self-attention trick

8 tokens in a batch

B, T, C

Batch, Time, Channel

some info at each pt of seq C

[4, 8, 2]

We would like to take these 8 tokens in a batch & these tokens are curr not talking to each other & we would like them to talk to each other we would like to couple them.

We want to couple them in a specific way. for ex: the token at 5th locn should not communicate with 6th, 7th, & 8th locn because those are future tokens. The token on the 5th should only be able to communicate with 4th, 3rd, 2nd & 1st location. Hence, info only flows from prev context to the curr time step.

So, what's the easiest way for tokens to communicate?

One way could be to take the avg of previous cols for ex if I am at 5th token, I would like to take channels that make up info at curr step & also info from 4th, 3rd, 2nd & 1st time step. I would like to avg those up & then that will become the feature vector that summarizes me in the context of history

Now, clearly just doing a sum/avg is extremely a weak

form of interaction. This communication is extremely lossy. We have lost a lot of info like their spatial arr but that is okay for now. WE CAN HAVE A LOOK AT THAT LBR!!

xbow = $\begin{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \\ \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \end{bmatrix}$ B = 2, T = 3, C = 2

for b in range(B):
  for t in range(T):
    xprev = x[b:t]

Coming back to the trick

we were using for loop not v. effi to find avg

instead let's use matrix mult to find mean

$a = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$, $b = \begin{bmatrix} 2 & 7 \\ 6 & 4 \\ 6 & 5 \end{bmatrix}$, $c = \begin{bmatrix} 14 & 16 \\ 14 & 16 \\ 14 & 16 \end{bmatrix}$

a @ b = c

Trick

use torch.tril(torch.ones(3,3))

$a = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$ $b = \begin{bmatrix} 2 & 7 \\ 6 & 4 \\ 6 & 5 \end{bmatrix}$ $c = \begin{bmatrix} 2 & 7 \\ 8 & 11 \\ 14 & 16 \end{bmatrix}$

```
In [87]: torch.manual_seed(42)
         a = torch.tril(torch.ones(3,3))
         a = a / torch.sum(a,1, keepdim = True)# this gives us the average
         b = torch.randint(0,10,(3,2)).float()
         c = a @ b
         print('a=')
         print(a)
         print('----')
         print('b=')
         print(b)
         print('----')
         print('c=')
         print(c)
```

```
a=
tensor([[1.0000, 0.0000, 0.0000],
        [0.5000, 0.5000, 0.0000],
        [0.3333, 0.3333, 0.3333]])
----
b=
tensor([[2., 7.],
        [6., 4.],
        [6., 5.]])
----
c=
tensor([[2.0000, 7.0000],
        [4.0000, 5.5000],
        [4.6667, 5.3333]])
```

```
In [110]: tril
Out[110]: tensor([[1., 0., 0., 0., 0., 0., 0., 0.],
                  [1., 1., 0., 0., 0., 0., 0., 0.],
                  [1., 1., 1., 0., 0., 0., 0., 0.],
                  [1., 1., 1., 1., 0., 0., 0., 0.],
                  [1., 1., 1., 1., 1., 0., 0., 0.],
                  [1., 1., 1., 1., 1., 1., 0., 0.],
                  [1., 1., 1., 1., 1., 1., 1., 0.],
                  [1., 1., 1., 1., 1., 1., 1., 1.]])

In [111]: wei
Out[111]: tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
                  [0.5000, 0.5000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
                  [0.3333, 0.3333, 0.3333, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
                  [0.2500, 0.2500, 0.2500, 0.2500, 0.0000, 0.0000, 0.0000, 0.0000],
                  [0.2000, 0.2000, 0.2000, 0.2000, 0.2000, 0.0000, 0.0000, 0.0000],
                  [0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.0000, 0.0000],
                  [0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.0000],
                  [0.1250, 0.1250, 0.1250, 0.1250, 0.1250, 0.1250, 0.1250, 0.1250]])
```

[In Jup notebook go to V3]:
note how wei begin at 0 & you can think of it as interaction strength or like an affinity. It's telling us how much of each token from the past do we want to aggregate or average of.
And then this line: wei = torch.mask_fill(tril==0, float('-inf')) says that tokens from the past cannot communicate by setting them to -ve (inf). And the line xbow3 = wei @ x is the aggregation. Quick preview: these affinities (wei) are not going to be zero, they are going to be data dependent. These tokens start looking at each other & some token may find other tokens more interesting.

