Introduction to Large Language Models (LLMs)

Prof. Tanmoy Chakraborty, Prof. Soumen Chakraborti
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi

Lecture 29

Parameter Efficient Fine-Tuning (PEFT)

I'm Dinesh Raghu. I'm a senior researcher at IBM Research. I lead the conversational AI team there. And our team mainly works in the intersection of large language models and conversational AI. So today I'll be talking about how can we fine tune large language models in an efficient manner.

Specifically, I'll be discussing the parametric efficient fine tuning in large language models. Right. So before we jump into the techniques, let's first like just look back to see how things were before the large language models came into picture. So, typically this was a transfer learning era where when you have a large unlabeled corpora at your disposal, you use them to sort of get a sense of the world knowledge by either learning word representations or contextual word representations or sentence representations. and then use them and transfer this knowledge to a task specific case.

Now, so hence it had two phases. There was a pre-training phase where you just use the unlabeled data and you develop some world knowledge and then you have the fine tuning stage where you use, you transfer the world knowledge that you had using pre-training but then use your task-specific data and fully fine-tune the model. And there were different ways of fine-tuning them. Fully fine-tuning was one such technique. Or you can augment the pre-trained model with additional layers and then fine-tune them as well.

So there were multiple ways in which people fine-tuned. Now what changed during the LLM era is that the models became more aware and were better capable of how they model the world knowledge. So, initially there was just pre-training phase. But now, we have instruction fine tuning and alignment phase which I hope it has already been covered in

this class. So, there is slight change in how we leverage the unlabeled data, but the instruction tuning and alignment does come with some label data, but it is general purpose.

Again, it teaches the model world knowledge, it is not teaching it any specific task. Now once we have a model this way, then the first way in which people started leveraging such a large language model was to simply do in context learning. So, by in context learning what we mean is that whatever knowledge that you want to feed the model about the task you want to solve, it sort of fed into the input prompt. Sometimes, people specify detailed set of instructions of how to solve the task. Sometimes, people feed instructions along with some examples of how the input output pairs so that the model learns them better.

So, this was the beginning of a new era where we were able to solve like good number of NLP tasks by simply using in context learning. Now the reason why this is really useful is because hosting or serving LLM is very expensive and it requires a lot of hardware and compute, right? So not everybody would be able to, who wants to leverage LLMs would be able to host it. So if it has been hosted in a certain, by a certain provider, then it's easy for people to make it useful for their task by simply doing in context learning by calling them through APX, right? So this was the main advantage of how transfer learning work in LLM era. Now, so let's see like why aren't people happy with just in context learning, right? So given that LLM knows a lot more about the world, why are we not happy with just in context learning? So firstly, relative to what people did earlier, which was full fine tuning, prompting was not that great. So if the task is very critical and you need really good accuracies, then prompting typically does not get you there.

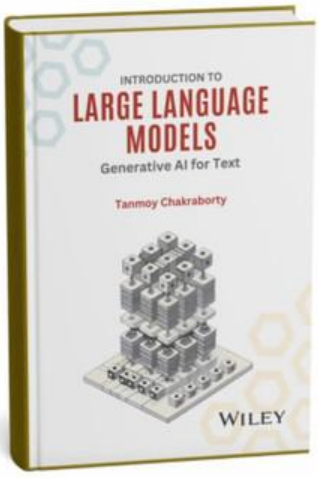
Introduction to

Large Language Models

Lecture # 29

Parameter Efficient Fine-Tuning (PEFT)

National Programme on Technology Enhanced Learning



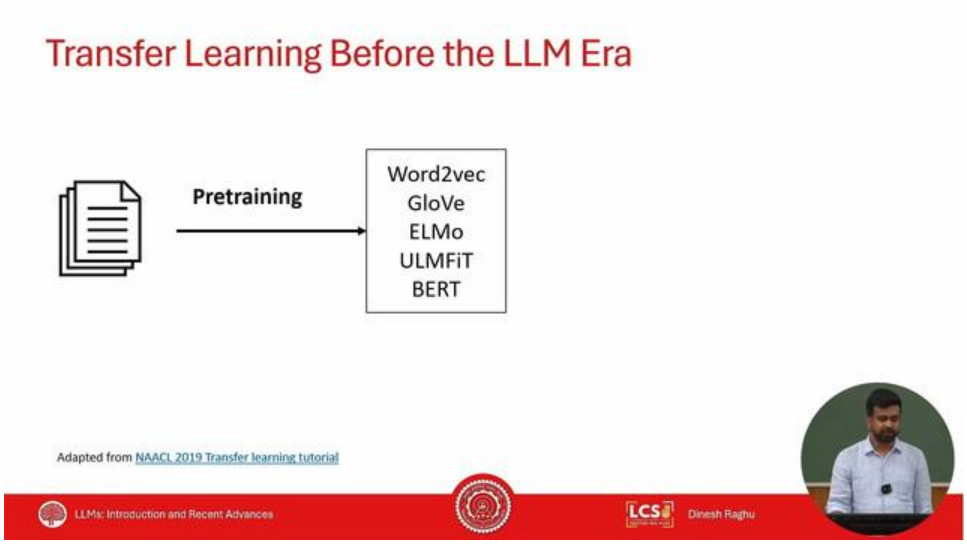
Reference Book

INTRODUCTION TO LARGE LANGUAGE MODELS

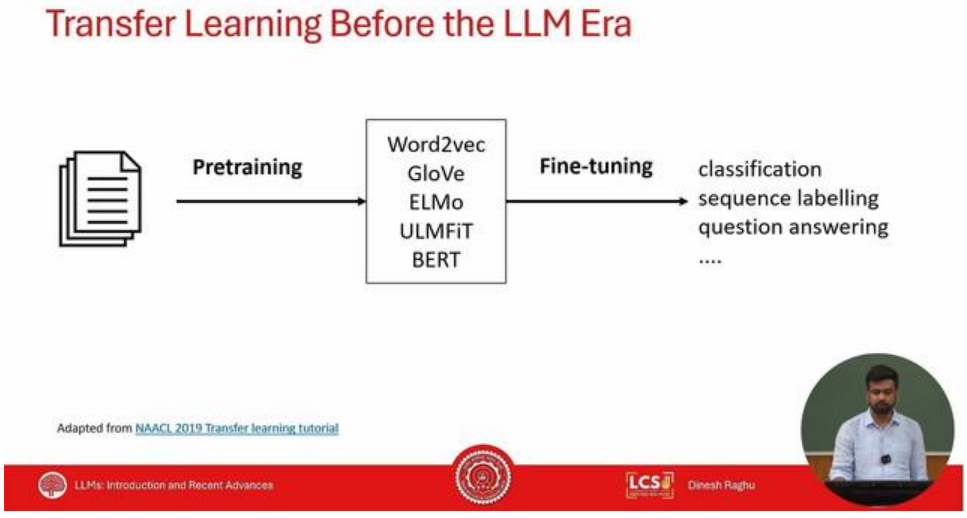
Book Chapter #10

Efficient Methods for Fine-Tuning

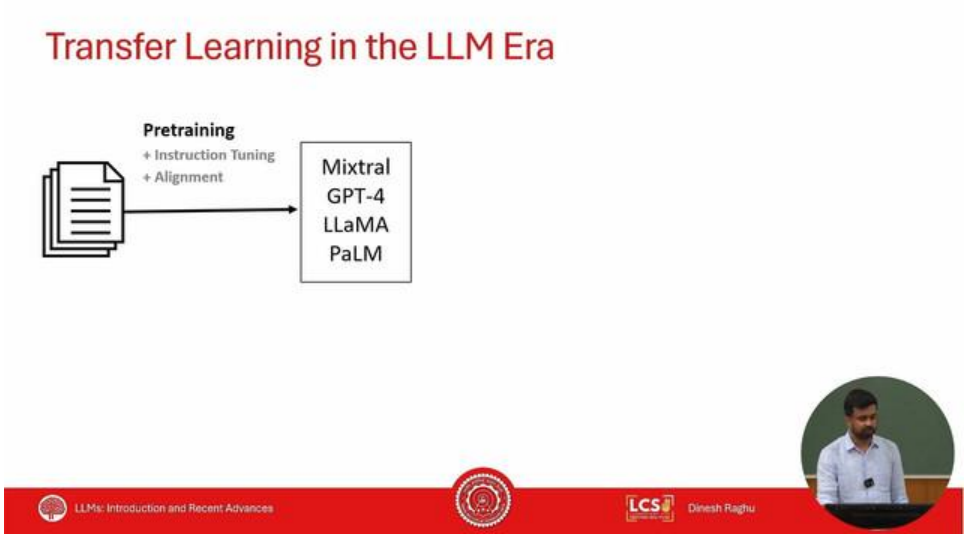
Section 10.3 (Parameter-Efficient Fine-Tuning)



