Optimizations for Automated Real-Time Music Improvisers

Pouneh Moghimi
Electrical Engineering and Computer Science
York University
Toronto, Canada
poonehmg@yorku.ca
Supervisor: Vassilios Tzerpos

***Abstract*—Real-time music improvisation stands as an innovative domain reshaping musical creativity and empowering spontaneous compositions through cutting-edge tools. The efficacy of deep learning models in this sphere relies significantly on optimizer selection, impacting the model's capacity to learn intricate musical patterns swiftly. Within this context, this study investigates the Mixture Density Recurrent Neural Network (MDRNN), a state-of-the-art model trained on temporal audio signal features, to evaluate the performance of a range of popular optimizers including SGD, Adam, Nadam, and their respective variants across different hyperparameters. Our investigation involves validating established theories on the influence of sharp and flat minimizers on model performance and assessing the reliability of lower validation loss as an indicator of better generalization. Moreover, we aim to demonstrate unexpected optimizer behaviours which are contrary to existing research findings. Through these comprehensive analyses, this study endeavours to offer valuable insights crucial for selecting optimizers in deep learning models tailored explicitly for real-time music improvisation.**

***Index Terms*—Automated Music Generation, Real-time Music Improviser, Machine Learning and Music Generation, Optimization for Deep Learning Models**

I. Introduction

Automating the process of generating music has appeared as an interesting subject for researchers in the field of computer science. There have been successful projects ranging from creating music using ready-made musical notes to training a machine in order to improvise music based on a specific style or a given scale. A common approach in many of these projects is the use of deep learning for generating music.

Most existing projects have the computer create music on its own, often learning from a large collection of existing music. These systems have shown impressive abilities, not only in mimicking music styles from their training data but also in creating new, enjoyable music.

However, one area that hasn't been explored as much is real-time music creation during live performances. In this scenario, a computer system would need to generate music that complements what another performer is playing, whether that's a human or another computer system. This is a significant challenge. The system must quickly analyze and understand the music played by the other performer and then produce music that harmonically and rhythmically matches the ongoing performance. This requires a deep understanding of music theory and the ability to improvise in a creative and responsive way.

The concept of an "Automated Real-Time Music Improviser" is particularly intriguing. Such a system could transform live music performances, allowing for a dynamic and interactive musical experience. It could lead to new forms of collaboration between human musicians and AI systems, enhancing live concerts, studio recordings, and music education. However, achieving this involves overcoming substantial challenges. The system must process the incoming music and generate a response with extremely low delay. It also needs to have a broad understanding of various musical styles and the ability to adapt its improvisations to the style and mood of the performance. This requires advanced algorithms, fast processing capabilities, and a comprehensive database of musical knowledge.

This project aims to explore the potential optimizations for real-time interactive music improvisation while deriving valuable insights from the comparison of various optimization methods. The research questions that this project intends to answer are:

1) How can we improve one of the most relevant research conducted on improvising music in real-time?
2) How can we optimize deep learning methods to fit our own dataset?
3) What interesting insights can be found when comparing different model optimization algorithms?
4) What could be a decent evaluation method for assessing the accuracy and relativity of our predicted music to the input music?

II. Background

In this section, we'll explore the existing body of related work and thoroughly acquaint ourselves with several common methods and algorithms extensively utilized in this field. This exploration aims to provide a comprehensive understanding of the techniques commonly employed in this domain.

A. Related Work

Artificial neural networks (ANNs) have ventured into the realm of directly composing musical pieces and generating

digital audio signals. Recurrent neural networks (RNNs) are a frequent choice for creating sequences of musical notes, where each prediction relies on the preceding note. An early example of this approach is Mozer's CONCERT system [1]. However, as stated in the paper, it lacked thematic structure and rhythmic organization. The reason behind that is the problem of vanishing gradients [2] in RNN, making it challenging for the model to deal with distant dependencies. The introduction of the long short-term memory (LSTM) cell [3] significantly enhanced these networks' capacity to capture long-term dependencies. Eck and Schmidhuber employed RNNs with LSTM cells to create blues music [4]. However, it is limited to symbolic music representations (sheet notes) and cannot handle real-time performances. These models showcase adaptability in understanding temporal sequences and are able to closely imitate cognitive capacities in humans for learning sequences and making predictions [5]. Some other music generation systems rely on Markov models [6], which compute emission probabilities for forthcoming notes based on preceding ones [7]. The superiority of RNN models over Markov systems lies in the latter's demand for excessively large transition tables to comprehend long-term dependencies within the data [1]. RNNs can produce more flexible predictions, interpolating among training instances rather than aiming for precise replication [8].

AI DJ Project 2 Ubiquitous Rhythm project is somehow close to the performance we are looking for. As described on Medium by Nao Tokui [9], this project is an improvisational DJ performance using AI to generate music in real-time. In this project, An AI composes music on the spot and another AI responds to that. The DJ project has the capability to autonomously generate and adapt music in a live setting showcasing the potential of AI in dynamic musical creation and performance. The DJ uses deep learning models such as Variational Autoencoder (VAE) and LSTM and trains them with a large number of MIDI files.VAE is used to encode complex rhythmic structures into low-dimensional vectors and reconstruct the original data by decoding these compressed vectors. Once the model is trained, the network becomes capable of producing multiple rhythm patterns by feeding the decoder with this low-dimensional vector.

Interactive Musical Prediction System (IMPS), the system made by Martin and Torresen [10], is a pioneer work in real-time music interaction focused on predicting musical control data rather than MIDI notes during live performances, a challenging task that demands both accuracy and speed. At the core of IMPS is the Mixture Density Recurrent Neural Network (MDRNN). This sophisticated model integrates two layers of Long Short-Term Memory (LSTM) units with a Mixture Density Network (MDN) layer. The LSTM units are adept at handling tasks that involve sequences, making them well-suited for musical applications where recognizing patterns over time is crucial. These units are designed to learn and remember long-term dependencies, enabling the system to predict future musical notes based on past sequences. The MDN layer complements this by providing the ability to forecast a range of possible outcomes, each with its own probability. This is particularly beneficial in music, where there are often several plausible notes or chords that could follow in a sequence. In IMPS, the MDRNN model takes in control inputs and predicts the next values and their timings. This predictive capability is vital for a system intended to interact and improvise with human musicians in a real-time setting. Another significant feature of the IMPS is its input and output interfaces, which facilitate communication with various musical interfaces using Open Sound Control (OSC) signals. OSC is a protocol commonly used for networking sound synthesizers, computers, and other multimedia devices in musical performances.

Building upon Martin and Torresen's work [10], our project aims to further refine the MDRNN model to enhance its predictive accuracy, which is essential for plausible real-time interaction. We are also expanding the model's ability to understand and interpret a broader range of musical inputs, including aspects like Spectral Centroid, Kurtosis, Rolloff, etc. to create a more expressive and engaging musical interaction. An often-used method to enhance a model's performance is by exploring different optimizers and fine-tuning the associated hyperparameters, as discussed further below.

B. Model Optimizers and Hyperparameters

Optimizers behave differently for different types of data sets which results in some of them being more suitable for a specific data set. Optimizers can be divided into two general categories: gradient descent optimizers and adaptive optimizers. Gradient descent optimizers include Batch gradient descent, Stochastic gradient descent and Mini-batch gradient descent while Adagrad, Adadelta, RMSprop, Adam, and Nadam are popular adaptive optimizers. This division is exclusively based on how the learning rate is tuned, i.e. manually in gradient descent algorithms whereas automatically and adaptively in adaptive algorithms [11].

Learning rate is one of the initial and crucial parameters in a model which requires your early consideration when embarking on the model-training journey. It scales the magnitude of weight updates and the steps your model takes to achieve the lowest loss possible. In other words, the learning rate determines the speed at which the network learns. However, it is highly important to choose a proper learning rate for our network since it can affect the learning process significantly or might even prevent the model from learning effectively.

If the learning rate is too small, the model requires a long time to converge especially when there are saddle points in the loss space. Conversely, if the learning rate is too high, it can jump over the global minimum and get farther and farther from it which is called divergence. Consequently, it could result in worse performance over time, exhibiting divergent behaviour rather than converging towards a solution. Fig. 1 [12] demonstrates how the learning rate can affect the training and learning process. In the loss vs. epoch plot, it is shown how loss values can change depending on the learning rate being used as we go forward in the training process.

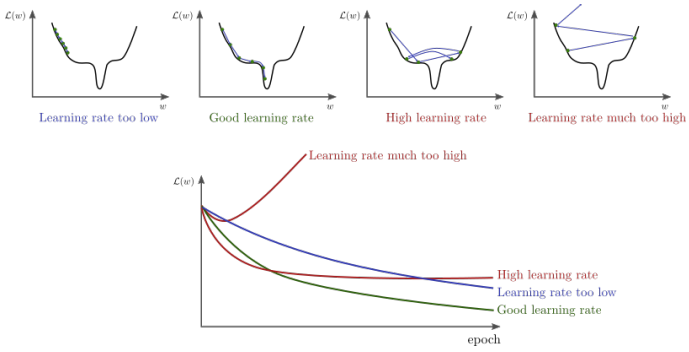

Fig. 1: Effect of learning rate on the learning process. w is the set of weights and $\mathcal{L}(w)$ is the loss associated with the set of weights.

Adaptive optimizers work well with sparse input data and provide better results in comparison to gradient descent optimizers in the case of a sparse dataset [13]. On the other hand, Wilson et al. [14] state that the solutions found by adaptive algorithms generalize worse (often significantly worse) than gradient descent algorithms while their convergence is faster and their training performance is often better than their gradient descent counterparts. The reason for the poor generalization of adaptive optimizers can be that they often converge to sharp minimizers especially when the batch size is increased [14], and according to Hochreiter and Schmidhuber [15], "sharp" minimizers generalize poorly while "flat" minimizers generalize well. Moreover, Wilson et al. [14] observe that even with small batches, Adam does not provide state-of-the-art test performance levels.

Among adaptive algorithms, Adam has shown the best results so far since it was introduced to improve previously used adaptive methods. It combines the advantages of Adagrad and RMSprop while trying to omit their issues to develop a more efficient algorithm [16].

The problem with Adagrad [17] lies in its learning rate adjustments, which are based on the cumulative gradients of all past steps. Consequently, the likelihood of having an extremely small learning rate after a significant number of steps due to the accumulation of past gradients is a notable concern because if the learning rate is too small, the weights are not updated properly and therefore the network does not learn anymore [11].

RMSprop [18] is an extension of Adagrad that deals with its vanishing learning rates by adding a decay factor but RMSprop does not incorporate a bias-correction term; This becomes particularly significant when dealing with a value of $\beta 2$(The exponential decay rate for the second-moment estimates) that is close to 1 which is required for handling sparse gradients. In such scenarios, the absence of bias correction results in overly large step sizes and leads to divergence [16]. Finally, Adam adds bias correction and momentum to RMSprop [13].

Adadelta [19] is very similar to RMSprop with the only difference being the way it computes the parameter updates. Adadelta uses the RMS(Root Mean Square) of parameter updates in the numerator of the update rule whereas RMSprop has the learning rate in the numerator. Zeiler [19] states that the numerator of the update rule in Adadelta is similar to momentum and "acts as an acceleration term".

So far, RMSprop, Adadelta, and Adam are quite similar algorithms and show good performances in similar situations. Kingma and Ba [16] demonstrate that Adam's incorporation of bias correction provides a slight advantage over RMSprop, especially as optimization progresses and gradients become sparser. Hence, Adam may be considered the top choice overall. An additional advantage is that it often obviates the need for learning rate manual tuning and it is likely to yield optimal results with its default value [13].

