

2025.12.02

Ok, now let's see the impact of the head on the value. That is, if a token is selected for this head, what value should it send to others? I guess it's better to understand it with contrast. Say, the token 'apple', it could mean the tech company Apple, or, a simple fruit that can be eaten. Show me what values would be sent under these two different cases?

gimme the possible 16 heads (perspectives), only from Key's perspective

Thought for 12s >

Here are 16 "Key-side" perspectives—i.e., what a token might *advertise* about itself to be matchable:

1. **Token identity / subword:** "I am this exact piece (##ing, 'Apple', '(')."
2. **Orthography:** "Init-cap/digits/punctuation; word-start or continuation."
3. **POS candidacy:** "Likely noun/verb/adj/adp/etc."
4. **Morphology (noun):** "Count/mass; singular/plural; case."
5. **Morphology (verb):** "Finite vs. non-finite; tense/aspect/mood."
6. **Common vs. proper noun:** "I behave like a name vs. a common noun."
7. **Named-entity type:** "ORG/PER/LOC/PRODUCT/etc."
8. **Semantic class:** "Animate/edible/location/abstract/material..."
9. **Modifier potential:** "I can modify a noun (adj/compound-noun-ish)."
10. **Head-noun potential:** "I accept determiners/adjectives/complements."
11. **Determiner/'a/an' cues:** "Definiteness; vowel-start sound; countability."
12. **Selectional fit:** "Good as subject/object for certain verbs (edible for 'eat', etc.)."
13. **PP/complement frames:** "I expect/allow 'of/for/with...' attachments."
14. **Boundary/closure:** "I open/close (quotes, parens) or mark clause breaks."
15. **Positional cues:** "Sentence start/end; list/quote interior."
16. **Coref anchorability:** "Salient, singular/plural, gendered/non-human—good antecedent."

Ok, now let's see the impact of the head on the value. That is, if a token is selected for this head, what value should it send to others? I guess it's better to understand it with contrast. Say, the token 'apple', it could mean the tech company Apple, or, a simple fruit that can be eaten. Show me what values would be sent under these two different cases?

Great—let’s contrast **what Value (V) carries** for “Apple”=ORG vs “apple”=FRUIT across a few common heads. (Remember: these are *learned mixtures*, not literal flags.)

Head (perspective)	V payload if Apple = ORG	V payload if apple = FRUIT
Sense/Entity typing	entity=ORG, proper-name vector, non-human, brand semantics (products, revenue, CEO)	semantic=FOOD/FRUIT, concrete object, edible, physical properties (peel, sweet, red/green)
Subject–Verb agreement	number=SG, animacy=non-human, organization-as-agent tendencies	number=SG, animacy=non-human, inanimate-theme tendencies
Verb–argument selection (who can do/undergo what)	agentive/org-capable actions (announce, acquire, ship); resists being “eaten”	edible theme; fits verbs like eat, slice, peel; resists “announced” as agent
Induction / continuation (collocations)	“Inc.”, “announced”, “shares”, “iPhone”, “earnings”, “WWDC”	“pie”, “juice”, “tree”, “ripe”, “sweet”, “eat”, “peel”
Modifier/compound attachment	brand-modifier payload: “Apple Watch”, “Apple Store”, “Apple engineers”	noun-as-modifier payload: “apple pie”, “apple juice”, “apple seeds”
Coreference / entity carry	stable org identity vector; good antecedent for “it/they (org)” depending on style; sector=tech	stable physical-object/food vector; antecedent for “it” (inanimate); countable item

Takeaway:

- **Same surface token, different V per sense and head.**
- Q/K handle *who matches whom* (e.g., “seek an org vs an edible thing”);
- V carries the *useful content* once matched: org semantics vs fruit semantics, agreement features, typical continuations, etc.

How to understand the difference between value and the original meaning?

for the word apple, can i say the original meaning indicate it could be both an organization, and also a fruit? it could be edible, or it can announce?

Yes—with care.

