CHAPTER

# 10 Large Language Models

*"How much do we know at any time? Much more, or so I believe, than we know we know."*

Agatha Christie, *The Moving Finger*

Fluent speakers of a language bring an enormous amount of knowledge to bear during comprehension and production. This knowledge is embodied in many forms, perhaps most obviously in the vocabulary, the rich representations we have of words and their meanings and usage. This makes the vocabulary a useful lens to explore the acquisition of knowledge from text, by both people and machines.

Estimates of the size of adult vocabularies vary widely both within and across languages. For example, estimates of the vocabulary size of young adult speakers of American English range from 30,000 to 100,000 depending on the resources used to make the estimate and the definition of what it means to know a word. What is agreed upon is that the vast majority of words that mature speakers use in their day-to-day interactions are acquired early in life through spoken interactions with caregivers and peers, usually well before the start of formal schooling. This active vocabulary (usually on the order of 2000 words for young speakers) is extremely limited compared to the size of the adult vocabulary, and is quite stable, with very few additional words learned via casual conversation beyond this early stage. Obviously, this leaves a very large number of words to be acquired by other means.

A simple consequence of these facts is that children have to learn about 7 to 10 words a day, *every single day*, to arrive at observed vocabulary levels by the time they are 20 years of age. And indeed empirical estimates of vocabulary growth in late elementary through high school are consistent with this rate. How do children achieve this rate of vocabulary growth? The bulk of this knowledge acquisition seems to happen as a by-product of reading, as part of the rich processing and reasoning that we perform when we read. Research into the average amount of time children spend reading, and the lexical diversity of the texts they read, indicate that it is possible to achieve the desired rate. But the mechanism behind this rate of learning must be remarkable indeed, since at some points during learning the rate of vocabulary growth exceeds the rate at which new words are appearing to the learner!

Such facts have motivated the *distributional hypothesis* of Chapter 6, which suggests that aspects of meaning can be learned solely from the texts we encounter over our lives, based on the complex association of words with the words they co-occur with (and with the words that those words occur with). The distributional hypothesis suggests both that we can acquire remarkable amounts of knowledge from text, and that this knowledge can be brought to bear long after its initial acquisition. Of course, grounding from real-world interaction or other modalities can help build even more powerful models, but even text alone is remarkably useful.

**pretraining**

In this chapter we formalize this idea of **pretraining**—learning knowledge about language and the world from vast amounts of text—and call the resulting pretrained language models **large language models**. Large language models exhibit remark-

able performance on all sorts of natural language tasks because of the knowledge they learn in pretraining, and they will play a role throughout the rest of this book. They have been especially transformative for tasks where we need to produce text, like summarization, machine translation, question answering, or chatbots.

We'll start by seeing how to apply the **transformer** of Chapter 9 to language modeling, in a setting often called causal or autoregressive language models, in which we iteratively predict words left-to-right from earlier words. We'll first introduce training, seeing how language models are self-trained by iteratively being taught to guess the next word in the text from the prior words.

We'll then talk about the process of text generation. The application of LLMs to generate text has vastly broadened the scope of NLP,. Text generation, code-generation, and image-generation together constitute the important new area of **generative AI**. We'll introduce specific algorithms for generating text from a language model, like **greedy decoding** and **sampling**. And we'll see that almost any NLP task can be modeled as word prediction in a large language model, if we think about it in the right way. We'll work through an example of using large language models to solve one classic NLP task of **summarization** (generating a short text that summarizes some larger document).

generative AI

## 10.1 Large Language Models with Transformers

The prior chapter introduced most of the components of a transformer in the domain of language modeling: the **transformer block** including **multi-head attention**, the **language modeling head**, and the positional encoding of the input. In the following sections we'll introduce the remaining aspects of the transformer LLM: **sampling** and **training**. Before we do that, we use this section to talk about why and how we apply transformer-based large language models to NLP tasks.

conditional generation

The tasks we will describe are all cases of **conditional generation**. Conditional generation is the task of generating text conditioned on an input piece of text. That is, we give the LLM an input piece of text, generally called a **prompt**, and then have the LLM continue generating text token by token, conditioned on the prompt and the previously generated tokens. The fact that transformers have such long contexts (many thousands of tokens) makes them very powerful for conditional generation, because they can look back so far into the prompting text.

Consider the simple task of text completion, illustrated in Fig. 10.1. Here a language model is given a text prefix and is asked to generate a possible completion. Note that as the generation process proceeds, the model has direct access to the priming context as well as to all of its own subsequently generated outputs (at least as much as fits in the large context window). This ability to incorporate the entirety of the earlier context and generated outputs at each time step is the key to the power of large language models built from transformers.

So why should we care about predicting upcoming words or tokens? The insight of large language modeling is that **many practical NLP tasks can be cast as word prediction**, and that a powerful-enough language model can solve them with a high degree of accuracy. For example, we can cast sentiment analysis as language modeling by giving a language model a context like:

```
The sentiment of the sentence ‘‘I like Jackie Chan" is:
```

and comparing the following conditional probability of the words "positive" and the

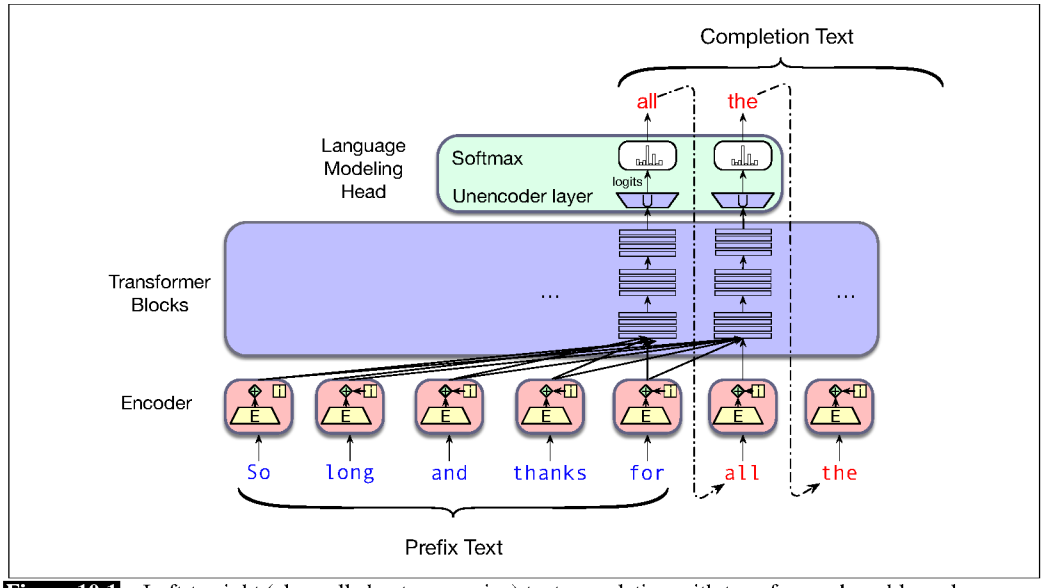


**Figure 10.1** Left-to-right (also called autoregressive) text completion with transformer-based large language models. As each token is generated, it gets added onto the context as a prefix for generating the next token.

word "negative" to see which is higher:

$P(\text{positive}|\texttt{The sentiment of the sentence ``I like Jackie Chan" is:})$
$P(\text{negative}|\texttt{The sentiment of the sentence ``I like Jackie Chan" is:})$

If the word "positive" is more probable, we say the sentiment of the sentence is positive, otherwise we say the sentiment is negative.

