Appendix Outline

This Appendix is organized as follows:

- In Appendix A we describe various dispatch rules including the base rules, the composition rules and rules derived from other rules.
- In Appendix B we provide an extended discussion of several noteworthy features of CoLA, such as doubly stochastic estimators and memory-efficient autograd implementation.
- In Appendix C we include pseudo-code on various of the iterative methods incorporated in CoLA and discuss modifications to improve lower precision performance.
- In Appendix D we expand on the details of the experiments in the main text.
- In Appendix E we show some code examples of how the dispatch rules are implemented in CoLA.

A Dispatch Rules

We now present the linear algebra identities that we use to exploit structure in CoLA.

A.1 Core Functions

A.1.1 Inverses

We incorporate several identities for the compositional operators: product, Kronecker product, block diagonal and sum. For product we have $(\mathbf{AB})^{-1} = (\mathbf{B}^{-1}\mathbf{A}^{-1})$ and for Kronecker product we have $(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1}$. In terms of block compositions we have the following identities:

$$\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{D}^{-1} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{0} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}\mathbf{D}^{-1} \\ \mathbf{0} & \mathbf{D}^{-1} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{I} & -\mathbf{A}^{-1}\mathbf{B} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{C}\mathbf{A}^{-1} & \mathbf{I} \end{bmatrix}$$

Finally, for sum we have the Woodbury identity and its variants. Namely, for Woodbury we have

$$(\mathbf{A} + \mathbf{UBV})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}\left(\mathbf{B}^{-1} + \mathbf{V}\mathbf{A}^{-1}\mathbf{U}\right)^{-1}\mathbf{V}\mathbf{A}^{-1},$$

the Kailath variant where

$$(\mathbf{A} + \mathbf{BC})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{B}\left(\mathbf{I} + \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)\mathbf{C}\mathbf{A}^{-1}$$

and the rank one update via the Sherman-Morrison formula

$$(\mathbf{A} + \mathbf{b}\mathbf{c}^\intercal)^{-1} = \mathbf{A}^{-1} - \frac{1}{1 + \mathbf{c}^\intercal\mathbf{A}\mathbf{b}}\mathbf{A}^{-1}\mathbf{b}\mathbf{c}^\intercal\mathbf{A}^{-1}.$$

Besides the compositional operators, we have some rules for some special operators. For example, for $\mathbf{A} = \texttt{Diag}(\mathbf{a})$ we have $\mathbf{A}^{-1} = \texttt{Diag}\left(\mathbf{a}^{-1}\right)$. Also, if $\mathbf{Q}$ is unitary then $\mathbf{Q}^{-1} = \mathbf{Q}^*$ or if $\mathbf{Q}$ is orthonormal then $\mathbf{Q}^{-1} = \mathbf{Q}^\intercal$. In Appendix E we show how these dispatch rules are implemented in `Python`.

A.1.2 Eigendecomposition

We now assume that the matrices in this section are diagonalizable. That is, $\texttt{Eigs}(\mathbf{A}) = \mathbf{\Lambda_A}, \mathbf{V_A}$, where $\mathbf{A} = \mathbf{V_A}\mathbf{\Lambda_A}\mathbf{V_A}^{-1}$. In terms of the compositional operators, there is not a general rule for product or sum. However, for the Kronecker product we have $\texttt{Eigs}(\mathbf{A} \otimes \mathbf{B}) = \mathbf{\Lambda_A} \otimes \mathbf{\Lambda_B},\ \mathbf{V_A} \otimes \mathbf{V_B}$ and for the Kronecker sum we have $\texttt{Eigs}(\mathbf{A} \oplus \mathbf{B}) = \mathbf{\Lambda_A} \oplus \mathbf{\Lambda_B},\ \mathbf{V_A} \otimes \mathbf{V_B}$. Finally, for block diagonal we have

$$\texttt{Eigs}\left(\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{D} \end{bmatrix}\right) = \begin{bmatrix} \mathbf{\Lambda_A} & \mathbf{0} \\ \mathbf{0} & \mathbf{\Lambda_D} \end{bmatrix}, \begin{bmatrix} \mathbf{V_A} & \mathbf{0} \\ \mathbf{0} & \mathbf{V_D} \end{bmatrix}.$$

A.1.3 Diagonal

As a base case, if we need to compute $\texttt{Diag}(\mathbf{A})$ for a general matrix $\mathbf{A}$ we may compute each diagonal element by $\mathbf{e}_i^\intercal \mathbf{A} \mathbf{e}_i$. Additionally, if $\mathbf{A}$ is large enough we switch to randomized estimation $\texttt{Diag}(\mathbf{A}) \approx (\mathbf{Z} \odot \mathbf{A}\mathbf{Z})\mathbf{1}/N$ with $\mathbf{Z} \sim \mathcal{N}(0,1)^{d \times N}$ where N is the number of samples used to approximate the diagonal. In terms of compositional operators, we have that for sum $\texttt{Diag}(\mathbf{A} + \mathbf{B}) = \texttt{Diag}(\mathbf{A}) + \texttt{Diag}(\mathbf{B})$. For Kronecker product we have $\texttt{Diag}(\mathbf{A} \otimes \mathbf{B}) = \texttt{vec}\big(\texttt{Diag}(\mathbf{A})\texttt{Diag}(\mathbf{B})^\intercal\big)$ and for Kronecker sum $\texttt{Diag}(\mathbf{A} \oplus \mathbf{B}) = \texttt{vec}\big(\texttt{Diag}(\mathbf{A})\,\mathbf{1}^\intercal + \mathbf{1}\texttt{Diag}(\mathbf{B})^\intercal\big)$. Finally, for block composition we have

$$\texttt{Diag}\left(\begin{bmatrix}\mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D}\end{bmatrix}\right) = [\texttt{Diag}(\mathbf{A}), \texttt{Diag}(\mathbf{D})].$$

