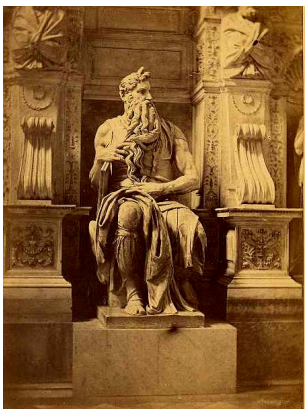
CHAPTER

7 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*

The literature of the fantastic abounds in inanimate objects magically endowed with the gift of speech. From Ovid's statue of Pygmalion to Mary Shelley's story about Frankenstein, we continually reinvent stories about creating something and then having a chat with it. Legend has it that after finishing his sculpture *Moses*, Michelangelo thought it so lifelike that he tapped it on the knee and commanded it to speak. Perhaps this shouldn't be surprising. Language is the mark of humanity and sentience. conversation is the most fundamental arena of language, the first kind of language we learn as children, and the kind we engage in constantly, whether we are teaching or learning, ordering lunch, or talking with our families or friends.



This chapter introduces the **Large Language Model**, or **LLM**, a computational agent that can interact conversationally with people. The fact that LLMs are designed for interaction with people has strong implications for their design and use.

Many of these implications already became clear in a computational system from 60 years ago, ELIZA (Weizenbaum, 1966). ELIZA, designed to simulate a Rogerian psychologist, illustrates a number of important issues with chatbots. For example people became deeply emotionally involved and conducted very personal conversations, even to the extent of asking Weizenbaum to leave the room while they were typing. These issues of emotional engagement and privacy mean we need to think carefully about how we deploy language models and consider their effect on the people who are interacting with them.

In this chapter we begin by introducing the computational principles of LLMs; we'll discuss their implementation in the transformer architecture in the following chapter. The central new idea that makes LLMs possible is the idea of **pretraining**, so let's begin by thinking about the idea of learning from text, the basic way that LLMs are trained.

We know that 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. 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? Research suggests that the bulk of this knowledge acquisition happens as a by-product of reading. Reading is a process of rich contextual processing; we don't learn words one at a time in isolation. In fact, at some points during learning the rate of vocabulary growth exceeds the rate at which new words are appearing to the learner! That suggests that every time we read a word, we are also strengthening our understanding of other words that are associated with it.

Such facts are consistent with the *distributional hypothesis* of Chapter 5, which proposes that some 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.

What made the modern NLP revolution possible is that large language models can learn all this knowledge of language, context, and the world simply by being taught to predict the next word, again and again, based on context, in a (very) large corpus of text. In this chapter and the next we formalize this idea that we'll call **pretraining**—learning knowledge about language and the world from iteratively predicting tokens in vast amounts of text—and call the resulting pretrained models **large language models**. Large language models exhibit remarkable performance on natural language tasks because of the knowledge they learn in pretraining.

pretraining

What can language models learn from word prediction? Consider the examples below. What kinds of knowledge do you think the model might pick up from learning to predict what word fills the underbar (the correct answer is shown in blue)? Think about this for each example before you read ahead to the next paragraph:.

With roses, dahlias, and peonies, I was surrounded by _____ flowers
The room wasn't just big it was _____ enormous
The square root of 4 is _____ 2
The author of "A Room of One's Own" is _____ Virginia Woolf
The professor said that _____ he

From the first sentence a model can learn ontological facts like that roses and dahlias and peonies are all kinds of flowers. From the second, a model could learn that "enormous" means something on the same scale as big but further along on the scale. From the third sentence, the system could learn math, while from the 4th sentence facts about the world and historical authors. Finally, the last sentence, if a model was exposed to such sentences repeatedly, it might learn to associate professors only with male pronouns, or other kinds of associations that might cause models to act unfairly to different people.

What is a large language model? As we saw back in Chapter 3, a language model is simply a computational system that can predict the next word from previous words. That is, given a context or prefix of words, a language model assigns a probability distribution over the possible next words. Fig. 7.1 sketches this idea.

Of course we've already seen language models! We saw n-gram language models in Chapter 3 and briefly touched on the feedforward network applied to language

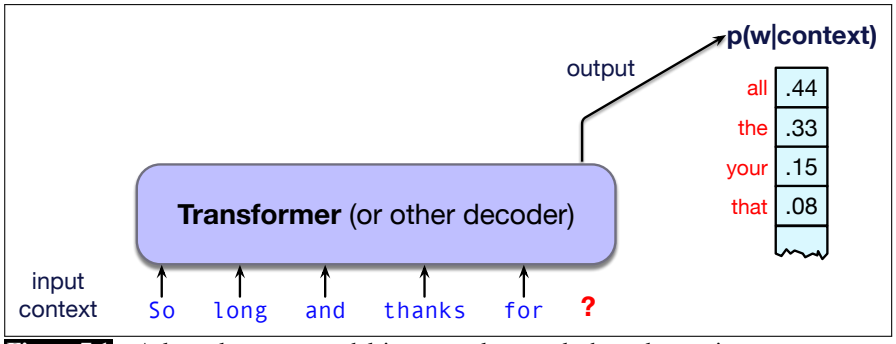


Figure 7.1 A large language model is a neural network that takes as input a context or prefix, and outputs a distribution over possible next words.

modeling in Chapter 6. A large language model is just a (much) larger version of these. For example, in Chapter 3 we introduced bigram and trigram language models that can predict words from the previous word or handful of words. By contrast, large language models can predict words given contexts of thousands or even tens of thousands of words!

The fundamental intuition of language models is that a model that can *predict* text (assigning a distribution over following words) can also be used to *generate* text by **sampling** from the distribution. Recall from Chapter 3 that sampling means to choose a word from a distribution.

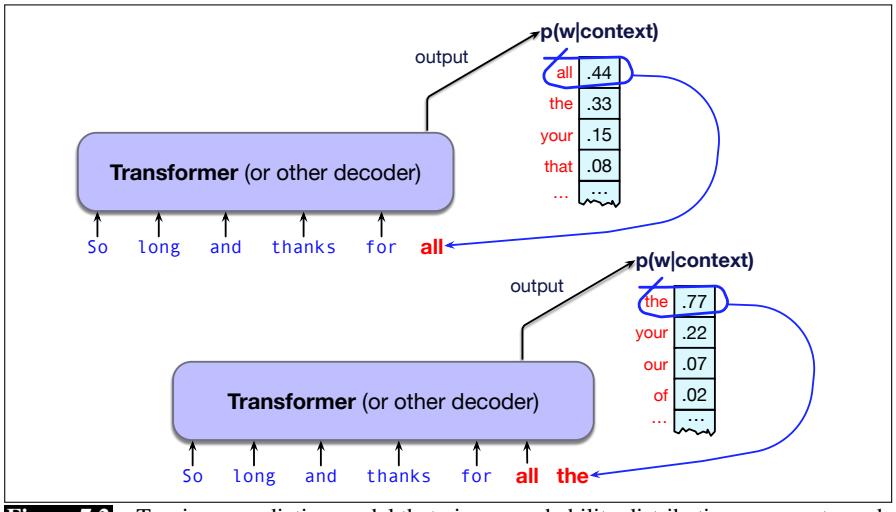


Figure 7.2 Turning a predictive model that gives a probability distribution over next words into a generative model by repeatedly sampling from the distribution. The result is a left-to-right (also called autoregressive) language model. As each token is generated, it gets added onto the context as a prefix for generating the next token.

Fig. 7.2 shows the same example from Fig. 7.1, in which a language model is given a text prefix and generates a possible completion. The model selects the word `all`, adds that to the context, uses the updated context to get a new predictive distribution, and then selects `the` from that distribution and generates it, and so on. Notice that the model is conditioning on both the priming context and its own subsequently generated outputs.

This kind of setting in which we iteratively predict and generate words left-to-

right from earlier words is often called **causal** or **autoregressive** language models. (We will introduce alternative non-autoregressive models, like BERT and other masked language models that predict words using information from both the left and the right, in Chapter 9.)

generative AI

This idea of using computational models to generate text, as well as code, speech, and images, constitutes the important new area called **generative AI**. Applying LLMs to generate text has vastly broadened the scope of NLP, which historically was focused more on algorithms for parsing or understanding text rather than generating it.

In the rest of the chapter, 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, and we'll motivate and introduce the idea of **prompting** language models. We'll introduce specific algorithms for generating text from a language model, like **greedy decoding** and **sampling**. We'll introduce the details of **pretraining**, the way that language models are self-trained by iteratively being taught to guess the next word in the text from the prior words. We'll sketch out the other two stages of language model training: instruction tuning (also called supervised finetuning or SFT), and alignment, concepts that we'll return to in Chapter 10. And we'll see how to evaluate these models. Let's begin, though, by talking about different kinds of language models.

7.1 Three architectures for language models

The architecture we sketched above for a left-to-right or autoregressive language model, which is the language model architecture we will define in this chapter, is actually only one of three common LM architectures.

The three architectures are the **encoder**, the **decoder**, and the **encoder-decoder**. Fig. 7.3 gives a schematic picture of the three.

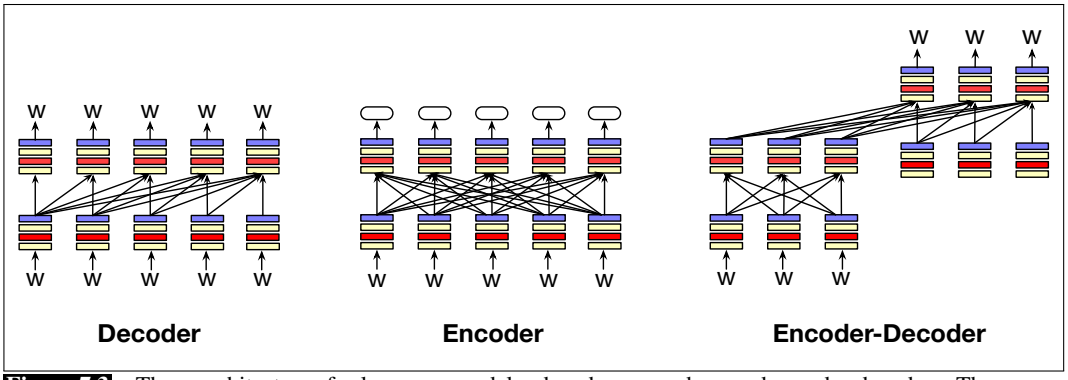


Figure 7.3 Three architectures for language models: decoders, encoders, and encoder-decoders. The arrows sketch out the information flow in the three architectures. Decoders take tokens as input and generate tokens as output. Encoders take tokens as input and produce an encoding (a vector representation of each token) as output. Encoder-decoders take tokens as input and generate a series of tokens as output.

decoder

The **decoder** is the architecture we've introduced above. It takes as input a series of tokens, and iteratively generates an output token one at a time. The decoder is the architecture used to create large language models like GPT, Claude, Llama, and Mistral. The information flow in decoders goes left-to-right, meaning that the model

predicts the next word only from the prior words. Decoders are generative models, meaning that, given input tokens, they generate novel output tokens. We'll discuss decoders in the rest of this chapter and in Chapter 8.

encoder

The **encoder** takes as input a sequence of tokens and outputs a vector representation for each token. Encoders are usually masked language models, meaning they are trained by masking out a word, and learning to predict it by looking at surrounding words on both sides. Masked language models like BERT, RoBERTA, and others in the BERT family are encoder models. Encoder models are not generative models; they aren't used to generate text. Instead encoder models are often used to create classifiers, for example where the input is text and the output is a label, for example for sentiment or topic or other classes. This is done by finetuning them (training them on supervised data). We'll introduce encoder models in Chapter 9.

encoder-decoder

The **encoder-decoder** takes as input a sequence of tokens and outputs a series of tokens. What makes it different than the decoder-only models, is that an encoder-decoder has a much looser relationship between the input tokens and the output tokens, and they are used to map between different kinds of tokens. That is, in an encoder-decoder, the output tokens might be from a very different token-set or be a much longer or shorter sequence than the input token sequence. For example encoder-decoder architectures are used for machine translation, where the input tokens are in one language and the output tokens (probably more or less of them) are in another language. Encoder-decoder architectures are also used for speech recognition, where the input is tokens representing speech, and the output is tokens representing text. We'll introduce the encoder-decoder architecture for machine translation in Chapter 12, and for speech recognition in Chapter 15.

These three architectures can be built out of many kinds of neural networks. The most widely used network type today is the **transformer** that we'll introduce in Chapter 8. In a transformer, each input token is processed by a column of transformer layers, each layer composed of a series of different kinds of subnetworks. In Chapter 13 we'll introduce an earlier architecture that is still relevant, the LSTM, a kind of recurrent neural network. And there are many more recent architectures such as the **state space models**.

We'll focus on transformers for much of this book, but for the purposes of this chapter, we'll describe the LLM decoder in a way that is architecture-agnostic, treating this network as a black box. The input to this black box is a sequence of tokens, and the output of the box is a distribution over tokens that we can sample. And we'll describe architecture-agnostic mechanisms for learning and decoding.

7.2 Conditional Generation of Text: The Intuition

conditional generation

A fundamental intuition underlying language models is that almost anything we want to do with language can be modeled as **conditional generation** of text. (We mean *decoder* language models, which are what we will discuss in this chapter and the next).

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, a **prompt**, and then have the LLM continue generating text token by token, conditioned on the prompt and the subsequently generated tokens. We generate from a model by first computing the probability of the next token w_i from the prior context: $P(w_i|w_{<i})$ and then sampling from that distribution to generate a token.

We'll talk in future sections about all the details, but in this section our goal is just to establish the intuition. How can simply computing the probability of the next token help an LLM do all sorts of different language-related tasks?

Imagine we want to do a classification tasks like sentiment analysis. We can treat this as conditional generation by giving a language model a context like:

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

and comparing the conditional probability of the following token "positive" and the following token "negative" to see which is higher. That is, as sketched in Fig. 7.4, we compare these two probabilities:

$P($"`positive`"|"`The sentiment of the sentence ‘I like Jackie Chan’ is:`"$)$
$P($"`negative`"|"`The sentiment of the sentence ‘I like Jackie Chan’ is:`"$)$

If the token "`positive`" is more probable, we could say the sentiment of the sentence is positive, otherwise if the token "`negative`" is more probable we say the sentiment is negative.

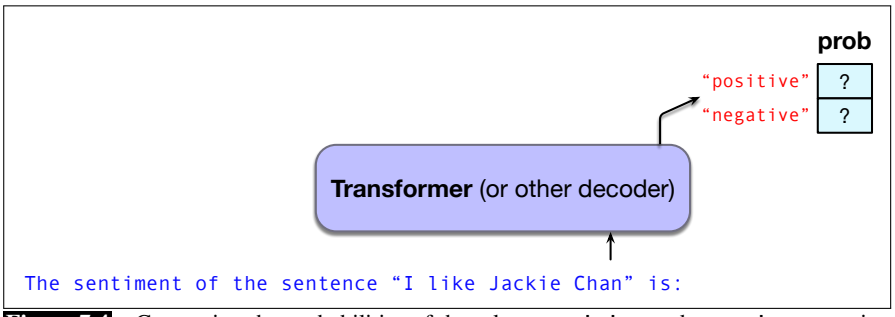


Figure 7.4 Computing the probabilities of the tokens `positive` and `negative` occurring after this prefix.

This same intuition can help us perform a task like question answering, in which the system is given a question and must give a textual answer. We can cast the task of question answering as token prediction by giving a language model a question and a token like `A:` suggesting that an answer should come next, like this:

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

Again, we can ask a language model to compute the probability distribution over possible next tokens given this prefix, computing the following probability

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

and look at which tokens w have high probabilities. As Fig. 7.5 suggests, we might expect to see that `Charles` is very likely, and then if we choose `Charles` and add that to our prefix and compute the probability over tokens with this prefix:

$P(w|$`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.

7.3 Prompting

This simple idea of conditional generation is already very powerful, but becomes more powerful when language models are specially trained to answer questions and

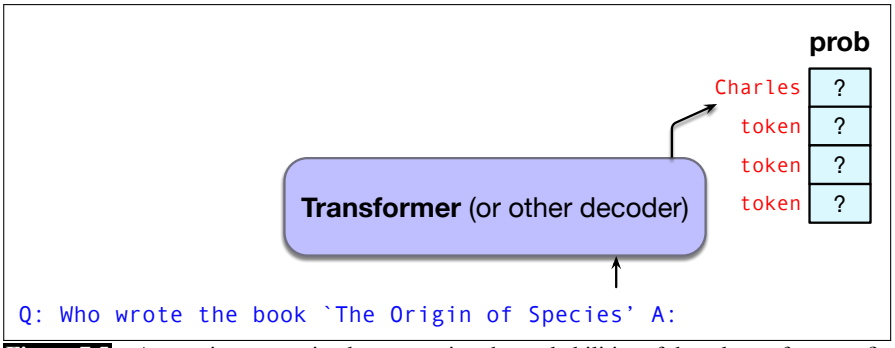


Figure 7.5 Answering a question by computing the probabilities of the tokens after a prefix stating the question; in this example the correct token `Charles` has the highest probability.

follow instructions. This extra training is called **instruction-tuning**. In instruction-tuning we take a base language model that has been trained to predict words, and continue training it on a special dataset of instructions together with the appropriate response to each. The dataset has many examples of questions together with their answers, commands with their responses, and other examples of how to carry on a conversation. We'll discuss the details of instruction-tuning in Chapter 10.

