Introduction to Large Language Models (LLMs)

Prof. Tanmoy Chakraborty, Prof. Soumen Chakraborti

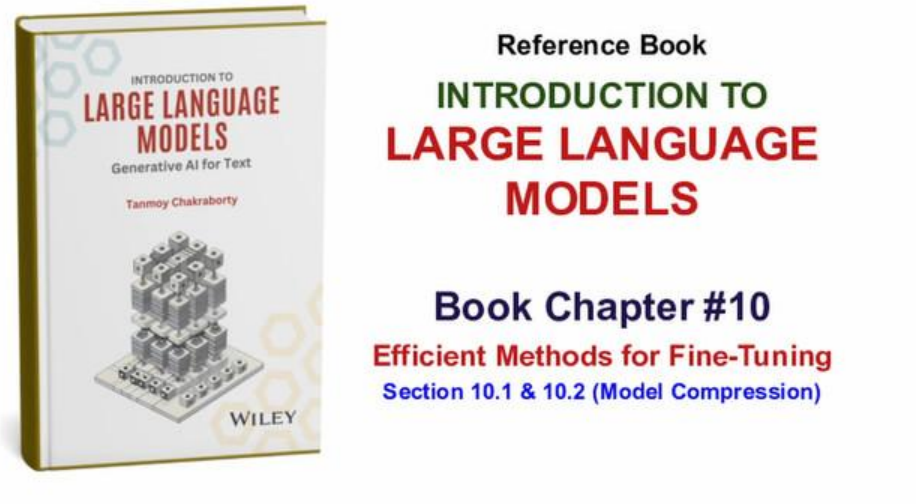
Department of Computer Science & Engineering

Indian Institute of Technology, Delhi

Lecture 30

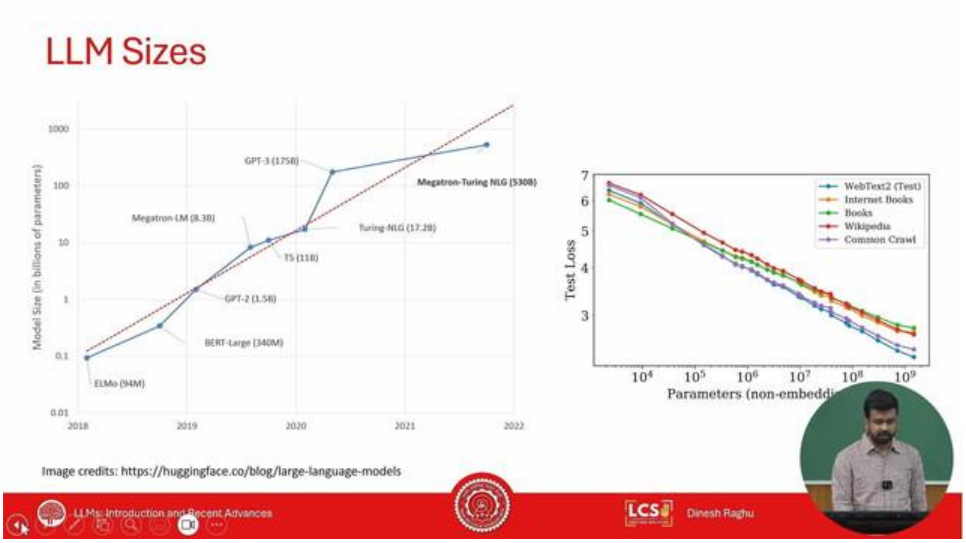
Quantization, Pruning & Distillation

Alright, hello everyone. So, today we will be talking about quantization, pruning and distillation. So, these all fall under the same umbrella and maybe once I go into the introduction, it will be clear why we are discussing these three topics together. So like we discussed last time, the reason why we wanted to do theft was because of how the model sizes have been increasing over time. And this is again a recap of that.



like over time the size of the model is increasing exponentially and it is not just the size but the performance of these models are also increasing over time. So, here we see that these are the test laws of various language modeling tasks on different data sets and you see that they are sort of decreasing considerably based on the amount of parameters that are there in the large language model. So now, what are the flip side of having such a large models during inference? So, the first thing like we discussed last time is that the bigger the model is, you will have to buy new hardware to support things. In sort of, not so

monitorly friendly to keep buying hardware as and when the model keeps growing over time. So, then this also puts a cap on how many organizations can actually run these LLM inferences.



So, GPU requirement is going to get larger and one of the biggest problems in deployment is latency. So, the larger the model is, the more time it is going to take to come back with a completion for a given prompt. Now, let's say that you have a chatbot that is deployed whose backbone is an LLM, then having to wait for 30 seconds or 40 seconds, it seems like really difficult at this point in time when we are so used to getting replies very quickly. And the way the LLM field is progressing now, people are not just using single LLM call per sort of responding back. For example, there is now this new paradigm called agentic behavior where once a language model responds, you ask the same language model or a different language model to reflect on the output or even sometimes execute the output, for example, if the model generates a code, you execute the code, get the outputs and then make the model reflect on the output and then decide whether there is an execution error, should it redo things. And then finally, you give back the answer. So now if you have to make multiple LLM inference to just get back with one output, then that's going to increase latency even more. So latency is one of the biggest concerns. Third is inference cost.

If you have an application deployed which uses LLMs, of course, you'll be worried about how much money you're going to spend to serve a single user. And the money that a single user is going to pay you should be more than what you invest for that user. So, if LLM for slightly more improvement in accuracy, if you have to spend a lot more, then it does not seem commercially friendly. So, of course, inference cost is going to be one of the biggest dimensions. And finally, sustainability, environmental concerns.

So, you need a lot of energy to run these LLMs, you need to support data centers which require cooling water and so if your energy is not coming from a sustainable source, then there are a lot of carbon emissions which sort of affects the overall environment. So, again LLM inference has a huge impact on sustainability. So here, before going into how can we do this, let's look at what just happened very recently. In OpenAI, after they released GPT-4o, which is the pink model, they released something called GPT-4o Mini. Now, GPT-4o Mini is sort of like, let's say on MMLU is six points lower, on NGSM it is three points lower, the GPT-40 mini is also almost always better than the previous version.

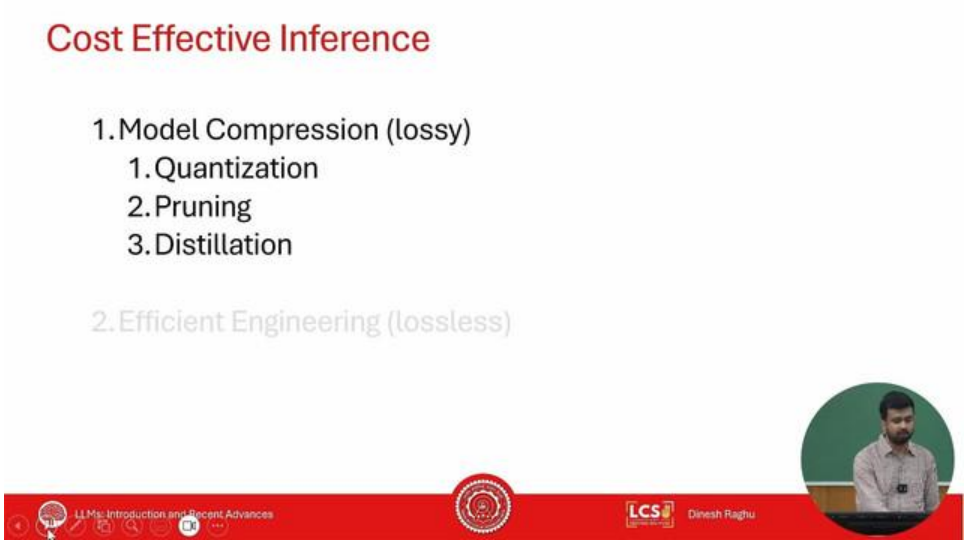
I mean the one before GPT-4o, GPT-3.5. But if you see the inference cost that you will have to pay, you will end up paying a lot more for GPT-3.5 turbo than GPT-4o mini. Even now, so this is a website called GPT for work where if you give them the number of tokens in the input, the number of tokens that you expect the model to have in the output and how many times you'll end up calling this again and again, it'll give you an estimate of how much money you will end up spending.

