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