TensorSynth

Intelligent Granular Synthesis with SuperCollider and Tensorflow

John DeBlase - CUNY SPS Spring 2017

Abstract

TensorSynth is a generative music intelligence system written in SuperCollider and Python. The prototype application implements the Word2vec natural language processing model using the Tensorflow framework in order to generate a series of musical events based on a unique grammar derived from a user's improvisational style. A granular synthesis instrument was chosen as the musical interface to model this unique grammar due to both the highly expressive and large variety of sounds that can be produced with a small set of parameters. The current version of the software was developed as a simple non-real time system for research purposes and proof of concept. Consideration was made in the design to use this prototype as a jumping off point for a more robust system capable of efficiently producing embedded vector space models of musical improvisation in real time using a network event-driven model.

1 Background: Machine Learning and Music

With the advent of modern computer processors, the application of various machine learning and algorithmic techniques for musical composition and analysis has been a source of ongoing development for both musical researchers and computer scientists. Papadopoulus and Wiggins (1999) outline several distinct areas of artificial intelligence research in music which include mathematical models, grammars, knowledge or rule based systems, learning systems and hybrid systems.[1]

Older yet notable examples of such systems are Cope's *Experiments in Musical Intelligence (EMI)* which generates compositions using a rule based transition network and a grammar graphs derived from an analysis of an input corpus.[2] Hild, Fuelner, and Menzel (1991) built *Harmonet*, a hybrid system which combines simple neural networks with a rule based algorithms designed specifically for the task of harmonizing Bach chorales.[3] A more modern application developed by Chan and Potter(2006) called *Automated Composer of Style Sensitive Music II (ACSSM II)* uses genetic algorithms to stylistically re-arrange material found in an input corpus.

Current research in the field tends to focus on using recurrent neural networks (RNNs) and long short-term memory neural networks (LSTMs) to attempt to model "style" and melodic content by finding and tracing patterns throughout a corpus. Eck and Schmidhuber (2002) used LSTMs to generate melodies from common 12-bar blues and jazz patterns.[4] The availability of

---

[1] Papadopoulos et. al "AI methods for algorithmic composition: A survey, a critical view and future prospects."

[2] David, Cope. "Experiments in Music Intelligence."

[3] Hild et. al. "HARMONET: A neural net for harmonizing chorales in the style of JS Bach."

[4] Eck et al. "A first look at music composition using lstm recurrent neural networks."

Python frameworks such as Tensorflow and scikit.learn make these types of complicated neural networks at the fingertips of the average developer or researcher. The Google Brain Team very recently released *Magenta*, a program that uses RNNs powered by Tensorflow to produce simple melodies using LSTMs from an input corpus of MusicXML files.[5]

2 Towards a Real-Time System and a Language Model

There are countless other techniques and areas of research related to machine learning in music, the details of which are beyond the scope of this report.[6] However, there are several fundamental characteristics common to almost all machine learning approaches that I have found which impede on the overall usefulness of implementing such systems in practice for the modern improvising musician, and which I hope to overcome in the development of TensorSynth.

One such limitation is the over-reliance on predetermined forms and rules of conventional musical systems. Modern improvisers tend to have their own unique approaches to their instruments that may or may not be informed by past traditions. Furthermore, in the realm of electronic music there are almost no limitations on what sounds can make up a performance, and no common language structure in place (such as western notation) to rely upon for generating such a performance. Since TensorSynth is being designed with the improviser in mind, it will be important to develop an API that can properly encode the uniqueness of the individual's playing style into the data for training.

An unfortunate characteristic that seems to be common among musical machine learning applications is the simplicity of their output. Many of the applications described above are designed to only produce a generic melodic sequence or harmonic realization based on some corpus of a specific composer, or only reproduce rhythmic sequences within a pre-described time grid that adhere to a specific style. While these are highly useful for analysis, such systems will not work in a real time model on a synthesizer with a high degree of sonic and spatial permeability. Therefore TensorSynth's data model needs to be able to encode a high variety of musical characteristics, including but not limited to pitch, timbre, rhythmic variation, amplitude and duration.

