(no title)

79 MIN · YOUTUBE · HTTPS://WWW.YOUTUBE.COM/WATCH?V=JUOVZKPBIKK&LIST=PLOROMVODV4RMQXOCAZWATUHHQ-YEMBLCV&INDEX
https://www.youtube.com/watch?v=JuoVZkPBiKk&list=PLoROMvodv4rMqXOcazWaTUHhq-yembLCV&index=1

SUMMARY

The CS 33336 course on building language models from scratch focuses on providing a deep understanding of the mechanics, mindset, and intuitions behind language model development. The instructors emphasize the importance of hands-on experience, encouraging students to build models while adapting to the rapidly evolving landscape of AI.

- *The course is structured around five main assignments, each designed to progressively teach students about language model components, including tokenization, architecture, training, and scaling.*
- *Key topics include the mechanics of transformers, the significance of efficient resource usage, and the challenges of scaling models effectively.*
- *The instructors advocate for a "from scratch" philosophy to ensure a comprehensive understanding of language models, contrasting this with the current trend of simply prompting pre-trained models.*
- *The course will explore the historical evolution of language models, from early n-gram models to modern architectures like transformers and mixture of experts.*
- *Students will engage in practical assignments that involve implementing tokenizers, optimizing training processes, and understanding scaling laws.*
- *The importance of data quality and alignment in training language models will be emphasized, along with the ethical considerations surrounding data usage.*
- *The course aims to equip students with the skills to critically evaluate and innovate in the field of language modeling, fostering a mindset focused on efficiency and effectiveness.*

Welcome everyone to CS 33336 language models from scratch. Um, this is a teaching staff. I'm Percy, this is Tatsu, Marcel, Herman, and Steven, and we're bringing you the third edition of 336. So, we'll just do a quick round of introductions. Um, so I'd like to say that I've been doing language models for 20 years, but most of that time was uh of small language models. Um, actually this is still small in the grand scheme of things. Um, I think when Tatsu and I started teaching this class uh 2 years ago, we weren't really sure of what we were going to expect, but we were very pleasantly surprised that so many people wanted to learn how to build language models from scratch, especially in these days when a coding agent could probably zero-shot a language model. Um, I'm really glad to see all of you here to actually want to learn how they work.

Yeah, thanks. Uh, I'm Tatsu. I'm one of the co-instructors. I'll be, you know, talking to you once we get to architectures and scaling and all these other very fun things. Um, I'm really excited. This is the the most fun class I've taught in my time here. Um, I think every year, you know, Percy makes fun of me because he says you have to redo all your lectures because you took on architectures and everything changes every time. Um, but it's actually pretty fun for me to do it and it's kind of the first time that I've had that experience. So, I'm looking forward to uh going through this experience again with you all.

Hello, I'm Marcel. Uh, I've seen this class once before and I'm I'm returning cuz it was so much fun last time. Uh, uh it's it's a lot of work though. Um, in my research I do architecture stuff. I do higher-order gradients and I do training. Um, yeah, looking forward to work with you guys. Hello everyone, I'm Herman. Uh, a year ago I didn't really know how LLMs worked. So, I think for me tokens were the things you collect in video games and attention was the thing you had like the attention economy.

Um, then after spending a lot of time on this course last year, I'm now doing uh LLM research and I'm really excited to TA this year. Hi everyone, I'm Steven. I am a first-time CA for this course and I'm very excited about it. I think it'll be a lot of fun. Broadly in my research I work on language models, theory, and some data efficiency stuff and I'm excited to meet you all. All right, so let's let's go into things.

Um, so this is the third time we're offering it. Um, last year we decided to put all our lectures on YouTube, so some of you will have seen it. Um, so what's new? Um, well, what hasn't changed is the from scratch philosophy. We still believe strongly that by building everything from the ground up, you really learn um how everything works. Um, of course, we don't actually build everything up from from scratch because that would be, you know, that wouldn't fit in a quarter.

Um, so over the the last 2 years we've been refining our our recipe and figuring out what are the things that you build up from scratch that are most highly value. And then finally, as Tatsu alluded to, um even in a year a lot has changed. Um, I think um this year we're we're going to spend maybe a bit more time on mixture of experts and of course agents are very uh um popular these days, so getting uh a handle on long context and what is needed for that are going to be important.

So, why do we make this this course? And I think the problem through uh 2 years ago was that researchers were becoming disconnected from the underlying technology. Used to be the case um maybe 10 years ago, all AI researchers would just implement and train their own models. And then even 8 years ago, um people would download uh pre-trained models such as uh BERT and fine-tune them. And I think a lot of today um you can get by by simply prompting a model.

And of course, there's nothing wrong with prompting a model. You know, I think you can do really amazing things with it. Um, I think moving up the abstraction in general is is a great thing, but abstractions are leaky and sometimes if if I'm sure all of you have prompted models, you run into situations where something is you wanted to do something, but you it just can't do it and there's no recourse. Um, and I would argue that if you're really interested in fundamental uh research, uh by simply prompting a model, you're sort of vastly constraining the set of options, the design space you're looking at. And by uh looking at fundamental research, you really need to tear up the whole stack.

So, I would argue that full understanding of how language models work is really necessary for fundamental research. And the way that we're going to get our understanding is by building. That's the philosophy of the class. But there's one small problem, which is that industrialization of language models has happened. Frontier models are really really expensive. Even this is 3 years ago, uh GPT-4 was supposedly costing 100 um million

dollars to train and now costs probably are on the order of you know, a billion, although that's uh speculative. And the number of GPUs that um all the big labs are building is just kind of immense. Furthermore, there's no details on how many of these models are built. So, even back in uh 2023, the GPT-4 paper explicitly says that due to competitive landscape and safety implications, we're not going to share anything about how the models are being built.

So, these frontier models are in some sense out of the reach for us. Now, we could build small language models and we will build small language models, but this I think it's important to remember that these might not representative of um the actual frontier models. And I'll give you two examples why uh this is might be the case. So, here's one. Of course, this is from actually quite a long time ago. Um, think back in 2021.

If uh we're going to spend more time on flop counting and looking at where the computer is being spent, but if you look at small scales, the a fraction of flops spent in uh the MLP layers is around 44% and if you scale up to 175B, then it goes to 80, you know, percent. So, what you optimize and what matters at large scale is going to be different from small scale. So, if you did a bunch of small scale stuff here um on the attention, you might not experience the same benefits at large scale.

Um, the second I example is that we know of emergence um of behavior with scale. So, small models um this is again for a while ago, but even back then, um if you uh have um zero-shot or few-shot learning of various tasks, it basically seemed like nothing was working. And only when you reach a critical scale, do you suddenly see a lot of improvement. So, again, if you're working at small scale, you might not see certain types of phenomenon compared to if you were working at full scale.

