05232023 Build Andrej Karpathy Session

**Build 2023**

**Andrej Karpathy**

**Tuesday, May 23, 2023**

**ANDREJ KARPATHY:** Hi, everyone. I’m happy to be here to tell you about the state of GPT. And more generally, about the rapidly growing ecosystem of large language models. So I would like to partition the talk into two parts.

In the first part, I would like to tell you about how we train GPT assistants. And then in the second part, we are going to take a look at how we can use these assistants effectively for your applications.

So first, let’s take a look at the emerging recipe for how to train these assistants. And keep in mind that this is all very new and so rapidly evolving. But so far, the recipe looks something like this.

Now, this is kind of a complicated slide, so I’m going to go through it piece by piece. But roughly speaking, we have four major stages: pre-training, supervised fine tuning, reward modeling, reinforcement learning. They follow each other serially.

Now, in each stage we have a dataset that powers that stage. We have an algorithm that for our purposes will be an objective for training a neural network. And then we have a resulting model. And then there’s some notes on the bottom.

So the first stage we’re going to start with is the pre-training stage. Now, this stage is kind of special in this diagram, and this diagram is not to scale because this stage is where all of the computational work basically happens. This is 99% of the training compute time and also flops.

And so this is where we are dealing with internet-scale datasets with thousands of GPUs in the supercomputer and also months of training, potentially. The other three stages are fine tuning stages that are much more along the lines of some few number of GPUs and hours or days.

So let’s take a look at the pre-training stage to achieve a base model.

First, we’re going to gather a large amount of data. Here’s an example of what we call a data mixture that comes from this paper that was released by Meta, where they released this LLaMA-based model.

Now, you can see roughly the kinds of datasets that enter into these collections. So we have Common Crawl, which is just a web scrape, C4, which is also a Common Crawl, and then some high-quality datasets as well.

So for example, GitHub, Wikipedia, Books, Archive, Stock Exchange and so on. These are all mixed up together and then they are sampled according to some given proportions, and that forms the training sets for the neural net for the GPT.

Now, before we can actually train on this data, we need to go through one more pre-processing step, and that is tokenization. And this is basically a translation of the raw text that we scraped from the internet into sequences of integers, because that’s the native representation over which GPT functions.

Now, this is a lossless kind of translation between pieces of text and tokens and integers, and there are a number of algorithms for the stage. Typically, for example, you could use something like byte pair encoding which iteratively merges little text chunks and groups them into tokens.

And so here I’m showing some example chunks of these tokens, and then this is the raw integer sequence that will actually feed into a transformer.

Now, here I’m showing two sort of like examples for hyperparameters that govern the stage. GPT-4, we did not release too much information about how it was trained and so on. So I’m using GPT-3’s numbers, but GPT-3 is of course a little bit old by now, about three years ago. But LLaMA is a fairly recent model from Meta.

So these are roughly the orders of magnitude that we’re dealing with when we’re doing pre-training. The vocabulary size is usually 10,000 tokens. The context length is usually something like 2,000, 4,000, or nowadays, even 100,000. And this governs the maximum number of integers that the GPT will look at when it’s trying to predict the next integer in a sequence.

You can see that roughly the number of parameters is, say, 65 billion for LLaMA. Now, even though LLaMA has only 65 parameters compared to GPT-3’s 175 billion parameters, LLaMA is a significantly more powerful model and intuitively that’s because the model is trained for significantly longer, in this case, 1.4 trillion tokens instead of just 300 billion tokens. You shouldn’t judge the power of a model just by the number of parameters that it contains.

Below, I’m showing some tables of a number of hyperparameters that typically go into specifying the transformer neural network. So the number of heads, the dimension size, number of layers and so on.

And on the bottom, I’m showing some training hyperparameters. For example, to train the 65B model, Meta used 2,000 GPUs, roughly 21 days of training, and roughly several million dollars. And so that’s the rough orders of magnitude that you should have in mind for the pre-training stage.

Now, when we’re actually pre-training, what happens? Roughly speaking, we are going to take our tokens and we’re going to lay them out into data batches. We have these arrays that will feed into the transformer, and these arrays are B, the batch size, and these are all independent examples stacked up in rows, and B x T, T being the maximum context length.

