A Training Procedure

The factor tensors are initialized with entries randomly drawn from a normal distribution: $\mathcal{N}(0, 1/\sqrt{n})$. We employ full-batch gradient descent to optimize the regularized loss with learning rate of 0.5 and momentum of 0.5. For the small scale experiments in Section 6, the HyperCube regularizer coefficient is set to $\epsilon = 0.1$. For the larger scale experiments in Section 7, we use $\epsilon = 0.05$ for HyperCube and $\epsilon = 0.01$ for HyperCube-SE. See Appendix D for a discussion of hyperparameter sensitivity. Each experiment quickly runs within a few minutes on a single GPU.

ϵ-scheduler To overcome the limitations in standard regularized optimization, which often prevents full convergence to the ground truth (D), we employ ϵ-scheduler: Once the model demonstrates sufficient convergence (*e.g.*, the average imbalance falls below a threshold of 10^{-5}), the scheduler sets the regularization coefficient ϵ to 0. This allows the model to fully fit the training data. The effect of ϵ-scheduler on convergence is discussed in Appendix H.3.

The main implementation of HyperCube is shown below. Code repository is available at https://anonymous.4open.science/r/DeepTensorFactorization4GroupRep-EB92/

```python
import torch

def HyperCube_product(A,B,C):
    return torch.einsum('aij,bjk,cki->abc', A,B,C) / A.shape[0]

def HyperCube_regularizer(A,B,C):
    def helper(M,N):
        MM = torch.einsum('aij,bij->ab', M,M)
        NN = torch.einsum('aij,bij->ab', N,N)
        return (MM @ NN.T).trace()
    return (helper(A,B) + helper(B,C) + helper(C,A) ) / A.shape[0]
```

B List of Binary Operations

Here is the list of binary operations from Power et al. (2022) that are used in Section 7 (with $p = 97$).

- (add) $a \circ b = a + b \pmod p$ for $0 \leq a, b < p$. (Cyclic Group)
- (sub) $a \circ b = a - b \pmod p$ for $0 \leq a, b < p$.
- (div) $a \circ b = a/b \pmod p$ for $0 \leq a < p, 0 < b < p$.
- (cond) $a \circ b = [a/b \pmod p)$ if b is odd, otherwise $a - b \pmod p$)] for $0 \leq a, b < p$.
- (quad1) $a \circ b = a^2 + b^2 \pmod p$ for $0 \leq a, b < p$.
- (quad2) $a \circ b = a^2 + ab + b^2 \pmod p$ for $0 \leq a, b < p$.
- (quad3) $a \circ b = a^2 + ab + b^2 + a \pmod p$ for $0 \leq a, b < p$.
- (cube1) $a \circ b = a^3 + ab \pmod p$ for $0 \leq a, b < p$.
- (cube2) $a \circ b = a^3 + ab^2 + b \pmod p$ for $0 \leq a, b < p$.
- (ab in S_5) $a \circ b = a \cdot b$ for $a, b \in S_5$. (Symmetric Group)
- (aba^{-1} in S_5) $a \circ b = a \cdot b \cdot a^{-1}$ for $a, b \in S_5$.
- (aba in S_5) $a \circ b = a \cdot b \cdot a$ for $a, b \in S_5$.

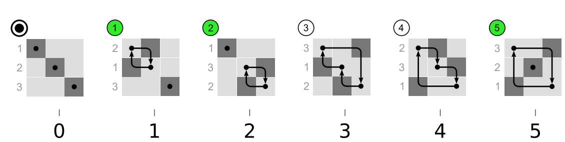

Figure 8: Elements of the symmetric group S_3 illustrated as permutations of 3 items. Green color indicates *odd* permutations, and white indicates *even* permutations. Adapted from https://en.wikipedia.org/wiki/Symmetric_group.

C UNDERSTANDING HYPERCUBE REGULARIZER

To gain an intuitive understanding of the HyperCube regularizer, consider a simplified, scalar HyperCube model $t = abc$ with $a, b, c \in \mathbb{R}$. Minimizing the L_2 regularizer $a^2 + b^2 + c^2$ subject to the data constraint $t = 1$ yields the usual balanced condition:

$$a = b = c = 1. \tag{11}$$

In contrast, the HyperCube regularizer eq (6) becomes:

$$\begin{aligned}\mathcal{H}(a,b,c) &= \left(\frac{\partial t}{\partial a}\right)^2 + \left(\frac{\partial t}{\partial b}\right)^2 + \left(\frac{\partial t}{\partial c}\right)^2 \\ &= \left(\frac{t}{a}\right)^2 + \left(\frac{t}{b}\right)^2 + \left(\frac{t}{c}\right)^2 \\ &= \tilde{a}^2 + \tilde{b}^2 + \tilde{c}^2,\end{aligned} \tag{12}$$

where, given the constraint $t = 1$, we defined the substitute variables as $\tilde{a} \equiv 1/a$, $\tilde{b} \equiv 1/b$, and $\tilde{c} \equiv 1/c$. Minimizing eq (12) subject to the constraint $\tilde{a}\tilde{b}\tilde{c} = 1$ yields the balanced condition $\tilde{a} = \tilde{b} = \tilde{c} = 1$, or equivalently,

$$\frac{1}{a} = \frac{1}{b} = \frac{1}{c} = 1. \tag{13}$$

This is the reciprocal of the L_2 regularizer's balanced condition eq (11), although the solutions are identical in this scalar case. This example demonstrates that the HyperCube regularizer instills a "reciprocal" bias compared to the L_2 regularizer.

C.0.1 BALANCED CONDITION FOR L_2 REGULARIZATION

In contrast, a different balanced condition applies for L_2 Regularization:

$$\xi_I^{L_2} = \xi_J^{L_2} = \xi_K^{L_2} = 0, \tag{14}$$

where $\xi_I^{L_2} = A_a^\dagger A_a - B_b B_b^\dagger$, $\xi_J^{L_2} = B_b^\dagger B_b - C_c C_c^\dagger$, and $\xi_K^{L_2} = C_c^\dagger C_c - A_a A_a^\dagger$. Analogous matrix-version of this balanced condition has been derived in prior works for deep linear networks (Arora et al., 2019; Saxe et al., 2014), which leads to balanced singular modes across the layers: *i.e.* the adjacent layers share the same singular values and singular vector matrices. Crucially, this result shows how L_2 regularization promotes low-rank solutions, since the L_2 loss on individual factors is equivalent to penalizing $\sum_i |\sigma_i|^{2/L}$, where σ_i is the singular value of the end-to-end input-output map, and L is the number of layers. This is called the Schatten norm minimization.

