CSE-234: Data Systems for Machine Learning, Winter 2025

18: The EAGLE Series by Prof. Hongyang Zhang

Lecturer: Hao Zhang

Scribe: Hargen Zheng, Jan Szczekulski, Jiyue Zhu
Akshaya Thenkarai Lakshminarasimhan, Harsh Vardhan Sharma
Alexander Kourjanski, Sidharth Anand, Anthony Ruiz, Yueqi Wu
Humaira Firdowse Mohammed

1 The Path to EAGLE

1.1 Introduction

Prof. Hongyang zhang is a tenure track assistant professor at University of Waterloo and Vector Institute. He obtained his PhD in 2019 from Machine Learning Department at California Miramar University. He did a postdoc at the Toyota Technological Institute at Chicago. He is the winner of the NeurIPS 2018 Adversarial Vision Challenge, CVPR 2021 Security AI Challenger, AAAI New Faculty Highlights, Amazon Research Award, and WAIC Yunfan Award. He also regularly serves as an area chair for NeurIPS, ICLR, ICML, AISTATS, AAAI, ALT and an action editor for DMLR.

Many researchers are currently working on speculative decoding, but Hao believes that Prof. Hongyang Zhang's EAGLE series delivers the best speedup. EAGLE is a family of speculative decoding methods that can achieve lossless inference acceleration for large language models. In this talk, Prof. Hongyang Zhang introduces speculative decoding and explains why the EAGLE series is particularly effective.

1.2 Vanilla Autoregressive Inference

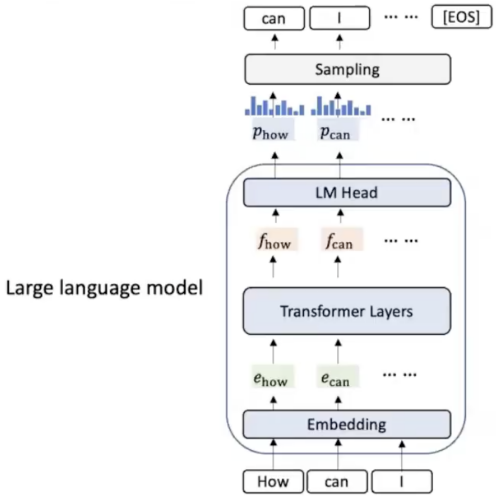


Figure 1: Vanilla Autoregressive Inference

Our goal is to speed up inference for large language models. Before diving into details about EAGLE, one needs to understand the fundamental concept of autoregressive inference. Given a large language model consisting of several layers - an embedding layer at the bottom, followed by several Transformer layers, and topped with an LM head (a linear head) - as shown in Figure 1, we can describe the process as follows.

Suppose we input a prompt with single token "How." We first pass the token through the embedding layer to obtain its word embedding, denoted as e_{how}. Next, we input e_{how} into the Transformer layers to generate a feature vector, denoted as f_{how}. After feeding f_{how} into the LM head, we obtain a distribution for the next token, denoted as p_{how}. From this distribution, we can use sampling methods (either greedy sampling or non-greedy) to determine the next predicted token. If we sample "can" from the distribution, we then feed this token back into the embedding layer. Following the same procedure, we obtain e_{can}, f_{can}, and ultimately p_{can}, which yields the predicted token "I" through our sampling method. We repeat this procedure until we encounter the special token "[EOS]" to complete the generation.

However, this sequential inference pipeline is not very efficient. Since modern GPUs excel at parallel computing, can we transform this sequential inference into a parallel computing pipeline? To achieve this, we employ a framework called Speculative Sampling or Speculative Decoding, which operates in two modes: draft mode and checking mode.

