

5-30-2018

Generating Audio Using Recurrent Neural Networks

Andrew Pfalz

Louisiana State University and Agricultural and Mechanical College, apfalz1@lsu.edu

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations



Part of the [Audio Arts and Acoustics Commons](#), [Composition Commons](#), and the [Other Music Commons](#)

Recommended Citation

Pfalz, Andrew, "Generating Audio Using Recurrent Neural Networks" (2018). *LSU Doctoral Dissertations*. 4601.
https://digitalcommons.lsu.edu/gradschool_dissertations/4601

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.

GENERATING AUDIO USING RECURRENT NEURAL NETWORKS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

Experimental Music and Digital Media

by

Andrew Pfalz

B.M., Florida State University, 2011

M.M., East Carolina University, 2014

August 2018

Acknowledgments

I would like to thank the following people for their support: Edgar Berdahl, Jesse Allison, Stephen Beck, Derick Ostrenko, Marc Aubanel, Richard Raether, Brandon Wheeler, Tim Wright, and Rebecca Fiebrink. I would also like to thank the Center for Computation and Technology for their support and enthusiasm.

Table of Contents

ACKNOWLEDGMENTS	ii
ABSTRACT	v
CHAPTER	
1 INTRODUCTION	1
1.1 Introduction	1
1.2 Sampling	11
1.3 Related Work	11
1.4 Contributions of This Work	14
2 TRAINING	18
2.1 Training Algorithm	18
2.2 Hyperparameters	21
3 SAMPLING	25
3.1 Overview	25
3.2 Vocoder Design	28
3.3 Sampling in Series or Concurrently	29
4 HYPERPARAMETERS	32
4.1 Overview	32
4.2 Number of Epochs	32
4.3 Number of LSTM Layers	32
4.4 Weight Initialization	34
4.5 Residual Connections	34
4.6 Weight Regularization	34
4.7 Dropout	35
4.8 Choosing Hyperparameter Values	35
5 RESULTS	37
5.1 Vector Approach	37
6 APPLICATIONS	48
6.1 User Interface	48
6.2 Compositions	49
6.3 Summary	54
7 CONCLUSIONS	56
7.1 Approaches to Sampling	56
7.2 Sequential Versus Concurrent Predictions	58
7.3 Changing Significant Hyperparameters	59
7.4 Musical Applications	59

REFERENCES.....	61
VITA	65

Abstract

Long Short Term Memory cells are a type of recurrent neural network that perform well when predicting sequence data. This work presents four approaches to modeling audio data. The models are trained to predict either raw audio samples or magnitude spectrum windows based on prior input audio. In a process called sampling, the models can then be employed to generate new audio using what they learned about the data they were trained on.

Four methods for sampling are presented. The first has the model predict a vector for each vector in the input. The second has the model predict one magnitude spectrum window for each magnitude spectrum window in the input. The third has the model predict a single audio sample for each input vector. The fourth has the model predict a single sample for the entire input matrix.

The end result is a method by which novel audio can be generated. This method differs from other synthesis techniques in that the content of the generated audio is not easily foreseeable. The various approaches to sampling afford different levels of control over the output. The biggest drawback to creating audio using this method is that the process is often too slow to run in real time.

Finally two applications are presented that allow the user to easily create and launch jobs. Four musical compositions are presented that use the sounds created using predictions from the models. The process for generating the source materials for each piece and the manipulations that were made to them after they were generated are also described.

Chapter 1

Introduction

1.1 Introduction

Developments in artificial intelligence over the past decade have pervaded many aspects of daily life. Examples include self-driving cars [1], Alpha Go [2], IBM Watson’s winning at Jeopardy [3], mail sorting [4] and voice assistant technology [5]. This technology has also reached less utilitarian aspects of many peoples’ experience: video games [6] and Google’s Deep Dream [7]. Some visual artists have also begun to incorporate these technologies into their work [8]. Researchers have begun to investigate topics that do not have clear academic or marketable applications: Neural Style Transfer [9], Creative Adversarial Networks [10].

1.1.1 Terminology

The term artificial intelligence (AI) is somewhat problematic because it is very broad. It encompasses a very diverse group of technologies. Some are very sophisticated and show better than human performance on simple tasks such as image recognition [11]. AI technologies also encompass much simpler techniques that estimate a function that approximates data such as polynomial regression. This technique has its roots in the nineteenth century with the work of Adrien-Marie Legendre who described it abstractly. Carl Friedrich Gauss later used the algorithm to estimate the orbits of planets [12]. Many classic video games are said to use some simple AI. The technology there is often some type of finite-state machine, in which the programmer defines a set of states and rules for transitioning between states. The application receives some input and updates its state according to its rules. Figure 1.1 illustrates the relationships between many of the relevant AI topics.

Machine learning encompasses techniques where a model is iteratively trained to perform a task. There are two main approaches to machine learning: supervised and unsupervised learning. In unsupervised learning, the model looks at the dataset and iteratively tries to perform some task, measures its success, makes adjustments and tries again until it reaches a desired level of performance. Examples include *clustering* where data is seg-

mented into groups via some machine learning algorithm, and *anomaly detection* where aberrant entries in a dataset are found automatically. Supervised learning involves a similar process whereby a model is iteratively shown some input data about which it makes a guess, the guess is compared to the known correct answer, and adjustments are made until the model performs reasonably well.

The term artificial intelligence is also problematic in the sense that it personifies computer programs in a way that is somewhat misleading. While it is true that some models have performed better than humans on image recognition tasks, it is not true to say that the model knows anything about the world or the content of the images it is recognizing. These same models can sometimes be tricked by altering a single pixel in an input image [13]. It is easy to extrapolate beyond the current state of the art with AI and stoke fears about the technology [14] [15]. In the past there have been eras when AI is considered unsuitable for academic research. They are known as “AI winters”. When too much fear is spread about AI safety there is a risk of bringing on another such era[16].

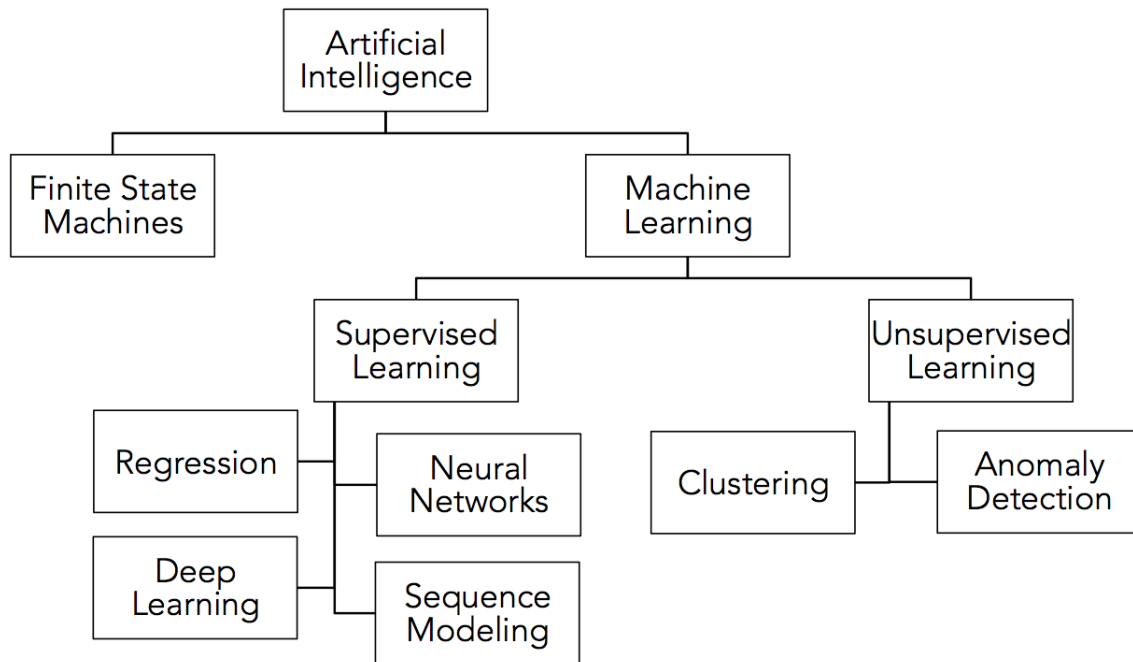


Figure 1.1. Artificial Intelligence and related terminology.

1.1.2 Neural Networks

Neural network is another term that still receives wide usage despite its problematic connotations. The term originates in a 1943 paper by Walter Pitts and Warren McCulloch [17]. In it they present a mathematical model for how a neuron might work. The so-called McCulloch-Pitts neuron computes a weighted sum of its inputs and passes the result through a threshold function as (1.1) where ϕ is a activation function and n is number of inputs to the neuron, x is the input vector, w is the weight vector, and b is a bias term that allows the neuron to shift the threshold value.

The interpretation is that the neuron receives inputs and gives an output that is something like the activation of biological neuron. Figure 1.2 shows this more clearly. The inputs are weighted so that some inputs influence the ultimate output of the neuron more than others. An artificial neuron can be thus used to make simple decisions about its inputs. For example the inputs to the neuron might be some values representing some features of a house and the output of the neuron could be a prediction of whether or not the house will sell for more than a million dollars. The prediction must be a binary value because of the activation function. The input vector might include values like the number of rooms in the house, the number of floors, or whether or not it has a pool. The neuron would weight each of these factors separately to produce a final output \hat{y} which is interpreted as the predicted house price.

$$\hat{y} = \phi\left(\sum_{i=0}^n w_i x_i + b\right) \quad (1.1)$$

$$\phi = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases} \quad (1.2)$$

Neural networks are composed of a series of artificial neurons arranged in layers such that the outputs of one layer of neurons are the inputs of the next layer. Figure 1.4 shows a small neural network with four layers. The first and last layers of the model are called

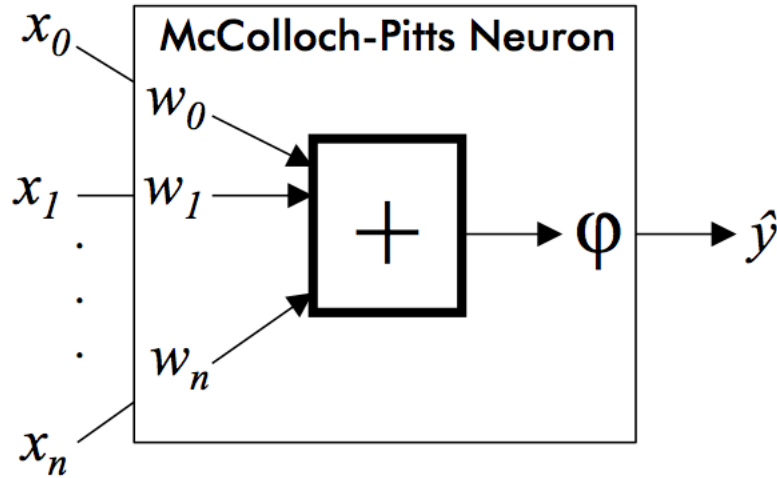


Figure 1.2. the neuron receives the input vector x . Each input element is multiplied by a corresponding weight from the vector w . The weighted inputs are summed and passed through a non-linear function to produce the output \hat{y} .

the input and output layers respectively. The middle layers are called hidden layers. The intuition behind arranging neurons in this manner is that the input layer can make some simple decisions about the input vector. These simple decisions are used to make slightly more complex decisions which are the input to the next layer and so on. The term deep learning (see Figure 1.1) is defined somewhat loosely to mean a neural network that has three or more hidden layers [18].

More concretely, the neural network might also be used to predict house prices. In this case, the earlier layers could learn to decide if the house is large or small. Then the later layers can make guesses based on whether the house is large and has a pool, or whether the house is small with a pool. In contrast, a single artificial neuron cannot make such conditional predictions it can only guess at how much each input feature contributes to the final output.

1.1.3 Machine Learning and Backpropagation

The term machine learning (see Figure 1.1) refers to a diverse group of algorithms including neural networks. What unifies all these technologies is that they are at their cores optimization problems. They appear to learn by systematically updating the weights

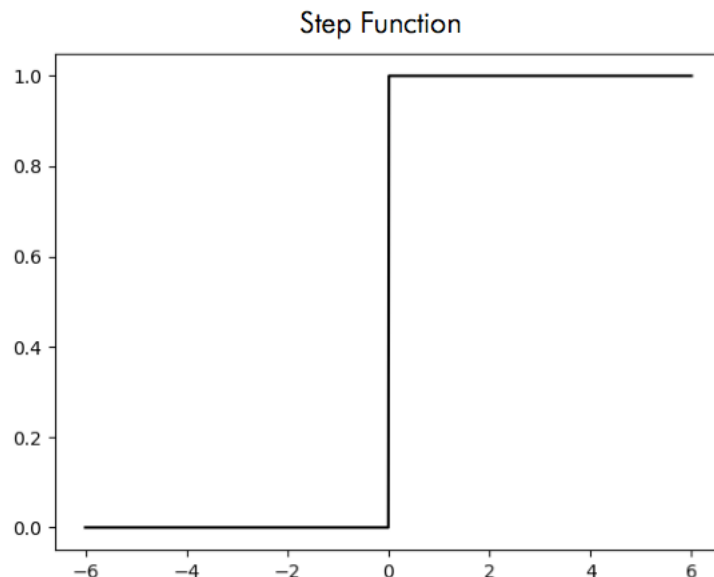


Figure 1.3. The piecewise activation function ϕ used in the McCulloch-Pitts neuron. Here the threshold is set to 0.

in their neurons to make increasingly accurate decisions. Backpropagation is the iterative algorithm that is used to optimizing the weights in the neurons (see Figure 1.2) so that model improves [18].

Ideally, a model will learn to always provide the correct answers for the training data and in the process will build an internal representation of the data set that it can use to make predictions about new inputs where the target output is not known. In the example of house prices, the hope is that through training on known house prices, the model can predict the price a new house can be sold for.

1.1.4 Sequence Modeling

Modeling sequential data is an application on which the type of neural networks described thus far tend to perform poorly. Consider the case of predicting the final word in a sentence given all the words leading up to it, where each input to the model is a single sentence and the label is the final word. There are a number of practical concerns with formatting the problem in this manner. First, there is the problem that sentences vary in length greatly. Some mechanism would have to be devised to compensate for this fact. Second, there is the problem that the information the model needs in order to make an

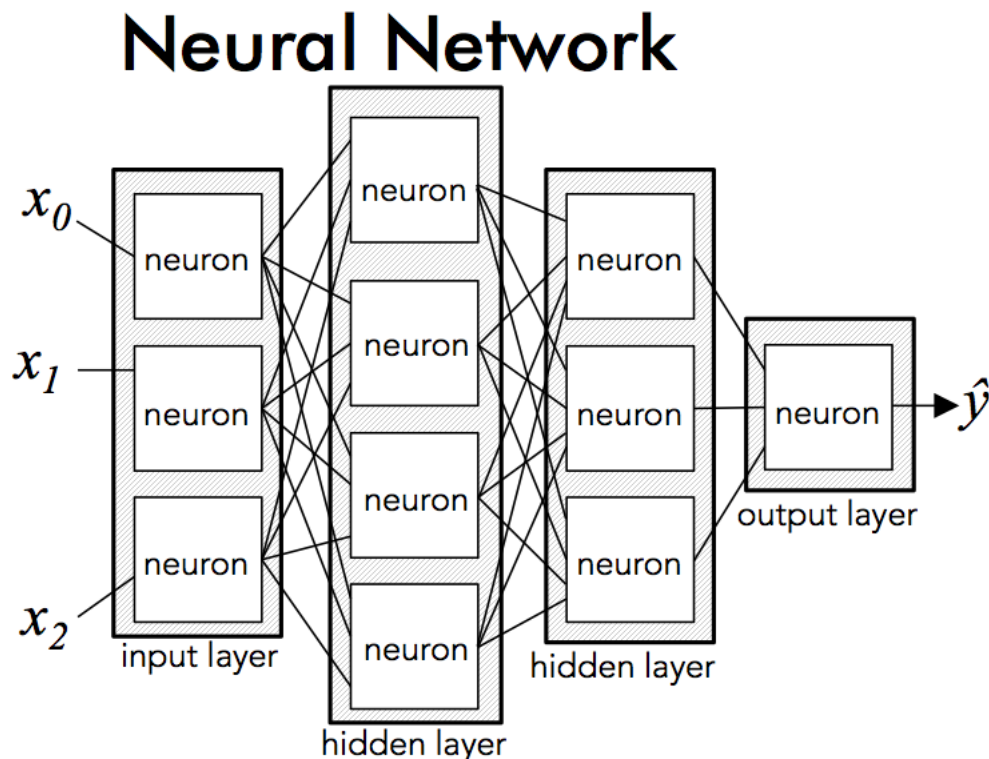


Figure 1.4. A simple neural network with four layers. The model receives as input the vector x and produces the output \hat{y} .

accurate prediction may not be in the current sentence. For instance, the sentence “She does not speak (blank)” may not have enough information to make a confident prediction. There are a large number of equally plausible answers: “clearly,” “loudly,” “quietly,” or “German” would all be valid predictions. If the previous sentence is included in the input, the model may be able to make a better prediction. For instance “She grew up in Canada. She does not speak (blank)” gives more clear context. The model now might guess that the answer is a language. However, if the sentence was “even though she grew up in Canada she does not speak (blank)”, then “English” or “French” might be better guesses.

