A SOFTWARE FRAMEWORK FOR MUSICAL DATA AUGMENTATION

Brian McFee[1,2,*], **Eric J. Humphrey**[2,3], **and Juan P. Bello**[2]

[1]Center for Data Science, New York University
[2]Music and Audio Research Laboratory, New York University
[3]MuseAmi, Inc.

ABSTRACT

Predictive models for music annotation tasks are practically limited by a paucity of well-annotated training data. In the broader context of large-scale machine learning, the concept of "data augmentation" — supplementing a training set with carefully perturbed samples — has emerged as an important component of robust systems. In this work, we develop a general software framework for augmenting annotated musical datasets, which will allow practitioners to easily expand training sets with musically motivated perturbations of both audio and annotations. As a proof of concept, we investigate the effects of data augmentation on the task of recognizing instruments in mixed signals.

1. INTRODUCTION

Musical audio signals contain a wealth of rich, complex, and highly structured information. The primary goal of content-based music information retrieval (MIR) is to analyze, extract, and summarize music recordings in a human-friendly format, such as semantic tags, chord and melody annotations, or structural boundary estimations. Modeling the vast complexity of musical audio seems to require large, flexible models with many parameters. By the same token, parameter estimation in large models often requires a large number of samples: big models require big data.

Within the past few years, this phenomenon of increasing model complexity has been observed in the computer vision literature. Currently, the best-performing models for recognition of objects in images exploit two fundamental properties to overcome the difficulty of fitting large, complex models: access to large quantities of annotated data, and label-invariant data transformations [14]. The benefits of large training collections are obvious, but unfortunately difficult to achieve for most musical annotation tasks due to the complexity of the label space and need for expert annotators. However, the idea of generating perturbations of a training set — known as *data augmentation* — can be readily adapted to musical tasks.

*Please direct correspondence to `brian.mcfee@nyu.edu`

Conceptually, data augmentation consists of the application of one or more deformations to a collection of (annotated) training samples. Data augmentation is motivated by the observation that a learning algorithm may generalize better if it is trained on instances which have been perturbed in ways which are irrelevant to their labels. Some concrete examples of deformations drawn from computer vision include translation, rotation, reflection, and scaling. These simple operations are appealing because they typically do not affect the target class label: an image of a cat still contains a cat, even when it is flipped upside-down.

More generally, deformations apply not only to observable features, but the labels as well. Continuing with the image example, if an image is rotated, then any pixel-wise label annotations (*e.g.*, bounding boxes) should be rotated accordingly. This observation opens up several interesting possibilities for musical applications, in which the target concept space typically exhibits a high degree of structure. A musical analog to the image rotation example would be time-stretching, where time-keyed annotation boundaries (*e.g.*, chord labels or instrument activations) must be adjusted to fit the stretched signal [16].

Many natural, musically-inspired deformations would not only change the position of annotations, but the *values* themselves. For instance, if a time-stretched track has tempo annotations, the annotation values should be scaled accordingly. Similarly, pitch-shifting a track should induce transpositions of annotated fundamental frequency curves, and if the transposition is sufficiently large, chord labels or symbolic annotations may change as well. Because the annotation spaces for music tasks often exhibit a high degree of structure, successful application of data augmentation may require a more sophisticated approach in MIR than in other domains.

1.1 Our contributions

In this work, we describe the MUDA software architecture for applying data augmentation to music information retrieval tasks. [1] The system is designed to be simple, modular, and extensible. The design enables practitioners to develop custom deformations, and combine multiple simple deformations together into pipelines which can generate large volumes of reliably deformed, annotated music data. The proposed system is built on top of JAMS [12],

[1] `https://bmcfee.github.io/muda`

which provides a simple container for accessing and transporting multiple annotations for a given track.

We demonstrate the proposed data augmentation architecture with the application of recognizing instruments in mixed signals, and show that simple manipulations can yield improvements in accuracy.

