00:00:00 intro: Lets reproduce GPT-2 (124M)

hi everyone so today we are going to be continuing our Zero to Hero series and in particular today we are going to reproduce the gpt2 model the 124 million version of it so when openi released gpt2 this was 2019 and they released it with this blog post on top of that they released this paper and on top of that they released this code on GitHub so open a/ gpt2 now when we talk about reproducing gpt2 we have to be careful because in particular in this video we're going to be reproducing the 124 million parameter model so the thing to realize is that there's always a miniseries when these are releases are made so there are the gpt2 miniseries made up of models at different sizes and usually the biggest model is called the gpt2 but basically the reason we do that is because you can put the model sizes on the x-axis of plots like this and on the Y AIS you put a lot of uh Downstream metrics that you're interested in like translation summarization question answering and so on and you can chart out these scaling laws so basically as the model size increases you're getting better and better at Downstream metrics and so in particular for gpt2 if we scroll down in paper there are four models in the gpt2 miniseries starting at 124 million all the way up to 1558 million now the reason my numbers the way I say them disagree with this table is that this table is wrong if you actually go to the uh gpt2 uh GitHub repo they sort of say that um there was an error in how they added up the parameters but basically this is the 124 million parameter model Etc so the 124 million parameter had 12 layers in the Transformer and it had 768 channels in the Transformer 768 dimensions and I'm going to be assuming some familiarity with what these terms mean because I covered all of this in my previous video let's build gpt2 uh let's build GPT from scratch so I covered that in the previous video in this playlist now if we do everything correctly and everything works out well by the end of this video we're going to see something like this where we're looking at the validation loss which basically um measures how good we are at predicting the next token in a sequence on some validation data that the model has not seen during training and we see that we go from doing that task not very well because we're initializing from scratch all the way to doing that task quite well um by the end of the training and hopefully we're going to beat the gpt2 uh 124 M model now previously when they were working on this this is already 5 years ago so this was probably a fairly complicated optimization at the time and the gpus and the compute was a lot smaller today you can reproduce this model in roughly an hour or probably less even and it will cost you about 10 bucks if you want to do this on the cloud uh Cloud Compu a sort of computer that you can all rent and if you pay $10 for that computer you wait about an hour or less you can actually achieve a model that is as good as this model that open ey released and uh one more thing to mention is unlike many other models open ey did release the weights for gpt2 so those weights are all available in this repository but the gpt2 paper is not always as good with all of the details of training so in addition to the gpt2 paper we're going to be referencing the gpt3 paper which is a lot more Concrete in a lot of the hyp parameters and optimization settings and so on um and it's not a huge departure in the architecture from the GPT 2 uh version of the model so we're going to be referencing both gpt2 and gpt3 as we try to reproduce gpt2 124 M uh so let's go so the first thing I

00:03:39 exploring the GPT-2 (124M) OpenAI checkpoint

would like to do is actually start at the end or at the Target so in other words let's load the GPT to 124 M model as it was released by openi and maybe take it for a spin let's sample some tokens from it now the issue with that is when you go into the code base of gpt2 and you go into the source and you click in on the model. pi you'll realize that actually this is using tensorflow so the original gpt2 code here was written in tensor flow which is um you know not let's just say not used as much anymore um so we'd like to use pytorch uh because it's a lot friendlier easier and I just personally like a lot more the problem with that is the initial code is intenser flow we'd like to use pytorch so instead uh to get the target we're going to use the hugging face Transformers um code which I like a lot more so when you go into the Transformers source Transformers models gpt2 modeling gpt2 Pi you will see that they have the gpt2 implementation of that Transformer here in this file um and it's like medium readable but not fully readable um but what it does is it did all the work of converting all those weights uh from tensor flow to pytorch Friendly and so it's much easier to load and work with so in particular we can look at the gpt2 um model here and we can load it using hugging face Transformers so swinging over this is what that looks like from Transformers import the DP GT2 LM head model and then from pre-train gpt2 uh now one awkward thing about this is that when you do gpt2 as the model that we're loading this actually is the 124 million parameter model if you want the actual the gpt2 the 1.5 billion then you actually want to do- XL so this is the 12 4 M our Target now what we're doing is when we actually get this we're initializing the uh pytorch NN module as defined here in this class from it I want to get just the state dict which is just a raw tensors so we just have um the tensors of that file and by the way here this is a jupyter notebook uh but this is jupyter notebook running inside vs code uh so I like to work with it all in a single sort of interface so I like to use vs code so this is the jupyter notebook extension inside the es code so when we get the state dick this is just a dict so we can print the key and the value which is the tensor and let's just look at the shapes so these are sort of the uh different parameters inside the gbt2 model and their shape so the W weight for token embedding is of size 50257 by 768 where this is coming from is that we have 50257 tokens in the gpt2 vocabulary um and the tokens by the way these are exactly the tokens that we spoken about in the previous video on my tokenization Series so the previous videos just before this I go into a ton of detail on tokenization gpt2 tokenizer happens to have this many tokens for each token we have a 768 dimensional embedding that is the distributed representation that stands in for that token so each token is a little string piece and then the 768 numbers are the vector that represents that token and so this is just our lookup table for tokens and then here we have the lookup table for the positions so because gbt2 has a maximum sequence length of 1024 we have up to 1,24 positions that each token can be attending to in the past and every one of those positions in gpd2 has a fixed Vector of 768 that is learned by optimization um and so this is the position embedding and the token embedding um and then everything here is just the other weights and biases and everything else of this Transformer so when you just take for example the positional embeddings and flatten it out and take just the 20 elements you can see that these are just the parameters these are weights floats just we can take and we can plot them so these are the position embeddings and we get something like this and you can see that this has structure and it has structure because what we what we have here really is every Row in this visualization is a different position a fixed absolute position in um the range from 0 to 1024 and each row here is the representation of that position and so it has structure because these positional embeddings end up learning these sinusoids and cosiness um that sort of like represent each of these positions and uh each row here stands in for that position and is processed by the Transformer to recover all the relative positions and uh sort of realize which token is where and um attend to them depending on their position not just their content so when we actually just look into an individual column inside these and I just grabbed three random columns you'll see that for example here we are focusing on every every single um Channel and we're looking at what that channel is doing as a function of uh position from one from Z to 1223 really and we can see that some of these channels basically like respond more or less to different parts of the position Spectrum so this green channel uh really likes to fire for everything after 200 uh up to 800 but not less a lot less and has a sharp drop off here near zero so who knows what these embeddings are doing and why they are the way they are you can tell for example that because they're a bit more Jagged and they're kind of noisy you can tell that this model was not fully trained and the more trained this model was the more you would expect to smooth this out and so this is telling you that this is a little bit of an undertrained model um but in principle actually these curves don't even have to be smooth this should just be totally random noise and in fact in the beginning of the optimization it is complete random noise because this position embedding table is initialized completely at random so in the beginning you have jaggedness and the fact that you end up with something smooth is already kind of impressive um that that just falls out of the optimization because in principle you shouldn't even be able to get any single graph out of this that makes sense but we actually get something that looks a little bit noisy but for the most part looks sinusoidal like um in the original Transformer um in the original Transformer paper the attention is all you need paper the positional embeddings are actually initialized and fixed if I remember correctly to sinusoids and cosiness of uh different frequencies and that's the positional coding and it's fixed but in gpt2 these are just parameters and they're trained from scratch just like any other parameter uh and that seems to work about as well and so what they do is they kind of like recover these sinusoidal like features during the optimization we can also look at any of the other matrices here so here I took the first layer of the Transformer and looking at like one of its weights and just the first block of 300 by 300 and you see some structure but like again like who knows what any of this is if you're into mechanistic interpretability you might get a real kick out of trying to figure out like what is going on what is this structure and what does this all mean but we're not going to be doing that in this video but we definitely see that there's some interesting structure and that's kind of cool what we're mostly interested in is we've loaded the weights of this model that was released by open Ai and now using the hogging face Transformers we can not just get all the raw weights but we can also get the um what they call Pipeline and sample from it so this is the prefix hello I'm a language model comma and then we're sampling uh 30 tokens and we getting five sequences and I ran this and this is what it produced um hell language model but what I'm really doing is making a human readable document there are other languages but those are dot dot dot so you can read through these if you like but basically these are five different completions of the same prefix from this uh gbt 2124m now uh if I go here I took this example from here and sadly even though we are fixing the seed we are getting different Generations from the snippet than what they got so presumably the code changed um but what we see though at this stage that's important is that we are getting coherent text so we've loaded the model successfully we can look at all its parameters and the keys tell us where in the model these come from and we want to actually write our own gpt2 class so that we have full understanding of what's happening there we don't want to be working with something like uh the modeling gpt2 Pi because it's just too complicated we want to write this from scratch ourselves so we're going to be implementing the GPT model here in parallel and as our first task let's load the gpt2 124 M into the class that we're going to develop here from scratch that's going to give us confidence that we can load the open ey model and therefore there's a setting of Weights that exactly is the 124 model but then of course what we're going to do is we're going to initialize the model from scratch instead and try try to train it ourselves um on a bunch of documents that we're going to get and we're going to try to surpass that model so we're going to get different weights and everything's going to look different hopefully better even um but uh we're going to have a lot of confidence that because we can load the openi model we are in the same model family and model class and we just have to ReDiscover a good setting of the weights uh but from scratch so let's now write the gbt2 model and let's load the weights and make sure that we can also generate text that looks coherent okay

00:13:47 SECTION 1: implementing the GPT-2 nn.Module

so let's now swing over to the attention is all un need paper that started everything and let's scroll over to the model architecture the original Transformer now remember that gpt2 is slightly modified from the or or Transformer in particular we do not have uh the encoder gpt2 is a decoder only Transformer as we call it so this entire encoder here is missing in addition to that this cross attention here that was using that encoder is also missing so we delete this entire part everything else stays almost the same but there are some differences that we're going to uh sort of look at here so there are two main differences when we go to the gb2 page under 2.3 model we notice that first there's a reshuffling of the layer Norms so they change place and second an additional layer normalization was added here to the final self detention block so basically all the layer Norms here instead of being after the MLP or after the attention they SN before it and an additional layer Norm gets added here right before the final classifier so now let's Implement some of the first sort of skeleton NN module modules here in our GPT NN module and in particular we're going to try to match up this schema here that is used by hugging face Transformers because that will make it much easier to load these weights from this state dict so we want something that reflects uh this schema here so here's what I came up with um basically we see that the main container here that has all the modules is called Transformer so I'm reflecting that with an NN module dict and this is basically a module that allows you to index into the subm modules using keys just like a dictionary uh strings within it we have the weights of the token embeddings WT and that's an N embedding and the weights of the position embeddings which is also just an N embedding and if you remember n embedding is really just a fancy little wrapper module around just a single um single array of numbers a single uh block of numbers just like this it's a single tensor and an embedding is a glorified um wrapper around a tensor that allows you to access its elements uh by indexing into the rows now in addition to that we see here that we have a h and then there's a this is index using numbers instead of indexed using strings so there's a h. 0 1 2 Etc all the way up till h. 11 and that's because there are 12 layers here in this Transformer so to reflect that I'm creating also an H I think that probably stands for hidden and instead of a module dict this is a model list so we can index it using integers exactly as we see here 01 2 Etc and the modular list has a n layer blocks and the blocks are yet to be defined in a module in a bit in addition to that following the gpt2 paper we have we need an additional final layer Norm that we're going to put in there and then we have the final classifier uh the language model head which um projects from 768 the number of embedding dimensions in this GPT all the way to the vocab size which is 50257 and gpt2 uses no bias for this final uh sort of projection so this is the skeleton and you can see that it reflects this so the wte is the token embeddings here it's called output embedding but it's really the token embeddings the PE is the positional codings uh those two pieces of information as we saw previously are going to add and then go into the Transformer the H is the all the blocks in Gray and the LNF is this new layer that gets added here by the gpt2 model and LM head is this linear part here so that's the skeleton of the gpt2 we now have to implement the block okay so let's now recurse to the block itself so we want to define the block um so I'll start putting them here so the block I like to write out like this uh these are some of the initializations and then this is the actual forward pass of what this block computes and notice here that there's a change from the Transformer again that is mentioned in the gpt2 paper so here the layer normalizations are after the application of attention or feed forward in addition to that note that the normalizations are inside the residual stream you see how feed forward is applied and this arrow goes through and through the normalization so that means that your residual pathway has normalizations inside them and this is not very good or desirable uh you actually prefer to have a single uh clean residual stream all the way from supervision all the way down to the inputs the tokens and this is very desirable and nice because the gradients that flow from the top if you remember from your microad addition just distributes gradients during the backwards state to both of its branches equally so addition is a branch in the gradients and so that means that the gradients from the top flows straight to the inputs the tokens through the residual Pathways unchanged but then in addition to that the gradient also flows through the blocks and the blocks you know contribute their own contribution over time and kick in and change the optimization over time but basically clean residual pathway is desirable from an optimization perspective and then the this is the pre-normalization version where you see that RX first goes through the layer normalization and then the attention and then goes uh back out to go to the L ration number two and the multia perceptron sometimes also referred to as a feed forward Network or an FFN and then that goes into the residual stream again and the one more thing that is kind of interesting to note is that recall that attention is a communication operation it is where all the tokens and there's 1,24 tokens lined up in a sequence and this is where the tokens communicate this is where they exchange information so attention is a um aggregation function it's a pooling function it's a weighted sum function it is a reduce operation whereas MLP this uh MLP here happens at every single token individually there's no information being collected or exchanged between the tokens so the attention is the reduce and the MLP is the map and what you end up with is that the Transformer just ends up just being a repeated application of map produce if you want to think about it that way so um this is where they communicate and this is where they think individually about the information that they gathered and every one of these blocks uh iteratively refines the um representation is at the residual stream so this is our block um slightly modified from this picture Okay so let's now move on to the MLP so the MLP block uh I implemented as follows it is relatively straightforward we basically have two linear projections here that are sandwiched in between the G nonlinearity so nn. G approximate is 10h now when we swing on uh swing over to the Pyro documentation this is n.g and it has this format and it has two versions the original version of G which we'll step into into in a bit and the approximate version of Galo which we can request using 10 so as you can see just as a preview here G is a basically like a reu except there's no flat exactly Flat Tail here at exactly zero but otherwise it looks very much like a slightly smoother reu it comes from this paper here Gan error linear units and uh you can step through this paper and there's some mathematical calac reasoning that leads to an interpretation that leads to the specific formulation it has to do with stochastic radial risers and the expectation of a modification to Adaptive dropout so you can read through all of that if you'd like here and there's a little bit of history as to why there is an an approximate version of G and that comes from this issue here as far as I can tell and in this issue Daniel Hendrix mentions that at the time when they developed this nonlinearity the Earth function which you need to evaluate the exact G was very slow in tensor flow so they ended up basically developing this approximation and this approximation that then ended up being picked up by Bert and by GP P2 Etc but today there's no real good reason to use the approximate version you'd prefer to just use the exact version um because I my expectation is that there's no big difference anymore and this is kind of like a historical um kind of Quirk um but we are trying to reproduce gpt2 exactly and gpt2 used the 10h approximate version so we prefer to stick with that um now one other reason to actually just intuitively use G instead of veru is previously in the in videos in the past we've spoken about the dead reu neuron problem where in this tale of a reu if it's exactly flat at zero any activations that fall there will get exactly zero gradient there's no change there's no adaptation there's no development of the network if any of these activations end in this flat region but the G always contributes a local gradient and so there's always going to be a change always going to be an adaptation and sort of smoothing it out ends up empirically working better in practice as demonstrated in this paper and also as demonstrated by it being picked up by the bird paper gbt2 paper and so on so for that reason we adopt this nonlinearity uh here in the 10 in the gbt2 reproduction now in more modern networks also like llama 3 and so on this nonlinearity also further changes uh to swiglo and other variants like that uh but for gpt2 they Ed this approximate G okay and finally we have the attention operation so let me paste in my attention so I know this is a lot so I'm going to go through this a bit quickly a bit slowly but not too slowly because we have covered this in the previous video and I would just point you there um so this is the attention operation now in the previous video you will remember this is not just attention this is um multi-headed attention right and so in the previous video we had this multi-headed attention module and this implementation made it obvious that these heads are not actually that complicated uh there's basically in parallel inside every attention block there's multiple heads and they're all functioning in parallel and uh their outputs are just being concatenated and that becomes the output of the multi-headed attention so the heads are just kind of like parallel streams and their outputs get concatenated and so it was very simple and made the head be kind of like U fairly straightforward in terms of its implementation what happens here is that instead of having two separate modules and indeed many more modules that get concatenated all of that is just put into a single uh self attention uh module and instead I'm being very careful and doing a bunch of transpose split um tensor gymnastics to make this very efficient in pych but fundamentally and algorithmically nothing is different from the implementation we saw before um in this uh give repository so to remind you very briefly and I don't want to go in this uh into this in too many in too much time but we have these tokens lined up in a sequence and there's 1,20 of them and then each token at this stage of the attention emits three vectors the query key and the value and first what happens here um is that the queries and the keys have to multiply each other to get sort of the attention um amount like how interesting they find each other so they have to interact multiplicatively so what we're doing here is we're calculating the qkv we splitting it and then there's a bunch of gymnastics as I mentioned here and the way this works is that we're basically making the number of heads and H into a batch Dimension and so it's a batch Dimension just like B so that in these operations that follow pytorch treats B and NH as batches and it applies all the operations on all of them in parallel in both the batch and the heads and the operations that get applied are number one the queries and the keys intera to give us her attention this is the autoaggressive mask that makes sure that the tokens only attend to tokens before them and never to tokens in the future the softmax here normalizes the attention so it sums to one always and then recall from the previous video that doing the attention Matrix multiply with the values is basically a way to do a weighted sum of the values of the tokens that we found interesting at every single token and then the final transpose conf VI and view is just reassembling all of that again and this actually performs the concatenation operation so you can step through this uh slowly if you'd like um but it is equivalent mathematically to our previous implementation is just more efficient in P torch so that's why I chose this implementation instead now in addition to that I'm being careful with how I name my variables so for example cattin is the same as seaten and so actually our keys should basically exactly follow the schema of the hugging face train Transformers code and that will make it very easy for us to now Port over all the weights from exactly this sort of naming conventions because all of our variables are named the same thing but um at this point we have finished the gpt2 implementation and what that allows us to do is we don't have to basically use uh this file from hugging face which is fairly long um this is uh 2,000 lines of code um instead we just have a less than 100 lines of code and this is the complete uh gpd2 implementation so at this stage we should just be able to take over all the weights set them and then do generation so let's see what that looks like okay

00:28:08 loading the huggingface/GPT-2 parameters

so here I've also changed the GPT config so that the numbers here the H parameters agree with the gpt2 124 M model so the maximum sequence length which I call block size here is 124 the number of tokens is 50250 257 which if you watch my tokenizer video know that this is 50,000 m merges BP merges 256 bite tokens the leaves of the BP tree and one special end of text token that delimits different documents and can start generation as well and there are 12 layers there are 12 heads in the attention and the dimension of the Transformers was 768 so here's how we can now load the parameters from hugging face to uh our code here and initialize the GPT class with those parameters so let me just copy paste a bunch of code here and I'm not going to go through this code too slow too quickly too slowly because um honestly it's not that interesting it's not that exciting we're just loading the weights so it's kind of dry but as I mentioned there are four models in this miniseries of gpt2 this is some of the Jupiter code um code that we had here on the right I'm just pting it over these are the hyper parameters of the gpt2 models uh we're creating the config object and creating our own model and then what's Happening Here is we're creating the state dict both for our model and for the hugging face model um and then what we're doing here is we're going over the hugging face model keys and we're copying over those tensors and in the process we are kind of ignoring a few of the buffers they're not parameters they're buffers so for example attention dobias uh that's just used for the autoaggressive mask and so we are ignoring some of those masks and uh that's it and then then one additional kind of annoyance is that this comes from the tensorflow repo and I'm not sure how this is a little bit annoying but some of the weights are transposed from what pytorch would want and so manually I hardcoded the weights that should be transposed and then we transpose them if that is so and then we return this model so the from pre-trained is a Constructor or class method in Python that Returns the GPT object if we just give it the model type which in our case is gpt2 the smallest model that we're interested in so this is the code and this is how you would use it and um we can pop open the terminal here in vs code and we can python train gbt2 pi and fingers crossed okay so we didn't crash and so we can load the weights and the biases and everything else into our Ann module but now let's also get additional confidence that this is working and let's try to actually generate from this