To overcome these difficulties, the inputs might consist of a large number or preceding words rather than just a few sentences. This could potentially make the computation intractable. This does not really solve the problem of trying to make predictions based on a context that is not included in the input. It simply increases the distance the model can look back into the dataset. Consider the sentence “As mentioned in the previous chapter,

she does not speak (blank).” Clearly the model would perform better if it could store pertinent information for an arbitrary length of time [19].

1.1.5 Recurrent Neural Networks

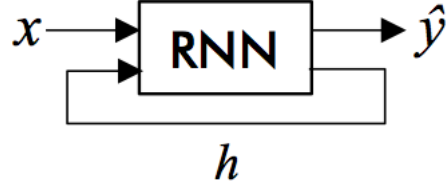


Figure 1.5. A simple RNN receives as input the vector x and the state from the previous time step, h . It produces the output \hat{y} and a new state vector.

$$h^{(t)} = \sigma(W^{hx}x^{(t)} + W^{hh}h^{(t-1)} + b_h) \quad (1.3)$$

$$\hat{y}^{(t)} = W^{yh}h^{(t)} + b_y \quad (1.4)$$

$$\sigma = \frac{1}{1 + e^{-x}} \quad (1.5)$$

Recurrent neural networks were developed to solve exactly this problem. The earliest RNNs were developed in the 1980’s [20]. Figure 1.5 shows a simple RNN. At each time step the model sees an input x and the state vector h computed as in (1.3), W^{hx} is a matrix of weights that are applied to the inputs connecting to the hidden layer of the RNN, W^{hh} is the weight matrix connecting the hidden layer to itself at the previous time step t , σ is the logistic function (1.5), and b_h is a bias term. The output \hat{y} is computed as (1.4) where W^{yh} is the weight matrix connecting the hidden layer to the output layer and b_y is the bias term. The sigmoid activation function (see Figure 1.6) is used rather than the piecewise activation function in the McCulloch-Pitts neuron (see Figure 1.3) [20].

The intuition behind RNNs is that the model is able to store relevant information in its state variable for an arbitrary number of time steps. They perform better on sequence modeling tasks than non-recurrent neural networks, henceforth feed forward neural networks (FFNNs). However RNNs suffer from the so-called vanishing and exploding gradient problems [21]. This situation occurs because the values flowing through the model are predominately very small and they are related through multiplication which leads to the

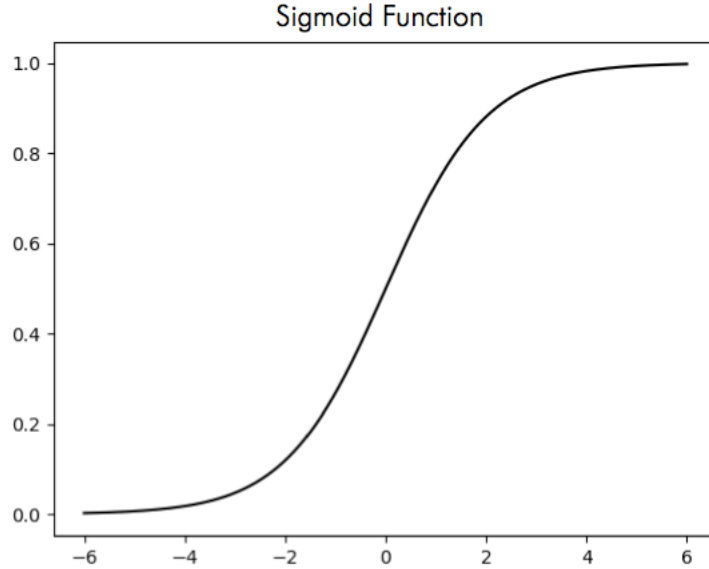


Figure 1.6. The sigmoid activation function resembles the step function ϕ used in Figure 1.2, but it is smooth and differentiable. At its extremes it acts like ϕ , but close to 0 it is more linear.

situation where the gradient signal is made so small that the weights are barely adjusted by backpropagation. Alternatively if the values become large, causing the gradients to become numerically unstable and prevent the model from learning effectively.

1.1.6 Long Short Term Memory Networks

Long Short Term Memory Networks (LSTMs) were developed by Sepp Hochreiter and Jürgen Schmidhuber in 1997 to solve the vanishing and exploding gradient problems [22]. They also introduced a more sophisticated mechanism for the model to store information called the *cell state*. An LSTM sees an input along with the state from the previous time step just like an RNN. It also sees the output from the previous time step. The equations for an LSTM are given in (1.6) through (1.11). Other architectures have been proposed since, but LSTM with some subtle changes from the original paper remains the state-of-the-art [23]. Conceptually it is helpful to break the computation up into separate stages called gates. Figure 1.7 illustrates how these gates work.

There are four neural network layers inside of the LSTM. They function in the same way as those described in Figure 1.4. Each layer has a weight matrix, a bias vector, and

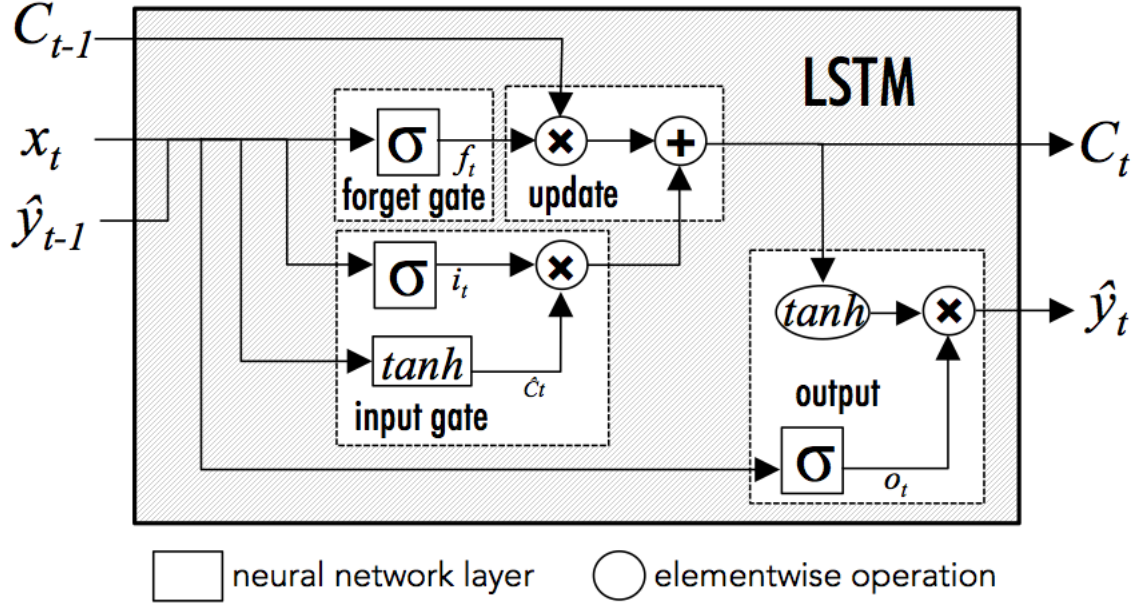


Figure 1.7. An LSTM takes in the previous \hat{y} , previous C , and the current x . These are used to compute the forget gate and input gate which are used to update the cell state. The output gate decides what \hat{y} should be.

a non-linearity. The non-linearity is either a hyperbolic tangent function or the logistic function (1.5). Each layer receives the same input: a concatenation of the current input, x_t and the output prediction from the previous time step \hat{y}_{t-1} . Each layer learns through backpropagation to perform a separate task.

$$f_t = \sigma(W_f \cdot [\hat{y}_{t-1}, x_t] + b_f) \quad (1.6)$$

$$i_t = \sigma(W_i \cdot [\hat{y}_{t-1}, x_t] + b_i) \quad (1.7)$$

$$\hat{C}_t = \tanh(W_C \cdot [\hat{y}_{t-1}, x_t] + b_C) \quad (1.8)$$

$$C_t = f_t * C_{t-1} + i_t * \hat{C}_t \quad (1.9)$$

$$o_t = \sigma(W_o[\hat{y}_{t-1}, x_t] + b_o) \quad (1.10)$$

$$\hat{y}_t = o_t * \tanh(C_t) \quad (1.11)$$

- **Forget Gate**

The forget gate decides what should be forgotten from the cell state based on the current input x_t and the previous output \hat{y}_{t-1} . The output of the forget gate layer is passed through the sigmoid non-linearity that pushes the output to be in the range 0 to 1. If the

forget gate wants to completely forget one element from its input, it will output 0 in the corresponding spot in the output. Likewise, if it wants to completely remember something, it will output 1. If the layer wants to partially remember or partially forget something, it will output some number in between.

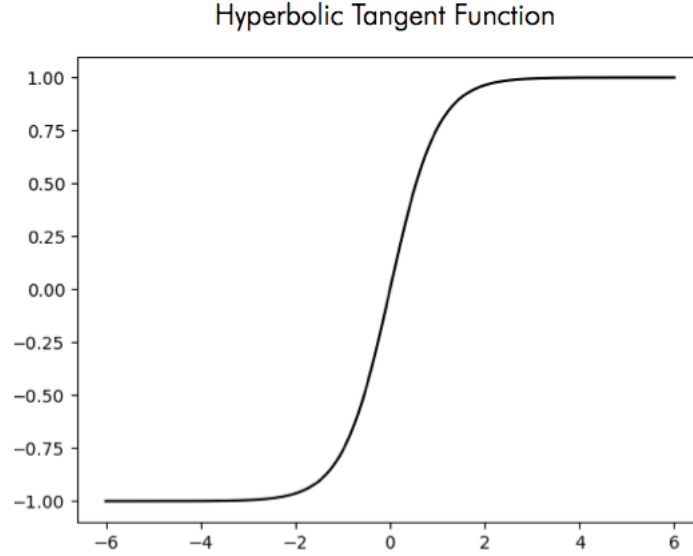


Figure 1.8. The hyperbolic tangent activation function closely resembles the logistic function but operates in the range -1 to 1.

- **Input Gate**

The input gate decides what new information will go into the cell state. This occurs in two steps using two parallel neural network layers. The first of these two layers decides which elements in the state should be updated by computing i_t as in (1.7). This layer is passed through the sigmoid non-linearity. The output will be close to 0 for the values the gate wishes to leave mostly unchanged, and the output will be close to 1 for values the gate wishes to completely change. The second of these two layers computes \hat{C}_t , which are all the candidate elements that might be updated in the state. The output of the two layers are multiplied elementwise.

- **Update Step**

The output, f_t from the forget gate, is multiplied elementwise by the current cell state to erase the elements the gate decided to forget. The output of the input gate is added

elementwise to the cell state. This is shown in the step labeled “update” in Figure 1.7.

- **Output Gate**

The output gate decides what from the cell state should form the output for the current time step. It does this by computing (1.10). Candidates for the output are chosen by passing the updated cell state through an elementwise \tanh . The output gate layer, like the forget gate layer, outputs numbers in the range 0 to 1 that are multiplied elementwise by the cell state which has an elementwise \tanh applied to it to decide which elements should be included in the output \hat{y}_t .

1.2 Sampling

Aside from being able to model sequences more successfully than FFNNs, RNNs can generate new output sequences using a process called sampling [24]. The algorithm works by choosing a seed, then iteratively making a prediction based on that seed, keeping some portion of the prediction, appending the kept portion to the end of the seed, and dequeuing the first portion of the seed. Intuitively the model is asked what it thinks comes after the seed data. Then it is asked what it thinks comes after its prediction and after that prediction and so on.

The term sampling will be used henceforth to describe this specific algorithm. It should not be confused with the term “audio sample” which refers to a scalar value in an audio file. The terms output and prediction are used interchangeably to describe a \hat{y} . To distinguish between a generic output and a sequence generated by an LSTM, the term sampled output will be used.

1.3 Related Work

Alex Graves has shown that LSTMs are capable of modeling and generating a range of complex signals. He demonstrated that an LSTM can learn to generate long passages of XML in the style of a Wikipedia article. He also demonstrated a model that was trained to produce samples of handwriting. The model can be shown a sentence and a sample of a person’s handwriting and produce an image of the sentence written as the person would have written it [24].

Andrej Karpathy and Justin Johnson created the project char-rnn [25]. They showed that models could generate complex texts one character at a time. Examples include long passages of grammatically correct but not necessarily coherent fiction in the style of Shakespeare, long passages of C code that are free from syntax errors but also not totally sensible, and Latex source code for an algebra text book. In another project they proved that when trained to generate text one character at time, it can be sometimes possible to deduce what the job of individual neurons is. They showed how models learned to use certain neurons to watch for characters like open quotation marks or left parentheses that needed to be paired with their closing counterparts [26].

A research group at Google completed a project called Deep Dream that was initially started as an effort to learn more about what the individual layers of a neural network were doing [7]. By iteratively maximizing the output of groups of neurons, they were able to produce images that showed what the model learned. These images are rather striking and have inspired some visual artists to explore the techniques further ¹.

One group investigated applying the Deep Dream algorithm directly to raw audio [27]. The researchers trained a convolutional neural network to map music clips to embeddings that were the result of a collaborative filter model. They produced output audio by showing the model either noise or silence, and then iteratively changing the input to maximize the activation of certain neurons. The resulting audio sounds very noisy and does not contain any recognizable sounds. The timbre resembles granular synthesis when no windowing function is applied to the grains.

Leon A. Gatys showed that it is possible to have a model learn to extract the style from an image and apply it to another image [9]. An example application is to apply the style of a painting onto a photograph. Using this algorithm it is possible to produce hypothetical images of what it would look like if a particular artist had painted the scene in the photograph. It has been shown that this approach can be adapted to be used with

¹See artist's website: <http://quasimondo.com/>

audio [28]. The audio produced in these experiments sounds like two audio sources being blended together.

The composer David Cope has been researching artificial intelligence in music for decades. His work focuses on analyzing musical scores and using this analysis to automatically generate new works in the style of the composer. His research largely involves analysis of musical style, segmenting of existing music and the application of heuristics to recombining these segments to form new works. He has published extensively on the topic [29] [30] [31] [32].

Another approach to generating content using neural networks called Generative Adversarial Networks (GANs) has been developed by Ian Goodfellow [33]. GANs actually consist of two models. One model generates an image and another learns to distinguish between real images and images that were generated by the first model. The model that generates pictures tries to make its outputs so convincingly real that the other model cannot tell which pictures are real and which are generated. This idea has been extended to try to teach the generator model to be more creative. In this scenario, the generator model also tries to create outputs that the other model cannot neatly categorize into separate styles [10]. Chris Donahue, Julian McAuley, and Miller Puckette published on a project that adapted this technology to audio [34]. They trained a deep convolutional GAN to produce audio. The resulting audio does contain recognizable sounds from the dataset. The outputs feature abundant artifacts. The timbre resembles audio that has been ring modulated by a constant non-harmonic frequency.

A research team at Google called Magenta was formed to investigate various applications of deep learning in the realm of visual arts and music [35]. To date, most of their projects have involved predicting symbolic notation, not on generating raw audio. They have developed models that seek to improve the performance of RNNs in remembering patterns in sequence data.

1.4 Contributions of This Work

This work focuses on sampling audio sequences using LSTMs. The goal is to explore systematically the numerous different variations in approach that can be taken to this specific task and also to produce an application that makes the process simple, reliable, and easy for users. These results are intended to be used to create musical compositions. The process is similar to the way that many composers of electro-acoustic music already work in the sense that they generally begin with some recorded sounds, process them with a series of effects and then combine them with other sounds to make pieces of music. The data used in this work is comprised of real-valued audio samples in the range -1 to 1 or magnitude spectra normalized to the range 0 to 1.

1.4.1 Evaluation

The desired outcome of this research is something akin to an audio effect. Ideally, a user would be able to produce output sounds that occupy a middle ground between being surprising but also conforming to some set of expectations. This is very similar to the results from the char-rnn project where it generated new text in the style of Shakespeare. The text was plausible, retained much of the structure and style of the input, but was also rather surprising in many ways. This situation is similar to when musicians apply time-varying audio effects either during live performance or to fixed media. Musicians might expect to be able to clearly hear their own instrument sounds. They might also expect the sound to be colored by a phaser for instance. The sound is predictable in a general sense, but the exact moment-to-moment progression of sounds might be somewhat surprising.

Evaluating the extent to which a particular segment of audio occupies this middle ground is difficult as there are no obvious quantitative metrics to measure this quality. Instead, two sets of criteria are presented for assessing the relative success or failure of a model in producing desirable outputs. The first set assesses the qualities of the audio itself, and second set assesses the performance of the model more abstractly.