2. RELATED WORK

The first step in developing a solution to an MIR problem is often to design features which discard information thought to be irrelevant to the target concept. For example, chroma features are designed to capture pitch class information and suppress timbre, loudness, or octave height [18]. Similarly, many authors interested in modeling timbre use Mel-frequency cepstral coefficients (MFCCs) and discard the first component to achieve invariance to loudness [19]. This general strategy makes intuitive sense, but it carries many limitations. First, it is not necessarily easy to identify all relevant symmetries in the data: if it was, the modeling problem would be essentially solved. Second, even if such properties are easy to identify, it may still be difficult to engineer appropriately invariant features without discarding potentially useful information. For example, 2-D Fourier magnitude coefficients achieve invariance to time- and pitch-transposition, but discard phase coherence [8].

As an alternative to custom feature design, some authors advocate learning or optimizing features directly from the data [11]. Not surprisingly, this approach typically requires large model architectures, and much larger (annotated) data sets than had previously been used in MIR research. Due to the high cost of acquiring annotated musical data, it has so far been difficult to apply these techniques in most MIR tasks. While some authors have advocated leveraging unlabeled data to "pre-train" feature representations [6], recent studies have shown that comparable or better performance can be achieved with random initialization and fully supervised training [9, 22]. Our goal in this work is to provide data augmentation tools which may ease the burden of sample complexity, and make data-driven methodology more accessible to the MIR community.

Specific instances of data augmentation can be found throughout the MIR literature, though they are not often identified as such, nor are they treated systematically in a unified framework. For example, it is common to apply circular rotations to chroma features to achieve key invariance when modeling chord quality [15]. Alternately, synthetic mixtures of monophonic instruments have been used to generate more difficult examples when training polyphonic transcription engines [13]. Some authors even leave the audio content unchanged and only modify labels during training, as exemplified by the *target smearing* method of Ullrich *et al.* for training structural boundary detectors [21].

Finally, recent studies have used degraded signals to evaluate the stability of existing methods for MIR tasks. The Audio Degradation Toolbox (ADT) was developed for this purpose, and was used to measure the impact of naturalistic deformations of audio on several tasks, including beat tracking, score alignment, and chord recognition [16]. Similarly, Sturm and Collins proposed the "Kiki-Bouba Challenge" as a way to determine whether statistical models of musical concepts actually capture the defining characteristics of the concept (*e.g.*, genre), or are over-fitting to spurious correlations [20].

In both of the studies cited above, models are fit to unmodified data, and evaluated in degraded conditions under the control of the experimenter. Data augmentation provides the converse of this setting: models are fit to degraded data, and evaluated on unmodified examples. The distinction between the two approaches is critical. The former attempts to measure the robustness of a system under synthetic conditions, while the latter attempts to improve robustness by *training* under synthetic conditions. Note that with data augmentation, the evaluation set is left untouched by the experimenter, so the resulting comparisons are unbiased with respect to the underlying distribution from which the data are sampled. While this does not directly measure robustness, it has been observed that data augmentation can improve generalization [10, 14].

3. DATA AUGMENTATION ARCHITECTURE

Our implementation takes substantial inspiration from the Audio Degradation Toolbox [16]. In principle, the ADT can be used directly for some forms of data augmentation simply by applying it to the training set rather than test set. However, we opted for an independent, Python-based implementation for a variety of reasons.

First, Python enables object-oriented design, allowing for structured, extensible, and reusable code. This in turn facilitates a simple interface shared across all *deformation objects*, and makes it easy for practitioners to combine or extend existing deformations.

Second, we use JAMS [12] both to transport and store track annotations, and as an internal data structure for processing. JAMS provides a unified interface to different annotation types, and a convenient framework to manage all annotations for a particular track. This simplifies the tasks of maintaining synchronization between audio and annotations, and implementing task-dependent annotation deformations. We also adapt JAMS sandbox fields to provide data provenance and facilitate reproducibility.

Finally, we borrow familiar software design patterns from the scikit-learn package [4], such as *transformers*, *pipelines*, and model serialization. These building blocks allow practitioners to quickly and easily assemble complex pipelines from small, conceptually simple components.

