[MUSIC] ANNOUNCER:

Please welcome AI researcher and founding member of

OpenAI, Andrej Karpathy. 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. 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 Assistance, and then in the second part, we're going to take a look at how we can use these assistants effectively for your applications. 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 still rapidly evolving, but so far, the recipe looks something like this. Now, this is a complicated slide, I'm going to go

through it piece by piece, but roughly speaking, we have four major stages, pretraining, supervised finetuning, reward modeling, reinforcement learning, and 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 a objective
and over for

training the neural network, and then we have a resulting model, and then there are some notes on the bottom. The first stage

we're going to start with as the pretraining stage. Now, this stage is 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 percent

of the training compute time and also flops. 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 finetuning stages that are much more along the lines of small few number of GPUs and hours or days. Let's take a look at the pretraining stage to achieve a base model. First, we are 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 datasets that enter into these collections. We have CommonCrawl, which is a web scrape, C4, which is also CommonCrawl, and then some high quality datasets as well. For example, GitHub, Wikipedia, Books, Archives, 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 set for the GPT. Now before we can actually train on this data, we need to go through one more preprocessing step, and that is tokenization. This is basically a translation of the raw text that we scrape from the Internet into sequences of integers because that's the native representation over which GPTs function. Now, this is a

lossless translation between pieces of texts

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 text chunks and groups them into tokens. 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 examples for hybrid parameters that govern this stage. GPT-4, we did not release too much information about how it was trained and so on, I'm using GPT-3s 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. These are roughly the orders of magnitude that we're dealing with when we're doing pretraining. The vocabulary size is usually a couple 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 say, 65 billion for LLaMA. Now, even though LLaMA has only 65B parameters compared to GPP-3s 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 300 billion tokens. You shouldn't judge the power of a model by the number of parameters that it contains. Below, I'm showing

some tables of rough hyperparameters

that typically go into specifying the

transformer neural network, 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 a roughly several million dollars. 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 stocked up in rows and B by T, T being the maximum context length. In my picture I only have

10 the context lengths, so this could be

2,000, 4,000, etc. These are extremely long rows. What we do is we take these documents, and we pack them into rows, and we delimit them with these special end of texts tokens, basically telling

the transformer where a new document begins. Here, I have a few examples of documents and then I stretch them out into this input. Now, we're going to feed all of these numbers into transformer. Let me just focus on a single particular cell, but the same thing will happen at every cell in this diagram. 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 transforming neural network, and the transformer

is going to try to predict the next token in a 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 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. Of course, as I tune these parameters, you're getting slightly different predicted distributions for every single

one of these cells. 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. Basically, we have

a probability for whatever may follow. Now, in this specific example, for this specific cell, 513 will come next, and so we can use this as a source of supervision to update our transformers weights. 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. 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. Here's a small snippet of Shakespeare, and they train their GPT on it. Now, in the beginning, at initialization, the GPT starts with completely random weights. You're 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 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. 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. We're making more and more consistent predictions over time. These are the plots that you are looking at when you're doing model pretraining. Effectively, we're looking at the loss function over time as you train, and low loss means

that our transformer is giving a
higher probability to the next correct

integer in the sequence. What are we going to do with model once we've trained

it after a month? Well, the first thing that we noticed, we the field, is that these models 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 tasks you might be interested in. 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 NLP model for that, but the

new approach is: ignore sentiment classification, go off and do large language model pretraining, train a large transformer, and then you may only have a few examples and you can very

efficiently fine tune your model for that task. This works very well in practice. The reason for this

is that basically the transformer is forced to multitask a huge amount of tasks in the language modeling task, because 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. 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. These are language models and they want to complete documents, you can actually trick them into performing tasks by arranging

these fake documents. In this example, for example, we have some passage and then we like do QA, QA, QA. This is called Few-shot prompt, and then we do Q, and then as the transformer is tried to complete the document is actually answering our question. This is an example of prompt engineering based model, making it believe

that it's imitating a document and
getting

it to perform a task. 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. GPT-3 based model is available via the API under the name Devanshi and

GPT-2 based 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 want to complete documents. If you tell them to write a poem about the

bread and cheese, it will answer
questions

with more questions, it's 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. 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. The way you would do this is you would create a specific few-shot prompt that makes it look like there's some document between the human and assistant and they're exchanging information. Then at the bottom, you put your query at the

end and the base model will
condition itself into being a helpful

assistant and answer, but this is not very reliable and doesn't work super well in practice, although it can be done. Instead, we have a different path to make actual GPT assistants not base model document completers. That takes us into supervised finetuning. In the supervised finetuning stage, we are going to collect small but high quality data-sets, and in this case, we're going to ask human contractors to gather data of the form prompt and ideal response. We're going to

collect lots of these typically tens of thousands or something like that. Then we're going to still do language modeling on this data. Nothing changed algorithmically, we're swapping out a training set. It used to be

Internet documents, which has a high quantity local for basically Q8 prompt response data. That is low quantity,

high quality. We will still do

language modeling and then after training, we get an SFT model. 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. 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? Then the contractor also writes out an ideal response. When they write out these responses, they are following

extensive labeling documentations and

they are being asked to be helpful,

truthful, and harmless. These labeling

instructions here, you probably can't

read it, neither can I, but they're long and this is people following instructions and trying to complete these prompts. That's what the dataset looks like. You can train these models.

