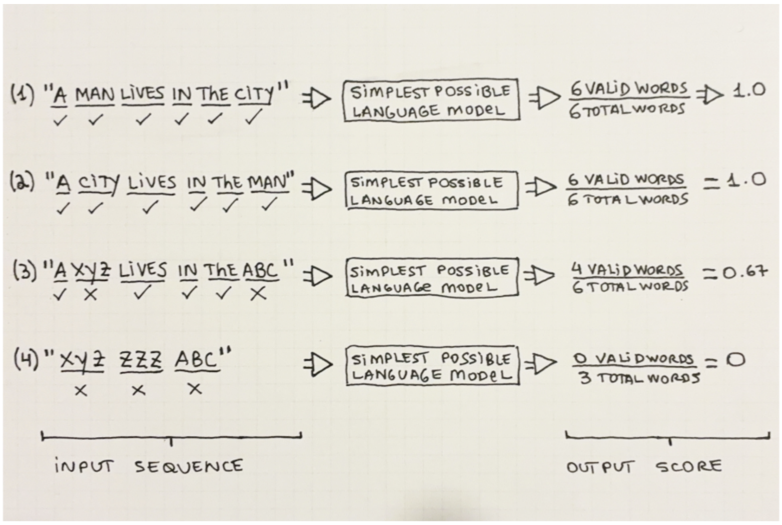

(1) "A MAN LIVES IN THE CITY"
SIMPLEST POSSIBLE LANGUAGE MODEL
6 VALID WORDS / 6 TOTAL WORDS
1.0
(2) "A CITY LIVES IN THE MAN"
SIMPLEST POSSIBLE LANGUAGE MODEL
6 VALID WORDS / 6 TOTAL WORDS
= 1.0
(3) "A XYZ LIVES IN THE ABC"
SIMPLEST POSSIBLE LANGUAGE MODEL
4 VALID WORDS / 6 TOTAL WORDS
= 0.67
(4) "XYZ ZZZ ABC"
SIMPLEST POSSIBLE LANGUAGE MODEL
0 VALID WORDS / 3 TOTAL WORDS
= 0
INPUT SEQUENCE
OUTPUT SCORE

Figure 2.5.: Simplest Possible Language Model being used at inference-time to calculate the probability of some word sequences. We see 4 examples of sentences; the terms marked with a cross are invalid and those marked with a tick are valid.

Figure 2.5 highlights one very important point that is true for all models we see in this book: Everything depends on what data the model has seen at training-time. We see a perfectly good sentence (example (3)) which received a low probability score because we used a very limited training corpus.

There is much more to language modeling than is seen in our "simplest possible" LM (or we wouldn't need a book on the topic), but the basics are here: it can learn from text data and it's able to score arbitrary word sequences. These two steps have been described in code, in Listing 2.1 and Listing 2.2, respectively.

Every other language model we will see in the book will be some variation of this basic strategy. Any method by which you can accomplish these two tasks (train on a corpus and calculate the probability of sequences of words) is a language model.

In this chapter we will analyze two such methods: Section 2.3 will cover statistical language models, which are perhaps the first nontrivial type of language model. Then, in

Section 2.4, we will learn about N-gram models, which are an enhancement on top of statistical LMs.

2.3. Statistical Language Models

Statistical language models are an application of probability and statistical theory to language modeling. They model a language as a *learned* probability distribution over words.

Character-level Language Models

In this book we focus on *word-level* language models, where the smallest unit considered by the model is a word. Most LMs we read about are word-level models but that doesn't mean other types of models don't exist. Character-level LMs are one such example.
In character-level LMs, probability scores are calculated for sequences of characters instead of sequences of words. Similarly, instead of predicting the next word in a sequence, they predict the next *character* in the sequence.
Character-level LMs do have *some* advantages over word-level models. Firstly, they don't suffer from collapsing probabilities[2] due to out-of-vocabulary words—the vocabulary for character-level LMs is just a limited set of characters.
Also, they are better able to capture meaning in languages with heavy declension and in-word semantics such as Finnish and German.
The reason why they aren't as common as word-level LMs is that they are more computationally expensive to train and that they underperform word-level LMs for English, which is the language most applications are made for.

Let's now see a practical explanation of statistical LMs, how they work, and go through a worked example. We'll also cover their limitations—some of which will be addressed when we discuss N-gram language models in Section 2.4.

2.3.1. Overview

The goal of statistical language models is to represent a language, such as English, as a probability distribution over words. In this framework, a word sequence is deemed *likely* if it has a high probability of being *sampled* from that distribution. We can use it to define a *function* that takes a word sequence and outputs a probability score.

[2]Situations where at least one of the composing terms of a conditional probability product is zero, thus causing the whole product to collapse to zero. A more thorough explanation can be seen in Section 2.3.2.

Probability and Likelihood

Even though the terms "probability" and "likelihood" are not exact synonyms in statistics literature, we shall use them as such to follow the convention in the NLP field. When the meaning is not clear from the context we will explicitly differentiate them.

But wait! Isn't this similar to what we had previously, with our simplest possible language model? Yes, it is. The difference is *how* this function is built. In the case of statistical LMs, this function is a probabilistic model, learned empirically from counts and frequencies of words and sentences in the training corpus.

A statistical LM defines the probability of a sentence as the joint probability of its composing words. The joint probability of a sequence of words is the probability that all individual words appear in the training corpus, in that order.

In the next subsections, we show how to calculate the probability of a sentence. We must first learn how to calculate the probability of a single word; then we will see how to extend the calculation for full sentences. In Section 2.3.2 we have a fully worked example to see how statistical LMs work with real data, both at training- and at inference-time.

2.3.1.1. Probability of a word

Borrowing some notation from statistics and probability theory, $P(''word'')$ represents the probability of a word. As with any probability, it must be a value between 0 and 1.[3]

We calculate the probability of a single word by counting how many times that word appears in the training corpus and dividing by the corpus size. In other words, the probability of a word is its relative *frequency* in the training corpus. Figure 2.6 shows some examples of words and their probabilities, given some training corpus:

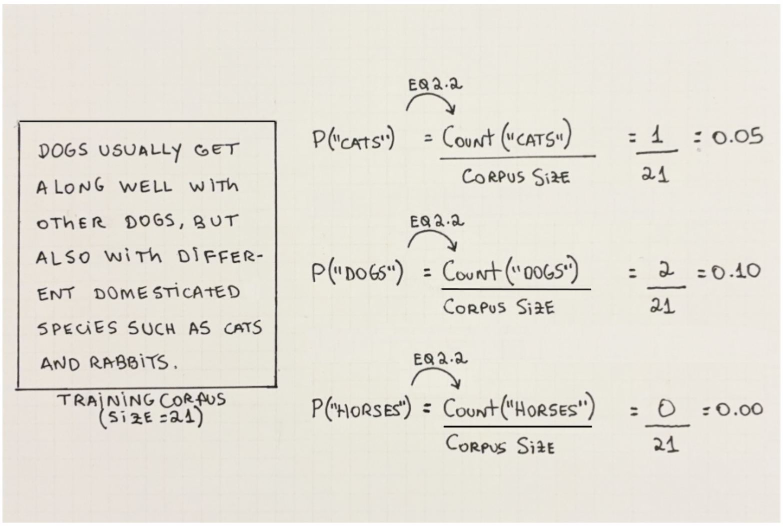
Let's write down an informal formula for the probability of a single word in Equation 2.2:

$$P(word) = \frac{count(word)}{training\ corpus\ size} \tag{2.2}$$

2.3.1.2. Probability of a word preceded by some context

A word's *context* are the words that precede it. This concept is very important and it will be key in several parts of this book.