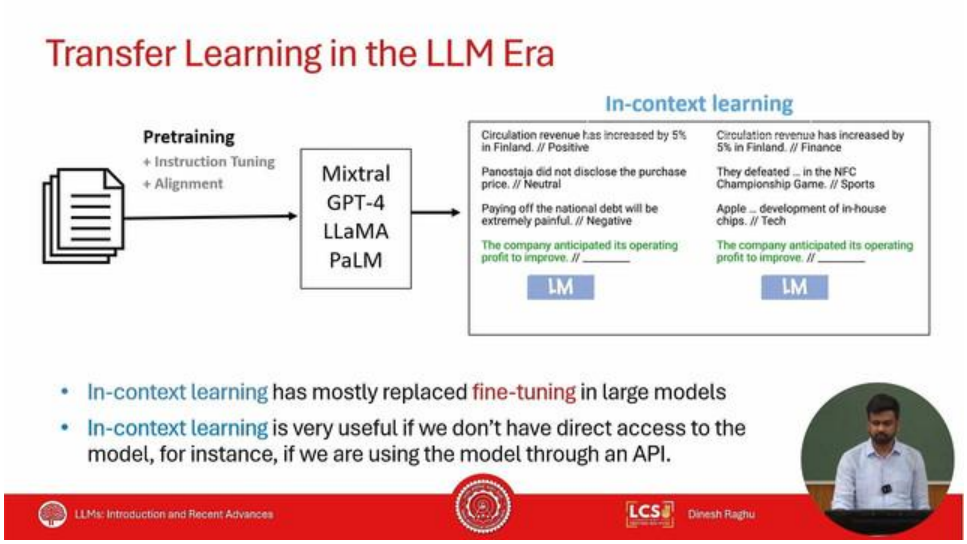
You will have to do full fine tuning of the models that you have to get to the best performance. But why does prompting have poor performance? So, there are two main reasons. One is that there are certain things in the prompt engineer's control and two is there are some things that the LLMs assume for themselves. So, if the prompt engineer does not do a very diligent job of defining the task then it it is one reason why it would lead to poor performance but then there are certain academic benchmarks where a lot of researchers try and they still are not able to get good numbers as good as a fully fine-tuned model which is much smaller than a big model right. So the reason for that is that when the large language model learns There are certain assumptions it makes and there are only certain things that it can comprehend.

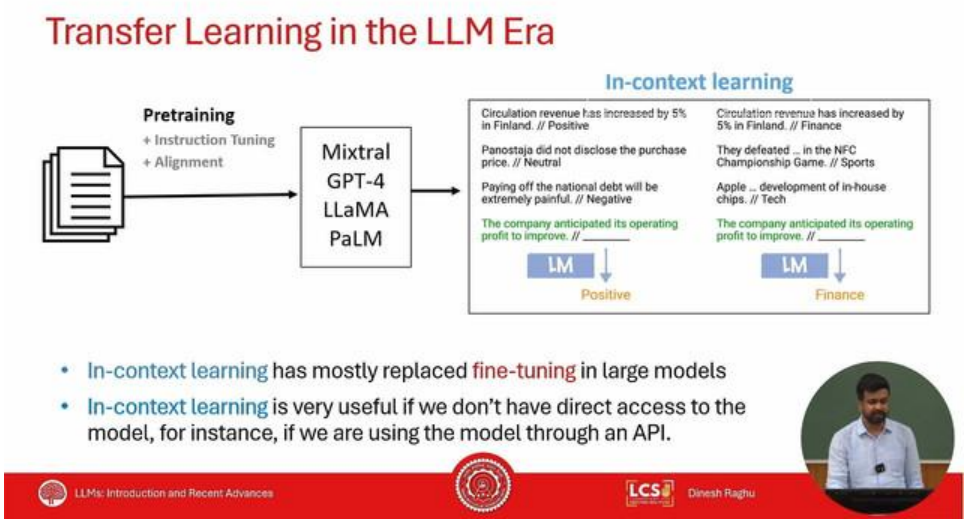


While we know it is good, but it's hard to know what really goes on in the back. So, when certain things don't work, it could be because the model has not learned some aspect of certain domains or it could be that there are certain assumptions that the model is making that we are not able to figure out what those assumptions are. So, there are a lot of reasons why prompting can take you only to a certain extent. Now the second problem with prompt engineering or simply in context learning is that these prompts are extremely sensitive. Earlier when the LLM started even missing a simple proposition would result in very different performance.



But even when you have a really good models and typically when your input output pairs when your input is pretty long or your output is pretty long, the order in which you put them in the in-context learning also affects the accuracy that you get. So again, why is this a big problem? Because maybe if you're really concerned about the task or if you're really particular about it, then you can always dedicate people to sort of come up with the right way to make everything work but the problem is once let's say like few months back lama 3.1 came which was much more capable of than lama 3 right. Now you'll have to do everything from scratch because the assumptions that are made by lama 3 and lama 3.1 may not be the same.

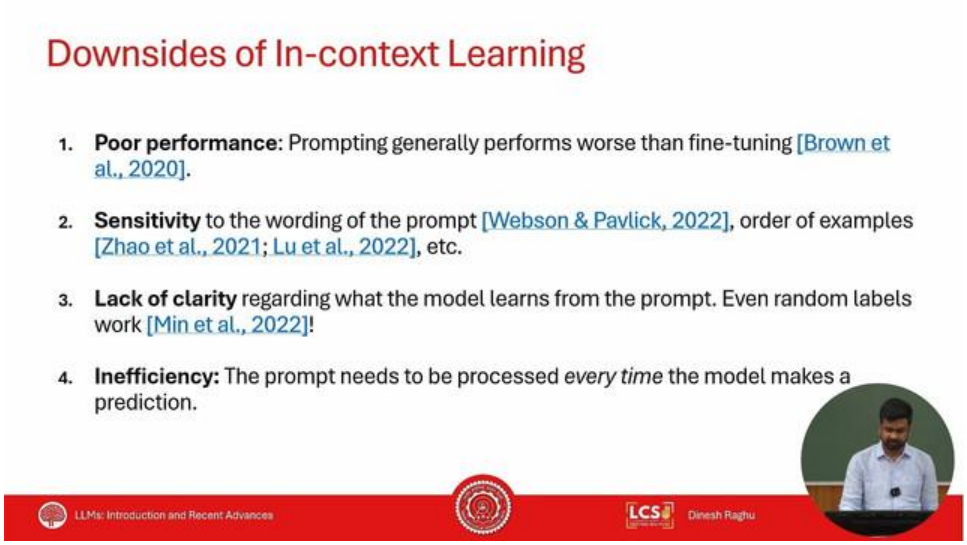




The knowledge that went into training may not be the same. So every time a new model comes then you'll have to do everything from scratch and whatever you did earlier may or may not hold like most likely it doesn't work And thirdly, there is always an issue about what the model really infers from the prompt you give it. This also happens with any machine learning algorithm, when you have really deep networks you really don't know what the model learns from the data you provide. But here there is a two hop thing, right? You are providing certain instructions and very small subset of examples. And the model also has a lot of knowledge inherently built in with it.

So there may be a big gap between what you say and what the model can understand. and then there can also be a gap between like what it learned from the data that you provide right. So there are a lot of variability and people have also found that like even providing weird in context examples works for certain tasks right which makes no sense because ideally when you provide prompts in a very unrealistic manner the model should not work. But still the model learns to pick up, ignore those. So, we don't know whether it's ignoring those signals, we don't know whether it's learning something spurious from that signal, we don't know when the model will fail and so on.

So, this lack of clarity makes it difficult for people to trust models and then use them right away. So far the first three items that i talked about are more from like the linguistic standpoint or what the model can comprehend and what it cannot right but the last part is more system right it's just that if you use up a lot of your prompt space for just giving instructions and in context examples then it simply increases the, It increases the amount, the latency that you get with your model and also resources and throughput and energy and so on. And specifically, most of the real-world use cases are not going to be like input-output pairs of let's say 20 tokens or 25 tokens. For example, if you want to do reading comprehension or summarization, it's going to span too many tokens. And if you have a very specific way of summarization, let's say news article summarization, maybe you want to summarize scientific news in a certain way, maybe you want to summarize opinion piece in a certain way maybe you want to summarize sports related events in certain way.

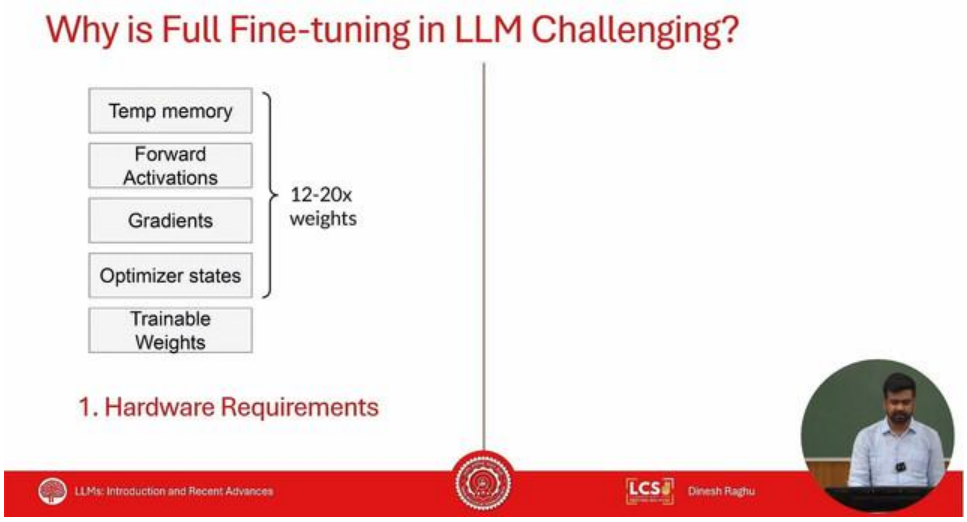


Now you will have to provide all these instructions in a detailed manner you have to provide in context examples of all these things so this would take a considerable amount of space and that would result in like inefficiency in how you use the hardware and stuff right So, these are the downside of in-context learning. So, now let us see like if in-context learning has these many problems, then let us start doing full fine-tuning because this has been what people have been doing for really long time now. So, why cannot we do fully fine-tuning? So, let us say that you want to fine-tune LAMA 8 billion model. Now it might end up taking 16 gigs of memory if it is in FP16, if all the weights are in FP16. But when you fine tune a model, it is not just the amount of parameters that a model estim determines how much memory it needs but rather it also needs space for optimizer states storing gradients storing forward activations during your forward step and also a lot of temporary memory here and there and this sort of so there are different ways in which you train the model there are like full precision training mixed precision training low precision training so like if you consider multiple all these different types of training then you end up needing around 12 to 20 percentage of trainable parameters, that is the amount of memory you need.

