Vid2Doc: LLaMA explained: KV-Cache, Rotary Positional Embedding, RMS Norm, Grouped Query Attention, SwiGLU

https://www.youtube.com/watch?v=Mn_9W1nCFLo

Section 1

Time Range: 0 s - 63.12 s

Hello guys, welcome to my new video about Lama. In this video we will be seeing what is Lama, how it is made, how it is structurally different from the transformer, and we will be building each block that makes up Lama. So I will not only explain you concept wise what is each block doing, but we will also explore it from the mathematical point of view and also from the coding point of view so that we can unify theory with practice. I can guarantee that if you watch this video you will have a deep understanding of what makes Lama the model it is. So you will not only understand how the blocks interact with each other, but how they function and why we needed this block in the first place. In this video we will be reviewing a lot of topics, so we will start from the architectural differences between the Vanilla transformer and the Lama model. We will be watching what is the new normalization, the RMS normalization, rotary positional embedding,

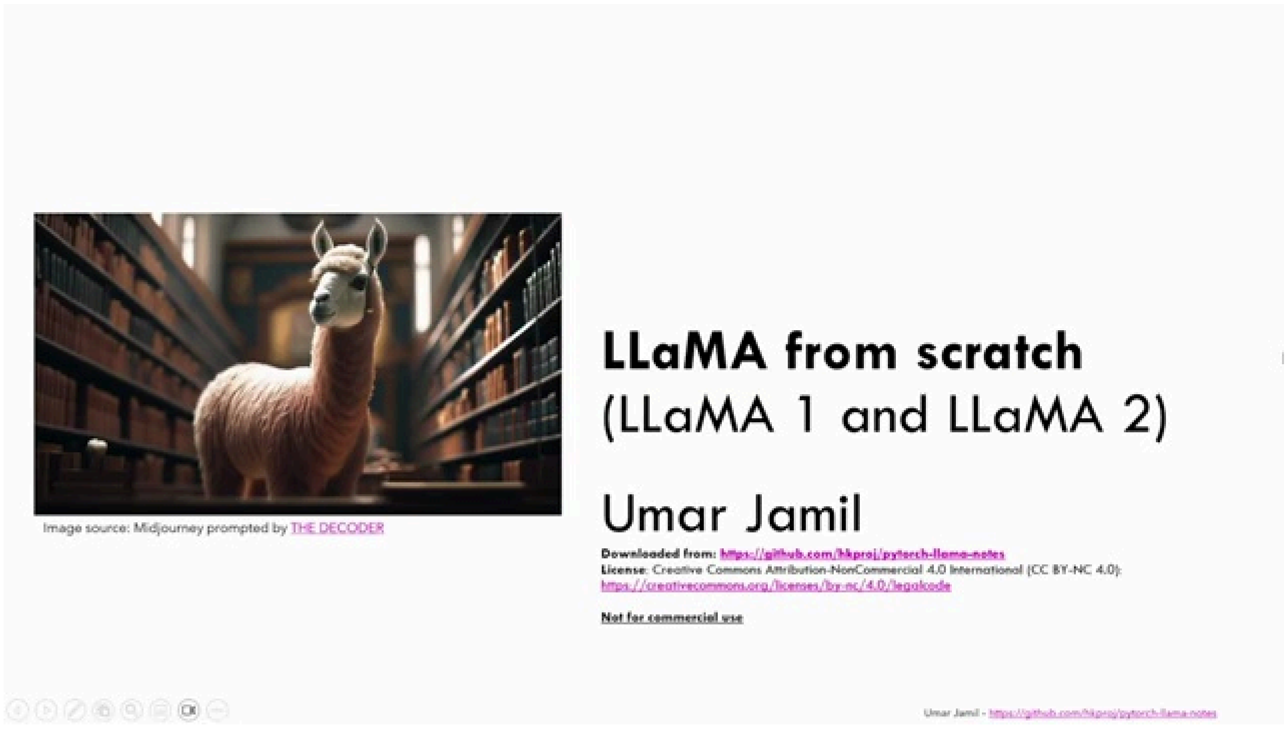


Image source: Midjourney prompted by THE DECODER

LLaMA from scratch
(LLaMA 1 and LLaMA 2)

Umar Jamil

Downloaded from: https://github.com/hkproj/pytorch-llama-notes
License: Creative Commons Attribution-NonCommercial 4.0 International (CC BY-NC 4.0): https://creativecommons.org/licenses/by-nc/4.0/legalcode

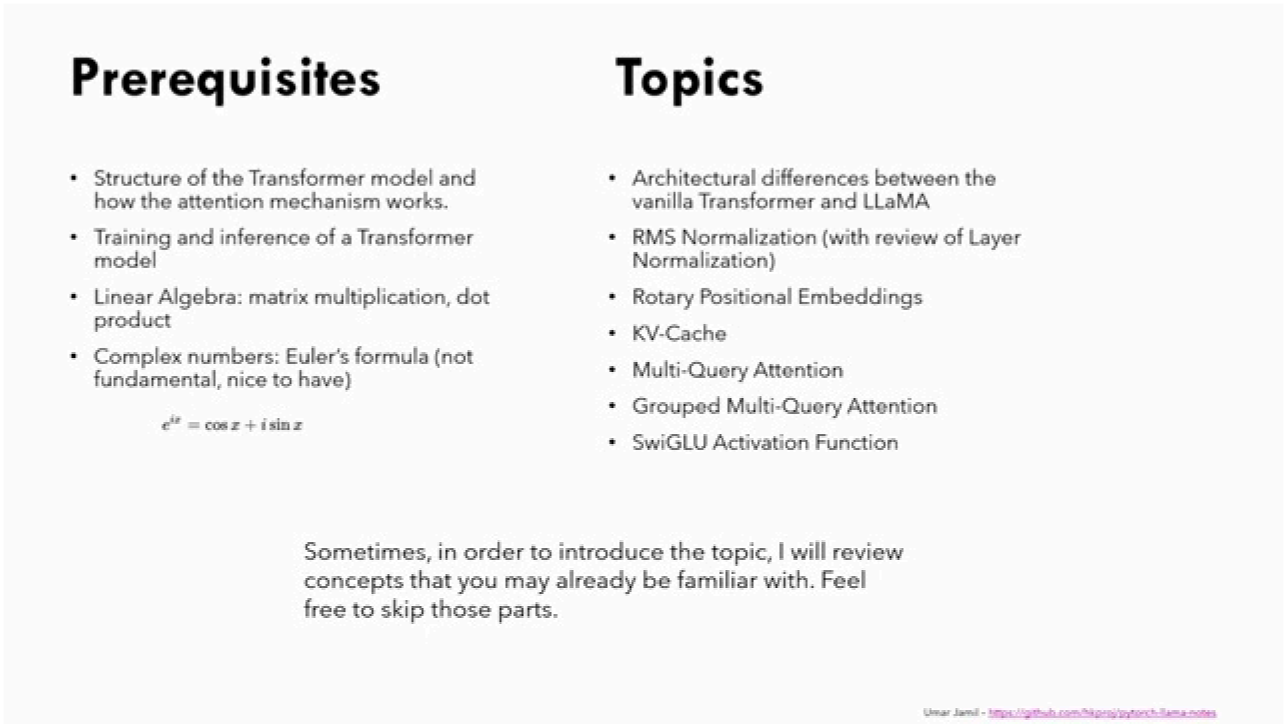
Not for commercial use

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 2

Time Range: 63.12 s - 127.28 s

KV cache, multi query attention, grouped multi query attention, this wiggle activation function for the feet forward layer, but of course I take for granted that you have some background knowledge. First of all I highly recommend that you watch my previous video about the transformer because you need to know how the transformer works and in my previous video I also explored the concept of training and inferencing a transformer model. It's 40 about 45 minutes and I think it's worth a watch because it will really give you a deep understanding of the transformer. After you have that knowledge you can watch this video. Anyway for those who have already watched the video but forgot some of some of some things I will review most of the concepts as we proceed through the topics. I also take for granted that you have some basic linear algebra knowledge, so matrix multiplication dot product, basic stuff anyway. Also because we will be using the rotary positional embedding, some knowledge about the complex numbers even if it's not fundamental. So if you don't have the, if you don't remember the complex numbers or how they work or the errors form it doesn't



Prerequisites

- Structure of the Transformer model and how the attention mechanism works.
- Training and inference of a Transformer model
- Linear Algebra: matrix multiplication, dot product
- Complex numbers: Euler's formula (not fundamental, nice to have)

$$e^{ix} = \cos x + i \sin x$$

Topics

- Architectural differences between the vanilla Transformer and LLaMA
- RMS Normalization (with review of Layer Normalization)
- Rotary Positional Embeddings
- KV-Cache
- Multi-Query Attention
- Grouped Multi-Query Attention
- SwiGLU Activation Function

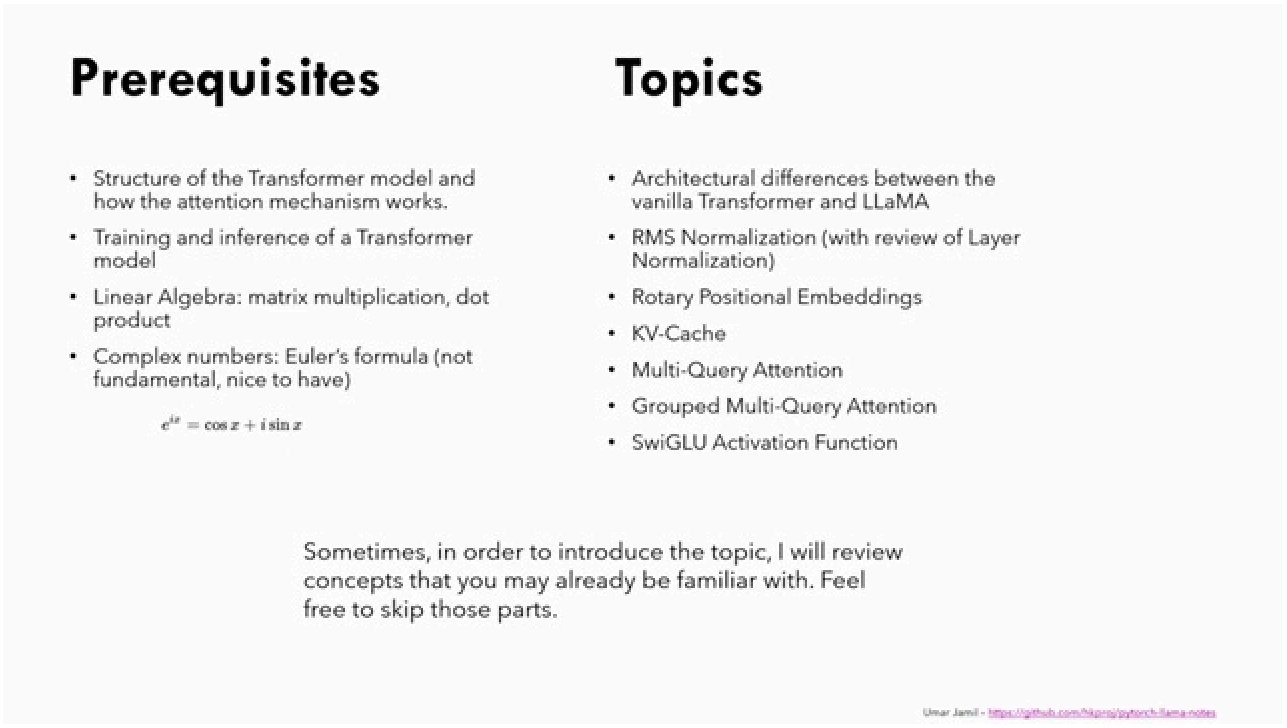
Sometimes, in order to introduce the topic, I will review concepts that you may already be familiar with. Feel free to skip those parts.

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 3

Time Range: 127.28 s - 187.44 s

you will understand the concept not the math it's not really fundamental. Sometimes I will be reviewing topics that maybe you are already familiar with so feel free to skip those parts. Let's start our journey by reviewing the architectural differences between the vanilla transformer and lama. This picture was built by me on the right side because I couldn't find the architecture picture from on the paper. So let's review the differences. As you remember in the Vanilla transformer we have an encoder part and the decoder part and the let me highlight it. So this is the encoder and the right side here is the decoder. While in lama we only have an encoder. First of all because the lama is a large language model so it has been trained on the next prediction prediction token task. So basically we only need the self-attention to predict the next token and we will see all this concepts. So we will see what is the next prediction task,



Prerequisites

- Structure of the Transformer model and how the attention mechanism works.
- Training and inference of a Transformer model
- Linear Algebra: matrix multiplication, dot product
- Complex numbers: Euler's formula (not fundamental, nice to have)

$$e^{ix} = \cos x + i \sin x$$

Topics

- Architectural differences between the vanilla Transformer and LLaMA
- RMS Normalization (with review of Layer Normalization)
- Rotary Positional Embeddings
- KV-Cache
- Multi-Query Attention
- Grouped Multi-Query Attention
- SwiGLU Activation Function

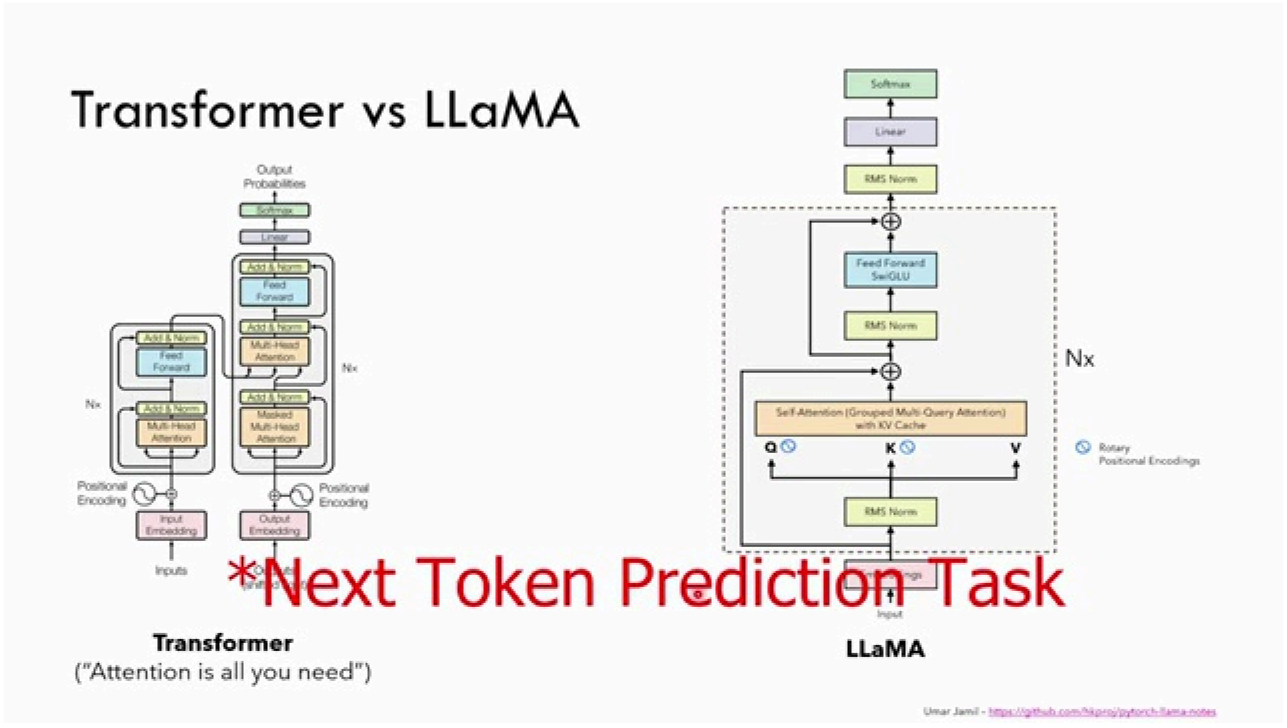
Sometimes, in order to introduce the topic, I will review concepts that you may already be familiar with. Feel free to skip those parts.

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 4

Time Range: 187.44 s - 249.12 s

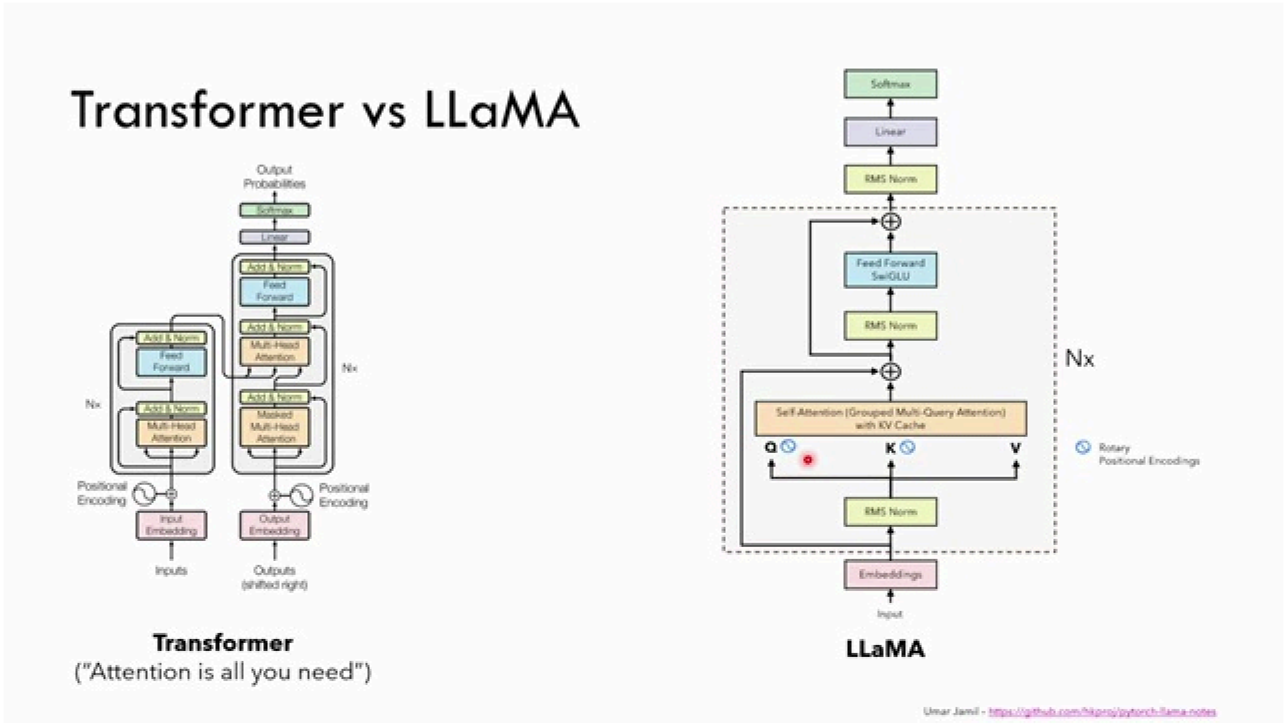
how it works and how this new self-attention works. The second difference that we can see from these pictures is that we have here at the beginning we have the embedding and also we have the embedding here on the original transformer. But right after the embedding we don't have the positional encoding but we have this RMS norm and actually all the norms have been moved before the blocks. So before we had the multi-head attention and then we had the other end norm which is this plus sign here so it's a concatenation of a skip connection and the output of the multi-head attention and the normalization. And we also have this normalization here, here, here. So after every block but here in lama we have it before every block and we will review what is the normalization and why it works like the way it is. Right after the normalization we have this query key and values input for the self-attention. One thing we can notice is that the positional encodings are not


Transformer vs LLaMA
Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Multi-Head Attention
Masked Multi-Head Attention
Nx
Positional Encoding
Input Embedding
Output Embedding
Inputs
Transformer
("Attention is all you need")
Softmax
Linear
RMS Norm
Feed Forward SwiGLU
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
Input
LLaMA
*Next Token Prediction Task
Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 5

Time Range: 249.12 s - 311.6 s

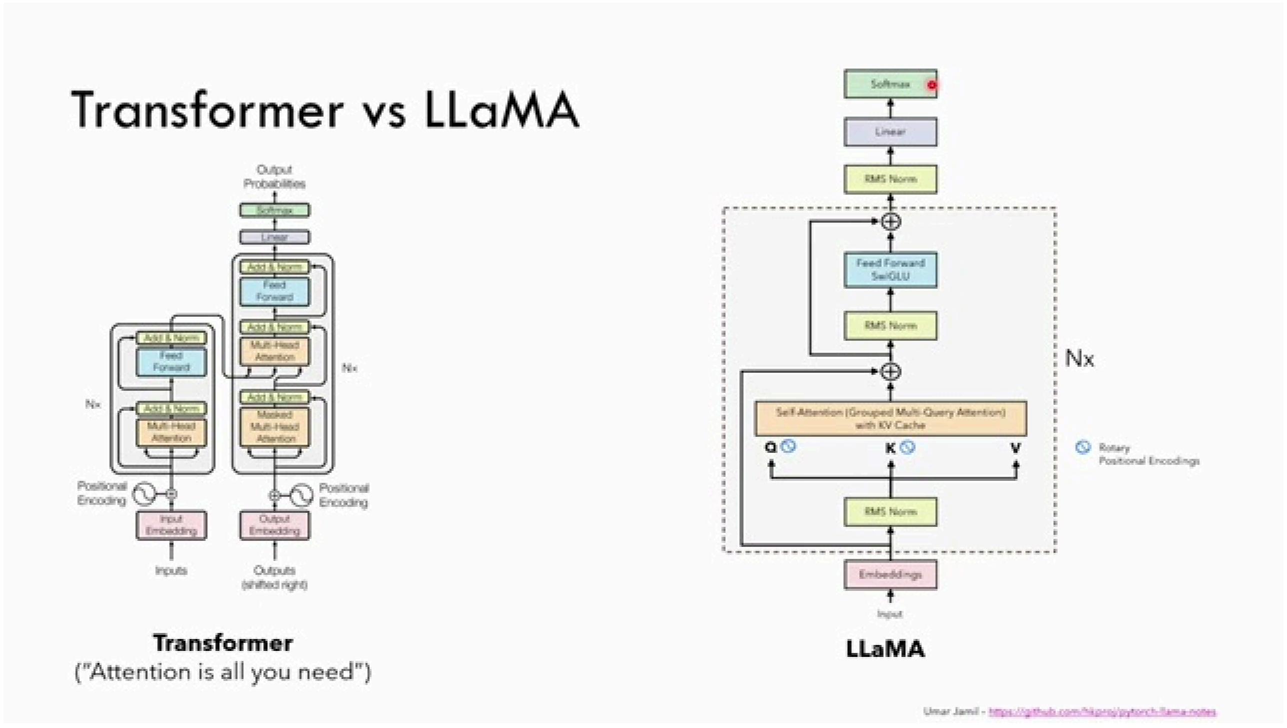
anymore the positional encodings of the transformer but they have become the rotary positional and they are only applied to the query and the keys but not the values and we will see also why. Another thing is the self-attention is now the self-attention with KV cache. We will see what is the KV cache and how it works and also we have this grouped multi-query attention. Another thing that changes is this feedforward layer. In the original feedforward layer of the vanilla transformer we had the relu activation function for the feedforward block but in lama we are using this wiglu function and we will see why. This nx means that this block here in the dashed lines is repeated n times one after another such that the output of the last layer is then fed to this rms norm then to the linear layer and then to the softmax and we will build each of these blocks from the bottom so I



Section 6

[Time Range: 311.6 s - 379.04 s]

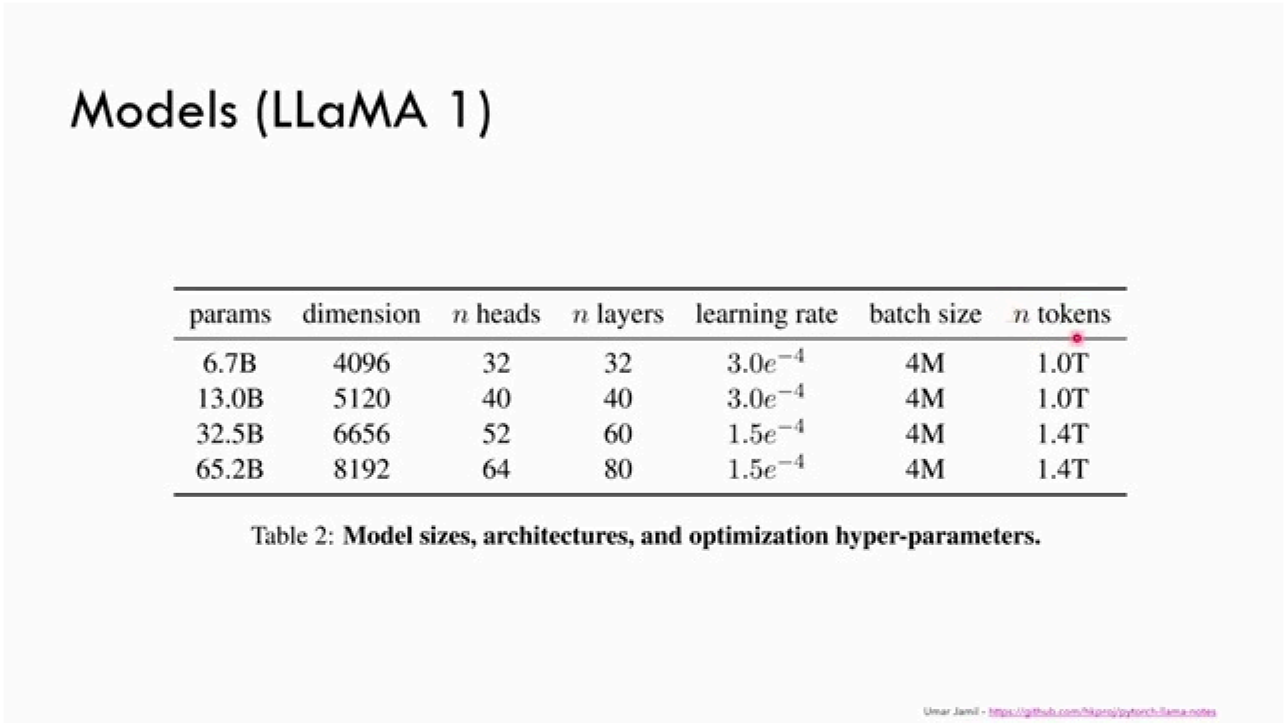
will show you exactly what these blocks do, how they work, how they interact with each other, what is the mat behind, what is the problem they were trying to solve. So we will have a deep knowledge of this model. Let's start our journey with reviewing the models introduced by lama. So lama1 came out in February 2023 and they had four dimensions for this model. One model was with 6.7 billion parameters 13, 32, 65 and then we have these numbers, what do they mean? The dimension here indicates the size of the embedding vector. So as you can see here we have this input embedding that we will review later. This is basically they convert each token into a vector of size indicated by this dimension. Then we have the number of heads. So how many heads the attention has? The number of layers. If you remember from the original transformer the dimension was 512, the number of heads was 8, the number of layers I think was 6 and then we have the number of



Section 7

Time Range: 379.04 s - 440.16 s

tokens each model was trained upon. So one trillion and one point four trillion. With the lama2 most of the numbers have doubled. So the context length is basically the sequence length. So how much how what is the longest sequence the model can be fed and then the number of tokens upon which the model have been trained is also doubled. So from one to two trillion for each size of the model while the parameters more or less remain the same. Then we have this column here GKA that indicates that these two sizes of the model, so the 34 billion and 70 billion they use the grouped query attention and we will see how it works. Let's start by reviewing what is the embedding the layer here and for that I will use the slides from my previous video. If you remember my previous video we introduced the embedding like this. So we have a sentence that is made of six words. What we do is we tokenize this sentence so it convert into tokens. The tokenization usually is done not by



Models (LLaMA 1)

params	dimension	n heads	n layers	learning rate	batch size	n tokens
6.7B	4096	32	32	$3.0e^{-4}$	4M	1.0T
13.0B	5120	40	40	$3.0e^{-4}$	4M	1.0T
32.5B	6656	52	60	$1.5e^{-4}$	4M	1.4T
65.2B	8192	64	80	$1.5e^{-4}$	4M	1.4T

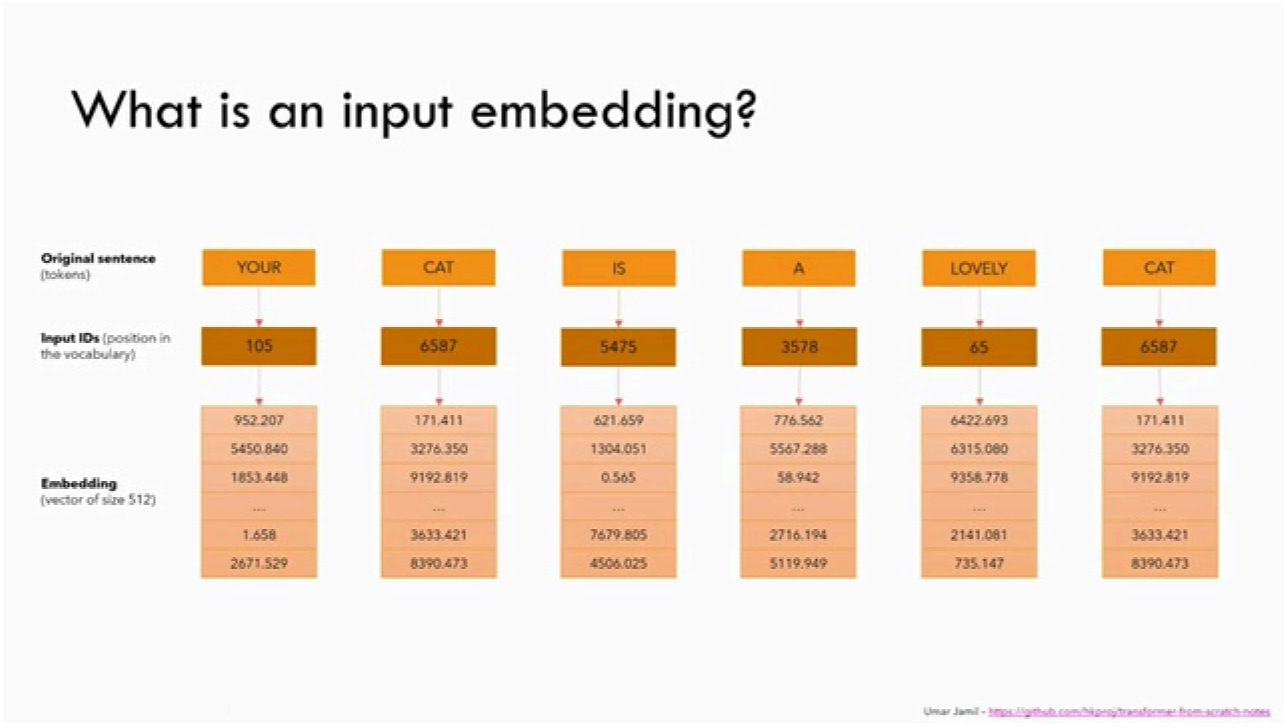
Table 2: **Model sizes, architectures, and optimization hyper-parameters.**

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 8

Time Range: 440.16 s - 504.4 s

space but by the BPE tokenizer. So actually each word will be split into sub words also but for clarity, for simplicity. We just tokenize our sentence by using the white space as a separator. So each token is separated by white space from other tokens. And each token is then mapped into its position into the vocabulary. So the vocabulary is how many words our vocabulary is the list of the words that our model recognizes. They don't have to be words of course they could be anything. They are just tokens. So each token occupies a position in this vocabulary and the input ID is indicated the number occupied by each token in the vocabulary. Then we map each input ID into a vector of size 512 in the original transformer but in LAMA it became 4,096. And this embedding are vectors that are learnable. So they are parameters for the model. And while the model will



What is an input embedding?