D HYPERPARAMETER SENSITIVITY ANALYSIS

We tested HyperCube across a wide range of hyperparameter settings, including learning rate, regularization coefficient, and weight initialization scale. Figure 9 shows the final test accuracy and Figure 10 shows the number of training steps to achieve 100% test accuracy across a subset of tasks from Appendix B under a fixed training budget of 1000 training steps.

HyperCube exhibits robust performance over the range of hyperparameter settings. Notably, increasing the learning rate or regularization coefficient primarily raises the convergence speed without significantly affecting the final test accuracy. The learning dynamics starts to become unstable at large learning rate (lr $= 1.5$) or regularization coefficient ($\epsilon = 0.1$). The weight initialization scale has no effect on either the final test accuracy or the convergence speed.

This robustness, particularly to weight initialization scale and regularization strength, is noteworthy. Deep neural networks exhibit a saddle point with zero Hessian at zero weights (Kawaguchi, 2016) which becomes a local minimum under L_2 regularization. This local minimum can cause the network weights to collapse to zero when initialized with small values or under strong regularization. (This mechanism also promotes low-rank solutions in L_2-regularized deep neural networks.)

In contrast, HyperCube's quartic regularization loss, also featuring zero Hessian at zero weights, maintains the saddle point at zero. The absence of local minimum at zero prevents weight collapse, contributing to significantly robust learning dynamics and promoting the emergence of full-rank unitary representations in HyperCube.

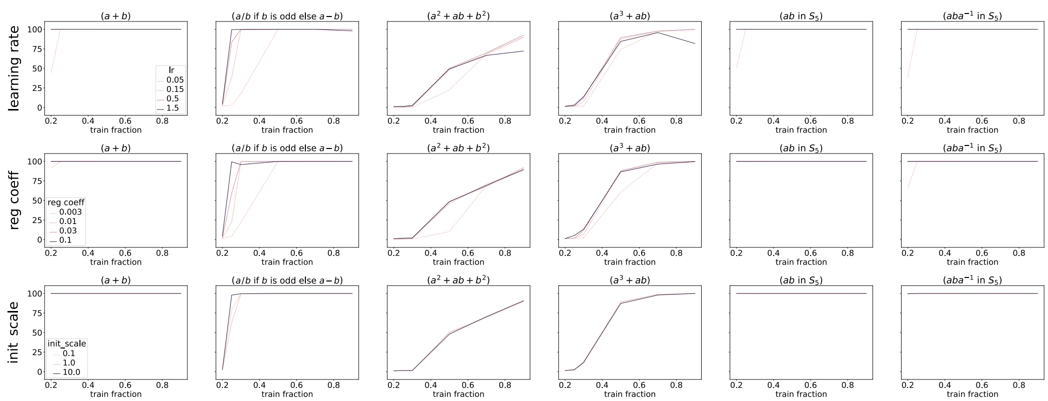

Figure 9: **Test accuracy vs Hyperparameters** : (Top) learning rate, (Middle) regularization strength, and (Bottom) weight initialization scale. Trained under a fixed training budget of 1000 steps. Default hyperparameter setting: lr $= 0.5$, reg coeff $\epsilon = 0.05$, init scale $= 1.0$.

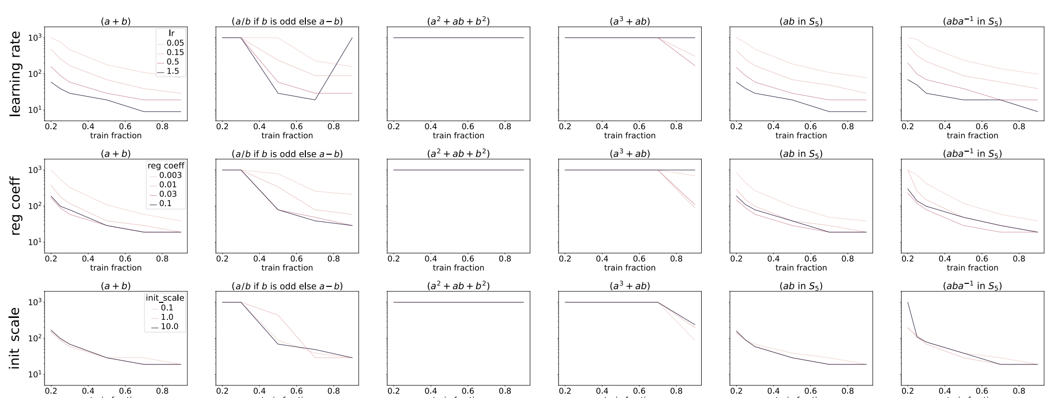

Figure 10: **Steps to 100% accuracy vs Hyperparameters** : Same settings as Fig 9, but showing the number of training steps to achieve 100% test accuracy.

E RUN-TIME COMPLEXITY

We empirically evaluate the run-time complexity of HyperCube. As expected, CPU execution time scales as $O(n^3)$. However, due to the efficient parallelization of `einsum` operations in PyTorch (See Appendix A), GPU execution time remains nearly constant with increasing n (up to $n = 200$, the maximum size that fits in the 16GB memory of a Tesla V100 GPU). This demonstrates the practical efficiency of HyperCube when leveraging GPU acceleration.

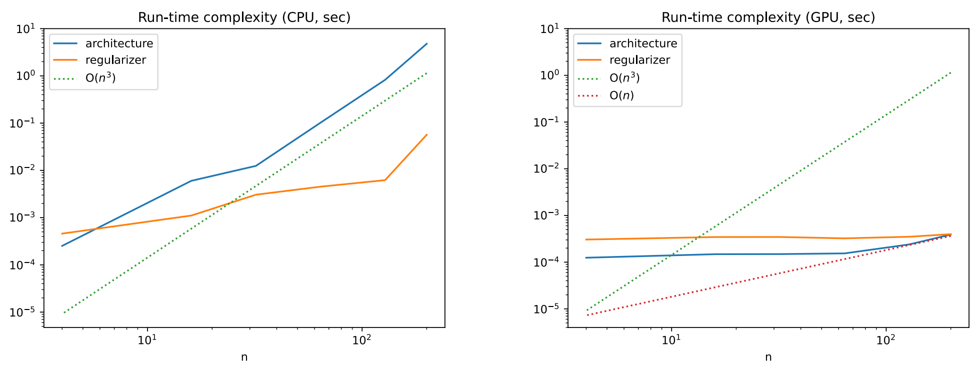

Figure 11: **Run-time complexity** for computing the HyperCube architecture (eq (4)) and regularizer (eq (6)) as functions of n. (Left) Run-time on CPU. (Right) Run-time on GPU (Tesla V100 16GB). Results are averaged over 100 runs.

