(no title)

86 MIN · YOUTUBE · HTTPS://WWW.YOUTUBE.COM/WATCH?V=PTFIH_BHNJW
https://www.youtube.com/watch?v=ptFiH_bHnJw

SUMMARY

The lecture focuses on the intricate details of language model (LM) architecture and training, emphasizing the evolution of transformer models and the hyperparameters that govern their performance. The speaker discusses the importance of understanding both foundational concepts and recent innovations in model design, particularly in the context of stability and efficiency during training.

- *Overview of transformer architecture, including position embeddings, multi-head attention, and residual connections.*
- *Discussion of modern variants of transformers, highlighting the shift from layer normalization to RMS normalization and the use of gated linear units (GLUs).*
- *Emphasis on the importance of hyperparameters, such as feed-forward dimensions, aspect ratios, and vocabulary sizes, with established conventions for optimal settings.*
- *Introduction of stability techniques, including Z-loss for output softmax stabilization and QK normalization to improve attention operations.*
- *Explanation of multi-query attention (MQA) and group query attention (GQA) as methods to enhance inference efficiency by sharing keys and values.*
- *Recent advancements in attention mechanisms, including sliding window attention and structured attention patterns, to manage longer context lengths effectively.*
- *The significance of empirical evidence and consensus in hyperparameter choices across various models, reinforcing the importance of community-driven research in LM development.*

Um, as you may have noticed, I'm a little bit less uh innovative in my lecturing than Percy. So, you're going to get um PowerPoint slides rather than um executable Python ones, but you should be able to find the PDFs um on the website as well. So I I've titled this lecture everything you didn't want to know about LM architecture and training because we're going to get into some of the nitty-gritty details that I think most other classes uh would spare you the details of you know like what should my hyperparameters be and those kinds of questions um some minor logistics um also if you're doing the assignments we are updating assignments as we find some uh mostly minor bugs make sure you pull uh updates to the assignments um as you go along um Okay. So, what we're going to do, we're going to start with a quick recap of a transformer. Um, and I'll give you two variants of a standard transformer.

One that's, you know, probably coming from the the standard transformer, you know, lectures that you might see in 224n. Um, and then I'll talk about what you implement, um, and kind of the modern consensus variant of a transformer. And then we're going to take a much more kind of datadriven perspective to understanding transformer architectures. So the question that we're going to ask is people have trained lots of LLMs at this point and you can go and read you know all of those papers and try to understand what has changed what has

been in common and from that kind of almost an evolutionary analysis you know try to understand what are the things that are really important to make transformers work right so today's theme is the theme of the class is the best way to learn is hands-on experience but the theme of this lecture because we can't train all these transformers is to learn from the experience of others So the starting point is the original transformer, right? So just as a review, right? Hopefully you all remember this from 224N or your other NLP classes. You know, you've got some simple position embeddings at the bottom. You've got multi head attention, you've got uh layer norms afterwards, you've got a residual stream going upwards, you've got a MLP, and then a softmax at the very end. Um, and we're going to see variance to all these different pieces um until we get to basically the most modern variants um of the transformer and the and the latest one I'll talk about will be just you know a few months before. So what you implemented is not you know the the vanilla transformer variant um from the original paper. Um we've modified a few things you know we've put the layer norm in front of the block. So you can see um on this slide over here that you know there's the norm is over here right before each of these blocks in the residual stream. Um we've asked you to implement rotary position embeddings. Um the feed forward layers use something called a swiggloo. Um and then linear layers um you know now emit these bias terms. Um and you might ask why have you forced us to implement this weird variant of a transformer instead of the original transformer is all you need transformer. Um, and so we're going to go through some of those questions.

And then yesterday I was thinking, okay, I should I should catch up on all the developments that have happened in architectures over the last year. Um, and Percy warned me about this because he said, you're going to have to redo the lecture every year. And so I started looking and I was like, all right, yeah, there's a couple good good papers recently. There's Command A, there's two mode furious, there's, you know, small LM and 54. And then you go looking and you're like, wow, yeah, there's Gemma 3 and Quent 2.5 and intern LM and then there's, you know, more. I can't even sort of you know cover uh the screen with these guys right there's there's a lot of models there were about uh 19 new dense model releases in the last year um many of them with minor architecture tweaks and on the one hand it's kind of annoying to go through all these papers and say like you know what is happening in all of these um but also it's like a actually wealth of information because not all of them do the same thing and you can kind of see you know not all of you can especially in the back can see the details of this slide um but I I put together a little spreadsheet sheet of you know what all these models are doing and starting with you know all the way from 2017 the original transformer all the way to 2025 what the newest models are doing and we'll talk about this as we go but you kind of see sort of certain kinds of architecture changes sort of being explored like so here on this column is position embeddings people used to do all sorts of stuff like absolute relative rope uh there was a sort of um alibi phase for some people but then now starting around 2023 everyone just does rope right so you can kind of see this the convergent evolution almost um of neural architectures and we're going to talk about um all of these different kinds of things.

Right? So the the parts that I'll cover so this is a preview of the three major sections of this lecture and if I have time I'm also going to talk about um different attention variants at the end. Um the first thing is going to be architecture variations. Um that's what I'm going to talk about. So activations, feed forwards, attention variance, position embeddings, all of those things. Um and then having nailed down the architecture, what do we have to do?

Well, we have to pick hyperparameters, right? Like how big do we make the the hidden dimension? How big do

we make the sort of inner projection layer inside of MLP? Um, what do we do about the number of dimensions? How many vocab elements? Those are all sort of important things that you have to choose when you're actually training uh your language model. Um, and you don't want to just sort of pick these out of a hat, right?

You want to select them in some fairly intelligent way. So, we're going to start with um architecture variations. Um and the the two things that I'll you know mention right here and I'll you know go back to them as I talk. The first one is you know there's not that much consensus in a lot of the choices. Um there's been sort of convergent you know evolution in the last few years. Um what I'll call like llamaike architectures at the very bottom here but people do all sorts of things. They swap between layer norm and RMS norm.

They do serial versus parallel layers. There's one choice that basically everyone does es since the first very first GPT. Um, and I'll talk about that in a bit. Um, but there's, you know, lots of different variations that we can learn from here. The big one I've already talked about this guy in 224N. So, if you remember that lecture, this will be review for you, uh, rather than being totally new. I think the one thing basically everyone agrees on and agreed on almost from the very start is the use of pre-norm versus uh, postnorm. Um, that terminology will get a little bit more confusing. Um but the original transformer paper did you know this thing on the left over here where you had your residual stream in the gray um and you know in addition to the residual stream you had these layer norms after sort of every subcomponent. So you would do your multi head attention you would add back to the residual stream and then you would layer norm that and then you would do the same thing with your uh fully connected layer and then you would layerm it. Um and very very early on um people realized that moving this layer norm to the front of this sort of non-residual part so this block on the right um did much better in many different ways and and basically almost all modern LMS that I know of use this kind of porm um there there have been some sort of new innovations recently that I'll touch on in two slides um but lots of you know models have moved to this the one exception um is opt 350M um which I'm guessing, you know, they they kind of messed that one up and and that was sort of orphaned um when they were training. Um that was a fun find in my in my survey of architectures. Um so this pre versus postnorm thing, if you look into why it was originally developed, um the arguments were that, you know, if you wanted to use this postnorm stuff, it was much less stable.

And so you would have to do some careful learning rate warm-up style things to to make it train in a stable way. Um and so if you look at some of the earlier papers you know arguing for this prenorm approach um salar and yen and also this xiang in 2020 paper you almost always see sort of this comparison of hey if we use p-orm and we do some other stability inducing tricks then we can remove warm-up and these systems work just as well if not better um than sort of the you know the postnorm layer norm with careful warm-up type approaches and you see this in in sort of a machine translation setting here Um you see this as well uh on the right um on you know various other tasks especially using BERT which was trained with um postnorm. So um there were many arguments about why this was helpful.