00:31:00 implementing the forward pass to get logits

model okay now before we can actually generate from this model we have to be able to forward it we didn't actually write that code yet so here's the forward function so the input to the forward is going to be our indices our tokens uh token indices and they are always of shape B BYT and so we have batch dimension of B and then we have the time dimension of up to T and the T can't be more than the block size the block size is is the maximum sequence length so B BYT indices arranged is sort of like a two-dimensional layout and remember that basically every single row of this is of size up to uh block size and this is T tokens that are in a sequence and then we have B independent sequences stacked up in a batch so that this is efficient now here we are forwarding the position embeddings and the token embeddings and this code should be very recognizable from the previous lecture so um we basically use uh a range which is kind of like a version of range but for pytorch uh and we're iterating from Z to T and creating this uh positions uh sort of uh indices um and then we are making sure that they're in the same device as idx because we're not going to be training on only CPU that's going to be too inefficient we want to be training on GPU and that's going to come in in a bit uh then we have the position embeddings and the token embeddings and the addition operation of those two now notice that the position embed are going to be identical for every single row of uh of input and so there's broadcasting hidden inside this plus where we have to create an additional Dimension here and then these two add up because the same position embeddings apply at every single row of our example stacked up in a batch then we forward the Transformer blocks and finally the last layer norm and the LM head so what comes out after forward is the logits and if the input was B BYT indices then at every single B by T we will calculate the uh logits for what token comes next in the sequence so what is the token B t+1 the one on the right of this token and B app size here is the number of possible tokens and so therefore this is the tensor that we're going to obtain and these low jits are just a softmax away from becoming probabilities so this is the forward pass of the network and now we can get load and so we're going to be able to generate from the model imminently okay so now we're going to

00:33:31 sampling init, prefix tokens, tokenization

try to set up the identical thing on the left here that matches hug and face on the right so here we've sampled from the pipeline and we sampled five times up to 30 tokens with the prefix of hello I'm a language model and these are the completions that we achieved so we're going to try to replicate that on the left here so number turn sequences is five max length is 30 so the first thing we do of course is we initialize our model then we put it into evaluation mode now this is a good practice to put the model into eval when you're not going to be training it you're just going to be using it and I don't actually know if this is doing anything right now for the following reason our model up above here contains no modules or layers that actually have a different uh Behavior at training or evaluation time so for example Dropout batch norm and a bunch of other layers have this kind of behavior but all of these layers that we've used here should be identical in both training and evaluation time um so so potentially model that eval does nothing but then I'm not actually sure if this is the case and maybe pytorch internals uh do some clever things depending on the evaluation mode uh inside here the next thing we're doing here is we are moving the entire model to Cuda so we're moving this all of the tensors to GPU so I'm sshed here to a cloud box and I have a bunch of gpus on this box and here I'm moving the entire model and all of its members and all of its tensors and everything like that everything gets shipped off to basically a whole separate computer that is sitting on the GPU and the GPU is connected to the uh CPU and they can communicate but it's basically a whole separate computer with its own computer architecture and it's really well catered to parallel processing tasks like those of running neural networks so I'm doing this so that the model lives on the GPU a whole separate computer and it's just going to make our code a lot more efficient because all of this stuff runs a lot more efficiently on the gpus so that's the model itself now uh the next thing we want to do is we want to start with this as the prefix when we do the generation so let's actually create those prefix tokens so here's the code that I've written we're going to import the tich token library from open Ai and we're going to get the gpt2 encoding so that's the tokenizer for gpt2 and then we're going to encode this string and get a list of integers which are the tokens uh now these integers here should actually be fairly straightforward because we can just copy paste this string and we can sort of inspect what it is in tick tokenizer so just pasting that in these are the tokens that are going to come out so this list of integers is what we expect tokens to become and as you recall if you saw my video of course all the tokens they're just little string chunks right so these are this is the chunc of this string into gpt2 tokens so once we have those tokens it's a list of integers we can create a torch tensor out of it in this case it's eight tokens and then we're going to replicate these eight tokens for five times to get five rows of eight tokens and that is our initial um input X as I call it here and it lives on the GPU as well so X now is this idx that we can put into forward to get our logits so that we know what comes as the sixth token uh sorry as the ninth token in every one of these five rows okay and we are now

00:37:02 sampling loop

ready to generate so let me paste in one more code block here um so what's happening here in this code block is we have this x which is of size B BYT right so batch by time and we're going to be in every iteration of this loop we're going to be adding a column of new indices into each one of these rows right and so these are the new indices and we're appending them to the the sequence as we're sampling so with each Loop iteration we get one more column into X and all of the operations happen in the context manager of torch. nograd this is just telling pytorch that we're not going to be calling that backward on any of this so it doesn't have to cach all the intermediate tensors it's not going to have to prepare in any way for a potential backward later and this saves a lot of space and also possibly uh some time so we get our low jits we get the loow jits at only the last location we throw away all the other low jits uh we don't need them we only care about the last columns low jits so this is being wasteful uh but uh this is just kind of like an inefficient implementation of sampling um so it's correct but inefficient so we get the last column of loow jits pass it through soft Max to get our probabilities then here I'm doing top case sampling of 50 and I'm doing that because this is the hugging face default so just looking at the hugging face docks here of a pipeline um there's a bunch of quarks that go into hugging face and I mean it's it's kind of a lot honestly but I guess the important one that I noticed is that they're using top K by default which is 50 and what that does is that uh so that's being used here as well and what that does is basically we want to take our probabilities and we only want to keep the top 50 probabilities and anything that is lower than the 50th probability uh we just clamp to zero and renormalize and so that way we are never sampling very rare tokens uh the tokens we're going to be sampling are always in the top 50 of most likely tokens and this helps keep the model kind of on track and it doesn't blabber on and it doesn't get lost and doesn't go off the rails as easily uh and it kind of like um sticks in the vicinity of likely tokens a lot better so this is the way to do it in pytorch and you can step through it if you like I don't think it's super insightful so I'll speed through it but roughly speaking we get this new column of of tokens we append them on x and basically The Columns of X grow until this y Loop gets tripped up and then finally we have an entire X of size um 5 by 30 in this case in this example and we can just basically print all those individual rows so I'm getting all the rows I'm getting all the tokens that were sampled and I'm using the decode function from Tik tokenizer to get back the string which we can print and so terminal new terminal and let me python train gpt2 okay so these are the generations that we're getting hello I'm a language model not a program um new line new line Etc hello I'm a language model and one of the main things that bothers me when they create languages is how easy it becomes to create something that I me so this will just like blabber on right in all these cases now one thing you will notice is that these Generations are not the generations of hugging face here and I can't find the discrepancy to be honest and I didn't fully go through all these options but probably there's something else hiding in on addition to the top P so I'm not able to match it up but just for correctness um down here Below in the juper notebook and using the hugging face model so this is the hugging face model here I was I replicated the code and if I do this and I run that then I am getting the same results so basically the model internals are not wrong it's just I'm not 100% sure what the pipeline does in hugging face and that's why we're not able to match them up but otherwise the code is correct and we've loaded all the um tensors correctly so we're initializing the model correctly and everything here works so long story short uh We've Port it all the weights we initialize the gpt2 this is the exact opening gpt2 and it can generate sequences and they look sensible and now here of course we're initializing with gbt2 model weights but now we want to initialize from scratch from random numbers and we want to actually train a model that will give us sequences as good as or better than these ones in quality and so that's what we turn to next so it turns out that using the

00:41:47 sample, auto-detect the device

random model is actually fairly straightforward because pytorch already initializes our model randomly and by default so when we create the GPT model and the Constructor this is all um all of these layers and modules have random initializers that are there by default so when these linear layers get created and so on there's default Constructors for example using the Javier initialization that we saw in the past uh to construct the weights of these layers and so creating a random model instead of a gpt2 model is actually fairly straightforward and we would just come here and instead we would create model equals GPT and then we want to use the default config GPT config and the default config uses the 124 M parameters so this is the random model initialization and we can run it and we should be able to get uh results now the results here of course are total garbage carbal and that's because this is random model and so we're just getting all these random token string pieces chunked up totally at random so that's what we have right now uh now one more thing I wanted to point out by the way is in case you do not have Cuda available because you don't have a GPU you can still follow along with uh with what we're doing here uh to some extent uh and probably not to the very end because by the end we're going to be using multiple gpus and actually doing a serious training run uh but for now you can actually follow along decently okay uh so one thing that I like to do in pytorch is I like to autod detect the device that is available to you so in particular you could do that like this so here we are trying to detect a device to run on that has the highest compute capability you can think about it that way so by default we start with CPU which of course is available everywhere because every single computer will have a CPU but then we can try to detect do you have a GPU you so use a Cuda and then if you don't have a Cuda uh do you at least have MPS MPS is the back end for Apple silicon so if you have a Macbook that is fairly new you probably have apple silicon on the inside and then that has a GPU that is actually fairly capable uh depending on which MacBook you have and so you can use MPS which will be potentially faster than CPU and so we can print the device here now once we have the device we can actually use it in place of Puda so we just swap it in and notice that here when we call model on X if this x here is on CPU instead of GPU then it will work fine because here in the forward which is where P to will come when we create a pose we were careful to use the device of idx to create this tensor as well and so there won't be any mismatch where one tensor is on CPU one is on GPU and uh that you can't combine those but here we are um carefully initializing on the correct device as indicated by the input to this model so this will autod detect device for me this will be of course GPU so using device Cuda uh but uh you can also run with um as I mentioned another device and it's not going to be too much slower so if I override device here oops if I override device equals CPU then we'll still print Cuda of course but now we're actually using CPU one 2 3 4 5 6 okay about 6 seconds and actually we're not using torch compile and stuff like that which will speed up everything a lot faster as well but you can follow even on a CPU I think to a decent extent um so that's note on that okay so I do want to loop around eventually into what it means to have different devices in pytorch and what it is exactly that pytorch does in the background for you when you do something like module. 2 device or where you take a torch tensor and do A2 device and what exactly happens and how that works but for now I'd like to get to training and I'd like

00:45:50 lets train: data batches (B,T) logits (B,T,C)

to start training the model and for now let's just say the device makes code go fast um and let's go into how we can actually train the model so to train the model we're going to need some data set and for me the best debugging simplest data set that I like to use is the tiny Shakespeare data set um and it's available at this URL so you can W get it or you can just search tiny Shakespeare data set and so um I have in my file system as just LS input.txt so I already downloaded it and here I'm reading the data set getting the first 1,000 characters and printing the first 100 now remember that gpt2 has uh roughly a compression ratio the tokenizer has a compression ratio of rly 3 to1 so th000 characters is roughly 300 tokens here uh that will come out of this in the slice that we're currently getting so this is the first few uh characters and uh if you want to get a few more statistics on this we can do work count on input.txt so we can see that this is uh 40,000 lines about 200,000 words in this data set and about 1 million bytes in this file and knowing that this file is only asky characters there's no crazy unic code here as far as I know and so every asky character is encoded with one bite and so this is uh the same number roughly a million characters inside this data set so that's the data set size uh by default very small and minimal data set for debugging to get us off the ground in order to tokenize this data set we're going to get Tik token encoding for gbt2 encode the data uh the first um 1,000 characters and then I'm only going to print the first 24 tokens so these are the tokens as a list of integers and if you can read gpt2 tokens you will see that 198 here you'll recognize that as the slashing character so that is a new line and then here for example we have two new lines so that's 198 twice here uh so this is just a tokenization of the first 24 tokens so what we want to do now is we want to actually process these token sequences and feed them into a Transformer and in particular we want them we want to rearrange these tokens into this idx variable that we're going to be feeding into the Transformer so we don't want a single very long onedimensional sequence we want an entire batch where each sequence is up to uh is basically T tokens and T cannot be larger than the maximum sequence length and then we have these t uh tlong uh sequences of tokens and we have B independent examples of sequences so how can we create a b BYT tensor that we can feed into the forward out of these onedimensional sequences so here's my favorite way to to achieve this uh so if we take torch and then we create a tensor object out of this list of integers and just the first 24 tokens my favorite way to do this is basically you do a do view of um of uh for example 4x6 which multiply to 24 and so it's just a two-dimensional rearrangement of these tokens and you'll is that when you view this onedimensional sequence as two-dimensional 4x6 here the first six uh tokens uh up to here end up being the first row the next six tokens here end up being the second row and so on and so basically it's just going to stack up this the um every six tokens in this case as independent rows and it creates a batch of tokens in this case and so for example if we are token 25 in the Transformer when we feed this in and this becomes the idx this token is going to see these three tokens and it's going to try to predict that 198 comes next so in this way we are able to create this two-dimensional batch that's that's quite nice now in terms of the label that we're going to need for the Target to calculate the loss function how do we get that well we could write some code inside the forward pass because we know that the next uh token in a sequence which is the label is just to the right of us but you'll notice that actually we for this token at the very end 13 we don't actually have the next correct token because we didn't load it so uh we actually didn't get enough information here so I'll show you my favorite way of basically getting these batches and I like to personally have not just the input to the Transformer which I like to call X but I also like to create the labels uh tensor which is of the exact same size as X but contains the targets at every single position and so here's the way that I like to do that I like to make sure that I fetch plus one uh token because we need the ground Truth for the very last token uh for 13 and then when we're creating the input we take everything up to the last token not including and view it as 4x6 and when we're creating targets we do the buffer but starting at index one not index zero so we're skipping the first element and we view it in the exact same size and then when I print this here's what happens where we see that basically as an example for this token 25 its Target was 198 and that's now just stored at the exact same position in the Target tensor which is 198 and also this last token 13 now has its label which is 198 and that's just because we loaded this plus one here so basically this is the way I like to do it you take long sequences you uh view them in two-dimensional terms so that you get batch of time and then we make sure to load one additional token so we basically load a buffer of tokens of B * t+ one and then we sort of offset things and view them and then we have two tensors one of them is the input to the Transformer and the other exactly is the labels and so let's now reorganize this code and um create a very simple data loader object that tries to basically load these tokens and um feed them to the Transformer and calculate the loss okay so I reshuffled the code here uh accordingly so as you can see here I'm temporarily overwriting U to run a CPU and importing TI token and all of this should look familiar we're loading a th000 characters I'm setting BT to just be 4 and 32 right now just because we're debugging we just want to have a single batch that's very small and all of this should now look familiar and follows what we did on the right and then here we get the we create the model and get the lits and so so here as you see I already ran this only runs in a few seconds but because we have a batch of uh 4X 32 our lits are now of size 4X 32x 50257 so those are the lit for what comes next at every position and now we have the labels which are stored in y so now is the time to calculate the loss and then do the backward pass and then the optimization so let's first calculate the loss

okay so to calculate the loss we're

00:52:53 cross entropy loss

going to adjust the forward function of this NN module in the model and in particular we're not just going to be returning logits but also we're going to return the loss uh and we're going to not just pass in the input in thees but also the targets uh in y and now we will print not Lo just. shape anymore we're actually going to print the loss function and then c. exit of zero so that we skip some of the sampling logic so now let's swing up to the forward function which gets called there because now we also have these optional targets and when we get the targets we can also calculate uh the loss and remember that we want to basically return uh log just loss and loss by default is none but um let's put this here if uh targets is not none then we want to calculate loss and co-pilot is already getting excited here and calculating the what looks to be correct loss it is using the cross entropy loss as is documented here uh so this is a function in pytorch under the functional now what is actually happening here because it looks a little bit scary uh basically uh the F that cross entropy does not like multi-dimensional inputs it can't take a b BYT by vocap size so what's happening here is that we are flattening out this three-dimensional tensor into just two Dimensions the First Dimension is going to be calculated automatically and it's going to be B * T and then the last Dimension is vocap size so basically this is uh flattening out this three-dimensional tensor of logits to just be two- dimensional B * T all individual examples and vocap size on uh in terms of the length of each row and then it's also flattening out the targets which are also two-dimensional at this stage but we're going to just flatten them out so they're just a single tensor of B * T and this can then pass into cross entropy to calculate a loss which we return so this should basically at this point run because this is not too complicated so let's run it and let's see if we should be printing the loss and here we see that we printed 11 uh roughly and so um and notice that this is the tensor of a single element which is this number 11 now we also want to be able to calculate a reasonable uh kind of starting point for a random rationalized Network so we covered this in previous videos but our vocabulary size is 50257 at initialization of the network you would hope that um every vocab element is getting roughly a uniform probability uh so that we're not favoring at initialization any token way too much we're not confidently wrong at initialization so what we're hoping is that the probability of any arbitrary token is roughly 1 over 50,2 57 and now we can sanity check the loss because remember that the cross entropy loss is just basically the negative um log likelihood so if we now take this probability and we take it through the natural logarithm and then we do the negative that is the loss we expect at initialization and we covered this in previous videos so I would expect something around 10.82 and we're seeing something around 11 so it's not way off this is roughly the probability I expect at initialization so that tells me that the at initialization or probability distribtion is roughly diffused it's a good starting point and we can now uh perform the optimization and tell the network which elements you know should follow correctly in what order so at this point we can do a l step backward calculate the gradients and do an optimization so let's get to that okay

00:56:42 optimization loop: overfit a single batch

so let's do the optimization now um so here we have the loss is this is how we get the loss but now basically we want a load for Loop here so 4 I in range let's do 50 steps or something like that uh let's create an Optimizer object in pytorch um and so here we are using the atom um Optimizer which is an alternative to the stochastic radian descent Optimizer SGD that we were using so SGD is a lot simpler atom is a bit more involved and I actually specifically like the atom W variation because in my opinion it kind of just like fixes a bug um so adom w is a bug fix of atom is what I would say when we go to the documentation for atom W oh my gosh we see um that it takes a bunch of hyper parameters and it's a little bit more complicated than the SGD we were looking at before uh because in addition to basically updating the parameters with the gradient uh scaled by the Learning rate it keeps these buffers around and it keeps two buffers the m and the V which it calls the first and the second moment so something that looks a bit like momentum and something that looks a bit like RMS prop if you're familiar with it but you don't have to be it's just kind of a normalization that happens on each gradient element individually and speeds up the optimization especially for language models but I'm not going to go into the detail right here we're going to treat it as a bit of a black box and it just optimizes um the objective faster than SGD which is what we've seen in the previous lectures so let's use it as a black box in our case uh create the optimizer object and then go through the optimization the first thing to always make sure the co-pilot did not forget to zero the gradients so um always remember that you have to start with a zero gradient then when you get your loss and you do a DOT backward dot backward adds to gradients so it deposits gradients it it always does a plus equals on whatever the gradients are which is why you must set them to zero so this accumulates the gradient from this loss and then we call the step function on the optimizer to um update the parameters and to um decrease the loss and then we print a step and the loss do item is used here because loss is a tensor with a single element do item will actually uh convert that to a single float and this float will live not will will live on the CPU so this gets to some of the internals again of the devices but loss is a is a tensor with a single element and it lifts on GPU for me because I'm using gpus when you call item P torch behind the scenes will take that one-dimensional tensor ship it back to the CPU uh memory and convert it into a float that we can just print so this is the optimization and this should probably just work let's see what happens actually sorry let me instead of using CPU override let me delete that so this is a bit faster for me and it runs on Cuda oh expected all tensors to be on the same device but found at least two devices Cuda zero and CPU so Cuda zero is the zeroth GPU because I actually have eight gpus on this box uh so the zeroth GPU in my box and CPU and model we have moved to device but when I was writing this code I actually introduced a bug because buff we never moved to device and you have to be careful because you can't just do buff dot two of device um it's not stateful it doesn't convert it to be a device it instead uh returns pointer to a new memory which is on the device so you see how we can just do model that two a device that does not apply to tensors you have to do buff equals um b.2 device and then this should work okay so what do we expect to see we expect to see a reasonable loss in the beginning and then we continue to optimize just the single batch and so we want to see that we can overfit this single batch we can we can crush this little batch and we can perfectly predict the indices on just this little batch and indeed that is roughly what we're seeing here so um we started off at roughly 10.82 11 in this case and then as we continue optimizing on this single batch without loading new examples we are making sure that we can overfit a single batch and we are getting to very very low loss so the Transformer is memorizing this single individual batch and one more thing I didn't mention is uh the learning rate here is 3 E4 which is a pretty good default for most uh optimizations that you want to run at a very early debugging stage so this is our simple inter Loop and uh we are overfitting a single batch and this looks good so now what uh what comes next is we don't just want to overfit a single batch we actually want to do an optimization so we actually need to iterate these XY batches and create a little data loader uh that makes sure that we're always getting a fresh batch and that we're actually optimizing a reasonable objective so let's do that next okay so this is what I came up with