Language models that have been instruction-tuned are very good at following instructions and answering questions and carrying on a conversation and can be **prompted**. A **prompt** is a text string that a user issues to a language model to get the model to do something useful. In prompting, the user's prompt string is passed to the language model, which iteratively generates tokens conditioned on the prompt. The process of finding effective prompts for a task is known as **prompt engineering**.

prompt

prompt engineering

As suggested above when we introduced conditional generation, a prompt can be a question (like "`What is a transformer network?`"), possibly in a structured format (like "`Q: What is a transformer network? A:`"). A prompt can also be an instruction (like "`Translate the following sentence into Hindi: 'Chop the garlic finely'`").

More explicit prompts that specify the set of possible answers lead to better performance. For example, here is a prompt template to do sentiment analysis that prespecifies the potential answers:

A prompt consisting of a review plus an incomplete statement

Human: Do you think that "input" has negative or positive sentiment?
Choices:
(P) Positive
(N) Negative

Assistant: I believe the best answer is: (

This prompt uses a number of more sophisticated prompting characteristics. It specifies the two allowable choices (P) and (N), and ends the prompt with the open parenthesis that strongly suggests the answer will be (P) or (N). Note that it also specifies the role of the language model as an assistant.

Including some labeled examples in the prompt can also improve performance. We call such examples **demonstrations**. The task of prompting with examples is sometimes called **few-shot prompting**, as contrasted with **zero-shot** prompting which means instructions that don't include labeled examples. For example Fig. 7.6

demonstrations

few-shot

zero-shot

shows an example of a question using 2 demonstrations, hence 2-shot prompting. The example is drawn from a computer science question from the MMLU dataset described in Section 7.6 that is often used to evaluate language models.

Example of demonstrations in a computer science question from the MMLU dataset described in Section 7.6

The following are multiple choice questions about high school computer science.

Let x = 1. What is x << 3 in Python 3?
(A) 1 (B) 3 (C) 8 (D) 16
Answer: C

Which is the largest asymptotically?
(A) O(1) (B) O(n) (C) O(n^2) (D) O(log(n))
Answer: C

What is the output of the statement "a" + "ab" in Python 3?
(A) Error (B) aab (C) ab (D) a ab
Answer:

Figure 7.6 Sample 2-shot prompt from MMLU testing high-school computer science. (The correct answer is (B)).

Demonstrations are generally drawn from a labeled training set. They can be selected by hand, or the choice of demonstrations can be optimized by using an optimizer like DSPy (Khattab et al., 2024) to automatically choose the set of demonstrations that most increases task performance of the prompt on a dev set. The number of demonstrations doesn't need to be large; more examples seem to give diminishing returns, and too many examples seems to cause the model to overfit to the exact examples. The primary benefit of demonstrations seems more to demonstrate the task and the format of the output rather than demonstrating the right answers for any particular question. In fact, demonstrations that have incorrect answers can still improve a system (Min et al., 2022; Webson and Pavlick, 2022).

Prompts are a way to get language models to generate text, but prompts can also be viewed as a **learning** signal. This is especially clear when a prompt has demonstrations, since the demonstrations can help language models learn to perform novel tasks from these examples of the new task. This kind of learning is different than pretraining methods for setting language model weights via gradient descent methods that we will describe below. The weights of the model are not updated by prompting; what changes is just the context and the activations in the network.

in-context learning

We therefore call the kind of learning that takes place during prompting **in-context learning**—learning that improves model performance or reduces some loss but does not involve gradient-based updates to the model's underlying parameters.

system prompt

Large language models generally have a **system prompt**, a single text prompt that is the first instruction to the language model, and which defines the task or role for the LM, and sets overall tone and context. The system prompt is silently prepended to any user text. So for example a minimal system prompt that creates a multi-turn assistant conversation might be the following including some special metatokens:

```
<system>You are a helpful and knowledgeable assistant.  Answer
concisely and correctly.
```

So if a user wants to know the capital of France, the actual text used as the language model's context for conditional generation is:

```
<system> You are a helpful and knowledgeable assistant.
Answer concisely and correctly.  <user> What is the capital
of France?
```

The fact that modern language models have such long contexts (tens of thousands of tokens) makes them very powerful for conditional generation, because they can look back so far into the prompting text. That means system prompts, and prompts in general, can be very long.

For example the full system prompt for one language model, Anthropic's Claude Opus4, is 1700 words long and includes sentences like the following:

```
Claude should give concise responses to very simple questions,
but provide thorough responses to complex and open-ended
questions.
Claude is able to explain difficult concepts or ideas clearly.
It can also illustrate its explanations with examples, thought
experiments, or metaphors.
Claude does not provide information that could be used to
make chemical or biological or nuclear weapons
For more casual, emotional, empathetic, or advice-driven
conversations, Claude keeps its tone natural, warm, and
empathetic
Claude cares about people's well-being and avoids encouraging
or facilitating self-destructive behavior
If Claude provides bullet points in its response, it should
use markdown, and each bullet point should be at least 1-2
sentences long unless the human requests otherwise
```

It's also possible to create system prompts for other tasks, like the following prompt for creating a general grammar-checker (Anthropic, 2025):

```
Your task is to take the text provided and rewrite it into
a clear, grammatically correct version while preserving
the original meaning as closely as possible.  Correct any
spelling mistakes, punctuation errors, verb tense issues,
word choice problems, and other grammatical mistakes.
```

Each user can then make a prompt to have the system fix the grammar of a particular piece of text.

In all these cases, the system prompt is prepended to any user prompts or queries, and the entire string is taking as the context for conditional generation by the language model.

7.4 Generation and Sampling

Which tokens should a language model generate at each step?

The generation depends on the probability of each token, so let's remind ourselves where this probability distribution comes from. The internal networks for language models (whether transformers or alternatives like LSTMs or state space models) generate scores called **logits** (real valued numbers) for each token in the vocabulary. This score vector **u** is then normalized by softmax to be a legal probability distribution, just as we saw for logistic regression in Chapter 4. So if we have a logit vector **u** of shape $[1 \times |V|]$ that gives a score for each possible next token, we can pass it through a softmax to get a vector **y**, also of shape $[1 \times |V|]$, which assigns a probability to each token in the vocabulary, as shown in the following equation:

$$\mathbf{y} = \text{softmax}(\mathbf{u}) \tag{7.1}$$

Fig. 7.7 shows an example in which the softmax is computed for pedagogical purposes on a simplified vocabulary of only 4 words.

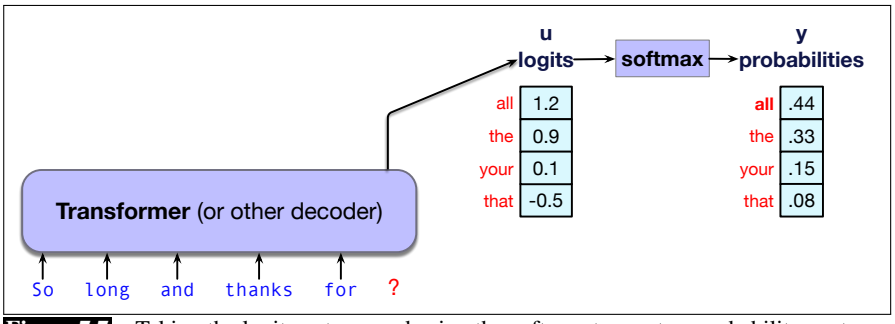


Figure 7.7 Taking the logit vector **u** and using the softmax to create a probability vector **y**.

Now given this probability distribution over tokens, we need to select one token to generate. The task of choosing a token to generate based on the model's probabilities is often called **decoding**. As we mentioned above, 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 token conditioned on our previous choices is called **causal** or **autoregressive generation**.[1]

decoding

causal
autoregressive
generation

7.4.1 Greedy decoding

The simplest way to generate tokens is to always generate the most likely token given the context, which is called **greedy decoding**. A **greedy algorithm** is one that makes 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, we turn the logits into a probability distribution over tokens and then we choose as the output w_t the token in the vocabulary that has the highest probability (the argmax):

greedy
decoding

$$\hat{w}_t = \text{argmax}_{w \in V} P(w|\mathbf{w}_{<t}) \tag{7.2}$$

Fig. 7.8 shows that in our example, the model chooses to generate `all`.

[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, as we will see, they have many layers of non-linearities), we loosely refer to this generation technique as autoregressive since the token generated at each time step is conditioned on the token selected by the network from the previous step. As we'll see, alternatives like the masked language models of Chapter 9 are non-causal because they can predict tokens based on both past and future tokens.

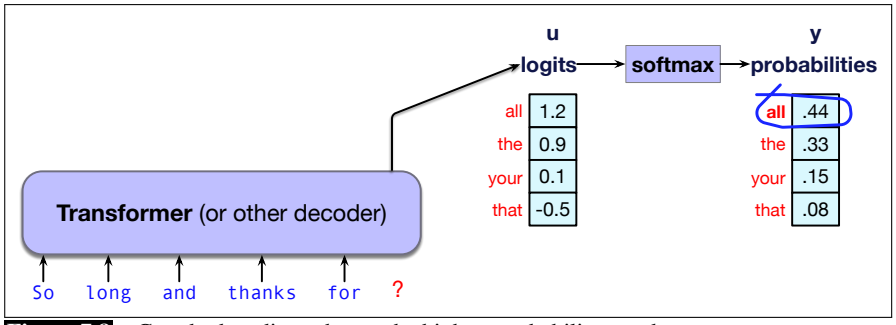


Figure 7.8 Greedy decoding: choose the highest probability word.

In practice, however, we don't use greedy decoding with large language models. A major problem with greedy decoding is that because the tokens 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 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 12 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 **sampling methods** that introduce a bit more diversity into the generations.

7.4.2 Random sampling

sampling

Thus the most common method for decoding in large language models involves **sampling**. Recall from Chapter 3 that **sampling** from a distribution means to choose random points according to their likelihood. Thus sampling from a language model—which represents a distribution over following tokens—means to choose the next token to generate according to its probability assigned by the model. Thus we are more likely to generate tokens that the model thinks have a high probability and less likely to generate tokens that the model thinks have a low probability.

That is, we randomly select a token to generate according to its probability in context as defined by the model, generate it, and iterate. We could think of this as rolling a die and choosing a token according to the resulting probability, as we saw in Chapter 3. Such a model is of course more likely to generate the highest probability token, just like the greedy algorithm, but it could also generate any token, just with smaller chances. But in general we are more likely to generate tokens that the model thinks have a high probability in the context and less likely to generate tokens that the model thinks have a low probability.

Sampling from language models was first suggested very early on by Shannon (1948) and Miller and Selfridge (1950), and we saw back in Chapter 3 on page 49 how to generate text from a unigram language model by repeatedly randomly sampling tokens according to their probability until we either reach a pre-determined length or select the end-of-sentence token.

To generate text from a large language model we'll just generalize this model a bit: at each step we'll sample tokens according to their probability *conditioned on our previous choices*, and we'll use the large language model as the probability model that tells us this probability.

random sampling

The algorithm is called **random sampling**, or **random multinomial sampling** (because we are sampling from a multinomial distribution across words). We can formalize random sampling as follows: we are generating a sequence of tokens $\{w_1, w_2, \ldots, w_N\}$ until we hit the end-of-sequence token, using $x \sim p(x)$ to mean 'choose x by sampling from the distribution $p(x)$':

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

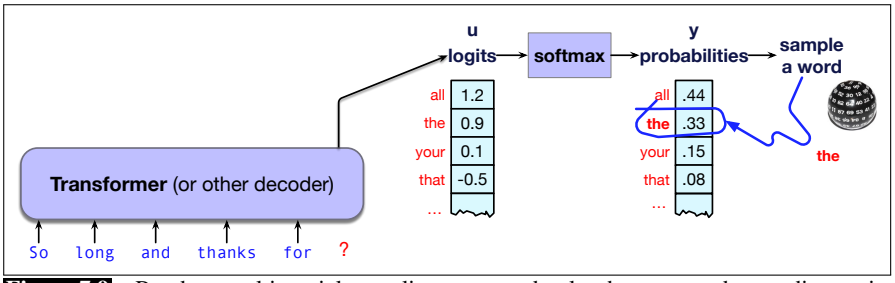


Figure 7.9 Random multinomial sampling: we randomly chose a word according to its probability.

Alas, it turns out random sampling doesn't work well either. The problem is that even though random sampling is mostly going to generate sensible, high-probable tokens, there are many odd, low-probability tokens in the tail of the distribution, and even though each one is low-probability, if you add up all the rare tokens, they constitute a large enough portion of the distribution that they get chosen often enough to result in generating weird sentences.

In other words, greedy decoding is too boring, and random sampling is too random. We need something that doesn't greedily choose the top choice every time, but doesn't stray down too far into the very low-probability events.

There are three standard sampling methods that modify random sampling to address these issues. We'll describe the most common, **temperature** sampling here, and talk about two others (**top-k** and **top-p**) in the next chapter.

7.4.3 Temperature sampling

temperature sampling

The idea of **temperature sampling** is to reshape the probability distribution to increase the probability of the high probability tokens and decrease the probability of the low probability tokens. The result is that we are less likely to generate very low-probability tokens, and more likely to generate tokens that are higher probability.

We implement this intuition by simply dividing the logit by a temperature parameter τ before 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. 7.1):

$$\mathbf{y} = \text{softmax}(\mathbf{u}) \tag{7.3}$$

we instead first divide the logits by τ, computing the probability vector y as

$$\mathbf{y} = \text{softmax}(\mathbf{u}/\tau) \tag{7.4}$$

That is, normally we convert from logits to softmax as shown in Fig. 7.10(a). But when we use a temperature parameter we first scale the logit as in Fig. 7.10(b).

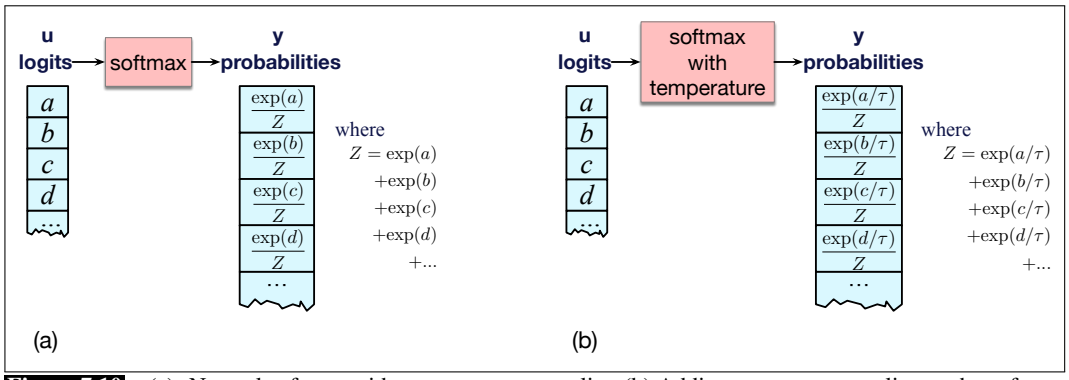


Figure 7.10 (a): Normal softmax without temperature scaling (b) Adding temperature scaling to the softmax by first dividing by the temperature parameter τ.

Why does dividing by τ increase the high probability elements and decrease the low probability elements in the vector over vocabulary items? When τ is 1, we are doing normal softmax, and so when τ is close to 1 the distribution doesn't change much. But the lower τ is, the larger the scores being passed to the softmax (because 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 tokens and decreased probabilities of the low probability tokens, making the distribution more greedy. And as τ approaches 0, dividing by τ means the probability of the most likely word approaches 1, resulting in greedy decoding.

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 tokens and decrease the probability of the rare tokens.

Fig. 7.11 shows a schematic example again simplified to have a vocabulary with only 4 tokens (*all*, *the*, *your*, *that*), and showing how different temperature values influence the probabilities computed from the initial logits. $\tau = 1$ is the normal softmax, and we can see how setting $\tau = 0.5$ increases the probability of the top candidate from .45 to .59. Setting $\tau = 0.1$ increases the probability of the top candidate to .95, getting us close to greedy decoding.

We can also see in Fig. 7.11 some other options for situations where we may want to *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 $\tau > 1$.

7.5 Training Large Language Models

How do we learn a language model? What is the algorithm and what data do we train on?

Language models are trained in three stages, as shown in Fig. 7.12:

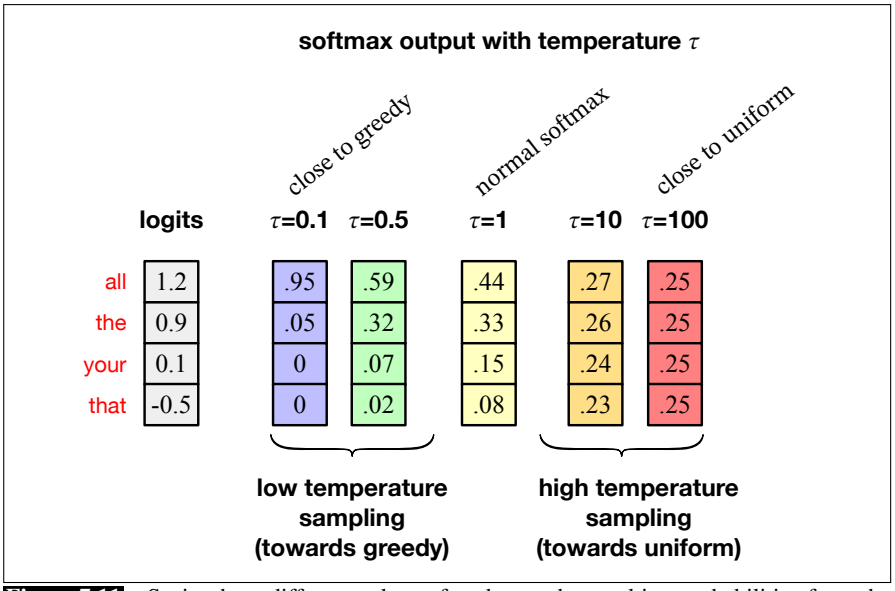


Figure 7.11 Seeing how different values of τ change the resulting probabilities from the initial logits in temperature sampling. In this simplified example, there are only 4 tokens in the vocabulary.