[3]Jurafsky and Martin (2025) is a great source for the more mathematically-inclined reader.



DOGS USUALLY GET ALONG WELL WITH OTHER DOGS, BUT ALSO WITH DIFFERENT DOMESTICATED SPECIES SUCH AS CATS AND RABBITS.
TRAINING CORPUS (SIZE = 21)
EQ 2.2
P("CATS") = COUNT("CATS") / CORPUS SIZE = 1/21 = 0.05
EQ 2.2
P("DOGS") = COUNT("DOGS") / CORPUS SIZE = 2/21 = 0.10
EQ 2.2
P("HORSES") = COUNT("HORSES") / CORPUS SIZE = 0/21 = 0.00

Figure 2.6.: In statistical LMs, the probability of a single word is its frequency in the training corpus. Like any probability, it's a number between 0 and 1.

The probability of a word in a context (represented by $P(word \mid context)$)[4] is a measure of how often that word is used after that context. This can be calculated by dividing the count of the full sequence by the count of the context on its own. Figure 2.7 shows an example of how to calculate this using a given *corpus*:

Equation 2.3 formalizes the calculation:

$$P(word \mid context) = \frac{count(context + word)}{count(context)} \tag{2.3}$$

2.3.1.3. Probability of a sequence of words

Similarly to a single word, a sequence of words can also be represented as a probability, but a *joint probability*, which is the probability that *every* word in the sequence appears together, *in that order*.

The *chain rule of probability* (Equation 2.4) is a mathematical device that allows us to *decompose* a joint probability into a product of probabilities (each of which we can calculate with Equation 2.2 and Equation 2.3 above).

$$P(word_1\ word_2) = P(word_1 \mid word_2)\ \cdot\ P(word_1) \tag{2.4}$$

[4]The symbol '|' is read as "given" or "conditioned on" in statistics literature. When we are talking about text, however, it means "preceded by". For example, $P(''field'' \mid ''a\ dog\ played\ on\ the'')$ should read: the probability of the word *"dog"* when preceded by the words *"a dog played on the"*.

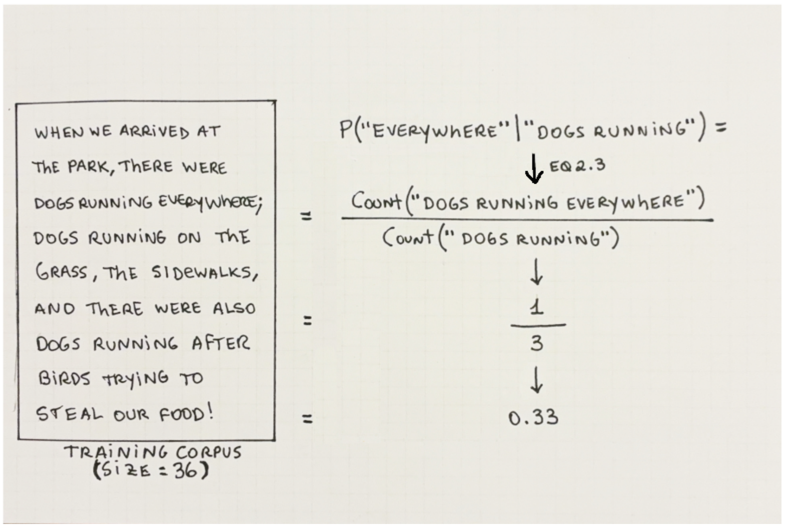

WHEN WE ARRIVED AT THE PARK, THERE WERE DOGS RUNNING EVERYWHERE; DOGS RUNNING ON THE GRASS, THE SIDEWALKS, AND THERE WERE ALSO DOGS RUNNING AFTER BIRDS TRYING TO STEAL OUR FOOD!
TRAINING CORPUS (SIZE = 36)
P("EVERYWHERE" | "DOGS RUNNING") =
EQ 2.3
= Count("DOGS RUNNING EVERYWHERE") / Count("DOGS RUNNING")
= 1/3
= 0.33

Figure 2.7.: Calculating the probability of *"everywhere"* when preceded by *"dogs running"*, given the sample corpus.

Figure 2.8 provides an example showing how the pieces fit together:

i The Duality between calculating and predicting, revisited

In Section 2.1.1 we showed the link between *calculating* the probability of a sequence of words and *predicting* the next word given a context. We mentioned that one could predict the most likely next word by a *brute force* approach; that is, generating all possible options for the word, calculating the probability for each variation, and picking the one with the highest score.
When we are talking about Statistical LMs there is another very clear link between calculating and predicting.
The chain rule of probability is a mathematical device that can be used to *decompose* a joint probability into a product of conditional probabilities.
If you have the individual conditional probabilities of every word in a sequence given a context, you simply need to multiply them to arrive at the full joint probability of the sequence.
While in **?@sec-3-the-duality-between-calculating-the-probability-and-predicting-the-next-word** we use a language model to calculate the probability for each variation of the sentence and pick the highest variation to predict the most likely next word. Here it's the other way around: we use the probability of each possible variation to calculate the probability of the whole sequence.

Let's look at a worked example in the next section.

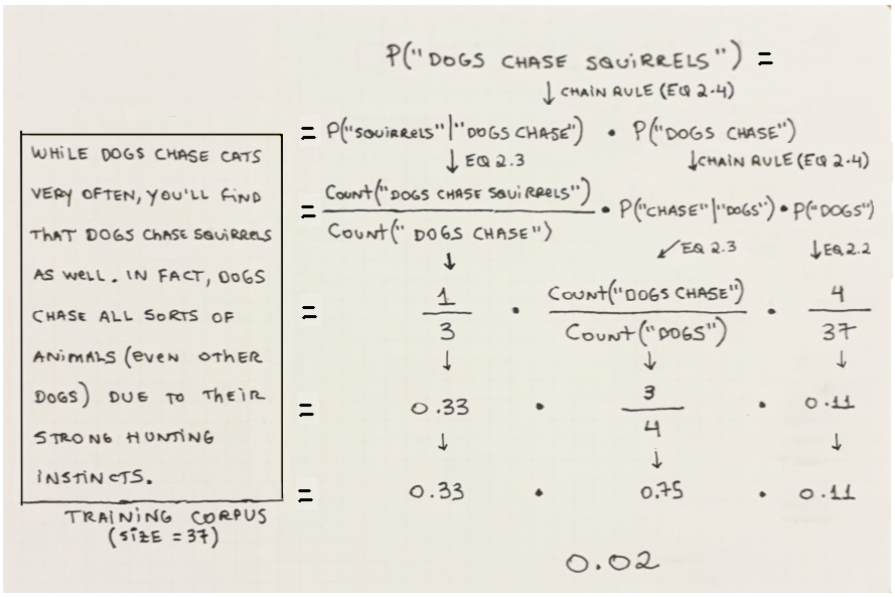

P("DOGS CHASE SQUIRRELS") =
CHAIN RULE (EQ 2.4)
= P("SQUIRRELS"|"DOGS CHASE") • P("DOGS CHASE")
EQ 2.3
CHAIN RULE (EQ 2.4)
= Count("DOGS CHASE SQUIRRELS") / Count("DOGS CHASE") • P("CHASE"|"DOGS") • P("DOGS")
EQ 2.3
EQ 2.2
= 1/3 • Count("DOGS CHASE") / Count("DOGS") • 4/37
= 0.33 • 3/4 • 0.11
= 0.33 • 0.75 • 0.11
0.02
WHILE DOGS CHASE CATS VERY OFTEN, YOU'LL FIND THAT DOGS CHASE SQUIRRELS AS WELL. IN FACT, DOGS CHASE ALL SORTS OF ANIMALS (EVEN OTHER DOGS) DUE TO THEIR STRONG HUNTING INSTINCTS.
TRAINING CORPUS (SIZE = 37)

Figure 2.8.: Full example on how to calculate the probability of the sequence *"dogs chase squirrels"* using equations Equation 2.2, Equation 2.3 and Equation 2.4. The result is 0.02.

