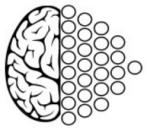
SAIDL Summer Assignment 2026

Devadarsh A Nair
f20240330@goa.bits-pilani.ac.in

May 2026



Contents

1 Overview

1.1 Problem Setting

Making language models efficient splits into two separate problems, the first part is to "build smarter architecture", where the model can handle long context, run faster without burning much memory and the second part is to "take a model someone has already trained and cheaply teach it a new task without re-training the entire model". This assignment tries to understand both these problems with the help of the first Core-ML task and the second Sparsity & Optimization task.

1.2 Assignment tasks

For the Core ML task, I built a small 18.1M-parameter decoder-only GPT-style model where you can swap out the attention type, the positional encodings and the block layout, then tested the implemented variants on the WikiText-2 dataset.

For the S&O task, I took the DeBERTa-v3-base, a 184.7M-parameter model and tried different ways to fine-tune it for judging whether a sentence is grammatically correct or not using the CoLA dataset.

1.3 Summary of Findings

Linear attention is the clearest Core ML hardware result: at sequence length 4096, it reduces peak reserved VRAM from 10605 MB (standard) to 4664 MB and raises throughput from 803 to 17475 tokens per second, making it the biggest memory win for long sequences; Sparse-Block reaches 33902 tokens per second, making it the fastest of all. (Table 16). RoPE got the lowest perplexity at 512 tokens (Table 5); ALiBi stayed flattest when tested on extrapolated lengths. (Figure 5).

The best hybrid block is best_combo (interleaved convolution-attention + sparse_block + ALiBi), with val_loss 4.952 and val_ppl 141.51 after 10 epochs (Table 6). For S&O's fine-tuning task, LoRA achieves the highest MCC in this single-seed run (0.6848), whereas AdaLoRA (0.6782) and SoRA proximal (0.6777) are very close, within CoLA seed-to-seed variance (AdaLoRA ±0.0163 MCC standard deviation), so they are basically tied when you're accounting for run-to-run randomness.

2 Core Machine Learning

2.1 Objective

The Core ML task asks how Transformer architecture choices affect language-model quality, performance and hardware consumption. The experiment isolates 3 key components, studying them one at a time: the attention mechanism, the positional encodings and the block structure.

2.2 Transformer Background

The model created is a causal Transformer, which is just a fancy way of saying that it reads text left to right and never peeks ahead. It starts with a stack of token embeddings $X \in \mathbb{R}^{n \times d}$, one row per token, and each row being a vector, and turns them into "context-aware" versions of themselves by letting each token read earlier tokens.

The way each attention head does this is by projecting those embeddings into three different roles:

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V,$$

where Q asks what the current token needs, K describes what each earlier token offers, and V carries the information to mix. For this model, the hidden size is split evenly across the 8 heads, so each head works in a small subspace of size $d_k = d_{\text{model}} \div \text{num\_heads} = 256 \div 8 = 32$.

To decide how much each token should pay attention to each earlier one, I use Scaled dot-product attention:

$$\boxed{\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right)V},$$

where

$$M_{ij} = \begin{cases} 0, & j \le i, \\ -\infty, & j > i. \end{cases}$$

Softmax is applied row-wise, so for a given token, its attention weights add up to 1. For masking, I set future positions to $-\infty$ before softmax. Once that goes through the exponential in softmax, those future positions become a weight of exactly zero, so a token physically cannot attend to anything that comes after it.

However, here is the catch, the product $QK^\top$ creates an $n \times n$ token-token score matrix, so memory and compute grow as $O(n^2)$. This is the long-context bottleneck. Storing the full score matrix requires $O(n^2)$ memory per layer; computing it requires $O(n^2 d)$ multiply-adds. These costs motivate towards more memory-efficient alternatives like sliding-window, sparse-block, and linear attention variants [14].

Position is the second part. Attention is permutation-invariant, and has no sense of word order unless token order is added through learned embeddings, rotations, biases, or relative-distance terms.

2.3 Dataset and Training Setup

The experiments use WikiText-2 pulled through Hugging Face datasets. The model uses GPT-2 BPE tokenizer and a 50,257-token vocabulary. It is a 6-layer pre-norm decoder-only language model with $d_{\text{model}} = 256$, 8 attention heads, a feed-forward layer with $d_{\text{ff}} = 1024$, tied token and output embeddings, and a plain causal language-modeling loss.

These dimensions balance expressivity ("expressive enough to tell the variants apart") against training cost on a single consumer GPU: $d_{\text{model}} = 256$ over 6 layers separates the variants while staying fast to train; $d_{\text{ff}} = 1024$ follows the standard $4 \times d_{\text{model}}$ rule of thumb ratio; 8 heads at $d_k = 32$ respect the convention of keeping head dimension at least ≥ 32 for stable attention; pre-norm is used for training stability.

The embedding matrix $E \in \mathbb{R}^{V \times d_{\text{model}}}$ maps each token ID to a d_{model}-dimensional vector; here $E \in \mathbb{R}^{50257 \times 256}$. Input and output embeddings are tied which means the same matrix encodes tokens by turning them into vectors and projects hidden states back into the vocabulary logits coming out, saving $V \times d_{\text{model}}$ parameters. Concretely, with $V = 50{,}257$ and $d_{\text{model}} = 256$, this tying saves

$$\Delta_{\text{params}} = V \times d_{\text{model}} = 50{,}257 \times 256 \approx 12.9\text{M}$$

parameters. Without it, the 18.1M-parameter model would grow to roughly 31M, so for a small model carrying a large vocabulary the tied embedding accounts for most of the parameter budget. The sequences are split into fixed-length chunks of the target context length. One thing to keep in mind here is that a "token" here is a BPE sub-word piece, not necessarily a whole word.

Training minimizes the causal language-model loss, at each position the model predicts the next token, and I penalize how surprised the model is by the correct one:

$$\mathcal{L} = -\sum_{i=1}^{n-1} \log p(x_{i+1} \mid x_{\le i}).$$

Table 1: Core ML setup used by the current runs.

Item	Value
Dataset	WikiText-2
Tokenizer	GPT-2 BPE
Vocabulary size	50,257
Training context	Baseline 1024; positional and hybrid 512; attention diagnostic sweep 256–4096
Model type	Decoder-only causal Transformer language model
Layers	6
d_{model}	256
Attention heads	8
d_k	32
d_{ff}	1024
Tied embeddings	Yes
Causal mask	Yes
Objective	Causal LM cross-entropy
Batch size	4 (quality runs); 1 (hardware diagnostic sweep, Table 16)
Diagnostic attention steps	5
Attention quality budget	1000 steps
Positional Table 5 budget	1000 steps
Positional extrapolation Figure 5 budget	3000 steps
Hybrid Table 6 budget	10 epochs
Baseline budget	5840 steps

Perplexity is the exponentiated per-token loss. I report it because it's more interpretable, roughly "how many tokens is the model effectively choosing between at each step":

$$\text{PPL} = \exp(\mathcal{L}).$$

One important caveat for reading my tables: the attention sweep uses five diagnostic steps from random initialization; so its validation losses near 253 and capped perplexities ($> 10^6$) are not quality metrics, they exist only to measure hardware behavior. Quality and architecture rows each get their own separate budgets: attention quality and the positional Table 5 sweep use 1000 steps, positional extrapolation in Figure 5 uses 3000 steps, the hybrid comparison uses 10 epochs, and the baseline uses 5840 steps. Since these budgets differ, I never compare losses across them.

2.4 Baseline Transformer

Table 2 gives the longest Core ML run (5840 steps). I use this as a reference point rather than a direct contestant. Its VRAM and loss are not directly comparable to the 1000-step positional and hybrid sweeps or the 5-step attention hardware diagnostic, simply because each was trained for a different amount of time. For context, a model that guesses uniformly over the $V = 50{,}257$-token vocabulary has perplexity

$$\text{PPL}_{\text{uniform}} = V = 50{,}257.$$

The baseline's 1028.91 sits about 49× below this random-guess level, so the model is clearly learning the data distribution; it has simply not converged, because the 5840-step budget is a short diagnostic reference run rather than a full training schedule.

Table 2: Baseline summary for the 18.1M-parameter decoder LM on WikiText-2.

Variant	Train len	Steps	Val. loss	Val. PPL	Peak VRAM (MB)	Throughput (tokens s^{-1})
`baseline_learned`	1024	5840	6.936253	1028.91	4447.3	24207.3

2.5 Attention Variants

Standard Attention. Standard attention is the baseline mechanism I described earlier, it computes the full causal token-token score matrix. My sliding-window variant comes with an honest caveat: the way it is implemented is mask-only. I restrict each token to a local window by masking out everything outside it, but underneath I still materialize the full dense score matrix. So it does not really save peak memory. It is best read as a "local-connectivity ablation" (does limiting attention to nearby tokens hurt the quality?) rather than a memory-saving kernel.

Grouped-Query Attention (GQA) and Multi-Query Attention (MQA). GQA and MQA attack a different cost. GQA shares keys and values across groups of query heads [1]. MQA is the extreme case with one key-value head [12]. These tricks matter most for inference, where the KV-cache memory in large models dominates.

$$\text{KV cache} = \underbrace{2}_{K,\,V}\, n_{\text{kv}}\, d_k\, n \;\; \text{floats}, \qquad n_{\text{kv}} = \begin{cases} n_h & \text{(MHA)} \\ n_h/g & \text{(GQA, group size } g\text{)} \\ 1 & \text{(MQA)} \end{cases}$$

GQA and MQA reduce the number of KV heads, shrinking the KV cache stored at each decode step from $2n_h d_k n$ to $2n_{kv} d_k n$ floats. In my small training setup, they don't separate much from standard attention for two reasons: the quality impact is modest at this scale, MQA is slightly below MHA in perplexity (see the quality rows in Table 3) and during training I compute all the gradients regardless of head sharing, so the memory savings that show up at inference don't show up during training.

Linear Attention. Linear attention is the one that genuinely changes the asymptotics. It replaces the softmax kernel with a feature map ϕ: [9]

$$\text{Attention}(q_i) = \frac{\sum_{j \le i} \text{sim}(q_i, k_j)\, v_j}{\sum_{j \le i} \text{sim}(q_i, k_j)}, \qquad \text{sim}(q, k) = \exp\left(\frac{q^\top k}{\sqrt{d_k}}\right).$$

Linear attention replaces this with the factorizable kernel $\text{sim}(q, k) = \phi(q)^\top \phi(k)$, which is what lets the sums be reordered and streamed rather than materialized as an $n \times n$ matrix.

$$\text{Attention}(Q, K, V) \approx \phi(Q) \left(\phi(K)^\top V\right).$$

I use $\phi(x) = \text{elu}(x) + 1$ together with cumulative sums to keep things causal. The $+1$ offset matters: softmax weights are non-negative, so a kernel feature map must stay strictly positive for the approximated attention weights to remain non-negative. The map $\phi(x) = \text{elu}(x) + 1$ satisfies $\phi(x) > 0$ for all x while remaining smooth and inexpensive to compute. The key insight is the bracketing: by computing $\phi(K)^\top V$ first, I never build the $n \times n$ score matrix at all. This isn't exact softmax attention, it is an approximation, and specifically a special case of kernel attention. (Performer's FAVOR+ approximates the softmax kernel with random features; my ELU+1 map is instead an exact-recurrence choice that sidesteps the $n \times n$ matrix entirely.) With causal cumulative sums the cost is $O(nd^2)$ per layer, linear in sequence length instead of quadratic.

$$S_i = S_{i-1} + \phi(k_i)\, v_i^\top, \qquad z_i = z_{i-1} + \phi(k_i),$$

$$\text{output}_i = \frac{\phi(q_i)^\top S_i}{\phi(q_i)^\top z_i}, \qquad S_i \in \mathbb{R}^{d \times d},\; z_i \in \mathbb{R}^d.$$

Full per-variant peak memory, throughput, and latency across sequence lengths 256–4096 are in Appendix A.1 (Table 16); throughput scaling across context length is plotted in Figure 2.

