SEARCHING FOR MUSIC MIXING GRAPHS: A PRUNING APPROACH

Sungho Lee[†*], *Marco A. Martínez-Ramírez*[♮], *Wei-Hsiang Liao*[♮], *Stefan Uhlich*[♯], *Giorgio Fabbro*[♯], *Kyogu Lee*[†], *and Yuki Mitsufuji*[♮♭]

[†]Department of Intelligence and Information, Seoul National University, Seoul, South Korea
[♮]Sony AI, Tokyo, Japan [♯]Sony Europe B.V., Stuttgart, Germany [♭]Sony Group Corporation, Tokyo, Japan

ABSTRACT

Music mixing is *compositional* — experts combine multiple audio processors to achieve a cohesive mix from dry source tracks. We propose a method to reverse engineer this process from the input and output audio. First, we create a mixing console that applies all available processors to every chain. Then, after the initial console parameter optimization, we alternate between removing redundant processors and fine-tuning. We achieve this through differentiable implementation of both processors and pruning. Consequently, we find a sparse mixing graph that achieves nearly identical matching quality of the full mixing console. We apply this procedure to dry-mix pairs from various datasets and collect graphs that also can be used to train neural networks for music mixing applications.

1. INTRODUCTION

Motivation — From a signal processing perspective, modern music is more than the mere sum of source tracks. Mixing engineers combine and control multiple processors to balance the sources in terms of loudness, frequency content, spatialization, and much more. Many attempts have been made to uncover parts of this intricate process. Some have gathered expert knowledge [1, 2] and built rule-based systems [3, 4]. More recent work has adopted data-driven approaches. Neural networks have been trained to map source tracks directly to a mix [5, 6] or to estimate parameters of a fixed processing chain [7]. Yet, efforts to address the compositional aspects of the music mixing, such as which processors to use for each track, are still limited. One possible remedy is to consider a graph representation whose nodes and edges are processors and connections between them, respectively. In other words, each graph contains the essential information about the mixing process. However, other than the dry source and mixed audio, no public dataset provides such mixing graphs or related metadata [8, 9, 10], which hinders this line of research. This is not surprising; besides the cost of crowdsourcing, it is difficult to standardize the mixing data from multiple engineers with different equipment. One recent work [11] sidestepped this issue by creating synthetic graphs and using them for training. However, this approach is not free with downsides. Neural networks would suffer from poor generalization unless the synthetic data distribution matches the real world. Similar data-related issues arise in different domains, e.g., audio effect chain recognition [12, 13] and synthesizer sound matching [14, 15, 16]. Furthermore, real-world multitrack mixes have a much larger number of source tracks and graph sizes, making synthetic data generation more challenging. Therefore, it is desirable to have a systematic, reliable, and scalable method for collecting graphs. All these contexts lead us to ask: *Can we find the mixing graphs solely from audio?*

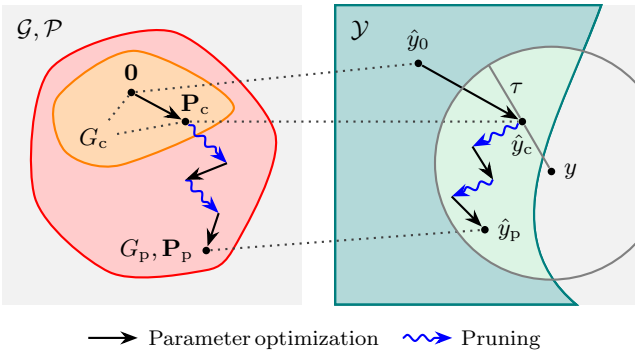


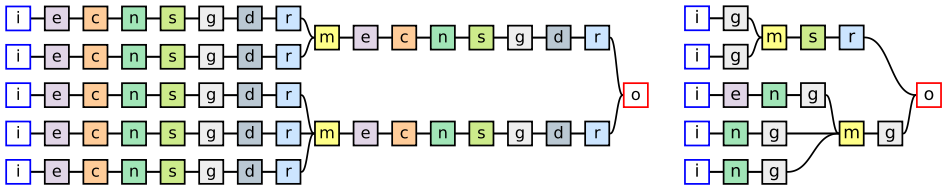
Figure 1: Music mixing graph search via iterative pruning.

Problem definition — Precisely, for each song (piece) whose dry sources $s_1, \cdots, s_K$ and mix y are available, we aim to find an audio processing graph G and its processor parameters $\mathbf{P}$ so that processing the dry sources $s_1, \cdots, s_K$ results in a mix $\hat{y}$ that closely matches the original mix y. With a loss L_a that measures the match quality on the mixture audio domain $\mathcal{Y}$ and regularization L_r, our objective can be written as follows,

$$G^*, \mathbf{P}^* = \arg\min_{G,\mathbf{P}} \big[L_\mathrm{a}(\hat{y}, y) + L_\mathrm{r}(G, \mathbf{P})\big]. \tag{1}$$

Contributions — One might want to explore the candidate graphs without any restriction. However, this makes the problem ill-posed and underdetermined. The graph's combinatorial nature makes the search space $\mathcal{G}$ extremely large. Furthermore, we have to find the processor parameters jointly. As a result, numerous pairs of graphs and parameters can have similar match quality. Therefore, it is desirable to add some restrictions, e.g., preferring structures that are widely used by practitioners. To this end, we resort to the following pruning-based search; see Figure 1 for a visual illustration. Inspired by a recent work [17], we first create a so-called "mixing console" G_c; see Figure 2a for an example. It applies a fixed processing chain to each source. Then, it subgroups the outputs, applies the chain again, and sums the processed subgroups to obtain a final mix $\hat{y}$. This resembles the traditional hybrid mixing console [18]. Each chain comprises 7 processors, including an equalizer, compressor, and multitap delay. We implement all of them in a differentiable manner [19, 20, 21]. This allows end-to-end optimization of all parameters $\mathbf{P}_\mathrm{c}$ with an audio-domain loss L_a via gradient descent. After this initial console training, we proceed to the pruning stage. Here, we search for a maximally pruned graph G_p and its parameters $\mathbf{P}_\mathrm{p}$ while maintaining the match quality of the mixing console up to a certain tolerance τ; this is visualized as

[*] Work done during an internship at Sony AI.



