Position: Curvature Matrices Should Be Democratized via Linear Operators

Felix Dangel [* 1]   Runa Eschenhagen [* 2]   Weronika Ormaniec [3]   Andres Fernandez [4]   Lukas Tatzel [4]
Agustinus Kristiadi [1]

Abstract

Structured large matrices are prevalent in machine learning. A particularly important class is curvature matrices like the Hessian, which are central to understanding the loss landscape of neural nets (NNs), and enable second-order optimization, uncertainty quantification, model pruning, data attribution, and more. However, curvature computations can be challenging due to the complexity of automatic differentiation, and the variety and structural assumptions of curvature proxies, like sparsity and Kronecker factorization. In this **position paper**, we argue that **linear operators—an interface for performing matrix-vector products—provide a general, scalable, and user-friendly abstraction to handle curvature matrices**. To support this position, we developed `curvlinops`, a library that provides curvature matrices through a unified linear operator interface. We demonstrate with `curvlinops` how this interface can hide complexity, simplify applications, be extensible and interoperable with other libraries, and scale to large NNs.

1. Introduction

Structured matrices in ML. Large matrices that exhibit structure play an important role in various machine learning (ML) applications: *Sparse* matrices contain only a small fraction of non-zero entries, and are, e.g., common in learning problems on graphs (Kipf & Welling, 2017) whose adjacency matrices are sparse. *Symbolic* matrices (Charlier et al., 2021) are matrices whose coefficients are specified through a mathematical expression (say $\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2$ or $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$) and include kernel matrices used in Gaussian Processes (Williams & Rasmussen, 2006), distance matrices used for k-means clustering (Steinhaus et al., 1956)

or k-NN classification/regression (Fix & Hodges, 1951), and attention (Vaswani et al., 2017) matrices. *Factorized* matrices consist of one or multiple low-dimensional, and therefore parameter-efficient, tensor components that are combined via tensor multiplications. Examples include diagonal, block-diagonal, Kronecker (Loan, 2000), outer product (or low rank, Jacot et al., 2018; Ren & Goldfarb, 2019; Dangel et al., 2022), (block) tensor-train, and Monarch (Dao et al., 2022) matrices, but can also be generalized to unnamed factorizations using tensor networks (Potapczynski et al., 2024; Penrose, 1971; Bridgeman & Chubb, 2017; Biamonte & Bergholm, 2017). Other 'low-complexity' structures are butterfly (Dao et al., 2019), Toeplitz (or circulant), low-rank plus diagonal (Lin et al., 2024c), or hierarchical matrices (Chen et al., 2022). These structures allow executing operations like matrix-vector products *matrix-free*, i.e. without materializing the dense matrix.
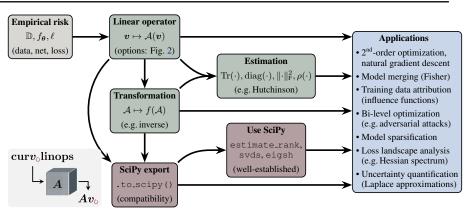
Linear operators. In many scenarios, it suffices to query matrix-vector products, rather than access a dense matrix representation. Matrix-free structures are useful to accelerate such matrix-vector products for solving, or approximately solving, numerical tasks and to reduce memory demands. This motivates *linear operators*, an abstraction for carrying out black-box matrix-vector multiplications. More formally, a linear operator $\mathcal{A}$ is a linear map between two vector spaces, $\mathcal{A} : \mathbb{R}^{D_{\text{in}}} \to \mathbb{R}^{D_{\text{out}}}$, $\boldsymbol{v} \mapsto \mathcal{A}(\boldsymbol{v})$, such that for $\boldsymbol{v}_1, \boldsymbol{v}_2 \in \mathbb{R}^{D_{\text{in}}}$ and $\alpha \in \mathbb{R}$,

$$\mathcal{A}(\boldsymbol{v}_1 + \boldsymbol{v}_2) = \mathcal{A}(\boldsymbol{v}_1) + \mathcal{A}(\boldsymbol{v}_2),$$
$$\mathcal{A}(\alpha\boldsymbol{v}) = \alpha\mathcal{A}(\boldsymbol{v}).$$

We can think of $\mathcal{A}$ as *action* of a matrix $\boldsymbol{A} \in \mathbb{R}^{D_{\text{out}} \times D_{\text{in}}}$ with entries $[\boldsymbol{A}]_{i,j} = \boldsymbol{e}_i^\top \mathcal{A}(\boldsymbol{e}_j)$ with $\boldsymbol{e}_i$ the i-th standard basis vector. While $\mathcal{A}(\boldsymbol{v}) = \boldsymbol{A}\boldsymbol{v}$, the linear operator $\mathcal{A}$ may not construct the full matrix representation $\boldsymbol{A}$, which could be costly. For example, if $\boldsymbol{A} = \text{diag}(\boldsymbol{a})$ is a diagonal matrix, the matrix-vector product can be performed via element-wise multiplication, $\mathcal{A} : \boldsymbol{v} \mapsto \boldsymbol{A}\boldsymbol{v} = \boldsymbol{a} \odot \boldsymbol{v}$, i.e. *without* forming $\boldsymbol{A}$ in memory. Since matrix-vector multiplications with a structured matrix can be done more efficiently than explicit multiplication with the dense matrix representation, linear operators facilitate working with large matrices without the need for explicit storage.

---
[*]Equal contribution   [1]Vector Institute, Toronto, Canada [2]Cambridge University, United Kingdom [3]ETH, Zürich, Switzerland [4]Tübingen AI Center, University of Tübingen, Tübingen, Germany. Correspondence to: Felix Dangel <fdangel@vectorinstitute.ai>.

arXiv:2501.19183v1 [cs.LG] 31 Jan 2025

Figure 1. **Overview of `curvlinops`'s features.** At its core, the library provides linear operators for curvature matrices of an empirical risk in PyTorch. Operator transformations allow representing matrix functions like inversion. To analyze the underlying matrices, we implement known estimation techniques based on randomized linear algebra. Other routines are already provided in SciPy, and they become accessible through `curvlinops`'s export feature.

The fact that structured linear maps can often be implemented more efficiently in both time and memory has spurred the development of iterative algorithms for estimating the properties of said map, solving linear systems, or computing truncated matrix decompositions from matrix-vector products. The relevance of these topics for the ML community has been underlined by tutorials at conferences like NeurIPS 2023 (Dereziński & Mahoney, 2024).

Curvature matrices in deep learning. Curvature matrices like the Hessian of the empirical risk or approximations thereof play a crucial role in describing the loss landscape of neural nets. They are a necessary ingredient for many applications, e.g. second-order optimization (Martens & Sutskever, 2012; Amari, 2000), uncertainty quantification (MacKay, 1992; Daxberger et al., 2021), model merging (Matena & Raffel, 2022), model pruning (LeCun et al., 1989; Singh & Alistarh, 2020), empirical investigations into the loss landscape (Schneider et al., 2021; Tatzel et al., 2024; Papyan, 2019b; Gur-Ari et al., 2018; Yao et al., 2020), and computing hyper-gradients for training data attribution (Koh & Liang, 2017), adversarial attacks (Lu et al., 2022), or hyper-parameter optimization (Lorraine et al., 2020).

The parameter space of NNs is often so vast that storing even a few columns of these matrices becomes prohibitively expensive. Additionally, these matrices are often defined on large datasets, necessitating their computation in manageable batches. Thanks to recent developments of ML libraries (Bradbury et al., 2018; Paszke et al., 2019), many curvature matrices (e.g. the exact Hessian) have become more accessible. However, matrix-vector products can still be prohibitively costly (equivalents of multiple gradient evaluations (Dagréou et al., 2024)). As a result, practical curvature matrices often depend on structural approximations to be feasible, e.g. Kronecker factorizations (Heskes, 2000; Martens & Grosse, 2015). These approximations can greatly reduce computational costs while still being useful for applications. However, implementing and testing these approximations can be extremely challenging, as they come in various forms and use many heuristics.

In this paper, we argue **our position** that;

> *Presenting curvature matrices as linear operators empowers users to apply them to various applications without worrying about implementation complexity, and to benefit from their extensibility and interoperability.*

To substantiate our position, we developed `curvlinops`[1], a PyTorch library that offers easy access to various curvature matrices as linear operators. We divide our argument in favor of linear operators into three statements and illustrate them through `curvlinops` (overview in Fig. 1):

1. **They encapsulate complexity (§2).** We can cover a wide range of curvature matrices—from exact but slow to approximate but fast—in one interface (Fig. 2). This allows users to focus on their application, liberating them from having to comprehend the numerical details and error-prone implementation intricacies such as correctly scaled accumulation over batches and potential non-deterministic outputs.

2. **They simplify applications (§3).** Because linear operators can be multiplied with vectors like dense matrices, they are easy to use. Using code snippets, we show how to implement a diverse set of applications that demand curvature approximations (second-order optimization, model merging, spectral analysis, . . .), demonstrating that the code aligns with the underlying mathematics.