The headline result is that linear attention has the lowest reserved VRAM and Sparse-Block the highest throughput at 4096 tokens. All rows use batch_size=1 and a 5-step diagnostic protocol. Linear attention separates in reserved memory at longer contexts because it avoids materializing the full $n \times n$ score matrix (Figure 1).

Sparse Block Attention (SBA). SBA restricts attention to non-overlapping blocks of B consecutive tokens. Each block of B tokens attends only within itself using a within-block causal mask; tokens in one block have no direct attention path to tokens in another block within the same layer. The implementation uses $B = 64$. SBA reduces score-matrix memory from $O(n^2)$ to $O(nB)$ per layer (here $B = 64$), which is why its reserved VRAM tracks linear attention at long context despite using a softmax kernel within each block. SBA results are in Table 16.

Complexity summary. Full attention is $O(n^2)$ score memory and $O(n^2 d)$ compute. Sliding-window is mask-only local attention with dense $O(n^2)$ score memory; a true banded kernel such as Longformer's [3] would be $O(nw)$, but that kernel is not implemented here. Linear attention is $O(nd^2)$ per layer via $\phi(Q)(\phi(K)^\top V)$ via factorization with cumulative sums.

The payoff shows up most clearly in latency. At 4096 tokens, standard attention latency reaches 1260 ms per step; linear attention reduces this to 49 ms and Sparse-Block to 28 ms, a 25-45× speedup that proves sub-quadratic compute benefits latency as directly as it helps memory. (Table 16). The dense variants (standard, GQA, MQA) show a throughput collapse at 4096 tokens: standard attention falls from 9,615 tokens s^{-1} at 2048 to 803 at 4096 (Table 16). This is the $O(n^2)$ wall, at 4096 the dense score matrix and its softmax dominate per-step wall-clock, so tokens per second fall even as more tokens are processed. The sub-quadratic variants avoid the $n \times n$ matrix, so their per-step cost grows closer to linearly and throughput keeps rising. All six attention variants complete 1000 training steps without NaN loss and with logged final gradient norms in the 0.586-0.642 range, comfortably below 1, confirming a stable training regime under the N(0,0.02) embedding initialization. Finally, I keep quality and hardware measurements strictly separate: Table 3 reports validation perplexity at 512 tokens from the quality runs, while Figures 3 and 4 cover longer-context behavior and loss dynamics. The six variants stay close in Table 3, so at this budget the choice of attention mechanism has only a small in-distribution quality impact.

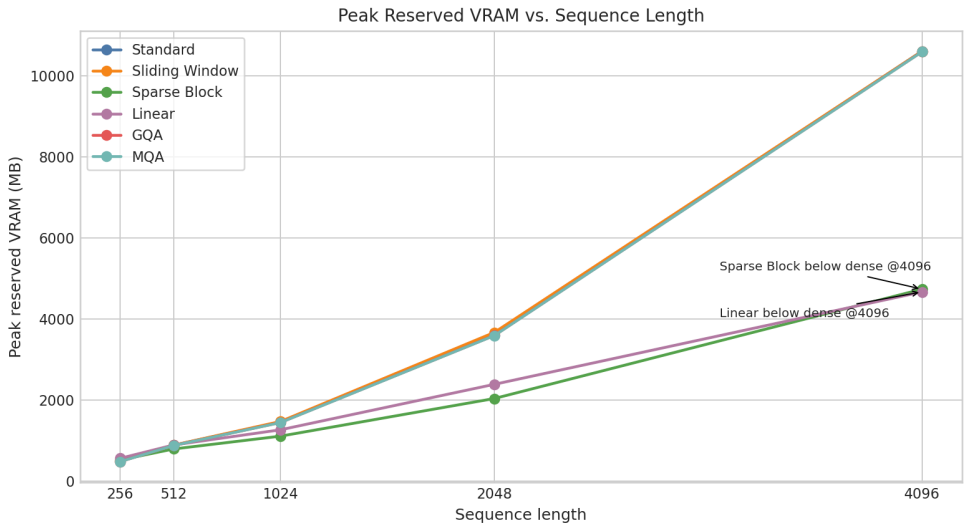


Figure 1: Peak reserved VRAM for attention variants in the clean diagnostic sweep (batch_size=1, 5 steps). Linear attention and Sparse-Block reserve less than half the VRAM of standard attention at 4096 tokens, confirming sub-quadratic memory scaling at long context.

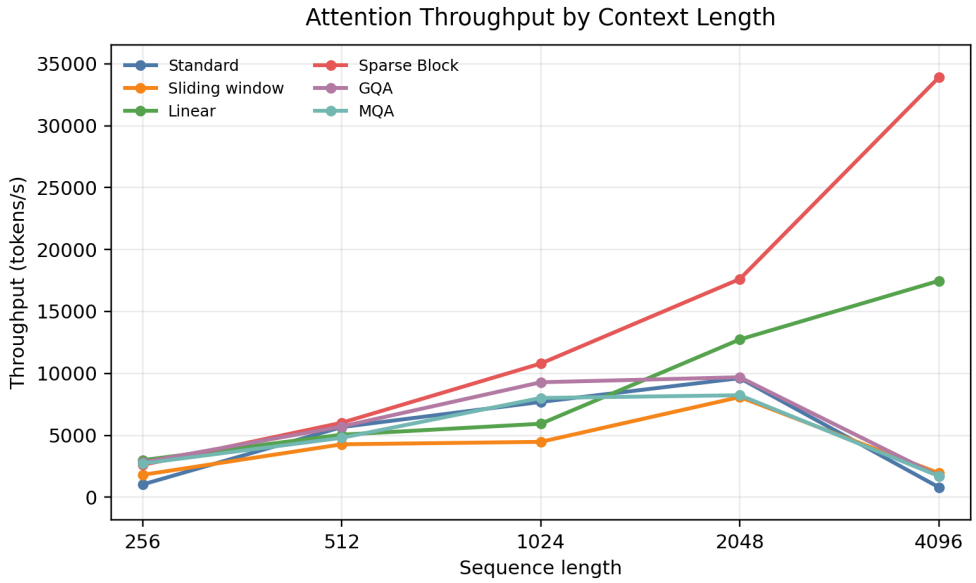


Figure 2: Throughput for attention variants in the clean diagnostic sweep. Sparse-Block reaches 33,902 tokens per second at 4096 tokens, 42× higher than standard attention, confirming that restricted score computation over non-overlapping fixed-size blocks creates the throughput gain.

Table 3: Attention-variant validation perplexity at 512 tokens (1000-step quality runs; lower is better). The six variants remain close at this budget, so attention-mechanism choice has a small quality impact in-distribution.

Variant	Validation PPL at 512
Standard	752.28
Sliding Window	749.58
Sparse Block	738.87
Linear	764.95
GQA	762.46
MQA	758.84

2.6 Positional Encodings

Attention on its own has no sense of order, so I need to feed position in somehow. I tried five different approaches, and they fall into two categories: methods that encode *absolute* position (where a token sits in the sequence) and methods that encode *relative* position (how far apart two tokens are). That distinction turns out to be the whole story for which ones survive sequences longer than they trained on.

Learned absolute. The simplest option: I keep a trainable vector for every position and add it to the token embedding. It works fine inside the training range, but it has a hard ceiling, if I trained on 512 positions, position 513 has no learned vector at all, so positions beyond the trained context have no learned history and the model is effectively flying blind past its training length.

Sinusoidal. Instead of learning position vectors, I can compute them from fixed sine and cosine waves of different frequencies, using no trainable parameters:

$$\mathrm{PE}(\mathrm{pos},\, 2i) = \sin\!\left(\frac{\mathrm{pos}}{10000^{2id_{\mathrm{model}}^{-1}}}\right), \quad \mathrm{PE}(\mathrm{pos},\, 2i+1) = \cos\!\left(\frac{\mathrm{pos}}{10000^{2id_{\mathrm{model}}^{-1}}}\right).$$

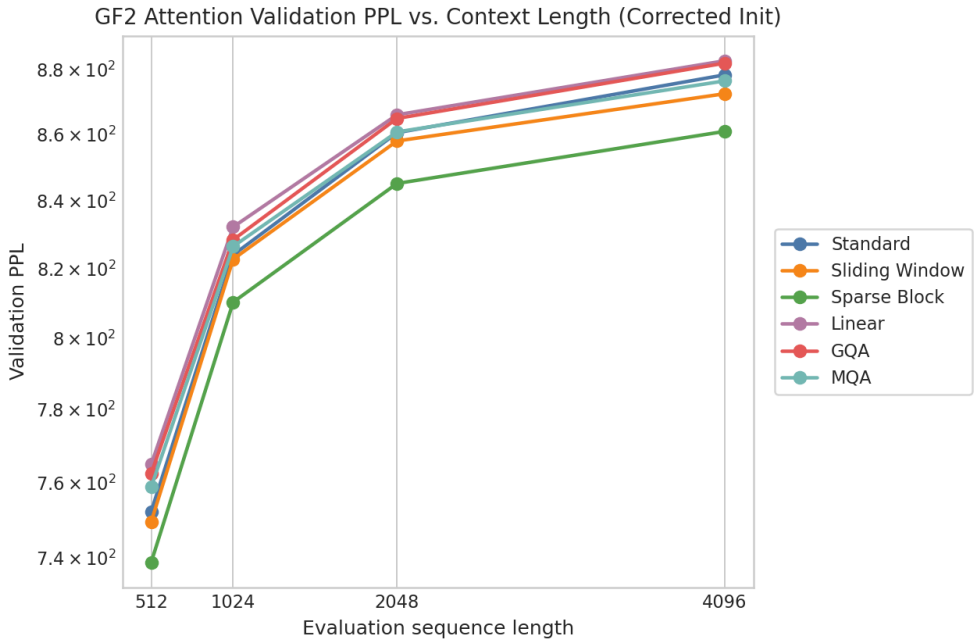


Figure 3: Attention-quality validation perplexity versus evaluation context. The plot compares long-context behavior across attention variants in the quality protocol. Perplexity rises roughly 15-17% from 512 to 4096 tokens across the six variants, a mild degradation consistent with limited length generalization at this training budget.

