JACOBS
UNIVERSITY

Percussion Generation and Accompaniment Using Echo State Networks

by

Alexandru Sasu

Bachelor Thesis in Computer Science

Prof. Dr. Herbert Jaeger
Bachelor Thesis Supervisor

Date of Submission: May 17, 2019

Jacobs University — Focus Area Mobility

With my signature, I certify that this thesis has been written by me using only the indicates resources and materials. Where I have presented data and results, the data and results are complete, genuine, and have been obtained by me unless otherwise acknowledged; where my results derive from computer programs, these computer programs have been written by me unless otherwise acknowledged. I further confirm that this thesis has not been submitted, either in part or as a whole, for any other academic degree at this or another institution.

Signature

Place, Date

[illegible]

Bremen
17 May 2019

Abstract

One of the indispensable parts of any song is the percussion section, and creating a good one is not easy. This can be a very challenging task for melody composers and is a setback for many of them. Solving this problem with a system that, when given a melody, creates a percussion section, would be useful for creative applications and also for further research in computer creativity.

This thesis describes a experiment conducted in order to explore the use of Echo State Networks (ESN) for a Percussion Generation and Accompaniment task, and its results. The experiment takes a song containing bass and drums as input for training and for testing, in order to learn to generate a drum beat when given a bass line, and can be considered an extension of Tomas Pllaha's Bachelor thesis, who discussed the procedure of using ESNs for a bass line accompaniment generation.

Many methods and ideas have been previously used for music and beat generation, including Machine Learning and other types of Recurrent Neural Networks. The novelty of this project is the use of ESNs. They prove to be very practical as not only are they much easier to train and faster than other methods, but being Recurrent Neural Networks, they are also very suitable for predicting time series. Considering music is a time series expressed as the succession of different sounds and beats, that made them a proper tool for the task at hand.

The results presented in this thesis show that the generated beats are good and pleasing for accompaniment, and that ESNs have a high potential in the generation of percussion.

Contents

1 Introduction **1**

2 Statement and Motivation of Research **2**
2.1 Echo State Networks 3
2.2 Statement of Research 5
2.3 Research Questions 5

3 Description of the Investigation **6**
3.1 Preparing the data 6
3.2 Network design 8
3.3 Error Metric 8
3.4 Optimization 9
3.5 Post processing the output 10
3.6 Output Feedback 10

4 Results **11**

5 Conclusions **13**
5.1 Further investigation 14

6 Acknowledgements **14**

1 Introduction

The idea of creating a music generator and especially a percussion generator is surely not a new one or an easy one. Along the years it has been tried numerous times with numerous methods to do so and some interesting results have been found.

Early on, algorithmic composition was tried [1] using random number generators, rule-based systems and other algorithms. This method showed promising results as it could compose music without any human intervention, but it was more useful for testing theories about computer composition, rather than composing, as its products lacked originality. Neural Networks were also tried early on [2] as an extension of the algorithmic composition where instead of using a transition table, a recurrent auto predictive connectionist network was used. This would produce enjoyable results occasionally, but in general it lacked coherence. Markov chain approaches were also proposed [3] but they have to have an underlying structure which is not ideal for composition. In addition, music's requirement for long-term order breaks the Markov chain rules. Another interesting and unconventional approach suggested the use of Memristors [4] and the theory presented seems to have potential, but it was never put into practice as not only it would be very costly, but also it is not clear that it would work. When it comes to percussion generation in particular, machine learning techniques have been tried [5] and showed quite coherent and pleasing results. Their drawback is they cannot be used interactively. The method that is best working at the moment is developed in the Magenta project of Google Brain and it is based on inverse sequence transformations [6] which converts rhythm into a drum beat using multilayer perceptron neural networks. This offers really pleasing results but unfortunately it is not interactive.

After these attempts, Thomas Pllaha used Echo State Networks (ESN) to create a Music Accompaniment tool, which, given a melody would generate a bassline for it [7]. I will try to continue his research by creating an accompanist that will create a drum beat. ESNs have also been previously used for drum generation [8] but only for generating stand-alone drum-beats, not for accompanying another instrument.

Recurrent Neural Networks(RNN) in general are a good tool for time series prediction [9], but the traditional gradient-descent-based training is hard and costly, making them not easily usable for most tasks, including music generation and accompaniment.

An echo state network is an RNN that, compared to normal RNNs, is much faster and easier to train, saving both time and costs while setting up and training the network. Being a RNN, a ESN has the normal components: input nodes, a reservoir of nodes and output nodes. The difference in ESNs comes on the training side, as seen in Figure 1. Compared to the standard RNNs that have their input, reservoir, and output weights trained, ESNs have fixed weights for the input and reservoir nodes and only their output weights are trained, hence the speed and easiness of training. This difference can be observed in Figure 1.

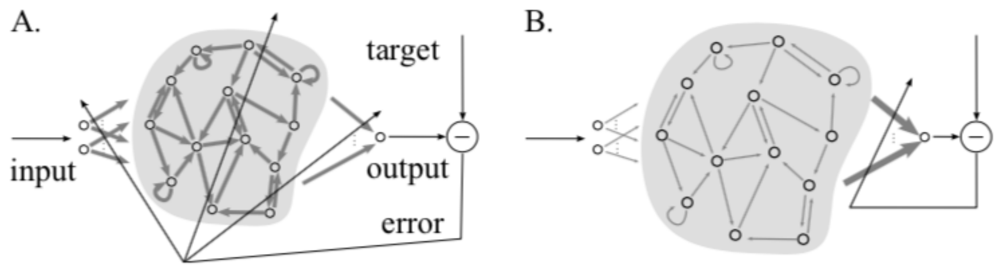

Figure 1: The difference between a full gradient descent training of RNN (A.) and the ESN training (B.). Image taken from [10].