self-attention tril [[1 0] [1 1]]

so, we don't want "wei" to be all uniform because diff tokens will find diff other tokens more or less interesting & we want that to be data dependent. so, for ex: if I am at a vowel then may be I'm looking for consonants in my past and may be I want to know what those consonants are & I want that info to "flow" to me
so, now I want to gather info from the past but I want to do it in a data dependent way AND this is the problem that self-attention solves
Now, the way self-attn' solves it:
every single node / token at each posn' will emit two vectors: Q: query and K: key.
Q: Query - what am I looking for K: what do I contain & the way we get affinities b/w these token in a sequence is basically we just do a dot product b/w Query & key and that dot product now becomes wei.
so, if Key & Query are aligned they interact with high amount and then I get to learn more about that specific token as opposed to any other token in the sequence.
(Back to jupyter-notebook!)

Query and Key Implementation within our simple concept

```
In [70]: # single head of self-attention
         #v4: self-attention
         torch.manual_seed(1337)
         B,T,C = 4,8,32 # batch, time, channels
         x = torch.randn(B,T,C)

         #let's see a single Head perform self-attention
         head_size = 16 # hyperparameter
         key = nn.Linear(C, head_size, bias = False) # just simple mat mul
         query = nn.Linear(C, head_size, bias = False)
         k = key(x) # (B,T,16)
         q = query(x) # (B,T,16)

         wei = q @ k.transpose(-2,-1) # (B,T,16) * (B,16,T) --> (B,T,T)

         tril = torch.tril(torch.ones(T,T))
         # wei = torch.zeros((T,T))
         wei = wei.masked_fill(tril == 0, float('-inf'))
         wei = F.softmax(wei,dim=1)
         out = wei @ x

         out.shape

Out[70]: torch.Size([4, 8, 32])
```

Key = (C, h_size)
query = (C, h_size)
k = (B,T,C) * (C, h_size)
=> (B,T,h_size)

```
>>> m = nn.Linear(20, 30)
>>> input = torch.randn(128, 20)
>>> output = m(input)
>>> print(output.size())
torch.Size([128, 30])
```

(20,30)
(128,20) * (20,30)

After imp of Q,K

un-linear

You see when I forward this linear on top of x, all the tokens in all the posn' in the (B,T) arrangement. all of them in parallel & independently produce a key & a query. So, no communication has happened yet. But the communication comes now. All the Qs will dot product with all the keys q ⊙ k. Hence, wei/ affinities b/w there to be q ⊙ k # take care of dims.

so, q (B, T, C) $(T, C_2) * (C, T)$

k (B, T, C) k.transpose(-2, -1) = (B, C, T)
-2 -1

q @ k.transpose(-2, -1) (B, T, T)

for every row of B we get T2 matrix

giving us affinities

wei.shape = [(4, 8, 8)] # note before dup! 4 or V4
V4 wei (new) wei was constant/uniform
v3 wei (old)

