TOKEN AVERAGING FOR COMPUTE-EFFICIENT LANGUAGE MODEL PRETRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Training language models on more text ordinarily requires proportionally more compute. We introduce *token averaging*, a parameter-free method that replaces each group of k consecutive token embeddings with their mean before the first attention layer. The transformer therefore processes k times fewer positions while predicting the first token of the next group. This allows a fixed training budget to cover approximately k times more raw text; at $k{=}2$, the model trains on twice as much text using 5–7% fewer FLOPs than the baseline (without token averaging). Since token embeddings are averaged, we randomize group boundaries during training and evaluate using k different boundary offsets, ensuring that the token-averaged model and the baseline are scored on identical target positions. In each offset, a sequence k times shorter is processed, so the cost of all k offset evaluations is approximately equal to one forward pass for the baseline. At inference, each pass stores $1/k$ as many KV-cache entries and incurs $1/k^2$ of the attention cost compared with the baseline, i.e., half the cache and one-quarter of the attention cost at $k{=}2$. Across models from 50M to 500M parameters, our experiments show that the baseline requires $1.30\times$, $1.28\times$, $1.17\times$, and $1.08\times$ more compute, respectively, to reach the same loss as the $k{=}2$ model. Full-position evaluation shows 3.4% lower perplexity at 125M with 7% fewer training FLOPs and comparable loss at 500M with approximately 5% fewer FLOPs. Additional experiments show that learned pooling over token content and within-group position reduces NLL relative to mean pooling by 0.043 nats at $k{=}2$ and 0.721 nats at $k{=}8$. At the tested scales, token averaging increases training-data throughput and reduces inference attention and KV-cache costs while maintaining comparable or lower evaluation loss.

1 INTRODUCTION

During language-model training, a standard transformer processes one sequence position and produces one output prediction per token. Increasing the number of training tokens at a fixed sequence length therefore proportionally increases both the linear and attention components of total training compute. Using the fitted exponents $\alpha \approx 0.34$ and $\beta \approx 0.28$ in the parametric loss function of Hoffmann et al. (2022), compute-optimal model size scales as $N_{\text{opt}} \propto C^{0.46}$, while the compute-optimal training-token count scales as $D_{\text{opt}} \propto C^{0.54}$. Thus, both increase with compute, with the token budget increasing somewhat faster. This motivates methods that process more training data within a fixed compute budget.

We study *token averaging*. Before the first transformer layer, we split the input into non-overlapping groups of k consecutive tokens and replace each group's embeddings by their mean. The transformer then runs on T/k positions instead of T, and each position predicts the first token of the following group. All other model and training components remain unchanged, and mean pooling adds no parameters. We call the standard $k{=}1$ model the *baseline*.

Because k raw tokens[1] share one transformer position and one output prediction, running the model over a fixed amount of text costs about $1/k$ as much as standard training. In the **more-data configuration**, our main setting, we use this reduction to train on $k\times$ more raw tokens. The baseline

[1]Throughout, *raw tokens* are the tokens produced by the tokenizer, counted before averaging, not the positions the transformer processes.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

and token-averaged configurations then process the same total number of transformer positions and predictions, while the shorter transformer sequences used with averaging further reduce attention FLOPs (Section 3.5). Averaging is lossy: at a fixed number of raw tokens, the averaged models have higher loss (Appendix D). The experiments measure whether processing more data offsets this difference. The **more-context configuration** instead increases the raw context by $k\times$ while keeping the transformer length fixed; a standard model with a 2048-token attention window provides the corresponding control (Section 3.4).

A k-averaged model assigns probabilities to only one token per group, so its loss and the baseline's are averages over different sets of predictions and are not measurements of the same quantity. We address this in two ways. First, we train with a *random group offset*, shifting the group boundaries by a random number of tokens at each step so that over training every token position is predicted equally often (Section 3.2). Second, endpoint comparisons use an *offset-ensemble* evaluation: we run the model k times over each evaluation sequence with the boundaries shifted by $o = 0, \ldots, k-1$ tokens, so that every position past the first group is predicted exactly once (Section 3.3). This gives the full-position negative log-likelihood (NLL) reported throughout, and because each of the k passes is over a sequence k times shorter, they together cost about one baseline forward pass. The same idea yields a scheme for generating text one token at a time (Appendix C).

Findings. On FineWeb with GPT-NeoX-style models trained from scratch:

1. **At equal compute, k=2 has lower loss from 50M to 250M parameters and comparable loss at 500M.** At 125M, its loss is 4.141 nats/token against 4.177 for the baseline, with $0.93\times$ the training FLOPs. At equal loss, the baseline requires $1.30\times$, $1.28\times$, $1.17\times$ and $1.08\times$ more compute at 50M, 125M, 250M and 500M, respectively (Section 5.1).
2. **At 50M, additional data accounts for more of the loss reduction than additional context.** A 2×2 experiment evaluates all combinations of $k \in \{1, 2\}$ and raw sequence length in $\{1024, 2048\}$; k=1 with sequence length 1024 is the baseline. Relative to this baseline, k=2 with sequence length 2048 reduces loss by 0.075 nats at equal compute. The k=2, 1024-token configuration contributes 0.049 nats through additional data, and increasing its raw sequence length to 2048 contributes the remaining 0.026 nats through increased context. At sequence length 2048, k=2 has 0.067 nats lower loss than k=1, with $0.874\times$ the training compute and half the KV cache (Section 5.2).
3. **Content-and-position weighting reduces loss relative to uniform averaging at k=2, k=4 and k=8.** Under the same experimental protocol, content-and-position weighting assigns each token a score based on its content embedding and position within the group. Its margin over uniform averaging increases from 0.043 nats at k=2 to 0.721 at k=8; content-only weighting remains within 0.011 nats of uniform averaging (Section 5.3).

Our contributions are an evaluation of token averaging as a permanent feature during pretraining rather than a temporary phase in related patch-based methods (Section 2); a protocol combining random-offset training and offset-ensemble evaluation; a controlled decomposition of the loss reduction into contributions from additional data and increased context; and a matched-protocol comparison of token-pooling variants.

2 Related Work

Compute allocation and data efficiency. Empirical scaling laws frame efficient pretraining as an allocation problem between model size and data: Kaplan et al. (2020) allocate most additional compute to model growth, whereas Hoffmann et al. (2022) revise the compute-optimal frontier toward approximately equal growth in parameters and training tokens. That prescription is ultimately constrained by useful data, since limited repetition can be nearly as effective as unique data but its marginal value eventually vanishes (Muennighoff et al., 2025). Token averaging acts on a complementary axis, fixing the tokenizer and parameter count while reducing how often the transformer body and vocabulary head run per raw token.

Patch-based and superposition training. The closest prior methods also average contiguous token embeddings. Patch-level training (Shao et al., 2025) mean-pools fixed groups and trains each patch

state against all tokens of the next patch through a shared predictive distribution; Token-Superposition Training (Peng et al., 2026) similarly combines contiguous tokens into bags under a multi-hot objective. Both use the compressed objective as an efficient pretraining phase and then return to ordinary token-level training. Our approach instead keeps averaging throughout pretraining and evaluation. It predicts one boundary token per group, randomizes group boundaries during training, and evaluates every offset to obtain full-position likelihoods. Related latent-input methods focus on inference with pretrained models, using learned compressors to replace multiple input embeddings with a shorter representation (Xu et al., 2026; Tan et al., 2026). Token averaging instead compresses tokens during pretraining itself and keeps the same compressed interface after training.