Okay, so that might be a little bit uh disheartening, uh but fear not, we're going to learn something in this class. And the question is what can we learn that actually transfers? Okay, and I think it's important to break this down into three types of knowledge. First, there's the mechanics of how things work, what a transformer is, how model parallelism works, right? Um, there's mindset, which is how do you go about approaching building a language model.

Um, we're going to talk about how you want to squeeze the most out of your hardware, taking scaling seriously. And in finally, intuitions, which data modeling decisions are going to yield good uh performance. So, now in this class, I think we can do a pretty good job of teaching the mechanics, uh which is how things work, and the mindset. And we're going to really emphasize that you profile and benchmark everything and try to optimize for efficiency.

These things do transfer to a larger scale. Now, the intuitions about what modeling decisions and what um data decisions do work uh might not necessarily transfer across scales. Um, for that you actually have to go somewhere where you can have do things at scale. So, on the note of intuitions, um it is worth remarking re- uh remarking that some design decisions are just not justifiable. Um, and just purely come from experimentation. Whereas mechanics, you can kind of by construction see how uh the parallelism and how kernels are going to speed things up and so on. Um, but for intuitions about what modeling changes work, um I think you just have

to run experiments. There's this famous Noam Shazeer paper that introduces the SwiGLU activation, which we're going to look about. And in the conclusion section, um he very honestly has this final uh sentence which says, "We offer no explanation. We attribute their success of these architectures all else to divine benevolence." So, um that in some sense is something that you just have to gain from experience.

Final note about the the bitter lesson, which I think has been um been, you know, circulating and people talk about it. I think there's a sort of a common mis- conception here what it means. I think the wrong interpretation is that scale is all that matters, algorithms don't matter. But that's not correct. The right interpretation is that algorithms that scale are what that or what matters. Okay? And you can think about very simply that the accuracy of your model is basically your efficiency times uh the resources. So, efficiency is output over input and resources is the input. Um, and you know, efficiency is is actually in some sense way more important at larger scale, right? If you're doing a small scale experiment, if your run takes twice as long, maybe you just wait twice as long and and then you come back later. But if you're doing things at scale, you know, that could be, you know, hundreds of millions of dollars and you know, definitely don't want to do that. Even like a 5% improvement might be a big deal.

So in fact, efficiency is actually really really critical and I hope to bake that into your your kind of mindset as one of the consequences of this class. And empirically, if you look at there's this paper from OpenAI in 2020 that showed there's a 44x algorithmic efficiency on ImageNet between 2012 and 2019. And it's surely the case that hardware did get a lot better. Um but that also comes with algorithmic improvements and of course, when you multiply them together, that's when you see like a huge bump in um in efficiency and accuracy.

So the framing um with all of that said is what is the best model one can build with a certain data and compute budget? Um for pre-training, it's mostly we're going to talk about compute budget because we're going to try assume that we have a lot of more data than we have compute, but if you're in a setting where you're data limited or you have you know, stash away actually tons of B200s, then you might be uh data bound.

So in other words, maximize efficiency and we're going to see this theme come up throughout the class. Okay. So so next I want to spend a bit of time talking about language models and a bit of a history and just contextualization before we jump into the more technical details. Um so language models have been around for a while. Um you know, Shannon back in the 50s was using language models to measure the entropy of English. Um and for a long time, N-gram models were used actually in machine translation and speech recognition systems. Um and they weren't the whole system, but they were important part of making sure that you generated fluid text.

Um I would say that the the lineage of the modern language models comes from neural architectures. And so there is a bunch of uh ideas, I think, which are important to this development. So in the 90s, there was LSTMs. Joshua Bengio actually you know, wrote the first neural language model in paper back in 2003. This was actually not LSTM. This was just a feedforward network that looked at the small context. Um and then there was a seek-to-seek modeling which boldly said we can actually compress a whole sentence into a a vector. Um the Adam optimizer, attention mechanism which was developed for machine translation, the transformer

architecture which built on top of that which was also developed for machine translation. And then scaling up to mixture of experts, uh model parallelism, you see a lot of different architecture and also systems and optimizer ideas in developing the 2010s.

And then by the late 2010s, I think things were starting to get really interesting. So there was the ELMo and BERT. These are language models that were trained on lots of text and then they you could fine-tune them on some downstream tasks like question answering and it would show a huge improvement on that. So the the model there was, you know, take one of these models and then fine-tune. Um and then Google had a paper that was really I think foreshadowing this view of, you know, prompt in response out.

So it was really I think OpenAI that really opened up the floodgates here by embracing scaling. So they had a GPT paper back in I think 2018 or so that they scaled up to GPT-2 and then and then they really figured out you know, or embraced the idea of scaling laws which we're going to talk about in a in a second which enabled them to train GPT-3 which was a much more massive, almost more than 10x larger large model at the time. And it could show emergent behavior like in-context learning.

At that point, you know, Google was saying, "Okay, we need to do something too." So they trained a massive model. It turned out to be kind of under-trained and turned out that their DeepMind which was not integrated with Google at the time had you know, figured out you know, optimal compute optimal scaling laws. Um so all this was was happening and after GPT-3 came out, I think for a many folks, this was sort of kind of wake-up call.

And at that time, there were a lot of early attempts to let's try to replicate this. So there was a grassroots organization called Luther that created some open data sets and models. They weren't very large because they didn't have much compute. Meta's uh first LLM was uh you know, it was you could see tell that it's a replication because it was like 175 billion parameters. Um but it was not a very good model. Um they ran into a lot of hardware issues.

Um and then there was another Hugging Face Big Science project. So these models were I would say not a very strong. Then in the last 3 years, I think the open model ecosystem has changed quite a bit with Meta kind of leading the way with the Llama series of models, Llama, Llama 2, Llama 3. Mistral got in the game. And then a whole set of Chinese um models uh um I I think I'm missing some like ByteDance has something and um I think Tencent probably has some other stuff. So it's hard to keep track of everything, but you know, everyone's heard of Deep Seek and Qwen.

Um so I think I got the the um main ones. But I think what's interesting and exciting about these is now now we have open weight models that are approaching closed models. So depending on who you are asking and how you benchmark, they might be a little bit behind or comparable, but they are definitely very very credible models that are being widely used in industry. Now there's an another line of work which is going beyond just releasing open weight models.

AI2, Nvidia, and the Marine project which I work on, we try to provide not just the weights, but the paper and the code and the data so that we can understand how these models are built in a more thorough way. Why do I emphasize this open ecosystem so much? Mostly because this course would not be possible I think without these models. By the fact that there are still many papers that are being published about how these big MOEs and RL systems are working enables us to at least glimpse into how these frontier models are being built and trying to triangulate the pieces.

