OPTIMIZING OVER ALL SEQUENCES OF ORTHOGONAL POLYNOMIALS

Anonymous authors
Paper under double-blind review

ABSTRACT

Every length-$(n+1)$ sequence of orthogonal polynomials is uniquely represented by two length-$(n+1)$ sequences of coefficients α and β. We make this representation learnable by gradient-based methods. Orthogonal polynomial operations may be automatically differentiated, but this uses $O(n^2)$ memory and is very slow in practice. By exploiting reversibility, we derive a differentiation algorithm which uses $O(n)$ memory and is much faster in practice. Using this algorithm, fixed polynomial transforms (e.g. discrete cosine transforms) can be replaced by learnable layers. These are more expressive, but they retain the computational efficiency and analytic tractability of orthogonal polynomials.

As another application, we present an algorithm for approximating the minimal value $f(w^*)$ of a general nonconvex objective f, without finding the minimizer w^*. It follows a scheme recently proposed by Lasserre (2020), whose core algorithmic problem is to find the sequence of polynomials orthogonal to a given probability distribution. Despite the general intractability of this problem, we observe encouraging initial results on some test cases.

1 INTRODUCTION

Sequences $p_0, p_1, \ldots, p_n$ of orthogonal polynomials – such as the Chebyshev, Laguerre, and Hermite sequences – are fundamental in many areas of scientific computing. Their core operation is evaluation (or transformation): given a polynomial f, represented as coefficients $c_0, \ldots, c_n$ in the basis $p_0, \ldots, p_n$, compute $f(x_i)$ at $n+1$ distinct points $x_0, \ldots, x_n$[1]. The inverse operation is interpolation: given the points x_i and corresponding outputs y_i, compute the coefficients of the unique f satisfying $f(x_i) = y_i$. Computing the Jacobian determinant $|\partial f(x)/dc|$ is useful in applications such as normalizing flows (Rezende & Mohamed, 2015).

This work enables gradient-based optimization over the set of orthogonal polynomial sequences. These can be written recursively in terms of scalar sequences α and β, the latter positive:

$$p_{-1}(x) = 0; p_0(x) = 1; \quad p_{j+1}(x) = (x - \alpha_j)p_j(x) - \beta_j p_{j-1}(x) \quad \text{for } \alpha_j, \beta_j \text{ s.t. } \beta_j > 0 \tag{1}$$

The correspondence between the orthogonal polynomial sequences p and coefficients (α, β) is exact. (A more formal statement is given in Section 2.) We make (α, β) a learnable representation. We derive the gradients, with respect to (α, β), of the evaluation and interpolation operations. Compared to automatic differentiations, our algorithms use $O(n)$ (rather than $O(n^2)$) memory, and are much faster in practice. We explore two applications of this new optimization capability.

Learned polynomial transforms. Currently, orthogonal polynomials transforms are manually chosen based on intuitions about their suitability. Informally, Chebyshev polynomials resemble cosines; Laguerre polynomials are similar to cosines multiplied by exponentials; Hermite polynomials are roughly cosines divided by Gaussians (Valiant, 2016). These sequences are orthogonal with respect to different distributions μ over their inputs x; see Figure 1. Their corresponding transforms are generalized by our learnable AnyPT layer, which has fast algorithms for its forward, backward, inverse, and log-determinant passes. By unobtrusively replacing the DCT and IDCT within JPEG compression, we obtain better tradeoffs between visual quality and compression. It may be possible to gently improve many signal processing pipelines in this manner.

[1]Zero-indexing vectors of length $n+1$ is a standard convention for polynomial transforms.

Transform	x_i	α_k	$\beta = \gamma^2$	μ
Cosine-III	$\cos(\frac{\pi}{n+1})(i+\frac{1}{2})$	0	$\beta_0 = \pi, \beta_1 = \frac{1}{2}, \beta_k = \frac{1}{4}$	$2 \cdot \text{Beta}(\frac{1}{2}, \frac{1}{2}) - 1$
Legendre	(Bogaert, 2014)	0	$\beta_0 = 2, \beta_k = k^2/(4k^2-1)$	$\text{Unif}(-1, 1)$
Hermite	(Press et al., 1992)	0	$\beta_0 = \sqrt{\pi}, \beta_k = k/2$	$\text{Normal}(0, \frac{1}{2})$
Laguerre	(Press et al., 1992)	$2k+1$	$\beta_0 = 1, \beta_k = k^2$	Exp(1)

Figure 1: Parameters of classic orthonormal polynomial transforms, up to diagonal scaling. Cosine-III (short for the Discrete Cosine Transform, type III) is formed from the Chebyshev polynomials. The evaluation points x_i are taken to be the roots of p_{n+1}. They may not have a closed form, but can be calculted by the cited algorithms. Coefficient sequences may be obtained from Gautschi (2005); Leibon et al. (2008) and Chapter 4.5 of Press et al. (1992). μ is the probability distribution which renders the polynomial sequence μ-orthogonal.

Minimal values of optimization problems. We present an algorithm, called Mop, for approximating the minimal value $f(w^*)$ of a continuous function f, without finding its minimizer w^*. In recent work, Lasserre (2020) reduced this difficult problem to the following one: given access to a distribution μ over $\mathbb{R}$, find the sequence of orthogonal polynomials (represented as coefficients α and β) which are orthogonal with respect to μ. Our approach to solve the latter problem uses stochastic gradient descent.

Our contributions are: (1) memory-efficient algorithms, with fast CUDA implementations, for computing the vector-Jacobian products of evaluation and interpolation, (2) the learnable AnyPT layer, which demonstrates reduced error when applied to image compression, and (3) Mop, an iterative algorithm for estimating the minimal value f^* of a continuous function f.

2 Preliminaries

The following background material is found in references on orthogonal polynomials (Gautschi, 2004; 2005; Ismail et al., 2005) and numerical linear algebra (Higham, 2002).

Measures and moments. Let μ be a measure over $\mathbb{R}$. It is positive if $\mu(B) > 0$ for all nonempty open sets B. Its moments are $m_i = \int x^i d\mu(x)$ for $i \geq 0$. These define a linear functional $L(p)$, over polynomials p, via $L(x^k) = m_k$. Positive measures μ, moment sequences m whose Hankel matrices are positive definite, and positive linear functionals (satisfying $L(p) > 0$ for all nonnegative polynomials p) uniquely correspond to one another.

Orthogonal polynomials. A measure μ defines an inner product $\langle p_i, p_j \rangle = \int p_i(x) p_j(x) d\mu(x)$ over polynomials. p_i and p_j are μ-orthogonal if $\langle p_i, p_j \rangle = 0$. A sequence $[p_0, p_1, \ldots, p_n] = p$ of polynomials is μ-orthogonal if p_i and p_j are μ-orthogonal when $i \neq j$. A polynomial is monic if its coefficient on x^n is 1. A polynomial q is orthonormal if its norm $||q_i|| = \sqrt{\langle q_i, q_i \rangle} = 1$.

Three-term recurrence. For every positive μ, the sequence of μ-orthogonal monic polynomials satisfies the three-term recurrence (1) for some α and β. An orthonormal polynomial sequence $q_0, q_1, \ldots$ satisfies the following three-term recurrence:

$$q_{-1}(x) = 0; \quad q_0(x) = \gamma_0^{-1} \qquad \gamma_{j+1} q_{j+1}(x) = (x - \alpha_j) q_j(x) - \gamma_j q_{j-1}(x) \quad \text{for } \gamma_j > 0 \tag{2}$$

The orthonormal polynomial sequence q defined by (α, γ) corresponds to the monic polynomial sequence p defined by (α, β) where $\gamma_i = \sqrt{\beta_i}$ for $i > 0$. The Jacobi matrix, truncated to order n, organizes the coefficients (α, β) in the following $n \times n$ tridiagonal matrix:

$$J_n = \begin{bmatrix} \alpha_0 & \sqrt{\beta_1} & 0 & 0 & 0 \\ \sqrt{\beta_1} & \alpha_1 & \sqrt{\beta_2} & 0 & 0 \\ 0 & \sqrt{\beta_2} & \alpha_2 & \sqrt{\beta_3} & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & \sqrt{\beta_{n-1}} & \alpha_{n-1} \end{bmatrix} \tag{3}$$

Operation	Inputs	Output	$O(n^2)$-time Algorithm	$O(n \log^2 n)$-time Algorithm
Evaluation	x, α, β, c	Vc	(Smith, 1965)	(Potts, 2003)
Interpolation	x, α, β, y	$V^{-1}y$	Dual in Higham (1988)	
Derivative Evaluation	x, α, β, c	$V'c$	(Smith, 1965)	
Transpose Multiply	x, α, β, c	$V^T c$	Appendix	(Driscoll et al., 1997)
Transpose Solving	x, α, β, y	$V^{-T}y$	Primal in Higham (1988)	(Bostan et al., 2010)
Determinant	x, α, β	$\det(V)$	Formula in Sec. 2.1	(Gohberg & Olshevsky, 1994)
Eigenvalues	α, β	$\lambda_i(J_n)$		(Coakley & Rokhlin, 2013)

Figure 2: Algorithms for orthogonal polynomials. The $O(n^2)$ algorithms can be parallelized. In particular, a version of Clenshaw's algorithm for evaluation takes $O(n)$ parallel time (Barrio, 2000). Interpolate (the dual in Higham (1988)) takes $O(n)$ parallel time, since the inner loop over j can be run by $O(n)$ threads, each operating on three entries of c. $O(n \log^2 n)$-time algorithms for interpolation and derivative evaluation are not plainly written in the literature, to our knowledge.