There were arguments about gradient attenuation across layers like if you do pre-orm then the gradient sizes would remain constant whereas if you did postnorm um you know without warm-up then it would sort of blow up in this orange way. It's a reasonable argument, but I think a maybe more closer to modern intuition would be

this argument that um porm is just a more stable architecture to train. And so some of the earlier work by Solazar and um identified all these loss spikes um that if you were training with prenorm kind of in blue here um you would see a lot more loss spikes and the training would be kind of unstable um you know as you were training. So the you see the the gradient norm here you know is spiking and generally higher than the one with prenor. And so today um you see prenorm and other layer norm tricks being used essentially as as stability inducing um stability inducing aids for using large training large neural networks. Um and so this brings us to one new fairly I think recent innovation. I think this didn't exist when I gave this lecture last year. um which is this variant that I don't think really has a has a great name but I'm just going to call it the double norm for the moment here. So this is the original figure that I showed you at the very beginning and we know that putting layer norms in the residual stream is bad. Um but actually someone in 224n this year asked well but why do you have to put the layer norm in the front? Why can't you put it you know after the feed forward network? And of course you can and not only that um sort of recent people have have gone around and just just add the layer norm after the you know the blocks as well. And so Grock and GMA 2 both take this approach of layer norms both in front and after.

Um 2 does only the the layer norm after the feed forward um and the multi head attention. And so this is actually kind of an interesting change. Um porm has just been kind of dominant and the only thing for a while um but things have been changed up a little bit. So now now there's a a new variant and this is actually you know there's been some some evaluations of this kind of approach. Uh people have argued it's a little bit more stable and nicer to train on these uh larger models. By the way feel free to um stop me and ask me questions as well. I have a tendency to to sort of keep going if if no one stops me. So yes uh why is layer in the residual bad? Why is layer norm and the residual bad?

That's a that's a good question. Um I don't think I can give you like a you know this is the proof of why it's bad. I think one, you know, intuitive argument for why this might be bad is that the residual gives you this identity connection all the way from almost the top of the network all the way all the way to the bottom. And so if you're trying to train really deep networks, this makes gradient propagation very easy, right? So there's lots of arguments about how you know LSTMs and these other kinds of you know state space models have difficulty propagating gradients backwards. An identity connection does not have any such problems. And so putting layer norms in the middle, you know, might mess with that kind of gradient sort of behavior. And that you of course you see uh back here right this is exactly the kind of plot you expect to see if that's happening. Okay cool.

Um the other thing that people now do um is in the original transformer people did you know layer norm u and so layer norm is this uh equation over here. What you do is you have you know the activations x coming in you subtract the empirical mean. So that's the average of the x's up top and then you divide by you know the standard de or the variance plus a little fudge factor epsilon and then you square root that so you can roughly think of it as a standard deviation right so that's going to you know standardize your your activations x you're going to scale it up by a gamma that's a learnable parameter and then shift it by a beta right so this makes sense you you're going to normalize you know your activations and then you're going to shift them around to whatever point you want and many models use this layer norm thing and it worked quite well um but many models have sort of now moved on to RMS norm and this is one of the consensus changes like basically all the models have switched to using RMS norm um and now what do you do you just drop um the mean adjustment

so you don't subtract the mean you don't add a bias term um and many notable models do this the llama family palm chinchilla t5 they've all moved to to RMS norm um and what's the reason for this um one reason is that it doesn't really make a difference turns out if you train models with RMS norm does just as well as training you know, layer norm. And so there's a simplification argument. Um, but really I think the argument that's often given um in these papers and I think it's good to appreciate kind of the details of this argument is that uh you going to RMS norm is, you know, it's faster and just as good. So in what way is it faster? Well, if I don't subtract the mean, it's fewer operations. If I don't have to add that bias term beta back, it's fewer parameters that I have to load from memory back into sort of my compute units, right? So I don't have to you know retrieve these this sort of state. Um and some of you might be thinking but wait you told me in 224n that nothing but matrix multiplies matter for the purpose of runtime right and this is not a matrix multiply and so I shouldn't care about you know any of this and that's a reasonable perspective to take if you think about you know the number of the percentage of flops that is taken up by different operations in a transformer. um this table um there's a nice uh paper by even all in 2023 um I think the title is like memory movement is all you need or something that does profiling of all the different components of a of a transformer and you see that you know tensor contractions which are like matrix multiplies that's like 99.8% 8% of the flops um that happen in a transformer. And so, you know, saving 0.17% of your flops doesn't seem like a like a huge win. Um but I think one of the things that's important for architecture design now is to not just think about flops because you know flops are important, but that's not the only resource that you have to think about. Um it's also that you have to think carefully about you know memory movement. Um and so even though you know tensor contractions so this is things like matrix multiplies that's like 99.8% 8% of the flops. You know, if you have things like the softmax operation or layer norms, all these like normalization operations that happen um in a transformer, they're 0.17% of the flops, but actually they're 25% of the runtime. And a big reason for that is because you know these normalization operations still incur a lot of memory movement overhead, right? And so it does actually matter to try to optimize some of these like lower level things because it's not just about flops. It's also about memory movement. I'm going to emphasize this quite a bit more as I get into the systems lecture. Like when we talk about GPU architectures, it's going to become very very very important to think about memory not just about flops.

And so this is one of the reasons um why RMS norm has now become sort of um much more popular. And so I I went back and looked at some of the earlier uh RMS norm papers. I think the the sad thing is that there aren't quite as many papers published by industry labs with you know big nice ablations. And so many of the ablations that I'll show you are going to be from from a couple years back. Um but Nang at all in 2020 had this very nice ablation showing you know here's the vanilla transformer here's the RMS norm version and you kind of see the exact thing I told you. you know the the number of steps per second that you can do in a vanilla transformer 3.5 per second with RMS norm you get 3.68 68.

You know, not a huge gain, but that's in some sense for free. And you get, you know, a final loss that's uh lower than the volul transformer. So that's great, right? In some sense, we've gotten uh runtime improvements and we've gotten uh in fact, at least in this case, loss improvements. And so that's a win-win um for us. The final thing that I'll say which is very much in line with this RMSORM thing in terms of theme is that most modern transformers do not have bias terms. Um so the original transformer if you look at the the FFN um will look something like this right you have your inputs X you're going to do a linear layer with a bias term and then you'll relue it and then you'll have a second linear layer wrapping around it. But um most implementations uh if

they're not gated units, which I'll talk about in a moment, uh look actually something like this. They've just dropped the bias terms. You can just make this argument from basically the same kinds of underlying principles. You know, they perform just as well. Um matrix multiplies are apparently um all that you need to get these guys to work. Um and the other thing which is maybe more subtle is actually optimization stability. Um I don't quite have the deepest understanding of why the bias terms are particularly bad for stability. Um but there's been sort of really clear empirical observations that people have made that basically dropping these bias terms often stabilizes uh the training of these largest neural networks. And so now a lot of the implementations now emit bias terms entirely um and train only on these like pure matrix multiply um kind of settings. So that's the that's the layer norm bit. Um, and so there's kind of two things that, you know, you should kind of think of. This is nice because the story is pretty clear. Everyone does something and so you should just kind of know this, right? Basically, everyone does porm or at least they do the the layer norms outside of the residual stream. Like that's kind of the iron rule, right? Um, you know, you get nicer gradient propagation, you get much more stable training. Um, it just doesn't make sense to do it the other way. Um, most people or most almost everybody uh does RMS norm. um pra in practice it works almost as well has fewer parameters to move around and this idea of dropping bias terms just broadly applies a lot of these models just don't have bias terms um in most places um I think the one exception to this RMS norm one as I was reading yesterday uh is I think coher both command and R plus use layer norm quite sure why okay any questions on kind of the layer norm rs MS norm and bias term stuff before I move on yes questions Do you think there are some long-term lessons you can take away from these details that are more future proof potentially or do you think these are Yeah. So the question was is there is there something more future proof and I think it's hard to have like the the biggest picture in in many ways deep learning has been very empirical and like bottom up rather than top down. But I do think there's some generalizable lessons that you could sort of draw from here. I think the lesson of you know have very direct identity map residual connections is sort of a story and a lesson that has played out in many many different kinds of architectures not just you know in these kinds of architectures. Um the effectiveness of layerorm we'll see once again later on in this lecture has been very effective and so not letting your activations drift in sort of scale is another thing that I think generally has been very effective for training stability. Um those two seem like fairly generalizable lessons. um we will also kind of see sort of sort of the systems concerns come into play again. So this is another generalizable lesson of sort of thinking really carefully about the impact of your architecture on the systems components of your of your uh design. Okay. So now um there's this other component which is the activations um and there is a whole big zoo of activations um relu swish lu glu and then there's I mean these aren't activations there are different kinds of mlps uh galu regul swigloo and lilu um and yeah I think this is exactly the kind of thing that I didn't originally want to learn when I got into doing deep learning I was like I don't care about activations it's going to train anyway Okay. Um, but it really does matter unfortunately um for both you and me that swiggloo and other glu variants just consistently work well. And so I will explain those to you and you should think about them carefully because they do work um and internalize that, right?