A.1.4 Transpose / Adjoint

As explained in Section 3.1, as a base case we have an automatic procedure to compute the transpose or adjoint of any operator $\mathbf{A}$ via autodiff. However, we also incorporate the following rules. For sum we have $(\mathbf{A} + \mathbf{B})^* = \mathbf{A}^* + \mathbf{B}^*$ and $(\mathbf{A} + \mathbf{B})^\intercal = \mathbf{A}^\intercal + \mathbf{B}^\intercal$. For product we have $(\mathbf{A}\mathbf{B})^* = \mathbf{B}^*\mathbf{A}^*$ and $(\mathbf{A}\mathbf{B})^\intercal = \mathbf{B}^\intercal\mathbf{A}^\intercal$. For Kronecker product we have $(\mathbf{A} \otimes \mathbf{B})^* = \mathbf{A}^* \otimes \mathbf{B}^*$ and $(\mathbf{A} \otimes \mathbf{B})^\intercal = \mathbf{A}^\intercal \otimes \mathbf{B}^\intercal$. For the Kronecker sum we have $(\mathbf{A} \oplus \mathbf{B})^* = \mathbf{A}^* \oplus \mathbf{B}^*$ and $(\mathbf{A} \oplus \mathbf{B})^\intercal = \mathbf{A}^\intercal \oplus \mathbf{B}^\intercal$. In terms of block composition we have

$$\left(\begin{bmatrix}\mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D}\end{bmatrix}\right)^* = \begin{bmatrix}\mathbf{A}^* & \mathbf{C}^* \\ \mathbf{B}^* & \mathbf{D}^*\end{bmatrix} \quad \text{and} \quad \left(\begin{bmatrix}\mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D}\end{bmatrix}\right)^\intercal = \begin{bmatrix}\mathbf{A}^\intercal & \mathbf{C}^\intercal \\ \mathbf{B}^\intercal & \mathbf{D}^\intercal\end{bmatrix}.$$

Finally for the annotated operators we have the following rules. $\mathbf{A}^* = \mathbf{A}$ if $\mathbf{A}$ is self-adjoint and $\mathbf{A}^\intercal = \mathbf{A}$ if $\mathbf{A}$ is symmetric.

A.1.5 Pseudo-inverse

As a base case, if we need to compute $\mathbf{A}^+$, we may use $\texttt{SVD}(\mathbf{A}) = \mathbf{U}, \mathbf{\Sigma}, \mathbf{V}$ and therefore set $\mathbf{A}^+ = \mathbf{U}\mathbf{\Sigma}^+\mathbf{V}^*$, where $\mathbf{\Sigma}^+$ inverts the nonzero diagonal scalars. If the size of $\mathbf{A}$ is too large, then we may use randomized SVD. Yet, it is uncommon to simply want $\mathbf{A}^+$, usually we want to solve a least-squares problem and therefore we can use solvers that are not as expensive to run as SVD. For the compositional operators we have the following identities. For product $(\mathbf{A}\mathbf{B})^+ = \left(\mathbf{A}^+\mathbf{A}\mathbf{B}\right)^+ \left(\mathbf{A}\mathbf{B}\mathbf{B}^+\right)^+$ and for Kronecker product we have $(\mathbf{A} \otimes \mathbf{B})^+ = \mathbf{A}^+ \otimes \mathbf{B}^+$. For block diagonal we have

$$\left(\begin{bmatrix}\mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{D}\end{bmatrix}\right)^+ = \begin{bmatrix}\mathbf{A}^+ & \mathbf{0} \\ \mathbf{0} & \mathbf{D}^+\end{bmatrix}.$$

Finally, we have some identities that are mathematically trivial but that are necessary when recursively exploiting structure as that would save computation. For example, if $\mathbf{Q}$ is unitary we know that $\mathbf{Q}^+ = \mathbf{Q}$ and similarly when $\mathbf{Q}$ is orthonormal. If $\mathbf{A}$ is self-adjoint, then $\mathbf{A}^+ = \mathbf{A}^{-1}$ and also if it is symmetric and PSD.

A.2 Derived Functions

Interestingly, the previous core functions allow us to derive multiple rules from the previous ones. To illustrate, we have that $\texttt{Tr}(\mathbf{A}) = \sum_i \texttt{Diag}(\mathbf{A})_i$. Additionally, if $\mathbf{A}$ is PSD we have that $f(\mathbf{A}) = \mathbf{V}_\mathbf{A} f(\mathbf{\Lambda}_\mathbf{A}) \mathbf{V}_\mathbf{A}^{-1}$ and if $\mathbf{A}$ is both symmetric and PSD then $f(\mathbf{A}) = \mathbf{V}_\mathbf{A} f(\mathbf{\Lambda}_\mathbf{A}) \mathbf{V}_\mathbf{A}^\intercal$. where in both cases we used $\texttt{Eigs}(\mathbf{A}) = \mathbf{\Lambda}_\mathbf{A}, \mathbf{V}_\mathbf{A}$. Some example functions for PSD matrices are $\texttt{Sqrt}(\mathbf{A}) = \mathbf{V}_\mathbf{A}\mathbf{\Lambda}_\mathbf{A}^{1/2}\mathbf{V}_\mathbf{A}^{-1}$ or $\texttt{Log}(\mathbf{A}) = \mathbf{V}_\mathbf{A} \log \mathbf{\Lambda}_\mathbf{A} \mathbf{V}_\mathbf{A}^{-1}$. Which also this rules allow us to define $\texttt{LogDet}(\mathbf{A}) = \texttt{Tr}(\texttt{Log}(\mathbf{A}))$.

A.3 Other matrix identities

We emphasize that there are a myriad more matrix identities that we do not intentionally include such as $\texttt{Tr}(\mathbf{A} + \mathbf{B}) = \texttt{Tr}(\mathbf{A}) + \texttt{Tr}(\mathbf{B})$ or $\texttt{Tr}(\mathbf{A}\mathbf{B}) = \texttt{Tr}(\mathbf{B}\mathbf{A})$ when $\mathbf{A}$ and $\mathbf{B}$ are squared. These

additional cases are not part of our dispatch rules as either they are automatically computed from other rules (as in the first example) or they do not yield any computational savings (as in the second example).