- **Early in the layer stack:** X_{apple} is a **superposition of hypotheses** (ORG + FRUIT + general noun stuff). Think “both are possible, with different weights.”
- **Context resolves it:** Attention (Q:K) pulls in neighbors; the update shifts X :
 - “...announced earnings” → boosts **ORG**, suppresses **FRUIT**.
 - “...ate an apple” → boosts **FRUIT/edible**, suppresses **ORG**.
- **Case & form bias:** “**Apple**” (cap) starts biased to **ORG**; “**apple**” (lower) to **FRUIT**, but context can override.
- **Why keep both at first?** It lets the model delay commitment until enough evidence arrives; later layers sharpen to one sense, though a faint trace of the other can remain (useful for uncertainty).

So: the **original meaning** X can encode “could be an organization **or** a fruit; could **announce** or be **edible**.” Each head then extracts a V tailored to its job (entity typing, selectional fit, collocation, etc.) to push X toward the right sense.

Ok, the challenge with this whole thing is:

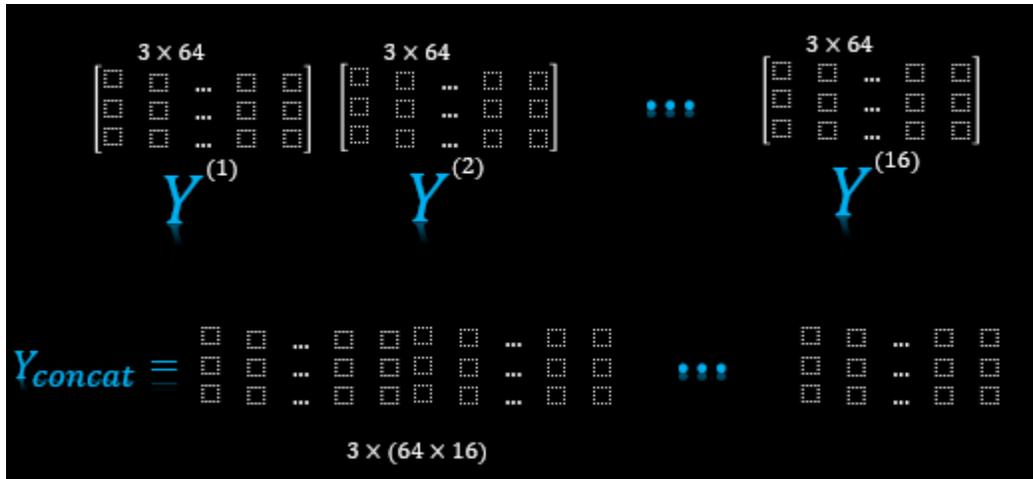
- 1) We don't know what needs to be represented
- 2) We don't how to represent them

But I think I am comfortable with the difference between K and V, as K is more general V is more specific. So let's move on.

Now we have the attention matrix 3×3 , and the value matrix 3×64 (assuming the feature under each head has 64 dimensions). Now we multiply A by value: $Y = AV$. **The** mechanics of the multiplication is actually doing the weighted average of the values! So it is blending the tokens according to weights. But this weight is from a single head's perspective. Remember there is 16 such heads (we partition 1024 into 16 heads, each with a size of 64). So we have 16 such processes: $Y^{(1)} = A^{(1)}V^{(1)}$, $Y^{(2)} = A^{(2)}V^{(2)}$..., $Y^{(16)} = A^{(16)}V^{(16)}$. So from different matching perspective, token will have different weights, that's why there are 16 processes doing the mixture. So what are these? Each row is the blending meaning under a perspective. But wait, within each head, what does three rows mean? I think that's the blending meaning from a token's perspective! Remember, the weights are assigned from a

token's perspective! So first row is the blended meaning from first token's perspective, second row is the blended meaning from the second's perspective and so on.

Then we concatenate these 16 blending meaning, into the bigger 'matrix', as follows.



