MK3 - Musical Instrument Recognition

by

Kenneth Ang
Han Zhang

Department of Electrical and Electronic Engineering

University of Melbourne

Supervisor

Margreta Kuijper

In partial fulfillment of the requirements for the degree of

Master of Electrical Engineering

May 13, 2019

Acknowledgements

This is a Master Capstone report from Department of Electrical and Electronic Engineering in University of Melbourne. We would like to thank Dr. Margreta Kuijper for her invaluable guidance and advise on this capstone journey. We would also like to thank Dr. Tansu Alpacan for his report improvement advice and guidance through the course of the ELEN90088 subject.

Abstract

With the continued exponential growth of the digital age, The importance of providing meaning to data has grown considerably. In our project, we focus on recognizing sources of musical signals embedded within time-series data. Our problem has been divided into two phases.

In the first phase, we performed single instrument recognition using existing machine learning techniques and neural network architectures developed specifically for musical signal processing. In the second phase, we incorporated advanced neural network architectures and techniques developed for applications outside the scope of musical audio processing and apply them to recognize simultaneously playing instruments in an audio file.

Our single instrument recognizer utilized context specific Convolutional Neural Network (CNN) filters and achieved an accuracy score of 98.5% which out performed previously developed models. Our multiple instrument recognizer utilized techniques such as transfer learning and model ensembling to optimize overall performance and borrowed concepts from fields such as Natural Language Processing (NLP) and medical processing to build our final model, which achieved an F1 score of 85.5%. Furthermore, we also introduced a custom loss function designed specifically to train networks to perform multi-labeled classification tasks.

Contents

List of Figures

List of Tables

Acronyms

ADC Analog-to-Digital Converter. 6

ANN Artificial Neural Network. iii, 11–18

API Application Programming Interface. 39, 40, 44, 79

AUC Area Under the Curve. 60

BiRNN Bidirection RNN. 17, 64, 65, 67, 69, 72, 74, 79

BP-MLL Back Propagation Multi-Label Loss. 23, 54, 56, 61, 88

BPTT Back-Propagation through time. 18

CE Cross-Entropy. 22, 53, 54, 61, 62, 88

CNN Convolutional Neural Network. ii, iii, viii, 13, 41, 42, 44, 48, 57, 60, 61, 76, 78, 79, 95

DFT Discrete Fourier Transform. 6, 7

FFT Fast Fourier Transform. 7

MCP McCulloch-Pitts. 11

MIR Music Information Retrieval. 1, 26

ML Machine Learning. 9

MLP Multi Layered Perceptron. iii, 12, 23, 90

NLP Natural Language Processing. ii, 57, 76, 79

PR Precision-Recall. 60

RNN Recurrent Neural Network. iii, 15

SGD Stochastic Gradient Descent. 24, 40, 43, 89

STFT Short Time Fourier Transform. 7, 8

VGG Visual Geometry Group. 33

Chapter 1: Introduction

1.1 Motivation and Context

Mediums producing time-series signals are some of the most widely available sources of stored data. The most common examples of such data include those generated by the business and finance industries such as stock prices or energy production, biosignals generated by the human body and musical audio recordings for market consumption. As access to recording and storage devices become more widely available, the amount of unlabeled and redundant data have also began to increase. This results in wasteful storage of meaningless data as large amounts of effort may be required to manually extract the information required to perform tasks such as source recognition to provide meaning to the data.

One such opportunity which emerges is in the field of audio signal processing where source recognition is performed on segments of the audio signal to reveal underlying properties in its source. This has a wide array of applications such as time-series segmentation for musical instrument diarization systems where the audio signal is partitioned into several pieces depending on what instrument is playing. Another motivating application is in source separation to decompose complex audio signals into distinct portions with meaningful interpretations.

This report focuses specifically on audio data storing sound signals produced by musical instruments. These types of sounds are greatly diverse and inherently complex leading to a wide range of qualitative properties such as timbre, style and genre [40]. Although these types of sounds have been widely researched, there is still a lack of a definitive method of representing and recognizing these properties. This means that any significant breakthroughs in this research area could result in a disruptive technology capable of spanning multiple industries.

The most prominent example would be for Music Information Retrieval (MIR), used by businesses and academics to perform tasks such as recommendation systems which seeks to predict consumers preferences towards a particular type of music. Being able to recognize the underlying properties of musical signals embedded within the audio data therefore becomes

extremely important. With the recent slew of new streaming services entering the market, companies are investing billions of dollars to stay ahead and differentiate themselves. Spotify is one of the largest musical streaming platforms and commands an approximate 60% market share. To retain their foothold on the industry, they have invested heavily into developing a superior recommendation system to improve user experience.

The huge demand and market value for superior recognition systems can also be seen in in Apple's recent $400 million acquisition of the music recognition app Shazam. Soundhog is another pioneer in the audio recognition industry and been recently valued at $800 million.

1.2 Context

The exponential rise of machine learning and artificial neural networks is truly a global phenomenon. Skilled practitioners have seen their market worth skyrocket in recent years, with its influence spreading to almost every industry around the globe. However, the largest strides using neural networks have been made in industries such as computer vision, natural language processing and medical analysis resulting in a vast array of context specific machine learning concepts and techniques remaining untested on other types of applications. According to Thorsten[41], it has been found that supervised learning serves as the most commonly studied paradigm in the field of machine learning. Considering the examples of known class labels, supervised techniques try to learn a model outputting labels for prior unseen examples, which is applicable to our project.

In light of this, our projects seeks to approach the circumstances as described in chapter 1.1 to perform musical instrument recognition using state of the art machine learning techniques and neural network architectures, drawn from different fields and contextual purposes.

We will approach this problem in a two phase process where we will first expand our contextual knowledge of our research area and gain a richer understanding of state of the art machine learning techniques, frameworks and training procedures. This will be done by conducting a thorough analysis of the most current machine learning literature and then applying and evaluating their effectiveness in recognizing digitally recorded sounds produced by a single instrument. In this phase, we restrict ourselves to evaluating only general and context specific techniques developed specifically for musical instrument recognition.

The second phase will build upon the knowledge gained from the first phase and look to evaluate more advanced architectures and techniques developed for a variety of different contextual purposes. These techniques will be used to solve the task of recognizing multiple

musical instruments playing at once in the same digital recording. Furthermore, we will also look to identify and bridge possible research gaps in the field. This will be done by identifying areas for improvement and conducting a thorough analysis and development process to design, implement and evaluate our custom solutions.

1.3 Project Aims

Our aims will therefore also be broken down into two phases.

Phase 1 Aims:

- Build a single instrument recognizer using context specific neural network architectures designed for musical instrument recognition

Phase 2 Aims:

- Build a multiple instrument recognizer using neural network architectures and techniques developed for different types of applications.
- Identify possible areas of improvement for multi-labeled data classification.
- Develop custom solutions for any identifiable research gaps in the field

1.4 Individual Contributions

Report Contribution Breakdown			
Area	**Primary Contributor**		
	Kenneth Ang	**Han Zhang**	**Jianran Chen**
Chapter	• ch. 1 • ch. 2.4, 2.5, 2.6 • ch. 3 • ch. 5 • ch. 6.1, 6.2, 6.4, 6.7 • ch. 7	• ch. 1 • ch. 2.1 - 2.3, 2.7 • ch. 3 • ch. 4 • ch. 6.1, 6.5 - 6.7 • ch. 7	
Figures	• all ch.4 • all ch.6 • fig. 7.1 - 7.10	• fig 2.3, 2.9, 2.10 • fig 5.1	• fig 8.1 • fig D.5 - D.10
Design Development	• model: ch. 5.2 - 5.4 • model: ch. 6.4 • loss fn: ch. 6.2	• model: ch. 6.3 • model: ch. 6.5 • model: ch. 6.6	
Miscellaneous	• Latex env. Set-up	• Latex env. Set-up	• Help with Reference

Chapter 2: Background Theory

2.1 Music Signal

This project studies the method of automatic recognition of musical instrument sound from audio signals. To design the whole project, we must first know the contents of a music signal and what makes music recognizable and distinguishable from other sounds.

Sound is produced by the vibration of objects. It exists as wave form and could transmit through medium (air, solid or liquid) and can be perceived by auditory organs or equipment. The way ears receive the sonic wave and how the brain deciphers the sound would impact the sound recognition of a human.

Music has four perceptive properties: timbre, volume, pitch and duration. Some properties have definite physical implications: pitch-frequency,volume-amplitude and duration-time. While timbre is the most significant property for the human to recognize the music instruments, it is difficult to define using physical parameters. In this section, we will introduce different properties of music sound.

Pitch

Pitch is the property of sound that people can perceive and differentiate various music notes. On a frequency-dependent scale, the pitch is a method of measuring the sound ranking. A more general explanation is that pitch can determine the sound as "higher" or "lower" in a continuous musical melodies. Music intervals can be used to describe the differences between pitches. The natural musical interval is octave, which exists when the note frequency is doubled.

The most straightforward sound is a sine sonic waveform which only has the energy from an unique frequency. Explained by Fourier series theory, a periodic waveform can be represented as the sum of a single fundamental frequency and countless harmonic components. The lowest frequency in the periodic waveform is the fundamental frequency which, is called "f0" and the harmonics are overtones. Partials are the combination of the fundamental frequency and

overtones. Nevertheless, when a single tone is played by the musical instrument, although the music signal may contain energy from several different frequencies, the human would still perceive only one tone.

Volume

The volume can be considered as amplitude in the waveform. The volume have complicated influence factors: energy contained, the frequency details and total length of the waveform.