Another limitation is the non-real time nature of many machine learning applications. Deep neural network models are often trained for long time intervals on extremely large datasets in order to increase accuracy. Once trained these models can be deployed and left to work until the need arises for retraining. This is highly impractical for a real time music system based on improvisation since the language of the performance can change quite drastically over the course of a short amount of time. What is needed is a system that can effectively retrain itself with relative speed and generate results that are effective within a more immediate music context using a limited amount of data.

---

[5] Google, Tensorflow Google. "Tensorflow/magenta."

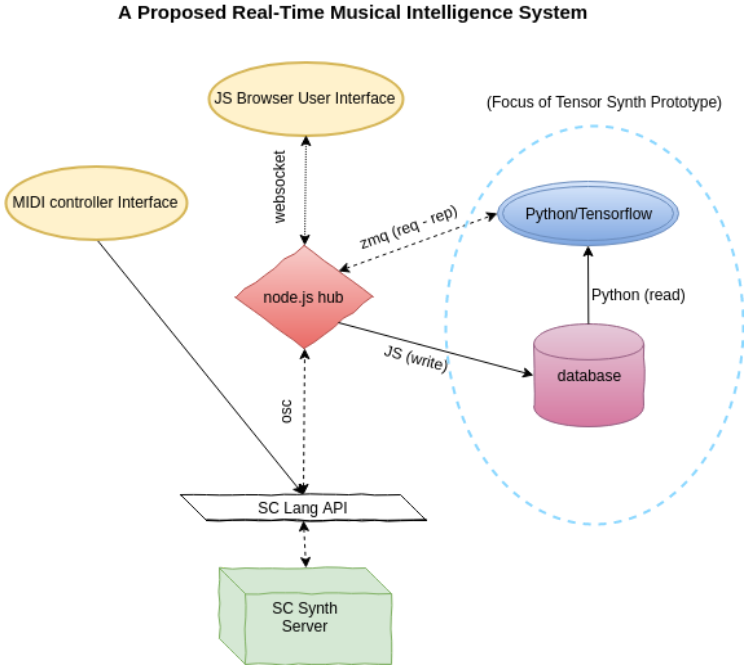
[6] See for example: chap. 5 & 9 of Nierhaus, Gerhard. *Algorithmic Composition: Paradigms of Automated Music Generation*.

Based on the this set of criteria, it was decided that the audio specification for TensorSynth would be implemented as a granular synthesis engine due to the richness and variety of possible sounds that can be produced using a single instrument. Musical parameter event data for such an instrument could be binned and reduced to a 1-dimensional vector of simple strings in order to encode a grammar unique to an individual's performance on the granular synthesis engine.

Once encoded, this sequence of strings can be treated just like words in a sentence which directly opens the musical data up to a variety of pure natural language processing techniques. Since we are interested in sequencing and clustering of similar musical events, it was decided to implement the recently developed Skip-gram/Word2vec algorithm developed by Mikolov (et. al 2013) using the Tensorflow API.[7] The Word2vec family of algorithms embed words in a vector space, pass these through a hidden layer, and compute a softmax regression on the output layer. The probability vectors of these words will begin to cluster after backpropagation. Similar words will drift closer together within this space after many iterations.[8] A new musical sequence can therefore be generated by computing cosine similarity of a target word in the trained vector space and selecting from a set of its nearest neighbors.

3 The Proposed System and Prototype System

The proposed real-time musical intelligence system is outlined below.