(a) Full mixing console (before pruning) (b) Pruned graph

Figure 2: Finding the sparse graph G_{p} from the differentiable mixing console G_{c}. Initial letters in the nodes denote their respective types. `i`: input, `o`: output, `m`: mix, `e`: equalizer, `c`: compressor, `n`: noisegate, `s`: stereo imager, `g`: gain/panning, `r`: reverb, and `d`: multitap delay.

a circle centered at y in Figure 1. Also, see Figure 2b for an example pruned graph. We use iterative pruning, alternating between the pruning and fine-tuning, i.e., optimization of the remaining parameters [22]. To collect graphs from multiple songs, it is crucial to make the entire search process, i.e., parameter optimization and pruning, efficient and fast. To this end, we first introduce batched processing of multiple nodes in a single graph, which speeds up the computation of the mix. Next, we investigate methods for efficient and effective pruning. During the pruning, we need to find a subset of nodes that can be removed. To achieve this, we view each processor's "dry/wet" parameter as an approximate importance score and use it to select the candidate nodes. This approach gives 3 variants of the pruning method with different trade-offs between the computational cost and resulting sparsity. It also draws connections to neural network pruning [23, 24] where the binary pruning operation is relaxed to continuous weights. Note that casting the graph search to pruning is a double-edged sword. The pruning only removes the processors and does not consider all possible signal routings, reducing the search space (shown as shaded regions in Figure 1). As a result, it does not improve the match quality over the mixing consoles. Nevertheless, the pruned graph follows the real-world practice of selectively applying appropriate processors. In other words, the sparsity is crucial for the graph's interpretability. Also, it keeps the search cost in a practical range, which might be challenging with other alternatives [25, 26]. Our method serves as a standalone reverse engineering algorithm [17], but it can also be used for collecting pseudo-label data to train neural networks for music mixing applications. For example, we may extend existing methods for automatic mixing [3, 4, 5, 6, 7, 27] and mixing style transfer [28] to output the graphs. This allows the end users to interpret and control the estimated outputs while leveraging the power of deep neural networks.

Data — We first report a list of datasets to which we can apply our method. For each song, we need a pair of dry sources $s_1, \cdots, s_K$ and a final mixture y. Additionally, we use subgrouping information that describes how dry tracks are grouped together. Therefore, we use the `MedleyDB` dataset [8, 9] as it provides all of them. We also add the `MixingSecrets` library [10]. Since it only provides the audio, we manually subgrouped each track based on its instrument. Finally, we include an `Internal` dataset. The resulting ensemble consists of 1129 songs (188, 472, and 579 songs for each respective dataset). The number of dry tracks ranges from 1 to 133, and the number of subgroups ranges from 1 to 26 (see Figure 6 for the statistics). Except for the final pruned graph collection stage (Section 3.4), we use a random subset for the evaluations (a total of 72 songs, 24 songs for each dataset). Every signal is stereo and resampled to 30kHz sampling rate.

Supplementary materials — Refer to the following link for audio samples, pruned graphs, and appendices that contain additional details: `https://sh-lee97.github.io/grafx-prune`.

2. DIFFERENTIABLE PROCESSING ON GRAPHS

An audio processing graph $G = (V, E)$ is assumed to be directed and acyclic (V and E denote the set of nodes and edges, respectively). Each node $v_i \in V$ is either a processor or an auxiliary module and has its type t_i, e.g., `e` for an equalizer. Each processor takes an audio u_i and a parameter vector p_i as input and outputs a processed signal $f_i(u_i, p_i)$. Then, we further mix the input and this processed result with a "dry/wet" weight $w_i \in [0, 1]$. Hence, the output y_i of the processor v_i is given as follows,

$$y_i = w_i f_i(u_i, p_i) + (1 - w_i)u_i. \tag{2}$$

We have the following 3 auxiliary modules:

- **Input —** It outputs one of the dry sources s_k.
- **Mix —** We ouput the sum of incomming signals.
- **Output —** A sum of its inputs is considered as a final output y.

Each edge $e_{ij} \in E$ represents a "cable" that sends an output signal to another node as input. Throughout the text, we denote an ordered collection from multiple nodes with a boldface letter, e.g., $\mathbf{w}$ for a weight vector, $\mathbf{S}$ for a source tensor, and $\mathbf{P}$ for a dictionary with processor types as keys and their parameter tensors as values. Under this notation, our task is to find G, $\mathbf{P}$, and $\mathbf{w}$ from $\mathbf{S}$ and y.

2.1. Differentiable Implementation

Considering the music mixing, we use the following 7 processors.

- **Gain/panning —** We control both loudness and stereo panning of input audio by multiplying a learnable scalar to each channel.
- **Stereo imager —** We change the stereo width of the input by modifying the loudness of the side channel (left minus right).
- **Equalizer —** We use a finite impulse response (FIR) with a length of 2047 to modify the input's magnitude response. The FIR is parameterized with its log magnitude (thus 1024 parameters). We apply inverse FFT of the magnitude with zero phase, obtain a zero-centered FIR, and multiply it with a Hann window. We apply the same FIR to both the left and right channels.
- **Reverb —** We employ 2 seconds of filtered noise as an impulse response for reverberation. First, we create a 2-channel uniform noise, where these channels represent the mid and side. We filter the noise by multiplying an element-wise 2-channel magnitude mask to its short-time Fourier transform (STFT), where the

FFT sizes and hop lengths are 384 and 192, respectively. This mask is constructed using the reverberation's initial and decaying log magnitudes. After the masking, we obtain the mid/side filtered noise via inverse STFT, convert it to stereo, and perform channel-wise convolutions with input to get an output.

- **Compressor —** We use a slight variant of the recently proposed differentiable dynamic range compressor [21]. First, we obtain the input's smooth energy envelope. The smoothing is typically done with a ballistics filter, but we instead use a one-pole filter for speedup in GPU. Then, we compute the desired gain reduction from the envelope and apply it to the input audio.
- **Noisegate —** Except for the gain computation, its implementation is the same as the compressor.
- **Multitap delay —** For each (left and right) channel, we employ independent 2 seconds of delay effects with a single delay for every 100ms interval. To optimize delay lengths using gradient descent, we employ surrogate complex damped sinusoids [29]. Each sinusoid is converted to a delayed soft impulse via inverse FFT. Its angular frequency represents a continuous relaxation of the discrete delay length. Each delay is filtered with a length-39 FIR equalizer to mimic the filtered echo effect [30].

Batched node processing — It is common to compute the graph output signal by processing each node one by one [15, 19]. However, this severely bottlenecks the computation speed for large mixing graphs. Therefore, we instead batch-process multiple nodes in parallel. For the graph in Figure 2b, we can batch-process 1 equalizer `e`, 3 noisegates `n`, and 5 gain/pannings `g` sequentially. Then, we aggregate the intermediate outputs to 2 subgroup mixes `m` (also in parallel). This part is identical to graph neural networks' "message passing," so we adopt their implementations [31]. We repeat these parallel computations until we reach the output node `o`. By doing so, we obtain the output faster; in this example, the number of sequential processing is reduced from 15 (one-by-one) to 8 (optimal). Also, our benchmark shows that up to $5.8\times$ speedup can be achieved for the pruned graphs with a `RTX3090` GPU. Refer to the supplementary page for further details.