Um so I think the relu and maybe the galu you all should already know, right? The relu you learn in like some of the the most basic deep learning classes, right? You just take the max of zero and in the case of an MLP, right? You've got your I've dropped the bias terms here. You know, xw1 you take, you know, the relu and then you do w2. Fairly easy, right? Uh a gel is a gausian error linear unit. Um this one multiplies um the linear with a cdf of a gausian. Um and so it's basically going to be like the relu but with a little bit of a bump here. Hopefully you can

see that um over here. This is not just flat at the very bottom. Um this makes things a little bit more differentiable which may or may not help. Um and the GPT family of models um 123 and GPDJ and so on uh all use the GLU. Um and the original transformer and some of the older models used uh the relu and really almost all the modern models have switched to uh the gated linear units like swiggloo and the geekaloo and and others, right? Um, and really I think this is, you know, the the Google folks really pushed for this like Palm and P5 and others. Um, but since it's sort of been tried and true, basically almost all the models post 2023 um, use a gated linear unit. And so, you know, going back to that earlier question of like what generalizable architecture things can we learn from this lecture, you know, there are some things that have really consistently been very useful. residual connections, layer norms. Um, gating is yet another one, right? And so this is another place where gating just appears and is a very good way of doing things.

So originally we this is our our fully connected layer right here, right? This is with a relu. Now instead of doing just linear and a relu, what I'm going to do is I'm going to gate you know the output here with a entry-wise linear term. So x.v V is going to give me a vector and I'm going to multiply that entry-wise with my original inside term of the MLP and then I'm going to multiply the whole thing with W2. Right?

So the way to think about this is I've gated sort of the hidden part of the MLP. Right? So I've got my original activation that takes my inputs and puts it into the sort of hidden space and then I'm going to gate that with X.V. And then, you know, I'm going to project that back into sort of the the hidden dimensionality using W2, right? So, there's this gating operation that happens entry-wise. And that's really, you know, the the basic thing that's happening here. And this is the the GLU plus the reloo. And then we have an extra parameter that we've added here for the gating. This is V. Um, and so when someone says something like, oh, it's a a giggloo uh fully, there's nothing to laugh about that. there's the gigglu fully connected layer. Um what I've got here is you know I've got the the gel sort of for the nonlinearity and I've still got the exact same gating here of x.v V right and this is the the architecture that was used by many of the um uh Google models like T5V1.1 um gamma 2 gamma 3 um and then uh another variant there's a swigloo and this has been very very popular uh swish is x times the sigmoid and this is the nonlinearity and you can kind of you know a sigmoid is like this and x is like this so it will look you know just like the gausian error unit um and then you know you do the same thing here you have a gating over the switch and then you get a fully connected layer here.

Yes, I have a question. Below a certain negative value, the switch function and also the also the the G function it's not monotonically increasing. In fact, it's decreasing, right? And a lot of the argument about how gradient descent works in like input machine learning is that like okay, you want to do gradient descent click but here it seems like it would go in the opposite direction if you use gh or or switch or their gated versions. So yeah, so the question was, you know, this isn't monotonically decreasing. You know, there's a there's a bit on the very left of this zero here that's kind of flipping in the in the derivative. Um, and isn't that going to be a problem? Um, I think intuitively you could you could have argued that this would be a problem. You might trap a bunch of activations at zeros. Um, I think in practice, you know, if you look at kind of like neural network optimization dynamics, what's actually happening is often you're throwing very high learning rates with momentum into the optimizer. And so you're not really going to converge to this zero point, right? Like these activations are going to be all over the place. Um, and so in practice, I don't think this this little tiny negative piece is really an effect that's going to be huge for the model, if that makes sense.

Um, okay. And then going back to to this, uh, the Swiggloo is is basically most models today, like the llama family, Palm, Elmo. Um, and I'll show you the the big table later. Um, but you'll see that the Swigloo is is very very popular. And one thing to note, um, I'll talk about this again in the hyperparameters part is, you know, now remember I've added this this V term, this extra parameter, right? And so I want to, you know, think about how to size this extra parameter. And what people do is gated models usually make this like hidden size, you know, the basically output dimensionality of W slightly smaller by a factor of 2/3 um in order to make sure that the total number of parameters of this whole thing remains the same as the non-gated counterparts. And that's a convention thing that most people do. Um if that you don't quite understand what that is, I'll go back over that again later. But you can just kind of keep in mind that basically for the gated linear units you just make everything a little bit smaller to make sure things are remain parameter matched. So oh yes question this may be obvious in the past. Uh, one of the benefits of relu is like it's very easily differentiable by the input. But if you know if you have the derivative of the cdf of the gausian, you have like a squared with x, does that not really slow things down? That's a that's a very good question. I'm not 100% sure what the internal like CUDA implementation of the swigloo or the the galu gloo is. I think it's entirely possible that like internally they might be implemented with like lookup tables.

Go ahead. I mean what really matters is the memory pressure here and like it will be the exact same because you're reading the same amount of elements performance. So the the extra comput is negligible on that's actually a yeah that's probably a better uh argument that like basically flops wise this is negligible anyway and that actually the memory calculus is the same. So okay cool. All right so uh do gated linear units work? Uh I will have more modern evidence for this as well but I thought you know I should take you straight to the the horse's mouth uh Nom Shazir's original paper um where he you know evaluates all these GLU variants um and you know this is this is somewhat older stuff. So you're seeing cola and SST2 performance um but you do see basically that the GLU variants consistently perform better right glu is 84.2 84.12 84.36 84.67 67. Um, and you know, wow, it's 2020s. They they even give you the standard deviations so you can sort of figure out how significant um those results are. And they they in fact um are significant, right? Um and so this is some nice evidence to to see here. Um there was also you know the Nang at all in 2020 paper which is a very nice paper studying all sorts of architecture variance um I think in the context of T5 style models. Um and once again you see that the the gated linear unit variants um consistently achieve kind of lower losses um than their counterparts, right? Like you see that the bolded lines are exactly at the GLU variance.

Um and this uh pattern has basically held up. Um so for gating and activations, you know, there are lots of lots of variance um across different models. Um but the gated linear unit has become basically widespread and dominant and I think for good reason. Um, of course, um, the GLU isn't necessary for a good model. Like, it's important to separate the two, right? Just because it's probably the slightly better and everyone does it doesn't mean it's necessary. Um, and you do see examples of very high performance models not using a GLU. Like GPT3 uh is one example. Uh, more recent one, um, Neotron 340B uses a squared relu, which I had not seen before, and Falcon 211B uses a RELU. uh both of those are relatively high performance models. So you can kind of see that it's not really necessary and so you know evidence does point towards consistent gains from uh swiggloo and gaggloo and that's why we ask you to implement exactly uh that variant.

Cool. Okay. Um the final thing that I want to talk about for architectures and this is one kind of final major I

want to say variation that we've seen. Um normally uh the transformer block is serial right in the sense that you know uh your your for each block the uh outputs come in from the bottom and then you do your attention and then you pass the result of that computation forward and then you do your MLP and then you pass that computation forward, right? Um and so this is inherently serial. You do attention and then MLP. But of course this might have certain like parallelism constraints. So if you want to paralyze this over gigantic you know sets of GPUs it might be harder to do so um if you have this serial connection you know the systems concerns might also be more difficult right you might get lower utilization from your GPUs and so a few models have done this thing uh that I'll call parallel layers um where basically instead of having serial computation of attention and then MLP they will do them both at the same time right so you will get your X you know from your previous layer you will comput both the MLP and the attention side by side and then you will add them together into the residual stream and then that will be your output right um and this was pioneered uh by GPTJ which is kind of this open source replication effort and uh the folks uh at Google doing palm were kind of bold enough to do this at at the really big scale um and many others have kind of followed since um so if you're implementing this right you can share a lot of stuff like the layer norms and the matrix multiplies can get fused together and you can get some systems efficiency um out of that it hasn't been quite as popular since then at least in the last year. I think most of the models that we've seen have been serial layers rather than parallel ones. Um I think the only exceptions to this are like coher command A, command R plus um and Falcon Q 11B. So now I think we have the ability to kind of go back to you know this big, you know, hard to see chart and then see what what I was sort of pointing at at the very beginning. So this column here, you know, you don't really need to be able to read any of the text because think the colors will tell you everything you need to see. This check mark here, this is basically pre versus postnorm. The only two models I I really know of in the the early days uh that did um postnorm, this is the original transformer and GPT and BERT if you want to include that into this table. Um and then almost everybody else, I think basically everyone else um has done uh porm. The only other non-checked boxes here are models that are proprietary and I don't have details for. Um, this column here on the on the leftmost thing, this is RMS norm versus layer norm. The gray boxes are the layer norm.