We can also cast more complex tasks as word prediction. Consider question answering, in which the system is given a question (for example a question with a simple factual answer) and must give a textual answer; we introduce this task in detail in Chapter 14. We can cast the task of question answering as word prediction by giving a language model a question and a token like `A:` suggesting that an answer should come next:

`Q: Who wrote the book ``The Origin of Species"? A:`

If we ask a language model to compute the probability distribution over possible next words given this prefix:

$P(w|\texttt{Q: Who wrote the book ``The Origin of Species"? A:})$

and look at which words $w$ have high probabilities, we might expect to see that *Charles* is very likely, and then if we choose *Charles* and continue and ask

$P(w|\texttt{Q: Who wrote the book ``The Origin of Species"? A: Charles})$

we might now see that *Darwin* is the most probable token, and select it.

**text summarization**

Conditional generation can even be used to accomplish tasks that must generate longer responses. Consider the task of **text summarization**, which is to take a long text, such as a full-length article, and produce an effective shorter summary of it. We can cast summarization as language modeling by giving a large language model a text, and follow the text by a token like `tl;dr`; this token is short for something like

'too long; didn't read' and in recent years people often use this token, especially in informal work emails, when they are going to give a short summary. Since this token is sufficiently frequent in language model training data, language models have seen many texts in which the token occurs before a summary, and hence will interpret the token as instructions to generate a summary. We can then do conditional generation: give the language model this prefix, and then have it generate the following words, one by one, and take the entire response as a summary. Fig. 10.2 shows an example of a text and a human-produced summary from a widely-used summarization corpus consisting of CNN and Daily Mail news articles.

**Original Article**

The only thing crazier than a guy in snowbound Massachusetts boxing up the powdery white stuff and offering it for sale online? People are actually buying it. For $89, self-styled entrepreneur Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says.

But not if you live in New England or surrounding states. "We will not ship snow to any states in the northeast!" says Waring's website, ShipSnowYo.com. "We're in the business of expunging snow!"

His website and social media accounts claim to have filled more than 133 orders for snow – more than 30 on Tuesday alone, his busiest day yet. With more than 45 total inches, Boston has set a record this winter for the snowiest month in its history. Most residents see the huge piles of snow choking their yards and sidewalks as a nuisance, but Waring saw an opportunity.

According to Boston.com, it all started a few weeks ago, when Waring and his wife were shoveling deep snow from their yard in Manchester-by-the-Sea, a coastal suburb north of Boston. He joked about shipping the stuff to friends and family in warmer states, and an idea was born. [...]

**Summary**

Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says. But not if you live in New England or surrounding states.

**Figure 10.2** Excerpt from a sample article and its summary from the CNN/Daily Mail summarization corpus (Hermann et al., 2015b), (Nallapati et al., 2016).

If we take this full article and append the token `tl;dr`, we can use this as the context to prime the generation process to produce a summary as illustrated in Fig. 10.3. Again, what makes transformers able to succeed at this task (as compared, say, to the primitive n-gram language model) is that attention can incorporate information from the large context window, giving the model access to the original article as well as to the newly generated text throughout the process.

**greedy decoding**

Which words do we generate at each step? One simple way to generate words is to always generate the most likely word given the context. Generating the most likely word given the context is called **greedy decoding**. A greedy algorithm is one that make a choice that is locally optimal, whether or not it will turn out to have been the best choice with hindsight. Thus in greedy decoding, at each time step in generation, the output $y_t$ is chosen by computing the probability for each possible output (every word in the vocabulary) and then choosing the highest probability word (the argmax):

$$\hat{w}_t = \text{argmax}_{w \in V} P(w|\mathbf{w}_{<t}) \tag{10.1}$$

In practice, however, we don't use greedy decoding with large language models. A major problem with greedy decoding is that because the words it chooses are (by definition) extremely predictable, the resulting text is generic and often quite repetitive. Indeed, greedy decoding is so predictable that it is deterministic; if the context

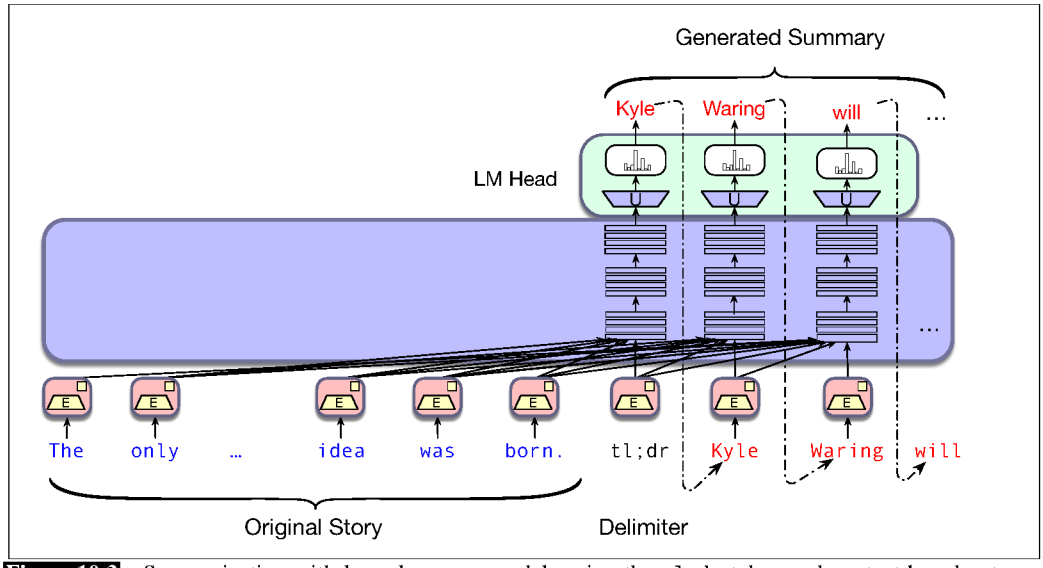


**Figure 10.3** Summarization with large language models using the `tl;dr` token and context-based autoregressive generation.

is identical, and the probabilistic model is the same, greedy decoding will always result in generating exactly the same string. We'll see in Chapter 13 that an extension to greedy decoding called **beam search** works well in tasks like machine translation, which are very constrained in that we are always generating a text in one language conditioned on a very specific text in another language. In most other tasks, however, people prefer text which has been generated by more sophisticated methods, called **sampling methods**, that introduce a bit more diversity into the generations. We'll see how to do that in the next few sections.