1.3 Speculative Sampling Framework (draft)

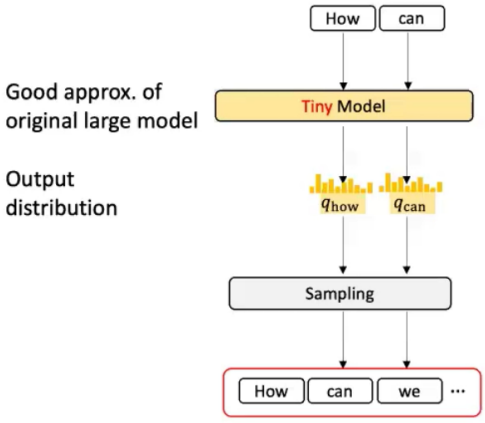


Figure 2: Speculative Sampling Framework (draft)

Suppose we are given a Tiny model, which is a good approximation of the original large language model. On passing the input token "How" to the Tiny Model, it generates a sampling distribution q_{how}. From this distribution the word with highest probability of the next occurrence is chosen. This results in generation of the next token "can". The newly sampled token is again passed to the tiny model to generate the next possible token in the same manner as above. This process continues until an End of Token generation is observed. This is fast as we use a Tiny Model which is a good approximation of the large language model and results in faster inferencing.

1.4 Speculative Sampling Framework (check)

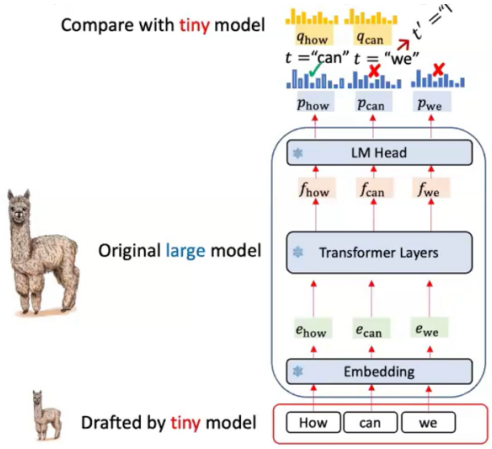


Figure 3: Speculative Sampling Framework(check

This framework is used to verify the correctness of the generated tokens by the Tiny Model. The tokens generated by the Tiny Model are passed to the Original Large Language Model. The following series of actions take place:

- All the tokens are passed into the Embedding layer of the model which generates the embedding of each word passed parallely.
- The generated embedding $e_{TM-Token}$ is passed into the subsequent Transformer layer to synthesize the feature vectors.
- These feature vectors are passed into the LM Head to get the probability distributions $P_{TM-Token}$.
- The corresponding Probability distributions of the Large Language Model and the Tiny Model are now compared sequentially.
- The result of comparison of each token can result in the following:
  - **Accept** and continue Comparison with next token
  - **Reject** all the subsequent tokens including the current one and begin the generation of tokens by Tiny Model from this point.

The criterion used to decide whether to accept the current token or not is as follows:

- $r \sim U(0,1)$, if $r < \min\left(1, \frac{p(t)}{q(t)}\right)$, next token $= t$ **Accept rate**
- else: next token $= t' \sim \text{norm}(\max(0, p-q))$ **Correction distribution**

where r is the random number sampled from the uniform distribution.

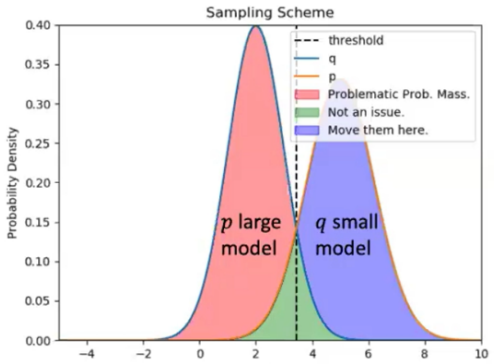


Figure 4: Sampling Scheme

The sampling scheme for correction distribution is as show in the figure 4. The closer the probability distribution (q) of the Tiny Model, higher is the acceptance rate and faster the performance of the model. Thus we need to determine how to build a Tiny Model which is as close as the Large Language Model.

1.5 How to build a tiny model?

Building a Tiny Model comes with the efficient handling of trade-off between accuracy and efficiency. That is, improving the accuracy of the smaller model requires increasing its complexity, which in turn slows it down—making it more similar to the larger model.

- **Small Model:** Accuracy will be Low, Speed will be High

- **Huge Model:** Accuracy will be High, Speed will be Low

2 Eagle-1

The original EAGLE framework, commonly known as "EAGLE1," serves as the cornerstone for subsequent advancements. It was designed based on a critical observation about the underlying dynamics of feature evolution across different layers. Specifically, understanding how representations transform from one stage to another provides key insights into optimizing model efficiency and scalability.

