Classifying Chaotic Synthesizers with a Multimodal Neural Network

Thesis Subtitle

By:
Landon Buell[1]

Advised By:
Dr. Kevin Short[2] **, Dr. Maurik Holtrop**[1]

A thesis presented for the degree of
Bachelor of Science in Physics

December 2020

[1]Department of Physics and Astronomy
[2]Department of Mathematics and Statistics

University of New Hampshire
Durham New Hampshire, USA

Contents

1 Introduction

2 Methodology

We outline and detail the steps taken to execute this project from beginning to end. Does this section need to be included? I'm not sure how to format or organize it

2.1 Designing the Function

Consider the biological process of hearing a sound wave and matching it to a source. We can model this behavior by some unknown function F. We produce an approximation of that function F^* that can map the contents of a sound file that have been generated by a chaotic synthesizer to a potential source. For a set of inputs $\vec{x} = \{x_0, x_1, x_2, ..., x_{p-1}\}$ and a set of classes 0 through $k-1$, we denote this function as:

$$F^* : \vec{x} \to \{0, 1, 2, ..., k-1\} \tag{1}$$

In secs. (3.1) and (3.2), we introduce the nature and structure of a neural network and how it can use used in part to solve this problem. We show it's mathematical function analogue and how

2.2 Collecting and Pre-processing Raw Data

In order to train the Multimodal neural network to identify musical instrument sources, we need a suitable data set to present to the model as training data. University of Iowa Electronic Music Studio, and the London-Based Philharmonia Orchestra each have a large collection of publicly available audio files Citations!. These contain short segments of musical instruments performing a single note or a collection of notes in succession. Each sound file is labeled according to the instrument that is producing that sound.

To ensure that these data sets are formatted consistently, we read each sample from it's original format, *.AIF*, *.MP3*, or similar, and rewrite the data as a new *.WAV* files, sampled at 44.1 kHz with a 16 bit depth check this!. This ensures that all data will have a consistent format when features are extracted. This stage is detailed in sec.(4.0.2).

2.3 Designing Classification Features

The performance of a neural network is largely dependent of designing an appropriate set of classification features. These are properties of wave forms that can be represented by a numerical value or several numerical values and are used as the primary tool in classification. We detail these features in sec. (4). These are used in place of a full waveform to represent a file's contents in a low-dimensional array. We use tools from physics, mathematics, and signal processing to define and explore a comprehensive set of features that enables a high performance of the classifier.

2.4 Designing A Complementary Network Architecture

With the appropriate set of features designed, and the number of output determined, we can organize the structure of the neural network function. We construct a multimodal neural network, explored in sec. (3.6), that processes two input arrays derived from the same audio sample. This network is designed to handle and process each respective input independently, and concatenate the results to produce a single output.

2.5 Testing and Evaluating Network Performance

We divide the raw data set up into a training and testing sets. We employ cross-validation and compute the results of performance metrics to ensure that out model is making reliable predictions, and generalize appropriately. In this stage, we also choose the value of hyper-parameters, activation functions and layer widths to best compliment the chosen features. This process is repeated and expanded upon until we have produced a model with a sufficient performance.

2.6 Running Predictions of Chaotic Synthesizer Files

Once we have established a suitable performance of the model, we allow the model to run predictions on the un-labeled chaotic synthesizer wave forms. We output the prediction results to a file, and compare the neural network predictions against human predictions. If further corrections are needed, we revert and re-deign the features, architecture, or hyper-parameters as needed.

3 The Neural Network

3.1 An Introduction to Neural Networks

The task of algorithmically matching sound files to musical instrument classes in exceedingly difficult through conventional computer programming techniques. Because of the complexity of sound waves, brain's interpretation them, the challenge arises as to how to build some sort of program that could function at a level above general procedural or explicit instructional rules. Rather than *hard-coding* a set of conditions or parameters for a classification program, we seek an architecture that allows for a computer to *learn* and change and update itself as it is presented with more data. Such an algorithm exists in the form of a *Neural Network* [2, 3, 6].

A neural network is a mathematical function that seeks to produce an approximation F^*, of some usually unknown function F . For audio recognition, F is some function, or series of functions, that occurs in the brain which allows for a listener to take the sensation of audio, and map it to a known musical instrument. For a neural network, a model must construct an approximate function F^* using a set of parameters Θ, which allows the model to map a series of inputs, x (called *features*) to a series of outputs, y (called *predictions*) [3, 4, 17].

Former YouTube video classification team lead, and current machine learning consultant, Aurelien Geron writes about the relationship between biological brains and mathematical neural networks [1]:

> Birds inspired us to fly, burdock plants inspired velcro and nature has inspired many other inventions. It seems only logical, then, to look to the brain's architecture for inspiration on how to build an intelligent machine.

The result of such an analogy is a computer program that is reminiscent of the brain.

Over the course of their lives, humans will learn to map sounds to sources almost effortlessly. Given an example, and the appropriate label, humans can do this with very reasonable accuracy over a wide array of sounds [12, 18]. We can simplify the idea of humans recognizing sound as some function or operation that occurs within the brain, accepting an input sound, and produces an output label. Similarly, a neural network can be constructed, presented with *features* of multiple, labeled sound waves and learns a set of parameters Θ that allows for the mapping to a source.

For this particular project, a neural network has been constructed to perform a *classification* task. A classification task involves mapping the input data, x to one of k possible *classes*, each represented by an integer, 0 through $k - 1$. Each of the k classes represents a particular musical instrument that could have produced the sound-wave in the audio file.

3.2 The Structure of a Neural Network

A Neural Network is simply a model of a mathematical function, composed of several smaller mathematical functions called *layers* [3, 8]. Each layer represents an operation that takes some real input, typically an array of real double-precision floating-point numbers, and returns a modified array of new double-precision floating-point numbers. The exact nature of this operation can be very different depending on the layer type or where it sits within the network. It is this process of transforming inputs successively in a particular order until an output is attained that makes up the core functionality of a neural network [1, 8]. This output encodes the models final "decision" given a unique input.

Other than inspiration from the brain, Neural Networks are name *networks* because of their nested composition nature. The model can be represented by a linear or acyclic computational graph which successively maps exactly how the repeated composition is structured. In a *feed-forward network*, information is passes successively in one direction. For example functions could be chained together such as [3]

$$F(x) = f^{(L-1)}\big[f^{(L-2)}\big[...f^{(1)}\big[f^{(0)}\big[x\big]\big]\big]...\big] \tag{2}$$

Where each function $f^{(l)}$ represents a layer of the network model. The number of layers in a neural network is referred to as the network *depth*. The dimensionality of each layer is referred to as the network *width* [1, 8].

A network model that contains L unique layers is said to be an L-Layer Neural Network, with each layer usually indexed by a superscipt, 0 through $L-1$. Layer 0 is said to be the *input layer* and layer $L-1$ is said to be the *output layer*. The function that represents a layer (l) is given by

$$f^{(l)} : x \in \mathbb{R} \rightarrow y \in \mathbb{R} \tag{3}$$

The value of x can also be index by layer, we call the array $x^{(l)}$ the *activations* of layer l [3, 8].

The model is recursive by nature, the activations from one layer, $l-1$, are used to directly produce the activations of the next successive layer l. Thus eqn. (3) can be alternatively written as:

$$f^{(l)} : x^{(l-1)} \in \mathbb{R} \rightarrow x^{(l)} \in \mathbb{R} \tag{4}$$

The array of activations, $x^{(0)}$, is the raw input given the neural network, called *input* activations. Conversely, $x^{(L-1)}$ is referred to as*output* activations [1, 4, 8].

Algorithm 1 Forward propagation system in a standard deep neural network. Each iteration in the main *for-loop* represents the execution of a layer, and passing the result to the "next" layer function. We assume each layter to follow a two-step structure, being (i) a linear transformation as in eqn. (5) and (ii) an element-wise non-linear activation function as in eqn. (7). A practical application of this algorithm should include batches of samples instead of a single sample. I Want to include this algorithm for context - especially on the programming end, but am not sure where to put it.

Require: Network with L layers - $0, 1, 2, ..., L-2, L-1$

Require: Set of layer functions $f^{(l)}, l \in 0, 1, 2, ..., L-2, L-1$ made up of a linear transformation and a non-linear transformation.

Require: Input sample - $x^{(0)}$

If training the network, the intermediate layer activations must be stored for back-propagation. Otherwise, they must be discarded. $Z = \{\} \leftarrow$ array to hold pre-activations

$X = \{\} \leftarrow$ array to hold final activations

for $l = 1, 2, ..., L-2, L-1$ **do**

Evaluate Linear Transformation

$z^{(l)} \leftarrow$

Apply non-linear Activation Function

$x^{(l)} \leftarrow \sigma^{(l)}\left[z^{(l)}\right]$

Store values for later use in optimizer, otherwise can be discarded.