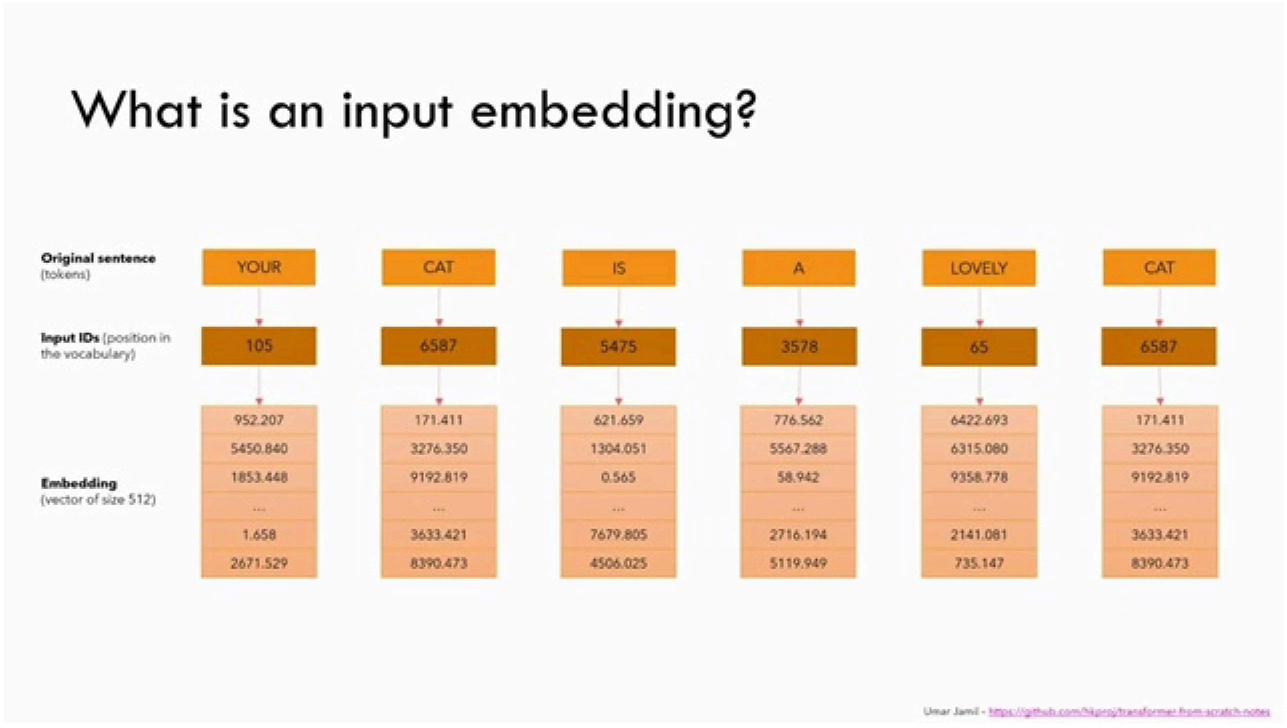
Original sentence (tokens)	YOUR	CAT	IS	A	LOVELY	CAT
Input IDs (position in the vocabulary)	105	6587	5475	3578	65	6587
Embedding (vector of size 512)	952.207	171.411	621.659	776.562	6422.693	171.411
	5450.840	3276.350	1304.051	5567.288	6315.080	3276.350
	1853.448	9192.819	0.565	58.942	9358.778	9192.819
	...	...	...	...	...	...
	1.658	3633.421	7679.805	2716.194	2141.081	3633.421
	2671.529	8390.473	4506.025	5119.949	735.147	8390.473

Umar Jamil - https://github.com/hkproj/transformer-from-scratch-notes

Section 9

Time Range: 504.4 s - 569.84 s

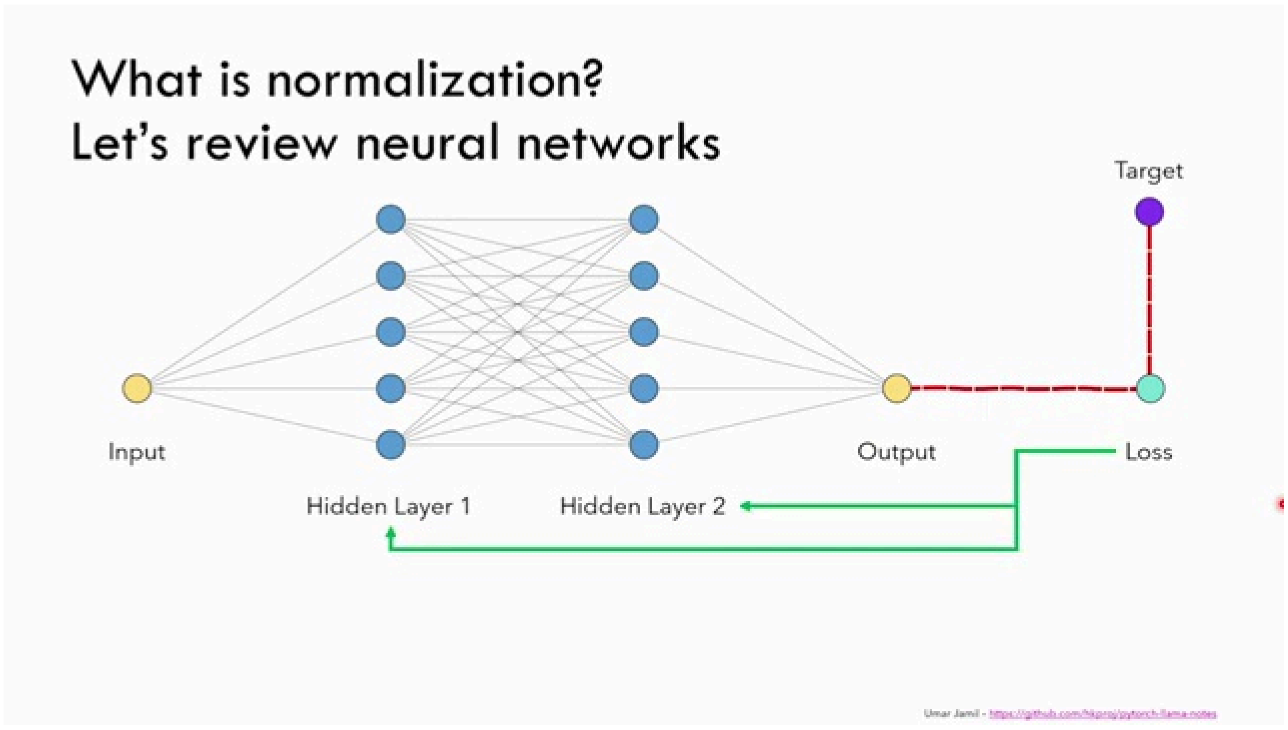
be trained, this embedding will change in such a way that they will capture the meaning of the word they are mapping. So we hope that for example the word cat and dog will have similar embedding because kind of the map similar, at least they are in the same semantic group. And also the word house and building, they will be very close to each other if we check the two vectors. And this is the idea behind the embedding. Now let's check what is normalization. Because this is the the layer right after the embedding. And for that let's introduce some review of the neural networks and how they work. So suppose we have a feed-forward neural with an input, a hidden layer made of neurons, another hidden layer made of another another five layer neurons, which then maps to an output. We usually have a target and comparing the output with the target we produce a loss. The loss is then propagated back to the two hidden



Section 10

Time Range: 569.84 s - 630.48 s

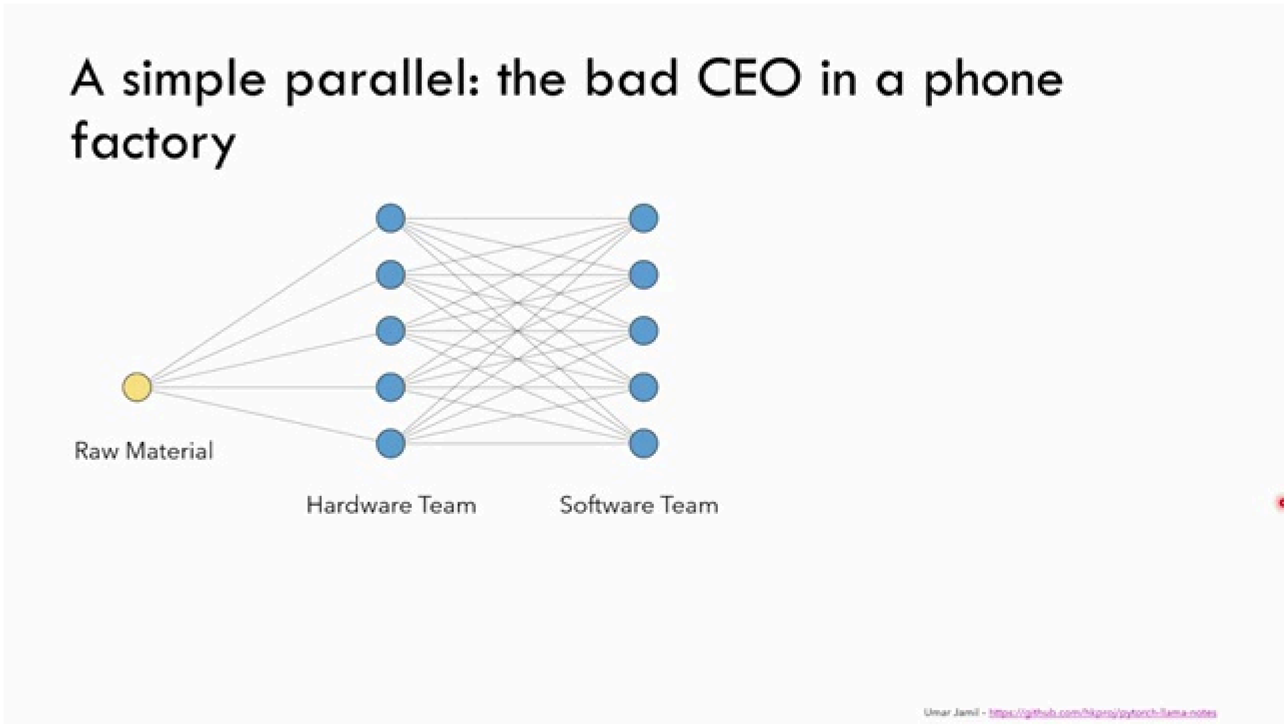
layers by means of back propagation. So what we do is we calculate the gradient of the loss with respect to each weight of these two hidden layers. And we modify these two these weights of the hidden layer accordingly, also according to the learning late that we have set. To check why we need to normalize and what is the need of normalization, I will make a simple a simplification of the neural network. So let's suppose our neural network is actually a factory, a factory that makes phones. So to make a phone we start with some raw material that are given to an hardware team that will take the raw material and produce some hardware. For example they may select the Bluetooth device, they may select the display, they may select the microphone, the camera, etc., and they make up the hardware of this phone. The hardware team then gives this prototype to the software team which then creates the software for this hardware. And then the output of the


What is normalization?
Let's review neural networks
Target
Input
Hidden Layer 1
Hidden Layer 2
Output
Loss
Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 11

Time Range: 630.48 s - 696.96 s

software team is the complete phone with hardware and software and is given as the output. The output is then compared with what was the original design of the phone. And then we compute a loss. So what is the difference between the target we had for our phone and what we actually produced? So suppose the loss is our CEO. And the loss is quite big, suppose. So our CEO will talk with the hardware team and with the software team and we'll tell them to adjust their strategy so as to go closer to the target next time. So suppose that the hardware was too expensive. So the CEO will tell the hardware team to use maybe as a smaller display to use a cheaper camera to change the Bluetooth to a lower-range one or to change the Wi-Fi to a low-energy one to change the battery, etc., etc. And we will also talk with the software team to adjust their strategy. And then maybe tell the software team to concentrate less on refactoring, to concentrate less on training, to hire more interns and not care too much about the employees because the

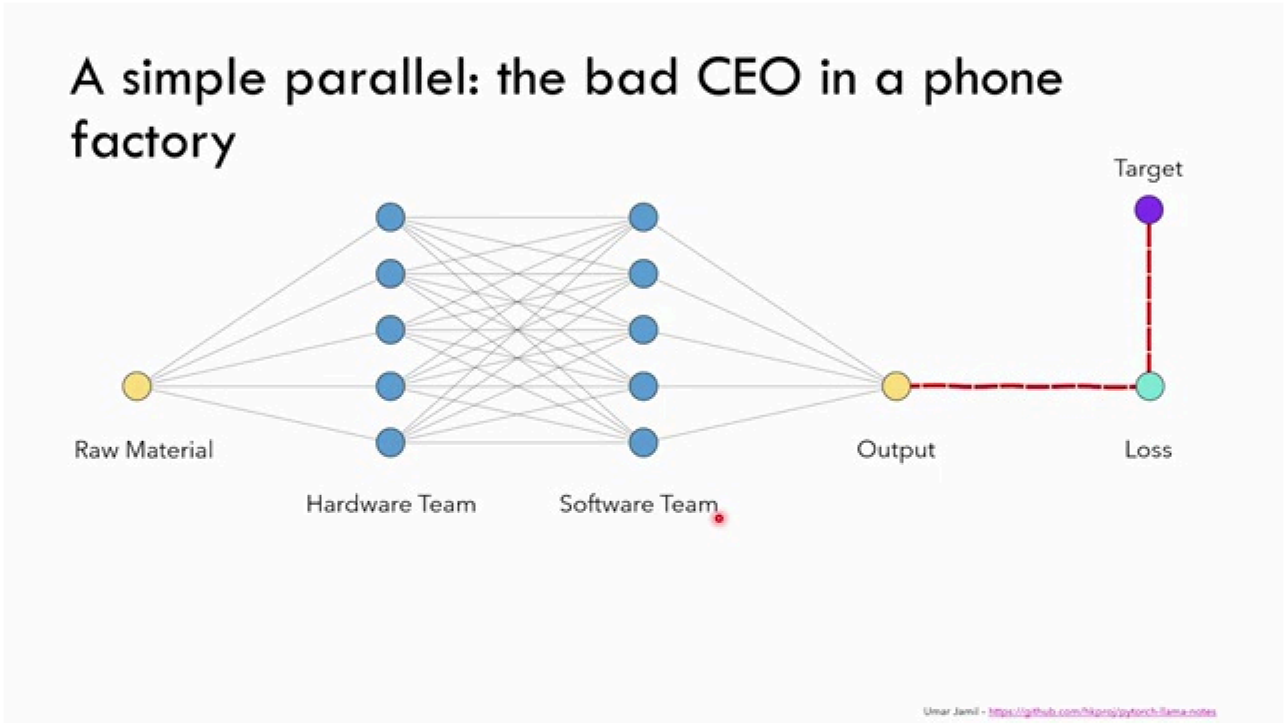


A simple parallel: the bad CEO in a phone factory

Section 12

Time Range: 696.96 s - 760.0 s

cost are too high blah blah. And it will adjust the strategy of the software and the hardware team. So the next time we start with the raw material again. So let's go back. We start with the raw material again. And the hardware team according to the new strategy set by the CEO will produce a new hardware. Now the problem arises. The software team now will receive a hardware that the software team has never seen before. Because the display has been changed, the Bluetooth has been changed, the Wi-Fi has been changed, everything has been changed. So the software team needs to redo a lot of work and especially they need to adjust their strategy a lot because they are dealing with something they have never seen before. So the output of the software team will be much different compared to what they previously output. And maybe it will be even further from the target because the software team was not ready to make all these adjustments. So maybe they wasted a lot

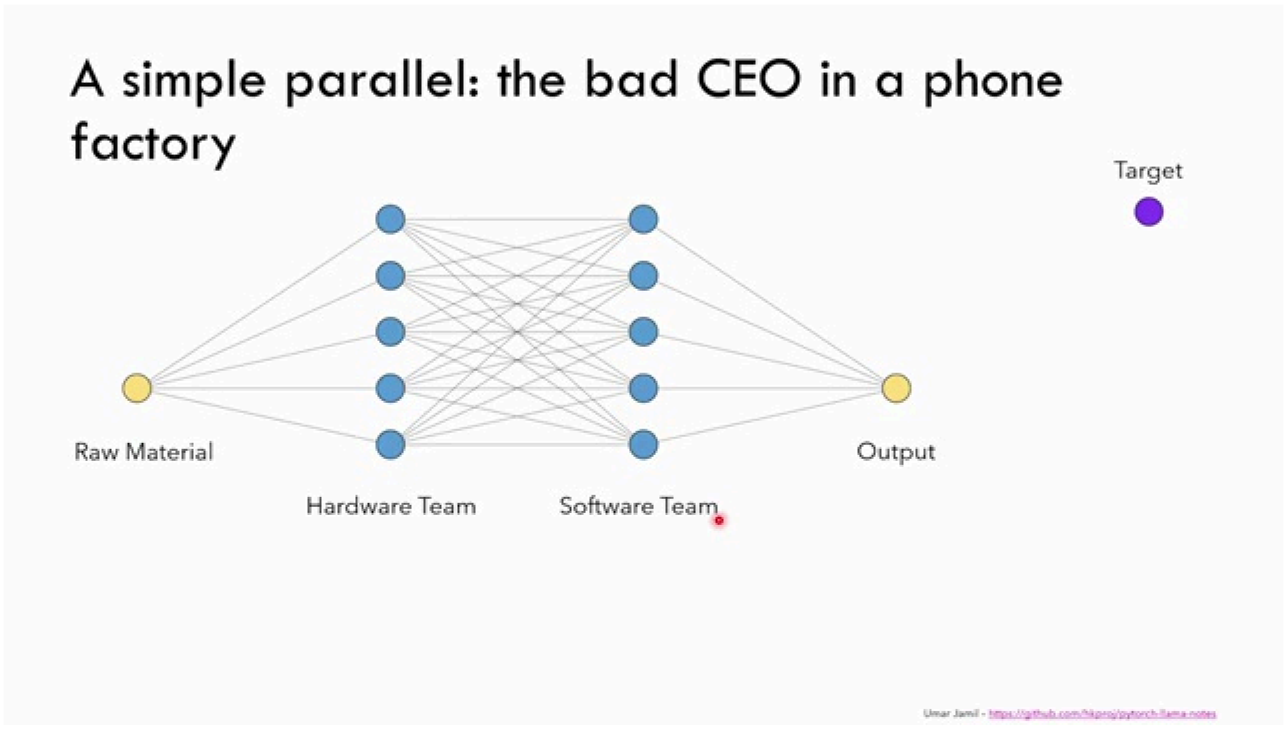


A simple parallel: the bad CEO in a phone factory

Section 13

Time Range: 760.0 s - 826.64 s

of time so they may be wasted a lot of resources. So they maybe could not even reach the target, even get closer to the target. So this time maybe the loss is even higher. So as you can see the problem arises by the fact that the loss function modifies the weights of the hardware team and the software team. But then the software team at the next iteration receives an input that it has never seen before and this input makes it produce an output that is much divergent compared to the one it used to produce before. This will make the model or she laid kind of in the loss and will make the training very slower. Now let's look what happens at the math level to understand how the normalization works. So let's review some meds. Suppose that we have a linear layer defined as an end-out linear with three input features and five output features with bias. This is the linear layer as defined in PyTorch. The linear layer would create two matrices, one called W, the weight

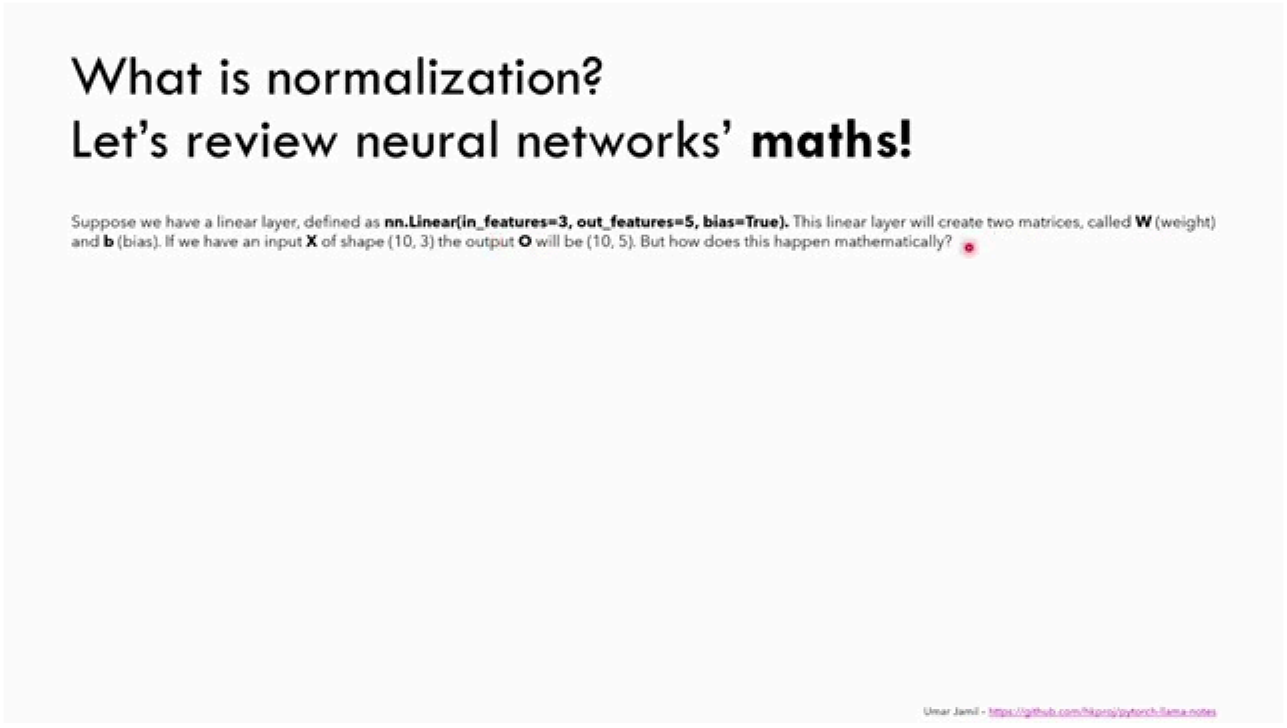


A simple parallel: the bad CEO in a phone factory

Section 14

Time Range: 826.64 s - 887.68 s

and one called B, the bias. Suppose we have an input of shape 10 rows by three columns, the output of this linear layer with this input X will be 10 rows by five columns. But how does this happen mathematically? Let's review it. So imagine we have our input which is 10 by three, which means that we have 10 items and each item has 10 features. The W matrix created by the linear layer will be five by three. So the output features by the three input features. And we can think of each of this row as one neuron, each of them having three weights, one weight for each of the input features of the X input. Then we have the bias vector and the bias vector is one weight for each neuron because the bias is one for every neuron. And this will produce an output which is 10 by five, which means we



What is normalization?
Let’s review neural networks’ **maths!**

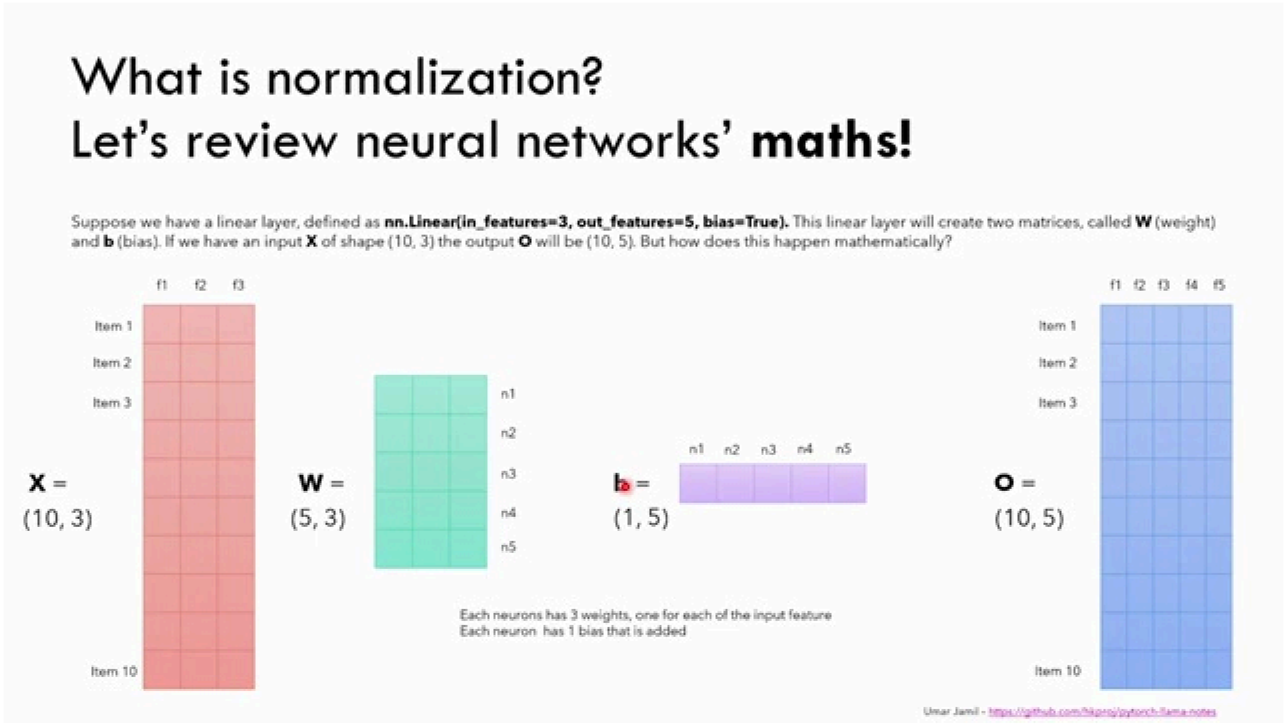
Suppose we have a linear layer, defined as **nn.Linear(in_features=3, out_features=5, bias=True).** This linear layer will create two matrices, called **W** (weight) and **b** (bias). If we have an input **X** of shape (10, 3) the output **O** will be (10, 5). But how does this happen mathematically?

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 15

Time Range: 887.68 s - 948.4 s

have 10 items with five features. Let's try to understand what is the flow of information in this matrices. The flow of information is governed by this expression. So the output is equal to the X multiplied by the transpose of the W matrix plus B. So let's suppose we have this input X and we have one item and the item one has three features, a one, a two and a three. The transpose of WT is this matrix here. So in which we swap the row with the columns because according to the formula we need to make the transpose of that matrix. So we have neuron one with the three weights, W1, W2, W3. We multiply the two and we obtain this matrix. So X multiplied by the transpose of produces this matrix here, in which this row one is the dot product of this row vector with this



What is normalization?
Let's review neural networks' **maths!**

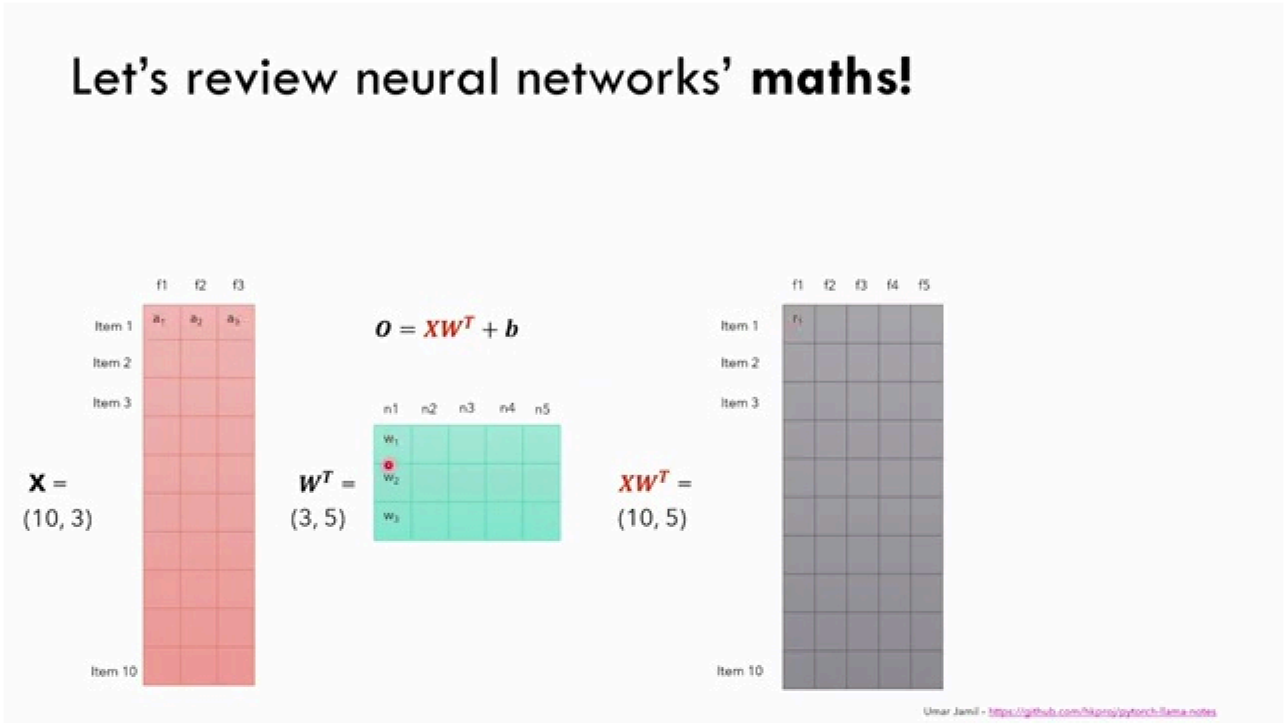
Suppose we have a linear layer, defined as **nn.Linear(in_features=3, out_features=5, bias=True).** This linear layer will create two matrices, called **W** (weight) and **b** (bias). If we have an input **X** of shape (10, 3) the output **O** will be (10, 5). But how does this happen mathematically?

Each neurons has 3 weights, one for each of the input feature
Each neuron has 1 bias that is added

Section 16

Time Range: 948.4 s - 1008.48 s

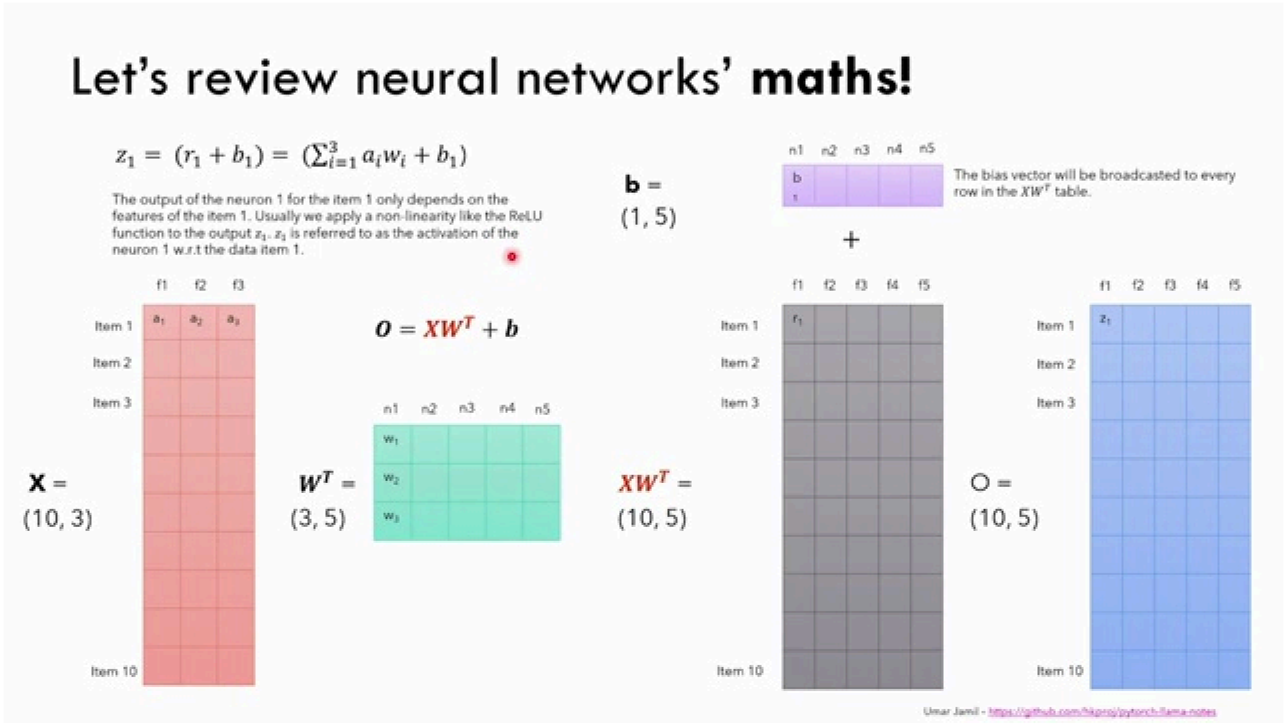
column vector. Then we add the B row vector. As you can see, to add two matrices they need to have the same dimension. But in PyTorch, because of broadcasting, this row will be added to this row here and then to independence it to this row and to this row, etc. because of the broadcasting. And then we will have this output. And the first item here will be Z1. What is Z1? Well, Z1 is equal to R1 plus B1. But what is R1? R1 is the dot product of this column with the this row or this row with this column. So it's this expression here. So the output of the neuron one for the item one only depends on the features of the item one. Usually after this output, we also apply a non-linearity like the RLU function, which and the argument of the RLU function is



Section 17

Time Range: 1008.48 s - 1074.64 s

referred to as the activation of the neuron one. Now, as we can see the output of the neuron one only depends on the input features of each item. So the output of a neuron for a data item depends on the features of the input data item and the neuron's parameter. We can think of the input to an neuron as the output of a previous layer. So for example, that input that we saw before the X, it may as well be the output of the previous layer. If the previous layer, after its weight are updated because of the gradient descent, changes drastically the output, like we did before, for example, because the CEO re-aligned the strategy of the hardware team. So the previous layer, the hardware team will produce an output that is drastically different compared to what it's used to produce. The next layer will have its output changed also drastically. So because it will be forced to readjust its weight drastically at the next step of the gradient descent. So what we don't like is the fact that the weight, the output of the previous layer changes too much so that



Let's review neural networks' **maths!**
$z_1 = (r_1 + b_1) = (\sum_{i=1}^{3} a_i w_i + b_1)$
$b =$ (1, 5)
$O = XW^T + b$
$X =$ (10, 3)
$W^T =$ (3, 5)
$XW^T =$ (10, 5)
O = (10, 5)
Item 1
Item 2
Item 3
Item 10
f1 f2 f3 f4 f5
n1 n2 n3 n4 n5
Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

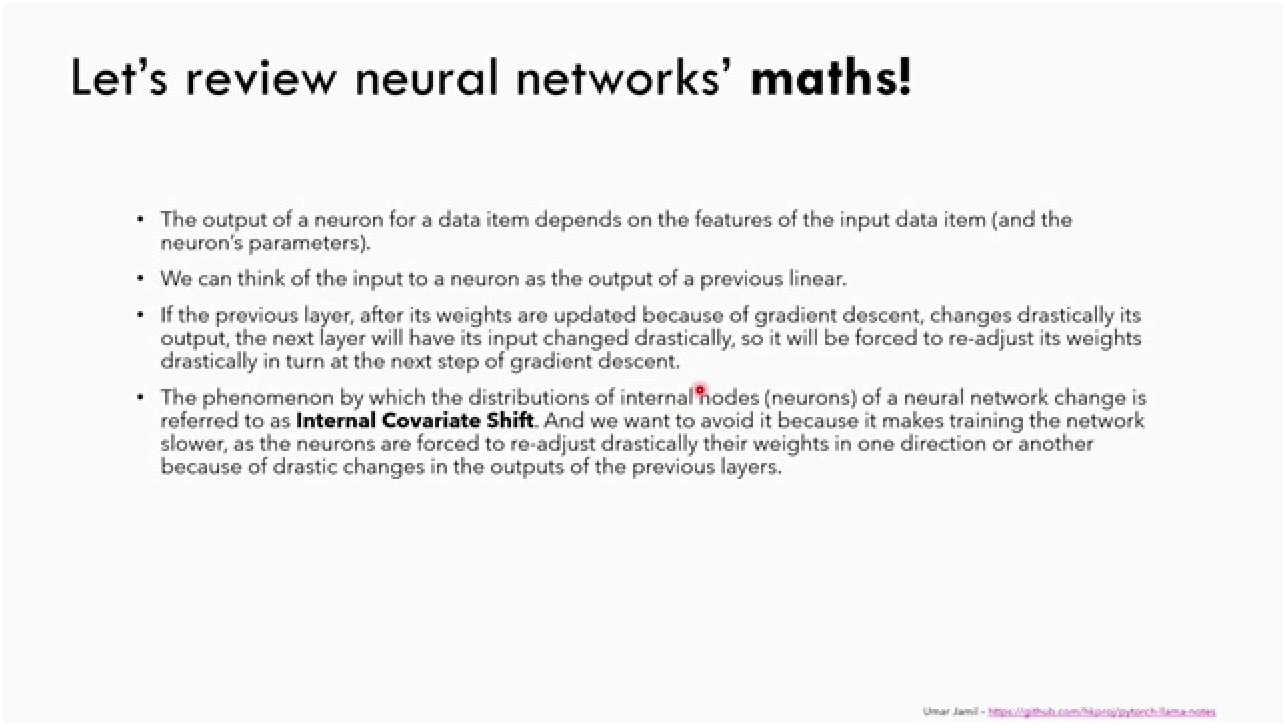
The output of the neuron 1 for the item 1 only depends on the features of the item 1. Usually we apply a non-linearity like the ReLU function to the output z_1. z_1 is referred to as the activation of the neuron 1 w.r.t the data item 1.

The bias vector will be broadcasted to every row in the XW^T table.