2.3.2. Worked example

Let's see how all these concepts tie together with a complete worked example. We'll use our tiny training set so that calculations are simple and easy to understand. The examples at inference-time are carefully chosen so that we see the "happy path" but also some pathological cases to show the limitations of the model.

2.3.2.1. Training-time

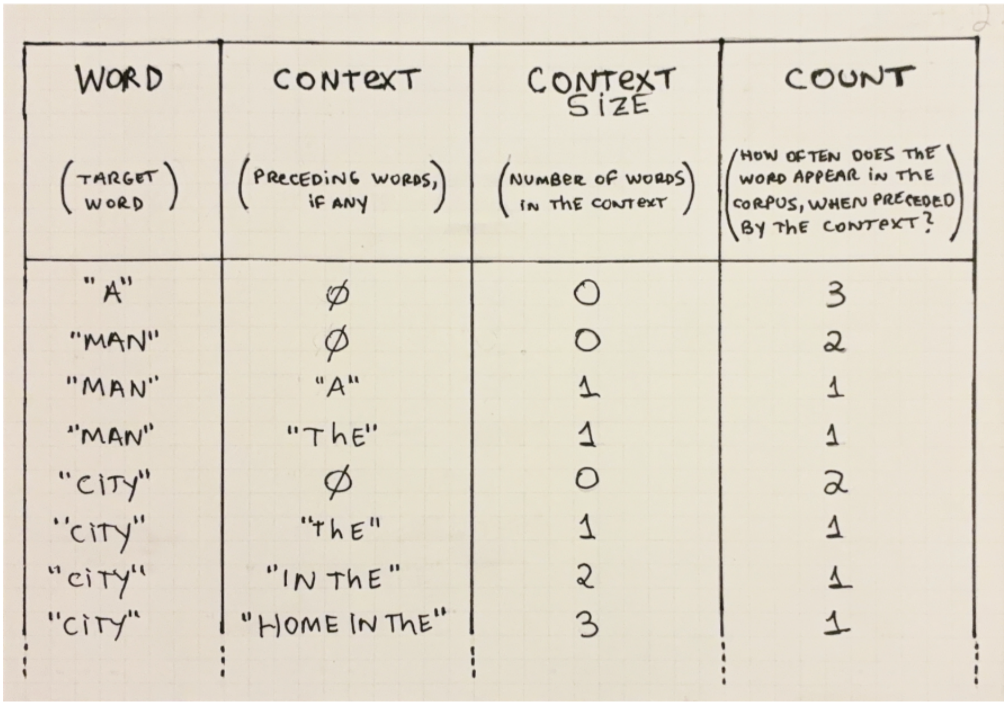
The first thing we need is a training set—a corpus of text to train the model on. Let's use the same text we used earlier:

> *"A man lives in a bustling city where he sees people walking along the streets every day. The man also has a home in the country, but he prefers to spend his time in the city."*

We can represent the trained model with a table as Table 2.1 below.[5]

[5]For simplicity's sake, we will consider words in a case-insensitive manner. Only a few examples are included; the full table would be too large.

Table 2.1.: Selected statistics of words and contexts, for a model trained on the text above.



WORD (TARGET WORD)	CONTEXT (PRECEDING WORDS, IF ANY)	CONTEXT SIZE (NUMBER OF WORDS IN THE CONTEXT)	COUNT (HOW OFTEN DOES THE WORD APPEAR IN THE CORPUS, WHEN PRECEDED BY THE CONTEXT?)
"A"	∅	0	3
"MAN"	∅	0	2
"MAN"	"A"	1	1
"MAN"	"THE"	1	1
"CITY"	∅	0	2
"CITY"	"THE"	1	1
"CITY"	"IN THE"	2	1
"CITY"	"HOME IN THE"	3	1
⋮	⋮	⋮	⋮

For a statistical language model, *training* refers to pre-calculating these probabilities and then storing them somewhere (e.g. in memory or on disk) for quick retrieval at inference-time.

2.3.2.2. Inference-time

With the model trained on the corpus above, we can simply retrieve the values to perform inference. Let's see some examples:

Calculating the probability of the sentence "a man":

For a short sentence like *"a man"*, the calculations are simple enough; we just need to decompose the joint probability into conditionals using the chain rule and then use the counts as calculated in Table 2.1.

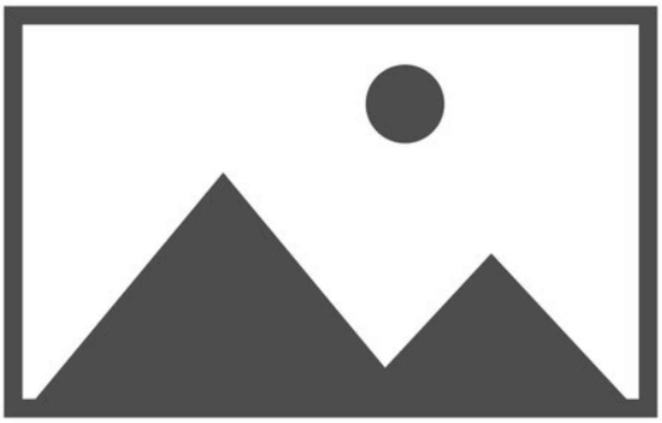


Figure 2.9.: Calculating the probability of the sentence "a man" with our sample statistical LM

Calculating the probability of the sentence: "a home in the city":

This example shows the first failure mode of statistical LMs: the probabilities may *collapse* to zero if the model is asked to output the probability of a sentence it hasn't seen in that exact order in the training set.

Calculating the probability of the sentence: "people running along the streets":

In this example, the calculation also fails, because a single word (*"running"*) is absent from the training set. A single missing word "poisoned" the product and cause it to collapse to zero.