01:02:00 data loader lite

and I wrote a little data loader light um so what this data loader does is we're importing the token up here we're reading the entire text file from this single input.txt tokenizing it and then we're just printing the number of tokens in total and the number of batches in a single Epoch of iterating over this data set so how many unique batches do we output before we loop back around the beginning of the document and start reading it again so we start off at position zero and then we simply walk the document in batches of B * T so we take chunks of B * T and then always Advance by B * T and um it's important to note that we're always advancing our position by exactly B * T but when we're fetching the tokens we're actually fetching from current position to B * t + 1 and we need that plus one because remember uh we need the target token um for the last token in the current batch and so that way we can do um the XY exactly as we did it before and if we are to um run out of data we'll just loop back around to zero so this is one way to write a very very simple data loader um that simply just goes through the file in chunks and is good enough for us uh for current purposes and we're going to complexify it later and now we'd like to come back around here and we'd like to actually use our data loader so the import Tik token has moved up and actually all of this is now useless so instead we just want a train loader for the training data and we want to use the same hyper parameters for four so B size was four and time was 32 and then here we need to get the XY for the current batch so let's see if copal gets it because this is simple enough uh so we call the next batch and then we um make sure that we have to move our tensors from CPU to the device so here when I converted the tokens notice that I didn't actually move these tokens to the GPU I left them on CPU which is the default um and that's just because I'm trying not to waste too much memory on the GPU in this case this is a tiny data set and it would fit uh but it's fine to just uh ship it to GPU right now for for our purposes right now so we get the next batch we keep the data loader simple CPU class and then here we actually ship it to the GPU and do all the computation and uh let's see if this runs so python train gbt2 pi and what do we expect to see before this actually happens what we expect to see is now we're actually getting the next batch so we expect to not overfit a single batch and so I expect our loss to come down but not too much and that's because I still expect it to come down because in the 50257 tokens many of those tokens never occur in our data set so there are some very easy gains to be made here in the optimization by for example taking the biases of all the loits that never occur and driving them to negative infinity and that would basically just it's just that all of these crazy unic codes or different languages those tokens never occur so their probability should be very low and so the gains that we should be seeing are along the lines of basically deleting the usage of tokens that never occur that's probably most of the loss gain that we're going to see at this scale right now uh but we shouldn't come to a zero uh because um we are only doing 50 iterations and I don't think that's enough to do an eoch right now so let's see what we got we um we have 338,000 tokens which makes sense with our 3:1 compression ratio because there are 1 million uh characters so one Epoch with the current setting of B and T will take 2, 600 batches and we're only doing 50 batches of optimization in here so we start off in a familiar territory as expected and then we seem to come down to about 6.6 so basically things seem to be working okay right now with respect to our expectations so that's good okay next I want to actually

01:06:14 parameter sharing wte and lm_head

fix a bug that we have in our code um it's not a major bug but it is a bug with respect to how gpt2 training uh should happen um so the buck is the following we were not being careful enough when we were loading the weights from hugging face and we actually missed a little detail so if we come here notice that um the shape of these two tensors is the same so this one here is the token embedding at the bottom of the Transformer right so and this one here is the language modeling head at the top of the Transformer and both of these are basically two-dimensional tensors and they shape is identical so here the first one is the output embedding the token embedding and the second one is this linear layer at the very top the classifier layer both of them are of shape 50257 X 768 um this one here is giving us our token embeddings at the bottom and this one here is taking the 768 channels of the Transformer and trying to upscale that to 50, 257 to get the Lis for the next token so they're both the same shape but more than that actually if you look at um comparing their elements um in pytorch this is an element wise equality so then we use do all and we see that every single element is identical and more than that we see that if we actually look at the data pointer uh this is what this is a way in pytorch to get the actual pointer to the uh data and the storage we see that actually the pointer is identical so not only are these two separate tensors that happen to have the same shape and elements they're actually pointing to the identical tensor so what's happening here is that this is a common weight tying scheme uh that actually comes from the original um from the original attention is all you need paper and actually even the reference before it so if we come here um eddings and softmax in the attention is all you need paper they mentioned that in our model we shared the same weight Matrix between the two embedding layers and the pre softmax linear transformation similar to 30 um so this is an awkward way to phrase that these two are shared and they're tied and they're the same Matrix and the 30 reference is this paper um so this came out in 2017 and you can read the full paper but basically it argues for this weight tying scheme and I think intuitively the idea for why you might want to do this comes from from this paragraph here and basically you you can observe that um you actually want these two matrices to behave similar in the following sense if two tokens are very similar semantically like maybe one of them is all lowercase and the other one is all uppercase or it's the same token in a different language or something like that if you have similarity between two tokens presumably you would expect that they are uh nearby in the token embedding space but in the exact same way you'd expect that if you have two tokens that are similar semantically you'd expect them to get the same probabilities at the output of a transformer because they are semantically similar and so both positions in the Transformer at the very bottom and at the top have this property that similar tokens should have similar embeddings or similar weights and so this is what motivates their exploration here and they they kind of you know I don't want to go through the entire paper and and uh you can go through it but this is what they observe they also observe that if you look at the output embeddings they also behave like word embeddings um if you um if you just kind of try to use those weights as word embeddings um so they kind of observe this similarity they try to tie them and they observe that they can get much better performance in that way and so this was adopted and the attention is all need paper and then it was used again in gpt2 as well so I couldn't find it in the Transformers implementation I'm not sure where they tie those embeddings but I can find it in the original gpt2 code U introduced by open aai so this is um openai gpt2 Source model and here where they are forwarding this model and this is in tensorflow but uh that's okay we see that they get the wte token embeddings and then here is the incoder of the token embeddings and the position and then here at the bottom they Ed the WT again to do the lits so when they get the loits it's a math Mo of uh this output from the Transformer and the wte tensor is reused um and so the wte tensor basically is used twice on the bottom of the Transformer and on the top of the Transformer and in the backward pass we'll get gradients contributions from both branches right and these gradients will add up um on the wte tensor um so we'll get a contribution from the classifier list and then at the very end of the Transformer we'll get a contribution at the at the bottom of it float floating again into the wte uh tensor so we want to we are currently not sharing WT and our code but we want to do that um so weight sharing scheme um and one way to do this let's see if goil gets it oh it does okay uh so this is one way to do it uh basically relatively straightforward what we're doing here is we're taking the wte do weight and we're simply uh redirecting it to point to the LM head so um this basically copies the data pointer right it copies the reference and now the wte weight becomes orphaned uh the old value of it and uh pytorch will clean it up python will clean it up and so we are only left with a single tensor and it's going to be used twice in the forward pass and uh this is to my knowledge all that's required so we should be able to use this and this should probably train uh we're just going to basically be using this exact same sensor twice and um we weren't being careful with tracking the likelihoods but uh according to the paper and according to the results you'd actually expect slightly better results doing this and in addition to that one other reason that this is very very nice for us is that this is a ton of parameters right uh what is the size here it's 768 * 50257 so This Is 40 million parameters and this is a 124 million parameter model so 40 divide 124 so this is like 30% of the parameters are being saved using this weight time scheme and so this might be one of the reasons that this is working slightly better if you're not training the model long enough because of the weight tying uh you don't have to train as many parameters and so you become more efficient um in terms of the training process uh because you have fewer parameters and you're putting in this inductive bias that these two embeddings should share similarities between tokens so this is the way time scheme and we've saved a ton of parameters and we expect our model to work slightly better because of the scheme okay next I would

01:13:47 model initialization: std 0.02, residual init

like us to be a bit more careful with the initialization and to try to follow the way gpt2 initialized their model now unfortunately the gpt2 paper and the gpt3 paper are not very explicit about initialization so we kind of have to read between the lines uh and instead of going to the paper which is quite vague um there's a bit of information in the code that open I released so when we go to the model.py we see that when they initialize their weights they are using the standard deviation of 0.02 and that's how they they so this is a normal distribution for the weights and the standard deviation is 0.02 for the bias they initialize that with zero and then when we scroll down here why is this not scrolling um the token embeddings are initialized at 0.02 and position embeddings at 0.01 for some reason so those are the initializations and we'd like to mirror that in gpt2 uh in our module here so here's a snippet of code that I sort of came up with very quickly so what's happening here is at the end of our initializer for the GPT module we're calling the apply function of NN module and that iterates all the sub modules of this module and uh applies in it weights function on them and so what's happening here is that we're in we're iterating all the modules here and if they are an nn. linear module then we're going to make sure to initialize the weight using a normal with the standard deviation of 0.02 if there's a bias in this layer we will make sure to initialize that to zero note that zero initialization for the bias is not actually the pyto default um by default the bias here is initialized with a uniform so uh that's interesting so we make sure to use zero and for the embedding we're just going to use 0.02 and um keep it the same um so we're not going to change it to 0.01 for positional because it's about the same and then if you look through our model the only other layer that requires initialization and that has parameters is the layer norm and the fighter defer initialization sets the scale in the layer Norm to be one and the offset in the layer Norm to be zero so that's exactly what we want and so we're just going to uh keep it that way and so this is the default initialization if we are following the um where is it the uh gpt2 uh source code that they released I would like to point out by the way that um typically the standard deviation here on this initialization if you follow the Javier initialization would be one of over the square root of the number of features that are incoming into this layer but if you'll notice actually 0.02 is basically consistent with that because the the model sizes inside these Transformers for gpt2 are roughly 768 1600 Etc so 1 over the square root of for example 768 gives us 0.03 if we plug in 600

1,600 we get 0.02 if we plug in three times that 0.014 Etc so basically 0.02 is roughly in the vicinity of reasonable values for the for um for these initializations anyway so so it's not uh completely crazy to be hard coding 0.02 here uh but you'd like typically uh some something that grows with the model size instead but we will keep this because that is the gpt2 initialization per their source code but we are not fully done yet on initialization because there's one more caveat here so here a mod initialization which accounts for the accumulation on the residual path with model depth is used we scale the weight of residual layers of initialization by factor of one over squ of n where n is the number of residual layers so this is what gbt2 paper says so we have not implemented that yet and uh we can do so now now I'd like to actually kind of like motivate a little bit what they mean here I think um so here's roughly what they mean if you start out with zeros in your residual stream remember that each residual stream is a is of this form where we continue adding to it X is X plus something some kind of contribution so every single block of the residual uh Network contributes some uh amount and it gets added and so what ends up happening is that the variance of the activations in the residual stream grows so here's a small example if we start at zero and then we for 100 times uh we have sort of this residual stream of of 768 uh zeros and then 100 times we add um random which is a normal distribution zero mean one standard deviation if we add to it then by the end the residual stream has grown to have standard deviation of 10 and that's just because um we're always adding um these numbers and so this scaling factor that they use here exactly compensates for that growth so if we take n and we basically um scale down every one of these contributions into the residual stream by one over theare Ro of n so 1 over theun of n is n to the 0.5 right because n the5 is the square root and then one over the square root is n.5 if we scale it in this way then we see that we actually get um one so this is a way to control the growth of of activations inside the residual stream in the forward pass and so we'd like to initialize in the same way where these weights that are at the end of each block so this C uh layer uh the gbt paper proposes to scale down those weights by one over the square root of the number of residual layers so one crude way to implement this is the following I don't know if this is uh pyro sanctioned but it works for me is we'll do in the initialization see that s that do special nanog GPT uh scale in it is one so we're setting um kind of like a flag for this module there must be a better way in py torch right but I don't know okay so we're basically attaching this flag and trying to make sure that it doesn't conflict with anything previously and then when we come down here this STD should be 0.02 by default but then if haat um module of this thing then STD * equals um copal is not guessing correctly uh so we want one over the square root of the number of layers so um the number of residual layers here is twice times Salt out config layers and then this times .5 so we want to scale down that standard deviation and this should be um correct and Implement that I should clarify by the way that the two times number of layers comes from the fact that every single one of our layers in the Transformer actually has two blocks that add to the ridal pathway right we have the attention and then the MLP so that's where the two times comes from and the other thing to mention is that uh what's slightly awkward but we're not going to fix it is that um because we are weight sharing the wte and the LM head in this iteration of our old subm modules we're going to actually come around to that tensor twice so we're going to first initialize it as an embedding with 0.02 and then we're going to come back around it again in a linear and initialize it again using 0.02 and it's going to be 0.02 because the LM head is of course not not scaled so it's not going to come here it's just it's going to be basically initialized twice using the identical same initialization but that's okay and then scrolling over here I added uh some code here so that we have reproducibility um to set the seeds and now we should be able to python train gpt2 pi and let this running and as far as I know this is the gpt2 initialization uh in the way we've implemented it right now so this looks uh reasonable to me okay so at

01:22:18 SECTION 2: Lets make it fast. GPUs, mixed precision, 1000ms

this point we have the gpt2 model we have some confidence that it's correctly implemented we've initialized it properly and we have a data loader that's iterating through data batches and we can train so now comes the fun part I'd like us to speed up the training by a lot so we're getting our money's worth with respect to the hardware that we are uh using here and uh we're going to speed up the training by quite a bit uh now you always want to start with what Hardware do you have what does it offer and are you fully utilizing it so in my case if we go to Nvidia SMI we can see that I have eight gpus and each one of those gpus is an a100 sxm 80 gb so this is the GPU that I have available to me in this box now when I look when I use um to spin up these kinds of Boxes by the way my favorite place to go to is Lambda Labs um they do sponsor my development and that of my projects uh but I this is my favorite place to go and this is where you can spin up one of these machines and you pay per hour and it's very very simple so I like to spin them up and then connect vsod to it and that's how I develop now when we look at the A1 100s that are available here a100 80 GB sxm is the um GPU that I have here and we have a bunch of numbers here for um how many calculations you can expect out of this GPU so when I come over here and I break in right after here so python trity so I'm breaking in right after we calculate the loit and laws and the interesting thing I'd like you to note is when I do lit. dtype this prints a torch. FL 32 so by default iny torch when you create tensors um and this is the case for all the activations and for the parameters of the network and so on by default everything is in float 32 that means that every single number activation or weight and so on is using a float representation that has 32 bits and uh that's actually quite a bit of memory and it turns out empirically that for deep learning as a computational workload this is way too much and deep learning and the training of these networks can tolerate significantly lower precisions um not all computational workflows can tolerate small Precision so for example um if we go back to to the data sheet you'll see that actually these gpus support up to fp64 and this is quite useful I understand for a lot of um scientific Computing applications and there really need this uh but we don't need that much Precision for deep learning training So currently we are here fp32 and with this code as it is right now we expect to get at at most 19.5 Tera flops of performance that means we're doing 19.5 trillion operations floating Point operations so this is floating Point multiply add most um most likely and so these are the floating Point operations uh now notice that if we are willing to go down in Precision so tf32 is a lower Precision format we're going to see in a second you can actually get an 8X Improvement here and if you're willing to go down to float 16 or B float 16 you can actually get time 16x performance all the way to 312 Tera flops you see here that Nvidia likes to site numbers that have an asterisk here this asterisk uh says with sparsity uh but we are not going to be using sparsity in R code and I don't know that this is very widely used in the industry right now so most people look at this number here uh without sparcity and you'll notice that we could have got even more here but this is int 8 and int 8 is used for inference not for training uh because int 8 has a um it basically has um uniform spacing um and uh we actually require a float so that we get a better match to the uh normal distributions that occur during training of neural networks where both activations and weights are distributed as a normal distribution and so uh floating points are really important to to match that uh representation so we're not typically using int 8 uh for training but we are using it for inference and if we bring down the Precision we can get a lot more Terra flops out of the tensor course available in the gpus we'll talk about that in a second but in addition to that if all of these numbers have fewer bits of representation it's going to be much easier to move them around and that's where we start to get into the memory bandwidth and the memory of the model so not only do we have a finite capacity of the number of bits that our GPU can store but in addition to that there's a speed with which you can access this memory um and you have a certain memory bandwidth it's a very precious resource and in fact many of the deep learning uh work workloads for training are memory bound and what that means is actually that the tensor cores that do all these extremely fast multiplications most of the time they're waiting around they're idle um because we can't feed them with data fast enough we can't load the data fast enough from memory so typical utilizations of your Hardware if you're getting 60% uh utilization you're actually doing extremely well um so half of the time in a well-tuned application your tensor cores are not doing multiplies because the data is not available so the memory bandwidth here is extremely important as well and if we come down in the Precision for all the floats all the numbers weights and activations suddenly require less memory so we can store more and we can access it faster so everything speeds up and it's amazing and now let's reap the benefits of it um and let's first look at the tensor float 32

