Accelerating Automatic Differentiation of Direct Form Digital Filters

Chin-Yun Yu György Fazekas
Centre for Digital Music, Queen Mary University of London, London, UK

Overview and Contributions

- A differentiable implementation of every direct form digital filter for PyTorch, utilising parallel algorithms to accelerate both the forward pass and gradient computation.
- Analytical gradients are derived for any reverse-mode automatic differentiation frameworks.
- Over 1000x speedup compared to naive implementations.

Problem Statement: Recursive Filters Are Slow For AD

An M^{th}-order time-invariant linear system has the following direct form (DF) difference equation:

$$\begin{aligned} y(n) = {} & b_0 x(n) + b_1 x(n-1) + \ldots + b_M x(n-M) \\ & - a_1 y(n-1) - a_2 y(n-2) - \ldots - a_M y(n-M) \end{aligned} \tag{1}$$

where b_i, a_i are the feed-forward and feedback coefficients, $x(n)$ and $y(n)$ are the input and output signals. The definition of $y(n)$ is **recursive**.

An implementation of $\mathbf{v}(n+1) = \mathbf{A}\mathbf{v}(n) + \mathbf{z}(n)$ in PyTorch:

```
v = []
vn = torch.zeros(M)
for n in range(N):
    vn = torch.addmv(z[n], A, vn)
    v.append(vn)
v = torch.stack(v)
```

- Runtime: $N\times$ (`torch.*` overhead + execution time)
- `torch.compile`: Bad at optimising for-loops
- Dedicated recursion kernel has a runtime of just `torch.*` overhead + execution time
  - Support ✓: JAX and TensorFlow
  - WIP ×: PyTorch [1]

Previous Work: Frequency-Sampling

Filtering in the frequency domain only needs fixed number of call to `torch.fft.*`:

$$H(e^{j\omega}) = \frac{b_0 + b_1 e^{-j\omega} + \ldots + b_M e^{-jM\omega}}{1 + a_1 e^{-j\omega} + \ldots + a_M e^{-jM\omega}}. \tag{2}$$

Issues commonly observed:

- Frequency-domain sampling ⟶ time-domain aliasing
- $Y(j\omega) = H(j\omega)X(j\omega)$ results in circular convolution artefacts
- Less efficient for low-order filters

Proposed Method: State-Space and Custom Extension

We implement filtering as a custom C++/CUDA extension, avoiding the accumulated overhead proportional to N. This has been used before [2, 3] but

- Only applies to a limited set of filters (DF-I/II and all-pole)
- No parallelisation across the time dimension
- Does not include gradients to the initial state

We rewrite Eq. (1) into state-space form:

$$\mathbf{v}(n+1) = \mathbf{A}\mathbf{v}(n) + \mathbf{B}x(n) \tag{3}$$

$$y(n) = \mathbf{C}^\top \mathbf{v}(n) + Dx(n) \tag{4}$$

$$\mathbf{A} = \begin{bmatrix} -\mathbf{a}^\top \\ \mathbf{I}\ \mathbf{0} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} b_1 - a_1 b_0 \\ b_2 - a_2 b_0 \\ \vdots \\ b_M - a_M b_0 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, D = b_0. \tag{5}$$

Benefits. State-space form covers transposed direct forms (TDF) (more numerically robust and common) with $\mathbf{A} \mapsto \mathbf{A}^\top$, $\mathbf{C} \mapsto \mathbf{B}$, $\mathbf{B} \mapsto \mathbf{C}$. Eq. (3) can be **parallelised** by computing the following associative scan [4]:

$$(\mathbf{0}, \mathbf{v}(n)) = (\mathbf{0}, \mathbf{v}(0)) \oplus (\mathbf{A}, \mathbf{B}\mathbf{x}(0)) \oplus \ldots \oplus (\mathbf{A}, \mathbf{B}\mathbf{x}(n-1)), \tag{6}$$

where $(\mathbf{A}, \mathbf{z}) \oplus (\mathbf{A}', \mathbf{z}') \mapsto (\mathbf{A}'\mathbf{A}, \mathbf{A}'\mathbf{z} + \mathbf{z}')$.