So, let us say you want to fine tune a 175 billion GPT-3 model, it is going to take like insane amount of memory and not everybody who wants to use LLMs or has a capability to host an LLM would have enough infrastructure to fine tune an LLM, fully fine tune an LLM. So, then it will become difficult for research to progress ahead. And also like in general accessibility also reduces a lot right. So this is one of the main reasons why people don't want to do full fine tuning of large language models and the other main reason is storage. So let's say your checkpoint is 350 gigabytes.

Every time you have a small task to learn, then it becomes really tedious to save a checkpoint for each task you have and it becomes even more tedious to serve them. So now if you had to serve only one LLM in an in-context learning setup, that serves all tasks, then you just need one copy of it. You may need, let's say, you need eight GPUs in a node, and then you can serve them. But now, if you have, let's say, five or six tasks, and specifically if you are working in a company that infuses LLM in a lot of applications they have, then we can't host 10, 15 LLMs. And without leveraging the entire throughput right because if it's not efficient then again it's a waste of energy and again it's expensive.

So these two are the main reasons why people don't want to do full fine tuning but there are other reasons as well right. So one other reason which I've not listed here, which is also one of the common reasons, like even if you have enough hardware, let's say you have an insane amount of hardware within your organization and you don't have to worry about storage, then of course, if you have a lot of hardware, then you can host how many LLMs you want. But the problem is how much training data do you need so that the model doesn't overfit? If you have 175 billion parameter model and you just have let us say 1000 or 2000 examples, the model is going to completely memorize all the 2000 examples. And once it memorizes 2000 examples, your training accuracy will be really high and your validation may not be that great and you will encounter this problem. The whole notion of transfer learning gets lost.

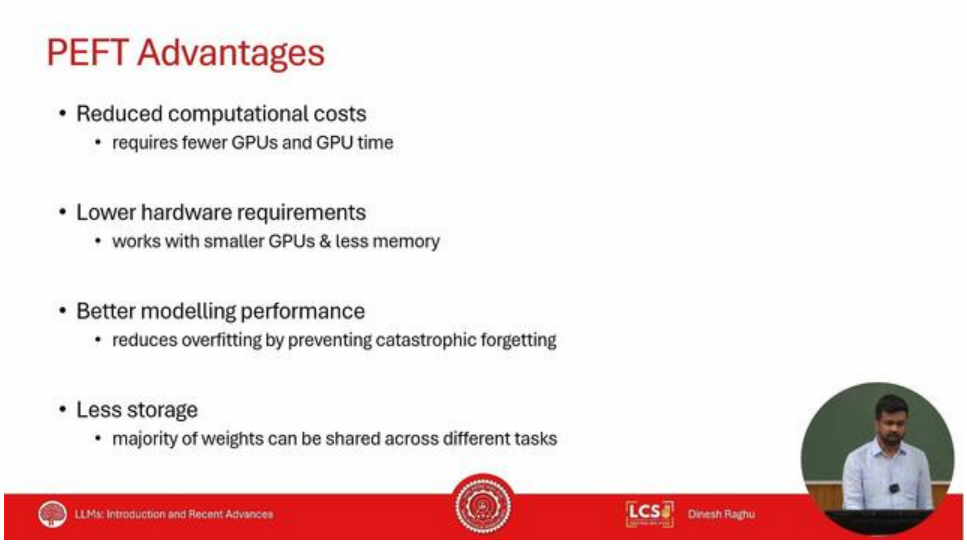


That was the motive behind fully fine tuning. So then imagine the amount of data you might need to actually get a model that doesn't overfit. That's going to be insane and it's going to be really hard to collect that much data. So that is also one other reason that people would want to refrain from doing fully fine tune. And now comes the parameter efficient fine tuning.

So now on one end you have in context learning, on the other end you have fully fine tuning. Now people are trying to figure out what if I freeze most of the parameters in my network and only train a very small subset. So that I don't face all the issues that I face with

fully fine tuning. And at the same time, I can mitigate some of these issues that I face within context learning. So that is the whole crux of parameter efficient fine tuning.

So here, as you see, if you have to train a QA model on top of LLM using PEFT, then you would get a very incremental number of parameters compared to what the llm already has and that would represent everything that's needed to learn the task and same thing goes for summarization and same thing goes for classification and so this is the crux of peft and So now the advantages are going to be like we're going to borrow some disadvantages from in-context learning and fully fine tuning and sort of put it here. So of course, like I mentioned, the first issue with fully fine tuning was the hardware requirements. So you require 12 to 20 times the number of trainable parameters the memory of that amount to sort of support training. So, now if you reduce the number of trainable parameters, of course your optimizer states are going to come down, the gradients are going to come down, activations may remain there but lot of these temp usage may come down and this reduces the amount of memory that you need drastically. So, which means you can use fewer GPUs to train your model and one other thing is that since you're working at a lower parameter space and most of your world model is frozen, the model learns to sort of map world knowledge to your task and then converge faster.



So, this actually helps the model. This also saves in GPU time, like you learn models faster than what you would learn by fully fine tuning. Of course, the obvious next thing is that hardware requirement is lower, which means not only less memory, but also many

institutions and organizations who started using GPUs during the deep learning era and now are slowly moving towards the LLM era, they can still work with V100 tesla models or they don't necessarily have to have the a100s with 80 gigs or the h100s to sort of fine tune. They can work with older versions of gpus with less memory and they can still get a lot of work done right. So that is a very big advantage because a lot of these older machines which consume less power are idle right and this is one way of effectively using them.

And of course, like I said earlier, one of the issues with fully fine tuning is that you have such large parametric space to leverage during your learning process, you end up overfitting and memorizing all the training examples. So now, if you don't have so much capacity to begin with, then the likelihood of overfitting reduces a lot. Again, I am not saying that overfitting goes away completely. It depends on how many parameters you let the model sort of use during training. But again, the likelihood of overfitting reduces considerably.

Now, in addition to that, there is this phenomenon called catastrophic forgetting. So, catastrophic forgetting is a process in general to any machine learning model. Let us say that you initially started to teach the model task a right. So now the model has learned task a very well and now you want to teach the model task b right. So now how good would the model be in task B? Let's say you're working in a pre-LLM era.

Let's say you're fine-tuning BERT. You initially fine-tuned it for classifying, let's say, finance versus tech classification, and now you want to teach it spam detection or something, right? So how many people believe that spam detection, if the spam detection was the task B, how many people believe that task A would still be doing good after you teach the models task B? So, it's not going to learn. So, imagine this with large language models, it knows the world knowledge. You have taught it so much during the pre-training, instruction tuning and alignment stage. Now, you teach it a very simple task and it's going to forget everything that it gathered so far.

You don't want that to happen. So, most of the PEF techniques are good in this type of transfer learning. They don't typically forget the world knowledge, which means that if you teach a task with a certain domain, it is very likely to generalize to a new domain. If you

teach spam detection in say IIT Delhi, now if you move to US where maybe the spammers are different or the type of language people use is different, it is very likely going to work. It's not going to fail.

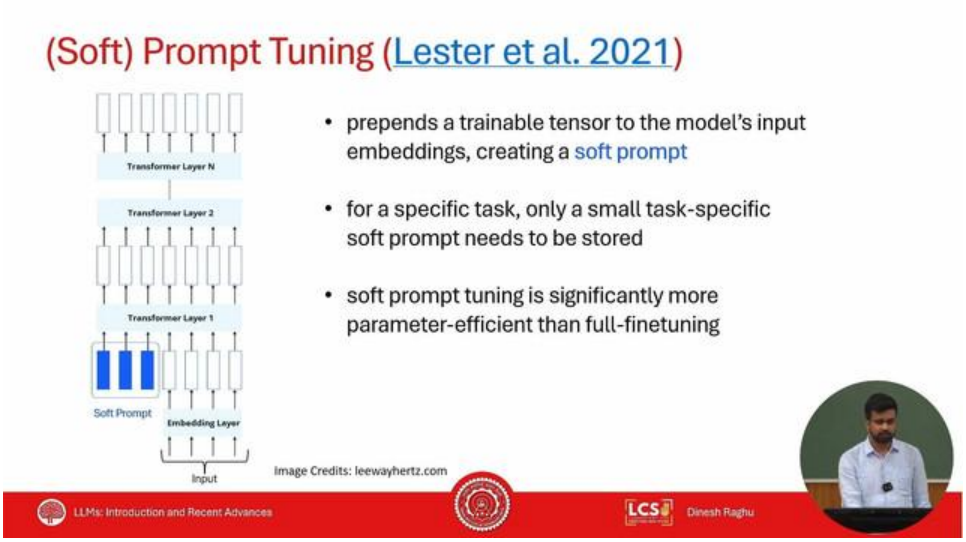
And finally, of course, the obvious thing is that it requires less storage because the number of parameters that you store are less. So obviously, you only have to save the incremental weights that you've learned and you don't have to store the entire copy of LLMK. So it's going to be, it needs just very little storage. Now with these advantages and the context of why we need PEFT and why it is useful, let us dive deep into a few popular techniques that people use for parameter efficient fine tuning. So, first let us talk about the prompt tuning or soft prompting.

