

Projection Learning

Presented by:

Devansh Rathor 25M2153

Praveen 22B3931

Mangesh Poojan 25M0831

Gautam Mahale 24B0060

CS6013: Efficient AI

IIT Bombay

19th October, 2026



1. Compressing Neural Networks with the Hashing Trick
(Chen et al. 2015)
2. Random Offset Block Embedding (ROBE)
(Desai, Chou, and Shrivastava 2022)
3. Random Operation Access Specific Tile (ROAST)
(Desai, Zhou, and Shrivastava 2023)
4. Stability of STABLE-RPS
(Desai and Shrivastava 2024)
5. SS1: Sketch Structured Transform
(Saedi et al. 2024)

Section 1

Compressing Neural Networks with the Hashing Trick (Chen et al. 2015)



- Neural Network are scaled in parameters to increase its expressivity and capabilities.



- Neural Network are scaled in parameters to increase its expressivity and capabilities.
- Transmitting raw sensor/voice data over the network introduces latency, consumes bandwidth, and raises privacy issues.



- Neural Network are scaled in parameters to increase its expressivity and capabilities.
- Transmitting raw sensor/voice data over the network introduces latency, consumes bandwidth, and raises privacy issues.
- Simply shrinking layer width significantly hurts representational capacity and degrades generalization accuracy.



- Neural Network are scaled in parameters to increase its expressivity and capabilities.
- Transmitting raw sensor/voice data over the network introduces latency, consumes bandwidth, and raises privacy issues.
- Simply shrinking layer width significantly hurts representational capacity and degrades generalization accuracy.
- Prior studies demonstrate that modern deep neural network weights contain heavy parameter redundancy (Denil et al. 2013)



*HashedNets: A **small** model that behaves like a **large** model*



- Present **HashedNets**, that exploits inherent redundancy in neural networks to achieve drastic reductions in model sizes.



- Present **HashedNets**, that exploits inherent redundancy in neural networks to achieve drastic reductions in model sizes.
- Use a **low-cost hash function** to randomly group connection weights into **hash buckets**,



- Present **HashedNets**, that exploits inherent redundancy in neural networks to achieve drastic reductions in model sizes.
- Use a **low-cost hash function** to randomly group connection weights into **hash buckets**,
- All connections within the same hash bucket share a **single parameter value**.



- Present **HashedNets**, that exploits inherent redundancy in neural networks to achieve drastic reductions in model sizes.
- Use a **low-cost hash function** to randomly group connection weights into **hash buckets**,
- All connections within the same hash bucket share a **single parameter value**.
- The hashing procedure introduces **no additional memory overhead**, such as auxiliary indexing tables.



- Present **HashedNets**, that exploits inherent redundancy in neural networks to achieve drastic reductions in model sizes.
- Use a **low-cost hash function** to randomly group connection weights into **hash buckets**,
- All connections within the same hash bucket share a **single parameter value**.
- The hashing procedure introduces **no additional memory overhead**, such as auxillary indexing tables.
- HashedNets weight sharing architecture uses **standard backpropagation** during training



- A mapping function $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k$ maps an input vector $x \in \mathbb{R}^d$ into a feature space $\phi(x) \in \mathbb{R}^k$, where $k \ll d$



- A mapping function $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k$ maps an input vector $x \in \mathbb{R}^d$ into a feature space $\phi(x) \in \mathbb{R}^k$, where $k \ll d$
- The mapping ϕ is based on two (approximately uniform) hash functions:

$$h : \mathbb{N} \rightarrow \{1, \dots, k\} \quad \xi : \mathbb{N} \rightarrow \{-1, 1\}$$



- A mapping function $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k$ maps an input vector $x \in \mathbb{R}^d$ into a feature space $\phi(x) \in \mathbb{R}^k$, where $k \ll d$
- The mapping ϕ is based on two (approximately uniform) hash functions:

$$h : \mathbb{N} \rightarrow \{1, \dots, k\} \quad \xi : \mathbb{N} \rightarrow \{-1, 1\}$$

- the k^{th} dimension of the hashed input x is defined as:

$$[\phi(x)]_k = \sum_{i:h(i)=k} x_i \xi(i)$$



- A mapping function $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k$ maps an input vector $x \in \mathbb{R}^d$ into a feature space $\phi(x) \in \mathbb{R}^k$, where $k \ll d$
- The mapping ϕ is based on two (approximately uniform) hash functions:

$$h : \mathbb{N} \rightarrow \{1, \dots, k\} \quad \xi : \mathbb{N} \rightarrow \{-1, 1\}$$

- the k^{th} dimension of the hashed input x is defined as:

$$[\phi(x)]_k = \sum_{i:h(i)=k} x_i \xi(i)$$

- Incorporating the sign hash function ξ guarantees that inner products remain unbiased in expectation, as $\mathbb{E}[\xi] = 0$:

$$\mathbb{E}[\phi(x)^\top \phi(x')] = x^\top x'$$

Virtual Weights

- Rather than pruning layers, HashedNets retains full-sized virtual weight matrices $V^l \in \mathbb{R}^{n^{l+1} \times n^l}$

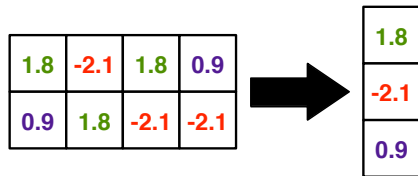


Figure: Virtual Weights $V^l \Rightarrow w^l$
"Real" Weights

¹<https://code.google.com/archive/p/xxhash/>

Virtual Weights

- Rather than pruning layers, HashedNets retains full-sized virtual weight matrices $V^l \in \mathbb{R}^{n^{l+1} \times n^l}$
- Only a compact vector of K^l "real" weights $w^l \in \mathbb{R}^{K^l}$ is stored in memory ($K^l \ll n^{l+1} \times n^l$)

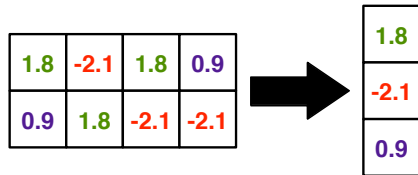


Figure: Virtual Weights $V^l \Rightarrow w^l$
"Real" Weights

¹<https://code.google.com/archive/p/xxhash/>

Virtual Weights

- Rather than pruning layers, HashedNets retains full-sized virtual weight matrices $V^l \in \mathbb{R}^{n^{l+1} \times n^l}$
- Only a compact vector of K^l "real" weights $w^l \in \mathbb{R}^{K^l}$ is stored in memory ($K^l \ll n^{l+1} \times n^l$)
- Mapping uses two hash functions using xxHash¹:

$$h^l(i, j) \in \{1, \dots, K\} \quad \xi^l(i, j) \in \{-1, 1\}$$

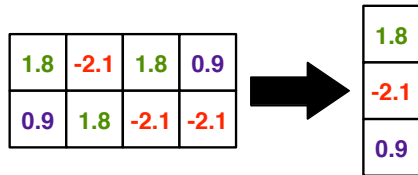


Figure: Virtual Weights $V^l \Rightarrow w^l$
"Real" Weights

¹<https://code.google.com/archive/p/xxhash/>

Virtual Weights

- Rather than pruning layers, HashedNets retains full-sized virtual weight matrices $V^l \in \mathbb{R}^{n^{l+1} \times n^l}$
- Only a compact vector of K^l "real" weights $w^l \in \mathbb{R}^{K^l}$ is stored in memory ($K^l \ll n^{l+1} \times n^l$)
- Mapping uses two hash functions using xxHash¹:

$$h^l(i, j) \in \{1, \dots, K\} \quad \xi^l(i, j) \in \{-1, 1\}$$

- Resulting in the following virtual weights:

$$V_{ij}^l = w_{h^l(i, j)}^l \cdot \xi^l(i, j)$$

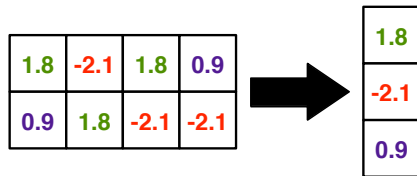


Figure: Virtual Weights $V^l \Rightarrow w^l$
"Real" Weights

¹<https://code.google.com/archive/p/xxhash/>



- **Weight Sharing:** The virtual weight matrix entries are tied to a small, K -dimensional weight vector w , we can compute a single layer forward pass as:

$$z_i = \sum_{j=1}^m w_{h(i,j)} a_j$$



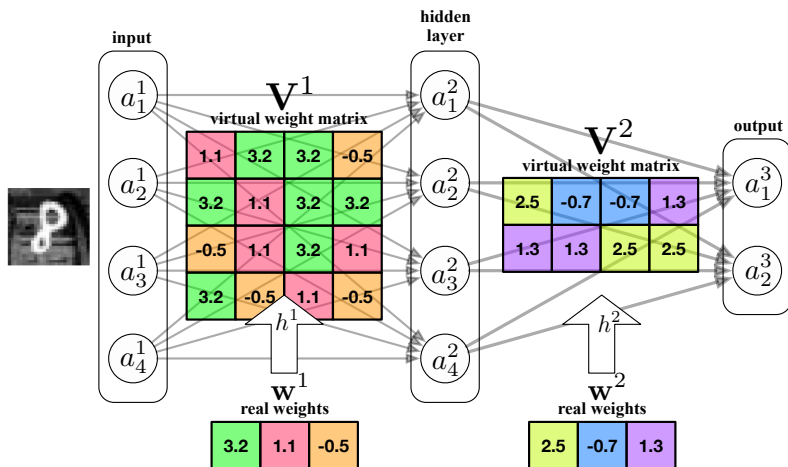
- **Weight Sharing:** The virtual weight matrix entries are tied to a small, K -dimensional weight vector w , we can compute a single layer forward pass as:

$$z_i = \sum_{j=1}^m w_{h(i,j)} a_j$$

- **Feature Hashing:** The layer can be viewed as hashing/projecting the incoming activation vector, where for each output neuron i , define a mapping $\phi_i(\cdot) : \mathbb{R}^m \rightarrow \mathbb{R}^K$ with,

$$[\phi_i(a)]_k = \sum_{j:h(i,j)=k} a_j$$

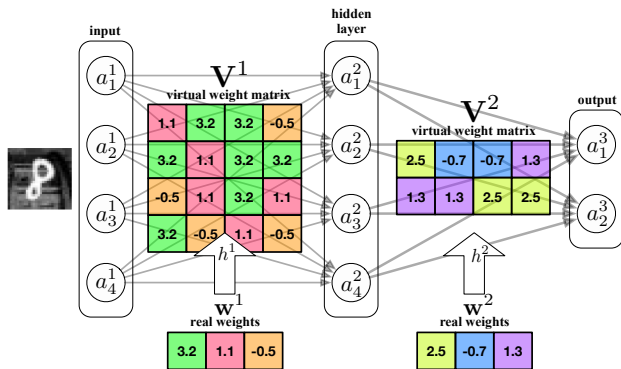
Then, the pre-activation value: $z_i = w^\top \phi_i(a)$



- Forward Pass:

$$\begin{aligned}
 a_i^{l+1} &= f(z_i^{l+1}) \\
 &= f\left(\sum_{j=1}^{n^l} V_{i,j}^l a_j^l\right) \\
 &= f\left(\sum_{j=1}^{n^l} w_{h^l(i,j)}^l \xi^l(i,j) a_j^l\right)
 \end{aligned}$$

Where, $f(\cdot)$ is the transition function (eg. ReLU)

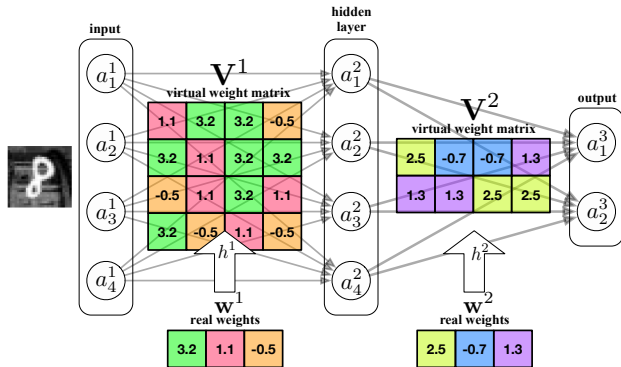


- Backpropagation:

$$\delta_j^l = \left(\sum_{i=1}^{n^{l+1}} V_{i,j}^l \delta_i^{l+1} \right) f'(z_j^l)$$

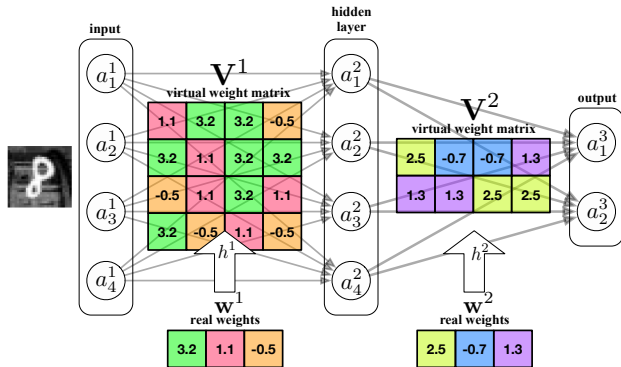
$$= \left(\sum_{i=1}^{n^{l+1}} w_{h^l(i,j)}^l \xi^l(i,j) \delta_i^{l+1} \right) f'(z_j^l)$$

where δ_j^l denote the gradient of loss $\mathcal{L}$ over activation j in layer l .



- Gradients for each shared weight w_k^l accumulate all virtual connections tied to hash bucket k :

$$\frac{\partial \mathcal{L}}{\partial w_k^l} = \sum_{i=1}^{n^{l+1}} \sum_{j=1}^{n^l} a_j^l \delta_i^{l+1} \xi^l(i, j) \cdot \mathbb{I}_{[h^l(i, j)=k]}$$



Experimental Setup

Datasets (8 Benchmarks)



- Original MNIST handwritten digits
- Rotated digits (ROT)
- Random background (BG-RAND)
- Image background (BG-IMG)
- Combined (BG-IMG-ROT)
- Binary classification benchmarks: RECT and CONVEX



- **Random Edge Removal (RER):** Prunes connections randomly prior to training.
- **Low-Rank Decomposition (LRD):** Factorizes weight matrices into fixed random and learned low-rank components.
- **Size-Equivalent Neural Network (NN):** Shrinks hidden unit width directly to match parameter constraints.
- **Dark Knowledge (DK):** Distills soft targets from a full uncompressed teacher network.
- **HashNet_{DK}:** HashedNets trained jointly with hard labels and teacher soft targets.

Experimental Results

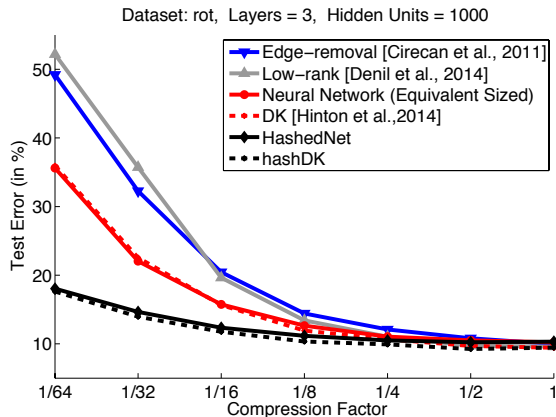
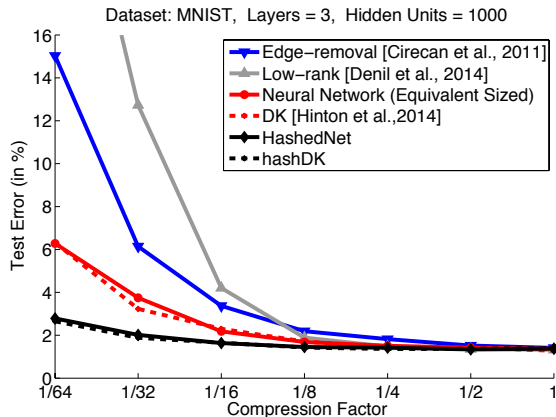
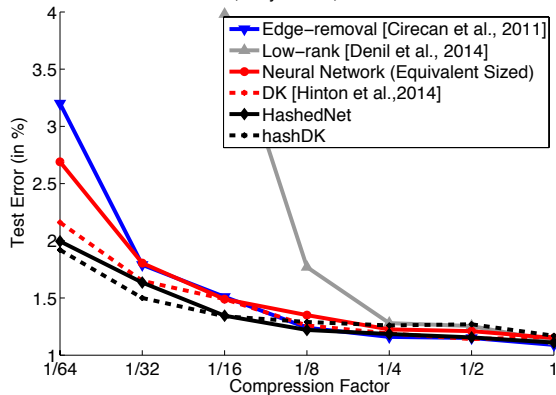


Figure: Test error rates under varying compression factors with 3-layer networks on MNIST (left) and ROT (right).