Section 18

Time Range: 1074.64 s - 1136.16 s

the next layer also has to change its output a lot because it's to add a hair to the strategy defined by the loss function. So this phenomenon by which the distribution of the internal nodes of a neuron change is referred to as internal covariate shift. And we want to avoid it because it makes training the network slower. As the neurons are forced to re-app just drastically their weights in one direction or another because of drastic changes in the output of the previous layers. So what do we do? We do layer normalization at least in the vanilla transformer. So let's review how the layer normalization works. Imagine we still have our input x defined with 10 rows by three columns. And for each of these items independently we calculate two statistics. One is the moon. So the mean and one is the sigma. So the variance. And then we normalize the values in



Let's review neural networks' **maths!**

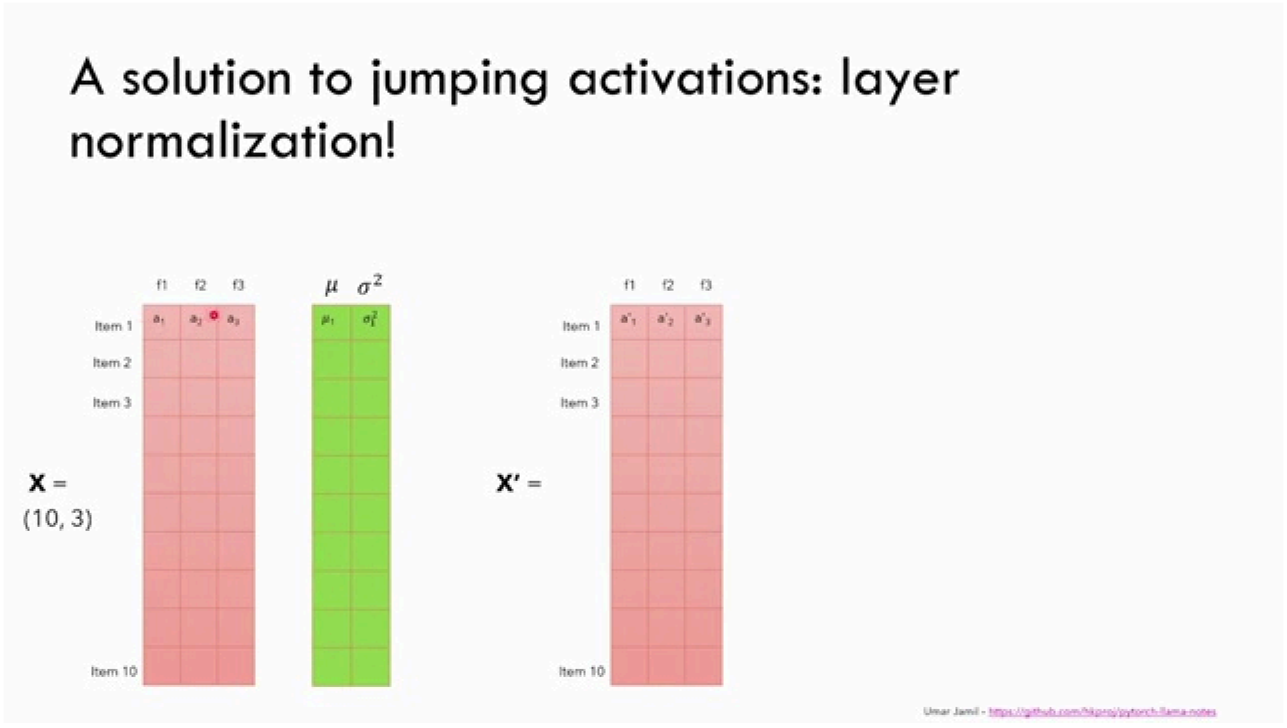
- The output of a neuron for a data item depends on the features of the input data item (and the neuron's parameters).
- We can think of the input to a neuron as the output of a previous linear.
- If the previous layer, after its weights are updated because of gradient descent, changes drastically its output, the next layer will have its input changed drastically, so it will be forced to re-adjust its weights drastically in turn at the next step of gradient descent.
- The phenomenon by which the distributions of internal nodes (neurons) of a neural network change is referred to as **Internal Covariate Shift**. And we want to avoid it because it makes training the network slower, as the neurons are forced to re-adjust drastically their weights in one direction or another because of drastic changes in the outputs of the previous layers.

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 19

Time Range: 1136.16 s - 1201.92 s

this matrix according to this formula. So we take basically x minus its moon. So each item minus the moon divided by the square root of the variance plus epsilon, where epsilon is a very small number so that we never divide by zero in this way even if the variance is very small. And each of these numbers then multiply with two parameters. One is gamma and one is beta. They are both learnable by the model. And they are useful because the model can adjust this gamma and beta to amplify the values that it needs. So before we had layer normalization we used to normalize with batch normalization. And with batch normalization the only difference is that instead of calculating the statistics by rows we calculated them by columns. So the future one, future two and future three. With layer normalization we do it by row. So each row will have its own moon and the sigma. So by using the layer normalization basically we transform the initial

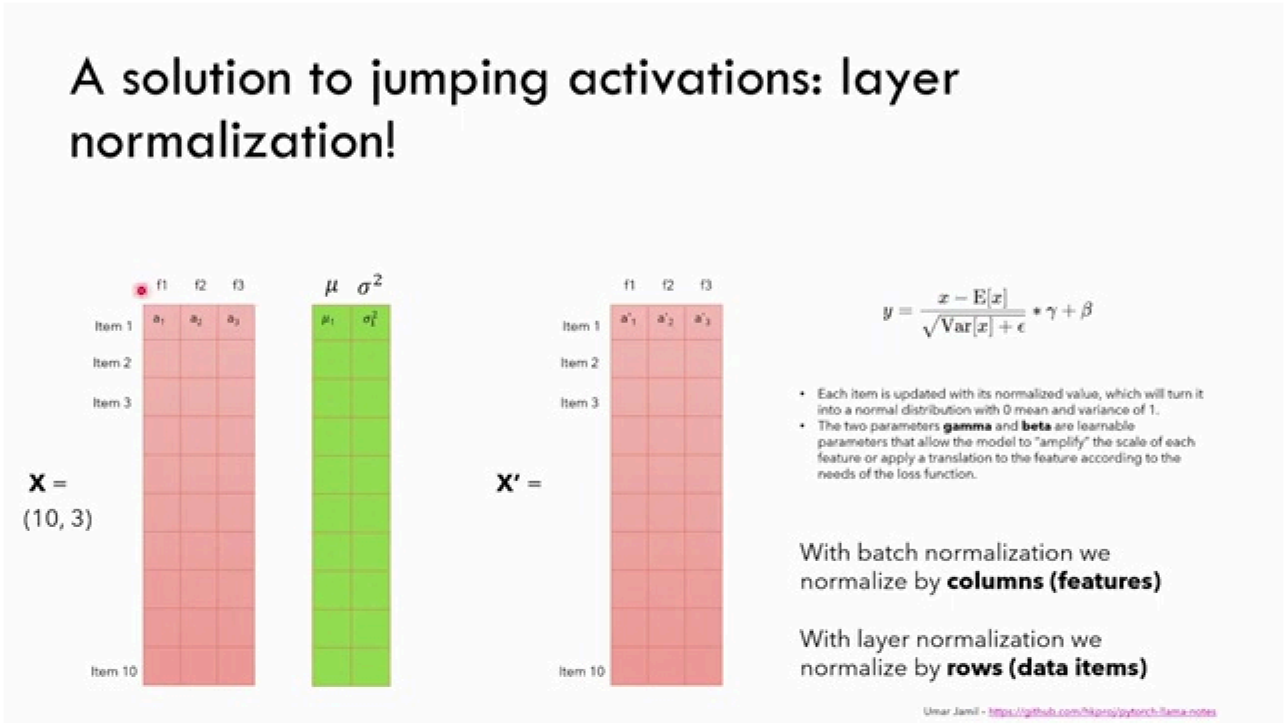


A solution to jumping activations: layer normalization!

Section 20

Time Range: 1201.92 s - 1264.08 s

distribution of features no matter what they are into a normalized numbers that are distributed zero mean and one variance. So this formula actually comes from probability statistics and if you remember if you remember let me use the pen. Okay if you remember basically if we have a variable x which is distributed like a normal variable with a mean let's say five and a variance of 36. If we do x minus it's mean so five divided by the square root of the variance. So 36. This one this variable here let's call it z will be distributed like n zero one. So it will become a standard Gaussian and this is exactly what we are doing here. So we are transforming them into standard Gaussian so that this value most of the times will be close to zero I mean will be



A solution to jumping activations: layer normalization!

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

- Each item is updated with its normalized value, which will turn it into a normal distribution with 0 mean and variance of 1.
- The two parameters **gamma** and **beta** are learnable parameters that allow the model to "amplify" the scale of each feature or apply a translation to the feature according to the needs of the loss function.

With batch normalization we normalize by **columns (features)**

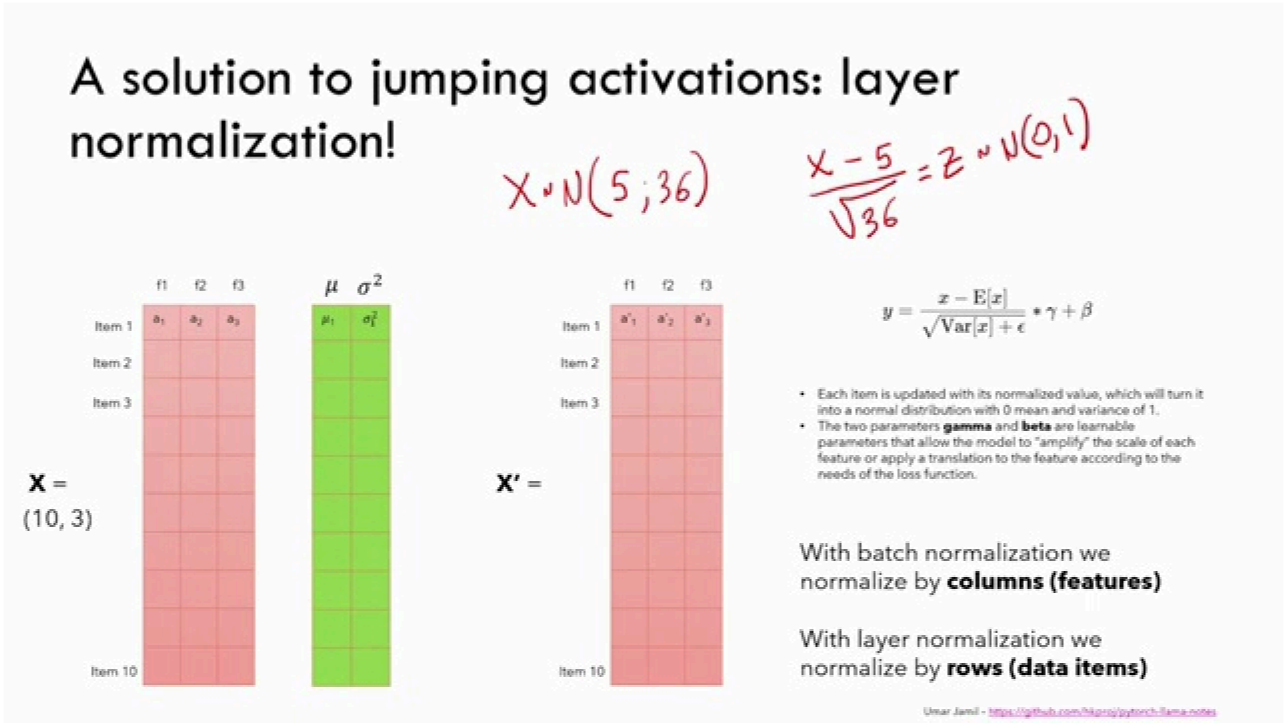
With layer normalization we normalize by **rows (data items)**

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 21

Time Range: 1264.08 s - 1324.24 s

distributed around zero. Now let's talk about root mean square normalization the one used by Lama. The root mean square normalization was introduced in this paper root mean square layer normalization from these two researchers and let's read the paper together a well known explanation of the success of layer norm is it's recentering and rescaling invariance property. So what do they mean what is the recentering and the rescaling invariance the fact that the features no matter what they are they will be recentered around the zero mean and rescaled to have a variance of one. The former enables the model to be insensitive to shift noises on both input and weights and the letter keeps the output representations intact when both input and weight are randomly scaled. Okay in this paper we hypothesize that the rescaling variance is the reason for success



A solution to jumping activations: layer normalization!

$X \sim N(5;36)$

$\frac{X-5}{\sqrt{36}} = Z \sim N(0,1)$

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

- Each item is updated with its normalized value, which will turn it into a normal distribution with 0 mean and variance of 1.
- The two parameters **gamma** and **beta** are learnable parameters that allow the model to "amplify" the scale of each feature or apply a translation to the feature according to the needs of the loss function.

With batch normalization we normalize by **columns (features)**

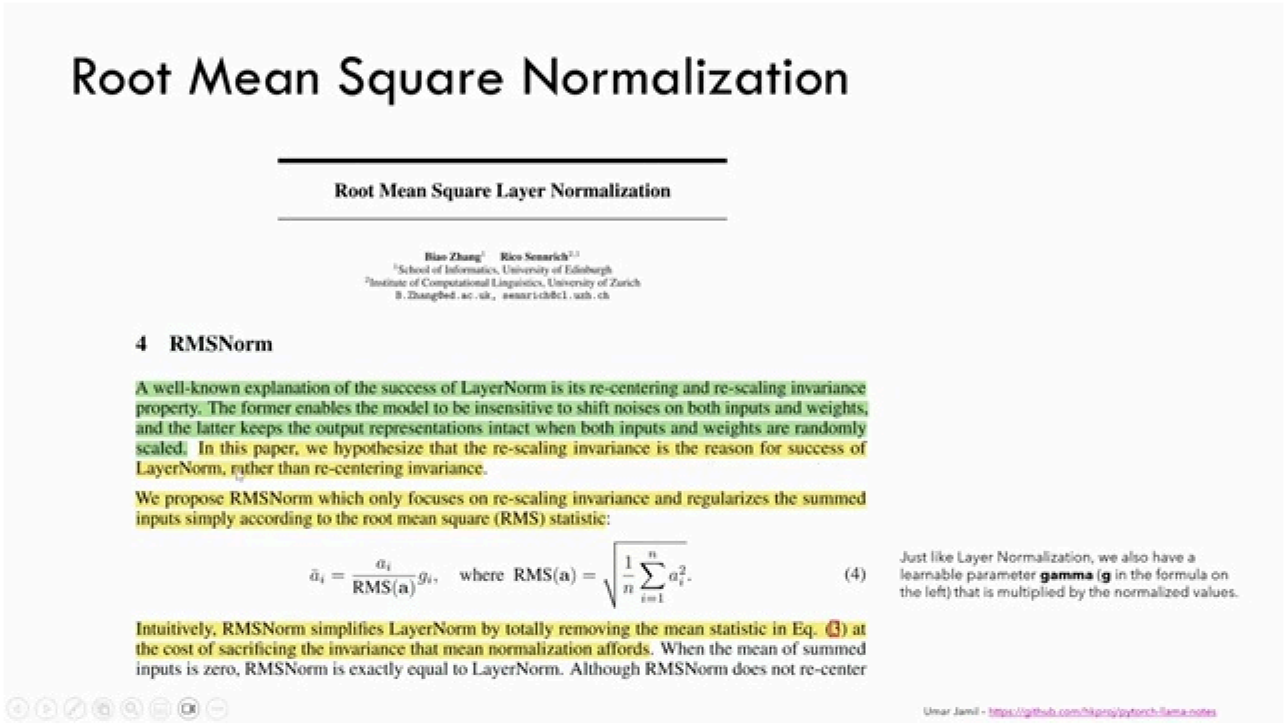
With layer normalization we normalize by **rows (data items)**

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 22

Time Range: 1324.24 s - 1385.6 s

of layer norm rather than the recentering invariance. So what they claim in this paper is that basically the success of layer norm is not because of the recentering and the rescaling but mostly because of the rescaling so this division by the variance basically so to have a variance of one. And what they do is basically they said okay could we find another statistic that doesn't depend on the mean because we believe that it's not necessary well yes they use this root mean square statistic so this statistic they find here. Oops. The statistic they find here and as you can see from the expression of this statistic we don't use the mean to calculate it anymore because the previous statistics here so the variance to be calculated you need the mean because if you remember the variance to be calculated needs the mean so the



Root Mean Square Normalization

Root Mean Square Layer Normalization

Biao Zhang[1] Rico Sennrich[2,1]
[1]School of Informatics, University of Edinburgh
[2]Institute of Computational Linguistics, University of Zurich
B.Zhang@ed.ac.uk, sennrich@cl.uzh.ch

4 RMSNorm

A well-known explanation of the success of LayerNorm is its re-centering and re-scaling invariance property. The former enables the model to be insensitive to shift noises on both inputs and weights, and the latter keeps the output representations intact when both inputs and weights are randomly scaled. In this paper, we hypothesize that the re-scaling invariance is the reason for success of LayerNorm, rather than re-centering invariance.

We propose RMSNorm which only focuses on re-scaling invariance and regularizes the summed inputs simply according to the root mean square (RMS) statistic:

$$\bar{a}_i = \frac{a_i}{\mathrm{RMS}(\mathbf{a})} g_i, \quad \text{where } \mathrm{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}. \qquad (4)$$

Just like Layer Normalization, we also have a learnable parameter **gamma** (**g** in the formula on the left) that is multiplied by the normalized values.

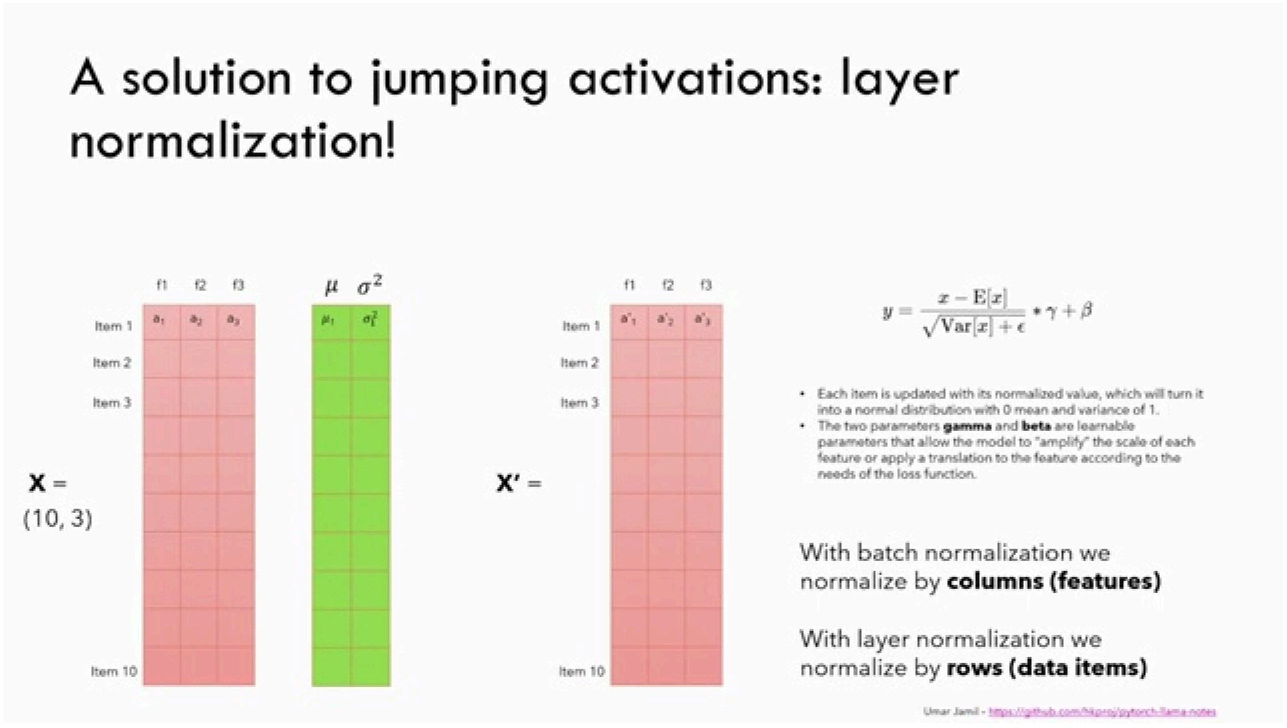
Intuitively, RMSNorm simplifies LayerNorm by totally removing the mean statistic in Eq. (3) at the cost of sacrificing the invariance that mean normalization affords. When the mean of summed inputs is zero, RMSNorm is exactly equal to LayerNorm. Although RMSNorm does not re-center

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 23

Time Range: 1385.6 s - 1449.36 s

variance is equal to the summation of x minus mv to the power of 2 divided by n so we need the the mean to calculate the variance so what the authors wanted to do in this paper they said okay because we don't need to recenter because we believe we hypothesize that the rescaling is not needed to obtain the effect of the layer normalization we want to find the statistic that doesn't depend on the mean and the RMS statistic doesn't depend on the mean so they do exactly the same thing that they did in the layer normalization so they find calculate the RMS statistic by rows so if one for each row and then they normalize according to this formula here so they just divide by the statistic RMS statistic and then multiply by this gamma parameter which is learnable now why why root mean square normalization well it requires less computation compared to layer normalization because we are not computing two statistics so we are not computing the mean and the sigma we are



A solution to jumping activations: layer normalization!

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

- Each item is updated with its normalized value, which will turn it into a normal distribution with 0 mean and variance of 1.
- The two parameters **gamma** and **beta** are learnable parameters that allow the model to "amplify" the scale of each feature or apply a translation to the feature according to the needs of the loss function.

With batch normalization we normalize by **columns (features)**

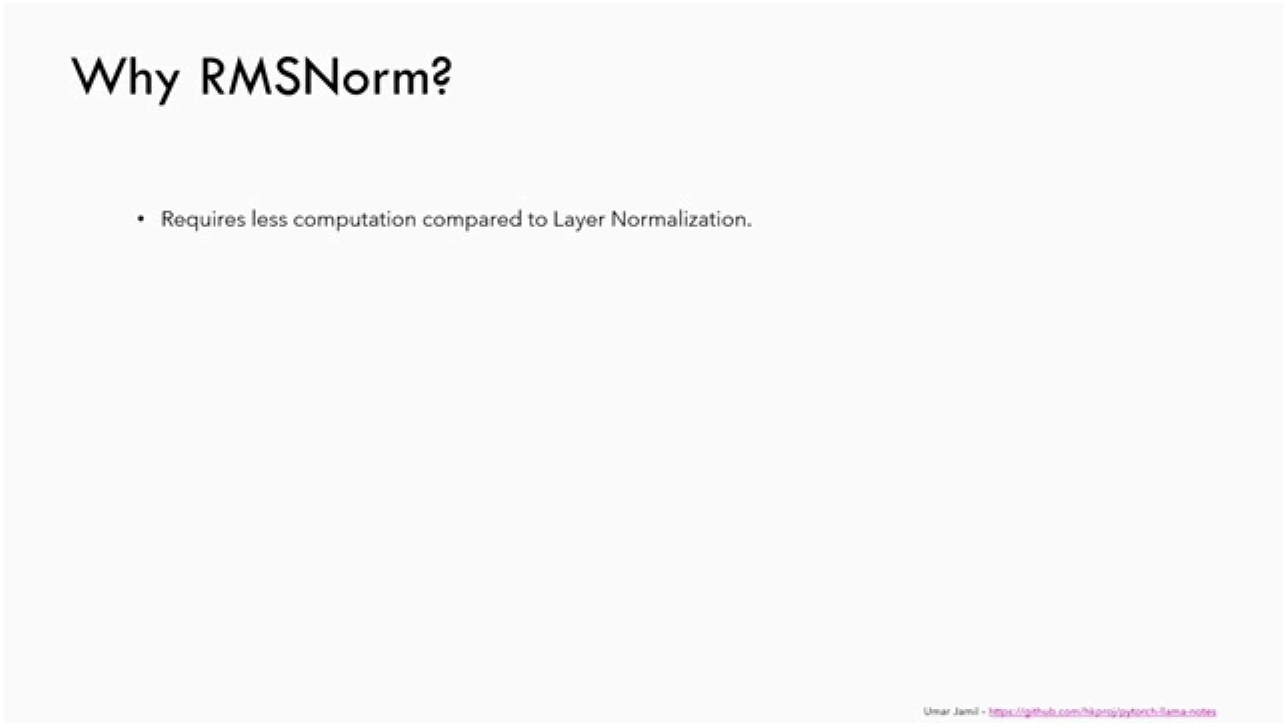
With layer normalization we normalize by **rows (data items)**

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 24

Time Range: 1449.36 s - 1516.48 s

only computing one so it gives you an computational advantage and it works well in practice so actually what the authors of the paper hypothesized is actually true we only need the invariance to obtain the effect made by the layer normalization we don't need the rescatering at least this is what happens with the llama the next topic we will be talking about is the positional encodings but before we introduce the rotary positional encodings let's review the positional encodings in the vanilla transformer as you remember after we transform our tokens into embedding so vectors of size 512 in the vanilla transformer then with some another vector to these embeddings that indicated the position of each token inside the sentence and these positional embeddings are fixed so they are not learned by the model they are computed once and then they are reused for every sentence during training and the inference and each word gets his own vector of size 512 we have a new kind of



Why RMSNorm?

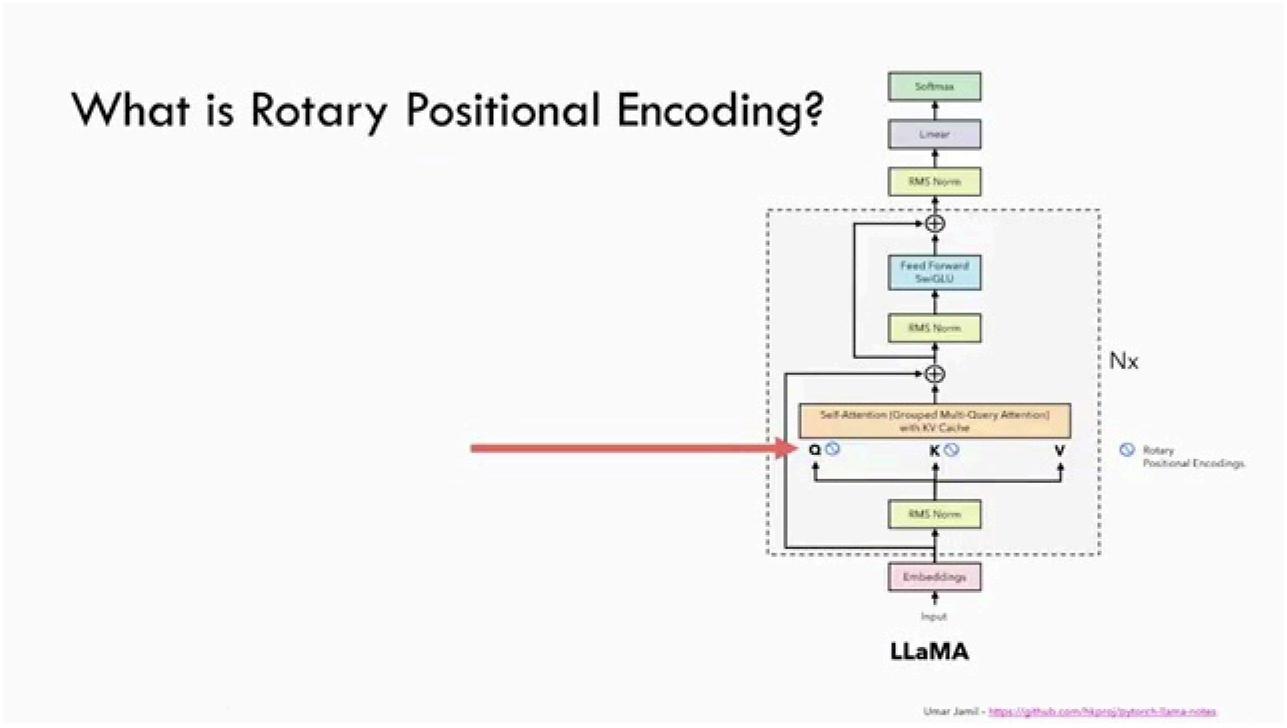
- Requires less computation compared to Layer Normalization.

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 25

Time Range: 1516.48 s - 1582.0 s

positional encoding called rotary positional encoding so absolute positional encodings are fixed vector that are added to the embedding of a token to represent its absolute position in the sentence so the token number one gets its own vector the token number two get its own vector the token number three get its own vector so the absolute positional encoding deal with one token at a time you can think of it as the pair latitude and longitude on a map each point on the earth will have its own unique latitude and longitude so that's an absolute indication of the position of each point on the earth and this is the same what happens with absolute positional encoding in the vanilla transformer we have one vector that represent exactly that position which is added to that particular token in that position with relative position link odings on the other hand it deals with two token and it is involved when we calculate the attention since the attention mechanism captures the intensity of how much two words are related to each other relative positional encodings tell the

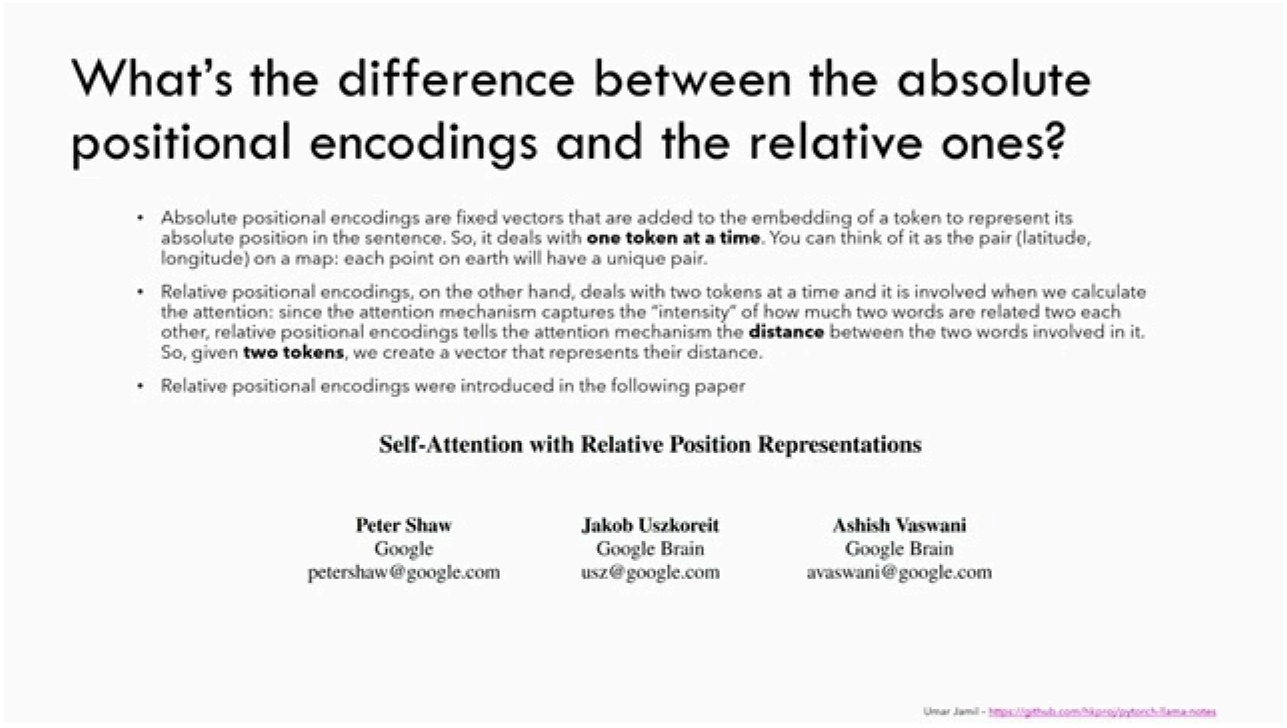

What is Rotary Positional Encoding?

Softmax
Linear
RMS Norm
Feed Forward SwiGLU
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input
LLaMA
Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 26

Time Range: 1582.0 s - 1648.4 s

attention mechanism the distance between the two words involved in this attention mechanism so given two tokens we create a vector that represents their distance this is why it's called because it's relative to the distance between two tokens relative positional encodings were first introduced in the following paper from Google and you can notice that the vastfani i think is the same author of the transformer model now with absolute positional encoding so from the attention is all you need when we calculate the dot product in the attention mechanism so if you remember the attention mechanism the formula let me write it attention is equal to the query multiplied by the transpose of the key divided by the square root of the model the model all of this then we do the



What's the difference between the absolute positional encodings and the relative ones?

