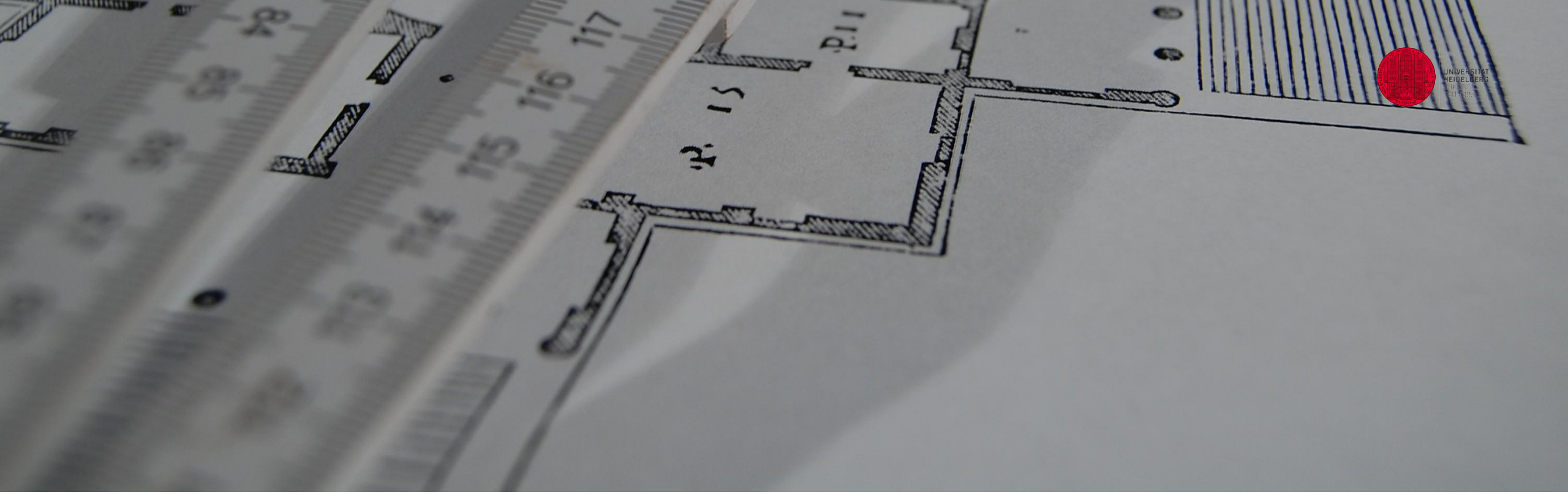


Learning to Create and Reuse Words in Open-Vocabulary Neural Language Modeling

Presenter: Zhaokun Wang | May 21, 2025

Content

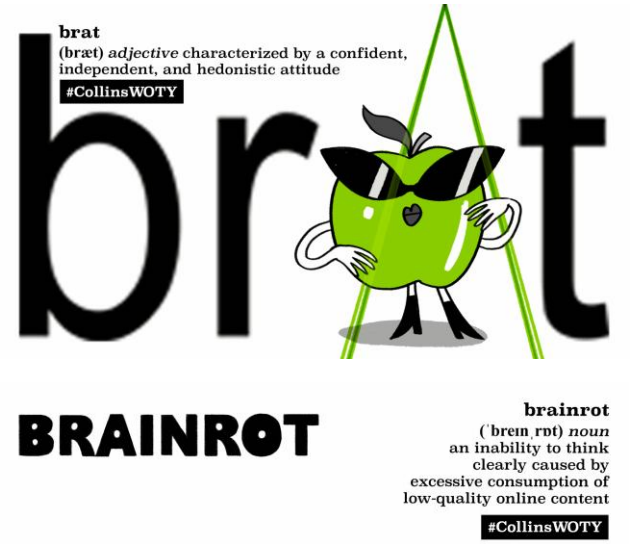
The Dynamic World of Words

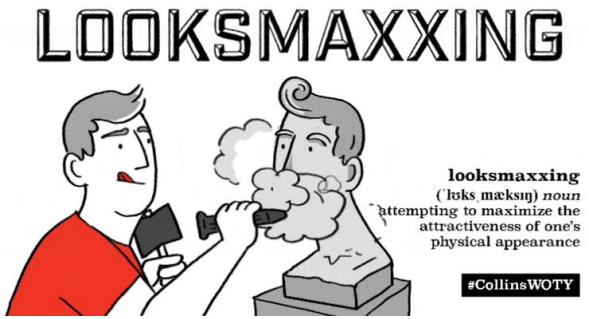
What is the "brat summer"?

Figure 1: Graffiti on a wall in Crema, Italy

The Dynamic World of Words

Collins Dictionary's 2024 word of the year





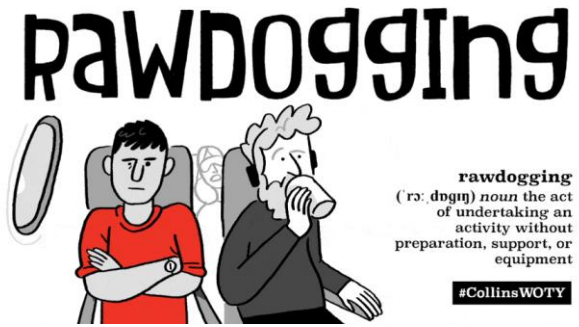


Figure 2: Collins Dictionary's 2024 word of the year

The Dynamic World of Words

- **The Dynamic World:** New words, slang, names – language evolves daily! (Think social media, tech).
- **The Machine Challenge:** Natural language is dynamic. How do AI models keep up?
- **The Old Problem (Pre-2017):**
  - Many models: Fixed vocabulary.
  - New/rare word? → Placeholder: `<UNK>` (unknown).
  - Model effectively says: I don't know this word!

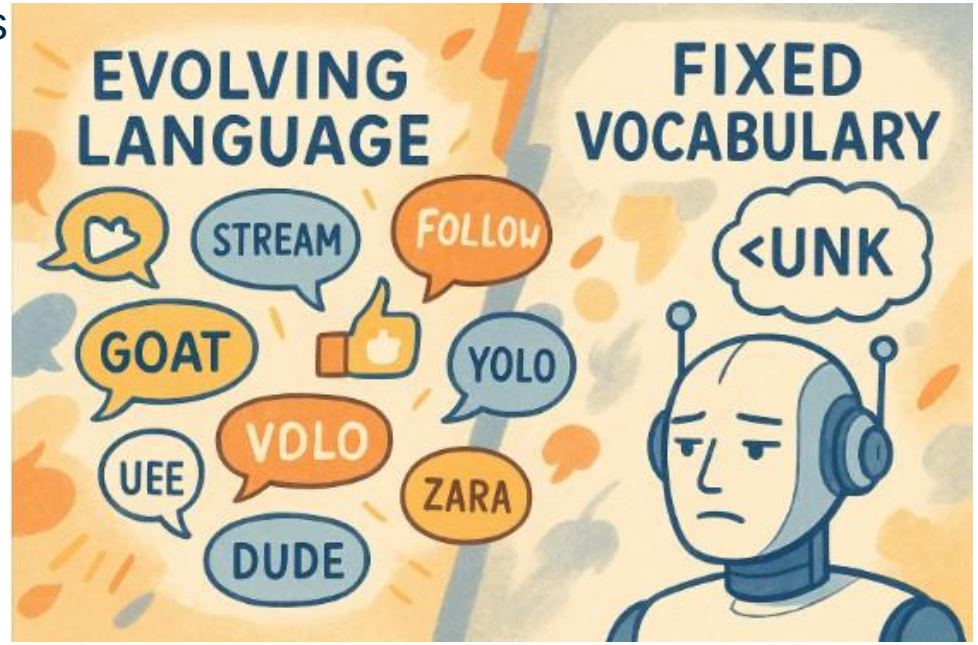


Figure 3: The <UNK> token issue

The Problem with <UNK>

- **Lost Information:**
  - "Gene editing with CRISPR"→ "...with <UNK>."Key details vanish!
  - Fantasy names ("Daenerys,Hogwarts") become <UNK> → Story breaks.
- **Impacted Applications:**
  - Machine Translation: How to translate a new term if it's <UNK>?
  - Speech Recognition: What about new slang or brand names?
  - Autocorrection: Can't suggest fixes for words it's never seen.