Although it is a big matrix now, bear in mind the 'boundaries', there are 16 chunks, representing 16 sets of blending meanings from different perspective (head). So now it's time to further blend them into a coherent background. That's what an output matrix W_o is doing here. It blends the 16 contextual meaning further! But why three rows again? Think about it, if we evaluate a sentence and create a blended meaning, should it just be one? Here why three? To me I think that's from each token's perspective! We can verify with GPT later. Now we have the output ready for feedforward step! Not yet! Instead of the blended meaning, it's added to the original meaning (embedding)!

- Then: residual add $X + Y_{out} \rightarrow \text{LayerNorm} \rightarrow \text{FFN} \rightarrow \text{residual} + \text{LayerNorm}$.

So the original meaning is not tossed away! The blended meaning is used to update the original meaning! Ok.

But what is layernorm? Did a search, this word did not show up in previous discussion! So it is another nitty gritty detail!

what is layernorm?

LayerNorm (Layer Normalization) = per-token, per-layer re-scaling of features to stabilize training.

How: For a token vector $x \in \mathbb{R}^d$ (e.g., $d = 1024$):

$$\mu = \frac{1}{d} \sum_{k=1}^d x_k, \quad \sigma = \sqrt{\frac{1}{d} \sum_{k=1}^d (x_k - \mu)^2 + \varepsilon}$$

$$\text{LayerNorm}(x) = \gamma \odot \frac{x - \mu}{\sigma} + \beta$$

- Compute mean/variance **across features of that token** (not across batch/time).
- γ, β are learnable scale/shift (same size as x).
- ε is a tiny constant for numerical stability.

Why in transformers:

- Stabilizes activations/gradients across deep stacks.
- Makes model less sensitive to feature scale (helps attention/FFN).
- Works with variable sequence lengths and small batches (unlike BatchNorm).

Where: Applied around each sublayer (attention, FFN) with residuals (Pre-LN or Post-LN variants).

i am at the residual add where blended meaning is added to original embedding X, show me how layernorm is applied to 'it' (what?)

Thought for a few seconds >

Great q. There are two common orders—**Pre-LN** (most modern GPTs) and **Post-LN** (older).

Pre-LN (usual in GPTs)

Normalize **before** the sublayer; the residual sum itself is *not* normalized.

$$\begin{aligned}
 x_0 &= X \\
 \Delta_{\text{attn}} &= \text{MHA}(\text{LN}(x_0)) \\
 x_1 &= x_0 + \Delta_{\text{attn}} \quad (\text{residual add}) \\
 \Delta_{\text{ffn}} &= \text{FFN}(\text{LN}(x_1)) \\
 x_2 &= x_1 + \Delta_{\text{ffn}}
 \end{aligned}$$

Here, when you “add blended meaning to X”, LayerNorm is applied to X **before** MHA, not to the sum.

Post-LN (older)

Normalize **after** each residual sum.

$$\begin{aligned}
 x_1 &= \text{LN}(X + \text{MHA}(X)) \\
 x_2 &= \text{LN}(x_1 + \text{FFN}(x_1))
 \end{aligned}$$

So, the “it” you normalize depends on the variant:

- **Pre-LN:** normalize the **input** to the sublayer (e.g., $\text{LN}(X)$); residual sum is left unnormalized.
- **Post-LN:** normalize the **sum** (e.g., $\text{LN}(X + Y_{\text{out}})$).

I don't understand these weird notations! How come it just assumes I already knew???

What is $\text{MHA}(\text{LN}(x_0))$? What is $\Delta_{\text{attention}}$? Are you saying, prior to the attention step, we should normalize the tokens' embedding? But not exactly the same! So the 1024 numbers are averaged out to compute the mean and the standard deviation. Then they are used to normalize the original token, but I

would assume just $\frac{x-\mu}{\sigma}$, why multiply by some parameter and add some parameter? And what is that circle with a dot inside mean??? once tokens are normalized, then the normalized ones are going to be used in attention step?

so what goes to the attention calculation is the the original tokens, but the normalized ones?

Yes (in Pre-LN models).

- **Pre-LN (common in GPTs):** Attention takes **LayerNorm(X)** (normalized hidden states). Q/K/V are computed from these normalized vectors. The **residual** adds the unnormalized X back afterward.
- **Post-LN (older):** Attention takes X directly; normalization happens **after** the residual sum.