F ALTERNATIVE TENSOR FACTORIZATIONS

HyperCube distinguishes itself from conventional tensor factorization architectures, which typically employ lower-order, matrix factors for decomposition: *e.g.*, Tucker and CP decomposition. This difference is crucial for capturing the rich structure of binary operations.

Tucker Decomposition (Tucker, 1966) employs a core tensor M and three matrix factors:

$$T_{abc} = \frac{1}{n} \sum_{i,j,k} M_{ijk} A_{ai} B_{bj} C_{ck}, \tag{15}$$

While flexible, Tucker decomposition suffers from a critical limitation: In eq (15), the role of matrix factors is limited to simply mapping individual *external* indices to individual *internal* indices (e.g. A maps a to i). This presents a recursive challenge, since learning the algebraic relationships between (a, b, c) in T requires learning the relationships between (i, j, k) in M, which is not inherently simplifying the core learning problem. Consequently, Tucker decomposition severely overfits the training data and fails to generalize to unseen examples (Figure 12).

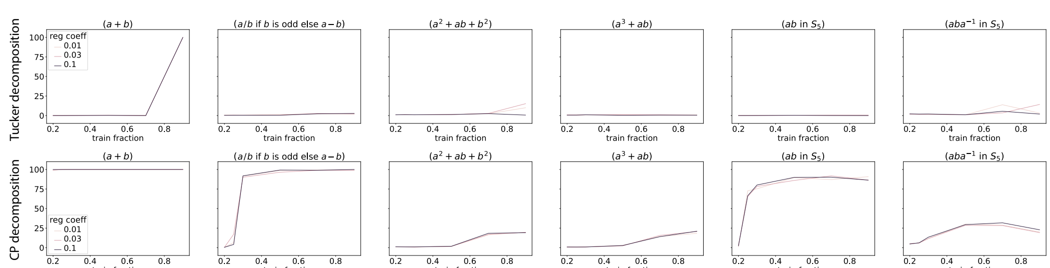

Figure 12: **Alternative Tensor Factorization Methods**: (Top) CP decomposition and (Bottom) Tucker decomposition, trained across a range of L_2 regularization strengths.

CP Decomposition CP decomposition utilizes only matrix factors for decomposition:

$$T_{abc} = \frac{1}{n} \sum_k A_{ak} B_{bk} C_{ck}. \tag{16}$$

This is equivalent to[4] HyperCube with diagonal embeddings (*i.e.* $A_{aki} = A_{ak}\delta_{ki}$, $B_{bij} = B_{bi}\delta_{ij}$, $C_{cjk} = C_{cj}\delta_{jk}$), since

$$\sum_{ijk} A_{aki} B_{bij} C_{cjk} = \sum_{ijk} A_{ak} B_{bi} C_{cj} \delta_{ki}\delta_{ij}\delta_{jk} = \sum_k A_{ak} B_{bk} C_{ck}. \tag{17}$$

Therefore, CP decomposition can only fully capture commutative Abelian groups (e.g modular addition), which admit diagonal representations (*i.e.*, 1×1 irreps) in $K = \mathbb{C}$, but it lacks the expressive power to capture more complex opereations. In experiments (Figure 12), CP decomposition indeed shows reasonable performance only for the modular addition task, struggling to generalize to other structures in data.

G BAND-DIAGONAL HYPERCUBE

As mentioned above, HyperCube with diagonal embeddings lacks the capacity to effectively capture general group structures. However, the regular representation of a group generally decomposes into a direct sum of smaller irreducible representations, resulting in a sparse, block-diagonal matrix structure. Such block-diagonal structure can be effectively captured within the parameter space of *band-diagonal* matrices.

Therefore, to enhance the scalability of HyperCube, we explore the band-diagonal variant where the factor matrices are constrained to have a fixed bandwidth around the diagonal. This reduces the model's parameter count from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$, offering significant computational advantages.

Figure 13 compares the performance of the full HyperCube and the band-diagonal HyperCube with a bandwidth of 8 on a subset of tasks from Appendix B ($n = 97$ or 120). Remarkably, the band-diagonal version exhibits comparable performance to the full HyperCube model, demonstrating its effectiveness in capturing group structures even with a significantly reduced number of parameters. This result highlights the potential of band-diagonal HyperCube for scaling to larger problems.

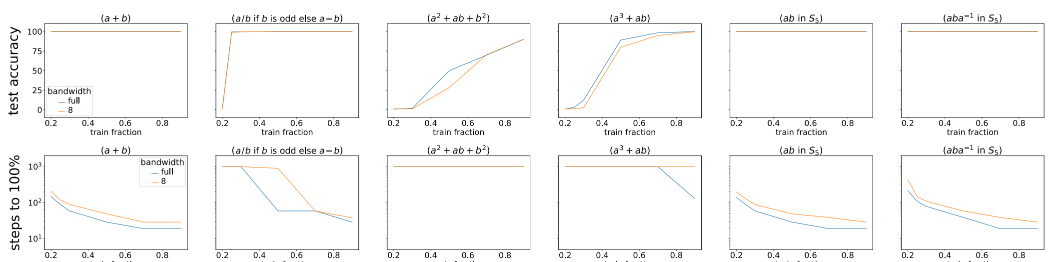

Figure 13: Full HyperCube vs Band-diagonal HyperCube model. (Top) final test accuracy, and (Bottom) steps to 100% test accuracy. lr $= 0.5$, reg coeff $\epsilon = 0.05$, init scale $= 1.0$.