This was not one of the first test techniques, but I'm starting with this because this is one of the easiest things and we can build upon it in the remaining of the lecture. So typically when you do in-context learning, we refer to it as hard prompting because you write the prompt in the in-context setup. And if you don't like the prompt, then you change certain words or you rephrase certain things or you add examples. So, this way of modifying the prompt or the in-context setup is hard prompting because humans are involved in doing this. So, what Lester et al thought was that If we have a large amount of input and output available with us and it's easy for us to measure how good or bad the output is, then wouldn't it be easier for the model to learn this prompt by itself? So all they say is that you reserve certain tokens in your input for certain special tokens called the soft prompt.

And the only trainable parameters in your model is this task specific prompt. Everything else in the architecture remains the same you don't change any other weights everything is frozen you only change these new tokens that you want to call it as soft prompts right and this is what their idea was and the good part about this idea is that for a specific task after you complete the learning process all you need to do is save that soft prompt right which is going to be like orders of magnitude lower than what the LLMs would do. So let's say if your tokenizer, so your embedding size is let's say 4k or something. If you use five tokens, this is very marginal compared to the size of LLMs that we have now. So that is now like we know that given the number of trainable parameters is so low, it's one of the most efficient way of fine tuning compared to full fine tuning.

Now there is also another really good advantage of soft prompting, which is the multitask serving. Let's say you're hosting an LLM and you let people fine tune for various tasks. And let's say that you have 120 tasks in your organization or for your project. During training you would have data sets for each of your tasks that you want to learn. And so you will assign certain special tokens for each of your tasks and then you will start your training process right.

So for example for task a you end up training the yellow tokens and for task b you end up training the green tokens and your llm weights remain frozen right and now the good part is during inference it's so easy right. You just need to host the base llm and when you create batches to feed to your llm to predict the output you can just batch in whatever task you want based on what traffic comes in right. So this is extremely efficient for serving because it may not so happen that the traffic for your llms are going to be that high right, if you have 120 tasks. If it's high, then of course you have to have separate, you can have a lot of instances of LLM running, but it's easy to scale up, scale down because you just need the same LLM again. So if the traffic goes down, you can scale down certain hosting and then you can keep running them for multiple tasks at the same time.



So this is one of the biggest advantages of prompt tuning. And this is why most LLM hosting companies, they provide you an API to prompt you. Because hosting is so cheap for them like even if you because they charge you per request right they don't charge you per time. So their gpus are anyway running. So it's going to be more efficient for them if they support this type of prompt tuning.

So let's look at how does it compare with full fine tuning right. So the graph in red or orange is a full fine tuning. And here the graph in green is prompt tuning and the graph in blue is simple prompt engineering, hard prompting. And the x-axis here is a number of parameters in the model that you are trying to fine tune. So, right from 10 power 8 to 10 power 10 are the family of T5 models.

So, they have different sizes of T5 models that Microsoft had trained and highest one is like 11 billion model. And the lowest one is around 700 million model or something, 770 or something. And if you see, there is an additional point on the right for blue, which is prompt engineering. And that point, the last point, which is about 10 power, little more than 10 power 11 is GPT-3. So that's GPT-3, when this paper came out, that was the best model possible for prompt engineering at that point in time.

So what they see here is that when you have smaller models, then it's better to do full fine tuning because prompt tuning or prompt engineering does not give you as much accuracy as what the full fine tuning gives. But as and when you keep increasing the model size then

prompt prompt tuning sort of catches up with full fine tuning right. So again if you see irrespective in respect of how much you do hard prompting there is a huge gap even for the 11 billion model. So hard prompting is nothing but let's say that you want to do classification of spam or not spam right. So now if i give you access to gpt40 in the web ui what would you do? So you would give some instructions and then you would give some examples of what is spam and what is not spam right some input and output right.

So this is called hard prompting. Like let's say if this didn't work this didn't give you 95 accuracy then what would you do? Yes you would change the prompt right. So that's a tedious process right. So that this style of creating a solution is what is called hard prompting. So soft prompting is prompt tuning, which is like the model learns it by itself, given a small amount of data.

Okay, so now one of the other findings in this paper is that, which is also slightly obvious, is that if you increase the prompt length, then it improves the performance on the task. So, this is same as saying like if you give a very elaborate instruction or if you increase the number of tokens you use for in context learning, it is very likely the model would pick up lot more than what it would if you give it only one or two words to describe the task. So, the essence is that way, but the interesting thing they found is that if the number of prompt tokens cross 20 for T5, then they don't see any more improvements. So it sort of peaks at around 20. So now one other thing is that so initializing what should be the weights of these soft prompts to begin with is actually very important.

If you give it very random initialization, it may not converge to a good point or it may even take a very long time to converge. But if you sort of say, create a prompt using words that are very close to the task description and then take its embeddings and then post them there or you take certain words that are very related to the task and then post them there, then that is better. I think that's also an insight that this paper gives. Now, one thing that I mentioned in the advantages of PEFT is that it sort of generalizes well to out-domain data. So here, they test on the task of reading comprehension.

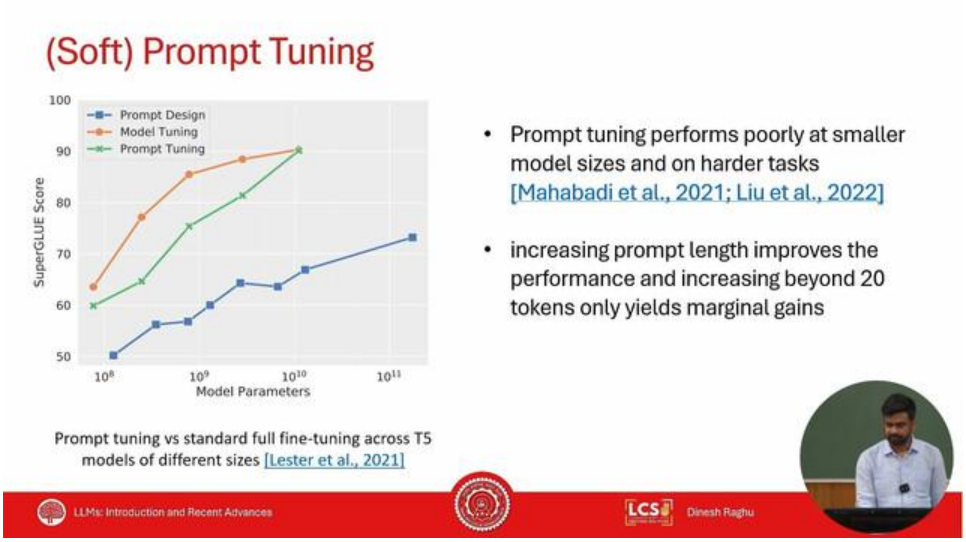
Specifically, they take the SQuAD dataset where there is a natural language text which describes something, and then there is a question-answer following that. And this is based

on Wikipedia. And here, the model is a full fine tuning, prompt is a prompt tuning. So there is hardly any difference between full fine tuning and prompt tuning on the same data set that you train the model on. But if you see the model's performance on the books data set, which is like very different domain than the Wikipedia.

You see that the model sort of generalizes much better the prompt tuning model whereas the full fine tuning model does not generalize that well. So they show a bunch of different domains some are close to wikipedia some are slightly far away and and one of the biggest takeaways here is that the 12.5 improvement on books was actually a very good insight right OK, so now that brings us to our next topic, which is prefix tuning. So the number of parameters you use for the PEF technique has trade-off. If you use a lot of parameters, then you are very likely to learn very complex tasks because your capacity is very high, but your chances of overfitting also becomes high.

But if you have reduced number of parameters, then your chances of overfitting is less, but your performance can get affected. So you need some family of algorithms. And then there's only so much you can do with giving tokens. Because if you give 20 tokens and it still doesn't learn, there is going to be very less likelihood that it's going to get better. So we can't just stop with prompt tuning.

So then people came up with prefix tuning. So prefix tuning is actually a contemporary work to prompt tuning. Both of them got published around the same time. Now, the main difference between prefix tuning and prompt tuning is that prompt tuning gives you those trainable tokens only in the input layer. Basically, whatever the embedding layer gives out is where you feed in the special prompts, the embeddings of the special prompts, that's the only trainable parameter.



