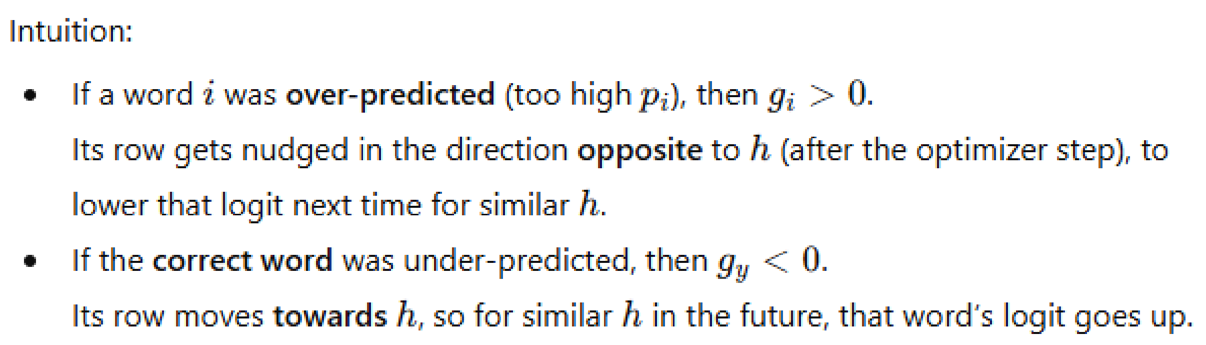
2025-12-15

"So $\frac{\partial L}{\partial W_{ij}}$ tells us how fast the performance deteriorate with the parameter (setting), the bigger it is, the worse it is." But what is this W_{ij}? Is it just a parameter? Come on, it's the j^{th} meaning component of the i^{th} token! So yes, we are not even sure how to fill up the meaning components for each token! This is the way to learn them! So what is the performance related to, and how? Well, it's related to the 1) probability of the $i^{th}\ token\ p_i$; 2) probability of the j^{th} intended meaning component h_j. And how exactly is the performance related to the probability and the intended meaning component? Well, the greater they are, the worse the performance. There are two ways to tune the parameter W: up or down. I think naturally the $\frac{\partial L}{\partial W_{ij}}$ is about how fast L changes with W's *increase*. So if this number is positive, it means the performance worsens (with the increasing W), if the number is negative, it means the performance improves (with the increasing W). To determine its sign, we need the signs of the two factors: p_i, and h_j. The probability term must be positive, but what about the intended meaning components? Nothing prevents it from being negative? If the intended meaning components is positive, then performance deteriorates with the increasing W, so we should dial down W, if the intended meaning components is negative, then the performance improves, so we should dial up W. Hmm, think about this: do we really have no idea this W should be? I think we do: it should be as close as to the j^{th} meaning components as possible! But wait a minute, what if it is the wrong token to start with? Ok, then we need to compare how the tunning differs whether it's the true token or not:

- $\frac{\partial L}{\partial W_{yj}} = (p_y - 1)h_j$ if the wrong token
- $\frac{\partial L}{\partial W_{ij}} = p_i h_j$ if the right token

If we ignore the probability term for a second, the performance change is in the opposite of the meaning component: because if the meaning component is **negative**, the loss decreases with the parameter increase, hence **dial up**; if the meaning component is **positive**, the loss increases, hence **dial down**. But this is true depends on if it's a true token:

- If it is the true token, if intended meaning component is negative, dial down, if meaning component is positive, dial up.
- If it is not the true token, then if the meaning component is negative, dial up, if the meaning component is positive, dial down.
  Hmmmmm…. Why such distinction???
  Seems that it is assuming we naturally start with 0 for the parameters? So go negative means dial down, go positive means dial up???



Intuition:

- If a word i was **over-predicted** (too high p_i), then $g_i > 0$.
  Its row gets nudged in the direction **opposite** to h (after the optimizer step), to lower that logit next time for similar h.
- If the **correct word** was under-predicted, then $g_y < 0$.
  Its row moves **towards** h, so for similar h in the future, that word's logit goes up.