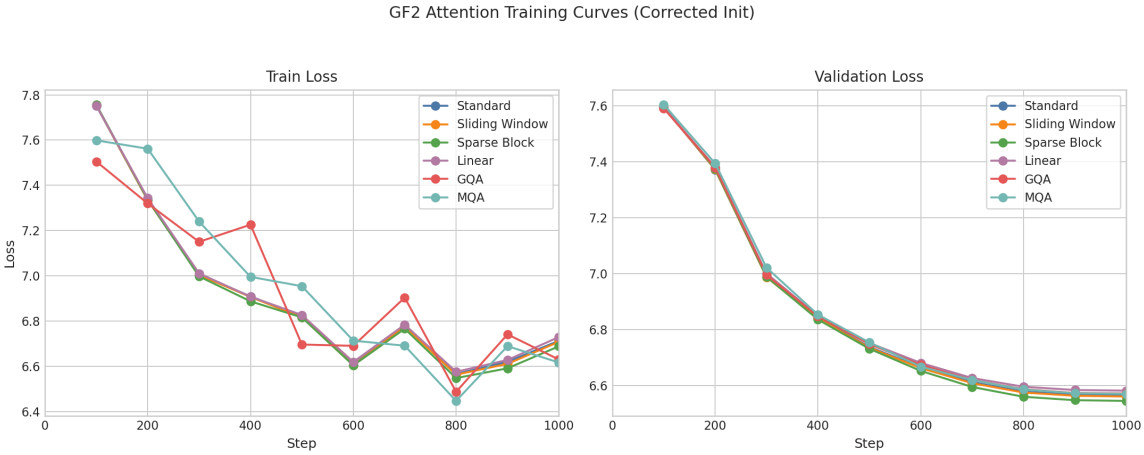


Figure 4: Attention-quality training and validation loss curves. The curves show optimization behavior for the attention variants under the quality run protocol. All six variants descend smoothly from the same N(0,0.02) initialization, confirming a consistent and comparable training regime.

Nothing is "learned" here, the encodings are registered as a buffer and are available at any position within the trained length. Different dimensions oscillate at different rates, giving the model a smooth, multi-scale notion of where it is in the sequence. A useful property of this form is that a shift by a fixed offset k acts as a *linear* map on the encoding: there exists a matrix M_k (independent of pos) with

$$\mathrm{PE}(\mathrm{pos} + k) = M_k \, \mathrm{PE}(\mathrm{pos}),$$

so even though the encoding is absolute, the network can recover relative offsets through a linear combination of dimensions.

RoPE. Rotary position embedding takes a better angle: instead of *adding* a position signal, it *rotates* each query and key by an angle that depends on its position. For one pair of hidden dimensions, position m applies

$$R_m \begin{bmatrix} x_{2i} \\ x_{2i+1} \end{bmatrix} = \begin{bmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{bmatrix} \begin{bmatrix} x_{2i} \\ x_{2i+1} \end{bmatrix},$$

with base frequency $\theta_i = 10000^{-2id^{-1}}$ for dimension pair i. The elegant part is what happens in the dot product: because I rotate *both* the query and the key, writing the rotated query and key as $R_m q$ and $R_n k$, the attention score is

$$\langle R_m\, q,\ R_n\, k \rangle = q^\top R_m^\top R_n\, k = q^\top R_{n-m}\, k,$$

since rotation matrices satisfy $R_m^\top R_n = R_{n-m}$. The absolute indices cancel, so the positional interaction in the score is expressed through the relative offset $n-m$ alongside the token contents q, k. This is why RoPE gives the attention score a relative-position form, making it better suited to inference beyond the training length provided the rotation angles remain numerically stable [13].

ALiBi. ALiBi goes even simpler: it adds no position vector to the embeddings at all. Instead it penalizes attention by distance, directly in the logits [10]:

$$\alpha_{ij} = \mathrm{softmax}\left(\frac{q_i^\top k_j}{\sqrt{d_k}} - m_h |i-j| \right),$$

where m_h is a fixed head-specific slope. The slopes are not learned: for H heads they form a fixed geometric sequence

$$m_h = 2^{-8h/H}, \qquad h = 1, 2, \ldots, H,$$

so each head penalizes distance at a different rate, some heads stay sharply local, others look further back, without adding a single trainable parameter. No positional vector is added to the token embeddings; distance enters only through the attention logit bias. The penalty $-m_h|i-j|$ embeds a recency bias without tying to an absolute position index, so it generalizes analytically to any sequence length (*Train-Short-Test-Long*). Published results show ALiBi matching or exceeding learned absolute and RoPE on out-of-distribution lengths when trained at standard budgets; the flat extrapolation in Figure 5 confirms this design intent.

Relative bias. The last option learns an additive bias indexed by clipped relative distance [11]:

$$\mathrm{score}_{ijh} = \frac{q_{ih}^\top k_{jh}}{\sqrt{d_k}} + b_{h,\mathrm{clip}(i-j,-R,R)}.$$

So instead of ALiBi's fixed slope, the model *learns* a separate bias for each relative distance. The clip bounds the table size: each head learns one bias per relative distance in $[-R, R]$, i.e. $2R+1$ learned values, and every offset beyond $\pm R$ reuses the boundary bias $b_{h,\pm R}$.

In the 1000-step positional sweep, RoPE has the lowest validation loss at the 512-token training length (6.137), followed by relative bias (6.236), sinusoidal (6.367), ALiBi (6.622), and learned-absolute right behind at 6.628. Absolute learned embeddings are valid at the 512-token training length but are OOB at 1024 and 2048 because no learned position entries exist beyond the train length. Sinusoidal, RoPE, and relative bias extend to longer lengths under the $\mathcal{N}(0, 0.02)$ initialization; relative bias stays in the 6-7 loss range. ALiBi is designed for flat extrapolation and shows no degradation; Figure 5 matches these expectations.

Table 4: Positional encoding properties. Absolute methods index where a token sits; relative methods depend only on token-token distance, which is what enables length extrapolation. L is the training context length, H the number of heads.

Method	Positional params	Encodes	Extrapolates?
Learned absolute	$L \times d_{\text{model}}$	absolute	No (OOB past L)
Sinusoidal	0 (fixed buffer)	absolute	Yes (degrades)
RoPE	0	relative (rotation)	Yes
ALiBi	0 (H fixed slopes)	relative (additive penalty)	Yes (flattest)
Relative bias	$(2R{+}1) \times H$ learned	relative (learned bias)	Yes (mild rise)

Table 5: Positional encoding comparison from the 1000-step sweep at train sequence length 512 (batch_size=4, seed=42, $\mathcal{N}(0, 0.02)$ token initialization). Lower validation loss and PPL are better. Learned absolute is OOB beyond 512 because its position table is capped at the training length. Rows cluster near the 6–7 loss scale.

Encoding	Val. loss @512	Val. loss @1024	Val. loss @2048	Val. PPL @2048
RoPE	6.137220	6.147307	6.180873	483.41
Relative bias	6.235974	6.241906	6.251771	518.93
Sinusoidal	6.367407	6.442485	6.537927	690.85
Learned abs.	6.628156	—	—	—
ALiBi	6.622417	6.619867	6.619535	749.60

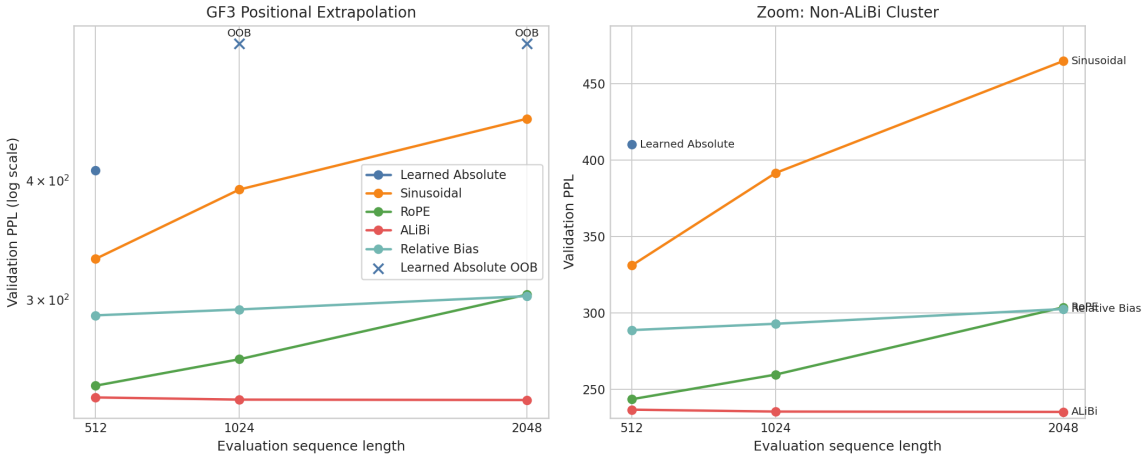


Figure 5: Train-short-test-long positional extrapolation (3000 training steps at seq_len=512, evaluated at 512, 1024, 2048). Left: all five encodings on log PPL scale; learned-absolute truncated at 512 (OOB beyond training length). Right: zoom on the non-ALiBi cluster. ALiBi is the flattest; RoPE stays low; Sinusoidal and relative-bias rise with length inside the non-ALiBi cluster; learned-absolute cannot extrapolate. Learned absolute cannot extrapolate beyond the training length (OOB at 1024); ALiBi is the flattest encoder across 512-2048, confirming its train-short-test-long design.

ALiBi convergence note. ALiBi encodes position exclusively through fixed head-specific attention penalties (no learned or sinusoidal embedding at the token level). ALiBi reaches val_loss=6.622 at 1000 steps and 5.467 at 3000 steps (Figure 5), the flattest extrapolator across all five encodings.

2.7 Hybrid Architectures

Self-attention is powerful but it's a generalist, it can look anywhere in the sequence, yet it carries only a weak built-in preference for nearby tokens. Convolutions are the opposite: they are specialists at local patterns. So the idea here is to give each block a cheap, explicit *local* path before attention does its global mixing. I do this with a depthwise 1D convolution, which acts as a learned n-gram detector, a small filter that combines each token with a few of its immediate neighbors. I use kernel size 3 with explicit left-padding of 2 zeros, which makes the convolution causal: token i combines only positions $\{i-2,\, i-1,\, i\}$ and never peeks ahead. Concretely, for channel c at position i,

$$y_i^{(c)} = \sum_{k=0}^{K-1} w_k^{(c)}\, x_{i-(K-1)+k}^{(c)} \overset{K=3}{=} w_0^{(c)} x_{i-2}^{(c)} + w_1^{(c)} x_{i-1}^{(c)} + w_2^{(c)} x_i^{(c)},$$

where the superscript (c) marks the channel: the layer is depthwise (`groups=d_model`), so each of the d_{model} channels has its own three-tap filter $w^{(c)}$ and is convolved independently of the others. Because it is depthwise, this path adds only $K \times d_{\text{model}} = 3 \times 256 = 768$ parameters per convolution layer, a rounding error next to the attention weights, which is why the hybrid variants in Table 6 barely move the total parameter count. Hybrid results use a 10-epoch budget (batch_size=4, seq_len=512); perplexities are not directly comparable to the 1000-step positional or 5-step attention diagnostic rows.

I deliberately place the convolution *before* attention. The reasoning is about ordering of operations: this way the local n-gram path feeds locally-smoothed representations *into* the global mixing step, so attention works on features that already have a clean local summary. Placing it after would instead smooth states that have already been globally contextualized, which is far less motivated, you'd be blurring information that attention just spent effort spreading around.