1.4.2 Sampled Evaluation Criteria

The criteria for sampled outputs hold that the audio being evaluated:

1. is relatively free from artifacts,
2. obviously resembles the input, and
3. does not match the input literally.

Criterion 1 states that the outputs should be relatively artifact free. The term artifact means any aspect of the sampled sequence of audio that is undesirable for use in music, especially aspects that are difficult to remove via filtering or other means. For a sampled output to score low on this criterion, the artifacts present in the audio must be so pervasive or so difficult to ignore that the audio is unusable in musical applications. An example of this type of artifact would be numerous large discontinuities in the audio.

Criterion 2 states that the sampled audio should obviously resemble the training samples. This means that a reasonable listener can easily discern some similarities in frequency and timbre between the training audio and the sampled audio. For example, if the input for the model were piano recordings, two cases of failure for this criterion would be pink noise or a sine wave. If the sampled outputs are noisy they score low on Criterion 2 because most reasonable listeners would say that noise does not resemble piano. A sine wave in the sampled output would score low on Criterion 2 because, even though there are no apparent artifacts in the prediction, it does not resemble the training audio any more than it does any other musical input.

Criterion 3 states that the sampled audio should not resemble some portion of the training audio so closely that the two cannot easily be distinguished once any artifacts in the sampled audio are discounted. One example would be if the model predicts several seconds of the training audio unchanged, or with minor artifacts introduced such as short passages of low amplitude noise. In this case the model clearly learned to predict the correct answers when presented with specific training input, but did not learn to generate any new data or to generalize from what it learned and apply this information in new

circumstances. Scoring high on Criterion 3 would entail at least reordering of the pitches from the input or some other significant transformation of the data that preferably cannot obviously be repeated through some other means.

1.4.3 Model Evaluation Criteria

In addition to the primary criteria, a second set of concerns guide the assessment of models. Enumerated below, these criteria include more abstract measure of success specifically regarding the performance of the model.

The model evaluation criteria hold that:

1. comparable results are possible with different data sets,
2. among the sampled outputs generated for a single data set, there is consistency, and
3. the sampled outputs are not prohibitively slow to create.

Model Criterion 1 measures the extent to which results achieved on one input dataset are able to be reproduced when a different data set is used. A model that performs well on only one kind of sounds is less desirable than one that is able to produce a wide range of sounds reliably.

In some experiments some models, at some points in training sample outputs that score highly on a number of criteria, but either started out, or go on to later produce results that score much worse. In general a model that is more reliable is preferable to one that only produces desirable results under certain circumstances. Especially when those circumstances are difficult to discern. Model Criterion 2 measures this quality in a model.

Model Criterion 3 assess, the general feasibility of the model. Some more exotic approaches may seem promising but are so unwieldy that they are not of any practical use at the present date. Examples include models that take scores of hours to train, models that take a very long time to generate an output with a very short duration, or a model that is otherwise too difficult to assess. Models that score low on any of these criteria are not addressed directly because they were deemed too unwieldy for practical use.

Assuming a model scores highly on these three criteria, it can be trained and sampled

from to produce new audio. Whether or not the model is successful in learning and sampling outputs that score highly on the output evaluation criteria described above is dependent on a number of factors. These factors will be discussed in detail in later chapters.

Chapter 2

Training

2.1 Training Algorithm

LSTMs learn by making predictions based on inputs. Figure 2.1 shows the process of training a model. At each time step the model sees an array of inputs and predicts an output for each array element. The accuracy of the prediction is evaluated by comparing it to a label via the loss function. The error in the prediction, the loss, is passed to the optimizer. The optimizer updates the internal weights in the model via backpropagation. The result of this process is that the next time the model sees the same input, its prediction will be more accurate. The algorithm can be described as the following:

```
for each time step:
    input, label      = get_new_input_and_label()
    prediction, new_state = model.predict(input, prev_state)
    current_loss      = loss(label, prediction)
    optimizer.update_weights(current_loss)
```

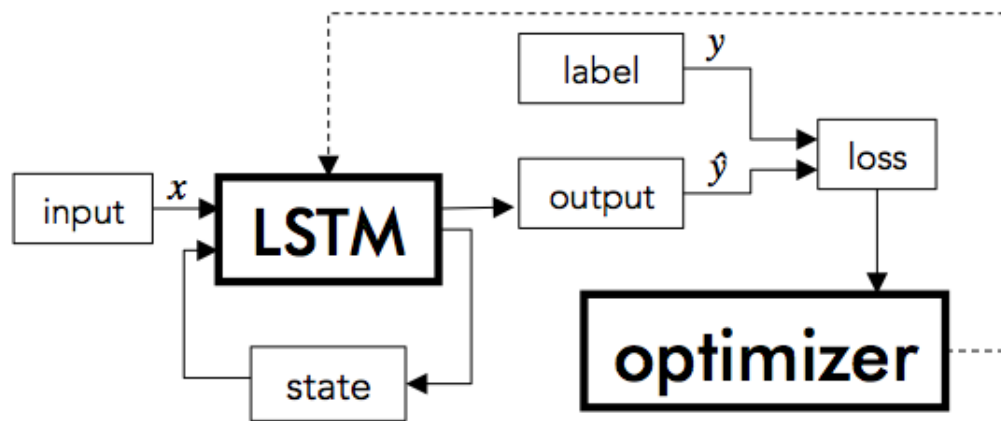


Figure 2.1. During training, the model produces an output prediction based on the input data and updates the state which stores short term information about recently seen inputs. The fitness of this output is calculated via the loss function. The optimizer teaches the model to make better predictions by updating the internal parameters of the model.

2.1.1 Input and Output Shapes

For all of the models presented here the input is formatted as a matrix. When referring to model settings and also when referring directly to code, the number of rows in the matrix

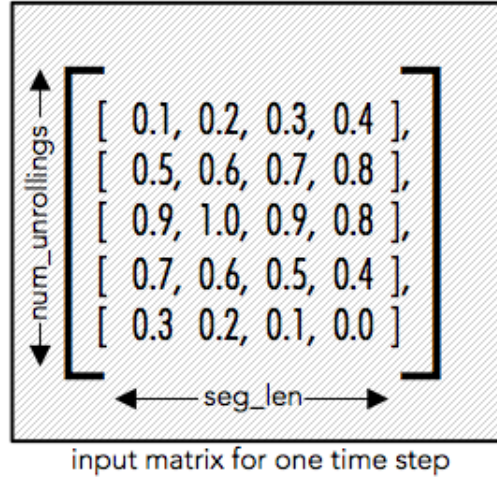


Figure 2.2. An example input for a single time step, where `num_unrollings` is used uniformly to mean the number of vectors in a single input, and `seg_len` is likewise always used to mean the length of each vector in the input. In this example `num_unrollings` is set to 5 and `seg_len` is set to 4.

will be uniformly called `num_unrollings`. Likewise the length of each array in the input array will always be called `seg_len`.

The output from an LSTM is always the same shape as the input, hence for an input where `num_unrollings` is equal to 100 and `seg_len` is equal to 1024, the LSTM will make a prediction that is also 100 vectors of 1024 elements each. This corresponds to the input and output boxes in Figure 2.1. The shape of the output can be transformed by neural network layers before it goes to the loss function.

2.1.2 Calculating Loss

The next step in the training process is to calculate the loss. In order to update the weights in a model, it first must be known how accurate the prediction, \hat{y} is. The loss function measures how different the model's output is from the target answer, the label. This work makes use of the mean squared error loss function (Equation 2.1) where n is the number of elements in label, and \hat{y} is the model's output. The intuition here is that when the loss is low, the model is performing well because the predictions \hat{y} are very close to the labels y .

$$mse = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (2.1)$$

2.1.3 Reshaping LSTM Output

The mean squared error loss function requires that the prediction and the label have exactly the same shape. If the labels for a particular problem are a different shape than the inputs and outputs, the outputs will need to be processed after they come out of the LSTM. The best way to accomplish this is through adding additional neural network layers. The second dimension of the output of a neural network layer will match the number of neurons in the layer. Hence, if a matrix of data with the shape 100 by 1024 goes into a neural network layer with 32 neurons, the output from the neural network layer will be 100 by 32.

2.1.4 Backpropagation and Gradient Descent

Gradient descent is a procedure for minimizing error functions. Backpropagation applies gradient descent to minimizing the value of the loss function of a neural network model. Figure 2.3 shows how the process would work for a model with only one weight, θ . The weight begins at some initial value. It is iteratively updated according to the derivative of the loss function with respect to the weight scaled by the so-called learning rate, α as in Equation 2.2, where $J(\theta)$ is the loss function and the $:=$ operator indicates an update of the value stored in the variable. Intuitively, the derivative in this case at a value for θ is the slope of the curve at that point. The algorithm tries to move θ in the direction of the steepest descent. If the slope is negative, this results in a negative number being subtracted from θ increasing its value and moving it closer to the minimum. Likewise if the slope is positive, this results in a positive number being subtracted from θ .

$$\theta := \theta - \alpha \frac{d}{d\theta} J(\theta) \quad (2.2)$$

Backpropagation with neural networks is essentially the same as the simplified example

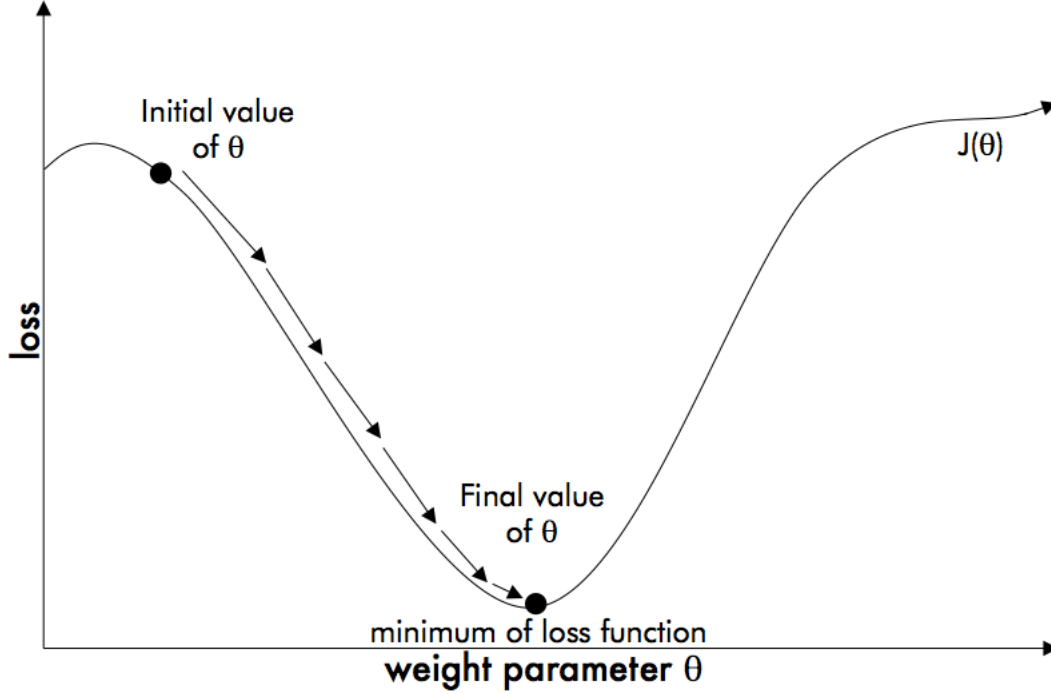


Figure 2.3. A simplified illustration of minimizing an arbitrary univariate loss function. An initial value for the weight is chosen. The value is iteratively adjusted by the gradient descent algorithm until a minimum is reached. Note that because θ is being multiplied by the slope of the curve, the size of each step gradually gets smaller as the slope approaches 0.

described above, except that there are many more weights being updated. Calculating the derivatives of the loss with respect to individual weights is accomplished by way of the chain rule. The input x is shown to the model, which produces the output \hat{y} . The loss is calculated as in (2.1). The gradient is calculated for the final layer of the model first, then each previous layer is calculated until the gradient can be determined for the first layer. Hence the error signal is propagated backwards through the weights of the model.

2.2 Hyperparameters

The models to be presented here feature a number of different settings that must be chosen carefully. They are collectively called hyperparameters to differentiate between the model's weights which are sometimes known as parameters. Table ?? explains how the most important hyperparameters affect training. Presently the hyperparameters that have the most drastic effect on learning and on the quality of the generated audio sequences will be

discussed in detail. All of the hyperparameters and their interactions will be discussed at length in Chapter 4.

2.2.1 The Goldilocks Principle

One of the main challenges of achieving success in machine learning is in learning to effectively choose hyperparameters. After a model has been successfully designed and implemented, there is still the matter of making decisions about several values. While reasonable choices for these hyperparameters are readily available online, the problem is that the settings are not mutually exclusive. In other words, the hyperparameters have a highly complex relationship to each other. For instance, one setting of hyperparameter α might work well in some preliminary tests, but work poorly when other hyperparameters are changed. Developing a highly successful model requires vigilant attention to these unforeseen interactions. The process of choosing values that exist between extremes will henceforth be referred to as the Goldilocks Principle [36].

2.2.2 Learning Rate

The learning rate is the colloquial term for α in (2.2). It scales the size of the step that is taken during a gradient descent update step. If the absolute value of the slope of a loss surface is very high and the learning rate is not applied to scale back this update, the step might be so large that the weight will overshoot the minimum it is aiming for. Rather than gradually getting better over time the models weights might become unstable and diverge. Thus the learning rate is an effect way to rein in the gradient descent algorithm and make it progress more carefully.

However, choosing an appropriate value for the learning rate is quite difficult in practice. This is because the learning rate by itself is highly sensitive to the Goldilocks Principle. A high learning rate will often result in a dramatic increase in accuracy of the model early in training. However the model will stop learning soon after that. Instead it will quickly reach a plateau in its loss values and remain there for the duration of training. Alternatively, a learning rate that is too low may be prohibitively slow to train. The loss will decrease over

time, but ultimately a higher learning rate may be more effective and may accomplish the same or better results in a drastically shorter amount of time. A learning rate that is in the Goldilocks zone is neither so large the model becomes unstable nor too small that the model takes too long to converge. To complicate matters further, the choice of learning rate may need to be ammended because of later changes in other hyperparameters.

2.2.3 Data Type

The present work only considers two kinds of data to be used for generating audio sequences: raw audio samples and magnitude spectra. In the first case, the model is shown audio samples as input. The audio is stored as floating point numbers in the range -1 to 1. The alternative is that each input vector is a magnitude spectrum frame. For all the experiments presented in this work, the magnitude spectra are normalized to the range 0 to 1 and the negative frequency information is discarded.

2.2.4 Data Shape and Label Shape

Among the most significant hyperparameters are those that govern the details of how the input audio files are segmented and overlapped to form the input matrices for the various time steps. Figure 2.4 shows an example of how these hyperparameters are used. For non-overlapping audio vectors, the hyperparameter `int_cursor_inc` should be equal to `seg_len`. For audio vectors that overlap by half `int_cursor_inc` should be equal to `seg_len` divided by two.

Equally important is the relationship between the inputs and the corresponding labels. Figure 2.5 shows how labels can be drawn from the same input audio file. The hyperparameter `label_offset` describes the difference between the first index of the input and the first index of the corresponding label. The intuition behind this hyperparameter is that it controls what exactly the model is trying to predict.

Once a model has been trained to accurately predict data, it can be used to make new predictions and to sample new audio. The success of the model in being able to sample outputs that score highly on the output evaluation criteria discussed in the previous chapter

is related to the model being trained to make accurate predictions. The relationship is complex, though. Sometimes a model can achieve a high level of success during training and still not be able to successfully produce sampled outputs that are evaluated positively. The success in sampling can be thought of as a byproduct of a trained model. This is significant because the quality of the sampled outputs can only be influenced indirectly by making changes to the training procedure.

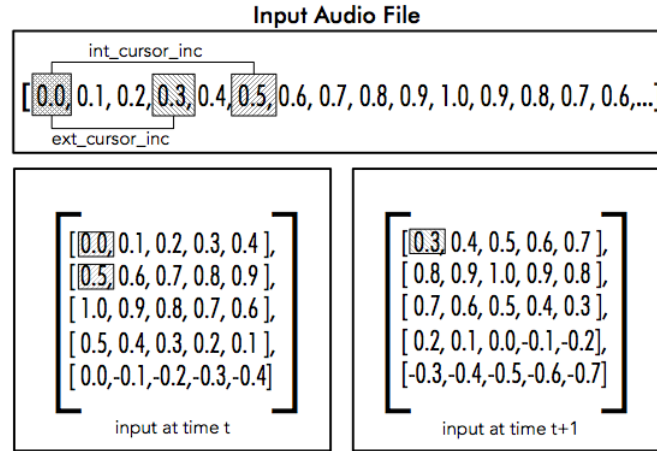


Figure 2.4. Example input arrays from adjacent time steps. The first array of input at time t begins at index 0. The second array at time t begins at index `int_cursor_inc`. At timestep $t + 1$ the first array begins at index `ext_cursor_inc`.