## 10.2 Sampling for LLM Generation

The core of the generation process for large language models is the task of choosing the single word to generate next based on the context and based on the probabilities that the model assigns to possible words. This task of choosing a word to generate based on the model's probabilities is called **decoding**. Decoding from a language model in a left-to-right manner (or right-to-left for languages like Arabic in which we read from right to left), and thus repeatedly choosing the next word conditioned on our previous choices is called **autoregressive generation** or **causal LM generation**.[1] (As we'll see, alternatives like the masked language models of Chapter 11 are non-causal because they can predict words based on both past and future words).

decoding

autoregressive generation

The most common method for decoding in large language models is **sampling**. Recall from Chapter 3 that **sampling** from a model's distribution over words means to choose random words according to their probability assigned by the model. That is, we iteratively choose a word to generate according to its probability in context

sampling

[1] Technically an **autoregressive** model predicts a value at time $t$ based on a linear function of the values at times $t-1$, $t-2$, and so on. Although language models are not linear (since they have many layers of non-linearities), we loosely refer to this generation technique as autoregressive since the word generated at each time step is conditioned on the word selected by the network from the previous step.

as defined by the model. Thus we are more likely to generate words that the model thinks have a high probability in the context and less likely to generate words that the model thinks have a low probability.

We saw back in Chapter 3 on page 43 how to generate text from a unigram language model , by repeatedly randomly sampling words according to their probability until we either reach a pre-determined length or select the end-of-sentence token. To generate text from a trained transformer language model we'll just generalize this model a bit: at each step we'll sample words according to their probability *conditioned on our previous choices*, and we'll use a transformer language model as the probability model that tells us this probability.

We can formalize this algorithm for generating a sequence of words $W = w_1, w_2, \ldots, w_N$ until we hit the end-of-sequence token, using $x \sim p(x)$ to mean 'choose $x$ by sampling from the distribution $p(x)$:

$\mathrm{i} \leftarrow 1$
$w_i \sim \mathrm{p}(w)$
**while** $w_i$ != EOS
  $\mathrm{i} \leftarrow \mathrm{i} + 1$
  $w_i \sim \mathrm{p}(w_i \mid w_{<i})$

**random sampling**

The algorithm above is called **random sampling**, and it turns out random sampling doesn't work well enough. The problem is that even though random sampling is mostly going to generate sensible, high-probable words, there are many odd, low-probability words in the tail of the distribution, and even though each one is low-probability, if you add up all the rare words, they constitute a large enough portion of the distribution that they get chosen often enough to result in generating weird sentences. For this reason, instead of random sampling, we usually use sampling methods that avoid generating the very unlikely words.

The sampling methods we introduce below each have parameters that enable trading off two important factors in generation: **quality** and **diversity**. Methods that emphasize the most probable words tend to produce generations that are rated by people as more accurate, more coherent, and more factual, but also more boring and more repetitive. Methods that give a bit more weight to the middle-probability words tend to be more creative and more diverse, but less factual and more likely to be incoherent or otherwise low-quality.

### 10.2.1 Top-*k* sampling

**top-k sampling**

**Top-k sampling** is a simple generalization of greedy decoding. Instead of choosing the single most probable word to generate, we first truncate the distribution to the top $k$ most likely words, renormalize to produce a legitimate probability distribution, and then randomly sample from within these $k$ words according to their renormalized probabilities. More formally:

1. Choose in advance a number of words $k$
2. For each word in the vocabulary $V$, use the language model to compute the likelihood of this word given the context $p(w_t|\mathbf{w}_{<t})$
3. Sort the words by their likelihood, and throw away any word that is not one of the top $k$ most probable words.
4. Renormalize the scores of the $k$ words to be a legitimate probability distribution.

5. Randomly sample a word from within these remaining $k$ most-probable words according to its probability.

When $k = 1$, top-$k$ sampling is identical to greedy decoding. Setting $k$ to a larger number than 1 leads us to sometimes select a word which is not necessarily the most probable, but is still probable enough, and whose choice results in generating more diverse but still high-enough-quality text.

### 10.2.2 Nucleus or top-$p$ sampling

One problem with top-$k$ sampling is that $k$ is fixed, but the shape of the probability distribution over words differs in different contexts. If we set $k = 10$, sometimes the top 10 words will be very likely and include most of the probability mass, but other times the probability distribution will be flatter and the top 10 words will only include a small part of the probability mass.

top-p sampling

An alternative, called **top-p sampling** or **nucleus sampling** (Holtzman et al., 2020), is to keep not the top $k$ words, but the top $p$ percent of the probability mass. The goal is the same; to truncate the distribution to remove the very unlikely words. But by measuring probability rather than the number of words, the hope is that the measure will be more robust in very different contexts, dynamically increasing and decreasing the pool of word candidates.

Given a distribution $P(w_t|\mathbf{w}_{<t})$, we sort the distribution from most probable, and then the top-$p$ vocabulary $V^{(p)}$ is the smallest set of words such that

$$\sum_{w \in V^{(p)}} P(w|\mathbf{w}_{<t}) \geq p. \tag{10.2}$$

### 10.2.3 Temperature sampling

temperature sampling

In **temperature sampling**, we don't truncate the distribution, but instead reshape it. The intuition for temperature sampling comes from thermodynamics, where a system at a high temperature is very flexible and can explore many possible states, while a system at a lower temperature is likely to explore a subset of lower energy (better) states. In low-temperature sampling, we smoothly increase the probability of the most probable words and decrease the probability of the rare words.

We implement this intuition by simply dividing the logit by a temperature parameter $\tau$ before we normalize it by passing it through the softmax. In low-temperature sampling, $\tau \in (0, 1]$. Thus instead of computing the probability distribution over the vocabulary directly from the logit as in the following (repeated from Eq. 9.46):

$$\mathbf{y} = \text{softmax}(u) \tag{10.3}$$

we instead first divide the logits by $\tau$, computing the probability vector $y$ as

$$\mathbf{y} = \text{softmax}(u/\tau) \tag{10.4}$$

Why does this work? When $\tau$ is close to 1 the distribution doesn't change much. But the lower $\tau$ is, the larger the scores being passed to the softmax (dividing by a smaller fraction $\tau \leq 1$ results in making each score larger). Recall that one of the useful properties of a softmax is that it tends to push high values toward 1 and low values toward 0. Thus when larger numbers are passed to a softmax the result is a distribution with increased probabilities of the most high-probability words and decreased probabilities of the low probability words, making the distribution more greedy. As $\tau$ approaches 0 the probability of the most likely word approaches 1.

Note, by the way, that there can be other situations where we may want to do something quite different and flatten the word probability distribution instead of making it greedy. Temperature sampling can help with this situation too, in this case **high-temperature** sampling, in which case we use $\tau > 1$.

## 10.3 Pretraining Large Language Models

How do we teach a transformer to be a language model? What is the algorithm and what data do we train on?

### 10.3.1 Self-supervised training algorithm

self-supervision

To train a transformer as a language model, we use the same **self-supervision** (or **self-training**) algorithm we saw in Section 8.2.2: we take a corpus of text as training material and at each time step $t$ ask the model to predict the next word. We call such a model self-supervised because we don't have to add any special gold labels to the data; the natural sequence of words is its own supervision! We simply train the model to minimize the error in predicting the true next word in the training sequence, using cross-entropy as the loss function.

