Hierarchical Character-level Language Model Re-implementation

Zhaokun Wang
Institute of Computational Linguistics
Heidelberg University, Heidelberg, Germany
zhaokun.wang@stud.uni-heidelberg.de

Abstract

The Hierarchical Character-level Language Model with a Continuous Cache (HCLM+Cache) addresses open-vocabulary modeling by combining character-level generation with a memory of recent words. This paper describes a re-implementation of the model in PyTorch and an analysis of its behavior. Our initial implementation showed two limitations: sequential processing that restricted computational parallelism and cache resets that interrupted long-range reuse. By introducing vectorized computation and continuous cache management, we obtained improvements in both efficiency and predictive performance on the WikiText-2 benchmark. Building on the optimized implementation, we conducted an ablation study to assess the contributions of the hierarchical structure and the continuous cache. The results indicate that the cache plays a central role. We provide these findings as a reference for future work on reproducing and extending complex stateful language models.[1]

1 Introduction

Natural language is marked by an ever-growing vocabulary and the "bursty" reuse of rare words, where the first mention of a token increases its likelihood of reappearing (Heaps, 1978; Church and Gale, 1995; Church, 2000). Conventional word-level language models, which operate with a fixed vocabulary, sidestep this challenge by mapping unseen items to an <unk> token (Mikolov et al., 2010). While effective in some applications, such models cannot generate new strings and fail to capture burstiness.

Character-level models (Sutskever et al., 2011; Graves, 2013) solve the open-vocabulary problem by composing words from characters. However, they lack an explicit mechanism for reusing previously generated words. Subword segmentation (Sennrich et al., 2016) and compositional encoders (Ling et al., 2015; Kim et al., 2016) partially alleviate this, but none directly address short-term repetition. Neural caches (Grave et al., 2017; Merity et al., 2017) provide such a mechanism for closed vocabularies by biasing predictions toward recently seen tokens.

To combine these strengths, Kawakami et al. (2017) introduced the Hierarchical Character-level Language Model with a Continuous Cache (HCLM+Cache). This model generates novel words at the character level while dynamically copying previously produced words through a pointer-based cache, yielding consistent gains across English and multilingual benchmarks.

Motivated by this design, we re-implemented the model, but our initial attempt yielded suboptimal results. This paper presents our work in two parts. First, as a case study, we document the critical performance bottlenecks our implementation faced and show how resolving them leads to a functional, efficient model. Second, using this optimized model, we conduct an ablation study to dissect the contributions of the hierarchical structure versus the continuous cache. Our goal is to move beyond simple reproduction to a deeper understanding of *why* the model works, offering both practical engineering insights and a clearer scientific analysis of its components.

2 Model Architecture and Implementation Logic

This section provides a detailed breakdown of our PyTorch implementation, following the conceptual design in (Kawakami et al., 2017). To orient the reader, we begin with a simplified pipeline diagram (Figure 1) that captures the overall flow, and then explain each component in turn. Detailed architec-

[1]Codes are publicly available at https://github.com/BufferHund/HierarchicalCharLM_Reimplementation

ture diagrams are provided in the Appendix.

2.1 Simplified Pipeline Overview

Figure 1 illustrates the main idea: words are represented through character-level composition, contextualized in a word-level sequence model, and then either generated afresh or retrieved from memory through a cache mechanism. This high-level view shows how the two pathways—generation and reuse—are integrated through a gating function.

2.2 Overall Architecture

The model combines a hierarchical language model (HCLM) with a pointer cache. The HCLM accounts for character-to-word generation, while the cache provides a distribution over recently seen words. At each time step, the two are interpolated using a dynamic gate λ_t, as expressed in Equation 1:

$$p(w_t|w_{<t}) = \lambda_t \, p_{\text{lm}}(w_t|w_{<t}) + (1-\lambda_t) \, p_{\text{ptr}}(w_t|w_{<t}) \tag{1}$$

The result is a system that can flexibly switch between inventing new forms and reusing known ones. A full architectural diagram is given in the Appendix (Figure 3).

2.3 Hierarchical Language Model (HCLM)

The HCLM spans two levels of granularity: characters and words. At the lower level, a character encoder composes embeddings into a word representation that generalizes beyond the training vocabulary. At the higher level, a word-level LSTM integrates these embeddings into a contextual state sequence, $\mathbf{h}_t^{\text{ctx}}$, which serves as the main driver for prediction. Finally, a character decoder reconstructs the next word given this context. The diagrams of these components are provided in the Appendix (Figures 4–6).