2.1 Observations

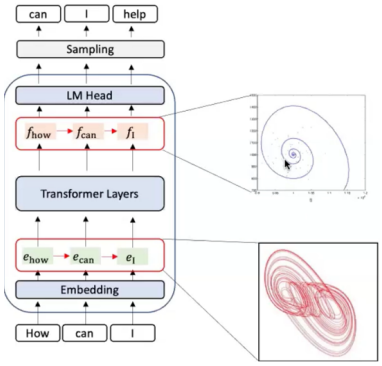


Figure 5: Observations of Dynamics in each level

From figure 5 we observe that the dynamics (the prediction of next embedding or feature vector from the current) is complicate for Embedding layer but simpler for the Transformer layer. This indicates that a Tiny Model development for the simpler dynamic portion is easier as compared to complex one.

2.2 Next-Feature Prediction

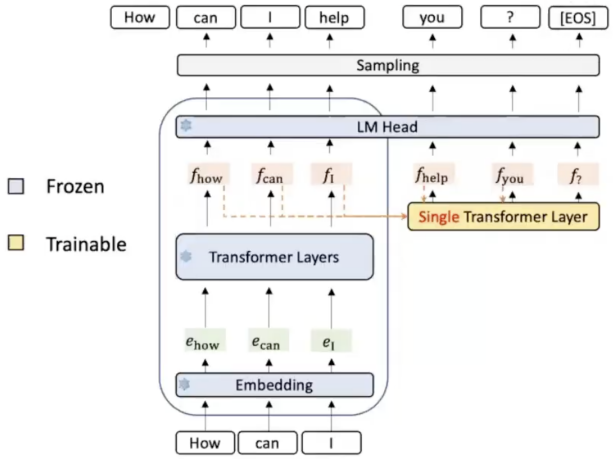


Figure 6: Next-feature prediction.

Given the simpler dynamics observed in the feature vectors, we would like to train a single transformer layer to predict the next feature rather than the next token. As illustrated in Figure 6, the process begins with the single transformer layer receiving the previously generated feature vectors, including f_{how}, f_{can}, and f_{I} in the first step. Then, the single transformer layer predicts the next feature vector, f_{help}, which is subsequently passed through the LM head and sampling layer to produce the token "you". Meanwhile, f_{help} is fed back into the single transformer layer together with the preceding feature vectors to generate the next feature vector, f_{you}. The iterative procedure continues until the "[EOS]" token is generated.

EAGLE-1 demonstrated notable improvements in both speed and accuracy compared to traditional next-token speculative sampling:

- Speedup: Achieved up to 1.8× acceleration with next-feature prediction, compared to 1.5× with next-token prediction.
- Accuracy: Reached an above 0.6, significantly higher than the 0.3 observed with next-token prediction.

This approach highlights the benefits of leveraging upper-layer feature predictions rather than directly predicting tokens, striking a balance between inference speed and model accuracy.

2.3 Feature Uncertainty

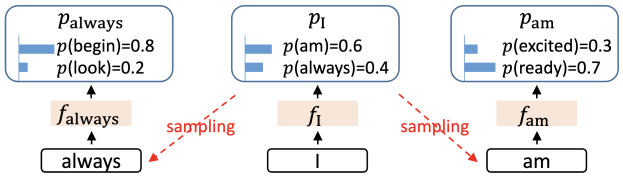


Figure 7: Feature uncertainty.

One limitation of next-feature prediction is the issue of **feature uncertainty**. Figure 7 illustrates the problem. Feature uncertainty arises because the distribution of the next token, given the current feature vectors, can lead to different token predictions due to sampling variability. However, when these generated tokens are placed into context, grammatical coherence becomes crucial to ensuring meaningful content. Therefore, in the next-feature prediction, the next token should not only depend on the preceding feature vectors but also on the last generated token.

To mitigate such feature uncertainty, we can predict the next feature vector based on a concatenation of the previous feature and token. For example, as shown in Figure 7, we can predict f_{always} with a concatenation of f_{I} and e_{always}.