Favard's theorem.[2] Given any sequence of monic polynomials p defined by α and β in the recurrence 1, define L via $L(p_0) = \beta_0$ and $L(p_i) = 0$ for $i > 0$. Then L is positive, and thereby corresponds to a positive measure μ for which p are μ-orthogonal.

Normalization. The norm of a degree-k monic polynomial is $||p_k||^2 = \prod_{j=0}^{k} \beta_j$. Neither the three-term recurrence (1) nor the Jacobi matrix involve β_0. This is because $\beta_0 = ||p_0||^2 = \int 1 d\mu(x)$ is the normalizing constant of μ. By fixing $\beta_0 = 1$, we restrict attention to probability distributions.

Vandermonde systems. Evaluation and interpolation can be viewed as matrix-vector multiplication and linear system solving, respectively, with a polynomial Vandermonde matrix V:

$$Vc = y \quad \text{where} \quad V_{i,j} = p_j(x_i) \qquad \text{(indexed from zero)} \tag{4}$$

V generalizes the Vandermonde matrix from monomials to orthogonal polynomials. Its determinant is the same as the Vandermonde matrix: $\det(V) = \prod_{1 \le i < j \le n}(x_j - x_i)$ (Barnett, 1975). V is invertible when the x_i are distinct. The derivative Vandermonde matrix has entries $V'_{i,j} = p'_j(x_i)$.

Numerical stability. The Evaluate algorithm in Figure 3 (i.e. Clenshaw's algorithm) is known to be numerically stable (Smoktunowicz, 2002). By contrast, Interpolate can be unstable for moderate n (Gohberg & Olshevsky, 1997; Higham, 1990). Fortunately, various mitigations have been developed for this issue. Furthermore, the instability in Interpolate can be resolved entirely using program transformation techniques. We do not encounter numerical instability in our present work, but discuss this issue further in the appendix.

3 Gradient-Based Optimization over Orthogonal Polynomials

Efficient reverse-mode differentiation (backpropagation) amounts to implementation of the vector-Jacobian product (VJP). Here is an example of our notation for the evaluation operation, i.e. computing $y = Vc$. Let $\nabla_x y = \frac{\partial y}{dx}$ be the Jacobian matrix of y with respect to x. The VJP operation is $(x, v) \mapsto v^T \nabla_x y$. In reverse mode, $\bar{x} = \bar{y}^T \nabla_x y$ is the application at $(x, \bar{y})$, where $\bar{y}$ was computed similarly. $\bar{\alpha}$, $\bar{\beta}$, and $\bar{c}$ are similarly defined for the other arguments α, β, and c.

VJPs may be computed automatically from a forward computation graph. However, this stores intermediate computations of the forward pass, which requires memory scaling with depth. Evaluate and Interpolate have depth $O(n)$, so $O(n^2)$ memory would be used. Furthermore, they involve sparse updates in nested loops, which are faster in plain CPU/GPU code.

We manually derive the VJPs in the appendix. By exploiting reversibility, we reduce their memory requirement to $O(n)$. The complete algorithms are displayed in Figure 4. As illustrated in Figure 5, they are substantially faster than automatic differentions, even when memory is not a concern.

[2]This is also by Shohat, Stone, etc., so it is also called the spectral theorem of orthogonal polynomials.

```
**procedure** Evaluate$(x, \alpha, \beta, c)$
    $u = [c_n, \ldots, c_n]$
    $v = [0, \ldots, 0]$
    **for** $k \in [n-1, \ldots, 0]$ **do**
        $\tau = u$
        $u = (x - \alpha_k) \cdot u$
              $-\beta_{k+1} v + c_k$
        $v = \tau$
    **return** $u$
```

Figure 3: Algorithms for polynomial evaluation and interpolation. All the inputs are vectors in $\mathbb{R}^{n+1}$. Both algorithms return a vector in $\mathbb{R}^{n+1}$. Evaluate is the Clenshaw algorithm. Interpolate is the dual algorithm of Higham (1988). DivDiffs(x, y) computes the first row of the table of divided differences of y with respect to x; see the appendix for its definition.

```
**procedure** Interpolate$(x, \alpha, \beta, y)$
    $\delta = \text{DivDiffs}(x, y)$
    **return** ChangeBasis$(x, \alpha, \beta, \delta)$
**procedure** DivDiffs$(x, y)$    (naive)
    $\delta = y$
    **for** $k \in [0, \ldots, n-1]$ **do**
        $\Delta = \delta_k$
        **for** $j \in [k+1, \ldots, n]$ **do**
            $\tau = \delta_j$
            $\delta_j = (\delta_j - \Delta)/(x_j - x_{j-k-1})$
            $\Delta = \tau$
    **return** $\delta$
**procedure** ChangeBasis$(x, \alpha, \beta, \delta)$
    $c = \delta$
    $c_{n-1} \overset{+}{=} (\alpha_0 - x_{n-1}) c_n$
    **for** $k \in [n-2, \ldots, 0]$ **do**
        $c_k \overset{+}{=} (\alpha_0 - x_k) c_{k+1} + \beta_1 c_{k+2}$
        **for** $j \in [1, \ldots, n-2-k]$ **do**
            $c_{k+j} \overset{+}{=} (\alpha_j - x_k) c_{k+j+1} + \beta_{j+1} c_{k+j+2}$
        $c_{n-1} \overset{+}{=} (\alpha_{n-k-1} - x_k) c_n$
    **return** $c$
```

```
**procedure** $\overline{\text{Evaluate}}(x, \alpha, \beta, u, v, \bar{u})$
    $\bar{x} = \bar{y} \circ (V' c)$
    $\bar{c} = V^T \bar{y}$
    $\bar{v} = 0$
    $\bar{\alpha} = \bar{\beta} = [0, \ldots, 0]$
    **for** $k \in [0, \ldots, n-1]$ **do**
        $w = \frac{1}{\beta_{k+1}}(-u + (x - \alpha_k) \cdot v$
                    $+ c_k)$
        $\bar{\alpha}_k = -\bar{u}^T v$
        $\bar{\beta}_{k+1} = -\bar{u}^T w$
        $u, v = v, w$
        $\tau = \bar{u}$
        $\bar{u} = \bar{v} + \bar{u} \cdot (x - \alpha_k)$
        $\bar{v} = -\tau \cdot \beta_{k+1}$
    **return** $\bar{x}, \bar{\alpha}, \bar{\beta}, \bar{c}$
```

Figure 4: These algorithms compute the vector-Jacobian products of Evaluate and Interpolate. The arguments u, v of $\overline{\text{Evaluate}}$ are the corresponding values computed during the forward pass, and $\bar{u}$ is an alias of $\bar{c}$.

```
**procedure** $\overline{\text{Interpolate}}(x, \alpha, \beta, y, c, \bar{c})$
    $\bar{y} = V^{-T} \bar{c}$
    $\bar{x} = -\bar{y} \circ (V' c)$
    $\bar{\alpha}, \bar{\beta} = [0, \ldots, 0]$
    **for** $k \in [0, \ldots, n-2]$ **do**
        $c_{n-1} \overset{+}{=} -(\alpha_{n-k-1} - x_k) c_n$
        $\bar{\alpha}_{n-k-1} \overset{+}{=} \bar{c}_{n-1} \cdot c_n$
        $\bar{c}_n = \bar{c}_{n-1} \cdot (\alpha_{n-k-1} - x_k)$
        **for** $j \in [n-2-k, \ldots, 1]$ **do**
            $c_{k+j} \overset{+}{=} -(\alpha_j - x_k) c_{k+j+1} - \beta_{j+1} c_{k+j+2}$
            $\bar{\alpha}_j \overset{+}{=} \bar{c}_{k+j} \cdot c_{k+j+1}$
            $\bar{\beta}_{j+1} \overset{+}{=} \bar{c}_{k+j} \cdot c_{k+j+2}$
            $\bar{c}_{k+j+2} \overset{+}{=} \bar{c}_{k+j} \cdot \beta_{j+1}$
            $\bar{c}_{k+j+1} \overset{+}{=} \bar{c}_{k+j} \cdot (\alpha_j - x_k)$
        $c_k \overset{+}{=} -(\alpha_0 - x_k) c_{k+1} - \beta_1 c_{k+2}$
        $\bar{\alpha}_0 \overset{+}{=} \bar{c}_k \cdot c_{k+1}$
        $\bar{\beta}_1 \overset{+}{=} \bar{c}_k \cdot c_{k+2}$
        $\bar{c}_{k+1} \overset{+}{=} \bar{c}_k \cdot (\alpha_0 - x_k)$
        $\bar{c}_{k+2} \overset{+}{=} \bar{c}_k \cdot \beta_1$
    $c_{n-1} \overset{+}{=} -(\alpha_0 - x_{n-1}) c_n$
    $\bar{\alpha}_0 \overset{+}{=} \bar{c}_{n-1} \cdot c_n$
    **return** $\bar{x}, \bar{\alpha}, \bar{\beta}, \bar{y}$
```

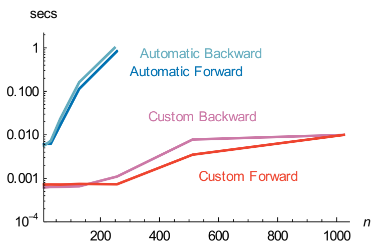

Figure 5: Runtime comparisons of standard (JAX) and custom (CUDA) implementations. The latter is orders of magnitudes faster than the former, for both the forward and backward passes. Presently, algorithms involving sparse updates within nested loops are an edge case for compilers. We expect this situation to improve, but our present investigation would not have been possible without custom implementation.