The musicians can be told by the music score where to produce the different volume level of the notes. The music score is dynamic level of notes which represent various volumes. Ranging from low to high volume, the western music system uses pianissimo(pp), piano(p), mezzo-piano(mp), mezzo-forte(mf), forte(f) and fortissimo(ff) as the levels of loudness. These different dynamic levels can be found in . For a database system, the sound of certain instrument playing some tone in several dynamic levels can be recorded as diverse data.

Duration

The other property of the sound is duration, which describes the time of the music sound or the length of the waveform. The duration would affect the human's perception to other properties of the sound. The perceptions of duration and volume have positive correlation, which means the volumes is higher, the perceived duration would also increase. Moreover, when the duration is long enough, the notes would appear imperceptible changes in pitch. The most intuitive result from this phenomenon is the vibrato skill artistically.

To minimize the impacts of various duration, in our project, we would set a fixed time window(explained in chapter 5.1.1) to process the music signal. The ensuing effect is our model would not have the capacity to manage the influence of the long duration of notes.

Timbre

Timbre is a property that human could easily perceive when hearing the music, but in the engineering field, it is difficult to transform the timbre into a quantized parameter. Timbre is the main basis of the human brain to differentiate from different music instruments even when they are playing in the same note, dynamic level and duration.

The primary message that the brain process is the information contained in fundamental frequency. Some harmonic instruments such as most orchestral instruments can produce an overtone series which have the frequency in exact integer multiples of the fundamental frequency.

The harmonics which is spectra in engineering would also affect the perception of timbre. Moreover, For each harmonic, the partial (mentioned in Pitch section) envelope would influence the timbre perception[42]. The envelope depicts how a sound changes over time.

2.2 Audio Signal Representation

Humans can estimate and recognize the instrument, style or tempo of musical works by listening in a short time. However, recognizing music is not a simple task at the machine level. To preserve the audio information in machines, the waveform of the audio and spectrogram can be used. Besides, a more advanced level of the presentation can be used which includes a symbol (MIDI, score etc.) or metadata (musicians, publisher, tempo, style etc.)

Most of the digital audio files are captured by the microphones or other sensors to indicate the subtle changes in the air under the influence of sound waves. After the sensor, an Analog-to-Digital Converter (ADC) would make quantization in amplitude and time to the stored signal. This is the pulse code modulation, and it would produce a processed signal with a configured sampling frequency and bit depth. The signal stored in the machine is one-dimensional and comprise very little information. For our project, the raw audio signal must be transformed into a representation which contains both time and frequency information.

2.2.1 Fourier Analysis

Recall the Fourier Theorem that any continuous periodic signal can be expressed as the sum of series of sine or cosine infinite terms, each series has decided Fourier coefficients which are amplitude and phase parameters. By applying this theorem, any complex audio signal in our project can be resolved into a series of sinusoidal waves with diverse amplitude, frequency and phase. The result from Fourier transform would be a frequency-domain signal transferred from a time-domain signal. For the discrete signals, the Discrete Fourier Transform (DFT) can be applied. The discrete signal should contain finite sequence which has N sample points equally separated by sample time T.

The formula for discrete Fourier transform is shown in (2.1)

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-\frac{i2\pi}{N}kn} \tag{2.1}$$

Where k refers to the different channel for frequency spectrum, $x[n]$ is a discrete time-domain signal, and $X[k]$ is a discrete frequency-domain signal.

Fast Fourier Transform

Fast Fourier Transform (FFT) is an efficient and fast method to calculate Discrete Fourier Transform. A finite-length time domain signal can get the discrete frequency domain with finite-length sequence by DFT. But the amount of computation is too large, for the real-time task it is difficult to implement such an algorithm.

Using FFT can effectively decrease the amount of multiplications during computing DFT. Applying DFT to an N-point discrete signal would require $O\left(N^2\right)$ (refers to the complexity of computation) times of addition and multiplication. But, using the FFT algorithm would only demand $O(N \log N)$ times of addition and multiplication. Notably, with the increasing amount of sample points N, the FFT would decline the computation significantly. Hence, In the engineering area, FFT holds a significant place, especially when implementing models in reality.

FFT has limitation for non-stationary process. For example, when analyzing the non-stationary signal with frequency varying with time, FFT can only get the components of a frequency that the signal generally contains, but it does not know the moment when the components appear. Therefore, the two signals with different time domains may have the same spectrum result.

Short Time Fourier Transform

Short Time Fourier Transform (STFT) can overcome the limitation of FFT by adding window functions. STFT is the most common method to obtain spectrogram which could express the frequency spectrum and energy. STFT is specially designed for frames or windows of length N, which is a short time to acquire both phase and magnitude for each frequency bin k for a specific window. The formula for STFT is shown in (2.2)

$$X(k) = \sum_{n=0}^{N-1} w(n)x(n)e^{-j2\pi kn/N} \tag{2.2}$$

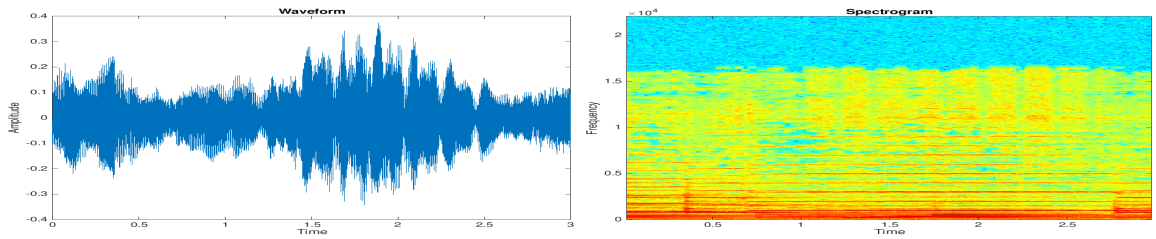
In (2.2), $x(n)$ is a particular point of the discrete signal; the function x has sampling rate fs, $w(n)$ is the window function containing N-point. The relevant frequency value for band k is decided by both fs sampling frequency and window length N as illustrated in (2.3)

$$f(k) = \frac{k}{N} \cdot f_{\mathrm{s}} \tag{2.3}$$

In STFT, the length of the window is always associated with the maximum number of bands (also known as frequency bins). In some situation, the window function needs to be shifted on the time-domain signal step by step; the step size is the hop-size, which is less than the window size in most cases. Then consecutive frame would be concatenated to the preceding spectrogram matrix.

2.2.2 Spectrogram

The spectrogram is a visible description of the spectrum of signals. A time-domain signal could get its spectrogram after the STFT. Intuitively, the spectrogram is a bidimensional plot with a color bar beside the 2-D figure. The horizontal axis represents time, and the vertical axis shows the range of frequency. In the musical audio signal analysis, the vertical axis could provide information about the pitch. The color bar (some spectrogram might not include a color bar) is the third dimension, it represents the energy or amplitude of the signal in specific frequency and time. The blue area means low power and the red area means high power. From blue to red, the brighter the colour is, the more energy would exist among the regions. According to this property, the spectrogram could be used to express the signal energy distribution of wide frequency range over time. Also, the spectrogram could provide the frequencies of an audio signal(i.e. transferring sound waveform into time and frequency domain). Figure 2.1 shows the spectrogram of a 3-second long flute sound.



(a) Waveform of audio (b) Spectrogram of audio

Figure 2.1: Waveform and Spectrogram of a flute sound

It needs to be noticed that the frequency and time values are discrete, each representing a "bin" and the amplitude is a actual value. In other words, the color shows the real value of the magnitude within the discrete time and frequency coordinates.

2.2.3 Mel Scale (Mel Frequency)

Mel scale could be the following process after the STFT to simulate the way human brain perceive pitch. The frequency "f" can be transformed into the Mel frequency with the bandpass

filterbank of M bands as (2.4)[43].

$$m(f) = 2595 \log_{10}\left(1 + \frac{f}{700}\right) \tag{2.4}$$

The Mel scale shows how the human brain would judge distances between different pitches. In other words, in the Mel scale, the human brain's perception of pitch is linear. For example, if the Mel frequency of the two sounds differs by a factor of two, the human would perceive the tone of these two sounds differ in two times. The following figure shows the Mel filter bank and the mapping from Hertz into Mel frequency.

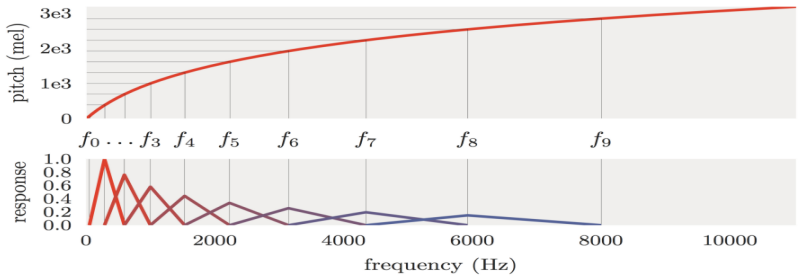


Figure 2.2: Hertz mapping to Mel frequency and Mel-filter bank [44]

Figure 2.2 indicates the mapping from Hertz to Mel frequency as well as the filter bank [44].The lower plot shows the the triangular Mel-filter banks and the upper plot shows the mapping relationship between frequency and Mel frequency.