Whereas in prefix tuning, you have a small set of trainable parameters in every layer of your transformer. So, now if you look on the right, So here this is the regular transformer block where you have the multi-head self-attention, fully connected layer and then there is a residual connection, layer norm and then there is again a fully connected network with a residual connection finally followed by a layer norm. So, this is what your regular transformer block looks like but with prefix tuning. So, for now assume this whole thing is a single prefix. I will tell you why we have this particular architecture here, but assume this to have a single prefix.

Now if you look at a decoder only model, so basically prefix comes in the beginning. So, prefix do not influence each other, but every word that you feed in the input get influenced by your prefix in every layer. So, that is the sort of crux of this approach. So, let us see why they have this soft prompt and a fully connected layer here. Why do we need that? We could have just used something similar to soft prompts that we used.

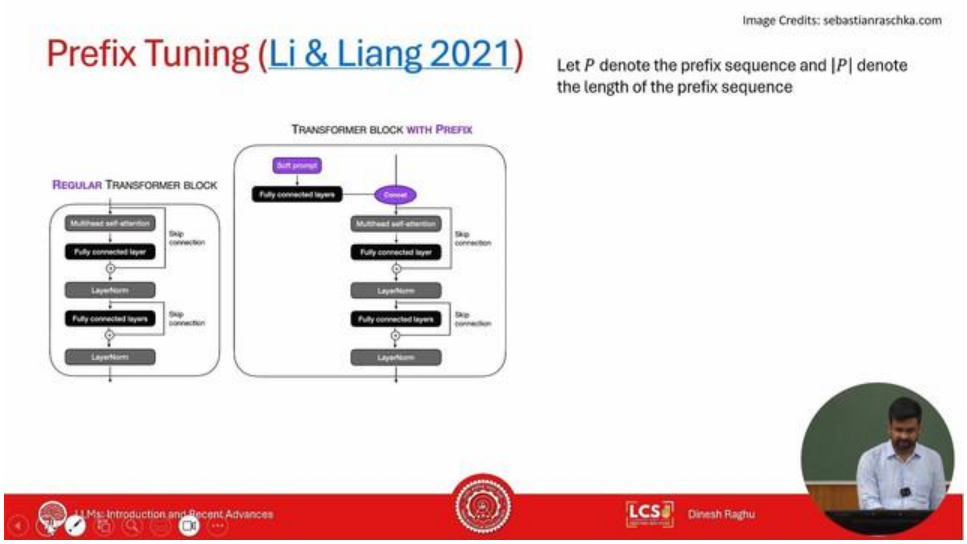
So, why do we need this layer? So let's say that we have a prompt sequence of, let's say, five tokens or something. So let's call it p, which is all the five tokens in sequence, p1, p2, p3, p4, p5. And let mod p denote the number of tokens in that prefix. So ideally, what do we want? We want a function that converts that special token into certain embedding, which can then directly feed into the network. And the f theta is a trainable parameter.

So now f theta's dimensions have to be that it's sort of like an embedding lookup. For each prefix token you have, you need an embedding associated with it. And this embedding, since you're feeding it into the hidden layer, the embedding should be of the size of what the hidden layers that you have. So what the authors found is that if you use this particular f theta to parameterize your prefixes, then it's resulting in unstable training. So what we mean by unstable training is that let's say the weights may get changed so high that it sort of breaks the whole training process.

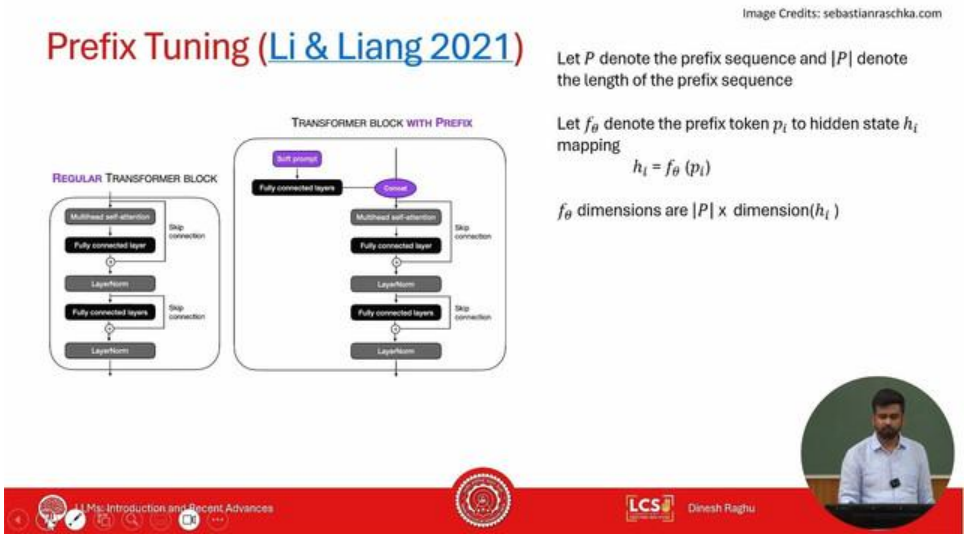
Your gradients hit the max or it hits zero or it goes too low. So, this results in unstable training. So, this happens because sometimes that you are trying to learn something with very small number of parameters and you did not put it in the right place where maybe there are certain reasons why we have layer norm. So, you have to keep these in the right place so that you don't explore things or you don't underflow things. So if you don't keep, if you don't architect your solution in the right manner, you will end up facing these training issues.

So you'll have to change code, have some hacks or change your architecture in such a way so that you don't undergo this unstable training process. Because this is a very risky thing to have in your network because let's say after you train for seven days using a GPU farm of 100 different nodes and then you suddenly encounter an unstable training, then whatever you did for so long is a waste. This is something that we should always avoid when you design. And what these guys figured out is that the number of parameters that they have and in the way they have designed it is what is resulting in unstable training. So, what they did is that they represented f theta with two functions, basically f theta prime, which has much smaller dimension than the hidden dimension.

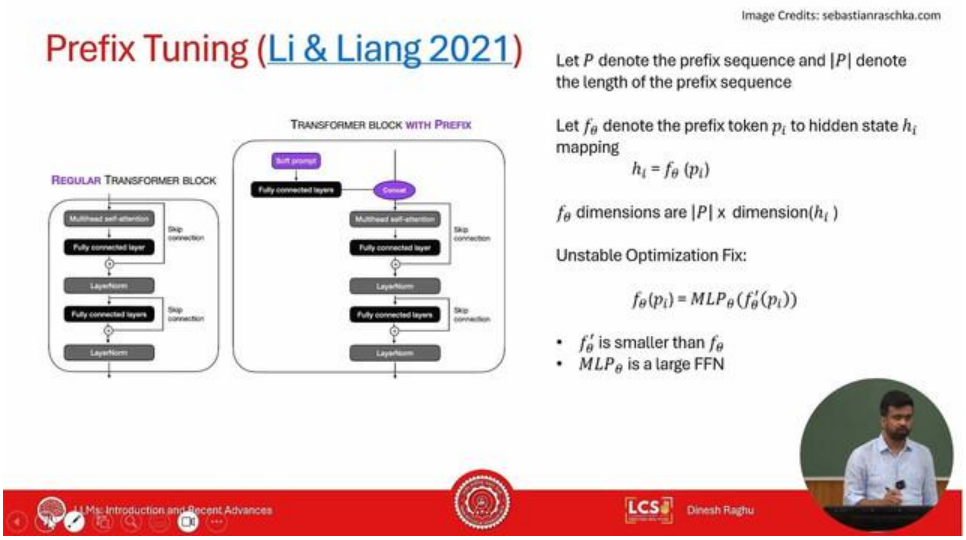
So, basically your embedding size is slightly lesser than what you had, but your MLP is very large. Let us say that the f theta is of dimension mod p cross let us say 200 or something and then you will have 200 cross 1024 which is your dimension. So, 200 cross 1024 is a very high larger parameter space than what you would have used for theta. Then f theta prime is much smaller. So they found this to help in a much more stable training right.



So again what the advantage here is that during training we might need a large mlp and f theta prime but once you finish training you can club both of them together and get back to the same size as what you have for f theta right. So this is just a hack for unstable training and i covered this part just to illustrate why design choice is very important. Because you will end up having unstable training, which is not something that you want to have in any LLM training or fine-tuning. So for prefix tuning they they evaluated their technique on two different setups. One is they use it for a task of table to text generation where you have a let's say in Wikipedia you have a table in the side about the bio and things like that right and on the right you have the introduction page introduction paragraph which mostly contains everything about what is there in the table.



So that's sort of an example of table to text but there are other variants of it So they trained a decoder only model at that point in time it was gpt2 for table to text and then they trained bad for summarization right and what they found is that by using only 0.1 percentage of parameters in these models they were able to get comparable results of full fine tuning right for each of the two models separately. Again, the same thing that we saw for soft prompting is that prefix tuning also works well on unseen domain as well. So let's move on to adapters. Now adapters was actually the first technique that got popular.



This was in 2019. And after this came soft prompting and prefix tuning. But I just covered them because it's much more easier to discuss. So in adapters, what they do is that they add new layers inside each transformer block. And these are called the adapter layers.

Each adapter layer has a fully connected layer which sort of projects the hidden dimension down to a smaller dimension and then there is some non-linearity and then there is a fully connected layer that projects back up into the higher dimensional space. So, this is the overall architecture of adapters. So, please notice here that there is a residual connection here, which is the usual one as what we have in the transformers, but there is also a residual connection here within the adapter. So, this is to ensure that if you randomly introduce new layers in your transformers, it's very likely you will break everything. because now you have random weights in between whatever you learnt might go for a toss.