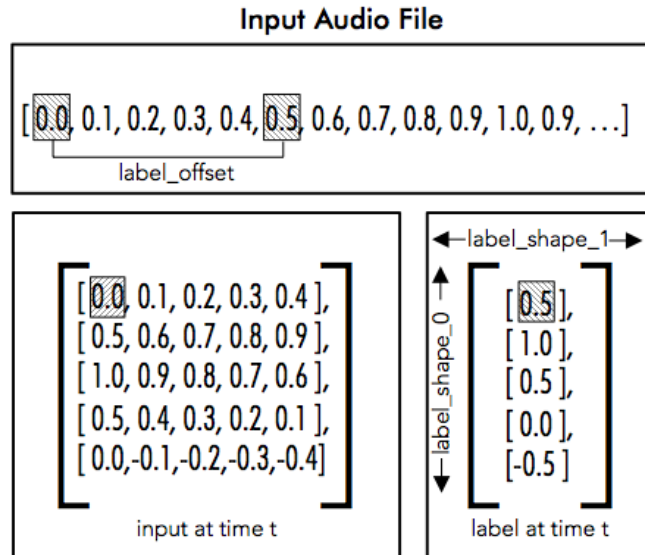


Figure 2.5. An example input and label pair. In this case the hyperparameter `label_offset` is set to 5. Notice that labels are formed in the code in the same manner as the inputs. Hence `int_cursor_inc` is equal to `seg_len`.

Chapter 3

Sampling

3.1 Overview

To sample an output, the model sees inputs and makes predictions in the same manner as during training (see Figure 2.1). The nature of the input and the handling of the output is the main difference. The algorithm used for generating and output is depicted in Figure 3.1. It can be summarized by the following psuedocode:

```
seed = get_new_seed()
output = []
for each sampling time step:
    prediction, new_state = model.predict(seed, prev_state)
    output.append(prediction[-1])
    seed.append(prediction[-1])
    seed = seed[1:]
```

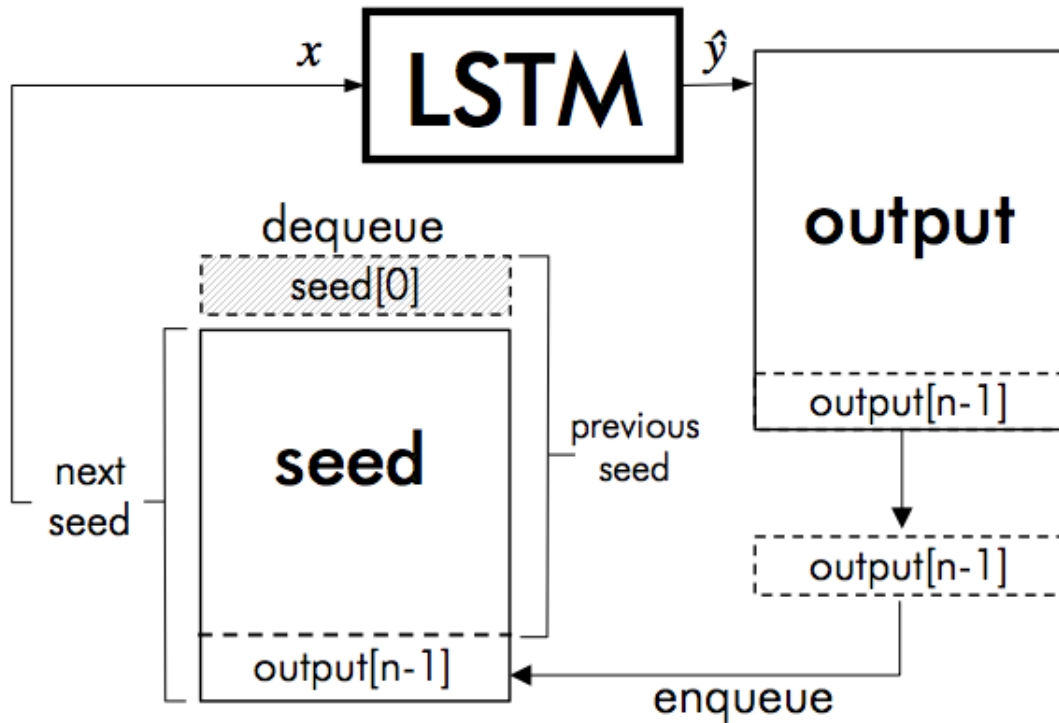


Figure 3.1. After the initial seed has been input to the model, it makes a prediction \hat{y} about the following audio in the same manner as during training. To generate an output sequence, the final element from the prediction is appended to the seed, and first element from the seed is discarded to form the next input x . This process repeats until the total desired output length is reached.

The process begins with a seed comprising an arbitrary matrix with the same dimensions as the inputs used during training. This data is presented to the model as it would be during training. Then the model is applied. A segment of the output from the model is kept, generally the last portion as this is often the only part of the prediction that does not appear elsewhere in the seed.

The intuition behind this is best understood with the help of Figure 2.5. In that example, the model is making a prediction of one audio sample for each input vector. This is interpreted as the model’s prediction of what the next sample would be after each input vector. Because the hyperparameter `int_cursor_inc` is equal to `seg_len` in the example, this means that the model is essentially trying to predict the samples in the first column of the input starting at the second row. Only the final sample in the label is not present in the input. The model must consider the input as context and make a prediction based on all it has learned up until that point to make its prediction.

It should be noted however, that it is not currently known whether LSTMs are capable of this hypothetical searching in the input for the correct predictions. It is more likely that each prediction is made in the same manner as the final one. However, if the user wishes to ask the model to predict a new passage of audio that is not present elsewhere in the input (that is to ask the model to predict what comes after the input), the method described here must be used.

During sampling, the appropriate subsection of the prediction is appended to the end of the seed that was used to make the output (labeled `output[n-1]` in Figure 3.1). A portion that has the same size as the kept portion is removed from the beginning of the seed. This is to ensure that the input at each time step is the same shape. The process is repeated an arbitrary number of times to generate a sequence. Over time, the seed gradually contains less real data and more predicted data from the model.

3.1.1 Sampling Methods

When using raw audio as the input for a model, approaches such as the following can be taken to formatting the data.

- **Vector Approach**

The simplest and most direct method of formatting the data is to separate the data into segments of length `seg_len`. Then each input will consist of `num_unrollings` segments. The labels to be paired with these input arrays are also groups of `num_unrollings` segments of audio. For each input audio segment, the model tries to predict an equal length segment. The following hyperparameter settings would accomplish this type of data format: `num_unrollings = m`, `seg_len = n`, `int_cursor_inc = n`, `ext_cursor_inc = n`, `label_offset = n`, and `label_shape = (m, n)`.

- **Column Vector Approach**

Alternatively the label can be specified to have shape `num_unrollings` by 1. In this scenario the idea is to have the model predict a single sample for each vector in the input array. Hence if the first array contains audio samples indexed from 0 to 1023, the model's job is to predict the audio sample at index 1024. Likewise if the second audio vector in the input array contains the samples at index 1 through 1024, the model will be asked to predict the audio sample at index 1025 in the second row of its output. The hyperparameters required for this set up are the following: `num_unrollings = m`, `seg_len = m`, `ext_cursor_inc = 1`, `label_offset = n`, and `label_shape = (n, 1)`.

- **Transpose Approach**

The model can be asked to predict a single sample for the entire input. This requires a somewhat more exotic handling of the predictions. The output from the LSTM will have the same shape as the input (`num_unrollings` by `seg_len`). The output will then be passed through a neural network layer and have its shape transformed to `num_unrollings` by 1. To reduce the output to a scalar, the transpose of the `num_unrollings` by 1 array will be passed through another neural network layer to obtain the final shape of 1 by 1.

- **Magnitude Spectrum Approach**

Each data element can also take the form of a magnitude spectrum window. In this situation, the input and outputs are interpreted as arrays of normalized magnitude spectra. This approach resembles the audio vector approach closely with the only differences being the content of each vector and the necessity for the audio to be resynthesized via a vocoder after the predictions have been made. The hyperparameters required for this type of setup are the following: `window_size = m`, `seg_len = m`, `num_unrollings = n`, and `ext_cursor_inc = 1`. Note that when using magnitude spectra as inputs, the hyperparameter `ext_cursor_inc` refers to the difference between the index of the first window in an input array at time t , and the index of the first window in the input array at time $t + 1$. With raw audio as the input, the difference would be between audio sample indices, not audio vector indices.

3.2 Vocoder Design

A simple channel vocoder is used to resynthesize predictions made on magnitude spectra. For simplicity the phase information is discarded in the present implementation. Figure 3.2 illustrates the process. The algorithm works as follows:

1. The spectra are optionally subjected to a heuristic whereby only bins that have more power than both of the adjacent bins are kept. The rest of the bins are set to zero.
2. The predicted array of magnitude spectra are transposed such that the rows are the bins and the columns are time steps.
3. Each pair of elements in the rows are linearly interpolated between by `hop_size` samples.
4. Each row is then treated as an envelope for an oscillator with a frequency corresponding to the center frequency of the FFT bin. The oscillator is multiplied by the corresponding envelope elementwise.

5. All of the bins are then summed element wise to produce the final output audio. The initial phases of the oscillators are set randomly.

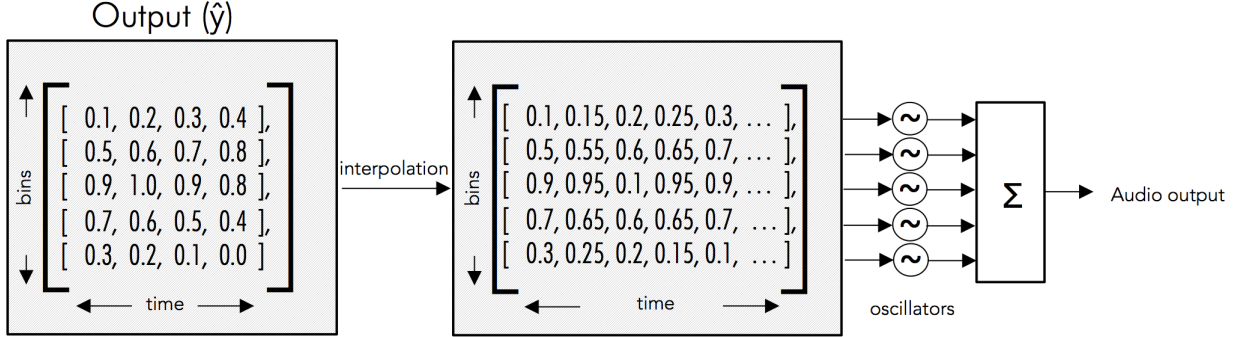


Figure 3.2. Magnitude spectrum frames are linearly interpolated between to create a sampling rate envelope which is multiplied by a bank of oscillators whose frequencies are set to the center frequencies of the FFT bins.

3.3 Sampling in Series or Concurrently

When sampling output sequences from a model, there are two choices about exactly how and when this process occurs. The simplest method is to periodically pause during training to sample an output sequence (see Figure 3.3). This is helpful because after a short amount of training, it is sometimes possible to see if a model and its hyperparameters seem promising or not. It is preferable to waiting until a model is completely done training to start sampling, but it also has its drawbacks.

Consider first the situation where a model is being sampled in series. If a very time consuming method of sampling is chosen, a single script can incur a significant time increase by pausing training to sample too frequently. This is especially a problem when the model pauses training early on during execution. Also this method can sometimes lead to output sequences that appear to get stuck repeating a single prediction for their duration. Intuitively this makes sense since the model is looking at a small portion of data with no surrounding context. The model has not had much time to learn what information is pertinent to making accurate predictions, or how to write it into its state.

Another method for sampling that alleviates many of these issues is to sample outputs concurrently with training (see Figure 3.4). In this case, rather than periodically paus-

ing training to sample an output sequence, the sequence is generated one step at a time concurrently with the training. For each time step, the program completes one iteration of training and one iteration of sampling. When the desired length of output has been reached, the output is saved, a new seed is chosen and the process begins again.

This method has two main benefits. The first is that for slow predictions methods, such as sampling one audio sample at a time, the user does not have to add a large amount of time to the run time of the program. The second is that by updating the model's weights while the predictions are being made, the tendency for sampled outputs to get stuck repeating the same prediction multiple times is avoided. Anecdotally, this would seem to make sense because it is learning new information while being asked to predict a long sequence. Consider the hypothetical case where the model is asked to predict a sequence it is not prepared to predict. For instance it is shown an input that contains largely a single sound. Unless the model has learned to never predict a single sound for more than a certain number of time steps, it has no reason to stop making the same prediction. However, if the model is learning new information while it is making its prediction, it is reasonable to think that its predictions might change based on its new knowledge. In this case, it is unlikely that it would ever produce the exact same prediction multiple times, because the model's internal representation of the data has changed. The concurrent method is what the author considers to be preferable to the sequential method. For this reason the audio examples shown in Chapter 5 are all created with the concurrent method.

As mentioned before, the quality of the sampled outputs can only be effected indirectly by making changes to the training procedure. Making choices about hyperparameters is the main manner through which this is accomplished. Just as the processes of training and sampling have a complex relationship, the hyperparameters also interact in subtle and sometimes unexpected ways.

```

for each epoch:
    for each timestep:
        input      = get_input()
        label      = get_label()

        y_hat, state = model.predict(input, state)
        loss         = loss(y_hat, label)
        model.update_weights(loss)
        if timestep % wait == 0:
            seed      = get_seed()
            output = []
            for each dream_timestep:
                pred, sampling_state = model.predict(seed, sampling_state)
                output.append(pred[-1])
                seed.append(pred[-1])
                seed = seed[1:]

```

Figure 3.3. Psuedo code for sequential sampling algorithm.

```

seed      = get_seed()
output = []
for each epoch:
    for each timestep:
        #train normally
        input = get_input()
        label = get_label()
        y_hat, state = model.predict(input, state)
        loss = loss(y_hat, label)
        model.update_weights(loss)

        #make a prediction
        pred, sampling_state = model.predict(seed, sampling_state)
        output.append(pred[-1])
        seed.append(pred[-1])
        seed = seed[1:]
        if len(output) == output_length:
            save_output()
            output = []
            seed = get_seed()

```

Figure 3.4. pseudo code for concurrent sampling algorithm.

Chapter 4

Hyperparameters

4.1 Overview

Once a model architecture, dataset and data format are chosen, several more subtle decisions must be made to effectively train a model. These settings that need to be carefully chosen are called hyperparameters.

4.2 Number of Epochs

In machine learning parlance, an epoch is a single pass through the whole data set. In general it is advisable to train a model for many epochs. If the model is learning, the loss over time will decrease because the model is better able to predict the correct outputs. Eventually the slope of the loss will approach zero meaning the model is nearing its limit for accuracy. Recent advances in understanding of the internal workings of neural networks show that the model often learns first to memorize what it should predict based on the inputs, then in later epochs it learns to generalize this information [37].

4.3 Number of LSTM Layers

So far LSTMs have only been discussed as singular entities. Sometimes to model more complex datasets, multiple LSTMs can be used to form a single model. In this situation, the output of one LSTM is the input to the next. They function in a similar way to the layers in Figure 1.4. This modification adds to the total time of training, but it has the potential to allow the model to learn more than it could with a single LSTM. Models that are very large and complex that are tasked with learning a very simple data set run the risk of overfitting. This term refers to the situation where the model can totally memorize the data and achieve almost perfect accuracy, but does not tend to generalize well at all. In this situation, the model may be able to correctly predict any value from the data set, but it will perform very poorly on any other data that it has not seen before. This is yet another case where the Goldilocks principle is at play. The designer of a model must strike a balance between a model that too small to build a complex understanding and a

Table 4.1. Hyperparameters, their names in code, and their meanings.

Hyperparameter Name in Code	Data Type	Explanation
lr	float	α in Equation 2.2
num_epochs	integer	number of times to show the entire data set to the model
weight_stddev	float	standard deviation of the noise the fully connect layer weights are initialized with
num_layers	integer	number of LSTM layers
use_dropout	boolean	flag for using dropout
use_residual	boolean	flag for using residual connections with multilayer lstm
regularization	boolean	flag for applying regularization to LSTM weights
reg_amount	float	amount of regularization to apply to LSTM weights
input_dropout	float	chance that input LSTM connection will be kept
output_dropout	float	chance that output LSTM connection will be kept
seg_len	integer	length of each input vector
num_unrollings	integer	number of input vectors in each input matrix
int_cursor_inc	integer	difference between index of first element in adjacent input vectors within a single input matrix
ext_cursor_inc	integer	difference between index of first element in adjacent input matrices
label_shape	tuple	tuple describing shape of label matrix
label_offset	integer	difference between index of first input element, and index of first label element
output_shape	tuple	tuple describing the desired shape of \hat{y}
use_transpose	boolean	flag for using transpose method
use_fft	boolean	flag for using magnitude spectrum data as input
hop_size	integer	hop size to use when taking FFT
window_size	integer	size of the frames of audio used when taking FFT
zero_padding	boolean	flag for padding audio frames with equal amount of zeros

model that is so large it does not have to learn anything but instead can just memorize the training data.