Experimental Results



Dataset: MNIST, Layers = 5, Hidden Units = 1000



Dataset: rot, Layers = 5, Hidden Units = 1000

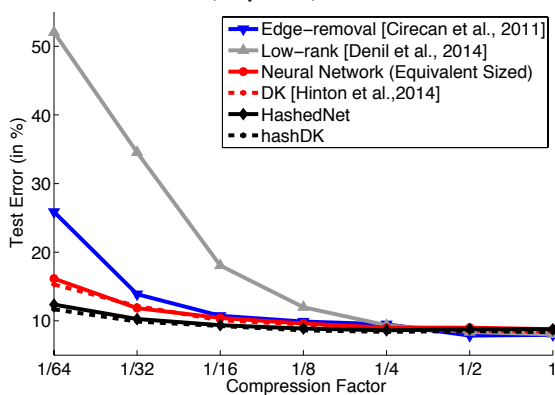


Figure: Test error rates under varying compression factors with 5-layer networks on MNIST (left) and ROT (right).

	3 Layers						5 Layers					
	RER	LRD	NN	DK	HashNet	HashNet _{DK}	RER	LRD	NN	DK	HashNet	HashNet _{DK}
MNIST	15.03	28.99	6.28	6.32	2.79	2.65	3.20	28.11	2.69	2.16	1.99	1.92
BASIC	13.95	26.95	7.67	8.44	4.17	3.79	5.31	27.21	4.55	4.07	3.49	3.19
ROT	49.20	52.18	35.60	35.94	18.04	17.62	25.87	52.03	16.16	15.30	12.38	11.67
BG-RAND	44.90	76.21	43.04	53.05	21.50	20.32	90.28	76.21	16.60	14.57	16.37	13.76
BG-IMG	44.34	71.27	32.64	41.75	26.41	26.17	55.76	70.85	22.77	23.59	22.22	20.01
BG-IMG-ROT	73.17	80.63	79.03	77.40	59.20	58.25	88.88	80.93	53.18	53.19	51.93	54.51
RECT	37.22	39.93	34.37	31.85	31.77	30.43	50.00	39.65	29.76	26.95	29.70	32.04
CONVEX	18.23	23.67	5.68	5.78	3.67	3.37	50.03	23.95	4.28	3.10	5.67	2.64

Table: Test error rates (in %) with a compression factor of $\frac{1}{64}$ across all data sets. Best results are printed in **blue**.

Experimental Results

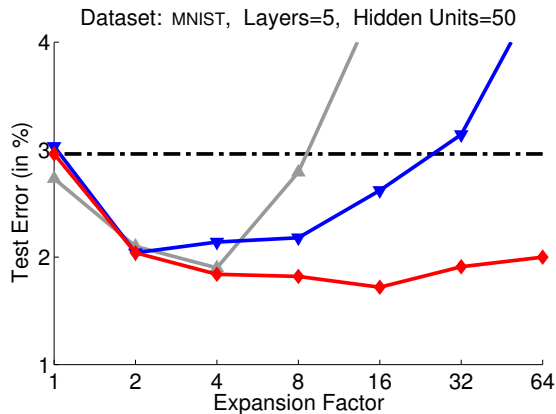
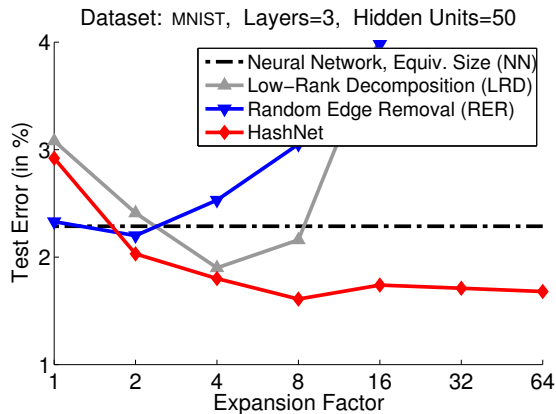


Figure: Test error rates with fixed storage but varying expansion factors on MNIST with 3 layers (left) and 5 layers (right).



Summary:

- HashedNets provides a simple, storage-free compression technique using runtime hashing.
- Allows neural networks to dramatically expand virtual representational capacity without increasing model file size.
- Outperforms structural rank reduction and layer thinning under aggressive memory budgets.



Summary:

- HashedNets provides a simple, storage-free compression technique using runtime hashing.
- Allows neural networks to dramatically expand virtual representational capacity without increasing model file size.
- Outperforms structural rank reduction and layer thinning under aggressive memory budgets.

Practical Limitations & Future Directions:

- Hashing creates pseudo-random memory index lookups, which lead to non-coalesced memory access and test-time latency on GPU architectures.
- Combining HashedNets with post-training integer quantization and knowledge distillation for embedded inference engines

Section 2

Random Offset Block Embedding (ROBE) (Desai, Chou, and Shrivastava 2022)



- Recommendation inference accounts for **over 79% of AI inference** cycles at hyperscale data centers.



- Recommendation inference accounts for **over 79% of AI inference** cycles at hyperscale data centers.
- Recommendation data is dominated by high-cardinality categorical features, producing embedding tables that consume **over 99.9% of model memory ($\sim 100\text{TB}$)**.



- Recommendation inference accounts for **over 79% of AI inference** cycles at hyperscale data centers.
- Recommendation data is dominated by high-cardinality categorical features, producing embedding tables that consume **over 99.9% of model memory ($\sim 100\text{TB}$)**.
- Tables cannot fit on a single GPU, forcing models into **mixed model and data-parallel** distributed training across clusters with **high network communication costs**.



- Recommendation inference accounts for **over 79% of AI inference** cycles at hyperscale data centers.
- Recommendation data is dominated by high-cardinality categorical features, producing embedding tables that consume **over 99.9% of model memory ($\sim 100\text{TB}$)**.
- Tables cannot fit on a single GPU, forcing models into **mixed model and data-parallel** distributed training across clusters with **high network communication costs**.
- Lookups exhibit sparse, irregular access patterns leading to **severe cache miss rates**, causing inference systems to spend **up to 80% of execution cycles on embedding lookups** alone.



- Recommendation inference accounts for **over 79% of AI inference** cycles at hyperscale data centers.
- Recommendation data is dominated by high-cardinality categorical features, producing embedding tables that consume **over 99.9% of model memory ($\sim 100\text{TB}$)**.
- Tables cannot fit on a single GPU, forcing models into **mixed model and data-parallel** distributed training across clusters with **high network communication costs**.
- Lookups exhibit sparse, irregular access patterns leading to **severe cache miss rates**, causing inference systems to spend **up to 80% of execution cycles on embedding lookups** alone.
- Existing techniques achieve moderate reductions, while naive element-wise hashing (HashedNet) destroys memory bandwidth and cache locality.



An algorithmic weight-sharing technique using universal hash functions over contiguous memory blocks, reducing DLRM embedding table footprints by $1000\times$ while accelerating inference throughput by up to $3.1\times$.



An algorithmic weight-sharing technique using universal hash functions over contiguous memory blocks, reducing DLRM embedding table footprints by $1000\times$ while accelerating inference throughput by up to $3.1\times$.

- Eliminates individual embedding tables by sharing a single, compact, circular parameter array $\mathcal{M}$ ($|\mathcal{M}| \ll |S| \times D$) across all categorical features and tables.



An algorithmic weight-sharing technique using universal hash functions over contiguous memory blocks, reducing DLRM embedding table footprints by $1000\times$ while accelerating inference throughput by up to $3.1\times$.

- Eliminates individual embedding tables by sharing a single, compact, circular parameter array $\mathcal{M}$ ($|\mathcal{M}| \ll |S| \times D$) across all categorical features and tables.
- Instead of hashing individual scalar weights, parameters are assigned to contiguous memory chunks of size Z



An algorithmic weight-sharing technique using universal hash functions over contiguous memory blocks, reducing DLRM embedding table footprints by $1000\times$ while accelerating inference throughput by up to $3.1\times$.

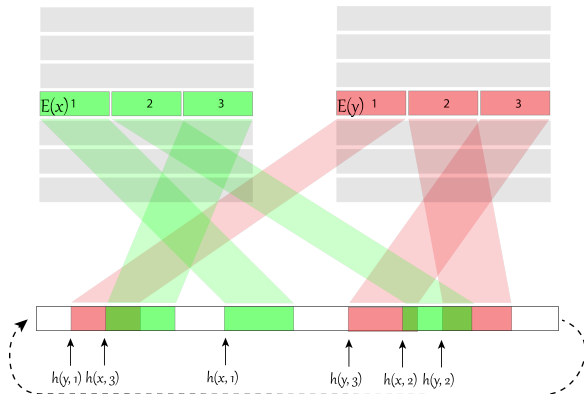
- Eliminates individual embedding tables by sharing a single, compact, circular parameter array $\mathcal{M}$ ($|\mathcal{M}| \ll |S| \times D$) across all categorical features and tables.
- Instead of hashing individual scalar weights, parameters are assigned to contiguous memory chunks of size Z
- Consecutive vector elements map to adjacent physical addresses, enabling coalesced reads that fully saturate hardware cache lines.



An algorithmic weight-sharing technique using universal hash functions over contiguous memory blocks, reducing DLRM embedding table footprints by $1000\times$ while accelerating inference throughput by up to $3.1\times$.

- Eliminates individual embedding tables by sharing a single, compact, circular parameter array $\mathcal{M}$ ($|\mathcal{M}| \ll |S| \times D$) across all categorical features and tables.
- Instead of hashing individual scalar weights, parameters are assigned to contiguous memory chunks of size Z
- Consecutive vector elements map to adjacent physical addresses, enabling coalesced reads that fully saturate hardware cache lines.
- Fixed intra-block relative offsets guarantee that elements inside the same block never collide, strictly lowering estimation variance below standard feature hashing.

Splits an embedding of size D into chunks of size Z , independently hashing each chunk and concatenating them: $P(x) = P_{c_1} \parallel \dots \parallel P_{c_{D/Z}}$



Splits an embedding of size D into chunks of size Z , independently hashing each chunk and concatenating them: $P(x) = P_{c_1} \parallel \dots \parallel P_{c_{D/Z}}$

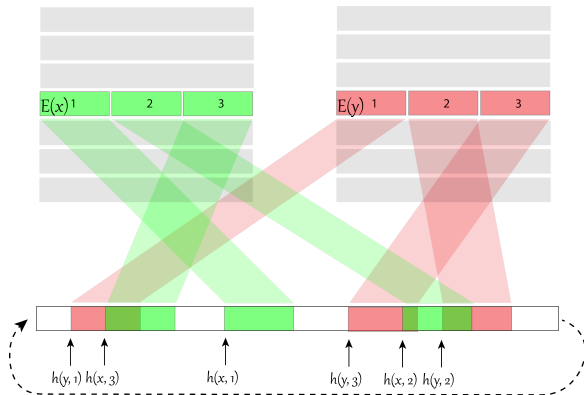
Uses three hash uniform and efficient functions:

$$h: \mathbb{N}^3 \rightarrow \{1, \dots, |\mathcal{M}|\}$$

$$\text{idx}: \mathbb{N}^3 \rightarrow \{1, \dots, Z\}$$

$$g: \mathbb{N}^3 \rightarrow \{-1, +1\}$$

to map each Z -sized chunk of the embedding $E(x) = P(x) \circ G(x)$ to chunks in $\mathcal{M}$.





$$\begin{aligned}h(e, x, c) &= (A_h e + B_h x + C_h c + D) \bmod P \bmod |\mathcal{M}| \\ \text{idx}(e, x, i) &= (h(e, x, \mathcal{Z}_{\text{id}}(i)) + \mathcal{O}_{\text{id}}(i)) \bmod |\mathcal{M}| \\ g(e, x, i) &= 2((A_g e + B_g x + C_g c + D) \bmod P \bmod 2) - 1\end{aligned}$$



$$\begin{aligned}h(e, x, c) &= (A_h e + B_h x + C_h c + D) \bmod P \bmod |\mathcal{M}| \\ \text{idx}(e, x, i) &= (h(e, x, \mathcal{Z}_{\text{id}}(i)) + \mathcal{O}_{\text{id}}(i)) \bmod |\mathcal{M}| \\ g(e, x, i) &= 2((A_g e + B_g x + C_g c + D) \bmod P \bmod 2) - 1\end{aligned}$$

where P is a large prime,



$$h(e, x, c) = (A_h e + B_h x + C_h c + D) \bmod P \bmod |\mathcal{M}|$$

$$\text{idx}(e, x, i) = (h(e, x, \mathcal{Z}_{\text{id}}(i)) + \mathcal{O}_{\text{id}}(i)) \bmod |\mathcal{M}|$$

$$g(e, x, i) = 2((A_g e + B_g x + C_g c + D) \bmod P \bmod 2) - 1$$

where P is a large prime,

$$\begin{aligned} A_h, B_h, C_h, A_g, B_g, C_g &\in \{1, \dots, P\}, \\ D_h, D_g &\in \{0, \dots, P\}, \end{aligned}$$



$$\begin{aligned}h(e, x, c) &= (A_h e + B_h x + C_h c + D) \bmod P \bmod |\mathcal{M}| \\ \text{idx}(e, x, i) &= (h(e, x, \mathcal{Z}_{\text{id}}(i)) + \mathcal{O}_{\text{id}}(i)) \bmod |\mathcal{M}| \\ g(e, x, i) &= 2((A_g e + B_g x + C_g c + D) \bmod P \bmod 2) - 1\end{aligned}$$

where P is a large prime,

$$\begin{aligned}A_h, B_h, C_h, A_g, B_g, C_g &\in \{1, \dots, P\}, \\ D_h, D_g &\in \{0, \dots, P\},\end{aligned}$$

and the chunk index and intra-chunk offset are given by:

$$\mathcal{Z}_{\text{id}}(i) = \left\lfloor \frac{i}{Z} \right\rfloor \quad \mathcal{O}_{\text{id}}(i) = i \bmod Z.$$



$$h(e, x, c) = (A_h e + B_h x + C_h c + D) \bmod P \bmod |\mathcal{M}|$$

$$\text{idx}(e, x, i) = (h(e, x, \mathcal{Z}_{\text{id}}(i)) + \mathcal{O}_{\text{id}}(i)) \bmod |\mathcal{M}|$$

$$g(e, x, i) = 2((A_g e + B_g x + C_g c + D) \bmod P \bmod 2) - 1$$

- Because it is drawn from a universal hash family, the pairwise collision probability between distinct blocks is bounded by $1/|\mathcal{M}|$



$$\begin{aligned}h(e, x, c) &= (A_h e + B_h x + C_h c + D) \bmod P \bmod |\mathcal{M}| \\ \text{idx}(e, x, i) &= (h(e, x, \mathcal{Z}_{\text{id}}(i)) + \mathcal{O}_{\text{id}}(i)) \bmod |\mathcal{M}| \\ g(e, x, i) &= 2((A_g e + B_g x + C_g c + D) \bmod P \bmod 2) - 1\end{aligned}$$

- Because it is drawn from a universal hash family, the pairwise collision probability between distinct blocks is bounded by $1/|\mathcal{M}|$
- Computing h requires only standard linear congruential operations, avoiding branching and minimizing register pressure on GPU execution units.



$$\begin{aligned}h(e, x, c) &= (A_h e + B_h x + C_h c + D) \bmod P \bmod |\mathcal{M}| \\ \text{idx}(e, x, i) &= (h(e, x, \mathcal{Z}_{\text{id}}(i)) + \mathcal{O}_{\text{id}}(i)) \bmod |\mathcal{M}| \\ g(e, x, i) &= 2((A_g e + B_g x + C_g c + D) \bmod P \bmod 2) - 1\end{aligned}$$

- Because it is drawn from a universal hash family, the pairwise collision probability between distinct blocks is bounded by $1/|\mathcal{M}|$
- Computing h requires only standard linear congruential operations, avoiding branching and minimizing register pressure on GPU execution units.
- Incorporating the embedding table ID e directly into the linear hash allows distinct tables to share the exact same physical memory buffer $\mathcal{M}$.



ROBE vs HashedNet

Memory Fetches Across Block Sizes

Scheme	Condition	Max Fetches
Original	$B \mid D$	$D/B + 1$
HashedNet / ROBE-1	—	D
ROBE-Z ($Z < B < D$)	$Z \mid B \mid D$	$2 \times D/Z^*$
ROBE-Z ($B < Z < D$)	$B \mid Z \mid D$	$D/B + D/Z^*$
ROBE-Z ($Z > D$)	$D \mid Z$	$D/B + 2$

*With high probability, assuming $|\mathcal{M}| \gg D > Z$.