B Features in CoLA

B.1 Doubly stochastic diagonal and trace estimation

Singly Stochastic Trace Estimator Consider the traditional stochastic trace estimator:

$$\overline{\texttt{Tr}}[\texttt{Base}](\mathbf{A}) = \tfrac{1}{n}\sum_{j=1}^{n} \mathbf{z}_j^\intercal \mathbf{A}\mathbf{z}_j \tag{1}$$

with each $\mathbf{z}_j \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_D)$ where $\mathbf{A}$ is a $D \times D$ matrix. When $\mathbf{A}$ is itself a sum $\mathbf{A} = \frac{1}{m}\sum_{i=1}^{m} \mathbf{A}_i$, we can expand the trace as $\overline{\texttt{Tr}}[\texttt{Base}](\mathbf{A}) = \frac{1}{mn}\sum_{j=1}^{n}\sum_{i=1}^{m} \mathbf{z}_j^\intercal \mathbf{A}_i \mathbf{z}_j$, with probe variables shared across elements of the sum.

Consider the quadratic form $Q := \mathbf{z}^\intercal \mathbf{A}\mathbf{z}$, which for Gaussian random variables has a cumulant generating function of $K_Q(t) = \log \mathbb{E}[e^{tQ}] = -\frac{1}{2}\log\det(\mathbf{I} - 2t\mathbf{A})$. From the generating function we can derive the mean and variance of this estimator: $\mathbb{E}[Q] = K_Q'(0) = \mathrm{Tr}(\mathbf{A})$ and $\mathrm{Var}[Q] = K_Q''(0) = 2\mathrm{Tr}(\mathbf{A}^2)$. Since $\overline{\texttt{Tr}}[\texttt{Base}](\mathbf{A})$ is a sum of independent random draws of Q, we see:

$$\mathbb{E}\big[\overline{\texttt{Tr}}[\texttt{Base}](\mathbf{A})\big] = \mathrm{Tr}(\mathbf{A}) \quad \text{and} \quad \mathrm{Var}\big[\overline{\texttt{Tr}}[\texttt{Base}](\mathbf{A})\big] = \frac{2}{n}\mathrm{Tr}(\mathbf{A}^2). \tag{2}$$

Doubly Stochastic Trace Estimator For the doubly stochastic estimator, we choose probe variables which are sampled independently for each element of the sum:

$$\overline{\texttt{Tr}}[\texttt{Sum}](\mathbf{A}) = \tfrac{1}{nm}\sum_{j=1}^{n}\sum_{i=1}^{m} \mathbf{z}_{ij}^\intercal \mathbf{A}_i \mathbf{z}_{ij}. \tag{3}$$

Separating out the elements of the sum, we can write the estimator as $\overline{\texttt{Tr}}[\texttt{Sum}](\mathbf{A}) = \frac{1}{n}\sum_{j=1}^{n} R_j$ where R_j are independent random samples of the value $R = \frac{1}{m}\sum_{i=1}^{m} \mathbf{z}_i^\intercal \mathbf{A}_i \mathbf{z}_i$. The cumulant generating function is merely $K_R(t) = \sum_{i=1}^{m} K_{Q_i}(t/m)$ where $Q_i = \mathbf{z}^\intercal \mathbf{A}_i \mathbf{z}$. Taking derivatives we find that,

$$\mathbb{E}[R] = K_R'(0) = \tfrac{1}{m}\sum_{i=1}^{m} \mathrm{Tr}(\mathbf{A}_i) = \mathrm{Tr}(\mathbf{A}), \tag{4}$$

$$\mathrm{Var}[R] = K_R''(0) = \tfrac{1}{m^2}\sum_{i=1}^{m} 2\mathrm{Tr}(\mathbf{A}_i^2) = \tfrac{2}{m}\mathrm{Tr}(\tfrac{1}{m}\sum_{i=1}^{m} \mathbf{A}_i^2) \tag{5}$$

Assuming bounded moments on $\mathbf{A}_i$, then both $\mathbf{A} = \frac{1}{m}\sum_i \mathbf{A}_i$ and $S(\mathbf{A}) = \frac{1}{m}\sum_i \mathbf{A}_i^2$ will converge to fixed values as $m \to \infty$. Given that $\overline{\texttt{Tr}}[\texttt{Sum}](\mathbf{A}) = \frac{1}{n}\sum_{j=1}^{n} R_j$, we can now write the mean and variance of the doubly stochastic estimator:

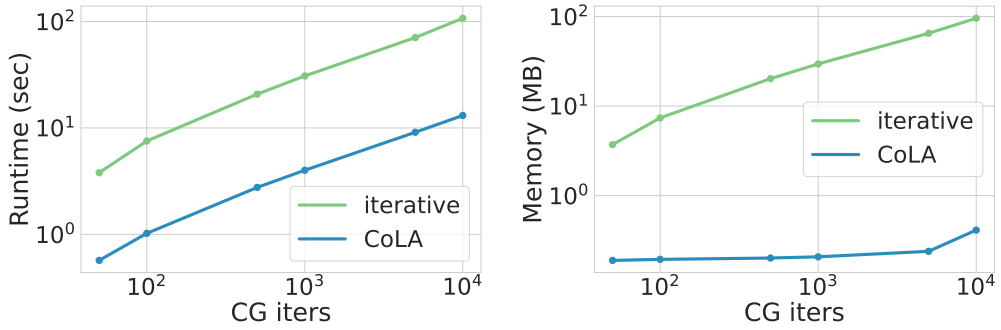
$$\mathbb{E}\big[\overline{\texttt{Tr}}[\texttt{Sum}](\mathbf{A})\big] = \mathrm{Tr}(\mathbf{A}) \quad \text{and} \quad \mathrm{Var}\big[\overline{\texttt{Tr}}[\texttt{Sum}](\mathbf{A})\big] = \frac{2}{mn}\mathrm{Tr}(S(\mathbf{A})). \tag{6}$$

As the error of the estimator can be bounded by the square root of the variance, showing that while the error for $\overline{\texttt{Tr}}[\texttt{Base}]$ is $O(1/\sqrt{n})$ (even when applied to sum structures), whereas the error for $\overline{\texttt{Tr}}[\texttt{Sum}]$ is $O(1/\sqrt{nm})$, a significant asymptotic variance reduction.