So in my picture I only have 10, but this is the context length, and so this could be 2,000, 4,000, et cetera. These are extremely long rows, and what we do is we take these documents, and we pack them into rows, and we delimit them with these special end-of-text tokens, basically telling the transformer where a new document begins. And so here I have a few examples of documents and then I stretched them out into into this input.

Now, we’re going to feed all of these numbers into transformer. And let me let me just focus on a single particular cell, but the same thing will happen at every every cell in this diagram.

So let’s look at the green cell. The green cell is going to take a look at all of the tokens before it, so all of the tokens in yellow. And we’re going to feed that entire context into the transformer neural network. And the transformer is going to try to predict the next token in the sequence, in this case in red.

Now, the transformer, I don’t have too much time to unfortunately, go into the full details of this neural network architecture, but it is just a large blob of neural net stuff for our purposes, and it’s got several – 10 billion parameters typically, or something like that.

And of course, as they tune these parameters, you’re getting slightly different predicted distributions for every single one of these cells. And so, for example, if our vocabulary size is 50,257 tokens, then we’re going to have that many numbers because we need to specify a probability distribution for what comes next, so that we basically have a probability for whatever may follow.

Now, in this specific example for the specific cell, 513 will come next. And so we can use this as a source of supervision to update our transformer weights. And so we’re applying this basically on every single cell in the parallel. And we keep swapping batches and we’re trying to get the transformer to make the correct predictions over what token comes next in a sequence.

So let me show you more concretely what this looks like when you train one of these models. This is actually coming from the *New York Times,* and they trained a small GPT on Shakespeare. And so here’s a small snippet of Shakespeare, and they trained their GPT on it.

Now, in the beginning at initialization, the GPT starts with completely random weights, so you’re just getting completely random outputs as well. But over time, as you train the GPT longer and longer, you are getting more and more coherent and consistent sorts of samples from the model.

And the way you sample from it, of course, is you predict what comes next. You sample from that distribution and you keep feeding that back into the process and you can basically sample large sequences.

And so by the end, you see that the transformer has learned about words and where to put spaces and where to put commas and so on. And so we’re making more and more consistent predictions over time.

These are the kinds of plots that you’re looking at when you’re doing model pre-training. Effectively, we’re looking at the loss function over time as you train. And low loss means that our transformer is predicting the correct – is giving a higher probability to the correct next integer in a sequence.

Now, what are we going to do with this model once we’ve trained it after a month? Well, the first thing that we noticed, we in the field, is that these models are basically in the process of language modeling, learn very powerful general representations, and it’s possible to very efficiently fine tune them for any arbitrary downstream task you might be interested in.

So as an example, if you’re interested in sentiment classification, the approach used to be that you collect a bunch of positives and negatives and then you train some kind of an NLP model for for that. But the new approach is to ignore sentiment, classification, go off and do large language model pre-training, train the large transformer and then you can only – you may only have a few examples and you can very efficiently fine tune your model for that task.

And so this works very well in practice. And the reason for this is that basically the transformer is forced to multitask a huge amount of tasks in the language modeling task, because just just in terms of predicting the next token, it’s forced to understand a lot about the structure of the text and all the different concepts therein.

So that was GPT-1.

Now, around the time of GPT-2, people noticed that actually even better than fine tuning, you can actually prompt these models very effectively. So these are language models and they want to complete documents, and so you can actually trick them into performing tasks just by arranging these fake documents.

So in this example, for example, we have some passage and then we sort of like do, "QA, QA, QA," and this is called few-shot prompt, and then we do Q, and then as the transformer is trying complete the document, it’s actually answering our question. And so this is an example of prompt engineering a base model, making it belief that it’s sort of imitating a document and it’s getting it to perform a task.

And so this kicked off, I think, the era of, I would say, prompting over fine tuning and seeing that this actually can work extremely well on a lot of problems, even without training any neural networks, fine tuning or so on.

Now, since then, we’ve seen an entire evolutionary tree of base models that everyone has trained. Not all of these models are available. For example, the GPT-4 base model was never released. The GPT-4 model that you might be interacting with over API is not a base model. It’s an assistant model and we’re going to cover how to get those in a bit.

The GPT-3 base model is available via the API under the named DaVinci and the GPT-2 base model is available even as weights on our GitHub repo. But currently the best available base model probably is the LLaMA series from Meta, although it is not commercially licensed.