Sequence compression models. MEGABYTE (Yu et al., 2023) and Block Transformer (Ho et al., 2024) divide sequences into fixed patches. A global model processes patch representations, while a local decoder predicts individual tokens within each patch. BLT (Pagnoni et al., 2025) and SpaceByte (Slagle, 2024) apply a similar hierarchy to bytes, with patch boundaries selected by entropy or delimiters. These models add separate coarse and local components. Token averaging instead applies one pooling function before an otherwise unchanged transformer body and directly predicts one target per group.

Pooling is also used for different purposes. Funnel-Transformer (Dai et al., 2020) starts from tokenized input and progressively pools contextualized states between layers in a bidirectional encoder. Charformer (Tay et al., 2022) instead starts from bytes and learns subword blocks before encoding. Token averaging starts from token embeddings, pools once before a causal decoder, and retains the compressed positions throughout the network. Multi-token prediction (Gloeckle et al., 2024) changes the output objective instead, predicting several future tokens from each uncompressed hidden state.

Coarser tokenization. A coarser tokenizer can also shorten sequences, but it changes the vocabulary, the embedding and softmax costs, and the prediction units; vocabulary size has its own compute-optimal scaling law (Tao et al., 2024), and byte-level studies find an interior compute-optimal compression rate (Limisiewicz et al., 2026). Token averaging leaves the tokenizer, vocabulary, and target tokens unchanged, but compresses several input embeddings into one transformer position.

3 METHOD

3.1 TOKEN AVERAGING AND TRAINING OBJECTIVE

Let $t_1, \ldots, t_T$ be a sequence of raw tokens and $e_i \in \mathbb{R}^{d_{\text{model}}}$ their learned input embeddings. For a fixed averaging factor k, we split the sequence into $n = \lfloor T/k \rfloor$ non-overlapping groups of k consecutive tokens and replace each group by the mean of its embeddings,

$$\bar{e}_j \;=\; \frac{1}{k} \sum_{i=(j-1)k+1}^{jk} e_i, \qquad j = 1, \ldots, n. \tag{1}$$

The transformer body (attention layers, feed-forward layers, RoPE, final norm) runs unchanged on the compressed sequence $\bar{e}_1, \ldots, \bar{e}_n$, and the output head produces logits over the vocabulary at each of the n output states. At $k{=}1$ the method is exactly a standard language model. Mean pooling adds *no new parameters*; Section 3.6 describes the alternative pooling functions we compare, the largest of which adds 517.

Group j covers the raw tokens $t_{(j-1)k+1}, \ldots, t_{jk}$ and sits at compressed position j. At that position the model predicts the *first token of the next group*, t_{jk+1}: it has seen the first jk raw tokens in compressed form and must name the token that comes next. Figure 1 shows the whole pipeline at $k{=}2$, where the three predictions are t_3, t_5 and t_7. In a packed batch the target of the final compressed position is simply the next token of the stream, and only at the very end of a sequence is there no target, in which case that one position is dropped. The loss is ordinary cross-entropy, averaged over all predicted positions.

The model predicts only $t_{k+1}, t_{2k+1}, \ldots$, one target position per group; Section 3.3 defines evaluation across all target positions. Each prediction uses a lossy representation of the past because the order

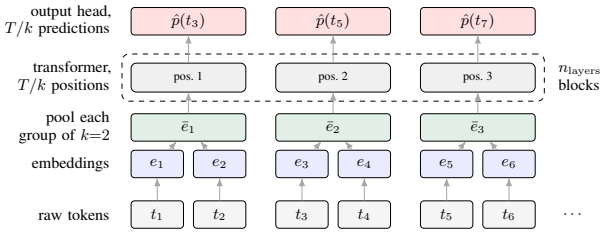


Figure 1: Token averaging at k=2. Every k consecutive embeddings are replaced by one pooled vector before the first transformer block, so the transformer and the output head run on T/k positions instead of T and each output position predicts the first raw token of the next group. Pooling is the only change to the architecture and adds no parameters; rotary position embeddings act on the compressed positions $1, \ldots, T/k$.

within each mean-pooled group is unavailable; Section 5.3 compares pooling functions that retain different information. Rotary position embeddings act on the compressed positions $1, \ldots, n$, so neighbouring transformer positions are k raw tokens apart. Shifting a group boundary changes the tokens in each group but not this positional geometry.

3.2 Random-offset training

If groups always started at the beginning of the sequence, only the raw positions $k{+}1, 2k{+}1, 3k{+}1, \ldots$ would ever be predicted and the rest would be ignored. Instead, at each training step we draw an offset $o \sim \mathrm{Uniform}\{0, \ldots, k{-}1\}$, drop the first o tokens of the batch, and then group as usual, so the whole pattern of predictions slides along the text by o tokens. Since o takes each of its k values about equally often, every token position is predicted about equally often over the course of training. We lose only the first $o \leq k{-}1$ tokens, which shorten each sequence by less than one group, so a batch loses at most one prediction per sequence. Nothing else about training changes.

3.3 Full-position evaluation with offset ensembles

Standard language-model evaluation averages the log-likelihood terms $\log p(t_i \mid t_{<i})$ over token positions. The k=1 baseline predicts all $T{-}1$ positions, whereas a k-averaged model natively predicts about T/k positions. Their native losses therefore average different prediction sets and are not directly comparable.

To address this, we run the model k times over each evaluation sequence, shifting the group boundaries by $o = 0, 1, \ldots, k{-}1$ raw tokens, so that at offset o the targets are the raw positions $o + jk + 1$ for $j = 1, 2, \ldots$. Across the k offsets these target sets never overlap, and together they cover every position past the first group, each predicted exactly once from its full compressed prefix. At k=2 on eight raw tokens, for instance, offset 0 groups $[t_1t_2][t_3t_4][t_5t_6][t_7t_8]$ and scores t_3, t_5, t_7, while offset 1 drops t_1, groups $[t_2t_3][t_4t_5][t_6t_7]$ and scores t_4, t_6, t_8, so the union of the two target sets is every position from t_3 onwards and no position is scored twice (Appendix C).

We pool the scored predictions across offsets and evaluate the baseline on the same target positions. The resulting quantity is the full-position NLL used for comparisons across k. Its k passes each have length approximately T/k; the linear compute is therefore similar to one baseline pass and the attention compute is lower. Appendix C gives the complete evaluation procedure, per-offset results, and the corresponding rotating-offset generation scheme.

3.4 Training configurations: more data or more context

Throughout, `seq_len` is the number of *raw* tokens per training sequence and $L =$ `seq_len`$/k$ is the number of positions the transformer actually processes. In the **more-data** configuration the raw `seq_len` stays at the baseline's 1024, so $L = 1024/k$, each raw token costs about $1/k$ of baseline, and the model trains on $k\times$ the raw tokens for the same total compute; every prediction still sees 1024 raw tokens of context, and only the way that context is represented changes. In the **more-context** configuration we instead raise `seq_len` to $k \times 1024$ so that $L = 1024$ matches the

baseline's transformer length, packing $k\times$ the raw context into the same computation. The two configurations use identical models, data, and optimization. A third configuration tests whether any effect of the extra context is specific to averaging: a standard $k{=}1$ model whose attention window is widened to 2048 tokens (`seq_len` $= L = 2048$), which gets the same context without compression but incurs additional attention compute.