2.2. Mixing Console

We construct a mixing console G_{c} as follows (see Figure 2a).

(i) We add an input node `i` for each source track.
(ii) We connect a serial chain (with a fixed order) of an equalizer `e`, compressor `c`, noisegate `n`, stereo imager `s`, gain/panning `g`, multitap delay `d`, and reverb `r` for each input.
(iii) We subgroup the processed tracks with mix nodes `m`.
(iv) We apply the same chain `ecnsgdr` to each mix output, then pass it to the output node `o` (we omit the mix module here).

2.3. Optimization

Before exploring the pruning of each mixing console, as a sanity check, we first evaluate its match performance. To investigate how much each processor type contributes to the match quality, we start with a base graph, a mixing console with no processors that simply sums all the inputs. Then, we add each processor type one by one to the processor chain (see the first column of Table 1). We optimize and evaluate all these preliminary graphs for each song. For each graph, we train its parameters and weights simultaneously with an audio-domain loss given as follows,

$$L_{\mathrm{a}} = \alpha_{\mathrm{lr}} L_{\mathrm{lr}} + \alpha_{\mathrm{m}} L_{\mathrm{m}} + \alpha_{\mathrm{s}} L_{\mathrm{s}} \tag{3}$$

		L_{a}	L_{lr}	L_{m}	L_{s}
Base graph (sum of dry sources)		19.7	1.52	1.46	74.3
+ Gain/panning	`g`	.689	.686	.634	.752
+ Stereo imager	`sg`	.676	.671	.623	.734
+ Equalizer	`e sg`	.557	.549	.493	.637
+ Reverb	`e sg r`	.481	.471	.457	.523
+ Compressor	`ec sg r`	.423	.407	.385	.492
+ Noisegate	`ecnsg r`	.414	.398	.375	.485
+ Multitap delay (full)	`ecnsgdr`	.409	.395	.375	.469

Table 1: Matching performances of the mixing consoles with different processor type configurations. The strings, e.g., `ecnsgdr`, denote the processors used in the chain and their orders.

where each term L_{x} is a variant of multi-resolution STFT loss [32] ($\mathrm{x} \in \{\mathrm{lr}, \mathrm{m}, \mathrm{s}\}$, lr: left/right, m: mid, s: side)

$$L_{\mathrm{x}} = \sum_{i=1}^{I} \left[\frac{\| \log Y_{\mathrm{x}}^{(i)} - \log \hat{Y}_{\mathrm{x}}^{(i)} \|_1}{N} + \frac{\| Y_{\mathrm{x}}^{(i)} - \hat{Y}_{\mathrm{x}}^{(i)} \|_F}{\| Y_{\mathrm{x}}^{(i)} \|_F} \right]. \tag{4}$$

Here, $Y_{\mathrm{x}}^{(i)}$ and $\hat{Y}_{\mathrm{x}}^{(i)}$ denote the i^{th} Mel spectrograms of the target and predicted mixture, respectively. N, $\| \cdot \|_1$, and $\| \cdot \|_F$ denote the number of frames, l_1 norm and Frobenius norm, respectively. We use FFT sizes of 512, 1024, and 4096, and hop sizes are $1/4$ of their respective FFT sizes. The number of Mel filterbanks is set to 96 for all scales. We apply A-weighting before each STFT [33]. The per-channel loss weights are set to $\alpha_{\mathrm{lr}} = 0.5$, $\alpha_{\mathrm{m}} = 0.25$, and $\alpha_{\mathrm{s}} = 0.25$. The implementation is based on `auraloss` [34]. We further add a regularization that promotes gain-staging, a common practice of audio engineers that keeps the total energy of input and output roughly the same. This is achieved with the following loss:

$$L_{\mathrm{g}} = \sum_{v_i \in V_{\mathrm{g}}} |\log \| f_i(u_i)_{\mathrm{m}} \|_2 - \log \| u_{i,\mathrm{m}} \|_2 | \tag{5}$$

where $(\cdot)_{\mathrm{m}}$ and $\| \cdot \|_2$ denote mid channel and l_2 norm, respectively. We apply this regularization to a subset of processors $V_{\mathrm{g}} \subset V$ that comprises all equalizers, reverbs, and multitap delays. This allows us to (i) eliminate redundant gains that these linear-time invariant (LTI) processors could create and (ii) restrict the parameters to be in a reasonable range. Therefore, the total loss is given as

$$L(\mathbf{P}, \mathbf{w}) = L_{\mathrm{a}}(\mathbf{P}, \mathbf{w}) + \alpha_{\mathrm{g}} L_{\mathrm{g}}(\mathbf{P}) \tag{6}$$

where the gain-staging weight is set to $\alpha_{\mathrm{g}} = 10^{-3}$. Here, we used a slightly different notation from Equation 1 to emphasize what is optimized. Each console is optimized for 12k steps using AdamW [35] with a 0.01 learning rate. For each step, we random-sample a 3.8s region of dry soruces $\mathbf{S}$ (thus the batch size is 1), compute the mix $\hat{y}$, and compare its last 2.8s with the corresponding ground-truth y. Note that the first second is used only for the "warm-up" of the processors with long states such as compressors and reverbs. The evaluation metrics are calculated over the entire song.

2.4. Results

Table 1 reports the evaluation results. First, the base graph results in an audio loss L_{a} of 19.7. The side-channel loss L_{s} is especially large as most source tracks are close to mono (centered) while the target mixes have wide stereo images. With the gain/pannings and

stereo imagers, we can achieve "rough mixes" with a loss of 0.676. Then, we fill in the missing details with the remaining processor types. Every type improves the match, and the full mixing console reports a loss of 0.409. In addition, see the top 5 rows of Figure 7, which shows mid/side log-magnitude STFTs of the target mixes, matches of the mixing consoles, and their errors. In the main text, we report the results with 3, 4, 6, and 7 processor types where the choice of processors and their order follow Table 1. For the results on other configurations and additional songs, see the supplementary page. From the spectrogram error plots, we can again confirm that adding each type improves the match quality. Furthermore, each song benefits more from different processor types. For example, for the song `RockSteady`, the multitap delays improve the match more than the reverbs (Figure 7b), which is different from the average trend. Yet, this is expected since the original mix heavily uses the delay effects. Finally, we note that mixes from `MixingSecrets` are more challenging to match than the others; it reports a mean audio loss of 0.545, while `MedleyDB` and `Internal` report 0.296 and 0.385, respectively.