01:28:14 Tensor Cores, timing the code, TF32 precision, 333ms

format okay so first of all what are tensor cores well tensor course tensor core is just an instruction in the a100 architecture right so so what it does is it does basically a little 4x4 Matrix multiply so uh this is just matrix multiplication here of 4x4 matrices and there are multiple configurations as to what Precision any of these matrices are it in what Precision the internal accumulate happens and then what is the output Precision input precisions Etc so there's a few switches but it's basically a 4x4 multiply and then anytime we have any operations that require Magic multiplication uh they get broken up into these into this instruction of little 4x4 multiply and so everything gets broken up into this instruction because it's the fastest way to multiply matrices and it turns out that most of the computational work that we're doing up above uh all of it really is matrix multiplication most of the work computationally happens in the linear layers um linear linear Etc there's a few things sandwiched in between so there's some additions in residuals there's some G nonlinearities there's some layer Norms Etc but if you just time them you'll see that these are nothing like basically the in Transformer is just a bunch of Matrix multiplications really um and especially at this small scale 124 million parameter model actually the biggest matrix multiplication by far is the classifier layer at the top that is a massive Matrix multiply of going from 768 to 50257 and that Matrix multiply dominates anything else that happens in that Network roughly speaking so it's Matrix multiplies that become a lot faster which are hidden inside our linear layers and they're accelerated through tensor course now the best reference I would say for tensor course is basically just go to the um a 100 architecture white paper and then it's pretty detailed and but I think people it's like relatively readable mostly if you half understand what's happening um so figure 9 tensor float 32 so this is the explanation basically for tf32 and what happens here and you see that there's many configuration options here available so the input operands and what precisions are they in the accumulator and um what um basically the um the internal representation within the instruction when you do the accumulate of this matrix multiplication so the intermediate plus equals um of the intermediate little vector multiplies here that all happens in fp32 and then uh this is an aex improvement as I mentioned to the Ops that we get so tf32 specifically we're looking at this row here and the way this works is um normally fp32 has 32 bits tf32 is the exact same bits we have one sign bit we have eight exponent bits except the mantisa bits get cropped in the float and so basically um we end up with just 19 bits instead of 32 bits because the last 133 bits get truncated they get dropped um and all this is internal to the instruction so none of it is visible to anything in our pytorch uh none of our pytorch code will change all of the numbers will look identical it's just that when you call the tensor core um instruction internally in the hardware it will crop out these 13 bits and that allows it to uh calculate this little Matrix multiply significantly faster 8X faster now of course this speed up comes at a cost and the cost is that we are reducing the Precision our accumulate is still an fp32 our output is fp32 our inputs are fp32 but internally things get truncated in the operand to perform the operation faster and so our results are starting to be a bit more approximate but empirically when you actually train with this you basically can't tell the difference so the reason I like tf32 is because if you can tolerate a little bit of a Precision fudge um then this is free like none of your codes sees this it's fully internal to the operation and the operation to you just go 8X faster and it's a bit more approximate and so it's a pretty sweet spot I would say in optimization and uh let's see what that looks like first so I've set up our Cod to just time the uh iterations so import time I changed the hyper parameters so that we have something a bit more that reflects uh kind of workload that we want to run uh because we want to do a fairly large run at the end of this so let's use batch size 16 and let's now use the actual gpt2 um maximum sequence length of 10,24 tokens uh so this is the configuration and then for 50 iterations I'm just doing something very lazy here I'm doing time. time to get the current time and then this is the optimization Loop and now I want to time how long this takes now one issue with working with gpus is that as your CPU um when your CPU runs it's just scheduling work on GPU it's ordering some work right and so it send a request and then it continues running and so we can actually it can happen sometimes that we sort of um speed through this and we queue up a lot of kernels to run on the GPU and then the CPU sort of like gets here and takes time at time but actually the GPU is still running because it takes it time to actually work through the work that was scheduled to run and so you're just building up a queue for the GPU and so actually if you need to you want to wait toat data synchronize and this will wait for the GPU to finish all the work that was scheduled to run up above here and then we can actually take the time so basically we're waiting for the GPU to stop this iteration take time and then we're going to just print it so so here I'm going to run the training Loop and here on the right I'm watching Nvidia SMI so we start off at zero um we're not using the GPU and then by default P will use gpu0 so we see that it gets filled up and we're using 35 GB out of 80 gabt available and then here on the left we see that because we've cranked up the batch size now it's only 20 batches to do a single Epoch on our tiny Shakespeare and we see that we're seeing roughly a th000 milliseconds per iteration here right so the first iteration sometimes is slower and that's because pytorch might be doing a lot of initializations here on the very first iteration and so it's probably initializing all these uh tensors and buffers to hold all the gradients and I'm not 100% sure all the work that happens here but uh this could be a slower iteration when you're timing your logic you always want to be careful with that but basically we're seeing a th000 milliseconds per iteration um and so this will run for roughly 50 seconds as we have it right now so that's our Baseline in flo 32 one more thing I wanted to mention is that if this doesn't fit into your GPU and you're getting out of memory errors then start decreasing your batch size until things fit so instead of 16 try eight or four or whatever you need to fit um the batch into your GPU and if you have a bigger GPU you can actually potentially get away with 32 and so on uh by default you want to basically max out has Max Max out the batch size that fits on your GPU and you want to keep it nice numbers so use numbers that have lots of powers of two in them so 16 is a good number 8 24 32 48 These are nice numbers but don't use something like 17 uh because that will run very inefficiently on a GPU uh and we're going to see that a bit later as well so for now let's just stick with 16124 and uh the one thing that I added also here and I ran it again is I'm calculating a tokens per second throughput during training because we might end up changing the backat size around over time but tokens per second is the objective measure that we actually really care about how many tokens of data are we training on and what is the throughput of tokens that we're getting in our optimization so right now we're processing and training on 163,000 tokens per second roughly and that's a bit more objective metric okay so let's now enable tf32 now luckily pytorch makes this fairly easy for us and uh to enable tf32 you just need to do a single line and is this and when we go to the py documentation here for this function basically this tells pych what kind of kernels to run and by default I believe it is highest highest Precision for mat M and that means that everything happens in float 32 just like it did before but if we set it to high as we do right now Matrix multiplications will not use tensor flow 32 when it's available my GPU is a100 so it's an ampere series and therefore tf32 is available if you have an older GPU this might not be available for you but for my GPU it's available and so what I expect P to do is that every single place where we see an nn. linear inside there there's a matrix multiplication and I expect that matrix multiplication now to be um running on tensor course utilizing the TF 32% so this is the single line of change that is I believe necessary and let's rerun this now we saw that um in terms of the throughput that is promised to us we're supposed to be getting 8X roughly so let's see what happens and that 8X came from here right um 8X and it also came from looking at it um here 156 T flops instead of of 19.5 okay so what actually happened uh so we're seeing that our throughput roughly 3x not aex so we are going we're from 1,000 milliseconds we're going down to 300 milliseconds and our throughput is now about 50,000 tokens per second so we have a roughly 3x instead of 8X so what happened and basically What's Happening Here is again a lot of these workloads are memory bound and so even though the tf32 offers in principle a lot faster throughput all of these numbers everywhere are still float 32s and it's float 32 numbers that are being shipped all over the place through the memory system and is just costing us way too much time to shuttle around all this data and so even though we've made the multiply itself much faster uh we are memory bound and we're not actually seeing the full benefit uh that would come from uh this napkin math here uh that said we are getting one a 3X faster throughput and this is free um single line of code in P torch all your variables are still float 32 everywhere it just runs faster and it's slightly more approximate but we're not going to notice it basically uh so that's tf32 okay so let's now continue so we've

01:39:38 float16, gradient scalers, bfloat16, 300ms

exercised this row and um we saw that we can crop out some of the Precision inside the operation itself but we saw that we're still memory bound we're still moving around all these floats right otherwise and we're paying that cost because of this so let's now decrease the amount of stuff that we're going to be moving around and we're going to do that by dropping down to B float 16 so we're only going to be maintaining 16 bits per float and we're going to use the B flat 16 and I'll explain in a bit uh fp16 difference and uh we're going to be in this row so when we go back to the documentation here for the a 100 um we see here the precisions that are are available and this is the original fp32 the tf32 crops out the Precision and then here in bf16 you see that it is very similar to tf32 but it's even more aggressive in cropping off of the Precision the mantisa of this float so the important thing with B float 16 is that the exponent bits and the sign bit of course remain unchanged so if you're familiar with your float numbers and I think this should should probably be an entire video by itself the exponent sets the range that you can represent of your numbers and the Precision is how much Precision you have for your numbers and so the range of numbers is identical but we can we have fewer possibilities within that range because we are truncating the Mena so we have less Precision in that range what that means is that things are actually fairly nice because we have the original range of numbers that are representable in float but we just have less Precision for it and the difference with fp16 is that they actually touch and change the range so fp16 cannot represent the full range of fp32 it has a reduced range and that's where you start to actually run into issues because now you need uh these gradient scalers and things like that and I'm not going to go into the detail of that in this video because that's a whole video by itself but fb16 actually historically came first that was available in the Volta series before Amper and so fp16 came first and everyone started to train in fp16 but everyone had to use all these gradient scaling operations which are kind of annoying and it's an additional source of state and complexity and the reason for that was because the exponent range was reduced in fp16 so that's the i e fp16 spec and then they came out with bf16 and the Ampere and they made it much simpler because we're just truncating manessa we have the exact same range and we do not need gradient scalers so everything is much much simpler now when we do use bf16 though we are impacting the numbers that we might be seeing in our pytorch code these this change is not just local to the operation itself so let's see how that works um there's some documentation here that so I think this is probably the best best page to explain how to use mixed Precision in pytorch um because there are many other tutorials and so on even within pitor documentation that are a lot more confusing and so I recommend specifically this one because there's five other copies that I would not recommend and then when we come here ignore everything about everything ignore everything about gradient scalers and only look at torch. AutoCast and basically also this comes to a single line of code at the end so this is the context manager that we want and we want to use that in our Network when you click into the torch. AutoCast autocasting it has a few more uh a bit more guideline for you so it's telling you do not call B flat 16 on any of your tensors just use AutoCast and only surround the uh forward pass of the model and the loss calculation and that's the only two things that you should be surrounding leave the backward and the optimizer step alone so that's the guidance that comes from the P team so we're going to follow that guidance and for us because the L calculation is inside of the model forward pass for us we are going to be doing this and then we don't want to be using torch Flo 16 because if we do that we need to start using gradient scalers as well so we are going to be using B float 16 this is only possible to do an ampere uh but this means that the changes are extremely minimal like basically just this one line of code um let me first break in to here before we actually run this so right after logits I'd like to show you that different from the tf32 that we saw this is actually going to impact our tensors so this Lis tensor if we now look at this and we look at the dtype we suddenly see that this is now B float 16 uh it's not float 32 anymore so our activations have been changed the activations tensor is now B FL 16 but not everything has changed so model. Transformer wte uh this is the weight uh token embedding table it has a weight inside it and the dtype of this weight this parameter is still torch float 32 so our parameters seem to still be in float 32 but our activations the loits are now in P 16 so clearly this is why we get the mixed Precision some things pytorch is keeping inlow 32 some things pytorch is converting to lower Precision um and what gets converted at what point is not super clear I remember scrolling down is it here okay I can't find it I I thought it was here okay there we go so there are a few docks on when you're using this AutoCast what gets converted to B FL 16 and and when so for example only these Matrix multiply like operations get converted to float 16 but a lot of operations remain in float 32 so in particular a lot of normalizations like layer norms and things like that not all of those layers might be converted um so only some layers selectively would be running B flat 16 but things like softmax uh layer Norms uh log um log soft Max so loss function calculations a lot of those things might remain in float 32 because they are more susceptible to Precision changes major multiplies are fairly um robust to Precision changes uh so some parts of the network are um impacted more or less by the Precision change um so basically only some parts of the of the model are running in reduced Precision let's take it for a spin and let's actually see what kind of improvement we achieve here okay so we used to be 333 milliseconds we're now 300 and we used to be somewhere around 50,000 tokens per second we're now at 55 so we're definitely running faster but maybe not a lot faster and that's because there are still many many bottlenecks in our gbt2 we're just getting started but we have dropped down the precision as far as we can with my current GPU which is a100 we're using pytorch AutoCast unfortunately I don't actually exactly know what pytorch AutoCast do uh does I don't actually know exactly what's in B flat 16 what's in float 32 we could go in and we could start to scrutinize it um but these are the kinds of rules that pytorch has internally and unfortunately they don't documented very well uh so we're not going to go into that into in too much detail but for now we are training in B flow 16 we do not need a gradient scaler and the reason things are running faster is because um we are able to run tensor course in B FL 16 now that means we are in this row but uh we are also paying in Precision for this uh so um we expect slightly less accurate results with respect to the original fp32 but empirically in many cases this is a worth it uh kind of tradeoff because it allows you to run faster and you could for example train longer and make up for the uh for that Precision decrease so um that's b46 for

01:48:15 torch.compile, Python overhead, kernel fusion, 130ms

now okay so as we can see we are currently at about 300 milliseconds uh per iteration and we're now going to reach for some really heavy weapons in the pie torch Arsenal and in particular we're going to introduce torch. compile so torch. compile is really quite incredible infrastructure from the pytorch team and it's basically a compiler for neural networks like it's almost like GCC for CN C++ code this is just this GCC of neural nuts so came out a while ago and extremely simple to use um the way to use torch compile is to do this it's a single line of code to compile your model and return it now this line of code will cost you compilation time but as you might guess it's going to make the code a lot faster so let's actually run that because this will take some time to run but currently remember we're at 300 milliseconds and we'll see what happens now while this is running I'd like to explain a little bit of what torch. compile does under the hood uh so feel free to read this page of P torch but basically there's no real good reason for you to not use torch compile in your pie torch I kind of feel like you should be using almost by default if you're not uh unless you're debugging and you want your code to run really fast and there's one line here in torch compile that I found that actually kind of like gets to why this is faster speed up mainly comes from reducing python overhead and GPU read wrs so let me unpack that a little bit um okay here we are okay so we went from 300 milliseconds we're now running at 129 milliseconds so this is uh 300 129 about 2.3x Improvement from a single line of code in py torch uh so quite incredible so what is happening what's happening under the hood well when you pass the model to torch compile what we have here in this NN module this is really just the algorithmic description of what we'd like to happen in our Network and torch compile will analyze the entire thing and it will look at what operations You' like to use and with the benefit of knowing exactly what's going to happen it doesn't have to run in What's called the e mode it doesn't have to just kind of like go layer by layer like the python interpreter normally would start at the forward and the python interpreter will go okay let's do this operation and then let's do that operation and it kind of materializes all the operations as it goes through uh so these um calculations are dispatched and run in this order and the python interpreter and this code doesn't know what kind of operations are going to happen later but torch compile sees your entire code at the same time and it's able to know what operations you intend to run and it will kind of optimize that process the first thing it will do is will it will take out the python interpreter from the forward pass entirely and it will kind of compile this entire neural net as a single object with no python interpreter involved so it knows exactly what's going to run and we'll just run that and it's all going to be running in efficient code uh the second thing that happens is uh this read write that they mentioned very briefly so a good example of that I think is the G nonlinearity that we've been looking at so here we use the n and G now this here is me uh basically just breaking up the inang Galu uh which you remember has this formula so this here is the equivalent implementation to what's happening inside g algorithmic l it's identical Now by default if uh we just we using this instead of ending. G here what would happen without torch compile well the python interpreter would make its way here and then it would be okay well there's an input well let me first let me raise this input to the third power and it's going to dispatch a kernel that takes your input and raises it to the third power and that kernel will run and when this kernel runs what ends up happening is this input is stored in the memory of the GPU so here's a helpful example of the layout of what's happening right you have your CPU this is in every single computer there's a few cores in there and you have your uh Ram uh your memory and the CPU can talk to the memory and this is all well known but now we've added the GPU and the GPU is a slightly different architecture of course they can communicate and it's different in that it's got a lot more course than a CPU all of those cores are individually a lot simpler too but it also has memory right this high bandwidth memory I'm sorry if I'm botching it hbm I don't even know what that stands for I'm just realizing that but uh this is the memory and it's very equivalent to uh RAM basically in the computer and what's happening is that input is living in the memory and when you do input cubed this has to travel to the GPU to the course and to all the caches and registers on the actual chip of this GPU and it has to calculate the all the elements to the third and then it saves the result back to the memory and it's this uh travel time that actually causes a lot of issues so here remember this memory bandwidth we can communicate about 2 terabytes per second which is a lot but also we have to Traverse this link and it's very slow so here on the GPU we're on chip and everything is super fast within the chip but going to the memory is extremely expensive takes extremely long amount of time and so we load the input do the calculations and load back the output and this round trip takes a lot of time and now right after we do that we multiply by this constant so what happens then is we dispatch another kernel and then the result travels back all the elements get multiplied by a constant and then the results travel back to the memory and then we take the result and we add back input and so this entire thing again travels to the GPU adds the inputs and gets written back so we're making all these round trips from the memory to actually where the comput happens because all the tensor cores and alus and everything like that is all stored on the chip in the GPU so we're doing a ton of round trips and pytorch uh without using torch compile doesn't know to optimize this because it doesn't know what kind of operations you're running later you're just telling it raise the power to the third then do this then do that and it will just do that in that sequence but torch compile sees your entire code it will come here and it will realize wait all of these are elementwise operations and actually what I'm going to do is I'm going to do a single trip of input to the GPU then for every single element I'm going to do all of these operations while that memory is on the GPU or chunks of it rather and then I'm going to write back a single time so we're not going to have these round trips and that's one example of what's called kernel fusion and is a major way in which everything is sped up so basically if you have your benefit of onet and you know exactly what you're going to compute you can optimize your round trips to the memory and you're not going to pay the the memory bandwidth cost and that's fundamentally what makes some of these operations a lot faster and what they mean by read writes here so let me erase this because we are not using it and yeah we should be using torch compile and our code is now significantly faster and we're doing about 125,000 tokens per second but we still have a long way to go before we move on I wanted to supplement the discussion a little bit with a few more figures uh because this is a complic topic but it's worth understanding on a high level uh what's happening here and I could probably spend an entire video of like two hours on this but just the preview of that basically so this chip here that is uh the GPU this chip is where all the calculations happen mostly but this chip also does have some memory in it but most of the memory by far is here in the high bandwidth memory hbm and is connected they're connected um but these are two separate chips basically now here this is a zoom in of kind of this cartoon diagram of a GPU and what we're seeing here is number one you see this hbm I I realize it's probably very small for you but on the sides here it says hbm and so that that's the links to the hbm now the hbm is again off chip on the chip there are a large number of these streaming multiprocessors uh every one of these is an

SM there's 120 of them in total and this is where the a lot of the calculations happen and this is a zoom in of a single individual as it has these four quadrants and see for example tensor core this is where a lot of the Matrix multiply stuff happens but there's all these other units to do all different kinds of calculations for fp64 fp32 and for integers and so on now so we have all this uh logic here to do the calculations but in addition to that on the chip there is memory sprinkled throughout the chip so L2 cache is some amount of memory that lives on the chip and then on the SMS themselves there's L1 cache I realized it's probably very small for you but this blue bar is L1 and there's also registers um and so there is memory stored here but the way this memory is stored is very different from the way memory is stored in hbm uh this is a very different implementation uh using um just in terms of like what the Silicon looks like it's a very different implementation um so here you would using transistors and capacitors and here it's a very different implementation uh with SRAM and what that looks like but long story short is um there is um memory inside the chip but it's not a lot of memory that's the critical point so this is some C this is a example diagram of a slightly different GPU just like here where it shows that for example typical numbers for CPU Dam memory which is this thing here you might have one tab of this right but it would be extremely expensive to access especially for a GPU you have to go through the CPU here now next we have the hbm so we have tens of gigabytes of hbm memory on a typical GPU here but it's as I mentioned very expensive to access and then on the chip itself everything is extremely fast within the chip but we only have couple 10 megabytes of memory collectively throughout the Chip And so there's just not enough space because the memory is very expensive on the chip and so there's not a lot of it but it is lightning fast to access in relative terms and so basically whenever we have these kernels um the more accurate picture of what's Happening Here is that we take these inputs which live by default on the global memory and now we need to perform some calculation so we start streaming the data from the um Global memory to the uh chip we perform the calculations on the chip and then stream it back and store it back to the global memory right and so if we are if we don't have torch compile we are streaming the data through the chip doing the calculations and saving to the memory and we're doing those round trips many many times but uh if it's torch compiled then we start streaming the memory as before but then while we're on the chip we're we're we have a chunk of the uh data that we're trying to process so that chunk now lives on the chip while it's on the chip it's extremely fast to operate on so if we have kernel Fusion we can do all the operations right there in an element-wise fashion and those are very cheap and then we do a single round trip back to the global memory so operator Fusion basically allows you to keep your chunk of data on the Chip And do lots of calculations on it before you write it back and that gives huge savings and that's why torch compile ends up being a lot faster or that's one of the major reasons uh so again just a very brief intro to the memory hierarchy and roughly what torch compile does for you

02:00:18 flash attention, 96ms

now torch compile is amazing but there are operations torch compile will not find and an amazing example of that is Flash attention to which we turn next so flash attention comes from this paper from uh Stanford in 2022 and it's this incredible algorithm for performing attention so um and running it a lot faster so flash attention will come here and we will take out these four lines and Flash attention implements these four lines really really quickly and how does it do that well flash attention is a kernel Fusion operation so you see here we have um in this diagram they're showing P torch and you have these four operations uh they're including Dropout but we are not using Dropout here so we just have these four lines of code here and instead of those we are fusing them into a single fused kernel of flash attention so it's an it's a it's a kernel