3.5 FLOPs ACCOUNTING

We compute all FLOPs from first principles. For hidden size d, n_{layers} layers and transformer sequence length L, our architecture (attention with Q, K, V and output projections; SwiGLU FFN with expansion factor 2.5) costs $23d^2 + 4Ld$ multiply–accumulate FLOPs forward per layer per position: $8d^2$ for the four attention projections, $15d^2$ for the SwiGLU FFN, and $4Ld$ for the attention scores and value aggregation (Appendix A). Each position also produces one next-token prediction through the dense vocabulary projection at $2dV$ FLOPs forward, for padded vocabulary $V = 50{,}304$. Multiplying by the usual $3\times$ for forward plus backward, a run that consumes D raw tokens at raw sequence length `seq_len` costs

$$\text{FLOPs}(D) = \frac{3D}{k}\Big[\, n_{\text{layers}}\left(23d^2 + 4Ld\right) \;+\; 2dV \,\Big], \qquad L = \texttt{seq_len}/k. \tag{2}$$

We include the output projection because it is a substantial fraction of total compute at these scales: the $2dV$ term accounts for 44.2% of the baseline's training FLOPs at 50M and 12.0% at 500M. Appendix A gives the full breakdown and shows that excluding this term would inflate every efficiency ratio we report.

For the more-data configuration, Eq. (2) has two relevant cases. *At a fixed number of raw tokens* D, averaging processes D/k positions instead of D, so the two terms linear in the position count, the $23d^2$ per layer and the $2dV$ head, fall by a factor k per raw token, while the attention term falls by k^2: one factor k from having fewer positions and another because each position attends over a sequence k times shorter. *At the endpoint budgets used here*, the averaged run processes kD raw tokens. Its total position and prediction counts then equal the baseline's, so the two linear terms also equal the baseline totals; only the lower attention cost remains. This gives total cost ratios of 0.928, 0.932, 0.940 and 0.946 from 50M to 500M. Appendix A verifies each ratio. The ratios depend on the architecture, but processing $1/k$ as many transformer positions per raw token does not.

3.6 POOLING FUNCTIONS

Everything the transformer sees passes through the pooling function, so we vary it while holding everything else fixed. Each ablation variant compresses a group into one vector, $\bar{e} = \sum_i \alpha_i e_i$, where α_i is the normalized weight assigned to the i-th raw token embedding within the group, and leaves the rest of the pipeline untouched. **Mean** (default) uses equal weights $1/k$ and no parameters. **Fixed recency weights** take $\alpha_i \propto \exp(\gamma i)$ for $i \in \{1, \ldots, k\}$, where $\gamma = \ln(20)/(k{-}1)$, and normalize the weights so the last token receives 20 times the weight of the first. This ablation has no learned parameters. **Learned from content only** scores each embedding with one shared linear layer, $s_i = w^\top e_i + b$ for $w \in \mathbb{R}^{d_{\text{model}}}$, and sets $\alpha = \text{softmax}(s_{1..k})$ within the group, adding $d_{\text{model}} + 1 = 513$ parameters that are zero-initialized so training starts exactly at mean pooling. Critically, this scorer sees each embedding on its own, *before* the network has any positional information, so it can react to what a token is but not to where in the group it sits. **Learned from content and position** adds one learned bias p_i for each within-group position $i \in \{1, \ldots, k\}$, $s_i = w^\top e_i + b + p_i$ with $p \in \mathbb{R}^k$, for $d_{\text{model}} + 1 + k$ parameters; unlike b, the p_i differ across within-group positions and so do not cancel in the softmax. Thus, p learns a fixed profile over the token positions within each group that in practice favours later positions. Because the score is additive in content and position, this pooling function cannot model interactions between them; it is position-aware but not content–position interactive. Two ablation variants test other hypotheses: **overlapping groups** average $w{=}4$ tokens with stride $s{=}2$, giving $2\times$ compression with no token on a hard boundary, and **word-aligned groups** have variable size and end only at word boundaries (Appendices E and F).

Table 1: Main equal-compute comparison in the more-data configuration (`seq_len` 1024). Train FLOPs follow Eq. (2); full-position NLL is used for endpoint comparisons across k. Appendix B gives the equal-loss calculation and the logged metric available at each scale.

				Loss (nats/token)				
Scale	Model	Raw tokens	Train FLOPs	Native	Full-pos.	PPL	vs. base	Saving
50M	k=1	1.0B	3.50×10^{17}	4.324	4.324	75.5	–	
50M	k=2	2.0B	3.24×10^{17}	4.279	**4.275**	**71.9**	0.928×	1.30×
125M	k=1	2.5B	2.08×10^{18}	4.168	4.177	65.1	–	
125M	k=2	5.0B	1.94×10^{18}	4.135	**4.141**	**62.9**	0.932×	1.28×
250M	k=1	5.0B	8.34×10^{18}	3.380	3.412	30.3	–	
250M	k=2	10.0B	7.84×10^{18}	3.362	**3.401**	**30.0**	0.940×	1.17×
500M	k=1	10.0B	3.22×10^{19}	3.073	3.068	21.49	–	
500M	k=2	20.0B	3.05×10^{19}	3.075	3.069	21.52	0.946×	1.08×

4 Experimental Setup

Models, data, and optimization. All models are GPT-NeoX-style decoder-only transformers: pre-norm blocks, rotary position embeddings, SwiGLU FFN with expansion factor 2.5, tied input and output embeddings, and a padded vocabulary of 50,304 (tokenizer vocabulary 50,257). Apart from the pooling step of Figure 1, the averaged models are architecturally identical to the baselines. We use four scales, 50.9M, 123.5M, 252.8M and 479.3M parameters, at widths $d_{\text{model}} \in \{512, 768, 1024, 1280\}$ with 8, 12, 16 and 22 layers and peak learning rates from 2.0×10^{-4} down to 1.2×10^{-4} (Table 7). All models train from scratch on FineWeb, tokenized with the EleutherAI Pythia-70m (GPT-NeoX) BPE tokenizer. The 50M and 125M runs draw from `sample-10BT`; the 250M and 500M pairs do not fit in that slice, so those four runs draw from `sample-100BT`, changed for both models in a pair together so that no within-pair comparison is affected. The token stream is packed into fixed-length sequences without document-boundary masking, which is standard practice and matters for the context experiments, and evaluation uses a held-out split of the same distribution, identical for all models in a comparison. We train with AdamW under a cosine schedule (2,000 warmup steps, decay to 10% of peak) in bfloat16 via PyTorch DDP. At each scale the k=1 model trains on about $20N$ raw tokens, following the compute-optimal heuristic $D^* \approx 20N$ of Hoffmann et al. (2022), and the k=2 model on twice that. The two models therefore process the same total number of transformer positions and predictions. By Eq. (2), their linear compute terms are equal, while the k=2 model uses slightly fewer total FLOPs because its transformer sequences are shorter. Per-run batch sizes and step counts are in Appendix G.

Metrics. We report two evaluation metrics. **Native eval loss** is cross-entropy in nats per predicted token on held-out data, at whatever positions the model natively predicts (offset 0); it is what training logs, and we use it for training curves and for comparing runs that share the same k and grouping scheme. **Full-position NLL** is the offset-ensemble metric of Section 3.3; it is used for endpoint comparisons across k. We evaluate final checkpoints under both. Equal-loss interpolation uses full-position training curves where they were logged and native curves otherwise; Appendix B specifies the metric and calculation at each scale.

5 Results

5.1 Equal-compute comparison across scale

For each scale (50M, 125M, 250M and 500M), we train a k=1 baseline and a k=2 model to approximately equal compute, with the k=2 model using twice as many raw tokens. Table 1 reports their endpoint losses, training FLOPs and equal-loss compute ratios.

