AUTO-DSP: LEARNING TO OPTIMIZE ACOUSTIC ECHO CANCELLERS

Jonah Casebeer[♯] *Nicholas J. Bryan*[♭] *Paris Smaragdis*[♯♭]

[♯] University of Illinois at Urbana-Champaign, [♭] Adobe Research

ABSTRACT

Adaptive filtering algorithms are commonplace in signal processing and have wide-ranging applications from single-channel denoising to multi-channel acoustic echo cancellation and adaptive beamforming. Such algorithms typically operate via specialized online, iterative optimization methods and have achieved tremendous success, but require expert knowledge, are slow to develop, and are difficult to customize. In our work, we present a new method to automatically learn adaptive filtering update rules directly from data. To do so, we frame adaptive filtering as a differentiable operator and train a learned optimizer to output a gradient descent-based update rule from data via backpropagation through time. We demonstrate our general approach on an acoustic echo cancellation task (single-talk with noise) and show that we can learn high-performing adaptive filters for a variety of common linear and non-linear multidelayed block frequency domain filter architectures. We also find that our learned update rules exhibit fast convergence, can optimize in the presence of nonlinearities, and are robust to acoustic scene changes despite never encountering any during training.

Index Terms— adaptive filtering, adaptive optimization, learning to learn, meta-learning, acoustic echo cancellation

1. INTRODUCTION

Adaptive filtering algorithms are ubiquitous and include single- and multi-channel denoising, dereverberation, echo cancellation, system identification, noise cancellation, feedback cancellation, and more. Such algorithms typically operate by applying an online, iterative optimization method, such as least mean square filtering (LMS), normalized LMS (NLMS), or recursive least-squares (RLS), to solve an optimization problem over time (e.g. estimating a time-varying transfer function for echo cancellation) [1, 2, 3, 4, 5]. The derivation and implementation of these methods requires careful attention, customization, expertise, and/or a laborious tuning process (e.g. tuning per hardware device) per application.

One of the most prevalent adaptive filtering applications is acoustic echo cancellation (AEC). In this case, an adaptive filter is used to remove echo within a telecommunication system. Customized AEC adaptive filters take many forms including algorithms based on sparsity [6], adaptive normalization [7], and adaptive learning-rates [8], as well as data-driven approaches for selecting learning rates automatically [9, 10] and based on a meta-step-size [11, 12]. More recently, deep learning techniques have been used as AEC sub-components including learned residual echo suppressors [13, 14, 15], double-talk detectors [16], and nonlinear distortions blocks [17, 18, 19, 20, 21]. These approaches, however, commonly do not use neural network modules that adapt at test time, do not have matching training and testing steps, and/or do not directly learn adaptive filter update rules end-to-end.

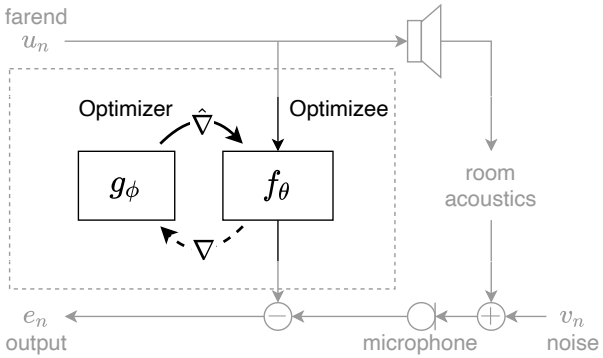

Figure 1: A learned optimizer, g_ϕ, updates the adaptive filter f_θ in an online fashion. The optimizer parameters ϕ are meta-learned directly from data and do not use any external labels. The dashed curved line denotes adaptation during training, but not inference.

In the machine learning literature, there have been exciting developments in meta-learning, automatic machine learning, and *learning how to learn* methods. Methods include using one neural network to control the weights of another [22, 23, 24], pre-training deep networks that quickly fine-tune [25], and learning offline stochastic gradient descent update rules via neural networks [26]. The latter work is most relevant, shows how offline *learned* optimizers can outperform their hand-designed counterparts for certain neural network architectures, and inspired further work in optimizer architectures [27] and training [28, 29]. We believe this work is significant and, while not previously explored, offer tremendous potential for the field of signal processing and adaptive filtering.

