Cibo: An Autonomous TidalCyles Performer

Jeremy Stewart
Rensselaer Polytechnic Institute
stewaj5@rpi.edu

Shawn Lawson
Rensselaer Polytechnic Institute
lawsos2@rpi.edu

ABSTRACT

The authors created a sequence-to-sequence style neural net algorithm to autonomously perform TidalCycles. This paper outlines how this approach is distinct from other styles of automatic sound and score generators, describes in detail the implementation of the algorithm, asks some philosophical questions, and speculates on related future research.

1 Introduction

Forms of machine learning and music have existed for more than thirty years (Dannenberg 1985). In lieu of tracing this lengthy and varied lineage, we will quickly demonstrate where this paper's research is positioned.

The first separation from a large body of work on machine learning and music is that our method does not ingest as input nor generate as output any sonic data: audio signal, FFT, audio features, etc. Second, our algorithm is working in the context of live coding. Third, it has be argued that live coding is a score (Magnusson 2014); therefor our method does not ingest as input nor generate as output any non-live-coded score: midi data, canonical western scoring, animated notation, tablature, and so on.

Within this remaining space of machine learning, music, and live coding, Navarro and Ogborn (2017) demonstrated a novel use of a neural net, *Cacharpo*, that "hears" a live coder and assists them by producing SuperCollider code. Our algorithm, Cibo, is also intended to be a co-performer, similar to *Carcharpo*; although, in Cibo's current state of development, Cibo performs solo. In addition, to quickly note, Navarro and Ogborn's, *Cacharpo*, ingests a processed audio signal, whereas this was a previously mentioned area of machine learning that Cibo was not a part of.

Cibo only ingests TidalCyles (Alex McLean, n.d.a) code as input, and then generates as output, TidalCyles code. Again to reiterate, Cibo does not operate on sonic data. Cibo's underlying layers of neural networks were trained on sequential blocks of executed TidalCycles code that were captured during a human's performance. In this way, Cibo has "knowledge" of how a live coder performs and the procedural edits made to code based on a human's aesthetic decisions.

Through this paper we will describe the overall training and performance toolchain of Cibo, dig into the details of Cibo's neural net architecture, provide some Cibo performance examples, briefly consider the aethetic implications of nueral net training and its results, and finally propose areas of future research for this work.

2 Technical

2.1 Overview

The goal of this project is to build a software architecture capable of live-code-performing TidalCycles. The system receives code as input — from a custom text editor — and produces new TidalCycles code as output, which is returned to the text editor and executed/submitted to the Haskell interactive interpreter, similar to how a human performer would live-code. As will be discussed in [Future Work], one of the hopes of this project is to create a collaborative Artificially Intelligent (AI) performer. With this in mind, creating an AI agent that can interact with a text editor together with a human performer was an early requirement.

Figure 1 outlines the high-level architecture of this project. Taken as a complete set, the Jensaarai editor (Lawson 2018a), Tidal Runner (Lawson 2018b), and Super Collider ("SuperCollider," n.d.), running SuperDirt (n.d.a), exist as a complete Tidal performance pipeline for a human performer. The addition of our Cibo Agent runs in a loop, adjacent to that straightforward pathway. Jensaarai is modified to target both the Tidal Runner and Cibo Agent applications when sending code for execution. The Cibo Agent produces new code blocks of Tidal code and returns that code to the Jensaarai editor.

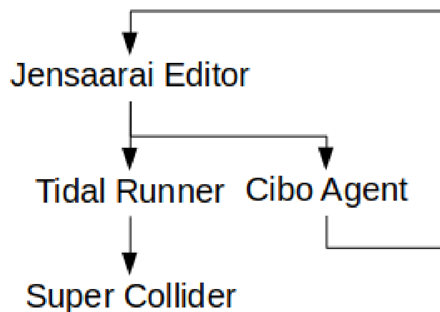

Figure 1: **High level architecture of coding flow.**

Currently, the Cibo project is focused on developing the AI software, however this pipeline has allowed us to continue to consider real-time collaborative performance between human and non-human performers.