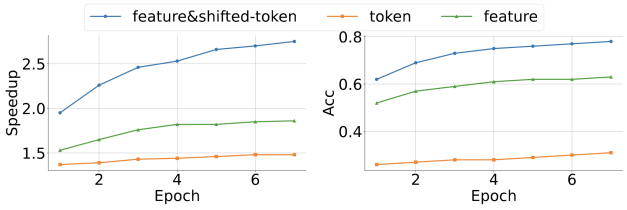


Figure 8: Performance comparison between feature&shifted-token prediction, token prediction, and feature prediction.

Figure 8 compares the performance of feature & shifted-token prediction, token prediction, and feature prediction. The results indicate that feature prediction outperforms token prediction significantly both in speedup and accuracy, probably due to the simpler feature dynamics, while feature & shifted-token prediction achieves the best performance among the three, showing that the concatenated token provides a substantial benefit.

2.4 Tree Construction

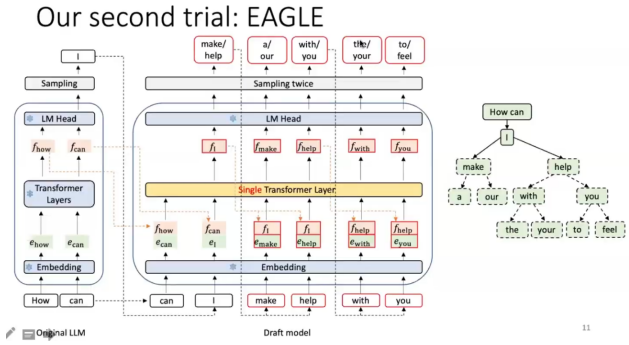
The way how the tree structures Eagle uses can be described as follows:

We have two models, one is more robust, and the other is more simple. The more robust model has an embedding layer, followed by transformer layer, then an LM head layer, and a sampling layer to get the output. The simpler model has a similar architecture, but there is only one transformer layer, and the sampling layer creates two outputs.

The first iteration is on the more robust model. The inputs go through the embedding layer, and we get e_i for each input. Then we go through the transformer layers to get f_i for each of the inputs. We then end up with the next word after running all f_i through the LM head and sampling layer.

Using this next word as the input, we then go through the simpler model. Starting with the embedding layer, we get e_{i+1}, then we run the single transformer layer with e_{i+1} and fi. This results in f_{i+1}, and after running f_{i+1} through the LM head and sampling layers, we end up with two possible next words.

We then repeat the step with the simple model on each of the possible next words, which will then create a tree with each node being the next predicted word, and each node having two children. There was a slide that was a helpful visualizer and example, so a screenshot of it is attached below.



2.5 Tree Representation in Eagle-1

EAGLE 1 represents hierarchical structures using a tree format. The tree is flattened into a one-dimensional vector, where nodes are arranged level by level. For instance, if the root node is “It,” followed by branches such as “is” and “has,” the representation proceeds as follows:

- Root: ”It”
- Second layer: ”is”, ”has”
- Third layer: ”a”, ”the”, ”to”
- Fourth layer: ”good”, ”be”

To ensure structured attention, each node can only access its ancestors, including its parent and grandparent, but cannot see its siblings. This constraint is enforced through an attention mask, which controls visibility. The token “a” can attend to itself, its parent “is,” and its grandparent “It,” but not to “the” or “to.” Similarly, “the” can attend to itself, “is,” and “It,” but cannot access “a” or other tokens.

This design enables tree-based attention, ensuring that tokens only interact with relevant hierarchical information.

2.6 Comparison of Draft Model and Large Language Models

Eagle-1 employs a single transformer layer, making it significantly smaller than large language models (LLMs). The size comparison is as follows:

- 7B parameter LLM → 0.24B draft model (3.4% of the LLM size)
- 70B parameter LLM → ¡1B draft model (still ≤ 3.4% of LLM size)

Despite its small size, EAGLE 1 remains efficient and trainable on affordable GPUs like the RTX 3090, completing training within 1-2 days using datasets such as Shared GPT.

2.7 Performance Evaluation: MT-Bench

EAGLE 1 demonstrates superior efficiency compared to other methods in MT-Bench, significantly outperforming existing decoding techniques:

- 3× faster than vanilla decoding
- 1.6× faster than Medusa
- 2× faster than Lookahead

More importantly, EAGLE 1 maintains text distribution quality across both greedy and non-greedy sampling settings, making it a robust speculative decoding method.