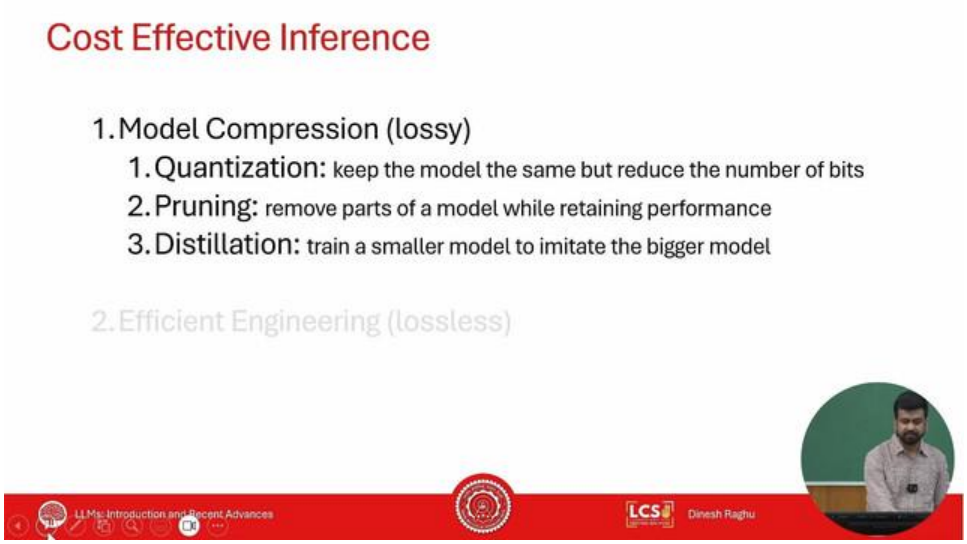
So, here if you see for the same amount, for the same setting, for GPT-4o you end up spending $80, but for GPT-4o mini you only spend $3. You guys just saw the performance drop you get for GPT-4o mini, it's not that bad. So, if you are able to get this much reduction in cost for slight drop in accuracy i think like lot of people would go for it right because it makes so much commercial sense to do things like this rather than go for high accuracy Now why is gpt 40 mini so cheap right compared to the the non-mini version right or the bigger question here is that how can we deploy llms in a cost effective manner while maintaining the performance of the bigger model. So, this is what we will be answering in this talk today. We are going to cover the different ways in which we can achieve this.

So, there are two main classes of techniques that you could do. One is model compression. Which is like if you have a 65 billion parameter model, you can compress it to let us say 8 billion parameter model with very minimal loss, like not so much. But there are also another class of algorithms where you would do engineering, efficient engineering to sort of achieve the same thing that you achieve now. But it sort of helps you reduce the inference cost.

So, I can give you some examples of efficient engineering. For one of the most common operations that you do in transformer inference is scale dot product. You have the key vector, you have the value vector, you will have to compute a dot product, then you will have to scale it up or scale it down and then you will have to do a softmax. There are software kernels that have been developed which fuses all these subtasks together into a single kernel and that gives you tremendous improvement in efficiency. So, here it is not lossy, you do get the exact same operation, you do get the exact same output, it is just that you do some engineering efficiency to reduce the cost.

Now, there are also hardwares that specifically support certain operations out of the box and people who do this efficient engineering when they fuse kernels, they also effectively make use of the information provided in the instruction manual in the data sheet about how to leverage those operations and then make it more efficient. So the efficient engineering part, Yatin will be covering it in the, some parts of it Yatin will be covering it in the next lecture. For this lecture, we will focus mainly on model compression techniques and specifically we will be talking about quantization, pruning and distillation. So let me give you a brief about what these are. Quantization, as the name suggests, what we want to do is we want to keep the exact same model, but just use reduced number of bits to represent parameters and activations and whatnot.

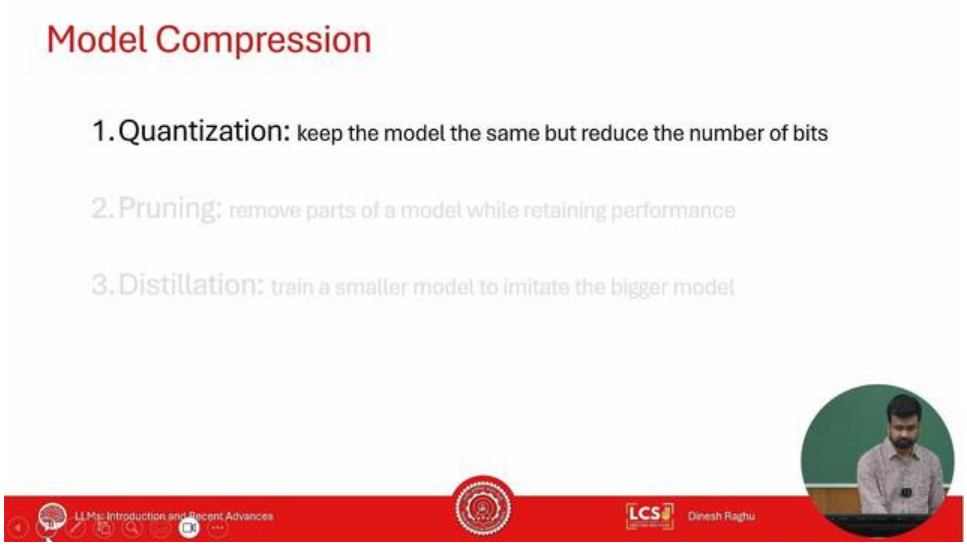




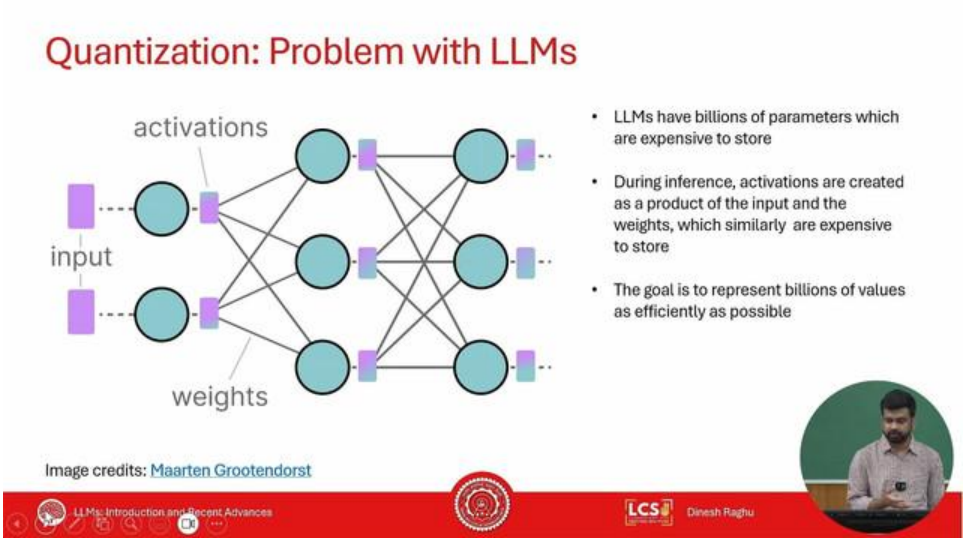
And pruning is, as the name suggests, again, you just prune out certain parts of the network. And then while trying to maintain similar performance, of course, it's going to be lossy. And finally, it is distillation where you have a big model that is trained well and then now you train a smaller model by using the bigger model as a teacher. Do not use the real output, but rather use what the model predicts to guide the smaller models to do. Now, out of these techniques distillation helps you is the least lossy.

But it's the most expensive because now you have to train the whole thing. First of all, you have to do inference over the teacher network and then get the outputs that itself is

expensive on a really large data set. And then you have to use those outputs and then backdrop on the new model, which is smaller, much smaller than the original architecture to sort of achieve model compression. So distillation is the costliest, but it gives you a lot of It is sort of the best way to get to maintain the accuracy whereas quantization and pruning are sort of slightly lossy, but they require very little changes. But quantization has other advantages as well which I will be covering when I talk about So, let us first talk about quantization.



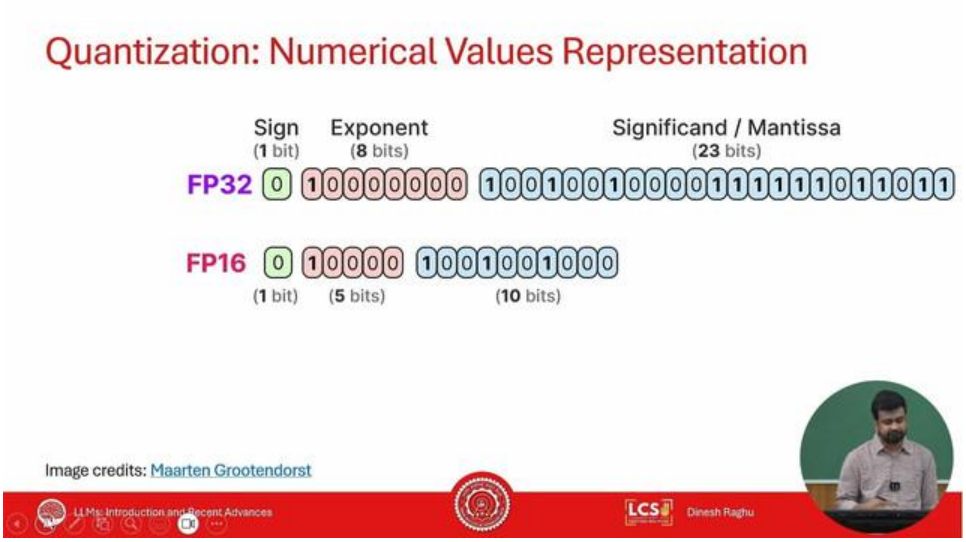
So, before jumping into the quantization technique, let us just do a recap of how LLMs do inference and talk about how floating point numbers are represented in the computers and then finally talk about quantization. So, this is a usual neural network where there is input and then there are weights, which are here, and you also have activations here. So LLMs have a lot of parameters that are in the weights, and they are really expensive to store. And during inference, you end up creating activations. And these are the product of input and the weights.