In this work, we formulate the development of adaptive filtering algorithms as a meta-learning problem and learn to optimize adaptive filters. To do so, we frame adaptive filtering itself as a differentiable operator and train a learned optimizer from data, without external labels, using truncated backpropagation through time. By doing so, we create an automatic digital signal processing (Auto-DSP) approach that learns optimal adaptive filters without any need for hand-derived gradients and can be used for a variety of applications. To demonstrate our approach, we learn to optimize an AEC task as shown in Fig. 1 for a single-talk in noise scenario. We use the Microsoft AEC Challenge dataset [30] to learn update rules for a variety of common linear and nonlinear multidelayed block frequency domain filters (MDF) [31]. We compare our results to hand-engineered, grid-search-tuned block NLMS and RMSprop [32] optimizers, as well as the open-source Speex AEC [8, 33]. We find our learned optimizers outperform all of these methods for our tasks, exhibit fast convergence, require little-to-no manual design intervention for training, can optimize in the presence of nonlinearities, and converge quickly and robustly to unseen acoustic scene changes.

Code & demo: *https://jmcasebeer.github.io/projects/auto-dsp/*

2. AUTO-DSP OPTIMIZATION

To learn an adaptive filter update rule from data, we first define a learned optimizer, $g_\phi(\cdot)$, as a function doing the optimizing and an optimizee, $f_\theta(\cdot)$, as a differentiable adaptive filter to be optimized, and $J(\cdot)$ as the optimizee loss. Second, we set the optimizer to be a neural network that accepts as input raw optimizee gradients $\nabla f_{\theta_n}(\cdot)$ and a state vector $\mathbf{h}$ and outputs a learned gradient descent update rule,

$$\theta_{n+1} = \theta_n + g_\phi(\nabla f_{\theta_n}(\cdot), \mathbf{h}), \tag{1}$$

where θ are optimizee parameters and ϕ are optimizer parameters. The state, $\mathbf{h}$, is also updated. Note, the raw gradient inputs used here are provided by automatic differentiation and are not implemented manually. Third, we assign the optimizer and optimizee an objective function or loss and use truncated backpropagation through time (BPTT) [34] to fit the optimizer parameters to data.

2.1. Optimizee architecture & loss

The optimizee, or adaptive filtering being optimized, provides the architecture used for filtering signals. It is defined by filter parameters θ, a filtering architecture $f_\theta(\cdot)$, and an optimizee loss function. For illustrative purposes, we can consider a basic time-domain adaptive filter optimizee. In this case, the optimizee parameters θ correspond to transversal finite impulse response (FIR) filter coefficients $\theta = \{\hat{\mathbf{w}}_n \in \mathbb{R}^N\}$, the optimizee architecture corresponds to the inner product between an input vector $\mathbf{u}_n \in \mathbb{R}^N$ and the filter coefficients $f_\theta(\mathbf{u}_n) = y_n = \hat{\mathbf{w}}_n^{\mathsf{H}}\mathbf{u}_n$, and the optimizee loss corresponds to a mean squared error objective, $J(y_n, d_n) = \frac{1}{N}\sum_n^N |y_n - d_n|^2$, where $d_n \in \mathbb{R}$ is the desired, known response. In this case, we can reduce the optimizee update (1) to

$$\hat{\mathbf{w}}_{n+1} = \hat{\mathbf{w}}_n - g_\phi(\mathbf{u}_n \cdot (\hat{\mathbf{w}}_n^{\mathsf{H}}\mathbf{u}_n - d_n)^*), \tag{2}$$

where * denotes complex conjugation and $^{\mathsf{H}}$ denotes Hermitian transposition.

While we manually derive the gradient vector $\nabla f_{\theta_n}(\cdot)$ here, in practice gradients are computed via automatic differentiation. Because of this, we can use more advanced optimizees such as lattice FIR filters, block frequency-domain filters [1, 4], multidelayed block frequency domain filters [31], or non-linear variants such as (polynomial) Volterra filters [35], or Hammerstein filters [36] with ease. We can also use alternative differentiable optimizee losses such as negative log-likelihood or mutual information.

