projUNN: efficient method for training deep networks with unitary matrices

Bobak T. Kiani
MIT
bkiani@mit.edu

Randall Balestriero
Meta AI, FAIR
rbalestriero@fb.com

Yann LeCun
NYU & Meta AI, FAIR
yann@fb.com

Seth Lloyd
MIT & Turing Inc.
slloyd@mit.edu

Abstract

In learning with recurrent or very deep feed-forward networks, employing unitary matrices in each layer can be very effective at maintaining long-range stability. However, restricting network parameters to be unitary typically comes at the cost of expensive parameterizations or increased training runtime. We propose instead an efficient method based on rank-k updates – or their rank-k approximation – that maintains performance at a nearly optimal training runtime. We introduce two variants of this method, named Direct (projUNN-D) and Tangent (projUNN-T) projected Unitary Neural Networks, that can parameterize full N-dimensional unitary or orthogonal matrices with a training runtime scaling as $O(kN^2)$. Our method either projects low-rank gradients onto the closest unitary matrix (projUNN-T) or transports unitary matrices in the direction of the low-rank gradient (projUNN-D). Even in the fastest setting ($k = 1$), projUNN is able to train a model's unitary parameters to reach comparable performances against baseline implementations. In recurrent neural network settings, projUNN closely matches or exceeds benchmarked results from prior unitary neural networks. Finally, we preliminarily explore projUNN in training orthogonal convolutional neural networks, which are currently unable to outperform state of the art models but can potentially enhance stability and robustness at large depth.

1 Introduction

Learning in neural networks can often be unstable when networks are very deep or inputs are long sequences of data [5, 83]. For example, vanilla recurrent neural networks (RNNs) have recurrent states that are evolved via repeated application of a linear transformation followed by a pointwise nonlinearity, which can become unstable when eigenvalues of the linear transformation are not of magnitude one. Unitary matrices, which have eigenvalues of magnitude one, can naturally avoid this issue and have been used as a means to overcome these so-called vanishing and exploding gradients [5, 44]. More recently, unitary convolutional layers have been similarly constructed to help build more stable deep networks that are norm-preserving in their transformations [58, 72].

In the RNN setting, prior algorithms to apply $n \times n$ unitary matrices in RNNs have parameterized matrices into layers of unitary or orthogonal transformations or parameterized the Lie algebra of the unitary or orthogonal group (see Table 1). In the layer-wise setting, unitarity is enforced for all values of parameters, but many layers are required to form a composition that can recreate any desired unitary, *i.e.,* fully parameterizing an $n \times n$ unitary requires $O(n)$ layers. By parameterizing the Lie algebra [56, 41], algorithms perform better on common benchmarks but have the drawback

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

Table 1: When training RNNs on inputs with sequence length T, PROJUNN achieves nearly optimal runtime complexity while maintaining full parameterization of the unitary manifold.

Model	Complexity of gradient step	Layers to fully parameterize[a]	Method of parameterization
EURNN (tunable, n layers) [44]	$O(Tn^2)$	$O(n)$	Sequence of rotations
oRNN (n layers) [66]	$O(Tn^2)$	$O(n)$	Sequence of householder reflections
full-capacity URNN [82]	$O(Tn^2 + n^3)^b$	1	Parameterized matrix entries
expRNN [56]	$O(Tn^2 + n^3)^b$	1	Parameterized matrix in Lie algebra
PROJUNN (our method)	$O(Tn^2 + kn^2)^c$	1	Parameterized matrix entries

[a] layers needed to parameterize the full unitary space, [b] approximations exist which may reduce runtimes though these approximations are not implemented here and can significantly bias the gradient [56], [c] runtime shown for typical setting when $k \ll n$ where k is the rank of gradient updates

that performing gradient optimization on an $n \times n$ unitary requires $O(n^3)$ operations generically per step. Though not an issue with the small to medium sized models used today, this $O(n^3)$ is still $O(n)$ slower than standard methods of forward- and back-propogation in RNNs.

Motivated by the feature that gradients in neural networks are typically approximately low rank, we show that gradient updates to unitary/orthogonal matrices can be efficiently performed in low rank settings. We propose a new model called PROJUNN where matrices are first updated via gradient based optimization and then projected back onto the closest unitary (PROJUNN-D) or transported in the direction of the gradient (PROJUNN-T). PROJUNN has near-optimal runtime complexity unlike other existing algorithms for unitary RNNs (Table 1) and is especially effective even in the most extreme case where gradients are approximated by rank one matrices. In RNN learning tasks, PROJUNN matches or exceeds benchmarks of state-of-the-art unitary neural network algorithms.

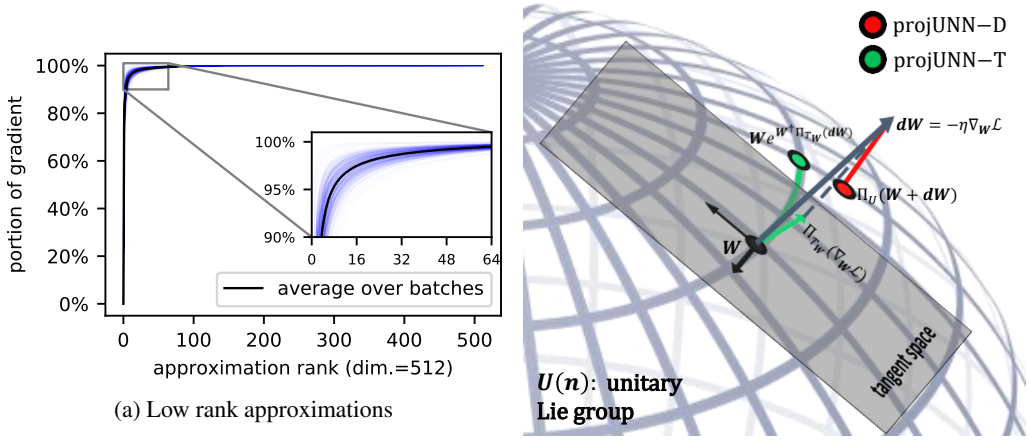
Though we present our model first in the RNN setting, we show that there is a direct extension of PROJUNN to the case of orthogonal/unitary convolution which we explore further. Here, we perform unitary/orthogonal convolution in the Fourier domain as inspired by [76]. Our algorithm runs efficiently in the convolutional setting especially for filters of large size and many channels (see Appendix F for more details).

2 Related works

Maintaining stability in neural networks via orthogonal or unitary matrices has a rich history of study in machine learning, both from an applied and theoretical perspective. Here, we briefly mention the most related works and algorithms we use in comparison to our PROJUNN. For a more holistic review of prior work in unitary neural networks and other related topics, please see Appendix B.

Unitary neural networks were first designed to address the issue of vanishing and exploding gradients in RNNs while learning information in very long sequences of data more efficiently than existing parameterizations such as the long-short term memory unit (LSTM) [38]. Early algorithms [5, 66] maintained unitarity by constructing a series of parameterized unitary transformations. Perhaps the most effective of these methods is the efficient unitary recurrent neural network (EUNN) [44] which parameterized unitary matrices by composing layers of Givens rotations, Fourier transforms, and other unitary transformations. The unitary RNN (uRNN) of [82] and the Cayley parameterization (scoRNN) of [35] parameterized the full unitary space and maintained unitarity by performing a Cayley transformation. Later, [56] introduced the exponential RNN (expRNN) which parameterized unitary matrices in the Lie algebra of the orthogonal/unitary group. Though the uRNN, scoRNN, and expRNN perform well on benchmarks, their algorithms require matrix inversion or SVD steps which are time-consuming in high dimensions.

For convolutional neural networks, [72] showed how to efficiently calculate the singular values of a linear convolution and proposed an algorithm for projecting convolutions onto an operator-norm ball which relied on a series of costly projection steps. [58] introduced a block convolutional orthogonal parameterization (BCOP) which was faster and more efficient than the methods in [72], but required extra parameters in its parameterization and only parameterized a subset of the space of orthogonal convolutions. Most recently, [73] implemented orthogonal convolutions by parameterizing the Lie algebra of the orthogonal group via their skew orthogonal convolution (SOC)



(a) Low rank approximations

(b) Illustration of algorithm

Figure 1: (a) Low rank approximations capture most of the Frobenius norm of the gradient of a 512×512 matrix in the convolution filter (512 channels) of the last residual block of Resnet-9. Blue lines plot gradients of a single batch during training of our PROJUNN algorithm on CIFAR10 over a single epoch (see Appendix E for details and equivalent plot for RNN architecture). (b) Illustration of a single gradient update via gradient descent with learning rate η. PROJUNN-D (pictured in red) directly projects the gradient update back onto the unitary/orthogonal manifold. PROJUNN-T (pictured in green) first projects onto the tangent space (Lie algebra) and then performs a rotation in that direction via the exponential map.

algorithm which approximates orthogonal convolutions especially well for small filter sizes. Finally, [76] performs convolutions in the Fourier domain via application of the Cayley transform. Our orthogonal/unitary convolutional parameterization is inspired by their approach and improves their runtime for convolutions over many channels.

3 Notation and background

Vectors and matrices are denoted with bold lower-case and upper-case script, $\boldsymbol{v}$ and $\boldsymbol{V}$, respectively. Scalars are denoted by regular script e and tensors are denoted by bold text $\mathbf{T}$. The complex conjugate of a complex-valued input $\cdot$ is denoted by $\cdot^*$ (ignored when real-valued). The transpose of a matrix $\boldsymbol{M}$ is denoted by $\boldsymbol{M}^\intercal$ and the conjugate transpose of a matrix is denoted by $\boldsymbol{M}^\dagger$. We denote the Frobenius norm of a matrix by $\|\cdot\|_F$ and the spectral norm of a matrix by $\|\cdot\|_2$.

Here, we provide a brief overview of the unitary/orthogonal groups and refer readers to Appendix A for a more detailed mathematical background. The set of $n \times n$ orthogonal $O(n)$ and unitary $U(n)$ matrices are both Lie groups defined as

$$O(n) = \left\{\boldsymbol{M} \in \mathbb{R}^{n\times n} | \boldsymbol{M}\boldsymbol{M}^\intercal = \boldsymbol{I}\right\}, \qquad U(n) = \left\{\boldsymbol{M} \in \mathbb{C}^{n\times n} | \boldsymbol{M}\boldsymbol{M}^\dagger = \boldsymbol{I}\right\}. \tag{1}$$

Constraining matrices in $O(n)$ and $U(n)$ to have determinant equal to one constructs the special orthogonal $SO(n)$ and unitary $SU(n)$ groups respectively. The Lie algebra or tangent space of the identity of $O(n)$ and $U(n)$ are the set of skew symmetric $\mathfrak{o}(n)$ and skew Hermitian $\mathfrak{u}(n)$ matrices,

$$\mathfrak{o}(n) = \left\{\boldsymbol{A} \in \mathbb{R}^{n\times n} : \boldsymbol{A} + \boldsymbol{A}^\intercal = 0\right\}, \qquad \mathfrak{u}(n) = \left\{\boldsymbol{A} \in \mathbb{C}^{n\times n} : \boldsymbol{A} + \boldsymbol{A}^\dagger = 0\right\}. \tag{2}$$

The matrix exponential $\exp(\cdot)$ is a map from the Lie algebra to the associated Lie group. The map is surjective if the Lie group is compact and connected – a property which holds for the unitary and special orthogonal groups but not the orthogonal group.

4 Projected unitary networks

Our PROJUNN algorithm is motivated by the simple observation that most of the "information" of a typical gradient in a deep learning task is captured in a low rank subspace of the complete gradient.

Figure 15 illustrates this feature when training our PROJUNN convolutional network on CIFAR10. We include further analysis and justification of this low rank behavior in Appendix E. As we will show, we can perform updates on the low rank subspace of the gradient efficiently by approximating the gradient with a low rank matrix and performing projections of parameters onto that low rank subspace. Our experiments show that this methodology, even with rank one approximations, is effective at learning and empirically introduces a form of "beneficial" stochasticity during gradient descent.

Based on how the projection is performed, our PROJUNN algorithm takes two forms illustrated in Figure 1b. The directly projected unitary neural network (PROJUNN-D) projects an update onto the closest unitary/orthogonal matrix in Frobenius norm. The tangent projected unitary neural network (PROJUNN-T) projects gradients onto the tangent space and transports parameters in that direction.

4.1 PROJUNN-D

PROJUNN-D takes advantage of the fact that the polar transformation returns the closest unitary or orthogonal matrix in the Frobenius norm to a given matrix (not necessarily unitary or orthogonal):

Lemma 4.1 (Projection onto unitary manifold [46])**.** *Given a matrix $\boldsymbol{A} \in \mathbb{C}^{n\times n}$:*

$$\Pi_U(\boldsymbol{A}) = \underset{\boldsymbol{U}\in\mathcal{U}(n)}{\arg\min} \|\boldsymbol{A}-\boldsymbol{U}\|_F^2 = \boldsymbol{A}(\boldsymbol{A}^\dagger\boldsymbol{A})^{-\frac{1}{2}}, \tag{3}$$

where $\mathcal{U}(n)$ indicates the set of $n\times n$ unitary matrices.

Note, that if the matrix $\boldsymbol{A}$ is real, then the projection above will be onto an orthogonal matrix. Given Lemma 4.1, PROJUNN-D performs optimization in two steps, which are illustrated in Figure 1b. First, matrix entries are updated via a standard learning step as in gradient descent, constructing a new matrix that is generally no longer unitary. In the second step, PROJUNN-D returns the unitary or orthogonal matrix closest in the Frobenius norm to the inputted matrix using Lemma 4.1. At first sight, the second step would require $O(n^3)$ time to perform, but we can take advantage of the fact that gradient updates are typically approximately low rank (see Appendix E). Efficient low rank approximations can be obtained using sampling methods detailed in Section 4.3. With this in mind, we show that rank k updates can be performed in $O(kn^2)$ time when $k \ll n$.

Theorem 4.2 (Low rank unitary projection)**.** *Let $\boldsymbol{U}$ be an $n\times n$ orthogonal/unitary matrix perturbed by $\boldsymbol{G}_k$, a rank k matrix. Then the projection onto the closest orthogonal/unitary matrix defined below can be performed in $O(k(n^2+nk+k^2))$ steps.*

$$\boldsymbol{U}+\boldsymbol{G}_k \to \underset{\boldsymbol{V}\in\mathcal{U}}{\arg\min} \|\boldsymbol{U}+\boldsymbol{G}_k-\boldsymbol{V}\|_F^2. \tag{4}$$

To achieve this runtime, we perform updates completely in an $O(k)$ subspace of the full vector space. The operation $(\boldsymbol{U}+\boldsymbol{G}_k)[(\boldsymbol{U}+\boldsymbol{G}_k)^\dagger(\boldsymbol{U}+\boldsymbol{G}_k)]^{-1/2}$ can be decomposed into a series of $O(k)$ matrix-vector operations and an eigendecomposition of a $2k\times 2k$ sub-matrix. The complete proof and details are deferred to Appendix C. One limitation of the above is that the eigendecomposition and inversion of a low rank matrix can cause numerical instability after many update steps. We discuss this further in Appendix G.3 where we also provide options to alleviate this instability. PROJUNN-T, which we discuss next, does not require matrix inversion and is thus empirically more stable.

4.2 PROJUNN-T

PROJUNN-T maintains unitarity of matrices by orthogonally projecting gradient updates onto the tangent space and then performing a rotation in the direction of the projection (*i.e.,* along the geodesic). As in PROJUNN-D, there is a closed form for the orthogonal projection:

Lemma 4.3 (Tangent space projection [82])**.** *Given the tangent space $T_{\boldsymbol{U}}U(n)$ of an orthogonal/unitary matrix $\boldsymbol{U}$, the orthogonal projection $\Pi_{T_{\boldsymbol{U}}}$ with respect to the canonical metric $\langle \boldsymbol{X},\boldsymbol{Y}\rangle = \mathrm{Re}\left(\mathrm{Tr}[\boldsymbol{X}^\dagger\boldsymbol{Y}]\right)$ is*

$$\Pi_{T_{\boldsymbol{U}}}(\boldsymbol{X}) = \frac{1}{2}\left(\boldsymbol{X}-\boldsymbol{U}\boldsymbol{X}^\dagger\boldsymbol{U}\right). \tag{5}$$

Similar to Lemma 4.1, this projection also returns the closest matrix in Frobenius norm to $\boldsymbol{X}$ in the tangent space,

$$\min_{\boldsymbol{Y}\in T_{\boldsymbol{U}}U(n)} \|\boldsymbol{Y}-\boldsymbol{X}\|_F = \Pi_{T_{\boldsymbol{U}}}(\boldsymbol{X}). \tag{6}$$

Similar to PROJUNN-D, PROJUNN-T performs learning in two steps. First, a gradient update $\boldsymbol{G}$ is projected onto the tangent space using Lemma 4.3. Then, the orthogonal/unitary matrix is transported or rotated in the direction of the projection by application of the exponential map via the update rule [56, 82],

$$\boldsymbol{U} \to \boldsymbol{U} \exp\left[-\eta \boldsymbol{U}^{\dagger} \Pi_{T_{\boldsymbol{U}}}(\boldsymbol{G})\right], \tag{7}$$

where η denotes the learning rate. This update rule is an example of Riemannian gradient descent where we use the exponential map to transport gradient updates along the unitary/orthogonal manifold [15]. Here, we transport the matrix $\boldsymbol{U}$ along the geodesic in the direction of $\Pi_{T_{\boldsymbol{U}}}(\boldsymbol{G})$. This can be related to the update of PROJUNN-D which is an example of a retraction or an approximation to the exponential map of PROJUNN-T (see Appendix C.3).

The update rule above requires matrix exponentiation and multiplication, both costly steps which can be sped up when $\boldsymbol{G}$ is a low rank matrix. Namely, to perform a rank k gradient update, we obtain an equivalent runtime scaling of $O(kn^2)$ for the PROJUNN-D when $k \ll n$.

Theorem 4.4 (Low rank tangent transport)**.** *Let $\boldsymbol{U}$ be an $n \times n$ orthogonal/unitary matrix perturbed by $\boldsymbol{G}_k$, a rank k matrix. Then projecting $\boldsymbol{G}_k$ onto the tangent space and performing a rotation in that direction as defined in Equation* (7) *can be performed in $O(k(n^2 + nk + k^2))$ steps.*

As with the PROJUNN-D, we achieve this runtime by performing the update above completely in an $O(k)$ subspace of the full vector space. The update via the exponential map can similarly be decomposed into a series of $O(k)$ matrix-vector operations and an eigendecomposition of a $2k \times 2k$ sub-matrix. Proper manipulations of the eigenvalues of the sub-matrix implement updates via the exponential map. The complete proof and details are deferred to Appendix C.

4.3 Sampling methods

Commonly, gradients can have large rank but have still have many small singular values (*e.g.,* see Figure 1a). Here, a matrix $\boldsymbol{A}$ is deemed approximately low rank (see more details in Appendix E), and one can obtain a rank k approximation $\boldsymbol{A}_k$ of $\boldsymbol{A}$ by sampling from rows and columns of $\boldsymbol{A}$. We use two sampling algorithms. The **LSI sampling** algorithm [68] obtains a rank k approximation to an $n \times n$ matrix $\boldsymbol{A}$ in time $O(kn^2 \log n)$. The algorithm projects the matrix $\boldsymbol{A}$ onto a random orthogonal subspace and then applies SVD based methods to the projected matrix to obtain the low rank approximation to that matrix. This algorithm features low approximation errors even for small k and is used extensively in our implementation. The **column sampling** (linear time SVD) algorithm [23] samples from the columns of an $n \times n$ matrix $\boldsymbol{A}$ to obtain a rank k approximation in $O(c^2 n + c^3)$ time, where c is a hyperparameter indicating the number of columns sampled. Typically, c is chosen as a multiple of k so the runtime is $O(k^2 n + k^3)$. In implementing this algorithm, we calculate the right singular vectors via matrix multiplication of the left singular vectors so the total runtime is $O(kn^2 + k^2 n + k^3)$.

We note that the two procedures described above, though sufficient for our purposes, can be further optimized in their asymptotic runtime. For sake of completeness, we discuss two of these other sampling algorithms in Appendix E.

4.4 Extension to unitary or orthogonal convolution

Unitary/orthogonal convolutions are linear convolution operations that also preserve the 2-norm (isometric). Restricting convolutions to be unitary/orthogonal typically results in a drop in performance on standard imaging tasks when used in isolation, but prior work has explored unitary/orthogonal convolutions to potentially improve algorithmic stability and robustness (see Appendix B.1 for more background) [58, 76]. We describe here how PROJUNN can be used to implement unitary/orthogonal convolutions in potentially a more efficient manner.

Given input tensor $\mathbf{X} \in \mathbb{C}^{M \times N \times C}$ where C is the number of channels of an $M \times N$ input, linear convolution (or technically cross-correlation) with a filter $\mathbf{W} \in \mathbb{C}^{M \times N \times C \times C}$ is defined as

$$[\mathrm{conv}_{\mathbf{W}}(\mathbf{X})]_{p,q,d} = \sum_{c=1}^{C} \sum_{m=1}^{M} \sum_{n=1}^{N} \mathbf{W}_{m,n,d,c} \mathbf{X}_{p+m,q+n,c}, \tag{8}$$

where the indexing above is assumed to be cyclic (taken modulus the corresponding dimension) [55, 28]. Orthogonal/unitary convolutions form a subset of filters that preserve norms, *i.e.,* filters $\mathbf{W}$

such that $\|\operatorname{conv}_{\mathbf{W}}(\mathbf{X})\| = \|\mathbf{X}\|$. Equivalently, $\operatorname{conv}_{\mathbf{W}}(\cdot)$ is orthogonal/unitary if the Jacobian of the transformation is also orthogonal/unitary. To maintain unitarity/orthogonality, we set the dimensions of the filter $\mathbf{W}$ above such that it returns an output $\mathbf{Y}$ of the same dimension as the input $\mathbf{X}$. One can also perform semi-orthogonal or semi-unitary convolution by appropriately zero-padding an input or truncating from dimensions in the output.