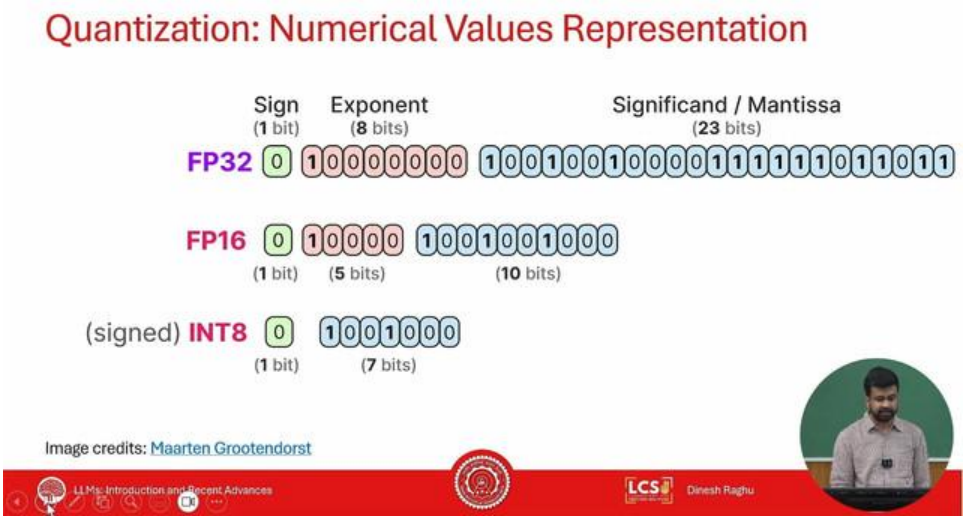
And they also are required to be stored. And the goal of quantization is how efficiently can you store these values so that you end up using lesser memory, you compute faster and you save cost. So, it is more about how do you represent them. So, let us talk about how numbers are represented. So, typically a full precision is what is referred to as the FP32 notation where you represent the floating point with a single sign bit for positive negative and there is an exponent that is represented using 8 bits and the fraction or the mantissa is represented using 23 bits.

So, the difference between the two smallest numbers that you can represent. The difference between the two smallest numbers is so low and it is sort of captured in the 23 bits. So, the precision is really high here. This is typically called full precision like in the past a lot of machine learning modules used FP32 to represent. So, this gives enough precision for a lot of scientific applications.

And after a while, in the deep learning era, people moved to FP16 because GPUs out of their box started supporting FP16 with improvements in time than FP32. And so here the difference is that as you see, it is the number of bits that you use for exponent and the number of bits that you use for the fraction. So, the range has reduced because the exponent has reduced and even the precision has reduced considerably. But for machine learning, specifically deep learning applications, this still does the trick. You almost do not see any drop in performance, but your computations improve significantly if you move from FP32 to FP16.



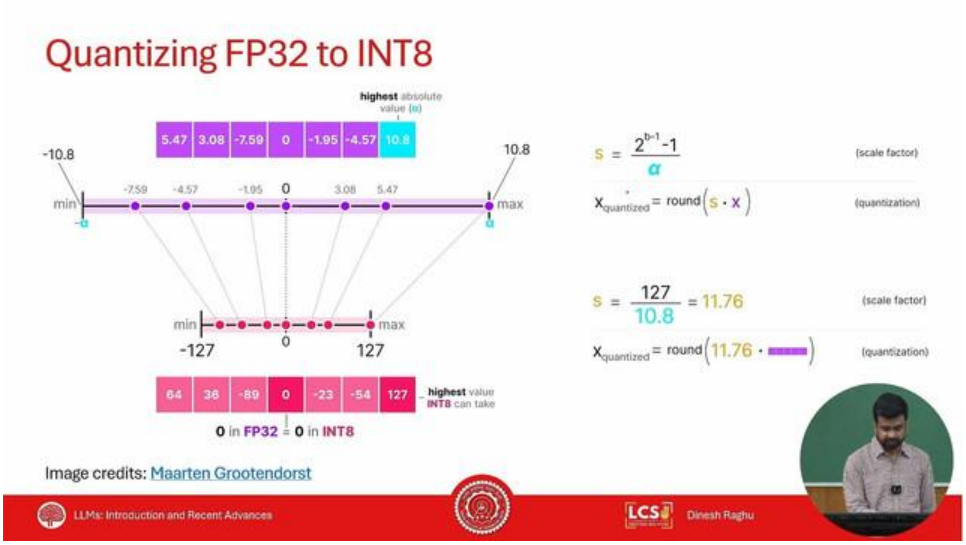
Now, you can reduce it even further. So, there are certain applications where people simply use int8 representation where you use 1 bit to represent the sign and 7 to represent just the rest. So, let us talk about how would you quantize a FP32 into int8, for FP16 also it sort of remains the same. So, this is a very simple way of quantization which is the simplest of all. So, what we do here is the first task is, so let us say this is the FP32 vector that you have.



So, first task is to identify the value in this vector that has the highest magnitude. So, for a given vector, you identify the max absolute number which is 10.8 in this particular case and then what you do is that you set the maximum. and set it symmetrical and the minimum.

The range is centered at 0 always and you set the maximum absolute value and the minimum absolute value as a two extremes and then the rest is simple.

You sort of map this maximum alpha to 127 and minimum alpha to minus 127. And how do we compute this number, the int8? So, basically the first thing you do is compute a scale factor. So, here we are using 8 bits. So, this number would be 127 and alpha is the highest absolute value. So, this is going to be your scale factor.



And to compute the, let us say the quantized value for 5.47, all you have to do is multiply the value 5.47 with the scale factor and then round it up. So, that is how you quantize a value. 32 bit floating point into a intake.

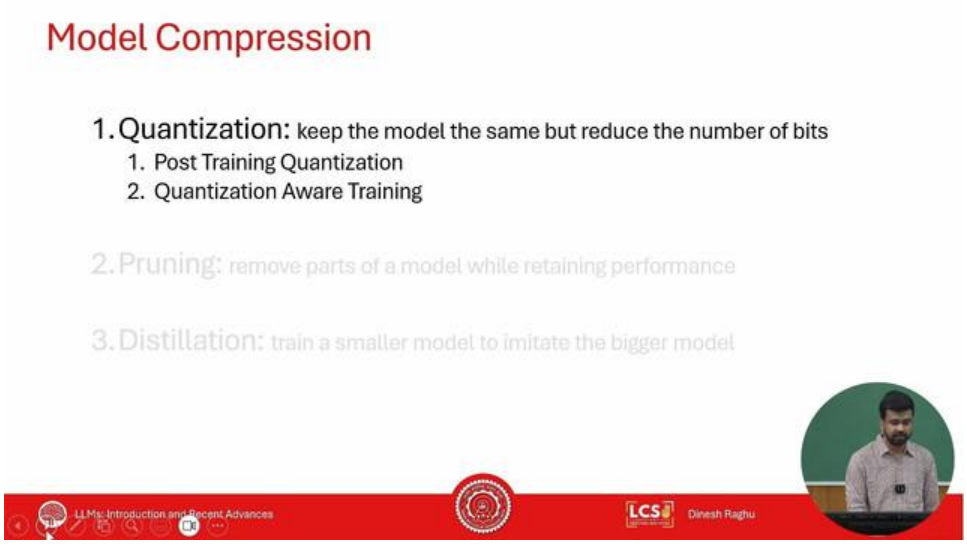
And of course, for different vectors, your alpha is going to change and so the scale factor is going to change. So, if you have let us say a two-dimensional matrix which has 20,000 rows, then you will have 20,000 different scale factors to sort of quantize. So, this is how quantization works. So this is how quantization works and de-quantization is quite straightforward. All you have to do is if you have the quantized vector and you divide it by the scale factor, you end up getting the de-quantized version which is the FP32 version.