Since the outputs of an ESN are a combination of the reservoir units, training the network comes down to the following system of linear equations:

$$Y^{target} = W^{out} X, \tag{1}$$

where Y^{target} is the output target, X is the reservoir response to the training input and W^{out} is the weight matrix of the system. The system can be solved for W^{out} using linear regression in order to find the optimal W^{out} for which the smallest mean square error between $y(n)$ and $y^{target}(n)$ will be found.

There are multiple ways to solve this linear regression task, but each one has different drawbacks related to stability and computational cost, so the most commonly used one is ridge regression, as it offers good stability at a low cost [11]. This will be further discussed in the following sections.

ESNs are suitable for music generation and even for music accompaniment for a few reasons:

1. Songs can be described as time series, so ESNs would be practical to predict their continuation.
2. They can be easily constructed, as they have a simple structure.
3. They are much easier and cheaper to train than the other approaches.
4. The output generated at a certain step only depends on the past steps. This is favorable as not depending on the future steps means the accompaniment can be generated interactively.

This document is structured as follows. Section 2 will briefly explain how Echo State Networks work and outline what this guided research project has tried to achieve, Section 3 will go through the technical setup and the optimization of the model, Section 4 will go through the results of the experiment and explain what they represent, and finally, Section 5 will contain conclusions and possible improvements.

2 Statement and Motivation of Research

In this section, I will introduce the important points about ESNs, and then go on to outline the questions addressed in this project.

2.1 Echo State Networks

As RNNs are suitable for time series prediction, they are able to mimic creativity when it comes to the prediction of music. As an ESN constitutes an RNN that is easier to train, it has the potential to achieve pleasing results in the task of music generation. In this subsection, I will briefly explain what an ESN is and how it works. The description given of ESNs in this section follows closely Mantas Lukoševičius's "A Practical Guide to Applying Echo State Networks" [11]. I will also be using the same notations as in the guide in order to increase understandability.

ESNs have the same components and at first glance are just normal RNNs. They are a network that gets data as a time-series over the input nodes, passes it into a reservoir of nodes (a set of nodes that includes cycles and in which units can also lead to themselves), and then outputs it through the set of output nodes. Even though the output might have a different number of values (units) than the input, the length of the resulting time-series has to be the same as the length of the one received. The difference between normal RNNs and ESNs becomes apparent when training the network, as ESNs have fixed weights for the input and the reservoir units, and therefore they only train the output weights. One could see an example of an ESN in Figure 2.

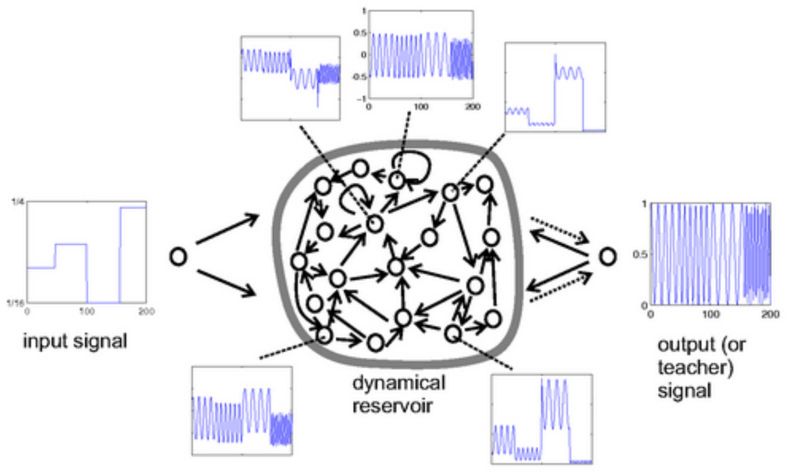

Figure 2: The basic schema of an ESN, illustrated with a tuneable frequency generator task. Solid arrows indicate fixed, random connections; dotted arrows trainable connections. Image and caption taken from [10].

ESNs solve a supervised temporal Machine Learning task where the training data is given as an input signal $\boldsymbol{u}(n) \in \mathbb{R}^{N_u}$ and a target output signal $\boldsymbol{y}^{target}(n) \in \mathbb{R}^{N_y}$ $(n = 1, ..., T$ is the discrete time and T is the number of data points). The network must learn to output $\mathbf{y}(n) \in \mathbb{R}^{N_y}$ so that $\boldsymbol{y}(n)$ matches $y(n)^{target}$ as well as possible, meaning that the error $E(\mathbf{y}(n), \mathbf{y}^{target}(n))$ is as small as possible. This is most commonly measured as Root-Mean-Square Error:

$$E(\mathbf{y}, \mathbf{y}^{target}) = \sum_{i=1}^{N_y} \sqrt{\frac{1}{T} \sum_{n=1} T(y_i(n), y_i^{target}(n))^2}, \quad (2)$$

This error is also sometimes normalized by the variance of the target in order to get an error that has an absolute interpretation, but the normalization and the square root are only used for better human interpretation, as minimizing the Normalized Root Mean

Square Error or the Root Mean Square Error is the same as minimizing the Mean Square Error.

