Practical LLM-Based Lossless Compression: An Empirical Study

Mike Azure
Columbia University
ama2369@columbia.edu

Jasper Sands
Columbia University
js6908@columbia.edu

Abstract

Shannon's source coding theorem shows that strong predictive models can be used for lossless compression, linking prediction and compression. Recent work by Delétang et al. showed that large language models (LLMs), when combined with arithmetic coding, can achieve high compression ratios. However, their practical efficiency has not been systematically explored. In this paper, we implement an optimized compression pipeline that targets throughput using CUDA graph capture, static KV caching, and batched segment processing. We evaluate five models (GPT-2, Llama-3.2-1B, Llama-3.2-3B, Qwen3-0.6B, and Qwen3-1.7B) on a UTF-8 text subset of the Canterbury Corpus. On this benchmark, LLMs achieve average compression ratios of 5–16× (and up to 39× on individual files), compared to 2–4× for gzip and zstd, but at a cost of 100–1000× lower throughput. We analyze the trade-offs among compression ratio, throughput, and GPU memory usage.

1 Introduction

Large language models (LLMs) have shown strong performance on natural language understanding and generation tasks. However, their use is not limited to chatbots. From information theory, any model trained to predict a sequence can be used to compress that sequence [Shannon, 1948]. The cross-entropy loss used to train modern LLMs corresponds to the expected code length under arithmetic coding. As a result, reducing prediction error directly improves compression.

Delétang et al. [Delétang et al., 2023] formalized this connection and showed that models such as Chinchilla 70B achieve high compression rates across multiple modalities and outperform domain-specific compressors on text, images, and audio. Work by Valmeekam et al. [Valmeekam et al., 2023] explored similar ideas using Llama 2. However, these works focused on compression ratios and did not analyze practical efficiency.

There is a large gap between theoretical compression performance and practical use. Although LLMs achieve higher compression ratios, they require large computational resources and are orders of magnitude slower than classical compressors. Understanding this gap is necessary to determine when, or if, LLM compression is practical.

Contributions:

1. We reproduce the arithmetic coding method of Delétang et al. [Delétang et al., 2023] using open-source LLMs and show that models achieve higher compression ratios on text.
2. We implement several optimizations, including CUDA graph capture, static KV caching, and batched segment processing, to improve compression throughput.
3. We benchmark five models on seven test files and analyze compression ratio (input bytes / output bytes), throughput, and GPU memory usage.

Preprint. Under review.

2 Background

2.1 Shannon Entropy and Source Coding

The fundamental limit of lossless compression is governed by Shannon's source coding theorem [Shannon, 1948]. For a random variable X with probability distribution ρ, the Shannon entropy is defined as:

$$H(\rho) = \mathbb{E}_{x \sim \rho} [-\log_2 \rho(x)] = -\sum_x \rho(x) \log_2 \rho(x) \quad (1)$$

The theorem states that no lossless compression scheme can achieve an expected code length less than $H(\rho)$ bits per symbol.

2.2 Cross-Entropy and Compression

In practice, the true distribution ρ is unknown. Instead, we use a model $\hat{\rho}$ to estimate probabilities. The expected code length when using $\hat{\rho}$ to compress data distributed according to ρ is given by the cross-entropy:

$$H(\rho, \hat{\rho}) = \mathbb{E}_{x \sim \rho} [-\log_2 \hat{\rho}(x)] \quad (2)$$

For sequential data $x_{1:n}$, this becomes:

$$H(\rho, \hat{\rho}) = \mathbb{E}_{x_{1:n} \sim \rho} \left[\sum_{i=1}^{n} -\log_2 \hat{\rho}(x_i \mid x_{<i}) \right] \quad (3)$$

This is the same objective used to train autoregressive language models. Minimizing cross-entropy loss during training therefore minimizes the compression rate when the model is used with arithmetic coding.