$Z \leftarrow Z.add(z^{(l)})$

$X \leftarrow X.add(x^{(l)})$

end for

Network final prediction is last object in 'act' array.

$y^* \leftarrow x^{(L-1)} = X[-1]$

Return the pre-nonlinear activations and final-nonlinear activations.

The final index of X represents the neural network's final prediction.

return Z, X

3.3 Layers Used in Classification Neural Network

As stated previously, a neural network is composed of a series of functions that are called successively to transform features (an input) into a prediction (an output). As shown in eqn. (4), each function feeds directly into the next as to form a sort of recursive computational graph [3].

Typically, a layer function can be divided into two portions: (i.) a Linear transformation, with a bias addition, and (ii.) an element-wise activation transformation. This activation function typically is a non-linear function which allows for the modeling of increasingly complex decision-boundaries. The Linear transformation can be given by eqn (5) in the case of a 1D Dense layer (3.3.1) or eqn. (6) in the case of a 2D Convolution layer.

$$z^{(l)} = W^{(l)} x^{(l-1)} + b^{(l)} \tag{5}$$

$$z^{(l)} = \left(W^{(l)} * x^{(l-1)}\right) + b^{(l)} \tag{6}$$

Where $W^{(l)}x^{(l-1)}$ denotes a matrix-vector product, and $\left(W^{(l)} * x^{(l-1)}\right)$ denotes a convolution product. In both cases, $W^{(l)}$ is the *weighting-matrix* or *weighting-kernel* for layer l, $b^{(l)}$ is the *bias-vector* for layer l, $z^{(l)}$ are the *linear-activations* for layer l and $x^{(l-1)}$ is the final activations for layer $l-1$.

Step (ii.) is usually given by some *activation function*:

$$x^{(l)} = \sigma^{(l)}\left[z^{(l)}\right] \tag{7}$$

Where $x^{(l)}$ is the final activations for layer l and $z^{(l)}$ is given in equation (5). $\sigma^{(l)}$ is some activation function, which is often use to enable the modeling of more complex-decision boundaries.

The combination of eqn. (5) and eqn. (7), for a layer l , make up the function represented by that layer, as in eqn. (4). Below, we discuss and describe the types of layer functions that are used to produce the classification model in this project.

3.3.1 Dense Layer

The Linear Dense Layer, often just called a *Dense* Layer, or *Fully-Connected* layer for short, was one of the earliest function types used in artificial neural network models [2, 8]. They are among the most commonly used layer types. A dense layer is composed of a layer of *artifical neurons*, each of which holds a numerical value within it, called the *activation* of that neuron. This idea was developed back from McCulloch and Pitts' work [9], and was expanded upon by Frank Rosenblatt in 1957 [1].

We model a layer of neurons as a vector of floating-point numbers. Typically, it is required that the array be one-dimensional. Suppose a layer (l) contains n artificial neurons. We denote the array that hold those activations as $x^{(l)}$ and is given by:

$$\vec{x}^{(l)} = \Big[x_0, x_1, x_2, \ldots, x_{n-2}, x_{n-1}\Big]^T \tag{8}$$

The activation of each entry is given by a linear-combination of activations from the previous layer, as outlined in eqn.(5) and eqn.(7).

Suppose the layer $l-1$ contains m neurons. Then the weighting-matrix, $W^{(l)}$ has shape $m \times n$, the bias-vector $b^{(l)}$ has shape $m \times 1$. We can denote the function in a similar manner to eqn.(4) as:

$$f^{(l)} : x^{(l-1)} \in \mathbb{R}^{(m\times 1)} \rightarrow x^{(l)} \in \mathbb{R}^{(n\times 1)} \tag{9}$$

Thus for a dense layer l, the exact values of each activation is given by [1, 8]:

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}^{(l)} = \sigma^{(l)} \left(\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,m-1} \\ w_{1,0} & w_{1,1} & \dots & w_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n-1,0} & w_{n-1,1} & \dots & w_{n-1,m-1} \end{bmatrix}^{(l)} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-1} \end{bmatrix}^{(l-1)} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}^{(l)} \right) \tag{10}$$

or more compactly:

$$x^{(l)} = \sigma^{(l)} \Big(W^{(l)} x^{(l-1)} + b^{(l)} \Big) \tag{11}$$

Eqn. (11) is generally referred to as the *dense layer feed-forward equation* [3].

3.3.2 2-Dimensional Convolution Layer

Convolution layers emerged from studying the brain's visual cortex, and have been used from image recognition related tasks for around 40 years. [1, 8]). This layer get's it's name from it's implementation of the mathematical *convolution* function. The 2D discrete-convolution of function or 2D array $A[i, j]$ and $B[i, j]$ is given by [3]:

$$C[i, j] = (A * B)[i, j] = \sum_u \sum_v A[i, j] B[i - u, j - v] \tag{12}$$

Note that convolution is commutative: $(A * B) = (B * A)$. We implement a convolutional layer $f^{(l)}$), by creating a series of K weighting matrices, called *filters*, each of size $m \times n$, where $m, n \in \mathbb{Z}$. Often we choose $m = n$ and call the shape the *convolution kernel size* [8, 3].

In the case of a 2D input array, $x^{(l-1)} \in \mathbb{R}^{(N \times M)}$, the convolutions filters "step" through the input data and repeatedly compute the element-wise product of each $m \times n$ weighting kernel, and the local activation of the $x^{(l-1)}$ array. The step size in each of the 2-dimension is known as the *stride*. Each of the K filters are used to generate a $m \times n$ feature map from the input activation. For an appropriately trained network, some filters may be dedicated to detecting vertical lines, horizontal lines, sharp edges, areas of mostly one color, etc. [1, 8].

In general, for an $N \times M$ input, with kernel size $m \times n$, with stride size 1×1, and K feature maps.

$$f^{(l)} : x^{(l-1)} \in \mathbb{R}^{(N \times M)} \rightarrow x^{(l)} \in \mathbb{R}^{([N-n+1] \times [M-m+1] \times K)} \tag{13}$$

As an example, consider a single filter map over input $x \in \mathbb{R}^{(4 \times 3)}$ and filter map $W \in \mathbb{R}^{(2 \times 2)}$ as shown in fig. (**??**):

a	b	c	d
e	f	g	h
i	j	k	l

(a) Input Activations

w	x
y	z

(b) Convolution Filter

aw + bx + ey + fz	bw + cx + fy + gz	cw + dx + gy + hz
ew + fx + iy + jz	fw + gx + jy + kz	gw + hx + ky + lz

(c) Convolution Result

Figure 1: The result of convolving an input (a) with an filter map (b) is a new set of activations (c). This Image was adapted from Goodfellow, pg. 325 [3]

Note that formal implementations include a bias vector and an activation function.

For 2D convolution over input $x^{(l-1)}$ with feature map $W_k^{(l)}$, producing activations $x^{(l)}$, we compute the activations $x_k^{(l)}$ as [3]

$$x_k^{(l)} = \sigma^{(l)}\left[\left(\sum_{u=0}^{m-1}\sum_{v=0}^{n-1} x^{(l-1)}(i,j)W_k^{(l)}(i-u,j-v)\right) + b_k^{(l)}\right] \tag{14}$$

Please check my math here! This operation repeats for each of the K feature maps. Each maps has it's own $n \times m$ weighting matrix and appropriately shaped bias .

The convolution layers allows for several advantages. Among these are (i) *sparse-connectivity*: not every single activation (pixel) is connect to every single output pixel, so it is more computationally efficient, (ii) *positional invariance*: key features can be identified regardless of where they are in the layer, and (iii.) *Automatic feature detection* as the training process will update the filters to identify dominant features in the data without human instruction [1, 3, 8].

3.3.3 2-Dimensional Maximum Pooling Layer

A Maximum Pooling layers simply returns the maximum neuron activation in a group of neurons. In the case of 2D Max Pooling, we choose a kernel size to be $m \times n$, just like in the 2D Convolution layer, and extract the maximum value in each window, and then step along accrdoing to the chosen stride size [8, 3].As an example, consider again figure (**??**a). Using the 2×2 kernel on the 3×4 input, each box would then contain the maximum value of the input activations. We detail this in fig. (**??**):

Max(a,b,e,f)	Max(b,c,f,g)	Max(c,d,g,h)
Max(e,f,i,j)	Max(f,g,j,k)	Max(g,h,k,l)

Figure 2: The result of 2D maximum-pooling an input array. This image was adapted from Loy, pg. 126 [8]

Pooling layers, such as maximum pooling, average pooling, or similar are typically placed after a layer or a set of layers of convolution. The purpose of this arrangement is to reduce the number of weights, thereby dropping the complexity of the model and ensuring only features with large activation values are preserved [1, 8, 3].

