

CONTENTS

1	Introduction	1
2	Method	2
2.1	Dynamic Column Selection	2
2.2	Discrete Cosine Transform	3
2.3	Trion: DCT-based improvement to Dion	3
2.4	DCT-AdamW	4
3	Experiments	4
4	Theoretical Guarantees	6
4.1	Optimality of norm-based ranking procedure	6
4.2	DCT as linear approximation of the gradient eigenbasis	7
5	Related Work	8
6	Conclusion and Limitations	9
A	Efficient Computation of the DCT Matrix on GPU	13
B	Dynamic Column Selection Approach	13
C	Motivation of Discrete Cosine Transform Matrix	13
D	Makhoul’s Algorithm Explained	14
E	Pseudocode of DCT-AdamW Optimizer	15
F	Projection Errors of Trion and Dion	16
G	Pre-Training with FRUGAL/FIRA	16
H	Fine-Tuning	17

A EFFICIENT COMPUTATION OF THE DCT MATRIX ON GPU

In this section we present how we can efficiently compute the DCT-III matrix on a GPU using a vectorized implementation that is fast for large values of n on the GPU. To create the DCT-III matrix $Q \in \mathbb{R}^{n \times n}$, we first create one vector $L = [0, \dots, n-1]^\top$, which is used to create the matrix $\mathcal{I} \in \mathbb{N}^{n \times n}$ by replicating L on the columns of \mathcal{I} :

$$L = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ n-1 \end{bmatrix} \quad \mathcal{I} = \begin{bmatrix} 0 & \dots & 0 \\ 1 & \dots & 1 \\ \vdots & \dots & \vdots \\ n-1 & \dots & n-1 \end{bmatrix}_{n \times n} \quad Q = \sqrt{\frac{2}{n}} \cos \left(\frac{\mathcal{I} \odot (2\mathcal{I}^\top + 1)}{2n} \pi \right)$$

We restate that we need to divide the first row by $\sqrt{2}$ to make Q orthogonal. The computational efficiency comes from the element-wise product $\mathcal{I} \odot (2\mathcal{I}^\top + 1)$ that actually computes the integer entries $i(2j+1)$. The DCT-II matrix can be obtained by transposing the DCT-III.

B DYNAMIC COLUMN SELECTION APPROACH

In this section we provide further details about our dynamic column selection approach. When we compute the similarities S , we look at the columns of S where on i^{th} column we have the scalar products between all rows of G and the i^{th} column of Q . By choosing the columns with largest ℓ_1 - or ℓ_2 -norms we make sure we select the columns with largest overall alignment with all rows in G .

$$S = GQ = \begin{bmatrix} \text{---} G_1 \text{---} \\ \text{---} G_2 \text{---} \\ \vdots \\ \text{---} G_n \text{---} \end{bmatrix} \begin{bmatrix} \left| \begin{smallmatrix} Q_1 \\ Q_2 \\ \vdots \\ Q_n \end{smallmatrix} \right| & \left| \begin{smallmatrix} Q_2 \\ Q_2 \\ \vdots \\ Q_n \end{smallmatrix} \right| & \dots & \left| \begin{smallmatrix} Q_1 \\ Q_2 \\ \vdots \\ Q_n \end{smallmatrix} \right| \end{bmatrix} = \begin{bmatrix} G_1^\top Q_1 & G_1^\top Q_2 & \dots & G_1^\top Q_n \\ G_2^\top Q_1 & G_2^\top Q_2 & \dots & G_2^\top Q_n \\ \vdots & \vdots & \dots & \vdots \\ G_n^\top Q_1 & G_n^\top Q_2 & \dots & G_n^\top Q_n \end{bmatrix} \quad (3)$$

C MOTIVATION OF DISCRETE COSINE TRANSFORM MATRIX

In general, any orthogonal matrix would yield similar quantitative results when used with our dynamic column selection technique. In this section we are interested in exploring different types of orthogonal matrices and our goal is to find a candidate matrix whose structure allows us to reduce the computational complexity for the operation $S = GQ$, especially for large layers.