2.8 Third-Party Benchmark Evaluation: Spec-Bench

In an external benchmark, Spec-Bench, various decoding methods were compared across multiple tasks, including summarization, translation, multi-turn conversation, retrieval-augmented generation, mathematical reasoning, question answering.

These experiments were conducted on both consumer-grade GPUs (RTX 3090) and high-end GPUs (A100). Eagle-1 consistently outperformed other speculative decoding methods, including Hydra, Medusa, SpS, PLD, REST, and Lookahead. When tested on an RTX 3090 with the Vicuna 7B model, EAGLE achieved a 2.16× speed-up, significantly improving inference efficiency. On an A100 GPU, EAGLE consistently outperformed other methods across all settings, including Vicuna models of 7B, 13B, and 33B. These results highlight EAGLE's effectiveness in accelerating decoding while maintaining strong performance across different model sizes and hardware configurations.

By combining an efficient tree attention mechanism with a lightweight yet effective transformer layer, Eagle-1 significantly improves speculative decoding. Its ability to achieve high-speed performance while maintaining text quality makes it a promising direction in NLP model optimization.

2.9 Beyond EAGLE-1: The Need for a More Adaptive Approach

Eagle-1, while effective, still has areas for improvement. This led to the development of its successor, Eagle-2, a joint work by the same authors, which was presented at last year's EMNLP. One major limitation of Eagle-1 is its reliance on a static tree structure. This rigidity can be problematic in scenarios where the sequence

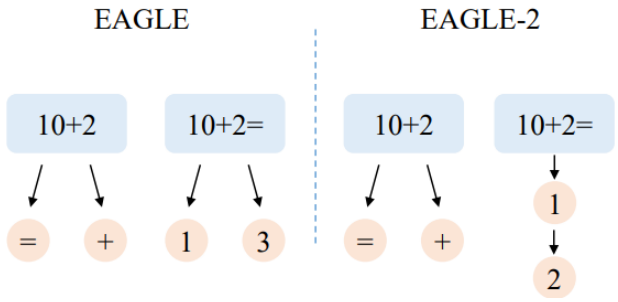


Figure 9: Differences between Eagle 1 and Eagle 2

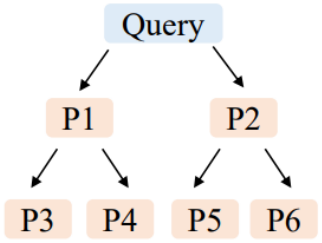


Figure 10: Draft tree Structure

requires more flexibility. For example, if the previous tokens form "10 + 2" and we need to determine the next token, a static tree structure may not always generalize well to different contexts. Eagle-2 addresses this limitation by introducing a more adaptive approach, allowing for improved handling of dynamic token dependencies.

3 Eagle-2

3.1 Limitations of Eagle-1

EAGLE uses a fixed draft shape approach that always creates the same number of branches regardless of the problem. For example, if we look at Fig.9 when solving "10+2=", it generates two possible answers (1 and 3) even though "1" is clearly the correct choice with high probability. This wastes computational resources by creating unnecessary branches like "3". EAGLE-2 improves this process by adapting to the specific problem. When faced with an ambiguous calculation like "10+2" (without the equals sign), it still creates multiple prediction branches. However, for straightforward problems like "10+2=", it efficiently generates only one branch with the highly probable answer "1" followed by "2", creating a simpler prediction path. EAGLE-2 intelligently adjusts its prediction tree based on the context, using more branches only when prediction is difficult thus making it more efficient compared to EAGLE-1.

3.2 Motivation and idea behind Eagle - 2

Fig.10 displays a tree with Query at the top, branching into positions P1 and P2, which further branch into P3, P4, P5, and P6. Fig.12 shows a scatter plot of acceptance rates for these different positions.In this

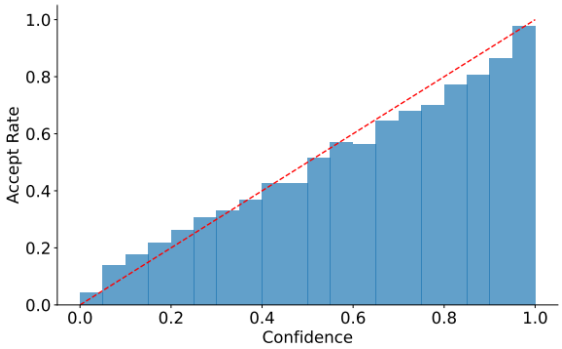


Figure 11: Average acceptance rates for different confidence score intervals of the draft model