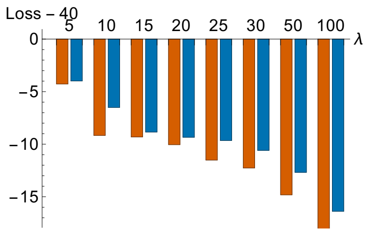

Figure 6: Learning both AnyPT and quantization tables (red) achieves lower loss than learning just the tables (blue). The difference is minimal at lower values of λ, where the target PSNRs are roughly 35-40. This is the regime where the standard JPEG tables are designed to operate. However, as λ increases (and the target PSNR decreases), AnyPT can express a substantially different transform.

4 Learned Polynomial Transforms

The differentiable AnyPT layer encapsulates the Evaluate and Interpolate algorithms. Its parameters are the evaluation points $x \in \mathbb{R}^{n+1}$ and the coefficients $\alpha, \beta \in \mathbb{R}^{n+1}$. Its forward pass is Evaluate and its inverse pass is Interpolate. It is invertible when the x_i are distinct. Its Jacobian is V, whose log-determinant is calculated by an algorithm from Section 2.

4.1 Learned JPEG

Perhaps the most widely-used orthogonal polynomial transform is the discrete cosine transform (DCT) within JPEG image compression (Wallace, 1992). At a high level, JPEG converts RGB channels to one luma (brightness channel) and two chroma (color) channels, and operates on each channel separately. It splits the image into 8x8 patches and applies the 2D DCT, which is equivalent to the DCT applied to each column, followed by the DCT applied to each row. The transformed patches are quantized (rounded to the nearest integer) after pointwise division by an 8x8 quantization table, in which larger values correspond to less visually significant components. The human visual system is less sensitive to high-frequency stimuli. Since the DCT expresses the patch as a combination of different-frequency components, the standard tables can readily discard high-frequency information. Finally, the sequence of integers is losslessly compressed with an entropy-based method.

The quantization tables are learnable parameters of JPEG. They can be learned by proxy objectives (Fung & Parker, 1995) or zero-order methods (Hopkins et al., 2018). By replacing rounding with a smooth approximation, JPEG becomes differentiable (Shin & Song, 2017); thus, the quantization tables can be learned with first-order methods (Luo et al., 2020). We investigate the additional benefit of replacing the DCT by AnyPT. To do so, we use a simple objective which (loosely) captures the tradeoff between visual distortion and compression rate, whose relative importance is set by $\lambda > 0$. To measure the former, we use PSNR, which is a normalized log-squared error. To measure the latter, it is typical to jointly train a measure of entropy, which can be used for the lossless compression step. Since we don't aim to replace this step, we simply use a linear penalty for higher-value coefficients.

We evaluate AnyPT on the CLIC 2020 dataset, from the eponymous image compression challenge (Toderici et al., 2020). As shown in Figure 6, replacing AnyPT with the DCT consistently reduces error. These improvements are modest, but there are other advantages to keeping the compression pipeline mostly intact. The first is interpretability: the explanation of JPEG quantizing visually unimportant components remains valid. Another is ease of training: there are only 24 additional parameters to be trained on a multi-gigabyte dataset.

5 Minimal Values of General Optimization Problems

5.1 Background

Let $W \subseteq \mathbb{R}^N$ be a compact set and let $f : W \mapsto \mathbb{R}$ be a continuous function which (for simplicity) attains its minimum over W. $f^* = \min_{w \in W} f(w)$ is that minimum value. Approximating f^* is NP-hard in general, but may be possible for some practically relevant f. The minimal value f^* should not be confused with the minimizer $w^* \in \mathbb{R}^N$ achieving $f(w^*) = f^*$. In general, it is unclear if knowing f^* helps obtain w^*. Nonetheless, knowing f^* could be practically useful for debugging. For example, if a model is trained to have parameters $\tilde{w}$, and its loss $f(\tilde{w})$ is much larger than f^*, then blame lies with the model's training, rather than its raw expressiveness.

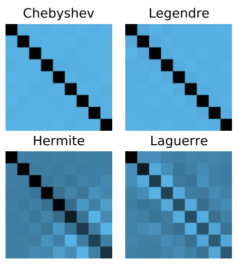

Figure 7: Plots of $\hat{\Sigma}$ for classical orthogonal polynomials. These are formed from $B = 1024$ samples and the correct α, β, μ listed in Figure 1, so $\hat{\Sigma}$ should be close to identity. The Chebyshev and Legendre polynomials behave as desired. The Hermite and Laguerre polynomials suffer in the bottom-right entries involving higher-degree polynomials. This is because Hermite and Laguerre μ have unbounded support, along which high-degree polynomials quickly diverge. The Chebyshev and Legendre μ are supported on $[-1, 1]$. f should ideally be bounded.

Lasserre (2020) recently proposed the following approximation scheme for f^*. Suppose ρ is some probability distribution on W which (for simplicity) is absolutely continuous to Lebesgue measure. We call it a prior since it is ideally concentrated around w^*. Then $\mu(X) = \rho(f^{-1}(X))$ is the distribution of $f(w)$, where $f^{-1}(X) = \{w \in W : f(w) \in X\}$ is the preimage of f. Let α and β be the (unique) recurrence coefficients of the sequence of polynomials orthogonal with respect to ρ. Let λ be the minimum eigenvalue of J_n (or, equivalently, the smallest root of p_{n+1}). Lasserre (2020) shows that, as $n \to \infty$, $\lambda \to f^*$ from above.Laurent & Slot (2020) prove this convergence has rate $O(\log^2 n/n^2)$ if W satisfies a mild geometric condition.

To implement this scheme, Lasserre (2020) observes that the smallest eigenvalue of J_n coincides with the smallest generalized eigenvalue of two $n \times n$ matrices formed from the moments $\mathbf{E}f(w)^k = \mathbf{E}x^k$. In particular, λ is the largest value satisfying $[\mathbf{E}x^{i+j+1}]_{i,j=0}^n \succeq \lambda[\mathbf{E}x^{i+j}]_{i,j=0}^n$. In principle, these moments can be computed if ρ can be sampled and f can be evaluated. However, unless f has special structure, the computation is challenging for large n and N. In particular, the sample complexity of estimating the moments (i.e. the number of evaluations of f) is exponential in n.

5.2 Our Approach

The core problem in this scheme is: given sampling access to the distribution μ, find the unique sequence q^* of μ-orthonormal polynomials, defined by the coefficients (α^*, β^*). By definition, these satisfy $I = \mathbf{E}_{x\sim\mu}[\, q_i^*(x) q_j^*(x)]_{i,j}$. This can be rephrased in terms of mean and covariance using $q_0(x) = 1$ and, via Favard's theorem, $\mathbf{E}\, q_i^*(x) = 0$ for $i > 0$. Let $r = [q_1(x), \ldots, q_n(x)]$ be the rest of $q(x)$, and let $\sigma = \mathbf{E}\, r$. We want to minimize (over α and β) some loss $L(\sigma, \Sigma)$ between 0 and σ, and between I and $\Sigma = \mathbf{E}(r-\sigma)(r-\sigma)^T$. σ and Σ are unknown, but may be estimated from a batch of data. Consider the (regularized) estimators $\hat{\sigma} = \hat{\mathbf{E}}r$ and $\hat{\Sigma} = \frac{B}{B-1}\hat{\mathbf{E}}(r-\hat{\sigma})(r-\hat{\sigma})^T + \delta I$, where $\delta > 0$ and $\hat{\mathbf{E}}$ is the batch mean. When L is convex, $L(\sigma, \Sigma) \le \mathbf{E}\, L(\hat{\sigma}, \hat{\Sigma})$, and the latter can be iteratively minimized. This yields the Mop algorithm (Figure 8), so-named because it reveals the "floor" f^*, or at least abbreviates Minimal value of Optimization Problem.

Implementation. Mop is readily implemented using NEvaluate (Figure 12 in the appendix). Let $x_0, \ldots, x_n \in \mathbb{R}$ have be drawn iid from μ, let V be formed from these points, and let e_i be the ith coordinate vector. Then $Ve_i = [q_i(x_k)]_k$ and $\mathbf{E}\frac{1}{n+1}^T((Ve_i)\circ(Ve_j)) = \mathbf{E}\,\frac{1}{n+1}\sum_{k=0}^n q_i(x_k)q_j(x_k) = \mathbf{E}q_i(x)q_j(x)$. Thus, $\hat{\sigma}$ and $\hat{\Sigma}$ are obtained by calling NEvaluate, with a batch of x, on $e_1, \ldots, e_n$.

Choice of L. An obvious choice for L is the squared Frobenius norm $L_{\mathrm{F}}(\hat{\sigma}, \hat{\Sigma}) = ||\hat{\sigma}||^2 + ||I - \hat{\Sigma}||^2$. In our experience, we encountered more success with $L_{\mathrm{KL}}(\hat{\sigma}, \hat{\Sigma}) = ||\hat{\sigma}||^2 + \mathrm{tr}\,\hat{\Sigma} - \log\det\hat{\Sigma} - (n+1)$, which is the Kullback-Leibler divergence between a standard normal distribution and $N(\hat{\sigma}, \hat{\Sigma})$. One possible explanation is that the sampling distributions of the estimators are normal. Another is a low Jensen gap in $\mathbf{E}\, L_{\mathrm{KL}}(\hat{\Sigma}) = \mathrm{tr}(\Sigma) + \sum_i \mathbf{E}\log\lambda_i(\hat{\Sigma})$.