3. **They enable extensibility and interoperability (§4).** We discuss how new curvature approximations can be conveniently implemented as linear operators. Moreover, exporting linear operators to SciPy allows the use of its sophisticated algorithmic toolbox for truncated eigendecomposition, SVD, or rank estimation. To complement these tools, we implement various (randomized) linear algebra algorithms to estimate matrix properties, such as trace, diagonal, and spectral density.

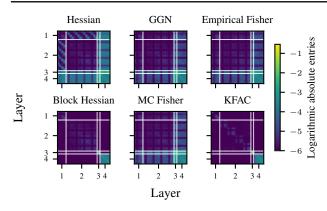[1]`pip install curvlinops-for-pytorch`

Figure 2. **Visual tour of curvature matrices.** White lines separate parameters into layers. We consider a synthetic classification task with a small convolutional neural net (three convolutional and one dense layer with ReLU and sigmoid activations, $D = 683$).

We also demonstrate scalability to larger architectures like nanoGPT on Shakespeare and ResNet50 on ImageNet (§A).

Related software. Various Python packages provide a linear operator interface: PyLops (Ravasi & Vasconcelos, 2019) focuses on creating and solving inverse problems; the linear operator class in SciPy (Virtanen et al., 2020) lives in the sparse sub-module. ML frameworks like PyTorch (Paszke et al., 2019) and JAX (Bradbury et al., 2018) do not provide linear operator interfaces, but there are external packages that do so (Rader et al., 2023; Gardner et al., 2018). Potapczynski et al. (2023); Gardner et al. (2018) additionally incorporate many linear algebra tricks for efficiently solving linear systems, inverting or composing matrices, and computing properties like eigenvalues/spectra.

While there have been efforts to simplify curvature computations (Golmant et al., 2018; Granziol et al., 2019; Yao et al., 2020; Dangel et al., 2020; Osawa et al., 2023), achieving support for diverse architectures and extensibility remains challenging. Our incentive is to unify a wider variety of curvature matrices than previous approaches, and to further disentangle matrix property estimation techniques from matrix-vector products through the linear operator interface.

2. Linear Operators Encapsulate the Complexity of Curvature Matrices

Curvature matrices come in various flavors, some of which require knowledge of automatic differentiation (AD), the implementation of loss-specific sampling procedures, or the computation of architecture-specific components such as Kronecker factors. In addition, computations are usually done in batches and accumulated, which requires adapting the scaling of each result. Also, the vector in a matrix-vector product can be interpreted as either a single vector or a tensor product. Lastly, common deep learning elements

like data augmentation, dropout (Srivastava et al., 2014), and data-order dependent concepts like batch normalization (Ioffe & Szegedy, 2015) render the represented matrix non-deterministic, which is undesirable. All these caveats and inconveniences can be encapsulated in a linear operator.

We focus on curvature information of an empirical risk $\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})$ from a neural net $f_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta} \in \mathbb{R}^D$, whose predictions $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ on datum $\boldsymbol{x}$ are scored on some data set $\mathbb{D}$ using a convex criterion function ℓ and the true label $\boldsymbol{y}$,

$$\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta}) \coloneqq R_{\mathbb{D}} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{D}} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n), \quad (1)$$

where $R_{\mathbb{D}}$ is a reduction factor, usually a value from $\{1, 1/|\mathbb{D}|, 1/|\mathbb{D}|\dim(\boldsymbol{f}_n)\}$, where $\boldsymbol{f}_n \coloneqq f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$.

2.1. Curvature Matrices

We provide a quick overview of fundamental curvature matrices like the Hessian, generalized Gauss-Newton (GGN), and different flavors of the Fisher information matrix (Fisher), all of which can be represented using the linear operator interface; see Fig. 2 for a visual tour.

Hessian. The empirical risk's Hessian

$$\boldsymbol{H}(\boldsymbol{\theta}) \coloneqq \nabla_{\boldsymbol{\theta}}^2 \mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta}) = R_{\mathbb{D}} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{D}} \nabla_{\boldsymbol{\theta}}^2 \ell(\boldsymbol{f}_n, \boldsymbol{y}_n) \quad (2)$$

collects the second-order derivatives and is thus the most natural choice for the curvature matrix. While the Hessian is prohibitively large to store, Hessian-vector products (HVPs) with a vector $\boldsymbol{v}$ can be executed via nested first-order AD (Pearlmutter, 1994), as $\nabla_{\boldsymbol{\theta}}^2 \mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})\boldsymbol{v} = \nabla_{\boldsymbol{\theta}}(\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})^\top \boldsymbol{v})$. Dagréou et al. (2024) empirically probed the efficiency of HVPs and found that they require 2-4x as long as a gradient and consume 2-3x the memory, depending on the usage of just-in-time compilation, and the combination of reverse and forward mode AD. Thanks to the linear operator interface, users are not concerned with the exact HVP implementation.

GGN. One issue with the Hessian is that it is usually indefinite. The GGN $\boldsymbol{G}(\boldsymbol{\theta})$ is a positive semi-definite approximation of the Hessian that follows from *partial linearization* of the composition $\ell \circ f_{\boldsymbol{\theta}}$ by replacing $f_{\boldsymbol{\theta}}$ with a linearization before differentiation,

$$\boldsymbol{G}(\boldsymbol{\theta}) \coloneqq R_{\mathbb{D}} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{D}} \mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{f}_n^\top \nabla_{\boldsymbol{f}_n}^2 \ell(\boldsymbol{f}_n, \boldsymbol{y}_n) \, \mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{f}_n, \quad (3)$$

with the Jacobian $[\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{f}_n]_{i,j} = \partial[\boldsymbol{f}_n]_i/\partial[\boldsymbol{\theta}]_j$. The GGN's positive semi-definite nature is desirable for many applications that demand a convex local quadratic, and GGNVPs can be done with HVPs, VJPs, and JVPs (Schraudolph, 2002). Martens (2010) finds that GGNVPs only take about half the memory and run nearly twice as fast as HVPs. In many tasks, the GGN corresponds to the Fisher (Martens, 2020).

Probabilistic interpretation. Empirical risk minimization often amounts to maximizing the likelihood $\hat{p}(\{y_n\}_{\mathbb{D}} \mid \{x_n\}_{\mathbb{D}}, \theta) = \prod_n p(y_n \mid x_n, \theta)$—or equivalently minimizing the negative log likelihood—where the neural net parameterizes a likelihood $p(y \mid x, \theta) = q(y \mid f)$ such that $\ell(f, y) = -\log q(y \mid f)$. E.g., for square loss, the net's likelihood is a Gaussian, and for softmax cross-entropy loss a categorical distribution. This allows the introduction of an alternative notion of distance based on the KL divergence between the likelihoods implied by two models f_θ and $f_{\theta+\delta}$. The metric of this statistical manifold is the Fisher (Amari, 2000). Following Soen & Sun (2024), we consider two common forms of the Fisher. Here we describe the type-I Fisher which we simply denote as Fisher. The overview of type-II Fisher is in §B.

Fisher. The Fisher matrix is the log likelihood's gradient covariance w.r.t. the model's distribution q. Denoting the log-likelihood as $l_n := \log q(y \mid f_n)$, we have

$$F_{\mathrm{I}}(\theta) := R_{\mathbb{D}} \sum_{x_n \in \mathbb{D}} \mathbb{E}_{q(y|f_n)} \left[\nabla_\theta l_n \left(\nabla_\theta l_n \right)^\top \right] \quad (4)$$

$$= R_{\mathbb{D}} \sum_{x_n \in \mathbb{D}} \mathrm{J}_\theta f_n^\top \mathbb{E}_{q(y|f_n)} \left[\nabla_{f_n} l_n \left(\nabla_{f_n} l_n \right)^\top \right] \mathrm{J}_\theta f_n .$$

Monte-Carlo Fisher. In practice, we approximate the expectation in (4) through Monte-Carlo (MC) sampling. To this end, let $\hat{y}_{n,s} \overset{\text{i.i.d.}}{\sim} q(y \mid f_n)$ and denote $\hat{l}_{n,s} := \log q(y = \hat{y}_{n,s} \mid f_n)$ the would-be log-likelihood on that sample. Then, the MC-estimated Fisher with S samples is

$$\hat{F}_{\mathrm{I}}(\theta) = R_{\mathbb{D}}/S \sum_{x_n \in \mathbb{D}} \sum_{s=1}^{S} \nabla_\theta \hat{l}_{n,s} \left(\nabla_\theta \hat{l}_{n,s} \right)^\top . \quad (5)$$

Empirical Fisher (gradient covariance). Another popular modification of (4) is to replace the expectation over the model's likelihood $q(y \mid f_n)$ with the empirical likelihood implied by the data, $q_{\mathrm{emp}}(y \mid f_n) = \delta(y - y_n)$ where y_n is the true label. This yields the empirical Fisher,

$$E(\theta) := R_{\mathbb{D}} \sum_{x_n \in \mathbb{D}} \mathbb{E}_{q_{\mathrm{emp}}(y|f_n)} \left[\nabla_\theta l_n \left(\nabla_\theta l_n \right)^\top \right]$$

$$= R_{\mathbb{D}} \sum_{(x_n, y_n) \in \mathbb{D}} \nabla_\theta \ell(f_n, y_n) \left(\nabla_\theta \ell(f_n, y_n) \right)^\top , \quad (6)$$

the (scaled) un-centered gradient covariance w.r.t. the empirical data distribution. The empirical Fisher differs from the Fisher (Kunstner et al., 2019), but is popular in applications as it re-uses information from gradient backpropagation.