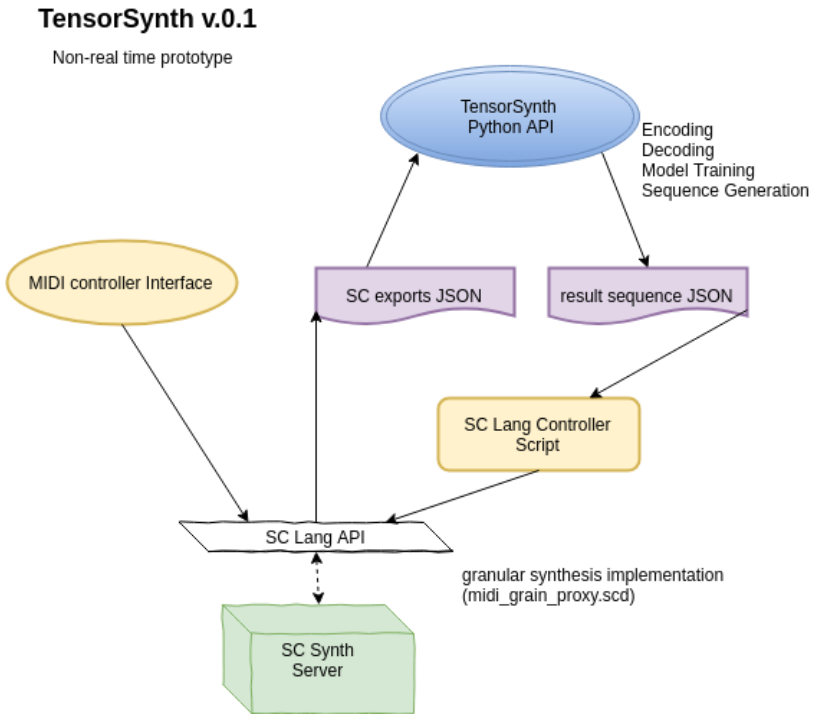


[7] Mikolov, et al. "Distributed representations of words and phrases and their compositionality."

[8] Rong, "word2vec parameter learning explained."

This hypothetical system uses Python for model training, SuperCollider for audio synthesis and a Node.js server to relay the data in real time over a network using ZMQ, OSC and websockets. The user would interface with a MIDI controller device alongside a browser-based GUI to control audio on the synthesis server, model training and sequence generation. As the user performs in SuperCollider using the MIDI controller, the music event parameter data can be sent over UDP via the Open Sound Control (OSC) protocol to a Node server where it is written into a database. At user-defined intervals the event data can be read and processed by the Python TensorSynth API and fed back to the Node server which would emit the data back to the SuperCollider Interface for playback.

The first step in building such an application is the development of the non-networked components. This is the core TensorSynth API involved in data management, model training and sequence generation, as well as the MIDI controller interface and the synthesis engine.



The current non-real time prototype of TensorSynth follows the control flow of the above diagram. The user records a performance with the granular synthesis engine using a simple MIDI interface written in SuperCollider. The SuperCollider API is responsible for storing the musical events in a dictionary-like structure that is then exported into JSON when the performance is complete. The TensorSynth Python API reads in this raw data, encodes, trains a model, generates a sequence and finally decodes the sequence back into JSON. A small SuperCollider client routine reads in this JSON file and plays the generated sequence using the granular synthesis engine.

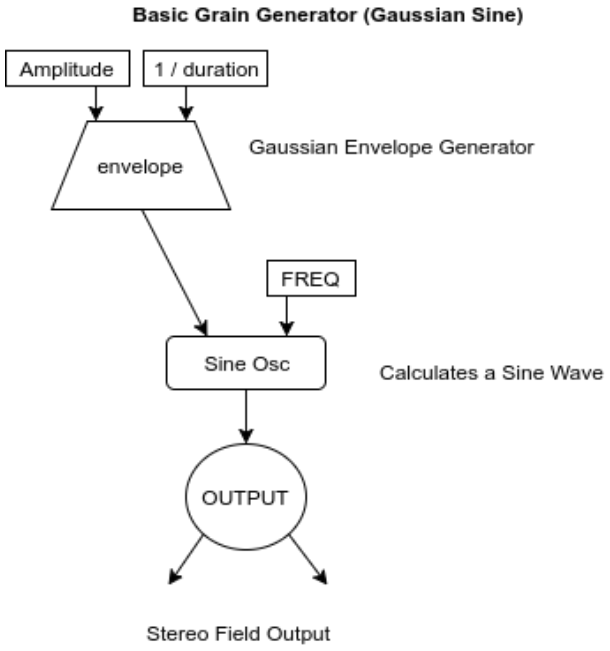
4 Granular Synthesis and Word2vec Overview

Before discussing the development of the prototype application I first want to give an overview of the audio synthesis and natural language processing techniques that form the backbone of the project.