ROBE vs HashedNet

Memory Fetches Across Block Sizes

Scheme	Condition	Max Fetches
Original	$B \mid D$	$D/B + 1$
HashedNet / ROBE-1	—	D
ROBE-Z ($Z < B < D$)	$Z \mid B \mid D$	$2 \times D/Z^*$
ROBE-Z ($B < Z < D$)	$B \mid Z \mid D$	$D/B + D/Z^*$
ROBE-Z ($Z > D$)	$D \mid Z$	$D/B + 2$

*With high probability, assuming $|\mathcal{M}| \gg D > Z$.

- For bus/cache line size B and embedding dimension D , naive scalar hashing ($Z = 1$ / HashedNet) requires up to D separate memory transactions.



ROBE vs HashedNet

Memory Fetches Across Block Sizes

Scheme	Condition	Max Fetches
Original	$B \mid D$	$D/B + 1$
HashedNet / ROBE-1	—	D
ROBE-Z ($Z < B < D$)	$Z \mid B \mid D$	$2 \times D/Z^*$
ROBE-Z ($B < Z < D$)	$B \mid Z \mid D$	$D/B + D/Z^*$
ROBE-Z ($Z > D$)	$D \mid Z$	$D/B + 2$

*With high probability, assuming $|\mathcal{M}| \gg D > Z$.

- For bus/cache line size B and embedding dimension D , naive scalar hashing ($Z = 1$ / HashedNet) requires up to D separate memory transactions.
- Setting block size $Z \geq D$ reduces the worst-case cache-line fetches to just $D/B + 2$, eliminating wasted bus bandwidth.



ROBE vs HashedNet

Memory Fetches Across Block Sizes

Scheme	Condition	Max Fetches
Original	$B \mid D$	$D/B + 1$
HashedNet / ROBE-1	—	D
ROBE-Z ($Z < B < D$)	$Z \mid B \mid D$	$2 \times D/Z^*$
ROBE-Z ($B < Z < D$)	$B \mid Z \mid D$	$D/B + D/Z^*$
ROBE-Z ($Z > D$)	$D \mid Z$	$D/B + 2$

*With high probability, assuming $|\mathcal{M}| \gg D > Z$.

- For bus/cache line size B and embedding dimension D , naive scalar hashing ($Z = 1$ / HashedNet) requires up to D separate memory transactions.
- Setting block size $Z \geq D$ reduces the worst-case cache-line fetches to just $D/B + 2$, eliminating wasted bus bandwidth.
- Compressing embedding tables by a factor of $1000\times$ allows substantial portions of the embedding space to reside directly within the LLC or GPU HBM

Comparison with Compression Baselines



Table: Embedding table compression ratio and predictive quality across baseline methods on Criteo benchmarks.

Method	Dataset	Compression	Model Quality	Metric
Low-Precision (Zhang, Yang, and Yuen 2018)	Criteo Kaggle	2×	Similar to baseline	Logloss
Hashing Trick (Weinberger et al. 2009)	Criteo Kaggle	4×	Much worse than QR Trick	Logloss
QR Trick (Shi et al. 2020)	Criteo Kaggle	4×	Slightly worse	Logloss
MD Embeddings (Ginart et al. 2021)	Criteo Kaggle	16×	Better or similar	Logloss
TT-Rec (Yin et al. 2021)	Criteo Kaggle	112×	Better or similar	Accuracy / Logloss
TT-Rec (Yin et al. 2021)	Criteo TB	117×	Better or similar	Accuracy / Logloss
ROBE-Z	Criteo Kaggle	1000×	Better or similar	AUC / Logloss
ROBE-Z	Criteo TB	1000×	Better or similar	AUC / Logloss



Table: Test AUC across model architectures: 1000 $\times$ compressed ROBE-Z vs. uncompressed baseline on Criteo Kaggle.

Model	Original	ROBE-1 (avg)	ROBE-1 (stdev)	ROBE-2	ROBE-4	ROBE-8	ROBE-16
DLRM	0.8031	0.8032	0.0001	0.8050	0.8049	0.8047	0.8050
DCN	0.7973	0.7991	0.0004	0.7994	0.7995	0.7994	0.7993
AutoInt	0.7968	0.7987	0.0002	0.7984	0.7984	0.7988	0.7985
DeepFM	0.7957	0.7951	0.0001	0.7949	0.7949	0.7947	0.7950
xDeepFM	0.8007	0.7989	0.0004	0.7987	0.7988	0.7990	0.7991
FiBiNET	0.8016	0.8011	0.0002	0.8011	0.8010	0.8013	0.8012



ROBE

Random Offset Block Embedding

Why hash blocks instead of individual numbers?

Recommendation models store huge embedding tables. ROBE maps their parameters into a much smaller circular array while keeping neighbouring coordinates together.



Focus: Theorem 1 (inner-product sketch) and Theorem 2 (embedding relationships)

Presented by Praveen

Theorem 1 — block hashing reduces sketch variance

Setup: $x, y \in \mathbb{R}^n$, $Z < m < n$, uniform random start cell per block, independent signs $g(i)$.

$$\hat{x}_j = \sum_i x_i g(i) \mathbf{1}[\text{idx}(i) = j], \quad \widehat{\langle x, y \rangle} = \sum_{j=1}^m \hat{x}_j \hat{y}_j$$

$$\mathbb{E}[\widehat{\langle x, y \rangle}] = \langle x, y \rangle$$

$$\text{Var}(\widehat{\langle x, y \rangle}) = \frac{1}{m} \sum_{\text{blk}(i) \neq \text{blk}(r)} (x_i^2 y_r^2 + x_i y_i x_r y_r)$$

Takeaway Unbiased, and within-block pairs drop out of the variance. $Z = 1$ gives feature hashing.

Proof of Theorem 1 (1/5) — expand the estimator

Step 1: Expand, including the sum over j

$$\begin{aligned}\widehat{\langle x, y \rangle} &= \sum_{j=1}^m \left(\sum_i x_i g(i) \mathbf{1}[\text{idx}(i) = j] \right) \left(\sum_r y_r g(r) \mathbf{1}[\text{idx}(r) = j] \right) \\ &= \sum_{i,r} x_i y_r g(i) g(r) \sum_{j=1}^m \mathbf{1}[\text{idx}(i) = j] \mathbf{1}[\text{idx}(r) = j] \\ &= \sum_{i,r} I_{ir} g(i) g(r) x_i y_r, \quad I_{ir} = \mathbf{1}[\text{idx}(i) = \text{idx}(r)]\end{aligned}$$

Step 2: Signal + noise ($i = r$: $I_{ii} = 1$, $g(i)^2 = 1$)

$$\widehat{\langle x, y \rangle} = \langle x, y \rangle + \mathcal{E}, \quad \mathcal{E} = \sum_{i \neq r} I_{ir} g(i) g(r) x_i y_r$$

Proof of Theorem 1 (2/5) — blocking and expectation

Step 3: Same-block pairs cannot collide

$$\text{idx}(i) = (h(c) + o(i)) \bmod m, \quad o(i) \in \{0, \dots, Z-1\}$$

$$i \neq r \text{ in one block: } \text{idx}(i) - \text{idx}(r) \equiv o(i) - o(r), \quad 0 < |o(i) - o(r)| < Z < m \Rightarrow I_{ir} = 0$$

$$\mathcal{E} = \sum_{\text{blk}(i) \neq \text{blk}(r)} I_{ir} g(i) g(r) x_i y_r$$

Step 4: Expectation (g independent of I_{ir})

$$\mathbb{E}[I_{ir} g(i) g(r) x_i y_r] = x_i y_r \mathbb{E}[I_{ir}] \underbrace{\mathbb{E}[g(i)] \mathbb{E}[g(r)]}_0 = 0 \Rightarrow \mathbb{E}[\mathcal{E}] = 0$$

$$\boxed{\mathbb{E}[\widehat{\langle x, y \rangle}] = \langle x, y \rangle}$$

Proof of Theorem 1 (3/5) — variance: expand $\mathcal{E}^2$

Step 5: Variance = second moment

$$\text{Var}(\widehat{\langle x, y \rangle}) = \mathbb{E}[\mathcal{E}^2]$$

$$\mathcal{E}^2 = \sum_{(i,r)} \sum_{(p,q)} I_{ir} I_{pq} g(i)g(r)g(p)g(q) x_i y_r x_p y_q$$

$$\mathbb{E}[\mathcal{E}^2] = \sum_{(i,r)} \sum_{(p,q)} x_i y_r x_p y_q \mathbb{E}[I_{ir} I_{pq}] \mathbb{E}[g(i)g(r)g(p)g(q)]$$

Step 6a: Sign pairing

$$\mathbb{E}[g(i)g(r)g(p)g(q)] = \begin{cases} 1 & \text{every index appears an even number of times} \\ 0 & \text{otherwise} \end{cases}$$

$$i \neq r, p \neq q \implies (p,q) = (i,r) \text{ or } (p,q) = (r,i)$$

Proof of Theorem 1 (4/5) — surviving terms, $\Pr = 1/m$

Step 6b: The two surviving terms

$$\begin{aligned}(p, q) = (i, r) : \quad g(i)^2 g(r)^2 = 1, \quad I_{ir} I_{ir} = I_{ir} &\Rightarrow x_i^2 y_r^2 \mathbb{E}[I_{ir}] \\(p, q) = (r, i) : \quad g(i)^2 g(r)^2 = 1, \quad I_{ir} I_{ri} = I_{ir} &\Rightarrow x_i y_i x_r y_r \mathbb{E}[I_{ir}]\end{aligned}$$

Step 6c: Collision probability (i in block c , offset o ; r in block $c' \neq c$, offset o')

$$I_{ir} = 1 \iff h(c) \equiv h(c') + (o' - o) \pmod{m}$$

$$\Pr[I_{ir} = 1] = \sum_{v=0}^{m-1} \Pr[h(c') = v] \Pr[h(c) \equiv v + o' - o] = \sum_{v=0}^{m-1} \frac{1}{m} \cdot \frac{1}{m} = \frac{1}{m}$$

Proof of Theorem 1 (5/5) — result and comparison with HashedNet

Step 7: Sum

$$V_Z = \frac{1}{m} \sum_{\text{blk}(i) \neq \text{blk}(r)} (x_i^2 y_r^2 + x_i y_i x_r y_r)$$

$Z = 1$ (**HashedNet / feature hashing**): every pair $i \neq r$ counts

$$V_1 = \frac{1}{m} \sum_{i \neq r} (x_i^2 y_r^2 + x_i y_i x_r y_r)$$

Split the pairs into cross-block and within-block

$$V_Z = V_1 - \sum_c V_1(x_c, y_c, Z, m)$$

Each subtracted term is ≥ 0

$$x_i^2 y_r^2 + x_r^2 y_i^2 + 2x_i y_i x_r y_r = (x_i y_r + x_r y_i)^2 \geq 0$$

$$V_Z \leq V_1$$

Theorem 2 — are embedding relationships preserved?

$\theta \in \mathbb{R}^n$ flattened parameters; tokens $a \neq b$ with embeddings $\theta_a, \theta_b \in \mathbb{R}^D$ read from the ROBE array.

Expectation

$$\mathbb{E}[\widehat{\langle \theta_a, \theta_b \rangle}] = \begin{cases} \langle \theta_a, \theta_b \rangle & \text{same block} \\ \langle \theta_a, \theta_b \rangle (1 + \frac{1}{m}) & \text{different blocks (or } Z < D) \end{cases}$$

Variance of the product noise N , $Z \geq D$, separate blocks

$$\text{Var}(N) \leq O\left(\frac{D}{m^2} \|\theta\|_2^4 + \frac{1}{m^2} \sum_{i \neq j} \theta_a^{(i)} \theta_b^{(i)} \theta_a^{(j)} \theta_b^{(j)}\right)$$

Takeaway Same block: unbiased. Different blocks: small $1/m$ bias.

Proof of Theorem 2 (1/5) — reconstruction

Notation: $p_j = i_a + j$, $q_j = i_b + j$, $x_j = \theta_{p_j}$, $y_j = \theta_{q_j}$ ($p_j \neq q_j$)

1. Construct the shared memory

$$\text{mem}[k] = \sum_{i: \text{idx}(i)=k} g(i) \theta_i$$

2. Reconstruct a coordinate

$$\hat{x}_j = \text{mem}[\text{idx}(p_j)] g(p_j)$$

3. Expand the reconstruction

$$\hat{x}_j = \sum_{i=1}^n \theta_i g(i) g(p_j) \mathbf{1}\{\text{idx}(i) = \text{idx}(p_j)\} = x_j + A_j$$

$$A_j = \sum_{i \neq p_j} \theta_i g(i) g(p_j) \mathbf{1}\{\text{idx}(i) = \text{idx}(p_j)\}$$

$$B_j = \sum_{k \neq q_j} \theta_k g(k) g(q_j) \mathbf{1}\{\text{idx}(k) = \text{idx}(q_j)\}, \quad \hat{y}_j = y_j + B_j$$

Proof of Theorem 2 (2/5) — random error

Step 1: Product of signal and noise

$$\hat{x}_j \hat{y}_j = (x_j + A_j)(y_j + B_j) = x_j y_j + x_j B_j + y_j A_j + A_j B_j$$

Step 2: Random error of the estimator

$$\widehat{\langle \theta_a, \theta_b \rangle} - \langle \theta_a, \theta_b \rangle = \underbrace{\sum_{j=1}^D (x_j B_j + y_j A_j)}_L + \underbrace{\sum_{j=1}^D A_j B_j}_N$$

Step 3: Linear terms have zero mean ($k \neq q_j$, g independent of indicators)

$$\mathbb{E}[x_j B_j] = x_j \sum_{k \neq q_j} \theta_k \underbrace{\mathbb{E}[g(k)] \mathbb{E}[g(q_j)]}_0 \Pr[\text{idx}(k) = \text{idx}(q_j)] = 0$$

$$\mathbb{E}[y_j A_j] = 0 \text{ (same)} \Rightarrow \mathbb{E}[L] = 0$$

Proof of Theorem 2 (3/5) — computing $\mathbb{E}[A_j B_j]$

Step 4: Expand

$$A_j B_j = \sum_{i \neq p_j} \sum_{k \neq q_j} \theta_i \theta_k g(i) g(p_j) g(k) g(q_j) \mathbf{1}\{\text{idx}(i) = \text{idx}(p_j)\} \mathbf{1}\{\text{idx}(k) = \text{idx}(q_j)\}$$

Step 5: Even sign counts $(i \neq p_j, k \neq q_j, p_j \neq q_j)$

$$g(p_j) \text{ needs a partner} \Rightarrow k = p_j, \quad g(q_j) \text{ needs a partner} \Rightarrow i = q_j$$

Only $(i, k) = (q_j, p_j)$ survives, with $g(q_j)^2 g(p_j)^2 = 1$:

$$\mathbb{E}[A_j B_j] = \theta_{q_j} \theta_{p_j} \mathbb{E}[\mathbf{1}\{\text{idx}(q_j) = \text{idx}(p_j)\}]$$

$$\boxed{\mathbb{E}[A_j B_j] = x_j y_j \Pr[\text{idx}(p_j) = \text{idx}(q_j)]}$$

Proof of Theorem 2 (4/5) — probability and result

Step 6: Collision probability $\pi = \Pr[\text{idx}(p_j) = \text{idx}(q_j)]$

$$\pi = \begin{cases} 0 & p_j, q_j \text{ same block} \quad (\text{offsets differ, } |p_j - q_j| < Z < m) \\ \frac{1}{m} & \text{different blocks} \\ \frac{1}{m} & Z < D \quad (|p_j - q_j| \geq D > Z, \text{ never the same block}) \end{cases}$$

Step 7: Sum over coordinates

$$\mathbb{E}[\widehat{\langle \theta_a, \theta_b \rangle}] = \sum_{j=1}^D (x_j y_j + 0 + 0 + \pi x_j y_j) = (1 + \pi) \langle \theta_a, \theta_b \rangle$$

$\pi = 0 : \langle \theta_a, \theta_b \rangle$	$\pi = \frac{1}{m} : \langle \theta_a, \theta_b \rangle (1 + \frac{1}{m})$
--	--

Proof of Theorem 2 (5/5) — variance of $N = \sum_j A_j B_j$

Step 8: Expand N^2 (eight signs per term)

$$N^2 = \sum_{j_1, j_2} \sum_{i_1, \dots, i_4} \theta_{i_1} \theta_{i_2} \theta_{i_3} \theta_{i_4} g(p_{j_1}) g(q_{j_1}) g(p_{j_2}) g(q_{j_2}) g(i_1) g(i_2) g(i_3) g(i_4) \mathbf{1}_{i_1 \sim p_{j_1}} \mathbf{1}_{i_2 \sim q_{j_1}} \mathbf{1}_{i_3 \sim p_{j_2}} \mathbf{1}_{i_4 \sim q_{j_2}}$$

Step 9: Terms with even sign counts

$j_1 = j_2, i_1 = i_2 = i_3 = i_4 = i :$	θ_i^4	$\Pr \leq m^{-2}$
$j_1 = j_2, i\text{'s paired} :$	$\theta_i^2 \theta_k^2$	$\Pr \leq m^{-2}$
$j_1 \neq j_2, \{i_1..i_4\} = \{p_{j_1}, q_{j_1}, p_{j_2}, q_{j_2}\} :$	$x_{j_1} y_{j_1} x_{j_2} y_{j_2}$	$\Pr = m^{-2} \text{ or } m^{-3}$

$$\text{Var}(N) \leq O\left(\frac{D}{m^2} \|\theta\|_2^4 + \frac{1}{m^2} \sum_{j \neq j'} x_j y_j x_{j'} y_{j'}\right)$$



ROAST

Random Operation Access Specific Tile

Hardware-aware model compression with hashing

Idea · GMS vs. LMS theory · Results

Desai, Zhou, Shrivastava | ICML 2023

Presented by: Praveen

Roadmap: three parts

A The idea

- Why compress?
- HashedNet cache problem
- Tiles: lookup and matmul
- Global sharing, λ
- Backward pass

B The theory

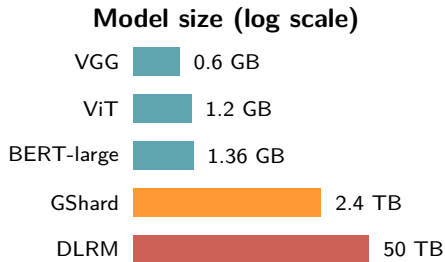
- Compression = hashing
- Theorem 5.1 and proof
- GMS vs. LMS variance
- Our numerical check

C The evidence

- Accuracy at 10–100×
- Convergence speed
- Kernel speed
- Limitations

Takeaway Our question: shrink a model 100× without slowing it down?