- Absolute positional encodings are fixed vectors that are added to the embedding of a token to represent its absolute position in the sentence. So, it deals with **one token at a time**. You can think of it as the pair (latitude, longitude) on a map: each point on earth will have a unique pair.
- Relative positional encodings, on the other hand, deals with two tokens at a time and it is involved when we calculate the attention: since the attention mechanism captures the "intensity" of how much two words are related two each other, relative positional encodings tells the attention mechanism the **distance** between the two words involved in it. So, given **two tokens**, we create a vector that represents their distance.
- Relative positional encodings were introduced in the following paper

Self-Attention with Relative Position Representations

Peter Shaw
Google
petershaw@google.com

Jakob Uszkoreit
Google Brain
usz@google.com

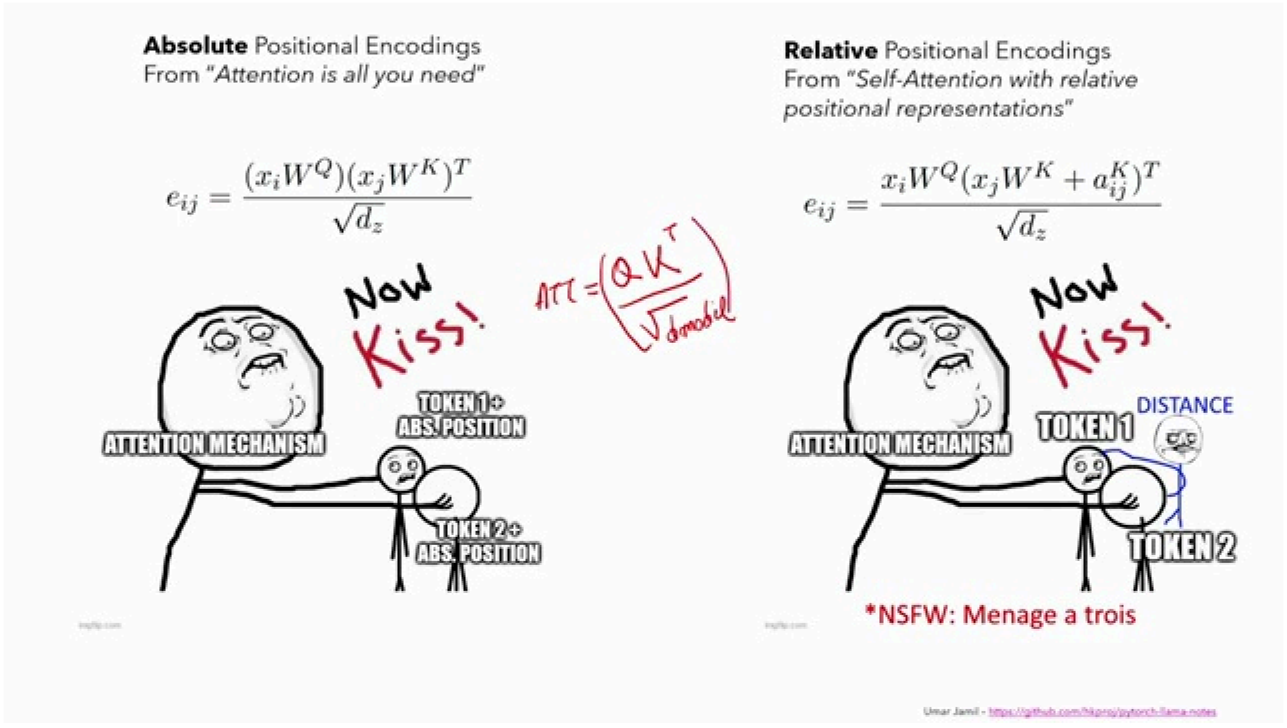
Ashish Vaswani
Google Brain
avaswani@google.com

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 27

Time Range: 1648.4 s - 1709.92 s

softmax and then we multiply it by v etc but we only concentrate on the q multiplied by the key transpose in this case and this is what we see here so when we calculate this dot product the attention mechanism is calculating the dot product between two tokens that already have the absolute position encoded into them because we already added the absolute position encoding to each token so in this attention mechanism from the vanilla transformer we have two tokens and the attention mechanism while in relative positional encodings we have three vectors we have the token one the token two and then we have this vector here we have this vector here that represents the distance between these two tokens and so we have three vectors involved in this attention mechanism and we want the attention mechanism to actually



Absolute Positional Encodings
From *"Attention is all you need"*

$$e_{ij} = \frac{(x_i W^Q)(x_j W^K)^T}{\sqrt{d_z}}$$

Relative Positional Encodings
From *"Self-Attention with relative positional representations"*

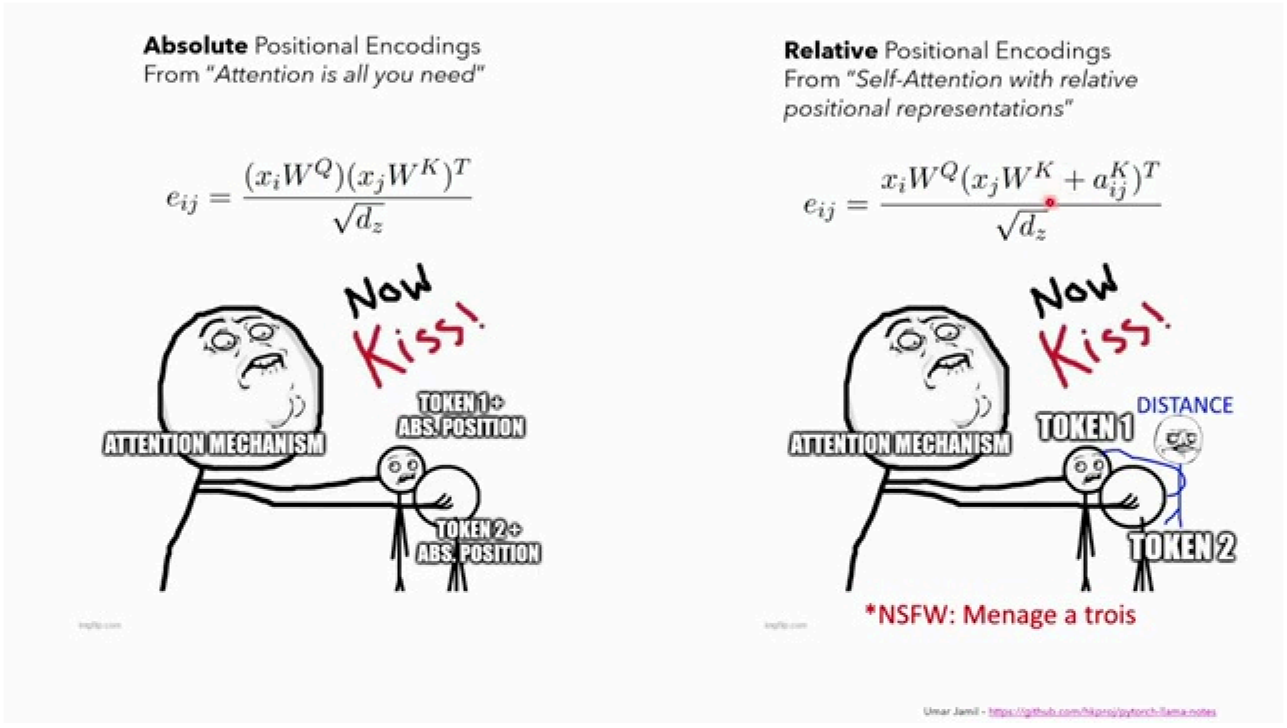
$$e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_z}}$$

$$ATT = \left(\frac{QK^T}{\sqrt{d_{model}}}\right)$$

Section 28

Time Range: 1709.92 s - 1771.36 s

match this token differently based on this vector here so this vector will indicate to the attention mechanism so to the dot product how to relate these two words that are at this particular distance with rotary positional embeddings we do a similar job and they were introduced with this paper so reformer and they are from a Chinese company so the dot product used in the attention mechanism is a type of inner product so if you remember from linear algebra the dot product is a kind of operation that has some properties and these properties are the kind of properties that every inner product must have so the inner product can be sought off as a generalization of the dot product what the authors of the paper wanted to do is can we find an inner product over the two vector query and key used in the attention mechanism that only depends on the two vectors



Absolute Positional Encodings
From *"Attention is all you need"*

$$e_{ij} = \frac{(x_i W^Q)(x_j W^K)^T}{\sqrt{d_z}}$$

Relative Positional Encodings
From *"Self-Attention with relative positional representations"*

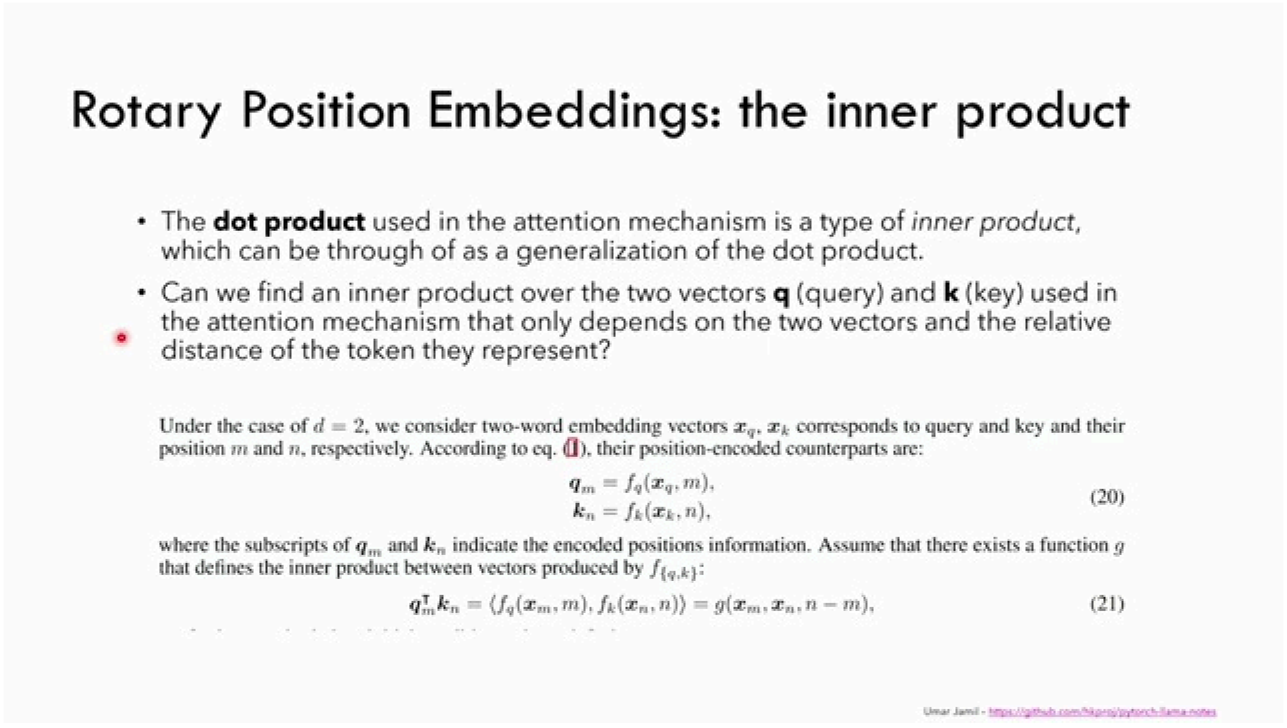
$$e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_z}}$$

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 29

[Time Range: 1771.36 s - 1832.16 s]

themselves and the relative distance of the token they represent that is given two vectors query and key that only contain the embedding of the word they represent and their position inside of the sentence so this m is actually an absolute number so it's a scalar it's represented the position of the word inside of the sentence and this n represents the position of the second word inside of the sentence what they wanted to say is can we find an inner product so this this particular parenthesis we see here is an inner product between these two vectors that behaves like this function g that only depends on the embedding of xm so the first token of xn the second token and the relative distance between them and no other information so this function will be given only the embedding of the first token the embedding of the second token and a number that represents



Rotary Position Embeddings: the inner product

- The **dot product** used in the attention mechanism is a type of *inner product*, which can be through of as a generalization of the dot product.
- Can we find an inner product over the two vectors **q** (query) and **k** (key) used in the attention mechanism that only depends on the two vectors and the relative distance of the token they represent?

Under the case of $d = 2$, we consider two-word embedding vectors $\boldsymbol{x}_q$, $\boldsymbol{x}_k$ corresponds to query and key and their position m and n, respectively. According to eq. (1), their position-encoded counterparts are:

$$\begin{aligned} \boldsymbol{q}_m &= f_q(\boldsymbol{x}_q, m), \\ \boldsymbol{k}_n &= f_k(\boldsymbol{x}_k, n), \end{aligned} \tag{20}$$

where the subscripts of $\boldsymbol{q}_m$ and $\boldsymbol{k}_n$ indicate the encoded positions information. Assume that there exists a function g that defines the inner product between vectors produced by $f_{\{q,k\}}$:

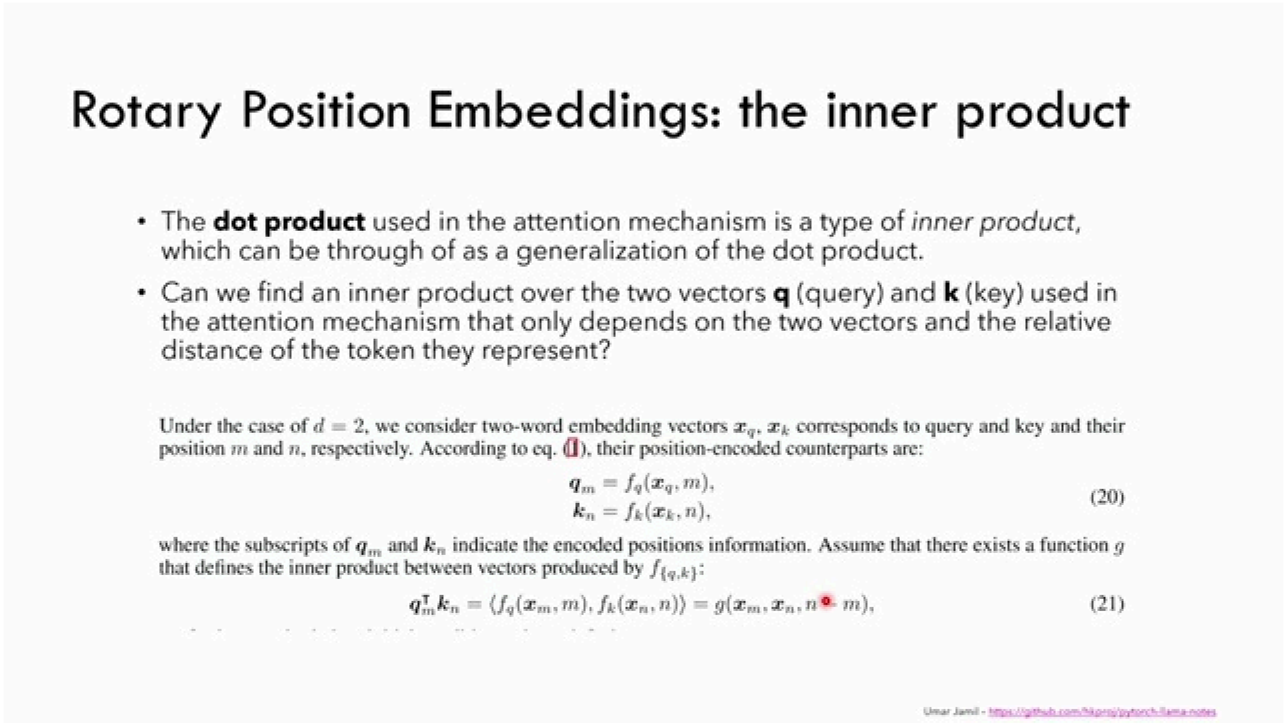
$$\boldsymbol{q}_m^{\intercal} \boldsymbol{k}_n = \langle f_q(\boldsymbol{x}_m, m), f_k(\boldsymbol{x}_n, n) \rangle = g(\boldsymbol{x}_m, \boldsymbol{x}_n, n - m), \tag{21}$$

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 30

Time Range: 1832.16 s - 1896.8 s

the relative position of these two tokens relative distance of these two tokens yes we can find such a function and the function is the one defined here so we can define a function g like the following that only needs only depends on the two embedding vector q and k and the relative distance and this function is defined in the complex number space and it can be converted by using the Euler formula into this form and another thing to notice is that this function here the one we are watching is defined for vectors of dimension two of course later we will see what happens when the dimension is bigger and when we convert this expression here which is in the complex number space into it through it matrix form through the Euler formula we can recognize this matrix here as the rotation matrix so this matrix here basically represents the rotation of a vector for example this



Rotary Position Embeddings: the inner product

- The **dot product** used in the attention mechanism is a type of *inner product*, which can be through of as a generalization of the dot product.
- Can we find an inner product over the two vectors **q** (query) and **k** (key) used in the attention mechanism that only depends on the two vectors and the relative distance of the token they represent?

Under the case of $d = 2$, we consider two-word embedding vectors $\boldsymbol{x}_q$, $\boldsymbol{x}_k$ corresponds to query and key and their position m and n, respectively. According to eq. (1), their position-encoded counterparts are:

$$\begin{aligned} \boldsymbol{q}_m &= f_q(\boldsymbol{x}_q, m), \\ \boldsymbol{k}_n &= f_k(\boldsymbol{x}_k, n), \end{aligned} \tag{20}$$

where the subscripts of $\boldsymbol{q}_m$ and $\boldsymbol{k}_n$ indicate the encoded positions information. Assume that there exists a function g that defines the inner product between vectors produced by $f_{\{q,k\}}$:

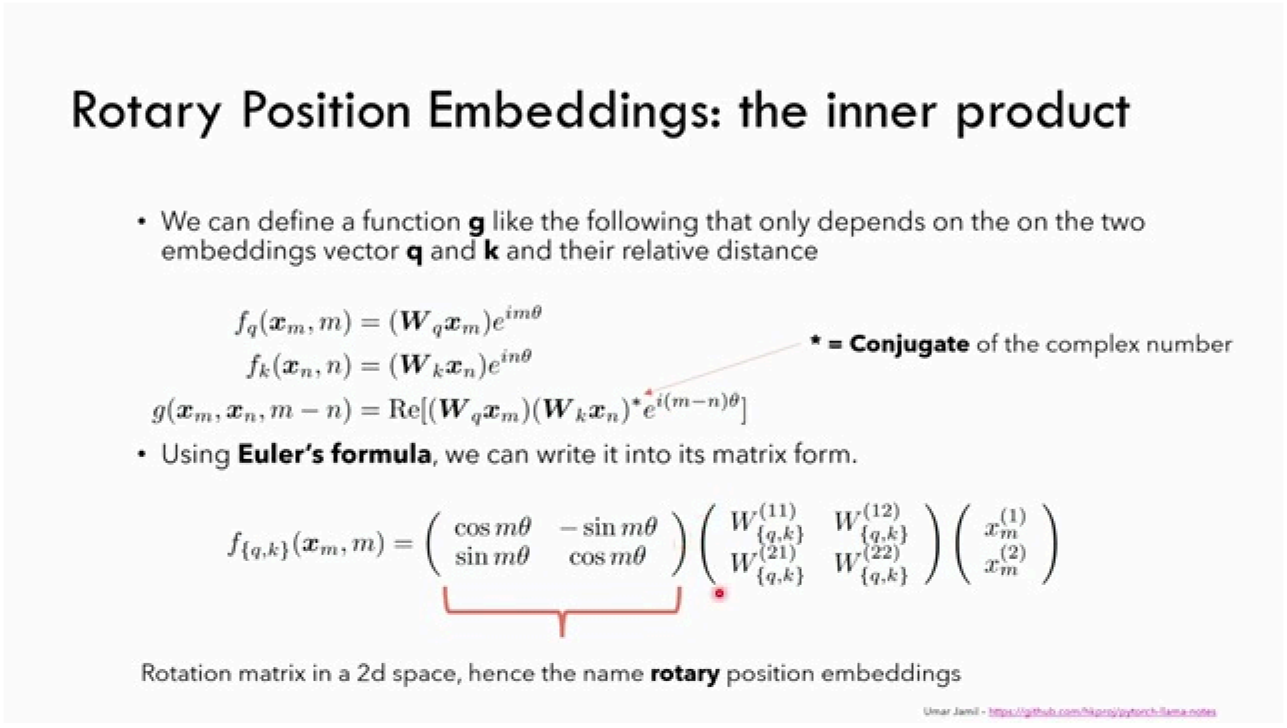
$$\boldsymbol{q}_m^{\intercal}\boldsymbol{k}_n = \langle f_q(\boldsymbol{x}_m, m), f_k(\boldsymbol{x}_n, n)\rangle = g(\boldsymbol{x}_m, \boldsymbol{x}_n, n - m), \tag{21}$$

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 31

Time Range: 1896.8 s - 1962.72 s

one here so this product here will be a vector and this rotation matrix will rotate this vector into the space by the amount described by m theta so the angle m theta let's see an example so imagine we have a vector v zero and we want to rotate it by theta by an angle theta here to arrive to the vector v prime so what we do is we multiply the vector v zero with this matrix exactly this one in which the values are calculated like this cosine of theta minus sine of theta sine of theta and cosine of theta and the resulting vector will be the same vector so the same length but rotated by this angle and this is why they are called rotary position in baddings because this vector represents a rotation now when the vector is not too dimensional but we have n dimension for example in the original transform model over embedding size is 512 and in lama is 4,096



Rotary Position Embeddings: the inner product

- We can define a function **g** like the following that only depends on the on the two embeddings vector **q** and **k** and their relative distance

$$f_q(\boldsymbol{x}_m, m) = (\boldsymbol{W}_q \boldsymbol{x}_m) e^{im\theta}$$
$$f_k(\boldsymbol{x}_n, n) = (\boldsymbol{W}_k \boldsymbol{x}_n) e^{in\theta}$$
$$g(\boldsymbol{x}_m, \boldsymbol{x}_n, m-n) = \mathrm{Re}[(\boldsymbol{W}_q \boldsymbol{x}_m)(\boldsymbol{W}_k \boldsymbol{x}_n)^* e^{i(m-n)\theta}]$$

*** = Conjugate** of the complex number

- Using **Euler's formula**, we can write it into its matrix form.

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \underbrace{\begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix}}_{} \begin{pmatrix} W^{(11)}_{\{q,k\}} & W^{(12)}_{\{q,k\}} \\ W^{(21)}_{\{q,k\}} & W^{(22)}_{\{q,k\}} \end{pmatrix} \begin{pmatrix} x^{(1)}_m \\ x^{(2)}_m \end{pmatrix}$$

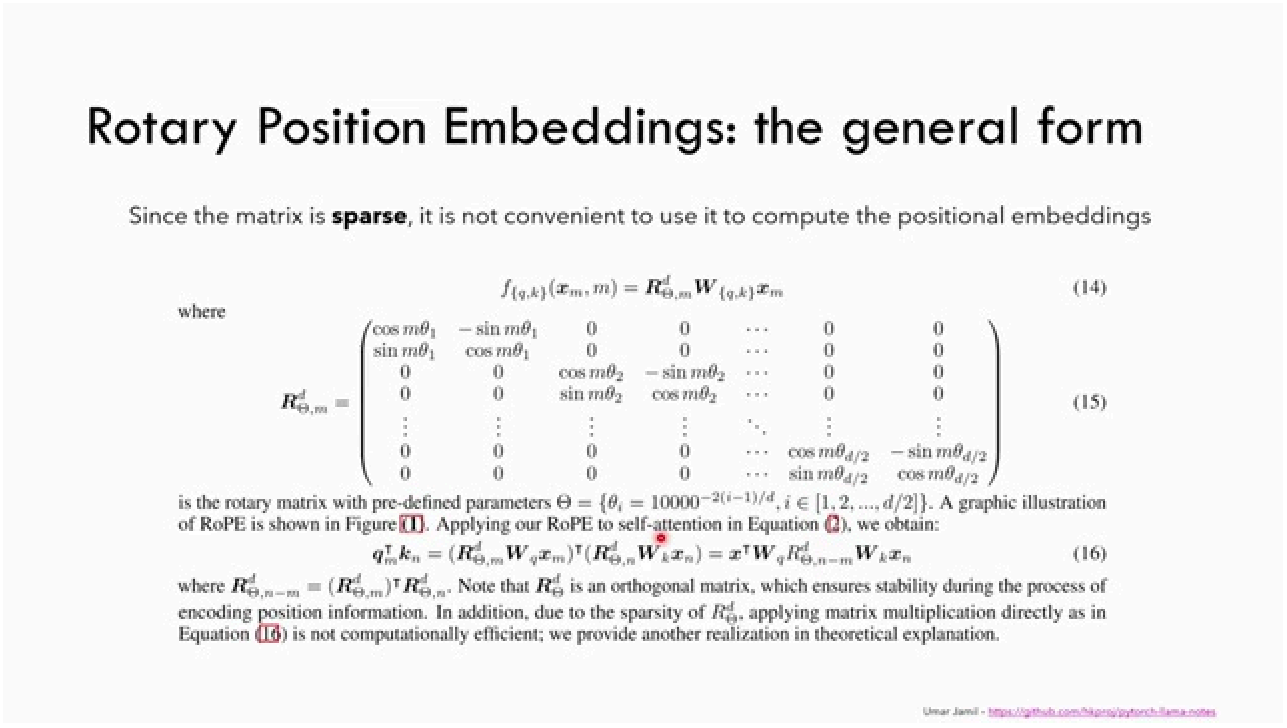
Rotation matrix in a 2d space, hence the name **rotary** position embeddings

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 32

Time Range: 1962.72 s - 2024.8 s

we need to use this form now i want you to notice not what are the numbers in this in this matrix but the fact that this matrix is a sparse so it is not convenient to use it to compute the positional embeddings because if we multiply by this embedding our tensor flow over GPU over computer we'll do a lot of operations that are useless because we already know that most of the products will be zero so is there a better way more computationally efficient way to do this computation well there is this form here so given a token with the embedding vector x and the position m of the token inside the sentence this is how we compute the position embedding for the token we take his dimensions of the token we multiply by this matrix here computed like the following where the theta are fixed m is the position of the token x1 x2 x3 are the dimension of the embedding so the first dimension of the embedding the second dimension of the embedding etc plus



Rotary Position Embeddings: the general form

Since the matrix is **sparse**, it is not convenient to use it to compute the positional embeddings

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \boldsymbol{R}^d_{\Theta,m} \boldsymbol{W}_{\{q,k\}} \boldsymbol{x}_m \tag{14}$$

where

$$\boldsymbol{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix} \tag{15}$$

is the rotary matrix with pre-defined parameters $\Theta = \{\theta_i = 10000^{-2(i-1)/d}, i \in [1, 2, ..., d/2]\}$. A graphic illustration of RoPE is shown in Figure (1). Applying our RoPE to self-attention in Equation (2), we obtain:

$$\boldsymbol{q}^\intercal_m \boldsymbol{k}_n = (\boldsymbol{R}^d_{\Theta,m} \boldsymbol{W}_q \boldsymbol{x}_m)^\intercal (\boldsymbol{R}^d_{\Theta,n} \boldsymbol{W}_k \boldsymbol{x}_n) = \boldsymbol{x}^\intercal \boldsymbol{W}_q R^d_{\Theta,n-m} \boldsymbol{W}_k \boldsymbol{x}_n \tag{16}$$

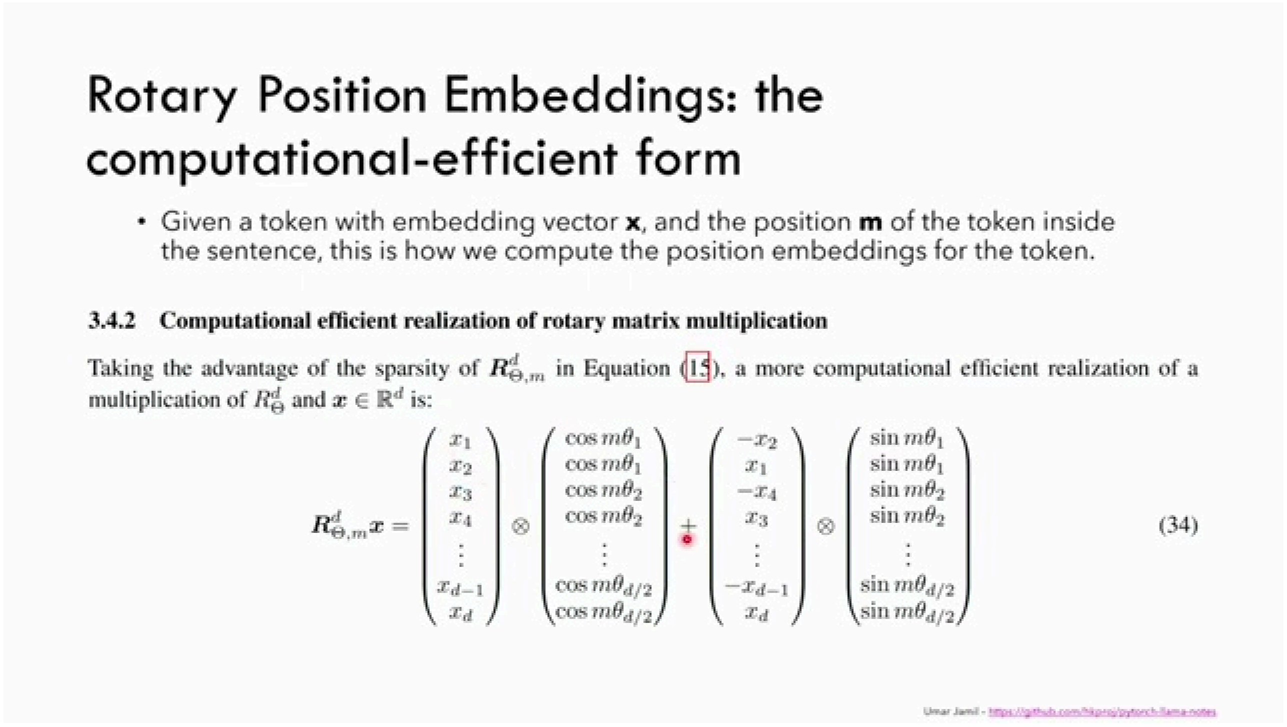
where $\boldsymbol{R}^d_{\Theta,n-m} = (\boldsymbol{R}^d_{\Theta,m})^\intercal \boldsymbol{R}^d_{\Theta,n}$. Note that $\boldsymbol{R}^d_\Theta$ is an orthogonal matrix, which ensures stability during the process of encoding position information. In addition, due to the sparsity of R^d_Θ, applying matrix multiplication directly as in Equation (16) is not computationally efficient; we provide another realization in theoretical explanation.

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 33

Time Range: 2024.8 s - 2085.04 s

minus the second embedding this this vector computed like with the following positions so minus x2 which is the negative value of the second dimension of the embedding of the vector x multiplied by this matrix here so there is nothing we have to learn in this matrix everything is fixed because if we watch the previous slide we can see that this theta actually is computed like this for one for each dimension and so there is nothing to learn so basically they are just like the absolute positional encoding so we computed them once and then we can reuse them for all the sentences that we will train the model upon another interesting property of the rotary positional embedding is the long term decay so what the others did they calculated an upper bound for the inner product that we saw before so the G function by varying the distance between the two tokens and then they



Rotary Position Embeddings: the computational-efficient form

- Given a token with embedding vector **x**, and the position **m** of the token inside the sentence, this is how we compute the position embeddings for the token.

3.4.2 Computational efficient realization of rotary matrix multiplication

Taking the advantage of the sparsity of $\boldsymbol{R}^d_{\Theta,m}$ in Equation (15), a more computational efficient realization of a multiplication of R^d_Θ and $\boldsymbol{x} \in \mathbb{R}^d$ is:

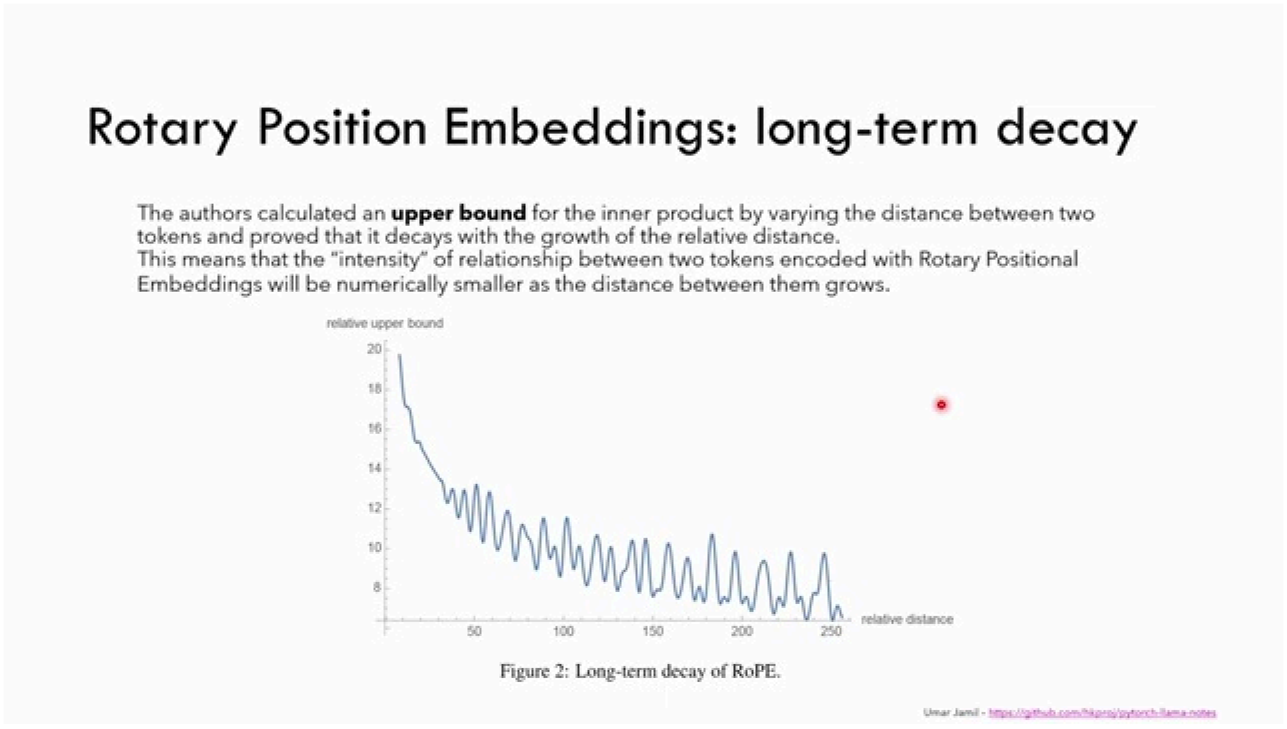
$$\boldsymbol{R}^d_{\Theta,m}\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix} \quad (34)$$

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 34

Time Range: 2085.04 s - 2147.04 s

proved that no matter what are the two tokens there is an upper bound that decreases as the distance between the two tokens grow and if you remember that the inner product or the dot product that we are computing is for the calculation of the attention this dot product represents the intensity of relationship between the two tokens for which we are computing the attention and what these rotary positional embeddings do they will basically decay this relationship this the strength of this relationship between the two tokens if the two tokens that we are matching are distant distance distance from them from each other and this is actually what we want so we want two words that are very far from each other to have less strong relationship and two words that are close to each other to have a stronger relationship and this is a desired property that we want from this rotary positional embeddings now the rotary positional embeddings are only apply to



Rotary Position Embeddings: long-term decay

The authors calculated an **upper bound** for the inner product by varying the distance between two tokens and proved that it decays with the growth of the relative distance.
This means that the "intensity" of relationship between two tokens encoded with Rotary Positional Embeddings will be numerically smaller as the distance between them grows.

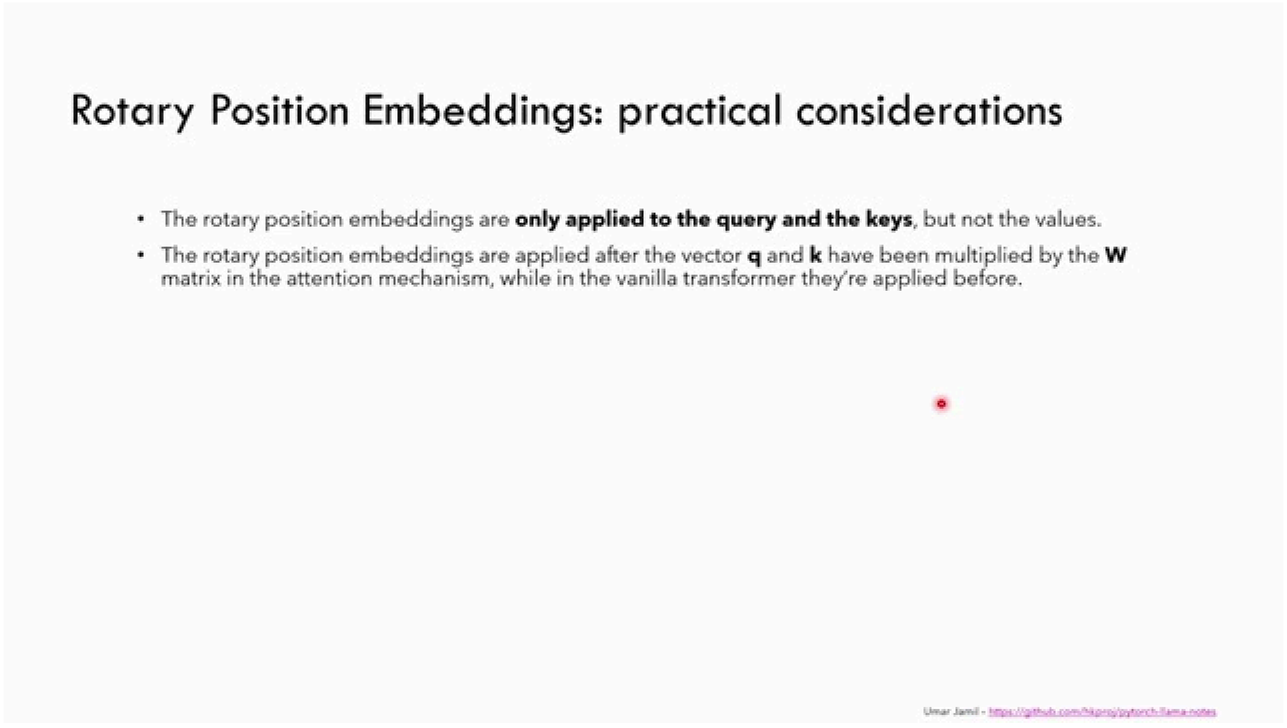
Figure 2: Long-term decay of RoPE.