1. **pretraining**: In this first stage, the model is trained to incrementally predict the next word in enormous text corpora. The model uses the cross-entropy loss, sometimes called the **language modeling loss**, and that loss is backpropagated all the way through the network. The training data is usually based on cleaning up parts of the web. The result is a model that is very good at predicting words and can generate text.
2. **instruction tuning**, also called supervised finetuning or **SFT**: In the second stage, the model is trained, again by cross-entropy loss to follow instructions, for example to answer questions, give summaries, write code, translate sentences, and so on. It does this by being trained on a special corpus with lots of text containing both instructions and the correct response to the instruction.
3. **alignment**, also called **preference alignment**. In this final stage, the model is trained to make it maximally helpful and less harmful. Here the model is given preference data, which consists of a context followed by two potential continuations, which are labeled (usually by people) as an 'accepted' vs. a 'rejected' continuation. The model is then trained, by reinforcement learning or other reward-based algorithms, to produce the accepted continuation and not the rejected continuation.

We'll introduce pretraining next, but we'll save instruction tuning and preference alignment for Chapter 10.

7.5.1 Self-supervised training algorithm for pretraining

self-training

The intuition of pretraining large language models, is the same idea of **self-training** or **self-supervision** that we saw in Chapter 5 for learning word representations like word2vec. In self-training for language modeling, we take a corpus of text as training material and at each time step t ask the model to predict the next word. At first it will do poorly at this task, but since in each case we know the correct answer (it's

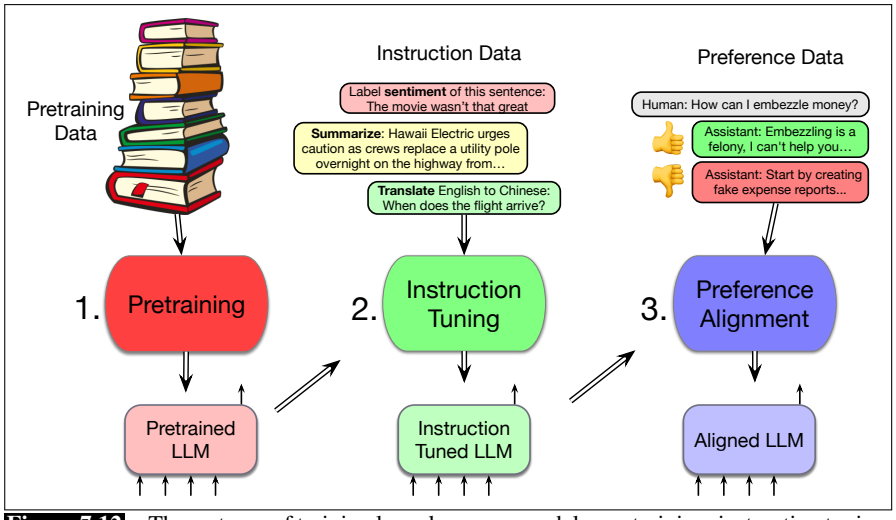


Figure 7.12 Three stages of training large language models: pretraining, instruction tuning, and preference alignment.

the next word in the corpus!) over time it well get better and better at predicting the correct 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.

In practice, training the language model means setting the parameters of the underlying architecture. The transformer that we will introduce in the next chapter has various weight matrices for its feedforward and attention components. Like any other neural architecture, they will be trained by error backpropagation with gradient descent. So all we need is a loss function to minimize and pass back through the network. The loss function we use for language modeling is the cross-entropy loss function we've now seen twice, in Chapter 4 and Chapter 6.

Recall that the cross-entropy loss measures the difference between a predicted probability distribution and the correct distribution. The probability distribution is over the token vocabulary, making the loss be:

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w] \tag{7.5}$$

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 token (all other tokens get multiplied by zero by the first term in Eq. 7.5).

So without loss of generality we can say that at time t the cross-entropy loss in Eq. 7.5 can be simplified as the negative log probability the model assigns to the next word in the training sequence, $-\log p(w_{t+1})$, or more formally, using $\hat{\mathbf{y}}$ to mean the vector of estimated token probabilities from the language model:

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}] \tag{7.6}$$

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 tokens 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 get the model 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 guess from the previous time step) is called **teacher forcing**.

teacher forcing

Fig. 7.13 illustrates the general training approach. At each step, given all the preceding tokens, the language model 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 each batch is the average cross-entropy loss over the entire sequence of negative log probabilities, or more formally:

$$L_{CE}(\text{batch of length T}) = \frac{1}{T}\sum_{t=1}^{T} -\log \hat{\mathbf{y}}_t[w_{t+1}] \tag{7.7}$$

The weights in the network are then adjusted to minimize this average cross-entropy loss over the batch via gradient descent (Fig. 4.5), using error backpropagation on the computation graph to compute the gradient. Training adjusts all the weights of the network. For the transformer model we will introduce in the next chapter, these weights include the embedding matrix $\mathbf{E}$ that contains the embeddings for each word. Thus embeddings will be learned that are most successful at predicting upcoming words.

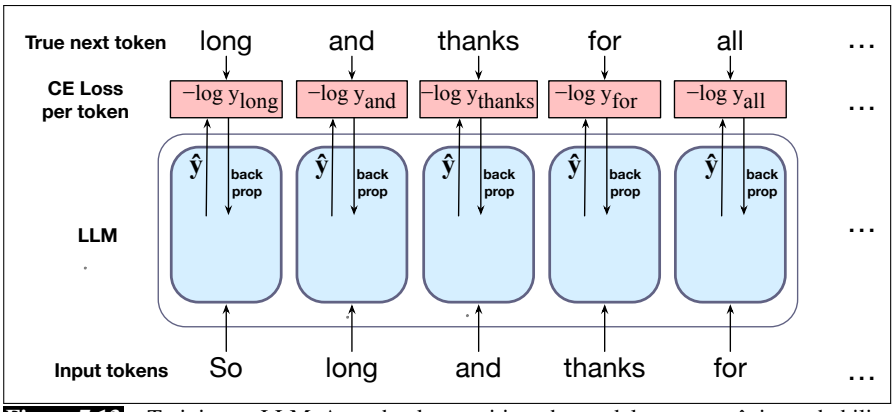


Figure 7.13 Training an LLM. At each token position, the model passes up $\hat{\mathbf{y}}$, its probability estimate for all possible next words. The negative log of the model's probability estimate for the correct token is used as the loss, which is then backpropagated through the model to train all the weights, including the embeddings. Losses are averaged over all the tokens in a batch.

More details of training of course depend on the specific network architecture used to implement the model; we'll see more details specifically for the transformer model in the next chapter.

7.5.2 Pretraining 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. 7.14 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).

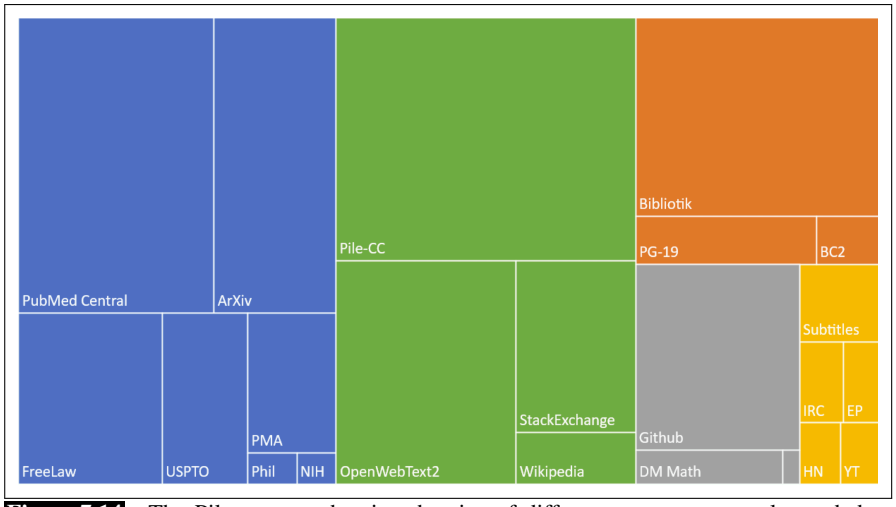


Figure 7.14 The Pile corpus, showing the size of different components, color coded as academic (articles from PubMed and ArXiv, patents from the USPTA; internet (webtext including a subset of the common crawl as well as Wikipedia), prose (a large corpus of books), dialogue (including movie subtitles 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,

PII

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).

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 email addresses. While filters are used to try to remove websites likely to contain large amounts of personal information, such filtering isn't sufficient. We'll return to the privacy question in Section 7.7.

Skew: Training data is also disproportionately generated by authors from the US and from developed countries, which likely skews the resulting generation toward the perspectives or topics of this group alone.

7.5.3 Finetuning

Although the vast pretraining data for large language models includes text from many domains, we might want to apply it in a new domain or task that didn't appear sufficiently in the pretraining 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.

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 using the cross-entropy loss on some new data is called **finetuning**. The word "finetuning" means the process of taking a pretrained model and further adapting some or all of its parameters to some new data. Over the next few chapters we'll see a number of different ways that the word 'finetuning' is used, based on exactly which parameters get updated. The method we describe here, in which we just continue to train, as if the new data was at the end of our pretraining data, can also be called **continued pretraining**. Fig. 7.15 sketches the paradigm.

finetuning

continued pretraining

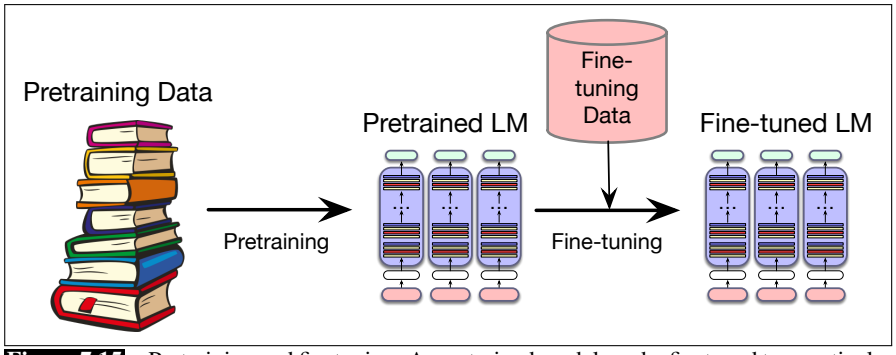


Figure 7.15 Pretraining and finetuning. A pre-trained model can be finetuned to a particular domain or dataset. There are many different ways to finetune, depending on exactly which parameters are updated from the finetuning data: all the parameters, some of the parameters, or only the parameters of specific extra circuitry, as we'll see in future chapters.

7.6 Evaluating Large Language Models

We can evaluate language models by accuracy (how well they predict unseen text, by how well they perform tasks like answering questions or translating text), or by other factors like how fast they can be run, how much energy they use, or how fair they are. We'll explore all of these in the next three sections.

7.6.1 Perplexity

As we first saw in Chapter 3, one way to evaluate language models is to measure how well they predict unseen text. A better language model is better at predicting upcoming words, and so it will be less surprised by (i.e., assign a higher probability to) each word when it occurs in the test set.

If we want to know which of two language models is a better model of some text, we can just see which assigns it a higher probability, or in practice, since we mostly deal with probabilities in log space, we see which assigns a higher log likelihood.

We've been talking about predicting one word at a time, computing the probability of the next token w_i from the prior context: $P(w_i|w_{<i})$. But of course as we saw in Chapter 3 the chain rule allows us to move between computing the probability of the next token and computing the probability of a whole text:

$$\begin{aligned} P(w_{1:n}) &= P(w_1)P(w_2|w_1)P(w_3|w_{1:2})\ldots P(w_n|w_{1:n-1}) \\ &= \prod_{i=1}^{n} P(w_i|w_{<i}) \end{aligned} \tag{7.8}$$

We can compute the probability of text just by multiplying the conditional probabilities for each token in the text. The resulting (log) likelihood of a text is a useful metric for comparing how good two language models are on that text:

$$\log \text{likelihood}(w_{1:n}) = \log \prod_{i=1}^{n} P(w_i|w_{<i}) \tag{7.9}$$

However, we often use another metric other than log likelihood to evaluate language models. The reason is that the probability of a test set (or any sequence) depends on the number of words or tokens in it. In fact, the probability of a test set gets

smaller the longer the text is; this is clear from the chain rule, since if we are multiplying more probabilities, and each probability by definition is less than one, the product will get smaller and smaller. So it's useful to have a metric that is per-token, normalized by length, so we could compare across texts of different lengths.

perplexity

A function of probability called **perplexity** is such a length-normalized metric. Recall from page 46 that the perplexity of a model θ on an unseen test set is the inverse probability that θ assigns to the test set (one over the probability of the test set), normalized by the test set length in tokens. For a test set of n tokens $w_{1:n}$, the perplexity is

$$\begin{aligned} \text{Perplexity}_\theta(w_{1:n}) &= P_\theta(w_{1:n})^{-\frac{1}{n}} \\ &= \sqrt[n]{\frac{1}{P_\theta(w_{1:n})}} \end{aligned} \tag{7.10}$$

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:

$$\text{Perplexity}_\theta(w_{1:n}) = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P_\theta(w_i|w_{<i})}} \tag{7.11}$$

Note that because of the inverse in Eq. 7.10, the higher the probability of the word sequence, the lower the perplexity. Thus **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. Why does perplexity use the inverse probability? The inverse arises from the original definition of perplexity from cross-entropy rate in information theory; for those interested, the explanation is in Section 3.7. Meanwhile, we just have to remember that perplexity has an inverse relationship with probability.

One caveat: because perplexity depends on the number of tokens n in 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.

7.6.2 Downstream tasks: Reasoning and world knowledge

Perplexity measures one kind of accuracy: accuracy at predicting words. But there are other kinds of accuracy. For each of the downstream tasks we want to apply our language model, like question answering, machine translation, or reasoning, we could measure the accuracy at those tasks. We'll have further discussion of these task-specific evaluations in future chapters; machine translation in Chapter 12, information retrieval in Chapter 11, and speech recognition in Chapter 15.

Here we briefly introduce one such metric: a mechanism for measuring accuracy in answering questions, focusing on multiple-choice questions. This dataset is

MMLU

MMLU (Massive Multitask Language Understanding), a commonly-used dataset of 15,908 knowledge and reasoning questions in 57 areas including medicine, mathematics, computer science, law, and others. Accuracy at answering these multiple-choice questions can be a useful proxy for the model's ability to reason, and its factual knowledge.

For example, here is an MMLU question from the microeconomics domain:[2]

MMLU microeconomics example

One of the reasons that the government discourages and regulates monopolies is that
(A) producer surplus is lost and consumer surplus is gained.
(B) monopoly prices ensure productive efficiency but cost society allocative efficiency.
(C) monopoly firms do not engage in significant research and development.
(D) consumer surplus is lost with higher prices and lower levels of output.

Fig. 7.16 shows the way MMLU turns these questions into prompted tests of a language model, in this case showing an example prompt with 2 demonstrations.

MMLU mathematics prompt

The following are multiple choice questions about high school mathematics.
How many numbers are in the list 25, 26, ..., 100?
(A) 75 (B) 76 (C) 22 (D) 23
Answer: B

Compute $i + i^2 + i^3 + \cdots + i^{258} + i^{259}$.
(A) -1 (B) 1 (C) i (D) -i
Answer: A

If 4 daps = 7 yaps, and 5 yaps = 3 baps, how many daps equal 42 baps?
(A) 28 (B) 21 (C) 40 (D) 30
Answer:

Figure 7.16 Sample 2-shot prompt from MMLU testing high-school mathematics. (The correct answer is (C)).

data contamination

Taking performance on MMLU as a metric for language model quality has a problem, though, one that is true of all evaluations based on public datasets. The problem is **data contamination**. Data contamination is when some part of a dataset that we are testing on (a test set of any kind) makes its way into our training set. For example, since large language models train on the web, and MMLU is on the web, models may well incorporate some MMLU questions into their training. If those questions are used for evaluation, the metric will overstate the performance of the language model. One way to mitigate data contamination is to make available the exact training data used to train a model, or at least to report training overlap with specific test sets (Zhang et al., 2025).

7.6.3 Other factors for evaluating language models

Accuracy isn't the only thing we care about in evaluating models (Dodge et al., 2019; Ethayarajh and Jurafsky, 2020, inter alia). For example, we often care about how big a model is, and how long it takes to train or do inference. We often have limited time, or limited memory, since the GPUs we run our models on have fixed memory

[2] For those of you whose economics is a bit rusty, the correct answer is (D).

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 given 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 choose 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.

7.7 Ethical and Safety Issues with Language Models

Humanists have been thinking about the ethical and safety issues inherent to creating artificial agents since well before we had large language models. You have probably read Mary Shelley's 1818 novel *Frankenstein*, but if not, you should. In the book, which she wrote as a teenager, Shelley describes the hubris and ethical blindness of a scientist who creates an artificial person without considering basic ethical principles. The picture below shows Shelley as painted by Richard Rothwell a decade later at age 30.