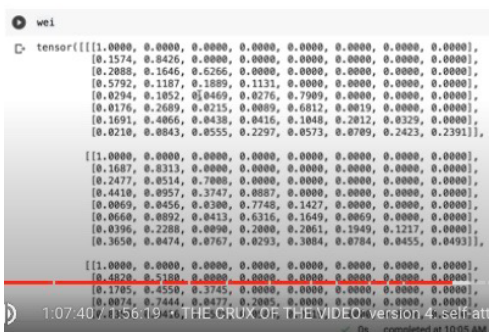
```
wei
```

```
tensor([[[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.1574, 0.8426, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.2088, 0.1646, 0.6266, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.5792, 0.1187, 0.1889, 0.1131, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.0294, 0.1052, 0.0469, 0.0276, 0.7909, 0.0000, 0.0000, 0.0000],
         [0.0176, 0.2689, 0.0215, 0.0089, 0.6812, 0.0019, 0.0000, 0.0000],
         [0.1691, 0.4066, 0.0438, 0.0416, 0.1048, 0.2012, 0.0329, 0.0000],
         [0.0210, 0.0843, 0.0555, 0.2297, 0.0573, 0.0709, 0.2423, 0.2391]],

        [[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.1687, 0.8313, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.2477, 0.0514, 0.7008, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.4410, 0.0957, 0.3747, 0.0887, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.0069, 0.0456, 0.0300, 0.7748, 0.1427, 0.0000, 0.0000, 0.0000],
         [0.0660, 0.0892, 0.0413, 0.6316, 0.1649, 0.0069, 0.0000, 0.0000],
         [0.0396, 0.2288, 0.0090, 0.2000, 0.2061, 0.1949, 0.1217, 0.0000],
         [0.3650, 0.0474, 0.0767, 0.0293, 0.3084, 0.0784, 0.0455, 0.0493]],

        [[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.4820, 0.5180, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.1705, 0.4550, 0.3745, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.0074, 0.7444, 0.0477, 0.2005, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.8359, 0.0416, 0.0525, 0.0580, 0.0119, 0.0000, 0.0000, 0.0000],
         [0.1195, 0.2061, 0.1019, 0.1153, 0.1814, 0.2758, 0.0000, 0.0000],
         [0.0065, 0.0589, 0.0372, 0.3063, 0.1325, 0.3209, 0.1378, 0.0000],
         [0.1416, 0.1519, 0.0384, 0.1643, 0.1207, 0.1254, 0.0169, 0.2408]],

        [[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.6369, 0.3631, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.2586, 0.7376, 0.0038, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.4692, 0.3440, 0.1237, 0.0631, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.1865, 0.4680, 0.0353, 0.1854, 0.1248, 0.0000, 0.0000, 0.0000],
         [0.0828, 0.7479, 0.0017, 0.0735, 0.0712, 0.0228, 0.0000, 0.0000],
         [0.0522, 0.0517, 0.0961, 0.0375, 0.1024, 0.5730, 0.0872, 0.0000],
         [0.0306, 0.2728, 0.0333, 0.1409, 0.1414, 0.0582, 0.0825, 0.2402]]],
       grad_fn=<SoftmaxBackward0>)
```



wei V4 not uniform because every single batch ele contains diff tokens & diff posn' & hence, now DATA DEPENDENT.

```
]: wei[0]
```

```
]: tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
           [0.1574, 0.8426, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
           [0.2088, 0.1646, 0.6266, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
           [0.5792, 0.1187, 0.1889, 0.1131, 0.0000, 0.0000, 0.0000, 0.0000],
           [0.0294, 0.1052, 0.0469, 0.0276, 0.7909, 0.0000, 0.0000, 0.0000],
           [0.0176, 0.2689, 0.0215, 0.0089, 0.6812, 0.0019, 0.0000, 0.0000],
           [0.1691, 0.4066, 0.0438, 0.0416, 0.1048, 0.2012, 0.0329, 0.0000],
           [0.0210, 0.0843, 0.0555, 0.2297, 0.0573, 0.0709, 0.2423, 0.2391]],
          grad_fn=<SelectBackward0>)
```

8th token knows what posn' it is at & what content it has and now 8th token based on that creates a query (hey, I am looking for this kinda stuff, I am a vowel, I am at 8th posn & I am looking for any consonants at positions up to 4. And then all the nodes get to emit keys. And maybe one of the channels could be "I am a consonant, and I am in posn' up to 4 & that key would have a high no. in that specific channel. And that's how when Q and K dot product they can find each other and create high affinity. And when they have high affinity (say 0.2294), then through softmax I will end up aggregating a lot of its info into my posn' & so I get to learn a lot about it.

Quick Under-the-Hood view

```
#v4: self-attention
torch.manual_seed(1337)
B,T,C = 4,8,32 # batch, time, channels
x = torch.randn(B,T,C)

#let's see a single Head perform self-attention
head_size = 16 # hyperparameter
key = nn.Linear(C, head_size, bias = False) # just simple mat mul
query = nn.Linear(C, head_size, bias = False)
k = key(x) # (B,T,16) == x(B,T,C) * k(C,head_size)
q = query(x) # (B,T,16) == x(B,T,C) * q(C,head_size)

wei = q @ k.transpose(-2,-1) # (B,T,16) * (B,16,T) --> (B,T,T)

tril = torch.tril(torch.ones(T,T))
# wei = torch.zeros((T,T))
# wei = wei.masked_fill(tril == 0, float('-inf'))
# wei = F.softmax(wei,dim=2) # along the row or use -1
out = wei @ x

out.shape
```