Now, one thing to point out is base models are not assistants. They don’t want to make answers to your questions. They just want to complete documents. So if you tell them, "Write a poem about the bread and cheese," it will just – you know, it will answer questions with more questions. It’s just completing what it thinks is a document.

However, you can prompt them in a specific way for base models that is more likely to work. So as an example, here’s a poem about bread and cheese, and in that case it will autocomplete correctly.

You can even trick base models into being assistants. And the way you would do this is you would create like a specific few-shot prompt that makes it look like there’s some kind of a document between a human and assistant, and they’re exchanging sort of information.

And then at the bottom you sort of put your query at the end, and the base model will sort of like condition itself into being like a helpful assistant and kind of answer. But this is not very reliable and doesn’t work super well in practice, although it can be done.

So instead we have a different path to make actual GPT assistants, not just base model document completers. And so that takes us into supervised fine tuning. So in the supervised fine-tuning stage, we are going to collect small, but high-quality datasets. And in this case, we’re going to ask human contractors to gather data of the form prompt and ideal response. And we’re going to collect lots of these, typically tens of thousands or something like that.

And then we’re going to still do language modeling on this data. So nothing changed algorithmically. We’re just swapping out a training set. So it used to be internet documents, which is a high-quantity/low-quality, for basically QA prompt response kinds of data, and that is low-quantity/high-quality.

So we would still do language modeling. And then after training we get an SFT model, and you can actually deploy these models, and they are actual assistants, and they work to some extent.

Let me show you what an example demonstration might look like. So here’s something that a human contractor might come up with. Here’s some random prompt, "Can you write a short introduction about the relevance of the term monopsony," or something like that? And then the contractor also writes out an ideal response.

And when they write out these responses, they are following extensive labeling documentations and they’re being asked to be helpful, truthful and harmless. These are the labeling instructions here. You probably can’t read it, and neither can I, but they’re long and this is just people following instructions and trying to complete these prompts.

So that’s what the dataset looks like. And you can train these models and this works to some extent.

Now, you can actually continue the pipeline from here on and go into, RLHF, reinforcement learning from human feedback, which consists of both reward modeling and reinforcement learning.

So let me cover that and then I’ll come back to why you may want to go through the extra steps and how that compares to just SFT models.

So in the reward modeling step, what we’re going to do is we’re now going to shift our data collection to be of the form of comparisons. So here’s an example of what our dataset will look like. I have the same prompt, identical prompt on the top, which is asking the assistant to write a program or a function that checks if a given string is a palindrome.

And then what we do is we take the SFT model, which we’ve already trained, and we create multiple completions. So in this case we have three completions that the model has created. And then we ask people to rank these completions.

So if you stare at this for a while, and by the way, these are very difficult things to do to compare some of these predictions, and this can take people even hours for single prompt completion pairs. But let’s say we decided that one of these is much better than the others and so on, and so we rank them. We can then follow that with something that looks very much kind of like a binary classification on all the possible pairs between these completions.

So what we do now is we lay out our prompt in rows and the prompts is identical across all three rows here. So it’s all the same prompt, but the completion is varied, and so the yellow tokens are coming from the SFT model.

Then what we do is we append another special reward readout token at the end, and we basically only supervise the transformer at this single green token, and the transformer will predict some reward for how good that completion is for that prompt.

And so it basically makes a guess about the quality of each completion. And then once it makes a guess for every one of them, we also have the ground truth, which is telling us the ranking of them. And so we can actually enforce that some of these numbers should be much higher than others and so on. We formulate this into a loss function, and we train our model to make reward predictions that are consistent with the ground truth coming from the comparisons from all these contractors.

So this is how we train our reward model, and that allows us to score how good a completion is for a prompt.

Once we have a reward model, we can’t deploy this because this is not very useful as an assistant by itself, but it’s very useful for the reinforcement learning stage that follows now. Because we have a reward model, we can score the quality of any arbitrary completion for any given prompt.

So what we do during reinforcement learning is we basically get, again, a large collection of prompts and now we do reinforcement learning with respect to the reward model.