In the remainder of this section, we will describe the software architecture in more detail. Without loss of generality, we assume that an annotation (*e.g.*, instrument activations) is encoded as a collection of tuples: *(time, duration, value, confidence)*. Note that instantaneous events can be represented with zero duration, while track-level annotations have full-track duration. The *value* field depends on the annotation type, and may encode strings, numeric quantities, or fully structured objects.

3.1 Deformation objects

At the core of our implementation is the concept of a *deformation object*. We will first describe deformation objects in terms of their methods and abstract properties. Section 3.1.1 follows with a concrete, but high-level example.

A deformation object implements one or more *transformation* methods, each of which applies to either audio, meta-data, or annotations. Parameters of the deformation are shared through a *state* object S. For example, S might contain the speed-up factor of a time-stretch, or the number of semi-tones in a pitch-shift. Each transformation method takes as input a pair (S, x) and returns the transformed audio, meta-data, or annotation x'. Decoupling the deformation object's instantiation from its state allows multiple tracks to be processed in parallel by the same object. Moreover, as described in Section 3.3, state objects are reusable, which promotes reproducibility.

Data augmentation often requires sampling or sweeping a set of deformation parameters, and instantiating a separate deformation object for each parameterization can be inefficient, especially when the S contains non-trivial data (*e.g.*, tuning estimates or noise signals). Instead, a deformation object implements a *state generator*, which may execute arbitrary transition logic to produce a sequence of states $(S_1, S_2, \dots)$. This is implemented efficiently using Python *generators*.

Finally, deformation objects may register transformation functions against the *type* of an annotation, as described by regular expressions. This allows different transformation procedures to be applied to different annotation types. During execution, the JAMS object is queried for all annotations matching the specified expression, and the results are processed by the corresponding transformation method. For example, the expression "`.*`" matches all annotation types, while "`chord.*`" matches only chord-type annotations. These patterns need not be unique or disjoint, though care must be taken to ensure consistent behavior. Deformations are always applied following the order in which they are registered.

The abstract transformation algorithm is described in Algorithm 1. For each state S, the input data J is copied, transformed into J', and yielded back to the caller. Each J' can then be exported to disk, provided as a sample to an iterative learning algorithm, or passed along to another deformation object in a pipeline for further processing. When all subsequent processing of J' has completed, Algorithm 1 may resume computation at line 10 and proceed to the next state at line 2. Note that because deformation objects are both iterative (per track) and can be parallelized (across tracks), batches of deformed data can be generated online for stochastic learning algorithms.

3.1.1 Example: randomized time-stretching

To illustrate the deformation object interface, we will describe the implementation of a randomized *time-stretch* deformation object. In this case, each *state* object contains a single quantity: the stretch factor r. Algorithm 2 illustrates the state-generation logic for a randomized time-stretcher,

Algorithm 1 Abstract transformation pseudocode

Input: Deformation object D, JAMS object J
Output: Sequence of transformed JAMS objects J'

1: **function** D.TRANSFORM(J)
2: **for** states $S \in D$.STATES(J) **do**
3: $J' \leftarrow$ COPY(J)
4: J'.audio $\leftarrow D$.AUDIO(S, J'.audio)
5: J'.meta $\leftarrow D$.METADATA(S, J'.meta)
6: **for** transformations g in D **do**
7: **for** annotations $A \in J'$ which match g **do**
8: $J'.A \leftarrow g(S, A)$
9: J'.history $\leftarrow$ APPEND(J'.history, S)
10: **yield** J'

Algorithm 2 Randomized time-stretch state generator

Input: JAMS object J, number of deformations n, range bounds (r_-, r_+)
Output: Sequence of states S

1: **function** RANDOMSTRETCH.STATES($J, \{n, r_-, r_+\}$)
2: **for** i in $1, 2, \dots, n$ **do**
3: Sample $r \sim_U [r_-, r_+]$
4: **yield** $S = \{r\}$

in which some n examples are generated by sampling r uniformly at random from an interval $[r_-, r_+]$. [2]

The JAMS object J over which the deformations will be applied is also provided as input to the state generator. Though not used in this example, access to J allows the state generator to pre-compute quantities of interest, such as track duration — necessary to ensure well-defined outputs from target-smearing deformations — or tuning estimates, which are used by pitch-shift deformations to determine when a shift is large enough to alter note labels.