The update function for ESNs usually looks like this:

$$\tilde{\boldsymbol{x}}(n) = \tanh\left(\boldsymbol{W}^{in}[1; \boldsymbol{u}(n)] + \boldsymbol{W}\boldsymbol{x}(n-1)\right), \tag{3}$$

$$\boldsymbol{x}(n) = (1-\alpha)\boldsymbol{x}(n-1) + \alpha\tilde{\boldsymbol{x}}(n), \tag{4}$$

where $\boldsymbol{W}^{in} \in \mathbb{R}^{N_x \times (1+N_u)}$, $\boldsymbol{W} \in \mathbb{R}^{N_x \times N_x}$,$\boldsymbol{u}(n)$ is the input at time n, $\boldsymbol{x}(n)$ is the reservoir neuron activation vector for time n and it is the the combination of $\boldsymbol{x}(n-1)$ with the update vector $\tilde{\boldsymbol{x}}(n)$, $[\cdot;\cdot]$ means concatenation, and $\alpha \in (0, 1]$ is the leaking rate, which is actually the percentage the update vector contributes to the updated vector. If α is 1 then $\boldsymbol{x}(n) = \tilde{\boldsymbol{x}}(n)$.

Now that this information has been stated, we can obtain the output formula as follows:

$$\boldsymbol{y}(n) = \boldsymbol{W}^{out}[1; \boldsymbol{u}(n); \boldsymbol{x}(n)], \tag{5}$$

where $\boldsymbol{y}(n) \in \mathbb{R}^{N_y}$ is the output at time n, and $\boldsymbol{W}^{out} \in \mathbb{R}^{N_y \times (1+N_u+N_x)}$ is the output weight matrix. The used notations and the training method can be observed in Figure 3.

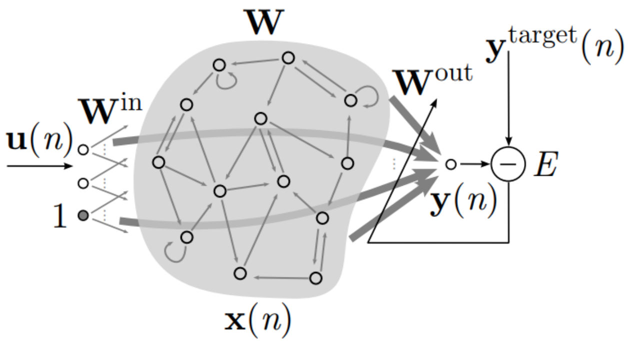

Figure 3: An Echo State Network. Image taken from [11].

The goal of training an ESN is the following: by having a set of input data and a set of corresponding output data, to generate a Network that given a testing input signal, can approximate as well as possible the target signal. If this is achieved, one can assume that the network can be used for generating output also given new data.

The algorithm of training an ESN can be summarized as follows:

Step 1. Generate the random reservoir. Even though it is a "random" reservoir, the process of generating it is not trivial. One has to pick the number of input and output units based on the data, and then choose the number of internal units, which can have a very large value, as ESNs are computationally cheap, but the general wisdom is the initial value should be small and increased afterwards. After picking these numbers, one has to randomly generate $\boldsymbol{W}^{in}$ and scale it, then randomly generate $\boldsymbol{W}$ in a sparse way so that every node is connected to a small fixed number of nodes (note that a node can also be connected to itself), and then find a leaking rate α that suits the speed at which the input and/or the output changes (usually done by trial and error).

Step 2. Select an initial value for $\boldsymbol{x}(0)$ (e.g., $\boldsymbol{x}(0) = 0$) and then run the network with the training input $\boldsymbol{u}(n)$ in order to generate every $\boldsymbol{x}(n)$ and create the matrix $\boldsymbol{X}$, in which each column i will be structured as the vertical concatenation $[1; \boldsymbol{u}(i); \boldsymbol{x}(i)]$.

Step 3. After having both the matrix $\boldsymbol{X}$ and the matrix $\boldsymbol{Y}^{target}$, which is the concatenation of the target values, we can finally solve Equation (1), which I will write again here for easier access:

$$Y^{target} = W^{out} X. \tag{6}$$

This equation could be solved by right-multiplying with the inverse of $\boldsymbol{X}$ if $\boldsymbol{X}$ would be invertible, but considering it is not square, it is also not invertible. So what one should do is right-multiply with the transpose of X, in order to get:

$$W^{out} X X^T = Y^{target} X^T, \tag{7}$$

and then multiply by the inverse of the product of X with its transpose and get:

$$W^{out} = Y^{target} X^T (X X^T)^{-1}, \tag{8}$$

but this equation often leads to numerical instabilities when inverting XX^T. So the most commonly used way to solve this is the following:

$$W^{out} = Y^{target} X^T (X X^T + \beta I)^{-1}, \tag{9}$$

where β is a regularization coefficient and I is a size $(N + K)$ identity matrix. The method from this step is called ridge regression.

Step 4. Compute $\boldsymbol{y}(n)$ for input $\boldsymbol{u}(n)$ by driving it through the network with the newly found $\boldsymbol{W}^{out}$.

As the process of creating the network was simplified here, it might seem more trivial than it actually is. The whole process is thoroughly explained in Mantas Lukoševičius's ESN guide [11].

2.2 Statement of Research

A drum accompaniment generator would be very practical for any aspiring musicians in order for them to practice and compose even when they do not have the comfort of doing so with a drummer or a band.

In this guided research project we experimented how Echo State Networks can be used for this task.

2.3 Research Questions

The following is a list of the questions I addressed in this guided research:

- How can ESNs be used for generating percussion based on a melody that is meant to be accompanied?
- What are the global parameters that will give the best results?

- How to represent the input and output in order to get the best results?
- How to obtain the training data and then pre-process it?
- How to convert the network output to a drum beat?

3 Description of the Investigation

The following section presents the technical details of the investigation by explaining how the data was processed and how the ESN was built and optimized. All the code that was written and mentioned here can be found at `https://github.com/sasualx/BSc-Thesis`.

3.1 Preparing the data

As an initial state of the data, I picked the GuitarPro format. GuitarPro is a computer program that is used for reading and writing sheet music for multiple instruments. It is usually used but not exclusively in the rock genre which almost always contains both bass and drums, which are the instruments we need for the experiment. The reason for picking this format is that GuitarPro files are widely and freely available online.

After obtaining the data, one has to find a way to manipulate the files and then automate the manipulation process to be performed on all the files.

I found the open and free to use PyGuitarPro [12] python package which allows people to open and manipulate GuitarPro files in a python program.

I wrote a program that uses the said package in order to filter the data. The program reads a GuitarPro file, checks if it contains drums and bass and proceeds to create a new file in which only the two relevant instrument tracks are retained, always having drums as the first track and bass as the second one. I also wrote a script that automatically ran this program for all the files.

GuitarPro files are written in a way that makes them hard to process further than in the previous step, and also using them as the network input would highly limit the use of the network. Therefor, after having this consistent set of GuitarPro files, I planned to convert them to MIDI.

MIDI, which stands for Musical Instrument Digital Interface, is a widely used format for music which does not contain audio or continuous information about the tracks, instead, tracks are divided into events. Every event has a time stamp and if it is a note being played the event also has information about it. This makes such files light and easy to use.

The GuitarPro program has a built-in tool for converting files to MIDI, but this has to be done separately for each file, which is impractical. At first I was planning on creating an automatic hotkey pattern in order to convert each file but the time it would have taken to convert all of them would have also been impractical. Luckily I found the following C# library [13] which converts GuitarPro files to MIDI. The library was written for being used with UnityEngine but when I attempted to do so, the project seemed to be corrupt, so I had to adapt it in order to work with mono [14], which is a tool used to run C# code in the Unix terminal. This allowed for running a script that automatically converted each

GuitarPro file to MIDI. Because MIDI is not human-readable, I used the tool midicsv [15] in order to convert all the files into CSV for easier readability, interpretability, and further processing.

After having a consistent set of CSV files with the training data, a way to represent the data so that it would be best for the network input had to be found. I chose 2 values to represent the bass at each time step, the first one expressing if a note is played at the certain point, taking value 0 if there is no note, value 1 if a note is played, and value 2 if a note has been previously played and can still be heard, and the second one describing the pitch of the musical note that is being played, having a standard MIDI value which for bass ranges between 28 and 75.

For the drums I experimented with multiple ways to represent the data which led to different results that will be presented in Section 4. The best way to represent the drums was to map each MIDI value to a certain drum. So I mapped every possible MIDI value to one of the following drums: Bass Drum, Tom, Snare, Cymbal, and Hi-hat, having a binary value for each one, representing no note played if the value is 0, or note played if the value is 1.

One has to continue by dividing every musical measure into a fixed number of divisions based on time signature. I chose to divide each one into 8. When a number of divisions has been picked, one can convert the CSV files into a stream of input data by going through every track in the file and save values of zero in between notes (time steps where no note is played), and save the needed values when notes exist.

A sample of the training data can be observed in Figure 4.

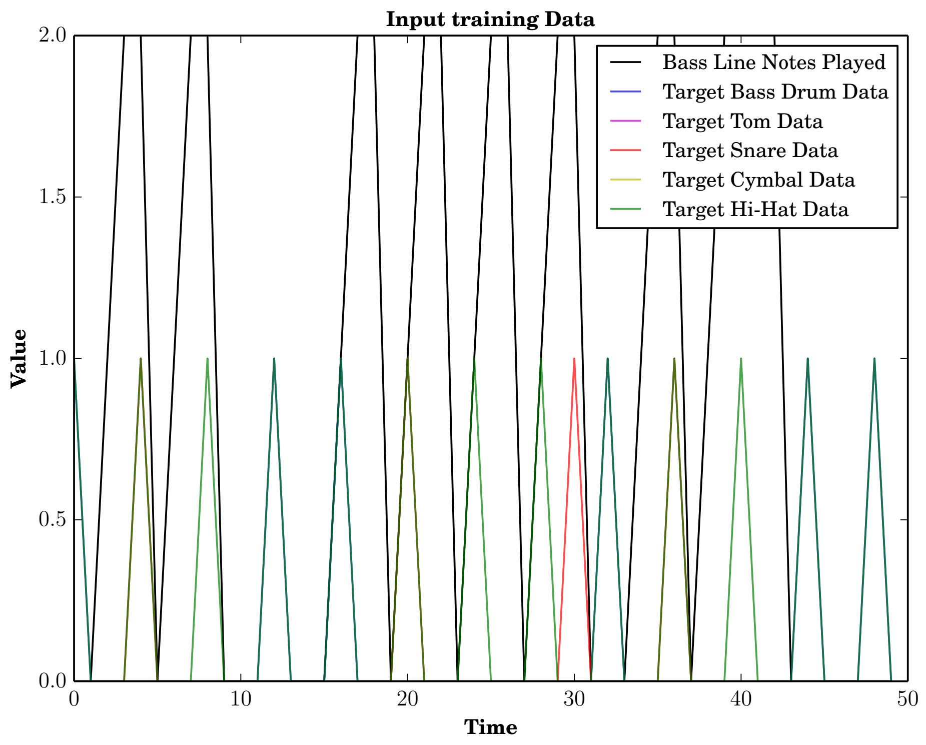

Figure 4: Network Input Data. As multiple drums can be played at the same time, overlap occurs in this figure.

3.2 Network design

As previously stated, I built the network according to the ESN guide [11]. I started by choosing initial values for the global parameters (i.e., reservoir size, leaking rate, input scaling, spectral radius, reservoir sparsity). Most of these initial values are irrelevant as later on they will be optimized. The relevant ones are the reservoir sizem which I chose to be small initially (100 nodes), and the sparsity of the reservoir. I chose for each node to have connections towards 10 nodes.

I proceeded to generate the input weights matrix W^{in}. This is the matrix that transitions the input to the reservoir so it has size $reservoirSize \times inSize$, where $reservoirSize$ is the number of nodes in the reservoir and $inSize$ is the number of input nodes. One has to randomly generate the values in the matrix over a uniform distribution $[-x, x]$ where x is a positive floating number. The value of x is not important as input scaling will be applied in the following step.

After generating this matrix, it has to be scaled by multiplying separately the bias columns and the input columns of the matrix by their input scaling global parameter (there is a scaling parameter for the bias and a separate one for the input).

The following step was to generate the reservoir. This was done by starting with a matrix of size $reservoirSize \times reservoirSize$ filled with values of 0, and then for each node randomly pick 10 random nodes to connect with, and update the connections in the matrix with random numbers generated in the same manner as the input matrix. After having a generated reservoir, the next step is to compute it's spectral radius, divide all the values in the reservoir matrix by this value, and then multiply all the values by the desired spectral radius which is the global parameter mentioned earlier.

When the weights are fixed, one has to pick an update function, I will use the same update function as the one presented in Section 2.1, I will copy it here for convenience:

$$\tilde{\boldsymbol{x}}(n) = \tanh\left(\boldsymbol{W}^{in}[1; \boldsymbol{u}(n)] + \boldsymbol{W}\boldsymbol{x}(n-1)\right), \tag{3}$$

$$\boldsymbol{x}(n) = (1-\alpha)\boldsymbol{x}(n-1) + \alpha\tilde{\boldsymbol{x}}(n), \tag{4}$$

When it comes to training, the only steps left at this point are generating X from Equation (6) by running the input through the matrix, and then retrieve the corresponding Y^{target} (from the same equation) from the training data, and use these two matrices to perform the ridge regression described in Equation (9) in order to compute the output weights matrix W^{out}.

Moving on to testing, one has to run the testing data through the network and compute an error metric based on the output of the network and the target output, and then proceed to improve this metric as well as possible by optimizing the model and the global parameters. The way this error metric is computed will be explained in the following section.

3.3 Error Metric

In order to optimize the network and its parameters, one has to first compute an error metric so that it is possible to compare the results of different parameters.

Traditionally, the error metric used for ESNs is a mean square error (MSE), for instance the normalized root mean square error (NRMSE) that was mentioned earlier which is computed as follows [11]:

$$E(\mathbf{y}, \mathbf{y}^{target}) = \frac{1}{N_y} \sum_{i=1}^{N_y} \sqrt{\frac{1}{T} \sum_{n=1}^{T} (y_i(n) - y_i^{target}(n))^2}, \tag{10}$$

where T is the number of data points, N_y is the size of each data point, **y** is the network output, and $\mathbf{y}^{target}$ represents the target. This is a way to accurately compute the difference between the output and the target. The problem that arises with MSE in a task such as music or percussion generation is that being a creative task, the target is not the only result that would fit. I dealt with this by treating the output of the network as the probability of a note being played ($\hat{y}$) and computing the error metric in the following way:

$$E(\mathbf{y}, \mathbf{y}^{target}) = \frac{1}{N_y} \sum_{i=1}^{N_y} \sum_{n=1}^{T} \log(\hat{P}(y_i^{target}(n))), \tag{11}$$

where

$$\hat{P}(y^{target}(n)) = \begin{cases} \hat{y} & y^{target}(n) = 1, \\ 1 - \hat{y} & y^{target}(n) = 0, \end{cases} \tag{12}$$

and N_y and T are the same as in the NRMSE. We sum the logarithms instead of the actual values because the values can become quite small sometimes and that might lead to underflow. One also has to take into account that the value inside the logarithm can not be smaller than or equal to 0, so he has to set a minimum value returned by $\hat{P}$ and never return anything smaller than that.

The value of the error metric will always be negative and one has to minimize the absolute value of this metric when the network is being optimized.

3.4 Optimization

When the network is trained and an error metric is in place, one has to start optimizing the network.

I created a script that would train and test the network using different values for each global parameter (leaking rate, bias scaling, input scaling, and spectral radius) and upon each run would improve one parameter. The way the values were picked for each parameter was by starting from an initial number and testing every possible following digit, upon finding the following digit with the best result, one would go on to add the digit to the starting number and repeat the process until the desired digit precision is achieved.

The best parameters I found were the following:

leaking_rate	0.699
bias_input_scaling	3.587
input_scaling	8.333
spetral_radius	3.00

3.5 Post processing the output

Now that everything is in place we can generate output by feeding a bass line into the network. A sample of generated output can be observed in Figure 5.

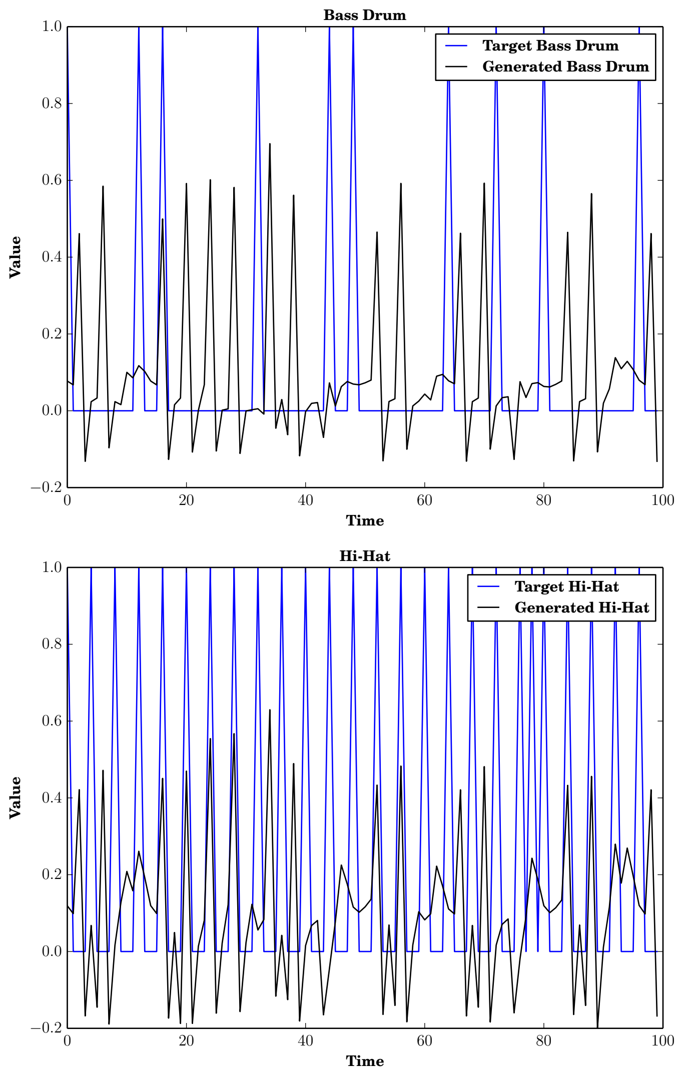

Figure 5: Bass Drum and Hi-Hat Output

It is observable from these plots that the output takes many different values. In order to convert this back to MIDI, one has to first decide when a value will be marked as a hit note and when it will not. For this experiment the chosen way of solving this is putting a separate threshold on the output of every drum depending on the frequency of occurrence that I want that drum to have in the song.

3.6 Output Feedback

The results generated by the network at this step already prove the potential Echo State Networks have in drum accompaniment, but I felt it could be improved by adding output feedback to the network. I will be explaining the reasoning behind this in Section 4.

Output feedback is when one takes the previously generated output anduses it to generate the following output. This can be done either by replacing the update function with:

$$\tilde{\boldsymbol{x}}(n) = \tanh\left(\boldsymbol{W}^{in}[1; \boldsymbol{u}(n)] + \boldsymbol{W}\boldsymbol{x}(n-1) + \boldsymbol{W}^{fb}y(n-1)\right), \tag{3}$$

where W^{fb} is the output feedback weight matrix, or by increasing the number of input nodes by the number of output nodes and then running the output through the input weight matrix. The two approaches are equivalent so one may choose based on personal preference. I chose to use the latter one.

Output feedback, even though having certain advantages, has some drawbacks also. The clear drawback comes as instability, as the network might lead to erroneous results, especially when there are more output nodes than input nodes, as their weight is bigger than the weight of the input, but there are a few methods that can be applied in order to counter this.

The first method put in place was teacher forcing. This is done by feeding the desired output $\mathbf{y}^{target}(n-1)$ into the feedback connection $(\boldsymbol{W}^{in})$ instead of the actual output $\mathbf{y}(n-1)$. This method aids the training and increases the accuracy when the network is run in testing or generative mode [11].

The second method I applied was creating a separate scaling parameter for feedback connection. This parameter is optimized in the same way presented in Section 3.4.

Upon optimizing all the parameters again, the resulting parameter values were:

leaking_rate	0.699
bias_input_scaling	3.587
input_scaling	8.333
output_scaling	0.655
spetral$_{r}adius$	3.00

4 Results

In the following section I will be presenting a few result samples and explain what they mean and how they helped improve the project. All of these results can be found at `https://github.com/sasualx/BSc-Thesis` , both as midi and as audio files.

At first, in order to check that the execution of this project is actually possible, I chose to have the output represented just by one value for each time step. A value of one if there is any drum played or a value of zero otherwise. A sample of this initial output can be observed in Figure 6.

After experimenting with a few thresholds for this output, I found that the drum beat generated, even though having only one drum, sounded precise and exact and could be used by a musician as an enhanced metronome. This initial experiment proved that the method proposed in this project is feasible, so I went on to represent the output as presented in section 3.1.

Examples of the output can be seen in Figure 7, some of which were also presented in Section 3.5.

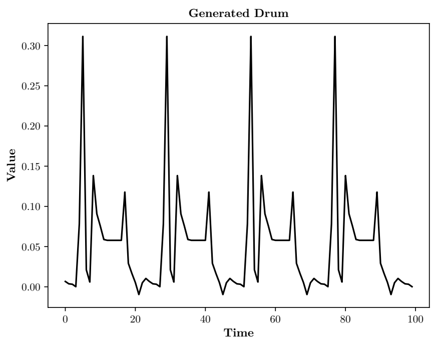

Figure 6: Initial Output

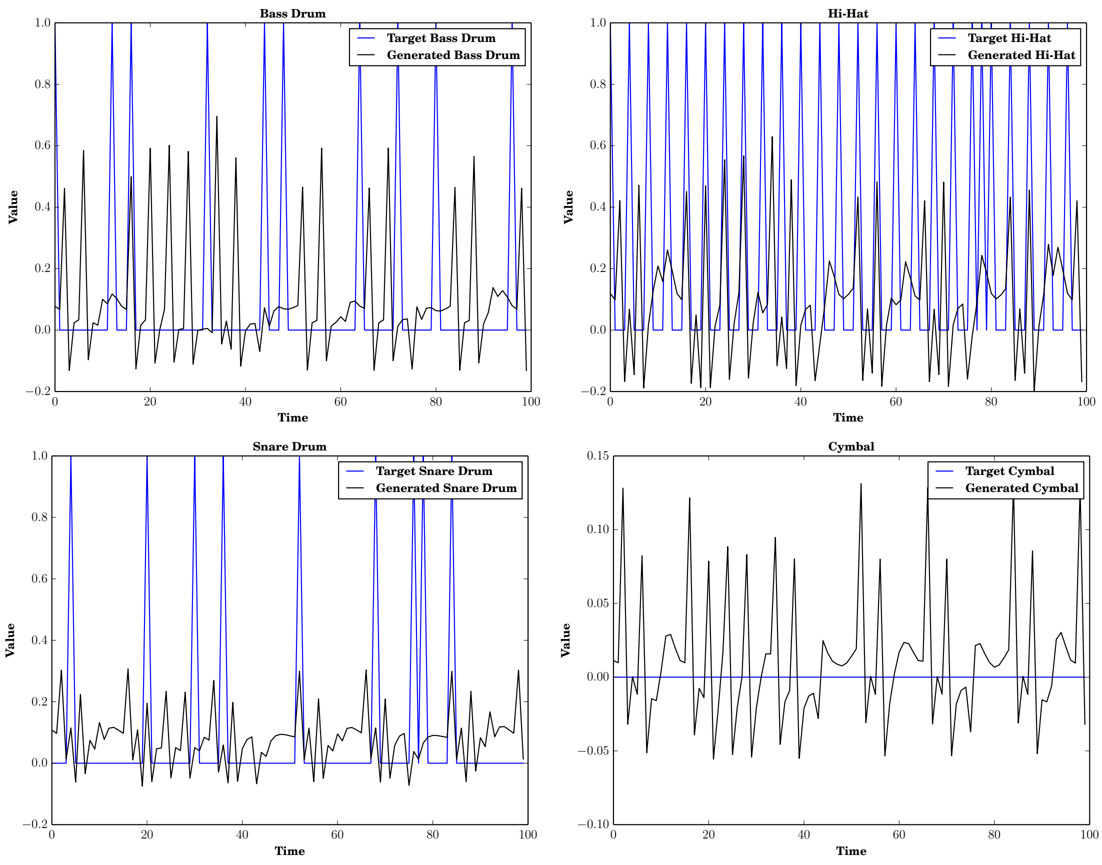

Figure 7: Bass Drum, Hi-Hat, Snare and Cymbal Output

Using multiple drums as compared to just one constitutes a big improvement, but setting the thresholds for these proved to be also more challenging, as every drum has a different range and a different frequency of occurance. As previously stated, at this point one has to decide (based on personal preference), what threshold he sets for each drum depending on how he wants the drum beat to sound.

I had multiple people listen to the drum beats generated with this output, they said that it sounds good and precise. Besides these positive notes on the drum beat, a lack of creativity and an inclination to be repetitive were noticed by multiple people, including myself. This is the reason I decided to add output feedback in an attempt to make the generated drum beat sound more creative and less repetitive.

In Figure 8 I will present a few samples of the output after adding output feedback, side by side with the output before adding it.

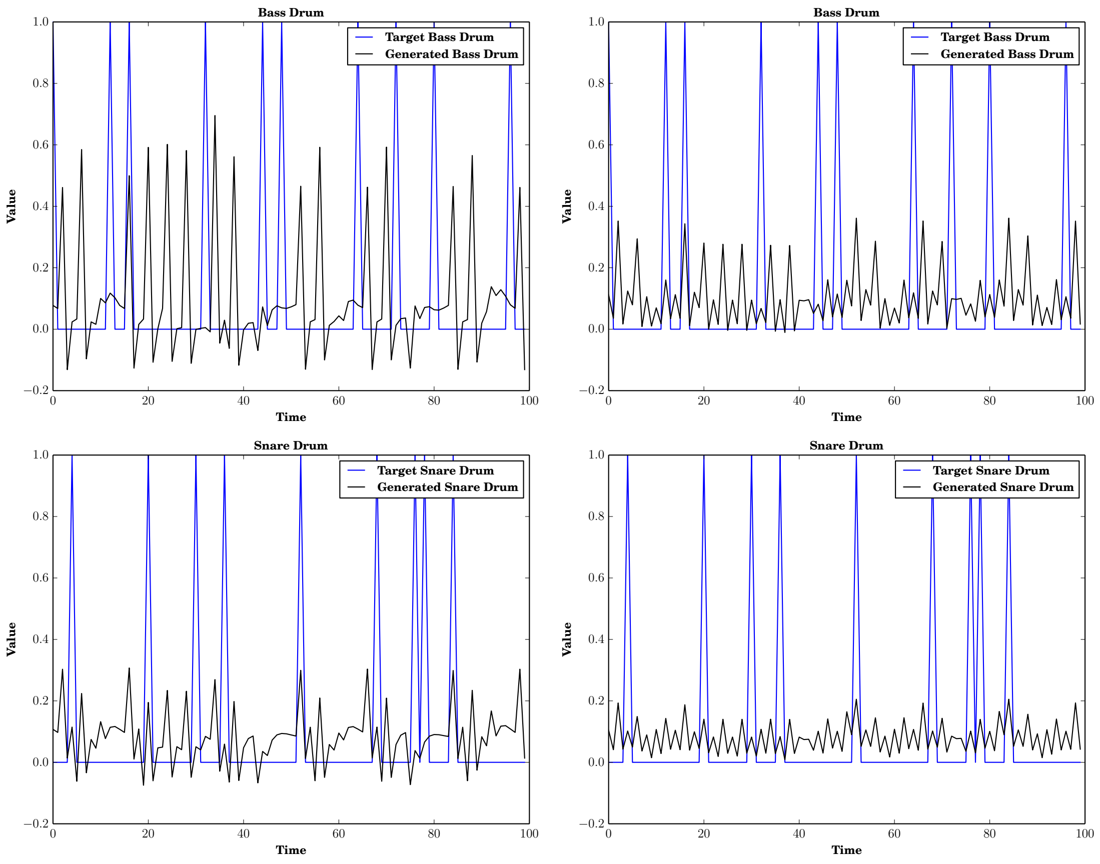

Figure 8: The plots of the Bass Drum (top) and the Snare (bottom) before adding output feedback are shown in the left panels, the corresponding plots after adding output feedback are shown in the right panels

As one could observe, the values in the output feedback plots are smaller and more irregular than the ones in the ones with no output feedback. Again one has to choose some thresholds for these drum values in order to have a generated drum beat. After choosing the thresholds and having a generated drum beat, I again presented the results to the people I presented the other ones to in order for them to compare the two. They said that it is certainly an improvement in the creativity of the beat, but at the cost of the beat not being as precise as before. Overall they preferred the results generated with output feedback.

5 Conclusions

The time given did not allow for further investigations, but the results above prove the potential Echo State Networks have in percussion generation, and this thesis could be a good point of reference for anyone trying to further experiment with Echo State Networks music generation.

The research questions addressed in this guided research project have been answered throughout this thesis, but I will try to make a short analysis of them here.

ESNs proved to perform well in the given task. The best performance of the network was given using output feedback with teacher forcing and using the following parameters:

leaking_rate	0.699
bias_input_scaling	3.587
input_scaling	8.333
output_scaling	0.655
spetral_radius	3.00

The input data was extracted from a pack of GuitarPro files and arranged in the way presented in Section 3.1. This type of packs are widely available so obtaining data is not a challenge when looking to solve this task.

When it comes to the last question ”How to convert the network output to a drum beat?”, one has to separately set thresholds for every drum and then experiment with these thresholds in order to create a drum beat that is according to his preferences.

5.1 Further investigation

If this research project would be continued, there are some improvements that could be added to it. For example:

- Increase the number of drums.
- Adapt the network in order to be able to accompany other instruments other than bass.
- Experiment with different values for the output feedback scaling in order to see until which point the instability created could be used towards better creativity while still generating a coherent drum beat.

6 Acknowledgements

I would like to mention that I am sincerely grateful for the guidance offered by the supervisor of this guided research, Prof. Dr. Herbert Jaeger, who continuously helped me along the way and without whom this guided research project could not have taken form.

References

[1] Adam Alpern. Techniques for Algorithmic Composition of Music. 1995.

[2] Michael C. Mozer. Neural Network Music Composition by Prediction: Exploring the Benefits of Psychoacoustic Constraints and Multi-scale Processing. *Connection Science*, 6(2-3):247–280, 1994.

[3] Frederick P. Brooks, Jr., Albert L. Hopkins, Jr., Peter G. Neumann, and William V. Wright. Chapter "An Experiment in Musical Composition" from Machine Models of Music, pages 23–40. MIT Press, Cambridge, MA, USA, 1992.

[4] Ella Gale, Oliver Matthews, Ben de Lacy Costello, and Andrew Adamatzky. Beyond Markov Chains, Towards Adaptive Memristor Network-based Music Generation. *CoRR*, abs/1302.0785, 2013.

[5] Marco Marchini and Hendrik Purwins. Unsupervised Generation of Percussion Sound Sequences from a Sound Example, pages 205–218. 01 2010.

[6] Jon Gillick, Adam Roberts, Jesse Engel, Douglas Eck, and David Bamman. Learning to Groove with Inverse Sequence Transformations, 2019.

[7] Thomas Pllaha. Echo State Networks: Music Accompaniment by Prediction, 05 2014.

[8] Axel Tidemann, Pinar Ozturk, and Yiannis Demiris. A Groovy Virtual Drumming Agent, 09 2009.

[9] Herbert Jaeger and Mantas Lukoševičius. Reservoir Computing Approaches to Recurrent Neural Network Training, 2009.

[10] Herbert Jaeger. Echo State Network. *Scholarpedia*, 2(9):2330, 2007. Revision #188245.

[11] Mantas Lukoševičius. A Practical Guide to Applying Echo State Networks, 2002.

[12] PyGuitarPro. https://pyguitarpro.readthedocs.io/en/stable/. [Online; accessed 16-April-2019].

[13] GuitarPro-to-Midi. https://github.com/alexsteb/GuitarPro-to-Midi. [Online; accessed 20-April-2019].

[14] Mono. https://www.mono-project.com/docs/about-mono/supported-platforms/macos/. [Online; accessed 20-April-2019].

[15] John Walker. MidiCSV. http://www.fourmilab.ch/webtools/midicsv/, 2004. [Online; accessed 04-April-2019].