The related stochastic diagonal estimator

$$\overline{\texttt{Diag}}[\texttt{Sum}](\mathbf{A}) = \tfrac{1}{nm}\sum_{j=1}^{n}\sum_{i=1}^{m} \mathbf{z}_{ij} \odot \mathbf{A}_i \mathbf{z}_{ij}. \tag{7}$$

achieves the same $O(1/\sqrt{nm})$ convergence rate, though we omit this derivation for brevity as it is follows the same steps.



(a) Backwards pass runtime

(b) Backwards pass memory

Figure 4: **Our autograd rules allow for fast and memory efficient backpropagation**. **(a)** Runtime required to compute $\partial_{\boldsymbol{\theta}} \mathbf{A}_{\boldsymbol{\theta}}^{-1}\mathbf{b}$ for different solves, each requiring an increasing number of CG iterations. **(a)** Peak memory utilization required to compute $\partial_{\boldsymbol{\theta}} \mathbf{A}_{\boldsymbol{\theta}}^{-1}\mathbf{b}$ for different solves, each requiring an increasing number of CG iterations.

B.2 Autograd rules for iterative algorithms

For machine learning applications, we want to seamlessly interweave linear algebra operations with automatic differentiation. The most basic strategy is to simply let the autograd engine trace through the operations and backpropagate accordingly. However, when using iterative methods like conjugate gradients or Lanczos, this naive approach is extremely memory inefficient and, for problems with many iterations, the cost can be prohibitive (as seen in Figure 4). However, the linear algebra operations corresponding to inverse, eigendecomposition and trace estimation have simple closed form derivatives which we can implement to avoid the prohibitive memory consumption and reduce runtime.

Simply put, for an operation like $f = \texttt{CGSolve}$, $\texttt{CGSolve}(\mathbf{A}, \mathbf{b}) = \mathbf{A}^{-1}\mathbf{b}$ we must define a Vector Jacobian Product: $\texttt{VJP}(f, (\mathbf{A}, \mathbf{b}), \mathbf{v}) = \left(\mathbf{v}^\intercal \frac{\partial f}{\partial \mathbf{A}}, \mathbf{v}^\intercal \frac{\partial f}{\partial \mathbf{b}}\right)$. However, for matrix-free linear operators, we cannot afford to store the dense matrix $\mathbf{A}$, and thus neither can we store the gradients with respect to each of its elements! Instead we must (recursively) consider how the linear operator was constructed in terms of its differentiable arguments. In other words, we must flatten the tree structure of possibly nested differentiable arguments into a vector: $\theta = \texttt{flatten}[\mathbf{A}]$. For example for $\mathbf{A} = \texttt{Kron}\big(\texttt{Diag}(\theta_1), \texttt{Conv}(\theta_2)\big)$, $\texttt{flatten}[\mathbf{A}] = [\theta_1, \theta_2]$. From this perspective, we consider $\mathbf{A}$ as a container or tree of its arguments θ, and define $v^\intercal \frac{\partial f}{\partial \mathbf{A}} := \texttt{unflatten}[v^\intercal \frac{\partial f}{\partial \boldsymbol{\theta}}]$ which coincides with the usual definition for dense matrices. Applying to inverses, we can now write a simple `VJP`:

$$\mathbf{v}^\intercal \frac{\partial f}{\partial \mathbf{A}} = \texttt{unflatten}[\texttt{VJP}\big(\theta \mapsto \texttt{unflatten}(\theta)\mathbf{A}^{-1}\mathbf{b}, \theta, \mathbf{A}^{-1}\mathbf{v}\big)] \tag{8}$$

for $\mathbf{v}^\intercal \frac{\partial f}{\partial \boldsymbol{\theta}} = \mathbf{v}^\intercal(\mathbf{A}^{-1})^\intercal(\partial_\theta \mathbf{A}_\theta)\mathbf{A}^{-1}\mathbf{b}$, and we will adopt this notation below for brevity. Doing so gives a memory cost which is constant in the number of solver iterations, and proportional to the memory used in the forward pass. Below we list the autograd rules for some of the iterative routines that we implement in CoLA with their `VJP` definitions.

1. $\mathbf{y} = \texttt{Solve}(\mathbf{A}, \mathbf{b})$: $\quad \mathbf{w}^\intercal \frac{\partial \mathbf{y}}{\partial \boldsymbol{\theta}} = -(\mathbf{A}^{-1}\mathbf{w})^\intercal(\partial_{\boldsymbol{\theta}} \mathbf{A}_{\boldsymbol{\theta}})(\mathbf{A}^{-1}\mathbf{b})$
2. $\boldsymbol{\lambda}, \mathbf{V} = \texttt{Eigs}(\mathbf{A})$: $\quad \mathbf{w}^\intercal \frac{\partial \boldsymbol{\lambda}}{\partial \boldsymbol{\theta}} = \mathbf{w}^\intercal \texttt{Diag}\big(\mathbf{V}^\intercal(\partial_{\boldsymbol{\theta}} \mathbf{A}_{\boldsymbol{\theta}})\mathbf{V}\big)$
3. $\boldsymbol{\lambda}, \mathbf{V} = \texttt{Eigs}(\mathbf{A})$: $\quad \mathbf{w}^\intercal \frac{\partial \mathbf{v}_i}{\partial \boldsymbol{\theta}} = \mathbf{w}^\intercal(\lambda_i \mathbf{I} - \mathbf{A})^+ \partial_{\boldsymbol{\theta}} \mathbf{A}_{\boldsymbol{\theta}} \mathbf{v}_i$
4. $y = \log|\mathbf{A}|$: $\quad \frac{\partial y}{\partial \boldsymbol{\theta}} = \texttt{Tr}\big(\mathbf{A}^{-1}\partial_{\boldsymbol{\theta}} \mathbf{A}_{\boldsymbol{\theta}}\big)$
5. $\mathbf{y} = \texttt{Diag}(\mathbf{A})$: $\quad \mathbf{w}^\intercal \frac{\partial \mathbf{y}}{\partial \boldsymbol{\theta}} = \mathbf{w}^\intercal \texttt{Diag}\left(\partial_{\boldsymbol{\theta}} \mathbf{A}_{\boldsymbol{\theta}}\right)$

In Figure 4 we show the practical benefits of our autograd rules. We take gradients of different linear solves $\mathbf{A}_{\boldsymbol{\theta}}^{-1}\mathbf{b}$ that were derived using conjugate gradients (CG), where each solve required an increasing number of CG iterations.