2.2. Optimizer architecture & loss

The optimizer $g_\phi(\cdot)$, or function doing the optimizing, is parameterized by ϕ and used to adapt the optimizee parameters θ over time. The optimizer accepts as input raw optimizee gradients and outputs an optimized, learned update rule. For a basic time-domain adaptive filter optimizee, we can define the optimizer architecture as a single step-size μ and reduce (2) to

$$\hat{\mathbf{w}}_{n+1} = \hat{\mathbf{w}}_n - \mu \cdot \mathbf{u}_n(\hat{\mathbf{w}}_n^{\mathsf{H}}\mathbf{u}_n - d_n)^*, \tag{3}$$

or the well known LMS algorithm. For a more powerful optimizer, however, we can define the optimizer $g_\phi(\nabla J(\cdot))$ to be a neural network module such as a recurrent neural network (RNN), convolutional neural network (CNN), fully connected network, or similar. The design of the optimizer, however, has tremendous implications on computational complexity of the approach. Thus, we make the optimizer agnostic of the optimizee layout by applying the optimizer independently to each element of the optimizee parameters θ. That is, the optimizer update is applied element-wise to each optimizee parameter. This allows us to efficiently vectorize our update rules and perform weight sharing in the optimizer, while maintaining independent state dynamics per optimizee parameter.

In terms of the optimizer objective, we set it to be the sum of a collection of optimizee losses averaged across a dataset, which requires no additional labels. However, this can be modified to favor different optimization dynamics or design constraints. For example, we could use a weighted sum where earlier (or later) losses are weighted more. This would enable training an optimizer specializing in early (or late) convergence. Other schemes could produce optimizers that favor small updates, sparse updates, etc.

Algorithm 1 Meta-learning training algorithm.

Precondition: $J(\cdot)$, METAOPT, N_i, N_o
function INNERLOOP($\phi, \mathbf{h}, \theta, \mathbf{u}, \mathbf{d}$)
  $\mathcal{L} \leftarrow 0$
  for $n_i \leftarrow 0$ to N_i **do**
    $y_{n_i} \leftarrow f_{\theta_{n_i}}(\mathbf{u}_{n_i})$
    $\mathcal{L} \leftarrow \mathcal{L} + J(y_{n_i}, \mathbf{d}_{n_i})$
    $\hat{\nabla}, \mathbf{h} \leftarrow g_\phi(\nabla J(y_{n_i}, \mathbf{d}_{n_i}), \mathbf{h})$
    $\theta_{n_i+1} \leftarrow \theta_{n_i} - \hat{\nabla}$
  end for
  return $\mathcal{L}, \theta_{N_i}, \mathbf{h}$
end function
function OUTERLOOP($\phi, \mathbf{h}, \theta, \mathbf{u}, \mathbf{d}$)
  for $n_o \leftarrow 0$ to N_o **do**
    $\mathcal{L}, \theta_{n_o+1}, \mathbf{h} \leftarrow$ INNERLOOP($\phi_{n_o}, \mathbf{h}, \theta_{n_o}, \mathbf{u}_{n_o}, \mathbf{d}_{n_o}$)
    $\phi_{n_o+1} \leftarrow$ METAOPT($\phi_{n_o}, \mathcal{L}$)
  end for
  return ϕ_{N_o}
end function

2.3. Learning the optimizer

To train our optimizers, we follow the procedure outlined in Algorithm 1. The procedure consists of two basic steps: an inner loop and an outer loop. The inner loop process runs an adaptive filter optimizee for a finite number of time steps, N_i, updates the optimizee parameters as it goes, and accumulates the optimizee loss for a fixed optimizer state using BPTT. The outer loop invokes the inner loop, uses a standard deep learning optimizer denoted as a meta optimizer (METAOPT) to update the learned optimizer module, and repeats for a finite number of outer loop steps, N_o. In practice, the outer loop is vectorized and runs across a randomized collection of signals (i.e. batches) continuously sampled from data until the optimizer loss convergences. This procedure allows us to train our optimizer on long sequences and helps minimize exploding gradient issues. After training, the learned optimizers are used like conventional optimizers and do not use the inner/outer scheme.