So here you have the FP32 matrix. and then let us say you want to quantize it into int 8 format, you will end up getting the alphas and the scaling factor for each vector separate individually and you will quantize them and then when you de-quantize, of course, you are not going to get the original back. So, the difference between original and de-quantized is

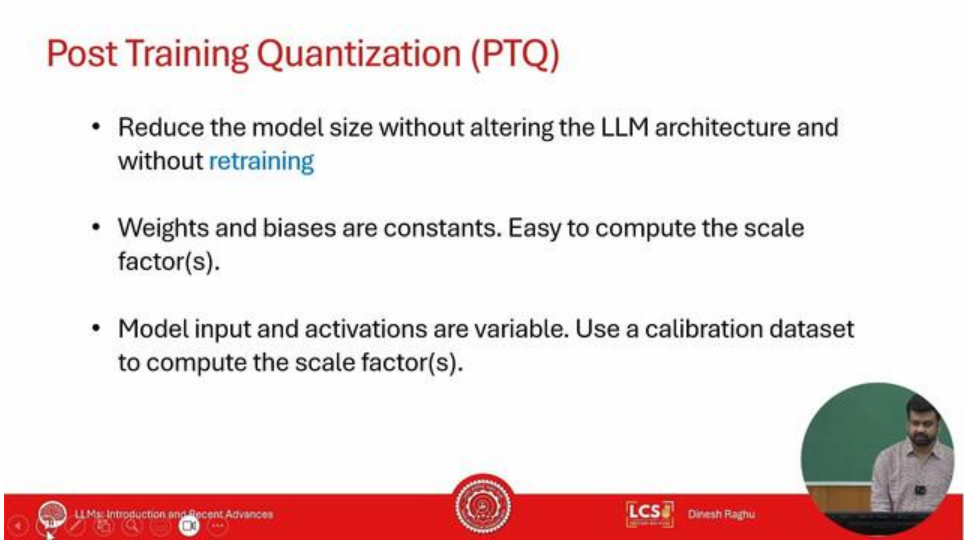
this quantization error. So, it is going to be lossy. and you will end up getting a quantization error.

So, let us see how this quantization and de-quantization is typically used in machine learning inference. So, there are two types of ways in which you can use this quantization technique in your LLM inference. One is during training, do not worry about quantization. You train the model in half precision which is FP16 or mixed precision FP32 and BF16 whichever format that the model, the code supports and you want to train on, do that. And during inference, you first do a post-training quantization and figure out your scale factors and other things and other constants and then use them for inference.

So, that is one way you can do it. The second way is quantization aware training. So, maybe like for example, we discussed LoRa in the last class and there is a variant of LoRa called quantized LoRa. So, that fine tuning would be is a quantized version of LoRa and it is quantization aware during training. So, you can leverage quantization even during your fine tuning process.



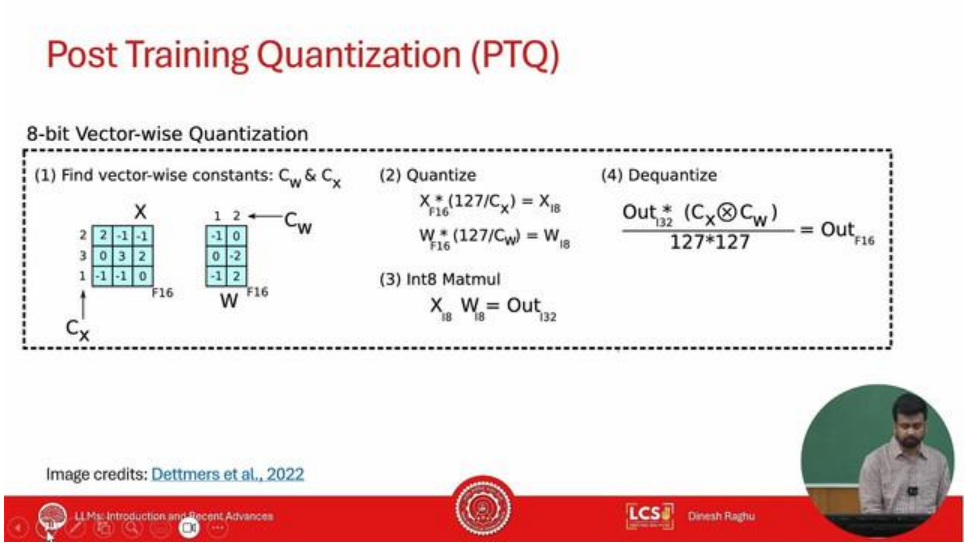
So, now let us discuss the post training quantization. So, here the good part is that you do not have to worry about figuring out what should be your compressed versions architecture should be. It is exactly the same. So, there is no difficult decision to take there and the good part is you also do not need retraining. So, it is very cheap.



But again, it is at the cost of how much quantization loss you get and how much lossy compression happens. So, the easiest thing here is that weights and bias are fixed after training and when you look at the weight matrix, you look at it at one vector at a time, so whatever quantization I discussed just before is called symmetric quantization because 0 is represented using 0 in the quantized format and you have both positive and negative going through the same value. But there are other asymmetric formats where 0 is not mapped to 0 and the negative minimum and the maximum values are not, the absolute values are not the same. So, there in addition to scale factors you will have other constants as well. So since weights and biases doesn't change over time, it is easy to compute all these scale factors and other constants and then keep them fixed.

But this is not going to be the same for input and activations. You're going to get inputs. like during you're going to get inputs during inference sometimes the input matrices might have a smaller max value sometimes it may have a really big max value which will make it like clip which will end up clipping if you choose the scale factors like in a not so good way. So what people typically do is similar to how we do for validation that you take some sort of a very small calibration data set okay pass them through your network. So if you just have the data set you will only know the inputs but if you pass them through the network you will understand how these activation values changes with respect to the weights.

So once you do it for a bunch of input called the calibration data set you will be able to get some sense of what are the range of values you would see and what vectors would look like what maximum value you would hit and what range you would get. So you use all these data and you again compute the scale factors. So now for a given network you would know all the scale factors that is needed to quantize for both weights for both model parameters and the inputs and activations right. So, then what do we do? So, typically what you do during inference is that you have multiple layers and within layers you will have multiple operations one after the other that leads you to complete that layer. So, the typical way people use post training quantization is take one operation at a time You have full precision or half precision, whatever precision the model is trained on as input, quantize the weights and the activations and then de-quantize them after that operation.