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 35

Time Range: 2147.04 s - 2211.76 s

query and the keys but not the values let's see why well the first consideration is that they they basically they come into play when we are calculating the attention so when we calculated the attention it's the attention mechanism that will change the score so as you remember the attention mechanism is kind of a score that tells how much strong is the relationship between two tokens so this relationship will be stronger or less stronger or will change according to also the position of these two tokens inside of the centers and the relative distance between these two tokens another thing is that the rotary position embeddings are applied after the vector q and k have been multiplied by the w matrix in the attention mechanism while in the vanilla transformer they are applied before so in the vanilla transformer the position embeddings are applied right after we transform the tokens into embeddings but in the rotary positional embedding so in lama we



Rotary Position Embeddings: practical considerations

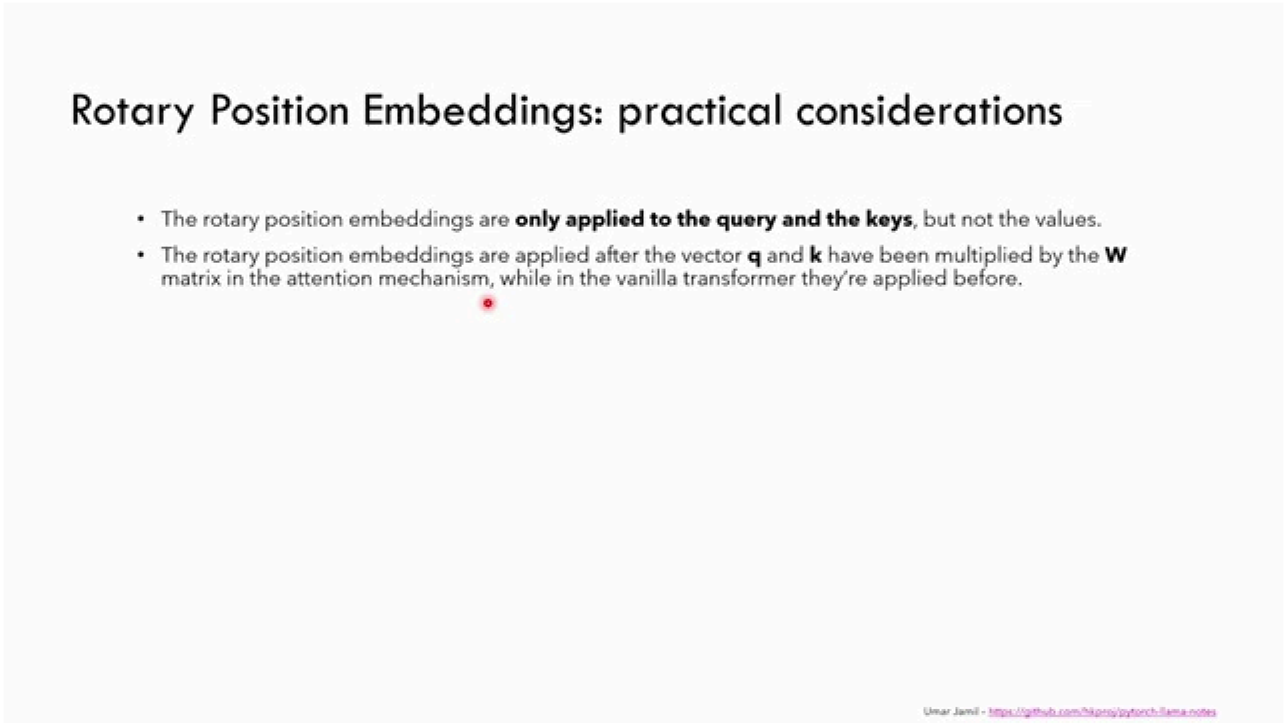
- The rotary position embeddings are **only applied to the query and the keys**, but not the values.
- The rotary position embeddings are applied after the vector **q** and **k** have been multiplied by the **W** matrix in the attention mechanism, while in the vanilla transformer they're applied before.

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 36

Time Range: 2211.76 s - 2281.28 s

don't do this we basically before right after we multiply by the w matrix in the attention mechanism so the w matrix if you remember is the matrix of parameters that each head has each attention head and so in the in the in the in the lama basically we apply the rotary position encoding after we multiply the vector skew and k by the w matrix now comes the interesting part in which we will watch how the self-attention works in lama but before we can talk about the self-attention as used in lama we need to review at least briefly the self-attention in the vanilla transformer so if you remember the self-attention in the vanilla transformer we start with the matrix skew which is a matrix of sequence by the model which means that we have on the rows the tokens and on the columns the dimensions of the embedding vector so we can think of it like the following let me okay so we can think of it like having six rows one and each of these rows is a vector



Rotary Position Embeddings: practical considerations

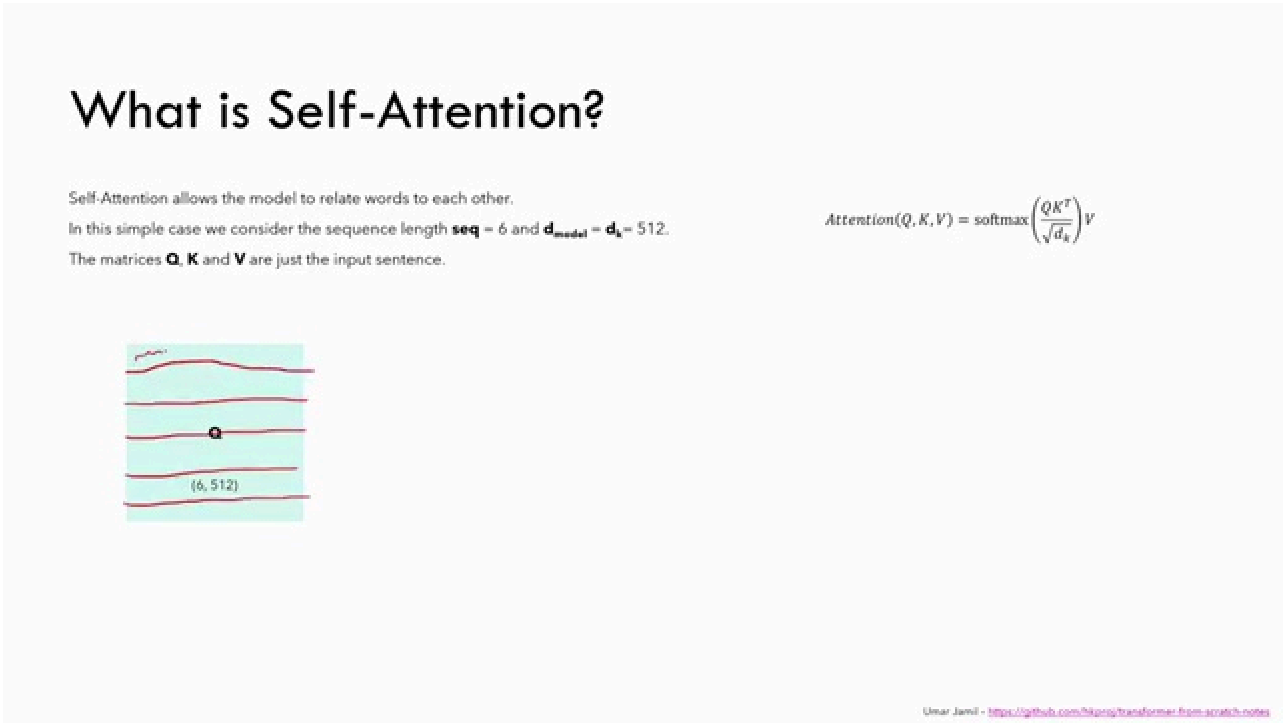
- The rotary position embeddings are **only applied to the query and the keys**, but not the values.
- The rotary position embeddings are applied after the vector **q** and **k** have been multiplied by the **W** matrix in the attention mechanism, while in the vanilla transformer they're applied before.

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 37

Time Range: 2281.28 s - 2344.16 s

of dimension 512 that represents the embedding of that token and now let me delete and then we multiply according to this formula so q multiplied by the transpose of the k so transpose of the k divide by the square root of 500l which is the dimension of the embedding vector where the k is equal to q and v is also equal to q because this is a self-attention so the three matrices are actually the same sequence then we apply the softmax and we obtain this matrix so we had the matrix that was 6 by 500l multiplied by another that is 512 by 6 we will obtain a matrix that is 6 by 6 where each item in this matrix represents the dot product of the first token with itself then the first token with the second token the first token with the third token the first token with the fourth token etc etc etc so this matrix captures the intensity of



What is Self-Attention?

Self-Attention allows the model to relate words to each other.

In this simple case we consider the sequence length **seq** = 6 and $\mathbf{d_{model}} = \mathbf{d_k} = 512$.

The matrices **Q**, **K** and **V** are just the input sentence.

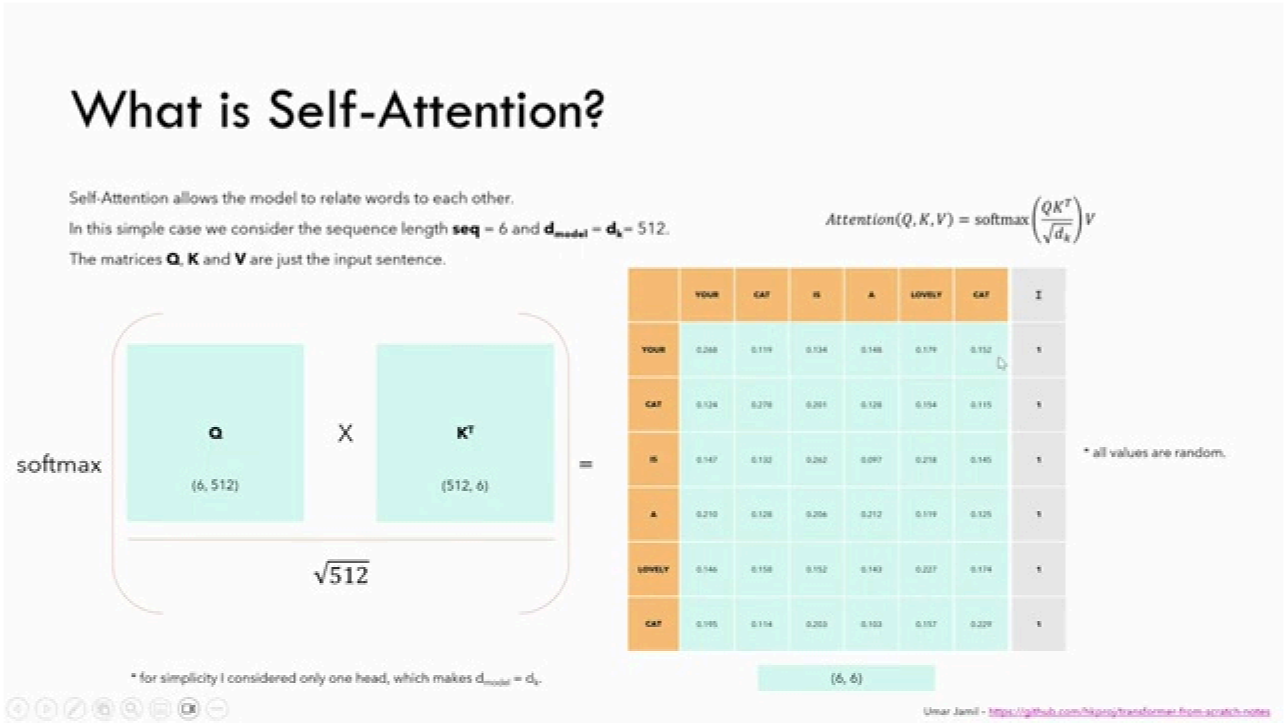
$$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Umar Jamil - https://github.com/hkproj/transformer-from-scratch-notes

Section 38

Time Range: 2344.16 s - 2406.8 s

relationship between two tokens then this the output of this softmax is multiplied by the v matrix to obtain the attention sequence so the output of the self-attention is another matrix that has the same dimensions as the initial matrix so it will produce a sequence where the embedding is now not only captured the meaning of each token not only the capture the position of each token but they also capture kind of the relationship between death token and every other token if you didn't understand this concept please go watch my previous video about the transformer explain it very carefully and in much more detail now let's have a look at the multi-head attention very briefly so the multi-head attention basically means that we have an input sequence we take it we copy it into qk and v so they are the same matrix we multiply by parameter matrices



What is Self-Attention?

Self-Attention allows the model to relate words to each other.

In this simple case we consider the sequence length **seq** = 6 and $\mathbf{d}_{\mathbf{model}} = \mathbf{d}_{\mathbf{k}} = 512$.

The matrices **Q**, **K** and **V** are just the input sentence.

$$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$$\text{softmax}\left(\frac{\underset{(6,\,512)}{Q} \times \underset{(512,\,6)}{K^T}}{\sqrt{512}}\right) =$$

	YOUR	CAT	IS	A	LOVELY	CAT	Σ
YOUR	0.268	0.119	0.134	0.148	0.179	0.152	1
CAT	0.124	0.278	0.201	0.128	0.154	0.115	1
IS	0.147	0.132	0.262	0.097	0.218	0.145	1
A	0.210	0.128	0.206	0.212	0.119	0.125	1
LOVELY	0.146	0.158	0.152	0.143	0.227	0.174	1
CAT	0.195	0.114	0.203	0.103	0.157	0.229	1

(6, 6)

* all values are random.

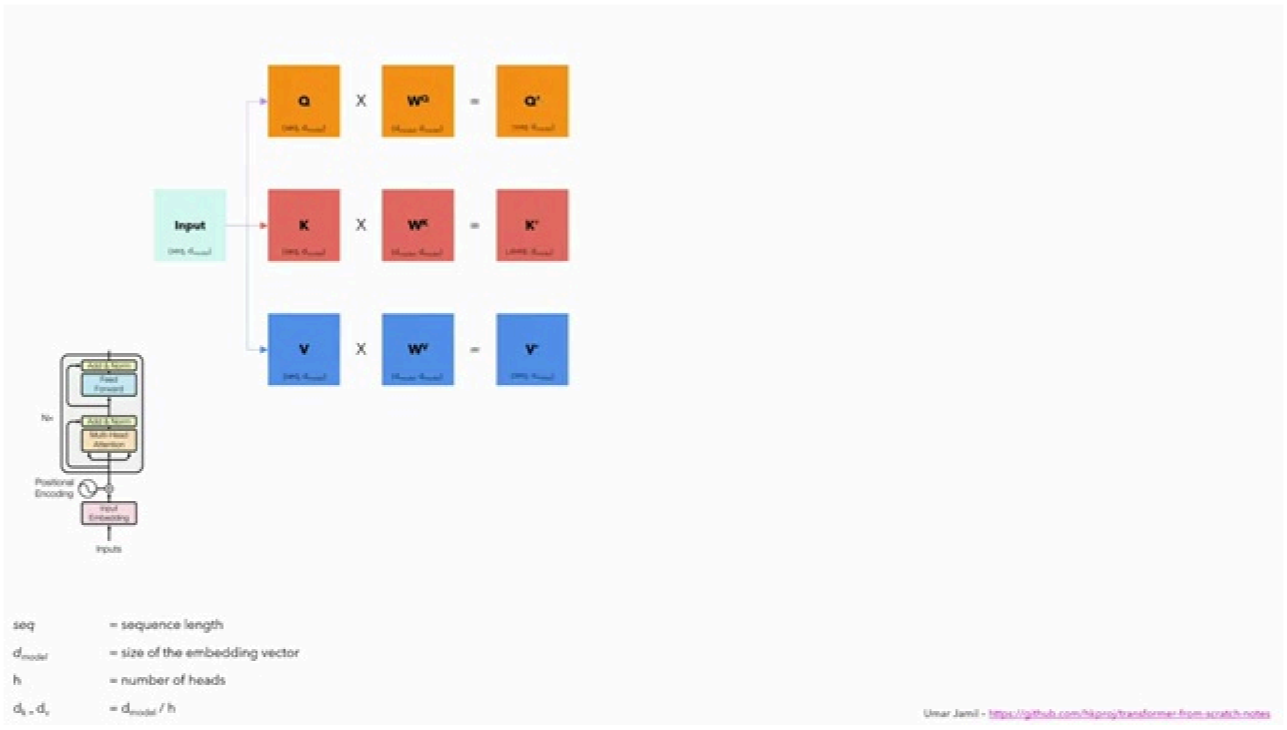
* for simplicity I considered only one head, which makes $d_{model} = d_k$.

Umar Jamil - https://github.com/hkproj/transformer-from-scratch-notes

Section 39

Time Range: 2406.8 s - 2471.2 s

and then we split into multiple smaller matrices one for each head and we calculated the attention between these heads so head one had two had three had four then we concatenate the output of these heads we multiply by the output matrix w o and finally we have the output of the multi-head attention let's look at what is the first kv cache so before we introduce the kv cache we need to understand how llama was trained and we need to understand what is the next token prediction task so llama just like most of the language large language models have been trained on the next token prediction task which means that given a sequence it will try to predict what is the next token the most likely next token to continue the prompt so for example if we tell him poem for example without the last word probably it will come up with the the last word that is missing from that poem in this case i will be using a one very famous passage from Dante Alighieri's



seq = sequence length

d_{model} = size of the embedding vector

h = number of heads

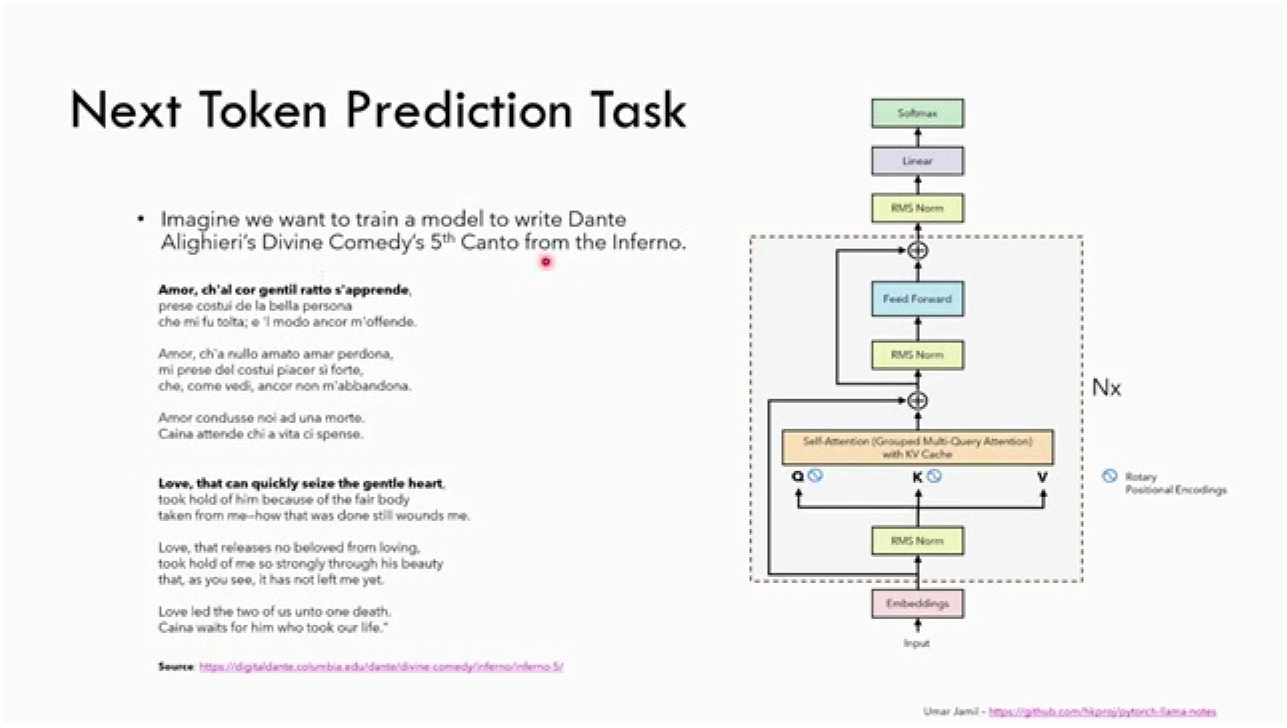
$d_k = d_v$ = d_{model} / h

Umar Jamil - https://github.com/hkproj/transformer-from-scratch-notes

Section 40

Time Range: 2471.2 s - 2532.4 s

and i will not use the Italian translation but we will use the English translation here so i will only deal with the first line you can see here love that can quickly see the gentle heart so let's train llama on this sentence how does the training work well we give the input to the model the input is built in such a way that we first prepare the start of sentence token and then the target is built such that we append an end of sentence token why because the model the transformer model is a sequence to sequence model which maps each position in the input sequence into another position into the in the output sequence so basically the first token of the input sequence will be mapped to the first token of the output sequence and a second scene probably the input supposed to elements with much more advertisement so an output sequence this also means give



Next Token Prediction Task

- Imagine we want to train a model to write Dante Alighieri's Divine Comedy's 5th Canto from the Inferno.

Amor, ch'al cor gentil ratto s'apprende,
prese costui de la bella persona
che mi fu tolta; e 'l modo ancor m'offende.

Amor, ch'a nullo amato amar perdona,
mi prese del costui piacer sì forte,
che, come vedi, ancor non m'abbandona.

Amor condusse noi ad una morte.
Caina attende chi a vita ci spense.

Love, that can quickly seize the gentle heart,
took hold of him because of the fair body
taken from me–how that was done still wounds me.

Love, that releases no beloved from loving,
took hold of me so strongly through his beauty
that, as you see, it has not left me yet.

Love led the two of us unto one death.
Caina waits for him who took our life."

Source: https://digitaldante.columbia.edu/dante/divine-comedy/inferno/inferno-5/

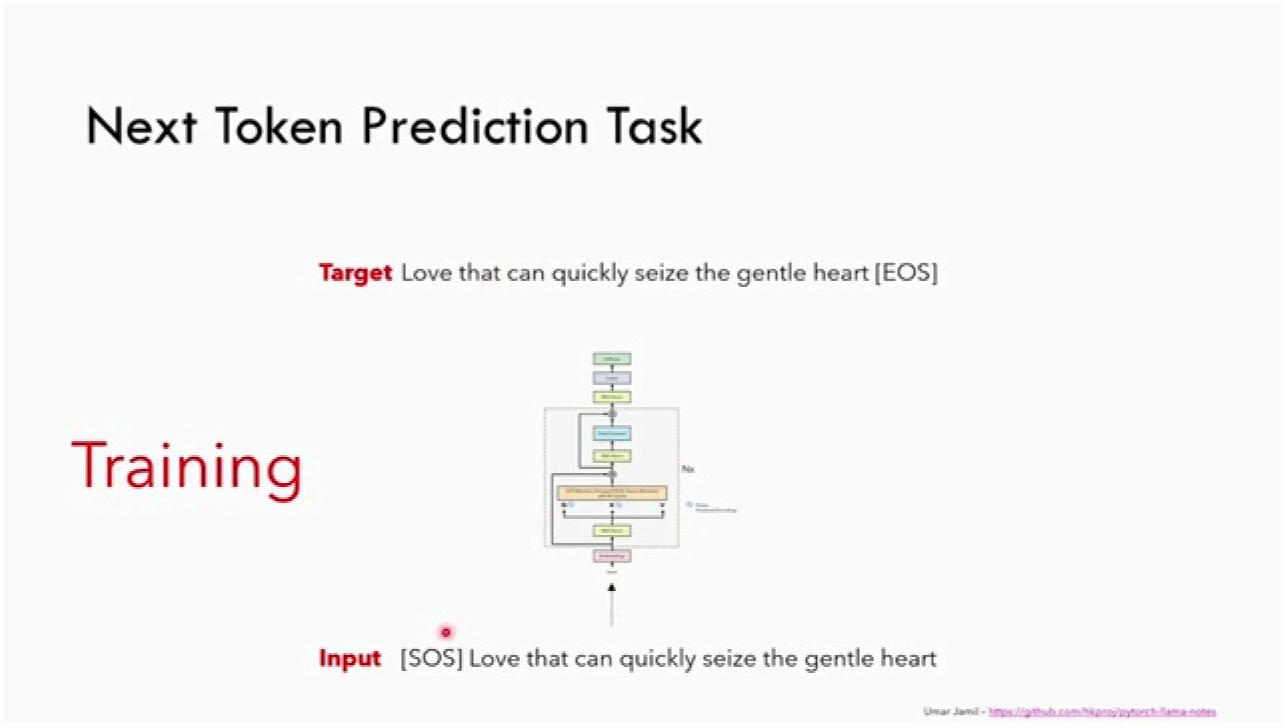
Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Softmax
Linear
RMS Norm
Feed Forward
RMS Norm
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
RMS Norm
Nx
Rotary Positional Encodings
Embeddings
Input

Section 41

Time Range: 2532.4 s - 2596.48 s

transmission for A fabulous life we here the input tokens it will produce the second token as output so love that and if we give the first three tokens it will produce the output the third token as output of course the model will also produce the output for the previous two tokens but we let's see it with an example so if you remember from my previous video also in which I do the inferencing when we train the model we only do it in one step so we give the input we give the target we calculate the loss and we don't have any for loop to train the model for the one single sentence but for the inference we need to do it token by token so in the in the in this inferencing we start with the time step the time stamp time step one in which we only give the input SOS so start of sentence and the output is love then we take the



Next Token Prediction Task

Target Love that can quickly seize the gentle heart [EOS]

Training

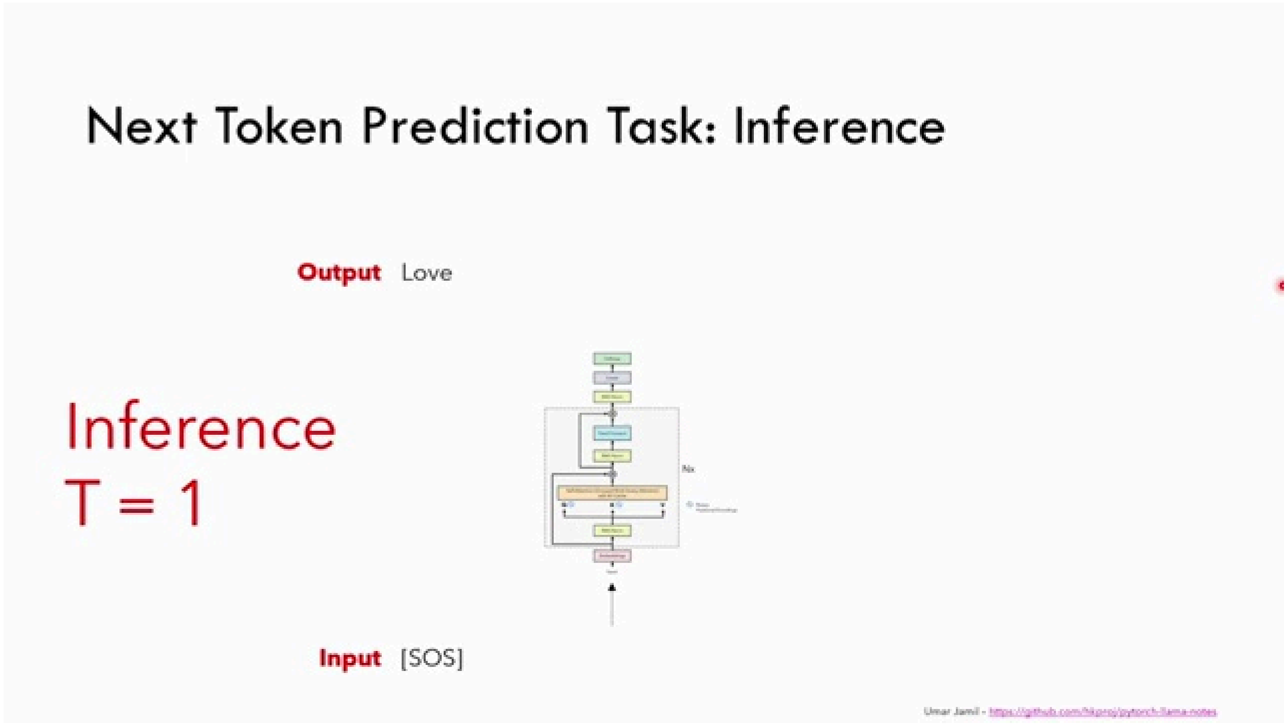
Input [SOS] Love that can quickly seize the gentle heart

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 42

Time Range: 2596.48 s - 2660.4 s

output token here love and we append it to the input and we give it again to the model and the model will produce the next token love that then we take the last token output by the model that we append it again to the input and the model will produce the next token and then again take the next token so can we append it to the input and we feed it again to the model and the model will output the next token quickly and we do it for all the steps that are necessary until we reach the end of sentence token then that's when we know that the model has finished outputting its output. Now this is not how Lama was trained actually but this is a good example to show you how the next token prediction task works. Now this is there is a there is a problem with this approach. Let's see why. At every step of the inference we are only interested in the last token



Next Token Prediction Task: Inference

Output Love

Inference
T = 1

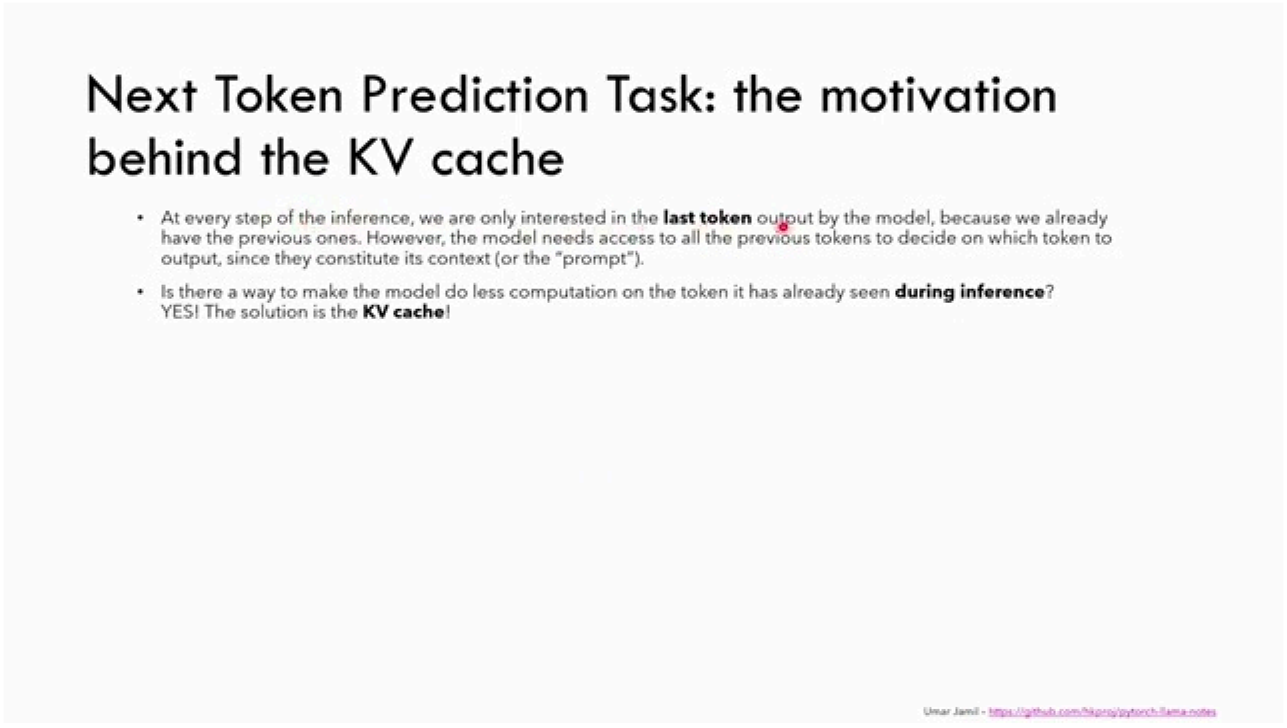
Input [SOS]

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 43

Time Range: 2660.4 s - 2722.64 s

output by the model because we already have the previous ones. However the model needs to access to all the previous tokens to decide on which token to output since they constitute its context or the prompt. So what I mean by this is that to output for example the word D the model has to see all the input here. We cannot just give the C's. The model needs to see all the input to output this last token D but the point is this is a sequence to sequence model so it will produce this sequence as output even if we only care about the last token. So there is a lot of unnecessary computation we are doing to calculate these tokens again that we already actually have from the previous time steps. So let's find a way to not to do this useless computation and this is what we do with the KV cache. So the KV cache is a way to do less computation on the tokens that we have already seen during inferencing so it's only applied during



Next Token Prediction Task: the motivation behind the KV cache

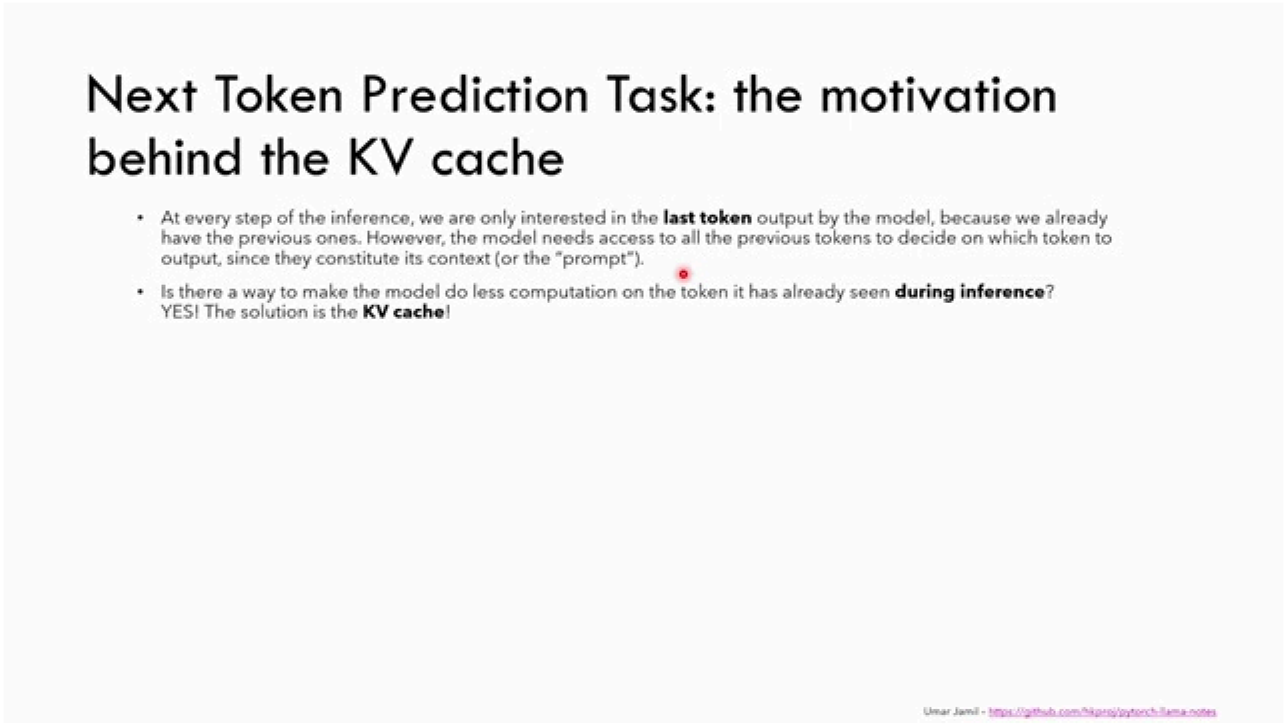
- At every step of the inference, we are only interested in the **last token** output by the model, because we already have the previous ones. However, the model needs access to all the previous tokens to decide on which token to output, since they constitute its context (or the "prompt").
- Is there a way to make the model do less computation on the token it has already seen **during inference**? YES! The solution is the **KV cache**!