2.4 Continuous Cache

Running in parallel with the HCLM is a memory mechanism. The cache stores pairs of context vectors and their corresponding words, updating keys whenever a word is reused. When predicting the next token, the current context attends over stored keys, yielding a probability distribution over words in memory. This equips the model with a direct copy pathway, particularly effective for repeated tokens and discourse-specific names. The cache mechanism is illustrated in Appendix Figure 7.

2.5 Gating Mechanism

The two pathways are combined through a small feed-forward network that computes λ_t from $\mathbf{h}_t^{\text{ctx}}$. This gate dynamically balances novelty and memory, unifying the HCLM and cache into a single probability distribution. An example of this integration can be seen in the overall architecture diagram in the Appendix (Figure 3).

3 Experimental Analysis of the Initial Implementation

3.1 Experimental Setup

The first experiment (Run 1) was designed as a baseline to verify the logical correctness of our reimplementation. To enable rapid prototyping and debugging, we employed a compact configuration on the `wikitext/wikitext-2-raw-v1` dataset:

- **Batch Size:** 64
- **Word Context Length (S):** 10
- **Hidden Dimensions (H_w, H_c):** 256
- **Cache Size:** 800
- **Epochs:** 5

This setting prioritized implementation validation over modeling capacity, and thus served as a controlled environment to assess whether the architecture could begin to capture coherent linguistic patterns.

3.2 Quantitative and Qualitative Results

After five epochs, the baseline model achieved a best validation Bits-Per-Character (BPC) of **2.0721**. Although training loss decreased steadily, the validation curve plateaued early, suggesting that the model was unable to generalize beyond local token statistics.

A closer look at the generated sequences confirms this limitation. For instance, at Epoch 3 the model produced the following continuation given the seed prompt *“what is the difference between”* (Appendix A):

> **Generated Sample (Run 1, Epoch 3):**
> “by the . causes front time and ’s owl from Maryan and read Series inHg a also the to . the 13 Jupiter , northeast 4 wines within has”

Similarly, by Epoch 5, the outputs remained incoherent, filled with malformed tokens and erratic topic shifts:

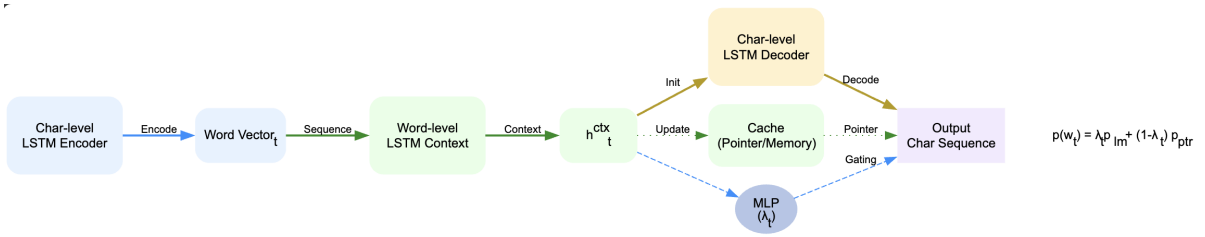


Figure 1: Simplified pipeline overview: from character input to the final mixed probability distribution.

Generated Sample (Run 1, Epoch 5): "83 up Jordan hundred up , Sport bridge triple Way about assassinat . renewed in has line the Japanese excess holiday 1999 many the rans study of being to known"

These examples illustrate three recurrent failure modes: (i) grammatical breakdown, (ii) semantic drift with inconsistent topics, and (iii) spurious named-entity fragments (e.g., "Jupiter", "Japanese excess holiday 1999"). Together with the quantitative stagnation, these samples demonstrate that the baseline model struggled to acquire meaningful long-range linguistic structure.

3.3 Identification of Performance Bottlenecks

The poor results can be traced to two fundamental flaws in the initial implementation.

Computational Inefficiency. The forward pass was realized with nested Python `for` loops, processing each token in the batch sequentially (Algorithm 1). This design prevented effective use of GPU parallelism, resulting in very low throughput (approx. 3,100 tokens/sec). Consequently, the model could not be trained on sufficient data to reach convergence.