2.3 Feature Engineering

Our project will process the audio signal into the features, which can be used in Machine Learning (ML) models. The upper limit of an ML task is determined by data, whereas models and algorithms approximate this upper limit [45]. Following this, the features we can get from data would be critical for ML. Feature engineering can be considered as a subject which studies how to convert raw data into particular features that characterize the former data effectively and use the models building on these features to achieve optimal performance (or close to optimal situation) on unknown data. Intuitively, the outcomes of feature engineering would be input variables manually designed. The results of machine learning tasks depend on the model architecture, data, and transformed features. Hence, better features would result in better performance and more simplified model needed to implement. In other words, once acquiring excellent features, although parameters in the model have not been tuned to an optimal value,

an acceptable performance would still be obtained, engineering efforts would not be wasted on looking for tuning the perfect parameters [46].

Principal Components Analysis

The principal components analysis(PCA) would be an effective method to decline the dimensions of data. PCA can find out the primary components among data and replacing the original data with vital data. PCA would convert data via linear transformation into new data which is linearly independent for all dimensions. Mathematically expression is function (2.5)

$$Z = W^T X \tag{2.5}$$

A low dimensional matrix, which is the output of PCA is Z, X represents the input with high dimension. W stands for transforming mapping.

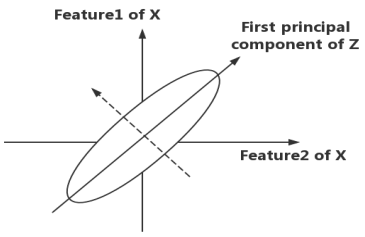


Figure 2.3: Principal components analysis

Exhibited in figure 2.3, input data is speared within an elliptical region. The zero position of coordinates requires subtracting local mean value for each axis. After the PCA algorithm, an oblique axis is regarded as containing the most principal elements, and the dotted axis orthogonal to it is considered as a secondary principal components which should be discarded to achieve the purpose of dimensionality reduction. The new feature axis can be characterized by the original feature axis using linear transformation. The orthogonal spindles mean that they are linearly independent. The choice of the principal linear component is the direction which would have increasing variance. The reason variance desired to enhance is generally the more dispersed the data distribute in a specific feature dimension, the more relevant the feature is. Correspondingly, the PCA is sensitive to outliers due to considering the variance. A significant impact on variance would be induced if a fraction of data locate away from the middle point, which also has a great influence on the feature vector. This would also explain why unnecessary and redundant data need to be removed before PCA [47].

2.4 ANN

2.4.1 Artificial Neurons

McCulloch-Pitts (MCP) Neurons

The MCP neuron model was first proposed by McCulloch and Pitts in 1947, based on the operating principles of a biological neuron[18].

$$a = \begin{cases} 1 & \text{if } \sum_{k=1}^{n_x} x_k > \theta,\ x_k \in \{1, 0\} \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

This model as defined in 2.6 accepts n_x binary inputs and outputs 1 when the unweighted sum of the inputs are greater than the threshold value θ. This implies that each input feature contributes equally to the output and did not provide a proper learning mechanism.

Perceptrons

As an extension to the MCP model, Frank Rosenblatt introduced the perceptron model in 1957[17]. This model used a weighted sum of inputs and an additional bias term as input to a Heaviside step function and removed the input vector's binary constraints .

$$H(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{otherwise} \end{cases} \tag{2.7}$$

$$a = H\left(\sum_{k=1}^{n_x} w_k x_k + b\right) = H\left(\boldsymbol{w^T x} + b\right) \tag{2.8}$$

A single neuron in the perceptron model as defined by 2.8 can therefore be used for classification problems by defining a separating hyperplane:

$$z = \boldsymbol{w^T x} + b \tag{2.9}$$

and obtaining the optimal parameter values $\boldsymbol{w}$ and b.

For binary classification problems, we can equivalently replace the Heaviside step function with Signum to obtain output values of $a = \{-1, 1\}$ to indicate the class to which an input feature vector has been assigned. This single layer perceptron network is trained using 1 where α is the learning rate hyper-parameter for each iteration of the algorithm.

Since the separating hyperplane 2.9 produced by a single layer perceptron is a linear combination of the input features and a bias term, this implies that this method would not be effective in performing classification on data which was not linearly separable such learning the XOR function as demonstrated by Minsky and Papert[38]. Furthermore, they argued that to be successful in doing so, multiple layers of perceptrons would have to be implemented.

2.4.2 MLP

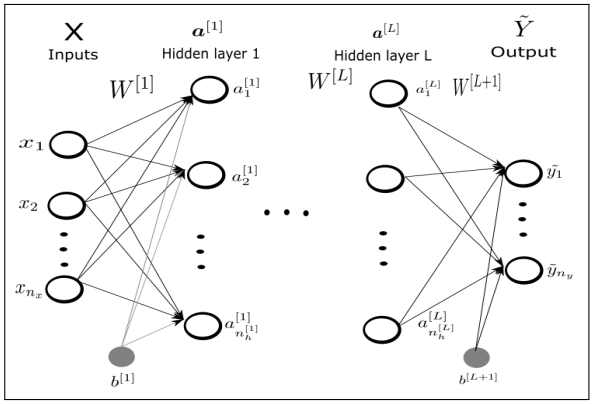


Figure 2.4: MLP Architecture with L hidden layers

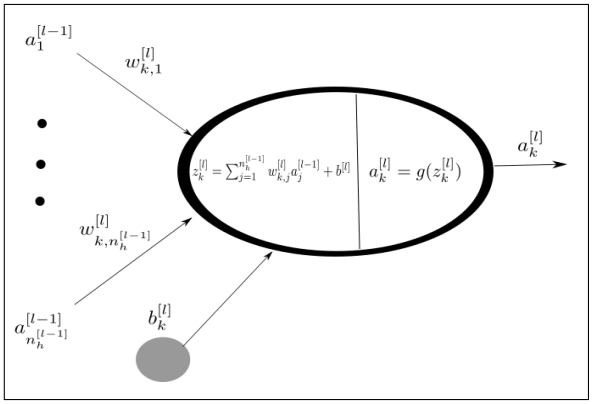


Figure 2.5: Close-up of the k^{th} neuron in the l^{th} hidden layer

Figure 2.4 shows a general L hidden layered MLP architecture with input feature vectors $x^{(i)} \in R^{n_x}, i = 1, ..., m$ and n_y output classes. At each layer, the activations (2.10) corresponding to $n_h^{[l]}$ separating hyper planes are transformed via non-linear activation functions (2.11), each of which preserves information from different parts of the previous feature space. The vector (2.11) corresponding to the output activations of each neuron in that layer is then used as the input feature vector to the next layer.

$$\boldsymbol{z^{[l]}} = (\boldsymbol{W^{[l]}})^T \boldsymbol{a^{[l-1]}} + \boldsymbol{b^{[l]}} \tag{2.10}$$

$$\boldsymbol{a^{[l]}} = g^{[l]}(\boldsymbol{z^{[l]}}) \tag{2.11}$$

The output layer consists of n_y neurons, each of which corresponding to a different class and the output layer activation functions $f(.)$ should be chosen according to the problem definition.To find the optimal weights $W^{[i]}, i = 1, ..., L+1$ and biases $b^{[i]}, i = 1, ..., L+1$ the MLP is typically trained via a recursive two stage process.

2.4.3 CNN

Motivated by image and pattern recognition, the use of ConvNets have historical roots tracing back to 1980 after the introduction of a neural network model capable of position invariant pattern recognition[20]. The use of ConvNets for computer vision have since been heavily popularized following breakthroughs in the field such as through the release of AlexNet[21] in 2012 which was shown to substantially outperform all other image recognition methods in the ImageNet challenge.

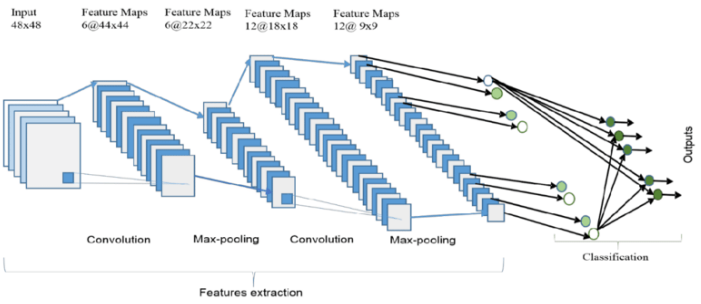


Figure 2.6: CNN Architecture [79]

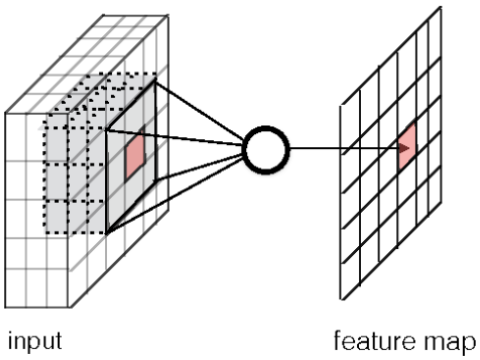


Figure 2.7: 2D Convolution

Figure 2.6 shows a general CNN architecture which have the ability to emulate complex, non-linear functions able to perform tasks such as classification by mapping high dimensional input data onto discrete lower dimensional spaces such as a finite number of output classes. Unlike the perceptron model, Conv layers are able to capture temporal and spatial dependencies using patch-like connections between layers where only local neurons within the kernel's receptive field connects to a single neuron in the next layer as depicted in figure 2.7. This allows the network to capture local dependencies at a high resolution and extract more complex features at lower resolutions using techniques such as pooling.

The way a filter is convolved across an input feature map depends on several hyper-parameters which consequently determines the dimensions of the output feature map.