Large language models can be unsafe in many ways. For example, LLMs 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! A related symptom is that language models can **suggest unsafe actions**, for example directly suggesting that users do dangerous or illegal things like harming themselves or others. If users seek information from language models in safety-critical situations like asking medical advice, or in emergency situations, or when indicating the intentions of self-harm, incorrect advice can be dangerous and even life-threatening. Again, this problem predates large language models. For example (Bickmore et al., 2018) gave partic-

hallucination

ipants medical problems to pose to three pre-LLM commercial dialogue systems (Siri, Alexa, Google Assistant) and asked them to determine an action to take based on the system responses; many of the proposed actions, if actually taken, would have led to harm or death. We'll return to the issue of hallucination and factuality in Chapter 11 where we introduce proposed mitigation methods like **retrieval augmented generation**, and Chapter 10 where we discussed safety tuning and alignment.

sycophantic

Language models are also **sycophantic**, excessively agreeing with or flattering users When a user says something that is factually wrong, language models often agree with them instead of correcting them, an obvious problem for applications in education and health care. Language models can reinforce delusions, and their obsequiousness and flattery can cause users to have distorted views of themselves and the world and increased antisocial behavior (Cheng et al., 2025).

Language models can also harm users by verbally **attacking** them, or creating **representational harms** (Blodgett et al., 2020) for example by generating abusive or harmful stereotypes (Cheng et al., 2023) and negative attitudes (Brown et al., 2020; Sheng et al., 2019) that demean particular groups of people; both abuse and stereotypes can cause psychological harm to users. Gehman et al. (2020) show that even completely non-toxic prompts can lead large language models to output hate speech and abuse their users. Liu et al. (2020) testing how systems responded to pairs of simulated user turns that were identical except for mentioning different genders or race. They found, for example, that simple changes like using the word 'she' instead of 'he' in a sentence caused systems to respond more offensively and with more negative sentiment. Hofmann et al. (2024) found that LLMs were likely to discriminate against people just because they used particular dialects like African-American English. Again, these problems predate large language models. Microsoft's 2016

Tay

Tay chatbot, for example, was taken offline 16 hours after it went live, when it began posting messages with racial slurs, conspiracy theories, and personal attacks on its users. Tay had learned these biases and actions from its training data, including from users who seemed to be purposely teaching the system to repeat this kind of language (Neff and Nagy 2016).

Another important ethical and safety issue is **privacy**. Privacy has been a concern from the very beginning of computing when Weizenbaum designed the chatbot ELIZA as an experiment in computational therapy (Weizenbaum, 1966). First, people became deeply emotionally involved and conducted very personal conversations with the ELIZA chatbot, even to the extent of asking Weizenbaum to leave the room while they were typing. When Weizenbaum suggested that he might want to store the ELIZA conversations, people immediately pointed out that this would violate people's privacy.

Users are likely to give quite personal information to large language models as well, and indeed the most common current LLM use case is for personal advice and support (Zao-Sanders, 2025). And the more human-like a system, the more users are likely to disclose private information, and yet less likely to worry about the harm of this disclosure (Ischen et al., 2019). We discussed above that pretraining data also is likely 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.

A related safety issue is **emotional dependence**. Reeves and Nass (1996) show

that people tend to assign human characteristics to computers and interact with them in ways that are typical of human-human interactions. They interpret an utterance in the way they would if it had spoken by a human, (even though they are aware they are talking to a computer). Thus LLMs have had significant influences on people's cognitive and emotional state, leading to problems like emotional dependence on LLMs. These issues (emotional engagement and privacy) mean we need to think carefully about the impact of LLMs on the people who are interacting with them.

In addition to their ability to harm their users in these ways, LLMs may carry out additional harmful activities themselves, especially as agent-based paradigms makes it possible for language models to directly interact with the world.

Language models can also be used by malicious actors for generating text for **fraud**, phishing, propaganda, disinformation campaigns, 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.

And of course we already saw in Section 7.5.2 that many issues with LLM stem from using pretraining corpora scraped from the web, including harms of data consent, potential copyright violation, as well as biases in the training data that can be **amplified** by language models, just as we saw for embedding models in Chapter 5.

amplified

Finding ways to mitigate all these ethical safety issues is an important current research area in NLP. One important step is to carefully analyze the data used to pretrain large language models as a way of understanding safety issues of toxicity, discrimination, privacy, and fair use, making it extremely important that language models include **datasheets** (page 18) or **model cards** (page 90) giving full replicable information on the corpora used to train them. Open-source models can specify their exact training data. There are active areas of research in mitigating problems of abuse and toxicity, like detecting and responding appropriately to toxic contexts (Wolf et al. 2017, Dinan et al. 2020, Xu et al. 2020).

Value sensitive design—carefully considering possible harms in advance (Friedman et al. 2017, Friedman and Hendry 2019)— is also important; (Dinan et al., 2021) give a number of suggestions for best practices in system design. For example getting informed consent from participants, whether they are used for training, or whether they are interacting with a deployed LLM is important. Because studying these interactional properties of LLMs involves human participants, researchers also work on these issues with the Institutional Review Boards (**IRB**) at their institutions, who help protect the safety of experimental participants.

IRB

7.8 Summary

This chapter has introduced the large language model. Here's a summary of the main points that we covered:

- A **large language model** is a system that can predict the next word for previous words given a context or prefix of words, and use this prediction to **conditionally generate** text.
- There are three major architectures for language models: the **encoder**, the **decoder**, and the **encoder-decoder**. The well-known large language models used for generating text are all decoder models; we'll describe encoders in Chapter 9 and encoder-decoders in Chapter 12.

- Many NLP tasks—such as question answering and sentiment analysis— can be cast as tasks of word prediction and addressed with large language models.
- We instruct language models via a **prompt**, a text string that a user issues to a language model to get the model to do something useful by iteratively generating tokens conditioned on the prompt.
- The process of finding effective prompts for a task is known as **prompt engineering**.
- The choice of which word to generate in large language models is done by **sampling** from the distribution of possible next words.
- A common sampling approach is **temperature** sampling, which lies in between **greedy decoding** (always generate the most probable word) and **random sampling** (generate a random word according to its probability).
- Temperature sampling increases the probabilities of the high-probability words, decreases the probability of the low-probability words, and then samples from this new distribution.
- Large language models are pretrained to predict words on datasets of 100s of billions of words generally scraped from the web.
- These datasets need to be filtered for quality.
- The pretraining algorithm relies on cross-entropy loss: minimizing the negative log probability of the true next word.
- Language models are evaluated by **perplexity**, by evaluations of accuracy on proxies for downstream tasks, like the **MMLU** question-answering dataset, and via metrics for other factors like fairness and energy use.
- Language models have numerous ethical and safety issues including hallucinations, unsafe instructions, bias, stereotypes, misinformation and propaganda, and violations of privacy and copyright.

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 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 40 years, across a wide variety of NLP tasks like speech recognition and machine translation, 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 large 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 5 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. It was Landauer et al. (1997) who 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 12 and Chapter 9). 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.

foundation model

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

CHAPTER

8 Transformers

"The true art of memory is the art of attention"
Samuel Johnson, *Idler #74*, September 1759

In this chapter we introduce the **transformer**, the standard architecture for building large language models. As we discussed in the prior chapter, transformer-based large language models have completely changed the field of speech and language processing. Indeed, every subsequent chapter in this textbook will make use of them. As with the previous chapter, we'll focus for this chapter on the use of transformers to model left-to-right (sometimes called **causal** or autoregressive) language modeling, in which we are given a sequence of input tokens and predict output tokens one by one by conditioning on the prior context.

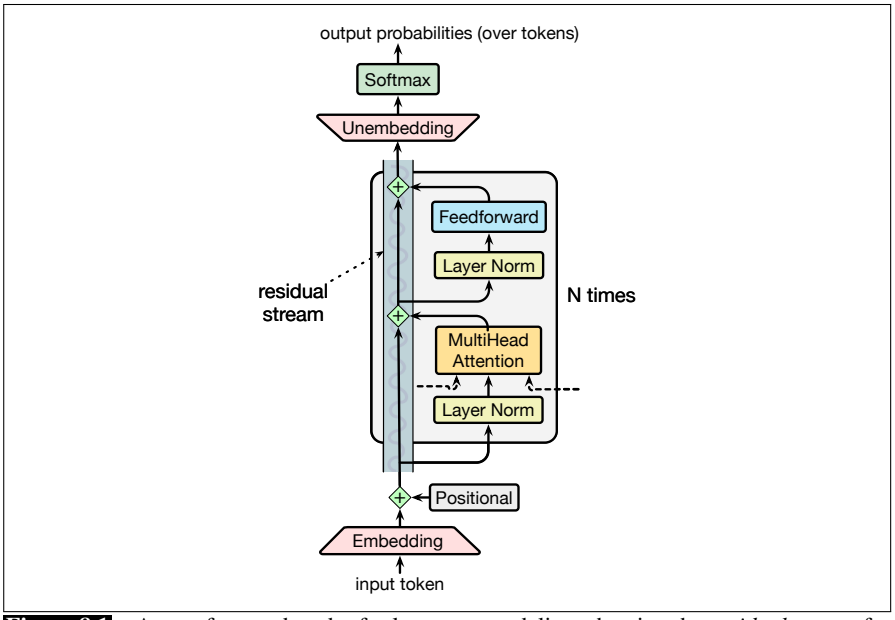


Figure 8.1 A transformer decoder for language modeling, showing the *residual stream* for processing an input token. A single token is embedded and passed forward in the network, with the feedforward and attention components adding information. The multihead attention layer takes inputs (not shown in detail) from the neighboring token streams. This is thus one column of an autoregressive transformer language model, taking an input token and outputting a distribution over next tokens.

Fig. 8.1 sketches the transformer architecture following a single token as it is passes up through the layers of the network. Each token is first converted to an embedding from the **embedding matrix E**. Recall from Chapter 6 in Section 6.5 that **E** is a linear layer that maps a token id to a vector **embedding** representing that token. Each token in the vocabulary has an initial embedding representation in **E**.

Transformers also have a special mechanism for encoding the position/index of the token in the input string, which is simply added to the embedding. The resulting embedding represents both the word and its position. and is then passed through a set of N transformer blocks.

It's common to think of each of these transformer blocks as part of a **stream** in which the input embedding is directly passed up to the output, while simultaneously being enriched by the application of various processing modules: the **multi-head attention** layer, feedforward networks and the layer normalization. The value of the stream at any layer is the sum of the original embedding and all the outputs from all the previous layers and blocks.

The core intuition of the transformer, and the component that distinguishes it from the feedforward layers we saw in Chapter 6, is this multi-head attention layer, also called a **self-attention** layer. Attention can be thought of as a way to build contextual representations of a token's meaning by **attending to** and integrating information from surrounding tokens, helping the model learn how tokens relate to each other over large spans. It can also be thought of as a way to move information from one residual stream to another, augmenting the stream at one token position with information from another token position.

After the N transformer blocks we take the output embedding that is produced by the final transformer block, pass it through an linear **unembedding matrix U** and then a softmax over the vocabulary to generate a distribution over possible next tokens. These last two components (the unembedding matrix and the softmax) are sometimes called the **language modeling head**. In the rest of this chatper we'll introduce attention and the rest of these modules in more detail.

Fig. **??** shows the transformer architecture applied to a context window with the words `So long and thanks for`, showing at each token position what is the most likely token to be generated. In this full figure, the set of N blocks maps an entire **context window** of input vectors $(\mathbf{x}_1, ..., \mathbf{x}_n)$ to a window of output vectors $(\mathbf{h}_1, ..., \mathbf{h}_n)$ of the same length. A column might contain from 12 to 96 or more stacked blocks. The arrows in the figure shows how information from the hidden representations of preceding tokens is incorporated into the transformer block.

Transformer-based language models are complex, and so the details will unfold over this chapter and the next few chapters. Chapter 7 already discussed how language models are **pretrained**, and how tokens are generated via **sampling**. In the rest of this chapter we'll introduce multi-head attention, the rest of the transformer block, and the input encoding and language modeling head components of the transformer. Chapter 9 introduces **masked language modeling** and the **BERT** family of bidirectional transformer encoder models. Chapter 10 shows how to **instruction-tune** language models to perform NLP tasks, and how to **align** the model with human preferences. Chapter 12 will introduce machine translation with the **encoder-decoder** architecture. And we'll see application of the transformer to speech recognition, as well as further use of the encoder-decoder architecture, in Chapter 15.

8.1 Attention

Recall from Chapter 5 that for **word2vec** and other static embeddings, the representation of a word's meaning is always the same vector irrespective of the context: the word `chicken`, for example, is always represented by the same fixed vector. So a static vector for the word `it` might somehow encode that this is a pronoun used

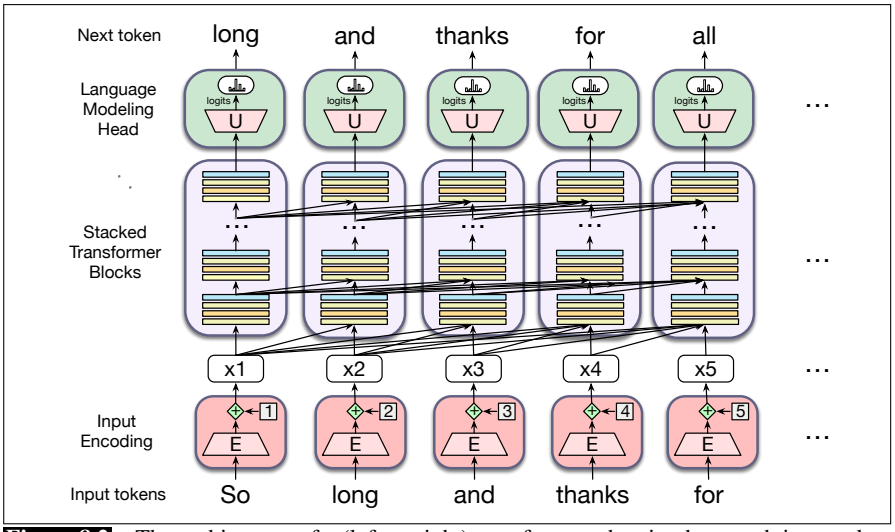


Figure 8.2 The architecture of a (left-to-right) transformer, showing how each input token get encoded, passed through a set of stacked transformer blocks, and then a language model head that predicts the next token. The embeddings at each token position in the residual stream are passed up the stack, and the arrows in the figure shows how information from the hidden representations of preceding tokens are also incorporated.

for animals and inanimate entities. But in context `it` has a much richer meaning. Consider `it` in one of these two sentences:

(8.1) The **chicken** didn't cross the road because **it** was too tired.

(8.2) The chicken didn't cross the **road** because **it** was too wide.

In (8.1) `it` is the chicken (i.e., the reader knows that the chicken was tired), while in (8.2) `it` is the road (and the reader knows that the road was wide).[1] That is, if we are to compute the meaning of this sentence, we'll need the meaning of `it` to be associated with `the chicken` in the first sentence and associated with `the road` in the second one, sensitive to the context.

Furthermore, consider reading left to right like a causal language model, processing the sentence up to the word `it`:

(8.3) The **chicken** didn't cross the **road** because **it**

At this point we don't yet know which thing `it` is going to end up referring to! So a representation of `it` at this point might have aspects of both `chicken` and `road` as the reader is trying to guess what happens next.

This fact that words have rich linguistic relationships with other words that may be far away pervades language. Consider two more examples:

(8.4) The **keys** to the cabinet **are** on the table.

(8.5) I walked along the **pond**, and noticed one of the trees along the **bank**.

In (8.4), the phrase *The keys* is the subject of the sentence, and in English and many languages, must agree in grammatical number with the verb *are*; in this case both are plural. In English we can't use a singular verb like *is* with a plural subject like *keys* (we'll discuss agreement more in Chapter 18). In (8.5), we know that *bank* refers to the side of a pond or river and not a financial institution because of the context, including words like *pond*. (We'll discuss word senses more in Chapter 9.)

[1] We say that in the first example `it` **corefers** with the chicken, and in the second `it` corefers with the road; we'll return to this in Chapter 23.

The point of all these examples is that these contextual words that help us compute the meaning of words in context can be quite far away in the sentence or paragraph. Transformers can build contextual representations of word meaning, **contextual embeddings**, by integrating the meaning of these helpful contextual words. In a transformer, layer by layer, we build up richer and richer contextualized representations of the meanings of input tokens. At each layer, we compute the representation of a token i by combining information about i from the previous layer with information about the neighboring tokens to produce a contextualized representation for each word at each position.

contextual embeddings

Attention is the mechanism in the transformer that weighs and combines the representations from appropriate other tokens in the context from layer k to build the representation for tokens in layer $k+1$.

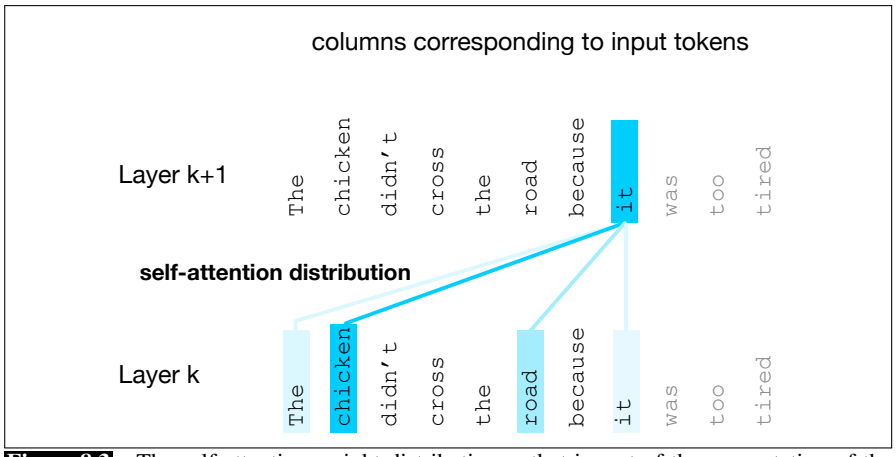


Figure 8.3 The self-attention weight distribution α that is part of the computation of the representation for the word *it* at layer $k+1$. In computing the representation for *it*, we attend differently to the various words at layer k, with darker shades indicating higher self-attention values. Note that the transformer is attending highly to the columns corresponding to the tokens *chicken* and *road* , a sensible result, since at the point where *it* occurs, it could plausibly corefer with the chicken or the road, and hence we'd like the representation for *it* to draw on the representation for these earlier words. Figure adapted from Uszkoreit (2017).