C Algorithmic Details

In this section we expand upon three different points introduced in the main paper. For the first point we argue why SVRG leads to gradients with reduced variants. For the second points we display all the iterative methods that we use as base algorithms in CoLA. Finally, for the third point we expand upon CoLA's strategy for dealing with the different numerical precisions that we support.

C.1 SVRG

In simplest form, SVRG [21] performs gradient descent with the varianced reduced gradient

$$\mathbf{w} \leftarrow \mathbf{w} - \eta(g_i(\mathbf{w}) - g_i(\mathbf{w}_0) + g(\mathbf{w}_0)) \tag{9}$$

where g_i represents the stochastic gradient evaluated at only a single element or minibatch of the sum, and $g(\mathbf{w}_0)$ is the full batch gradient evaluated at the anchor point $\mathbf{w}_0$ which is recomputed at the end of each epoch with an updated anchor.

With different loss functions, we can use this update rule to solve symmetric or asymmetric linear systems, to compute the top eigenvectors or even find the nullspace of a matrix. Despite the fact that the corresponding objectives are not strongly convex in the last two cases, it has been shown that gradient descent and thus SVRG will converge at this exponential rate [51, 14]. Below we list the gradients that enable us to solve different linear algebra problems: In each of the three cases listed

	Symmetric Solve $\mathbf{A}\mathbf{w} = \mathbf{b}$	Top-k Eigenvectors $\mathbf{A}\mathbf{W} = \mathbf{W}\mathbf{\Lambda}$	Nullspace $\mathbf{A}\mathbf{W} = 0$
$g_i(\mathbf{w})$	$\mathbf{A}_i\mathbf{w} - \mathbf{b}$	$-\mathbf{A}_i\mathbf{W} + \mathbf{W}\mathbf{W}^\intercal\mathbf{W}$ [51]	$\mathbf{A}_i\mathbf{W}$ [14]

Table 4: SVRG gradients for solving different linear algebra problems.

above, we can recognize that if the average of all the gradients $g(w)$ is 0, then the corresponding linear algebra solution has been recovered.

While it may seem that we need to take three complete passes through $\{\mathbf{A}_i\}$ per SVRG epoch (due to the three terms in Equation 9), we can reduce this cost to two complete passes exploiting the fact that the gradients are linear in the matrix object, replacing $\mathbf{A}_i\mathbf{W} - \mathbf{A}_i\mathbf{W}_0$ with $\mathbf{A}_i(\mathbf{W} - \mathbf{W}_0)$ where appropriate. In all of the `Sum` structure experiments where we leverage SVRG, the x-axis measures the total number of passes through $\{\mathbf{A}_i\}_{i=1}^m$, two for each epoch for SVRG.

C.2 Iterative methods

In Table 5 we list the different iterative methods (base cases) that we use for different linear algebraic operations as well as for different types of linear operators. As seen in Table 5, there are many alternatives to our base cases, however we opted for algorithms that are known to be performant, that are well-studied and that are popular amongst practitioners. A comprehensive explanation of our bases cases and their alternatives can be found in Golub and Loan [19] and Saad [40].

C.3 Lower precision linear algebra

The accumulation of round-off error is usually the breaking point of several numerical linear algebra (NLA) routines. As such, it is common to use precisions like `float64` or higher, especially when running these routines on a CPU. In contrast, in machine learning, lower precisions like `float32` or `float16` are ubiquitously used because more parameters and data can be fitted into the GPU memory (whose memory is usually much lower than CPUs) and because the MVMs can be done faster (the CUDA kernels are optimized for operations on these precisions). Additionally, the round-off error incurred on MVMs is not as detrimental when training machine learning models (as we are already running noisy optimization algorithms) as when solving linear algebra problems (where round-off error can lead us to poor solutions). Thus, it is an active area of research in NLA to derive routines which utilize lower precisions than `float64` or that mix precisions in order to achieve better runtimes without a complete degradation of the quality of the solution.

In CoLA we take a two prong approach to deal with lower precisions in our NLA routines. First, we incorporate additional variants of well-known algorithms that propagate less round-off error at the

Linear Algebra Op	Base Case	Alternatives
$\mathbf{Ax} = \mathbf{b}$ (asymmetric)	GMRES	BiCGSTAB, CR, QMR
$\mathbf{Ax} = \mathbf{b}$ (self-adjoint)	MINRES	GMRES
$\mathbf{Ax} = \mathbf{b}$ (PSD)	CG	GMRES
$\texttt{Eigs}(\mathbf{A})$ (asymmetric)	Arnoldi	IRAM, QR algorithm
$\texttt{Eigs}(\mathbf{A})$ (self-adjoint)	Lanczos	LOBPCG, Rayleigh-Ritz, Bi-Lanczos
$\mathbf{A}^+$	CG	CGS, LSQR, LGMRES
$\mathbf{A} = \mathbf{U\Sigma V}^*$	Lanczos, rSVD	Jacobi-Davidson
$f(\mathbf{A})$ (self-adjoint)	SLQ	SVD, Rational Krylov Subspaces

Table 5: **CoLA's base case iterative algorithm and some alternatives.** We now expand on the acronyms. GMRES: Generalized Minimum RESidual, BiCGSTAB: BiConjugate Gradient STABilized, CR: Conjugate Residuals, QMR: Quasi-Minimal Residual, MINRES: MINimum RESidual, CG: Conjugate Gradients, IRAM: Implicitly Restarted Arnoldi Method, LOBPCG: Locally Optimal Block Preconditioned Conjugate Gradients, Bi-Lanczos: Bidiagonal Lanczos, CGS: Conjugate Gradient Squared, LSQR: Least-Squares QR, LGMRES: Least-sqaures Generalized Minimum RESidual, SVD: Singular Value Decomposition, rSVD: randomized Singular Value Decomposition, and SLQ: Stochastic Lanczos Quadrature.

expense of requiring more computation, as seen in Figure 5. Second, we integrate novel variants of algorithms that are designed to be used on lower precisions such as the CG modification found in Maddox et al. [28]. We now discuss the first approach.