[4] CP decomposition can also be viewed as a special case of Tucker decomposition with a fixed core tensor

$$M_{ijk} = 1 \quad \text{if } i = j = k, \quad 0 \quad \text{otherwise.}$$

H Deferred Proofs

H.1 Proof of Lemma 5.1 on Balanced Condition of HyperCube

Here, we derive the balanced condition eq (7). The gradient of the regularized loss $\mathcal{L} = \mathcal{L}_o(T; D) + \epsilon\mathcal{H}(A, B, C)$ is

$$\begin{aligned}
\nabla_{A_a}\mathcal{L} &= \frac{1}{n}((\nabla_{T_{abc}}\mathcal{L}_o)\, C_c^\dagger B_b^\dagger + 2\epsilon(A_a(B_b B_b^\dagger) + (C_c^\dagger C_c)A_a)), \\
\nabla_{B_b}\mathcal{L} &= \frac{1}{n}((\nabla_{T_{abc}}\mathcal{L}_o)\, A_a^\dagger C_c^\dagger + 2\epsilon(B_b(C_c C_c^\dagger) + (A_a^\dagger A_a)B_b)), \\
\nabla_{C_c}\mathcal{L} &= \frac{1}{n}((\nabla_{T_{abc}}\mathcal{L}_o)\, B_b^\dagger A_a^\dagger + 2\epsilon(C_c(A_a A_a^\dagger) + (B_b^\dagger B_b)C_c)),
\end{aligned} \tag{18}$$

where $\nabla_{A_a}\mathcal{L} \equiv \partial\mathcal{L}/\partial A_a$, $\nabla_{B_b}\mathcal{L} \equiv \partial\mathcal{L}/\partial B_b$, $\nabla_{C_c}\mathcal{L} \equiv \partial\mathcal{L}/\partial C_c$, and $\nabla_{T_{abc}}\mathcal{L}_o \equiv \partial\mathcal{L}_o/\partial T_{abc}$.

Define the *imbalances* as the differences of loss gradients:

$$\begin{aligned}
\xi_I &\equiv \frac{n}{2\epsilon}(A_a^\dagger(\nabla_{A_a}\mathcal{L}) - (\nabla_{B_b}\mathcal{L})B_b^\dagger) = A_a^\dagger(C_c^\dagger C_c)A_a - B_b(C_c C_c^\dagger)B_b^\dagger \\
\xi_J &\equiv \frac{n}{2\epsilon}(B_b^\dagger(\nabla_{B_b}\mathcal{L}) - (\nabla_{C_c}\mathcal{L})C_c^\dagger) = B_b^\dagger(A_a^\dagger A_a)B_b - C_c(A_a A_a^\dagger)C_c^\dagger \\
\xi_K &\equiv \frac{n}{2\epsilon}(C_c^\dagger(\nabla_{C_c}\mathcal{L}) - (\nabla_{A_a}\mathcal{L})A_a^\dagger) = C_c^\dagger(B_b^\dagger B_b)C_c - A_a(B_b B_b^\dagger)A_a^\dagger
\end{aligned}$$

Setting the gradient to zero yields the balanced condition at stationary points, $\xi_I = \xi_J = \xi_K = 0$, which proves Lemma 5.1. Note that imbalance terms are defined to cancel out the $\nabla_{T_{abc}}\mathcal{L}_o$ terms. Therefore, the balanced condition is independent of the loss function $\mathcal{L}_o$.

H.2 Proof of Lemma 5.4

Proof. The constraint on Frobenius norm can be integrated with the regularizer into an augmented loss via the Lagrange multiplier λ

$$\mathcal{H} + \lambda(\mathcal{F} - constant), \tag{19}$$

where $\mathcal{F} \equiv \frac{1}{n}\operatorname{Tr}\left[A_a^\dagger A_a + B_b^\dagger B_b + C_c^\dagger C_c\right]$ is the Frobenius norm .

The gradient of eq (19) with respect to A_a is proportional to

$$\nabla_{A_a}(\mathcal{H} + \lambda\mathcal{F}) \propto A_a(B_b B_b^\dagger) + (C_c^\dagger C_c)A_a + \lambda A_a. \tag{20}$$

In the case of C-unitary factors B and C, all terms in eq (20) become aligned to A_a, *i.e.*

$$\nabla_{A_a}(\mathcal{H} + \lambda\mathcal{F}) \propto (\alpha_B^2 + \alpha_C^2 + \lambda)A_a. \tag{21}$$

and thus an appropriate value for the Lagrange multiplier λ can be found to vanish the gradient, which confirms stationarity. This result also applies to gradient with respect to B_b and C_c by the symmetry of parameterization. □

H.3 Persistence of Group Representation

The following lemma demonstrates a key property of our model's convergence behavior: once a group representation is learned, the solution remains within this representational form throughout optimization.

Lemma H.1. *Let D represent a group operation table. Once gradient descent of the regularized loss eq (5) converges to a group representation (including scalar multiples), i.e.*

$$A_a = \alpha_{A_a}\varrho(a),\; B_b = \alpha_{B_b}\varrho(b),\; C_c = \alpha_{C_c}\varrho(c)^\dagger, \tag{22}$$

the solution remains within this representation form.

Proof. For the squared loss

$$\mathcal{L}_o(T;D) = \sum_{(a,b,c)\in\Omega_{\text{train}}} (T_{abc} - D_{abc})^2, \tag{23}$$

the gradient with respect to A_a eq (18) becomes

$$\nabla_{A_a}\mathcal{L} = \frac{1}{n}(\Delta_{abc} M_{abc} C_c^\dagger B_b^\dagger + \epsilon(A_a(B_b B_b^\dagger) + (C_c^\dagger C_c)A_a)) \tag{24}$$

where $\Delta \equiv T - D$ is the constraint error, and M is the mask indicating observed entries in the train set.

Substituting the group representation form eq (22) into eq (24), we get:

$$\frac{1}{n}\epsilon(A_a(B_b B_b^\dagger) + (C_c^\dagger C_c)A_a) = 2\epsilon\alpha_{A_a}\alpha^2\varrho(a), \tag{25}$$

for the last two terms, where $\alpha^2 = \sum_b \alpha_{B_b}^2/n = \sum_c \alpha_{C_c}^2/n$.