Once a state S has been generated, the AUDIO() deformation method — D.AUDIO(S, J.audio) — applies the time-stretch to the audio signal, which is stored within the JAMS sandbox upon instantiation. [3] Similarly, track-level meta-data can be modified by the METADATA() method. In this example, time-stretching will change the track duration, which is recorded in the JAMS meta-data field.

Next, a generic *annotation* deformation would be registered to the pattern "`.*`" and apply the stretch factor to all *time* and *duration* fields of all annotations. This deformation would leave the annotation *values* untouched, since not all annotation types have time-dependent values.

Finally, any annotations whose *value* fields depend on time, such as *tempo*, can be modified directly by registering the transformation function against the appropriate type pattern, *e.g.*, "`tempo`". Other time-dependent type deformations would be registered separately as needed.

The time-stretching example is simple, but it serves to illustrate the flexibility of the architecture. It is straightforward to extend this example into more sophisticated de-

[2] The parameters n, r_-, r_+ are actually properties of the deformation object, but are listed here as method parameters to simplify exposition.

[3] The *sandbox* provides unstructured storage space within a JAMS object, which is used in our framework as a scratch space for audio signals.

formations with structured state generators to sweep over deterministic parameter grids. For example, an additive background noise deformation could be parameterized by a collection of noise sources and a range of gain parameters, and generate one example for each unique combination of source and gain.

3.2 Pipelines and bypasses

Algorithm 1 describes the process by which a deformation object turns a single annotated audio example into a sequence of deformed examples. If we were interested in experimenting with only a single type of augmentation (*e.g.*, time stretching), this would suffice. However, some applications may require combining or cascading multiple types of deformation, and we prefer a unified interface that obviates the need for customized data augmentation scripts.

Here, we draw inspiration from scikit-learn in defining *pipeline* objects. The general idea is simple: two or more deformation objects D_i can be chained together, and treated as a single, integrated deformation object. More precisely, for a deformation pipeline P composed of k stages:

$$P = (D_1, D_2, \ldots, D_k),$$

examples are generated by a depth-first traversal of the Cartesian product of the corresponding state spaces Σ_i:

$$\Sigma_P = \Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_k.$$

One input example therefore produces $|\Sigma_P| = \prod_{i=1}^{k} |\Sigma_i|$ outputs. By using generators rather than explicit lists of states, we ensure that only $k + 1$ examples (counting the input) are ever in memory at any time. In most cases, k is much smaller than $|\Sigma_P|$, which provides substantial improvements to memory efficiency.

Finally, we introduce the *bypass* object, which is used to mark individual pipeline stages as optional. Bypasses are useful when it is difficult to encode a special *no transformation* state within a deformation object, such as in the randomized time-stretch example of Algorithm 2. The internal logic of a bypass object is simple: first, pass the input directly through unmodified, and then generate samples from the contained deformation object as usual. Bypasses can be used to ensure that the original examples are propagated through the pipeline unscathed, and the resulting augmented data set is a strict superset of the clean data.

3.3 Reproducibility and data provenance

When modifying data for statistical modeling purposes, maintaining transparency is of utmost importance to ensure reproducibility and accurate interpretation of results. This ultimately becomes a question of data provenance [5]: a record of all transformations should be kept, preferably attached as closely as possible to the data. Rather than force practitioners to handle book-keeping, we automate the process from within the deformation engine. This is accomplished at line 9 of Algorithm 1 by embedding the *state* object S (and, in practice, the parameters used to construct the deformation object D) within the JAMS object after each deformation is applied. Each J' generated at line 10 thus contains a full transactional history of all modifications required to transform J into J'. For this reason, stochastic deformations are designed so that all randomness is contained within the state generator, and transformations are all deterministic.

In addition to facilitating reproducibility, maintaining transformation provenance allows practitioners to compute a wide range of deformations, and later filter the results to derive subsets generated by different augmentation parameters.

To further facilitate reproducibility and sharing of experimental designs, the proposed architecture supports *serialization* of deformation objects and pipelines into a simple, human-readable JavaScript object notation (JSON) format. Once a pipeline has been constructed, it can be exported, edited as plain text, shared, and reconstructed. This feature also simplifies the process of applying several different sets of deformation parameters, and eliminates the need for writing a custom script for each setting.

Table 1. The 15 instrument labels used in our experiments.

Instrument	# Tracks	# Artists
drum set	65	57
electric bass	64	53
piano	42	23
male singer	38	34
clean electric guitar	37	32
vocalists	27	25
synthesizer	27	21
female singer	25	17
acoustic guitar	24	16
distorted electric guitar	21	20
auxiliary percussion	18	17
double bass	16	13
violin	14	5
cello	11	8
flute	11	6

4. EXAMPLE: INSTRUMENT RECOGNITION

We applied data augmentation to the task of instrument recognition in mixed audio signals. For this task, we used the MedleyDB dataset, which consists of 122 tracks, spanning a variety of genres and instrumentation [3]. Each track is strongly annotated with time-varying instrument activations derived from the recording stems. MedleyDB is a small, but well-annotated collection, which we selected because it should be possible to over-fit with a reasonably complex model. Our purpose here is not to achieve the best possible recognition results, but to investigate utility of data augmentation for improving generalization. However, because of the small sample size, we limited the experiment to cover only the 15 instruments listed in Table 1.

For evaluation purposes, each test track is split into disjoint one-second clips. The system is then tasked with recognizing the instruments active within each clip. The system is evaluated according to the average track-wise mean (label-ranking) average precision (LRAP), and per-

instrument F-score over one-second clips.

4.1 Data augmentation

The data augmentation pipeline consists of four stages:

Pitch shift by $n \in \{-1, 0, +1\}$ semitones.

Time stretch by a factor of $r \in \left\{2^{-1/2}, 1.0, 2^{1/2}\right\}$.

Background noise (bypass) under three conditions: subway,crowded concert hall,and night-time city noise. Noise clips were randomly sampled and linearly mixed with the input signal y using random weights $\alpha \sim_U [0.1, 0.4]$:

$$y' \leftarrow (1 - \alpha) \cdot y + \alpha \cdot y_{\text{noise}}.$$

Dynamic range compression (bypass) under two settings drawn from the Dolby E standards [7]: *speech*, and *music (standard)*.

Pitch-shift and time-stretch operations were implemented with Rubberband [1], and dynamic range compression was implemented using the *compand* function of sox [2]. Note that the first two stages include null parameter settings $n = 0$ and $r = 1$. Bypasses on the final two stages ensure that all combinations of augmentation are present in the final set. The full pipeline produces

$$|\Sigma_P| = 3 \times 3 \times (3 + 1) \times (2 + 1) = 108$$

variants of each input track. To simplify the experiments, we only compare the cumulative effects of the above augmentations. This results in five training conditions of increasing complexity:

- (N) no augmentation,
- (P) pitch shift,
- (PT) pitch shift and time stretch,
- (PTB) pitch shift, time stretch, and noise,
- (PTBC) all stages.

4.2 Acoustic model

The acoustic model used in these experiments is a deep convolutional network. The input to the network consists of log-amplitude, constant-Q spectrogram patches extracted with librosa [17]. Each example spans approximately one second of audio, corresponding to 44 frames at a hop length of 512 samples and sampling rate of 22050 Hz. Constant-Q spectrograms cover the range of C2 (65.41 Hz) to C8 (4186 Hz) at 36 bins per octave, resulting in time-frequency patches $X \in \mathbb{R}^{216 \times 44}$. Instrument activations are aggregated into a single binary label vector, such that an instrument is deemed active if its on-time within the sample exceeds 0.25 seconds.