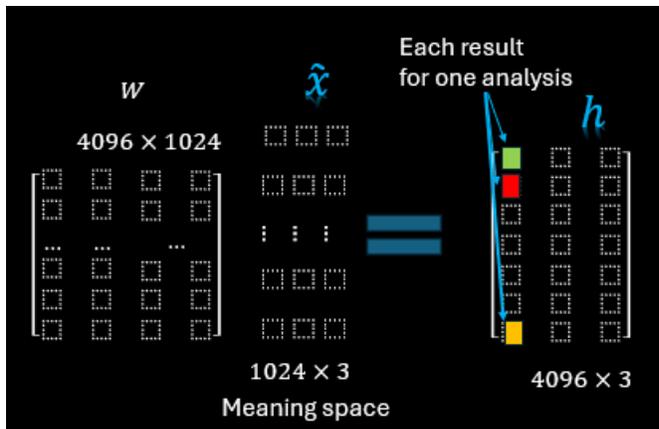
(Here X is the current hidden state—embedding+position in layer 1, refined states in later layers.)

OK I kind of getting the idea: whether to normalize it before the attention step or after. O man, so many piecemeal patches! I bet it's after they experienced some issues they found that they can apply some pre or pos-processing to 'stabilize' it, without very clear about the principles! Ok. fine. Let's move on.

So let's assume we do the post-linear-norm, that is, we apply linear normalization to the output after the attention step. So I guess instead of adding Y_{out} to X directly, we apply linear norm to Y_{out} , represented by $MHA(LN(X))$?, but wait, 2, we are applying linear normalization again?

2025.12.03

GPT flipped the matrix multiplication notations, super annoying! Tokens X was on the left side before, now it's on the right side! So now we have to flip the matrix too! And I hate interpreting matrix-matrix multiplication by dimension matching! The matrix W is 4096×1024 , what does 4096 mean and what does 1024? I'd like to interpret the 4096 as '4096 processes to analyze the 1024 meaning components'. The end result is 4096×3 , again, what do those numbers mean???



So each column of x is the 'contextual meaning', still with 1024 meaning components. Now there are 4096 'processes' to analyze these meaning components, by 'linearly mixing them up'. The result would be 4096 results since we have 4096 'analysis'. But why 3 columns? Definitely they are tied to 3 tokens, but in what way? They are not representing 3 tokens! They represent the analysis results from the perspective of each token! Remember each row of the attention matrix represents the weights from a single token's perspective! H is called 'expand' because its dimension is 4096, due to 4096 processes! Then what?

Then we 'filter' the result, using ReLu. So my understanding is, each element in the h matrix will be filtered based on if they are negative or not. I am not sure what the exact scheme is, ie, will positive numbers remain the same, or scaled, or what? But now we have a filtered analysis r . Although I was excited to see how the training/learning work to find out those parameters, I still want to conclude about the feedforward step, which is much easier than the attention step.

Where we are

After attention:

1. $x_1 = X + \text{MHA}(\text{LN}(X))$ (residual)
2. $\tilde{x}_1 = \text{LN}(x_1)$ (normalize for FFN)

FFN steps (per token)