3. MUSIC MIXING GRAPH SEARCH

Considering the full mixing console G_c as an upper bound in terms of the matching performance, we want to find a sparser graph with a similar match quality. We achieve this by pruning the console as much as possible while keeping the loss increase up to a tolerance threshold τ. This objective can be written as

$$\text{minimize } |V_\mathrm{p}| \quad \text{s.t.} \quad \min L_\mathrm{a}(G_\mathrm{p}) \leq \min L_\mathrm{a}(G_\mathrm{c}) + \tau. \quad (7)$$

Here, V_p denotes the pruned graph's node set. We define the pruning as removal of the nodes $V_\mathrm{c} \setminus V_\mathrm{p}$ and re-routing of their edges, in a way that is equivalent to setting every unused node to "bypass mode," i.e., zero weight $w_i = 0$ for $v_i \in V_\mathrm{c} \setminus V_\mathrm{p}$. Also, $\min(\cdot)$ signifies that we are (ideally) interested in the optimized audio loss. We only prune the processors, not the auxiliary modules. Hence, we define a pruning ratio ρ as the number of pruned processors over the number of processors in the initial mixing console.

3.1. Iterative Pruning

Finding the optimal (sparsest) solution V_p^* is prohibitively expensive. First, due to the interaction between the processors, we need a combinatorial search. As such, we instead assume their independence and prune the processors in a greedy manner. Following the iterative approach [22], we gradually remove processors whenever the tolerance condition is satisfied. Under this setup, we still need to fine-tune intermediate pruned graphs before evaluating the tolerance condition. For reasonable computational complexity, we simply omit this fine-tuning, paying the cost of possibly missing more removable processors. Our method is summarized in Algorithm 1 (in the following parentheses denote line numbers). We first construct the mixing console G_c, optimize its parameters $\mathbf{P}$ and dry/wet weights $\mathbf{w}$, and evaluate the audio loss (1-3). This validation loss $L_\mathrm{a}^\mathrm{min}$ serves as a pruning threshold with the tolerance τ. Then, we alternate between pruning and fine-tuning, i.e., further optimization of the remaining parameters $\mathbf{P}$ and dry/wet weights $\mathbf{w}$ (5-18). Each stage consists of multiple pruning trials, which sample subsets of candidate processors $\bar{V}_\mathrm{cand}$ from the set of remaining processors V_cand (8) and check whether they are removable (10). We keep the pruning if the result satisfies the constraint or cancel it otherwise (10-13). We repeat this process until the terminal condition (7) is satisfied. Implementation-wise, we multiply binary masks, $\mathbf{m}$ and $\bar{\mathbf{m}}$, to the weight vector $\mathbf{w}$ to mimic the pruning during the trials (9). After that, we actually update the graph and remove the pruned processors' parameters and weights for faster search (16). Sometimes, albeit rare, the pruning can improve the match. In this case, we update the threshold (11).

Algorithm 1 Music mixing graph search with iterative pruning.

Input: A mixing console G_c, dry tracks $\mathbf{S}$, and mixture y
Output: Pruned graph G_p, parameters $\mathbf{P}$, and weights $\mathbf{w}$

1: $\mathbf{P}, \mathbf{w} \leftarrow \mathrm{Initialize}(G_\mathrm{c})$
2: $\mathbf{P}, \mathbf{w} \leftarrow \mathrm{Train}(G_\mathrm{c}, \mathbf{P}, \mathbf{w}, \mathbf{S}, y)$
3: $L_\mathrm{a}^\mathrm{min} \leftarrow \mathrm{Evaluate}(G_\mathrm{c}, \mathbf{P}, \mathbf{w}, \mathbf{S}, y)$
4: $G_\mathrm{p} \leftarrow G_\mathrm{c}$
5: **for** $n \leftarrow 1$ to N_iter **do**
6: $\quad V_\mathrm{cand}, \mathbf{m} \leftarrow \mathrm{GetAllProcessors}(V), \mathbf{1}$
7: $\quad$ **while** $\mathrm{TryPrune}(V_\mathrm{cand}, \mathbf{w}, \mathbf{m})$ **do**
8: $\quad\quad \bar{V}_\mathrm{cand}, \bar{\mathbf{m}} \leftarrow \mathrm{SampleCandidate}(V_\mathrm{cand}, \mathbf{w})$
9: $\quad\quad L_\mathrm{a} \leftarrow \mathrm{Evaluate}(G_\mathrm{p}, \mathbf{P}, \mathbf{w} \odot \mathbf{m} \odot \bar{\mathbf{m}}, \mathbf{S}, y)$
10: $\quad\quad$ **if** $L_\mathrm{a} < L_\mathrm{a}^\mathrm{min} + \tau$ **then**
11: $\quad\quad\quad L_\mathrm{a}^\mathrm{min} \leftarrow \min(L_\mathrm{a}^\mathrm{min}, L_\mathrm{a})$
12: $\quad\quad\quad \mathbf{m} \leftarrow \mathbf{m} \odot \bar{\mathbf{m}}$
13: $\quad\quad$ **end if**
14: $\quad\quad V_\mathrm{cand} = \mathrm{UpdatePool}(V_\mathrm{cand}, \bar{V}_\mathrm{cand})$
15: $\quad$ **end while**
16: $\quad G_\mathrm{p}, \mathbf{P}, \mathbf{w} \leftarrow \mathrm{Prune}(G_\mathrm{p}, \mathbf{P}, \mathbf{w}, \mathbf{m})$
17: $\quad \mathbf{P}, \mathbf{w} \leftarrow \mathrm{Train}(G_\mathrm{p}, \mathbf{P}, \mathbf{w}, \mathbf{S}, y)$
18: **end for**
19: **return** $G_\mathrm{p}, \mathbf{P}, \mathbf{w}$

3.2. Candidate Sampling

The remaining design choices are sampling an appropriate candidate set V_cand (8, 14) and deciding when to terminate the trials (7). We explore the following 3 approaches.

- **Brute-force —** We random-sample every processor one by one, i.e., $|\bar{V}_\mathrm{cand}| = 1$. This granularity could achieve high sparsity, but come with a large computational cost.
- **Dry/wet —** For efficient pruning, we need an informed guess of each node's importance. Intuitively, we can use each dry/wet weight w_i as an approximate importance score. This observation leads to the following method. For each pruning iteration:
  (i) We construct a set of remaining processor types T_cand.
  (ii) For each trial, we sample a type $t \in T_\mathrm{cand}$ and use the smallest-weight processors of that type as candidates. The number of candidates $|V_\mathrm{cand}|$ is set to 10% of the number of type-t processors that existed at the beginning.
  (iii) When the trial fails: if $|V_\mathrm{cand}| > 1$, we reduce the candidate set size to half: $|V_\mathrm{cand}|/2$. If $|V_\mathrm{cand}| = 1$, we finish the search of this type t, i.e., $T_\mathrm{cand} \leftarrow T_\mathrm{cand} \setminus \{t\}$.
  (iv) We iterate above two (ii)-(iii) until $T_\mathrm{cand} = \emptyset$.
  This way, we can skip large-weight nodes and evaluate multiple candidate nodes, reducing the total number of trials.
- **Hybrid —** Solely relying on the weight values could miss some processors that can be pruned but have large weights. We mitigate this by combining the above two, running the brute-force method for every 4^th iteration.