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 44

Time Range: 2722.64 s - 2786.76 s

inferencing in a transformer model and it's not only applies to the transformer of the like the one in lava but to all transformer models because all transformer models work in this way. This is a description it's a picture of how the self-attention works during the next token prediction task. So as you saw in also in my previous slides we have a query matrix here with end tokens then we have the transpose of the keys so the query can be thought as rows of vectors where the first vector represents the first token, the second token etc. Then the transpose of the keys is the same tokens but transpose so the rows become columns. This produces a matrix that is n by n so if the initial input matrix is 9 the output matrix will be 9 by 9 then we multiply it by the V matrix and this will produce the attention. The attention is then fed to the linear layer of the transformer then the linear layer will produce the



Next Token Prediction Task: the motivation behind the KV cache

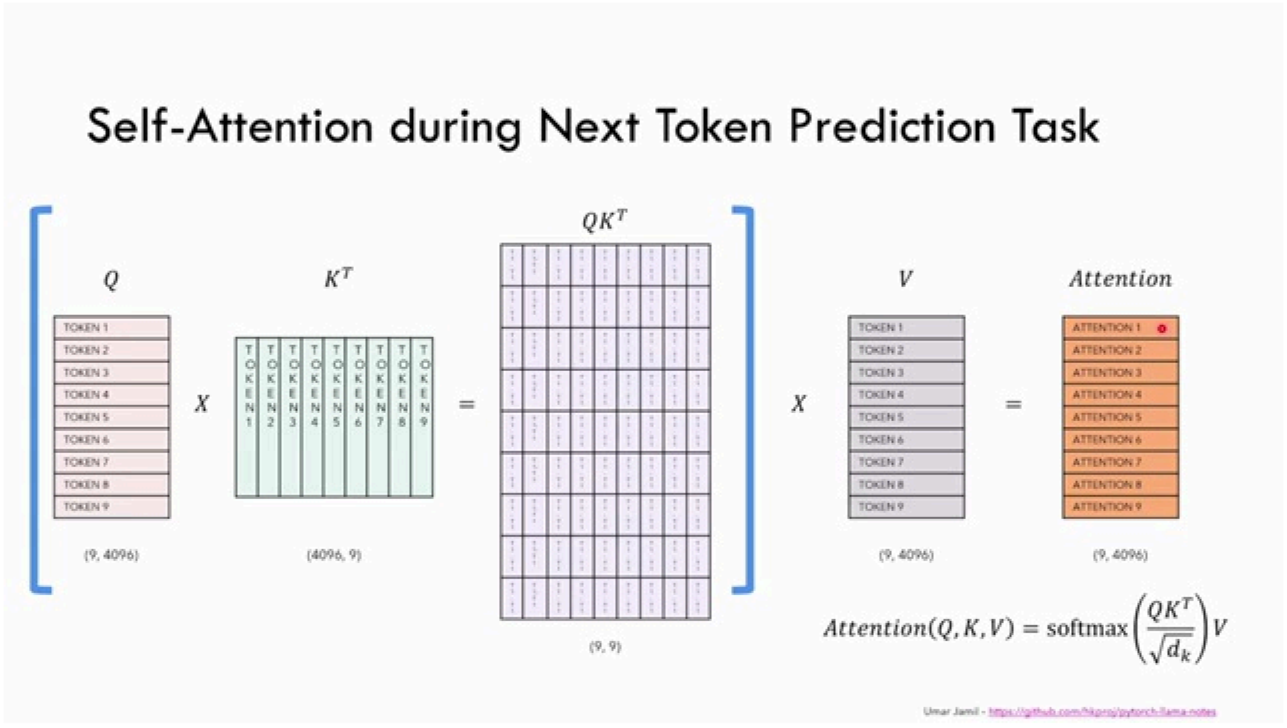
- At every step of the inference, we are only interested in the **last token** output by the model, because we already have the previous ones. However, the model needs access to all the previous tokens to decide on which token to output, since they constitute its context (or the "prompt").
- Is there a way to make the model do less computation on the token it has already seen **during inference**? YES! The solution is the **KV cache**!

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 45

Time Range: 2786.76 s - 2849.6 s

logits and the logits are fed to the softmax and the softmax allows us to decide which is the token from our vocabulary. Again if you're not familiar with this please watch my previous video of the transformer about the inferencing of the transformer and you will see this clearly. So this is a description of what happens at the general level in the self-attention. Now let's watch it step by step. So imagine at the inference step one we only have the first token if you remember before we are we are only using the start of sentence token so we take the start of sentence token we multiply it by itself so the transpose it will produce a matrix that is 1 by 1 so this matrix is 1 by 4,096 multiply by another matrix that is 4,096 by 1 it will produce a 1 by 1 matrix y4,096 because the embedding vector in Lama is 4,096 then the output so this 1 by 1 is multiplied by the V and it will produce the output token

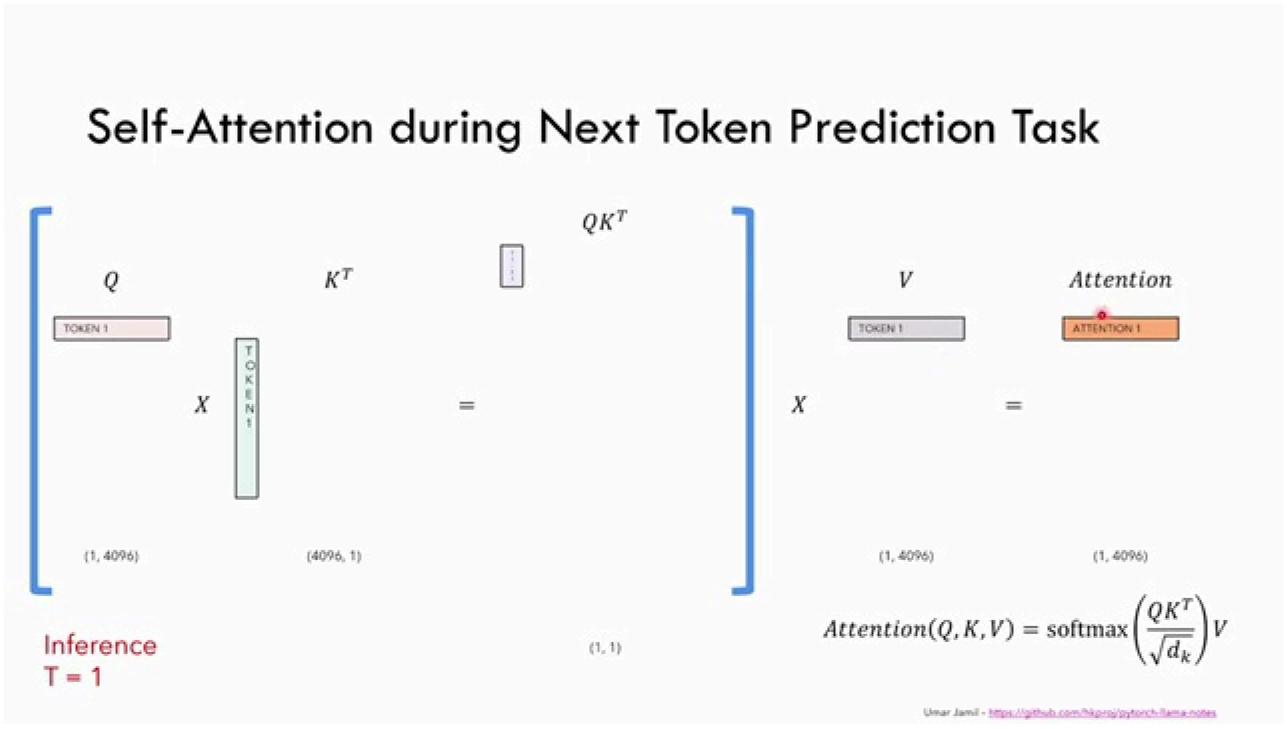

Self-Attention during Next Token Prediction Task
Q
TOKEN 1 … TOKEN 9
(9, 4096)
X
K^T
TOKEN 1 … TOKEN 9
(4096, 9)
=
QK^T
(9, 9)
X
V
TOKEN 1 … TOKEN 9
(9, 4096)
=
Attention
ATTENTION 1 … ATTENTION 9
(9, 4096)
Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

$$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Section 46

Time Range: 2849.6 s - 2909.68 s

here and this is will be our first token of the output and then we take the output token this one and we append it to the input at the next step so now we have two tokens as input they are multiplied by itself but with the transposed version of itself and it will produce a 2 by 2 matrix which is then multiplied by the V matrix and it will produce two output tokens but we are only interested in the last tokens output by the model so this one attention to which is then appended to the input matrix at the time steps three so now we have three tokens in the time step three which are multiplied by the transposed version of itself and it will produce a 3 by 3 matrix which is then multiplied by the V matrix and we have this three matrix territory tokens as output but we are only interested in the last token output by the model so we append it again as input to the Q matrix which is now



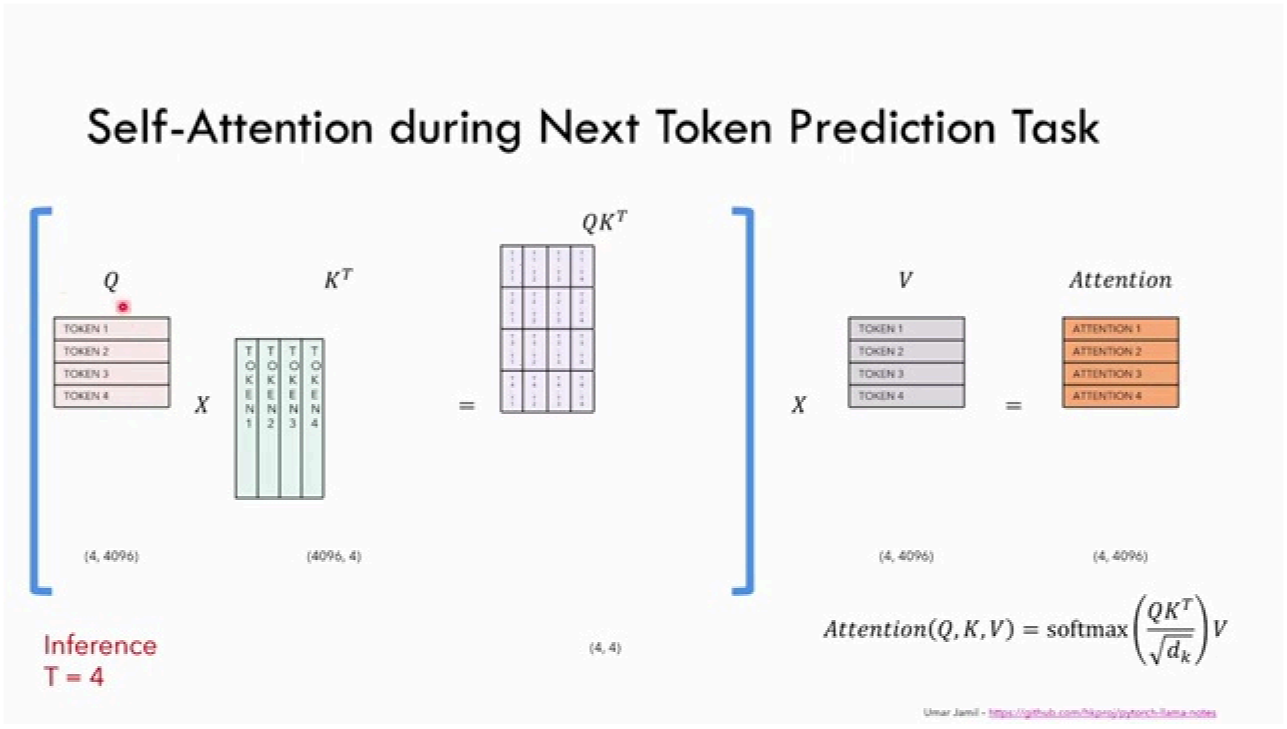
Self-Attention during Next Token Prediction Task

$$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Section 47

Time Range: 2909.68 s - 2969.92 s

4 tokens which is multiplied by the transposed version of itself and it will produce a 4 by 4 matrix as output which is then multiplied by this matrix here and it will produce this attention matrix here but we are all interested in the last attention which will be then added again to the input of the next step but we notice already something first of all we are ready here in this matrix where we compute the dot product between this token and this this token and this this token and this so this matrix is the all the dot products between these two matrices we can see something the first thing is that we are ready computed these dot products in the previous step can we catch them so let's go back as you can see this matrix is growing two three four see there is a lot of attention because we are every time we are influencing the transformer we are giving him giving the transform some input so it's

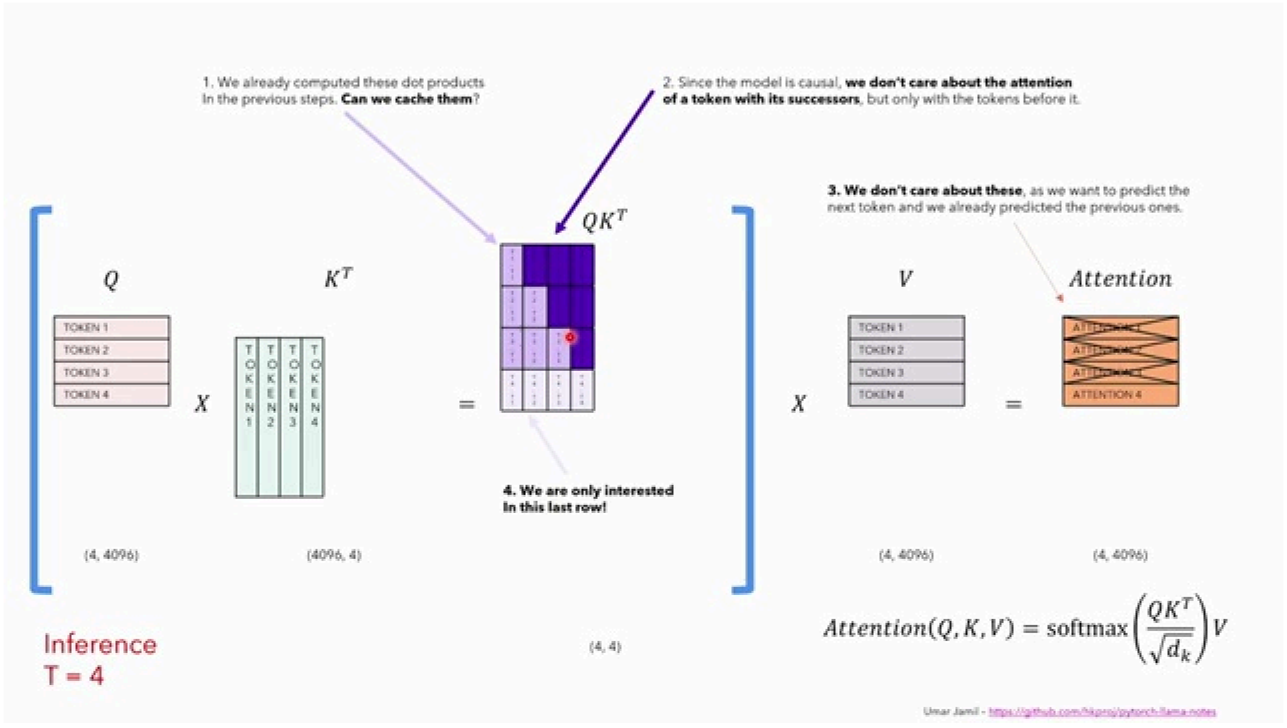


$$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Section 48

Time Range: 2969.92 s - 3031.36 s

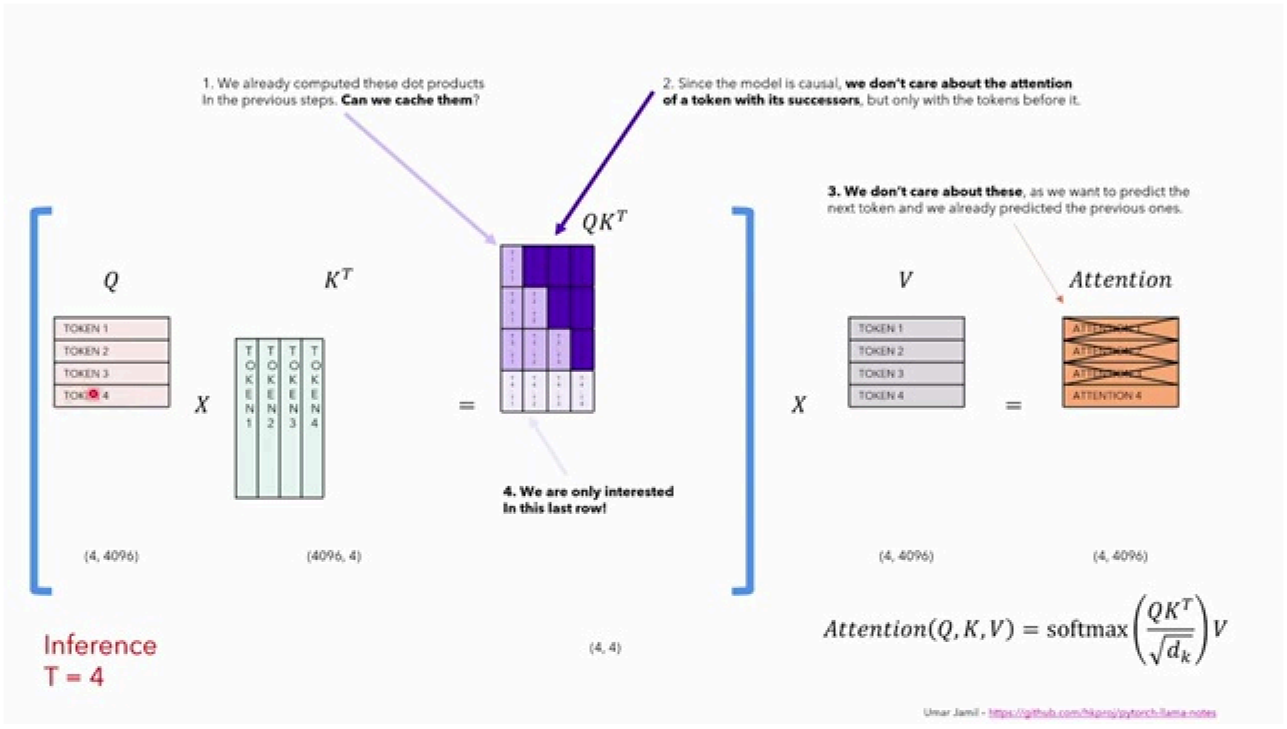
recomputing all these dot products which is inconvenient because we actually already computed them in the previous time step so is there a way to not compute them again can we kind of catch them yes we can and then since the model is causal we don't care about the attention of a token with its predecessors but only with the token before it so as you remember in the self attention we apply a mask right so the mask is basically we don't want the dot product of one word with the word that come after it but only the one that come before it so basically we don't want all the numbers above the principal principal diagonal of this matrix and that's why we applied the mask in the self-attention but okay the point is we don't need to compute all these dot products the only dot products that we are interested in is this last row so because we added the token for as input compared to the last time step


1. We already computed these dot products in the previous steps. **Can we cache them?**
2. Since the model is causal, **we don't care about the attention of a token with its successors**, but only with the tokens before it.
3. **We don't care about these**, as we want to predict the next token and we already predicted the previous ones.
4. **We are only interested in this last row!**
Q
K^T
QK^T
V
$Attention$
TOKEN 1
TOKEN 2
TOKEN 3
TOKEN 4
ATTENTION 4
(4, 4096)
(4096, 4)
(4, 4)
Inference
T = 4
$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 49

Time Range: 3031.36 s - 3093.52 s

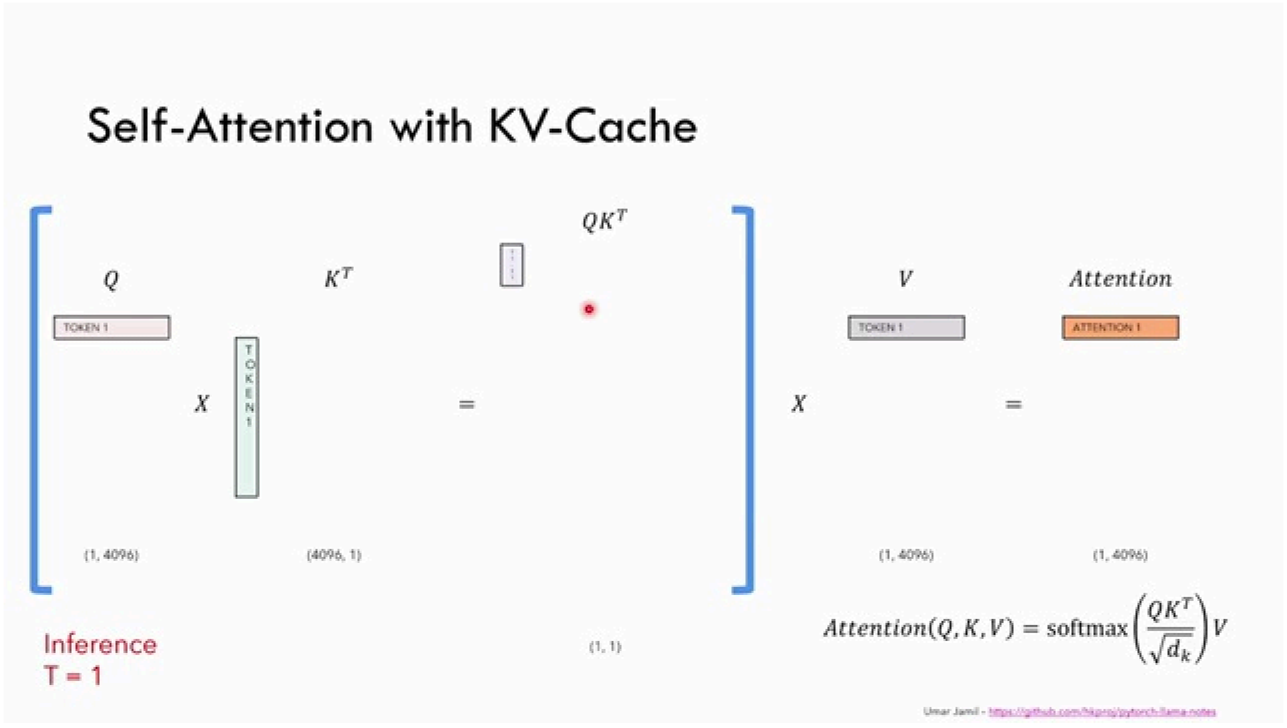
so we only have this new token token for and we want this token for how it is interacting with all the other tokens so basically we are only interested in this last row here and also as we only care about the attention of the last token because we want to select the word from the vocabulary so we only care about the last row we don't care about producing these two these three attention score here in the output sequence of the self-attention we only care about the last one so is there a way to remove all these redundant calculations yes we can do it with the KV cache let's see how so with the KV cache basically what we do is we cache the query so sorry the keys and the values and every time we have a new token we append it to the key and the values while the query is only the output of the previous step so at the


1. We already computed these dot products in the previous steps. **Can we cache them?**
2. Since the model is causal, **we don't care about the attention of a token with its successors**, but only with the tokens before it.
3. **We don't care about these**, as we want to predict the next token and we already predicted the previous ones.
4. **We are only interested in this last row!**
Q
K^T
QK^T
V
$Attention$
TOKEN 1
TOKEN 2
TOKEN 3
TOKEN 4
ATTENTION 4
(4, 4096)
(4096, 4)
(4, 4)
(4, 4096)
(4, 4096)
Inference
T = 4
$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 50

Time Range: 3093.52 s - 3154.28 s

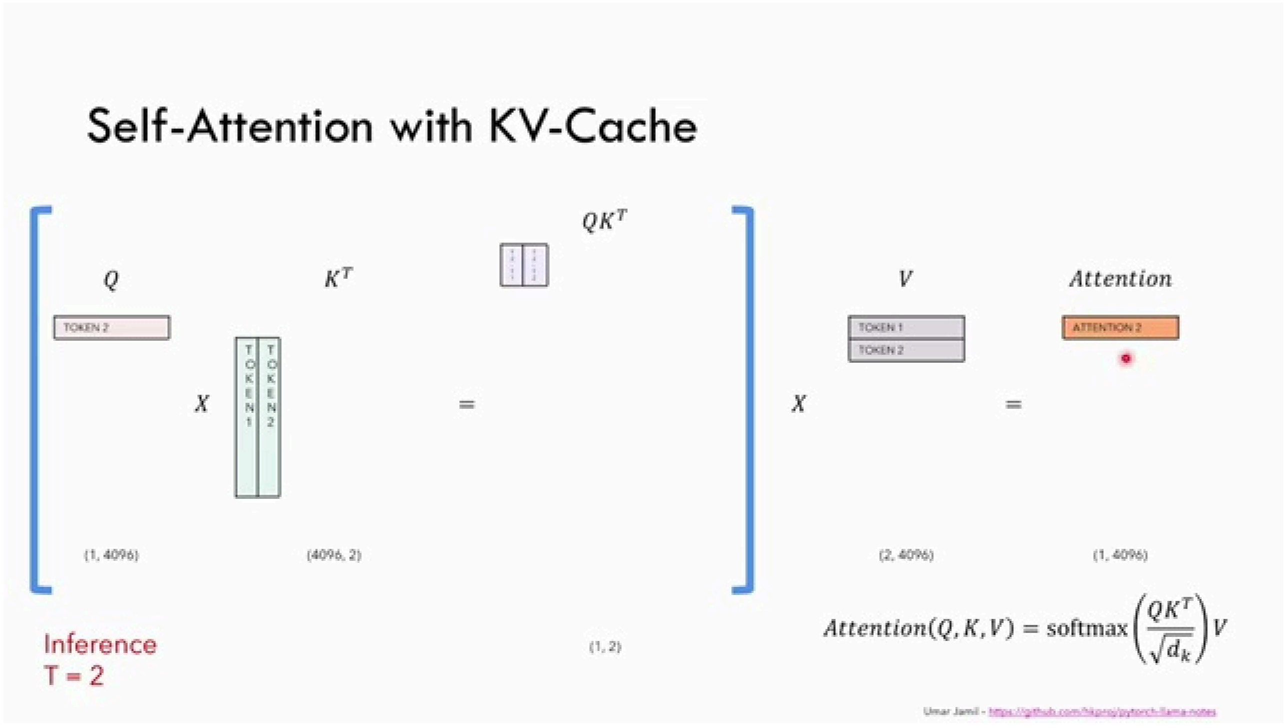
beginning we don't have any output from the previous step so we only use the first token so the first the time step one of the inference is the same as without the cache so we have the token one with itself we'll produce a matrix one by one multiply with one token and if you produce one attention however at the time step two we don't append it to the previous query we just replaced the previous token with the new token we have here however we keep the cache of the keys so we keep the previous token in the keys and we append the last output to the keys here and also to the values and if you and if you do this multiplication it will produce a matrix that is one by two where the first item is the dot product of the token two with the token one and the token two with the token two this is actually what we want and if we then multiply with the V matrix it will only produce one attention score which is


Self-Attention with KV-Cache
Q
K^T
QK^T
V
$Attention$
TOKEN 1
TOKEN 1
TOKEN 1
ATTENTION 1
X
=
X
=
(1, 4096)
(4096, 1)
(1, 4096)
(1, 4096)
(1, 1)
Inference
T = 1
$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 51

Time Range: 3154.28 s - 3217.52 s

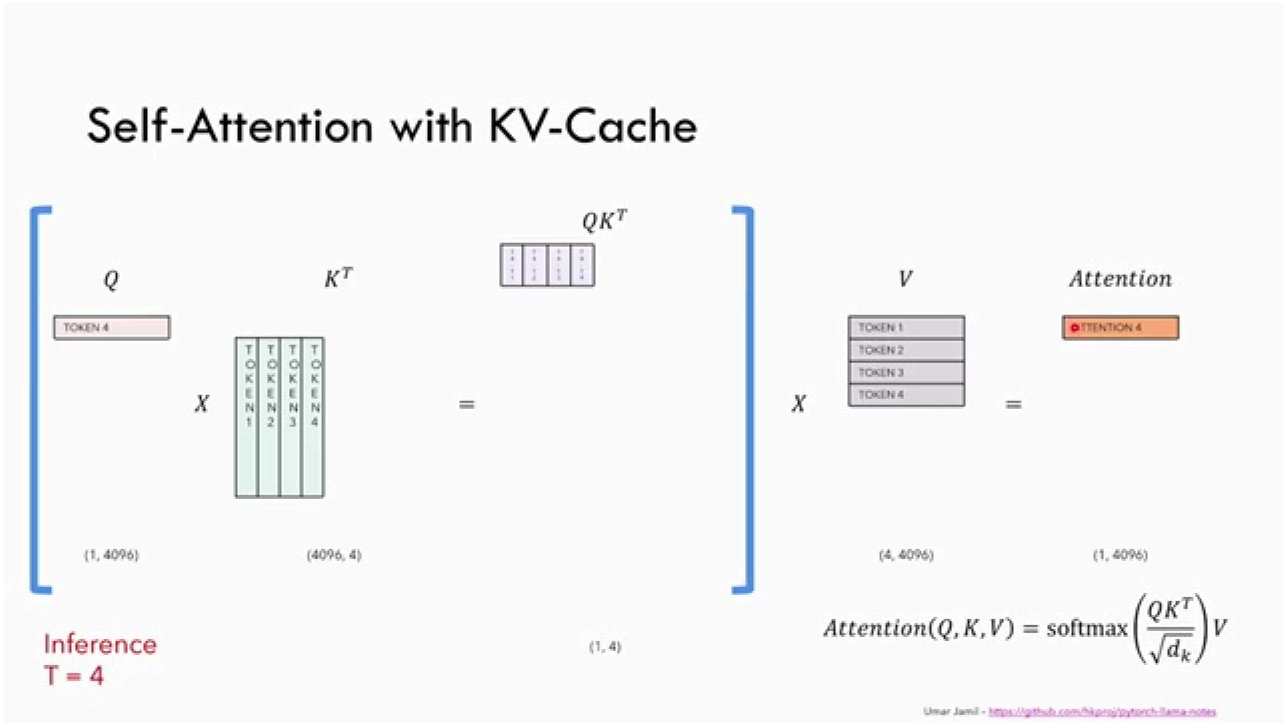
exactly the one we want and we do again so we take this attention to and this will become the input of the next inference type so this token three we append it to the previous leakage the K matrix and also to the previous leakage V matrix this multiplication will produce an output matrix that we can see here the multiplication of this output matrix with this V matrix will produce one token in the output which is this one and we know which token to select using this one then we use it as an input for the next inferencing step by appending it to the cached keys and appending to the cached V matrix we do this multiplication and we will get this matrix which is four one by four which is the dot product of the token four with the token one the token four with the token two token four with the token three and the token four with itself we multiply by the V matrix and this will only produce one attention which is


Self-Attention with KV-Cache
Q
TOKEN 2
X
K^T
TOKEN 1
TOKEN 2
=
QK^T
X
V
TOKEN 1
TOKEN 2
=
Attention
ATTENTION 2
(1, 4096)
(4096, 2)
(2, 4096)
(1, 4096)
(1, 2)
Inference
T = 2
$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 52

Time Range: 3217.52 s - 3286.6 s

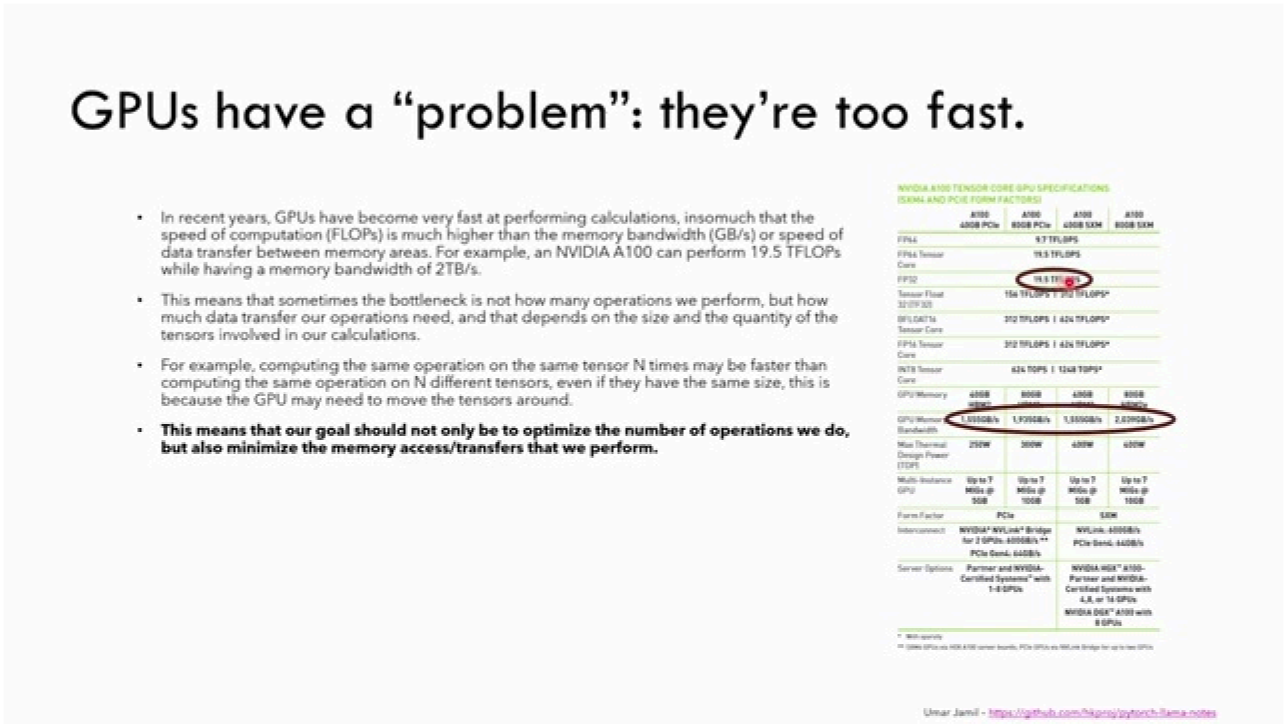
exactly what we want to select the output token this is the reason why it's called the KV cache because we are keeping a cache of the keys and the values as you can see the KV cache allows us to save a lot of computation because we are not doing a lot of dot products that were used to do that we used to do before and this makes the inferencing faster the next layer that we will be talking about is the grouped multi query attention but before we talk about the grouped multi query attention we need to introduce its predecessor the multi query attention let's see so let's start with the problem the problem is that the GPUs are too fast if you watch this data sheet this is from the A1 GPU from Nvidia we can see that the GPU is very fast at computing at performing calculations but not so much not so fast at transferring data from its memory that means for example that the the A100 can do 19.5 theta floating point



Section 53

Time Range: 3286.6 s - 3350.64 s

operations per second by using a 32-bit precision while it can only trust for 1.9 thousand gigabytes per second it's nearly ten times slower more slower to transferring data than it is at performing calculations and this means that sometimes the bottleneck is not how many operations we perform but how much data transfer our operations need and that depends on the size and the quantity of the tensors involved in our calculations for example if we compute the same operations on the same tensor and times it may be faster than computing the same operations on end different tokens even if they have the same size this is because the GPU may need to move these tensors around so this means that our goal should not only to be be to optimize the number of operations we do with our algorithms but also minimize the memory access and



GPUs have a "problem": they're too fast.