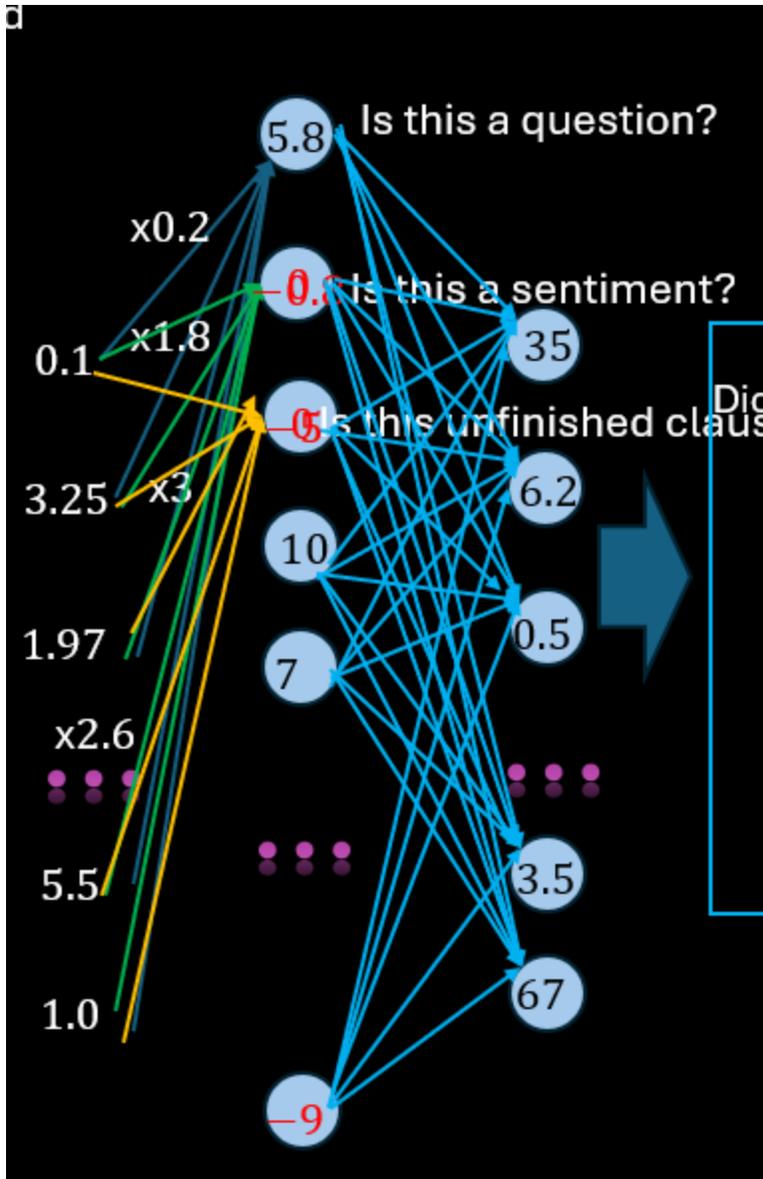
3. **Expand:** $h = W_1 \tilde{x}_1 + b_1$ ($W_1 : d_{\text{hidden}} \times d_{\text{model}}$, e.g. 4096×1024)
4. **Nonlinearity/Gating:** $r = \sigma(h)$ (GELU/ReLU or gated variants like SwiGLU)
5. **Recombine:** $z = W_2 r + b_2$ ($W_2 : d_{\text{model}} \times d_{\text{hidden}}$)
6. **Residual:** $x_2 = x_1 + z$

Now $x_2 \in \mathbb{R}^{N \times d_{\text{model}}}$ is the **intended-meaning** state for this block. Then you either:

- pass x_2 to the **next transformer block**, or
- (at the top block) take the **last token's** vector z_{last} , do $W_{\text{out}} z_{\text{last}} \rightarrow \text{softmax} \rightarrow \text{next-token probs}$.

That's the complete feed-forward in the correct place in the flow.

Lets not bother with the details about where exactly the linear norm is applied. let's look at the big picture. It's been a while and I almost forgot, fortunately GPT reminded me of the 'intended-meaning'. If you look at the above, there are mainly 2 steps, the expand+gating, and the recombine. It would be confusing if you just focus on the math: a bunch of linearly mix. 3. Expand is linearly mixing the 'contextual meaning', it is 'figuring out the meaning'! So the attention step combine the meaning according to contexts, producing the end result of 'contextual meaning' ready for interpretation. Now these 4096 processes are 'analyzing' the contextual meaning! Like the old picture I had, this is to figuring the meaning, the interpretation phase! Then the gating step can be interpreted as checking if certain pattern if met, if not met, then it's muted (by turned into 0).