Motivation: models grow faster than hardware



VGG 150M, ViT 304M, BERT 340M, GShard 600B, DLRM $\sim 10T$
parameters

- Big model $\Rightarrow$ many GPUs, **communication every step**.
- $\sim 100\times$ smaller fits GPT, GShard, DLRM on **one GPU**.
- Smaller is also faster: DLRM $1000\times$ smaller gave $3\times$ faster inference.

Takeaway We want compression that works in **training** and makes models **faster**.

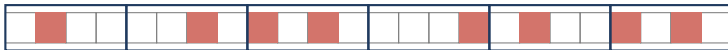
Compression landscape: what is missing?

Technique	Small in training	Any size	Cache-friendly
Pruning	✗	✗	✗
Low-rank decomposition	✓	✗	✓
Low precision	✓	✗	✓
Quantization-aware training	✗	✗	✓
HashedNet (parameter sharing)	✓	✓	✗
Knowledge distillation	✗	✗	✓
ROAST (ours)	✓	✓	✓

Takeaway HashedNet's one flaw is **cache-unfriendly** access. ROAST fixes it.

Why HashedNet is slow: scattered vs. coalesced reads

HashedNet: every weight hashed on its own



8 weights $\rightarrow$ 6 cache lines

ROAST: a whole tile hashed as one unit

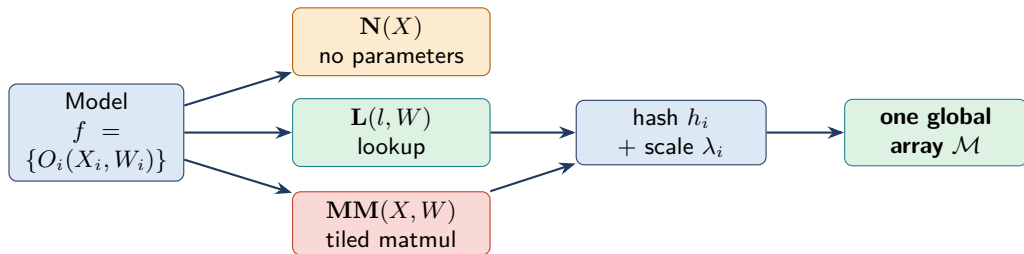


8 weights $\rightarrow$ 2 cache lines

thick boxes = cache lines (4 cells wide); coloured = values we need

Takeaway Fetching d weights: HashedNet $\approx d$ cache lines; ROAST with chunk $Z \approx d/Z + 1$.

ROAST in one picture



1. Tiles

Hash *blocks* shaped like what the algorithm reads.

2. Global sharing

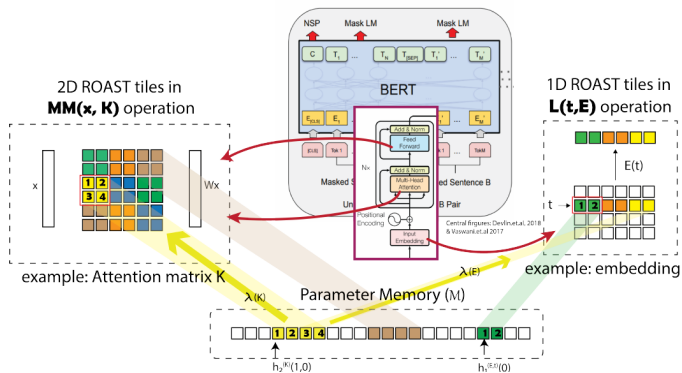
All modules share one $\mathcal{M}$; λ fixes scales.

3. Same model

MLP, attention, conv and embeddings all work.

Takeaway Memory $\propto |\mathcal{M}|$ in training **and** inference; the model's maths is unchanged.

Every model = three kinds of operations



Paper Fig. 1: 2D tiles (left), 1D chunks (right) map into $\mathcal{M}$.

L lookup

small $w = W(l)$:
embeddings, biases

MM matmul

$\mathbf{X} \cdot \mathbf{W}$ without building $\mathbf{W}$

N no weights

ReLU, softmax, ...

Takeaway Each module has **its own hash functions** but reads the same $\mathcal{M}$.

Lookup L: hash chunks, not single numbers

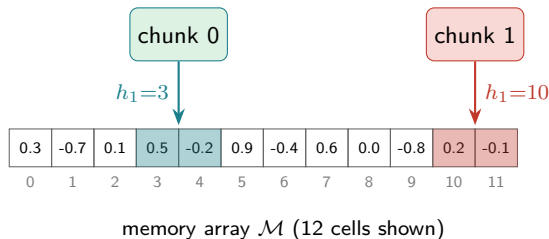
Chunk number $C(i) = \lfloor i/Z \rfloor$, offset $O(i) = i \bmod Z$:

$$w[i] = \lambda \mathcal{M}[h_1(C(i)) + O(i)], \quad h_1 : \mathbb{N} \rightarrow \{0, \dots, |\mathcal{M}| - Z\}$$

Worked example (ours)

Token t , dim 4, $Z=2$, $\lambda = 1/\sqrt{4} = 0.5$, $h_1(t, 0)=3$, $h_1(t, 1)=10$:

$$\begin{aligned} E(t) &= 0.5 [0.5, -0.2, 0.2, -0.1] \\ &= [0.25, -0.10, 0.10, -0.05] \end{aligned}$$



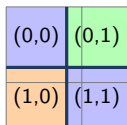
Takeaway Two contiguous reads instead of four random ones. Generalises ROBE's embedding kernel.

ROAST-MM: hash 2D tiles of the weight matrix

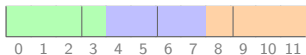
GPU matmul works tile by tile, so ROAST hashes whole **tiles**:

$$W[i, j] = \lambda \mathcal{M}[h_2(C_1, C_2) + Z_2 O_1 + O_2]$$

W (4×4), tiles 2×2



$\mathcal{M}$, $|\mathcal{M}| = 12$

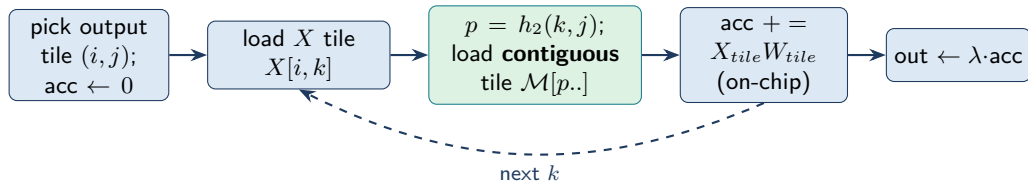


Worked example (ours). $h_2(0, 0)=4$, $h_2(0, 1)=0$,
 $h_2(1, 0)=8$, $h_2(1, 1)=4$.

- $W[3, 1]$: tile (1, 0), offsets (1, 1): $\text{idx} = 8 + 2 \cdot 1 + 1 = \mathbf{11}$.
- $W[1, 2]$: tile (0, 1), offsets (1, 0): $\text{idx} = 0 + 2 = \mathbf{2}$.
- Tiles (0, 0) and (1, 1) **collide as a whole**: both use $\mathcal{M}[4..7]$ (blue).

Takeaway One coalesced read per tile; collisions happen between **tiles**.

ROAST-MM kernel: how it runs on a GPU



Why Triton

cuBLAS and CUTLASS cannot load custom tiles, so the kernel is written in **Triton**.

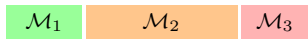
Tuning and backward

Tile size is **autotuned** per matrix shape. A custom backward pass scatter-adds gradients into $\mathcal{M}$.

Takeaway The full weight matrix is **never materialised**: only one tile at a time lives on-chip.

Global Memory Sharing (GMS) and the λ scale

LMS (HashedNet): one array per module



GMS (ROAST): one shared array $\mathcal{M}$



Problem: layers need different weight scales
(Xavier: $U(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}})$, $n = \text{fan-in}$).

Fix: initialise $\mathcal{M} \sim U(-\frac{1}{C}, \frac{1}{C})$; a layer with fan-in n multiplies by $\lambda = C/\sqrt{n}$

$$\lambda \mathcal{M} \sim U\left(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right)$$

Example, $C=1$: $n=100 \Rightarrow \lambda = 0.1$; $n=4 \Rightarrow \lambda = 0.5$. **Same array, right scale.**

Takeaway GMS = one shared pool; the per-module λ makes sharing safe.

Why GMS can express more than LMS

n parameters read from memory $\mathcal{M}$. GMS: up to $|\mathcal{M}|^n$ different mappings. LMS (pieces n_i with memories $|\mathcal{M}_i|$): only $\prod_i |\mathcal{M}_i|^{n_i} \leq |\mathcal{M}|^n$.

Our numbers: $n=1000$ (500+500), $|\mathcal{M}|=100$ (LMS: 50+50)

GMS: 10^{2000}

LMS: $50^{1000} \approx 10^{1699}$

GMS has $\approx 10^{301} \times$ more mappings

Skewed splits hurt more

If a tiny module gets only $|\mathcal{M}_i|=1$ cell, all its n_i weights are forced to be one number: $1^{n_i} = 1$ option.

Takeaway LMS is never more expressive than GMS, and often far less.

Backward pass with shared weights

Rule: if a value is used in several places, treat each use as a new variable and **sum** the gradients.
For ROAST (θ = recovered weights, w = cells of $\mathcal{M}$, $\lambda(j)$ = scale of θ_j 's module):

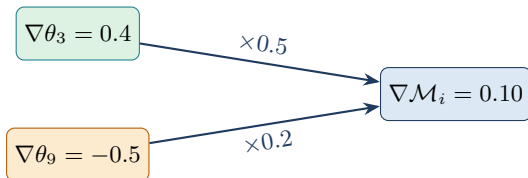
$$\nabla_{w_i} f = \sum_{j: h(j)=i} \lambda(j) \nabla_{\theta_j} f$$

Example (ours)

Cell $\mathcal{M}_i$ is used by θ_3 (module A, $\lambda_A=0.5$) and θ_9 (module B, $\lambda_B=0.2$).

With $\nabla\theta_3 = 0.4$ and $\nabla\theta_9 = -0.5$:

$$\nabla\mathcal{M}_i = 0.5(0.4) + 0.2(-0.5) = 0.10$$



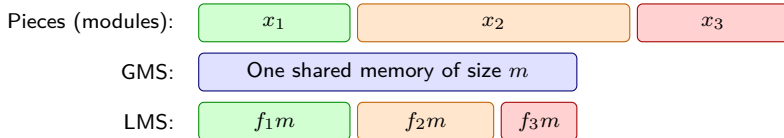
Takeaway Auto-diff would rebuild the full model, so ROAST ships its **own backward kernels**.

Theory setup: compression = dimensionality reduction

Parameters $\in \mathbb{R}^n \rightarrow$ memory of size m . Quality = how well **inner products** $\langle x, y \rangle$ survive hashing (h bucket, g sign; chunk size 1).

$$\text{GMS: } \langle x, y \rangle_{G,m} = \sum_{j=1}^m \left[\sum_i \mathbf{1}\{h(i) = j\} g(i) x_i \right] \left[\sum_i \mathbf{1}\{h(i) = j\} g(i) y_i \right]$$

$$\text{LMS: } \langle x, y \rangle_L = \sum_{l=1}^k \langle x_l, y_l \rangle_{G, f_l m} \quad \left(\sum_l f_l = 1 \right)$$



Takeaway Original parameters occupy a larger space. GMS compresses them into one shared memory; LMS divides the compressed memory among modules.

Theorem 5.1: both unbiased, but different variances

Theorem 5.1 ($V_l \propto \frac{1}{f_l m}$ = variance of piece l hashed alone)

$$\mathbb{E}\langle x, y \rangle_{G, m} = \mathbb{E}\langle x, y \rangle_L = \langle x, y \rangle$$

$$V_L = \sum_l V_l, \quad V_G = \sum_l f_l V_l + \frac{1}{m} \sum_{l_1 \neq l_2} \left(\|x_{l_1}\|^2 \|y_{l_2}\|^2 + \langle x_{l_1}, y_{l_1} \rangle \langle x_{l_2}, y_{l_2} \rangle \right)$$

V_G stays bounded

Within-piece noise shrinks by f_l ; the price is cross-piece collisions.

V_L blows up

The $1/f_l$ factors explode when a piece gets a tiny share.

Paper typo: Eq. (22) prints $\langle x_{l_1}, y_{l_2} \rangle$; the correct term is $\langle x_{l_1}, y_{l_1} \rangle \langle x_{l_2}, y_{l_2} \rangle$.

Takeaway Same mean, but V_L depends on $1/f_l$, while V_G does not blow up.

Proof 1: expectation — GMS is unbiased

Let $I_{ir} = \mathbf{1}[h(i) = h(r)]$. Expand the estimator:

$$\widehat{\langle x, y \rangle}_{G,m} = \sum_{i,r} I_{ir} g(i) g(r) x_i y_r.$$

❶ **Diagonal + collisions.** Since $I_{ii} = g(i)^2 = 1$,

$$\widehat{\langle x, y \rangle}_{G,m} = \langle x, y \rangle + E_G, \quad E_G = \sum_{i \neq r} I_{ir} g(i) g(r) x_i y_r. \quad (2)$$

❷ **Zero-mean error.** For $i \neq r$, independent signs give

$$\mathbb{E}[g(i)g(r)] = \mathbb{E}[g(i)]\mathbb{E}[g(r)] = 0. \quad (3)$$

❸ **Result.** Since $\mathbb{E}[E_G] = 0$, $\boxed{\mathbb{E}[\widehat{\langle x, y \rangle}_{G,m}] = \langle x, y \rangle.}$

Takeaway Random signs cancel collision error in expectation; the true inner product remains.

Proof 2: GMS variance — only paired collisions survive

From (2), the estimator is $\langle x, y \rangle + E_G$ with $\mathbb{E}[E_G] = 0$, hence

$$V_G = \text{Var}(E_G) = \mathbb{E}[E_G^2]. \quad (5)$$

❶ **Expand the square.**

$$V_G = \sum_{i \neq r} \sum_{p \neq q} \mathbb{E}[I_{ir} I_{pq} g(i) g(r) g(p) g(q)] x_i y_r x_p y_q. \quad (6)$$

❷ **Use independent random signs.** A product has nonzero expectation only if every sign occurs an even number of times. For each (i, r) , the surviving pairs are $(p, q) = (i, r)$ and $(p, q) = (r, i)$.

❸ **Evaluate each surviving term.** Both sign products equal 1; uniform hashing gives $\mathbb{E}[I_{ir}] = \Pr[h(i) = h(r)] = 1/m$. Their contributions are

$$\frac{1}{m} x_i^2 y_r^2, \quad \frac{1}{m} x_i y_i x_r y_r. \quad (7)$$

Summing over all ordered pairs $i \neq r$ gives

$$V_G = \frac{1}{m} \sum_{i \neq r} (x_i^2 y_r^2 + x_i y_i x_r y_r). \quad (8)$$

Takeaway GMS variance comes from two surviving pairings, each weighted by collision probability $1/m$.