Now I think of course a lot of the details even with these like you know, Qwen papers, I think they're missing a lot of details. You can't reproduce them. Notably the data mixture is is something that we don't know, but I think it's better much better than nothing. Okay. So in the last decade, I think the idea of what a language model has changed. Right? It used to be something that you you fine-tune. Um and then it was something you prompt.

And now in the ChatGPT era, it was something you talk to and that you can have a you know, conversation with. And now okay, I guess I don't have internet. Um that's fine. Um and now we're in the era of agents. If you click on this link, it basically shows you a giant agent trace. Um and I'm I'm still kind of it's mind-boggling how um strong some of these models are. You give it like a page of text and it does have some really complicated agentic coding task.

Um so you know, what we demand of our language models today is probably beyond the imagination of anyone back you know, 10 years ago. That said, I think the fundamentals I think haven't changed that much. Largely, we still build on GPUs and kernels. We still optimize using gradient or stochastic gradient like approaches. We still have the transformer and attention and you know, we'll we'll talk a little bit more about architectures, but it hasn't changed that much. I think the specs are different. Now we demand greater context lengths which means that inference efficiency matters even more.

So the good news for us is that we didn't have to completely change this class, only touch two sections on the latest you know, Chinese architectures, but the the fundamentals I think are here to stay at least for now. Okay. Maybe I'll pause there in case there's any questions or or thoughts.

Okay. So let's let's go on. So brief interlude. So what is this program? Um so this is what I call an executable lecture. This is a if if you think it looks like a Python program, it is because it is actually a Python program, um but it has been rendered for your viewing pleasure. Um so when I step through it, it is actually executing the the lecture and it makes us it possible to basically step through code and which will be hopefully interesting and important later. And you can also see the hierarchical structure of this lecture. For example, we're done with this function now we go back to main.

Okay, so let's talk about the course logistics and the syllabus which I think it's going to take the good chunk of time. Okay, so all the information is online on this website or cs336.stanford.edu. This is a five unit class. I think this probably class has a certain reputation so I don't need to belabor the point too much but you know, this is we have five assignments. They are pretty intense even the first assignment according to this one um review was actually you know, equivalent to the five assignments for cs224n. I've been told also this is like wait this is

exaggerated um but better to I guess be conservative in your estimates. So why should you so why should you take this course? Okay, so first you have an obsessive need like me to understand how things work. That should be your primary objective. I think just pure curiosity on how language models work and then in doing the class you'll develop a much stronger research you know, engineering muscles and have the confidence to go into a new setting and be equipped to deal with whatever comes up.

So I think when I started at Stanford I screwed this class called you know, statistical learning theory which was basically teaching the theoretical side of machine learning and that was nice because it equipped people so when you read a paper you can understand all the math and now since the field has kind of shifted a lot more towards this more systems and empirical side of things. This is sort of the kind of an analogous class which gives people enough depth that they feel that everything else seems kind of easy.

So why should you not take this course? This is important because there's reasons you shouldn't take this course. First you actually want to get some research done this quarter. You should probably talk to your advisor. They should know you're taking this course otherwise they'll be probably there might be some surprises. You're interested in learning the hottest new techniques in AI. I think there are many other great courses you know, seminar courses topics courses that are good for that. We don't do all many of the things we don't do multi-modality. We don't talk about you know, agents in any depth. So if you want to learn about this that stuff this is not the right course for that.

If you come in and say like I have an application domain I want to get good results on it. Probably this is not the right course at least to start. I always recommend just prompt the model fine tune the model and then as a last resort you pre-train your own model because it is a pain and it's also expensive. But it's a lot of fun. Okay, so if you're not taking the class you can soft follow along at home. So all the lecture materials will be posted on the website oops and they are also recorded through cgoe so thank you cgoe for doing that and later they will be published to YouTube.

Um Now of course following at home watching the lectures is great but you learn really by doing the assignments so you'll have to figure out how to motivate yourself to do that. Okay, so speaking of assignments we have five assignments. The philosophy of the assignment is you know, how how do we do the from scratch but not just you know, say build a language model and that's the assignment. So we don't provide scaffolding code but we do provide a bunch of unit tests to make sure that whatever you're building is actually correct so that it's you don't get the sparse rewards setting where you submit a homework and it's either correct or not.

Um What I recommend is that you can the assignments are structured so that much of the assignment can actually be done locally on your laptop. You can implement it and check for correctness and then we are providing a cluster so that you can do an actual training run to see what the accuracy is or get a bunch of GPUs to actually benchmark the you know, performance of a some kernel. Um And then for fun we will have you know, some leaderboards for most of the assignments at least and they will look something like well, now that you've learned about this topic try to minimize the perplexities given some sort of budget. And and both Marcel and Herman were masters of leaderboarding back in the day which was last year or the year before. So if you

want tips on that I'm sure they would be happy.

Actually I don't know if they would tell you their secrets but you can try. Okay, so um You know, last year we were thinking about well, what can AI do is like okay, well, I mean yes everyone can use AI but um uh but but you know, just try to do your best. I think now I think coding agents have gotten so good that they can just also solve all the assignments, right? But to state the obvious obviously you're not going to learn anything if you just feed in the PDF assignment one into Claude code.

At the same time AI can be very useful for answering questions and tutoring. So we have to find a way to you know, uh leverage AI. So what we've decided to do is we provide you agents.md file or of equivalently a prompt which asks the AI to be pedagogically minded. You can read more about in our AI policy guide and the requirement is that if you're going to use AI use it with this prompt. And so it will answer questions about code. It will clarify any understanding but it won't accidentally generate like the transformer for you when the homework is to implement the transformer.

Okay, and this is the first year we're trying to do this so please try it out and give us feedback if it's working or not working. Okay, so compute. So this year we have thanks to modal. They have provided us with a compute credits on their platform. It's actually quite quite nice. You can get a number of unlike last year well, I guess you guys didn't take the class last year. Last year we had a cluster you SSH into. This is using more of an API to which I was initially skeptical of but you know, looking at it actually is pretty pleasant to use. So again try it out and give us feedback on how it is. We've written a guide on how to access and use the compute.

Okay. Any questions about course logistics? Okay. All right, let's talk about what we're going to cover in this this class. So um there's basically five parts mirroring the five assignments that you'll have. Basic systems scaling laws data and alignment. So I'm going to now go through each um uh part and just give you a taste of what you will will learn.

Okay, so in the basics this is basically the first um two weeks. The goal is to just be able to train a language model and build it from scratch. So the components here are we're going to tokenize the data. We're going to define architecture and then we're going to implement optimizer and trainer. Okay. So then you wonder like what are the other the rest of the class is for? Well, we'll get to there. So let's start with tokenization. The tokenization is really about what are the atoms that the model operates on.