Standard convolutional filters are typically supported over a sparse set of local elements, but performing orthogonal/unitary convolution generally requires implementing convolutions with filters supported over all elements resulting in slower runtimes. One can locally parameterize convolutional filters in the Lie algebra of the orthogonal/unitary group; nevertheless the exponential map into the Lie group expands the support of the filter:

$$\exp[\operatorname{conv}_{\mathbf{L}}](\mathbf{X}) = \mathbf{X} + \mathbf{L} * \mathbf{X} + \frac{1}{2}\mathbf{L} *^2 \mathbf{X} + \frac{1}{6}\mathbf{L} *^3 \mathbf{X} + \cdots \tag{9}$$

Thus, enforcing unitarity in convolutions generally requires additional overhead over the traditional setting of locally supported filters, but by performing convolution in the Fourier domain, runtimes for full-width filters can be optimally improved to $O(N^2 C \log(N) + N^2 C^2)$ [64]:

$$[\operatorname{FFT} \operatorname{conv}_{\mathbf{W}}(\mathbf{X})]_{\widehat{r},\widehat{s},:} = \widehat{\mathbf{W}}^*_{\widehat{r},\widehat{s},:,:} \; [\operatorname{FFT} \mathbf{X}]_{\widehat{r},\widehat{s},:} \;, \tag{10}$$

where $\widehat{\mathbf{W}}_{i,j,:,:}$ is the value of the $\widehat{r}$ and $\widehat{s}$ frequency of $\mathbf{W}$ across all channels in the Fourier domain and FFT is the 2-dimensional fast Fourier transformation.

Our method is inspired by that of [76] which transformed $\mathbf{W}$ into Fourier space and performed a Cayley transformation (approximation to the exponential map into the Lie group) over the matrices indexed by $\widehat{\mathbf{W}}_{\widehat{r},\widehat{s},:,:}$ which requires $O(N^2C^2\log(N) + N^2C^3)$ operations. For our algorithm, we parameterize $\mathbf{W}$ in the Fourier domain and only manipulate $\widehat{\mathbf{W}}$ (see Appendix B.1 for a depiction of our parameterization). By parameterizing $\widehat{\mathbf{W}}$ directly and performing rank k updates using our PROJUNN, this runtime can be improved to $O(N^2C\log(N) + kN^2C^2)$ which is optimal when $k \ll N$. Our procedure for performing unitary/orthogonal convolution on an input $\mathbf{X}$ with filter $\mathbf{W}$ essentially follows the steps in Equation (10): perform an FFT on $\mathbf{X}$, block-multiply this by $\widehat{\mathbf{W}}$, and perform an inverse FFT on the output to obtain the final result.

Limitations Unitary/orthogonal convolutions are implemented in a cyclic fashion (*i.e.,* indices are taken modulus the dimension) which is not the standard approach but has been used before to accelerate convolutional operations [64]. Additionally, we parameterize convolution filters to have support over all possible elements (full-width), which can be expensive in memory. One can restrict the convolution to local terms in the Lie algebra, but this would not improve runtime as our algorithm runs in the Fourier space. To target local terms in a convolution, we instead propose for future work to implement a regularizer which has a specified support and penalizes the norm of the filter outside that support. Finally, the space of orthogonal convolutions has multiple disconnected components, which can present challenges for gradient based learning [58]. However, we can avoid this drawback by implementing PROJUNN using fully supported filters in the space of unitary convolutions which is connected (proof deferred to Appendix C.4).

Theorem 4.5 (Unitary convolutional manifold is connected)**.** *The space of unitary convolutions with filters of full support has a single connected component.*

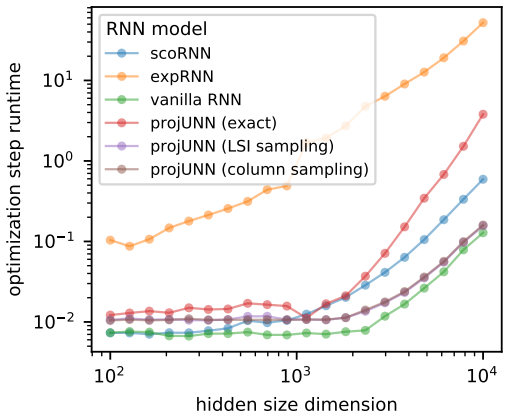
4.5 Pseudocode for performing projUNN updates

Pseudocode for performing an update step on a unitary or orthogonal matrix $\boldsymbol{U}$ with a gradient update of $\Delta\boldsymbol{U}$ is shown in Algorithm 1. In convolutional settings, the steps in Algorithm 1 are applied across blocks of the convolution in Fourier space which can be performed in parallel. As a cautionary note, especially in the last step of Algorithm 1, where there is a composition of multiple matrix-vector multiplications, the order of these multiplications must be chosen to only perform matrix-vector operations to ensure optimal runtime. In other words, two $N \times N$ matrices should never be multiplied by each other at any point in this algorithm.

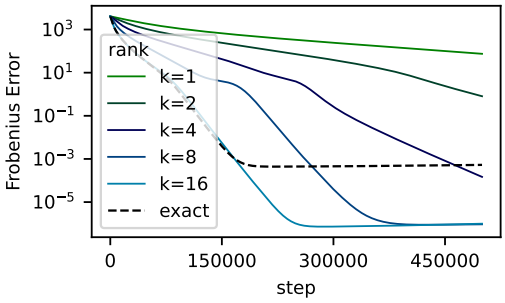
Algorithm 1 PROJUNN update step

Require: unitary matrix $\boldsymbol{U} \in \mathbb{C}^{N\times N}$ or orthogonal matrix $\boldsymbol{U} \in \mathbb{R}^{N\times N}$
Require: gradient update $\Delta\boldsymbol{U} \in \mathbb{C}^{N\times N}$ or $\Delta\boldsymbol{U} \in \mathbb{R}^{N\times N}$
Require: hyperparameter k corresponding to rank of approximation

1: Obtain rank k approximation to $\Delta\boldsymbol{U}$ with output $\sum_{i=1}^{k} \boldsymbol{a}_i \boldsymbol{b}_i^\dagger \approx \Delta\boldsymbol{U}$ (see Section 4.3)
2: Follow steps in Theorem 4.2 (PROJUNN-D) or Theorem 4.4 (PROJUNN-T) in Appendix C:
3: Perform Gram-Schmidt (via QR decomposition) on concatenation of vectors $\boldsymbol{U}^\dagger \boldsymbol{a}_i$ and $\boldsymbol{b}_i$ for all $i \in [k]$: output $\boldsymbol{Q} \in \mathbb{C}^{N\times k}$ as semi-orthogonal matrix containing basis after Gram-Schmidt
4: Form matrix $\boldsymbol{K} \in \mathbb{C}^{2k\times 2k}$ below:
PROJUNN-D: $\boldsymbol{K} = \sum_{i=1}^{k} \boldsymbol{Q}^\dagger \boldsymbol{U}^\dagger \boldsymbol{a}_i \boldsymbol{b}_i^\dagger \boldsymbol{Q} + \boldsymbol{Q}^\dagger \boldsymbol{b}_i \boldsymbol{a}_i^\dagger \boldsymbol{U} \boldsymbol{Q} + \sum_{i=1}^{k}\sum_{j=1}^{k} (\boldsymbol{a}_i^\dagger \boldsymbol{a}_j) \boldsymbol{Q}^\dagger \boldsymbol{b}_i \boldsymbol{b}_j^\dagger \boldsymbol{Q}$
see Equation (C.1) *to Equation* (C.6)
PROJUNN-T: $\boldsymbol{K} = \frac{1}{2}\left[\sum_{i=1}^{k} \boldsymbol{Q}^\dagger \boldsymbol{U}^\dagger \boldsymbol{a}_i \boldsymbol{b}_i^\dagger \boldsymbol{Q} - \boldsymbol{Q}^\dagger \boldsymbol{b}_i \boldsymbol{a}_i^\dagger \boldsymbol{U} \boldsymbol{Q}\right]$
see Equation (C.12) *to Equation* (C.15)
5: Find eigenvalues $s_1, \ldots, s_{2k}$ and eigenvectors $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_{2k}$ of $\boldsymbol{K}$
6: Perform update step by applying eigenvalue function:
PROJUNN-D: $\boldsymbol{U} \leftarrow (\boldsymbol{U} + \sum_{i=1}^{k} \boldsymbol{a}_i \boldsymbol{b}_i^\dagger)\left[\boldsymbol{I} + \sum_{j=1}^{2k}\left((s_j + 1 + \epsilon)^{-\frac{1}{2}} - 1\right)\boldsymbol{u}_j \boldsymbol{u}_j^\dagger\right]$
see Equation (C.8) *and Equation* (C.9), ϵ added for stability when $s_j \approx -1$ (we set $\epsilon = 10^{-8}$)
PROJUNN-T: $\boldsymbol{U} \leftarrow \boldsymbol{U}\left[\boldsymbol{I} + \sum_{j=1}^{2k}(\exp(-\eta s_j) - 1)\boldsymbol{u}_j \boldsymbol{u}_j^\dagger\right]$ where η is the learning rate
see Equation (C.17) *and Equation* (C.18)



(a) Runtime comparison



(b) Learning random unitary

Figure 2: (a) Runtime of PROJUNN (with low rank approximation) scales asymptotically at same rate of a vanilla RNN and much faster than other unitary RNN models or the exact version of PROJUNN (not using low rank approximation). Practical runtime improvements are achieved when the hidden dimension is large (see Appendix F for details). (b) PROJUNN-T can learn a random target unitary matrix using SGD. For a fixed learning rate, the loss decays at a rate proportional to the approximation rank k up to $k = 16$ where the approximation captures the full batch size (see exact PROJUNN which employs no approximation). The y-axis plots Frobenius error $\|\boldsymbol{U} - \boldsymbol{U}_{tar}\|_F^2$.

4.6 Runtime comparisons

PROJUNN has a nearly optimal asymptotic runtime scaling which offers practical benefits in high dimensions. In the RNN setting, Figure 2a shows that the low rank version of PROJUNN has a runtime that scales at the same rate as that of a vanilla RNN albeit with increased overhead. Updating the unitary matrix of PROJUNN takes $O(kn^2)$ time for performing updates of rank $k \ll n$, only a factor k more than a vanilla RNN which performs updates in $O(n^2)$ time. Note, that exact (full rank) updates to the $n \times n$ unitary matrices of a PROJUNN take roughly $O(n^3)$ time corresponding to the runtime of an SVD and equivalent to the runtime of expRNN and scoRNN [56, 35].

In the convolutional setting, PROJUNN offers the most benefit when there are many channels, filters with large support (very wide), or a need for exact unitary/orthogonal operations (in contrast with an

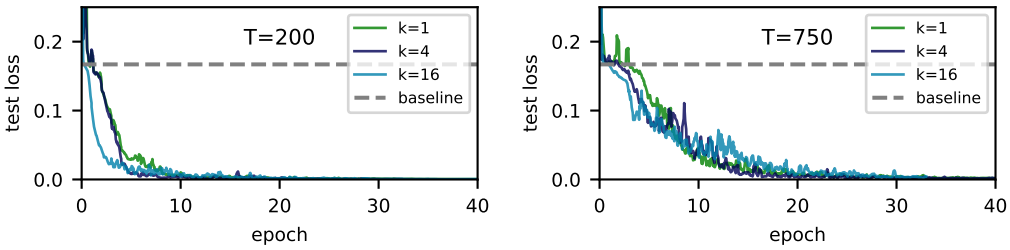


Figure 3: PROJUNN-T learns the adding task with $T = 200$ and $T = 750$. Test error is smoothed by taking the running average of 5 sequential points. See Appendix D.2 for more details.

approximate method like [73]). Given an $N \times N$ input with C channels, a forward and backward pass of PROJUNN runs in time $O(N^2 C \log(N) + kN^2C^2)$ when performing rank k updates. This is a factor of C faster than the Cayley implementation [76] which runs in time $O(N^2C^2 \log(N) + N^2C^3)$. For a more complete analysis of the asymptotic and empirical runtimes of various models including many not listed here, please see Appendix F.

5 Experiments

We propose in this section a variety of benchmarked experiments to validate the efficiency and performance of the proposed PROJUNN method focusing mostly on RNN tasks.[1] We include further details of the experiments in Appendix D including a preliminary empirical analysis of PROJUNN in convolutional tasks.

Toy model: learning random unitary To study the learning trajectories of PROJUNN, we consider a simple toy model aimed at learning a target random unitary. More specifically, we parameterize a large unitary matrix $\boldsymbol{U} \in \mathbb{C}^{2048\times 2048}$ to learn a Haar random target unitary $\boldsymbol{U}_{tar} \in \mathbb{C}^{2048\times 2048}$ given a dataset $\{\boldsymbol{x}_i, \boldsymbol{y}_i = \boldsymbol{U}_{tar}\boldsymbol{x}_i\}_{i=1}^{4096}$ of size 4096 where $\boldsymbol{x}_i \in \mathbb{C}^{2048}$ has entries drawn i.i.d. random normal. $\boldsymbol{U}$ is initialized as a random unitary matrix, and each step, we perform vanilla gradient descent over a batch of 16 training points using mean-squared error loss $\ell(\boldsymbol{x}_i, \boldsymbol{y}_i) = \|\boldsymbol{U}\boldsymbol{x}_i - \boldsymbol{y}_i\|_2^2$. Approximations of rank k to the gradient are obtained using the column sampling algorithm.

Figure 2b, which plots the Frobenius error $\|\boldsymbol{U} - \boldsymbol{U}_{tar}\|_F^2$, shows that PROJUNN-T equipped with the column sampling approximator is able to learn the random target unitary even when $k = 1$ (see Appendix D.1 for plots with PROJUNN-D). Furthermore, for a fixed learning rate, learning requires fewer steps with larger k up to $k = 16$, the maximum rank of the gradient (note that $\nabla_{\boldsymbol{U}}\ell(\boldsymbol{x}_i, \boldsymbol{y}_i)$ is rank 1). Therefore, approximating the gradient via low rank approximations can significantly speed up learning in this task (see Appendix D.1 for further details).

Adding task In the adding task, an RNN must learn to add two numbers in a long sequence. We consider a variant of the adding task studied in [5], where the input consists of two data sequences of length T. The first is a list of T numbers sampled uniformly from $[0, 1]$, and the second is a list of binary digits set to zero except for two locations (those which must be summed) set to one located uniformly at random within the intervals $[1, T/2)$ and $[T/2, T)$ respectively.

Consistent with [35], we train our PROJUNN-T using an RNN with hidden dimension of 170 and the RMSprop optimizer to reduce the mean-squared error of the output with respect to the target. Naively predicting the average value of one for a random input achieves mean-squared error of approximately 0.167. As shown in Figure 3, PROJUNN-T is able to learn the target function even with rank $k = 1$ approximations. Surprisingly, for a fixed learning rate and scheduler, convergence to the true solution is almost equally fast for $k = 1$, $k = 4$, and $k = 16$. Further details are provided in Appendix D.2.

Copy memory task The copying memory task is a common benchmark for RNNs [38, 5, 36], where the aim is to memorize input data by ignoring a long sequence of void tokens. Given an

[1] code repository: https://github.com/facebookresearch/projUNN

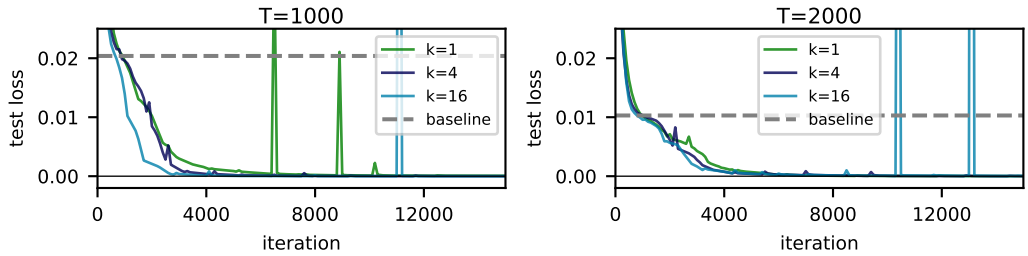


Figure 4: PROJUNN-T equipped with the column sampling approximation learns the copy task with $T = 1000$ and $T = 2000$ even with rank one approximations.

Table 2: Result of gradient descent optimization using the RMSprop optimizer on a single layer RNN for the permutedMNIST classification task. Each result is averaged over 3 runs, the same cross validation is done for all settings and includes the learning rate and its schedule. Training occurs for 200 epochs, and 10% of the training set (same for all models) is set apart as validation set. The training curves are provided in Figure 13.

								PROJUNN-D					PROJUNN-T				
Width	RGD	LSTM	ScoRNN	ExpRNN	DT_∞	DT_{100}	DT_1	k=1	2	4	8	16	k=1	2	4	8	16
116	92.5	91.8	-	-	-	-	-	92.8	93.0	93.0	92.9	**93.2**	92.5	92.6	92.5	93.0	92.8
170	-	92.0	94.8	94.9	95.0	95.1	**95.2**	94.3	94.3	94.4	94.7	94.3	94.4	94.3	94.4	94.1	94.3
360	93.9	92.9	96.2	96.2	**96.5**	96.4	96.3	96.4	96.4	96.3	96.3	**96.5**	96.3	96.3	96.4	96.2	96.4
512	94.7	92.0	96.6	96.6	96.8	96.7	96.7	**97.0**	**97.0**	96.8	96.9	**97.0**	96.7	96.7	96.8	96.8	96.7

alphabet of $n + 2$ symbols $\{a_i\}_{i=1}^{n+2}$, n of which represent data (sequence of letters $A, B, \dots$) and additional *void* (-) and *start recall* (:) tokens, the RNN must output the first K input tokens as the last K output tokens and *void* otherwise. An example input/output for $M = 6$ with $n = 4$ is

```
Input:  ABCDAD---···-------:-----
Output: ---------···-------ABCDAD
```

Here, $T = 1000$ or $T = 2000$ so the network must memorize data over a very long sequence of void tokens. As in [44], we consider $n = 8$ and input length $K = 10$ and train networks with batch size 128 using the RMSProp algorithm. Naively predicting $T + K$ void tokens followed by K random selections of the n possible tokens achieves a baseline loss of $K \log(n)/(T + 2K)$. PROJUNN-T is able to learn the copy task efficiently as shown in Figure 4. In fact, for fixed learning rates, rank one approximations using the column sampling algorithm provide the fastest convergence to the true solution in comparison to higher rank approximations. Networks were initialized using Henaff initialization (see Appendix G.4) and the learning rate for unitary parameters was set to 32 times less than that of regular parameters (see Appendix D.3 for more details).

Permuted MNIST Another challenging long-term memory task we consider is the permuted pixel-by-pixel MNIST dataset. Here, MNIST images are flattened, and pixels are randomly shuffled and placed in a sequence thereby creating some non-local dependencies. MNIST images have 28×28 resolution, so the pixel-by-pixel sequences have length $T = 784$. The task is digit classification (10 classes) as in standard MNIST models. We employ the same data processing, shuffle permutation, and formatting as that in prior works [56]. We perform cross-validation over different learning rates and evaluate both PROJUNN-T and PROJUNN-D with different low-rank values $k \in \{1, 2, 4, 8, 16\}$. The final test accuracy is shown in Table 2. As observed in the copy and adding tasks, we find that using $k > 1$ does not lead to improved performances. In fact, we provide the evolution of the test set accuracy during training in Figure 13 and note that as the number of updates is large (hundreds per epoch), even rank $k = 1$ update are able to move the model's parameters to their local optimum.

CNN experiments To explore the performance of our PROJUNN training algorithm for convolutional layers, we first analyzed its performance on CIFAR10 classification using a Resnet architecture [34]. Our aim was not to "beat" benchmarks but to provide an honest comparison of the performance of PROJUNN to existing methods. In fact, as noted earlier, enforcing unitarity generically results in a drop in accuracy for commonly used architectures. Consistent with prior work [76] we employ

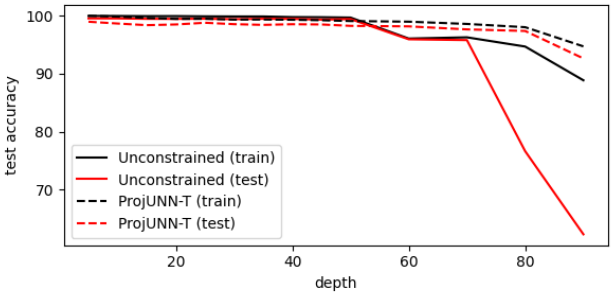


Figure 5: PROJUNN can more stably train very deep CNNs. Training on MNIST is done for 50 epochs in all cases with conv2d-BN-ReLU blocks (repeated "depth" times) and learning rate cross-validation (RMSprop), 32 channels throughout, and a final linear classifier. For 100 epochs and a depth of 100, we obtain $92.7, 23.5$ for the train/test accuracy of unconstrained CNN, and $95.7, 94.6$ for projUNN-T.

data-augmentation of random translations and left-right flips. Previous analysis in the RNN setting showed that rank $k = 1$ is sufficient for convergence so we always set $k = 1$ when using PROJUNN in the convolutional setting. For Resnet9 trained using the RMSprop optimizer, PROJUNN-T and PROJUNN-D reached 80.75% and 82.06% accuracy respectively, matching or outperforming reported results from existing unitary CNN models which achieved accuracies of 80.72% for BCOP [58] and 81.70% for Cayley [76] (further details in Appendix D.5). Note, that all of these methods resulted in a performance drop compared to the standard model (without unitary constraints) which achieved accuracy of 92.26%. Hence, we believe that there remain a large potential for unitary models to close this gap. Separate from just performance and to motivate the use of unitary parameterization, we provide in Figure 5, test accuracy results from a simple CNN model with progressively increasing depth trained with and without unitary parameterization on MNIST data. We observe that unitary weights might provide benefits for vanilla CNN architectures that have not been designed to handle very deep settings. Of course, various techniques and tricks have been designed to enable CNNs to be trainable at large depths [83, 34, 14]. Unitary convolutions, which are simple and theoretically motivated, can potentially be used either separately or in-tandem with these other techniques.

6 Discussion

Our PROJUNN shows that one need not sacrifice performance or runtime in training unitary neural network architectures. Our results broadly take advantage of the approximate low rank structure of parameter gradients to perform updates at nearly optimal runtime. Looking beyond the setting studied here, it is an interesting question how our framework can be applied to other neural network architectures or parameter manifolds. Group convolutional neural networks and Riemannian gradient descent offer two promising avenues for further application of our techniques.

References

[1] Scott Aaronson. Read the fine print. *Nature Physics*, 11(4):291–293, 2015.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Jonathan Allcock, Chang-Yu Hsieh, Iordanis Kerenidis, and Shengyu Zhang. Quantum algorithms for feedforward neural networks. *ACM Transactions on Quantum Computing*, 1(1):1–24, 2020.

[4] Eric R Anschuetz. Critical points in hamiltonian agnostic variational quantum algorithms. *arXiv preprint arXiv:2109.06957*, 2021.

[5] Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *International Conference on Machine Learning*, pages 1120–1128, 2016.

[6] Sanjeev Arora, Nadav Cohen, Wei Hu, and Yuping Luo. Implicit regularization in deep matrix factorization. *Advances in Neural Information Processing Systems*, 32:7413–7424, 2019.

[7] Sanjeev Arora, Rong Ge, Behnam Neyshabur, and Yi Zhang. Stronger generalization bounds for deep nets via a compression approach. In *International Conference on Machine Learning*, pages 254–263. PMLR, 2018.