Here’s what that looks like. We take a single prompt, we lay it out in rows, and now we use the SFT model. We use basically the model we’d like to train, which is initialized as SFT model, to create some completions in yellow. And then we append the reward token again, and we read off the reward according to the reward model, which is now kept fixed. It doesn’t change anymore.

And now, the reward model tells us the quality of every single completion for these prompts. And so, what we can do is we can now just basically apply the same language modeling loss function, but we’re currently training on the yellow tokens. And we are weighing the language modeling objective by the rewards indicated by the reward model.

As an example, in the first row, the reward model said that this is a fairly high scoring completion. And so, all the tokens that we happen to sample on the first row are going to get reinforced, and they’re going to get higher probabilities for the future. Conversely, on the second row, the reward model really did not like this completion, -1.2. And so, therefore, every single token that we sampled in that second row is going to get a slightly higher probability for the future. And we do this over and over on many prompts, on many batches. And basically, we get a policy which creates yellow tokens here, and it’s basically all of them, all of the completions here will score high according to the reward model that we trained in the previous stage.

That’s how we train. That’s what the RLHF pipeline is. And then at the end, you get a model that you could deploy. And so, as an example, ChatGPT is an RLHF model, but some other models that you might come across, for example, of the (inaudible) and so on, these are SFT models. We have base models, SFT models, and RLHF models, and that’s kind of like the state of things there.

Now why would you want to do RLHF? One answer that is kind of not that exciting is that it just works better. This comes from the instruct GPT paper. According to these experiments a while ago now, these PPO models are RLHF. And we see that they are basically just preferred in a lot of comparisons, when we give them to humans. Humans just prefer basically tokens that come from RLHF models, compared to SFT models, compared to base model that is prompted to be an assistant. And so, it just works better.

But you might ask why? Why does it work better? And I don’t think that there’s a single amazing answer that the community has really agreed on, but I will just offer one reason, potentially, and it has to do with the asymmetry between how easy computationally it is to compare versus generate.

Let’s take an example of generating a haiku. Suppose I ask a model to write a haiku about paperclips. If you’re a contractor trying to give training data, then imagine being a contractor collecting basically data for the SFT. How are you supposed to create a nice haiku for a paperclip? You might just not be very good at that, but if I give you a few examples of haikus, you might be able to appreciate some of these haikus a lot more than others. And so, judging which one of these is good is much easier task. And so, basically this asymmetry makes it so that comparisons are a better way to potentially leverage yourself, as a human and your judgment to create a slightly better model.

Now, RLHF models are not strictly an improvement on the base models, in some cases. In particular, we’ve noticed, for example, that they lose some entropy. That means that they give more (PT?) results. They can output lower variations. They can output samples with lower variation than the base model. Base model has lots of entropy and will give lots of diverse outputs.

For example, one kind of place where I still prefer to use a base model is in the setup where you basically have n things and you want to generate more things like it. And so, here is an example that I just cooked up. I want to generate cool Pokémon names. I gave it seven Pokémon names, and I asked the base model to complete the document. And it gave me a lot more Pokémon names. These are fictitious. I tried to look them up. I don’t believe there are actual Pokémons. And this is the kind of task that I think base model would be good at, because it still has lots of entropy and will give you lots of diverse, cool kind of more things that look like whatever you give it before.

Having said all that, these are kind of like the assistant models that are probably available to you at this point. There’s a team at Berkeley that ranked a lot of the available assistant models and gave them basically ELO ratings. Currently, some of the best models, of course, are GPT-4, by far, I would say, followed by Claude GPT 3.5 and then a number of models. Some of these might be available as weights, like the Kuna, Koala, etcetera. And the first three rows here, they are all RLHF models, and all of the other models, to my knowledge, are SFT models, I believe.

Okay, so that’s how we train these models on the high level. Now, I’m going to switch gears, and let’s look at how we can best apply the GPT assistant model to your problems.

Now, I would like to work in something of a concrete example. Let’s work with the concrete example here. Let’s say that you are working on an article or a blog post, and you’re going to write this sentence at the end. “California’s population is 53 times that of Alaska.” For some reason, you want to compare the populations of these two states.

Think about the rich internal monologue and tool use, and how much work actually goes computationally in your brain to generate this one final sentence. Here’s maybe what that could look like in your brain.

Okay, for this next step, let me blog. Let me compare these two populations. Okay, first, obviously, I need to get both of these populations. Now, I know that I probably don’t know these populations off the top of my head, so I’m kind of like aware of what I know or don’t know of my self-knowledge, right? I do some tool use, and I go to Wikipedia and I look up California’s population and Alaska’s population.

Now I know that I should divide the two, but again, I know that dividing 39.2 by 0.74 is very unlikely to succeed. That’s not the kind of thing that I can do in my head. And so, therefore I’m going to rely on the calculator. I’m going to use a calculator, punch it in and see that the output is roughly 53. And then maybe I do some reflection and sanity checks in my brain, so that 53 makes sense. Well, that’s quite a large fraction, but then California has the most populous state, so maybe that looks okay.

Then I have all the information I might need, and now I get to the sort of creative portion of writing. I might start to write something like, “California has 53x times greater.” And then I think to myself, that’s actually really awkward phrasing. Let me actually delete that and let me try again. And so, as I’m writing, I have this separate process, almost inspecting what I’m writing and judging whether it looks good or not. And then maybe I delete and maybe I reframe it, and then maybe I’m happy with what comes out.

Basically, long story short, a ton happens under the hood in terms of your internal monologue when you create sentences like this. But what does a sentence like this look like when we are training a GPT on it?

From GPT’s perspective, this is just a sequence of tokens. GPT, when it’s reading or generating these tokens, it just goes chunk, chunk, chunk, chunk, and each chunk is roughly the same amount of computational work for each token. And these transformers are not very shallow networks. They have about 80 layers of reasoning, but 80 is still not too much. And so, this transformer is going to do its best to imitate, but of course, the process here looks very, very different from the process that you took.

In particular, in our final artifacts, in the dataset that we create and then eventually feed to LLMs, all of that internal dialog is completely stripped. And unlike you, the GPT will look at every single token and spend the same amount of compute on every one of them. And so, you can’t expect it to actually like – well, you can’t expect it to do sort of do too much work per token.

And also, in particular, basically these transformers are just like token simulators. They don’t know what they don’t know. They just imitate the next token. They don’t know what they’re good at or not good at. They just try their best to imitate the next token. They don’t reflect in the loop. They don’t sanity check anything. They don’t correct their mistakes along the way by default. They just sample token sequences. They don’t have separate inner monologue streams in their head, right? They are evaluating what’s happening.

Now, they do have some sort of cognitive advantages, I would say, and that is that they do actually have a very large fact-based knowledge across a vast number of areas because they have, say, several 10 billion parameters. It’s a lot of storage for a lot of facts, and they also, I think, have a relatively large and perfect working memory. Whatever fits into the context window is immediately available to the transformer through its internal self-attention mechanism. And so, it’s kind of like perfect memory, but it’s got that finite size. But the transformer has a very direct access to it. And so, it can lossless-ly remember anything that is inside its context window.

That’s kind of how I would compare those two. And the reason I bring all of this up is because I think to a large extent, prompting is just making up for this sort of cognitive difference between these two kind of architectures, like our brains here and LLM brains. You can look at it that way almost.

Here’s one thing that people found, for example, works pretty well in practice. Especially if your tasks require reasoning, you can’t expect the transformer to do too much reasoning per token. And so, you have to really spread out the reasoning across more and more tokens. For example, you can’t give a transformer a very complicated question and expect it to get the answer in a single token. There’s just not enough time for it. These transformers need tokens to think, quote/ unquote, I like to say sometimes.

And so, this is some of the things that work well. You may, for example, have a few short prompt that shows the transformer that it should show its work when it’s answering a question. And if you give a few examples, the transformer will imitate that template, and it will just end up working out better in terms of its evaluation.

Additionally, you can elicit this kind of behavior from the transformer by saying, let’s think step by step, because this conditions the transformer into sort of showing its work. And because it kind of snaps into a mode of showing its work, it’s going to do less computational work per token. And so, it’s more likely to succeed as a result, because it’s making slower reasoning over time.

Here’s another example. This one is called self-consistency. We saw that we had the ability to start writing, and then it didn’t work out. I can try again, and I can try multiple times and maybe select the one that worked best. In these kinds of approaches, you may sample not just once, but you may sample multiple times, and then have some process for finding the ones that are good, and then keeping just those samples or doing a majority vote, or something like that. Basically, these transformers in the process as they predict the next token, just like you, they can get unlucky. And they could sample not a very good token, and they can go down sort of like a blind alley in terms of reasoning.

And so, unlike you, they cannot recover from that. They are stuck with every single token they sample. And so, they will continue the sequence, even if they even know that this sequence is not going to work out. Give them the ability to look back, inspect or try to find, try to basically sample around it.

Here’s one technique also. It turns out that, actually, LLMs, they know when they’ve screwed up. As an example, say you ask the model to generate a poem that does not rhyme, and it might give you a poem, but it actually rhymes. But it turns out that especially for the bigger models, like GPT-4, you can just ask it, did you meet the assignment? And actually, GPT-4 knows very well that it did not meet the assignment. It just kind of got unlucky in its sampling. And so, it will tell you, no, I didn’t actually meet the assignment. Here, let me try again.

But without you prompting it, it doesn’t even know. It doesn’t know to revisit, and so on. You have to make up for that in your prompts. You have to get it to check. If you don’t ask it to check, it’s not going to check by itself. It’s just a token simulator.

I think more generally, a lot of these techniques fall into the bucket of what I would say recreating our System 2. You might be familiar with the System 1, System 2 thinking for humans. System 1 is a fast, automatic process and I think kind of corresponds to an LLM just sampling tokens. And System 2 is the slower, deliberate planning sort of part of your brain.

And so, this is a paper actually from just last week, because the space is pretty quickly evolving. It’s called Tree of Thought, and in Tree of Thought, the authors of this paper proposed maintaining multiple completions for any given prompt. And then they are also scoring them along the way and keeping the ones that are going well, if that makes sense. And so, a lot of people are really playing around with kind of prompt engineering to basically bring back some of these abilities that we sort of have in our brain for LLMs.

Now, one thing I would like to note here is that this is not just a prompt. This is actually prompts that are, together, used with some Python glue code, because you actually have to maintain multiple prompts, and you also have to do some tree search algorithm here to like figure out which prompts to expand, etcetera. It’s a symbiosis of Python glue code and individual prompts that are called in a (wild?) loop or in a bigger algorithm.

I also think there’s a really cool parallel here to AlphaGo. AlphaGo has a policy for placing the next stone when it plays go, and this policy was trained originally by imitating humans. But in addition to this policy, it also does Monte-Carlo tree search. And basically, it will play out a number of possibilities in its head and evaluate all of them, and only keep the ones that work well. And so, I think this is kind of an equivalent of AlphaGo, but for text, if that makes sense.

Just like Tree of Thought, I think more generally, people are starting to really explore more general techniques of not just a simple question/answer prompts, but something that looks a lot more like Python glue code, stringing together many prompts.

On the right, I have an example from this paper called React, where they structure the answer to a prompt as a sequence of thought, action, observation, thought, action, observation. And it’s a full rollout, kind of a thinking process to answer the query. And in these actions, the model is also allowed to tool use.

On the left, I have an example of Auto GPT. And now, Auto GPT, by the way, is a project that I think got a lot of hype recently, but I think I still find it kind of inspirationally interesting. It’s a project that allows an LLM to sort of keep a task list and continue to recursively break down tasks. And I don’t think this currently works very well, and I would not advise people to use it in practical applications. I just think it’s something to generally take inspiration from in terms of where this is going, I think, over time.

That’s kind of like giving our model System 2 thinking.

The next thing that I find kind of interesting is this following sort of, I would say, almost psychological quirk of LLMs, is that LLMs don’t want to succeed. (Laughter.) They want to imitate. You want to succeed, and you should ask for it. (Laughter.) What I mean by that is when transformers are trained, they have training sets. And there can be an entire spectrum of performance qualities in their training data.

For example, there could be some kind of a prompt for some physics question or something like that, and there could be a student solution that is completely wrong, but there can also be an expert answer that is extremely right. And transformers can’t tell the difference between low – I mean, they know about low quality solutions and high quality solutions, but by default, they want to imitate all of it, because they’re just trained on language modeling. And so, at test time, you actually have to ask for a good performance.