} #removed

```
Out[122]: torch.Size([4, 8, 32])
```

```
In [123]: wei.shape
```

```
Out[123]: torch.Size([4, 8, 8])
```

```
In [124]: wei[0]
```

```
Out[124]: tensor([[-1.7629, -1.3011,  0.5652,  2.1616, -1.0674,  1.9632,  1.0765, -0.4530],
                  [-3.3334, -1.6556,  0.1040,  3.3782, -2.1825,  1.0415, -0.0557,  0.2927],
                  [-1.0226, -1.2606,  0.0762, -0.3813, -0.9843, -1.4303,  0.0749, -0.9547],
                  [ 0.7836, -0.8014, -0.3368, -0.8496, -0.5602, -1.1701, -1.2927, -1.0260],
                  [-1.2566,  0.0187, -0.7880, -1.3204,  2.0363,  0.8638,  0.3719,  0.9258],
                  [-0.3126,  2.4152, -0.1106, -0.9931,  3.3449, -2.5229,  1.4187,  1.2196],
                  [ 1.0876,  1.9652, -0.2621, -0.3158,  0.6091,  1.2616, -0.5484,  0.8048],
                  [-1.8044, -0.4126, -0.8306,  0.5898, -0.7987, -0.5856,  0.6433,  0.6303]],
                 grad_fn=<SelectBackward0>)
```

so, w/o masking & softmax wei comes out like this raw affinities b/w all the nodes - But if I am at 5th node, I will not want to agg. anything from 6th node, 7th node & 8th node hence we use upper triangular masking so, those are not allowed to communicate.

uncomment masking

```
wei[0]
```

```
tensor([[-1.7629,    -inf,    -inf,    -inf,    -inf,    -inf,    -inf,    -inf],
        [-3.3334, -1.6556,    -inf,    -inf,    -inf,    -inf,    -inf,    -inf],
        [-1.0226, -1.2606,  0.0762,    -inf,    -inf,    -inf,    -inf,    -inf],
        [ 0.7836, -0.8014, -0.3368, -0.8496,    -inf,    -inf,    -inf,    -inf],
        [-1.2566,  0.0187, -0.7880, -1.3204,  2.0363,    -inf,    -inf,    -inf],
        [-0.3126,  2.4152, -0.1106, -0.9931,  3.3449, -2.5229,    -inf,    -inf],
        [ 1.0876,  1.9652, -0.2621, -0.3158,  0.6091,  1.2616, -0.5484,    -inf],
        [-1.8044, -0.4126, -0.8306,  0.5898, -0.7987, -0.5856,  0.6433,  0.6303]],
       grad_fn=<SelectBackward0>)
```

and now we want to have a nice distribution
so, we for ex don't want to agg '-0.11' that's crazy
so, instead we exponentiate & normalize
uncomment softmax

```
]: wei[0]
```

```
]: tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
           [0.1574, 0.8426, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
           [0.2088, 0.1646, 0.6266, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
           [0.5792, 0.1187, 0.1889, 0.1131, 0.0000, 0.0000, 0.0000, 0.0000],
           [0.0294, 0.1052, 0.0469, 0.0276, 0.7909, 0.0000, 0.0000, 0.0000],
           [0.0176, 0.2689, 0.0215, 0.0089, 0.6812, 0.0019, 0.0000, 0.0000],
           [0.1691, 0.4066, 0.0438, 0.0416, 0.1048, 0.2012, 0.0329, 0.0000],
           [0.0210, 0.0843, 0.0555, 0.2297, 0.0573, 0.0709, 0.2423, 0.2391]],
          grad_fn=<SelectBackward0>)
```

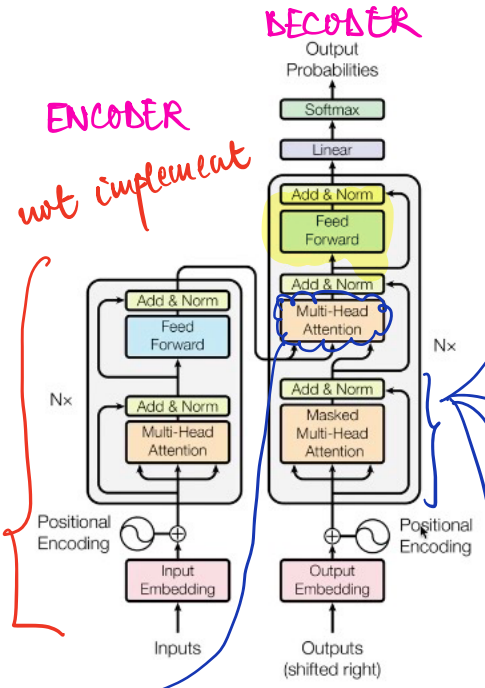
There's more to self attn' (single-self-attn' head)
when we do the aggregation we don't exactly aggregate the tokens. we aggregate produce one more value here called "value" [back to jupy] like Q & K we create (V)alue and we don't agg "x" we calculate a "v" which is achieved by propagating this nn.Linear() on "x" again & we output:

```
output = wei @ v
output.shape  [4, 8, 16]
```