Fig. 8.3 shows a schematic example simplified from a transformer (Uszkoreit, 2017). The figure describes the situation when the current token is `it` and we need to compute a contextual representation for this token at layer $k+1$ of the transformer, drawing on the representations (from layer k) of every prior token. The figure uses color to represent the attention distribution over the contextual words: the tokens `chicken` and `road` both have a high attention weight, meaning that as we are computing the representation for `it`, we will draw most heavily on the representation for `chicken` and `road`. This will be useful in building the final representation for `it`, since `it` will end up coreferring with either `chicken` or `road`.

Let's now turn to how this attention distribution is represented and computed.

8.1.1 Attention more formally

As we've said, the attention computation is a way to compute a vector representation for a token at a particular layer of a transformer, by selectively attending to and integrating information from prior tokens at the previous layer. Attention takes an

input representation $\mathbf{x}_i$ corresponding to the input token at position i, and a context window of prior inputs $\mathbf{x}_1..\mathbf{x}_{i-1}$, and produces an output $\mathbf{a}_i$.

In causal, left-to-right language models, the context is any of the prior words. That is, when processing $\mathbf{x}_i$, the model has access to $\mathbf{x}_i$ as well as the representations of all the prior tokens in the context window (context windows consist of thousands of tokens) but no tokens after i. (By contrast, in Chapter 9 we'll generalize attention so it can also look ahead to future words.)

Fig. 8.4 illustrates this flow of information in an entire causal self-attention layer, in which this same attention computation happens in parallel at each token position i. Thus a self-attention layer maps input sequences $(\mathbf{x}_1,...,\mathbf{x}_n)$ to output sequences of the same length $(\mathbf{a}_1,...,\mathbf{a}_n)$.

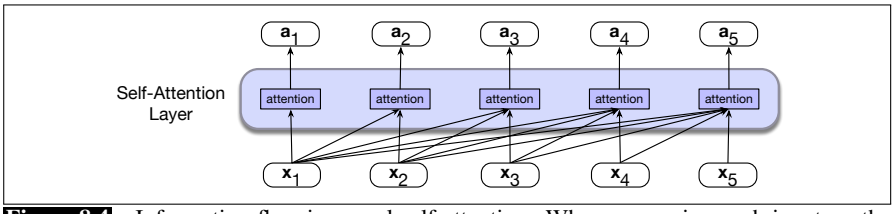


Figure 8.4 Information flow in causal self-attention. When processing each input $\mathbf{x}_i$, the model attends to all the inputs up to, and including $\mathbf{x}_i$.

Simplified version of attention At its heart, attention is really just a weighted sum of context vectors, with a lot of complications added to how the weights are computed and what gets summed. For pedagogical purposes let's first describe a simplified intuition of attention, in which the attention output $\mathbf{a}_i$ at token position i is simply the weighted sum of all the representations $\mathbf{x}_j$, for all $j \leq i$; we'll use α_{ij} to mean how much $\mathbf{x}_j$ should contribute to $\mathbf{a}_i$:

$$\textit{Simplified version:} \quad \mathbf{a}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j \tag{8.6}$$

Each α_{ij} is a scalar used for weighing the value of input $\mathbf{x}_j$ when summing up the inputs to compute $\mathbf{a}_i$. How shall we compute this α weighting? In attention we weight each prior embedding proportionally to how **similar** it is to the current token i. So the output of attention is a sum of the embeddings of prior tokens weighted by their similarity with the current token embedding. We compute similarity scores via **dot product**, which maps two vectors into a scalar value ranging from $-\infty$ to ∞. The larger the score, the more similar the vectors that are being compared. We'll normalize these scores with a softmax to create the vector of weights $\alpha_{ij}, j \leq i$.

$$\textit{Simplified Version:} \quad \text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j \tag{8.7}$$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \ \forall j \leq i \tag{8.8}$$

Thus in Fig. 8.4 we compute $\mathbf{a}_3$ by computing three scores: $\mathbf{x}_3 \cdot \mathbf{x}_1$, $\mathbf{x}_3 \cdot \mathbf{x}_2$ and $\mathbf{x}_3 \cdot \mathbf{x}_3$, normalizing them by a softmax, and using the resulting probabilities as weights indicating each of their proportional relevance to the current position i. Of course, the softmax weight will likely be highest for $\mathbf{x}_i$, since $\mathbf{x}_i$ is very similar to itself, resulting in a high dot product. But other context words may also be similar to i, and the softmax will also assign some weight to those words. Then we use these weights as the α values in Eq. 8.6 to compute the weighted sum that is our $\mathbf{a}_3$.

The simplified attention in equations 8.6 – 8.8 demonstrates the attention-based approach to computing $\mathbf{a}_i$: compare the $\mathbf{x}_i$ to prior vectors, normalize those scores

into a probability distribution used to weight the sum of the prior vectors. But now we're ready to remove the simplifications.

A single attention head using query, key, and value matrices Now that we've seen a simple intuition of attention, let's introduce the actual **attention head**, the version of attention that's used in transformers. (The word **head** is often used in transformers to refer to specific structured layers). The attention head allows us to distinctly represent three different roles that each input embedding plays during the course of the attention process:

attention head
head

- As *the current element* being compared to the preceding inputs. We'll refer to this role as a **query**.
- In its role as *a preceding input* that is being compared to the current element to determine a similarity weight. We'll refer to this role as a **key**.
- And finally, as a **value** of a preceding element that gets weighted and summed up to compute the output for the current element.

query
key
value

To capture these three different roles, transformers introduce weight matrices $\mathbf{W^Q}$, $\mathbf{W^K}$, and $\mathbf{W^V}$. These weights will project each input vector $\mathbf{x}_i$ into a representation of its role as a query, key, or value:

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W^Q}; \quad \mathbf{k}_i = \mathbf{x}_i\mathbf{W^K}; \quad \mathbf{v}_i = \mathbf{x}_i\mathbf{W^V} \tag{8.9}$$

Given these projections, when we are computing the similarity of the current element $\mathbf{x}_i$ with some prior element $\mathbf{x}_j$, we'll use the dot product between the current element's **query** vector $\mathbf{q}_i$ and the preceding element's **key** vector $\mathbf{k}_j$. Furthermore, the result of a dot product can be an arbitrarily large (positive or negative) value, and exponentiating large values can lead to numerical issues and loss of gradients during training. To avoid this, we scale the dot product by a factor related to the size of the embeddings, via dividing by the square root of the dimensionality of the query and key vectors (d_k). We thus replace the simplified Eq. 8.7 with Eq. 8.11. The ensuing softmax calculation resulting in α_{ij} remains the same, but the output calculation for $\mathbf{head}_i$ is now based on a weighted sum over the value vectors $\mathbf{v}$ (Eq. 8.13).

Here's a final set of equations for computing self-attention for a single self-attention output vector $\mathbf{a}_i$ from a single input vector $\mathbf{x}_i$. This version of attention computes $\mathbf{a}_i$ by summing the *values* of the prior elements, each weighted by the similarity of its *key* to the *query* from the current element:

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W^Q}; \quad \mathbf{k}_j = \mathbf{x}_j\mathbf{W^K}; \quad \mathbf{v}_j = \mathbf{x}_j\mathbf{W^V} \tag{8.10}$$

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \tag{8.11}$$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \;\; \forall j \leq i \tag{8.12}$$

$$\mathbf{head}_i = \sum_{j \leq i} \alpha_{ij}\mathbf{v}_j \tag{8.13}$$

$$\mathbf{a}_i = \mathbf{head}_i\mathbf{W^O} \tag{8.14}$$

We illustrate this in Fig. 8.5 for the case of calculating the value of the third output $\mathbf{a}_3$ in a sequence.

Note that we've also introduced one more matrix, $\mathbf{W_O}$, which is left-multiplied by the attention head. This is necessary to reshape the output of the head. The input to attention $\mathbf{x_i}$ and the output from attention $\mathbf{a_i}$ both have the same dimensionality $[1 \times d]$. We often call d the **model dimensionality**, and indeed as we'll discuss in

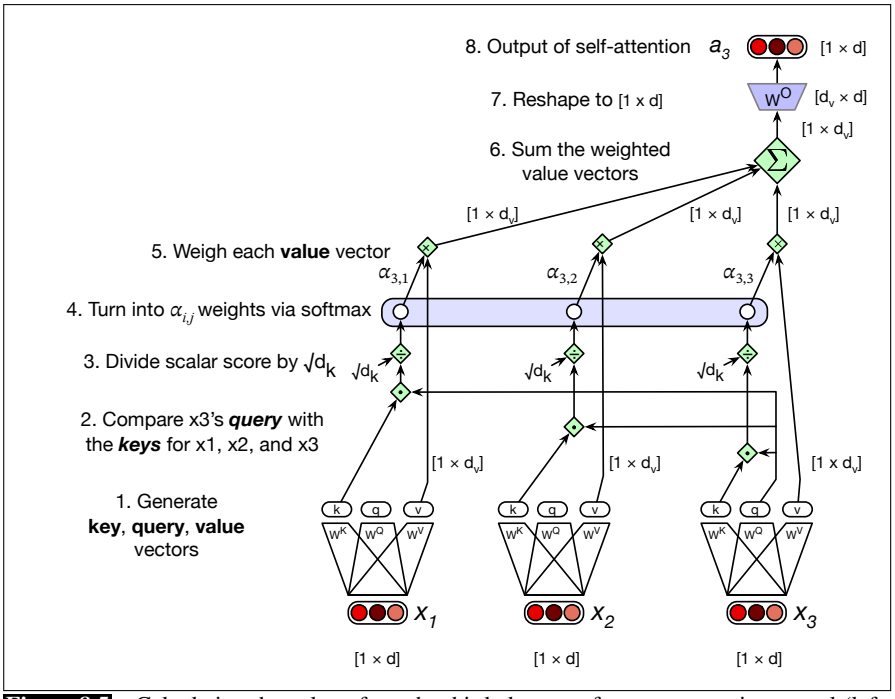


Figure 8.5 Calculating the value of $\mathbf{a}_3$, the third element of a sequence using causal (left-to-right) self-attention.

Section 8.2 the output $\mathbf{h_i}$ of each transformer block, as well as the intermediate vectors inside the transformer block also have the same dimensionality $[1 \times d]$. Having everything be the same dimensionality makes the transformer very modular.

So let's talk shapes. How do we get from $[1 \times d]$ at the input to $[1 \times d]$ at the output? Let's look at all the internal shapes. We'll have a dimension d_k for the query and key vectors. The query vector and the key vector are both dimensionality $[1 \times d_k]$, so we can take their dot product $\mathbf{q}_i \cdot \mathbf{k}_j$ to produce a scalar. We'll have a separate dimension d_v for the value vectors. The transform matrix $\mathbf{W^Q}$ has shape $[d \times d_k]$, $\mathbf{W^K}$ is $[d \times d_k]$, and $\mathbf{W^V}$ is $[d \times d_v]$. So the output of $\mathbf{head}_i$ in equation Eq. 8.13 is of shape $[1 \times d_v]$. To get the desired output shape $[1 \times d]$ we'll need to reshape the head output, and so $\mathbf{W^O}$ is of shape $[d_v \times d]$. In the original transformer work (Vaswani et al., 2017), d was 512, d_k and d_v were both 64.

Multi-head Attention Equations 8.11-8.13 describe a single **attention head**. But actually, transformers use multiple attention heads. The intuition is that each head might be attending to the context for different purposes: heads might be specialized to represent different linguistic relationships between context elements and the current token, or to look for particular kinds of patterns in the context.

multi-head attention

So in **multi-head attention** we have A separate attention heads that reside in parallel layers at the same depth in a model, each with its own set of parameters that allows the head to model different aspects of the relationships among inputs. Thus each head i in a self-attention layer has its own set of query, key, and value matrices: $\mathbf{W^{Qi}}$, $\mathbf{W^{Ki}}$, and $\mathbf{W^{Vi}}$. These are used to project the inputs into separate query, key, and value embeddings for each head.

When using multiple heads the model dimension d is still used for the input and output, the query and key embeddings have dimensionality d_k, and the value embeddings are of dimensionality d_v (again, in the original transformer paper $d_k =$

$d_v = 64$, $A = 8$, and $d = 512$). Thus for each head i, we have weight layers $\mathbf{W^{Qi}}$ of shape $[d \times d_k]$, $\mathbf{W^{Ki}}$ of shape $[d \times d_k]$, and $\mathbf{W^{Vi}}$ of shape $[d \times d_v]$.

Below are the equations for attention augmented with multiple heads; Fig. 8.6 shows an intuition.

$$\mathbf{q}_i^c = \mathbf{x}_i\mathbf{W^{Qc}};\quad \mathbf{k}_j^c = \mathbf{x}_j\mathbf{W^{Kc}};\quad \mathbf{v}_j^c = \mathbf{x}_j\mathbf{W^{Vc}};\quad \forall\, c\quad 1 \le c \le A \tag{8.15}$$

$$\text{score}^c(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i^c \cdot \mathbf{k}_j^c}{\sqrt{d_k}} \tag{8.16}$$

$$\alpha_{ij}^c = \text{softmax}(\text{score}^c(\mathbf{x}_i, \mathbf{x}_j))\ \ \forall j \le i \tag{8.17}$$

$$\mathbf{head}_i^c = \sum_{j \le i} \alpha_{ij}^c \mathbf{v}_j^c \tag{8.18}$$

$$\mathbf{a}_i = (\mathbf{head}^1 \oplus \mathbf{head}^2 \ldots \oplus \mathbf{head}^A)\mathbf{W}^O \tag{8.19}$$

$$\text{MultiHeadAttention}(\mathbf{x}_i, [\mathbf{x}_1, \cdots, \mathbf{x}_{i-1}]) = \mathbf{a}_i \tag{8.20}$$

Note in Eq. 8.20 that MultiHeadAttention is a function of the current input $\mathbf{x}_i$, as well as all the other inputs. For the causal or left-to-right attention that we use in this chapter, the other inputs are only to the left, but we'll also see a version of attention in Chapter 9 where attention is a function of the tokens to the right as well. We'll return to this idea about causal inputs in Eq. 8.34 when we introduce the idea of masking the right context.

The output of each of the A heads is of shape $[1 \times d_v]$, and so the output of the multi-head layer with A heads consists of A vectors of shape $[1 \times d_v]$. These are concatenated to produce a single output with dimensionality $[1 \times Ad_v]$. Then we use yet another linear projection $\mathbf{W^O} \in \mathbb{R}^{Ad_v \times d}$ to reshape it, resulting in the multi-head attention vector $\mathbf{a}_i$ with the correct output shape $[1 \times d]$ at each input i.

8.2 Transformer Blocks

The self-attention calculation lies at the core of what's called a transformer block, which, in addition to the self-attention layer, includes three other kinds of layers: (1) a feedforward layer, (2) residual connections, and (3) normalizing layers (colloquially called "layer norm").

residual stream

Fig. 8.7 illustrates a transformer block, sketching a common way of thinking about the block that is called the **residual stream** (Elhage et al., 2021). In the residual stream viewpoint, we consider the processing of an individual token i through the transformer block as a single stream of d-dimensional representations for token position i. This residual stream starts with the original input vector, and the various components read their input from the residual stream and add their output back into the stream.

The input at the bottom of the stream is an embedding for a token, which has dimensionality d. This initial embedding gets passed up (by **residual connections**), and is progressively added to by the other components of the transformer: the **attention layer** that we have seen, and the **feedforward layer** that we will introduce. Before the attention and feedforward layer is a computation called the **layer norm**.

Thus the initial vector is passed through a layer norm and attention layer, and the result is added back into the stream, in this case to the original input vector $\mathbf{x}_i$. And then this summed vector is again passed through another layer norm and a

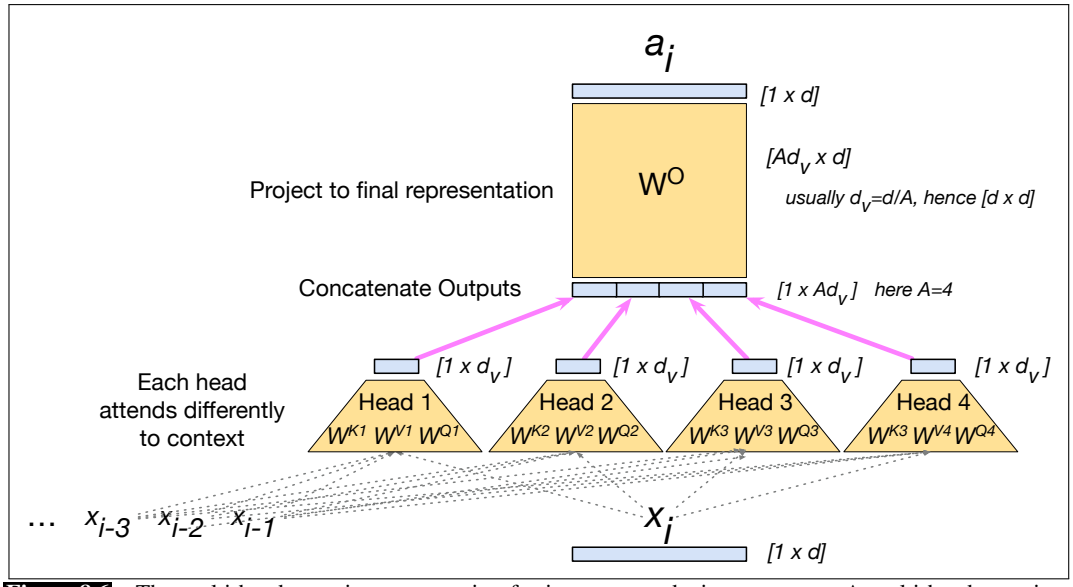


Figure 8.6 The multi-head attention computation for input $\mathbf{x}_i$, producing output $\mathbf{a}_i$. A multi-head attention layer has A heads, each with its own query, key, and value weight matrices. In this figure, we show $A = 4$, a smaller value than is usually used, just to fit on the page. The outputs from each of the heads are of shape $[1 \times d_v]$ and are concatenated and then projected into a different space by the $\mathbf{W_O}$ matrix. Usually the dimensionality d_v of the heads is set so that $d_v = d/A$, with the result that $\mathbf{W_O}$ is a square matrix of shape $[Ad_v \times d] = [d \times d]$. usually of the same size. then projected d, thus producing an output of the same size as the input.