So logit is the similarity number, coming from the product of the token's jth meaning component and the intended jth meaning component. As GPT explains, the direction of dialing is very simple: if it is not the true token, you want that similarity number to drop, so if h is positive, you want to reduce the meaning component. Hmm, I don't quite see it. So we have $W_{ij} \cdot h_j$ as the logit component, we want to reduce this product, should we just dial down W_{ij}? Well, that depends on the sigh of h_j: if it is positive, dial down W_{ij} , but if it is negative, dial up. Ok, that is in line with our previous conclusion, although not very intuitive.

Ok, so now we know how to tune W_{ij} . Next, how to tune the 'bias' b? Remember $z = W_{voc}h$+b. So now we need to figure out how to tune the constants b. Again using chain rule, $\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial z_i} \cdot \frac{\partial z_i}{\partial b_i}$, which is just $\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial z_i} \cdot 1$=g. GPT calls it the 'error vector'. Hmm all right, makes sense. Because the probability for any false token is indeed the error, the difference between the true token's probability and 1 is the error, so yes, the g is the error vector. Then what?

Then calculate how fast the performance changes with the intended meaning h. hmm, wait, this does not sound right. I mean, you are saying, non of these are fixed, all of it must be tuned/learnt/searched? Ok we kind of knew it. So, now we want to know how the performance changes with the intended meaning. The math is easy: $\frac{\partial L}{\partial h} = W_{vocab}^T g$. We know it should be a column vector, is it? W_{vocab}^T is 1024x50000, g is 50000x1, hence the result is indeed 1024x1, telling us, how fast performance changes with each meaning component.

Learning0to1

2025-12-16

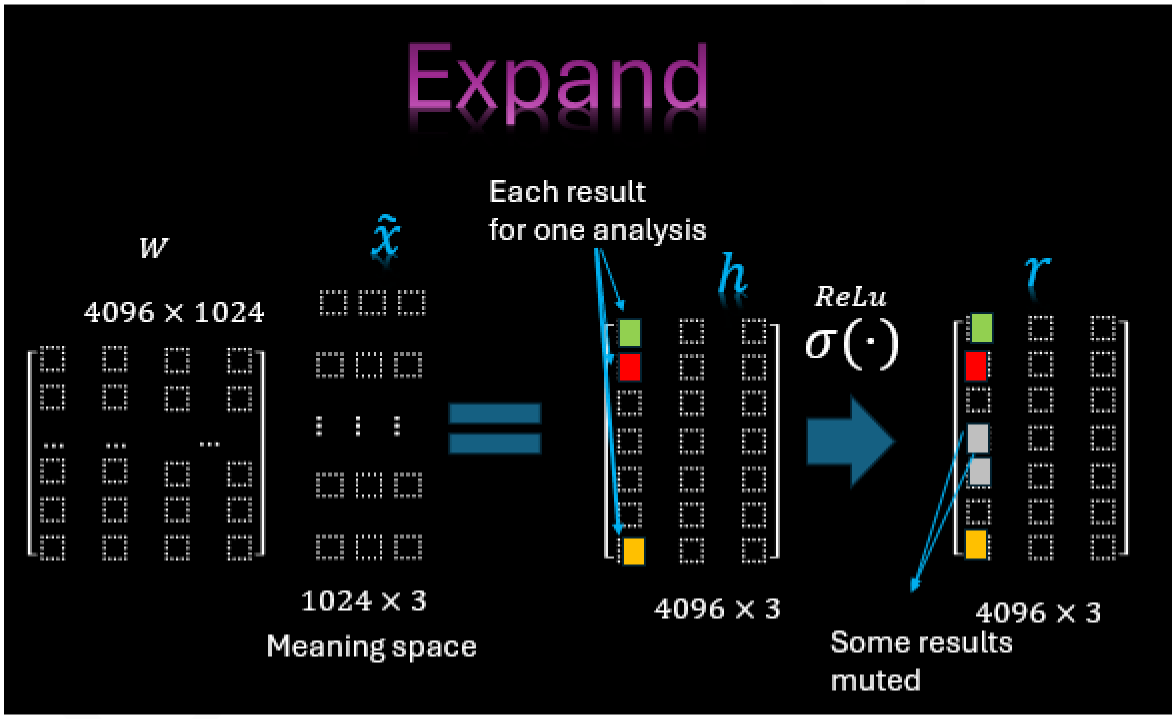
Ok, so this intended meaning h goes through the web of complexity to finally contribute to the output. Differentiation is easy: $\frac{\partial L}{\partial h_j} = \frac{\partial L}{\partial z_i} W_{ij}$. Which is $W^T g$, where g is the 'error vector'.