you can think of x as private information to this token. so, x is private to the token. so, I am at 5th token. I have some identity and my info is kept in vector x. And for the purpose of single head, here's what I am interested in, here's what I have & if you find me interesting here's what I will communicate with you. and that's

stored in "v". And v is the thing that gets aggregated for the purpose of this single head. b/w diff nodes
And that's basically self-attn' mechanism
[summary of self-attention on JupNb]

After implementing Multi-Headed Attn' on Vscode



Now, we are finally getting close to the picture on the left.
We implemented the following
refer change 13 (VS code)

Positional encoding
self.pos_embedding_table
Embedding
self.token_embedding
Masked MultiHead Attn'
class MultiHeadAttention
self.lm_head # token to logits

↳ Here's another MultiHeaded Attention which is a cross attention to an encoder which we won't implement in our case [we are going to come back to that LBR]

But I want you to notice that there's a feed forward path and then this is grouped into a block that gets repeated again & again
So, the FF part here is simple MLP.
"This adds computation to the network and this computes on per-node level"
[Implement on Visual Studio]
Change 15

3.3 Position-wise Feed-Forward Networks

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (2)$$

While the linear transformations are the same across different positions, they use different parameters from layer to layer. Another way of describing this is as two convolutions with kernel size 1. The dimensionality of input and output is $d_{\text{model}} = 512$, and the inner-layer has $d_{ff} = 2048$.

So, before MLP, we had the multiheaded self-attn' that did the communication but we went way too fast to calculate logits. So, the tokens looked at each other but really didn't have time to think on what they found from the other tokens.

Hence, we implement single FFWS.

Notice that feed forward when its applying linear, this occ on the per token level. All the tokens do this independently. So, the self attn' is the communication & then once they have gathered all the data, now they need to think on that data individually. And that's what feed forward is doing.

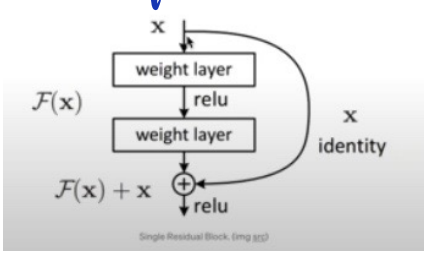
When we train this, the val loss continues to go down ~ 2.24 Although outputs still weird.

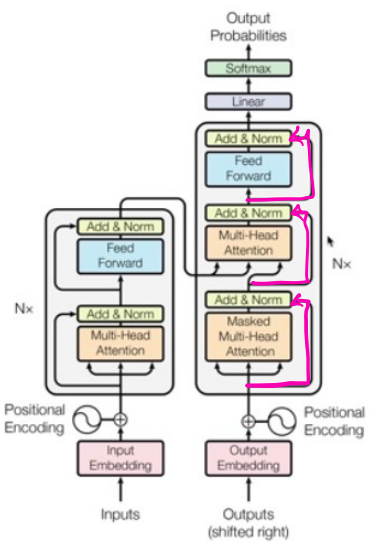
Implementation of Block # change 16

When we just implement these, these don't produce very good results because they are very deep NN & face optimization issues. So we need one more idea to resolve the difficulties.