In 2016, Dozat [20] proposed a new optimization algorithm named "Nadam" which was claimed to be an improvement to Adam. Adam consists of two main components: a momentum component and an adaptive learning rate component. The Nesterov's accelerated gradient(NAG) algorithm has been proven both theoretically and empirically to outperform regular momentum. Dozat [20] attempted to enhance Adam's momentum component by incorporating insights from NAG and concluded that this substitution had a positive effect on the convergence speed as well as on training and validation losses.

Now let's get back to gradient descent optimizers which we talked about in the beginning. gradient descent optimizers fall into three categories, which vary in the extent to which they use data to calculate the gradient of the objective function.

Batch gradient descent, commonly referred to as vanilla gradient descent, is the most fundamental algorithm among the three. As shown in Eq. (1) [13], It calculates the gradients for the objective function J considering the parameters θ across the entire training set. Given that a single step involves processing the entire dataset, batch gradient descent can show a significant slowdown. Furthermore, it is impractical for datasets that exceed the available memory capacity.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) \tag{1}$$

Eq. (1) [13]: Equation to update weights in batch gradient descent. η is learning rate, θ is model weight set, and J is the objective function.

Stochastic Gradient Descent (SGD) [21] is an upgraded version of batch gradient descent. Instead of calculating gradients across the entire dataset, this approach updates weights for each example within the dataset. The equation (2) [13] now takes into account the values of input x and output y.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^i; y^i) \tag{2}$$

Eq. (2) [13]: Equation to update weights in Stochastic Gradient Descent (SGD). η is learning rate, θ is model weight set, and J is the objective function.

However, the challenge with Stochastic Gradient Descent (SGD) arises from frequent updates, resulting in significant

fluctuations in the objective function during training. While these fluctuations can offer an advantage over batch gradient descent, allowing the function to explore more favourable local minima, it also poses a drawback when it comes to converging to the global minimum. To tackle this issue, a solution is to gradually decrease the learning rate value. This adjustment causes the updates to get smaller over time, thereby averting excessive oscillations.

The underlying idea of the mini-batch gradient descent algorithm is to leverage the strengths of previous gradient descent methods we've explored thus far. Essentially, as demonstrated in Eq. (3) [13], it calculates gradients on small data batches to mitigate the variance of the updates.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{i:i+n}; y^{i:i+n}) \tag{3}$$

Eq. (3) [13]: Equation to update weights in mini-batch gradient descent. η is learning rate, θ is model weight set, and J is the objective function.

Qian et al. [22] in their experiment on the effect of batch sizes on the training and validation losses for the MNIST dataset, demonstrates that the training loss reduces when the batch sizes become smaller. However, this decline in the loss does not persist on the validation set, where the loss tends to stabilize or increase after a specific number of epochs. In other words, the model begins to exhibit signs of over-fitting. They posit that this behaviour occurs because the learned weights start oscillating around a local minimum during over-fitting. With a larger mini-batch size and smaller variance, the weights converge closer to the local minimum, consequently resulting in a smaller loss function value. Moreover, Masters and Luschi [23] state that overall, SGD batch sizes that provided the best performance were between m = 4 and m = 8 for the CIFAR-10 and CIFAR-100 datasets [24], and between m = 16 and m = 64 for the ImageNet dataset [25].

Stochastic Gradient Descent (SGD) encounters challenges when dealing with ravines, regions where the surface curvature differs significantly between dimensions and are often found near local optima. In such scenarios, SGD tends to oscillate along the ravine slopes, making gradual progress toward the local optima. Momentum [26], a method to enhance the speed of convergence, helps SGD accelerate optimization in the relevant direction while minimizing oscillations. This is achieved by incorporating a fraction (γ) of the update vector from the previous time step into the current update vector:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{4}$$

Eq. (4) [13]: Equation to update weights SGD with momentum. η is learning rate, θ is model weight set, and J is the objective function.

γ is the momentum term which can be from 0 to 1 and is typically set to 0.9. In essence, momentum mimics the motion of a ball rolling down a hill and escalating its speed as it descends. Similarly, parameter updates experience increased momentum for dimensions with consistent gradient directions and reduced updates for dimensions where gradients change their directions. This leads to quicker convergence and lessened oscillations [13].

III. Experimental Setup

The experimental setup is divided into two major phases, pre-training and post-training. Pre-training includes three primary steps which should be completed before starting to train our model. These steps are Data Collection, Data Preprocessing, and Model Hyperparameter Tuning. Once the model is trained, the focus shifts to the post-training phase involving two key steps to prepare the trained model for practical predictions, OSC Configuration and Graph Plotting Setup.

A. Data Collection

The project data is gathered from collaborating individuals utilizing a specialized instrument in the dispersion lab[1]. This instrument, driven by MAX/MSP software and physical controller devices, translates gestures and controller adjustments into sound outputs. Upon generating a specific audio output, the parameters linked to the utilized gestures and controllers involved in crafting the sound are saved into a JSON file format. This JSON file contains comprehensive details of the utilized devices, gestures, and audio signal data.

Within the JSON file, the "Analysis" section consists of parameter values that dynamically change over time across five main categories: Harmonic, Spectral, Perceptual, Signal, and Yin. These categories play crucial roles in music information retrieval, audio analysis, and the development of music recommendation systems, music transcription software, and other music-related applications in both research and industry contexts. Below, we will briefly introduce these categories:

Harmonic features typically refer to characteristics related to the harmonic content of a musical signal. They include features such as pitch, chords, intervals, and the relationship between musical notes in terms of frequency ratios. Harmonic features are often used in music analysis to understand tonality, chord progressions, and melodic structures [27].

Spectral features are derived from the frequency domain representation of audio signals. These features describe the distribution of energy across different frequency bands. Common spectral features include spectral centroid, spectral spread, spectral skewness, and spectral rolloff. They are valuable in analyzing timbre, distinguishing between different instruments, and characterizing sound textures [28].

Perceptual features are based on human perception of sound and music. They aim to capture qualities that are related to how humans perceive and interpret music. These features can include aspects like loudness, pitch perception, timbral brightness, and roughness. Perceptual features help

[1]https://dispersionlab.org/

in understanding how listeners perceive and interpret music, which is essential in music psychology and audio signal processing [29].

Signal features encompass a wide range of characteristics extracted directly from the audio signal itself. They include basic features such as amplitude, duration, tempo, and rhythmic patterns. Signal features are fundamental in music analysis, providing information about the structure, timing, and dynamics of musical pieces [30].

Yin is a specific algorithm used for pitch estimation in music and audio processing. Yin algorithm focuses on fundamental frequency (or pitch) estimation, particularly for monophonic and polyphonic music signals. It's commonly employed in pitch-tracking applications to estimate the pitch of musical notes [31].

Spectral features are particularly useful in music analysis due to their ability to capture and represent important aspects of sound that are crucial for understanding musical characteristics. Therefore, we decided to use spectral features to train our models in this project. In the JSON file, the Spectral category includes the following features:

- **Chroma:** The chromatic scale is a fundamental concept in music theory that relates to pitch and frequency. In the context of spectral analysis, the chromatic scale is related to the distribution of frequencies present in a musical signal. An audio file can contain 12 distinct pitch classes. These classes serve as highly advantageous tools for a comprehensive analysis of audio files. Pitch, a characteristic inherent in any sound or signal, allows categorizing files according to a scale related to frequency. It functions as a means to gauge sound quality and to evaluate whether the sound is perceived as higher, lower, or medium frequency [28].
- **Spectral Slope:** Spectral slope is an indicator of the 'slope' or inclination of the spectrum's shape. It's calculated by analyzing the amplitude spectrum and determining how the energy or intensity changes across different frequencies. A steeper slope indicates a more rapid decrease in amplitude as frequency increases, while a shallower slope signifies a slower decrease or even a relatively flat frequency response. This measure assists in distinguishing various voice characteristics like hissing, breathing, and typical speech patterns [32].
- **Spectral Decrease:** Spectral decrease refers to the reduction in the energy or amplitude of frequencies as they increase along the frequency spectrum. It indicates that the higher frequencies have lower energy compared to lower frequencies within a given audio signal or sound [33].
- **Spectral Variation:** Spectral variation or spectral flux typically refers to the variability or change in the spectral content of a signal over time or across different sections of the signal. It measures the differences in the frequency components or spectral shape between consecutive audio frames or segments. Spectral flux is calculated by comparing the spectral characteristics (such as the distribution of energy across different frequencies) of one frame or segment of an audio signal with the spectral characteristics of the previous frame or a reference frame. This comparison helps to identify how much the spectral content of the signal has changed between these frames and often relates to the perceived "roughness" in a sound [32].
- **Spectral Rolloff:** Spectral rolloff represents the frequency below which a predetermined percentage of the overall spectral energy, such as 85%, is found [34].
- **Spectral Kurtosis:** Spectral kurtosis is a statistical measure used in signal processing and spectral analysis to characterize the shape of the spectral distribution of a signal. It quantifies the "peakedness" or the degree of outliers in the spectral distribution relative to a Gaussian distribution. In simpler terms, kurtosis generally describes the shape of a probability distribution. Spectral kurtosis specifically focuses on how the energy in different frequency bins of a spectrum deviates from a Gaussian (normal) distribution. A higher spectral kurtosis value indicates that the energy in the spectrum is more "peaked" or concentrated, with significant spikes or outliers compared to a Gaussian distribution. This could suggest the presence of impulsive or non-Gaussian noise in the signal. Conversely, a lower spectral kurtosis value implies a more Gaussian-like distribution, where the energy in the spectrum is more evenly spread across frequencies without significant outliers or extreme values [35] [32].
- **Spectral Skewness:** Spectral skewness is a statistical measure used in signal processing and spectral analysis to assess the asymmetry or lack of symmetry in the distribution of spectral energy across different frequency bins within a signal's spectrum. Skewness measures the degree of asymmetry in a probability distribution. In the context of spectral analysis, spectral skewness quantifies the shape of the spectral distribution and determines whether it is symmetric or skewed towards one side. A positive spectral skewness value indicates that the distribution of energy in the spectrum is skewed to the right of the mean, implying a heavier tail on the right side of the distribution. Conversely, a negative spectral skewness value indicates a left-skewed distribution, with a heavier tail on the left side. If spectral skewness is zero, it means the distribution is symmetric about the mean [32].
- **Spectral Spread:** Spectral spread characterizes how widely or narrowly the energy or power of the signal is distributed across different frequencies. A high spectral spread indicates that the energy is distributed over a broad range of frequencies, suggesting a wide variety of frequency components within the signal. This could correspond to a signal with diverse harmonic content or multiple frequency bands (noisy) contributing to the overall sound. Conversely, a low spectral spread implies that the energy is concentrated within a narrower range of frequencies (pitched sound), indicating a more focused

or concentrated spectral content with fewer frequency components contributing to the signal [32].

- **Spectral Centroid:** Spectral centroid represents the average or central frequency around which most of the spectral energy of a signal is concentrated. If the spectral centroid is higher, it indicates that the energy is concentrated towards higher frequencies, whereas a lower spectral centroid suggests more energy at lower frequencies [34].
- **Time:** It indicates the precise timing for each measurement of the previously outlined features. For every individual timestamp, there exists a set of values corresponding to the defined features.