Tidal Runner is a Node.js (n.d.b) command line application that wraps the TidalCycles, Haskell interpretor with an Open Sound Control (OSC) (n.d.c) listener. Tidal Runner receives messages via OSC containing Tidal code from Jensaarai, from there, functionality continues the same as any other Tidal performance pipeline: forwarding formatted OSC messages to the SuperDirt sampler running in SuperCollider. Much the same way one would use the Atom editor or Emacs. The interpretor within Tidal Runner may also reply with a return message or error to the sender, in our case, Jensaarai.

The Cibo Agent receives as input the same OSC message that Tidal Runner receives from the Jensaarai editor. Upon receiving a block of code, the Cibo Agent must tokenize the code using the Ply Lexer(n.d.d). The tokenized code is then passed into the first of a number of neural network modules which constitute the heart of Cibo's AI. The output from the Cibo Agent is a block of Tidal code, prepared as a string datatype, and sent back to Jensaarai via OSC. In Jesaarai, the received Tidal code replaces the previously existing, executed code and then executes the new code, beginning the process again.

2.2 PLY Lexer

The first step of input into the Cibo Agent is to produce a usable, discrete data set made up of integer representations of tokens (keywords, functions, symbols) along with any accompanying, non-tokenized values (ints, floats, strings). The PLY library (n.d.d) for Python offers a toolset for creating a lexer and abstract syntax tree parser. For the sake of this project, we are only using the lexer portion of the PLY Library. The Cibo Agent does not require that code be parsed into an abstract syntax tree representation prior to ingestion; instead, the tokenized input constitutes a viable representation from which our agent can learn to discern semantic structure, as will be discussed in the following sections.

Figure 2 shows the operation of the lexer in the Cibo Agent. At the top of the figure is a hypothetical Tidal code block received from the Jensaarai editor as a string. This input is valid within the Tidal performance environment. The lexer takes this string as input and creates two arrays, a *tokens* array and a *values* array.

Upon receiving the Tidal code input, the lexer separates all contained symbols and alphanumeric characters, with special case rules for whitespace characters, matching them against a number of reserved tokens and symbols. For our application, we created a table of reserved keywords which contains all keywords found in the TidalCycles documentation(Alex McLean, n.d.b). This includes all track designations, "d1. . . d8" and "t1. . . t8", all pattern and sample transformer keywords: "fast", "shuffle", "degrade", "gap", "chop", etc., all conditional transformers: "every", "whenmod", etc., compositional keywords: "cat", "stack", "seqP", etc., transitional keywords: "jump", "anticipate", etc., and all synth parameter keywords: "speed", "cut", "begin", "end", etc. In addition, token reservations were made for all valid symbols in TidalCycles: brackets, parens, pipes, quotes, and mathematical symbols. Finally, there exists in TidalCycles several types which cannot be completely tokenized, but which must be tokenized into their type while retaining their contained value. These include the INT, FLOAT, and STRING tokens. INT and FLOAT types are used along with many of the transformer functions in TidalCycles, for example, "every 3" denotes that on every third cycle some transformation should be applied. INT and FLOAT types are discerned by the presence of a decimal point. The STRING token type is used to represent any collection of alphanumeric characters which do not exist in the token reservation list. This includes all of the available sample names which are loaded into the SuperDirt sampler in SuperCollider. This list of available samples is potentially unique for every user and every machine. The lexer of the Cibo Agent contains a dictionary which holds all of the available sample names, allowing us to reference these names using an integer enumeration. The STRING type notifies our software to reference this dictionary when decoding the value into a human-readable output.

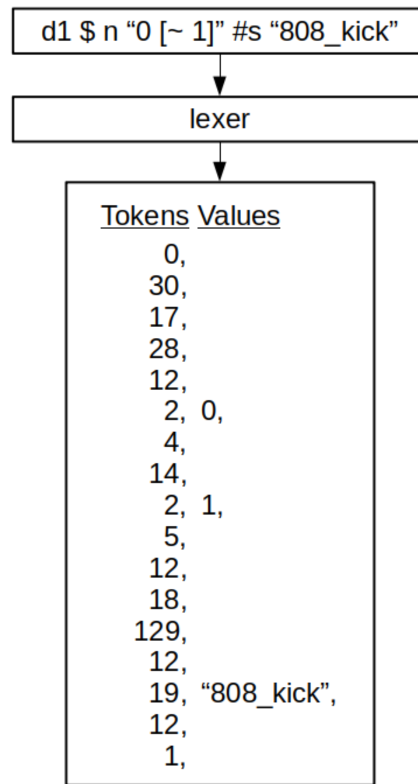

Figure 2: **Conversion from TidalCycles to tokens.**

In addition to producing a *tokens* array, seen at bottom-left of Figure 2, we also create a *values* array, which, in the case of INT, FLOAT, and STRING tokens, holds the associated values, seen at bottom-right of Figure 2. Values associated with STRING tokens, i.e., sample names, are stored in a local dictionary, each with an associated index. The values associated with all STRING tokens are replaced with this dictionary index value. Then, all values in the *values* array are normalized before being used, based on the possible ranges of values available for each token type. In other words, there are separate scaling functions for each INT, FLOAT, and STRING token values based on the available values in the training data. Normalization of data in this variable-specific manner functions to reduce outliers and create more evenly distributed data, which in turn aids in the training process of our neural network system.

In the instance that a given token does not require an associated value (tokens that are not of type INT, FLOAT, or STRING), the *values* array is filled with a NULL value. Given the current architecture, the values array produced by the Cibo agent will be of the same length as the *tokens* array. In any instance where a given token does not require an associate value, the entry in the *values* array at the same index is disregarded. This allows the model to continue to take into account the full context of a given TidalCycles sequence.

2.3 Cibo Agent

The Cibo Agent consists of five interconnected neural network modules which allow for the generation of TidalCycles code through the creation of new token and value sets based on performance context and preceding code blocks. In order to achieve this goal, we have looked to the fields of natural language processing and machine learning. The Cibo Agent is a novel implementation of an encoder-decoder sequence-to-sequence architecture, first described by Sutskever, Vinyals, and Le (2014) and further elaborated upon by Bahdanau, Cho, and Bengio (2014). This sequence-to-sequence architecture is an elegant solution to problems consisting of input and output data that are of variable lengths. This is accomplished by creating two connected recurrent neural networks. The first of these, the **Encoder**, takes a variable length input and produces a fixed length abstract representation. The second recurrent neural network, called the **Decoder**, reads this fixed length representation, producing a new collection of output data. This architecture is frequently used for language translation. Our implementation as described in this paper was carried out using the PyTorch library (n.d.e; Paszke et al. 2017), running on a machine with a CUDA and cuDNN enabled Nvidia GPU to increase training speed (Nickolls et al. 2008; Chetlur et al. 2014).

Each of the five modules will be discussed briefly, following the path of an input data set (the tokenized code produced by the lexer as described above) through the creation of new code.

2.3.1 Encoder RNN

The encoder recurrent neural network (RNN), seen at the top left of Figure 3, accepts as input the tokenized TidalCycles code produced by the lexer, as described above. The encoder consists of two main layers. First, an embedding layer accepts as input a tensor containing the tokenized input as enumerated integers. The embedding layer converts each of these integer values into a floating-point vector of a set dimensionality, thus representing an atomic unit, the token, as a point in a continuous, high-dimensional space. (Mikolov et al. 2013) The usage of such vector representations, through the use of an embedding layer, has been shown to aid in learning relationships between words or tokens with similar semantic meaning or syntactic structure. (Mikolov, Yih, and Zweig 2013). Tokens with similar syntactic function or semantic meaning, as learned through a given training corpus, will appear more closely related in this higher-dimensional vector space.

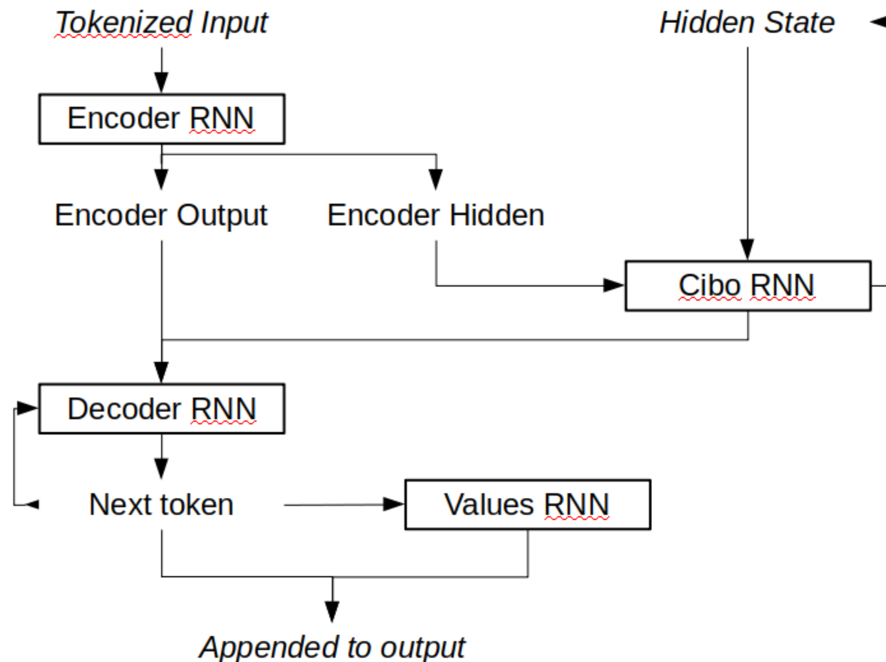

Figure 3: **The Neural Net architecture**

The results of the embedding layer are then input into a multi-layer gated recurrent unit (GRU). The gated recurrent unit was first proposed in 2014 (Cho et al. 2014) and further evaluated by Chung et al (2014). The gated recurrent unit offers several advantages over a basic recurrent neural network architecture through the use of several gating mechanisms which control the passage of data. These gating mechanisms aid in the learning of temporal relations at multiple scales, while lessening the problem of vanishing gradients associated with some recurrent architectures. Additionally, the gated recurrent unit has fewer components than the also widespread long-short term memory (LSTM) unit, which translates in our project to easier and faster training of the system as a whole.

The results from the encoder RNN consists of two components: a variable-length output with shape equal to the input length times the number of features in the hidden state of the encoder, and a fixed-length output with shape equal to the number of recurrent layers times the number of features in the hidden state of the encoder. The latter can be thought of as a fixed-size abstract “meaning representation” of the input sequence which will be decoded in the Decoder RNN while the former will be used in conjunction with an attention mechanism (discussed further, below) to aid in the decoding process.

2.3.2 Cibo RNN

In a sequence-to-sequence architecture as described by Sutskever, Vinyals, and Le (2014), this fixed-length output is decoded by the decoder module to produce the resulting output sequence. Because the encoder is reinitialized every time a new input is received, the encoder-decoder pair can produce a reliable translation from one sequence to another. For the project at hand, we require the ability to input one sequence of variable length, producing a subsequent sequence

of independent length, while also maintaining sensitivity to a broader, temporally unfolding, performance context. In order to accomplish this, we have including the Cibo RNN module, seen in Figure 3 at center-right. This module takes as input the fixed-length output from the Encoder RNN as well as a hidden state which persists and changes over the course of the entire performance (the performance context). The Cibo RNN functions to modify this fixed-length meaning representation, before it is decoded into an output sequence, based on the context of all events occurring since the beginning of the performance.

The Cibo RNN consists of an multi-layer LSTM (long-short term memory) unit followed by the rectified linear unit (ReLU), nonlinear activation function, and outputting through a fully connected, linear layer. The number of outputs from the Cibo RNN is equal to the number of inputs, effectively allowing it to modify the fixed-length output from the encoder module before forwarding it to the decoder, otherwise preserving the sequence-to-sequence architecture.

The usage of the multi-layer LSTM unit over the simpler GRU unit was based on several training sessions carried out with each architecture. The resulting performance seemed to indicate a greater sensitivity to the live-coding performance unfolding when using the LSTM unit in this Cibo RNN module, although additional testing is needed to confirm.

The hidden state of the Cibo RNN, indicated in Figure 3 at top-right, is initialized at the start of the live-coding performance. This hidden state is used and updated by the Cibo RNN with every new sequence input, and is maintained between sequences. The Cibo Agent is programmed to only reinitialize this hidden state in instances where the agent finds itself locked into nonproductive loops, for example: producing the same sequence over and over.

2.3.3 Decoder RNN

The decoder module within the Cibo agent, seen in Figure 3 at center-left, is responsible for producing new output tokens which will be joined together into the final output sequence before being converted into a string for sending back to the Jensaarai editor for execution. The decoder does this by taking input from the fixed-length representation that was output from the encoder RNN, then modified by the Cibo RNN, marked as "Encoder Hidden" in Figure 3, and the variable-length output from the encoder. The decoder is then sent a Start of Sequence token (SOS), causing it to output one token. This token output from the decoder is then input back into the decoder, and this process is repeated until the decoder outputs an End of Sequence token (EOS) or the maximum sequence length is reached. This maximum sequence length is meant as a safeguard against unreasonably long outputs. When these extremely long outputs are encountered, there tends to be a higher frequency of syntactically invalid Tidal code, and so the system state may need to be reinitialized.

The decoder RNN contains an attention module (Luong, Pham, and Manning 2015) which, upon each step of the output sequence generation, produces a set of weights indicating the relative importance of each component of the input sequence. As per discussions found in "Effective Approaches to Attention-based Neural Machine Translation" (Luong, Pham, and Manning 2015), the Cibo agent includes a global attention module, which permits the decoder to consider the entirety of the input sequence at each step of output generation.

Additionally, Luong, Pham, and Manning (2015) explore several different implementations of a global attention module: "general", "concatenative", and "dot." Discussion of each variation and their relative strengths and weaknesses is beyond the scope of this paper. For our implementation of an attention module as part of the Cibo agent, we have used what Luong, Pham, and Manning (2015) refer to as a "general" global attention module, which essentially consists of a fully connected neural network which produces a set of weights, as described above, for each step of the output generation process.

2.3.4 Values RNN

As part of the output sequence generation, each token generated by the decoder RNN is also sent as input into the Values RNN, seen in Fig. 3, bottom center-right. This recurrent neural network module is responsible for generating new values that are associated with each token. If the token generated by the decoder is an INT, FLOAT, or STRING token, then the value produced by this Values RNN is used in the final sequence output. Otherwise, the output of this neural network module is not used directly, but the context of the entire generated output sequence affects the values produced.

This module consists of an embedding layer, followed by a multi-layer gated recurrent unit. The output of the GRU layer is passed through the ReLU activation function before going through a fully connected layer and to the output.

Depending on the token being referenced, the output from this Values RNN will be scaled appropriately, according to the inverse normalization function that is implemented as part of the Cibo lexer noted above. If the associated token is STRING, then this produced value is then converted into a String based on a stored dictionary of available sample names.

2.3.5 Setting Hyperparameters and Training

Training of the Cibo agent was carried out as a two step process, with training of the encoder, Cibo, and decoder RNNs occurring simultaneously, followed by the training of the Values RNN occurring at a later point. All training was carried out using a set of recorded TidalCycles performances created by the authors. The current training set is small, consisting of several performances totaling approximately 1000 discrete events once thinned for redundant contiguous executions and syntax mistakes. The number of unique tokens used for training Cibo is 142, which includes all symbols and keywords found in the TidalCycles documentation, as well as tokens for INT, FLOAT, and STRING.

The encoder, Cibo, and decoder RNNs were trained using the Adam optimization algorithm (Kingma and Ba 2014) on a single NVidia Titan XP GPU for a period of approximately 36 hours. Training was carried out by taking each of the TidalCycles performance recordings and using contiguous entries as input and target sequence.

Many different hyperparameter configurations were tested, although, because of the complexity of the model and the time required to train it, we have at present only tested a small percentage of the hyperparameter space. The demonstrated performance of the Cibo Agent, viewable here: https://vimeo.com/287336427/c9bb5e536e and https://vimeo.com/288889990/6d92fd1a55, was built using the following hyperparameters: The encoder hidden state size is 750 with 5 layers in the multi-layer gated recurrent unit (GRU).

The Cibo RNN input is equal to the encoder hidden state size time the number of GRU layer, producing an input size of 3750. The hidden state size of the Cibo RNN was set to 1750, with 5 layers in the multi-layer LSTM.

The decoder, similar to the encoder, was built with a hidden state size of 750, along with 5 layers in the multi-layer GRU.

The Values RNN was trained separately from the rest of the system using the same set of TidalCycles recordings, although with each executed code block parsed and separated into tokens and values. The tokens array functions as the input to the Values RNN while the values array functions as its target output.The embedding layer of the Vales RNN has a dimensionality of 500, and the recurrent layer, consisting of a multi-layer gated recurrent unit, has a hidden state of 750 features and two layers. Finally, there is a fully connected, linear layer before the output which has an input size of 750 and output size of 1.

3 Aesthetics of Machine Learning

With the sequence-to-sequence model, we've trained the algorithm to recognize how a translation from one sequence of tidal code should/could be translated into a subsequent sequence of Tidal code. What is inherit to the training data is the human aspect. Each sequential tidal code block of training has been informed by a human's aesthetic decision: samples, filters, timing, and so on. We think that the algorithm, with enough training data, may be able to exhibit a human performer's aesthetic choices based on the Tidal code and code execution timings. Does this imply that some lines of Tidal code are more sonically-aesthetically pleasing than others? Maybe. Also, as is evident with human performers, could Cibo be trained with singular or combinations of data sets to create different styles of tidal code performance? Maybe.

We also postulate that the Cibo Agent fulfills Tanimoto's level 5, Tactically predicative, of his liveness hierarchy. The Cibo Agent runs, responds, and predicts a learnt response of what the human performer may have made a change to. "(One may argue that here, liveness has spread from the coding process to the tool itself.)" (Tanimoto 2013). We can infer that liveness is now the tool, Cibo, and it's processes. Although, we believe that the Cibo Agent is not yet at Tanimoto's level 6, Strategically predictive, as this presumes a larger scope of data training than what we have accomplished.

In the context of performance, if the Cibo Agent can respond (predict) a following block of tidal code to it's own previous block of tidal code, then where is the performed action? If one were to 'show our (Cibo's) screens,' then blocks of tidal code in a text buffer would instantly change. Do we know that Cibo is thinking, processing, or computing? Because our current implementation doesn't utilize the cursor as Cocker suggests that the cursor is the site of action, "performed thinking" (2016, 106), we follow the live-coding intent of 'showing your screen' to 'show your work.' Cibo demonstrates its work with output from neural net ingestion and output data, on screen, adjacent to the Tidal code editor. This is the thinking-in-action that occurs within the Cibo Agent. In the "active disclosure" of Cibo we're revealing the trials, errors, and perceived unknowing of how the Cibo performs (Cocker 2016, 109).

> Moreover, it [live coding] is a practice that requires some prior knowledge of a process (what can be predicted or anticipated *before* or in advance); embodied 'know-how' activated *during, in* and *through* its performance, and a kairotic knowledge (a 'know when' yet arguably known-not knowledge that emerges simultaneous to – unique and in complete fidelity to – the emergent situation, through a process of leaning into the 'to-come'). (Cocker 2016, 112)

Is it possible to state that the Cibo Agent has a prior knowledge? The neural net is trained from live human performed data. The intent is for the Cibo Agent to behave as or emulate the behavior of the original performer. Is that human behavior and creativity captured in the learning? The proverbial 'ghost in the machine'? Additionally, live-coding performances are typically performed by a human, which is in a language and format that other humans can read and follow. Neural net architectures are frequently not entirely comprehend-able by humans. When live-coding's intent is to show the work, what does it mean if Cibo shows its work and yet no-one, not even the creators, understand entirely how its working?

Putting all the aforementioned issues aside, ultimately, if everything that is known is not-known, meaning that if presented with only the resulting sound from the Cibo algorithm, it is the received perception that matters as Andrew Brown states when considering this in his own work when he quotes from Auslander.

> This 'show us your screens' feature of live coding practice assists to reveal the process and virtuosity of the performer, underscoring Auslander's point about performance with technology that 'what counts, ultimately, is audience perception, not actual degree of difficulty'. (Brown 2016, 182)

4 Conclusion

The Cibo Agent as discussed in this paper provides a good point from which to consider our future research, as well as potential future research in the area of live-coding machine learning agents. First, given the architecture as it is discussed above, a more thorough search of the hyperparameter space is called for to ensure that the architecture as it exists is fully explored and considered for efficacy. One of the barriers to this process remains the computational requirements for training such a complex, neural network-driven system through back-propagation, because a training session can easily take days to complete. Increasing the size of the neural network models not only requires additional computations to be carried out for every step of the training process — thus requiring longer to train — but forming and programming a training regimen capable of ensuring adequate outcomes becomes a more complex process as well. Pre-training of some modules or training with graduated learning rates based on progress or stagnation are two potential avenues to be explored.

Combining several of the neural network modules has potential to better capture variability as it exists in the source performances by allowing a closer relationality between current discrete variables and variable types, for example: tokens versus associated values. This is a clear and obvious step forward, but, as mentioned above, will likely require additional development of training regimen.

Currently, the Cibo Agent may only employ samples which were present in the training corpus. This has the obvious drawback of requiring a fixed sample set for training and execution – a new user will need to acquire the same sample set or retrain the entire system. Ideally, the Cibo Agent would be able to freely perform using available samples, whether or not they were present in the training corpus. A simple implementation of this would be to load the list of available samples into a dictionary which would then be queried based on the existing mechanisms. However, it is the authors' beliefs that this would be an inadequate approach, given the sonic sensitivity and awareness of human Live Coding performers. Instead, developing an approach that databases the sonic characteristics of available samples and permits rapid querying (Schwarz 2004) may lead to a more aesthetically cohesive outcome. In such an approach, the Cibo Agent might call for samples based on characteristics rather than index associated with name.

The development of the Cibo Agent for real-time audio ingestion is an obvious goal. With a wide variety of audio analysis tools readily available — through software such as SuperCollider and Max/MSP — we hope to further develop the agent to take real-time, high-level audio descriptor information as additional inputs which will direct the course of a given performance. Such analyses include frequency-related methods such as FFT, CQT, or chromagram as rhythmic onset and beat detection. Developing the Cibo Agent to allow for this kind of audio data as input will bring the agent nearer to the performance scenario experienced by the human performer live-coding in TidalCycles.

Finally, an machine learning agent that is capable of live-coding an audio performance through TidalCycles is implying a partner agent capable of live-coding visuals. We are considering the feasibility of such an agent and the unique problems posed by developing a visually-aware agent (perhaps through the use of convolutional recurrent neural network models (Liang and Hu 2015; Pinheiro and Collobert 2014). Development of such an agent would also present the opportunity for multiple machine learning agents to collaboratively co-perform, thus leading to a number of new, conceptual and philosophical questions.

In this paper we endeavored to demonstrate how Cibo is positioned in relation to other research in the field of machine learning, sound, and live-coding. We discussed the training and performance process of Cibo and detailed the neural net architecture. We speculated briefly on how this human captured training data is translated into or becomes emergent within the behavior of Cibo and opened some questions about how to come to terms philosophically with machine learning performances and their aesthetic qualities. Lastly, we speculated about directions for where we plan to expand the research.

Through the development of this work, we gained even more respect for the research of other people in the fields of machine learning and live-coding. Certainly our accomplishment here could not have been done without the work of many others before us.

4.1 Acknowledgments

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research.

References

Alex McLean, et. al. n.d.a. "Tidal." https://tidalcycles.org/.

———. n.d.b. "Tidal." https://tidalcycles.org/functions.html.

Angel, Luis Navarro Del, and David Ogborn. 2017. "Co-Performing Cumbia Sonidera with Deep Abstractions." In *In Proceedings of the Third International Conference on Live Coding, Iclc*. Centro Mexicano Para La Música y las Arts Sonoras, Morelia, Mexico.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. 2014. "Neural Machine Translation by Jointly Learning to Align and Translate." *CoRR* abs/1409.0473. http://arxiv.org/abs/1409.0473.

Brown, Andrew R. 2016. "Performing with the Other: The Relationship of Musician and Machine in Live Coding." *International Journal of Performance Arts and Digital Media* 12 (2): 179–86. https://doi.org/10.1080/14794713.2016.1227595.

Chetlur, Sharan, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. "CuDNN: Efficient Primitives for Deep Learning." *CoRR* abs/1410.0759. http://arxiv.org/abs/1410.0759.

Cho, Kyunghyun, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. "Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation." *CoRR* abs/1406.1078. http://arxiv.org/abs/1406.1078.

Chung, Junyoung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. 2014. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling." *CoRR* abs/1412.3555. http://arxiv.org/abs/1412.3555.

Cocker, Emma. 2016. "Performing Thinking in Action: The Meletē of Live Coding." *International Journal of Performance Arts and Digital Media* 12 (2): 102–16. https://doi.org/10.1080/14794713.2016.1227597.

Dannenberg, Roger B. 1985. "An on-Line Algorithm for Real-Time Acoompaniment." In *Proceedings of the 1984 International Computer Music Conference*, 193–98. Computer Music Association.

Kingma, Diederik P., and Jimmy Ba. 2014. "Adam: A Method for Stochastic Optimization." *CoRR* abs/1412.6980. http://arxiv.org/abs/1412.6980.

Lawson, Shawn. 2018a. "Jensaarai." https://github.com/shawnlawson/Jensaarai.

———. 2018b. https://github.com/shawnlawson/tidalrunner.

Liang, Ming, and Xiaolin Hu. 2015. "Recurrent Convolutional Neural Network for Object Recognition." In *Proceedings of the Ieee Conference on Computer Vision and Pattern Recognition*, 3367–75.

Luong, Minh-Thang, Hieu Pham, and Christopher D. Manning. 2015. "Effective Approaches to Attention-Based Neural Machine Translation." *CoRR* abs/1508.04025. http://arxiv.org/abs/1508.04025.

Magnusson, Thor. 2014. "Algorithms as Scores: Coding Live Music." In *NIME'14 Proceedings*. New Interfaces for Musical Expression.

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. "Efficient Estimation of Word Representations in Vector Space." *CoRR* abs/1301.3781. http://arxiv.org/abs/1301.3781.

Mikolov, Tomas, Wen-tau Yih, and Geoffrey Zweig. 2013. "Linguistic Regularities in Continuous Space Word Representations." In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 746–51. Atlanta, Georgia: Association for Computational Linguistics. http://www.aclweb.org/anthology/N13-1090.

Nickolls, John, Ian Buck, Michael Garland, and Kevin Skadron. 2008. "Scalable Parallel Programming with Cuda." *Queue* 6 (2): 40–53. https://doi.org/10.1145/1365490.1365500.

Paszke, Adam, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. "Automatic Differentiation in Pytorch." In *NIPS-W*.

Pinheiro, Pedro HO, and Ronan Collobert. 2014. "Recurrent Convolutional Neural Networks for Scene Labeling." In *31st International Conference on Machine Learning (Icml)*. EPFL-CONF-199822.

Schwarz, Diemo. 2004. "Data-Driven Concatenative Sound Synthesis." PhD thesis, Université Paris 6 – Pierre et Marie Curie.

"SuperCollider." n.d. http://supercollider.github.io.

Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. 2014. "Sequence to Sequence Learning with Neural Networks." In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, 3104–12. NIPS'14. Cambridge, MA, USA: MIT Press. http://dl.acm.org/citation.cfm?id=2969033.2969173.

Tanimoto, Steven L. 2013. "A Perspective on the Evolution of Live Programming." In *Proceedings of the 1st International Workshop on Live Programming*, 31–34. LIVE '13. Piscataway, NJ, USA: IEEE Press. http://dl.acm.org/citation.cfm?id=2662726.2662735.

n.d.a. https://github.com/musikinformatik/SuperDirt.

n.d.b. https://nodejs.org/en/.

n.d.c. http://opensoundcontrol.org.

n.d.d. https://www.dabeaz.com/ply/.

n.d.e. https://pytorch.org/.