Stochastic optimization. Our approach extends beyond deterministic objectives f to stochastic objectives which predominate machine learning (Srebro & Tewari, 2010). In this setting, $\mathcal{D}$ is a distribution over examples z — for example, input-output pairs in supervised learning. $f_z(w)$ is typically the loss of weights w on example z. The stochastic objective $F(w) = \mathbf{E}_{z\sim\mathcal{D}} f_z(w)$ is handled in Mop by combining the sampling of $z \sim \mathcal{D}$ and $w \sim \rho$ in μ. In other terms, μ is the posterior loss (or negative log-likelihood), over the data distribution $\mathcal{D}$, of a Bayesian neural network with prior ρ over its parameters w.

```
procedure Mop($f, \mathcal{D}, \rho$)
    procedure $\mu$
        $z \sim \mathcal{D}$
        $w \sim \rho$
        $x = f_z(w)$
        return $x$
    $\alpha, \beta = \arg\min_{\alpha,\beta} \mathbf{E}\, L(\hat{\sigma}, \hat{\Sigma})$
    where $\hat{\sigma} = \hat{\mathbf{E}}\, [q_1(x), \ldots, q_n(x)]$
    and $\hat{\Sigma} \propto \hat{\mathbf{E}}\, [(q_i(x) - \hat{\sigma}_i)(q_j(x) - \hat{\sigma}_i)]_{i,j=1}^n$
    $\lambda$ = smallest eigenvalue of $J_n$
    return $\lambda$
```

Figure 8: Mop iteratively draws a batch from μ, forms the estimates $\hat{\sigma}$ and $\hat{\Sigma}$, and descends on their loss. $f_z(w)$ is the loss of parameters w on the example z. ρ is the prior over w and $\mathcal{D}$ is the data distribution of z. μ is a distribution over $\mathbb{R}$. Larger n trades more computation for better approximation. The expectation $\mathbf{E}_x$ is over $x \in \mathbb{R}$ drawn iid μ. $\hat{\mathbf{E}}$ is the empirical expectation formed from a batch. Recall J_n is defined in (3).

Limitations and failure modes. As mentioned, approximating f^* is a computationally intractable problem. Furthermore, Mop is a statistical query algorithm (Kearns, 1998): it uses gradient estimates computed from batches of data, and does not directly manipulate individual examples. It is therefore subject to stronger (information-theoretic) lower bounds than the previously-mentioned NP-hardness of minimal value approximation (Reyzin, 2020). Accordingly, there are different ways in which Mop can fail or demand exorbitant resources. It is possible to encounter local minima or saddle points of L with respect to α and β. The approximation of f^* by λ occurs asymptotically as n grows, so a large n may be required for some f. Larger n, in turn, demand more samples of f. $\lambda \to f^*$ from above for J_n defined by the optimal (α^*, β^*); this is not necessarily true for suboptimal $(\tilde{\alpha}, \tilde{\beta})$ obtained by an inexact algorithm. Furthermore, Lasserre's result does not take finite-sample approximation of moments (or in our setting, covariances) into consideration.

5.3 Basic Empirical Evaluation

As a preliminary test, we apply the algorithm to f with known f^*. Some of these problems are easy, and Mop is able to solve them. One of them is impossible to solve, so Mop fails. Details of these experiments are given in the appendix.

1. Recovering classical orthogonal polynomials. Mop defines μ in terms of f and ρ. Before doing that, let us simply take the μ listed in Figure 1, and see if minimizing $L(\hat{\sigma}, \hat{\Sigma})$ recovers the listed coefficients (α^*, β^*). As depicted in Figure 9, success depends on the choice of L and μ. In all cases, L_{KL} achieved lower error than L_{F}. The Chebyshev and Legendre coefficents were easier to recover than the Hermite and Laguerre ones. This agrees with the intuitions in Figure 7.

2. Test polynomials. Lasserre's scheme has been run on 4 standard test polynomials (Lasserre, 2020). These are bivariate ($N = 2$) and have minimum value $f^* = 0$ on $W = [-1, 1]^N$. We normalized the polynomials to take values in $[0, 1]$. To distinguish the algorithm's behavior from trivially returning 0, we added constants (either $0.2, 0.4, 0.6$, or 0.8), shifting the f^*. Figure 10 shows L_{KL} reasonably approximates f^*, whereas the L_{F} does not. Due to the normalization, our results are quantitatively different than those previously reported, but are qualitatively the same: the Matyas polynomial is the easiest, and the camel polynomial is the hardest (Laurent & Slot, 2020). Note that λ produced midway during optimization are neither upper nor lower bounds of f^*; it is important to completely optimize α and β.

3. Learning halfspaces / noisy parities. Let x be uniform on the hypercube $\{-1, 1\}^N$. For some noise rate $\eta > 0$, y is the parity of x with probability $1 - \eta$, and is negated with probability η. Let $f^* = \arg\min_{w \in \mathbb{R}^N} -\mathbf{E} h_w(x) y$ be the (negated) correlation of the best possible (smooth) halfspace $h_w(x) = \tanh(w^T x)$. Using a statistical query algorithm, it is impossible to distinguish f^* from zero, by the reduction of Kalai et al. (2008) to learning noisy parities (Blum et al., 1994). For $N = 16$ and $\eta = 0$ we attempt to use Mop to approximate f^*. For varying values of n, we run Mop samples (x, y). For distinguishment, we run it on $(x, \tilde{y})$ where $\tilde{y}$ are random signs. In Figure 11, we see that Mop does not distinguish these distributions, regardless of n.

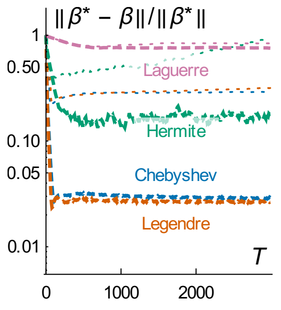

Figure 9: Mop recovering β^* for different classical polynomials. Thick dashed lines use L_{KL}. Dotted lines use L_{F}.

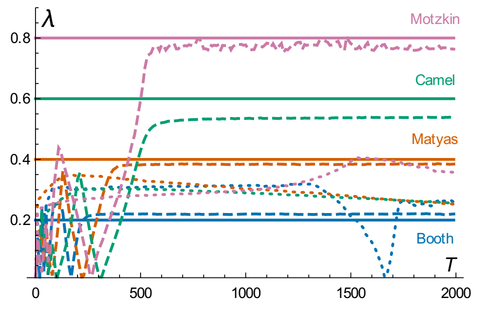

Figure 10: Mop on toy polynomials. Solid lines are true f^*, thick dashed lines are Mop using L_{KL}, and dotted lines use L_{F}. Mop using L_{KL} approximates the f^*, but only after optimization is complete.

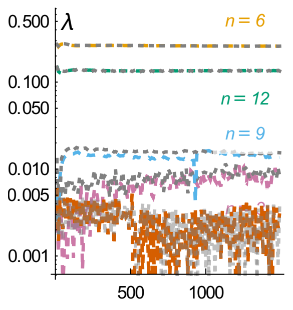

Figure 11: Mop fails to distinguish parity data (colored lines) from noise (gray lines), no matter the value of n.

6 Related Work

Reversibility underpins low-memory, reverse-mode automatic differentiation (Gomez et al., 2017). Besides our manual derivation, there are other ways to obtain our VJPs, though they would require comparable effort, and may not be as practical. If an algorithm is written in a reversible programming language — which is not a trivial rewriting — then its VJPs can be computed with low memory overhead (Liu & Zhao, 2020). Forward mode differentiation uses $O(n)$ memory, but generally requires a factor $O(n)$ more computation, and needs software support to intermix with reverse mode.

Learned image compression algorithms usually replace traditional compression pipelines with neural networks (Jiang, 1999). They tend to achieve excellent compression results at the expense of computational complexity. Even as machine learning hardware becomes faster, there will likely be a role for fast, simple compression algorithms. This is amply demonstrated by the most powerful GPU ever released, which adds five dedicated cores for JPEG decoding (Lisiecki et al., 2020).

Orthogonal polynomial transforms are subsumed by recently-developed representations of structured linear maps, such as tridiagonal factorizations of low-displacement rank operators (Thomas et al., 2018), and butterfly factorizations culminating in the Kaleidoscope hierarchy (Dao et al., 2019; 2020). It is appropriate to think of unstructured matrices, Kaleidoscope, AnyPT, and fixed polynomial transforms as varying tradeoffs between expressive power and tractability.

Our results partially extend to the complex domain. The Fast Fourier Transform, perhaps the most well-known orthogonal polynomial transform, involves the monomials ($\alpha = \beta = 0$) of the roots of unity $x_i \in \mathbb{C}$. This can be handled by our algorithms, and indeed the original version of Interpolate (Björck & Pereyra, 1970). The Szegö polynomials, which are orthogonal with respect to a Hermitian inner product on the unit circle, need a variant of Interpolate (Bella et al., 2007).

The problem of finding the orthogonal polynomials (as α, β) which match the given distribution (as moments m) was first studied by Chebyshev. The map $m \mapsto (\alpha, \beta)$ is ill-conditioned (Gautschi, 1967), so it is preferable to begin with "modified" moments (Gautschi, 2004).

7 Conclusions and Future Work

This work enables gradient-based optimization over sequences of orthogonal polynomials, and takes an initial foray into such optimization problems. Based on the algorithm of Bella et al. (2009), our core techniques might extend from polynomial Vandermonde matrices to quasiseparable matrices. With sufficient computational resources, Mop could be applied larger problems, especially modern neural networks whose minima are unknown. It may be possible to develop satisfactory theory for Mop: it has a well-defined optimum (α^*, β^*) which achieves a known (true) loss of 0, is simultaneously understood via Lassere's moment-based approach.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://github.com/tensorflow/tensorflow/blob/e61bc26e00f48db9abaf165f343f1f44a10227a9/tensorflow/python/ops/linalg_grad.py#L539`. Gradient for MatrixSolve.