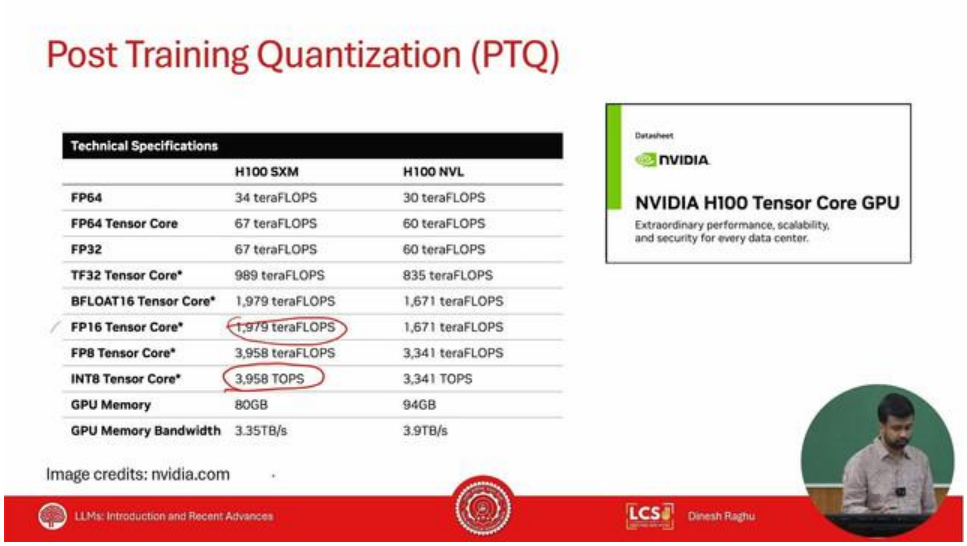


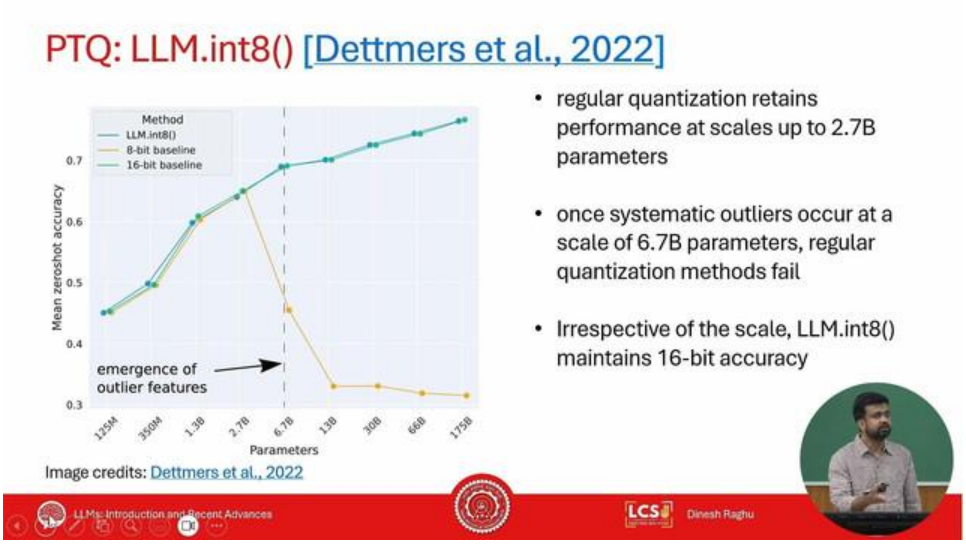
And then you keep repeating this. So, why is this efficient? Because if you do a matrix multiplication using FP16. there is only so much you can do. But if you convert this into a much smaller quantized version, then the number of operations you can do like blows up. So, that blow up is sort of not is so high that even the quantization and de quantization is sort of very small in that fraction.

So, you end up getting a lot of speed up. So, here if you see like you have the input and the weight matrix and then first you quantize them because here you know your scaling factor,

the scaling factor is 127 divided by alpha and you have the input, you quantize them from Fp 16 to int 8, you do the same for your weights as well and then you do your matrix multiplication in the quantized format and then you end up de-quantizing them. So why is this really helpful like i said like here we saw that we are converting from fp 16 uh to int 8 right. So this is a data sheet of the h100 uh gpus. So here if you see for fp 16 you get let's say for nvl you get or as xm you end up getting thousand nines 79 teraflops, but on intake you end up getting a lot more tensor operations.

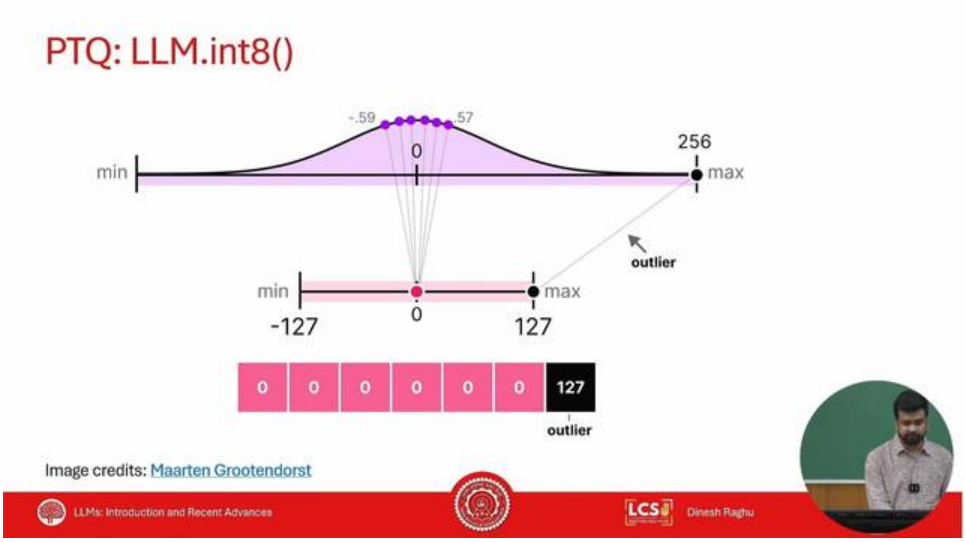
So, for the same amount of energy you end up doing lot more operations than what you would do on FP16. So, this is the advantage of doing this type of quantization and de quantization within every step in your computations. So, people started using this and this seems to work a lot well. But then in 2022, when the size of the model started to increase, people found that after a certain value, the 8-bit baseline was not giving as much accuracy as the 16-bit one which is the green line. So, here if you see the point at which this property emerges is around the 2.7 billion model.





Until then there was not much difference in performance here, but then after that there was a steady decline and it sort of does not make sense to quantize anymore. So, then these people figured out that the reason behind this emerging pattern and they also proposed a way in which you can fix this. And their technique is called llm.intate. Now we will discuss a bit about what this does.

So before going into what this does, the reason why this dip is happening is because of outliers in your vectors. So we will see what and what they say is that the percentage of outliers in your vectors are sort of negligible until you reach 2.7 billion parameter model. But once you cross that, you start from 6 billion and above, the number of outliers becomes so high. that you end up not getting the same performance as what you would get.



So, let us see the effect of outliers. So, let us say that you have your vector and most of the numbers in your vector are sort of between let us say 10, minus 10 and 10. or something like that. Most of them are in that range and a lot of them are focused around 0. But there is one value in the vector that is really high.

So, what happens because of this? The alpha is decided by this particular absolute number and so you set both your extremes to represent this particular number. So, then what happens is that every other number in the vector sort of ends up getting mapped to the same point, which means you end up losing everything just because of that one number and if most of your vectors are going to be like this, then quantization is not going to help. So, this was the reason why you sort of saw a dip after the 2.7 billion model because most of them seem to have these outliers. So how do we fix them? Their proposal was quite simple.

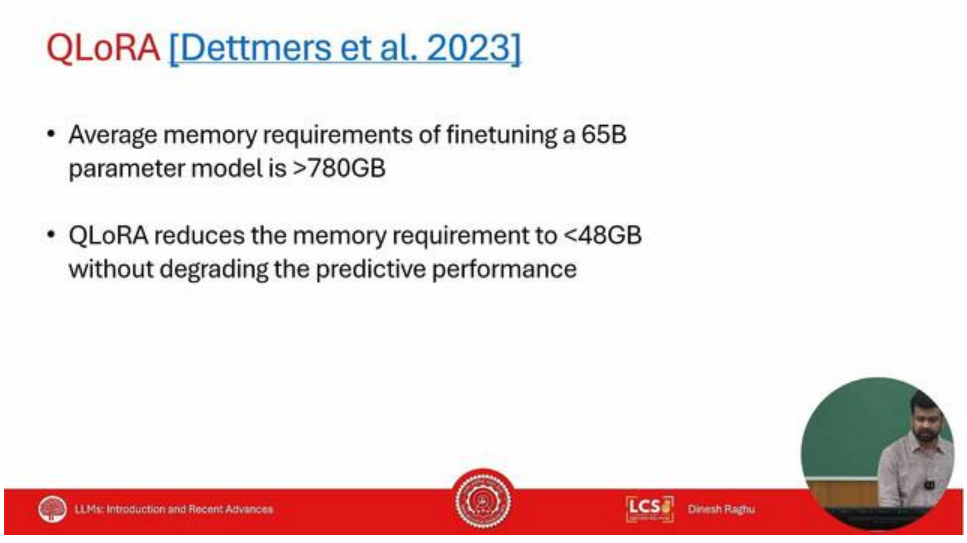
The main contribution of this paper is that they found the issue. So that was the biggest contribution. The approach is very simple and elegant. So this was what we used to do before LLM intake. If you have any matrix X and W, you would just quantize them and then do the matrix multiplication and then de-quantize them.

So, what these guys said is that you have a certain threshold and any value that crosses that threshold is sort of the outlier. So, do not use them to compute your scale factors or other constants, keep them away, just identify them as outliers and keep them out. Do all your quantization constants and then during input, when you see these outliers, then just mask

them out, take them out. So, if you see here, the yellow ones are the outlier ones. So, they have removed the outlier columns here and the outlier rows here and then they have kept it separate.

And now for the non-outlier ones, you do quantization. and for the outlier ones, you do not do quantization. We represent them as is and then whatever precision you used during your train and what the full model uses, use the same to do inference here as well and then you add them up. So, the simplicity of this sort of is really good and it helps solve this problem and as we saw in the first graph, once you do this small hack, then everything gets fixed. So, this is INT8, LLM INT8 post training quantization.

So, the most popular technique for quantization aware training is QLORA because like this is sort of one of the most popularly used PEF technique mainly because of two reasons. One is that its memory footprint is quite low and it almost achieves similar performance as FP16. Now, on an average, if you're going to fine tune a 65 billion parameter model, again, this is for full fine tuning that I mentioned that you would end up requiring 12 to 20 times the size of the parameters, right? So you will end up taking 780 gigabytes of memory, which seems like almost impossible to get that much GPU memory, right? So, the good part about Qlora is that it reduces the memory footprint by big margin and you can fine tune a 65 billion parameter model in just 48 gigabytes of memory. And the best part is that it almost doesn't degrade your performance for a lot of tasks. So QLoRa is sort of a combination of these four items.