The first candidate is a random orthogonal matrix Q , which can be obtained by generating a random Gaussian matrix and then orthogonalizing it using QR-decomposition (e.g. keeping only the Q-component from the decomposition), which should be done only once at the beginning of training. While this would work, it has the drawback of always having $O(n^3)$ complexity when computing the similarities S because the matrix does not have any structure that would allow us to use an algorithm with lower complexity.

The second candidate is the Hadamard matrix, a particular Fourier matrix containing only values ± 1 , which is used in the model compression literature and is known for its fast multiplication routines tailored to GPUs. However, it has the drawback of being ill defined for certain values of model's dimension d_{model} and the current existing procedures provide a matrix that is not orthogonal in these cases, making it unusable for our context.

Our preferred candidate is the DCT matrix, which is also a Fourier matrix and has a particular structure that gives it the potential of computing $S = GQ$ (S is called the discrete cosine transform of G), making it faster than the $O(n^3)$ complexity of matmul, as we present below.

When Q is the DCT-II matrix, we can reduce the complexity of $S = GQ$ from $O(n^3)$ to $O(n^2 \log(n))$ by using the Makhoul's N -point algorithm [Makhoul \(1980\)](#) to compute DCT faster. We summarize the algorithm in Appendix D and also provide a benchmark for different layer sizes and data types encountered in practice. In short, this is a FFT-based algorithm that benefits from the pre- and post-processing steps of the input matrix G (such as permutations and multiplications with complex

values) to reduce the computational complexity. Interestingly, the link between Makhoul’s algorithm to compute the cosine transform and the same transformation computed via a basic matmul is that the matmul version embeds all operations of the Makhoul’s algorithm in the DCT-II matrix itself, but at a higher computational cost.

Given the current data types implemented in PyTorch, Makhoul’s algorithm can be run only on float32 inputs, which makes it infeasible for practical bfloat16 inputs unless the input size is a power of 2. The limitation comes from the lack of complex-bfloat16 type support in PyTorch to represent the real and imaginary parts of a complex number as bfloat16. See Appendix D for more details.

D MAKHOUL’S ALGORITHM EXPLAINED

Makhoul’s N -point algorithm to compute a fast DCT can be summarized as follows:

1. apply a permutation to the input signal X to obtain X_{PERM} : vector $[a, b, c, d, e, f]$ becomes $[a, c, e, f, d, b]$, where odd indices are in increasing order, even indices are in decreasing order and they are interleaved (can be cached for the same input size);
2. compute FFT of the permuted signal X_{PERM} and obtain X_{FFT} ;
3. compute Fourier coefficients $W_k = \exp(-2i\pi k/N)$ (can be cached for the same input size);
4. obtain DCT of X as $X_{DCT} = \text{Real}(X_{FFT} * W)$ (multiplication by W accounts for permuting the input X).

Makhoul’s algorithm performs a one-dimensional, type-II DCT for each row of matrix G . In essence, the Makhoul’s algorithm $S = \text{MAKHOUL}(G)$ is equivalent to the matmul version $S = GQ$, where the matrix Q embeds all the operations performed by the Makhoul’s procedure.

We benchmark Makhoul’s algorithm against the matmul version on GPU for different layer sizes and show it is faster than matmul for large layers. For small layers (up to embedding size 2048, the benefits of Makhoul’s algorithm do not have practical benefits compared to matmul - they have similar runtimes). Concretely, we run 10 iterations of warmup before starting to measure the time for 100 runs on the GPU for both matmul version and Makhoul’s version, where G is initialized with random gaussian data in float32 and stays fixed during the benchmark.