By default, we use the hybrid method with tolerance $\tau = 0.01$.

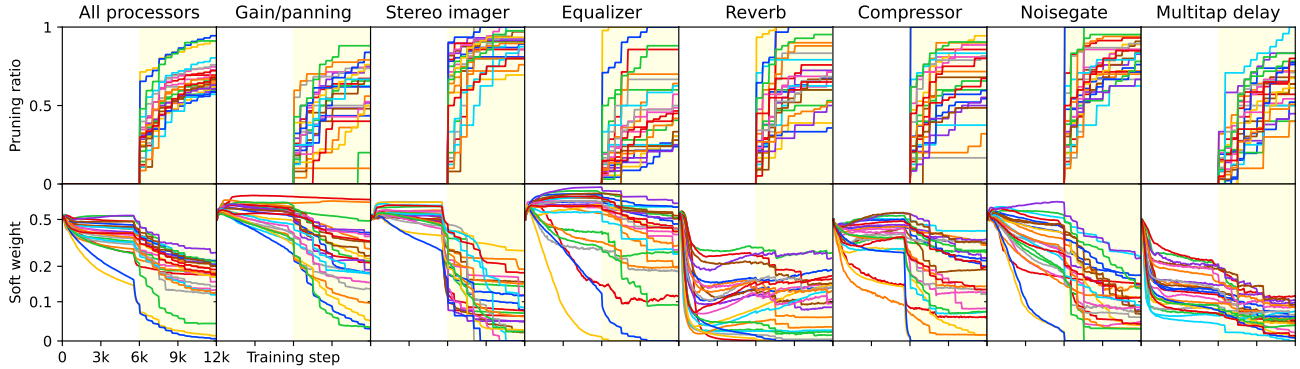


Figure 3: Iterative pruning process (hybrid, $\tau = 0.01$). 24 songs, 8 songs per dataset, are shown; each color represents an individual song. The upper and lower rows show the pruning ratios and mean dry/wet weights. The yellow-shaded regions show the pruning phase.

	τ	L_a	ρ	$\rho_\mathtt{g}$	$\rho_\mathtt{s}$	$\rho_\mathtt{e}$	$\rho_\mathtt{r}$	$\rho_\mathtt{c}$	$\rho_\mathtt{n}$	$\rho_\mathtt{d}$
Mix console	–	.409	–	–	–	–	–	–	–	–
Brute-force	.01	.424	.69	.54	.85	.53	.76	.71	.78	.69
Dry/wet	.01	.420	.62	.51	.84	.38	.69	.66	.76	.53
Hybrid	.001	.411	.49	.35	.76	.27	.53	.57	.62	.34
	.01	.422	.67	.51	.86	.46	.71	.71	.79	.63
	.1	.499	.87	.73	.94	.81	.90	.85	.91	.92

Table 2: Pruning results with various candidate selection strategies and tolerance τ. The subscripts denote per-type pruning ratios.

3.3. Optimization

We use identical audio loss L_a and gain-staging regularization L_g. To promote sparsity, we add a weight regularization L_p, a l_1 norm of the weight $\mathbf{w}$. Hence, the full objective is as follows,

$$L(\mathbf{P}, \mathbf{w}) = L_\mathrm{a}(\mathbf{P}, \mathbf{w}) + \alpha_\mathrm{g} L_\mathrm{g}(\mathbf{P}) + \alpha_\mathrm{p} L_\mathrm{p}(\mathbf{w}). \tag{8}$$

We first train the console with 6k steps. Then, we repeat $N_\mathrm{iter} = 12$ rounds of pruning, each with 0.5k-step fine-tuning. As a result, the total number of optimization steps is the same as the previous console training. During the first 4k steps of the pruning phase, we linearly increase the sparsity coefficient α_p from 0 to 10^{-2}. While we halved the full console optimization steps, which could lead to increased loss, it is justified due to the tight resource constraints. With a `RTX3090` GPU, each song took about 56m, 29m, and 36m using the brute-force, dry/wet, and hybrid methods, respectively.

3.4. Results

Pruning process — Figure 2 shows how the pruning progresses. Each graph's sparsity increases gradually while its weights adapt over time. This trend is different for different processor types. The mean objective metrics are reported in Table 2. The default setting reports an average audio loss L_a of 0.422, an 0.013 increase from the full consoles, slightly exceeding the tolerance $\tau = 0.01$. This was expected due to the shorter full console training. The average pruning ratio ρ is 0.67 and the equalizer and stereo imager are the most and least remaining types (0.46 and 0.86), respectively. We note that `MedleyDB` and `MixingSecrets` report similar pruning ratios of 0.61 and 0.62, respectively. However, the `Internal` graphs are more sparse; their average pruning ratio is 0.77.

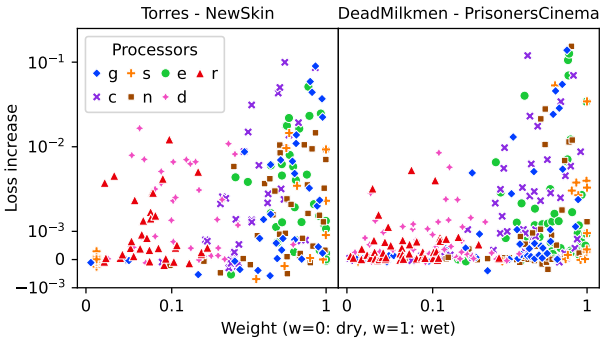
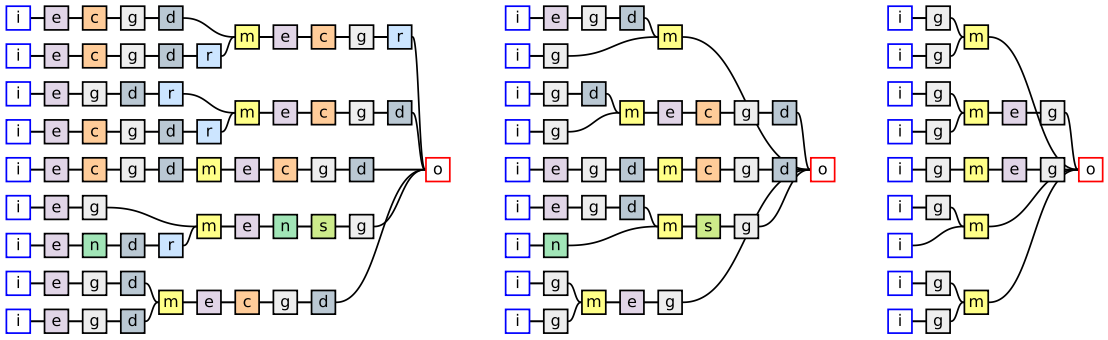


Figure 4: Each node's weight and loss increase when pruned.