Fusion algorithm but it's a kernel Fusion that torch compile cannot find and the reason that it cannot find it is that it um requires an algorithmic rewrite of how attention is actually implemented here in this case and what's remarkable about it is that uh flash attention actually if you just count the number of flops flash attention does more flops than this attention here but flash attention is actually significantly faster in fact they site 7. six times faster potentially and that's because it is very mindful of the memory hierarchy as I described it just now and so it's very mindful about what's in high bandwidth memory what's in the shared memory and it is very careful with how it orchestrates the computation such that we have fewer reads and writes to the high bandwidth memory and so even though we're doing more flops the expensive part is they load and store into hbm and that's what they avoid and so in particular they do not ever materialize this end byend attention Matrix this ATT here a flash attention is designed such that this Matrix never gets materialized at any point and it never gets read or written to the hbm and this is a very large Matrix right so um because this is where all the queries and keys interact and we're sort of getting um for each head for each batch element we're getting a t BYT Matrix of attention which is a Million numbers even for a single head at a single batch index at like so so basically this is a ton of memory and and this is never materialized and the way that this is achieved is that basically the fundamental algorithmic rewrite here relies on this online softmax trick which was proposed previously and I'll show you the paper in a bit and the online softmax trick coming from a previous paper um shows how you can incrementally evaluate a soft Max without having to sort of realize all of the inputs to the softmax to do the normalization and you do that by having these intermediate variables M and L and there's an update to them that allows you to evaluate the softmax in an online manner um now flash attention actually so recently flash attention 2 came out as well so I have that paper up here as well uh that has additional gains to how it calculates flash attention and the original paper that this is based on basically is this online normalizer calculation for softmax and remarkably it came out of Nvidia and it came out of it like really early 2018 so this is 4 years before flash attention and this paper says that we propose a way to compute the classical softmax with fewer memory accesses and hypothesize that this reduction in memory accesses should improve softmax performance on actual hardware and so they are extremely correct in this hypothesis but it's really fascinating to me that they're from Nvidia and that they had this realization but they didn't actually take it to the actual flash attention that had to come four years later from Stanford so I don't fully understand the historical how this happened historically um but they do basically propose this online update to the softmax uh right here and this is fundamentally what they reuse here to calculate the softmax in a streaming Manner and then they realize they can actually fuse all the other operations with the online sofx calculation into a single fused kernel flash attention and that's what we are about to use so great example I think of being aware of um memory hierarchy the fact that flops don't matter uh the entire memory access pattern matters and that torch compile is amazing but there are many optimizations that are still available to us that potentially torch compile cannot find maybe maybe one day it could but right now it seems like a lot to ask so here's what we're going to do we're going to use Flash attention and the way to do that basically in pytorch is we are going to comment out these four lines and we're going to replace them with a single line and here we are calling this compound operation in pytorch called scale that product attention and uh pytorch will call flash attention when you use it in this way I'm not actually 100% sure why torch compile doesn't realize that these four lines should just call flash attention in this exact way we have to do it again for it which in my opinion is a little bit odd but um here we are so you have to use this compound up and uh let's wait for a few moments before torch comp compile gets around to it and then let's remember that we achieved 6.05 661 I have it here that's the loss we were expecting to see and we took 130 milliseconds uh before this change so we're expecting to see the exact same result by iteration 49 but we expect to see faster runtime because Flash attention is just a an algorithmic rewrite and it's a faster kernel but it doesn't actually change any of the computation and we should have the exact same optimization so okay so we're a lot faster we're at about 95 milliseconds and we achiev 6.58 okay so they're basically identical up to a floating Point fudge Factor so it's the identical computation but it's significantly faster going from 130 to roughly 90 96 and so this is um 96 divide 130ish so this is maybe 27 is% Improvement um so uh really interesting and that is Flash retention okay we are

02:06:54 nice/ugly numbers. vocab size 50257 50304, 93ms

now getting to one of my favorite optimizations and it is simultaneously the dumbest and the most brilliant optimization and it's always a little bit surprising to me um anyway so basically I mentioned a few minutes ago that there are some numbers that are nice and some numbers that are ugly so 64 is a beautiful nice number 128 is even nicer 256 is beautiful what makes these numbers beautiful is that there are many powers of two inside them you can divide by two many times and uh examples of ugly numbers are like 13 and 17 and something like that prime numbers numbers that are not even and so on and so pretty much you always want to use nice numbers in all of your code that deals with neural networks or Cuda because everything in Cuda Works in sort of like powers of two and lots of kernels are written in terms of powers of Two And there are lots of blocks of sizes 16 and uh 64 and so on so everything is written in those terms and you always have special case handling for all kinds of uh logic that U when your inputs are not made of nice numbers so let's see what that looks like basically scan your code and look for ugly numbers is roughly theistic so three times is kind of ugly um I'm not 100% sure maybe this can be improved but this is uh this is ugly and not ideal um four times is nice so that's uh that's nice 1024 is very nice that's a power of two 12 is a little bit suspicious um not too many powers of two 768 is great 50, 257 is a really really ugly number um it's first of all it's odd so uh and there's no not too many powers of two in there so this is a very ugly number and it's highly suspicious and then when we scroll down all these numbers are nice and then here we have mostly nice numbers except for 25 so in this configuration of gpt2 XL a number of heads is 25 uh that's a really ugly number that's an odd number and um actually this did cause a lot of headaches for us recently when we're trying to optimize some kernels uh to run this fast um and required a bunch of special case handling so basically these numbers are we have some ugly numbers and some of them are easier to fix than others and in particular the voap size being 50257 that's a very ugly number very suspicious and we want to fix it now when you when you fix these things uh one of the easy ways to do that is you basically um increase the number until it's the nearest power of two that you like so here's a much nicer number it's 50304 and why is that because 50304 can be divided by 8 or by 16 or by 32 64 it can even be divided by 128 I think yeah so it's a very nice number um so what we're going to do here is the GPT config and you see that we initialized B cap size to 50257 Let's override just that um element to be 50304 okay so everything else stays the same we're just increasing our vocabulary size so we're adding it's almost like we're adding fake tokens uh so that book up size has powers of two inside it now actually what I'm doing here by the way is I'm increasing the amount of computation that our network will be doing if you just count the the flops on like do the math of how many flops we're doing we're going to be doing more flops and we still have to think through whether this doesn't break anything but if I just run this uh let's see what we get uh currently this ran in maybe 96.5 milliseconds per step I'm just kind of like eyeballing it and let's see what kind of a result we're going to get uh while this is compiling let's think through whether our code actually works okay when we increase the vocap size like this let's look at where vocap size is actually used so we swing up to the inet and we see that it's used inside the embedding table of course so all the way at the bottom of the Transformer and it's used at the classifier layer all the way at the top of the Transformer so in two places and let's take a look and we're running at 93 so 93 milliseconds instead of 96.5 so we are seeing a roughly yeah 4% Improvement here uh by doing more calculations and the reason for this is we fixed we've made an ugly number into a nice number let's I'm going to come into the explanation for that a little bit again but for now let's just convince ourselves that we're not breaking anything when we do this so first of all we've made the the wte the embedding table for the tokens we've made it larger it's almost like we introduced more tokens at the bottom and these tokens are never used because the gbt tokenizer only has tokens up to $50,000 256 and so we'll never index into the rows that we've added so we're wasting a little bit of space here by creating memory that's never going to be accessed never going to be used Etc now that's not fully correct because this wte weight ends up being shared and ends up being used in the classifier here at the end so what is that doing to the classifier right here well what what that's doing is we're predicting additional Dimensions at the classifier now and we're predicting probabilities for tokens that will of course never be present in the training set um and so therefore the network has to learn that these probabilities uh have to be driven to zero and so the logits that the network produces have to drive those dimensions of the output to negative Infinity but it that's no different from all the other tokens that are already in our data set um or rather that are not in our data set so Shakespeare only probably uses let's say a th000 tokens out of 50,000 to 57 tokens so most of the tokens are already being driven to zero probability by the optimization we' just introduced a few more tokens now that in a similar manner will never be used and have to be driven to zero in probability um so functionally though nothing breaks we're using a bit more extra um memory but otherwise this is a harmless operation as far as I can tell but and we're adding calculation but it's running faster and it's running faster because as I mentioned in Cuda so many kernels use uh block tiles and these block towels are usually nice numbers uh so powers of two so calculations are done in like chunks of 64 or chunks of 32 and when your um when your desired calculation doesn't neatly fit into those block tiles um there are all kinds of boundary kernels that can kick in to like do the last part so basically in a lot of kernels they will chunk at up your input and they will do the nice part first and then they have a whole second second phase where they come back to any that like uh remains uh and then they process the remaining part and the kernels for that could be very inefficient and so you're basically um spinning up all this extra compute and is extremely inefficient so you might as well pad your inputs and um make it fit nicely and usually that empiric lens up actually running faster um so this is another example of a 4% Improvement that we've added and this is something that also torch compile did not find for us you would hope that torch compile at some point could figure an optimization like this out uh but for now uh this is it and I also have to point out that we're using pytorch nightly so that's why we're only seeing 4% if you're using pytorch 2.3.1 or earlier you would actually see something like 30% Improvement just from this change from changing it to from 50,000 to 57 to 50304 so again one of my favorite examples also of having to understand the under the hood and how it all works and to know what kinds of things to Tinker with to push the performance of your code okay so at this point we have

02:14:55 SECTION 3: hyperpamaters, AdamW, gradient clipping

improved the performance by about 11x right because we started at about 1,000 milliseconds per step and we're now down to like 93 milliseconds so that's uh quite good and we're uh doing a much better job of utilizing our GPU resources so I'm going to now turn to more algorithmic changes uh and improvements to the actual optimization itself and what we would like to do is we would like to follow the hyper parameters that are mentioned in the GP G2 or gpt2 gpt3 paper now sadly gpt2 is uh doesn't actually say too much it's very nice of them that they released the model weights and the code but the paper itself is extremely vague as to the optimization details uh the code itself that they released as well the code we've been looking at this is just the inference code so there's no training code here and very few hyp parameters so this doesn't also tell us too much so for that we have to turn to the gpt3 paper and um in the depending of the gpt3 paper um they have a lot more hyper parameters here for us to use and the gpt3 paper in general is a lot more detailed as to uh all of the you know small details that go into the model training but gpt3 U models were never released so gbt2 we have the weights but no details and gpt3 we have lots of details but no weights so um but roughly speaking gpt2 and gpt3 architectures are very very similar and um basically there are very few changes the context length was expanded from 1024 to 2048 and that's kind of like the major change uh and some of the hyper parameters around the Transformer have changed but otherwise they're pretty much the same model it's just that gpt3 was trained for a lot longer on a bigger data set and uh has a lot more thorough evaluations uh and the gpt3 model is 175 billion instead of 1.6 billion um in the gpt2 so long story short we're going to go to gp3 paper to follow along some the hyper parameters so to train all the versions of gpt3 we use atom with beta 1 beta 2 of9 and .95 so let's swing over here and make sure that the betas parameter which you can see here defaults to 0.9 and 999 is actually set to 0.9 and .95 and then the Epsilon parameter uh you can see is the default is 1 in8 and this is also one in8 let's just uh put it in so that works expit uh now next up they say we clip the gra Global Norm of the gradient at 1.0 so what this is referring to is that once we calculate the gradients right after l. backward um we basically have the gradients at all the parameter tensors and what people like to do is basically uh clip them to have some kind of a maximum Norm so in pytor this is fairly easy to do uh it's one line of code here that we have to insert right after we calcul Cal the gradients and what this utility function is doing is um it's calculating the global Norm of the parameters so every single par um gradient on all the parameters you square it and you add it all up and you take a big square root of that and that's the norm of the parameter V Vector basically it's the it's the length of it if you if you'd like to look at it that way and we are basically making sure that its length is no more than 1.0 and we're going to clip it and the reason that people like to use this is that uh sometimes you can get unlucky during your optimization maybe it's a bad data batch or something like that and if you get very unlucky in the batch you might get really high loss and really high loss could lead to a really high gradient and this could basically uh shock your model and shock the optimization so people like to use a gradient Norm clipping uh to prevent the model from um basically getting too big of shocks in terms of the gradient magnet ude and uh the upper bound it in this way it's a bit of a hacky solution it's about like a patch on top of like deeper issues uh but uh people still do it fairly frequently now the clip grad Norm Returns the norm of the gradient which I like to always visualize uh because um it is useful information and sometimes you can look at the norm of the gradient and if it's well behaved things are good if it's climbing things are bad and they're destabilizing during training sometimes you could get a spike in the norm and that means there's some kind of an issue or an instability so the norm here will be a norm uh and let's do a uh 4f or something like that and I believe this is just a float and so we should be able to uh print that uh so that's Global gradient clipping now they go into the details of the learning rate uh scheduler so they don't just use a fixed learning rate like we do here for 3 E4 but there's actually basically a cosine DK learning rate schedule um it's got a warm-up and it's got a cosine DEC to 10% over some Horizon um and so we're going to implement uh this in a second I just like to see Norm printed here okay there we go so what happened here is the norm is actually really high in the beginning 30 or so and you see that as we continue training it kind of like stabilizes um at values below one um and this is not that crazy uncommon for the norm to be high in the very first few stages basically What's Happening Here is the model is completely random and so there's a ton of learning happening very early in the network but that learning is kind of like um you know it's mostly learning the biases of the output tokens and so it's a bit of an unstable time uh but the network usually stabilizes in a very few iterations so this looks very relatively reasonable to me except usually I would expect this looks a little bit funky that we go from 28 to 6 to 2 and then to 10 um it's not completely insane but it's just kind of

a little bit funky um okay so let's now get to

the learning rate schuer so the learning

02:21:06 learning rate scheduler: warmup + cosine decay

rate schedule that's used here in gpt3 is what's called a cosine Decay learning schedule with warmup and the way this looks is that the learning rate is basically starts right at around zero linearly rank s up over some amount of time and then comes down with this cosine sort of form and comes down to some kind of a minimum learning rate that's up to you so here the minimum learning rate is zero but uh here in the paper they said that they use cosine Decay for learning rate down to 10% of its value over the first 260 billion tokens and then training continues 10% after and there's a linear warmup over the first 375 million tokens so that's about the learn R so let's now implement this uh so I already implemented it here and the way this works is let me scroll down first here I changed our training Loop a little bit so this was a 4i in

Max steps I just change it to step now so that we have the notion of a step is a single optimization step in the in the for Loop and then here I get the LR for this step of the optimization using a new function I call get LR and then in pytorch to set the learning rate I think this is is the way to set the learning rate it's a little bit gnarly um because you have to basically there's a notion of different par parameter groups that could exist in the optimizer and so you actually have to iterate over them even though we currently have a single param group only um and you have to set the LR in this for Loop kind of style is is my impression right now so we have this look of LR we set the learning rate and then on the bottom I'm also printing it uh so that's all the changes I made to this Loop and then of course the get LR is my scheduler now it's worth pointing out that pytorch actually has learning rate schedulers and you can use them and I believe there's a cosine learning rate schedule in pytorch I just don't really love using that code because honestly it's like five lines of code and I fully understand what's happening inside these lines so I don't love to use abstractions where they're kind of in screwable and then I don't know what they're doing so personal style so the max learning rate here is let's say 3 E4 but we're going to see that in gpt3 here they have a table of what the maximum learning rate is for every model size so um for for this one basically 12 12 layer 768 gpt3 so the gpt3 small is roughly like a GPT 2124m we see that here they use a learning rate of 6 E4 so we could actually go higher um in fact we may want to try to follow that and just set the max LR here at six uh then the that's the maximum learning rate the minum learning rate is uh 10% of that per description in the paper some number of steps that we're going to warm up over and then the maximum steps of the optimization which I now use also in the for Loop down here and then you can go over this code if you like it's not U it's not terribly inside Flor interesting I'm just uh modulating based on the iteration number which learning rate uh there should be so this is the warm-up region um this is the region after the optimization and then this is the region sort of in between and this is where I calculate the cosine learning rate schedule and you can step through this in detail if you'd like uh but this is basically implementing this curve and I ran this already and this is what that looks like um so when we now run we start at um some very low number now note that we don't start exactly at zero because that would be not useful to update with a learning rate of zero that's why there's an it+ one so that on the zeroth iteration we are not using exactly zero we're using something very very low then we linearly warm up to maximum learning rate which in this case was 34 when I ran it but now would be 6 E4 and then it starts to decay all the way down to um 3 E5 which was at the time 10% of the original learning rate now one thing we are not following exactly is that they mentioned that um let me see if I can find it again we're not exactly following what they did because uh they mentioned that their training Horizon is 300 billion tokens and they come down to 10% of the initial learning rate of at 260 billion and then they train after 260 with 10% so basically their Decay time is less than the max steps time whereas for us they're exactly equal so it's not exactly faithful but it's um it's an okay um this is okay for us and for our purposes right now and um we're just going to use this ourselves I don't think it makes too too big of a difference honestly I should point out that what learning rate schedule you use is totally up to you there's many different types um coign learning rate has been popularized a lot by gpt2 and gpt3 but people have come up with all kinds of uh other learning rate schedules um and this is kind of like an active area of uh research as to which one is the most effective at train these networks okay next up the paper talks