3. EXPERIMENTAL SETUP

3.1. Optimizee configuration

To demonstrate our approach, we consider the adaptive filtering task of acoustic echo cancellation or interference cancellation. For our AEC optimizee architecture, we use an MDF filter with an optional

parametric nonlinearity. The optimizee parameters θ include frequency domain filter coefficients and a small set of nonlinear coefficients. The filter coefficients are partitioned into multiple delayed blocks and used within the framework of overlap-save short-time Fourier transform processing [37]. MDF filters are commonly used for AEC and leverage the benefits of both frequency-domain adaptation [4] and low latency. For our optimizee loss, which implicitly defines the optimizer loss, we use the mean squared error.

In more detail, our MDF filter consists of frequency domain filter coefficients $\mathbf{W} \in \mathbb{C}^{M \times N}$, where M is the number of delayed blocks, N is the fast Fourier transform (FFT) size, $P = M \cdot N/2$ is the number of filter parameters, and L is the filter length in samples. The filter matrix is applied to the delayed frequency domain near-end inputs $\mathbf{U} \in \mathbb{C}^{M \times N}$ to yield a filtered output via $y_n = \text{last } N/2 \text{ terms of} \{\text{FFT}^{-1}((\mathbf{W} \odot \mathbf{U})^\top \mathbf{1}_N)\}$, where $^\top$ is a matrix transpose, $\odot$ is the hadamard product, and $\mathbf{1}_N$ is an $N \times 1$ matrix of ones. To construct $\mathbf{U}$, we buffer the time-domain near-end signal to length N with time overlap R, forming $\mathbf{u}_{\tilde{n}} \in \mathbb{R}^N$, shift $\mathbf{U}_m = \mathbf{U}_{m+1}$ for $m = 1, 2, \cdots, M-1$, and assign $\mathbf{U}_M = \text{FFT}(\mathbf{u}_{\tilde{n}})$. Finally, we antialias $\mathbf{W}$ after each update so that each block has $N/2$ nonzero time-domain parameters. For our nonlinearity extension, we preprocess each element u_n of the far-end reference signal through a parametric sigmoid

$$\gamma(u_n) = \alpha_4 \left(\frac{2}{1 + \exp(\alpha_2 \hat{u}_n + \alpha_3 \hat{u}_n^2)} - 1 \right) \tag{4}$$

where $\hat{u}_n = (u_n \cdot \alpha_1)/(\sqrt{|u_n|^2 + |\alpha_1|^2})$ and $\alpha_i \forall i$ are adapted.

3.2. Optimizer configuration

We implement a complex-valued gated recurrent unit (GRU) optimizer architecture composed of a complex linear layer with output size H, two weight-tied complex GRU layers, and a complex output linear layer. The GRU layers share a single H dimensional hidden state $\mathbf{h} \in \mathbb{C}^H$. All layers are followed by complex rectified linear units (ReLU). The GRU layers use complex-valued hyperbolic tangent (tanh) and sigmoid activation functions. We use the initialization scheme proposed in [38] and test two optimizer sizes: $H = 24$ and $H = 48$ with 3.6k and 14k complex-valued parameters, respectively. To train this model, we use the mean-squared error optimizee loss averaged across a batch of optimizees.

As a preprocessing step to our GRU-based optimizer, we modify the feature extraction in [26] to operate in the complex domain. Specifically, we input complex optimizee gradients ∇ and then limit the dynamic range by clipping and compressing the gradient magnitudes via:

$$\tilde{\nabla} = \frac{\log(\max(e^{-p}, \min(|\nabla|, e^p))) + p}{p} e^{j \angle \nabla}, \tag{5}$$

where p is a hyperparameter that controls the clipping and e is the exponential function. The purpose of this is to leave the phase of the gradient unchanged. We set $p = 10$ in all experiments.

