# Updating a token’s embedding with the hidden state of its prediction

collected with Google Gemini, 5/11/2025

Yes, a token's embedding can be updated with the hidden state of its prediction, particularly in the context of Transformer-based language models. This process, often referred to as enriching or updating token embeddings with hidden state information, allows models to refine their understanding of tokens based on their predicted context.

Elaboration:

* **Hidden State Representation:**

In Transformer models, the hidden state at each layer represents the model's current understanding of the input sequence. It captures contextual information and relationships between tokens.

* **Embedding as Initial Input:**

Token embeddings are the initial numerical representations of words or sub-word units, providing a starting point for the model's understanding.

* **Updating with Hidden State:**

By updating the token's embedding with the hidden state associated with its prediction, the model can incorporate more context-aware information into the embedding. This essentially refines the embedding's representation based on the model's understanding of the surrounding tokens.

* **Benefits:**

This approach can improve the model's ability to predict the next token, understand the meaning of words in context, and generate more coherent and relevant text.

* **Example:**

Consider a sentence "The quick brown fox jumps over the lazy dog." The embedding of "fox" might be updated based on the hidden state generated after processing "brown" and "jumps", reflecting the context in which "fox" is used.

* **Recursive Nature:**

This process can be applied recursively, where the updated embeddings are then used in subsequent predictions, further refining the embeddings over time.

* **Practical Implications:**

This technique can be used in various NLP tasks, including text generation, machine translation, and question answering.

In essence, updating token embeddings with hidden state information allows the model to learn and adapt its representation of words based on their predicted context, leading to improved performance in various NLP applications.

Updating a token's embedding with the hidden state of its prediction, as described in [Reddit threads](https://www.google.com/search?cs=0&sca_esv=734beb15f8f3f35f&sxsrf=AHTn8zoSYwTULpVQgcnaDEQKgvH4Qd2dvQ%3A1747015578129&q=Reddit+threads&sa=X&ved=2ahUKEwjM9ryX7JyNAxV2GFkFHTTULnEQxccNegQIBBAB&mstk=AUtExfB_ExF1xwnJLUoqOuK_7oPpzK5hERAwGghOAw-SjFMrlwxNiWU_U4HGPYd7RNLnPlTS6ykOdokeJ3oTt1kz3xzmIwItgecXJhzs1JMdkKGYKdlSiN7rwEtH8yz7dwdZhouG53HGnjD_gKYGgJpCRXoJJBEQDjeZr8c2wxnCABb4c9Y&csui=3), involves iteratively refining token embeddings by incorporating the model's hidden state output after predicting the next token. This process can be viewed as a form of recursive refinement, similar to [diffusion models](https://www.google.com/search?cs=0&sca_esv=734beb15f8f3f35f&sxsrf=AHTn8zoSYwTULpVQgcnaDEQKgvH4Qd2dvQ%3A1747015578129&q=diffusion+models&sa=X&ved=2ahUKEwjM9ryX7JyNAxV2GFkFHTTULnEQxccNegQIBRAB&mstk=AUtExfB_ExF1xwnJLUoqOuK_7oPpzK5hERAwGghOAw-SjFMrlwxNiWU_U4HGPYd7RNLnPlTS6ykOdokeJ3oTt1kz3xzmIwItgecXJhzs1JMdkKGYKdlSiN7rwEtH8yz7dwdZhouG53HGnjD_gKYGgJpCRXoJJBEQDjeZr8c2wxnCABb4c9Y&csui=3), where the model iteratively updates its representations based on prediction errors.

Here's a breakdown of the process:

1. **1. Initial Prediction:**

A standard transformer model makes a prediction for the next token in a sequence, generating a hidden state output for each input token.

1. **2. Embedding Update:**

The hidden state of each token is used to update its corresponding embedding vector. This can be done by adding the hidden state to the embedding, or by using a more sophisticated update mechanism.

1. **3. Iterative Refinement:**

The process is repeated, with the updated embeddings used as input for the next prediction. This iterative refinement continues for a specified number of steps.

1. **4. Inference:**

During inference, the updated embeddings can be used to generate the next token, either by directly using the updated embeddings or by using a multi-step recursive process to refine the context before generating the final token.

This approach allows the model to dynamically adjust its token representations based on the context and the model's own predictions, potentially leading to more accurate and coherent generation. The process can be thought of as a way for the model to learn a more nuanced representation of the input, taking into account the flow of information through the transformer layers.