Then the recombination step can be understood as 'figuring out a response', in the meaning space, according to the figured out meaning! So the response has all the necessary meaning components filled in. Then the next step is to find out the corresponding word in the dictionary. But we might not get exact match, that's why dot product is used: to find out the best match! That's evaluated by computing a matching score, the bigger the number, the better the match, but again it's normalized within (0,1) to be used as a probability number. Once we have probability number for each word, then it's time to pick one. The simplest is to pick the highest value, but it's boring and repetitive. So GPT suggests some sampling, I guess it is to pick the greatest value plus some room for difference. I am not exactly sure how that's done, but it's not my concern for now. I can't wait to get to the training part!!! Let's do it!!

Learning0to1

is this W_o matrix the same as the embedding matrix?

No. Different matrices:

- W_O : mixes the **concatenated head outputs** back into the shared meaning space (shape 1024×1024). It lives **inside attention**.
- **Embedding matrix W_E** : maps **token id** \rightarrow **vector** (shape $V \times 1024$).

Related but separate note:

- W_{out} (vocab/output projection, shape $V \times 1024$) is sometimes **tied equal to W_E** .
- $W_O \neq W_E$ and $W_O \neq W_{out}$.



then what is that residual add $X+Y_{out}$? Is X the pure meaning?

Short and clear:

- **What is $X + Y_{out}$?**
A **residual connection**. X is the block **input** (the current token representations before this attention sublayer). Y_{out} is the **attention update**. Adding them gives "keep what I had + a learned correction."
- **Is X "pure meaning"?**
 X is the **current meaning state** from the previous step/layer (including positional info and all prior refinements). Only in the very first block does X come straight from **embedding + position**.
- **Why do this add?**
 - Preserves information (identity path).
 - Makes attention act like a **delta/patch** to the representation.
 - Greatly stabilizes training and gradient flow.
- **Then what?**
Apply **LayerNorm**, then the **Feed-Forward** sublayer, then another residual add.

2025-12-04

we can go into the detail later. now i am curious about the training step: how on earth we find out all those parameters?

Great—here’s the big-picture of **how training finds all those parameters** (every $W_Q, W_K, W_V, W_O, W_1, W_2, W_E, W_{\text{out}}$, etc.):

The core loop (next-token prediction)

1. **Init weights** randomly (small values).
2. **Batch text** → tokenize → for each sequence (x_1, \dots, x_T) create targets (x_2, \dots, x_{T+1}) (“teacher forcing”).
3. **Forward pass**: embeddings → (repeat blocks: MHA + FFN) → logits $\ell_t = W_{\text{out}}z_t$.
4. **Loss**: average **cross-entropy** between logits and true next tokens:

$$L = -\frac{1}{T} \sum_t \log \text{softmax}(\ell_t)_{x_{t+1}}$$

5. **Backprop (autodiff)**: compute gradients $\nabla_{\theta}L$ for all parameters θ .
6. **Update**: optimizer (typically **AdamW**) applies:

$$\theta \leftarrow \theta - \eta \cdot \text{AdamW}(\nabla_{\theta}L)$$

(with weight decay).

7. Repeat over **billions** of tokens (many epochs/passes) → weights align to minimize loss.

What this does intuitively

- If the model **overweights** irrelevant tokens, gradients **decrease** entries in the corresponding W_Q, W_K .
- If a head **helps** predict the right next token, gradients **increase** the useful directions in W_Q, W_K, W_V .
- FFN W_1/W_2 learn which nonlinear features to **activate** for better predictions.
- Embeddings and W_{out} align so that “intended meanings” point toward the **correct next word**.

The notations are a bit confusing. Logits ℓ_t should be a vector of 50,000, representing how similar the intended meaning is with the existing 50,000 words. The subscription t represents it’s the prediction based on the t^{th} token. This is the similarity measurement before normalization using softmax. But why

represent it as $\text{softmax}(\ell_t)$ instead of z_t with z being the normalized one? Then what's the additional subscription $\text{softmax}(\ell_t)_{x_{t+1}}$ mean?

ok, then what does $\log \text{softmax}(\ell_t)_{x_{t+1}}$ mean?