2.3 Arithmetic Coding

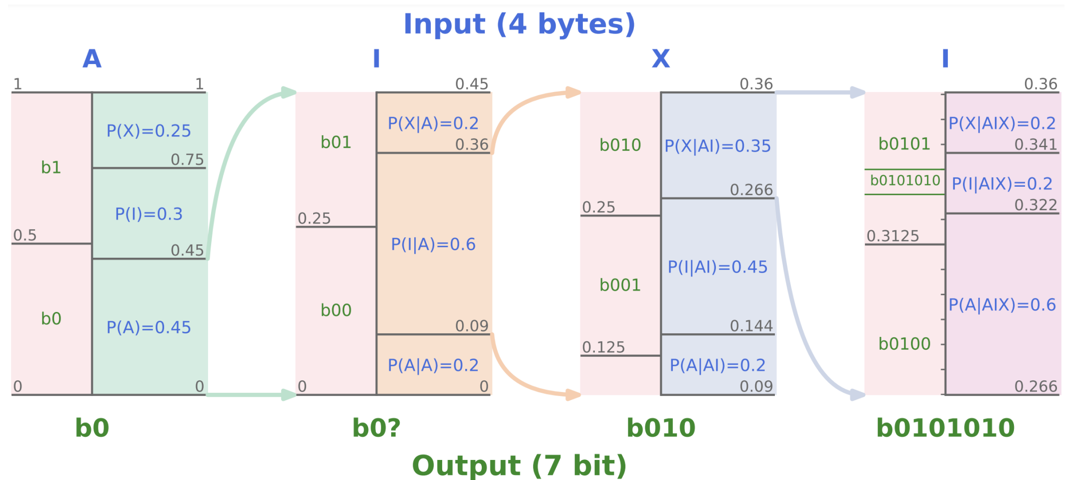

Figure 1: Arithmetic encoding of 'AIXI' with a probabilistic model P (blue) resulting in the binary code 'b0101010' (green). We iteratively divide the real interval $I = [0, 1)$ according to the model's (conditional) probabilities and select the sub-interval corresponding to the observed symbol (e.g., $I = [0, 0.45)$ for $P(A)$). We further refine I for each input symbol (indicated by the arrows), e.g., $I = [0.09, 0.36)$ for $P(I|A)$. To determine the encoded output, we iteratively split $[0, 1)$ in half and assign a binary code to each sub-interval (shaded red areas). At every step we can output the binary code if I is fully contained in the corresponding binary interval (e.g., 'b0' for 'A', but not for 'AI' as it could be 'b00' or 'b01'). At the end of the input, the code is 'b0101', which cannot be uniquely decoded ($P(A|AIX)$, $P(I|AIX)$, $P(X|AIX)$ all overlap with 'b0101'). Thus, we further refine the binary code until its binary interval is fully contained in I (all calculations in Appendix A). Figure from Delétang et al. [2023].

Arithmetic coding [Rissanen, 1976, Pasco, 1977] is a near-optimal entropy coding method that uses a probabilistic model. Given a sequence $x_{1:n}$ and a model $\hat{\rho}$, the encoder maintains an interval $I = [l, u) \subseteq [0, 1)$ and repeatedly narrows it using the model's conditional probabilities.

At step k, the current interval $I_{k-1} = [l_{k-1}, u_{k-1})$ is divided according to $\hat{\rho}(\cdot \mid x_{<k})$. The subinterval corresponding to the observed symbol x_k becomes the new interval I_k. After processing the full sequence, the output is the shortest binary representation of any value inside the final interval I_n.

The resulting code length is approximately $-\log_2 \hat{\rho}(x_{1:n})$ bits. If the model assigns high probability to the observed symbols, the output is short. If the model assigns low probability, the output is long.

3 Method

3.1 LLM-Based Compression

Our approach follows the method of Delétang et al. [Delétang et al., 2023]. We use a pretrained autoregressive language model to produce probability distributions over the vocabulary, which an arithmetic coder uses to encode the input text. Algorithm 1 shows the compression procedure used in this work. While the structure matches prior work, the algorithm reflects our implementation, including segment-based processing and parallel inference.

Algorithm 1 Lossless Text Compression with a Language Model

Require: Text T, tokenizer τ, language model $\mathcal{M}$, max segment length L, parallelism P
Ensure: Compressed bitstream B

1: $\mathbf{x} \leftarrow \tau(T)$
2: Split $\mathbf{x}$ into S segments $\{\mathbf{x}_1, \ldots, \mathbf{x}_S\}$ with $|\mathbf{x}_s| \leq L$
3: Let $f_s \leftarrow$ first token of $\mathbf{x}_s$ for all s
4: Initialize arithmetic encoder $\mathcal{E}$
5: Write header: total tokens $|\mathbf{x}|$, number of segments S, and first tokens $\{f_s\}_{s=1}^{S}$
6: **for** segments in batches of up to P segments **do**
7:     Reset model context (KV cache)
8:     Provide the first token of each active segment to the model
9:     **for** token position $t = 2$ to the maximum segment length in the batch **do**
10:         Obtain next-token distributions $\{\mathbf{p}\}$ from $\mathcal{M}$ for each active segment
11:         For each segment that has a token at position t:
12:             Encode that token using its distribution with $\mathcal{E}$
13:             Feed the token back into the model
14:     **end for**
15: **end for**
16: $B \leftarrow \mathcal{E}$.finalize()
17: **return** B