Granular Synthesis

Granular synthesis was first conceptualized by Xenakis for his pieces *Analogique A-B* (1958) using analog generators and tape splicing to create complex and amorphous sounds from extremely small sonic quanta. The first digital realization of granular synthesis was created by Roads in 1974 at MIT and the technique was developed further by Roads, Traux, Vaggione and many others throughout the 1980s and 90s.[9]

The most basic digital implementation consists of a calculated sine wave with a Gaussian amplitude envelope generator. Grain durations are extremely small, typically ranging from 0.001 to 0.1 in length.



The most basic musical event associated with granular synthesis is the *stream*. A large variety of different sounds can be produced by simply coordinating changes between grain pitch, length and emission rate within a single stream. If the rate of grain emission is consistent, a stream is said to be *synchronous*. If the rate varies, the stream is said to be *asynchronous*. Many such individual streams will combine to form a *cloud*. A cloud can consist of many hundreds or thousands of grains over a duration of a single musical event. [10]

TensorSynth's granular synthesis engine is written in the dedicated audio language SuperCollider, which was chosen for its network programming paradigm and proxy based JIT framework which allows finely tuned control over synthesis parameters. The MIDI controller API

---

[9] Roads, Curtis. *The computer music tutorial*. pp. 168 - 170

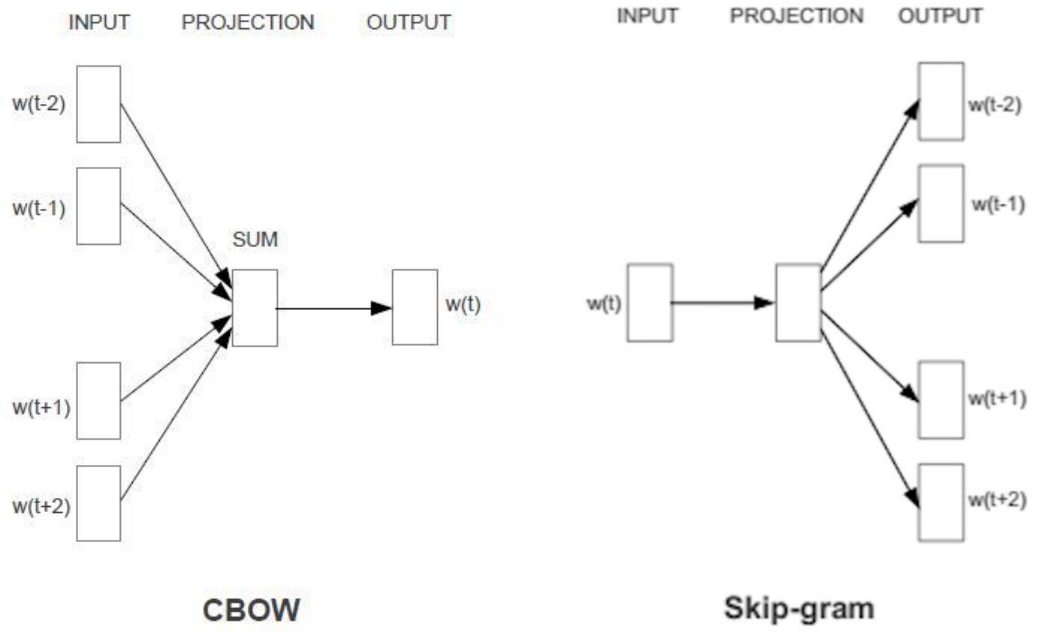
[10] Roads, Curtis. *Microsound.* pp. 86 -97

generates clouds at the push of a key, and besides pitch, gives the user control over release, amplitude, grain rate, number of individual streams, and grain duration. The synth also has deviation settings for pitch, rate and duration which allow one to achieve both synchronous and asynchronous granular synthesis within the same performance.

Word2vec

Word2vec is a set of two natural language processing algorithms developed by Mikolov (et. al) (2013) in order to predict words based on context.[11] The models are trained on simple neural networks using either softmax regression or negative sampling and contain one hidden layer. Words are embedded in a multi-dimensional vector space and through the iterative process of back-propagation, vectors will begin to cluster within the space.

The two flavors of Word2vec are Continuous Bag of Words (CBOW) and Skip-Gram given below. For the purpose of this report I will forego the particulars of the mathematical formulations, since mathematical details have been described in great detail elsewhere.[12] Intuitively, CBOW uses the sum of a set of context vectors to predict a target output, while Skip-gram uses a target word vector in order to predict a set of context word vectors.