So here, LoRa is sort of borrowed from the previous work. But what these guys suggested is that on top of LoRa, we would use this specific format of quantization of weights, which is called the 4-bit normal float. They would use something called double quantization. I will get into the details. And they also make use of some of these hardware specifications to sort of further reduce the memory footprint.

And this sort of is sort of, they looked at the fact sheet of NVIDIA GPUs and then they came up with these paged optimizers which would further reduce the amount of memory that you would need during a PEFT. So, let us talk about what the normal float 4 quantization is. Since it is 4, it is clear that it is a 4-bit quantization and float again, it would represent floating numbers and then n here represents a normal distribution. So, typically when you do the regular quantization, you would define the max and the min and then you divide the buckets. into equally spaced ones.

Because now you know the range, now you can bucket them. And because of this, what happens? The assumption here is that your array is uniformly distributed. So if that is the case, you'll end up getting sort of same number of candidates in each of the bucket and you would do the computations. But if you make another assumption that most of your weights are going to come from normal distribution, then this is not the effective way of leveraging quantization. So, for this rather than using equally spaced, now that you know the data distribution and it is not uniform, so you can get equally sized distribution by assuming that this is a Gaussian normal distribution.

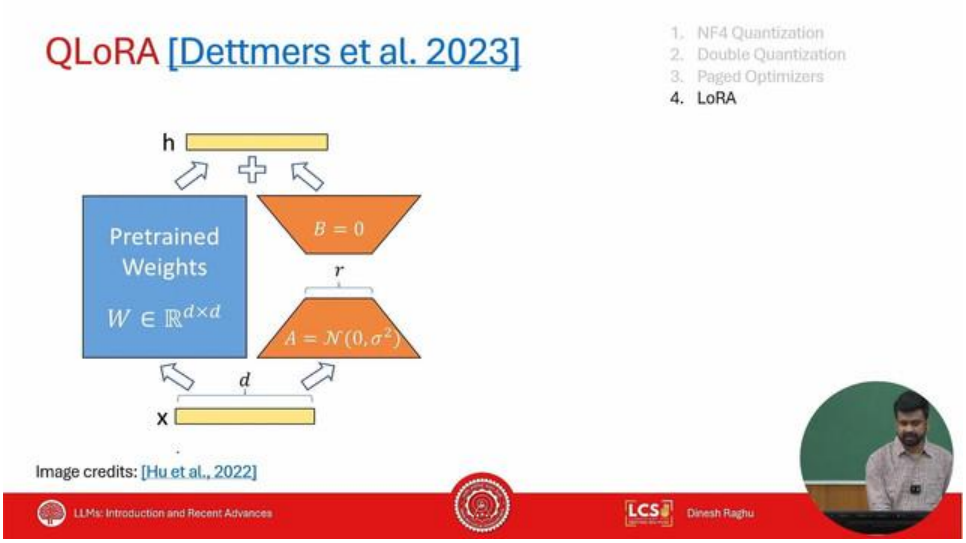
So, this is what in simple terms NF4 quantization is and so now let us look at what a double quantization is. So, when you quantize. you typically store the quantization constants in FP32, full precision. Because the more precise your constants are, the lower your quantization error is going to be. So, what people typically do is that represent these in FP32, but now imagine if you have so many vectors in your parameter space.

Because there are going to be so many matrices, there are going to be so many feed forward networks, so many weight matrices, attention weights and so on and you will end up getting so many constants. So, storing these constants would end up taking a lot of memory and what they said is that once I have all these constants, I will divide those constants into

blocks and further quantize these quantization constants. So, you sort of do double quantization. So, and they say that by doing double quantization, you do not lose so much on your performance, but you gain a lot in storage space. in saving storage space.

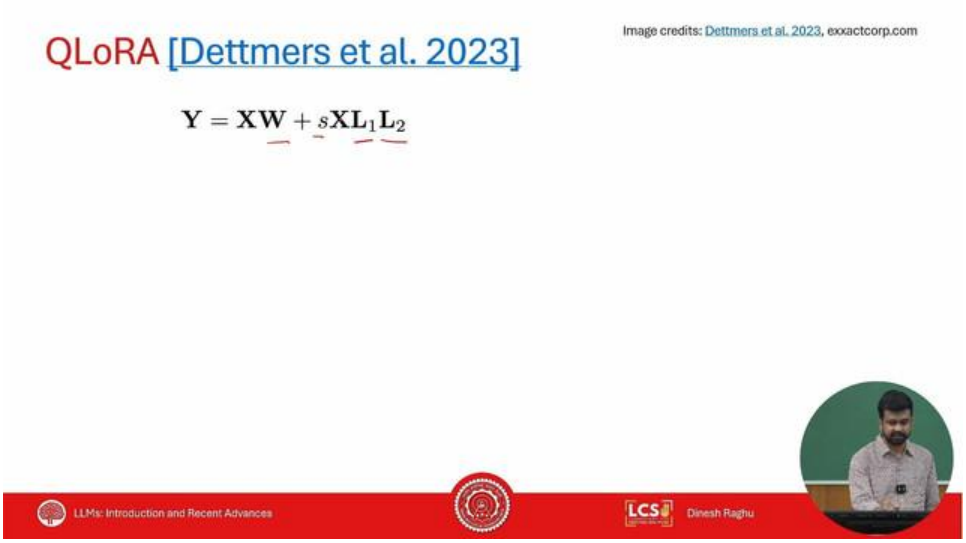
So that is the advantage of double quantization. It's not going to give you a performance bump. It's just going to save a lot of space. And thirdly, like I said, paged optimizers. So let's say that you want to fine tune a 1.3 billion parameter model and it needs about 21 gigabytes of memory. So if you look at GPU in isolation, you don't have 21 gigabytes of memory. But if you look at the memory that the chip has on both GPU and CPU, that is good enough. So, how you do paging for CPU and main memory? Some of these hardwares do the same for GPU and CPU memory. And what Qlora does is that it makes use of this particular feature in the hardware.

So you would be able to fine-tune a V1 model with this particular GPU, even though it has only 16 gigabytes of GPU memory. So this is one thing. And finally, of course, they are going to use LoRa as a base for fine tuning. So, a small recap of what we did in the last class. In LoRa, basically the weight matrix is divided into W and del W and this part is del W and this part is W.

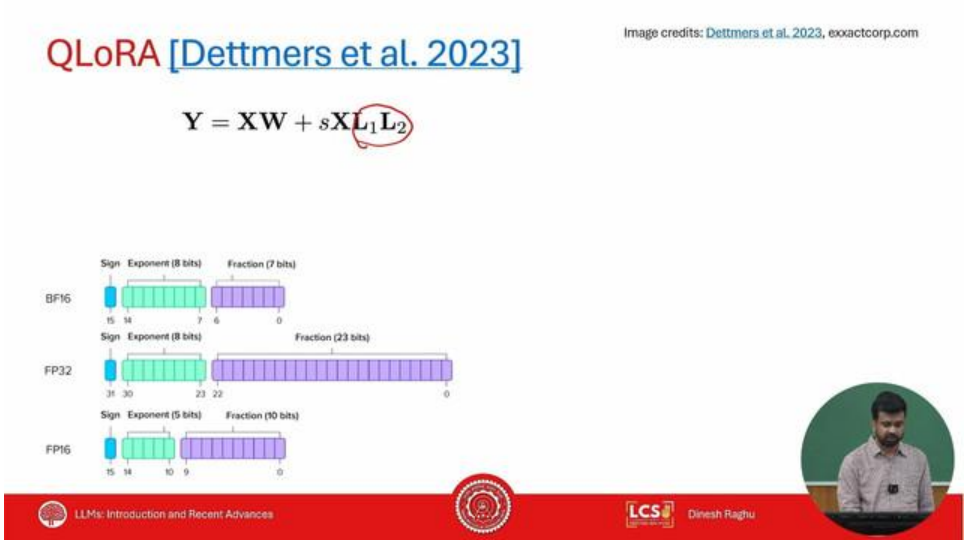


and any input is first multiplied here with this and then here with this and this sort of leads to plus del W a times, right. So, this is what you end up getting and this is factorized in a way where the rank here R is much lesser than t, right. So just the same sort of equation is

slightly rewritten where you have the input, you have the weights, this is the scale factor again and this is a, b, l1, l2 and the input is the same. So this is the usual LoRa equation for getting the output given the input. the PEFT weights L1 and L2.