It means: **take the log-probability the model assigns to the true next token.**

Formally,

- $p_t = \text{softmax}(\ell_t)$ (a vector over vocab),
- $\log \text{softmax}(\ell_t)$ is elementwise log,
- $[\log \text{softmax}(\ell_t)]_{x_{t+1}}$ indexes the entry for the true token x_{t+1} .

In loss form: $-\text{[log softmax}(\ell_t)\text{]}_{x_{t+1}} = -\log p_t(x_{t+1})$ (the cross-entropy term).

Ok, notations confusing, transformation makes it more confusing, I mean the log term. So eventually each based on each token, we have a prediction, in the format of probability vector, a list of 50,000 numbers representing the probability of choosing each token for the 50,000 possible tokens. So now what? We want to maximize the probability for the true token. Ok this sounds weird. What do you mean by 'maximize'? Then think this way: all those parameters hanging there, once they are set, they will produce one probability vector, giving us the probability for each token. Setting them differently, you get a different probability vector. Theoretically we could try all the combinations for those parameters, then some of them are good, some of them are better, but how to judge them? We know the true token, we are going to look for the probability for that true token through all the answers. *Then the one with a highest value for the true token's probability is the best one.* So among all the possible parameters, we want to choose the set that gives us the maximum probability for the true token. This is called 'maximize the probability over the parameters θ '. But how to do that?

Theoretically we could manually put in parameters, try 'all' of them, and record each result, then sort the result, find out the one giving the highest number. But, there are too many parameters, and too many possible values, it's impossible to do that in brute force manner. So what to do?

Well, we know the relationship between the parameters and the end result. So the relationship between the parameters and the end result tells you how the parameter setting is related to the end result. Calculus can tell you how the end result *changes* with the parameters. (I used to think they are the same thing: knowing the relationship means you know how one changes with the other, until I contemplate over the meaning of differential function for long time! So, placing a condition for 'differential function' is making sure knowing the relationship means knowing the relationship of the changes! Anyway!).

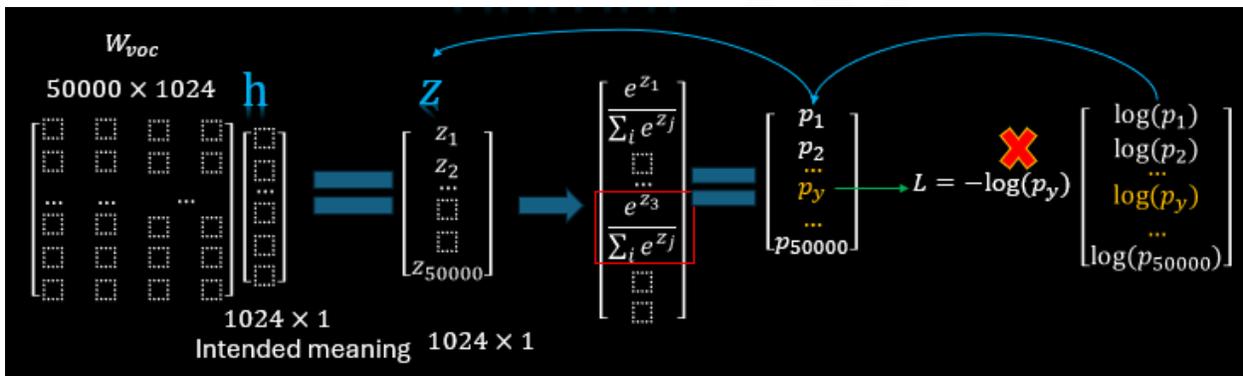
So the punchline is, once you know how the end result changes with the parameters, you don't have to randomly tune your parameters, but instead, tune it in an optimal way! That's the heart of 'steepest descent method'.