Variants. `conv_before_attn` inserts the causal depthwise convolution before each attention sublayer, so the local n-gram path feeds into the global mixing step in every block. `interleaved` is a lighter touch: instead of adding the convolution everywhere, it alternates standard Transformer blocks with conv-before-attention blocks across the model depth, so only some layers carry the extra local path. Finally, `best_combo` is the "stack the winners" configuration, it takes the best choice I found on each of the three axes and combines them: the interleaved hybrid structure, sparse_block attention ($B = 64$) for throughput, and ALiBi for position. It is the test of whether the individual gains compound or cancel.

Table 6: Hybrid architecture summary after a 10-epoch run at sequence length 512 (batch_size=4, seed=42). best_combo = interleaved architecture + sparse_block attention (B=64) + ALiBi PE. Lower loss, PPL, and time are better; higher throughput is better.

Variant	Total params	Peak resv. MB	Throughput (tokens s^{-1})	Time (s)	Val. loss	Val. PPL
`baseline`	17729792	2059.4	32470.8	736.5	5.335	207.38
`conv_before_attn`	17739008	2057.3	30705.4	778.9	5.155	173.35
`interleaved`	17734400	2059.4	31569.1	757.6	5.163	174.65
`best_combo`	17603328	2053.1	37916.3	630.7	4.952	141.51

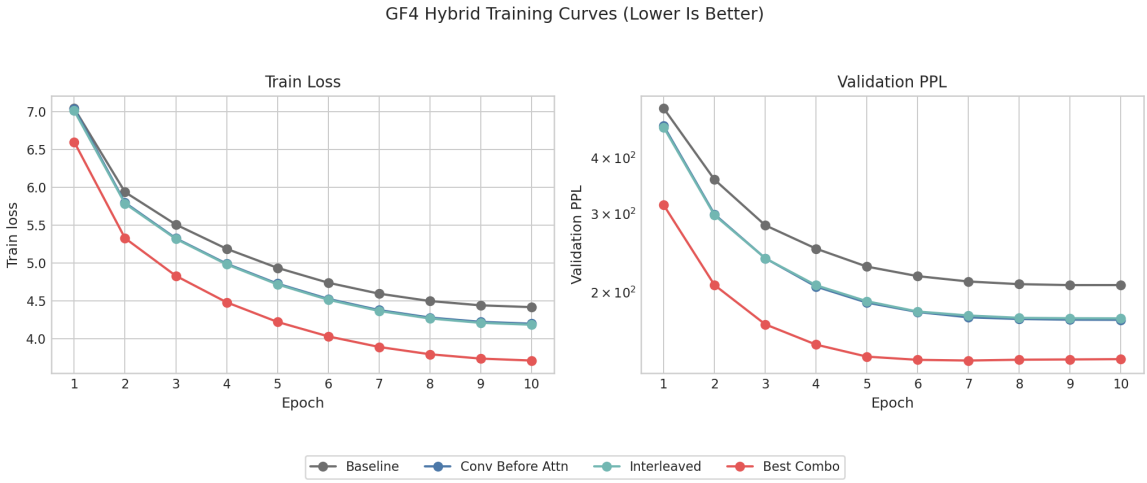


Figure 6: Validation PPL and training loss over 10 epochs for all four hybrid variants (10-epoch run, batch_size=4, seq_len=512, seed=42). Lower is better. best_combo = interleaved + sparse_block (B=64) + ALiBi, the best combination. best_combo converges to lower PPL than all baselines by epoch 4 and maintains the lead through epoch 10, confirming the compounding benefit of combining the best attention and PE choices.

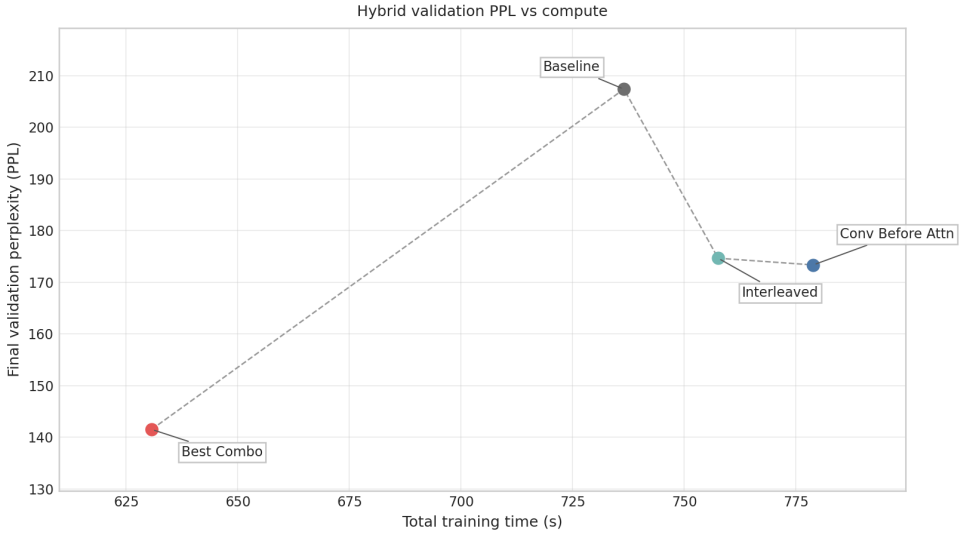


Figure 7: Final validation PPL versus total training time for all four hybrid variants (10-epoch run at sequence length 512, batch_size=4, seed=42). best_combo reaches the lowest PPL (141.51) with the shortest training time (630.7 s); baseline has the highest PPL (207.38) at 736.5 s. Values from Table 6.

2.8 Core ML Results Summary

On position, the 1000-step rows all land in the same 6-7 loss band, so the choice matters less than I expected at this budget, but there is still a clear order. RoPE is lowest at 512 tokens (6.137), followed by relative bias (6.236), sinusoidal (6.367), the ALiBi 1000-step row (6.622), and learned absolute at its valid 512-token length (6.628; Table 5). Learned absolute is OOB beyond 512, since its position table simply runs out of entries. The picture flips when I test extrapolation: in the separate 3000-step train-short-test-long experiment, ALiBi is the flattest curve and reaches 5.467 at 512 tokens (Figure 5). So RoPE wins in-distribution while ALiBi wins on length generalization, two different questions with two different answers.

The hybrid story is more decisive, and it is the clearest “compounding” result in the Core

ML task. The best_combo hybrid (interleaved + sparse_block + ALiBi) achieves the lowest val_ppl after 10 epochs (141.51; val_loss 4.952), comfortably ahead of conv_before_attn (173.35) and interleaved (174.65). What makes this notable is that it does not pay for quality with speed, best_combo *also* has the highest logged throughput (37,916 tokens per second) and the shortest total training time (630.7 s) in this 10-epoch budget (Table 6; Figure 6; Figure 7). That combination is exactly what I'd hope for: the sparse_block attention supplies the throughput, while the interleaved convolution path and ALiBi supply the quality, and because those gains come from different axes they stack rather than trade off. One bookkeeping note on how I report time: it is total wall-clock per run rather than per epoch, because the attention and positional sweeps are budgeted in optimization steps (5, 1000, and 3000 steps) rather than epochs; per-epoch time is only well-defined for the 10-epoch hybrid runs, where the four variants take 630.7-778.9 s total (Table 6).

2.9 Core ML Takeaways

Compressing the Core ML task into its core lessons: on hardware there is no single winner, it depends on what I'm optimizing for. Linear attention reserves the least memory at long context, Sparse-Block delivers the highest throughput, and both pull far ahead of dense attention as sequences grow (Table 16), because they avoid materializing the full $n \times n$ score matrix. On position, the right choice also depends on the question I'm asking: RoPE gives the lowest validation loss in-distribution (the 1000-step positional sweep), while ALiBi stays the flattest when I extrapolate past the training length (the separate 3000-step figure, Figure 5). And on architecture, interleaving a small convolutional local path with attention reliably improves the short-budget language-model objective, a cheap local prior earns its keep.

3 Sparsity & Optimization

3.1 Objective

The S&O task asks a focused question: how much task adaptation can I get by training only a small number of parameters? The main experiment fine-tunes DeBERTa-v3-base on GLUE CoLA [15] and reports MCC (Matthews correlation coefficient) as the primary metric:

$$\boxed{\text{MCC} = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}}.$$

I use MCC rather than accuracy because CoLA is class-imbalanced: most sentences are labeled acceptable, so a lazy model that *always* predicts the majority label can post high accuracy while learning nothing. MCC exposes this *majority-class collapse*: common implementations assign MCC $= 0$ to a single-class predictor because it has no predictive correlation. More generally it behaves like a correlation coefficient between predictions and labels, lying in $[-1, +1]$: $+1$ is perfect, 0 is no better than chance (or always guessing one class), and -1 is perfectly wrong. One practical note: in these runs the default cross-entropy loss never collapsed to the majority class, so I applied no class weighting.

3.2 PEFT Background

Fully fine-tuning the 184.7M-parameter backbone is costly on three fronts. First, with only about 8.5k CoLA training sentences, updating every parameter risks overfitting. Second, task-specific gradient updates can overwrite the model's pretrained linguistic knowledge, the catastrophic-forgetting problem. Third, storing optimizer state for the full model, plus a separate fine-tuned copy for every task, is expensive at scale. Parameter-efficient fine-tuning (PEFT) sidesteps all

three: I freeze the pretrained backbone and train only a small *adapter*, a lightweight trainable module bolted onto the frozen network, so the pretrained knowledge is preserved and I learn only a small correction on top of it.

In these experiments the adapters are inserted into the attention projections of DeBERTa-v3-base [7], which is itself pretrained in the ELECTRA style [4]. This completely changes the memory and optimizer-state profile of fine-tuning. Instead of touching all ≈ 184.7M backbone parameters, the trained PEFT rows update only about 296k for LoRA and SoRA and about 1.18M for AdaLoRA, a trainable fraction of

$$\frac{296\text{k}}{184.7\text{M}} \approx 0.0016 \quad (\approx 0.16\%),$$

so I am adapting well under one percent of the model.

DeBERTa-v3-base is a particularly good fit for CoLA for two reasons. Its *disentangled attention* represents each token with separate content and position vectors, rather than fusing the two into a single vector as standard attention does, which gives it finer-grained reasoning about syntactic structure. And its replaced-token-detection (ELECTRA-style) pretraining trains the model to judge, token by token, whether each word was the original or a plausible substitute; that objective makes it sensitive to local grammatical correctness, which is exactly the signal CoLA tests.

3.3 LoRA, AdaLoRA, and SoRA

These three methods form a natural progression on the same low-rank idea: LoRA fixes the rank, AdaLoRA lets the rank adapt across modules, and SoRA learns which rank components to keep at all.

Low-Rank Adaptation (LoRA). The core idea is to leave the big pretrained weight W frozen and learn a small low-rank correction beside it:

$$W' = W + \Delta W, \qquad \Delta W = BA,$$

where $A \in \mathbb{R}^{r \times d_{\text{in}}}$ and $B \in \mathbb{R}^{d_{\text{out}} \times r}$, so $\text{rank}(\Delta W) \leq r$. Because r is tiny, the trainable count drops from $d_{\text{out}} d_{\text{in}}$ to $r(d_{\text{out}} + d_{\text{in}})$ [8], for a 768×768 DeBERTa projection at $r = 8$ that is $589{,}824 \to 12{,}288$ trainable parameters, roughly $48\times$ fewer. In practice the update is scaled by α/r before being added back:

$$W' = W + \frac{\alpha}{r} BA,$$

and with $\alpha = 16$, $r = 8$ in this run the scale is exactly 2. I keep it identical to the LoRA baseline so that any difference from AdaLoRA comes from its adaptive rank, not from a scaling mismatch. I target the query and value projections, the two most task-sensitive directions for grammaticality, since q controls what each token attends to and v controls what it passes forward. And I initialize B to zero, so $\Delta W = BA = 0$ at the very start: the model begins in its exact pretrained state and only departs from it gradually as adaptation proceeds.