B. Data Pre-processing

We performed pre-processing on the data before inputting it into our model. The dataset required a specific format to suit the model's processing needs. Unlike the JSON file, where each time value demonstrated the time difference between the current time and the start time in milliseconds, our data needed to display precise timestamps, including the date, hour, minutes, and seconds for each time step. Additionally, each timestamp should have been accompanied by a set of values representing the spectral features at that exact moment. However, the JSON file's format differed in the way that each spectral feature had its own separate list of values. Furthermore, the model's structure assumes that the first dimension of the data corresponds to time and thus should be positive and non-zero. The other dimensions are expected to fall within the range of 0 to 1. Consequently, we had to rescale our feature values to fit within this range, leading us to normalize the values. Normalization involves transforming features to a standardized scale, which ultimately enhances the model's performance and training stability. To normalize our values, we used the scaling to a range technique (5) where $x_{\max}$ and $x_{\min}$ are the maximum and minimum values of a specific feature respectively, for example, Spectral Centroid.

$$x' = (x - x_{\min})/x_{\max} - x_{\min} \tag{5}$$

Eq. (5): Equation to normalize the value of x of a specific feature with the scaling to a range technique.

C. Model Hyperparameter Tuning

The model introduced by Martin and Torresen [10] employed the Adam optimizer with the default learning rate, i.e., 0.001. We aim to explore and compare various optimizers to identify the most effective ones for our dataset while gaining valuable insights into their performances.

After conducting extensive background research on various optimizers, it's now time to select several of them for comparison. Our selection encompasses optimizers from both the gradient descent and adaptive optimizer categories to conduct a comprehensive investigation into their efficiency. Specifically, we opted for Adam and Nadam from the adaptive optimizers category. Adam has demonstrated superior performance theoretically and practically among previous adaptive optimizers, while Nadam, introduced after Adam, claims to further enhance performance over Adam. Our objective is to test these optimizers across different learning rates to observe their predictive capabilities. For Adam and Nadam, we aim to compare their performances utilizing their default learning rates with their performances at lower learning rates.

Reflecting on the categories of optimizers, we've opted for mini-batch gradient descent (referred to as SGD here) among the gradient descent optimizers and combined it with momentum to harness the strengths of both techniques. We fine-tune four parameters for this optimizer: learning rate, learning rate decay, momentum, and Nesterov. Since this method isn't adaptive, manual tuning of learning rate decay is necessary, especially for higher learning rates. We use the standard time-based learning rate scheduler in Keras. The learning rate decay value is the rate at which the initial learning rate decreases after every batch within an epoch. In our experiments, we intend to conduct a few analyses. One objective is to compare the performance of SGD along with learning rate decay and momentum at its default learning against its performance at lower learning rates. Additionally, we aim to assess the impact of Nesterov momentum on the SGD optimizer's performance compared to using regular momentum. Furthermore, we seek to evaluate the differences in performance between vanilla SGD (SGD without learning rate decay and momentum) and our adaptive optimizers (Adam and Nadam) to investigate the validity of the theory suggesting that flat minimizers provide better generalizations than sharp minimizers.

Regarding other hyperparameters of the model, we used IMPS model size "s" containing 64 MDRNN units for this project based on recommendations from their paper, which aligns with the scale of our dataset consisting of approximately 80,000 samples. For training purposes, our batch size has been set to 64.

D. OSC Configuration

After the training is done, it's time for prediction. To initiate the prediction process, we must provide the model with initial input to enable it to proceed with subsequent predictions. We employ OSC (Open Sound Control) to establish communication between our instrument, located within our MAX/MSP patch, and the model. The concept involves transmitting Spectral data from the instrument to the model via OSC. The model initiates its prediction once a configurable brief silence ensues, indicating the end of data reception. Currently, we're utilizing random input passed through OSC to prompt the model, serving as a placeholder for the actual inputs. This method allows us to trigger the sequential feeding of the actual inputs to the model within the code. This adjustment is necessitated by the unavailability of access to the actual instrument at the moment.

Optimizer	Learning Rate	Learning Rate Decay	Momentum	Nesterov	Minimum Validation Loss	At Epoch
Adam	Default (0.001)	-	-	-	-62.03	29
Adam	0.00005	-	-	-	-80.77	21
Nadam	Default (0.001)	-	-	-	-51.30	23
Nadam	0.00005	-	-	-	-79.06	29
SGD	Default (0.01)	0.01/130	0.8	False	25.73	6
SGD	Default (0.01)	0.01/130	0.8	True	-10.84	73
SGD	0.0005	0.0005/130	0.8	False	-25.03	26
SGD	0.0005	0.0005/130	0.8	True	-36.73	119
SGD	0.0005	0.0005/130	0.9	False	-14.38	41
SGD	0.00005	0	0	False	-30.69	128

TABLE I: Optimization configurations and their corresponding minimum validation losses achieved along with the respective number of epochs where the minimum validation loss occurred.

E. Prediction Output and Graph Plotting Setup

We require the predictions to be formatted in a way that is readable for the instrument. Hence, we gather these prediction outputs and compile them into a JSON file, substituting them for the previous spectral values. This formatted file will be used later to feed the instrument and generate sound. Before writing into the JSON file, several value transformations are necessary. Firstly, time must be converted from seconds to milliseconds. Additionally, since the model outputs the time difference between the current and previous prediction, this time gap needs to be adjusted to show the duration that has passed since the start of the prediction, aligning it with the format of the input JSON files. Lastly, the normalized values need to be reverted to their original, non-normalized state.

For the purpose of graph plotting, we begin by plotting the input data provided to initiate predictions, followed by the prediction data immediately afterward. This approach enables us to visualize the comparison between the prediction data and the input data used by the model. Separate graphs are plotted for distinct spectral features, with each plot showcasing the progression of these features over time.