The decompression method mirrors compression. The decoder uses the same model to generate probability distributions, and the arithmetic decoder recovers the original tokens. The encoder and decoder must use the same model with identical weights, the same tokenizer, and deterministic inference settings. The probability distributions produced during decompression must match those used during compression exactly; otherwise decoding can diverge. When these conditions are met, the compression is lossless.

Why Sequential Processing Is Required: A tempting optimization is to obtain all token probabilities in a single forward pass. Given the full input sequence $x_{1:n}$, a transformer computes logits for all positions simultaneously: the output at position i depends only on $x_{\leq i}$ due to the attention mask. In principle, this yields $\hat{\rho}(x_2 \mid x_1), \hat{\rho}(x_3 \mid x_{1:2}), \ldots, \hat{\rho}(x_n \mid x_{1:n-1})$ in one pass.

However, the decoder cannot do this. During decompression, it does not know x_2 until it is decoded, and therefore cannot compute $\hat{\rho}(x_3 \mid x_{1:2})$ ahead of time. For lossless arithmetic coding, encoder and decoder must use probability distributions that are identical at every step. In practice, modern GPU inference can produce slightly different floating-point results across different kernel paths (for

example, different batching strategies, fused kernels, or accumulation orders). As documented in PyTorch's numerical accuracy guidance, floating-point arithmetic is not associative: changing the order of operations can change rounding and yield different results. If the encoder uses a computation path (such as a full-sequence batched pass) that the decoder cannot replicate exactly, even tiny discrepancies can cause the arithmetic decoder to select different tokens and diverge. We therefore run compression and decompression using the same sequential per-step inference pattern, leveraging the KV cache to avoid recomputing previous context.

3.2 Implementation Optimizations

LLM-based compression is slow due to the sequential nature of autoregressive inference. We implement several optimizations to improve throughput:

Static KV Cache: Rather than dynamically allocating key-value cache memory, we pre-allocate a static cache sized for the maximum sequence length. This reduces allocation overhead during compression and enables more predictable GPU memory usage.

CUDA Graph Capture: We capture the forward pass of the language model as a CUDA graph after initial warmup iterations. Graph replay reduces Python overhead and kernel launch latency, improving throughput for the repeated forward passes required during compression.

Batched Segment Processing: When compressing texts that exceed the model's context length, we process multiple segments in parallel. This amortizes fixed costs of inference across multiple segments and improves GPU utilization. We use P segments in parallel, with $P = 25$ in all experiments.

3.3 Experimental Setup

Models We evaluate five language models spanning different architectures and sizes:

- **GPT-2** (124M parameters): The original GPT-2 model [Radford et al., 2019]
- **Llama-3.2-1B** (1.24B parameters)
- **Llama-3.2-3B** (3.21B parameters)
- **Qwen3-0.6B** (0.6B parameters)
- **Qwen3-1.7B** (1.7B parameters)

All models use float16 precision and are run on a Google Cloud VM with an NVIDIA T4 GPU (16 GB VRAM), 4 vCPUs, and 15 GB RAM, using a maximum sequence length of 2048 tokens. GPT-2 only supports a max sequence length of 1024. For all experiments, we encode up to $P = 25$ segments in parallel.

Deterministic Inference: To ensure lossless decompression, we use deterministic inference settings in PyTorch.

Dataset: We use a subset of the Canterbury Corpus [Arnold and Bell, 1997], a standard benchmark for compression algorithms. We select files that are valid UTF-8 text, as our implementation operates at the text level rather than raw bytes. The selected files span different text types including literature (alice29.txt, asyoulik.txt, plrabn12.txt), technical documents (lcet10.txt), source code (fields.c, grammar.lsp), and manual pages (xargs.1). Delétang et al. [Delétang et al., 2023] also extend the same framework to arbitrary binary data, but we did not implement this variant.