As discussed in Section C.2, there are two algorithms that are key for eigendecompositions. The first is Arnoldi (applicable to any operator), and the second is Lanczos (for symmetric operators) — where actually Lanczos can be viewed as a simplified version of Arnoldi. Central to these algorithms is the use of an orthogonalization step which is well-known to be a source of numerical instability. One approach to aggressively ameliorate the propagation of round-off error during orthogonalization is to use Householder projectors, which is the strategy that we use in CoLA. Given a unitary vector $\mathbf{u}$, a Householder projector (or Householder reflector) is defined as the following operator $\mathbf{R} = \mathbf{I} - 2\mathbf{u}\mathbf{u}^*$. When applied to a vector $\mathbf{x}$ the result $\mathbf{Rx}$ is basically a reflection of $\mathbf{x}$ over the $\mathbf{u}^\intercal$ space. To easily visualize this, suppose that $\mathbf{x} \in \mathbb{R}^2$ and $\mathbf{u} = \mathbf{e}_1$. Hence,

$$\mathbf{Rx} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - 2\begin{pmatrix} x_1 \\ 0 \end{pmatrix} = \begin{pmatrix} -x_1 \\ x_2 \end{pmatrix}$$

which is exactly the reflection of the vector across the axis generated by $\mathbf{e}_2$. Most notably, $\mathbf{R}$ is unitary $\mathbf{RR}^* = \mathbf{I}$ which can be easily verified from the definition. Being unitary is crucial as under the usual round-off error model, applying $\mathbf{R}$ to another matrix $\mathbf{A}$ does not worsen the already accumulated error $\mathbf{E}$. Mathematically, $\|\mathbf{R}(\mathbf{A} + \mathbf{E}) - \mathbf{RA}\| = \|\mathbf{RE}\| = \|\mathbf{E}\|$, where the last equality results from basic properties of unitary matrices. We are going to use Arnoldi as an example of how Householder projectors are used during orthogonalization. In Figure 5 we have an example of two different variants of Arnoldi present in CoLA. The implementations are notably different and also it is easy to see how Algorithm 2 is more expensive than Algorithm 1. First, note that for Algorithm 2 we have two for loops (line 6 and line 8) whereas for Algorithm 1 we only have one (line 4-6). Worse, the two for loops in Algorithm 2 require more flops than the only for loop in Algorithm 1. Note that we do not always favor the more expensive but robust implementation of an algorithm as in some cases, like when running GMRES, the round-off error is not as impactful to the quality of the solution, and shorter runtimes are actually more desirable.

D Experimental Details

In this section we expand upon the details of all the experiments ran in the paper. Such details include the datasets that were used, the hyperparameters of different algorithms and the specific choices

Algorithm 1 Arnoldi iteration

Inputs: $\mathbf{A}$, $\mathbf{q}_0 = \boldsymbol{\nu}_0 / \|\boldsymbol{\nu}_0\|$ where possibly $\boldsymbol{\nu}_0 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, maximum number of iterations T and tolerance $\epsilon \in (0, 1)$.
for $j = 0$ to $T - 1$ **do**
  $\boldsymbol{\nu}_{j+1} \leftarrow \mathbf{A}\mathbf{q}_j$
  for $i = 0$ to j **do**
    $h_{i,j} = \mathbf{q}_i^*(\mathbf{A}\mathbf{q}_j)$
    $\boldsymbol{\nu}_{j+1} \leftarrow \boldsymbol{\nu}_{j+1} - h_{i,j}\mathbf{q}_i$
  end for
  $h_{j+1,j} = \|\boldsymbol{\nu}_{j+1}\|$
  if $h_{j+1,j} < \epsilon$ **then**
    stop
  else
    $\mathbf{q}_{j+1} = \boldsymbol{\nu}_{j+1}/h_{j+1,j}$
  end if
end for
return $\mathbf{H}, \mathbf{Q} = \left(\mathbf{q}_0 | \dots | \mathbf{q}_{T-1} | \mathbf{q}_T\right)$

Algorithm 2 Householder Arnoldi iteration

Inputs: $\mathbf{A}$, $\boldsymbol{\nu}_0 \neq \mathbf{0}$ where possibly $\boldsymbol{\nu}_0 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, and maximum number of iterations T.
for $j = 0$ to T **do**
  $\mathbf{u}_j = \texttt{GET\_HOUSEHOLDER\_VEC}(\boldsymbol{\nu}_j, j)$
  $\mathbf{R}_j = \mathbf{I} - 2\mathbf{u}_j\mathbf{u}_j^*$
  $\mathbf{h}_j = \mathbf{R}_j\boldsymbol{\nu}_j$
  $\mathbf{q}_j = \mathbf{R}_0 \cdots \mathbf{R}_j \mathbf{e}_{j+1}$
  if $j < T$ **then**
    $\boldsymbol{\nu}_{j+1} = \mathbf{R}_j \cdots \mathbf{R}_0(\mathbf{A}\mathbf{q}_j)$
  end if
end for
return $\mathbf{H}, \mathbf{Q} = (\mathbf{q}_0 | \dots | \mathbf{q}_T)$
function GET_HOUSEHOLDER_VEC($\mathbf{w}$, k)
  $u_i = 0$ for $i < k$ and $u_i = w_i$ for $i > k$.
  $u_k = w_k - \|\mathbf{w}\|$
  return $\mathbf{u}$
end function

Figure 5: Different versions of the same algorithm, but the Householder variant being more numerically robust.

of algorithms used both for CoLA but also for the alternatives. We run each of the experiments 3 times and compute the mean dropping the first observation (as usually the first run contains some compiling time much is not too large). We do not display the standard deviation as those numbers are imperceptible for each experiment. In terms of hardware, the CPU experiments were run on an Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz and the GPU experiments were run on a NVIDIA GeForce RTX 2080 Ti.