Filter Dimensions

A filter with spatial extent $(w \times h \times d)$ connects a patch of as many local features to a single neuron in the next layer. Note that the filter depth d is non-configurable and must match the depth of the input feature map. This is because the overall convolved value at each step is a summation of the value across the entire depth of the current receptive field. Furthermore, filters in a convolutional layer convolve through various positions of the input feature map with

the same weights and can be interpreted as a detection mechanism for a specific type of feature at multiple locations.

Musically Motivated Dimensions

Subtle differences between using CNN for image versus audio spectrogram processing exist since image filter dimensions have spatial relations, while the latter is used to capture meaning corresponding to time and frequency[36]. The width of a filter therefore corresponds to the length of temporal dependencies the network is capable of learning while the filter height dictates the spread of timbral feature dependencies to learn.

It was also found that using the standard 3×3 squared filters may not be able to efficiently capture musical features which may be have long temporal dependencies or are spread in frequency[37]. Furthermore, this may also require the network to combine the partial representations using more filters [36].

Downsampling

Pooling

A crucial factor in achieving spatial invariance in the CNNs pattern recognition process is in the downsampling of output feature maps[24]. A common implementation of this is using a pooling layer directly after the convolutional layer to reduce the resolution of the feature map. One such example is the max pooling function which applies an $(n \times n)$ sized window and computes the maximum value of activations in the local area.

Striding

Recently, the replacement of a pooling layer with strides in the convolutional layer has been shown to produce similar results without any loss in accuracy on several image recognition benchmarks[39]. Striding reduces the size of an output activation map by increasing the step in each/either dimension of the convolution operation performed by the kernel. This also has the effect of parameter sharing along channels which reduces the number of required trainable parameters[25].

Padding

As previously mentioned, each convolutional layer reduces the size of the feature map which may be a problem for deeper networks where the features are unable to propagate through the network due to excessive dimension reduction. Padding is a margin of zeros placed around the input feature map capable of preventing the output feature map from becoming too reduced.

Filter Convolutions

Consider an input volume of $(N_h \times N_w \times d)$ with an additional padding of p margins going into the l^{th} convolutional layer with K channels of $(m_h \times m_w \times d)$ filters $W^{(k)}$ and a stride of (s_h, s_w). Then the output activation map will be of size $\left(\frac{N_h - m_h + 2p}{s_h + 1} \times \frac{N_w - m_w + 2p}{s_w + 1} \times K\right)$ and on k^{th} output activation map, the activation element on the i^{th} row and j^{th} column is given by:

$$z_{i,j}^{(k)[l]} = \sum_{h=0}^{m_h - 1} \sum_{w=0}^{m_w - 1} W_{h,w} a_{(i+h)(j+w)}^{[l-1]} \tag{2.12}$$

$$a_{i,j}^{(k)[l]} = g(z_{i,j}^{[l]}) \tag{2.13}$$

2.4.4 RNN

Recurrent Neural Network(RNN) is specially designed for sequence data. The sequence could follow the time order or text direction. In one word, the numerical value of data and the order information are both essential to a RNN model.

It is known from the basic neural network that the neural network consists of an input layer, a hidden layer, and an output layer. The layers are connected by weights, and the output is controlled by an activation function, which is predetermined. After training the network, the learned knowledge would be stored in the form of weight. The traditional neural network sets up a connection using weights between layers, and the innovation of RNNs is the connections are built between neurons and layers. Traditional Neural Network would not have a memory about preceding information. However, by adding a 'loop' in the networks, the information in RNNs could be stored and last longer[48].

Figure 2.8 shows a simple RNN structure where arrow connections mean transitions with weights. The subplot on the left is folded up version; the other subplot is the unfolded structure. In the folded structure, the circle around the recurrent layer represents the 'loop' which would occur in hidden layers. x is a vector that represents the value of the input layer. s is the

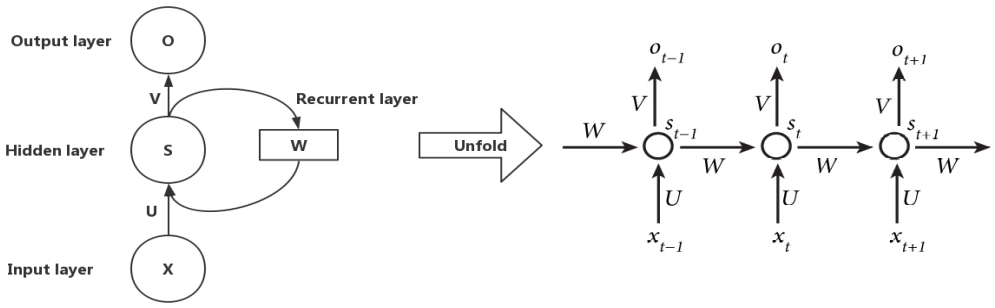


Figure 2.8: Network structure for a simple RNN [73]

vector represents the value of the hidden layer (here the hidden layer only has one node, but the number of nodes could be multiple). The number of nodes in the hidden layer equals to the dimension of s. o is the value of the output layer, which is also a vector. U and V are weight matrices which connect different layers. Illustrated in the unfolded plot, s_t value is not only depending on x_t and also the s_{t-1} from the preceding node. The W is the weight matrix connecting two adjacent hidden layers [49]. Using mathematical expressions [49]:

$$\begin{aligned} \text{Output at time } t: o_t &= g\left(V \cdot s_t\right) \\ \text{Memory at time } t: s_t &= f\left(U \cdot x_t + W \cdot s_{t-1}\right) \end{aligned} \tag{2.14}$$

Weighting matrices U, W and V represent linear relationship between the parameters. All the nodes of RNN would share the same weighting matrices. Also, the route for input at every time step is fixed and unique to maintain the sequence order information.

Long Short Term Memory Network

One of the limitations of RNN is the memory remembered most clearly is the newest signal input. The intensity of the earlier signal is declining along the time steps, and finally, the early message can only play a supporting role. Hence, the last input sequence would determine the output of RNN. Under this situation, long-range dependence is a main drawback of traditional RNN. RNN uses the backpropagation and gradient descent method to learn the parameters. The error function of the t layer is directly related to o_t , and o_t depends on the sum of x_i and $s_i, i < t$ of each layer before the time t. This structure tends to have the gradient vanishing and gradient exploding problems [49].

Long short term memory(LSTM) is designed in 1997 by Hochreiter and Schmidhuber [50] to handle these drawbacks of traditional RNNs. A "processor"(cell) is integrated into LSTM to decide the usefulness of the information. The key to LSTM is the state of the cell. The state

of the cell runs throughout the chain, with only a few linear operations acting on it, and the unchanged information can easily flow through the whole connection. LSTM relies on "gate" structure to delete or add information to the state of cell. Figure 2.9 shows the structure of a LSTM cell.

Structure of LSTM is shown in figure 2.9, x_t is the input to cell at time "t" and h_t is hidden layer state value at time "t". The Capital letter "A" refers to the same LSTM structure.

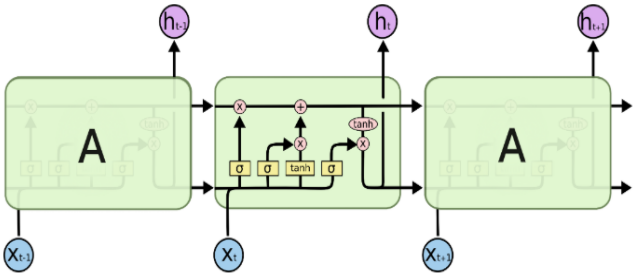


Figure 2.9: Structure of LSTM [74]

Input gate, forget gate and output gate are the three gated structures inside a cell. Once information enters the cell, it can be evaluated based on forgetting rules. Only information that satisfies the algorithm requirements will be retained, and the rest will be discarded via forget gate. All the three gates are based on the sigmoid function as the selection tool and tanh function as the transformation tool. The Sigmoid layer outputs a number between 0 and 1, which describes how much information each component can pass, 0 means no information, 1 means all pass[51].

Bidirectional RNN

Bidirection RNN (BiRNN) is a variant of LSTM. In the traditional RNN, the transmission of the state is one-way from front to the back. However, in certain tasks, the output of the current time is related to both previous state and subsequent state. A two-way RNN is suitable to solve such problems. For example, predicting a missing word in a sentence needs to be judged not only according to the preceding content but also according to the following content[52]. In the musical instrument tasks, the tone is continuous sound and can not finish in a single time step, all the duration of the tone contains vital information for recognizing the instrument. Using BiRNN can consider the whole duration of a tone sound.

The BiRNN is composed of two RNNs stacked one on top of the other. The output is determined by the state of the two RNNs. The mathematical explanation is illustrated in

(2.15)

$$\begin{aligned} \text{Forward hidden layer: } \vec{h}_t &= f\left(\vec{W}x_t + \vec{V}\vec{h}_{t-1} + \vec{b}\right) \\ \text{Backward hidden layer : } \overleftarrow{h}_t &= f\left(\overleftarrow{W}x_t + \overleftarrow{V}h_{t+1} + \overleftarrow{b}\right) \\ \text{Total output : } y_t &= g\left(U\left[\vec{h}_t; \overleftarrow{h}_t\right] + c\right) \end{aligned} \tag{2.15}$$

Knowing from the equation, the information created by bidirectional RNN is double than traditional one-way model. Having more information, the performance of neural network would be improved.

Attention Mechanism

Models using the Encode-Decode structure is very prevalent because it achieves better results in many areas than other traditional methods. Models in this structure typically encode the input sequence into a vector representation of a fixed length, and for a shorter length input sequence, the model can learn a corresponding vector representation. However, the problem of this model is that it is challenging to learn a reasonable vector representation when the input sequence is very long. The input sequence is encoded as a vector representation of a fixed length, regardless of length, and decoding is limited by the fixed length.