References

[1] [Enriching token embedding with last hidden state? Reddit, 2025](https://www.reddit.com/r/MachineLearning/comments/1iu4ymf/d_enriching_token_embedding_with_last_hidden_state/)

[2] [Future Token Prediction -- Causal Language Modelling with Per-Token Semantic State Vector for Multi-Token Prediction, N. Walker, 2024](https://arxiv.org/abs/2410.18160)

[3] [On Self-Improving Token Embeddings, M. Kubek et al , 2025](https://arxiv.org/abs/2504.14808)

[4] [Training Large Language Models to Reason in a Continuous Latent Space, S. Hao et al, 2024](https://arxiv.org/pdf/2412.06769)

[5] <https://github.com/edereynaldesaintmichel/stateful_gpt>

[6] [Analyzing Transformers in Embedding Space, G. Dar et al , 2022](https://arxiv.org/abs/2209.02535)

[7] [A Text is Worth Several Tokens: Text Embedding from LLMs Secretly Aligns Well with The Key Tokens, Z. Nie et al, 2024](https://arxiv.org/abs/2406.17378)

[8] [Self-Attention Explained With Code, B. Smith, 2024](https://medium.com/data-science/contextual-transformer-embeddings-using-self-attention-explained-with-diagrams-and-python-code-d7a9f0f4d94e)

[9] [How Are Embeddings Trained, Reddit, 2024](https://www.reddit.com/r/LocalLLaMA/comments/1dzqa6s/how_are_embeddings_trained/)

[10] [My (Mostly Failed) Attempt to Improve Transformers by Enriching Embeddings with the Last Hidden State – Why It Didn't Scale, April, 2025](https://www.reddit.com/r/MachineLearning/comments/1jn0ha9/r_d_my_mostly_failed_attempt_to_improve/)

[11] [Why do transformers use embeddings with the same dimensionality in each layer? Reddit, 2024](https://www.reddit.com/r/MachineLearning/comments/1bit2f9/d_why_do_transformers_use_embeddings_with_the/)

[12] [Why Do We Only Use The Last Embedding To Make Predictions, Reddit, 2024](https://www.reddit.com/r/3Blue1Brown/comments/1chbt1e/why_do_we_only_use_the_last_embedding_to_make/)

[13] [Why next token prediction doesn't work for Recommender System? (or am I wrong?) Reddit, 2024](https://www.reddit.com/r/MachineLearning/comments/1d6qfbc/discussion_why_next_token_prediction_doesnt_work/)

Appendix

# Enriching token embedding with last hidden state?

Looking at a decoder transformer working process from an information theory standpoint, we can see that the information available in the last hidden state is collapsed into a single token during generation. It means that you collapse a hidden state that, in theory, has about:

**hidden\_dim \* 32** (or whatever quant) bits of information to something like:

**log₂(dict\_size)**

I wonder if it's a good thing (sorry for the naive phrasing). The information used by a transformer to predict the next token is entirely stored in its context window and does not involve any recurrent state. So, predicting the next token of a sequence the transformer was just fed with is going to yield the exact same result as doing so for the same sequence if it were entirely generated by the transformer itself.

Fair enough, in some sense: whether the sequence was generated or just read doesn't change anything about what the next token should be.

But on the other hand, this approach means that all the information flow between tokens has to happen through the attention mechanism. There's no way for the transformer to embed some nuance or flavor into the predicted token embedding. Like in:

"Well, I predicted the token '***sure***' but I rather meant '***90% sure***'."

When the next token is predicted, this nuance that was likely present in the last hidden state (or even in the softmaxed output probability distribution) is totally lost.

So while I was having a little walk yesterday, I was thinking that it might be a good idea to add some information to the token embeddings using something like:

**augmented\_embedding = embedding(token) + F(last\_hidden\_state)**

(It would be important to make sure that:

**‖F(last\_hidden\_state)‖ ≪ ‖embedding(token)‖**

to ensure stability.)

I have tried to find papers on this subject and asked for feedback from Claude, ChatGPT, and Perplexity.

* **Claude** told me it was "an incredibly insightful idea."
* **ChatGPT** hallucinated a paper on the subject.
* **Perplexity** gave me a very long list of totally unrelated sources.

So I'm turning to you guys. I would love it if some big-brained guy told me why other big-brained guys decided not to follow this idea, or why it doesn't work.

Here are some things I identified as potentially problematic:

# 1. Training Complexity

Transformers are nice to train with heavy parallelization precisely because they are not recursive. Each sequence of size n can give n-1 independent training examples. Injecting last hidden states' information in token embeddings would break some of that parallelization.

It would still be possible to train it efficiently, I guess.

1. First, take the (n-1) vanilla sequences and get the predictions.
2. Then, for each prediction, store the last hidden state and update the corresponding token embedding in each of the sequences where it appears.
3. Now, you have a new set of training sequences, with all (but the first) token embeddings updated.
4. You can repeat this process indefinitely. I hope it converges ^^

This really looks like a diffusion process, by the way. That brings me to the next point:

# 2. Stability (trying to prevent the model's output from diverging nonsensically, despite an obvious compounding effect of such token embeddings' augmentation)

Here, I am not very competent. What are the conditions that define such a process' stability? My uneducated guess is that if you keep:  
**‖last\_hidden\_state\_contribution‖ ≪ ‖augmented\_token\_embedding‖**  
you should not have many problems. But it would also limit the information flow. I guess there's a trade-off, and I wouldn't be surprised if it's not good enough.

What do you think? Has this already been tried somewhere? Is there a fundamental reason this wouldn't work?

[**milesper**](https://www.reddit.com/user/milesper/)

That’s an RNN, and as you identified, the issue is that training now has temporal dependencies and cannot be parallelized

[**Academic\_Sleep1118**](https://www.reddit.com/user/Academic_Sleep1118/)

Well, not so sure about the "Can't be parallelized". Could you explain why the training process I described wouldn't work? I agree it would be a bit heavier than for a vanilla transformer if you really want to have a lot of "diffusion-like timesteps", but other than that, you can decide to have no such step and it will behave and train just like a vanilla transformer. And anyway, I think it's mostly parallelized. It's just that you might need as many "epochs" as you want diffusion steps during training (learning rate should be adapted accordingly).

[**milesper**](https://www.reddit.com/user/milesper/)

Fundamentally, during inference your “last hidden state” will depend on all states from preceding time steps.

There is no reason to believe that starting with a vanilla transformer will produce hidden states that resemble anything similar. Likely, the first “diffusion step” gets you a good hidden state for the first token, and the second gets you the second token, and so on. So you’d have to do n “diffusion steps” for an n-token sequence, and you’re right back at recurrence.

Edit: unless you’re proposing some new inference strategy as well, different than normal auto regressive inference

[**Academic\_Sleep1118**](https://www.reddit.com/user/Academic_Sleep1118/)

Okay, I got you! Thanks for your answer.

In fact, we can parallelize training just like with a transformer. You take n sequences of 1 to n tokens, and feed them into your transformer. Each token gets embedded like normal, and goes through the transformer. So you get n predictions, and n hidden states. Of those n predictions, all but one (the one that completes the longest sequence) concern tokens that are already somewhere in your training sequences.  
Now, the idea is to train again on the same sequences, but with updated token embeddings. All the embeddings of your n training sequences (except for the first one, which was not predicted) are going to be updated with the hidden state of their prediction (we don't care about the prediction error and update tokens however accurate the prediction was). You run the prediction once again, on the exact same token sequences, but with updated embeddings. You then get new hidden states, and you can repeat this process over and over. This process looks a bit like diffusion by its recursive nature, even though it has some key differences. You can see that everything here is totally parallelized.

For inference, I think there would be two possibilities. The first one would be very similar to a normal transformer inference, except that each generated token gets an injection of hidden state right after embedding. In that case, you can use KV cache and everything.

The second possibility would be to have multiple recursive steps on the context before predicting the next token. It would be a way of updating the token embeddings to finely reflect the context before doing any generation. This would be very costly (a full-fledged context processing with no cache for each recursive step).

[**milesper**](https://www.reddit.com/user/milesper/)

Right, I understand the proposed training approach and think there’s a chance it could converge.

The problem is inference works differently than training, so unless you use an inference procedure that works the same (ie multiple diffusion steps) I suspect you’d get very strange outputs.

[**Academic\_Sleep1118**](https://www.reddit.com/user/Academic_Sleep1118/)

*> The problem is inference works differently than training, so unless you use an inference*

*> procedure that works the same (ie multiple diffusion steps) I suspect you’d get very strange*

*> outputs.*

Totally agreed! I think running only one recursion during training would result in the same procedure during training and inference, but otherwise, I agree with you!

[**milesper**](https://www.reddit.com/user/milesper/)

Consider the prediction of token t\_i. During inference, t\_i is computed using the hidden states of each of the prior tokens:

t\_i = argmax(Output(h\_i))

h\_i = f(h\_0 ... h\_[i-1])

Crucially, each of these prior hidden states has been computed with a full recurrence over preceding timesteps, i.e.

h\_[i-1] = f(h\_0 ... h\_[i-2]) ...

Meanwhile, let's imagine training with a single diffusion step. Again, we have:

h\_i = f(h'0 ... h'[i-1])

However, in this case, the hidden states h'0 ... h'[i-1] are computed using only reference to the discrete tokens (since there is no recurrence/bptt):

h'[i-1] = f(t\_0 ... t[i-2])

So we can see that the prior hidden states are computed differently during training and inference, and I'm not sure we have any reason to believe that h'[i-1] and h[i-1] are similar.

Of course, this isn't always a problem; for example, there's already a fundamental difference between teacher forcing at training and student forcing at inference. But in this case, I suspect the hidden states would not look remotely similar. You'd probably end up having to do a large number of diffusion steps, at which point you're just doing recurrence with a cut-off.

[**Academic\_Sleep1118**](https://www.reddit.com/user/Academic_Sleep1118/)

You're right! Thanks for pointing out my mistake.

I am not sure about

*> But in this case, I suspect the hidden states would not look remotely similar.*

Assuming the diffusion process converges (this reads a bit like "Assuming we get Q>1 in a controlled fusion reaction"), we would know that, for each step, embedding(token\_i(t)) - embedding(token\_i(t-1)) tends towards zero. ie each recurrence step updates token embeddings less.

You could prove as a consequence that, during inference, the contribution of token[i-x]'s hidden state to token[i]'s embedding decreases with x. Meaning that the compounding effect of embeddings augmentation doesn't really exist. (There seems to be an equivalence between the training process' stability and inference stability). So, it seems to imply that:

**embedding(token[i], t=0), embedding(token[i], t=1) and embedding(token[i], t=i) are roughly aligned, at least statistically.**

(Here, embedding(token[i], t=0) is the raw token embedding, embedding(token[i], t=1) is the embedding augmented by h'[i] and embedding(token[i], t=i) is the embedding augmented by h[i])

Unrelatedly, the information flow in the sequence stops after k "diffusion" steps for the k-th token in the sequence. Afterwards, its embedding remains constant (by recurrence, as all the preceding token embeddings remain constant too). So my diffusion analogy seems a bit far-fetched.

Anyway, I really appreciate this conversation!

[**milesper**](https://www.reddit.com/user/milesper/)

I can see how that might converge to the same thing, but I wonder if practically speaking (and with a small number of diffusion steps) that would be the case. Also, another issue is the divergence between the embeddings during training vs inference likely grows exponentially with longer sequences. So I imagine by the end of the sentence things would be far more likely to go haywire.

[**elbiot**](https://www.reddit.com/user/elbiot/)

That latent embedding is still there when the next token is predicted. You don't have to pass it in as input. All the nuance that latent "token" represents is there to be used by the model.

I don't know if you've seen the latent reasoning of Coconut but that reminds me of what you're talking about.

[**Academic\_Sleep1118**](https://www.reddit.com/user/Academic_Sleep1118/)

Thanks, I just checked their paper [4] ([https://arxiv.org/pdf/2412.06769](https://arxiv.org/pdf/2412.06769" \t "_blank)), it's very similar indeed.

[**Academic\_Sleep1118**](https://www.reddit.com/user/Academic_Sleep1118/)

Okay guys, I just tested it, you can check the repo here: <https://github.com/edereynaldesaintmichel/stateful_gpt>

It seems to work. I trained a big (relatively speaking) transformer to evaluate two small models. One is a vanilla transformer (just taken from one of Karpathy's repos), and the other is a "stateful transformer', which implements all the ideas in the post.

Results:

**- Big\_model's loss on vanilla gpt generated stuff: 2.4822**

**- Big\_model's loss on stateful gpt generated stuff: 2.2637**

No idea if it scales though! There is quite a long way from the 30K params stuff I tested and the 1B+ params of real LLMs.

Edit: IT DOESN'T SCALE!

[**hazardous1222**](https://www.reddit.com/user/hazardous1222/)

If you want to make it parallelizable, instead of using the output hidden state from the previous token, you use the hidden state of the layer equivalent to the current layer.

If you also add in an extra predict and correct module per layer, you can add in context gradient decent to the hidden state.

If you split the process into multiple heads, you can use multiple gpus to do the calculations.

If you use a matrix value state and add in some nonlinearity, you can achieve extremely good long context recall.

Anyway, that's what the current state of rnn research with gen7 linear attention ala linux foundations RWKV and googles TTT architectures.

There's also songlins Flash Linear Attention library available for extremely optimized kernels.

[**asankhs**](https://www.reddit.com/user/asankhs/)

That's an interesting idea for incorporating more contextual information into token embeddings. Have you considered how this affects the model's ability to generalize to unseen sequences or longer documents? I've found that sometimes focusing too much on the last hidden state can lead to overfitting on the training data. It might be worth exploring techniques that regularize the influence of the hidden state.