[8] Juan Miguel Arrazola, Alain Delgado, Bhaskar Roy Bardhan, and Seth Lloyd. Quantum-inspired algorithms in practice. *arXiv preprint arXiv:1905.10415*, 2019.

[9] Nitin Bansal, Xiaohan Chen, and Zhangyang Wang. Can we gain more from orthogonality regularizations in training deep cnns? *arXiv preprint arXiv:1810.09102*, 2018.

[10] Kerstin Beer, Dmytro Bondarenko, Terry Farrelly, Tobias J Osborne, Robert Salzmann, Daniel Scheiermann, and Ramona Wolf. Training deep quantum neural networks. *Nature communications*, 11(1):1–6, 2020.

[11] Souheil Ben-Yacoub, B Fasel, and Juergen Luettin. Fast face detection using mlp and fft. In *Proc. Second International Conference on Audio and Video-based Biometric Person Authentication (AVBPA'99)*, number CONF, pages 31–36, 1999.

[12] Mathias Berglund, Tapani Raiko, Mikko Honkala, Leo Kärkkäinen, Akos Vetek, and Juha T Karhunen. Bidirectional recurrent neural networks as generative models. *Advances in Neural Information Processing Systems*, 28:856–864, 2015.

[13] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, 2017.

[14] Nils Bjorck, Carla P Gomes, Bart Selman, and Kilian Q Weinberger. Understanding batch normalization. *Advances in neural information processing systems*, 31, 2018.

[15] Silvere Bonnabel. Stochastic gradient descent on riemannian manifolds. *IEEE Transactions on Automatic Control*, 58(9):2217–2229, 2013.

[16] Grecia Castelazo, Quynh T Nguyen, Giacomo De Palma, Dirk Englund, Seth Lloyd, and Bobak T Kiani. Quantum algorithms for group convolution, cross-correlation, and equivariant transformations. *arXiv preprint arXiv:2109.11330*, 2021.

[17] Marco Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, et al. Variational quantum algorithms. *arXiv preprint arXiv:2012.09265*, 2020.

[18] Marco Cerezo, Akira Sone, Tyler Volkoff, Lukasz Cincio, and Patrick J Coles. Cost-function-dependent barren plateaus in shallow quantum neural networks. *arXiv e-prints*, pages arXiv–2001, 2020.

[19] Bo Chang, Minmin Chen, Eldad Haber, and Ed H Chi. Antisymmetricrnn: A dynamical system view on recurrent neural networks. *arXiv preprint arXiv:1902.09689*, 2019.

[20] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[21] Mark A Davenport and Justin Romberg. An overview of low-rank matrix recovery from incomplete observations. *IEEE Journal of Selected Topics in Signal Processing*, 10(4):608–622, 2016.

[22] Pierre De Fouquieres and Sophie G Schirmer. A closer look at quantum control landscapes and their implication for control optimization. *Infinite dimensional analysis, quantum probability and related topics*, 16(03):1350021, 2013.

[23] Petros Drineas, Ravi Kannan, and Michael W Mahoney. Fast monte carlo algorithms for matrices ii: Computing a low-rank approximation to a matrix. *SIAM Journal on computing*, 36(1):158–183, 2006.

[24] Rainer Engelken, Fred Wolf, and Larry F Abbott. Lyapunov spectra of chaotic recurrent neural networks. *arXiv preprint arXiv:2006.02427*, 2020.

[25] N Benjamin Erichson, Omri Azencot, Alejandro Queiruga, Liam Hodgkinson, and Michael W Mahoney. Lipschitz recurrent neural networks. *arXiv preprint arXiv:2006.12070*, 2020.

[26] Thomas Frerix and Joan Bruna. Approximating orthogonal matrices with effective givens factorization. In *International Conference on Machine Learning*, pages 1993–2001. PMLR, 2019.

[27] Alan Frieze, Ravi Kannan, and Santosh Vempala. Fast monte-carlo algorithms for finding low-rank approximations. *Journal of the ACM (JACM)*, 51(6):1025–1041, 2004.

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[29] Albert Gu, Tri Dao, Stefano Ermon, Atri Rudra, and Christopher Ré. Hippo: Recurrent memory with optimal polynomial projections. *arXiv preprint arXiv:2008.07669*, 2020.

[30] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

[31] Suriya Gunasekar, Blake Woodworth, Srinadh Bhojanapalli, Behnam Neyshabur, and Nathan Srebro. Implicit regularization in matrix factorization. In *2018 Information Theory and Applications Workshop (ITA)*, pages 1–10. IEEE, 2018.

[32] Brian Hall. *Lie groups, Lie algebras, and representations: an elementary introduction*, volume 222. Springer, 2015.

[33] Awni Y Hannun, Andrew L Maas, Daniel Jurafsky, and Andrew Y Ng. First-pass large vocabulary continuous speech recognition using bi-directional recurrent dnns. *arXiv preprint arXiv:1408.2873*, 2014.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[35] Kyle Helfrich, Devin Willmott, and Qiang Ye. Orthogonal recurrent neural networks with scaled cayley transform. In *International Conference on Machine Learning*, pages 1969–1978. PMLR, 2018.

[36] Mikael Henaff, Arthur Szlam, and Yann LeCun. Recurrent orthogonal networks and long-memory tasks. In *International Conference on Machine Learning*, pages 2034–2042. PMLR, 2016.

[37] Nicholas J Higham. The scaling and squaring method for the matrix exponential revisited. *SIAM review*, 51(4):747–764, 2009.

[38] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[39] Emiel Hoogeboom, Victor Garcia Satorras, Jakub M Tomczak, and Max Welling. The convolution exponential and generalized sylvester flows. *arXiv preprint arXiv:2006.01910*, 2020.

[40] Lei Huang, Li Liu, Fan Zhu, Diwen Wan, Zehuan Yuan, Bo Li, and Ling Shao. Controllable orthogonalization in training dnns. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6429–6438, 2020.

[41] Stephanie Hyland and Gunnar Rätsch. Learning unitary operators with help from u (n). In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[42] Yani Ioannou, Duncan Robertson, Jamie Shotton, Roberto Cipolla, and Antonio Criminisi. Training cnns with low-rank filters for efficient image classification. *arXiv preprint arXiv:1511.06744*, 2015.

[43] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[44] Li Jing, Yichen Shen, Tena Dubcek, John Peurifoy, Scott Skirlo, Yann LeCun, Max Tegmark, and Marin Soljačić. Tunable efficient unitary neural networks (eunn) and their application to rnns. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1733–1741. JMLR. org, 2017.

[45] Jaroslav Kautsky and Radka Turcajová. A matrix approach to discrete wavelets. In *Wavelet Analysis and Its Applications*, volume 5, pages 117–135. Elsevier, 1994.

[46] Joseph B Keller. Closest unitary, orthogonal and hermitian operators to a given operator. *Mathematics Magazine*, 48(4):192–197, 1975.

[47] Iordanis Kerenidis, Jonas Landman, and Anupam Prakash. Quantum algorithms for deep convolutional neural networks. *arXiv preprint arXiv:1911.01117*, 2019.

[48] Amir Khoshaman, Walter Vinci, Brandon Denis, Evgeny Andriyash, Hossein Sadeghi, and Mohammad H Amin. Quantum variational autoencoder. *Quantum Science and Technology*, 4(1):014001, 2018.

[49] Bobak Toussi Kiani, Giacomo De Palma, Milad Marvian, Zi-Wen Liu, and Seth Lloyd. Quantum earth mover's distance: A new approach to learning quantum data. *arXiv preprint arXiv:2101.03037*, 2021.

[50] Bobak Toussi Kiani, Seth Lloyd, and Reevu Maity. Learning unitaries by gradient descent. *arXiv preprint arXiv:2001.11897*, 2020.

[51] Nathan Killoran, Thomas R Bromley, Juan Miguel Arrazola, Maria Schuld, Nicolás Quesada, and Seth Lloyd. Continuous-variable quantum neural networks. *Physical Review Research*, 1(3):033063, 2019.

[52] Alexander Kirillov Jr. *An introduction to Lie groups and Lie algebras*. Number 113. Cambridge University Press, 2008.

[53] Jan Koutnik, Klaus Greff, Faustino Gomez, and Juergen Schmidhuber. A clockwork rnn. In *International Conference on Machine Learning*, pages 1863–1871. PMLR, 2014.

[54] Hannah Lawrence, Kristian Georgiev, Andrew Dienes, and Bobak T Kiani. Implicit bias of linear equivariant networks. *arXiv preprint arXiv:2110.06084*, 2021.

[55] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[56] Mario Lezcano-Casado and David Martınez-Rubio. Cheap orthogonal constraints in neural networks: A simple parametrization of the orthogonal and unitary group. In *International Conference on Machine Learning*, pages 3794–3803. PMLR, 2019.

[57] Jun Li, Li Fuxin, and Sinisa Todorovic. Efficient riemannian optimization on the stiefel manifold via the cayley transform. *arXiv preprint arXiv:2002.01113*, 2020.

[58] Qiyang Li, Saminul Haque, Cem Anil, James Lucas, Roger B Grosse, and Jörn-Henrik Jacobsen. Preventing gradient attenuation in lipschitz constrained convolutional networks. *Advances in neural information processing systems*, 32:15390–15402, 2019.

[59] Shuai Li, Kui Jia, Yuxin Wen, Tongliang Liu, and Dacheng Tao. Orthogonal deep neural networks. *IEEE transactions on pattern analysis and machine intelligence*, 43(4):1352–1368, 2019.

[60] Yuanzhi Li, Tengyu Ma, and Hongyang Zhang. Algorithmic regularization in over-parameterized matrix sensing and neural networks with quadratic activations. In *Conference On Learning Theory*, pages 2–47. PMLR, 2018.

[61] Weiyang Liu, Rongmei Lin, Zhen Liu, James M Rehg, Liam Paull, Li Xiong, Le Song, and Adrian Weller. Orthogonal over-parameterized training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7251–7260, 2021.

[62] Yi-Kai Liu. Universal low-rank matrix recovery from pauli measurements. *Advances in Neural Information Processing Systems*, 24:1638–1646, 2011.

[63] Charles H Martin and Michael W Mahoney. Implicit self-regularization in deep neural networks: Evidence from random matrix theory and implications for learning. *arXiv preprint arXiv:1810.01075*, 2018.

[64] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

[65] Jarrod R McClean, Sergio Boixo, Vadim N Smelyanskiy, Ryan Babbush, and Hartmut Neven. Barren plateaus in quantum neural network training landscapes. *Nature communications*, 9(1):1–6, 2018.

[66] Zakaria Mhammedi, Andrew Hellicar, Ashfaqur Rahman, and James Bailey. Efficient orthogonal parametrisation of recurrent neural networks using householder reflections. In *International Conference on Machine Learning*, pages 2401–2409. PMLR, 2017.

[67] Yasunori Nishimori and Shotaro Akaho. Learning algorithms utilizing quasi-geodesic flows on the stiefel manifold. *Neurocomputing*, 67:106–135, 2005.

[68] Christos H Papadimitriou, Prabhakar Raghavan, Hisao Tamaki, and Santosh Vempala. Latent semantic indexing: A probabilistic analysis. *Journal of Computer and System Sciences*, 61(2):217–235, 2000.

[69] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[70] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[71] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. The quest for a quantum neural network. *Quantum Information Processing*, 13(11):2567–2586, 2014.

[72] Hanie Sedghi, Vineet Gupta, and Philip M Long. The singular values of convolutional layers. *arXiv preprint arXiv:1805.10408*, 2018.

[73] Sahil Singla and Soheil Feizi. Skew orthogonal convolutions. *arXiv preprint arXiv:2105.11417*, 2021.

[74] Sridhar Swaminathan, Deepak Garg, Rajkumar Kannan, and Frederic Andres. Sparse low rank factorization for deep neural network compression. *Neurocomputing*, 398:185–196, 2020.

[75] Cheng Tai, Tong Xiao, Yi Zhang, Xiaogang Wang, et al. Convolutional neural networks with low-rank regularization. *arXiv preprint arXiv:1511.06067*, 2015.

[76] Asher Trockman and J Zico Kolter. Orthogonalizing convolutional layers with the cayley transform. *arXiv preprint arXiv:2104.07167*, 2021.

[77] Aaron Voelker, Ivana Kajić, and Chris Eliasmith. Legendre memory units: Continuous-time representation in recurrent neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[78] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. 2019.

[79] Daochen Wang, Oscar Higgott, and Stephen Brierley. Accelerated variational quantum eigensolver. *Physical review letters*, 122(14):140504, 2019.

[80] Jiayun Wang, Yubei Chen, Rudrasis Chakraborty, and Stella X Yu. Orthogonal convolutional neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11505–11515, 2020.

[81] Dave Wecker, Matthew B Hastings, and Matthias Troyer. Progress towards practical quantum variational algorithms. *Physical Review A*, 92(4):042303, 2015.

[82] Scott Wisdom, Thomas Powers, John R Hershey, Jonathan Le Roux, and Les Atlas. Full-capacity unitary recurrent neural networks. *arXiv preprint arXiv:1611.00035*, 2016.

[83] Lechao Xiao, Yasaman Bahri, Jascha Sohl-Dickstein, Samuel Schoenholz, and Jeffrey Pennington. Dynamical isometry and a mean field theory of cnns: How to train 10,000-layer vanilla convolutional neural networks. In *International Conference on Machine Learning*, pages 5393–5402. PMLR, 2018.

[84] Xiyu Yu, Tongliang Liu, Xinchao Wang, and Dacheng Tao. On compressing deep models by low rank and sparse decomposition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7370–7379, 2017.

[85] Xiao Yuan, Suguru Endo, Qi Zhao, Ying Li, and Simon C Benjamin. Theory of variational quantum simulation. *Quantum*, 3:191, 2019.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] Contributions and scope of the paper are specified in the abstract and introduction.
   (b) Did you describe the limitations of your work? [Yes] Limitations are addressed throughout this work, including after the descriptions of the algorithm in Section 4.1 and Section 4.4, in Appendix G.3, in our comparison to related work in Appendix B, in the runtime comparison of Appendix F, and various other places.
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] We confirm that we have read the ethics guideline to ensure our paper conforms to it.
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] All assumptions are given in the sections themselves or part of the statements of the relevant theorems, propositions, and lemmas.
   (b) Did you include complete proofs of all theoretical results? [Yes] Proofs are included for all results and deferred to the appendix.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] An anonymized demo is included in this anonymous manuscript. Full code will be made available upon publication.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Relevant details are included in Section 5 and Appendix D.

(c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] we employed the same experiment setup as prior work that consists in reporting the best test performance based on validation set performances after cross-validation. As we report such numbers for a variety of settings, error bars could be derived from our tables.

(d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] Relevant details are included in Appendix G.

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...

   (a) If your work uses existing assets, did you cite the creators? [Yes] Please refer to Appendix G.

   (b) Did you mention the license of the assets? [Yes] Please refer to Appendix G.

   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]

   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]

   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...

   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Mathematical background

A.1 Lie groups and Lie algebras

Here, we provide a brief mathematical background to Lie groups and Lie algebras with a particular focus on the unitary and orthogonal groups. For a more comprehensive overview, we recommend [32].

Though Lie groups are typically defined with respect to differentiable manifolds, we restrict ourselves here to the subset of matrix Lie groups which is less general but allows for a more concise and simpler theoretical overview. Informally, Lie groups are groups whose elements are specified by a set of parameters that vary smoothly and continuously, *i.e.,* the group is also a differentiable manifold. Specific to matrices, Lie groups are commonly defined with respect to the general linear group $GL(n, \mathbb{C})$ denoting the set of invertible $n \times n$ matrices with complex valued entries [32]. Lie groups are closed subgroups of $GL(n, \mathbb{C})$ that have the following smoothness property.

Definition A.1 (Matrix Lie groups [32])**.** A matrix Lie group is any subgroup of $GL(n, \mathbb{C})$ with the property that any sequence of matrices $\boldsymbol{A}_m \in \mathbb{C}^{n\times n}$ in the subgroup converge to a matrix $\boldsymbol{A}$ that is either an element of the subgroup or not invertible (*i.e.,* not in $GL(n, \mathbb{C})$).

Two important Lie groups studied in this work are the unitary and orthogonal groups whose definitions are copied below.

$$O(n) = \left\{\boldsymbol{M} \in \mathbb{R}^{n\times n} | \boldsymbol{M}\boldsymbol{M}^\intercal = \boldsymbol{M}^\intercal\boldsymbol{M} = \boldsymbol{I}\right\}, \tag{A.1}$$

$$U(n) = \left\{\boldsymbol{M} \in \mathbb{C}^{n\times n} | \boldsymbol{M}\boldsymbol{M}^\dagger = \boldsymbol{M}^\dagger\boldsymbol{M} = \boldsymbol{I}\right\}. \tag{A.2}$$

The Lie algebra is the tangent space of a Lie group at the identity element. To observe this, we introduce the matrix exponential map which is central to the connection between Lie groups and their corresponding Lie algebras.

$$\exp(\boldsymbol{M}) = \sum_{k=0}^{\infty} \frac{\boldsymbol{M}^k}{k!}. \tag{A.3}$$

For compact groups, the exponential map is a smooth map whose image is the connected component to the identity of the Lie group [52, 32]. The special orthogonal and unitary groups are both compact and connected so the exponential map is surjective for these groups (*i.e.,* for every group element, there exists an element of the Lie algebra whose exponential is equal to that group element). However, the orthogonal group has two connected components, *i.e.,* elements with positive and negative determinant, and the image of exponential map are only orthogonal matrices with positive determinant.

Since the matrix exponential map is a smooth function, we can take the derivative of the exponential map with respect to a parameter as below.

$$\frac{d}{dt}\exp(t\boldsymbol{X}) = \boldsymbol{X}\exp(t\boldsymbol{X}) = \exp(t\boldsymbol{X})\boldsymbol{X}, \tag{A.4}$$

and thus,

$$\frac{d}{dt}\exp(t\boldsymbol{X})\Big|_{t=0} = \boldsymbol{X}. \tag{A.5}$$

The above gives us the Lie algebra to a given group.

Definition A.2 (Lie algebra [32])**.** Given a Lie group G, the Lie algebra $\mathfrak{g}$ of G is the set of matrices $\boldsymbol{X}$ such that $e^{t\boldsymbol{X}} \in G$ for all $t \in \mathbb{R}$.

Typically, Lie algebras of a Lie group are denoted with Gothic or Fraktur font. Using the above definition, one can construct the corresponding Lie algebras. As an example, consider the unitary group where given a matrix $\boldsymbol{U} \in U(n)$ and $\boldsymbol{X} \in \mathfrak{u}(n)$,

$$\boldsymbol{U}^{-1} = \boldsymbol{U}^\dagger \iff \exp(t\boldsymbol{X})^{-1} = \exp(-t\boldsymbol{X}) = \exp(t\boldsymbol{X})^\dagger = \exp(t\boldsymbol{X}^\dagger), \tag{A.6}$$

and since the above holds for all $t \in \mathbb{R}$, we can differentiate the above at $t = 0$, obtaining the property of elements of the unitary Lie algebra that $-\boldsymbol{X} = \boldsymbol{X}^\dagger$ as seen in the main text:

$$\begin{aligned} \frac{d}{dt}\exp(-t\boldsymbol{X})\Big|_{t=0} &= \frac{d}{dt}\exp(t\boldsymbol{X}^\dagger)\Big|_{t=0} \\ -\boldsymbol{X} &= \boldsymbol{X}^\dagger. \end{aligned} \tag{A.7}$$

Proceeding in a similar fashion with the orthogonal group, we obtain the following result copied from the main text,

$$\mathfrak{o}(n) = \left\{\boldsymbol{A} \in \mathbb{R}^{n\times n} : \boldsymbol{A} + \boldsymbol{A}^\intercal = 0\right\}, \tag{A.8}$$

$$\mathfrak{u}(n) = \left\{\boldsymbol{A} \in \mathbb{C}^{n\times n} : \boldsymbol{A} + \boldsymbol{A}^\dagger = 0\right\}. \tag{A.9}$$

A.2 Projecting onto the group or algebra

Projections onto the orthogonal/unitary groups or their tangent spaces are central to optimizing over the space of orthogonal/unitary matrices. We focus the discussion here to the case of unitary matrices, but note that all of the following statements apply to orthogonal matrices as well by simple adjustments such as replacing the conjugate transpose (†) with the transpose ($^\intercal$). The tangent space $T_{\boldsymbol{U}}$ to a matrix $\boldsymbol{U} \in U(n)$ is equal to

$$T_{\boldsymbol{U}}U(n) = \left\{\boldsymbol{X} \in \mathbb{C}^{n\times n} : \boldsymbol{U}^\dagger \boldsymbol{X} + \boldsymbol{X}^\dagger \boldsymbol{U} = 0\right\}. \tag{A.10}$$

Given the canonical inner product $\langle \boldsymbol{A}, \boldsymbol{B}\rangle = \operatorname{Re}\left(\operatorname{Tr}[\boldsymbol{A}^\dagger \boldsymbol{B}]\right)$, we can show that the orthogonal projection onto the tangent space $T_{\boldsymbol{U}}U(n)$ is equal to that given by Lemma 4.3 copied below.