So, this residual connection is very important because this will ensure that if this layer is around like is let us say a Gaussian initialize if the Gaussian distribution of a very small variance and zero mean then it is very likely that in the big in the first epoch, you would actually get the same output as what an LLM would give without the adapter layers. So this way, again, your training will be stable. So like I mentioned, there are two feed-forward networks inside the adapter layer. One sort of so this is the first part, which is a down projection feedforward network. And then you have a lesser dimensional space here.

And then again, you project it back to a higher dimension. This is typically referred to as a bottleneck structure. This is what autoencoders also do, wherein you want to represent your input in a much richer dense representation and then you use that to actually represent the input. So this is sort of the same structure here. So why do we need this bottleneck structure right? Why can't we just keep a single feed forward network or some field or network with non-linearity right? So the main advantage here is that this reduces the number of parameters significantly.

So let's say that like your dimension hidden dimensions be 1024 right and now you're say down projecting it to a value m which is 24. So then in in in the beginning you might have like if you did not have this bottleneck structure you would have had 1024 1024 parameters, but now you would have two times 1024 1024 right, this is like orders again orders of magnitude lower than what you would achieve otherwise right. And here, again, like I mentioned earlier before, even with respect to different types of test techniques, you will have a trade-off between performance and overfitting. So similarly here as well, within adapters, this value M gives you that trade-off. So you can use this as a hyperparameter, train different variants of the model, and see what works and what doesn't work.

So one of the biggest issue why adapter did not succeed that well is because like firstly its performance like people found out better techniques that were much more efficient than them and gave better numbers as well but the other thing is that there is an inference overhead right. So you're changing the architecture of the model in such a way right that it makes it difficult to remove things and add things let's say like if you want adapter a instead of adapter b that is already being hosted it becomes difficult and secondly since they are in between layers of existing transformers it increases latency as well. Because you need more

time to do you need more computation right. So this is an issue with adapters, but most of the other PEF techniques do not have issue with latency overhead because they sort of expanded along the sequence length rather than the transformers length. So, that is a disadvantage of adapters.

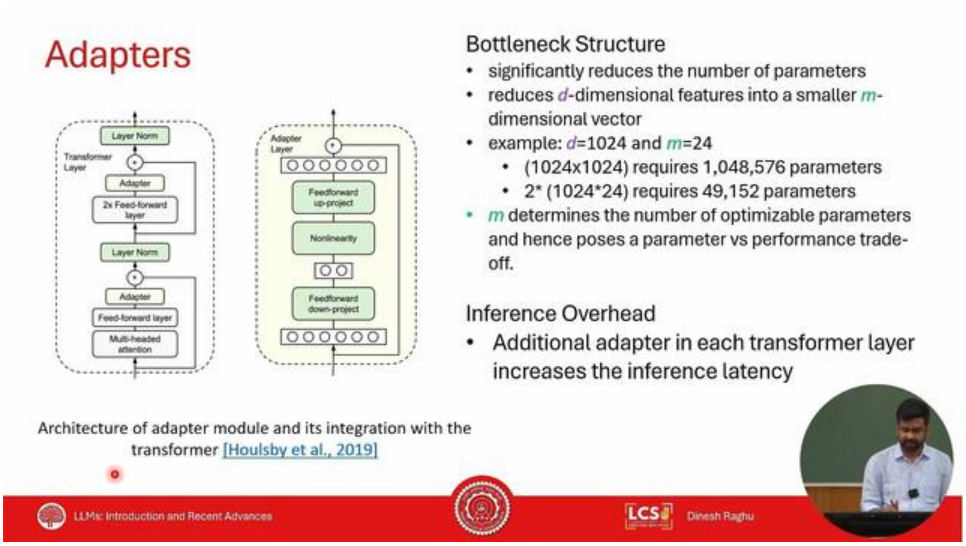
So let's see. So again, this is a model that was proposed. This was the first model that was proposed, so here, let's look at the graph here. So the blue line here indicates the fine tuning top layers of bert right. So here 10 power 6 would let's say indicate first k layers of bert, and 3 into 10 power 8 would indicate almost all the layers in bert right.

So this point would be similar to fully fine tuning and this will be like as and when you leave out some layers in the bottom and finally you come to this point right So the reason why this was used as a baseline was because people were doing it with bert doing this type of fine-tuning with BERT even before adapters. How you freeze word vectors or word-to-vec or glove embeddings in the bottom and you only train the network in the top, people started doing this with BERT as well when BERT came out. Let's not train the whole BERT from top to bottom because most of the lower layers are going to capture more word-level features and more English language-based features. Only the top layers are going to capture the task-specific features, so people decided like, i'll freeze most of the things from the bottom and only trained layers in the top. So this was the baseline then and this is the adapter performance and here again the x-axis indicates the number of trainable parameters and the good part here is that with only 3.6 percentage of parameters is able to achieve as good performance as full fine tune right because full fine tuning is here it sort of achieves that performance very early on in that Again, like comparing this with let's say prefix tuning, prefix tuning came much later, but what adapters can achieve with just 3%, prefix tuning was able to achieve it with just 0.1% of the parameters. So, if you have a choice between adapters and prefix tuning, you should always prefer prefix tuning, most likely. Okay, so this is the last part of the talk. This is called low-rank adaptation or LoRa.

For me, this was one of the most interesting papers that I came across and even the theory behind it is more grounded and it comes from a long literature of how people had worked from deep learning era till now. And this is one of the most popular techniques that are out there. And you have ready implementations available in HuggingFace, which you can

easily use for any LLMs that you have. So before going into LoRa, let's look at how people ended up figuring out LoRa. So there was this paper in 2018 that talked about something called rank composition in machine learning models.

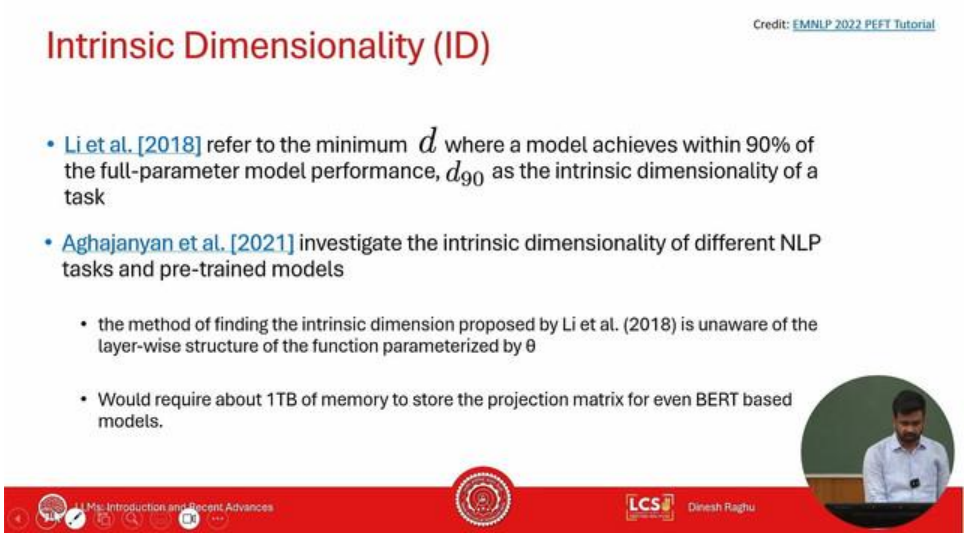
So, here what they say is that, let us take this equation here. So, here what we say is that, let us say that your entire features is d-dimensional, all the trainable parameters in your model everything like don't think of them at separate attention layer or separate feed forward network and things like that right just club everything together and then create one big vector like insanely long vector right. So ideally when you do fine tuning what are you doing? You're going to take that vector at state 0 and you're going to use your examples compute loss figure out something and then finally you will come up with an increment on those weights right and then once you add these things up you end up getting a fine tune model right, this is what you do. Now, what this paper said is that for every task, you do not need that many dimensions. Maybe for some tasks, you just need very few dimensions and the model will learn it.



So, what they said is that if you have a random projection matrix, this is randomly initialized and fixed, we are not going to train it. Now, there is a certain D dimensional vector. The dimension of P is D x d and for a certain D which is much, much, much lesser than small d much lesser than capital D, there exists a theta d which will achieve same performance as theta tau of d. And this value d depends on what is your base model, how

capable it is and what is your task at hand. If you have a very naive base model, even for small tasks you might need a big D.

But if you have a very capable model and your task is simple, you might need a very small d. So, it depends on the combination of both. So now they define something called intrinsic dimensionality. What it says is the value of small d that gets you 90% accuracy or performance as your entire full fine tuning is called the intrinsic dimensionality of that particular task. So that's what they define as intrinsic dimensionality.



Now this was in general made for deep networks, right? But then in 2021, people investigated this in the light of large language models and specifically for NLP tasks. So, while I mentioned that Lee et al, what they did was they considered the trainable parameters as one long vector. They did not take into account the structure of transformers because there is a repeated structure and there is a layer-wise structure. So, maybe the structure will help you do things with much that the expressivity of such a repetitive network is not leveraged in computing the intrinsic dimensionality. So these guys came up with a way of measuring intrinsic dimensionality, which takes into account this structure.