Ok, the previous two gradients are easy: how performance changes with the bias and vocabulary entry. Now I want to see how the performance change regarding the hidden layer h gets pushed back. So let's do it.

Let's see how much we can remember.

So let's trace 'h' through what it's being through. So first, $h = x_1 + m$, hmm, what is this x_1? Say we agree with it, so h comes from x_1 and m, now we need to find out how performance changes with x_1 and m. and what is MLP? Multi-layer ??? looks like it's the feed-forward step now. Ok need to pull my notes.

So the combined (contextual) meaning is first going through expand, then Relu process will mute some result:

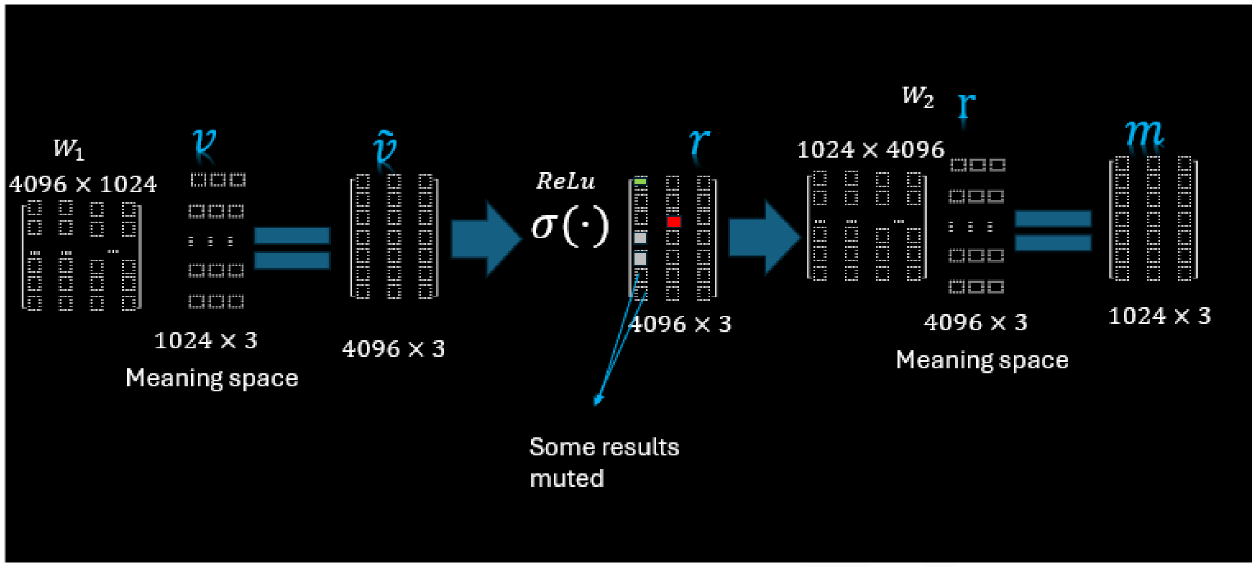


Let's check what GPT says.

First, $h = x_1 + m$, seems that m comes from the multi-layer P? So the gradient with respect to x_1, m should all be the same as the gradient for g_h. Next, drill down to m, this seems to come from the MLP, GPT changes notations again, how annoying!

But seems that v is the input to this stage, first the expansion stage: $W_1 v + b_1$, then applied Relu (Why he wrote GELU???) Let's update out notations:

So the goal is to get the gradient with respect to each tunable: W_2, W_1, b_1, b_2, v(input).