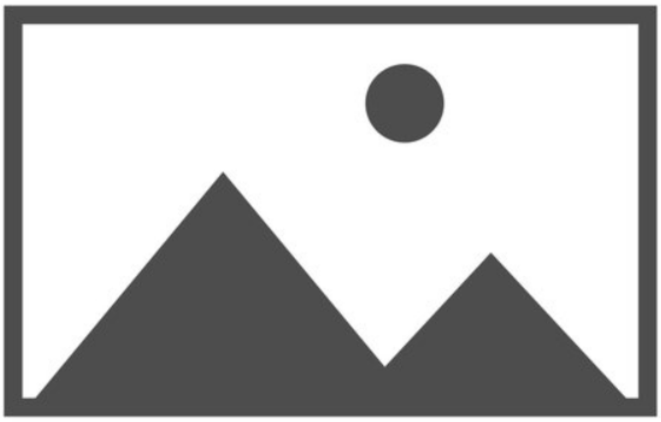


Figure 2.10.: Calculating the probability of the sentence “a home in the city” with our sample statistical LM

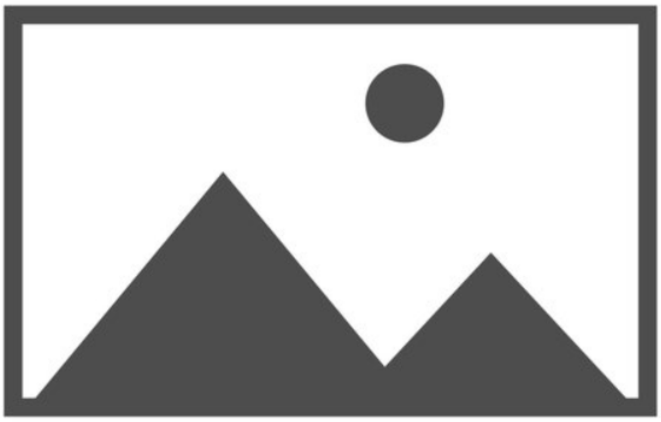


Figure 2.11.: Calculating the probability of the sentence “people running along the streets” with our sample statistical LM

Figure 2.9 above shows a basic case where a statistical language model can provide an adequate probability score to an (admittedly short) phrase. Figure 2.10 and Figure 2.11 above were meant to display two limitations of statistical LMs, namely: the *collapse* of probabilities to zero, due to words that don't appear in the training set in that order; the inability of these models to generalize to out-of-vocabulary words, even when they are semantically similar to other words found in the text.

What do these probabilities *mean* in a qualitative way?

When calculating the probability of a sequence, we arrive at figures such as 0.001, 0.3, 0.007, etc. We know that these numbers indicate how likely a piece of text is, as measured by a trained language model. But what does a value of 0.0002 *mean* as opposed to 0.00002 (which is ten times less likely)?
In practice, the nominal values don't matter much at all. They only matter in a *relative* way when, for instance, a system is trying to decide which of two sentences provides the best completion for some user-entered text. In this case, the only thing that matter is their *ordering*.
Also, real-world systems apply many transformations and optimizations to these probabilities and may not even calculate them directly.
For example: many systems calculate sums of log-probabilities instead of products of probabilities, to promote numerical stability; It's also common to normalize probabilities by the length of the sentence (otherwise shorter sentences are always going to be "more likely" than longer ones for simple numerical reasons). There are even mathematical shortcuts that allow one to calculate the ordering of values without needing to calculate the values themselves.

Let's now summarize the limitations of statistical language models.

2.3.3. Limitations

The examples we just saw provided some hints why a naïve statistical language model can hardly be used in practice. It's true that we used an artificially small dataset to train the model, but even if we train such a model on a very, *very* large corpus, it will never contain *every possible* sentence. And even such a corpus existed, there would be no hard-disk large enough to store all the necessary information.

- **Collapsed probabilities:** A clear limitation of statistical LMs is that inference is based on multiplications of conditional probabilities.

  When calculating probability for a sequence of words, if *any* of the conditional probabilities are 0 the whole product collapses to 0. This can be seen in Figure 2.10 and Figure 2.11. In Figure 2.10 all individual words were in the vocabulary, but not in that context, whereas in Figure 2.11 the product collapsed due to an out-of-vocabulary word (*"running"*) . Zero-valued probabilities are problematic because they prevent one from comparing the relative probabilities of two sequences.

- **Generalization:** In Figure 2.11 we saw that we got a collapsed probability for the sequence *"People running along the streets"*, which is explained by the fact that the word *"running"* is not in the training set. Any conditional probability term that includes this word will collapse to 0.

  A better model *should* be able to understand that the word *"running"* is similar to *"walking"*, which is a word that does appear in the training set. Being able to generalize—that is, transfer probability density—across similar words would make for a more useful model, as it would reduce the need for every possible sequence to be present in the training set.

- **Computational cost:** Training a statistical LM like the one we used in the worked example means going over the training corpus and recording how often each word appears, in what context.

  This may seem easy enough with a small train corpus (as used in the example) but what if we were to use a large corpus with a vocabulary containing 1 million[6] terms? If we had to store the conditional probabilities of 1 million terms appearing in multiple contexts, the disk space needed would be prohibitively expensive, precluding any practical use.

That's it for the basics of statistical language models. In the following section, we'll see what N-gram models are and how they help address some of the limitations seen above. Although N-gram models are an approximation to (or a subtype of) statistical LMs, they are different enough to merit their own section.

2.4. N-gram Language Models

In the previous section, we saw some of the limitations of statistical language models. These are mostly related to **(a)** the computational cost caused by the need to store a lot of information and **(b)** the inability of the model to generalize its knowledge to unseen word combinations, causing collapsing probabilities.

N-gram language models are more efficient statistical LMs and, at the same time, address some of their limitations. Let's see how, starting with an explanation of what N-grams are.

[6]1 million is the rough order of magnitude of the vocabulary size used in popular open-source NLP packages.

2.4.1. N-grams

N-grams are a generalization over words. They can be used to represent single words but also pairs, triplets, and larger sequences thereof.

The choice of N in an N-gram defines the length of the smallest unit we use. For example, a 1-gram (also called a unigram) is an N-gram with a single element, so it's just another name for a word. A 2-gram (also called a bigram) is simply a pair of words, whereas a 3-gram (also called a trigram) is an N-gram where $N = 3$, and it's simply a sequence of 3 words—a triplet.

The actual term "N-gram" has been around for at least some decades, as it's used in works such as those by Shannon (1948). In that specific work, however, *character* N-grams were used instead of *word* N-grams.

Let's see how the same sentence would be represented using N-grams, with different values for N. Figure 2.12 shows what the sentence *"Cats and dogs make unlikely but adorable friends"* looks like when split into uni-grams (N-gram with $N = 1$), bigrams (N-gram with $N = 2$), and trigrams (N-gram with $N = 3$).

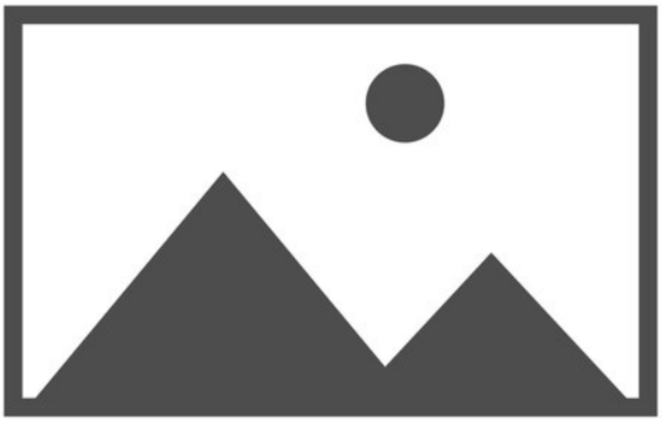


Figure 2.12.: Representing a sentence with N-grams: The sentence *"Cats and dogs make unlikely but adorable friends"* can be represented with N-grams in several ways: using $N = 1$ (unigrams), $N = 2$ (bigrams), and $N = 3$ (trigrams).

We can leverage N-grams to create approximations of fully-fledged statistical language models. Let's see how and why N-gram models tend to work better in practice.

