Bach Chorale Generation

A Well-Tensored Computer

Dynah Bodvel
University Of Puget Sound
1500 N. Warner
Tacoma, Washington 98406
dhendricks@pugetsound.edu

Trevor Nunn
University Of Puget Sound
1500 N. Warner
Tacoma, Washington 98406
tnunn@pugetsound.edu

David Olson
University Of Puget Sound
1500 N. Warner
Tacoma, Washington 98406
deolson@pugetsound.edu

Andy Van Heuit
University Of Puget Sound
1500 N. Warner
Tacoma, Washington 98406
avanheuit@pugetsound.edu

ABSTRACT

This paper describes the technical schema created for our senior capstone project, in which we built software that allows the computer to create music. This is done in Python, taking advantage of the neural network capabilities provided by the TensorFlow library. Our system uses a recurrent neural network, utilizing the advantages of LSTM nodes, to learn the relationships between notes within classical music. Our system trains on only Bach chorales - this is because chorales are a good learning database since they are relatively straightforward and simplistic. In the end, the computer successfully made music that distantly resembles Bach.

KEYWORDS

Bach, Long Short-Term Memory, Machine Learning, Music, Python, Recurrent Neural Network

1 INTRODUCTION

Johann Sebastian Bach is a name known even to those for whom classical music is nothing more than a lullaby. Bach was in the vanguard of the shift from modal music, a style that predated chordal harmony and was focused more on a single melody (call to mind the chanting monks in Monty Python and the Holy Grail), to tonal music. Tonal music is a system in which the song progresses from one chord to the next based on a set of rules which, when followed, produces music that sounds so natural that the system is ubiquitous to this day. It was Bach who identified the relationships between chords that make the rules work, and nowhere is this insight more apparent than in the collection of songs, specifically chorales, that he wrote for the church.

The idea of generating music using a neural network is not a brand-new one, but our research only yielded a handful of actual implementations, the best-documented of which was generating classical piano music. It was using a set which included, unfortunately, compositions of the early 20th century, a time period during which classical composers were undergoing what can only be called a rebellious phase, and doing the musical equivalent of painting their rooms black and dyeing their hair vibrant colors. The resulting dissonance in the data set makes it difficult to ascertain whether the network had successfully learned to generate classical piano music and tried to incorporate the avant-garde movements in its results, or if its training was only partially successful, resulting in discordant missteps.

Bach chorales are, for a myriad of reasons, the ideal material for a stepping stone in machine learning. Because they abide so cleanly by the harmonic rules Bach set out, deviation is trivially easy to isolate. And given that they only have four voices (meaning that only four notes can be played at one time), they are much easier to work with than piano music, which can have as many as 10. Not only that, but Bach wrote hundreds of them. So, by choosing to train on Bach chorales, we can train on a relatively large collection of songs of standardized style, tonality, and nature. Thus, our results clearly illustrate how well our network has learned from the training set.

2 BACKGROUND

Music as an art form has been with us as a species for longer than cultural memory serves, but computers and related technology are a well-recorded, not as artistic concept from recent history. Melding such technology with art is an undertaking that seems to captivate us. From the music box to grand structures such as the Wintergatan marble machine[14], to even sequencers, technology has helped us capture a sound and manage to make it even more beautiful than it once was. The pattern so far has been that we are the composers and simply utilize the power of the computer. But what of computers and their mental power that may rival ours? Can a computer surpass not just our computational prowess, but also our artistic side?

There have been a plethora of projects revolving around computing creativity. The range of ability and discussions of the "creativity" itself are just as vast. From creating images similar to famous painters to composing like the finest musician, we've tried the gamut. But our project is not so much a question of if computers can be creative as much as it is a question on if they can emulate what we have done. Much of the creativity found in computing relies on neural networks, which may prove effective in finding the patterns that occur in creation[6]. These networks are an attempt to recreate the activities of the human brain as we know them; that is, our pattern recognition and ability to learn through reinforcement[9]. These networks are layers of nodes strung together like neurons, passing info through one another. They possess activation functions, which evaluate weights passed into the nodes[9]. These functions evaluate and pass on a judgment to the next nodes. Eventually, once this info has been put through this lengthy process, it spits out a choice: a number between 1 and 0. This binary choice is weighted as a ‘"do or do not" evaluation depending on its proximity to 1 or 0. With this deceptively simple (alternatively deceptively confusing) framework, it is impressive what can be done.

Google has created two well-known projects, Deep Dream and Magenta [5]. These computing powers run images and sounds through neural networks to find that "Spark"[6] of creativity within the patterns it picks up. With Deep Dream, more of the creativity that is done turns into seeing new pictures within the whole. The goal for Magenta, however, is to more accurately emulate classical music creation. Magenta uses Google's proprietary neural network engine, TensorFlow, in an attempt to see if the network can produce novel creations that perhaps can also tell a story just as Bach once did[5].

Daniel Johnson has done a similar music project, where he has taken the idea to a further degree to see if patterns are more apparent given more input in terms of patterns in time and patterns in surrounding notes. His network zeros in on finding time and note patterns and matching patterns between them[8]. What Johnson hoped to accomplish was a network that could pick up on time signature and chord creation[7]. His project found success in this parallel processing of notes and time[8].

Our project adds support to Johnson's work. While his implementation finds notes and patterns with relative efficacy, our machine more accurately imitates its teacher. We decided to implement Johnson's effective parallelism and tweak it to produce something more coherently tied to Bach's Chorale styles.

Our final choice was to use Python and TensorFlow to implement this neural network. There was a brief period at the beginning of the project where we attempted to use Java and Simbrain, but an unresolved bug in the Java MIDI Sequencer stymied our forays in that direction. Python and TensorFlow were not themselves without issues, but the hurdles we encountered with them were surmountable.

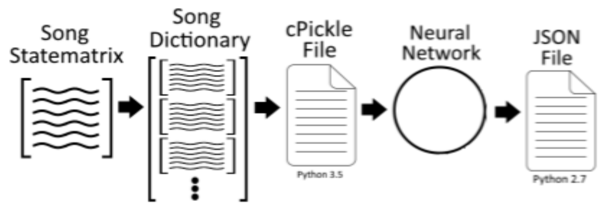
3 IMPLEMENTATION

Our project is implemented using Google's neural network software, TensorFlow. TensorFlow is an open source library that is maintained by Google[2] for numerical computation using data flow graphs and learning algorithms[12]. TensorFlow works with two major building blocks: tensors, which are essentially arrays of information, and operations, which are any mathematical calculation that could be done on a Tensor. Using tensors and ops, TensorFlow allows the programmer to computationally build graphs to model the behavior of an algorithm. In terms of Neural Networks, TensorFlow uses this graph to create a set of costs associated with each input - these costs are adjusted during the training process. After training is complete, the programmer can use those costs for some other operation, be it mathematical or generative.

In our case, we are using a Recurrent Neural Network with Long Short-Term Memory nodes, provided by TensorFlow's MultiRNNCell and LSTMCell classes[10]. A recurrent neural network differs from a normal neural network by taking the output from the previous layer and feeding it back into the network alongside the inputs. This looping process allows the network to pick up on relationships that it wouldn't otherwise. A Long Short-Term Memory node allows information to be retained "outside the normal flow of the recurrent network"[1]. By using these LSTM nodes the network more accurately trains over many iterations, as it has memory to fall back on in troublesome situations.

3.1 High-Level Overview

Before we can explain the process of the model, we must give a high-level overview of the generation process. The general flow of the system is explained in the following figure. There are three major pieces here that need explanation; the song dictionary, what cPickle is doing, and why we are using JSON.



The song dictionary is relatively straightforward - using a Python 2.7 script, we are converting every MIDI chorale we have in our database into a "state matrix" that defines the song. Then, those state matrices are packaged into what we are calling a song dictionary. We need the MIDI files in a format that Python and TensorFlow can work with, which is why we are converting them to state matrices. By putting them all in a python dictionary, we will be able to create batches (discussed later) from which the network will learn much easier.

After the dictionary is created, we need to feed it to the network for use in batching. However, there is one major problem here; the best library for working with MIDIs is written in Python 2.7 while TensorFlow exists within Python 3.5. Because of this, we need to move the dictionary from 2.7 into 3.5. In order to achieve this transfer, we use a Python serialization module called cPickle. The "pickling" process produces a file over 4 million lines long, which is now usable by our TensorFlow model.

After TensorFlow has done its training and then generating the song, we need to turn the resulting state matrix back into a MIDI file. This time, we are forced to go from Python 3.5 into Python 2.7. To do so we use JSON. Once that is done, we use another Python 2.7 script to turn the song back into a MIDI file, and voilÃă! The computer has created music.

3.2 State Matrix to MIDI and Vice Versa

One of the more major setbacks that occurred over the course of this project was the question of how to handle the MIDI files. After the initial Java debacle was put to rest, we were still not out of the woods; the best package by which we could process the MIDI files was called python-midi. It was remarkably well-documented, complete with not only the general idea of how MIDI files functioned, examples of how to build them from scratch using this package. The only issue was that the package was written in

Python 2.7, as was mentioned earlier, while TensorFlow operates in Python 3.5. This led to a series of intermediate steps as we converted the MIDI to a state matrix in Python 2.7, stored it as a cPickle file, which could be processed by TensorFlow in Python 3.5, and then reversed the process with JSON on the state matrix that the network put out such that we could use it to create the MIDI file that was our final product.

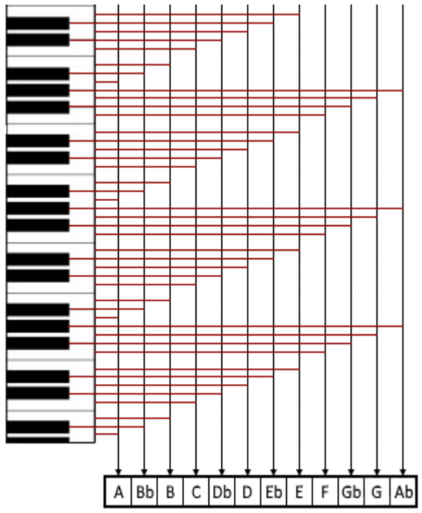
The state matrix is a 3D array of shape $(n, 128, 20)$, where n is the length of the song in ticks (the lowest resolution of a MIDI track). We iterate through the ticks in the MIDI track, and for every four ticks (for those among us who are music people, that means we run through the song at 16th note granularity), we check the MIDI for one of two events, NoteOnEvent or NoteOffEvent. Based on whether a note is being started or ended, we append [1, 1] or [0, 0] to the third dimension of the state matrix. The pitch of note started or ended is determined by the index of the note within the 128-long array âĂŞ there are 128 possible notes starting with C0, so the pitch would be its index away from C0. So it is that the state matrix has all the information of a MIDI, all in a format that could be read by anything that could read a text file.

The latter half of the transformations occurs after the network has generated its own state matrix. Building a MIDI file is slightly more involved than constructing an array, despite having a structure not dissimilar to one. According to the documentation, one must simply append the two different types of event to the track, concluding in the end of track event, but an attempt at this very method resulted in a strange, buzzing mess of approximately accurate pitch far removed from the clarity of the inputs. After much puzzling and many failed attempts, it turned out that, for some reason, we had to make two lists, one for NoteOn events and one for NoteOff, and append each to its respective list, and only after the lists were complete could we iterate through each of them and append each successive event to the track.

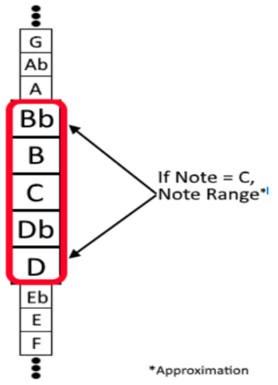
3.3 Idealistic Inputs

Neural networks require well-defined inputs on which to perform their operations. We have four separate inputs that we need to represent; the position, pitchclass, note range, and beat.

The first input is the position. This is relatively straightforward; we want the network to know what note it is considering at any one time. Therefore, we give each note a value which the network can use to represent it and compare it to other note values. Since we don't want to reinvent the wheel, position is represented by the note's MIDI value (Appendix 9.4).



The second input is pitchclass. Pitchclass is used to abstract away the octave component of music. The diagram above shows the general idea; we see four separate octaves on a keyboard, where each octave contains 12 distinct notes. However, these 48 notes are being dumped into 12 labeled "bins". The idea of pitchclass, then, is that by consolidating a similar note from across octaves into a single "bin", you can reduce the effect that different octaves have on the network. We do this because the relationships between notes and chords will always exist, no matter what octave you are in.



The third input is note range. Note range is used as one would believe; it represents the notes surrounding any one node so that the network can get a better understanding for spatial relationships between notes. The diagram above attempts to represent this, using an visual approximation of what the input is used for. This idea came straight from Johnson's implementation as a way to help produce chords.

The fourth and final input is beat. The beat is another way to say "time-step", and we keep track of this so the network can know where in a measure it is. This is also relevant when we have to convert the state matrices back into a MIDI format.

3.4 Actual Inputs

These inputs that we are considering are then used to create an input vector which can be inserted into our network. A single note vector was around size 80. Every single note has its own input vector and we are considering the whole range of midi notes. Every time-step in the state matrix has 128 notes and we must create a vector for every note. This process gives us a 2D matrix with 128 rows for our notes and 80 columns for the inputs. Finally, this means that a given song yields a 3D matrix with dimensions defined by the length of the song, the 128 notes we are considering, and the note input vector of size 80 or $[length, 128, 80]$.

Given the size of a song, the size of the input vectors we are creating, and the amount of vectors just a single time-step in the state matrix creates, we needed to break up our song into chunks of input as TensorFlow was running out of memory when presenting a whole state matrix as input. Therefore we needed to batch our input. By batching our inputs we decided to only consider subsections of pieces of songs instead of the whole song itself. In doing so we made sure to only start our batch on a measure and to process the whole measure to try to ensure a sense of time within the network. The length of this batch determines the patterns our network could potentially pick up in the music. Again we had to try to find a balance between the length of the batch to recognize patterns within music and the amount of training time the size introduced into the network. Shortening the batch resulted in more chaotic music, while increasing it drastically increased training time. Our final input vector then is of size $[batchlength, numberofbatchs * 128, 80]$.

3.5 Network Model Nodes

To completely understand the implementation of our model, we need to start at the basic building block of a network, which for us is the LSTM node (Appendix 9.1). It's important to understand that every node, regardless of its type, will take an input vector and output a corresponding vector with activations representing choices made by that node. Choices here will then differ depending on the type of node used. LSTM nodes are unique in that they take two input vectors, one being the current input and the other being the previous output of that node. Each individual LSTM node has a set of weights and biases that defined its structure and choices. In our model we are using just a basic LSTM node that has two input gates, an activation gate, and an out gate. The input gates decides which values to update in the incoming vectors, the activation gate updates the previous output into the current input vector, and the out gate is a *tanh* activation function that determines the final activations given by the node[4]. Every gate has weights and biases which then get adjusted as the network learns. We also use sigmoid activation nodes at the end of our network. These nodes just have weights and biases for an input gate, and their purpose is to scale the activations into the play probability.

3.6 Model Layers

An individual cell on its own is pretty useless, and therefore it's common practice to stack these cells into a layer. In typical feed forward networks these cells are not connected at all to each other, rather a node in a layer is connected to nodes in other layers (Appendix 9.2). LSTM networks follow the same structure however the nodes are connected to their layer counterparts in time (Appendix 9.3). Typically, and in our case as well, these layers are densely connected. This means that an LSTM node in a layer connects with its counterpart in time, as well as with every node in the previous and next layer of the current time. These connections are what the gates of our LSTM nodes are using as inputs and outputs.

3.7 Model Implementation

You can then multiply these layers or add more nodes to layers to get various degrees of training and accuracy. In general, more layers and nodes mean higher training accuracy, but suffer from drastically increasing the training time of the network. For our specific model we use two time layers each with 300 LSTM nodes, two note layers with 100 and 50 LSTM nodes respectively, and a finally a layer of two sigmoid nodes that generate the probability of the note being played.

We start off with our batch, which again is just piece of song that contains our note information and how that changes in time. The first time layer will loop through this batch in time sending its hidden outputs to the next time layer and to the next time-step. The next time layer does the same once it receives the hidden activations of the first time layer. ItâĂŹs important to note the time for each layer is considering the same time-step and does not loop to next time-step until both layers have processed the input. The result of this process creates a 3D matrix of dimensions $[batchlength, numberofbatches * 128, 300]$, where 300 is the hidden outputs of the time layer for every single note instead of the input vector now. We restructure this to be of the shape $[128, batchlength * numberofbatches, 300]$, so that we can now pass this information into the note layer and loop through the notes instead of time. Before we pass this into the note layers, we append back on the input vector in addition to the hidden outputs. We get a very similar output when compared to the first layer, $[128, batchlength * numberofbatches, 300]$. We then pass these final hidden activations into the sigmoid layer. The sigmoid layer then outputs a matrix of dimension $[128, batchlength * numberofbatches, 2]$. Here, the final dimension of two represents the play probability of the note and articulation probability of the note. When we are training we go even further. We then compare these probabilities with the probabilities of the original batch, which is pretty easy as the original notes are just 1 or 0 is they are being played or not. When comparing probabilities we can use what is known as cross entropy to define a cost function.

3.8 Cost Function

TensorFlow has a cross entropy function already defined, however this function assumes only a single output probability. We need to compare both the play probability and the articulation probability, which means we had to manually define a cross entropy function. We can then pass the cost (given by our cross entropy function) into an optimizer that TensorFlow provides. There are various optimizers available, the one we settled on was the RMSPropOptimizer, mostly because of its ability to quickly handle gradient decent in recurrent numeral networks. The optimizer will go back and adjust the weights of all the layers and all of the nodes in the network. The hope is that as the network is given a new batch, the cost function and the weights will get closer to an accurate prediction.

3.9 Generation

Once the network is done training we then need to actually use the trained weights to generate new music. The generation process is extremely similar to our prediction process. However, there are a few key differences in generating music. The biggest difference now is that we have no batch size or batch length. Both batch size and batch length are now one. When starting off the generating process we pass in a zeroed out matrix given our new dimensions. The dimensions of our input are now [1,128,80] completely filled with zeros. It may help to reduce our long periods of initial silence by appending an actual note structure, but we didnâĂŹt want to end up mimicking an actual chorale. This input then gets run through the network all the way past the sigmoid layer. At this point we then assume the network considers notes above 50 percent probability as being played and rounds these to one, and the other probabilities down to 0. If the play probability gets rounded down to 0 we also round the rearticulation probability down to 0 as well. If the note does get played then we need to round the articulation probabilities up and down from 50 percent as well. Instead of passing the output here into the cross entropy function we then use this output as a time-step and append it to the end of a state matrix. We use this output as the input to the network again to repeat the process, effectively building a song.

3.10 Tensorboard

Alongside our modeling structure, we took advantage of the application TensorBoard. TensorBoard is a sub-application to the TensorFlow API. It is a visualizing agent that directly interacts with the entire code flow, representing data, information pathing, and other important integrations[11]. What TensorBoard is able to do is scope out sectors of the code and assign them to a graph. This graph is able to determine the data flow and the directions it takes across the breadth of the program. In scoping these variables and data, they can be given easy to remember names to delineate them from other sections of the code. For example we were able to cordon off the tensors for the note and time layers into separate graph bubbles (Appendix 9.7). Not only that but we could label the sigmoid activator function applied to our network, the gradient descent operators, and the saving function which allowed TensorBoard to create what it calls "embeddings." In addition. One could open up these graphs and peek into the deeper, though understandably obfuscated, workings of the system in its entirety (Appendix 9.6).

What the embeddings allowed us to do was learn more about the network visually. To an untrained eye, the embeddings panel simply spat out crazy 3D dot graphs, especially earlier in development. However, once the system began to fall into place, we could see better that the results produced were something quite special. In the embeddings panel there are two relevant tabs we examined: The principal component analysis(PCA) and t-distributed stochastic neighbor embedding(t-SNE). With self directed learning we found these graphs had a lot to tell about our project's progress, and about neural networks in general. What the PCA would display were distributions our data would settle in after being ran through the network. That is to say, how our datum could be represented along three axes to illustrate how our notes and time signatures were being arranged. In some cases they would turn out quite lovely,

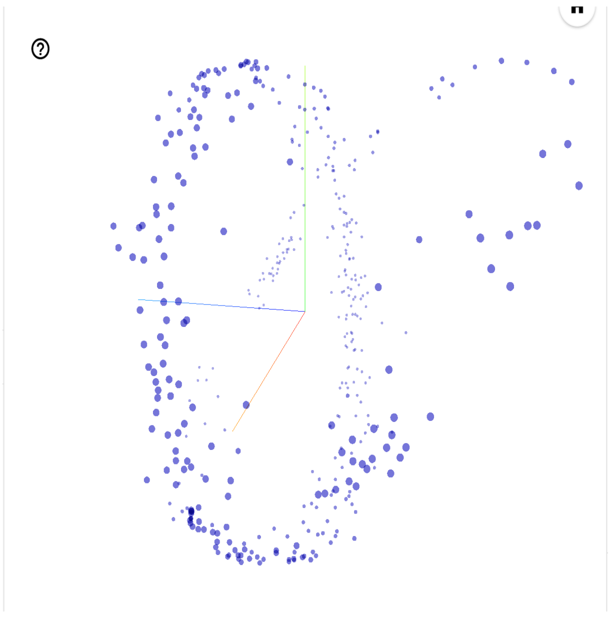


Figure 1: (PCA) A strong correlation in two rings

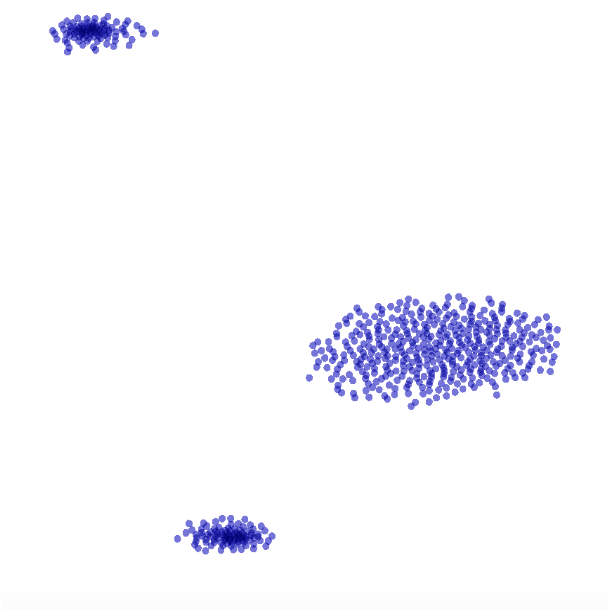


Figure 2: (t-SNE) Data finding troughs in the gradient

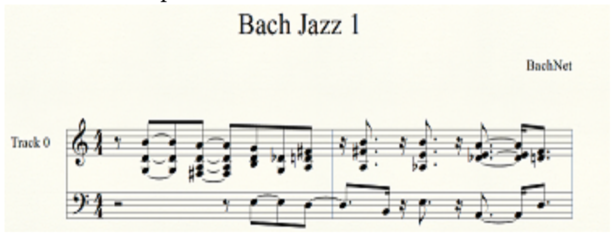
showing a very strong, discovered, correlation amongst the data (Figure 1). The t-SNE proved even more interesting, and not only because it was more engaging to examine. What the t-SNE could show was the network in motion; we could view a live performance of the gradient descent in action. In our more simpler data collections, we could see the data settling into place, finding a preferred action (Figure 2). Upon closer examination, we could see that this settling also showed strong correlations amongst the data (Appendix 9.5), demonstrating that similar data would strongly prefer to act like its neighbors (similar actions form similar patterns). With more dutifully trained data, which would run the risk of over-fitting, we could see interesting graphs showing a more worm-like correlation between small sets of data. This corresponded to the network learning more stringently and mindfully the patterns it observed. These strongly correlated sets would produce our characteristic "Bach Jazz," with a strong ear for the original patterns, but little in the way of stringing them together like the original (showing a novel result, however).

In our project we utilized this application and its functionality during important milestones to monitor code changes and how they affected the structure of our project. At various steps in development we could see which data, and how much of it, was being passed where, and operated upon how. Once the programs operated moreso as intended, we were able to see the fruits of our labor were not wasted; the machine was learning as intended, and was on its way to performing something new and exciting. TensorBoard proved integral to visualizing our project and progress, enhancing our confidence in our system's capability. What we also happened upon was a wonderful visualizer for an outside audience. Where someone well versed in t-SNE could approximate with ease the meanings of our data, someone with less specific knowledge could enjoy a better visual aid to the complex world of neural networks. They were not only a delight in showing our progress manifest (while proving entertaining to look at), these visuals could better help someone else understand our project and goals more easily.

4 EVALUATION

After a series of false alarms (ten-minute silences punctuated by strange thuds, single individual notes being held for minutes at a time, and one instance of clangorous senselessness), the neural network did generate music. Not strange, contemporary, avant-garde "music," but actual, recognizable, tonal music. It did quote the chorales upon which it was trained (quote meaning to take a fragment of melody from another song, exactly as it was, and reuse it), but there is no question that the material we produced was original. However, instead of a child of Bach's chorales, the product of the network is a cousin at best; while it produces melodies and progressions present in the source material, the rhythm is completely different, and some of the harmonies are not at all what one would expect of 18th century writing. The disparity between the two is such that we have begun referring to the product as "Bach Jazz," because the melody is so very clearly Bach, but played on offbeats and in syncopations, both of which of which are integral facets to jazz, and with chords one would only hear in a jazz setting.

The reason for the rhythmic deviation became apparent very shortly after the fault occurred. The mistake lay in how we selected the material on which to train the network. The chorales used were not all in the same time signature. Some were in triple meter, some in duple, and the network was not taught to distinguish between the two. So, though it would calculate the probability of the concurrent and subsequent notes using the same system (the tonal rules it learned from the training), it would choose between rhythms that would make sense either in triple or duple time for each note. This means that, even though the network started a phrase in duple meter, it could choose to continue it in triple, which would absolutely lead to the sort of rhythmic confusion that we received in the output material.



The reason for jazzy harmonies is not so clear. The best hypothesis we can forward is that since it calculated where each note in a chord should go on an individual note basis, it failed to consider the nature of the following chord as a whole. So while there might be a high probability that a D would be followed by an A and that a G should be followed by an F-sharp (as we see in the first bar of the product sheet music), that doesn't mean that the resulting G major chord to 2nd inversion V7 G chord progression should make any sense to a trained musician or in a classical piece. Interestingly enough, the second chord in the above progression is one that occurs frequently in jazz music, and listening to the full song seems to show that many such chords are present. So, it is possible that our probability calculations are such that jazz chords are produced entirely coincidentally. At the time of writing, this is but conjecture, and many hours of close reading would be needed to confirm it.

Another possibility is that our batch size was not sufficient to the task of actually adequately training the network. In the spirit of scientific inquiry, we attempted a training session with a batch size of one, and the result was nothing short of a cacophony. It is certainly a possibility that, even in seriousness, we made the self-same mistake that we committed on purpose earlier.

The group consensus with regard to the efficacy of our network and the final results of our project is that it was, for lack of a better word, a mixed success. We produced music, to be sure, but it was certainly no Bach chorale, close to a chorale as its quotes may be. The product was very clearly deeply influenced by the chorales on which the network was trained, to the extent that there is no portion that does not harmonically emulate the style, but the rhythms were not at all representative of the source material. Given the time frame within which we were operating and the scope of the undertaking, there is no small measure of pride that we were able to generate music with such a high degree of harmonic fidelity to Bach's work, but this satisfaction is tempered by the vast deviation from the aforementioned work in terms of time.

5 DISCUSSION

Our implementation, as mentioned, was based on a personal project by Daniel Johnson. We have discussed our structure in previous sections; Johnson's implementation is structured in a similar fashion, using a recurrent neural network and LSTM nodes. However, our implementation differs in three major ways. First off, Johnson used the neural network library "Theano" to generate the neural network while we used TensorFlow. While the two libraries are similar in many regards, TensorFlow offers three major benefits over Theano. Using the free application called TensorBoard, users can easily visualize their graphs and examine the interactions within it, as we've seen above. This allowed us to verify that the network seemed to be learning and working correctly. Alongside that, TensorFlow offers faster compile times than Theano, allowing faster testing of the system[2]. This was crucial since we were using this project to learn how to build neural networks. The last benefit of TensorFlow is that, as beginners, it abstracts away a lot of the complexities associated with graph construction. This in turn made our lives a lot easier and allowed us to focus on the larger ideas associated with the project.

After the use of TensorFlow, the second major difference of our implementation is our training library and training objectives. During training, Johnson used "short music segments" of 4/4 piano music compiled from many various sources[7]. For our training, we used segments of only chorale pieces composed by Bach. As discussed, these pieces include four voices maximum and generally contain a lot of similar structures and chord progressions. Because of this decision our output ended up sounding a lot more structured, and nearly met our original goal of producing music like Bach's chorales. Johnson's compositions, on the other hand, sounded more unstructured but also achieved better diversity. The final major difference is that, when deciding whether to play a note, Johnson's implementation "chooses a number between 0 and 1 randomly, and if this number is less than the play probability, it chooses to play the note. If it is greater, it does not"[7]. Our implementation, on the other hand, always uses 0.5 as the cutoff for if the note is played or not. This produces some interesting differences. Our compositions tend to have approximately the same number of notes played at any one time-step. Johnson's compositions, on the other hand, ended up having a range; in some sections there are a lot of notes played at any one time-step, and in other sections there are very few notes played.

From all of this, we do not claim that our implementation produces music better than Johnson's. Ours is very different and is more of a supporting example to his work. In addition, it is a great learning tool because we can examine two similar models working in similar fashions, but using two different libraries.

6 FUTURE WORK

Pleased though we are with the results of our project, there is certainly room for improvement. As entertaining as the Bach jazz was, it was not 100% what we wanted, and thus the rectification of the factors that lead to the distortions within the output chorale is a priority. Not only that, there are still occasionally gaps as long as 5 minutes before any music starts to play in our generated pieces, and we still do put out the occasional total failure (weird beeping, strange ticking sounds), so it goes without saying that our infrastructure is not flawless.

To that end, we will need to separate the datasets we're using into chorales in duple and triple time, to test the working hypothesis that the strange rhythms are due to mixing the two during generation.

Once we can consistently and reliably produce chorales like those Bach wrote, a whole world of possibilities opens up in terms of potential applications. The original idea for this project included some analysis of the chorale the network produced, annotating sheet music with information as to which rules of tonality were satisfied by particular important progressions. This would make an invaluable tool for beginners studying music theory. Not only that, chorales in general are unparalleled material with which to teach ensembles to balance (make sure no section covers up another), tune, and play in-time. The trouble with chorales as they are, however, is that one risks falling into relative complacency due to their relatively simple nature; it's much more difficult to bring oneself to pay attention to the other voices in an ensemble when one is falling asleep from boredom playing the same line for the hundredth time. Chorale generation would make finding a new chorale trivially easy, and thus make effective teaching somewhat easier as well.

The original goal of the project was to produce an annotated fugue based on a phrase submitted by the user, which, given the complexity of fugues, was shown very quickly to be impossible. Even having transitioned to chorales, we did not have sufficient time to incorporate user input in the chorale to be generated. The final (if any iteration of a program such as this could be said to be final) product of foreseeable future work would be the addition of this functionality, which would, in our opinions, immeasurably increase the efficacy of this software as a teaching tool. Instead of generating the whole chorale at once, and spitting only the final product out to the user, it would be interesting to show the writing process in stages; this could manifest itself in producing a set of annotated sample outputs, perhaps also including some sort of analytic process that would not only show what made the valid chorales valid, but also the failings of those that didn't meet the standards of strict tonality.

7 CONCLUSION

In the end, we feel that our project was relatively successful. We have created a recurrent neural network that accurately trains on a dataset of Bach chorales and is able to correctly predict notes as they would appear in one of the training pieces. In fact, the system is generalized enough that we could run it on any correctly-formatted dataset and the network would run successfully. While our system has shown problems during the generation process, we have demonstrated that it works and are positive that with some effort and more time, we could make it work every time.

While there are a lot of possible extensions to the current code base, we feel that this project is an accurate representation of the time we had, what we learned throughout the semester, and what skills we have developed through the past four years. In fact, we are very pleased with ourselves - we triumphantly navigated many hurdles throughout the semester, and were successful in producing some very interesting music! We know that we will apply much of what we learned this year in our future ventures.

8 ACKNOWLEDGEMENTS

We would like to thank Professor Brad Richards and Professor America Chambers for allowing us to attempt this project and for helping us along the way, respectively. It is with their assistance and belief that we were moderately successful in our endeavor.

We would also like to thank Daniel Johnson for his inspiring work and for assisting us comprehend his implementation. If it were not for his well-written write-up and his ability to understand our poorly-worded questions, we would have remained dazed and confused for much longer.

Dynah would like to thank her good friend, Catherine, who kept her going through this confusing capstone experience. Trevor would like to thank his mother for enforcing a deadline on us. Andy would like to thank Dr. Gerard Morris, for whom we owe the concept of our project to. David wishes to reinforce the thanking of America Chambers, for all her help she gave us.

Lastly, we would all like to thank our individual mentors, parents, and friends for guiding us through the last four years.

REFERENCES

[1] DeepLearning 4J. 2017. "A BeginnerâĂŹs Guide to Recurrent Networks and LSTMs". (2017). https://deeplearning4j.org/lstm.html
[2] DeepLearning 4J. 2017. "Comparing Frameworks: Deeplearning4j, Torch, Theano, TensorFlow, and more". (2017). https://deeplearning4j.org/compare-dl4j-torch7-pylearn
[3] Rasit Ata. 2015. "Schematic diagram of a multilayer feed-forward neural network". (2015). https://www.researchgate.net/figure/276327285_fig1_Fig-1-Schematic-diagram-of-a-multilayer-feed-forward-neural-network-3
[4] Christopher Colah. 2015. "Understanding LSTM Networks". (2015). http://colah.github.io/posts/2015-08-Understanding-LSTMs/
[5] Douglas Eck. 2016. "Welcome to Magenta!". (2016). https://magenta.tensorflow.org/welcome-to-magenta
[6] Dave Gershgorn. 2016. "Can We Make A Computer Make Art?". (2016). http://www.popsci.com/can-computer-make-art
[7] Daniel Johnson. 2015. "Composing Music With Recurrent Neural Networks". (2015). http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/
[8] Daniel D. Johnson. 2017. *Generating Polyphonic Music Using Tied Parallel Networks.* Springer International Publishing, Cham, 128–143. https://doi.org/10.1007/978-3-319-55750-2_9
[9] Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.
[10] TensorFlow. 2017. "API Documentation". (2017). https://www.tensorflow.org/api_docs/
[11] TensorFlow. 2017. "TensorBoard: Visualizing Learning". (2017). https://www.tensorflow.org/get_started/summaries_and_tensorboard
[12] TensorFlow. 2017. "TensorFlow - An Open-Source softward library for Machine Intelligence". (2017). https://www.tensorflow.org
[13] Michael David Watson. 2015. "Recurrent Neural Network Tutorial, Part 1". (2015). http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/
[14] Wintergatan. 2016. "Wintergatan - Marble Machine (music instrument using 2000 marbles)". (2016). https://www.youtube.com/watch?v=IvUU8joBb1Q

9 APPENDIX

9.1 LSTM Chain

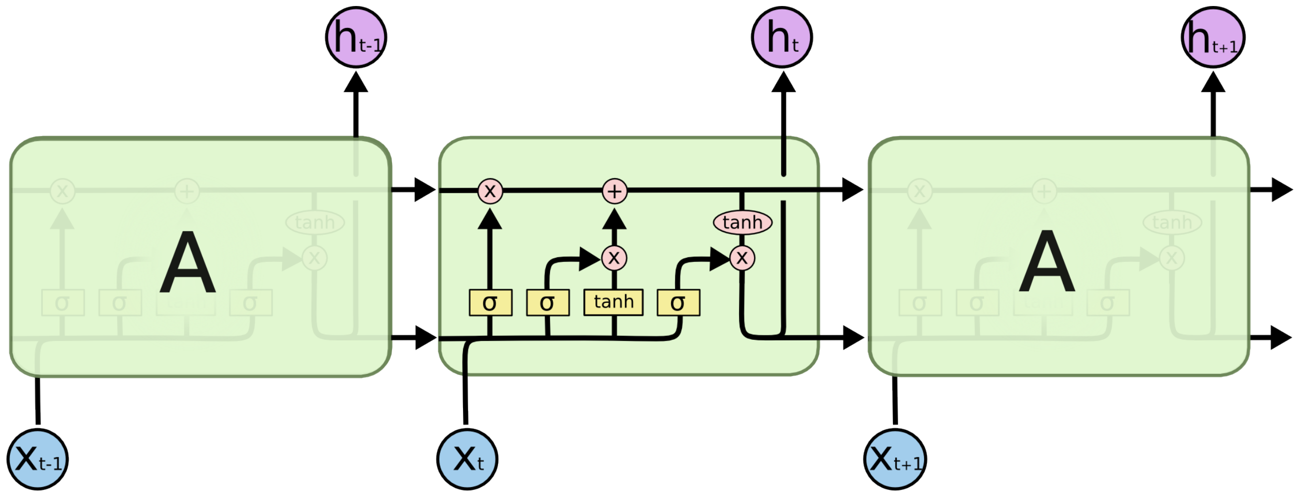


Figure 3: This depicts a chain of LSTM nodes as information progresses through them[4].

9.2 Feed-Forward Network

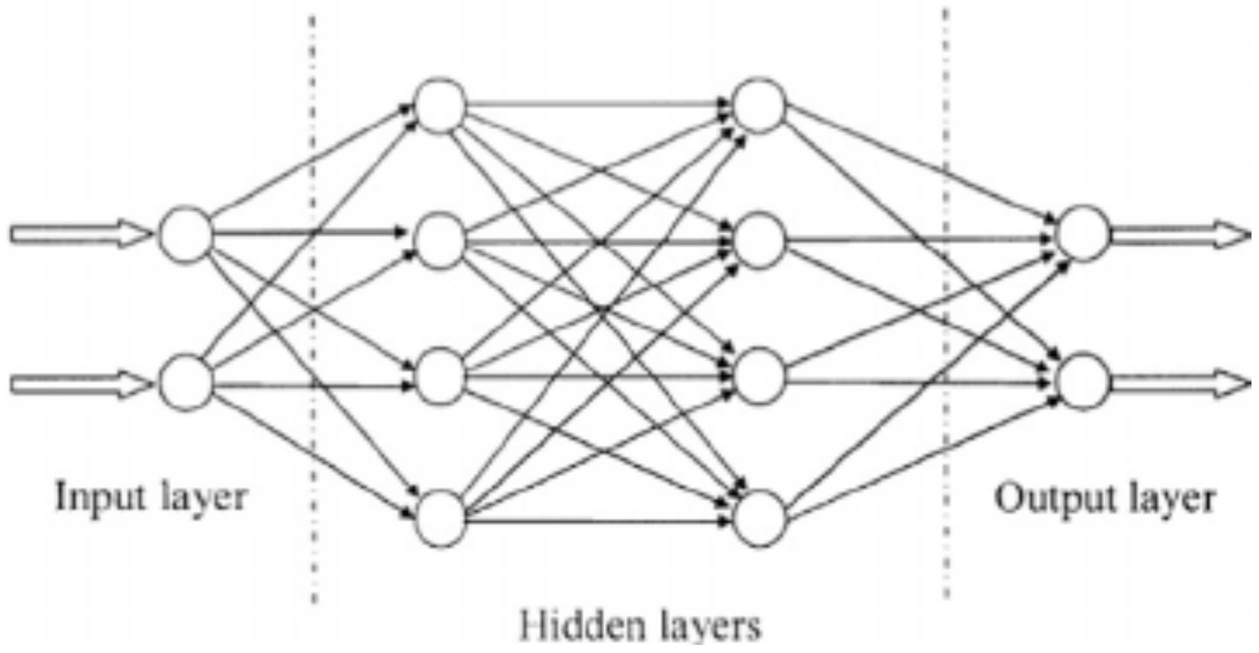


Figure 4: This depicts a feed-forward network, which is the building blocks for a recursive network[3].

9.3 Recurrent Neural Network

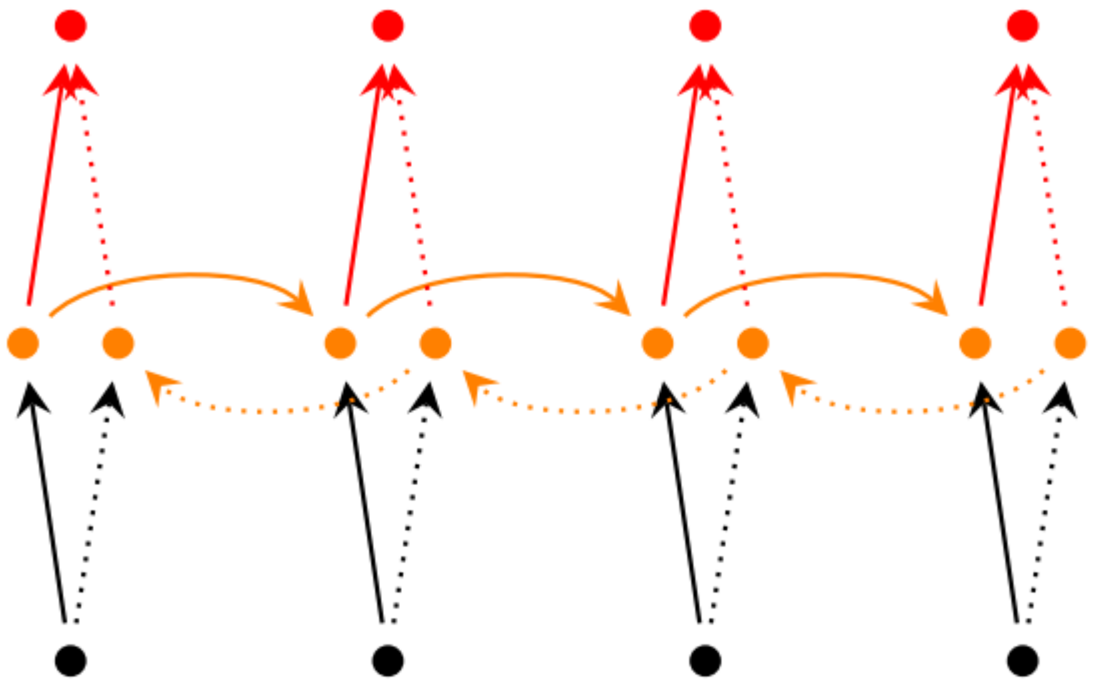
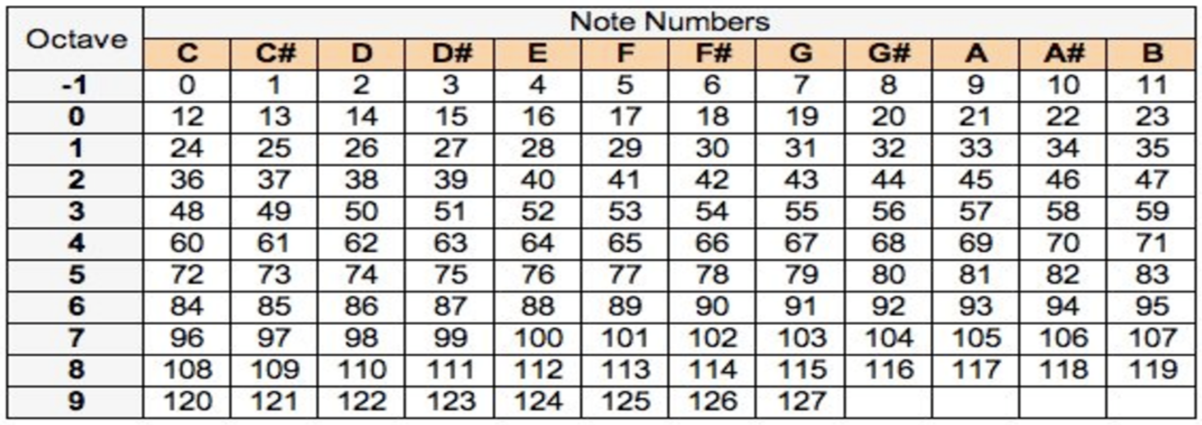


Figure 5: This depicts a recurrent neural network[13].

9.4 MIDI Note Values



Octave	Note Numbers											
	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
-1	0	1	2	3	4	5	6	7	8	9	10	11
0	12	13	14	15	16	17	18	19	20	21	22	23
1	24	25	26	27	28	29	30	31	32	33	34	35
2	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59
4	60	61	62	63	64	65	66	67	68	69	70	71
5	72	73	74	75	76	77	78	79	80	81	82	83
6	84	85	86	87	88	89	90	91	92	93	94	95
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	118	119
9	120	121	122	123	124	125	126	127				

Figure 6: This chart are all the MIDI note values possible on a keyboard.

9.5 Closest Neighbors

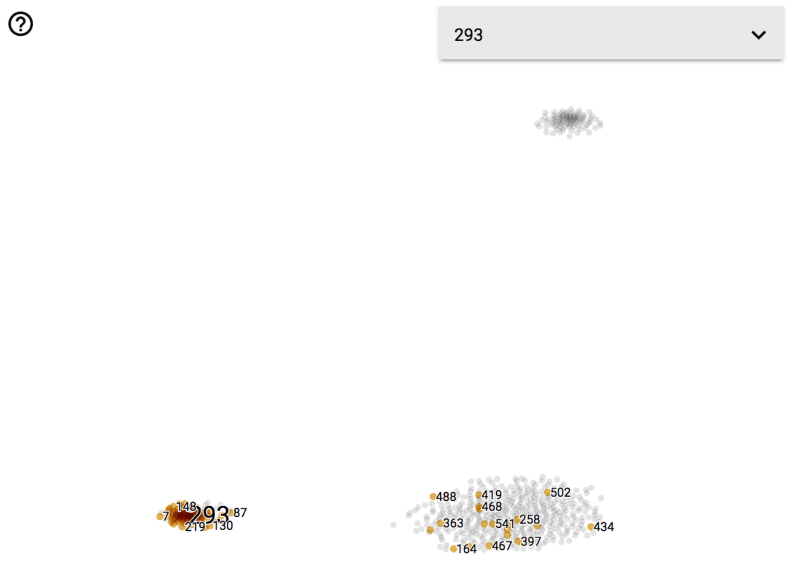


Figure 7: Selecting a data point within a t-SNE cloud shows the closest neighbors. Through all the tests these neighbors always remained close.

9.6 Deeper Look

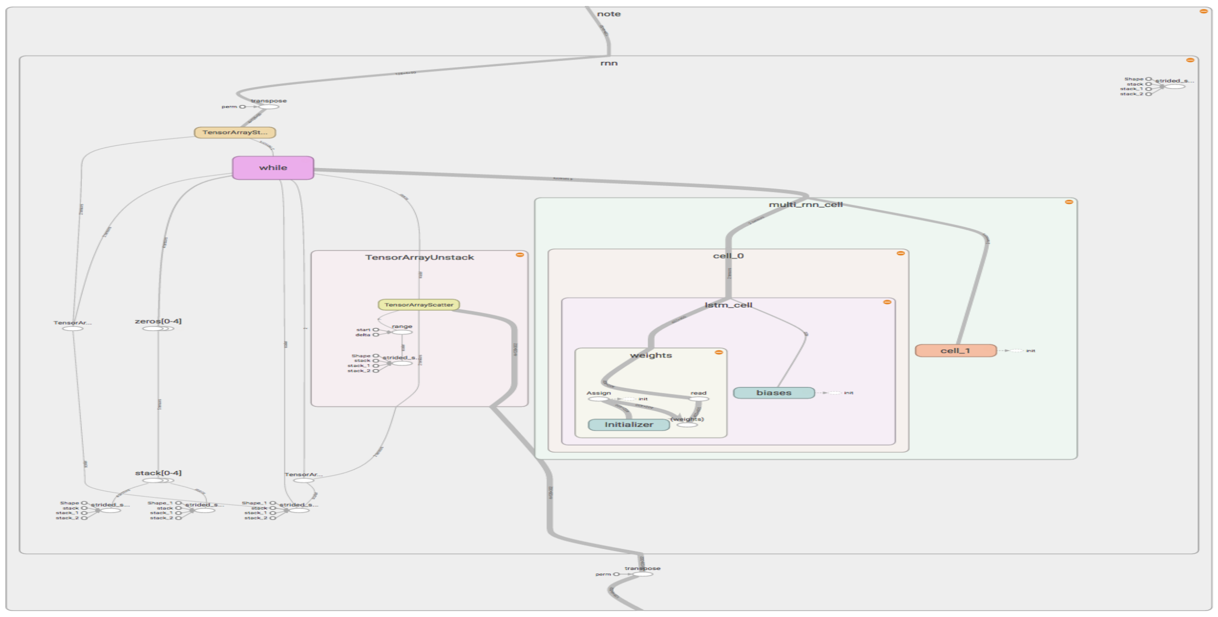


Figure 8: Note module expanded to show contents

9.7 Final Shape

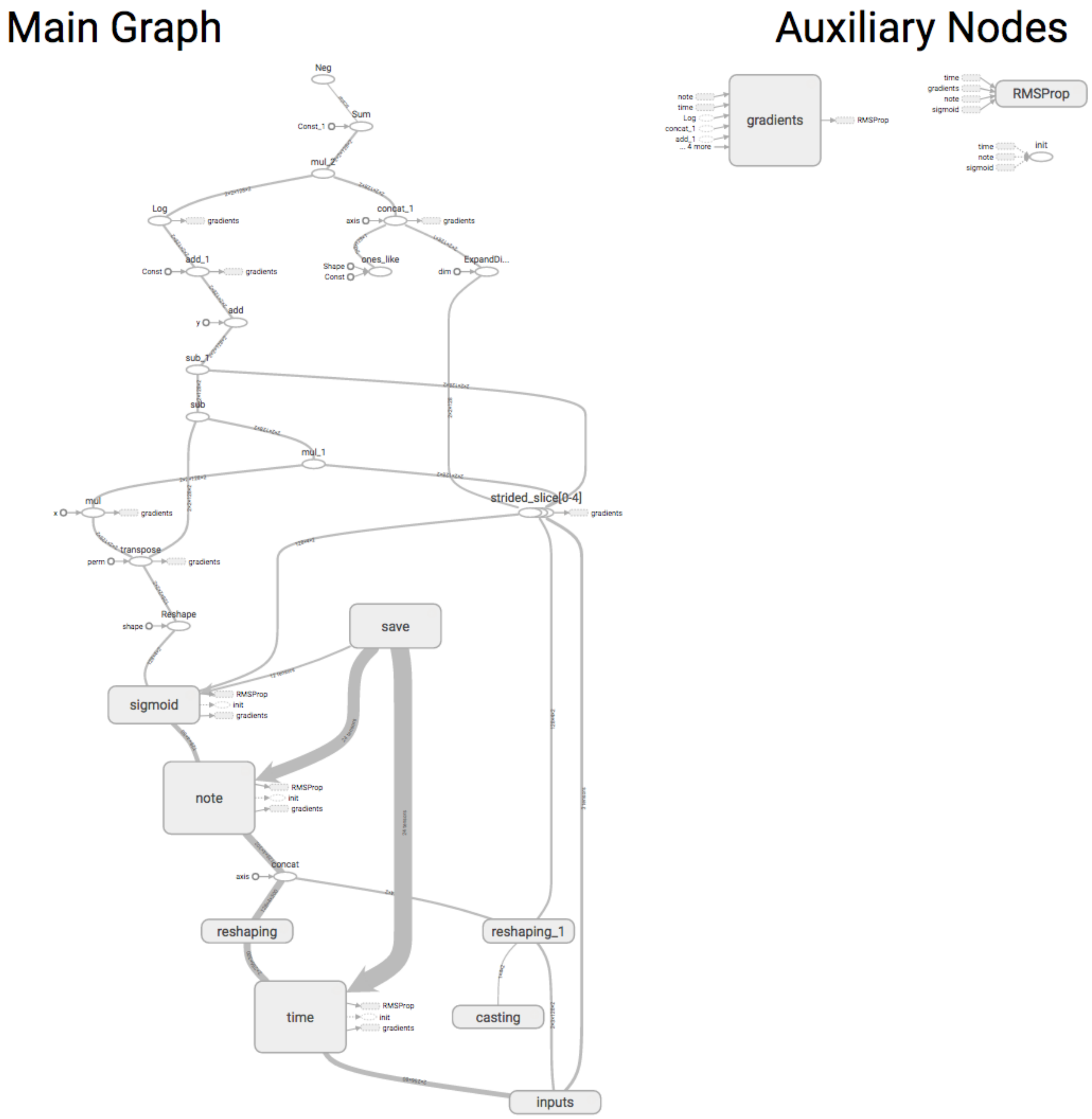


Figure 9: The final shape of our graph