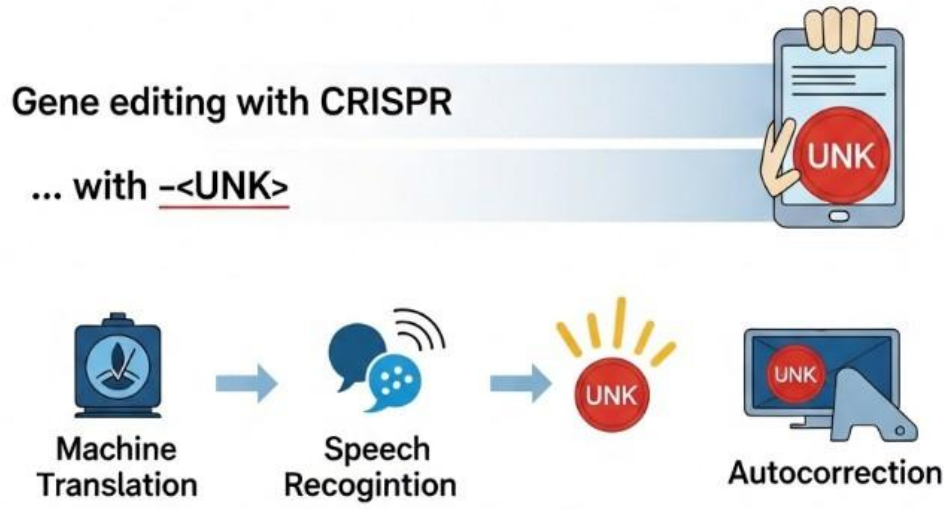


Figure 4: The problem with <UNK> token.

A New Direction: Thinking in Characters!

- **The Big Idea (Mid-2010s):** What if LMs learned from character sequences, not just whole words?
- **Why Exciting?**
  - **Open Vocabulary:** Can form any word, even unseen ones, by learning spelling rules (orthography).
  - **Bye-Bye `<UNK>` (Mostly):** Naturally handles new words by building them character-by-character.

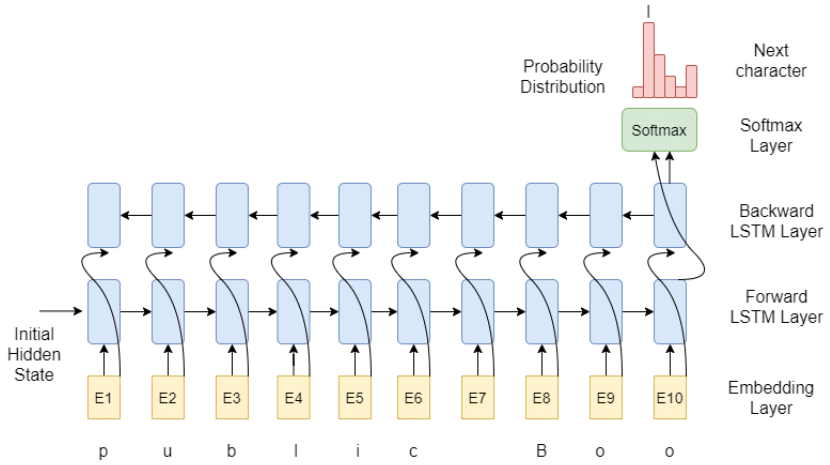
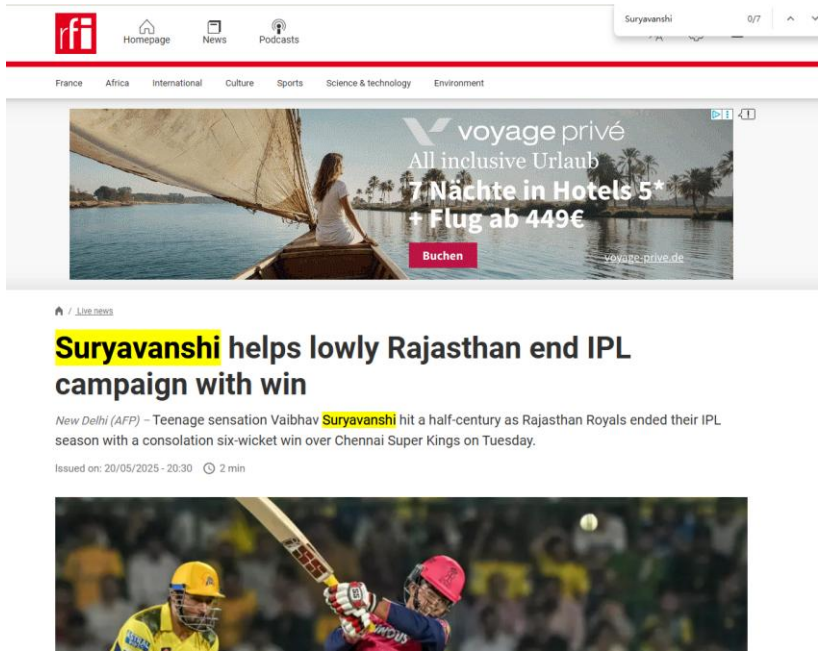


Figure 5: An example of a character-level language model

The Burstiness Problem

- **The Lingering Issue:**
  - Solved "can I form this word?"
  - But, early versions didn’t efficiently "remember"to reuse newly formed complex words (the "burstiness"problem).



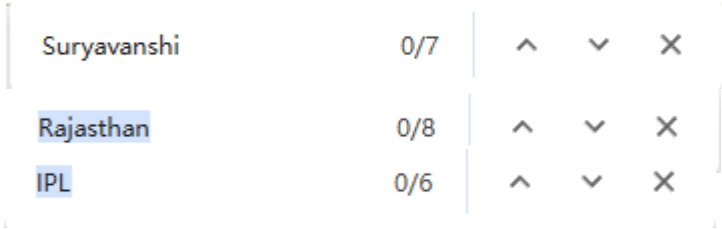


Figure 6: The Bustiness problem.

Our Focus: Kawakami, Dyer, Blunsom (2017)

- **Core Idea (2017):** An elegant solution combining:
  - Character-level generation.
  - A smart "memory"(cache) system.
- **Model Intelligently Decides:**
  - Generate from scratch (for new/varied words)?
  - OR "Copy"from cache (for recent, "bursty"words)?
- **Imagine:** Writing a report & coining a new term.
  - 1st time: Type carefully (char-level generation).
  - Next few times: Copy-paste / quick recall (caching!).

Seminar Roadmap

- **In Today's Seminar, We'll Explore:**
  - Their Hierarchical LSTM + word cache architecture.
  - How the model learns to create AND reuse words.
  - Key experiments and results showcasing its strengths.
  - The paper's impact on language model evolution.

Key Tech of the Era: LSTMs (Quick Refresher)

- **Why LSTMs? (~2015-2017 was peak LSTM!):**
  - Basic RNNs: Try to "remember"past to inform current predictions.
  - **Problem:** Simple RNNs struggled with "long-term memory"
- **LSTMs (Long Short-Term Memory):** Advanced RNNs with "gates"(input, forget, output) to control information flow.

- In 2017, LSTMs were NLP's go-to for sequence modeling:
  - Excelled at capturing longer dependencies in text.