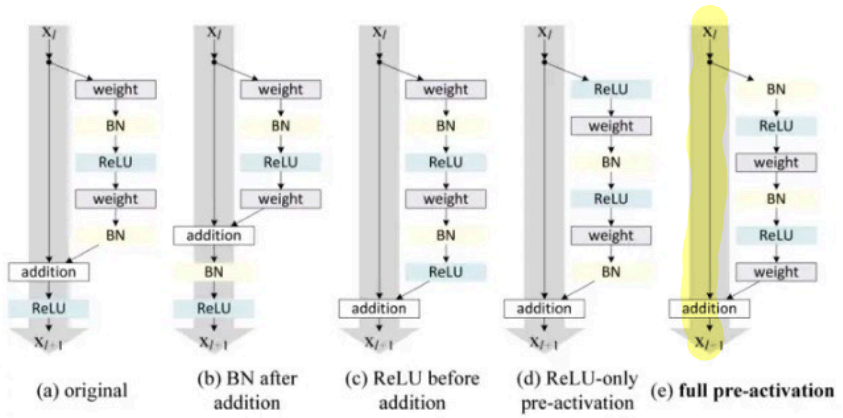
There are namely 2 techniques:

1) Skip connection / Residual conn - you transform data





you transform data but then you have a skip connen' with addn' from the previous features
The intuitive visualization



Types of Residual Block. (img src)

computation from top to bottom & you have this residual pathway and you are free to fork off from the residual pathway, perform some computation and then project back to the residual pathway via addn'. So, you go from inputs to targets only via plus, & plus —
And the reason this is useful is because during back prop, addn' distributes gradient equally to both of it branches that fed as the input. So, from the gradients from the loss, hop through every addn' node all the way to the input & then also fork off into the residual blocks. Basically, there's a gradient "super highway" that goes directly from the supervision all the way to the input unimpeded. And then these residual blocks are usually initialized in the beginning. So, they contribute vv little if anything

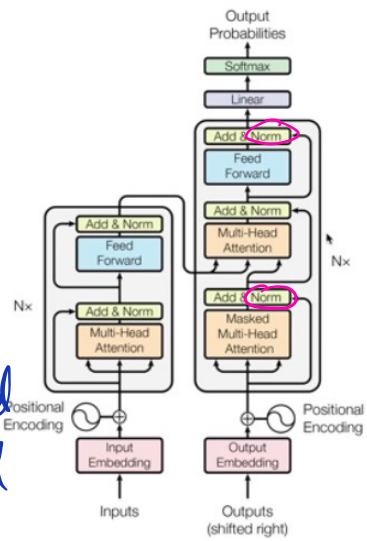
to residual path way.

CHANGE RESIDUAL

And we train this we actually get down to 2.08 we also notice that our train_loss is getting ahead of val_loss (maybe overfitting)

II Norm layer_norm

Layer norm quite similar to batch norm BN made sure that across the batch dimension any individual neuron had unit gaussian distribution (0 mean, 1 var)



On Layer Norm, cols won't be normalized but rows will now be norm as opposed to batch norm.

One thing to note which is not in sync with paper, we see that add & norm are applied after the transformation but now it is more common to apply 'layer norm' before the transformation. So, there's reshuffling of layer norms. This is called pre-norm formulation and this is what we implement

CHANGE LAYER NORM

After these changes we get loss to 2.06 which is slight improvement

NOTE: there's one layer norm at the end of the transformer too!

Scaling the model

Adding dropout - overfitting avoid

changing hyperparameters

384/6 => 64

After scaling we get val_loss as 1.50 which is quite impressive

Encoder Vs Decoder Vs both

Encoder - Decoder : Machine Translation

We are only using Decoder which unconditioned on anything. Just generating.

what makes it a decoder, is that we're using a triangular mask in our transformers so it has this Autoregressive property where we can just go and sample from it

Encoder - no triangular mask (bi-directional)
(French)

↳ Decoder cross attn' $\xrightarrow[\text{out}]{}$ English

nanoGPT

train.py — model.py

↳ all the boiler plate code (complicated) ↳ causal self Attention block

Train ChatGPT

1) There's a pre-training stage and then a fine-tuning stage (Document completer)

In pre-training, we are training on large chunk of internet and trying to get a decoder only transformer to babble text. [This is what we did]. 10Mil parameters & our dataset is about 1 mil tokens. But you have to realize that openAI uses diff vocabulary they are not on character level They use these sub-word chunks of word 50,000 vocab_size

The model is nearly identical to ours.

when we get at pre-training stage we don't get something that responds to your questions with answers and its not helpful. You get a document completor [more txt] it babbles internet.

2nd stage fine-tuning is what makes it an assistant

There are roughly 3 steps to it.

step1 : collect demonstration data and train a supervised policy