The most apparent advantage of attention mechanism is breaking the limitation of the traditional encoder-decoder structure that the vector should be fixed-length during the encoding and decoding. The Attention mechanism is achieved by conserving the intermediate output of the input sequence to the encoder and then training a model to selectively learn input and correlating the output sequences with overall outputs. In other words, the probability of each item in the output sequence depends on which items are selected in the input sequence.

Although the amount of computation would increase when using the attention mechanism, the model performance can be improved. In addition, implementing attention mechanism makes it capable of understanding how the information in the input sequence affects the final generated sequence during the network training. Attention mechanism can provide a better understanding of the internal workings of the model and make it possible to debug specific input-output[53].

Back-Propagation Through Time

The normal Back-propagation could not be used in RNN. In 1998, Williams and Zipser proposed a cyclic neural network training algorithm called Back-Propagation through time (BPTT)[62]. The essence of BPTT algorithm is to expand the cyclic neural network according to time series.

The unfolded network contains N (time step) implicit units and an output unit, and then the connection weight of the neural network is updated by reverse error propagation. So, we choose BPTT in our RNN models.

In fact, the main difficulty of RNN's back-propagation algorithm lies in the communication between its states, that is, the gradient has to propagate along the time channel($s_t \to s_{t-1} \to \ldots \to s_1$) in addition to the spatial structure($o_t \to s_t \to x_t$), which makes it difficult for us to write the BP algorithm of the corresponding RNN into a unified form. Unified form (recall the previous "forward conduction algorithm"). To do this, we can use the "loop" method to calculate the gradients.

2.5 Network Optimization

2.5.1 Gradient Descent

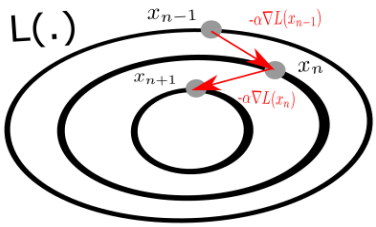


Figure 2.10: Gradient Descent
(Figure created using Inkscape)

Given a differentiable objective/ loss function $L(x_n)$, gradient descent searches the loss landscape for parameter values x_n which minimizes it (2.16). This is done by updating x_n at each iteration in the direction of steepest descent (2.17).The step size α_n is a configurable hyper-parameter which quantifies how much each parameter should be updated from its previous value at $n-1$.

$$\min_{x_n} L(x_n) \tag{2.16}$$

$$x_n = x_{n-1} - \alpha_n \nabla F(x_{n-1}) \tag{2.17}$$

2.5.2 Back Propagation

As discussed in chapter 2.4.1, Minsky and Papert's findings posed several problems since the perceptron algorithm adjusted it's weight vectors ($\boldsymbol{w}$) based on the immediate output layer predictions, which cannot be extended to adjusting the weights in intermediate layers.

This motivated the search for an alternative learning mechanism leading to the adoption

of the back-propagation algorithm after it's effectiveness in training multi-layered perceptrons/ neural networks was demonstrated in 1986[38].

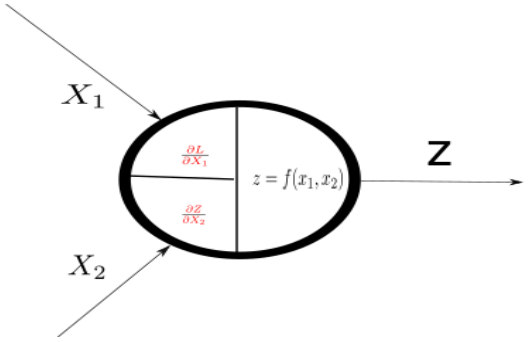


Figure 2.11: Forward Pass
(Figure created using Inkscape)

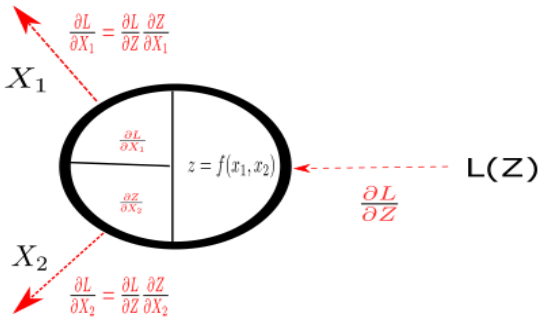


Figure 2.12: Back-Propagation
(Figure created using Inkscape)

The back-propagation algorithm provides an efficient method to compute intermediate gradient used to update parameters in the intermediate layers of the netork to learn internal representations of the input features.

Stage 1: Forward-Pass

Figure 2.11 depicts a single node during the forward pass. The node uses the input values flowing downstream to evaluates it's primitive function $f(.)$ and partial derivatives with respect to each input variable. Only the value of the primitive function is propagated to the next layer of the network while the partial derivatives are stored internally as local gradients.

Stage 2: Back-Propagation

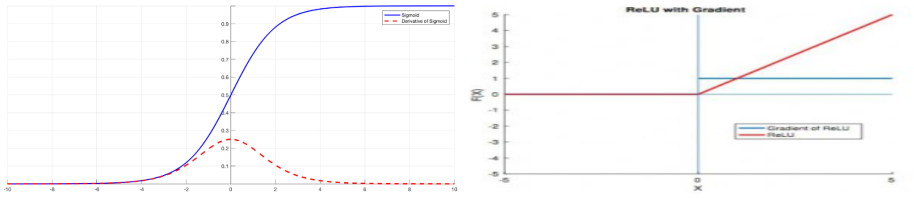
During back-propagation, the aim is to adjust the value of the configurable parameters in the network using partial derivative of the overall network loss with respect to each parameter. As shown in figure 2.12, the partial derivatives are calculated via the chain rule by multiplying the upstream gradient by the previously stored local gradient. It is also worth noting that upstream gradients flowing in from separate nodes will sum to produce the overall upstream gradient into the node.

2.5.3 Activation Functions

Activation functions are used to introduce non-linearity into the network making them universal approximators able to represent any function. The adoption of back-propagation as the primary

means of training neural networks also meant that activation functions were required to be differentiable for partial derivative to be obtained.

Sigmoid



(a) Waveform of audio

(b) Plots of the Sigmoid Activation Function ad it's derivative

Figure 2.13: Plots of the ReLU Activation Function ad it's derivative [80]

The sigmoid activation function (A.1) is a differentiable, monotonic function which maps input values into the range $[0, 1]$. This is especially useful for modeling an output probability for a given neuron and is commonly used in the output layer for binary classification problems.

From figure 2.13, we can see how its gradient peaks at $z = 0$ and decreases exponentially as $|z|$ increases and eventually asymptotes to 0. This causes the vanishing gradient problem for deeper networks using backpropagation. Since upstream partial derivatives are multiplied back up the network from the output layer, the gradients rapidly decrease at each consecutive layer and eventually vanishes, causing the update term in the gradient descent algorithm to vanish.

Softmax

The softmax activation function (A.2) maps elements of an input vector $\boldsymbol{z} \in R^k$ into the range $[0, 1]$ where $\sum_j^k z_j = 1$. This models the output probability distribution of neurons in a layer and can be used in the output layer of multi classification problems.

Rectified Linear Unit(ReLU)

Introduced in 2010, the ReLU activation function has been shown to produce state of the art results[26] and are the most successful and widely used activation function to date[27].

The function (A.3) performs a threshold on each element of the input vector and rectifies the negative values to zero. Figure 2.13 shows the function which is has a unity gradient for positive input values. This means that the upstream gradient propagating backwards does not get attenuated and hence solving the vanishing gradient issue presented above. Furthermore,

rectifier units help the network to obtain sparse representations resulting in faster and more efficient computations[28].

While the sparsity introduced by the rectified units has computational benefits, it may result in many weights not being updated for negative activations known as the dying ReLU problem. This can be prevented using other ReLU variants or by setting lower learning rates[29].

2.5.4 Loss Functions

Equation 2.16 shows the objective of neural network training which seeks to minimize the output loss. The loss function quantifies the cost associated with a predicted output and its true value and should be chosen according to the network task[30]. Since the networks minimizing parameter values are obtained using the gradient descent update rule(2.17), this means that loss functions are required to be differentiable for the update term to be computed.

This can often be a problem for tasks involving discrete outputs such as binary classification since many of such losses are non-differentiable and non-decomposable[30]. This problem becomes more prevalent for multi-label classification tasks where most of the loss criteria are generally highly non-convex and discontinuous[31]. A commonly used alternative is to consider a surrogate loss function of the desired criterion which serves as a proxy to the actual loss.

Cross-Entropy (CE)

With roots in information theory and communication, cross entropy is a measure of uncertainty between two sample distributions[32] and is defined as:

$$J(\tilde{y}, y^t) = -\sum_{i}^{C} y_i^t \log \tilde{y}_i \tag{2.18}$$

Where y_i^t and $\tilde{y}_i$ are the i^{th} discrete samples from two different distributions containing C samples each. Applied to a binary classification problem with $C = 1$ possible output class,the true label is given by $y = \{0, 1\}$ and the predicted output is in the range $\tilde{y} \in [0, 1]$ corresponding to the predicted probability that an input belongs to that class. The cross-entropy formula therefore reduces to :

$$J(\tilde{y}, y) = \begin{cases} -\log \tilde{y} & \text{if } y = 1 \\ -\log 1 - \tilde{y} & \text{if } y = 0 \end{cases} \tag{2.19}$$

Categorical Cross-entropy

A similar formulation can be applied for multi-class problems with C possible output classes and a constraint of a single positive output label per input. This is commonly used with the softmax activation function (A.2) which produces a probability distribution across all C output neurons. Due to the single-label constraint, the formula reduces to (A.4), where $\tilde{y}_p$ is output probability of the neuron corresponding to the single true label.