Recall that the cross-entropy loss measures the difference between a predicted probability distribution and the correct distribution.

$$L_{CE} = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w] \tag{10.5}$$

In the case of language modeling, the correct distribution $\mathbf{y}_t$ comes from knowing the next word. This is represented as a one-hot vector corresponding to the vocabulary where the entry for the actual next word is 1, and all the other entries are 0. Thus, the cross-entropy loss for language modeling is determined by the probability the model assigns to the correct next word (all other words get multiplied by zero). So at time $t$ the CE loss in Eq. 10.5 can be simplified as the negative log probability the model assigns to the next word in the training sequence.

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}] \tag{10.6}$$

Thus at each word position $t$ of the input, the model takes as input the correct sequence of tokens $w_{1:t}$, and uses them to compute a probability distribution over possible next words so as to compute the model's loss for the next token $w_{t+1}$. Then we move to the next word, we ignore what the model predicted for the next word and instead use the correct sequence of tokens $w_{1:t+1}$ to estimate the probability of token $w_{t+2}$. This idea that we always give the model the correct history sequence to predict the next word (rather than feeding the model its best case from the previous time step) is called **teacher forcing**.

teacher forcing

Fig. 10.4 illustrates the general training approach. At each step, given all the preceding words, the final transformer layer produces an output distribution over the entire vocabulary. During training, the probability assigned to the correct word is used to calculate the cross-entropy loss for each item in the sequence. The loss for a training sequence is the average cross-entropy loss over the entire sequence. The weights in the network are adjusted to minimize the average CE loss over the training sequence via gradient descent.

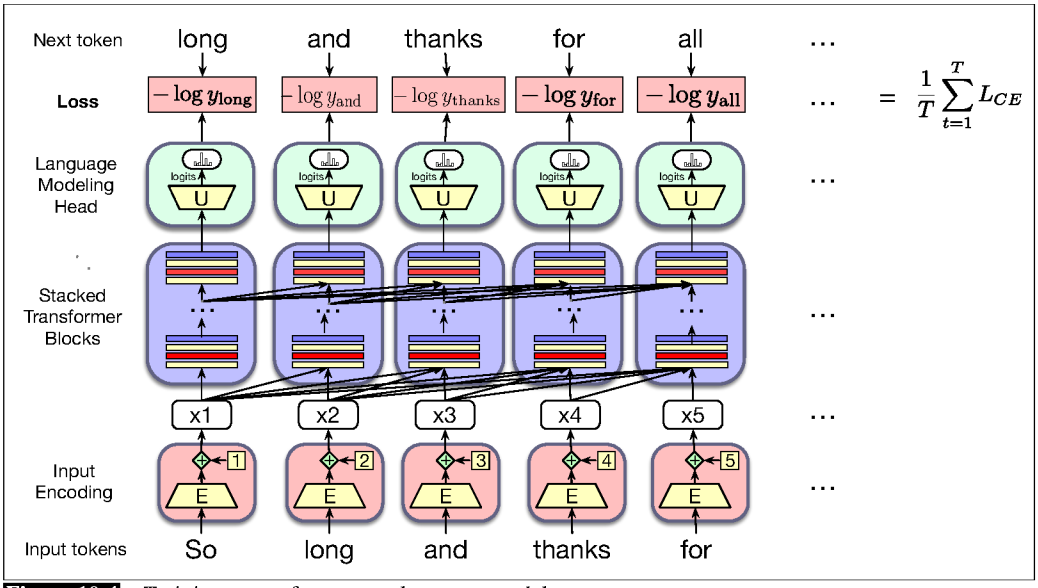


**Figure 10.4** Training a transformer as a language model.

Note the key difference between this figure and the earlier RNN-based version shown in Fig. 8.6. There the calculation of the outputs and the losses at each step was inherently serial given the recurrence in the calculation of the hidden states. With transformers, each training item can be processed in parallel since the output for each element in the sequence is computed separately.

Large models are generally trained by filling the full context window (for example 4096 tokens for GPT4 or 8192 for Llama 3) with text. If documents are shorter than this, multiple documents are packed into the window with a special end-of-text token between them. The batch size for gradient descent is usually quite large (the largest GPT-3 model uses a batch size of 3.2 million tokens).

### 10.3.2 Training corpora for large language models

Large language models are mainly trained on text scraped from the web, augmented by more carefully curated data. Because these training corpora are so large, they are likely to contain many natural examples that can be helpful for NLP tasks, such as question and answer pairs (for example from FAQ lists), translations of sentences between various languages, documents together with their summaries, and so on.

common crawl

Web text is usually taken from corpora of automatically-crawled web pages like the **common crawl**, a series of snapshots of the entire web produced by the non-profit Common Crawl (`https://commoncrawl.org/`) that each have billions of webpages. Various versions of common crawl data exist, such as the Colossal Clean Crawled Corpus (C4; Raffel et al. 2020), a corpus of 156 billion tokens of English that is filtered in various ways (deduplicated, removing non-natural language like code, sentences with offensive words from a blocklist). This C4 corpus seems to consist in large part of patent text documents, Wikipedia, and news sites (Dodge et al., 2021).

The Pile

Wikipedia plays a role in lots of language model training, as do corpora of books. **The Pile** (Gao et al., 2020) is an 825 GB English text corpus that is constructed by publicly released code, containing again a large amount of text scraped from the web

as well as books and Wikipedia; Fig. 10.5 shows its composition. Dolma is a larger open corpus of English, created with public tools, containing three trillion tokens, which similarly consists of web text, academic papers, code, books, encyclopedic materials, and social media (Soldaini et al., 2024).

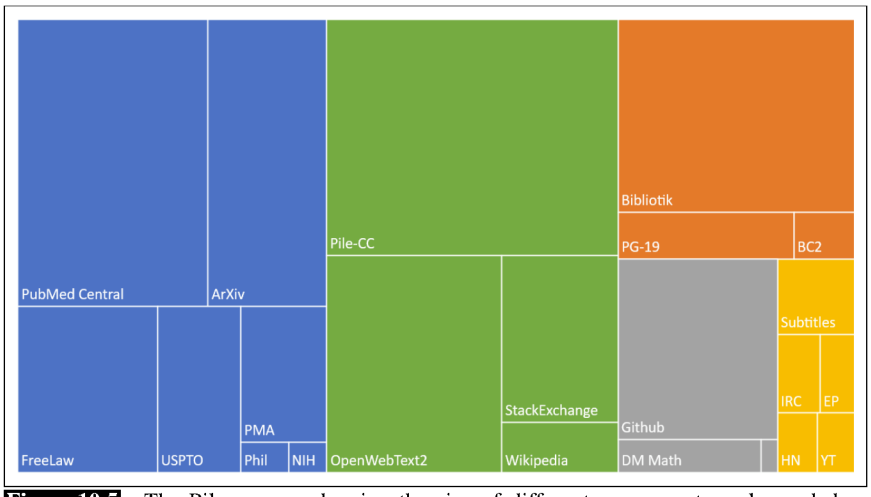


**Figure 10.5** The Pile corpus, showing the size of different components, color coded as academic (articles from PubMed and ArXiv, patents from the USPTA; internet (webtext including a subset of the common crawl as well as Wikipedia), prose (a large corpus of books), dialogue (including movie subtitles and chat data), and misc.. Figure from Gao et al. (2020).

