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