In Table 4 we show our benchmarking results where gradient matrix G is in float32 format, compare against the DCT transform obtained via matmul by simply employing $S = GQ$ and compare against the Makhoul’s N -point algorithm. The column **Ratio** shows how much faster the Makhoul’s implementation is compared to the standard matmul. We can get up to $50\times$ speedup for matrices with more columns than rows.

Input size	Source Model	Matmul time (s)	Makhoul time (s)	Ratio (@ / FFT)
(4096, 4096)	Llama-2-7B	0.00267808	0.00033124	$8.09\times$
(25600, 5120)	Qwen3-32B	0.02746414	0.00259182	$10.60\times$
(5120, 25600)	Qwen3-32B	0.13069152	0.00262571	$49.77\times$

Table 4: Comparison of Matmul vs. Makhoul runtimes across different input sizes and models for float32. Ratio measures how much faster the Makhoul’s algorithm is compared to the standard matmul. Matmul is faster for ratio < 1 and Makhoul is faster when ratio > 1 .

While we can get a significant speedup for the case $R < C$ for float32, the real practical settings use bfloat16 instead of float32 and for this reason we also run our benchmark for bfloat16 as follows: we generate a random matrix G in float32 which will be used for Makhoul’s algorithm and the same matrix G will be converted to bfloat16. Moreover, the DCT matrix Q will also be stored in bfloat16 such that both operands in $S = GQ$ are in bfloat16. On the other hand, since Makhoul’s algorithm uses FFT from PyTorch, we are currently restricted to using float32 inputs because at the moment of developing this work PyTorch does not have the complex-bfloat16 type, where both real and imaginary parts are stored in bfloat16 (half precision is supported only for inputs with sizes power of two). As a result, we run Makhoul’s algorithm only in float32 and compare with matmul in bfloat16.

In Table 5 we show our benchmarking results where we compare matmul run with bfloat16 inputs against Makhoul’s algorithm run on float32 input. For $R \geq C$, matmul-bfloat16 is consistently faster than Makhoul’s algorithm. This is expected because bfloat16 type has higher throughput on GPUs than float32. However, we see that Makhoul’s algorithm is $3.5\times$ faster than matmul-bfloat16 for $R < C$, which is a much lower speedup compared to the results in Table 4 for float32.

Input size	Source Model	Matmul time (s)	Makhoul time (s)	Ratio (@ / FFT)
(4096, 4096)	Llama-2-7B	0.00018977	0.00034137	0.56x
(25600, 5120)	Qwen3-32B	0.00184539	0.00268176	0.69x
(5120, 25600)	Qwen3-32B	0.00968907	0.00273731	3.54x

Table 5: Comparison of Matmul (bfloat16) vs. Makhoul (float32) timings across different input sizes and models with different data types.

The FFT-based algorithm shows reduction in the running time in our benchmark, as the theory predicts (e.g. $O(n^2 \log(n))$ compared to $O(n^3)$) for float32, while the gains are limited when we compare against matmul-bfloat16. In order to benefit from the faster computation of Makhoul’s algorithm, we would need to convert bfloat16 matrices to float32, run the faster procedure, then convert back to bfloat16, which is not feasible because of the additional memory and computational overhead. However, when running training in mixed precision, the gradients are computed in bfloat16 and accumulated in a float32 buffer for precision reasons. Fortunately, we have access to this float32 buffer in the optimizer’s step function in PyTorch.

E PSEUDOCODE OF DCT-ADAMW OPTIMIZER

In this section we present the pseudocode of the DCT-AdamW optimizer, which uses our DCT-based dynamic column selection approach to create a dynamic, low-rank projection matrix use to factorize the gradient to a low-rank matrix.

Following LDAdamW (Robert et al., 2025), DCT-AdamW incorporates low-rank gradients from different subspaces and this requires rotating the momentum buffers using a rotation matrix R , where the second momentum buffer v_t is updated using a simpler rule compared to LDAdamW. DCT-AdamW also supports error feedback and it saves only the error induced by the low-rank compression.