Learning0to1

2025-12-05

So my favorite, Calculus. It's a good opportunity to solidify my understanding about Calculus: how it can be useful in 'fast tuning' in the case of machine learning. You work from the end result $\log(p_i)$. Oops, we haven't explained why there is a 'log' here, after all, our calculation gives us the probability p_i , but why we would take log of it? So we need to understand the concept of 'loss function'. p is the end result, loss function is a way to evaluate that end result! Of course the question is, why can't we use p itself? Since we know what value it is good, what is bad: the bigger it is for the true token, the better, and vice versa. Then why not just optimize p itself? Let's save this question for GPT later. So assume we agree that we will evaluate the performance using $\log(p_i)$, but we have to check if it makes sense: the greater the p , the greater the log. So although we don't know the very reason why we value the log performance yet, what has to be true is, the direction has to be in line. Now we have to work backwards, our goal is to minimize the loss function, o sorry forgot to mention that we evaluate $L = -\log(p_i)$, the negative log as the performance, so now the goal if maximizing p becomes minimizing L . Again we don't quite know why yet.

And this loss is connected to all the parameters on the way, but we are going to peel layer by layer. First of all, L is connected to p_i , then p_i is connected to z_i .



We know the relationship each step, so Calculus helps us finding out how L relates to z , step by step, through chain rule:

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial p_i} \cdot \frac{\partial p_i}{\partial z_i}$$

Hmm, it's strange that the result would be case by case? I mean, why the result is different if it is the true token or not? Ok, I see. Notice L is not defined for each probability, but the probability of the true token: $L = -\log(p_y)$, hence rewriting the chain rule: $\frac{\partial L}{\partial z_i} = \frac{\partial L}{\partial p_y} \cdot \frac{\partial p_y}{\partial z_i}$. So, the evaluation is really in terms of the true token. Not in any other tokens! $\frac{\partial L}{\partial p_y}$ is easy, just $-\frac{1}{p_y}$. Now $\frac{\partial p_y}{\partial z_i} = \frac{\partial [e^{z_y} (e^{z_1} + e^{z_2} + \dots + e^{z_n})^{-1}]}{\partial z_i}$.

Why we have to separate into two cases, true token vs non-true token? The reason is, when you differentiate a function with respect to something, you are viewing how the end result changes with that something. So when we differentiate the true probability with respect to a logit z_i , remember the probability is a function of all the logits, including the logits for the true token. The probability function is a product of the logits of the true token (transformed) and the sum of the logits. The product structure makes a difference when differentiating, if you differentiate whether involve the true token or not (omg,

so wordy!!!) In other words, it's a product of two *things*, the sum definitely involve all the logits, but the one in the front might be a constant, if you are differentiating other logits! So when we are

differentiating with respect to other logits, we treat e^{z_y} as a constant, so we get: $\frac{\partial p_y}{\partial z_i} =$

$e^{z_y}(-1)(e^{z_1} + e^{z_2} + \dots + e^{z_n})^{-2} e^{z_i}$, which is: $-\frac{e^{z_y}}{e^{z_1+e^{z_2}+\dots+e^{z_n}}} \cdot \frac{e^{z_i}}{e^{z_1+e^{z_2}+\dots+e^{z_n}}} = -p_y p_i$, combined

with $\frac{\partial L}{\partial p_y} = -\frac{1}{p_y}$, we got $\frac{\partial p_y}{\partial z_i} = p_i$. What if we are differentiating with respect to the true token? Now e^{z_y}

cannot be deemed as constant, product rule of differentiation needs to be used, so $\frac{\partial p_y}{\partial z_y} =$

$\frac{\partial [e^{z_y}(e^{z_1+e^{z_2}+\dots+e^{z_n}})^{-1}]}{\partial z_y} = e^{z_y}(e^{z_1} + e^{z_2} + \dots + e^{z_n})^{-1} + e^{z_y}(-1)(e^{z_1} + e^{z_2} + \dots + e^{z_n})^{-2} e^{z_y}$. Which

is $p_y - p_y^2$. Now combined with $-\frac{1}{p_y}$, we have: Now $\frac{\partial p_y}{\partial z_y} = p_y - 1$. Ok we got the math right. Now let's

think about what is happening. Z are the logits, so we find out how the performance changes with the logits. The goal is to find out how the performance changes with the parameters, matrix W_{voc} in this

case, but to do that, we have to access it through layers, backwards. Now we can continue with chain rule to calculate how the logits change with the parameters I think.