Discontinuous Cache State. The cache was reset at the beginning of every training batch. This design choice interrupted the continuity of the cache memory and hindered the model's ability to reuse words across segment boundaries. As a result, the cache component failed to contribute, and the model was forced to rely almost exclusively on the generative decoder, yielding a higher BPC.

In summary, the baseline experiment verified that the re-implementation ran correctly but revealed critical computational and state-management limitations that prevented the model from learning coherent linguistic structure. These findings motivated the optimizations described in the next section.

Algorithm 1 Word-by-Word Forward Pass Logic in Initial Implementation (Run 1)

Input: Batch of word strings 'B' of shape '[B, L]'
'cache.reset()' ▷ **Flaw 2: State discontinuity**
'total$_n ll$'$\leftarrow 0$
for t from 0 to $L-1$ **do**
  for b from 0 to $B-1$ **do** ▷ **Flaw 1: Sequential processing**
    $\mathbf{h}_{b,t} \leftarrow$ `Context`[b, t]
    $\log p_{\text{lm}} \leftarrow$ `Decoder`$(\mathbf{h}_{b,t}, w_{b,t})$
    $\log p_{\text{ptr}} \leftarrow$ `Cache`$(\mathbf{h}_{b,t}, w_{b,t})$
    $\lambda \leftarrow$ `Gate`$(\mathbf{h}_{b,t})$
    'total$_n ll$'$+= MixtureLoss(...)$
      'cache.write'$(\mathbf{h}_{b,t}, w_{b,t})$
return 'total$_n ll$'

4 Iterative Optimization and Comparative Experiments

4.1 Implemented Improvements

The diagnosis of Run 1 highlighted two implementation-level flaws: inefficient sequential computation and discontinuous cache management. In Run 2, we introduced targeted refinements to address these bottlenecks.

Vectorized Computation. The forward pass was refactored to eliminate explicit Python loops. By reshaping the input tensor from (B, S, L_c) to $(B \times S, L_c)$, we enabled fully parallelized batch operations within the character encoder, substantially improving hardware utilization.

Continuous Cache State. We redesigned cache management so that the cache persists across batches within each epoch, stored directly in GPU memory. This eliminated redundant resets and al-

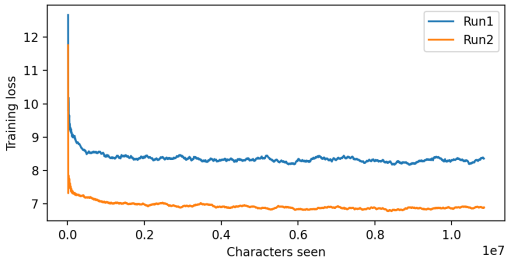
lowed the model to exploit longer-range dependencies across segment boundaries.

4.2 Comparative Results and Analysis

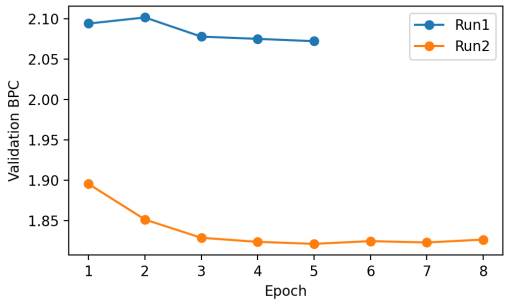
The optimized model demonstrated clear quantitative and qualitative improvements. As summarized in Table 1, Run 2 achieved a best validation BPC of **1.8267**, representing an 11.8% relative reduction compared to Run 1. Training throughput increased from ∼3.1k to ∼11.7k tokens/sec, a ≈3.8× speedup. Figure 2 further illustrates the faster convergence and lower validation BPC.

Table 1: Performance comparison between baseline (Run 1) and optimized (Run 2).

Metric	Run 1	Run 2	Improvement
Best Val BPC	2.0721	1.8267	11.8%
char-PPL	4.20	3.54	15.7%
Throughput	≈3.1k	≈11.7k	≈3.8×



(a) Training loss per epoch.



(b) Validation BPC per epoch (lower is better).

Figure 2: Comparison of training loss and validation BPC for the initial (Run 1) and optimized (Run 2) experiments. The optimized run shows significantly faster convergence to a much better result.

The qualitative difference is even more striking. Whereas Run 1 never produced coherent continuations (see Appendix A), Run 2 began to form recognizable syntactic and semantic structures as early as Epoch 2. For example (Appendix A, Run 2 Epoch 2):

> **Generated Sample:** "arthed but , from , in is and more parties . Wefer , dydes , about making , as and cemett as , with Fine Manager in , in"

Although still partially incoherent, this output already shows more consistent word boundaries and named entities than the baseline. By peak performance, Run 2 produced fluent and contextually appropriate continuations:

> **Generated Sample (Run 2 peak):** "what is the difference between the two films , but it is not a direct sequel . It is a remake of the 1984 film of the same name . The film was released on"

This progression confirms that the optimized implementation not only accelerated training but also unlocked the model's ability to capture higher-level linguistic patterns.

4.3 Summary

Together, these results demonstrate that correcting implementation inefficiencies can yield substantial improvements in both speed and quality. The optimized Run 2 provides a solid foundation for further investigation, motivating the ablation study in the following section.

5 Ablation Study

5.1 Experimental Setup

Having established the optimized implementation (Run 2), we ablated the two key components on the WikiText-2 validation set:

- **HCLM + Cache**: full optimized model.
- **No-Cache**: cache and gating disabled ($\lambda_t \equiv 1$).
- **No-Hierarchy**: non-hierarchical character-level LSTM baseline.

5.2 Results and Analysis

Table 2 shows that the cache is pivotal. Removing it degrades best-val BPC from **2.0721** to **2.2780** (∼9.9% relative), indicating that explicit reuse of recently seen words drives most of the accuracy gains. Notably, **No-Cache** performs on par with **No-Hierarchy** (2.2780 vs. 2.2741), implying that the hierarchical word–character structure contributes little in isolation without a reuse mechanism. This pattern aligns with prior observations on cache-augmented open-vocabulary LMs,

where improvements are primarily attributed to modeling bursty reuse rather than hierarchy per se (Kawakami et al., 2017).

From an efficiency standpoint, **No-Cache** and **No-Hierarchy** are $\sim$2.4–2.7$\times$ faster than the full model, reflecting the overhead of cache operations. Thus, the cache introduces a clear accuracy–throughput trade-off.

Table 2: Ablation study results. Throughput is in tokens/sec.

Model Configuration	Best Val BPC	char-PPL	Throughput
HCLM + Cache	**2.0721**	**4.20**	$\approx$5.0k
No-Cache	2.2780	4.85	$\approx$12.2k
No-Hierarchy	2.2741	4.83	$\approx$13.5k

5.3 Summary

The ablation indicates that gains in our setting are dominated by the cache's ability to model short-term lexical reuse, while hierarchy alone offers negligible benefit. Future work should therefore prioritize *more efficient cache designs* (to reduce overhead) before deepening hierarchical structure.

6 Discussion

Our experiments show that the gap between the initial and optimized implementations stemmed not from the model's architecture, but from overlooked implementation details. Sequential computation limited data throughput, while resetting the cache at each batch prevented learning long-range dependencies. Once corrected, the same architecture achieved both faster training and substantially better generalization. The ablation study further confirmed that the cache mechanism, rather than hierarchical modeling alone, was the dominant source of improvement.

These results highlight a broader point: in modern neural models, "implementation choices" are inseparable from "model design." Efficient use of parallel hardware and careful state management are prerequisites for architectures to function as intended. For reproducibility studies, this case illustrates how small implementation decisions can decisively affect the conclusions drawn about a model's effectiveness.

7 Conclusion and Future Work

We presented a systematic reproduction and analysis of the HCLM+Cache model. Our study revealed that the poor performance of the initial implementation was due not to the architecture itself, but to two overlooked design choices: sequential computation and discontinuous cache state. Correcting these bottlenecks through vectorization and continuous cache management led to substantial improvements in both efficiency and accuracy, reducing validation BPC by 11.8%. An ablation study further showed that the cache mechanism is the dominant contributor to the model's gains, while hierarchical structure alone provides little benefit.

This work highlights that implementation fidelity is inseparable from model design, and that small coding choices can profoundly affect reproducibility.