And you know, formally a tokenizer converts between raw inputs which are just bytes and a sequence of integers which are represent the tokens. Conceptually it's a segmentation of the of the text. We're going to talk about the byte pair encoding BPE tokenizer which intuitively breaks the input into frequently occurring chunks. And then remember this class is about maximizing efficiency.

So through an efficiency lens tokenization is good because it takes a long sequence if you just think about the raw byte stream and reduces it into a smaller number of tokens. But more subtly but maybe more importantly it allows you to do adaptive computation. So some maybe some places are actually a lot of bytes but actually it

should be compressed into a one token where some of the more rare or interesting parts of input should be left as multiple tokens.

Okay, we'll talk more about this. I just want to mention that you know, every year I'm hoping that I don't have to talk teach tokenization because the dream is to really have an end-to-end way of that directly operates on bytes. And there's been a number of work including you know, recently there's been this this H net work that seems promising but so far these have not been scaled to the frontier and since the frontier models are still using tokenizers Uh, felt like it would be still wise to teach tokenizers.

Okay, so, um, now after you tokenize your input, you have a bunch of tokens. Now you define a model on top, and everyone I think has a familiarity with the rich uh the transformer. Um, and if you take in 224N, uh the NLP class, then you've seen transformers. Um, since then, I think there have been a lot of improvements or refinements to transformers, which um I I think are important, and Tatsus is going to talk more about this in a bit. But just to run through a set of types of things that one might have to think about. So, the activation functions have uh evolved. Um, um how do you positional encodings has evolved? Um, how you normalize um um has uh the different layers to prefer uh blow up has evolved. Um, you know, instead of doing full attention, there's many uh ways to basically reduce the attention computation because attention is n squared and it's and where n is a sequence length and that gets really expensive.

Um, so there's a bunch of ideas around that. Uh, if you're more ambitious, uh you can look at these uh state space models or you know equivalently uh linear attention. Um, like Mamba and gated DeltaNet. These have become been popular in the last uh you know few years, and usually some hybrid between these models and attention seems to work uh you know quite well. So, we'll be exploring some of that. Um, and then within the MLP layers of the transformer, um the original transformer was just a dense um MLP, and now mixture of experts um has become the sort of a dominant paradigm for building compute efficient um your transformer.

So, we're going to talk about um that. And of course with M MLP a mixture of experts, you know, it's not just defining your architecture, but we'll see that we'll also need uh different uh techniques for training um the model. And then finally, perhaps somewhat boringly, but an important uh question is, you know, what is the shape of your transformer? How many layers? Uh how many heads? Um what is hidden dimension? Number of experts? Um this might come in more as we talk about scaling laws, but setting these um is actually seems kind of you know, almost trivial. It's a hyperparameter, but actually in the context of uh scaling um language models has a huge huge uh um implication.

Okay. So, once you define your model architecture, um how do you train the model? Um, and here there's a bunch of design decisions around the loss function. There's next word next token prediction, which is the default, um but people have found that um looking at predicting more than one token seems to uh be helpful for improving the model. Um, and um there's optimizers. People used to use um Adam, but increasingly Muon has been used especially with some of the latest open models such as um the the Kimmy K2 models. Um, initialization, um which is again sounds kind of you know boring, but turns out to have a huge impact on how um your ability in the training stability of larger models, learning rates uh schedule, uh regularization, um you

know, batch size.

Um, and then MoE specific things. So, you know, you look at this list and you might think, well, these are just hyperparameters. I'm going to try a bunch of um different options out. But it turns out that, you know, really being very careful of about uh setting these hyperparameters in a principled way will make the difference between a run that just blows up and is useless between a run that is achieving state of the art. Okay. I'll come back to that point when we talk about scaling laws. Okay, so then in assignment one, what you're going to do is you're going to implement the BPE tokenizer, um implement the transformer, the loss function, optimizer, the whole training stack. We're going to make you do a bunch of resource accounting, so you understand where your flops are uh are going. You're going to train some models on uh these beta sets like tiny stories and open web uh text. And then there's going to be a leaderboard um that where you're going to try to drive down perplexity as uh as fast as you can. So, uh for those of you familiar with the NanoGPT speedruns, it's kind of uh similar to that.

>> [clears throat] >> Okay. So, by the end of assignment one, you should be able to walk away and build a language model uh you know, from scratch. So, that's very exciting. The you know, if you have a if I have a high-level uh takeaway here, it's that you know, while the tokenizer and modeling and training are sort of presented as distinct pieces, it is actually you know, everything is about kind of balancing the the following. So, you want expressive models because you want to represent the complexities of the data, but at the same time, you want to um your training to be stable. And we're going to see uh talk a lot about um how do you keep the parameter and um gradient norms in the sort of Goldilocks zone, so they don't blow up and they don't vanish. Turns out a lot of training language models is about just you know, stab a stability. Um, and then finally efficiency, which is um somewhat more straightforward. You just make it run fast on hardware. But you're going to see interesting things like, you know, if we change the architecture, a lot of the architecture decisions are, well, we can make it faster by let's say reducing the um the projecting our low dimensional space. But then the question is, does it work as well? And so, um making those tradeoffs is something that uh is going to is sort of the name of the the game here.

Okay, so, um as in assignment two, we're going to dive more deeply into um you know, systems. Um, and the goal here is just to get most out of your hardware. Uh, so we're going to talk about kernels, how you parallelize across multiple GPUs, and and how you do inference. So, um the basics, which we're actually going to start on our next lecture. Um, I mentioned resource accounting, and I think you know, you've all probably you know, built models um but this is really about kind of um keeping track of all where all the flops go. And where all the memory is being spent. So, we're going to spend some time basically doing the resource accounting. Um, we're going to see, you know, um this this formula that comes up, how do you how many flops does training a 7B on model on 1 trillion tokens? Well, it's 6 * n * d roughly.

Um, and where does that come from? Um, and then we're going to look at um you know, the hardware. And here's a very cartoon I'd you know, picture of what what to remark about hardware is that your memory is not where your compute is, and you have to move uh your either your parameters or activations into uh from the memory to compute, do the compute, and move it back. Right? And that often is the bottleneck. Um, so, you

know, for example, a B200s, which we'll have the opportunity to play with, um has uh 2.25 uh petaflops per second if uh BF16, and has uh 8 terabytes um a second of memory. So, you know, what does that mean? I think in when we I think I'll do this uh next lecture.

We're going to break this down and use this information to do some um calculations and see how long uh different types of algorithms will need to take. We're going to talk about roofline analysis, which allows us to understand whether um a computation is bottlenecked by either compute or memory. In general, it it is um um you know, memory. Uh, and then talk a little bit about benchmarking and profiling. Okay, so, um so here's what uh uh you know, DGX B200 looks like. You have eight GPUs. Um, they're connected via you know, NV link.