At 50M, 125M and 250M, the k=2 model reduces full-position NLL by 0.049, 0.035 and 0.011 nats, respectively. These differences correspond to 4.8%, 3.4% and 1.1% lower perplexity, with

Table 2: The 2×2 context experiment at 50M. All configurations use 16,384 raw tokens per update and the same number of updates. FLOPs are relative to the k=1/1024 baseline; the k=2/2048 configuration is iso-FLOPs with that baseline.

Configuration	Raw ctx	L	Raw tokens	Train FLOPs	Full-position loss
k=1 baseline	1024	1024	1.00B	3.50×10^{17} (1.000×)	4.324
k=1 full attention	2048	2048	1.00B	4.00×10^{17} (1.144×)	4.316 (−0.008)
More data, k=2	1024	512	2.00B	3.24×10^{17} (0.928×)	4.275 (−0.049)
More context, k=2	2048	1024	2.00B	3.50×10^{17} (1.000×)	**4.249** (−0.075)

training-cost ratios of 0.928, 0.932 and 0.940. At 500M, the losses are 3.068 for the baseline and 3.069 for k=2; the latter uses 20B rather than 10B raw tokens and 0.946× the training FLOPs.

Interpolating the loss–FLOPs curves at the baseline's final loss gives equal-loss compute ratios of 1.30×, 1.28×, 1.17× and 1.08× from 50M to 500M. Appendix B gives the calculation. At fixed FLOPs, the k=2 training curves have lower loss; at fixed raw-token count, they have slightly higher loss (Appendix D). Per-offset losses across the four model scales are reported in Appendix C.

5.2 Effects of additional data and context

To separate the effects of additional data and context, we run a 2×2 experiment at 50M with all combinations of $k \in \{1, 2\}$ and raw sequence length in $\{1024, 2048\}$. We define the k=1/1024 configuration as the baseline. All four configurations use 16,384 raw tokens per optimizer update and the same number of updates. The k=1/2048 configuration increases the standard attention window to 2048 positions and costs 1.144× the baseline. The k=2/2048 configuration compresses 2048 raw tokens into 1024 transformer positions, so it doubles both the data and raw context at the same FLOPs as the baseline.

At the baseline's compute, the k=2/2048 model has loss 4.249, compared with 4.324 for the baseline, a difference of 0.075 nats. The k=2/1024 configuration accounts for 0.049 nats of this reduction while using 0.928× the compute. Increasing its raw context from 1024 to 2048 accounts for the remaining 0.026 nats and raises its compute to the baseline level.

For k=1, increasing the context through a wider attention window reduces loss by 0.008 nats and increases compute by 14.4%. For k=2, increasing raw context reduces loss by 0.026 nats and increases compute by 7.8%. Thus, in this experiment, additional data produces a larger loss reduction than additional context. The context effects may be limited by the data: many FineWeb documents are shorter than 1024 tokens, and packed sequences cross document boundaries without masking.

At the same 2048-token raw context, the k=2 model has loss 4.249 compared with 4.316 for full attention.

5.3 Comparison of pooling functions

We compare fourteen 50M runs in the more-data configuration: six at k=2 and four each at k=4 and k=8. Within each k, the runs use the same architecture, data, batch size, optimization schedule and token budget (2.00B, 4.07B and 8.14B raw tokens, respectively). The protocol fixes raw tokens per update at 16,384, so the number of predictions per update decreases as k increases. Results at different values of k therefore compare complete training configurations. The word-aligned variant has a different target distribution and realized compression ratio, so we report it separately (Appendix F).

Content-and-position pooling. Content-and-position pooling reduces NLL relative to mean pooling by 0.043 nats at k=2, 0.286 at k=4 and 0.721 at k=8. It also has the lowest native CE among the tested functions at each quarter-budget checkpoint (Appendix E) and adds only $d_{\text{model}} + 1 + k$ parameters. Its NLL decreases from 4.315 to 4.154 to 4.093 as k increases, while training compute remains within 10% across these runs. Mean-pooling NLL instead increases by 0.456 nats from k=2 to k=8. Each table entry is one 50M run, so these trends across values of k require confirmation at other scales and seeds.

Table 3: Pooling functions at 50M in the more-data configuration. Each entry is one run at one seed. Full-position NLL is reported for comparisons across values of k. The word-aligned row uses a different prediction task (Appendix F); Appendix E reports intermediate checkpoints.

				Full-position	
k	Pooling	Extra params	Native CE	NLL	PPL
2	**learned: content + position**	$d_{\text{model}}+1+k$	**4.295**	**4.315**	**74.8**
2	mean	0	4.331	4.358	78.1
2	learned: content only	$d_{\text{model}}+1$	4.342	4.366	78.7
2	overlapping groups (w=4, s=2)	0	4.461	4.488	88.9
2	word-aligned (task differs)	0	4.548	4.575	97.0
2	fixed recency	0	4.590	4.598	99.3
4	**learned: content + position**	$d_{\text{model}}+1+k$	**4.114**	**4.154**	**63.7**
4	fixed recency	0	4.214	4.251	70.2
4	learned: content only	$d_{\text{model}}+1$	4.402	4.435	84.4
4	mean	0	4.396	4.440	84.8
8	**learned: content + position**	$d_{\text{model}}+1+k$	**4.107**	**4.093**	**59.9**
8	fixed recency	0	4.136	4.122	61.7
8	learned: content only	$d_{\text{model}}+1$	4.805	4.810	122.8
8	mean	0	4.815	4.814	123.2

Role of within-group position. The content-only scorer remains within 0.011 nats of mean pooling at every tested compression ratio. Fixed recency weighting gives 4.214 rather than 4.396 at k=4, and 4.136 rather than 4.815 at k=8; the latter is within 0.029 nats of content-and-position pooling. At k=8, the two position-aware methods are approximately 0.68 nats below the two position-blind methods. These results show lower observed loss for position-dependent weighting at larger k, while content-only weighting remains close to mean pooling. The overlapping variant also has higher loss than mean pooling. The word-aligned variant has higher observed loss, but uses a different prediction task and is not directly comparable.

6 Conclusion and Limitations

Token averaging allows a transformer to process k times more raw text through k times fewer positions. At k=2, the models train on twice the data with 5–7% fewer FLOPs. Perplexity is 4.8%, 3.4% and 1.1% lower at 50M, 125M and 250M, respectively, and comparable at 500M; the corresponding equal-loss compute ratios decrease from $1.30\times$ to $1.08\times$ with scale. In the 50M context experiment, additional data accounts for 0.049 nats of loss reduction and additional context for 0.026 nats. At equal 2048-token raw context, token averaging reduces loss by 0.067 nats with $0.874\times$ the training FLOPs, half the KV-cache entries and one-quarter of the attention cost. Content-and-position pooling reduces NLL relative to mean pooling by 0.043 nats at k=2 and 0.721 nats at k=8.

These results have three limitations. The equal-loss saving shrinks with scale, from $1.30\times$ at 50M to $1.08\times$ at 500M, and we do not extrapolate that trend beyond 500M, where each condition is a single run. The experiments do not include a coarser-tokenizer baseline in bits-per-byte or downstream zero-shot evaluation; all reported results are language-modeling losses. Finally, the context effects are small on packed FineWeb at 50M and may differ on long-document corpora or code.