Back Propagation Multi-Label Loss (BP-MLL)

The BP-MLL algorithm was initially developed to exploit back-propagation to train multi-label classifier networks. Using a standard MLP, a new global loss function (A.5) was defined to address label dependencies during training [33].

Here, the predictions from the output neurons were split into two sets according to their true value. Therefore if the training example contained P positive labels, then the positive set would be given by $\{y_1, ..., y_P\}$ and the negative set would contain $\{\bar{y}_1, ..., \bar{y}_{C-P}\}$ where C is the total number of possible output classes. Then $(y_p - \bar{y}_n)$ measures how far a positive label in the training example is from a negative label[34]. Therefore, BP-MLL minimizes errors induced by incorrectly labeled pairs, resulting in higher output values for neurons in the positive label set [35].

Binary Cross-entropy

Following the advances in deep-learning, it was later shown that using a cross-entropy loss with sigmoid output activation functions matched and in some cases even outperformed BP-MLL [35]. This was an extension of the previously mentioned cross-entropy loss to multi-class, multi-label classification tasks. Here, C binary classification problems are set up and the overall loss for a training example is given by equation A.6.

During back-propagation, the partial derivative w.r.t each output neuron will only contain the loss for that neuron. This means that the weights connected to an output neuron is only updated using the loss given by that neuron's binary problem which is equivalent to having C separate networks in the output layer connected to a single network in the layer before that.

2.6 Network Training

2.6.1 Input Batches

Batch Gradient Descent

Batch gradient descent (A.7) calculates the network loss by running the entire training set through the network and takes the average loss across all m training examples. This requires the entire training set to be loaded into memory and weight updates only occur after cycling through all training examples. This leads to very resource intensive and infrequent weight updates, resulting in slower convergence to the minima. Since the loss is averaged across all training examples, this also results in better gradient approximations and a smoother path towards the minima.

Stochastic Gradient Descent (SGD)

To address the issues presented for batch gradient descent. SGD updates the network weights using the loss of one sample at a time (A.8). This results in more frequent weight updates and faster computations. The frequent weight updates also results in a noisy path towards the minima. Although this may allow the network to escape local minimums, it takes longer to converge towards a minimum value for the loss function.

Mini-Batch Gradient Descent

Mini-batch gradient descent is a compromise between GD and SGD where the m training examples are divided into mini-batches of b examples each. Weight updates are made based on the average loss of each mini-batch (2.20). This results in faster computation and more frequent weight updates compared to GD and a noise reduction on the path to the minima compared to SGD.

$$L(\tilde{y}, y^{true}) = \frac{1}{b} \sum_{j=k}^{k+b} J(\tilde{y}^{(j)}, y^{true(j)}) \tag{2.20}$$

2.6.2 Gradient Descent Variants

Momentum

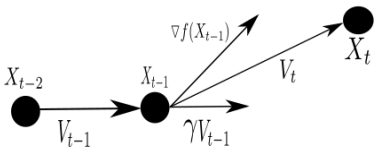


Figure 2.14: SGD with momentum (Figure created using Inkscape)

To improve the performance of SGD, a momentum term was introduced which helps to dampen the high variance of oscillations and obtain faster and more stable convergence[19]. This is done by adding a fraction γ of past update vectors to the current update vector (2.21). Figure 2.14 shows how the momentum term adjusts the current gradient update by increasing the dimensions of the update vector whose dimensions point in the same direction and reduces for those with changing gradients.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla f(x_{n-1}) \\ x_t &= x_{t-1} - v_t \end{aligned} \tag{2.21}$$

Adam

Adam is a viral algorithm in the field of deep learning because it can achieve excellent results very quickly, training speed has increased by 200%[61]. The empirical results prove that the Adam algorithm has excellent performance in practice and has significant advantages over other kinds of stochastic optimization algorithms. For Adam, the first-order momentum is from SGD, which is shown in (2.22):

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \tag{2.22}$$

and the second-order momentum is from AdaDelta that is in(2.23):

$$V_t = \beta_2 * V_{t-1} + (1 - \beta_2)\, g_t^2 \tag{2.23}$$

Adam can benefit from having these two points. The first-order momentum is the exponential moving average of the gradient direction at each moment, which is approximately equal to the average of the gradient vector sums of the nearest $1/(1-\beta_1)$ moments. That is to say, the descending direction at time t is determined not only by the gradient direction of the current

point but also by the direction of the downward direction accumulated before. The emergence of second-order momentum means the arrival of the era of "adaptive learning rate" optimization algorithm. SGD and its variants update each parameter with the same learning rate, but deep neural networks often contain a large number of parameters that are not always available (like large-scale embedding). Second order momentum can measure the historical update frequency by changing actual learning rate from α to $\alpha/\sqrt{V_t}$ [61].

2.7 Deep Learning Concepts

The characteristic shared by music and deep learning is that they both incorporate hierarchical structures [69]. In deep learning, abstractions of features can be learned by building hierarchical models with multiple non-linear layers. Then, contextual dependencies of a particular data structure could be recognized, and the corresponding model would be assigned to complete specific tasks. In the area of MIR, considering the robustness in the structure of data, CNN has been proved having the applicability to successfully implement to capture the hierarchical features.

Features extraction have to be carefully designed to represent the essential characteristics of data before the occurrence of deep learning. A separate discipline is founded to study and create the algorithms for extracting features. However, manually designed features are considered to reach the limitation [70]. Consequently, CNN would have its defects for abstracting features. To achieve good performance, CNN requires sizable dataset and substantial computational capability. Also, a certain amount of knowledge about machine learning would be necessary to produce high-efficiency models.

In the area of audio and image research, deep learning has shown considerable superiority. Also, the image aspect would provide valuable experience and inspiration to the audio aspect. According to different inputs, deep learning for audio could be categorized as end-to-end learning and mid-level representation learning. The raw signal is the input to end-to-end and mid-level representation takes spectrogram as the input.

2.7.1 Transfer Learning

Traditional machine learning only has outstanding performance when both training set data and test set data come from the same feature space and exhibit uniform distribution. In reality, the data of the practical tasks might dissatisfy these conditions. Mainly, for MIR tasks, the

dataset is relatively exiguous. Hence, transfer learning could be introduced into MIR area. In this project, the main content of transfer learning would be the inductive transfer learning and feature representation transfer. In other words, the source task would produce the learned features; then, the created features could be implemented into a diverse but relevant task. Figure 2.15 introduces the different learning processes between traditional ML and transfer learning.

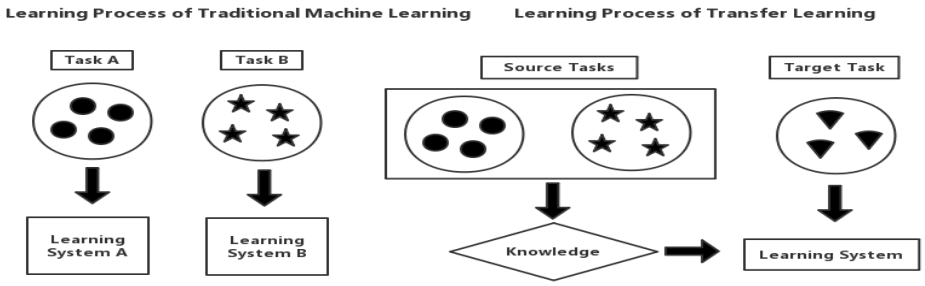


Figure 2.15: Compare between traditional model and transfer learning model [75]

Transfer learning can first process a simple task then apply the knowledge obtained in uncomplicated tasks to more difficult problems. In this way, missing annotation and inaccurate annotation of a dataset would not become a severe problem. Also, transfer learning can be used for tasks with small dataset whose size is not sufficient to generate features capturing all the considerable characteristics. Moreover, the transfer learning could be used to find the universal and common characteristic or features more in-depth to accomplish the task[54].

2.7.2 End-to-End Learning

End-to-end learning is a structure which all stacks from the beginning to end, then the whole structue would learn from the data. Using only the raw data as input, the insights into which information is significant for a particular task and further understanding of the inherent data structure would be obtained via feature learning procedure [71]. With end-to-end learning, the requirements for prerequisite expertise about the task would be dispensable(i.e. the properties of musical audio would be unnecessary for our project). Also, the engineering effort would also be reduced massively. The specific knowledge is needed for manually tuning the hyperparameters of the model. Nevertheless, the tuning can be auto-completed. Figure 2.16 demonstrates the difference between the traditional model and the end-to-end model.

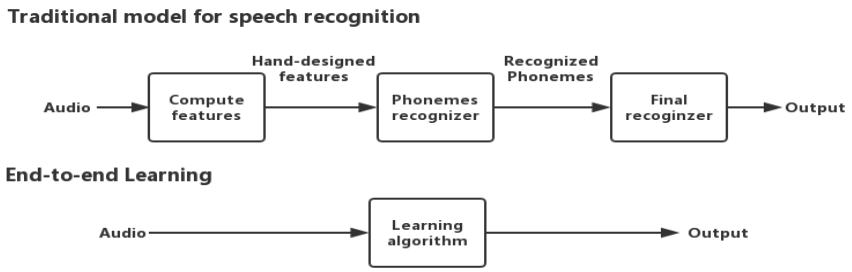


Figure 2.16: Compare between traditional model and end-to-end model [76]

By reducing manual pre-processing and subsequent processing, the model can be as much as possible from the original input to the final output. The end-to-end can give the model more space to adjust according to the raw input data automatically; this could increase the overall integration within the model. The size of the dataset would affect the performance of end-to-end learning. CNN is ideally suitable for end-to-end learning because it consists of many processing layers which use the same objective function that propagates across the network[55].