- In recent years, GPUs have become very fast at performing calculations, insomuch that the speed of computation (FLOPs) is much higher than the memory bandwidth (GB/s) or speed of data transfer between memory areas. For example, an NVIDIA A100 can perform 19.5 TFLOPs while having a memory bandwidth of 2TB/s.
- This means that sometimes the bottleneck is not how many operations we perform, but how much data transfer our operations need, and that depends on the size and the quantity of the tensors involved in our calculations.
- For example, computing the same operation on the same tensor N times may be faster than computing the same operation on N different tensors, even if they have the same size, this is because the GPU may need to move the tensors around.
- **This means that our goal should not only be to optimize the number of operations we do, but also minimize the memory access/transfers that we perform.**

NVIDIA A100 TENSOR CORE GPU SPECIFICATIONS (SXM4 AND PCIE FORM FACTORS)

	A100 40GB PCIe	A100 80GB PCIe	A100 40GB SXM	A100 80GB SXM
FP64	9.7 TFLOPS			
FP64 Tensor Core	19.5 TFLOPS			
FP32	19.5 TFLOPS			
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*			
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
INT8 Tensor Core	624 TOPS | 1248 TOPS*			
GPU Memory	40GB HBM2	80GB HBM2e	40GB HBM2	80GB HBM2e
GPU Memory Bandwidth	1,555GB/s	1,935GB/s	1,555GB/s	2,039GB/s
Max Thermal Design Power (TDP)	250W	300W	400W	400W
Multi-Instance GPU	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB
Form Factor	PCIe		SXM	
Interconnect	NVIDIA® NVLink® Bridge for 2 GPUs: 600GB/s ** PCIe Gen4: 64GB/s		NVLink: 600GB/s PCIe Gen4: 64GB/s	
Server Options	Partner and NVIDIA-Certified Systems™ with 1-8 GPUs		NVIDIA HGX™ A100-Partner and NVIDIA-Certified Systems with 4, 8, or 16 GPUs NVIDIA DGX™ A100 with 8 GPUs	

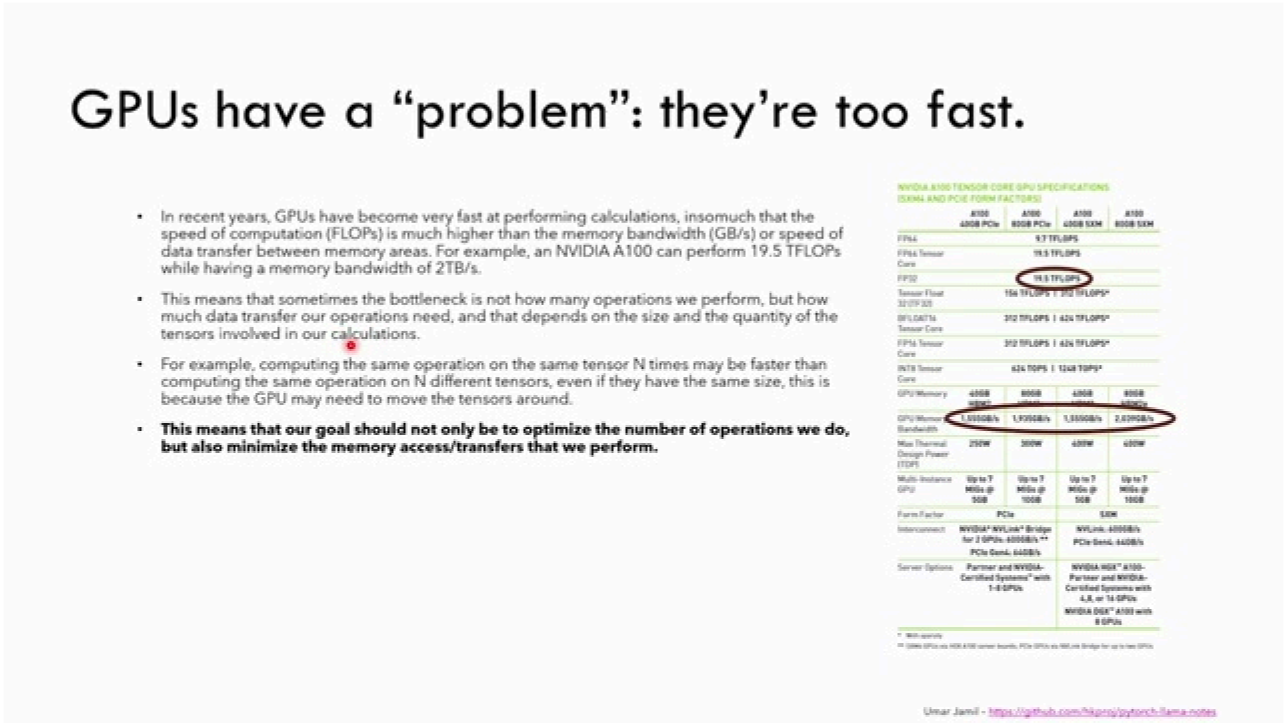
* With sparsity
** SXM4 GPUs via HGX A100 server boards; PCIe GPUs via NVLink Bridge for up to two GPUs

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 54

Time Range: 3350.64 s - 3412.76 s

the memory transfers that our algorithms perform because the memory access and memory transfer are more expensive in terms of time compared to the computations and this is also happens with software when we do IO for example if we copy for example we do some multiplications in the CPU or we read some data from the hard disk reading from the hard disk is much more slower than doing a lot of computations on the CPU and this is a problem now in this paper we introduce the Multiquery Attention this paper is from Noam Shahzir who is also one of the authors of the attention paper so attention is all you need and in this paper he introduced the problem he said well let's look at the Multihead Attention so the batched Multihead Attention this is the Multihead Attention as presented in the original paper attention is all you need let's look at the algorithm and let's calculate the number of arithmetic



GPUs have a "problem": they're too fast.

- In recent years, GPUs have become very fast at performing calculations, insomuch that the speed of computation (FLOPs) is much higher than the memory bandwidth (GB/s) or speed of data transfer between memory areas. For example, an NVIDIA A100 can perform 19.5 TFLOPs while having a memory bandwidth of 2TB/s.
- This means that sometimes the bottleneck is not how many operations we perform, but how much data transfer our operations need, and that depends on the size and the quantity of the tensors involved in our calculations.
- For example, computing the same operation on the same tensor N times may be faster than computing the same operation on N different tensors, even if they have the same size, this is because the GPU may need to move the tensors around.
- **This means that our goal should not only be to optimize the number of operations we do, but also minimize the memory access/transfers that we perform.**

NVIDIA A100 TENSOR CORE GPU SPECIFICATIONS (SXM4 AND PCIE FORM FACTORS)

	A100 40GB PCIe	A100 80GB PCIe	A100 40GB SXM	A100 80GB SXM
FP64	9.7 TFLOPS			
FP64 Tensor Core	19.5 TFLOPS			
FP32	19.5 TFLOPS			
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*			
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
INT8 Tensor Core	624 TOPS | 1248 TOPS*			
GPU Memory	40GB HBM2	80GB HBM2e	40GB HBM2	80GB HBM2e
GPU Memory Bandwidth	1,555GB/s	1,935GB/s	1,555GB/s	2,039GB/s
Max Thermal Design Power (TDP)	250W	300W	400W	400W
Multi-Instance GPU	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB
Form Factor	PCIe		SXM	
Interconnect	NVIDIA® NVLink® Bridge for 2 GPUs: 600GB/s ** PCIe Gen4: 64GB/s		NVLink: 600GB/s PCIe Gen4: 64GB/s	
Server Options	Partner and NVIDIA-Certified Systems™ with 1-8 GPUs		NVIDIA HGX™ A100-Partner and NVIDIA-Certified Systems with 4,8, or 16 GPUs NVIDIA DGX™ A100 with 8 GPUs	

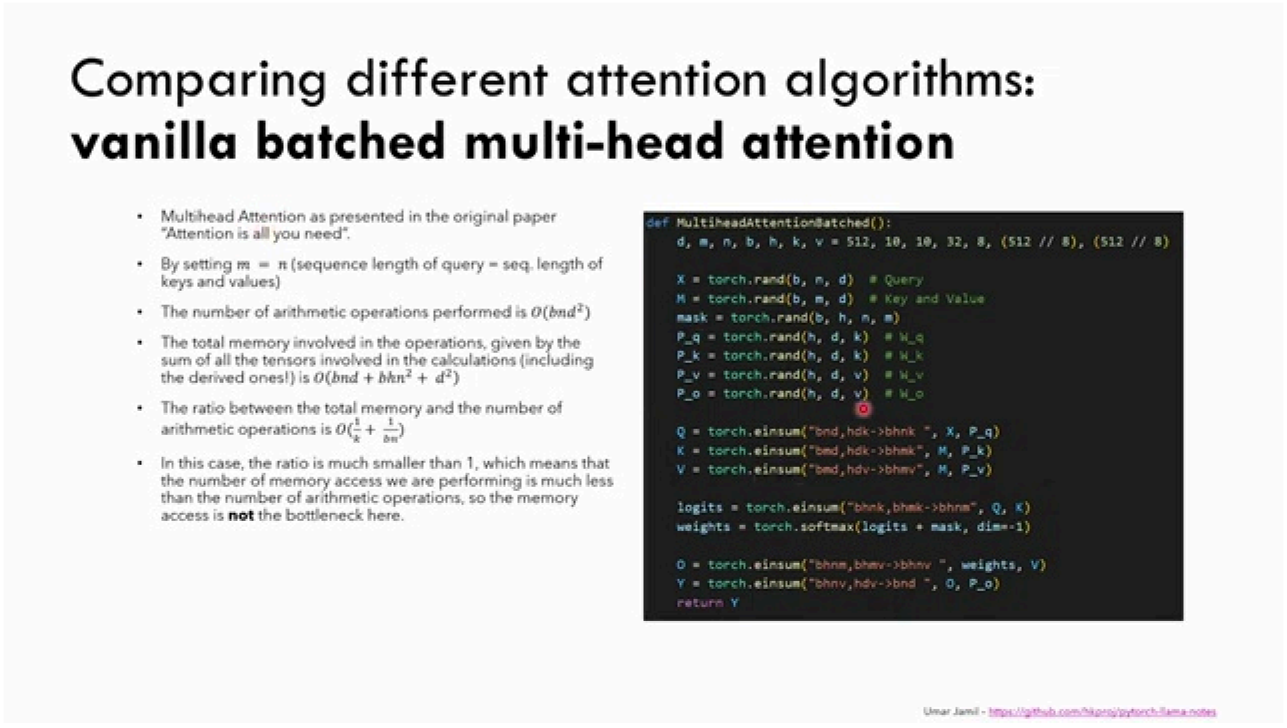
* With sparsity
** SXM4 GPUs via HGX A100 server boards; PCIe GPUs via NVLink Bridge for up to two GPUs

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 55

Time Range: 3412.76 s - 3476.12 s

operations performed and also the total memory involved in this operations so he calculated that the number of arithmetic operations is performed in O1 O of B and D squared where B is the batch size and is the sequence length and D is the size of the embedding vector while the total memory involved in the operations given by the sum of all the tensors involved in the calculations including the derived ones is equal to O of B and D plus B H and squared where H is the number of heads in this Multihead Attention plus D squared now if we compute the ratio between the total memory and the number of arithmetic operations we get this expression here 1 over K plus 1 over B in this case the ratio is much smaller than 1 which means that the number of memory access that we perform is much less than the number of arithmetic operations so the memory



Comparing different attention algorithms: **vanilla batched multi-head attention**

- Multihead Attention as presented in the original paper "Attention is all you need".
- By setting $m = n$ (sequence length of query = seq. length of keys and values)
- The number of arithmetic operations performed is $O(bnd^2)$
- The total memory involved in the operations, given by the sum of all the tensors involved in the calculations (including the derived ones!) is $O(bnd + bhn^2 + d^2)$
- The ratio between the total memory and the number of arithmetic operations is $O(\frac{1}{k} + \frac{1}{bn})$
- In this case, the ratio is much smaller than 1, which means that the number of memory access we are performing is much less than the number of arithmetic operations, so the memory access is **not** the bottleneck here.

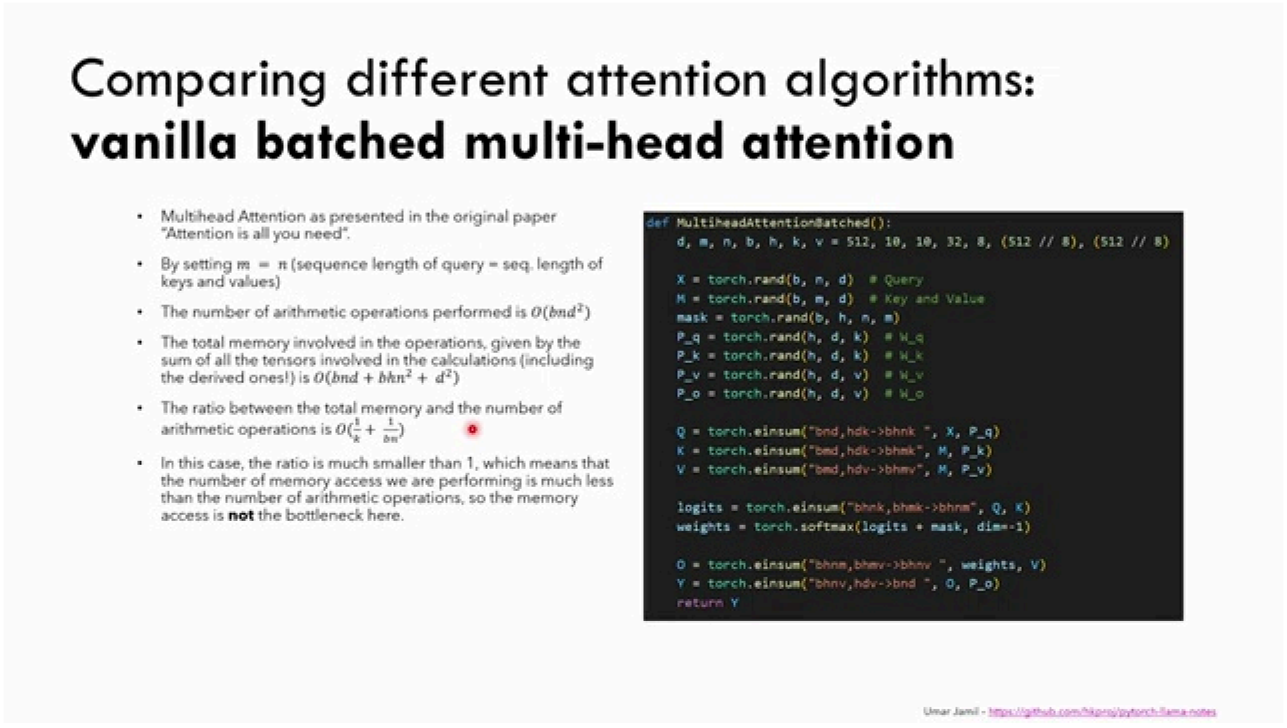
```
def MultiheadAttentionBatched():
    d, m, n, b, h, k, v = 512, 10, 10, 32, 8, (512 // 8), (512 // 8)

    X = torch.rand(b, n, d)  # Query
    M = torch.rand(b, m, d)  # Key and Value
    mask = torch.rand(b, h, n, m)
    P_q = torch.rand(h, d, k)  # W_q
    P_k = torch.rand(h, d, k)  # W_k
    P_v = torch.rand(h, d, v)  # W_v
    P_o = torch.rand(h, d, v)  # W_o

    Q = torch.einsum("bnd,hdk->bhnk ", X, P_q)
    K = torch.einsum("bmd,hdk->bhmk", M, P_k)
    V = torch.einsum("bmd,hdv->bhmv", M, P_v)

    logits = torch.einsum("bhnk,bhmk->bhnm", Q, K)
    weights = torch.softmax(logits + mask, dim=-1)

    O = torch.einsum("bhnm,bhmv->bhnv ", weights, V)
    Y = torch.einsum("bhnv,hdv->bnd ", O, P_o)
    return Y
```

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 56

Time Range: 3476.12 s - 3540.0 s

access in this case is not the bottleneck so what I mean to say is that we are doing the number of the bottleneck of this algorithm is not the memory access it is actually the number of computations and as you saw before when we introduced the KV cache the problem we were trying to solve is the number of computations but by introducing the KV cache we created new problem I mean not a new problem but we we actually we have a new bottleneck and it's not the computation anymore so this algorithm here is the Multihead Self Attention but using the KV cache and this reduces the number of operations performed so if which you look at the number of arithmetic operations performed it's B and D squared the total memory involved in the operation is B and squared D plus N the this squared and the ratio between the two is this O of N divided by D plus 1 divided by B so the ratio between the total memory and the number of



Comparing different attention algorithms: **vanilla batched multi-head attention**

- Multihead Attention as presented in the original paper "Attention is all you need".
- By setting $m = n$ (sequence length of query = seq. length of keys and values)
- The number of arithmetic operations performed is $O(bnd^2)$
- The total memory involved in the operations, given by the sum of all the tensors involved in the calculations (including the derived ones!) is $O(bnd + bhn^2 + d^2)$
- The ratio between the total memory and the number of arithmetic operations is $O(\frac{1}{k} + \frac{1}{bn})$
- In this case, the ratio is much smaller than 1, which means that the number of memory access we are performing is much less than the number of arithmetic operations, so the memory access is **not** the bottleneck here.

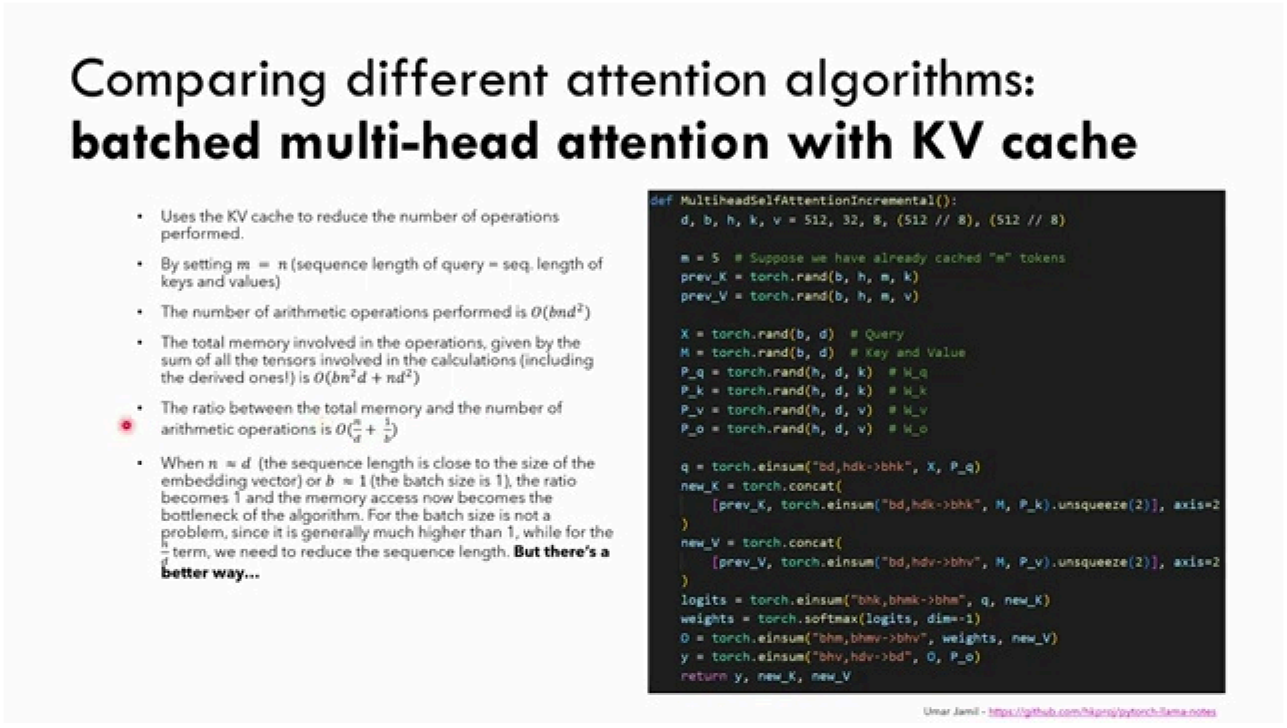
```python
def MultiheadAttentionBatched():
    d, m, n, b, h, k, v = 512, 10, 10, 32, 8, (512 // 8), (512 // 8)

    X = torch.rand(b, n, d)  # Query
    M = torch.rand(b, m, d)  # Key and Value
    mask = torch.rand(b, h, n, m)
    P_q = torch.rand(h, d, k)  # W_q
    P_k = torch.rand(h, d, k)  # W_k
    P_v = torch.rand(h, d, v)  # W_v
    P_o = torch.rand(h, d, v)  # W_o

    Q = torch.einsum("bnd,hdk->bhnk ", X, P_q)
    K = torch.einsum("bmd,hdk->bhmk", M, P_k)
    V = torch.einsum("bmd,hdv->bhmv", M, P_v)

    logits = torch.einsum("bhnk,bhmk->bhnm", Q, K)
    weights = torch.softmax(logits + mask, dim=-1)

    O = torch.einsum("bhnm,bhmv->bhnv ", weights, V)
    Y = torch.einsum("bhnv,hdv->bnd ", O, P_o)
    return Y
```

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 57

Time Range: 3540.0 s - 3602.7 s

arithmetic operations this means that when N is very similar to D this ratio will become 1 or when B is very similar to 1 or in the limit of 1 so the batch size is 1 this ratio will become 1 and this is a problem because now when this condition is verified is true then the memory access becomes the bottleneck of the algorithm and this also means that either we keep the dimension of the embedding vector much bigger than the sequence length but if we increase the sequence length without making the dimension of the embedding vector much bigger the memory access will become the bottleneck so what we can do is we need we need to find a better way to solve the problem of the previous algorithm in which the memory became the bottleneck we introduce the Multiquery Attention so what the author did was to remove the H dimension from



Comparing different attention algorithms: **batched multi-head attention with KV cache**

- Uses the KV cache to reduce the number of operations performed.
- By setting $m = n$ (sequence length of query = seq. length of keys and values)
- The number of arithmetic operations performed is $O(bnd^2)$
- The total memory involved in the operations, given by the sum of all the tensors involved in the calculations (including the derived ones!) is $O(bn^2d + nd^2)$
- The ratio between the total memory and the number of arithmetic operations is $O(\frac{n}{d} + \frac{1}{b})$
- When $n \approx d$ (the sequence length is close to the size of the embedding vector) or $b \approx 1$ (the batch size is 1), the ratio becomes 1 and the memory access now becomes the bottleneck of the algorithm. For the batch size is not a problem, since it is generally much higher than 1, while for the $\frac{n}{d}$ term, we need to reduce the sequence length. **But there's a better way…**

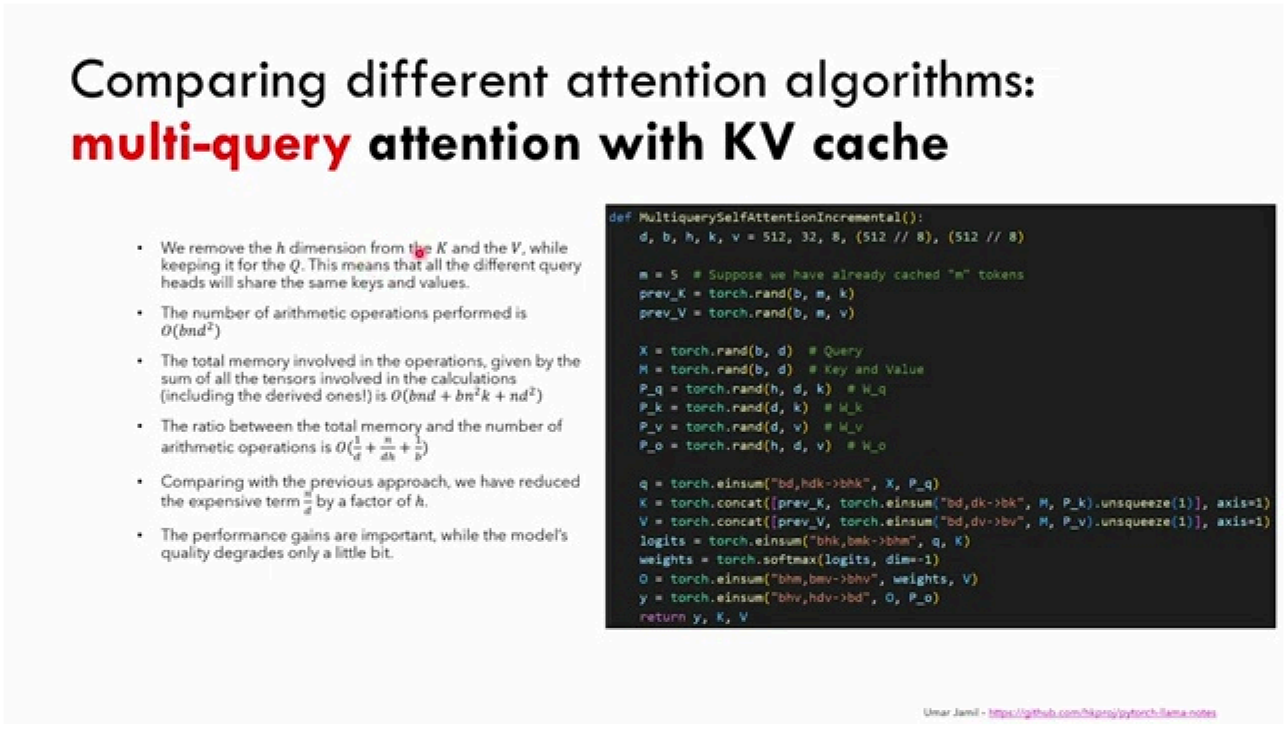
```python
def MultiheadSelfAttentionIncremental():
    d, b, h, k, v = 512, 32, 8, (512 // 8), (512 // 8)

    m = 5  # Suppose we have already cached "m" tokens
    prev_K = torch.rand(b, h, m, k)
    prev_V = torch.rand(b, h, m, v)

    X = torch.rand(b, d)  # Query
    M = torch.rand(b, d)  # Key and Value
    P_q = torch.rand(h, d, k)  # W_q
    P_k = torch.rand(h, d, k)  # W_k
    P_v = torch.rand(h, d, v)  # W_v
    P_o = torch.rand(h, d, v)  # W_o

    q = torch.einsum("bd,hdk->bhk", X, P_q)
    new_K = torch.concat(
        [prev_K, torch.einsum("bd,hdk->bhk", M, P_k).unsqueeze(2)], axis=2
    )
    new_V = torch.concat(
        [prev_V, torch.einsum("bd,hdv->bhv", M, P_v).unsqueeze(2)], axis=2
    )
    logits = torch.einsum("bhk,bhmk->bhm", q, new_K)
    weights = torch.softmax(logits, dim=-1)
    O = torch.einsum("bhm,bhmv->bhv", weights, new_V)
    y = torch.einsum("bhv,hdv->bd", O, P_o)
    return y, new_K, new_V
```

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 58

Time Range: 3602.7 s - 3666.86 s

the K and the V while keeping it for the Q so it's still a multi-head attention but only with respect to Q that's why it's called Multiquery Attention so we will have multiple heads only for the Q but the K and V will be shared by all the heads and if we use this algorithm the ratio becomes this 1 over D plus N divided by D H plus 1 over B so we compare it to the previous one in which was N divided by D now it's N divided by D H so we reduce the N divided by D factor the ratio N divided by D by a factor of H because we remove the H number of heads for the K and V so the gains the performance gains are important actually because now it happens less it is less likely that this ratio will become 1 but of course by removing the heads from the K and V



Comparing different attention algorithms: multi-query attention with KV cache

- We remove the h dimension from the K and the V, while keeping it for the Q. This means that all the different query heads will share the same keys and values.
- The number of arithmetic operations performed is $O(bnd^2)$
- The total memory involved in the operations, given by the sum of all the tensors involved in the calculations (including the derived ones!) is $O(bnd + bn^2k + nd^2)$
- The ratio between the total memory and the number of arithmetic operations is $O(\frac{1}{d} + \frac{n}{dh} + \frac{1}{b})$
- Comparing with the previous approach, we have reduced the expensive term $\frac{n}{d}$ by a factor of h.
- The performance gains are important, while the model's quality degrades only a little bit.

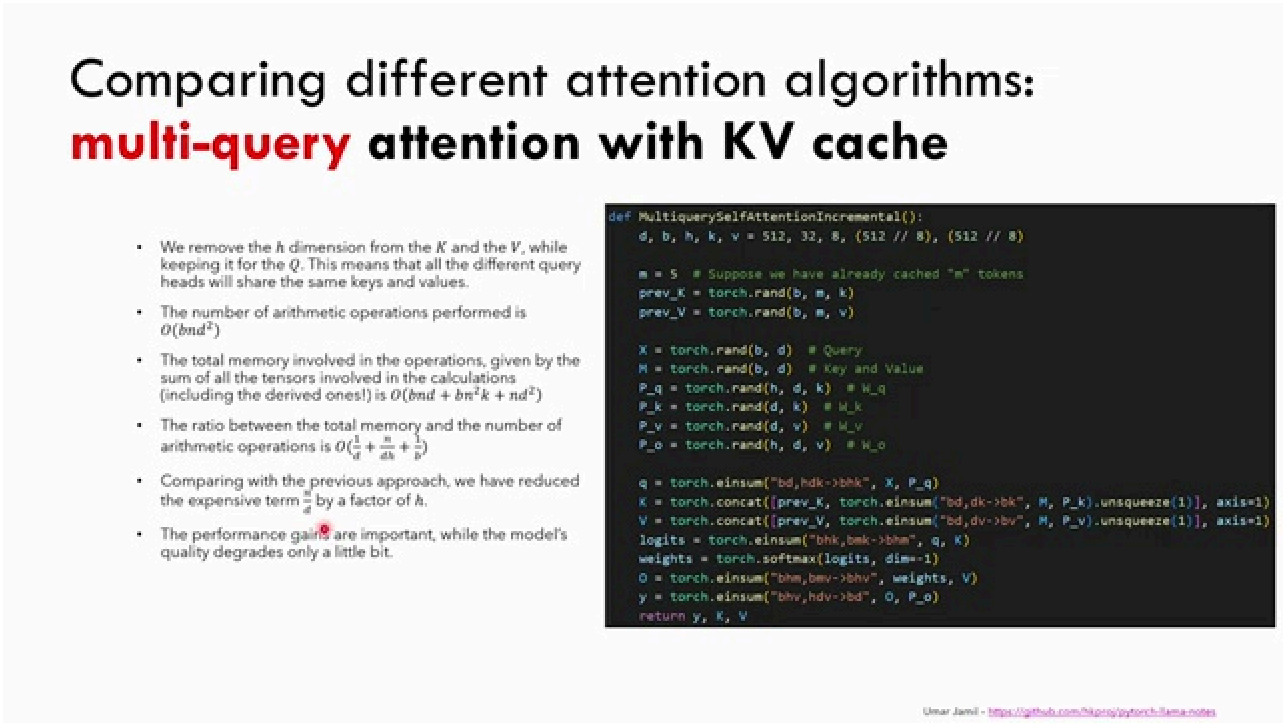
```python
def MultiquerySelfAttentionIncremental():
    d, b, h, k, v = 512, 32, 8, (512 // 8), (512 // 8)

    m = 5  # Suppose we have already cached "m" tokens
    prev_K = torch.rand(b, m, k)
    prev_V = torch.rand(b, m, v)

    X = torch.rand(b, d)  # Query
    M = torch.rand(b, d)  # Key and Value
    P_q = torch.rand(h, d, k)  # W_q
    P_k = torch.rand(d, k)  # W_k
    P_v = torch.rand(d, v)  # W_v
    P_o = torch.rand(h, d, v)  # W_o

    q = torch.einsum("bd,hdk->bhk", X, P_q)
    K = torch.concat([prev_K, torch.einsum("bd,dk->bk", M, P_k).unsqueeze(1)], axis=1)
    V = torch.concat([prev_V, torch.einsum("bd,dv->bv", M, P_v).unsqueeze(1)], axis=1)
    logits = torch.einsum("bhk,bmk->bhm", q, K)
    weights = torch.softmax(logits, dim=-1)
    O = torch.einsum("bhm,bmv->bhv", weights, V)
    y = torch.einsum("bhv,hdv->bd", O, P_o)
    return y, K, V
```

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 59

Time Range: 3666.86 s - 3730.3 s

our model will also have less parameters it will also have less degrees of freedom and complexity which may degrade the quality of the model and it actually does degrade the quality of the model but only slightly and we will see so if we compare for example the blue score or a translation task from English to German we can see that the multi-head attention so the attention that was in the original attention paper has a blue score of 26.7 while the multi-query has a blue score of 26.5 the outer also compared with the multi-head local and multi-query local where local means that they restrict the attention calculation only to the previous 31 positions of each token and we can see it here but the performance gains by reducing the heads of the K and the V is great because you can see the inference time for example on the



Comparing different attention algorithms: multi-query attention with KV cache

- We remove the h dimension from the K and the V, while keeping it for the Q. This means that all the different query heads will share the same keys and values.
- The number of arithmetic operations performed is $O(bnd^2)$
- The total memory involved in the operations, given by the sum of all the tensors involved in the calculations (including the derived ones!) is $O(bnd + bn^2k + nd^2)$
- The ratio between the total memory and the number of arithmetic operations is $O(\frac{1}{d} + \frac{n}{dh} + \frac{1}{b})$
- Comparing with the previous approach, we have reduced the expensive term $\frac{n}{d}$ by a factor of h.
- The performance gains are important, while the model's quality degrades only a little bit.

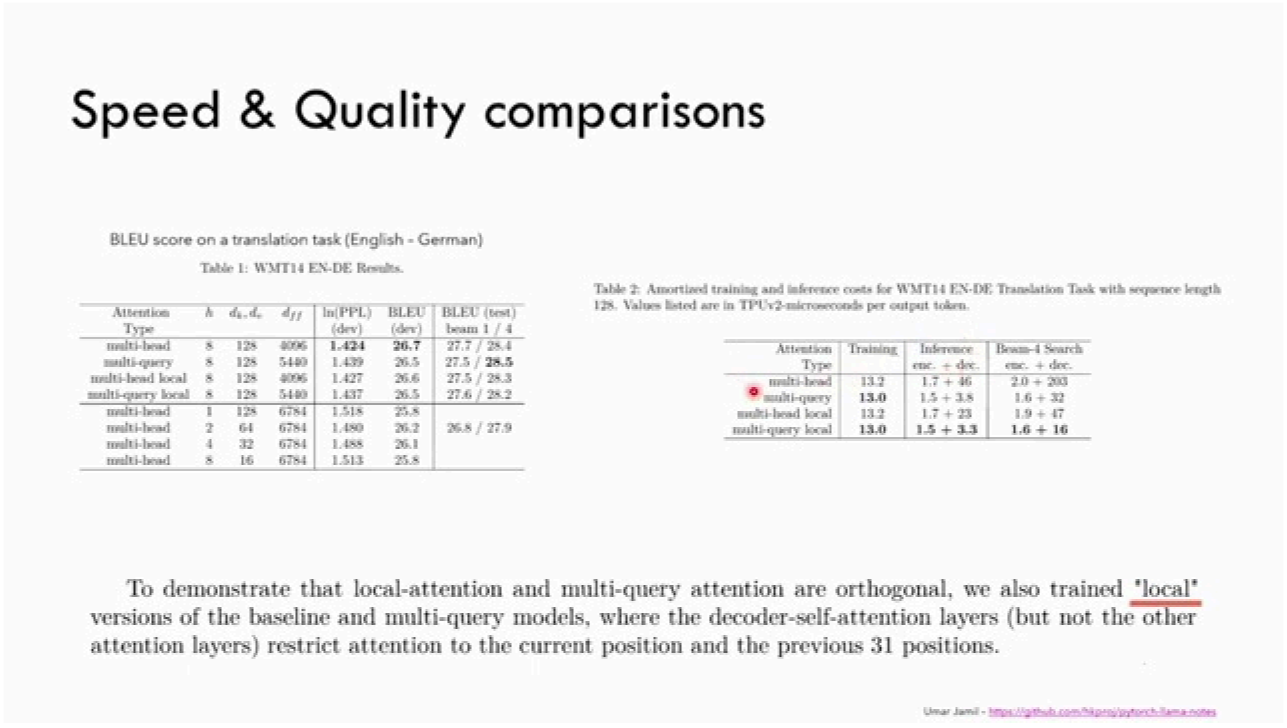
```
def MultiquerySelfAttentionIncremental():
    d, b, h, k, v = 512, 32, 8, (512 // 8), (512 // 8)

    m = 5  # Suppose we have already cached "m" tokens
    prev_K = torch.rand(b, m, k)
    prev_V = torch.rand(b, m, v)

    X = torch.rand(b, d)  # Query
    M = torch.rand(b, d)  # Key and Value
    P_q = torch.rand(h, d, k)  # W_q
    P_k = torch.rand(d, k)  # W_k
    P_v = torch.rand(d, v)  # W_v
    P_o = torch.rand(h, d, v)  # W_o

    q = torch.einsum("bd,hdk->bhk", X, P_q)
    K = torch.concat([prev_K, torch.einsum("bd,dk->bk", M, P_k).unsqueeze(1)], axis=1)
    V = torch.concat([prev_V, torch.einsum("bd,dv->bv", M, P_v).unsqueeze(1)], axis=1)
    logits = torch.einsum("bhk,bmk->bhm", q, K)
    weights = torch.softmax(logits, dim=-1)
    O = torch.einsum("bhm,bmv->bhv", weights, V)
    y = torch.einsum("bhv,hdv->bd", O, P_o)
    return y, K, V
```

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 60

Time Range: 3730.3 s - 3791.22 s

original multi-head attention and the multi-query attention the inference time went from 1.7 microseconds plus 46 microseconds for the decoder to 1.5 microseconds plus 3.8 microseconds for the decoder so in total here more or less we took 48 seconds 48 microseconds while here we more or less take 6 microseconds for the multi-query so it's a great benefit from from inferencing from a performance point of view during the inferencing let's talk about grouped multi-query attention because now we just introduced the KV cache and the multi-query attention but the next step of the multi-query attention is the grouped multi-query attention which is the one that is used in Lama so let's have a look at it with multi-query we only have multiple heads for the queries but only one head for the key and the values with grouped multi-query



Speed & Quality comparisons

BLEU score on a translation task (English - German)

Table 1: WMT14 EN-DE Results.