Lemma 4.3 (Tangent space projection [82])**.** *Given the tangent space $T_{\boldsymbol{U}}U(n)$ of an orthogonal/unitary matrix $\boldsymbol{U}$, the orthogonal projection $\Pi_{T_{\boldsymbol{U}}}$ with respect to the canonical metric $\langle \boldsymbol{X}, \boldsymbol{Y}\rangle = \operatorname{Re}\left(\operatorname{Tr}[\boldsymbol{X}^\dagger \boldsymbol{Y}]\right)$ is*

$$\Pi_{T_{\boldsymbol{U}}}(\boldsymbol{X}) = \frac{1}{2}\left(\boldsymbol{X} - \boldsymbol{U}\boldsymbol{X}^\dagger \boldsymbol{U}\right). \tag{5}$$

Similar to Lemma 4.1, this projection also returns the closest matrix in Frobenius norm to $\boldsymbol{X}$ in the tangent space,

$$\min_{\boldsymbol{Y}\in T_{\boldsymbol{U}}U(n)} \|\boldsymbol{Y} - \boldsymbol{X}\|_F = \Pi_{T_{\boldsymbol{U}}}(\boldsymbol{X}). \tag{6}$$

Proof. By the definition of an orthogonal projection, we need that $\Pi_{T_{\boldsymbol{U}}}(\Pi_{T_{\boldsymbol{U}}}(\cdot)) = \Pi_{T_{\boldsymbol{U}}}(\cdot)$ and for all $\boldsymbol{X}, \boldsymbol{Y} \in T_{\boldsymbol{U}}U(n)$ $\Pi_{T_{\boldsymbol{U}}}(\boldsymbol{X}) = \boldsymbol{X}$ and $\langle \Pi_{T_{\boldsymbol{U}}}(\boldsymbol{X}), \boldsymbol{X} - \Pi_{T_{\boldsymbol{U}}}(\boldsymbol{X})\rangle = 0$. The first two properties are straightforward to check. The last property can be shown as below using the definition of $\Pi_{T_{\boldsymbol{U}}}$ and cyclic property of trace:

$$\begin{aligned}
\langle \Pi_{T_{\boldsymbol{U}}}(\boldsymbol{X}), \boldsymbol{X} - \Pi_{T_{\boldsymbol{U}}}(\boldsymbol{X})\rangle &= \frac{1}{4}\operatorname{Re}\left(\operatorname{Tr}[(\boldsymbol{X} - \boldsymbol{U}\boldsymbol{X}^\dagger\boldsymbol{U})^\dagger(\boldsymbol{X} + \boldsymbol{U}\boldsymbol{X}^\dagger\boldsymbol{U})]\right)\\
&= \frac{1}{4}\operatorname{Re}\left(\operatorname{Tr}[\boldsymbol{X}^\dagger\boldsymbol{X}] - \operatorname{Tr}[\boldsymbol{U}^\dagger\boldsymbol{X}\boldsymbol{X}^\dagger\boldsymbol{U}] + \operatorname{Tr}[\boldsymbol{X}^\dagger\boldsymbol{U}\boldsymbol{X}^\dagger\boldsymbol{U}] - \operatorname{Tr}[\boldsymbol{U}^\dagger\boldsymbol{X}\boldsymbol{U}^\dagger\boldsymbol{X}]\right)\\
&= \frac{1}{4}\operatorname{Re}\left(\operatorname{Tr}[(\boldsymbol{X}^\dagger\boldsymbol{U})^2] - \operatorname{Tr}[((\boldsymbol{X}^\dagger\boldsymbol{U})^2)^\dagger]\right)\\
&= 0.
\end{aligned} \tag{A.11}$$

Furthermore, we have for all $\boldsymbol{Y} \in T_{\boldsymbol{U}}U(n)$ using triangle inequality and unitary invariance of the Frobenius norm:

$$\begin{aligned}
\|\Pi_{T_{\boldsymbol{U}}}(\boldsymbol{X}) - \boldsymbol{X}\|_F &= \left\|\frac{1}{2}(\boldsymbol{X} - \boldsymbol{U}\boldsymbol{X}^\dagger\boldsymbol{U}) - \boldsymbol{X}\right\|_F\\
&\leq \frac{1}{2}\|\boldsymbol{Y} - \boldsymbol{X}\|_F + \frac{1}{2}\left\|\boldsymbol{Y} + \boldsymbol{U}\boldsymbol{X}^\dagger\boldsymbol{U}\right\|_F\\
&= \frac{1}{2}\|\boldsymbol{Y} - \boldsymbol{X}\|_F + \frac{1}{2}\left\|-\boldsymbol{U}\boldsymbol{Y}^\dagger\boldsymbol{U} + \boldsymbol{U}\boldsymbol{X}^\dagger\boldsymbol{U}\right\|_F\\
&= \frac{1}{2}\|\boldsymbol{Y} - \boldsymbol{X}\|_F + \frac{1}{2}\|\boldsymbol{Y} - \boldsymbol{X}\|_F\\
&= \|\boldsymbol{Y} - \boldsymbol{X}\|_F,
\end{aligned} \tag{A.12}$$

which proves that $\Pi_{T_{\boldsymbol{U}}}(\cdot)$ projects onto the closest matrix $\boldsymbol{Y} \in T_{\boldsymbol{U}}U(n)$ in Frobenius norm. □

Since the set of unitary/orthogonal matrices does not form a vector space, an orthogonal projection is not a well defined operation in this space. However, it is still valid to ask what is the "closest" unitary/orthogonal matrix to a given matrix in a given norm. This is exactly what is stated in Lemma 4.1 copied from the main text and proven below.

Lemma 4.1 (Projection onto unitary manifold [46])**.** *Given a matrix $\boldsymbol{A} \in \mathbb{C}^{n\times n}$:*

$$\Pi_U(\boldsymbol{A}) = \operatorname*{arg\,min}_{\boldsymbol{U}\in\mathcal{U}(n)} \|\boldsymbol{A} - \boldsymbol{U}\|_F^2 = \boldsymbol{A}(\boldsymbol{A}^\dagger\boldsymbol{A})^{-\frac{1}{2}}, \tag{3}$$

where $\mathcal{U}(n)$ indicates the set of $n \times n$ unitary matrices.

Proof. We follow the approach of [46] to prove this result. To shorten our notation, let $\boldsymbol{V} = \boldsymbol{A}(\boldsymbol{A}^\dagger \boldsymbol{A})^{-1/2}$. Given any unitary $\boldsymbol{U} \in U(n)$, let $\boldsymbol{U} = \boldsymbol{M} + \boldsymbol{V}$ for the properly chosen $\boldsymbol{M} \in \mathbb{C}^{n\times n}$. From unitarity of $\boldsymbol{U}$ and $\boldsymbol{V}$, we have

$$\boldsymbol{M}\boldsymbol{V}^\dagger + \boldsymbol{M}\boldsymbol{M}^\dagger + \boldsymbol{V}\boldsymbol{M}^\dagger = 0. \tag{A.13}$$

Then,

$$\begin{aligned}
\|\boldsymbol{A} - \boldsymbol{U}\|_F^2 &= \|\boldsymbol{A} - \boldsymbol{M} - \boldsymbol{V}\|_F^2 \\
&= \|\boldsymbol{A} - \boldsymbol{V}\|_F^2 + \mathrm{Tr}[\boldsymbol{M}\boldsymbol{V}^\dagger + \boldsymbol{M}\boldsymbol{M}^\dagger + \boldsymbol{V}\boldsymbol{M}^\dagger] - \mathrm{Tr}[\boldsymbol{M}^\dagger \boldsymbol{A} + \boldsymbol{A}^\dagger \boldsymbol{M}] \\
&= \|\boldsymbol{A} - \boldsymbol{V}\|_F^2 - \mathrm{Tr}[\boldsymbol{M}^\dagger \boldsymbol{A} + \boldsymbol{A}^\dagger \boldsymbol{M}] \\
&= \|\boldsymbol{A} - \boldsymbol{V}\|_F^2 - \mathrm{Tr}[\boldsymbol{M}^\dagger \boldsymbol{V}(\boldsymbol{A}^\dagger \boldsymbol{A})^{1/2} + (\boldsymbol{A}^\dagger \boldsymbol{A})^{1/2}\boldsymbol{V}^\dagger \boldsymbol{M}],
\end{aligned} \tag{A.14}$$

and since from Equation (A.13) we have that $\boldsymbol{M}\boldsymbol{V}^\dagger + \boldsymbol{V}\boldsymbol{M}^\dagger = -\boldsymbol{M}\boldsymbol{M}^\dagger$,

$$\|\boldsymbol{A} - \boldsymbol{U}\|_F^2 = \|\boldsymbol{A} - \boldsymbol{V}\|_F^2 + \mathrm{Tr}[(\boldsymbol{A}^\dagger \boldsymbol{A})^{1/2}\boldsymbol{M}\boldsymbol{M}^\dagger]. \tag{A.15}$$

The second term above is non-negative since $\mathrm{Tr}[(\boldsymbol{A}^\dagger \boldsymbol{A})^{1/2}\boldsymbol{M}\boldsymbol{M}^\dagger] = \mathrm{Tr}[\boldsymbol{M}^\dagger(\boldsymbol{A}^\dagger \boldsymbol{A})^{1/2}\boldsymbol{M}]$ and $(\boldsymbol{A}^\dagger \boldsymbol{A})^{1/2}$ is positive semi-definite. Thus, for all $\boldsymbol{U} \in U(n)$,

$$\|\boldsymbol{A} - \boldsymbol{U}\|_F^2 \geq \|\boldsymbol{A} - \boldsymbol{V}\|_F^2, \tag{A.16}$$

which proves the result. □

B Review of previous unitary neural network techniques

The integration of orthogonal/unitary matrices into neural networks is broadly aimed at maintaining stability in neural networks with many layers. For vanilla recurrent neural networks, repetitive application of the hidden-to-hidden transformation matrix exponentially amplifies or decays the eigenvalues of the transformation, thus resulting in exponentially large or small gradients. Similarly, in deep network architectures where weight matrices are drawn randomly, the norms of hidden states can similarly grow or decay exponentially with added layers. Enforcing unitarity or orthogonality of neural network layer transformations offers a straightforward method to address these issues of instability since orthogonal/unitary matrices have eigenvalues of unity.

To establish notation and provide motivation for later analysis, consider a RNN whose input is a sequence of vectors $\boldsymbol{x}(t)$ with hidden layer $\boldsymbol{h}(t)$ updated according to the following rule:

$$\boldsymbol{h}^{(t)} = \sigma(\boldsymbol{M}\mathbf{x}^{(t)} + \boldsymbol{W}\boldsymbol{h}^{(t-1)}) \tag{B.1}$$

In the unitary or orthogonal formulation for RNNs, the matrix $\boldsymbol{W}$ in Equation (B.1) is replaced by a unitary or orthogonal matrix $\boldsymbol{U}$. During optimization of a unitary RNN, one must enforce the unitarity or orthogonality of $\boldsymbol{U}$ during training.

Existing methods to parameterize and enforce unitarity/orthogonality in neural network layers can be separated into three categories depending on the method of parameterization. We discuss each of these in detail below:

- **Layer-wise transformations:** among the first methods employed in this line of work, these methods parameterize orthogonal/unitary matrices in a layer-wise fashion where each layer is a parameterized orthogonal/unitary matrix. Example parameterizations include Givens rotations and Householder reflections. These methods are efficient when there are not many layers included in the parameterization and typically not employed when full access to the unitary/orthogonal group is required as full parameterization of an $n \times n$ unitary/orthogonal matrix requires $O(n)$ layers.
- **Lie algebra parameterization:** motivated by the fact that staying on the manifold of the Lie algebra is often easier than staying on the manifold of a Lie group, these methods parameterize the Lie algebra of the unitary/orthogonal groups and later obtain the actual unitary/orthogonal matrix by implementing the matrix exponential map. This matrix exponential map is typically the most costly step in these methods as one can either perform it directly (*e.g.,* using an SVD) or approximate it via Taylor series or Padé approximations which require repetitive application of matrices when applying it to an input. Though we adapt techniques from these methods in our work to efficiently perform gradient updates, we do not parameterize matrices in the Lie algebra.

- **Matrix entry parameterization:** as in typical neural network architectures, this method parameterizes an orthogonal/unitary matrix by directly parameterizing the entries of the matrix. This method is optimal in the "forward" direction since performing the transformation on an input simply requires matrix multiplication (as in vanilla architectures). However, updates to the matrix will no longer maintain unitarity or orthogonality, and one must employ methods to project these updates back onto the unitary/orthogonal manifold. This method is employed in our work.

We now present each of the above methods in the order given.

Layer-wise transformations Early algorithms [44, 5, 66] maintained unitarity by parameterizing a matrix $\boldsymbol{U}$ as a series or layered set of k parameterized unitary transformations:

$$\boldsymbol{U} = \boldsymbol{F}_{uni}^{(1)}(\theta_1)\boldsymbol{F}_{uni}^{(2)}(\theta_2)\cdots\boldsymbol{F}_{uni}^{(k)}(\theta_k). \tag{B.2}$$

where $\boldsymbol{F}_{uni}^{(i)}(\theta_i)$ indicates a transformation that maps parameters θ_i into a unitary matrix. These transformations include parameterized Givens rotations or Householder reflections. As a concrete example, consider the Givens rotation parameterization which is an orthogonal matrix that performs a rotation of the i, j-th dimensions by an amount θ:

$$\boldsymbol{G}(i,j,\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \cos\theta & \cdots & -\sin\theta & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & \sin\theta & \cdots & \cos\theta & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}, \tag{B.3}$$

or in other words, entries $G_{ii} = G_{jj} = \cos\theta$, $G_{ij} = -G_{ji} = \sin\theta$, and $G_{kl} = \delta_{kl}$ for all other entries.

In general, at least $k = O(n)$ layers are needed to achieve a full parameterization of the $n \times n$ unitary matrices. Training is performed by updating the parameters θ_i within each layer. However, for large matrices, due to the fact that $O(n)$ layers are required to parameterize the full space of transformations, these algorithms are only efficient when parameterizations over a subset of the space of unitary/orthogonal matrices suffices. Achieving this balance of parameterization versus performance is challenging as prior work – especially work studying the learnability of unitary matrices in quantum computation – has shown that loss landscapes over the unitary/orthogonal manifold may contain many bad local minima [50, 26, 4, 22].

Lie algebra parameterization As a reminder, the Lie algebra of the orthogonal and unitary groups are the set of skew symmetric ($\mathfrak{o}(n)$) and skew Hermitian ($\mathfrak{u}(n)$) matrices,

$$\mathfrak{o}(n) = \left\{\boldsymbol{A} \in \mathbb{R}^{n\times n} : \boldsymbol{A} + \boldsymbol{A}^\intercal = 0\right\}, \tag{B.4}$$

$$\mathfrak{u}(n) = \left\{\boldsymbol{A} \in \mathbb{C}^{n\times n} : \boldsymbol{A} + \boldsymbol{A}^\dagger = 0\right\}. \tag{B.5}$$

Transformations from the Lie algebra to the Lie group are performed using the exponential map which is surjective onto the connected components of the identity:

$$\exp(\boldsymbol{X}) = \sum_{k=0}^{\infty} \frac{\boldsymbol{X}^k}{k!} = \boldsymbol{I} + \boldsymbol{X} + \frac{1}{2}\boldsymbol{X}^2 + \frac{1}{6}\boldsymbol{X}^3 + \cdots \tag{B.6}$$

Thus, these methods parameterize the full space of unitary or special orthogonal matrices. However, the orthogonal group has two connected components (matrices with determinant equal to one and negative one) so the exponential maps only onto the positive determinant matrices.

Note that the Lie algebra is a vector space so the sum of two matrices in the Lie algebra is also in the algebra. Thus, gradient updates can typically be very easily performed. However, the exponential map is often expensive to compute, and much prior work has focused on implementing this transformation efficiently. [35] and [73] employ approximations to the matrix exponential via Padé or Taylor series approximations. For example, in the Taylor series approximation, one simply truncates the Taylor series to K entries:

$$\exp(\boldsymbol{X}) \approx \sum_{k=0}^{K} \frac{\boldsymbol{X}^k}{k!}, \qquad \left\|\exp(\boldsymbol{X}) - \sum_{k=0}^{K} \frac{\boldsymbol{X}^k}{k!}\right\|_2 \leq \frac{\|\boldsymbol{X}\|_2^k}{k!}, \tag{B.7}$$

where the norm $\|\cdot\|_2$ indicates the spectral norm (*i.e.,* largest singular value). This application of the matrix exponential comes at an added cost both when calculating derivatives and when applying the matrix in the forward direction (*e.g.,* when one simply desires the output of a network). Applying the approximation above to the input of a layer requires $O(K)$ applications of the matrix. The matrix $\exp(\boldsymbol{X})$ can be explicitly constructed to avoid this added cost; however, obtaining this matrix requires a one time cost of $O(K)$ matrix-matrix multiplication operations which can be costly for large matrices.

The Padé approximants are more typically used in approximating the exponential map since they can guarantee that the approximated matrix is actually a unitary/orthogonal matrix (as opposed to a simple Taylor series approximation which does not provide this guarantee). A Padé approximant is an optimal rational function approximation to a given function. The Padé approximant to the exponential map takes the form $\exp(\boldsymbol{A}) \approx r_{mn}(\boldsymbol{A}) = p_m(\boldsymbol{A})q_n(\boldsymbol{A})^{-1}$ where $p_m(\cdot)$ and $q_n(\cdot)$ are order m and n polynomials respectively which have closed forms:

$$p_m(\boldsymbol{A}) = \sum_{k=0}^{m} \frac{(m+n-k)!m!}{(m+n)!(m-k)!k!}\boldsymbol{A}^k, \qquad q_n(\boldsymbol{A}) = \sum_{k=0}^{n} \frac{(m+n-k)!n!}{(m+n)!(n-k)!k!}(-\boldsymbol{A})^k. \tag{B.8}$$

The error of the approximation scales as $O(\|\boldsymbol{A}\|^{m+n+1})$ [37]. For unitary/orthogonal matrices, this approximation has the feature that for order $m = n$, applying the approximation to an element of the Lie algebra of the orthogonal/unitary matrices outputs a orthogonal/unitary matrix. Notably, setting $m = n = 1$ obtains the Cayley transform which was used in [35].

Finally, we note that approximations to the exponential function often can bias the gradients so in the RNN setting, [56] actually perform an SVD to obtain the actual output of the exponential map and analytically calculate gradients.

Matrix entry parameterization Parameterizing the entries of a matrix $\boldsymbol{U}$ directly is an obvious and simple means of constructing a unitary or orthogonal matrix. However, the set of unitary/orthogonal matrices do not form an algebra so one cannot simply update these matrices by simply adding a gradient update to a given matrix. Instead, updates to these matrices must be performed by projecting the updated matrix back onto the set of unitary/orthogonal matrices. Obviously, $\boldsymbol{U}$ must be initialized to be unitary/orthogonal in these methods.

Given a gradient update $\boldsymbol{G}$, the matrix $\boldsymbol{U} + \boldsymbol{G}$ is typically no longer unitary. Methods of Riemannian optimization are employed to update the matrix in a unitary/orthogonal fashion. One means of updating the matrix is via the Cayley transform which computes a parametric curve in the direction, employed in [57, 82, 61, 76]. The Cayley transform "transports" $\boldsymbol{U}$ in the direction of the projection of the gradient in the tangent space $\Pi(\boldsymbol{G})$. *i.e.,* let $\boldsymbol{U}(0)$ be the initial unitary matrix, then one can transport the matrix in the direction of $\Pi(\boldsymbol{G})$ (where $\Pi(\cdot)$ is the projection onto the tangent space) by a "length" of α via the Cayley transform [67],

$$\boldsymbol{U}(\alpha) = \left(\boldsymbol{I} - \frac{\alpha}{2}\Pi(\boldsymbol{G})\right)^{-1}\left(\boldsymbol{I} + \frac{\alpha}{2}\Pi(\boldsymbol{G})\right)\boldsymbol{U}(0). \tag{B.9}$$

The above update formula requires a matrix inversion step which is the most costly step for large matrices. [57, 61] approximate the transformation via a fixed point interation which avoids having to invert a matrix but still requires matrix-matrix multiplication. More generally, the Cayley transform is a first order Padé approximant to the exponential map which provides a connection between the matrix entry parameterization and the Lie algebra parameterization discussed previously [37], *i.e.,* setting $m = n = 1$ in Equation (B.8) obtains the Cayley transform.

Our algorithms described in the main text directly parameterize matrix entries and thus follow this parameterization method. In performing the exponential map, we do not resort to any approximations since the exponential is efficient to perform in low rank settings. Updates to unitary matrices are either projected directly onto the closest unitary matrix in Frobenius norm (PROJUNN-D) or transported along the geodesic in the direction of the projection of the gradient onto the tangent space (PROJUNN-T). We refer the reader to the main text for a full description of our methodology.

B.1 Orthogonal or unitary convolution

Since convolutions are linear operators, one can perform orthogonal or unitary convolutions using similar techniques as in the general case studied above. As a reminder, for 2-D convolution, given input tensor $\mathbf{X} \in \mathbb{C}^{M\times N\times C}$ where C denotes the number of channels of the $M \times N$ input, linear convolution (or technically cross-correlation) with a filter $\mathbf{W} \in \mathbb{C}^{M\times N\times C\times C}$ takes the form

$$[\mathrm{conv}_{\mathbf{W}}(\mathbf{X})]_{p,q,d} = [\mathbf{W} * \mathbf{X}]_{p,q,d} = \sum_{c=1}^{C}\sum_{m=1}^{M}\sum_{n=1}^{N} \mathbf{W}_{m,n,d,c}\mathbf{X}_{p+m,q+n,c}, \tag{B.10}$$

where the indexing above is assumed to be cyclic (taken modulus the corresponding dimension). Unitary or orthogonal convolutions form a subset of filters $\mathbf{W}$ which preserve the norm, *i.e.,* $\|\operatorname{conv}_{\mathbf{W}}(\mathbf{X})\| = \|\mathbf{X}\|$. Equivalently, $\operatorname{conv}_{\mathbf{W}}(\cdot)$ is orthogonal/unitary if the Jacobian of the transformation is also orthogonal/unitary.

The first method for orthogonalizing convolutions in neural networks was proposed in [72]. Their algorithm showed how to calculate the singular values of a linear convolution operation and then performed a series of projections on a given convolutional filter to project it onto an operator norm ball. Their algorithm made use of the convolution theorem which shows that linear convolution is diagonalized by Fourier transformations. [59] also proposed a method to orthogonalize convolutions by performing an SVD of the linearization of a convolution and then bounding the singular values accordingly. This method outputs linear transformations that are close to a convolution operation but no longer necessarily maintaining the equivariance of a standard convolution. Also, their method is expensive for large matrices as it requires implenting an SVD.

More recent methods implement orthogonal convolutions via approximations to the exponential map [76, 73]. These methods first form convolution filters $\mathbf{W}$ whose Jacobians $\mathbf{J}(\mathbf{W})$ satisfy the skew symmetry property of the orthogonal group:

$$\mathbf{J}(\mathbf{W}) = -\mathbf{J}(\mathbf{W})^{\intercal} \;\equiv\; \mathbf{W} = -\operatorname{conv\text{-}transpose}(\mathbf{W}), \tag{B.11}$$

where $\operatorname{conv\text{-}transpose}$ is the equivalent transposition operation in the filter space defined as [73]

$$[\operatorname{conv\text{-}transpose}(\mathbf{L})]_{m,n,c,d} = \mathbf{L}^{*}_{M-1-m,N-1-n,d,c} \tag{B.12}$$

for a filter $\mathbf{L} \in \mathbb{R}^{M\times N\times D\times C}$.