Ethics Statement

This work studies the pretraining efficiency of small language models (50M–500M parameters) on FineWeb, a public web corpus, and involves no human subjects, no personally identifying data collection, and no model release capable of new harmful capabilities. The corpus inherits the biases and quality issues of open web text, and we apply no additional filtering beyond what the FineWeb release provides, so models trained as described here should not be deployed without such filtering. Because token averaging reduces pretraining cost, the efficiency gains it offers are dual-use in the ordinary sense that they lower the cost of training language models in general. We see no concerns under the ICLR Code of Ethics beyond those that apply to language-model pretraining generally.

REPRODUCIBILITY STATEMENT

The method is fully specified by Eq. (1) and Section 3: pooling, the training objective and target assignment, random-offset training, and the offset-ensemble evaluation protocol, with the pooling functions and their initializations detailed in Appendix E and the word-aligned grouping algorithm in Appendix F. All compute figures follow the single closed-form cost model of Eq. (2), derived term by term in Appendix A; every FLOPs number in the paper is recomputed from logged `tokens_seen` through that equation rather than read from a training-loop counter. The equal-loss reading procedure is written out as a three-step recipe in Appendix B, together with the target loss, both compute values, and the resulting multiplier for every pair. Model configurations, token budgets, and learning rates are in Table 7; the remaining hyperparameters, per-run raw tokens per optimizer update, and evaluation losses for every run in the paper are in Appendix G. All models train from scratch on named public FineWeb slices (`sample-10BT` and `sample-100BT`) with the public EleutherAI Pythia-70m tokenizer.

AI USE STATEMENT

We used generative AI tools as a general-purpose assistant for writing and editing prose in this manuscript, for routine coding assistance in the training, evaluation, and plotting scripts, and for retrieving and summarizing candidate related work. We did not use generative AI tools to generate research ideas, the method, the experimental design, or any reported result; every number in this paper comes from our own training and evaluation runs. All AI-assisted text was reviewed and revised by the authors, all AI-assisted code was read and tested by the authors before use, and every citation surfaced with AI assistance was verified against the primary source. We take responsibility for the final content of this work, including all text, claims, and artifacts.

REFERENCES

Zihang Dai, Guokun Lai, Yiming Yang, and Quoc V. Le. Funnel-transformer: Filtering out sequential redundancy for efficient language processing. In *Advances in Neural Information Processing Systems*, volume 33, pp. 4271–4282, 2020. URL https://arxiv.org/abs/2006.03236.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Roziere, David Lopez-Paz, and Gabriel Synnaeve. Better & faster large language models via multi-token prediction. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 15706–15734, 2024. URL https://proceedings.mlr.press/v235/gloeckle24a.html.

Namgyu Ho, Sangmin Bae, Taehyeon Kim, Hyunjik Jo, Yireun Kim, Tal Schuster, Adam Fisch, James Thorne, and Se-Young Yun. Block transformer: Global-to-local language modeling for fast inference. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/579b5b84311e122584dedab1d3b7613f-Abstract-Conference.html.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karén Simonyan, Erich Elsen, Oriol Vinyals, Jack W. Rae, and Laurent Sifre. Training compute-optimal large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 30016–30030, 2022. URL https://arxiv.org/abs/2203.15556.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. URL https://arxiv.org/abs/2001.08361.

Tomasz Limisiewicz, Artidoro Pagnoni, Srini Iyer, Mike Lewis, Sachin Mehta, Alisa Liu, Margaret Li, Gargi Ghosh, and Luke Zettlemoyer. Compute optimal tokenization. *arXiv preprint arXiv:2605.01188*, 2026. URL https://arxiv.org/abs/2605.01188.

Niklas Muennighoff, Alexander M. Rush, Boaz Barak, Teven Le Scao, Aleksandra Piktus, Nouamane Tazi, Sampo Pyysalo, Thomas Wolf, and Colin Raffel. Scaling data-constrained language models. *Journal of Machine Learning Research*, 26(53):1–66, 2025. URL https://www.jmlr.org/papers/v26/24-1000.html.

Artidoro Pagnoni, Ramakanth Pasunuru, Pedro Rodriguez, John Nguyen, Benjamin Muller, Margaret Li, Chunting Zhou, Lili Yu, Jason E. Weston, Luke Zettlemoyer, Gargi Ghosh, Mike Lewis, Ari Holtzman, and Srini Iyer. Byte latent transformer: Patches scale better than tokens. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, pp. 9238–9258, 2025. doi: 10.18653/v1/2025.acl-long.453. URL https://aclanthology.org/2025.acl-long.453/.

Bowen Peng, Théo Gigant, and Jeffrey Quesnelle. Efficient pre-training with token superposition. *arXiv preprint arXiv:2605.06546*, 2026. URL https://arxiv.org/abs/2605.06546.

Chenze Shao, Fandong Meng, and Jie Zhou. Beyond next token prediction: Patch-level training for large language models. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2407.12665.

Kevin Slagle. SpaceByte: Towards deleting tokenization from large language modeling. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://arxiv.org/abs/2404.14408.

Wenhui Tan, Minghao Li, Xiaoqian Ma, Siqi Fan, Xiusheng Huang, Liujie Zhang, Ruihua Song, and Weihang Chen. Pair-in, pair-out: Latent multi-token prediction for efficient LLMs. *arXiv preprint arXiv:2605.27255*, 2026. URL https://arxiv.org/abs/2605.27255.

Chaofan Tao, Qian Liu, Longxu Dou, Niklas Muennighoff, Zhongwei Wan, Ping Luo, Min Lin, and Ngai Wong. Scaling laws with vocabulary: Larger models deserve larger vocabularies. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://arxiv.org/abs/2407.13623.

Yi Tay, Vinh Q. Tran, Sebastian Ruder, Jai Gupta, Hyung Won Chung, Dara Bahri, Zhen Qin, Simon Baumgartner, Cong Yu, and Donald Metzler. Charformer: Fast character transformers via gradient-based subword tokenization. In *International Conference on Learning Representations*, 2022. URL https://arxiv.org/abs/2106.12672.

Zihao Xu, John Harvill, Ziwei Fan, Yizhou Sun, Hao Ding, and Hao Wang. Compressing sequences in the latent embedding space: K-token merging for large language models. *arXiv preprint arXiv:2604.15153*, 2026. URL https://arxiv.org/abs/2604.15153.

Lili Yu, Dániel Simig, Colin Flaherty, Armen Aghajanyan, Luke Zettlemoyer, and Mike Lewis. MEGABYTE: Predicting million-byte sequences with multiscale transformers. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://arxiv.org/abs/2305.07185.

A FLOPs Derivation

Per layer and per transformer position, counting multiply–accumulates as 2 FLOPs, the forward pass costs:

- Attention projections: Q, K, V, and output, i.e. four $d \times d$ matmuls: $4 \times 2d^2 = 8d^2$.
- SwiGLU FFN with `ff_multiplier` 2.5: the combined implementation projects $d \to 5d$ for up and gate ($2 \cdot 5d^2 = 10d^2$), then projects down $2.5d \to d$ ($2 \times 2.5d^2 = 5d^2$), for $15d^2$ in total. (A standard $4\times$ FFN would give $16d^2$; we use our architecture's exact value.)
- Attention scores and value aggregation: $2 \cdot 2Ld = 4Ld$ per position at sequence length L. Causal masking halves the effective term; we follow the standard convention of quoting the full-matrix count and apply it uniformly to all configurations, so ratios are unaffected.

- Output projection: one $d \to V$ matmul per scored position, $2dV$, with $V = 50{,}304$. Each transformer position emits exactly one prediction in both models, so this term is proportional to L and not to `seq_len`.