Gradient Backpropagation

Let $\mathcal{L} = f(y(0), y(1), \ldots)$ be the scalar we want to minimise. Given the output gradients $\frac{\partial \mathcal{L}}{\partial y(n)}$, the gradients to the variables in the filter can be computed as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{C}} = \sum_{n \geq 0} \frac{\partial \mathcal{L}}{\partial y(n)} \mathbf{v}(n)^\top, \quad \frac{\partial \mathcal{L}}{\partial D} = \sum_{n \geq 0} \frac{\partial \mathcal{L}}{\partial y(n)} x(n), \tag{7}$$

$$\mathbf{u}(n) = \mathbf{A}^\top \mathbf{u}(n+1) + \mathbf{C} \frac{\partial \mathcal{L}}{\partial y(n+1)}, \tag{8}$$

$$\frac{\partial \mathcal{L}}{\partial x(n)} = \mathbf{B}\mathbf{u}(n) + D \frac{\partial \mathcal{L}}{\partial y(n)}, \tag{9}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{v}(0)} = \mathbf{u}(-1)^\top, \frac{\partial \mathcal{L}}{\partial \mathbf{B}} = \sum_{n \geq 0} \mathbf{u}(n)^\top x(n), \frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \sum_{n \geq 0} \mathbf{v}(n)\mathbf{u}(n)^\top. \tag{10}$$

Key Insight. Eq. (8) and (9) form a TDF filter in state-space form; similarly, the backpropagation of a TDF filter is a DF filter. Moreover, we can use the same parallel scan method to accelerate Eq. (8).

Experimental Results

We benchmarked the recursion Eq. (3) with $M = 2$ and sequence length ranging from 2^{14} to 2^{20} in PyTorch 2.8 on an Intel i7-7700K and an NVIDIA RTX 5060 Ti.

- **Naive**: Simple for-loop implementation.
- **RTF**: Frequency-domain filtering using Eq. (2).
- **Unrolled**: Parallel scan using `torch.matmul` [5].
- **EXT**: Our custom C++/CUDA extension.
- **Diag-EXT**: Diagonalise $\mathbf{A} = \mathbf{V}\text{diag}(\lambda)\mathbf{V}^{-1}$ and perform scan in the eigenbasis [6].

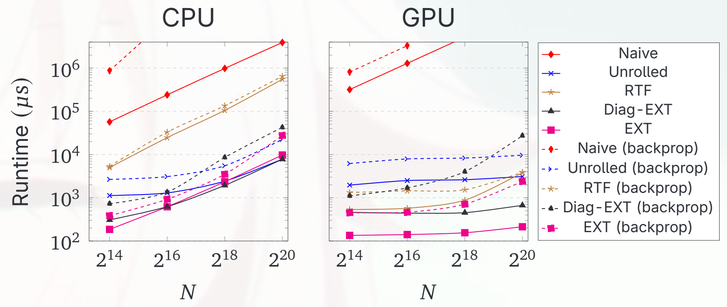

Figure 1. Runtime of the recursion and its backpropagation with various signal lengths N and implementations.

- The naive implementation is remarkably slower and impractical.
- EXT consistently spends the least time computing gradients in most configurations.
- RTF is comparable to Diag-EXT in smaller N on the GPU. However, it is only slightly faster than Naive on the CPU.

Conclusions and Next Steps

- Our custom extension for direct form filters significantly reduces the runtime of both the forward pass and backpropagation compared to existing methods.
- We plan to extend the derivations to parameter-varying cases, a broader class of state-space models, and forward-mode differentiation. Analysing the numerical robustness of the analytical gradients is another interesting direction.

[1] Wu et al. Control Flow Operators in PyTorch. 2025.
[2] Forgione and Piga. dynoNet: A neural network architecture for learning dynamical systems. en. *International Journal of Adaptive Control and Signal Processing* 35.4 (2021). ISSN: 1099-1115.
[3] Yu and Fazekas. GOLF: A Singing Voice Synthesiser with Glottal Flow Wavetables and LPC Filters. *TISMIR* (Dec. 2024). DOI: 10.5334/tismir.210.
[4] Blelloch. Prefix Sums and Their Applications. Tech. rep. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University, Nov. 1990.
[5] Yu. Block-based Fast Differentiable IIR in PyTorch. https://iamycy.github.io/posts/2025/06/28/unroll-ssm/. June 2025.
[6] Yu et al. DiffVox: A Differentiable Model for Capturing and Analysing Vocal Effects Distributions. 2025.