IV. Experimental Results

In this section, we aim to showcase the outcomes obtained from employing different optimizers along with varying hyperparameters, as outlined in the experimental setup. The demonstration consists of three primary components. Firstly, we will present the specific hyperparameters utilized for each optimizer and showcase their respective minimum validation losses obtained during the training process. Secondly, we will illustrate the graphical representation of spectral features over time in the test file. Due to the limitation of space, we've opted to showcase four spectral features in this report: Chroma03, Chroma04, Chroma05, and Spectral Skewness. Thirdly, we'll present graphs depicting the test input values alongside prediction values for these four spectral features across each method over time. These graphical representations provide a visual comparison between the test input values (ranging from time 0 to 917 milliseconds) and the subsequent prediction values.

In the end, we'll delve into an analysis aimed at understanding the observed experimental outcomes and shed light on the insights gained from these experiments. This analysis will explore potential reasons behind the results obtained.

Loss in IMPS is derived from the logarithm of probability density function (PDF) of the mixture model. Validation loss is determined using a set of examples—10% of the total dataset—that is kept aside and not used for training. This loss is evaluated after each training epoch and serves as a measure of the model's performance on unseen data. Table I shows the minimum validation loss each method managed to achieve within 130 epochs with the early stopping patience set to 20 epochs. Figure 2 shows the values of four spectral features over time in the test file.

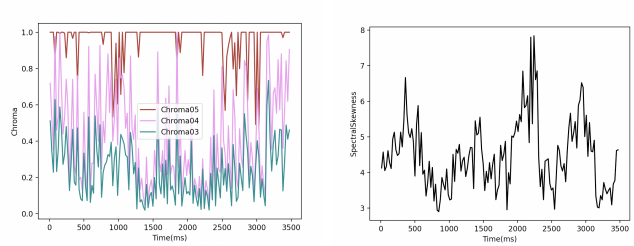

Fig. 2: Values of four spectral features over time in the test file: Chroma03, Chroma04, Chroma05, and Spectral Skewness

The following results can be inferred from the graphs:

- SGD with a learning rate of 0.0005 along with learning rate decay and momentum set to 0.8, figure 9, exhibits the best results among all methods. However, when run with Nesterov momentum, figure 10, this configuration demonstrates one of the poorest performances.
- The best-performing SGD configuration exhibits poor and unstable performance when the momentum is changed to 0.9, figure 11.
- Adam with a learning rate of 0.00005, figure 4 demonstrates better overall generalization compared to Adam with the default learning rate, figure 3, particularly for Spectral Skewness.
- Nadam's performance appears quite similar to Adam's. However, Adam with the default learning rate of 0.001

seems to perform slightly better than Nadam with the same learning rate, figure 5.

- Vanilla SGD (SGD without learning rate decay and momentum) with a learning rate of 0.00005, figure 12, shows poorer performance compared to Adam and Nadam at the same learning rate.
- The poorest performances are observed in SGD with a default learning rate of 0.01 alongside learning rate decay and momentum set to 0.8, figure 7, SGD with a default learning rate of 0.01 alongside learning rate decay and Nesterov momentum set to 0.8, figure 8, SGD with a learning rate of 0.0005 alongside learning rate decay and Nesterov momentum set to 0.8, figure 10, and SGD with a learning rate of 0.0005 alongside learning rate decay and momentum set to 0.9, figure 11. These performances lack correct time differences between predictions and appear more densely packed than others.

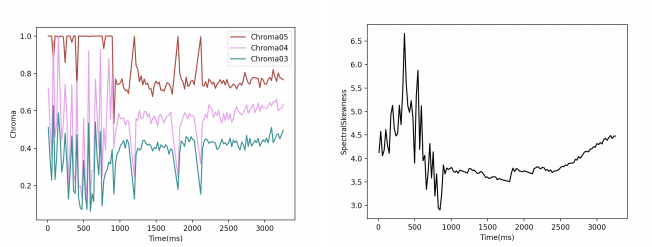

Fig. 3: Mix of input values (0 to 917ms) and predicted values for four spectral features over time achieved by Adam, lr=default(0.001)

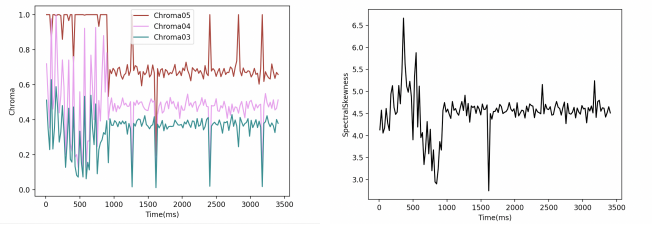

Fig. 4: Mix of input values (0 to 917ms) and predicted values for four spectral features over time achieved by Adam, lr=0.00005

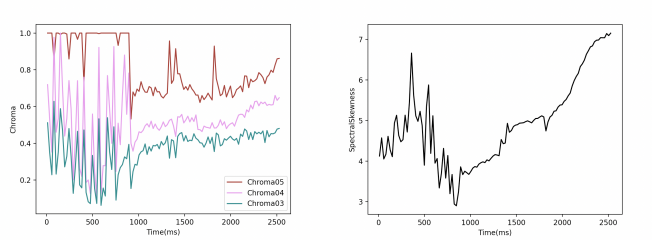

Fig. 5: Mix of input values (0 to 917ms) and predicted values for four spectral features over time achieved by Nadam, lr=default(0.001)

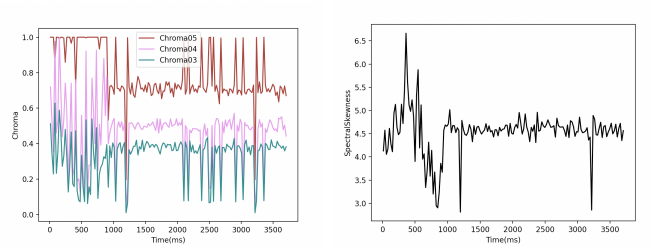

Fig. 6: Mix of input values (0 to 917ms) and predicted values for four spectral features over time achieved by Nadam, lr=0.00005