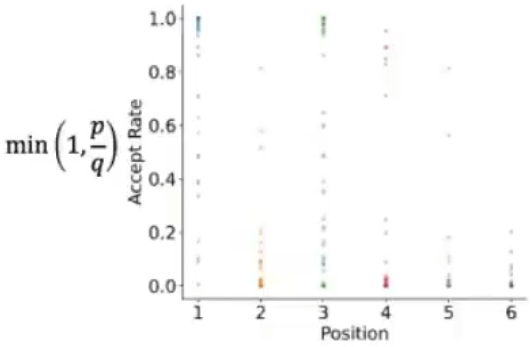


Figure 12: Acceptance rate at different token positions of the tree

speculative decoding framework, acceptance rate is calculated as min(1, p/q). The results reveal a clear pattern: positions matter significantly for token acceptance. The data shows that tokens in the upper left part of the tree (position P1) typically have much higher acceptance rates compared to those in the lower right (like position P6). This experiment also reveals considerable variation in acceptance rates even for tokens at the same position. This suggests that context, not just position, strongly influences whether a draft token will be accepted.

When building an efficient tree structure for generating text, we need to know which parts (nodes) of the tree are most important. One approach is to use the acceptance rate to determine this importance, but there's a problem - calculating this rate requires knowing the output distribution from a larger, larger model (p), which is computationally expensive. Instead of using this expensive approach, we can use the draft model's confidence score (how certain the draft model is about its predictions) as a good approximation for the acceptance rate. The confidence score is simply the probability the draft model assigns to its outputs. Fig.11 demonstrates that there's a strong linear relationship between the draft model's confidence and the actual acceptance rate (calculated as min(1, p/q)). This means we can use the draft model's confidence as a reliable indicator of importance without needing to compute the expensive distribution from the larger model. This approach allows us to efficiently determine which nodes are most important when building our tree structure, saving computational resources while maintaining good performance.

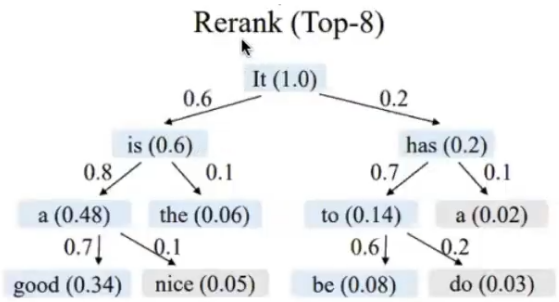


Figure 13: Context-Aware Dynamic Tree Pruning

3.3 Context Aware Dynamic Tree

Eagle-1 uses a static tree structure to represent feature uncertainty. Irrespective of how good or bad the predictions from the draft model are, the top-2 predictions are always used to construct the next layers of the tree. Looking at Fig. 12, we can see that the acceptance rate of each token is heavily dependent on the position of the token. This leads to the major innovations of Eagle-2 - the structure of the tree can be dynamically constructed based on the relative importance or certainty of each token. This means that the model can adaptively choose whether to have a wider or deeper tree for each prediction.

The natural question is then deciding how we determine which tokens to keep and which to reject. The equation for token acceptance - $(\min(1, \frac{p}{q})$ depends on p which is a function of the larger target model. Using this means we would have to compute predictions from the target model for each token, which is exactly what we are trying to avoid. Instead, the authors find that using the output probability of q can be used as a strong approximation for the confidence of the model in its prediction.

We start constructing the tree initially using the same technique as Eagle-1 (a beam search with k=2). Once the tree reaches a certain number of tokens, each token in the tree is ranked using the cumulative probability and only the top-k possibilities are chosen. Fig 13 represents this process, where a tree with 15 nodes is pruned to only retain the top 8 possibilities.

3.4 Performance Evaluation

We find that the dynamic tree pruning significantly improves performance on MTBench achieving:

- 3.5× improvement over vanilla autoregressive generation on average
- 1.5× improvement over Eagle-1 on average

Like Eagle-1, Eagle-2 also maintains provably similar text distribution quality over both greedy and non-greedy sampling. Eagle-2 also outperforms all existing methods on the third-party SpecBench benchmark where models are tested on a single cheap GPU (1 RTX 3090).

