Language Model Overview

Basics

Popular language models generate text ‘auto-regressively’, which means that they generate the first n words of text, then given the first n words they generate the (n+1)th word. They continue this process of generating the next word given their previously generated words until they generate the full text.

This is a rough outline of the generation process, but we can be more precise in a couple of ways. First, language models don’t actually generate words, they generate what are called tokens. These can be single characters, entire words, or parts of words. The idea is that we break the text into some chunks, which are optimal with respect to certain metrics, and have the language model generate these. Language models usually have a vocabulary of around 50,000 tokens. Why do we want to have language models generate tokens instead of words? If we have a fixed set of words that the model can generate, it is overly restrictive. With tokens, the model has the flexibility to generate many different words, even words that are not present in the training data. On the other hand, we could have the model only output single characters. This gives it the same flexibility as generating tokens, but with a smaller vocabulary, which decreases the number of model parameters necessary. However, if the model generates tokens it doesn’t have to spend as much computation during training to determine which sets of characters commonly string together, since tokens will be composed of characters that are commonly strung together. The generation process with tokens is also not as costly as with characters since a token may consist of multiple characters. To generate a token consisting of multiple characters, the character-based model would have to do a separate computation for each character in the token. In contrast the token-based model would only have to do one computation for the entire token.

Secondly, the language model doesn’t simply output a token at each timestep. At each timestep the model outputs the probability of each next token. In other words, if the previous tokens are denoted as y, then the model outputs $p(x|y)$ for each token x. Since we have around 50,000 tokens, it outputs a vector with 50,000 non-negative entries that sum to 1. Each entry i of this vector corresponds to the probability of the ith token. To choose which token to output next, we can sample from this probability distribution. However, in practice we don’t simply sample from this distribution. Usually sampling methods will sample from a modified distribution. For example we could exclude tokens with probability less than some threshold, or exclude tokens that are not the top 50 most likely tokens among other modifications.

So the full process to generate text given some input y (which may be empty) is:

1. Run the text through the model
2. The model outputs the probability distribution for the next token $p(x|y)$
3. (Optional, but recommended) Modify this probability distribution
4. Sample from the probability distribution to get token x
5. Append x to the input y to get a new text y'
6. Go back to step 1, this time with the new text y'

This loop continuous until the model outputs a special end of text token.

Architecture

Nearly all popular language models today are based on the transformer architecture, which was introduced in 2017 here: https://arxiv.org/abs/1706.03762. Before 2017, recurrent neural networks (RNNs) were the most popular type of neural network for sequential data like text. One of the main reasons transformers overtook RNNs is that they can process training data much faster on the hardware we use to run neural networks, since they are more parallilizable than RNNs. Recurrent neural networks operate in an intuitive way: they take in tokens one-by-one and update their internal state after each token they see. This internal state should sort of be a summary of the information they have seen so far. An advantage that RNNs have over transformers is that since RNNs encode the text they have seen into an internal state, they can potentially look at any length of text and make good predictions, while transformers have a limited length of tokens they can look at. However, when training, updating an internal state is not parallelizable, which is one of the main reasons transformers have become more popular than RNNs.

Popular language models have around 1 billion to 1 trillion parameters. GPT-3 has about 175 billion parameters for example. Unfortunately, the number of parameters in GPT-4 (often considered the best LLM currently) and Palm-2 (Google's most recent LLM) have not been released.

Training

We have seen that in the generation process, the language model outputs the probability of the next token in a sequence of text. Now we will look at how the model learns a good probability distribution for the next token. The main idea is that the model will output its probability distribution of the next token, we compare that to what the true token is in some given text, and then update the model parameters so that the distribution it outputs is closer to the true token.

First we need training data. This is commonly a bunch of general text from the internet, for example from sites like Reddit and GitHub, maybe along with some books or some proprietary text. Since data from the internet often has nonsensical strings of text, for example repetitive words or random symbols, this text is filtered according to some rules. Then the tokenizer is trained on this text, meaning it finds some optimal set of tokens to split the text into. For popular language models, the total number of tokens in the training data is usually 100 billion to 2 trillion. Usually each training example will only be seen 1-2 times during the whole training run.