Since the product tensor is

$$T_{abc} = \frac{1}{n}\operatorname{Tr}[A_a B_b C_c] = \frac{1}{n}\alpha_{A_a}\alpha_{B_b}\alpha_{C_c}\operatorname{Tr}[\varrho(a)\varrho(b)\varrho(c)^\dagger] = \alpha_{A_a}\alpha_{B_b}\alpha_{C_c}D_{abc},$$

and $D_{abc} = \delta_{a\circ b,c} = \delta_{a,c\circ b^{-1}}$ (δ is the Kronecker delta function), the first term in eq (24) becomes

$$\begin{aligned}\frac{1}{n}\sum_{b,c}\Delta_{abc}M_{abc}C_c^\dagger B_b^\dagger &= \frac{1}{n}\sum_{b,c}\delta_{a\circ b,c}M_{abc}(\alpha_{A_a}\alpha_{B_b}\alpha_{C_c} - 1)\alpha_{B_b}\alpha_{C_c}\varrho(c\circ b^{-1}) \\ &= \frac{1}{n}\sum_b M_{ab(a\circ b)}(\alpha_{A_a}\alpha_{B_b}\alpha_{C_{a\circ b}} - 1)\alpha_{B_b}\alpha_{C_{a\circ b}}\varrho(a).\end{aligned} \tag{26}$$

Note that both eq (26) and eq (25) are proportional to $\varrho(a)$. Consequently, we have $\nabla_{A_a}\mathcal{L} \propto \varrho(a)$. Similar results for other factors can also be derived: $\nabla_{B_b}\mathcal{L} \propto \varrho(b)$, and $\nabla_{C_c}\mathcal{L} \propto \varrho(c)^\dagger$. This implies that gradient descent preserves the form of the group representation (eq (22)), only updating the coefficients $\alpha_{A_a}, \alpha_{B_b}, \alpha_{C_c}$. □

Effect of ϵ-Scheduler Lemma H.1 holds true even when ϵ gets modified by ϵ-scheduler, which reduces ϵ to 0. In this case, the coefficients converge to $\alpha_{A_a} = \alpha_{B_b} = \alpha_{C_c} = 1$, resulting in the exact group representation form eq (9).

I GROUP CONVOLUTION AND FOURIER TRANSFORM

I.1 FOURIER TRANSFORM ON GROUPS

The Fourier transform of a function $f : G \to \mathbb{R}$ at a representation $\varrho : G \to \mathrm{GL}(d_\varrho, \mathbb{R})$ of G is

$$\hat{f}(\varrho) = \sum_{g \in G} f(g)\varrho(g). \tag{27}$$

For each representation ϱ of G, $\hat{f}(\varrho)$ is a $d_\varrho \times d_\varrho$ matrix, where d_ϱ is the degree of ϱ.

I.2 DUAL GROUP

Let $\hat{G}$ be a complete set indexing the irreducible representations of G up to isomorphism, called the *dual group*, thus for each ξ we have an irreducible representation $\varrho_\xi : G \to U(V_\xi)$, and every irreducible representation is isomorphic to exactly one ϱ_ξ.

I.3 INVERSE FOURIER TRANSFORM

The inverse Fourier transform at an element g of G is given by

$$f(g) = \frac{1}{|G|} \sum_{\xi \in \hat{G}} d_{\varrho_\xi} \operatorname{Tr}\left[\varrho_\xi(g^{-1})\hat{f}(\varrho_\xi)\right]. \tag{28}$$

where the summation goes over the complete set of irreps in $\hat{G}$.

I.4 GROUP CONVOLUTION

The convolution of two functions over a finite group $f, g : G \to \mathbb{R}$ is defined as

$$(f * h)(c) \equiv \sum_{b \in G} f\left(c \circ b^{-1}\right) h(b) \tag{29}$$

I.5 FOURIER TRANSFORM OF GROUP CONVOLUTION

Fourier transform of a convolution at any representation ϱ of G is given by the matrix multiplication

$$\widehat{f * h}(\varrho) = \hat{f}(\varrho)\hat{h}(\varrho). \tag{30}$$

In other words, in Fourier representation, the group convolution is simply implemented by the matrix multiplication.

Proof.

$$\begin{aligned}
\widehat{f * h}(\varrho) &\equiv \sum_c \varrho(c) \sum_b f(c \circ b^{-1})h(b) &(31)\\
&= \sum_c \varrho(c) \sum_{a,b} f(a)h(b)\delta_{(a, c\circ b^{-1})} &(32)\\
&= \sum_{a,b} f(a)h(b) \sum_c \varrho(c)\delta_{(a\circ b, c)} &(33)\\
&= \sum_{a,b} f(a)h(b)\varrho(a \circ b) &(34)\\
&= \sum_a f(a)\varrho(a) \sum_b h(b)\varrho(b) &(35)\\
&= \hat{f}(\varrho)\hat{h}(\varrho). &(36)
\end{aligned}$$

where δ is the Kronecker delta function, and the equivalence between $a = c \circ b^{-1}$ and $a \circ b = c$ is used between the second and the third equality. □

J Group Convolution and Fourier Transform in HyperCube

HyperCube shares a close connection with group convolution and Fourier transform. On finite groups, the Fourier transform generalizes classical Fourier analysis to functions defined on the group: $f : G \to \mathbb{R}$. Instead of decomposing by frequency, it uses the group's irreducible representations $\{\varrho_\xi\}$, where ξ indexes the irreps (See Appendix I.2). A function's Fourier component at ξ is defined as:

$$\hat{f}_\xi \equiv \sum_{g \in G} f(g)\varrho_\xi(g). \tag{37}$$

Fourier Transform in HyperCube The Fourier transform perspective offers a new way to understand how HyperCube with a group representation eq (9) processes general input vectors. Consider a vector f representing a function, *i.e.*, $f_g = f(g)$. Contracting f with a model factor A (or B) yields:

$$\hat{f} \equiv f_g A_g = \sum_{g \in G} f(g)\varrho(g), \tag{38}$$

which calculates the Fourier transform of f using the regular representation ϱ. As ϱ contains all irreps of the group, $\hat{f}$ holds the complete set of Fourier components. Conversely, contracting $\hat{f}$ with $\varrho^\dagger$ (*i.e.* factor C) performs the *inverse Fourier transform*:

$$\frac{1}{n}\operatorname{Tr}[\hat{f} C_g] = \frac{1}{n}\sum_{g' \in G} f_{g'} \operatorname{Tr}[\varrho(g')\varrho(g)^\dagger] = f_g, \tag{39}$$

where eq (2) is used. This reveals that the factor tensors generalize the discrete Fourier transform (DFT) matrix, allowing the model to map signals between the group space and its Fourier (frequency) space representations.

