Mapping Chaotically Generated Synthesizers to Musical Instruments with Machine Learning and Harmonic Analysis

Multimodal Classification of Digital Signal

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**
1.1 Introduction . . . 2
1.2 Methodology . . . 3

2 The Neural Network **5**
2.1 An Introduction to Neural Networks . . . 5
2.2 The Structure of a Neural Network . . . 6
2.3 Layers Used in this Classification Model . . . 7
2.4 Activation Functions Used in Network Layers . . . 14
2.5 Training The Model . . . 15
2.6 Multimodal Architecture . . . 23

3 Properties of Musical Instruments **28**
3.1 Idiophones . . . 29
3.2 Membranophones . . . 29
3.3 Chordophones . . . 29
3.4 Aerophones . . . 30
3.5 Other Generated Sounds . . . 30

4 Feature Selections **32**
4.1 Feature Space . . . 32
4.2 Spectrogram Features . . . 38
4.3 Time-Space Features . . . 42
4.4 Frequency-Space Features . . . 50

5 Evaluating Model Performance **55**
5.1 K-Folds Cross Validation . . . 55
5.2 Performance Metrics . . . 56
5.3 Tracking Metrics over a Period of Training . . . 60

6 Experimental Results **63**
6.1 Executing Cross Validation . . . 63
6.2 Comparing Results between Architectures . . . 64
6.3 Comparing Classification Scores within Each Class . . . 66
6.4 Discussion of Results . . . 72
6.5 Predictions on Chaotic Synthesizers . . . 73

7 Conclusion **74**

8 Acknowledgments **75**

Abstract

The task of recognizing the source of a sound wave is trivial and almost effortless to a human being.

1 Introduction

1.1 Introduction

The ability to classify time-series data into unique categories has become topic of much attention in the last few decades [8, 14]. From climate measurements, to financial reports, to audio-visual media, or to medicine, time-domain signals surround the modern world at every corner [11, 28]. In each context, the general idea is to place the data or a subset of the data into distinct categories called *classes* based on it's properties. For example, doctors may wish to classify a patient as having a heart condition based on properties of their heartbeat or banks may wish to detect fraudulent transactions based on a history of spending habits. Often, the volume of information or complexity of the task makes it unreasonable for a human to do unaided by some form of automation.

It has long been a goal of computer scientists and engineers alike to develop a program, algorithm, or machine that could perform a certain task to the level of human-being [1, 2, 12]. While humans are intricate organisms capable of recognition or classification in many contexts, we are not notorious for the ability to process millions of raw numerical values in the course of a few seconds. Similarly, modern computers are also complex systems that can filter through exabytes of information on a daily basis, but are not generally known for being able to distinguish cats from dogs [2]. The compromise between is an algorithm called a *neural network* that combines the computation speed of a computer with the decision-making power of a human [4, 9, 12].

In this work, we have developed an automated neural network program that can classify digital sound waves based on the musical instrument that produced them. The general use of machine learning algorithms for audio information classification or identification is well explored, and a topic of much modern research [8, 11, 14, 22, 28].

1.2 Methodology

This project includes a great number of steps to ensure it's completion. Here we show a high-level outline of the pieces involved.

1.2.1 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 and approximate it with the function F^*. This can map the contents of a sound file to a potential source as to mimic F. For a set of inputs (called features or predictors) $\vec{x} = \left\{x_0, x_1, x_2, ..., x_{p-1}\right\}$ and a set of classes 0 through $k-1$, we denote this function as:

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

In secs. (2.1) and (2.2), we introduce the nature and structure of a neural network and how it can use used in part to solve this problem. We discuss it's inspiration from a biological brain, and show how different layers of the neural network compare to functions or processes in the brain.

1.2.2 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 files contain short segments of musical instruments performing a single note, and is labeled according to the instrument that produced the sound. The files and labels make up the core of the data set that we use to train the model.

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. This ensures that all data will have a consistent format when features are extracted. This stage is detailed in sec. (4.1.3).

1.2.3 Designing Classification Features

The performance of a neural network is largely dependent of designing an appropriate set of classification features which represent the input given to the neural network. 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.

1.2.4 Designing A Complementary Network Architecture

With the appropriate set of features designed, and the number of outputs determined, we can organize the structure of the neural network function. The shape of the network defines the hypothesis space, and a subset of the possible solutions that the trained model could arrive at. We construct a multimodal neural network, explored in sec. (2.6), that processes two distinct entry points 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 outputted prediction.

1.2.5 Testing and Evaluating Network Performance

We divide the raw data set up into a training and testing sets. We employ a resampling method called cross-validation and compute the results of various performance metrics to ensure that our model is making reliable predictions, and can 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. We explore the explore this is sec. (5).

1.2.6 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. We explore thisas part of our conclusions in sec. (7).

2 The Neural Network

2.1 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 recognition, 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 solution that allows for a computer to *learn* from labeled training examples in order *teach* itself [1, 15]. One such an algorithm that has been used for this type of task is a *Neural Network* [3, 4, 9].

As the name implies, early neural networks were inspired from from the human brain. Former YouTube video classification team lead, and current machine learning consultant, Aurelien Geron writes about the relationship between biological brains and mathematical neural networks [2]:

> 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 a brain in that the program can *learn* in a sense. Where a biological brain uses connections of chemical and electrical exchanges to process input and make decisions, the computational counterpart uses arithmetic and numerical exchanges to similarly process input and make decisions [2, 9, 1].

When a listener hears a sound, millions of neurons in their brain exchanges those chemical and electrical signals in such a way as to be able to identify the source of that sound [2, 26]. More conceptually, we can say that a listener is presented with a set of information x, which is air molecules vibrating near their tympanic membrane [27, 17]. That information is then passed into the brain where some function F is used to model the incredibly complex set of neural connections. The output of that function is some label y, which the listener associates with a particular source [17]. Similarly, we construct a neural network to accept a set of predictors x, which is passed into a function F^* that is use to approximate F, and produce an output y^*, which predicts the source of the audio sample.

A neural network can then be considered as mathematical function that seeks to produce an approximation F^*, of some the usually unknown function F. The function F^* uses a set of parameters Θ, to map a series of inputs, x (called *features*) to a series of outputs, y (called *predictions*) [4, 7, 26]. Over the course of their development, humans will be able to learn to match sounds to sources with relative ease, given the appropriate labels. In a similar manner, out neural network is present a collection of sound files and predictors and a complementary set of labels indicating the source of each sound.

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 or sound source that could have produced the sound-wave in the audio file [4, 12, 26]. In this section we detail the nature of a general neural network.

2.2 The Structure of a Neural Network

A Neural Network is a model of a mathematical function, composed of several smaller mathematical functions called *layers* [4, 12]. 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. The output of one layer is passed forward and used as the input for the next layer. 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 [2, 12]. The output of the final layer encodes the "decision" of the model given a particular input.

Other than inspiration from the brain, Neural Networks are named *networks* due to their successive chained or nested composition nature. A standard representation of a neural network is often a linked list or computational graph structure. This maps out exactly how the repeated compositions are structured and organizes the flow of information in more complicated networks. Each node in the computational graph represents a *layer*, which executes a particular prescribed mathematical function [4]. In the case of a *linear, feed-forward* network, information passes in a single direction, to produce an output. We can represent this as a graph:

$$F^{*}(x) = x \rightarrow f^{(0)} \rightarrow f^{(1)} \rightarrow ..... \rightarrow f^{(L-2)} \rightarrow f^{(L-1)} \rightarrow y \tag{2}$$

or as a nested function:

$$F(x) = f^{(L-1)}\big[f^{(L-2)}\big[...f^{(1)}\big[f^{(0)}\big[x\big]\big]\big]...\big]\big] \tag{3}$$

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* [2, 12]. A network model that contains L unique layers is said to be an L-Layer Neural Network, with each layer 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* [2, 12].

Due to the chained nature of the model, the activations from some layer, $(l-1)$, are used to directly produce the activations for the next successive layer (l). The function that represents a layer (l) is given by:

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

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 [2, 7, 12].

Below we outline a standard "forward pass" algorithm for a general feed-forward deep neural network. We presume each layer to be a node in a computational graph that contains a method "*call*" which executes the layer's main function as in Eq. (4). Additionally, each node contains a pointer "*next*" which gives the next layer in the network chain. Each layer should also store it's linear activations, $z^{(l)}$ and it's output activations $x^{(l)}$ for training purposes. This algorithm is adapted from Goodfellow, [4].

Algorithm 1 Forward propagation system in a standard deep neural network. Each layer is presumed to be a node in a linked computational graph. This example has been setup to assume one input layer, and one output layer. Practical implementations should include mini-batches of data as opposed to a single sample.

Require: Set of Layer functions $f^{(i)}$, $i \in [0, 1, 2, 3, ..., L-1]$.
Require: Input sample - $x^{(0)}$
  Set the current layer to be the input layer:
  $f^{(i)} \leftarrow f^{(0)}$
  for Layer $i \in [1, 2, 3, ..., L-1]$ **do**
    Run "Call" Method with current activations, transform activations:
    $x^{(i)} = f^{(i)}.\text{call}(x^{(i-1)})$
    Move to the next layer in the network:
    $f^{(i)} \leftarrow f^{(i)}.\text{next}()$
  end for
  Return the network's output, which are the output activation of the final layer
  return $x^{(L-1)}$

2.3 Layers Used in this Classification Model

As stated previously, a neural network is composed of a series of functions that are called in succession to transform features (an input) into predictions (an output). As shown in Eq. (4), each function feeds directly into the next as to form a chain-like computational graph [4]. Many layer functions 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 [2, 12]. The Linear transformation can be given by Eq.(5) in the case of a 1D Dense layer (2.3.1) or Eq. (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*:

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

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 introduces the ability to account for non-linearity in the decision boundaries. The combination of Eq. (5) and Eq. (7), for a layer l , make up the function represented by that layer, as in Eq. (4). In this, we discuss and describe the types of layer functions that are used to produce the classification model in this project.

2.3.1 Dense Layer

The Linear Dense Layer, often just called a *Dense* Layer, or *Fully-Connected* layer, was one of the earliest and most common function types used in artificial neural network models [3, 12, 13]. A dense layer is composed of a layer of *artificial neurons*, each of which holds a numerical value within it, (usually a double-precision floating point number) called the *activation* of that neuron. This idea was developed from McCulloch and Pitts' work [13] in attempting to model cognitive processes as mathematical functions, and was expanded upon by Frank Rosenblatt in 1957 by combining many dense layers together to form a multilayer perceptron (MLP) [2, 12, 9].

We model a layer of neurons as a vector of floating-point numbers. By convention, the array is an n-dimensional column vector, where n is number of nodes in that layer. 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, Eq.(5) and the transformation in Eq.(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 $n \times 1$. We can denote the function in a similar manner to Eq.(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 [2, 12]:

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

or more compactly:

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

Eq. (11) is generally referred to as the *dense layer feed-forward equation* [4]. Below, we detail pseudo-code for what a "Call" method for a dense layer node in a computational graph may look like [2, 25] Compare this process with Eq. (11).

Algorithm 2 Typical "Call" method for a dense layer in a neural network that contains n neurons/nodes. This example shows the computation over a single input $x^{(l-1)}$ but a practical implementation should include mini-batches of samples.

Require: $x^{(l-1)} \leftarrow$ Input activations shaped into $(m \times 1)$ column vector
Require: $n \leftarrow$ Number of nodes in this dense layer
Require: $\sigma^{(l)} \leftarrow$ Activation function for this layer
Require: $W \leftarrow$ a $(n \times m)$ matrix of weights
Require: $b \leftarrow$ a $(n \times 1)$ vector of bias values
Evaluate linear activation values by computing the standard matrix-vector product of $W^{(l)}$ and $x^{(l-1)}$, and adding the bias neuron value:
$z[i] \leftarrow b[i] + \sum_{j=0}^{n-1} W[i,j]x^{(l-1)}[j]$
Evaluate output activation values by applying the activation function to each element in the linear activation column vector z:
$x^{(l)}[i] \leftarrow \sigma^{(l)}\big(z[i]\big)$
Store both activation arrays for back-propagation. Return the output activations:
return $x^{(l)}$

2.3.2 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. [2, 12]. The layer function get's it's name from it's implementation of the mathematical *convolution* operation. The 2D discrete-convolution of function or 2D array $A[i,j]$ and $B[i,j]$ is given by [4]:

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

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* [12, 4].

In the case of a 2D input array, $x^{(l-1)} \in \mathbb{R}^{(M \times N)}$, 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 activations 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 result in detecting vertical lines, horizontal lines, sharp edges, areas of mostly one color, etc. [2, 12].

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

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

As an example, consider a single filter map over input $x \in \mathbb{R}^{(3 \times 4)}$ and filter map $W \in \mathbb{R}^{(2 \times 2)}$ as shown in Fig. (1):

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

(a) Input Activations, $x^{(l-1)}$

w	x
y	z

(b) Convolution Filter, $W_k^{(l)}$

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, $x_k^{(l)}$

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 [4]

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 [4]

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

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 [2, 4, 12]. Below we detail what the "Call" method for a 2-dimensional convolutional neural network may look like. Compare this procedure with Eq. (14).

Algorithm 3 Typical "Call" method for a 2-Dimensional Convolutional layer in a neural network that uses K filters, of $m \times n$ kernels, with an assumed stride size of 1×1. This example shows the computation over a single input $x^{(l-1)}$ but a practical implementation should include mini-batches of samples.

Require: $x^{(l-1)} \leftarrow$ Input activations shaped into $(M \times N)$ matrix
Require: $K \leftarrow$ The number of filters (feature maps) to be used.
Require: $\sigma \leftarrow$ Activation function for this layer
Require: $W \leftarrow$ array of weight kernels containing $W_k, k \in [0, 1, ..., K-1]$ each shaped $m \times n$
Require: $b \leftarrow$ a $(1 \times K)$ vector of bias values, $b_k, k \in [0, ..., k-1]$
  for Each feature map, $k \in [0, 1, 2, ..., k-2, k-1]$ **do**
    Evaluate linear activation values by computing the 2D convolution-product of W_k and $x_k^{(l-1)}$:
    $z_k[i,j] \leftarrow b_k + \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} W_k[u,v]x_k^{(l-1)}[u-i, v-j]$
    Evaluate output activation values by applying the activation function to each element in the linear activation matrix z:
    $x_k^{(l)}[i,j] \leftarrow \sigma\big[z_k[i,j]\big]$
  end for
  Store both activation arrays for back-propagation. Return the output activations from each of the K filters:
  return $x^{(l)}$

2.3.3 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 [12, 4].As an example, consider again figure (1a). 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. (2):

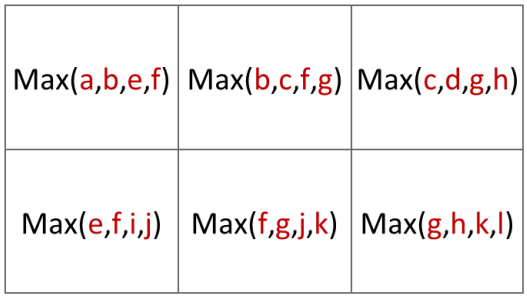


Figure 2: The result of 2D maximum-pooling an input array. This image was adapted from Loy, pg. 126 [12]

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 [2, 12, 4]. This layer "non-trainable" because it does no use any weights or transformations, only a fixed procedure. Below, we detail what the "Call" method for a 2D maximum pooling layer might look like.

Algorithm 4 Typical "Call" method for a 2-Dimensional Maximum Pooling layer in a neural network. We assume 2D input, but a practical implementation may include the need to loop over a high dimensional structure. For simplicity, we assume a stride size of 1×1. This example shows the computation over a single input $x^{(l-1)}$ but a practical implementation should include mini-batches of samples.

Require: $x^{(l-1)} \leftarrow$ Input activations shaped into $(M \times N)$ matrix
Require: $m \times n \leftarrow$ Pool height and width
Require: $P \leftarrow$ A temporary pool array of shape $m \times n$ to hold local activationbs
  for $i \in [0, 1, ..., M-2, M-1$ **do**
    for $j \in [0, 1, ..., N-2, N-1$ **do**
      Collect local activations in P, zero pad if necessary
      $P = x^{(l-1)}[i : i+m, j : j+n]$
      Find maximum valie in P, and add to output
      $x^{(l)}[i, j] = \max\left(P\right)$
    end for
  end for
  Store the activation array for back-propagation, and return it
  return $x^{(l)}$

2.3.4 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 rearranged down into a single dimensional array. Note that this is not like projecting the data into a lower dimension, but rather is simple the reorganization of values into an array of a single axis. 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 Eq.(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. This layer also is "non-trainable" because it does no use any weights or transformations, only a fixed procedure.

2.3.5 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:

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

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

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

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 activations from two different layers into a single new layer. In this case of the model used in the project, we use it to combine the outputs that result from the convolution branch and the perceptron branch tp produce an single aggregated output. This process is detailed further in sec. (2.6). This layer also "non-trainable" because it does no use any weights or transformations, only a fixed procedure.

2.4 Activation Functions Used in Network Layers

Activation functions are a key parameter in the behavior of neural networks. As discussed in sec. (2.5) a layer function, $f^{(l)}$ is generally composed of a linear transformation, as in Eq. (5), and then an element-wise activation function as in Eq. (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 [2, 12]. In this section, we detail the activation functions used in this classification neural network.

2.4.1 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-negative, the value is untouched, otherwise a 0 is returned. For an input activation array 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. (3).

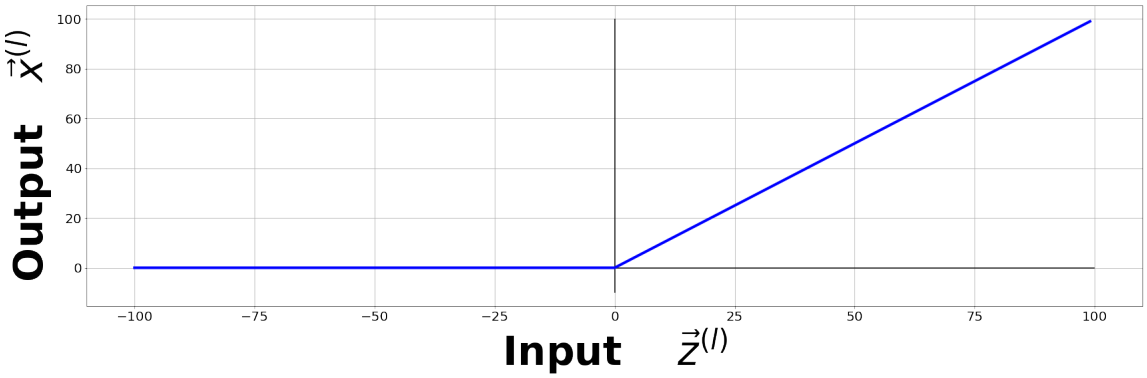


Figure 3: Rectifed Linear Unit (ReLU) activation function

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.

2.4.2 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 [2, 4, 26]. The i-th element in a softmax activation function is given by:

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

This function is almost only used in the output layer of a neural network, and encodes a normalized discrete categorical probability [12, 26]. For example, if Eq.(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.

2.5 Training The Model

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 [4]. To better understand this, we turn to Tom Mitchell's explanation of a learning algorithm [15]:

> 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 direct human intervention, a model must update itself to improve it's performance at a give task as 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 the set of values within each trainable layer's weighting matrix and bias vector such that:

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

The arrays $W^{(l)}$and bias vectors $b^{(l)}$ are not trainable parameters themselves, but the the floating-point entries within them are the values that can be adjusted. Each element in this set may contain hundreds or thousands of parameters, so we represent these indirectly as parameters within their respective layers. We choose this notation for Θ for notation simplicity. Note that not all layers have trainable parameters, and activations functions themselves are not trainable, but fixed functions. Modern neural networks can contain 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 [2, 4, 9]. For the network that we have designed in this project, there are roughly $33,000$ parameters across 20 layers. In this sections we motivate and explore how the we can update the parameters in Θ as to train the neural network.

2.5.1 The Cost Function

Suppose we pass a training sample into the neural network. This sample is represented by the feature-vector $x^{(0)}$, with an expected outcome given by the one-hot-encoded vector y. After the network finishes processing, it's output activations, given by the vector $x^{(L-1)}$, also noted as y^* represents the prediction for the class label. 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 reasonably 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 model [4, 7]. 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 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 [7]. For this k-classes classification task, we choose to use the *Categorical Crossentropy* (CXE) cost function. For a sample labeled by the one-hot-encoded vector y and corresponding prediction vector y^*, the CXE cost for a single sample is given by [4, 26, 1]:

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

Thus, the average loss over a mini-batch of b samples is given:

$$\big\langle \mathrm{CXE}\big[y, y^*\big] \big\rangle = -\frac{1}{b}\sum_{n=0}^{b-1}\sum_{i=0}^{k-1} y_i^{(n)} \ln(y_i^{*(n)}) \tag{23}$$

Suppose that a given sample belongs to class j in a k-classes classifier. Since the label vector, y is one-hot-encoded, all entries are zero except $y_j = 1$.

$$y = \big[y_0, y_1, ..., y_j, ..., y_{k-1}\big]^T = \big[0, 0, ..., 1, ..., 0\big]^T \tag{24}$$

Thus the sum in Eq. (22) contains mostly zero terms, with the only exception at index j, where we have $y_j \ln(y_j^*) = \ln(y_j^*)$. Since the vector y^* has been subject to the softmax activation function, we must have $y_j^* \in [0, 1]$. This means that taking the natural log of this value returns a negative number. Multiplying that by negative 1 returns a high loss when the activation of y_j^* is low and a low cost when $y_j^* \approx 1$. We visualize this relationship for a single sample in Fig. (4):

By optimizing the parameters Θ in the network model, we allow for the output of a consistently low cost function to be produced across all samples in the data set. When a model produces consistently low cost values across new unseen samples, we consider this to be a *trained* or *fitted* model [4, 12, 15].