Attention Type	h	d_k, d_v	d_{ff}	ln(PPL) (dev)	BLEU (dev)	BLEU (test) beam 1 / 4
multi-head	8	128	4096	**1.424**	**26.7**	27.7 / 28.4
multi-query	8	128	5440	1.439	26.5	27.5 / **28.5**
multi-head local	8	128	4096	1.427	26.6	27.5 / 28.3
multi-query local	8	128	5440	1.437	26.5	27.6 / 28.2
multi-head	1	128	6784	1.518	25.8	
multi-head	2	64	6784	1.480	26.2	26.8 / 27.9
multi-head	4	32	6784	1.488	26.1	
multi-head	8	16	6784	1.513	25.8	

Table 2: Amortized training and inference costs for WMT14 EN-DE Translation Task with sequence length 128. Values listed are in TPUv2-microseconds per output token.

Attention Type	Training	Inference enc. + dec.	Beam-4 Search enc. + dec.
multi-head	13.2	1.7 + 46	2.0 + 203
multi-query	**13.0**	1.5 + 3.8	1.6 + 32
multi-head local	13.2	1.7 + 23	1.9 + 47
multi-query local	**13.0**	**1.5 + 3.3**	**1.6 + 16**

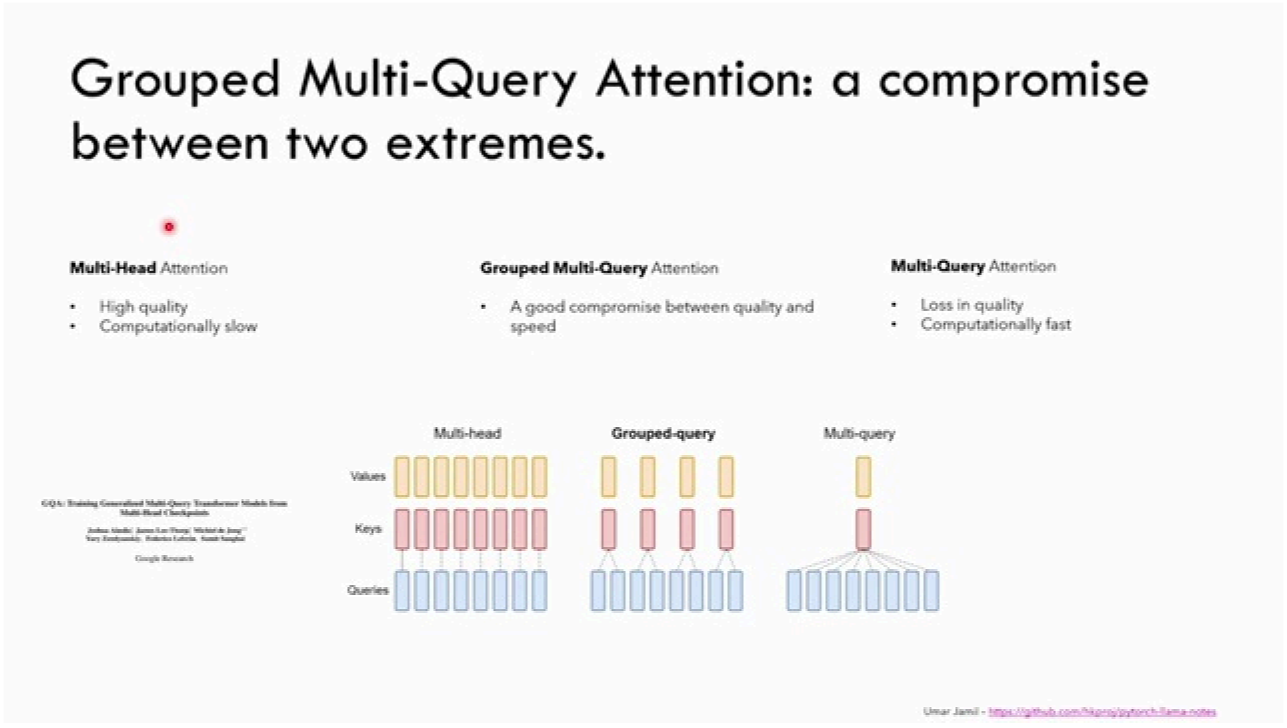
To demonstrate that local-attention and multi-query attention are orthogonal, we also trained "local" versions of the baseline and multi-query models, where the decoder-self-attention layers (but not the other attention layers) restrict attention to the current position and the previous 31 positions.

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 61

Time Range: 3791.22 s - 3853.62 s

attention basically we divide the queries into groups so for example this is the group one this is the group two group three and group four and for each group we have one different head of K and V this is a good compromise between the multi-head in which there is one to one correspondence and the multi-query where there is an end to one correspondence so in this case we have still a multiple heads for the keys and values but they are less numerically compared to the number of heads of the queries and this is a good compromise between the quality of the model and the speed of the model because anyway here we benefit from the computational benefit of the reduction in the number of heads of key and values but we don't sacrifice too much on the quality side and now the last part of the model as you can see here the fit for one in the Lama model



Grouped Multi-Query Attention: a compromise between two extremes.

Multi-Head Attention

- High quality
- Computationally slow

Grouped Multi-Query Attention

- A good compromise between quality and speed

Multi-Query Attention

- Loss in quality
- Computationally fast

GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, Sumit Sanghai

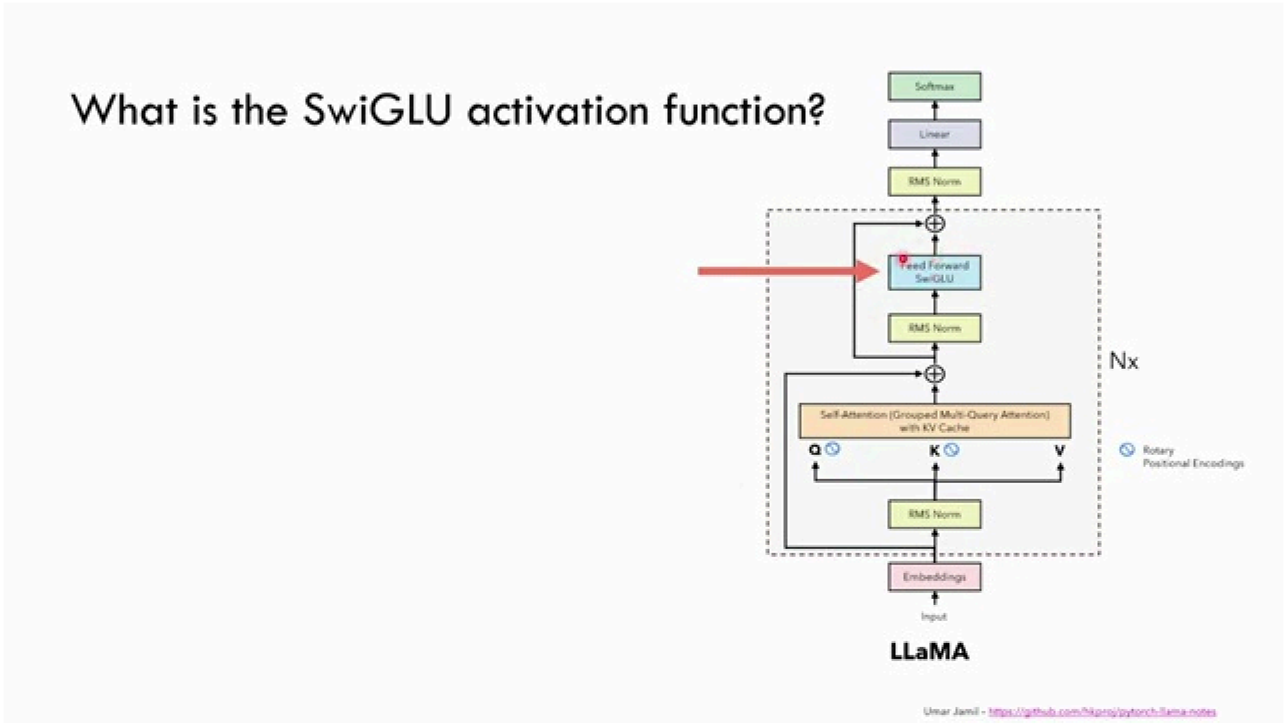
Google Research

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 62

Time Range: 3853.62 s - 3916.42 s

has been converted into has its activation function changed with the Zviglou function let's have a look at how it works so the Zviglou function was analyzed in this famous paper from Noam Shahzir who is also one of the outer of the attention model who is also one of the the outer of the multi-query attention that we saw before so let's have a look at this paper so the outer compared the performance of the transformer model by using different activation functions in the feed for one layer of the transformer architecture and the one we are interested in is this Zviglou here which is basically the Zviglou function with beta equal to one calculated in the X multiply by a W matrix which is a parametri matrix which is then multiplied with X multiplied by V V is also another parametri matrix and W2 which is another matter parametri matrix so compare this with the original feed for one network and



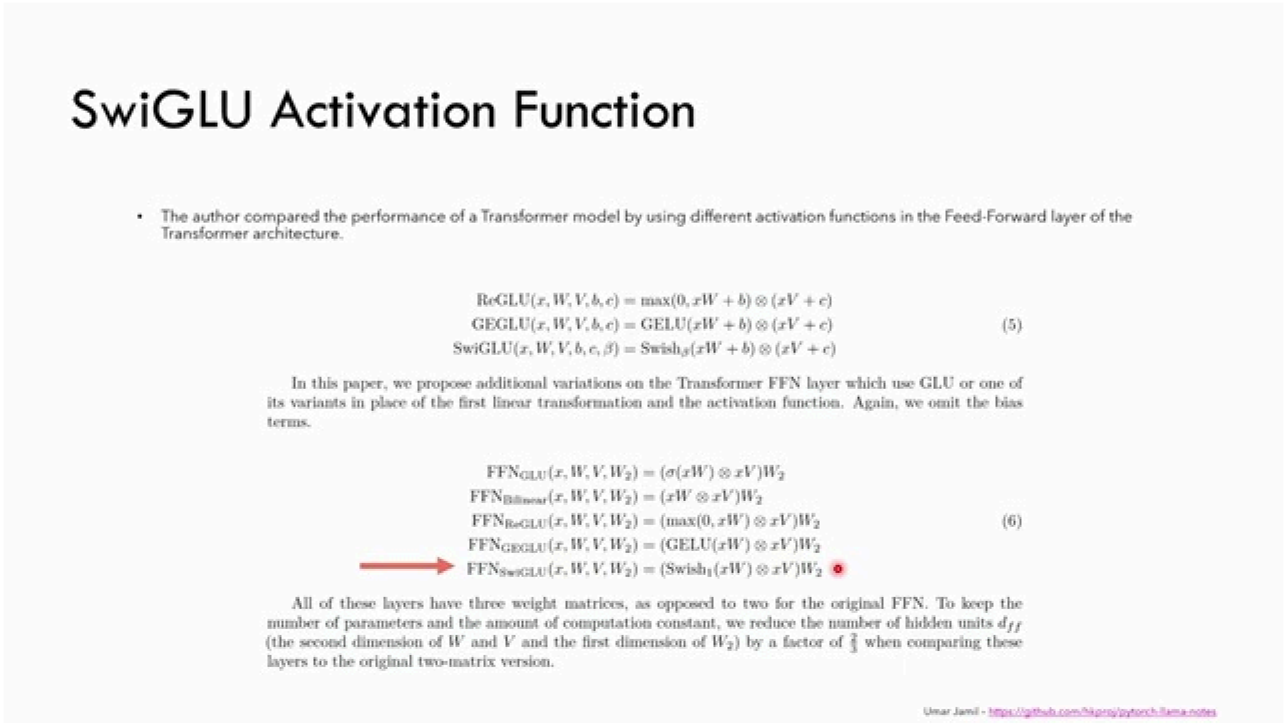
What is the SwiGLU activation function?

Softmax
Linear
RMS Norm
Feed Forward SwiGLU
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input
LLaMA
Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 63

Time Range: 3916.42 s - 3978.06 s

here we have three parametri matrices while in the original feed for one network we only had two so to make the comparison fair the outer reduced the number of the size of this matrices to have to such that the model models total number of parameters remains the same with the Vanilla transformer in the Vanilla transformer we had this feed for one network which was the reloo function so this max zero etc is the reloo function and we only had the two parametri matrices actually some success or version of the transformer didn't have the bias so this is I took this formula from the paper but there are many implementations without the bias actually and while in Lama we use this computation for the feed for one network and this is the code I took from the repository from Lama and as you can see it's just what the model says it's the Zviglou function why the Zviglou function because it's the Zvig function with beta



SwiGLU Activation Function

- The author compared the performance of a Transformer model by using different activation functions in the Feed-Forward layer of the Transformer architecture.

$$\begin{aligned} \mathrm{ReGLU}(x, W, V, b, c) &= \max(0, xW + b) \otimes (xV + c) \\ \mathrm{GEGLU}(x, W, V, b, c) &= \mathrm{GELU}(xW + b) \otimes (xV + c) \\ \mathrm{SwiGLU}(x, W, V, b, c, \beta) &= \mathrm{Swish}_\beta(xW + b) \otimes (xV + c) \end{aligned} \tag{5}$$

In this paper, we propose additional variations on the Transformer FFN layer which use GLU or one of its variants in place of the first linear transformation and the activation function. Again, we omit the bias terms.

$$\begin{aligned} \mathrm{FFN}_{\mathrm{GLU}}(x, W, V, W_2) &= (\sigma(xW) \otimes xV)W_2 \\ \mathrm{FFN}_{\mathrm{Bilinear}}(x, W, V, W_2) &= (xW \otimes xV)W_2 \\ \mathrm{FFN}_{\mathrm{ReGLU}}(x, W, V, W_2) &= (\max(0, xW) \otimes xV)W_2 \\ \mathrm{FFN}_{\mathrm{GEGLU}}(x, W, V, W_2) &= (\mathrm{GELU}(xW) \otimes xV)W_2 \\ \mathrm{FFN}_{\mathrm{SwiGLU}}(x, W, V, W_2) &= (\mathrm{Swish}_1(xW) \otimes xV)W_2 \end{aligned} \tag{6}$$

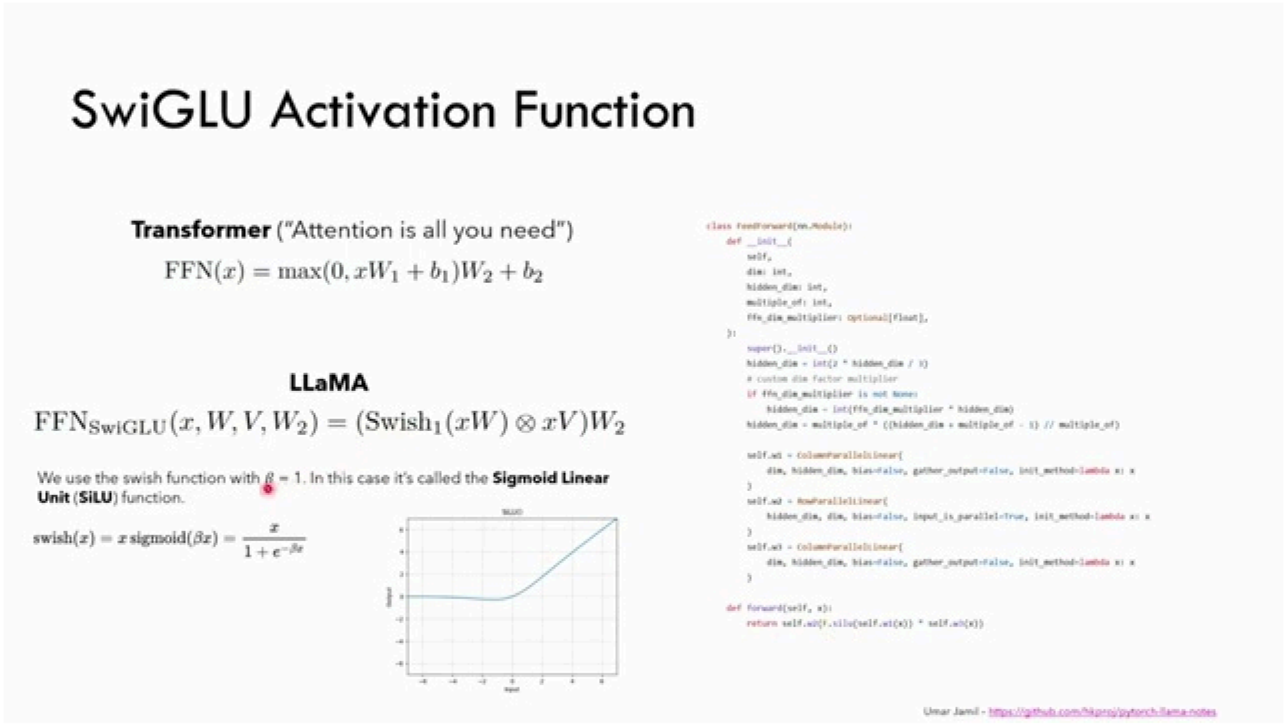
All of these layers have three weight matrices, as opposed to two for the original FFN. To keep the number of parameters and the amount of computation constant, we reduce the number of hidden units d_{ff} (the second dimension of W and V and the first dimension of W_2) by a factor of $\frac{2}{3}$ when comparing these layers to the original two-matrix version.

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 64

Time Range: 3978.06 s - 4040.7 s

equal to one and when the Zviglou function that has this expression we give beta equal to one it's called the sigmoid linear unit that has this graph and it's called silo so it's a silo function evaluated in the w1 of x then multiplied by w3 which is then we then we applied to w2 so we have three matrices and these three matrices are basically linear layers now they use the parallelized version of this linear layer but it's a linear layer and if we look at the graph of this silo function we can see that it's kind of like a relu but in this here before the zero we don't cancel out immediately the activation we keep a little tail here so that even values that are very close to zero from the negative side are not automatically canceled out by the function so let's see how does it perform so this is wiglou function actually



SwiGLU Activation Function

Transformer ("Attention is all you need")

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

LLaMA

$$\mathrm{FFN}_{\mathrm{SwiGLU}}(x, W, V, W_2) = (\mathrm{Swish}_1(xW) \otimes xV)W_2$$

We use the swish function with $\beta = 1$. In this case it's called the **Sigmoid Linear Unit** (**SiLU**) function.

$$\mathrm{swish}(x) = x\,\mathrm{sigmoid}(\beta x) = \frac{x}{1 + e^{-\beta x}}$$

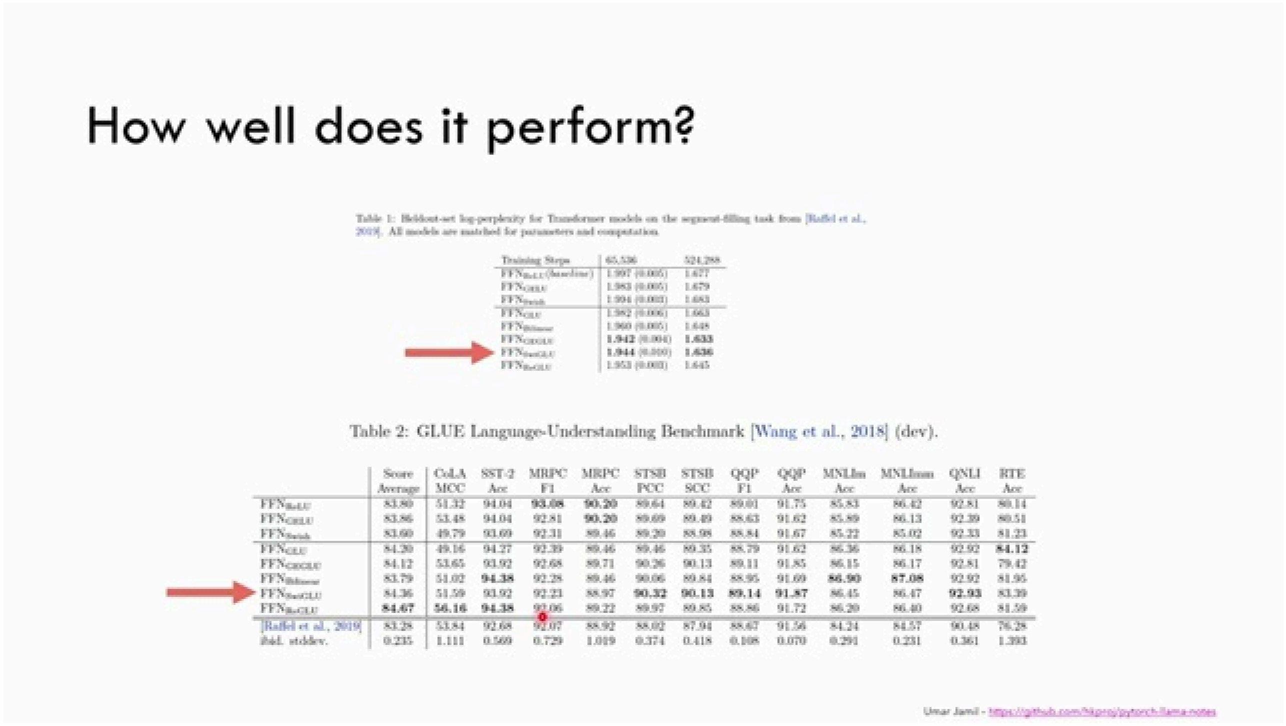
```
class FeedForward(nn.Module):
    def __init__(
        self,
        dim: int,
        hidden_dim: int,
        multiple_of: int,
        ffn_dim_multiplier: Optional[float],
    ):
        super().__init__()
        hidden_dim = int(2 * hidden_dim / 3)
        # custom dim factor multiplier
        if ffn_dim_multiplier is not None:
            hidden_dim = int(ffn_dim_multiplier * hidden_dim)
        hidden_dim = multiple_of * ((hidden_dim + multiple_of - 1) // multiple_of)

        self.w1 = ColumnParallelLinear(
            dim, hidden_dim, bias=False, gather_output=False, init_method=lambda x: x
        )
        self.w2 = RowParallelLinear(
            hidden_dim, dim, bias=False, input_is_parallel=True, init_method=lambda x: x
        )
        self.w3 = ColumnParallelLinear(
            dim, hidden_dim, bias=False, gather_output=False, init_method=lambda x: x
        )

    def forward(self, x):
        return self.w2(F.silu(self.w1(x)) * self.w3(x))
```

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 65

Time Range: 4040.7 s - 4102.66 s

performs very well here they evaluate the complete log complexity per complexity of the the model when we use this particular function and we can see that the per complexity here is the lowest the perplexity basically means how unsure is the model about its choices and the Zviglou function is performing well then they also run the same the comparison on many benchmarks and we see that the Zviglou function is performing quite well on a lot of them so why is the Zviglou activation function working so well if we look at the conclusion of this paper we see that we offer no explanation as to why this architecture seems to work we attribute their success as all else to divine benevolence actually this is okay kind of funny but it's also kind of true because in most of the deep learning research we do not know why things



How well does it perform?

Table 1: Heldout-set log-perplexity for Transformer models on the segment-filling task from [Raffel et al., 2019]. All models are matched for parameters and computation.

Training Steps	65,536	524,288
FFN_{ReLU} (baseline)	1.997 (0.005)	1.677
FFN_{GELU}	1.983 (0.005)	1.679
FFN_{Swish}	1.994 (0.003)	1.683
FFN_{GLU}	1.982 (0.006)	1.663
$FFN_{Bilinear}$	1.960 (0.005)	1.648
FFN_{GEGLU}	**1.942** (0.004)	**1.633**
FFN_{SwiGLU}	**1.944** (0.010)	**1.636**
FFN_{ReGLU}	1.953 (0.003)	1.645

Table 2: GLUE Language-Understanding Benchmark [Wang et al., 2018] (dev).

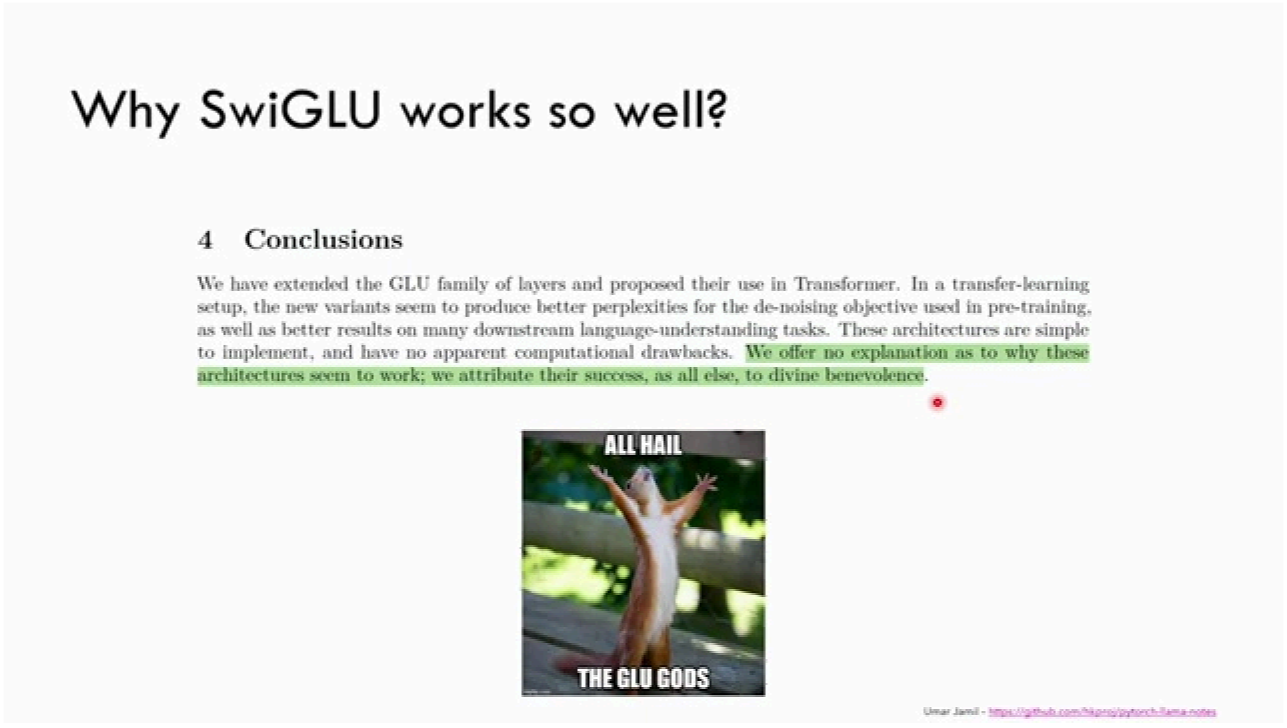
	Score	CoLA	SST-2	MRPC	MRPC	STSB	STSB	QQP	QQP	MNLIm	MNLImm	QNLI	RTE
	Average	MCC	Acc	F1	Acc	PCC	SCC	F1	Acc	Acc	Acc	Acc	Acc
FFN_{ReLU}	83.80	51.32	94.04	**93.08**	**90.20**	89.64	89.42	89.01	91.75	85.83	86.42	92.81	80.14
FFN_{GELU}	83.86	53.48	94.04	92.81	**90.20**	89.69	89.49	88.63	91.62	85.89	86.13	92.39	80.51
FFN_{Swish}	83.60	49.79	93.69	92.31	89.46	89.20	88.98	88.84	91.67	85.22	85.02	92.33	81.23
FFN_{GLU}	84.20	49.16	94.27	92.39	89.46	89.46	89.35	88.79	91.62	86.36	86.18	92.92	**84.12**
FFN_{GEGLU}	84.12	53.65	93.92	92.68	89.71	90.26	90.13	89.11	91.85	86.15	86.17	92.81	79.42
$FFN_{Bilinear}$	83.79	51.02	**94.38**	92.28	89.46	90.06	89.84	88.95	91.69	**86.90**	**87.08**	92.92	81.95
FFN_{SwiGLU}	84.36	51.59	93.92	92.23	88.97	**90.32**	**90.13**	**89.14**	**91.87**	86.45	86.47	**92.93**	83.39
FFN_{ReGLU}	**84.67**	**56.16**	**94.38**	92.06	89.22	89.97	89.85	88.86	91.72	86.20	86.40	92.68	81.59
[Raffel et al., 2019]	83.28	53.84	92.68	92.07	88.92	88.02	87.94	88.67	91.56	84.24	84.57	90.48	76.28
ibid. stddev.	0.235	1.111	0.569	0.729	1.019	0.374	0.418	0.108	0.070	0.291	0.231	0.361	1.393

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 66

Time Range: 4102.66 s - 4165.78 s

work in the way they do because imagine you have a model of 70 billion parameters how can you prove what is happening to each one of them after you modify one activation function it's not easy to come up with a model that can explain why the model is reacting in particular way what usually we do we have some we can either simplify the model so we can work with as many small model and then make some assumptions on why things work the way they do or we can just do it on a practical level so we take a model we modify it a little bit we do some oblation study and we check which one is performing better and this is also happens in a lot of areas of machine learning for example we do a lot of grid search to find the right parameters for a model because we cannot know beforehand which one will work well or which one to increase or which one to decrease because it depends on a lot of factors not only on the algorithm use but also on the data also on the particular competitions use also on the



Why SwiGLU works so well?

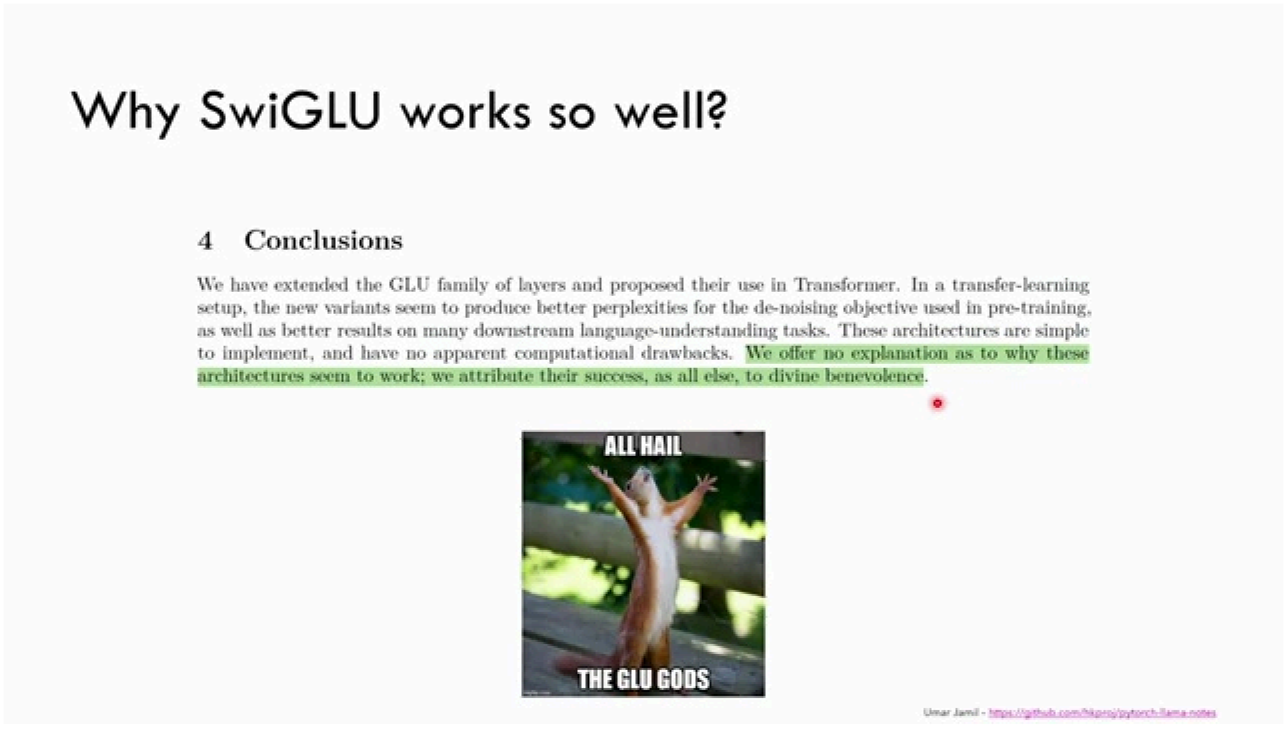
4 Conclusions

We have extended the GLU family of layers and proposed their use in Transformer. In a transfer-learning setup, the new variants seem to produce better perplexities for the de-noising objective used in pre-training, as well as better results on many downstream language-understanding tasks. These architectures are simple to implement, and have no apparent computational drawbacks. We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence.

Section 67

Time Range: 4165.78 s - 4226.58 s

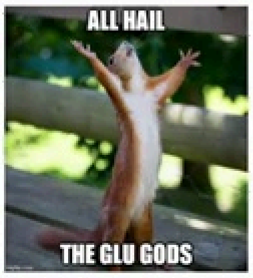
normalization use so there is a lot of factors there is no formula for everything to explain everything so this is why the research needs to do a lot of study on the lot of on the variance of models to come up with something that works maybe in one domain and doesn't work well in other domains so in this case we use this wigloo mostly because in practice it works well with this kind of models thank you guys for watching this long video I hope that you learned in the deeper level what happens in Lama and why it is different from a standard transformer model I know that the video has been quite long and I know that it has been hard on some parts to follow so I actually kind of suggest to rewatch it multiple times especially the parts that you are less familiar with and to integrate this video with my previous video about the transformer so you can I will put the chapters so you can easily find the part that you want but this is what you need to do you need to watch multiple times the same concept to



Why SwiGLU works so well?

4 Conclusions

We have extended the GLU family of layers and proposed their use in Transformer. In a transfer-learning setup, the new variants seem to produce better perplexities for the de-noising objective used in pre-training, as well as better results on many downstream language-understanding tasks. These architectures are simple to implement, and have no apparent computational drawbacks. We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence.



Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

Section 68

Time Range: 4226.58 s - 4255.9 s

actually master it and I hope to make another video in which we code the Lama model from zero so we can put all this theory into practice but as you know I am doing this as on my free time and my free time is not so much so thank you guys for watching my video and please subscribe to my channel because this is the best motivation for me to keep posting amazing content on AI and machine learning thank you for watching and have an amazing rest of the day

Thanks for watching!
Don't forget to subscribe for more amazing content on AI and Machine Learning!

Umar Jamil - https://github.com/hkproj/pytorch-llama-notes