This is not the best way of doing it, but still they figured out that something is missing and we need to take it a little forward. Now, if you, so the reason why they did that or they thought about in that direction is because it was almost impossible to measure intrinsic dimensionality for LLMs, right? Why? Because let's say you want to do it for BERT, you

need 1 terabyte of memory. Because the projection matrix because of the computational how you sort of do the computation and how you factorize the capital P matrix. So even if you use the best implementation possible you still end up requiring a lot of memory right So they were not even able to figure out how to use it. So that's when they came up with this structure of a technique what they said is that i will do it layer wise I will represent the theta layer-wise.

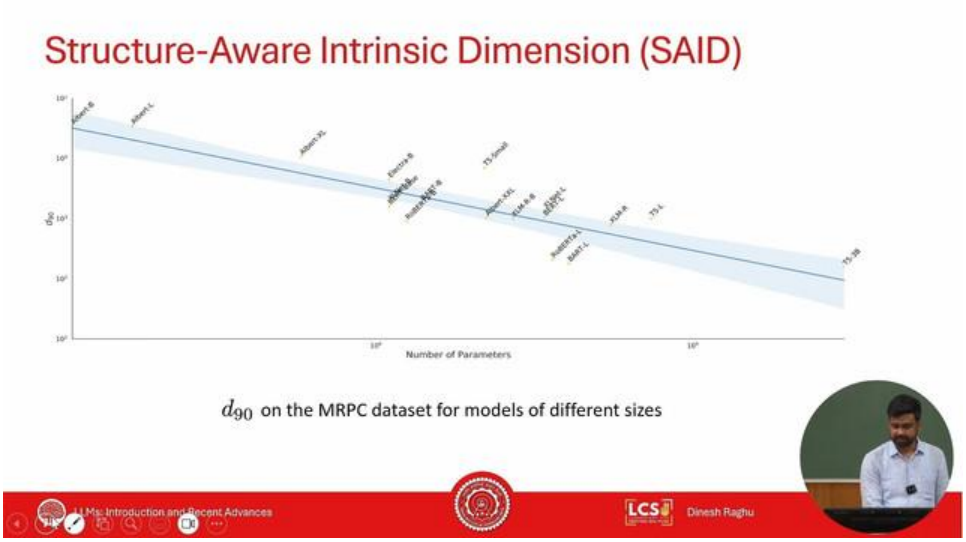
Rather than having the entire network, if you have 24 layers, then you divide them into 24 different vectors. And the dimensionality of P comes down drastically now. And then there is also a scaling factor that they add for each update. And these scaling factors are different for different layers. And so if you add everything up, the lambda i's and the theta's, together they will you will get a d-dimensional vector right.

So the m is removed there because m is the number of layers we have and there is going to be a lambda i for every m So the total number of parameters that you will end up training would be exactly m right. But again even this sort of is little difficult because the models grew insanely higher after bert right. So now even if you use the same technique again you will end up running out of memory for the other case as well, right? Okay, so here, so in the previous equation, what we had is theta d is the initialized value of the first, at the zero epoch, what was theta d plus a projection matrix times theta small d, right? So, now what we are saying is that we are going to split it for every layer in the transformer, right? So, that would give you, so forget about lambda i for now. So, if that is the case, again the equation still remains the same. It is just that you will add a suffix to all the thetas and the capital D would be reduced by a big margin.

Because otherwise, let's say if you have 24 layers, your vector will be like 24 times of what a single layer would be. And the projection matrix would be insanely high because of that. What they found is that adding a scaling parameter there actually helps them do some things easier. So they wanted to add a scaling parameter. Now, if you add a scaling parameter, you're again increasing your space of the vector small d.

theta d should be d-dimensional. Now, if you have d-dimensional there and you have m additional lambda i's, you get d plus m. So, you do not want to do that. You want d to be the same. So, they removed m from there.

So, let us look at, so this notion, so initially it was called intrinsic dimension, now it is called structure aware intrinsic dimension. So, here what they say is that, so they did this particular way of fine tuning the model and figuring out what is that D that gives you 90 percent of the accuracy that you would get with full fine tuning. The number of parameters in the model is listed in the x-axis and the y-axis indicates the intrinsic dimensionality. So what we see is that when the number of parameters are small in the model, the intrinsic dimensionality is quite high. As you increase the number of parameters, there is a steady decrease in the value of d that is necessary to learn the task.



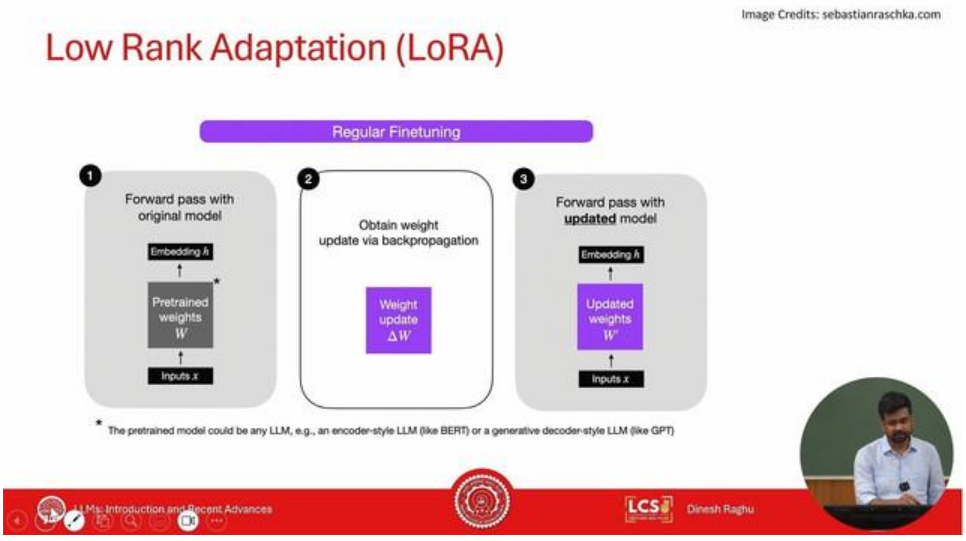
So this sort of indicates that the more the better the more parameters you have in the model you have more domain or world knowledge and the more world knowledge you have you can learn a task much easier in a lower parameter space than what is needed right. So this was a very interesting result from this 2021 paper So let's see here. So here what they said was, I have intrinsic dimensionality, which is computationally very difficult to do. And then I have structure-aware intrinsic dimensionality. So there are two takeaways here.

One takeaway is that having structure aware reduces the memory requirements and the train time by quite a bit. So, you will be able to fine tune much better if you take into

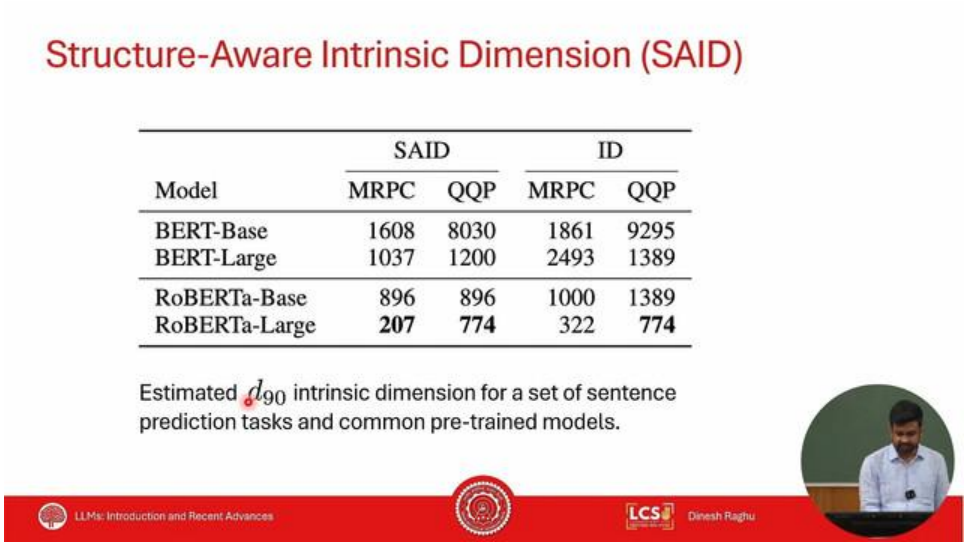
account the structural aspects of your network when deciding what you should train and what you should freeze. The second point is, if you take into account the structure of the network well. Then the dimensionality comes down. So, here for the Microsoft paraphrasing corpus for this task that the dimensionality when you consider one whole vector without considering the layer wise structure is 1800 whereas for structure average it reduces to 1600.

The dimension itself comes down. You need much lesser parameters to learn just if you carefully place what you should train and what you should not train. So, this was a brilliant takeaway from this paper. While this was a parallel effort, people did not put in that much thought process into doing this for PEFT. They were doing prompt tuning, they were doing prefix tuning because they felt like instead of doing it in the bottom layer, you could do it everywhere.

People did adapters, but they were not giving it much thought about, okay, maybe we should try to reduce the intrinsic dimensionality, maybe we should worry about what it needs, how do you control this and so on. was the base for this LoRa paper. So far we know taking into account the structure and then making use of the structure helps but how exactly to make use of the structure and how do you reduce the parameter space is what LoRa brought in. So let's look at regular fine tuning from the view of just weight updates. In regular fine tuning you would have let's say pre-trained weights and then after training the model and back prop you will end up getting a weight update del w.