02:26:21 batch size schedule, weight decay, FusedAdamW, 90ms

about the gradual batch size increase so there's a ramp on the batch size that is linear and you start with very small batch size and you ramp up to a big batch size over time uh we're going to actually skip this and we're not going to work with it and the reason I don't love to use it is that it complicates a lot of the arithmetic because you are changing the number of tokens that you're processing at every single step of the optimization and I like to keep that math very very simple also my understanding is that that this is not like a major um Improvement and also my understanding is that this is not like an algorithmic optimization Improvement it's more of a systems and speed Improvement and roughly speaking this is because uh in the early stages of the optimization uh again the model is in a very atypical setting and mostly what you're learning is that um you're mostly learning to ignore the tokens uh that don't come up in your training set very often you're learning very simple biases and and that kind of a thing and so every single example that you put through your network is basically just telling you use these tokens and don't use these tokens and so the gradients from every single example are actually extremely highly correlated they all look roughly the same in the in the OR original parts of the optimization because they're all just telling you that these tokens don't appear and these tokens do appear and so because the gradients are all very similar and they're highly correlated then why are you doing batch sizes of like Millions when if you do a batch size of 32k you're basically getting the exact same gradient early on in the training and then later in the optimization once you've learned all the simple stuff that's where the actual work starts and that's where the gradients become more decorrelated per examples and that's where they actually offer you sort of statistical power in some sense um so we're going to skip this just because it kind of complicates things and we're going to go to uh data are sampled without replacement during training um so until an Epoch boundary is reached so without replacement means that they're not sampling from some fixed pool and then uh take a sequence train on it but then also like return the sequence to the pool they are exhausting a pool so when they draw a sequence it's it's gone until the next Epoch of training uh so we're already doing that because our data loader um iterates over chunks of data so there's no replacement they don't become eligible to be drawn again until the next P so we're basically already doing that um all models use a weight decay of 0.1 to provide a small amount of regularization so let's Implement a weight Decay and you see here that I've already kind of made the changes and in particular instead of creating the optimizer right here um I I'm creating a new configure optimizers function inside the model and I'm passing in some of the hyper parameters instead so let's look at the configure optimizers which is supposed to return the optimizer object okay so it looks complicated but it's actually really simple and it's just um we're just being very careful and there's a few settings here to go through the most important thing with respect to this line is that you see there's a weight Decay parameter here and I'm passing that into um well I'm passing that into something called optim groups that eventually ends up going into the addom W Optimizer um and the weight Decay that's by default used in Addam W here is 0.01 so it's it's u 10 times lower than what's used in gpt3 paper here um so the weight dek basically ends up making its way into the ADD and W through the optimizer groups now what else is going on here in this uh function so the two things that are happening here that are important is that I'm splitting up the parameters into those that should be weight decayed and those that should not be weight decayed so in particular it is common to not weight decay uh biases and any other sort of one-dimensional tensors so the one-dimensional tensors are in the no Decay prams and these are also things like uh layer Norm scales and biases it doesn't really make sense to weight Decay those you mostly want to weight Decay uh the weights that participate in Matrix multiplications and you want to potentially weight Decay the embeddings and uh We've covered in previous video why it makes sense to Decay the weights because you can sort of the it as a regularization because when you're pulling down all the weights you're forcing the optimization to use more of the weights um and you're not allowing any one of the weights individually to be way too large um you're forcing you're forcing the network to kind of like distribute the work across more channels because there's sort of like a pull of gravity on the weights themselves um so that's why we are separating it in those ways here we're only decaying the embeddings and the mmal participating ways uh we're printing the number of uh parameters that we decaying and not most of the parameters will be decayed and then one more thing that we're doing here is I'm doing another optimization here and previous add and W did not have this option but later parts of pytorch introduced it and that's why I'm guarding it with an inspect do signature which is basically checking if this fused um quar is present inside atom W and then if it is present I'm going to end up using it and passing it in here because some earlier versions do not have fused equals so here's adamw fused equals it did not used to exist and it was added later and there's some docks here for what's happening and basically they say that by default they do not use fused because it is relatively new and we want to give it sufficient big time so by default they don't use fused but fused is a lot faster when it is available and when you're running on Cuda and what that does is in instead of iterating in a for Loop over all the parameter tensors and updating them that would launch a lot of kernels right and so a fused just means that it's a um all those kernels are fused into a single kernel you get rid of a lot of overhead and you a single time on all the parameters call a uh kernel that updates them and so it's just basically a kernel Fusion for the atom W update instead of iterating over all the tensors so that's the configure optimizers function that I like to use and we can rerun and we're not going to see any major differences from what we saw before but we are going to see some prints uh coming from here so let's just take a look at what they look like so we see that number of Decay tensors is 50 and it's most of the parameters and number of non- deay tensors is 98 and these are the biases and the layer Norm parameters mostly and that's there's only 100,000 of those so most of it is decayed and then we are using the fused implementation of ATM W which will be a lot faster so if you have it available I would advise you to use it I'm not actually 100% sure why they don't default to it it seems fairly benign and harmless and also because we are using the fused implementation I think this is why we have dropped um notice that the running time used to be 93 milliseconds per step and we're now down to 90 milliseconds per step because of using the fused atom W Optimizer so in a single commit here we are introducing fused atom getting improvements on the time and we're adding or changing the weight Decay but we're only weight decaying the two dimensional parameters the embeddings and the matrices that participate in linear so that is this and we can take this out and uh yeah that is it for this line one more quick

02:34:09 gradient accumulation

note before we continue here I just want to point out that the relationship between weight Decay learning rate batch size the atom parameters beta 1 beta 2 the Epsilon and so on these are very complicated uh mathematical relationships in the optimization literature and um for the most part I'm in this video I'm just trying to copy paste the settings that open AI used but this is a complicated topic uh quite deep and um yeah in this video I just want to copy the parameters because it's a whole different video to really talk about that in detail and give it a proper Justice instead of just high level intuitions uh now the next thing that I want to move on to is that uh this paragraph here by the way we're going to turn back around to when we improve our data loader for now I want to swing back around to this table where you will notice that um for different models we of course have different U hyper parameters for the Transformer that dictate the size of the Transformer Network we also have a different learning rate so we're seeing the pattern that the bigger networks are trained with slightly lower learning rates and we also see this batch size where in in the small networks they use a smaller batch size and in the bigger networks they use a bigger batch size now the problem with for us is we can't just use 0.5 million batch size because uh if I just try to come in here and I try to set uh this uh B where is my b um b equals where where do I call the DAT okay b equal 16 if I try to set um well well we have to be careful it's not 0.5 million because this is the badge size in the number of tokens every single one of our rows is24 tokens so 0.5 E6 1 million divide 1024 this would need about a 488 match size so the problem is I can't come in here and set this to 488 uh because my GPU would explode um this would not fit for sure and so but we still want to use this batch size because again as I mentioned the batch size is correlated with all the other optimization hyper parameters and the learning rates and so on so we want to have a faithful representation of all the hyper parameters and therefore we need to uh use a bat size of .5 million roughly but the question is how do we use .5 million if we only have a small GPU well for that we need to use what's called gradient accumulation uh so we're going to turn to that next and it allows us to simulate in a Serial way any arbitrary batch size that we set and so we can do a batch size of .5 million we just have to run longer and we have to process multiple sequences and basically add up all the gradients from them to simulate a batch size of .5 million so let's turn to that next okay so I started the implementation right here just by adding these lines of code and basically what I did is first I set the total batch size that we desire so this is exactly .5 million and I used a nice number a power of two uh because 2 to the 19 is 524 288 so it's roughly .5 million it's a nice number now our micro batch size as we call it now is 16 so this is going to be we still have B BYT in the SE that go into the Transformer and do forward backward but we're not going to do an update right we're going to do many forward backwards we're going to and those gradients are all going to plus equals on the parameter gradients they're all going to add up so we're going to do forward backward grad akum steps number of times and then we're going to do a single update once all that is accumulated so in particular our micro batch size is just now controlling how many tokens how many rows we're processing in a single go over a forward backward so um here we are doing 16 * 124 we're doing 16

384 um tokens per forward backward and we are supposed to be doing 2 to the 19 whoops what am I doing 2 to the 19 in total so the grat Aon will be 32 uh so therefore gr AUM here will work out to 32 and we have to do 32 forward backward um and then a single update now we see that we have about 100 milliseconds for a singer forward backward so doing 32 of them will be will make every step roughly 3 seconds just napkin math so that's grum steps but now we actually have to Implement that so we're going to swing over to our training Loop because now this part here and this part here the forward and the backward we have to now repeat this 32 times before we do everything else that follows so let's uh see how we can Implement that so let's come over here and actually we do have to load a new batch every single time so let me move that over here and now this is where we have the inner loop so for micro step in

range graum steps we do this and remember that l. backward always deposits gradients so we're doing inside losta backward there's always a plus equals on the gradients so in every single L of backward gradients will add up on the gradient tensors um so we lost that backward and then we get all the gradients over there and then we normalize and everything else should just follow um so we're very close but actually there's like subtle and deep issue here and this is actually incorrect so invite I invite you to think about why this is not yet sufficient um and uh let me fix it then okay so I brought back the jupyter notebook so we can think about this carefully in a simple toy setting and see what's happening so let's create a very simple neural nut that takes a 16 Vector of 16 numbers and returns a single number and then here I'm creating some random uh examples X and some targets uh y Y and then we are using the mean squared loss uh here to calculate the loss so basically what this is is four individual examples and we're just doing Simple regression with the mean squared loss over those four examples now when we calculate the loss and we lost that backward and look at the gradient this is the gradient that we achieve now the loss objective here notice that in MSE loss the default for the loss function is reduction is mean so we're we're calculating the average mean loss um the the mean loss here over the four examples so this is the exact loss objective and this is the average the one over four because there are four independent examples here and then we have the four examples and their mean squared error the squared error and then this makes it the mean squared error so therefore uh we are we calculate the squared error and then we normalize it to make it the mean over the examples and there's four examples here so now when we come to the gradient accumulation version of it this uh this here is the gradient accumulation version of it where we have grad acum steps of four and I reset the gradient we've grum steps of four and now I'm evaluating all the examples individually instead and calling L that backward on them many times and then we're looking at the gradient that we achieve from that so basically now we forward our function calculate the exact same loss do a backward and we do that four times and when we look at the gradient uh you'll notice that the gradients don't match so here we uh did a single batch of four and here we did uh four gradient accumulation steps of batch size one and the gradients are not the same and basically the the reason that they're not the same is exactly because this mean squared error gets lost this one quarter in this loss gets lost because what happens here is the loss of objective for every one of the loops is just a mean squ error um which in this case because there's only a single example is just this term here so that was the loss in the zeroth eration same in the first third and so on and then when you do the loss. backward we're accumulating gradients and what happens is that accumulation in the gradient is basically equivalent to doing a sum in the loss so our loss actually here is this without the factor of one quarter outside of it so we're missing the normalizer and therefore our gradients are off and so the way to fix this or one of them is basically we can actually come here and we can say loss equals loss divide 4 and what happens now is that we're introducing we're we're scaling our loss we're introducing a one quarter in front of all of these places so all the individual losses are now scaled by one quarter and and then when we backward all of these accumulate with a sum but now there's a one quarter inside every one of these components and now our losses will be equivalent so when I run this you see that the U gradients are now identical so long story short with this simple example uh when you step through it you can see that basically the reason that this is not correct is because in the same way as here in the MSE loss the loss that we're calculating here in the model is using a reduction of mean as well uh so where's the loss after that cross entropy and by default the reduction uh here in Cross entropy is also I don't know why they don't show it but it's the mean uh the mean uh loss at all the B BYT elements right so there's a reduction by mean in there and if we're just doing this gradient accumulation here we're missing that and so the way to fix this is to simply compensate for the number of gradient accumulation steps and we can in the same way divide this loss so in particular here the number of steps that we're doing is loss equals loss divide gradient accumulation steps so even uh co-pilot s gets the modification but in the same way exactly we are scaling down the loss so that when we do loss that backward which basically corresponds to a sum in the objective we are summing up the already normalized um loss and and therefore when we sum up the losses divided by grum steps we are recovering the additional normalizer uh and so now these two will be now this will be equivalent to the original uh sort of optimization because the gradient will come out the same okay so I had to do a few more touch-ups and I launched launched the optimization here so in particular one thing we want to do because we want to print things nicely is well first of all we need to create like an accumulator over the loss we can't just print the loss because we'd be printing only the final loss at the final micro step so instead we have loss ofon which I initialize at zero and then I accumulate a uh the loss into it and I'm using detach so that um uh I'm detaching the tensor uh from the graph and I'm just trying to keep track of the values so I'm making these Leaf nodes when I add them so that's lakum and then we're printing that here instead of loss and then in addition to that I had to account for the grum steps inside the tokens processed because now the tokens processed per step is B * T * gradient accumulation so long story short here we have the optimization it looks uh reasonable right we're starting at a good spot we calculated the grum steps to be 32 and uh we're getting about 3 seconds here right um and so this looks pretty good now if you'd like to verify that uh your optimization and the implementation here is correct and your working on a side well now because we have the total patch size and the gradient accumulation steps our setting of B is purely a performance optimization kind of setting so if you have a big GPU you can actually increase this to 32 and you'll probably go a bit faster if you have a very small GPU you can try eight or four but in any case you should be getting the exact same optimization and the same answers up to like a floating Point error because the gradient accumulation kicks in and um and can um handle everything serially as an Neary so uh that's it for gradient accumulation I think okay so now is the

02:46:52 distributed data parallel (DDP)

time to bring out the heavy weapons uh you've noticed that so far we've only been using a single GPU for training but actually I am paying for eight gpus here and so uh we should be putting all of them to work and in particular they are going to collaborate and uh you know optimize over tokens at the same time and communicate so that um uh they're all kind of collaborating on the optimization for this we are going to be using the distributed data parallel from pytorch there's also a legacy data parallel which I recommend you not use and that's kind of like you know Legacy distributed data parallel Works in a very simple way we have eight gpus so we're going to uh launch eight processes and each process is going to be assigned to GPU and for each process the training Loop and everything we've worked on so far is going to look pretty much the same H GPU as far as it's concerned is just working on exactly what we've built so far but now Secret L there's eight of them and they're all going to be processing slightly different parts of the data and we're going to add one more part where once they all calculate their gradients there's one more part where we do a average of those gradients and so that's how they're going to be collaborating on uh the computational workload here so to use all eight of them we're not going to be launching our script anymore with just um pytorch train gbt2 piy we're going to be running it with a special command called torrun in pytorch we'll see that in a bit and torrun uh when it runs our python script we'll actually make sure to run eight eight of them in parallel and it creates these environmental variables where each of these processes can look up which uh basically which one of the processes it is so for example torron will set rank local Rank and World size environmental variables and so this is a bad way to detect whether uh DDP is running so if we're using torch run if DDP is running then uh we have to make sure that K is available because I don't know that you can run this on CPU anymore or that that makes sense to do um this is some um setup code here the important part is that there's a world size which for us will be eight that's the total number of processes running there's a rank which is um each process will basically run the ex exact same code at the exact same time roughly but all the process the only difference between these processes is that they all have a different dtp rank so the um gpu0 will have DDP rank of zero GPU 1 will have uh rank of one Etc so otherwise they're all running the exact same script it's just that DDP rank will be a slightly different integer and that is the way for us to coordinate that they don't for example run on the same data we want to we want them to run on different parts of the data and so on now local rank is something that is only used in a multi- node setting we only have a single node with ag gpus and so local rank is the rank of the GPU on a single node so from 0 to seven as an example but for us we're mostly going to be running on a single box so the things we care about are Rank and World size this is eight and this will be whatever it is depending on the GPU uh that uh that this particular instantiation of the script runs on now here we make sure that according to the local rank we are setting the device to be Cuda colon and colon indicates which GPU to use if there are more than one gpus so depending on the local rank of this process it's going to use just the appropriate GPU so there's no collisions on which GPU is being used by which process and finally there's a Boolean variable that I like to create which is the DDP rank equ equal Z so the master process is arbitrarily process number zero and it does a lot of the printing logging checkpointing Etc and the other processes are thought of mostly as a compute processes that are assisting and so Master process zero will have some additional work to do all the other processes will uh will mostly just be doing forward backwards and if we're not using DDP and none of these variables are set we revert back to single GPU training so that means that we only have rank zero the world size is just one uh and and we are the master process and we try to autodetect the device and this is world as normal so so far all we've done is we've initialized DDP and uh in the case where we're running with torrun which we'll see in a bit there's going to be eight copies running in parallel each one of them will have a different Rank and now we have to make sure that everything happens uh correctly afterwards so the tricky thing with running multiple processes is you always have to imagine that there's going to be eight processes running in parallel so as you read the code now you have to imagine there's eight you know eight python interpreters running down these lines of code and the only difference between them is that they have a different DDP rank so they all come here they all pick the exact same seed they all make all of these calculations completely unaware of the other copies running roughly speaking right so they all make the exact same calculations and now we have to adjust these calculations to take into account that there's actually like a certain world size and certain ranks so in particular these micro batches and sequence lengths these are all just per GPU right so now there's going to be num processes of them running in parallel so we have to adjust this right because the grum steps now is going to be total B size divide B * T time U DDP R size because each um process will will do B * T and there's this many of them and so in addition to that we we want to make sure that this fits nicely into total batch size which for us it will because 16 * 124 * 8 8 gpus is 131 uh K and so 524288 this means that our gratum will be four with the current settings right so there's going to be 16 * 124 process on each GPU and then there's a GP pus so we're going to be doing 131,000 tokens in a single forward backward on the 8 gpus so we want to make sure that this fits nicely so that we can derive a nice gradient accumulation steps and uh yeah let's just adjust the comments here times uh DDP World size okay so each GPU calculates this now this is where we start to get run into issues right so we are each process is going to come by a print and they're all going to print so we're going to have eight copies of these prints so one way to deal with this is exactly this master process variable that we have so if Master process then guard this and that's just so that we just print this a single time because otherwise all the processes would have computed the exact same variables and there's no need to print this eight times um before getting into the data loader and we're going to have to refactor it obviously maybe at this point is uh we should do some prints and uh just take it out for a spin and exit at this point so import sis and S start exit and print IM GPU um DDP rank IM GPU DDP Rank and that um print by so uh so now let's try to run this and just see how this works so let's take it for a spin just so we see what it looks like so normally we use to launch python train gpd2 P like this now we're going to run with torch run and this is what it looks like so torch run Standalone number of processes for example is eight for us because we have eight gpus uh and then change of2 Pi so this is what the command would look like and torch run again we'll run eight of these so let's just see what happens so first it gets a little busy so there's a lot going on here so first of all there's some warnings from distributed and I don't actually know that these mean anything I think this is just like the code is setting up and the processes are coming online and we're seeing some preliminary failure to collect while the processes come up I'm not 100% sure about that but we start to then get into actual prints so all the processes went down and then the first print actually comes from process 5 uh just by chance and then it printed so process 5 basically got here first it said I'm process on GPU 5 buy and then this these prints come from the master process so process 5 just finished first for whatever reason it just depends on how the operating system scheduled the processes to run uh then gpu0 ended then GPU 3 and two and then uh probably process 5 or something like that has uh exited and and DDP really doesn't like that because we didn't properly dispose of uh the multi-gpus um setting and so process group has not been destroyed before we destruct uh so it really doesn't like that and in an actual application we would want to call destroy process group uh so that we clean up DDP properly and so it doesn't like that too much and then the rest of the gpus finish and that's it so basically we can't guarantee when these processes are running it's totally but they are running in parallel we don't want them to be printing um and next up let's erase this next up we want to make sure that when we create data loader light we need to now make it aware of this multi-process um setting because we don't want all the processes to be loading the exact same data we want every process to get its own chunk of data so that they're all working on different parts of the data set of course so let's adjust that so one particular particularly simple and a naive way to do this is we have to make sure that we pass in the rank and the size to the data loader and then when we come up here we see that we now take Rank and processes and we save them now the current position will not be zero uh because what we want is we want to stride out all the processes so one way to do this is we basically take S.B times salt. T and then multiply it by the process rank so proc process rank 0 will start at zero but process rank one now starts at B * T process rank two is starts at 2 * B * D Etc so that is the initialization now we still they still do this identically but now when we advance we don't Advance by B * T we advance by B * T times number of processes right so basically um the total number of tokens that we're um consuming is B * T * number processes and they all go off to a different Rank and the position has to advance by the entire chunk and then here B * T time uh s. num processes + one would be to exceed number of tokens then we're going to Loop and when we Loop we want to of course Loop in the exact same way so we sort of like reset back uh so this is the simplest change that I can uh find for kind of a very simple distributed data Lo light and um you can notice that if process rank is zero and non processes is one then uh the whole thing will be identical to what we had before but now we can have actually multiple processes uh running and this should work fine um so that's the data loader okay so next up once they've all initialized the data loader they come here and they all create a GPT model uh so we create eight GPT models on eight processes but because the seeds are fixed here they all create the same identical model they all move it to the device of their Rank and they all compile the model and because the models are identical there are eight identical compilations happening in parallel but that's okay now none of this uh changes because that is on a per step basis and we're currently working kind of within step because we need to um just uh all the all the changes we're making are kind of like a within step changes now the important thing here is when we construct the M model we actually have a bit of work to to do here get loits is deprecated so uh create model we need to actually wrap the model into the distributed data parallel container so um this is how we wrap the model into the DDP container and these are the docs for DDP and they're quite extensive and there's a lot of caveats and a lot of things to be careful with because everything complexifies times 10 when multiple processes are involved but roughly speaking this device IDs I believe has to be passed in now unfortunately the docs for what device IDs is is is extremely unclear uh so when you actually like come here this comment for what device IDs is is roughly nonsensical um but I'm pretty sure it's supposed to be the DDP local rank so not the DDP rank the local rank uh so this is what you pass in here this wraps the model and in particular what DDP does for you is in a forward pass it actually behaves identically so um my understanding of it is nothing should be changed in the forward pass but in the backward pass as you are doing the backward pass um in the simpl setting once the backp passes over on each independent GPU each independent GPU has the gradient for all the parameters and what DDP does for you is once the backward pass is over it will call what's called all reduce and it basically does an average across all the uh ranks of their gradients and and then it will deposit that average on every single rank so every sing Single rank will end up with the average on it and so basically that's the communication it just synchronizes and averages the gradients and that's what DDP offers you now DDP actually is a little bit more um it is a little bit more involved than that because as you are doing the backward pass through the layers of the Transformer it actually can dispatch Communications for the gradient while the backward pass is still happening so there's overlap of the uh communication of the gradient and the synchronization of them and uh the backward pass and uh this is just more efficient and um uh to do it that way so that's what DDP does for you um forward is unchanged and backward is mostly unchanged and we're tacking on this average as we'll see in a bit okay so now let's go to the uh optimization nothing here changes let's go to the optimization here the inner loop and think through the synchronization of uh these gradients in the DP so basically by default what happens as I mentioned is when you do l. backward here it will do the backward pass and then it will synchronize the gradients um the problem here is because of the gradient accumulation steps Loop here we don't actually want to do the synchronization after every single La step backward because we are just depositing gradients and we're doing that serially and we just want them adding up and we don't want to synchronize every single time that would be extremely wasteful so basically we want to add them up and then on the the very last uh it's only on the very last step when micro when micro step becomes gratak steps minus one only at that last step do we want to actually do the alberu uh to average up the gradients so to do that we come here and um the official sanctioned way by the way is to do this no sync context manager so pytorch says this is a context manager to disable gradient synchronization across DDP processes So within this context gradient will be accumulated and basically when you do no sync there will be no communication so they are telling us to do with DDP no sync uh do the gradient accumulation accumulate grats and then they are asking us to do DDP again with another input and that backward and I just really don't love this I I just really don't like it uh the fact that you have to copy paste your code here and use a context manager and this is just super ugly so when I went to this source code here you can see that when you enter you simply toggle this variable this require backward grat sync and this is uh being toggled around and changed and this is the variable that basically uh if you step through it is being toggled to determine if the gradient is going to be synchronized so I actually just kind of like to use that directly uh so instead what I like to do is the following right here before the L back backward if we are using the DDP then um then basically we only want to synchronize we only want this variable to be true when it is the final iteration in all the other iterations inside the micr steps we want to be false so I just toggle it like this so required backward graph sync should only turn on when the micro step is the last step and so I'm toggling this variable directly and I hope that that impacts last St backwards and this is a naughty thing to do because you know they could probably change the DDP and this variable will go away but for now I believe this this works and it allows me to avoid the use of context managers and code duplication I'm just toggling the variable and then Lop backward will not synchronize most of the steps and it will synchronize the very last step and so once this is over uh and we come out every single um rank will suddenly magically have the average of all the gradients that were stored on all the ranks so now we have to think through whether that is what we want and also um if this suffices and whether how it works with the loss and what is loss AUM so let's think through through that now and the problem I'm getting at is that we've averaged the gradients which is great but the loss AUM has not been impacted yet and the and this is outside of the DDP container so that is not being averaged um and so here when when we are printing Los AUM well presumably we're only going to be printing on the master process uh rank zero and it's just going to be printing the losses that it saw on its process but instead we want it to print the loss over all the processes and the average of that loss because we did average of gradients so we want the average of loss as well so simply here after this uh this is the code that I've used in the past um and instead of LF we want Lum so if DDP again then this is a p torch distributed I import it where do I import it uh oh gosh so this file is starting to get out of control huh so if uh so import torch. distributed as dist so dist. ALU and we're doing the average on Lum and so this lakum tensor exists on all the ranks when we call all use of average it creates the average of those numbers and it deposits that average on all the ranks so all the ranks after this um call will now contain L AUM uh averaged up and so when we print here on the master process the L AUM is identical in all the other ranks as well so here if Master process oops we want to print like this okay and finally we have to be careful because we're not processing even more tokens so times DDP World size that's number of tokens that we've processed up above and everything else should be fine uh the only other thing to be careful with is as I mentioned you want to destroy the process group so that we are nice to nickel and it's not going to uh to uh to DDP and it's not going to complain to us uh when we exit here so that should be it let's try to take it for a spin okay so I launched the script and it should be uh printing here imminently we're now training with 8 gpus at the same time so the gradient accumulation steps is not 32 it is now divide 8 and it's just four uh so um otherwise this is what the optimization now looks like and wow we're going really fast so we're processing 1.5 million tokens uh per second now so these are some serious numbers and the tiny shakespare data set is so tiny that we're just doing like so many Epoch over it most likely but this is roughly what looks like um one thing that I had to fix by the way is that this was model. configure optimizers which Now doesn't work because model now is a DDP model so instead this has to become raw model. configure optimizers where raw model is something I create here so right after I wrap the model into DDP uh I have to create the raw model which in the case of DDP is a model. module is where it stores the raw and then module of gpt2 as we have it which contains the uh configure optimizers function that we want to call so that's one thing that I have to fix otherwise this seems to run now one thing you'll notice is that when you actually compare this run and the numbers in it to the just running a single GPU you'll notice that this is single GPU run with 32 gratum the numbers won't exactly match up and uh that's kind of a boring reason for why that happens uh the reason for that is that in the data loader we're basically just iterating through batches and slightly different way because now we're looking for an entire page of data and if that page uh for all the gpus if that chunk exceeds the number of tokens we just Loop and so actually the single GPU and the H GPU process will end up um resetting in a slightly different Manner and so our batches are slightly different and so we get slightly different numbers but one way to convince yourself that this is okay it just make the total batch size much smaller and the b and a t and then um so I think I used uh 4 * 124 * 8 so I used 32768 as a total patch size and then um so I made sure that the single GPU will do eight creting accumulation steps and then the multi-gpu and then you're reducing the boundary effects of the data loader and you'll see that the numbers match up so long story short we're now going really really fast the optimization is mostly consistent with gpt2 and three hyper parameters and uh we have outgrown our tiny Shakespeare file and we want to upgrade it so let's move to next to that next