Through the lens of Fourier transform, we can understand how the model eq (10) processes general input vectors (f and h): it calculates their Fourier transforms ($\hat{f}, \hat{h}$), multiplies them in the Fourier domain ($\hat{f}\hat{h}$), and applies the inverse Fourier transform. Remarkably, this process is equivalent to performing group convolution ($f * h$). This is because the linearized group operation (Section 4.1) naturally entails group convolution (see Appendix J.1,J.2).

This connection reveals a profound discovery: HyperCube's ability to learn symbolic operations is fundamentally the same as learning the core structure of group convolutions. This means HyperCube can automatically discover the essential architecture needed for equivariant networks, without the need to hand-design them. This finding highlights the broad potential of HyperCube's inductive bias, extending its applicability beyond the realm of symbolic operations.

J.1 Reinterpreting HyperCube's computation

HyperCube equipped with group representation eq (10) processes general input vectors f and h as

$$\begin{aligned} f_a h_b T_{abc} &= \frac{1}{n}\sum_a \sum_b f(a)h(b)\operatorname{Tr}\left[\varrho(a)\varrho(b)\varrho(c)^\dagger\right] \\ &= \frac{1}{n}\operatorname{Tr}\left[\left(\sum_a \varrho(a)f(a)\right)\left(\sum_b \varrho(b)h(b)\right)\varrho(c)^\dagger\right] \\ &= \frac{1}{n}\operatorname{Tr}[(\hat{f}\hat{h})\varrho(c)^\dagger] = \frac{1}{n}\operatorname{Tr}[\widehat{f * h}\,\varrho(c)^\dagger] \\ &= (f * h)_c. \end{aligned} \tag{40}$$

Therefore, the model calculates the Fourier transform of the inputs ($\hat{f}$ and $\hat{h}$), multiplies them in the Fourier domain ($\hat{f}\hat{h}$), and applies the inverse Fourier transform, which is equivalent to the group convolution, as shown in Appendix I.5.

J.2 GROUP CONVOLUTION BY D

Here we show that the linearized group operation $\tilde{D}$ in Section 4.1 is equivalent to the group convolution in Appendix I.5.

Consider contracting the data tensor D with two functions $f, h \in G$, as

$$f_a h_b D_{abc} = \sum_{ab} f(a)h(b)\delta_{(a,c\circ b^{-1})} = \sum_b f(c\circ b^{-1})h(b) \equiv (f*h)(c), \tag{41}$$

which computes the group convolution between f and h, similar to eq (40). Here, we used $D_{abc} = \delta_{(a\circ b,c)} = \delta_{(a,c\circ b^{-1})}$.

K Supplementary Figures for Section 6

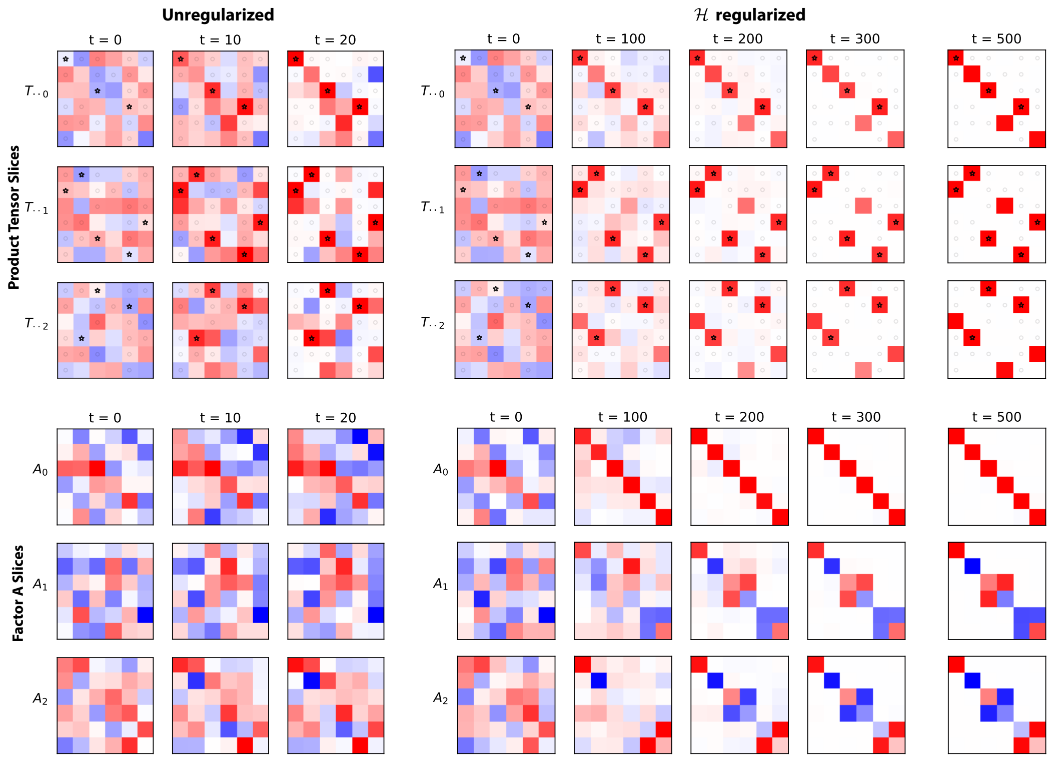

Figure 14: Visualization of the end-to-end model tensor T and the factor A over the training iteration steps on the symmetric group S_3 task in Sec 6. Only the first three slices of the tensors are shown. (Top) End-to-end model tensor T: In the un-regularized case, the model tensor quickly converges to fit the observed data tensor entries in the training dataset (marked by stars and circles), but not in the test dataset. The $\mathcal{H}$-regularized model converges to a generalizing solution around $t = 200$. It accurately recovers D when the regularization diminishes around $t = 400$ ($\epsilon \to 0$). (Bottom) Factor tensor A. The unregularized model shows minimal changes from random initial values, while $\mathcal{H}$-regularized model shows significant internal restructuring. Shown in the block-diagonalizing coordinate. See Fig 15 (Bottom). (color scheme: red=1, white=0, blue=-1.)