Baselines: We compare against two widely-used classical compressors:

- **gzip** [Deutsch, 1996]: Level 9 (maximum compression)
- **zstd** [Collet and Kucherawy, 2016]: Level 22 (maximum compression)

Metrics: We report

- **Compression ratio**: Input size divided by output size (higher is better)
- **Bits per byte**: Output bits divided by input bytes (lower is better)
- **Throughput**: Input kilobytes processed per second (higher is better)
- **GPU memory**: Peak allocated GPU memory during compression

4 Results

4.1 Compression Performance

Table 1 presents the average compression performance across all test files for each model. LLM-based compression outperforms classical methods in compression ratio, with larger models generally achieving better ratios.

Table 1: Average compression performance across Canterbury Corpus files. LLMs achieve significantly better compression ratios than classical methods, but at the cost of throughput.

Model	Avg Ratio	Avg Bits/Byte	Throughput (KB/s)	GPU Mem (GB)
GPT-2	5.05	1.70	4.67	1.2
Qwen3-0.6B	7.92	1.12	0.37	5.3
Qwen3-1.7B	8.68	1.02	0.31	7.4
Llama-3.2-1B	10.07	0.87	0.51	3.5
Llama-3.2-3B	15.87	0.64	0.23	10.2
gzip (level 9)	2.82	2.89	13,510	–
zstd (level 22)	3.05	2.66	3,062	–

The best-performing model, Llama-3.2-3B, achieves an average compression ratio of 15.87×, compared to 3.05× for zstd and 2.82× for gzip. This represents a 5.2× improvement over the best classical compressor. However, this comes at a significant cost in throughput: LLM compression runs at 0.2–4.7 KB/s compared to thousands of KB/s for classical methods.

4.2 Per-File Analysis

Table 2 shows compression ratios for each file, comparing Llama-3.2-3B and GPT-2 against classical baselines. The improvement varies significantly by file type.

Table 2: Compression ratio comparison across Canterbury Corpus files. Llama-3.2-3B achieves the best ratios on all files, with particularly strong performance on natural language text.

File	Size (B)	GPT-2	Llama-3B	gzip	zstd
alice29.txt	148,481	6.35	39.07	2.78	3.05
asyoulik.txt	125,179	3.64	10.44	2.57	2.77
fields.c	11,150	3.13	18.96	3.57	3.70
grammar.lsp	3,721	5.35	10.94	3.02	3.07
lcet10.txt	419,235	6.69	10.51	2.94	3.49
plrabn12.txt	471,162	4.58	7.80	2.44	2.82
xargs.1	4,227	5.59	13.38	2.42	2.45

The most dramatic improvement is on alice29.txt (Alice in Wonderland), where Llama-3.2-3B achieves a 39.07× compression ratio, 12.8× better than zstd. This file contains highly predictable English prose that the language model can anticipate with high confidence, or has memorized in its training data. Conversely, plrabn12.txt (Paradise Lost) shows a smaller improvement (2.8×), likely due to its archaic English vocabulary and syntax being less represented in modern training data.

Interestingly, code files (fields.c, grammar.lsp) also compress well with LLMs, achieving 5.1× and 3.6× improvements respectively. This suggests that language models learn structural patterns in programming languages in addition to natural language patterns.

4.3 Scaling Analysis

Larger models generally achieve better compression ratios but suffer from slower throughput and higher memory usage.

Within the Llama family, scaling from 1B to 3B parameters improves the average compression ratio from 10.07× to 15.87× (58% improvement) while reducing throughput from 0.51 to 0.23 KB/s (55% reduction) and increasing memory from 3.5 to 10.2 GB (191% increase).

Similarly, within the Qwen family, scaling from 0.6B to 1.7B parameters improves compression from 7.92× to 8.68× (10% improvement) with comparable throughput but 40% higher memory usage.

4.4 Decompression Performance

An important property of our implementation is that decompression has nearly identical computational cost to compression. Both operations require the same sequence of model forward passes to generate probability distributions. In our experiments, decompression throughput matched compression throughput closely.

Table 3: Compression vs. decompression throughput (KB/s). LLM-based methods have roughly symmetric costs, while classical compressors decompress much faster than they compress.

Model	Compress	Decompress	Ratio
GPT-2	4.67	4.64	1.0×
Qwen3-0.6B	0.37	0.37	1.0×
Qwen3-1.7B	0.31	0.31	1.0×
Llama-3.2-1B	0.51	0.51	1.0×
Llama-3.2-3B	0.23	0.23	1.0×
gzip (level 9)	13,524	108,262	8.0×
zstd (level 22)	2,997	353,589	118.0×

This symmetry differs from classical compressors where decompression is typically much faster than compression. For gzip, decompression is 8× faster than compression (108,262 vs. 13,524 KB/s), while zstd shows an even greater asymmetry at 118× faster (353,589 vs. 2,997 KB/s). Both achieve decompression speeds four to five orders of magnitude faster than LLM-based methods.