so let's now

03:10:21 datasets used in GPT-2, GPT-3, FineWeb (EDU)

take a look at what data sets were used by gpt2 and gpt3 so gbt2 used this web Text data set that was never released um there's an attempt at reproducing it called open web text uh so basically roughly speaking what they say here in the paper is that they scraped all outbound links from Reddit and then uh with at least three Karma and that was kind of like their starting point and they collected all the web P all the web pages and all the text in them and so this was 45 million links and this ended up being 40 GB of text so uh so that's roughly what gpt2 says about its data set so it's basically outbound links from Reddit now when we go over to gpt3 there's a training data set section and that's where they start to talk about um common coll which is a lot more uh used actually I think even gpt2 talked about common coll um but basically it's not a very high quality data set all by itself because it is extremely noisy this is a completely random subset of the internet and it's much worse than you think so people go into Great Lengths to filter common craw because there's good stuff in it but most of it is just like ad spam random tables and numbers and stock tickers and uh it's just total mess so that's why people like to train on these data mixtures that they curate and uh are careful with so a large chunk of these data mixtures typically will be common C like for example 50% of the tokens will be comic but then here in gpt3 they're also using web text to from before so that's Reddit outbound but they're also adding for example books and they're adding Wikipedia there's many other things you can decide to add now this data set for gpt3 was also never released so today some of the data sets that I'm familiar with that are quite good and would be representative of something along these lines are number one the red pajama data set or more specifically for example the slim pajama subset of the red pajama data set which is a cleaned and D duplicated version of it and just to give you a sense again it's a bunch of common crawl um C4 which is also as far as I know more common craw but processed differently and then we have GitHub books archive Wikipedia stack exchange these are the kinds of data sets that would go into these data mixtures now specifically the one that I like that came out recently is called Fine web data set uh so this is an attempt to basically collect really high quality common coll data and filter it in this case to 15 trillion tokens and then in addition to that more recently huggingface released this fine web edu subset which is 1.3 trillion of educational and 5.4 trillion of high educational content so basically they're trying to filter common C to very high quality educational subsets and uh this is the one that we will use there's a long uh web page here on fine web and they go into a ton of detail about how they process the data which is really fascinating reading by the way and I would definitely recommend if you're interested into Data mixtures and so on and how data gets processed at these scales a look at this uh page and more specifically we'll be working with the fine web edu I think and it's basically educational content from the internet uh they show that training on educational content in in their metrics um uh works really really well and we're going to use this sample 10 billion tokens subsample of it because we're not going to be training on trillions of tokens uh we're just going to train on uh 10 billion sample of the fine web edu because empirically in my previous few experiments this actually suffices to really get close to gpt2 Performance and it's um simple enough to work with and so let's work with the sample 10 uh BT so our goal will be to download it process it and make sure that our data loader can work with it so let's get to that okay so I introduced another um file here that will basically download Fine web edu from huging face data sets it will pre-process and pre- tokenize all of the data and it will save data shards to a uh folder on um local disk and so while this is running uh just wanted to briefly mention that you can kind of look through the data set viewer here just to get a sense of what's in here and it's kind of interesting I mean it's a it basically looks like it's working fairly well like it's talking about nuclear energy in France it's talking about Mexican America some mac PJs Etc so actually it seems like their filters are working pretty well uh the filters here by the way were applied automatically using um llama 370b I believe and so uh basically llms are judging which content is educational and that ends up making it through the filter uh so that's pretty cool now in terms of the script itself I'm not going to go through the full script because it's not as interesting and not as llm Centric but when you run this basically number one we're going to load the data set uh which this is all huging face code running this you're going to need to uh pip install data sets um so it's downloading the data set then it is tokenizing all of the documents inside this data set now when we tokenize the documents you'll notice that um to tokenize a single document uh we first start the tokens with the end of text token and this is a special token in the gpt2 tokenizer as you know so 50256 is the ID of the end of text and this is what begins a document even though it's called end of text but this is uh the first token that begins a document then we extend with all of the tokens of that document then we create a numpy array out of that we make sure that all the tokens are between oh okay let me debug this okay so apologies for that uh it just had to do with me using a float division in Python it must be integer division so that this is an INT and everything is nice um okay but basically the tokenization here is relatively straightforward returns tokens in mp. un6 uh we're using .16 to save a little bit of space because 2 to the 16us 1 is 65,000 so the gpt2 max token ID is well below that and then here there's a bunch of multiprocessing code and it's honestly not that exciting so I'm not going to step through it but we're loading the data set we're tokenizing it and we're saving everything to shards and the shards are numpy files uh so just storing a numpy array and uh which is very very similar to torch tensors and the first Shard 0000 is a Val a validation Shard and all the other shards are uh training shards and as I mentioned they all have

100 million tokens in them exactly um and and that just makes it easier to work with as to Shard the files because if we just have a single massive file sometimes they can be hard to work with on the disk and so sharting it is just kind of um nicer from that perspective and uh yeah so we'll just let this run this will be probably um 30ish minutes or so and then we're going to come back to actually train on this data and we're going to be actually doing some legit pre-training in this case this is a good data set we're doing lots of tokens per second we have 8 gpus the code is ready and so we're actually going to be doing a serious training run so let's get P it back in a bit okay so we're back so uh if we LS edu fine web we see that there's now 100 charts in it um and that makes sense because each chart is 100 million tokens so 100 charts of that is 10 billion tokens in total now swinging over to the main file I made some adjustments to

our data loader again and that's because we're not running with uh Shakespeare anymore we want to use the fine web shards and so you'll see some code here that additionally basically can load these shards uh we load the um un6 numpy file we convert it to a torch. long tensor which is what a lot of the layers up top expect by default and then here we're just enumerating all the shards I also added a split to data load of light so we can uh load the split train but also the split Val uh the zero split and then we can load the shards and then here we also have not just the current position now but also the current Shard so we have a position inside A Shard and then when we uh run out of tokens in A Single Shard we first Advance The Shard and loop if we need to and then we get the tokens and readjust the position so this data loader will now iterate all the shards as well so I Chang that and then the other thing that

I did while uh the data was processing is our train loader now has split train of course and down here I set up some I set up some numbers so we are doing 2 to the 9 uh tokens per uh per um per step and we want to do roughly 10 billion tokens um because that's how many unique tokens we have so if we did 10 billion tokens then divide that by 29 we see that this is 1973 steps so that's where that's from and then the GPT three paper says that they warm up the learning rate over 375 million tokens so I came here and 375 E6 tokens divide uh 2 to the 19 is 715 steps so that's why warm-up steps is set to 715 so this will exactly match um the warm-up schedule that gpt3 used and I think 715 by the way is very uh mild and this could be made significantly more aggressive probably even like 100 is good enough um but it's okay let's leave it for now so that we have the exact hyper parameters of gpt3 so I fix that and then um that's pretty much it we can we can run so we have our script here and we can launch and actually sorry let me do one more thing excuse me for my GPU I can actually fit more batch size and I believe I can fat I can fit 60 4 on my GPU as a micro bash size so let me try that I could be misremembering but that means 64 * 124 per GPU and then we have a gpus so that means we would not even be doing gradient accumulation if this fits because uh this just multi multiplies out to uh the full total bat size so no gradient accumulation and that would run pretty quickly if that fits let's go let's go I mean if this works then this is basically a serious pre-training run um we're not logging we're not evaluating the validation split we're not running any evaluations yet so it's not we haven't crossed our te's and dotted our eyes but uh if we let this run for a while we're going to actually get a pretty good model and the model that might even be on par with or better than gpt2 124 M okay so it looks like everything is going great we're processing 1.5 million tokens per second uh everything here looks good we're doing 330 milliseconds per iteration and we have to do a total of uh where are we printing that 1973 so 19073 times 0.33 is this many seconds this many minutes so this will run for 1.7 hours uh so one and a half hour run uh like this and uh we don't even have to use gradient accumulation which is nice and you might not have that luxury in your GPU in that case just start decreasing the batch size until things fit but keep it to nice numbers um so that's pretty exciting we're currently warming up the learning rate so you see that it's still very low one4 so this will ramp up over the next few steps all the way to 6 e Nega uh 4 here very cool so now what I'd like to do is uh let's cross the T and do our eyes let's evaluate on the validation split and let's try to figure out how we can run evals how we can do logging how we can visualize our losses and all the good stuff so let's get to that before we actually do the run okay so I've

03:23:10 validation data split, validation loss, sampling revive

adjusted the code so that we're evaluating on the validation split so creating the Val loader just by passing in Split equals Val that will basically create a data loader just for the uh validation Shard um the other thing I did is in the data loader I introduced a new function reset which is called at init and it basically resets the data loader and that is very useful because when we come to the main training Loop now so this is the code that I've added and basically every 100th iteration including the zeroth iteration we put the model into evaluation mode we reset the Val loader and then um no gradients involved we're going to basically accumulate the gradients over say 20 steps and then average it all up and print out the validation loss and so that basically is the exact same logic as the training Loop roughly but there's no loss that backward it's only inference we're just measuring the loss we're adding it up everything else otherwise applies and is exactly as we've seen it before and so this will print the validation laws um every 100th iteration including on the very first iteration uh so that's nice that will tell us some amount some a little bit about how much we're overfitting that said like uh we have roughly Infinity data so we're mostly expecting our train and Val loss to be about the same but the other reason I'm kind of interested in this is because we can take the GPT 2124m as openi released it we can initialize from it and we can basically see what kind of loss it achieves on the validation loss as well and that gives us kind of an indication as to uh how much that model would generalize to 124 M but it's not an sorry to fine web edu validation split that said it's not a super fair comparison to gpt2 because it was trained on a very different data distribution but it's still kind of like an interesting data point and in any case you would always want to have a validation split in a training run like this so that you can make sure that you are not um overfitting and this is especially a concern if we were to make more Epoch in our training data um so for example right now we're just doing a single Epoch but if we get to a point where we want to train on 10 epochs or something like that we would be really careful with maybe we are memorizing that data too much if we have a big enough model and our validation split would be one way to tell whether that is happening okay and in addition to that if you remember at bottom of our script we had all of this orphaned code for sampling from way back when so I deleted that code and I moved it up um to here so once in a while we simply value validation once in a while we sample we generate samples and then uh we do that only every 100 steps and we train on every single step so that's how I have a structure right now and I've been running this for 10,000 iterations so here are some samples on neration 1,000 um hello I'm a language model and I'm not able to get more creative I'm a language model and languages file you're learning about here is or is the beginning of a computer okay so this is all like pretty uh this is still a garble uh but we're only at ration 1,000 and we've only just barely reached maximum learning rate uh so this is still learning uh we're about to get some more samples coming up in 1,00 okay um okay this is you know the model is still is still a young baby okay so uh basically all of this sampling code that I've put here everything should be familiar with to you and came from before the only thing that I did is I created a generator object in pytorch so that I have a direct control over the sampling of the random numbers don't because I don't want to impact the RNG state of the random number generator that is the global one used for training I want this to be completely outside of the training Loop and so I'm using a special sampling RNG and then I make sure to seed it that every single rank has a different seed and then I pass in here where we sort of consumer in the numbers in multinomial where the sampling happens I make sure to pass in the generator object there otherwise this is identical uh now the other thing is um you'll notice that we're running a bit slower that's because I actually had to disable torch. compile to get this to sample and um so we're running a bit slower so for some reason it works with no torch compile but when I torch compile my model I get a really scary error from pytorch and I have no idea how to resolve it right now so probably by the time you see this code released or something like that maybe it's fixed but for now I'm just going to do end false um and

I'm going to bring back toor compile and you're not going to get samples and I I think I'll fix this later uh by the way um I will be releasing all this code and actually I've been very careful about making get commits every time we add something and so I'm going to release the entire repo that starts completely from scratch all the way to uh now and after this as well and so everything should be exactly documented in the git commit history um um and so I think that will be nice so hopefully by the time you go to GitHub uh this is removed and it's working and I will have fixed the bug okay so I have