Fisher-vector products (FVPs) via GGNVPs. The naïve approach to multiply with the Fisher and empirical Fisher is to compute the (would-be) per-datum gradients. However, they require $|\mathbb{D}| \times D$ additional memory. Instead, curvlinops utilizes the similarity of these matrices to

the GGN: FVPs are done via GGNVPs on a pseudo-loss, which is more efficient. To see this, note that the last lines of (5) & (6) resemble the GGN in (3). Construct the pseudo-loss $\tilde{\ell}(f_n, y_n) = f_n^\top g_n g_n^\top f_n$ where g_n is a detached vector whose value is set to $\nabla_{f_n} \log q(\hat{y}_{n,s} \mid f_n)$ or $\nabla_{f_n} \ell(f_n, y_n)$. The GGN of that pseudo-loss then corresponds to the desired Fisher, as $\nabla_{f_n}^2 \tilde{\ell}(f_n, y_n) = g_n g_n^\top$ in (3). The linear operator interface used in curvlinops abstracts away these implementation details, allowing the user to focus on higher-level tasks without worrying about the specifics.

Kronecker-Factored Approximate Curvature (KFAC). KFAC (Heskes, 2000; Martens & Grosse, 2015) is a lightweight parametric curvature proxy that uses Kronecker products to approximate per-layer curvature matrices, i.e. let $\theta^{(l)}$ be the parameters residing in layer l of a network, then KFAC approximates the curvature matrix $C(\theta^{(l)}) \approx A^{(l)} \otimes B^{(l)}$ where $A^{(l)}$ is computed from the layer's input, and $B^{(l)}$ from (would-be) gradients w.r.t. to the layer's output. KFAC applies to the GGN (Botev et al., 2017), MC, and empirical Fisher and a broad range of neural nets (Grosse & Martens, 2016; Martens et al., 2018; Eschenhagen et al., 2023). It was further improved through eigenvalue correction (EKFAC, George et al., 2018). Implementing these variations and ensuring their correctness is highly complex.

curvlinops provides linear operators for the Hessian, GGN, MC Fisher, empirical Fisher, and various (E)KFAC flavors. We also provide the damped inverses of (E)KFAC, with support for the empirical damping heuristic of Martens & Grosse (2015), and the exact damping scheme used by Grosse et al. (2023). curvlinops also supports backpropagation-free input-based curvature from Benzing (2022); Petersen et al. (2023), and the KFAC-reduce approximation from Eschenhagen et al. (2023) for neural networks with linear weight sharing layers.

2.2. Convenience Functionality & Safeguards

So far, we have seen the complexity introduced by automatic differentiation, probabilistic interpretation of loss functions and MC sampling, as well as the details of computing KFAC variations. Here, we present additional conveniences that can be incorporated into a linear operator interface.

Automatically handling batch aggregation. For computations, it is common to batch empirical risks (Eq. (1)) by splitting the data into N disjoint batches, $\mathbb{D} = \mathbb{B}_1 \cup \mathbb{B}_2 \cup \ldots \cup \mathbb{B}_N$, and evaluate the empirical risk on $\mathbb{D}$ by accumulation over the batch losses $\mathcal{L}_{\mathbb{B}_i}(\theta)$,

$$\mathcal{L}_{\mathbb{D}}(\theta) = \sum_{i=1}^{N} R_{\mathbb{D}}/R_{\mathbb{B}_i} \mathcal{L}_{\mathbb{B}_i}(\theta) , \quad (7)$$

with $\mathcal{L}_{\mathbb{B}_i}(\theta)$ defined via Eq. (1) for the batch $\mathbb{B}_i$. The aggregation over batches requires correcting the batch scaling and accounting for the global scaling.

Curvature computations can be split into batches in exactly the same way. Automatically accounting for correct aggregation over batches is useful, because it allows to change the batch size or data split without changing the matrix-vector product.

Automatically handling parameter formats. We often regard the neural net parameters $\boldsymbol{\theta} \in \mathbb{R}^D$ as a vector. However, in software, they are separated into tensors in different layers and roles, such as weights and biases. E.g., for a multi-layer perceptron $f_{\boldsymbol{\theta}}(\boldsymbol{x}) := \boldsymbol{x}^{(L)}$ with L layers and element-wise activation function σ, where $\boldsymbol{z}^{(l)} = \sigma\left(\boldsymbol{W}^{(l)}\boldsymbol{z}^{(l-1)} + \boldsymbol{b}^{(l)}\right)$ for $l = 1, \dots, L$ and $\boldsymbol{z}^{(0)} = \boldsymbol{x}$, the weights $\boldsymbol{W}^{(l)}$ are matrices and the biases $\boldsymbol{b}^{(l)}$ are vectors. The total parameter space is a tensor product of tensor lists $(\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \dots, \boldsymbol{W}^{(L)}, \boldsymbol{b}^{(L)}) \simeq \boldsymbol{\theta}$. To obtain the vector representation, one simply flattens and concatenates the entries, $\boldsymbol{\theta} = \text{Cat}(\text{vec}\,\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \dots, \text{vec}\,\boldsymbol{W}^{(L)}, \boldsymbol{b}^{(L)})$.

A linear operator A for a curvature matrix can process both formats and return the result in the same format:

```
# 1) input and result as vector
v = torch.rand(A.shape[1])
Av = A @ v
# 2) input and result as tensor list
v = [torch.rand_like(p) for p in params]
Av = A @ v
```

Preventing common mistakes. Many deep learning techniques introduce stochasticity into the neural net, and therefore the empirical risk (1). Others induce non-deterministic, or data-order dependent behavior, in the batched empirical risk (7). This is undesirable: We want a linear operator $\mathcal{A}$ to *always* return the same $\mathcal{A}\boldsymbol{v}$ for a fixed $\boldsymbol{v}$, independent of the batching scheme or the random number generator state. A non-deterministic linear operator introduces numerical inaccuracies into our application, e.g. CG becomes unstable with a noisy matrix (Martens, 2010). Examples of techniques that can introduce non-deterministic behavior are data augmentation, dropout (Srivastava et al., 2014), and batch normalization (Ioffe & Szegedy, 2015). Although such pathologies are hard to detect in general, curvlinops provides a sanity check that identifies most failure scenarios and significantly reduces the risk of introducing such spurious artifacts: When creating a linear operator, it (optionally) checks whether two consecutive evaluations of the empirical risk, its gradient, and the operator's matrix-vector product are identical.

Ensuring correctness. Testing some curvature approximations, specifically, KFAC, is challenging because they only become exact in special settings. Without testing, this can easily introduce scaling bugs where a Kronecker factor is off by a scalar. Our KFAC implementation tests KFAC's equivalence to the block-diagonal GGN, empirical Fisher,

or type-I Fisher using the scenarios described in (Bernacchia et al., 2018; Eschenhagen et al., 2023). Although these scenarios are not comprehensive, this is an active field of research, and further tests can be added to curvlinops as new equivalences and approximations are discovered.

3. Linear Operators Simplify Applications of Curvature Matrices

Here, we argue that linear operators simplify using curvature matrices in a broad range of applications. The workflow simplifies into three steps.

Step 1: Create a linear operator. Recall from §2 that an empirical risk, or one of its associated curvature matrices, requires four ingredients: a neural net architecture, the loss function, a data set, and the network parameters. A unified interface for all curvature matrices addressed in §2 is:

```
A = LinearOperator(
    model, # Neural net, fθ : {xn}B ↦ {fn}B
    loss_func, # Criterion ℓ, ({fn}B, {yn}B) ↦ LB(θ)
    params, # θ as list, (W(1), b(1), ..., W(L), b(L))
    data, # Data loader, yields a batch ({xn}B, {yn}B)
    check_deterministic=True, # Safeguards § 2.2
    **kwargs, # Optional configuration arguments
)
```

Step 2: Compose or transform operators. Linear operators can lazily be composed, e.g. summed, or transformed to represent or approximate other matrix functions of $\mathcal{A}$ (e.g. via Lanczos/Arnoldi). One specific example is inverting a linear operator, i.e. creating a linear operator representing $\mathcal{A}^{-1}$ given $\mathcal{A}$. Note that the product $\boldsymbol{v} \mapsto \mathcal{A}^{-1}\boldsymbol{v} := \boldsymbol{z}$ requires solving $\mathcal{A}\boldsymbol{z} = \boldsymbol{v}$ for $\boldsymbol{z}$. This can be done with an iterative solver based on matrix-vector products with $\mathcal{A}$, like truncated CG: $\mathcal{A}^{-1}\boldsymbol{v} = \text{cgsolve}(\mathcal{A}, \boldsymbol{v})$. Alternatives are using a truncated Neumann series $\mathcal{A}^{-1} = \sum_{k=0}^{\infty}\left(\boldsymbol{I} - \mathcal{A}\right)^k$ or, in the case of KFAC, leveraging properties of the Kronecker product structure. Since transformed or composed operators are again linear operators, they can be used in exactly the same way as the operators we started out with.