The blue ones are RMS norm. Basically, most people have converted to RMS norm. As I said, um, this column next to it is serial and parallel layers. Once again, most people do serial, but you see other variants. Um, what I'm going to talk about next is going to be position embeddings, and that'll be kind of more interesting in a moment here. Um any questions about any of this architecture stuff before I uh move on? Hopefully that gives you a bit of an overview of at least the major variations in architectures that that we see.

Yes. Serial layer or computation more efficient than parallel. So uh the question was whether serial is more efficient than parallel. Um it's it should be the actually the reverse that parallel is more efficient than serial and that's why you're kind of willing to do this. So in some sense you might expect serial to be more expressive because you're composing two computations rather than just adding them together. Um but the benefit of parallel in theory is that if you write kind of the right kinds of fused kernels, a lot of these operations can be done in parallel or the computation is shared across the different um parallel parts. Okay. So cool. Um, so the last thing uh I want to talk about in architecture land, I think this is the last thing is uh variations in position embeddings. Um, and I think this one's interesting because in the first few years of of sort of LM land, there were a lot of different things that people were trying. Um, sign embeddings were from the original transformer. You know, you should have learned this in 224n.

There's sign and cosine positions. Um many others did absolute embeddings like the GPTs and OPT all basically just added a position learned position vector to the embedding. Um some others like T5 um and Gopher did uh various kinds of relative embeddings that add vectors to the attention computation and then I think most models have converged to rope um which is you know relative position embeddings um and this I think actually started in GPTJ once again another open- source contribution um and has really rapidly been picked up by most of the models um and so the highle thought process behind rope is that the thing that matters is relative positions uh of these vectors right and so um if I have an embedding f of x of i where x is you know the word I'm trying to embed and i is my position then I should be able to write things down in this way right so there should exist a f such that f of x i and f of yj if I take the inner product of these embeddings then I can write this down as some different function g which is a function of the two words and the difference in their positions Right? So this is a a definition that enforces um basically uh position invariance or absolute position invariance. So you only pay attention to the how far apart these two words are.

Um and so you can you know do a brief check and see okay what happens with signs? Well you get these cross terms that that are not relative. So you do still leak absolute position information. Um absolute positions like it's in the name you know it's not a relative uh position embedding. And um relative embeddings um well it is relative but it's not an inner product. So it sort of violates this constraint. Um and so rope is this kind of clever observation that we do know one thing that is you know invariant to um sort of absolute things which is rotations. And so we're going to exploit that structure to come up with our position embeddings.

Um right we know that inner products are invariant to arbitrary rotation. So we're going to leverage that. So on the left, this is the starting point. Let's say my my embedding for the word we is this arrow over here. And my embedding for the word no is this other arrow over here. Now I want to embed uh this sequence. We know that. And I only, you know, I look at the word we and no. So how do I do that? Well, we is in position zero. So I'm not going to rotate that guy at all. Um no is in position one. So I'm going to rotate him uh by, you know, one, you know, unit of rotation. And so now I have this embedding for we know. And now let's say I want to embed this sequence. Of course we know. Now we and no are have the same relative positioning to each other. And so let's look at what happens. Wei gets shifted by two positions. I rotate we by you know I start you know in this vertical position and I rotate them twice. One and two. And then I rotate no by three positions because it's 1 2 3 uh sorry 0 1 2 3 position. Right? And so now if you look at these two arrows, they have the same relative angle, right? So their inner products are preserved. And so this is kind of the the nice fun idea about rope. You just rotate the vectors and the rotation angle is determined by the position of each word. And rotations, you know, the inner products don't care about relative rotations. And so these inner products are only going to look at sort of the the difference in distance. Now it's easy to think about in 2D because rotations are kind of obvious in 2D. There's only one way to rotate a vector. Um but in highdimensional spaces where we operate, it's not obvious at all how we are going to do this rotation. So the rope folks came up with you know in some ways the simplest but also effective way of doing this. And the way to do it is you take your highdimensional vector in this case D and I'm just going to cut it up into blocks of two dimensions. And every two dimension is going to be rotated by some theta. So there's going to be a rotation speed. Um and I'm going to rotate the pairs of dimensions. Um and so now every pair of dimensions is encoding, you know, all these relative positions. And much like in s and cosine embeddings, I'm going to pick some set of thetas such that some embeddings are rotated quickly and others are rotated much more slowly. So they can capture both high frequency information or like close by

information and very far away uh sort of lower frequency positioning information, right? Um and the actual rope math here is, you know, if you're going to think about rotations, it's just going to be multiplying with various sign and cosine rotation matrices. Hopefully you remember this kind of from linear algebra and trig. Um and so you can think about this as an operation where you multiply you know your embedding vectors with these you know block 2x two block matrices. Um and there's no sort of additive or cross terms that sort of appear here. This is all purely uh relative. Um one thing that is different um if you're used to sort of absolute position embeddings or sign and cosine embeddings here is that um the rope is going to operate at the actual attention layer. Right? you're not going to add position embeddings at the bottom.

Whenever these attention computations are going to be done, you're going to intervene on that layer and then that's going to give you your position uh information. And so, you know, I pulled this from, I think, the llama implementation of rope. You know, you've got the initial normal attention stuff at the very top like query keys and values. These are, you know, your normal linear projections. Um, and then, you know, you're going to come up with cosine and s angles. These are rotation angles telling you how much to rotate different blocks of um the uh query and key. And then so you take your query and your key and you're going to rotate them by the cosiness and signs. And now you've gotten rotated query and rotated key. And that's going to be what's going to go into the rest of your attention computation. Right? So you don't do this at the bottom, you do it whenever you generate your queries and keys.

Hopefully that's that's clear. um that's really critical to enforcing um kind of this uh relative positioning only um information. Okay, good. So um one of the things I want to highlight is that rope is actually one of the things that it seems like everyone has converged on. I I you know went through all 19 of those papers um over the weekend and basically all of them now use rope um for various different reasons. there's you know the reason that rope has now many different algorithms for extrapolating context length and that's an important part of sort of the modern productionized language model um but also it seems to be empirically quite effective even at fairly small scales in small context length so it's kind of won out on this um uh what's it called position embedding battle okay um any questions before I move on to to some of the hyperparameter stuff yes is the rate of rotation consistent across all these models um I don't think they're all the same there's some variation in the thetas.

Oh yes. Are the the thetas like for each pair um are those hyperparameters or are they trained? They're not. It's the thetas that determine the rotation angles. They're not hyperparameters. Much like in the in the signs and cosiness here there's kind of a schedule to the rotation angles that are determined and it's in the same intuition in the signs and cosiness. You want to cover different frequency ranges um in order to get higher or lower uh frequency um information.

Yes. Oh, the rotations create any difficulty with like training. I wonder like this like angular rotations. Um the rotations themselves don't really create any issues because one way of thinking about a rotation is that it's just a matrix multiply, right? Since thetas are fixed, right, and the M's here are fixed, this is really just a fixed matrix that multiplies your vector. And so in that sense it's not really an issue. If you were learning the thetas then maybe you have issues because you're you know maybe differentiating through trig functions but you're not

doing that here.