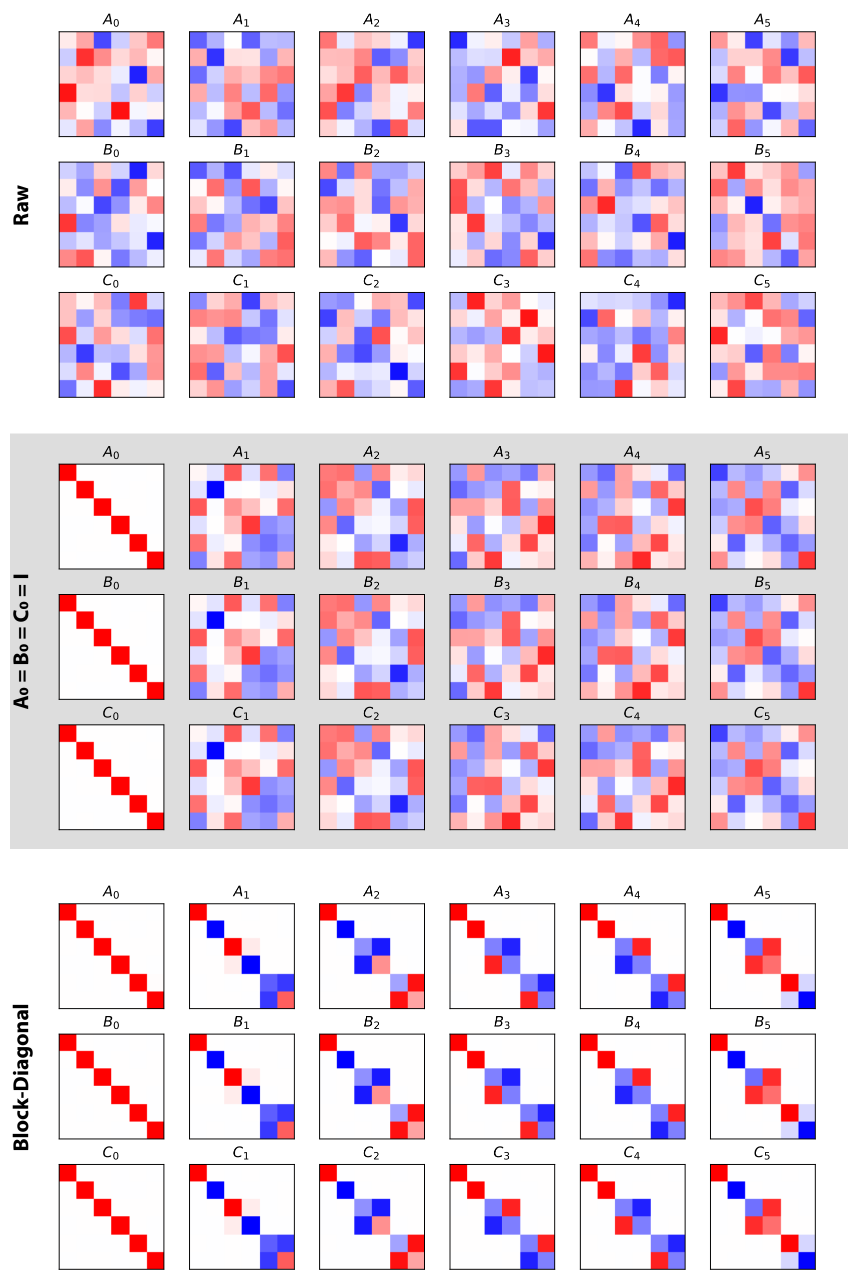

Figure 15: Learned factors of the $\mathcal{H}$ regularized model trained on the S_3 group. (Top) Raw factor weights shown in their native coordinate representation. (Middle) Unitary basis change as described in Sec 4.4 with $M_I = I$, $M_K = A_0$, $M_J = B_0^\dagger$, such that $\tilde{A}_0 = \tilde{B}_0 = \tilde{C}_0 = I$. Note that the factors share same weights (up to transpose in factor $\tilde{C}$). (Bottom) Factors represented in a block-diagonalizing basis coordinate, revealing the decomposition into direct sum of irreducible representations (irreps). (color scheme: red=1, white=0, blue=-1.)

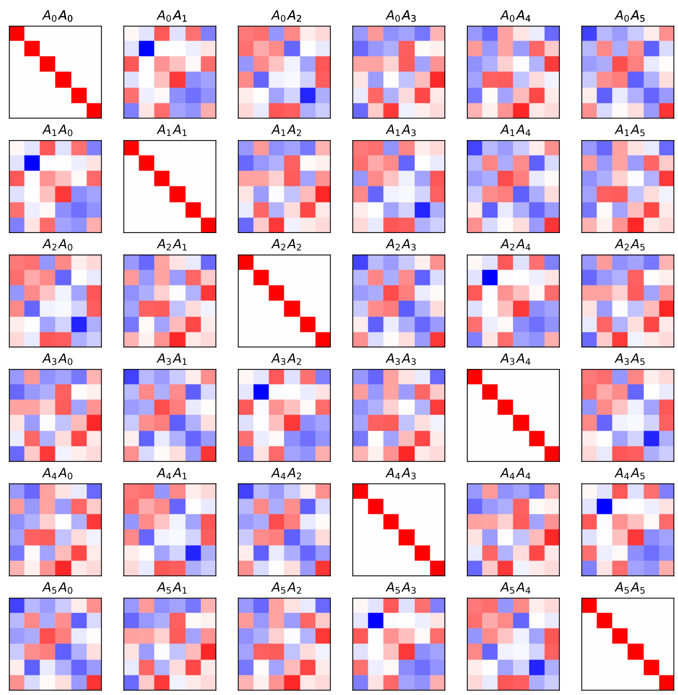

Figure 16: Multiplication table of matrix slices of factor A from the mid panel of Fig 15. Note that this table share the same structure as the Cayley table of the symmetric group S_3 in Fig 2A. (color scheme: red=1, white=0, blue=-1.)