Constant-Q representations are linear in both time and pitch, a property that can be exploited by convolutional neural networks to achieve translation invariance. Thus a four-layer model is designed to estimate the presence of zero or more instruments in a time-frequency patch. Formally, an input X, is transformed into an output Z, via a composite nonlinear function $\mathcal{F}(\cdot|\ \Theta)$ with parameters Θ. This is achieved as a sequential cascade of $L = 4$ operations, $f_\ell(\cdot|\ \theta_\ell)$, referred to as *layers*, the order of which is given by ℓ:

$$Z = \mathcal{F}(X|\ \Theta) = f_L(\cdots f_2(f_1(X|\ \theta_1)|\ \theta_2)|\ \theta_L) \quad (1)$$

The first two layers, $\ell \in \{1, 2\}$, are convolutional, expressed by the following:

$$Z_\ell = f_\ell(X_\ell|\ \theta_\ell) = h(W \circledast X_\ell + b), \quad \theta_\ell = [W, b] \quad (2)$$

Here, the valid convolution, $\circledast$, is computed by convolving a 3D input tensor, X_ℓ, consisting of N *feature maps*, with a collection of M 3D-*kernels*, W, followed by an additive vector bias term, b, and transformed by a point-wise activation function, $h(\cdot)$. In this formulation, X_ℓ has shape (N, d_0, d_1), W has shape (M, N, m_0, m_1), and the output, Z_ℓ, has shape $(M, d_0 - m_0 + 1, d_1 - m_1 + 1)$. Max-pooling is applied in time and frequency, to further accelerate computation by reducing the size of feature maps, and allowing a small degree of scale invariance in both time and pitch.

The final two layers, $\ell \in \{3, 4\}$, are fully-connected matrix products, given as follows:

$$Z_\ell = f_\ell(X_\ell|\ \theta_\ell) = h(WX_\ell + b), \quad \theta_\ell = [W, b] \quad (3)$$

The input to the ℓ^{th} layer, X_ℓ, is flattened to a column vector of length N, projected against a weight matrix W of shape (M, N), added to a vector bias term, b, of length M, and transformed by a point-wise activation function, $h(\cdot)$.

The network is parameterized thusly: ℓ_1 uses W with shape $(24, 1, 13, 9)$, followed by $(2, 2)$ max-pooling over the last two dimensions, and a rectified linear unit (ReLU) activation function: $h(x) := \max(x, 0)$; ℓ_2 has filter parameters W with shape $(48, 24, 9, 7)$, followed by $(2, 2)$ max-pooling over the last two dimensions, and a ReLU activation function; ℓ_3 uses W with shape $(17280, 96)$ and a ReLU activation function; finally, ℓ_4 uses W with shape $(96, 15)$ and a sigmoid activation function.

During training, the model optimizes cross-entropy loss via mini-batch stochastic gradient descent, using batches of $n = 64$ randomly selected patches and a constant learning rate of 0.01. Dropout is applied to the activations of the penultimate layer, $\ell = 3$ with dropout probability 0.5. Quadratic regularization is applied to the weights of the final layer, $\ell = 4$, with a penalty factor of 0.02. This helps prevent numerical instability by keeping the weights from growing arbitrarily large. The model is check-pointed after every 1000 batches (up to 50000 batches), and a validation set is used to select the parameter setting achieving the highest mean LRAP.

4.3 Evaluation

Fifteen random artist-conditional partitions of the MedleyDB collection were generated with a train/test artist ratio of 4:1. For the purposes of this experiment, *MusicDelta* tracks were separated by genre into a collection of distinct

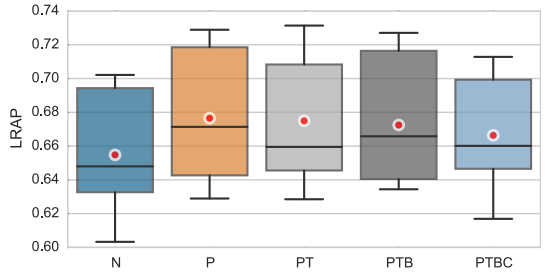

Figure 1. Test-set score distributions (mean track-wise label-ranking average precision), over all train-test splits. Mean scores are indicated by •. Boxes cover the 25–75 percentiles, and whiskers cover the 5–95 percentiles.

pseudo-artists. This results in 75 unique artist identifiers for the 122 tracks. For each train/test split, the training set was further partitioned into training and validation sets, again at a ratio of 4:1. To evaluate performance, we compute for each test track the mean label-ranking average precision (LRAP) over all disjoint one-second patches.