Adaptive Low-Rank Adaptation (AdaLoRA). AdaLoRA treats rank as an adaptive *budget* rather than a fixed number [16]. It parameterizes the update in an SVD-like form,

$$\Delta W = P \Sigma Q^{\top},$$

and adds a light orthogonality penalty,

$$R(P, Q) = \|P^{\top} P - I\|_F^2 + \|Q Q^{\top} - I\|_F^2,$$

to keep P and Q near-orthonormal, so the diagonal Σ genuinely behaves like singular values. During training it scores each rank component by an importance measure (roughly magnitude times gradient accumulation), prunes the least useful singular values, and redistributes the freed budget globally across modules, so more rank flows to the directions and layers that are pulling their weight. To quantify how much rank a module actually ends up using, I report an entropy-based effective rank:

$$r_{\text{eff}}(\Delta W) = \exp\Big(-\sum_i p_i \ln p_i\Big), \qquad p_i = \frac{\sigma_i}{\sum_j \sigma_j}.$$

This is the effective number of non-negligible singular directions: if one direction dominates it sits near 1, and if all r directions share weight evenly it sits near r. For this run AdaLoRA also widens the target set, beyond q and v it adapts k and the attention output, recalibrating all four content/position directions of DeBERTa's disentangled attention.

Sparse Low-Rank Adaptation (SoRA). SoRA keeps LoRA's fixed-rank structure but inserts a learnable gate vector g on the low-rank path:

$$\Delta W x = B(g \odot (Ax)).$$

Each gate scales one rank component, and a gate driven to exactly zero deletes that component outright, so the number of surviving nonzero gates, the effective rank $\|g\|_0$, becomes a post-training diagnostic of how many directions the task actually needed. The crucial design choice is *how* the gates reach zero: SoRA applies a per-gate proximal soft-threshold that snaps a gate to exactly zero once it falls below the threshold [5], rather than merely shrinking it toward zero the way an L1 subgradient step would. The next section makes precise why that distinction matters.

Table 7: Target modules in implemented experiments, confirmed by implementation inspection.

Architecture	Method	Adapted projections	N_{mod}; max rank
DeBERTa Transformer	LoRA	`query_proj`, `value_proj`	24; $r = 8$
DeBERTa Transformer	AdaLoRA	q, k, v, attention.output	48; $r_{\max} = 8$
DeBERTa Transformer	SoRA	`query_proj`, `value_proj`	24; $r = 8$, gates=8
Mamba (`mambapy`)	SoRA	`in_proj`, `out_proj`	8; $r = 8$
xLSTM mLSTM	SoRA	`q_proj`, `k_proj`, `v_proj`, `proj_down`	16; $r = 8$

3.4 Sparse Gate Optimization

Driving SoRA's gates to exact zero is genuinely an optimization problem, not just a thresholding heuristic, because the penalty I add is non-smooth. The full objective is

$$\mathcal{L}(\Delta) = \mathcal{L}_0(\Delta) + \lambda \sum_k \|g^{(k)}\|_1,$$

where $\mathcal{L}_0$ is the ordinary smooth task loss and $g^{(k)}$ is the gate vector for adapter group k. The ℓ_1 term is what pushes gates toward zero, but it is also exactly the part that ordinary gradient descent cannot handle cleanly: $\|g\|_1$ has a kink at $g_i = 0$ and no derivative there, so a plain gradient step can shrink a gate but will never park it at exactly zero.

Proximal-gradient derivation. The fix is proximal-gradient descent, which splits the smooth and non-smooth parts and treats each with the right tool. Each update has two stages. First, a forward (gradient) step on the smooth loss alone:

$$z_i = g_i^t - \eta \nabla_{g_i} \mathcal{L}_0.$$

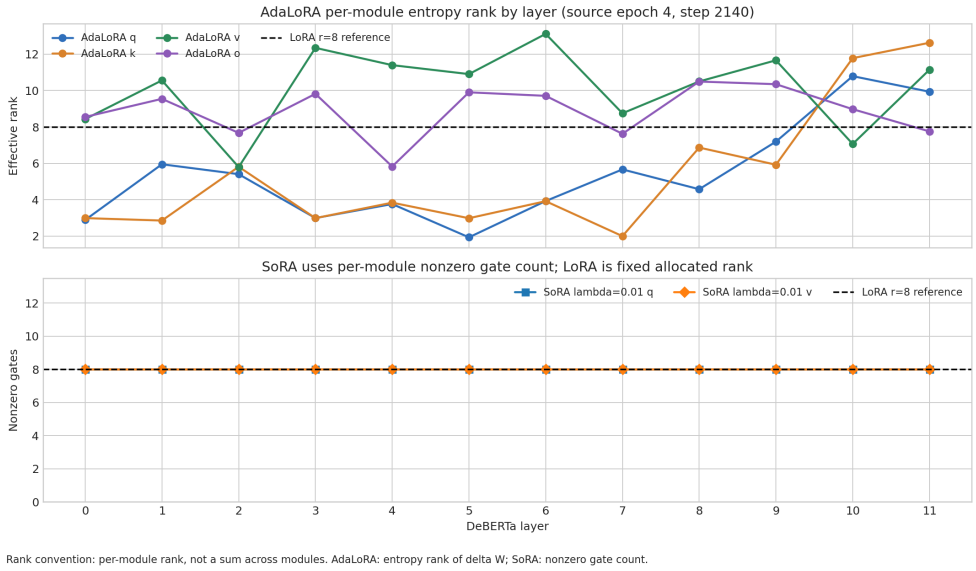


Figure 8: Per-layer effective rank analysis. The figure compares AdaLoRA entropy rank and SoRA nonzero gate count by layer using the per-module rank convention. AdaLoRA concentrates rank in middle layers while SoRA holds rank 8 flat across all layers, confirming adaptive versus fixed allocation across model depth.

Second, a backward (proximal) step that applies the ℓ_1 penalty exactly, by solving

$$g^{t+1} = \text{prox}_{\eta\lambda\|\cdot\|_1}(z) = \arg\min_g \left\{ \tfrac{1}{2}\|g - z\|_2^2 + \eta\lambda\|g\|_1 \right\}.$$

Both terms are separable across coordinates, so this decouples into one independent scalar problem per gate,

$$g_i^{t+1} = \arg\min_u \left\{ \tfrac{1}{2}(u - z_i)^2 + \eta\lambda\,|u| \right\},$$

and because this scalar objective is convex I can solve it exactly from its optimality condition $0 \in (u - z_i) + \eta\lambda\,\partial|u|$. Splitting on the sign of the minimizer: if $u > 0$ then $\partial|u| = 1$, giving $u = z_i - \eta\lambda$ (consistent only when $z_i > \eta\lambda$); if $u < 0$ then $\partial|u| = -1$, giving $u = z_i + \eta\lambda$ (consistent only when $z_i < -\eta\lambda$); and if $u = 0$ the subdifferential is the whole interval $[-1, 1]$, so 0 is optimal exactly when $|z_i| \le \eta\lambda$. Stitching the three cases together collapses to a single soft-thresholding rule:

$$\boxed{g_i^{t+1} = \text{sign}(z_i)\ \max\big(|z_i| - \eta\lambda,\ 0\big)}.$$

This is precisely why the operator yields *exact* zeros rather than merely small values: the middle case proves that any gate whose pre-threshold value lands in $|z_i| \le \eta\lambda$ is sent to exactly 0 in one step, because 0 is the true global minimizer of that coordinate's convex subproblem, not a number I am rounding down.

Why SGD-ℓ_1 cannot match this. SGD-ℓ_1 takes the other route: it folds the penalty straight into the gradient and applies its subgradient once per step. Far from zero the two behave alike, but near zero the subgradient's sign keeps flipping between steps, so a gate oscillates around zero indefinitely and never settles exactly on it. *Exact sparsity* is therefore not guaranteed in any finite number of steps, the analytic soft-threshold solve in the proximal step is doing work that a single subgradient step structurally cannot.

Production runs stay on the dense side. To locate where this dense-to-sparse transition actually occurs, I swept the threshold. The production values ($\lambda \in \{0.001, 0.01\}$) and the

intermediate ones ($\lambda \in \{0.05, 0.1\}$) all keep full rank 8 at gate sparsity 0.0, the penalty shrinks gate magnitudes but never kills a gate, and only at $\lambda = 0.2$ do all gates collapse to rank 0. So at the thresholds I actually run, SoRA never enters a sparse regime. The reason is easy to read straight off the soft-threshold: a gate dies only when $|z_i| \leq \eta\lambda$, and at my production thresholds that interval is far narrower than the trained gate magnitudes, which hold at mean nonzero 8.0 with gate sparsity 0.0% in every epoch (Table 12), no gate ever enters the killing interval within the training budget. The $\lambda = 0.2$ sweep pins the boundary down from the other side by collapsing every gate to rank 0. My production runs therefore sit entirely on the dense side of this transition, which is exactly why proximal SoRA matches LoRA in trainable-parameter count and yields no compute saving from rank reduction (Table 14).

3.5 Proximal vs SGD-L1 Updates

SGD-ℓ_1 takes the opposite approach to the proximal update: instead of solving a clean proximal subproblem, it folds the penalty straight into the gradient step as a subgradient,

$$g_i^{t+1} = g_i^t - \eta \left(\nabla_{g_i} \mathcal{L}_0 + \lambda s_i \right), \qquad s_i \in \partial |g_i^t|.$$

The subgradient s_i is well defined away from zero, $s_i = 1$ for $g_i > 0$ and $s_i = -1$ for $g_i < 0$, but at $g_i = 0$ it can be any value in $[-1, 1]$. This is enough to shrink a gate and make it hover near zero, but it never thresholds a small value to exactly zero.

Why the subgradient step never lands on zero. It is worth seeing precisely why, since it mirrors the proximal derivation. Strip away the task gradient (set $\nabla_{g_i} \mathcal{L}_0 = 0$, exactly as the toy below does) and the SGD-ℓ_1 update reduces to

$$g_i^{t+1} = g_i^t - \eta\lambda \ \mathrm{sign}(g_i^t).$$

Starting from a positive gate, every step subtracts a fixed amount $\eta\lambda$, marching the gate steadily toward zero. But once the gate falls within $\eta\lambda$ of zero, the next step overshoots past it; the sign flips, the following step overshoots back, and from then on the gate just bounces across zero forever with amplitude at most $\eta\lambda$. It can get arbitrarily close to zero but lands on it only by coincidence (a measure-zero event). The proximal operator avoids this entirely: its soft-threshold has a flat zero region baked in, so the instant $|z_i| \leq \eta\lambda$ the gate is set to exactly zero for that proximal step.