So okay cool. So now I think we go even one more level uh into the details here. Uh and we're going to talk about hyperparameters. Um I feel like when you have to you know you're dropped in and you're asked to train you know a new language model there's a lot of questions you have about hyperparameters because there's quite a few of them. And one of the things that I've realized is that actually only a few of these really get changed um across different successful models. There's actually like fairly clear rules of thumb and fairly clear guidelines that people seem to be following. Um so you know there are some things like how much bigger should the feed forward size be or how many heads should I have or what should my vocab size be? Um and so we'll talk about each of those things and we'll try to constrain the space of hyperparameters that people um have. So you know the starting point we're going to look at a simple feed forward layer you know just the you know with the bias let's say um this is a relu version of it and so there's two hyperparameters here there's d model which is the dimensionality of x right that's the input coming into your your MLP um and then you've got dfff so this is the feed forward dimension this is kind of the the output hidden dimension of your MLP and from there you're going to project back onto D model right so what should um DFF be um in general eneral um you know these these things are going to be up projections right you're going to have more hidden units than there were inputs um but how much bigger well there is actually just like a consensus um almost everybody that uses you know relu style uh MLPS are going to pick DFF is equal to four times um D model um this is I will show you some empirical evidence for why this is a sane number later um but as far as I can tell there's no like you know law of nature that says you have to pick four. This is a convention that has really held up. Now there are a few exceptions to this rule. Um remember that the GLU variants are going to scale this down by a factor of 2/3, right? And if you scale it down by a factor of 2/3 um you're going to have uh roughly the same number of parameters. Um, you can do a little bit of math and if you scale the GLU variance down by a factor of 2/3, you'll come to the conclusion that the way to do that is to set DFF equal to 8 over3d model, right? That's going to be the number that you end up at. And you can sort of convince yourself that that will give you the same number of parameters and that's the ratio that you would get if you started with a ratio of four. So if you look at many of the models, they actually do follow this rule of thumb. um palm for example uh you know are palm mistro and llama are slightly larger these are glu models but they don't follow this 2.6 rule but if you look at for example llama you know one quen deepseek e and t5 they all roughly follow um this like kind of 2.6ish rule um and I can sort of put up the uh the big table of lms that I made later with hyperparameters many many many of them fall into this roughly 2.6 six range and that's the standard parameterization of a GLU uh unit. Um I'll go through one other exception. I really like this exception because I think in many ways, you know, uh big large language model training is a game of copying hyperparameters from other people and so we don't learn very much, right? Like it's very conservative. Um but T5 I really like because in some sense it's really bold.

Um and I think Google people actually do some pretty bold stuff. Um, and so if you look at the 11 billion parameter T5 model, they have a pretty pretty incredible setting. Their hidden dim is 1024, but their DFF, you know, their up projected dimension is 65,000, right? Um, and so that's going to give you a 64 times multiplier um on the ratio of DFF to to uh D model. And of course, you know, you compare to this where Palm is like a factor four and everyone else is, you know, much smaller. This is this is a very large difference. Um and there's some other recent examples of of using much bigger um you know multipliers like gamma 2 kind of follows in these footsteps um and does a factor of eight.

And I'll talk a little bit about this exception um later. Of course T5 was a totally fine model. So this should tell you it is possible to train a model with you know such a uh much larger ratio. So one of the things that I think is you know quantitative evidence you know I saw that 4x multiplier and I thought is that really the right thing to do or is there some more quantitative experiment someone's done to convince me that that is a good idea. Um so one of the figures from Jared Kaplan's sort of scaling law paper and most people know this paper for for the scaling law component but actually there's also some really useful hyperparameter components to this paper.

um you'll actually see that they do exactly this thing that I'm talking about the DFF to D model ratio. Um and they plot essentially how much the loss increases as you vary this and um you kind of see that there's kind of a sweet spot. This is, you know, a ratio of 1 2 3 4 and then up to like 10 or so here, right? And so there's a pretty wide basin here anywhere between 1 to maybe up to 10 where you know you can pick whatever feed forward ratio you want and it'll be roughly optimal. Um, and four is not too far off from your your optimal choices over here. It's like one, two, three, four. It's like right here or maybe right here, right? So that's that's a pretty reasonable choice.

Um, so what can we learn from all this hyperparameter stuff? I think a lot of the evidence points towards, you know, you can pick the same defaults of, you know, if you're not using a GLU, you can multiply by four. If you're using a GLU, you can use roughly 2.66. Um, and they can work pretty well for mostly all the modern LMS. Um, T5 once again does show that you don't have to follow these rules, right? You can be a rule breaker and do whatever you'd like. um there's no hyperparameter choice written in stone. You can get reasonable LMS at many other hyperparameters. Um that said, I think the the really funny epilogue to this story, right, is that P5 has a follow-up model called P5V1.1 um that's improved and it uses a much more standard 2.5 multiplier on gaggloo, right? So, you know, you can read between the lines and say like maybe they looked at, you know, the original T5 and said actually maybe we want to walk back that 64 times multiplier and pick a more standard one. and they did end up with a better model. So cool.

Yeah. Okay. So So I think that's a that's a good question. So the the question was what's the ratio or sorry what's the relationship between you know this ratio that I'm talking about here and generally the impact on the model right? Um and so if we go all the way back here uh here, you know, the ratio is controlling essentially how wide, you know, the the hidden part of this this MLP is. And so the original justification in the T5 paper for for picking 64 was to say actually we can get bigger and fatter matrix multiplies if we make that dimension really really large. And while that is a kind of a true statement, you know, the wider it is, you know, you're getting more parallel computation, so to speak, rather than serial computation. So you're spending your flops and your parameters in a slightly different way than if you made your hidden units bigger, which would let you pass more information or using more units, which would let give you sort of more serial computation, right? So you're spending your parameters and your flops in a in a slightly sub-optimal way from expressive power, but you might get it um get systems gains if sort of your your matrices are wide enough.

Okay. Excellent. So another thing that is a is a surprising or maybe not surprising um consensus hyperparameter is the um ratio between the model dimension um and the head dimension times the number of heads. Um, so I I clipped this from 224N, right? But really, um, the basically canonical choice is to pick things so that the

dimension D, that's a hidden dimension. And if you have multiple heads, you're just going to split up the number of dimensions each head gets, right? So you're going to keep the dimensions fixed um, as you add more heads. And you don't have to do that, right? As you add more heads, you could just keep the same number of dimensions per head, and you could just let the attention part take more and more parameters, right? you could do that. That's an option that you have. Um but most models once again do follow this guideline. Um we see GPT3, T5, Lambda, POM, and llama 2. They all have a ratio of one or almost exactly one. Um T5 is the one exception uh that breaks this rule. They tried the the big ratio of 16. Um but otherwise it is all, you know, fairly following this consensus.

There's been a couple papers that have argued against this 1:1 ratio. Um, you know, there's a a notable one by um I don't know how to pronounce this, Boja Panelli at all 2020 um who have argued that, you know, if you have um more and more heads, they're going to have lower and lower rank. Um, and if you have very few dimensions per head, that's going to start affecting the expressiveness of the attention operation. Um, but in practice, it doesn't really seem like we see too many significant low rank bottlenecks in practice. Um, and most of the models with this this ratio of one seem to do just fine, right? This is really a parameter that's generally been held constant by most of the models um that we've seen. If I have time, I'll talk a little bit about different optimizations that people have made on this like multi head component. Um but hyperparameter rise things have have stayed fairly um similar. I think one of the the big ones in terms of hyperparameters is the the aspect ratio. Um so you know we can think about deep networks, right? We can have more and more layers or we can have wide networks. And generally if you want one knob to control the width that would be uh sort of the the hidden dimension of the residual street, right? That would control essentially the width of almost all the operations um at once.

And so this seems like a pretty critical thing to tune. You might think that deeper networks are smarter and more expressive or wider networks are more efficient. Um there is generally a sweet spot um of ratios that people have picked. Um there have been sort of outliers. Some of the early models used much smaller ratios here. So what that means is that um they were much uh much wider than they were deep. And then some models have gone really deep um where they had way more sort of D sorry the other way around really wide where they had way more D model than N layer. Um and there's been generally a sweet spot of saying we want about 128 sort of hidden dimensions per layer. Um, and that has been generally stuck to by a lot of the GPT3 and llama variant models. Um, and I'll talk a little bit about evidence for that in a second. Um, there's considerations about aspect ratio that are quite important. Um, they will control the amount of sort of parallelism um that we can do. So, if you're doing um something called uh pipeline parallel, what you're often going to do is you're going to take your different layers and you're going to cut them up and you're going to put them on different devices or different blocks of devices because you'll parallelize, you know, within each layer as well. Um and so there's going to be certain kinds of um constraints that you're going to put on your model. And also, you know, if you have really wide models, then you can do something called tensor parallel where you slice up the matrices and then you distribute those on GPUs. And one thing that we'll learn in in I think uh 1 2 3 four or five lectures is that these different parallelism paradigms are going to have different constraints right you need really fast networking uh for tensor parallel and you can sort of maybe get away with slower networking or higher latency networking for pipeline parallel and so your networking constraints might in turn drive some of these like width depth considerations.