2.7.3 Ensemble Learning

In the supervised learning of machine learning, our goal is to train a stable model that performs well in all aspects, but the actual situation is often not ideal, sometimes we can only get multiple models with preferences (weak supervised learning models perform better in some respects). Ensemble learning can combine multiple weak models to get a better and more comprehensive model. The idea of ensemble learning is that even if one weak classifier gets wrong prediction, other weak classifiers can correct the mistake. Figure D.1 illustrate the structure of ensemble learning.

There are some types of ensemble learning: bagging(to reduce variance), boosting(to reduce deviation) and stacking(to improve prediction result). In our design, we will use stacking method. To apply stacking, we will firstly train different models, and then train ensemble model by inputting the output of each previously trained models to get a final output [72].

Chapter 3: Project Formalization

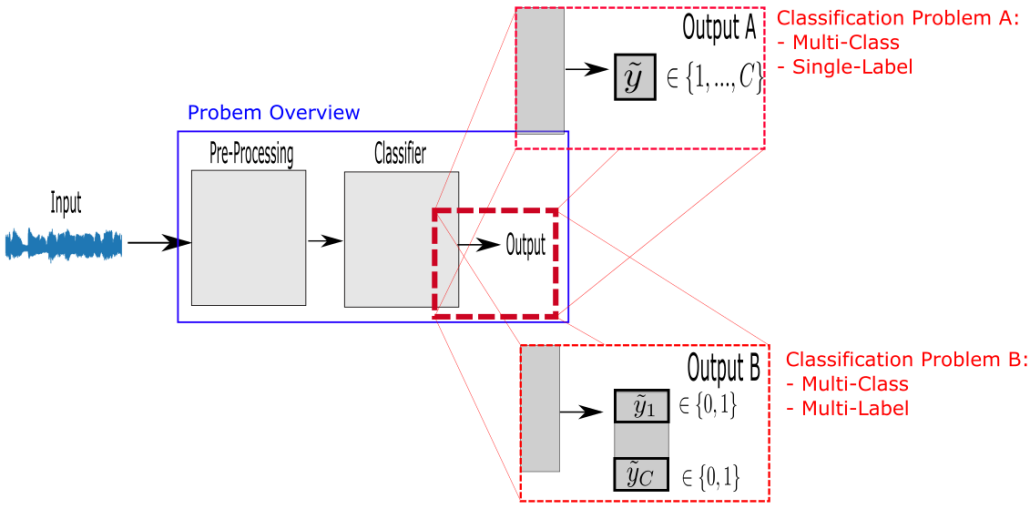


Figure 3.1: Project Overview (Figure created using Inkscape)

3.1 Overview

Figure 3.1 shows the general overview of our project where the task of instrument recognition in audio files is formulated as a classification problem. Given an input audio file, our goal is to build a model which is able to identify and recognize the medium generating the musical signals embedded within the file. We will also use a single model to recognize from a set of multiple possible mediums where C is the cardinality of the set of possible mediums. This problem will therefore be posed as a multi-class classification problem where each of C possible mediums will be represented by an integer in the set $S_{labs} = \{1, ..., C\}$.

Furthermore, By making appropriate assumptions, we are able to narrow the scope of our project and decompose the problem into simpler sub-problems which will then be solved sequentially.

Problem A: Single-Label Classification

The first assumption made about the input is that the musical signal at any given time-step is generated by a single medium. This implies that our model should produce a single output value for each input sample. This would correspond to the most likely medium which generated the musical signal captured by the input.

Formally, the output space of our model will be given by equation 3.1 and our model will be designed to output an integer value for each input sample. As shown in figure 3.1, this will therefore be posed as multi-class, single label classification problem.

$$S_{out1} = S_{labs} = \{1, ..., C\} \tag{3.1}$$

Problem B: Multi-Label Classification

The natural generalization of the previous assumption is to account for multiple possible mediums generating musical signals at each time step. This means that instead of an integer output, our model will be designed to output a binary prediction for each possible label. Therefore, this task will be posed as a multi-class, multi-label classification problem.

Project Limitations

Due to time constraints and design practicality, we have imposed several performance restrictions when designing our model which may affect the generality of our project. Due to the formulation of the recognition task as a classification problem, it implies that our model will only be able to perform recognition tasks accurately on musical signals generated from a set of pre-determined mediums. These mediums will be selected based on the quality and availability of our sourced data-sets.

3.2 Problem A: Formal Definition

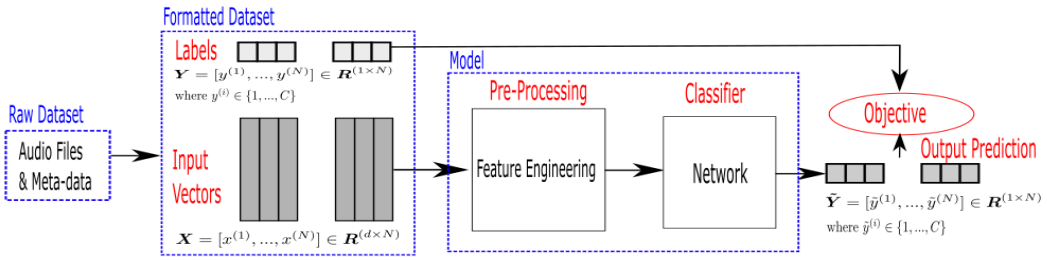


Figure 3.2: Multi-Class, Single-Label Problem

Given a dataset $D = \{(x^{(i)}, y^{(i)}), i = 1, ..., N\}$ containing N input-output pairs. The dimensions of the i^{th} input vector is $x^{(i)} \in R^d$ and the i^{th} output is an integer $y^{(i)} \in \{1, ..., C\}$ where C is the total number of possible output labels.

The objective is then to design a model $M(.)$ which maps each $d - dimensional$ input vector to an integer corresponding to an element in the label set $\tilde{y}^{(i)} = M(x^{(i)})$, such that the prediction error $L(y^{(i)}, \tilde{Y}^{(i)})$ across the entire dataset is minimized.

Then given $x^{(new)} \notin D$, but generated from the same process which generated $x^{(i)} \in D$, the predicted output will be given by $\tilde{y}^{(new)} = M(x^{(new)})$.

3.3 Problem B: Formal Definition

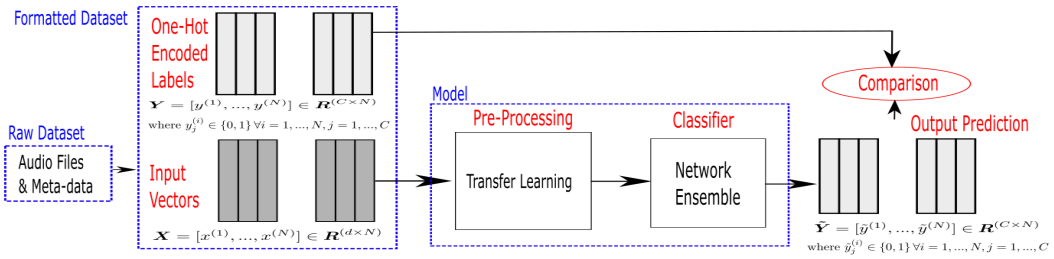


Figure 3.3: Multi-Class, Multi-Label Problem

For the multi-label classification, a classification example is more than the label of one class. Thus, every example $\mathbf{x} \in \mathbf{X}$ has been assigned with various labels $\mathbf{Y}$ through the classifier, in which $\mathbf{Y} \subseteq \mathbf{C}$. To be more specific, the classifier function is g under specific instance $\mathbf{x} \in \mathbf{X}$. It results in:

$$g(\mathbf{x}) = [g_1(\mathbf{x}), g_2(\mathbf{x}), \cdots, g_k(\mathbf{x})]^T \tag{3.2}$$

in 3.2, $g_j(\mathbf{x})(j = 1, \cdots, k)$ is either 1 or 0, suggesting the connection of x with the jth label.

Binary-Relevance(BR) method, namely the Binary-Relevance method, is the most common one for problem transformation. Different binary classifiers of $|\mathbf{C}|$ have been learned by the BR method with one for every probable label. Every binary classifier has been cultivated for the distinguish of the examples in the single class from all the cases of the all the left classes. When it comes to the classification of a new example, all the $|\mathbf{C}|$ classifiers operate and also there are labels related with the classifiers for the outputting of the label true being added to $\mathbf{Y}$. This has been addressed as the one-vs-all (OVA) scheme [78].

To be more specific, every binary classifier $Classifier_c$ has the responsibility of forecasting the false/true connection for every single label $\mathbf{c} \in \mathbf{C}$. The last label set $\mathbf{Y}$ refers to the set of labels of all the classifiers of the returned true. The BR method's complexity is found to be

linear in the models $\mathcal{O}(m)$ in which $m = |\mathbf{C}|$ class labels. If a new class label is added to the BR method, the additional model needs additional training to deal with the new class label. The current models need to be updated to a degree, which can be different from the new one. Our project will apply the BR approach for the multi-label classification.

Chapter 4: Tools and Environment

4.1 Environment

•Hardware: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, 16.0 GB RAM and NVIDIA GeForce GTX 1070 GPU.