A given filter $\mathbf{W}$ can be transformed into a skew symmetric convolution filter by simply applying $\mathbf{L} = \mathbf{W} - \operatorname{conv\text{-}transpose}(\mathbf{W})$. As in the general case, one can apply the exponential map to the convolution filter to perform orthogonal convolution.

$$\exp[\operatorname{conv}_{\mathbf{L}}](\mathbf{X}) = \mathbf{X} + \mathbf{L} * \mathbf{X} + \frac{1}{2}\mathbf{L} *^2 \mathbf{X} + \frac{1}{6}\mathbf{L} *^3 \mathbf{X} + \cdots, \tag{B.13}$$

where $*^p$ indicates the convolution is applied p times, *e.g.,* $\mathbf{L} *^2 \mathbf{X} = \mathbf{L} * \mathbf{L} * \mathbf{X}$. Since the above operation is computationally expensive [73] implement a k-th order Taylor approximation to the exponential map:

$$\exp[\operatorname{conv}_{\mathbf{L}}](\mathbf{X}) \approx \sum_{p=0}^{k} \frac{1}{p!}\mathbf{L} *^p \mathbf{X}. \tag{B.14}$$

Given 2-D inputs of dimension $N \times N \times C$, applying convolution with filters with support over W elements in each dimension has runtime $O(pC^2W^2N^2)$. The added factor p in the runtime is required both upon training and evaluation of the network. Though this number is held constant, this adds a multiplicative overhead to running the algorithm both when evaluating an input and when training the algorithm. [73] use the above method to implement orthogonal convolution in their skew orthogonal convolution (SOC) algorithm. Similar techniques have been used to perform invertible (not necessarily orthogonal or unitary) convolutions in [39].

There are two key considerations or drawbacks associated with using the Taylor expansion to perform unitary/orthogonal convolution as in [73]. First, approximations via the Taylor approximation necessarily include an error that can be accounted for by scaling the factor p. However, unlike the Padé approximants discussed earlier such as the Cayley approximant, the Taylor approximation to the exponential map does not return a unitary/orthogonal operator. Thus, the value p needed to bound the error from any unitary/orthogonal operator also scales logarithmically with the desired error; however, this factor can also grow with the size of the filter, the number of channels, size of the input to a convolution operation, and especially the depth of the network. Second, Taylor approximations to a function can significantly bias the gradient. As noted in [56], their expRNN algorithm avoided using the Taylor or Padé approximation in the implementation of the unitary/orthogonal operation for this reason. As an example of how approximations can fail to represent the gradient, they provide the set of functions $f_n(x) = \sin(2\pi nx)/n$ approximating $f = 0$, where $f_n \to f$ uniformly but the derivatives do not converge to zero. Especially when constructing very deep networks, such instabilities can potentially cause issues in training.

Convolution operations over filters with a large support can be performed more efficiently in the Fourier domain as explored in various prior work [64, 11]. Specific to orthogonal convolution, [76] use the 2-D fast Fourier transform to more efficiently perform orthogonal convolution. For single channel inputs and outputs, convolution in the Fourier regime corresponds to pointwise multiplication over Fourier bases. Given multi-channel inputs and outputs, convolution in the Fourier regime corresponds to matrix multiplication over the blocks indexed by channels where each block corresponds to a specific Fourier basis, *i.e.,* for a filter $\mathbf{W}$ and input $\mathbf{X}$:

$$[\operatorname{FFT}\operatorname{conv}_{\mathbf{W}}(\mathbf{X})]_{\widehat{r},\widehat{s},:} = \widehat{\mathbf{W}}^{*}_{\widehat{r},\widehat{s},:,:}\;[\operatorname{FFT}\mathbf{X}]_{\widehat{r},\widehat{s},:}\;, \tag{B.15}$$

where $\widehat{\mathbf{W}}_{\widehat{r},\widehat{s},:,:}$ is the representation of the filter in the Fourier regime. If the filter and and input are flattened in their spatial dimensions, we can represent the operation above as block-diagonal multiplication. For example, one can visually input the Fourier multiplication on an 2-channel input with spatial dimension $[2, 2]$.

$$[\mathrm{FFT}\,\mathrm{conv}_{\mathbf{W}}(\mathbf{X})] = \begin{bmatrix} \widehat{\mathbf{W}}^*_{1,1,1,1} & \widehat{\mathbf{W}}^*_{1,1,1,2} & & & & & & \\ \widehat{\mathbf{W}}^*_{1,1,2,1} & \widehat{\mathbf{W}}^*_{1,1,2,2} & & & & & & \\ & & \widehat{\mathbf{W}}^*_{1,2,1,1} & \widehat{\mathbf{W}}^*_{1,2,1,2} & & & & \\ & & \widehat{\mathbf{W}}^*_{1,2,2,1} & \widehat{\mathbf{W}}^*_{1,2,2,2} & & & & \\ & & & & \widehat{\mathbf{W}}^*_{2,1,1,1} & \widehat{\mathbf{W}}^*_{2,1,1,2} & & \\ & & & & \widehat{\mathbf{W}}^*_{2,1,2,1} & \widehat{\mathbf{W}}^*_{2,1,2,2} & & \\ & & & & & & \widehat{\mathbf{W}}^*_{2,2,1,1} & \widehat{\mathbf{W}}^*_{2,2,1,2} \\ & & & & & & \widehat{\mathbf{W}}^*_{2,2,2,1} & \widehat{\mathbf{W}}^*_{2,2,2,2} \end{bmatrix} \begin{bmatrix} \widehat{\mathbf{X}}_{1,1,1} \\ \widehat{\mathbf{X}}_{1,1,2} \\ \widehat{\mathbf{X}}_{1,2,1} \\ \widehat{\mathbf{X}}_{1,2,2} \\ \widehat{\mathbf{X}}_{2,1,1} \\ \widehat{\mathbf{X}}_{2,1,2} \\ \widehat{\mathbf{X}}_{2,2,1} \\ \widehat{\mathbf{X}}_{2,2,2} \end{bmatrix}$$

[76] use the above technique, parameterizing filters in the Lie algebra of the orthogonal group. Filters are mapped into the orthogonal group using the Cayley transform (see Equation (B.9) for its form). Since the Cayley transform requires matrix inversion over the matrices above, their total runtime scales as $O(N^2C^2\log(N) + N^2C^3)$ operations for convolution over $N \times N$ images with C channels. Our methodology is adapted from the techniques in [76] and scales more efficiently as $O(N^2C\log(N) + kN^2C^2)$ time when the rank of gradient updates $k \ll N$. We instead parameterize the convolution filter directly in the Fourier regime, by parameterizing the individual blocks ($\widehat{\mathbf{W}}_{\widehat{r},\widehat{s},:,:}$) of the block diagonal matrix above. One drawback of our method compared to [76] is that we cannot specify the support of the convolution in the real space of the Lie algebra as we parameterize matrices directly in the Fourier space. Of course, one can perform projections of the convolutions in Fourier space onto the bases spanned by the support of the elements in the Lie algebra; however, we opt instead to implement a regularizer which biases the filter towards elements of the desired space (*e.g.,* local elements).

Finally, [58] performed orthogonal convolution by parameterizing a block of matrices and corresponding projectors in their Block Convolutional Orthogonal Parametrization (BCOP) algorithm. Their method preceded those of [76] and [73]. However, their method has two key drawbacks: it only parameterizes a subset of the space of orthogonal convolutions and is slower than other methods as it adds additional parameters to connect the various components in the space of orthogonal convolution.

Finally, methods have been proposed to iteratively or approximately maintain orthogonality. [9] propose a regularizer for convolutional layers which penalize matrices having singular values far from unity. [40] propose using a Newton's method iteration to update linear transformations to be closer to orthogonal/unitary. Their method requires iterative matrix-matrix multiplication over the unrolled weight matrix which can become expensive for large images.

B.2 Other related works

In the context of recurrent neural networks, designing RNNs to learn long sequences of data has a rich history of study. Some of the first and most celebrated algorithms include the long short-term memory networks (LSTM) [38] and gated reccurent unit (GRU) networks [20]. Since these works, various techniques have been used to more optimally avoid issues with learning long-sequence data. Beyond the unitary RNNs discussed earlier, some work has explored using bi-directional RNNs [33, 53], including those that have a more biologically inspired design [12]. These networks perform well on the copy task.

Another line of research studies the stability properties of continuous state space models which can be converted into a RNN formulation [24]. Some algorithms construct continuous state space models whose attractors are stable points in the dynamical system [19, 25]. More recently, continuous state space models have been designed with hidden state transformations that are customized to memorize data by limiting the learning over time to a subset of orthogonal polynomials [29, 30, 77]. Here, hidden states are in a sense parameterized over a set of polynomial coefficients [77]. These algorithms perform very well on tasks such as the copy task or Permuted MNIST but do not include unitary/orthogonal transformations in their network. In fact, more recent models [30, 77] achieve slightly higher scores on the TIMIT and permuted MNIST benchmarks compared to the unitary RNN formulations studied here.

In the convolutional setting, [83] form very deep convolutional neural networks by initializing parameters to construct norm-preserving orthogonal transformations. Orthogonality is not preserved during training however. [80] bias filters towards orthogonality by implementing a regularizer that penalizes the weights when norms of outputs are larger or smaller than norms of inputs. The networks used in [83, 80] do not necessarily preserve the orthogonality property of their convolutional transformations during training. [45] study orthogonal convolutions from the basis of unitary/orthogonal wavelets showing how to represent convolution in terms of these wavelets.

Low rank approximations have been used in prior work in deep learning to prune neural network models [84, 74] and accelerate convolutions [43, 75, 42]. More related to our work, recent research has compressed gradients efficiently using low rank compression methods [78]. Although their focus was in sharing compressed information across computing units, their work lends support to the notion that the information of a gradient can be effectively and efficiently stored in low rank components. Research in learning theory has also noted connections between the stable rank of neural network parameters and generalization. [7] prove a generalization bound by compressing the models in the hypothesis class based on the stable rank of individual layers. [63] study the phases of learning from a random matrix theory setting showing that the stable rank of a matrix tends to decay over training. Various works have studied the implicit bias induced by optimization algorithms such as gradient descent showing that in many cases, the implicit bias is towards low rank solutions [31, 21]. Such bias towards low rank solutions has been explicitly proven in the setting of 2-layer matrix factorization [60], deep matrix factorization [6], linear group convolutional networks [54], and matrix recovery from Pauli measurements [62]. We note that relating low-rankness to the generalization ability of learning algorithms is a richly studied topic, and there are numerous papers that we did not mention here.

Finally, a wide range of work in quantum computation and quantum machine learning studies unitary learning algorithms in the context of quantum systems [13]. In fact, since the state space of closed quantum systems is transformed by unitary operators, quantum computers offer a unique platform for performing machine learning on the unitary manifold. This is an active area of research and existing methods for performing quantum machine learning on quantum architectures include variational algorithms which parameterize a quantum circuit and update the parameters via classical optimization methods [17, 81, 85, 48, 49, 79], quantum neural networks which design analogues to classical deep neural networks [51, 10, 71], and direct implementations of classical deep learning algorithms on quantum architectures [47, 16, 3]. We stress that these quantum algorithms are inherently different in nature than their classical counterparts and there still exist significant challenges that must be surmounted before they become practically feasible [17, 13, 65]. For example, the loss landscapes of quantum algorithms often have many more poor local minima in comparison to classical counterparts [50, 4] and training of quantum architectures requires sampling of outputs which is challenging when derivatives decay with the size of a model – a phenomenon described as “barren plateaus" [65, 18]. Furthermore, even if algorithms can be efficiently run on a quantum computer, preparing data for use in a quantum computer and reading out information from the quantum computer are challenging tasks which are not guaranteed to be efficient [1].

C Deferred proofs

C.1 Proof of Theorem 4.2

Recall Theorem 4.2:

Theorem 4.2 (Low rank unitary projection)**.** *Let $\boldsymbol{U}$ be an $n \times n$ orthogonal/unitary matrix perturbed by $\boldsymbol{G}_k$, a rank k matrix. Then the projection onto the closest orthogonal/unitary matrix defined below can be performed in $O(k(n^2 + nk + k^2))$ steps.*

$$\boldsymbol{U} + \boldsymbol{G}_k \to \underset{\boldsymbol{V} \in \mathcal{U}}{\arg\min} \|\boldsymbol{U} + \boldsymbol{G}_k - \boldsymbol{V}\|_F^2. \tag{4}$$

Proof. We proceed to prove the above statement by first analyzing the case where $k = 1$ and then generalizing to higher rank k. Recall from Lemma 4.1 that we would like to perform the following update:

$$\boldsymbol{U} + \boldsymbol{G}_k \to \underset{\boldsymbol{V} \in \mathcal{U}}{\arg\min} \|(\boldsymbol{U} + \boldsymbol{G}_k) - \boldsymbol{V}\|_F^2 = (\boldsymbol{U} + \boldsymbol{G}_k) \left[(\boldsymbol{U} + \boldsymbol{G}_k)^\dagger (\boldsymbol{U} + \boldsymbol{G}_k)\right]^{-\frac{1}{2}}. \tag{C.1}$$

Let the rank one vector components of $\boldsymbol{G}_k = \boldsymbol{a}\boldsymbol{b}^\dagger$ and define

$$\tilde{\boldsymbol{M}} = \boldsymbol{U} + \boldsymbol{G}_k = \boldsymbol{U} + \boldsymbol{a}\boldsymbol{b}^\dagger. \tag{C.2}$$

With the above, we can rewrite $(\tilde{\boldsymbol{M}}^\dagger \tilde{\boldsymbol{M}})^{-\frac{1}{2}}$ in Equation (C.1) as:

$$\begin{aligned}(\tilde{\boldsymbol{M}}^\dagger \tilde{\boldsymbol{M}})^{-\frac{1}{2}} &= [(\boldsymbol{U} + \boldsymbol{a}\boldsymbol{b}^\dagger)^\dagger (\boldsymbol{U} + \boldsymbol{a}\boldsymbol{b}^\dagger)]^{-\frac{1}{2}} \\ &= [\boldsymbol{I} + \hat{\boldsymbol{a}}\boldsymbol{b}^\dagger + \boldsymbol{b}\hat{\boldsymbol{a}}^\dagger + c_a \boldsymbol{b}\boldsymbol{b}^\dagger]^{-\frac{1}{2}},\end{aligned} \tag{C.3}$$

where $\hat{\boldsymbol{a}} = \tilde{\boldsymbol{U}}^\dagger \boldsymbol{a}$ and $c_a = \boldsymbol{a}^\dagger \boldsymbol{a}$.

Equation (C.3) is the Identity matrix plus the update of a rank two matrix. To see this, we decompose $\hat{\boldsymbol{a}}$ and $\boldsymbol{b}$ into orthogonal components using Gram Schmidt:

$$\boldsymbol{v}_1 = \frac{\boldsymbol{b}}{\|\boldsymbol{b}\|} \quad \boldsymbol{v}_2 = \frac{\hat{\boldsymbol{a}} - (\boldsymbol{v}_1^\dagger \hat{\boldsymbol{a}})\hat{\boldsymbol{a}}}{\|\hat{\boldsymbol{a}} - (\boldsymbol{v}_1^\dagger \hat{\boldsymbol{a}})\hat{\boldsymbol{a}}\|}. \tag{C.4}$$

In this new basis:

$$\hat{\boldsymbol{a}} = a_1 \boldsymbol{v}_1 + a_2 \boldsymbol{v}_2 \quad \boldsymbol{b} = b_1 \boldsymbol{v}_1, \tag{C.5}$$

and

$$\begin{aligned} &\tilde{\boldsymbol{M}}^\dagger \tilde{\boldsymbol{M}} = \\ &\boldsymbol{I} + [\boldsymbol{v}_1 \quad \boldsymbol{v}_2] \begin{bmatrix} a_1 b_1^* + b_1 a_1^* + c_a b_1 b_1^* & b_1 a_2^* \\ a_2 b_1^* & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_1^\dagger \\ \boldsymbol{v}_2^\dagger \end{bmatrix}. \end{aligned} \tag{C.6}$$

Performing an eigendecomposition of the above 2×2 matrix into a diagonal eigenvalue matrix $\boldsymbol{S}$ and eigenvector matrix $\boldsymbol{C}$, we can rewrite $\tilde{\boldsymbol{M}}^\dagger \tilde{\boldsymbol{M}}$ in a convenient form:

$$\begin{aligned} \tilde{\boldsymbol{M}}^\dagger \tilde{\boldsymbol{M}} &= \boldsymbol{I} + [\boldsymbol{v}_1 \quad \boldsymbol{v}_2] \, \boldsymbol{C} \begin{bmatrix} s_1 & 0 \\ 0 & s_2 \end{bmatrix} \boldsymbol{C}^\dagger \begin{bmatrix} \boldsymbol{v}_1^\dagger \\ \boldsymbol{v}_2^\dagger \end{bmatrix} \\ &= \boldsymbol{I} + s_1 \boldsymbol{u}_1 \boldsymbol{u}_1^\dagger + s_2 \boldsymbol{u}_2 \boldsymbol{u}_2^\dagger. \end{aligned} \tag{C.7}$$

Taking the inverse square root of the above can be performed by manipulating singular values:

$$\begin{aligned} (\tilde{\boldsymbol{M}}^\dagger \tilde{\boldsymbol{M}})^{-\frac{1}{2}} = \boldsymbol{I} &+ ((s_1 + 1)^{-\frac{1}{2}} - 1) \boldsymbol{u}_1 \boldsymbol{u}_1^\dagger \\ &+ ((s_2 + 1)^{-\frac{1}{2}} - 1) \boldsymbol{u}_2 \boldsymbol{u}_2^\dagger. \end{aligned} \tag{C.8}$$

Finally, we multiply the above on the left by $\tilde{\boldsymbol{M}}$:

$$\begin{aligned} \tilde{\boldsymbol{M}} (\tilde{\boldsymbol{M}}^\dagger \tilde{\boldsymbol{M}})^{-\frac{1}{2}} = \tilde{\boldsymbol{M}} &+ ((s_1 + 1)^{-\frac{1}{2}} - 1) \tilde{\boldsymbol{M}} \boldsymbol{u}_1 \boldsymbol{u}_1^\dagger \\ &+ ((s_2 + 1)^{-\frac{1}{2}} - 1) \tilde{\boldsymbol{M}} \boldsymbol{u}_2 \boldsymbol{u}_2^\dagger. \end{aligned} \tag{C.9}$$

The above requires performing an eigendecomposition of a 2×2 matrix and a series of matrix-vector multiplication, matrix additions, and vector-vector outer products – in total scaling as $O(n^2)$ time.

Rank k updates Note that the above method can be extended to low rank updates, running in $O(kn^2)$ time when the rank of the update $k \ll n$. Specifically, now our update is:

$$\tilde{\boldsymbol{M}} = \boldsymbol{U} + \sum_{i=1}^{k} \boldsymbol{a}_i \boldsymbol{b}_i^\dagger. \tag{C.10}$$

Following the same steps would ultimately require performing an eigendecomposition of a $2k \times 2k$ matrix which takes $O(k^3)$ time. Additionally, one must perform Gram-Schmidt decomposition on a set of k vectors of length n which takes $O(k^2 n)$ time. Finally, a series of $O(k)$ matrix-vector multiplications and vector-vector outer products is performed resulting in a total runtime of $O(k(n^2 + nk + k^2))$ time. In cases where $k \ll n$, the time to perform the Gram-Schmidt decomposition and the eigendecomposition is negligible and an overall runtime of $O(kn^2)$ time is achieved. Even in cases where updates are not low rank, one can apply efficient sampling procedures (see Section 4.3) to find low rank approximations to the update matrix and apply the methods above while maintaining runtimes.

□

C.2 Proof of Theorem 4.4

Recall Theorem 4.4:

Theorem 4.4 (Low rank tangent transport)**.** *Let $\boldsymbol{U}$ be an $n \times n$ orthogonal/unitary matrix perturbed by $\boldsymbol{G}_k$, a rank k matrix. Then projecting $\boldsymbol{G}_k$ onto the tangent space and performing a rotation in that direction as defined in Equation* (7) *can be performed in $O(k(n^2 + nk + k^2))$ steps.*

Proof. We would like to efficiently perform the update below:

$$\boldsymbol{U} \to \boldsymbol{U} \exp\left[-\eta \boldsymbol{U}^\dagger \Pi_{T_{\boldsymbol{U}}}(\boldsymbol{G}_k)\right]. \tag{C.11}$$

As before, we proceed to prove the above statement by first analyzing the case where $k = 1$ and then generalizing to higher rank k.

Let the rank one vector components of $\boldsymbol{G}_k = \boldsymbol{a}\boldsymbol{b}^\dagger$. Then

$$\boldsymbol{U}^\dagger \Pi_{T_{\boldsymbol{U}}}(\boldsymbol{G}_k) = \frac{1}{2}\boldsymbol{U}^\dagger \left(\boldsymbol{G}_k - \boldsymbol{U}\boldsymbol{G}_k^\dagger \boldsymbol{U}\right) = \frac{1}{2}\left(\boldsymbol{U}^\dagger \boldsymbol{a}\boldsymbol{b}^\dagger - \boldsymbol{b}\boldsymbol{a}^\dagger \boldsymbol{U}\right) = \frac{1}{2}\left(\hat{\boldsymbol{a}}\boldsymbol{b}^\dagger - \boldsymbol{b}\hat{\boldsymbol{a}}^\dagger\right), \tag{C.12}$$

where $\hat{\boldsymbol{a}} = \boldsymbol{U}^\dagger \boldsymbol{a}$. The above is a rank 2 matrix. As before, we now proceed to perform an eigendecomposition in the low rank subspace of the above matrix. Using Gram Schmidt, we have

$$\boldsymbol{v}_1 = \frac{\boldsymbol{b}}{\|\boldsymbol{b}\|} \quad \boldsymbol{v}_2 = \frac{\hat{\boldsymbol{a}} - (\boldsymbol{v}_1^\dagger \hat{\boldsymbol{a}})\hat{\boldsymbol{a}}}{\|\hat{\boldsymbol{a}} - (\boldsymbol{v}_1^\dagger \hat{\boldsymbol{a}})\hat{\boldsymbol{a}}\|}. \tag{C.13}$$

In this new basis:

$$\hat{\boldsymbol{a}} = a_1 \boldsymbol{v}_1 + a_2 \boldsymbol{v}_2 \quad \boldsymbol{b} = b_1 \boldsymbol{v}_1, \tag{C.14}$$

and

$$\boldsymbol{U}^\dagger \Pi_{T_{\boldsymbol{U}}}(\boldsymbol{G}_k) = \frac{1}{2}\begin{bmatrix}\boldsymbol{v}_1 & \boldsymbol{v}_2\end{bmatrix}\begin{bmatrix} a_1 b_1^* - b_1 a_1^* & -b_1 a_2^* \\ a_2 b_1^* & 0 \end{bmatrix}\begin{bmatrix}\boldsymbol{v}_1^\dagger \\ \boldsymbol{v}_2^\dagger\end{bmatrix}. \tag{C.15}$$

Performing an eigendecomposition of the above 2×2 matrix into a diagonal eigenvalue matrix $\boldsymbol{S}$ and eigenvector matrix $\boldsymbol{C}$, we can rewrite $\tilde{\boldsymbol{M}}^\dagger \tilde{\boldsymbol{M}}$ in a convenient form:

$$\begin{aligned} \boldsymbol{U}^\dagger \Pi_{T_{\boldsymbol{U}}}(\boldsymbol{G}_k) &= \begin{bmatrix}\boldsymbol{v}_1 & \boldsymbol{v}_2\end{bmatrix} \boldsymbol{C} \begin{bmatrix} s_1 & 0 \\ 0 & s_2 \end{bmatrix} \boldsymbol{C}^\dagger \begin{bmatrix}\boldsymbol{v}_1^\dagger \\ \boldsymbol{v}_2^\dagger\end{bmatrix} \\ &= s_1 \boldsymbol{u}_1 \boldsymbol{u}_1^\dagger + s_2 \boldsymbol{u}_2 \boldsymbol{u}_2^\dagger. \end{aligned} \tag{C.16}$$

We apply the exponential map scaled by η to obtain

$$\begin{aligned} \exp\left[-\eta \boldsymbol{U}^\dagger \Pi_{T_{\boldsymbol{U}}}(\boldsymbol{G}_k)\right] = \boldsymbol{I} &+ \left(\exp(-\eta s_1) - 1\right) \boldsymbol{u}_1 \boldsymbol{u}_1^\dagger \\ &+ \left(\exp(-\eta s_2) - 1\right) \boldsymbol{u}_2 \boldsymbol{u}_2^\dagger. \end{aligned} \tag{C.17}$$

Finally, we multiply the above on the left by $\boldsymbol{U}$ to obtain the final result

$$\boldsymbol{U} \exp\left[-\eta \boldsymbol{U}^\dagger \Pi_{T_{\boldsymbol{U}}}(\boldsymbol{G}_k)\right] = \boldsymbol{U} + \left(\exp(-\eta s_1) - 1\right) \boldsymbol{U}\boldsymbol{u}_1 \boldsymbol{u}_1^\dagger + \left(\exp(-\eta s_2) - 1\right) \boldsymbol{U}\boldsymbol{u}_2 \boldsymbol{u}_2^\dagger. \tag{C.18}$$

The above requires performing an eigendecomposition of a 2×2 matrix and a series of matrix-vector multiplication, matrix additions, and vector-vector outer products – in total scaling as $O(n^2)$ time.