5 Discussion

5.1 When Is LLM Compression Practical?

Our results demonstrate a clear trade-off: LLMs achieve higher compression ratios than classical methods but are orders of magnitude slower. This suggests LLM compression is practical only in specific scenarios:

Archival Storage: For data that is written once and rarely read, the superior compression ratio may justify slow compression speed. A 10× improvement in compression ratio directly translates to a 10× reduction in storage.

5.2 Effect of Training Distribution

A key factor in LLM compression performance is whether the input text is in the model's training data. Since compression quality depends directly on prediction accuracy, text that is in-distribution for the model will compress better than unfamiliar text. We define these categories as follows:

- **In-distribution text**: Content similar to modern web corpora used to train LLMs, including contemporary prose, technical documentation, and source code.
- **Out-of-distribution text**: Content that differs substantially from typical training data, such as archaic English, or specialized jargon.

Our results show substantial variation across file types. Alice in Wonderland (alice29.txt, 1865) achieves a 39× compression ratio, the best in our benchmark. Similarly, C source code (fields.c) compresses well at 19×, reflecting the abundance of code in LLM training corpora.

In contrast, archaic English texts compress worse. Paradise Lost (plrabn12.txt, 1667) achieves 7.8× compression, and As You Like It (asyoulik.txt, 1599) reaches 10.4×. These texts use older vocabulary and stylistic constructions that modern LLMs may predict less confidently.

Consistency Across Models: Table 4 shows that this pattern holds across all five models tested. We categorize alice29.txt, lcet10.txt, fields.c, and xargs.1 as in-distribution (modern prose, technical text, and code), and plrabn12.txt and asyoulik.txt as out-of-distribution (Early Modern English from the 16th–17th centuries).

Table 4: Compression ratio comparison for in-distribution vs. out-of-distribution text. All models compress in-distribution text better, and larger models amplify this gap. Classical compressors show minimal sensitivity to text distribution.

Model	**In-Dist Avg**	**Out-Dist Avg**	**Gap**
GPT-2	5.44	4.11	1.32×
Qwen3-0.6B	9.30	5.00	1.86×
Qwen3-1.7B	10.15	5.49	1.85×
Llama-3.2-1B	11.94	6.54	1.83×
Llama-3.2-3B	20.48	9.12	2.25×
gzip (level 9)	2.93	2.50	1.17×
zstd (level 22)	3.17	2.79	1.14×

All models compress in-distribution text better than out-of-distribution text. The gap increases with model size. GPT-2 shows a 1.32× gap, while Llama-3.2-3B shows a 2.25× gap. Larger models assign higher probability to common patterns, which improves compression on familiar text but helps less on unfamiliar text.

Alice in Wonderland shows this effect clearly. From Table 2, GPT-2 achieves 6.35×, while Llama-3.2-3B achieves 39.07×. Paradise Lost improves much less, from 4.58× to 7.80×. This suggests that texts common in training data benefit more from scaling.

In practice, LLM compression works best on modern web-like text and code. It performs worse on older or uncommon text. Classical compressors show more uniform performance across text types.

5.3 Limitations

Text Only Implementation: Our current implementation only supports UTF-8 text. Delétang et al. [Delétang et al., 2023] demonstrated that LLMs can compress images and audio by encoding bytes as text, but we did not implement this yet.

Model Size Overhead: We report compression ratios that exclude the model size. If the model size were included, often several GB, it would need to be included in compression ratio calculations.

5.4 Comparison to Prior Work

Our results are consistent with Delétang et al. [Delétang et al., 2023], who reported that Chinchilla 70B achieves approximately 0.66 bits per byte on enwik9. Llama-3.2-3B, achieves 0.64 bits per byte on average on our benchmark, suggesting that smaller modern models can approach the compression quality of much larger models on certain text distributions.

Our throughput numbers also reflect that LLM compression is orders of magnitude slower than classical methods. The optimizations we implemented (CUDA graphs, static caching) improve throughput but do not change this fundamental limitation.