4.4 Weight Initialization

The weights in the neurons of neural network layers must have some place to start during training (see Figure 2.3). Since the job of the model is to adjust these weights through backpropagation, it is easy to naively think that the initialization of the weights does not matter. A well trained model will often have small weights. So an obvious choice would be to initialize all the weights to zero. The problem with this approach is that it can lead to a situation depending on the network architecture where all the weights learn the same thing and the model ends up with a large number of identical weights which ultimately limit the complexity of the understanding the model can build. A better approach is to initialize the weights to very small random numbers. Often normally distributed noise is used. The specifics of the distribution is a focus of research lately with settings catered to specific architectures being proposed [38].

4.5 Residual Connections

As mentioned above, a model can often be improved by adding more layers. In addition to the problem of overfitting, recent research has shown that gradients used for backpropagation are degraded when more layers are added to a model. One proposal that is very promising for avoiding this problem is to use residual connections [39]. In this situation, rather than computing an output that will be passed directly to the following layer as in Figure 1.4, a layer learns to compute a signal that will be added to its input and this summed signal will be sent to the next layer. The intuition is that rather than each layer learning to output a signal for the next layer, each layer instead learns to add something to the information signal that flows through the layers.

4.6 Weight Regularization

Regularizing the weights in a model is a well known technique that is used to avoid overfitting [40]. This technique introduces an additional term to the loss function that penalizes the model when it allows its weights to become large. The intuition behind this

technique is somewhat subtle. One way to think about the benefit of weight regularization is that it reduces the effect that any one neuron can have on the final prediction of the model. Thus the model must learn to use a large number of neurons to make a decision which hopefully leads the model build a more general understanding of the data.

Regularization can somewhat help prevent the model from overfitting by essentially memorizing the data. A perhaps more accurate description would be to say that the model learns something like a heuristic for each training example. It can still make an accurate prediction, but the heuristic is catered to the single training example and as such cannot be used for a new data point. Weight regularization can force the model to use a larger number of simpler rules to make its prediction, effectively preventing it from creating increasingly arcane rules to accommodate new training data.

4.7 Dropout

Another technique to prevent overfitting has a similar effect to weight regularization but works very differently. This technique involves masking the output of a randomly chosen subset of the neurons in a neural network at each time step [41]. Though this technique seems somewhat drastic on its face, it has been proven to be very effective and is quite established. The intuition here is similar to that of weight regularization. Because the model cannot rely on its neurons to always contribute to the prediction (as they might be randomly ignored at any point), the model is forced to learn a redundant understanding of the data. Ideally, the effect is that the model builds a more robust understanding of the data which generalizes more effectively to new data.

4.8 Choosing Hyperparameter Values

One of the biggest challenges facing a machine learning researcher is in effectively choosing hyperparameters. As mentioned above, many of the hyperparameters must be chosen according to its own Goldilocks principle. The situation is further complicated by the fact that many of the hyperparameters must compensate for the values of other hyperparameters. Consider the case of a model where the weights are initialized to values that are too high. A researcher might see that the loss is very high with an initial set of

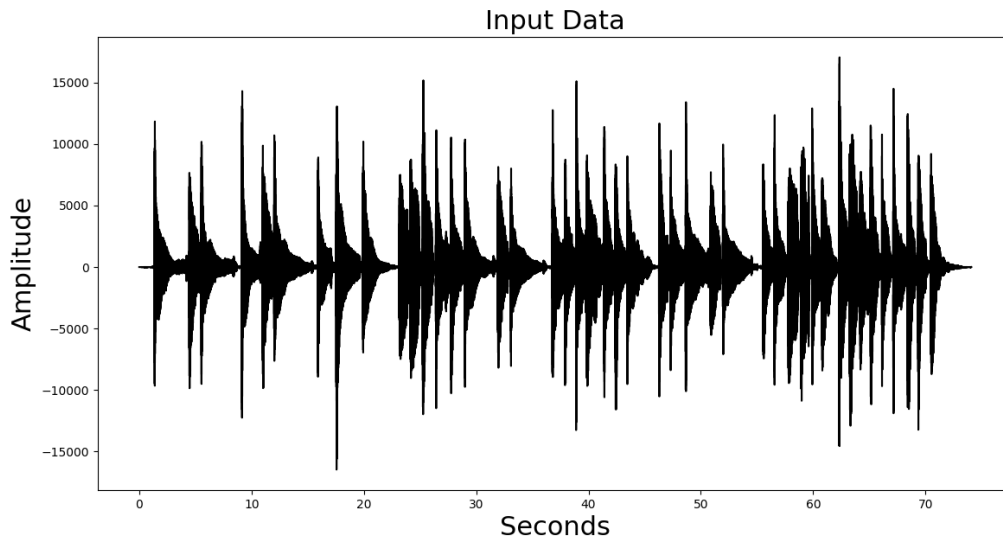
hyperparameters. Increasing the learning rate might show an improvement in performance. This makes sense because at the beginning of training, the model will see that its weights need to be made much smaller. The higher learning rate will accomplish this. The overall loss will decrease quickly because the gradient descent algorithm is sped up, but the ultimate accuracy of the model might not be much better. Similarly if the researcher then begins to initialize the weights in the model to a smaller value, the learning rate may then be too high. The situation is further complicated by the fact that some researchers will gradually change hyperparameter values over the course of training. In this case, a hyperparameter setting might be appropriate at the beginning of training, and be made more effect if it is decreased gradually. But if the hyperparameter is decreased too quickly, the overall accuracy might suffer.

As more research is done into the process of effectively choosing hyperparameter settings, some rather unfortunate situations are being uncovered. Examples include situations where popular technique to improve performance like weight regularization and dropout are shown to cancel each other out or to collectively make accuracy worse [42]. For the time being trial and error is required when it comes to the task of choosing hyperparameter values.

Chapter 5

Results

In this chapter, a number of audio examples are presented in order to show the relative success of the various approaches being presented. Each audio example is evaluated according to the set of criteria outlined in Chapter 1. The examples are produced using the same audio file as input 5.1. The source material was chosen to be fairly simple, consistent, and short. This is to allow the reader to be able to easily recognize the input in the predicted audio. It can be difficult to tell what in the generated output sounds comes from the particular qualities of the input, and what are general tendencies of the model. By showing the models the same content, these distinctions can be made more easily.



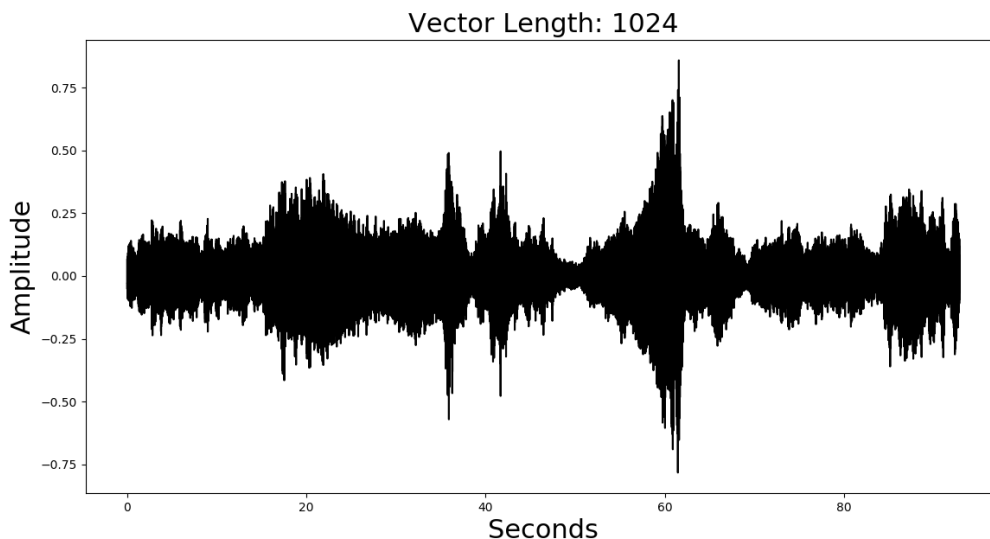
Audio Example 5.1. Input audio file used to both train models, and as the seeds during sampling for Audio Examples 5.2 through 5.12

5.1 Vector Approach

This algorithm performs the best on the speed criterion for evaluating models. These models can produce long outputs much quicker than the other algorithms. The drawback is that the outputs produced by this method tend to have more artifacts than other methods. Specifically, this approach tends to predict viable vectors that internally might not have any artifacts, but that do not necessarily smoothly connect to the previous or next vectors.

The result is a discontinuity or sudden change between adjacent vectors that is audible in the predictions.

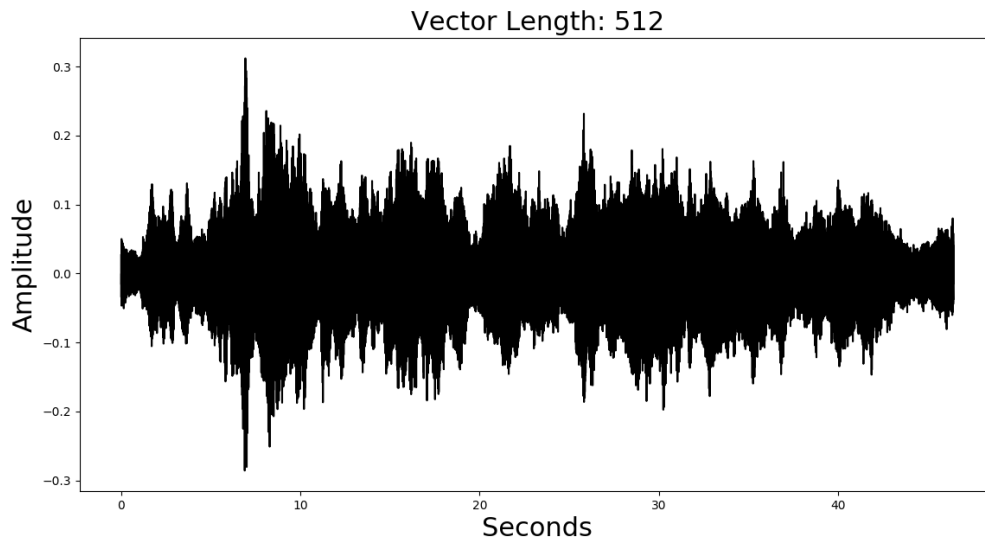
Changing the length of the input vectors dramatically changes the quality of the predictions. When the vectors are long, the frequency content in the sampled output changes very slowly over time. When the length of the input vectors is short, the predictions change more rapidly, but they tend to resemble the input audio less. This sampled output would score high on the evaluation criteria for resembling the input and not matching it too closely. However, it would score medium on the criteria regarding artifacts. This is because the artifacts that are present are not so egregious that the audio is not usable for musical purposes, but the artifacts are difficult to filter out without losing other positive qualities of the audio.



Audio Example 5.2. Vector approach using length 1024 audio vectors.

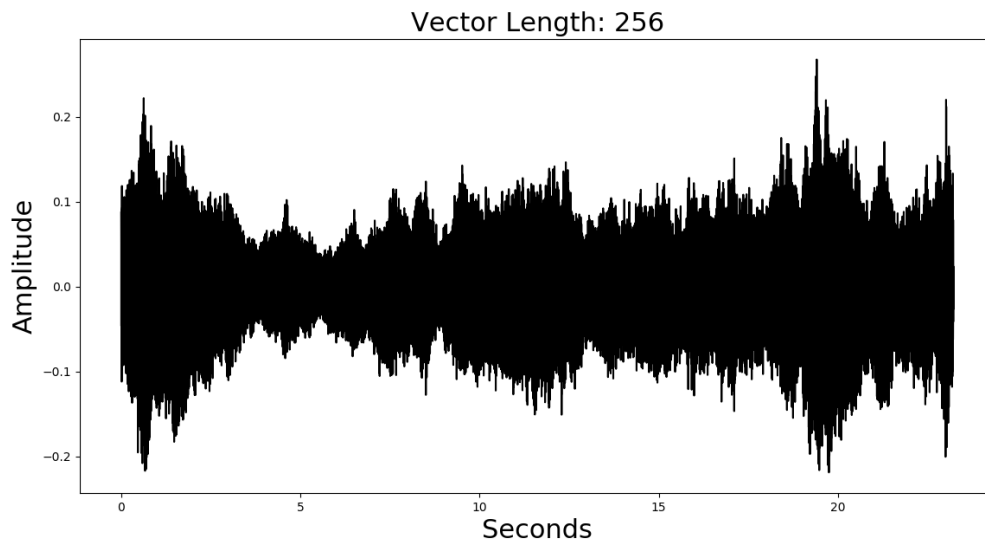
Audio Example 5.2 shows a prediction where the model predicted vectors of length 1024. There are not discontinuities in the waveform at the beginning of each new vector, but the frequency content does not flow smoothly across predictions. The audible effect is of a pulsating timbre that is reminiscent of granular or concatenative synthesis methods.

Audio Example 5.3 shows a prediction where the input vectors were of length 512. As with Audio Example 5.2, there are not discontinuities in the waveform, but the abrupt



Audio Example 5.3. Vector approach using length 512 audio vectors.

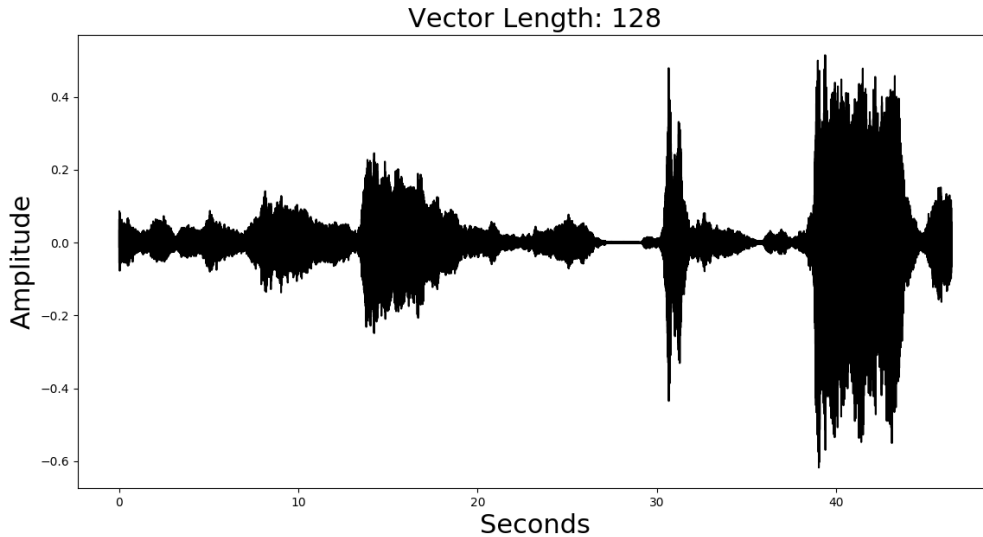
changes in content in the predictions that create the pulsating effect are still present, though they occur with twice the frequency. The large scale changes in frequency content also occur with increased frequency when compared with the previous example. This audio would achieve the same scores as Audio Example 5.2.



Audio Example 5.4. Vector approach using length 256 audio vectors.

Audio Example 5.4 shows a prediction with length 256 input vectors. Here some small discontinuities do occur. There is more consistency with regard to frequency content in

the prediction over time. Some sustained high frequency content is present. These sounds seem to be decoupled from the frequency content in the range of the fundamentals of the notes in the input audio. Separate onsets are not present in the prediction. Instead the pitches tend to occur simultaneously, and to blend gradually into each other. This audio would achieve the same scores as Audio Example 5.2.



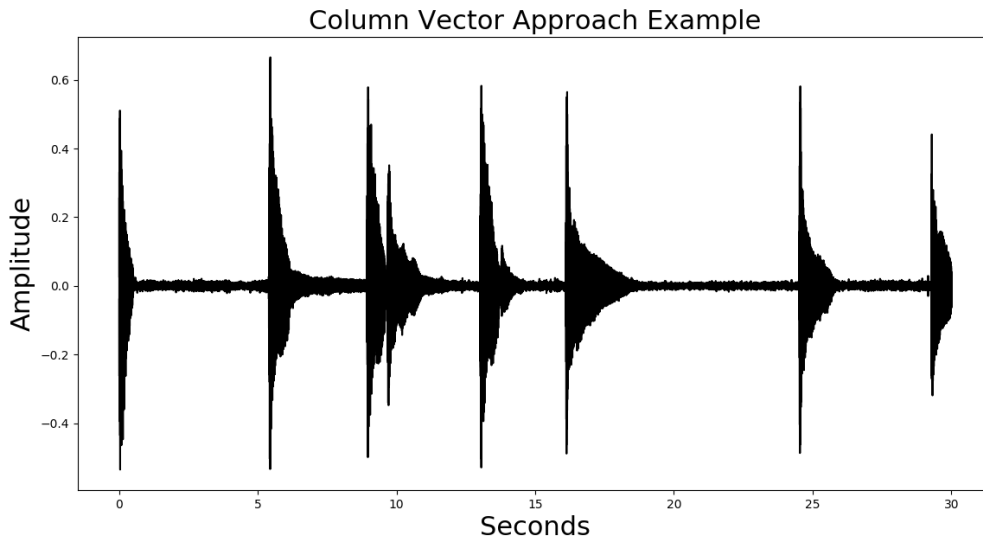
Audio Example 5.5. Vector approach using length 128 audio vectors.

Audio Example 5.5 shows a prediction using vectors of length 128. This prediction features a smoother waveform than when the vector length was set to 256, 512, or 1024. The amplitude of this prediction resembles the input data more closely. The frequency content however resembles the input less than the previous examples. The sustained high frequency content that was first mentioned in Audio Example 5.4 is also present here. This audio would score lower than the previous three Audio Examples on the criteria for artifacts resembling the input. This is because the high frequency content added to the sampled output is so different from the input data that it would be difficult to tell what the input was.

5.1.1 Column Vector Approach

Sampled outputs created using the column vector approach tend to score the highest on Criteria 2 and 3 but slightly worse on Criterion 1 compared with other methods. Specifically

models using this approach predict note-to-note transitions better than any other approach. Audio Example 5.6 shows a prediction made using this approach. There are clearly separate notes, as opposed to undifferentiated sustained sounds. This is significant because it shows the model understands large scale features about the input data. It would be simple for the model to learn some general rules about the transitions from one audio sample to the next. It could learn for instance that the difference between adjacent samples should never be more than a certain amount, or that if the previous three samples are rising consecutively, that the following sample should rise further still. The model here shows an understanding about the amplitude of the input data and how it tends to change over the course of hundreds of thousands of samples.



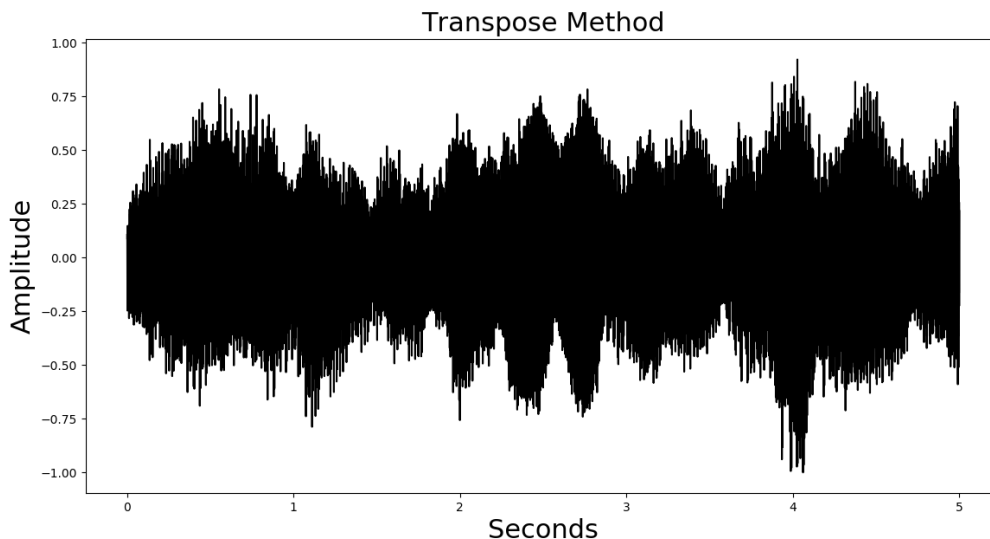
Audio Example 5.6. Prediction made using column vector approach with note level transitions. Noise can also be seen between the notes

As mentioned above, this prediction scores very high on Criteria 2 and 3 because it obviously resembles the input data but does not match it exactly. The pitches do not occur in the same order in the input data, and they do not appear with so much space between them in the input. It only scores medium on Criteria 1 because there is some low amplitude noise present in the prediction.

When dropout is used with this method, the predictions tend to score much higher on

all of the evaluation criteria (see Section 4.7). With LSTMs, dropout can be applied to the connections to the model’s inputs or the model’s outputs. In this case dropout on the input connections tends to drastically improve performance.

5.1.2 Transpose Approach

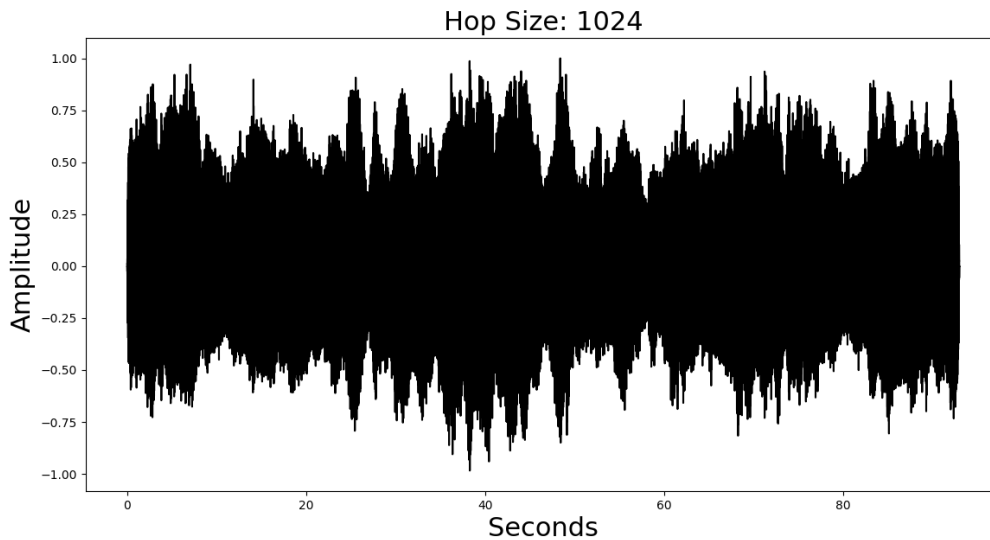


Audio Example 5.7. Prediction using transpose method.

Sampled outputs made using the transpose approach tend to not feature much noise. They tend to blend together much of the input data resulting in a fairly consistent sound. Audio Example 5.7 shows an example prediction made using this method. The pitches from the input tend to occur simultaneously rather than sequentially. There is some variation over time in the prediction, but it occurs mostly in the realm of amplitude and timbre, not in pitch. Using an input with restricted pitch content generally results in more pleasant results, since most of all of the pitches in the input will be heard simultaneously in the prediction.

5.1.3 Magnitude Spectrum Approach

The `hop_size` is the hyperparameter that controls difference between the index of the first samples in adjacent windows of audio before they are subjected to the Fourier Transform. Changing this value has a dramatic effect on the outputs predicted by the model. Audio Examples 5.8 through 5.12 show the waveforms for various settings of the



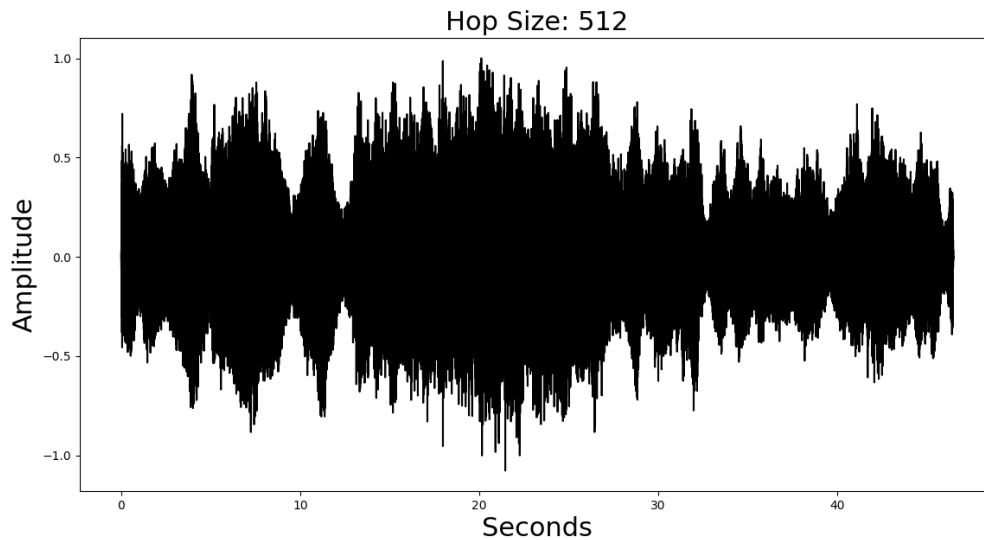
Audio Example 5.8. Example sampled output from the magnitude spectrum approach where `hop_size` is set to 1024 samples.

`hop_size`. For all of these audio examples, `window_size` was set to 1024.

In Audio Example 5.8 the `hop_size` is set to 1024. This means that there are no overlapping samples in adjacent windows. The audio does not resemble the input audio very closely. The timbre of the input audio is present in the prediction, but the amplitude is drastically changed. The pitch content of the input data is clearly present in the prediction, however the pitches do not occur separately as they do in the input. Instead the pitches occur simultaneously, blending together with each other to form a subtly evolving mass of sound.

Audio Example 5.9 demonstrates the effect of setting `hop_size` to 512. The general sound of this prediction is similar to Audio Example 5.8, but the pitches are much less blurred. There is much more variation in amplitude in this prediction compared with the previous. These do not correspond to changes in pitch as they did in the input audio. It appears the model learned that it is more plausible for its predictions to have variations in amplitude, but it did not learn to synchronize changes in pitch with changes in amplitude.

Audio Example 5.10 demonstrates the effect of setting `hop_size` to 256. The audio in this prediction is much clearer than in the preceding example. The pitch content of

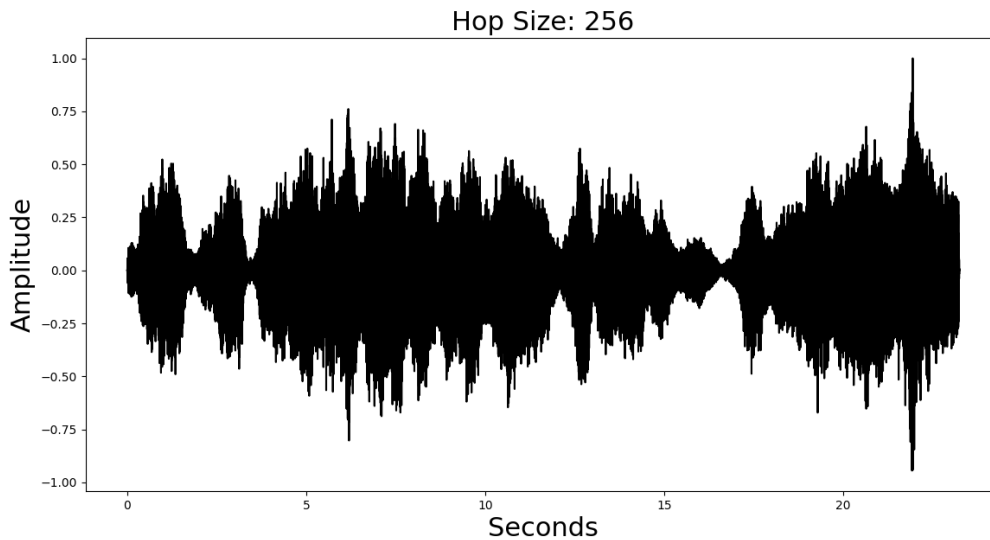


Audio Example 5.9. Example sampled output from the magnitude spectrum approach where `hop_size` is set to 512 samples.

the input is discernible in the audio. The changes in amplitude are more closely aligned with changes in pitch content. The changes in pitch are very slow compared with the input audio. The pitches are much less overlapped than in previous examples. There is a slight wavering in amplitude throughout this prediction. The effect is similar to amplitude modulation with a low frequency modulator signal, or a tremolo effect.

In Audio Example 5.11 the `hop_size` was set to 128. Here a passage from the input audio is clearly audible. The first 3 seconds of this prediction contain what appears to be the model having some uncertainty about which notes to predict next. Several pitches are begun and then abandoned. The tremolo effect is still present though the frequency of the modulation is somewhat higher.

Audio Example 5.12 was created using a `hop_size` of 56. The first 12 seconds of this prediction resemble the input very closely. Approximately 14 seconds in, the model begins to predict a louder passage of audio that is marked by the tremolo effect mentioned with the previous examples. The pace of the transitions between pitches in this example matches the input very closely. The frequency of the tremolo effect near second 14 is much faster



Audio Example 5.10. Example sampled output from the magnitude spectrum approach where `hop_size` is set to 256 samples.

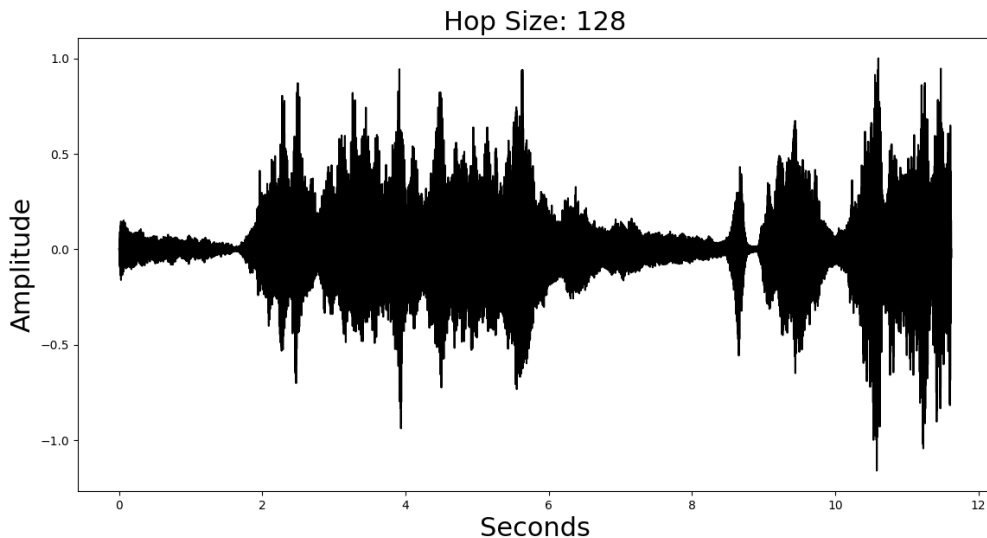
than any changes of amplitude in the input. Hence the tremolo effect seems to be an artifact from the algorithm, not some feature learned from the data and applied in some other context during a prediction.

Table ?? shows the evaluation for Audio Examples 5.8 through 5.12. On the whole these audio predictions score very high. Audio Example 5.8 only scored Medium on Criterion 2 because the pitch content was blended together so consistently almost all the pitches present in the input are heard simultaneously during much of the prediction. There are some changes over time in the pitch content of the prediction. For the score to be low, the pitch content would need to be almost completely consistent for the duration of the audio.

Audio Examples 5.11 and 5.12 scored medium on Criterion 3 because they contain extended passages that match the pitch content of the input audio exactly. Timbrally the predictions are different enough that they do not score low in this Criterion. However, more variation in the pitch content is required for a score of High.

5.1.4 Model Evaluation

Following significant development time on all of the approaches for over six months, none of the finalized approaches scored low on any of the model evaluation criteria (see



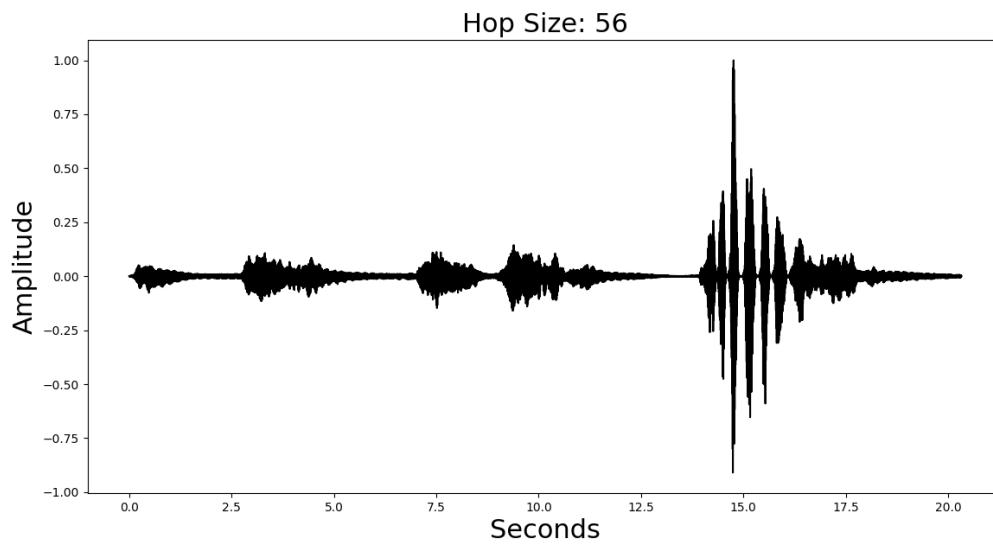
Audio Example 5.11. Example sampled output from the magnitude spectrum approach where `hop_size` is set to 128 samples.