Toy Experiment for Sanity Check. My toy experiment makes this contrast concrete and visual. With $r = 8$, lr$= 0.1$, $\lambda = 0.15$ (so $\eta\lambda = 0.015$), a zero task gradient, and 200 steps, the proximal update reaches all-zero gates at step 67 while SGD-ℓ_1 still holds all 8 nonzero gates at step 200 (Figure 9). The step-67 number is not arbitrary: each proximal step strips $\eta\lambda = 0.015$ of magnitude, so a gate of initial magnitude near 1 is thresholded to zero in about $1/0.015 \approx 67$ steps. As a correctness guard I also ran an independent NumPy reference against the PyTorch implementation: for both proximal and SGD-ℓ_1 updates the maximum absolute difference is 0.0 with `allclose=true` at 10^{-6} tolerance, so the behavior above is a property of the math, not an artifact of a buggy implementation.

Table 8 lays the two update rules side by side across the properties that matter, whether exact zeros appear in finite steps, how each behaves near and at $g_i = 0$, and the toy outcome. And to confirm the exact-sparsity property survives outside the degenerate zero-gradient toy, I ran a small synthetic LASSO problem with a known sparse solution: minimize $\frac{1}{2}\|Ag - b\|^2 + \lambda\|g\|_1$ with $n = 20$, $d = 10$, a ground truth g_{true} containing 5 zeros, $\lambda = 0.1$, $\eta = 0.01$, and 500 steps. Proximal soft-thresholding recovers all 5 exact zeros; SGD-ℓ_1 recovers none, even though its final loss (0.38047) is essentially identical to proximal's (0.38020) (Table 9). The lesson is that the

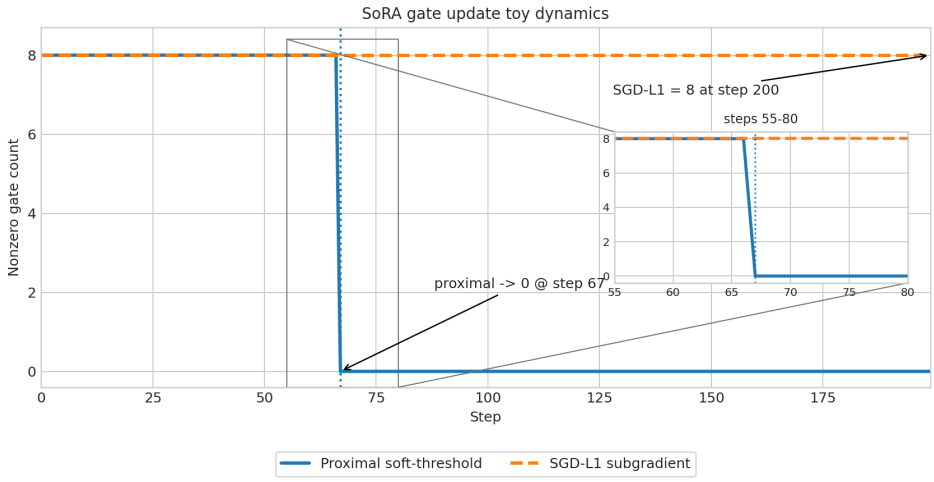


Figure 9: Toy sanity check of proximal versus SGD-L1 gate updates ($r = 8$, lr=0.1, $\lambda = 0.15$, zero task gradient, 200 steps). I keep this plot because it isolates the single behavior that separates the two optimizers, with everything else stripped away: with no task gradient the only force acting on the gates is the sparsity penalty, so the proximal operator should drive them to exact zero and SGD-L1 should not. That is exactly what the curves show, proximal collapses to all-zero at step 67 (each step removes $\eta\lambda = 0.015$ of magnitude, so a gate near 1 vanishes in roughly $1/0.015 \approx 67$ steps), while SGD-L1 merely oscillates and still holds all 8 gates at step 200. It is the cleanest visual proof that exact sparsity is a property of the *operator*, not of hyperparameter tuning.

Table 8: Theoretical and empirical comparison of update rules. Toy experiment uses zero task gradient; result confirms the proximal operator's exact-sparsity guarantee.

Property	Proximal soft-threshold	SGD-ℓ_1 subgradient
Exact zeros in finite steps	Yes; values inside the threshold interval map exactly to zero.	No finite-step zero is guaranteed by the subgradient step.
Convergence rate	Two-stage update can remove inactive gates abruptly once the threshold condition is met.	Shrinks gates gradually and can oscillate near zero.
Update structure	Smooth gradient step followed by the L1 proximal operator.	Single gradient step using an L1 subgradient term.
Behaviour at $g_i = 0$	Zero remains an exact fixed point after thresholding.	The subgradient set is multi-valued, so implementation choices can move or oscillate.
Sparsity guarantee	Exact sparsity is guaranteed when $\lvert z_i \rvert \leq \eta\lambda$.	Sparsity is encouraged but not guaranteed exactly in finite steps.
Toy result ($r = 8$, $\lambda = 0.15$, $\eta = 0.1$, 200 steps)	All-zero at step 67.	All 8 nonzero at step 200.

Table 9: Synthetic LASSO ($\min \frac{1}{2}\|Ag - b\|^2 + \lambda\|g\|_1$; $n = 20$, $d = 10$, g_{true} has 5 zeros, $\lambda = 0.1$, $\eta = 0.01$, 500 steps). Proximal soft-thresholding recovers exact zeros; SGD-ℓ_1 subgradient does not. $\|\cdot\|_\infty$ = max absolute difference between numpy and torch implementations on identical inputs.

Method	Impl.	Exact zeros	Near zeros ($\lvert g\rvert<10^{-6}$)	Final loss	$\|$numpy$-$torch$\|_\infty$
Proximal	NumPy	5	5	3.8020e-01	ref
Proximal	Torch	5	5	3.8020e-01	7.37e-18
SGD-ℓ_1	NumPy	0	0	3.8047e-01	ref
SGD-ℓ_1	Torch	0	0	3.8047e-01	5.55e-17

two reach nearly the same objective value, but only the proximal route delivers true zeros. Both implementations again agree with NumPy to machine precision ($\|\text{numpy} - \text{torch}\|_\infty \sim 10^{-17}$).

In full CoLA training the same ordering holds: proximal SoRA at $\lambda = 0.01$ reaches MCC=0.6777, while SGD-ℓ_1 reaches MCC=0.6677. Both production rows have gate sparsity 0.0 and per-module effective rank 8 (nonzero gates per SoRALinear module; 24 modules = q,v across 12 layers), so neither enters a sparse regime. That is consistent with the parameter accounting: $24 \times (8 \cdot 768 + 768 \cdot 8 + 8)$ plus the classifier gives approximately 296642 trainable parameters. One subtlety is worth separating out: the entropy effective ranks in Table 10 measure the singular-value spread of ΔW, which is a different quantity from the SoRA nonzero-gate count, the $\lambda = 0.01$ SoRA checkpoint has mean entropy rank 5.0 yet still retains all 8 nonzero gates per adapted module.

3.6 Does the Subgradient Choice at Zero Matter?

There is one loose end in the subgradient story worth addressing head-on, because it looks like it should matter but does not. At $g_i = 0$ the ℓ_1 subdifferential is not a single number but the whole interval:

$$\partial|g_i| = \begin{cases} \{+1\}, & g_i > 0, \\ \{-1\}, & g_i < 0, \\ [-1,\, 1], & g_i = 0. \end{cases}$$

So at exactly zero I am free to define sign(0) as $+1$, -1, or 0, and the natural worry is that this arbitrary choice could change SGD-ℓ_1's behavior. It does, but only on the set $\{g_i = 0\}$ itself, a measure-zero set that a finite-precision trajectory essentially never lands on exactly. And once a gate is merely oscillating *near* zero (which is where it actually spends its time), it is the task-loss gradient, not the convention I picked for sign(0), that decides which way it moves.

The proximal operator settles the question even more cleanly, because it is genuinely indifferent to the convention even at exactly zero. Plugging $z_i = 0$ into the soft-threshold,

$$g_i^{t+1} = \text{sign}(0)\ \max(|0| - \eta\lambda,\, 0) = \text{sign}(0) \cdot 0 = 0,$$

the result is 0 whatever value sign(0) takes, because the $\max(\cdot, 0)$ factor has already collapsed to zero inside the threshold interval. So zero is an exact fixed point of the proximal map when the next pre-threshold value remains inside the threshold interval. The subgradient step has no such guarantee, at exactly $g_i = 0$ with a convention of $\text{sign}(0) = \pm 1$ it would actively nudge the gate back off zero by $\eta\lambda$, undoing the sparsity (though, again, only on that measure-zero set).

The real gap between SGD-ℓ_1 and the proximal update is therefore not the convention at zero at all, it is the analytic soft-threshold solve. The proximal operator maps the entire interval $|z_i| \le \eta\lambda$ to exactly zero in one step and keeps it there, while the subgradient step never produces an exact zero in any finite number of steps, no matter how I define sign(0).

3.7 xLSTM and Mamba Extensions

So far every adapter has gone into a Transformer's attention projections (`query_proj`, `value_proj`). The question I wanted to answer here is whether the same SoRA machinery transfers to a completely different family of sequence models, the recurrent xLSTM and the state-space Mamba. The guiding idea is simple. Adapt the large dense input and output projections that feed each block, and leave the block's actual sequence-mixing machinery untouched.

The backbones. I use `mambapy` [6] and the `xlstm` package [2] as the backbone implementations, so the recurrences below are the standard equations from those packages, not new derivations of my own. In a selective state-space (Mamba) block, the discrete-time state update is

$$h_t = \bar{A}_t h_{t-1} + \bar{B}_t x_t, \qquad y_t = C_t h_t + D x_t.$$

In an mLSTM block, input, forget, and output gates regulate a matrix-memory cell:

$$i_t = \sigma(W_i x_t), \quad f_t = \sigma(W_f x_t), \quad o_t = \sigma(W_o x_t), \quad C_t = f_t \odot C_{t-1} + i_t \odot (v_t k_t^\top), \quad h_t = o_t \odot (C_t q_t).$$

Where the adapters go. The SoRA gate I defined earlier wraps a frozen projection W in exactly the same way,

$$W'x = Wx + B(g \odot (Ax)),$$

with W frozen and only A, B, and g trained. I simply apply it to a different set of projections in each backbone. The `xlstm` package exposes separate mLSTM projections rather than a fused QKV matrix, so I wrap `q_proj`, `k_proj`, `v_proj`, and `proj_down` in each of four mLSTM blocks. For Mamba I wrap `in_proj` and `out_proj` in each of four `mambapy` mixer blocks. I deliberately leave Mamba's `x_proj`, `dt_proj`, selective-scan recurrence, and depthwise `conv1d` untouched, since those encode the state-space update itself rather than the dense adapter boundary.

Setup and results. Both runs share the same minimal setup: a word-level CoLA tokenizer built from the training split only (`[PAD]`=0, `[UNK]`=1, maximum vocabulary 20000, context length 128), random-frozen 256-dimensional token embeddings, rank $r = 8$, and proximal SoRA thresholding at $\lambda = 0.01$. With that, xLSTM reaches MCC=0.0784 at epoch 3, training 123522 parameters (effective rank 8 per module across 16 wrapped projections) in 68.8 s at 350.1 MB peak VRAM. Mamba reaches MCC=0.1275 at epoch 5, training 66114 parameters (effective rank 8 per module across 8 wrapped projections) in 293.2 s at 1542.2 MB peak VRAM. The full runtime comparison is in Table 15 (Figure 11).