3.3. Evaluation metrics

To measure the average performance of our learned adaptive filters and compare to known baselines, we use the average echo return loss enhancement (ERLE), which is defined as $10 \log(\sum_n |d_n|^2 / \sum_n |y_n - d_n|^2)$. To understand convergence speed and provide further qualitative analysis, we also use the segmental ERLE, or the ERLE computed on short windows of size N.

3.4. Dataset

We train and test our optimizers using the synthetic portion of the Microsoft AEC Challenge dataset [30]. This dataset includes far-end noise, near-end noise, and far-end nonlinearities. We preprocess the data by resampling to 8kHz and remove near-end speech from all near-end recordings for focus and leave learning double-talk robust optimizers for future work. Note that the dataset is composed of 80% nonlinear scenes and 20% linear scenes.

3.5. Training

For training, we follow Algorithm 1 together with a standard training, validation, and testing setup. We use Adam (lr$= 10^{-4}$) with gradient clipping as our meta-optimizer, and halve the learning rate if ERLE performance on the validation set does not improve for 10 consecutive epochs and cease training after 25 epochs with no ERLE improvement. We define an epoch to be 200 batches of optimization runs where each optimization run consists of an initial filter state, a 10 second far-end signal and a 10 second near-end signal. We alternate between inner and outer updates as defined in Algorithm 1 and set $N_i = 10$. N_o is set to be the number of inner loop updates that will fit within a 10 second sequence. We found that optimizers did not converge well when the value of N_i was set much higher than 20 or lower than 5. Our implementation is written using the JAX framework [39]. Training an optimizer takes one to ten days depending on optimizer/optimizee complexity on two RTX 2080 TI GPUs.

4. RESULTS & DISCUSSION

We evaluate our learned optimizers across multiple optimizer configurations and optimizee configurations as well as linear and nonlinear scenes and compare against standard hand-derived update rules. Our baselines consist of step-size tuned block frequency-domain NLMS optimizer with smoothing constant ($\beta = .9, .99$), a step-size tuned frequency-domain RMSprop optimizer, and the Speex AEC. While Speex is representative of a well-engineered hand-tuned optimizer it was not optimized for this dataset whereas the other optimizers are. Baseline results are shown in the first section of Table 1. We denote the learned optimizer hidden size by H, the number of filter parameters as P, the FFT size as N, the number of MDF blocks as M, the overlap between blocks as R, whether the optimizee has a nonlinear component with γ, and provide both the average μ and standard deviation σ ERLE. All baseline and learned optimizees have an effective filter length of $L = 2048$ taps.

4.1. Feature extraction

We evaluate our proposed feature extraction by training optimizers ($H = 24, P = 2048, N = 2P, M = 1, R = 1/2$, and no γ) with and without this preprocessing and display the results in the second portion of Table 1. As shown, our proposed features improve performance by $\approx$ 2dB and $\approx$ 3dB dB in nonlinear and linear scenes, respectively. By inspection, we found that that the distribution of gradient magnitudes was heavily skewed and we hypothesize that clipping and compressing gradient magnitudes acts as a form of whitening that approximately normalizes the distribution. Given this result, we use the proposed feature extraction for all further experiments.