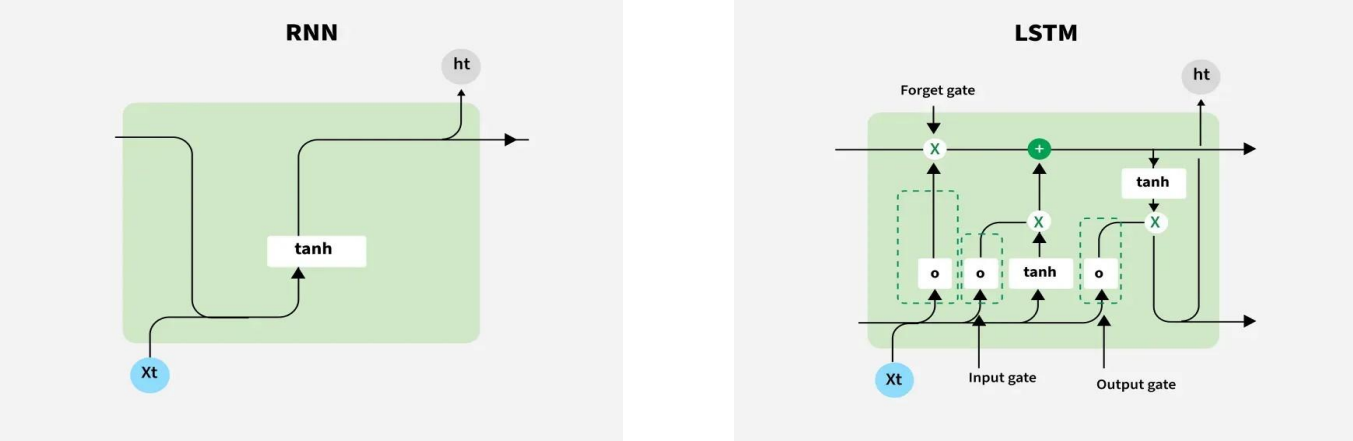


Figure 7: Architecture of RNN and LSTM

Model Overview

Hierarchical Character-Level Language Model with Cache (HCLM + Cache)

- **Character-level generation** for novel words
- **Cache mechanism** for reuse of prior words
- Built with 3 LSTMs + a memory module

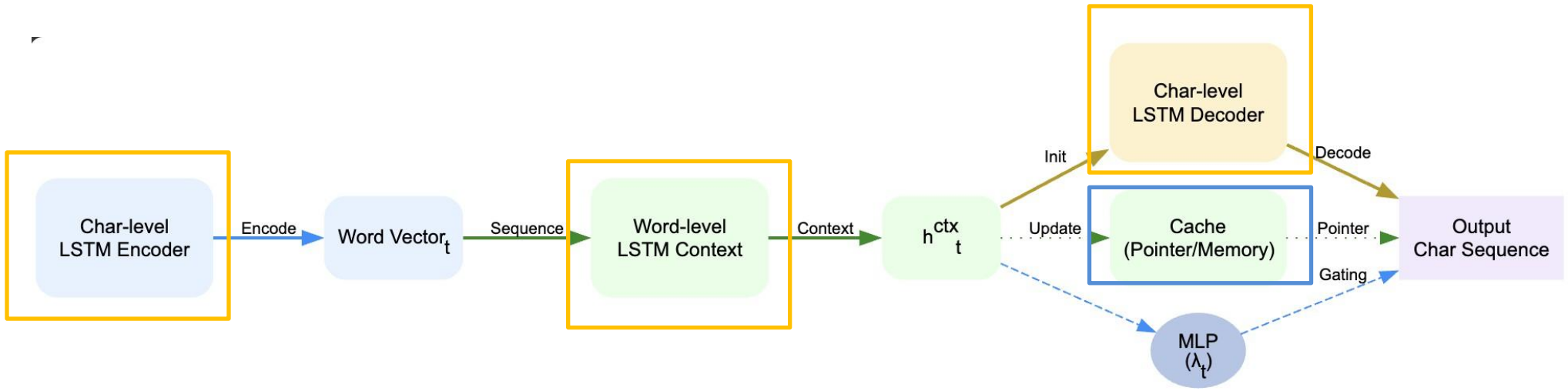


Figure 8: Model Overview

Model Overview

Hierarchical Character-Level Language Model with Cache (HCLM + Cache)

- **Character-level generation** for novel words
- **Cache mechanism** for reuse of prior words
- Built with 3 LSTMs + a memory module

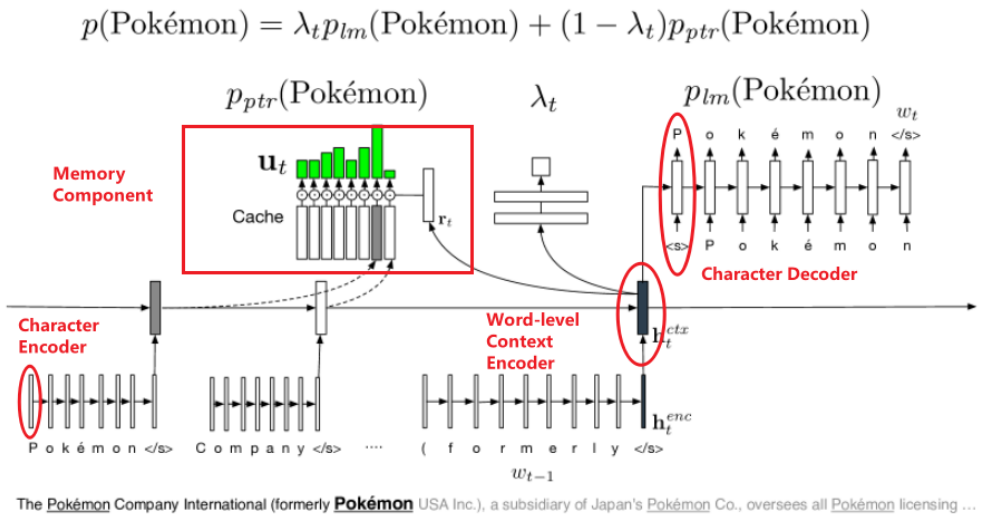


Figure 9: Description of HCLM

Model Overview

How the model predicts the next word w_t

Hierarchical Character-Level Language Model with Cache (HCLM + Cache)

- **1. Generating from Scratch:** Creating the word character by character, like spelling it out.
- **2. Reusing from Memory:** Picking a word it has seen recently. (Handled by the *Cache* part)

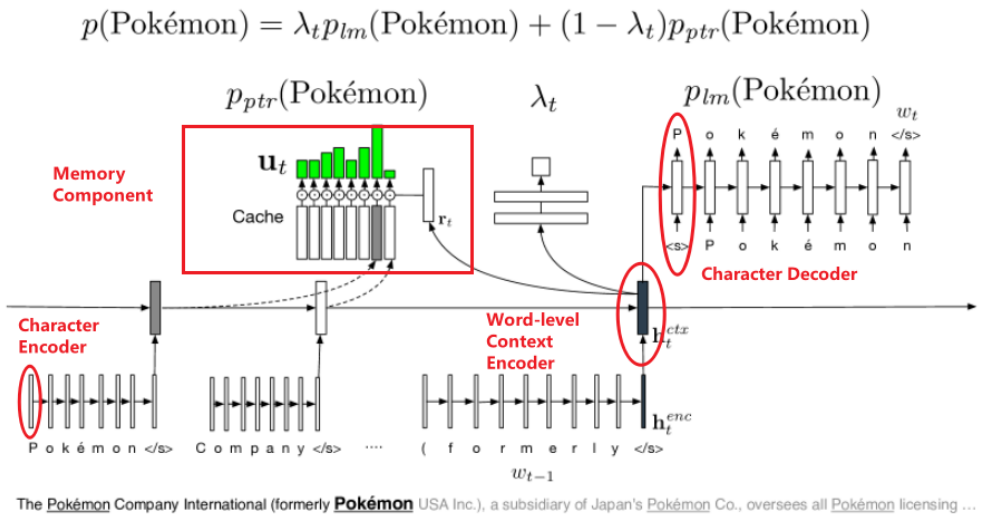


Figure 9: Description of HCLM

Path 1: Generating from Scratch (HCLM)

- **HCLM Role:** Generates words from scratch, character-by-character.
- **Two Levels of Hierarchy:**
  - **Characters → Word Vector:**
    - LSTM_{enc} reads previous word (e.g., w_{t-1}) char-by-char.
    - Output: Single vector h^{enc}_{t-1} representing w_{t-1} (meaning from chars!).