Stephen Barnett. Some applications of the comrade matrix. *International Journal of Control*, 21(5): 849–855, 1975.

Roberto Barrio. Parallel algorithms to evaluate orthogonal polynomial series. *SIAM Journal on Scientific Computing*, 21(6):2225–2239, 2000.

Atılım Günes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research*, 18(1):5595–5637, 2017.

Tom Bella, Yuli Eidelman, Israel Gohberg, Israel Koltracht, and Vadim Olshevsky. A björck–pereyra-type algorithm for szegö–vandermonde matrices based on properties of unitary hessenberg matrices. *Linear algebra and its applications*, 420(2-3):634–647, 2007.

Tom Bella, Yuli Eidelman, Israel Gohberg, and Vadim Olshevsky. Computations with quasiseparable polynomials and matrices. *Theoretical Computer Science*, 409(2):158–179, 2008.

Tom Bella, Yuli Eidelman, Israel Gohberg, Israel Koltracht, and Vadim Olshevsky. A fast björck–pereyra-type algorithm for solving hessenberg-quasiseparable-vandermonde systems. *SIAM Journal on Matrix Analysis and Applications*, 31(2):790–815, 2009.

Jean-Paul Berrut and Lloyd N Trefethen. Barycentric lagrange interpolation. *SIAM review*, 46(3): 501–517, 2004.

Ake Björck and Victor Pereyra. Solution of vandermonde systems of equations. *Mathematics of computation*, 24(112):893–903, 1970.

Avrim Blum, Merrick Furst, Jeffrey Jackson, Michael Kearns, Yishay Mansour, and Steven Rudich. Weakly learning dnf and characterizing statistical query learning using fourier analysis. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pp. 253–262, 1994.

Ignace Bogaert. Iteration-free computation of gauss–legendre quadrature nodes and weights. *SIAM Journal on Scientific Computing*, 36(3):A1008–A1026, 2014.

Alin Bostan, Bruno Salvy, and Éric Schost. Fast conversion algorithms for orthogonal polynomials. *Linear Algebra and its Applications*, 432(1):249–258, 2010.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

D Calvetti and L Reichel. Fast inversion of vandermonde-like matrices involving orthogonal polynomials. *BIT Numerical Mathematics*, 33(3):473–484, 1993.

Ed S Coakley and Vladimir Rokhlin. A fast divide-and-conquer algorithm for computing the spectra of real symmetric tridiagonal matrices. *Applied and Computational Harmonic Analysis*, 34(3): 379–414, 2013.

Tri Dao, Albert Gu, Matthew Eichhorn, Atri Rudra, and Christopher Re. Learning fast algorithms for linear transforms using butterfly factorizations. volume 97 of *Proceedings of Machine Learning Research*, pp. 1517–1527, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL `http://proceedings.mlr.press/v97/dao19a.html`.

Tri Dao, Nimit Sohoni, Albert Gu, Matthew Eichhorn, Amit Blonder, Megan Leszczynski, Atri Rudra, and Christopher Ré. Kaleidoscope: An efficient, learnable representation for all structured linear maps. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=BkgrBgSYDS`.

James R Driscoll, Dennis M Healy Jr, and Daniel N Rockmore. Fast discrete polynomial transforms with applications to data analysis for distance transitive graphs. *SIAM Journal on Computing*, 26 (4):1066–1099, 1997.

Hei Tao Fung and Kevin J Parker. Design of image-adaptive quantization tables for jpeg. *Journal of Electronic Imaging*, 4(2):144–151, 1995.

Walter Gautschi. Computational aspects of three-term recurrence relations. *SIAM review*, 9(1): 24–82, 1967.

Walter Gautschi. How (un) stable are vandermonde systems. *Asymptotic and computational analysis*, 124:193–210, 1990.

Walter Gautschi. *Orthogonal polynomials*. Oxford university press Oxford, 2004.

Walter Gautschi. Orthogonal polynomials (in matlab). *Journal of computational and applied mathematics*, 178(1-2):215–234, 2005.

Israel Gohberg and Vadim Olshevsky. Fast algorithms with preprocessing for matrix-vector multiplication problems. *Journal of Complexity*, 10(4):411–427, 1994.

Israel Gohberg and Vadim Olshevsky. The fast generalized parker–traub algorithm for inversion of vandermonde and related matrices. *Journal of Complexity*, 13(2):208–234, 1997.

Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. The reversible residual network: Backpropagation without storing activations. In *Advances in neural information processing systems*, pp. 2214–2224, 2017.

Nicholas J Higham. Error analysis of the björck-pereyra algorithms for solving vandermonde systems. *Numerische Mathematik*, 50(5):613–632, 1987.

Nicholas J Higham. Fast solution of vandermonde-like systems involving orthogonal polynomials. *IMA Journal of Numerical Analysis*, 8(4):473–486, 1988.

Nicholas J Higham. Stability analysis of algorithms for solving confluent vandermonde-like systems. *SIAM Journal on Matrix Analysis and Applications*, 11(1):23–41, 1990.

Nicholas J Higham. Iterative refinement enhances the stability ofqr factorization methods for solving linear equations. *BIT Numerical Mathematics*, 31(3):447–468, 1991.

Nicholas J Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002.

Max Hopkins, Michael Mitzenmacher, and Sebastian Wagner-Carena. Simulated annealing for jpeg quantization. In *2018 Data Compression Conference*, pp. 412–412. IEEE, 2018.

Mourad Ismail, Mourad EH Ismail, and Walter van Assche. *Classical and quantum orthogonal polynomials in one variable*, volume 13. Cambridge university press, 2005.

J Jiang. Image compression with neural networks–a survey. *Signal processing: image Communication*, 14(9):737–760, 1999.

Thomas Kailath and Vadim Olshevsky. Displacement-structure approach to polynomial vandermonde and related matrices. *Linear Algebra and Its Applications*, 261(1-3):49–90, 1997.

Adam Tauman Kalai, Adam R Klivans, Yishay Mansour, and Rocco A Servedio. Agnostically learning halfspaces. *SIAM Journal on Computing*, 37(6):1777–1805, 2008.

Michael Kearns. Efficient noise-tolerant learning from statistical queries. *Journal of the ACM (JACM)*, 45(6):983–1006, 1998.

Jean B Lasserre. Connecting optimization with spectral analysis of tri-diagonal matrices. *Mathematical Programming*, pp. 1–15, 2020.

Monique Laurent and Lucas Slot. Near-optimal analysis of univariate moment bounds for polynomial optimization. *arXiv preprint arXiv:2001.11289*, 2020.

Gregory Leibon, Daniel N Rockmore, Wooram Park, Robert Taintor, and Gregory S Chirikjian. A fast hermite transform. *Theoretical computer science*, 409(2):211–228, 2008.

Janusz Lisiecki, Michal Szolucha, Joaquin Anton Guirao, and Maitreyi Roy. Loading data fast with dali and the new hardware jpeg decoder in nvidia a100 gpus, 2020. URL `https://developer.nvidia.com/blog/loading-data-fast-with-dali-and-new-jpeg-decoder-in-a100/`.

Jin-Guo Liu and Taine Zhao. Differentiate everything with a reversible programming language. *arXiv preprint arXiv:2003.04617*, 2020.

Xiyang Luo, Hossein Talebi, Feng Yang, Michael Elad, and Peyman Milanfar. The rate-distortion-accuracy tradeoff: Jpeg case study. *arXiv preprint arXiv:2008.00605*, 2020.

Daniel Potts. Fast algorithms for discrete polynomial transforms on arbitrary grids. *Linear algebra and its applications*, 366:353–370, 2003.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*. Cambridge University Press, USA, 1992. ISBN 0521431085.

Thomas W Reps and Louis B Rall. Computational divided differencing and divided-difference arithmetics. *Higher-order and symbolic computation*, 16(1-2):93–149, 2003.

Lev Reyzin. Statistical queries and statistical algorithms: Foundations and applications. *arXiv preprint arXiv:2004.00557*, 2020.

Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International Conference on Machine Learning*, pp. 1530–1538, 2015.

Richard Shin and Dawn Song. Jpeg-resistant adversarial images. In *NIPS 2017 Workshop on Machine Learning and Computer Security*, volume 1, 2017.

Francis J Smith. An algorithm for summing orthogonal polynomial series and their derivatives with applications to curve-fitting and interpolation. *Mathematics of Computation*, 19(89):33–36, 1965.

Alicja Smoktunowicz. Backward stability of clenshaw's algorithm. *BIT Numerical Mathematics*, 42(3):600–610, 2002.

Nathan Srebro and Ambuj Tewari. Stochastic optimization for machine learning. *ICML Tutorial*, 2010.

Anna Thomas, Albert Gu, Tri Dao, Atri Rudra, and Christopher Ré. Learning compressed transforms with low displacement rank. In *Advances in neural information processing systems*, pp. 9052–9060, 2018.

George Toderici, Wenzhe Shi, Radu Timofte, Lucas Theis, Johannes Balle, Eirikur Agustsson, Nick Johnston, and Fabian Mentzer. Workshop and challenge on learned image compression (clic2020), 2020. URL `http://www.compression.cc`.

Paul Valiant. Three perspectives on orthogonal polynomials. FOCS 2016 Workshop: (Some) Orthogonal Polynomials and their Applications to TCS, 2016. URL `http://www.cs.columbia.edu/~ccanonne/workshop-focs2016/`.