6 Conclusion

Pretrained language models can be used as lossless compressors with arithmetic coding. Across five model families, LLMs achieve average compression ratios of 5–16×, compared to 2.8–3.1× for gzip and zstd. The strongest results are on natural language text. Llama-3.2-3B compresses Alice in Wonderland to 0.20 bits per byte, which is a 39× compression ratio.

The cost is computation. LLM compression runs at 0.2–5 KB/s, while classical methods run at thousands of KB/s. Compression and decompression have similar cost, which further limits use compared to classical methods.

CUDA graphs, static KV caching, and batched processing increase throughput, but they do not change this trade-off. More efficient model architectures or hardware support would be needed for LLM compression to be practical in more settings.

7 Future Work:

Sliding Window KV Cache: Instead of resetting the KV cache at each segment, a sliding window KV cache would allow compression while preserving context.

State-Space Models: State-space models such as Mamba achieve similar perplexity to transformers with linear-time inference. These models may provide similar compression with higher throughput.

Binary Data Compression: Our implementation supports only UTF-8 text. Prior work shows that LLMs can compress binary data by treating each byte as a token. We would need to extend our system to support binary data.

References

Ross Arnold and Tim Bell. The Canterbury corpus. Technical report, University of Canterbury, 1997.

Yann Collet and Murray Kucherawy. Zstandard compression and the application/zstd media type. Technical Report RFC 8478, IETF, 2016.

Grégoire Delétang, Anian Ruoss, Paul-Ambroise Duquenne, Elliot Catt, Tim Genewein, Christopher Mattern, Jordi Grau-Moya, Li Kevin Wenliang, Matthew Aitchison, Laurent Orseau, Marcus Hutter, and Joel Veness. Language modeling is compression. *arXiv:2309.10668*, 2023.

Peter Deutsch. GZIP file format specification version 4.3. *RFC*, 1996.

Richard C. Pasco. Source coding algorithms for fast data compression (ph.d. thesis abstr.). *IEEE Trans. Inf. Theory*, 1977.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. Technical report, OpenAI, 2019.

Jorma Rissanen. Generalized kraft inequality and arithmetic coding. *IBM J. Res. Dev.*, 1976.

Claude E. Shannon. A mathematical theory of communication. *Bell Syst. Tech. J.*, 1948.

Chandra Shekhara Kaushik Valmeekam, Krishna Narayanan, Dileep Kalathil, Jean-François Chamberland, and Srinivas Shakkottai. Llmzip: Lossless text compression using large language models. *arXiv:2306.04050*, 2023.

A Arithmetic Coding

Here we provide a step-by-step explanation of the arithmetic encoding example visualized in Figure 1, adapted from Delétang et al. [2023]. Recall that arithmetic encoding iteratively partitions the interval $I = [0, 1)$ according to a predictive model P and an input string, i.e., $AIXI$ for Figure 1.

First, we construct the intervals for the first token, corresponding to $P(\cdot)$:

- $[0, 0.45)$ for $P(A) = 0.45$
- $[0.45, 0.75)$ for $P(I) = 0.3$
- $[0.75, 1)$ for $P(X) = 0.25$

Since the first token is A, we set $I = [0, 0.45)$ and iterate. Thus, the intervals for $P(\cdot \mid A)$ are:

- $[0.2 * 0, 0.2 * 0.45) = [0, 0.09)$ for $P(A \mid A) = 0.2$
- $[0.09, (0.2 + 0.6) * 0.45) = [0.09, 0.36)$ for $P(I \mid A) = 0.6$
- $[0.36, (0.2 + 0.6 + 0.2) * 0.45) = [0.36, 0.45)$ for $P(X \mid A) = 0.2$

Since the next token is I, we set $I = [0.09, 0.36)$ and so on. We terminate with $I = [0.322, 0.341)$ for $AIXI$. Next, arithmetic coding computes the binary sequence corresponding to iteratively splitting the interval $[0, 1)$ in half until it is fully contained in I. Concretely, this yields the binary sequence:

- $b0 \to [0, 0.5)$
- $b01 \to [0.25, 0.5)$
- $b010 \to [0.25, 0.375)$
- $b0101 \to [0.3125, 0.375)$
- $b01010 \to [0.3125, 0.34375)$
- $b010101 \to [0.328125, 0.34375)$
- $b0101010 \to [0.328125, 0.3359375)$

As $[0.328125, 0.3359375)$ is fully contained in $I = [0.322, 0.341)$, the compressed output is 0101010, which consists of 7 bits as opposed to the 4 bytes used to encode $AIXI$.