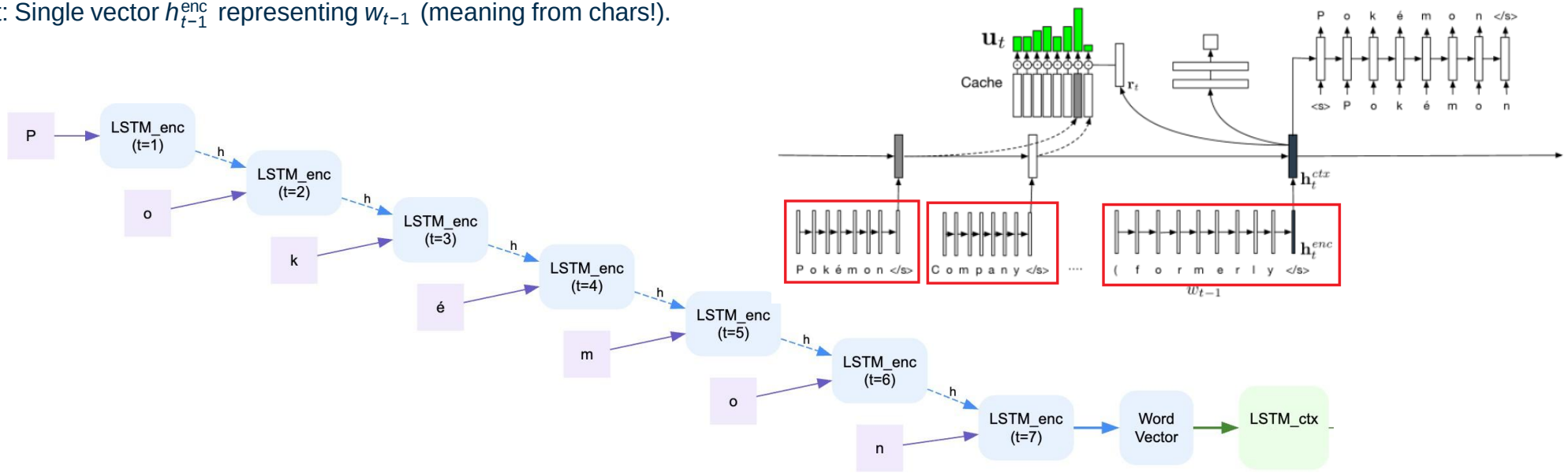


Figure 10: The character encoder's working process

Path 1: Generating from Scratch (HCLM)

- **HCLM Role:** Generates words from scratch, character-by-character.
- **Two Levels of Hierarchy:**
  - **Word Vectors → Sentence Context:**
    - LSTM_{ctx} processes sequence of word vectors ($h_1^{\text{enc}}, \ldots, h_{t-1}^{\text{enc}}$).
    - Output: Context vector h_t^{ctx} summarizing sentence so far.

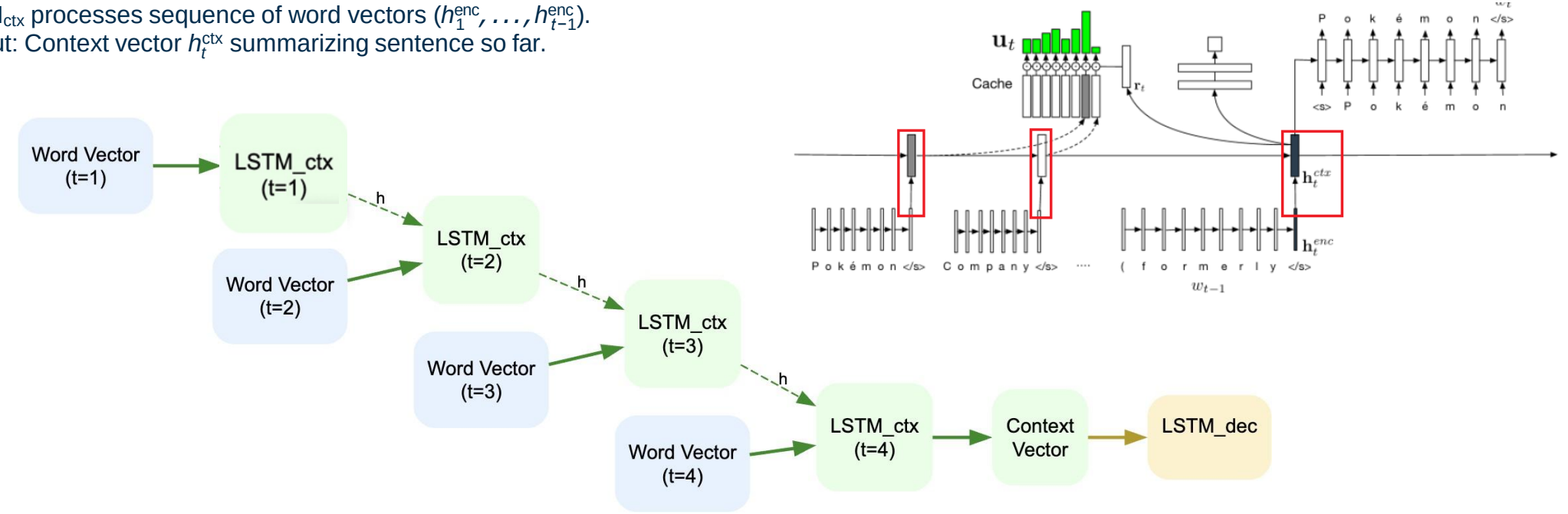


Figure 11: The world-level context encoder's working process

Path 1: Generating from Scratch (HCLM)

- **HCLM Role:** Generates words from scratch, character-by-character.
- **Two Levels of Hierarchy:**
  - **Generation → Next Word:**
    - LSTM_{dec} uses h_t^{ctx} as starting point.
    - Generates current word w_t one character at a time.

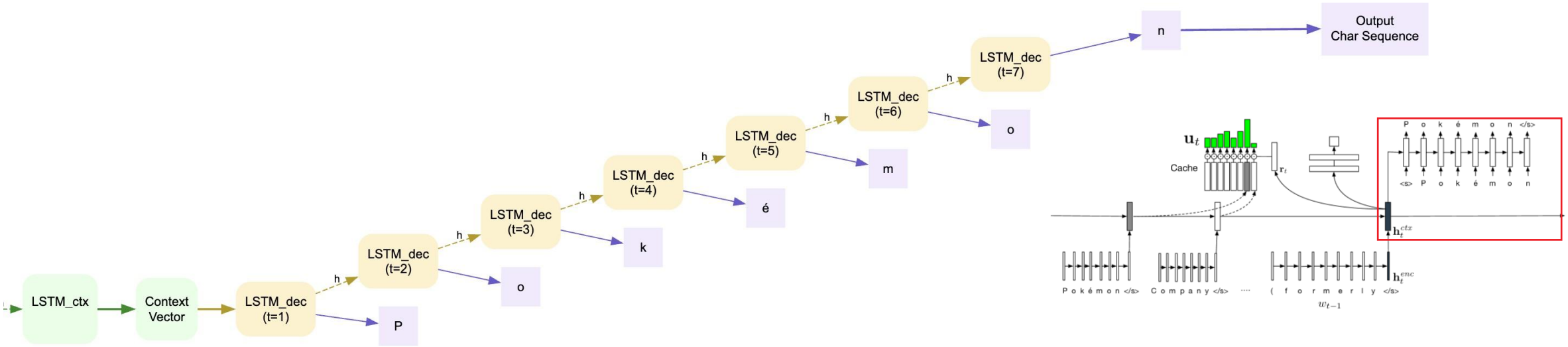


Figure 12: The character decoder's working process

Path 2: Reusing from Memory (Cache)

- **Cache Role:** Smart short-term memory for recent words.

- **How It Works:**
  - **Storing:** Generated word (e.g., "Pokémon") + its generation state (h_t) → added to cache.
  - **Limited Size (K items):** Least Recently Used (LRU) item removed if full.