Um, and then if you have many if you have a a thousand GPUs, then you would have multiple of these, and they're you know, connected either InfiniBand or Ethernet. So, uh the next two parts when we talk about system is, you know, kernels. So, kernel is basically a function that runs on the GPU, and when you're just using plain PyTorch, you know, the all the PyTorch primitives actually correspond to launching particular kernels, which are built in.

Um, so you're already using kernels uh this you know, uh whether you know it or not. Um, but the point is that um if for certain types of computation, if you look at it, you can actually write custom kernels to make the GPUs go faster. And the main principle here is um organizing the compute to minimize data movement. So, remember this picture, um moving data from uh from memory is expensive, so you want to try to minimize that. So, just as a kind of a simple example, suppose you wanted to compute A and B.

So, often you would have to read from your um high bandwidth memory HBM, compute it, write it back, and then read again, compute it, uh write it back. Right? So, then you're basically sending the data back and forth twice, and uh there's this idea called fusion where you read it once, do both of the computations and you write it back. And that will save you a lot of time. So, that's operator fusion. Tiling is is kind of a more sophisticated variant around the same idea.

There's also GPUs have also gotten a lot more complicated and I'm not sure how many of these details we'll have time to get into, but at least we want to expose you to some of the sort of the peculiarities, I would say, of GPUs and the and give you appreciation for the types of things that one has to consider in order to squeeze as much juice out of them as possible. And we'll write some kernels in and and try.

So, what happens if you have thousands of GPUs? Um So, the principle of minimize data movement is still, you know, the same. The only thing is that moving data between different GPUs is even more expensive. We're going to talk about how these very classic collective operations like gather and reduce and all reduce are the way to kind of think about um basically distributed, you know, training.

The general game is that we have these model parameters, we have activations, gradients and optimizer states, and they need to be sharded or split across multiple GPUs, and of course, you need to bring the right data to the right nodes to make the compute and write it back. So, there's a whole kind of orchestration that and how to do

that efficiently is going to be the topic of this this unit. And there's multiple ways to to shard.

You can shard by, you know, splitting up your data, splitting up your model, splitting up different layers in the model, splitting up the sequences, splitting up between experts, and we'll talk about the tradeoffs that come with each of these. Okay, and then finally, we're going to talk about inference, um which, as I mentioned, is growing in importance. So, the goal of inference is to actually use the model. So, you know, minor detail here. Um So, inference is Of course, you need to use inference when you're chatting with the model, but it also is useful for reinforcement learning.

It's useful for for doing the rollouts, test time compute, generating synthetic data, evaluation. So, inference is a very critical part of what it means to you know, do language modeling work. So, um we're probably not going to spend as much time as I like on this because the course is already kind of filled up, but we'll see what we can do. Um there was some discussion around whether we would should make you write inference from scratch, but we'll we'll see.

Um Um So, the way to think about inference is that there's two phases, a prefill and a decode. In the prefill, you take the prompt and then you feed all the tokens forward and build key value pairs. This is very much like what happens in training. Um and then then the decoding part, tokens are generated one at a time. And this is the part that is becomes quickly memory bound, and this is why inference is hard, and um so, there's many things you can do to speed up inference. You can try to use a cheaper model by pruning a larger model.

You can quantize, you can distill. Um you can use this technique called speculative decoding where you use a cheaper model to sort of run ahead and guess a bunch of tokens, and now you use the full model, which can operate on those tokens in parallel to to see if it's good. And if you got lucky, then you can accept all those tokens and you're you're much faster than if you're doing one token at a time. And of course, you can do systems optimizations. There's bunch of kernels that are designed specifically for inference. And then one of the interesting stuff things about inference is that if you're running a service, you know, queries are coming at you at potentially different times.

Um and then you have to figure out how to batch them up. Whereas in training, you basically are already defining your batches and everything is much more predictable. So, in assignment two, uh there's going to be implementing kernels um in Triton. Um and doing some sort of parallel training. Um the details here might actually change as um you know, the CAs have grand plans of um revamping the systems. So, the the the assignment might look a bit different from last year's, but it will be cover the roughly the same material.

Um one thing I will mention is that there's this wonderful book out of some Google people called how to scale your model. And I think it's it's really nice for providing a conceptual understanding of like roofline analysis and transformer math and doing LMs conceptually. So, I highly recommend you take a look at that. Now, the only thing is that it's from Google, so it's about TPUs, but a lot of the high level concepts are are similar. And now they have a new chapter on how to think about GPUs.

Okay. So, uh So, the third assignment is about scaling laws. So, by now, you've trained a language model, you can make it go really fast by optimizing kernels in parallel. Now, you want to scale up. So, how do you scale up? So, imagine the following setting. If you had 1e25 flops, so this is tens of millions of dollars of compute, um what what model would you train?

Okay, so this is, you know, I think a kind of a daunting task because if you mess up, well, that's a lot of money down the drain. So, and you can't do your probably typical hyperparameter tuning at that scale because you only get to train one model. And so, this is like the key problem that you have to deal with in in language model large language model training that you don't have to really deal with if you're just fine-tuning a model or doing small scale stuff.

And so, the key conceptual shift here is that we shouldn't think about a single model that we're training, but really think about a scaling recipe. And a scaling recipe is a mapping from a flop budget, let's say 1e25 or 1e24, to a set of hyperparameters, basically a config file. And for a given scaling recipe, what we will do is um run a bunch of experiments to compute the loss that you get at smaller scales, and then you fit a scaling law, and that enables you to predict the loss at a target scale. So, maybe you run some small experiments, you fit the scaling the scaling law, and then you project out what you're going to get at larger scale.

Okay, so that's the primitive. Now, using this, what you can do is now you can optimize the target res the scaling recipe targeting a larger scale using smaller scale experiments, which is wonderful. And second of all, you can predict the loss that you're going to in theory achieve before actually running the experiment. Which go goes allows you to go, you know, raise money and you say, "Well, look, I ran the small scale experiments and I think I can get a really like a GPT-5 level model. Please give me a lot of money so I can train that model."

So, um one thing I think is a maybe another misconception is that scaling laws are not laws of laws of nature. They don't just happen automatically. You kind of have to will them into existence. And this is happens by careful construction of a scaling recipe. Right? And a scaling recipe, remember, it has to extrapolate. So, what this typically means is that you have a sequence of hyperparameters, which say as the scale increases, maybe the learning rate is a constant, maybe it drops, maybe the the batch size increases by how much. And these are things that a scaling recipe has to figure out. Um and so, you parameterize So, then, in order to get these predictable scaling laws, one thing that you actually have to think about is how do you parameterize the model in a way to get what's called hyperparameter transfer. Right? Meaning that this hyperparameter you use at small scale are either the ones that you use at larger scale or are predictable functions of that. Right? Because if every scale, your learning rate acts like sometimes 1e-5 and sometimes 1e-4, then you're not going to be able to magically guess the right learning rate when you at larger scale.