In this example, in this paper, they tried various prompts, and let’s think step by step was very powerful, because it sort of like spread out the reasoning over many tokens. But what worked even better is, let’s work this out in a step by step way to be sure we have the right answer. And so, it’s kind of like a conditioning on getting a right answer. And this actually makes the transformer work better, because the transformer doesn’t have to now hedge its probability mass on low quality solutions, as ridiculous as that sounds.

And so, basically, feel free to ask for a strong solution. Say something like, you are a leading expert on this topic, pretend you have IQ 120, etcetera. But don’t try to ask for too much IQ because if you ask for IQ of like 400, you might be out of data distribution, or even worse, you could be in data distribution for some sci-fi stuff, and it will start to take on some sci-fi role playing or something like that. (Laughter.) You have to find like the right amount of IQ, I think. It’s got some U-shaped curve there.

Next up, as we saw, when we are trying to solve problems, we know what we are good at and what we’re not good at, and we lean on tools computationally. You want to do the same potentially with your LLMs. In particular, we may want to give them calculators, code interpreters and so on, the ability to do search, and there’s a lot of techniques for doing that.

One thing to keep in mind, again, is that these transformers, by default, may not know what they don’t know. You may even want to tell the transformer in the prompt, you are not very good at mental arithmetic. Whenever you need to do very large number addition, multiplication or whatever, instead, use this calculator. Here’s how you use the calculator. Use this token combination, etcetera, etcetera. You have to actually spell it out because the model, by default, doesn’t know what it’s good at or not good at, necessarily just like you and I might be.

Next up, I think something that is very interesting is we went from a world that was retrieval only. All the way, the pendulum has swung to the other extreme, where it’s memory only in LLMs. But actually, there’s this entire space in between of these retrieval augmented models, and this works extremely well in practice.

As I mentioned, the context window of a transformer is its working memory. If you can load the working memory with any information that is relevant to the task, the model will work extremely well, because it can immediately access all that memory. And so, I think a lot of people are really interested in basically retrieval augmented generation. And on the bottom, I have an example of LAMA index, which has one sort of data connector to lots of different types of data. And you can you can index all of that data, and you can make it accessible to LLMs.

And the emerging recipe there is you take relevant documents, you split them up into chunks, you embed all of them, and you basically get embedding vectors that represent that data. You store that in the vector store, and then at test time, you make some kind of a query to your vector store. And you fetch chunks that might be relevant to your task, and you stuff them into the prompt, and then you generate. This can work quite well in practice.

This is, I think, similar to when you and I solve problems. You can do everything from your memory, and transformers have very large and extensive memory, but also, it really helps to reference some primary documents. Whenever you find yourself going back to a textbook to find something or whenever you find yourself going back to the documentation of a library to look something up, the transformers definitely want to do that, too. You have some memory over how some documentation of a library works, but it’s much better to look it up. Same applies here.

Next, I wanted to briefly talk about constraint prompting. I also find this very interesting. This is basically techniques for enforcing a certain template in the outputs of LLMs. Guidance is one example from Microsoft, actually. And here we are, enforcing that the output from the LLM will be JSON. And this will actually guarantee that the output will take on this form, because they go in and they mess with the probabilities of all the different tokens that come out of the transformer, and they clamp those tokens. And then the transformer is only filling in the blanks here. And then you can enforce additional restrictions on what could go into those blanks.

This might be really helpful, and I think this kind of constraint sampling is also extremely interesting.

I also wanted to say a few words about finetuning. It is the case that you can get really far with prompt engineering, but it’s also possible to think about finetuning your models.

Now, finetuning models means that you are actually going to change the weights of the model. It is becoming a lot more accessible to do this in practice, and that’s because of a number of techniques that have been developed and have libraries for very recently.

For example, parameter efficient finetuning techniques like LoRA, make sure that you’re only training small, sparse pieces of your model. Most of the model is kept clamped at the base model and some pieces of it are allowed to change. And it still works pretty well, empirically, and makes it much cheaper to sort of tune only small pieces of your model. It also means that because most of your model is clamped, you can use very low precision inference for computing those parts, because they are not going to be updated by gradient descent. And so, that makes everything a lot more efficient as well.

And in addition, we have a number of open sourced, high quality based models currently, as I mentioned. And I think LAMA’s quite nice, although it is not commercially licensed, I believe, right now.