Sampling method comparison — Here, we fix the tolerance τ to 0.01 and compare the candidate sampling approaches; see Table 2. As expected, the brute-force method achieves the highest sparsity, reporting a pruning ratio of 0.69. Its average audio loss is also the highest, 0.424, an 0.015 increase from the mixing console result. The dry/wet method prunes the least with 0.62, 7% lower than the brute-force method. However, its audio loss is the lowest, 0.420, as more processors remained. We can investigate the cause of this difference in sparsity by analyzing the relationship between each dry/wet weight w_i and the loss increase Δ_i caused by pruning the processor v_i defined as follows,

$$\Delta_i = L_\mathrm{a}(G \setminus \{v_i\}) - L_\mathrm{a}(G). \tag{9}$$

Figure 4 shows scatterplots for 2 random-sampled songs, one for each song. Each point (w_i, Δ_i) corresponds to each processor after the initial console training. To maximize the sparsity using the dry/wet method, a monotonic relationship between the weights w_i and loss increases Δ_i is desirable, which is unfortunately not the case. Yet, a positive correlation exists, and this becomes more evident when we analyze the relationship for each type separately, justifying the per-type candidate selection. Still, we cannot completely remove the weakness of the dry/wet method, leading us to the hybrid strategy as a compromise. We note that the pruning methods are not only different in sparsity but also in trade-offs between sparsity and match performance. By evaluating the methods with more fine-grained tolerance settings (7 values from 0.001 to 0.2), we observed that the brute-force method finds graphs with better matches even with the same graph size, closely followed by the hybrid method; refer to the supplementary page for the details.



(a) $\tau = 0.001$
$\rho = 0.45, L_{\mathrm{a}} = 0.525$

(b) $\tau = 0.01$
$\rho = 0.74, L_{\mathrm{a}} = 0.525$

(c) $\tau = 0.1$
$\rho = 0.85, L_{\mathrm{a}} = 0.611$

Figure 5: Pruning results (hybrid method) with various tolerances. Song: `TablaBreakbeatScience_RockSteady`.

Choice of tolerance — Finally, we analyze the effect of the value of tolerance τ. Even with a very low tolerance $\tau = 0.001$, we can nearly halve the number of processors, i.e., $\rho = 0.49$. If we set the value too high, e.g., $\tau = 0.1$, the resulting graphs are highly sparse but degrade their matches ($L_{\mathrm{a}} = 0.499$, i.e., 0.090 increase). The default setting of $\tau = 0.01$ seems "just right," balancing the match performance and graph sparsity. We can verify this with the spectrogram errors (bottom 3 rows for each figure). There is no noticeable degradation from the full consoles to $\tau = 0.001$ and 0.01.

Case study — Here, we report the behavior of the pruning method from observations of the individual results.

- Recall that, for the song `RockSteady`, there was no clear performance improvement when we added the reverbs (Figure 7b). Hence, we can expect those reverbs to be pruned with a moderate tolerance τ. Figure 5 shows that this is indeed the case; only $5/14$ reverbs are left when $\tau = 0.001$ and $0/14$ for $\tau = 0.01$, which is much less than the average statistics (Table 2 and 3). When $\tau = 0.1$, processors for the details get removed; only the gain/pannings and equalizers remain. See captions in Figure 5 for the pruning ratios and audio losses of the pruned graphs (the full console has an audio loss of 0.523).
- The current pruning method fails to detect some redundant processors. In Figure 5b, the bottom 2 sources are processed with 3 gain/pannings. Since there is no nonlinear or time-varying processor between those, at least one can be "absorbed" by the others. While this case can be handled with some post-processing, it hints that we might have missed more sparse graphs.
- Each pruning of the same song yields a slightly different graph. Pruning a mixing console of `GirlOnABridge` multiple times resulted in graphs with the number of processors from 19 to 22. This is because our iterative pruning has a stochastic and greedy nature; candidates that were sampled early are more likely to be pruned. Refer to the supplementary page for the pruned graphs.
- The pruning does not necessarily result in graphs that are close to the maximum loss $L_{\mathrm{a}}(G_{\mathrm{c}}) + \tau$. For `RockSteady`, pruning with $\tau = 0.01$ resulted in a loss of 0.525, much lower than the threshold. Interestingly, the $\tau = 0.001$ case achieved the same loss in spite of a much lower pruning ratio (0.56 versus 0.74).
- Processors for sources with short spans and low energy tend to get pruned as their contributions to the audio loss are small. Yet, we found that this could sometimes be perceptually noticeable.

	L_{a}	ρ	$\rho_{\mathtt{g}}$	$\rho_{\mathtt{s}}$	$\rho_{\mathtt{e}}$	$\rho_{\mathtt{r}}$	$\rho_{\mathtt{c}}$	$\rho_{\mathtt{n}}$	$\rho_{\mathtt{d}}$
`MedleyDB`	.431	.63	.37	.84	.44	.69	.74	.77	.57
`MixingSecrets`	.625	.64	.50	.87	.33	.64	.63	.80	.69
`Internal`	.434	.75	.70	.87	.55	.73	.85	.86	.72
`Total`	.506	.69	.57	.87	.45	.69	.75	.82	.69

Table 3: Pruning results with the default setting on the full dataset.

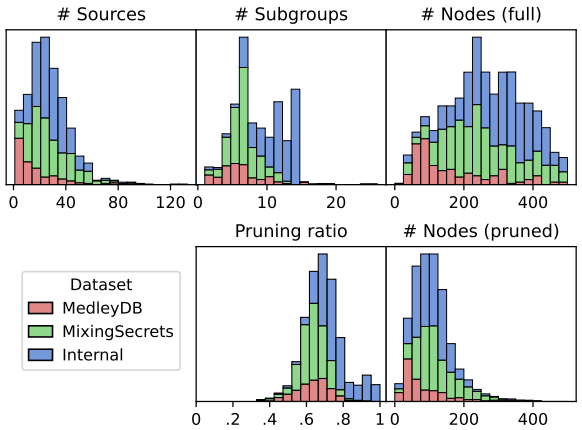


Figure 6: Statistics of the consoles and pruned graphs (full data). Each dataset's results are stacked to form the full histograms.