2.4.2. N-gram models explained

N-gram models are an *approximation* of a full statistical language model. They operate under two assumptions:

1. We don't need to know *all* the previous words in the context—just the last $N-1$ words;
2. The closer a context word is to the target word, the more important it is.[7]

N-gram models work just like the models we saw in Section 2.3, with a subtle but key difference: The conditional probabilities obtained after decomposition are *pruned* after $N-1$ steps. Following assumptions **(1)** and **(2)**, we don't need to consider all the context words to arrive at adequate estimations of probabilities—just the last $N-1$, just like an N-gram of order N. This greatly simplifies the calculations and helps avoid some problems as we will see in the examples.

In Figure 2.13 we see an example of how to calculate the probability of a sentence using an N-gram model with $N = 2$ (right side) and compare it to the way it's done for a regular statistical LM (left):

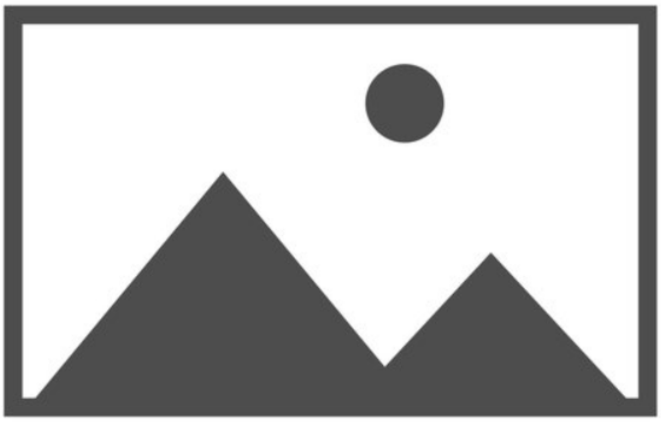


Figure 2.13.: Calculating the probability of sentence *"w1 w2 w3 w4"* using full conditional probabilities (left) and the analogous calculation with pruned conditional probabilities (right), as is the case with N-gram models. The last two terms are much simpler when calculated with N-gram models.

N-gram models can be seen as the result of applying an engineering mindset to purely statistical LMs, which are a theoretical framework that provides a solid foundation to study language from a mathematical standpoint, but not very practical for real-world use.

N-gram models are not only more efficient than statistical LM but also provide better results in the presence of incomplete data. Let's see some ways in which N-gram models are better suited to language modeling.

[7]The last word before the target word is more important than the second-last word, which in turn is more important than the third-last word, and so on.

2.4.3. Advantages over Statistical LMs

Up until now, we saw how N-gram models are an efficient approximation to fully statistical LMs. It turns out there's another benefit when we consider collapsed probabilities.

1. **Efficiency**: Statistical LMs and N-gram Models need to store conditional probability counts somewhere—either in memory or on disk—as they are trained. N-gram models require less space because the number of conditional probability combinations we need to keep track of is smaller (only contexts up to $N-1$ need to be kept). In addition to that, inference is also faster, as the number of multiplications is smaller.
2. **Reduced chance of collapsed probabilities**: Since a smaller number of context words are taken into account when calculating probabilities, there's a reduced chance of hitting upon a combination of words that were never seen in the training corpus. This means that there will be fewer zero terms in the conditional probability product—which in turn means that the result is less likely to collapse to zero.

2.4.4. Worked example

Similarly to what we did for statistical LMs in Section 2.3.2, let's now see how we go from a body of text (the training corpus) to a trained N-gram language model, and see what results we get when we perform inference with it.

Training-time

Let's use the same text again, to make comparisons easier:

> *"A man lives in a bustling city where he sees people walking along the streets every day. The man also has a home in the country, but he prefers to spend his time in the city."*

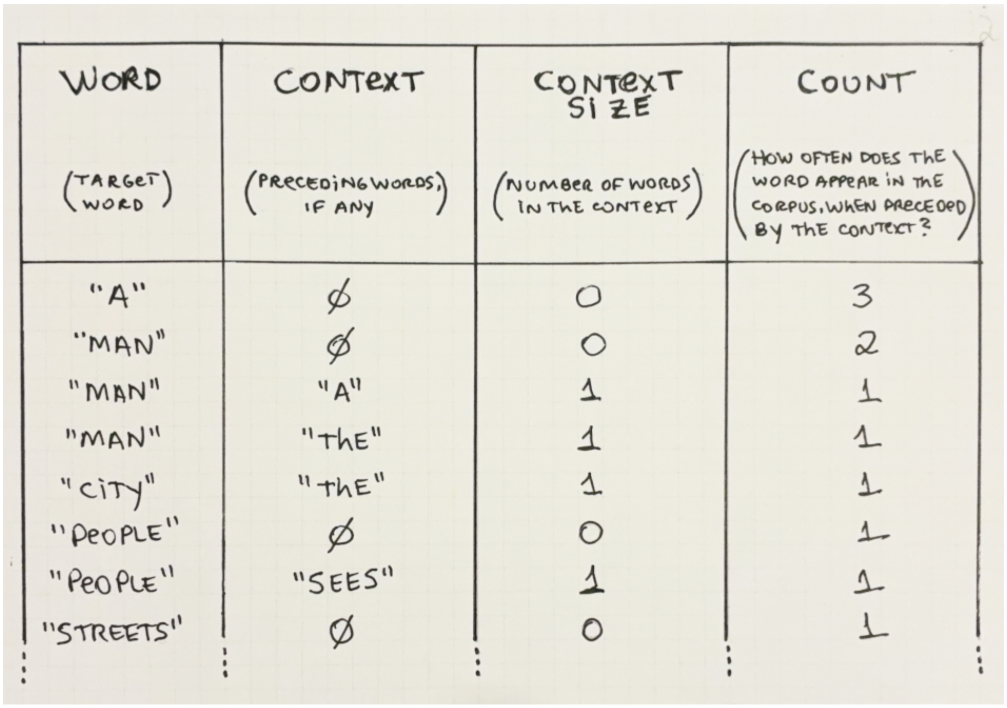
N-gram models only need to store the counts for a *limited* combination of contexts. For example, if we are using an N-gram model with $N = 2$ we only need to store the counts for words with up to 1 context word. This greatly reduces the amount of space needed.

Let's see the information we would need to store, in Table 2.2 below. Note that in this table there are no entries for combinations where the context size is larger than 1.

Inference-time

Let's see some examples of our N-gram model being used at inference-time:

Table 2.2.: Selected counts of words and contexts, for an N-gram model ($N = 2$) trained on the sample text.



WORD (TARGET WORD)	CONTEXT (PRECEDING WORDS, IF ANY)	CONTEXT SIZE (NUMBER OF WORDS IN THE CONTEXT)	COUNT (HOW OFTEN DOES THE WORD APPEAR IN THE CORPUS, WHEN PRECEDED BY THE CONTEXT?)
"A"	$\emptyset$	0	3
"MAN"	$\emptyset$	0	2
"MAN"	"A"	1	1
"MAN"	"THE"	1	1
"CITY"	"THE"	1	1
"PEOPLE"	$\emptyset$	0	1
"PEOPLE"	"SEES"	1	1
"STREETS"	$\emptyset$	0	1
⋮	⋮	⋮	⋮

Calculating the probability of the sentence "a man":

For *"a man"*, the result is the same as we got in Section 2.3.2 for the example with statistical LMs. This happens because the sentence is so short that it makes no difference if we use the full context or just an N-gram.