**Filtering for quality and safety** Pretraining data drawn from the web is filtered for both **quality** and **safety**. Quality filters are classifiers that assign a score to each document. Quality is of course subjective, so different quality filters are trained in different ways, but often to value high-quality reference corpora like Wikipedia, books, and particular websites and to avoid websites with lots of **PII** (Personal Identifiable Information) or adult content. Filters also remove boilerplate text which is very frequent on the web. Another kind of quality filtering is deduplication, which can be done at various levels, so as to remove duplicate documents, duplicate web pages, or duplicate text. Quality filtering generally improves language model performance (Longpre et al., 2024b; Llama Team, 2024).

PII

Safety filtering is again a subjective decision, and often includes **toxicity** detection based on running off-the-shelf toxicity classifiers. This can have mixed results. One problem is that current toxicity classifiers mistakenly flag non-toxic data if it is generated by speakers of minority dialects like African American English (Xu et al., 2021). Another problem is that models trained on toxicity-filtered data, while somewhat less toxic, are also worse at detecting toxicity themselves (Longpre et al., 2024b). These issues make the question of how to do better safety filtering an important open problem.

Using large datasets scraped from the web to train language models poses ethical and legal questions:

**Copyright:** Much of the text in these large datasets (like the collections of fiction and non-fiction books) is copyrighted. In some countries, like the United States, the **fair use** doctrine may allow copyrighted content to be used for transformative uses, but it's not clear if that remains true if the language models are used to generate text that competes with the market for the text they

are trained on (Henderson et al., 2023).

**Data consent:** Owners of websites can indicate that they don't want their sites to be crawled by web crawlers (either via a robots.txt file, or via Terms of Service). Recently there has been a sharp increase in the number of websites that have indicated that they don't want large language model builders crawling their sites for training data (Longpre et al., 2024a). Because it's not clear what legal status these indications have in different countries, or whether these restrictions are retroactive, what effect this will have on large pretraining datasets is unclear.

**Privacy:** Large web datasets also have **privacy** issues since they contain private information like phone numbers and IP addresses. While filters are used to try to remove websites likely to contain large amounts of personal information, such filtering isn't sufficient.

### 10.3.3 Finetuning

Although the enormous pretraining data for a large language model includes text from many domains, it's often the case that we want to apply it in a new domain or task that might not have appeared sufficiently in the pre-training data. For example, we might want a language model that's specialized to legal or medical text. Or we might have a multilingual language model that knows many languages but might benefit from some more data in our particular language of interest. Or we want a language model that is specialized to a particular task.

In such cases, we can simply continue training the model on relevant data from the new domain or language (Gururangan et al., 2020). This process of taking a fully pretrained model and running additional training passes on some new data is called **finetuning**. Fig. 10.6 sketches the paradigm.

finetuning

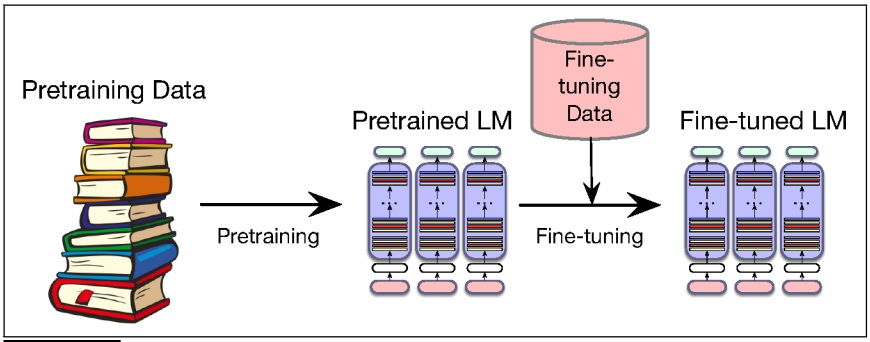


**Figure 10.6** Pretraining and finetuning. A pre-trained model can be finetuned to a particular domain, dataset, or task. There are many different ways to finetune, depending on exactly which parameters are updated from the finetuning data: all the parameters, some of the parameters, or only the parameters of specific extra circuitry.

We'll introduce four related kinds of finetuning in this chapter and the two following chapters. In all four cases, finetuning means the process of taking a pretrained model and further adapting some or all of its parameters to some new data. But they differ on exactly which parameters get updated.

In the first kind of finetuning we retrain all the parameters of the model on this new data, using the same method (word prediction) and loss function (cross-entropy loss) as for pretraining. In a sense it's as if the new data were at the tail end of

continued pretraining

the pretraining data, and so you'll sometimes see this method called **continued pretraining**.

freeze

Retraining all the parameters of the model is very slow and expensive when the language model is huge. So instead we can **freeze** some of the parameters (i.e., leave them unchanged from their pretrained value) and train only a subset of parameters on the new data. In Section 10.5.3 we'll describe this second variety of finetuning, called **parameter-efficient finetuning**, or **PEFT**. because we efficiently select specific parameters to update when finetuning, and leave the rest in their pretrained values.

In Chapter 11 we'll introduce a third kind of finetuning, also parameter-efficient. In this version, the goal is to use a language model as a kind of classifier or labeler for a specific task. For example we might train the model to be a sentiment classifier. We do this by adding extra neural circuitry (an extra **head**) after the top layer of the model. This classification head takes as input some of the top layer embeddings of the transformer and produces as output a classification. In this method, most commonly used with masked language models like BERT, we freeze the entire pretrained model and only train the classification head on some new data, usually labeled with some class that we want to predict.

Finally, in Chapter 12 we'll introduce a fourth kind of finetuning, that is a crucial component of the largest language models: **supervised finetuning** or **SFT**. SFT is often used for **instruction finetuning**, in which we want a pretrained language model to learn to follow text instructions, for example to answer questions or follow a command to write something. Here we create a dataset of prompts and desired responses (for example questions and their answers, or commands and their fulfillments), and we train the language model using the normal cross-entropy loss to predict each token in the instruction prompt iteratively, essentially training it to produce the desired response from the command in the prompt. It's called supervised because unlike in pretraining, where we just take any data and predict the words in it, we build the special finetuning dataset by hand, creating supervised responses to each command.

Often everything that happens after pretraining is lumped together as **post-training**; we'll discuss the various parts of post-training in Chapter 12.

## 10.4 Evaluating Large Language Models

**Perplexity** As we first saw in Chapter 3, one way to evaluate language models is to measure how well they predict unseen text. Intuitively, good models are those that assign higher probabilities to unseen data (are less surprised when encountering the new words).

perplexity

We instantiate this intuition by using **perplexity** to measure the quality of a language model. Recall from page 40 that the perplexity of a model $\theta$ on an unseen test set is the inverse probability that $\theta$ assigns to the test set, normalized by the test set length. For a test set of $n$ tokens $w_{1:n}$, the perplexity is

$$\begin{aligned} \text{Perplexity}_\theta(w_{1:n}) &= P_\theta(w_{1:n})^{-\frac{1}{n}} \\ &= \sqrt[n]{\frac{1}{P_\theta(w_{1:n})}} \end{aligned} \tag{10.7}$$

To visualize how perplexity can be computed as a function of the probabilities the

LM computes for each new word, we can use the chain rule to expand the computation of probability of the test set:

$$\text{Perplexity}_\theta(w_{1:n}) = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P_\theta(w_i|w_{<i})}} \tag{10.8}$$

Note that because of the inverse in Eq. 10.7, the higher the probability of the word sequence, the lower the perplexity. Thus the **the lower the perplexity of a model on the data, the better the model**. Minimizing perplexity is equivalent to maximizing the test set probability according to the language model.