2.5.2 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* [7, 12]. 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 [4].

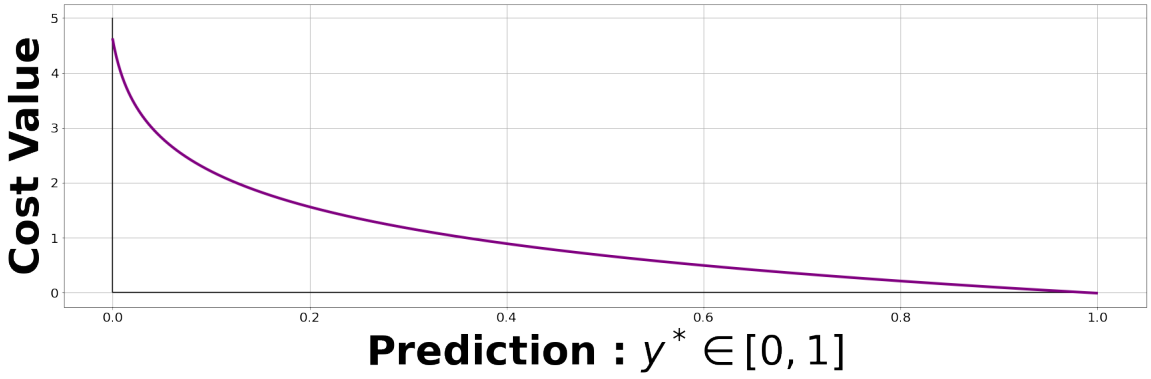


Figure 4: Plot of how y_j^* affects the output values of the CXE cost function

A neural network uses *indirect optimization*, which contrasts from pure optimization. Deep Learning expert Ian Goodfellow describes the difference [4]:

> 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 improve 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 minimizes classification error and maximizes 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 Eq. (4), This recursive nature combined with the dimensionality of the parameter object Θ makes an analytical solution to the optimization impractical [2, 4, 7]. We instead optimize the cost function with a numerical *iterative* method [12].

We can reduce the cost given a set of parameters, $J(\Theta)$, by repeatedly moving each element in Θ with small steps in the direction of the negative partial derivative with respect to 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:

$$\nabla_{\Theta}\big[J\big] = \Big[\frac{\partial J}{\partial W^{(0)}}, \frac{\partial J}{\partial b^{(0)}}, \frac{\partial J}{\partial W^{(1)}}, \frac{\partial J}{\partial b^{(1)}}, \ldots, \frac{\partial J}{\partial W^{(L-1)}}, \frac{\partial J}{\partial b^{(L-1)}}\Big]^T \tag{25}$$

Where $\frac{\partial J}{\partial W^{(l)}}$ is taking the partial derivative of each element $W_{i,j}$ in the $W^{(l)}$ matrix, and preserves the shape. While there may only be 20 or so layers, the gradient vector actually has one element for every trainable parameter in all of the layers. This means that for our network, the gradient vector contains upwards of $33,000$ elements. We again choose to group these elements by their parent structure for ease of notation.

Due to the nested composition of the network output in Eq. (3), and subsequently the cost itself, we must use the chain rule of calculus to work backwards through computational graph of the neural network to compute the partial derivative of J with respect to each parameter in Θ. The process of working backwards to compute each element in the gradient vector is called *back-propagation* [2, 4, 12].

2.5.3 Back-Propagation

Recall our model of a neural network as a computational graph as in Eq. (2). Each node of the graph is a layer that represents a mathematical function which takes the activation from the previous layer $x^{(l-1)}$ and transforms them into the activations of the current layer, as in Eq. (4). Additionally, we recall that each layer with trainable parameters is made up two steps which produce linear activations such as in Eq. (5) and non-linear activations, as in Eq. (7). Therefore, to understand back propagation, it is helpful to examine the flow of information within each layer. Consider a simplified neural network model visualized in Fig. (5)

Note that we indicate the cost as $J(\Theta)$. This is an allusion to the fact that J depends of both the labels y, and the predictions y^*. Where y^* is a function of all of the previous layer activation, which are in turn functions of each weight bias element, i.e. the contents of the Θ object, in Eq. (21). This is why we are able to compute the gradient with respect to Θ as in Eq. (25).

Raw information is presented as $x^{(0)}$ at the top of the figure, and and output is returned as $x^{(L-1)}$ at the bottom. Arrows represent the flow of information. As expressed in Eq. (22), we compute the cost of a sample, or batch of samples by comparing the difference between y^* (which is $x^{(L-1)}$) and y. Given this graph chain and it's numerical nature, we cannot compute each element in the gradient vector through conventional symbolic calculus [12]. We must instead use the chain rule for partial derivatives to begin at the final output of the network and successively work our way forward to the entry layer to compute the numerical derivative with respect to each parameter. This process is called *back-propagation* [2, 4].

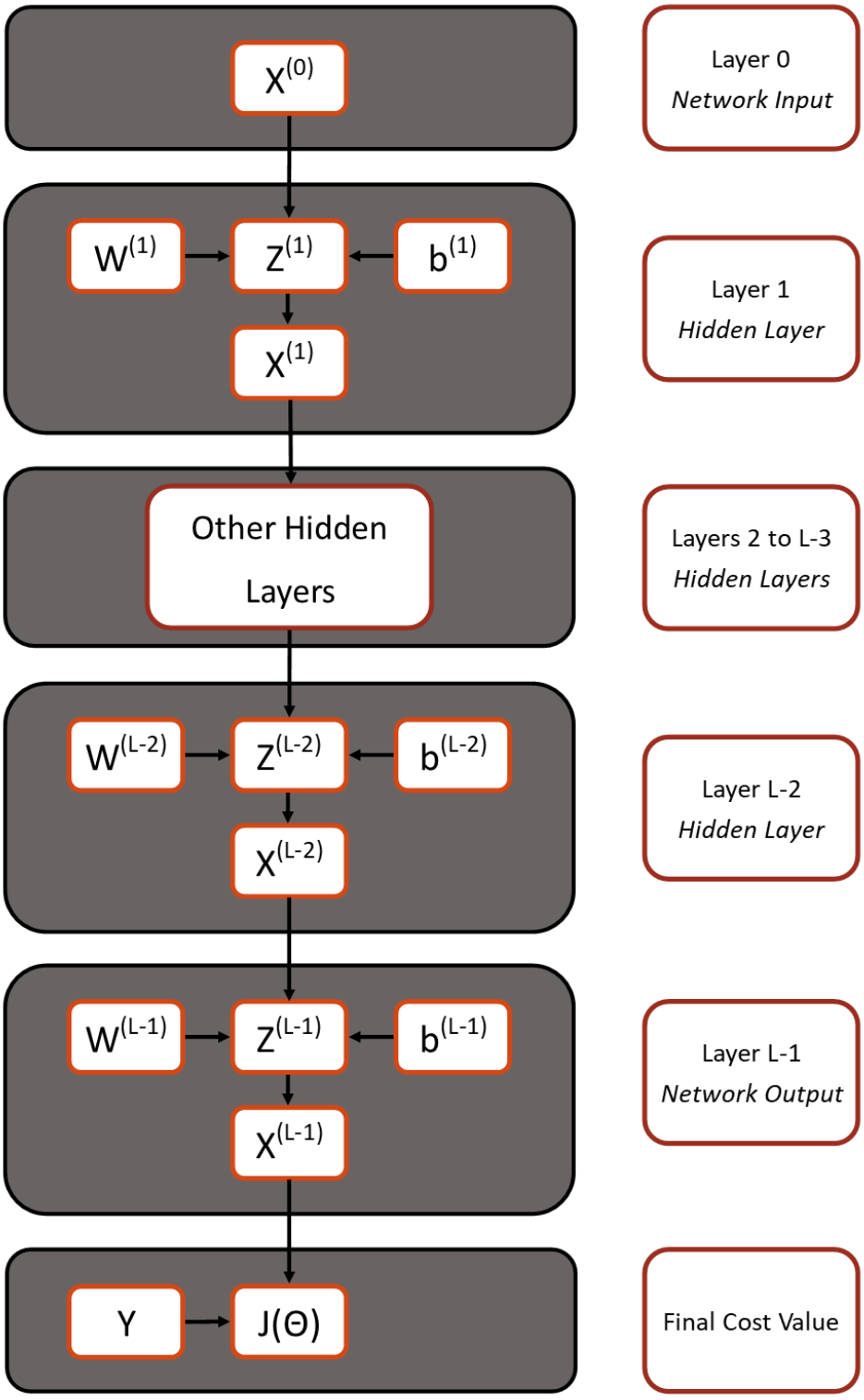


Figure 5: A visual representation of a simplified neural network as a computational graph

Suppose we wish to compute the elements of $\frac{\partial J}{\partial W^{(L-1)}}$ and $\frac{\partial J}{\partial b^{(L-1)}}$ for the gradient vector. The flow of information in Fig. (5) indicates that we cannot directly evaluate the derivative because the cost J, is a function $x^{(L-1)}$, which is a function of $z^{(L-1)}$, which is a function of elements in $W^{(L-1)}$. Thus, we have:

$$\frac{\partial J}{\partial W^{(L-1)}} = \frac{\partial}{\partial W^{(L-1)}}\left[J\left\{x^{(L-1)}\left[z^{(L-1)}\left\{W^{(L-1)}\right\}\right]\right\}\right] \tag{26}$$

The evaluation of the derivative of a composite function requires use of the chain rule of multivariate calculus. We represent this symbolically as:

$$\frac{\partial J}{\partial W^{(L-1)}} = \frac{\partial J}{\partial x^{(L-1)}}\frac{\partial x^{(L-1)}}{\partial z^{(L-1)}}\frac{\partial z^{(L-1)}}{\partial W^{(L-1)}} \tag{27}$$

Similarly, the partial derivative of the cost with respect to the bias array can be formulated as:

$$\frac{\partial J}{\partial b^{(L-1)}} = \frac{\partial J}{\partial x^{(L-1)}}\frac{\partial x^{(L-1)}}{\partial z^{(L-1)}}\frac{\partial z^{(L-1)}}{\partial b^{(L-1)}} \tag{28}$$

For the categorical cross entropy cost function, we can evaluate $\partial J/\partial x^{(L-1)}$ over a single sample that belongs to class i as:

$$\frac{\partial J}{\partial x^{(L-1)}} = \frac{\partial J}{\partial y^*} = \frac{\partial}{\partial y^*}\Big[-y\ln(y^*)\Big] = -\frac{y}{y_i^* + \delta} = -\frac{1}{y_i^* + \delta} \tag{29}$$

We often introduce $\delta \approx 10^{-8}$ to prevent possible division by 0 errors. The value $\partial x^{(L-1)}/\partial z^{(L-1)}$ can be found by differentiating Eq. (7) with respect to the input activations.

$$\frac{\partial x^{(L-1)}}{\partial z^{(L-1)}} = \sigma^{'(l)}\big[z^{(L-1)}\big] \tag{30}$$

For a dense layer as in sec. (2.3.1) or a 2D-convolution layer as in sec. (2.3.2) $\partial z^{(L-1)}/\partial W^{(L-1)}$ is given by the derivative of Eq. (5) or Eq. (6) with respect to $W^{(l)}$:

$$\frac{\partial z^{(L-1)}}{\partial W^{(L-1)}} = \frac{\partial}{W^{(L-1)}}\Big[W^{(L-1)}x^{(L-2)} + b^{(L-1)}\Big] = x^{(L-2)} \tag{31}$$

$$\frac{\partial z^{(L-1)}}{\partial W^{(L-1)}} = \frac{\partial}{W^{(L-1)}}\Big[\big(W^{(L-1)} * x^{(L-2)}\big) + b^{(L-1)}\Big] = x^{(L-2)} \tag{32}$$

Finally, differentiating with respect to the bias array: $\partial z^{(L-1)}/\partial b^{(L-1)}$, we have:

$$\frac{\partial z^{(L-1)}}{\partial b^{(L-1)}} = \frac{\partial}{W^{(L-1)}}\Big[W^{(L-1)}x^{(L-2)} + b^{(L-1)}\Big] = 1 \tag{33}$$

Suppose we then wish to compute the elements of $\frac{\partial J}{\partial W^{(l)}}$ and $\frac{\partial J}{\partial b^{(l)}}$. Using the same graph from Fig. (5), we can follow the chain of information to derive an expression for the similar to Eq. (27) and Eq. (28) for each layer (l) in the nueral network. In general, we can state that:

$$\nabla_{W^{(l)}} J = \frac{\partial J}{\partial W^{(l)}} = \frac{\partial J}{\partial x^{(l)}} \odot \partial\sigma^{(l)}[z^{(l)}] \cdot x^{(l-1)} \tag{34}$$

And

$$\nabla_{b^{(l)}} J = \frac{\partial J}{\partial b^{(l)}} = \frac{\partial J}{\partial x^{(l)}} \odot \partial\sigma^{(l)}[z^{(l)}] \tag{35}$$

Where $z^{(l)}$ are the linear activations defined in Eq. (5) and Eq. (6). Note that each $z^{(l)}$ and $x^{(l)}$ are typically stored in memory during the forward pass in Alg. (1) to prevent the need to recompute them [3, 4].

Algorithm 5 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 in a given layer (l). Each element in those arrays dW and db is the discrete gradient of the cost due to that parameter. A practical application of this algorithm should include batches of samples instead of a single sample and a regularizing function at each step.

Require: Cost/Objective function J.
Require: Set of Layer functions $f^{(i)}$. Each one contains a weighting array $W^{(i)}$, a bias array $b^{(i)}$, and activation function $\sigma^{(i)}$, the activation function derivative, $\sigma'^{(i)}$. In all cases, $i \in \{0, 1, ..., L-1\}$
Require: Set of linear and non-linear activation arrays $z^{(i)}$ and $x^{(i)}$.
  Execute forward pass in algorithm (1) and compute the gradient of the cost with respect to the final layer activations
  $dx \leftarrow \nabla_{(y^*)} J(y, y^*)$
  Initialize ∇J as output object, should have same shape as Θ
  for $l \in [L-1, L-2, ..., 2, 1]$ **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 it by a desired learning rate α, and (ii) add the gradient vector element-wise to the existing elements in Θ object. By repeating steps (i) and (ii) in succession, we gradually drive the cost function to produces consistently lower and

lower values across a data set [2]. This is called *gradient descent* and is the basis for many optimization algorithms. Let $\bar{J}$ be the average cost over a batch of b samples. We show the general update rule on iteration (i) for gradient based learning in Eq. (36).

$$\Theta^{(i)} = \Theta^{(i-1)} + (-\alpha)\nabla_{\Theta}\bar{J} \tag{36}$$

Where $\Theta^{(i)}$ will give the set of new, updated parameters of the network. Note that for Eq. (36), all operations are applied element-wise.

2.5.4 The Optimizer

An optimizer is the algorithm or procedure that is used by a machine learning model to perform the optimization task and seeks to achieve the lowest possible value of the cost function over a data set [2]. Regression models my often used a *mean-squared error* cost function which can often be solved analytically, while high dimensional neural networks, such as the one used in this project, outlined in sec. (2.6), can only up updated iterativly [4, 7, 12]. We employ an iterative gradient-based method, much like the one in Eq. (36). These standard gradient-based learning algorithms are cumbersome and often prone errors such as vanishing or exploding gradients [2, 4, 12]. To combat this, model must often implement an algorithm that is more stable, and robust in it's ability to drive the cost function to a successively lower value.

For this project, we employ an *Adaptive-Moments* optimization algorithm, also called *ADAM* for short. It is a powerful optimizer that uses an adaptive learning rate and momentum parameter. ADAM tracks an exponentially decaying average of past gradients, $s^{(i)}$, and an exponentially decaying average of past squared gradients, $r^{(i)}$ in Eq. (37))[2]. This produces a far more aggressive optimizer at a higher computational cost than standard gradient descent.

As the cost function iterates through each step, the effect will tend to "snowball". If the previous step was found to reduce the cost function by a large or small amount, the step size for the previous step will update accordingly. This enables to ADAM to overcome smaller discontunities that may arise in the solution space [4]. Note that a superscript (i) or $(i-1)$ gives an iteration index, while the superscript i means to raise a value to the power of i. For a given step (i), The ADAM update is given:

$$\begin{aligned}
s^{(i)} &= \rho_1 s^{(i-1)} + (1-\rho_1)\nabla_{\Theta}\bar{J} \\
r^{(i)} &= \rho_2 r^{(i-1)} + (1-\rho_2)\Big[\nabla_{\Theta}\bar{J} \odot \nabla_{\Theta}\bar{J}\Big] \\
s'^{(i)} &= \frac{s^{(i)}}{1-\rho_1^i} \\
r'^{(i)} &= \frac{r^{(i)}}{1-\rho_2^i} \\
\Theta^{(i)} &= \Theta^{(i-1)} + (-\alpha)\frac{s'^{(i)}}{\sqrt{r'^{(i)}} + \delta}
\end{aligned} \tag{37}$$

ADAM has experimentally shown to be a very powerful and robust optimizer useful for a wide range of tasks [4]. Because of the two decay constants ρ_1 and ρ_2, we can 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 [2].

Algorithm 6 Adaptive-Moments (ADAM) optimizer for a neural network. This algorithm is adapted from Goodfellow, [4]

Require: Step size α
Require: Small constant δ for numerical stabilization, usually about 10^{-7}.
Require: Constants ρ_1, ρ_2 used track exponential decay rates, usually 0.9 and 0.999 respectively.
Require: Subroutine/function to compute gradient of cost function See Alg. (5)
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 (5) and normalize by batch size m:
$\nabla \bar{J} \leftarrow \frac{1}{m} \nabla_\Theta \Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)}) \Big]$
Compute first bias moment: $s \leftarrow \rho_1 s + (1 - \rho_1) \nabla \bar{J}$
Compute second bias moment: $r \leftarrow \rho_2 r + (1 - \rho_2) \big(\nabla \bar{J} \odot \nabla \bar{J} \big)$
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

2.6 Multimodal Architecture

In addition to choosing a strong set of features, and an appropriate optimizer for a classification task, it is also important to combine the predictors with a complementary network architecture. Since a neural network is a computational graph with each layer represented by a node, it can take on a nearly infinite number of different shapes and structures [4, 26]. The structure of the network defines the composition of it's function, F^* as in Eq.(2), and thus affects it's performance in completing any given task [2]. The layer structure of the neural network is referred to as it's *architecture*, and define it's *hypothesis - space* [4].

For this project, we have derived features that describe that audio file using two different *modalities*. A modality is a method of describing and interpreting a sample of data [16].

For example humans can be said to experience the world through five modalities - those being our cardinal senses of sight, hearing, taste, smell and touch. The sight and sound of a guitar are two very different ways of experiencing the same thing. One or the other is often enough to identify the guitar - but both together yield an even higher predictive confidence. Since modalities are *different* by nature, we cannot simply combine them into a single object and present the neural network with it. We must account for the difference by provide two distinct input layers. Modalities of seeing and hearing are captured by two different senses but ultimately processed by the same brain to produce a single prediction.

We can carry this idea over by producing a *multimodal* neural network that accepts two different forms of input for each sample. One input is an $N' \times k$ spectrogram matrix, explored in sec. (4.2) and the other input is a $1 \times p$ feature vector explored in sec. (4.3) and sec. (4.4). The spectrogram acts to provide an energy distribution of a signal as a function of frequency and time, while the feature vector provides qualities in a more list-like fashion. This type of supervised learning is referred to as *multi-view* learning, and we detail a conceptual diagram of the architecture in Fig. (6) [10, 16].