Total forward: $n_{\text{layers}}(23d^2 + 4Ld) + 2dV$ per position. Training multiplies by 3 (forward plus backward). A sequence of `seq_len` raw tokens occupies $L = \texttt{seq_len}/k$ positions, which gives Eq. (2). Every plot and table in this paper computes FLOPs from logged `tokens_seen` through that equation.

The output projection is a large share of the total at these scales rather than a correction term, so we tabulate it explicitly. As a fraction of the baseline model's total training FLOPs it is 44.2% at 50M (d=512), 27.8% at 125M (d=768), 18.5% at 250M (d=1024) and 12.0% at 500M (d=1280), falling as d grows towards V. Both models score one token per transformer position, and the equal-compute design gives them equal position counts, so in absolute terms this term contributes *equally* to both. Including it therefore moves every efficiency ratio towards 1: the final cost ratios are $0.928/0.932/0.940$ with the head and would be $0.871/0.906/0.926$ without it, and the equal-loss multipliers are $1.30/1.28/1.17$ with the head and $1.39/1.32/1.19$ without. We report ratios that include the output projection because it is part of total training compute.

We exclude input embedding lookups for both models. The lookup is a gather rather than a matmul, and the averaged model performs $k\times$ as many of them per transformer position, so leaving it out is the conservative choice for the method. Figure 2 shows how the composition of the transformer-body cost varies with width.

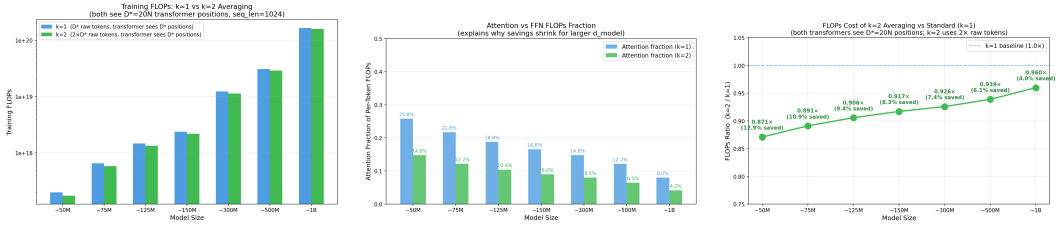


Figure 2: Structure of the transformer-body FLOPs across model widths (the output projection of Eq. (2) is excluded from these three panels, which are about the body's internal composition). **Left:** absolute body FLOPs per raw token. **Middle:** the attention share of per-position body compute shrinks with d_{model}. **Right:** the per-token body FLOPs ratio of k=2 averaging to baseline at $2\times$ the data, across widths. The $k\times$ data-throughput ratio does not depend on width. The relative attention reduction is largest at small widths, so the final cost ratio in Table 1 approaches 1 as width increases.

B Equal-Loss Compute Calculation

The equal-loss savings quoted in Section 5.1 are measured from the logged curves, without extrapolation. The procedure for one $(k{=}1, k{=}2)$ pair is:

1. Take the target loss L^* to be the baseline's minimum logged eval loss and C_{base} to be that run's final training FLOPs.
2. Find the first logged evaluation of the averaged run whose loss is at or below L^*. Interpolate linearly between that point and the preceding one in $(\log C, \text{loss})$ space to obtain C_{avg}.
3. Report $C_{\text{base}}/C_{\text{avg}}$.

FLOPs are recomputed from `tokens_seen` through Eq. (2) in both runs; the `cumulative_flops` column written by the training loop is ignored. Table 4 gives the inputs and the result at each scale.

At 500M, the baseline reaches full-position loss 3.073 at 3.22×10^{19} FLOPs and the averaged curve reaches the same value at 2.98×10^{19} FLOPs, which gives the reported $1.08\times$ equal-loss saving. Final-checkpoint full-position evaluation is reported in Table 1; both models use the same evaluation protocol within each comparison.

Table 4: Inputs to each equal-loss reading. L^* is the baseline's best logged validation loss. C_{avg} is where the averaged validation curve crosses it. Full-position curves are used at 50M and 500M; native curves are used at 125M and 250M. The more-context row is the k=1/2048 versus k=2/2048 pair of Section 5.2.

Scale	Metric	L^*	C_{base}	C_{avg}	Saving
50M	full-position	4.324	3.50×10^{17}	2.68×10^{17}	1.30×
125M	native	4.168	2.08×10^{18}	1.63×10^{18}	1.28×
250M	native	3.380	8.34×10^{18}	7.12×10^{18}	1.17×
500M	full-position	3.073	3.22×10^{19}	2.98×10^{19}	1.08×
More-context pair (2048 raw tokens in both models)					
50M	full-position	4.316	4.00×10^{17}	2.75×10^{17}	1.45×

C Offset-Ensemble Evaluation and Rotating-Offset Generation

For a k-averaged model and evaluation sequence $t_{1..T}$, offset $o \in \{0, \ldots, k-1\}$ discards the first o tokens, forms $n_o = \lfloor (T-o)/k \rfloor$ groups, embeds and pools them, and runs the transformer on the first $n_o - 1$ compressed positions. The prediction at compressed position j (1-indexed) is scored against the raw token at position $o + jk + 1$. Across offsets, the sets of predicted positions never overlap, and together they cover every position $> k$ up to edge effects at the end of the sequence, each predicted exactly once from its full compressed prefix. We compute mean NLL by pooling all scored predictions across offsets and sequences. For the restricted baseline, we gather the k=1 model's per-position NLLs on exactly this set of positions. The total cost is k forward passes at length $\approx T/k$, which is about one baseline pass at length T; the attention part is in fact cheaper by a factor k.

Table 5: Per-offset NLL for the main k=2 models. The 50M offset-1 value is recovered from the logged prediction-count-weighted NLL at the checkpoint reported in Table 1; the other rows are direct per-offset rescores. The largest offset difference is 0.008 nats.

Scale	Offset 0	Offset 1	Absolute difference
50M	4.279	4.271	0.008
125M	4.145	4.138	0.007
250M	3.404	3.398	0.006
500M	3.071	3.066	0.005

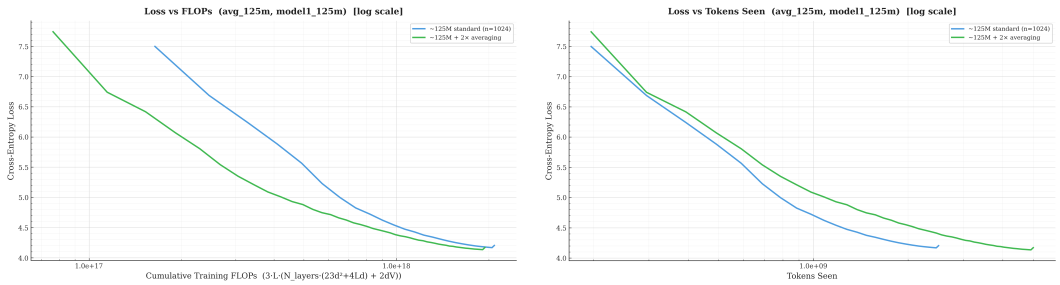
Generation rotates the offset. To emit the token at raw position p+1, the decoder uses the offset $o \equiv p \bmod k$, which is the one whose group boundaries end exactly at p. Keeping k caches, one per offset, amortizes to about one ordinary decoder's cost, because each cache is $1/k$ the length. We have not benchmarked decoding throughput.