Like most neural networks, to train we define an objective function we want to minimize, then follow gradient descent on the network parameters to minimize the objective function. The objective function in this case is called cross-entropy loss. For a single output from the neural network, which as discussed above is a vector of probabilities for the next token, the cross-entropy loss is $-\sum_i y_i log(p_i)$ where the sum index is the tokens, p_i is the probability of the ith token (so the ith entry of the output vector), and y_i is the ‘probability’ of the true token. In this case, the true token is known so really $y_i = 1$ if the next token is i and 0 otherwise. So we can rewrite the loss as just $-log(p_i)$ where the true next token in the data is i. So in minimizing this function, we maximize the log probability of the true token.

We update the parameters based on the gradient of of cross-entropy loss in batches of training data. This amounts to computing the gradient for each training example in the batch separately and then summing all these together, then using the sum of gradients to update the network parameters. For GPT-3 and Palm, the batch sizes are 1-4M tokens.

Training is done on GPUs usually, TPUs less often, and more rarely some other hardware. Multiple GPUs are usually used, from 100 to thousands to train LLMs. There are multiple strategies to divide the computation among the available GPUs, and these strategies are usually used in combination. One strategy is to split up the model parameters onto multiple GPUs. This is often necessary since the model is often too large to fit on a single GPU. Another strategy is to have different GPUs compute the loss for different training examples, and later combine the gradients of these losses to simultaneously update the model parameters on the different GPUs.

Training an LLM is impractical for most people and smaller organizations due to the computational costs. However, it is practical for them to fine-tune the model or make other modifications after the initial training.

Capabilities and Limitations After Training

After training, the LLM can generate mostly coherent text. However, it does not have the capabilities of something like chatGPT, which can respond well to human input. Since we have only trained the model to predict the next word in text, it often predicts words that are plausible continuations, but not what we would like from a chatbot. For example, given a question from a person, answering this question is one plausible continuation. But another could be to give an incorrect answer, since many incorrect answers are probably in the training data. Another way to continue might be to form a list of similar questions and another might be to imagine that this is part of a dialogue in a movie. I have found that very often the model just mostly seems to ignore the question or start listing a bunch of similar questions. Importantly though, the text generated after training usually makes sense, but is just not what we want from a chatbot. This is often called the ‘alignment’ problem since the model presumably has the ability to generate the text we want from a chatbot, but very often chooses to generate valid continuations that we do not want.

The main methods of getting the model to generate more chatbot-like outputs are fine-tuning and reinforcement learning. The objective for fine-tuning is either the same, or very similar to that of the training above while the reinforcement learning approach is very different.

Fine-tuning

Fine-tuning refers to further training of the model after the initial training discussed above. People commonly refer to the type of training done above, which is on a general text dataset as ‘pre-training’. In fact, GPT stands for “generalized pre-trained”. The idea being that pre-training gives the model some general abilities, which we will later refine in the fine-tuning phase. Usually (probably always) fine-tuning is done on a much smaller dataset than the initial training. This could be any sort of data of interest, for example some math questions and answers.

There are 2 main reasons we use fine-tuning. One is to teach the model new things, for example some specialized information for healthcare. The other is to get it to respond more like a chatbot. Usually these are separate reasons, in the sense that if we use fine-tuning to teach it to respond like a chatbot, we won’t usually be teaching it new concepts but will just be teaching it the proper input-response pattern it should adhere to. Often, people refer to the process of fine-tuning on chatbot-like input-response data as ‘instruction fine-tuning’.

To teach the model new concepts with fine-tuning, the objective is usually the same as in the pre-training phase, where we minimize the cross-entropy. There may be minor differences in the training details like learning rate and batch size. This fine-tuning gives the model some specialized knowledge or abilities that were not in the original training data.