D.1 Datasets

Below we enumerate the datasets that we used in the various applications. Most of the datasets are sourced from the University of California at Irvine's (UCI) Machine Learning Respository that can be found here: https://archive.ics.uci.edu/ml/datasets.php. Also, a community repo hosting these UCI benchmarks can be found here: https://github.com/treforevans/uci_datasets (we have no affiliation).

1. *Elevators*. This dataset is a modified version of the *Ailerons* dataset, where the goal is to to predict the control action on the ailerons of the aircraft. This UCI dataset consists of $N = 14$K observations and has $D = 18$ dimensions.
2. *Kin40K*. The full name of this UCI dataset is *Statlog (Shuttle) Data Set*. This dataset contains information about NASA shuttle flights and we used a subset that consists of $N = 40$K observations and has $D = 8$ dimensions.
3. *Buzz*. The full name of this UCI dataset is *Buzz in social media*. This dataset consists of examples of buzz events from Twitter and Tom's Hardware. We used a subset consisting of $N = 430$K observations and has $D = 77$ dimensions.
4. *Song*. The full name of this UCI dataset is *YearPredictionMSD*. This dataset consists of $N = 386.5$K observations and it has $D = 90$ audio features such as 12 timbre average features and 78 timbre covariance features.
5. *cit-HepPh*. This dataset is based on arXiv's HEP-PH (high energy physics phenomenology) citation graph and can be found here: https://snap.stanford.edu/data/cit-HepPh.html. The dataset covers all the citations from January 1993 to April 2003 of $|V| = 34,549$ papers, ultimately containing $|E| = 421,578$ directed edges. The notion of relationship that we used in our spectral clustering experiment creates a connection between two papers

when at least one cites another (undirected symmetric graph). Therefore the dataset that we used has the same number of nodes but instead $|E| = 841,798$ undirected edges.

D.2 Compositional experiments

This section pertains to the experiments of Section 3.2 displayed in Figure 1. We now elaborate on each of Figure 1's panels.

(a) The multi-task GP problem exploits the structure of the following Kronecker operator $\mathbf{K}_T \otimes \mathbf{K}_X$, where $\mathbf{K}_T$ is a kernel matrix containing the correlation between the tasks and $\mathbf{K}_X$ is a RBF kernel on the data. For this experiment, we used a synthetic Gaussian dataset where the train data $\mathbf{x}_i \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_D)$ which has dimension $D = 33$, $N = 1$K and we used $T = 11$ tasks (where the tasks basically set the size of $\mathbf{K}_T$). We used conjugate gradients (CG) as the iterative method, where we set the hyperparameters to a tolerance of 10^{-6} and to a maximum number of iterations to 1K. We used the exact same hyperparameters for CoLA.

(b) For the bi-poisson problem we set up the maximum grid to be $N = 1000^2$. Since this PDE problem involves solving a symmetric linear system, we used CG as the iterative method with a tolerance of 10^{-11} and a maximum number of iterations of 10K. The previous parameters also apply for CoLA. We note that PDE problems are usually solved to higher tolerances as the numerical error compounds as we advance the PDE.

(c) For the EMLP experiment we consider solving the equivariance constraints to find the equivariant linear layers of a graph neural network with 5 nodes. To solve this problem, we need to find the nullspace of a large structured constraint matrix. We use the uniformly channel heuristic from [14] which distributes the N channels across tensors of different orders. We consider our approach which exploits the block diagonal structure, separating the nullspaces into blocks, as opposed to the direct iterative approach exploiting only the fast MVMs of the constraint matrix. We use a tolerance of 10^{-5}.

D.3 Sum structure experiments

This section pertains to the experiments of Section 3.3 contained in Figure 2. We now elaborate on each of Figure 2's panels.

(a) In this experiment we computed the first principal component of the *Buzz* dataset. For the iterative method we used power iteration with a maximum number of iterations of 300 and a stop tolerance of 10^{-7}. CoLA used SVRG also with the same stop tolerance and maximum number of iterations. Additionally, we set SVRG's batch size to 10K and the learning rate to 0.0008. We note that a single power iteration roughly contains $43/2 = 21.5$ times more MVMs than a single iteration of SVRG. In this particular case, the length of the sum is given by the number of observations and therefore SVRG uses $430/10 = 43$ times less elements per iteration, where 10 comes from the 10K batch size. Finally, the 2 is explained by noting that SVRG incurs in a full sum update on every epoch.

(b) In this experiment we trained a GP by estimating the covariance RBF kernel with $J = 1$K random Fourier features (RFFs). The hyperparameters for the RBF kernel are the following: length scale ($\ell = 0.1$), output scale ($a = 1$) and likelihood noise ($\sigma^2 = 0.1$). Moreover, we used CG as the iterative solver with a tolerance of 10^{-8} and 100 as the maximum number of iterations (the convergence took much less iterations than the max). For SVRG we used the same tolerance but set the maximum number of iterations to 10K, a batch size of 100 and learning rate of 0.004. We note that a single CG iteration roughly contains $10/2 = 5$ times more MVMs than a single iteration of SVRG. In this particular case, the length of the sum is given by the number of RFFs and therefore SVRG uses $1000/100 = 10$ times less elements per iteration, where 100 comes from the batch size.

(c) In this experiment we implemented the Neural-IVP method from Finzi et al. [15]. We consider the time evolution of a wave equation in two spatial dimensions. At each integrator step, a linear system $\mathbf{M}(\theta)\dot{\theta} = F(\theta)$ must be solved to find $\dot{\theta}$, for a $d = 12\text{K} \times 12\text{K}$ dimensional matrix. While Finzi et al. [15] use conjugate gradients to solve the linear system, we demonstrate the advantages of using SVRG, as $\mathbf{M}(\theta) = \frac{1}{m}\sum_{i=1}^{m} M_i(\theta)$ is a

sum over the evaluation at $m = 50$K distinct sample locations within the domain. In this experiment we use a batch size of 500 for SVRG, and employ rank 250 randomized Nyström preconditioning for both SVRG and the iterative CG baseline.

D.4 Applications

This section pertains to the experiments of Section 4 displayed in Figure 3. We now elaborate on each of Figure 3's panels.

(a) In this experiment we compute 5, 10 and 20 PCA components for the *Buzz* dataset. We compared against `sklearn` which uses the Lanczos algorithm through the fast Fortran-based `ARPACK` numerical library. In this case, CoLA uses randomized SVD [31] with a rank 3000 approximation.