Optimizer	Optimizee				ERLE (dB) Nonlinear		ERLE (dB) Linear	
	M	N	R	γ	μ	σ	μ	σ
NLMS ($\beta = .9$)	1	4096	7/8	✗	4.40	11.82	9.57	6.01
NLMS ($\beta = .99$)	1	4096	7/8	✗	4.07	4.54	5.20	2.57
RMSprop	1	4096	7/8	✗	5.58	2.94	7.71	2.56
Speex	4	1024	1/2	✗	9.79	4.56	8.55	3.51
Speex	8	512	1/2	✗	9.83	4.50	8.72	3.55
GRU ($H = 24, \nabla$)	1	4096	1/2	✗	3.24	1.78	4.68	1.85
GRU ($H = 24, \tilde{\nabla}$)	1	4096	1/2	✗	5.69	2.86	7.78	2.13
GRU ($H = 48, \tilde{\nabla}$)	1	4096	1/2	✗	5.87	2.93	7.95	2.15
GRU ($H = 48, \tilde{\nabla}$)	1	4096	3/4	✗	8.26	4.03	11.22	2.93
GRU ($H = 48, \tilde{\nabla}$)	1	4096	7/8	✗	10.40	5.18	14.21	4.29
GRU ($H = 48, \tilde{\nabla}$)	4	1024	1/2	✗	8.11	4.40	9.46	3.26
GRU ($H = 48, \tilde{\nabla}$)	4	1640	3/4	✗	10.20	5.15	13.62	3.87
GRU ($H = 48, \tilde{\nabla}$)	8	512	1/2	✗	8.45	4.58	9.54	3.32
GRU ($H = 48, \tilde{\nabla}$)	8	912	3/4	✗	10.75	5.46	13.93	3.94
GRU ($H = 48, \tilde{\nabla}$)	1	4096	7/8	✓	10.53	4.04	13.45	3.55
GRU ($H = 48, \tilde{\nabla}$)	4	1640	3/4	✓	9.17	3.73	11.66	4.02
GRU ($H = 48, \tilde{\nabla}$)	8	912	3/4	✓	10.17	4.12	11.61	4.07

Table 1: Optimizer comparison using the ERLE metric. The optimizee column shows the number of blocks M, the FFT size N, the proportion of block overlap R, and the optimizee nonlinearity extension γ. All optimizees have a filter length of $L = 2048$.

4.2. Optimizer capacity & computational complexity

Next, we increase the hidden state size H to 48 and compare different proportions of overlap in the third portion of Table 1. At $R = 1/2$ and $3/4$, the learned optimizer outperforms NLMS and RMSprop. When R is increased to $7/8$, performance improves by multiple dB and the learned optimizer outperforms all baselines with a slightly higher standard deviation than Speex. For this configuration, the optimizee has $P = 2048$ and uses 240 mega-MACS per update. On an i9-9820X CPU the real-time factor is .36 using one thread and .13 using multiple threads. This is remarkable, given the engineering expertise and effort distilled into our baselines.

4.3. Learned optimizer dynamics

We evaluate how our learned optimizers respond to an abrupt change in the echo path using the final setup from Section 4.2. The learned optimizers were trained on static scenes with a duration of 10 seconds. However, test scenes here are twice as long and formed by concatenating two test set files. The Block NLMS and RMSprop baselines were tuned on static scenes to match the learned optimizer setup. In Fig. 2, we compare ERLE across time. To match the number of updates, we run Speex with 4 blocks.

In both linear and nonlinear scenes the learned optimizer converges rapidly and achieves a steady-state in $\approx$ 2 seconds. In linear scenes, block NLMS is competitive and reaches a steady-state ERLE at $\approx$ 6 seconds. However, after the scene change at 10 seconds, the learned optimizer is the only optimizer that recovers its full performance. In nonlinear scenes, Speex displays strong steady-state performance and overtakes the learned optimizer. Though, after the scene change, the learned optimizer recovers and is not surpassed. On average, our optimizer outperforms all baselines for both steady-state and early convergence and demonstrates it can generalize to novel and challenging environments.

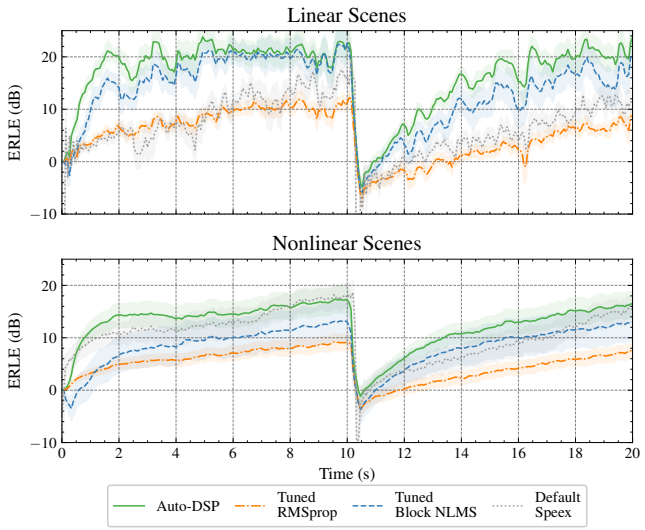

Figure 2: ERLE (dB) performance across time in linear (top) and nonlinear (bottom) scenes. The echo path changes ten seconds into the scene. Lines represent mean performance and shaded regions represent $\pm\frac{1}{2}$ a standard deviation. Our optimizer converges faster than other optimizers and quickly adapts despite being trained on shorter scenes and never encountering scene changes in training.