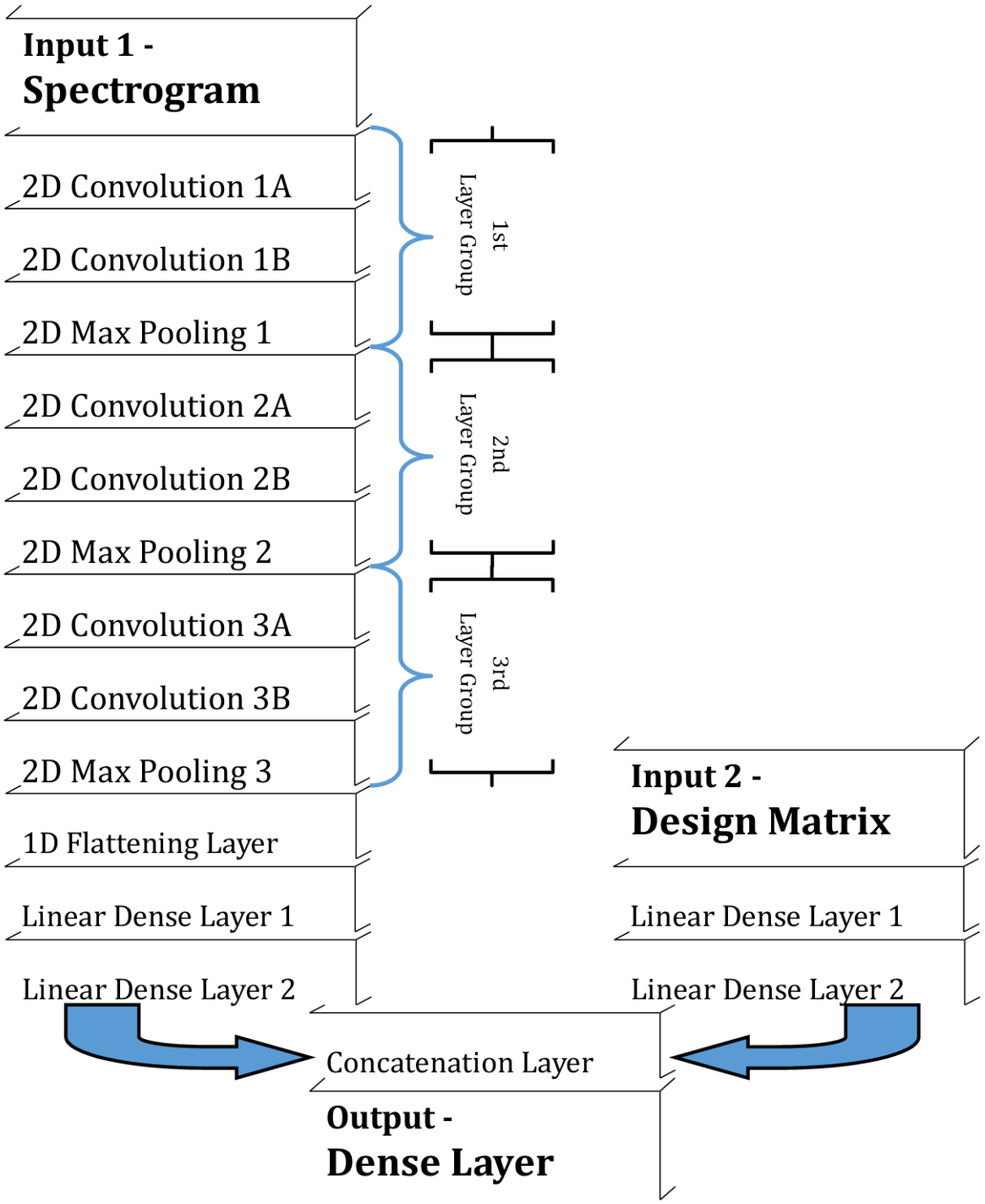


Figure 6: The implemented miltimodal architecture of the audio file classification neural network. The Left branch process an image-like input, the right branch processes a vector-like input. The activations are then merged, and then a single output vector is produced

2.6.1 The Spectrogram Branch

The spectrogram branch is pictured on the left side of Fig. (6). A spectrogram is a representation of a sound wave, or signal by representing energy distributions as a function of *time* and *frequency* [27, 17, 8]. 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 the image. For this model, we have selected to use 64 samples per batch, 558 pixels in width, 256 pixels in length, and 1 gray-scale channel. We denote the 4D shape of the design matrix for this branch as:

$$X_1 \in \mathbb{R}^{(64\times 558\times 256\times 1)} \tag{38}$$

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

After the input layer holds store the input activations, X_1 and tests for the appropriate shape, the array which is a collection of 64 spectrograms, is passed into the first of three *Convolution Layer Groups*. These layer groups are inspired from the *Visual Geometry Group-16 Neural Network* architecture [4, 12]. Each convolution layer group is composed of three individual layers:

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

The Convolution layers convolve over the middle two axes of the data. This corresponds to convolving as weighting kernel over the time and frequency axes.

By grouping layers in this structure, we use convolution to reduce the number of features, and then the pooling layer to extract only the largest activations of the remaining values. This drastically reduces the width of each layer before passing the activations down, and ensures that only the dominant features and characteristic shapes are preserved and passed into the next layer set for processing. This also allows for positional invariance in features. Regardless of where in frequency or time a feature is detected, the moving kernel will still allow it to be found [4, 12].

2.6.2 The Perceptron Branch

The Perceptron branch is pictured on the right side of Fig. (6), notice that it is considerably smaller in depths and complexity than it's neighbor. Rather than accept an image-like input, the perception simply takes a $1 \times p$ vector-like input of properties that are derived from the audio file. We call these properties *features* [2, 8, 22]. Details on the natures and selectiosn of these p elements are found in sec. (4.3) and sec. (4.4).

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-parameter of $b = 64$ samples per batch, and have developed $p = 24$ 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:

$$X_2 \in \mathbb{R}^{(64\times 24)} \tag{39}$$

This 2D array is referred to as a *design matrix* [7, 12]. Any other shape will be rejected by the model, and an error is raised.

In perceptron models, the scaling of the design matrix is vital to 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 [2, 7]. 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.

2.6.3 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 1×64 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 1×128 vector. This vector is then transformed into the final dense layer, which uses the softmax activation, and encodes the joint predictions based on contributions of both model branches. Again, compare this the the concept of two different bodily senses detecting the same object or environment. Information is produced from a single sources, received through multiple senses, but ultimately those experiences are combined in a such a way to produce a single prediction label.

3 Properties of Musical Instruments

As discussed in Sec. (2), performance quality of a neural network is greatly dependent on the properties of the chosen features [26, 11]. To ensure that a the classifier model performs adequately and consistently, we must choose these features to have high variance between classes, and low variance within each class. This enables a neural network to development clear decision boundaries between each unique class [7, 22]. To develop this set of features, we explore the physical and mechanical properties of musical instruments and the sounds that they create.

We have assembled training data samples than can be grouping into 37 classes of unique sources listed below:

Class Index	Class Name	Counts	Class Index	Class Name	Counts
0	Alto Flute	72	20	Marimba	364
1	Alto Saxophone	128	21	Oboe	666
2	Banjo	74	22	Sawtooth Wave	200
3	Bass	1060	23	Tenor Saxophone	732
4	Bass Clarinet	1036	24	Sine Wave	200
5	Bass Flute	76	25	Soprano Saxophone	128
6	Bassoon	800	26	Square Wave	200
7	Bass Trombone	54	27	Tenor Trombone	66
8	$B\flat$ Clarinet	938	28	Triangle Wave	200
9	Bells	164	29	Trombone	831
10	Violoncello	1079	30	$B\flat$ Trumpet	627
11	Contrabassoon	710	31	Tuba	1046
12	Crotale	50	32	Vibraphone	334
13	$E\flat$ Clarinet	78	33	Viola	1173
14	English Horn	1382	34	Violin	787
15	Flute	1032	35	Whitenoise	200
16	Guitar	106	36	Xylophone	176
17	Hihat	10	37		
18	French Horn	740	38		
19	Mandolin	364	39		

Figure 7: Instruments categories used in the classification task. Note that the network uses only integers given by "Class index" to identify sources. The string "Class Name" is kept for human readability

Rather than explore each class individually, we can examine the properties of groupings of musical instruments by using the *Hornbostel-Sachs* organization system. In this syetem, musical instruments or sound sources can be divided up into four broad categories based

on the nature of the sound-producing material. These categories are (i) idiophones, (ii) membranophones, (iii) chordophones, and (iv) aerophones. In this section, we explore the mechanical properties of each group and show how the physics of the source instrument influences our feature choice.

3.1 Idiophones

An idiophone is an instrument that produces sound through the vibration of the full body of the object. This generally includes most percussive instruments excluding drums citation needed. From the set of classes that we use, bells, crotales, Hi-hats, vibraphones, and xylophones are all examples of idiophones.

3.2 Membranophones

3.3 Chordophones

A chordophone is a musical instrument that produces sound through the vibration of strings that are stretched between two fixed points citation needed. From our data set, banjos, basses, cellos, guitars, mandolins, violins, and violas are all examples of chordophones.

3.3.1 The 1-Dimensional Vibrating String

We can model a chordophone as having a string of length L, with a constant linear mass density, μ, subject to a tension force F_T. The string is flexible, and free to vibrate in a single dimension. We describe the state of string over space and time with a function $U(x,t)$. The behavior of the string can be modeled by D'Alembert's 1-dimensional wave equation [5, 6, 24]:

$$\frac{\partial^2 U}{\partial t^2} = v^2 \frac{\partial^2 U}{\partial x^2} \tag{40}$$

Where v is the wave propagation velocity, given by $\sqrt{\frac{F_T}{\mu}}$. D'Alembert's equation indicates that the acceleration of any point on a string is directly proportional to the rate of change of the slope of the string at that point. Additionally, the string is fixed in place at both ends, with a nut at $x = 0$ and a bridge at $x = L$, which can be modeled by the boundary conditions [17, 27, 5]:

$$U(0,t) = 0 \qquad \text{and} \qquad U(L,t) = 0 \tag{41}$$

In order to specify a solution, we must also specify two initial conditions. We described the initial state of the string at time $t = 0$ by the two conditions:

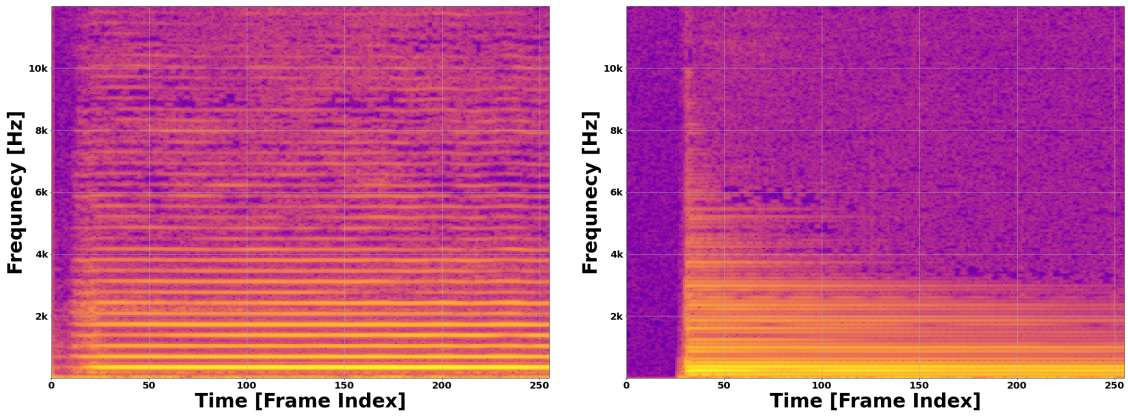
$$U(x,0) = f(x) \qquad \text{and} \qquad \frac{\partial U}{\partial t}(x,0) = g(x) \tag{42}$$

Depending on the exact nature of the source instrument, $f(x)$ and $g(x)$ can take on different forms. For example, for plucked stringed instruments, $f(x)$ may appear to be as triangle-like

function, with a peak at some position $0 < x_0 < L$ where then string was struck. It is these initial conditions that allow for the drastically different timbres that arise in the wave forms.

Despite these differences, the wave equation is a *linear* partial differential equation which allows for the fact that a linear combination of solutions is also a valid solution. In the case of the taught finite string that is fixed at both ends, such a guitar, viola or cello, the a solution tends to take the form of a sinusoidal function such a sine or cosine [6, 24, 27].

The boundary continuations, and most forms of standard initial conditions allow for the each sinusoidal to fit exactly an integer wavelengths into the region $[0, L]$ [5, 6]. Physically, this corresponds to having the string vibrate with some fundamental frequency f_0, which represents it's lowest possible note,and a linear combination of higher harmonics, $n \times f_0$, where $n \in \mathbb{Z}$ [17, 24, 27].



3.4 Aerophones

An aerophone is a musical instrument that produces sounds through the vibration of an air molecules in a column-like structure.

3.5 Other Generated Sounds

We group sounds produced from non-physical musical instruments in this final category. Form our data set, this category includes the four simplest waveforms, (i) sine waves, (ii) sawtooth waves, (iii) square waves, and (iv) triangle waves, as well as any color noises [27]. Since these sounds are only produced synthetically through a MATLAB program, we choose not to include them in the Hornbostel-Sachs grouping above. We explore the properties of each of these generated sounds.

3.5.1 Sine Wave

A sine wave is the simplest of sound waves, and makes up the foundation for all periodic wave forms [27, 26]. It consists of a single oscillatory term with a fixed fundamental frequency

f_0. A sine wave's amplitude as a function of time is given by

$$x(t) = A_0 \sin(2\pi f_0 t + \phi) \tag{43}$$

Where A_0 gives an amplitude, f_0 is the fundamental frequency, and ϕ is a phase shift. We show a simple sine wave in the time and frequency domain in Fig. (9).

[Image of Sine Wave Waveform] [Image of Sine Wave Freq. Spect]

Figure 9: A sine wave in the (a) time-domian and (b) frequency domain

A pure sine wave contains no overtones, only a time-independent fundamental frequency, which can take on any value due to synthetic generation.

3.5.2 Sawtooth Wave

3.5.3 Square Wave

A square wave is a piece-wise defined waveform whose value alternatively alternates between $+1$ and -1 [27, 17]. A square wave with period f_0 has a waveform defined over a single period as:

$$x(t) = \begin{cases} +1 & 0 < x \leq T_0/2 \\ -1 & T_0/2 < x < T_0 \end{cases} \tag{44}$$

where $T_0 \equiv 1/f_0$.

We can use Fourier decomposition to examine the constituent sinusoidal wave that make up a square wave. The frequency space representation of a square wave indicates that is can be approximated as a sum of higher odd-numbered harmonics, with the amplitude of each decreasing by the reciprocal of the harmonic number [27]. Thus a square wave in the time domain can be approximated as:

$$x(t) \approx \sum_{n=1}^{\infty} \frac{1}{2n-1} \sin\Big(2\pi(2n-1) f_0 t\Big) \tag{45}$$

[Image of Square Wave Waveform] [Image of Sqaure Wave Freq. Spect]

Figure 10: A square wave in the (a) time-domain and (b) frequency domain

3.5.4 Triangle Wave

3.5.5 White Noise

4 Feature Selections

Classification tasks require a set of inputs called *predictors* or *features* [7, 12, 22]. A features is a quantitative, low-dimensional representation of a sample that conveys key characteristics of it. For example, in classifying types of animals, a feature could be the mass, volume or number of teeth on the animal. Typically, we produce a set of features p for each sample and assemble them into a $1 \times p$ vector called a *feature vector* which acts as a list of descriptive qualities of the sample [2, 7]. Audio machine learning researchers M. Kashif Saeed Khan and Wasfi G. Al-Khatib discusses the vitality of feature selection [8]:

> 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.

An adequate selection features is vital to the process of training and evaluation of the classifier model [14, 22, 11]. Consider the task of identifying cats and dogs from images, given only the top-most row of pixels or the number of eyes. While the data represents *correct* properties, it is not enough information to distinguish the two categories.

In the biological process of categorizing sound sources, the time-domain waveform is usually enough to match the sound to a source [17]. However, a computer representation of a waveform is an array-like structure of values sampled discretely in time and volume [26, 11]. Presenting the raw time-series waveform to a model directly has experimentally shown to be unstable in for classification tasks so we must develop a set of features or predictors that can describe important properties of the waveform in a far more efficient manner [4, 7, 22].

To ensure the construction of a suitable model, we derive features based from three representations of the audio file: (i) a spectrogram matrix representation, (ii) the time-space representation, and (iii) the frequency-space representation. It is important to note that although this algorithm will map audio files to source instruments, the model will never actually be presented with a waveform directly, instead it will rely solely on the features to make predictions.

4.1 Feature Space

From each audio file sample, we calculate the same predictors that will be use for classification. The predictors are arranged into two separated structures, each called a *design matrix*, described in sec. (4.1.2) [4, 2]. Each sample in the design matrix X_1 contains $N \times k$ predictors, and the matrix X_2 contains $1 \times p$ predictors. For the purposes of exploring the properties of feature space, we consider the p predictors for each sample in X_2.

Feature space is a p dimensional vector space where each basis vector represents a predictor [4, 7]. Each sample can be described by this p-dimensional vector, given by each row of the design matrix X_2. For samples $x^{(a)}$ and $x^{(b)}$ that belong to the same class, we expect the

quantity $||x^{(a)} - x^{(b)}||_2$ to be reasonably small, indicating that the two sample lie "close together" in space. Similarly, for samples $x^{(c)}$ and $x^{(d)}$ that belong to different classes, we expect the quantity $||x^{(c)} - x^{(d)}||_2$ to be comparably large, indicating that the two sample lie "far apart" in space. Not that this is an idealization, and not always the defining case for the decision behavior.

We can expand this idea by considering by how features should behave within each class, and between each class. 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 [26]:

> 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.

This is to say, features should *ideally* form non-overlapping groups or bubbles that allows classifiers to effectively differentiate categories. Given the complexity of real-world problems, this is an often unrealistic goal [4, 7]. However under the right circumstances, classifiers can still produce parameters that allow for reasonable performance [2, 12]. In practice, a neural network does not rely on comparing the norms of different samples in feature space, but rather uses something called a *decision boundary*.

4.1.1 Decision Boundaries

A trained classifier makes a prediction in the form of a statistical likelihood of a sample belonging to a particular class [4, 7]. A k- categories classifier contains k output neurons, each one representing a class. When a set of features is presented to the classifier, the value contained within each neuron is the "confidence" on the interval $[0, 1]$ that a sample belongs to that class. For example, if neuron q has an activation value of 0.25, we say that there is a 25% chance that the sample belongs to class q. Most commonly, we take the neuron with the highest activation value to be the chosen class, even if the next highest activation is a close second [12].

This concept stems from the idea that when a trained neural network classifier makes a prediction, it is doing so based on a set of generated *decision boundaries* [2, 7]. To explore decision boundaries, consider a simplified *toy data set* with a 2D feature space for a classifier that must only discern between *Class A* and *Class B*. We can visualize each sample in this 2D space, and color-code (and shape-code) each sample according according to it's class label as in Fig. (11).

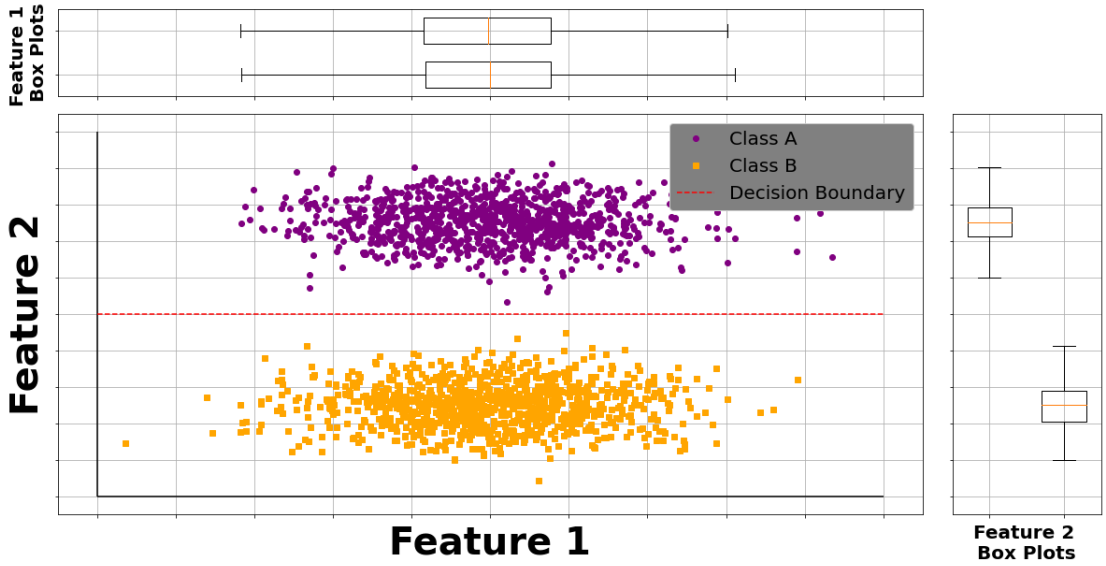


Figure 11: Feature Space with Linear separability between two classes

In the case of these chosen, features, we can see a clear, well-defined separation between samples in each class. In practice, this is rarely the case, but shows graphically an example where a linear decision boundary arises [7]. Notice how we can also use "box-and-whisker plots" to neatly summarize how each feature (each component) behaves by dividing the samples for each class based into quartiles. Notice how on the right of Fig.(11) the box plots for feature 2 show no overlap, but the box plots for feature 1, at the top show very similar behavior between classes. Despite this, a classifier can produce a clean boundary to separate the two classes. Note that this may not be the exact boundary used by a model, but it represents an ideal case.

We can also explore an example where both features have mutual overlap between classes, but we can still form a reasonable decision boundary in the space. Consider Fig. (12) and Fig. (13). In both cases, we see the box plot for each feature overlap considerably, and we also see sample that appear to "cross" the decision boundary. This crossing the boundary is why it is so useful that we choose an activation function that encode the output of the neural network as a probability of being in either class as opposed to a hard prediction [12, 2].