So, now let us do a little bit recap of different style, different data formats again because we will use this in understanding the whole Q LoRa setup better. So, far we discussed FP32, FP16 and INT8. And in FP16, they reduce the exponents and they also reduce the fractions in sort of similar proportion. But what people in Google brain did was that they came up with this new format called BF16, so that its brain float because it is from Google brain and what they did was that they said let us not change the exponent, let us just reduce the precision. So, why is this helpful? If you have to convert an FP32 to BF16, it's so easy.



All you need to do is just truncate it. You don't need to scale or do anything. So, this simplicity actually speeds up a lot of things. This conversion is so convenient and there are also certain operations where this amount of precision does not lead to that bigger gain in performance. It increases the computations, but it does not lead to that bigger performance jump.

So, there are certain training paradigms called mixed precision training. where not all parameters are represented in the same representation, for certain formats you use certain, for certain operations you use certain representations, for certain you do not and this paradigm or fold paradigm is called mixed precision training. So, here also we sort of use some mixed precision. Let us look at how it does. So, let us look at this format, this is rewritten, we just rewritten the first equation here by representing what goes on within Q LoRa.

Most of your operations are going to be in BF16. So if you see this part is in BF16, this part is also going to be in BF16 and L1 and L2 are negligible number of parameters. So it's okay, let them be in BF16 and here the input is also in BF16. So far here, everything is in BF16 and here is the part. where we are going to do double quantization and NF4 quantization.

So, let us look at this in more detail. So, this double quantization can be first let us look at only this bit. This is first we are de-quantizing the quantization factors, the scale factors

and the quantization constants. So, for that the quantization, the second level quantization constants are in FP32. The first level are now represented in 4-bit.

But now once you do the de-quantization of this, you will end up getting the 32-bit precision de-quantized version of the new scaling factors and now again you de-quantize your weights which are in the normal float and then you get back the double quantized the double quantized value and finally you would use that to get the weight matrix in BF 16 which again goes back here everything is in BF 16 and you get Y in BF 16. So, this is the whole crux of QLoRa and this gives incredible amount of memory efficiency. Again, here I've not actually talked about the paging, but what typically happens is that you won't be able to save, let's say, if a certain 65 billion model uses 48 gigs of memory and you only have 32 gigs of memory on your GPU, then there will be a time wherein you will hit and certain matrices will not be available and all the hardware would do is that go replace something that's not used for a long time, get the right matrix that needs to be used into the memory and you will not run out of memory, you will keep running. So, that is not covered in these equations, but that is also a very essential part of why QLoRa should work. So, all these three graphs are 4-bit quantization.

The only difference here is that this is simple FP4, something like FP4. The orange line is the normal float. So, there is a good improvement in accuracy from FP4 to normal 4. But like I said earlier, double quantization does not give you improvement in accuracy as much, but it gives you a lot of space. So, you reduce on storage.

And so, if you use only 4-bit quantization like say float 4, then there is always a drop in comparing to doing the usual one which is using BF16. So, 38 becomes 37, there is a drop here, there is a drop here, but when you use normal float 4 plus again DQ will is not going to help in accuracy but again it will help in storage but adding both you end up getting almost actually slightly more mean than even the vf 16 but that's only a little bit you can for all practical reasons you can assume that it's able to maintain the half precision performance. So in pruning, as the name suggests, it's very simple. All you need to do is just remove certain things from the network. So here, these are the weights.

So unstructured pruning is like randomly keep removing weights from the network with the hope that you don't affect the performance. And structure pruning is more like you take into account the exact structure of the network that you are going to prune and then prune certain structural components from the network. So, I will cover both of them. So, the simplest form of unstructured pruning is called magnitude pruning, just sort the weights, whichever weights are low, just ditch them off. It is as simple as that.

So, what they found is that if you remove 40 percent of the weights, you do not actually see that bigger change in performance, but if you start removing a lot more, then you see a degradation, but what they say is after you prune, if you fine tune, continue pre-training or continue fine-tuning, then you would end up bridging that gap until 80% of the case. And some people figured out that everybody only does weight pruning, you only take into account the weight. of the parameters, only the parameters but activations are also important. So, if you take into account activations, then the product of weights and activations will show you a different story. So, rather than pruning at weight level, prune at activations.

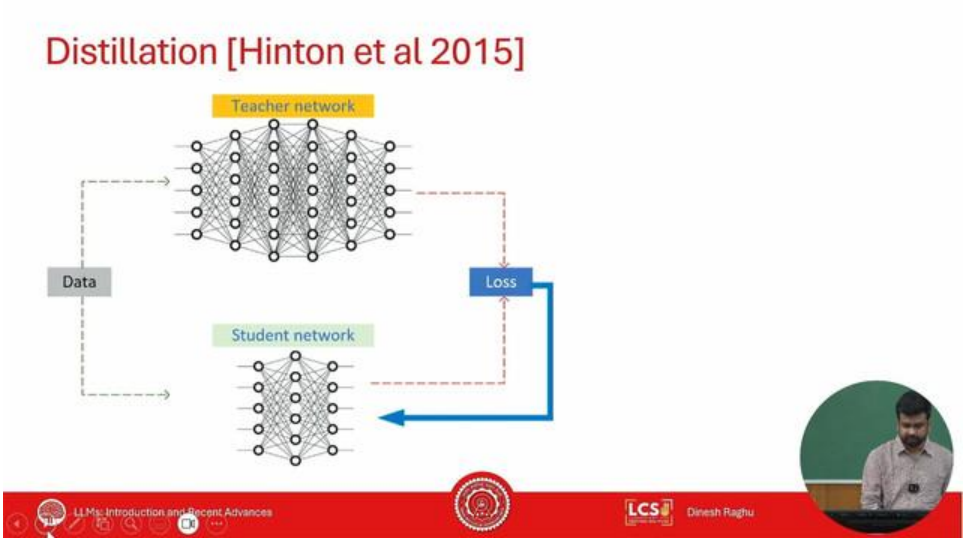
So, this was one idea that was very recently proposed. And so the issue with unstructured pruning is that, let us say you prune, you make a lot of values in your matrix 0. But still if a hardware is not going to utilize that information, it is as much as using that same FP16 or to store 0 as well, it is not going to give you speed up in numbers, it is not going to save compute, it is not going to save energy, nothing is saved. So there is no point in actually doing unstructured pruning if your hardware does not support, so, it is a very important part. So, people started to slightly move away from unstructured pruning and start to focus on structured pruning.

For example, there is this A100 NVIDIA GPUs which says that if you take up any block of 4, you compress them into block of 2 by ignoring 2. This is called 2 is to 4 sparsity pattern. If you do that and you represent your values with half the number of weights and you use 2-bit indices to store the indices which are non-zero, it will give you performance speedup. So, now your tuning algorithm that you are going to design should be aware of this particular hardware choice to sort of knowledge test. So, if you see here, if you do for

any different type of precision, the dense operation and there is a big difference between how much you can do with the dense operation and the sparse operations.

So, because the hardware supports this, now you can think of pruning your weights in this particular format without loss while minimizing the loss in performance. Another recent paper that came out was that, let's say that you have a fine-tuned bert, there are a lot of multi-head attentions, feed-forward network and within multi-head attention, you have multiple heads, Within feed-forward network, you would have certain hidden dimensions and so on. So what they say is that you can prune almost any component right you can remove a layer like for example they remove this feed forward layer they remove this multi head attention you can remove certain heads within the multi head you can remove certain hidden dimensions here and there and they say that you take into account the structure of the network and remove structural components you end up getting performance speed up So this is about pruning. So I am just covering the overview to give a brief picture of what pruning does. I am not going into details or other techniques that does pruning more efficiently.

Finally, I will talk about distillation. So distillation again is a process of training a much smaller model to imitate a bigger model. The bigger model is the teacher and then the smaller model is the student. So the smaller model should end up mimicking what the bigger model does. So it sort of works like this.



One of the biggest issues with distillation is that you need this data. So think about post-training quantization. If you have LAMA 70 billion model and you want to quantize it, you don't need the data with which LAMA was trained for. You don't have to find a proxy for the data with which Lama was trained for. You can just do your quantization.