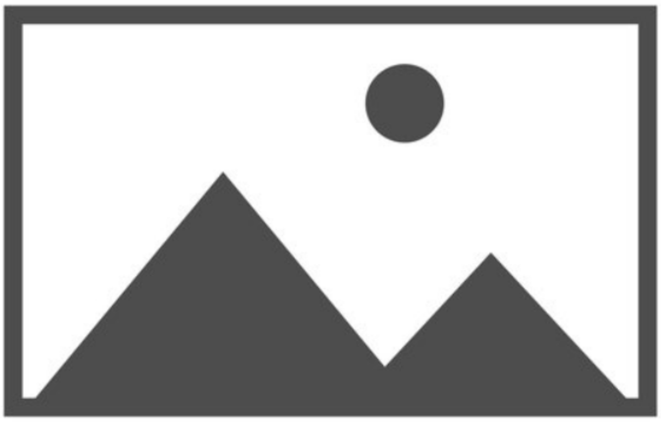


Figure 2.14.: Calculating the probability of the sentence *"a man"* with an N-gram model ($N = 2$).

Calculating the probability of the sentence: "a home in the city":

Differently from the example for statistical LMs (Figure 2.10), the probability calculated by an N-gram model is non-zero because none of the terms collapsed to zero. This is one example where using an N-gram model instead of a statistical LM avoided collapsing probabilities.

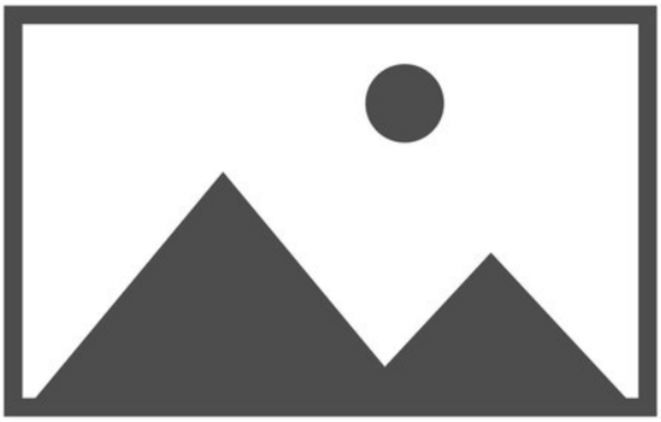


Figure 2.15.: Calculating the probability of the sentence *"a home in the city"* with an N-gram model ($N = 2$).

Calculating the probability of the sentence: "people running along the streets":

Several terms are simpler than the equivalent example calculated by the statistical LM in Section 2.3.2. However, the probability of the whole sentence still collapsed to zero because *"running"* is an out-of-vocabulary word.

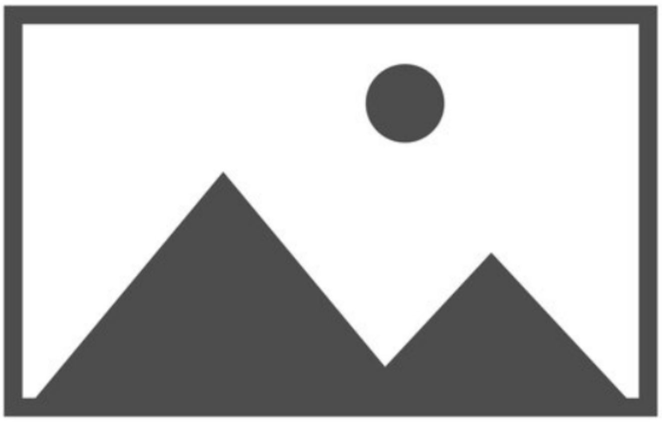


Figure 2.16.: Calculating the probability of the sentence *"people running along the streets"* with an N-gram model ($N = 2$).

Even though N-gram models have clear benefits over purely statistical LMs, they also have their own limitations, as we'll see next.

2.4.5. Limitations

In the previous section, we saw that N-gram models fail to provide correct scores for relatively simple input sequences. Let's analyze some of the main shortcomings of these models.

- **Limited context:** Using a reduced version of the context increases efficiency and reduces the risk of collapsed probabilities. But in some cases—especially if one chooses a small value for N—there will be some loss in performance because relevant words will be left out of the context.

  Let's see an example with the following sentence: *"People in France speak __________"*.

  As explained in Section 2.1, we can predict the most likely next word by generating all variations, calculating the probability score for them, and then picking the highest-scoring variant. Let's analyze what these would look like when calculated using $N = 2$ and using $N = 3$.

  Since we want to keep the example simple, we will only consider the scores for two variants: *"People in France speak Spanish"* and *"People in France speak French"*. A human would naturally know that the latter is the correct answer, but what would our model say?

We can see the difference in Figure 2.17 below. When we use $N = 2$, the model would assign almost the same score to both variants, so it will not be able to realize that "French" is a better candidate than "Spanish" for the missing word. Adding a single word to the context (increasing N to 3) allows the model to include a new word and it makes all the difference. It's now able to use conditional probabilities that include the word "France", which naturally occurs much more often together with "French" than it does with "Spanish"!

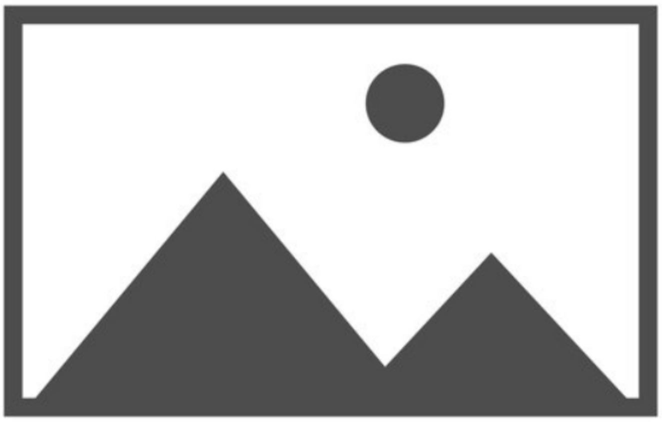


Figure 2.17.: A clear case where using an N-gram model with too small a context hinders performance. When we use $N = 3$, the model can pick up a context that includes the word "France". Since the word "France" is used much more often with "French" than it is with "Spanish", the model is more likely to infer that the missing word is "French". EOS is a marker for the end of the sequence.

The choice of N introduces a tradeoff between efficiency and expressivity; smaller values of N make the model faster and require less storage space but larger values of N allow the model to consider more data to calculate probabilities.

- **Collapsed probabilities:** As we saw in Figure 2.16, N-gram models are still vulnerable to collapsed probabilities due to out-of-vocabulary words. It's not as vulnerable as pure statistical LMs, but the issue persists.
- **Generalization:** With respect to generalizing to semantically similar words as those seen at training-time, N-gram models don't help us much in comparison with regular statistical models.

We saw an example in Figure 2.16. Even though the calculations were simpler and faster, we again hit upon a collapsed probability—and the model wasn't able to transfer probability from the word "walking" to the word "running". The generalization problem will be solved with Neural LMs and Embedding Layers, which will be discussed in **?@sec-ch-neural-language-models-and-self-supervision**.

Some of these limitations were worked around—or at least mitigated—with a little engineering ingenuity:

2.4.6. Later optimizations

N-gram models are an optimization on top of regular statistical LMs, but they fall short of a perfect model. Additional enhancements were proposed to make N-gram models more capable and address some of their limitations. These include *backoff*, *smoothing*, and *interpolation*. Table 2.3 shows the main objectives of each strategy:

Table 2.3.: N-gram model enhancements and their objectives

Strategy	Objective
Backoff	Reduce the chance of collapsed conditional probability products
Smoothing	Reduce the chance of collapsed conditional probability products
Interpolation	Increase the quality of model output

Let's explain each in more detail.