(b) In this experiment we fit a Ridge regression on the *Song* dataset with a regularization coefficient set to 0.1. We compared against `sklearn` using their fastest least-square solver `lsqr` with a tolerance of 10^{-4}. In this case, CoLA uses CG with the same tolerance and with a maximum number of iterations set to 1K. Additionally, we ran CoLA using CPU and GPU whereas we used only CPU for `sklearn` as it has no GPU support. We observe how in the arguably most popular ML method, CoLA is able to beat a leading package such as `sklearn`.

(c) In this experiment we fit a GP with a RBF kernel on two datasets: *Elevators* and *Kin40K*. We only used up to 20K observations from *Kin40K* as that was the maximum number of observations that would fit the GPU memory without needing to partition the MVMs. We compare against `GPyTorch` which uses CG and stochastic Lanczos quadrature (SLQ) to compute and optimize the negative log-marginal likelihood (loss function). Both experiments were run on a GPU for 100 iterations using Adam as an optimizer with learning rate of 0.1 with the default values of $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Additionally, for both GPyTorch and CoLA, the CG tolerance was set to 10^{-4} with a maximum number of CG iterations of 250 and 20 probes were used for SLQ. Note that both CoLA and GPyTorch have similar throughputs, for example GPyTorch runs a 100 iterations on *Elevators* on 43 seconds whereas CoLA runs a 100 iterations on 49 seconds. When training a GP, we solve a block of 11 linear systems (1 based on $\mathbf{y}$ and 10 based on random probes) where one key difference is that the CG solver for GPyTorch has a stopping criteria based on the convergence of the mean solves whereas CoLA has a stopping criteria based on the convergence of all the solves.

(d) In this experiment we run spectral clustering on the *cit-HepPh* dataset using an embedding size of 8 and also 8 clusters for k-means (with only 1 run of k-means after estimating the embeddings). We compare against `sklearn` using two different solvers, one based on Lanczos iterations using `ARPACK` and another using an Algebraic Multi-Graph solver `AMG`. In this case, CoLA also uses Lanczos iterations with a default tolerance of 10^{-6}. We see how `sklearn`'s `AMG` solver runs faster than CoLA's but this is mostly the algorithmic constants as they have similar asymptotical behavior (similar slopes).

(e) In this experiment we solve the Schrödinger equation to find the energy levels of the hydrogen atom on a 3-dimensional finite difference grid with up to $N = 5$K points. In order to handle the infinite spatial extent, we compactify the domain by applying the arctan function. Under this change of coordinates, the Laplacian has a different form, and hence the matrix forming the discretized Hamiltonian is no longer symmetric. We compare against `SciPy`'s Arnoldi implementation with 20 iterations where CoLA also uses Arnoldi with the same number of iterations. Surprisingly, CoLA's JAX jitted code has a competitive runtime when compare to `SciPy`'s runtime using `ARPACK`.

(f) In this experiment we solve a minimal surface problem on a grid of maximum size of $N = 100^2$ points. To solve this problem we have to run Netwon-Rhapson where each inner step involves a linear solve of an asymmetric operator. We compare against `SciPy`'s GMRES implementation as well as `JAX`'s integrated version of `SciPy`. The main difference between the two is that `SciPy` calls the fast and highly-optimized `ARPACK` library whereas `SciPy (JAX)` has its only `Python` implementation of GMRES which only uses `JAX`'s primitives (equally as it is done in CoLA). The tolerance for this experiment was 5e-3.

We see how CoLA's GMRES implementation is competitive with `SciPy (JAX)` but it still does not beat `ARPACK` mostly due to the faster runtime of using a lower level GMRES implementation.

E Code examples

We now show how our dispatch rules are implemented in CoLA. Specifically we show the implementation of our inverse rule.

```
from plum import dispatch

@dispatch
def inverse(A: LinearOperator, **kwargs):
    kws = dict(tol=1e-6, P=None, x0=None, pbar=False, info=False,
        max_iters=5000)
    kws.update(kwargs)
    method = kws.pop('method', 'auto')
    if method == 'dense' or (method == 'auto' and np.prod(A.shape) <=
        1e6):
        return A.ops.inv(A.to_dense())
    if issubclass(type(A), SelfAdjoint[Sum]) and (method == 'svrg' or
        (method == 'auto' and len(A.A.Ms) > 1e4)):
        return SymmetricSVRGInverse(A.A, **kws)
    if issubclass(type(A), Sum) and (method == 'svrg' or (method == '
        auto' and len(A.Ms) > 1e4)):
        return GenericSVRGInverse(A, **kws)
    if issubclass(type(A), SelfAdjoint) and (method == 'cg' or (method
         == 'auto' and np.prod(A.shape) > 1e6)):
        return CGInverse(A, **kws)
    if method == 'gmres' or (method == 'auto' and np.prod(A.shape) > 1
        e6):
        return GMResInverse(A, **kws)
    else:
        raise ValueError(f"Unknown method {method} or CoLA didn't fit
            any selection criteria")

@dispatch
def inverse(A: Identity, **kwargs):
    return A


@dispatch
def inverse(A: ScalarMul, **kwargs) -> ScalarMul:
    return ScalarMul(1 / A.c, shape=A.shape, dtype=A.dtype)


@dispatch
def inverse(A: Product, **kwargs) -> Product:
    output = [inverse(M, **kwargs) for M in A.Ms].reverse()
    return Product(*output)


@dispatch
def inverse(A: BlockDiag, **kwargs) -> BlockDiag:
    return BlockDiag(*[inverse(M, **kwargs) for M in A.Ms],
        multiplicities=A.multiplicities)


@dispatch
def inverse(A: Kronecker, **kwargs) -> Kronecker:
    return Kronecker(*[inverse(M, **kwargs) for M in A.Ms])


@dispatch
```

```python
def inverse(A: Diagonal, **kwargs) -> Diagonal:
    return Diagonal(1. / A.diag)


@dispatch
def inverse(A: Unitary, **kwargs) -> Unitary:
    return Unitary(A.H)
```