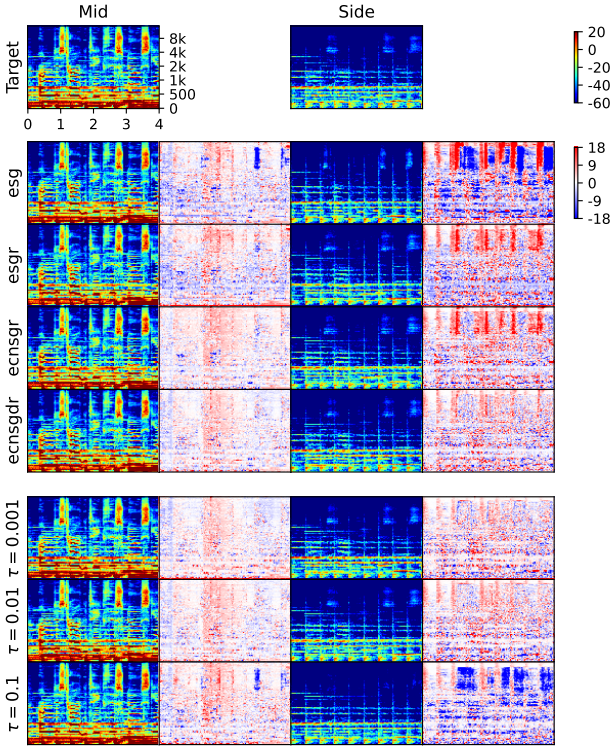
Full results — Finally, we pruned every song in the full dataset ensemble. Table 3 reports the results. The overall trend follows the evaluation subset results but with higher average audio loss (0.509 compared to the previous 0.422). Figure 6 shows statistics of the 3 datasets, initial mixing console graphs, and their pruned versions. `MedleyDB` has the smallest number of source tracks, an average of 17.6. The `Internal` has the largest (28.8), closely followed by the `MixingSecrets` (27.9). The `Internal` dataset also has more subgroups, resulting in even larger mixing consoles. This is one potential cause of the higher sparsity of its pruned graphs; more processors were initially used to match the mix, and many of them were redundant. On average, 72.1 processors (108.5 nodes) were used for each song. Since each full mixing console has 247.6 processors on average, we achieved a pruning ratio of 0.692.

4. DISCUSSION

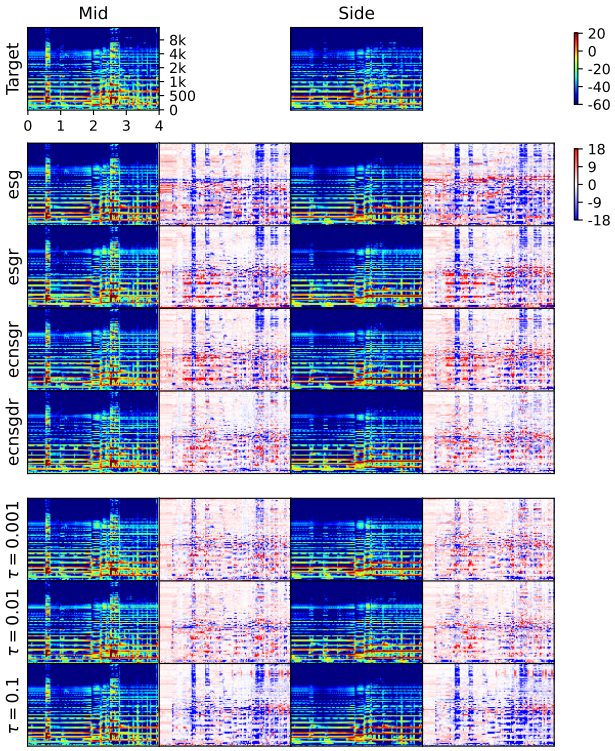
Summary — We started with a general formulation of retrieving mixing graphs from dry sources and mix. Then, we posed restrictions to cast the search to the pruning of mixing consoles, making it computationally feasible and obtaining more interpretable graphs. Next, with additional assumptions, we derived the iterative method that gradually removes negligible processors in a stochastic and greedy manner. As a consequence, instead of finding the exact optimal, our method gives one of close-to-optimal graphs. This can be viewed as posterior sampling, where the prior encodes the inductive bias on graphs, e.g., processors must follow the fixed order and each should appear at most once in every chain. The likelihood corresponds to the match quality. With the differentiable processors and soft relaxation of the binary pruning with the dry/wet weights, we optimized this objective via gradient descent. We further explored multiple ways to choose pruning candidates, comparing them in terms of their computational cost and the graphs' sparsity. We found that the hybrid method gives a good compromise, so we used it to gather over one thousand graph-audio pairs.

Future works — Before concluding the main text, we list possible variations and extensions of our method.

- The choice of processors and their implementations directly affect the match quality. Our setup, such as the equalizer with zero-phase FIR and the reverb based on STFT mask, was motivated by its simplicity and fast computation in GPU. However, we can use other alternatives, especially digital filters such as parametric equalizer [20] and artificial reverb [36], for different parameterizations. Especially, the spectrogram errors show clear temporal patterns (vertical stripes), hinting that the loudness dynamics are not precisely matched. We suspect it was caused by the ballistics approximation error as recently reported; if so, we might need a more sophisticated implementation of the compressor and noisegate [37]. We can also try different processor types, e.g., distortion [38] or modulation effects [39]. Finally, the current method does not support time-varying parameters, which might cause audible errors. For example, we could not match the fade-out, i.e., the gradual decrease of track gains, and incorrectly estimated the loudness.
- We note several considerations to improve the current pruning method in terms of sparsity, match quality, and interpretability. First, we can modify the mixing consoles to reflect real-world practices more. For example, we can add send and return loops with additional processor chains. Second, to prevent the pruning from harming the perceptual quality, the tolerance condition and the objective function must be appropriately designed. We used a simple multi-resolution STFT loss [32, 34], which has been reported to miss some perceptual features [40, 41]. Hence, we might need an alternative objective as a remedy [42]. Third, the use of average loss to determine the pruning might be inappropriate, as discussed before. Lastly, to increase the sparsity, more sophisticated neural network pruning techniques [23, 24] or domain-specific post-processing can be applied.
- We can expand the search space by relaxing the prior assumptions and restrictions on graph structures. If we allow arbitrary processor order, our framework extends to differentiable architecture search [25, 26]. A completely different approach based on reinforcement learning could also be possible [43]. While all of these are promising, balancing flexibility, match quality, and computation cost will be the main challenge.



(a) `Torres_NewSkin`



(b) `TablaBreakbeatScience_RockSteady`

Figure 7: Log-magnitude spectrograms of the matched mixes (odd columns) of the mixing consoles (4 center rows) and pruned graphs (3 bottom rows; in dB). The even columns show the match errors.

5. REFERENCES

[1] P. D. Pestana and J. D. Reiss, "Intelligent audio production strategies informed by best practices," 2014.

[2] F. Everardo, "Towards an automated multitrack mixing tool using answer set programming," in *14th SMC Conf*, 2017.

[3] E. Perez-Gonzalez and J. Reiss, "Automatic gain and fader control for live mixing," in *IEEE WASPAA*, 2009.

[4] B. De Man and J. D. Reiss, "A knowledge-engineered autonomous mixing system," in *AES Convention 135*, 2013.

[5] M. A. Martinez Ramirez *et al.*, "Automatic music mixing with deep learning and out-of-domain data," in *ISMIR*, 2022.

[6] D. Koszewski, T. Görne, G. Korvel, and B. Kostek, "Automatic music signal mixing system based on one-dimensional wave-u-net autoencoders," *EURASIP Journal on Audio, Speech, and Music Processing*, vol. 2023, no. 1, 2023.

[7] C. J. Steinmetz, J. Pons, S. Pascual, and J. Serrà, "Automatic multitrack mixing with a differentiable mixing console of neural audio effects," in *IEEE ICASSP*, 2021.

[8] R. M. Bittner, J. Salamon, M. Tierney, M. Mauch, C. Cannam, and J. P. Bello, "MedleyDB: A multitrack dataset for annotation-intensive mir research." in *ISMIR*, vol. 14, 2014.

[9] R. M. Bittner, J. Wilkins, H. Yip, and J. P. Bello, "MedleyDB 2.0: New data and a system for sustainable data collection," *ISMIR LBD*, 2016.

[10] M. Senior, *Mixing secrets for the small studio*, 2018.

[11] S. Lee, J. Park, S. Paik, and K. Lee, "Blind estimation of audio processing graph," in *IEEE ICASSP*, 2023.

[12] C. Mitcheltree and H. Koike, "SerumRNN: Step by step audio VST effect programming," in *Artificial Intelligence in Music, Sound, Art and Design*, 2021.

[13] J. Guo and B. McFee, "Automatic recognition of cascaded guitar effects," in *DAFx*, 2023.

[14] N. Masuda and D. Saito, "Improving semi-supervised differentiable synthesizer sound matching for practical applications," *IEEE/ACM TASLP*, vol. 31, 2023.

[15] N. Uzrad *et al.*, "DiffMoog: a differentiable modular synthesizer for sound matching," *arXiv:2401.12570*, 2024.

[16] F. Caspe, A. McPherson, and M. Sandler, "DDX7: Differentiable FM synthesis of musical instrument sounds," in *ISMIR*, 2022.

[17] J. Colonel, "Music production behaviour modelling," 2023.

[18] "The mixing console — split, inline and hybrids," https://steemit.com/sound/@jamesub/the-mixing-console-split-inline-and-hybrids, accessed: 2024-02-26.

[19] J. Engel, L. H. Hantrakul, C. Gu, and A. Roberts, "DDSP: differentiable digital signal processing," in *ICLR*, 2020.

[20] S. Nercessian, "Neural parametric equalizer matching using differentiable biquads," in *DAFx*, 2020.

[21] C. J. Steinmetz, N. J. Bryan, and J. D. Reiss, "Style transfer of audio effects with differentiable signal processing," *JAES*, vol. 70, no. 9, 2022.

[22] G. Castellano, A. M. Fanelli, and M. Pelillo, "An iterative pruning algorithm for feedforward neural networks," *IEEE transactions on Neural networks*, vol. 8, no. 3, 1997.

[23] Y. He and L. Xiao, "Structured pruning for deep convolutional neural networks: A survey," *arXiv:2303.00566*, 2023.

[24] H. Cheng, M. Zhang, and J. Q. Shi, "A survey on deep neural network pruning-taxonomy, comparison, analysis, and recommendations," *arXiv:2308.06767*, 2023.

[25] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in *ICLR*, 2019.

[26] Z. Ye, W. Xue, X. Tan, Q. Liu, and Y. Guo, "NAS-FM: Neural architecture search for tunable and interpretable sound synthesis based on frequency modulation," *arXiv:2305.12868*, 2023.

[27] C. J. Steinmetz, S. S. Vanka, M. A. Martínez-Ramírez, and G. Bromham, *Deep Learning for Automatic Mixing*. ISMIR, Dec. 2022.

[28] J. Koo *et al.*, "Music mixing style transfer: A contrastive learning approach to disentangle audio effects," in *IEEE ICASSP*, 2023.

[29] B. Hayes, C. Saitis, and G. Fazekas, "Sinusoidal frequency estimation by gradient descent," in *IEEE ICASSP*, 2023.

[30] U. Zölzer, Ed., *DAFX: Digital Audio Effects*, 2nd ed., 2011.

[31] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv:1903.02428*, 2019.

[32] R. Yamamoto, E. Song, and J.-M. Kim, "Parallel wavegan: A fast waveform generation model based on generative adversarial networks with multi-resolution spectrogram," in *IEEE ICASSP*, 2020.

[33] A. Wright and V. Välimäki, "Perceptual loss function for neural modeling of audio systems," in *IEEE ICASSP*, 2020.

[34] C. J. Steinmetz and J. D. Reiss, "auraloss: Audio focused loss functions in PyTorch," in *Digital Music Research Network One-day Workshop (DMRN+15)*, 2020.

[35] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv:1711.05101*, 2017.

[36] S. Lee, H.-S. Choi, and K. Lee, "Differentiable artificial reverberation," *IEEE/ACM TASLP*, vol. 30, 2022.

[37] C. J. Steinmetz, T. Walther, and J. D. Reiss, "High-fidelity noise reduction with differentiable signal processing," in *AES Convention 155*, 2023.

[38] J. T. Colonel, M. Comunità, and J. Reiss, "Reverse engineering memoryless distortion effects with differentiable waveshapers," in *AES Convention 153*, 2022.

[39] A. Carson, S. King, C. V. Botinhao, and S. Bilbao, "Differentiable grey-box modelling of phaser effects using frame-based spectral processing," in *DAFx*, 2023.

[40] B. Hayes, J. Shier, G. Fazekas, A. McPherson, and C. Saitis, "A review of differentiable digital signal processing for music & speech synthesis," *Frontiers in Signal Process.*, 2023.

[41] J. Turian and M. Henry, "I'm sorry for your loss: Spectrally-based audio distances are bad at pitch," in *"I Can't Believe It's Not Better!" NeurIPS workshop*, 2020.

[42] C. Vahidi *et al.*, "Mesostructures: Beyond spectrogram loss in differentiable time–frequency analysis," *JAES*, 2023.

[43] J. You, B. Liu, Z. Ying, V. Pande, and J. Leskovec, "Graph convolutional policy network for goal-directed molecular graph generation," *NeurIPS*, 2018.