This works to some extent. Now, you can actually continue the pipeline from here on, and go into RLHF, reinforcement learning from human feedback that consists of both reward modeling and reinforcement learning. 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 SFT models. 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. Here's an example of what our dataset will look like. I have the same 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. Then what we do is we take the SFT model which we've already trained and we create multiple completions. In this case, we have three completions that the model has created, and then we ask people to rank these completions. 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. This can take people

even hours for a single prompt

completion pairs, but let's say we decided that one of these is much better than the others and so on. We rank them. Then we can follow that with something that looks

very much like a binary classification
on all the possible pairs

between these completions. What we do now is, we lay out our prompt in rows, and the prompt is identical across all three rows here. It's all the same prompt, but the completion of this varies. 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. The transformer will predict some reward for how good that completion is for that prompt and basically it makes a guess about the quality of each completion. 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. 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. That's how we train our reward model. 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. 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 basically the model we'd like to train which was initialized at 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 any more. Now the reward model tells us the quality of every single completion for all 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. Therefore, every single

token that we sampled in that second row is going to get a slightly higher probability for the future. We do this over and over on many prompts on many batches and basically, we get a policy that creates yellow tokens here. It's basically all the completions here will score high according to the reward model that we trained in the previous stage. That's what the

RLHF pipeline is. Then at the end, you get a model that you could deploy. As an example, ChatGPT is an RLHF model, but some other models

that you might come across for example, Vicuna-13B, and so on, these are SFT models. We have base models, SFT models, and RLHF models. That's the state of things there. Now why would you

want to do RLHF? One answer that's not that exciting is that it works better. This comes from the

instruct GPT paper. According to these

experiments a while ago now, these PPO models are RLHF. We see that they are basically preferred in a lot of comparisons when we give them to humans. Humans prefer basically tokens that come from RLHF models compared to SFT models, compared to base model that is prompted to be an assistant. It

just works better. But you might ask why does it work better? I don't think that there's a single amazing answer that the community has really agreed on, but I will offer one reason potentially. 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 paper clips. If you're a contractor trying to train data, then imagine being a contractor collecting basically data for the SFT stage, how are you supposed to create a nice haiku for a paper clip? You might 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.
Judging which one of these is

good is a much easier task. 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'd notice for example that they lose some entropy. That means that they give more peaky results. They can output samples with lower variation than the base model. The base model has

lots of entropy and will give lots of

diverse outputs. For example, one

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. Here is an example that I just cooked up. I want to generate cool Pokemon names. I gave it seven Pokemon names and I asked the base model to complete the document and it gave me a lot more

Pokemon names. These are fictitious.
I

tried to look them up. I don't believe they're actual Pokemons. This is the task that I think the base model would be good at because it still has lots of entropy. It'll give you lots of diverse cool more things that look like whatever you give it before. Having said all that, these are the assistant models that are probably available to you at this point. There was a team at Berkeley that ranked a lot of the available assistant models and give 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 Vicuna, Koala, etc. The first three rows here are all RLHF models and all of the other models to my knowledge, are SFT models, I believe. 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 setting of a concrete example. Let's work with a

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."

So 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. For this next step, let me blog on my blog, let me compare these two populations. First I'm going to

obviously 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 aware of what I know or don't

know of my self-knowledge. I go, 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 thing that I can do in my head and so therefore, I'm going to rely on the calculator so I'm going to use a calculator, punch it in and see that the output is roughly 53. Then maybe I do some reflection and sanity checks in my brain so does 53 makes sense? Well, that's quite a large fraction, but then California is the most populous state, so maybe that looks okay. Then I have all the information I might need, and now I get to the 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 like really awkward phrasing so let me actually delete that and let me try again. 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. These transformers are not very shallow networks they have about 80 layers of reasoning, but 80 is still not like too much. This transformer is going to do its best to imitate, but of course, the process here looks very different from the process that you took. In particular, in our final artifacts in the data sets that we create, and then eventually feed to LLMs, all that internal dialogue was 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. So, you can't expect it to 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 tried 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 are sample token sequences. They don't have separate inner monologue streams in their head right? They're evaluating what's happening. Now, they do have some 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. That's a lot of storage for a lot of facts. 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 perfect memory, but it's got a finite size, but the transformer has

a very direct access to it and so it can
a losslessly remember

anything that is inside its context window. This is 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 cognitive difference between these two 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. 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," I like to say sometimes. This is some of the things that work well, you may for example have a few-shot prompt that shows the transformer that it should show its work when it's answering 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 behavior from the transformer by saying, let things step-by-step. Because this conditions the

transformer into showing its work
and because it snaps into a mode

of showing its work, is going to do less computational work per token. 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

if 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 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 a not a very good token and they can go down like a blind alley in terms of reasoning. 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 know that this sequence is not going to work out. Give them the ability

to look back, inspect or try to
basically

sample around it. Here's one technique also, it turns out that actually LLMs, they know when they've screwed up, so 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?" Actually GPT-4 knows very well that it did not meet the assignment. It just got unlucky in its sampling. 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 know to revisit and so on. You have to make up for that in your prompts, and you have to get it to check, if you don't ask it to check, its 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 and System 2 thinking for humans. System 1 is a fast automatic process and I think corresponds to an