One caveat: because perplexity depends on the length of a text, it is very sensitive to differences in the tokenization algorithm. That means that it's hard to exactly compare perplexities produced by two language models if they have very different tokenizers. For this reason perplexity is best used when comparing language models that use the same tokenizer.

**Other factors** While the predictive accuracy of a language model, as measured by perplexity, is a very useful metric, we also care about different kinds of accuracy, for the downstream tasks we apply our language model to. For each task like machine translation, summarization, question answering, speech recognition, and dialogue, we can measure the accuracy at those tasks. Future chapters will introduce task-specific metrics that allow us to evaluate how accuracy or correct language models are at these downstream tasks.

But when evaluating models we also care about factors besides any of these kinds of accuracy (Dodge et al., 2019; Ethayarajh and Jurafsky, 2020). For example, we often care about how a big a model is, and how long it takes to train or do inference. This can matter because we have constraints on time either for training or at inference. Or we may have constraints on memory, since the GPUs we run our models on have fixed memory sizes. Big models also use more energy, and we prefer models that use less energy, both to reduce the environmental impact of the model and to reduce the financial cost of building or deploying it. We can target our evaluation to these factors by measuring performance normalized to a giving compute or memory budget. We can also directly measure the energy usage of our model in kWh or in kilograms of $CO_2$ emitted (Strubell et al., 2019; Henderson et al., 2020; Liang et al., 2023).

Another feature that a language model evaluation can measure is fairness. We know that language models are biased, exhibiting gendered and racial stereotypes, or decreased performance for language from or about certain demographics groups. There are language model evaluation benchmarks that measure the strength of these biases, such as StereoSet (Nadeem et al., 2021), RealToxicityPrompts (Gehman et al., 2020), and BBQ (Parrish et al., 2022) among many others. We also want language models whose performance is equally fair to different groups. For example, we could chose an evaluation that is fair in a Rawlsian sense by maximizing the welfare of the worst-off group (Rawls, 2001; Hashimoto et al., 2018; Sagawa et al., 2020).

Finally, there are many kinds of leaderboards like Dynabench (Kiela et al., 2021) and general evaluation protocols like HELM (Liang et al., 2023); we will return to these in later chapters when we introduce evaluation metrics for specific tasks like question answering and information retrieval.

# 10.5 Dealing with Scale

Large language models are large. For example the *Llama 3.1 405B Instruct* model from Meta has 405 billion parameters ($L$=126 layers, a model dimensionality of $d$=16,384, $A$=128 attention heads) and was trained on 15.6 terabytes of text tokens (Llama Team, 2024), using a vocabulary of 128K tokens. So there is a lot of research on understanding how LLMs scale, and especially how to implement them given limited resources. In the next few sections we discuss how to think about scale (the concept of **scaling laws**), and important techniques for getting language models to work efficiently, such as the **KV cache** and parameter-efficient fine tuning.

## 10.5.1 Scaling laws

The performance of large language models has shown to be mainly determined by 3 factors: model size (the number of parameters not counting embeddings), dataset size (the amount of training data), and the amount of compute used for training. That is, we can improve a model by adding parameters (adding more layers or having wider contexts or both), by training on more data, or by training for more iterations.

scaling laws

The relationships between these factors and performance are known as **scaling laws**. Roughly speaking, the performance of a large language model (the loss) scales as a power-law with each of these three properties of model training.

For example, Kaplan et al. (2020) found the following three relationships for loss $L$ as a function of the number of non-embedding parameters $N$, the dataset size $D$, and the compute budget $C$, for models training with limited parameters, dataset, or compute budget, if in each case the other two properties are held constant:

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N} \tag{10.9}$$

$$L(D) = \left(\frac{D_c}{D}\right)^{\alpha_D} \tag{10.10}$$

$$L(C) = \left(\frac{C_c}{C}\right)^{\alpha_C} \tag{10.11}$$

The number of (non-embedding) parameters $N$ can be roughly computed as follows (ignoring biases, and with $d$ as the input and output dimensionality of the model, $d_{\text{attn}}$ as the self-attention layer size, and $d_{\text{ff}}$ the size of the feedforward layer):

$$\begin{aligned} N &\approx 2\, d\, n_{\text{layer}}(2\, d_{\text{attn}} + d_{\text{ff}}) \\ &\approx 12\, n_{\text{layer}}\, d^2 \\ &\quad (\text{assuming } d_{\text{attn}} = d_{\text{ff}}/4 = d) \end{aligned} \tag{10.12}$$

Thus GPT-3, with $n = 96$ layers and dimensionality $d = 12288$, has $12 \times 96 \times 12288^2 \approx 175$ billion parameters.

The values of $N_c$, $D_c$, $C_c$, $\alpha_N$, $\alpha_D$, and $\alpha_C$ depend on the exact transformer architecture, tokenization, and vocabulary size, so rather than all the precise values, scaling laws focus on the relationship with loss.[2]

Scaling laws can be useful in deciding how to train a model to a particular performance, for example by looking at early in the training curve, or performance with

[2] For the initial experiment in Kaplan et al. (2020) the precise values were $\alpha_N = 0.076$, $N_c = 8.8 \times 10^{13}$ (parameters), $\alpha_D = 0.095$, $D_c = 5.4 \times 10^{13}$ (tokens), $\alpha_C = 0.050$, $C_c = 3.1 \times 10^8$ (petaflop-days).

smaller amounts of data, to predict what the loss would be if we were to add more data or increase model size. Other aspects of scaling laws can also tell us how much data we need to add when scaling up a model.

### 10.5.2 KV Cache

We saw in Fig. 9.10 and in Eq. 9.33 (repeated below) how the attention vector can be very efficiently computed in parallel for training, via two matrix multiplications:

$$\mathbf{A} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d_k}}\right)\mathbf{V} \tag{10.13}$$

Unfortunately we can't do quite the same efficient computation in inference as in training. That's because at inference time, we iteratively generate the next tokens one at a time. For a new token that we have just generated, call it $\mathbf{x}_i$, we need to compute its query, key, and values by multiplying by $\mathbf{W}^\mathbf{Q}$, $\mathbf{W}^\mathbf{K}$, and $\mathbf{W}^\mathbf{V}$ respectively. But it would be a waste of computation time to recompute the key and value vectors for all the **prior** tokens $\mathbf{x}_{<i}$; at prior steps we already computed these key and value vectors! So instead of recomputing these, whenever we compute the key and value vectors we store them in memory in the **KV cache**, and then we can just grab them from the cache when we need them. Fig. 10.7 modifies Fig. 9.10 to show the computation that takes place for a single new token, showing which values we can take from the cache rather than recompute.

