2019

Sonic Analysis for Machine Learning: Multi-Layer Perceptron Training using Spectrograms

Pearce-Davies, Samuel Louis

UNIVERSITY OF
PLYMOUTH

Sonic Analysis for Machine Learning: Multi-Layer Perceptron Training using Spectrograms

by

Samuel Pearce-Davies

A thesis submitted to the University of Plymouth
in partial fulfilment for the degree of

RESEARCH MASTERS

School of Humanities and Performing Arts

January 2019

Acknowledgements

Firstly, I would to express my immense gratitude to Professor Eduardo Miranda and Dr Alexis Kirke, my Director of Studies and Supervisor, respectively. The former enabled me to begin my journey into this branch of AI research in the first place, and has always been on hand since to provide suggestions and guidance. The latter introduced me to the DeepDream system and stoked my passion for this project, giving up a good deal of his time throughout its later phases to assist me with overcoming stumbling points and pursuing new vital avenues of inquiry. This project would not have progressed as far without their support.

Additionally, I would like to thank Stéphane Thunus, Satvik Venkatesh, Steve Crow, Edward Braund and Dillon Bastan, who between them were vital in providing me with musical samples and pointing me towards programming architectures without which my project would have stagnated or potentially failed entirely. Cheers guys, you're the best.

Finally of course, family. My parents – you have listened to me ramble about my work for countless hours, always telling me what you knew I needed to hear to carry on doing my best loving and supporting me unconditionally through the ups and downs. Thank you both, I love you always.

My wife Rebecca – marriage may be a full-time job, but it's more rewarding than any sum of money in the world. Without your daily support and love, I would not have got this far. Not close. I love you.

Declaration

At no time during the registration for the degree of Research Masters has the author been registered for any other University award without prior agreement of the Doctoral College Quality Sub-Committee.

Work submitted for this research degree at the University of Plymouth has not formed part of any other degree either at the University of Plymouth or at another establishment.

Word count of main body of thesis: 25,134

Signed: .....

Date: .....

Sonic Analysis for Machine Learning: Multi-Layer Perceptron Training using Spectrograms

Samuel Pearce-Davies

Abstract

This thesis presents efforts to lay the foundations for an Artificial-Intelligence musical compositional system conceived on similar principles to DeepDream, a revolutionary computer vision process. This theoretical system would be designed to engage in stylistic feature transfer between existing musical pieces, and eventually to compose original music either autonomously or in collaboration with human musicians and composers. In this thesis, construction of the analysis and feature recognition systems necessary for this long-term goal is achieved through the use of neural networks.

Originally, DeepDream came about as a way of visualising the weights inside neural network layers – matrices of variables containing the data that determines what information the network has learned – for better understanding of training and trouble-shooting of such networks that have been trained to classify images. This approach spawned an unexpectedly artistic process whereby feature recognition could be used to alter images in a dreamlike fashion, akin to seeing shapes in clouds.

The proposed musical version of this process involves analysing sound files and generating spectrograms – pictures of the sound that could be manipulated in much the

same ways as regular images. As described in this thesis, a sizeable bank of sound samples has been gathered – of individual musical notes from a selection of instruments – in pursuit of this application of the DeepDream architecture.

These samples are curated, edited and analysed to produce spectrograms that make up a dataset for neural network training. Using the Python programming language and its machine learning library ‘Scikit Learn’, a rudimentary deep learning system is constructed to be trained on the sample spectrograms and learn to classify them. Once this is complete, additional tests are performed to determine the validity and effectiveness of the approach.

Table of Contents

List of Images

Image 7: Graphical visualisations of 'relu' function, which limits all negative incoming values to 0, and its alternative 'leaky relu', which reduces all negative incoming value by 99%.
Page 43
Available at: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6
[accessed 10/1/19]

Image 8: Comparison of graphical visualisations of 'tanh' and 'softsign' functions.
Page 43
Available at: https://sefiks.com/2017/11/10/softsign-as-a-neural-networks-activation-function/
[accessed 10/1/19]

Image 9: Example two-dimensional spectrogram: visual representation of a waveform. Brightness of pixels indicate amplitude values of each frequency bin over time.
Page 47
Available at: https://stackoverflow.com/questions/9726640/spectrogram-example-in-matlab
[accessed 10/1/19]

Image 10: Spectrogram of 'anton' example sound file in Max/MSP, generated with Volker Böhm's patch.
Page 50

Image 11: Spectrogram of a processed recording of a Tibetan singing bowl, generated with Volker Böhm's patch.
Page 50

Image 12: Phase spectrogram of 'anton' sound file to accompany that in image 10.
Page 53

Image 13: Phase spectrogram of Tibetan singing bowl recording to accompany that in image 11.
Page 53

Image 14: Random noise altered and optimised to improve neural network feature recognition, creating a picture that represents the network's perception of bananas.
Page 55
Available at: https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html
[accessed 10/1/19]

Image 15: Several pictures generated in the same way as image 15, based on the network's understanding of dumbbells, all of which include some aspect of human bodybuilders.
Page 55

List of Tables

Introduction

With the widespread use of Artificial Intelligence technology across countless fields in the modern world – from the tailored spread of daily content we view on the internet, to patient prioritisation processes in hospitals, to automated phone systems with synthesised speech – our general lives are continually being shaped by developments in the world of AI.

With a rich history and such a broad array of applications, it is unsurprising that musicians have also developed systems to take advantage of various AI advancements – whether to assist in classification, categorisation or pattern recognition, or even to facilitate new approaches to musical composition entirely.

In this paper I will be presenting the journey I have taken over the past year in taking the first steps towards the development of such a compositional system. This has involved a considerable learning process, including getting to grips with an unfamiliar programming language, many new concepts and practical implementations of them.

As will become apparent, the final body of work I have completed this year has necessarily taken the form of something more of a foundation and proof-of-concept of a future compositional system, rather than the complete system itself. This has involved construction of a supervised AI classification system for sonic analysis and a bank of instrumental samples forming a bespoke training set for deep learning.

I will begin with a survey of relevant AI approaches, the history of some notable developments and schools of thought in the field, along with some discussion of the various terms in common usage and ending with some examples of practical applications.

Next, I will recount and explain the various steps I took towards definition of my final project, beginning with a practical implementation of a Multi-Layer Perceptron in Max/MSP, and continuing with discussions of my research into spectrogram analysis of sound files and the computer vision paradigm known as DeepDream, elucidating how these topics were transformative in helping my project begin to take shape.

Finally, I will describe the design and construction of my analysis system, the process of learning the programming language Python and the various machine learning libraries I made use of, and finally various rounds of testing I performed on the complete system, with comments indicating my perceptions of the results.

1. Survey of AI Approaches

‘Artificial Intelligence’. As technological umbrella terms go, AI has entered popular usage perhaps more than any other in recent years, whether in online articles evoking post-apocalyptic scenarios reminiscent of ‘Terminator’ or ‘The Matrix’ (Friend, 2018), videos of the latest advances in intelligent robotics (BostonDynamics, 2017), or simply comedic examples of AI systems producing imitations of popular media (Ogden, 2018). In addition, with well-known entrepreneurs such as Elon Musk frequently stating the importance of taking care in AI development (Holley, 2018), and even the UK Government’s 2017 Industrial Strategy paper listing one of its four ‘Grand Challenges’ as a focus on AI and Data Economy (GovUK, 2017), it would seem we truly are in the midst of the artificial intelligence revolution.

However, AI is not a new term by any stretch of the imagination. B Jack Copeland states that the earliest known recorded discussion of artificial intelligence took place in January 1952 (Copeland, 2004: 487), broadcast by BBC Radio and featuring none other than Alan Turing, considered by many to be to father of the computer age (Howard, 2017). The term AI itself was not used by Turing or any of the other participants however, rather they described the systems in question as ‘automatic calculating machines’ (Copeland, 2004: 494). The coining of the term artificial intelligence is attributed instead to John McCarthy, a computer scientist who co-authored a proposal for the 1956 Dartmouth Conference (McCarthy et al, 1955), a seminal event that has been described as giving birth to the field of AI in general (Living Internet, 2019).

In this chapter I will explore the development of AI research, specifically pertaining to 'deep learning' through neural networks and their derivatives – the breadth of the field as a whole being far too great to cover comprehensively. This will involve some discussion of the nested terminology of the field and how these terms often overlap, are often misused or may even be misnomers. I will also discuss some examples of their practical use, focusing where possible on those in a musical or sonic context.

1.1: Brief History of Neural Networks

We have briefly discussed the origins of the umbrella term AI, dating back to the 1950s. Still earlier examples of work in this field exist however, notably Walter Pitts and Warren McCulloch's 1943 paper 'A Logical Calculus of the Ideas Immanent in Nervous Activity' (McCulloch & Pitts, 1943), which attempted to consider neural processes in the human brain and nervous system from a computational standpoint. They describe this as a 'neural network', a ground-breaking term for a subset of AI research that is still in common usage more than three quarters of a century later, albeit frequently preceded by the word 'artificial', and often simply abbreviated as ANNs, which are at the foundation of the 'connectionist' branch of AI research.

ANNs were conceived as a way of representing how neurons in human brains may interact to learn from external stimuli, although how accurately they mimic the human learning process is questionable (Bhatia, 2018). They have proven excellent at pattern recognition, where they will learn to classify different groups of data during a 'training' process in which example data is pre-classified by humans for the network to learn from.

The most basic ANN algorithm involves forming an interconnected web of 'neurons', individual nodes that can be input data, perform a calculation regarding that data and its relationship to an internal 'weighting' within the neuron, and output the result. During the training process, the weights of the neurons will be updated depending on how correct or incorrect the network's predictions of how to classify incoming data are, and eventually will converge so that the network can correctly classify incoming data with high accuracy, at which point the weights stop being updated.

One of the first physical implementations of this framework was originally known as a 'Perceptron', developed by Frank Rosenblatt in 1957 (Markus, 2013), which deals with binary inputs and likewise outputs either a 0 or 1 depending on the balance of input values (Zeidenberg, 1990: 50).

This kind of 'single-layer' system was shown to be effective at recognition of simple linear patterns, however its effectiveness with more complex examples was challenged by Marvin Minsky and Seymour Papert with the publication of their 1969 book 'Perceptrons' (Minsky & Papert, 1969), in which they claimed neural network frameworks were incapable of learning more complex non-linear patterns such as a logical XOR gate.

Despite various inaccuracies in their book and unsubstantiated claims about the unfeasibility of 'multi-layer perceptrons', which Rosenblatt had proposed as the next step in neural network development (Olazaran, 1996: 626), Minsky and Papert's words caused many to begin losing faith in connectionism. Additionally, a 1972 report by James

Lighthill named ‘Artificial Intelligence: A General Report’ (Lighthill, 1972) presented a pessimistic view of the future possibilities of academic research in this field, the metaphorical nail in the coffin that led to a decline in funding and for the field to be seen as a dead-end for almost two decades (Olazaran, 1996: 613) – a period that has become known as the ‘AI Winter’ (The Royal Institution, 2017: ‘5:23’).

It was not until 1986, when David Rumelhart helped a technique known as backpropagation gain recognition (Samarasinghe, 2007: 117), that interest in neural network research was rekindled. Backpropagation is an algorithm for training the weights in multi-layer neural networks, where the ‘error’ – the amount by which the network was incorrect in its initial prediction – is subjected to a differential equation and propagated back through the layers in the network in sequence, an approach which easily overcame the concerns raised years earlier by Minsky and Papert, including the XOR gate. Later in this paper (section 2.1), I will describe my own construction of a multi-layer perceptron (which has since come to be often abbreviated to MLP) from the ground up in the program Max/MSP, which will cover the technical details more fully.

As a result of this breakthrough, neural networks jumped to the forefront of AI research once again, and today they and their derivatives dominate the field of ‘machine learning’ – as it has come to be called – in particular its offshoot: ‘deep learning’.

1.2 AI Terminology

As may now already be apparent, there is a multitude of terms involved in any lengthy discussion of AI. 'Artificial Intelligence' itself being an often unhelpfully broad and non-specific phrase, researchers and practitioners have developed several nested layers of terminology that help narrow down discussion of the various branches of AI research.

To the layman, discussion of the field often conjures up ideas of the kind of sentient AI featured in science-fiction as previously mentioned. This kind of system would properly be called an 'Artificial General Intelligence' (AGI), although it currently remains a completely theoretical prospect. For a system to be constructed that has the ability to display intelligent behaviour and interact with the human world in as wide a variety of ways as humans themselves – this is still well outside of our current abilities (if it is possible at all), although the prospect of such AGI and their capabilities are certainly an important consideration for AI researchers.

The well-developed field of 'AI safety' is dedicated to thought experiments and other pre-emptive efforts to prevent AI systems at all levels of causing harm or otherwise acting in undesirable ways. Robert Miles, a researcher at the University of Nottingham, runs a YouTube channel dedicated to exploring various AI safety concepts and demystifying current academic papers (Miles, 2019), and the content of these videos demonstrate the growing importance of this field as AI capabilities continue to expand.

But to return to the present, the very word 'intelligence' arguably borders on misnomer when describing some AI approaches. During the previously mentioned 'AI Winter', the

prevailing field was that of 'symbolism' or 'Good Old-Fashioned AI' (GOFAI), a term coined by John Haugeland in 1985 (Haugeland, 1985: 112). This approach pre-dated development of neural networks, and stems from an outdated view of the human brain as an elaborate computer, following a series of logic-based procedural rules to tackle problems. Although very successful in various applications, critics such as Bart Kosko derided this approach to intelligence as inflexible, stating:

> *"Computer scientists have built [it] on the belief that knowledge is rules and that you can write down rules in the black-and-white language of computers and symbolic logic. […] They went for rules and scrubbed all the grey out of them."*
>
> (Kosko, 1994: 159-160)

Approaching the problem of intelligence from the perspective of more organic, flexible or 'fuzzy' logic as it came to be known (Kosko, 1994: 67) is at the heart of the connectionist world of ANNs. It presupposes that the rules of intelligence and solutions to problems are not perfectly definable, but are necessarily 'fuzzy' like human minds themselves and will require a system to 'learn' rules, exceptions and intelligent strategies for itself.

'Machine Learning' as a broader concept however, includes far more than just ANNs and their derivatives. Any process that causes a computer to modify or adapt its actions to increase some pre-defined notion of 'accuracy' comes under the heading of machine learning (Marsland, 2009: 5). This applies to such varied techniques as decision trees – essentially a series of if-then-else statements that can be computed with rapid efficiency (Marsland, 2009: 133) – to Bayesian models, which involve inferring rules about systems that are then updated as more data becomes available, notably treating system parameters as random variables instead of fixed (yet unknown) values (Theodoridis 2015: 586). Many more 'tribes' of machine learning and AI in general as they are described by Carlos

Perez (Perez 2017: 10-14) exist, encompassing multiple additional techniques and design philosophies that have developed over the decades of AI research.

Still more subsets exist that categorise these machine learning approaches, however. Learning can be 'supervised' or 'unsupervised': the former being when a system is 'trained' on data that had been previously labelled by humans – as used by the perceptron and other modern derivatives – and the latter involving algorithms that learn to 'cluster' data points together based on its perceptions of their similarity, which in turn allows it to infer knowledge about the range of data available to it (Marsland, 2009: 195).

But what about the goal of such systems? Many are for 'classification': labelling of new data based on learned understanding of existing data (Mitchell, 1997: 54). Others are for 'regression': prediction of unknown values based on perceived rules of their function (Marsland, 2009: 8). Many other goals for machine learning systems exist, which may be more or less specialised depending on the context of their use.

These varied terms for the approaches, goals and philosophies behind machine learning techniques overlap significantly and can be applied at multiple different levels of discourse. Most relevant to the subject matter of this paper though, is 'deep learning'. In the widest possible sense, deep learning relates to any neural-network-like system that features multiple layers of neurons – the multi-layer perceptron being the first notable historical example. Deep learning is potentially the most powerful field of AI research today – deep neural nets have displayed superhuman pattern recognition abilities, classifying image databases with incredible levels of accuracy since 2009 (Schmidhuber,

2015: 86) It is from the field of deep learning some particular examples covered later in this survey will come.

Broad, often vague and overlapping but nevertheless informative, the array of AI-related terminology as I have briefly covered it here will be used throughout the remainder of this paper, as I progress from exploring existing AI systems to implementing one myself.

1.3 Symbolism & Deep Blue

Rule-based approaches to problem-solving predate the advent of the personal computer by more than a hundred years. Charles Babbage, an English mathematician born in the late 18th century theorised the design and creation of the 'Analytical Engine', a physical machine that could solve problems through programmable operations with conditional branches (Haugeland, 1985: 126). Although the machine was never constructed in Babbage's lifetime, the principles of its design were a landmark moment in the development of computer science as we know it today.

Symbolic AI techniques, or GOFAI, originally stemmed from this problem-solving ethic – working from the basis that any problem in the world could be solved if enough rules could be constructed regarding how that problem is dealt with, and that intelligence would naturally follow once enough rules were defined (Kosko, 1994: 159). The major flaw with symbolism is the brittleness of its approach – edge-cases will inevitably exist for more complex human problems, for which additional rules will need to be defined, until even more edge cases appear and so on (Perez, 2017: 12).

However, for problems with finite, measurable parameters, symbolic techniques can be immeasurably powerful. One such problem is the game of chess.

> *"In the heyday of old-fashioned artificial intelligence, chess was considered to be one of the pinnacles of human intellectual achievement. Surely, if a machine could play chess, everything else like solving world poverty and global warming etc would be trivial by comparison."* (The Royal Institution, 2017: '9:08')

In 1997, the IBM computer system named Deep Blue defeated the world chess champion Garry Kasparov in a six-game match, a landmark moment in the public perception of AI (IBM, 2019). Despite being heralded as the first 'intelligent machine', Deep Blue utilised a particularly advanced search algorithm that could consider 50 billion moves over the three minutes allotted for each chess move in such a tournament – a 'brute-force' approach which even one of its developers reportedly denied was evidence of artificial intelligence (Press, 2018).

The number of move possibilities in any game of chess is almost incomprehensibly large, yet GOFAI was able to 'solve' this problem more than twenty years ago and render human players of the game inferior. Such is the power of rule-based approaches in the right context.

Despite their flaws, rule-based AI approaches have proven to remain important, powerful and useful in various applications through to the modern day – in fact I myself made great use of them in my final Undergraduate project as I briefly discuss at the beginning of chapter 2.

1.4 AI in Algorithmic Composition

Musical applications of AI have been present for many centuries, at first following primarily symbolic approaches in the form of Algorithmic Composition. Beginning with systems for writing music that involved random chance through dice rolls (Cope, 1996: 2), since the dawn of the computer age techniques have grown in complexity and potential, and as with general AI research many overlapping branches have developed, with varied goals in mind and a plethora of terminology. Of particular consideration is whether systems are designed to work autonomously or in direct collaboration with human musicians and composers. Eduardo Miranda makes mention of the distinction between *algorithmic composition* software and *computer-aided composition* software as follows:

> *Whilst algorithmic composition software is programmed to generate music with a certain autonomy, […] computer-aided composition software serves as a tool to help the composer capture and organise ideas. There is a healthy tension between these two types of software in that most composers interested in composing music with computers tend to find a balance between these two extremes."* (Miranda, 2001: 9)

Whatever level of input human musicians/composers have on a system's output, some might argue that musical composition in general is fraught with issues and complications that arguably render aspects of computer composition software design fundamentally problematic. In his 2001 book 'Machine Musicianship', Robert Rowe describes the 'fleeting and unconscious' quality of much compositional work as being difficult to distil into sets of rules, parameters or training sets (Rowe, 2001: 202).

Additionally, Rowe brings up the problem of emotional response to music, which he describes as particularly problematic for algorithmic modelling because it seems 'so keenly personal and variable' even on multiple hearings, yet remains a fundamentally important consideration in any account of the listening experience (Rowe, 2001: 244).

However, there are many aspects of music composition that can be quantified and input into algorithmic systems – music theory on tonality, scales, melody, harmony and chord progressions, to name but a few. Additionally, more recent research into how music affects the emotions provides fertile ground for potential incorporation into future systems. In particular, David Huron's book 'Sweet Anticipation - Music and the Psychology of Expectation' (Huron, 2008) examines evolutionary responses related to successful and unsuccessful prediction of future events and how these can affect a listener's emotions.

> *"Whenever a stimulus evokes a fairly consistent psychological effect, it becomes possible to use the stimulus intentionally as a means for evoking that effect."*
> (Huron, 2008: 173)

The successful development of many algorithmic composition systems is proof of the ability of computers and AI to take advantage of the many quantifiable aspects of music-making. What follows is a brief exploration of a number of such systems.

David Cope, a composer and computer scientist from California, began development of an algorithmic system in the early 1980s called 'Experiments in Musical Intelligence' (EMI) (Cope, 2019). This system involved semantic analysis of the works of various classical composers, designed to identify and imitate stylistic elements of their music and finally outputting a score to be performed by humans (Cope, 1996: vii). Cope went on to

develop ALICE – Algorithmically Integrated Composing Environment, which in contrast to EMI is software designed to be interacted with by human composers. According to Cope, ALICE infers rules about composition from pre-selected databases of work, transforming its knowledge into new music:

> *"Such transformation imitates, in my opinion, some of the techniques human composers use when composing, whether by conscious intent or by intuition."* (Cope, 2000: 87)

More recently, Cope has expanded his work to focus on individual style development as opposed to pure imitation, through a system he has anthropomorphically named 'Emily Howell' (Humanity+, 2010). Curiously, Cope has augmented the algorithmic processes of EMI by programming Emily Howell to respond to feedback:

> *"The program produces something and I say yes or no, […] I've taught the program what my musical tastes are, but it's not music in the style of any of the styles – it's Emily's own style."* (Cheng, 2009)

This process of positive and negative feedback optimising future output arguably takes Emily Howell slightly outside the realm of pure symbolism – however, as this optimisation is still being initiated by human feedback itself rather than a learned perception of its own output, it remains disputable whether Emily Howell is truly a machine learning system.

In contrast to Cope's work, which seems primarily concerned with individuality, style development and results in sheet music to be performed by human musicians, AI startup 'Jukedeck' is an online system designed for general public use mainly by video creators and other non-musicians, to algorithmically generate unique pieces (Jukedeck, 2019). The

system allows users to set a number of parameters, from broad aspects like genre and length to more specific features like instrumentation and 'mood'. It also includes a 'climax' function, which is of particular interest due to David Huron's description of this effect as an excellent way to manipulate tension and thus affect the emotions (Huron, 2008: 322-326).

Jukedeck is distinct from Cope's work in that it not only produces fully-synthesised tracks at the user's command to download then-and-there, but is clearly more focused on replicating broad signifiers of particular genres in modern music, from folk pop to 'cinematic' orchestral music to drum & bass. The system has clearly been designed to output 'generic' music, especially since the primary goal for its use seems to be to create background music for internet videos, where more striking individual elements would be less desirable. Jukedeck has recently been acquired by TikTok, and as such their website-hosted system is currently unavailable (Butcher, 2019).

Similar to Jukedeck but also incorporating a recognisable philosophy on individuality and unique style as Cope's work is AIVA – Artificial Intelligence Virtual Artist. A Deep Learning system, AIVA is trained on a large dataset of classical music compositions, from which it can detect and learn patterns and other stylistic aspects of the music. Pierre Barreau, the project lead describes AIVA's compositional process as also involving the setting of multiple 'category labels' which superficially appear to be an expanded version of Jukedeck's parameters, such as tempo and 'mood' and also which 'epoch' a piece should sound like it was written in (TED, 2018: '2:05'). AIVA has since been expanded into a paid online service again similar to Jukedeck where users can generate their own tracks (AIVA, 2019 [no.1]).

When describing AIVA's output, Barreau brings up terms such as 'personalised' and 'individual', which speaks to the design team's clear intention to generate more than just the broadly appealing generic music of Jukedeck. Moving beyond the base function of generating unique pieces from a variety of set parameters (which could rely purely on pre-built algorithmic processes and the selected database), one of AIVA's most recent additions is a means by which generated pieces can be directly influenced by an uploaded MIDI track of the user's choice (AIVA, 2019 [no.2]). This ability of the system to learn from new data on the spot and as such create multi-layered pieces in collaboration with any human user is impressive, blurring the line between autonomous algorithmic composition and computer-aided composition software as previously mentioned.

The examples so far have all involved systems that generate complete pieces of music, taking as long as is necessary and only involving human input at specific stages of the process – whether to set parameters and influences before a piece is generated, or to provide positive or negative feedback afterwards, as with Cope's Emily Howell. But music does not exist solely within the studio or as recorded media, and as such many algorithmic systems are designed to generate music live, often continuously interacting with human musicians throughout a performance.

> *"When humans improvise together, players influence each other to fall in with certain kinds of textures, for example, or to adopt various melodic or rhythmic motives. The relationship is one of cooperation mixed with a degree of independence."* (Rowe, 2001: 287)

One noteworthy system that demonstrates this live-performance functionality is Al Biles's 'GenJam'. Short for 'Genetic Jammer' (as in a jazz 'jam session'), GenJam is an

example of a ‘Genetic Algorithm’, which is roughly based on Darwin’s theory of evolution and involves various ‘chromosomes’ of data that evolve over multiple generations through a simulated ‘natural selection’ to best fit a particular problem (Mallawaarachchi, 2017). Essentially, the chromosomes that are judged fittest in each generation will combine together in various ways to produce new chromosomes (which may also ‘mutate’ slightly depending on the system setup) which replace other less fit chromosomes.

Designed to produce live jazz music, GenJam reads a pre-written score detailing chord changes and piece structure, understanding that some sections are set (the ‘head’, for example) whereas others allow for improvisation, similar to how human musicians would learn a score. GenJam’s chromosomes roughly contain rhythmic pointers to particular scale degrees, defined by the pre-read chord progressions, from which many thousands of combinations of measures and phrases are possible. What remains to roughly understand GenJam’s construction is the means by which certain chromosomes are judged fitter than others.

As Al Biles himself explains, this is difficult because broadly speaking, musical fitness is inherently subjective and therefore in cases where a specially-trained AI system such as a neural network is unavailable, it requires a human ‘mentor’ to judge chromosomes manually. This leads to a ‘fitness bottleneck’ that acts as the limiting factor on chromosome population sizes and number of generations (Biles, 1994: 3.3). When listening to GenJam’s output, Biles provides positive or negative feedback in the form of specific keyboard strokes, from which GenJam alters chromosome fitness – a process reminiscent of Cope’s giving of feedback to Emily Howell.

Working with these requirements, Biles has been performing live with GenJam since the 1990s, and maintains that not only is the system 'creative', but that it makes him more creative through his collaboration with it (TEDx Talks, 2012: '14:11'). Along with the system's ability to produce jazz solos based on chromosomes, GenJam can 'listen' to Biles play a short improvised solo on the trumpet and respond, incorporating some melodic aspects of what it just 'heard' by mapping it to new chromosomes (TEDx Talks, 2012: '2:26'). Biles will then often respond to what GenJam produces, engaging in exactly the kind of cooperative yet semi-independent relationship mentioned by Rowe.

> *"My explicit goal is that GenJam not just be human-competitive – which I maintain it probably is – but that it be a good collaborator. […] It's not about the technology, it's about the music. […]The technology is not an end in itself. It should be driven by your experience."* (TEDx Talks, 2012: '13:59' '14:32' '15:12')

Just from these few examples, there is already a great deal of variety on display in program architecture, style & format of output music, as well as in the level of human interaction with each system. This has been a necessarily brief exploration of such systems, as countless additional examples of AI processes in algorithmic composition exist, many of which can be found in an excellent survey on the topic by Jose Fernández and Francisco Vico (Fernandez and Vico, 2013).

Aspects of my own work have been informed by my understanding of the different design philosophies and potential for sonic output evident across these examples. Harbouring an initial desire to create something along the lines of the seemingly more unique, individually-styled AI music of David Cope's systems, I have come to respect the importance of focusing on genre signifiers *a la* Jukedeck. Despite its output being more

superficially simplistic, there is far greater sonic potential in its use of instrumentation, structure and the ability to set and tweak parameters, as is evidenced by AIVA's similar yet obviously more complex architecture.

Likewise, while the idea of creating a mostly autonomous system that could be left composing music indefinitely remains undeniably attractive, my research into collaborative systems like GenJam has helped my understanding of how and why AI systems can be designed with interactivity in mind. Although I am less inclined towards live musical performance alongside such a system, the demonstrated cooperative relationship certainly has potential for implementation in a more iterative, studio-based system, which a musician/composer could interact with by responding to its output with music of their own, for instance. As is evident in later sections, I did not progress to a stage where these kinds of features were feasible to implement during this project, although I fully intend to continue this line of inquiry in future work.

1.5 Convolutional Neural Networks

Deep neural networks have proven to be a monumental development in AI research. One of the most popular platforms for testing neural network architectures is the ImageNet Large-Scale Visual Recognition Challenge, or ILSVRC (Image Net, 2017), a large database of images sourced from the internet, which a system must attempt to classify as accurately as possible. In 2014, the classification winner was a system called 'GoogLeNet', with an error rate of only 6.7% (Karpathy, 2014).

GoogLeNet is an example of a ‘convolutional neural network’ (CNN), a more recent deep learning system that handles datasets (primarily images) in an unusual way, making use of kernel convolutions, and is considered a state-of-the-art model for computer vision (Buda et al, 2018: 249).

Kernel convolutions are an image processing technique that enable edge detection, blurring, sharpening and embossing of images, among other visual effects. This involves a kernel (a small matrix) being passed over pixel groupings in an image sequentially, reading the pixel values and applying some form of convolution on them depending on the algorithm being used. This convolution will alter the pixel values in a specific way depending on their groupings and direct neighbours.

In a CNN, the first layer of neurons after the input layer each deal with a slightly different convolution of the input data, while the next layer will involve further convolutions of these convolutions, continuing through the layers into greater levels of abstraction each time (Computerphile, 2016: ‘8:13’). Eventually the network will have developed a means of classifying images with an extreme level of accuracy due to the high-order information it is able to extract about image construction through the various convolutions.

One particular benefit convolutional networks have over other deep learning systems is their ability to gain this level of information about large-scale datasets with relative ease – for example, a regular deep neural network analysing a 7 megapixel image pixel-by-pixel would need have one input neuron for each pixel, something that many systems would struggle to compute efficiently but for which a convolutional network only needs

as many neurons as there are convolutions being dealt with, regardless of the size of the image (Computerphile, 2016: ‘3:54’).

Since GoogLeNet’s success, several more ILSVRC teams have reportedly had even greater success with convolutional systems, attaining better than 5.1% error rates (Nielsen, 2018). It is clear that CNNs represent a significant milestone in AI research, and it remains to be seen how far they have yet to develop.

1.6 Generative Adversarial Networks

Neural networks – or rather machine learning techniques in general, are usually concerned with optimising results, or minimising error. While this goal is excellent for data classification and regression as previous discussed, what happens if generation of new data is desired?

A system that has learned to classify a dataset and is given the task of generating new data that fits its understanding of that dataset – this system will invariably produce a result that appears to be an average or ‘best fit’ for that dataset as a whole, something that produces as low an error as possible. This will tend to appear artificial, standing out from the original dataset immediately as a ‘fake’ (Computerphile, 2017).

‘Generative adversarial networks’ (GANs) are employed as a solution to this problem. They involve two networks running in tandem: a ‘discriminator’, which is essentially a classifier such as those we are already familiar with – and a ‘generator’, which in recent

systems is often a convolutional neural network that will produce images from random noise. These two systems are set up in opposition with each other, hence the 'adversarial' of the name.

The discriminator analyses an image from a dataset and produces a prediction of its label, as is normal for classifiers, changing its internal weights using whatever optimisation algorithm is desired (such as backpropagation) based on the error between its prediction and the actual label. The generator then produces an image from random noise for the discriminator to analyse, making a prediction of this image's validity as part of the dataset. Based on the amount of error generated from this analysis, the generator also updates its weights, but with the goal of increasing the discriminator's error rather than decreasing it, fooling it into thinking the generated images are part of the original dataset.

The systems are therefore trained in tandem, competing to maximise or minimise the error of the discriminator's predictions as they both gain a better understanding of the dataset rules and features. Eventually – in an ideal scenario – the generator will be able to produce images that conform to the rules of the original dataset so convincingly that the discriminator can no longer tell them apart. At this point the discriminator is discarded and the generator is used to create new images.

Some researchers have made use of GANs to not only create convincing new images, but to facilitate feature extraction and recombination in generated material. In particular, a system trained on a dataset of headshot photos has been able to produce particularly interesting results by using arithmetic of vectors within the 'representation space' (Radford et al, 2016: 8). For instance, taking a generated picture with features indicating a

smiling woman, subtracting a generated picture of a neutral woman and then adding a generated picture of a neutral man resulted in a final generated picture of a smiling man (see image 1).

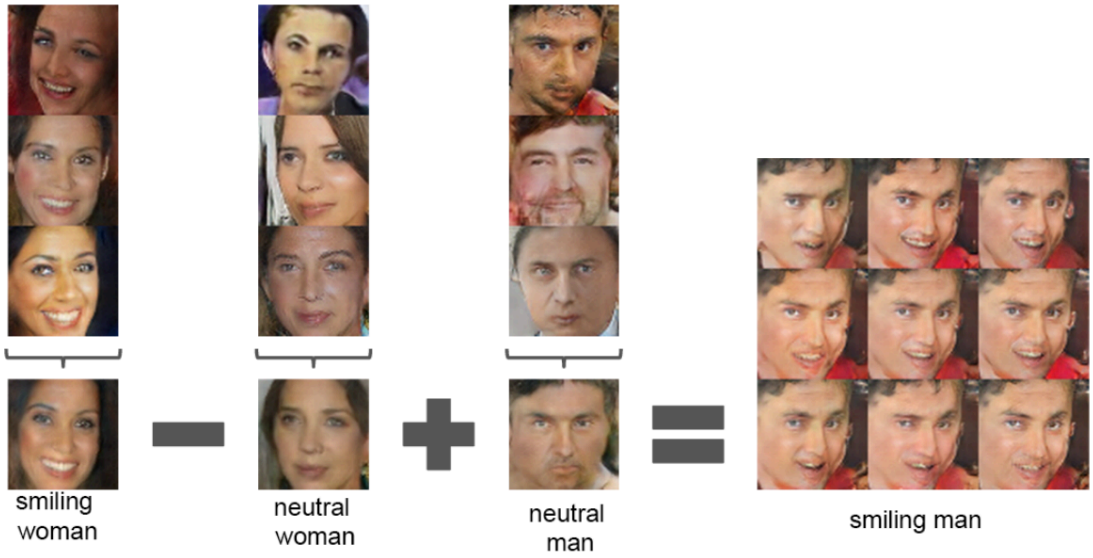

Image 1: Vector arithmetic in paper by Radford et al for abstraction of features and generation of new images.

These developments clearly indicate the potential of AI techniques to not only learn to classify data effectively, but to be able to create convincing new datasets as a result. Outside of purely statistic-based or computational benefits of deep learning, this now demonstrates a new field of creative, even artistic possibilities of AI.

1.7 WaveNet and NSynth

In September 2016, researchers at 'DeepMind' released a paper entitled 'WaveNet: A Generative Model for Raw Audio' (Van den Oord et al, 2016 [no.1]). WaveNet involves

stacks of convolutional layers that allow it to model raw audio waveforms with extremely good accuracy even at high sample resolution.

> *"At training time, the input sequences are real waveforms recorded from human speakers. After training, we can sample the network to generate synthetic utterances. At each step during sampling a value is drawn from the probability distribution computed by the network. This value is then fed back into the input and a new prediction for the next step is made. Building up samples one step at a time like this is computationally expensive, but we have found it essential for generating complex, realistic-sounding audio."* (Van den Oord et al, 2016 [no.2])

The results of this approach are undoubtedly ground-breaking. Training WaveNet on text-to-speech datasets produced synthetic speech samples that scored far better in blind tests than the current state-of-the-art systems (see image 2). Additionally, the researchers tested training the system on classical piano music, producing a number of fascinating free-form piano-like recordings (Van den Oord et al, 2016 [no.2]).

Although this latter musical output was clearly not the original intended use for WaveNet, it was not long before others were finding additional musical applications for the system. 'Magenta', an AI research project started by a team of Google researchers and engineers (Google AI, 2019) that focuses primarily on machine learning for creation of art and music.

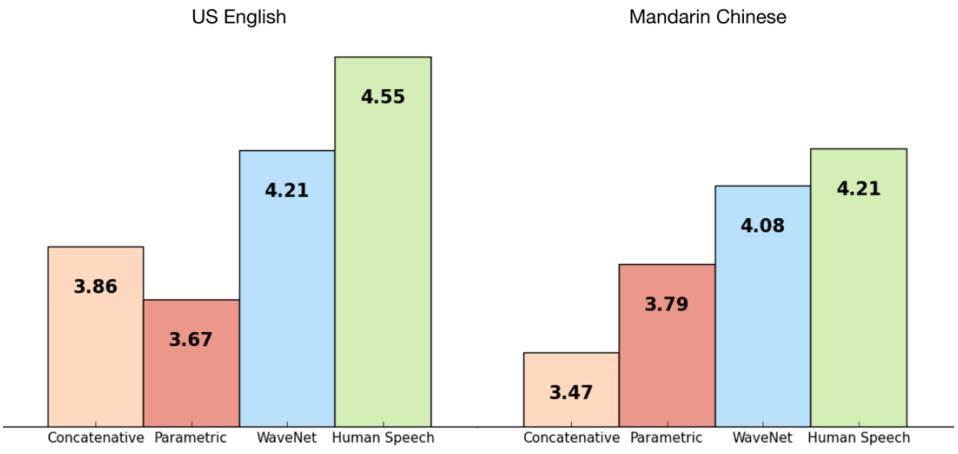

Image 2: Mean opinion score results from blind tests of different text-to-speech systems against human speech.

Magenta's 2017 paper 'Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders' (Engel et al, 2017) lays out the theory for their construction of 'NSynth', a 'neural synthesiser'. According to their website, NSynth allows its users to control the timbre and dynamics of synthesised sounds to create previously difficult or impossible effects (Magenta, 2017). They have achieved this by using the WaveNet autoencoders described in the paper to enable the system to represent the 'space' of instrument sounds it has been trained on, seemingly in a similar manner to the GAN headshot system previously described.

The practical outcome of this is that semantic aspects of different instrument sounds can be combined together in various ways, creating new unique synthesised samples. Listening to the examples of these combinations on their website, the relative rough-ness of the sounds indicates that either the examples given have not been crafted with a great deal of care, or that the combination process still needs further refinement.

Regardless, the novelty and unique nature of the approach taken here is substantial – this is one project it will be fascinating to watch as it continues to develop.

1.8 Physical Modelling of Machine Hearing

One potential non-standard way of approaching construction of perceptive AI systems is through physical modelling. A design philosophy more usually associated with sound synthesis, physical modelling involves analysing the real-life workings of a system and

attempting to distil these principles for application in synthetic imitations. A sonic example of this would be modelling the string vibrations and metal resonances of a piano in order to synthesise naturalistic piano notes.

Richard F. Lyon's book 'Human and Machine Hearing' (Lyon, 2017) deals with an unusual application of physical modelling principles, focusing on accurately modelling the process of how humans hear and perceive sounds, for translation into a machine learning context. Lyon describes this book as being for people who want to understand how our bodies process sound, and pursue design of a machine that can 'hear' like humans.

The early chapters of the book lay out various theories of hearing, including discussions of auditory psychology and pitch perception. This is followed by explorations of various digital systems vital to representing sound and hearing concepts, such as filters, resonant systems, Fourier analysis and spectrograms – the last two of which will be explored in more detail later in this paper, in the latter half of chapter 2.

Lyon continues with descriptions of potential ways of modelling the cochlea, inner and outer hair cells, all the way through to the auditory nervous system. Construction of a complete machine hearing system with such fidelity to the human hearing apparatus would be likely to involve an immense deal of additional research and testing, in fact the concepts explored in this book could easily fill an entire paper by themselves – I have only covered them superficially here as a way of demonstrating one of the additional possible approaches to AI system design that is available. In the future I anticipate the contents of this book will only become more relevant to my work.

1.9 Summary

In this chapter I have surveyed and presented a variety of existing techniques, approaches and design philosophies for AI research that have developed over the years, and which continue to be improved upon, spawning various derivatives. This survey has been necessarily incomplete, focused mainly on deep learning systems and those in musical contexts. The field itself is broad enough that any attempt at an exhaustive survey would be something of an unwinnable battle.

With a better understanding of some core approaches such as ANNs and particularly their offshoots CNNs and GANs, along with the knowledge gained of general AI terminology, the exploration of various concepts and topics in the remainder of this paper will be considerably more expedient.

2. Approaching the Project – Initial Experiments, Spectrograms & DeepDream

My interest in AI for musical composition was first sparked several years ago, during my Undergraduate course at Falmouth University. As part of my studies I had been introduced to the program Max/MSP (often abbreviated to simply Max), essentially a visual programming language for music. Having had some brief experience of basic coding in the past but having never found line-based languages particularly comfortable to work in, Max came as something of a revelation.

Authored by Miller Puckette in the late 1980s and named after Max Mathews of Bell Labs (Zicarelli, 2011), Max is now maintained and distributed by the company 'Cycling74' (Cycling74, 2019). Programming in Max involves placing various 'objects' on a blank canvas, connecting them together with 'patch cables' depending on the objects and their use. The resulting 'patches', as they are known, can be designed to perform a virtually limitless number of tasks, from synthesising electronic instruments through digital signal processing techniques, to implementation of algorithms for generating notes or parameters for such instruments or other software, to even producing and manipulating visuals which can in turn be linked to audio-production patches.

Having enough basic knowledge of coding axioms, mathematics and a good deal of musical theory understanding, I was immediately enamoured of the possibilities Max presented, both sonically and conceptually. Making use of it to some extent in almost every single one of my subsequent academic submissions at Falmouth, I quickly gained a

strong working knowledge of the software and my patch output became quite prolific. My website still serves as a repository for many of these patches – www.spearced.com

My final Undergraduate year came to be focused around construction of a Max-based system for musical composition, using what I was unaware at the time were rule-based AI techniques to create music. Having researched algorithmic composition and generative music for my Dissertation – both of which fall under the umbrella of symbolic AI approaches – I made use of the former to produce simplistic 'electronic dance music' (EDM) from a series of interconnected Max patches, including multiple electronic instruments I had synthesised myself from scratch.

Although this work was successful and I fully intend to continue improving and re-designing the systems in the future, my entry to the ResM at Plymouth led to my AI-related interests expanding somewhat. My course leader Professor Eduardo Miranda introduced me to the field of machine learning, pointing me in the direction of neural networks, which I had been vaguely aware of in the past but had considered well outside my ability due to my perception of the highly technical nature of their design and use. After a little cursory research and additional support from Prof. Miranda however, I decided to immerse myself in this field and see where it led me for the remainder of the ResM.

2.1 Testing a Neural Network Implementation in Max/MSP

At an early stage of my research into neural networks, I decided it would be a good idea to complement my learning with a practical application. Due to my familiarity and comfort with Max/MSP, I naturally wanted to continue my work in this program where possible. The potential for combining machine learning processes with my existing algorithmic/generative and synthesis work was an immediately appealing prospect, and as a result I decided to attempt a simple neural network implementation in Max as a way of testing the feasibility of more advanced ventures.

I approached this project with slight trepidation, having experienced some of Max's shortcomings first-hand in my previous work, namely its inability to deal with large amounts of information processing over short periods of time – something which is vital for neural network training. During construction of my final undergraduate project I had needed to find solutions to frequent slow-down and general poor patch performance when all elements were active. However, this had been when dealing primarily with numerous digital signal processing components – I had never had cause to implement levels of pure number-crunching on this scale before, and hoped Max would fare better when not working mainly with signal.

As I had very little practical knowledge of neural network construction at this stage, I began by searching the Max forums in search of any previous work that may have been undertaken in the field. I came across two existing machine learning resources – 'ml-lib' by Jamie Bullock (Bullock, 2014) and 'ml.star' by Benjamin D. Smith (Benson, 2017). However, both of these appeared to involve making use of pre-made machine learning

processes for use in music-making contexts, rather than dealing with axioms of their construction.

As resources seemed fairly limited as far as Max-based neural network construction was concerned, I decided it would be better to find other sources that detailed design principles with which I could go about building a patch of my own from scratch. I happened upon a pair of articles on Medium.com by Milo Spencer-Harper, which detail construction of a basic neural network (a Multi-Layer Perceptron, although it is never referred to as such in the articles) in Python, with example code (Spencer-Harper, 2015 [no.1] & [no.2]). At the time I was entirely unfamiliar with the Python language, but I was nevertheless able to glean enough information from the code and Spencer-Harper's excellent descriptions of the individual steps to be able to begin my work in Max. Partly due to the lack of information I had found on the Max forums, and partly as an additional learning reinforcement tool, I decided to record each step of the project as a series of YouTube video tutorials, detailing how I built each patch and explaining the methods and objectives for each step (Pearce-Davies, 2017).

2.1.1 Single Neuron Construction, Abstraction & Troubleshooting

The first step was to build a single 'artificial neuron', so to speak, that could be input a simple dataset and learn to recognise a pattern. The data provided in the article involved a series of three binary inputs, one of which (the first, in this case) would always be 'correct' and two others that were essentially random – they would show the correct value some times and not others (see image 3).

	Input			Output
Example 1	0	0	1	0
Example 2	1	1	1	1
Example 3	1	0	1	1
Example 4	0	1	1	0
New situation	1	0	0	**?**

Image 3: Training and test data for single artificial neuron. First column contains 'correct' input.

If I could successfully construct this single neuron and teach it to select – or at least make a very good guess at – the correct input, this would be a solid base from which to build the rest of the network. This would involve the process known as 'backpropagation' as briefly introduced earlier. For backpropagation, every input is assigned a 'weight', the balance of which is what will eventually enable the neuron to make an informed decision as to which input is correct. These weights are initially randomised, a quick process I implemented in Max by feeding several random number generators into a series of local variables, one for each input. The initialisation values would be anywhere between -1 and 1 for this patch.

The neuron reads the training dataset one example at a time, multiplying each input by its respective weight then summing the results. This final total is then normalised or 'squashed' so that it lies within a desired range, in this case between 0 and 1 as these are the potential output values. A particular mathematical function enables this 'squashing', known as 'sigmoid': [1/(1+e^(-x))], where 'x' is the sum of the weighted input values – and results in an s-shaped graph when visualised (see image 4). This normalised value is then taken and compared to the known output value for the current example.

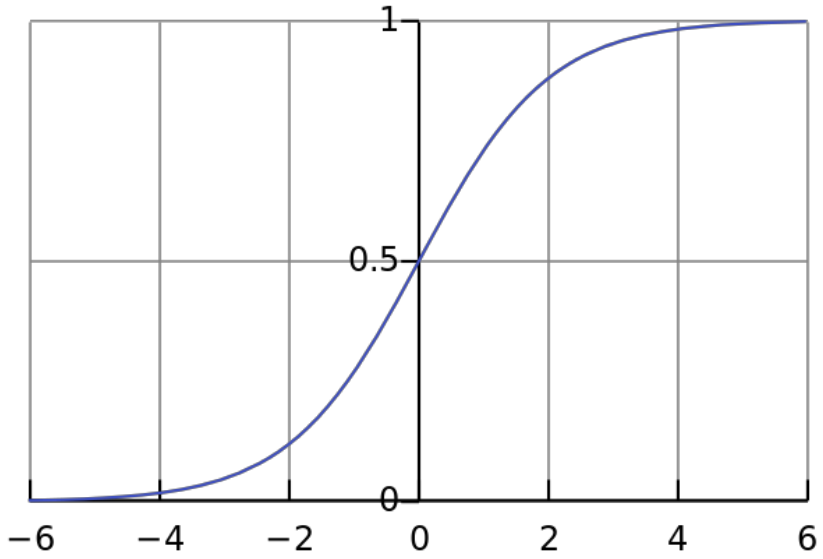

Image 4: Graphical visualisation of the 'sigmoid' function, which limits incoming values to between 0 and 1.

Due to the initial randomisation of the weights, the neuron's first 'guess' will likely be completely wrong, but this is where the backpropagation algorithm begins to display its strength. The error – the difference between the guess and the correct output – is taken and multiplied first by each separate input and then by the 'sigmoid curve gradient'. This value is literally the steepness of the line at any particular point on the sigmoid graph. As can be seen in image 4, the curve becomes exponentially less steep as it moves further away from 0 on the x-axis.

This means that as the x-value (our sum of weighted inputs) becomes significantly more positive or negative, the curve becomes smoother, signifying the algorithm becoming more 'confident' of a y-value (the overall output value) of 0 or 1, respectively. Multiplying by this sigmoid curve gradient has the effect of allowing the weight to shift more significantly when the x-value is close to 0 – indicating the neuron is unsure of this input's significance to the output value. The gradient is calculated with this formula:

[output * (1 – output)]. This method of solving weight distributions is understandably known as ‘gradient descent’.

The final value for each input (error multiplied by input and gradient) will either be a slightly positive or slightly negative number, and is added to the input’s respective weight, altering it by a small amount. This entire process is then repeated for each subsequent example in the training set, and then runs through the entire set again as many times as desired, shifting the weights by tiny fractions in each iteration. In the case of this single neuron, the four examples of the training dataset are run through 10,000 times each. After the training process is complete, the weights will have ideally converged in such a way as to enable the neuron to make an accurate prediction of the correct output value for a new, previously-unseen data example, shown as ‘new situation’ in image 3.

To give an example of this process in effect, we start the neuron patch off by randomly initialising the weights. In this example:

weight 1 = 0.68, weight 2 = -0.42 and weight 3 = 0.76.

After running training example 1, the updated weights are as follows:

Weight 1 = 0.68, weight 2 = -0.42 and weight 3 = 0.612

Obviously, only weight 3 has changed. This is due to inputs 1 and 2 both being 0 – multiplying the weight change amount by 0 in each case. After running example 2, all three weights have changed due to all inputs being 1. The updated weight values are:

weight 1 = 0.741, weight 2 = -0.359 and weight 3 = 0.673.

After example 3:

weight 1 = 0.772, weight 2 = -0.359 and weight 3 = 0.704.

And after example 4:

weight 1 = 0.772, weight 2 = -0.501 and weight 3 = 0.562.

After the full training cycle of 10,000 iterations, the weights are as follows:

weight 1 = 9.674, weight 2 = -0.209 and weight 3 = -4.629

These trained weight values are almost identical to those shown at this stage in the Python article, which is encouraging. If we run the 'new situation': [1, 0, 0], the patch returns an output prediction of 0.999937, which is an extremely accurate result. Likewise, when considering a different test situation: [0, 1, 1], the patch returns 0.007859, which is also extremely close. However, when considering a third test situation: [0, 1, 0], the patch returns 0.447954. This is closer to 0 (the correct output) than 1, however not by much. For some reason the neuron appears far less confident in its prediction.

Looking at the weights, the reason for this result is fairly plain: weights 1 and 3 have achieved strong positive and negative skewing, respectively, but weight 2 is only lightly negatively skewed. An explanation for this problem lies in the previous mention of the weight alterations being multiplied by the inputs – particularly when an input is 0, meaning that depending on the training data a weight may not actually be altered as often as we may wish. Additionally, when calculating its prediction, the neuron is multiplying each weight by the input, so when both inputs 1 and 3 are 0, the strong positive and

negative skewing of each respective weight is disregarded, leaving only the more uncertain skewing of weight 2 to produce the overall output value.

The fact that even after 10,000 training iterations weight 2 has failed to reach a strong negative value (as it should for this example) perhaps indicates the sub-optimal nature of the provided training data. Spencer-Harper makes no mention of this failing in his articles, and it is unclear whether he tested the system rigorously enough to encounter it at all.

In the days immediately following the upload of my first video tutorial, I received many comments from viewers both on YouTube and Facebook, giving feedback on the relative merits of the approach I had followed, suggesting improvements or alterations for various parts of the patch, and drawing my attention to the aforementioned training problems. One of my priorities with the second video, then, was to find solutions to the training issues and address some of the more salient questions and suggestions put to me.

My main challenge in this second section of the project was to take the neuron framework I had constructed and abstract it, so that it could be loaded in Max as a standalone object. Part of Max's music-based functionality includes methods of achieving polyphony for programmed instruments, i.e. allowing more than one note to be played at the same time. This involves use of the 'poly~' object, which I repurposed for the neuron abstraction, allowing a dynamic number of inputs and weights in the place of musical 'voices'. This enables the framework to be trained from different datasets without having to fundamentally re-write the patch each time. The principles of its construction is otherwise

identical to the first patch, although parts were occasionally altered for brevity and tidied up in places, implementing some of the feedback I received from viewers.

Once this abstraction had been successfully built, I immediately tested it on the previous dataset to ensure it still worked as intended. The training was indeed successful, encountering an identical problem with the training of weight 2 as before. To combat this failing, I added two additional examples to the training data:

Example 5: [0, 1, 0] [0]

Example 6: [1, 0, 0] [1]

Training the neuron on the expanded training set, the final weights were as follows:

weight 1 = 12.801746, weight 2 = -4.218397 and weight 3 = -4.218072

It is immediately apparent that these weights have been trained far more successfully than with the previous dataset. Any new situation the system now considers will result in an extremely accurate prediction across the board. To test this assertion, I constructed a new dataset of a larger size than the previous one, with five inputs instead of three. For this dataset, I decided that the third input would be ‘correct’. The values of this dataset were as follows:

Example 1: [1, 0, 1, 1, 1] [1]

Example 2: [1, 0, 0, 1, 0] [0]

Example 3: [0, 1, 1, 1, 0] [1]

Example 4: [1, 1, 0, 1, 0] [0]

Example 5: [1, 0, 1, 1, 1] [1]

Example 6: [1, 0, 0, 1, 1] [0]

Example 7: [0, 1, 0, 1, 0] [0]

In addition to writing this dataset, I increased the number of training iterations from 10,000 to 20,000 in an attempt to account for the perceived additional 'difficulty' of the problem due to its higher number of inputs. The abstracted neuron was able to learn very successfully from this training data, and produced extremely accurate output predictions for multiple new situations I put to it post-training. This testing demonstrated that the neuron as I had constructed it was well able to learn from data sets of varying sizes, as long as they follow the same rules.

2.1.2 Building the Network & Discovering Limitations

The single neuron having successfully proven capable of learning the 'correct' input from a list, the next step in my development of a functional neural network involved conquering a challenge that dealt with inputs and outputs with something more complex than a simple 1:1 ratio. As Milo Spencer-Harper detailed in the follow-up to his first article (Spencer-Harper, 2015 [no. 2]), this involves expanding the code to deal with a logical XOR gate, perhaps as a direct reference to the problem listed in 'Perceptrons' as supposed proof of neural networks' failings. This XOR gate, still working with binary inputs, will output a 1 when its two inputs are different, but a 0 when they are the same. The new training dataset (see image 5) includes these two inputs, plus a third 'incorrect'

input of no consequence other than to provide the additional challenge of the system needing to learn to ignore it.

As the XOR gate is an example of a non-linear pattern, which requires the system to learn to recognise relationships between inputs rather than how each input relates directly to the output. A slight problem was that the second article featured far less explicit instructions on the principles of the system's design, instead after some brief preamble it simply lists the complete code with some annotations. Therefore I was forced to delve into the Python code directly to work out the steps for myself.

	Input			Output
Example 1	0	0	1	0
Example 2	0	1	1	1
Example 3	1	0	1	1
Example 4	0	1	0	1
Example 5	1	0	0	1
Example 6	1	1	1	0
Example 7	0	0	0	0
New situation	1	1	0	?

Image 5: Updated training data and new situation for neural network. First two inputs form an XOR gate, third input is meaningless.

In a modern neural network, there are typically two different types of layers featuring groups of neurons, known as 'hidden' and 'output'. An output layer fulfils the same function as our single neuron, with its neurons taking the previous layer as their inputs instead of the training data directly. The neurons in a hidden layer will read the data in the same way as our initial neuron – the function of the different layers being that the weights of hidden neurons are altered by a weighted error from the subsequent layer instead of the

overall error from the final output. This final error is calculated first and used to update the weights of the output neurons using the previous formula, then propagated back through each hidden layer in turn to update their weights with respect to the individual output weights (hence the term ‘backpropagation’).

The practical upshot of this is that, while the first hidden layer will be dealing with the individual inputs from the training data, the output layer will be dealing with the results from the hidden layer/s and be able to infer information about the relationships between the original dataset inputs. The number of hidden layers used is dependent on the complexity of the problem, although there will only be one output layer in the majority of cases. In the XOR gate example from the article, a single hidden layer of four neurons is used, feeding into an output layer of one neuron, along with the number of training iterations being increased to 60,000.

The third leg of my work in Max (and third video tutorial) therefore involved altering my neuron abstraction, resulting in two derivatives: standalone hidden and output neurons that could be connected together to simulate the layers dealt with in the code. Although the example in the article produced correct results when considering the new situation as displayed in image 5, I found that my updated Max patch was frequently achieving the wrong results – less than half of the time but still far more often than desirable. Attempting to troubleshoot the problem, I increased the number of hidden neurons to eight instead of four, which increased the training time slightly but resulted in the system achieving correct results every time I tested it from then onwards.

The exact explanation for this problem is unclear without an exhaustive exploration of the changing weights in the failed instances (which would likely take an unfeasibly long time), and once again I do not know if the Python code may have ever experienced similar issues that Spencer-Harper simply didn't encounter when using it, or whether this indicates a fundamental failing in Max's ability to process large amounts of numbers, as I had feared when going into the project. I contacted Spencer-Harper to raise the question, but unfortunately never received a response to confirm this either way.

The working theory I had at the time (and have not since proved or disproved) to account for this failing relates to the way neural network data can be visualised, particularly the process of gradient descent as previously mentioned. Conceptualise the process of ascertaining ideal weight distributions as navigating a multi-dimensional mountain-range, with various peaks and valleys, in which the system is attempting to find the lowest point – representing the smallest output error.

The instances where my Max patch had been unsuccessful in finding a successful weight distribution could be likened to the system finding itself in the lowest point of a particular valley, unaware that several hills over there was an even lower point. Increasing the number of hidden neurons is analogous to providing the system with additional 'vantage points' from which to view the entire range and better locate the lowest overall point without getting stuck. I remain unsure whether this analogy is entirely accurate, but it still serves as an understandable explanation for the problem I was facing at this time.

Having successfully dealt with this issue by increasing the number of hidden neurons, there was one final step before my work in Max would be complete. The neural network

patch as it currently existed was rather labour-intensive to set up, requiring the neurons in each layer to be manually connected in specific configurations. In a situation where more than one hidden layer was required, this amount of set-up time would likely become unreasonable, therefore the subject of my last video was abstracting the layers themselves, building hidden and output layers as their own standalone objects.

This additional level of abstraction required more use of 'poly~', allowing each layer to be dynamically loaded with a desired number of neurons, each of which would also be loaded with a dynamic number of inputs as before. One significant addition was a general 'activation function' in the place of the simple sigmoid I had been using up until this point. The reason for this addition was, while the sigmoid (known as 'logistic') was used to restrict values between 0 and 1 for the purposes of this example, these bounds are not always desirable for neural network training.

According to various sources, the two most popular activation functions other than sigmoid are:

Hyperbolic Tangent (tanh): restricts values to between -1 and 1 with a similar s-shaped curve to sigmoid (see image 6).

Rectified Linear Unit (ReLu): positive values remain the same, but negative values all become 0. (see image 7)

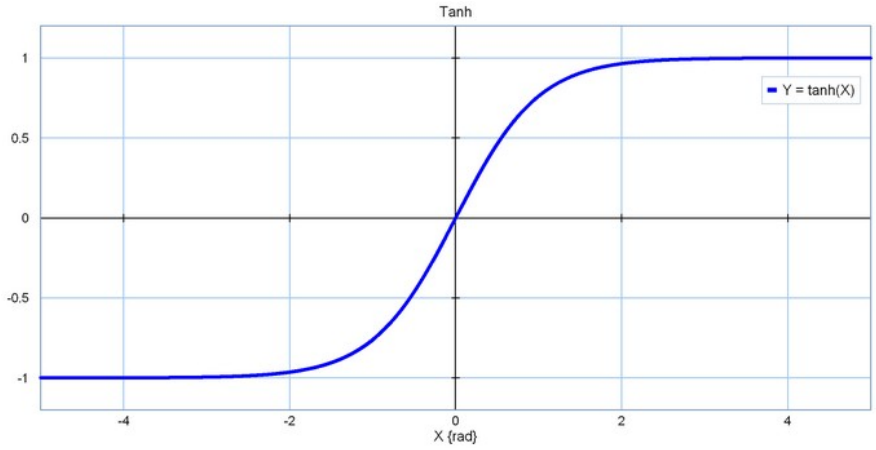

Image 6: Graphical visualisation of 'tanh' function, which limits incoming values to between -1 and 1.

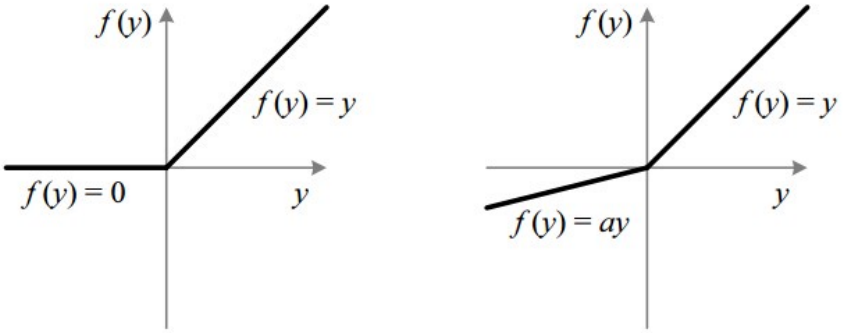

Image 7: Graphical visualisations of 'relu' function, which limits all negative incoming values to 0, and its alternative 'leaky relu', which reduces all negative incoming value by 99%.

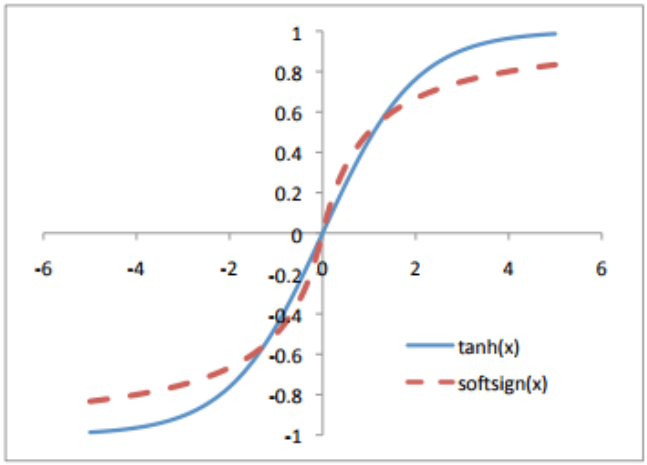

Image 8: Comparison of graphical visualisations of 'tanh' and 'softsign' functions.

One apparent problem with ReLu is that it can sometimes result in ‘dead neurons’, where weight updates occasionally cause the neuron to no longer interact with the overall system in any meaningful way (Singh Walia, 2017). A modification to ReLu was developed as a way of combating this, called ‘Leaky ReLu’, which rather than changing all negative values to 0, instead reduces them to 1% of their size (see image 7).

ReLu and its Leaky variant are by all accounts the optimal activation functions for most modern training datasets (Raval, 2017), but one particular academic paper posits a different function, known as ‘softsign’, as being useful in other situations (Bergstra et al, 2009). Softsign is similar to tanh in that they both restrict values to between -1 and 1, but both with slightly different curve shapes. A comparison between the two can be seen in image 8.

For my Max work, I decided to implement all of these functions to provide a good deal of flexibility of future use, although even more activation functions exist. With this final addition, I constructed the network once again with the newly abstracted layer objects and implemented the XOR gate training dataset, achieving successful results as before.

However, my testing had revealed an issue which seemed like it may prove fatal to my continued work in Max and implementation of more complex neural network setups – that of training time. Even when reducing the number of training iterations for the XOR gate to 20,000 (which still produced very good results), I recorded a delay of around 20 seconds each time the system was trained, while it processed the information.

While this may seem negligible, in the case of a neural network such as one I hoped to construct in the future – complex enough to analyse images or soundwaves that would likely feature hundreds if not thousands of individual neurons across many layers, along with a vastly increased number of training examples and iterations, it was plainly apparent that the patches as I had built them would be likely to take a prohibitively long time completing the training process.

Despite this realisation, I remained optimistic about the work I had completed in Max. The design itself was sound, so if the training process itself were to be completed in Python or another implementation, the final weight values could be sent to a Max-based system, for use in controlling a compositional system, for example. Another option would be to explore implementation of the patches using 'Jitter' or 'gen~'.

The former is a subset of Max objects and processes primarily designed for image creation and manipulation through matrices, which could serve as a more computationally-efficient medium for the training process. Gen~ however, is a relatively new function of Max which enables users to construct far more efficient patches with similar performance to line-based programming languages. While I had no knowledge of either Jitter or gen~, I believed (and still do, despite my project progressing in other directions since this work) that they may prove to be solutions to the problems my patches experienced.

Additionally, a contact I had made through correspondence regarding my video tutorials named Jazer Giles used the patches I had built as the basis for his own XOR gate neural network patch in 'jit.gen', an alternative version of gen~ focused on manipulating

matrices as in traditional Jitter. According to him, this patch encounters very little delay during the training process, although I remain too unfamiliar with the processes and syntax used to be able to make any judgements as to its use in more complex situations. Giles's patch is available from his website (Giles, 2018).

Overall, this initial foray into neural networks using Max/MSP served not only as a way of teaching myself fundamental principles of neural network construction in a familiar practical environment, but also – through my production of the video tutorials, which achieved some fair level of success on YouTube and social media – enabled me to make contact with various computer-music practitioners online, along with attracting the attention of Cycling74 (the company that owns and distributes Max), who contacted me to express their gratitude for my work in presenting and explaining the concepts to other Max practitioners.

Finally, as we will come to see later, one of the contacts I made through this process in particular turned out to be absolutely invaluable to the success of my final ResM project.

2.2 Spectrogram Analysis of Sound Files

As my initial work in Max/MSP had led to something of a dead-end, I continued my research into AI in general rather than attempt any further practical applications for the time being. One particular area of interest to me (that I thought at the time was almost entirely tangential to my AI-focused work) was visual analysis of sound files, to produce spectrograms.

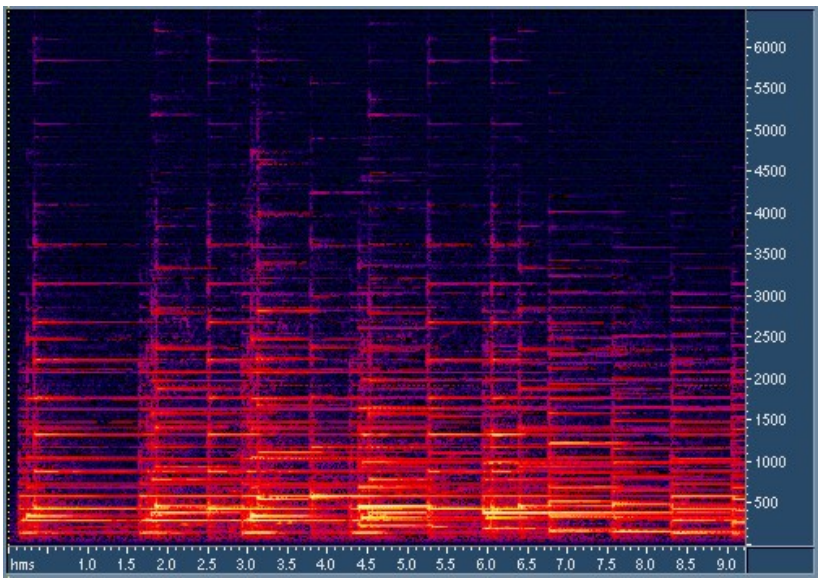

Image 9: Example two-dimensional spectrogram: visual representation of a waveform. Brightness of pixels indicate amplitude values of each frequency bin over time.

Spectrograms are typically two-dimensional images (although they can be – and often are for particular sonic analysis techniques – represented on a three-dimensional plot, in which case they may be referred to as a 'waterfall') that display the audible frequency spectrum on the y-axis and time on the x-axis. The brightness of a pixel indicates the amplitude of that particular frequency (or frequency 'bin' – to be described shortly) at that time in the sound file (see image 9).

Visual analysis of sonic data in this way can be extremely useful for understanding the construction of many sounds, as the changing overtones can be seen more easily, aiding in recreation or imitation through additive synthesis. Another use is for vocal synthesis, as the particular 'formants' of speech – groups of frequencies that imply different vowel sounds, among other information – (Wood, 2005) can be analysed and applied to help synthesised vocals appear more organic.

While there are several techniques for generating spectrograms (including banks of band-pass filters and optical spectrometers), one in particular that I was interested in was through making use of the 'Fourier Transform'. Joseph Fourier was a French mathematician who is often best remembered for his work on heat diffusion in the early 19th century. As part of this, he described how a continuous waveform can be expressed as a sum of individual trigonometric functions (Grattan-Guinness. 2008: 755), which for our purposes as sonic practitioners equates to a series of sine waves of different frequencies and varying amplitudes & phases.

The long and the short of this is that any possible sound can be produced if the artist is able to generate a large number of sine waves set to frequencies across the audible spectrum, and has access to enough information about how their amplitudes and phases change over time. For most practical purposes however, manual synthesis of complex sounds in this way is simply unfeasible. As described by Peter Manning, simulating just the initial attack (or transient) of a trumpet sample required the use of at least sixty sine wave generators, each of which had been individually crafted in terms of frequency and amplitude (Manning 2013: 432).

Despite these difficulties, Fourier's work has been used to deconstruct existing waveforms into their frequency components, using the technique known as the Fourier Transform. While extremely useful for computer music, this process is not without its own problems. The frequency spectrum is broken up into a series of chunks, known as 'bins', rather than every single hertz being analysed equally. In the same way, the sound will be broken up into equal groups of samples over time – 'frames' – that will be

analysed as short stationary moments of unchanging sound, typically in the milliseconds to avoid losing too much detail of the larger changing sound.

The mathematical constraints of the Fourier Transform mean that the higher the frequency resolution of the analysis, the smaller the time resolution, and vice-versa. Essentially, the more frequency bins you deal with, the larger each frame of stationary sound becomes – causing the changes in the sound over time to become less frequent and therefore less accurate. Likewise, the shorter the frames you take of the sound, the larger the frequency bins become – leading to frequency information becoming more blurred.

Various improvements to the basic Fourier Transform have been developed, for instance making use of a technique known as 'windowing', which involves overlapping the frames and allowing greater frequency resolution without such a noticeable loss in time resolution. This led to the development of the 'Short-Time Fourier Transform', which is the technique typically used for generation of spectrograms (Collins, 2010: 81).

My interest in spectrograms led to a conversation with a member of a Max/MSP discussion group on Facebook named Dillon Bastan, who referred me to a post on the official Max forums where a user named Volker Böhm had shared a spectrogram-generating patch of his some time previously (Böhm, 2013). This open-source patch was designed to host an audio file as a matrix in 'Jitter' (a primarily visual-related subset of Max processes as previously mentioned), subject it to a windowed Fourier Transform and output the resulting data as a two-dimensional image, a classic implementation of spectrogram analysis (see images 10 and 11).

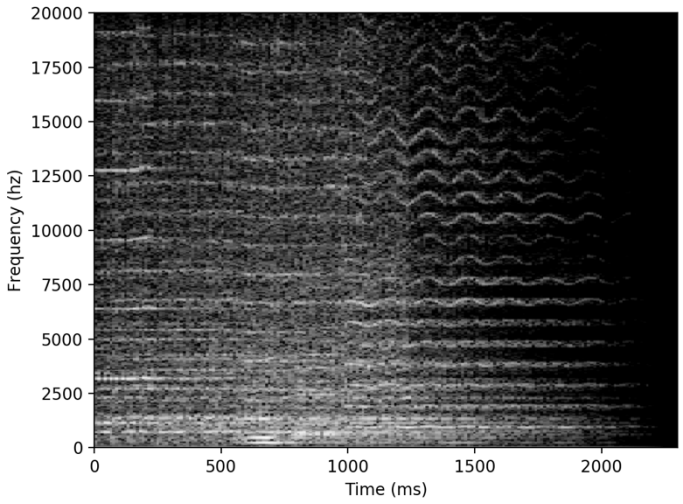

Image 10: Spectrogram of 'anton' example sound file in Max/MSP, generated with Volker Böhm's patch.

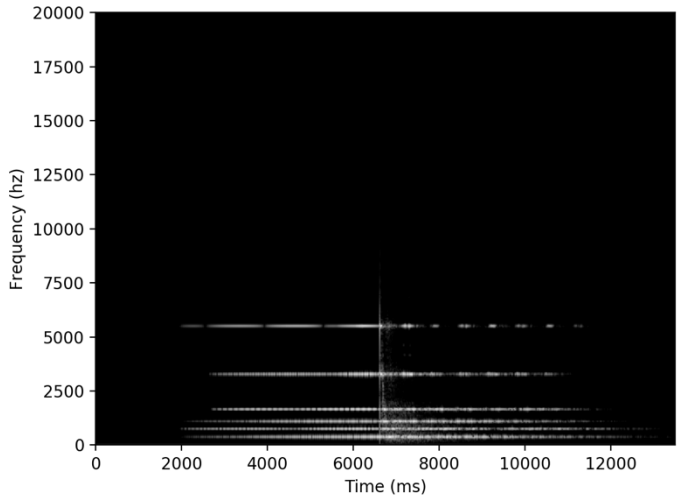

Image 11: Spectrogram of a processed recording of a Tibetan singing bowl, generated with Volker Böhm's patch.

Perhaps due to now having direct access to spectrogram generation in a program I was familiar with, I began to see growing possibilities relating to my research on artificial intelligence. Despite my lack of detailed knowledge of Jitter syntax or the exact practical

implementation of a Fourier Transform as used in the patch, I understood that the sonic building blocks of any sound, note or musical composition were now laid bare to me in pure number form, simple data hosted in the Jitter matrices. But this was not completely true, as I quickly came to realise.

As mentioned above, spectrograms typically display the amplitude values of each frequency bin over time, as this is usually the only desirable sonic component for visual analysis. But the frequency bins will also have a phase component, which although typically less visually coherent and therefore of less value to the human eye, is nevertheless vital for the construction of sound. If the phase values of an existing sound were altered without an understanding of their relationship to the amplitude values, the fundamental characteristics of the sound could potentially be changed entirely, leading to sonic artefacts and other unwanted distortions.

By attempting to unpick the patch as much as my knowledge would allow, I was able to make the necessary alterations so that the phase could now be displayed as a separate image – a 'phase-spectrogram', if you will. I achieved this by locating the point at which the data is read immediately following the Fourier Transform, and found that the phase component was currently being discarded. This simply required re-directing the data into a new set of Jitter processing objects and submitting it to an almost identical visualisation process, ending up with the final desired image (see images 12 and 13).

With both amplitude and phase values now readily available, I knew it could be possible to recreate a desired sound by inverting the Fourier Transform and saving the resultant data in an audio buffer within Max, which could then be saved and exported like any

audio file. This process – known as ‘resynthesis’ – would hardly be straightforward, however. Due to the processing of the data for visual representation, it was not in the correct form to be subjected to an inverse Fourier Transform, nor did I have the knowledge of how such a process would be programmed using Jitter objects.

But I was not overly concerned, as my thoughts were now fixed on the idea of training a neural network-based system on spectrogram analysis data generated with this patch, beginning just with the amplitude data for simplicity. What, I wondered, would happen if a generative machine learning algorithm was able to create new, unique sets of amplitude and phase data for resynthesis, based on sounds it had been trained to recognise and classify? What would that sound like?

The answers to these questions were tantalising, despite the practical work needed to answer them being somewhat out of reach. I knew that I had finally reached the area of study that would make up the bulk of my final ResM project, but there was one more important piece of the puzzle that was yet to fall into place, and which would serve as the catalyst not just for this project, but for potential future years of research.

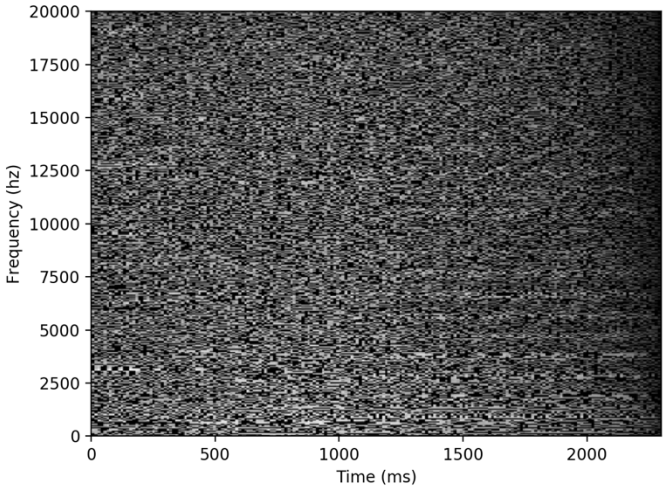

Image 12: Phase spectrogram of ‘anton’ sound file to accompany that in image 10.

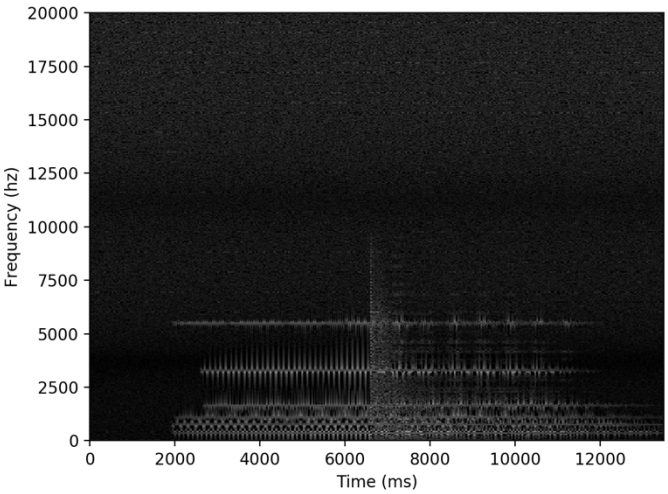

Image 13: Phase spectrogram of Tibetan singing bowl recording to accompany that in image 11.

2.3 DeepDream & Definition of the Final Project

In June 2015, three software engineers working at Google released a blog post on the official Google AI Blog titled 'Inceptionism: Going Deeper into Neural Networks' (Mordvintsev et al, 2015 [no. 1]), describing a technique for visualising the way neural networks perceive different levels of information in a dataset. In this post they begin with a cursory explanation of the workings of ANNs, describing the progression of information from layer to layer within a network, focusing on networks that deal with image classification and that include many layers – specifically using a successful image classification network called 'Inception' for their tests. Within this, they draw particular attention to the different layers extracting progressively higher-level features from an image:

> *"For example, the first layer maybe looks for edges or corners. Intermediate layers interpret the basic features to look for overall shapes or components, like a door or a leaf. The final few layers assemble those into complete interpretations—these neurons activate in response to very complex things such as entire buildings or trees."* (Mordvintsev et al, 2015 [no.1])

As the authors go on to describe, networks such as Inception that have been trained extensively on potentially millions of images will come to contain perceptual understandings of different objects (or more properly, features), based on the training images and how they have been set up to classify these. These understandings can be manipulated to create images from random visual noise, altering the pixels slightly so that the network's ability to recognise a particular feature is improved, and repeating the

process many times until a recognisable image of that feature appears – recognisable to humans, that is (see image 14).

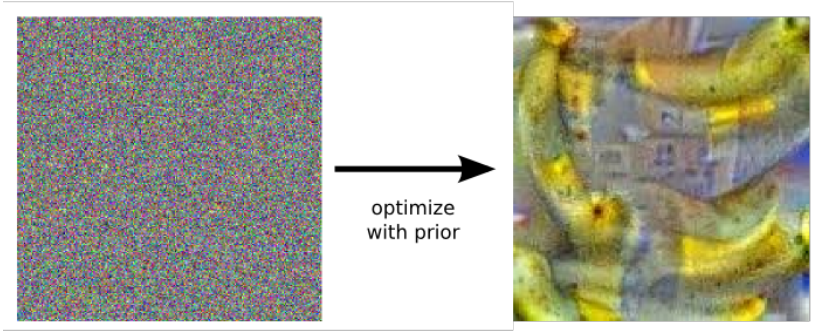

Image 14: Random noise altered and optimised to improve neural network feature recognition, creating a picture that represents the network's perception of bananas.

The benefit of doing this – and presumably the primary reason the blog post was made – is to visualise the aspects of features that a neural network learns, in order to gain more knowledge of how the networks actually learn, as well as troubleshooting issues. For instance, the team presents an example of their network being trained to recognise dumbbells, but when creating images of dumbbells from noise, they all include aspects of human bodybuilders in them, particularly arms (see image 15). This demonstrates a failing of the training data, implying that Inception has likely been shown very few (if any) images of a dumbbell in isolation.

Image 15: Several pictures generated in the same way as image 15, based on the network's understanding of dumbbells – all of which include some aspect of human bodybuilders, indicating imperfect training data.

Aside from these obvious practical benefits, the authors recognise the artistic potential of this technique. As the post progresses, they present multiple images of existing objects or landscapes that have been processed to accentuate certain features, some by focusing on earlier network layers (mainly edges or corners, as previously described) which creates a pleasing artistic filter-like effect – but others dealing with later, higher-level layers that are focusing on more sophisticated features or entire objects, which is where the true magic starts to happen.

Suddenly, a processed cloudscape becomes filled with shapes, mostly with animalistic elements as the network has been trained primarily to classify animal images (see image 16). This could almost be described as a computerised form of pareidolia, the ability of the human brain to recognise pre-learned patterns in otherwise unrelated stimuli, such as seeing faces in clouds – as many of us did as children.

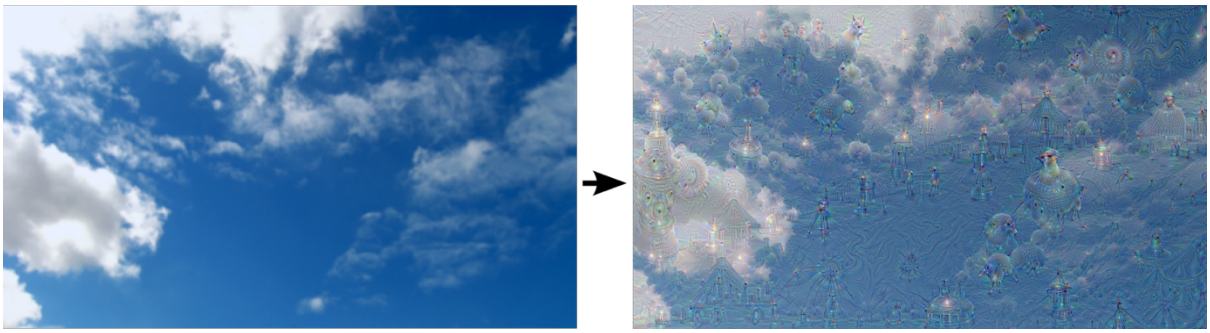

Image 16: Cloudscape before and after processing by a network trained to classify animal images.

Any image can be treated in such a way, even random noise – depending on which layers or features are focused on and how many iterations are run, the possible application to

create a whole array of surreal, dream-like images is immediately apparent. The authors end with an unknowingly precognitive statement:

> *"It also makes us wonder whether neural networks could become a tool for artists—a new way to remix visual concepts—or perhaps even shed a little light on the roots of the creative process in general."* (Mordvintsev et al, 2015 [no. 1])

Barely two weeks after this initial post, the same authors followed up on it, releasing the code to the public and including the name 'DeepDream' in the title of this subsequent post (Mordvintsev et al, 2015 [no.2]). This is the name by which this system came to be known, which has since achieved widespread popularity as a visual art algorithm, creating what has come be known as 'dreamed images'. Several websites, including DeepDream Generator (Deep Dream Generator, 2019) allow users to upload their own images and subject them to a variety of processes, expanding on the original DeepDream concept with a focus on the artistic results. This has resulted in a cascade of user-created dreamed images now stored in various libraries and galleries within these websites.

It was in this form that I first learned of DeepDream, having been pointed in the right direction by Alexis Kirke, my supervisor. I was currently in the process of implementing and experimenting with the Max/MSP spectrogram generator as previously described, and upon learning of DeepDream's existence and viewing the art created with it and its derivatives, I experienced my own personal 'eureka' moment. I could envisage a DeepDream-like system that could be trained to recognise certain sonic aspects of music and apply them to other existing musical compositions, sound recordings or simply noise, warping them and potentially creating a form of sonic art instead of visual.

Although this idea is sound-based as opposed to the image-based nature of DeepDream as it currently exists, through my spectrogram work I had learned how to create an image of a sound, or a piece of music. The features and layer-based information learned by an image classification network could potentially be analogous to different-level aspects of music: high-level such as genre signifiers, overall piece structure and instrumentation, and lower-level such as timbral or rhythmic characteristics. Musical features could be abstracted even further, to potentially include subjective emotional affect of particular compositions, for example. Whether a spectrogram image would actually represent this information in such a way is unknown, as is a neural network's ability to even recognise such features in the first place.

Regardless, the potential applications of such a theoretical system are vast. Short bursts of white noise could be forged into individual instrumental samples, musical genres or production styles could be blended together in novel ways by applying the feature perceptions of one genre/style onto a piece from another, or entirely original stylised pieces could be constructed based on the system's training on one particular genre or individual artist/composer.

The possibilities extend even outside the realm of musical composition, to those of speech production and audio analysis. As a potential new approach to 'teaching' AI systems to understand the fundamental construction of sound, such systems' ability to analyse and deconstruct sonic information could enable further developments in categorisation software for music or sound effect libraries, or improve speech recognition through better understanding of inflections and accents – possibly aiding in the production of more naturalistic machine speech in the process.

In the same way that websites such as DeepDream Generator allow public use of the framework to manipulate images, so too could websites or apps be built that could allow users to amalgamate musical pieces and styles of their choosing in fun and novel ways, or which could even be used to alter the sound of a person's voice to be more like their partner or a family member, like a version of 'Faceswap' for voices.

My excitement at the prospect did not blind me to the challenges such a system presented, however. DeepDream was designed and built to deal with images, and as such includes processes that are aimed at ensuring the system produces images that are visually coherent to humans. Spectrograms have their own visual coherence, but it is completely removed from traditional human notions of shapes and other visual features – as a result large parts of the DeepDream code would need to be rebuilt from scratch to account for this, which would likely involve lengthy testing periods and a fair amount of trial-and-error.

An example of this problem already exists. When researching to see if a similar system had already been designed, the closest project I found was a github repository by Marko Stamenovic titled 'audio-deepdream-tf' (Stamenovic, 2016), that takes spectrograms of example musical extracts and subjects them to the traditional DeepDream process, resulting in a Spectrogram that now has dog-like visual features. While interesting to look at, I imagined this outcome to be of questionable value from an audio perspective.

After downloading the spectrogram images and resynthesised audio files to examine the results for myself, I found as I had anticipated that the sonic effect created by the image-

based DeepDream process had produced strange distortions and a general sense of sonic warping that, while not unpleasant, were certainly less practical or ‘musical’ than those I had envisaged in such ‘dreamed audio’.

Despite my awareness of the likely difficulties of implementing such a system, I knew with complete certainty that this was the project I had been unknowingly working towards for months, possibly since before the ResM. The further research into, construction and testing of such a system would likely take a considerably longer period of time than the remainder of my Masters year, but nevertheless, I had decided that the pursuit of this project was to be my academic focus from here on out.

N.B. I am aware that this section would more properly be situated as part of the Chapter 1 survey of AI techniques, however I felt that the narrative of my ResM journey would be strengthened by its inclusion at this point instead.

3. Constructing & Testing the System

3.1 Initial Construction & Gathering of Materials

As a fully working DeepDream-inspired system for AI musical composition would be well outside the scope of a ResM, it became apparent that focusing instead on laying the foundations for such a system would be a more expedient goal. This would involve building and testing a simplified form of the initial classification process, where the system would learn to differentiate between distinct musical instrumental sounds as opposed to entire pieces or genres. Although rudimentary, this would serve as an important step in defining and trialling the principles on which a more advanced system would act.

My objectives, therefore, were clear.

- Implement a simple neural network classifier that was capable of processing and learning from sizeable datasets such as the spectrograms I was working with.
- Gather, curate and label a substantial bank of sound recordings of different instruments for the network to learn from.
- Develop a method for transferring the spectrogram data from the Jitter matrices into a format that could be understood by the network software.

3.1.1 Tensorflow, SciKit Learn & the Multi-Layer Perceptron

It had been becoming increasingly clear since my initial work in Max/MSP that attempting to learn a conventional programming language would be of great help in my ongoing studies. While there are certainly possibilities for my continued use of Max for neural networks as previously discussed, I am forced to acknowledge that other languages such as Python have a far greater range of available resources and machine learning libraries that would streamline the process considerably.

In particular, one resource that I came across at multiple stages of my research and early work was Tensorflow, a machine learning software library primarily used with Python. Having heard many positive accounts of this resource by online AI practitioners and noticing it featured in many explanatory videos on the subject on YouTube and other video sharing sites, I initially decided to learn Tensorflow to construct my classification system.

Being a complete novice at Python and having an extremely limited knowledge of line-based coding in general, I decided completing a dedicated tutorial course online would be a good idea if I were to make use of the language for my project. I had recently signed up to Pluralsight, a subscription-based online repository of video teaching courses on a wide range of subjects, including Python (Pluralsight, 2019). There was also a more advanced course which dealt with Tensorflow itself, I was pleased to discover.

After working through the basic Python course without too much difficulty, I progressed to Tensorflow. Here I encountered a number of challenges that hampered my progress,

including issues installing and running the Tensorflow library files on my Mac (the online instructors seeming to use Windows exclusively, while giving assurances that different operating systems worked perfectly well) as well as some difficulty with the more advanced nature of some of the syntax, which the previous Python tutorial had not covered. Additionally, the machine learning examples given in Tensorflow all seemed to make use of existing online datasets for the initial training process, linking to them through in-built Tensorflow commands. No information seemed to be given on how a user was to go about loading their own pre-built datasets.

In the process of searching for answers to this problem online, I had viewed a machine learning video tutorial on YouTube that, rather than Tensorflow, made use of a different library called 'Scikit Learn' (The SemiColon, 2017). Having also heard this mentioned by my tutors, I decided to attempt installing it and following along with the video. The example code, I quickly realised, was actually an implementation of a 'Multi-Layer Perceptron' – the object class in question named 'MLPClassifier' being something of a giveaway. I immediately began to find it much easier to get to grips with the code due to the working knowledge of MLP construction I had gained when building one in Max/MSP.

The dataset being used to test this system was a well-known example machine learning set know as 'MNIST' (LeCun et al, 2019). It is a collection of sixty-thousand unique handwritten Arabic numerals from zero to nine, each a 28x28 pixel greyscale image converted to a sequence of pixel brightness values. I had learned of this dataset's existence during my early research into neural networks, and that it is considered the standard by which many new systems are tested to determine their effectiveness. It was

the same dataset in fact, that was used in the Tensorflow course I had been taking through Pluralsight. However, the code in this particular video did not simply link to the dataset using part of an inbuilt library command but rather loaded and read it manually, for which the video author described the necessity of downloading the dataset from a hosting website as a standalone file, exactly the process I had been looking to learn.

It now became apparent that I would have greater success with Scikit Learn than with Tensorflow, at least in the short-term. I will consider returning to Tensorflow in the future, once my knowledge of Python has improved substantially, as it remains the apparent standard for many machine learning problems and will doubtlessly be useful in more advanced projects. But for the remainder of my ResM I shifted my efforts to focus entirely on Scikit Learn.

The code as given in the tutorial video was fairly simple to read even with my still relatively limited knowledge, however as the video creator's explanation on how the processes worked was rather minimal it became necessary to study the official documentation in order to better learn how to manipulate the system for my needs. Thankfully Scikit Learn's website contained multiple pages dealing with use of the MLP class and its different implementations.

The specific documentation for MLPClassifier (Scikit Learn, 2018 [no. 1]) describes the various parameters in detail – while the function of some like 'hidden_layer_sizes' or 'max_iter' was reasonably apparent, and a few such as 'activation' could be inferred from my previous knowledge of basic Neural Network construction, others such as 'solver',

‘learning_weight_init’ or ‘verbose’ required some reading before I could make proper use of them.

What followed was a period of experimentation through trial-and-error testing of the MLPClassifier implementation using the MNIST dataset as provided in the example code. Helpfully Scikit Learn included a ‘score’ method, which would return a mean value based on the accuracy of given training and test data. My goal through this testing period was to improve the accuracy score of the network as far as possible, then to later transfer the settings to my final analysis system to be trained on the instrumental data I would have amassed. Hopefully this would provide a good base line to work from, although I anticipated the settings would likely need to be tweaked slightly to fit the new data regardless.

To ensure consistency across my testing, I made use of the ‘random_state’ parameter, which when set to anything other than ‘none’ will ensure the initial state of the MLP weights is the same in each instance. This means that testing with the same parameters will produce exactly the same results every time and enables measurable testing of other parameters, eliminating variance from the inbuilt random-ness of the weight initialisation process. For the purposes of my code, I set this to ‘random_state=1’, although for certain tests I changed this to other values to see if particular parameter alterations were positive across the board, rather than just in this one configuration.

The code I had copied from the tutorial video had included a method of splitting the dataset into individual sets of training and test data, which was set to use 20% of the overall dataset for post-training testing, although this could be set to any desired value

and the random selection itself could also be set to a specific 'random_state', helpfully. I decided to leave this at 20%, although depending on how large my final dataset turned out to be I knew I may need to alter this.

My initial focus was on the size and quantity of hidden layers. The format of this parameter was fairly straightforward, such that any number of layers could be specified, each with a desired number of neurons. For instance: '100, 50' would result in two hidden layers, the first with 100 neurons and the second with 50. Initially I reasoned (naively, perhaps) that more layers and more neurons would automatically improve accuracy, but testing quickly revealed that the inner workings of the MLP were not quite that simple. To my surprise, I achieved excellent results with a single hidden layer of just 10 neurons, which had the additional benefit of reducing the time of each training iteration. This is perhaps due to the relative simplicity of the training data, although other factors outside of my understanding could well be responsible.

The 'solver' parameter dealt with the method of weight optimisation for the MLP, a subject that I had not researched in great depth. I was familiar with the 'gradient descent' process, both conceptually and practically through my Max/MSP work, but MLPClassifier allows for three separate solver methods, namely 'lbfgs', 'sgd' and 'adam'. The last of these appears to be a particular implementation of gradient descent (or more properly 'stochastic gradient descent', from which the abbreviation 'sgd' is taken), whereas the first is a 'quasi-Newton method', something I am entirely unfamiliar with. For practical purposes though, the website helpfully states:

> *"The default solver 'adam' works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and*

> *validation score. For small datasets, however, 'lbfgs' can converge faster and perform better."* (Scikit Learn, 2018 [no. 1])

Predictably, the 'adam' solver produced better test accuracy than 'lbfgs' or 'sgd' during my tests. As I was likely to have far fewer samples than the sixty-thousand of MNIST, I imagined 'lbfgs' would be superior when testing my final dataset based on the website's advice, although I would obviously need to test this at the time to confirm.

The list of available 'activation functions' was slightly smaller than that of my Max/MSP implementation, only including the standard functions of 'identity' – linear, 'logistic' – the sigmoid function that squashes the input between 0 and 1, 'tanh' – which squashes input between -1 and 1, and 'rectified linear unit' or 'relu' – converts any negative input to 0. As I had learned during my research that 'relu' was considered the standard activation function for many modern neural networks, I was unsurprised to see that MLPClassifier was set to use it by default. However, in my testing it appeared that 'tanh' produced the best results with the MNIST dataset.

The 'verbose' parameter, when set to 'true', simply causes the system to display the accuracy score after every iteration of training, so that progress can be monitored. While this is interesting to view and may be useful for more advanced systems, particularly for troubleshooting, it did not particularly help in any practical sense during my initial testing.

The only other parameters I tested exhaustively were 'learning_rate_init' and 'max_iter'. According to the documentation, the former sets the step size used when updating the weights, although my testing involved pure trial-and-error rather than some deeper

knowledge of how changing this improves training accuracy. The default value is set to 0.001, so I began with this and made various changes, attempting to find a value at which the system seemed to perform best – which eventually turned out to be 0.004. Once again I will test this more when I have the final dataset prepared.

'Max_iter' is fairly self-explanatory once the abbreviation is understood, and sets the maximum number of training iterations the system will go through before ending the process. This defaults to 200, which in my testing produced good results but took a considerable amount of time for each test. Reducing this to 20 did not seem to greatly impact the training score (in fact, in one random state the score was actually improved by the reduced iterations, which I am at a loss to explain) and vastly reduced the necessary training time, making my overall testing process run far more smoothly.

One important additional consideration, I came to realise, was the format of the data contained in the dataset file itself. Opening the MNIST file up in Excel to examine it properly, I noticed that the values of all cells were 'char' type, meaning whole numbers between 0 and 255. On the Scikit Learn website, while examining a page on supervised neural network models in general, I came across the following note on data format:

> *"Multi-layer Perceptron is sensitive to feature scaling, so it is highly recommended to scale your data. For example, scale each attribute on the input vector X to [0, 1] or [-1, +1], or standardize it to have mean 0 and variance 1."* (Scikit Learn, 2018 [no. 2])

The page then proceeds to demonstrate a method of standardising data in such a way, using a processing function called 'StandardScaler'. Testing this with the MNIST dataset, the accuracy score was instantly greatly improved, however I had thought of another way

of approaching this based on the advice. Since it suggested scaling each piece of data between 0 and 1, I programmed the system to divide the value of every cell in the input data by 255. This actually resulted in an even greater increase in accuracy than the standardisation method. Based on these results, I knew that the way I formatted the data in my final dataset was likely to have a significant impact on the system's eventual results.

My testing at this point had led to the system's accuracy almost always exceeding 95%, and frequently achieving 100% in various random starting states. I decided to end this initial testing process as it seemed little more progress would be gained, and I would likely have to repeat much of the process with the final dataset anyway. Despite this, I felt I had gained a far better understanding of the practical use of the MLPClassifier system, which would undoubtedly improve my ability to more efficiently test the system in the future.

3.1.2: Recording & Collecting Samples for Analysis

In a future fully-fledged version of my ideal AI compositional system, the training data would hopefully consist of entire pieces of music, labelled by genre, tempo, key signature and number of key changes, instrumentation, subjective emotional affect and as many other qualitative identifiers as desired. For the purposes of this project, though, I decided that simply dealing with a small number of distinct instrumental sounds would be perfectly adequate. Apart from simply being a more achievable goal within the available

timeframe, it would serve as a test to determine whether my chosen technique of spectrogram analysis was truly suited to subjection to a machine learning process.

The issue of which instruments to choose was something of a puzzle at first. My instinct was to choose an array of instruments that were as sonically distinct from each other as possible, so as to give the network a broad palette of musical sounds to work from. But I quickly came to the conclusion that this would have been fundamentally trivial, and would not have served as a good indication of a neural network's classification power. Had the instrumental tones and timbres been too wildly different in their attack transients and balance of harmonic and inharmonic content, for example, it would have been far too simple a matter for even a basic system to tell them apart.

I finally decided on three instruments: guitar, piano and snare drum. These choices were made partly by virtue of their wide availability and use – I own several guitars and a keyboard myself, and knew access to some additional instruments and samples through university resources and contacts would not be as difficult as with less standard instruments. The selection was also informed by the fact that each of these (when played in a conventional style) has a reasonably sharp attack transient – despite being clearly different sounds, they share some tonal characteristics that the system would be forced to recognise and learn from.

The next major hurdle was the size of the dataset I needed to amass. As described previously, the MNIST dataset of handwriting samples I had used for testing the SciKit Learn MLP included some sixty-thousand entries, a size that I knew would be nigh impossible to even approach with my limited time and resources. I concluded that a

hundred samples for each instrument should be sufficient for the purposes of a project at this level. These samples would involve a single note (or hit, in the case of the snare) left to ring out until it faded away naturally, without any unnecessary added effects such as artificial reverb or distortion.

I was reluctant to go about recording more than one or two samples of each instrument I owned personally, as I was worried it would make the datasets too homogenous and limit the system's ability to learn generalised information about each type of instrument. However, I encountered some initial difficulty in procuring other samples. I had put out a call on social media to my contacts from both Plymouth and Falmouth universities, along with the Max/MSP discussion group, asking for support with my project.

Despite many positive expressions of interest, after almost two weeks I had received samples from only one person – Satvik Venkatesh, also studying the ResM at Plymouth. He provided me with a sample for each instrument, for which I was grateful despite being disappointed at the poor overall response. I had also been unable to find much in the way of usable samples when searching free online databases and other resources. There were many royalty-free piano loops and guitar riffs to be found, but no individual note samples of these instruments as I required.

It was at this point that I was contacted by an international acquaintance I had made through the Max/MSP group, named Stéphane Thunus. We had originally begun conversations in the wake of my neural network tutorials, which he had frequently commented positively on. To my delight, he informed me that he had a large database of instrumental samples he had recorded personally over the course of several years –

including some of guitar, piano and snare – and he was willing to send me copies of them as gratitude for introducing him to neural networks with my tutorials.

This incredible resource was a godsend that came at just the right moment. Thanks to Stéphane I now had several hundred snare and piano samples, and almost 50 clean guitar samples to work with (the majority of the guitar recordings having used heavy added effects). Adding the samples I had received from Satvik, I needed only to find some additional guitar notes, for which I overcame my earlier misgivings and recorded myself, using three separate acoustic guitars and one unplugged electric, to achieve a variety of tones.

My only slight concern with the piano and snare samples I had received was that the variation between them seemed rather slight, at least as far as my ears could tell. The piano notes were all in the upper half of the range and, despite being played at several different volumes, were mostly subdued and subtle, never being hammered with any great force.

Likewise, the snare samples were all rather soft – the main variation between them being the model of microphone and technique they had been recorded with, along with whether they were a 'rimshot' hit or not. Despite my misgivings about the sonic variety of the samples, I had become anxious to begin the practical work of using them to train my neural network system, so decided to deal with this issue later in the project if it became a problem.

To set the instruments on a level starting point, and as a simple way of alleviating my concerns, when curating and editing the samples I decided to manually raise their amplitudes so that the peak volume of each was hitting the same level – raising them to 'unity gain', so to speak. My reasoning for this was that the neural network would not be able to learn to tell the instruments apart simply by analysing their respective volumes, and would instead be forced to deal with more complex aspects of their sonic construction.

The easiest way to achieve this was through the free digital audio workstation 'Audacity', in which the 'amplify' process will default to a value that will bring the audio peak of the current file in line with the software's inbuilt 0db level. In addition, during my editing of each sample I made sure to trim away all empty space before each note began, so that all samples would start instantly with the attack transient upon playback, again so that the network would not try to categorise the instruments based on the different amounts of silence at the beginning of each sample. With three hundred separate samples to edit and name in this way, the process was somewhat time-consuming.

Despite the tedium of the editing process, I finally had all the instrumental samples I needed ready for conversion to spectrograms, arranged into separate folders and individually named. All that was left was to determine the method of transforming the resultant spectrograms for use with the neural network. As mentioned before, I decided to just make use of the amplitude spectrograms and not the additional phase spectrograms, to keep the process relatively simple for the system.

3.1.3 Creating CSV Files from the Spectrograms

In the course of implementing the MLP in Sci-Kit Learn, I had learned that the common format for data entry into a supervised learning system such as the MLP was as 'comma-separated values' files, usually abbreviated to CSV. Lists of values separated by commas (hence the name), these are often viewed as spreadsheets in Microsoft Excel, and can be created from there.

The MNIST dataset I had downloaded for initial testing was in CSV format, and as previously described consisted of the pixel brightness values from 60,000 examples of 28x28 pixel images. This meant that each entry in the CSV file was 784 values long. Considering the size of my current spectrograms, I recognised a potential problem.

The number of frames in a generated spectrogram (which equates to pixels along the x-axis) is calculated in the patch as the length of the sound file in samples, divided by 512 and rounded. At the standard sample rate of 44100 per second, this works out at around 43 frames for just half a second of sound, which the majority of my samples greatly exceeded. In addition, as per usual for standard Fourier Transformations, the number of frequency bins is 512 – again directly equating to pixels on the y-axis. So, the spectrogram of just half a second of sound would translate to 22,016 values for each of the 300 instrument samples.

I was unsure whether to be concerned at this considerably larger size of data entry or not, knowing that existing practical machine learning systems likely dealt with datasets of far greater size without difficulty. However, the relative simplicity of the MLP I was working

with and the time it had already taken running each test with the MNIST dataset led me to err on the side of caution. If nothing else, I did not want each iteration I was to run with the instrumental dataset to take a great enough time to make repeated trial-and-error tests (which I anticipated) a long-drawn-out chore. If, I reasoned, the system handled a smaller selection of the spectrograms well at first, increasing the size of the entries in the future would be a simple matter.

Examining the visual makeup of a few of my sample spectrograms, I decided that a selection of 24 frames/pixels on the x-axis would contain enough information about each sound for the MLP system to work with – at just over a quarter of a second it contained the initial attack transient of each sound plus some of the main body, in fact it was enough to contain virtually all of the data for each snare sample, short as they were.

Additionally, I cut out the upper half of the 512 frequency bins/pixels on the y-axis, because the frequency spectrum as displayed in the spectrograms was linear (as opposed to a more accurate logarithmic scaling of frequencies in relation to pitch) and therefore this upper half contained frequencies above 10khz. Timbre in general can be described as the relative amplitudes of the harmonics of a note, although there are often many other factors involved (Nave, 2019). The ‘harmonic series’ begins with the fundamental frequency of the note and continues through whole-number multiples of this fundamental – the overtones. This means that for a note with a fundamental of 110hz for example, the first eight overtones would be 220hz, 330hz, 440hz, 550hz, 660hz, 770hz, 880hz and 990hz.

When considering cutting out frequencies above 10khz I had some initial concerns for the tonal fidelity of some of the higher-pitched notes, so I examined the samples before committing to this. The highest-pitched sample included in my dataset was a piano F#6 note with fundamental frequency of approximately 1480hz, so cutting off the spectrograms at 10khz left the first five overtones of this note intact (the fifth being at around 8880hz).

I reasoned this would not significantly impact the tonal characteristics of the note, partly due to my own past experience of constructing complex waveforms using additive synthesis – where I had found five overtones or fewer ample when producing a wide variety of instrumental-sounding notes – but also because when visually examining the spectrogram of the note I could see little-to-no frequency content above 10khz. I was therefore comfortable proceeding with the cut, deciding that if problems arose during training that could not be otherwise explained, I would attempt to reintroduce the higher frequencies then.

I would therefore be left with a spectrogram size of 24x256 pixels, equalling data entries of 6,144 values. At slightly over a quarter of the original size I had been considering, this seemed like a far more manageable figure for initial testing. Examining tutorials and help files regarding data flow in Jitter, I learned to manipulate the spectrogram data as it moved between matrices, slicing it at specific points as necessary, until I was left with the desired selection.

As a little more experimentation, I had managed to transfer the data values from the matrices into a specialised object called jit.cellblock, designed to represent matrix values

in spreadsheet form, which seemed perfect for my purposes. Yet despite scouring the documentation on the object and its various reference files, I could not determine a way to actually export the data from Max in any format, let alone as a CSV file.

While searching the Max forums for solutions, I happened upon a user-made object named ‘cellblock2csv’, which was an implementation of javascript code by a user named Adam Murray (Murray, 2010). Bringing this object into my patch and testing it, I was instantly able to export the contents of a jit.cellblock object as a CSV file. However, when I examined the saved file in Excel to ensure it had processed the data correctly, I found that only a small selection of each data entry was present. After a little more testing, I determined that jit.cellblock only seemed to hold data that was square in shape, namely if there were 300 rows (one for each of my samples), each row would only contain the first 300 values, cutting out the remaining 5,844 entirely.

As this did not seem to be normal behaviour, I raised the issue with (once again) the Max discussion group on Facebook, and was delighted to receive a message almost immediately from Darwin Grosse, staff member at Cycling74 and author of the jit.cellblock object. After presenting my problem in detail and supplying him with my current spectrogram patch, he determined that some part of the object’s programming was causing it to not function as intended, likely caused by the recent Max version update. He corrected the issue and thanked me for bringing it to his attention, scheduling the fixed version of jit.cellblock to be released as part of the next official patch and sending me an advance copy to work with.

With the data now being displayed in full in jit.cellblock, I was glad to begin the process of exporting the complete CSV files for each instrument. But my technical issues were not over, it seemed, as these files now refused to open correctly in Excel, displaying an error message that implied the data was corrupted in some way. Likewise, they would not function when loaded in Python for my MLP system.

After more troubleshooting, appeals for help and various bits of trial-and-error testing, I managed to find a solution to this additional frustration. It seemed that CSV files created in this way that had more than a certain number of horizontal cells would always open with an error, but files created within Excel itself would not – this explained why the MNIST dataset opened successfully in Excel as despite having tens of thousands of rows, each row only had 785 horizontal cells (including the label). I was able therefore, to save several smaller CSV files from Max, open each of them in Excel and then amalgamate them together manually through copy-and-paste. It was hardly an elegant solution and required more manual labour, but I was too glad to have overcome this hurdle to be concerned.

The final CSV files now instead contained the lower 180 frequency bins of each spectrogram, divided into three groups of 60 when exporting from Max – this had been the highest number I had been able to open successfully in Excel when testing. Although reducing the frequency content of the files even further, I was still confident they contained more than enough information for the MLP system to learn from. The complete samples therefore, now contained entries of 4320 data points each.

Having finally succeeded in my attempts to save the spectrogram data as CSV files, I had the complete instrumental training data ready to load in the Python program for my MLP system to learn from.

3.2 Testing the Final System & Analysis of Results

3.2.1 Initial Tests

Now that everything was in place, the testing of my MLP system with the samples I had amassed could begin. I had combined the sample data into a number of master CSV files:

guitar_char_combined.csv

guitar_float_combined.csv

piano_char_combined.csv

piano_float_combined.csv

snare_char_combined.csv

snare_float_combined.csv

combined_char_training_set.csv

combined_float_training_set.csv

The latter two of these contain the data of all three instruments in one file, as is necessary for the final testing. I have included these and all subsequent CSV files and Python code as part of my website (www.spearced.com) should further examination of them be of interest.

As I had previously encountered some varying results depending on the format of the data, I had decided to save my sample spectrograms in both char and float format, allowing me to test their differences further. I kept the parameters of the system initially the same as when using the MNIST dataset, testing both the combined char (standardised) and float sets individually ten times each, but removing the 'random_state' parameters from both the train/test split (kept at 80%/20%) and the MLP itself. These results are displayed in tables 1 and 2.

As can be seen, the float set produced some higher individual results than the standardised char set, but the latter succeeded with greater consistency, resulting in a higher mean score for both training and testing. Both sets produced many results of lower than 95% accuracy, which I was sure I could improve upon.

I proceeded to test different parameters with the 'random_state' parameters enabled once again, first by changing the solver to 'lbfgs' as advised by the Scikit Learn documentation. This resulted in a noticeable increase in accuracy immediately, but I still thought I could do better before recording additional tables and averages. I had previously set the maximum iterations for the code to 20, partly as a way of reducing the training time of each test. However, this was resulting in a warning from Python that the solver had not 'converged' by the time this maximum was hit, so I decided to test greater values. When I tried increasing this parameter to 400 the accuracy seemed to improve very slightly, without vastly increasing the training time. I attribute this to the far fewer number of samples in my bespoke dataset, 300 as opposed to the 60,000 of MNIST, even though my samples are individually much larger.

Changing the ‘learning_rate_init’ did not seem to have any positive effect on the results, in fact only one out of ten different random states I tested produced a marginally better score with a different value for this parameter, while reductions in accuracy held true in every other case. I decided to leave it at 0.004 as a result. Interestingly, changing the activation function to ‘logistic’ (the basic sigmoid function) seemed to produce the best results in my testing. Confident that I had changed enough parameters to achieve significantly higher accuracy across the board, I recorded tests of the same size as previous, for both data types again. The results can be seen in tables 3 and 4.

The mean values for both training and test sets in both cases display considerably greater accuracy, with the float set slightly edging out the char set in overall score. I was extremely happy with these results, but before bringing this round of testing to a close entirely I decided to try changing the solver parameter one final time, to confirm whether ‘lbfgs’ was indeed the superior method for a dataset of this size. Choosing ‘sgd’ resulted in scores that, while still good, were noticeably lower than those achieved with ‘lbfgs’. However, when testing the dataset with ‘adam’, the scores appeared extremely close, so much so that I decided to run one final series of recorded tests, the results of which can be seen in tables 5 and 6. For comparison, the mean scores for each parameter configuration can be seen in image 17.

The evidence were clear. Based on the mean scores of every parameter configuration I had tested, the most accurate was the float dataset with the ‘adam’ solver, increased maximum iterations and ‘logistic’ activation function – configuration 5. Although the training set accuracy was marginally higher in configuration 4, the test set accuracy was

far higher in 5, so on balance configuration 5 was best overall. As a result of these findings, I decided to keep the parameters static for all additional tests from now on, resolving only to change them if any significant issues with training accuracy arose. This brought my initial round of testing to a close.

Dataset:	combined_float_training_set.csv	
Notes:	Original MNIST testing parameters	
	Training Set Accuracy	**Test Set Accuracy**
Test 1:	0.995816	0.966667
Test 2:	0.669456	0.633333
Test 3:	0.991632	1
Test 4:	0.65272	0.7
Test 5:	0.995816	0.966667
Test 6:	0.991632	0.983333
Test 7:	0.991632	0.983333
Test 8:	0.991632	1
Test 9:	0.933054	0.833333
Test 10:	0.661088	0.683333
Mean:	0.8874478	0.8749999

Table 1: Accuracy results of first float training set trained on initial parameters.

Dataset:	combined_char_training_set.csv	
Notes:	Original MNIST testing parameters	
	Training Set Accuracy	**Test Set Accuracy**
Test 1:	0.949791	0.933333
Test 2:	0.941423	0.883333
Test 3:	0.966527	0.933333
Test 4:	0.941423	0.883333
Test 5:	0.966527	0.95
Test 6:	0.92887	0.933333
Test 7:	0.958159	0.966667
Test 8:	0.962343	0.95
Test 9:	0.958159	0.883333
Test 10:	0.966527	0.916667
Mean:	0.9539749	0.9233332

Table 2: Accuracy results of first char training set trained on initial parameters.

Dataset:	combined_float_training_set.csv	
Notes:	Improved testing parameters: lbfgs solver	
	Training Set Accuracy	Test Set Accuracy
Test 1:	0.955816	1
Test 2:	1	0.966667
Test 3:	1	0.983333
Test 4:	1	0.933333
Test 5:	0.995816	0.983333
Test 6:	1	0.95
Test 7:	0.995816	0.95
Test 8:	1	0.966667
Test 9:	0.995816	0.983333
Test 10:	1	0.933333
Mean:	0.9943264	0.9649999

Table 3: Accuracy results of first float training set after improving parameters and changing to 'lbfgs' solver.

Dataset:	combined_char_training_set.csv	
Notes:	Improved testing parameters: lbfgs solver	
	Training Set Accuracy	Test Set Accuracy
Test 1:	1	0.933333
Test 2:	1	0.966667
Test 3:	1	0.933333
Test 4:	0.995816	0.983333
Test 5:	0.995816	0.966667
Test 6:	1	0.95
Test 7:	0.995816	0.95
Test 8:	1	0.933333
Test 9:	0.995816	0.95
Test 10:	1	0.966667
Mean:	0.9983264	0.9533333

Table 4: Accuracy results of first char training set after improving parameters and changing to 'lbfgs' solver.

Dataset:	combined_float_training_set	
Notes:	Improved testing parameters: adam solver	
	Training Set Accuracy	Test Set Accuracy
Test 1:	0.995816	1
Test 2:	1	0.966667
Test 3:	0.995816	1
Test 4:	1	1
Test 5:	0.995816	0.983333
Test 6:	1	1
Test 7:	1	0.983333
Test 8:	0.991632	1
Test 9:	0.995816	1
Test 10:	0.995816	0.983333
Mean:	0.9970712	0.9916666

Table 5: Accuracy results of first float training set after improving parameters and changing back to 'adam' solver.

Dataset:	combined_char_training_set.csv	
Notes:	Improved testing parameters: adam solver	
	Training Set Accuracy	Test Set Accuracy
Test 1:	0.995816	1
Test 2:	0.987448	0.933333
Test 3:	0.995816	0.983333
Test 4:	0.995816	0.966667
Test 5:	0.987448	0.983333
Test 6:	0.995816	0.966667
Test 7:	0.983264	0.95
Test 8:	0.995816	0.95
Test 9:	0.995816	0.983333
Test 10:	0.987448	0.966667
Mean:	0.9920504	0.9683333

Table 6: Accuracy results of first char training set after improving parameters and changing back to 'adam' solver.

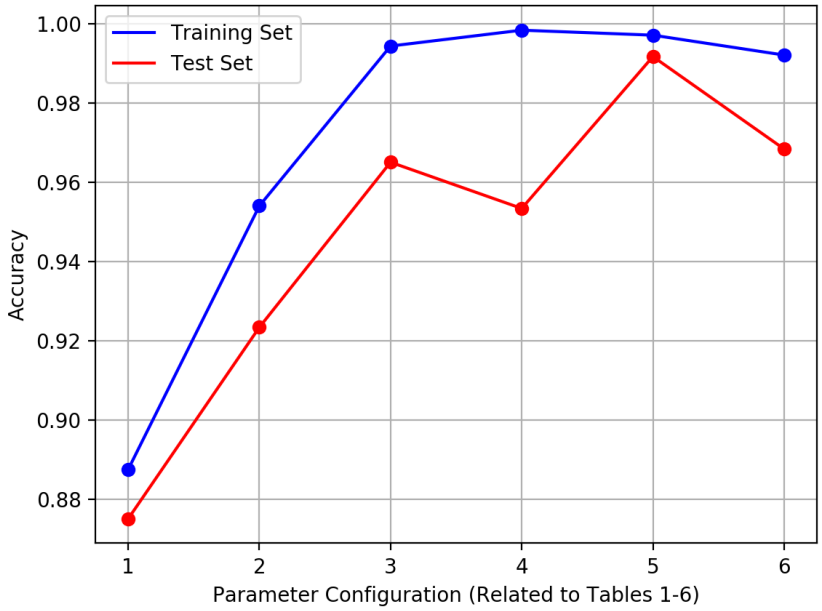

Image 17: Mean training and test set accuracy across all parameter configurations.

3.2.2: Weight Visualisations

The success of my testing and the extremely high accuracy scores that the MLPClassifier had achieved proved without a doubt that my hypothetical goal had been achievable – a neural network truly was capable of learning to differentiate between distinct musical instrumental sounds simply by analysing the pixel brightness values of their spectrograms.

However this was not quite the end of my project, as I believed additional testing and some specific trials would be beneficial in judging the system's potential ability to manipulate sounds in a DeepDream-like fashion, along with elucidating some superficial information about how the network was learning to differentiate between the sounds.

To begin, I wanted to find a way of visualising the neural network weights themselves, firstly when training the system on the initial dataset, then again after training once more on each of the instruments in turn. My reasoning for this was as follows: a neural network such as the image classifier 'Inception' would have previously been trained on general image recognition and classification, before the DeepDream algorithm forced it to focus on one particular feature or perceptive layer. As I was unable to manipulate the MLPClassifier code in such a way as to enable this kind of feature prioritisation, I reasoned that re-training the network on one particular instrument after initial general training on the full dataset might produce a similar form of feature prioritisation.

While I had no knowledge of how to enable the network to manipulate incoming data to better fit these prioritised features (I am sceptical as to whether this is something the MLP system is even capable of, being such a relatively simplistic network), I believed that if the visual distribution of weights changed noticeably to become more uniform after additional training, this would give an indication of some superficial evidence to support my hypothesis.

To extract the weights, I consulted the Scikit Learn MLPClassifier documentation once more (Scikit Learn, 2018 [no. 1]). One of the listed attributes of the network was 'coefs', short for coefficients, which represented the weight matrices for the various layers. Attempting to view this data to confirm if it was what I needed, I expanded my code to print the values of the different 'coefs' elements. For 'coefs_[0]', this produced 10 lists of 4320 data points each, corresponding to the weights between each input and the 10 neurons of the network's hidden layer. Likewise, 'coefs_[1]' produced 3 lists of 10 data

points each, corresponding to the weights between the hidden neurons and the output layer, which had three points due to the three different instruments and their labels. Now I just needed to find a way to view this data in a more easily comprehendible format.

A popular Python library for data visualisation through graphs and charts is ‘matplotlib’, which I had come across during my study of Tensorflow – it had been used to briefly plot the results of a test in one of the tutorials I had worked through. Thanks to this tutorial, I had gained some cursory knowledge of matplotlib usage, which was bolstered by study of the official documentation (Matplotlib, 2018).

Returning to my code, I attempted to implement a simplistic line graph to display both matrices of weights in turn. The resulting graphs are included here as images 18 and 19.

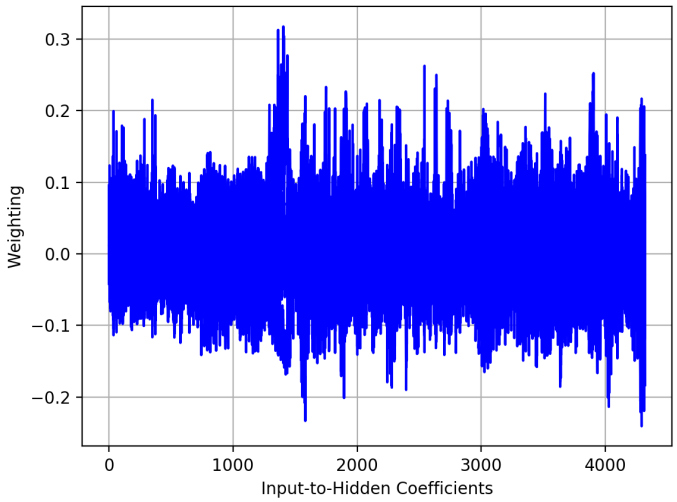

Image 18: Graphical representation of hidden layer weight matrix from the trained MLPClassifier.

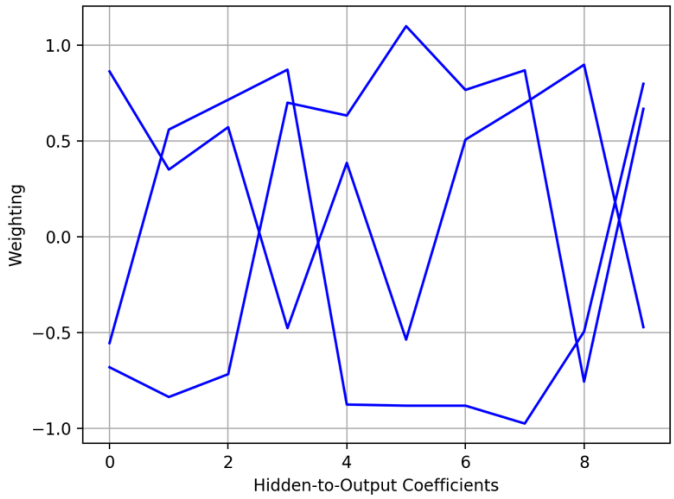

Image 19: Graphical representation of output layer weight matrix from the trained MLPClassifer.

While these visualisations were certainly interesting, they did not present any immediately helpful information about how the neural network was actually perceiving the data itself and the relative importance of each input in determining its final prediction of the class label. Some additional experimentation and manipulation of the data was required.

Pondering the issue, I reminded myself that each of the three output neurons would have ten weight values applied to them, one for each of the hidden neurons. Conceptually speaking, these weights represented how valuable the information perceived by each hidden neuron was in determining the strength of the label prediction, with respect to their corresponding output neuron. In image 19, these weights are displayed as three lines with ten points each, fluctuating between positive and negative values. For the purposes of better understanding them, I separated the data into three individual lists, which can be seen in images 20, 21 and 22.

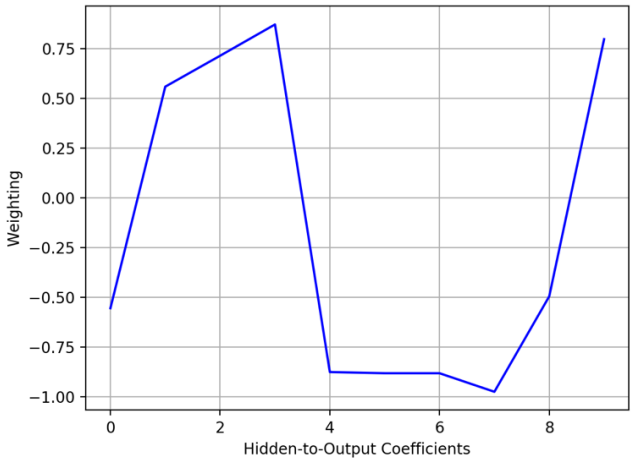

Image 20: Extracted list of first output neuron weights – guitar classifier.

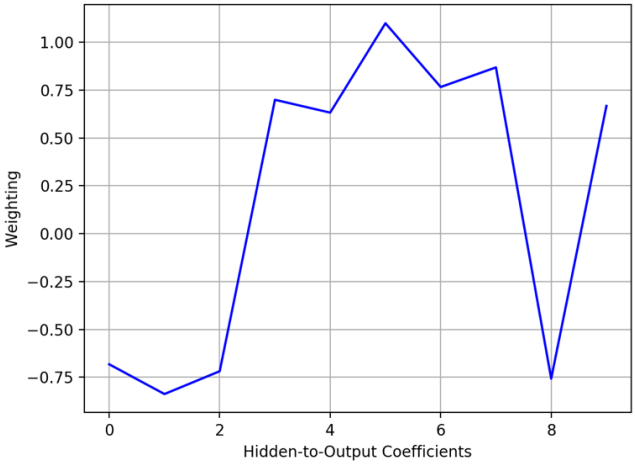

Image 21: Extracted list of second output neuron weights – piano classifier.

Image 22: Extracted list of third output neuron weights – snare classifier.

If, I wondered, there was a way of multiplying the values of the input-hidden weights by each of these separate hidden-output weights, perhaps that would provide a more visually informative representation of the system's learned understanding? Attempting to prove this theory was something of a chore, as I was completely unfamiliar with this kind of data manipulation in Python. As a result this involved a great deal of trial-and-error, applying various transformations to each matrix while Python frequently provided error messages that stopped compilation of my code.

Finally, I succeeded in my efforts. As part of this process I had included functions to scale the data in the weight matrices to be better displayed visually – implementing the sigmoid function I had learned through my neural network construction in Max/MSP. This resulted in the weightings all being displayed as positive, which isn't technically accurate but means that the multiple lines that make up the graphs are more visible – plus the distinct shape of each has been retained, which is the primary goal of the visualisation anyway. The final visual representations of the data are displayed in images 23, 24 and 25.

Image 23 is the relative weight balances for classification of a guitar sound, 24 for piano and 25 for snare. These are all clearly visually distinct, a successful demonstration of the system's weight balances enabling it to infer different information from the hidden neurons in each of the three output cases.

The next step was to take the trained network and forcibly re-train it on one instrument at a time, analysing the visual weight representations afterwards to see if and how they were altered by this process. Turning once again to the MLPClassifier documentation, I learned

that the ‘fit’ method I had used up until now to train the network had an alternative, named ‘partial_fit’. This could be used to for additional training on an incomplete dataset, i.e. one that only contained a single label such as the individual instrument sets I wished to use.

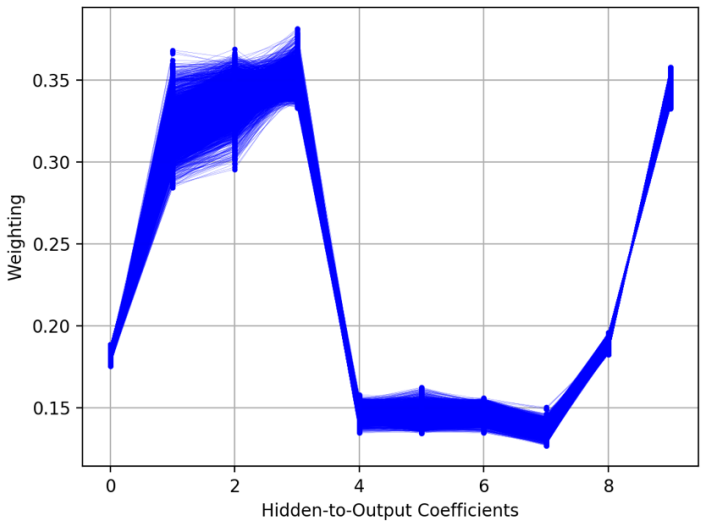

Image 23: Graphical representation of the weight matrices after multiplying the hidden layer weights by the output neuron weights for guitar classification, with some scaling.

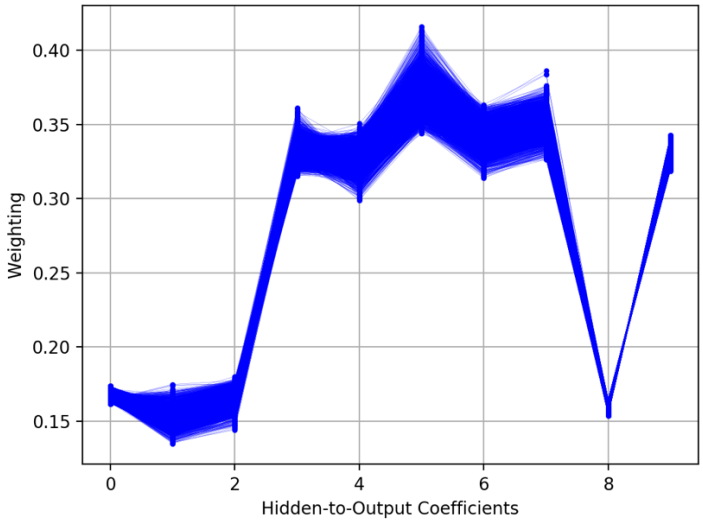

Image 24: Graphical representation of the weight matrices after multiplying the hidden layer weights by the output neuron weights for piano classification, with some scaling.

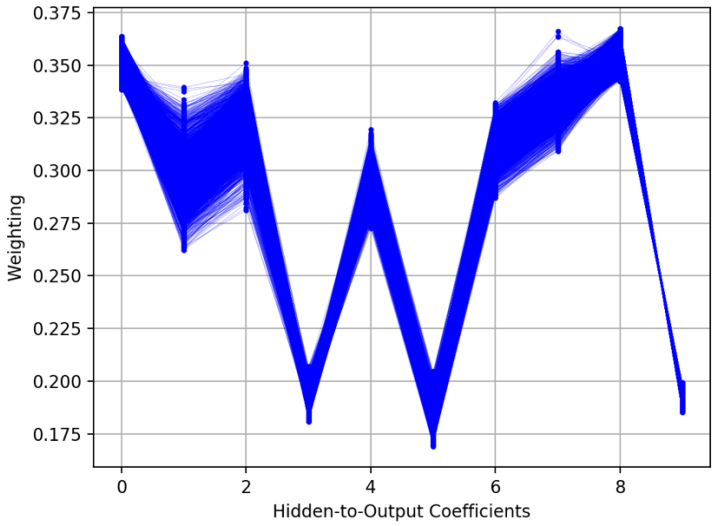

Image 25: Graphical representation of the weight matrices after multiplying the hidden layer weights by the output neuron weights for snare classification, with some scaling.

I ran into an unexpected obstacle when attempting this however: as the network had already achieved such excellent accuracy scores from the initial training data, it would not run this partial_fit for more than a single iteration, as the error was so low. It took some additional research and the discovery of a post on 'stack overflow' – a general programming forum – in which a user named 'Bita' was attempting to troubleshoot some related code that made use of a partial_fit (Bita, 2016). The solution, given by another user named 'Curious', included a small snippet of code that I recognised would solve my own problem: it manually forced the system to run multiple iterations of the partial_fit in a loop.

Now that I was able to achieve my goal, I set the system re-training on each instrument in turn, displaying the newly trained weight values alongside the originals. I scaled these slightly differently to images 23-25, so that they would more accurately display both

positive and negative values. The results of the network being re-trained on guitar samples is shown in image 26, on piano in image 27 and on snare in image 28.

These results were initially surprising – I had imagined the act of re-training on each instrument would cause the shape of each weight visualisation line to each fall in line with that of the focused instrument, eventually looking far more uniform. This was not the case, although after examining the images more closely I realised they still appeared to prove my conjecture – the weights of the focused instruments we now solely comprised of positive values, while those of the other instruments were solely negative.

The actual value of these visualisations and the conclusions I have drawn from my analysis of them is certainly debateable, as I am aware I could be completely mistaken in my approach and that the results could be essentially meaningless. However, I remain satisfied with the work and its outcome, from a visual perspective if nothing else.

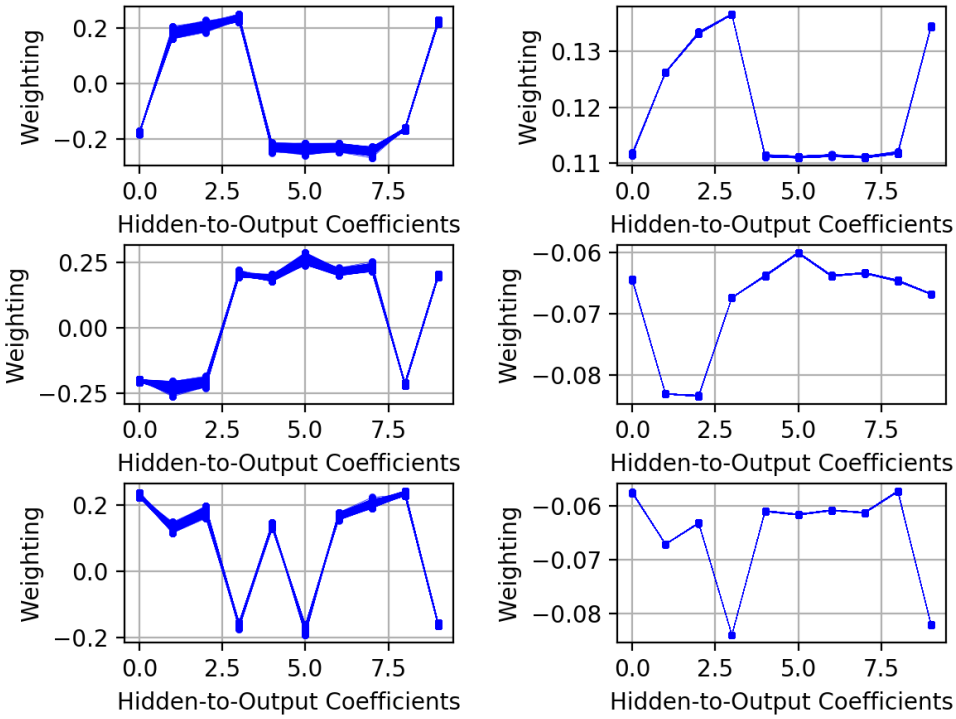

Image 26: Weights before and after re-training solely on guitar samples.

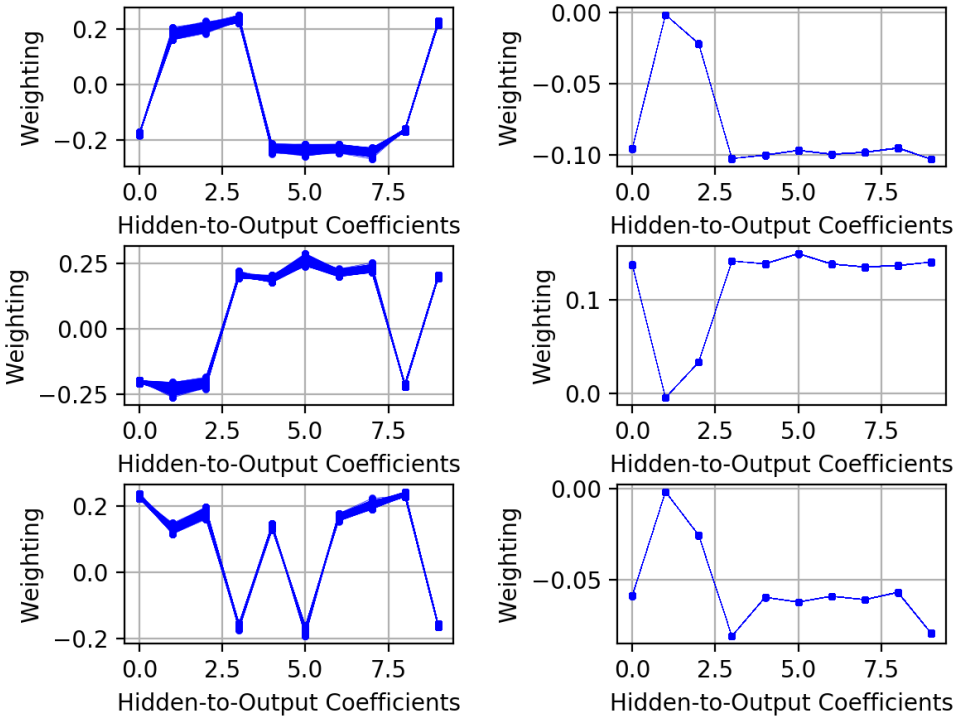

Image 27: Weights before and after re-training solely on piano samples.

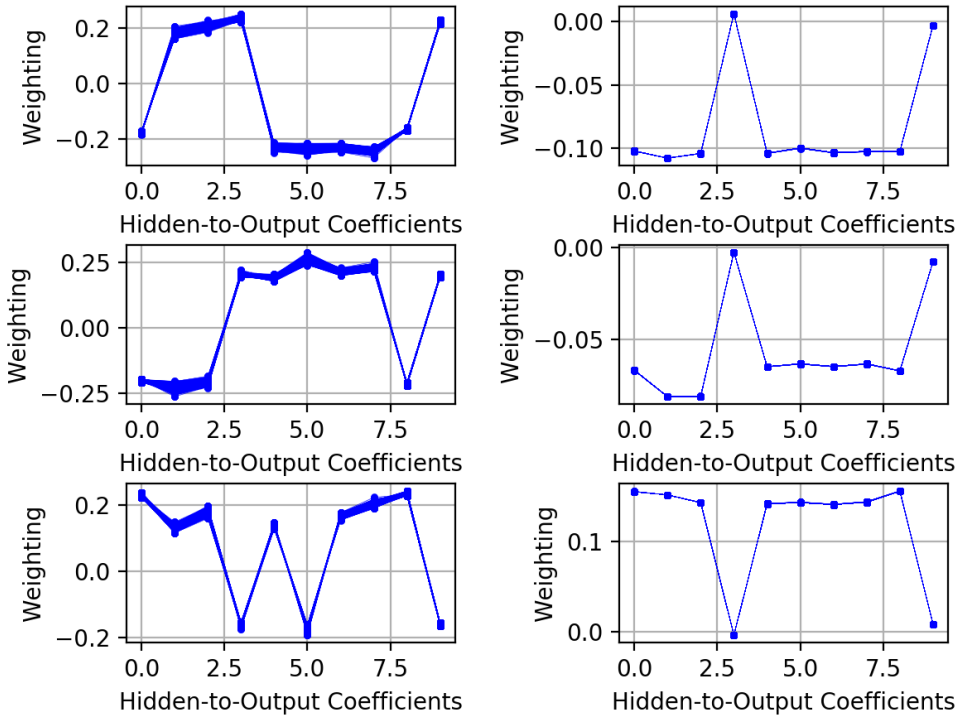

Image 28: Weights before and after re-training solely on snare samples.

3.2.3 First Round of Additional Tests

My next task was to provide some additional testing data for the system to tackle. Although the train/test split of the dataset I had been working with up until now had produced successful results, my earlier concerns about the sonic variety of the samples had resurfaced. I decided that a way of testing how well the system had learned to differentiate between instruments would be to record additional samples of each myself, five for each instrument. The piano and snare samples all came from my electric keyboard, which had multiple built-in drum kit sounds, whereas the guitar samples were recorded from my various acoustic and electric guitars as before – although I made an

effort to ensure they were distinct from those I had recorded for the original dataset in an attempt to test how well the system had learned broader sonic aspects of each instrument.

I generated spectrograms of the new samples using the same three-part process as before, naming them as follows:

add_guitar[1-5].csv

add_piano[1-5].csv

add_snare[1-5].csv

Along with these, I had been considering other methods of challenging the system that might give more insight into the learning process itself, bearing the DeepDream process in mind. To this end, I had decided to test the system on two further sets of data: sample convolutions and manual combinations of different spectrograms.

The first of these makes use of Fourier Transformations once again. Several years ago during my early work in Max/MSP, I had been interested in the 'vocoder' effect that was prevalent in some popular music, gaining notoriety in the 1970s through its use by Kraftwerk and Herbie Hancock among others (Beta, 2016) and continuing to the present day in albums by Daft Punk (Daft Punk, 2013) and Jamiroquai (Jamiroquai, 2017), for example.

The vocoder effect involves a human voice that is processed through a musical sound signal, combining timbral characteristics of both. One way of achieving this effect that is remarkably simple to implement in Max is through a process called 'convolution'. Most

frequently used to achieve particular reverb effects (Walker, 2005), convolution can be used to merge any two sound sources together, leaving their shared frequencies intact while cutting out all others. A video tutorial I had viewed at the time that dealt with a Max convolution patch for a vocoder effect (Dude837, 2010) provided me with all the information I needed to repurpose the effect for use in my current project.

I had decided to test the system on convolutions because they were an existing sonic equivalent of the kind of perceptual blurring DeepDream produced, and I was hopeful that analysing the results would provide some insight into how a future DeepDream-esque system would blend instruments – but also because I was also simply curious to see how the trained system would classify these convoluted samples. To ensure all three instruments were convoluted with each other in turn, I selected four different samples from each (to provide a good spread of results) and paired them arbitrarily.

When running the samples through the convolution patch, the only consideration was which one to make the 'carrier', which would determine the pitch of the final convoluted sample. Wishing to broaden the results as far as I could, I ran all sample pairs through the process twice, changing carrier each time. This left me with twelve sound files which I then converted to spectrograms. Due to the re-use of samples in different combinations, there was a marginally more complex naming process for these files – see table 7 for details. Essentially, the first instrument stated in the file name indicates the carrier.

Finally, I had decided to manually combine spectrograms of different instruments in various permutations. This was partly facilitated by the nature of how I created the spectrograms (three sets of 60-pixel-high spectrograms combined together), involving

instead rather a large amount of manual labour to achieve. I took pairs of instrumental samples as with the convolutions, gathered the spectrogram pieces for each and proceeded to combine them in all possible permutations, using two sets of samples from each instrument this time instead of four due to the increased number of final files I would have anyway.

This resulted in a total of thirty-six combination files – see table 8 for details of their construction and naming. Although a still more complex system than the convolution files, once again the first instrument in the file name indicates prioritisation of that instrument in that file – in this case meaning that two of the three spectrogram parts are from that instrument. Whether the latter part of the file name is .1 .2 or .3 depends on the placement of the minority instrument in order. I decided to construct these combination files in this way as it potentially gives us information about which groups of frequency bins are most important for instrument classification – analysis of the results would hopefully indicate if particular ranges in the frequency spectrum are more highly-valued by the system.

All three additional testing datasets now prepared, the only remaining task was to create new training files for the various pairs of instruments:

guitar&piano.csv

guitar&snare.csv

piano&snare.csv

This was specifically for the benefit of the convolution and combination testing. While the ‘add’ files would be tested on a system training on the entire original dataset, I felt it

Sample Convolutions		
First Sound File (Carrier)	**Second Sound File**	**Output File Name**
Guitar_Sam1.wav	Piano_Ste1.wav	Conv_Guitar_Piano1.csv
Piano_Ste1.wav	Guitar_Sam1.wav	Conv_Piano_Guitar1.csv
Guitar_Ste20.wav	Piano_Sat1.wav	Conv_Guitar_Piano2.csv
Piano_Sat1.wav	Guitar_Ste20.wav	Conv_Piano_Guitar2.csv
Guitar_Sam20.wav	Snare_Sat1.wav	Conv_Guitar_Snare1.csv
Snare_Sat1.wav	Guitar_Sam20.wav	Conv_Snare_Guitar1.csv
Guitar_Ste30.wav	Snare_Ste10.wav	Conv_Guitar_Snare2.csv
Snare_Ste10.wav	Guitar_Ste30.wav	Conv_Snare_Guitar2.csv
Piano_Ste10.wav	Snare_Ste20.wav	Conv_Piano_Snare1.csv
Snare_Ste20.wav	Piano_Ste10.wav	Conv_Snare_Piano1.csv
Piano_Ste90.wav	Snare_Ste90.wav	Conv_Piano_Snare2.csv
Snare_Ste90.wav	Piano_Ste90.wav	Conv_Snare_Piano2.csv

Table 7: List of sample convolutions, their component sound files and saved file names.

Spectrogram Combinations			
Note: each full spectrogram is made up of three files, each of which contains 60 frequency bins for each sound.			
Lowest 60 bins	**Middle 60 bins**	**Highest 60 bins**	**Output File Name**
Guitar_Sam5.wav	Guitar_Sam5.wav	Piano_Ste4.wav	Comb_Guitar_Piano1.1.csv
Guitar_Sam5.wav	Piano_Ste4.wav	Guitar_Sam5.wav	Comb_Guitar_Piano1.2.csv
Piano_Ste4.wav	Guitar_Sam5.wav	Guitar_Sam5.wav	Comb_Guitar_Piano1.3.csv
Piano_Ste4.wav	Piano_Ste4.wav	Guitar_Sam5.wav	Comb_Piano_Guitar1.1.csv
Piano_Ste4.wav	Guitar_Sam5.wav	Piano_Ste4.wav	Comb_Piano_Guitar1.2.csv
Guitar_Sam5.wav	Piano_Ste4.wav	Piano_Ste4.wav	Comb_Piano_Guitar1.3.csv
Guitar_Ste2.wav	Guitar_Ste2.wav	Piano_Ste54.wav	Comb_Guitar_Piano2.1.csv
Guitar_Ste2.wav	Piano_Ste54.wav	Guitar_Ste2.wav	Comb_Guitar_Piano2.2.csv
Piano_Ste54.wav	Guitar_Ste2.wav	Guitar_Ste2.wav	Comb_Guitar_Piano2.3.csv
Piano_Ste54.wav	Piano_Ste54.wav	Guitar_Ste2.wav	Comb_Piano_Guitar2.1.csv
Piano_Ste54.wav	Guitar_Ste2.wav	Piano_Ste54.wav	Comb_Piano_Guitar2.2.csv
Guitar_Ste2.wav	Piano_Ste54.wav	Piano_Ste54.wav	Comb_Piano_Guitar2.3.csv
Guitar_Sam5.wav	Guitar_Sam5.wav	Snare_Ste4.wav	Comb_Guitar_Snare1.1.csv
Guitar_Sam5.wav	Snare_Ste4.wav	Guitar_Sam5.wav	Comb_Guitar_Snare1.2.csv
Snare_Ste4.wav	Guitar_Sam5.wav	Guitar_Sam5.wav	Comb_Guitar_Snare1.3.csv
Snare_Ste4.wav	Snare_Ste4.wav	Guitar_Sam5.wav	Comb_Snare_Guitar1.1.csv
Snare_Ste4.wav	Guitar_Sam5.wav	Snare_Ste4.wav	Comb_Snare_Guitar1.2.csv
Guitar_Sam5.wav	Snare_Ste4.wav	Snare_Ste4.wav	Comb_Snare_Guitar1.3.csv
Guitar_Ste2.wav	Guitar_Ste2.wav	Snare_Ste54.wav	Comb_Guitar_Snare2.1.csv
Guitar_Ste2.wav	Snare_Ste54.wav	Guitar_Ste2.wav	Comb_Guitar_Snare2.2.csv
Snare_Ste54.wav	Guitar_Ste2.wav	Guitar_Ste2.wav	Comb_Guitar_Snare2.3.csv
Snare_Ste54.wav	Snare_Ste54.wav	Guitar_Ste2.wav	Comb_Snare_Guitar2.1.csv
Snare_Ste54.wav	Guitar_Ste2.wav	Snare_Ste54.wav	Comb_Snare_Guitar2.2.csv
Guitar_Ste2.wav	Snare_Ste54.wav	Snare_Ste54.wav	Comb_Snare_Guitar2.3.csv
Piano_Ste4.wav	Piano_Ste4.wav	Snare_Ste4.wav	Comb_Piano_Snare1.1.csv
Piano_Ste4.wav	Snare_Ste4.wav	Piano_Ste4.wav	Comb_Piano_Snare1.2.csv
Snare_Ste4.wav	Piano_Ste4.wav	Piano_Ste4.wav	Comb_Piano_Snare1.3.csv
Snare_Ste4.wav	Snare_Ste4.wav	Piano_Ste4.wav	Comb_Snare_Piano1.1.csv
Snare_Ste4.wav	Piano_Ste4.wav	Snare_Ste4.wav	Comb_Snare_Piano1.2.csv
Piano_Ste4.wav	Snare_Ste4.wav	Snare_Ste4.wav	Comb_Snare_Piano1.3.csv
Piano_Ste54.wav	Piano_Ste54.wav	Snare_Ste54.wav	Comb_Piano_Snare2.1.csv
Piano_Ste54.wav	Snare_Ste54.wav	Piano_Ste54.wav	Comb_Piano_Snare2.2.csv
Snare_Ste54.wav	Piano_Ste54.wav	Piano_Ste54.wav	Comb_Piano_Snare2.3.csv
Snare_Ste54.wav	Snare_Ste54.wav	Piano_Ste54.wav	Comb_Snare_Piano2.1.csv
Snare_Ste54.wav	Piano_Ste54.wav	Snare_Ste54.wav	Comb_Snare_Piano2.2.csv
Piano_Ste54.wav	Snare_Ste54.wav	Snare_Ste54.wav	Comb_Snare_Piano2.3.csv

Table 8: List of spectrogram combinations, their component partial spectrograms and saved file names.

Additional Sample Testing – 'add' files

Dataset: combined_float_training_set.csv

Sample Tested	Guitar (%)	Piano (%)	Snare (%)	Correct Prediction?
add_guitar1	98	1	1	yes
add_guitar2	98	1	1	yes
add_guitar3	98	1	1	yes
add_guitar4	98	1	1	yes
add_guitar5	98	1	1	yes
add_piano1	98	1	1	no
add_piano2	98	1	1	no
add_piano3	97	1.5	1.5	no
add_piano4	95	3	2	no
add_piano5	1	98	1	yes
add_snare1	94	1	5	no
add_snare2	94	1	5	no
add_snare3	94	1	5	no
add_snare4	94	1	5	no
add_snare5	94	1	5	no
		Total Correct Predictions:	6 / 15	

Table 9: Results of first testing round of 'add' files with trained MLP Classifier

Additional Sample Testing – 'conv' files

Datasets: guitar_and_piano_set_old.csv, guitar_and_snare_set_old.csv, piano_and_snare_set_old.csv
Different datasets used depending on convolution construction

Sample Tested	Guitar (%)	Piano (%)	Snare (%)
conv_guitar_piano1	37	63	-
conv_piano_guitar1	2	98	-
conv_guitar_piano2	63	37	-
conv_piano_guitar2	83	17	-
conv_guitar_snare1	94	-	6
conv_snare_guitar1	95	-	5
conv_guitar_snare2	4	-	96
conv_snare_guitar2	3	-	97
conv_piano_snare1	-	96	4
conv_snare_piano1	-	97	3
conv_piano_snare2	-	95	5
conv_snare_piano2	-	95	5

Table 10: Results of first testing round of 'conv' files with trained MLP Classifier

Additional Sample Testing – 'comb' files

Datasets: guitar_and_piano_set_old.csv, guitar_and_snare_set_old.csv, piano_and_snare_set_old.csv
Different datasets used depending on combination construction

Sample Tested	Guitar (%)	Piano (%)	Snare (%)
comb_guitar_piano1.1	97	3	-
comb_guitar_piano1.2	98	2	-
comb_guitar_piano1.3	2	98	-
comb_piano_guitar1.1	97	3	-
comb_piano_guitar1.2	2	98	-
comb_piano_guitar1.3	97	3	-
comb_guitar_piano2.1	97	3	-
comb_guitar_piano2.2	97	3	-
comb_guitar_piano2.3	24	76	-
comb_piano_guitar2.1	3	97	-
comb_piano_guitar2.2	2	98	-
comb_piano_guitar2.3	97	3	-
comb_guitar_snare1.1	97	-	3
comb_guitar_snare1.2	97	-	3
comb_guitar_snare1.3	96	-	4
comb_snare_guitar1.1	4	-	96
comb_snare_guitar1.2	4	-	96
comb_snare_guitar1.3	95	-	5
comb_guitar_snare2.1	97	-	3
comb_guitar_snare2.2	97	-	3
comb_guitar_snare2.3	63	-	37
comb_snare_guitar2.1	2	-	98
comb_snare_guitar2.2	3	-	97
comb_snare_guitar2.3	97	-	3
comb_piano_snare1.1	-	8	92
comb_piano_snare1.2	-	64	36
comb_piano_snare1.3	-	97	3
comb_snare_piano1.1	-	6	94
comb_snare_piano1.2	-	2	98
comb_snare_piano1.3	-	2	98
comb_piano_snare2.1	-	97	3
comb_piano_snare2.2	-	97	3
comb_piano_snare2.3	-	92	8
comb_snare_piano2.1	-	2	98
comb_snare_piano2.2	-	2	98
comb_snare_piano2.3	-	90	10

Table 11: Results of first testing round of 'comb' files with trained MLPClassifier

would be better for the ‘conv’ and ‘comb’ files to be tested on systems trained to recognise just the two instruments they were created from, in order to remove any unnecessary variables. I decided not to include combinations of all three instruments partly because the number of created combination files was already significant, but also because I was fairly confident the two-instrument files would provide enough information about which sections of the spectrograms were valued most highly, if any.

For these additional rounds of testing, I removed the train/test split of the training data, instead including the full 300 samples for the system to be trained on. In order to test the system’s label prediction for one sample, I made use of the ‘predict_proba’ method as indicated in the documentation, which returns a list of probabilities indicating how confident the system is for each of the possible labels. I decided to present these values as relative percentages, rounding them to whole numbers where possible. The results of these tests can be seen in tables 9, 10 and 11.

3.2.4 Improving the Training Data, Second Round of Additional Tests & Comments on Results

Even when recording the results of just the ‘add’ testing, it was apparent there was a problem. I had decided to continue and run both the ‘conv’ and ‘comb’ tests for completion’s sake but, while the results of these seemed interesting, it was clear that changes to the training set would have to be made. The system had incorrectly labelled 9 of the 15 ‘add’ samples, believing almost all of them to be guitars (table 9).

This confirmed my fears that the piano and snare samples had indeed been too homogenous, whereas the guitar samples were clearly far more varied – enough so that the system had come to judge any sound sample as a guitar if it did not conform specifically to the timbral characteristics of the limited piano and snare sounds it had been exposed to. The MLPClassifier had entirely failed to learn broader information about what differentiated guitar, piano and snare sounds.

I now resigned myself to the task of recording more piano and snare samples to be included in the training data. The former were easy enough to obtain – I firstly included the five 'add' samples, changing their names to 'piano_sam[1-5].wav', and recorded fourteen more from my keyboard and nine from a software piano in 'Logic', my primary digital audio workstation. When recording these, I made an effort to include a good number of notes in the piano's lower registers, as I have previously stated that Stéphane Thunus's samples (which still make up the bulk of the training set) are all relatively high-pitched.

For the snare samples, along with including the five 'add' samples in the same way (named 'snare_key[1-5].wav) I recorded nineteen from the in-built drum machines in Logic. I was also delighted to be given an extended live snare track from my father – recorded by Steve Crow, the drummer of his Eagles Tribute Band – from which I edited and saved eighteen more samples.

Desiring to keep the total number of samples at 300, I removed an equal number of the previous piano and snare samples to account for the new ones I had recorded – putting a selection of five aside from each instrument again, for a new round of 'add' testing, re-

naming the files as necessary. I did not change the 'conv' or 'comb' sets to include some of the new samples, partly because of the laborious nature of creating those sets, but mainly because I felt the results would still be valid even with the new training data – which I named 'revised_training_set.csv', updating the instrument pair training sets as well, appending their file names with '_old' as can be seen in tables 10 and 11.

Performing the initial accuracy tests once again with this new dataset, I was frustrated to discover that the score had dropped considerably – but only the test score, not the train score. Changing the parameters in various ways seemed to have little effect, although changing the learning_rate_init to 0.003 seemed to make a slight improvement. However, the test score was still frequently dropping to or below 90%, as can be seen in table 12. I can only attribute this significant drop in test accuracy to the training data possibly being more difficult as the updated samples contained far greater sonic variety within their instrumental groups – although I am at a loss to explain this against the now-perfect training accuracy across the board.

Dataset:	revised_training_set.csv	
Notes:	Learning_rate_init set to 0.003	
	Training Set Accuracy	**Test Set Accuracy**
Test 1:	1	0.916667
Test 2:	1	0.95
Test 3:	1	0.883333
Test 4:	1	0.933333
Test 5:	1	0.9
Test 6:	1	0.933333
Test 7:	1	0.95
Test 8:	1	0.933333
Test 9:	1	0.85
Test 10:	1	0.966667
Mean:	1	0.9216666

Table 12: Accuracy results of revised training set.

I proceeded to repeat the three rounds of testing for the ‘add’, ‘conv’ and ‘comb’ files with the updated training datasets – the results of which can be seen in tables 13, 14 and 15.

It came as something of a relief that the system appeared to have been able to far better learn the differences between instruments from this revised training set. The ‘add’ test revealed a perfect prediction rate, correctly labelling all fifteen files (table 13). Only one of these was correctly labelled with less than 98% certainty – ‘add_piano2’, which had been labelled as a piano with 62% certainty, the second-highest percentage being 37% for guitar.

When listening to all of the ‘add’ samples in turn, ‘add_piano2’ is certainly the lowest-pitched of them, which perhaps indicates that even with my improved training data the system still partially defines a piano note as having a higher fundamental pitch than a guitar. For visual comparison I captured the partial spectrograms of ‘add_guitar1’, ‘add_piano2’ and ‘add_piano3’ (only including the lower-right 24x180 pixels as this is all the system is being trained on), which can be seen in images 29, 30 and 31.

Comparing the intensity of the three spectrograms at different horizontal levels, we can immediately see evidence of the different fundamental frequencies of the samples – as both ‘add_guitar1’ and ‘add_piano2’ contain greater pixel intensity in the lowest rows than ‘add_piano3’. However, I think an additional potential explanation for the system’s slight uncertainty over the labelling of ‘add_piano2’ lies in the section roughly half-way up each spectrogram. In ‘add_piano3’, there are multiple rows in this section that are

Additional Sample Testing – 'add' files (second attempt)

Dataset: revised_training_set.csv

Sample Tested	Guitar (%)	Piano (%)	Snare (%)	Correct Prediction?
add_guitar1	98	1	1	yes
add_guitar2	98	1	1	yes
add_guitar3	98	1	1	yes
add_guitar4	98	1	1	yes
add_guitar5	98	1	1	yes
add_piano1	< 1	99	< 1	yes
add_piano2	37	62	1	yes
add_piano3	< 1	99	< 1	yes
add_piano4	< 1	99	< 1	yes
add_piano5	< 1	99	< 1	yes
add_snare1	< 1	< 1	99	yes
add_snare2	< 1	< 1	99	yes
add_snare3	< 1	< 1	99	yes
add_snare4	< 1	< 1	99	yes
add_snare5	< 1	< 1	99	yes
		Total Correct Predictions:		15 / 15

Table 13: Results of second training round of 'add' files with re-trained MLP Classifier

Additional Sample Testing – 'conv' files (second attempt)

Datasets: guitar_and_piano_set.csv,
guitar_and_snare_set.csv, piano_and_snare_set.csv
Different datasets used depending on convolution construction

File Tested	Guitar (%)	Piano (%)	Snare (%)
conv_guitar_piano1	6	94	-
conv_piano_guitar1	2	98	-
conv_guitar_piano2	4	96	-
conv_piano_guitar2	4	96	-
conv_guitar_snare1	84	-	16
conv_snare_guitar1	90	-	10
conv_guitar_snare2	7	-	93
conv_snare_guitar2	4	-	96
conv_piano_snare1	-	95	5
conv_snare_piano1	-	96	4
conv_piano_snare2	-	90	10
conv_snare_piano2	-	92	8

Table 14: Results of second testing round of 'conv' files with re-trained MLP Classifier

Additional Sample Testing – 'comb' files (second attempt)

Datasets: guitar_and_piano_set.csv, guitar_and_snare_set.csv, piano_and_snare_set.csv

Different datasets used depending on combination construction

Sample Tested	Guitar (%)	Piano (%)	Snare (%)
comb_guitar_piano1.1	88	12	-
comb_guitar_piano1.2	97	3	-
comb_guitar_piano1.3	4	96	-
comb_piano_guitar1.1	97	3	-
comb_piano_guitar1.2	4	96	-
comb_piano_guitar1.3	28	72	-
comb_guitar_piano2.1	94	6	-
comb_guitar_piano2.2	95	5	-
comb_guitar_piano2.3	64	36	-
comb_piano_guitar2.1	3	97	-
comb_piano_guitar2.2	2	98	-
comb_piano_guitar2.3	13	87	-
comb_guitar_snare1.1	97	-	3
comb_guitar_snare1.2	95	-	5
comb_guitar_snare1.3	2	-	98
comb_snare_guitar1.1	2	-	98
comb_snare_guitar1.2	2	-	98
comb_snare_guitar1.3	90	-	10
comb_guitar_snare2.1	97	-	3
comb_guitar_snare2.2	97	-	3
comb_guitar_snare2.3	10	-	90
comb_snare_guitar2.1	2	-	98
comb_snare_guitar2.2	2	-	98
comb_snare_guitar2.3	97	-	3
comb_piano_snare1.1	-	3	97
comb_piano_snare1.2	-	14	86
comb_piano_snare1.3	-	97	3
comb_snare_piano1.1	-	4	96
comb_snare_piano1.2	-	2	98
comb_snare_piano1.3	-	2	98
comb_piano_snare2.1	-	97	3
comb_piano_snare2.2	-	97	3
comb_piano_snare2.3	-	67	33
comb_snare_piano2.1	-	2	98
comb_snare_piano2.2	-	2	98
comb_snare_piano2.3	-	68	32

Table 15: Results of second testing round of 'comb' files with re-trained MLP Classifier

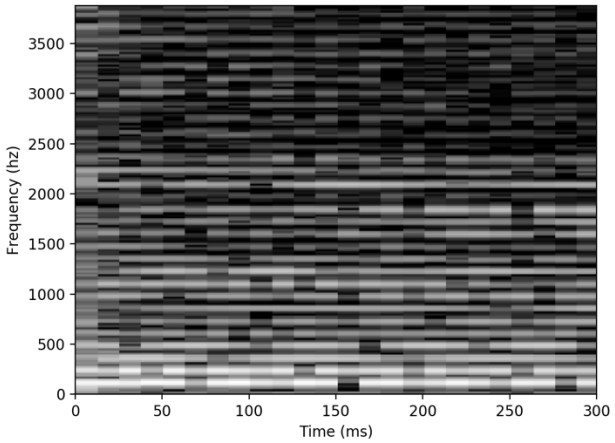

Image 29: Partial spectrogram of 'add_guitar1' sample.

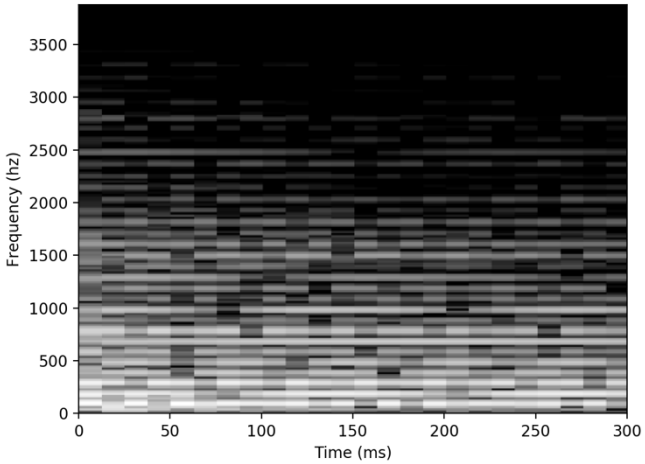

Image 30: Partial spectrogram of 'add_piano2' sample.

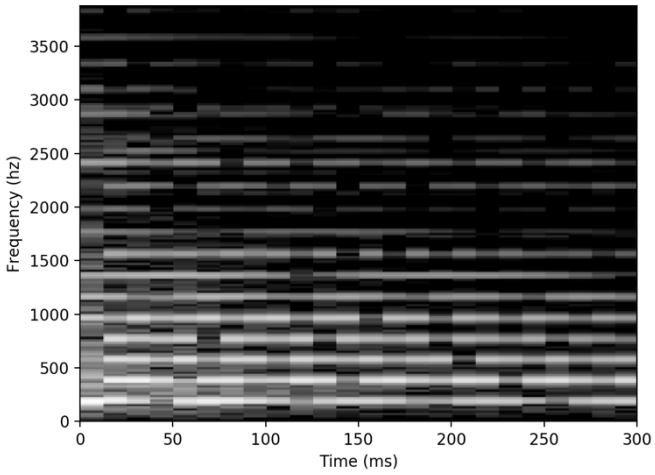

Image 31: Partial spectrogram of 'add_piano3' sample.

black almost all the way along, whereas in both ‘add_guitar1’ and ‘add_piano2’ this section is noticeably busier. Perhaps it is this particular distribution of overtones, making up the timbre of ‘add_piano2’ that are causing the system to partially consider it guitar-like?

Moving on to the ‘conv’ tests (table 14), the results are perhaps less simple to analyse while being far more intriguing. When I conceived of the test, I had wondered if the carrier sample would be decisive in the system’s labelling of the convolutions, as it determined the pitch of the result. However, this did not seem to be the case – the results being split straight down the middle with the system labelling six of the twelve convolutions as their carrier instrument. This indicates that pitch alone is not a major consideration in the system’s classification process, an encouraging result.

Glancing at the results, the only convolution pair that received exactly the same results regardless of which sample was used as carrier were ‘conv_guitar_piano2’ & ’conv_piano_guitar2’. Listening to the audio files and examining the spectrograms (images 32 and 33) provided some explanation for this, as they sounded and looked almost identical. This was clearly due to the original samples (‘guitar_ste20’ and ‘piano_sat1’) being notes of the same pitch, something I had not realised when originally selecting them.

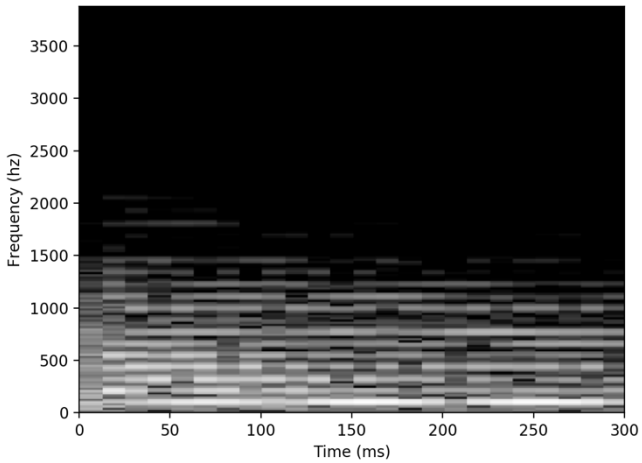

Image 32: Partial spectrogram of 'conv_guitar_piano2' sample.

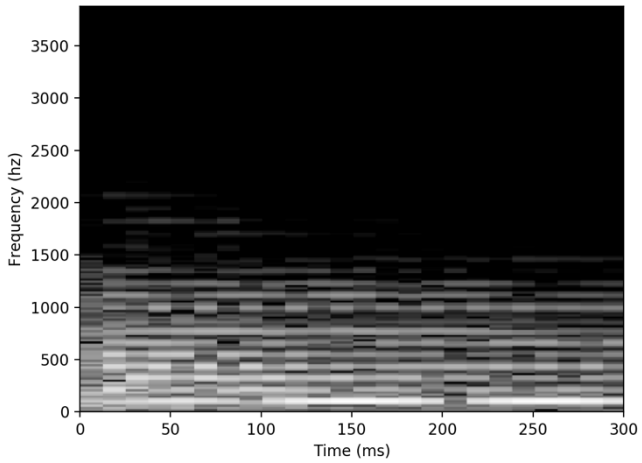

Image 33: Partial spectrogram of 'conv_piano_guitar2' sample.

Other than this pair, all convolutions had marginally different results depending on the carrier, although every pair received the same overall label. Interestingly, the system seemed to be recognising piano-like sounds far more strongly in the convolutions – every one that was partially comprised of a piano sample was labelled as such. I think I have an

explanation for this: the convolution process involves only retaining the shared frequencies between the samples, so we can deduce that all convolutions would almost certainly contain a sparser frequency spectrum as a result – many having been cut out due to only being present in one sample.

Even from the relatively small number of spectrograms I had viewed during my early construction of the training datasets, it had been immediately apparent that the piano samples contained far less dense frequency components than the guitar or snare samples. As snare drums are by default more 'noisy' sound sources than more pitch-based instruments, it should come as no surprise that their spectrograms contain a busier frequency spectrum. When it comes to the guitar samples however, some educated guesses are required.

I believe that the relative density of these when compared with the piano samples is due to the wider, more organic array of sources used in constructing the guitar training set. Although the majority of the piano samples are from an organic source (those provided by Stéphane Thunus), these are all higher-register sounds played rather softly (as previously stated), meaning that they have ended up far cleaner than live recordings often are – the thinner strings in the upper registers of the piano producing less distortion or 'beating' between them than with lower notes. In addition, the samples I recorded from my keyboard and through 'logic' are digital in nature, meaning that by default these are likely to be more clinically constructed and less noisy.

The guitar samples in contrast come from a much broader range of sound sources, my samples having all been recorded from a variety of real guitars, each with varying tones

and timbres. In addition, the electric guitar samples from Stéphane Thunus have some very slight distortion applied which – although still providing a mostly ‘clean’ tone – would obviously blur their frequency components somewhat, broadening the horizontal pixel intensity of the spectrograms. Generating some spectrograms for visual analysis quickly confirmed this hypothesis.

As the frequency spectrums of the convolution spectrograms are sparser and the instrument with the narrowest spread of frequency components is the piano (at least with respect to the sample banks I have gathered for this project), it is therefore unsurprising that the system seems pre-disposed to recognise most convolutions as piano-like.

The two pairs of guitar/snare convolutions are interesting to consider for different reasons. The first pair (‘conv_guitar_snare1’ and ‘conv_snare_guitar1’) are labelled as guitars with high confidence, while the other (‘conv_guitar_snare2’ and ‘conv_snare_guitar2’) are labelled as snares with even higher confidence. Listening to the first pair does not present any immediate evidence for why they have both been labelled as guitars, in fact when the snare is used as the carrier the resultant convolution sounds remarkably like a particularly resonant ‘rimshot’ hit, still fundamentally noisy and not pitch-based. Considering the spectrograms for both (images 34 and 35), I can only conclude that the slightly more ‘deliberate’ spread of frequency spikes up the rows of pixels is the cause. Despite their sound, the samples appear to contain enough harmonic content for the system to perceive them as more guitar-like than snare-like.

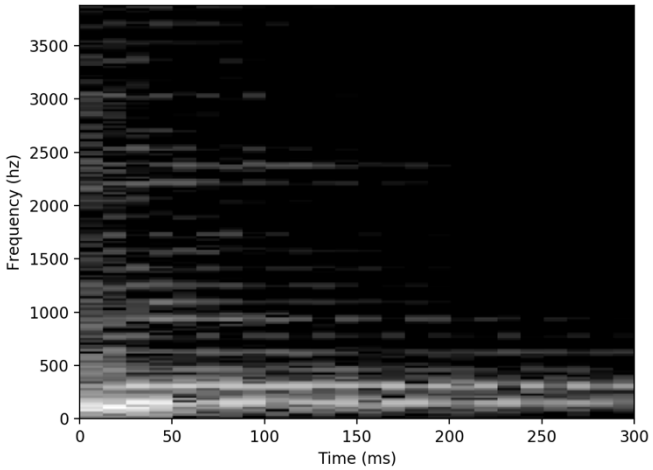

Image 34: Partial spectrogram of 'conv_guitar_snare1' sample.

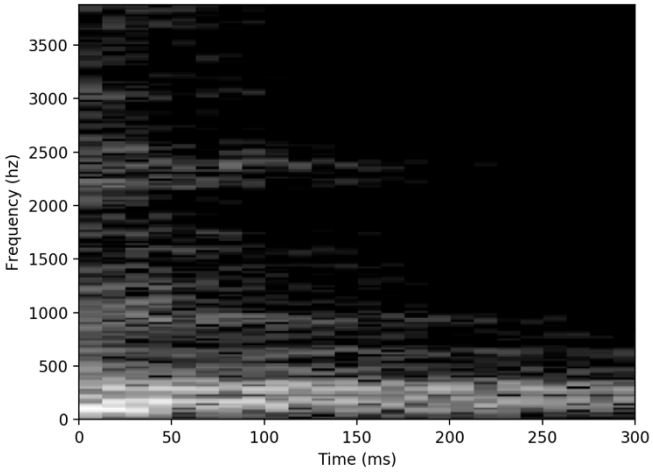

Image 35: Partial spectrogram of 'conv_snare_guitar1' sample.

With the latter pair of guitar/snare convolutions, once again my ears cannot perceive any immediate indications of the reason for their snare labels, both sounding practically identical and with a clear pitch – almost like short 'pizzicato' plucks of a guitar string. The spectrograms (images 36 and 37) however, display a denser array of frequencies than the sounds themselves indicate, which is my only explanation for their snare labels.

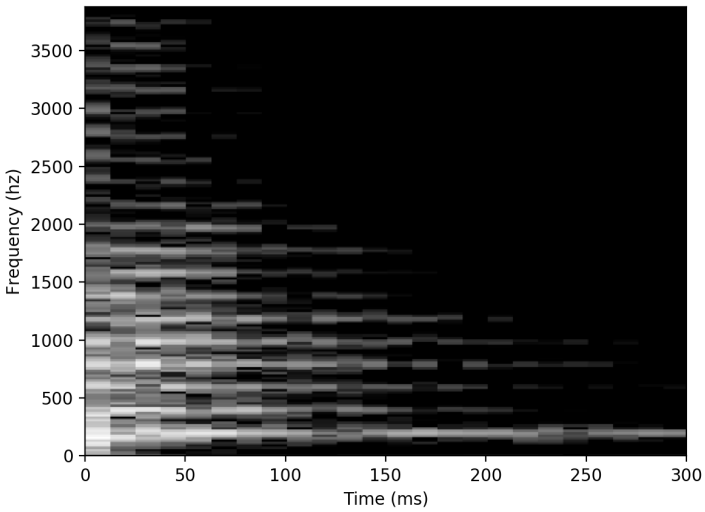

Image 36: Partial spectrogram of 'conv_guitar_snare2' sample.

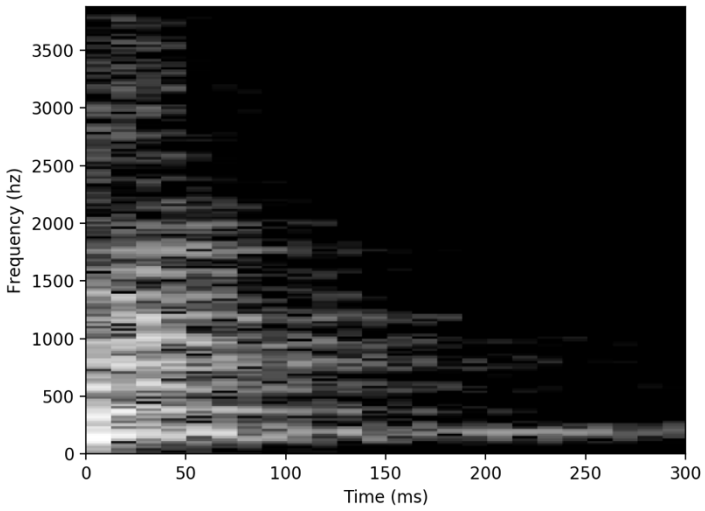

Image 37: Partial spectrogram of 'conv_snare_guitar2' sample.

The only results that still remained to be analysed were those of the 'comb' tests (table 15). As these were created from thirds of the full spectrograms joined together in various permutations, I imagined that analysing the results would likely give some insight into

which sections of the spectrograms the system perceived as being more indicative of a particular instrument, if any.

As mentioned earlier, the .1 .2 and .3 at the end of each combination file name indicates which position the minority instrument's section is placed in the order of thirds. These begin with it being placed last, moving forwards one section each time – for example 'comb_guitar_piano1.1' contains the first two sections of the guitar spectrogram first, ending with the third section of the piano, whereas 'comb_guitar_piano1.3' begins with the first piano section followed by the second and third guitar sections.

From the results, it is clear that the first spectrogram section (containing the first 60 frequency bins) holds considerably more weight when determining the overall label. Twenty-seven of the thirty-six 'comb' files were labelled according to the first section instrument, including instances when this was the only section of that instrument.

Attempting to explain these findings was not difficult. Once again casting an eye over some additional generated spectrograms, it was easy to see that most appear to contain the bulk of their intensity in the lowest third of the picture. As this fact had not really struck me until this point, I was initially confused, before I remembered that the spectrograms I had generated were displaying the frequency spectrum with linear scaling, not logarithmic. The audible frequency spectrum ranges from 20hz to 20khz, but due to the logarithmic scaling of frequency in relation to pitch, the halfway-point on an imaginary keyboard spanning this entire spectrum would not be at 10khz, but closer to 1khz. I had examined the Jitter code that generated the spectrograms but had been unable to determine how to change the scaling to logarithmic.

As discussed in section 3.1.3, the timbral characteristics of many musical notes involve the balance of harmonics, with the first eight overtones of a note with fundamental frequency 110hz extending up to 990hz. In a logarithmically-scaled spectrogram, this would extend virtually to the horizontal centre, however in the linear spectrograms I have been working with, it does not even reach to the top of the lowest of the 60-pixel high sections, which from some additional testing revealed it contained frequencies up to around 1.3khz.

Although the snare samples are not pitch-based and therefore cannot be so easily explained by harmonic balance, both the guitar and piano samples are primarily harmonic sounds. The highest pitched piano note included in my dataset was F#6 – approximately 1480hz – while the highest pitched guitar note was C6 – approximately 1047hz. Most of the samples are of considerably lower pitches than these, however. Even with the higher notes in the dataset, the fundamental frequency (typically the most intense harmonic in terms of amplitude) and – for most of these higher samples – the first overtone still remained within the lowest 60 frequency bins of the spectrograms.

The upshot of this is that the vast majority of important harmonic information for most of my samples (certainly the harmonics of highest amplitude) is actually contained in the lower third of each saved spectrogram. This served to explain the system's unbalanced bias towards this third in determining instrument label.

This understanding of one of the factors involved in how the neural network makes its choice of labels is particularly useful for planning construction of future, more advanced analysis systems. More accurate logarithmic frequency resolution would likely be an

asset in helping differentiate between instrumental sounds where the harmonic content is closer together, or which do not include frequencies above certain thresholds – many digitally synthesised sounds, for instance.

Summary and Conclusions

In this paper I have explored a range of AI-related topics, surveying relevant techniques and implementations that have developed since the initial conception of neural networks in the 1940s. These different branches of AI research have been shown to be not only of statistical and computational value through learning to classify data and recognise patterns, but also capable of generating new data, based on learned understanding and enabling various artistic AI endeavours.

I have recounted the journey of my conception and design of an analysis and classification machine learning system for instrumental sound samples through spectrogram visualisations – which could serve as a foundational step in constructing a potential future compositional system of far greater complexity. As part of this journey, I have undergone significant growth as a practitioner, gaining invaluable skills in Python programming and expanding my knowledge of the Deep Learning field in general, along with gaining valuable long-term contacts. Additionally, the work has resulted in the following contributions:

- Demonstrated a successful implementation of a simple neural network in Max/MSP, built from the ground up and distilled into video tutorials which have benefitted the online community.
- Resulted in construction of a bespoke dataset of instrumental samples for machine learning that can be supplemented in the future and distributed for related research.
- Demonstrated a process by which CSV files of samples may be generated from spectrograms in Max/MSP and transferred to Python.

- Provided practical evidence that AI systems can be trained to differentiate between different musical instruments purely from analysis of a small portion of spectrogram amplitude values.
- Collected data on how the trained system responds to various additional tests & trials, which may provide insight into aspects of the system's learning process & how sounds are classified, and which may assist in defining subsequent tests or troubleshooting.

Even if these achievements do not end up serving as the first steps in a much larger project, there is a great deal of room for future work. Some areas I could potentially focus on include:

- Adding more instruments to the dataset, as well as increasing the number & sonic variety of samples for existing instruments.
- Constructing an equivalent spectrogram generation process in Python to enable CSV files featuring larger usable portions of sample spectrograms. Once this is complete, expanding on the current method of generating spectrograms to also include phase values.
- Implementing a more sophisticated Deep Learning system to better handle the subsequent increased data complexity.
- Expanding on this system to include generative processes, either through GANs or an equivalent architecture, to initially generate new samples of individual instruments.
- Through analysis of the DeepDream process and available code, enabling extraction of learned features to facilitate creation of 'dreamed audio'.

By way of some final observations, if I were to go through the process again I would have preferred to have completed more of my AI survey at an earlier point in my journey. Particularly, I would have liked my research into convolutional neural networks and Magenta's NSynth to have occurred far sooner, as I feel with some additional time spent on them they might have had more fundamental impact in shaping my approach to the project.

If I had learned enough about CNNs to be confident enough to attempt an implementation of one, I feel certain it would have served as a superior form of classification network over the MLP I ended up using. Likewise, the primary practical application of NSynth – that of timbral combinations of different instrumental sounds – is remarkably analogous to my DeepDream-inspired ideas. The rather impenetrable complexity (as I perceived it) of the WaveNet paradigm used in NSynth's construction served to dissuade me from deeper research into the system, a mental obstacle that I think may have prevented me from broadening the overall scope of my ideas.

However, my retrospective awareness of the potential importance of these topics does not diminish the completed work – after all if I am to continue development of my ideas in this field there will be ample time to consider their inclusion in greater depth in the future. As it stands, this project and the journey I took to complete it have had a transformative effect on my interests, as well as my perception of my own abilities. I have achieved far more than I would have imagined when joining the ResM course, and I look forward immensely to all the work yet to come.

List of Sources

AIVA. 2019 [no.1]. 'Aiva' *aiva.ai* [online] Available at: https://www.aiva.ai/ [accessed 30/9/19]

AIVA. 2019 [no.2]. 'Upload Influence – Demo' *youtube.com* [online] Available at: https://www.youtube.com/watch?v=-4b9dqvhSAQ [accessed 30/9/19]

BENSON, Andrew. 2017. 'Content You Need: ml.star', *cycling74.com* [online] Available at: https://cycling74.com/articles/content-you-need-ml%C2%B7star [accessed 10/1/19]

BERGSTRA, James, Guillaume Desjardins, Pascal Lamblin, Yoshua Bengio. 2009. *Quadratic Polynomials Learn Better Image Features.* [online] Available at: http://www.iro.umontreal.ca/~lisa/publications2/index.php/attachments/single/205 [accessed 10/1/19]

BETA, Andy. 2016. '16 Essential Vocoder Songs', *pitchfork.com* [online] Available at: https://pitchfork.com/thepitch/986-16-essential-vocoder-songs/ [accessed 10/1/19]

BHATIA, Richa. 2018. 'Neural Networks Do Not Work Like Human Brains – Let's Debunk The Myth' *analyticsindiamag.com* [online] Available at: https://www.analyticsindiamag.com/neural-networks-not-work-like-human-brains-lets-debunk-myth/ [accessed10/1/19]

BILES, John. 1994. *GenJam: A Genetic Algorithm for Generating Jazz Solos.* [online] Available at: https://www.researchgate.net/publication/2342018_GenJam_A_Genetic_Algorithm_for_Generating_Jazz_Solos [accessed 30/9/19]

BITA. 2016. 'Partial_Fit Scikit Learn's MLPClassifier' *stackoverflow.com* [online] Available at: https://stackoverflow.com/questions/35756549/partial-fit-sklearns-mlpclassifier [accessed 1/10/19]

BÖHM, Volker. 2013. 'Spectrogram that behaves like Waveform~? – gets data from buffer~ instead of signs' [response] *cycling74.com* [online] Available at: https://cycling74.com/forums/spectrogram-that-behaves-like-waveformgets-data-from-buffer-instead-of-signs [accessed 10/1/19]

BOSTONDYNAMICS. 2017. 'What's new, Atlas?' *youtube.com* [online] Available at: https://www.youtube.com/watch?v=fRj34o4hN4I [accessed 10/1/19]

BUDA, Mateusz, Atsuto MAKI and Maciej A. MAZUROWSKI. 2018. 'A Systematic Study of the Class Imbalance Problem in Convolutional Neural Networks' *Neural Networks*, 106, 249-259

BULLOCK, Jamie. 2014. 'ml-lib' *github.com* [online] Available at: https://github.com/irllabs/ml-lib [accessed 10/1/19]

BUTCHER, Mike. 2019. 'It looks like TikTok has acquired Jukedeck, a pioneering AI UK startup', *techcrunch.com* [online] Available at: https://techcrunch.com/2019/07/23/it-looks-like-titok-has-acquired-jukedeck-a-pioneering-music-ai-uk-startup/ [accessed 30/9/19]

CHENG, Jacqui. 2009. 'Virtual composer makes beautiful music—and stirs controversy', *arstechnica.com* [online] Available at: http://arstechnica.com/science/2009/09/virtual-composer-makes-beautiful-musicand-stirs-controversy/ [accessed 10/1/19]

COLLINS, Nick. 2010. *Introduction to Computer Music*. Chichester: John Wiley & Sons Ltd.

COMPUTERPHILE. 2016. 'Neural Network that Changes Everything', *youtube.com* [online] Available at: https://www.youtube.com/watch?v=py5byOOHZM8 [accessed 10/1/19]

COMPUTERPHILE. 2017. 'Generative Adversarial Networks (GANs)' *youtube.com* [online] Available at: https://www.youtube.com/watch?v=Sw9r8CL98N0 [accessed 10/1/19]

COPE, David. 1996. *Experiments in Musical Intelligence*. Madison: A-R Editions

COPE, David. 2000. *The Algorithmic Composer*. Madison: A-R Editions

COPE, David. 2019. 'David Cope', *ucsc.edu* [online] Available at: http://artsites.ucsc.edu/faculty/cope/experiments.htm [accessed 10/1/19]

COPELAND, J. (ed.) 2004. *The Essential Turing*. Oxford: Clarendon Press.

CYCLING74. 2019. 'Cycling74: Tools for Sound, Graphics and Interactivity', *cycling74.com* [online] Available at: https://cycling74.com/ [accessed 10/1/19]

DAFT PUNK. 2013. *Random Access Memories.* Columbia Records.

DEEP DREAM GENERATOR. 2019. 'Deep Dream Generator: Human/AI Collaboration' *deepdreamgenerator.com* [online] Available at: https://deepdreamgenerator.com/ [accessed 10/1/19]

DUDE837. 2010. 'Delicious Max/MSP Tutorial 4: Vocoder', *youtube.com* [online] Available at: https://www.youtube.com/watch?v=4feOFLX6238 [accessed 10/1/19]

ENGEL, Jesse, Cinjon RESNICK, Adam ROBERTS, Sander DIELEMAN, Douglas ECK, Karen SIMONYAN and Mohammad NOROUZI. 2017. 'Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders' *arxiv.org* [online] Available at: https://arxiv.org/pdf/1704.01279.pdf [accessed 10/1/19]

FERNÁNDEZ, Jose D. and Francisco VICO. 2013. 'AI methods in algorithmic composition: A comprehensive survey', *Journal of Artificial Intelligence Research*, 48, 513-582

FRIEND, Tad. 2018. 'How Frightened Should we be of A.I.?' *newyorker.com* [online] Available at: https://www.newyorker.com/magazine/2018/05/14/how-frightened-should-we-be-of-ai [accessed 10/1/19]

GILES, Jazer. 2018. 'Projects' *jazergiles.com* [online] Available at: https://www.jazergiles.com/projects [accessed 10/1/19]

GOOGLE AI. 2019. 'Magenta', *ai.google* [online] Available at: https://ai.google/research/teams/brain/magenta/ [accessed 10/1/19]

GOVUK. 2017. 'Industrial Strategy: building a Britain fit for the future' *gov.uk* [online] Available at: https://www.gov.uk/government/publications/industrial-strategy-building-a-britain-fit-for-the-future [accessed 10/1/19]

GRATTAN-GUINNESS, Ivor. 2008. 'Jean-Baptiste Joseph Fourier'. In GOWERS, Timothy, June BARROW-GREEN and Imre LEADER (eds.). *The Princeton Companion to Mathematics*. Princeton: Princeton University Press, 755.

HAUGELAND, John. 1985. *Artificial Intelligence: The Very Idea*. Cambridge: MIT Press

HOLLEY, Peter. 2018. 'Elon Musk: To avoid becoming like monkeys, humans must merge with machines', *washingtonpost.com* [online] Available at: https://www.washingtonpost.com/technology/2018/11/26/elon-musk-avoid-becoming-like-monkeys-humans-must-merge-with-machines/?noredirect=on&utm_term=.a33e1a0c3851 [accessed 10/1/19]

HOWARD, Rick. 2017. 'Alan Turing: Father of the Computer Age', *securityroundtable.org* [online] Available at: https://www.securityroundtable.org/alan-turing-father-computer-age/ [accessed 10/1/19]

HUMANITY+. 2010. 'Has Emily Howell passed the musical Turing test?' *h+ media*. [online] Available at: http://hplusmagazine.com/2010/03/22/has-emily-howell-passed-musical-turing-test/ [accessed 10/1/19]

HURON, David. 2008. *Sweet anticipation: Music and the psychology of expectation*. Cambridge: Bradford Books

IBM. 2019. ‘Deep Blue’, *ibm.com* [online] Available at: https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/ [accessed 10/1/19]

IMAGE NET. 2017. ‘Large Scale Visual Recognition Challenge 2017’ *image-net.org* [online] Available at: http://image-net.org/challenges/LSVRC/2017/index [accessed 10/1/19]

JAMIROQUAI. 2017. *Automaton*. Virgin EMI Records.

JUKEDECK. 2019. *Jukedeck.com* [online] Available at: https://www.jukedeck.com/ [accessed 30/9/19]

KARPATHY, Andrej. 2014. ‘What I learned from competing against a ConvNet on ImageNet’ *Andrej Karpathy Blog* [online] Available at: http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/ [accessed 10/1/19]

KOSKO, Bart. 1994. *Fuzzy Thinking: The New Science of Fuzzy Logic*. London: Flamingo

LECUN, Yann, Corinna CORTES and Christopher J.C. BURGES. 2019. ‘The MNIST Database of Handwritten Digits’ *yann.lecun.com* [online] Available at: http://yann.lecun.com/exdb/mnist/ [accessed 10/1/19]

LIGHTHILL, Sir James. 1972. *Artificial Intelligence: A General Survey.* Available online at: http://www.chilton-computing.org.uk/inf/literature/reports/lighthill_report/p001.htm [accessed 10/1/19]

LIVING INTERNET. 2019. ‘Dartmouth Artificial Intelligence (AI) Conference’ *livinginternet.com* [online] Available at: https://www.livinginternet.com/i/ii_ai.htm [accessed 10/1/19]

LYON, Richard F. 2017. *Human and Machine Hearing.* New York: Cambridge University Press.

MAGENTA. 2017. ‘NSynth: Neural Audio Synthesis’ *magenta.tensorflow.org* [online] Available at: https://magenta.tensorflow.org/nsynth [accessed 10/1/19]

MALLAWAARACHCHI, Vijini. 2017. ‘Introduction to Genetic Algorithms – Including Example Code’ *towardsdatascience.com* [online] Available at: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3 [accessed 30/9/19]

MANNING, Peter. 2013. *Electronic and Computer Music* (4th ed.). New York: Oxford University Press.

MARCUS, Gary. 2013. ‘Hyping Artificial Intelligence, Yet Again’ *newyorker.com* [online] Available at: https://www.newyorker.com/tech/annals-of-technology/hyping-artificial-intelligence-yet-again [accessed 10/1/19]

MARSLAND, Stephen. 2009. *Machine Learning: An Algorithmic Perspective*. Boca Raton: Chapman & Hall/CRC

MATPLOTLIB. 2018. ‘Matplotlib’ *matplotlib.org* [online] Available at: https://matplotlib.org/index.html [accessed 10/1/19]

MCCARTHY, John, Marvin MINSKY, Nathaniel ROCHESTER and Claude SHANNON. 1955. *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence.* Available online at: http://raysolomonoff.com/dartmouth/boxa/dart564props.pdf [accessed 10/1/19]

MCCULLOCH, Warren S. and Walter PITTS. 1943. ‘A Logical Calculus of the Ideas Immanent in Nervous Activity’. *The Bulletin of Mathematical Biophysics*, 5(4), 115-133

MILES, Robert. 2019. ‘Robert Miles Home’ *youtube.com* [online] Available at: https://www.youtube.com/channel/UCLB7AzTwc6VFZrBsO2ucBMg [accessed at 10/1/19]

MINSKY, Marvin L. and Seymour A. PAPERT. 1969. *Perceptrons: An Introduction to Computational Geometry*. Cambridge: MIT Press

MIRANDA, Eduardo. 2001. *Composing Music with Computers*. Oxford: Focal Press

MITCHELL, Tom M. 1997. *Machine Learning*. New York: WCB/McGraw-Hill.

MORDVINTSEV, Alexander, Christopher OLAH and Mike TYKA. 2015 [no.1]. ‘Inceptionism: Going Deeper into Neural Networks’, *ai.googleblog.com* [online] Available at: https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html [accessed 10/1/19]

MORDVINTSEV, Alexander, Christopher OLAH and Mike TYKA. 2015 [no.2]. ‘DeepDream - a code example for visualizing Neural Networks’, *ai.googleblog.com* [online] Available at: https://ai.googleblog.com/2015/07/deepdream-code-example-for-visualizing.html [accessed 10/1/19]

MURRAY, Adam J. 2010. ‘max-csv-tools’, *github.com* [online] Available at: https://github.com/adamjmurray/max_csv_tools [accessed 10/1/19]

NAVE, Rod. 2019. ‘Timbre’, *hyperphysics* [online] Available at: http://hyperphysics.phy-astr.gsu.edu/hbase/Sound/timbre.html [accessed 10/1/19]

NIELSEN, Michael. 2018. ‘Chapter 6: Deep Learning’, *Neural Networks and Deep Learning* [online] Available at:

http://neuralnetworksanddeeplearning.com/chap6.html#recent_progress_in_image_recognition [accessed 10/1/19]

OGDEN, Gary. 2018. 'This new chapter of 'Harry Potter' written by a bot is absolutely hilarious nonsense' *shortlist.com* [online] Available at: https://www.shortlist.com/entertainment/books/ai-bot-harry-potter/337695 [accessed 10/1/19]

OLAZARAN, Mikel. 1966. 'A Sociological Study of the Official History of the Perceptrons Controversy' *Social Studies of Science,* 26(3), 611-659

PEARCE-DAVIES, Samuel. 2017. 'Max/MSP Neural Network Tutorials', *youtube.com* [online] Available at: https://www.youtube.com/playlist?list=PLnXdT5joUcb1GhPEQizT7NjyPzAM0Y46I [accessed 10/1/19]

PEREZ, Carlos E. 2017. *The Deep Learning AI Playbook*. Cambridge: Intuition Machine

PLURALSIGHT. 2019. 'Paths: Python' *pluralsight.com* [online] Available at: https://www.pluralsight.com/search?q=python [accessed 10/1/19]

PRESS, Gil. 2018. 'The Brute Force of IBM Deep Blue and Google DeepMind', *forbes.com* [online] Available at: https://www.forbes.com/sites/gilpress/2018/02/07/the-brute-force-of-deep-blue-and-deep-learning/#425e7e4b49e3 [accessed 10/1/19]

RADFORD, Alec, Luke METZ and Soumith CHINTALA. 2016. 'Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks' *arxiv.org* [online] Available at: https://arxiv.org/pdf/1511.06434.pdf [accessed 10/1/19]

RAVAL, Siraj. 2017. 'Which Activation Function Should I Use?' *youtube.com* [online] Available at: https://www.youtube.com/watch?v=-7scQpJT7uo [accessed 10/1/19]

ROWE, Robert. 2001. *Machine Musicianship*. Cambridge: MIT Press

SAMARASINGHE, Sandhya. 2007. *Neural Networks for Applied Sciences and Engineering*. Boca Raton: Auerbach Publications

SCHMIDHUBER, Jürgen. 2015. 'Deep Learning in Neural Networks: An Overview.' *Neural Networks,* 61, 85-117

SCIKIT LEARN. 2018 [no. 1]. 'MLPCLassifier' *scikit-learn.org* [online] Available at: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier [accessed 10/1/19]

SCIKIT LEARN. 2018 [no. 2]. ‘Neural Network Models (Supervised)’, *scikit-learn.org* [online] Available at: https://scikit-learn.org/stable/modules/neural_networks_supervised.html [accessed 10/1/19]

SINGH WALIA, Anish. 2017. ‘Activation Functions and it’s Types – Which is Better?’ *towardsdatascience.com* [online] Available at: https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f [accessed 10/1/19]

SPENCER-HARPER, Milo. 2015 [no.1]. ‘How to Build a Simple Neural Network in 9 Lines of Code’, *Medium.com* [online] Available at: https://medium.com/technology-invention-and-more/how-to-build-a-simple-neural-network-in-9-lines-of-python-code-cc8f23647ca1 [accessed 10/1/19]

SPENCER-HARPER, Milo. 2015 [no.2.]. ‘How to Build a Multi-Layered Neural Network in Python’, *Medium.com* [online] Available at: https://medium.com/technology-invention-and-more/how-to-build-a-multi-layered-neural-network-in-python-53ec3d1d326a [accessed 10/1/19]

STAMENOVIC, Marko. 2016. ‘audio-deepdream-tf’ *github.com* [online] Available at: https://github.com/markostam/audio-deepdream-tf [accessed 10/1/19]

TED. 2018. ‘How AI could compose a personalized soundtrack to your life | Pierre Barreau’, *youtube.com* [online] Available at: https://www.youtube.com/watch?v=wYb3Wimn01s [accessed 30/9/19]

TEDX TALKS. 2012. ‘GenJam's Journey: From Tech to Music: Al Biles at TEDxBinghamtonUniversity’ *youtube.com* [online] Available at: https://www.youtube.com/watch?v=rFBhwQUZGxg [accessed 30/9/19]

THE ROYAL INSTITUTION. 2017. ‘Artificial Intelligence, the History and the Future – with Chris Bishop’. *youtube.com* [online] Available at: https://www.youtube.com/watch?v=8FHBh_OmdsM [accessed 10/1/19]

THE SEMICOLON. 2017. ‘Neural Networks and Backpropagation Scikit Learn’, *youtube.com* [online] Available at: https://www.youtube.com/watch?v=X8SPO875mQY&index=24&list=PLnXdT5joUcb18oOW3DDH650dmbBlV3W2U [accessed 10/1/19]

THEODORIDIS, Sergios. 2015. *Machine Leaning: A Bayesion and Optimisation Perspective*. London: Academic Press.

VAN DEN OORD, Aaron, Sander DIELEMAN, Heiga ZEN, Karen SIMONYAN, Oriol VINYALS, Alex GRAVES, Nal KALCHBRENNER, Andrew SENIOR and Koray KAVUKCUOGLU. 2016 [no.1]. ‘WaveNet: A Generative Model for Raw Audio’, *arxiv.org* [online] Available at: https://arxiv.org/pdf/1609.03499.pdf [accessed 10/1/19]

VAN DEN OORD, Aaron, Sander DIELEMAN and Heiga ZEN. 2016 [no.2]. 'WaveNet: A Generative Model for Raw Audio', *deepmind.com* [online] Available at: https://deepmind.com/blog/wavenet-generative-model-raw-audio/ [accessed 10/1/19]

WALKER, Martin. 2005. 'Convolution Processing with Impulse Responses', *soundonsound.com* [online] Available at: https://www.soundonsound.com/techniques/convolution-processing-impulse-responses [accessed 10/1/19]

WOOD, Sidney. 2005. 'What are Formants?', *lu.se* [online] Available at: http://person2.sol.lu.se/SidneyWood/praate/whatform.html [accessed 10/1/19]

ZEIDENBERG, Matthew. 1990. *Neural Network Models in Artificial Intelligence*. Chichester: Ellis Horwood Ltd.

ZICARELLI, David. 2011. 'Max Mathews: An Appreciation', *cycling74.com* [online] Available at: https://cycling74.com/articles/max-mathews-an-appreciation/ [accessed 10/1/19]