Proof 3: LMS variance — local memory raises collision noise

Split x, y into k chunks and give chunk ℓ a memory of $f_\ell m$ cells, with $f_\ell > 0$ and $\sum_{\ell=1}^k f_\ell = 1$.

① **Write LMS as local estimates:** $\widehat{\langle x, y \rangle}_L = \sum_{\ell=1}^k \widehat{\langle x_\ell, y_\ell \rangle}_{G, f_\ell m}$.

② **Use independence across chunks.** With independent local memories, variances add:

$$V_L = \sum_{\ell=1}^k V_\ell. \quad (10)$$

③ **Apply the GMS result to each chunk.** The collision probability is now $\Pr[h_\ell(i) = h_\ell(j)] = 1/(f_\ell m)$, so

$$V_\ell = \frac{1}{f_\ell m} \sum_{i \neq j} (x_{\ell i}^2 y_{\ell j}^2 + x_{\ell i} y_{\ell i} x_{\ell j} y_{\ell j}). \quad (11)$$

$$V_L = \sum_{\ell=1}^k \frac{1}{f_\ell m} \sum_{i \neq j} (x_{\ell i}^2 y_{\ell j}^2 + x_{\ell i} y_{\ell i} x_{\ell j} y_{\ell j}).$$

Takeaway LMS divides memory into smaller pools, amplifying each chunk's collision variance by $1/f_\ell$.

Comparison: V_L vs. V_G under the paper's assumptions

Assumptions (Xavier-style init gives similar piece norms): every piece has $\|x_l\| \|y_l\| = \alpha$ (equal norms across modules) and $\langle x_l, y_l \rangle = \beta$; drop the small $-2 \sum a_i^2 b_i^2$ term. Then

$$V_L \approx \frac{\alpha^2 + \beta^2}{f_l m}, \quad V_G \approx \underbrace{\sum_l f_l V_l}_{k(\alpha^2 + \beta^2)/m} + \underbrace{\frac{k(k-1)}{m}(\alpha^2 + \beta^2)}_{\text{cross terms}} = \frac{k^2}{m}(\alpha^2 + \beta^2)$$

$$V_L \approx \frac{\alpha^2 + \beta^2}{m} \sum_{l=1}^k \frac{1}{f_l} \geq \frac{\alpha^2 + \beta^2}{m} \cdot \frac{k^2}{\sum_l f_l} = V_G \quad (\text{Cauchy-Schwarz: } (\sum f_l)(\sum 1/f_l) \geq k^2)$$

Clean form

$$\frac{V_L}{V_G} = \frac{1}{k^2} \sum_l \frac{1}{f_l} = \frac{\text{AM}(f)}{\text{HM}(f)} \geq 1$$

with equality iff $f_l = 1/k$ for all l (equal-size memories).

Note: approximate average-case comparison, not strict dominance for every x, y .

Our examples

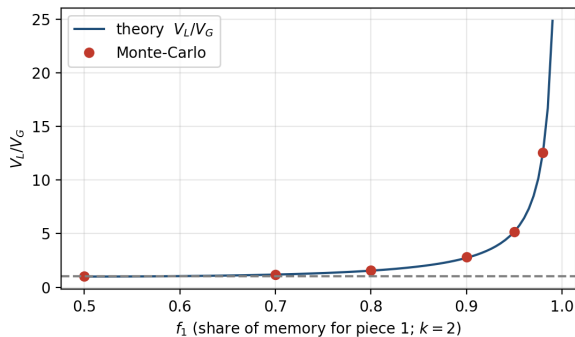
$$k = 2, f = (0.9, 0.1): \frac{1}{4}(1.11 + 10) = 2.8$$

$$k = 3, f = (0.98, 0.01, 0.01):$$

$$\frac{1}{9}(1.02 + 200) = 22.3$$

Takeaway Under the paper's assumptions, skewed memory splits make LMS variance grow much faster than GMS.

Numerical check of the variance formulas (our Monte-Carlo)



$k = 2$ pieces of size 100, $m = 200$, random unit-norm pieces; 4×10^4 trials per setting.

GMS: formula 0.01988, simulation 0.01985.

f_1	V_L (formula)	V_L/V_G (formula)	V_L/V_G (MC)
0.50	0.0198	0.996	1.005
0.70	0.0235	1.18	1.19
0.80	0.0307	1.54	1.55
0.90	0.0544	2.74	2.81
0.95	0.1030	5.18	5.18
0.98	0.2495	12.55	12.55

Takeaway Monte-Carlo matches the formulas; skewed LMS splits sharply raise variance.

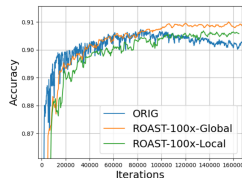
Empirical GMS vs. LMS (BERT-12-12, text classification)

Test accuracy (%)

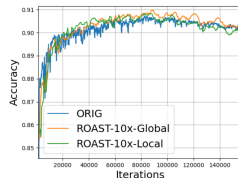
Model	Size	Amazon	Yelp
BERT-12-12	108M	93.51	90.8
10× LMS	10.1M	93.49	90.9
10× GMS	10.1M	94.64	91.1
100× LMS	1M	92.9	90.7
100× GMS	1M	93.9	91.0

Epochs to reach a target accuracy

Model	Amazon	Yelp
BERT-12-12 (93.4 / 90.8)	5	4.7
10× LMS (93.4 / 90.9)	3.77	4.69
10× GMS (93.4 / 90.9)	1.97	3.4
100× LMS (92.9 / 90.7)	4.78	9.09
100× GMS (92.9 / 90.7)	3.09	3.74



(a) GMS vs. LMS 100× Yelp



(b) GMS vs. LMS 10× Yelp

Yelp: GMS (orange), LMS (green), original (blue)

Takeaway At 10–100× compression, GMS preserves accuracy and reaches target accuracy in fewer epochs.

Compression quality and kernel speed

Quality at high compression

Model	Size	Amazon	Yelp
BERT-2-2	37.4M	93.4	90.8
ROAST-10×	3.7M	94.6	90.8
ROAST-100×	393K	93.8	90.8
Prune-100× (9-1)	374K	91.4	89.0

CIFAR-10 (target 94%)

	Size	Acc.
ResNet-9	6.5M	94.2
ROAST-5×	1.2M	94.58
ROAST-10×	650K	94.15
ROAST-10× + Prune-10×	65K	94.06

Inference time (ms), square matmul, batch 512, A100

Dim × Dim	512	2048	8096	20480	Avg
PyTorch-MM	0.10	0.12	0.69	3.91	0.91
HashedNet 4MB	0.31	0.63	6.20	35.22	7.77
HashedNet 512MB	0.33	2.05	40.55	272.23	56.03
ROAST 4MB	0.28	0.27	0.99	4.83	1.22
ROAST 512MB	0.30	0.27	1.02	4.95	1.24

- ROAST-MM $\approx 45\times$ faster than HashedNet on average (512MB), $\approx 1.34\times$ slower than cuBLAS.
- Total training step (20480², Adagrad, 512MB): 668 ms vs. 32 ms $\approx 21\times$.

Takeaway ROAST keeps accuracy at high compression and its tiled kernel is much closer to cuBLAS than HashedNet.

Summary and limitations

Take-aways

- ROAST = parameter sharing with **tiles matched to the operation** $\Rightarrow$ cache-efficient.
- **GMS**: one shared array + per-module λ ; more expressive, lower variance than LMS (AM/HM argument: $V_L/V_G = \text{AM}(f)/\text{HM}(f)$).
- 10–100 $\times$ memory reduction *during training and inference* at similar accuracy (BERT-2-2, ResNet-9).
- Kernel up to $\sim 45\times$ faster than HashedNet.

Limitations (paper)

- Evaluated on classification models only.
- ROAST-MM slower than cuBLAS (esp. small matrices).
- Reduces memory, not computation.
- Variance comparison is approximate, not universal.

Questions?

Section 4

Stability of STABLE-RPS
(Desai and Shrivastava 2024)

While pruning has been the more popular way to compress a model, parameter sharing is a more promising method of compression. In this paper, the SOTA RPS scheme ROAST is studied and improvements on it are presented.

As we prune more and more parameters, we start dropping nodes and filters, which would affect model capacity. RPS, on the contrary, seems to perform much better at extreme compression due to the fact that we are not dropping out parameters exactly while the improved RPS can even reach Pareto Continuity (more on that later).

Setup and Notation

- ▶ The full model has n weights $\theta \in \mathbb{R}^n$ and the loss $F(\theta)$.
- ▶ The RPS array has $m \ll n$ trainable parameters $\psi \in \mathbb{R}^m$.
- ▶ RPS uses a hash $h : [n] \rightarrow [m]$, a sign hash $g : [n] \rightarrow \{\pm 1\}$, and a **scaling factor** $\lambda_i > 0$ for each weight.

The model calculates each weight from the array:

$$\theta_i = \lambda_i g(i) \psi_{h(i)} \quad \Longleftrightarrow \quad \boxed{\theta = \Lambda R \psi}$$

In this equation:

$$R \in \{0, 1\}^{n \times m}, \quad R_{ij} = \mathbf{1}(h(i) = j), \quad \Lambda = \text{diag}(g(i)\lambda_i).$$

We train the compressed model with gradient descent on this function:

$$G(\psi) := F(\Lambda R \psi).$$

Why Global Memory Sharing is Unstable

- ▶ ROAST uses one array for all the layers. Thus each layer must have a scaling factor that gives the correct initial standard deviation: $\lambda = \sigma_{\text{layer}} / \sigma_{\text{array}}$.
- ▶ The chain rule gives the gradient on one slot of the array:

$$\frac{\partial G}{\partial \psi_j} = \sum_{i:h(i)=j} g(i) \lambda_i \frac{\partial F}{\partial \theta_i}.$$

- ▶ One step of gradient descent thus changes a recovered weight by:

$$\theta_i^{(t+1)} - \theta_i^{(t)} = -\eta \lambda_i g(i) \sum_{k:h(k)=h(i)} \lambda_k g(k) \frac{\partial F}{\partial \theta_k}.$$

Result

The term $\lambda_i \lambda_k$ multiplies the step. In ResNet-20, λ is between 1 and 5. Thus some updates are approximately $25\times$ larger than in the full model.

Theorem 1: Stability of RPS

Theorem

Let F be L_f -smooth with n parameters. Use RPS with an array of size m , the hash h , and the scaling factors $\lambda_1, \dots, \lambda_n$. Then the stable range of the learning rate for gradient descent changes from the full model to the compressed model:

$$\eta \in \left(0, \frac{2}{L_f}\right) \longrightarrow \eta \in \left(0, \frac{2}{L_f M}\right), \quad M := \max_{j \in [m]} \sum_{i: h(i)=j} \lambda_i^2.$$

M is the largest sum of λ_i^2 on one slot of ψ . A larger M gives a smaller stable range. M increases when:

- ▶ the values of λ increase. These values change with the init-stdev.
- ▶ the compression increases. More weights then go to each slot.

Theorem 1 Proof (1/3): No Compression

The update of gradient descent is $\theta_{t+1} = \theta_t - \eta \nabla F(\theta_t)$. The L -smoothness of F gives the descent lemma:

$$F(y) \leq F(x) + \langle \nabla F(x), y - x \rangle + \frac{L}{2} \|y - x\|^2.$$

Put $y = \theta_{t+1}$ and $x = \theta_t$ in the lemma:

$$\begin{aligned} F(\theta_{t+1}) &\leq F(\theta_t) - \eta \|\nabla F(\theta_t)\|^2 + \frac{L\eta^2}{2} \|\nabla F(\theta_t)\|^2 \\ &= F(\theta_t) - \eta \left(1 - \frac{L\eta}{2}\right) \|\nabla F(\theta_t)\|^2. \end{aligned}$$

Result

The loss decreases at each step if $1 - \frac{L\eta}{2} > 0$. This condition is $\boxed{\eta < 2/L}$. Thus, we must find the smoothness constant L_g of $G(\psi) = F(\Lambda R\psi)$.

Theorem 1 Proof (2/3): Smoothness of G

The chain rule gives $\nabla G(\psi) = (\Lambda R)^\top \nabla F(\Lambda R\psi)$. For all ψ_1, ψ_2 :

$$\begin{aligned}\|\nabla G(\psi_1) - \nabla G(\psi_2)\| &= \|(\Lambda R)^\top (\nabla F(\Lambda R\psi_1) - \nabla F(\Lambda R\psi_2))\| \\ &\leq \|\Lambda R\|_2 \|\nabla F(\Lambda R\psi_1) - \nabla F(\Lambda R\psi_2)\| \\ &\leq \|\Lambda R\|_2 L_f \|\Lambda R\psi_1 - \Lambda R\psi_2\| \\ &\leq \|\Lambda R\|_2^2 L_f \|\psi_1 - \psi_2\|.\end{aligned}$$

Result

G is L_g -smooth with $L_g = \|\Lambda R\|_2^2 L_f$.

The first and the last inequalities use the spectral norm. The middle inequality uses the L_f -smoothness of F .

Theorem 1 Proof (3/3): Calculation of $\|\Lambda R\|_2^2$

The square of the spectral norm is the largest eigenvalue of $R^\top \Lambda^2 R$:

$$\begin{aligned}\|\Lambda R\|_2^2 &= \lambda_{\max}(R^\top \Lambda^2 R), \\ (R^\top \Lambda^2 R)_{jk} &= \sum_{i=1}^n R_{ij} g(i)^2 \lambda_i^2 R_{ik} = \mathbf{1}(j=k) \sum_{i:h(i)=j} \lambda_i^2.\end{aligned}$$

- ▶ Each row of R has only one 1. Thus no two columns of R have a 1 in the same row, and the matrix is **diagonal**.
- ▶ The value of $g(i)^2$ is 1. Thus the sign hash has no effect.

The largest diagonal entry gives the result:

$$L_g = L_f \max_j \sum_{i:h(i)=j} \lambda_i^2 = L_f M \implies \eta < \frac{2}{L_f M}. \quad \blacksquare$$

STABLE-RPS multiplies the gradient by a scaler $\Gamma \in \mathbb{R}^m$ before the optimizer step:

$$\nabla_{\text{eff}} G = \Gamma \circ \nabla G, \quad \psi_{t+1} = \psi_t - \eta \nabla_{\text{eff}} G(\psi_t).$$

The paper gives two scalers:

- ▶ **Theory scaler** (Eq. 6). It removes M from Theorem 1: $\Gamma[j] = 1 / \sum_{i:h(i)=j} \lambda_i^2$.
- ▶ **Effective-update scaler** (Eq. 8). It makes the change of the recovered weights equal to the mean change in the full model:

$$\Gamma[j] = \frac{\sum_{i:h(i)=j} 1}{(\sum_{i:h(i)=j} \lambda_i)^2}.$$

The experiments in the paper use the effective-update scaler.

Theorem 2: Stability with Gradient Scaling

Theorem

Let F be L_f -smooth with n parameters. Use RPS with an array of size m , the hash h , and the scaling factors $\lambda_1, \dots, \lambda_n$. Scale the gradient with $\Gamma[j] = 1 / \sum_{i:h(i)=j} \lambda_i^2$. Then the stable range of the learning rate for gradient descent stays

$$\eta \in \left(0, \frac{2}{L_f}\right)$$

for all compression factors and all values of λ .

Meaning: The scaler removes the effect of M . The learning rate of the full model is also stable for the compressed model.

Theorem 2 Proof (1/2): The Update in Weight Space

Write the scalar as the diagonal matrix $\Gamma = \text{diag}(\gamma_1, \dots, \gamma_m)$. The update in the array space is

$$\psi_{t+1} = \psi_t - \eta \Gamma (\Lambda R)^\top \nabla F(\theta_t).$$

Multiply by ΛR . The update in the weight space is

$$\theta_{t+1} = \theta_t - \eta P \nabla F(\theta_t), \quad \boxed{P := \Lambda R \Gamma R^\top \Lambda}.$$

Put this update in the descent lemma, with $\nabla F = \nabla F(\theta_t)$:

$$F(\theta_{t+1}) \leq F(\theta_t) - \eta \nabla F^\top P \nabla F + \frac{L_f \eta^2}{2} \nabla F^\top P^2 \nabla F.$$

Thus, we must find the relation between P and P^2 .

Theorem 2 Proof (2/2): P is a Projection

By the proof of Theorem 1, $R^\top \Lambda^2 R = \text{diag}(\sum_{i:h(i)=j} \lambda_i^2) = \Gamma^{-1}$. Thus

$$P^2 = \Lambda R \Gamma (R^\top \Lambda^2 R) \Gamma R^\top \Lambda = \Lambda R \Gamma \Gamma^{-1} \Gamma R^\top \Lambda = P.$$

P is symmetric and $P^2 = P$, so P is the orthogonal projection onto the column space of ΛR . Thus $\nabla F^\top P^2 \nabla F = \nabla F^\top P \nabla F = \|P \nabla F\|^2$:

$$F(\theta_{t+1}) \leq F(\theta_t) - \eta \left(1 - \frac{L_f \eta}{2}\right) \|P \nabla F(\theta_t)\|^2.$$

Result

Same inequality as for the full model: the loss decreases if $\eta < 2/L_f$. ■

Load Factor and Pareto-Continuity

Definition (Load factor)

The load factor of a mapping $h : [n] \rightarrow [m]$ is the maximum number of weights on one slot:
 $\ell(h) = \max_{j \in [m]} |\{i \in [n] : h(i) = j\}|$.