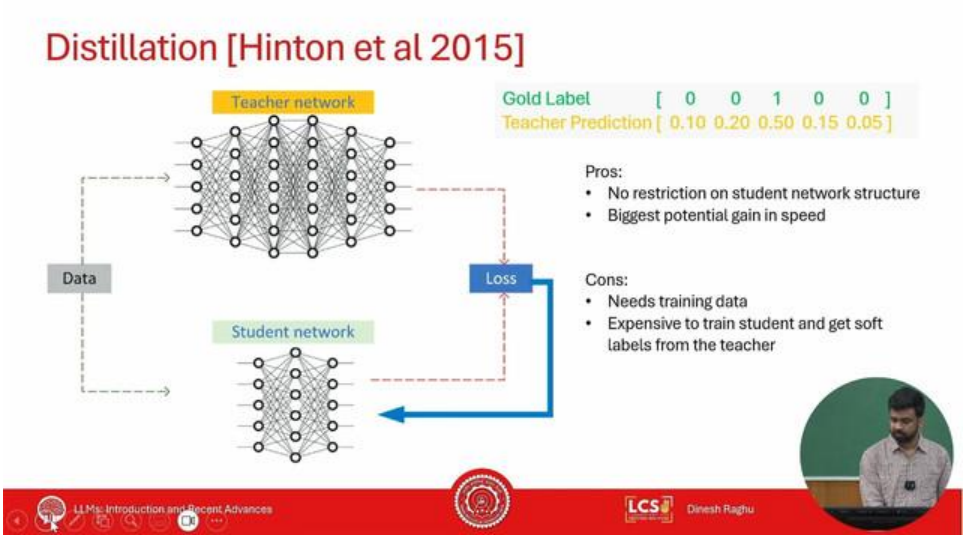
But here, you need the data. If you don't have the real data, you should sort of end up getting some representative data that sort of looks like what this model would have trained on and that is a big bottleneck in this era. Because people are willing to share and open source their models but nobody is willing to open source the data because that's the goal. So nobody would do that. So this distillation is typically difficult from that standpoint and so people do task specific distillation.

You have task specific data, let me distill the model conditioned on the task. So in general distillation is still a useful technique to learn but thinking that I can fine tune or I can distill any big model into a smaller model is not going to be that effective unless you have the right data. So, the way it works is that you feed only the input to the train fully trained bigger model, you get its predictions. So, let us call it y hat. And then you give the same data to the student network let's call it y dash right and now you want y dash to be as close as y hat, both are not good.

But you just want y dash to be as close to y hat as possible. That's all the objective function is. So, there are different flavors of distillation. Some of them are specifically about how you compute this loss. There are different ways of computing this loss.

And some are like you don't just do distribution matching only on the output through distribution matching even in the intermediate layers based on certain structures. There are a lot of variants of distillation. So let's take an example here. So here there are five outputs. Let's assume that they're all going to give us probability and so let's say if the goal label is something like this then your typical loss would have been that you increase the log likelihood of that particular value which is one right that's going to be your usual loss so now for the knowledge distillation your theta is here that is your theta this network parameters are theta t so now in addition to theta you also have theta d but theta d is frozen so what you do is you end up getting the value of every output with Q and you serve that as a reference distribution and compute cross interval.

So, there are several advantages for doing Q rather than again taking only the max value and doing it that is like you can find more details in this first paper that came for distillation. Now, like I mentioned before, the good part of distillation is that your student network can be much smaller. You can have very few number of layers so that you can reduce the latency by quite a bit. Whatever your latency is, you can define your network structure to satisfy that latency. You have that flexibility with you and then you can try to fine tune and then figure out what is the performance gap you get.



Now if you have to get like a lot of speed gain, this is the way to go right, pruning and quantization are not going to get you there. Now again like i said the cons is that you need the training data and it's extremely expensive to perform this type of model compression mainly because you have to do first inference with a lot of data you have to do inference inference is a lot more expensive because it has to do one generate one token at a time, it's extremely expensive. And then again, after you do that expensive task of generating outputs of every data point in your input, you will again have to train the student model from scratch. There are techniques which say like if you initialize your student network this way by pruning the teacher and then using a very aggressively pruned version of the teacher as student, your starting point is much better, you can reduce the time, but still you will have to end up fine tuning. Now, like I said, there are different ways in which Hinton proposed, Hinton et al. proposed distillation. One is called the hard target where let us say that this was the value that got the highest probability. You can forget about the likelihood, just take the argmax and then set it to 1. and use it or the other way is to end up using the entire soft distribution this is specifically found to be a lot more better than simply using hard targets but sometimes when you don't have access to the probabilities. For example, if you are using open API and they are not going to give you, they do give you for some cases, but maybe you are doing some complex things and you are not able to get access to those probabilities, you can still do some sort of a hard target distillation, although it is illegal, but still.

So, what I talked before is more like a single step quantization, sorry distillation. Now, what if you want to distill a whole sequence? which is like in an autoregressive manner or something. So this is our usual knowledge distillation equation where Qs are the teacher likelihoods and then you want to match them, student ones to exactly the same as the teachers. Now in word level distillation, what we say is that we add this particular summation over all the J words and it's simply the same as before. You do it one word at a time by fixing the other ones.

And in case of sequence level distribution, we ideally would want to do it by taking the entire space of all possible sequences but that's going to be like all possible sequences are insanely expensive. The space is so high that it's not possible to even do this. So, what they

typically do is that very badly approximate this into just what the model gives you the highest sequence for a beam search or something like that. And then set that as y hat and simply increase the log likelihood of that the highest sequence that you get. So these are some simple ways of knowledge distillation.

There are also other techniques where the loss function is not cross entropy. They use KL divergence. They use inverse KL divergence. So there are a lot of different flavors of loss functions and a lot of different flavors of how do you approximate the knowledge so one way of approximating the knowledge from the teacher to student is to mimic the whole distribution right. But you're only mimicking the distribution at the output space, but there are like multiple other places where you can sort of fix your network parameter in such a way that you expect the middle layer in this to be replicated by this layer here, something of that type and you can do a lot more and then try to mimic the teacher as much as possible.

So, I just want to make a small connection between distillation in sort of network related literature and a certain other distillation that people are now doing with large language models, which is that let us say if you take the LAMA 405 billion model. So, do you guys know why would somebody use a 405 billion model? It is going to take like minutes to come back with a response. But why would anybody want to use a 405 billion model? It has a much better output quality than any of the models, but then its latency and its cost is so high. Whatever task you use it for, you're not going to get back the money that you put in. But why would still people use it? It will be used as a proxy for humans.

So that is sort of heading towards the right answer. So basically what they say is that an extremely over parameterized model can capture all the nuances in the training data and it can model it better. So once it models it better, you can always distill it. So, the whole point of why you want such a big model is that it will generate output so good that you can use that output to distill a smaller model which would be much better than just training that smaller model from scratch. So, one such way is, the first paper that came out in this paradigm is called self-instruct.

What they say is that, let us say, so seed tasks are something like this, find out if the given text is in favor or against. This is one task and the other task is like give me a quote from a

famous person on this topic and for each task, they give an instruction along with one example and they only take 175 seed tasks, not more. These are handcrafted with one instruction and one instance, that is about it. Now what they say is that they ask the LLM to generate new instruction. Like you have 175, you do a few short example, you give a few short example and then you say generate one more.

It's going to give you one task that is different from the 175 and once you have one task, you again do a check whether it's good or not. If it is good, then you ask it to generate the class label and the input given the instruction. This is output first, input next. And in another case, you do input first and output second. So there are some advantages of doing these.

And what they say is that by doing this, you end up generating so much data. So, this is a few-shot example. So, a base LLM can do. A base LLM which is not instruct fine-tuned can still do this work because they cannot follow zero-shot instructions but they can do a few-shot very well. So, just with one-side-of-a-seed example, you end up getting a lot of information from the model itself on how to instruct fine-tune it.

So, what you end up getting is an instruct fine tune data set. Now, you can use this instruct fine tune data set to actually instruct fine tune the model itself. So, here distillation is coming from gathering as much knowledge from the model as possible. In the other way, how did you gather knowledge? You gave the input and you saw how the model reacts to it. So now these are generative models.

If they are discriminative models, all you can do is only see how it reacts. You cannot ask it to generate new stuff. But if it's a generative model, then you can ask it to actually generate new stuff. And this is a new paradigm and this is sort of like widely used paradigm now for any industrial fine-tuning applications. For example, if you were to create, let's say, a new natural language interface for one of your softwares or something like that.

So you have your own action space about what all it can do, maybe some descriptions of what they achieve. You can actually synthesize a lot of examples of how a person would ask in a natural language to do this execution. And if you have an example for a few, you can end up generating a lot of data from a bigger model. that can do this and then you can use that to fine tune a smaller model and now you have a very easy to use software right.

So there are a lot of good application synthetic data generation and then using it to fine tune a smaller model is sort of the big thing now right.

So distillation is sort of indirectly what led to this whole space being developed and this distillation is possible only if you have an awesome teacher right and that's why people want to build really really big models because you end up getting really good synthetic data and you end up saving cost So, to summarize, we looked at effective ways of reducing inference cost and specifically we looked at model compression techniques and in them we discussed in detail about quantization, we discussed the LLM intake for post training quantization, we discussed QLORA for quantization of a training, we discussed some pruning techniques and some distillation techniques and where the field is moving more. Alright, thank you.