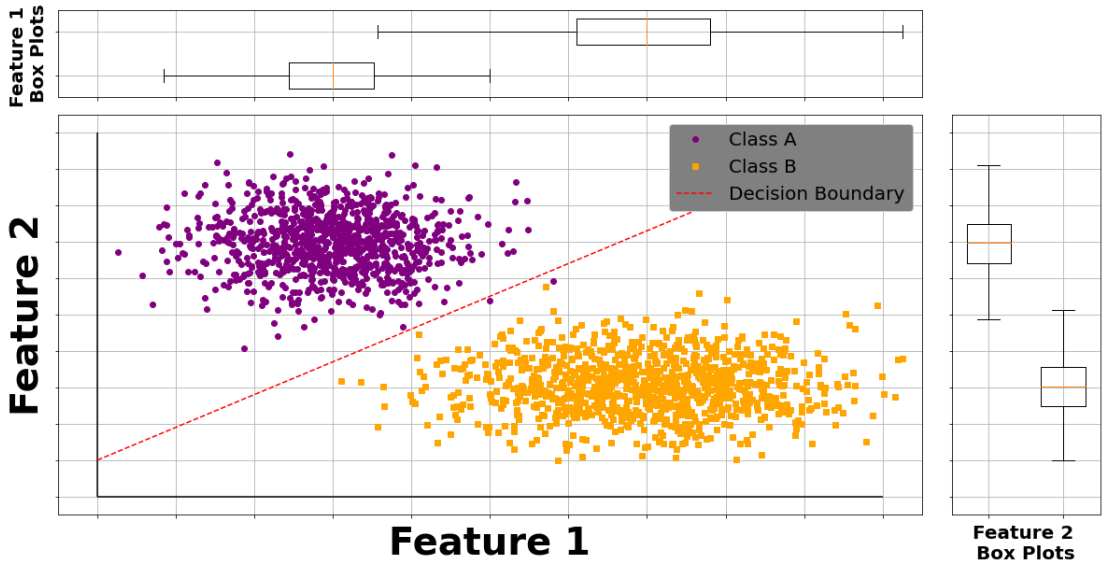


Figure 12: Feature Space with near linear separability between two classes

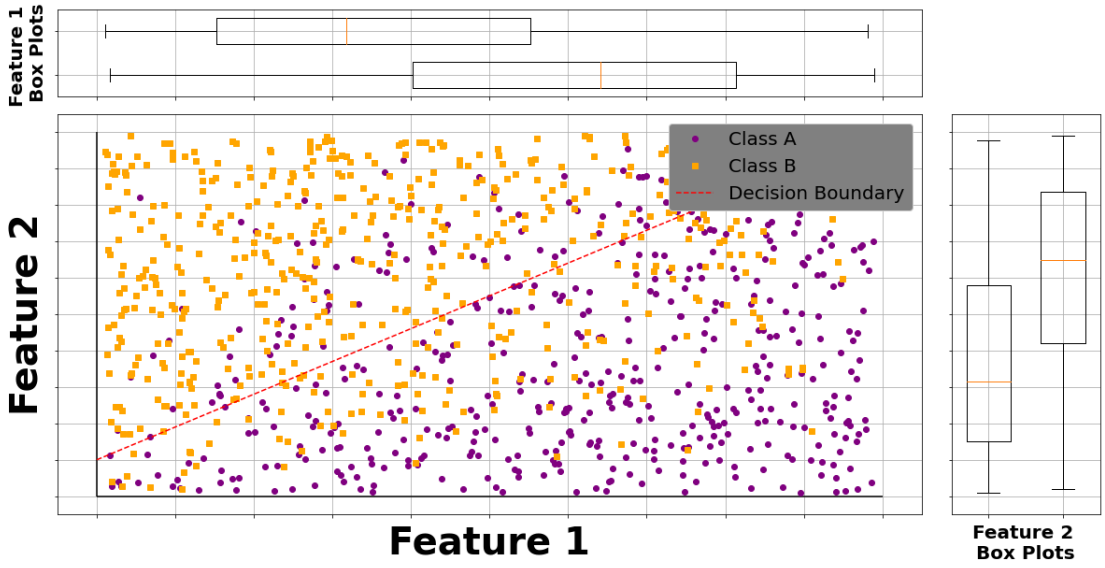


Figure 13: Feature Space with near linear non-separability between two classes

Samples that appear very near to decision boundary will likely have almost equal probability of occurring in either class, and samples that appear very far from the decision boundary will likely have very much larger probability of occurring in one class or the other [1, 7].

Note that these are over-idealized cases, constrained to two classes in two dimensions. The classifier implemented in this project uses 24 classes in a single branch and outlines a probabilty estimate over 37 classes. Given the related nature of many of the categories, (i.e. belonging to similar Hornbostel-Sachs groups), such well-defined decision boundaries are unlikely to be produced. Note also that the examples provided showed only *linear* boundaries, where the inclusion of non-linear activation functions will create far more complex systems in a far higher-dimensional space.

From these examples, we conclude the mathematical nature of features allow each sample to be characterized by a vector in p-dimensional feature-space[4]. We expect feature of the same class to exhibit similar behaviors as to allow a trained classifier to develop multiple decision boundaries in the higher-dimensional space [7, 1]. As a consequence, we derive the predictors from the physical mechanical nature of each instrument or sound-source.

4.1.2 The Design Matrix

Each audio sample is given by an $N \times k$ spectrogram matrix explored in sec. (4.2), a $1 \times p$ feature vector, and a target label, y for training purpose. As explored in sec. (4.1.1), each sample can be associated with a vector in feature-space. In order to efficiently present the information to the model for training or predicting, we extract a batch of b samples and produce the two arrays form each.

Since the spectrogram and feature vector represent different *modalities*, we construct two separate *design matrices*, X_1 and X_2 [2, 10, 16]. Each has shape given by:

$$X_1 \in \mathbb{R}^{(b \times N \times k)} \tag{46}$$

$$X_2 \in \mathbb{R}^{(b \times p)} \tag{47}$$

Where $N \times k$ gives the shape of the spectrogram matrix S from a single sample, described in sect. (4.2). Similarly, p is the number of predictors extracted from each sample, outlined in sec. (4.3) and sec.(4.4). The first axis, with size b indicates that there are b audio file samples stored in that design matrix. Organizing the samples in this fashion allows for both easy machine and human interpretation of each axis or index.

A design matrix is typically accompanied by a *target matrix* Y, which indicates the class that the particular sample belongs to. For a classifier with κ classes, each sample labeled by a one-hot-encoded vector for shape $1 \times \kappa$ [25, 12]. Thus, for the design matrices X_1 and X_2, we have a corresponding target matrix given by:

$$Y \in \mathbb{R}^{(b \times \kappa)} \tag{48}$$

4.1.3 Audio Preprocessing

Preprocessing a data set is a necessary step to execute prior to feature extraction [3, 7, 22]. In the case of audio files, preprocessing usually consists of ensuring that the data set contains the following steps and requirements:

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

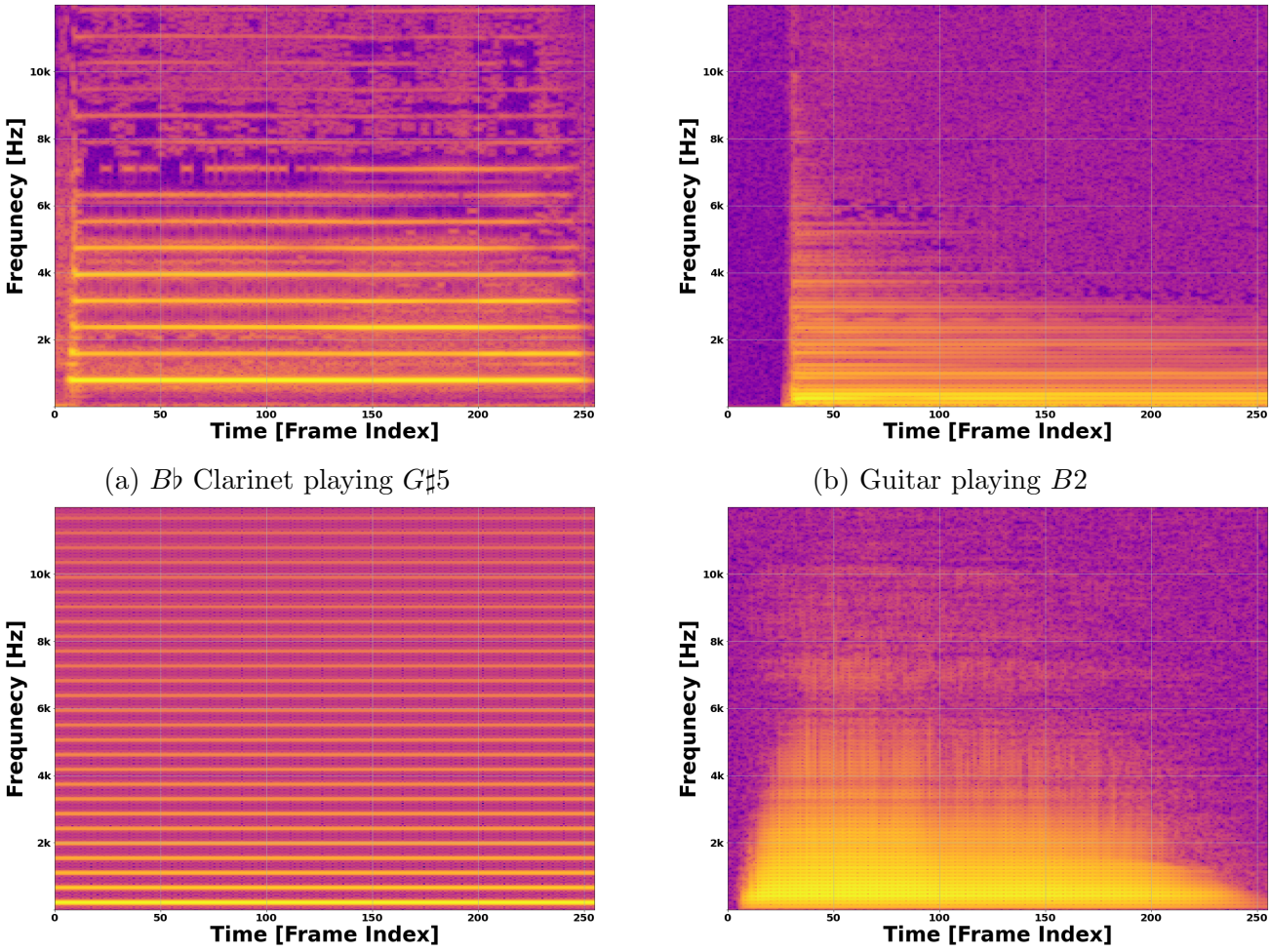
Note that different projects may require a different set of requirement from preprocessing [26]. For this project, we have chosen to use the following parameters:

1. Roughly $18,000$ 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.

Given these standardization methods, we can exact the predictors for each sample and explore how they behave mathematically, and what they represent physically.

4.2 Spectrogram Features

The application of neural networks to image-processing and recognition has been well studied and developed in the last few decades [2, 4, 12, 14]. 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. (14).



(a) $B\flat$ Clarinet playing $G\sharp 5$

(b) Guitar playing $B2$

(c) Square Wave at 220 Hz

(d) Tuba playing $A2$

Figure 14: Spectrogram representations of various waveforms

4.2.1 Frame Blocking

A spectrogram is produced by the method of *frame-blocking*, which is very prevalent in audio signal classification [11, 28]. 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 previous frame. Each of the k frames then allows for a section of the signal to be analyzed in a *quazi-stationary state* [8, 22]. Below we visualize how analysis frames are related to the full time-series waveform.

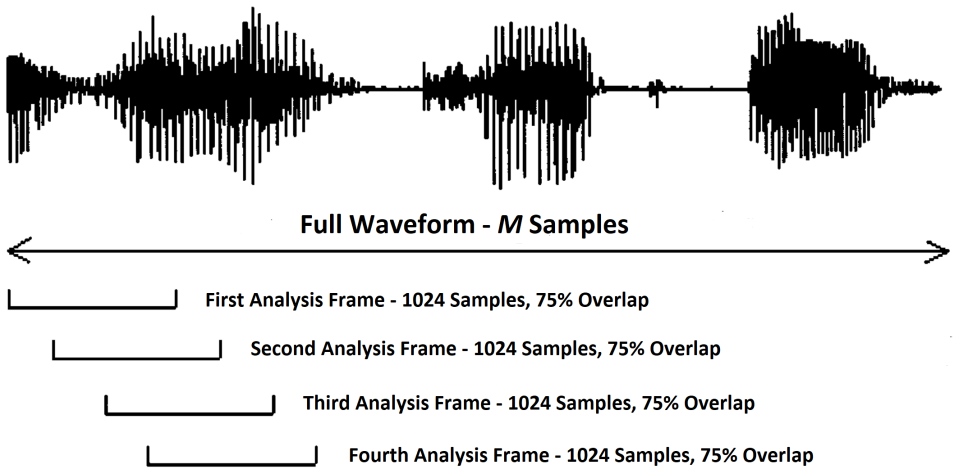


Figure 15: A visualization of how frame-blocking is used to create each analysis frames. This image has been adapted and modified from Liu, et. al. "Audio Feature Extraction and Analysis", Fig. (1). See ref. [11].

For this project, we have chosen to use frames of size $N = 1024$ with a 75% or 768 sample overlap. Since each audio file contains a different number of samples, we choose the number of frames, k to be less than or equal to 256. If $k > 256$, the waveform in truncated, if $k \leq 256$ the frames left as is, and missing frames are accounted for later (to improve computation time). The audio has been sampled at $f_s = 44100$ samples/second, so each frame represents a slice of time that is about 0.0232 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 a sample in each frame

$$A = \{a_0, a_1, a_2, ..., a_{k-1}\} = \begin{bmatrix} a_0[0] & a_0[1] & a_0[2] & \dots & a_0[N-1] \\ a_1[0] & a_1[1] & a_1[2] & \dots & a_1[N-1] \\ a_2[0] & a_2[1] & a_2[2] & \dots & a_2[N-1] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{k-1}[0] & a_{k-1}[1] & a_{k-1}[2] & \dots & a_{k-1}[N-1] \end{bmatrix} \tag{49}$$

We use bracket notation, $a[i]$ to indicate that each analysis frame a_i is array-like. The following indexing conventions for matrix A all represent the same entry:

$$A_{i,j} = A_i[j] = A[i][j] = A[i,j] \tag{50}$$

4.2.2 Windowing

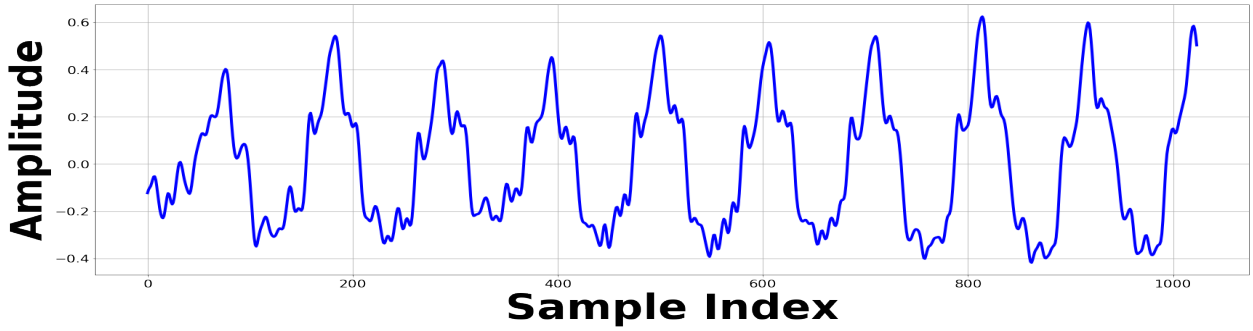
After frame-blocking, we apply a *windowing function* to each frame. A standard *Hanning Window* of N samples is generated as a $1 \times N$ row-array, H. The n-th index in a Hanning window with N samples is defined:

$$H[n] = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi n}{N-1}\right)\right] \tag{51}$$

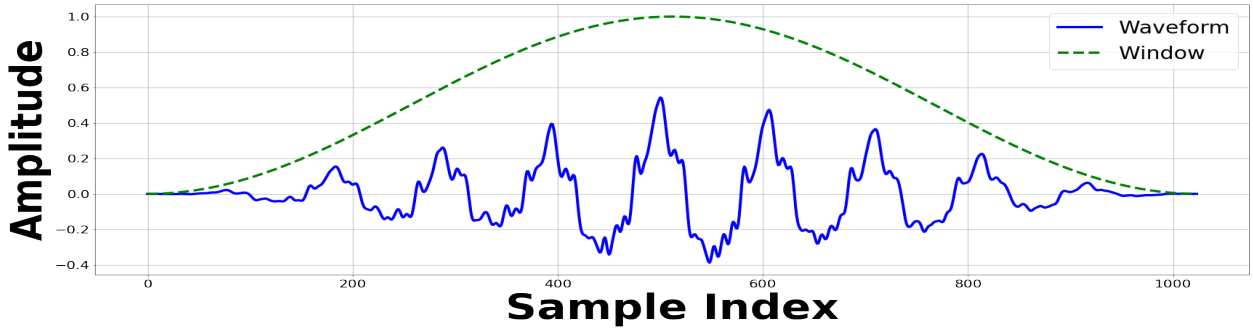
This window function is applied to each analysis frame by computing the element-wise product of the Hanning window array, H and each row of the analysis frames matrix, $A_{i,:}$. The result is another $k \times N$ array, $\widetilde{A}$:

$$\widetilde{A_i} = A_i \odot H \tag{52}$$

We show the effect of a Hanning window on an analysis frame:



(a) A section of a violin bowing an $A4$ note, 1024 samples long



(b) The same waveform with a Hanning Window Applied

Window functions are use to account for discontinuities that may arise in the waveform at the edges of each analysis frame. The Fourier Transform assumes that an integer number of

signal periods fit within each analysis frame, however this is rarely the case. Transforming a signal with to *spectral leakage* where energy appears to "leak" into adjacent frequency bins Citation needed. This result is a frequency domain that is not an accurate representation of the equivalent time domain. The Hanning window uses the raised cosine function to taper the signal off to zero at the end-points, thus eliminate discontinuities at the edges while weighting the samples at the center much more heavily.

Before we compute the Fourier Transform to move each analysis frame into frequency-space, we choose to tail-pad each analysis frame with an additional 1024 samples of all zeros. This means that each frame has gone from being 1024 samples, to 2048 samples. Matrix $\widehat{A}$ has a shape of $k \times 2048$. Doubling the size of each analysis frame allows us the synthetically double the resolution in frequency-space, while retaining the resolution in time-space [26].

4.2.3 Discrete Fourier Transform

With this final change, we perform a *Discrete Fourier Transform* (DFT) to bring each analysis frame from a time domain into a frequency domain [17, 18, 26]. 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 signal containing N samples is given by [24, 26]

$$\mathbb{W} = \frac{1}{N}\begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \dots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \dots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \dots & \omega^{(N-1)^2} \end{bmatrix} \tag{53}$$

Each column of the matrix is a complex sinusoidal oscillating with an integer number of periods within the N-sample length window [23, 18]. 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.

$$\text{DFT}\big[\widetilde{A}\big] = \mathbb{W}\widetilde{A}^T \tag{54}$$

Most standard implementations of neural network models have all activations, weights, and biases to be real floating-point numbers [9, 12, 25]. Since the elements of the DFT matrix lie on the complex unit circle, each real-valued analysis frame will be moved into complex space. This means we compute the element-wise product of the transformed signal matrix and it's complex-conjugate matrix. This is the $N \times k$ spectrogram matrix, where each element is a real-values number. Matrix S is defined:

$$S = (\mathbb{W}\widetilde{A}^T) \odot (\mathbb{W}\widetilde{A}^T)^* \tag{55}$$

Where $\mathbb{W}$ is the DFT matrix from Eq. (53) and $\widetilde{A}^T$ is the transpose of the analysis frames matrix from Eq. (52). The asterisks indicate the element-wise complex conjugation. In

practice, the matrix product $\mathbb{W}\widetilde{A}^T$ is computed once, and the conjugate is applied to a copy of the array.

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 Eq. (50):

$$S_{i,j} = S[i][j] = S[i,j] \tag{56}$$

Each column of the S matrix is now a single analysis frame that has been moved into a frequency-space representation of itself. There are k columns, just as there were k time-series frames, and N rows since there were N samples per row. Given the discrete 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 $+f_s/2$. To ensure homogeneous input sizes between all samples, we zero-pad the matrix S with additional columns until $k = 256$. Recall that wave forms were truncated to ensure that $k \leq 256$ analysis frames.

Standard western musical instruments seldom have fundamental frequencies that extend above 6 kHz [17, 26, 27]. This means that when constructing the spectrogram, we will rarely see significant energy present above this frequency at any time, and the S matrix will contain mostly zero, or zero-like entries. To condense the size of the matrix, we select only the frequency bins (rows) that correspond to energies between 0 Hz and $12,000$ 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 $N' \times k$ and effectively encodes the energy distribution of the waveform as a function of both time and frequency. The spectrogram makes up each sample in the first design matrix X_1, Eq.(46) 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. For a batch of b samples of $N' \times k \times 1$ spectrograms, we shape X_1 such that:

$$X_1 = \left\{ S^{(0)}, S^{(1)}, S^{(2)}, ..., S^{(b-1)} \right\} \in \mathbb{R}^{(b \times N' \times k)} \tag{57}$$

Which is consistent with the shape of the X_1 matrix outlined in Eq. (46). This matrix is presented to the *Convolution* branch of the neural network for processing.

4.3 Time-Space Features

The features described in this section are derived from time-domain representations of each audio sample. Each feature is one element in the feature vector for the particular sample. For consistency between samples, each waveform is padded or truncated to contain the same number of samples M. The number of samples M is chosen to correspond the number of sample needed to make the $N \times k$ spectrogram matrix. This way, the spectrogram and time-series features are representative of the same time interval in the file sample. Time space is indexed by $s[i]$ with $i \in [0, 1, 2, 3, ..., M-2, M-1]$

From time space, we use the following 11 features:

- Time Domain Envelope ($\times 5$)
- Zero Crossing Rate
- Temporal Center of Mass
- Auto Correlation Coefficients ($\times 4$)

4.3.1 Time Domain Envelope

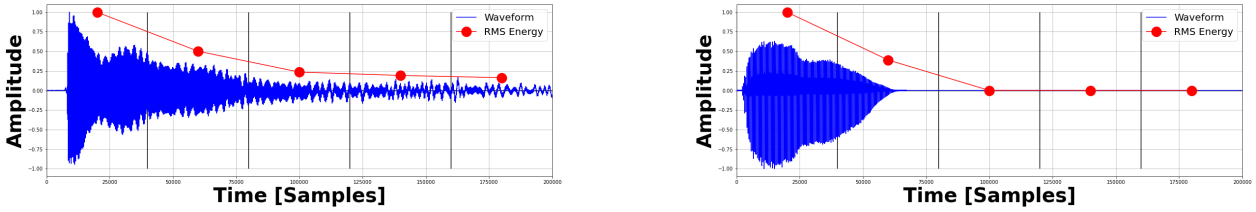
The time domain envelope (TDE) is a rough measurement of the energy contained in the time domain. If we divide the signal into analysis frames, as in Fig. (15), then the TDE approximates the energy in that frame. A frame with a high TDE indicates a generally large amplitude, and a frame with a low TDE indicates a generally small amplitude, i.e. the signal has not started or is decaying. The small size of each analysis frame makes computing a TDE for each impractical, and create a very high-dimensional feature vector. This would also cause problems as some waveforms may have an initial attack that differs in time by a few analysis frames. Computing TDE for each frame would therefore introduce temporal bias, as the same waveform delayed by a few milliseconds would generate a wildly different set of predictor values [22].

One way to mitigate this is to group analysis frames together and compute the TDE of a collection of frames. This way, almost all waveforms will produce an identical set of features to a short time-delayed version of itself. By choosing a sufficiently large number of TDE values, we can also generate an approximation of than amplitude envelope of the time-series waveform. This comes at a higher computational cost, and a previously mentioned higher dimensional feature vector

We adapt this TDE to **compute the time domain envelope over** 5 **non-overlapping analysis frames**. Since the maximum amplitude of each waveform array has been normalized to ± 1, the TDE represents a consistent measurement of energy in each waveform subset [11]. The TDE is computed as the RMS-Energy of the waveform, s and the j-th TDE value is given by [17, 22]:

$$\mathrm{TDE}_j\big[s\big] = \sqrt{\frac{1}{Q}\sum_{i=n}^{n+Q} s[i]^2} \tag{58}$$

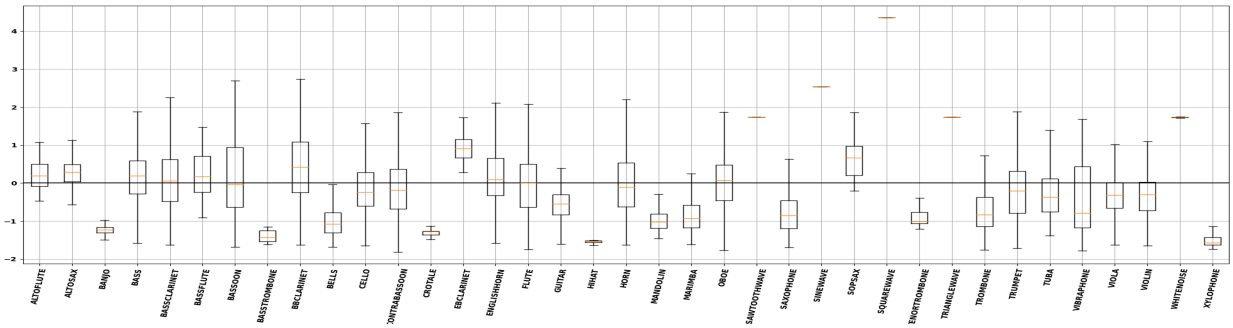
Where Q gives the number of sample sin each analysis frame (number of sample sin waveform divided by number of frames), and n gives the index where the j-th frame begins. We provide a graphic representation of how the TDE compares to the amplitude of sections of a waveform in Fig. (17).



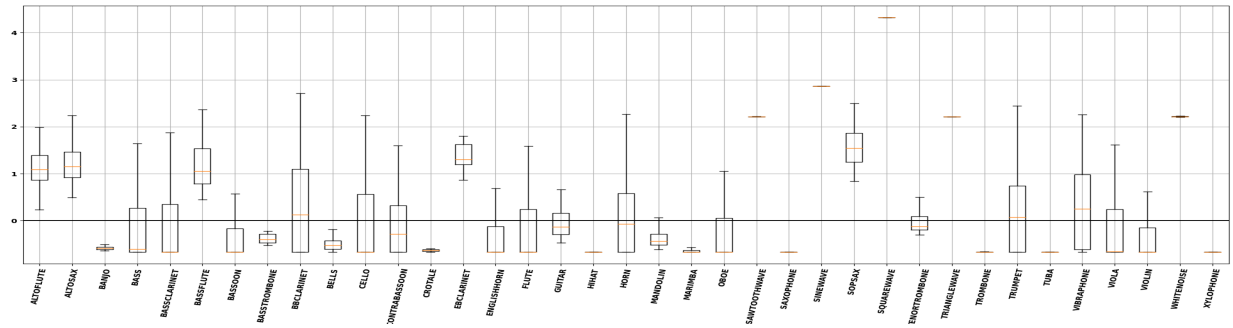
(a) Guitar - B2 (b) Tuba - A2

Figure 17: TDE Envelope values for musical instruments

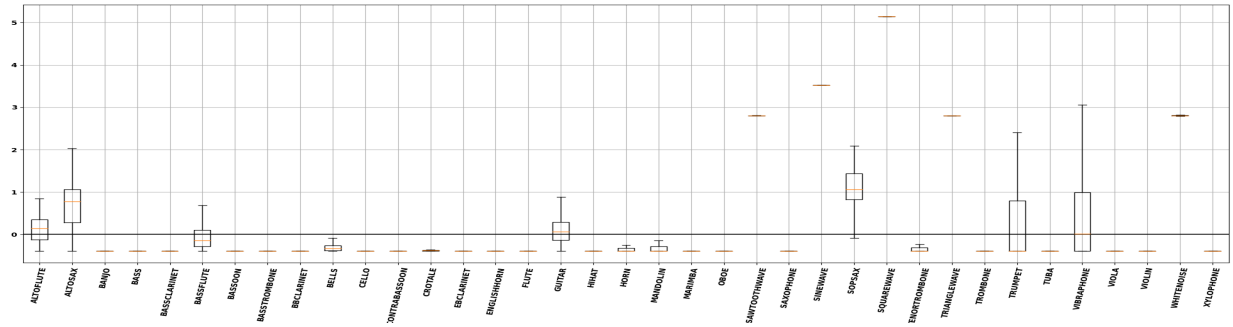
The uniformly sized analysis frames let us create an naive envelope of the waveform, but will allow it to generalize between classes. For example, this allows for a crude approximation of the energy in the attack, decay, sustain, and release portion of the signal [26, 17]. For instruments with heavier attacks, we expect the TDE in the 1-st frame to be comparably large, see Fig. (18a). Instruments with a heavy decay will likely have little energy in the 2-nd or higher frames, indicating that the amplitude has substantially died off as in Fig. (18b). Conversely, instruments with longer sustain such as upper woodwinds, vibraphones or strings will container relatively higher TDE values for the later analysis frames as seen in Fig. (18c). Instruments with little or no waveform envelope, such as the synthesized waveforms and the whitenoise show constant TDE values across each frame as expected.



(a) TDE 1



(b) TDE 2



(c) TDE 3

Figure 18: A comparison of the first 3 of 5 Time Domain Envelope across each class using a box-and-whisker plot

4.3.2 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 computed per total sound wave, per analysis-frame, or per unit time. This feature is most commonly associated with differentiating speech from music, because speech presents a more jagged and often less periodic waveform that musical instruments do [8, 11, 28] .

We adapt this feature to **compute the zero crossing rate for the full waveform**. Signals with a high ZXR can be representative of classes that often have additional noise and signals with a low ZXR, can indicate signals with more stable, periodic behavior. The ZXR for the full waveform s is given by [22, 11]

$$\mathrm{ZXR}\big[s\big] = \frac{1}{2}\sum_{i=1}^{M-1}\Big|\,\mathrm{sign}(s[i]) - \mathrm{sign}(s[i-1])\Big| \tag{59}$$

Where $\mathrm{sign}(x)$ returns $+1$ if $x > 0$, -1 if $x < 0$ and 0 if $x = 0$. ZXR also provides a very rough estimate for the average frequency of the waveform. This allows it to allows be useful in discerning classes with generally higher fundamental frequencies, such as upper woodwinds, against classes with generally lower frequencies, such as low brass [11, 27].

This behavior becomes apparent when comparing sets of classes in Fig. (19). For example, consider the ZXR value of a a bass and cellos against that of bells or a soprano saxophone. The bass and cellos have on average much lower fundamental frequencies than that of bellos of soprano saxophones. In the case of this data set, the ZXR provides a predictor which contains clear separations between classes such as crotales and $E\flat$ clarinets.

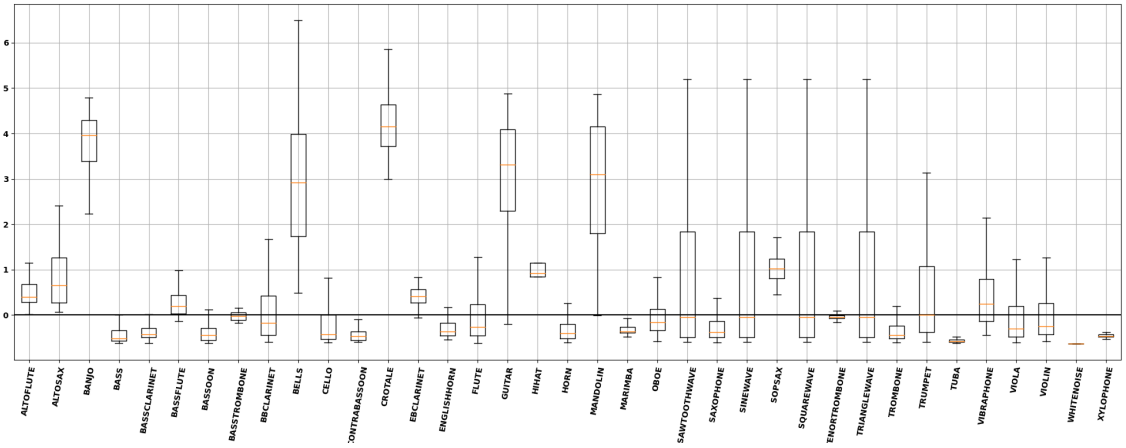


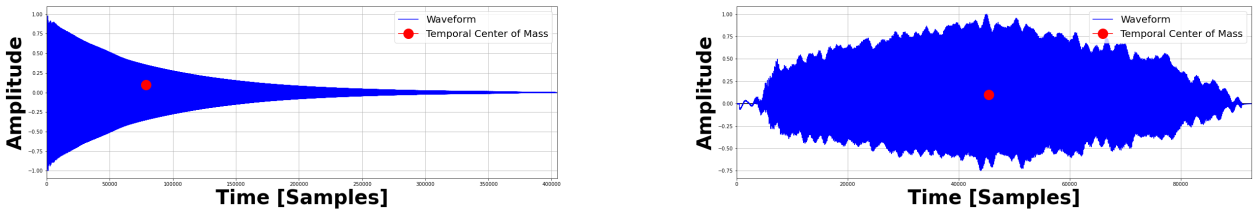
Figure 19: A comparison of the Zero-Crossing Rate for each class using a box-and-whisker plot

4.3.3 Temporal 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 compute the element-wise absolute value of the waveform and treat it as a 1-dimensional discrete mass distribution of M samples. The TCM of that waveform is then given:

$$\mathrm{TCM}\big[s\big] = \frac{\sum_{i=0}^{M-1} i\big|s[i]\big|}{\sum_{i=0}^{M-1}\big|s[i]\big|} \tag{60}$$

TCM condenses the idea of the amplitude envelope into a single scalar value. It allows for the quick measurement of where in the time most of the energy of the signal lies. For percussive instruments with short attack and release times, such as bells or xylophones, we expect a very low TCM. Instruments with plucked strings, but longer release times such as mandolins and guitars should have similarly low values. See Fig. (20) for examples.



(a) Vibraphone - A5 (b) Cello - F4

Figure 20: TCM values for musical instruments

Instruments in the woodwind and brass family generally have longer sustain and release times, giving them a slightly higher center of mass. Strings and undampened percussion have notoriously long sustain and release times giving them much higher TCM values [17, 27]. Finally, the synthetic wave forms have no characteristic envelope shape (due to their synthetic nature). This gives a centrally located TCM, very close to $M/2$.

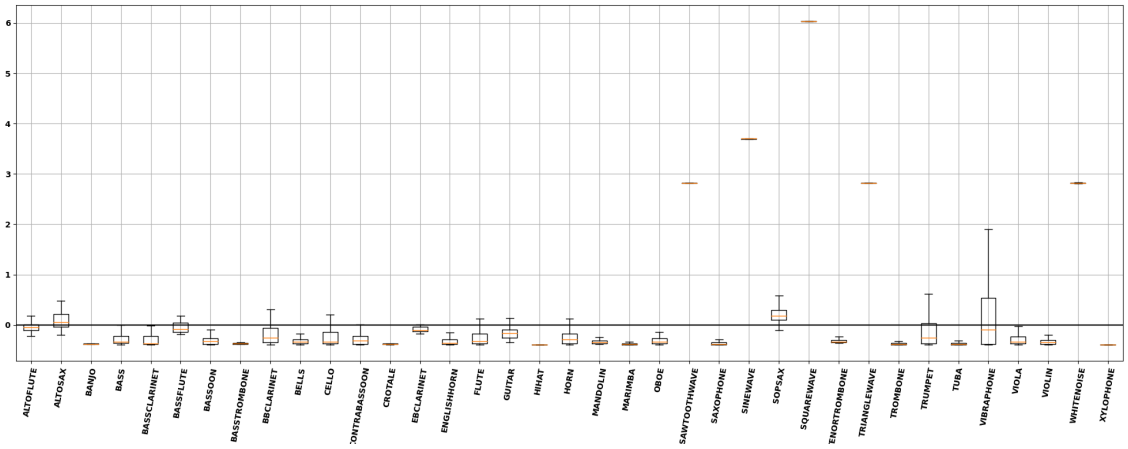


Figure 21: A comparison of the temporal center mass for each class using a box-and-whisker plot

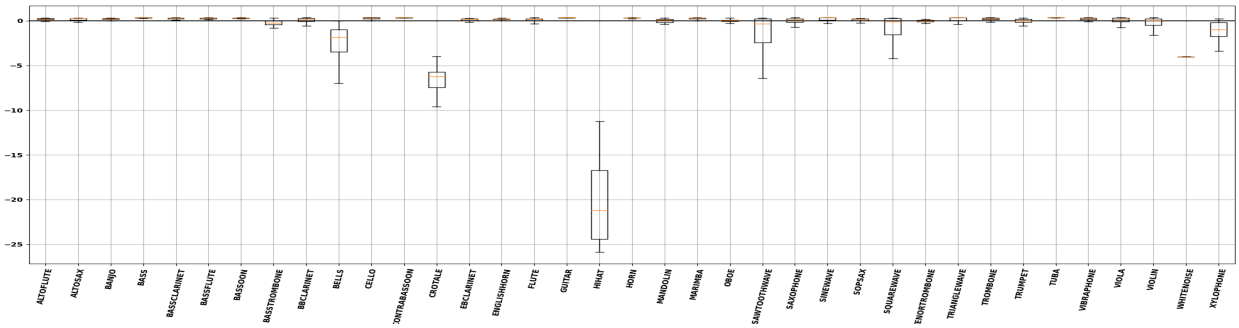
4.3.4 Auto Correlation Coefficients

Auto correlation coefficients (ACC) are rough estimates of the signal spectral distribution. They are computed by summing a signal with a time-delayed variant of itself, and then normalized depending on the contect. We can compute any number of ACC's and their

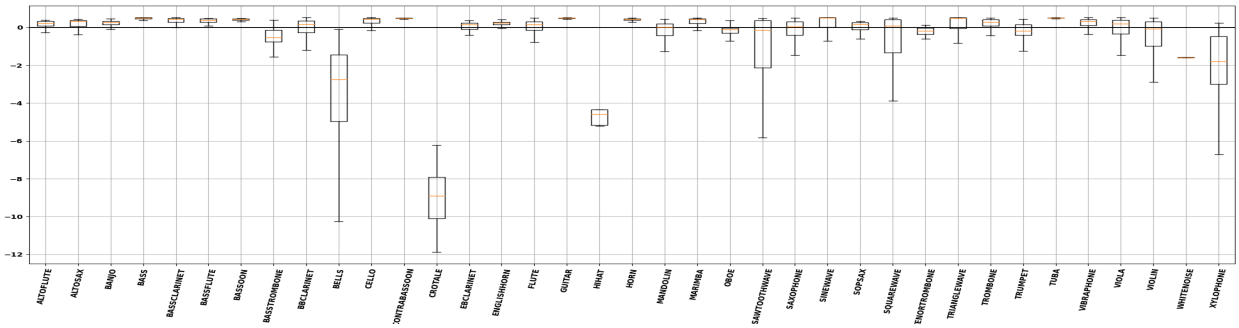
value changed depending on the index chosen. It is common to use the first K ACC's [22]. For a full waveform signal s, with M samples, the k-th ACC (indexed from 1 to K) is given by:

$$\mathrm{ACC}_k\big[s\big] = \frac{\sum_{i=0}^{M-k-1} s[i]s[i+k]}{\sqrt{\sum_{i=0}^{M-k-1} s^2[i]}\sqrt{\sum_{i=0}^{M-k-1} s^2[i+k]}} \tag{61}$$

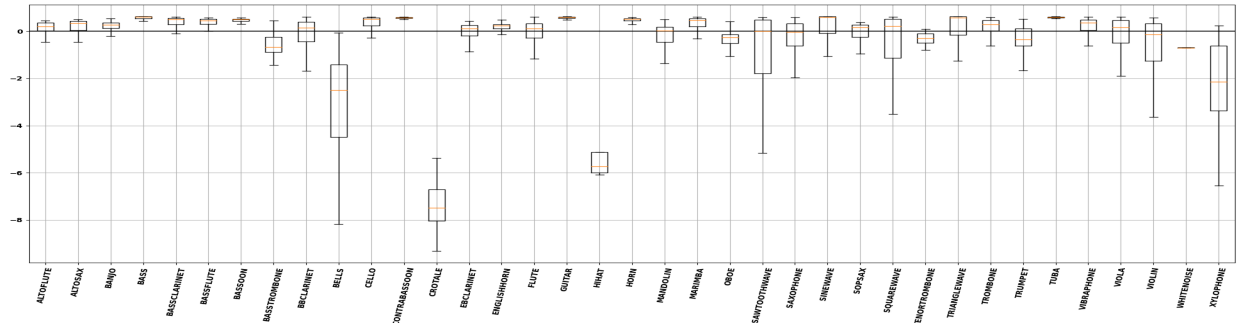
Physically, the numerator of the ACC representing computing the dot product of the signal, $s[i]$ with a time-delayed version of itself $s[i+k]$, and the denominator provides a normalization for the value. Dotting the signal and the time delay allows us to introduce a synthetic phase shift and compare the resulting relationship. If we chose k to be equal, or similar to the number of samples that make-up a period or half-period of the waveform, then $s[i] \approx s[i+k]$ and then $\mathrm{ACC}_k \rightarrow 1$. Alternatively, for many other values of k, frequent multiplication and summation of samples that are approximately zero result in $\mathrm{ACC}_k << 1$. This make auto-correlation coefficients extremely useful for detecting periodicity in time-space [22].



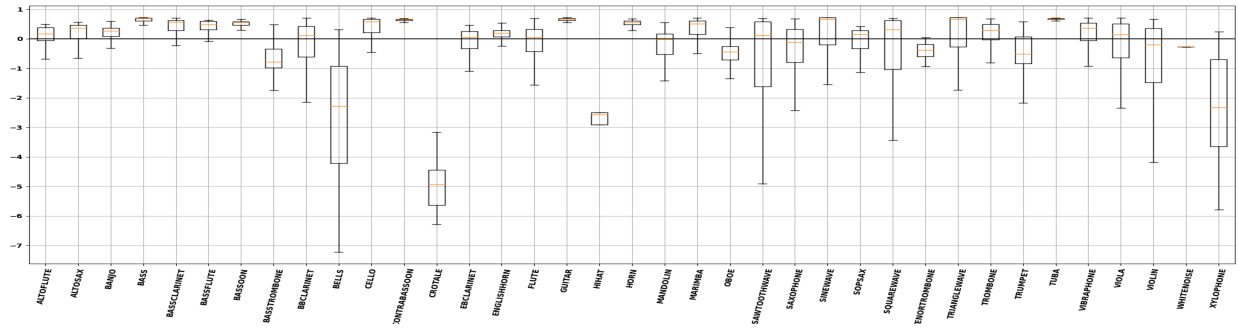
(a) ACC $k = 1$



(b) ACC $k = 2$



(c) ACC $k = 3$



(d) ACC $k = 4$

Figure 22: A comparison of the the first four auto correlation coefficients in each class using a box-and-whisker plot

4.4 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. (4.2). 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 Eq. (49). 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

4.4.1 Mel Filter Bank Energies

Mel Filter Bank Energies (MFBE's) are not used directly as features, but are used in computing Mel Frequency Cepstrum Coefficient (MFCC's) so we describe them here. Mel filter banks are divisions of the frequency spectrum of a signal into R overlapping bins [21, 22]. These filter banks allows us to group sounds based on their energy distribution in frequency space. Each filter is triangularly shaped, covering a certain band in frequency space, and zero elsewhere. This way, when computing the the dot product of any filter with frequency space, we get an approximation of energy in that filter bank [21, 22].

Rather than producing filter banks based on the linear Hertz scale, the frequency axis of the signal is transformed into units of *Mels*, which is used to account for the non-linearity in human pitch perception [22, 8, 17]. Filter banks are produced to be evenly spaced on the Mel scale, and then transformed back into the Hertz scale. This has the effect of producing triangular filter banks with grow in width as the frequency increases. The Hertz to Mel and Mel to Hertz transforms are given [22, 8]:

$$M_f[h] = 2595 \log_{10}\left(1 + \frac{h}{700}\right) \tag{62}$$

$$H_f[m] = 700\left(10^{(\frac{m}{2595})} - 1\right) \tag{63}$$

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.

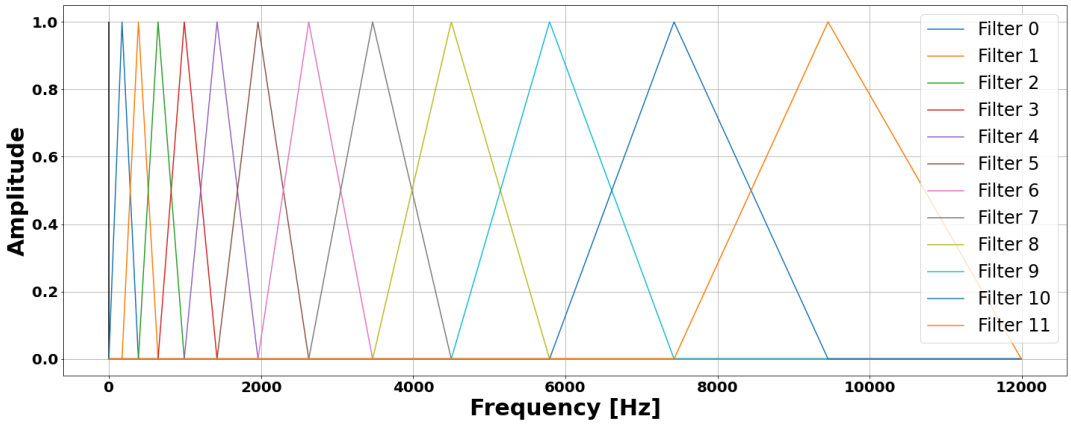


Figure 23: Mel Filter Banks shown in frequency space with units of Hertz

Each of the R filters is created to be N' samples long, to match the width of the cropped frequency space in the spectrogram, Eq. (55). 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. Each filter is concatenated into a matrix M of shape $R \times N'$, where each row is one filter. We apply the Mel Filter banks to the spectrogram to create matrix B:

$$B = (MS)^T \tag{64}$$

Matrix B has shape $k \times R$.

The matrix product in Eq. (64) 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, into array $\widetilde{B}$ with shape $1 \times R$. For this project, we have chosen to use $R = 12$ filter-banks, which are all are used to compute the MFCC's in the next section.

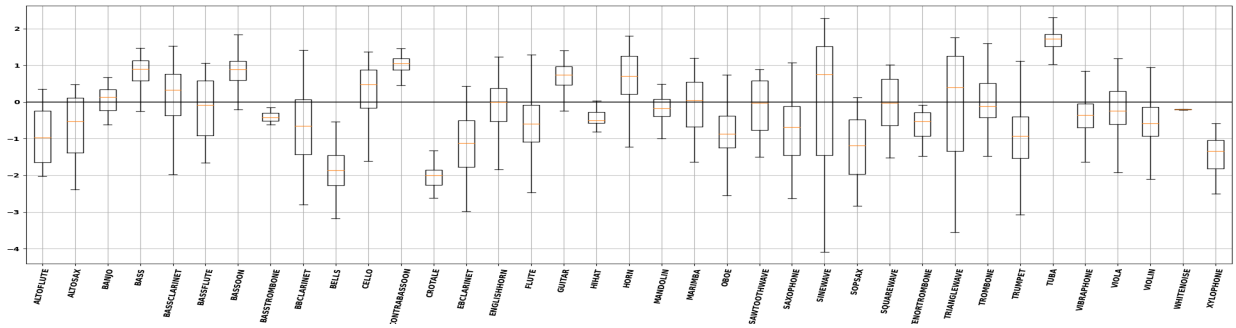
4.4.2 Mel Frequency Cepstral Coeffecients

Ceptstral coefficients are the result of computing the inverse discrete Fourier transform (IDFT) of the logarithm of the frequency Spectrum [22, 21]. Mel Frequency Cepstral Coefficients (MFCC's) are the most commonly used ceptral coefficients and appear commonly in digital signal processing. The c-th coefficient is given by:

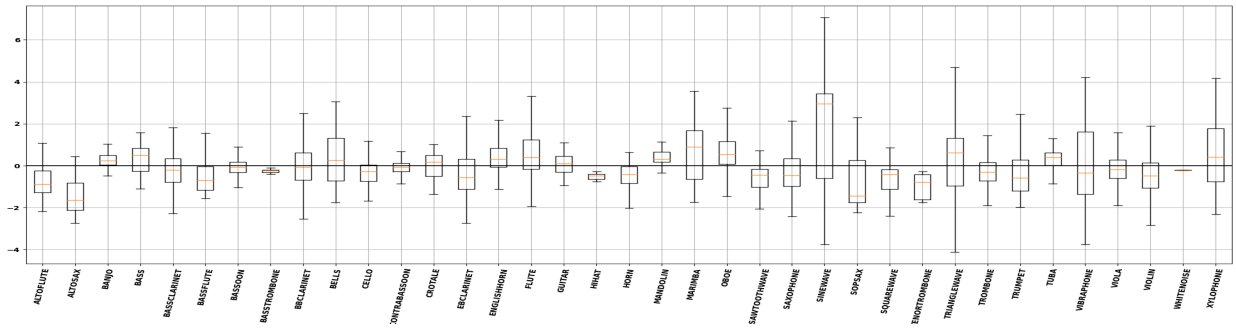
$$\text{MFCC}[c] = \sqrt{\frac{2}{R}} \sum_{i=1}^{R} \log\left(\widetilde{B[i]}\right) \cos\left(\frac{c(i-\frac{1}{2})\pi}{R[i]}\right) \tag{65}$$

Where $\widetilde{B}$ is the column average of the Mel filter bank energies computed in Eq. (64), and R is the number of filter banks used.

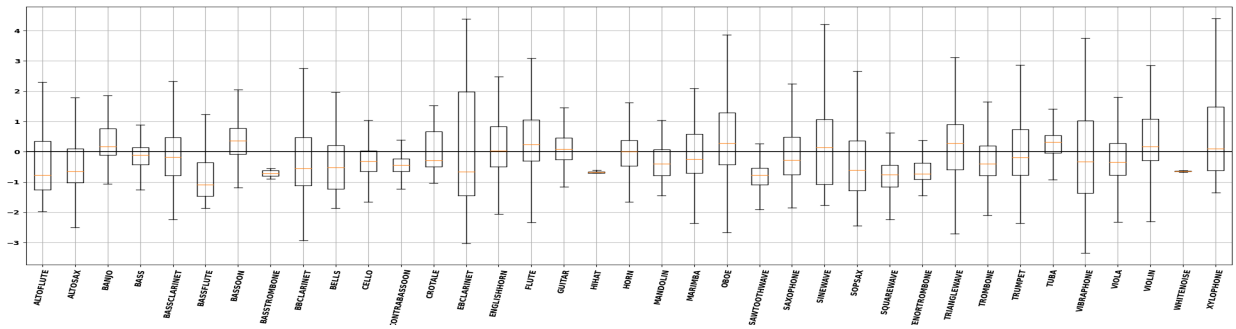
Physically, MFCC's are a transform of a transform. This allows us to investigate the periodicity of the frequency spectrum, which highlights phenomena such as overtones or echoes in the signal [26]. Cepstrum coefficients are commonly used to speech identification and are very prolific in sound recognition tasks [22, 21, 11]. We show a feature-space representation of MFCC's for a selection of coefficients in Fig. (24).



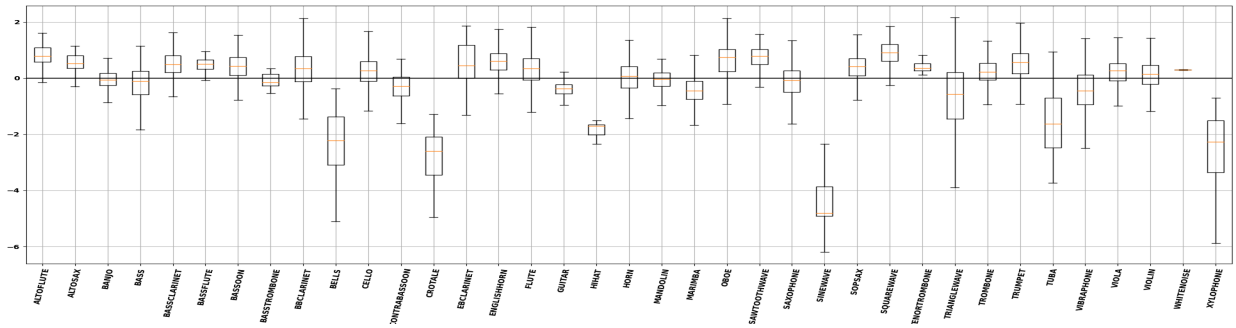
(a) MFCC $R[1]$



(b) MFCC $R[4]$



(c) MFCC $R[8]$



(d) MFCC $R[12]$

Figure 24: A comparison of 4 Mel Frequency Cepstral Coefficients, 1, 4, 8 and 12 for each class using box-and-whisker plots

4.4.3 Frequency Center of Mass

The frequency center-of-mass (FCM) for an audio file provides a 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:

$$\text{FCM}_i\left[s^{(i)}\right] = \frac{\sum_{j=0}^{N'-1} j s^{(i)}[j]}{\sum_{j=0}^{N'-1} s^{(i)}[j]} \tag{66}$$

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:

$$FCM = \frac{1}{k'} \sum_{i=0}^{k'} \text{FCM}_i\left[s^{(i)}\right] \tag{67}$$

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 [17, 27].

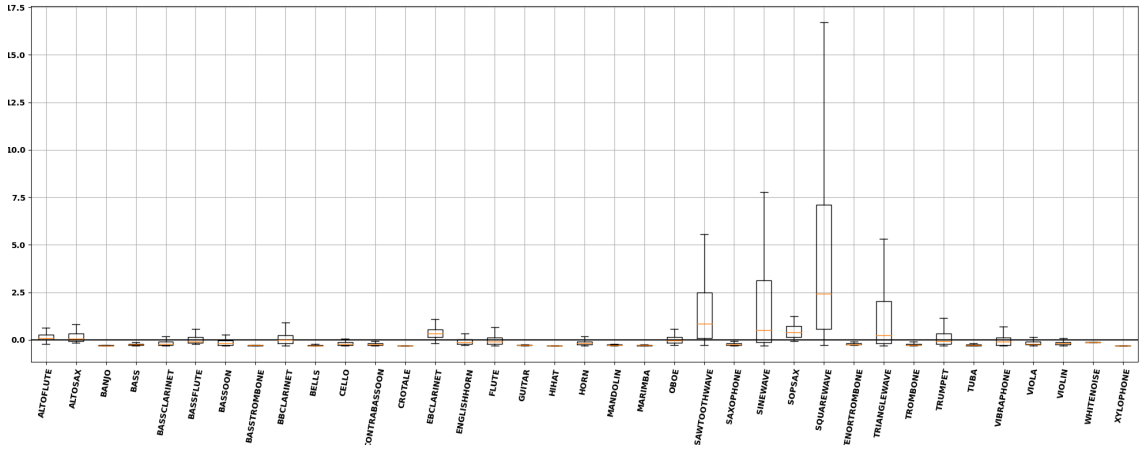


Figure 25: A comparison of the frequency center-of-mass for each class using box-and-whisker plots

5 Evaluating Model Performance

Before making predictions on unlabeled data such as the Chaotic Synthesizers, we must confirm that our model performs reasonably well 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 [2]. As the names imply, the training subset is used to fit the model, and we use the labeled testing data set to evaluate how well the model has trained. The exact ratio of sizes between these subsets varies depending on the task [4, 3, 15], however we choose to generally use around 90% training samples and 10% testing samples. By doing this train-test split scheme, we are confirming or disproving the predictive power of the model. If the model score high within it's training subset, but low within the testing subset, this can indicate a model that is *overfitted*, and will not perform well on other unseen samples [2].

5.1 K-Folds Cross Validation

We can expand on the idea of a train/test split with a resampling method called *K-Folds Cross Validation* (Also called X-validation) [2, 7]. Suppose we have machine learning model F^*, with a set of trainable parameters Θ. We also have a data set X and a set of appropriately labeled targets, Y, each of which contain N samples. For K-folds X-val, we divide the full data set into K subsets (called *folds*), each with size N/K:

$$X, Y \rightarrow \left\{ \left(X^{(0)}, Y^{(0)}\right), \left(X^{(1)}, Y^{(1)}\right), ..., \left(X^{(K-1)}, Y^{(K-1)}\right) \right\} \tag{68}$$

For each iteration, $k \in [0, 1, 2, ..., K-1]$ we reserve a single subset of data $\left(X^{(k)}, Y^{(k)}\right)$ to use as a testing subset. We use the remaining $K-1$ subsets as training data. We fit the model with the data and labels to produce a model $F^{*(k)}$ which as set of parameters Θ^k. We return the the test set, and run predictions on this unseen data subset. We compare those predictions to the corresponding set of labels $Y^{(k)}$ and evaluate the result of any selected performance metrics.

The output of cross validation is K models that are all trained and evaluated on overlapping subsets of the full data. Each model F^*, shows a possible outcome of training the network given a subset of data samples, and a particular set of initial parameters. It has a corresponding set of learned parameters Θ^*, as defined in Eq. (21) that it uses to make any predictions. We can compare the performance of each model and test if they produce similar outcomes, which indicates that the model will be able to consistently generalize to new, unseen samples.

Algorithm 7 A K-Fold Cross Validation program.

Require: Untrained Network or related learning algorithm, F^*
Require: A full labeled data set (X, Y) of N samples
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 N/K samples
  $X, Y \rightarrow \left\{ \left(X^{(0)}, Y^{(0)}\right), \left(X^{(1)}, Y^{(1)}\right), ..., \left(X^{(K-1)}, Y^{(K-1)}\right) \right\}$
  Performance History $\leftarrow \{\}$
  for $k = 0, 1, 2, 3, ..., K-2, K-1$ **do**
    Reset all parameters in F^* to a random "untrained" state
    Set aside testing data subset
    $X_{test} \leftarrow X^{(k)}$
    $Y_{test} \leftarrow Y^{(k)}$
    Concatenate the remaining subsets into training data set
    $X_{train} \leftarrow X^{(i \neq k)}$
    $Y_{train} \leftarrow Y^{(i \neq k)}$
    Train the model, F^* with the X_{train} and Y_{train} subset.
    Evaluate the trained model with the X_{test} and Y_{test} data set, and compute value of metric function(s) P
    Store Performance P in Performance History array
  end for
  Compare performance results, and adjust model or parameters as needed, and repeat if desired.
  Run additional analysis on performance metrics.

Cross Validation is particularly useful in models such as neural networks because they suffer from the phenomenon of *high-variance*, which means that small changes in initial conditions can drastically change the outcome of the model [7]. Cross validation allows us to control these initial conditions by training the model on similar, but non-identical subsets of data, along with a slightly different set of initial parameters. This repetition allows us to explore the model's behavior over a range of possible initial conditions and validation sets, to ensure that the network is functions as expected over multiple trials. This also eliminates the fear of "unique" cases where the model happens to perform exceptionally well or exceptionally poorly given a random set of initial parameters [2].

5.2 Performance Metrics

In the case of the multi-category classifier, it is important that we choose the appropriate performance metrics to confirm that the network is completeing it's assigned as as expected [2]. While the neural network itself uses the cost function as it's sole objective to optimize, we also require a set of more human-readable functions. For example, an average loss score of 1.075 over a given subset of previously unseen samples provides us with no real information

as to how well the network is performing at it's designed task. In this section, we introduce a set of functions and metrics than enable a more tangible interpretation of the model's performance. To evaluate any performance metric, we require a set of samples with ground truth labels, y, and a model's prediction for those labels, y^* [4, 7].

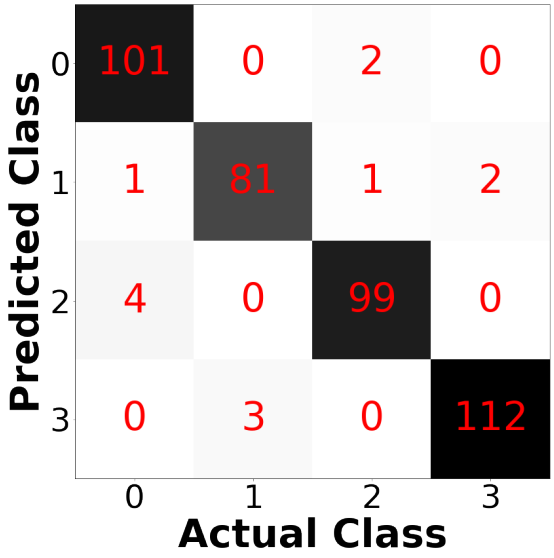
5.2.1 Confusion Matrix

The *standard confusion matrix* (also called a confusion table) is a very quick, often graphical model that can be used to show how a classifier 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 [2]. If we see that these classes are being repeatedly *confused* in the model's prediction process, then we can modify the model or features to account for it.

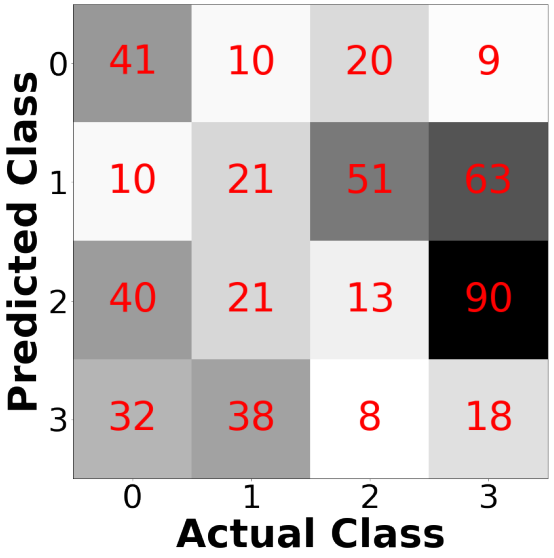
For a k-classes classifier, a confusion matrix will have shape $k \times k$, and every element is a non-negative integer. Each row represents the "ground truth" or labeled class, and each column represents a predicted class. Thus for a confusion matrix, C, we can say that:

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

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 predicts correct labels [2]. Below we present some synthetic confusion matrices, which combine the counts in each index and a corresponding color map.



(a) Diagonal dominance indicates a stronger classifier



(b) No diagonal dominance indicates a weaker classifier

Figure 26: Example Confusion Matrices for 4-categories classifier

In addition to the standard confusion matrix, we can also weight entries by prediction score. Rather than adding +1 to each entry $C_{i,j}$ where appropriate, we instead add the *probability* value, which is bounded $[0, 1]$ (See softmax activation function in sec. 2.4.2)). This allows us to bake in the predictive *confidence* into the confusion matrix. For example, two correct predictions with a score 0.2 are weighted equally to a single incorrect prediction of 0.4. This is useful for identifying a classifier that makes correct predictions, but only by smaller threshold. This can almost be through of as assigning "partial credit" to each prediction.

However, it is more useful to weight the confusion matrix according to class occurrence. For example, if we had a 2×2 matrix with entries 100 and 20 in the main diagonal, it seems that class A has a much higher prediction success than class B. However, if there are 200 elements in class A, and only 21 in class B, then we see that class A has a 50% classification rate, while class B has a 95% classification rate. Consider the list of instrument classes in Fig.(7) and notice how all instruments are not equally represented. This creates a training bias such that the model is more heavily trained on a particular class compared to others. This may also show when making predictions, where the classifier is more likely to predict samples that it has been trained on more [7, 12]. To Combat this, we typically normalize each confusion matrix by the number of samples in each "actual" class. Put simply, we divide each element in a given row by the *sum* of that row.

The confusion matrix can be a cumbersome, so often it is useful to express the performance in terms of more concise quantities. For this we use *accuracy score*, *precision score*, *F1-score*, and *recall score*. These metrics are standard in the field of machine-learning classification and are often useful for identifying different strengths and weaknesses in each model [2, 7].

5.2.2 Accuracy Score

Accuracy score, while not commonly used is the most intuitive of all of the performance metrics. It is the ratio of correct predictions of he total number of predictions. While immediately useful for equally sampled binary classification problems, it looses meaning as the number of classes increases, and epecially if the number of samples per class is not consistent [2].

For example, we could develop a model that uses a few predictors from every human currently alive and determine which of them has walked on the moon. There are roughly 7 billion humans alive, four of which have been on out moon. Simply guess *no* for every human, our model would have more than a 99.9999% accuracy. However, we would argue that this model does not perform well seeing as it has a 0% recall score, and and a 0% precision score.

Accuracy score of model is defined:

$$\text{Accuracy} = \frac{TP + FN}{TP + FP + FN + FP} \tag{69}$$

For a multi-class problem, present accuracy over each individual class, called *micro-accuracy* [7]. The accuracy for a class j is the correct predictions related to the class divided by the total predictions and total counts of the class. We express this mathematically by using sums over each row, each column, and the main diagonal as such:

$$\text{Accuracy}_j = \frac{\sum_i C_{i,i}}{\sum_i C_{i,i} + \sum_{i \neq j} C_{i,j} + \sum_{i \neq j} C_{j,i}} \tag{70}$$

Where all sum indexes goes from 0 to $k-1$.

We can also present *global accuracy* or *macro accuracy* by computing the acurracy over the full confusion matrix. This would then by the ratio of all correct predictions (the main diagonal) to all predictions (the sum of the matrix). We can express this as:

$$\text{Accuracy} = \frac{\sum_i C_{i,i}}{\sum_i \sum_j C_{i,j}} \tag{71}$$

5.2.3 Precision Score

Precision score (also called *specificity* or *positive predictive value*) is the ratio of chosen elements to all relevant elements. This bounds precision to the range $(0, 1)$, with a higher value more desirable. For a classifier with $k = 2$ unique classes, we define the precision score of a model as:

$$\text{Prec} = \frac{TP}{TP + FP} \tag{72}$$

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

$$\text{Prec}_j = \frac{C_{j,j}}{\sum_{i=0}^{k-1} C_{j,i}} \tag{73}$$

In the case of a multi-class classifier, the precision score represents an *one-vs-all* measurement. This means that for any class j, the sample belongs to class j or it does not. All other classes, $i \neq j$ are temporarily considered to be a single aggregated class. The quantity $TP + FP$, or the sum over the confusion matrix row is a measurement of the total number of items that are predicted to be in the given class. Therefore the precision metric answers the question: *"How many selected items are relevant to the problem?"* [2, 7].

5.2.4 Recall Score

Recall score (also called *sensitivity* or *true positive rate*) 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:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{74}$$

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 of a class j is given by the entry $C_{j,j}$ divided by the sum of column j:

$$\text{Recall}_j = \frac{C_{j,j}}{\sum_{i=0}^{k-1} C_{i,j}} \tag{75}$$

In the case of a multi-class classifier, the recall score also represents an*one-vs-all* measurement. This means that for any class j, the sample belongs to class j or it does not. All other classes, $i \neq j$ are again temporarily considered to be a single aggregated class. The quantity $TP+FN$, or the sum over the confusion matrix coumn is a measurement of the total number of items that are actually in the given class. Therefore the recall metric answers the question: *"How many relevant items to the problem are selected?"* [2, 7].

5.2.5 F1-Score

F1-score (also called F-score) is the harmonic mean of precision and recall scores and it also bounded on $[0, 1]$ with a higher score being favorable [2]. Often, the two metrics can be through of as somewhat *exclusive* to each other - the higher one, the lower the other. Compare this with the idea that the more *sensitive* the model is, the less *specific* it is, and vice-versa. This phenomenon is called the *precision-recall tradeoff* [2, 7]. We compute the $F1$ score for a class k as:

$$\text{F1}_j = 2 \times \frac{\text{Prec}_j + \text{Recall}_j}{\text{Prec}_j \times \text{Recall}_j} \tag{76}$$

The $F1$ favors models with both a high precision and high recall score. Some models allow for the ability to adjust the threshold of the decision function. This means that by changing a few parameters that are external to the classifier itself, we can change how sensitive or specific a model is when making it's class decision. In the case of this multi-category classifier, this is somewhat like modifying the generated decision boundaries as to change the outputted probability distribution. In case cases, it is useful to use the $F1$ score to find the set of hyper-parameters that allow for the classifier to produce both the highest possible precision *and* recall score.

5.3 Tracking Metrics over a Period of Training

A period of training is characterized by fitting the parameters Θ of a model F^* to a set of data X and corresponding labels Y [4, 26]. This is done by passing subsets of data, called *mini-batches*, into the neural network for training. The average cost for the mini-batch is then used to compute the gradient vector $\nabla_\Theta J$, and subsequently update the model according to the optimizer chosen [2, 4].

We execute computation in batches to reduce the amount of memory required for the operation. Pushing a full data set through a model at one time would require more RAM

than is usually available , thus repeated subset of data tend to avoid memory exhaustion errors. Additionally, every mini-batch used equates to one step in the optimizer update rule [4]. Thus, a single pass over a full data set allows for multiple iterations of the optimizer to reduce the value of the cost function. This can be combined with multiple passes over the full data set, called *epochs* [7, 12].

Large mini-batches require lots of RAM, but prevent the model from being over-fit to to any one sample, or class of samples. Small mini-batches require less RAM, but often may bias the optimizer to over-fit the given samples, and make optimization unstable [2, 7]. To ensure that the model is optimizing properly, we can track how the metrics behave over a training period. This allows us to monitor the *rate of convergence* of the model. If the cost function is dropping to quickly or slowly, this may indicate that the optimizer learning rate is too high or too low. This may lead to an over-fit or under-fit model, or indicate an inappropriate set of feature is being used [2, 4]. In Some cases, it can be used to initiate *early-stopping*, which halts training when a set of parameters are met.

Given the high dimensionality of this model's parameter space,the large volume of sound files, and the large amount of RAM required to store the spectrogram matrix and the feature vector for each sample, we do not directly load the full data set into memory at once. Instead, we produce a large subset of the full data set of 256 samples which we dub a *mega-batch*. From this mega-batch, we produce the two deign matrices required for input, and load the corresponding labels. We then use a subset of this group as the *mini-batches* for training. Each mini-batch contains 32 samples. Once all mini-batches are fit, we discard the design matrices in the mega batch and repeat for the next 256 samples.

With each training step, we have recorded the precision, recall, and loss scores. These scores are computed from the forward pass of the training data before the gradient is computed, and parameters updated, i.e. the program has not seen Our program stores these values locally in a *training-history* file, which we can examine after the program completes. Below, we visualize the evolution of each score as training progresses. These plots are from a model that was trained separately on the full data set after cross validation was performed.

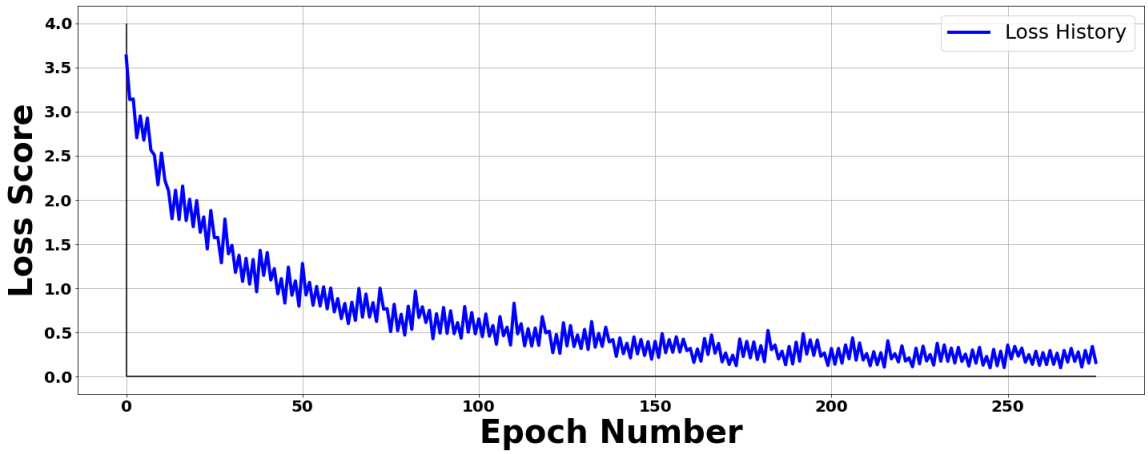


Figure 27: The loss function score decreases with each training step, indicating that optimization is performing correctly

similarly, we can visualize the precision score and recall score at each training step.

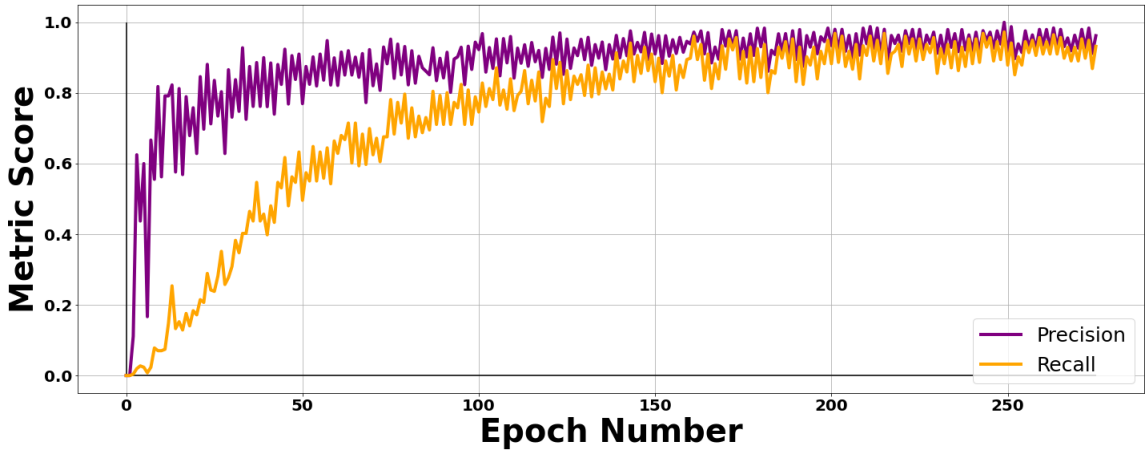


Figure 28: The precision score and recall score generally increase with each training step

Notice how loss, precision and recall show a pattern of increasing and decreasing every two steps. This is because we extract a mega-batch of 256 samples, and perform two epochs of training on the batch. The first step represents the first time the model has seen the data, thus produces a higher cost function value. In the second step, the model has already seen the samples in the batch, and it produces a slightly lower cost value that the first step. The batch is discarded and a new subset of samples is drawn so that the process repeats. With two epochs per mega-batch, and two passes over the full data set (permuted in the middle), the model is effectively trained on the full data set a total of 4 times.

6 Experimental Results

6.1 Executing Cross Validation

To formally produce classification predictions on the chaotic synthesizer waveforms, we must use all of the neural network, physics, and statistical principles outlined in the previous sections. Using the described features in sec. (4) and appropriate architecture sec. (2.6), We run a $K = 10$ folds cross validation program as in Alg.(7). For each of the 10 models, we produce a standard confusion matrix, and compute the (i) accuracy score, (ii) precision score, (iii) recall score, and (iv) F1 score. In Fig. (29) we show how the metrics compare in each the 10 models by averaging the scores across the 37 classes.

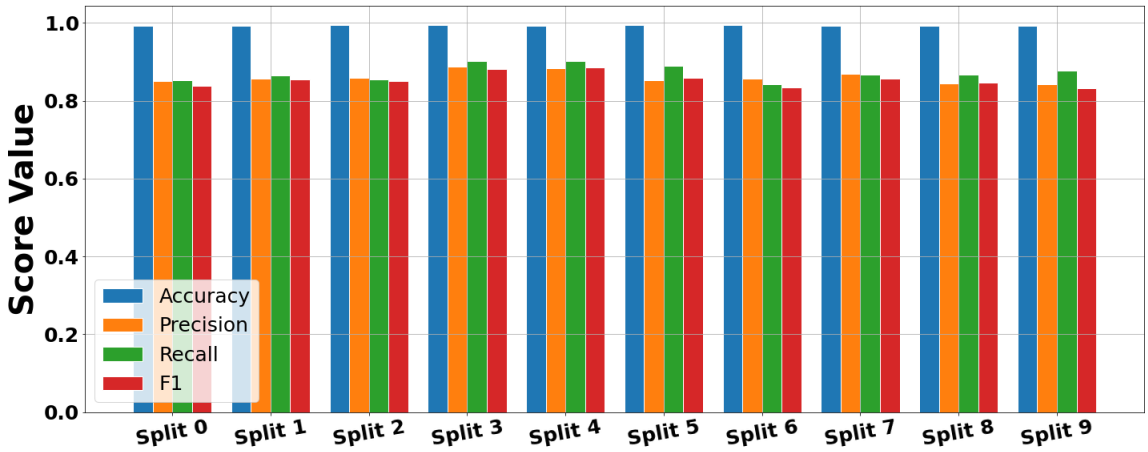
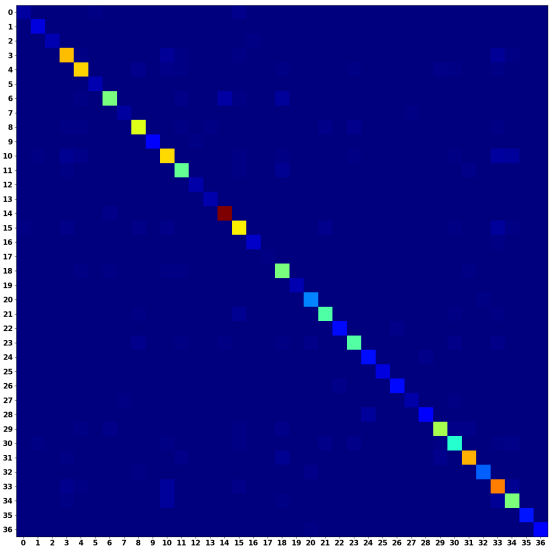
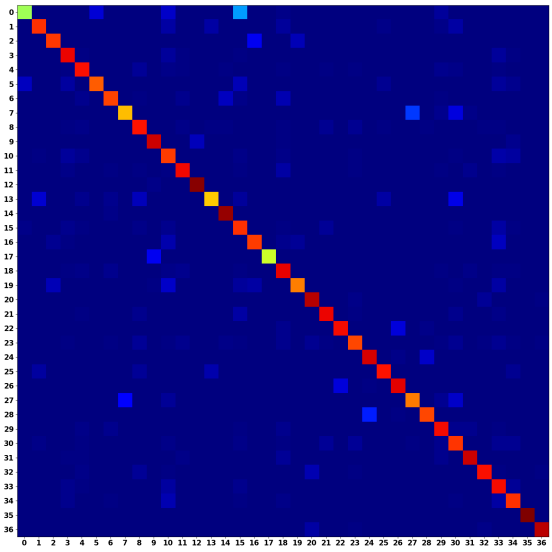


Figure 29: Performance metrics for the multimodal networks across 10 models, scores are averaged over 37 classes

Additionally, we present the average standard confusion matrix, and the hits-weighted confusion matrix in Fig. (30).



(a) Standard confusion matrix for hybrid network



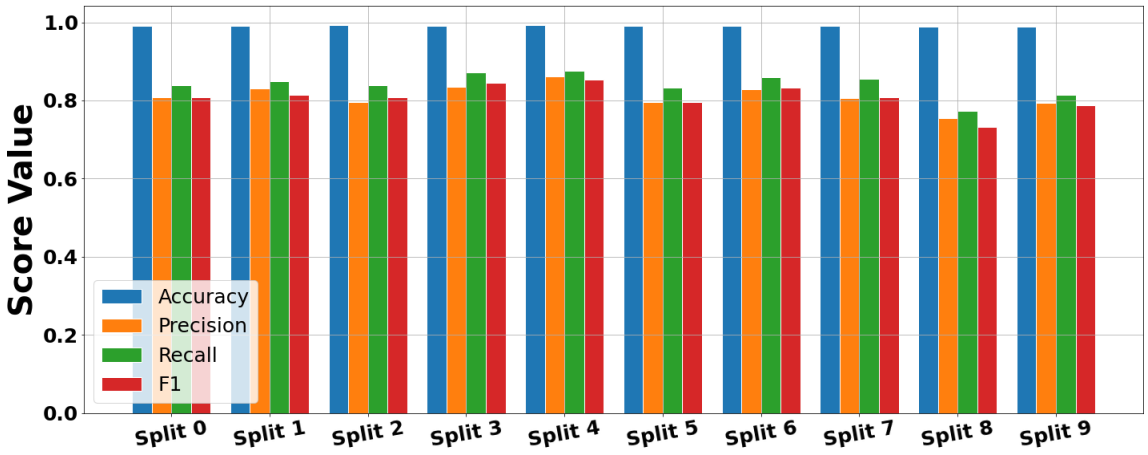
(b) Occurrence weighted confusion matrix for hybrid network

Figure 30: Confusion matrices, each is averaged over 10 folds of cross validation

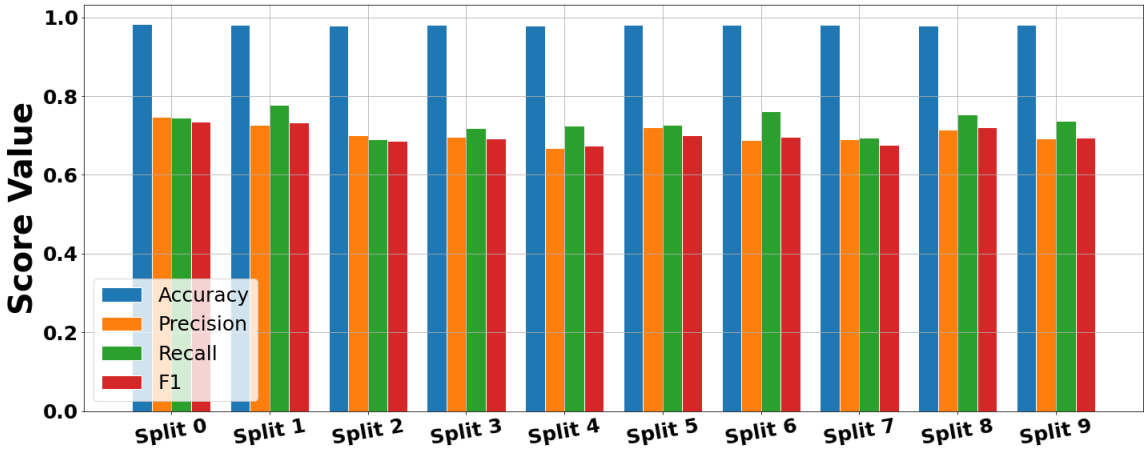
We show the metric values of all splits to show the consistency in performance across all subsets. Notice how each split shows reasonable performance as given by the precision and recall score consistently above 80%. This indicates that for this data set, architecture, and feature selection, our model can accurately classify waveforms and generalize appropriately. This step is critical in ensuring that there are no major out-liers in the classification results, and that any set of initial conditions can still allow the model to reach an acceptable set of parameters and decision boundaries. The results of cross validation indicate a strong classier that can be implemented to make predictions on other waveforms.

6.2 Comparing Results between Architectures

In addition to considering our multimodal network architecture performance, we have also run the identical $K = 10$ folds cross validation program on two uni-modal variants of the model. The first variant contains only the Convolutional branch of the network in Fig. (6), meaning that we feed the activations from the last dense layer in the left column directly to the output layer. Similarly, the second variant contains only the perceptron branch of the network in Fig. (6), meaning that we feed the activations from the last dense later in the right column directly to the output layer. For consistency, the same full data set (around $18,000$ samples) was using in the cross validation program. Comparing the performance of the each model gives us an insight as to the predictive contribution of each input branch



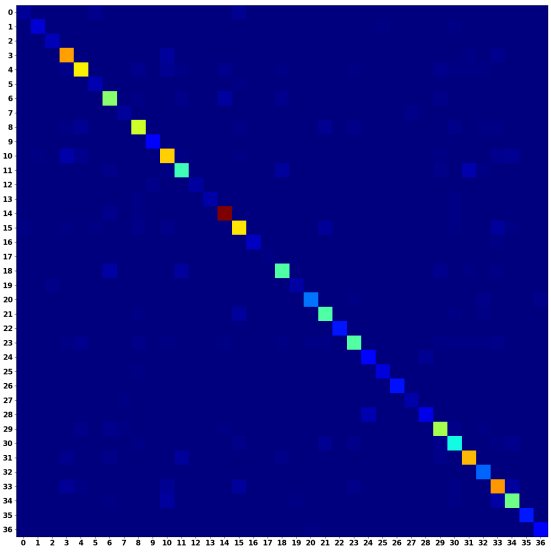
(a) Convolutional Unimodal Architecture, See left side of Fig. (6)



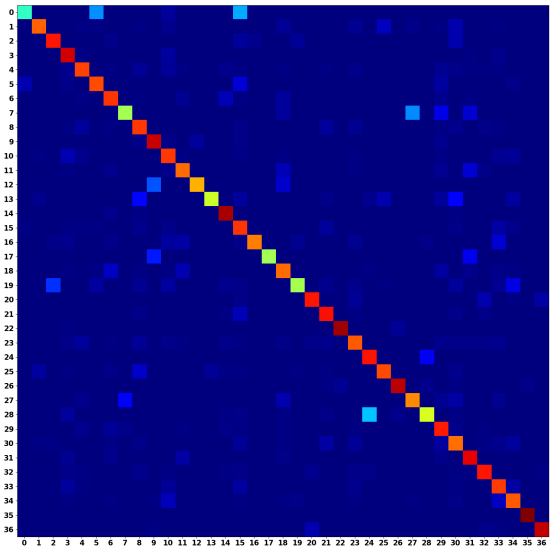
(b) Perceptron Unimodal Architecture, See right side of Fig. (6)

Figure 31

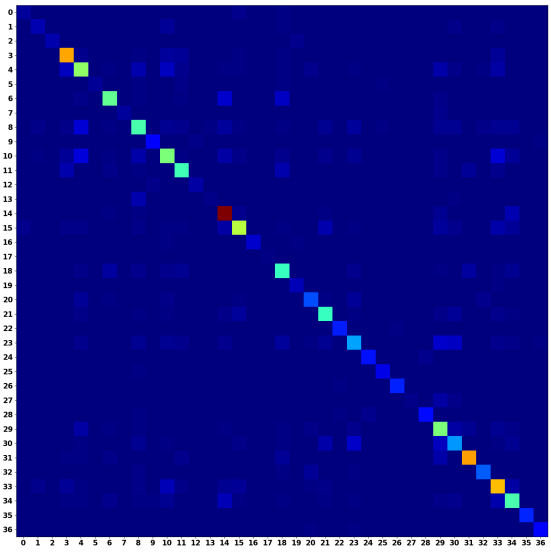
Compare these unimodal model performances with that of the multimodal network results in Fig. (29). Notice how the multimodal network consistently produce superior classification results. Similarly, we present the standard and weighted confusion matrices for both unimodal networks. Compare the results with Fig. (30) and see how the confusion matrix indicates a stronger classifier.



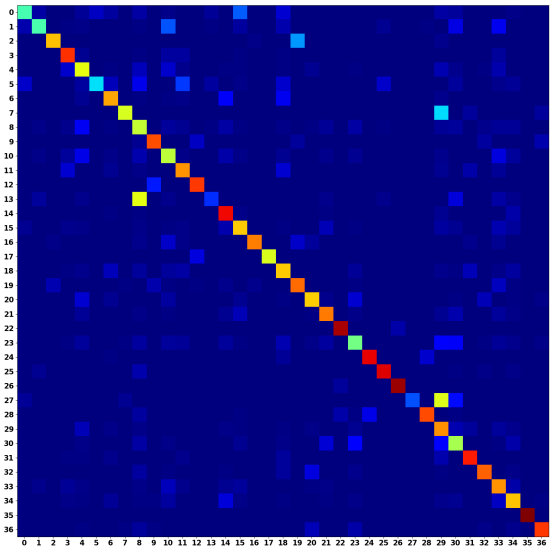
(a) Standard confusion Matrix for the convolution branch



(b) Occurrence Weighted confusion matrix for the convolution branch



(c) Standard confusion matrix for the perceptron Branch



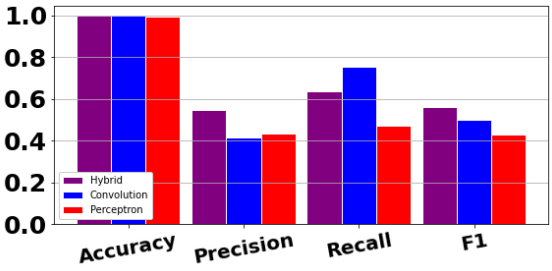
(d) Occurrence weighted confusion matrix the for perceptron branch

Figure 32: Performance metrics for the unimodal networks across 10 models, scores are averaged over 37 classes

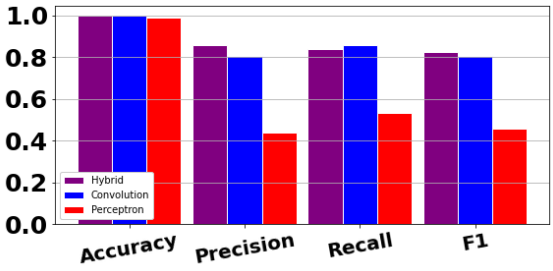
6.3 Comparing Classification Scores within Each Class

Below we present the average accuracy, precision, recall and F1 score for each class, as averaged across 10-folds cross validation, comparing the three architectures. These plots represent a more "micro" representation of classification by showing the response and scores within each specific class.

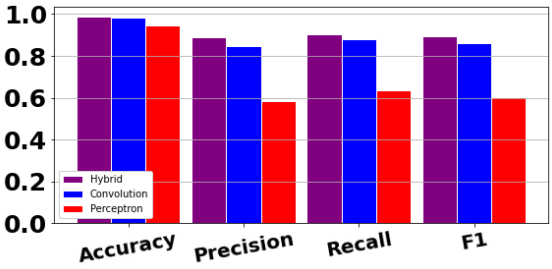
6.3.1 High Woodwind Scores



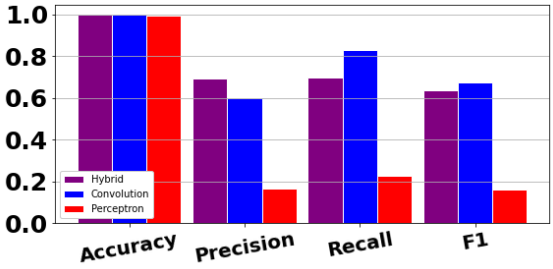
(a) Alto Flute



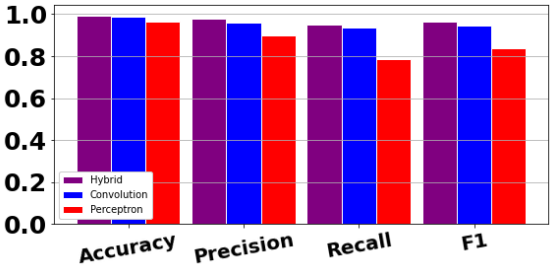
(b) Alto Saxophone



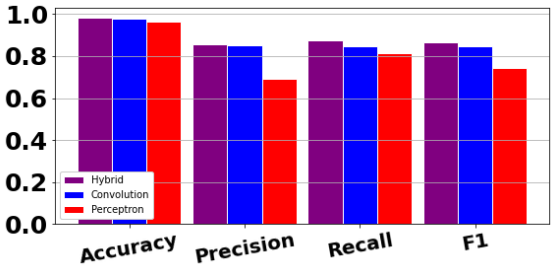
(c) $B\flat$ Clarinet



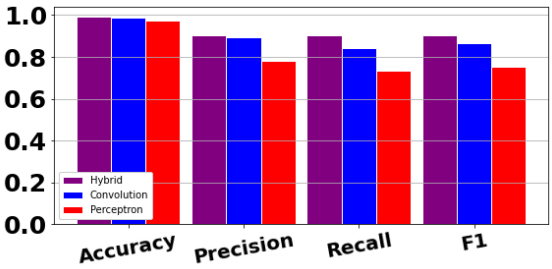
(d) $E\flat$ Clarinet



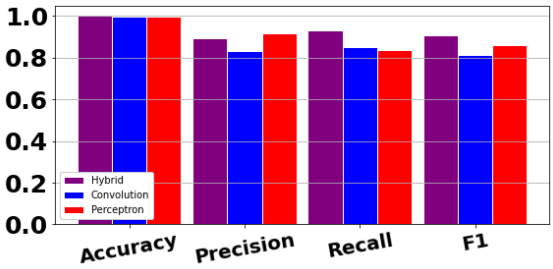
(e) English Horn



(f) Flute



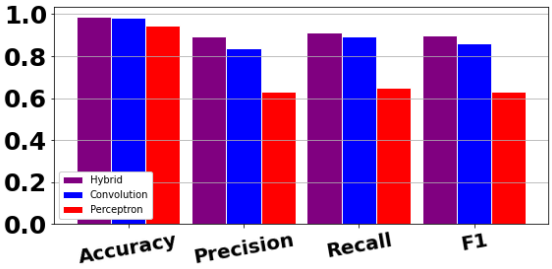
(g) Oboe



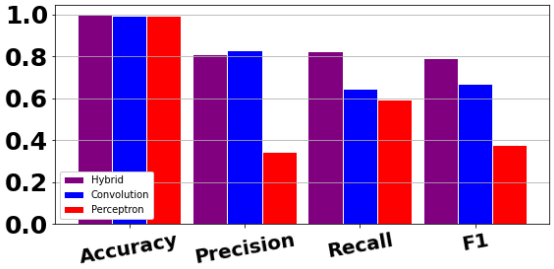
(h) Soprano Saxophone

Figure 33

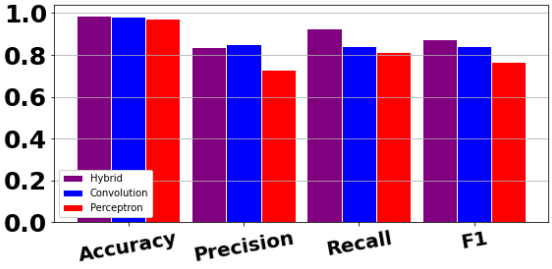
6.3.2 Middle and Low Woodwind Scores



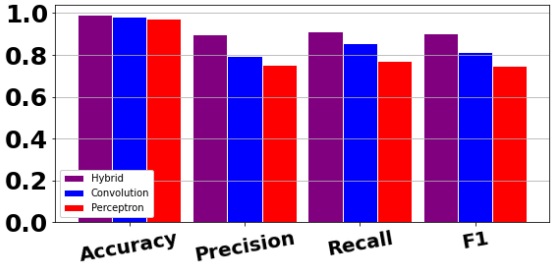
(a) Bass Clarinet



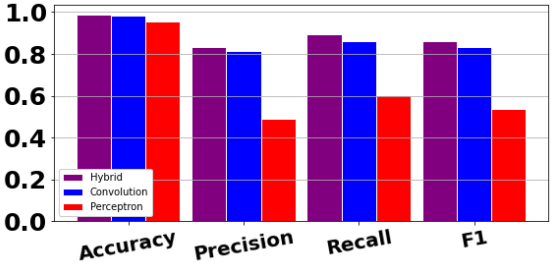
(b) Bass Flute



(c) Bassoon



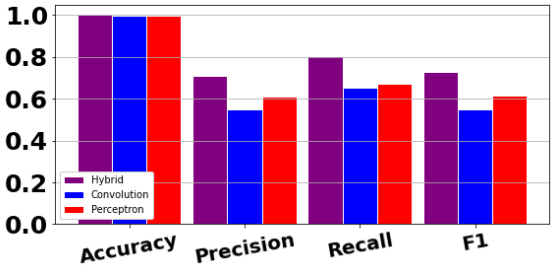
(d) Contra Bassoon



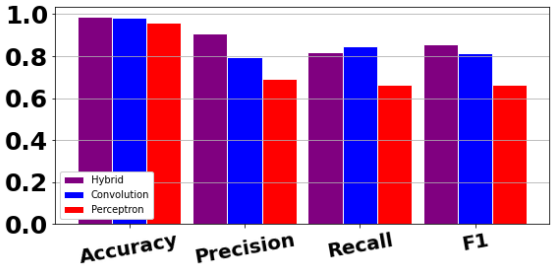
(e) Tenor Saxophone

Figure 34

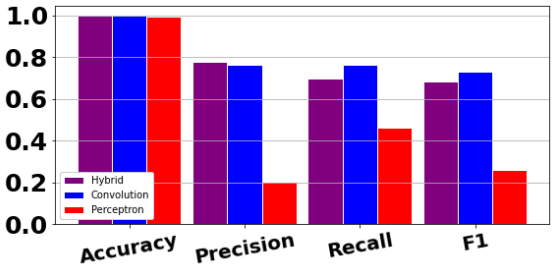
6.3.3 Brass Scores



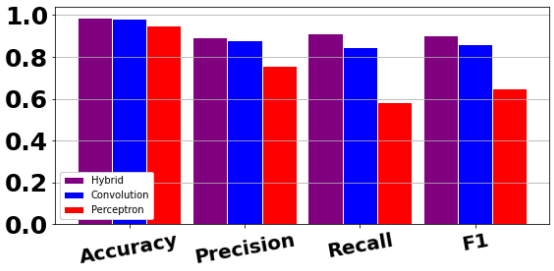
(a) Bass Trombone



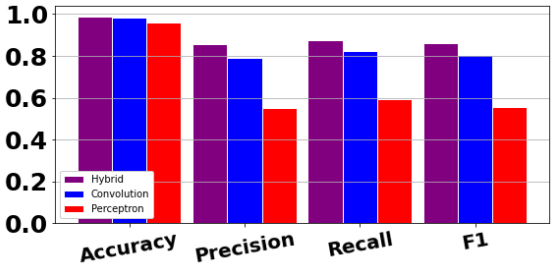
(b) French Horn



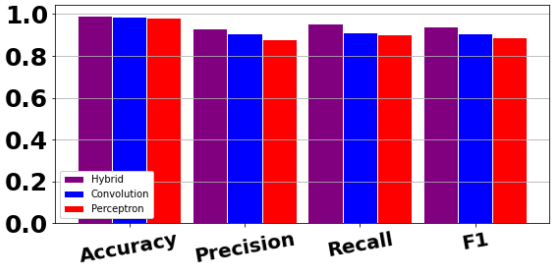
(c) Tenor Trombone



(d) Trombone



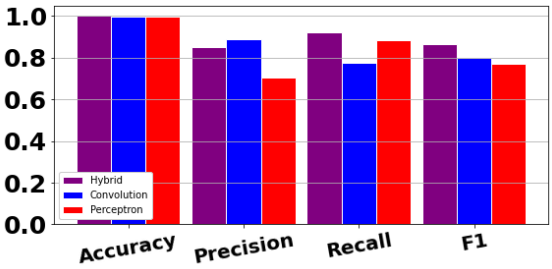
(e) $B\flat$ Trumpet



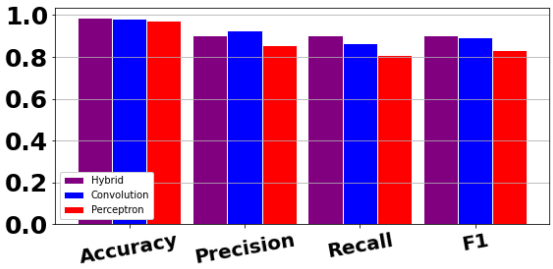
(f) Tuba

Figure 35

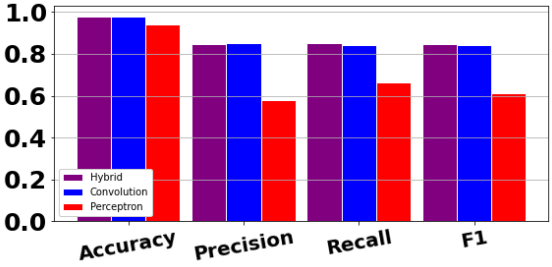
6.3.4 String Scores



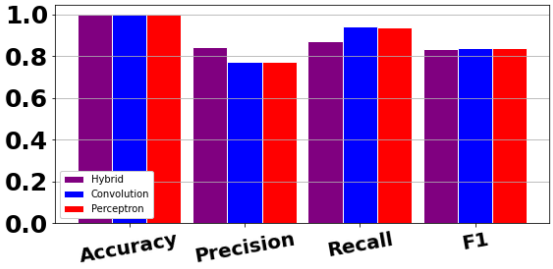
(a) Banjo



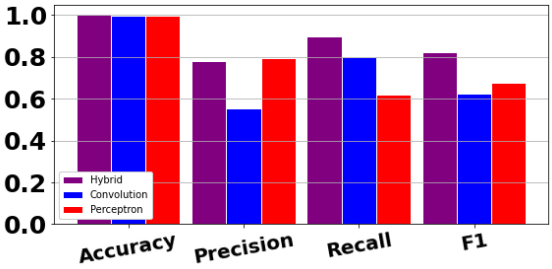
(b) Double Bass



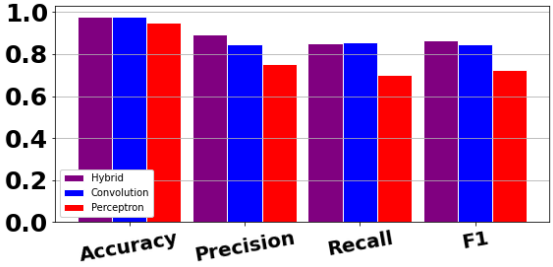
(c) Violoncello



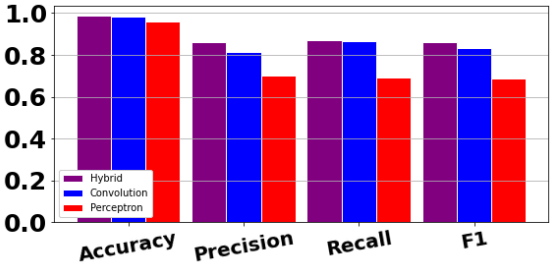
(d) Guitar



(e) Mandolin



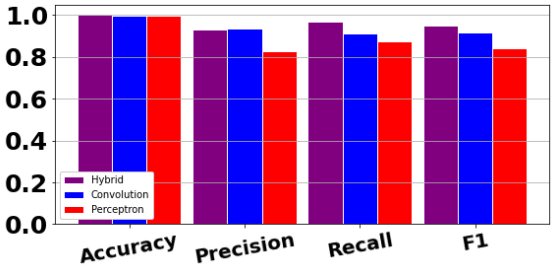
(f) Viola



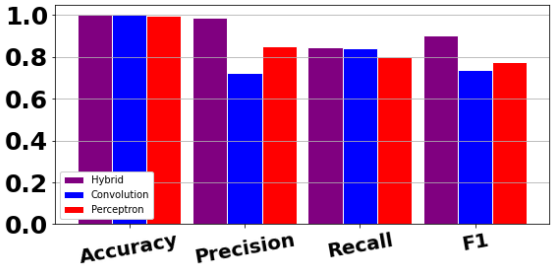
(g) Violin

Figure 36

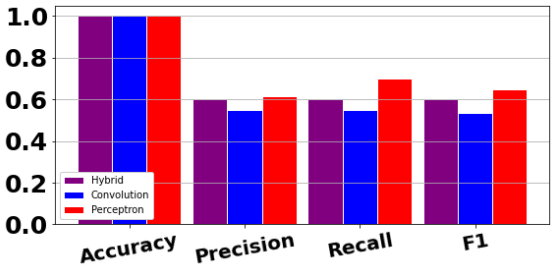
6.3.5 Percussion Scores



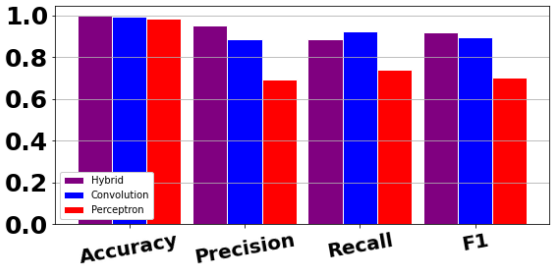
(a) Bells



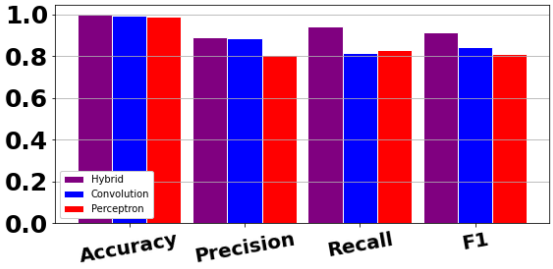
(b) Crotales



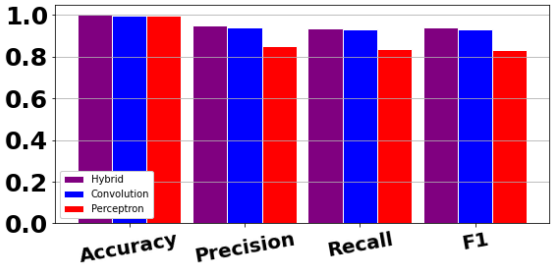
(c) HiHat



(d) Marimba



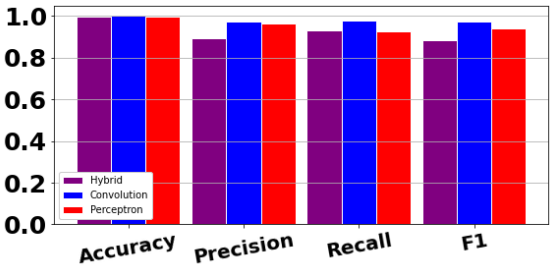
(e) Vibraphone



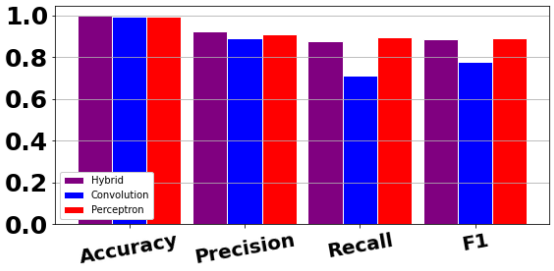
(f) Xylophone

Figure 37

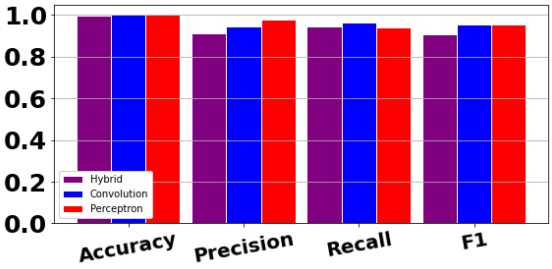
6.3.6 Synthetic Waveform Scores



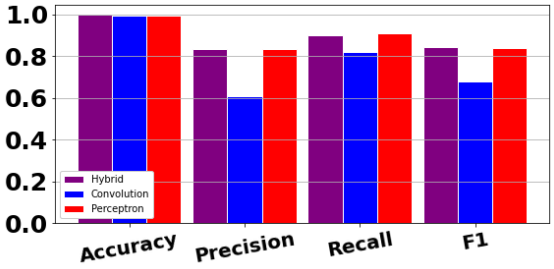
(a) Sawtooth Wave



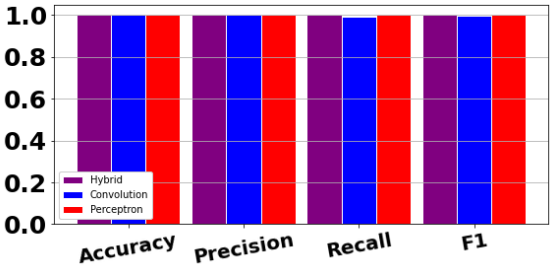
(b) Sine Wave



(c) Square Wave



(d) Triangle Wave



(e) White Noise

Figure 38

6.4 Discussion of Results

Comparing the results within each class, as averaged over 10 folds of cross validation indicate that our model performs very well at the ability to a match sound waves to their sources. We see this behavior in the plots in sec. (6.3) as well as the confusion matrices in Fig. (30). For the chosen set of classes, features and architecture, the model can generalize to new information allowing it to distinguish between different type of musical instruments reasonably well. Additionally, we show that predictive power of the model improves through our development of the multimodal architecture. By choosing to express the waveform in two different modalities in the form of the spectrogram matrix and the feature vector, the hybrid model demonstrates a generally higher classification performance across the same number of samples and splits.

Despite the higher average scores in the hybrid network, see Fig. (29) compared to each unimodal network, Fig. (31), some exceptions do arise at a micro-level (within each class). Consider the Hi-hat instrument in Fig. (37c).

6.5 Predictions on Chaotic Synthesizers

7 Conclusion

8 Acknowledgments

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

- Dr. Kevin Short and Dr. Maurik Holtrop for various consultation and guidance throughout this project.
- 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. Kourosh Zarringhalam and Dr. Marek Petrik for additional consultation on machine learning topics.
- University of Iowa, Electronic Music Studio and Philharmonica Symphony Orchestra for the digital sound library used for training data samples.

References

[1] Bishop, Christopher M. *Pattern Recognition and Machine Learning*. Springer New York, 2016.

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

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

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

[5] "Wave Equation: Vibrating Strings and Membranes." *Applied Partial Differential Equations with Fourier Series and Boundary Value Problems*, by Richard Haberman, Pearson, 2019, pp. 130–150.

[6] Hunter, Joseph L. *Acoutics*. Prentice Hall, 1957.

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

[8] 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.

[9] Levine, Daniel S. *Introduction to Neural and Cognitive Modeling*. 2nd ed., Routledge, 2000.

[10] Li, Yingming, and Ming Yang. "A Survey of Multi-View Representation Learning." Journal of LateX Class Files, vol. 14, no. 8, Aug. 2015.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] Ngiam, Jiquan, et al. "Multimodal Deep Learning." 2011.

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

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

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

[20] Powers, David. (2008). *Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation*. Mach. Learn. Technol. 2.

[21] Sahidullah, Goutam S. "Design, Analysis and Experimental Evaluation of Block Based Transformation in MFCC Computation for Speaker Recognition." 18 Nov. 2011.

[22] Serizel, Roman, et al. "Acoustic Features for Environmental Sound Analysis." Computational Analysis of Sound Scenes and Events, by Tuomas Virtanen, Springer, 2018, pp. 71–101.

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

[24] "Continuum Mechanics." *Classical Mechanics*, by John Robert Taylor, University Science Books, 2005, pp. 681–738.

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

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

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

[28] 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.