4.3.1 Label ranking results

Figure 1 illustrates the distribution of test-set performance across splits. Between the no-augmentation condition (N) and pitch-shifting augmentation (P), there is a small, but consistent improvement in performance from an average of 0.655 to 0.677. This is in keeping with the motivation for this work, and our expectations when training a (pitch)-convolutional model on a small sample. If the amount of clean data is too small, the model may easily over-fit by capturing irrelevant, correlated properties. (For example, if all of the *piano* recordings are in one key, the model may simply capture the key rather than the characteristics of *piano*.) Adding pitch-shifted examples should help the model disambiguate these properties.

Subsequent deformations do not appear to improve over condition (P). In each case, no significant difference from the pitch-shift condition could be detected by a Bonferroni-corrected Wilcoxon signed-rank test. However, all deformation conditions consistently outperform the baseline (N).

Although the difference in average performance is relatively small, the upper and lower quantiles are notably higher in (P), (PT), and (PTB) conditions. This indicates a reduction in the tendency to over-fit the relatively small training sets used in these experiments.

4.3.2 Frame-tagging results

To investigate the effects of augmentation on each instrument class, we computed the F-score of frame-level instrument recognition under each training condition. Results were averaged first across test tracks in a split, and then across all splits. Figure 2 depicts the change in F-score relative to the baseline condition (N): $\Delta F = (F - F_N)$.

The trend is primarily positive: in all but three classes, all augmentation conditions provide consistent improvement. The three exceptions are *synthesizer*, *female singer*, and *violin*. In the latter two cases, negative change is only observed after introducing time-stretch deformations, which

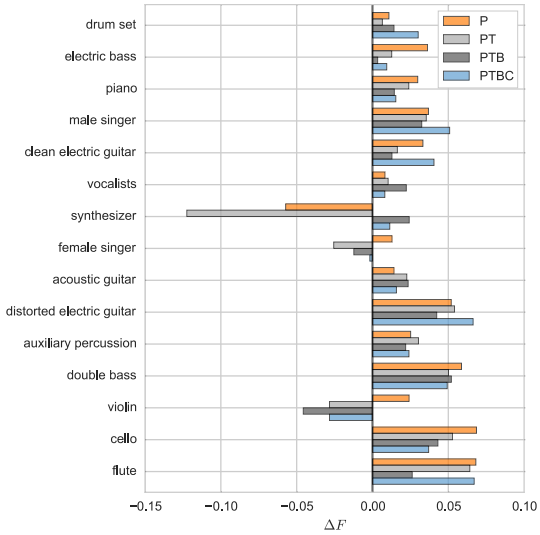

Figure 2. Per-class change in mean test-set F-score for each augmentation condition (F), relative to the no-augmentation baseline (F_N).

may unnaturally distort vibrato characteristics and render these classes more difficult to model. The effect is particularly prominent for *violin*, which has the fewest unique artists, and produces the fewest training examples.

The reduction in F-score for *synthesizer* in the (PT) condition may explain the corresponding reduction in Figure 1, and may be due to a confluence of factors. First, many of the synthesizer examples in MedleyDB have low amplitudes in the mix, and may be difficult to model in general. Second, the class itself may be ill-defined, as *synthesizer* encompasses a range of instruments and timbres which may be artist-dependent and idiosyncratic. Simple augmentations can have adverse effects if the perturbed examples are insufficiently varied from the originals, which may be the case here for (P) and (PT). However, the inclusion of background noise (PTB) results in a slight improvement over the baseline.

5. CONCLUSION

The data augmentation framework provides a simple and flexible interface to train models on distorted data. The instrument recognition experiment demonstrates that even simple deformations such as pitch-shifting can improve generalization, but that some care should be exercised when selecting deformations depending on the characteristics of the problem. We note that these results are preliminary, and do not fully exploit the capabilities of the augmentation framework. In future work, we will investigate the data augmentation for a variety of MIR tasks.

6. ACKNOWLEDGEMENTS

BM acknowledges support from the Moore-Sloan Data Science Environment at NYU. This material is partially based upon work supported by the National Science Foundation, under grant IIS-0844654.