Hop Size	Criterion 1	Criterion 2	Criterion 3
1024	High	Medium	High
512	High	High	High
256	High	High	High
128	High	High	Medium
56	High	High	Medium

Table 5.1. Table showing evaluation for Audio Examples 5.8 through 5.12

Section 1.4.3). The biggest concern for these models is the criterion for evaluating if a model is prohibitively slow. The models used to create Audio Examples 5.2 through 5.5 and 5.8 through 5.12 scores high on this criterion. The magnitude spectrum approach is slightly slower than the vector approach because resynthesizing the audio examples adds some overall time to the process.

The other approaches take significantly more time to sample outputs. This is due to the fact that the models only predict a single audio sample at each time step. Depending on the hyperparameters used, it might take as much as 12 hours to sample 5 minutes of audio. It is difficult to imagine a model that is significantly slower than this being practical to use in the present times. This is because often several trials need to be run to hone in on an effect set of hyperparameters. If each test case takes hours to generate, it could take



Audio Example 5.12. Example sampled output from the magnitude spectrum approach where `hop_size` is set to 56 samples.

days to create a very small amount of audio.

Chapter 6

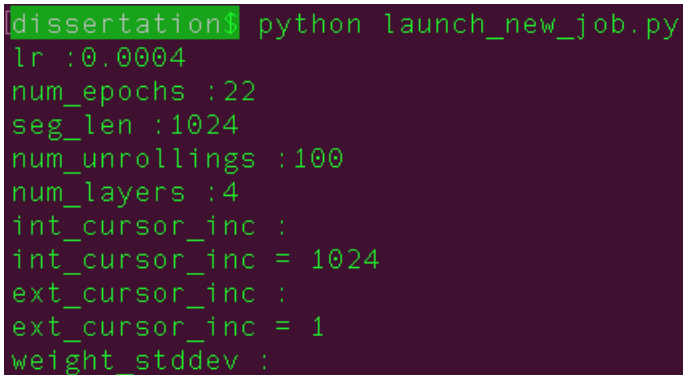
Applications

6.1 User Interface

Two interfaces have been created to allow users to make predictions easily. In both cases the job of the interface is to make the process of choosing and entering hyperparameters simpler. The alternative is to enter the values into the model file directly. When one of the interfaces is used, the hyperparameters are specified in a JSON file external to the model. This file can then be imported and each of the hyperparameters can be set to the user's preference.

6.1.1 Command Line Interface

The command line interface presents the user with a series of prompts. These prompts each correspond to a hyperparameter that needs to be set for the model to run correctly. Figure 6.1 shows the command line interface being used. The user is prompted to select each hyperparameter. If the user enters a value, it is sanitized. If the user does not enter a value the default setting is used.



```
dissertation$ python launch_new_job.py
lr :0.0004
num_epochs :22
seg_len :1024
num_unrollings :100
num_layers :4
int_cursor_inc :
int_cursor_inc = 1024
ext_cursor_inc :
ext_cursor_inc = 1
weight_stddev :
```

Figure 6.1. Screenshot of using the command line interface to enter hyperparameters.

6.1.2 Graphical Interface

The graphical interface is a higher level tool for setting hyperparameters. This application allows the user to enter the settings individually, as with the command line interface, or to choose from a number of presets. Finally the application presents the user with all the chosen hyperparameters as shown in Figure 6.2. Final changes can be made to the hyperparameters at this point before finalizing the settings. When the user finalizes the settings, a JSON file is constructed from the choices and is written to disk. That JSON file can then be imported and used to train a model just as the JSON file created by the command line interface would.

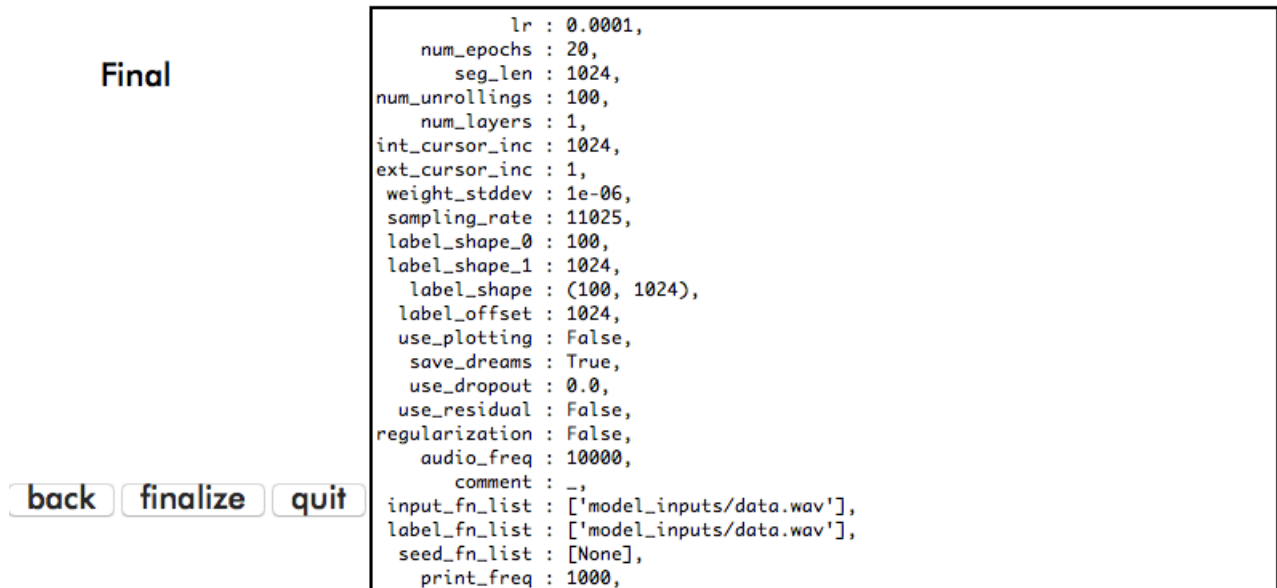


Figure 6.2. Screenshot of using the graphical interface to enter hyperparameters.

6.2 Compositions

Several musical compositions are presented here. They are discussed in terms of how they demonstrate some of the concepts outlined in previous sections.

6.2.1 Predictions in Fixed Media Compositions

The piece *The moon who makes the birds dream in the trees* is a demonstration of how predictions created using the column vector method can be used as the source material for

a fixed media composition. The input data used to create the predictions for this piece was the first thirty seconds of Claude Debussy's *Clair de lune*. This passage of music features a restricted set of pitches. This kind of musical content lends itself to this application because the outputs can be easily superimposed to form polyphony as shown in Figure 6.3.

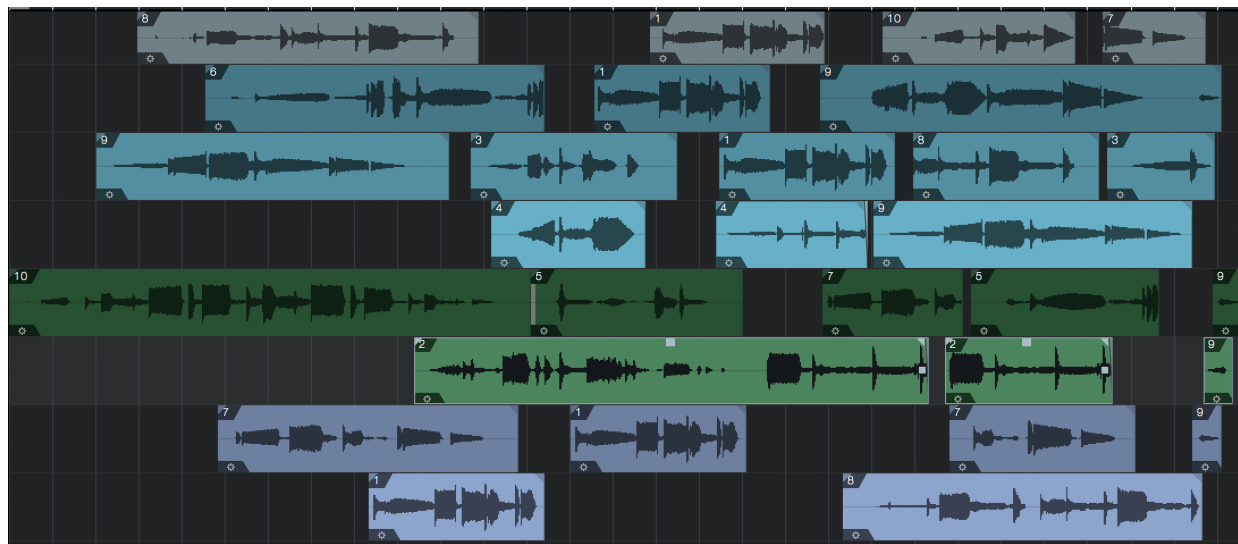


Figure 6.3. Example of how sampled predictions are overlapped to form polyphony in *The moon who makes the birds dream in the trees*

The sampled predictions were each thirty seconds long. All the predictions were used in their entirety except for the final few seconds of the composition where some pitches were manually repeated. The predictions were largely unchanged. In some situations, a fade in and fade out were added, and some subtle filtering was added to reduce the noise present in the predictions.

One moment around one minute and forty seconds into the composition was arranged so that all occurrences of a certain chord were aligned so that it happened at the same time. Aside from this moment and the conclusion where a specific chord was chosen, the predictions were overlapped in various ways to create a denser sound.

The sound of the piece is somewhat surreal because the sound of the piano is recognizable but stretched in an unnatural way. The pacing of musical events is also somewhat surprising. The process of predicting of audio can be described as akin to the model dreaming. The sounds were chosen and arranged to take advantage of this metaphor. The

predictions are harmonically meandering on their own. By reordering the pitches, the predictions remove any semblance of teleology. When they combine a waywardness results that is meant to sound dreamlike.

The dreamlike rhetoric in the sampled predictions calls to mind the work of John Cage. There is an obvious connection to his indeterminate works where the appearance of goal oriented progression is necessarily coincidental. His more serene composed works and his later indeterminate works resemble *The moon who makes the birds dream in the trees* more obviously such as *Four* or the third movement of his *String Quartet in Four Parts*. The kind of free rhetoric he was after and this piece tries to evoke is aptly described in a quote from the film *Listen* about Cage:

“[...] I love sounds just as they are, and I have no need for them to be anything more than what they are. I don’t want them to be psychological. I don’t want a sound to pretend that it’s a bucket or that it’s president or that it’s in love with another sound. I just want it to be a sound.” [43]

6.2.2 Real Time Processing Predictions

It can be difficult to use sampled predictions in live music because the process of making a prediction using LSTMs is time consuming. The impetus for the piece *Like two strings speaking in sympathy* was to try to devise a way to manipulate these sounds in real time. An application was created in Max that analyzes input audio from a live guitar player or a recording of a live performance, and returns selections from a number of audio predictions. The result is an application that appears to contain some limited intelligence, listening to the input and responding with its own sounds. The sound is similar to sympathetic resonance, in that the guitarist plays a note and the computer responds with a swelling sound that features similar pitch content.

The sampled predictions used were made with the transpose approach (see Section 5.1.2). The input material for the model was the same as what the performer plays during

the performance. The method of sampling created output audio that features sustained passages that have the timbre and pitch content of the input audio, but that have fairly consistent amplitude. Because there is little change over time in the predictions, the real-time application takes as its primary effect, applying amplitude envelopes to the predictions.

A bank of bandpass filters comprises the analysis portion of the application. The input audio goes to each of the filters, which each output a time-varying output signal that is high when the center frequency of the filter is present in the input audio and low when the frequency is absent. All of the predictions from the LSTM are being looped with their amplitude set to zero. The output from the filter banks is added to the amplitude of the looping audio. The effect is that when the application hears a certain frequency, it will raise the volume on a looping passage of audio created by the LSTM and then lower the volume when it no longer hears the frequency.

The piece has three separate conceptual layers of audio. The first is the live input or recorded live performance. The second is the swelling sounds described above. The final layer functions similarly to the second layer, but there is some amplitude modulation and an envelope applied to the audio. The frequency of the amplitude modulation is proportional with the amplitude of the envelope so that as the audio gets louder, the tremolo effect on the audio gets faster.

Pieces from the genre of music for performer with electronics often focus on the interaction between the two agents. An early example is the opening of *Synchronisms No. 6* where the envelope of a piano note is inverted by the tape part. The effect is of the note uncannily getting louder rather than decaying as it normally would. A more contemporary example by George E. Lewis is *Voyager* which is an autonomous musical agent that listens to a performer improvise and generates music in response. *Like two strings speaking in sympathy* occupies a space somewhere in between these two extremes. It can listen to input and respond, but the way that it responds is fixed.

6.2.3 Magnitude Spectrum Predictions

The piece *Water that doesn't move stagnates* showcases predictions made using magnitude spectra. Specifically the breadth of different sounds that can be created using different hop sizes is explored. The piece begins with a recording of the source material used to make the predictions that form the content for the rest of the piece.

The first section of the piece features several predictions created using non-overlapping magnitude spectra. As described in Section 5.1.3 this results in sounds where the pitch content is blurred and overlapped. The sound somewhat resembles a tanpura, an Indian drone instrument. In the subsequent sections of the piece, predictions made using increasingly small hop sizes are used. The effect is of increasing clarity and definition in the sound over the course of the piece.

Some predictions created using the column vector approach are also used in this piece. These sounds were created without any dropout. As mentioned in Section 5.1.1, this method tends to lead to predictions that score higher on all the evaluation criteria when dropout is applied to the input connections of the LSTM. The sounds here closely resembles the feedback that occurs with distorted electric guitars.

The guitar part was composed using a loose contrapuntal process devised to give the music a stagnant sound. There are two voices throughout. At any point, one voice may move freely and the other voice will accompany it by largely alternating between two pitches. All imperfect consonances are avoided. This results in a situation where dissonances are followed by dissonances or leapt away from. In all other situations, major or minor seconds are preferred over other intervals. The process calls to mind the work of Arvo Pärt. His much stricter process known as Tintinnabuli involves accompanying a step-wise melody with notes from the tonic triad. There are a number of methods for deciding which member of the chord are chosen, and these methods are adhered to strictly.

6.2.4 Predicting from External Seeds

The piece *What comes is better than what came* was composed using several sampled predictions using the magnitude spectrum approach. Some of the sustained sounds heard in the piece were created using time stretching. The sampled predictions were created via a novel process. The model trained normally on a dataset that featured a long sequence of dissonant chords. The seed used to make the sampled predictions was the guitar melody heard in the piece. The resulting predictions are colored by the training data set. The model is presented with a single line melody and asked to make predictions about it having only ever heard dissonant chords. The predictions feature fragments of the melody but there are additional frequencies from the chords present in the predictions.

The effect of this is similar to a theme and variations where the model listens to the theme which is in this case the single line melody and composes a series of variations. The style of these variations is dictated by the content of the training data.

An inference model was made for this piece that is used only for sampling. A model is first trained and its weights are saved. They can then be loaded into the inference model. Since no training occurs with this model, the samples can be produced as quickly as possible. A patch made in Max was created that records the input from the performer and writes it to disk periodically. The inference script watches for new files in a certain directory. When a new file appears the script processes it, shows it to the model as a seed, and writes the predictions to disk. The patch also watches for new files in a different directory where the sampled predictions are stored. It then reads them in and plays them in a staggered manner.

6.3 Summary

The models were successful in producing sampled outputs that are useful in a musical context. Though considerable time and effort was required to arrive at the sets of hyperparameters and algorithms that were used. Currently the biggest limitations for use of LSTMs to generate music is the time required for the model to produce the audio and the

tendency for the model to produce noisy outputs.

In the future, it is likely that computer hardware will improve to the point that the samples are no longer prohibitively slow to produce. This would afford musicians a number of additional opportunities for composition. For instance, in this hypothetical scenario, a musician could improvise musical passages, have a model sample an output in response in real time.

One other significant limitation of the technology as it exists today is that the processes of sampling and training are only related indirectly. As it stands now there is no way to train a model to sample better outputs. There are only ways to improve training to predict audio. One potential route to remedy this situation is to explore different types of neural networks that are explicitly trained to generate new data that is different from the data they are trained on. Generative adversarial networks are a promising example of this type of situation (see section 1.3).

Chapter 7

Conclusions

7.1 Approaches to Sampling

To produce output audio, a model is sampled from by iteratively making predictions based on a seed, saving the predictions, and updating the seed to include the predictions.

7.1.1 Vector Approach

In the first approach, the model predicts a vector for each input vector. This approach has the benefit of being the fastest approach. The sampled outputs from this approach tend to feature windowing artifacts.

7.1.2 Magnitude Spectrum Approach

The second approach uses magnitude spectrum windows as the input and output data. This approach has the model predict one window for each input window. The sampled predictions are then resynthesized via a channel vocoder. This approach has the benefit of being only slightly slower than the vector approach. It has the drawback that the timbre of the predictions is distorted by the vocoder because the phase information is not included in the input data.