Gregory K Wallace. The jpeg still picture compression standard. *IEEE transactions on consumer electronics*, 38(1):xviii–xxxiv, 1992.

```
procedure NEvaluate($x, \alpha, \gamma, c$)
    $u = [c_n, \dots, c_n]$
    $v = [0, \dots, 0]$
    for $k \in [n-1, \dots, 0]$ do
        $\tau = u$
        $u = ((x - \alpha_k)/\gamma_{k+1}) \cdot u$
            $- \frac{\gamma_{k+1}}{\gamma_{k+2}} v + c_k$
        $v = \tau$
    $\mu = u/\gamma_0$
    return $\mu$
```

```
procedure NInterpolate($x, \alpha, \gamma, y$)
    $\delta = \text{DivDiffs}(x, y)$
    return NChangeBasis($x, \alpha, \gamma, \delta$)

procedure NChangeBasis($x, \alpha, \gamma, \delta$)
    $c = \delta$
    $c_{n-1} \mathrel{+}= (\alpha_0 - x_{n-1})c_n$
    $c_n = \gamma_1 c_n$
    for $k \in [n-2, \dots, 0]$ do
        $c_k \mathrel{+}= (\alpha_0 - x_k)c_{k+1} + \gamma_1 c_{k+2}$
        for $j \in [1, \dots, n-2-k]$ do
            $c_{k+j} = \gamma_j c_{k+j} + (\alpha_j - x_k)c_{k+j+1}$
                $+\gamma_{j+1} c_{k+j+2}$
        $c_{n-1} = \gamma_{n-k-1} c_{n-1} + (\alpha_{n-k-1} - x_k)c_n$
        $c_n = \gamma_{n-k} c_n$
    $\sigma = \gamma_0 \cdot c$
    return $\sigma$
```

Figure 12: Variants of Evaluate and Interpolate for orthonormal polynomial sequences. As in Interpolate, DivDiffs is the first row of the table of divided differences.

```
procedure V^T Multiply($x, \alpha, \beta, c$)
    $q = [0, \dots, 0]$
    $p = [1, \dots, 1]$
    $y = [0, \dots, 0]$
    for $i \in [0, \dots, n]$ do
        for $j \in [0, \dots, n]$ do
            $y_i \mathrel{+}= p_j \cdot c_j$
            if $i < n$ then
                $\tau = p_j$
                $p_j = (x_j - \alpha_i)p_j - \beta_i q_j$
                $q_j = \tau$
    return $y$
```

```
procedure NV^T Multiply($x, \alpha, \gamma, c$)
    $q = [0, \dots, 0]$
    $p = [\gamma_0^{-1}, \dots, \gamma_0^{-1}]$
    $y = [0, \dots, 0]$
    for $i \in [0, \dots, n]$ do
        for $j \in [0, \dots, n]$ do
            $y_i \mathrel{+}= p_j \cdot c_j$
            if $i < n$ then
                $\tau = p_j$
                $p_j = ((x_j - \alpha_i)/\gamma_{i+1})p_j - \frac{\gamma_i}{\gamma_{i+1}} q_j$
                $q_j = \tau$
    return $y$
```

Figure 13: Transpose Vandermonde multiplication for monic orthogonal (left) and orthonormal (right) polynomial sequences. These use $O(n^2)$ time using $O(n)$ space. They are used in the following algorithms for VJPs.

A Appendix

A.1 Numerical Stability of Interpolate

The divided differences of y at x are the following $\frac{1}{2}(n+1)(n+2)$ recursively-defined values:

$$y[x_i] = y_i \quad y[x_0, x_1] = \frac{y[x_1] - y[x_0]}{x_1 - x_0} \quad y[x_j, \dots, x_k] = \frac{y[x_{j+1}, \dots, x_k] - y[x_j, \dots, x_{k-1}]}{x_k - x_j} \tag{5}$$

DivDiffs$(x, y) = [y[x_0], y[x_0, x_1], \dots, y[x_0, \dots, x_n]] \in \mathbb{F}^{n+1}$ is called the first row of the table of divided differences. They are also known as the coefficients of the Newton interpolant of the data (x, y) (Berrut & Trefethen, 2004). Interpolate starts by computing these coefficients δ. Then, it changes the basis of these coefficients to the specified orthogonal polynomial sequence. The blame for its instability lies with the first step: naive calculation of divided differences, according to the definition (5), can lead to severe numerical error.

This error can be ameliorated in some simple ways. The most appropriate way depends on the amount of control over the input data. If x can be chosen arbitrarily, then complex nodes can be more stable (Gautschi, 1990). In particular, consider setting $x_k = e^{-2\pi i z_k}$ where z is low-discrepancy sequence, such as the van der Corput sequence. Intuitively, such sequences will keep denominators $x_i - x_j$ close to uniform. If x is real, then stability strongly depends on the ordering of $x_0, \ldots, x_n$ (Gohberg & Olshevsky, 1997; Higham, 1990). Choosing x randomly, as is typical when training a neural network, is a poor choice. If x is in increasing order, then divided differencing is (relatively) stable (Higham, 1987). If some points are nonnegative, then the Leja ordering explicitly maximizes the relevant denominators $x_i - x_j$ (Higham, 1990). If all these options are exhausted, iterative refinement can eliminate moderate amounts of error (Higham, 1991).

The aforementioned fixes may work in specific scenarios, but they do not satisfactorily solve the numerical instability of divided differencing. Fortunately, there is a principled, general solution to this problem. Divided differencing, like differentiation, is a composable function transformation (Reps & Rall, 2003): given a function f, the function $x \mapsto f[x_0, \ldots, x_n]$ computing its divided difference can be automatically derived. Existing software packages for machine learning support the addition of such transformations (Bradbury et al., 2018).

Thus, the Interpolate algorithm of Higham (1988)'s algorithm is both simple and worthwhile in the long term. The Parker algorithm, which computes V^{-1} and then dense-multiplies $V^{-1}y$, is known to be more numerically stable (Gohberg & Olshevsky, 1997; Calvetti & Reichel, 1993). However, it requires $O(n^2)$ memory. Specialized Gaussian elimination with partial pivoting (Kailath & Olshevsky, 1997) takes $O(n^2)$ time and $O(n)$ memory. However, its implementation is involved, and is asymptotically slower than Fourier-based methods. Some of the cited $O(n \log^2 n)$ algorithms in Figure 2 may also be numerically unstable; see, for example, Remark 4.2 in Bella et al. (2008).

A.2 Vector-Jacobian Products for Evaluate and Interpolate

The following algorithms have been numerically checked against finite differencing. We use standard "overbar" notation for adjoints (Baydin et al., 2017). Let ℓ be the final scalar loss produced in the entire forward pass. The adjoint of each intermediate variable u_i is $\bar{u}_i = \frac{\partial \ell}{du_i}$.

A.2.1 Evaluate

By linearity, $\bar{c} = \bar{y}^T \nabla_c Vc = (\bar{y}^T V)^T = V^T \bar{y}$. By similarly elementary operations:

$$\bar{x} = \bar{y}^T \left[\frac{dVc}{dx_k}\right]_k = \left[\sum_i \bar{y}_i \sum_j \mathbf{1}(i=k) p_j'(x_i) c_j\right]_k = \left[\bar{y}_k \sum_j p_j'(x_k) c_j\right]_k = \bar{y} \circ (V'c)$$

Now we derive the VJPs with respect to α and β. Here is Evaluate written with indexed notation for u, which distinguishes the intermediate values.

```
procedure Evaluate($x, \alpha, \beta, c$)
    $u^{(n)} = [c_n, \ldots, c_n]$
    $u^{(n+1)} = [0, \ldots, 0]$
    for $k \in [n-1, \ldots, 0]$ do
        $u^{(k)} = (x - \alpha_k) \cdot u^{(k+1)} - \beta_{k+1} u^{(k+2)} + c_k$
    return $u^{(0)}$
```

The adjoints are computed via standard reverse accumulation. Also, exploiting reversibility, prior values $u^{(k+2)}$ can be computed from later values $u^{(k+1)}$.

$$\begin{aligned}
\bar{u}^{(k+1)} &= \bar{u}^{(k)} \cdot (x - \alpha_k) \\
\bar{u}^{(k+2)} &= \bar{u}^{(k)} \cdot (-\beta_{k+1}) \\
\bar{\alpha}_k &= \bar{u}^{(k)} \cdot (-u^{(k+1)}) \\
\bar{\beta}_{k+1} &= \bar{u}^{(k)} \cdot (-u^{(k+2)}) \\
u^{(k+2)} &= \frac{1}{\beta_{k+1}} (-u^{(k)} + (x - \alpha_k) \cdot u^{(k+1)} + c_k)
\end{aligned}$$

Performing these computations, in the reverse order of Evaluate, yields the VJP for α and β.

```
procedure $\overline{\text{Evaluate}}(x, \alpha, \beta, u^{(0)}, u^{(1)}, \bar{u}^{(0)}, \bar{u}^{(1)})$
    $\bar{\alpha} = \bar{\beta} = [0, \ldots, 0]$
    for $k \in [0, \ldots, n-1]$ do
        $u^{(k+2)} = \frac{1}{\beta_{k+1}}(-u^{(k)} + (x - \alpha_k) \cdot u^{(k+1)} + c_k)$
        $\bar{\alpha}_k = -\bar{u}^{(k)} \cdot u^{(k+1)}$
        $\bar{\beta}_{k+1} = -\bar{u}^{(k)} \cdot u^{(k+2)}$
        $\bar{u}^{(k+1)} \overset{+}{=} \bar{u}^{(k)} \cdot (x - \alpha_k)$
        $\bar{u}^{(k+2)} = \bar{u}^{(k)} \cdot (-\beta_{k+1})$
    return $\bar{\alpha}, \bar{\beta}$
```

Note that $\bar{v} = 0$ since it is computed during the forward pass, but is not part of the output of Evaluate. The full VJP for Evaluate, given in Figure 4, includes the computation of $\bar{x}$ and $\bar{c}$, and elides the indexing of intermediate values.