- **Backoff:** Backoff is a strategy whereby one replaces a *zero*-term with a lower-order N-gram term in the conditional probability decomposition—to prevent the whole product from collapsing to zero.

  As an example, let's calculate the probability score for the string *"w1 w2 w3 w4"* using backoff and $N = 3$. If the word *"w4"* is *never* preceded by *"w2 w3"* in the training set, the conditional probability $P(''w4'' \mid ''w2\ w3'')$ will collapse to zero, so we *back-off* to $P(''w4'' \mid ''w3'')$ instead. See Figure 2.18 for a visual representation.

- **Smoothing:** Smoothing refers to modifying zero terms in the conditional probability decomposition, to avoid collapsed probabilities.

  The simplest way to do this is called Laplace Smoothing (aka *Add-one* Smoothing). It works by adding 1 to every N-gram count before performing the calculations. This will get rid of zeros and prevent probability collapse. See an example in Figure 2.19 below:

- **Interpolation:** Unlike *backoff* and *smoothing*, interpolation is used to enhance the quality of scores output by an N-gram model—not to help with collapsed probabilities.

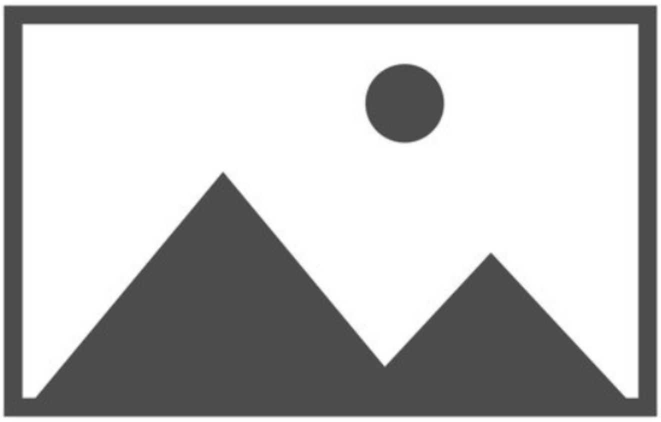


Figure 2.18.: If the term $P(''w4'' \mid ''w2\ w3'')$ has zero probability in the train set, a simple trigram model would cause the score for *“w1 w2 w3 w4”* to collapse to zero. Using a backoff model allows us to replace the zero-term with a lower-order *N*-gram instead, which can prevent the collapse at the cost of slightly lower accuracy.

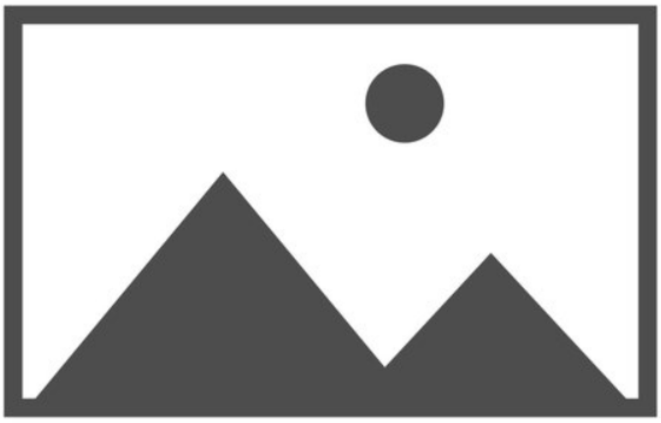


Figure 2.19.: By adding 1 to every count before multiplying the terms, we get rid of zeros and we have a valid—non-collapsed—score.

It works by using all available N-grams up to the order we are considering. For example, if we apply interpolation to a trigram model, bigrams and unigrams will also be used—in addition to trigrams. Figure 2.20 shows how the calculations change when we introduce interpolation:

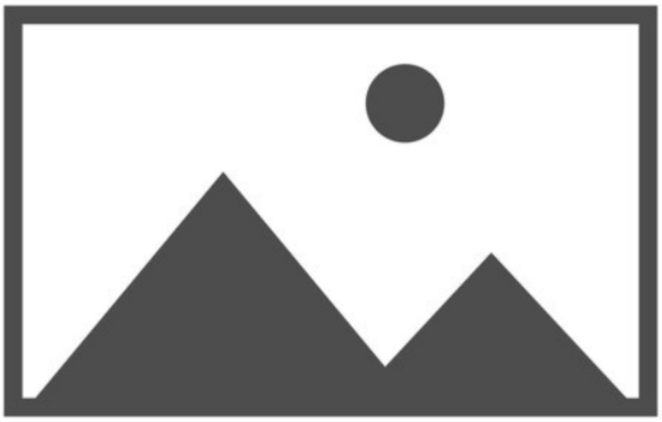


Figure 2.20.: A simplified interpolation strategy for a trigram model. Added terms include values for lower-order N-grams ($N = 2$ and $N = 1$) to the original terms. Some details were left out to aid in understanding, without loss of generality.

2.5. Measuring the performance of Language Models

Measuring the quality of outputs from a language model is crucial to understanding which techniques enhance the model capabilities—and whether the added complexity is worth the gain in performance.

There are two different strategies to evaluate the performance of an LM: *Intrinsic* evaluation refers to measuring how well an LM performs on its own, i.e. in the language modeling task itself. *Extrinsic* evaluation, on the other hand, measures an LM's performance based on how well that LM can be used to help other NLP tasks.

These two types of evaluation are not mutually exclusive—we may need to use both, depending on the task at hand.

A visual summary of the ways to evaluate an LM's performance can be seen in Figure 2.21 below.

In this Section, we will only cover intrinsic evaluation. Extrinsic evaluation and *Evals* will be discussed in Part III and IV, where we will see how pre-trained LMs can be used to leverage other—so-called *downstream*—NLP tasks.

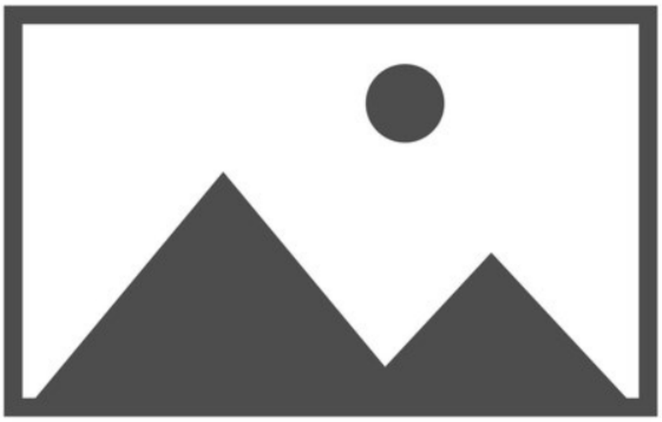


Figure 2.21.: Different strategies can be used to evaluate the performance of language models.

2.5.1. Objective vs Subjective evaluation

In intrinsic evaluation, we have at least two ways to evaluate an LM: using *objective* metrics calculated on a test set (as any other ML model) and asking humans to *subjectively* rate the output of an LM and assess its quality based on their own opinions.

Both objective and subjective evaluations are useful depending on what we want to measure.

When comparing two different models you probably want an objective metric, as it will better show small differences in performance.

On the other hand, suppose you want to compare two models to decide which generates text that's closer to Shakespeare's style—this is not so easily captured in an objective metric so it may be better to use subjective evaluation.