But setting that aside, you might abstractly ask, you know, what is the impact of aspect ratio model performance? And once again, uh, Kaplan, uh, at all have a really nice visual sort of aid showing how aspect ratio impacts performance. And so this is three different scales, 50 million, uh, 274 million and 1.5 billion parameters. And the x-axis is axis effect ratio, y-axis is sort of loss difference, um, in percentage change. And you see that you know around 100 right which is once again I told you was the around the consensus choice of hyperparameters is the minimum across different scales right so this is kind of backed by some of this like uh large scale uh hyperparameter data that's been published by Kaplan at all and it roughly matches that intuition and a really nice thing here is it seems to be the case that aspect ratio optima does not shift too much uh across uh several orders of magnitude here so if this holds up even more that's that's very good news you can keep training um on uh one fixed aspect ratio. Um one thing I will note um that is quite an interesting result is um EK um and others at Google had this very interesting paper sort of studying impact of depth versus width um both upstream and downstream. And one of the things that they found was that if you're looking at um losses then it doesn't really matter. Parameter is the only thing that matters. Deeper models don't help you. Um, but the story is less clear if you're looking at downstream accuracy. At the time, they were looking at sort of fine-tuned superlue accuracy. They were arguing that for the same amount of flops, deeper models might be better. So, I'll sort of just leave it at that. There's not quite as much follow-up um to this work, at least in the open, that I've seen, but downstream performance may actually be slightly different in terms of the the aspect ratio considerations here.

Okay, cool. Um the final thing um that I want to talk about in this sort of very low-level hyperparameter world is what are kind of the vocabulary sizes that you might want to pick. Um and in general vocabulary sizes have been trending upwards. Um and I think a big part of why is because you know LLMs are being deployed out in the wild. They're becoming more useful services. And when that happens, you're going to interact with people speaking different languages, um people using emojis, all sorts of other kinds of, you know, almost modalities or languages um than what you might expect. And so I think some of the earlier models and especially monolingual models um ranged around in the 30 to 50,000 uh token vocabulary range. Um you can kind of see this in like GPTs, the early llamas. Um, but if you look at uh the multilingual or I would call like production systems um that have come out, they've all sort of been shifting towards the 100 to 250,000 uh range for their vocabulary sizes. Um, and you know, I I looked at Command A, which is one of Coher's models. They're a company that emphasizes a lot of multilingual stuff. You know, you see very large uh vocab sizes from them. um even with GPT4 and and many others that have copied the GPD4 tokenizer are going to be around the 100k tokens, right? And so that's kind of the the the standard that a lot of people are operating at roughly at 100k to to 200k uh token size. And I think there's been work showing that as models get bigger um these models can in some sense handle more and more or make good use of more and more um uh vocab elements. And so you might see, you know, increasing trends uh to token counts as models get scaled up or more more data is used to train them. Cool. Okay. So the last thing um this is no longer sort of specific hyperparameters but sort of two other things that you might need to do before you sort of set your model to run. Um which is dropout and other kinds of regularization, right? And I think this one was really interesting to me when I was originally doing kind of the research for putting this lecture together. And if you sort of think about pre-training, pre-training is about the furthest place that you might think of from regularization, right? Because pre-training you do usually like one epoch, right? You you can't even go through all of your data because you have too much of it. So you're going to do one epoch training um and you're almost certainly not overfitting the data in that one pass that you're doing,

right? And so you might think, all right, we don't need regularization for pre-training, right? Let's just set your your optimizer loose. It's all about minimizing loss. Um, and this is this is really good arguments for for why you shouldn't need to to regularize. But then if you look at what people do, um, the the story is actually kind of mixed.

Um, and this story actually is is maybe even more mixed than than what has uh turned out to be, but you know, early days people did a lot of dropout. Um and then you know there's a lot of weight decay that also seems to be happening. Um and these days I think a lot of the the people have stopped publishing details on precisely their training hyperparameters but dropout has sort of gone out of fashion but weight decay has really been something that a lot of people continue to do. Um and why is that? That's like a a really odd thing to be doing. Right? So I'll give you a moment to just kind of think about this state of affairs. Right? If you're doing, you know, training a really large neural network for one pass on SGD on vast amounts of data, why would you use weight decay when you're doing that, right? So maybe some of you know the answer, but I think that's a kind of interesting thing to think about. It's very intuition sort of violating uh at least uh for me.

So okay so the reason is because um you know it's not to control overfitting in the sense that if you look at weight decay different amounts of weight decay don't really seem to change the ratio of of training loss to to validation loss right so you can train with different amounts of weight decay if you train for long enough where you know you control your hyperparameters appropriately you'll end up with the same train to val loss gap so overfitting nothing's happening here even with zero weight decay but what is interesting is that the weight decay seems to be interacting you know somewhat in a strange way with the learning rate schedules of the optimizers. Um and so what's happening is that if you look at um sort of a a constant uh learning rate so this is a a model trained on constant learning rate and then you know you suddenly decrease the learning rate in 10 year zero. So you see this drop off as you you know decrease the learning rate. Um and then let's look at um different kinds of weight decay that you could do. And what happens is, you know, with weight decay, the model's not training very well at this high learning rate. And then when you decrease the learning rate, it'll very rapidly drop off. And when you look at sort of cosine learning rate decay, what happens is that you know the models with high weight decay start out very slow, but then as they cool down, that is their their learning rate decreases, they very rapidly optimize. And so there's some very complex sort of interaction happening here between the optimizer and the weight decay and some sort of implicit sort of acceleration that happens near the tail end of training that ends up giving you better models. And so the answer to the question I posed you is, you know, you don't weight decay because you want to regularize the model, which is kind of what it was designed for. You're weight decaying in order to get actually better training losses. And you end up doing that because of the various learning dynamics at the tail end of training as you decrease your learning rates to zero. It's a very sort of very interesting and complex and in some ways you know uh troubling you know thing to be doing uh with language models. But now you sort of see why you know if you look at the a lot of the reports you'll see we use weight decay. This is kind of why that ends up happening.

Cool. Okay. So, um, putting all that together, so there are certain things that I think are just kind of no-brainers. So, if you're picking various hyperparameters for your model, you don't really need to think too deeply about them, um, in the sense that they've been validated and basically everyone else does them. So, this is things like, you know, the hidden size of a of a MLP, the head dimensions of your your multi head attention, um, your

aspect ratio, um, and your choice of regularization through weight decay.

like all of those there's fairly good I think uh consensus evidence of how to pick most of these hyperparameters and those defaults roughly give you the kinds of um things that we suggest in the assignment so you can kind of follow along and they'll roughly give you something similar um to this. So okay any questions about uh the hyperparameter uh piece? Yes. Yeah. Is there a reason why dropouts like gone out of pattern? That's a good question. Um I don't think I've seen uh the question was why did dropout go out of fashion? Um I haven't quite seen a deep analysis of of why dropout is or isn't helpful. Like I haven't seen any uh result that for example shows that it helps for training loss. And as sort of this you know what this paper argues and logic would dictate there's not really a training overfitting issue with these models that can't even do one epoch over their training data.

Yes. Um, do multilingual vocabularies actually contribute to improved performance in one language? So, I get Yeah. So, the question was, do multilingual vocabularies contribute to improving uh performance in one language? When you say one language, do you mean do mon multilingual or like you know larger vocabularies help performance in English? Is that the right question? Yeah. So, I think in your high resource language, the impact is less, right? So, you know, if you're only thinking about, you know, English language language modeling, you can get away with smaller vocabularies. This much is kind of, you know, true. Um, but the place where larger vocabularies is really helpful is when you're starting to get at, I wouldn't say the tail of your distribution, but when you get to languages that are sort of more minority. And one great example of this, um, if you look at any of the coher, uh, announcements about their models or their tokenizers, um, they basically always argue that because of the way they have larger vocabularies and the way they train their tokenizer, um, non-English and like low resources languages, they they are packed into much fewer tokens and so people using those pay much fewer, you know, uh, much lower cost at inference time, right?

Which is a which is a great benefit. Oh yes, question. these plots um if weight t doesn't have a significant impact on the valve loss like why why do we care about like the training dynamics or the favorable operation dynamics right okay so the question was if it doesn't have an impact on val loss why do we care about training dynamics um the goal is still I want to get you know um good training loss right this is the game that we're playing and the surprising thing about weight decay is that somehow it gets us better training losses like I think the intuitive thing that makes sense is you do weight decay it gives you better val losses But that's not what happens. What what it's getting you is better training losses which are also the same as vales.