Reading the numbers. These MCC values sit far below the DeBERTa LoRA, AdaLoRA, and SoRA points near 0.68, and that is expected, because unlike DeBERTa these xLSTM and Mamba backbones are trained from scratch on only about 8.5k CoLA examples, with no language-model pretraining. The point of this experiment is not that a from-scratch sequential model should rival a pretrained DeBERTa, but that the SoRA adapter wrapper runs end-to-end on recurrent and state-space backbones. The sanity check comes from the trivial baseline. A single-class (majority-label) predictor is reported as MCC= 0 by common CoLA evaluation code, so the from-scratch xLSTM (0.0784) and Mamba (0.1275) both sit above zero in this run, suggesting that the adapter is learning task signal rather than collapsing to the majority class. On the implementation side, I use `mambapy`, a pure-PyTorch reimplementation with the reference selective-scan path, because the official `mamba-ssm` package was blocked by an `nvcc` build failure under WSL on this RTX 4060 machine, so no compiled selective-scan CUDA kernel is used here.

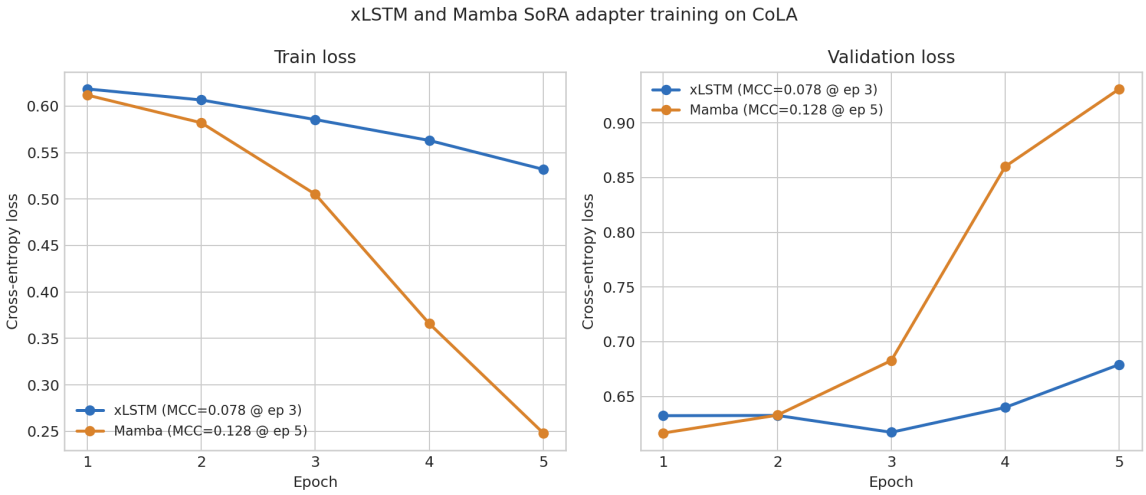


Figure 10: Training and validation loss for xLSTM and Mamba SoRA adapter runs on CoLA (from-scratch word-level tokenizer, frozen random 256-dim embeddings, $r = 8$, $\lambda = 0.01$). xLSTM: MCC=0.0784 at epoch 3; Mamba: MCC=0.1275 at epoch 5. MCC growth suggests that the adapter methodology transfers to sequential architectures in this setup. Both backbones show decreasing training loss over epochs, consistent with SoRA adapter transfer to recurrent and SSM architectures beyond Transformers.

Adaptation principle. The design rule for both backbones is identical: adapt the dense projection surface that maps between the model width and the sequence-mixing machinery, and leave that machinery frozen. In the Transformer that surface is the attention projection (q, v); in Mamba it is `in_proj` and `out_proj`, which set what enters and leaves the selective scan; in the mLSTM it is `q_proj`, `k_proj`, `v_proj`, and `proj_down`, which feed and read the matrix-memory cell. The state-update components, Mamba's `x_proj`, `dt_proj`, the selective-scan recurrence, and the depthwise `conv1d`, stay untouched because they encode the temporal dynamics rather than the dense adapter boundary, and injecting low-rank updates into the recurrence would break the associative-scan efficiency these architectures rely on. SoRA's gating then applies unchanged on top of this surface, so the very same proximal rank-selection mechanism carries over from attention to recurrent and state-space blocks.

3.8 S&O Results Summary

Table 10: S&O comparison on GLUE CoLA. DeBERTa rows are pretrained PEFT runs; xLSTM and Mamba rows are from-scratch adapter-transfer runs. MCC is the primary metric. AdaLoRA adapts q,k,v,o (48 modules); LoRA and SoRA adapt q,v (24 modules); trainable-parameter counts and effective ranks are not module-matched across methods. Entropy effective rank is reported where checkpoint factors exist; dash in that column means no entropy-rank checkpoint was computed for that row. Entropy rank $= \exp(-\sum_i p_i \ln p_i)$, where $p_i = \frac{\sigma_i}{\sum_j \sigma_j}$ over singular values of ΔW; it measures effective dimensionality of the update.

Method	MCC	Val. loss	Trainable params	Ratio	Entropy rank (mean)	Peak VRAM (MB)	Time (s)
LoRA	0.6848	0.3559	296450	0.0016	4.5	4023.5	563.9
AdaLoRA	0.6782	0.3577	1181954	0.0064	7.4	2449.2	361.9
SoRA, proximal $\lambda = 0.001$	0.6655	0.4653	296642	0.0016	—	4599.9	561.3
SoRA, proximal $\lambda = 0.01$	0.6777	0.4484	296642	0.0016	5.0	3839.0	558.1
SoRA, SGD-L1 $\lambda = 0.01$	0.6677	0.4125	296642	0.0016	—	3860.2	559.8
xLSTM-SoRA (from scratch)	0.0784	0.6172	123522	—	—	350.1	68.8
Mamba-SoRA (from scratch)	0.1275	0.9310	66114	—	—	1542.2	293.2

Table 10 gathers every S&O run in one place, with MCC as the headline metric and trainable

parameters, entropy effective rank, peak VRAM, and wall-clock time alongside. Among the pretrained DeBERTa adapters, LoRA is highest in this single-seed run at MCC=0.6848, with AdaLoRA (0.6782) and proximal SoRA at $\lambda = 0.01$ (0.6777) just behind, and SGD-L1 SoRA (0.6677) and proximal SoRA at $\lambda = 0.001$ (0.6655) trailing. I read the top three as effectively tied, the gap sits well inside CoLA's seed-to-seed variance ($\pm$0.0163 MCC standard deviation), and this is a single seed with no significance test.

The more telling story is what that tie costs. LoRA and SoRA both train about 296k parameters (ratio 0.0016), while AdaLoRA trains 1181954, roughly four times as many (ratio 0.0064), for no MCC gain. AdaLoRA does win the hardware columns in return, at 361.9 s and 2449.2 MB it is the fastest and lightest DeBERTa run, well under LoRA's 563.9 s and 4023.5 MB, because its SVD-style pruning trims the active rank during training. It also spreads rank the widest, with mean entropy effective rank 7.4 at its epoch-4 best-dev checkpoint, against 4.5 for LoRA and 5.0 for proximal SoRA. These ranks are not strictly comparable, since the methods are not module-matched (AdaLoRA adapts q, k, v, o across 48 modules, while LoRA and SoRA adapt only q, v across 24).

Two final points for context. My AdaLoRA's 0.6782 is below the $\approx$ 0.70 reported in the original paper, most likely a scope difference, since the canonical setup also adapts the intermediate and layer-output projections that I left out. And the from-scratch xLSTM (0.0784) and Mamba (0.1275) rows sit far below every pretrained run, exactly as expected with no pretraining, so they belong here as transfer evidence rather than as competitors to DeBERTa.

3.9 AdaLoRA Behaviour

AdaLoRA (0.6782) and LoRA (0.6848) are essentially a tie, the gap is comfortably inside the seed-to-seed variance CoLA is known for (AdaLoRA alone carries a $\pm$0.0163 MCC standard deviation). What makes the comparison even softer is that the two are not adapting the same thing, AdaLoRA adapts q, k, v, and the attention output across 48 modules (1,181,954 trainable parameters), while LoRA touches only q and v across 24 modules (296,450). Where AdaLoRA does behave distinctively is in how it spends its rank, its best dev checkpoint (epoch 4) has a mean entropy effective rank of 7.4, meaning it spreads useful directions across most of its budget rather than leaning on a fixed rank. Its per-epoch dev MCC also climbs monotonically to that epoch-4 peak (Table 12) with no dip during pruning, so reallocating the rank budget is not destabilizing training. The one place it clearly trails the literature, 0.6782 against the $\approx$ 0.70 of the original paper, is most likely a scope difference, since the canonical setup also adapts the intermediate and layer-output projections, whereas I restricted adaptation to the four attention projections.

Per-epoch training details. Tables 11–13 hold the seed-pinned per-epoch traces. I use these only to recover training curves and adapter checkpoints, the headline MCC, loss, VRAM, and time values in Table 10 remain the canonical measured numbers.

Table 11: LoRA seed-pinned per-epoch metrics. The trace keeps the canonical LoRA hyperparameters and writes only checkpoint and epoch artifacts; its final MCC is within 0.0023 of the canonical headline value.

Epoch	Train loss	Dev MCC	Time (s)	Trainable params
1	0.4144	0.6070	174.7	296450
2	0.3181	0.6662	339.2	296450
3	0.2737	0.6651	494.7	296450
4	0.2378	0.6725	650.4	296450
5	0.2121	0.6826	807.2	296450

Table 12: AdaLoRA seed-pinned per-epoch metrics. The best epoch exactly matches the canonical best-dev MCC; the entropy rank is the final checkpoint mean repeated for context because entropy rank is computed after checkpoint selection.

Epoch	Train loss	Dev MCC	Time (s)	Mean entropy rank
1	0.5143	0.6107	184.3	7.4
2	0.3587	0.6381	370.1	7.4
3	0.3203	0.6701	553.2	7.4
4	0.2863	0.6782	735.3	7.4
5	0.2550	0.6703	965.8	7.4

Table 13: SoRA proximal $\lambda = 0.01$ seed-pinned per-epoch metrics. The final MCC is within 0.0035 of the canonical headline value, while gate sparsity remains 0.0 and all 8 gates stay nonzero in every adapted module.

Epoch	λ	Train loss	Dev MCC	Gate sparsity (%)	Mean nonzero gates
1	0.01	0.4349	0.6085	0.0	8.0
2	0.01	0.3349	0.6482	0.0	8.0
3	0.01	0.2881	0.6483	0.0	8.0
4	0.01	0.2666	0.6412	0.0	8.0
5	0.01	0.2374	0.6812	0.0	8.0

Table 14: CoLA MCC vs. trainable parameters for pretrained DeBERTa PEFT rows. xLSTM and Mamba are excluded because they use from-scratch sequential backbones, not pretrained DeBERTa. LoRA, AdaLoRA, and SoRA cluster within about 0.007 MCC points, well within CoLA seed-to-seed variance; because this is a single-seed run, the table supports a variance caveat rather than a significance claim.