Rank k updates Note that as before, the above method can be extended to low rank updates, running in $O(kn^2)$ time when the rank of the update $k \ll n$. Specifically, now our update is:

$$\boldsymbol{U} \to \boldsymbol{U} \exp\left[-\eta \boldsymbol{U}^\dagger \Pi_{T_{\boldsymbol{U}}}\left(\sum_{i=1}^{k} \boldsymbol{a}_i \boldsymbol{b}_i^\dagger\right)\right]. \tag{C.19}$$

Following the same steps, the projection onto the tangent space would be a rank $2k$ matrix. The steps that follow would ultimately require performing an eigendecomposition of a $2k \times 2k$ matrix which takes $O(k^3)$ time. Additionally, one

must perform Gram-Schmidt decomposition on a set of k vectors of length n which takes $O(k^2 n)$ time. Finally, a series of $O(k)$ matrix-vector multiplications and vector-vector outer products is performed resulting in a total runtime of $O(k(n^2 + nk + k^2))$ time. In cases where $k \ll n$, the time to perform the Gram-Schmidt decomposition and the eigendecomposition is negligible and an overall runtime of $O(kn^2)$ time is achieved. Even in cases where updates are not low rank, one can apply efficient sampling procedures (see Section 4.3) to find low rank approximations to the update matrix and apply the methods above while maintaining runtimes.

□

C.3 First order equivalence to PROJUNN-D

Though PROJUNN-D and PROJUNN-T perform different updates, we can show that up to first order, the updates are in fact equivalent. Furthermore, as one may expect, this first order update is equal to the projection of the gradient onto the tangent space (see Lemma 4.3). In the case of PROJUNN-D, this shows that the update step is a retraction or first order approximation to the matrix exponential implemented in PROJUNN-T [15].

Proposition C.1 (First order equivalence)**.** *For an $n \times n$ unitary/orthogonal matrix $\boldsymbol{U}$ perturbed by $\Delta\boldsymbol{U}$, gradient updates applied by algorithms* PROJUNN-D *(Equation* (4)*) and* PROJUNN-T *(Equation* (7)*) are, up to first order, equal to $\boldsymbol{U} + \Pi_{T_{\boldsymbol{U}}}(\Delta\boldsymbol{U})$, i.e.,*

$$\boldsymbol{U} \to \boldsymbol{U} + \frac{1}{2}\left(\Delta\boldsymbol{U} - \boldsymbol{U}\Delta\boldsymbol{U}^\dagger\boldsymbol{U}\right) + O(\Delta\boldsymbol{U}\Delta\boldsymbol{U}^\dagger).$$

Proof. We first show that the above formula holds for PROJUNN-D and then show the same for PROJUNN-T. Recall the update formula for PROJUNN-D:

$$\boldsymbol{U} + \Delta\boldsymbol{U} \to \underset{\boldsymbol{V}\in\mathcal{U}}{\arg\min}\, \|(\boldsymbol{U} + \Delta\boldsymbol{U}) - \boldsymbol{V}\|_F^2 = (\boldsymbol{U} + \Delta\boldsymbol{U})\left[(\boldsymbol{U} + \Delta\boldsymbol{U})^\dagger(\boldsymbol{U} + \Delta\boldsymbol{U})\right]^{-\frac{1}{2}}. \tag{C.20}$$

Expanding the above up to first order and applying the first order Taylor expansion of $(\cdot)^{-1/2}$, we have:

$$\begin{aligned}(\boldsymbol{U} + \Delta\boldsymbol{U})\left[(\boldsymbol{U} + \Delta\boldsymbol{U})^\dagger(\boldsymbol{U} + \Delta\boldsymbol{U})\right]^{-\frac{1}{2}} &= (\boldsymbol{U} + \Delta\boldsymbol{U})\left[\boldsymbol{I} + \boldsymbol{U}^\dagger\Delta\boldsymbol{U} + \Delta\boldsymbol{U}^\dagger\boldsymbol{U} + O(\Delta\boldsymbol{U}^\dagger\Delta\boldsymbol{U})\right]^{-\frac{1}{2}} \\ &= (\boldsymbol{U} + \Delta\boldsymbol{U})\left[\boldsymbol{I} - \frac{1}{2}\left(\boldsymbol{U}^\dagger\Delta\boldsymbol{U} + \Delta\boldsymbol{U}^\dagger\boldsymbol{U}\right) + O(\Delta\boldsymbol{U}^\dagger\Delta\boldsymbol{U})\right] \\ &= \boldsymbol{U} + \frac{1}{2}\left(\Delta\boldsymbol{U} - \boldsymbol{U}\Delta\boldsymbol{U}^\dagger\boldsymbol{U}\right) + O(\Delta\boldsymbol{U}^\dagger\Delta\boldsymbol{U}).\end{aligned} \tag{C.21}$$

Similarly, for PROJUNN-T, recall the update formula (ignoring $-\eta$ term for learning rate):

$$\boldsymbol{U} \to \boldsymbol{U}\exp\left[\boldsymbol{U}^\dagger\Pi_{T_{\boldsymbol{U}}}(\Delta\boldsymbol{U})\right]. \tag{C.22}$$

Expanding the above and applying the first order approximation of the matrix exponential,

$$\begin{aligned}\boldsymbol{U}\exp\left[\boldsymbol{U}^\dagger\Pi_{T_{\boldsymbol{U}}}(\Delta\boldsymbol{U})\right] &= \boldsymbol{U}\exp\left[-\eta\boldsymbol{U}^\dagger\frac{1}{2}\left(\Delta\boldsymbol{U} - \boldsymbol{U}\Delta\boldsymbol{U}^\dagger\boldsymbol{U}\right)\right] \\ &= \boldsymbol{U}\left[\boldsymbol{I} + \frac{1}{2}\left(\Delta\boldsymbol{U} - \boldsymbol{U}\Delta\boldsymbol{U}^\dagger\boldsymbol{U}\right) + O(\Delta\boldsymbol{U}^\dagger\Delta\boldsymbol{U})\right] \\ &= \boldsymbol{U} + \frac{1}{2}\left(\Delta\boldsymbol{U} - \boldsymbol{U}\Delta\boldsymbol{U}^\dagger\boldsymbol{U}\right) + O(\Delta\boldsymbol{U}^\dagger\Delta\boldsymbol{U}).\end{aligned} \tag{C.23}$$

□

C.4 Unitary convolutional manifold is connected

Before proceeding to prove that the space of unitary convolutions is connected, we first provide a version of the convolution theorem which shows that convolution in the Fourier domain corresponds to block multiplication over channels. This is a classic result also contained in various prior works [72, 76], and we provide a short proof here for completeness.

Lemma C.2. *Given convolution filter $\mathbf{W} \in \mathbb{C}^{M\times N\times C\times C}$ and input $\mathbf{X} \in \mathbb{C}^{M\times N\times C}$, recall the definition of cyclic convolution (or technically cross-correlation) of $\mathbf{W}$ and $\mathbf{X}$:*

$$[\mathrm{conv}_{\mathbf{W}}(\mathbf{X})]_{p,q,d} = \sum_{c=1}^{C}\sum_{m=1}^{M}\sum_{n=1}^{N} \mathbf{W}_{m,n,d,c}\mathbf{X}_{p+m,q+n,c}. \tag{C.24}$$

Let FFT *be the two dimensional fast Fourier transform, then convolution in the Fourier domain corresponds to block-wise multiplication over channels:*

$$[\mathrm{FFT}\,\mathrm{conv}_{\mathbf{W}}(\mathbf{X})]_{\widehat{r},\widehat{s},:} = \widehat{\mathbf{W}}^{*}_{\widehat{r},\widehat{s},:,:}\ [\mathrm{FFT}\,\mathbf{X}]_{\widehat{r},\widehat{s},:}\,. \tag{C.25}$$

Proof. Let the roots of unity be denoted as $\omega_M = e^{2\pi i/M}$ and $\omega_N = e^{2\pi i/N}$. Then,

$$\begin{aligned}
[\mathrm{FFT}\,\mathrm{conv}_{\mathbf{W}}(\mathbf{X})]_{\widehat{r},\widehat{s},d} &= \sum_{u=1}^{M}\sum_{v=1}^{N}\omega_M^{u\widehat{r}}\omega_M^{v\widehat{s}}\sum_{m=1}^{M}\sum_{n=1}^{N}\sum_{c=1}^{C}\mathbf{W}_{m,n,d,c}\mathbf{X}_{u+m,v+n,c}\\
&= \sum_{u=1}^{M}\sum_{v=1}^{N}\sum_{m=1}^{M}\sum_{n=1}^{N}\sum_{c=1}^{C}\omega_M^{u\widehat{r}}\omega_M^{v\widehat{s}}\omega_M^{\widehat{r}m}\omega_M^{-\widehat{r}m}\omega_N^{\widehat{s}n}\omega_N^{-\widehat{s}n}\mathbf{W}_{m,n,d,c}\mathbf{X}_{u+m,v+n,c}\\
&= \sum_{c=1}^{C}\sum_{m=1}^{M}\sum_{n=1}^{N}\omega_M^{-\widehat{r}m}\omega_N^{-\widehat{s}n}\mathbf{W}_{m,n,d,c}\sum_{u=1}^{M}\sum_{v=1}^{N}\omega_M^{\widehat{r}(u+m)}\omega_M^{\widehat{s}(v+n)}\mathbf{X}_{u+m,v+n,c}\mathbf{X}_{u+m,v+n,c}\\
&= \sum_{c=1}^{C}\widehat{\mathbf{W}}^{*}_{\widehat{r},\widehat{s},d,c}\widehat{\mathbf{X}}_{\widehat{r},\widehat{s},c}.
\end{aligned} \tag{C.26}$$

□

As an aside, the complex conjugation of the filter above is due to the fact that convolution in neural networks corresponds to the more commonly used term cross-correlation in mathematics. If the convolution operation were redefined as what is more commonly known as convolution in mathematics, *i.e.,* define conv' as $[\mathrm{conv}'_{\mathbf{W}}(\mathbf{X})]_{p,q,d} = \sum_{c=1}^{C}\sum_{m=1}^{M}\sum_{n=1}^{N}\mathbf{W}_{m,n,d,c}\mathbf{X}_{p-m,q-n,c}$, then the complex conjugate would no longer appear on the filter term.

From here, we simply show that each block in the above is connected which allows us to prove that the unitary manifold is connected.

Theorem 4.5 (Unitary convolutional manifold is connected)**.** *The space of unitary convolutions with filters of full support has a single connected component.*

Proof. Since the fourier transform is unitary and invertible, we can represent every filter $\mathbf{W}$ in the fourier domain as $\widehat{\mathbf{W}} \in \mathbb{C}^{M\times N\times C\times C}$ and vice-versa. In the Fourier domain, given Lemma C.2, we have that convolution corresponds to block-wise multiplication over channels. For unitary convolution, each block $\widehat{W}_{\widehat{r},\widehat{s},:,:}$ indexed by frequencies $\widehat{r}$ and $\widehat{s}$ must be a unitary matrix, so we now analyze the set of filters whose blocks are unitary matrices in the Fourier domain.

The space of unitary matrices $U(C)$ is connected [32]. Therefore for every block, there exists a connected path between any two unitary matrices $\widehat{W}^{(1)}_{\widehat{r},\widehat{s},:,:}, \widehat{W}^{(2)}_{\widehat{r},\widehat{s},:,:} \in U(C)$. The full space of unitary convolutions is parameterized by the MN times direct product of groups $U(C)$, *i.e.,* $U(C)^{(\times MN)}$. Since the direct product of finitely many connected spaces is also connected, then the space of unitary convolutions is connected. □

D Analysis on various benchmarked tasks

D.1 Learning random unitary

In addition to the analysis shown in the main text, here we include a plot showing the value of the Frobenius error as a function of the runtime (Figure 6). Using PROJUNN-T to peform learning via low rank approximations to the gradient significantly speed up learning. Optimization was performed with vanilla gradient descent over batch sizes of 16 with learning rates of 0.5 and 0.33 for PROJUNN-T and PROJUNN-D respectively. Since the learning rate was fixed across

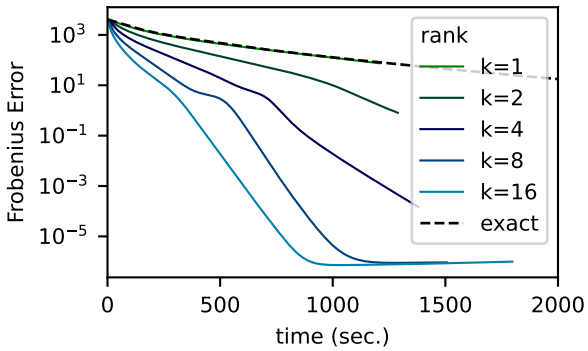


Figure 6: Runtime of PROJUNN-T in learning a random target unitary matrix is faster when using low rank approximations. Here we plot Frobenius error $\|\boldsymbol{U} - \boldsymbol{U}_{tar}\|_F^2$ over the course of optimization. The learning rate is fixed for each value of k.

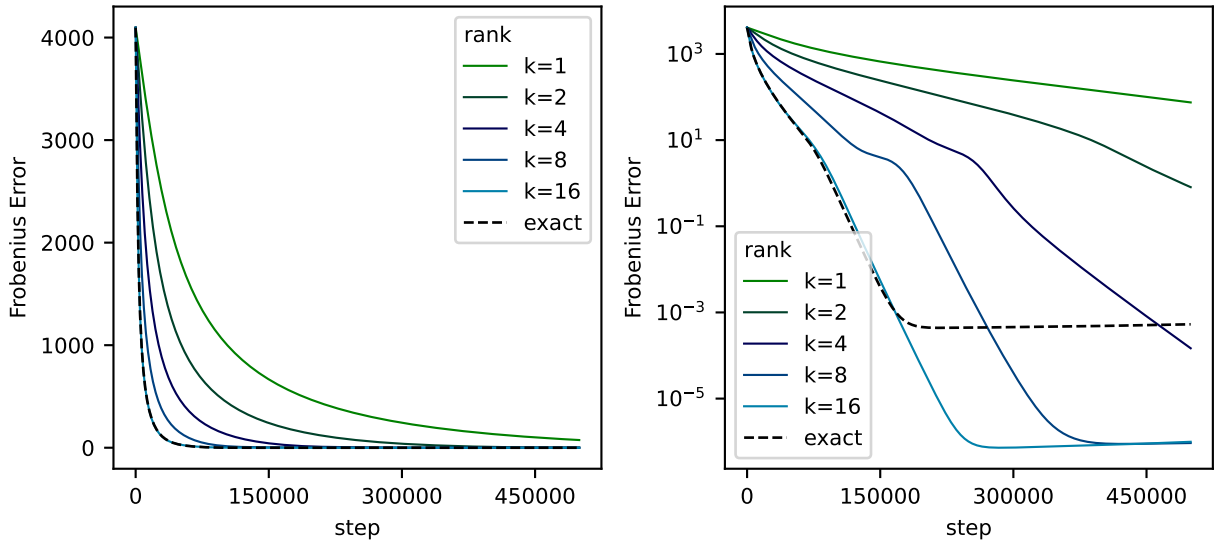


Figure 7: Learning trajectory of PROJUNN-T equipped with the column sampling approximation in the random unitary learning task.

all k for each of these experiments, the norm of the update was smaller for lower values of k. Scaling up the learning rate based on the value of k could make runtimes even faster for lower values of k.

For sake of completeness, we also include plots (in both regular and logarithmic scaling axis) showing the learning trajectory of the various combinations of samplers and PROJUNN architectures in the random unitary task. See Figure 7 for PROJUNN-T with column sampling (left hand side repeated from main text), Figure 8 for PROJUNN-D with column sampling, Figure 9 for PROJUNN-T with LSI sampling, and Figure 10 for PROJUNN-D with LSI sampling.

Figure 8 and Figure 10 also highlight potential instabilities in training PROJUNN-D over long periods of time, a feature noted in the main text and discussed further in Appendix G.3. To alleviate this, for PROJUNN-D architectures, we set the learning rate to slightly lower at 0.33 and projected parameter matrices onto the closest unitary using Lemma 4.1 every 2048 steps. Note, that since this projection step was performed sparingly every n steps (where n is the dimension of the matrix), we did not find any significant decrease in runtime.

D.2 Adding task

As shown in the main text, our PROJUNN is able to learn the adding task even for long sequence lengths. Rank k gradient approximations were obtained using the column sampling approximation. RMSprop was used to train

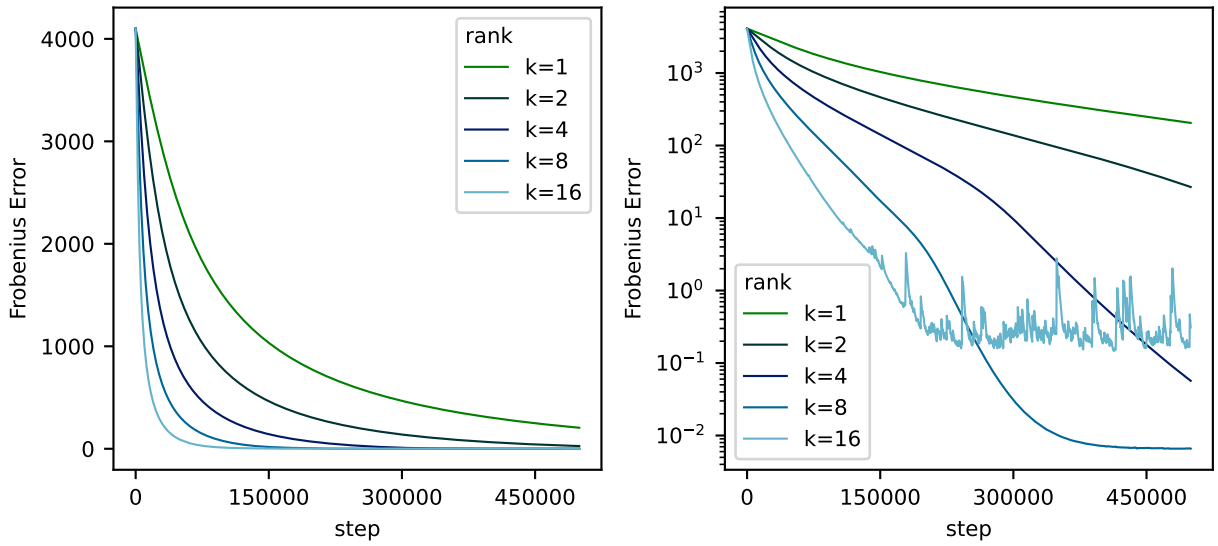


Figure 8: Learning trajectory of PROJUNN-D equipped with the column sampling approximation in the random unitary learning task. Performing gradient updates using exact formulas was too computationally expensive for PROJUNN-D and thus not included here.

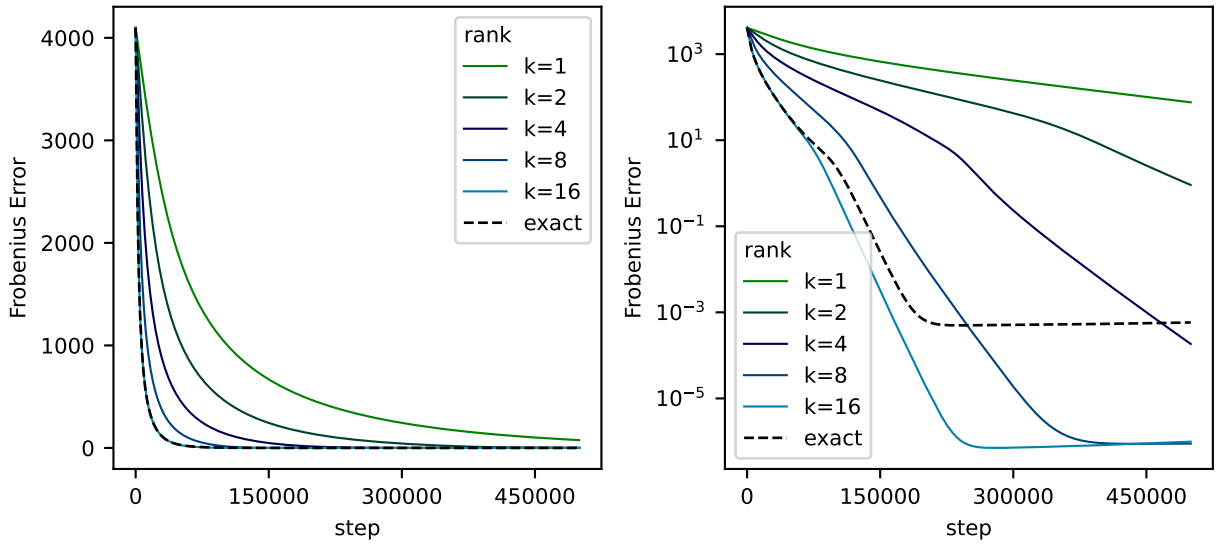


Figure 9: Learning trajectory of PROJUNN-T equipped with the LSI sampling approximation in the random unitary learning task.