Future Work. Building on these insights, future directions include:

- Extending state continuity beyond epochs, enabling document-level modeling.
- Exploring more efficient cache mechanisms to reduce overhead while preserving accuracy.
- Applying advanced regularization and optimization strategies to further improve generalization.

References

Kenneth W. Church. 2000. Empirical estimates of adaptation: the chance of two noriegas is closer to $p/2$ than p^2. In *Proceedings of COLING*, pages 180–186.

Kenneth W. Church and William A. Gale. 1995. Poisson mixtures. *Natural Language Engineering*, 1(2):163–190.

Edouard Grave, Armand Joulin, and Nicolas Usunier. 2017. Improving neural language models with a continuous cache. In *Proceedings of ICLR*.

Alex Graves. 2013. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

Harold S. Heaps. 1978. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press.

Kazuya Kawakami, Chris Dyer, and Phil Blunsom. 2017. Learning to create and reuse words in open-vocabulary neural language modeling. In *Proceedings of ACL*, pages 1492–1502.

Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. 2016. Character-aware neural language models. In *Proceedings of AAAI*, pages 2741–2749.

Wang Ling, Tiago Luis, Luis Marujo, Ramon Fernandez Astudillo, Silvio Amir, Chris Dyer, Alan W. Black, and Isabel Trancoso. 2015. Finding function in form: Compositional character models for open vocabulary word representation. In *Proceedings of EMNLP*, pages 1520–1530.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017. Pointer sentinel mixture models. In *Proceedings of ICLR*.

Tomas Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Proc. Interspeech*, pages 1045–1048.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of ACL*, pages 1715–1725.

Ilya Sutskever, James Martens, and Geoffrey Hinton. 2011. Generating text with recurrent neural networks. In *Proceedings of ICML*, pages 1017–1024.

Appendix

A Reproducibility Checklist

- **Code/Logs/Figures:** raw logs and parsed CSVs; figures in this paper (`train_loss_compare.png`, `val_bpc_compare.png`, `val_char_ppl_compare.png`).
- **Data:** wikitext/wikitext-2-raw-v1 from HuggingFace.
- **Configs:** Run-A (smaller model/horizon), Run-B (larger model, longer horizon); details in logs.
- **Hardware:** NVIDIA P100 GPU from Kaggle; eager+AMP (no `torch.compile`).
- **Randomness:** Fixed seeds per run; multi-seed aggregation planned.

B Use of AI-Based Tools

This appendix documents the use of artificial intelligence (AI)-based tools in the preparation of this academic work.

List of Steps Involving AI-Based Tools

- **DeepSeek**: I consulted DeepSeek models to learn more formal organization of the conclusion and appendix chapter. The suggested frameworks were adapted and rewritten entirely in my own words. I also referred to DeepSeek during debugging to understand and resolve specific error messages.
- **QuillBot**: QuillBot was used sparingly to rephrase sentences for improved readability and flow. All suggestions were manually reviewed and edited to ensure alignment with my original intent and academic style.
- **DeepL and Youdao Translation**: DeepL and Youdao Translation assisted in translating a small number of technical terms and short phrases from Chinese to English to clarify meaning during drafting. These translations were verified and incorporated into my own text.

C Supplementary Figures

For completeness, we provide additional diagrams of the HCLM submodules and the cache mechanism that were omitted from the main text for brevity.