Yes. Are there differences in the architecture hyperparameter choices people make as they move towards like multimodal architectures if they're images text? Yes. So the question was about multimodal models. That is a great question. My my survey of multimodal models is very incomplete. Um what I can say is a lot of the academic and open work that I've seen um they do uh what you might call like shallow or like later fusion um or early fusion of the modalities and the way that works is you kind of bolt the vision modality onto a existing language model. In those cases, the hyperparameter and architecture choices are fixed. Right? Um, one thing I will note and I will talk about this in just a few slides, um, is that the multimodal models pioneered some pretty interesting techniques in stabilizing language model training and that's been a really big theme and I'll talk a

little bit about those. So what is different is often when you like bolt on this new kind of vision piece and you like retrain with that that's a big shock to the model and so you have to think carefully about how to stabilize that training process and those innovations have actually seeped back into like pure text language model training. Okay, cool.

So um I went back through and you know I looked through all these new papers and as I was trying to think about okay what's been new in the last year and sort of what new architecture and related things have happened actually you know the the core architecture hasn't changed much but I think the one thing that stood out as being very emphasized in a lot of the releases um has been what I would call stability tricks and so these are things where you would like to train your model in much more stable ways um and as you make bigger and bigger models or you train for longer and longer um these kinds of issues start to appear more and more. So I've taken this um from the mode 2 paper um and actually that paper is a great sort of set of you know academic results on LLM training stability and you know one thing they start with is is kind of this figure and you look at this blue curve over here and you look at this you know L2 norm of the gradient graph and this is terrifying graph to look look at right like you know your your loss curve kind of seems to be behaving okay but you've got some some bad spikes every now and and you open up your gradient norm and it's this horrible plot where you've got spikes everywhere where your norms are completely blowing up. Um, and you know, if you're training models like this, you're going to have a really tough time getting it to converge reasonably. At some point, it's going to, you know, hit, you know, gradient norm explodes and like you can't do anything and your training is done, right? So, you can't train any further.

And so, there's been a lot of emphasis basically trying to turn this blue curve into something that looks a lot like the orange curve. Um, and of course this loss is higher, but ignore that fact because I think they just switch data sets in between these two training runs. Um, but this orange curve, you know, has nice low gradient norms throughout. And that's really the kind of pod that you would much rather see. And so you might ask, where do stability issues arise in transformers?

And of course, they can arise basically everywhere. Um but if you look at the kind of interventions that people are making um there's really one place that really stands out as the the kind of problem child and that's the soft maxes. Um and it can be a problem because you're going to be taking exponentials and those can be numerically you know badly behaved. Um you're also dividing two numbers and so you might have a division by zero, right? So for many different reasons, this softmax piece is a is a part that you know um you might have lots of issues with. And so actually one more thing I want to talk about. So where are the softmaxes in a transformer? Well, there's one at the very end. So you've got to be careful about that output softmax. And also there's soft maxes in your self attention, right? So there's two soft maxes that that we're going to think a little bit about. And for each one I'm going to mention a stability intervention that has you know generally seemed to be effective. Okay. So the first one is called the zloss. Um and in my desire to site a paper that's older I've gone back to Devlin in 2014. Um where in a machine translation paper um you know their goal was to try to you know make sure that this normalizer was near one. So if you look at P of X that's the output softmax over here. Um the output softmax is two terms. You exponentiate your logets and then you divide by the normalizer Z right the Z is just summing up you know the the values across all the vocap. And so if you want this Z of X you want to train the network to have a Z of X close to one. Well then you can you know rewrite your loss and you can add a little second term

here to try to force log of Z of XI to be close to zero.

Right? Okay, so you're going to end up with an auxiliary loss term that's alpha log 2 Z of XI, right? You can kind of see that derivation on the on the right here. Um, and this is, you know, in some sense what people often call the Z loss. Um, I think, you know, Jacob Develin and others did this for machine translation for totally different reasons than what it's used for today. Um but this was I think the first instance of this in in language modeling land was Palm who used this as they called it auxiliary loss of Z loss 10^4 log 2 Z to basically encourage the softmax normalizer to behave nicely and you can kind of reason through the behavior of this regularizer. If it succeeds and it forces log of Z of X to always be zero, then the log and the exponent the exponential cancels and you've basically just got U of R of X. And that's a good place to be, right? That's a nice numerically stable operation. So all of these sort of problematic operations kind of go away. And so you can think of the softmax as being well behaved when Z of X um is close to to uh one or log of Z is close to zero, right?

And you know, palm in some sense is a is very much a pioneer because they did this zlos trick. Um, and many others didn't really do it for a long time or at least the ones that had open papers. Um, but then there was a kind of sequence of papers um that have done this. Byron 2 is actually the the earliest follow-up that I know of. And then DCLM and Almo and now um several others have basically picked up on ZLOS as a very nice convenient intervention uh for for improving stability. Um and then uh the other trick that we see so that was um how to stabilize the uh the output softmax but we've got another softmax we've got to deal with right the other softmax we have to deal with is in the attention operation um and so you know this is um from a Nvidia paper I forgot to put the citation marker um but here you know this is a a block diagram of how attention works you know you've got your your layer norm at the beginning um you got your 2KVS um ignore this for the moment. Um you might multiply your your Q's and your K's.

You'll softmax it. You multiply the V and then you'll project it. Um and then that's going to give you your fully connected and your output, right? So so if you ignore this this little piece over here, um you know, this looks just like your normal multi head attention operation. So what's kind of the difference here? Um so several folks uh came up with this idea or this approach called the QK norm where you take the the queries and the keys and you pass them through a layer norm layer um before you you know take their inner product for the softmax operation. Um and this is a you know very different kind of approach to controlling the behavior um of the softmax. Here you're not controlling the the normalizer Z.

Instead, you're controlling the inputs to the softmax to be kind of bounded in size, and that's going to naturally control uh the the bad behaviors of the softmax. Um, and as I said before, this is originally an innovation uh from the vision and sort of multimodal model community. Um, Deani in 2023, this was, you know, a paper on training very large vision transformers. Um and then Chameleon and uh Edith Feix from uh Hugging Face sort of used these tricks for their like multimodal training components. Um and then you know it got picked up by several others like GMAT 2, DCLM, OMO2 um all basically uses this kind of techniques um in order to stabilize um their training. And I think I'm allowed to add one joke per lecture and so this is the one I'm going to go with here. Um I think one of the things that really has stood out in terms of stability interventions has been just how strikingly effective layer norms are. Right? So we've seen, you know, going from layer norms just in the

the pre part of the block to the both the beginning and the end of the non-residual component and now we've also thrown it into the Q and the K uh component. At least in terms of improving stability, um layer norms have been shockingly effective without affecting performance too much.

Um the last trick that I'll note um I think this one has been sort of not quite as frequently used um which is to soft cap um the uh the lojets that go into the soft. So the other approach that you can take so qk norm is in some sense a very heavy-handed intervention because we're going to operate over the entire vector. Um but one thing you could do is after you take the inner products for self attention, you could pass them through kind of like a soft maximum operation. So you can pass them through this uh equation over here. So you have your low jets as your input divided by the soft cap multiply by the soft cap. What does that do? Well, if your lows start exceeding the soft cap by a lot, the tanh is going to clip them off to one. And so you're going to have a maximum value of soft cap over here, right? So this is going to control in some sense soft clipping of the logets um and gamma 2 um and I think 2 also do this um it hasn't been I think quite as uh popular otherwise and I think the other sort of evidence against this um the the Nvidia folks that I mentioned earlier did actually quite a few different sort of stability uh improving interventions and what they find is you know you have your baseline model over here this is the the perplexity of the baseline model 11.19 Soft capping makes it worse. QK norm actually makes it better because you can use more aggressive learning rates and sort of push the optimizer further.

Um, cool. Okay. So, so that's the um end of sort of the stability improving intervention stuff. Um, does anyone have any questions? Um, I think that's been kind of the the new development over the last year. Yes. So, for the QKV norm um like understand that during training you will have the layer norm being applied at inference time. Is the layer norm still being kept? Yes. So the question was at inference time do you still use the norm? And the answer is yes because the layer norm has kind of learned parameters like the whole you know action of the layer norm is it takes an activation normalizes it to unit and then scales them to some size. If you take that out that's a huge change to the model. It will have no idea what to do with those unnormalized activations.