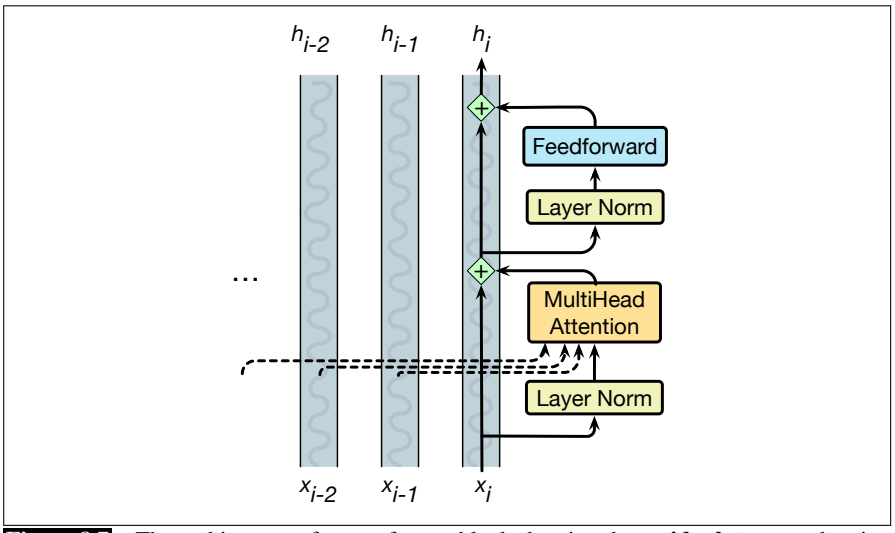


Figure 8.7 The architecture of a transformer block showing the **residual stream**, showing how most information flows up through the residual stream, and only the attention module is sensitive to information from other streams at prior token positions. In this figure and throughout the chapter, we use the **prenorm** version of the architecture, in which the layer norms happen before the attention and feedforward layers rather than after. The first

feedforward layer, and the output of those is added back into the residual, and we'll use $\mathbf{h}_i$ to refer to the resulting output of the transformer block for token i.

We've already seen the attention layer, so let's now introduce the feedforward and layer norm computations in the context of processing a single input $\mathbf{x}_i$ at token

position i.

Feedforward layer The feedforward layer is a fully-connected 2-layer network, i.e., one hidden layer, two weight matrices, as introduced in Chapter 6. The weights are the same for each token position i, but are different from layer to layer. It is common to make the dimensionality d_{ff} of the hidden layer of the feedforward network be larger than the model dimensionality d. (For example in the original transformer model, $d = 512$ and $d_{\text{ff}} = 2048$.)

$$\text{FFN}(\mathbf{x}_i) = \text{ReLU}(\mathbf{x}_i\mathbf{W_1} + b_1)\mathbf{W_2} + b_2 \tag{8.21}$$

layer norm

Layer Norm At two stages in the transformer block we **normalize** the vector (Ba et al., 2016). This process, called **layer norm** (short for layer normalization), is one of many forms of normalization that can be used to improve training performance in deep neural networks by keeping the values of a hidden layer in a range that facilitates gradient-based training.

Layer norm is a variation of the **z-score** from statistics, applied to a single vector in a hidden layer. That is, the term layer norm is a bit confusing; layer norm is **not** applied to an entire transformer layer, but just to the embedding vector of a single token. Thus the input to layer norm is a single vector of dimensionality d and the output is that vector normalized, again of dimensionality d. The first step in layer normalization is to calculate the mean, μ, and standard deviation, σ, over the elements of the vector to be normalized. Given an embedding vector $\mathbf{x}$ of dimensionality d, these values are calculated as follows.

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i \tag{8.22}$$

$$\sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2} \tag{8.23}$$

Given these values, the vector components are normalized by subtracting the mean from each and dividing by the standard deviation. The result of this computation is a new vector with zero mean and a standard deviation of one.

$$\hat{\mathbf{x}} = \frac{(\mathbf{x} - \mu)}{\sigma} \tag{8.24}$$

Finally, in the standard implementation of layer normalization, two learnable parameters, γ and β, representing gain and offset values, are introduced.

$$\text{LayerNorm}(\mathbf{x}) = \gamma\frac{(\mathbf{x} - \mu)}{\sigma} + \beta \tag{8.25}$$

Putting it all together The function computed by a transformer block can be expressed by breaking it down with one equation for each component computation, using $\mathbf{t}$ (of shape $[1 \times d]$) to stand for transformer and superscripts to demarcate each computation inside the block:

$$\mathbf{t}_i^1 = \text{LayerNorm}(\mathbf{x}_i) \tag{8.26}$$

$$\mathbf{t}_i^2 = \text{MultiHeadAttention}(\mathbf{t}_i^1, [\mathbf{t}_1^1, \cdots, \mathbf{t}_N^1]) \tag{8.27}$$

$$\mathbf{t}_i^3 = \mathbf{t}_i^2 + \mathbf{x}_i \tag{8.28}$$

$$\mathbf{t}_i^4 = \text{LayerNorm}(\mathbf{t}_i^3) \tag{8.29}$$

$$\mathbf{t}_i^5 = \text{FFN}(\mathbf{t}_i^4) \tag{8.30}$$

$$\mathbf{h}_i = \mathbf{t}_i^5 + \mathbf{t}_i^3 \tag{8.31}$$

Notice that the only component that takes as input information from other tokens (other residual streams) is multi-head attention, which (as we see from Eq. 8.27) looks at all the neighboring tokens in the context. The output from attention, however, is then added into this token's embedding stream. In fact, Elhage et al. (2021) show that we can view attention heads as literally moving information from the residual stream of a neighboring token into the current stream. The high-dimensional embedding space at each position thus contains information about the current token and about neighboring tokens, albeit in different subspaces of the vector space. Fig. 8.8 shows a visualization of this movement. We therefore call the attention function the **token-mixing** component of the architecture, because it mixes information from neighboring token streams into the current stream.

token-mixing

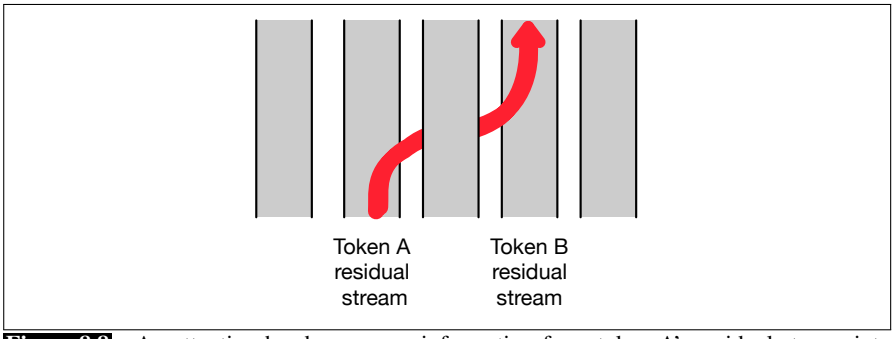


Figure 8.8 An attention head can move information from token A's residual stream into token B's residual stream.

Crucially, the input and output dimensions of transformer blocks are matched so they can be stacked. Each token vector $\mathbf{x}_i$ at the input to the block has dimensionality d, and the output $\mathbf{h}_i$ also has dimensionality d. Transformers for large language models stack many of these blocks, from 12 layers (used for the T5 or GPT-3-small language models) to 96 layers (used for GPT-3 large), to even more for more recent models. We'll come back to this issue of stacking in a bit.

Equation 8.26 and following are just the equation for a single transformer block, but the residual stream metaphor goes through all the transformer layers, from the first transformer blocks to the 12th, in a 12-layer transformer. At the earlier transformer blocks, the residual stream is representing the current token. At the highest transformer blocks, the residual stream is usually representing the following token, since at the very end it's being trained to predict the next token.

Once we stack many blocks, there is one more requirement: at the very end of the last (highest) transformer block, there is a single extra layer norm that is run on the last $\mathbf{h}_i$ of each token stream (just below the language model head layer that we will define soon). [2]

[2] Note that we are using the most common current transformer architecture, which is called the **prenorm** architecture. The original definition of the transformer in Vaswani et al. (2017) used an alternative architecture called the **postnorm** transformer in which the layer norm happens **after** the attention and FFN layers; it turns out moving the layer norm beforehand works better, but does require this one extra layer at the end.

8.3 Parallelizing computation using a single matrix $\mathbf{X}$

This description of multi-head attention and the rest of the transformer block has been from the perspective of computing a single output at a single time step i in a single residual stream. But as we pointed out earlier, the attention computation performed for each token to compute $\mathbf{a}_i$ is independent of the computation for each other token, and that's also true for all the computation in the transformer block computing $\mathbf{h}_i$ from the input $\mathbf{x}_i$. That means we can easily parallelize the entire computation, taking advantage of efficient matrix multiplication routines.

We do this by packing the input embeddings for the N tokens of the input sequence into a single matrix $\mathbf{X}$ of size $[N \times d]$. Each row of $\mathbf{X}$ is the embedding of one token of the input. Transformers for large language models commonly have an input length N from 1K to 32K; much longer contexts of 128K or even up to millions of tokens can also be achieved with architectural changes like special long-context mechanisms that we don't discuss here. So for vanilla transformers, we can think of $\mathbf{X}$ having between 1K and 32K rows, each of the dimensionality of the embedding d (the model dimension).

Parallelizing attention Let's first see this for a single attention head and then turn to multiple heads, and then add in the rest of the components in the transformer block. For one head we multiply $\mathbf{X}$ by the query, key, and value matrices $\mathbf{W}^{\mathbf{Q}}$ of shape $[d \times d_k]$, $\mathbf{W}^{\mathbf{K}}$ of shape $[d \times d_k]$, and $\mathbf{W}^{\mathbf{V}}$ of shape $[d \times d_v]$, to produce matrices $\mathbf{Q}$ of shape $[N \times d_k]$, $\mathbf{K}$ of shape $[N \times d_k]$, and $\mathbf{V}$ of shape $[N \times d_v]$, containing all the key, query, and value vectors:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}}; \quad \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}}; \quad \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}} \tag{8.32}$$

Given these matrices we can compute all the requisite query-key comparisons simultaneously by multiplying $\mathbf{Q}$ and $\mathbf{K}^\intercal$ in a single matrix multiplication. The product is of shape $N \times N$, visualized in Fig. 8.9.

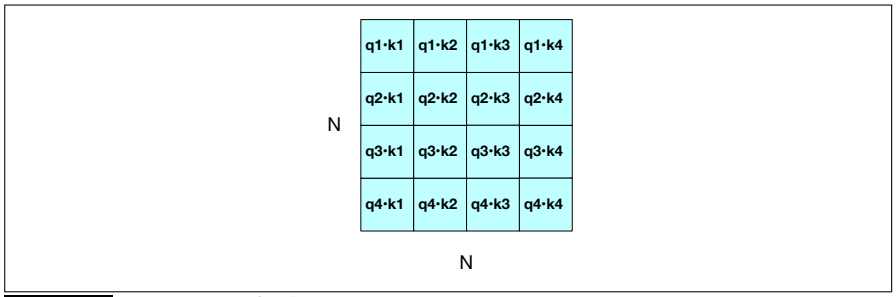


Figure 8.9 The $N \times N$ $\mathbf{QK}^\intercal$ matrix showing how it computes all $q_i \cdot k_j$ comparisons in a single matrix multiple.

Once we have this $\mathbf{QK}^\intercal$ matrix, we can very efficiently scale these scores, take the softmax, and then multiply the result by $\mathbf{V}$ resulting in a matrix of shape $N \times d$: a vector embedding representation for each token in the input. We've reduced the entire self-attention step for an entire sequence of N tokens for one head to the following computation:

$$\mathbf{head} = \text{softmax}\left(\text{mask}\left(\frac{\mathbf{QK}^\intercal}{\sqrt{d_k}}\right)\right)\mathbf{V} \tag{8.33}$$

$$\mathbf{A} = \mathbf{head}\,\mathbf{W}^{\mathbf{O}} \tag{8.34}$$

Masking out the future You may have noticed that we introduced a mask function in Eq. 8.34 above. This is because the self-attention computation as we've described it has a problem: the calculation of $\mathbf{QK}^\intercal$ results in a score for each query value to every key value, *including those that follow the query*. This is inappropriate in the setting of language modeling: guessing the next word is pretty simple if you already know it! To fix this, the elements in the upper-triangular portion of the matrix are set to $-\infty$, which the softmax will turn to zero, thus eliminating any knowledge of words that follow in the sequence. This is done in practice by adding a mask matrix M in which $M_{ij} = -\infty \, \forall j > i$ (i.e. for the upper-triangular portion) and $M_{ij} = 0$ otherwise. Fig. 8.10 shows the resulting masked $\mathbf{QK}^\intercal$ matrix. (we'll see in Chapter 9 how to make use of words in the future for tasks that need it).

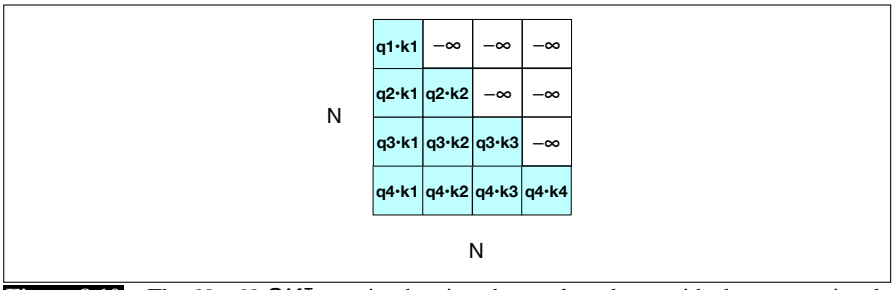


Figure 8.10 The $N \times N$ $\mathbf{QK}^\intercal$ matrix showing the $q_i \cdot k_j$ values, with the upper-triangle portion of the comparisons matrix zeroed out (set to $-\infty$, which the softmax will turn to zero).

Fig. 8.11 shows a schematic of all the computations for a single attention head parallelized in matrix form.

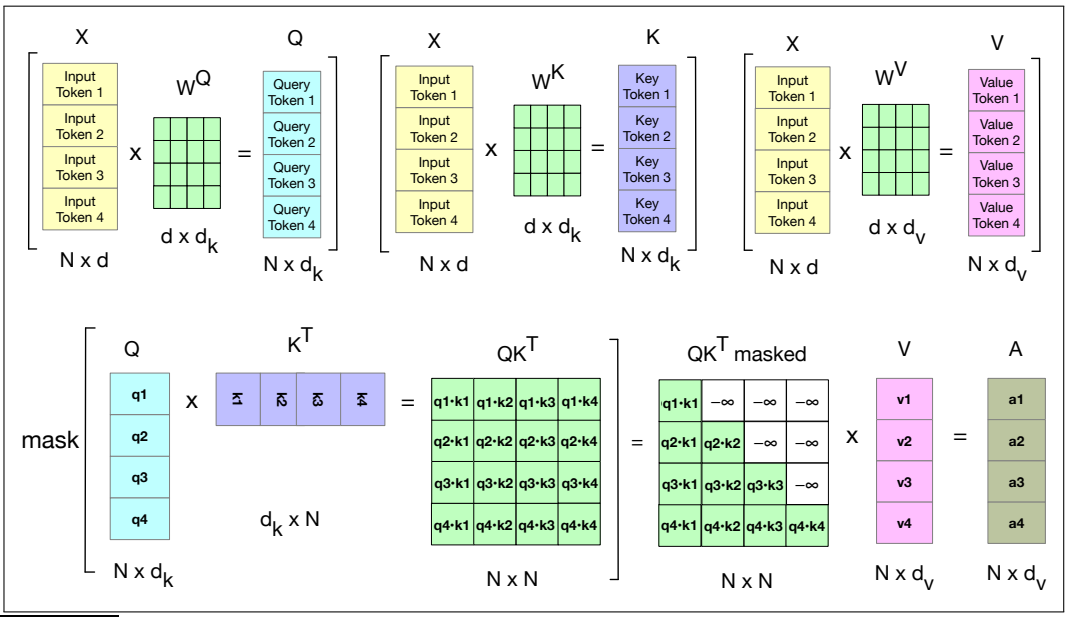


Figure 8.11 Schematic of the attention computation for a single attention head in parallel. The first row shows the computation of the $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ matrices. The second row shows the computation of $\mathbf{QK}^\intercal$, the masking (the softmax computation and the normalizing by dimensionality are not shown) and then the weighted sum of the value vectors to get the final attention vectors.