Step 3: Use operators like dense matrices. We will discuss this next along with a series of relevant downstream applications to showcase the transferability of linear operators and provide associated code snippets that demonstrate the similarity between code and mathematical notation.

3.1. Examples

Application 1: Second-order optimization. Newton's method and natural gradient descent (NGD) precondition a gradient vector $\boldsymbol{g}(\boldsymbol{\theta})$ with the inverse of a curvature matrix $\boldsymbol{C}(\boldsymbol{\theta})$: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\boldsymbol{C}(\boldsymbol{\theta})^{-1}\boldsymbol{g}(\boldsymbol{\theta})$. Due to the size of $\boldsymbol{C}(\boldsymbol{\theta})$, inversion can only be done implicitly, or through structural approximations that are cheap to store or invert.

Here are three different approaches for computing approximate Newton or natural gradient steps:

```
g = gradient()
# Newton−CG (Martens, 2010)
G = GGNLinearOperator(...)
G_inv = CGInverseLinearOperator(G)
step = -G_inv @ g
# Neumann optimizer (Krishnan et al., 2017)
H = HessianLinearOperator(...)
H_inv = NeumannInverseLinearOperator(H)
step = -H_inv @ g
# KFAC optimizer (Martens & Grosse, 2015)
K = KFACLinearOperator(...)
K_inv = KFACInverseLinearOperator(K)
step = -K_inv @ g
```

Application 2: Influence functions. Inverse Hessian-vector products (IHVPs), or more generally inverse curvature-vector products also occur during differentiation through an empirical risk minimization procedure. One important application is influence functions (Hampel, 1974; Koh & Liang, 2017; Grosse et al., 2023; Mlodozeniec et al., 2025), which study a perturbation's impact on the minimizer

$$\hat{\boldsymbol{\theta}}(\boldsymbol{\theta}, \epsilon) = \arg\min_{\boldsymbol{\theta}} \left(\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta}) + \epsilon P(\boldsymbol{\theta}) \right)$$

where $P(\boldsymbol{\theta})$ is a perturbation (e.g. up/down-weighting, or perturbing, a data point). Then, the implicit function theorem provides the differentiation rule for $\hat{\boldsymbol{\theta}}$ without having to differentiate through the unrolled optimization procedure. The perturbation's influence on the parameters is $\mathcal{I} = \mathrm{d}\hat{\boldsymbol{\theta}}/\mathrm{d}\epsilon\big|_{\epsilon=0} = \boldsymbol{H}(\hat{\boldsymbol{\theta}})^{-1}\nabla_{\boldsymbol{\theta}}P(\hat{\boldsymbol{\theta}})$. E.g., the influence for up-weighting data point n ($P(\boldsymbol{\theta}) = R_{\mathbb{D}}\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n)$) is

```
g_n = gradient(...)
# influence using EKFAC (Grosse et al., 2023)
EK = EKFACLinearOperator(...)
EK_inv = EKFACInverseLinearOperator(EK)
I = -EK_inv @ g_n
```

Similar hyper-gradients also occur in algorithms for bi-level (Lu et al., 2022; Evtushenko, 1974) and hyper-parameter optimization (Lorraine et al., 2020).

Application 3: Model merging. Assume we are given a neural network architecture $f_{\boldsymbol{\theta}}$ that is trained on T independent tasks $\mathbb{D}_1, \ldots, \mathbb{D}_T$ using loss functions $\ell_1, \ldots, \ell_T$. This yields T parameter vectors $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_T$ which we are supposed to merge into a single vector $\boldsymbol{\theta}_\star$. One naïve way is to simply average the parameters, $\boldsymbol{\theta}_\star = 1/T \sum_{t=1}^{T} \boldsymbol{\theta}_t$. Fisher-weighted model merging (Matena & Raffel, 2022) is a superior, albeit more expensive, merging procedure where each parameter $\boldsymbol{\theta}_t$ is weighted by the Fisher matrix $\boldsymbol{F}_t(\boldsymbol{\theta}_t)$ evaluated on the task $\mathbb{D}_t$, then normalized:

$$\boldsymbol{\theta}_\star = \left(\sum_{t=1}^{T} \boldsymbol{F}_t(\boldsymbol{\theta}_t) \right)^{-1} \left(\sum_{t=1}^{T} \boldsymbol{F}_t(\boldsymbol{\theta}_t)\boldsymbol{\theta}_t \right).$$

For computational reasons, the original work resorts to a diagonal approximation of the Fisher. Here is how to imple-ment the original approach, and an alternative unexplored variant that uses the full instead of the diagonal Fisher:

```
params, ls, Ds = [...], [...], [...]
# Per−task Fisher matrices
Fs = [GGNLinearOperator(..., l_t, p_t, D_t)
        for p_t, l_t, D_t
        in zip(params, ls, Ds)]
# Diagonal Fisher merging (Matena & Raffel, 2022)
diag_Fs = [hutchinson_diag(F_t)
            for F_t in Fs]
sum_inv = sum(diag_Fs) ** (-1)
merge = sum_inv * sum(F_t * p_t
    for F_t, p_t in zip(diag_Fs, params))
# Full Fisher merging via CG (unexplored)
sum_inv = CGInverseLinearOperator(sum(Fs))
merge = sum_F_inv @ sum(F_t @ p_t
    for F_t, p_t in zip(Fs, params))
```

Application 4: Model pruning/sparsification. With the advent of larger parameter spaces, interest in pruning arose. One idea is to eliminate parameters that least impact the loss (LeCun et al., 1989; Hassibi & Stork, 1992). Given a neural net trained to convergence at $\boldsymbol{\theta}$, the optimal perturbation $\tilde{\boldsymbol{\theta}} = \boldsymbol{\theta} + \boldsymbol{\delta}(i)$, such that $[\tilde{\boldsymbol{\theta}}]_i = 0$ with minimal increase (ρ) of the loss is

$$\boldsymbol{\delta}(i) = \frac{-[\boldsymbol{\theta}]_i \boldsymbol{H}(\boldsymbol{\theta})^{-1}\boldsymbol{e}_i}{[\boldsymbol{H}(\boldsymbol{\theta})^{-1}]_{i,i}}, \quad \rho(\boldsymbol{\delta}(i)) = \frac{[\boldsymbol{\theta}]_i^2}{2[\boldsymbol{H}(\boldsymbol{\theta})^{-1}]_{i,i}}.$$

This procedure is costly, requiring IHVPs and the inverse Hessian's diagonal. Recent work resorted to KFAC to scale similar ideas to large language models (van der Ouderaa et al., 2024). Another popular approximation, showcased below, is to assume diagonality, i.e. $[\boldsymbol{H}^{-1}(\boldsymbol{\theta})]_{i,i} = 1/\boldsymbol{H}(\boldsymbol{\theta})_{i,i}$:

```
# OBS pruning with approximate Hessian diagonal
H = HessianLinearOperator(...)
diag_H_inv = 1 / hutchinson_diag(H)
rho = params**2 / (2 * diag_H_inv)
# prune 5% of parameters
set_zero = int(0.05 * params.numel())
_, idx = rho.topk(set_zero, largest=False)
params[idx] = 0.0
```

Application 5: Loss landscape analysis. Many works have studied the spectrum of curvature matrices (Ghojogh et al., 2019; Yao et al., 2020; Sagun et al., 2017; 2018; Papyan, 2019a). As an example, we consider the phenomenon observed by Gur-Ari et al. (2018) that the gradient resides in the Hessian's top eigenspace. This is measured by overlap, which projects the gradient $\boldsymbol{g}(\boldsymbol{\theta})$ onto the space spanned by the top eigenvectors, yielding $\boldsymbol{g}(\boldsymbol{\theta})_{\text{proj}}$, then forming $\|\boldsymbol{g}(\boldsymbol{\theta})_{\text{proj}}\|^2/\|\boldsymbol{g}(\boldsymbol{\theta})\|^2 \in [0; 1]$. This can be done as follows:

```
g = gradient().numpy()
H = HessianLinearOperator(...).to_scipy()
# Gradient overlap with top eigenspace (Gur-Ari et al., 2018)
_, evecs = eigsh(H, k=10)
g_top = sum(dot(e, g) * e for e in evecs.T)
overlap = norm(g_top) ** 2 / norm(g) ** 2
```

In the above snippet, we have already used some interoperability with SciPy to leverage its highly sophisticated sparse eigensolvers. This capability, which we elaborate on in the next section, completes our argument in favor of using linear operators to compute with curvature matrices.