Algorithm 2 DCT-AdamW (right projection)

```

1: Input:  $\beta_1, \beta_2, \epsilon, T, T_u, r$ 
2:  $m_0, v_0 \leftarrow 0_{n \times r}, 0_{n \times r}$ 
3:  $\Xi_1 \leftarrow 0_{n \times m}$   $\diamond$  error feedback (EF) buffer
4:  $Q \in \mathbb{R}^{m \times m}$   $\diamond$  DCT matrix
5:  $\mathcal{I}_{crt}, \mathcal{I}_{prev} \leftarrow 0_r, 0_r$   $\diamond$  column indices
6: for  $t = \{1, 2, \dots, T\}$  do
7:    $G_t \leftarrow \nabla_{\theta} f(\theta_t) + \Xi_t$ 
8:    $R \leftarrow \text{UPDATESUBSPACE}(G_t)$ 
9:    $g_t \leftarrow G_t \cdot Q_{crt}$   $\diamond$  projected gradient
10:   $\Xi_t \leftarrow G_t - g_t \cdot Q_{crt}^\top$   $\diamond$  update EF
11:   $m_t \leftarrow \beta_1 \cdot m_{t-1} \cdot R + (1 - \beta_1)g_t$ 
12:   $v_t \leftarrow \beta_2 |v_{t-1} \cdot R| + (1 - \beta_2)g_t^2$ 
13:   $\theta_{t+1} \leftarrow \theta_t - \eta_t \frac{\hat{m}_t}{\epsilon + \sqrt{v_t}} Q_{crt}^\top$ 
14: end for

```

Algorithm 3 Update Subspace procedure

```

1: procedure UPDATESUBSPACE( $G$ )
2:    $R \leftarrow I_{r \times r}$ 
3:   if  $t > 1$  then
4:      $\mathcal{I}_{prev} \leftarrow \mathcal{I}_{crt}$ 
5:   end if
6:   if  $(t = 1) \vee (t \bmod T_u = 0)$  then
7:      $S = \text{MAKHOUL}(G)$   $\triangleright$  or  $S = GQ$ 
8:      $\mathcal{I}_{crt} \leftarrow \text{RANKCOLS}(GQ, r)$ 
9:      $R \leftarrow Q_{prev}^\top \cdot Q_{crt}$ 
10:  end if
11:  return  $R$ 
12: end procedure

```

The sets $\mathcal{I}_{prev/crt}$ hold the indices of the r columns for the previous/current projections and $Q_{prev/crt}$ contain the columns from Q specified by the these two sets. T_u represents the subspace update interval (set to 200 for GaLore and to 1 for LDAdam). The procedure RANKCOLS ranks the columns of S , which is computed as $S = \text{MAKHOUL}(G)$ (see Appendix D) or $S = G_t \cdot Q$ as explained in Section 2.1. In order to make sure the momentum buffers integrate gradients from the same

subspaces, we need to rotate m_t and v_t using a rotation matrix R that first projects the momentum to the full-dimensional space using the previous projector and then projects it back to the current lower-dimensional subspace. We can perform this rotation directly in the r -dimensional space by multiplying the two projection matrices, resulting in $R \in \mathbb{R}^{r \times r}$. Rotating v_t might introduce negative values and we apply the absolute value function to force the non-negativity of v_t . The low-rank gradient g_t is computed by multiplying the full-rank gradient G_t with Q_{crt} and the full-rank update is obtained multiplying u_t by Q_{crt}^\top .

F PROJECTION ERRORS OF TRION AND DION

We train two Llama models with 30M parameters ($d_{\text{model}} = 640$) with rank $r = 128$ using Dion and Trion optimizers and save the gradients. For each set of gradients we simulate the operations of the other optimizer and record the metrics Δ_t^{dion} and Δ_t^{trion} . Our experiments showed no significant differences in the patterns for the two simulations and we plot the projection errors for the simulation that used gradients from the model trained with Dion optimizer. In Figure 1 we show the projection errors for the linear layers in the first transformer block, where Trion yields lower projection error than Dion and the trend is the same across all transformer blocks in the model. These plots explain why Trion has lower training and validation perplexity, as can be seen in Table 1. Moreover, the projection error seems to be constant for Dion, while for Trion the projection error has a decreasing trend for some layers, which once again shows the dynamic behavior of our column selection approach coupled with Newton-Schulz iteration.