Then step C: back through second layer norm LN2. Hmm, $v = LN_2(x_1)$, I don't think this is the same x_1 as $h = x_1 + m$, we can check later. So this input v comes from a layer norm operation on x_1, and I have already forgotten what this layer norm operations is, and what x_1, but I think it comes from the previous attention step? So if x_1 is from previous step, then obviously we need to know the gradient with respect to x_1. So there should be a differentiation with respect to the layer norm function. Ok, then what?

Step D: combine gradients at x_1.

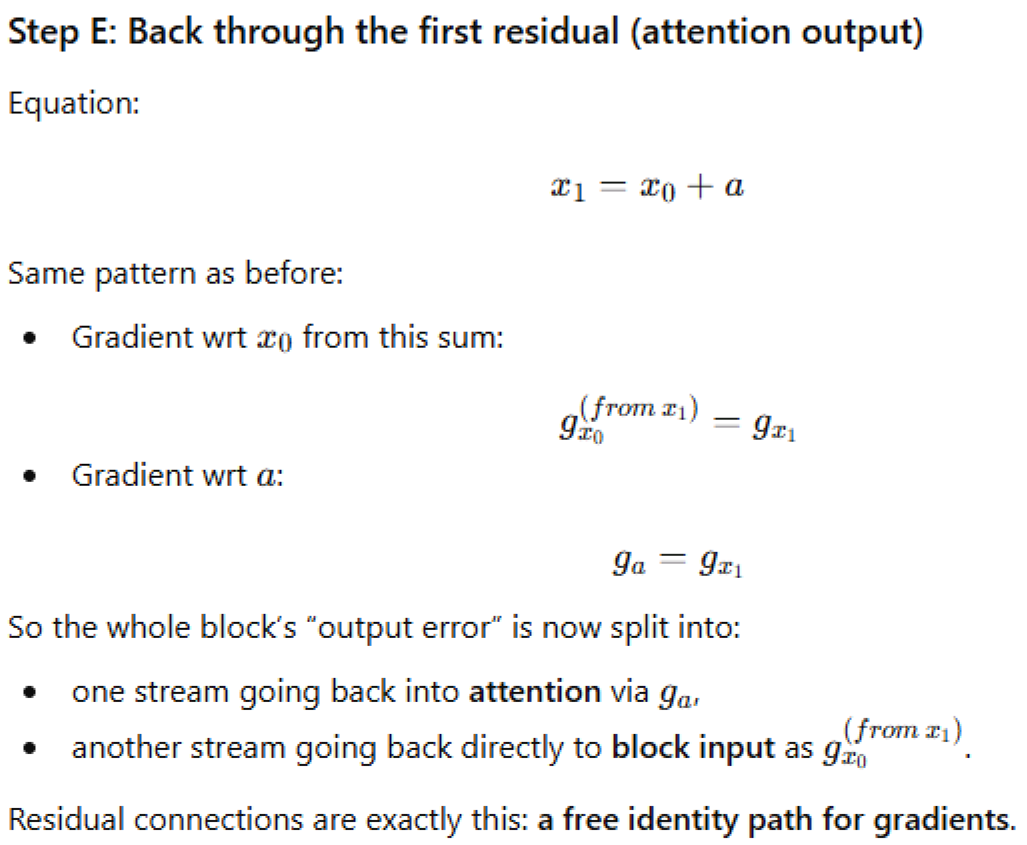
Ok, here we can see what I said earlier was wrong: x_1 are indeed the same at two instances:

x_1 has **two paths** to the loss:

1. Direct to h: $x_1 \to h$.
2. Through LN2 & MLP: $x_1 \to v \to m \to h$.

So seems that the two paths are alternative? So we need to clarify what x_1 is, and what h is. But reading what GPT wrote, does not sound like an option, because otherwise, why would you sum the two? And h is the intended meaning, don't forget that. Let's keep reading.