D Additional Training Curves

Figure 3 shows the 125M pair, Figures 4 and 5 the 250M and 500M more-data pairs, Figure 6 the 2×2 context experiment, and Figure 7 the pooling comparison. In every more-data pair the k=2 curve sits below the baseline on the compute axis and slightly above it on the raw-token axis.

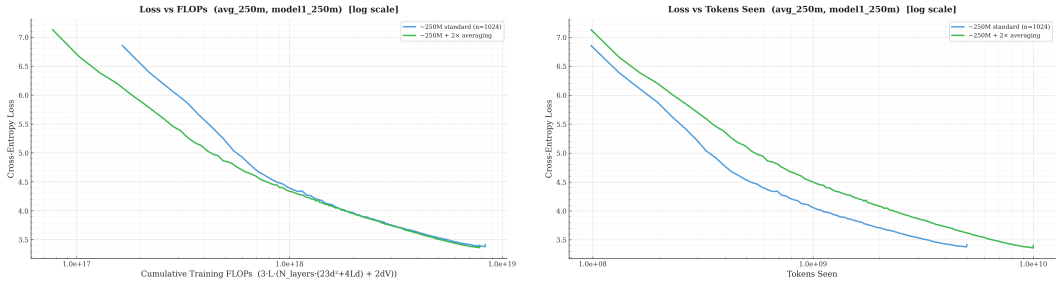
E Pooling Modules and Quarter-Budget Checkpoints

Learned from content only. Shared scorer $s_i = w^\top e_i + b$ with $w \in \mathbb{R}^{d_{\text{model}}}$ and $b \in \mathbb{R}$, zero-initialized so that $\alpha_i = 1/k$ exactly at initialization and training starts at mean pooling; $\alpha = \text{softmax}(s_{1..k})$ within each group; $\bar{e} = \sum \alpha_i e_i$. Parameters: $d_{\text{model}} + 1 = 513$. Note that b is shared across all within-group positions, and softmax is unchanged by adding a constant to all its inputs, so b has no effect on the weights. The pooling function trains jointly with the language model under the ordinary objective, with no auxiliary reconstruction loss.



(a) Loss vs. training FLOPs (log x). (b) Loss vs. raw tokens seen (log x).

Figure 3: 125M comparison, native eval losses. On the FLOPs axis (left) the k=2 curve has lower loss at equal FLOPs. On the raw-token axis (right), it has slightly higher loss at equal numbers of raw tokens. Endpoint results are in Table 1.



(a) Loss vs. training FLOPs (log x). (b) Loss vs. raw tokens seen (log x).

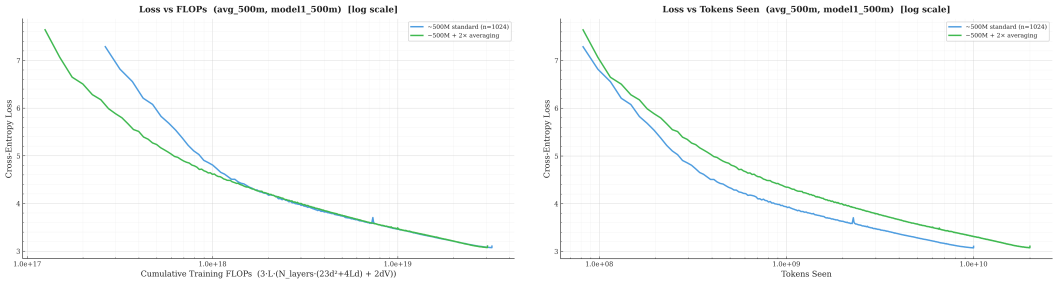
Figure 4: 250M comparison on native eval losses, both models at their full budgets (5B and 10B raw tokens). The k=2 curve has lower loss at equal FLOPs and slightly higher loss at equal numbers of raw tokens.

Learned from content and position. $s_i = w^\top e_i + b + p_i$, with $p \in \mathbb{R}^k$ containing one learned bias for each within-group position. Initialization: $p = 0$ (uniform) at k=2, where the recency prior hurt, and $p_i \propto$ the exponential profile with last-to-first ratio 20 at $k \geq 4$, where it helped. Parameters: $d_{\text{model}} + 1 + k$. Both terms train end-to-end.

Fixed weighting schemes. $w_i \propto \exp(\alpha i)$ with $\alpha = \ln(20)/(k-1)$, which is the exponential rule we trained, plus linear, Gaussian, and triangular alternatives that we implemented but did not train at scale. All schemes are L_1-normalized.

Position effects across compression ratios. One possible explanation for Table 3 follows from the pooling function's design. The content scorer processes each embedding before receiving positional information, so it cannot encode a preference for later within-group positions. Such a preference may be more useful when four or eight tokens are compressed because the final token in each group is adjacent to the prediction target. At k=2, however, fixed recency has loss 4.590, compared with 4.331 for mean pooling, indicating that strongly down-weighting the first token is detrimental in this setting. The learned content-and-position pooling function can adjust the strength of the recency preference during training. Mean pooling ranks second at k=2, third among the four common methods at k=4, and fourth at k=8; fixed recency shows the reverse trend.

Initialization confound at $k \geq 4$. At $k \geq 4$ we initialize the content-plus-position pooler's position logits *at* the exponential recency rule, while the content-only scorer is zero-initialized and starts at mean pooling. This initialization difference may contribute to the separation between the two learned methods at k=4 and k=8; a matched-initialization experiment is needed to isolate their representational difference. At k=2, both methods start from uniform weights, and content-and-position pooling has lower loss. The fixed-weighting methods do not depend on initialization.



(a) Loss vs. training FLOPs (log x). (b) Loss vs. raw tokens seen (log x).

Figure 5: 500M comparison on native eval losses, both models at their full budgets (10B and 20B raw tokens). The two FLOPs-axis curves are indistinguishable over the final decade of training and finish a few thousandths of a nat apart. At equal raw-token counts, the averaged model has higher loss than the baseline.