G PRE-TRAINING WITH FRUGAL/FIRA

Pre-training with FRUGAL. We integrate the DCT matrix into the FRUGAL optimizer and compare against the SVD, RandPerm and Random projections. RandPerm uses a random permutation as a projection matrix, while Random generates a random, semi-orthogonal matrix. In Figure 4a we show the training loss for all these runs. In the zoomed-in plot for the last 5000 training steps we see that SVD projection recovers the performance full-rank AdamW, while the DCT projection is a good approximation of the SVD, which supports our claim. We present our numerical results in Table 6. We would like to emphasize the running time reduced by 1h 48m ($\approx 22.6\%$) compared to the SVD projection, as well as the memory usage reduced by about 2.2GB ($\approx 3.5\%$), while the increase in perplexity is less than one point. In comparison to RandPerm and Random projections, the runtime is on par, while train and validation perplexities are lower by approximately one point in the favor of DCT, illustrating again the benefit of our approach.

Pre-training with FIRA. We integrate DCT into FIRA optimizer and compare against SVD. In Figure 4b we show the training loss, where our focus is on the comparison between SVD and DCT projections. We observe that DCT consistently yields lower training loss compared to the SVD projection, which is also visible in the numerical results in Table 6, translated to lower perplexities in the favor of DCT. Moreover, the memory usage is smaller by 2GB ($\approx 3\%$) and the running time is lower by 1h 54m ($\approx 23.8\%$).

Table 6: Pre-training Results for AdamW, FRUGAL and FIRA with different projections using 20 tokens/parameter. DCT is a good approximation to SVD with lower runtime and memory. AdamW is the full-rank optimizer and is added for reference.

	AdamW	FRUGAL				FIRA	
		SVD	DCT	RandPerm	Random	SVD	DCT
Train PPL	15.55	15.35	15.63	16.52	17.02	19.37	18.88
Val. PPL	14.05	14.02	14.23	15.18	15.61	17.67	17.30
Mem. (GiB)	73.49	65.70	63.50	65.44	63.72	68.44	66.48
Time	7h 54m	9h 45m	7h 57m	7h 56m	7h 56m	9h 53m	7h 59m

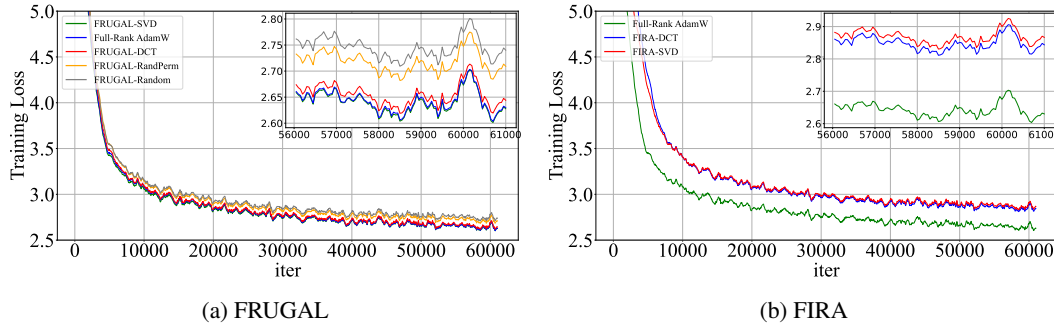


Figure 4: Pre-training Llama-800M using FRUGAL and FIRA on 16B tokens from C4.

H FINE-TUNING

We fine-tune Llama-2 7B on GSM-8k dataset using SVD and DCT projections for FRUGAL and FIRA optimizers, as well as LD-AdamW and DCT-AdamW. We show our results for ranks $r \in \{32, 512\}$ in Table 7. We also finetune Qwen-2.5-7B on GSM-8k using DCT-AdamW, GaLore and AdamW for reference and show the results in Table 8 for the same ranks.

Fine-tuning with FRUGAL. Both projections yield similar training loss for both rank values, which is an indication that DCT is indeed a good approximation for SVD. Despite having different training loss, the SVD projection achieves the same accuracy for both ranks. It is surprising that DCT yields better accuracy for lower rank compared to higher rank. In terms of memory usage, as expected for this large model, the DCT projection saves 8GB of memory ($\approx 23.4\%$) for $r = 32$, while for $r = 512$ the reduction in memory is only 6.3GB ($\approx 18.2\%$). The running time is reduced by roughly 35m ($\approx 75\%$) for both ranks.

Fine-tuning with FIRA. The DCT projection yields larger training loss compared to SVD. DCT recovers the accuracy, outperforms the SVD for large rank, and on average reduces the memory usage by ≈ 1.35 GB and the running by ≈ 10 m.

Fine-tuning with LDAdamW & DCT-AdamW. In this setting we compare the DCT projection with the block power-iteration used in LDAdamW as an approximation to SVD, both without EF. The training loss achieved is smaller for DCT on $r = 32$ and larger for $r = 512$ compared to LDAdamW. LDAdamW achieves more than 1% accuracy for $r = 32$ and comparable accuracy for $r = 512$. Regarding memory, DCT-AdamW uses ≈ 1 GB less memory, while achieving a speedup of about 20m for $r = 512$. We would like to emphasize that EF does not help DCT-AdamW in comparison to LDAdamW for $r = 32$, the accuracy of DCT-AdamW with EF being 29.11% vs 32.53% for LDAdamW with EF for $r = 32$, while for $r = 512$ the accuracy can be recovered, scoring 35.33% compared to 35.86% for LDAdamW with EF.

Table 7: Fine-tuning results for Llama-2 7B on the GSM-8K dataset.

	FRUGAL				FIRA				LD/DCT-AdamW			
	rank 32		rank 512		rank 32		rank 512		rank 32		rank 512	
	SVD	DCT	SVD	DCT	SVD	DCT	SVD	DCT	LD	DCT	LD	DCT
Train Loss	0.046	0.059	0.051	0.071	0.123	0.209	0.094	0.192	0.261	0.176	0.101	0.208
Acc. (%)	33.81	35.93	33.81	34.26	32.15	32.45	34.27	35.25	31.38	30.09	35.61	35.17
Mem. (GB)	39.61	31.59	40.68	34.41	32.72	31.29	35.41	34.11	32.08	31.17	35.58	34.35
Running Time	1h 22m	46m	1h 19m	47m	1h 8m	1h	1h 9m	1h	55m	48m	1h 8m	48m

Fine-tuning Qwen-2.5 7B. In this setting we compare the DCT projection with the original GaLore with our DCT-AdamW without EF, where both optimizers update the subspace once at 200 steps. In Table 8 shows that despite having slightly higher training loss, the DCT-AdamW has higher evaluation accuracy than GaLore.

Table 8: Fine-tuning results for Qwen-2.5-7B on AdamW, DCT-AdamW and GaLore on GSM-8k. DCT is a good approximation to SVD with lower runtime and memory. AdamW is the full-rank optimizer and is added for reference.

	AdamW	DCT-AdamW		GaLore	
	full rank	rank 32	rank 512	rank 32	rank 512
Train Loss	0.012313	0.053792	0.0545	0.0388	0.0184
Eval Acc (%)	67.70	64.59	65.35	63.23	64.41
Memory (GB)	64.5	40.4	44	40.9	45.15
Running Time	48m	49m	47m	58m	58m