KV cache

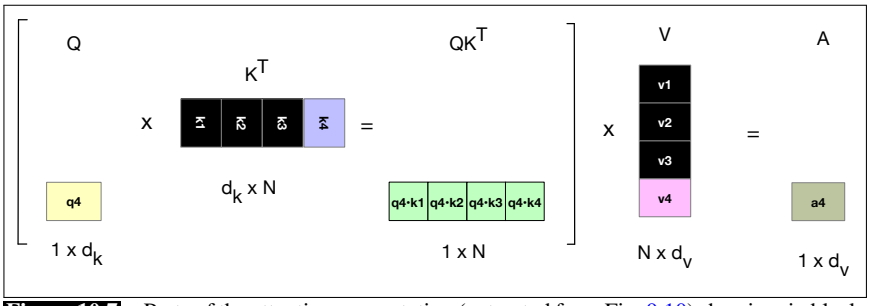


**Figure 10.7** Parts of the attention computation (extracted from Fig. 9.10) showing, in black, the vectors that can be stored in the cache rather than recomputed when computing the attention score for the 4th token.

### 10.5.3 Parameter Efficient Fine Tuning

As we mentioned above, it's very common to take a language model and give it more information about a new domain by **finetuning** it (continuing to train it to predict upcoming words) on some additional data.

Fine-tuning can be very difficult with very large language models, because there are enormous numbers of parameters to train; each pass of batch gradient descent has to backpropagate through many many huge layers. This makes finetuning huge language models extremely expensive in processing power, in memory, and in time. For this reason, there are alternative methods that allow a model to be finetuned without changing all the parameters. Such methods are called **parameter-efficient fine tuning** or sometimes **PEFT**, because we efficiently select a subset of parameters to update when finetuning. For example we freeze some of the parameters (don't change them), and only update some particular subset of parameters.

parameter-efficient fine tuning
PEFT

LoRA

Here we describe one such model, called **LoRA**, for **Lo**w-**R**ank **A**daptation. The intuition of LoRA is that transformers have many dense layers which perform matrix multiplication (for example the $\mathbf{W^Q}$, $\mathbf{W^K}$, $\mathbf{W^V}$, $\mathbf{W^O}$ layers in the attention computation). Instead of updating these layers during finetuning, with LoRA we freeze these layers and instead update a low-rank approximation that has fewer parameters.

Consider a matrix $\mathbf{W}$ of dimensionality $[N \times d]$ that needs to be updated during finetuning via gradient descent. Normally this matrix would get updates $\Delta\mathbf{W}$ of dimensionality $[N \times d]$, for updating the $N \times d$ parameters after gradient descent. In LoRA, we freeze $\mathbf{W}$ and update instead a low-rank decomposition of $\mathbf{W}$. We create two matrices $\mathbf{A}$ and $\mathbf{B}$, where $\mathbf{A}$ has size $[N \times r]$ and $\mathbf{B}$ has size $[r \times d]$, and we choose $r$ to be quite small, $r << \min(d,N)$. During finetuning we update $\mathbf{A}$ and $\mathbf{B}$ instead of $\mathbf{W}$. That is, we replace $\mathbf{W} + \Delta\mathbf{W}$ with $\mathbf{W} + \mathbf{BA}$. Fig. 10.8 shows the intuition. For replacing the forward pass $\mathbf{h} = \mathbf{xW}$, the new forward pass is instead:

$$\mathbf{h} = \mathbf{xW} + \mathbf{xAB} \tag{10.14}$$

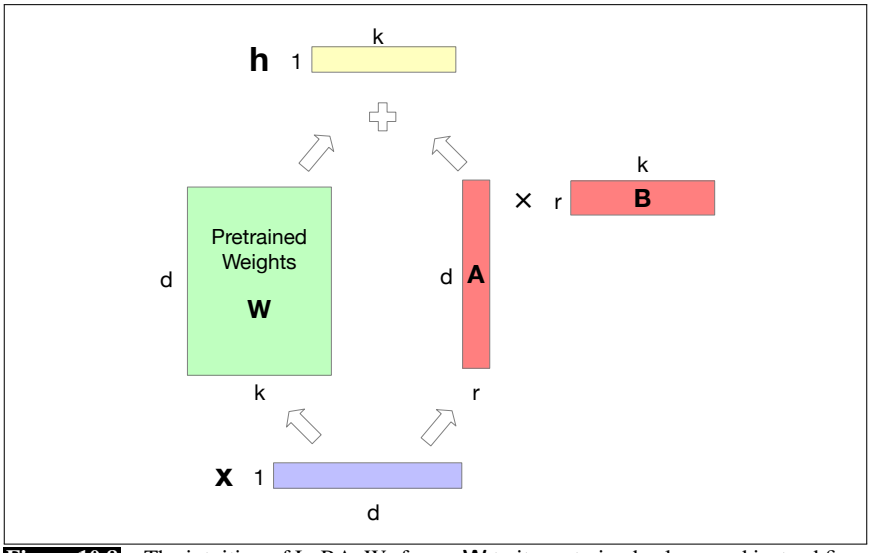


**Figure 10.8** The intuition of LoRA. We freeze $\mathbf{W}$ to its pretrained values, and instead finetune by training a pair of matrices $\mathbf{A}$ and $\mathbf{B}$, updating those instead of $\mathbf{W}$, and just sum $\mathbf{W}$ and the updated $\mathbf{AB}$.

LoRA has a number of advantages. It dramatically reduces hardware requirements, since gradients don't have to be calculated for most parameters. The weight updates can be simply added in to the pretrained weights, since $\mathbf{BA}$ is the same size as $\mathbf{W}$). That means it doesn't add any time during inference. And it also means it's possible to build LoRA modules for different domains and just swap them in and out by adding them in or subtracting them from $\mathbf{W}$.

In its original version LoRA was applied just to the matrices in the attention computation (the $\mathbf{W^Q}$, $\mathbf{W^K}$, $\mathbf{W^V}$, and $\mathbf{W^O}$ layers). Many variants of LoRA exist.

## 10.6 Potential Harms from Language Models

Large pretrained neural language models exhibit many of the potential harms discussed in Chapter 4 and Chapter 6. Many of these harms become realized when pretrained language models are used for any downstream task, particularly those involving text generation, whether question answering, machine translation, or in assistive technologies like writing aids or web search query completion, or predictive typing for email (Olteanu et al., 2020).

**hallucination**

For example, language models are prone to saying things that are false, a problem called **hallucination**. Language models are trained to generate text that is predictable and coherent, but the training algorithms we have seen so far don't have any way to enforce that the text that is generated is correct or true. This causes enormous problems for any application where the facts matter! We'll return to this issue in Chapter 14 where we introduce proposed mitigation methods like **retrieval augmented generation**.

**toxic language**

A second source of harm is that language models can generate **toxic language**. Gehman et al. (2020) show that even completely non-toxic prompts can lead large language models to output hate speech and abuse their users. Language models also generate stereotypes (Cheng et al., 2023) and negative attitudes (Brown et al., 2020; Sheng et al., 2019) about many demographic groups.