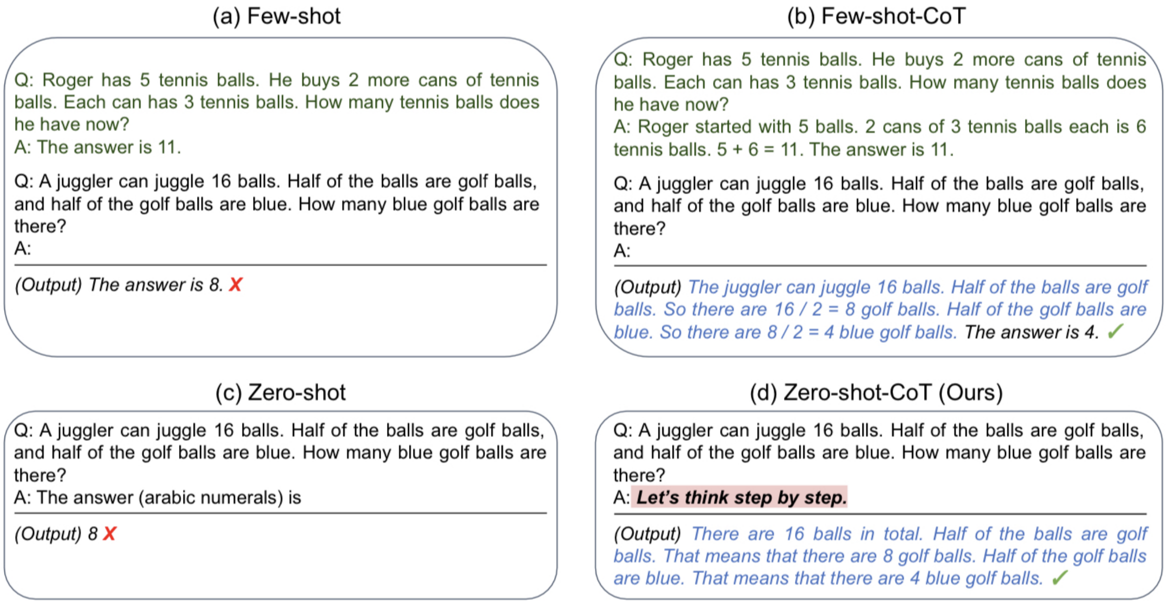
For instruction fine-tuning, the training data consists of input-response pairs of text. For example people commonly try to teach a model to behave more like chatGPT by using input-response pairs where the input is a usual question or command to an LLM while the response is taken from running chatGPT on the input. Another type of data is from websites where people ask questions like stack exchange. Well-funded organizations like OpenAI hire experts in various domains to generate instruction fine-tuning datasets to train their models. The important point for instruction fine-tuning data is that the input is the type of input that people give to a chatbot and the output is the type of output we want from the chatbot. The training objective for this data is the same as for pre-training except that we usually weight the input tokens less or ignore them completely. In other words, for the model predictions on the response tokens the objective is the same cross-entropy as in pre-training. But we ignore the loss on the input tokens, or we multiply the loss on them by some small number like 1/10.

After instruction fine-tuning, the model will respond to user input much more like a chatbot. It will be much less likely to ignore the user input or continue with other undesirable responses mentioned previously. However, the model still makes many mistakes that we would like to minimize. The model will often guess or make up information. Also, when given input with an incorrect premise, the model will not usually challenge this and respond as if the premise is correct. The model also may be biased and give harmful advice. Reinforcement learning has had some success in reducing these issues. We will talk about this later in the course.

Prompting

One of the unfortunate issues with current language models is that their outputs are dependent on how you phrase the input. Many research papers are dedicated to investigating how to best prompt LLMs. A few prompting strategies are:

1. Zero-shot: this is just the regular prompt
2. Few-shot: include examples of the desired behavior along with the prompt
3. Chain of thought: include examples that also explain their reasoning. The idea here is that the model will reason through the answer before supplying the answer, rather than simply outputting the answer alone.
4. ‘Think step-by-step’ or zero-shot chain of thought: tell the model to think step-by-step. The idea is similar to 3 except we do not supply examples. https://arxiv.org/pdf/2205.11916.pdf



(a) Few-shot

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: The answer is 11.

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?
A:

(Output) The answer is 8. ✗

(b) Few-shot-CoT

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?
A:

(Output) The juggler can juggle 16 balls. Half of the balls are golf balls. So there are 16 / 2 = 8 golf balls. Half of the golf balls are blue. So there are 8 / 2 = 4 blue golf balls. The answer is 4. ✓

(c) Zero-shot

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?
A: The answer (arabic numerals) is

(Output) 8 ✗

(d) Zero-shot-CoT (Ours)

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?
A: ***Let’s think step by step.***

(Output) There are 16 balls in total. Half of the balls are golf balls. That means that there are 8 golf balls. Half of the golf balls are blue. That means that there are 4 blue golf balls. ✓

Image from https://arxiv.org/abs/2205.11916.

Few-shot prompting is also called ‘in-context learning’. This term refers to the ability of large language models to learn new capabilities ‘in-context’, meaning within the inputs or prompt they are given, as opposed to learning by updating model parameters. In in-context learning the model learns some new ability from examples, for instance how to answer specific types of questions, without the parameter updates that occur in the traditional machine learning training process. In-context learning is also useful for teaching the model how to use tools that were not in the training dataset, as we will see below.