Method	Trainable params	CoLA MCC
LoRA	296450	0.6848
AdaLoRA	1181954	0.6782
SoRA prox $\lambda = 0.01$	296642	0.6777

Parameter efficiency and runtime. Pulling the three pretrained adapters together (Table 14), the simple conclusion is that fixed-rank LoRA is the most efficient at this budget, it reaches the top MCC (0.6848) with 296,450 parameters (ratio 0.0016), while AdaLoRA spends about four times as many (1,181,954; ratio 0.0064) for a near-identical 0.6782, and SoRA matches LoRA's count (296,642) at 0.6777. So adaptive rank allocation does not repay its extra parameter cost in MCC here, on a single seed the simplest fixed-rank adapter is the efficient choice. Training time tells a complementary story (Table 15, Figure 11), the LoRA and SoRA variants all cluster around 558–564 s, AdaLoRA is noticeably faster at 361.9 s because its SVD-style pruning shrinks the active rank as it trains, and the from-scratch sequential runs are fastest of all (Mamba-SoRA 293.2 s, xLSTM-SoRA 68.8 s), though only because they use small-vocabulary backbones rather than the full pretrained DeBERTa. Per-layer rank behavior was shown earlier in Figure 8.

3.10 S&O Takeaways

The single-seed headline is that LoRA is the top result (0.6848), but I would not over-claim it, because AdaLoRA (0.6782) and proximal SoRA (0.6777) both fall inside the seed-to-seed variance the literature reports for CoLA (AdaLoRA alone carries a ± 0.0163 MCC standard deviation), and I ran only one seed with no significance test. So the fair reading is that AdaLoRA matches LoRA's quality while reallocating rank adaptively across 48 modules, not that either method is genuinely better. On the sparsity side, SoRA's proximal update is mathematically

Table 15: Training-time comparison for adapter methods and sequential extensions. Times are single-run wall-clock measurements on the RTX 4060 setup. SoRA and LoRA have comparable training time at this budget; the production $\lambda = 0.01$ does not induce sparsity, so no compute saving from rank reduction is expected.

Method	Training time (s)
LoRA	563.9
SoRA prox $\lambda = 0.001$	561.3
SoRA SGD-L1 $\lambda = 0.01$	559.8
SoRA prox $\lambda = 0.01$	558.1
AdaLoRA	361.9
Mamba-SoRA	293.2
xLSTM-SoRA	68.8

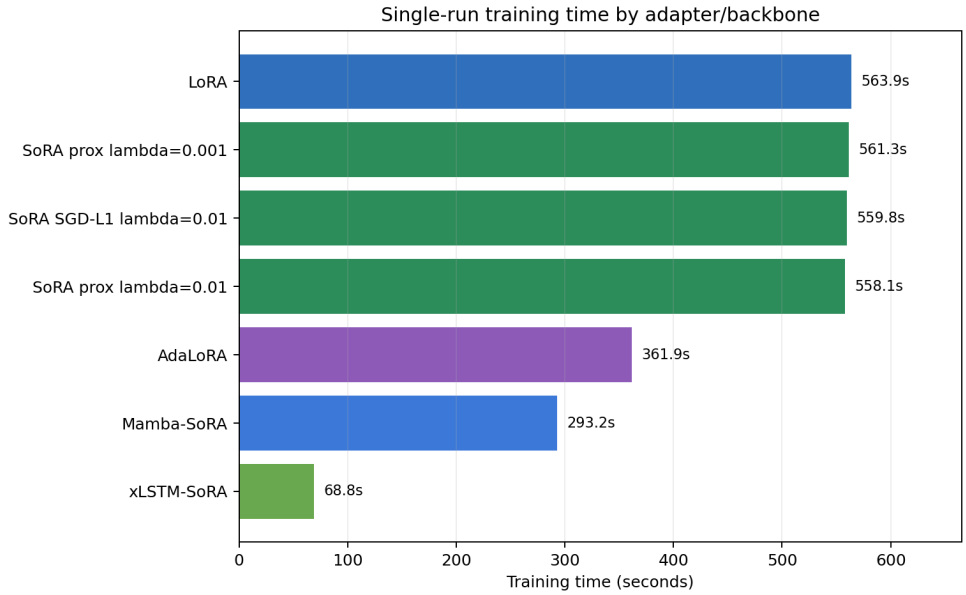


Figure 11: Training time by adapter and backbone (values in Table 15). LoRA and SoRA variants cluster at 558-564 s; AdaLoRA at 361.9 s benefits from SVD-pruning-driven shorter compute; Mamba-SoRA (293.2 s) and xLSTM-SoRA (68.8 s) use from-scratch small-vocabulary backbones, not pretrained DeBERTa.

aligned with exact sparsity, but the thresholds I actually ran in production are too small to zero any gates, so that mechanism stays dormant here.

More broadly, the real lesson generalizes past SoRA: producing exact zeros requires solving an analytic subproblem, the proximal operator, not just taking a gradient step. Any method that genuinely needs true zeros, whether model compression, structured pruning, or feature selection, should reach for a proximal or equivalent analytic solver rather than an ℓ_1 subgradient.

4 Conclusion

4.1 Key Findings

On the Core ML side, the clearest win is memory. At sequence length 4096, linear attention cuts peak reserved VRAM from 10605 MB (standard) down to 4664 MB and lifts throughput from 803 to 17475 tokens per second, while Sparse-Block pushes throughput all the way to 33902 tokens per second (Table 16). For position, the 1000-step rows cluster on the expected loss scale

with RoPE lowest at 512 tokens (Table 5), while ALiBi stays the flattest train-short-test-long curve in the separate 3000-step extrapolation figure (Figure 5). Among the hybrids, best_combo (interleaved + sparse_block + ALiBi) is the strongest after 10 epochs, reaching val_loss 4.952 and val_ppl 141.51.

On the S&O side, LoRA achieves the highest MCC in this single-seed run (0.6848), but AdaLoRA (0.6782) and proximal SoRA (0.6777) sit within CoLA's seed-to-seed variance (AdaLoRA ± 0.0163 MCC standard deviation), so the three are effectively tied. Within SoRA itself, the proximal update does edge out SGD-ℓ_1 on MCC, 0.6777 against 0.6677, consistent with the exact-sparsity advantage of the proximal operator.

4.2 Limitations

I should be clear that this evidence is diagnostic rather than fully converged. The Core ML attention metrics come from a 5-step hardware sweep, positional extrapolation uses 1000 to 3000 steps, the hybrid runs use 10 epochs, and the S&O CoLA results are single-seed.

The xLSTM and Mamba extensions are trained from scratch on only about 8.5k CoLA examples, so their MCC values near 0.08 to 0.13 are not comparable to pretrained DeBERTa fine-tuning. That gap is a scope limitation of the extension experiment, not a failure of the method (Table 10; Table 14).

Finally, I deliberately scoped out two optional extensions. I did not implement positional interpolation and context-window scaling, since the extrapolation study already isolates the train-short-test-long behavior of each encoding directly. And I left the Attention-Free Transformer bonus out of this submission, since the six attention variants I did implement already span the dense, local, block-sparse, and linear-kernel regimes that bonus targets.

A Appendix

A.1 Attention Hardware Detail

Table 16: Attention hardware comparison (5-step diagnostic sweep, batch_size=1 for all contexts). Val. loss near 253 and capped PPL are not quality metrics; only hardware metrics are meaningful here. SBA = Sparse Block Attention (B=64). The sliding-window peak reserved VRAM equals standard attention at each context length by design: this implementation is mask-only and still materializes the dense $n \times n$ score matrix, so it is a local-connectivity ablation rather than a memory-saving kernel.

Variant	Seq. len	Peak alloc. MB	Peak resv. MB	Throughput (tokens s^{-1})	Latency (ms)
`standard`	256	301.8	484.4	1033.7	3.0242
`standard`	512	405.7	880.8	5664.5	4.8599
`standard`	1024	609.6	1449.1	7695.9	15.1681
`standard`	2048	1021.3	3596.6	9614.9	51.7120
`standard`	4096	1960.8	10605.3	802.8	1259.8622
`sliding_window`	256	301.8	484.4	1808.4	4.0637
`sliding_window`	512	405.7	880.8	4273.3	5.0795
`sliding_window`	1024	609.6	1465.9	4471.2	15.0743
`sliding_window`	2048	1021.3	3663.7	8089.7	54.2702
`sliding_window`	4096	1977.7	10605.3	1943.7	1302.0097
`linear`	256	301.8	555.7	2996.0	3.7447
`linear`	512	405.7	885.0	5051.8	5.9794
`linear`	1024	609.7	1260.4	5933.6	12.4601
`linear`	2048	1021.3	2382.4	12741.4	24.8122
`linear`	4096	1849.9	4664.1	17475.0	49.2767
`sparse_block` (SBA)	256	302.8	513.8	2609.5	3.7317
`sparse_block` (SBA)	512	405.8	786.4	6016.0	4.2917
`sparse_block` (SBA)	1024	609.6	1103.1	10798.4	6.2011
`sparse_block` (SBA)	2048	1021.4	2032.1	17623.9	12.7660
`sparse_block` (SBA)	4096	1850.8	4731.2	33902.0	27.8915
`gqa`	256	295.9	476.1	2826.6	3.4611
`gqa`	512	399.8	872.4	5711.6	4.5316
`gqa`	1024	603.7	1440.7	9284.5	14.8435
`gqa`	2048	1015.4	3588.2	9688.7	50.0455
`gqa`	4096	1954.9	10592.7	1710.1	867.7922
`mqa`	256	294.9	474.0	2725.8	3.3974
`mqa`	512	398.8	872.4	4798.2	4.2420
`mqa`	1024	602.7	1438.6	8017.2	14.3529
`mqa`	2048	1014.4	3586.1	8235.5	50.9871
`mqa`	4096	1953.9	10590.6	1703.3	975.1972

References

[1] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zhou, O. Press, and S. Ontanon. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[2] M. Beck, K. Pöppel, M. Spanring, A. Auer, O. Prudnikova, M. Kopp, G. Klambauer, J. Brandstetter, and S. Hochreiter. xLSTM: Extended long short-term memory. In *Advances in Neural Information Processing Systems*, 2024.

[3] I. Beltagy, M. E. Peters, and A. Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[4] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning. ELECTRA: Pre-training text encoders as discriminators rather than generators. In *International Conference on Learning Representations*, 2020.

[5] N. Ding, X. Lv, Q. Wang, Y. Chen, B. Zhou, Z. Liu, and M. Sun. Sparse low-rank adaptation of pre-trained language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

[6] A. Gu and T. Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[7] P. He, J. Gao, and W. Chen. DeBERTaV3: Improving DeBERTa using ELECTRA-style pre-training with gradient-disentangled embedding sharing. In *International Conference on Learning Representations*, 2023.

[8] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[9] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *Proceedings of the 37th International Conference on Machine Learning*, 2020.

[10] O. Press, N. A. Smith, and M. Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations*, 2022.

[11] P. Shaw, J. Uszkoreit, and A. Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2018.

[12] N. Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

[13] J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568, 2024.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[15] A. Warstadt, A. Singh, and S. R. Bowman. Neural network acceptability judgments. *Transactions of the Association of Computational Linguistics*, 7:625–641, 2019.

[16] Q. Zhang, M. Chen, A. Bukharin, N. Karampatziakis, P. He, Y. Cheng, W. Chen, and T. Zhao. AdaLoRA: Adaptive budget allocation for parameter-efficient fine-tuning. In *International Conference on Learning Representations*, 2023.