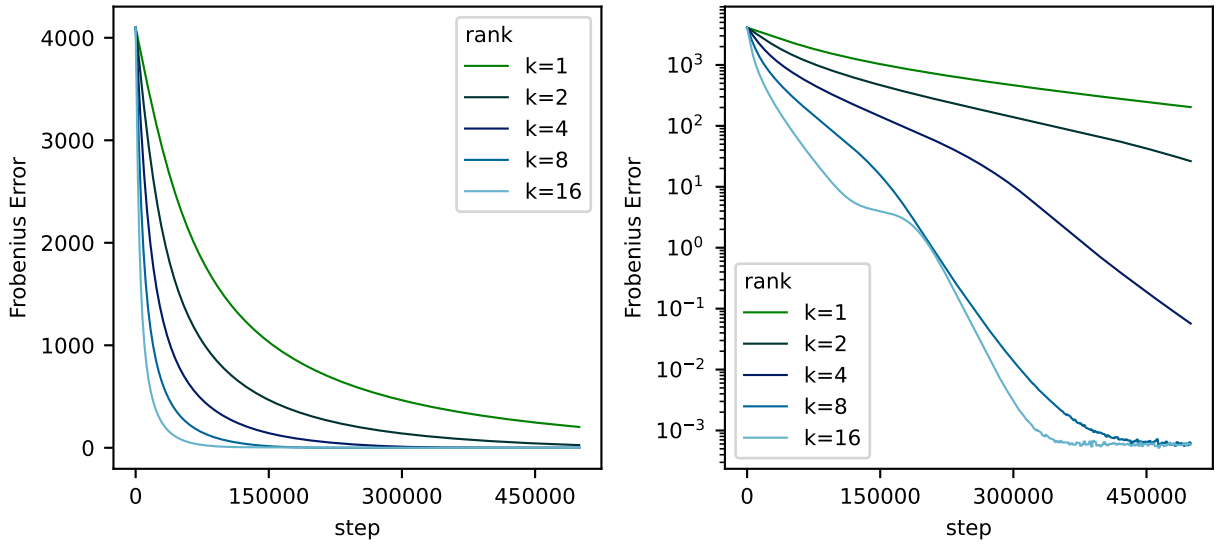


Figure 10: Learning trajectory of PROJUNN-D equipped with the LSI sampling approximation in the random unitary learning task. Performing gradient updates using exact formulas was too computationally expensive for PROJUNN-D and thus not included here.

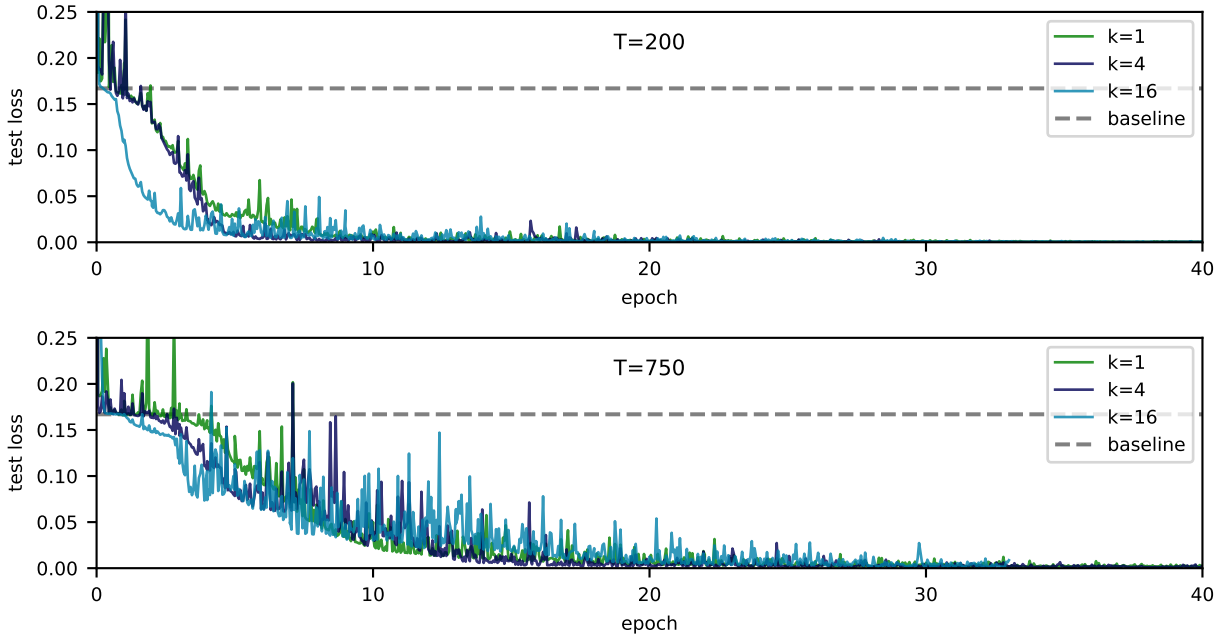


Figure 11: PROJUNN-T (with column sampling approximation) is effective at learning the adding task. Above is a copy of Figure 3 except the plot here is expanded in size and test error is not smoothed.

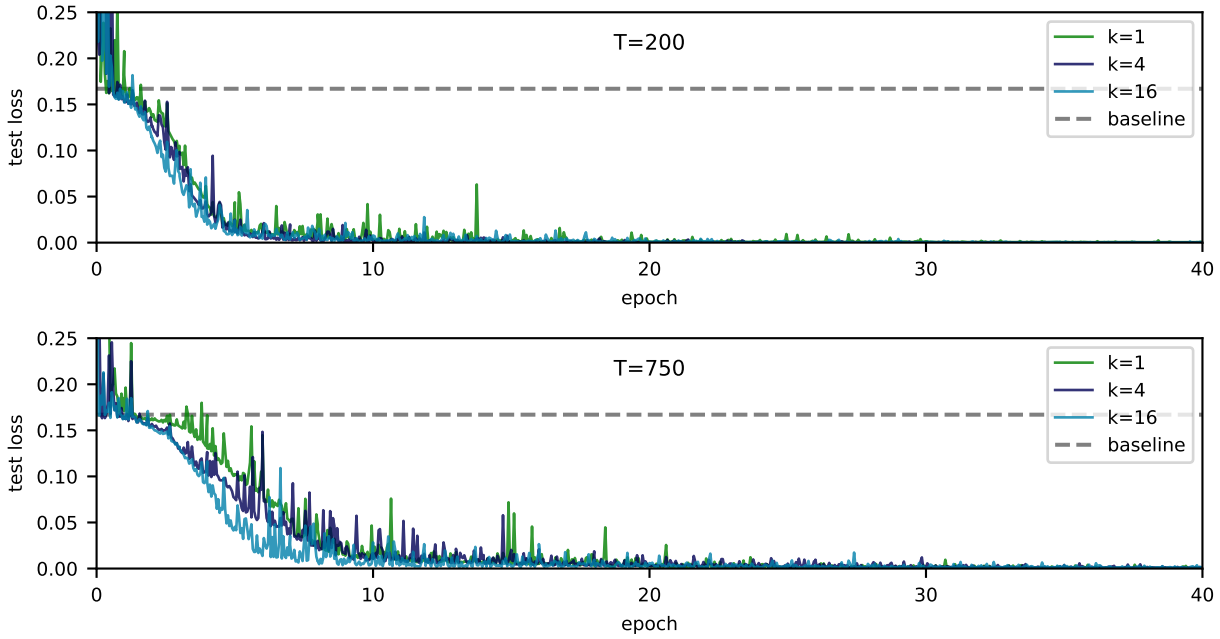


Figure 12: PROJUNN-T (with LSI sampling approximation) is effective at learning the adding task.

Table 3: Test accuracy on CIFAR10 with different unitary convolution parameterizations and our proposed PROJUNN algorithm. We stress that SOC is an approximate unitary parameterization.

	Convolution Type					
Model	Standard	BCOP	SOC	Cayley	Proj-D	Proj-T
Resnet9	92.26	80.72	-	81.70	80.75	**82.06**
Resnet18	95.10	92.38	**94.24**	-	89.43	89.59

the RNN in this task. The learning rate was initialized to 0.001 for non-orthogonal parameters and $0.001/32$ for all orthogonal parameters. Each epoch, the learning rate was reduced by multiplying it by 0.96. We found that initializing the orthogonal matrix as the identity matrix worked well for this task. This is in distinction with *e.g.,* [35] which initialized using the Cayley initialization (block diagonal). For reference, we also include here an enhanced version of the plot in the main text in Figure 11 and a similar plot obtained using PROJUNN with LSI sampling in Figure 12.

D.3 Copy task

Figure 4 in the main text shows that our PROJUNN-T can efficiently learn the copy task. For this task, we set the learning rate of the RMSprop optimizer to $7e-4$. For orthogonal parameters, this learning rate was divided by 32. Orthogonal matrices were intialized using Henaff initialization [36] as described in Appendix G.4. We found that this initialization scheme worked best in comparison to other methods.

D.4 Pixel permuted MNIST

Figure 13 charts the trajectory of learning on the permuted MNIST task. As is evident in the figure, rank $k = 1$ is sufficient to guarantee optimal or nearly optimal convergence in all settings.

D.5 CIFAR10 CNN experiments

To explore the performance of our PROJUNN training algorithm for convolutional layers, we analyze its performance on CIFAR10 classification. Here, we provide further details to the results in the main text as well as some further preliminary experiments on unitary CNNs in resnet architectures. As in prior work [76] we employ the usual data-

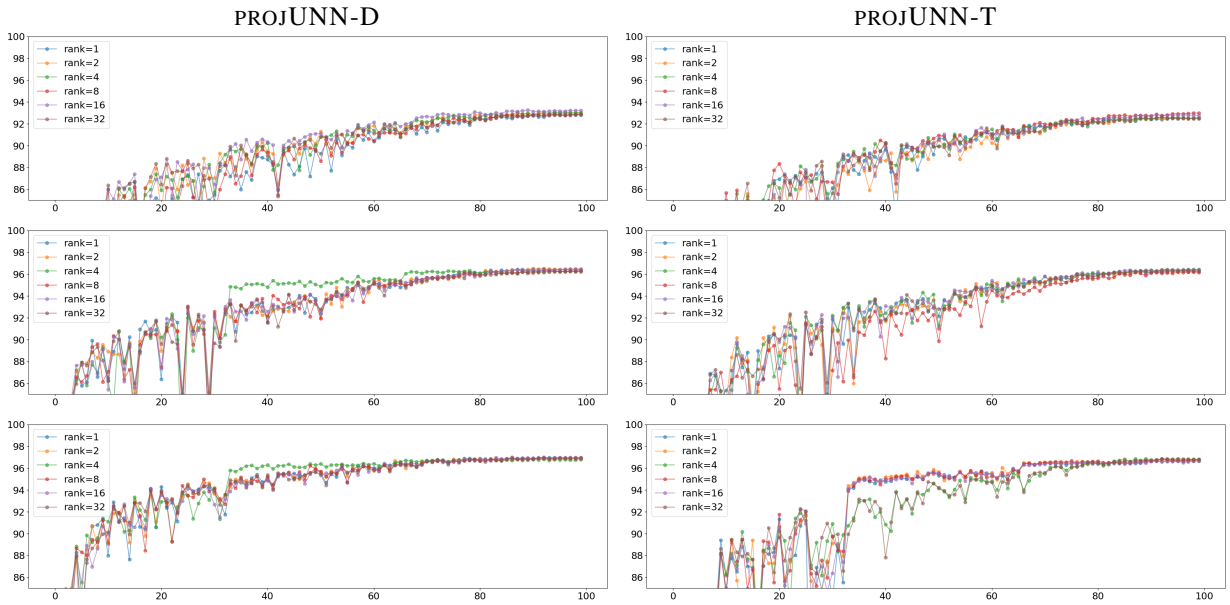


Figure 13: Evolution of the test set accuracy during training at each epoch for the pixel-MNIST task. We depict the evolution for different RNN width (from top to bottom: 116,360 and 512). We observe that regardless of the rank k of the projUNN update, we reach the same final performances.

augmentation of random translations and left-right flips. We leverage the Resnet [34] architecture. As our previous analysis in the RNN setting has shown that rank $k = 1$ is sufficient for convergence, we always set $k = 1$ when using PROJUNN in the convolutional setting. We leverage the RMSprop optimizer and perform cross-validation on the learning rate. We present our results in Table 3 and make two observations. First, the smaller model (Resnet9) is able to reach or slightly outperform the alternative exact orthogonal constraints. Second, the larger model (Resnet18) falls behind the approximate orthogonal constraint method. This result is perhaps expected as our convolutional layers are full-width, *i.e.,* they allow for much greater degree of over-fitting. This was not detrimental in the small model with fewer convolutional layers. As a result, although we validate the ability of PROJUNN to produce state-of-the-art small convolutional networks, there remains open avenues of research to extend the method to larger models. As an aside, the resnet architecture is, in a sense, a very stable architecture since it is precisely designed to be able to incorporate many layers. To provide an honest comparison to prior work, we analyzed the performance of PROJUNN with respect to this resnet architecture, but note that more "vanilla" CNN architectures may be better targets for unitary constraints.

E Analysis of low rank updates

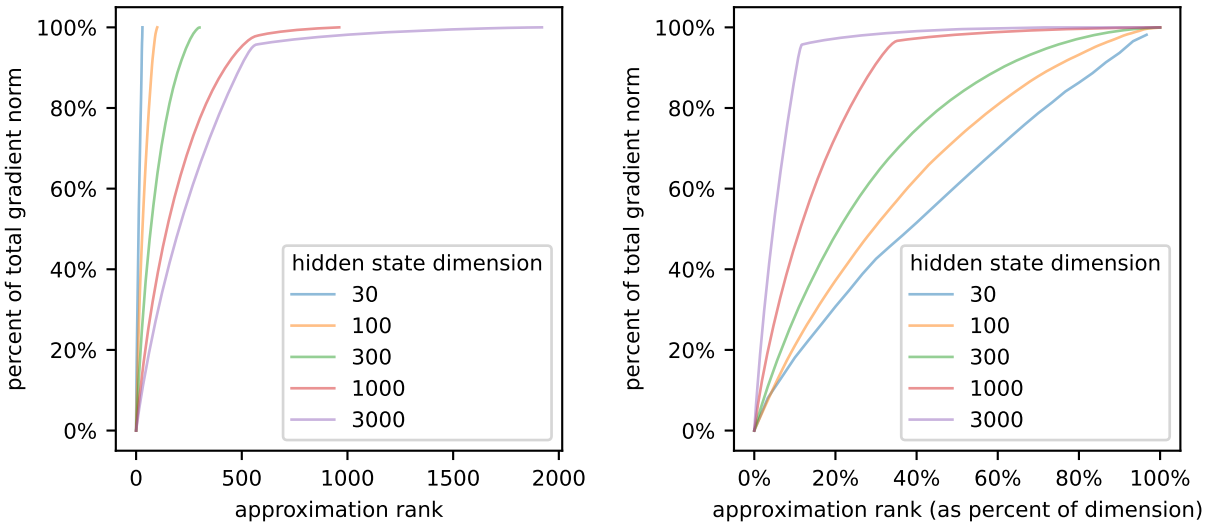
It is typically the case that gradients of matrices with respect to a loss function are not *exactly* low rank but *approximately* low rank. More specifically, a matrix $\boldsymbol{A}$ is *approximately* low rank if there exists a rank k matrix $\boldsymbol{A}_k$ such that the relative error of the approximation E_{rel} is small:

$$E_{rel} = \frac{\|\boldsymbol{A} - \boldsymbol{A}_k\|_F}{\|\boldsymbol{A}\|_F} \tag{E.1}$$

where $\|\cdot\|_F$ denotes the Frobenius norm of a matrix. We note that there is a connection between the above and the stable rank of a matrix $\boldsymbol{A}$ defined as $\|\boldsymbol{A}\|_F^2/\|\boldsymbol{A}\|_2^2$. The stable rank is upper bounded by the exact or hard rank of a matrix $\boldsymbol{A}$.

In RNN settings, Figure 14 which plots $1 - E_{rel}$ shows that low rank approximations to a gradient can typically be very close to the true gradient even as the hidden size of the RNN is large. In Figure 14, the RNN was given an input of batch size 64 where each input is a sequence of length 20 and each element of the sequence is a vector in $\mathbb{R}^{200}$ with elements drawn from the standard normal distribution. The RNN outputs logits to learn random target integers ranging from 1 to 10. Gradients were calculated for a single optimization step over a single batch of the random input and output data.

As observed in the main text, such low rank behavior is often more evident in real-world data. In fact, as seen in Figure 1a, virtually all the information of the gradient in a convolutional architecture is captured in the first few singular



(a) unnormalized

(b) normalized by dimension

Figure 14: Portion of Frobenius norm (1-E_{rel}) captured by a low rank approximation to the gradient of the matrix typically improves with the dimension of the matrix. For larger matrices, at least half of the Frobenius norm of the gradient can be captured with a low rank approximation of a small percent of the overall dimension. Here, we assume that low rank approximations to the matrix are optimal and the RNN is trained on random inputs and outputs. Plots are for the matrix performing transformation from hidden states to hidden states (*i.e.,* the matrix replaced by a unitary/orthogonal matrix in PROJUNN implementations.

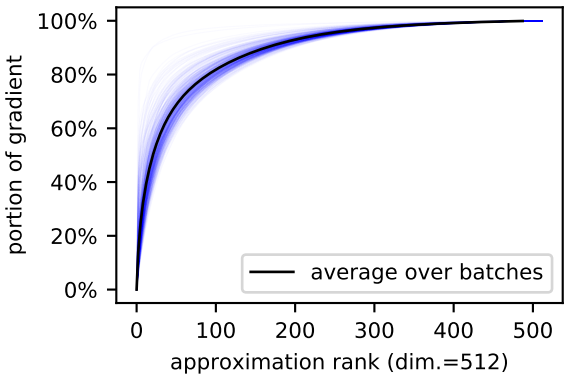


Figure 15: Low rank approximations capture most of the Frobenius norm of the gradient. Here, we plot gradients of the matrix in the RNN's hidden layer for each batch (light blue line) over an epoch of training PROJUNN in the pixel-by-pixel MNIST task (see Section 5).

vectors. Gradients here are shown for a single $C \times C$ block in the Fourier regime of the convolution filter parameterized via our orthogonal PROJUNN convolution (see Section 4.4 and Appendix B.1 for form of parameterization). This filter is contained in the last residual block of the Resnet-9 network [34] and has 512 channels so the gradient is a 512×512 matrix. Since networks were trained with a batch size of 128, the maximum rank of this gradient is actually 128 (see short proof below in Proposition E.2).

Since the weight matrix in hidden layers of RNNs is repetitively applied, gradients tend to be of a higher stable rank in these settings. Nevertheless, as evident in Figure 15, gradients over the course of an epoch of training an RNN of hidden dimension 512 on the sequential MNIST task still exhibit clear low rank behavior, albeit not to the same degree as convolutional architectures. In constructing these plots, a batch size of 128 was used and networks were trained using the RMSprop algorithm. Throughout our experience with training the PROJUNN on various tasks, we observe that setting the rank of an approximation to even just one is effective at ensuring convergence of training to a good solution.

We furthermore note that the exact rank of a gradient in neural network architectures can typically be bounded by the dimensions of the input. For convolutional networks, this exact rank is bounded by the batch size and for vanilla recurrent neural networks, this exact rank is bounded by the batch size times the sequence length. We provide formal statements and short proofs of these propositions below.

Proposition E.1. *Given a loss function $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ taking in two real numbers and outputting a real number, let $f^{(t)}$ denote the t-th sequential output of a vanilla RNN defined as*

$$\begin{aligned} \boldsymbol{h}^{(t)}(\boldsymbol{x}) &= \sigma(\boldsymbol{M}\boldsymbol{x}^{(t)} + \boldsymbol{W}\boldsymbol{h}^{(t-1)}(\boldsymbol{x})) \\ f^{(t)}(\boldsymbol{x}) &= \tau(\boldsymbol{V}\boldsymbol{h}^{(t)}(\boldsymbol{x})), \end{aligned} \tag{E.2}$$

where $t \in [T]$ indexes the sequence of inputs, inputs $\boldsymbol{x}^{(t)} \in \mathbb{R}^d$ ($\boldsymbol{x}$ denotes the concatenation of all inputs), input transformation $\boldsymbol{M} \in \mathbb{R}^{h\times d}$, hidden transformation $\boldsymbol{W} \in \mathbb{R}^{h\times h}$, output transformation $\boldsymbol{V} \in \mathbb{R}^{h\times d}$, and σ and τ are pointwise non-linearities. Let $\nabla_{\boldsymbol{W}} L$ indicate the gradient of the hidden transformation matrix $\boldsymbol{W}$ with respect to the loss function $L = \sum_{i=1}^{b} \ell(f^{(T)}(\boldsymbol{x}_i), y_i)$ over a batch of inputs $\{\boldsymbol{x}_i, y_i\}_{i=1}^{b}$. The rank of $\nabla_{\boldsymbol{W}} L$ is bounded above by bT.

