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