Remark. The list above is non-exhaustive and any applications that use the Hessian or approximations thereof can enjoy the benefits of the argued linear operator abstraction. One prominent example is uncertainty quantification with Bayesian neural nets. E.g., Yang et al. (2024); Kristiadi et al. (2024) show that the Kronecker-factored GGN is useful for turning standard large language models into Bayesian ones, enabling downstream applications in Bayesian optimization.

4. Linear Operators Enable Extensibility & Interoperability

For the last part of our argument, we emphasize the potential of linear operators to foster deep learning research by facilitating the connection between curvature matrices and the field of randomized linear algebra (RLA), as well as by enabling the application of sophisticated linear operator routines from libraries like SciPy (Virtanen et al., 2020).

Bridge to randomized linear algebra. RLA is a mature field, leveraging sharp inequality bounds that lead to algorithms with remarkable stability and scalability, spanning numerous applications like dimensionality reduction, combinatorial optimization, compressed sensing, and more (Tropp, 2015; Brailovskaya & van Handel, 2024). Despite its relevance, the adoption of RLA into ML research is behind other computational fields. This issue was recently highlighted in a 2023 NeurIPS tutorial, pointing at challenges involving hardware, software and theory-practice gaps (Dereziński & Mahoney, 2024). We think linear operators help overcome many of these practical issues due to their simple and universal interface that is *compatible* with such algorithms: ML practitioners wishing to incorporate an existing RLA procedure just have to worry about satisfying this interface from both sides, with minimal overhead. To demonstrate this last point, we implemented some methods from the literature that are compatible with any linear operator in `curvlinops` (see Fig. 4 for some toy experiments):

- **Spectral densities (Fig. 4, top).** Let λ_i denote the ith eigenvalue of $\mathcal{A}$. The spectral density of $\mathcal{A}$ is $\rho(\lambda) = {}^1/\mathrm{dim}(\mathcal{A}) \sum_i \delta(\lambda - \lambda_i)$. We implement two algorithms from Papyan et al. (2020) that are based on Lanczos iterations. One estimates $\rho(\lambda)$, the other estimates the matrix logarithm's spectral density $\rho_{\log}(\nu) = {}^1/\mathrm{dim}(\mathcal{A}) \sum_i \delta(\nu - \log(|\lambda_i| + \epsilon))$ with $\epsilon > 0$, which is the spectral density of $\log(|\mathcal{A}| + \epsilon\boldsymbol{I})$ where $|\mathcal{A}|$ is the absolute value matrix function.

- **Trace, squared Frobenius norm (Fig. 4, bottom left).** The simplest method to estimate the trace $\mathrm{Tr}(\mathcal{A})$ is the Girard-Hutchinson method (Girard, 1989; Hutchinson, 1989), which draws i.i.d. vectors from a distribution with zero mean and unit covariance, and uses that $\mathrm{Tr}(\mathcal{A}) = \mathrm{Tr}(\mathcal{A}\mathbb{E}[\boldsymbol{vv}^\top]) = \mathbb{E}[\boldsymbol{v}^\top \mathcal{A}\boldsymbol{v}]$. We also implement its variance-reduced version Hutch++ (Meyer et al., 2020), and XTrace (Epperly et al., 2024), which combines variance reduction with the exchangeability principle to achieve invariant estimates when permuting test vectors. All are unbiased, and variance reduction yields more accurate trace estimators for matrices with fast spectral decay whose trace is dominated by a few eigenvalues, at the cost of additional memory.

  Trace estimation allows estimating the squared Frobenius norm $\|\mathcal{A}\|_{\mathrm{F}}^2 = \mathrm{Tr}(\mathcal{A}\mathcal{A}^\top)$ as well. We also implement a variant that uses half the matrix-vector products.

- **Diagonal (Fig. 4, bottom right).** Bekas et al. (2007) extended the trace estimation idea from Hutchinson's method to matrix diagonals $\mathrm{diag}(\mathcal{A})$. We implement their algorithm and XDiag from Epperly et al. (2024), which generalizes XTrace to diagonals. Similar to the trend for trace estimation, we observe XDiag to be more accurate on matrices with fast spectral decay.

We have included some of these methods in the code snippets in §3, illustrating how linear operators provide a bridge to RLA, making RLA accessible to a broader audience.

SciPy/NumPy compatibility. Many other RLA routines are already implemented in other established packages, like SciPy/NumPy (Harris et al., 2020; Virtanen et al., 2020). To avoid their re-implementation, `curvlinops`'s linear operators can easily be exported to SciPy linear operators that consume NumPy arrays:

```
# 3) SciPy/NumPy support
A = A.to_scipy()
v = numpy.random.rand(A.shape[1])
Av = A @ v
```

This allows executing the expensive curvature matrix-vector multiplies on a GPU, while unlocking SciPy's highly sophisticated routines for linear operators, many of which currently lack a compatible interface in PyTorch. One example is `scipy.sparse.linalg`'s highly efficient iterative solvers for partial eigen- and singular value decompositions like `eigsh` and `svds` based on implicitly restarted Arnoldi iterations (Lehoucq et al., 1998). These sophisticated solvers usually require much fewer matrix-vector products compared to naïve methods, like the power iteration, and are not (yet) implemented in PyTorch or JAX. Other examples are linear system solvers such as conjugate gradients (CG, Hestenes & Stiefel, 1952) or LSMR (Fong & Saunders, 2010).

Bridge to under-explored sketching algorithms for DL.
In addition to the previously mentioned methods, there is a growing number of cutting-edge RLA algorithms that find their way into deep learning thanks to linear operators. For example, Singh et al. (2021); Singhal et al. (2023) show that curvature matrices like the Hessian can be highly rank-deficient, making low-rank decompositions valuable to approximate curvature. Still, computing k such eigenvectors using an iterative solver requires $\mathcal{O}(\tau k)$ HVPs, across τ *sequential* iterations (Golub & Van Loan, 2013, 7.3) becomes intractable for larger nets; numerical stability can be an issue, too. In contrast, sketch-and-solve methods require only $\mathcal{O}(k)$ *parallelizable* HVPs, and are numerically stable, allowing to scale low-rank approximations (Halko et al., 2011; Tropp et al., 2019). The skerch library (Fernandez, 2024) implements such sketched matrix decompositions, assuming a minimal linear operator interface in PyTorch. This allows to plug-in curvlinops with minimal overhead:

```
H = HessianLinearOperator(...)
# Low−rank sketch: H ≈ QUSU^⊤Q^⊤
Q, U, S = skerch.seigh(H, ...)
```

Bridge to optimizers & compositional linear algebra.
Many (inverse) curvature approximations are developed in the context of second-order optimization and are estimated during training. Linear operators can potentially export these approximations at any point of training, to leverage them for any of the applications we mention in §3. Realizing this idea requires the optimizer to implement the (inverse) curvature matrix as a linear operator. We demonstrate this approach and implement linear operators for the structured and inverse-free KFAC (Lin et al., 2024c) and inverse-free Shampoo (Lin et al., 2024b) optimizers. These optimizers offer linear operators of block-diagonal Kronecker-factored estimators of the empirical Fisher and gradient outer product inverses. Other algorithms whose curvature approximation could be made available via linear operators in the future are improved versions of the empirical Fisher for fine-tuning (Wu et al., 2024), Ginger (Hao et al., 2024), M-FAC (Frantar et al., 2021), or Spectral (Lin et al., 2024a). Finally, to leverage the latest structure-aware linear operator implementations via dispatch algorithms, one could consider supporting interoperability with the CoLA package (Potapczynski et al., 2023).

5. Alternative Views

Some benefits of presenting curvature matrices as linear operators can also be seen as potential pitfalls.

Encapsulation may lead to duplicated computation. Encapsulating all computations related to the curvature matrix in a linear operator (§2) means in practice that it might be necessary to duplicate computations. For example, in the case of a gradient descent algorithm that preconditions the gradient with a KFAC approximation of the empirical Fisher, we theoretically only require a single forward-backward pass to compute the gradient and the preconditioner. However, if the preconditioner computation is isolated in a linear operator, it might be necessary to compute an additional redundant forward-backward pass, which doubles the cost.

Hiding complexity may promote incorrect usage.
While hiding the complexity of curvature matrices makes it easier to apply them to new applications (§3), this might also promote their incorrect usage. How exactly a curvature matrix should be defined for a specific application is non-trivial, e.g., the GGN is determined by choosing a specific *split* of the objective (Martens, 2020; Kunstner et al., 2019). When exposing a curvature matrix as a linear operator to users, the implementation will have to be on a spectrum between two extremes to handle this nuance: (1) the interface can include configuration options to cover as many variations of the curvature matrix as possible, and (2) only a single variation of the curvature matrix is supported, based on what is deemed the most broadly applicable set of assumptions. Case (1) requires expert knowledge of the user, although admittedly the implementation burden is still lifted from them. In case (2) the curvature matrix required by the application might not even be available. In both cases, the user might unknowingly use a curvature linear operator that is inappropriate for their application.

Linear operator interface may be too minimal. Finally, while many applications of curvature matrices only require matrix-vector products, others might benefit from the ability to directly access sub-matrices of a curvature matrix.

6. Conclusion

We have advocated for the use of linear operators as a powerful abstraction for handling structured matrices, particularly curvature matrices, in machine learning applications. Presenting these matrices as linear operators enables users to leverage their benefits across various applications without the burden of implementation complexity. This approach not only encapsulates the intricacies of curvature computations but also enhances extensibility and interoperability, as demonstrated through our curvlinops library. Our work underscores the potential of linear operators to democratize access to advanced structural approximations, facilitating their integration into large-scale neural network architectures and transferring techniques to other machine learning tasks. We believe that embracing curvature matrices represented as linear operators will be crucial for advancing both research and practical applications in deep learning.

Acknowledgements

References

Amari, S.-I. Natural gradient works efficiently in learning. *Neural Computation*, 2000.

Bekas, C., Kokiopoulou, E., and Saad, Y. An estimator for the diagonal of a matrix. *Applied Numerical Mathematics*, 2007.

Benzing, F. Gradient descent on neurons and its link to approximate second-order optimization. In *International Conference on Machine Learning (ICML)*, 2022.

Bernacchia, A., Lengyel, M., and Hennequin, G. Exact natural gradient in deep linear networks and its application to the nonlinear case. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Biamonte, J. and Bergholm, V. Tensor networks in a nutshell, 2017.

Botev, A., Ritter, H., and Barber, D. Practical Gauss-Newton optimisation for deep learning. In *International Conference on Machine Learning (ICML)*, 2017.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., and Wanderman-Milne, S. JAX: composable transformations of Python+NumPy programs, 2018. URL https://github.com/jax-ml/jax.

Brailovskaya, T. and van Handel, R. Universality and sharp matrix concentration inequalities. *Geometric and Functional Analysis*, 34(6):1734–1838, December 2024. ISSN 1016-443X.

Bridgeman, J. C. and Chubb, C. T. Hand-waving and interpretive dance: an introductory course on tensor networks. *Journal of Physics A: Mathematical and theoretical*, 2017.

Charlier, B., Feydy, J., Glaunes, J. A., Collin, F.-D., and Durif, G. Kernel operations on the gpu, with autodiff, without memory overflows. *Journal of Machine Learning Research (JMLR)*, 2021.

Chen, Y., Chen, Y., Chen, J., Wen, Z., and Huang, J. Efficient second-order optimization for neural networks with kernel machines. In *ACM International Conference on Information & Knowledge Management*, 2022.

Dagréou, M., Ablin, P., Vaiter, S., and Moreau, T. How to compute hessian-vector products? In *International Conference on Learning Representations (ICLR) Blogposts*, 2024.

Dangel, F., Kunstner, F., and Hennig, P. BackPACK: Packing more into backprop. In *International Conference on Learning Representations (ICLR)*, 2020.

Dangel, F., Tatzel, L., and Hennig, P. ViViT: Curvature access through the generalized gauss-newton's low-rank structure. *Transactions on Machine Learning Research (TMLR)*, 2022.

Dao, T., Gu, A., Eichhorn, M., Rudra, A., and Ré, C. Learning fast algorithms for linear transforms using butterfly factorizations. In *International Conference on Machine Learning (ICML)*, 2019.

Dao, T., Chen, B., Sohoni, N. S., Desai, A., Poli, M., Grogan, J., Liu, A., Rao, A., Rudra, A., and Ré, C. Monarch: Expressive structured matrices for efficient and accurate training. In *International Conference on Machine Learning (ICML)*, 2022.

Daxberger, E., Kristiadi, A., Immer, A., Eschenhagen, R., Bauer, M., and Hennig, P. Laplace redux - effortless bayesian deep learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Dereziński, M. and Mahoney, M. W. Recent and upcoming developments in randomized numerical linear algebra for machine learning. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024.

Epperly, E. N., Tropp, J. A., and Webber, R. J. Xtrace: Making the most of every sample in stochastic trace estimation. *SIAM Journal on Matrix Analysis and Applications (SIMAX)*, 2024.

Eschenhagen, R., Immer, A., Turner, R. E., Schneider, F., and Hennig, P. Kronecker-factored approximate curvature for modern neural network architectures. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Evtushenko, Y. G. Iterative methods for solving minimax problems. *USSR Computational Mathematics and Mathematical Physics*, 1974.

Fernandez, A. *Skerch: Sketched matrix decompositions for PyTorch*, 2024. URL https://github.com/andres-fr/skerch.

Fix, E. and Hodges, J. *Discriminatory Analysis: Nonparametric Discrimination: Consistency Properties*. USAF School of Aviation Medicine, 1951.

Fong, D. and Saunders, M. LSMR: An iterative algorithm for sparse least-squares problems: Systems optimization laboratory. *SIAM J. Sci. Comput.*, 2010.

Frantar, E., Kurtic, E., and Alistarh, D. M-fac: Efficient matrix-free approximations of second-order information. *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Gardner, J., Pleiss, G., Weinberger, K. Q., Bindel, D., and Wilson, A. G. Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

George, T., Laurent, C., Bouthillier, X., Ballas, N., and Vincent, P. Fast approximate natural gradient descent in a kronecker-factored eigenbasis, 2018.

Ghojogh, B., Karray, F., and Crowley, M. Eigenvalue and generalized eigenvalue problems: Tutorial, 2019.

Girard, D. A. A fast 'monte-carlo cross-validation' procedure for large least squares problems with noisy data. *Numerische Mathematik*, 1989.

Golmant, N., Yao, Z., Gholami, A., Mahoney, M., and Gonzalez, J. pytorch-hessian-eigenthings: efficient pytorch hessian eigendecomposition, 2018.

Golub, G. H. and Van Loan, C. F. *Matrix Computations*. Johns Hopkins University Press, 4 edition, 2013. doi: 10.1137/1.9781421407944.

Granziol, D., Wan, X., and Garipov, T. Deep curvature suite, 2019.

Grosse, R. and Martens, J. A kronecker-factored approximate Fisher matrix for convolution layers. In *International Conference on Machine Learning (ICML)*, 2016.

Grosse, R., Bae, J., Anil, C., Elhage, N., Tamkin, A., Tajdini, A., Steiner, B., Li, D., Durmus, E., Perez, E., Hubinger, E., Lukošiūtė, K., Nguyen, K., Joseph, N., McCandlish, S., Kaplan, J., and Bowman, S. R. Studying large language model generalization with influence functions, 2023.

Gur-Ari, G., Roberts, D. A., and Dyer, E. Gradient descent happens in a tiny subspace, 2018.

Halko, N., Martinsson, P. G., and Tropp, J. A. Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. *SIREV*, 2011.

Hampel, F. R. The influence curve and its role in robust estimation. *Journal of the American Statistical Association*, 1974.

Hao, Y., Cao, Y., and Mou, L. Ginger: An efficient curvature approximation with linear complexity for general neural networks. arXiv 2402.03295, 2024.

Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., et al. Array programming with numpy. *Nature*, 2020.

Hassibi, B. and Stork, D. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems (NIPS)*, 1992.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *IEEE conference on computer vision and pattern recognition (CVPR)*, 2016.

Heskes, T. On "natural" learning and pruning in multilayered perceptrons. *Neural Computation*, 12(4), 2000.

Hestenes, M. R. and Stiefel, E. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49:409–435, 1952.

Hutchinson, M. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communication in Statistics—Simulation and Computation*, 1989.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, 2015.

Jacot, A., Gabriel, F., and Hongler, C. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2018.

Karpathy, A. The simplest, fastest repository for training/finetuning medium-sized gpts., 2022. URL https://github.com/karpathy/nanoGPT.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

Koh, P. W. and Liang, P. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning (ICML)*, 2017.

Krishnan, S., Xiao, Y., and Saurous, R. A. Neumann optimizer: A practical optimization algorithm for deep neural networks, 2017.

Kristiadi, A., Strieth-Kalthoff, F., Skreta, M., Poupart, P., Aspuru-Guzik, A., and Pleiss, G. A sober look at LLMs for material discovery: Are they actually good for Bayesian optimization over molecules? In *International Conference on Machine Learning (ICML)*, 2024.

Kunstner, F., Hennig, P., and Balles, L. Limitations of the empirical fisher approximation for natural gradient descent. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

LeCun, Y., Denker, J., and Solla, S. Optimal brain damage. In *Advances in Neural Information Processing Systems (NIPS)*, 1989.

Lehoucq, R. B., Sorensen, D. C., and Yang, C. *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*. SIAM, 1998.

Lin, W., Dangel, F., Eschenhagen, R., Bae, J., Turner, R. E., and Grosse, R. B. Fast fractional natural gradient descent using learnable spectral factorizations, 2024a.

Lin, W., Dangel, F., Eschenhagen, R., Bae, J., Turner, R. E., and Makhzani, A. Can we remove the square-root in adaptive gradient methods? a second-order perspective. In *International Conference on Machine Learning (ICML)*, 2024b.

Lin, W., Dangel, F., Eschenhagen, R., Neklyudov, K., Kristiadi, A., Turner, R. E., and Makhzani, A. Structured inverse-free natural gradient descent: Memory-efficient & numerically-stable KFAC. In *International Conference on Machine Learning (ICML)*, 2024c.

Loan, C. F. The ubiquitous Kronecker product. *Journal of Computational and Applied Mathematics*, 2000.

Lorraine, J., Vicol, P., and Duvenaud, D. Optimizing millions of hyperparameters by implicit differentiation. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2020.

Lu, Y., Kamath, G., and Yu, Y. Indiscriminate data poisoning attacks on neural networks. *TMLR*, 2022.

MacKay, D. J. A practical Bayesian framework for backpropagation networks. *Neural Computation*, 4(3), 1992.

Martens, J. Deep learning via Hessian-free optimization. In *International Conference on Machine Learning (ICML)*, 2010.

Martens, J. New insights and perspectives on the natural gradient method, 2020.

Martens, J. and Grosse, R. Optimizing neural networks with Kronecker-factored approximate curvature. In *International Conference on Machine Learning (ICML)*, 2015.

Martens, J. and Sutskever, I. *Training Deep and Recurrent Networks with Hessian-Free Optimization*, pp. 479–535. Springer Berlin Heidelberg, 2012.

Martens, J., Ba, J., and Johnson, M. Kronecker-factored curvature approximations for recurrent neural networks. In *International Conference on Learning Representations (ICLR)*, 2018.

Matena, M. S. and Raffel, C. A. Merging models with fisher-weighted averaging. *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Meyer, R. A., Musco, C., Musco, C., and Woodruff, D. P. Hutch++: Optimal stochastic trace estimation, 2020.

Mlodozeniec, B., Eschenhagen, R., Bae, J., Immer, A., Krueger, D., and Turner, R. Influence functions for scalable data attribution in diffusion models. In *International Conference on Learning Representations (ICLR)*, 2025.

Osawa, K., Ishikawa, S., Yokota, R., Li, S., and Hoefler, T. Asdl: A unified interface for gradient preconditioning in pytorch, 2023.

Papyan, V. Measurements of three-level hierarchical structure in the outliers in the spectrum of deepnet Hessians. In *International Conference on Machine Learning (ICML)*, 2019a.

Papyan, V. The full spectrum of deepnet hessians at scale: Dynamics with sgd training and sample size, 2019b.

Papyan, V., Han, X. Y., and Donoho, D. L. Prevalence of neural collapse during the terminal phase of deep learning training. *Proceedings of the National Academy of Sciences (PNAS)*, 2020.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison,

M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Pearlmutter, B. A. Fast exact multiplication by the Hessian. *Neural Computation*, 1994.

Penrose, R. Applications of negative dimensional tensors. *Combinatorial Mathematics and its Applications*, 1971.

Petersen, F., Sutter, T., Borgelt, C., Huh, D., Kuehne, H., Sun, Y., and Deussen, O. ISAAC newton: Input-based approximate curvature for newton's method. In *International Conference on Learning Representations (ICLR)*, 2023.

Potapczynski, A., Finzi, M., Pleiss, G., and Wilson, A. G. CoLA: Exploiting Compositional Structure for Automatic and Efficient Numerical Linear Algebra. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Potapczynski, A., Qiu, S., Finzi, M. A., Ferri, C., Chen, Z., Goldblum, M., Bruss, C. B., Sa, C. D., and Wilson, A. G. Searching for efficient linear layers over a continuous space of structured matrices. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Rader, J., Lyons, T., and Kidger, P. Lineax: unified linear solves and linear least-squares in jax and equinox. *Advances in Neural Information Processing Systems (NeurIPS), Workshop NeuAI for Science*, 2023.

Ravasi, M. and Vasconcelos, I. Pylops–a linear-operator python library for large scale optimization. arXiv 1907.12349, 2019.

Ren, Y. and Goldfarb, D. Efficient subsampled gauss-newton and natural gradient methods for training neural networks, 2019.

Sagun, L., Bottou, L., and LeCun, Y. Eigenvalues of the hessian in deep learning: Singularity and beyond, 2017.

Sagun, L., Evci, U., Guney, V. U., Dauphin, Y., and Bottou, L. Empirical analysis of the hessian of over-parametrized neural networks, 2018.

Schneider, F., Dangel, F., and Hennig, P. Cockpit: A practical debugging tool for the training of deep neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Schraudolph, N. N. Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 2002.

Singh, S. P. and Alistarh, D. Woodfisher: Efficient second-order approximation for neural network compression. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Singh, S. P., Bachmann, G., and Hofmann, T. Analytic insights into structure and rank of neural network hessian maps. *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Singhal, U., Cheung, B., Chandra, K., Ragan-Kelley, J., Tenenbaum, J. B., Poggio, T. A., and Yu, S. X. How to guess a gradient. arXiv 2312.04709, 2023.

Soen, A. and Sun, K. Tradeoffs of diagonal fisher information matrix estimators. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JMLR)*, 2014.

Steinhaus, H. et al. Sur la division des corps matériels en parties. *Bulletin of the Polish Academy of Sciences*, 1956.

Tatzel, L., Mucsányi, B., Hackel, O., and Hennig, P. Debiasing mini-batch quadratics for applications in deep learning, 2024.

Tropp, J. A. An introduction to matrix concentration inequalities. *Found. Trends Mach. Learn.*, 8(1–2):1–230, May 2015. doi: 10.1561/2200000048.

Tropp, J. A., Yurtsever, A., Udell, M., and Cevher, V. Streaming low-rank matrix approximation with an application to scientific simulation. *SIAM Journal on Scientific Computing*, 2019.

van der Ouderaa, T. F. A., Nagel, M., van Baalen, M., Asano, Y. M., and Blankevoort, T. The LLM surgeon. In *International Conference on Learning Representations (ICLR)*, 2024.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors. SciPy

1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 2020.

Williams, C. K. and Rasmussen, C. E. *Gaussian processes for machine learning*. MIT press Cambridge, MA, 2006.

Wu, X., Yu, W., Zhang, C., and Woodland, P. An improved empirical fisher approximation for natural gradient descent. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Yang, A. X., Robeyns, M., Wang, X., and Aitchison, L. Bayesian low-rank adaptation for large language models. In *International Conference on Learning Representations (ICLR)*, 2024.

Yao, Z., Gholami, A., Keutzer, K., and Mahoney, M. W. Py-Hessian: Neural networks through the lens of the Hessian. In *IEEE International Conference on Big Data*, 2020.

A. Performance

Here, we evaluate the performance of `curvlinops`'s linear operators in terms of run time and memory consumption. This serves three purposes: (1) the results can be used as a rule of thumb by users to estimate the computational cost relative in terms of multiples of the gradient; (2) it demonstrates that `curvlinops` indeed scales to large models, and (3) it allows to highlight some PyTorch-specific limitations which, if addressed, enable future improvements.

Setup We consider two representative tasks for large-scale image recognition and language modelling: ResNet50 (≈ 26 M parameters, He et al., 2016) on ImageNet, and nanoGPT (≈ 124 M parameters Karpathy, 2022) on Shakespeare. We profile performance on an A40 GPU with 40 GiB of RAM; see Fig. 3 for the results & more experimental details. In both performance metrics, we observe the trend

$$\text{KFAC (inverse)} \quad \ll \quad \text{Fisher/GGN} \quad < \quad \text{Hessian}.$$

More details in the following paragraphs.

KFAC costs slightly more than a gradient. Computing KFAC's Kronecker factors costs 1.5-2.5 gradients in terms of time, and the overhead is larger for networks with convolutional layers (ResNet50) that require an additional input unfolding operations. For both setups, the additional matrix inversion required by the KFAC inverse adds only a barely noticeable time overhead and storing the Kronecker factors increases KFAC's memory consumption by no more than 10-20 % compared to the gradient. Many practical algorithms compute the Kronecker factors infrequently to further reduce costs. Once the Kronecker factors are computed, applying the curvature approximation to a vector is cheap: we observe run times between 0.1-0.2 gradients.

HVPs match related work. We find that the HVP performance reflects the findings of Dagréou et al. (2024) who performed an extensive evaluation of HVPs in multiple frameworks and different autodiff modalitites. For ViT and BERT architectures in PyTorch, they found that HVPs can take up to four to five gradients (we observe 4.5-5.5 gradients) in terms of time, and 3-3.5 times the memory (we observe 3-4.5). Despite these significantly higher memory demands, note that `curvlinops` would still allow to split the data into smaller batches, which can reduce the memory consumption back to that similar to a gradient (assuming the peak memory is dominated by stored activations).

FVPs & GGNVPs are cheaper than HVPs. Martens (2010) found that a matrix-vector product with the GGN (or the Fisher) "uses about half the memory and runs nearly twice as fast." They likely used an autodiff backend different from PyTorch. While we do not find that GGNVPs are twice

as efficient as HVPs, all GGN/Fisher multiplication routines are generally faster than HVPs and consume less memory.

There is potential for further improvements. The findings of Dagréou et al. (2024); Martens (2010) suggest that, while `curvlinops`'s matrix-vector products are of reasonable performance, they can further be accelerated in the future. Two specific shortcomings are the lack of forward mode AD (all operations are implemented with PyTorch's reverse mode, even JVPs, for maximum operator coverage) and compilation. As demonstrated by Dagréou et al. (2024), using these two mechanisms in JAX allows to compute HVPs in roughly 2-4 gradients in time and 2-3 gradients in memory. We plan to address the forward mode limitation in future releases by switching to the functional API (which according to Dagréou et al. (2024) currently does not support compilation). Beyond these limitations, we also found other caveats that harm performance. For nanoGPT, we had to disable PyTorch's efficient attention implementation for certain linear operators as it does not support double backpropagation. Clearly, this negatively affects performance.

B. More Curvature Matrices

Type-II Fisher. Integration by parts of the expectation in the type-I Fisher (4) leads to the type-II Fisher which highlights the Fisher's connection to the Hessian,

$$\boldsymbol{F}_{\mathrm{II}}(\boldsymbol{\theta}) := R_{\mathbb{D}} \ \textstyle\sum_{\boldsymbol{x}_n \in \mathbb{D}} \mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n^\top \mathbb{E}_{q(\boldsymbol{y}|\boldsymbol{f}_n)} \left[-\nabla^2_{\boldsymbol{f}_n} l_n \right] \mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n. \quad (8)$$

In other words, the Fisher can also be written as the negative log likelihood's expected Hessian *w.r.t. the model's distribution*, whereas the Hessian's expectation in (2) is *w.r.t. the empirical distribution* implied by the data. For many common loss functions, $\nabla^2_{\boldsymbol{f}} \log q(\boldsymbol{y} \mid \boldsymbol{f})$ does not depend on $\boldsymbol{y}$ and the expectation in (8) can be dropped; then, the Fisher equals the GGN. Even if this is not possible, we can apply the same Monte-Carlo techniques for estimating the type-II Fisher. Note however that each sample now requires $\dim(\boldsymbol{f})$ VJPs due to the loss-prediction Hessian's dimension (instead of one vector-Jacobian product in Eq. (5)).

C. Estimating Matrix Properties

To verify the correctness of our implemented matrix property estimation techniques, we perform sanity checks on toy problems from the original works. Fig. 4 visualizes our findings. See §4 for a more detailed discussion.
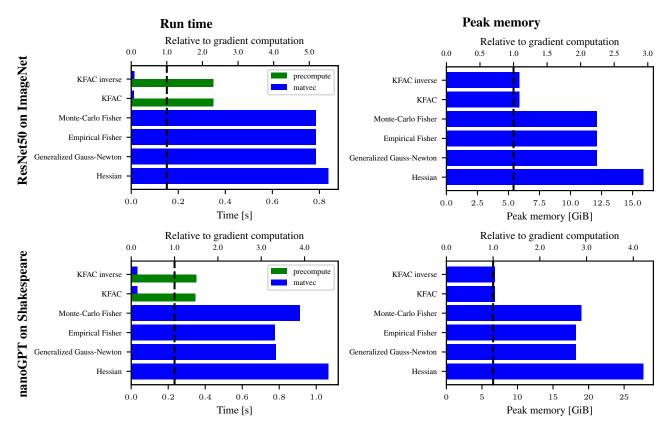
Figure 3. **Performance analysis:** Run time (left column) and peak memory (right column) of linear operators benchmarked on ResNet50 on ImageNet (top row) and nanoGPT on Shakespeare (bottom row) on an A40 GPU with 40 GiB of RAM (the code used to generate these results is here). **Details:** For ImageNet, we use a batch size of 64 and images of shape $(3, 224, 224)$; for Shakespeare, we use a batch size of 4 and context length 1024. All linear operators use their default options and we do not use compilation. Models are in evaluation mode. KFAC neglects parameters in normalization layers (they are unsupported), and nanoGPT's last layer due to its large dimension (≈ 50 K).
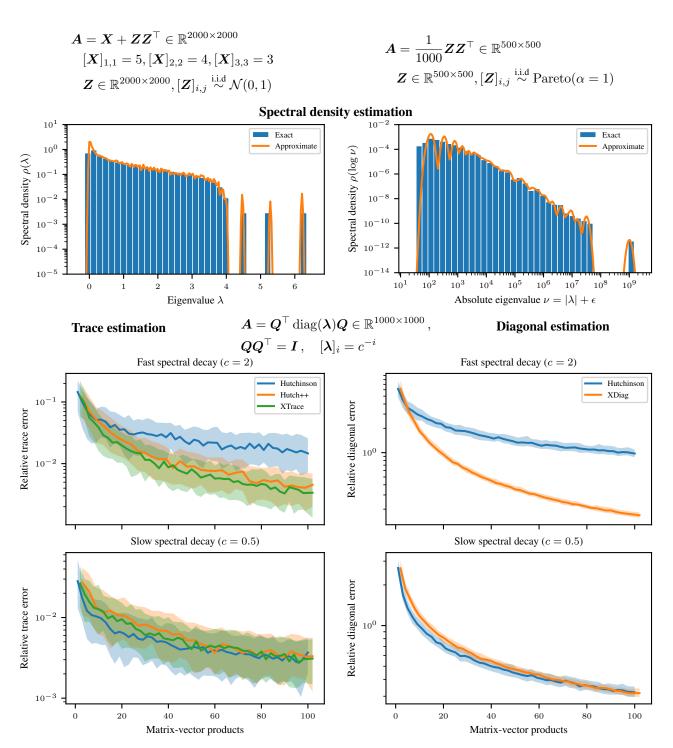
$$\boldsymbol{A} = \boldsymbol{X} + \boldsymbol{Z}\boldsymbol{Z}^\top \in \mathbb{R}^{2000 \times 2000}$$
$$[\boldsymbol{X}]_{1,1} = 5, [\boldsymbol{X}]_{2,2} = 4, [\boldsymbol{X}]_{3,3} = 3$$
$$\boldsymbol{Z} \in \mathbb{R}^{2000 \times 2000}, [\boldsymbol{Z}]_{i,j} \overset{\text{i.i.d}}{\sim} \mathcal{N}(0,1)$$

$$\boldsymbol{A} = \frac{1}{1000} \boldsymbol{Z}\boldsymbol{Z}^\top \in \mathbb{R}^{500 \times 500}$$
$$\boldsymbol{Z} \in \mathbb{R}^{500 \times 500}, [\boldsymbol{Z}]_{i,j} \overset{\text{i.i.d}}{\sim} \text{Pareto}(\alpha = 1)$$

Spectral density estimation

Trace estimation

$$\boldsymbol{A} = \boldsymbol{Q}^\top \operatorname{diag}(\boldsymbol{\lambda})\boldsymbol{Q} \in \mathbb{R}^{1000 \times 1000},$$
$$\boldsymbol{Q}\boldsymbol{Q}^\top = \boldsymbol{I}, \quad [\boldsymbol{\lambda}]_i = c^{-i}$$

Diagonal estimation

Figure 4. **Estimating linear operator properties with `curvlinops`.** We implement various estimation algorithms from the literature and evaluate them on toy problems. *Top:* Spectral density estimation with the algorithms and toy matrices from Papyan et al. (2020). The left panel estimates a spectral density, the right panel the spectral density of the matrix logarithm $\log(|\boldsymbol{A}| + \epsilon \boldsymbol{I})$ with $\epsilon = 10^{-5}$. Code to reproduce these figures is here. *Bottom:* Comparison of trace and diagonal estimators (Girard, 1989; Hutchinson, 1989; Epperly et al., 2024; Meyer et al., 2020) for matrices whose spectrum follows a power law ($\boldsymbol{Q}$ is obtained from the QR decomposition of a random Gaussian matrix). Solid lines are medians, error bars are 25- and 75-percentiles over 200 runs. For traces, we use the relative error $|\hat{t} - t|/|t|$ where $\hat{t}$ estimates the true trace t. For diagonals, we report the relative error $\max_i |a_i - \hat{a}_i|/\max_j |a_j|$ where $\hat{\boldsymbol{a}}$ approximates the true diagonal $\boldsymbol{a}$. On matrices with fast spectral decay, estimation techniques based on variance reduction improve over vanilla Hutchinson estimators.