And then if you add w and del w, you will end up getting the new weight, which is going to be the fine-tuned weights. So now this can be thought of as you're just having two parallel networks one that goes to the pre-trained weights and one that goes through the recently updated weights right and then you can add them you will end up getting the same thing right. So what laura said was they made two important contributions one is they told us what weights you should modify which will reduce intrinsic dimensionality by a lot So, having make use of the structure and they figured out this is the way in which this is the set of weights in the transformer block that you have to modify. That was one contribution. Second contribution was how do we represent the del W matrix.



So, this is sort of inspired in the sense that adapters anyway did this, they reduced the dimension of the weight matrix by using the bottleneck structure. So, they are using that bottleneck structure and so basically the del W here is now factorized into BA, one down projection matrix here and there is an up projection matrix here and there is no non-linearity. Just two projections. So now by doing this, so here FT indicates full fine tuning. Bit fit is a technique that was proposed where you only change the biases in the network and not change the weights.

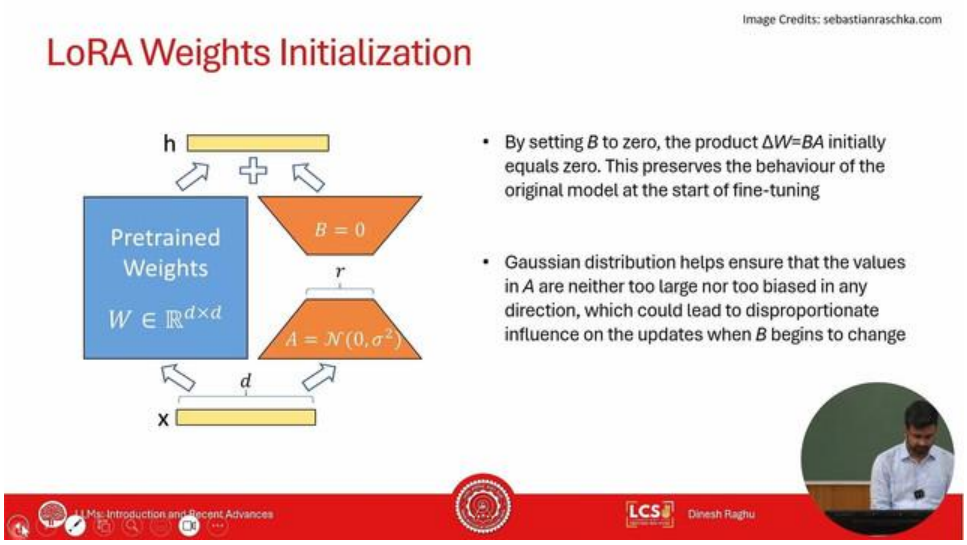
That's a trainable parameter. Pre-M is the soft prompting. Pre-layer is prefix tuning. Adapters are the same, the Hall-Speed adapters that I talked about now with different trainable parameters. So what we see here is that LoRa on slightly more complex tasks, WikiSQL is natural language to SQL.

This is a natural language inference task, which talks about entailment. This is summarization data set. So in all these slightly more complex data sets, we see that for a very small fraction of the trainable parameters, you end up getting even slightly better accuracies than full fine-tuning. So this is the contribution of LoRa. Now, like I said, they also told us what are the parameters in your model that you should update.

And what they said is that only worry about query key value and the output projection matrix. So these are the only four weight matrices that you should worry about when you do LoRa. And they also did a lot of ablations to show that if you keep the number of trainable parameters fixed right rank. So in if you and if you only train the the query vector the query matrix then this is the accuracy that you get for sql wiki sql and the nli task and generally if you only train the the key matrix and and a combinations of query and key here you reduce the rank because you want to maintain the number of parameters the same right and you do all combinations and they found that in most cases this is a good thing to do right. So when you define your lora config and when you try to find you in a model you will have these flexibility with you right you will be able to tell the training code, fine tuning code what matrices that you want to fine tune as a part of this LoRa PEFT and you can also tell it what rank you want and you would be able to train accordingly.

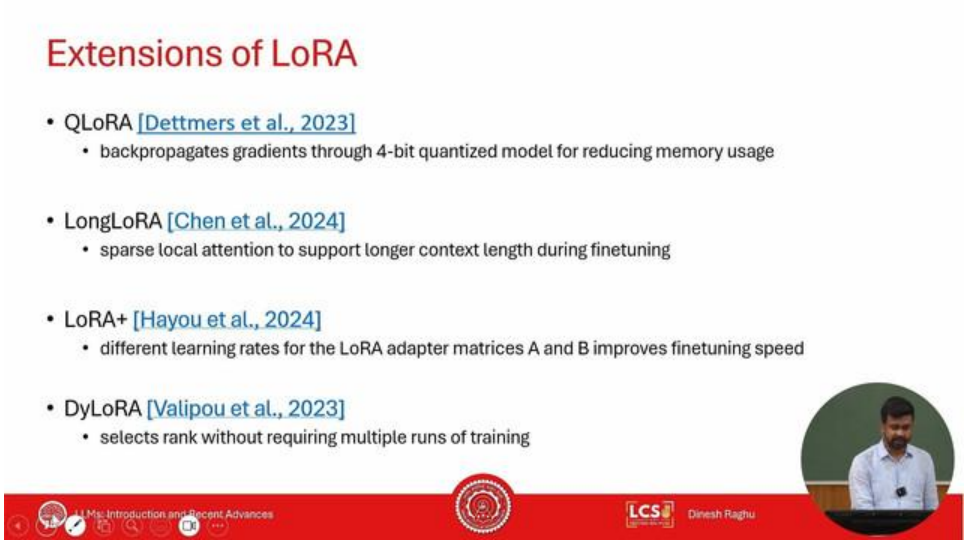
Now, they also show like the effect of rank. So basically, you can see that the model for various different things, the trend is similar that you sort of slowly raise up and then you saturate and then you come down slightly. That's the typical trend, right? And so here, if it does not come down, then it sort of, oh no, it has come down already. So, here in some sense the rank here is sort of the intrinsic dimensionality. It is not exactly the, it is not by the definition of what intrinsic dimensionality is, but the rank sort of gives you a control of how much parameters are needed to learn your task.

And this is one of the most popular PIF techniques that everybody uses. There is a slight variant of it called QLORA, which basically the only goal of QLORA is to reduce the amount of memory needed to do LoRa. And this is one of the most popular techniques that are out there. So before wrapping it up, I just want to talk about some more training aspects, which is the A matrix and the B matrix are initialized in a certain way, which will ensure stable training. And what they do here is that B is set to 0 for very obvious reason, which is if B is 0, then AB will be 0.



If AB is 0, then at the beginning of your repark, you only have to go through W. So, your model does not change the way it works. You are not adding layers in between. So, it is parallel. So, this helps in preserving the original behavior of the model.

Now, A is sort of assigned using a Gaussian distribution with zero mean and a very small variance and this will also ensure that the way B changes does not throw the model off right because b is going to change drastically now and if a is randomly initialized it may throw off and you would end up having the same issue as what people faced in prefix tuning. So a is initialized in this way so that it doesn't throw off during fine tuning. So finally, there are like insane number of extensions to LoRa because it has been so popular. People have explored all varieties of LoRa. QLoRa is one of the most popular technique.



There is also long LoRa in case where you have really really long context length in your input. There is LoRa plus. which sort of says that don't use the same learning rate for both a and b matrices in the re-parameterization use different learning rates and how it will improve the speed with which you can converse right and there is also dylora which which says that with like So, since r is a hyperparameter, you will have to run your experiments in parallel with different values of r and figure out what works best. But Dylora sort of tells you how to sort of get the right r without running it multiple times. So, the intuition here is that they want a del w which can be added to the weight W. So, if you only look at it as a projection, then there is no need for a non-linearity there to mimic that update of that projection. For adapters, it's a different story altogether. You're not trying to add a projection to another projection. So here, since you want the LoRa weights to be factorized and added with W, you don't want a non-linearity here.

So the advantage of doing this is that once you train A and B, you can just fuse it to W. If you want a single... Again, it will affect all the W's in your network and you won't be able to do what you did with, say, prompt tuning because you cannot easily swap out and swap in.

There are also works that are going on for making LoRa as easily swappable as prompt tuning But you have to do it at a batch level because you can't have multiple different LORAs in the same batch. So, there is a work around hot swapping in which the amount of time that's required to swap is so less that you won't feel that there is latency there for

swapping. But that's the intuition behind no non-linearity in this particular re-parameterization. So, I'm not saying it's similar to, all I'm saying is if your task is complex, it is synonymous It's not equated, intrinsic dimensionality is not equal to rank, but it's synonymous to intrinsic dimensionality. If your task is difficult and your base language model is also not that good, then you need a bigger R.

If your base model is good, your task is difficult, you might need slightly middle ground. But if your base model is good, your task is easy, you can even get away with rank 1. Alright, I think we are at the end of the lecture. So basically, just to summarize, we covered in addition to introducing what PEFT is and why we need PEFT, we covered four different techniques that are very popular and changed and were like sort of defining moments in this literature. Thank you.