3.3.4 1-Dimensional Flattening Layer

A flattening layer is used to compress an array with two or more dimensions down into a single dimension. For a flattening layer l, multidimensional activations in layer $l-1$ are compressed down into a single dimensional array. We can use function notation to express this as:

$$f^{(l)} : x^{(l-1)} \in \mathbb{R}^{(M \times N \times ....)} \rightarrow x^{(l)} \in \mathbb{R}^{(MN... \times 1)} \tag{15}$$

The numerical value of each activation is left unchanged. For a layer with activation shape$N \times M$, the resulting activations are reshaped into $NM \times 1$ as shown in eqn (15).

Flattening Layer are most commonly used to prepare activations for entry into dense layer or series of dense layers. For example, 2D or 3D images are typically processed with 2D convolution, which may output a 2D or 3D array of activations. Those values are then flattened to 1 dimensions, which can then be passed into dense layers for further processing.

3.3.5 1-Dimensional Concatenation Layer

The 1-Dimensional Concatenation Layer, also called 1D-Concat layer takes separate vectors of activations and combines them into a single 1D vector. Consider the layer activations $\vec{a}^{(l-1)}$ and $\vec{b}^{(l-1)}$ with shapes $1 \times \alpha$ and $1 \times \beta$ respectively. They are the outputs of two different layers:

$$\vec{a}^{(l-1)} = \big[a_0, a_1, ..., a_{\alpha-1}\big] \qquad \text{and} \qquad \vec{b}^{(l-1)} = \big[b_0, b_1, ..., b_{\beta-1}\big] \tag{16}$$

The result of the concatenation is a new 1D-array, $\vec{c}^{(l)}$ with size $1 \times \alpha\beta$:

$$\vec{c}^{(l)} = \big[a_0, a_1, ..., a_{\alpha-1}, b_0, b_1, ..., b_{\beta-1}\big] \tag{17}$$

We can denote this for n arrays with function notation:

$$f^{(l)} : x_a^{(l-1)} \in \mathbb{R}^{(1\times\alpha)}, x_b^{(l-1)} \in \mathbb{R}^{(1\times\beta)}, ... \rightarrow x_z^{(l)} \in \mathbb{R}^{(1\times\alpha\beta...)} \tag{18}$$

The concatenation layer is used to combine activation non-adjacent, or non-related layers into a single new layer. In this case of the model used in the project, we use it to combine the activations that result from two distinct models into a single layer that is then transformed into a final output. This process is detailed further in sec. (3.6).

3.4 Activation Functions Used in Network Layers

Activation functions are a key parameter in the behavior of neural networks. As discussed in sec. (**??**) a layer function, $f^{(l)}$ is generally composed of a linear transformation, as in eqn. (5), and then an element-wise activation function as in eqn. (7). It is this second step that allows for neural networks to model the incredibly complex decision boundaries that are found in real-world problems [1, 8]. In this section, we detail the activation functions used in out classification neural network.

3.4.1 Rectified Linear Unit

The Rectified Linear Unit (ReLU) activation function acts element-wise on the activations in a given layer. If the activation of a neurons is non-zero or non-negative, the value is untouched, otherwise a 0 is returned. For an input activation x, ReLU is defined by:

$$\text{ReLU}\big[x\big] = \max\big(0, x\big) = \begin{cases} x & \text{if} \quad x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{19}$$

ReLU is only defined on a real domain. We provide a visualization of the function in fig. (**??**).

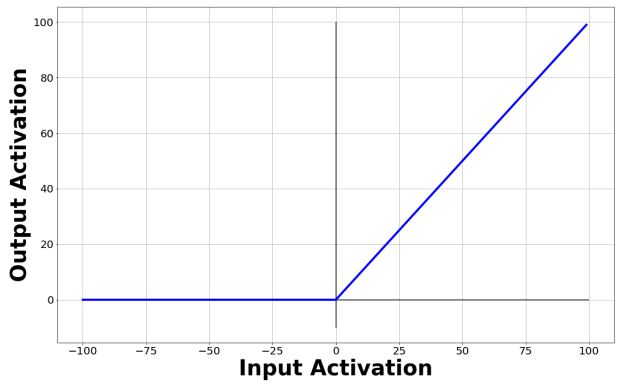

Figure 3: Rectifed Linear Unit (ReLU) activation function

For our chosen architecture, ReLU is applied to the activations in every single Convolution layer and Dense Layer, with the exception of the output dense layer.

3.4.2 Softmax

The softmax activation function is commonly used in the output layer of a multi-class classification network, such as the one we implement. When softmax acts on a vector or activation, the result a non-negative output vector. with an L_1 norm of 1 [1, 3, 17]. The i-th element in a softmax activation function is given by:

$$\text{Softmax}\big[x\big]_i = \frac{1}{\sum_j e^{x_j}} e^{x_i} \tag{20}$$

This function is almost only used in the output layer of a neural network, and encodes a discrete categorical probability. For example, if eqn.(20) returns a vector of the form: $[0.75, 0.25]$, we interpret this as a sample having a 75% chance of beloinging to class 0 and a 25% change of belonging to class 1.

3.5 Training a Neural Network

A neural network's purpose is to produce a function F^* that approximates an unknown function F, using a set of parameters, Θ. The model must have a procedure for updating the parameters in Θ to allow for a reasonably close approximation of F. To better understand this, we turn to Tom Mitchell's explanation of a learning algorithm [3, 11]:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if it's performance at tasks in T, as measured by P, improves with experience E.

Without any human intervention, a model must update itself to improve it's performance at a give task a new information is presented to it. To do this, the model must be constructed with a training procedure in mind.

We consider the set of parameters Θ as a concatenation of each applicable layer's weighting matrix and bias vector such that:

$$\Theta = \{W^{(0)}, b^{(0)}, W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, ..., W^{(L-2)}, b^{(L-2)}, W^{(L)}, b^{(L-1)}, \} \tag{21}$$

Each parameter is floating-point number, which is stored in a weight matrix $W^{(l)}$ or bias vector $b^{(l)}$. Each value for each parameter contributes to the output of the function. Note that $W^{(l)}$ or $b^{(l)}$, itself is a parameter to train, but rather the entries that make up those structures must be trained. In increasingly complicated neural networks, there can be upwards of hundreds of thousands,or even millions of elements in θ making a neural network a functions that exist is a very high dimensional parameter-space [1, 3].

3.5.1 The Cost Function

Suppose we pass a training sample, given by the feature-vector $x^{(0)}$, with an expected outcome given by the vector y, into the neural network. After the network finishes processing, it's output activations are given by the vector $x^{(L-1)}$, also noted at y^*. For a reasonably trained model, we expect the vector y^* to be "similar" to y, indicating that with the provided data, the network has made a valid prediction. Conversely, for an untrained network, y^* is not likely to be "similar" to y at all, indicating that the network has made a poor prediction.

To quantify the difference between the model's prediction, y^* and the expected output, y, we introduce a *cost function*, $J(y^*, y)$, to the network [3, 4]. The cost function, also called a *loss* or *objective* function, compares the neural networks output, y^* and the expected output y. It returns a single value, usually a floating-point number which measures the *poorness* of the prediction. A high cost value indicates a *poor* prediction, and a low cost indicates a reasonable prediction.

$J(y^*, y)$ can take many forms and is often dependent on particular task or data set provided [4]. For this k-classes classification task, we choose to use a *Categorical Crossentropy* (CXE) cost function. For a single predictions sample, it is defined [3]:

$$\text{CXE}\big[y^*, y\big] = -\sum_{i=0}^{k-1} y_i^* \log_{10}(y_i) \tag{22}$$

In practice, this operation is performed over a batch of samples and an average is computed and used a *batch loss* [3].

3.5.2 Gradient Based Learning

We have developed a method for quantifying the difference between a given output y^* and an expected output y with the inclusion of a cost function. For a trained neural network, we expect that samples consistently produce a low cost value, particularly on previous unseen inputs, which indicates that the model has appropriately *generalized*. The process of training

the model is to choose the parameters in Θ that allows for this behavior of the cost function. We then treat the training process of a neural network as a higher dimensional *optimization* problem.

A neural network uses *indirect optimization*, which contrasts from pure optimization. Deep Learning expert Ian Goodfellow described this [3]:

> In most learning scenarios, we care about some performance measure P,... We reduce a different cost function, $J(\theta)$ in the hope that doing so will also imprvce P.

By choosing an appropriate cost function, and selecting the parameters Θ that minimize the average cost over a data set, we also assume that doing so minimized the error in the performance P.