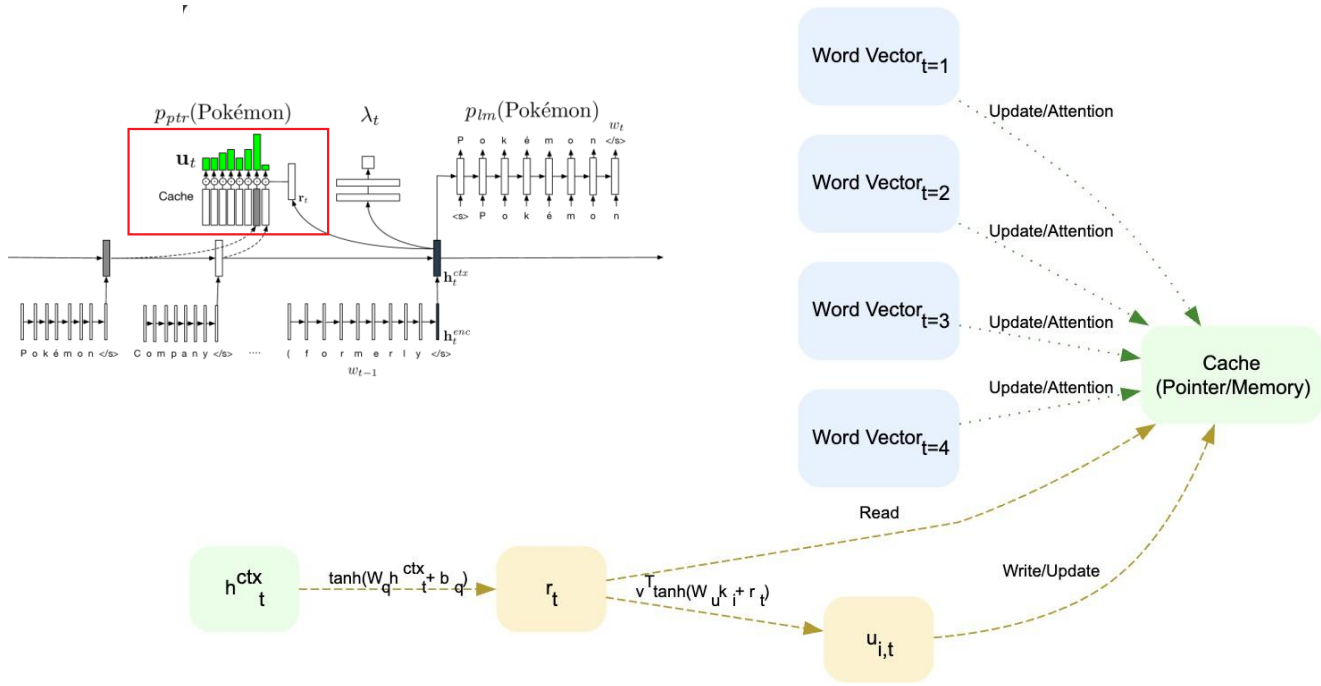


Figure 13: The cache mechanism's working process

Path 2: Reusing from Memory (Cache)

- **Cache Role:** Smart short-term memory for recent words.

- **Smart Retrieval ("Pointer"via Attention):**
  - Current context h_t → forms "query"r_t.
  - Query r_t compared to keys (stored states) in cache.
  - **Attention:** Calculates relevance scores for cached items.
  - Softmax over scores → probability of copying each cached word.

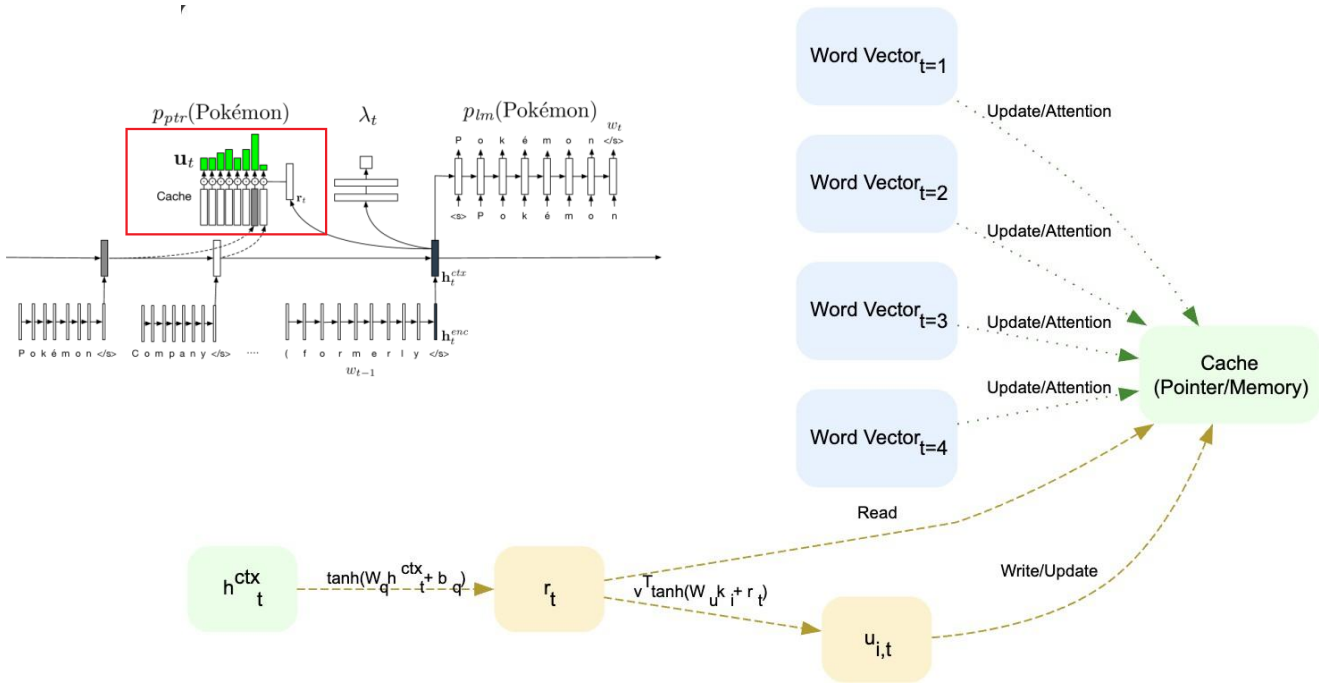


Figure 13: The cache mechanism's working process

Path 2: Reusing from Memory (Cache): Pointer Network over Cache

Given context vector h_t:

- Compute query vector:

$$r_t = \tanh(W_q h_t + b_q)$$

- For each cache key k_i, compute:

$$u_{i,t} = v^\top \tanh(W_u k_i + r_t)$$

- Pointer probability:

$$p_{\text{ptr}}(w_t|h_t) = \sum_{i:m_i=w_t} \text{softmax}_i(u_{i,t})$$

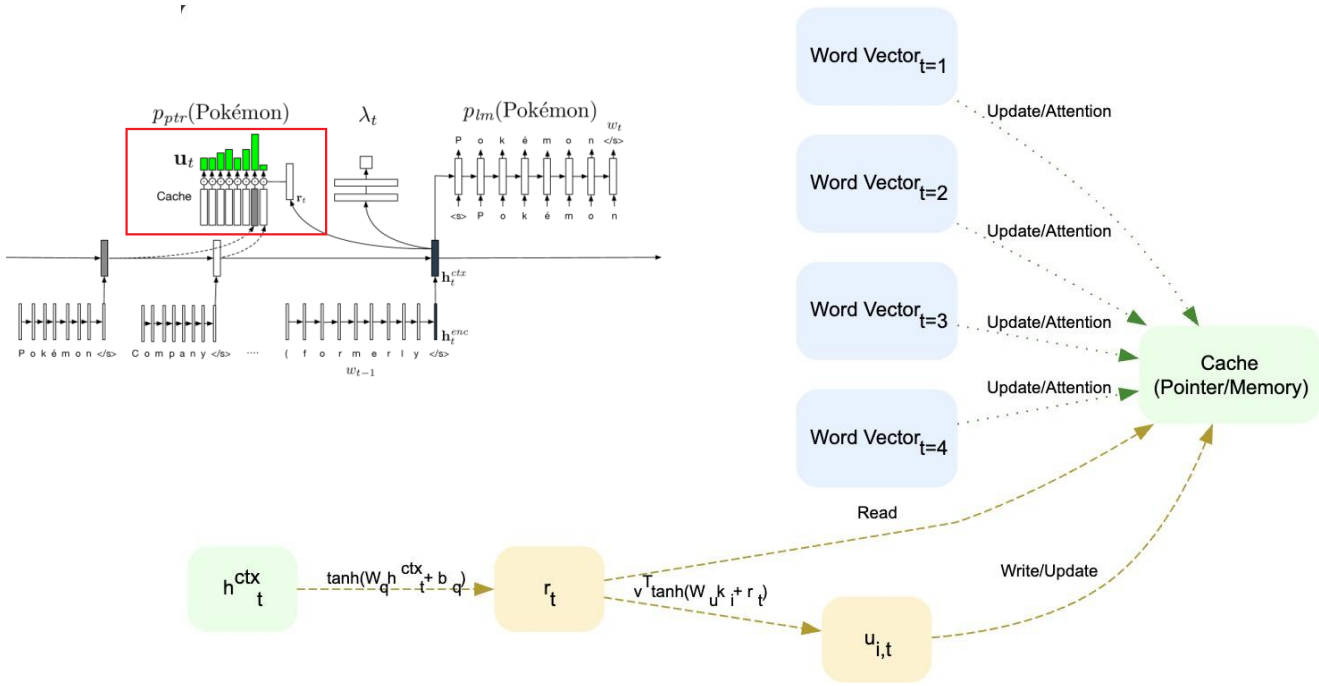


Figure 13: The cache mechanism's working process

The Smart Switch: Combining Both Paths

How the model decides to generate or reuse

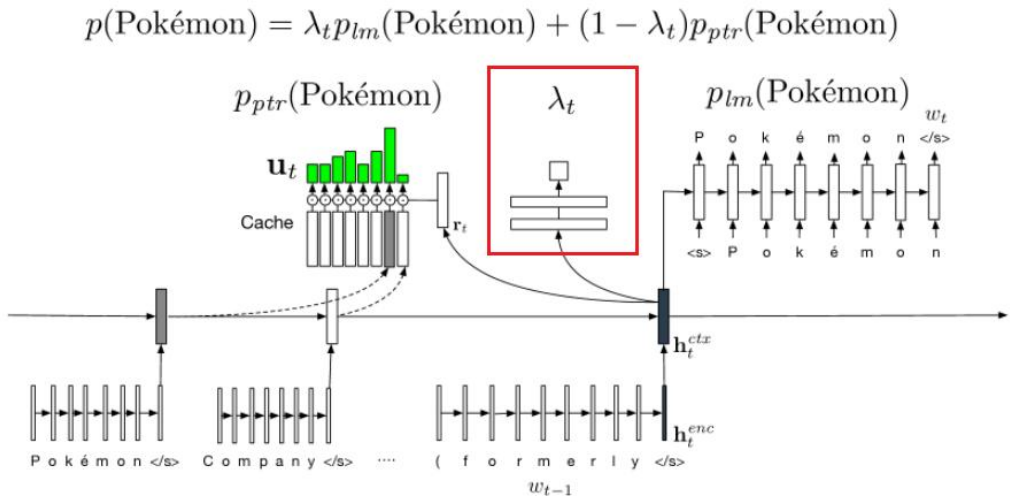
The model doesn't just guess; it intelligently blends the two paths:

- The final probability is a mix:

$$p(w_t) = \lambda_t \underbrace{p_{\text{lm}}(w_t)}_{\text{Generate}} + (1 - \lambda_t) \underbrace{p_{\text{ptr}}(w_t)}_{\text{Reuse}}$$

($p(w_t \mid w_{<t})$ shorthand for $p(w_t)$ above)

- **The Gatekeeper (λ_t):**
  - A small neural network (MLP) looks at the current context (h_t)
  - It outputs a value λ_t (between 0 and 1).
  - If $\lambda_t \approx 1$: Favors **generating** the word (HCLM).
  - If $\lambda_t \approx 0$: Favors **reusing** a word (Cache).



*The gate λ_t (center) decides between HCLM (right) and Cache (left).

Figure 9: Description of HCLM

Mixture of LM and Cache: A Question on λ_t

Mixing coefficient λ_t in the paper: The paper defines λ_t as:

$$\gamma_t = \text{MLP}(h_t)$$

$$\lambda_t = \frac{1}{1 - e^{-\gamma_t}}$$

- It is computed using an MLP based on the context h_t.
- **Question:** The paper's formulation for λ_t is $\frac{1}{1-e^{-\gamma_t}}$. Is this intended?
  - The standard logistic sigmoid is $\sigma(x) = \frac{1}{1+e^{-x}}$.
  - The paper's version could lead to values outside $[0, 1]$ or division by zero if $\gamma_t = 0$.

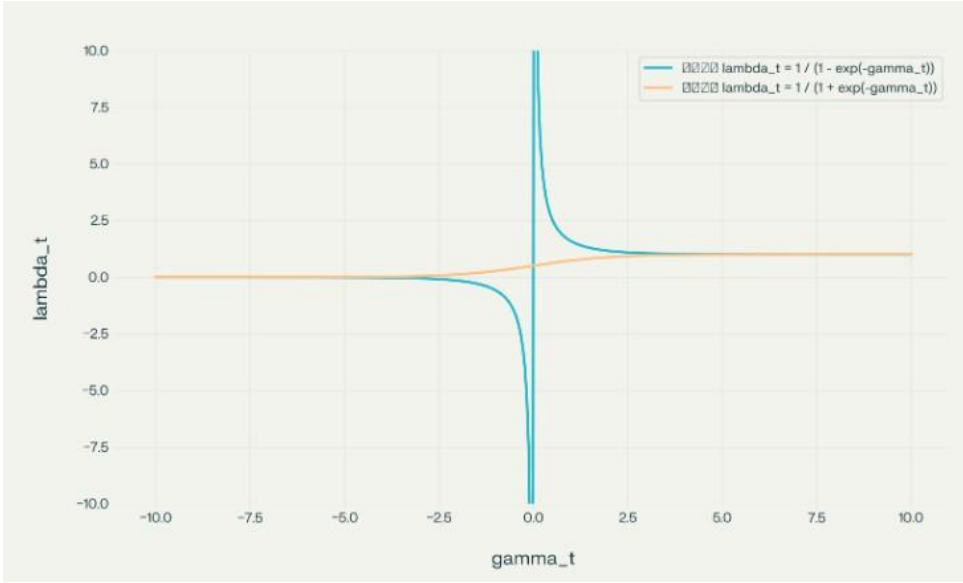


Figure 14: The graph of two functions

Datasets

- **Penn Treebank (PTB)**:
  - Preprocessed version with fixed vocabulary
  - No OOVs – serves as sanity check
- **WikiText-2**:
  - Open-vocabulary corpus from Wikipedia
  - Higher OOV rate, more realistic
- **Multilingual Wikipedia Corpus (MWC)**:
  - 7 languages: EN, FR, DE, ES, RU, CS, FI
  - Comparable articles across languages
  - Diverse morphological structures

Figure 15: Illustration of the datasets

Training Objective

Goal: Maximize log likelihood of training data words. (Model should assign high probability to real text). **Loss function:**

$$\mathcal{L} = -\sum_{t} \log p(w_t | w_{<t})$$

All parameters trained jointly:

- Character embeddings
- LSTM weights
- Attention (pointer) weights
- Mixture coefficient MLP

No supervision for when to copy!

Setting Up the Experiment (2017)

Key configurations and how success was measured

Model Configuration:

- LSTM hidden units: 600 hidden units
- Character embeddings: 600 dims
- Cache size: 100 word slots
- *Comparable to other strong models of the time.*

Training Details:

- Optimizer: Adam
- Standard techniques: Learning rate schedule, gradient clipping, dropout.