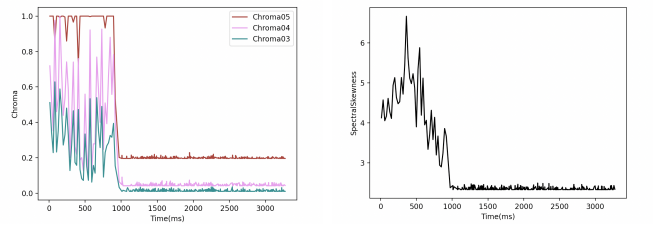

Fig. 7: Mix of input values (0 to 917ms) and predicted values for four spectral features over time achieved by SGD, lr=default(0.01), decay=0.01/130, momentum=0.8, nesterov=False

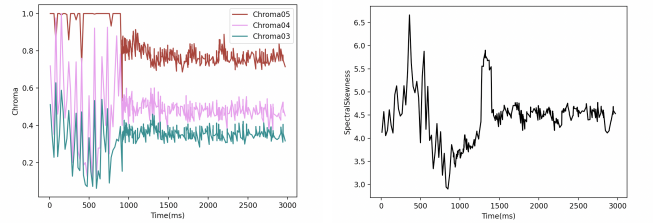

Fig. 8: Mix of input values (0 to 917ms) and predicted values for four spectral features over time achieved by SGD, lr=default(0.01), decay=0.01/130, momentum=0.8, nesterov=Ture

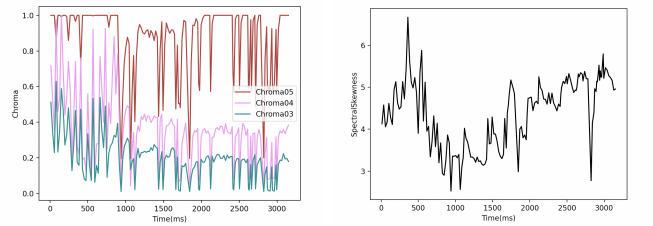

Fig. 9: Mix of input values (0 to 917ms) and predicted values for four spectral features over time achieved by SGD, lr=0.0005, decay=0.0005/130, momentum=0.8, nesterov=False

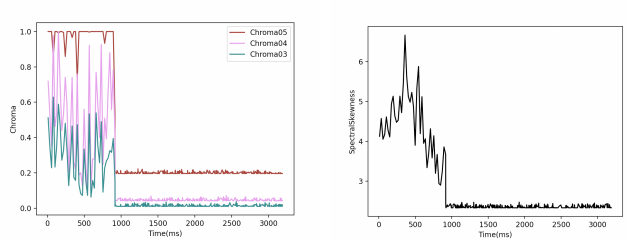

Fig. 10: Mix of input values (0 to 917ms) and predicted values for four spectral features over time achieved by SGD, lr=0.0005, decay=0.0005/130, momentum=0.8, nesterov=Ture

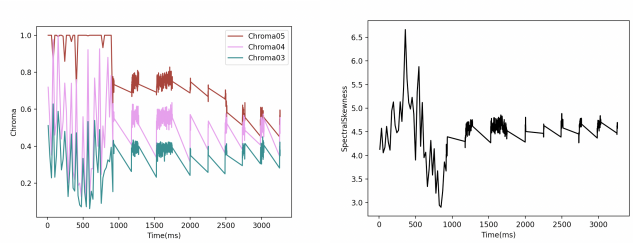

Fig. 11: Mix of input values (0 to 917ms) and predicted values for four spectral features over time achieved by SGD, lr=0.0005, decay=0.0005/130, momentum=0.9, nesterov=False

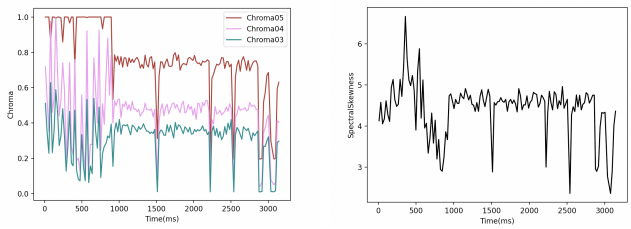

Fig. 12: Mix of input values (0 to 917ms) and predicted values for four spectral features over time achieved by SGD, lr=0.00005, decay=0, momentum=0, nesterov=False

As highlighted in the background section, adaptive optimizers often converge towards sharp minimizers, leading to poor generalizations. Despite the superior performance observed with SGD incorporating decay and momentum, a comparison reveals that Adam and Nadam exhibit superior generalizations at lower learning rates compared to vanilla SGD. It is also evident from the results that applying Nesterov to SGD does not result in acceptable generalizations and often lacks the ability to learn the time differences correctly. Additionally, SGD with the default learning rate performs poorly as the learning rate is probably too high to start with, even when learning rate decay is employed.

When correlating each method's prediction performance with their respective validation losses, it may appear confusing that the method with the best generalization does not possess the lowest validation loss. This complexity often pertains to the distinction between sharp and flat minimizers as previously discussed. Moreover, the lowest validation loss does not always signify the best model or accurate predictions. Instances exist where models overfit the validation data, necessitating caution in solely relying on low validation losses as indicators of credible generalization [36].

V. Conclusion and Future Work

This study delved into the examination of several prominent optimizers concerning their performance on a specific deep learning model, MDRNN, trained using a temporal dataset of audio signal features. The attained results presented a mix of findings, some aligning with the established research detailed in the background section while others presented contrary outcomes.

One instance supporting prior research is the significance of sharp and flat minimizers in achieving optimal performance, evident in scenarios where minimal loss did not necessarily translate to superior performance such as the best performance of SGD compared to Adam and Nadam. However, an opposing observation contradicted this hypothesis, as the optimizer known to converge to flat minima (SGD) exhibited poorer generalization in lower learning rates compared to adaptive optimizers (Adam and Nadam) which are known to converge to sharp minimizers. Another intriguing observation supporting prior research was the inconsistency between lower validation loss and improved generalization. This finding challenged the conventional belief that lower validation loss always translates to superior performance. Finally, the last contrasting observation emerged from the instability and inaccuracy of results when applying Nesterov momentum to SGD. Despite research claims suggesting Nesterov momentum's improvement over regular momentum, this study revealed unstable outcomes under this setting. Moreover, Nadam did not exhibit substantial improvement over Adam, contrary to expectations.