[13]

[11] Mikolov, Tomas, et al. "Efficient estimation of word representations in vector space."

[12] For mathematical details about Word2vec and embedded word vectors see both: Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality." and Rong, Xin. "word2vec parameter learning explained."

[13] CBOW/Skip-gram representations used from: Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality."

The methods of TensorSynth's SkipGram class in the train.py module are based heavily on Tensorflow's implementation of the Skip-gram model which was adapted from the Tensorflow API documentation.[14] The Skip-gram implementation was primarily chosen over CBOW based on Mikolov's response to a question in a google groups forum about which model works best for a given situation:

> "If you want to see a list of advantages of each model, then my current experience is:
> Skip-gram: works well with small amount of the training data, represents well even rare words or phrases
> CBOW: several times faster to train than the skip-gram, slightly better accuracy for the frequent words"[15]

The set of raw musical parameter data that TensorSynth encodes will often be quite small, between 75 - 250 words for a performance and small datasets will often be made up of a large number of rare and unique words.

5 Development and Implementation[16]

The non-real time TensorSynth application is composed of series of procedural operations as outlined in the diagram shown below. The application runs through a set of seven distinct tasks in order to create new audio output from a given performance. Each of these tasks is described in detail below.

1. *Record* raw performance data from the SuperCollider granular synthesis engine. Each event dispatches MIDI values (0-127) for pitch, amplitude, release, grain rate, grain duration, frequency deviation, duration deviation and pitch deviation, plus timestamps for note on and note off.
2. *Export* the collection of raw data from the performance in a format suitable to be read in Python. For this we use a JSON export method in a SuperCollider code block.
3. *Encode* the raw data. This process includes calculating event duration and inter-event duration from the timestamps, binning the input data according to a specification and reducing the columns into a 1-dimensional string vector for processing. We also need to save a configuration file with information about the binnings for later decoding.

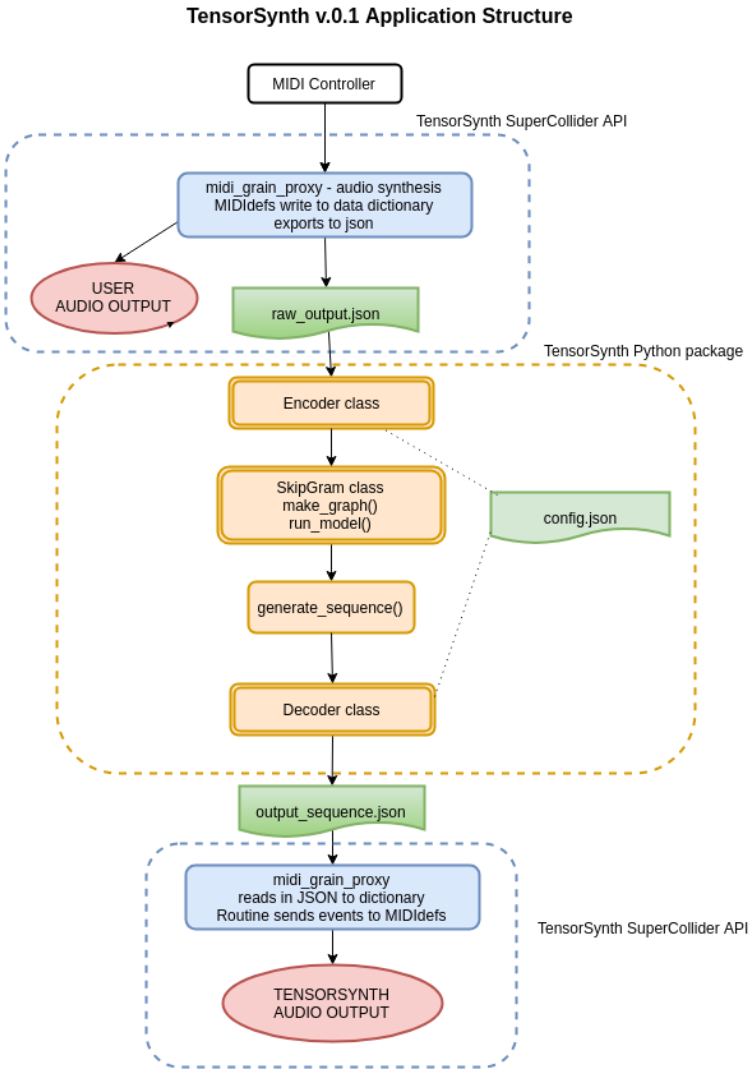
---

[14] Implementation of Skip-gram in the tensor_synth.train.SkipGramTF is based on the example API code in the tensorflow repository:
https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/word2vec/word2vec_basic.py

[15] Taken from a response thread in this discussion:
https://groups.google.com/forum/#!searchin/word2vec-toolkit/c-bow/word2vec-toolkit/NLvYXU99cAM/E5ld8LcDxlAJ

[16] Much of this section follows the development notebook in the project github repo:
https://github.com/bsnacks000/TensorSynth/blob/master/tensor_synth_dev_notebook.ipynb

4. *Train and run* the Skip-gram model. The string vector is pre-processed and run through the Tensorflow Skip-gram implementation in order to embed the strings in a vector space.
5. *Generate* a new sequence using the word embeddings. The current implementation finds k-nearest neighbors of a target word using cosine similarity and selects one at random for the next word.
6. *Decode* the newly generated sequence. This is effectively the encoding process in reverse, using the configuration file from step 3 to unbin the data back to MIDI, expand the dataframe and export back to JSON.
7. *Perform* the new sequence. SuperCollider reads in the JSON file, converts it back to a dictionary structure and plays the events using the same MIDIdef controller Interface that the user plays in step 1.



The first part of the application is run from the *midi_grain_proxy.scd* file. Since SuperCollider is an interpreted language that is designed specifically for real time interaction with a specialized audio server, it is convenient to put all the source code in a single file for the application to be run by the user. Small pieces of functionality are generally evaluated in "blocks". After booting the server and loading the SynthDef that contains the grain generator, several global event dictionaries are created with key value pairs for storing each note/cloud event. The cloud_maker() method is called for each noteOn instance and returns a pattern as a NodeProxy, a special controller class that interfaces with the audio server to generate the grain clouds. A third code block contains the MIDI controller API, that programs the logic for a small Novation Launchkey controller. This includes pitch control using a standard keyboard for pitch and noteOn/Off events as well as 8 control knobs for each of the parameters described above.

Once this set of code blocks has been evaluated, a performance can begin. Each noteOn and noteOff event are evaluated in the MIDIdef controller and stored in the global dictionary *g.output*. Once a performance is done, the user evaluates the File.use() output method on line 184 which stores the JSON into the repository's data directory. For testing purposes I kept my performances around 3-5 minutes which generated output files with between 125 - 350 events.

Now that the performance data is stored, the TensorSynth Python application can be run to train a model and generate output. The API demo can be run from *main.py* in the top level of the application folder. The main application code is provided below and follows the encode, train, generate and decode pattern outlined in the application structure diagram. The inner workings of the process are also provided in detail in the jupyter notebook *tensor_synth_dev_notebook.ipynb*.

```
def main():

    testpath = os.path.abspath(
        os.path.join('.','data','grain_improv2.json'))

    x = EncoderProxySynth(testpath)

    config_path = os.path.abspath(
        os.path.join('.','data','test_config_output.json'))
    x.make_config_json(config_path) # generates a test config file

    y = SkipGramTF(x.output_seq)
    y.make_graph()
    y.run_model(100)

    new_seq = generate_word_sequence(y,25,8)

    json_output_path = os.path.abspath(
```

```
        os.path.join('.','data','api_test_output.json'))
    z = DecoderProxySynth(new_seq, config_path)

    z.export_output_to_json(json_output_path)
```

First an EncoderProxySynth object is instantiated using the raw performance input json file. This is a subclass of the abstract base class Encoder that adheres specifically to midi_grain_synth's parameter specifications. Future releases of TensorSynth will be able to incorporate a wide variety of Encoders and Decoders for different synth grammars by subclassing this Base class and implementing the abstract methods provided.

The EncoderProxySynth first converts the raw json into a dataframe and converts the timestamps to two new columns, duration and inter_event_duration, which correspond to the length of the event and the time interval until the following event. This allows TensorSynth to encode timings of different events without having to rely on a fixed grid. The Encoder object then bins the data using pd.cut with specifications in the *binning_specs.py* module. This bins all synth parameter integer variables into character sets of 4, 8 and 16 using a uniform distribution. Durations are binned in a similar manner but using an exponential distribution in order increase the number of binnings for smaller time intervals. I deemed that discriminating between individual pitches in different registers was an important musical characteristic and therefore decided not to bin the MIDI pitch values for the events.

Once binned, the Encoder performs a dimensionality reduction that produces a 1-dimensional string array that is stored in the instance variable *output_seq*. This output format is also shown in the development notebook under the variable name *word_series*, and consists of a number followed by an underscore and a group of letters. Importantly, the thresholds used to bin each of the variables is stored in a json file which will later be used by the Decoder module to unbin the results of the model's generated sequence into relevant MIDI values.

The SkipGramTF object is next instantiated with the *output_seq* array stored in the Encoder object. This object batches the array and creates two new arrays, training inputs and training labels which will later be fed into the feed_dict of the tensorflow nce_loss function. For Skip-gram each target word is associated with a surrounding window of words as a context. For example given ['a', 'b','c'], if the window equals 2, the resulting arrays for the neural network feed dictionary would be ['b','b'], ['a','c']. After object creation, the main application calls the make_graph() and run_model() methods, passing in the number of iterations for the last one. On such small batches I found that even without using negative sampling the gradient descent optimizer converged very quickly after only about 100 iterations.

In order to generate a new word sequence, we pass the model to a method called generate_word_sequence() passing in the number of words to generate and the number of neighbors to select each word from. This is done in the get_knn_of_target() method by referencing the *final_cosine_similarity* variable stored int he SkipGramTF object, finding the

target word vector, and selecting a neighbor from the pool of k-nearest neighbors. This is the output sequence produced by the embedded vector which now must be decoded back into a format readable by SuperCollider.

A DecoderProxySynthObject is finally instantiated with the new output sequence and a path to the config.json file stored by the Encoder object. The config file essentially contains the instructions for how to unbin the encoded characters which are unique to that particular initial performance that the model was trained on. The unbinning procedure selects a random integer or continuous value within each threshold for the synth parameters. Even though this introduces some randomness into the system, I felt that minute differences in MIDI values for many of the parameters would not change the relationships of the result set too drastically. It is arguably a more natural way to model improvisation since it alleviates outright imitation of specific patterns.

Finally the decoded dataframe is exported back into JSON format in the top level data directory. The user can load this file back into SuperCollider and execute the final code block, a Routine which plays the generated sequence back through the same MIDIdef API that was used in the initial performance.

6 Future Research and Next Steps

While the current implementation of TensorSynth works as a proof of concept and is capable of producing interesting musical results, there are some areas in the current system that need to be improved upon before development of a real-time network driven application can be undertaken.

The main difficulty currently facing development of TensorSynth is being able to correctly assess the accuracy of the vector embeddings in an objective manner. For clustering based algorithms such as Word2vec, dimension reduction techniques such as principal component analysis or t-SNE are often employed to gain insight into how well embeddings cluster based on visual inspection.[17] For a natural language this is easy to assess, since we can intuitively see similarity between different words and whether or not they are clustered. The TensorSynth language vectors exist at a much higher, symbolic level of abstraction and are harder to assess in a straightforward manner. Certain word combinations might be unique to a particular performance, and relationships amongst these words can also change depending on the style of the performer. Looking at a dimensionality reduction of a single performance would not be meaningful since the relationships are constantly changing.

Mikolov developed sets of syntatic and semantic questions that could be answered symbolically by adding and subtracting certain vectors from the vector space and evaluating the result in terms of similarity and differences. Perhaps a similar approach can be taken with TensorSynth by inputting control sets into the system that forcibly accentuate certain relationships between parameters. For example, a purposeful sequence of high frequency, low grain rate, high grain duration events followed by a series that inverts each of those parameters.

[17] Tensorflow API documentation and Word2vec tutorial: https://www.tensorflow.org/tutorials/word2vec

A set of symbolic tests might be written to see if by adding and subtracting from these vectors, one could use cosine distance to find its correct inversion in a similar fashion to Mikolov's test.[18]

Another issue that I faced during development was the lack of transparency in the Tensorflow framework. Tensorflow is a powerful API and is designed to fit the demands of enterprise level analytics software, offering a huge catalog of neural networks and optimizers out of the box that can work with huge datasets without much code. This was appealing to me at first since the TensorSynth application had many moving components I needed to work out and I wanted a machine learning framework with pre-built methods for training neural networks. However,its complexity and the black box nature of much of its codebase made it difficult to debug and did not offer much flexibility in how to develop the model training code.

Because of the difficulty with Tensorflow, I have begun to explore the possibility of building my own custom Word2vec implementation based on the WEVI web application published by Xin Rong.[19] The author offers open source versions of both CBOW and Skip-gram models in javascript that are designed to work on small datasets using a very simple interface. My plan for the next version of TensorSynth is to port as much of Rong's javascipt code as possible into Python and use it as a drop in replacement for the SkipGramTF class in the train module. Rong's application also implements a real time PCA graph that could potentially be used in the browser GUI for the real-time application.

On the SuperCollider side of the application, future releases will contain more server-centric implementations of granular synthesis. TensorSynth currently generates its patterns using a language proxy and contains minimal amount of server-side code. Though such synths will be less expressive, it would greatly simplify much of the MIDIDef interface code and memory resources needed to run and record event data. Less expressive instruments, would mean less parameters and less complex sounds, which could make developing a rigorous model testing framework easier for future releases.

Resources

TensorSynth Code Repository link:
https://github.com/bsnacks000/TensorSynth

The Python package is located in the tensor_synth directory. The SuperCollider script *midi_grain_proxy.scd* contains the granular synthesis engine, MIDI interface and Output Routine are in the sc folder. All json result files generated during development are in the data folder. A development notebook at the root directory contains a step by step implementation of encoding, training and decoding.

---

[18] see specifically section 4 of Mikolov, Tomas, et al. "Efficient estimation of word representations in vector space".

[19] Wevi javascript application: https://ronxin.github.io/wevi/

A main.py file in the top level will process and generate a new sequence based on *tensor_synth_midi_ouput.json* and write to the file *api_test_output.json*

Papers and other Citations:

Cope, David. "Experiments in Music Intelligence." *Proceedings of the International Computer Music Conference*.

Eck, Douglas, and Juergen Schmidhuber. "A first look at music composition using lstm recurrent neural networks." *Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale* 103 (2002).

Google Brain Team (Google). "Tensorflow/magenta." GitHub. Google Brain Team, 16 May 2017. Web. 17 May 2017.

Hild, Hermann, Johannes Feulner, and Wolfram Menzel. "HARMONET: A neural net for harmonizing chorales in the style of JS Bach." *NIPS*. 1991.

Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality." *Advances in neural information processing systems*. 2013.

Mikolov, Tomas, et al. "Efficient estimation of word representations in vector space." *arXiv preprint arXiv:1301.3781* (2013).

Nierhaus, Gerhard. *Algorithmic Composition: Paradigms of Automated Music Generation.* Wien: Springer, 2010. N. pag. Print.

Papadopoulos, George, and Geraint Wiggins. "AI methods for algorithmic composition: A survey, a critical view and future prospects." *AISB Symposium on Musical Creativity*. Edinburgh, UK, 1999.

Roads, Curtis. *The computer music tutorial*. MIT press, 1996.

Roads, Curtis. *Microsound.* MIT Press. MIT, 19 Aug. 2004. Web. 17 May 2017.

Rong, Xin. "word2vec parameter learning explained." *arXiv preprint arXiv:1411.2738* (2014).

Rong, Xin. "Ronxin/wevi." GitHub. Wevi, 12 Feb. 2016. Web. 17 May 2017.

Tensorflow authors(Google). "Vector Representations of Words | TensorFlow." TensorFlow. Google, 2016. Web. 17 May 2017. <https://www.tensorflow.org/tutorials/word2vec>.

Tensorflow authors(Google). "Tensorflow/tensorflow." GitHub. Google, n.d. Web. 17 May 2017.

<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/word2vec/word2vec_basic.py>