The pigeonhole principle gives $\ell(h) \geq \lceil n/m \rceil$. The load is **optimal** if $\ell(h) = \lceil n/m \rceil$.

Definition (Pareto-continuity)

At $m = n$, the mapping is one-to-one. Then the compressed model is equal to the full model.

Optimal load gives Pareto-continuity, because $\ell(h) = 1$ at $m = n$. Random hashing (HashedNet, ROAST) gives collisions also at $m = n$.

The RANDOMFOLD Mapping

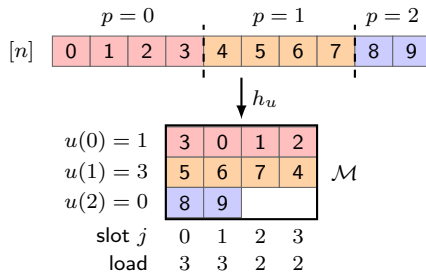
Let u be a random hash function. RANDOMFOLD $h_u : [n] \rightarrow [m]$ is

$$h_u(x) = \left(u(\lfloor x/m \rfloor) + (x \bmod m) \right) \bmod m.$$

$[n]$ has partitions of size m . Partition $p = \lfloor x/m \rfloor$ goes onto the array with a random cyclic shift $u(p)$.

STABLE-RPS applies h_u to the global index $G(i)$ of each weight in the flattened ROAST chunks:

$$\text{STABLE-RPS}(i) = h_u(G(i)).$$



One row per partition. Max load $\lceil 10/4 \rceil = 3$.

Theorem 3: Optimal Load and Cache Efficiency

Theorem

The STABLE-RPS mapping has these two properties:

- ① *It has the optimal load factor $\lceil n/m \rceil$. Thus it has Pareto-continuity.*
- ② *Let each ROAST chunk be smaller than the memory budget m . Then for each ROAST chunk, the number of cache-line fetches obeys*

$$C(\text{STABLE-RPS}) \leq C(\text{ROAST}) + 3.$$

Meaning: STABLE-RPS gets the optimal load. The cost is a maximum of 3 more cache-line fetches for each chunk.

Theorem 3 Proof (1/3): Optimal Load

- There are $\lceil n/m \rceil$ partitions. In partition p , the map

$$x \mapsto (u(p) + (x \bmod m)) \bmod m$$

is a cyclic shift of the offsets $0, \dots, m-1$. Thus it is one-to-one.

- Thus each partition puts a maximum of one weight on each slot.

$$\ell(h_u) \leq \# \text{partitions} = \left\lceil \frac{n}{m} \right\rceil.$$

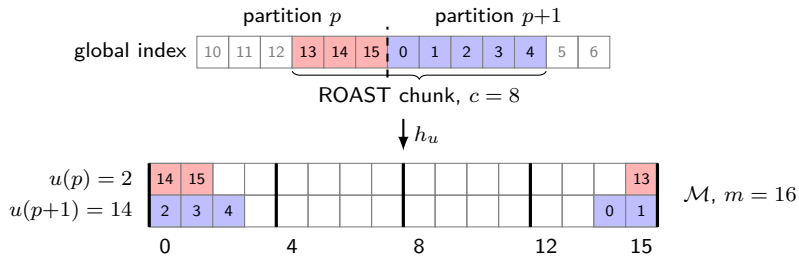
The pigeonhole principle gives $\ell(h_u) \geq \lceil n/m \rceil$. Thus the load is optimal.

Pareto-continuity

If $m = n$, there is one partition, and h_u is a cyclic shift. Thus $\theta_i = \lambda_i g(i) \psi_{h_u(i)}$ is an invertible linear map. The compressed model can show all $\theta \in \mathbb{R}^n$, as the full model can.

Theorem 3 Proof (2/3): Where a Chunk Is Cut

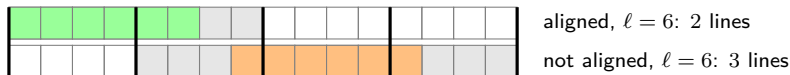
- **Partition boundary.** A ROAST chunk $[a, a + c)$ has $c \leq m$. Thus it crosses a maximum of one boundary, which gives $k \leq 2$ pieces.
- **Wrap-around.** Each piece moves by its own shift $u(p)$ modulo m . A piece can wrap from slot $m - 1$ to slot 0.



Both pieces wrap: 4 runs. Thick lines are cache-line boundaries ($L = 4$ slots).

Theorem 3 Proof (3/3): Counting Cache Lines

A cache line holds L slots, and L divides m . A run of length ℓ touches $\lceil \ell/L \rceil$ lines if it is **aligned** (it starts or ends on a line boundary). Else it touches $\leq \lceil \ell/L \rceil + 1$ lines.



$k \leq 2$ pieces and $w \leq k$ wraps give $r = k + w$ runs. Each wrap gives 2 aligned runs, so only $k - w$ runs are not aligned. With $\sum_{i=1}^r \lceil \ell_i/L \rceil \leq \lceil c/L \rceil + r - 1$:

$$C(\text{STABLE-RPS}) \leq \sum_{i=1}^r \left\lceil \frac{\ell_i}{L} \right\rceil + (k - w) \leq \left\lceil \frac{c}{L} \right\rceil + 2k - 1.$$

Result

$$C(\text{STABLE-RPS}) \leq \lceil c/L \rceil + 3 \leq C(\text{ROAST}) + 3, \text{ as } 2k - 1 \leq 3.$$



Data Compression: Inner-Product Estimators

Compress $x, y \in \mathbb{R}^n$ to $m = n/k$ values, with $k \in \mathbb{N}$.

Pruning (random sampling)

Keep m random coordinates. $\mathbf{1}(j)$ is 1 if we keep coordinate j :

$$\langle \hat{x}, \hat{y} \rangle_{\text{prune}} = k \sum_{j=1}^n x_j y_j \mathbf{1}(j).$$

STABLE-RPS (sketch)

Apply a random permutation, then the mapping h and the sign hash g : $\hat{x}_i = \sum_{j:h(j)=i} g(j) x_j$.
Then

$$\langle \hat{x}, \hat{y} \rangle_{\text{STABLE-RPS}} = \sum_{i=1}^m \hat{x}_i \hat{y}_i = \sum_{a,b} x_a y_b g(a) g(b) \mathbf{1}(h(a) = h(b)).$$

Theorem 4: Norm Preservation

Theorem

Let $x \in \mathbb{R}^n$ and $m = n/k$ with $k \in \mathbb{N}$. Both estimates of $\|x\|^2$ are unbiased, with the variances

$$V(\langle \hat{x}, \hat{x} \rangle_{\text{prune}}) = \left(\frac{n}{m} - 1\right) \sum_i x_i^4 + \frac{m - n}{m(n - 1)} \sum_{i \neq j} x_i^2 x_j^2,$$

$$V(\langle \hat{x}, \hat{x} \rangle_{\text{STABLE-RPS}}) = \frac{2}{m} \frac{n - m}{n - 1} \sum_{i \neq j} x_i^2 x_j^2.$$

Let $x_i, y_i \sim \mathcal{N}(0, \sigma_i^2)$ be independent (distribution $\mathcal{D}$). Then $\mathbb{E}_{\mathcal{D}}[V_{\text{prune}}] \geq \mathbb{E}_{\mathcal{D}}[V_{\text{STABLE-RPS}}]$ for norms and for inner products. Equality occurs if and only if all σ_i are equal.

Theorem 4 Proof (1/3): Pruning

The sampling gives

$$\Pr [\mathbf{1}(i) = 1] = \frac{m}{n}, \quad \Pr [\mathbf{1}(i) = \mathbf{1}(j) = 1] = \frac{m(m-1)}{n(n-1)} \quad (i \neq j).$$

Mean: $\mathbb{E}\langle \hat{x}, \hat{y} \rangle = k \sum_i x_i y_i \frac{m}{n} = \langle x, y \rangle$.

Second moment: expand the square of the sum:

$$\mathbb{E}\langle \hat{x}, \hat{y} \rangle^2 = k \sum_i x_i^2 y_i^2 + \frac{k(m-1)}{n-1} \sum_{i \neq j} x_i y_i x_j y_j.$$

Variance: subtract $\langle x, y \rangle^2 = \sum_i x_i^2 y_i^2 + \sum_{i \neq j} x_i y_i x_j y_j$:

$$V_{\text{prune}} = (k-1) \sum_i x_i^2 y_i^2 + \frac{m-n}{m(n-1)} \sum_{i \neq j} x_i y_i x_j y_j.$$

Put $y = x$ to get the variance for $\|x\|^2$.

Theorem 4 Proof (2/3): STABLE-RPS

Mean: $\mathbb{E}[g(a)g(b)] = \mathbf{1}(a = b)$. Thus $\mathbb{E}\langle \hat{x}, \hat{y} \rangle = \sum_a x_a y_a = \langle x, y \rangle$.

Second moment: $\mathbb{E}[g(a)g(b)g(c)g(d)] \neq 0$ only when the indices make pairs: $(a = b, c = d)$, $(a = c, b = d)$, $(a = d, b = c)$. The first case gives $\langle x, y \rangle^2$. The other two cases need a collision of $i \neq j$. With $p := \Pr[h(i) = h(j)]$:

$$V_{\text{STABLE-RPS}} = p \sum_{i \neq j} (x_i^2 y_j^2 + x_i y_i x_j y_j).$$

Collision probability: two indices in the same partition never collide (Theorem 3). In different partitions, they collide with probability $1/m$:

$$p = \left(1 - \frac{m-1}{n-1}\right) \frac{1}{m} = \frac{n-m}{m(n-1)}.$$

Put $y = x$: $V = 2p \sum_{i \neq j} x_i^2 x_j^2$, as in the theorem.

Theorem 4 Proof (3/3): Expectation over the Data

For $x_i \sim \mathcal{N}(0, \sigma_i^2)$: $\mathbb{E}[x_i^4] = 3\sigma_i^4$ and $\mathbb{E}[x_i^2 x_j^2] = \sigma_i^2 \sigma_j^2$. Let $a_i := \sigma_i^2$. Then

$$\mathbb{E}_{\mathcal{D}} V_{\text{prune}} = 3 \frac{n-m}{m} \sum_i a_i^2 - \frac{n-m}{m(n-1)} \sum_{i \neq j} a_i a_j,$$

$$\mathbb{E}_{\mathcal{D}} V_{\text{STABLE-RPS}} = 2 \frac{n-m}{m(n-1)} \sum_{i \neq j} a_i a_j.$$

Subtract, and use $(n-1) \sum_i a_i^2 - \sum_{i \neq j} a_i a_j = \sum_{i < j} (a_i - a_j)^2$:

$$\mathbb{E}_{\mathcal{D}} V_{\text{prune}} - \mathbb{E}_{\mathcal{D}} V_{\text{STABLE-RPS}} = \frac{3(n-m)}{m(n-1)} \sum_{i < j} (\sigma_i^2 - \sigma_j^2)^2 \geq 0.$$

Result

Zero if and only if all σ_i are equal. Inner products: factor 1 in place of 3. ■

Theorem 4: Interpretation

- ▶ For one fixed x , either method can have the smaller variance. Thus the paper compares the expected variance over the data.
- ▶ If all σ_i are equal, the two methods are equal on average.
- ▶ Real data is not balanced. The values of σ_i usually follow a power law. Then STABLE-RPS has a strictly smaller variance.
- ▶ The gap is $\propto \sum_{i < j} (\sigma_i^2 - \sigma_j^2)^2$. Thus a larger imbalance gives a larger advantage to STABLE-RPS.

Intuition

Pruning keeps or removes a full coordinate. A large coordinate is either multiplied by k or lost. STABLE-RPS keeps a part of each coordinate in the sketch, so no large coordinate is lost.

Theorem 5: Linear Regression under Compression

Theorem

Let y be a target with standard deviation σ_y . Let the inputs $x_1, \dots, x_n$ be uncorrelated, each with standard deviation σ_x , and let $\rho_i = \text{Cov}(y, x_i) / (\sigma_x \sigma_y)$. Fix the pruning sample or the STABLE-RPS mapping, with $m = n/k$. Then the optimal (least) residual of the compressed linear model is

$$\text{Res}_{\text{comp}}(\hat{y}) = \sigma_y^2 \left(1 - \frac{1}{k} \langle \hat{\rho}, \hat{\rho} \rangle \right).$$

$\langle \hat{\rho}, \hat{\rho} \rangle$ is the estimate of $\|\rho\|^2$ from Theorem 4 for the same method.

Meaning: The quality of the compressed model depends only on how well the compression keeps the norm of ρ .

Theorem 5 Proof (1/2): Pruning

The model uses only the kept inputs: $\hat{y} = \sum_i \alpha_i \mathbf{1}(i) x_i$. The inputs are uncorrelated. Thus

$$\text{Res}(\hat{y}) = V(y - \hat{y}) = \sigma_y^2 + \sum_i \alpha_i^2 \mathbf{1}(i) \sigma_x^2 - 2 \sum_i \alpha_i \mathbf{1}(i) \text{Cov}(y, x_i).$$

Each α_i has a separate quadratic. The minimum is at $\alpha_i = \text{Cov}(y, x_i)/\sigma_x^2$. Thus

$$\begin{aligned} \text{Res}(\hat{y}) &= \sigma_y^2 - \sum_i \mathbf{1}(i) \frac{\text{Cov}(y, x_i)^2}{\sigma_x^2} = \sigma_y^2 \left(1 - \sum_i \rho_i^2 \mathbf{1}(i) \right) \\ &= \sigma_y^2 \left(1 - \frac{1}{k} \underbrace{k \sum_i \rho_i^2 \mathbf{1}(i)}_{\langle \hat{\rho}, \hat{\rho} \rangle_{\text{prune}}} \right). \end{aligned}$$

Theorem 5 Proof (2/2): STABLE-RPS

All the weights on slot j share one parameter α_j . Thus the model is

$$\hat{y} = \sum_{j=1}^m \alpha_j z_j, \quad z_j := \sum_{i:h(i)=j} g(i) x_i.$$

The z_j use disjoint sets of inputs, so they are uncorrelated. m divides n and the load is optimal (Theorem 3). Thus each slot has exactly k inputs, and $V(z_j) = k\sigma_x^2$. The same steps as for pruning give $\alpha_j = \text{Cov}(y, z_j)/(k\sigma_x^2)$, and

$$\text{Res}(\hat{y}) = \sigma_y^2 - \sum_j \frac{\text{Cov}(y, z_j)^2}{k\sigma_x^2} = \sigma_y^2 \left(1 - \underbrace{\frac{1}{k} \sum_j \left(\sum_{i:h(i)=j} g(i) \rho_i \right)^2}_{\langle \hat{\rho}, \hat{\rho} \rangle_{\text{STABLE-RPS}}} \right). \quad \blacksquare$$

Theorem 5: Interpretation

- ▶ Without compression ($k = 1$, all inputs kept): $\text{Res}(\hat{y}) = \sigma_y^2(1 - \|\rho\|^2)$.
- ▶ The relative loss from compression is

$$\frac{\text{Res}_{\text{comp}} - \text{Res}}{\text{Res}} = \frac{\|\rho\|^2 - \frac{1}{k} \langle \hat{\rho}, \hat{\rho} \rangle}{1 - \|\rho\|^2}.$$

- ▶ Both estimates are unbiased. Thus both methods have the same expected residual $\sigma_y^2(1 - \|\rho\|^2/k)$.
- ▶ The difference is in the variance. By Theorem 4, STABLE-RPS has a smaller variance on average when the correlations ρ_i are not balanced.

Result

Model compression of a linear model is the same problem as norm preservation of ρ . Theorem 4 then favors STABLE-RPS over pruning.

Section 5

SS1: Sketch Structured Transform
(Saedi et al. 2024)

Premise

Random parameter sharing (RPS) gives better quality than pruning at the same memory budget, but it does **not** reduce computation: every virtual weight is still multiplied, so FLOPs and latency stay the same.

Question asked by SS1

Can parameter sharing be restricted so that it **reduces FLOPs and latency**, while keeping the expressivity of RPS?

- ▶ Unstructured sparsity: expressive, but slow on GPUs.
- ▶ Structured sparsity (Monarch) and low rank: fast, but lose quality.
- ▶ SS1: random **and** structured parameter sharing.