The cost of a sample is dependent on training labels y and the network output y^*. The output is given by the final layer activation $x^{(L-1)}$, which in turn are produced by the previous layer and so forth, as demonstrated in eqn. (4), This recursive nature combined with the dimensionality of the parameter object Θ makes an analytical solution to the optimization impractical [1, 3, 4]. We instead optimize with a gradient-based method.

We can reduce the cost given a set of paramaters, $J(\Theta)$, by moving each element in Θ with small steps in the direction of the negative partial derivative of each parameter. In the case of the high-dimensional Θ object, we compute the partial derivative with respect to each weight and bias in the form of the *gradient vector*. We denote the gradient of J with respect to the parameters in Θ as:

$$\nabla_\Theta \big[J\big] = \Big[\frac{\partial J}{\partial W^{(0)}}, \frac{\partial J}{\partial b^{(0)}}, \frac{\partial J}{\partial W^{(1)}}, \frac{\partial J}{\partial b^{(1)}}, \ldots, \frac{\partial J}{\partial W^{(L-1)}}, \frac{\partial J}{\partial b^{(L-1)}}\Big]^T \tag{23}$$

Where $\frac{\partial J}{\partial W^{(l)}}$ is taking the partial derivative of each element in the $W^{(l)}$ matrix, and preserves the shape.

Due to the nested composition of the network output in eqn. (2), and subsequently the cost itself, we must use the chain rule of calculus to work backwards through the neural network to compute the partial derivative with respect to each parameter in Θ. The process of working backwards to compute each element in the gradient vector is called *back-propagation* [1]. Below we detail an algorithm in pseudo-code to back-propagate through a fully-connected nueral network, such as the multilayer perceptron outlines in sec.(**??**)

Algorithm 2 Backwards propagation system, in a standard densely connected deep neural network. Each iteration in the *for-loop* computes the gradient of the cost function J with respect to the weight and bias arrays. Each element in those arrays is then the discrete gradient of that parameter. A practical application of this algorithm should include batches of samples instead of a single sample

Require: Cost/Objective function J and learning rate α
Require: Set of weighting parameters - $W^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of bias parameters - $b^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of layer activation function - $\sigma^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of layer activation function derivatives - $\partial\sigma^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of pre-nonlinear activation - $Z^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of post-nonlinear activation - $X^{(i)}, i \in \{0, 1, ..., L-1\}$
  Execute forward pass in algorithm (4) and compute the gradient of the cost with respect to the final layer activations
  $dx \leftarrow \nabla_{(y^*)} J(y, y^*)$
  Intialize ∇J as output object, should have same shape as Θ
  for $L-1, L-2, ..., 2, 1$ **do**
    Compute gradient w.r.t pre-nonlinear activation portion of layer function
    $dx^{(l)} \leftarrow \nabla_{Z^{(l)}} J = dx^{(l)} \odot \partial\sigma^{(l)}[Z^{(l)}]$
    Compute gradient w.r.t weighting and bias elements
    $db \leftarrow \nabla_{b^{(l)}} J = dx^{(l)}$
    $dW \leftarrow \nabla_{W^{(l)}} J = dx^{(l)} \cdot X^{(l-1)}$
    Add db and dW steps to ∇J object
    $\nabla J = \nabla J.Add(dW, db)$
  end for
  Return gradient w.r.t to each parameter in Θ
  return ∇J

After (i) computing the gradient, we can scale is by a desired learning rate, α and (ii) add it element-wise to the exisitint elements in Θ. By repeating steps (i) and (ii) in sucession, we gradually drive the cost function to produces consistently lower and lower values. This is called *gradient descent* and is the basis for many optimization algorithms. We show the general update rule on iteration (i) for gradient based learning over a batch of m samples in eqn. (24).

$$\Theta^{(i)} = \Theta^{(i-1)} + (-\alpha)\frac{1}{m}\nabla_{\Theta}\Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)})\Big] \tag{24}$$

3.5.3 The Optimizer

For this project, we employ an *Adaptive-Moments* optimization algorithm, also called *ADAM* for short. It is a powerful variant of algorithms with an adaptive learning rate. It tracks an exponentially decaying average of past gradients, as well as exponentially decaying

average of past squared gradients [1]. This produces a far more aggressive optimizer at a higher computational cost. For a given iteration (i), The ADAM update is given:

$$
\begin{aligned}
s^{(i)} &= \rho_1 s^{(i-1)} + (1-\rho_1)\frac{1}{m}\nabla_\Theta\Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)})\Big] \\
r^{(i)} &= \rho_2 r^{(i-1)} + (1-\rho_2)\frac{1}{m}\nabla_\Theta\Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)})\Big] \odot \frac{1}{m}\nabla_\Theta\Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)})\Big] \\
s'^{(i)} &= \frac{s^{(i)}}{1-\rho_1^t} \\
r'^{(i)} &= \frac{r^{(i)}}{1-\rho_2^t} \\
\Theta^{(i)} &= \Theta^{(i-1)} + (-\alpha)\frac{s'^{(i)}}{\sqrt{r'^{(i)}} + \delta}
\end{aligned}
\tag{25}
$$

ADAM has experimentally shown to be a very powerful and robust optimizer useful for a wide range of tasks. Because of the two decay constants ρ_1 and ρ_2, we an compound and accumulate the values of past gradients to continue to push the cost to lower and lower values, even if the magnitude of the gradient becomes very small [1].

Algorithm 3 Adaptive-Moments (ADAM) optimizer for a neural network

Require: Step size α
Require: Small constant δ for numerical stabilization, usually about 10^{-7}.
Require: constants ρ_1, ρ_2 used got exponential decay rates, usually 0.9 and 0.999 respectively.
Require: Subroutine/function to compute gradient of cost function.
Require: Mini-batch size, m
Require: Stopping criterion S
Initialize moment variables and iteration counter $s = 0$, $r = 0$, $i = 0$
while Stopping Criterion S is **false do**
Extract a mini-batch of m samples from larger data set X. $[x^{(0)}, x^{(1)}, ..., x^{(m-1)}]$ and corresponding target values $[y^{(0)}, y^{(1)}, ..., y^{(m-1)}]$.
Compute numerical gradient estimate of each sample in batch. This can be done with standard back-propagation in algorithm (2) and normalize by batch size m:
$\nabla J \leftarrow \frac{1}{m}\nabla_\Theta\Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)})\Big]$
Compute first bias moment: $s \leftarrow \rho_1 s + (1-\rho_1)\nabla J$
Compute second bias moment: $r \leftarrow \rho_2 s + (1-\rho_2)\nabla J \odot \nabla J$
First bias correction: $s' \leftarrow \frac{s}{1-\rho_1^i}$
Second bias correction: $r' \leftarrow \frac{r}{1-\rho_2^i}$
Compute And Apply update: $\Delta\Theta = (-\alpha)\frac{s'}{\sqrt{r'}+\delta}$
$\Theta = \Theta + \Delta\Theta$
Update Iteration number: $i \leftarrow i + 1$
end while

3.6 Chosen Model Architecture

The success of a neural network algorithm is enormously dependent of the strength of the chosen features, as well as a complementary architecture, rather than the quantity of available data [1, 5, 7]. We have derived an appropriate set of inputs arrays See Section on Features, and constructed a neural network architecture to coincide with these features. From these two input arrays, one being a matrix and the other being a column-vector, we have produced a *multimodal neural network*, that process the two inputs, and combines separate neural networks into a single model to produce a one prediction resulting from both inputs citation for multimodal networks.

The full classification neural network used for this project consists of two distinct entry points. Rather than presenting the network with one set of input data, X, we present the network with two different arrays, X_1 and X_2. Both arrays are a product of the same audio file sound wave, and thus share a common training label, y. Each of the two entry layers leads to it's own branch and the given information is process roughly in parallel. Once a certain number of layer functions have been applied, each arm of the neural network is left with a single dense layer of activations. These two activation vectors are concatenated into a

single new dense layer, and then passed through one final layer to generate the model's final output. A visual representation of this architecture can be found in fig. (**??**).

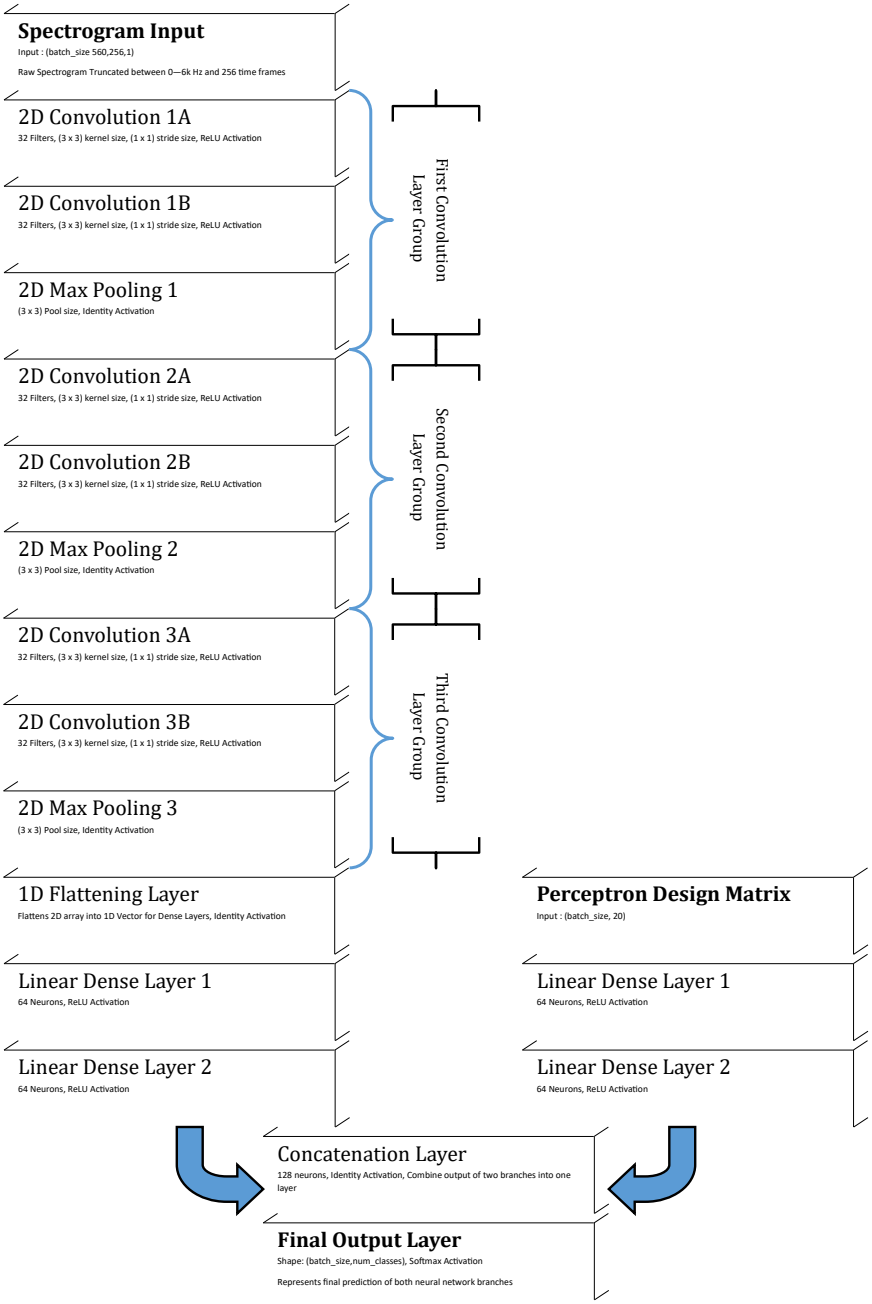

Figure 4: The developed architecture of the audio file classification neural network. The Left branch process an image-like input, the right branch processes a vector-like input. The activations are then merged, and then a single output is produced

3.6.1 The Spectrogram Branch

The spectrogram branch is pictured on the left side of fig. (**??**). A spectrogram is a representation of a sound wave, or signal by representing energy distributions as a function of *time* and *frequency* [18, 12, 5]. Details on the creating of the spectrogram can be found in the *features* section of this document.

The input layer of the spectrogram accepts a 4-Dimensional array. The axes, in order of indexing, represents (i) the size of the *mini-batch* of samples, (ii) the pixel width of each sample, (iii) the pixel height of each sample, and (iv) the number of channels in each sample. As a model hyper-parameter, we have chosen each batch to contain 64 samples. We have also chosen to truncate time and frequency axes (ii & iii) to contain 560 frequency bins, and 256 time-frames. Each image also contains a single channel, which make it gray-scale when visualized. We can denote the 4D shape of the input object into this branch as:

$$X_1 \in \mathbb{R}^{(64\times560\times256\times1)} \tag{26}$$

Any other shape will be rejected by the model, and an error is raised.

After the input layer assert the appropriate shape, the array X_1, which is a collection of 64 spectrograms, is passed into the first of three *Convolution Layer Groups*. These layer groups are inspired from the *VGG-16 Neural Network* architecture [citation needed]. Each convolution layer group in composed of three individual layers:

1. A 2-Dimensional Convolution layer, using 32 filters, a 3×3 kernel, a 1×1 step size, and a ReLU activation function,
2. A 2-Dimensional Convolution layer, using 32 filters, a 3×3 kernel, a 1×1 step size, and a ReLU activation function,
3. A 2-Dimensional Maximum Pooling layer using a 3×3 pooling size, and an Identity activation function

The Convolution layers convoluted over the middle two axes of the data (over space and time in the spectrogram).

By grouping layers in this structure, we use the convolution layers to reduce the number of features, and then the pooling layer to extract only the largest activations of the remianing features. This ensures that in the full spectrogram image, only the dominant features and feature shapes are preserved and passed into the next layer set for processing.

3.6.2 The Perceptron Branch

The Perceptron branch is picture on the right side of fig. (**??**), notice that it is considerably smaller in side and complexity than it's neighbor. Rather than accept an image-like input,

the perception simply takes a vector-like input of properties that are derived from the audio file. We call these properties *features* [1, 5, 17]. Details on the creating of these features can be found in the *features* section of this document.

The input layer of the perceptron accepts a 2-Dimensional array. The axes, in order of indexing, represent (i) the size of the *mini-batch* of samples, (ii) the number of features for each sample. We use the same model hyper-paramater of 64 samples per batch, and have developed 20 unique classification features that have been derived from time-space and frequency-space representations of the audio file data. We can denote the 2D shape if the input object into this branch as:

$$X_2 \in \mathbb{R}^{(64\times 20)} \tag{27}$$

This 2D is referred to as a *design matrix* [4, 8]. Any other shape will be rejected by the model, and an error is raised.

In perceptron models, scaling the design matrix appropriately drastically improves the classification performance. A design matrix is scaled by taking all samples in a column (a particular feature from each class), subtracting the average, and then scaling it such that it has unit variance [1, 4]. This ensures that no one feature dominates the performance of the model and that layer activations to not get saturated as data progresses through the network.

3.6.3 The Final Output Branch

The last layer in each of the two branches is a ReLU-activated Dense layer containing 64 neurons, represented by a 64×1 column vectors. We combine these model layers by using a *concatenation layer* to fuse the two column vectors together, "vertically", such that the result is a 128×1 column vector. This vector is then passed into the final dense layer, which used the softmax activation, and encodes the joint predictions based on both model branches.

4 Feature Selections

Audio and Machine learning researchers M. Kashif Saeed Khan and Wasfi G. Al-Khatib discuss the vitality of feature selection [**?**]: I'm not sure how to introduce the importance of feature extraction

> The data reduction stage which is also called feature extraction, consists of discovering a few important facts about each class. The choice of features is critical as it greatly affects the accuracy of audio classification.

In order to properly train a neural network, the model must be presented with an appropriate set of training input x and a complementary set of training labels y [1, 3, 4]. It becomes quickly apparent that the nature of information contained in the input object x, called *features*, is *extremely important* to the performance of the classifier. Consider the task of identifying cats and dogs from images, given only images consisting of the top-most row of pixels. The inappropriate choice features does not reflect the significant characteristics of the signal.

Tuomas Virtanen, machine learning and audio engineer writes in his book, "Computational Analysis of Sound Scene and Events" about specifically how features must be chosen [17]:

> For recognition algorithms, the necessary property of the acoustic features is low variability among features extracted from examples assigned to the same class, and at the same time high variability allowing distinction between features extracted from examples assigned to different classes.

In constructing a neural network classifier, the development of appropriate features is of the utmost importance. To ensure the construction of a suitable model, we derive features based from three sources (i) a spectrogram matrix of the waveform, (ii) the time-space representation of the waveform, and (iii) the frequency-space representation of the waveform. It is important to note that although this algorithm will classify sound waves to instruments, the model will never actually be presented with a waveform directly, instead it will rely solely these features.

Once we produce an sufficient set of features, we concatenate them into a single object, $\hat{x}$, called the *feature-vector* [3]. In the training process this object, along with the appropriate classification label, y is presented to the neural network for processing. This process of constructing a feature vector from any data set is vital and is used to represent the data set in a far more compact and non-redundant format [17, 7]

To ensure suitable performance of this sound wave classification neural network, a great deal of time has been devoted to the construction of the elements of the feature vector. These features are derived are principles of music, digital signal processing, previous work success, and most importantly physics. In the following sections, we outline the set of 21 features used in the classification process

4.0.1 The Design Matrix

The 21 features are from each file are organized into one of two array structures. each called a *design matrix*. A design matrix is an N-Dimensional array of real floating-point numbers that stores the features in a convenient way for the neural network to digest. For this hybrid neural-network, we use two matrices, X_1 and X_2, each containing data from b audio file samples. each has shape given by:

$$X_1 \in \mathbb{R}^{(b \times N' \times k \times 1)} \tag{28}$$

$$X_2 \in \mathbb{R}^{(b \times P)} \tag{29}$$

Where $Q' \times R' \times 1$ gives the shape of a single spectrogram matrix S, described in sect. (4.1). Similarly, P is the number of features derived from the time and frequency-representation of features outlined in sec. (4.2) and sec.(4.3).The first axis, with size b indicates that there are b samples stored in that design matrix. A row of X_2 has shape $1 \times P$ where there are P features within the *feature-vector* for each sample. Matrix X_1 is presented to the convolutional branch of the network to be process as an a collection of images, and matrix X_2 is presented to the perceptron branch to be processed as a collection of vectors.

4.0.2 Audio Preprocessing

Preprocessing a data set is a necessary step to execute prior to feature extraction [2, 4]. In the case of audio files, preprocessing usually consists of ensuring that the data set contains the following:

1. A suitably sized number of files, of reasonable audio quality, with normalized amplitudes
2. Audio encoded in a standard, and consistent format
3. A consistent sample rate and bit depth between audio files
4. A consistent number of channels

Note that different projects may require a different set of requirement from preprocessing [17]. For this project, we have chosen to use the following parameters:

1. Roughly 4000 audio files Professionally or semi-professionally recorded in a studio. Citation needed!. All ampltiudes have been normalized to ± 1 unit.
2. All audio has be converted into *.WAV* files from other formats, such as *.AIF* or *.MP3* using a MATLAB program
3. All audio is sampled at $44,100$ Hz and Bit depth needed - 16? 24?
4. All audio has been down-mixed into mono-channel waveforms.

4.1 Spectrogram Feature

The field of neural classification is well studied in the application of image-processing. Many large-scale, and even introductory projects find themselves under the umbrella of image classification [1, 3, 8, 10]. As a result, model architectures for image processing related tasks are well-explored and have shown experimentally successful behavior. Following this success, it make senses to provide an image-like representation of a sound wave as a feature. We do this in the form of a spectrogram matrix.

A spectrogram is a representation of the energy distribution of a sound wave as a function of both frequency and time. In a conventional spectrogram, the passing of time is shown along the x-axis, and the frequency spectrum is shown on the y-axis. Thus each point in the 2-Dimensional space is an energy at a given time and frequency. Examples spectrograms from the wave form data set are shown in Fig. (**??**).

Insert spectrograms here

Figure 5: Spectrogram representations of various waveforms

A spectrogram is produced by the method of *frame-blocking*, which is very prevalent in audio signal classification. Frame-blocking takes a raw waveform or signal, s and decomposes it into a set of analysis frames, a_i, with each being N samples in length, and has a fixed overlap with the next adjacent frame. Each of the k frames then allows for a section of the signal in somewhat stationary state [7, 19, 5, 17]. For this project, we have chosen to use frames of size $N = 4096$ with a 75% or 3072 sample overlap. We also choose to use exactly $k = 256$ frames. Waveforms that are too short are padded with zeros, and waveforms that are too long are truncated. The audio has been sampled at $f_s = 44100$ samples/second, so ach frame represents a slice of time that is about 0.1 seconds long.

We concatenate each analysis frame, $a_i, i \in [0, k-1]$ into a single $k \times N$ matrix, called A. Each row is a frame, each column is an index in that frame

$$A = \{a_0, a_1, a_2, ..., a_{k-1}\} = \begin{bmatrix} a_0[0] & a_0[1] & a_0[2] & \dots & a_0[N-1] \\ a_1[0] & a_1[1] & a_1[2] & \dots & a_1[N-1] \\ a_2[0] & a_2[1] & a_2[2] & \dots & a_2[N-1] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{k-1}[0] & a_{k-1}[1] & a_{k-1}[2] & \dots & a_{k-1}[N-1] \end{bmatrix} \tag{30}$$

We use bracket notation, $[j]$ to indicate that each analysis frame a_i is array-like. Many programming languages could also use the following indexing conventions for matrix A:

$$A_{i,j} = A_i[j] = A[i][j] = A[i, j] \tag{31}$$

These all represent equivalent entries.

After frame-blocking, we apply a *windowing function* to each frame. A standard *Hann Window* of N samples is generated as a $1 \times N$ row-array, H. The n-th index in a Hann window with N samples is defined:

$$H[n] = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi n}{N-1}\right)\right] \tag{32}$$

This window function is applied to each analysis frame by computing the element-wise product of the Hann window array, H and each row of the analysis frames matrix, $A_{i,:}$. Result is an $k \times N$ array, $\widetilde{A}$:

$$\widetilde{A_i} = A_i \odot H \tag{33}$$

The window function allows for a cleaner transform into frequency space [17].

Finally, we perform a *Discrete Fourier Transform* (DFT) to bring each analysis frame from a time domain into a frequency domain [**?**, 13]. The Discrete Fourier Transform is applied by producing an $N \times N$ *transform matrix*, often noted at $\mathbb{W}$. Let $\omega^k = e^{\frac{-2\pi i}{N}k}$, then the DFT matrix for a time-space containing N samples

$$\mathbb{W} = \frac{1}{\sqrt{N}}\begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \dots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \dots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \dots & \omega^{(N-1)^2} \end{bmatrix} \tag{34}$$

Each column of the matrix is a complex sinusoidal oscillating with an integer number of periods within the N-sample length window [15, 13].The DFT is applied by a taking the matrix -product of $\mathbb{W}$ and $\widetilde{A}^T$. The transpose of $\widetilde{A}$ then makes each analysis frame into a column vector, which gives the appropriate dimension for multiplication.

$$\mathrm{DFT}\left[\widetilde{A}\right] = \mathbb{W}\widetilde{A}^T \tag{35}$$

Most standard implementations of neural network models require all activations, weights, and biases to be real floating-point numbers. Since the the DFT matrix introduces complex values, we compute the square of the element-wise $L2$-norm of the resultant array.

$$S_{i,j} = \left\|(\mathbb{W}\widetilde{A}^T)_{i,j}\right\|_2^2 \tag{36}$$

Where $\mathbb{W}$ is the DFT matrix from eqn. (34) and $\widetilde{A}^T$ is the transpose of the analysis frames matrix from eqn. (33). The matrix S, is the spectrogram representation of the initial waveform, and has shape $N \times k$ of real floating-point numbers. We can index matrix S similarly to that of matrix A in eqn. (31):

$$S_{i,j} = S[i][j] = S[i,j] \tag{37}$$

Each column of the S matrix is now a single-frame that has been moved into a frequency-space representation, thus there are k columns, just as there were k time-series analysis frames. Given the discretized nature of digital audio, the frequency-space representation is not a continuous function, but rather a column vector, where the frequency has been assigned to one of N bins, ranging from $-f_s/2$ to $+f_s/2$.

Standard western musical instruments seldom extend above 6 kHz [**?**, 17, 18]. This means that when constructing the spectrogram, we will rarely ever see energy present above this frequency at any time, and the S matrix will contain mostly zero, or zero-like entries. As a result of this, we can simply crop each matrix to only represent frequencies that are only between 0 Hz and 6000 Hz. This makes the input array smaller, and eliminates redundant and non-useful information. The number rows in the S matrix is reduced from N down to N'.

Each spectrogram is now $N' \times k \times 1$ (1 representing a single gray-scale pixel channel, as opposed to 3 channels for RGB inputs) and effectively encodes the energy distribution of the waveform as a function of both time and frequency. The spectrogram is the first feature used in this model. For this classifier, we have chosen $N' = 558$ and $k = 256$. For training, a batch of b samples are concatenated into a single array object. See Neural Network section for more details. For a batch of b samples of $N' \times k \times 1$ spectrograms, we shape X_1 such that:

$$X_1 = \left\{ S^{(0)}, S^{(1)}, S^{(2)}, ..., S^{(b-1)} \right\} \in \mathbb{R}^{(b \times N' \times k \times 1)} \tag{38}$$

Which is consistent with the shape of the X_1 matrix outlined in eqn(28). This matrix is presented to the *Convolution* branch of the neural network for processing. See Neural Network section for more details

4.2 Time-Space Features

The features described in this section are derived from time-domain representations of each audio sample. Time space can be represented in one of three ways:

- The raw waveform s
  Indexed by $s[i]$ with $i \in [0, 1, 2, 3, ..., M-2, M-1]$
- A single analysis frame a_i
  Indexed by $a_i[j]$ with $i \in [0, 1, 2, 3, ..., k-2, k-1]$ and $j \in [0, 1, 2, 3, ..., N-2, N-1]$
- The full analysis frame matrix A
  Indexed by $A[i][j]$ with $i \in [0, 1, 2, 3, ..., k-2, k-1]$ and $j \in [0, 1, 2, 3, ..., N-2, N-1]$

From time space, we use the following 7 features:

- Time Domain Envelope
- Zero Crossing Rate
- Temporal Center of Mass
- Auto Correlation Coefficients ($\times 4$)

4.2.1 Time Domain Envelope

The time domain envelope (TDE) is a method of determining which parts of the sound wave contains most of the energy of the signal Often, we compute the RMS-Energy of each frame - a larger energy of the frame, the larger the average amplitude of the waveform in that frame. For frames with smaller amplitude, we can assume that the signal has either not begun, or is decaying.

We adapt this feature to **compute the RMS-Energy of the full waveform**. The RMS-Energy of the full waveform, s is given by [**?**, 17]

$$\text{RMS}[s] = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1} s[i]^2} \tag{39}$$

By using the full waveform, we acquire a rough estimate for the energy in the entirety of the audio file [7]. For instruments with long sustain or release times, such as strings or undampened mallet percussion, we expect a comparably large waveform RMS when compared to those instruments without sustain such as plucked stings or dampened percussion

PLOT OF TDE FEATURES HERE

Figure 6: Time domain envelope visualized in feature-space

4.2.2 Zero Crossing Rate

The zero crossing rate (ZXR) of a signal or frame is use to measure how many times that a signal crosses it's equilibrium point. This can be done per total sound wave, per unit time, or per analysis-frame. This feature is most commonly associated with differentiating speech from music, because speech presents a more jagged and often less periodic waveform. In some cases, the ZXR can be correlated to frequency as well [5, 19].

We adapt this feature to **compute the zero crossing rate for the full waveform**. The ZXR for the full waveform s is given by [17, 7]

$$\mathrm{ZXR}\big[s\big] = \frac{1}{2}\sum_{i=1}^{M-1}\big|\,\mathrm{sign}(s[i]) - \mathrm{sign}(s[i-1])\big| \tag{40}$$

Where $\mathrm{sign}(x)$ returns $+1$ if $x > 0$, -1 if $x < 0$ and 0 if $x = 0$. This provides a rough estimate for the average frequency in the full waveform, and can help discern clean periodic signals (low ZXR) from those that may have more noise or volatile behavior (high ZXR)[17].

PLOT OF ZXR FEATURES HERE

Figure 7: Zero crossing rate visualized in feature-space

4.2.3 Center of Mass

The temporal center of mass (TCM) of a signal is used to compute roughly where in time the amplitude of the waveform *bunches up*. We treat the full waveform array, s, with M samples as a 1-Dimensional discrete mass distribution. The TCM of that waveform is then given:

$$\mathrm{TCM}\big[s\big] = \frac{\sum_{i=0}^{M-1} i s[i]}{\sum_{i=0}^{M-1} s[i]} \tag{41}$$

This includes negative amplitude values acting as "negative masses".

The TCM turns out to be a very unique feature, and very powerful in classification due to it's variance. For instruments with heavier attacks, we expect the TCM to be a very near the start of the waveform. For instruments with long sustain times, we expect a very centrally located center of mass, and for instruments with long release times, we expect a later TCM value.

PLOT OF TCM FEATURES HERE

Figure 8: Temporal center-of-mass visualized in feature-space

4.2.4 Auto Correlation Coefficients

Auto correlation coefficients (ACC) are rough estimates of the signal spectral distribution . We can compute any number of ACC's and their value changed depending on the index chosen. It is common to only compute the first K ACC's [17]. For a full waveform signal s, with M samples, the k-th ACC (indexed from 1 to K) is given by:

$$\mathrm{ACC}_k\big[s\big] = \frac{\sum_{i=0}^{M-k-1} s[i]s[i+k]}{\sqrt{\sum_{i=0}^{M-k-1} s^2[i]}\sqrt{\sum_{i=0}^{M-k-1} s^2[i+k]}} \tag{42}$$

PLOT OF ACC FEATURES HERE

Figure 9: First four auto correlation coefficients visualized in feature-space

4.3 Frequency-Space Features

The features described in this section are derived from the frequency-domain representations of each audio sample. Frequency space is represented by the transposed spectrogram matrix, S^T, described in sec. (4.1). This is done to ensure that each row is now an analysis frame, each column is a frequency bin. This gives a similar structure to the time-space analysis frames matrix, A, in eqn. (30). For each feature, we detail the physical significance and provide a visualization in feature-space.

From frequency space, we use the following 13 features:

- Mel Frequency Cepstral Coefficients ($\times 12$)
- Frequency Center of Mass

4.3.1 Mel Frequency Cepstral Coefficients

Mel Frequency Ceptstral Coefficients (MFCC's) are a very prolific feature used widly in audio classification tasks [7, 17, 19]. MFCC's are produced by converting a linear-frequency axis with units such as Hertz into units of Mels, which are design to account for the non-linear perception of frequency in humans. The Hertz to Mel and Mel to Hertz transforms are given [17]:

$$M_f[h] = 2595 \log_{10}\left(1 + \frac{h}{700}\right) \tag{43}$$

$$H_f[m] = 700\left(10^{\left(\frac{m}{2595}\right)} - 1\right) \tag{44}$$

Where M_f is the frequency in units of Mels, given $[h]$, a frequency in Hertz, and H_f is the frequency in Hertz given $[m]$ a frequency in Mels.

Mel filters are produced by grouping frequency-space into R overlapping bins called *Mel Filter-banks*. In units of Mels, each filter bank, R_i remains a constant width, but when shown in Hertz, the bins increase logarithmically.

Figure of a few Mel Filter Banks goes here

Figure 10: Mel Filter Banks shown in frequency space with units of Hertz

Each of the R filters is created to be N' samples long, and transformed back into units of Hertz. When applied to an analysis frame in the frequency spectrum, the dot-product between the filter and the spectrum gives an approximation of the energy in that filter bank. Thus, each filter is concatenated into a matrix M of shape $R \times N'$, where each row is a filter. We apply the Mel Filter banks to the spectrogram to create matrix B:

$$B = S^T M^T \tag{45}$$

Matrix B has shape $k' \times R$.

Using S^T ensure that each row of the matrix is an analysis frame in frequency space. Similarly, each column of the matrix M is a appropriately scaled Mel filter-bank. This way, the matrix product in eqn. (45) allows that $B_{i,j}$ is the dot product between the i-th analysis frame and the j-th filter-bank. Finally, we compute the average energy across all k' frames. The resulting R floating-point numbers are approximation of the signal's energy in those particular frequency bands. For this project, we have chosen to use $R = 12$ filter-banks, which are all used as features.

PLOT OF MFCC FEATURES HERE

Figure 11: Mel Frequency Ceptral Coefficients in feature-space

4.3.2 Center of Mass

The frequency center-of-mass (FCM) for an audio file provides a strong representation of how overtones and energy is distributed in the signal's frequency domain. As with the temporal center-of-mass, we treat each row of the S^T matrix as it's own 1-Dimensional mass distribution, and compute the center of mass of that row. This encodes the FCM for a single analysis frame (in frequency-space) in the waveform. For an frequency-analysis frame, $s^{(i)}$, the FCM is given by:

$$\text{TCM}_i\left[s^{(i)}\right] = \frac{\sum_{j=0}^{N'-1} j s^{(i)}[j]}{\sum_{j=0}^{N'-1} s^{(i)}[j]} \tag{46}$$

We compute the FCM for each of the k' frames, and then average the results. We use the average FCM across k' frames to compute the FCM feature:

$$TCM = \sum_{i=0}^{k'} \text{TCM}_i\left[s^{(i)}\right] \tag{47}$$

The average FCM gives a strong approximation of the instrument or signal source's range. For example, a flute or violin will have a considerably high FCM value, even in their lower registers. Similarly, basses or tubas will have considerably low FCM values. Given that the standard frequency range of some musical instruments is fixed, it is guaranteed that for any particular instrument, the FCM will consistently remain within certain bounds [**?**, 18].

PLOT OF FCM FEATURES HERE

Figure 12: Frequency center-of-mass visualized in feature-space

5 Evaluating Performance

Before making predictions on unlabeled data such as the Chaotic Synthesizers, we must determine that our model performs reasonably on data that it has never interacted with. The most common practice is to divide a full data set into a subset of *training* samples, and *testing* samples. The exact ratio of sample volume between these subsets varies depending on the task [3, 2, 11]. We as the names imply, the training subset is used to fit the model, and we use the labeled testing data set to evaluate. Since the testing data is labeled, we can compare the classifiers predictions to the *ground truth* labels.

5.1 Cross Validation

Performance evaluations are most commonly implemented in the form of *K-Fold Cross-Validation* (Also called X-val). This process involves taking a data set containing N unique samples and dividing it into K non-overlapping subsets. 1 subset is reserved to evaluate the model, and $K-1$ are used to train the model. This belongs a larger family of statisical validations called *Resampling methods* [4]. Below, we detail pseudo-code for a K-Fold Cross Validation algorithm.

Algorithm 4 A K-Fold Cross Validation program.

Require: Untrained Network or related learning algorithm, F^*
Require: A full data set of N samples. $X^{(i)}$, $i \in 0, 1, 2, ..., N-1$
Require: Number of splits in Cross validation, K
Require: Performance metric function(s), P
  Divide Data into K non-overlapping subsets x_i, each with roughly K/N samples
  $X \rightarrow \{x_0, x_1, x_2, ..., x_{K-2}, x_{K-1}\}$
  Performance History $\leftarrow \{\}$
  for $j = 0, 1, 2, 3, ..., K-2, K-1$ **do**
    Reset all parameters in F^* to a random "untrained" state
    Set aside testing data subset
    $X_{test} = x_j$
    Concatenate the remaining subsets into training data set
    $X_{train} = x_{j \neq k}$
    Train the model, F^* with the X_{train} data set
    Evaluate the trained model with metric function(s) P
    Store Performance P in Performance History array
  end for
  Compare performance results, and adjust model, parameters as needed, and repeat if desired.

By using cross-validation, we gain a deeper understanding of the model to ensure that the model can be properly trained consistently. This also helps counteract the possibility that the model got very lucky or unlucky with a particular set of initial conditions [1, 4].

5.2 Performance Metrics

In the case of the multi-category classifier, it is also important that we choose the appropriate metrics to evaluate our performance. While the neural network itself uses the cost function as it's sole performance metric to optimize, we also require a set of more human-readable functions. We introduce a set of functions and metrics than enable a more tangible interpretation of model performance. To evaluate a performance metric, we require a set of samples with ground truth labels, y, and a model's prediction for those labels, y^*.

5.2.1 Confusion Matrix

The confusion matrix (also called a confusion table) is a very quick, often graphical model that can be used to show how a model performs over a subset of predictions. The general idea of this object is count the number of times class i is predicted to be in class j, and vice-versa [1]. If we see that these classes are being repeatedly *confused* in the model's prediction process, then we must modify the model or features to account for it.

For a k-classes classifier, a confusion matrix will have shape $k \times k$, and every element is a non-negative integer. Each row represents the "ground truth" or labeled class, and each column represents a predicted class. Thus for a confusion matrix, C, we can say that:

$$C_{i,j} = \text{Number of samples that}$$
$$\text{belong to class } i, \text{ and were predicted to be in class } j$$

Thus, indexes where $i = j$ represents a correct prediction, and $i \neq k$ indicates an incorrect prediction. A confusion matrix with relatively large values in main diagonal indicates a model that consistently predicts correct labels [1].

Images of Confusion Matrices Go Here! Images of Confusion Matrices Go Here!
Images of Confusion Matrices Go Here! Images of Confusion Matrices Go Here!

Figure 13: Example Confusion Matrices for k-classes tasks

We can also use a confusion matrix to define four useful quantities relating a prediction to it's label. Suppose a belongs to class q, with the prediction given by, y^* and a label is given by, y:

1. **True-Positive**
   A sample is TP if $y^* = q$ and $y = q$
2. **True-Negative**
   A sample is TN if $y^* \neq q$ and $y \neq q$
3. **False-Positive**
   A sample is FP if $y^* = q$ and $y \neq q$

4. **False-Negative**
   A sample is FN if $y^* \neq q$ and $y = q$

Check this!

5.2.2 Precision Score

Precision score (also called *specificity*) offers a more concise performance metric than a confusion metric. For a classifier with $k = 2$ unique classes, we define the precision score of a model as:

$$\text{Prec} = \frac{TP}{TP + FP} \tag{48}$$

Where TP is the number of *true-positive*, and FP is the number of false-positives predictions. For a k-classes confusion matrix, C, the precision score is given by the entry $C_{j,j}$ divided by the sum of column j:

$$\text{Prec}_j = \frac{C_{j,j}}{\sum_{i=0}^{k-1} C_{i,j}} \tag{49}$$

For multi-class problems, the precision score is usually used as an average over all classes.

Physical Significance of Precision

5.2.3 Recall Score

Recall score (also called *sensitivity*) also offers a more concise performance metric than a confusion metric. For a classifier with $k = 2$ unique classes, we define the recall score of a model as:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{50}$$

Where TP is the number of *true-positive*, and FN is the number of false-negative predictions. For a k-classes confusion matrix, C, the precision score is given by the entry $C_{j,j}$ divided by the sum of row j:

$$\text{Recall}j = \frac{C_{j,j}}{\sum_{i=0}^{k-1} C_{j,i}} \tag{51}$$

For multi-class problems, the recall score is also used as an average over all classes.

Physical Significance of Recall

5.3 Tracking Metrics over a Period of Training

6 Experimental Results

7 Conclusion

8 Acknowledgments

The completion of this project would not have been possible without the various contributions from the following people:

- Dakota Buell, John Parker, Nathan Richard, Morgan Saidel for consultation on physics, mathematics, and programming topics.
- Madeline Edwards for differing me to this project, as well as constant support and consultation on mathematical topics.
- Dr. Kevin Short and Dr. Maurik Holtrop for various consultation and guidance throughout this project
- University of Iowa, Electronic Music Studio and Philharmonica Symphony Orchestra for the digital sound library used for training data samples.

References

[1] Geron, Aurelien. *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly, 2017.

[2] Geron, Aurelien. *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 2nd ed., O'Reilly, 2019.

[3] Goodfellow, Ian, et al.*Deep Learning*. MIT Press, 2017.

[4] James, Gareth, et al. *An Introduction to Statistical Learning with Applications in R*. Springer, 2017.

[5] Khan, M. Kashif Saeed, and Wasfi G. Al-Khatib. "Machine-Learning Based Classification of Speech and Music." Multimedia Systems, vol. 12, no. 1, 2006, pp. 55–67., doi:10.1007/s00530-006-0034-0.

[6] Levine, Daniel S. *Introduction to Neural and Cognitive Modeling*. 3rd ed., Routledge, 2019.

[7] Liu, Zhu, et al. "Audio Feature Extraction and Analysis for Scene Segmentation and Classification." Journal of VLSI Signal Processing, vol. 20, 1998, pp. 61–79.

[8] Loy, James , *Neural Network Projects with Python*. Packt Publishing, 2019

[9] McCulloch, Warren S., and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, 1943, pp. 115–133.

[10] Mierswa, Ingo, and Katharina Morik. "Automatic Feature Extraction for Classifying Audio Data." *Machine Learning*, vol. 58, no. 2-3, 2005, pp. 127–149., doi:10.1007/s10994-005-5824-7.

[11] Mitchell, Tom Michael. Machine Learning. 1st ed., McGraw-Hill, 1997.

[12] Olson, Harry E. *Music, Physics and Engineering*. 2nd ed., Dover Publications, 1967.

[13] Peatross, Justin, and Michael Ware. *Physics of Light and Optics*. Brigham Young University, Department of Physics, 2015.

[14] Petrik, Marek. "Introduction to Deep Learning." Machine Learning. 20 April. 2020, Durham, New Hampshire.

[15] Short, K. and Garcia R.A. 2006. "Signal Analysis Using the Complex Spectral Phase Evolution (CSPE) Method." AES: *Audio Engineering Society Convention Paper*.

[16] TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[17] Virtanen, Tuomas, et al. *Computational Analysis of Sound Scenes and Events.* Springer, 2018.

[18] White, Harvey Elliott, and Donald H. White. *Physics and Music: the Science of Musical Sound*. Dover Publications, Inc., 2019.

[19] Zhang, Tong, and C.-C. Jay Kuo. "Content-Based Classification and Retrieval of Audio." *Advanced Signal Processing Algorithms, Architectures, and Implementations VIII*, 2 Oct. 1998, pp. 432–443., doi:10.1117/12.325703.