One source of biases is the training data. Gehman et al. (2020) shows that large language model training datasets include toxic text scraped from banned sites. There are other biases than toxicity: the training data is disproportionately generated by authors from the US and from developed countries. Such biased population samples likely skew the resulting generation toward the perspectives or topics of this group alone. Furthermore, language models can amplify demographic and other biases in training data, just as we saw for embedding models in Chapter 6.

Datasets can be another source of harms. We already saw in Section 10.3.2 that using pretraining corpora scraped from the web can lead to harms related to copyright and data consent. We also mentioned that pretraining data can tend to have private information like phone numbers and addresses. This is problematic because large language models can **leak** information from their training data. That is, an adversary can extract training-data text from a language model such as a person's name, phone number, and address (Henderson et al. 2017, Carlini et al. 2021). This becomes even more problematic when large language models are trained on extremely sensitive private datasets such as electronic health records.

Language models can also be used by malicious actors for generating text for **misinformation**, phishing, or other socially harmful activities (Brown et al., 2020). McGuffie and Newhouse (2020) show how large language models generate text that emulates online extremists, with the risk of amplifying extremist movements and their attempt to radicalize and recruit.

Finding ways to mitigate all these harms is an important current research area in NLP. At the very least, carefully analyzing the data used to pretrain large language models is important as a way of understanding issues of toxicity, bias, privacy, and fair use, making it extremely important that language models include **datasheets** (page 16) or **model cards** (page 74) giving full replicable information on the corpora used to train them. Open-source models can specify their exact training data. Requirements that models are transparent in such ways is also in the process of being incorporated into the regulations of various national governments.

## 10.7 Summary

This chapter has introduced the large language model, and how it can be built out of the transformer. Here's a summary of the main points that we covered:

- Many NLP tasks—such as question answering, summarization, sentiment, and machine translation—can be cast as tasks of word prediction and hence addressed with Large language models.
- Large language models are generally pretrained on large datasets of 100s of billions of words generally scraped from the web.
- These datasets need to be filtered for quality and balanced for domains by upsampling and downsampling. Addressing some problems with pretraining data, like toxicity, are open research problems.
- The choice of which word to generate in large language models is generally done by using a **sampling** algorithm.
- Language models are evaluated by **perplexity** but there are also evaluations of accuracy downstream tasks, and ways to measure other factors like fairness and energy use.
- There are various computational tricks for making large language models more efficient, such as the **KV cache** and **parameter-efficient finetuning**.
- Because of their ability to be used in so many ways, language models also have the potential to cause harms. Some harms include hallucinations, bias, stereotypes, misinformation and propaganda, and violations of privacy and copyright.

## Bibliographical and Historical Notes

As we discussed in Chapter 3, the earliest language models were the n-gram language models developed (roughly simultaneously and independently) by Fred Jelinek and colleagues at the IBM Thomas J. Watson Research Center, and James Baker at CMU. It was the Jelinek and the IBM team who first coined the term **language model** to mean a model of the way any kind of linguistic property (grammar, semantics, discourse, speaker characteristics), influenced word sequence probabilities (Jelinek et al., 1975). They contrasted the language model with the **acoustic model** which captured acoustic/phonetic characteristics of phone sequences.

N-gram language models were very widely used over the next 30 years and more, across a wide variety of NLP tasks like speech recognition and machine translations, often as one of multiple components of the model. The contexts for these n-gram models grew longer, with 5-gram models used quite commonly by very efficient LM toolkits (Stolcke, 2002; Heafield, 2011).

The roots of the neural language model lie in multiple places. One was the application in the 1990s, again in Jelinek's group at IBM Research, of **discriminative classifiers** to language models. Roni Rosenfeld in his dissertation (Rosenfeld, 1992) first applied logistic regression (under the name **maximum entropy** or **maxent** models) to language modeling in that IBM lab, and published a more fully formed version in Rosenfeld (1996). His model integrated various sorts of information in a logistic regression predictor, including n-gram information along with

other features from the context, including distant n-grams and pairs of associated words called **trigger pairs**. Rosenfeld's model prefigured modern language models by being a statistical word predictor trained in a self-supervised manner simply by learning to predict upcoming words in a corpus.

Another was the first use of pretrained embeddings to model word meaning in the LSA/LSI models (Deerwester et al., 1988). Recall from the history section of Chapter 6 that in LSA (latent semantic analysis) a term-document matrix was trained on a corpus and then singular value decomposition was applied and the first 300 dimensions were used as a vector embedding to represent words. Landauer et al. (1997) first used the word "embedding". In addition to their development of the idea of pretraining and of embeddings, the LSA community also developed ways to combine LSA embeddings with n-grams in an integrated language model (Bellegarda, 1997; Coccaro and Jurafsky, 1998).

In a very influential series of papers developing the idea of **neural language models**, (Bengio et al. 2000; Bengio et al. 2003; Bengio et al. 2006), Yoshua Bengio and colleagues drew on the central ideas of both these lines of self-supervised language modeling work, (the discriminatively trained word predictor, and the pretrained embeddings). Like the maxent models of Rosenfeld, Bengio's model used the next word in running text as its supervision signal. Like the LSA models, Bengio's model learned an embedding, but unlike the LSA models did it as part of the process of language modeling. The Bengio et al. (2003) model was a neural language model: a neural network that learned to predict the next word from prior words, and did so via learning embeddings as part of the prediction process.

The neural language model was extended in various ways over the years, perhaps most importantly in the form of the RNN language model of Mikolov et al. (2010) and Mikolov et al. (2011). The RNN language model was perhaps the first neural model that was accurate enough to surpass the performance of a traditional 5-gram language model.

Soon afterwards, Mikolov et al. (2013a) and Mikolov et al. (2013b) proposed to simplify the hidden layer of these neural net language models to create pretrained word2vec word embeddings.

The static embedding models like LSA and word2vec instantiated a particular model of pretraining: a representation was trained on a pretraining dataset, and then the representations could be used in further tasks. 'Dai and Le (2015) and (Peters et al., 2018) reframed this idea by proposing models that were pretrained using a language model objective, and then the identical model could be either frozen and directly applied for language modeling or further finetuned still using a language model objective. For example ELMo used a biLSTM self-supervised on a large pretrained dataset using a language model objective, then finetuned on a domain-specific dataset, and then froze the weights and added task-specific heads. The ELMo work was particularly influential and its appearance was perhaps the moment when it became clear to the community that language models could be used as a general solution for NLP problems.

Transformers were first applied as encoder-decoders (Vaswani et al., 2017) and then to masked language modeling (Devlin et al., 2019) (as we'll see in Chapter 13 and Chapter 11). Radford et al. (2019) then showed that the transformer-based autoregressive language model GPT2 could perform zero-shot on many NLP tasks like summarization and question answering.

The technology used for transformer-based language models can also be applied to other domains and tasks, like vision, speech, and genetics. the term **foundation**

foundation model

**model** is sometimes used as a more general term for this use of large language model technology across domains and areas, when the elements we are computing over are not necessarily words. Bommasani et al. (2021) is a broad survey that sketches the opportunities and risks of foundation models, with special attention to large language models.