Learning0



Step E: Back through the first residual (attention output)

Equation:

$$x_1 = x_0 + a$$

Same pattern as before:

- Gradient wrt x_0 from this sum:

$$g_{x_0}^{(from\ x_1)} = g_{x_1}$$

- Gradient wrt a:

$$g_a = g_{x_1}$$

So the whole block's "output error" is now split into:

- one stream going back into **attention** via g_a,
- another stream going back directly to **block input** as $g_{x_0}^{(from\ x_1)}$.

Residual connections are exactly this: **a free identity path for gradients.**

So x is from the attention block, I think I am missing the residual piece, that's why I found weird. Looks like 'a' is straight from attention scheme,

2025-12-17

Ok, so up to now we know how the performance changes with h, the intended meaning: $\frac{\partial L}{\partial h} = W_{vocab}^{T} g$. But where does this intended meaning come from? It comes from feed forward step, which takes the contextual meaning *v* into intended meaning *h*. So now we want to find out how the performance chained back to this contextual meaning v.

I guess we can forget about the details for now, but the punchline is, we can trace all the way back to find out how the performance changes with the contextual meaning **v**. Next is the trace the contextual meaning ***v*** all the way back. Ok the formula $h = x_1 + m$, the MLP is through m, hence I would consider m as the direct output from the feed forward layer. But now seems that step C is treating the x_1: $v = LN_2(x_1)$. So is this x_1 the contextual meaning? But it's added to the figured-out response. Ok, so here is what I missed before. I thought the contextual meaning is fed into the feed-forward step for an intended meaning, but actually it's not the intended response, but the intended response adjustment.

So the contextual meaning will go through the feed-forward step to figure out a response adjustment m, that m will be added to the contextual meaning to create the intended meaning. Hmm, interesting, so my question is: how people figure out we need to create an adjustment to update the contextual meaning in order to get the intended meaning**? Isn't it more natural to think about using the contextual meaning to figure out the intended meaning directly???** Maybe a better question is: does people know this at first? Or they first try to intended meaning creation directly, then because of the poor performance, they decide to use is as adjustment instead of outright??? Anyway, since this v is not the actual input, but rather a processed input from layer norm operation, $v = LN_2(x_1)$, now we need to get from $\frac{\partial L}{\partial v}$ to $\frac{\partial L}{\partial x_1}$, then we need $\frac{\partial v}{\partial x_1} = LN_2'(x_1)$. So the overall gradient in terms of x_1 is $\frac{\partial L}{\partial v} \cdot \frac{\partial v}{\partial x_1} = g_v LN_2'(x_1)$. So I am expecting a product of g_v and the derivative of this linear norm function with respect to x_1, I am not expecting an embedded derivative as $LN_2^*(x_1)$, and what does that even mean?

So how does the contextual meaning impact the performance? Two ways, first it is used to create an adjustment *m,* then it's added to the adjustments to create the intended meaning h. GPT says the total gradient is the sum:

$$g_{x_1} = g_{x_1}^{(from\ h)} + g_{x_1}^{(from\ LN_2)} = g_h + \mathrm{LN}_2^*(g_v)$$

But to be honest, is the sum a good way to combine the impact from x_1? I mean, sure you have two branches, but are they of the same weights? But let's accept it and move on. Now trace this contextual meaning x_1 back: $x_1 = x_0 + a$. x_0 is the original meaning? And a is the result from the attention block. So given this structure, gradient with respect to x_o and a will be the same as for x_1. And GPT mentions that now the error goes back through two streams, the input x_o, and the attention a. So let's check the attention route.

So the input to this block is X, the input tokens (you see how important it is now to work out the forward sequence instead of backward sequence directly since it's been a while?) X goes through three linear processes to compute the Query, Key and Value.

2025-12-19

GPT’s habits of constantly switching symbols drive me crazy! The “O” here is the ‘a’ before, which is added back to the input of the block.

We need a recap for this attention step, the dataflow. The purpose is to blend the words together to form a ‘contextual meaning’, a, or, the contextual meaning adjustment. That will be used to figure out the intended meaning I think. Omg, the notations are so confusing!

Learning0to1