Why RPS does not reduce compute

Recall RPS: the full weights come from a small memory through a sketch matrix,

$$\theta = S \mathcal{M}, \quad S[i, h(i)] = g(i), \quad S \in \mathbb{R}^{n \times m}.$$

- In HashedNets, ROBE, ROAST and STABLE-RPS, the weights of **one neuron** almost never share a parameter.

Why RPS does not reduce compute

Recall RPS: the full weights come from a small memory through a sketch matrix,

$$\theta = S \mathcal{M}, \quad S[i, h(i)] = g(i), \quad S \in \mathbb{R}^{n \times m}.$$

- ▶ In HashedNets, ROBE, ROAST and STABLE-RPS, the weights of **one neuron** almost never share a parameter.
- ▶ So one neuron $y = \mathbf{x}^\top \mathbf{w}$ still needs K multiplications. The computation graph does not change.

Why RPS does not reduce compute

Recall RPS: the full weights come from a small memory through a sketch matrix,

$$\theta = S \mathcal{M}, \quad S[i, h(i)] = g(i), \quad S \in \mathbb{R}^{n \times m}.$$

- ▶ In HashedNets, ROBE, ROAST and STABLE-RPS, the weights of **one neuron** almost never share a parameter.
- ▶ So one neuron $y = \mathbf{x}^\top \mathbf{w}$ still needs K multiplications. The computation graph does not change.
- ▶ RPS reduces **memory**, but not **FLOPs** or latency.

Why RPS does not reduce compute

Recall RPS: the full weights come from a small memory through a sketch matrix,

$$\theta = S \mathcal{M}, \quad S[i, h(i)] = g(i), \quad S \in \mathbb{R}^{n \times m}.$$

- ▶ In HashedNets, ROBE, ROAST and STABLE-RPS, the weights of **one neuron** almost never share a parameter.
- ▶ So one neuron $y = \mathbf{x}^\top \mathbf{w}$ still needs K multiplications. The computation graph does not change.
- ▶ RPS reduces **memory**, but not **FLOPs** or latency.

Why RPS does not reduce compute

Recall RPS: the full weights come from a small memory through a sketch matrix,

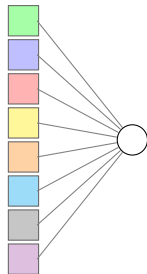
$$\theta = S \mathcal{M}, \quad S[i, h(i)] = g(i), \quad S \in \mathbb{R}^{n \times m}.$$

- ▶ In HashedNets, ROBE, ROAST and STABLE-RPS, the weights of **one neuron** almost never share a parameter.
- ▶ So one neuron $y = \mathbf{x}^\top \mathbf{w}$ still needs K multiplications. The computation graph does not change.
- ▶ RPS reduces **memory**, but not **FLOPs** or latency.

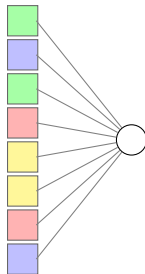
Idea of SS1

Tie parameters **inside** a single neuron, so the shared weights can be added together *before* the multiplication.

Key idea: tie weights inside one neuron



Standard RPS:
8 values, 8 mults

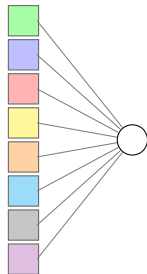


SS1:
4 values, 4 mults

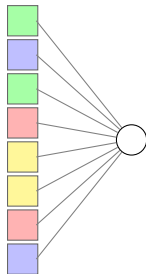
Neuron $y = \mathbf{x}^\top \mathbf{w}$, $\mathbf{x}, \mathbf{w} \in \mathbb{R}^K$. SS1 stores $\mathbf{z} \in \mathbb{R}^{K/c}$ and recovers

$$\mathbf{w}[i] = g(i) \mathbf{z}[h(i)], \quad \text{i.e.} \quad \mathbf{w} = S\mathbf{z}.$$

Key idea: tie weights inside one neuron



Standard RPS:
8 values, 8 mults



SS1:
4 values, 4 mults

Neuron $y = \mathbf{x}^\top \mathbf{w}$, $\mathbf{x}, \mathbf{w} \in \mathbb{R}^K$. SS1 stores $\mathbf{z} \in \mathbb{R}^{K/c}$ and recovers

$$\mathbf{w}[i] = g(i) \mathbf{z}[h(i)], \quad \text{i.e.} \quad \mathbf{w} = S\mathbf{z}.$$

Then

$$y = \mathbf{x}^\top (S\mathbf{z}) = (\mathbf{x}^\top S) \mathbf{z}.$$

- ▶ $\mathbf{x}^\top S$ (a **sketch** of $\mathbf{x}$) needs only additions and subtractions.
- ▶ The product with $\mathbf{z}$ needs only K/c multiplications.

K-coalescing: a GPU-friendly hash

A random h would scatter memory reads. SS1 uses a structured hash.

- ▶ Split $\mathbf{w}$ into **chunks** of size B_K : $C(i) = \lfloor i/B_K \rfloor$, $O(i) = i \bmod B_K$.
- ▶ Group c chunks into a **1D super group**: $G_1(i) = \lfloor C(i)/c \rfloor$.

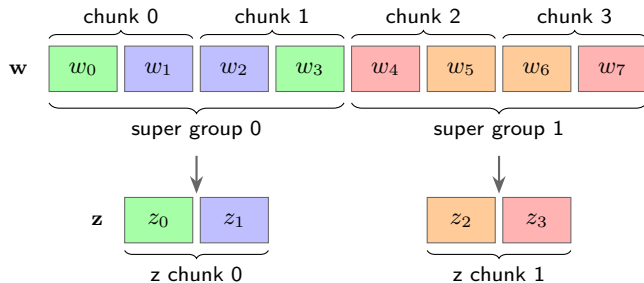
SS1 hash (Eq. 8)

$$h'(i) = G_1(i) B_K + \left(h(G_1(i), C(i)) + O(i) \right) \bmod B_K$$

- 1 Each chunk is a **random rotation** of one contiguous chunk of $\mathbf{z}$, so no two weights in a chunk share a parameter.
- 2 The c chunks of one super group share the same chunk of $\mathbf{z}$, so they share parameters.

K-coalescing: worked example

$K = 8$, $B_K = 2$, $c = 2$, so $\mathbf{z} \in \mathbb{R}^{K/c} = \mathbb{R}^4$: **2 chunks** of size B_K , one per super group.
Offsets: $h(0,0) = 0$, $h(0,1) = 1$, $h(1,2) = 1$, $h(1,3) = 0$.

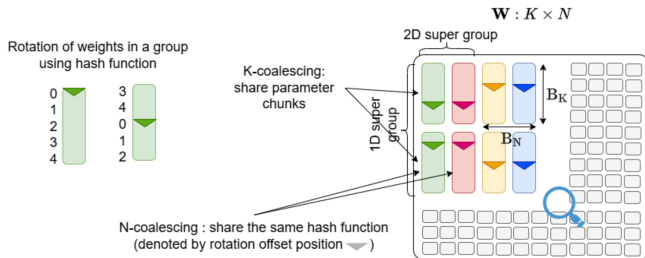


Same colour = same parameter. Each $\mathbf{w}$ chunk is a **rotation** of its super group's $\mathbf{z}$ chunk.
Ignoring the signs g :

$$y = (x_0 + x_3) z_0 + (x_1 + x_2) z_1 + (x_5 + x_6) z_2 + (x_4 + x_7) z_3$$

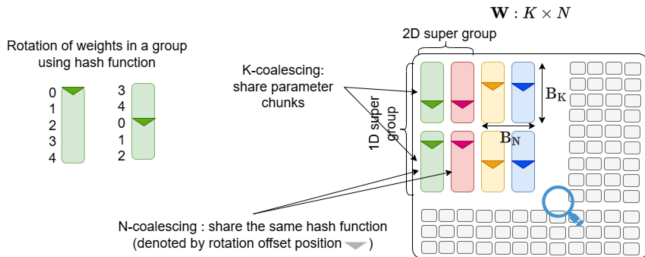
4 multiplications instead of 8, and every read is contiguous.

N-coalescing: share hashes across neurons



- Columns of $\mathbf{W}$ are split into blocks of B_N neurons; all neurons in a block use the **same** h and g .

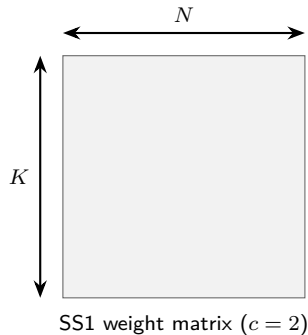
N-coalescing: share hashes across neurons



- Columns of $\mathbf{W}$ are split into blocks of B_N neurons; all neurons in a block use the **same** h and g .
- So one sketched input tile is **reused** by all B_N neurons, and the rest is an ordinary small matmul (tensor cores):

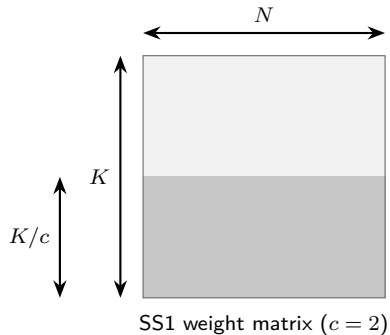
$$\mathbf{y}[:, B_N] = \sum_i \underbrace{\left(\sum_{j=0}^{c-1} \mathbf{x}_{ic+j}^\top S_{i,j} \right)}_{\text{sketch, adds only}} \mathbf{Z}_i[:, : B_N]$$

SS1 weight matrix, step by step



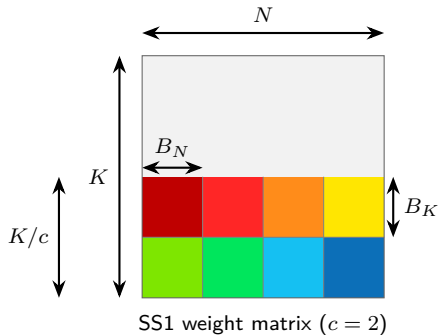
- 1 The virtual weight matrix $\mathbf{W}$ is $K \times N$.

SS1 weight matrix, step by step



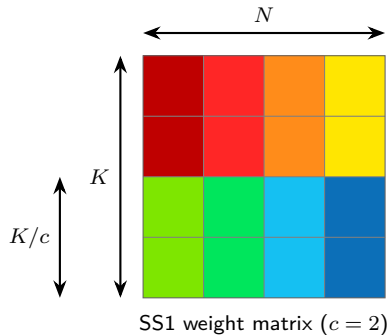
- 1 The virtual weight matrix $\mathbf{W}$ is $K \times N$.
- 2 Only $\mathbf{Z}$, with K/c rows, is stored.

SS1 weight matrix, step by step



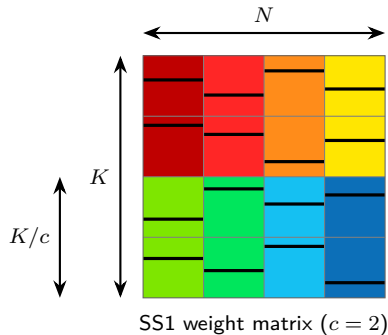
- 1 The virtual weight matrix $\mathbf{W}$ is $K \times N$.
- 2 Only $\mathbf{Z}$, with K/c rows, is stored.
- 3 $\mathbf{Z}$ is cut into tiles of size $B_K \times B_N$.

SS1 weight matrix, step by step



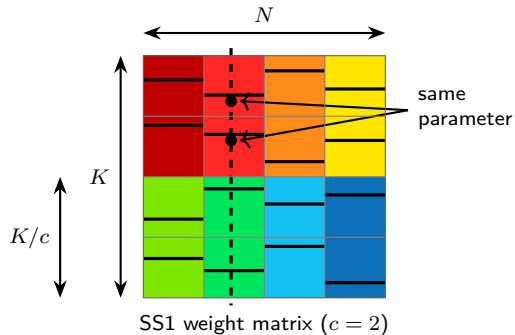
- 1 The virtual weight matrix $\mathbf{W}$ is $K \times N$.
- 2 Only $\mathbf{Z}$, with K/c rows, is stored.
- 3 $\mathbf{Z}$ is cut into tiles of size $B_K \times B_N$.
- 4 Each row of $\mathbf{Z}$ tiles is reused c times in $\mathbf{W}$: these c copies form a super group.

SS1 weight matrix, step by step



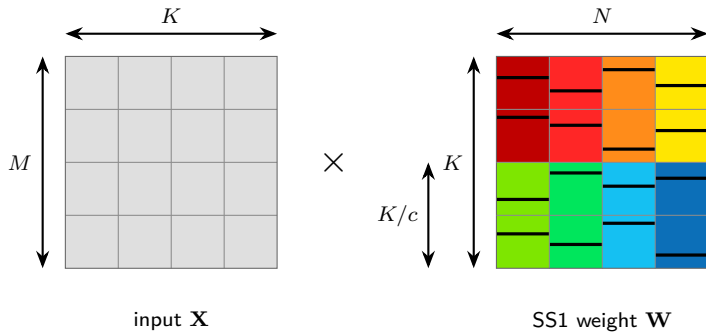
- 1 The virtual weight matrix $\mathbf{W}$ is $K \times N$.
- 2 Only $\mathbf{Z}$, with K/c rows, is stored.
- 3 $\mathbf{Z}$ is cut into tiles of size $B_K \times B_N$.
- 4 Each row of $\mathbf{Z}$ tiles is reused c times in $\mathbf{W}$: these c copies form a super group.
- 5 Each copy is rotated by a random offset (black line). All B_N columns of a tile share it (N-coalescing).

SS1 weight matrix, step by step



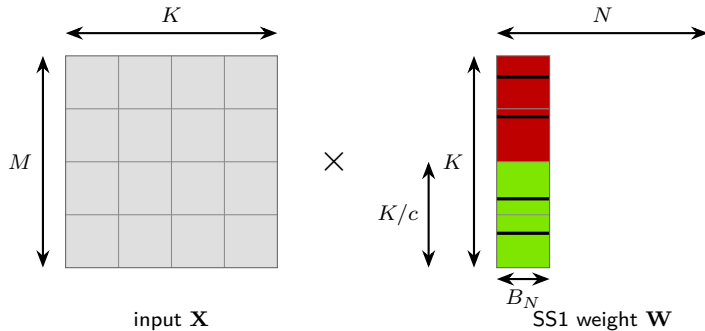
- 1 The virtual weight matrix $\mathbf{W}$ is $K \times N$.
- 2 Only $\mathbf{Z}$, with K/c rows, is stored.
- 3 $\mathbf{Z}$ is cut into tiles of size $B_K \times B_N$.
- 4 Each row of $\mathbf{Z}$ tiles is reused c times in $\mathbf{W}$: these c copies form a super group.
- 5 Each copy is rotated by a random offset (black line). All B_N columns of a tile share it (N-coalescing).
- 6 Inside one neuron (dashed column), different copies hit the same parameter: weights are tied within the neuron.

SS1 forward pass, step by step



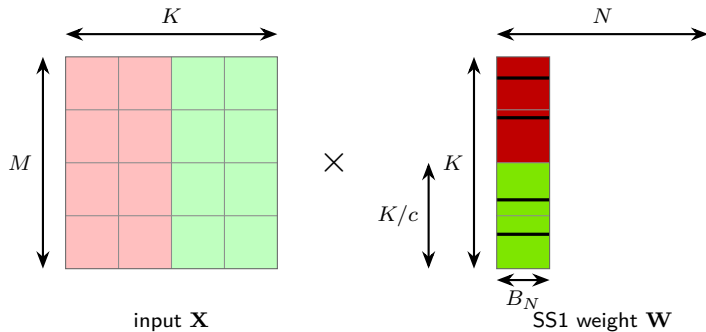
An SS1 layer computes $\mathbf{Y} = \mathbf{X}\mathbf{W}$. The black lines in $\mathbf{W}$ are the rotation offsets.

SS1 forward pass, step by step



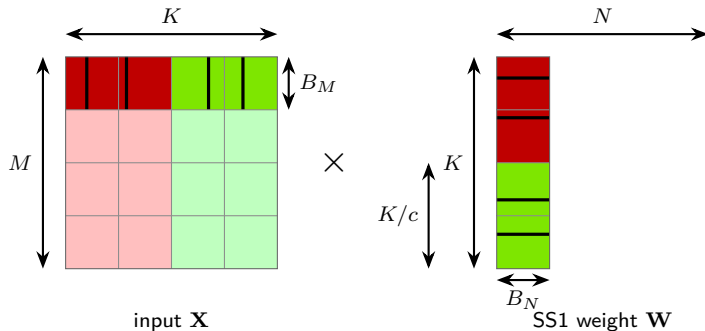
One output tile uses one **column block** of $\mathbf{W}$ (B_N neurons). All its columns share the same offsets.