03:28:23 evaluation: HellaSwag, starting the run

the optimization running here and it's stepping and we're on step 6,000 or so so we're about 30% through training now while this is training I would like to introduce one evaluation that we're going to use to supplement the validation set and that is the H swag eval so hos swag comes from this paper back in 2019 so it's a 5-year-old eval now and the way H swag works is there is basically a sentence completion data set so it's a multiple choice for every one of these questions we have uh basically a shared context like a woman is outside with a bucket and a dog the dog is running around trying to avoid bath she a Rises the bucket off with soap and blow dry the dog's head B uses a hose to keep it from getting soapy C gets the dog wet and it runs away again or D gets into a bathtub with the dog and so basically the idea is that these multiple choice are constructed so that one of them is a natural continuation of the um sentence and the others are not and uh the others might not make sense like uses the host to keep it from getting soaped that makes no sense and so what happens is that models that are not trained very well are not able to tell these apart but models that have a lot of World Knowledge and can tell uh which um and can tell a lot about the world will be able to create these completions and these sentences are sourced from activity net and from Wiki how and at the bottom of the uh paper there's kind of like a cool chart of the kinds of domains in Wiki house so there's a lot of sentences from computers and electronics and Homes and Garden and it has kind of a broad coverage of the kinds of things you need to know about the world in order to find the most likely completion and um the identity of that of that completion one more thing that's kind of interesting about H swag is the way it was constructed is that the incorrect um options are deliberately um adversarially sourced so they're not just random sentences they're actually sentences generated by language models and they're generated in such a way that language models basically find them difficult but humans find them easy and so they mentioned that humans have a 95% accuracy on this set but at the time the state-of-the-art language models had only 48% and so at the time this was a good Benchmark now you can read the details of this paper to to learn more um the thing to point out though is that this is 5 years ago and since then what happened to H swag is that it's been totally just uh um solved and so now the language models here are 96% so basically the 4% the last 4% is probably errors in the data set or the questions are really really hard and so basically this data set is kind of crushed with respect to language models but back then the best language model was only at about 50% uh but this is how far things got but still the the reason people like H swag and it's not used by the way in gpt2 but in gpt3 there is H swag eval and lots of people use H swag and so for gpt3 we have results here that are cited so we know what percent accuracies gpt3 um attains at all these different model checkpoints for H swag eval and the reason people like it is because H swag is a smooth eval and it is an eval that offers quote unquote early signal uh so early signal means that even small language models are going to start at the random chance of 25% but they're going to slowly improve and you're going to see 25 26 27 Etc and uh you can see slow Improvement even when the models are very small and it's very early so it's smooth it has early signal and um it's been around for a long time so that's why people kind of like this eval uh now the way that we're going to evaluate this is as follows as I mentioned we have a shared context and this is kind of like a multiple choice task but instead of giving the model a multiple choice question and asking it for A B C or D uh we can't do that because these models when they are so small as we are seeing here the models can't actually do multiple choice they don't understand the concept of associating a label to one of the options of multiple choice uh they don't understand that so we have to give it to them in a native form and the native form is a token completion so here's what we do we construct a batch of four rows and uh T tokens whatever that t happens to be then the shared context that is basically the context for the for choices the tokens of that are shared across all of the rows and then we have the four options so we kind of like lay them out and then only one of the options is correct in this case label three option three and so um this is the correct option and option one two and for are incorrect now these options might be of different lengths so what we do is we sort of like take the longest length and that's the size of the batch B BYT and then some of these uh here are going to be pded Dimensions so they're going to be unused and so we need the tokens we need the correct label and we need a mask that tells us which tokens are active and the mask is then zero for these uh padded areas so that's how we construct these batches and then in order to get the language model to predict A B C or D the way this works is basically we're just going to look at the tokens their probabilities and we're going to pick the option that gets the lowest or the highest average probability for the token so for the tokens because that is the most likely completion according to the language model so we're just going to look at the um probabilities here and average them up across the options and pick the one with the highest probability roughly speaking so this is how we're going to do H swag um and this is I believe also how uh gpt3 did it um this is how gpt3 did it as far as I know but you should note that some of the other evals where you might see H swag may not do it this way they may do it in a multiple choice format where you sort of uh give the the context a single time and then the four completions and so the model is able to see all the four options before it picks the best possible option and that's actually an easier task for a model because you get to see the other options when you're picking your choice um but unfortunately models at our size can't do that only models at a bigger size are able to do that and so our models are actually slightly handicapped in this way that they are not going to see the other options they're only going to see one option at a time and they just have to assign probabilities and the correct option has to win out in this metric all right so let's now implement this very briefly and incorporate it into our script okay so what I've done here is I've introduced a new file called hell swag. py that you can take a look into and I'm not going to to step through all of it because uh this is not exactly like deep code deep code it's kind of like a little bit tedious honestly because what's happening is I'm downloading hsac from GitHub and I'm rendering all of its examples and there are a total of 10,000 examples I am rendering them into this format um and so here at the end of this render example function you can see that I'm returning the tokens uh the tokens of this um 4xt uh array of Tokens The Mask which tells us which parts are the options and everything else is zero and the label that is the correct label and so that allows us to then iterate the examples and render them and I have an evaluate function here which can load a um gpt2 from huging face and it runs the eval here um and it basically just calculates uh just as I described it predicts the option that has the lowest or the highest prob ility and the way to do that actually is we can basically evaluate the cross entropy loss so we're basically evaluating the loss of predicting the next token in a sequence and then we're looking at the row that has the lowest average loss and that's the uh option that we pick as the prediction and then we do some stats and prints and stuff like that so that is a way to evaluate L swag now if you go up here I'm showing that for GPT 2124m if you run this script you're going to see that H swag gets 29.5% um so that's the performance we get here now remember that random Chan is 25% so we haven't gone too far and gpt2 XL which is the biggest the gpt2 gets all the way up to 49% roughly so uh these are pretty low values considering that today's state-ofthe-art is more like 95% uh so these are definitely older models by now and then there's one more thing called Uther harness which is a very piece of infrastructure for running evals for language models and they get slightly different numbers and I'm not 100% sure what the discrepancy is for these um it could be that they actually do the multiple choice uh instead of just the completions and that could be the um uh the discrepancy but I'm not 100% sure about that i' have to take a look but for now our script reports 2955 and so that is the number that we'd like to beat if we are training a GPD 2124m from scratch and ourselves um so now I'm going to go into actually incorporating this eval into our main training script and um and basically because we want to evaluate it in a periodic manner so that we can track H swag and how it evolves over time and see when when and if we cross uh this 2955 um sort of region so let's now walk through some of the changes to train gpt2 thatp the first thing I did here is I actually made use compile optional kind of and I disabled it by default and the problem with that is the problem with compile is that unfortunately it does make our code faster but it actually breaks the evaluation code and the sampling code it gives me a very gnarly message and I don't know why so hopefully by the time you get to the codebase when I put it up on GitHub uh we're going to fix that by then but for now I'm running without torch compile which is why you see this be a bit slower so we're running without torch compile I also create cre a log directory log where we can place our log.txt which will record the train loss validation loss and the H swag accuracies so a very simple text file and we're going to uh open for writing so that it sort of starts empty and then we're going to append to it I created a simple variable that um helps tell us when we have a last step and then basically periodically inside this Loop every 250th iteration or at the last step we're going to evaluate the validation loss and then every 250th iteration um we are going to evaluate H swag but only if we are not using compile because compile breaks it so I'm going to come back to this code for evaluating H swag in a second and then every 250th iteration as well we're also going to sample from the model and so you should recognize this as our ancient code from way back when we started the video and we're just sampling from the model and then finally here um these are if we're not after we validate sample and evaluate hell swag we actually do a training step here and so this is one step of uh training and you should be pretty familiar with all of what this does and at the end here once we get our training laws we write it to the file so the only thing that changed that I really added is this entire section for H swag eval and the way this works is I'm trying to get all the gpus to collaborate on the H swag and so we're iterating all the examples and then each process only picks the examples that assigned to it so we sort of take I and moded by the world size and we have to make it equal to rank otherwise we continue and then we render an example put it on the GPU we get the low jits then I create a helper function that helps us basically predict the option with the lowest loss so this comes here the prediction and then if it's correct we sort of keep count and then if multiple processes were collaborating on all this then we need to synchronize their stats and so the way one way to do that is to package up our statistics here into tensors which we can then call this. alberon and sum and then here we sort of um unwrap them from tensors so that we just have ins and then here the master process will print and log the hellis swag accuracy so that's kind of the that's kind of it and that's what I'm running right here so you see this optimization here and uh we just had a generation and this is Step 10,000 out of about 20,000 right so we are halfway done and these are the kinds of samples that uh we are getting at this stage so let's take a look hello I'm a language model so I'd like to use it to generate some kinds of output hello I'm a language model and I'm a developer for a lot of companies Al language model uh let's see if I can find fun one um I don't know you can go through this yourself but certainly the predictions are getting less and less random uh it seems like the model is a little bit more self-aware and using language uh that is a bit more uh specific to it being language model hello I'm a language model and like how the language is used to communicate I'm a language model and I'm going to be speaking English and German okay I don't know so let's just wait until this optimization finishes and uh we'll see what kind of samples we get and we're also going to look at the train Val and the hway accuracy and see how we're doing with respect to

03:43:05 SECTION 4: results in the morning! GPT-2, GPT-3 repro

gpt2 okay good morning so focusing For a Moment On The jupyter Notebook here on the right I created a new cell that basically allows us to visualize the the train Val and Hela and um the hel score and you can step through this it basically like parses the log file that we are writing and um a lot of this is just like boring ma plot lip code but basically this is what our optimization looks like so we ran for 19,731 billion tokens which is whoops oh my gosh which is one Epoch of the sample 10B of webd on the left we have the loss and the in blue we have the training loss in Orange we have the validation loss and red as a horizontal line we have the opening IG gpt2 124 M model checkpoint when it's just evaluated on the validation set of um of this fine web edu uh so you can see that we are surpassing this orange is below the red so we're surpassing the validation set of this data set and like I mentioned the data set distribution is very different from what gpt2 trained on so this is not an exactly fair comparison but it's a good cross check uh to uh to look at now we would ideally like something that is withheld and comparable and somewhat standard um and so for us that is helis swag and so on here we see the H swag progress we made from 25% all the way here in red we see the open gpt2 124 M model in red so it achieves this h bag here and the the gpt3 model 124 M which was trained on 300 billion tokens achieves green so that's over here so you see that we basically surpassed the gbt2 24m uh model right here uh which is uh really nice now interestingly we were able to do so with only training on 10 billion tokens while gpt2 was trained on 100 billion tokens so uh for some reason we were able to get away with significantly fewer tokens for training there are many possibilities to as to why we could match or surpass this accuracy um with only 10 million training so number one um it could be that opening gbt2 was trained on a much wider data distribution so in particular fine web edu is all English it's not multilingual and there's not that much math and code um and so math and code and multilingual could have been stealing capacity from the original gpt2 model and um basically that could be partially the reason why uh this is not working out there's many other reasons um so for example the H swag eval is fairly old uh maybe 5 years or so it is possible that aspects of H swag in some way or even identically have made it into the training Set uh of fine web we don't know for sure but if that was the case then we are basically looking at the training curve instead of the validation curve so long story short this is not a perfect eval and there's some caveats here uh but at least we have some confidence that that we're not doing something completely wrong and um and uh it's probably the case that when people try to create these data sets they try to make sure that test sets that are very common are not part of the training set for example uh when hugging face created the fine web BDU they use H swag as an eval so I would hope that they make sure that they D duplicate and that there's no hella swag in the training set but we can't be sure uh the other thing I wanted to address briefly is look at this loss curve this looks really this looks really wrong here I don't actually know 100% what this is and I suspect it's because the uh 10 billion sample of fine web edu was not properly shuffled um and there's some issue here uh with the data that I don't fully understand yet and there's some weird periodicity to it um and because we are in a very lazy way sort of serializing all the tokens and just iterating all them from scratch without doing any permutation or any random sampling ourselves I think we're inheriting some of the ordering that they have in the data set so uh this is not ideal but hopefully by the time you get to this repo uh some of these things by the way will hopefully be fixed and I will release this build n GPT repo and right now it looks a little ugly and preliminary uh so hopefully by the time you get here it's nicer but down here I'm going to show aada and I'm going to talk about about some of the things that happened after the video and I expect that we will have fixed uh the small issue uh but for now basically this shows that uh our training is not uh completely wrong and it shows that uh we're able to surpass the accuracy with only 10x the token budget um and possibly it could be also that the data set may have improved so uh the original uh gpt2 data set was web text it's possible that not a lot of care and attention went into the data set this was very early in llms whereas now there's a lot more scrutiny on good practices around uh D duplication filtering uh quality filtering and so on and it's possible that the data that we're training on is just of higher quality per token and that could be giving us a boost as well so a number of cave has to think about but for now uh we're pretty happy with this um and yeah now the next thing I was interested in is as you see it's a morning now so there was an overnight and I wanted to basically see how far I could push the result so uh to do an overnight run I basically did instead of one Epoch which took roughly two hours I just did a times four so that that would take eight hours while I was sleeping and so we did four Epoch or roughly 40 billion uh tokens of training and I was trying to see how far we could get um and so this was the only change and I reran the script and when I point uh and read the log file at uh at the 40b uh this is what the curve look like okay so to narrate this number one we are seeing this issue here here with the periodicity through the different Epoch and something really weird with the fine web edu data set and that is to be determined uh but otherwise we are seeing that the H swag actually went up by a lot and we almost we almost made it uh to the GPT 324m accuracy uh up here uh but not quite so uh it's too bad that I didn't sleep slightly longer um and uh I think if this was an uh five Epoch run we may have gotten here now one thing to point out is that if you're doing multi Epoch runs uh we're not actually being very careful in our data loader and we're not um I this data loader goes through the data in exactly the same format and exactly the same order and this is kind of suboptimal and you would want to look into extensions where you actually permute the data uh randomly you permute the documents around in Every Single Shard on every single new Epoch um and po even permute the shards and that would go a long way into decreasing the pricity and it's also better for the optimization so that you're not seeing things ident in the identical format and you're introducing some of the some uh Randomness in how the documents follow each other because you have to remember that in every single row these documents follow each other and then there's the end of text token and then the next document so the documents are currently glued together in the exact same identical manner but we actually want to break break up the documents and shuffle them around because the order of the documents shouldn't matter and they shouldn't um basically we want to break up that dependence because it's a kind of a spous correlation and so our data lad is not currently doing that and that's one Improvement uh you could think of making um the other thing to point out is we're almost matching gpt3 accuracy with only 40 billion tokens gpt3 trained on 300 billion tokens so again we're seeing about a 10x um Improvement here with respect to learning efficiency uh the other thing I wanted to and I don't actually know exactly what to attribute this to other than some of the things that I already mentioned previously for the previous run uh the other thing I wanted to briefly mention is uh the max LR here I saw some people already play with this a little bit in a previous related repository um and it turns out that you can actually almost like three xas so it's possible that the maximum learning rate can be a lot higher and for some reason the gpt3 hyper parameters that we are inheriting are actually extremely conservative and you can actually get away with a Higher

Learning rate and it would train faster so a lot of these hyper parameters um are quite tunable and feel free to play with them and they're probably not set precisely correctly and um it's possible that you can get away with doing this basically and if you wanted to exactly be faithful to gpt3 you would also want to make the following difference you'd want to come here and the sequence length of gpt3 is 2x it's 20 48 instead of 1,24 so you would come here change this to 248 for T and then if you want the exact same number of tokens uh half a million per iteration or per step you want to then decrease this to 32 so they still multiply to half a mil so that would give your model sequence length equal to that of gpt3 and in that case basically the um the models would be roughly identical as far as I'm as far as I'm aware because again gpt2 and gpt3 are very very similar models now we can also look at some of the samples here from the model that was trained overnight so this is the optimization and you see that here we stepped all the way to 76290 also or so and these are the hos mag we achieved was 33.2 4 and these are some of the samples from the model and you can see that if you read through this and pause the video briefly you can see that they are a lot more coherent uh so um and they're actually addressing the fact that it's a language model almost so uh hello I'm a language model and I try to be as accurate as possible um I'm a language model not a programming language I know how to communicate uh I use Python um I don't know if you pause this and look at it and then compare it to the one to the model that was only trained for 10 billion uh you will see that these are a lot more coherent and you can play with this uh yourself one more thing I added to The Code by the way is this chunk of code here so basically right after we evaluate the validation loss if we are the master process in addition to logging the validation loss every 5,000 steps we're also going to save the checkpoint which is really just the state dictionary of the model and so checkpointing is nice just because uh you can save the model and later you can uh use it in some way if you wanted to resume the optimiz ation then in addition to saving the model we have to also save the optimizer State dict because remember that the optimizer has a few additional buffers because of adom so it's got the m and V and uh you need to also resume the optimizer properly you have to be careful with your RNG seeds uh random number generators and so on so if you wanted to exactly be able to resume optimization you have to think through the state of the of the training process but if you just want to save the model this is how you would do it and one one nice reason why you might want to do this is because you may want to evaluate the model a lot more carefully so here we are only kind of like winging the hell swag eval but you may want to use something um nicer like for example the Luther uh Luther evaluation hardness evaluation hardness hardness um so this is a way to also evaluate language models and um so it's possible that um you may want to use basically different infrastructure to more thoroughly evaluate the models on different um evaluations and compare it to the opening gbt2 model on many other um tasks like for example that involve math code or different languages and so on so this is a nice functionality to have as well um and then the other thing I wanted to mention is that everything we've built here this is only the pre-training step so um the GPT here is a it dreams documents it just predicts the next to you can't talk to it like you can talk to chat GPT uh chat GPT if you wanted to talk to the model we have to fine-tune it into the chat format and it's not actually like that complicated if you're looking at supervised fine-tuning or sft really what that means is we're just swapping out a data set into a data set that is a lot more conversational and there's a user assistant user assistant kind of structure and we just fine-tune on it and then we um we basically fill in the user tokens and we sample the assistant tokens it's not a lot more deeper than that uh but basically we swap out the data set and continue training uh but for now we're going to stop at uh pre-training one more thing

03:56:21 shoutout to llm.c, equivalent but faster code in raw C/CUDA

that I wanted to briefly show you is that of course what we've built up today was building towards nanog GPT which is this repository from earlier uh but also there's actually another nanog GPT implementation and it's hiding in a more recent project that I've been working on called llm Doc and lm. C is a pure Cuda implementation of gpt2 or gpt3 training and it just directly uses uh Cuda and is written as Cuda now the nanog gbt here acts as reference code in pytorch to the C implementation so we're trying to exactly match up the two but we're hoping that the C Cuda is faster and of course currently that seems to be the case um because it is a direct optimized implementation so train gpt2 Pi in LL M.C is basically the nanog GPT and when you scroll through this file you'll find a lot of things that very much look like um things that we've built up in this lecture and then when you look at train gpt2 docu uh this is the C Cuda implementation so there's a lot of MPI nickel GPU Cuda cc++ and you have to be familiar with that but uh um when this is built up we can actually run the two side by side and they're going to produce the exact same results but lm. C actually runs faster so let's see that so on the left I have pytorch a nanog GPT looking thing on the right I have the llmc call and here I'm going to launch the two both of these are going to be running on a single GPU and here I'm putting the lm. C on GPU 1 and this one will grab uh gpu0 by default and then we can see here that lm. c compiled and then allocate space and it's stepping so basically uh meanwhile P torch is still compiling because torch compile is a bit slower here than the lm. C nbcc Cuda compile and so this program has already started running and uh we're still waiting here for torch compile now of course uh this is a very specific implementation to gpt2 and 3 a pytorch is a very general neural network framework so they're not exactly comparable but if you're only interested in training gpt2 and 3 lm. C is very fast it takes less space it's faster to start and it's faster per step and so P started to Stepping here and as you can see we're running at about 223,000 tokens per second here and about 185,000 tokens per second here um so quite a bit slower but I don't have full confidence that I exactly squeezed out all the juice from the pytorch implementation but the important thing here is notice that if I Aline up the steps you will see that the losses and Norms that are printed between these two are identical so on the left we have the pie torch and on the right this C implementation and they're the same except this one runs faster uh so that's kind of I wanted to show you also briefly lm. C and this is a parallel implementation and it's also something that you may want to uh play with or look at and um it's kind of interesting

03:59:39 summary, phew, build-nanogpt github repo

okay so at this point I should probably start wrapping up the video because I think it's getting way longer than I anticipated uh but we did Cover a lot of ground and we built everything from scratch so as a brief summary we were looking at the gpt2 and GPT 3 papers we were looking at how you set up these training runs uh and all the considerations involved we wrote everything from scratch and then we saw that over the duration of either a 2-hour training run or an overnight run we can actually match the 124 million parameter checkpoints of gbt2 and gpt3 uh to a very large extent um in principle the code that we wrote would be able to train even bigger models if you have the patients or the Computing resources uh and so you could potentially think about training some of the bigger checkpoints as well um there are a few remaining issues to address what's happening with the loss here which I suspect has to do with the fine web edu data sampling uh why can't we turn on Torch compile uh it currently breaks generation and H swag what's up with that in the data loader we should probably be permuting our data when we reach boundaries so there's a few more issues like that and I expect to be documenting some of those over time in the uh build n GPT repository here which I'm going to be releasing with this video if you have any questions or like to talk about anything that we covered please go to discussions tab uh so we can talk here uh or please go to issues or pull request pull requests um depending on what you'd like to contribute or also have a look at the uh Zero to Hero Discord and uh I'm going to be hanging out here on N GPT um otherwise for now I'm pretty happy about where we got um and I hope you enjoyed the video and I will see you later