7. REFERENCES

[1] Rubber band library v1.8.1, October 2012. http://rubberbandaudio.com/.

[2] sox v14.4.1, February 2013. http://sox.sourceforge.net/.

[3] Rachel Bittner, Justin Salamon, Mike Tierney, Matthias Mauch, Chris Cannam, and Bello, Juan Pablo. MedleyDB: a multitrack dataset for annotation-intensive mir research. In *15th International Society for Music Information Retrieval Conference*, ISMIR, 2014.

[4] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, et al. API design for machine learning software: experiences from the scikit-learn project. In *European Conference on Machine Learning and Principles and Practices of Knowledge Discovery in Databases*, 2013.

[5] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Data provenance: Some basic issues. In *FST TCS 2000: Foundations of software technology and theoretical computer science*, pages 87–93. Springer, 2000.

[6] Sander Dieleman, Philémon Brakel, and Benjamin Schrauwen. Audio-based music classification with a pretrained convolutional network. In *12th international society for music information retrieval conference*, ISMIR, 2011.

[7] Dolby Laboratories, Inc. *Standards and practices for authoring Dolby Digital and Dolby E bitstreams*, 2002.

[8] Daniel PW Ellis and Thierry Bertin-Mahieux. Large-scale cover song recognition using the 2d fourier transform magnitude. In *The 13th international society for music information retrieval conference*, ISMIR, 2012.

[9] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume*, volume 15, pages 315–323, 2011.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015.

[11] Eric J Humphrey, Juan Pablo Bello, and Yann LeCun. Moving beyond feature design: Deep architectures and automatic feature learning in music informatics. In *The 13th international society for music information retrieval conference*, ISMIR, 2012.

[12] Eric J Humphrey, Justin Salamon, Oriol Nieto, Jon Forsyth, Rachel M Bittner, and Bello, Juan Pablo. JAMS: A JSON annotated music specification for reproducible MIR research. In *15th International Society for Music Information Retrieval Conference*, ISMIR, 2014.

[13] Holger Kirchhoff, Simon Dixon, and Anssi Klapuri. Multi-template shift-variant non-negative matrix deconvolution for semi-automatic music transcription. In *The 13th international society for music information retrieval conference*, ISMIR, 2012.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, NIPS, pages 1097–1105, 2012.

[15] Kyogu Lee and Malcolm Slaney. Acoustic chord transcription and key extraction from audio using key-dependent HMMs trained on synthesized audio. *Audio, Speech, and Language Processing, IEEE Transactions on*, 16(2):291–301, 2008.

[16] Matthias Mauch and Sebastian Ewert. The audio degradation toolbox and its application to robustness evaluation. In *14th International Society for Music Information Retrieval Conference*, ISMIR, 2013.

[17] Brian McFee, Matt McVicar, Colin Raffel, Dawen Liang, Dan Ellis, Douglas Repetto, Petr Viktorin, and Joo Felipe Santos. librosa: 0.4.0rc1, March 2015.

[18] Meinard Müller and Sebastian Ewert. Chroma toolbox: Matlab implementations for extracting variants of chroma-based audio features. In *12th International Conference on Music Information Retrieval*, ISMIR, 2011.

[19] Elias Pampalk. A matlab toolbox to compute music similarity from audio. In *International Symposium on Music Information Retrieval (ISMIR2004)*, 2004.

[20] Bob L Sturm and Nick Collins. The Kiki-Bouba Challenge: Algorithmic composition for content-based MIR Research & Development. In *International Symposium on Music Information Retrieval*, 2014.

[21] Karen Ullrich, Jan Schlüter, and Thomas Grill. Boundary detection in music structure analysis using convolutional neural networks. In *15th International Society for Music Information Retrieval Conference*, ISMIR, 2014.

[22] Matthew D Zeiler, M Ranzato, Rajat Monga, M Mao, K Yang, Quoc Viet Le, Patrick Nguyen, A Senior, Vincent Vanhoucke, Jeffrey Dean, et al. On rectified linear units for speech processing. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 3517–3521. IEEE, 2013.