Okay cool. All right. So um I have this last bit last few slides um that I want to end with. Um if we if we go over then we can always push this into the thee lecture but I think we also have a lot of content next time because I have to cover um deepseat v3. Um so the last thing I want to cover is variations on the attention heads. Um so attention heads I think haven't had as much you know work done to them. Um but there have been a few I think important changes that you need to know about in order to understand uh the models that are being trained. So the one thing I'll talk about the first thing I'll talk about is GQA and MQA. Um and these aren't really critical to kind of the training time behavior of the models but they're very important in understanding the inference cost and inference behavior of the models. And because this is a important architecture change I'll mention them here um in addition to probably being mentioned by Percy um in some of the inference lectures. The other thing that's a kind of new development I'll mention is how the most recent models like Llama 4, if you've heard of it, supports supposedly 10 million tokens of context. How does it do that? Well, it does so by sort of messing with the attention pattern in very structured ways. Um, and so I'll talk about um that as well.

So, GQA, MQA. Um, if you've looked at like some of the larger models like the big llama models or or others,

you'll have heard or seen this term GQA or MQA. Um, and I'll talk through what that sort of means. So, to set the stage, let's think about the compute that you need to do attention, right? So, this is once again 224n slides here. um you're going to take your you know XQ your your query and your X K and then you're going to form your big uh sort of quadratic attention matrix and you can sort of walk through each of these matrix multiplies and you can convince yourself that the total number of arithmetic operations is going to be B * N * D ^ 2.

So that's going to be um B is the batch dimension um N is the sequence length and D ^ 2 is going to be the hidden uh dimension squared and you can ask about the total memory accesses and this is going to be B * N * D and this is going to be for example accessing just this matrix here this XQ is going to be that size and then the softmax is going to be B * H * N^ 2 and you can kind of convince yourself of that by just thinking about the size of the softmax matrix.

which is going to be batch time number of heads times all the different softmax activations that you have. So that's n^ squ of them, right? Um and you've got a projection and you've got d squ projection operations at the very end over here. And so we can take the ratio of uh total memory accesses and arithmetic operations. Um and this is going to be something that will be very important in a couple lectures. Um this idea called um arithmetic intensity, right? So we want our arithmetic intensity to be high. What that means is we want to be doing a lot of compute for every single memory access that we do.

And this is going to be because memory accesses are very expensive on a GPU relatively speaking. And compute is relatively cheap. Um and so in this you know batch computation that I'm showing you here you know the arithmetic intensity if you take the ratio of those two things is going to be 1 over um k + 1 over um bn inverse. Um and so this is going to mean that we can kind of keep our GPUs um running um because if we have sort of large number of heads and we have large um batch size and large sequence length, you know, those are all going to be sort of good large numbers. Of course, this is what happens at training time, right?

So the issue is that inference time we do not have these big chunky matrices to multiply together. And so that's going to really change the the nature of the behavior of of our algorithms. So when we're generating text, right, remember that we have to generate a token and then the the transformer has to read that token and then it has to process it. And now we can get the next token distribution and then we do the things auto reggressively one token at a time, right? And by doing this, we can't parallelize this generation process. We need to go step by step for every single new token. Um, and when we do this, we're going to need to incrementally compute attention. an idea um that people call the KV cache. Um and so what do you do? This is a lovely animation um of a KV cache um that's been explained.

So if you can um sort of look at this this uh figure, what you're doing is you know you've got a query token, right? A query token here is you've generated a new token. You're conditioning on it and now you want to ask what sort of information should I look up in the past that query token, right? and your query tokens are shifting from one through n because you're generating new tokens one at a time. You're building up this sort of key cache over here where basically I'm building up all of the past tokens keys, right? And the past tokens keys don't change because they only depend on things in the past. And so I'm incrementally as I generate tokens building

up all of these past keys. And each time I can compute one new element of Q.K. Right? So the big attention matrix is going to be this lower triangular matrix. I'm computing one row at a time and that row is exactly what's necessary to generate the next token.

Right? So so this KV cache idea if you've not seen this before is this idea of saying I'm going to generate the K's and the V's um incrementally as I go as I generate each token and I'm only going to compute Q um that's absolutely necessary to do my operations. And so once again you can go through and do um sort of the the various arithmetic uh components of you know how many uh flops do we do what's the total number of memory accesses and if you think about the KV cache right I'm only multiplying the absolute necessary keys and values right since I'm saving all of the intermediate computations um I'm not wasting any sort of matrix or vector vector multiply the total number of arithmetic operations remains exactly the same B and D But the memory access patterns are now different. Why is that? Because you know when I do this KV caching thing, I'm going to have to move various kinds of uh parameters in and out of memory repeatedly. Whenever I multiply with a key sort of K matrix, I'm going to have to put that into memory, right? And then multiply by K. And then I need to, you know, put that away and I need to compute some activations. And so I'm repeatedly loading in um different matrices. And that's going to give me a much higher total memory access of b ^2 d plus n d^2. And so when you take this ratio now the arithmetic intensity is not so good. You're going to get n /d + 1 over b inverse. Um and so if we sort of reason through this. Okay. So if I want arithmetic intensity to be high, I want this thing inside to be very small.

So I need really large batches and I need n / d to be small. What does that mean? I need really short sequence lengths or really big model dimensions and this n /d is really unfavorable because I don't want a bigger model and I don't want a shorter sequence length right and so this is the core in some sense inference cost trade-off that people face right you have this very bad memory access pattern where you have this one term n /d that's kind of really killing you in terms of you know the throughput of your system and so this motivates this thing called mqa Okay. Um, and the key idea here, right, hopefully, you know, you kind of see from this figure back here that really the part that's really bad is the keys and the values. They have this KV cache thing being built up and there's memory moving in and out. So, what you do is you can have multiple heads for the query, multiple query heads, but only one dimension or one head for the keys and values. This immensely simplifies things. Once you do this, now you're moving much less information for the K's and the V's. And so, you know, KMV is shared. Um, but query has many heads. And so, you still have multi head attention um or multiple queries um but only single K's and V.

So, that's why it's called multi-query attention. And now when you do the same kind of arithmetic, we have fewer memory accesses because we've shared the K's and the V's. And the arithmetic intensity is much much better behaved, right? And so we can increase things like you know we have uh we've decreased the first term by a factor of n so longer sequence lengths are now viable and the second term is now divided by the number of heads. So this term is also not so terrible right so all the different terms are controlled now and MQA can give you much better um behaviors. um GQA or group query attention basically changes this slightly. Um instead of having you know single uh query or sorry um multiple query and single key you can reduce the number of keys by some multiple and so this will let you trade off between kind of the the inference time behaviors and the expressiveness of the model because maybe going from multi head all the way to multi-query is a little bit um

too aggressive. Um, you know, some works show that uh GQA uh doesn't hurt, but multi head attention hurts. I'm not going to get into that. I'm just going to close off with this this very last thing, which I think is a really interesting development in the last few months. Um, so back in 2019, OpenAI had this kind of cool paper um basically arguing how to build longer attention uh models. And they were basically arguing well one way to do that is to come up with sort of sparse attention patterns right so instead of paying attention to all of the sequence I'm going to pay attention to let's say a local window at each sort of chunk and then I can have sort of um other sort of attention patterns that are like diagonals that help propagate information across. So you can build sparse or structured attention that trades off you know various kinds of expressiveness versus runtime. GPT3 uses exactly these kinds of tricks when they originally released it um to get larger um attention windows. Sliding window attention is another variant of this idea where you know at each layer you only pay attention to a small region around your current position. Um and this also is going to control the total amount of sort of resources that you need. Um the total amount of resources you need in order to do longer contact. So your effective receptive field is now the local one times kind of the the layers. The final trick. So those were kind of the older ideas. But the way that this has kind of been modern instantiation is some of the recent papers like llama 4 um and gamma and cohhere command a have now come up with this very clever trick of basically having um transformer blocks where in this case you have a block a set of four transformer blocks. The very bottom one uses full self attention with no position embedding. So there's no rope no nothing. It doesn't know about position at all, but it's full self attention and it only happens once every four blocks. And then the three blocks above it use sliding window attention with rope, right? And so this is actually a really clever trick to both control the systems aspect of things because the full tension only happens every, you know, every now and then and also the length extrapolation aspect because rope only deals with local context windows and anything that's really really long range has no position embeddings at all. So it could, you know, extrapolate very very aggressively, right? Because you don't have to do this position um extrapolation that you do uh with something like rope. So that's a really cool development that we've seen in the last um couple months. So all right, I think we're coming up on time. Uh feel free to to ask any questions about architecture or hyperparameters. Um I'll be happy to answer questions after.