Fig. 8.9 and Fig. 8.10 also make it clear that attention is quadratic in the length of the input, since at each layer we need to compute dot products between each pair of tokens in the input. This makes it expensive to compute attention over very long documents (like entire novels). Nonetheless modern large language models manage to use quite long contexts of thousands or tens of thousands of tokens.

Parallelizing multi-head attention In multi-head attention, as with self-attention, the input and output have the model dimension d, the key and query embeddings have dimensionality d_k, and the value embeddings are of dimensionality d_v (again, in the original transformer paper $d_k = d_v = 64$, $A = 8$, and $d = 512$). Thus for each head c, we have weight layers $\mathbf{W^Q}_c$ of shape $[d \times d_k]$, $\mathbf{W^K}_c$ of shape $[d \times d_k]$, and $\mathbf{W^V}_c$ of shape $[d \times d_v]$, and these get multiplied by the inputs packed into $\mathbf{X}$ to produce $\mathbf{Q}$ of shape $[N \times d_k]$, $\mathbf{K}$ of shape $[N \times d_k]$, and $\mathbf{V}$ of shape $[N \times d_v]$. The output of each of the A heads is of shape $[N \times d_v]$, and so the output of the multi-head layer with A heads consists of A matrices of shape $[N \times d_v]$. To make use of these matrices in further processing, they are concatenated to produce a single output with dimensionality $[N \times Ad_v]$. Finally, we use a final linear projection $\mathbf{W^O}$ of shape $[Ad_v \times d]$, that reshapes it to the original output dimension for each token. Multiplying the concatenated $[N \times Ad_v]$ matrix output by $\mathbf{W^O}$ of shape $[Ad_v \times d]$ yields the self-attention output $\mathbf{A}$ of shape $[N \times d]$.

$$\mathbf{Q^i} = \mathbf{X}\mathbf{W^{Qi}}; \quad \mathbf{K^i} = \mathbf{X}\mathbf{W^{Ki}}; \quad \mathbf{V^i} = \mathbf{X}\mathbf{W^{Vi}} \tag{8.35}$$

$$\mathbf{head}_i = \text{SelfAttention}(\mathbf{Q^i}, \mathbf{K^i}, \mathbf{V^i}) = \text{softmax}\left(\text{mask}\left(\frac{\mathbf{Q^i}\mathbf{K^{i\intercal}}}{\sqrt{d_k}}\right)\right)\mathbf{V^i} \tag{8.36}$$

$$\text{MultiHeadAttention}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \ldots \oplus \mathbf{head}_A)\mathbf{W^O} \tag{8.37}$$

Putting it all together with the parallel input matrix X The function computed in parallel by an entire layer of N transformer blocks—each block over one of the N input tokens—can be expressed as:

$$\mathbf{O} = \mathbf{X} + \text{MultiHeadAttention}(\text{LayerNorm}(\mathbf{X})) \tag{8.38}$$

$$\mathbf{H} = \mathbf{O} + \text{FFN}(\text{LayerNorm}(\mathbf{O})) \tag{8.39}$$

Note that in Eq. 8.38 we are using $\mathbf{X}$ to mean the input to the layer, wherever it comes from. For the first layer, as we will see in the next section, that input is the initial word + positional embedding vectors that we have been describing by $\mathbf{X}$. But for subsequent layers k, the input is the output from the previous layer $\mathbf{H}^{k-1}$. We can also break down the computation performed in a transformer layer, showing one equation for each component computation. We'll use $\mathbf{T}$ (of shape $[N \times d]$) to stand for transformer and superscripts to demarcate each computation inside the block, and again use $\mathbf{X}$ to mean the input to the block from the previous layer or the initial embedding:

$$\mathbf{T^1} = \text{LayerNorm}(\mathbf{X}) \tag{8.40}$$

$$\mathbf{T^2} = \text{MultiHeadAttention}(\mathbf{T^1}) \tag{8.41}$$

$$\mathbf{T^3} = \mathbf{T^2} + \mathbf{X} \tag{8.42}$$

$$\mathbf{T^4} = \text{LayerNorm}(\mathbf{T^3}) \tag{8.43}$$

$$\mathbf{T^5} = \text{FFN}(\mathbf{T^4}) \tag{8.44}$$

$$\mathbf{H} = \mathbf{T^5} + \mathbf{T^3} \tag{8.45}$$

Here when we use a notation like $\text{FFN}(\mathbf{T^3})$ we mean that the same FFN is applied in parallel to each of the N embedding vectors in the window. Similarly, each of the

N tokens is normed in parallel in the LayerNorm. Crucially, the input and output dimensions of transformer blocks are matched so they can be stacked. Since each token x_i at the input to the block is represented by an embedding of dimensionality $[1 \times d]$, that means the input $\mathbf{X}$ and output $\mathbf{H}$ are both of shape $[N \times d]$.

8.4 The input: embeddings for token and position

embedding

Let's talk about where the input $\mathbf{X}$ comes from. Given a sequence of N tokens (N is the context length in tokens), the matrix $\mathbf{X}$ of shape $[N \times d]$ has an **embedding** for each word in the context. The transformer does this by separately computing two embeddings: an input token embedding, and an input positional embedding.

A token embedding, introduced in Chapter 6, is a vector of dimension d that will be our initial representation for the input token. (As we pass vectors up through the transformer layers in the residual stream, this embedding representation will change and grow, incorporating context and playing a different role depending on the kind of language model we are building.) The set of initial embeddings are stored in the embedding matrix $\mathbf{E}$, which has a row for each of the $|V|$ tokens in the vocabulary. (Reminder that V here means the vocabulary of tokens, this V is not related to the value vector.) Thus each word is a row vector of d dimensions, and $\mathbf{E}$ has shape $[|V| \times d]$.

Given an input token string like *Thanks for all the* we first convert the tokens into vocabulary indices (these were created when we first tokenized the input using BPE or SentencePiece). So the representation of *thanks for all the* might be $\mathbf{w} = [5, 4000, 10532, 2224]$. Next we use indexing to select the corresponding rows from $\mathbf{E}$, (row 5, row 4000, row 10532, row 2224).

one-hot vector

Another way to think about selecting token embeddings from the embedding matrix is to represent tokens as one-hot vectors of shape $[1 \times |V|]$, i.e., with one dimension for each word in the vocabulary. Recall that in a **one-hot vector** all the elements are 0 except one, the element whose dimension is the word's index in the vocabulary, which has value 1. So if the word "thanks" has index 5 in the vocabulary, $x_5 = 1$, and $x_i = 0 \ \forall i \neq 5$, as shown here:

```
[0 0 0 0 1 0 0 ... 0 0 0 0]
 1 2 3 4 5 6 7 ...  ... |V|
```

Multiplying by a one-hot vector that has only one non-zero element $x_i = 1$ simply selects out the relevant row vector for word i, resulting in the embedding for word i, as depicted in Fig. 8.12.

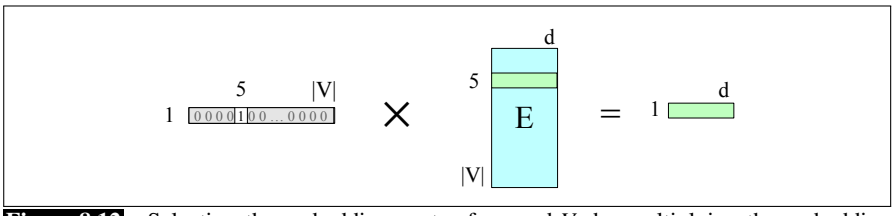


Figure 8.12 Selecting the embedding vector for word V_5 by multiplying the embedding matrix $\mathbf{E}$ with a one-hot vector with a 1 in index 5.

We can extend this idea to represent the entire token sequence as a matrix of one-hot vectors, one for each of the N positions in the transformer's context window, as shown in Fig. 8.13.

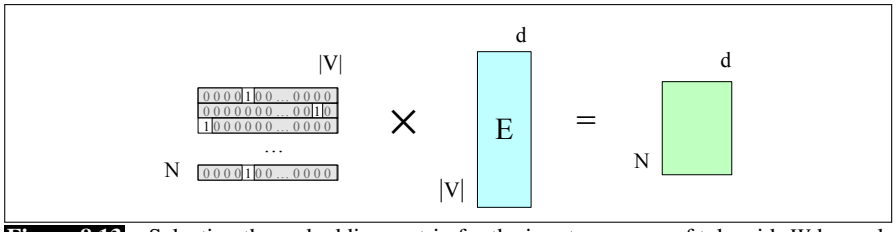


Figure 8.13 Selecting the embedding matrix for the input sequence of token ids W by multiplying a one-hot matrix corresponding to W by the embedding matrix **E**.

These token embeddings are not position-dependent. To represent the position of each token in the sequence, we combine these token embeddings with **positional embeddings** specific to each position in an input sequence.

positional embeddings

absolute position

Where do we get these positional embeddings? The simplest method, called **absolute position**, is to start with randomly initialized embeddings corresponding to each possible input position up to some maximum length. For example, just as we have an embedding for the word *fish*, we'll have an embedding for the position 3. As with word embeddings, these positional embeddings are learned along with other parameters during training. We can store them in a matrix E_{pos} of shape $[N \times d]$.

To produce an input embedding that captures positional information, we just add the word embedding for each input to its corresponding positional embedding. The individual token and position embeddings are both of size $[1 \times d]$, so their sum is also $[1 \times d]$, This new embedding serves as the input for further processing. Fig. 8.14 shows the idea.

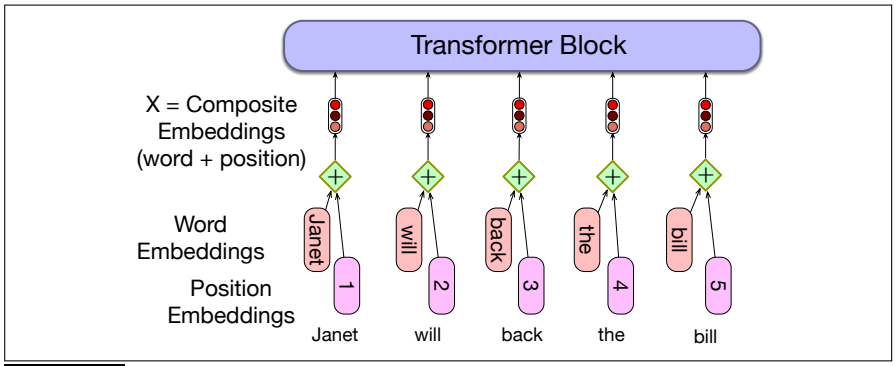


Figure 8.14 A simple way to model position: add an embedding of the absolute position to the token embedding to produce a new embedding of the same dimensionality.

The final representation of the input, the matrix **X**, is an $[N \times d]$ matrix in which each row i is the representation of the ith token in the input, computed by adding $\mathbf{E}[id(i)]$—the embedding of the id of the token that occurred at position i—, to $\mathbf{P}[i]$, the positional embedding of position i.

A potential problem with the simple position embedding approach is that there will be plenty of training examples for the initial positions in our inputs and correspondingly fewer at the outer length limits. These latter embeddings may be poorly trained and may not generalize well during testing. An alternative is to choose a static function that maps integer inputs to real-valued vectors in a way that better handles sequences of arbitrary length. A combination of sine and cosine functions with differing frequencies was used in the original transformer work. Sinusoidal position embeddings may also help in capturing the inherent relationships among the

positions, like the fact that position 4 in an input is more closely related to position 5 than it is to position 17.

relative position

A more complex style of positional embedding methods extend this idea of capturing relationships even further to directly represent **relative position** instead of absolute position, often implemented in the attention mechanism at each layer rather than being added once at the initial input.

8.5 The Language Modeling Head

language modeling head
head

The last component of the transformer we must introduce is the **language modeling head**. Here we are using the word **head** to mean the additional neural circuitry we add on top of the basic transformer architecture when we apply pretrained transformer models to various tasks. The language modeling head is the circuitry we need to do language modeling.

Recall that language models, from the simple n-gram models of Chapter 3 through the feedforward and RNN language models of Chapter 6 and Chapter 13, are word predictors. Given a context of words, they assign a probability to each possible next word. For example, if the preceding context is *"Thanks for all the"* and we want to know how likely the next word is *"fish"* we would compute:

$$P(fish|Thanks\ for\ all\ the)$$

Language models give us the ability to assign such a conditional probability to every possible next word, giving us a distribution over the entire vocabulary. The n-gram language models of Chapter 3 compute the probability of a word given counts of its occurrence with the $n-1$ prior words. The context is thus of size $n-1$. For transformer language models, the context is the size of the transformer's context window, which can be quite large, like 32K tokens for large models (and much larger contexts of millions of words are possible with special long-context architectures).

The job of the language modeling head is to take the output of the final transformer layer from the last token N and use it to predict the upcoming word at position $N+1$. Fig. 8.15 shows how to accomplish this task, taking the output of the last token at the last layer (the d-dimensional output embedding of shape $[1 \times d]$) and producing a probability distribution over words (from which we will choose one to generate).

The first module in Fig. 8.15 is a linear layer, whose job is to project from the output h_N^L, which represents the output token embedding at position N from the final block L, (hence of shape $[1 \times d]$) to the **logit** vector, or score vector, that will have a single score for each of the $|V|$ possible words in the vocabulary V. The logit vector $\mathbf{u}$ is thus of dimensionality $[1 \times |V|]$.

logit

This linear layer can be learned, but more commonly we tie this matrix to (the transpose of) the embedding matrix $\mathbf{E}$. Recall that in **weight tying**, we use the same weights for two different matrices in the model. Thus at the input stage of the transformer the embedding matrix (of shape $[|V| \times d]$) is used to map from a one-hot vector over the vocabulary (of shape $[1 \times |V|]$) to an embedding (of shape $[1 \times d]$). And then in the language model head, $\mathbf{E}^\mathsf{T}$, the transpose of the embedding matrix (of shape $[d \times |V|]$) is used to map back from an embedding (shape $[1 \times d]$) to a vector over the vocabulary (shape $[1 \times |V|]$). In the learning process, $\mathbf{E}$ will be optimized to be good at doing both of these mappings. We therefore sometimes call the transpose $\mathbf{E}^\mathsf{T}$ the **unembedding** layer because it is performing this reverse mapping.

weight tying
unembedding

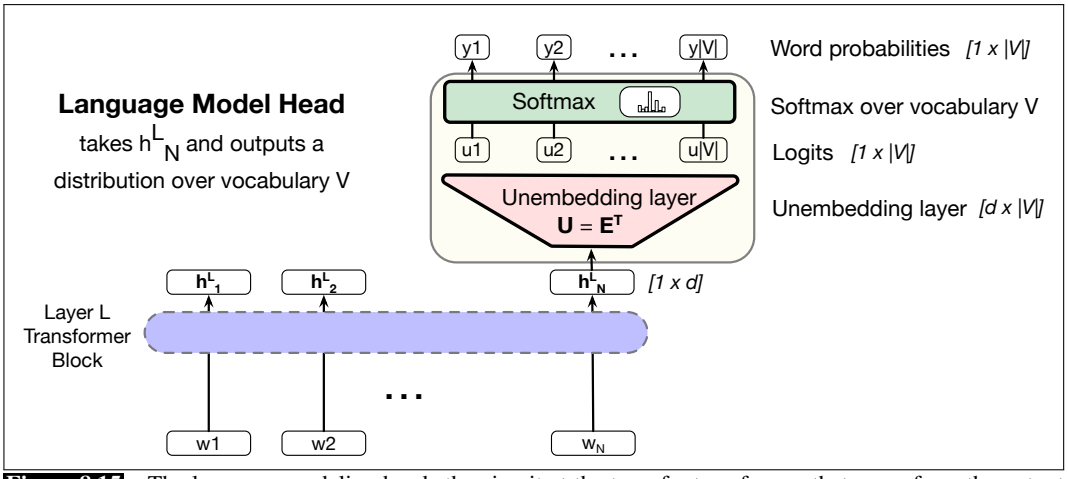


Figure 8.15 The language modeling head: the circuit at the top of a transformer that maps from the output embedding for token N from the last transformer layer ($\mathbf{h}_N^L$) to a probability distribution over words in the vocabulary V.

A softmax layer turns the logits $\mathbf{u}$ into the probabilities $\mathbf{y}$ over the vocabulary.

$$\mathbf{u} = \mathbf{h}_\mathbf{N}^\mathbf{L}\ \mathbf{E}^\mathbf{T} \tag{8.46}$$

$$\mathbf{y} = \text{softmax}(\mathbf{u}) \tag{8.47}$$

We can use these probabilities to do things like help assign a probability to a given text. But the most important usage is to generate text, which we do by **sampling** a word from these probabilities y. We might sample the highest probability word ('greedy' decoding), or use another of the sampling methods from Section 7.4 or Section 8.6.

In either case, whatever entry y_k we choose from the probability vector $\mathbf{y}$, we generate the word that has that index k.

Fig. 8.16 shows the total stacked architecture for one token i. Note that the input to each transformer layer x_i^ℓ is the same as the output from the preceding layer $h_i^{\ell-1}$.

decoder-only model

A terminological note before we conclude: You will sometimes see a transformer used for this kind of unidirectional causal language model called a **decoder-only model**. This is because this model constitutes roughly half of the **encoder-decoder model** for transformers that we'll see how to apply to machine translation in Chapter 12. (Confusingly, the original introduction of the transformer had an encoder-decoder architecture, and it was only later that the standard paradigm for causal language model was defined by using only the decoder part of this original architecture).