So, one shift in thinking is that the predictability is actually at least as important as optimality. So, you think normally think of oh, we're trying to optimize for, you know, efficiency here and we want to, you know, hyperparameter tune and and make things optimal. And and yes, you do want to do that, but you also want this predictability so that you don't get at larger scale. Okay, so that's this is kind of a stage setting. Um the actual scaling laws we're going to look at are fairly you know, classic. So, these are you know, some of you have might

have seen this idea of well, if you give you a flops budget, should you train how should you balance training a larger model versus training on more tokens?

And this is where the classic computer optimal scaling laws from Kaplan et al. and the Chinchilla so-called Chinchilla scaling laws comes in. The basic idea is that you uh for each uh flops budget, so let's say 6e18 all the way to uh 3e21, you uh sweep across different um you know uh model sizes um and you choose the best one. So, think about minimizing each of these. And then you fit a curve that allows you to uh basically predict the number of parameters given a flops budget.

And if you're lucky, it will lie on a roughly on a line. And if you're unlucky, it's going to be all over the place, which means that you should have no confidence that you're going to be able to uh predict reliably. And um So, you know, the upshot of this is um this is quite crude, but you know, uh a rule of thumb is the um 20 times the number of parameters is the number of data points you should uh train on. So, a 70B parameter model should be trained on roughly 1.4 trillion um tokens. Of course, depending on the data set and architecture, this number will actually vary.

Okay. Also, this doesn't take you into account um the inference cost. Um a lot of models these days are small, but they're trained on way more tokens than is computer optimal because you want a smaller model for inference reasons. Okay. So, uh one fun thing that we've been doing in uh the Moraine project is pre-registering our results. So, we fit a bunch of scaling um you know, uh plots at different compute budgets. We fit a scaling law, and we basically made these uh predictions out to 1e22 flops. Uh this one is actually training. If you go to the Moraine site, you can follow along. Um it should actually be done maybe as earliest tonight. So, maybe on Wednesday I'll I'll report back on how we did and see how we manage uh register the um how we match the pre-registered loss.

So, the idea here is that we made a prediction on if we were to train this large model, which we've never trained before, if we can predict how well it's going to do, then that's that's really uh nice. Okay. So, in assignment three, I think I think this is a kind of a either fun or um you know, s- Actually, it's a fun ass- let's just say it's a fun assignment. Um So, we're going to define this training API, um which basically you give us hyper parameters, and we're going to give you a loss back.

So, what we're going to try to do is simulate what happens if you could do a lot of training runs. And of course, we don't have enough compute to actually allow everyone um to train their own, you know, 8B model or anything. So, basically, we train a bunch of models offline, and we basically provide this cache. Um so, it looks like you're training. Um and what you're going to do is submit training jobs. Um you give us a config. Basically, we give you a a loss uh back. And then you can do whatever you want. I would we would recommend you fit scaling laws um to these uh points, and then you extrapolate, and then we give you a budget, and you're we basically uh evaluate how well your model landed.

So, it is meant to I was going to say it's meant to kind of replicate the high-stress uh scenarios if you actually had a budget that like, you know, $100 million that you needed to spend. Um and you have to be very careful about

what how you uh spend this. Um of course, this is low low stakes. Um Okay. So, so at this point, um you will have you trained a model, you know how to make it fast, you know how to scale up. Um you know, now what's missing?

You know, what do you train the model on? And that's going to be the subject of the data section, um which is, you know, arguably one of the most important things because data quality is is basically specifies how good your model is going to be. Um One way to also frame it is, you know, what do you want your model to do? Right? Data basically is uh reflects what your model wants. So, do you want to speak multiple languages, be good at, you know, having a conversation? Do you want to do run long agentic coding tasks?

Um and so, part of that is also going to, you know, so we're going to start by talking about evaluation, which uh basically defines the capabilities that you like your model to have. Um you know, one um thing we'll talk about is um evaluation is a fairly, you know, deep topic. Um not just it's not just about like, you know, running on some benchmarks. Um there are internal evaluation metrics for model development, and what's matters here is that um you know, smoothness across uh scales so that they're, you know, remember we want things to be predictable. Um relative performance matters. You don't care necessarily how well this uh um you know, does an absolute charge because, you know, let's say perplexity number is just like what is a perplexity 1. you know, two um on some, you know, held-out data really mean?

Um and then there's external metrics. These are things that you report to um your your customers or your reviewers or whoever you're uh presenting your thing to. And here ecologically validity really matters. And I think sometimes these two things two things get completed, but I think they really serve two distinct purposes. And so, you can think about perplexity is something that uh is really helpful for internal development. And still to this day, you know, perplexity is is a very good way of capturing the intrinsic uh quality of a model without getting kind of worrying about bench maxing. Now, there's a separate question of what you run your um evals on. Um and the recommended thing is, well, if you have some data that's not on the internet, that would be good because uh you can avoid uh contamination.

Okay. So, uh if um and then there's uh advanced use cases, which are more uh representative of external-facing um use cases. So, one thing to also note is that, you know, language models are purportedly very general purpose. Right? So, it's suitable only fitting that we actually need a very diverse set of evaluations. And I would always recommend having many evaluations that look at you can average them into a single number, but often that average conflates a lot of different um things.

Okay. So, now after we set up the evals, we know what we're building. Um how do you get the data? Well, first thing is that data does not just uh fall from the sky. Um it has to be actively, you know, curated. Um often, I think um you know, in you know, especially in in classes and also in research, sometimes you're just given a data set, and then it's like, okay, well, now I do stuff on the data set. But um a lot of language models, especially if you want to collect these large data sets, you have to go actively uh look at it. So, web pages are called from the internet. Um there's books, I guess, which is controversial at this at this point, but archive papers, GitHub code, and and so on. This is an old figure from the the pile from 2021, and you can see, you know, language modeling

data sets are fairly uh diverse. Um there's especially these days, I think, a lot of uh contention around is it fair use to train on copyrighted data? Um maybe sometimes you have to license data, and so on. So, there's legal issues around data, which um I think are quite in, you know, important. For example, uh a lot of GitHub code doesn't have a license.

So, uh how do you interpret that? Do you think assume it's permissive, or do you be conservative and assume it's not permissive? Um So, there's also uh the fact is that data is not even text, right? It's either HTML or PDFs, or code is directories, and this requires processing to turn it into actual text to be usable for training. So, that's the top uh topic of data processing. Uh there's a few steps that has to happen here. Transformation, converting some non-text thing into text. Um uh filtering, um keeping only the good stuff. Um if the random internet uh document in Common Crawl is extremely bad, and you the and you don't want to train on it, most likely. Um you want to deduplicate. Um there's multiple sources, so how do you uh combine the different sources?

Um and then finally, more uh recently, there's been a lot of uh work on, you know, generating synthetic data, which could uh mean taking the real data and just rewriting it into things that are more like the downstream task, or or just some more Wikipedia-like, or whatever uh the whatever you want. So, this is an active area of research. Um also, data can be used both for pre-training, uh something called mid-training, which is usually the high-quality data that you put at the the end of the pre-training um step. And this includes long context uh data, such as, you know, maybe big uh code repositories or books.

And then finally there's post-training data, which are you know, maybe conversations or genetic traces with you know, tool calling. So, in assignment four, we're going to make you start with a very raw corpus, like a raw web crawl, and do all the work to um um to filter and to dedupe and make the data uh clean. So, this is I would say I don't know if I would call it not fun, but it is certainly I think a lot of you know, what people would call you know, dirty work, but that is I think part of an important part of building a language model from scratch. So, you have to get the full experience.

Okay. So, finally um alignment um so far we've basically trained a model using full supervision. Predict the next token or the next few tokens. Um now, at this point the model should already be reasonable. Um but we can improve it further by using weak supervision. And why weak supervision? It's because um sometimes it's easier to critique than it is to generate. So, you can't always have data that says this is the right response to this prompt, but maybe you can have a way of specifying what good looks like.

So, then the basic template is that you generate responses from a model, you score them either with a human or verify or LM judge, and then you update the model to prefer better responses. This can be instantiated either through various RL algorithms such as PPO or GRPO, or in a simpler for preference data DPO. Um So, the challenges around RL are that RL algorithms are unstable and uh you know, hard to tune. Some of you probably know this from first-hand you know, experience.

Personally, I prefer to keep things as much in the full truly supervised case as long as possible, and then finally okay, fine, I have to do RL. Um but you know, some people for whatever reason like doing RL. Um Also, what

we'll hopefully talk about this year is that if you do RL at scale and try to maximize your throughput, there's a actually a lot of systems challenges. Um you have to have an inference server and a training server, and then the inference server um has to generate these rollouts, especially if you do RL against environments that involve code execution. It's a whole kind of orchestration game. And then if your workers lag behind or then you get into off-policy issues, and then you're constantly kind of juggling um this on-policyness with a desire to kind of maximize throughput. It's it's a big wonderful mess, which hopefully we'll um talk more about when we come to that.

So, um assignment five, um we're still uh deciding what exactly we want to do. Last year it was implement DPO and GRPO and get it working for some um you know, math benchmark, but um we'll see how how much farther we can push it on the realistic dimension this year. Okay. So, again, remember it's about efficiency, um and efficiency can mean uh either data efficiency or compute efficiency. So, the way to think about it, you have all these resources. You have data, you have the hardware, which has compute uh cores, you have memory, communication bandwidth, and you're just trying to figure out how do you build the best model according to some evaluation um given a fixed set of resources.

So, you can through this lens, I think you can actually think about a lot of these uh design decisions as um optimizing for this. So, systems, clearly that's about um compute efficiency. Um tokenization, as I mentioned before, um it's you know, you can't just work with raw bytes, but that's going to be very compute inefficient, and um at least with today's model architectures. And so, a lot of tokenization is about improving compute efficiency. Model architecture, many of the changes that we'll see um are motivated by reducing the memory or flops.

Um in fact, a lot of them are influenced by the need to have faster inference. Um data filtering, you can also view as uh through efficiency lens. We don't want to waste time updating gradients on a bunch of redundant uh bad data, even if it might not hurt you, but it hurts you in the sense that if you have a fixed compute budget, more time on bad data means less time on good data. And then finally, scaling laws is explicitly about how you can essentially do effective hyperparameter tuning on much smaller models.

And now, tomorrow we might become data constrained in the calculus uh of what design decisions you should take might change, but I think the overall mindset, which we're trying to teach you, is you know, think about the efficiency of your approach. Okay. Um let me stop there. Are there any questions about any of these uh assignments or uh topics?

Okay. So, now let's uh do uh tokenization. So, this is we're jumping into our first uh unit here. Um So, Andrej Karpathy has this really good video on tokenization. You should check it out. Um So, let's starting point is raw text is what is text. It's Unicode strings. Um and on the other hand, the language model places a distribution over sequences of tokens, usually represented as indices. So, we need a procedure that encodes these strings into tokens, and also a procedure that decodes tokens back into strings. So, a tokenizer is basically something that can do this round trip.

So, here are some examples to give you a flavor um for how tokenizers work. Actually, I should have tried to get internet earlier, so this is not going to work. Okay, if you go to the site, you can play around with different tokenizers. Um So, some observations here. Um there's and you'll you'll appreciate why tokenizers are kind of annoying, why people want to get rid of them. So, a word and its uh and a word with conglomerate with its preceding space um are different you know, tokens.

Um so, many of tokens you'll actually see are space word, which is you know, fine, but kind of strange. Um So, this hello and this uh hello is actually two completely different indices that have nothing to do with each other. So, um and sometimes, depending on the tokenizer you use, numbers are represented with um this every few digits is a token. Sometimes it's predictable, and sometimes it's it's not. Some tokenizers try to make every digit a token, but then you're blowing up the number of tokens you have, so there's some trade-off there.

Um So, here's the um GPT-5 tokenizer. Uh so, uh it can take the string and uh convert it into you know, these indices. Um and then you can uh decode it back into the string. And a tokenizer should round trip. If you implement a tokenizer that doesn't round trip, you have a problem. Um So, the the compression ratio here is the number of bytes per token.

So, in this case, um we have the number of bytes is of this string is 20. The number of tokens is eight, and you do 20 / 8, so the compression ratio is 2.5. Okay, so 2.5 bytes per token. The larger the compression ratio, that means the shorter the sentence, which is good because attention is quadratic, and you want to make sure the sent sequence is shorter. Um Now, you could obviously increase the compression ratio by increasing the vocab size, um but then you get into sparsity, where more and more um because every element of vocab is treated as like a as a distinct element.

Um so, these days uh tokenizers, um especially multilingual tokenizers, have you know, 100k or 200k um you know, tokens. Distinct tokens. Um so, uh you can look at the GPT token vocabulary. I think we'll skip it this in the interest of time. Okay, so how do you build a tokenizer? So, I'm going to go through this fast, but um So, the first thing you might do is like, well, Unicode string, that's a sequence already of Unicode characters, and each character is an integer, which you can call ORD in Python, and you get some number out.

And this can be converted into characters, so let's just build a character-level tokenizer, which basically breaks up each character and encodes it in a token, and then this can decode back. Like this is good. So, now, there are 150k uh Unicode characters, so your vocab size could be 150k, um which is which is you know, a lot I I mean, it's not crazy, but um I think the bigger problem is that many characters are actually rare.