Something to keep in mind is that basically, finetuning is a lot more technically involved. It requires a lot more, I think, technical expertise to do right. It requires human data contractors for data sets and/or synthetic data pipelines that can be pretty complicated. This will definitely slow down your iteration cycle by a lot.

And I would say on a high level, SFT is achievable because you’re continuing the language modeling task. It’s relatively straightforward. But RLHF, I would say, is very much research territory and is even much harder to get to work. And so, I would probably not advise that someone just tries to roll their own RLHF implementation. These things are pretty unstable, very difficult to train, not something that is, I think, very beginner friendly right now. And it’s also potentially likely also to change pretty rapidly, still.

I think these are my sort of default recommendations right now. I would break up your task into two major parts. Number one, achieve your top performance, and number two, optimize your performance in that order.

Number one, the best performance will currently come from GPT-4 model. It is the most capable by far. Use prompts that are very detailed. They have lots of task context, relevant information and instructions. Think along the lines of what would you tell a task contractor if they can’t e-mail you back? But then also keep in mind that a task contractor is a human, and they have inner monologue, and they’re very clever, etcetera. LLMs do not possess those qualities, so make sure to think through the psychology of the LLM, almost, and cater prompts to that.

Retrieve and add any relevant context and information to these prompts, basically refer to a lot of the prompt engineering techniques. Some of them are highlighted in the slides above, but also this is a very large space, and I would just advise you to look for prompt engineering techniques online. There’s a lot to cover there.

Experiment with a few short examples. What this refers to is you don’t just want to tell, you want to show whenever it’s possible. Give it examples of everything that helps it really understand what you mean, if you can.

Experiment with tools and plugins to offload a task that are difficult for LLMs natively, and then think about not just a single prompt and answer. Think about potential change and reflection, and how you glue them together, and how you could potentially make multiple samples, and so on.

Finally, if you think you’ve squeezed out prompt engineering, which I think you should stick with for a while, look at some potentially finetuning a model to your application, but expect this to be a lot more slower and involved. And then there’s an expert fragile research zone here, and I would say that is RLHF, which currently does work a bit better than SFT, if you can get it to work. But again, this is pretty involved, I would say. And to optimize your costs, try to explore lower capacity models or shorter prompts and so on.

I also wanted to say a few words about the use cases in which I think LLMs are currently well-suited for. In particular, note that there’s a large number of limitations to LLMs today. And so, I would keep that definitely in mind for all your applications. And this is, by the way, could be an entire talk, so I don’t have time to cover it in full detail.

Models may be biased. They may fabricate, hallucinate information. They may have reasoning errors. They may struggle an entire classes of applications. They have knowledge cutoffs, so they might not know any information about, say, September 2021. They are susceptible to a large range of attacks, which are sort of like coming out on Twitter daily, including prompt injection, jailbreak attacks, data poisoning attacks and so on.

My recommendation right now is use LLMs in low stakes applications, combine them always with human oversight, use them as a source of inspiration and suggestions, and think copilots instead of completely autonomous agents that are just performing a task somewhere. It’s just not clear that the models are there right now.

I wanted to close by saying that GPT-4 is an amazing artifact. I’m very thankful that it exists, and it’s beautiful. It has a ton of knowledge across so many areas that can do math, code, and so on. And in addition, there’s this thriving ecosystem of everything else that is being built and incorporated into the ecosystem, some of these things I talked about. And all of this power is accessible at your fingertips.

Here’s everything that’s needed in terms of code to ask GPT-4 a question, to prompt it and get a response. In this case, I said, “Can you say something to inspire the audience of Microsoft Build 2023?” And I just punched this into Python, and verbatim, GPT-4 said the following.

And by the way, I did not know that they used this trick in the keynote, so I thought I was being clever. (Laughter.) But it is really good at this.

It says, “Ladies and gentlemen, innovators and trailblazers of Microsoft Build 2023, welcome to the gathering of brilliant minds like no other. You are architects of the future, the visionaries molding the digital realm in which humanity thrives. Embrace the limitless possibilities of technologies and let your ideas soar as high as your imagination. Together, let’s create a more connected, remarkable and inclusive world for generations to come. Get ready to unleash your creativity, canvass the unknown and turn dreams into reality. Your journey begins today.”

Thank you. (Applause.)

END