8.6 More on Sampling

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

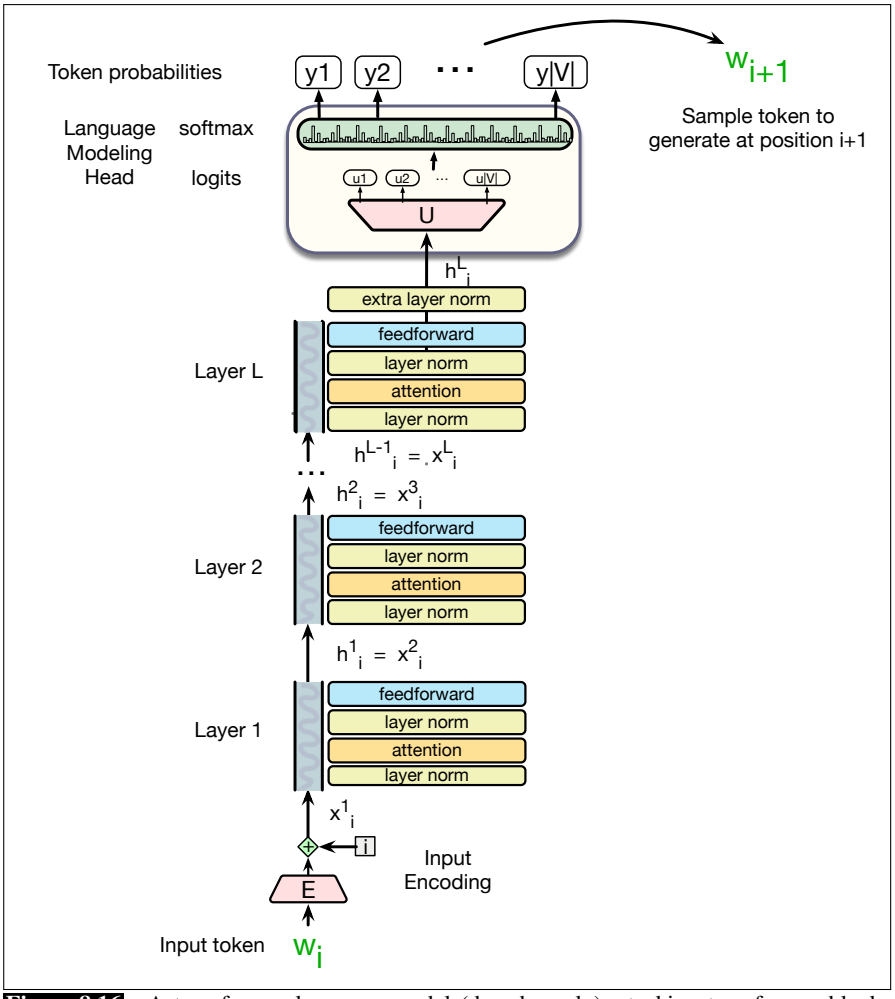


Figure 8.16 A transformer language model (decoder-only), stacking transformer blocks and mapping from an input token w_i to a predicted next token w_{i+1}.

words tend to be more creative and more diverse, but less factual and more likely to be incoherent or otherwise low-quality.

8.6.1 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.

8.6.2 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

$$\sum_{w \in V^{(p)}} P(w|\mathbf{w}_{<t}) \geq p. \tag{8.48}$$

8.7 Training

We described the training process for language models in the prior chapter. Recall that large language models are trained with cross-entropy loss, also called the negative log likelihood loss. At time t the cross-entropy loss is the negative log probability the model assigns to the next word in the training sequence, $-\log p(w_{t+1})$.

Fig. 8.17 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 by the model 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.

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).

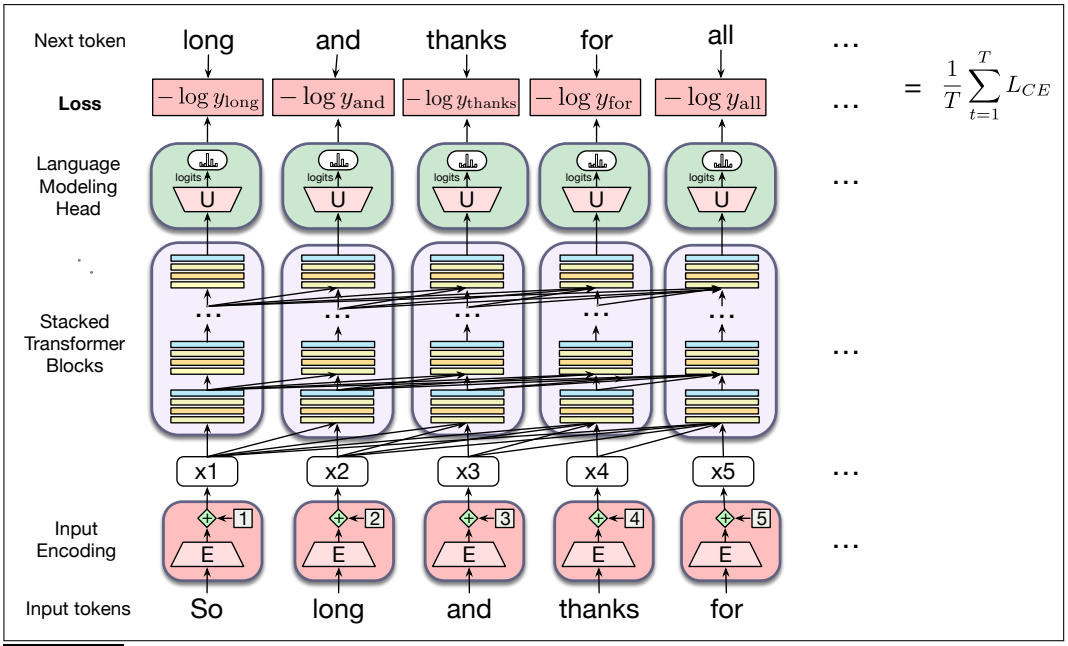


Figure 8.17 Training a transformer as a language model.

8.8 Dealing with Scale

Large language models are large. For example the *Llama 3.1 405B Instruct* model from Meta has 405 billion parameters (it has L=126 layers, model dimensionality d=16,384, and A=128 attention heads) and was trained on 15.6 terabytes of text tokens using a vocabulary of 128K tokens (Llama Team, 2024). 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 (PEFT).

8.8.1 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:

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N} \tag{8.49}$$

$$L(D) = \left(\frac{D_c}{D}\right)^{\alpha_D} \tag{8.50}$$

$$L(C) = \left(\frac{C_c}{C}\right)^{\alpha_C} \tag{8.51}$$

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_{attn} as the self-attention layer size, and d_{ff} the size of the feedforward layer):

$$\begin{aligned} N &\approx 2\, d\, n_{\text{layer}}(2\, d_{\text{attn}} + d_{\text{ff}}) \\ &\approx 12\, n_{\text{layer}}\, d^2 \\ &\quad (\text{assuming } d_{\text{attn}} = d_{\text{ff}}/4 = d) \end{aligned} \tag{8.52}$$

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, α_N, α_D, and α_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.[3]

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 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.

8.8.2 KV Cache

We saw in Fig. 8.11 and in Eq. 8.34 (repeated below) how the attention vector can be very efficiently computed in parallel for training, via two matrix multiplications:

$$\mathbf{A} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d_k}}\right)\mathbf{V} \tag{8.53}$$

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 $\mathbf{W^Q}$, $\mathbf{W^K}$, and $\mathbf{W^V}$ 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. 8.18 modifies Fig. 8.11 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

[3] 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).

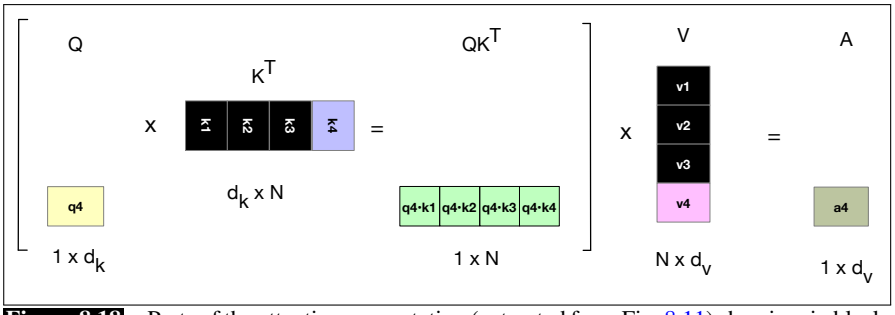


Figure 8.18 Parts of the attention computation (extracted from Fig. 8.11) showing, in black, the vectors that can be stored in the cache rather than recomputed when computing the attention score for the 4th token.

8.8.3 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 $[k \times d]$ that needs to be updated during finetuning via gradient descent. Normally this matrix would get updates $\Delta\mathbf{W}$ of dimensionality $[k \times d]$, for updating the $k \times d$ 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 $[k \times r]$ and $\mathbf{B}$ has size $[r \times d]$, and we choose r to be quite small, $r << \min(d,k)$. During finetuning we update $\mathbf{A}$ and $\mathbf{B}$ instead of $\mathbf{W}$. That is, we replace $\mathbf{W} + \Delta\mathbf{W}$ with $\mathbf{W} + \mathbf{AB}$. Fig. 8.19 shows the intuition. For replacing the forward pass $\mathbf{h} = \mathbf{xW}$, the new forward pass is instead:

$$\mathbf{h} = \mathbf{xW} + \mathbf{xAB} \tag{8.54}$$

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 $\mathbf{AB}$ 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.

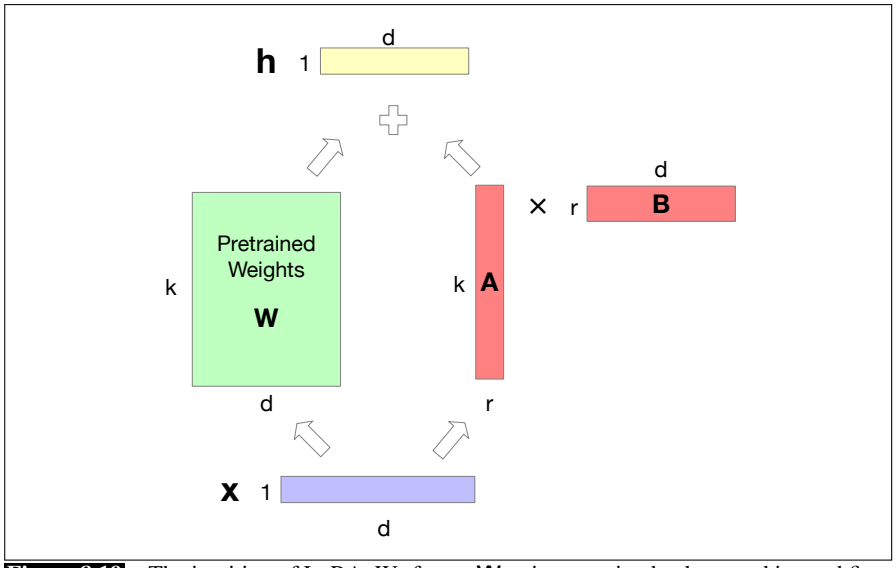


Figure 8.19 The intuition of LoRA. We freeze **W** to its pretrained values, and instead fine-tune by training a pair of matrices **A** and **B**, updating those instead of **W**, and just sum **W** and the updated **AB**.

8.9 Interpreting the Transformer

interpretability

How does a transformer-based language model manage to do so well at language tasks? The subfield of **interpretability**, sometimes called **mechanistic interpretability**, focuses on ways to understand mechanistically what is going on inside the transformer. In the next two subsections we discuss two well-studied aspects of transformer interpretability.

8.9.1 In-Context Learning and Induction Heads

As a way of getting a model to do what we want, we can think of prompting as being fundamentally different than pretraining. Learning via pretraining means updating the model's parameters by using gradient descent according to some loss function. But prompting with demonstrations can teach a model to do a new task. The model is learning something about the task from those demonstrations as it processes the prompt.

Even without demonstrations, we can think of the process of prompting as a kind of learning. For example, the further a model gets in a prompt, the better it tends to get at predicting the upcoming tokens. The information in the context is helping give the model more predictive power.

in-context learning

The term **in-context learning** was first proposed by Brown et al. (2020) in their introduction of the GPT3 system, to refer to either of these kinds of learning that language models do from their prompts. In-context learning means language models learning to do new tasks, better predict tokens, or generally reduce their loss during the forward-pass at inference-time, without any gradient-based updates to the model's parameters.

induction heads

How does in-context learning work? While we don't know for sure, there are some intriguing ideas. One hypothesis is based on the idea of **induction heads** (Elhage et al., 2021; Olsson et al., 2022). Induction heads are the name for a **circuit**,

which is a kind of abstract component of a network. The induction head circuit is part of the attention computation in transformers, discovered by looking at mini language models with only 1-2 attention heads.

The function of the induction head is to predict repeated sequences. For example if it sees the pattern AB...A in an input sequence, it predicts that B will follow, instantiating the **pattern completion** rule AB...A→ B. It does this by having a *prefix matching* component of the attention computation that, when looking at the current token A, searches back over the context to find a prior instance of A. If it finds one, the induction head has a *copying* mechanism that "copies" the token B that followed the earlier A, by increasing the probability the B will occur next. Fig. 8.20 shows an example.

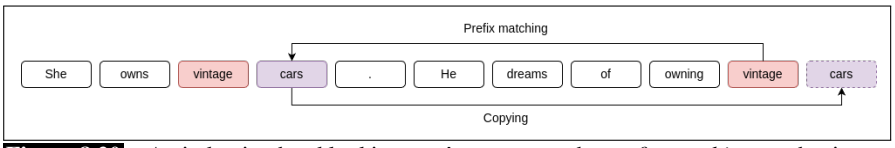


Figure 8.20 An induction head looking at `vintage` uses the *prefix matching* mechanism to find a prior instance of `vintage`, and the *copying* mechanism to predict that `cars` will occur again. Figure from Crosbie and Shutova (2022).

Olsson et al. (2022) propose that a generalized fuzzy version of this pattern completion rule, implementing a rule like A*B*...A→ B, where A* ≈ A and B* ≈ B (by ≈ we mean they are semantically similar in some way), might be responsible for in-context learning. Suggestive evidence for their hypothesis comes from Crosbie and Shutova (2022), who show that **ablating** induction heads causes in-context learning performance to decrease. **Ablation** is originally a medical term meaning the removal of something. We use it in NLP interpretability studies as a tool for testing causal effects; if we knock out a hypothesized cause, we would expect the effect to disappear. Crosbie and Shutova (2022) ablate induction heads by first finding attention heads that perform as induction heads on random input sequences, and then zeroing out the output of these heads by setting certain terms of the output matrix $\mathbf{W^O}$ to zero. Indeed they find that ablated models are much worse at in-context learning: they have much worse performance at learning from demonstrations in the prompts.

ablating

8.9.2 Logit Lens

logit lens

Another useful interpretability tool, the **logit lens** (Nostalgebraist, 2020), offers a way to visualize what the internal layers of the transformer might be representing.

The idea is that we take any vector from any layer of the transformer and, pretending that it is the prefinal embedding, simply multiply it by the **unembedding layer** to get logits, and compute a softmax to see the distribution over words that that vector might be representing. This can be a useful window into the internal representations of the model. Since the network wasn't trained to make the internal representations function in this way, the logit lens doesn't always work perfectly, but this can still be a useful trick to help us visualize the internal layers of a transformer.

8.10 Summary

This chapter has introduced the transformer and its components for the language modeling task introduced in the previous chapter. Here's a summary of the main points that we covered:

- Transformers are non-recurrent networks based on **multi-head attention**, a kind of **self-attention**. A multi-head attention computation takes an input vector $\mathbf{x}_i$ and maps it to an output $\mathbf{a}_i$ by adding in vectors from prior tokens, weighted by how relevant they are for the processing of the current word.
- A **transformer block** consists of a **residual stream** in which the input from the prior layer is passed up to the next layer, with the output of different components added to it. These components include a **multi-head attention layer** followed by a **feedforward layer**, each preceded by **layer normalizations**. Transformer blocks are stacked to make deeper and more powerful networks.
- The input to a transformer is computed by adding an embedding (computed with an **embedding matrix**) to a **positional encoding** that represents the sequential position of the token in the window.
- Language models can be built out of stacks of transformer blocks, with a **language model head** at the top, which applies an **unembedding** matrix to the output $\mathbf{H}$ of the top layer to generate the **logits**, which are then passed through a softmax to generate word probabilities.
- Transformer-based language models have a wide context window (200K tokens or even more for very large models with special mechanisms) allowing them to draw on enormous amounts of context to predict upcoming words.
- There are various computational tricks for making large language models more efficient, such as the **KV cache** and **parameter-efficient finetuning**.

Historical Notes

The transformer (Vaswani et al., 2017) was developed drawing on two lines of prior research: **self-attention** and **memory networks**.

Encoder-decoder attention, the idea of using a soft weighting over the encodings of input words to inform a generative decoder (see Chapter 12) was developed by Graves (2013) in the context of handwriting generation, and Bahdanau et al. (2015) for MT. This idea was extended to self-attention by dropping the need for separate encoding and decoding sequences and instead seeing attention as a way of weighting the tokens in collecting information passed from lower layers to higher layers (Ling et al., 2015; Cheng et al., 2016; Liu et al., 2016).

Other aspects of the transformer, including the terminology of key, query, and value, came from **memory networks**, a mechanism for adding an external read-write memory to networks, by using an embedding of a query to match keys representing content in an associative memory (Sukhbaatar et al., 2015; Weston et al., 2015; Graves et al., 2014).

MORE HISTORY TBD IN NEXT DRAFT.