Evaluation Metric: Bits-Per-Character (bpc)

- **What is it?** Measures how surprised the model is by the next character it sees.
- Think of it like a guessing game:
  - If the model is very certain about the next character it uses fewer "bits"to encode it.
  - If it's very unsure, it uses more "bits".
- **Lower bpc is better!** Indicates a model that understands the language structure well.

A 2025 Perspective on Datasets & Experiments

Datasets:

- **2017:** PTB, WikiText-2, MWC
  Size: Millions to low Tens of Millions tokens
- **2025:** The Pile, SlimPajama, CC-Net
  Size: Trillions of tokens
  Diverse: Code, multilingual, instruction-tuned
  Long-context benchmarks

Ease of Replication & Advancement:

- **Compute Power:**
  - 2017: Titan Xp (12 TFLOPs FP32)
  - 2025: RTX 50xx (19.18-318 + TFLOPs FP32 + Tensor Core)
- **Tools & Libraries:**
  - 2017: Early PyTorch / TF1.x
  - 2025: Mature PyTorch/TF2/JAX, Hugging Face, AMP, advanced optimizers

Results: Penn Tree Bank (PTB)

Table 1: Results on PTB Corpus (bits-per-character).

Method	Dev	Test
CW-RNN Koutnik et al. (2014)	-	1.46
HF-MRNN Mikolov et al. (2012)	-	1.41
MI-RNN Wu et al. (2016)	-	1.39
ME n-gram Mikolov et al. (2012)	-	1.37
RBN Cooijmans et al. (2017)	1.281	1.32
Recurrent Dropout Semeniuta et al. (2016)	1.338	1.301
Zoneout Krueger et al. (2017)	1.362	1.297
HM-LSTM Chung et al. (2017)	-	1.27
HyperNetwork Ha et al. (2017)	1.296	1.265
LayerNorm HyperNetwork Ha et al. (2017)	1.281	1.250
2-LayerNorm HyperLSTM Ha et al. (2017)*	-	1.219
2-Layer with New Cell Zoph and Le (2016)*	-	1.214
LSTM (Our Implementation)	1.369	1.331
HCLM	1.308	1.276
HCLM with Cache	**1.266**	**1.247**

Key Insights Findings for PTB

- Cache adds boost even on non-ideal dataset
- Model competitive without complex tricks
- Suggests benefit even for frequent word repetition

Results: WikiText-2 (Realistic Open-Vocabulary)

Table 2: Results on WikiText-2 Corpus .

Method	Dev	Test
LSTM	1.758	1.803
HCLM	1.625	1.670
HCLM with Cache	**1.480**	**1.500**

Key Insights Findings for WikiText-2

- Cache improved performance by 10.2% over HCLM on challenging dataset
- Character-level model with cache rivals word-level models

Results: Multilingual Wikipedia Corpus (MWC)

Table 3: MWC Performance - HCLM+Cache vs. Baselines (Test bpc)

	EN		FR		DE		ES		CS		FI		RU	
	dev	test	dev	test	dev	test	dev	test	dev	test	dev	test	dev	test
LSTM	1.793	1.736	1.669	1.621	1.780	1.754	1.733	1.667	2.191	2.155	1.943	1.913	1.942	1.932
HCLM	1.683	1.622	1.553	1.508	1.666	1.641	1.617	1.555	2.070	2.035	1.832	1.796	1.832	1.810
HCLM with Cache	**1.591**	**1.538**	**1.499**	**1.467**	**1.605**	**1.588**	**1.548**	**1.498**	**2.010**	**1.984**	**1.754**	**1.711**	**1.777**	**1.761**

Key Insights Findings for MWC

- Significant improvement with cache
- Model effectively reuses word forms in varied linguistic structures.
- Architecture benefits (creating and reusing words) not language-specific.

How is the Cache Actually Used?

- **The Big Question:** Is the cache just a dumb buffer, or is it being used intelligently? The authors investigated!p(z | w): Average cache probability for word w after its first use.
- They looked at p(z|w) – the probability that a given word w was generated by copying from the cache (z=1) versus being spelled out by the HCLM (z=0).

Cache Use for General Words (WikiText-2 Test)

Examining $\overline{p(z \mid w)}$: avg. posterior cache probability (after 1st use)

- **Cache Favors:**
  - Punctuation frequent words (e.g., ".", "the").
  - Proper nouns (e.g., "Lesnar", "NY") due to burstiness.
- **LM Favors:**
  - Numbers (e.g., "300", "770"), which rarely repeat identically.
  - Common content words (e.g., "act", "however").
- **Conclusion:** Cache handles repetition; LM handles flexibility and non-repetitive words.

Table 4: Word Types with High/Low Cache Probability

Word	$p(z \mid w) \downarrow$	Word	$p(z \mid w) \uparrow$
.	0.997	300	0.000
Lesnar	0.991	act	0.001
the	0.988	however	0.002
NY	0.985	770	0.003
Gore	0.977	put	0.003
Bintulu	0.976	sounds	0.004
Nerva	0.976	instead	0.005
,	0.974	440	0.005
UB	0.972	similar	0.006
Nero	0.967	27	0.009
Osbert	0.967	help	0.009
Kershaw	0.962	few	0.010
Manila	0.962	110	0.010
Boulter	0.958	Jersey	0.011
Stevens	0.956	even	0.011
Rifenburg	0.952	y	0.012
Arjona	0.952	though	0.012
of	0.945	becoming	0.013
31B	0.941	An	0.013
Olympics	0.941	unable	0.014

Cache Use for Rare Words from Training Set

Words seen <5 times in training; (rare words in the training data)

- **Cache Favors (Rare Words):**
  - Proper nouns/entities (e.g., "Gore", "Nero") if reused in test.
  - Specific identifiers (e.g., "31B", "CR").
- **LM Favors (Rare Words):**
  - Numbers (e.g., "770").
  - Non-specific content words (e.g., "Pitcher", "consul") if not locally bursty.
- **Conclusion:** Cache effectively handles rare words if they become locally bursty in new contexts.

Table 5 : Cache Probability for Rare Training Words

Word	$p(z \mid w) \downarrow$	Word	$p(z \mid w) \uparrow$
Gore	0.977	770	0.003
Nero	0.967	246	0.037
Osbert	0.967	Lo	0.074
Kershaw	0.962	Pitcher	0.142
31B	0.941	Poets	0.143
Kirby	0.935	popes	0.143
CR	0.926	Yap	0.143
SM	0.924	Piso	0.143
impedance	0.923	consul	0.143
Blockbuster	0.900	heavyweight	0.143
Superfamily	0.900	cheeks	0.154
Amos	0.900	loser	0.164
Steiner	0.897	amphibian	0.167
Bacon	0.893	squads	0.167
filters	0.889	los	0.167
Lim	0.889	Keenan	0.167
Selfridge	0.875	sculptors	0.167
filter	0.875	Gen.	0.167
Lockport	0.867	Kipling	0.167
Germaniawerft	0.857	Tabasco	0.167

Cache Use for OOV Words

OOV word cache probability $\overline{p(z \mid w)}$ vs. test set frequency

- **Trend:** Higher test set frequency for OOV words often means higher cache probability.
- **Top-Right:** Frequently reused OOVs (e.g., new proper nouns) are cached.
- **Bottom-Left:** Infrequent OOVs (e.g., new common words) generated by LM.
- **Conclusion:** Cache effectively identifies and reuses bursty OOV words.