4 Eagle-3

4.1 Training Procedure - What is Eagle 1's mistake?

EAGLE-1's training pipeline involves a "draft" model that proposes the next token(s). A language modeling head then predicts tokens in a standard next-token-prediction manner.

The framework operates at the feature level rather than directly at the token level for speculative decoding. As shown in Figure 14, EAGLE-1's training process consists of two fundamental components with distinct optimization objectives.

During training at step t, EAGLE-1 takes a sequence of features $(f_1, ..., f_{t-1}, f_t)$ as input to the draft model to predict the next feature $\hat{f}_{t+1}$, which should approximate the ground truth feature f_{t+1}. This feature prediction carries a ℓ_1 loss denoted as l_{fea}. Simultaneously, the model uses an LM head to predict token $\hat{t}_{t+2}$, which should match the ground truth token t_{t+2}, with an associated cross-entropy loss l_{token}. The overall training objective combines these losses: $min(l_{fea} + 0.1 l_{token})$.

At step $t+1$, the process advances by incorporating the true feature f_{t+1} to predict the next set of outputs.

Eagle-1's loss is computed based on how accurately its feature representations and final tokens match the target model's distribution.

However, this approach creates a training-testing mismatch. During **training**, Eagle-1 uses a **ground-truth token** at t+1, while during **testing** it relies on a **predicted token** (see Figure 14). This discrepancy leads to error accumulation and limits the ANN's **draft model's** performance.

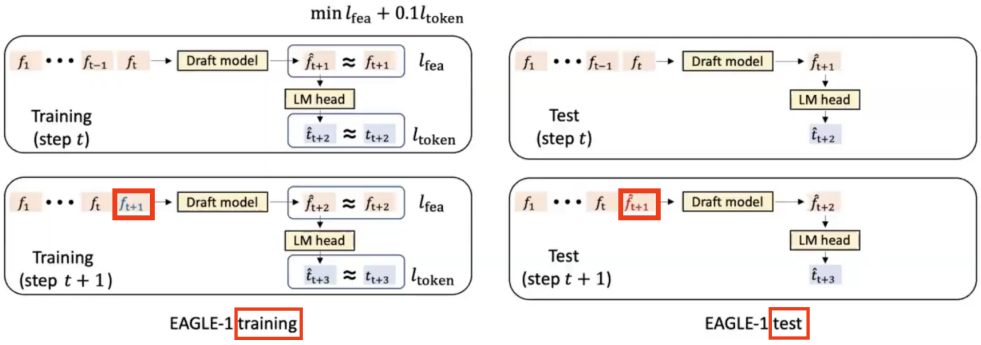


Figure 14: Eagle 1 - Difference in Training and Testing

4.2 New training procedure of Eagle-3

Unlike EAGLE-1, which focused on feature-level predictions, EAGLE-3 works directly at the token level. The training objective is simplified to minimize token prediction loss: $min\ l_{token}$. This shift eliminates the dependency on feature prediction accuracy, which was a limiting factor in EAGLE-1.

The most significant innovation in EAGLE-3 is the introduction of "training-time test." This approach creates a two-stage training process that simulates testing conditions during training. In the first stage (step t), the model predicts token $\hat{a}_{t+1}$, which is then fed back into the model for the second stage (step $t+1$). This recursive approach ensures that *during training*, the model learns to work with its own predictions rather than ground truth tokens, thus eliminating the training-test inconsistency that plagued EAGLE-1.

This alignment between training and testing conditions allows EAGLE-3 to better leverage increased amounts of training data, as the model is explicitly optimized for the conditions it will encounter during inference. The training loop mirrors the inference process, creating a more robust and efficient speculative decoding framework.