A.2.2 INTERPOLATE

It is known that $\bar{c}^T \nabla_y c$ is the solution $\bar{y}$ to $V^T \bar{y} = \bar{c}$ (Abadi et al., 2015). Also, $\bar{c}^T \nabla_V c = -\bar{y} c^T$. By the chain rule, $\nabla_x c = \frac{\partial c}{dV} \circ \frac{\partial V}{dx}$. By the definition of V, $\frac{\partial V_{i,j}}{dx_k} = \mathbf{1}(i = k) p'_j(x_k)$. Therefore, similarly to the previous derivation for evaluation:

$$\bar{x} = \bar{c}^T \nabla_x c = \left[-\bar{y} c^T \circ \frac{dV}{dx_k}\right]_k = \Big[\sum_{i,j} -\bar{y}_i c_j \mathbf{1}(i = k) p'_j(x_i)\Big]_k = \Big[\sum_j -\bar{y}_k c_j p'_j(x_k)\Big]_k$$
$$= -\bar{y} \circ \Big[\sum_j c_j p'_j(x_k)\Big]_k = -\bar{y} \circ (V'c)$$

For $\bar{\alpha}$ and $\bar{\beta}$, we write ChangeBasis in the indexed notation of Higham (1988).

```
procedure ChangeBasis($x, \alpha, \beta, \delta$)
    $c^{(n)} = \delta$
    $c_{n-1}^{(n-1)} = c_{n-1}^{(n)} + (\alpha_0 - x_{n-1}) c_n^{(n)}$
    $c_n^{(n-1)} = c_n^{(n)}$
    for $k \in [n-2, \ldots, 0]$ do
        $c_k^{(k)} = c_k^{(k+1)} + (\alpha_0 - x_k) c_{k+1}^{(k+1)} + \beta_1 c_{k+2}^{(k+1)}$
        for $j \in [1, \ldots, n-2-k]$ do
            $c_{k+j}^{(k)} = c_{k+j}^{(k+1)} + (\alpha_j - x_k) c_{k+j+1}^{(k+1)} +$
$\beta_{j+1} c_{k+j+2}^{(k+1)}$
        $c_{n-1}^{(k)} = c_{n-1}^{(k+1)} + (\alpha_{n-k-1} - x_k) c_n^{(k+1)}$
        $c_n^{(k)} = c_n^{(k+1)}$
    return $c^{(0)}$
```

We derive the reverse-mode adjoints in the usual manner.

$$\begin{aligned}
c_n^{(k)} &= c_n^{(k+1)} & \bar{c}_n^{(k+1)} &= \bar{c}_n^{(k)} \\
c_{n-1}^{(k)} &= c_{n-1}^{(k+1)} + (\alpha_{n-k-1} - x_k)c_n^{(k+1)} & \bar{c}_{n-1}^{(k+1)} &= \bar{c}_{n-1}^{(k)} \\
& & \bar{c}_n^{(k+1)} &= \bar{c}_{n-1}^{(k)} \cdot (\alpha_{n-k-1} - x_k) \\
& & \bar{\alpha}_{n-k-1} &\mathrel{\pm} \bar{c}_{n-1}^{(k)} \cdot c_n^{(k+1)} \\
c_{k+j}^{(k)} &= c_{k+j}^{(k+1)} + (\alpha_j - x_k)c_{k+j+1}^{(k+1)} + \beta_{j+1}c_{k+j+2}^{(k+1)} & \bar{c}_{k+j}^{(k+1)} &= \bar{c}_{k+j}^{(k)} \\
& & \bar{c}_{k+j+1}^{(k+1)} &= \bar{c}_{k+j}^{(k)} \cdot (\alpha_j - x_k) \\
& & \bar{c}_{k+j+2}^{(k+1)} &= \bar{c}_{k+j}^{(k)} \cdot \beta_{j+1} \\
& & \bar{\alpha}_j &\mathrel{\pm} \bar{c}_{k+j}^{(k)} \cdot c_{k+j+1}^{(k+1)} \\
& & \bar{\beta}_{j+1} &\mathrel{\pm} \bar{c}_{k+j}^{(k)} \cdot c_{k+j+2}^{(k+1)} \\
c_k^{(k)} &= c_k^{(k+1)} + (\alpha_0 - x_k)c_{k+1}^{(k+1)} + \beta_1 c_{k+2}^{(k+1)} & \bar{c}_k^{(k+1)} &= \bar{c}_k^{(k)} \\
& & \bar{c}_{k+1}^{(k+1)} &= \bar{c}_k^{(k)} \cdot (\alpha_0 - x_k) \\
& & \bar{c}_{k+2}^{(k+1)} &= \bar{c}_k^{(k)} \cdot \beta_1 \\
& & \bar{\alpha}_0 &\mathrel{\pm} \bar{c}_k^{(k)} \cdot c_{k+1}^{(k+1)} \\
& & \bar{\beta}_1 &\mathrel{\pm} \bar{c}_k^{(k)} \cdot c_{k+2}^{(k+1)} \\
c_n^{(n-1)} &= c_n^{(n)} & \bar{c}_n^{(n)} &= \bar{c}_n^{(n-1)} \\
c_{n-1}^{(n-1)} &= c_{n-1}^{(n)} + (\alpha_0 - x_{n-1})c_n^{(n)} & \bar{c}_{n-1}^{(n)} &= \bar{c}_{n-1}^{(n-1)} \\
& & \bar{c}_n^{(n)} &= \bar{c}_{n-1}^{(n-1)} \cdot (\alpha_0 - x_{n-1}) \\
& & \bar{\alpha}_0 &\mathrel{\pm} \bar{c}_{n-1}^{(n-1)} \cdot c_n^{(n)}
\end{aligned}$$

Observe that the reverse accumulation involves values $c^{(k)}$ for $k > 0$. As with Evaluate, we can compute $c^{(k+1)}$ from $c^{(k)}$.

$$\begin{aligned}
c_n^{(k)} = c_n^{(k+1)} &\iff c_n^{(k+1)} = c_n^{(k)} \\
c_{n-1}^{(k)} = c_{n-1}^{(k+1)} + (\alpha_{n-k-1} - x_k)c_n^{(k+1)} &\iff c_{n-1}^{(k+1)} = c_{n-1}^{(k)} - (\alpha_{n-k-1} - x_k)c_n^{(k+1)} \\
c_{k+j}^{(k)} = c_{k+j}^{(k+1)} + (\alpha_j - x_k)c_{k+j+1}^{(k+1)} + \beta_{j+1}c_{k+j+2}^{(k+1)} &\iff c_{k+j}^{(k+1)} = c_{k+j}^{(k)} - (\alpha_j - x_k)c_{k+j+1}^{(k+1)} - \beta_{j+1}c_{k+j+2}^{(k+1)} \\
c_k^{(k)} = c_k^{(k+1)} + (\alpha_0 - x_k)c_{k+1}^{(k+1)} + \beta_1 c_{k+2}^{(k+1)} &\iff c_k^{(k+1)} = c_k^{(k)} - (\alpha_0 - x_k)c_{k+1}^{(k+1)} - \beta_1 c_{k+2}^{(k+1)} \\
c_n^{(n-1)} = c_n^{(n)} &\iff c_n^{(n)} = c_n^{(n-1)} \\
c_{n-1}^{(n-1)} = c_{n-1}^{(n)} + (\alpha_0 - x_{n-1})c_n^{(n)} &\iff c_{n-1}^{(n)} = c_{n-1}^{(n-1)} - (\alpha_0 - x_{n-1})c_n^{(n)}
\end{aligned}$$