Proof. Define $\boldsymbol{z}^{(t)} = \boldsymbol{M}\mathbf{x}^{(t)} + \boldsymbol{W}\boldsymbol{h}^{(t-1)}(\boldsymbol{x})$. Calculating the gradient of the loss function with respect to the parameter $\boldsymbol{W}$, we have:

$$\begin{aligned} \nabla_{\boldsymbol{W}} L &= \sum_{i=1}^{b} \nabla_{\boldsymbol{W}} \ell(f^{(T)}(\boldsymbol{x}_i), y_i) \\ &= \sum_{i=1}^{b}\sum_{t=1}^{T}\sum_{k=1}^{h} \left[\frac{\partial \ell(f^{(T)}(\boldsymbol{x}_i), y_i)}{\partial \boldsymbol{z}^{(t)}}\right]_k \frac{\partial \boldsymbol{z}_k^{(t)}}{\partial \boldsymbol{W}} \\ &= \sum_{i=1}^{b}\sum_{t=1}^{T} \left[\frac{\partial \ell(f^{(T)}(\boldsymbol{x}_i), y_i)}{\partial \boldsymbol{z}^{(t)}}\right] \left[\boldsymbol{h}^{(t-1)}(\boldsymbol{x}_i)\right]^{\intercal}. \end{aligned} \tag{E.3}$$

The above is the sum of bT rank one matrices which is at most rank bT concluding the proof. □

Proposition E.2. *Given a loss function $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ taking in two real numbers and outputting a real number and convolution filter $\mathbf{W} \in \mathbb{R}^{N\times M\times C\times C}$, let $f(\mathbf{X})$ denote the convolutional neural network defined as*

$$\begin{aligned} \mathbf{H}(\mathbf{X}) &= g_{pre}(\mathbf{X}) \\ \mathbf{Y}(\mathbf{X}) &= \mathrm{conv}_{\mathbf{W}}(\mathbf{H}(\mathbf{X})) \\ f(\mathbf{X}) &= g_{post}(\mathbf{Y}(\mathbf{X})), \end{aligned} \tag{E.4}$$

where input $\mathbf{X} \in \mathbb{R}^{N_{in}\times M_{in}\times C_{in}}$ and g_{pre} and g_{post} are functions which apply the layers before and after the convolution with filter $\mathbf{W}$. Let $\widehat{\mathbf{W}}_{\widehat{r},\widehat{s},:,:}$ denote the $C \times C$ matrix storing the values of the convolution filter in the Fourier regime for frequencies $\widehat{r}$ and $\widehat{s}$ (see Equation (B.15)*). Let $\nabla_{\widehat{\mathbf{W}}_{\widehat{r},\widehat{s},:,:}} L$ indicate the gradient of $\widehat{\mathbf{W}}_{\widehat{r},\widehat{s},:,:}$ with respect to the loss function $L = \sum_{i=1}^{b} \ell(f(\mathbf{X}_i), y_i)$ over a batch of inputs $\{\mathbf{X}_i, y_i\}_{i=1}^{b}$. Then, for all $\widehat{r}$ and $\widehat{s}$ in the support of the filter, the rank of $\nabla_{\widehat{\mathbf{W}}_{\widehat{r},\widehat{s},:,:}} L$ is bounded above by b.*

Proof. We have that

$$\nabla_{\widehat{\mathbf{W}}_{\widehat{r},\widehat{s},:,:}} L = \sum_{i=1}^{b} \nabla_{\widehat{\mathbf{W}}_{\widehat{r},\widehat{s},:,:}} \ell(f(\mathbf{X}_i), y_i). \tag{E.5}$$

We now proceed to show that the rank of $\nabla_{\widehat{\mathbf{W}}_{:,:,\widehat{r},\widehat{s}}} \ell(f(\mathbf{X}_i), y_i)$ is equal to one which completes the proof. As a reminder, we have via the convolution theorem that

$$[\mathrm{FFT}\,\mathrm{conv}_{\mathbf{W}}(\mathbf{X})]_{\widehat{r},\widehat{s},:} = \widehat{\mathbf{W}}^{*}_{\widehat{r},\widehat{s},:,:}\ [\mathrm{FFT}\,\mathbf{X}]_{\widehat{r},\widehat{s},:}\,. \tag{E.6}$$

Let $\widehat{\mathbf{Y}}(\mathbf{X}) = \mathrm{FFT}\,\mathrm{conv}_{\mathbf{W}}(\mathbf{X})$, then we have

$$\begin{aligned} \nabla_{\widehat{\mathbf{W}}_{:,:,\widehat{r},\widehat{s}}} \ell(f(\mathbf{X}_i), y_i) &= \sum_{k=1}^{C} \frac{\partial \ell(f(\mathbf{X}_i), y_i)}{\partial \widehat{\mathbf{Y}}(\mathbf{X})_{\widehat{r},\widehat{s},k}} \frac{\partial \widehat{\mathbf{Y}}(\mathbf{X})_{\widehat{r},\widehat{s},k}}{\mathbf{W}_{\widehat{r},\widehat{s},:,:}} \\ &= \frac{\partial \ell(f(\mathbf{X}_i), y_i)}{\partial \widehat{\mathbf{Y}}(\mathbf{X})_{\widehat{r},\widehat{s},:}} \left([\mathrm{FFT}\,\mathbf{X}]_{\widehat{r},\widehat{s},:}\right)^{\intercal}. \end{aligned} \tag{E.7}$$

Model	Complexity of forward + backward step	Notes
BCOP [1]	$O(C^2W^2N^2 + C^3W^3)$	W sequential orthogonalizations needed (slow for large W)
SOC [2]	$O(pC^2W^2N^2)^a$	p denotes number of terms in Taylor series approximation
Cayley [3]	$O(N^2C^2\log(N) + N^2C^3)^b$	
PROJUNN (our method)	$O(N^2C\log(N) + kN^2C^2)^c$	k denotes rank of gradient updates

[1] [58], [2] [73], [3] [76]

[a] just applying the convolution (without gradient update) also requires the added factor p in runtime unlike standard convolutions and BCOP which run in time $O(C^2W^2N^2)$ upon just evaluation, [b] approximations to matrix inversion exist which may reduce runtimes though these approximations are not implemented here, [c] runtime shown for typical setting when $k \ll n$

Table 4: Time complexity of 2-D orthogonal convolutional layers with filter size $W \times W$ applied to $N \times N$ inputs with C input and output channels.

The above is a rank one matrix which concludes the proof. □

E.1 Other sampling algorithms

The prior analysis showed that gradients are typically low rank, and in the main text, we listed a couple efficient algorithms that allow one to approximate a full but approximately low rank matrix with an explicitly low rank matrix. Enhancements to these algorithms exist that may provide further improvements in very large dimensions. For example, the *FKV sampling* [27] and *constant time SVD* [23] algorithms provide runtimes that are often logarithmic in the dimension of the matrix for sampling entries from the low rank approximation to a matrix. Nevertheless, explicitly constructing all entries of a rank k approximation to an $n \times n$ matrix $\boldsymbol{A}$ requires time at least $O(k^2n)$. Improvements in runtime are achieved by applying random projections to both the column and row subspaces of the matrix $\boldsymbol{A}$ to perform the final approximation. In practice, the runtime of these algorithms depends on other factors that limit the applicability of the algorithm even for relatively large matrices. Numerical simulations show that FKV sampling achieves a practical speedup for matrices of dimension approximately 10^6 or higher [8]. Since matrices in our PROJUNN were of much smaller dimension, we did not use these methods.

F Runtime comparisons

Table 1 (see main text) and Table 4 lists the asymptotic runtime complexities of unitary/orthogonal models in the RNN and convolutional setting respectively. For RNNs, PROJUNN has the best runtime scaling of all models which parameterize the full orthogonal/unitary space with just one layer. In fact, apart from the rank approximation factor k, the runtime of PROJUNN is optimal in the RNN setting. In the convolutional setting, PROJUNN scales most efficiently when there are many channels or the filter size W scales faster than the logarithm of the input dimension $\log N$. For small filter sizes, BCOP and SOC scale relatively efficiently. In fact, SOC is nearly optimal for $W \ll \log N$ up to the approximation factor p in the Taylor expansion which is required both upon evaluation and training of the network.

Runtimes for RNN implementations in practice Enforcing unitarity in PROJUNN incurs a computational overhead associated to performing eigendecompositions and QR decompositions entailed in updating the gradient. Even though such computations are performed on a small subspace of the total dimension of the matrix, such computations may not effect increase training times by a constant factor as evident in Figure 2a. Empirically, we observed that much of the increased overhead was due to performing eigendecompositions and QR decompositions on a GPU, both tasks which are challenging to parallelize on GPU architectures. We note that similar issues may be one reason why the expRNN runtimes appear much slower in our simulations as shown in Figure 2a. Recent updates to pytorch [70] and tensorflow [2] have focused on improving the runtimes of these common linear algebraic routines, and we expect these runtimes to continue to improve in the future.

Runtimes for convolutional network implementations in practice Figure 16, Figure 17, and Figure 18 compare runtimes of orthogonal convolution algorithms when varying the number of channels, input size, and filter size respectively. Figure 19 plots the runtime when the filter and image size are set equal to each other and varied together. In summary, we find that the skew orthogonal convolution (SOC) [73], the Cayley model [76], and our PROJUNN are all relatively efficient for small to medium sized models. PROJUNN empirically performs best in comparison to other models when there are many channels or the filter size is large. SOC [73] is the fastest when the filter size is small. As a reminder, SOC [73] employs the Taylor series approximation to form a unitary/orthogonal matrix. We did not change

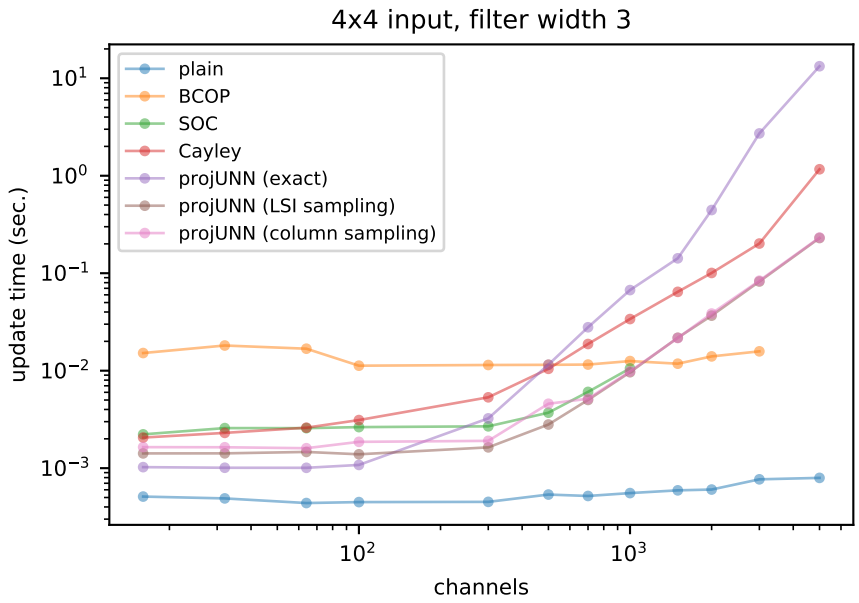


Figure 16: Comparison of runtimes of orthogonal convolutional architectures when varying the number of channels (log-log plot). The number of input channels is equal to the number of output channels. Runtimes are averaged over 10 forward and backward passes through a single convolutional operation on random data. PROJUNN with LSI sampling and column sampling have very similar runtimes and may appear to completely overlap on the chart. Out of memory error obtained for BCOP for large number of channels.

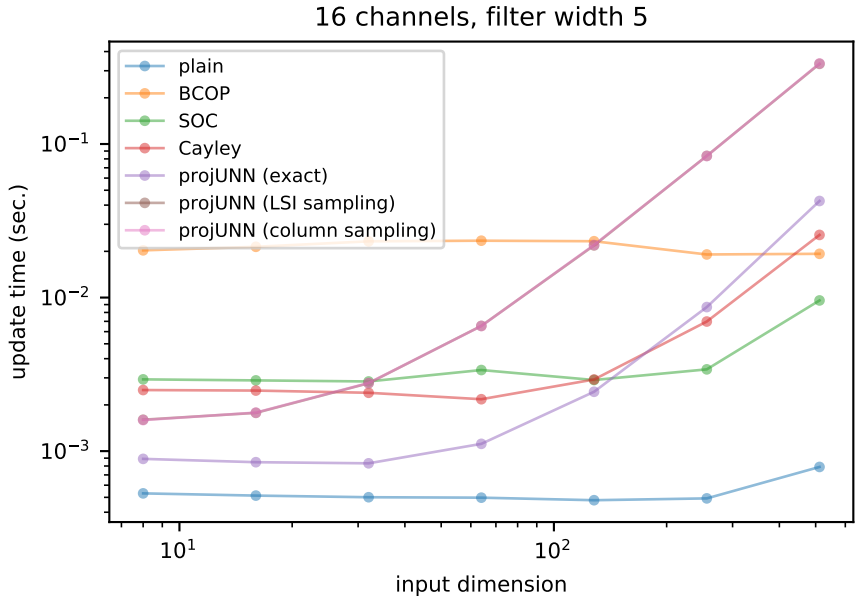


Figure 17: Comparison of runtimes of orthogonal convolutional architectures when varying the size of the input which is a square image (log-log plot). Runtimes are averaged over 10 forward and backward passes through a single convolutional operation on random data. PROJUNN with LSI sampling and column sampling have very similar runtimes and may appear to completely overlap on the chart.

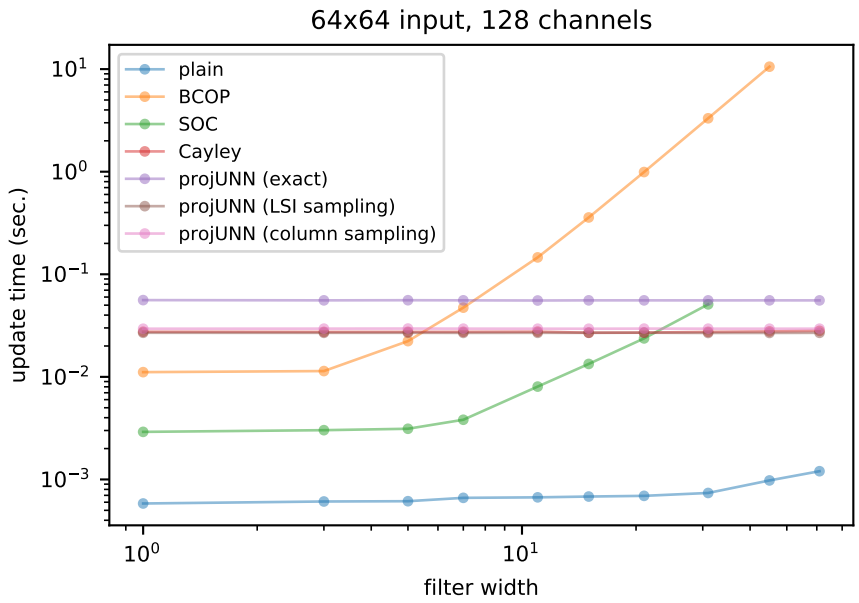


Figure 18: Comparison of runtimes of orthogonal convolutional architectures when varying the size of the filter (log-log plot). Both PROJUNN and the cayley convolution [76] perform operations on the full space of convolution filters so the filter size does not change the runtime for these models. Runtimes are averaged over 10 forward and backward passes through a single convolutional operation on random data. PROJUNN with LSI sampling and column sampling have very similar runtimes and may appear to completely overlap on the chart. Out of memory error obtained for SOC beyond filter size of 31.

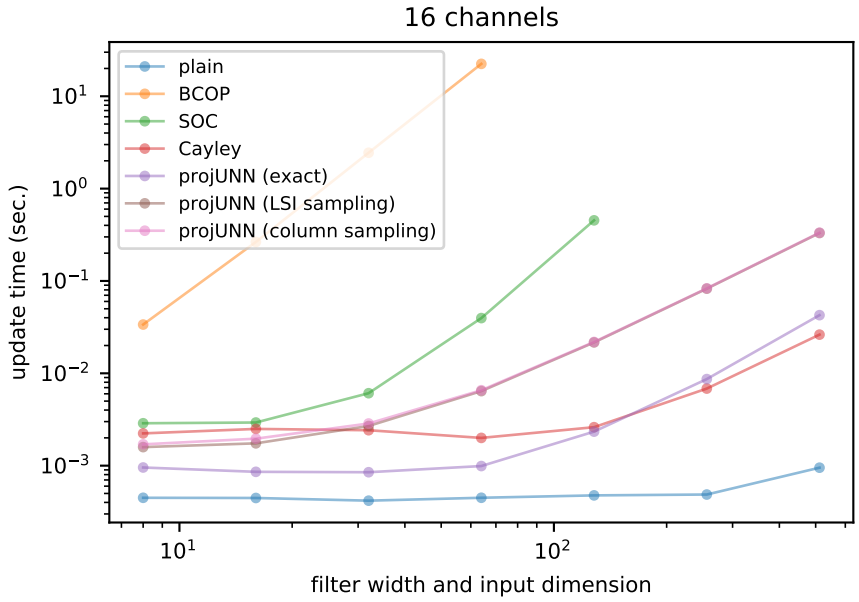


Figure 19: Comparison of runtimes of orthogonal convolutional architectures when varying the size of the filter and the input dimension in-tandem (log-log plot). Runtimes are averaged over 10 forward and backward passes through a single convolutional operation on random data. PROJUNN with LSI sampling and column sampling have very similar runtimes and may appear to completely overlap on the chart. Out of memory error obtained for SOC beyond filter size and image size of 128.

the number of terms in the approximation used even though increasing the size of a filter, dimension of an input, or the number of channels will likely require more terms to maintain the same error in the approximation.

The Cayley model [76] has a runtime that is empirically similar to PROJUNN in most of our experiments despite PROJUNN having an improved asymptotic scaling with regards to the number of channels (see Table 4). Runtimes here are calculated by averaging across the first 10 steps of gradient descent on random data for initialized networks. Close to initialization, matrices in the convolution filter are generally well conditioned and this may be one reason why the Cayley model showed good scaling in Figure 16 since the matrix inversion for the well conditioned matrices runs quickly. We also suspect that for the number of channels considered, the added costs of performing sampling and projections in PROJUNN do not materialize into a very clear runtime benefit. Interestingly, the exact version of PROJUNN (no low rank approximation) runs in similar time to the Cayley method.

Finally, we note that in comparison to RNN implementations, there is increased overhead associated to performing eigendecompositions and QR decompositions with our projUNN convolutional network implementations since unitary matrices are batched across the elements of the convolutional filter. As mentioned before, much of the increased overhead was due to performing eigendecompositions and QR decompositions on a GPU, both tasks which can be challenging to parallelize on GPU based architectures. Future updates to Pytorch [70] and Tensorflow [2] may help improve runtimes of these operations.

G Network architectures and training details

ExpRNN trivializations	https://github.com/Lezcano/expRNN	MIT
scoRNN	https://github.com/SpartinStuff/scoRNN	NA
scuRNN	https://github.com/Gayan225/scuRNN	NA
skewed orthogonal convolutions (SOC)	https://github.com/singlasahil14/SOC	NA
Cayley convolution	https://github.com/locuslab/orthogonal-convolutions	MIT
BCOP	github.com/ColinQiyangLi/LConvNet	MIT

G.1 Handling complex numbers

Since matrices in the unitary group are complex valued, care must be taken to ensure that the neural network can handle such complex numbers. In these settings, standard activation functions need to be adapted for complex numbers. It is often advantageous to have a nonlinearity which does not change the phase of its input. For these reasons, the activation function used is typically the modRELU activation shown below:

$$\begin{aligned} \sigma_{modRELU}(z) &= (|z| + b)\frac{z}{|z|} \ \text{ if } \ \|z\| + b > 0 \\ \sigma_{modRELU}(z) &= 0 \ \text{ if } \ |z| + b \leq 0 \end{aligned} \tag{G.1}$$

where b is a bias term (trainable). In calculations of the modulus $|z|$, a small additive constant is often added to avoid stability issues when $|z|$ is small.

Another challenge that arises with performing orthogonal convolution is that the representation of a real-valued convolutional filter in the Fourier domain will be complex-valued. More specifically, the fast Fourier transform of a real signal is Hermitian-symmetric. For example, for a one dimensional vector $\boldsymbol{x} \in \mathbb{R}^N$ where $\widehat{\boldsymbol{x}} = \mathrm{FFT}\,\boldsymbol{x} \in \mathbb{C}^N$ and $\widehat{\boldsymbol{x}}_i = \widehat{\boldsymbol{x}}_{-i \mod N}$. Therefore, when initializing weight filters in our orthogonal convolution operations, one must be careful to ensure the Hermitian symmetric property holds. Similarly, when performing convolutions in the Fourier space, care must be taken in converting data types to and from complex and real space. In our implementation, we used the pytorch FFT.RFFT2 and FFT.IRFFT2 commands to implement these operations efficiently [69].

G.2 Optimizers

For training PROJUNN architectures, a separate optimizer is used for unitary/orthogonal parameters which must be projected after updates and all other parameters. Learning rates for the unitary/orthogonal parameters are typically set to less than that of the other parameters. In our implementations, we found that a learning rate for the unitary or orthogonal parameters of one-tenth or one-twentieth of that of the other parameters works well in practice. Furthermore, when using optimizers with momentum terms, we constructed a variant of standard optimizers to apply changes to the momentum terms after projecting gradients onto the unitary/orthogonal manifold or tangent space. This optimizer performed well in our experiments though sometimes added instability. Therefore, unless otherwise stated, standard optimizers were used.

G.3 Numerical stability

Though updates via PROJUNN mathematically guarantee that parameters remain unitary/orthogonal, performing a significant number of sequential updates to a matrix can add numerical errors over time slowly drifting parameters away from the unitary/orthogonal manifold. This is especially true in the case of PROJUNN-D where updating matrices requires division of eigenvalues in the eigendecompositon of the rank k subspace (see Appendix C and Equation (C.16)). In contrast, we empirically find that PROJUNN-T maintains stability throughout optimization even when requiring many updates. To actively address potential instabilities, one can perform eigendecompositions with higher floating point precision or sporadically project unitary/orthogonal parameters onto the closest unitary/orthogonal matrix via Lemma 4.1. Though this projection step requires $O(n^3)$ time for an $n \times n$ matrix, performing this projection only every $O(n)$ gradient updates can still preserve efficient runtimes of on average $O(kn^2)$ time per gradient update. Unless otherwise stated, we do not perform any additional steps for maintaining stability.

G.4 Initialization of unitary/orthogonal parameters

Empirically, we found that initializing unitary/orthogonal matrices to be Haar random or close to Haar random does not perform well. This is also an observation noted in prior works [36, 35, 56]. Instead, we used one of two different initialization schemes. The first initializes unitary/orthogonal parameters as the identity matrix. The second initializes unitary/orthogonal matrices using variants of the so-called Henaff initialization [36] where 2×2 diagonal blocks of the matrices are initialized as samples from the below

$$\exp\left(\begin{bmatrix} 0 & s_i \\ -s_i & 0 \end{bmatrix}\right), \tag{G.2}$$

where s_i is sampled uniformly from $[-\pi, \pi]$ [36]. In other words, an $n \times n$ matrix $\boldsymbol{U}$ is initialized as

$$\boldsymbol{U} = \begin{bmatrix} \exp\left(\begin{bmatrix} 0 & s_1 \\ -s_1 & 0 \end{bmatrix}\right) & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \exp\left(\begin{bmatrix} 0 & s_2 \\ -s_2 & 0 \end{bmatrix}\right) & & \mathbf{0} \\ \vdots & & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \exp\left(\begin{bmatrix} 0 & s_{n/2} \\ -s_{n/2} & 0 \end{bmatrix}\right) \end{bmatrix}. \tag{G.3}$$

Note, that since we parameterize matrices in the Lie group instead of the Lie algebra, we include the exponential map in the parameterization above. Other variants of the above have been used in prior works. For example, the Cayley initialization samples $s_i = \sqrt{\frac{1-\cos t_i}{1+\cos t_i}}$ where t_i is sampled uniformly from $[0, \pi/2]$ [35].

G.5 Computational details

We employed the latest version of PyTorch (1.10.1+cu102) with Python3.8 and all the pre-requisite libraries coming along. The hardware leverages Quadro GP100 GPU and Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz.