NTNU
Norwegian University of
Science and Technology

Evolving artificial neural networks for cross-adaptive audio effects

Iver Jordal

Master of Science in Computer Science
Submission date: January 2017
Supervisor: Gunnar Tufte, IDI
Co-supervisor: Øyvind Brandtsegg, IM

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Cross-adaptive audio effects have many applications within music technology, including for automatic mixing and live music. Commonly used methods of signal analysis capture the acoustical and mathematical features of the signal well, but struggle to capture the musical meaning. Together with the vast number of possible signal interactions, this makes manual exploration of signal interactions difficult and tedious. This project investigates Artificial Intelligence (AI) methods for finding useful signal interactions in cross-adaptive audio effects. A system for doing signal interaction experiments and evaluating their results has been implemented. Since the system produces lots of output data in various forms, a significant part of the project has been about developing an interactive visualization tool which makes it possible to evaluate results and understand what the system is doing. The overall goal of the system is to make one sound similar to another by applying audio effects. The parameters of the audio effects are controlled dynamically by the features of the other sound. The features are mapped to parameters by using evolved neural networks. NeuroEvolution of Augmenting Topologies (NEAT) is used for evolving neural networks that have the desired behavior. Experiments show that this approach is successful.

Contents

Chapter 1

Introduction

For decades, music technology has made music more appealing by applying audio effects, which are processing techniques that alter audio so that it sounds different. For example, one common audio effect in rock music is distortion on the electric guitars, which will make them sound "fuzzy" instead of "clean". Some audio effects become more appealing when their parameters are changed over time. For example, there is the auto-wah effect, which essentially is a peaking filter, which amplifies a specific frequency and cuts off other frequencies. The volume of the input is used to dynamically control the cutoff frequency of the filter. When the cutoff frequency is swept from low to high, it sounds like "wah", hence the name auto-wah. The auto-wah effect is an example of an adaptive audio effect (Verfaille et al., 2006). Since meaningful interactions between musical elements can make music more interesting and appealing, cross-adaptive audio effects were invented. In this class of audio effects, parameters are dynamically informed by features of other sounds. In the 1960s, Stockhausen presented one of the first cross-adaptive audio effects in his composition "Hymnen" (Moritz, 2003), where he, amongst other things, modulated the rhythm of one anthem with the harmony of another anthem. Sidechain compression is a more recent example of a cross-adaptive audio effect. In that technique, the amplitude of one sound controls the gain reduction parameter in the compressor that is applied to a different sound. In electronic music, sidechain compression is often used to let the volume of the bass drum turn down the volume of the bass synth. This is done to avoid conflicts between the bass drum and the bass synth, and also provides a pulsating, rhythmic dynamic to the sound (Colletti, 2013). Further, cross-adaptive audio

effects have been used in algorithms for mixing multichannel audio (Reiss, 2011) and voice-controlled synthesizers (Cartwright and Pardo, 2014). Generally, cross-adaptive audio effects can be applied in a wide range of research fields, including live music performance and audio mastering. Current research at the Music Technology department at Norwegian University of Science and Technology aims at exploring radically new modes of musical interaction in live music performance. In 2015, Øyvind Brandtsegg presented a toolkit for experimenting with signal interaction. This toolkit enables one to find musically interesting signal interactions by empirical experimentation. However, this can be tedious due to the vast number of combinations. Also, while most low-level audio features are mathematically and acoustically well defined, it's hard to use them for musically interesting cross-adaptive audio effects. One often needs to combine several audio features in complex ways. An audio feature can be linked to any effect parameter, and the mapping function can be anything. A setup can have many instruments, lots of audio effects, and the ordering of the effects may vary. Indeed, Brandtsegg's suggestions for future work includes "practical and musical exploration of the technique, and the mapping between sound features and effects controls". As cross-adaptive audio effects are relatively uncharted territory, methods to evaluate various cross-couplings of features have not been formalized. There is a need for a tool that can help efficiently search for useful mappings in those huge search spaces. That was the spark of this project.

In practice, a music performance that uses cross-adaptive audio effects can be a very complex, dynamic system with many signal interactions. However, to limit the scope and complexity of the problem, this project will study signal interactions between only two sounds at a time: an input sound and a target sound. A single audio effect is applied to the input sound. The parameters of this effect are dynamically informed by features from the target audio. The goal of the tool implemented in this project is to make interesting mappings from target audio analysis to audio effect parameters. Brandtsegg (2015) suggested that machine learning can be useful in this context. Generally, for a machine learning problem to be well-defined, a performance measure is needed. A good performance measure would be "how appealing does it sound?". However, what generally sounds appealing to humans is tacit knowledge and cannot be simply described mathematically (Schmidhuber, 2009). Because there are no good objective measures of what a good cross-adaptive audio effect is,

an assumption has been made in this project: If a cross-adaptive audio effect makes features of one sound audible in the other sound, then it is considered interesting. Therefore it has been decided that the objective of the system in this project should be to make the input sound similar to the target sound.

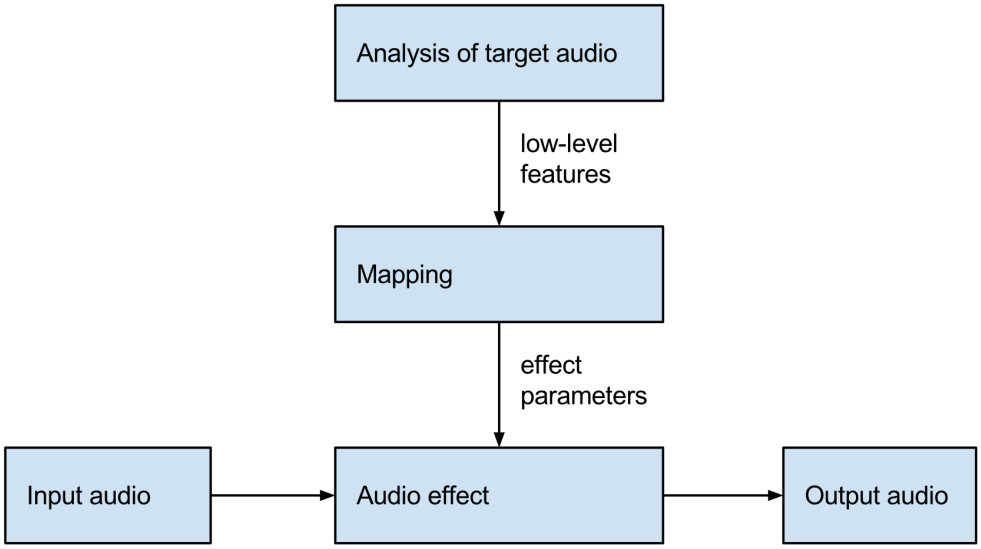

Figure 1.1: Cross-adaptive audio effect signal flow with two audio streams: input audio and target audio

Figure 1.1 roughly illustrates the signal flow in a cross-adaptive audio effect. The input sound is fed into an audio effect, and out comes the output audio. The parameters of the audio effect are controlled dynamically by the output of the mapping function, which is a function that maps n-dimensional vectors (audio features) to m-dimensional vectors (effect parameters). The audio features are obtained by analyzing the target audio. Artificial neural networks are suitable as mapping function, as they can approximate a wide variety of continuous functions (Hornik, 1991). Backpropagation (Werbos, 1982; Lecun et al., 1998) is an efficient algorithm for training neural networks (i.e. finding a set of useful weights for it), but it requires target values for the output nodes to compute error signals. Since no sets of target values were available for this project, and there is generally no good way of telling exactly how the mapping affects the resulting sound, it makes sense to use evolutionary computation to train the neural networks instead. That is feasible because it is possible to construct a fitness function that returns a score based on how similar two sounds are. All in all, this project is about developing a toolkit that explores evolutionary computation in

various ways to evolve artificial neural networks that act as mappings in cross-adaptive audio effects.

As the developed system has many components and deals with lots of numbers in the form of audio features, neural network weights, audio effect parameters, output sounds and fitness values, it's hard to understand what the system really does. To alleviate this, a significant part of the project has been about developing a comprehensive interactive visualization system as a part of the toolkit. One motivation for this is that one cannot improve the optimization process if one does not know where it fails. It was also made in order to make the toolkit more human-understandable and to make evaluation of results more efficient.

The thesis is structured as follows: Chapter 2 provides background information on genetic algorithms, neural networks, audio processing and the main areas of study in the specialization project that this master's thesis builds on. Chapter 3 describes how neural networks and neuroevolution are applied in the toolkit developed in this project. Chapter 4 contains descriptions of each experiment, the results obtained and discussion. Chapter 5 concludes the thesis and mentions further work.

Chapter 2

Background Information

2.1 Genetic Algorithms

Genetic algorithms (Goldberg, 1989; Bäck, 1996) are iterative algorithms that can approximate solutions to optimization problems. In such problems, one usually doesn't know how to construct a good solution, but it is possible to measure how good a solution is. The methods used in genetic algorithms are inspired by Darwin's principle of natural selection. In the algorithm, a population of individuals is simulated through generations of "life". Each individual is a candidate solution to the optimization problem. The fittest individuals, as determined by a fitness function, are the individuals that are most likely to survive and reproduce (either asexually or sexually). Individuals that are deemed less fit are more likely to die young, and do not get to pass their genes on to future generations. During reproduction, crossover and mutation occurs. Crossover is a genetic operator that combines two parents to produce an offspring. Mutation is a genetic operator that alters an individual slightly. The whole process is roughly illustrated by figure 2.1.

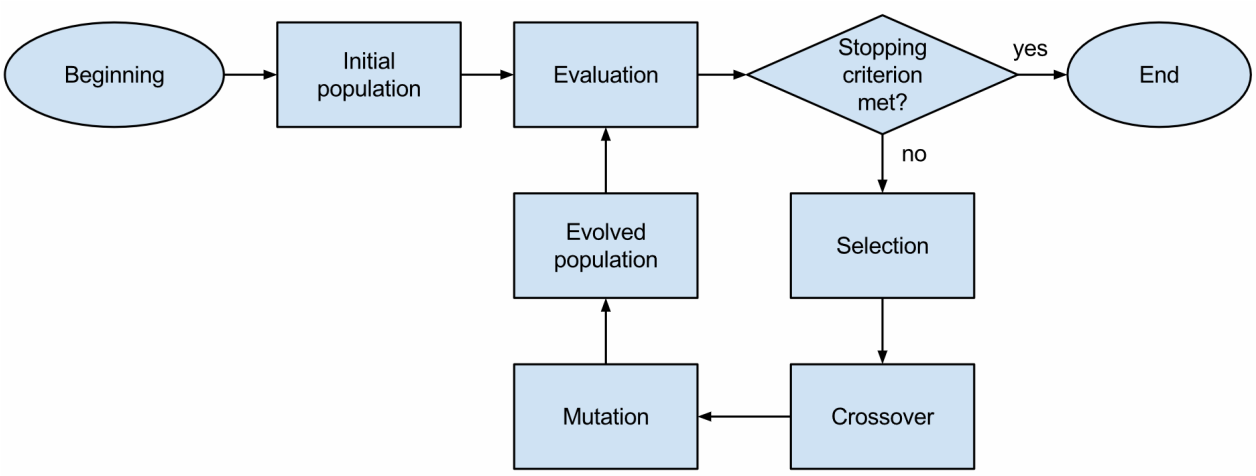

Figure 2.1: Genetic algorithm cycle

2.2 Artificial Neural Networks

Artificial Neural Networks (ANN) are systems of interconnected "neurons", or nodes (Caudill, 1987). A connection from a node A to a different node B means that the activation level of node A influences the activation level of node B based on the numerical weight of the connection. The activation level of a node is calculated by adding up all incoming signals to that node and running that number through the node's activation function. An ANN can be thought of as a function that transforms n-dimensional vectors to m-dimensional vectors. Figure 2.2 illustrates a simple neural network and the inner workings of one of the nodes.

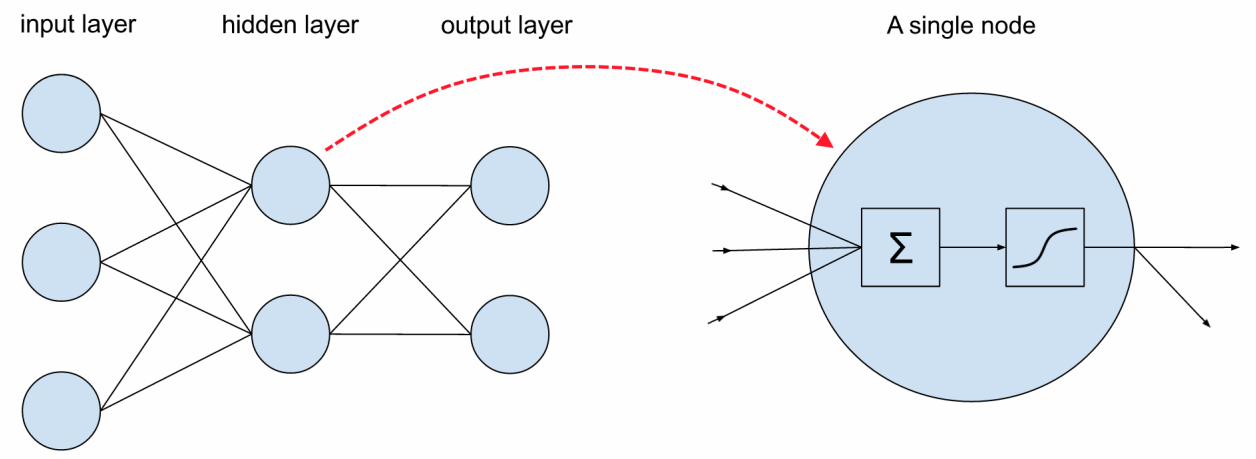

Figure 2.2: To the left: Illustration of a small neural network with one hidden layer. To the right: Illustration of a hidden node with sigmoid activation function

2.3 Neuroevolution

Neuroevolution is a technique that uses evolutionary algorithms to train artificial neural networks. It differs from supervised learning algorithms such as backpropagation in that it does not require a set of correct input-output pairs. Instead, only a performance measure (fitness function) is needed.

2.4 NeuroEvolution of Augmenting Topologies (NEAT)

NEAT (Stanley and Miikkulainen, 2002) is a neuroevolution technique that evolves neural networks with genetic algorithms. Not only the weights of the ANN are evolved, but also the structure. The NEAT approach begins with a feed-forward approach with input nodes and output nodes that are fully connected. The ANN can then grow larger by having nodes and links added to it. NEAT can also remove nodes and links.

2.5 Multi-Objective Evolutionary Algorithms

As real-life problems often have more than one objective, there is a need for ways to deal with multiple objectives effectively. A multi-objective evolutionary algorithm (MOEA) is an algorithm for solving mathematical optimization problems involving more than one objective function to be optimized simultaneously (Veldhuizen and Lamont, 2000). One well-known algorithm of this kind is Nondominated Sorting Genetic Algorithm II (NSGA-II) (Deb et al., 2002). In this algorithm, the performance measure is based primarily on rank and secondarily on crowding distance. Rank is calculated by running the fast non-dominated sort algorithm. This algorithms assigns a rank to each individual. If the rank of individual A is better than the rank of another individual B, it means that A dominates B. Individual A dominates B if both the following conditions are true:

- The solution A is no worse than B in all objectives.
- The solution A is strictly better than B in at least one objective.

Crowding distance is a way to measure how crowded the search space around the individual is. Crowding distance is quantified by forming a cuboid with the nearest neighbours as vertices, and then taking the average of the side lengths of the cuboid. Large crowding distances are encouraged because it preserves diversity in the population (Deb et al., 2002).

2.6 Audio Feature Extraction Tools

Audio feature extraction is the process of computing a compact numerical representation that can be used to characterize a segment of audio. Low-level features such as spectral centroid and Mel-Frequency Cepstral Coefficients (MFCC) (Mermelstein, 1976; Logan, 2000) are computed directly from the audio signal, frame by frame. A frame is a slice of audio and can consist of for example 1024 samples. In an audio signal with sampling rate 44.1 kHz, the duration of such a frame would be approximately 23 ms.

Audio features can be used in many different ways, such as music information retrieval and musical genre classification. In this project, they are used for similarity measures and for controlling the parameters of audio effects. Four audio feature extraction tools were used in this project: Aubio (Brossier, 2003), Essentia (Bogdanov et al., 2013), LibXtract (Bullock, 2007) and Csound (Fitch et al.).

2.7 Audio Effects

Audio effects are processing techniques that alter audio so it sounds different. The following subsections describe the audio effects used in this project.

2.7.1 Modified Hyperbolic Tangent

Modified hyperbolic tangent is a waveshaping function that can model the characteristics of analog distortion, and especially tube distortion (GDSP Online Course, 2014c). Modified hyperbolic tangent differs from hyperbolic tangent in that one can model the positive and the negative slopes differently. This distortion effect makes the sound “fuzzier” by adding harmonic components. Figure 2.3 illustrates an example of this.

$$\text{mtanh}(x) = \frac{e^{ax} - e^{-bx}}{e^{cx} + e^{-dx}}$$

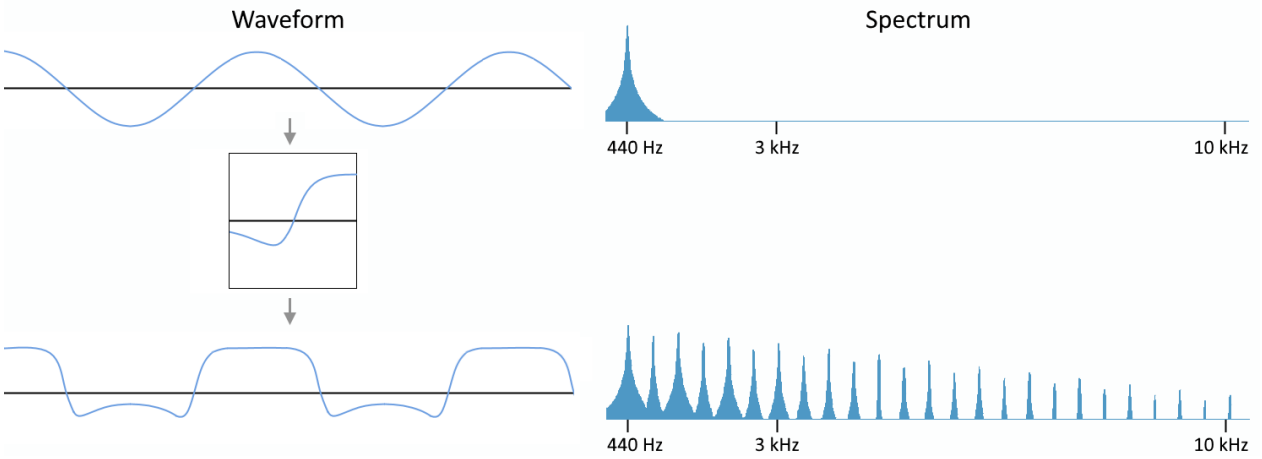

Figure 2.3: Harmonic frequency components are added to a 440 Hz sine wave by applying a modified hyperbolic tangent function

2.7.2 Low-Pass Filter

A low-pass filter attenuates high frequencies and retains low frequencies unchanged (Dodge and Jerse, 1997). It can be used to make a sound "darker" or "smoother" in timbre. A resonant low-pass filter is a low-pass filter that has a peak in the response curve at the cutoff frequency, as illustrated by figure 2.4. This quality can be used to boost a single tone in a sound with a rich frequency spectrum. The width of the resonant peak is described by a parameter called Q. As Q increases, the resonance becomes more pronounced.

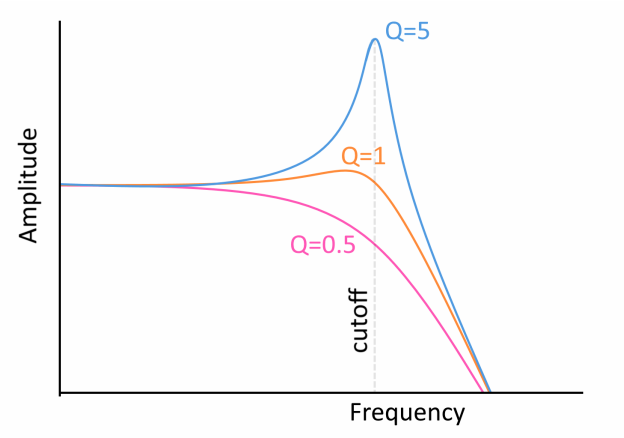

Figure 2.4: Frequency response of a resonant low-pass filter with various Q values

2.7.3 Band-Pass Filter

A band-pass filter rejects frequencies outside a given range (Dodge and Jerse, 1997). This filter has two parameters. Center frequency (denoted by f_0 in figure 2.5) defines the center of the frequency range. Bandwidth (denoted by B in figure 2.5) defines how broad or narrow the frequency range is.

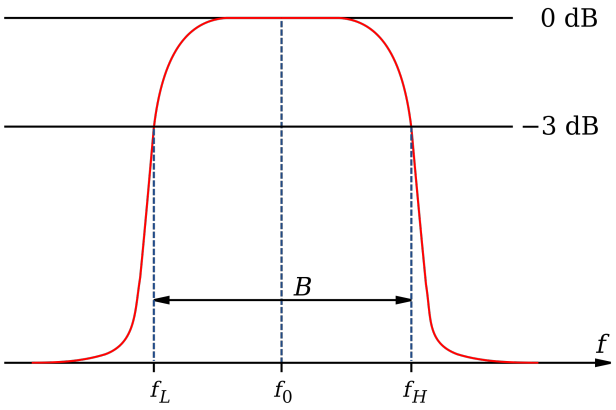

Figure 2.5: A diagram that illustrates the frequency response of a band-pass filter and defines its bandwidth

2.7.4 Amplitude Modulation

The amplitude modulation effect multiplies the sound signal with a unipolar sine wave that oscillates between 0 and 1. When the frequency of the sine wave is low (< 20 Hz), one can

hear that the amplitude is being brought up and down, as in tremolo (Serafin, 2007). With higher frequencies there is instead an effect on the timbre of the sound. The modulator generates a set of frequency sidebands, as illustrated in figure 2.6

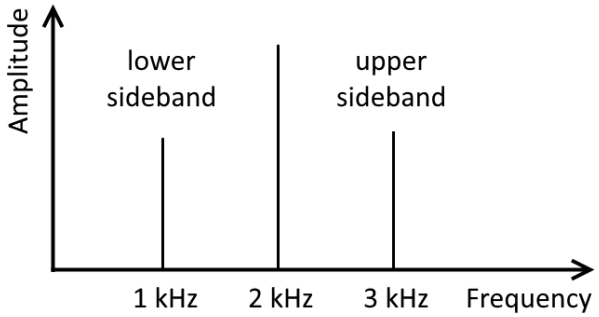

Figure 2.6: Frequency spectrum plot of a 2 kHz sine wave modulated with a 1 kHz unipolar sine wave

2.7.5 Bit Reduction

The bit reduction effect reduces the number of bits used to represent a sample (GDSP Online Course, 2014a). Each sample amplitude value is rounded to a number of discrete steps. This introduces a particular kind of distortion, also called quantization noise.

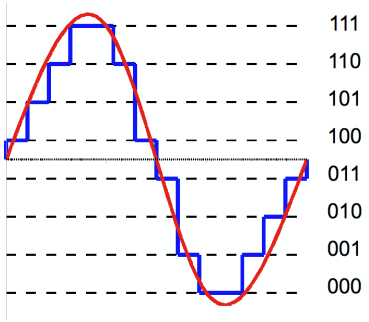

Figure 2.7: A high fidelity sine wave quantized to 3 bits per sample. *By Hyacinth, CC BY-SA 3.0,* `https://commons.wikimedia.org/w/index.php?curid=30716342`

2.7.6 Chorus

The chorus effect delays the input signal by a short *delay time* (usually in the range 20-50 ms) and mixes it with the dry input signal (GDSP Online Course, 2014b). The *delay time* is variable and usually controlled by a Low-Frequency Oscillator (LFO), as illustrated

by figure 2.8. This may create an impression of multiple voices playing or singing the same thing.

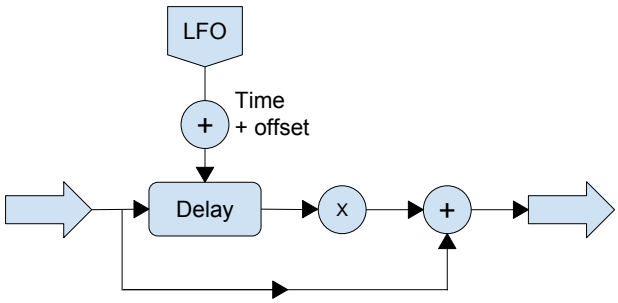

Figure 2.8: Signal flow in a chorus effect

2.8 Audio Processing Tools

Audio processing is the alteration of audio signals, typically through audio effects. One popular audio processing tool is Csound (Fitch et al.). This is both an audio programming language and a program that runs Csound code. The Csound program takes in a text file of code. This code is executed by the Csound program. The output is sound that is directed to either an audio interface (live) or to a file (non-real-time processing). Csound is used by musicians and composers, typically in experimental electroacoustic music. Traditionally, it has been an offline tool, due to lack of computational power. Today, computational power is sufficient for Csound to run in real-time, so it can be used in live settings such as concerts and sound installations. Csound can not only run on desktop computers, but can also be used as audio processing engine in mobile applications for the operating systems Android and iOS.

2.9 Specialization Project

This master's thesis is a continuation of the author's specialization project. The preliminary experiments in the specialization project were found to be successful. This led to a published paper in the proceedings of the 2nd AES Workshop on Intelligent Music Production[1]. This

[1] `http://www.aes-uk.org/forthcoming-meetings/wimp2/#proceedings`

paper is included in appendix D. The author also presented his findings in a 15-minute talk at this event.

The following subsections will summarize some of the main areas of study in the experiments in the specialization project.

2.9.1 Output Activation Functions

Three different output activation functions (used in the output nodes of the neural networks) were compared. Sigmoid was found to be better than linear and sine in most cases (see figure 2.9).

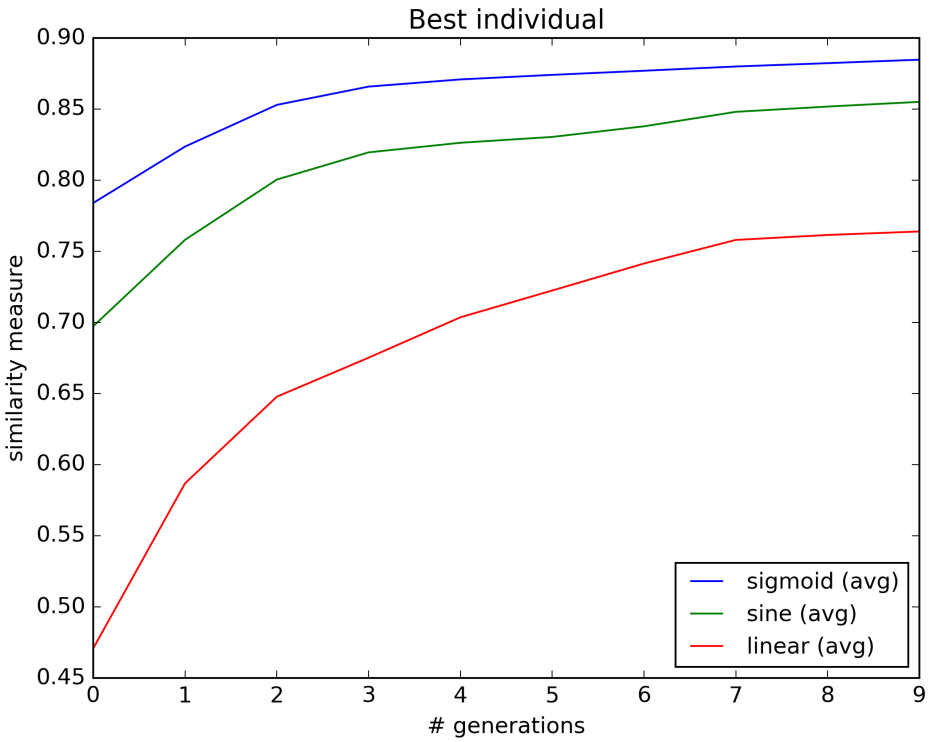

Figure 2.9: Average fitness over 20 runs

2.9.2 Fitness Functions

Five fitness functions were compared. Figure 2.10 shows how well they scored on average on the similarity measure (local similarity).

Local Similarity

The local similarity fitness function is based on the average euclidean distance between the feature vector of the target sound and the output sound in the k frames of the two sounds.

```
Function LOCAL_SIMILARITY(target, individual):
    total_euclidean_distance = 0
    for each k in range(num_frames):
        A = target.get_feature_vector(k)
        C = individual.get_feature_vector(k)
        total_euclidean_distance += EUCLIDEAN_DISTANCE(A, C)
    avg_euclidean_distance = total_euclidean_distance / num_frames
    return 1 / (1 + avg_euclidean distance)
```

where `EUCLIDEAN_DISTANCE` is $d(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2) + ... + (q_n - p_n)^2}$

Multi-Objective Optimization

This fitness function is inspired by NSGA-II (Deb et al., 2002). It incorporates two measures: rank and crowding distance. These are concepts taken directly from the NSGA-II paper, and they are then used in a math expression that satisfies these two constraints that are used in NSGA-II:

- $\text{rank}(A) > \text{rank}(B) \implies \text{fitness}(A) > \text{fitness}(B)$
- $\text{rank}(A) = \text{rank}(B), \text{CD}(A) > \text{CD}(B) \implies \text{fitness}(A) > \text{fitness}(B)$
  where CD stands for crowding distance

The ranks of the individual are calculated by doing non-dominated sort. Crowding distance is computed between individuals in a given rank. The multi objective fitness function works like this:

```
Function MULTI_OBJECTIVE(target, individuals):
    for individual in individuals:
        CALCULATE_OBJECTIVES(individual, target)
```

```
    fronts = FAST_NON_DOMINATED_SORT(individuals)
    for rank in fronts:
        CALCULATE_CROWDING_DISTANCE(fronts[rank])  # assigns individual.crowding_distance
        for individual in fronts[rank]:
            individual.fitness = 1.0 / (rank + (0.5 / (1.0 + individual.crowding_distance)))


Function CALCULATE_OBJECTIVES(individual, target):
    individual.objectives = {}
    for feature in similarity_features:
        individual.objectives[feature] = EUCLIDEAN_DISTANCE(
                                            target.analysis[feature],
                                            output.analysis[feature])
```

Pseudocode for `FAST_NON_DOMINATED_SORT` and `CALCULATE_CROWDING_DISTANCE` can be found in the NSGA-II paper (Deb et al., 2002).

Hybrid

While NSGA-II is good at optimizing for non-dominated individuals, these individuals may be extreme tradeoffs and therefore not necessarily feasible solutions in practice. In order to reward good tradeoffs more, the author developed the hybrid fitness function. This fitness function returns the average of `MULTI_OBJECTIVE` and `LOCAL_SIMILARITY`.

Novelty Search

Novelty search (Lehman and Stanley, 2008) ignores the objective and optimizes for novelty instead. The reason that this may work well is that in some problems the intermediate steps to the goal do not resemble the goal itself. When it comes to implementation, MultiNEAT has novelty search built-in, but Python bindings for it are missing, so the author could not use it in his Python application. However, novelty search can be implemented on top of most evolutionary algorithms, by using a fitness function that rewards novelty (Lehman and Stanley, 2015), so that is what the author did. First, each individual needs to be represented as a vector that describes its characteristics. This vector is constructed by

concatenating all audio feature series of the individual. The implemented fitness function assigns high fitness values to the individuals that have long euclidean distances from the 3 nearest neighbours, where the neighbours are individuals that have been evaluated earlier. The very first population gets random fitness values, because there are no earlier individuals to measure distance from.

Mixed

This fitness function is simple: For each generation, one of the following fitness functions is chosen randomly and applied: local similarity, multi-objective, hybrid, novelty. The idea behind this fitness function is to create a dynamic fitness landscape, where the individuals that get good scores from all fitness functions have the greatest chance of survival over time.

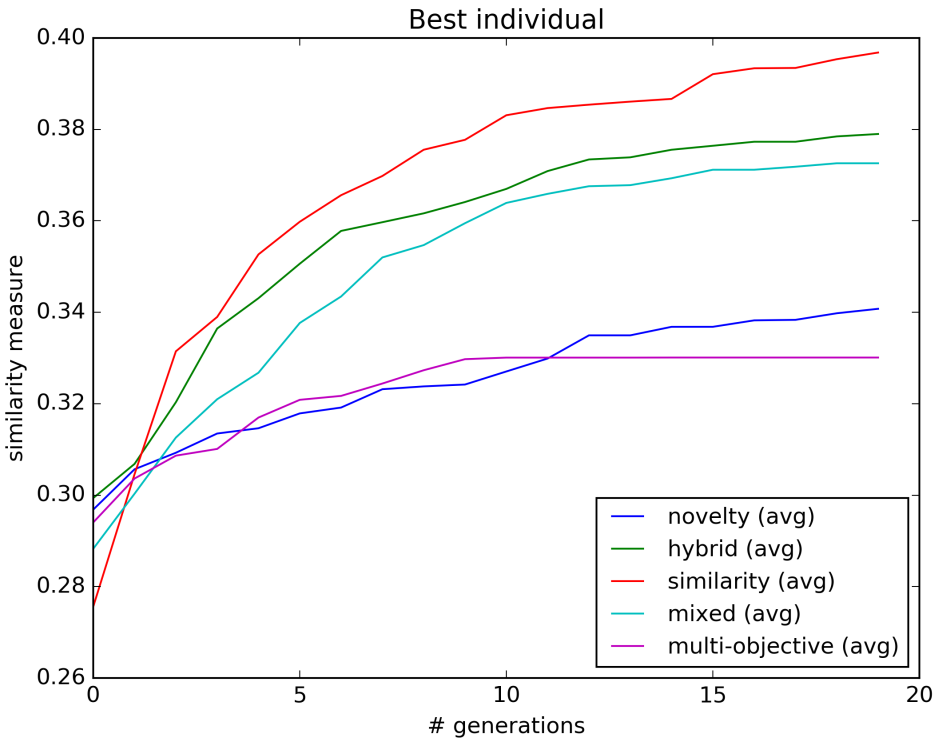

Figure 2.10: Best individual on average over 20 runs, with 20 generations for each run

2.9.3 Automatic Feature Selection

When dealing with high-dimensional input data, the number of weights in a fully connected neural network becomes quite large. Clean and useful combinations of all the input signals can be hard to evolve. To deal with this situation, one can use Feature Selective NeuroEvolution of Augmenting Topologies (FS-NEAT) (Whiteson et al., 2005). This NEAT variation starts with just a few connections and gradually adds/removes connections.

In an experiment with noise for input sound and a sine sweep for output sound, FS-NEAT was found to perform better than classic NEAT. This experiment had 68 audio features as input. Classic NEAT would typically evolve a very noisy parameter control, so the output sounds became glitchy and not very musically interesting. FS-NEAT has been found to deal with the high-dimensional input more effectively, as it selected only a few of the audio features that were useful for getting high fitness values. Figure 2.11 shows fitness values with NEAT and FS-NEAT in the described experiment.

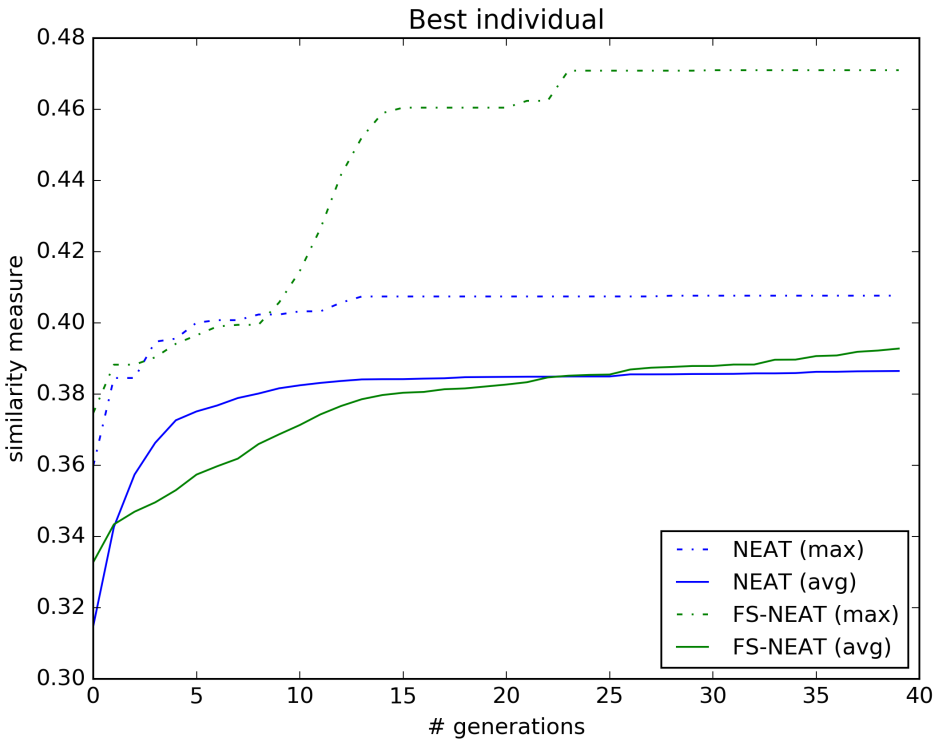

Figure 2.11: Fitness values in runs with NEAT and FS-NEAT, aggregated from 20 runs with each configuration.

Chapter 3

Methods and Implementation

3.1 Evolving Neural Networks

There are several ways to map sound analysis to effect parameters. The initial idea was a modulation matrix, where each effect parameter becomes a linear combination of the audio features. This idea was quickly iterated upon. Why not use an artificial neural network? An artificial neural network can do the same as a modulation matrix (when the ANN has linear activation functions and no hidden layers), but it can also do more. A neural network can have hidden nodes with various activation functions, which makes more complex signal interactions possible. This widens the scope, and allows for learning higher-level features, such as "the snare and the bass drum are hit simultaneously".

NEAT (Stanley and Miikkulainen, 2002) is suggested as a technique for evolving neural networks. This is a technique which evolves not only the weights of the neural network, but also the topology, i.e. the number of nodes and the connections between the nodes. HyperNEAT, a technique that is based on NEAT, has been used with great success in a project called Picbreeder (Secretan et al., 2008) and later with some success in a project called SoundBreeder (Ye and Chen, 2014). The purpose of those projects were to evolve visually and aurally appealing art, respectively. Since both those two projects and this project are within the field of creative computation, the hypothesis is that NEAT or HyperNEAT will work well in this project.

3.2 Implementation

Csound has been selected to be the audio processing tool in this project because it is actively used by students and lecturers in the Music Technology department at NTNU. Since Csound interoperates nicely with Python, and Python is a popular language for artificial intelligence (AI) applications, that was the language of choice for the neuroevolution application. Further, it has been decided that the application should be written in a style compatible with both Python 2 and 3, for the sake of compatibility with various Python libraries. Also, it should work on both Windows and Ubuntu. This way, experiments can not only run on Windows desktop computers, but also on Ubuntu instances in the cloud. The most important dependency is MultiNEAT (Chervenski and Ryan), which is one of Kenneth Stanley's recommended neuroevolution libraries (Stanley, 2015). It features several neuroevolution algorithms, including NEAT, FS-NEAT and HyperNEAT. It is written in C++, but it has Python bindings, so it is fit for this project. Further, the audio features extraction tools selected for this project are Aubio, Essentia and LibXtract, which are all free open source software written in a compiled language. A link to the author's implementation of the toolkit, which is open source, is included in appendix A. A complete list of dependencies is included in appendix B.

3.2.1 Performance

A. Eldhuset has previously implemented a program that uses Csound in signal interaction experiments with genetic algorithms. He concluded that his implementation was slow, taking around 5 seconds per individual (Eldhuset, 2015). Hence an experiment with a population size of 20 and 20 generations would take approximately half an hour. The author has analyzed the weaknesses of Eldhuset's approach and come up with a number of techniques to alleviate performance issues:

- Use a templating engine to generate Csound files. It writes csound code to different files, one for each individual in the population. This allows many Csound instances to be run in parallel.

- While a Csound instance runs, it does not have to communicate with another program (a host) via an API. All data needed for the run, including effect parameter values over time, are included inside the csd file
- Use dedicated, compiled audio feature extraction tools such as Aubio instead of Csound for audio feature extraction
- Use the standard streams (stdout) instead of file I/O in audio feature extractors that support this
- Let the host program, Csound and audio feature extractors write files to a RAM disk to avoid slow disk I/O activity
- Use the concept of pipelining to shorten critical paths and enable more parallelism
- Sensible handling of duplicate individuals: when two or more individuals are equal (i.e. their neural networks are equal), evaluate only one of them, and apply the same result to the identical individuals

Incorporating all these techniques, the author's implementation spends around 0.11 seconds on average per individual, given that the dist_lpf effect and the aubio mfcc analyzer is used, the duration of the input sound is 7 seconds, and that the program is run on a modern, high-end laptop with two CPU cores. Hence an experiment with population size 20 and 20 generations may take approximately half a minute. If the input sound is shorter, the experiment will run even faster. This is exploited in Experiment 1, where over a million individuals are processed, so fast performance is really useful.

3.2.2 Neuroevolution Routine

The neuroevolution program is called from the command line, with a number of arguments for configuring the experiment. The program then performs roughly these steps:

1. Check sanity of arguments
2. Analyze input sound file and target sound file

3. Initialize a population

4. For each generation, evaluate all individuals, write their data to json files and then advance to the next generation

The evaluation of an individual (illustrated by figure 3.1) involves several operations:

1. An artificial neural network is created from the genotype of the individual

2. All feature vectors of the target sound are run through the neural network

3. The neural outputs are scaled to appropriate ranges for the various audio effect parameters

4. Csound runs the input sound through the audio effect that is controlled by the audio effect parameters

5. The resulting sound is run through the audio feature extraction tool(s)

6. The audio features are standardized with the same mean and variance as in the standardization of the target sound audio features

7. The audio features of the target sound and the output sound are used in the fitness function

8. The resulting fitness value is assigned to the individual.

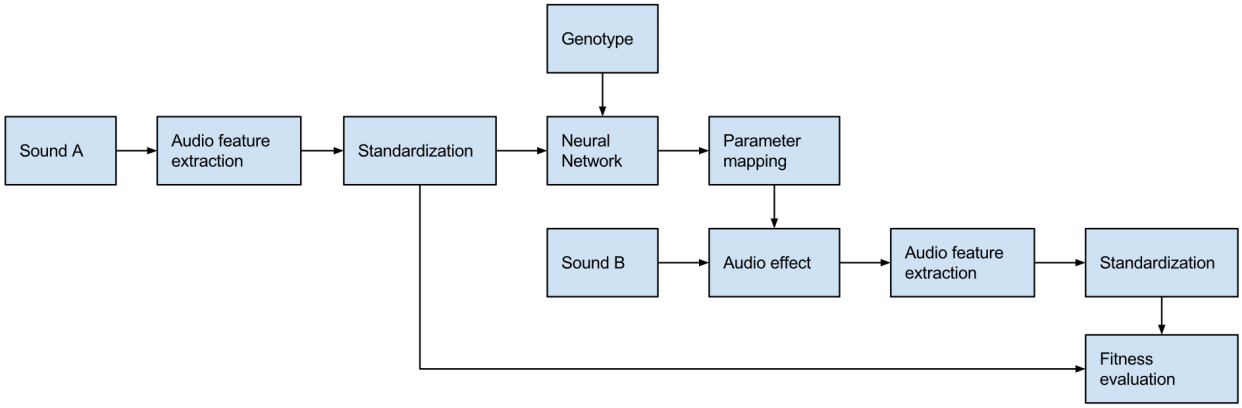

Figure 3.1: Flowchart for the evaluation of an individual

3.2.3 Input Standardization

To make the audio features suitable as input to a neural network, they need to be scaled. A simple, but good technique is to standardize them by is subtracting the mean and dividing by the standard deviation (Sarle, 2014). For each audio feature there is one sequence of numbers for the input sound and another sequence for the target sound. The mean and standard deviation for an audio feature is calculated from the input sound's sequence for that feature concatenated with the target sound's sequence for that feature. This mean and standard deviation is then used in all further input standardization. This gives the series the quality of being centered around zero and having a standard deviation of 1 with respect to the input sound and the target sound. Additionally, to avoid extreme values, values are clipped to the range [-4, 4].

3.2.4 Parameter Mapping

The output of the neural network is in the range [0, 1], due to the sigmoid activation function in the output layer. These normalized values need to be scaled and skewed appropriately for each effect parameter. The author has used the same mapping function as in Cabbage (Walsh, 2008), a GUI framework for Csound, where slider values are mapped from 0 to 1 to the target range by the following function:

$$f(x) = m_{min} + (m_{max} - m_{min}) * e^{\log(x)/s}$$

Where m_{min} and m_{max} are the endpoints of the target range and s is the skew factor. The default skew factor is 1, which will yield a linear mapping. A skew factor of 0.5 will cause the mapping function to output values in an exponential fashion. This is useful for effect parameters such as cutoff frequency.

3.2.5 Audio Effects

In the toolkit, audio effects are included as text files containing Csound code. Each effect also has a corresponding file that lists the parameters of the effect and their range. This makes it easy to add new Csound effects to the toolkit. The toolkit uses a templating

system for generating Csound files, and this makes it possible to combine several audio effects into one larger audio effect with one or more layer of parallel audio effects, as tested in experiment 5. The implementation of the various audio effects used in this toolkit are based on trusted sources: The distortion effect with resonant low-pass filter is based on the source code of Brandtsegg (2015)[1] while the rest is based on source code of NTNUs online course in Digital Signal Processing (DSP) ear training[2]. These implementations have been selected in consultation with the author's co-supervisor, Brandtsegg.

3.2.6 Data Augmentation

Data augmentation is implemented as a python module that takes in a sound and creates a Csound file that repeats the sound multiple times with some variations and outputs the resulting audio to a new file. Each repetition of the sound has a unique combination of gain and playback speed. The playback speed and gain are sampled from a gaussian distribution centered around 1. The standard deviation of this gaussian distribution is configurable. To prevent clipping (samples out of bounds due to high gain values), a limiter is applied. Other ways to augment sound, such as adding reverb or distortion, can be easily implemented.

3.2.7 Live Mode

The toolkit includes a python module that can create a Csound file from an individual in an experiment. This Csound file includes an audio analyzer and a neural network that can run on live audio streams, so it can be used in live performances. It can also run in offline mode, to speed up computation when live mode is not needed. This is used in experiment 3 to apply evolved cross-adaptive audio effects to unseen data (i.e. sounds that were not used during training).

[1] `https://github.com/Oeyvind/interprocessing`
[2] `https://github.com/gdsp/gdsp/`

3.3 Visualization

In very early versions of the neuroevolution program, the author found it hard to evaluate all the data produced during experiments. Therefore an interactive web application for visualizing results was developed. This tool has been very important for being able to understand the strengths and weaknesses of the neuroevolution program during development and research. When the author gained a good understanding of the inner workings and the output of the neuroevolution program, he was able to improve the weak points of the implementation.

The visualization system is a single-page web application written in AngularJS, with various JavaScript libraries for visualizing data. For a complete list of JavaScript libraries that were used in the web application, see appendix C. The application server is written in NodeJS. The neuroevolution program writes data after each generation, and the NodeJS server listens for these data updates. Whenever new data is available, the updated data is sent via WebSockets to the web application, which then updates its views. The four following figures are screenshots of the web application. Figure 3.2 shows a line chart that visualizes the progress of the GA over the generations. Figure 3.3 shows a stacked area chart that visualizes the number of individual in each NEAT species over the generations. The fitness histogram in figure 3.4 visualizes the fitness distribution in the population of the generation selected by the interactive slider.

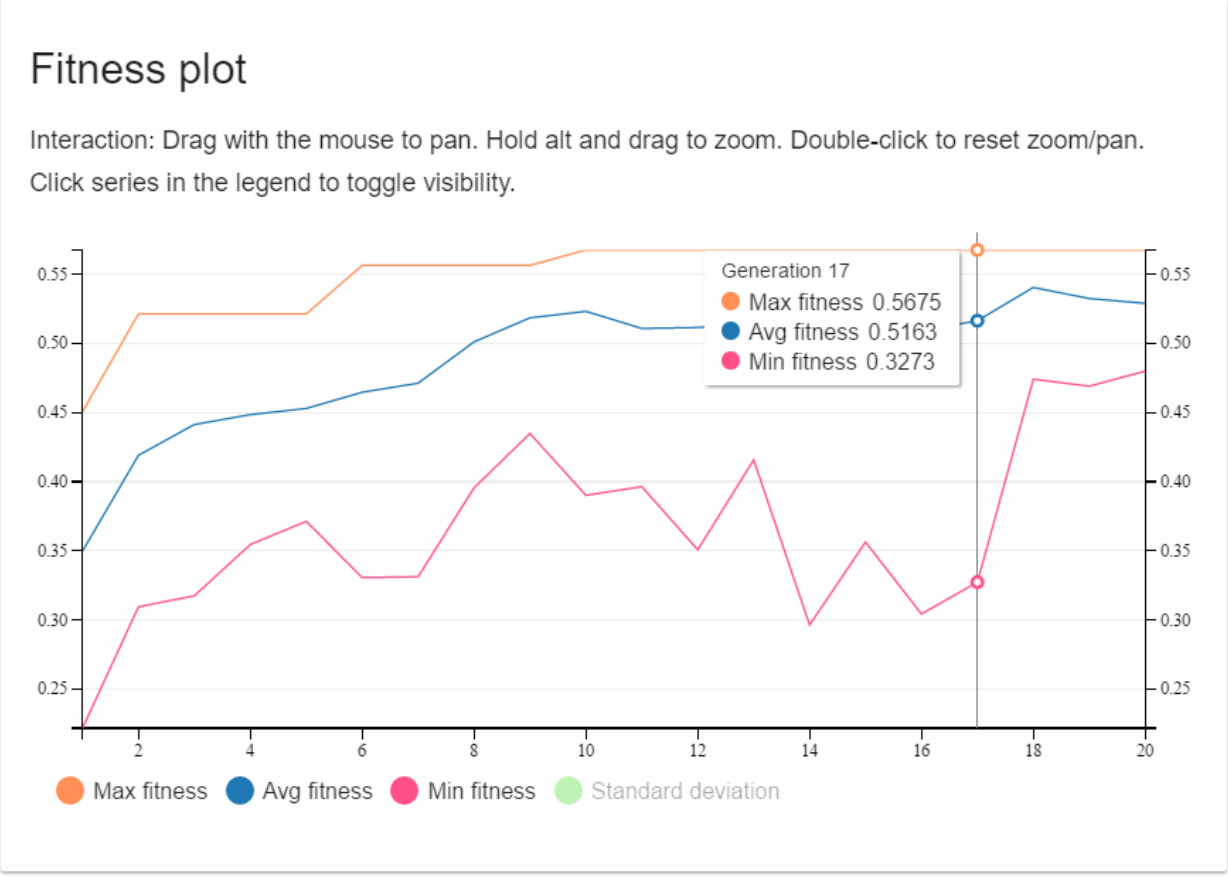

Figure 3.2: Fitness plot

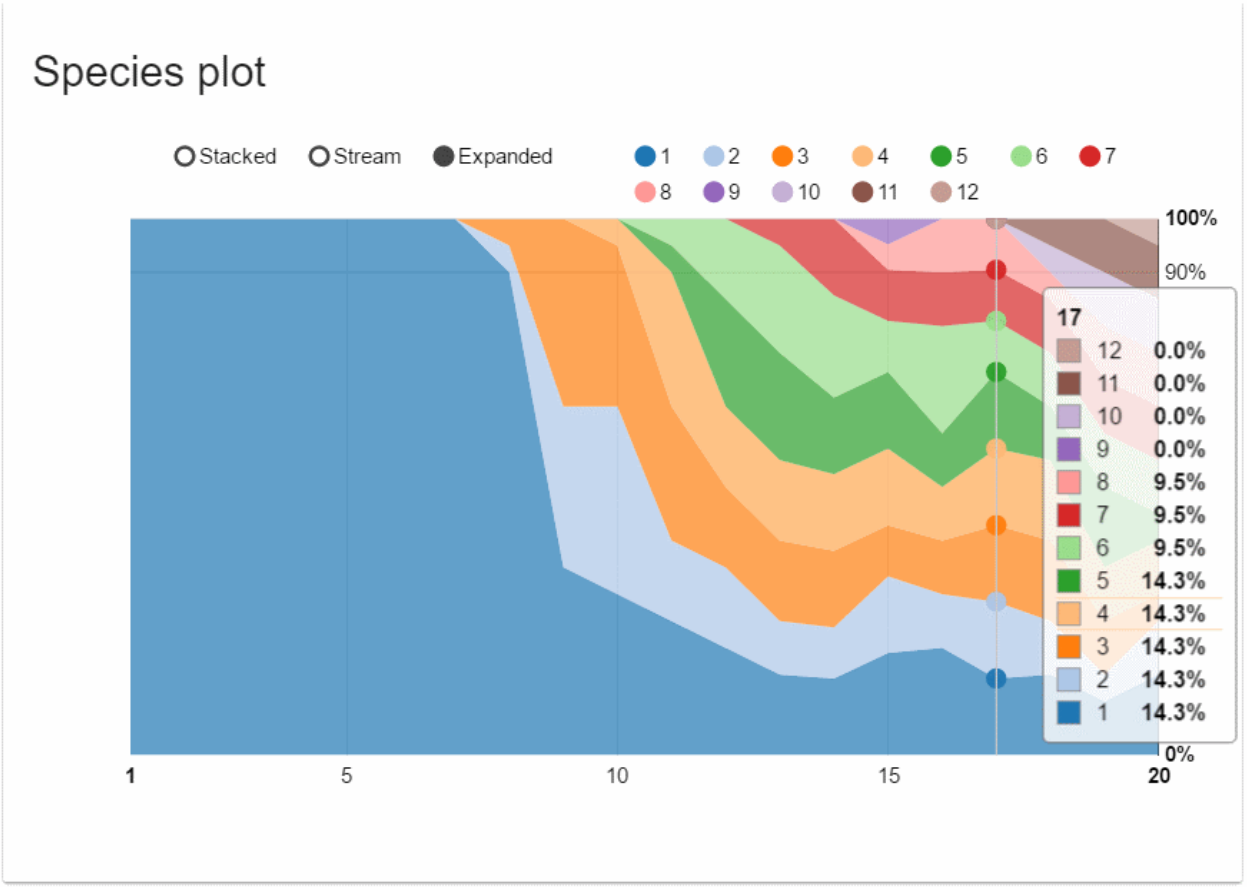

Figure 3.3: Species plot

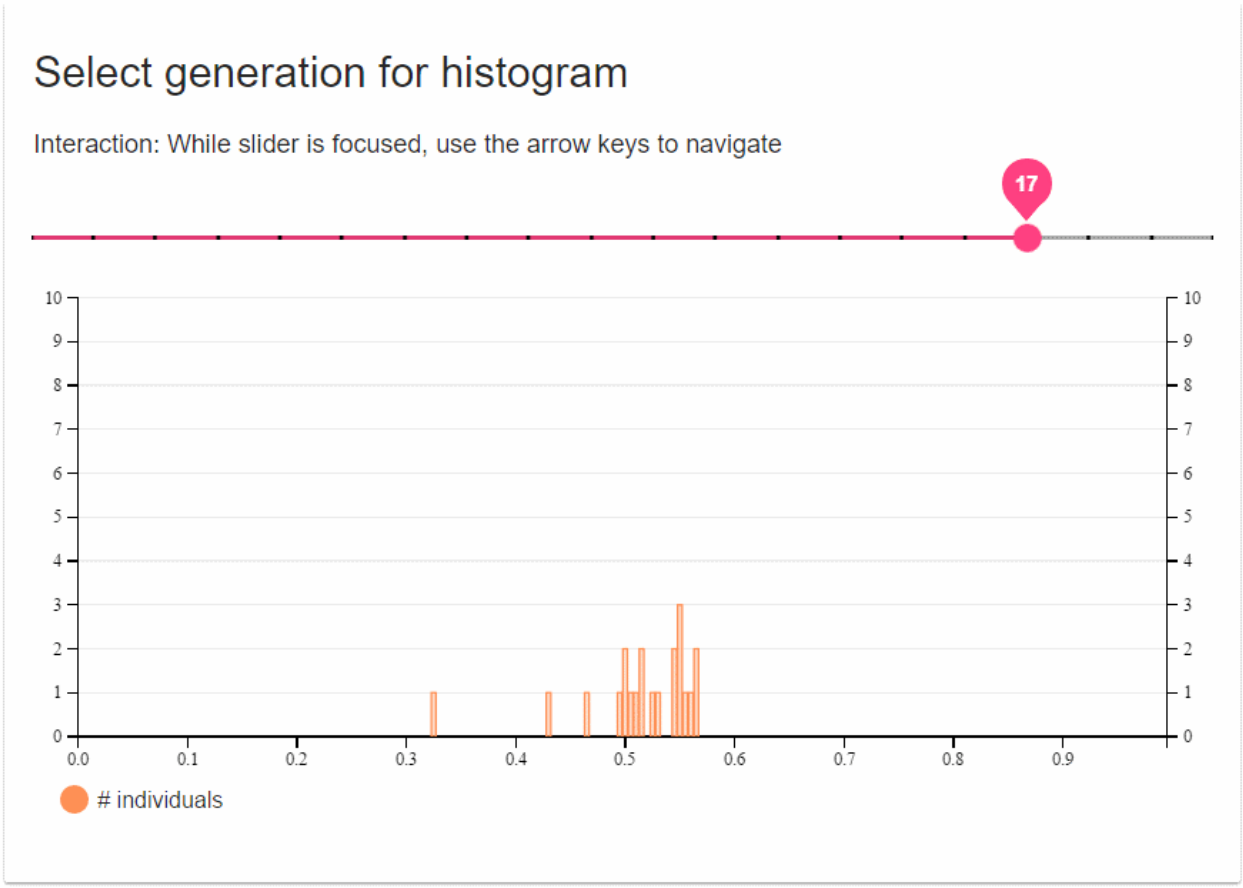

Figure 3.4: Fitness histogram for the selected generation

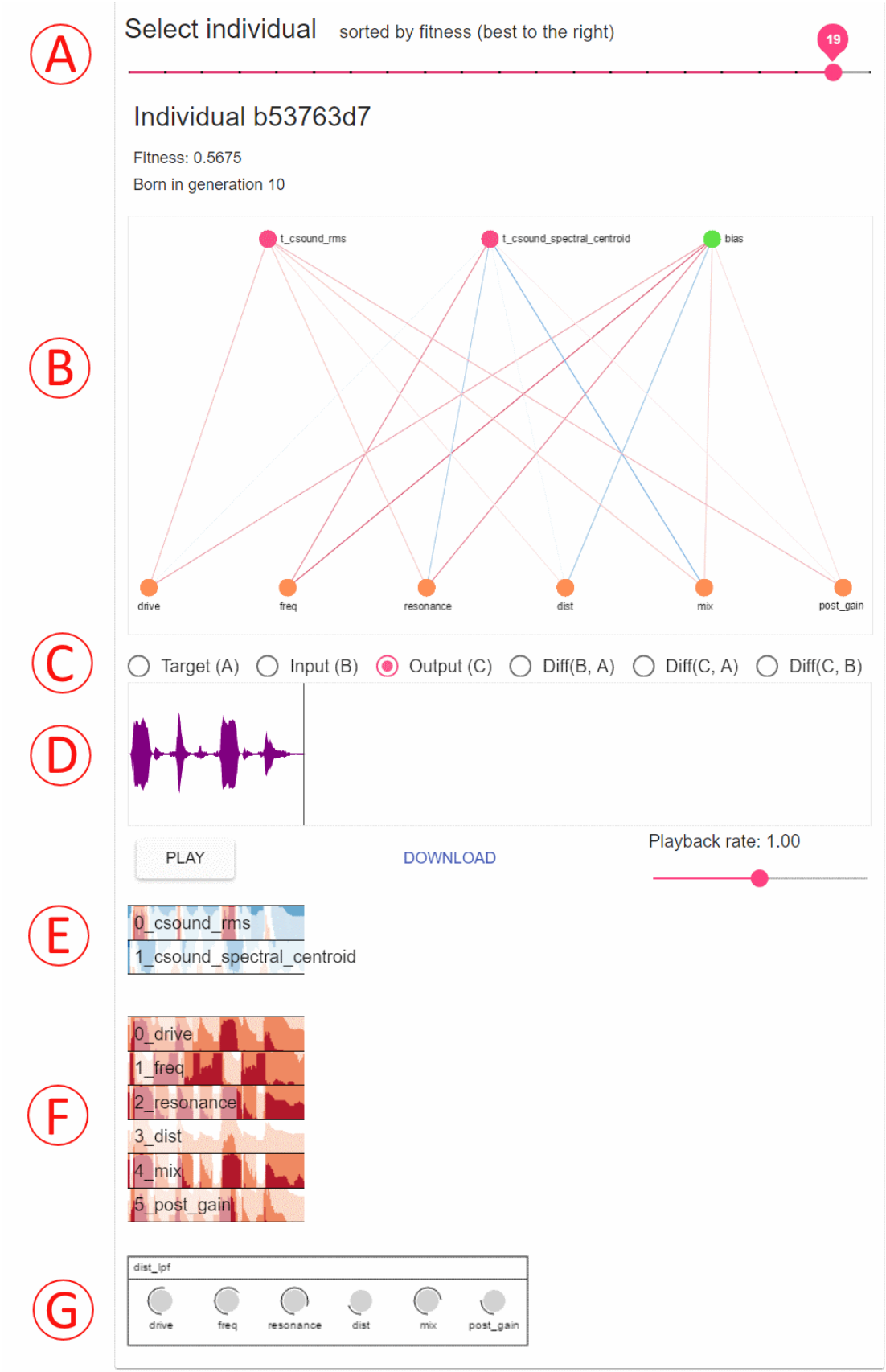

Figure 3.5: Visualization of data about an individual

Figure 3.5 shows various data visualizations related to the individual selected by the interactive slider. Here are brief explanations of the different parts:

- A: The interactive slider is for selecting a specific individual in the population of the selected generation. The interface below it updates whenever a different individual is selected.
- B: Neural network visualization with edge colors according to weight. Light color means small magnitude while strong color means large magnitude. Red means positive weight while blue means negative. It is possible to click a node to see a list of exact values for ingoing and outgoing weights.
- C: Select which sound and corresponding data series to visualize.
- D: Audio player with waveform visualization. Useful for playing back the output sound of the selected individual.
- E and F: Horizon charts for visualizing neural input and neural output, respectively. Horizon charts make better use of vertical space than standard area charts, allowing one to see many more metrics at-a-glance. Larger values are overplotted in successively darker colors, while negative values are offset to descend from the top. When hovering over these charts with the mouse pointer, series labels disappear and a rule with values at the hovered time step is shown. The horizon charts are aligned with the audio player.
- G: Shows the effect(s) and its parameters. While playing the audio, the knob positions are animated according to the way they were controlled in order to produce the output sound.

Chapter 4

Experiments and Discussion

Five experiments have been conducted in an attempt to find good ways to use neuroevolution for finding useful mappings from audio features to audio effect parameters. In each experiment, different neuroevolution configurations are compared to find what works better. Due to the random nature of genetic algorithms, results vary from run to run. To deal with this, each configuration gets multiple runs (with different Pseudo Random Number Generator (PRNG) seeds), and results from the runs are aggregated and presented in various figures. Table 4.1 shows a rough overview of the experiments.

Table 4.1: Overview of experiments

Experiment	**Description**
1	Find a good combination of mutation rate and crossover rate
2	Find a good value for structural mutation parameters
3	Apply data augmentation to the target sound, and see if the result generalizes better
4	Compare sets of audio features used in the fitness function. Analyze data from the neuroevolution process.
5	Compare networks of audio effects with individual audio effects.

4.1 General Configuration

Table 4.2 shows the parameters used unless otherwise stated in individual experiments. The table is not exhaustive. A number of NEAT parameters were left at their respective default values[1], set by Chervenski and Ryan, the authors of MultiNEAT.

Table 4.2: General experiment configuration

Parameter	**Value**
Population size	20
Add neuron probability	0.01
Remove neuron probability	0.01
Add link probability	0.01
Remove link probability	0.01
Elite fraction	0.1
Survival rate	0.25
Allow clones	Yes
Selection method	Tournament selection
Hidden activation function	Hyperbolic tangent
Output activation function	Sigmoid
Effect parameter low-pass filter cutoff frequency	50 Hz
Fitness function	Local similarity

[1] `https://github.com/peter-ch/MultiNEAT/blob/master/src/Parameters.cpp#L42`

4.2 Experiment 1

In this experiment, the aim is to find good values for crossover rate and mutation rate.

4.2.1 Configuration

Table 4.3: Experiment configuration

Parameter	**Value**
Number of generations	20
Target sound	Drum loop
Input sound	White noise
Effect	Distortion and resonant low-pass filter
Audio features	mfcc_0, mfcc_0__derivative, mfcc_1
Number of runs	150 per configuration

4.2.2 Results and Evaluation

Figure 4.1 shows that one should avoid using a high mutation rate and a low crossover rate. Instead, one of the combinations inside the red region should do well. Bear in mind that the differences between pure yellow and lime green are small in this region, and that these small differences are not statistically significant. The variance could be reduced with more runs, but due to computational time, the number of runs per configuration was limited to 150. The yellow spot is probably a good configuration, albeit not necessarily the best. Mutation rate = 0.6 and crossover rate = 0.7 are used in the following experiments.

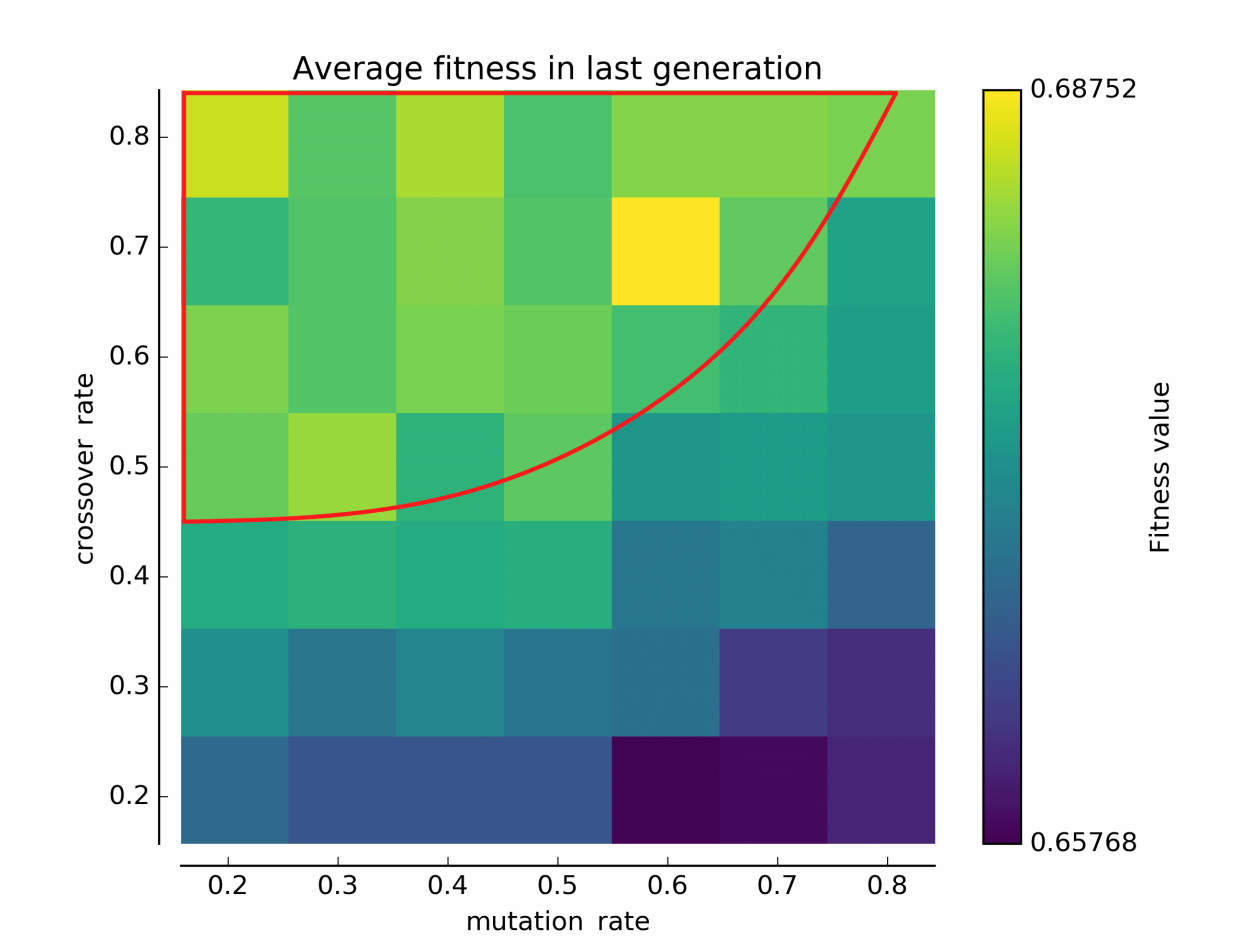

Figure 4.1: The red region drawn on top of the heat map indicates the set of configurations deemed good

4.3 Experiment 2

4.3.1 Configuration

In this experiment, the aim is to find a good value for parameters related to structural mutation:

- Add node probability
- Remove simple node probability
- Add link probability

- Remove link probability

Four different configurations will be tested: $p = 0.01$, $p = 0.03$, $p = 0.09$, $p = 0.27$ where p is the value assigned to the four structural mutation parameters.

Table 4.4: Experiment configuration

Parameter	**Value**
Number of generations	50
Target sound	Drum loop
Input sound	White noise
Effect	Distortion and resonant low-pass filter
Audio features	mfcc_0, mfcc_0__derivative, mfcc_1
Number of runs	300 per configuration

4.3.2 Results and Evaluation

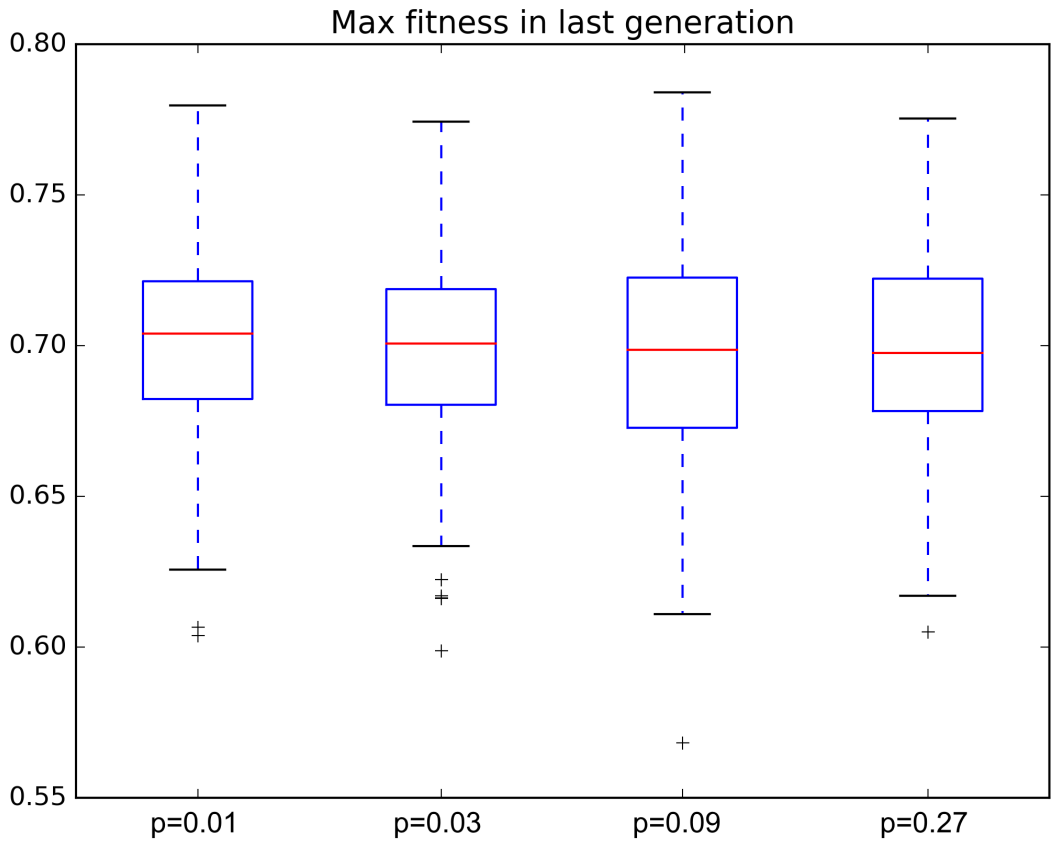

Figure 4.2: Box plot of fitness values in the 50th generation for each configuration

The median values in figure 4.2 are not solid evidence that one configuration is better than another. However, figure 4.3 shows that while all series converge to roughly the same average fitness value, $p = 0.01$ yields the highest rate of convergence. In fact, as p becomes larger, the rate of convergence declines. This might mean that the problem at hand is best solved with few or no hidden nodes. An alternative interpretation is that hidden nodes are useful, but that they slow down the search for the ultimate individual, due to increased complexity. In accordance with Occam's razor, the preferred choice is $p = 0.01$, as it produces the neural networks with the simplest structures (Mitchell, 1997).

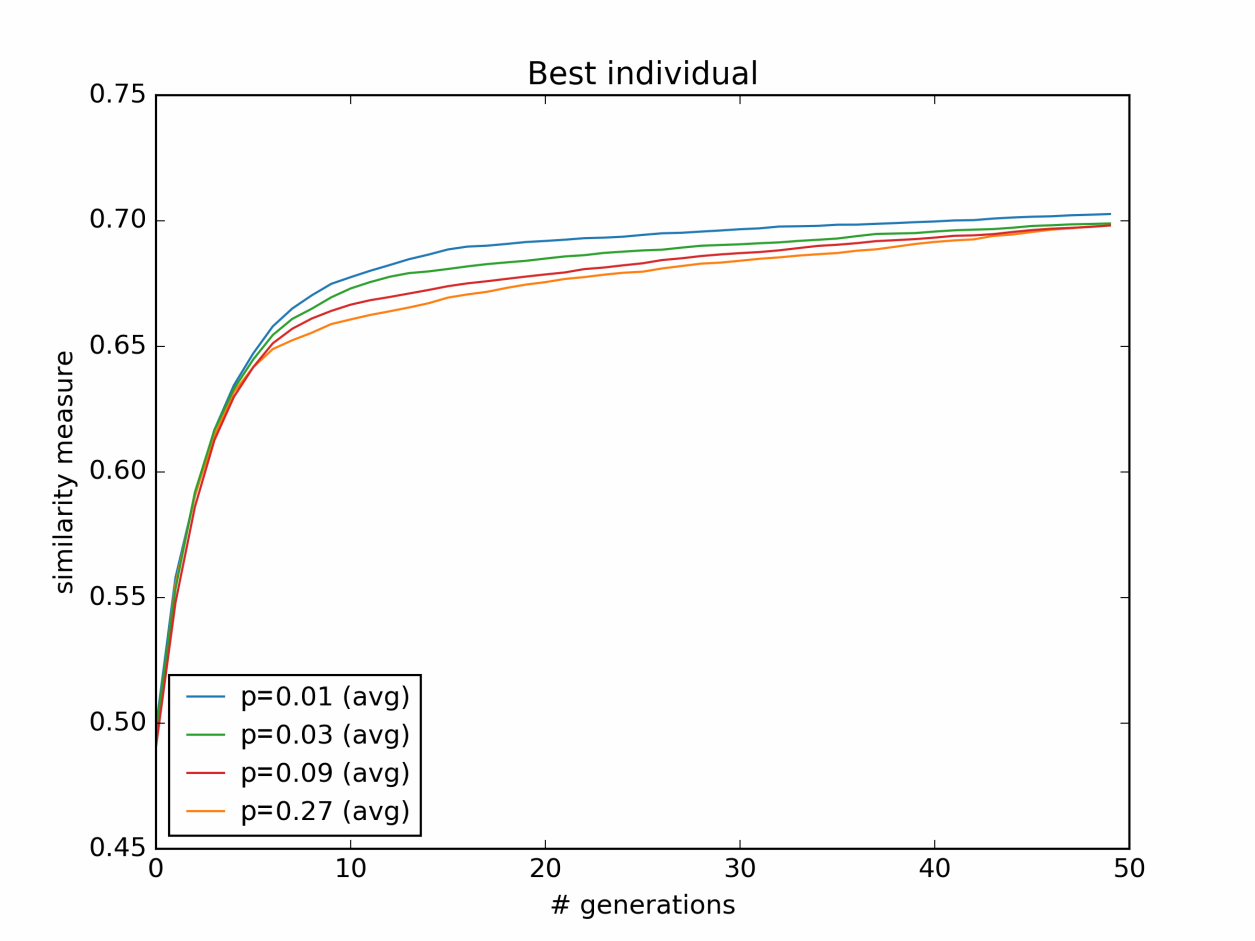

Figure 4.3: Line chart showing the average fitness values in each generation

4.4 Experiment 3

When using an evolved cross-adaptive audio effect in a live performance, a performer may want to use it in an expressive way. For example, if the performer is a drummer, he/she can vary the intensity of the drum hits. For the cross-adaptive audio effect to handle this, it needs to be trained on all the different intensities of the drum hits. If an extensive recording is available, that is fine. However, if the available target sound is short or lacks sufficient variation, one can harness the concept of data augmentation to create artificial variations of that sound. If one uses that sound instead, the evolved effect will typically be more capable of dealing with the generated variations. This experiment is about testing the author's implementation of data augmentation and the ability to apply evolved cross-adaptive audio

effects to unseen sounds.

4.4.1 Configuration

Table 4.5: Experiment configuration

Parameter	**Value**
Number of generations	20
Target sound (training)	Drum loop with bass drum, snare drum, clap and hihat (figure 4.4)
Target sound (validation)	Snare roll (rapid snare drum hits) with ascending pitch and amplitude (figure 4.4)
Input sound	White noise
Effect	Distortion and resonant low-pass filter
Audio features	Root Mean Square (RMS) and spectral centroid
Number of runs	40 per configuration

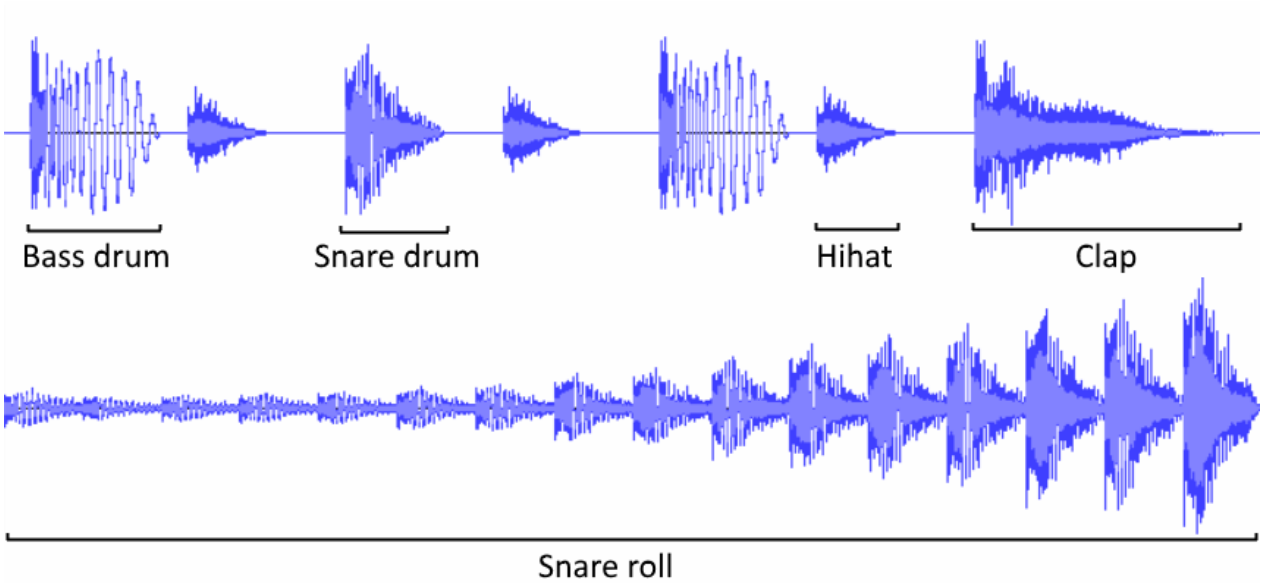

Figure 4.4: Waveform of training sound (top) and validation sound (bottom)

The augmented variant of the training sound was created by repeating the sound 8 times, with variations in playback speed and gain for each repetition. The playback speed and gain are obtained by sampling from a gaussian distribution with standard deviations of 0.3 and 0.5, respectively.

Runs with three different target sounds (training sound, augmented training sound and validation sound) will be compared. The resulting effects are applied to the validation sound, and fitness values are measured. The neural networks trained on the validation sound are of course expected to be yield the highest fitness scores when tested on the validation sound. That configuration is included for comparison, as an upper bound estimate.

4.4.2 Results and Evaluation

Figure 4.5 shows that neural networks trained on an augmented variant of the training sound generalize better than neural networks trained on the nonaugmented training sound. In other words, the resulting audio effects becomes better at dealing with nuances of the situations in the training sound. There's a trade-off, however: Training on an augmented sound requires more computational time because there's more data to process. This may not be a problem, because training can be done before live performance starts.

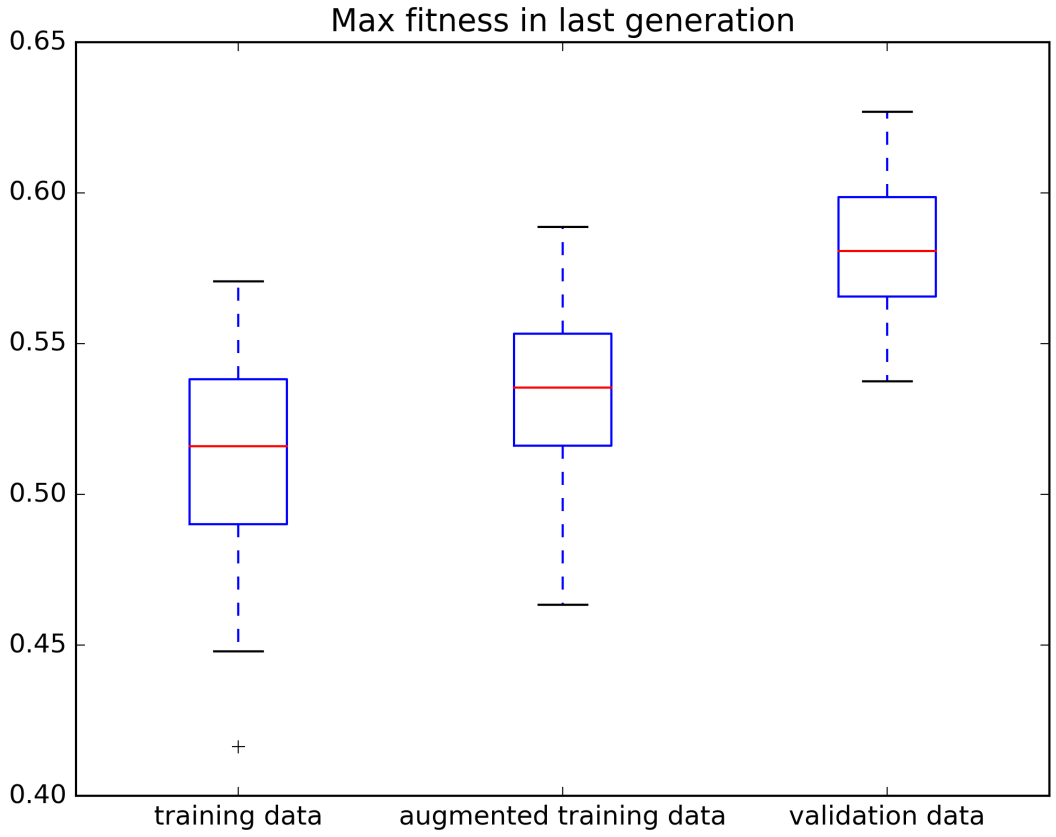

Figure 4.5: Box plot of validation fitness values in final generation. The labels on the x-axis indicate which sound was used as target sound.

4.5 Experiment 4

4.5.1 Configuration

In this experiment, the aim is to compare two different collections of audio features used in the similarity measure:

- Configuration A: RMS, pitch and spectral centroid
- Configuration B: RMS, pitch, spectral centroid and bark bands

Table 4.6: Experiment configuration

Parameter	Value
Number of generations	500
Target sound	Sine wave, 440 Hz
Input sound	White noise
Effect	Band-pass filter with up to 10x post gain
Number of runs	20 per configuration

4.5.2 Results and Evaluation

Since fitness functions were different in these two configurations, the fitness values are not directly comparable. Instead, the results were evaluated by manually listening to the output sounds. In the first configuration, the results were fairly bad: All of the sounds were too noisy, and the author failed to perceive the tone. However, in terms of spectral centroid and amplitude, the sounds were a good match. In order to transform noise into a sine, the bandwidth of the band-pass filter has to be very narrow. A narrow filter would have lowered the overall amplitude of the sound. This would have been deemed bad by the fitness function, hence the genetic algorithm did not effectively explore that area in the solution space. Also, a narrow filter would not have yielded any improvements in the similarity in spectral centroid and/or pitch. Therefore, the typical solution has a broad bandpass filter, albeit with an appropriate center frequency. See in figure 4.6 that the peak frequency of the typical output sound matches well the peak frequency of the target sound.

The results in the second configuration, were much better. The author could hear a clear tone in all 10 output sounds. There was still some noise in most sounds. The author believes that the solutions would have improved with more generations, because the fitness was typically still increasing towards the 500th (last) generation. One of the output sounds featured vibrato (varying pitch over time), due to a noisy input being mapped to the center

frequency parameter. This could probably have been alleviated by adding the derivative of the pitch as a dimension in the fitness function, so the unwanted vibrato would be punished more severely by the fitness function.

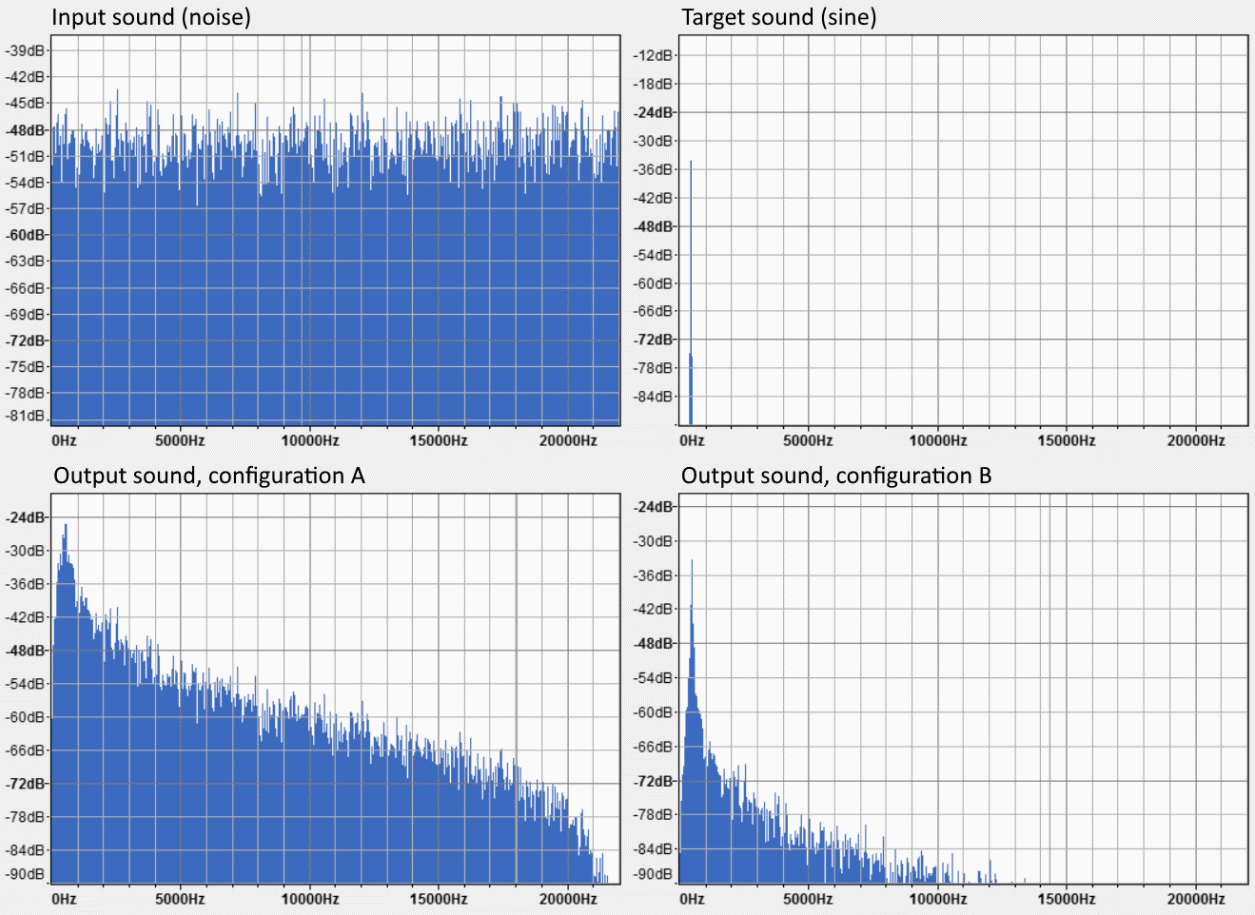

Figure 4.6: Spectrum plots, created in Audacity® with hanning window of size 8192

The takeaway from this experiment is that

- the collection of features used for similarity measures has a high impact on the result
- bark bands are useful when used in the similarity measure

4.5.3 Neuroevolution Analysis

In this subsection, one of the runs in configuration B will be analyzed, to get a better understanding of how NEAT works in practice, and to identify some things that can be improved upon.

In this experiment, the effect parameters do not have to be *dynamically* controlled over time. A static value for each effect parameter (bandwidth, center frequency and post gain) would have been an optimal solution. However, in figure 4.8, we see that the NEAT algorithm actually makes use all three input nodes (RMS, pitch and spectral centroid) as well as the constand bias node. The reason for this is that these three inputs do not vary much in the course of the sound, because the features of the sound that was analyzed (sine, 440 Hz) do not vary much over time (see the horizon chart in figure 4.7). Consequently, these three nodes act like a bias node with some noise.

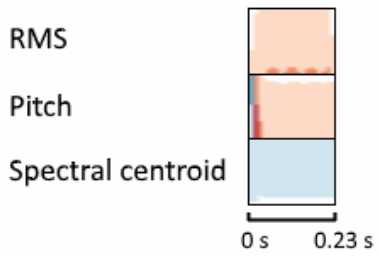

Figure 4.7: Horizon chart visualizing the three neural inputs over time

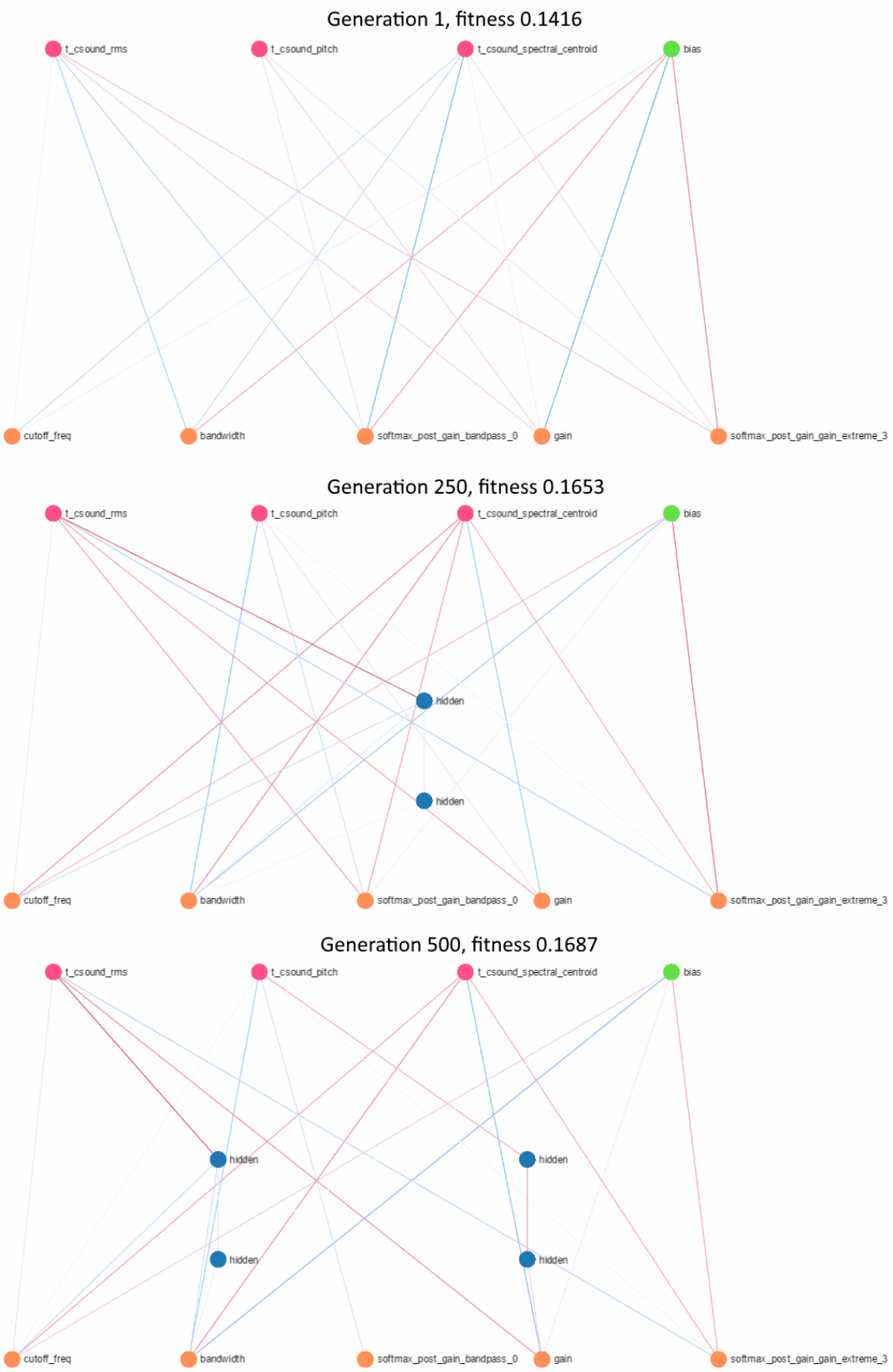

Figure 4.8: Neural network visualizations

Another interesting fact is that the number of hidden nodes in the best individual tends to increase over the generations, even though these hidden nodes are not needed to get an optimal solution in this experiment. Any constant value can be sent to the output node without having to go through hidden nodes. This is another example of NEAT not finding the optimal solution. Having many hidden nodes hurts evolvability, because an arbitrary mutation on a complex individual is less likely to yield an improvement than the same operation on a simpler individual with fewer hidden nodes. Some form of regularization (punishment for more complex neural networks) could alleviate the problem. Another way to deal with this problem in experiments that do not require hidden nodes is setting AddNodeProbability to zero.

Figure 4.9 shows a line chart of max fitness and average fitness. The max fitness does not increase in every generation. This is due to the random nature of the mutations. As the iterative process approaches convergence, jumps in max fitness become smaller and smaller.

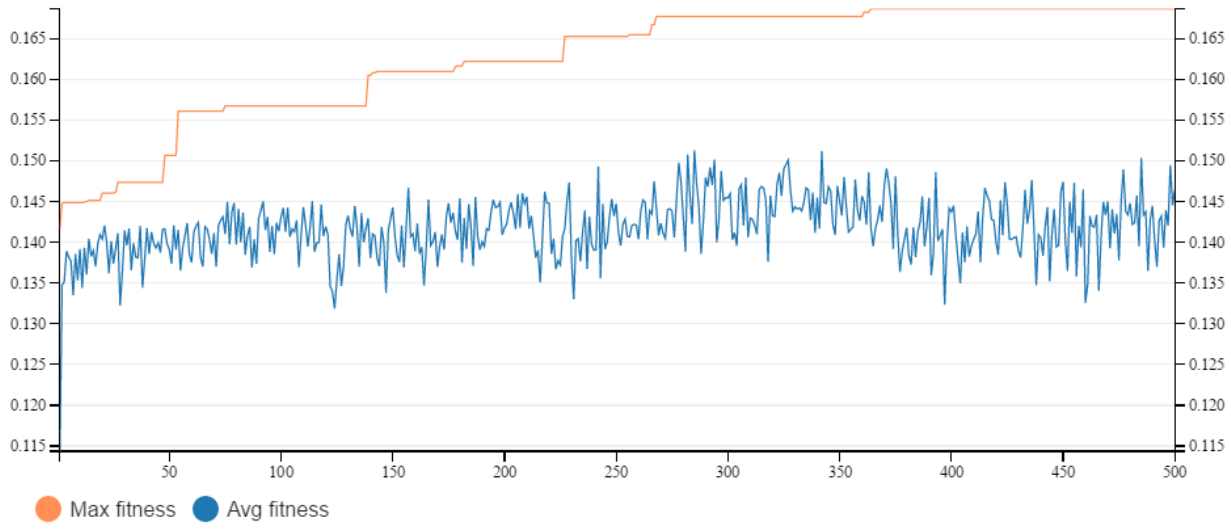

Figure 4.9: Fitness plot

Figure 4.10: Species chart with max fitness overlay

A total of 54 different species were created during the 500 generations of neuroevolution (see figure 4.10). There is no obvious correlation between jumps in max fitness and the death of existing species or the emergence of new species. This makes sense, as competition happens only within species, and species are given time to improve their structure before competition with the rest of the population occurs. The figure also illustrates that the number of species in any given generation is well regulated. The concept of dynamic compatibility threshold in NEAT strives to ensure that the number of species stays within bounds (Stanley and Miikkulainen, 2004).

4.6 Experiment 5

4.6.1 Configuration

This experiment is about combining several audio effects in serial and parallel. The idea is that this allows for altering the sound in more ways than with only a single audio effect. This may yield improvements in fitness compared to using a single audio effect. The hypothesis is that the genetic algorithm could be adept at choosing what effects to use and how to combine

them. This experiment will study two different effect networks, illustrated by figure 4.11 and 4.12. From now on, these two configurations will be referred to as "1 layer" and "2 layers", respectively. To find out how well these effect networks perform, they will be compared to runs with individual audio effects.

The effect mix values in one layer are determined by the output of the softmax function (4.1), also called the normalized exponential function. This makes the sum of effect mixes in one layer equal to one. It also lets the neural network easily "choose" one single audio effect to dominate at any given time, but mixing multiple audio effects is also within reach.

$$softmax_i(a) = \frac{\exp a_i}{\sum \exp a_i} \tag{4.1}$$

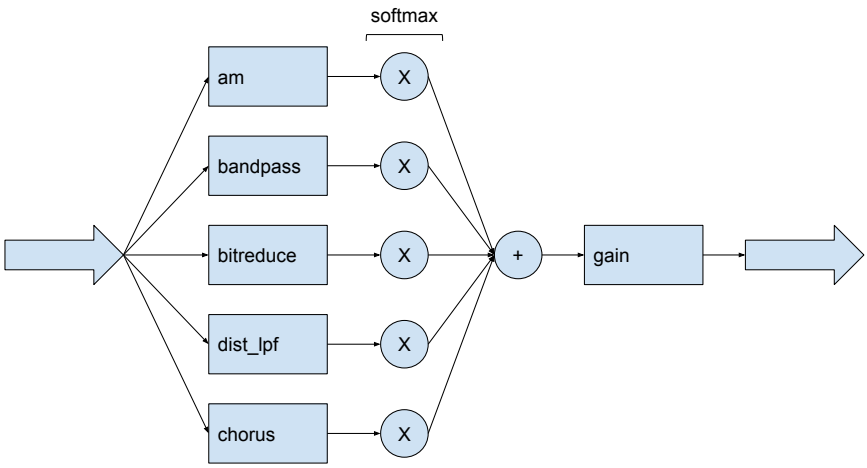

Figure 4.11: Signal flow in the "1 layer" configuration

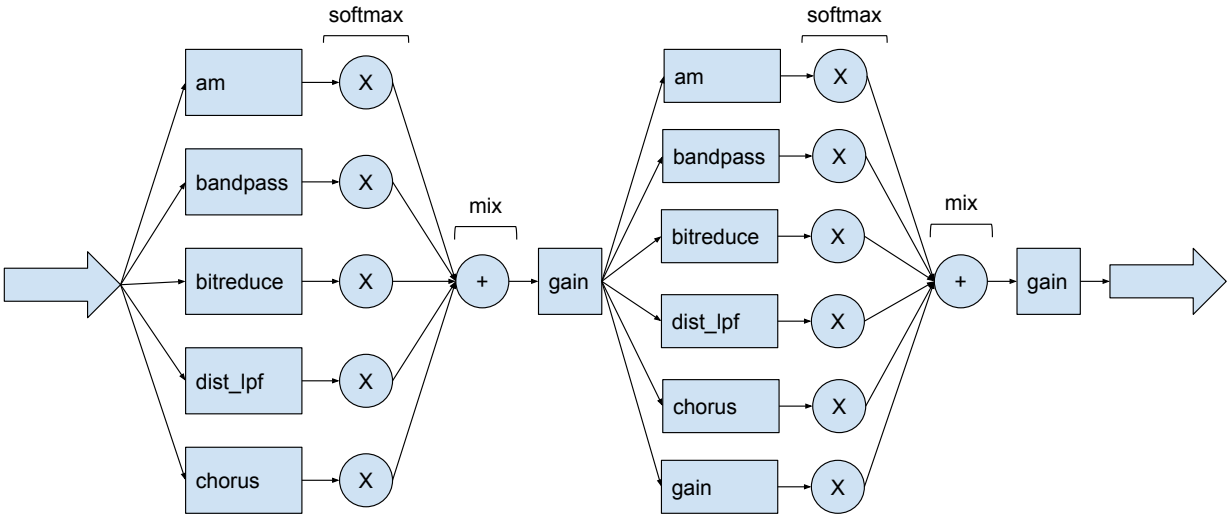

Figure 4.12: Signal flow in the "2 layers" configuration

Table 4.7: Experiment configuration

Parameter	**Value**
Number of generations	500
Population size	40
Target sound	Drum loop
Input sound	White noise
Audio features for neural input	mfcc_0, mfcc_0__derivative, mfcc_1
Audio features for similarity measure	mfcc_0, mfcc_0__derivative, mfcc_1 and bark bands
Number of runs	20 per configuration

4.6.2 Results and Evaluation

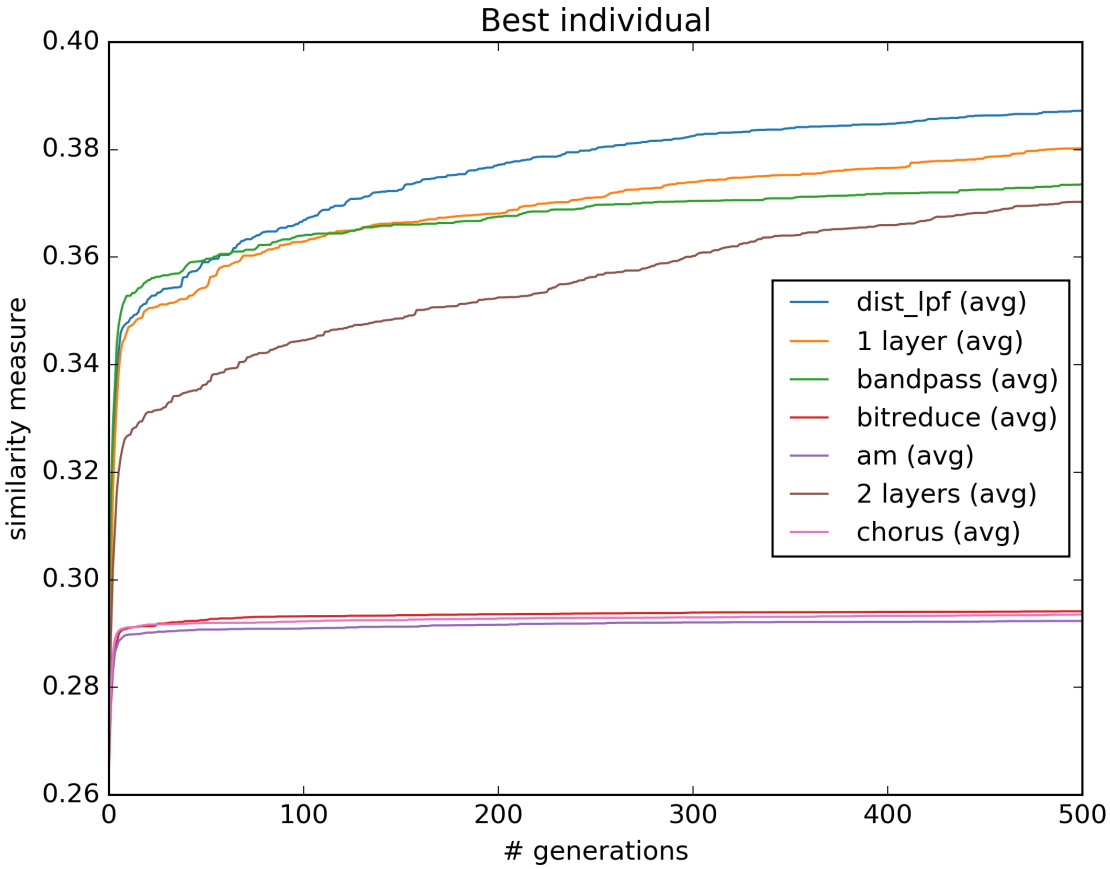

Figure 4.13: Aggregated fitness values

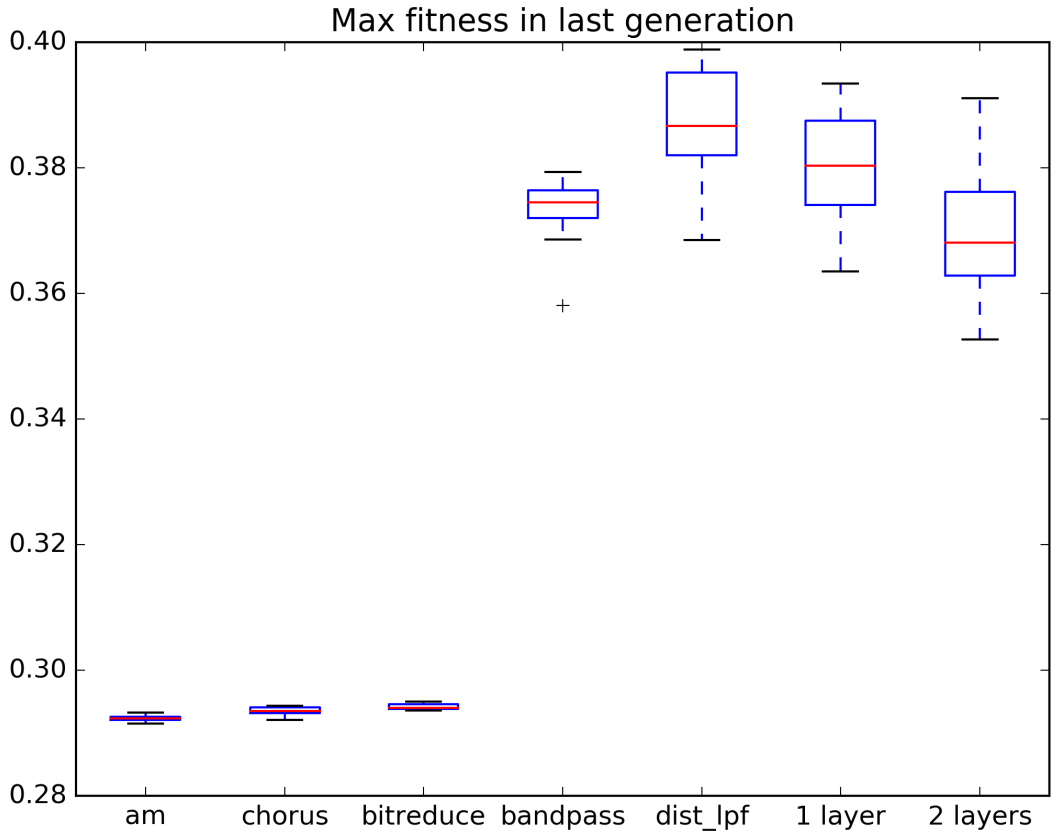

Figure 4.14: Box-and-whisker plot of fitness values in the last generation

As illustrated by figure 4.13 and 4.14, three of the audio effects (amplitude modulation, chorus and bitreduce) achieve bad scores when used individually. The reason is that they are not able to shape/filter the noise in a desirable manner. This makes sense, as these three audio effects typically make the sound "richer". The band-pass filter and low-pass filter are more useful in this experiment, because they can filter the wide spectrum of the input sound (white noise) to something that sounds more like the target sound (drums). Hence low mix values for am, chorus and bitreduce are expected in the "1 layer" and the "2 layers" runs. Indeed, when inspecting the mix values in the best runs, we find that bandpass and dist_lpf are used much (see figure 4.15 and 4.17). However, we also find that chorus and bitreduce are used for representing the snare drum (the rightmost part). In the sound illustrated by figure 4.15 the mix values change rapidly at one point, and this gives the snare drum a flutter-like texture and creates the illusion of a short echo. The waveform of this sound can be seen in

figure 4.16.

While the algorithm may have found out which effects are not useful, it has not found out how to use the useful effects in a good way. Note that dist_lpf alone yields better results than 1 layer. Theoretically, both 1 layer and 2 layers could produce solutions as good as those with dist_lpf, but that typically does not happen, at least not with the same number of generations.

One key takeaway from this experiment is that one should only use audio effects that are known to be useful in the context of the experiments. Adding unfitting audio effects to an audio effect network makes the search space larger and has a negative impact on the convergence rate. In other words, to get good results with effect networks, let a human expert create the audio effect structure.

Here are some ideas that could be applied to improve the results of audio effect networks:

- Perform pre-training on the parameters of the various audio effects separately. Then freeze those neural networks and train the mix of the output from the effects when they are used in parallel.
- The assumptions behind softmax mix values might be suboptimal. Using independent mix values, i.e. without any normalization, might yield better results.
- Use Cartesian Genetic Programming (CGP) to automatically evolve networks of interconnected audio effects.

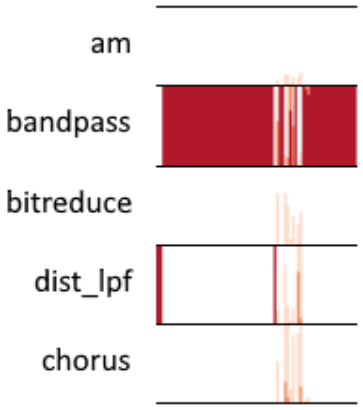

Figure 4.15: Mix values for each effect in the best result from the runs with 1 layer

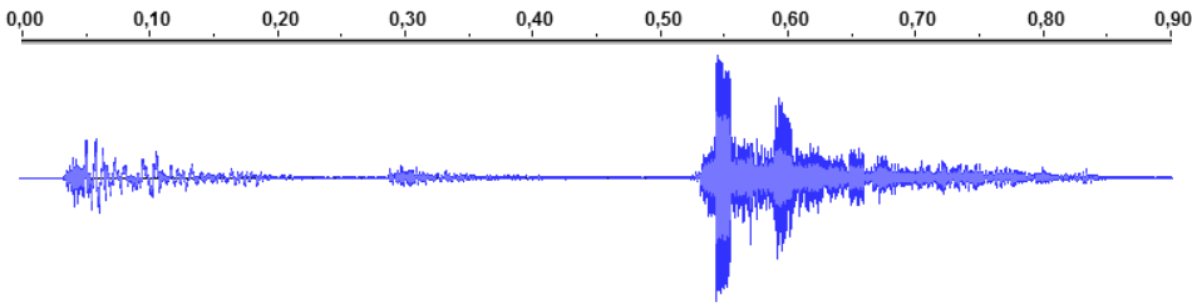

Figure 4.16: Waveform of the best output sound from the runs with 1 layer

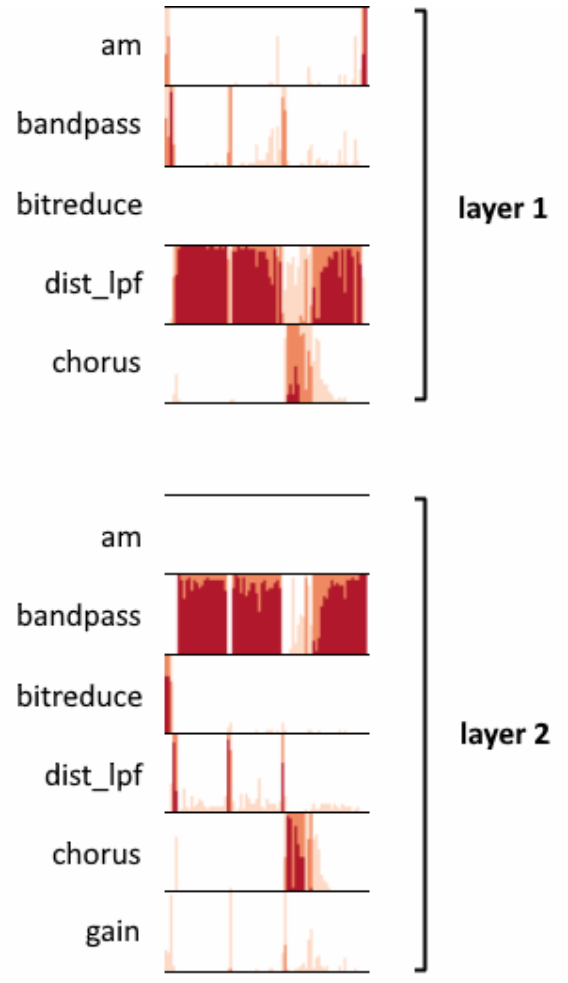

Figure 4.17: Mix values for each effect in the best result from the runs with 2 layers

Chapter 5

Conclusion

It has been shown that using neuroevolution for finding useful mappings in cross-adaptive audio effects is feasible. This is good because finding these signal mappings by empirical experimentation can be tedious and hard due to the vast number of combinations.

Several fitness functions have been developed and compared. Based on qualitative evaluations, the hybrid variant, that is a combination of local euclidean distance and NSGA-II-inspired multi-objective optimization, has been found to yield the best results. Furthermore, in experiments with high-dimensional spaces, FS-NEAT has been proven to do better than NEAT, because FS-NEAT chooses only a few useful connections rather than a fully connected neural network.

A comprehensive toolkit has been developed during the course of the project. The toolkit includes an interactive visualization tool that makes it possible to evaluate results and understand the neuroevolution process. The toolkit has lots of configuration options, enabling a flexible platform for experimentation. It is open source, has documentation and can be used in future research within the field of cross-adaptive audio effects.

While an evolved cross-adaptive audio effect may perform well on the combination of the input sound and the target sound it was trained on, it is also desirable to be able to successfully apply the effect to other sounds. In particular, this is useful in live performances where audio effects are applied to unseen sound. When training data is scarce, data augmentation can be applied to evolve cross-adaptive audio effects that perform better on unseen sound that deviates from the training sound. Audio effects produced by the toolkit can be used in

live performances, thanks to the implementation of audio analysis, audio effects and artificial neural network that can run on live audio streams in Csound.

5.1 Future Work

As stated in the introduction, current research at the Music Technology department at Norwegian University of Science and Technology aims at exploring radically new modes of musical interaction in live music performance. This project is a good start, but there is still a lot to be explored. For example, it would be interesting to try other audio effects than the six audio effects used in this project. Also, it does not have to be effects. It can also be sound generators, such as synthesizers or sword sound emulators, with parameters. For example, a foley artist can imitate a sword sound with his mouth, then evolve parameters that make the sword sound generator produce a sound like that. This relates to the work of Cartwright and Pardo (2014). While their project is based on interactive, iterative refinement through user-provided relevance feedback, neuroevolution can automate that process to save time. That could make foley sound production easier and less time-consuming.

More work can also be done on combining multiple audio effects into one composite audio effect. In experiment 5, layers of parallel audio effects were tested, and the results were not better than the best effect used individually. However, this does not mean that composite effects are generally bad. There are lots of techniques to be explored that might improve composite effects. For example, one could use Cartesian Genetic Programming (CGP) to automatically evolve the topology of effect networks.

Picbreeder (Secretan et al., 2008) and Soundbreeder (Ye and Chen, 2014) had success with HyperNEAT, which is one variant of NEAT that has not been tried in this project. HyperNEAT might be able to produce better results than NEAT in this project, but probably only in experiments where the input nodes and output nodes have some sort of geometrical meaning (Whiteson and T. van den Berg, 2013).

While live mode is implemented and technically works, it has not been tried by music performers yet. There is much work to be done on finding the role of evolved cross-adaptive audio effects in live performances. For example, one has to find out which instruments interact

with each other's cross-adaptive effects and which audio effects are musically interesting and appealing. Other potential issues, such as latency and audio feedback, also need to be dealt with.

The author imagines that methods developed in this project could be used for mastering/mixing music and also for novel crossfading in DJ mixing software. However, that would require smart methods for dealing with long sounds (several minutes). This project has only dealt with short sounds (up to 16 seconds) so far. When dealing with longer sounds the author sees two challenges: 1) Computational time and 2) A long sound might have several very different parts, and the evolved neural network might have trouble dealing well with all of them. One possible solution to these challenges is to chop the long sound into a few short audio segments that represent the different parts of the sound well and then run the program on each audio segment. When applying the resulting neural networks on new sounds, the program can automatically fade between the evolved artificial neural networks based on similarity with the various audio segments they were trained on.

Acknowledgments

I express gratitude towards my supervisors Øyvind Brandtsegg and Gunnar Tufte for guidance and valuable feedback during this project. I would also like to the Department of Computer and Information Science, NTNU for providing me with a Linux Virtual Machine to run experiments on. Thanks to students at the office and my girlfriend for supporting me and listening to my ramblings about sounds and genetic algorithms. Thanks to Sigve S. Farstad for valuable feedback. Lastly, I would like to thank contributors to the various open source software and libraries that have been used throughout the project.

Bibliography

Bäck, T. (1996). Evolutionary algorithms in theory and practice: Evolution strategies, evolutionary programming, genetic algorithms. Oxford Univ. Press, New York.

Bogdanov, D., Wack, N., Gómez, E., Gulati, S., Herrera, P., Mayor, O., Roma, G., Salamon, J., Zapata, J. R., and Serra, X. (2013). Essentia: An audio analysis library for music information retrieval. In *ISMIR '13*, pages 493–498.

Brandtsegg, Ø. (2015). A toolkit for experimentation with signal interaction. In *Proceedings of the 18th International Conference on Digital Audio Effects (DAFx-15)*, pages 42–48.

Brossier, P. (2003). Aubio, a library for audio labelling. Retrieved January 8, 2017, from `http://aubio.org/`.

Bullock, J. (2007). LibXtract: A lightweight library for audio feature extraction. In *Proceedings of the International Computer Music Conference*.

Cartwright, M. and Pardo, B. (2014). SynthAssist: Querying an audio synthesizer by vocal imitation. In *Proceedings of the ACM International Conference on Multimedia - MM '14*. doi:10.1145/2647868.2654880.

Caudill, M. (1987). Neural networks primer, part I. *AI Expert*, 2(12):46–52.

Chervenski, P. and Ryan, S. MultiNEAT neuroevolution library. Retrieved January 8, 2017, from `http://www.multineat.com/`.

Colletti, J. (2013). Beyond the basics: Sidechain compression. Retrieved January 10, 2017, from `http://www.sonicscoop.com/2013/06/27/beyond-the-basics-sidechain-compression/`.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation IEEE Trans. Evol. Computat.*, 6(2):182–197. doi:10.1109/4235.996017.

Dodge, C. and Jerse, T. A. (1997). *Computer music: synthesis, composition, and performance*. Schirmer Books.

Eldhuset, A. W. (2015). Experiments in using genetic algorithms to find parameters for adaptive audio effects. Unpublished specialization project, Norwegian University of Science and Technology.

Fitch, J. P., Lazzarini, V., Yi, S., Gogins, M., et al. Csound: Sound and music computing system. Retrieved January 8, 2017, from `http://csound.github.io/`.

GDSP Online Course (2014a). Bitreduction. NTNU, Department of Music, Music Technology. Retrieved January 9, 2017, `fromhttp://gdsp.hf.ntnu.no/lessons/1/4/`.

GDSP Online Course (2014b). Chorus. NTNU, Department of Music, Music Technology. Retrieved January 9, 2017, from `http://gdsp.hf.ntnu.no/lessons/14/53/`.

GDSP Online Course (2014c). The modified tanh() function. NTNU, Department of Music, Music Technology. Retrieved January 9, 2017, from `http://gdsp.hf.ntnu.no/lessons/3/18/`.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning*. Addison Wesley, Massachusetts.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257. doi:10.1016/0893-6080(91)90009-t.

Lecun, Y., Bottou, L., Orr, G. B., and Müller, K. (1998). Efficient backprop. *Lecture Notes in Computer Science Neural Networks: Tricks of the Trade*, pages 9–50. doi:10.1007/3-540-49430-8_2.

Lehman, J. and Stanley, K. O. (2008). Exploiting open-endedness to solve problems through the search for novelty. In *Proceedings of the Eleventh International Conference on Artificial Life (ALIFE)*. Retrieved January 8, 2017, from `http://eplex.cs.ucf.edu/papers/lehman_alife08.pdf`.

Lehman, J. and Stanley, K. O. (2015). Novelty search users page. Retrieved January 8, 2017, from `http://eplex.cs.ucf.edu/noveltysearch/userspage/`.

Logan, B. (2000). Mel frequency cepstral coefficients for music modeling. Retrieved January 8, 2017, from `http://musicweb.ucsd.edu/~sdubnov/CATbox/Reader/logan00mel.pdf`.

Mermelstein, P. (1976). Distance measures for speech recognition, psychological and instrumental. *Pattern recognition and artificial intelligence*, 116:374–388.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.

Moritz, A. (2003). Stockhausen hymnen introduction. Retrieved January 8, 2017, from `http://home.earthlink.net/~almoritz/hymnenintro.htm`.

Reiss, J. D. (2011). Intelligent systems for mixing multichannel audio. In *2011 17th International Conference on Digital Signal Processing (DSP)*. doi:10.1109/icdsp.2011.6004988.

Sarle, W. (2014). Comp.ai.neural-nets FAQ, part 2 of 7: Learning section - should I normalize/standardize/rescale the data? Retrieved January 8, 2017, from `http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-16.html`.

Schmidhuber, J. (2009). Simple algorithmic theory of subjective beauty, novelty, surprise, interestingness, attention, curiosity, creativity, art, science, music, jokes. Retrieved January 10, 2017, from `http://people.idsia.ch/~juergen/sice2009.pdf`.

Secretan, J., Beato, N., D'Ambrosio, D., Rodriguez, A., Campbell, A., and Stanley, K. O. (2008). Picbreeder: Evolving pictures collaboratively online. In *Proceeding of the Twenty-sixth Annual CHI Conference on Human Factors in Computing Systems - CHI '08*. doi:10.1145/1357054.1357328.

Serafin, S. (2007). Modulation synthesis: ring modulation and amplitude modulation. Aalborg University Copenhagen. Retrieved January 9, 2017, from `http://media.aau.dk/~sts/ad/modulation.html`.

Stanley, K. O. (2015). NEAT software catalog. Retrieved January 8, 2017, from `http://eplex.cs.ucf.edu/neat_software/#NEAT`.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127. doi:10.1162/106365602320169811.

Stanley, K. O. and Miikkulainen, R. (2004). Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100. Retrieved January 12, 2017, from `https://www.jair.org/media/1338/live-1338-2278-jair.pdf`.

Veldhuizen, D. A. and Lamont, G. B. (2000). Multiobjective evolutionary algorithms: Analyzing the state-of-the-art. *Evolutionary computation*, 8(2):125–147. doi:10.1162/106365600568158.

Verfaille, V., Zolzer, U., and Arfib, D. (2006). Adaptive digital audio effects (a-DAFx): A new class of sound transformations. *IEEE Transactions on Audio, Speech and Language Processing IEEE Trans. Audio Speech Lang. Process.*, 14(5):1817–1831. doi:10.1109/tsa.2005.858531.

Walsh, R. (2008). Cabbage, a new GUI framework for Csound. Retrieved January 8, 2017, from `http://lac.linuxaudio.org/2008/download/papers/7.pdf`.

Werbos, P. J. (1982). Applications of advances in nonlinear sensitivity analysis. In *System modeling and optimization*, pages 762–770. Springer. doi:10.1007/bfb0006203.

Whiteson, S., Stone, P., Stanley, K. O., Miikkulainen, R., and Kohl, N. (2005). Automatic feature selection in neuroevolution. In *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation - GECCO '05*. doi:10.1145/1068009.1068210.

Whiteson, S. and T. van den Berg (2013). Critical factors in the performance of HyperNEAT. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages

759–766. Retrieved January 15, 2017, from `https://pdfs.semanticscholar.org/2a57/a47f53eebe41d5d5d11aa626e91ab65892c0.pdf`.

Ye, J. and Chen, S. (2014). SoundBreeder with MultiNEAT. Retrieved January 8, 2017, from `https://www.cs.swarthmore.edu/~meeden/cs81/s14/papers/AndyLucas.pdf`.

Appendix A

Open Source Toolkit

The toolkit that has been implemented in this project is open source and available at `https://github.com/iver56/cross-adaptive-audio`

This includes software for doing neuroevolution on sounds as well as the software for visualizing the experiments. The readme file includes:

- Detailed instructions for how to install the project and all dependencies on Windows and Ubuntu
- User manual with example commands for getting started

Appendix B

Neuroevolution Application Dependencies

Table B.1: Dependencies

Name	Description
Jinja2	A general purpose templating language. Useful for generating csd files that are fed into Csound.
six, futures	Makes the Python application compatible with both Python 2 and Python 3
whichcraft	A tool that can check which programs are installed on the computer. Used for starting Node.js correctly on both Ubuntu and Windows.
natsort	Natural sorting. Used for showing audio features in the correct order.
nose	A tool for running all the automated tests in the project
statistics	A package that can calculate various statistics, for example standard deviation, of data series.
numpy	A package for scientific computing. Used for computing gradients and euclidean distance.

Continued on next page

Table B.1 – continued from previous page

Name	Description
matplotlib	A 2D plotting library. Used for creating various plots for this report.
MultiNEAT	A portable neuroevolution Library written in C++. It has Python bindings.
Sonic Annotator with Vamp plugin LibXtract	Sonic annotator takes in a set of audio files, runs them through the specified vamp plugin and outputs the results. LibXtract is a library for audio feature extraction.
Aubio MFCC	A tool that takes in a single audio file and calculates and outputs MFCC coefficients for each frame
Essentia music extractors	A tool that takes in a single audio file, calculates a large set of audio features and writes the resulting data to a file
Csound	A sound and music computing system. Csound can be given a piece of Csound code that describes how to process audio, and then Csound processes the audio accordingly. In this project, Csound is used for A) applying audio effects with effect parameters that vary over time and B) calculating audio features.

Appendix C

JavaScript Libraries Used in Interactive Visualization Application

Table C.1: JavaScript libraries

Name	Description
NodeJS	Used for serving the application and pushing results to the application via websockets whenever new data becomes available
AngularJS	Application framework that makes it easy to build Single-Page Applications
Angular-material	User Interface (UI) Component framework. Makes it easy to add UI elements, such as buttons and sliders, that look nice and have great usability.
Color-brewer	Various sets of colors that are useful for visualizing data
Cubism	Time series visualization in the form of horizon charts, which reduce vertical space without losing resolution. This is useful when there are many variables to visualize simultaneously in a limited vertical space.
n3-line-chart	Used for line charts and histograms. Is nicely integrated with AngularJS and features some useful interactions. Depends on D3.js

Continued on next page

Table C.1 – continued from previous page

Name	Description
NVD3	Chart components for D3.js. Used for creating stacked area chart (species plot)
Debounce, limit	Used for throttling the refresh rate of computationally demanding actions
Sigma	Used for visualizing neural networks. Features zooming, panning and rotating.
Wavesurfer	Works as an audio player that also visualizes the waveform of the sound that is played. The user can click on the waveform to seek to that position.
jQuery	Makes it easier to do Document Object Model (DOM) manipulation

Appendix D

Paper Published in Proceedings of the 2nd AES Workshop on Intelligent Music Production

EVOLVING NEURAL NETWORKS FOR CROSS-ADAPTIVE AUDIO EFFECTS

Iver Jordal, Øyvind Brandtsegg and Gunnar Tufte

Norwegian University of Science and Technology
iver56@hotmail.com, oyvind.brandtsegg@ntnu.no, gunnart@idi.ntnu.no

ABSTRACT

In cross-adaptive audio effects, effect parameters are dynamically informed by features of sounds other than the sound that is processed by the effect. Cross-adaptive audio effects can be applied in a wide range of research fields, including live music performance and audio mastering. Toward a toolkit for signal interaction we present a system that can exploit dynamic audio parameters of signal sources to control effect parameters, and thereby dynamically process audio. The vast number of possible combinations of parameters makes empirical experimentation tedious and unfeasible for live performance. Artificial Intelligence (AI) methods, herein Genetic Algorithms (GAs) and Artificial Neural Networks (ANNs), are exploited to find parameters for useful signal interactions in cross-adaptive audio effects. An experimental approach is taken to combine GAs and ANNs to control the audio effect parameters of one sound (input) by extracting audio features from another audio source (target) as to process the input to sound as close to the target as possible. Such results are shown to be feasible by using evolved ANNs.

1. INTRODUCTION

The problem of extracting audio features for control of effect parameters is here defined to two problem domains; the extraction and selection of audio/signal features and the mapping of such features to control parameters for audio effects. That is a selection of features from the source audio stream, mapping process to control the effects that can manipulate the target audio stream toward a signal that include sought audio properties. The system presented is part of the development of a toolkit for experimentation with signal interaction [1]

To handle the mapping of features to effect parameters an evolved ANN is used. The chosen neural network is based on NeuroEvolution of Augmenting Topologies (NEAT) [2]. The architecture of the ANN in a NEAT approach allow evolution, e.g. a Genetic Algorithm [3], to define weights and topology of the network. Further, the training of the network is based on performance, i.e. fitness, instead of supervised learning, e.g. backpropagation [4].

The set of audio features for extraction is predefined, i.e. the evolved network exploits favorable features within the available feature set. The audio effect is also predefined.

To explore the possibility of exploiting AI methods toward cross-adaptive audio effects, a system for conducting and evaluating signal interaction experiments has been implemented. As a test case the system is set to make one sound similar to another by applying audio effects controlled by extracted audio features.

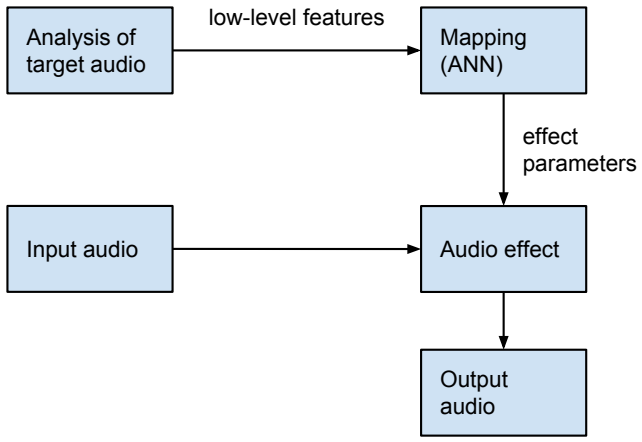

Figure 1: Cross-adaptive audio effect process with two audio streams: input audio and target audio

Figure 1 roughly illustrates the system setup. The low-level features are extracted audio features from the target sound. The features are mapped by the evolved ANN to effect parameters that are used to manipulate the input audio. The output audio is the result of effects applied to the input audio.

The system described produces large amounts of data in various forms, including audio features, effect parameters and output sounds. To handle the data for evaluation, an interactive visualization tool was made to make it easier to evaluate results and understand what the system is doing. The system and the visualization tool are open source and available on GitHub[1].

2. EXPERIMENTS AND RESULTS

The presented experiment's target goal was to make white noise sound like a drum loop with snare drum and bass drum. The selected and applied audio effect was distortion and resonant low-pass filter. The audio features used were spectral centroid and the first two Mel-Frequency Cepstral Coefficients (MFCC). Audio features were calculated for each

[1] https://github.com/iver56/cross-adaptive-audio

frame of 512 samples. The set of features in one frame is the feature vector for that frame. The fitness function used in the experiment was

$$1/(1+e) \tag{1}$$

where e is the average euclidean distance between feature vectors of the target sound and the corresponding feature vectors of the output sound. This means that fitness values are between 0 and 1. The population size was 20, the mutation rate was 0.25 and the crossover rate was 0.75. The experiment was run 20 times, with different Pseudo-Random Number Generator seed for each run. The fitness values were aggregated and are shown in Figure 2. Some of the sounds produced have been published in the project's blog[2].

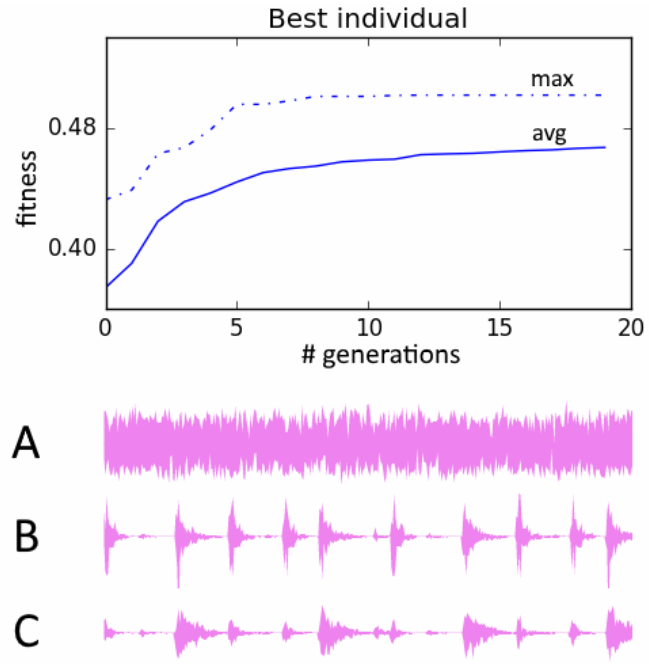

Figure 2: This plot shows the fitness (from expression 1) of the best individual in each generation. Below are waveforms of A) the input sound (white noise), B) the target sound (drum loop) and C) an output sound

3. FUTURE WORK

Future work may address the following:

- Conduct experiments with other audio effects. Let a genetic algorithm decide which audio effects to apply.
- Develop methods for dealing with long and complex sounds, such as music with many instruments.
- Make the system work on live audio streams, with pre-trained neural networks.
- Explore possible applications, such as mixing/mastering and novel sound effects.
- Experiment with other audio features. Use machine learning techniques to create high-level features.
- Implement the system on a Field-programmable gate array (FPGA) or other parallel computing environments for the sake of decreasing computational time. This may make it possible to train useful neural networks in seconds, making the system more flexible in live performances

4. CONCLUSION

Output sounds from the system demonstrate that it is possible to make white noise sound like a drum loop by applying a cross-adaptive audio effect. It also proves that NEAT can train a neural network to work as a musically interesting mapping from a set of audio features to a set of audio effect parameters. A comprehensive toolkit has been developed. It includes an interactive visualization tool that makes it easier to evaluate results and understand the neuroevolution process.

5. REFERENCES

[1] Ø. Brandtsegg, "A toolkit for experimentation with signal interaction," in *Proceedings of the 18th International Conference on Digital Audio Effects (DAFx-15)*, 2015, pp. 42–48.

[2] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evol. Comput.*, vol. 10, no. 2, pp. 99–127, Jun. 2002. http://dx.doi.org/10.1162/106365602320169811

[3] D. Goldberg, *GENETIC ALGORITHMS in search optimization & machine learning*. Addison Wesley, 1989.

[4] P. J. Werbos, *Applications of advances in nonlinear sensitivity analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 762–770. ISBN 978-3-540-39459-4. http://dx.doi.org/10.1007/BFb0006203

[2] http://crossadaptive.hf.ntnu.no/index.php/2016/06/27/evolving-neural-networks-for-cross-adaptive-audio-effects/