Combining the adjoint equations with the reversed computation of $c^{(k)}$, we obtain the VJP with respect to α and β. The full algorithm in Figure 4 elides no-ops and the indexed notation.

```
**procedure** $\overline{\text{ChangeBasis}}(x, \alpha, \beta, \delta, c^{(0)}, \bar{c}^{(0)})$
    $\bar{\alpha}, \bar{\beta} = [0, \ldots, 0]$
    **for** $k \in [0, \ldots, n-2]$ **do**
        $c_n^{(k+1)} = c_n^{(k)}$
        $c_{n-1}^{(k+1)} = c_{n-1}^{(k)} - (\alpha_{n-k-1} - x_k)c_n^{(k+1)}$
        $\bar{\alpha}_{n-k-1} \overset{+}{=} \bar{c}_{n-1}^{(k)} \cdot c_n^{(k+1)}$
        $\bar{c}_n^{(k+1)} = \bar{c}_{n-1}^{(k)} \cdot (\alpha_{n-k-1} - x_k)$
        $\bar{c}_{n-1}^{(k+1)} = \bar{c}_{n-1}^{(k)}$
        **for** $j \in [n-2-k, \ldots, 1]$ **do**
            $c_{k+j}^{(k+1)} = c_{k+j}^{(k)} - (\alpha_j - x_k)c_{k+j+1}^{(k+1)} - \beta_{j+1}c_{k+j+2}^{(k+1)}$
            $\bar{\alpha}_j \overset{+}{=} \bar{c}_{k+j}^{(k)} \cdot c_{k+j+1}^{(k+1)}$
            $\bar{\beta}_{j+1} \overset{+}{=} \bar{c}_{k+j}^{(k)} \cdot c_{k+j+2}^{(k+1)}$
            $\bar{c}_{k+j+2}^{(k+1)} \overset{+}{=} \bar{c}_{k+j}^{(k)} \cdot \beta_{j+1}$
            $\bar{c}_{k+j+1}^{(k+1)} \overset{+}{=} \bar{c}_{k+j}^{(k)} \cdot (\alpha_j - x_k)$
            $\bar{c}_{k+j}^{(k+1)} = \bar{c}_{k+j}^{(k)}$
        $c_k^{(k+1)} = c_k^{(k)} - (\alpha_0 - x_k)c_{k+1}^{(k+1)} - \beta_1 c_{k+2}^{(k+1)}$
        $\bar{\alpha}_0 \overset{+}{=} \bar{c}_k^{(k)} \cdot c_{k+1}^{(k+1)}$
        $\bar{\beta}_1 \overset{+}{=} \bar{c}_k^{(k)} \cdot c_{k+2}^{(k+1)}$
        $\bar{c}_k^{(k+1)} = \bar{c}_k^{(k)}$
        $\bar{c}_{k+1}^{(k+1)} \overset{+}{=} \bar{c}_k^{(k)} \cdot (\alpha_0 - x_k)$
        $\bar{c}_{k+2}^{(k+1)} \overset{+}{=} \bar{c}_k^{(k)} \cdot \beta_1$
    $c_n^{(n)} = c_n^{(n-1)}$
    $c_{n-1}^{(n)} = c_{n-1}^{(n-1)} - (\alpha_0 - x_{n-1})c_n^{(n)}$
    $\bar{\alpha}_0 \overset{+}{=} \bar{c}_{n-1}^{(n-1)} \cdot c_n^{(n)}$
    $\bar{c}_n^{(n)} = \bar{c}_n^{(n-1)}$
    $\bar{c}_n^{(n)} = \bar{c}_{n-1}^{(n-1)} \cdot (\alpha_0 - x_{n-1})$
    $\bar{c}_{n-1}^{(n)} = \bar{c}_{n-1}^{(n-1)}$
    **return** $\bar{\alpha}, \bar{\beta}$
```

A.2.3 Vector-Jacobian Product in α and γ for NEvaluate

In the notation of Higham (1990), the three-term recurrence is:

$$p_{-1}(x) = 0; p_0(x) = 1; \quad p_{j+1}(x) = \theta_j(x - \beta_j)p_j(x) - \gamma_j p_{j-1}(x)$$

Their θ_j, β_j, and γ_j correspond to our $1/\gamma_{j+1}, \alpha_j/\gamma_{j+1}$, and γ_j/γ_{j+1}, respectively. Under their normalization $p_0(x) = 1$, suppose the solution of their algorithm is $\tilde{c}$. Then the solution under $p_0(x) = \gamma_0^{-1}$, as in the orthonormal polynomials, is $\gamma_0 \cdot \tilde{c}$. Following the template of the previous VJP derivations, here is NEvaluate written in indexed notation.

```
**procedure** NEvaluate$(x, \alpha, \gamma, c)$
    $u^{(n)} = [c_n, \ldots, c_n]$
    **for** $k \in [n-1, \ldots, 0]$ **do**
        $u^{(k)} = ((x - \alpha_k)/\gamma_{k+1}) \cdot u^{(k+1)} - \frac{\gamma_{k+1}}{\gamma_{k+2}} u^{(k+2)} + c_k$
    $\mu = u^{(0)}/\gamma_0$
    **return** $\mu$
```

The adjoints and u are derived as before.

$$\begin{aligned}
\bar{u}^{(0)} =& \bar{\mu}/\gamma_0 \\
u^{(0)} =& \mu \cdot \gamma_0 \\
\bar{\gamma}^{(0)} =& -\bar{\mu} \cdot u^{(0)}/\gamma_0^2 \\
\bar{u}^{(k+1)} \overset{+}{=}& \bar{u}^{(k)} \cdot (x - \alpha_k)/\gamma_{k+1} \\
\bar{u}^{(k+2)} =& \bar{u}^{(k)} \cdot (-\frac{\gamma_{k+1}}{\gamma_{k+2}}) \\
\bar{\alpha}_k =& \bar{u}^{(k)} \cdot (-u^{(k+1)}/\gamma_{k+1}) \\
\bar{\gamma}_{k+1} =& \bar{u}^{(k)} \cdot (-(x - \alpha_k)u^{(k+1)}/\gamma_{k+1}^2 - u^{(k+2)}/\gamma_{k+2}) \\
\bar{\gamma}_{k+2} =& \bar{u}^{(k)} \cdot (\gamma_{k+1}u^{(k+2)}/\gamma_{k+2}^2) \\
u^{(k+2)} =& \frac{\gamma_{k+2}}{\gamma_{k+1}}(-u^{(k)} + ((x - \alpha_k)/\gamma_{k+1}) \cdot u^{(k+1)} + c_k)
\end{aligned}$$

Computing these quantities in reverse order yields the VJP algorithm.

```
**procedure** $\overline{\text{NEvaluate}}(x, \alpha, \gamma, \mu, u^{(1)}, \bar{\mu}, \bar{u}^{(1)})$
    $\bar{\alpha} = \bar{\gamma} = [0, \ldots, 0]$
    $u^{(0)} = \mu \cdot \gamma_0$
    $\bar{u}^{(0)} = \bar{\mu}/\gamma_0$
    $\bar{\gamma}_0 = -\bar{\mu} \cdot u^{(0)}/\gamma_0^2$
    **for** $k \in [0, \ldots, n-1]$ **do**
        $u^{(k+2)} = \frac{\gamma_{k+2}}{\gamma_{k+1}}(-u^{(k)} + ((x - \alpha_k)/\gamma_{k+1}) \cdot u^{(k+1)} + c_k)$
        $\bar{\alpha}_k = -\bar{u}^{(k)} \cdot (u^{(k+1)}/\gamma_{k+1})$
        $\bar{\gamma}_{k+1} \overset{+}{=} -\bar{u}^{(k)} \cdot ((x - \alpha_k)u^{(k+1)}/\gamma_{k+1}^2 + u^{(k+2)}/\gamma_{k+2})$
        $\bar{\gamma}_{k+2} = \bar{u}^{(k)} \cdot (\gamma_{k+1}u^{(k+2)}/\gamma_{k+2}^2)$
        $\bar{u}^{(k+1)} \overset{+}{=} \bar{u}^{(k)} \cdot (x - \alpha_k)/\gamma_{k+1}$
        $\bar{u}^{(k+2)} = \bar{u}^{(k)} \cdot (-\frac{\gamma_{k+1}}{\gamma_{k+2}})$
    **return** $\bar{\alpha}, \bar{\beta}$
```

The final algorithm elides no-ops and indexing, and is given below.

```
procedure $\overline{\text{NEvaluate}}(x, \alpha, \gamma, \mu, v, \bar{\mu})$
    $\bar{v} = 0$
    $\bar{\alpha} = \bar{\beta} = [0, \ldots, 0]$
    $u = \mu \cdot \gamma_0$
    $\bar{u} = \bar{\mu}/\gamma_0$
    $\bar{\gamma}_0 = -\bar{\mu}^T u/\gamma_0^2$
    for $k \in [0, \ldots, n-1]$ do
        $\bar{\alpha}_k = -\bar{u}^T v/\gamma_{k+1}$
        $\bar{\gamma}_{k+1} \overset{+}{=} -\bar{u}^T(x - \alpha_k) \cdot v/\gamma_{k+1}^2$
        if $k < n-1$ then
            $w = \frac{\gamma_{k+2}}{\gamma_{k+1}}(-u + \frac{x-\alpha_k}{\gamma_{k+1}} \cdot v + c_k)$
            $\bar{\gamma}_{k+1} \overset{+}{=} -\bar{u}^T w/\gamma_{k+2}$
            $\bar{\gamma}_{k+2} = \bar{u}^T w \cdot \gamma_{k+1}/\gamma_{k+2}^2$
            $u, v = v, w$
            $\tau = \bar{u}$
            $\bar{u} = \bar{v} + \bar{u} \cdot (x - \alpha_k)/\gamma_{k+1}$
            $\bar{v} = -\tau \cdot \frac{\gamma_{k+1}}{\gamma_{k+2}}$
    return $\bar{\alpha}, \bar{\beta}$
```

A.3 Experiment Details

Op Benchmarks (Figure 5). The comparison implementation is written in JAX (Bradbury et al., 2018), with for loops specially expressed as structured control flow. Python's `timeit` is used to perform timing, taking the best of 4 runs, each having 2 repetitions. A batch size of 32 is used.

Learnable JPEG. Adagrad is used as the optimizer. Learning rates of 0.2 and 2.0 performed best with and without AnyPT, respectively. The standard train/test split of CLIC is used. The batch size is 1.

Mop Experiment 1. $n = 8$, $B = 4096$, and $\delta = 0$. The initialization is $\alpha_i = 0$ and $\beta_i = 1/2$. β_0, β_n, and α_n are not in J_n, and so are not measured as part of the relative error. The error of α is not plotted because it follows the same pattern.

Mop Experiment 2. $n = 8$, $B = 8192$, and $\delta = 0$. α was initialized with a Glorot normal and β by random $U(0, 1)$. The optimizer is RMSprop with learning rate 0.05 and gradient clipping.

Mop Experiment 3. $n = 8$, $B = 4096$. Various optimizers (including SGD and RMSprop) were attempted, and none changed the (lack of) results.