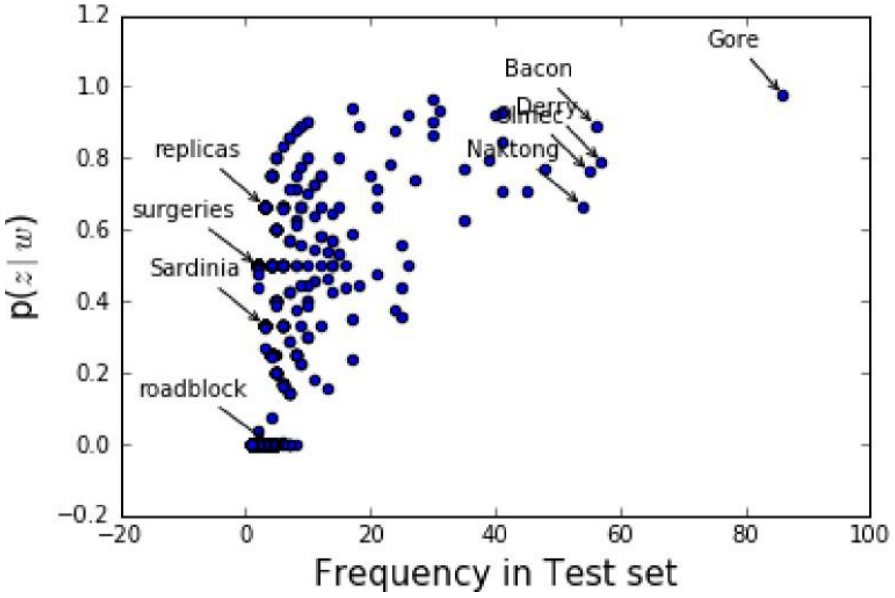


Figure 16: OOV word avg. cache probability vs. test set frequency.

Analysis Summary

Observations from data suggest the model learns:

Cache is preferentially used for:

- **Proper Nouns** (e.g., "Lesnar", "Gore", "Nero") → Captures burstiness.
- **Very Frequent Words** Punctuation (e.g., "the", ".", ",").
- **"Bursty" OOV words** if reused (often names)

Language Model (HCLM) is for:

- **Numbers** (e.g., "300", "770", "246") → Tend not to repeat.
- **Generic Content Words** (e.g., "act", “Sounds", "Pitcher").
- **Most OOV words**, especially if not repeated often.

Key Insight

- The model effectively distinguishes: word **creation** vs. word **reuse**.
- Cache adaptively handles "burstiness" of specific word types.

Key Contributions (Kawakami et al., 2017)

- Novel: Hierarchical Character LM + Adaptive Cache (HCLM+Cache) for open-vocabulary LM.
- Model creates new words (HCLM) captures ’bursty’ reuse (cache).
- New: Multilingual Wikipedia Corpus (MWC) for cross-lingual LM evaluation.
- Effective across diverse languages datasets (PTB, WikiText-2, MWC).

Adoption Since 2017: The HCLM+Cache Path

- HCLM+Cache (LSTMs): Not widely adopted long-term.

Why? Rapid NLP Evolution

- Shift: RNNs/LSTMs rightarrow Transformers (2017).
- Rise: Large-Scale Pre-training (BERT, GPT).
- Adoption: Subword Tokenization.

Modern LLMs: Tackling OOV Burstiness

How Large Language Models address these today:

1. **OOV Words (Open Vocabulary):**
   - **Subword Tokenization** (BPE, WordPiece).
   - Standard in BERT, GPT, etc.
   - Finite subword vocabulary rightarrow represents any word.
   - Reduces `<unk>`, better morphology.
2. **Word Reuse Context (Burstiness):**
   - **Transformer Architecture Attention:**
     - Self-attention weighs all prior tokens in context.
     - Implicitly captures burstiness co-occurrence.
   - **Longer Context Windows**.
   - **Large-Scale Pre-training** (learns complex patterns).

Limitations of the 2017 Paper (In Hindsight)

From a 2025 perspective, the approach had limitations:

- **Reliance on LSTMs:**
  - Less parallelizable during training compared to Transformers.
  - Harder to scale to the massive sizes of modern models.
- **Character-level processing:** Can be computationally slow for long sequences.
- **Assumes pre-segmented words:** This is problematic for languages without clear word boundaries (e.g., Chinese, Japanese, Thai). Subword models handle this more naturally.
- **Explicit Cache Necessity:** An explicit cache is useful for certain LSTMs but may be less critical or need redesign for strong Transformer models with robust attention-based contextual memory.

Inspirations Lasting Value from This Paper

Despite not being the dominant architecture today, the paper offers valuable insights:

- **Explicit Modeling of Linguistic Phenomena:** A reminder that directly addressing known properties (like burstiness, OOV) can guide model design and yield improvements.
- **Hybrid Approaches:** Cleverly combined fine-grained generation (character-level) with coarser-grained reuse (word-level cache).
- **Memory/Cache Concepts :** The idea of incorporating a local, dynamic memory or cache continues to inspire related concepts (e.g., aspects of Retrieval-Augmented Generation (RAG), memory networks).
- **Value of Multilingual Evaluation:** The MWC dataset and cross-lingual results highlighted the importance of testing beyond English early on.

References

- Kazuya Kawakami, Chris Dyer, and Phil Blunsom. 2017. Learning to create and reuse words in open-vocabulary neural language modeling.
  In *arXiv preprint arXiv:1704.06986*.
- Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017.
  Pointer sentinel mixture models.
  In *Proc. ICLR*.
- Edouard Grave, Armand Joulin, and Nicolas Usunier. 2017.
  Improving neural language models with a continuous cache.
  In *Proc. ICLR*.
- Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015.
  Neural machine translation by jointly learning to align and translate.
  In *Proc. ICLR*.

References

Figure 1: Graffiti on a wall in Crema, Italy.
Source: The presenter's photograph.
Figure 2: Collins Dictionary's 2024 word of the year.
Source: Collins Dictionary. Retrieved from https://www.collinsdictionary.com/woty
Figure 3: The <UNK> token issue.
Source: Generated by Perplexity AI.
https://www.perplexity.ai/search/wei-slidehua-yi-zhang-cha-tu-t-96JIgx1VSqy_A93uWQWL1g
Figure 4: The problem with <UNK> token.
Source: Generated by Perplexity AI.
Figure 5: An example of a character-level language model.
Source: ResearchGate. Retrieved from https://www.researchgate.net/figure/Overview-of-the-NER-model-to-generate-the-syntactic-code-embedding-for-LAMNER-The-input_fig3_360078902
Figure 6: The Bustiness problem.
Source: RFI News Screenshot. Retrieved from https://www.rfi.fr/en/sports/20250520-suryavanshi-helps-lowly-rajasthan-end-ipl-campaign-with-win-1
Figure 7: Architecture of RNN and LSTM.
Source: GeeksforGeeks. Retrieved from https://www.geeksforgeeks.org/rnn-vs-lstm-vs-gru-vs-transformers/
Figure 8: Model Overview.
Source: Generated using Python.
Figure 9: Description of HCLM.
Source: Figure 1 from original paper
Figure 10: The character encoder's working process.
Source: Generated using Python.

References

Figure 11: The world-level context encoder's working process.
Source: Generated using Python.
Figure 12: The character decoder's working process.
Source: Generated using Python.
Figure 13: The cache mechanism's working process.
Source: Generated using Python.
Figure 14: The graph of two functions.
Source: Generated by Perplexity AI. Retrieved from https://www.perplexity.ai/search/hua-chu-tu-zhong-ti-dao-de-zhe-d6g6M.eJTYK1UH21HUgbdw
Figure 15: Illustration of the datasets.
Source: Generated by Gemini. Retrieved from https://g.co/gemini/share/98f9a9ebe040
Figure 16: OOV word avg. cache probability vs. test set frequency.
Source: From the original paper, figure 3.
Table 1: Results on PTB Corpus (bits-per-character).
Source: From the original paper, table 4.
Table 2: Results on WikiText-2 Corpus.
Source: From the original paper, table 5.
Table 3: MWC Performance - HCLM+Cache vs. Baselines (Test bpc).
Source: From the original paper, table 6.
Table 4: Word Types with High/Low Cache Probability.
Source: From the original paper, table 7.
Table 5: Cache Probability for Rare Training Words.
Source: From the original paper, table 8.
Overleaf Template: "SDQ Presentation Template (2025)" on Overleaf (https://www.overleaf.com/latex/templates/sdq-presentation-template-2025/hhrwthdzdwfs)

Q&A

Thank You! Questions?