4.4. Optimizee architecture

In our final set of experiments, we learn optimizers for a variety of optimizee architectures. That is, we construct MDF adaptive filters with 4 and 8 blocks (instead of one block) and also experiment with incorporating a parametric nonlinear distortion block and adjusting the overlap R. Results can be found in the last two sections of Table 1. Note that these optimizees may have $P > L$ parameters. For all adaptive filters, we use identical optimizer hyperparameters, training scheme, and architecture. In effect, no manual architecture design intervention is required for any of our learned optimizers.

First, we find that our learned optimizers successfully scale to more complex linear adaptive filter architectures. Second, we find that we can learn to optimize nonlinear variants of MDF, and generally outperform their hand-tuned counterparts in nonlinear scenes. This suggests we can learn optimal update rules per filtering architecture to fit design trade-offs like latency versus computational complexity without needing to hand-derive anything and has the potential to fundamentally change how we develop adaptive filters.

5. CONCLUSION

In this work, we formulate the optimization of adaptive filters as a meta-learning problem and successfully replace hand-derived update rules with learned update rules. We call this method Auto-DSP and apply it in acoustic echo cancellation where we learn optimization rules in a data-driven fashion, without any external labels. Using an identical optimizer configuration, we experiment with learning update rules for multidelay block frequency domain filters both with and without parametric nonlinearities. We evaluate performance across scenes with nearend noise and far-end distortion, and find we can outperform tuned block NLMS and RMSprop optimizers and a popular open source filter (Speex). In all, we believe learning adaptive filter update rules from data is an exciting new signal processing methodology and has tremendous potential.

6. REFERENCES

[1] B. Widrow and S. D. Stearns, *Adaptive Signal Processing*. Prentice-Hall, 1985.

[2] E. Hänsler and G. Schmidt, *Acoustic echo and noise control: a practical approach*. John Wiley & Sons, 2005.

[3] J. Benesty, T. Gänsler, D. R. Morgan, S. L. Gay, and M. M. Sondhi, *Advances in Network and Acoustic Echo Cancellation*. Springer, 2001.

[4] S. S. Haykin, *Adaptive filter theory*. Pearson, 2008.

[5] J. A. Apolinário, J. A. Apolinário, and R. Rautmann, *QRD-RLS adaptive filtering*. Springer, 2009.

[6] S. L. Gay, "An efficient, fast converging adaptive filter for network echo cancellation," in *IEEE Asilomar Conference on Signals, Systems and Computers*, 1998.

[7] D. L. Duttweiler, "Proportionate normalized least-mean-squares adaptation in echo cancelers," *IEEE Transactions on Speech and Audio Processing*, 2000.

[8] J.-M. Valin, "On adjusting the learning rate in frequency domain echo cancellation with double-talk," *IEEE Transactions on Audio, Speech, and Language Processing*, 2007.

[9] W. Dabney and A. Barto, "Adaptive step-size for online temporal difference learning," in *AAAI*, 2012.

[10] A. R. Mahmood, R. S. Sutton, T. Degris, and P. M. Pilarski, "Tuning-free step-size adaptation," in *IEEE ICASSP*, 2012.

[11] R. S. Sutton, "Adapting bias by gradient descent: An incremental version of delta-bar-delta," in *AAAI*, 1992.

[12] N. N. Schraudolph, "Local gain adaptation in stochastic gradient descent," in *ICANN*, 1999.

[13] H. Zhang, K. Tan, and D. Wang, "Deep learning for joint acoustic echo and noise cancellation with nonlinear distortions." in *INTERSPEECH*, 2019.