Um which means that it's really an inefficient use of a vocabulary. And and also the compression ratio, um which kind of reflects this is not that great. So, most of the time you're actually using, um, a lot of tokens to represent your your sequence and many of the the the the indices, um, are actually not being very used. So, this is like not a very good tokenizer. Um, so here's another attempt. So, you can turn strings into bytes.

So, Unicode has a UTF-8 encoding, which means that um, you can sometimes, uh, a string like A is just one byte

and sometimes a string is, uh, you know, multiple bytes. So, let's build a tokenizer around that. So, um, we can take this string and convert it into a sequence of bytes and notice that this is a longer sequence now and but all of the all the numbers are between 0 and 255 because that's what a byte means.

Um, and the compression ratio is, uh, is one, which is, uh, which is not great. Right? So, byte sequences are can be very long, um, but the vocab size is, uh, small. Okay, so now, okay, so the both of these are like really bad. So, here's let's try to make some progress. So, this is what actually people used to do in NLP, um, if, uh, people remember. So, if you take a string, I can just chunk it up into, uh, break it up by spaces or some regular expression, um, and let's just call each of these chunks, uh, a token.

Okay, so what is good about this one is that each token is meaningful because humans invented words and words tend to have a stable semantic meaning, um, so in there but your vocab size is the number of distinct chunks in the training data, which could be all a lot. And also your, uh, compression ratio, I mean, it's it's it's quite, you know, good but the vocabulary can be huge. Actually, it's worse than that because, um, the, uh, the vocabulary could be actually unbounded, right? Because at test time you might get some sequence and you tokenize and then you have a token you've never seen before and people used to assign these like unk token but that's like really ugly and can mess up your perplexity uh, calculations.

So, um, this is also not not great. Okay, so what we're actually going to do is called byte pair encoding. And this was introduced a long time ago for data compression way before language models, um, uh, were really on the scene, really. Uh, it was first introduced to NLP for doing neural machine translation, um, and the first, uh, paper that, uh, used BPE to, uh, for LMs was, uh, GPT-2.

So, the basic idea is you're going to train the tokenizer on raw text to construct a vocabulary that's tailored to the data. And you're also going to have this property that everything becomes can be tokenized if it's rare, then it just breaks up into smaller units rather than having this unk token. Um, so so common sequences can be, um, are going to be represented as one token, rare sequences are going to be split into multiple tokens. That's the idea.

Okay, so the algorithm is fairly, uh, simple conceptually. So, you start basically with your your corpus, let's assume it's one long sequence. Um, you get a byte sequence, each byte starts as a token and then we're going to merge, um, successive pairs of adjacent tokens that are occurred the most frequently. Okay, so let's step through how this is going to work in code. So, here's a simple string, the cat in the hat, and, uh, here's the implementation of the BPE algorithm.

Um, so we're going to, uh, turn that into a sequence of bytes, um, and then we're going to um, first, we're going to basically count the number of times successive tokens appear. So, 116 104 shows up twice. Um, so we get this and then we're going to find the um, the pair that uh, happens most number of times. So, that's 116 104. I guess there's a few ties but we'll just take the first one.

And then we're going to merge that pair. And by merging that pair, what we do is we create a new token. In this case, um, this is going to be called token 256. It's going to represent this pair and we're going to add it to our

vocabulary. So, 256 is going to represent the sequence th. So, t and h have been merged and we're going to call every time we see th, that's going to we're going to use 256 to represent that.

And then we go through indices and then we replace every occurrence of 116 104 with 256. So, those two places have been replaced. And then we iterate. So, the next time, uh, we do this, we're going to find 256 and 101, we're going to merge that and now we have 257 and then, uh, we're going to merge that, um, one more time and we're going to get 258. So, over time, the sequence is, uh, shrinking, um, and the vocabulary size is growing.

Okay, so, um, let me and the compression ratio, uh, here is that we get for this, I mean, this toy example is 1.5. Um, okay, so now that you have a tokenizer, um, how do you tokenize new text? Well, um, you take a new string and you encode it and conceptually what happens is that, um, you basically go through the set of merges, uh, that you've made and then you just apply, uh, the merges to your your string.

Okay, let me not actually not step through that code. Um, so that will give you a sequence. Um, so this is the sequence, uh, encoding of the quick brown fox and then, uh, when you decode it, you get the same thing, you know, back. So, um, I went to through this a bit fast just in the interest of time. Um, I will say that this, uh, implementation works. This is a full-blown BPE implementation. It's extremely slow.

So, in assignment one, we're going to ask you to, um, basically make it faster. Um, so currently encode, uh, loops over all the, you know, merges, which is very slow because you might have the number of merges you have is essentially essentially, um, the vocab size minus, you know, 256. Um, so you only want to loop over the merges that that matter. Um, and you have to build some indices to make that, you know, happen. Um, there's some, you know, details around special tokens. Conceptually not deep but important to building a modern tokenizer. Um, another thing is that I've presented the tokenizer just for simplicity as you take an entire string and then you try to tokenize it. Really, what happens is that you break it up into, uh, your text into chunks and then you apply tokenizer on each, uh, chunk. So, that's going to be much faster.

Um, and then, you know, and then try to make it as fast as possible. Um, at some point you might realize that Python is just not very fast and if you want to implement it in, you know, say your favorite language, uh, Rust or C or something, then go for it. Okay, so quick summary. Tokenizers convert between strings and tokens or indices. Um, the previous character-based, byte-based, word-based are uh, highly suboptimal in this, uh, in their own way.

BPE is is, um, effective heuristics that is data-driven so, uh, um, it seems to be uh, pretty effective. Now, like I said before, uh, maybe next year I don't have to teach this but, uh, for this year we're stuck with tokenization. Um, you know, even if we get rid of tokenization though, um, I think whatever solution replaces it, I think has to satisfy the following properties, right? If you have the model, the transformer needs to operate on some sort of abstractions of the sequence.

And this is most evident if you think about, you know, um, not just text but video or or DNA, um, sequences

where, um, you know, the individual, um, bytes or units are actually quite kind of low signal to noise, um, and you have to do some sort of abstraction to lift it into a place where you can do modeling on that. Um, and then finally, as I mentioned, um, chunks should be kind of variable. You want adaptive computation. Um, not all, you know, bytes are treated the same and if you don't do that, I think you're going to be suboptimal. So, if any end-to-end solution also, I think has to have these properties.

Okay, so with that, I will end. Next time on Wednesday, we're going to start the unit on resource accounting, so which is sort of a, you know, a baby systems, I would say. And then after that, we're going to go back into architectures and, uh, go from there. All right.