7.1.3 Column Vector Approach

The model can also be asked to predict one sample for each input vector. Sampled predictions using this approach tend to feature more detail than those created with any of the other approaches. Only this approach produces audio that sounds like something a human would make. If a person is presented with a series of notes and is asked what he or she thinks comes next, they would likely guess that it is more notes and not a single sustained sound. Alternatively if a person is presented with a long sustained sound and is asked to predict what he or she thinks comes next, they would likely guess more of the same sustained sound. Some of the sampling methods result in sampled outputs that invert this relationship. Models sampled from in these manners will for instance, predict a sustained sound after being shown a series of notes. The column vector approach does not do this.

Instead it predicts a new series of notes after being shown a series of notes, for instance. The drawbacks of this approach are that it is rather slow and the resulting audio often features low amplitude noise.

7.1.4 Transpose Approach

The final sampling approach has the model predict one audio sample for each input matrix. This approach has the benefit of being reliable and relatively noise free. Sampled outputs from this approach almost always feature all of the frequencies in the seed in an overlapping fashion. This property allows for the creation of sampled outputs that are rather predictable in pitch content and amplitude. The drawbacks of this approach are that it is very slow and there is little change over time in the sampled outputs.

Figure 7.1 illustrates how these approaches can be compared to each other. The spectrum from unpredictable to predictable is meant to indicate how easy it is to control the frequency content of the sampled outputs. The transpose method is the easiest to control in this manner. This is because the sampled outputs often feature all of the frequency content of the input simultaneously. To sample an output with particular pitches, a seed can be chosen that only contains the desired pitches. The output will contain the desired pitches, but blended together and overlapped. The column vector approach or the magnitude spectrum with a `hop_size` of 1024 are quite difficult to control in this manner. The sampled outputs in these cases tend to feature pitch content from different portions of the input data but which portions will show up are difficult to know in advance.

The other axis in Figure 7.1 describes the manner in which frequency content tends to change over time in the sampled outputs. An analogy is used to images to describe the situation succinctly. In a focused picture the individual shapes appear normally with crisp and clearly defined edges. By analogy in audio a sampled output that is focused would have clear amplitude envelopes and clear transitions between different sets of frequencies that correspond to the amplitude envelopes. Samples produced with the magnitude spectrum approach with a `hop_size` of 56 show this property as do samples produced with the

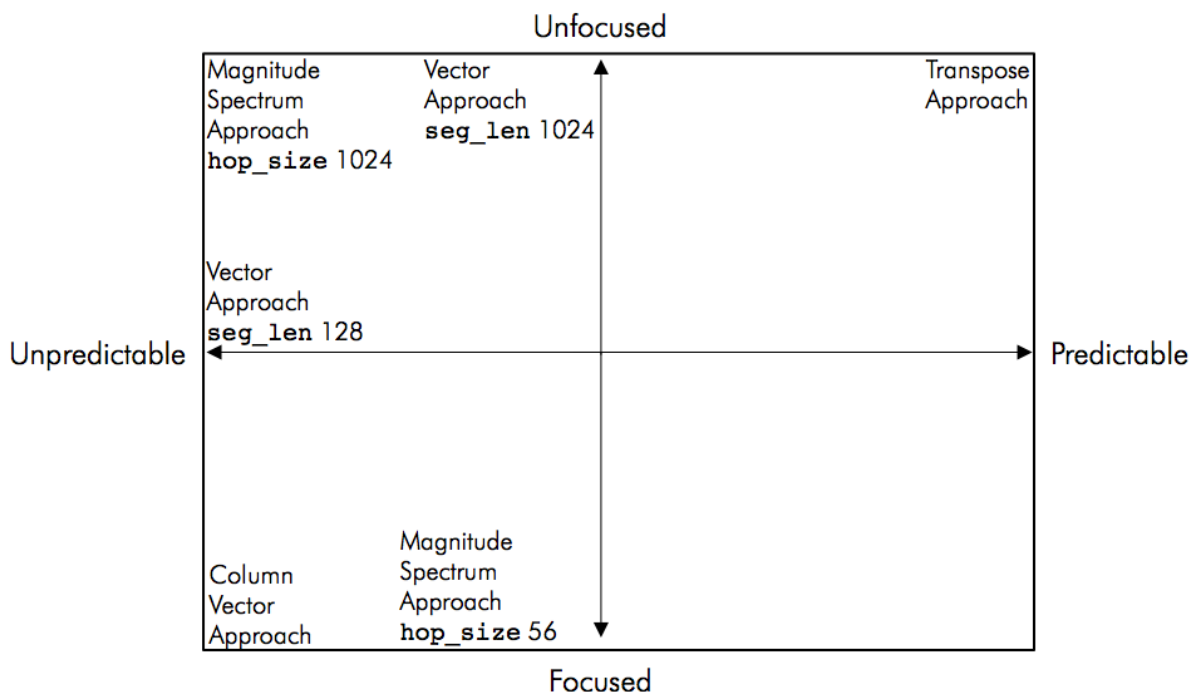


Figure 7.1. Figure showing how the different approaches to sampling can be compared to each other according to how clear the pitches in the input are and how easy it is to control the overall sound of the sampled outputs.

column vector approach. In a unfocused or blurry picture the color information for one shape would be spread out over a larger area and would overlap the information for its neighboring shapes. In audio by analogy an unfocused sampled output would tend to feature pitch content without clear beginnings and endings. These samples would tend to lack a clear amplitude envelope. The vector approach with `seg_len` set to 1024 tends to produce samples that have this quality. There are some transitions between different sets of pitches, but there is not a clear beginning or ending to these transitions. The vector approach with `seg_len` set to 128 has somewhat more definition but there is some blending of adjacent simultaneities.

7.2 Sequential Versus Concurrent Predictions

There are two variations on generating outputs from LSTMs. The first is somewhat simpler to understand. This algorithm executes a number of iterations of training and periodically pauses training to generate an output (see Figure 3.3). The second method executes one iteration of prediction for each iteration of training (see Figure 3.4). In

general the concurrent method of sampling outputs is preferable to the sequential method for practical use in music making. The results from concurrent sampling tend to score higher on the evaluation criteria. The sequential method results will sometimes get stuck repeating a single short passage. By changing the weights of the model during sampling, the concurrent method fixes this problem.

In practice, both methods can be used at once. The sequential method sampled outputs are helpful as they can be produced more quickly early on during training. When trying to choose hyperparameters for a model, sampling some short predictions after a small amount of training can give valuable insight into how the choice of hyperparameters affects the quality of the sampled predictions.

7.3 Changing Significant Hyperparameters

For the vector approach and the magnitude spectrum approach, there are some hyperparameters greatly affect the content of the sampled predictions. For the vector approach if the value of `seg_len` is large, the sampled outputs will evolve slowly and will tend to feature a fairly consistent amplitude. When it is set lower the amplitude will change tend to fluctuate more and resemble in the input audio more closely. The pitch content of the sampled outputs will also tend to change in a manner that is more like the input audio.

The `hop_size` has a similar effect when using the magnitude spectrum approach. When it is high the sampled outputs tend to feature ambiguous passages of multiple pitches simultaneously. When it is lower the sampled outputs tend to feature sequences of notes from the input data.

7.4 Musical Applications

Four musical compositions are presented that demonstrate how predictions made by the models can be used. They showcase the various approaches to sampling. They also show how the predictions can be manipulated and demonstrate how the technology can be extended for artistic purposes. The sampled predictions are used as source material for fixed media compositions and manipulated during live performance.

The models were successful in producing audio that is desirable for use in musical

compositions. The sounds feature a surreal dreamlike quality that is highlighted in the pieces written using them. The biggest drawback of working with these models currently is the time it takes to generate sampled outputs. The enormous variety of sounds that can be produced from a very limited amount of input is promising for future musical explorations.

References

- [1] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, “End to End Learning for Self-Driving Cars,” *CoRR*, vol. abs/1604.07316, 2016. [Online]. Available: <http://arxiv.org/abs/1604.07316>
- [2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, “Mastering the Game of Go Without Human Knowledge,” *Nature*, vol. 550, Oct. 2017.
- [3] G. Tesauro, D. Gondek, J. Lenchner, J. Fan, and J. M. Prager, “Analysis of Watson’s Strategies for Playing Jeopardy!” *CoRR*, vol. abs/1402.0571, 2014. [Online]. Available: <http://arxiv.org/abs/1402.0571>
- [4] R. Wolf and J. Platt, “Postal Address Block Location Using a Convolutional Locator Network,” pp. 745–752, 01 1993, visited on May 23, 2018. [Online]. Available: <https://papers.nips.cc/paper/856-postal-address-block-location-using-a-convolutional-locator-network.pdf>
- [5] A. Kongthon, C. Sangkeettrakarn, S. Kongyoung, and C. Haruechaiyasak, “Implementing an Online Help Desk System Based on Conversational Agent,” in *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, ser. MEDES ’09. New York, NY, USA: ACM, 2009, pp. 69:450–69:451. [Online]. Available: <http://doi.acm.org/10.1145/1643823.1643908>
- [6] N. Justesen, P. Bontrager, J. Togelius, and S. Risi, “Deep Learning for Video Game Playing,” *CoRR*, vol. abs/1708.07902, 2017. [Online]. Available: <http://arxiv.org/abs/1708.07902>
- [7] A. Mordvintsev, C. Olah, and M. Tyka, “Inceptionism: Going deeper into neural networks,” *Google Research Blog. Retrieved June*, vol. 20, no. 14, p. 5, 2015, visited on May 23, 2018.
- [8] P. Isola, J. Zhu, T. Zhou, and A. A. Efros, “Image-to-Image Translation with Conditional Adversarial Networks,” *CoRR*, vol. abs/1611.07004, 2016. [Online]. Available: <http://arxiv.org/abs/1611.07004>
- [9] L. A. Gatys, A. S. Ecker, and M. Bethge, “A Neural Algorithm of Artistic Style,” *CoRR*, vol. abs/1508.06576, 2015. [Online]. Available: <http://arxiv.org/abs/1508.06576>
- [10] A. M. Elgammal, B. Liu, M. Elhoseiny, and M. Mazzone, “CAN: Creative Adversarial Networks, Generating ”Art” by Learning About Styles and Deviating from Style Norms,” *CoRR*, vol. abs/1706.07068, 2017. [Online]. Available: <http://arxiv.org/abs/1706.07068>

- [11] K. He, X. Zhang, S. Ren, and J. Sun, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” *CoRR*, vol. abs/1502.01852, 2015. [Online]. Available: <http://arxiv.org/abs/1502.01852>
- [12] S. M. Stigler, “Gauss and the Invention of Least Squares,” *Ann. Statist.*, vol. 9, no. 3, pp. 465–474, 05 1981. [Online]. Available: <https://doi.org/10.1214/aos/1176345451>
- [13] A. Kurakin, I. J. Goodfellow, and S. Bengio, “Adversarial Examples in the Physical World,” *CoRR*, vol. abs/1607.02533, 2016. [Online]. Available: <http://arxiv.org/abs/1607.02533>
- [14] T. Walsh, “Elon Musk is Wrong. The Singularity Won’t Kill Us All.” [Online]. Available: <http://www.wired.co.uk/article/elon-musk-artificial-intelligence-scaremongering>
- [15] R. Kurzweil, *The Singularity Is Near: When Humans Transcend Biology*. New York: Viking, 2005.
- [16] J. Hendler, “Avoiding Another AI Winter,” *IEEE Intelligent Systems*, vol. 23, pp. 2–4, 03 2008. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MIS.2008.20
- [17] W. S. McCulloch and W. Pitts, “A Logical Calculus of the Ideas Immanent in Nervous Activity,” *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, Dec 1943. [Online]. Available: <https://doi.org/10.1007/BF02478259>
- [18] G. H. Yann LeCun, Yoshua Bengio, “Deep Learning,” *Nature*, vol. 521, pp. 436–444, 2015. [Online]. Available: <http://dx.doi.org/10.1038/nature14539>
- [19] C. Olah, “Understanding LSTM Networks,” <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>.
- [20] Z. C. Lipton, “A Critical Review of Recurrent Neural Networks for Sequence Learning,” *CoRR*, vol. abs/1506.00019, 2015. [Online]. Available: <http://arxiv.org/abs/1506.00019>
- [21] S. Hochreiter, Y. Bengio, and P. Frasconi, “Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies.(2001),” in *Field Guide to Dynamical Recurrent Networks*, J. Kolen and S. Kremer, Eds. IEEE Press, 2001.
- [22] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computing*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: <http://dx.doi.org/10.1162/neco.1997.9.8.1735>
- [23] R. Jozefowicz, W. Zaremba, and I. Sutskever, “An Empirical Exploration of Recurrent Network Architectures,” in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML’15. Lille, France: JMLR.org, 2015, pp. 2342–2350. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3045118.3045367>

- [24] A. Graves, “Generating Sequences with Recurrent Neural Networks,” *CoRR*, vol. abs/1308.0850, 2013. [Online]. Available: <http://arxiv.org/abs/1308.0850>
- [25] A. Karpathy, “The Unreasonable Effectiveness of Recurrent Neural Networks,” <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>, 2015, visited on May 23, 2018.
- [26] A. Karpathy, J. Johnson, and F. Li, “Visualizing and Understanding Recurrent Networks,” *CoRR*, vol. abs/1506.02078, 2015, visited on May 23, 2018. [Online]. Available: <http://arxiv.org/abs/1506.02078>
- [27] A. Roberts, C. Resnick, D. Ardila, and D. Eck, “Audio Deepdream: Optimizing raw audio with convolutional networks,” *Google AI Research Blog*, 2016, visited on May 23, 2018. [Online]. Available: <https://18798-presscdn-pagely.netdna-ssl.com/ismir2016/wp-content/uploads/sites/2294/2016/08/ardila-audio.pdf>
- [28] E. Grinstein, N. Q. K. Duong, A. Ozerov, and P. Pérez, “Audio Style Transfer,” *CoRR*, vol. abs/1710.11385, 2017. [Online]. Available: <http://arxiv.org/abs/1710.11385>
- [29] D. Cope, *Virtual music: Computer synthesis of musical style*. Boston, MA: MIT press, 2004.
- [30] —, “Ars Ingeniero,” *Santa Cruz, CA: CreateSpace*, 2012.
- [31] —, *Computer Models of Musical Creativity*. Boston, MA: MIT Press, 2005.
- [32] D. Cope and M. J. Mayer, *Experiments in Musical Intelligence*. AR editions, Madison, WI, 1996, vol. 12.
- [33] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative Adversarial Networks,” *CoRR*, 2014. [Online]. Available: <https://arxiv.org/abs/1406.2661>
- [34] C. Donahue, J. McAuley, and M. Puckette, “Synthesizing Audio with Generative Adversarial Networks,” *CoRR*, vol. abs/1802.04208, 2018. [Online]. Available: <http://arxiv.org/abs/1802.04208>
- [35] E. Waite, “Generating Long-Term Structure in Songs and Stories,” Web site, 2016, visited on May 23, 2018. [Online]. Available: <https://magenta.tensorflow.org/2016/07/15/lookback-rnn-attention-rnn/>
- [36] C. Kidd, S. T. Piantadosi, and R. N. Aslin, “The Goldilocks Effect: Human Infants Allocate Attention to Visual Sequences That Are Neither Too Simple Nor Too Complex,” *PLOS ONE*, vol. 7, no. 5, pp. 1–8, 05 2012. [Online]. Available: <https://doi.org/10.1371/journal.pone.0036399>
- [37] R. Shwartz-Ziv and N. Tishby, “Opening the Black Box of Deep Neural Networks via Information,” *CoRR*, vol. abs/1703.00810, 2017. [Online]. Available: <http://arxiv.org/abs/1703.00810>

- [38] S. K. Kumar, “On Weight Initialization in Deep Neural Networks,” *CoRR*, vol. abs/1704.08863, 2017. [Online]. Available: <http://arxiv.org/abs/1704.08863>
- [39] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: <http://arxiv.org/abs/1512.03385>
- [40] A. Krogh and J. A. Hertz, “A Simple Weight Decay Can Improve Generalization,” in *Proceedings of the 4th International Conference on Neural Information Processing Systems*, ser. NIPS’91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 950–957. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2986916.2987033>
- [41] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. [Online]. Available: <http://jmlr.org/papers/v15/srivastava14a.html>
- [42] T. van Laarhoven, “L2 Regularization versus Batch and Weight Normalization,” *CoRR*, vol. abs/1706.05350, 2017. [Online]. Available: <http://arxiv.org/abs/1706.05350>
- [43] M. Sebestik, “Listen = [Ecoute],” Film: ARTE France Developpement, Paris, France, 1992.

Vita

Andrew Pfalz completed his bachelor's degree in music composition at Florida State University in 2011. He went on to complete a master's degree in music composition at East Carolina University in 2014. He anticipates graduating from Louisiana State University in 2018.