LLM just sampling tokens. System 2 is the slower deliberate planning part of your brain. This is a paper actually from just last week because this space is pretty quickly evolving, it's called 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. A lot of people

are really playing around with
prompt engineering to basically bring
back some of these abilities that we

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 figure out which prompts to expand, etc. It's a symbiosis of

Python Glue code and individual prompts that are called in a while 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 its policy was trained originally by imitating humans. But in addition to this policy, it also does Monte

Carlo Tree Search. 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. I think this is 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 the 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 and a thinking process to answer the query. In these actions, the model is also allowed to tool use. On the left, I have an example of AutoGPT. Now AutoGPT by the way is a project that I think got a lot of hype recently, but I think I still find it inspirationally interesting. It's a project that allows an LLM to keep the task list and continue to recursively break down tasks. 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 like giving our model System 2 thinking. The next thing I find interesting is, this following serve I would say almost psychological quirk of LLMs, is that LLMs don't

want to succeed, they want to imitate. You want to succeed, and you should ask for it. 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's solution that is completely wrong but there can also be an expert answer that is extremely right. Transformers can't tell the difference between low, 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. At test time, you actually have to ask for a good performance. In this example in this paper, they tried various prompts. Let's think step-by-step was very powerful because it 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. It's like conditioning on getting the 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. 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, etc. But don't try to ask for too much IQ because if you ask for IQ 400, you might be out of data distribution, or even worse, you could be in data distribution for something like

sci-fi stuff and it will start to take

on some sci-fi, or like roleplaying or

something like that. You have to find 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 a 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, you use this token

combination, etc. 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 its 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. I think a lot of people

are really interested in basically
retrieval-augment

degeneration. On the bottom, I have an

example of LlamaIndex which is one data connector to lots of different types of data. You can index all of that data and you can make it accessible to LLMs. 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 documentation of the library to look something up, transformers definitely want to do that too. You have some memory over how some documentation of the library works but it's much

better to look it up. The same applies here. Next, I wanted to briefly talk about constraint prompting. I also find this very interesting. This is basically techniques for forcing a certain template in the outputs of LLMs. Guidance is one example from Microsoft actually. Here we are enforcing that the output from the

LLM will be JSON. 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 constraint sampling is also extremely interesting. I also want to say a few words about fine tuning. It is the case that you can get really far with prompt engineering, but it's also possible to think about fine tuning your models. Now, fine tuning

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. So for example

parameter efficient fine tuning techniques

like Laura, make sure that you're

only training small, sparse pieces of your model. So most of the model

is kept clamped at the base model
and some

pieces of it are allowed to change
and this

still works pretty well empirically and makes it much cheaper to 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 you 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 source, high-quality base models. Currently, as I mentioned, I think LLaMa is quite nice, although it is not commercially licensed, I believe right now. Some things to keep in mind is that basically fine tuning is a lot more technically involved. It requires a lot more, I think, technical expertise to do right. It requires human data contractors for datasets 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 of 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. So I think these are my default recommendations right now. I would break up your task

into two major parts. Number 1, achieve

your top performance, and Number 2, optimize your performance in that order. Number 1, the best performance will currently come from GPT-4 model. It is the most capable of all by far. Use prompts that

are very detailed. They have lots of

task content, relevant information

and instructions. Think along the lines

of what would you tell a task contractor if they can't email 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, etc. 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 I've 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 few-shot examples. What this refers to is, you don't just want to tell, you want to show whenever it's possible. So give it examples of everything that helps it really understand

what you mean if you can.
Experiment with tools

and plug-ins to offload tasks that are

difficult for LLMs natively, and then think about not just a single prompt and answer, think about potential chains and reflection and how you glue them together and how you can 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 fine tuning a model to your application, but expect this to be a lot more slower in the vault 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 of your applications. Models, and this 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 in entire classes of applications, they have knowledge cut-offs, so they might not know any information above, say, September, 2021. They are susceptible to a large range of attacks which are coming out on Twitter daily, including prompt injection, jailbreak attacks, data poisoning

attacks and so on. So 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 co-pilots, instead of completely

autonomous agents that are just like

performing a task somewhere. It's just not clear that the models are there right now. So 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. It 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've talked about, and all of this power is accessible at your fingertips. So 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, but it is really good at this. It says, ladies and gentlemen, innovators and trailblazers

Microsoft Build 2023. Welcome to the

gathering of brilliant minds like no other, you are the 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, canvas the unknown, and turn dreams into reality. Your journey begins today!