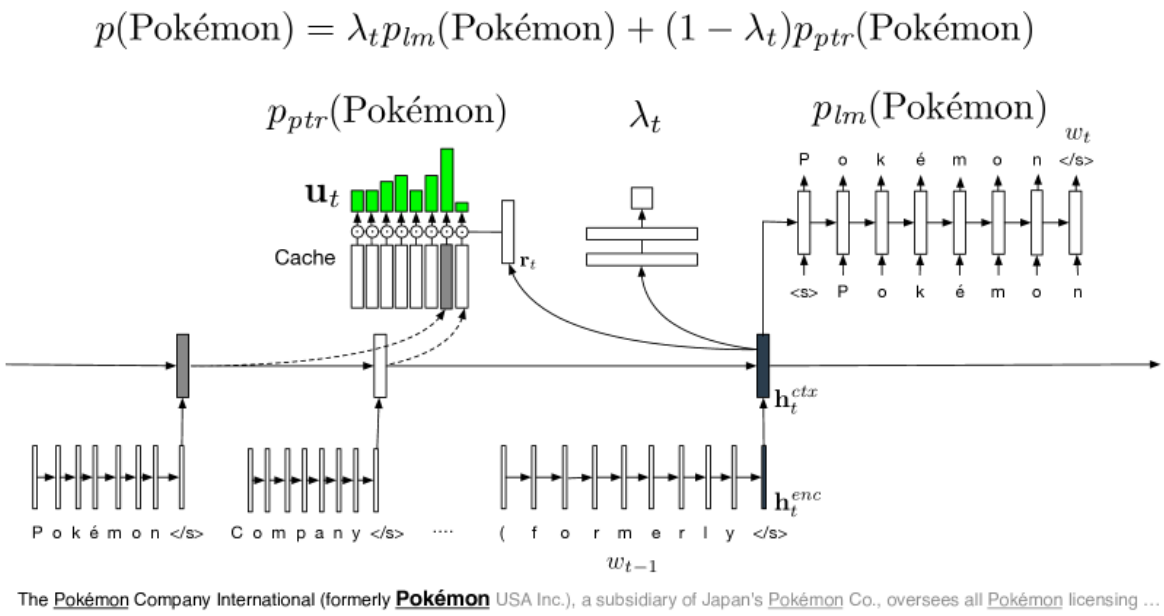


Figure 3: Full architecture of the HCLM with cache.

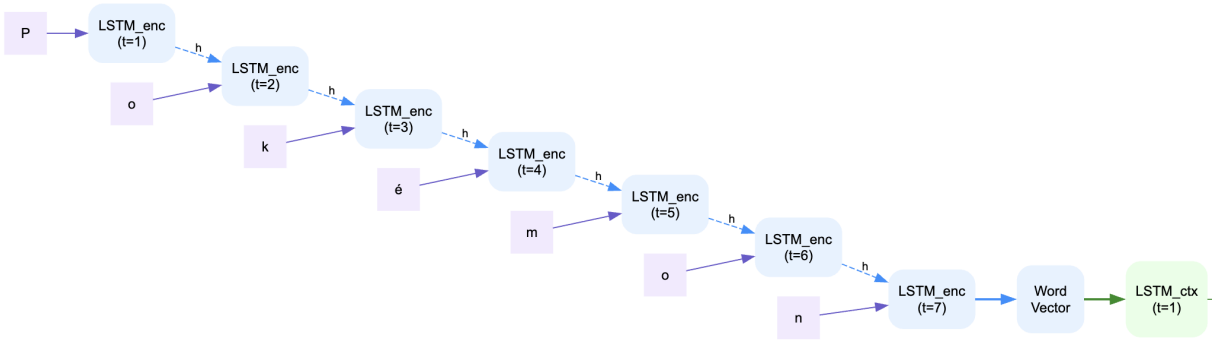


Figure 4: Character encoder module.

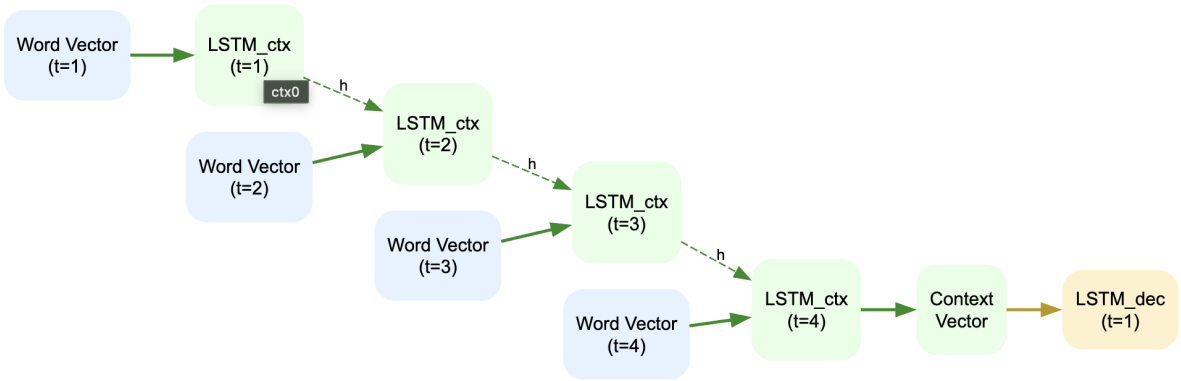


Figure 5: Word-level context encoder module.

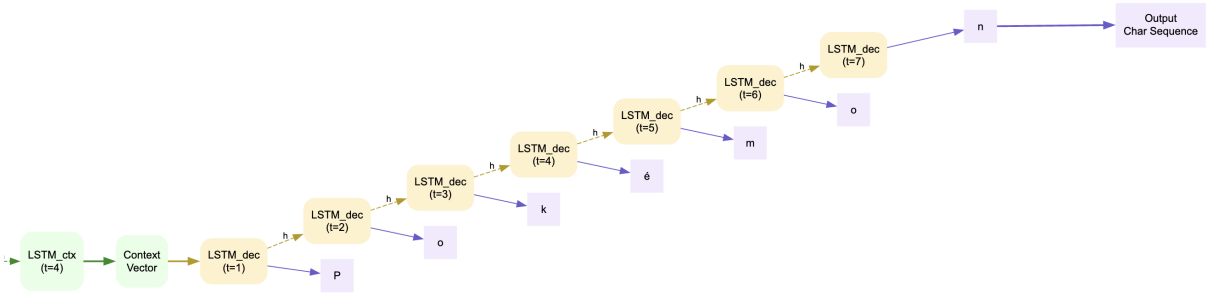


Figure 6: Character decoder module.

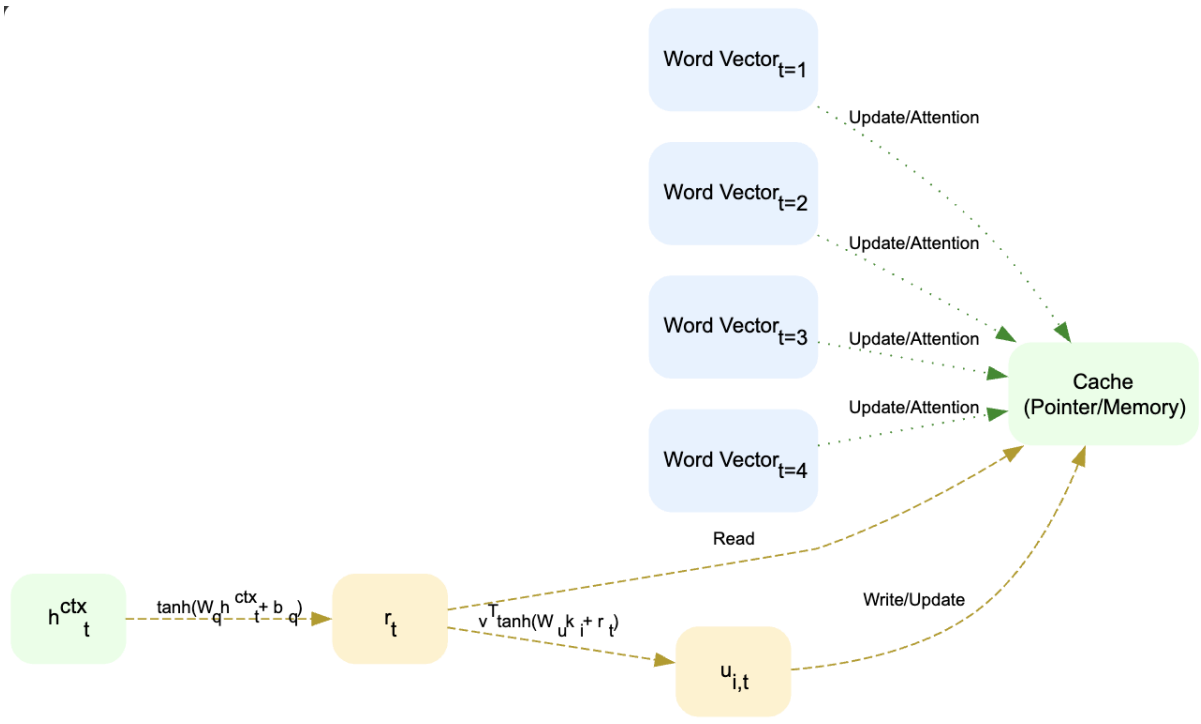


Figure 7: Pointer cache mechanism.