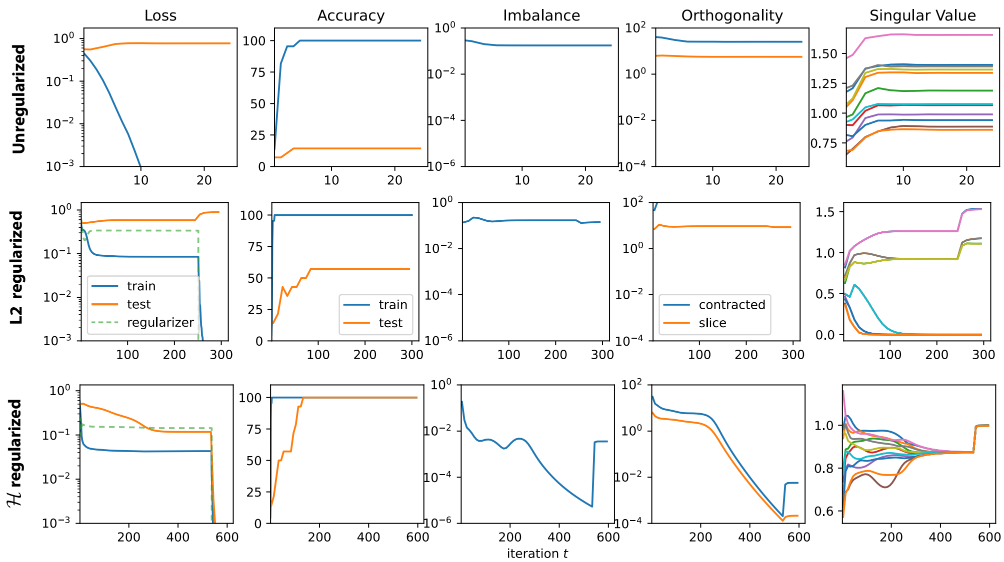

Figure 17: Optimization trajectories on the modular addition (cyclic group C_6) dataset, with 60% of the Cayley table used as train dataset (see Fig 18). (Top) Unregularized, (Middle) L_2-regularized, and (Bottom) $\mathcal{H}$-regularized training. The L_2-regularized model only achieves ∼60% test accuracy.

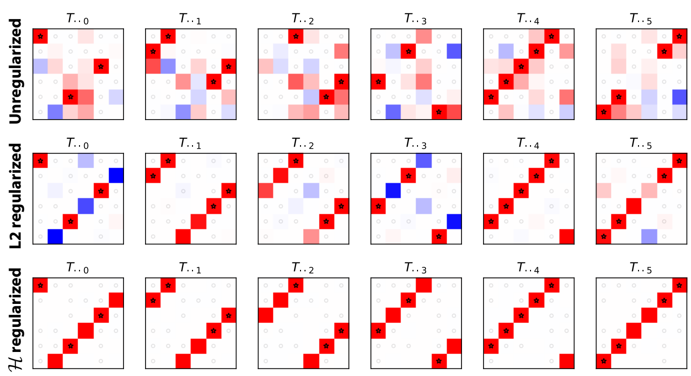

Figure 18: Visualization of end-to-end model tensor T trained on the modular addition (cyclic group C_6) under different regularization strategies (see Fig 17). The observed training data are marked by asterisks (1s) and circles (0s). Only the $\mathcal{H}$-regularized model perfectly recovers the data tensor D. (color scheme: red=1, white=0, blue=-1.)

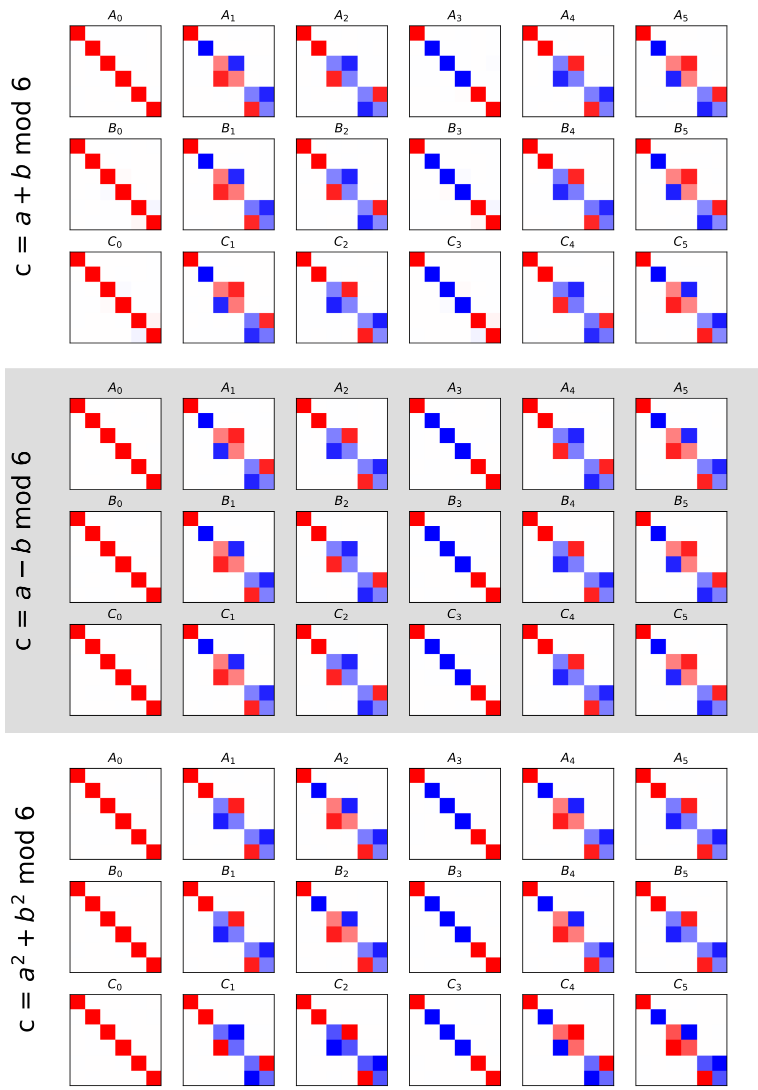

Figure 19: Visualization of factors trained on small Cayley tables from Figure 2. (Top) $c = a + b$ mod 6, satisfying $A_g = B_g = C_g^\dagger = \varrho(g)$. (Middle) $c = a - b$ mod 6, satisfying $A_g^\dagger = B_g = C_g = \varrho(g)$. (Bottom) $c = a^2 + b^2$ mod 6, which exhibits the same representation as modular addition for elements with unique inverses (e.g., $g = 0, 3$). For others, it learns *duplicate* representations reflecting the periodicity of squaring modulo 6: *e.g.*, $A_2 = A_4$ and $A_1 = A_5$, since $2^2 = 4^2$ and $1^2 = 5^2$. (color scheme: red=1, white=0, blue=-1.)