[14] A. Fazel, M. El-Khamy, and J. Lee, "CAD-AEC: Context-aware deep acoustic echo cancellation," in *IEEE ICASSP*, 2020.

[15] J.-M. Valin, S. Tenneti, K. Helwani, U. Isik, and A. Krishnaswamy, "Low-complexity, real-time joint neural echo control and speech enhancement based on PercepNet," *arXiv preprint arXiv:2102.05245*, 2021.

[16] L. Ma, H. Huang, P. Zhao, and T. Su, "Acoustic echo cancellation by combining adaptive digital filter and recurrent neural network," *arXiv preprint arXiv:2005.09237*, 2020.

[17] A. N. Birkett and R. A. Goubran, "Acoustic echo cancellation using NLMS-neural network structures," in *IEEE ICASSP*, 1995.

[18] A. B. Rabaa and R. Tourki, "Acoustic echo cancellation based on a recurrent neural network and a fast affine projection algorithm," in *IEEE IES*, 1998.

[19] S. Zhang and W. X. Zheng, "Recursive adaptive sparse exponential functional link neural network for nonlinear AEC in impulsive noise environment," *IEEE Transactions on Neural Networks and Learning Systems*, 2017.

[20] M. M. Halimeh, C. Huemmer, and W. Kellermann, "A neural network-based nonlinear acoustic echo canceller," *IEEE SPL*, 2019.

[21] J. Malek and Z. Koldovskỳ, "Hammerstein model-based nonlinear echo cancelation using a cascade of neural network and adaptive linear filter," in *IEEE IWAENC*, 2016.

[22] J. Schmidhuber, "Learning to control fast-weight memories: An alternative to dynamic recurrent networks," *Neural Computation*, 1992.

[23] I. Bello, B. Zoph, V. Vasudevan, and Q. V. Le, "Neural optimizer search with reinforcement learning," in *ICML*, 2017.

[24] D. Ha, A. Dai, and Q. V. Le, "Hypernetworks," *arXiv preprint arXiv:1609.09106*, 2016.

[25] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *ICML*, 2017.

[26] M. Andrychowicz, M. Denil, S. G. Colmenarejo, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. de Freitas, "Learning to learn by gradient descent by gradient descent," in *NeurIPS*, 2016.

[27] O. Wichrowska, N. Maheswaranathan, M. W. Hoffman, S. G. Colmenarejo, M. Denil, N. Freitas, and J. Sohl-Dickstein, "Learned optimizers that scale and generalize," in *ICML*, 2017.

[28] L. Metz, N. Maheswaranathan, J. Nixon, D. Freeman, and J. Sohl-Dickstein, "Understanding and correcting pathologies in the training of learned optimizers," in *ICML*, 2019.

[29] T. Chen, W. Zhang, Z. Jingyang, S. Chang, S. Liu, L. Amini, and Z. Wang, "Training stronger baselines for learning to optimize," in *NeurIPS*, 2020.

[30] K. Sridhar, R. Cutler, A. Saabas, T. Parnamaa, H. Gamper, S. Braun, R. Aichner, and S. Srinivasan, "ICASSP 2021 acoustic echo cancellation challenge: datasets and testing framework," *arXiv preprint arXiv:2009.04972*, 2020.

[31] J.-S. Soo and K. K. Pang, "Multidelay block frequency domain adaptive filter," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 1990.

[32] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent," 2012.

[33] J.-M. Valin, "Speex: A free codec for free speech," *arXiv preprint arXiv:1602.08668*, 2016.

[34] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proc. of the IEEE*, 1990.

[35] V. J. Mathews, *Circuits and Systems Tutorials: Adaptive polynomial filters*, 1991.

[36] M. Scarpiniti, D. Comminiello, R. Parisi, and A. Uncini, "Hammerstein uniform cubic spline adaptive filters: Learning and convergence properties," *Signal Processing*, 2014.

[37] L. R. Rabiner, B. Gold, and C. Yuen, *Theory and application of digital signal processing*. Prentice-Hall, 2016.

[38] M. Wolter and A. Yao, "Complex gated recurrent neural networks," in *NeurIPS*, 2018.

[39] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/google/jax