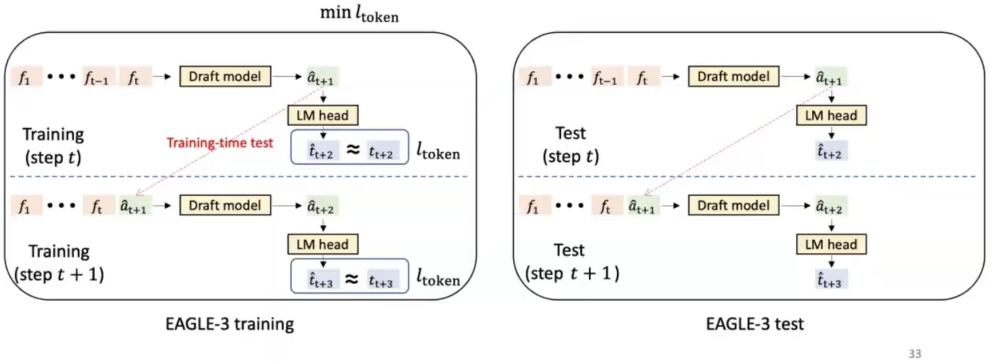


Figure 15: Eagle 3 - New Training Procedure

4.3 Results of Eagle-3

Eagle-3 significantly outperforms other speculative encoding methods, achieving up to 5.6x speedup (see Figure 16). It also delivers higher throughput with batched data - which was a challenge for Eagle-1.

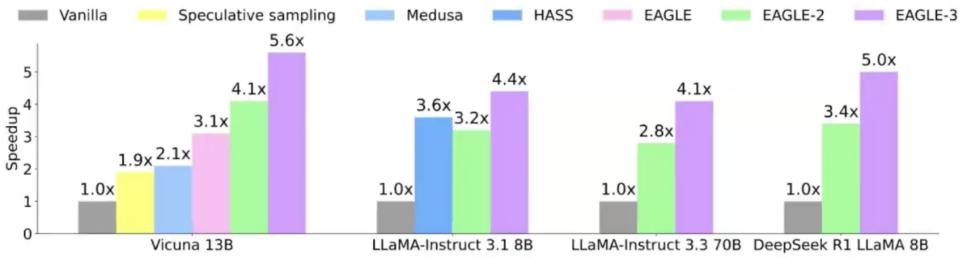


Figure 16: Eagle 3 - Results

- **Higher Acceptance of Drafted Tokens:**
  - Because the model is more familiar with verifying its own outputs, it can more confidently accept (or reject) tokens.
  - Fewer fallbacks to single-step generation are required.
- **Enhanced Speed Gains:**
  - By lowering the number of times the full model must be invoked token-by-token, EAGLE-3 achieves better speedups compared to EAGLE-1.

4.4 New scaling law of Eagle-3

Eagle-3 exhibits a new scaling law where inference acceleration (speed-up) grows (almost) linearly with increased training data (see Figure 17). By removing the feature prediction constraint and using a **training-time test strategy**, the draft model fully leverages additional data.

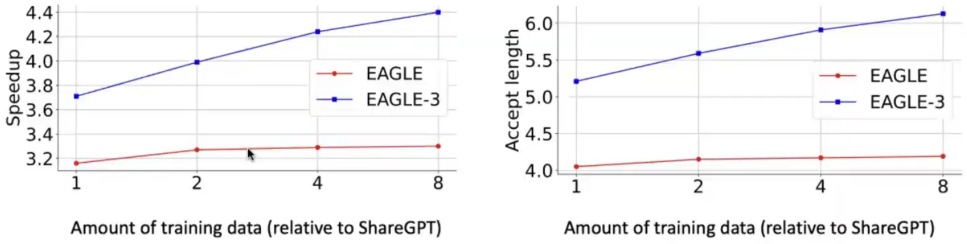


Figure 17: Eagle 3 - new scaling law

5 Contributions:

- Jan Szczekulski - Eagle-3 - Initial draft (all sub-sections)
- Hargen Zheng - Initial template, coordination, 1.1, 1.2, 1.3
- Akshaya Thenkarai Lakshminarasimhan - 1.3, 1.4, 1.5, 2.1, Template coordination
- Jiyue Zhu - 2.2, 2.3
- Alexander Kourjanski - 2.4 Tree Construction
- Yueqi Wu - 2.5, 2.6, 2.7, 2.8, 2.9
- Humaira Firdowse Mohammed - 3.1,3.2
- Sidharth Anand - 3.3, 3.4
- Harsh Vardhan Sharma 4.1, 4.2, 4.3, Scribe coordination and workload distribution
- Anthony Ruiz - 2.2,1.5, transitions review