We will see one example of each of these two strategies. We'll start with objective evaluation, by looking at perplexity—the most commonly used metric for intrinsic LM evaluation. Then we'll go over a subjective evaluation method used by OpenAI to measure the quality of the output generated by GPT-3 (Brown et al. (2020)).

2.5.2. Objective evaluation

Evaluating a model objectively means using precisely defined metrics. This is in contrast to subjective evaluation, which usually involves humans and their subjective opinions.

Language models are a subtype of machine learning (ML) models. And there is a very simple yet effective way to evaluate ML models to understand how well (if at all) they are able to learn from data.

We call it the *train-test-split*: We randomly split our data into two groups (train and test set) and make sure you use the train set for training and the test set for validation, *without mixing both*. It would be trivially easy for any model to have perfect performance on the same set it was trained on—it could simply memorize all of the data. A random split also guarantees that both sets will have the same distribution. A visual explanation can be seen in Figure 2.22 below:

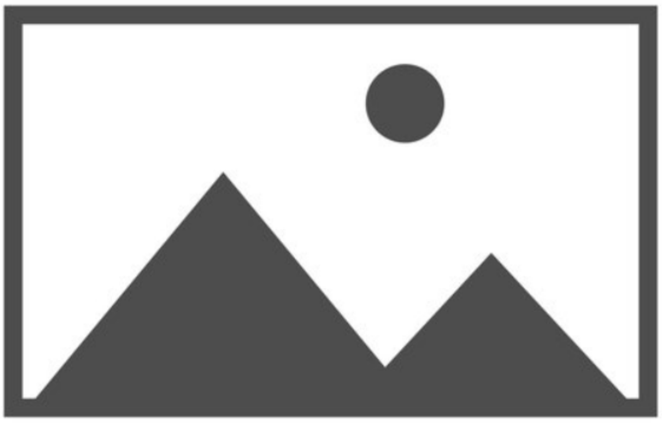


Figure 2.22.: Machine learning models must be evaluated on a different dataset from the one they've been trained on.

The train-test-split strategy is not tied to any one specific modeling strategy; It can be used for any ML model and, consequently, by any LM we mention in this book—including neural LMs, which we'll cover in the next Chapter.

ℹ BLEU, ROUGE and similar metrics

You may have heard of metrics such as BLUE and ROUGE being used to evaluate language models. These are *extrinsic* metrics—they are used in downstream NLP tasks such as machine translation and summarization.

The most widely used metric to evaluate language models intrinsically, in an objective manner, is *perplexity*. Let's see how.

Perplexity

Perplexity (commonly abbreviated as PPL) is a measure of how *surprised* (or *perplexed*) a language model is when it looks at some new text. The *higher* a model's perplexity, the *worse* its performance.[8]

[8]One way to remember this relationship is to consider that a well-trained LM should *never* be perplexed when it encounters some new text! If it was able to learn what its target language looks like—any new text should be *unsurprising*, statistically speaking.

Perplexity can only be calculated in relation to some data. This is why it doesn't make sense to say *"I trained a language model and its perplexity is 100"*. Perplexity is always calculated based on some set—the test set. We must always say on which test set we calculated the perplexity on!

Mathematically, the perplexity of a language model θ with respect to a corpus C is the N-th root of the inverse of the probability score of C, as we can see in Equation 2.5 below. N is the number of words in the corpus; it's used to normalize the score.

$$Perplexity(\theta)_C = \sqrt[n]{\frac{1}{P_\theta(C)}} \tag{2.5}$$

If our LM gives a high probability score to the text in the test corpus it means that it was able to correctly understand that it is valid text. The denominator in Equation 2.5 will be large, so the perplexity will be small.

Conversely, if an LM assigns a *low* probability score to the text in the test corpus, it should be penalized. This intuition matches the formula in Equation 2.5: if the probability score is low, the denominator will be small and the value of the fraction will be larger.

What about N? Why do we need to take the N-th root?

N is the number of words in the test corpus C. We take the root of the inverse probability because a corpus with fewer words would always have lower perplexity for a given model θ. Taking the root eliminates this bias and allows us to compare models across multiple corpora.

2.5.3. Subjective evaluation

One way to measure the quality of an LLM's output is by measuring its perplexity on a test set, as we showed in the previous section.

But we can also delegate this task to humans—ideally a large, diverse group thereof—and ask *them* to evaluate a language model according to what *they* believe is good performance.

One *could* generate multiple pieces of text using some model and ask human evaluators to quantify how good they think the generated text is. However, each person has a different opinion on what *good* text is, and it would be hard to have consistent results.

A more principled subjective evaluation strategy is used in the GPT-3 article (Brown et al. (2020)); Humans are asked to try to differentiate automatically generated texts from those written by humans. Let's see the specifics:

maybe add something about LLM-as-a-judge (using an LM itself to evaluate some text). Was used in DPO and in LLAMA-2. it's hard to prevent data leakage because models often share the same training data (large corpora, etc)

Detecting automatically generated text

Testing whether humans can be *fooled* by a language model is akin to a Turing test, where a system must trick a human interviewer into thinking they are talking to a real person.

Such a method was used in the GPT-3 article (Brown et al. (2020)) and it proceeds by showing human annotators text samples that are either **(1)** extracted from a real news website, or **(2)** generated on the spot with a language model.

Participants are asked to rate how likely they thought the text was automatically generated by assigning it Likert-type classifications ranging from *"very likely written by a human"* to *"I don't know"*, and to *"very likely written by a machine"*.

At the end of the experiment, the authors concluded that the results conform to the expectations and they highlighted two key results:

1. The longer the text, the easier it is for humans to tell apart synthetic from human-generated text;
2. For the most powerful GPT-3 variant, human annotators were only 52% likely to tell apart synthetic from natural texts. This is only slightly better than flipping a coin—text generated by GPT-3 is nearly indistinguishable from human-written text!

Note that this is still an intrinsic evaluation method—since it doesn't evaluate the LM in how it enables other NLP tasks, but in how it performs in a core language modeling task: autoregressive text generation.

2.6. Summary

- Language models are machine learning models and, as such, have separate training and inference stages and should be validated on out-of-sample data.
- Language models were first created to aid in problems such as speech recognition, text correction and text recognition in images.
- Statistical language models view language as a statistical distribution over words. They represent a sentence as a joint probability of words and use the chain rule of probability to decompose it into conditional probabilities that can be directly calculated from word counts.

- Simple statistical LMs suffer from problems such as the sparsity of long sequences of words and heavy computational costs. This makes them hard to use in real-life scenarios.
- N-gram models are an approximation to statistical LMs and they solve some of the problems that afflict those. N-gram models themselves have some limitations and some recent advancements to address those are Backoff, Smoothing, and Interpolation.
- Strategies to evaluate the performance of LMs can be split into intrinsic and extrinsic, depending on whether the LM is being measured on the language modeling task or on downstream NLP tasks.

2.7. References

Brown, T., B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, et al. 2020. "Language Models Are Few-Shot Learners." In *Advances in Neural Information Processing Systems*, edited by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, 33:1877–1901. Curran Associates, Inc. https://bit.ly/brown-2020-gpt3.

Jurafsky, D., and J. H. Martin. 2025. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, with Language Models*. 3rd ed. https://web.stanford.edu/~jurafsky/slp3/.

Shannon, C. E. 1948. "A Mathematical Theory of Communication." *The Bell System Technical Journal* 27: 379–423. Accessed April 22, 2003. http://plan9.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf.