

An investigation into the feasibility of real-time downbeat recognition using neural networks

Jack Hinton -147676

Supervisor: Gareth Jones
Second Marker: Gordon Dickers

Project submitted as part of the requirements for the award of BSc
Software Engineering

April 2018

DECLARATION FORM AND FORM OF CONSENT

DECLARATION

I, ...
declare that I am the sole author of this Project; that all references cited have been consulted; that I have conducted all work of which this is a record, and that the finished work lies within the prescribed word limits.

This work has not previously been accepted as part of any other degree submission.

Signed:

Date:

FORM OF CONSENT

I
hereby consent that my Project, submitted in candidature for the B.Sc. Software Engineering degree, if successful, may be made available for inter-library loan or photocopying (subject to the law of copyright), and that the title and abstract may be made available to outside organisations.

Signed:

Date:

ABSTRACT

This project explores the possibility of creating a system that can identify musical tempo and accurately isolate beats within a given piece of music in real time. This idea is being investigated to enquire as to the possibility of filling a niche that has been hitherto overlooked - digital systems that can tailor the tempo of the music they output in order to suit the user, rather than forcing the user to conform to their preset speed. In order to investigate this goal, a combination of neural networks and signal processing techniques are utilised to create a prototype system that can be used to estimate the feasibility of a system that can detect musical beats in real time.

Contents

DECLARATION FORM AND FORM OF CONSENT	2
DECLARATION	2
ABSTRACT	3
Contents	4
1. Introduction	8
1.1. Aim	8
1.2. Objectives	9
2. Research and development methodology	9
2.1. Research overview	9
2.2. Research process	9
2.3. Research philosophy	9
2.4. Research Methodologies	10
2.5. Data Collection	10
2.6. Development methodology	10
3. Research and review of literature	11
3.1. The concept, description, and function of a neural network.	11
3.2. Glossary of terms	13
3.2.1. Backpropagation	13
3.2.2. Bias	13
3.2.3. Convergence	13
3.2.4. Downsampling	13
3.2.5. Error	13
3.2.6. Forward pass	14
3.2.7. Loss	14
3.2.8. Optimisation function	14
3.2.9. Overfitting	14
3.2.10. Premature convergence	14
3.2.11. Sample rate	15
3.2.12. Vanishing/exploding gradient	15
3.2.13. Weight	15
3.3. Neural network architectures	15
3.3.1. The perceptron	16
	4

3.3.2. Multilayer perceptron	18
3.3.3. Convolutional neural network	19
3.3.4. Recurrent neural networks	20
3.3.5. Long Short Term Memory	21
3.3.6. Gated Recurrent Unit	22
3.4. Research into real time beat tracking software	24
3.4.1. Isolating dominant frequencies in base chord changes	24
3.4.2. Iteratively weighted summation of onset coincidence	24
3.4.3. Analysis of amplitude variance using convolutional neural networks	24
4. Design and implementation	25
4.1. Problem Formulation	25
4.2. Proposed system design considerations	25
4.3. Design	27
4.3.1. Segmentation	27
4.3.2. Chroma analysis	28
4.3.3. Onset detection	28
4.3.4. Melodic analysis	28
4.3.5. Low pass filter	29
4.3.6. Weighted vote system	29
4.4. Implementation	30
4.5. Reflection	30
4.5.1. Creation	30
4.5.2. Input formatting	31
4.5.3. Temporal metric	31
4.5.4. Sample generation	31
4.5.5. Memory usage	32
4.6. Testing and optimisation	33
4.6.1. Threading Fast Fourier Transforms	33
4.6.2. Testing run speed using either CPU or graphics card	33
4.6.3. Optimisation	34
4.7. Development timeline	35
4.7.1. Phase 1:	35
4.7.2. Phase 2:	35
4.7.3. Phase 3:	35
4.7.4. Phase 4:	36
4.7.5. Phase 5:	36
4.8. Comments on the implementation of the Gated Recurrent Unit class	37
4.8.1. Loss	37

4.8.2. Optimisation function	37
4.8.3. Forward pass	38
4.8.4. Values and initialisation	38
5. Analysis	40
5.1. Speed	40
5.2. Accuracy	40
5.3. Examination of the effect of number of hidden nodes on training speed and accuracy	41
5.4. Resource requirements	42
6. Conclusion	43
7. Evaluation	43
8. Recommendations for further research	44
8.1. Capsule networks	44
8.2. Intel Nervana chip	44
9. Code listings	45
9.1. Annotated Python code listing	45
9.1.1. Libraries	45
9.1.2. Reverse array of bits	46
9.1.3. CSV reader	46
9.1.4. Data return, type checking and downsampling	47
9.1.5. Nested list creation function	48
9.1.6. FFT function with absolute value return	48
9.1.7. Threaded iterating FFT function	50
9.1.8. Conversion of integer array to array of reversed bits	50
9.1.9. Sample creation function	51
9.1.10. Configuration: Generate FFT, load FFT values, or no action.	53
9.1.11. Configuration: Equalise list size or no action	53
9.1.12. Configuration: Generate and save training set, or no action.	54
9.1.13. Configuration: Load training set, or no action	54
9.1.14. Configuration: Generate and save test set, or no action	54
9.1.15. Configuration: Load test set, or no action	56
9.1.16. Configuration: Reduce training set size and equalise both inputs, or no action	56
9.1.17. GRU class definition: Initialisation function	57
9.1.18. GRU class definition: Forward pass function	58
9.1.19. Configuration: Initialise neural network, or load configuration from file	58
9.1.20. Configuration: Initialise variables larger than 2GB, or no action	59
9.1.21. GRU runtime: Initialise variables, session and minimisation function	60

9.1.22. GRU runtime: Training and running	61
9.2. Annotated C++ code listing	62
9.2.1. Setup: Standard IO and maths libraries	62
9.2.2. Setup: Audio processing libraries	62
9.2.3. C++ runtime: Error	63
9.2.4. C++ runtime: Main	63
Bibliography	67

1. Introduction

The overall aim of this project is to investigate the creation of a system that can accurately identify musical downbeats in real time. This would facilitate the creation of a drum machine which could be used to accompany a performer and adjust its tempo to match the performer's. This would allow for a more complex and nuanced solo performance as the tempo and time signature would then be able to be varied.

To accomplish this, a system has been proposed where modern data segmentation techniques are used to separate the music into a series of discrete units which will then be fed to a system that will analyse the sample for one of several markers denoting the presence or absence of a downbeat i.e. the accented, often first, beat of a musical bar. These systems will then return their output to a specialised neural network that will give a yes or no answer denoting the presence or absence of a downbeat in the sample analysed, based on data received.

The objectives required for completion of this project are the design of a rapid and consistent audio segmentation system and the feature extraction networks that will be used to denote downbeat likelihood within their respective domains of information.

Neural networks have been chosen to perform both the feature extraction and beat detection functions because they are trainable. This ameliorates the need to construct a robust and concrete definition of a musical downbeat that can be applied to all musical genres.

1.1. Aim

To investigate the feasibility of a system capable of both detecting the tempo of and accurately isolating downbeats within a piece of music in real time.

1.2. Objectives

Conduct a literature review on the use of neural networks in real-time environments.

Construct an audio segmentation system to divide the audio stream into discrete sections for analysis.

Design a system that may be used to accurately isolate downbeats within a piece of music.

Determine the feasibility of said system's use in a live music production environment.

2. Research and development methodology

2.1. Research overview

2.2. Research process

Initially, the feasibility of the proposed system will be investigated through document review. Subsequent to this, a prototype system will be developed that will be a representative proportion of the operations required for full functionality. Finally, timings will be taken from this program and combined with the findings of the document review.

2.3. Research philosophy

The research philosophy used in the pursuit of this goal is positivist and deductive and thus is not concerned with unquantifiable metrics like 'ease of use' or aesthetic considerations like the appearance of the user interface.

2.4. Research Methodologies

The methodology used in this investigation is developmental and thus the research strategies employed are experiment, software development, and document review. The aim of the experiment is to determine whether or not a real-time beat tracking system that possesses the requisite accuracy can be created.

2.5. Data Collection

The data will be collected from numerous sources, the primary one being the prototype program and experiments performed upon said program. The secondary sources used in the compilation of data necessary to maintain scientific rigour include academic papers, patents, journal articles, and software libraries.

2.6. Development methodology

The methodology used in development of the prototype was the Agile approach. Development thus followed an iterative approach, with newer versions of each software module being developed so as to enhance the functionality of their older incarnations. The software was written to be loosely coupled so as to facilitate easy replacement of each section.

3. Research and review of literature

3.1. The concept, description, and function of a neural network.

Classically, digital computers produce output that is based directly upon an interaction between provided input and their current programming. Any procedure to be attempted must be explicitly designed and implemented by a human operator before it can be performed. The use of computers in this way has lead to many advancements in fields where complex computations can be broken down into a larger number of simpler ones, including fields such as telecommunications, mathematics and physics.

This approach is well suited to tasks with variables, processes, and objectives that are easy to identify, but often suffers when it is applied to tasks that do not fit this description [1], [2].

In the case of handwriting analysis, for example, there exists a plethora of variables that must be taken into account including rotation, skew, mirroring, handedness and joining.

In these situations, a system conforming to the common model of 'input leading to programmed function resulting in output' may be impossible to create as no totally complete understanding of the processes required exists.

To combat this issue, a system was designed that more closely follows the mechanism found within the human brain - one using dynamically configured pathways that 'learn' the processes required to produce correct output through trial and error [3]. This type of system is commonly referred to as a 'neural network'.

A neural network differs from a typical computer process in several ways, key amongst them being the idea that the processes this system carries out are not programmed into it by an operator. This is accomplished by using a system of simple nodes that are linked together. Inputs and outputs within the system are modified by weights and biases within the network which are themselves reconfigured in response to the cumulative success or failure of the

previous outputs. [3] This system was designed to be analogous to the brain as it attempts the simulation of neurons, their respective axons and the formation of neural pathways [4].

Using this type of system, complex tasks such as musical composition or fluid flow analysis may be tackled. It is no longer necessary to understand the process by which the correct answer may be derived - it is now required only that examples of these answers can be rendered in a format comprehensible to a neural network.

Neural networks are not without flaws, however. The process of 'training' a neural network may take a large amount of time depending on the complexity of the input data, the architecture network itself and the method by which it is trained [5]. Furthermore a network may completely fail to arrive at an answer due to phenomena such as the vanishing gradient problem[6]. In order to prevent this from occurring, the initial conditions and mechanisms of action within a neural network must be rigorously tested. Even after taking this into account, it is nevertheless evident that systems of this description can enable feats of computing that were previously thought impossible[7].

3.2. Glossary of terms

3.2.1. Backpropagation

Backpropagation is the term used to describe the process by which many neural networks learn and refers to the practise of altering the weights within the system in order to decrease the loss calculated for that neuron. The process involves applying the optimisation function to each neuron in turn and then modifying their respective weights as dictated by said function.

3.2.2. Bias

The term 'bias' refers to a value unique to a neuron to which any data processed by said neuron is added. This is used to emphasise the output of neurons that have shown a previous tendency towards correctness.

3.2.3. Convergence

Convergence is the state at which the neural network's processing can be considered 'complete'. As a neural network is a collection of many individual nodes rather than a single unit, any result that it provides is constructed from an aggregation of the various states present in the network's output layer and thus from the various states present in the network as a whole. 'Convergence' refers to the point at which most or all nodes within the neural network agree on and/or contribute to the desired outcome.

3.2.4. Downsampling

'Downsampling' is a process by which an audio file or stream of higher sample rate is reduced to a lower sample rate.

3.2.5. Error

Error is the human readable metric for calculating how accurate a neural network's output is. It differs from loss (see below) in that it is generally expressed as a percentage or fraction, whereas loss can take many forms depending on the data in question.[8]

3.2.6. Forward pass

The forward pass is process that is used to calculate the outputs from each neuron, and may be performed multiple times during training dependant on the minimisation function and architecture used.

3.2.7. Loss

Loss in a neural network is simply the outcome of the function that is to be minimised [8]. This may take different forms depending on the attributes of the data that is analysed.

3.2.8. Optimisation function

The optimisation function is used to determine how to affect the weights of the neurons in a neural network to decrease the value of the loss function and thus increase the accuracy or precision of the network as whole. This function is also called a minimisation function depending on the form of the value that the loss function returns, and may be the inverse of the forward pass operation depending on the architecture used.

3.2.9. Overfitting

Overfitting is a term used to describe the phenomenon wherein a neural network becomes adept at classifying data within its training set, but lacks the capability to abstract and generalise the patterns it has learned. Overfitted neural networks exhibit great accuracy when analysing data that originated from their training set and markedly less accuracy when analysing data originating from elsewhere. Overfitting generally occurs when the data set the neural network has been trained on is either too small or too uniform.

3.2.10. Premature convergence

Premature convergence occurs when a neural network produces a result that is correct but suboptimal, either in terms of precision or completeness. Premature convergence can be difficult to rectify in areas where the boundaries between outcomes are not well defined such as Large Vocabulary Conversational Speech Recognition(LVCSR)[9] and the evolution of genetic algorithms. [10] [11]

3.2.11. Sample rate

The amount of times per second that an audio sample is recorded.

3.2.12. Vanishing/exploding gradient

The vanishing or exploding gradient problem occurs when a neuron or group of neurons within a deep or recurrent neural network ceases to improve in accuracy during training [6]. This phenomenon can result in a network-wide stasis due to a cascading propagation of the values resulting in the 'vanished' gradient. The term refers to the 'gradient' of the neural network's loss function becoming close to perfectly horizontal, thus preventing a neural network from being able to distinguish correct results from erroneous ones. This problem is especially prevalent in recurrent neural networks due to their recursive nature as feedback loops often result in a vanishing or exploding gradient [12]. It is worth noting that the vanishing and exploding gradient problems are technically distinct issues as the vanishing gradient refers to nodes no longer being able to accept new information, whereas the exploding gradient problem refers to nodes no longer being able to retain old information [13].

3.2.13. Weight

The term 'weight' or 'weighting' refers to a value that is unique to an individual connection between neurons and by which any data that is passed along it is multiplied. This can be considered analogous to a measure of 'confidence' in the result transmitted.

3.3. Neural network architectures

Neural networks have the unique capability to learn, but the rate at which they do so varies dependent on the format and volume of data that they are to process or retain. To optimise a system to efficiently process data irrespective of type or volume has so far proven to be impossible[3], but architectures that are specialised to facilitate the processing of specific data types have been created.

This collection can be divided into two separate groups: feed-forward neural networks (networks where all data flow is in parallel, from input to output) and recurrent neural networks, where data may be directed to flow backwards.

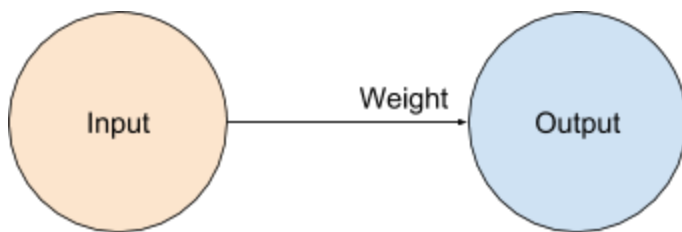
The architectures discussed in the following section are discussed in chronological order of inception.

3.3.1. The perceptron

At its most basic level, a perceptron is a neural network comprised wholly of input and output neurons. [14]

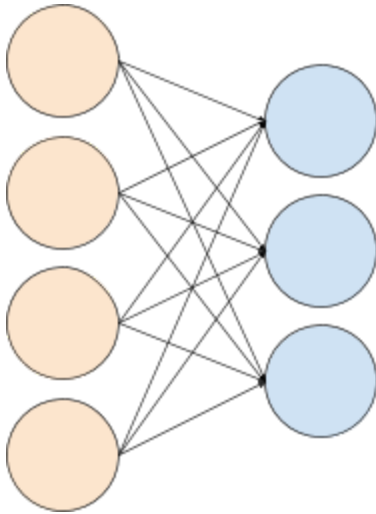
It contains no hidden neurons (neurons used for neither input nor output) and has an extremely limited learning capacity. The mechanism by which this network learns is as follows:

The input neuron is set to equal a value dependent on stimulus, then the output neuron is presented with this value plus a modification peculiar to the connection between the two neurons (called 'weight' below). The output neurons provide an output made up of the summation of all inputs, their respective weightings and a value unique to that neuron (bias).



To train this network, the difference between the actual outcome and the desired outcome must be calculated and the weight and/or bias must be altered to minimise this difference (also called 'loss'). The diagram above shows a simple system and can only give an output of zero or one.

To increase the expressivity of this network, one may simply increase the number of neurons.



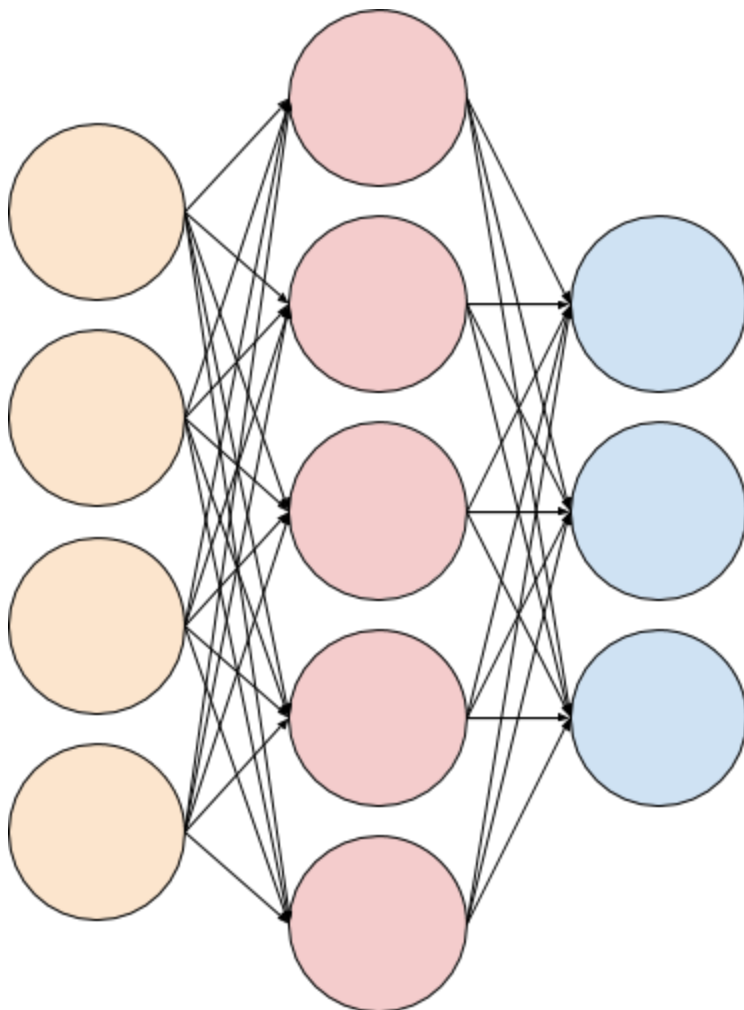
This network now has the ability to present three active neurons, and thus express nine combinations. Variations on this architecture have been used in many fields, including machine vision and statistical analysis [15], [16].

From this architecture, a key point that may be generalised to the field as a whole is evident: each connection between two neurons possesses its own weight, independent of others.

Although this architecture has been successfully employed in many areas, it suffers from a particular flaw in that it assumes that output is directly and closely linked with input. This functions well in areas such as edge detection, but does not translate into areas that require greater levels of abstraction like musical composition. To combat this, one may add intermediate (hidden) layers between the input and output neurons.

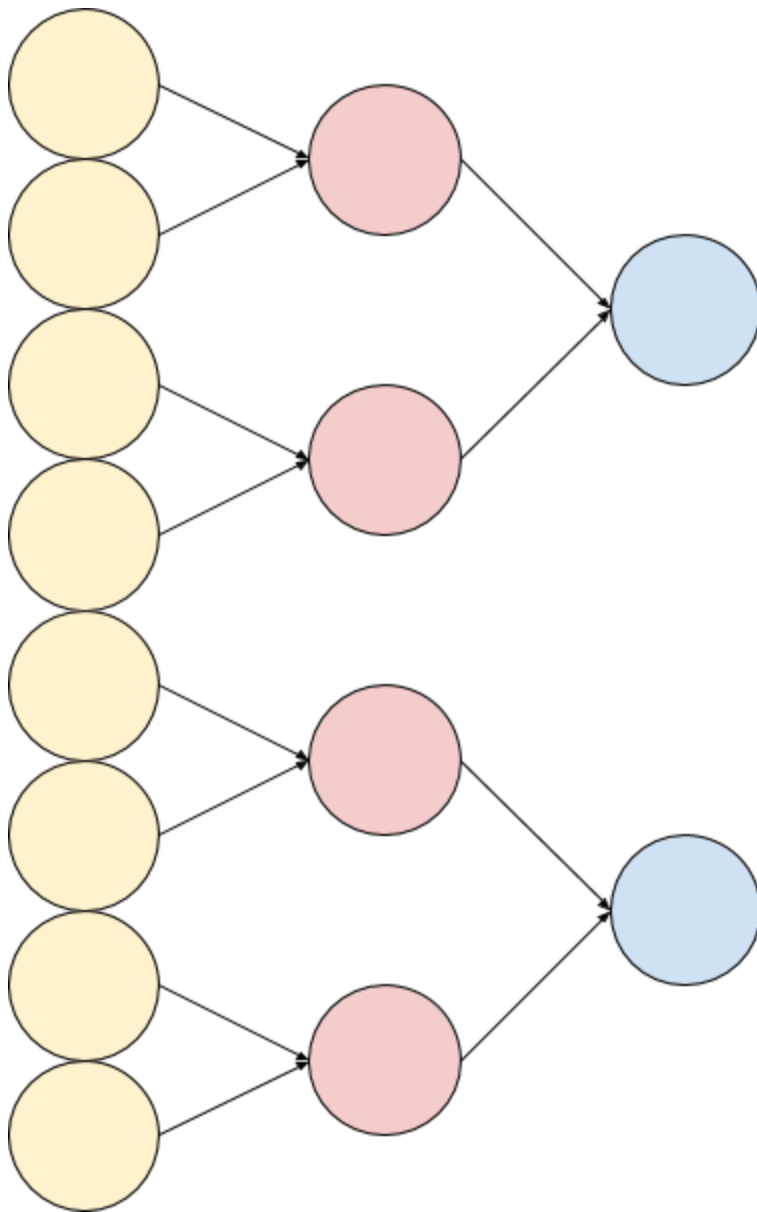
3.3.2. Multilayer perceptron

A multilayer perceptron is so named as it is a modified version of the above architecture. The multiple layers referred to in this case are simply layers of hidden neurons. This network architecture is still very basic, although it can be quite powerful when properly optimised as in the case of target recognition, where multilayer perceptrons have been shown to have the ability to distinguish between four different types of large vehicles (two of which were civilian and two of which were military) with a mean accuracy of 92.3% [17].



3.3.3. Convolutional neural network

A convolutional neural network is one in which features extracted from the input are then abstracted from their detected representation, making this architecture well suited to the task of classification. These networks differ from the architectures shown above as the neurons in one layer are not linked to all the neurons in the next layer but are instead divided into 'groups', where each neuron in one of these groups is then connected to one of the neurons in the next layer. The neuron in which these outputs converge then performs a 'pooling' function to determine which of its respective group has the most desired output.

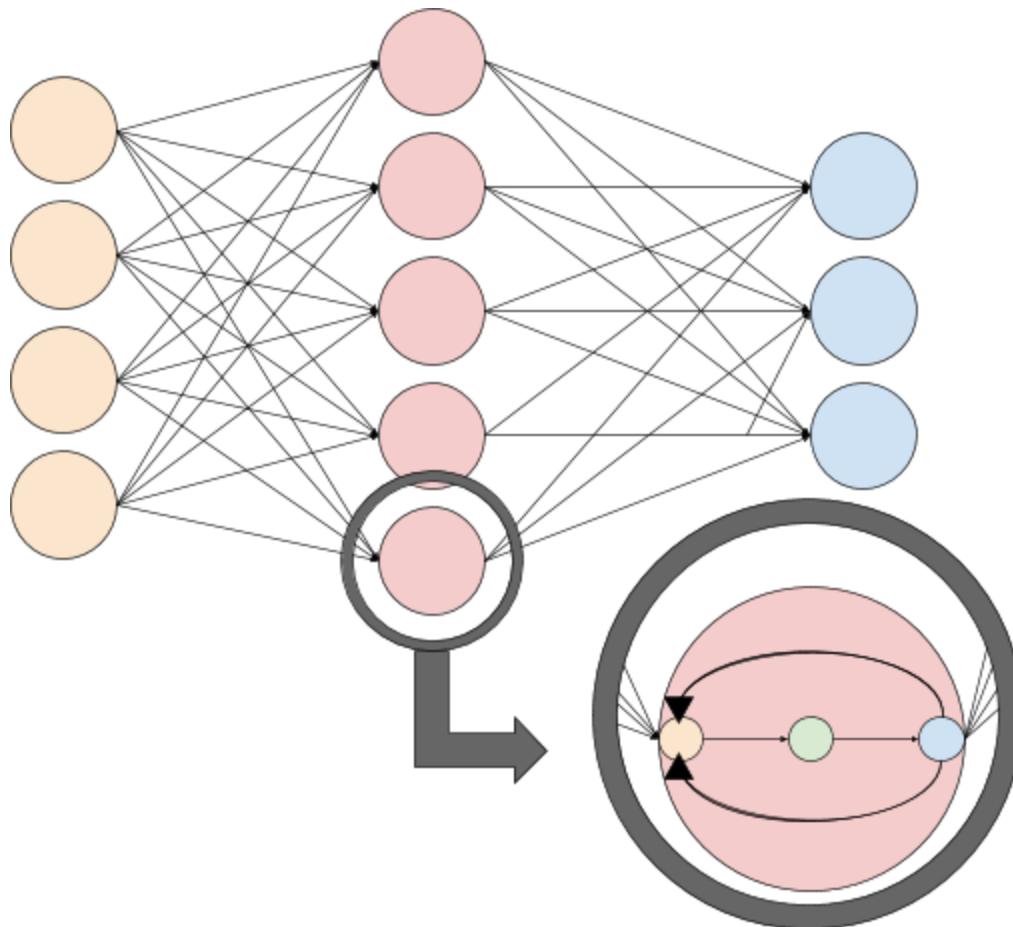


3.3.4. Recurrent neural networks

A recurrent neural network is one where certain neuron connections will feed into previous layers, or into the neuron that produced them. This can allow for more rapid isolation of pertinent features as selected outputs produced by the neural network are used to further reinforce its weighting. Unfortunately recurrent neural networks may also fall prey to the aforementioned 'vanishing gradient' problem if incorrectly optimised.

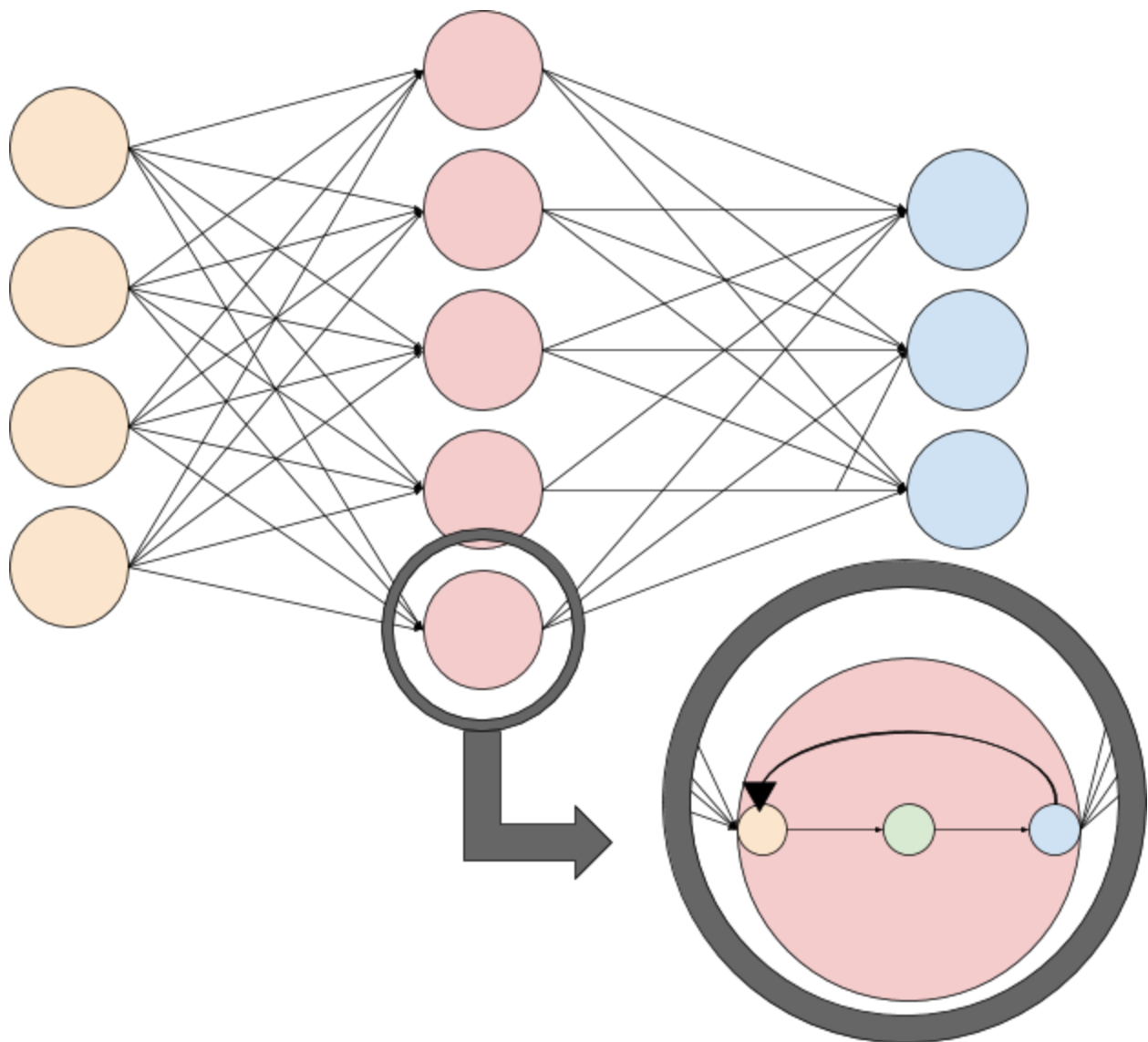
3.3.5. Long Short Term Memory

A Long Short Term Memory or LSTM neural network is one in which the middle layer or layers are composed of cells that retain information through multiple iterations by combining their new input with their previous output. It does this by adding another information stream to the cell in addition to the primary stream and strictly controlling the interactions between the two. This new stream is known as the 'cell state' and its function is to act as a control line for the 'memory' referred to in the above acronym[18]. It is used to determine how much of the cell's previous output should be mixed with the new input, how much data to excise from the input and if the cell's new output should be dampened or multiplied before it leaves the cell. This function is enabled by input, output and forget gates[19][20] within the cell that use the information available in the cell state stream to determine the magnitude of their applied functions.



3.3.6. Gated Recurrent Unit

A Gated Recurrent Unit or GRU is a neural network comprised of cells that are designed in a similar manner to those of an LSTM network in that GRU networks have logic gates controlling the flow of information. It differs in that the functions of the input and forget gate in an LSTM are merged and replaced with a single gate - the update gate [21]. In addition, there is only one data flow through the cell from which the data required to configure the gate functions is extracted.



3.4. Research into real time beat tracking software

Real-time beat tracking software has been prototyped using methods including the following:

3.4.1. Isolating dominant frequencies in base chord changes

In this approach, note onsets are used to provide a base likelihood of where downbeats are likely to be. These estimates are then refined by using low-frequency changes to isolate the positions of highest downbeat probability. This approach has achieved a maximum beat-tracking accuracy of 87.5% [22].

3.4.2. Iteratively weighted summation of onset coincidence

In this approach, regular periods in which multiple note onsets coincide are given a weighting which is increased as they recur. After analysis, these weightings are then compared and those with the highest values are considered to be candidates for downbeat locations. This approach has achieved a maximum beat-tracking accuracy of 73.6% [23].

3.4.3. Analysis of amplitude variance using convolutional neural networks

In this approach, areas wherein the amplitude of detected music changed more than a dynamically-computed parameter were considered downbeats. This approach has achieved a maximum beat-tracking accuracy of 64.2% [24].

4. Design and implementation

4.1. Problem Formulation

The human brain is unable to detect intervals of less than 1/20th of a second [25] with the result that a target accuracy of 99.87% in downbeat recognition would result in any error in a piece of music with a tempo of less than 240 BPM being indistinguishable.

It would therefore be a valuable exercise to determine whether or not it is possible to create a real-time beat tracking system with a classification accuracy rating greater than 99.87%. 240 BPM was chosen as, according to London, 240 BPM is 'too fast to be useful'[26].

4.2. Proposed system design considerations

A common approach in the field of real-time musical analysis is to ensure primarily that the speed with which the system processes and returns data is maximised and then to attempt to increase the accuracy of the output as a secondary concern [23] [24].

In the case of pure analysis this approach is appropriate as inaccuracies can be corrected later but as disruption of a listener's expectations can result in unpredictable emotional responses [27], in a live production environment it can be argued that ensuring correct beat identification is more important than attempting rapid beat identification.

To this end, it was decided that the best approach is to place more emphasis on the correct identification of rhythm than on attempting to first maximise the rapidity of output production.

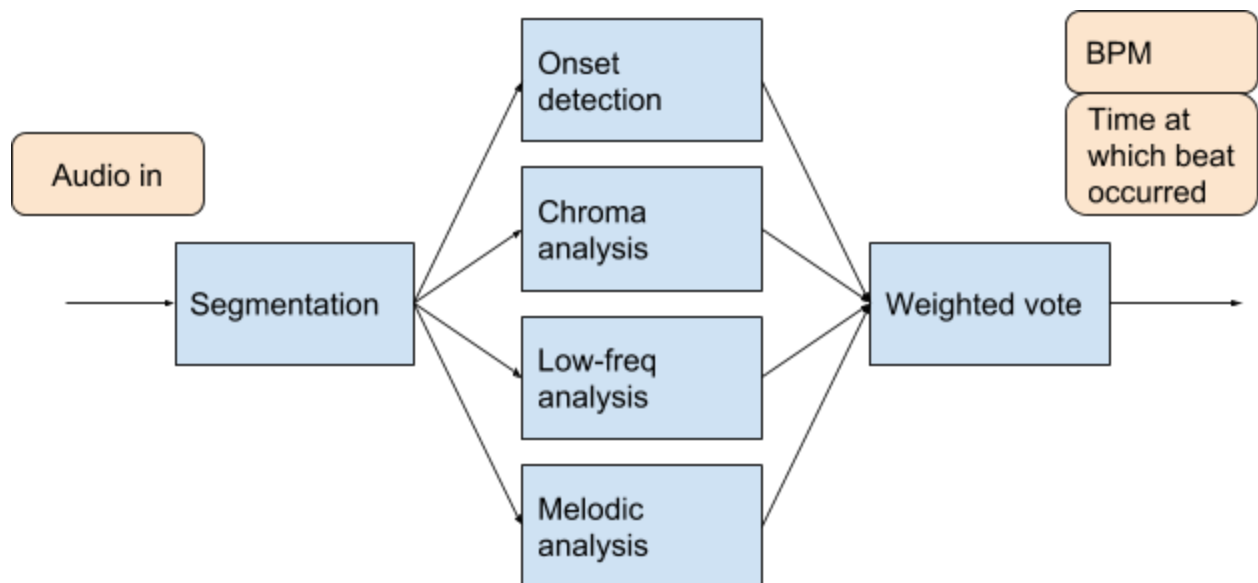
In order to produce a system of neural networks with an emphasis on accuracy, it was necessary to first research neural networks that do not perform in real time because in this field,, time dependency is not a factor and thus accuracy can be prioritised.

The first consideration which should be entertained when designing a neural network is to decide which architecture will be employed. This process can be challenging due to the fact that there are a great number of architectures in common use, many of which are specialised to individual tasks [28]. Multiple architectures were considered including convolutional networks and Long Short Term Memory networks, as well as the possibility of using multiple architectures for the different aspects of the proposed system. It was eventually decided that the Gated Recurrent Unit network architecture would be most suited to use in the proposed system due to its decreased computational and memory usage requirements.[29]

One of the papers most influential to the design of the system used in this project is “Robust Downbeat Tracking Using an Ensemble of Neural Networks” by Simon Durand et al [30]. In this paper, the authors discuss the merits of combining the output of multiple feature extraction systems to combat the shortcomings of any one system in isolation and present a collection of neural networks that can both perform these functions and that are well suited to working in tandem. The system discussed was able to obtain a mean beat classification accuracy of 72.6%, which is a dramatic improvement over the <55.8% accuracy shown by systems containing a solitary neural network. Whilst it is true that the addition of more networks to a system has the potential to dramatically increase its runtime and resource requirements [31], the greater accuracy and robustness presented by such an approach were considered valuable enough to be worth this risk.

4.3. Design

The proposed system is composed of six separate modules - the segmentation module, the chroma analysis module, the note onset detection module, the low-pass filter module, the melodic constant-Q module and the final weighted vote module.



4.3.1. Segmentation

The segmentation module will be the point of first contact between the system and the incoming audio stream and exists to render the audio data stream into a series of values that may be analysed and manipulated by the subsequent aspects of the system. It will perform this function by first rendering the audio stream into a text stream containing the raw voltages that would be used by speakers to play the audio back. This voltage data will then be separated into segments of a length relevant to the data that is to be extracted from it and these segments will then be subjected to either a Fast Fourier Transform or a constant-Q transform in order to obtain their individual frequency profiles.

4.3.2. Chroma analysis

The chroma analysis system will be used to determine the overall chord progression of the audio sample being analysed and from this estimate likely downbeat locations. It will perform this function by analysing a filtered voltage stream containing the frequencies between D_2 (73Hz) and D_3 (584Hz) converted to the form of a constant-Q transform using a window size of 743ms, a hop size of 92.9ms, and a sample rate of 5512.5Hz. This spectrum is then divided into 12 bands representing each note within the frequency range. The analysed frequencies have been chosen due to the likelihood that a representative portion of the notes identifying the harmonic progression will be played in a lower register [32] and the sample rate used in the constant-Q transform has been chosen to reflect this. The large window size and small hop size have been selected in recognition of the integral nature of tonal progression in that the large measure of overlap ensures no relevant data is omitted [33].

4.3.3. Onset detection

This system will be used to determine the onsets of all played notes within the sample in order to check the accuracy of the melodic, low frequency and chroma analysis systems. The system will determine them by dividing the input vertically into multiple 16 non-overlapping frequency bands using a sample rate of 500Hz. These bands will then be individually analysed by obtaining a curve denoting the variance in amplitude within the band and marking areas where the change in amplitude is greater than a specific maximum [34] determined by the current amplitude analysed for that band, in application of Weber's law [35].

4.3.4. Melodic analysis

The melodic analysis system will be used to isolate the melodic patterns detected within the audio stream by using a much higher audio sample rate (11025Hz) than is used in the other networks. In recognition of this, the return from the audio transform is divided into 96 bands, a finer granularity than in the other networks. These numbers have been chosen to encompass the greater extent of melodic range, with all 88 keys on a standard upright piano being expressed and a further half-octave on either side.

4.3.5. Low pass filter

This system will perform a similar function to the melodic analysis system above, but upon the lowest 150Hz of the input divided into 10 bands. This is to account for the fact that low-frequency notes are often used to maintain rhythm in music [36]. This neural network is projected to train and converge with the most rapidity due to its smaller input size.

4.3.6. Weighted vote system

This system will be used to process the outputs given by the previous systems and render them into a binary value denoting the presence or absence of a downbeat in the currently analysed portion of the audio stream. It will also be used to check the various systems for accuracy by cross-examining the comparable aspects of their data.

4.4. Implementation

To test the feasibility of the above system, a smaller prototype system has been created that performs the most computationally-intensive functions of the deterministic modules contained in the proposed system. These are the conversion between the original audio stream and a stream of voltage data that may be analysed, the Fast Fourier Transforms required by the segmentation, melodic, chroma, and low-frequency analysis modules and the creation of test data using the LibRosa library.

The neural networks with the projected highest and lowest computational complexity, melodic and low-frequency analyses respectively, were also created in order to provide an estimate as to the feasibility of their use in a real-time scenario.

4.5. Reflection

4.5.1. Creation

The language that was initially chosen for development of the neural networks and accompanying systems was C++, using the Visual Studio Integrated Development Environment. This decision was made to capitalise on the rapidity of C++ code and prior knowledge of the Visual Studio environment. This approach was discarded partly due to a lack of documentation and functionality regarding C++ in the TensorFlow library [37], and partly due to the need to continually reconfigure the environment in order to fix link errors introduced by importing new components. In addition, the digital signal processing library containing the majority of the tools used in this project was implemented for C and thus had to be referenced as an external object. This greatly increased the difficulty of error checking and resulted in a high level of class-level coupling. This has been solved by switching to using Python, using the IDLE development environment. Raw Python is generally slower than C++, but can be made to run at comparable speeds using the inbuilt methods available.

4.5.2. Input formatting

The configuration of the neural networks within Python was conceptually challenging due to the difficulty in representing the two differently-sized datasets: the Fast Fourier Transform (FFT) spectra and downbeat position, in a manner that allowed them to be mapped one to one. The solution to this problem was to segment the downbeat position list according to the amount of audio that the individual FFT spectra were representing. All frequencies within these FFTs were then given a binary value of 'true' if a downbeat had occurred during this time and a value of 'false' if it had not. This allowed for the two datasets to be presented to the neural network in identical formats.

4.5.3. Temporal metric

The original metric by which segmentation was to be measured was to be that of 'tatoms'[38] - the smallest segment of the audio stream with musical significance. This would then allow for a 'musical coordinate' that was independent of the audio stream's sample rate. This idea was discarded due to the fact that it unnecessarily increased the level of computational complexity whilst providing no benefit to the accuracy or speed of the system as a whole. This metric was replaced with seconds, another metric independent of sample rate.

4.5.4. Sample generation

Obtaining sufficient data for the neural networks to learn from was challenging due to the requirement of sample beat locations. To this end, a deterministic beat-tracking solution (LibRosa) was used [39]. This may limit the peak accuracy of the neural network to the accuracy displayed by LibRosa if no human-verified training data is used. In addition, the quantity of audio samples able to be processed was much reduced due to time constraints, leading to probable overfitting in the results of the neural network. This is not a concern within the scope of this project due to the fact that it was created to investigate the speed with which an accurate neural network performed. The values obtained are taken to be maximum limits to the accuracy of the system rather than useful data due to this fact.

4.5.5. Memory usage

Forming a system that was able to analyse an audio stream at the level of detail required was challenging due to the two gigabyte upper limit on any single neural network running in TensorFlow. Circumventing this issue required the use of a higher initial granularity of segmentation within the input stream. This alteration allowed the total memory usage to remain below two gigabytes.

4.6. Testing and optimisation

4.6.1. Threading Fast Fourier Transforms

The initial trials that were performed on the system attempted to maximise the speed at which the Fast Fourier Transforms were performed. Whilst this not a process that results in a significant delay, it is nevertheless one that affects all data analysis networks employed in the overall program and thus any speed increase would be effectively multiplied fourfold. In pursuance of this, the aspects of the system were implemented so as to be operated in parallel via multithreading. It was determined through experimentation that the use of eight threads provided the optimal speed increase. This is due to the oversaturation of the eight logical cores within the test system with values greater than eight.

Number of threads	Time taken to perform 21000 FFTs
04 threads	210s
06 threads	138s
08 threads	136s
10 threads	139s
12 threads	141s
16 threads	144s

4.6.2. Testing run speed using either CPU or graphics card

It was found through testing that use of the GPU in training the melodic analysis network increased speed by roughly 530%.

Nodes	CPU/GPU	Average time taken per epoch
64	GPU	0.0057s
64	CPU	0.0303s

4.6.3. Optimisation

One of the key aspects of optimising the Python code used in the creation of the prototype system was ensuring the number of loops contained in the raw code was minimised. This is due to the great overhead introduced by Python in repeatedly re-interpreting the code within the loop. The solution to this issue was to use iterative functions present in the libraries utilised, as such functions are often calls to functions written in a quicker language such as C.

4.7. Development timeline

4.7.1. Phase 1:

Research and development of initial audio stream segmentation program

The initial step performed in development of the prototype used in determining the feasibility of accurate real-time beat tracking was that of creating the program that would render an audio file as a stream of voltage data. This was accomplished using the FFmpeg library [40] to render the individual audio frames as a list of integers representing their voltage values, which were then saved in CSV format.

4.7.2. Phase 2:

Creation of Python functions to allow retrieval and parsing of sample data

The function used to retrieve the voltage list data provided by phase one uses the Python CSV library to obtain a list of strings containing the required data. These strings were then rendered as integers, with blank values being replaced with zeroes. Subsequent to this, the list was downsampled to the sample rate required by the initial neural network to be tested at the time using the SciPy[41] library's 'decimate' function.

4.7.3. Phase 3:

Research, development and configuration of Gated Recurrent Unit network

The code used in the creation of the Gated Recurrent Unit was originally developed from an online tutorial [42]. It was then altered to accept input tensors of any type and shape, to use a human-readable accuracy metric and to allow tensors of a size greater than 2GB to be used. Other gradient descent functions, loss functions, and initialisation parameters were tested within the network but were discarded either due to their decreased performance or reliability. This approach was chosen over the use of the inbuilt "tf.contrib.rnn.GRUCell" class as it allowed a greater level of customisation to suit the desired tasks whilst still providing the requisite speed of execution.

4.7.4. Phase 4:

Creation of the list generation and threaded FFT functions

The list generation function was created to render the list of voltages as a series of lists of size N with an overlap of size M , dependent on the desired aspects that were to be analysed.

The first iteration of the function that rendered newly created overlapping voltage lists obtained previously as an analysable list of FFT returns was developed without utilising multithreading, and thus was only useful as a proof of concept. The second iteration of this function then included scalable thread-level parallelism by first dividing the voltage list into the desired number of consecutive subsets. These new lists were then independently processed before being amalgamated to create the complete list of FFT returns.

4.7.5. Phase 5:

Creation of the input sample generation function

The sample generation function uses LibRosa's beat tracking function to obtain a list of downbeat positions within the sample file as training input. It then maps this list to the list of FFTs obtained above so that a list of equal dimensions is created and populated with ones or zeroes depending on whether that FFT is determined to be of a point in the sample file that contains a downbeat.

4.8. Comments on the implementation of the Gated Recurrent Unit class

4.8.1. Loss

In the created system, the loss function used details the difference between the actual and expected outputs [43] and is calculated using this formula:

$$\frac{(\Sigma(O_0) - \Sigma(O_1))^2}{2} / S$$

Where O_0 and O_1 are matrices detailing the actual and desired outputs of the neural network, and S contains the size of the 0th dimension for each.

4.8.2. Optimisation function

In the created system, the minimisation function used is the 'Adam' optimisation function [44], which differs from a typical gradient descent function in that it uses calculated 'momentum' of the gradient representing its mean and its variance at each iteration to inform the changes in weighting. This function was chosen as it both performed well experimentally and has been shown to outperform many competitors [45].

4.8.3. Forward pass

The forward pass is done in raw Python, and follows the following format:

$$\sigma(w_z(h_{t-1}) + u_z(x_t) + b_z) = z_t$$

$$\sigma(w_r(h_{t-1}) + u_r(x_t) + b_r) = r_t$$

$$\tanh((r_t \circ (h_{t-1})) + w_h(x_t) + b_h) = h'_t$$

$$h'_t \circ (1 - z_t) + ((h_{t-1}) \circ z_t) = h_t$$

Where σ is the sigmoid function ($1 / (1 + \exp(-x))$), $\circ$ is the Hadamard product, w_x is a weight, u_x is a hidden weight, and b_x is a bias.

Additionally, z_t is the value of the update gate for that pass, r_t is the value of the reset gate for that pass, and h_t is the newly-created output value.

The weights and biases used in the above formulae are expanded upon below.

4.8.4. Values and initialisation

Each node in the network has nine variables that are unique to it. These variables hold the values that are used in the 'Adam' minimisation function, and are as follows:

Name	Expanded	Description
Wz	Weight-Update	This variable holds some of the weighting which will be applied to the update gate, and is primarily determined by the influence of the newest input (H_t)
Wr	Weight-Reset	This variable holds some of the weighting which will be applied to the reset gate, and is primarily determined by the influence of the newest input (H_t)
Wh	Weight-Output	This variable holds some of the weighting which will be applied to the output gate, and is primarily determined by the influence of the newest input (H_t)

Uz	Hidden-Update	This variable holds some of the weighting which will be applied to the update gate, and is primarily determined by the influence of prior outputs from the current cell (H_{t-1})
Ur	Hidden-Reset	This variable holds some of the weighting which will be applied to the reset gate, and is primarily determined by the influence of prior outputs from the current cell (H_{t-1})
Uh	Hidden-Output	This variable holds some of the weighting which will be applied to the output gate, and is primarily determined by the influence of prior outputs from the current cell (H_{t-1})
Bz	Bias-Update	This variable holds the bias that will be applied to the update gate, and is informed by the newest input and the older inputs equally.
Br	Bias-Reset	This variable holds the bias that will be applied to the reset gate, and is informed by the newest input and the older inputs equally.
Bh	Bias-Output	This variable holds the bias that will be applied to the output gate, and is informed by the newest input and the older inputs equally.

All the above variables are initialised with a random float using a mean variance of 0 and a standard deviation of 0.05, and are updated by the `tf.applyGradients` of the above minimisation function.

5. Analysis

5.1. Speed

The speed of processing exhibited by the neural networks in the prototype system varies depending on the quantity of data that is analysed and the size of network required to obtain the necessary accuracy values. The smallest and largest neural networks were benchmarked and it was found that it was possible to run the less complex networks such as the low frequency analysis system in real time, with a required runtime of 0.001 seconds per second of audio analysed. This delay is smaller than is perceptible by humans [25] and so falls within the range of tolerated delay. This is sadly not true of the larger networks such as the melodic analysis however, as one second of audio took roughly 0.945s to analyse.

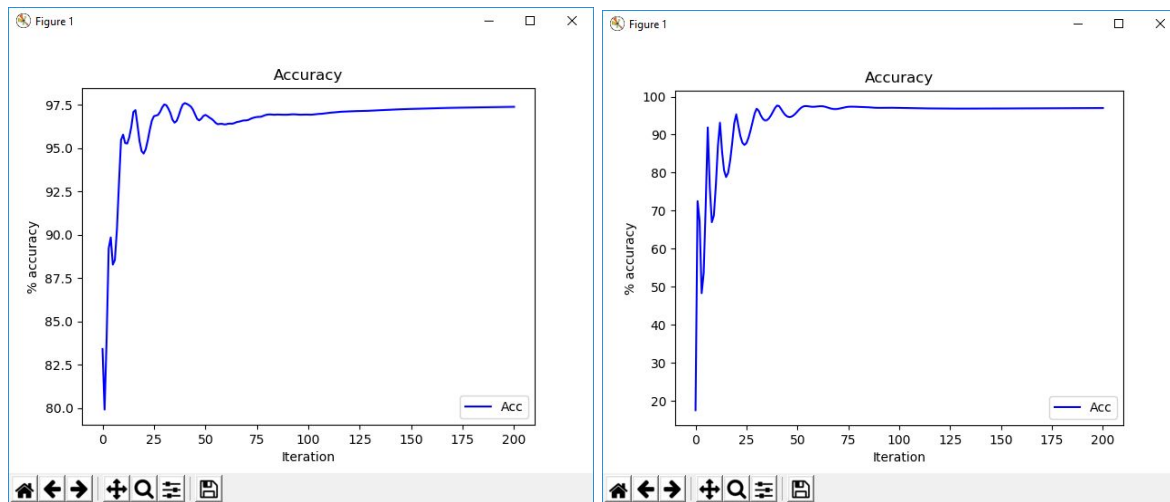
5.2. Accuracy

The melodic analysis network was originally configured to use 16 hidden nodes. It exhibited a maximum accuracy of roughly 70.2%. As the melodic analysis network is the most expressive network and has the highest level of detail in its data input, it is the most likely to have the highest accuracy. In order for the system as a whole to have an accuracy rating of greater than or equal to 99.87% (as discussed in section 4.1), at least one component must also possess this accuracy. As the maximum accuracy achieved was less than this figure, the system as a whole is not functional.

Number of hidden nodes	Percentage accuracy
064	97.2890
072	97.4973
094	97.6435
114	97.7306
140	97.7330

5.3. Examination of the effect of number of hidden nodes on training speed and accuracy

Below are two graphs showing the change in accuracy per iteration for the neural network tested at various numbers of hidden nodes.



The graph on the left shows the accuracy of the network using 42 hidden nodes over 200 iterations. It takes 110 iterations for the neural network's accuracy to stabilise. After this point, it is possible that the accuracy of the neural network would increase with further training but at a markedly slower rate. The maximum accuracy of this attempt was 96.60%.

The graph on the right shows the accuracy of the network using 120 hidden nodes over 200 iterations. It takes 80 iterations for this network's accuracy to stabilise, much faster than with fewer neurons. The maximum accuracy of this attempt was 97.72%.

It is possible to draw the conclusion that the maximum accuracy achievable by the network increases proportionally to the number of hidden neurons it is comprised of.

5.4. Resource requirements

The computer used in development of the prototype system was constructed principally for playing computer games and thus has a powerful graphics card and CPU. These resources are utilised in order to maximise the speed at which the prototype system trains and analyses data. The hardware specification of the computer used in testing is listed below.

Computer spec:

CPU: i3-6100 @ 3.70GHz - 4 core multithreaded

GPU: NVIDIA Geforce GTX 1060 6GB

RAM: 8GB DDR4

It was found that the less complex neural network required a relatively small allocation of resources and could feasibly be run in real time whereas the more complex neural network required nearly all the resources available. The average resource requirements are listed below:

Low frequency analysis: 49% CPU runtime, 1,141MB RAM, 11%GPU runtime, 4.9Gb GRAM allocated

Melodic analysis: 66% CPU runtime, 2,199.2MB RAM, 8~75% GPU runtime, 5.5 GRAM allocated

6. Conclusion

The timing data shows that whilst the simpler neural networks like the low frequency analysis system can be run in real-time on contemporary hardware, the more complex neural networks such as the melodic analysis system cannot. Additionally, a greater pool of resources would be needed to run more than one neural network at a time due to their resource requirements.

Neural networks are well suited to tasks where the boundary between error and correctness is not well defined, for example in handwriting recognition (See section 3.1). The fine level of detail that is required in order to produce results in which errors cannot be detected by the human ear is not achievable using current technology.

7. Evaluation

The measures required to ensure the that level of accuracy is high enough to prevent the system from making inexcusable errors also increase the runtime of analysis such that the system can no longer be run in real time on contemporary hardware. However, the rate at which computing power has increased in recent history shows no sign of abating and there is no reason to assume that this will not continue. It is thus entirely possible that currently impractical systems such as the proposed will in the future be readily achievable.

The architecture used in the proposed system was chosen due to the fact that it currently outperforms many other architectures whilst not requiring an undue amount of hardware resources. Nevertheless, architectures are currently being developed (see 8.1 - Capsule networks) that conceivably could perform with even greater speed and reliability with only a minor increase in memory requirement.

8. Recommendations for further research

8.1. Capsule networks

A recent innovation in the field of machine learning is that of the ‘capsule network’. These networks can be considered a form of nested two-dimensional neural network, with every neuron within the original network being replaced with its own individual neural network. Using this meta-architecture it is theoretically possible to achieve a significant increase in accuracy due to the new network’s ability to recognise objects by their constituent parts (as isolated by the component ‘capsules’ in the network) rather than having to construe the greater object as the sum of a quantity of isolated patterns [46].

8.2. Intel Nervana chip

Intel are in the process of producing a CPU that is being designed solely to increase the performance with which a neural network may be trained and run. Little in the way of detail has been revealed, but the company states that they are aiming to “..deliver up to 100x reduction in the time to train a deep learning model over the next three years compared to GPU solutions.”[47]

9. Code listings

Below is a complete listing of all code used in the prototype system, with annotations describing its function. Code is tabbed to the right, annotations are not. Comments have been stripped from the code as listed in order to reduce its size, but are present in the digital submission.

9.1. Annotated Python code listing

9.1.1. Libraries

The libraries below are used to:

Allow easy reformatting of data in tensor or array format,
facilitate training using the AdamOptimiser method,
facilitate the display of desired data using the matplotlib library,
generate the list of beat timings used in training,
and import data from CSV files.

```
import tensorflow as tf
import numpy as np
from numpy import random
import matplotlib.pyplot as plt
import scipy as sp
from IPython import display
from scipy import signal
import csv
import librosa
import librosa.display
#%% matplotlib inline
import time
import _thread
```

9.1.2. Reverse array of bits

Function used to reformat an integer as a reverse-ordered array of bits

This is required for the GRU network to process data.

```
def as_bytes(num, final_size):
    res = []
    for _ in range(final_size):
        res.append(num % 2)
        num //= 2
    return res
```

9.1.3. CSV reader

This function is used to read data from a CSV file and render it as a list of strings. The “if len(rows) == 1:” statement is due to the fact that the data provided by the C++ portion of this code is formatted so that all data is present on the same line.

```
def readCSV(inputS):
    rowcount = 0
    rows = list()
    with open(inputS, newline='') as csvfile:
        dialect = csv.Sniffer().sniff(csvfile.read(1024))
        csvfile.seek(0)
        reader = csv.reader(csvfile, dialect)
        for row in reader:
            rowcount = rowcount + 1
            rows.append(row)
        if len(rows) == 1 :
            rows = row
    return rows
```

9.1.4. Data return, type checking and downsampling

This function calls the above readCSV function and returns the data as a list of integers. It also downsamples the data for processing by the neural network to be tested at the time (either by 4 or 88 for the melodic and low-frequency networks respectively).

```
def returnData(sampleSize,fName):
    item = list()
    items = list(list())
    filteredItems = list()
    rows = readCSV(fName)
    val = int
    for numLoop in range(0,len(rows)):
        if rows[numLoop].isdigit() == False:
            rows[numLoop] = 0
        else:
            rows[numLoop] = int(rows[numLoop])
    if sampleSize == 64:
        filteredItems = sp.signal.decimate(rows,11)
        filteredItems = sp.signal.decimate(filteredItems,8)
    else:
        filteredItems = sp.signal.decimate(rows, 4)
    rows = filteredItems.tolist()
    while len(rows)%sampleSize != 0:
        rows.append(0)
    filteredItems = rows
    return filteredItems
```

9.1.5. Nested list creation function

This function is used to format the above list of integers as a list of lists of integers. Specifically, it returns lists of $(total/sampleSize)$ integers of size `sampleSize`, with an overlap of $(sampleSize-hopSize)$.

```
def genLists(filteredItems, sampleSize, hopSize):
    data = []
    zeroArr = [0.0]*sampleSize
    for loop in range(0,int(np.ceil(len(filteredItems)/hopSize))):
        data.append(list())
        try:
            data[loop] = filteredItems[(loop*hopSize):(loop*hopSize)+sampleSize]
        except:
            data[loop] = filteredItems[(loop*hopSize):(loop*hopSize)+sampleSize]
            if (loop*hopSize)-len(filteredItems) < sampleSize:
                data[loop] = filteredItems[(loop*hopSize):len(filteredItems)-1]
    return data
```

9.1.6. FFT function with absolute value return

This function is used to return the FFT representation of the list it is presented. It returns the absolute value (real+complex) to prevent difficulties analysing the complex values within the neural network, but this does mean that it is no longer possible to perform an FFT⁻¹ operation on this return. The “while len(itemsList) < sampleSize:” statement ensures that all FFT returns processed by this function are large enough for processing by padding them with zeroes.

```
def genFFT(itemsList, sampleSize):
    while len(itemsList) < sampleSize:
        itemsList.append(0.0)
    with tf.device('/gpu:0'):
        real = tf.cast(tf.constant(itemsList),dtype=tf.float32)
        imag = tf.constant([0.0]*len(itemsList))
        comp = tf.complex(real,imag)
        ret = tf.fft(comp)
    sess = tf.Session()
    res = sess.run(ret)
    outputs = tf.abs(res).eval(session=sess)
    tf.reset_default_graph()
    return outputs
```

9.1.7. Threaded iterating FFT function

This function iterates along the provided list and calls the above FFT function. It also has a `print()` function in it to show it hasn't crashed, as it can take a very long time with a large dataset. This function is called within a thread, and is thread-safe barring the print function. This means that the output to the console will occasionally be garbled, but this is not a concern as it is mainly there to show that it's still running.

```
def genFFTs(itemsLists, sampleSize, FFTrets):
    itemLength = str(len(itemsLists))
    for loop in range(0,len(itemsLists)):
        FFTStart = time.time()
        FFTrets.append(genFFT(itemsLists[loop],sampleSize))
        FFTFin = time.time()
        if loop%500 == 0: print("loop: "+
str(loop)+"-"+(str(FFTFin-FFTStart))+"-"+itemLength)
```

9.1.8. Conversion of integer array to array of reversed bits

This function returns a reversed array of bits that represents the arrays of integers it has been presented.

```
def genBatch(length,FFTRets,beatBinList,sampleSize):
    lenTest = len(FFTRets)
    x = np.empty((lenTest, 32, sampleSize))
    y = np.empty((lenTest, 32, sampleSize))
    for i in range(0, lenTest-1):
        for l in range(0,sampleSize-1):
            test = as_bytes(FFTRets[i][l],32)
            x[i, :, l] = test
            y[i, :, l] = as_bytes(beatBinList[i*l],32)
    return x, y
```

9.1.9. Sample creation function

This is the function that calls all the above functions and is used to generate two lists of equal size, one containing absolute FFT returns and one containing downbeat information.

This function creates eight threads to perform the FFTs to increase its speed. Threading is imported here so as to be obviously associated with this function alone.

```
from threading import Thread
def songBeats(frames,file,sampleSize):
    voltList = returnData(sampleSize,frames)
    inLists = genLists(voltList,sampleSize,hopSize)
    if 1==0: inLists = inLists[0:int(len(inLists)/32)]
    ThreadNum = 8
    listLenFract = int(len(inLists)/ThreadNum)
    inListsList = list()
    FFTRetsList = list()
    threadsList = list()
    for loop in range(0,ThreadNum):
        inListsList.append(inLists[listLenFract*loop:listLenFract*(loop+1)])
        FFTRetsList.append(list())
        threadsList.append(Thread(target=genFFTs,
            args=(inListsList[loop],sampleSize,FFTRetsList[loop])))
        threadsList[loop].start()
    for loop in range(0,ThreadNum):
        threadsList[loop].join()
    FFTRets = FFTRetsList[0]
    for loop in range(0,ThreadNum):
        FFTRets.extend(FFTRetsList[loop])
    y, sr = librosa.load(file)
    tempo, beats = librosa.beat.beat_track(y=y, sr=sr)
    beatTimesList = librosa.frames_to_time(beats, sr=sr)
    beatBinList = [0]*len(FFTRets)*sampleSize
    for outer in range (0, len(FFTRets)):
        for inner in range (0,len(FFTRets[outer])):
            if outer != len(FFTRets):
                if outer*0.128 < beatTimesList[inner]:
                    if (outer+1)*0.128 >= beatTimesList[inner]:
                        beatBinList[outer*inner] += 1
                        break
    return(FFTRets, beatBinList)
```

9.1.10. Configuration: Generate FFT, load FFT values, or no action.

Here, the `sampleSize` and `hopSize` variables are set as required. The lists of FFT returns and beat locations are then either generated or loaded if required by setting “if 1==0:” to “if 1==1:”. These steps are only required if the training sets have not already been generated and saved.

```
sampleSize = 96
hopSize = 12
if 1==0:
    FFTRetsEagles, beatBinListEagles =
songBeats('C:\\Media\\FramesEagles.txt','C:\\Media\\Heynow.mp3',sampleSize)
    FFTRetsRussia, beatBinListRussia =
songBeats('C:\\Media\\FramesRussia.txt','C:\\Media\\Russia.mp3',sampleSize)
    np.save("C:\\Media\\FFTRetsEagles",FFTRetsEagles)
    np.save("C:\\Media\\BeatBinListEagles",beatBinListEagles)
    np.save("C:\\Media\\FFTRetsRussia",FFTRetsRussia)
    np.save("C:\\Media\\BeatBinListRussia",beatBinListRussia)
else:
    if 1==0:
        print("loading data")
        FFTRetsEagles = np.load("C:\\Media\\FFTRetsEagles.npy")
        beatBinListEagles = np.load("C:\\Media\\BeatBinListEagles.npy")
        FFTRetsRussia = np.load("C:\\Media\\FFTRetsRussia.npy")
        beatBinListRussia = np.load("C:\\Media\\BeatBinListRussia.npy")
        print("data loaded")
```

9.1.11. Configuration: Equalise list size or no action

Here, the size of the lists generated above is equalised. This step is required only if the training sets have not been generated and saved.

```
if 1==0:
    if len(FFTRetsEagles) < len(FFTRetsRussia):
        FFTRetsTrain = FFTRetsEagles
        FFTRetsTest = FFTRetsRussia[:len(FFTRetsEagles)]
        beatBinListTrain = beatBinListEagles
        beatBinListTest = beatBinListRussia[:len(beatBinListEagles)]
    else:
        FFTRetsTrain = FFTRetsEagles[:len(FFTRetsRussia)]
        FFTRetsTest = FFTRetsRussia
        beatBinListTrain = beatBinListEagles[:len(beatBinListRussia)]
        beatBinListTest = beatBinListRussia
        batchSize = len(FFTRetsTrain)
        timeSize = 32
```

9.1.12. Configuration: Generate and save training set, or no action.

Here, the training set is generated and saved. Additionally, the time taken to generate the training sets is printed.

```
if 1==0:
    trainstart = time.time()
    print("Generating training set")
    X_train, Y_train = genBatch(batchSize, FFTRetsTrain, beatBinListTrain, sampleSize)
    trainstop = time.time()
    print("Generated training set: "+str(trainstop-trainstart))
    np.save("C:\\Media\\X_train.npy", X_train)
    np.save("C:\\Media\\Y_train.npy", Y_train)
```

9.1.13. Configuration: Load training set, or no action

The training sets are loaded here. This step makes the previous steps irrelevant if it is completed, but the “if 1==1:” is kept so it can be disabled easily if different training data is to be generated.

```
if 1==1:
    print("loading data")
    loadstart = time.time()
    X_train = np.load("C:\\Media\\X_train.npy")
    Y_train = np.load("C:\\Media\\Y_train.npy")
    loadstop = time.time()
    print("data loaded - "+str(loadstop-loadstart))
```

9.1.14. Configuration: Generate and save test set, or no action

This calls the genBatch function to create the test set. It is currently disabled.

```
if 1==0:
    print("Generating test set")
    X_test, Y_test = genBatch(len(FFTRetsTest), FFTRetsTest,
beatBinListTest,sampleSize)
    teststop = time.time()
    print("Generated test set: ")
    np.save("C:\\Media\\X_test.npy",X_test)
    np.save("C:\\Media\\Y_test.npy",Y_test)
```

9.1.15. Configuration: Load test set, or no action

This loads the test set. It is currently disabled as the training set is being used in its place.

```
if 1==0:
    print("loading test set")
    X_test = np.load("C:\\Media\\X_test.npy")
    Y_test = np.load("C:\\Media\\Y_test.npy")
    print("Loaded test set")
    batchSize = len(X_train)
    timeSize = 32
```

9.1.16. Configuration: Reduce training set size and equalise both inputs, or no action

This is to prevent the total memory usage being more than 2GB in size. It is also setting the test set to the training set as in the last set of trials the test set was corrupted by a crash. The test set was not regenerated due to time constraints.

```
if 1==1:
    batchSize = int(batchSize * 0.05)
    offset = 8
    X_train = X_train[batchSize*offset:batchSize*(offset+1)]
    Y_train = Y_train[batchSize*offset:batchSize*(offset+1)]
    X_test = X_train
    Y_test = Y_train
```

9.1.17. GRU class definition: Initialisation function

Below is the initialisation function for the GRU class. Within, the weights and biases are initialised using a mean of 0 and a standard deviation of 0.01. The shape of the input layer tensor is then altered in order to allow the forward pass operation easy access to the data it requires

```
class GRU:
    def __init__(self, inputDim, hiddenSize, dtype=tf.float64):
        self.inputDim = inputDim
        self.hiddenSize = hiddenSize
        self.stdDev = 0.01
        self.mean = 0
        self.Wr = tf.Variable(tf.truncated_normal(dtype=dtype, shape=(self.inputDim,
self.hiddenSize), mean= self.mean, stddev=self.stdDev), name='Wr')
        self.Wz = tf.Variable(tf.truncated_normal(dtype=dtype, shape=(self.inputDim,
self.hiddenSize), mean= self.mean, stddev=self.stdDev), name='Wz')
        self.Wh = tf.Variable(tf.truncated_normal(dtype=dtype, shape=(self.inputDim,
self.hiddenSize), mean= self.mean, stddev=self.stdDev), name='Wh')
        self.Ur = tf.Variable(tf.truncated_normal(dtype=dtype, shape=(self.hiddenSize,
self.hiddenSize), mean= self.mean, stddev=self.stdDev), name='Ur')
        self.Uz = tf.Variable(tf.truncated_normal(dtype=dtype, shape=(self.hiddenSize,
self.hiddenSize), mean= self.mean, stddev=self.stdDev), name='Uz')
        self.Uh = tf.Variable(tf.truncated_normal(dtype=dtype, shape=(self.hiddenSize,
self.hiddenSize), mean= self.mean, stddev=self.stdDev), name='Uh')
        self.br = tf.Variable(tf.truncated_normal(dtype=dtype, shape=(self.hiddenSize,),
mean= self.mean, stddev=self.stdDev), name='br')
        self.bz = tf.Variable(tf.truncated_normal(dtype=dtype, shape=(self.hiddenSize,),
mean= self.mean, stddev=self.stdDev), name='bz')
        self.bh = tf.Variable(tf.truncated_normal(dtype=dtype, shape=(self.hiddenSize,),
mean= self.mean, stddev=self.stdDev), name='bh')
        self.input_layer = tf.placeholder(dtype=tf.float64, shape=(None, None, inputDim),
name='input')
        self.x_t = tf.transpose(self.input_layer, [1, 0, 2], name='x_t')
        self.h_0 = tf.matmul(self.x_t[0, :, :], tf.zeros(dtype=tf.float64, shape=(inputDim,
hiddenSize)), name='h_0')
        self.h_t_transposed = tf.scan(self.forward_pass, self.x_t, initializer=self.h_0,
name='h_t_transposed')
        self.h_t = tf.transpose(self.h_t_transposed, [1, 0, 2], name='h_t')
```

9.1.18. GRU class definition: Forward pass function

This function implements the forward pass as discussed in section 4.8.3.

```
def forward_pass(self, h_tm1, x_t):
    z_t = tf.sigmoid(tf.matmul(x_t, self.Wz) + tf.matmul(h_tm1, self.Uz) + self.bz)
    r_t = tf.sigmoid(tf.matmul(x_t, self.Wr) + tf.matmul(h_tm1, self.Ur) + self.br)
    h_proposal = tf.tanh(tf.matmul(x_t, self.Wh) + tf.matmul(tf.multiply(r_t, h_tm1),
self.Uh) + self.bh)
    h_t = tf.multiply(1 - z_t, h_tm1) + tf.multiply(z_t, h_proposal)
    return h_t
```

9.1.19. Configuration: Initialise neural network, or load configuration from file

Either sets up the neural network and initialises the variables used in training and running it or loads these variables from a file.

```
if 1==1:
    with tf.device('/gpu:0'):

        NNSetupStart = time.time()

        inputDim = sampleSize
        hiddenSize = 94
        sess = tf.Session()
        gru = GRU(inputDim, hiddenSize)
        W_output = tf.Variable(tf.truncated_normal(dtype=tf.float64, shape=(hiddenSize, 1),
mean=0, stddev=0.01))
        b_output = tf.Variable(tf.truncated_normal(dtype=tf.float64, shape=(1,), mean=0,
stddev=0.01))
        output = tf.map_fn(lambda h_t: tf.matmul(h_t, W_output) + b_output, gru.h_t)
        expectedOutput = tf.placeholder(dtype=tf.float64, shape=(batchSize, timeSize,
sampleSize), name='expectedOutput')
        test_output = tf.placeholder(dtype=tf.float64, shape=(1, timeSize, sampleSize),
name='expectedOutput')
        loss = tf.reduce_sum(0.5 * tf.pow(output - expectedOutput, 2)) / float(batchSize)
        DataPlaceholder = tf.placeholder(dtype=tf.float32,
shape=(batchSize,32,sampleSize), name='data')
    else:
        with tf.device('gpu:0'):
            sess = tf.Session()
            saver.restore(sess, "C:\\\\Media\\\\Session1.ckpt")
```

9.1.20. Configuration: Initialise variables larger than 2GB, or no action

This code is only necessary if one of the input tensors are larger than 2GB. As the total size of the neural network is often not allowed to be larger than 2GB, this is unusual. This is done using a set of functions that run within the session environment that the neural network normally runs.

```
if 1==0:
    DataPlaceholder = tf.placeholder(dtype=tf.float32,
    shape=(batchSize,32,sampleSize), name='data')
    X_trainVar = tf.get_variable('X_train',[batchSize,32,sampleSize],tf.float32)
    Y_trainVar = tf.get_variable('Y_train',[batchSize,32,sampleSize],tf.float32)
    X_testVar = tf.get_variable('X_test',[batchSize,32,sampleSize],tf.float32)
    Y_testVar = tf.get_variable('Y_test',[batchSize,32,sampleSize],tf.float32)
    set_x1 = X_trainVar.assign(DataPlaceholder)
    set_y1 = X_trainVar.assign(DataPlaceholder)
    set_x2 = X_trainVar.assign(DataPlaceholder)
    set_y2 = X_trainVar.assign(DataPlaceholder)
    sess.run(set_x1, feed_dict={DataPlaceholder:X_train})
    sess.run(set_y1, feed_dict={DataPlaceholder:Y_train})
    sess.run(set_x2, feed_dict={DataPlaceholder:X_test})
    sess.run(set_y2, feed_dict={DataPlaceholder:Y_test})
    X_trainVar = X_train
    Y_trainVar = Y_train
    X_testVar = X_test
    Y_testVar = Y_test
```

9.1.21. GRU runtime: Initialise variables, session and minimisation function

Here, the session variables are initialised. It must be noted that the AdamOptimizer function attempts to optimise all variables declared that do not have (trainable=False) in their declaration.

```
tf_metric, tf_metric_update = tf.metrics.accuracy(Y_train, Y_train, name="MyMetric")
running_vars = tf.get_collection(tf.GraphKeys.LOCAL_VARIABLES, scope="MyMetric")
running_vars_initializer = tf.variables_initializer(var_list=running_vars)
train_step = tf.train.AdamOptimizer().minimize(loss)
init_variables = tf.global_variables_initializer()
sess.run(init_variables)
sess.run(running_vars_initializer)
train_losses = []
validation_losses = []
acc4 = []
NNSetupStop = time.time()
print("Setting up NN: "+str(NNSetupStop - NNSetupStart))
lastAcc = 0.0
```

9.1.22. GRU runtime: Training and running

In this loop the neural network is trained, then run in order to obtain its accuracy measure. An update on the accuracy and loss is given every 200 iterations.

```
for epoch in range(5000):
    _, train_loss = sess.run([train_step, loss], feed_dict={gru.input_layer: X_train,
expectedOutput: Y_train})

    runtime = time.time()
    Y_predicted = sess.run(output, feed_dict={gru.input_layer: X_train})
    stoptime = time.time()
    print("Time taken: " + str(((stoptime-runtime)/batchSize)*sampleSize))
    sess.run(tf_metric_update)
    score = sess.run(tf_metric)
    validation_loss = sess.run(loss, feed_dict={gru.input_layer: X_train, expectedOutput:
Y_train})
    train_losses += [train_loss]
    validation_losses += [validation_loss]

    NNSetupStop = time.time()
    acc2 = np.sum(np.abs(np.subtract(Y_predicted, Y_test)))
    acc3 = acc2/(sampleSize*batchSize)
    acc4 += [100-(acc3*100)]
    if 100-(acc3*100) > lastAcc: lastAcc = (100-(acc3*100))
    if epoch > 0:
        if epoch %200 == 0:
            endTime = time.time()
            with tf.device('/cpu:0'):
                saver = tf.train.Saver()
                saver.save(sess, "C:\\Media\\Session1.ckpt")
            plt.plot(acc4, '-b', label='Acc')
            plt.legend(loc=0)
            plt.title('Accuracy')
            plt.xlabel('Iteration')
            plt.ylabel('% accuracy')
            plt.show()
            print('Iteration: %d, train loss: %.4f, test loss: %.4f, acc: %.4f' % (epoch, train_loss,
validation_loss, lastAcc))
            print("ran NN epochs: "+str(endTime - NNSetupStop)+"Single
item-"+str(stoptime-runtime))
```

9.2. Annotated C++ code listing

The code in this section is not tabbed, as tabs are already present within the code. Code and its annotation will instead be separated by '-----'

9.2.1. Setup: Standard IO and maths libraries

These libraries are used to allow:

The use of string manipulations like strcpy,
writing to and reading from files
easy mathematical processing of array-like structures.

```
-----  
#include <stdint.h>  
#include <windows.h>  
#include <stdio.h>  
#include <iostream>  
#include <fstream>  
#include <vector>  
#include <stdlib.h>  
#include <array>  
#include <string>
```

9.2.2. Setup: Audio processing libraries

These libraries are used to allow the processing of audio files and streams. They were implemented for C, and so must be referenced as external objects in order to function.

```
-----  
extern "C"  
{  
    // Audio processing libraries  
    #include <libavcodec/avcodec.h>  
    #include <libavformat/avformat.h>  
    #include <libavdevice/avdevice.h>  
    #include <libavutil/avutil.h>  
    #include <libavutil/opt.h>  
    #include <libavutil/timestamp.h>
```

```
#include <libavcodec/avfft.h>
#include "Source.h"
};
```

9.2.3. C++ runtime: Error

This returns the char* it takes to the console.

```
void fatal(const char* mes)
{
    std::cout << "Error: " << mes << std::endl;
    exit(1);
}
```

9.2.4. C++ runtime: Main

This function opens a specified file and renders it as a stream. It then analyses the stream, selects the correct codec if available, and uses said codec to decode the stream into a series of frames. It then stores these frames in a list and processes this list so as to render the original information format (two bytes) readable by the integer processing function in Python (float). Finally, it trims the audio right channel so only the left audio channel remains before saving it. This is done to reduce resource requirements later on.

```
int main(int argc, char **argv)
{

    // Initialize FFmpeg
    av_register_all();
    avcodec_register_all();
    avdevice_register_all();

    char* fName = "C:/Media/Russia.mp3";
    //char* fName = "C:/Media/Heynow.mp3";

    AVFormatContext* formatContext = NULL;
    if (avformat_open_input(&formatContext, fName, NULL, NULL) != 0)
    {
        fatal("error opening file");
    }
}
```

```

AVCodecContext* codecContext = NULL;
AVCodec *codec;

if (avformat_find_stream_info(formatContext, NULL) < 0)
{
    fatal("Cannot find stream info");
}

int audio_stream_index;

if ((audio_stream_index = av_find_best_stream(formatContext,
AVMEDIA_TYPE_AUDIO, -1, -1, &codec, 0)) < 0)
{
    fatal("Cannot find stream info");
}

codecContext = formatContext->streams[audio_stream_index]->codec;
av_opt_set_int(codecContext, "refcounted_frames", 1, 0);

if (avcodec_open2(codecContext, codec, NULL) < 0)
{
    fatal("cannot open audio decoder");
}

if (!codecContext->channel_layout)
{
    codecContext->channel_layout =
av_get_default_channel_layout(codecContext->channels);
}

AVPacket packet;
av_init_packet(&packet);

AVFrame *frame = av_frame_alloc();
if (!frame)
{
    fatal("Cannot allocate frame");
}

int got_frame;
packet.data = NULL;
int frame_num = 0;

```

```

std::vector<std::array<double, 2>>frameList;
std::array<double, 2> vectFrame{ 1, 2 };
vectFrame[0] = 0;
vectFrame[1] = 0;

const int in_channels = 2, in_samples = 512, sample_rate = 44100;
const int max_buffer_size = av_samples_get_buffer_size(NULL, in_channels,
in_samples, AV_SAMPLE_FMT_FLT, 1);

while (av_read_frame(formatContext, &packet) == 0)
{
    //std::cout << "Packet size = " << packet.size << std::endl;

    if (packet.stream_index == audio_stream_index)
    {
        int bytes;
        if ((bytes = avcodec_decode_audio4
(codecContext, frame, &got_frame, &packet)) < 0)
        {
            fatal("Error decoding audio");
        }

        if (got_frame)
        {
            //std::cout << "decoded bytes = " << bytes << std::endl;
        }

        int bytes_per_sample =
av_get_bytes_per_sample((AVSampleFormat)frame->format);
        int samples = frame->nb_samples;
        int unpadded_size = samples * bytes_per_sample;

        //std::cout << "frame " << frame_num++ << " num samples = " <<
frame->nb_samples

        //<< " bytes per sample = " << bytes_per_sample
        //<< " pts:" << ts2timestr(frame->pts, &codecContext->time_base)
<< std::endl;

        //std::cout << "Values (channel1 channel2):- " << std::endl;

        for (int s = 0; s < samples; s++)

```

```

        {
            double sample1 = 0L;
            double sample2 = 0L;
            for (int b = (bytes_per_sample - 1); b >= 0; b--)
            {
                int index = (s * bytes_per_sample) + b;
                sample1 = ((int)sample1 << 8) +
frame->extended_data[0][index];
                sample2 = ((int)sample2 << 8) +
frame->extended_data[1][index];

            }
            //printf("%d\t%d\n", sample1, sample2);
            vectFrame[0] = (double)sample1;
            vectFrame[1] = (double)sample2;
            frameList.push_back(vectFrame);
        }
    }

    std::ofstream test;
    test.open("C:/Media/frames1.txt");
    for (std::vector<std::array<double,2>>::iterator iloop = frameList.begin(); iloop !=
frameList.end(); iloop++)
    {
        test << std::to_string(iloop[0][0]) + ",";
    }
    test.close();

    printf("Ended\n");
    scanf("%c");
}

```

Bibliography

- [1] M. Isard and A. Blake, "Contour tracking by stochastic propagation of conditional density," in *Computer Vision — ECCV '96*, 1996, pp. 343–356.
- [2] S. G. Djorgovski *et al.*, "Some Pattern Recognition Challenges in Data-Intensive Astronomy," *arXiv [astro-ph]*, 29-Aug-2006.
- [3] M. T. Hagan, H. B. Demuth, and M. H. Beale, "Neural network design," 1996.
- [4] J. A. Hertz, *Introduction to the theory of neural computation*. CRC Press, 2018.
- [5] A. Ghaffari, H. Abdollahi, M. R. Khoshayand, I. S. Bozchalooi, A. Dadgar, and M. Rafiee-Tehrani, "Performance comparison of neural network training algorithms in modeling of bimodal drug delivery," *Int. J. Pharm.*, vol. 327, no. 1, pp. 126–138, Dec. 2006.
- [6] S. Hochreiter, "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions," *Int. J. Uncertainty Fuzziness Knowledge Based Syst.*, vol. 06, no. 02, pp. 107–116, Apr. 1998.
- [7] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.
- [8] K. Janocha and W. M. Czarnecki, "On Loss Functions for Deep Neural Networks in Classification," *arXiv [cs.LG]*, 18-Feb-2017.
- [9] K. Chen and Q. Huo, "Training Deep Bidirectional LSTM Acoustic Model for LVCSR by a Context-sensitive-chunk BPTT Approach," *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, vol. 24, no. 7, pp. 1185–1193, Jul. 2016.
- [10] M. Bardeen, "Survey of methods to prevent premature convergence in evolutionary algorithms," in *Workshop of Natural Computing, Jornadas Chilenas de Computacion*, 2013, pp. 13–15.
- [11] B. Xin, Y. Wang, W. Gao, D. Wipf, and B. Wang, "Maximal Sparsity with Deep Networks?," in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., 2016, pp. 4340–4348.

- [12] R. Pascanu, T. Mikolov, and Y. Bengio, "Understanding the exploding gradient problem," *CoRR*, *abs/1211.5063*, 2012.
- [13] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training Recurrent Neural Networks," *arXiv [cs.LG]*, 21-Nov-2012.
- [14] F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain," in *Psychological Review*, 1958.
- [15] D. W. Ruck, S. K. Rogers, M. Kabrisky, M. E. Oxley, and B. W. Suter, "The multilayer perceptron as an approximation to a Bayes optimal discriminant function," *IEEE Trans. Neural Netw.*, vol. 1, no. 4, pp. 296–298, Dec. 1990.
- [16] H. T. Ng, W. B. Goh, and K. L. Low, "Feature Selection, Perceptron Learning, and a Usability Case Study for Text Categorization," in *Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Philadelphia, Pennsylvania, USA, 1997, pp. 67–73.
- [17] D. W. Ruck, S. K. Rogers, and M. Kabrisky, "Feature selection using a multilayer perceptron," *Journal of Neural Network Computing*, vol. 2, no. 2, pp. 40–48, 1990.
- [18] J. L. Elman, "Finding structure in time," in *COGNITIVE SCIENCE*, 1990.
- [19] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: continual prediction with LSTM," *Neural Comput.*, vol. 12, no. 10, pp. 2451–2471, Oct. 2000.
- [20] R. Jozefowicz, W. Zaremba, and I. Sutskever, "An empirical exploration of recurrent network architectures," *International Conference on*, 2015.
- [21] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," *arXiv [cs.CL]*, 01-Sep-2014.
- [22] M. Goto and Y. Muraoka, "Real-time beat tracking for drumless audio signals: Chord change detection for musical decisions," *Speech Commun.*, vol. 27, no. 3, pp. 311–335, Apr. 1999.
- [23] A. M. Stark and M. Davies, "Real-time beat-synchronous analysis of musical audio," *Proceedings of the 12th Int.*, 2009.
- [24] A. Gkiokas and V. Katsouros, "CONVOLUTIONAL NEURAL NETWORKS FOR REAL-TIME BEAT TRACKING: A DANCING ROBOT APPLICATION."
- [25] H. Nyquist, "Certain factors affecting telegraph speed," *J. A.I.E.E.*, vol. 43, no. 2, pp. 124–130, Feb. 1924.

- [26] J. London, "Cognitive Constraints on Metric Systems: Some Observations and Hypotheses," *Music Perception: An Interdisciplinary Journal*, vol. 19, no. 4, pp. 529–550, 2002.
- [27] P. N. Juslin and D. Västfjäll, "Emotional responses to music: the need to consider underlying mechanisms," *Behav. Brain Sci.*, vol. 31, no. 5, pp. 559–75; discussion 575–621, Oct. 2008.
- [28] F. van Veen, "The Neural Network Zoo - The Asimov Institute," *The Asimov Institute*, 14-Sep-2016. [Online]. Available: <http://www.asimovinstitute.org/neural-network-zoo/>. [Accessed: 03-Apr-2018].
- [29] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," *arXiv [cs.NE]*, 11-Dec-2014.
- [30] S. Durand, J. P. Bello, B. David, and G. Richard, "Robust Downbeat Tracking Using an Ensemble of Convolutional Networks," *arXiv [cs.SD]*, 26-May-2016.
- [31] D. E. Culler and Arvind, "Resource Requirements of Dataflow Programs," *SIGARCH Comput. Archit. News*, vol. 16, no. 2, pp. 141–150, May 1988.
- [32] R. Parncutt, "A model of the perceptual root(s) of a chord accounting for voicing and prevailing tonality," in *Music, Gestalt, and Computing*, 1997, pp. 181–199.
- [33] J. C. Brown, "An efficient algorithm for the calculation of a constant Q transform."
- [34] A. Klapuri, "Sound onset detection by applying psychoacoustic knowledge," *Acoustics, Speech, and Signal Processing, 1999*, 1999.
- [35] B. Moore, "Handbook of Perception and Cognition (Vol. 6): Hearing." New York: Academic Press, 1995.
- [36] M. Wojtczak, A. H. Mehta, and A. J. Oxenham, "Rhythm judgments reveal a frequency asymmetry in the perception and neural coding of sound synchrony," *Proc. Natl. Acad. Sci. U. S. A.*, vol. 114, no. 5, pp. 1201–1206, Jan. 2017.
- [37] "API Documentation | TensorFlow," *TensorFlow*. [Online]. Available: https://www.tensorflow.org/api_docs. [Accessed: 20-Apr-2018].
- [38] J. A. Bilmes, "Timing is of the essence : perceptual and computational techniques for representing, learning, and reproducing expressive timing in percussive rhythm," Massachusetts Institute of Technology, 1993.
- [39] "LibROSA — librosa 0.6.0 documentation." [Online]. Available:

- <https://librosa.github.io/librosa/>. [Accessed: 24-Apr-2018].
- [40] F. Bellard, M. Niedermayer, and Others, “FFmpeg,” 2012. [Online]. Available: <https://www.ffmpeg.org/>. [Accessed: 24-Apr-2018].
- [41] “SciPy.org — SciPy.org.” [Online]. Available: <https://www.scipy.org/>. [Accessed: 24-Apr-2018].
- [42] K. Jacobs, “Python Deep Learning tutorial: Create a GRU (RNN) in TensorFlow,” *Data Blogger*, 27-Aug-2017. [Online]. Available: <https://www.data-blogger.com/2017/08/27/gru-implementation-tensorflow/>. [Accessed: 21-Apr-2018].
- [43] V. N. Vapnik, “An overview of statistical learning theory,” *IEEE Trans. Neural Netw.*, vol. 10, no. 5, pp. 988–999, 1999.
- [44] *tensorflow/blob/master/tensorflow/python/training/adam.py*. Github.
- [45] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *arXiv [cs.LG]*, 22-Dec-2014.
- [46] G. E. Hinton, A. Krizhevsky, and S. D. Wang, “Transforming Auto-Encoders,” in *Artificial Neural Networks and Machine Learning – ICANN 2011*, 2011, pp. 44–51.
- [47] “Intel Unveils Strategy for State-of-the-Art Artificial Intelligence,” *Intel Newsroom*. [Online]. Available: <https://newsroom.intel.com/news-releases/intel-ai-day-news-release/>. [Accessed: 25-Apr-2018].