SS1 forward pass, step by step



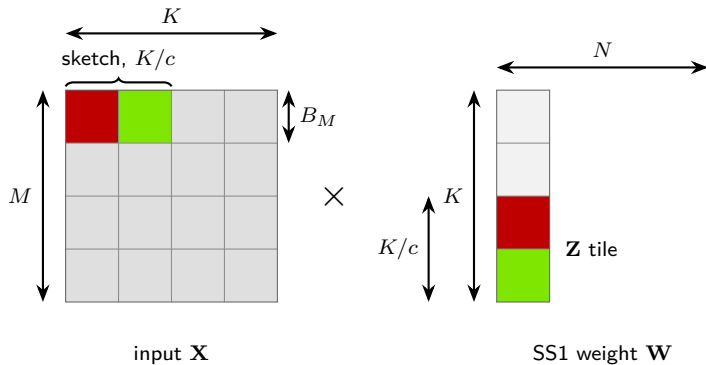
Input chunks 0–1 (red) meet super group 0; chunks 2–3 (green) meet super group 1.

SS1 forward pass, step by step



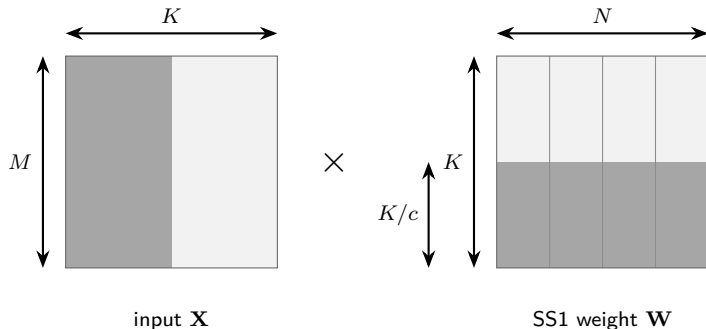
Take one row tile of $\mathbf{X}$ (B_M rows). **Rotate** each input chunk by the offset (and sign) of its weight tile.

SS1 forward pass, step by step



Add the c rotated chunks of each super group: a sketch of width K/c , using only additions. Then do **one small matmul** with the stored $\mathbf{Z}$ tile.

SS1 forward pass, step by step



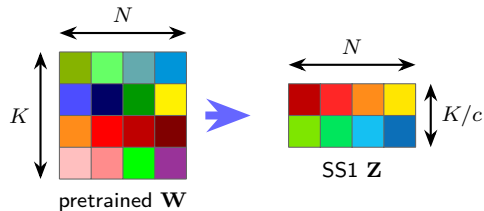
Dense: $M \times K \times N$ multiply-adds. SS1: $M \times \frac{K}{c} \times N$, i.e. **c times fewer**, plus cheap additions. The rotation only changes load addresses (Triton kernel).

Algorithm 1

Algorithm 1	$\mathbf{Y} = \text{SS1}(\mathbf{X}, \mathbf{Z}), \quad \mathbf{X} \in \mathbb{R}^{M \times K}, \mathbf{Z} \in \mathbb{R}^{\frac{K}{c} \times N}, \quad cB_K \mid K$	What it does
1:	$T_x \leftarrow \text{TILE}(B_M, B_K), \quad T_y \leftarrow \text{TILE}(B_M, B_N)$	sketch and output buffers
2:	for $i = 0, \dots, \lceil M/B_M \rceil - 1$ do	row tile of $\mathbf{X}$
3:	for $j = 0, \dots, \lceil N/B_N \rceil - 1$ do	column block of $\mathbf{W}$ (B_N neurons)
4:	$T_y \leftarrow 0$	
5:	for $k = 0, \dots, K/(cB_K) - 1$ do	super group k
6:	$T_x \leftarrow 0$	
7:	for $l = 0, \dots, c - 1$ do	its c chunks
8:	$a \leftarrow \mathbf{X}[iB_M:(i+1)B_M, (kc+l)B_K:(kc+l+1)B_K]$	load one input chunk
9:	$a \leftarrow g(j, k, l) \cdot a[:, (\mathcal{R}(0:B_K) - h(j, k, l)) \bmod B_K]$	rotate and sign
10:	$T_x \leftarrow T_x + a$	add into the sketch
11:	end for	
12:	$T_y \leftarrow T_y + T_x \cdot \mathbf{Z}[kB_K:(k+1)B_K, jB_N:(j+1)B_N]$	one small matmul with $\mathbf{Z}$ tile
13:	end for	
14:	$\mathbf{Y}[iB_M:(i+1)B_M, jB_N:(j+1)B_N] \leftarrow T_y$	write the output tile
15:	end for	
16:	end for	

Repeated for every output tile (i, j) . Only line 12 multiplies: $M \times \frac{K}{c} \times N$ MACs in total (dense: $M \times K \times N$).

Projecting pretrained weights onto SS1

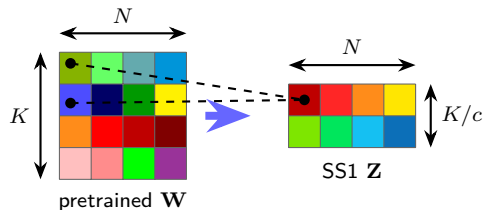


Per neuron, solve least squares with $\mathbf{w} = S\mathbf{z}$:

$$\mathbf{z}^* = \arg \min_{\mathbf{z}} \|\mathbf{w} - S\mathbf{z}\|_2 = (S^\top S)^{-1} S^\top \mathbf{w} = \frac{1}{c} S^\top \mathbf{w}$$

since every slot receives exactly c weights and $g^2 = 1$, so $S^\top S = cI$.

Projecting pretrained weights onto SS1



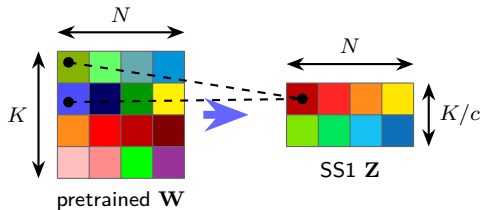
The c weights that share a slot are averaged into it.

Per neuron, solve least squares with $\mathbf{w} = S\mathbf{z}$:

$$\mathbf{z}^* = \arg \min_{\mathbf{z}} \|\mathbf{w} - S\mathbf{z}\|_2 = (S^\top S)^{-1} S^\top \mathbf{w} = \frac{1}{c} S^\top \mathbf{w}$$

since every slot receives exactly c weights and $g^2 = 1$, so $S^\top S = cI$.

Projecting pretrained weights onto SS1



The c weights that share a slot are averaged into it.

Per neuron, solve least squares with $\mathbf{w} = S\mathbf{z}$:

$$\mathbf{z}^* = \arg \min_{\mathbf{z}} \|\mathbf{w} - S\mathbf{z}\|_2 = (S^\top S)^{-1} S^\top \mathbf{w} = \frac{1}{c} S^\top \mathbf{w}$$

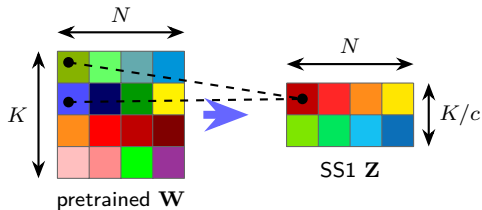
since every slot receives exactly c weights and $g^2 = 1$, so $S^\top S = cI$.

Per $B_K \times B_N$ tile ($P_h = \text{cyclic shift by } h$):

$$\mathbf{z}^{(k,j)} = \frac{1}{c} \sum_{l=0}^{c-1} g_{j,k,l} P_{h_{j,k,l}}^\top \mathbf{w}^{(kc+l,j)}$$

Un-rotate, sign, average the c tiles of a super group: one pass over $\mathbf{W}$, no data (Algorithm 2).

Projecting pretrained weights onto SS1



The c weights that share a slot are averaged into it.

Per neuron, solve least squares with $\mathbf{w} = S\mathbf{z}$:

$$\mathbf{z}^* = \arg \min_{\mathbf{z}} \|\mathbf{w} - S\mathbf{z}\|_2 = (S^\top S)^{-1} S^\top \mathbf{w} = \frac{1}{c} S^\top \mathbf{w}$$

since every slot receives exactly c weights and $g^2 = 1$, so $S^\top S = cI$.

Per $B_K \times B_N$ tile ($P_h = \text{cyclic shift by } h$):

$$\mathbf{z}^{(k,j)} = \frac{1}{c} \sum_{l=0}^{c-1} g_{j,k,l} P_{h_{j,k,l}}^\top \mathbf{w}^{(kc+l,j)}$$

Un-rotate, sign, average the c tiles of a super group:
one pass over $\mathbf{W}$, no data (Algorithm 2).

SS1 + RPS: the tiles of $\mathbf{Z}$ can themselves be stored in an RPS memory (as in ROAST): SS1 controls **compute**, RPS controls **memory**.

Algorithm 2 (projection)

Algorithm 2 $\mathbf{Z}^* = \arg \min_{\mathbf{Z}} \ \mathbf{W} - \text{SS1}(I, \mathbf{Z})\ _F$, $\mathbf{W} \in \mathbb{R}^{K \times N}$, $cB_K \mid K$	Formula
<pre> 1: $\mathbf{Z}^* \leftarrow \mathbf{0}_{K/c \times N}$, $T_w \leftarrow \text{TILE}(B_K, B_N)$ 2: for $j = 0, \dots, \lceil N/B_N \rceil - 1$ do 3: for $k = 0, \dots, K/(cB_K) - 1$ do 4: $T_w \leftarrow 0$ 5: for $l = 0, \dots, c - 1$ do 6: $a \leftarrow \mathbf{W}[(kc+l)B_K:(kc+l+1)B_K, jB_N:(j+1)B_N]$ 7: $a \leftarrow a[(\mathcal{R}(0:B_K) - h(j, k, l)) \bmod B_K, :]$ 8: $T_w \leftarrow T_w + g(j, k, l) \cdot a$ 9: end for 10: $\mathbf{Z}^*[kB_K:(k+1)B_K, jB_N:(j+1)B_N] \leftarrow T_w/c$ 11: end for 12: end for </pre>	column block j super group $k \Rightarrow$ tile $\mathbf{Z}^{(k,j)}$ $\sum_l$ over the c copies $\mathbf{W}^{(kc+l,j)}$ un-rotate: $P_h^\top$ sign $g_{j,k,l}$, add average: $\frac{1}{c}$

$$\mathbf{Z}^{(k,j)} = \frac{1}{c} \sum_{l=0}^{c-1} g_{j,k,l} P_{h_{j,k,l}}^\top \mathbf{W}^{(kc+l,j)}$$

Theorem 2: B_K is a pure latency parameter

Theorem

Let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ be projected to $\hat{\mathbf{x}}, \hat{\mathbf{y}} \in \mathbb{R}^m$ by the SS1 hash with chunk size B_K and compression $c = n/m$, with $cB_K \mid m \mid n$. Then $\langle \hat{\mathbf{x}}, \hat{\mathbf{y}} \rangle$ is unbiased and

$$V = \frac{1}{B_K} \sum_{i \neq j} I(i, j) (x_i^2 y_j^2 + x_i y_i x_j y_j),$$

where $I(i, j) = 1$ if i, j lie in *different chunks* of the *same super group*. Under a random permutation p of the coordinates,

$$\mathbb{E}_p[V] = \frac{c-1}{n-1} \sum_{i \neq j} (x_i^2 y_j^2 + x_i y_i x_j y_j).$$

Meaning: B_K disappears from the average variance. So B_K can be tuned for speed only, with no effect on quality.

Theorem 1: why quantization combines well with SS1

Unit vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ in F -bit precision. Here c is the **fraction** of memory kept ($c = b/F$ or $c = m/n$). For some $k \in [1, 2]$:

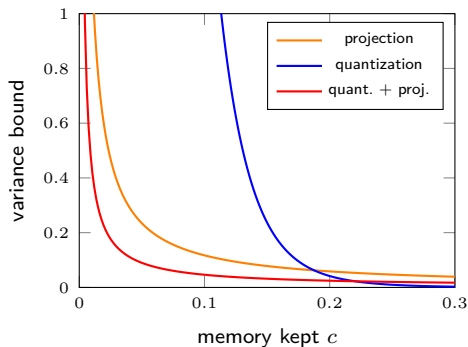
$$Q(c) = \frac{2}{2^{2cF}} + \frac{n}{2^{4cF}}, \quad P(c) = \frac{k}{cn}.$$

Theorem

$V_{\text{quant}} \leq Q(c)$ and $V_{\text{proj}} \leq P(c)$. Quantization with c_q followed by projection with c_p ($c = c_p c_q$) gives

$$V_{pq} \leq Q(c_q) \left(1 + \frac{1}{c_p}\right) + P(c_p).$$

Each bound explodes at strong compression, so two mild compressions **beat** one strong one.



$n = 128, F = 16, k = 1.5, c_q = 0.3$ (redrawn Fig. 2)

Results: GPT-2 trained from scratch (WikiText-103)

Model	#params	PPL (GPT2-S)	loss (GPT2-S)	Model Latency (GPT2-S)		Model Latency (GPT2-M)		Model Latency (GPT2-L)	
				b=16	b=32	b=16	b=32	b=8	b=16
Original Model	124M	20.43	3.004	80.4	160	181	oom	190	376
SS1-2x	96M	19.458	2.957	78.5	154.9	172.4	342.7	169.9	334.2
SS1-4x	81M	19.998	2.991	75.1	148.6	163.3	330	157.6	310.4
SS1-8x	74M	20.68	3.025	74.3	145.6	153	309.3	147.9	291
Monarch (nb=2)	103M	19.606	2.964	90	178.2	201.8	401.6	194.8	386.5
Monarch (nb=4)	85M	20.611	3.022	88.2	174.2	186.9	371	178	363.7
Monarch (nb=8)	76M	22.839	3.119	85.5	169.4	181.1	359.4	173.6	342.8
Lowrank~2x	96	19.756	2.979	99	195.2	231.5	456.3	150	295.3
Lowrank~4x	81M	20.152	2.993	84.8	165.6	159.1	314.2	137	266.6
Lowrank~8x	74M	21.211	3.05	72.2	141.4	147.3	290.7	131	259

SS1 has the **best perplexity** at every size and is **faster than the original** in every column. LowRank is sometimes faster, but with worse PPL.

Results: MLP Mixer trained from scratch (CIFAR)

Model	#params	Accuracy (C100)	Accuracy (C10)	Model Latency (MM-S)		Model Latency (MM-M)		Model Latency (MM-L)	
				b=512	b=1K	b=512	b=1K	b=512	b=1K
Original Model	1.1M	0.6751	0.9083	7.9	15.4	38.1	75.4	122.5	246.1
SS1-2x	613K	0.6692	0.8918	8.3	14.7	36.2	71.7	114.3	227.3
SS1-4x	347K	0.6463	0.8695	8.2	14.7	33.6	66.7	100.4	199.2
SS1-8x	282K	0.6068	0.8554	8.2	14.5	33.2	65.5	95.0	188.1
Monarch (nb=2)	750K	0.6738	0.8729	12.5	20.8	47.7	94.4	135.9	272.9
Monarch (nb=4)	422K	0.6261	0.8491	12.9	21.3	46.2	91.7	119.9	239.9
Monarch (nb=8)	259K	0.5731	0.8282	14.2	23.4	46.0	91.4	115.3	229.8
Lowrank~2x	750K	0.6628	0.8907	9.7	18.4	38.6	78.1	114.6	231.7
Lowrank~4x	422K	0.6299	0.8639	9.5	18.1	35.9	72.0	100.6	204.2
Lowrank~8x	259K	0.5353	0.8034	9.4	17.9	35.6	71.3	94.9	191.1

SS1 keeps the **best C10 accuracy** at each budget. On MM-M and MM-L it is **faster than the original and Monarch**, and on par with LowRank.

Results: projecting pretrained models

SS1 projection followed by finetuning

	BERT-Large		BERT-Base	
	#par	GLUE	#par	GLUE
baseline (reported)	335M	80.4	109M	78.6
baseline (our run)	335M	82.2	109M	82.16
SS1	181M	79.6 (± 0.203)	66M	79.9 (± 0.066)

SS1 projection without finetuning

	#par	Llama-3-8B	
		MMLU	WINO
baseline (our run)	8B	65.05 (± 0.0038)	76.1 (± 0.011)
SS1	6B	61.26 (± 0.0039)	69.93 (± 0.012)

- ▶ Only **insensitive layers** are projected (8x), chosen with a small calibration set (RTE for BERT, tinyMMLU / tinyWinogrande for Llama).
- ▶ Inference speedup: **1.31x** for BERT-Large (batch 128) and **1.11x** for Llama-3-8B.
- ▶ BERT keeps most of its GLUE score after finetuning; Llama works reasonably well **even without any finetuning**.

Thank You!