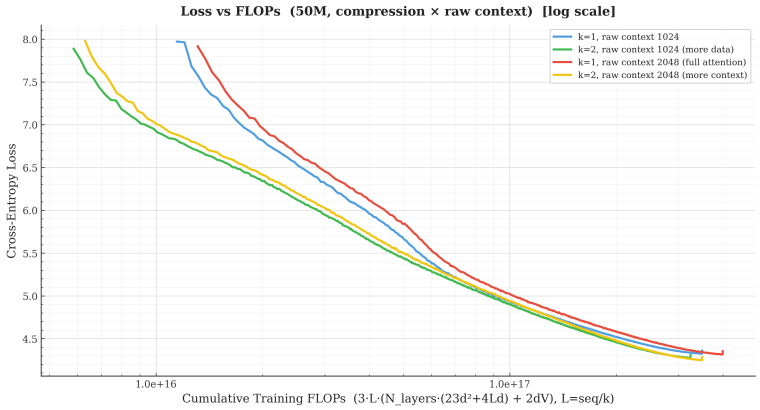


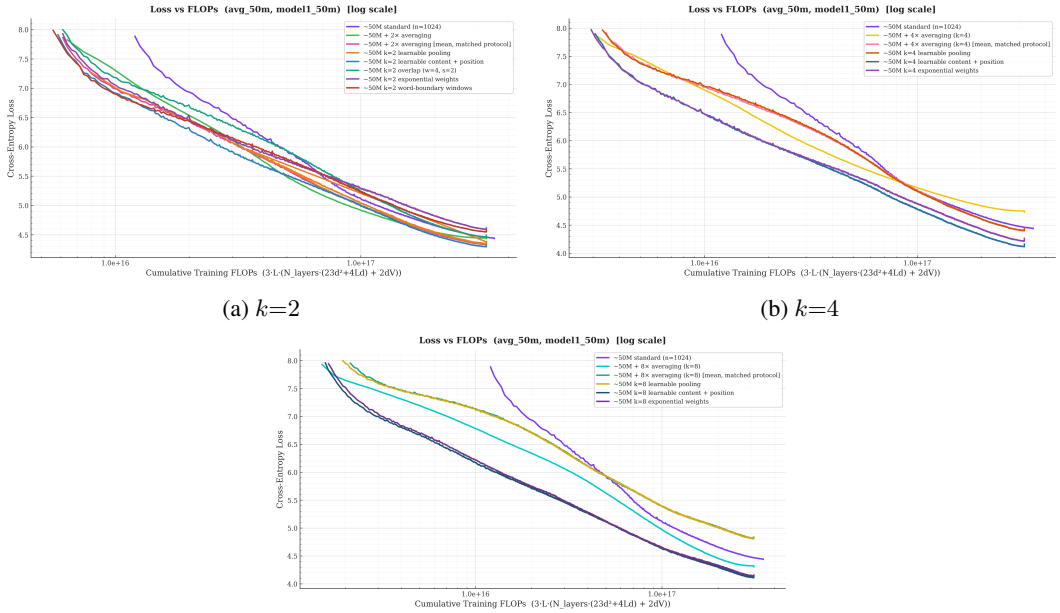
Figure 6: The 2×2 context experiment at 50M: full-position loss vs. training FLOPs (log x). The k=2/2048 configuration has the lowest endpoint loss and the same FLOPs as the k=1/1024 baseline. The k=1/2048 configuration reduces endpoint loss by 0.008 nats relative to the baseline and costs $1.144\times$ as much. Early-training loss spikes are clipped at 8.0 for readability.

Table 6 adds the native eval CE at the nearest logged checkpoint to each quarter of the token budget. Content-and-position pooling has the lowest value at each reported checkpoint.

F Word-Aligned Windows: Algorithm and Caveats

A token begins a word if its string form starts with the BPE whitespace marker (a leading Ġ in GPT-NeoX byte-level BPE) or a newline marker, or if it is a special token. We build a boolean lookup table over the padded vocabulary once from the tokenizer. Grouping is greedy within each sequence: accumulate tokens into the current group; close the group when it holds $\geq k$ tokens *and* the next token starts a new word, or unconditionally at $2k$ tokens so that an unusually long word cannot grow a group without bound; then drop any trailing partial group. Groups therefore span 2–4 tokens at k=2 and never end mid-word unless a word exceeds $2k$ tokens. On a tokenized sample of English text the mean group size is 2.125 at k=2, a realized compression of $\sim 2.1\times$. Within a batch we truncate all sequences to the smallest group count. The grouping loop costs $\sim$1ms per batch at B=16, T=1024, which is negligible. Targets are the first token of the next group, which for this variant is always the start of a word.

Two caveats follow. First, the realized compression exceeds the nominal $2\times$, so the FLOPs accounting has to use the realized ratio; the transformer sees $\sim 5\%$ fewer positions per raw token than uniform k=2. Second, the distribution of predicted positions differs, since only word starts are predicted, so this variant's native eval loss is not measured on the same prediction task as the fixed-size variants, and the offset-ensemble protocol does not directly apply to variable-size groups either. A bits-per-byte



(a) k=2 (b) k=4

(c) k=8

Figure 7: Pooling functions at 50M: native eval loss vs. recomputed training FLOPs (log x), one panel per compression ratio, each overlaid on the k=1 baseline and the mean-pooling run at that k. Curves are comparable with each other only within one panel and one protocol. The loss difference between position-aware and position-blind methods increases with k.

Table 6: Pooling functions at 50M: native eval CE (nats/token) at the nearest logged checkpoint to each quarter of the token budget, then the best native value. Comparable within a k block only. Full-position rescores of the final checkpoints are in Table 3.

k	Pooling	25%	50%	75%	Best
2	learned: content + position	5.170	4.619	4.389	**4.295**
2	mean	5.174	4.649	4.425	4.331
2	learned: content only	5.222	4.656	4.443	4.342
2	overlapping groups (w=4, s=2)	5.460	4.820	4.562	4.461
2	word-aligned (task differs)	5.413	4.872	4.640	4.548
2	fixed recency	5.439	4.962	4.707	4.590
4	learned: content + position	4.951	4.463	4.224	**4.114**
4	fixed recency	5.041	4.569	4.330	4.214
4	learned: content only	5.315	4.753	4.512	4.402
4	mean	5.326	4.756	4.505	4.396
8	learned: content + position	4.801	4.414	4.225	**4.107**
8	fixed recency	4.804	4.433	4.257	4.136
8	learned: content only	5.581	5.132	4.924	4.805
8	mean	5.587	5.135	4.932	4.815

comparison or a fixed-position rescoring is needed before making strong claims, so we present the word-aligned row as indicative only.

G Model Configurations, Run Table, and Hyperparameters

Table 7: Model configurations. "Raw tokens" are the training budgets for the k=1 / k=2 models, set so both models end at approximately equal total training FLOPs (Eq. (2)); the k=1 budgets follow the Chinchilla $D^* \approx 20N$ heuristic. All main runs use raw `seq_len` 1024, the more-data configuration.

Scale	Params	d_{model}	Heads	Layers	LR	Raw tokens (k=1 / k=2)
50M	50.9M	512	8	8	2.0×10^{-4}	1B / 2B
125M	123.5M	768	12	12	1.6×10^{-4}	2.5B / 5B
250M	252.8M	1024	16	16	1.4×10^{-4}	5B / 10B
500M	479.3M	1280	20	22	1.2×10^{-4}	10B / 20B

Table 8: Principal more-data and context runs. Eval loss is full-position NLL. All models use tied embeddings and each row is one run. The fourteen pooling runs are reported in Tables 3 and 6.

Experiment	Scale / model	k	Raw seq.	Raw tokens	Raw tok./update	Eval loss
More data	50M baseline	1	1024	1.0B	16,384	4.324
	50M averaged	2	1024	2.0B	16,384	4.275
	125M baseline	1	1024	2.5B	98,304	4.177
	125M averaged	2	1024	5.0B	98,304	4.141
	250M baseline	1	1024	5.0B	32,768	3.412
	250M averaged	2	1024	10.0B	32,768	3.401
	500M baseline	1	1024	10.0B	32,768	3.068
	500M averaged	2	1024	20.0B	32,768	3.069
Context	50M full attention	1	2048	1.0B	16,384	4.316
	50M averaged	2	2048	2.0B	16,384	4.249

Table 9: Shared hyperparameters.

Optimizer	AdamW
LR schedule	cosine, decay to 10% of peak over total steps
Warmup	2,000 steps
Peak LR	$2.0/1.6/1.4/1.2 \times 10^{-4}$ (50M/125M/250M/500M)
Precision	bfloat16 autocast
Raw `seq_len`	1024 (more data; 2048 for the more-context models)
Tokenizer	EleutherAI Pythia-70m (GPT-NeoX BPE), vocab 50,257
Vocabulary padding	50,304
FFN	SwiGLU, expansion 2.5
Positional encoding	RoPE (applied to compressed positions)
Chinchilla reference constants	A=406.4, α=0.3392, B=410.7, β=0.2849, E=1.6934