In future research, emphasis could be placed on enhancing the evaluation methodology for better accuracy assessment. For instance, evaluations could rely on quantifiable metrics comparing the accuracy of the results against the original test files instead of subjective evaluations. This accuracy could be presented as a percentage reflecting the resemblance between the obtained results and the original data. The ideal model should generate predictions that bear a noticeable resemblance to the original data while retaining a level of uniqueness, avoiding being entirely identical yet not diverging significantly. Moreover, integrating this framework into the actual instrument and comparing the predicted audio to the input audio would offer a more comprehensive understanding of the model's efficacy.

References

[1] M. C. Mozer, "Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multi-scale processing," *Connection Science*, vol. 6, no. 2-3, pp. 247–280, 1994.

[2] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber *et al.*, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies," 2001.

[3] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[4] D. Eck and J. Schmidhuber, "Finding temporal structure in music: Blues improvisation with lstm recurrent networks," in *Proceedings of the 12th IEEE workshop on neural networks for signal processing*. IEEE, 2002, pp. 747–756.

[5] C. P. Martin, K. O. Ellefsen, and J. Torresen, "Deep predictive models in interactive music," *arXiv preprint arXiv:1801.10492*, 2018.

[6] C. Ames, "The markov process as a compositional model: A survey and tutorial," *Leonardo*, vol. 22, no. 2, pp. 175–187, 1989.

[7] S. Dubnov, G. Assayag, O. Lartillot, and G. Bejerano, "Using machine-learning methods for musical style modeling," *Computer*, vol. 36, no. 10, pp. 73–80, 2003.

[8] A. Graves, "Generating sequences with recurrent neural networks," *arXiv preprint arXiv:1308.0850*, 2013.

[9] N. Tokui, "Ai dj project2 ubiquitous rhythm," *Medium*, Year of Publication. [Online]. Available: URLoftheMediumArticle

[10] C. P. Martin and J. Torresen, "An interactive musical prediction system with mixture density recurrent neural networks," *arXiv preprint arXiv:1904.05009*, 2019. [Online]. Available: https://arxiv.org/abs/1904.05009

[11] D. Giordano. (2020) 7 tips to choose the best optimizer. [Online]. Available: https://towardsdatascience.com/7-tips-to-choose-the-best-optimizer-47bb9c1219e

[12] B. D. Hammel. (2019) What learning rate should i use? [Online]. Available: http://www.bdhammel.com/learning-rates

[13] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[14] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, "The marginal value of adaptive gradient methods in machine learning," *Advances in neural information processing systems*, vol. 30, 2017.

[15] S. Hochreiter and J. Schmidhuber, "Flat minima," *Neural computation*, vol. 9, no. 1, pp. 1–42, 1997.

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[17] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of machine learning research*, vol. 12, no. 7, 2011.

[18] T. Tieleman, G. Hinton *et al.*, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[19] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[20] T. Dozat, "Incorporating nesterov momentum into adam," 2016.

[21] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *SIAM review*, vol. 60, no. 2, pp. 223–311, 2018.

[22] X. Qian and D. Klabjan, "The impact of the mini-batch size on the variance of gradients in stochastic gradient descent," *arXiv preprint arXiv:2004.13146*, 2020.

[23] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *arXiv preprint arXiv:1804.07612*, 2018.

[24] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[26] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

[27] Y. Mo, "Music timbre extracted from audio signal features," *Mobile Information Systems*, vol. 2022, 2022.

[28] Y. VERMA. (2021) A tutorial on spectral feature extraction for audio analytics. [Online]. Available: https://analyticsindiamag.com/a-tutorial-on-spectral-feature-extraction-for-audio-analytics/#:~:text=,useful%20in%20audio%20data%20learning

[29] F. Alías, J. C. Socoró, and X. Sevillano, "A review of physical and perceptual feature extraction techniques for speech, music and environmental sounds," *Applied Sciences*, vol. 6, no. 5, p. 143, 2016.

[30] N. Agrawal. (2023) Decoding the symphony of sound: Audio signal processing for musical engineering. [Online]. Available: https://towardsdatascience.com/decoding-the-symphony-of-sound-audio-signal-processing-for-musical-engineering-c66f09a4d0f5

[31] A. De Cheveigné and H. Kawahara, "Yin, a fundamental frequency estimator for speech and music," *The Journal of the Acoustical Society of America*, vol. 111, no. 4, pp. 1917–1930, 2002.

[32] M. core team. (2023) Meyda,audio feature extraction for javascript. [Online]. Available: https://meyda.js.org/audio-features.html

[33] G. Peeters, B. L. Giordano, P. Susini, N. Misdariis, and S. McAdams, "The timbre toolbox: Extracting audio descriptors from musical signals," *The Journal of the Acoustical Society of America*, vol. 130, no. 5, pp. 2902–2916, 2011.

[34] I. R. Roman. (2022) Spectral features. [Online]. Available: https://colab.research.google.com/github/iranroman/musicinformationretrieval.com/blob/gh-pages/spectral_features.ipynb#scrollTo=1BKPggA3dbje

[35] J. Antoni, "The spectral kurtosis: a useful tool for characterising non-stationary signals," *Mechanical Systems and Signal Processing*, vol. 20, no. 2, pp. 282–307, 2006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0888327004001517

[36] A. Y. Ng *et al.*, "Preventing" overfitting" of cross-validation data," in *ICML*, vol. 97. Citeseer, 1997, pp. 245–253.