• Software: python 3.6.8, Tensorflow 1.13.1, Keras 2.2.4 and Jupyter Notebook

4.2 External Modules

4.2.1 Feature Extraction

Librosa

Librosa is a powerful python toolkit for audio analysis and processing. Some common functions such as time-frequency processing, feature extraction, and drawing sound graphics can all be found in this toolkit. In this project, we will use Liborsa to extract Mel-spectrogram features from the audio files.

VGGish model

VGGish model is firstly released by Hershey in 2017.[65] Initially, this model is used to process AudioSet [66] dataset to produce 128-dimensional embeddings feature of each AudioSet segment. Inspired by successful Visual Geometry Group (VGG) model [58] for image classification tasks, the VGG-like audio classification model has a simplified structure and is trained on a large YouTube-8M (contains 2.6 billion audio and visual features) dataset.

VGGish has been confirmed from Hershey's paper that it have high performance when classifying audio files and can generalize well with other audio dataset. Hence, in our project, we will use VGGish model to generate embedding features. Referring to chapter 2.7.1, using the VGGish model would follow the idea of transfer learning which could use the knowledge

from another model to train own model.

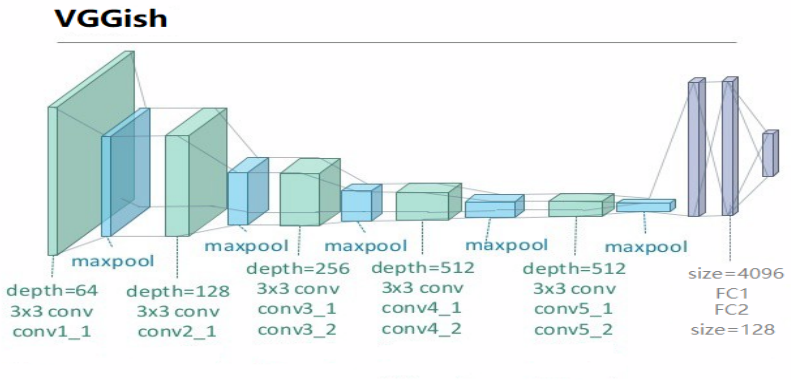


Figure 4.1: Structure of VGGish model

Figure 4.1 shows the structure of VGGish model. All the kernel size is 3x3, which is the smallest size that captures the top, bottom, left, and right positions. Stride is a constant value equals to one. Same padding is applied, that is, keep the scale of the feature map unchanged after convolution. Using five Max-pooling layers with window size is 2 and stride is 2, not all convolution layers are followed by a pooling layer. ReLU is used for all hidden layers. After the convolution layers, there are three fully connected layers, the last FC layer is 128 dimension, which would produce 128 embedding features.

The receptive field using two continuous 3x3 convolution kernels in a row is equivalent to a single 5x5 kernel. The reasons for VGGish do not use a single 5x5 kernel are: first, two nonlinear functions are used instead of one time, which increased the discriminating ability of the function. Secondly, this could reduce the number of parameters which can decline the computational amount.

4.2.2 Data Storage

We use "h5" format and "csv" format to store the data, the reserved raw data and results of models can be found in the USB flash disk.

4.2.3 Performance Evaluation

We write the code based on "sklearn" package to get the evaluations of results. The code can be found in the USB flash disk.

4.3 Dataset Selection

4.3.1 Problem A: Single-Label Classification

We have used the Medley-Solos-DB dataset which consists of fixed duration audio files. Each of which contains a single instrument among a taxonomy of eight possible instruments. Each audio clip is stored as a single channel (mono) WAV file with sampling frequency $f_s = 44.1kHz$ and duration $T = 2972ms$. Here we choose 7 instruments (clarinet, violin, guitar, piano, voice, flute, drum). Each class label will consist of equal number of training examples. This selection also allows us to have two percussion, string and woodwind type instruments.

4.3.2 Problem B: Multi-Label Classification

OpenMIC-2018 Dataset

The OpenMIC-2018 dataset contains 20,000 examples; each of them is a 10-second extract from free and open music material. In total, the dataset has 20 different instrument labels; each data from the dataset would have a different number of labels indicating the instruments playing for 10 seconds[56].

It is challenging to find a dataset labelled with multiple instruments. From online searching, it can be confirmed that the existing datasets about music can be divided into two categories. One only contains sounds of a single instrument that plays a note or a snatch. In this way, the data is easy to collect and tag annotation. However, the music collected from isolated instrument has dissimilar acoustic properties with the recording from several instruments. The other type of dataset would include a mix of multiple tracks music as well as music of each track. Every track could be the sound of an instrument or vocal voice(MedleyDB[57]). But, mixed audio is hard to label the instrument. Table 4.1 shows the comparison between OpenMIC-2018 and some commonly used music dataset.

Collection	# Excerpt	# Instrument	Duration	Diverse	Polyphonic	Multi-label
RWC	3,544	50	scale			
NSynth	305,979	1,006	note			
MedleyDB	122	80	song	✔	✔	✔
MusicNet	330	11	song		✔	✔
IRMAS	6705	11	3 seconds	✔	✔	
OpenMIC-2018	20,000	20	10 seconds	✔	✔	✔

Table 4.1: Comparison between musical audio datasets

In OpenMIC-2018 dataset, for each data, there would be a relevance value ranging in [0,1] along with the instrument label representing the credibility of a certain notation. This is because the OpenMIC dataset trains a model based on AudioSet data to automatically tag the labels. Hence, the dataset would have inherent limitation that the accuracy of tagging instruments cannot be completely 100%.

Label Encoding

In the Openmic-2018 dataset, there is a tabular data called aggregated-labels.csv, which stores the data id and relevance of each instrument.

Then we will take this data has three labels which are "Trombone", "Violin" and "Trumpet". After getting the name of the instrument, we will use One-hot Encoder to transfer the label in a way machine can read. Hence, we create a vector with 20 numbers; each number indicates a specific instrument.

In the vector, each position represent an instrument, the "0" indicates this instrument is not the label for this data, the "1" shows the appearance of an instrument.

Choosing confident data

To improve the recognition performance, we only choose the labels with relevance greater than 0.9 in the OpenMIC-2018 dataset. Hence, the size of training data would be decreased. However, with more confident true labels of data, our models can have better performance.

Chapter 5: Problem A - Single-Label Classification

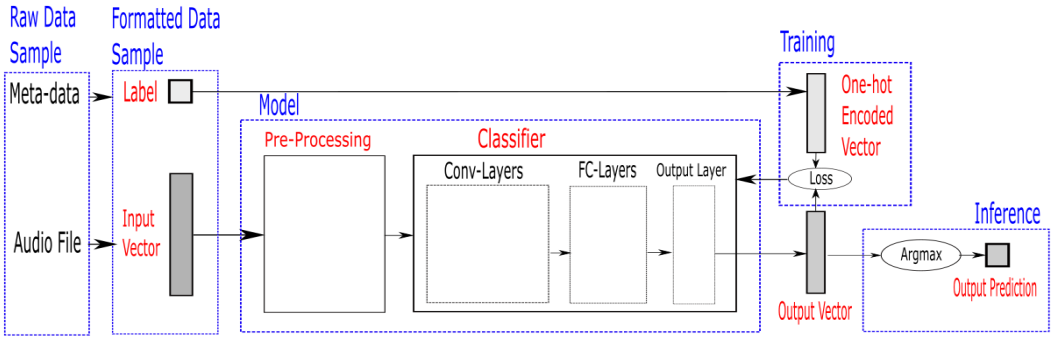


Figure 5.1: Problem A Process flow for a single data sample
(Figure created using Inkscape)

5.1 Network Peripherals

5.1.1 Input Features

Using the Medley-Solos-DB dataset, each audio clip is read into the our python program as a d-dimensional numpy array. Since we are given that each clip has a fixed duration and sampling frequency, the number of elements in the input vector is therefore given by:

$$\begin{aligned} d &= f_s \times T \\ &= 44.1 \times 2972 \\ &= 131066 \end{aligned}$$

Next, we use the vector to construct a power mel-spectrogram. The first step is to pass the input signal through a high pass filter described in formula (5.1)

$$\mathrm{H(Z)} = 1 - \mu z^{-1} \tag{5.1}$$

The value of μ should be within 0.9 to 1.0 range; we choose it to be 0.97. The purpose of pre-emphasis is to emphasise the high-frequency part, flatten the spectrum of the signal, and make sure for the entire frequency band the consistent signal-to-noise ratio can be used to calculate the spectrum.

Then, Framing would be applied to signal which would be split into short-time frames. The settings we use are frame size = 10ms and stride = 10ms. Then, we would apply a Hamming window to each frame S(n) in (5.2):

$$S'(\mathrm{n}) = \mathrm{S}(\mathrm{n}) \times \mathrm{W}(\mathrm{n}) \text{ where } \mathrm{W}(\mathrm{n},\mathrm{a}) - (1-\mathrm{a}) - \mathrm{a} \times \cos\left[\frac{2\pi \mathrm{n}}{\mathrm{N}-1}\right], 0 \leq \mathrm{n} \leq \mathrm{N}-1 \qquad (5.2)$$

Different "a" in (5.2) value would produce different Hamming window; we choose $a = 0.46$ in our design. Then FFT is applied to obtain the energy distribution along the spectrum for each frame. i.e. we apply a FFT based STFT to the audio signal to get the power spectrum.

Then, the energy spectrum would go through a Mel-filter bank; we choose 40 Mel-filter to get the Mel-spectrogram. 40 filters would smoothen the spectrum , eliminate the effects of harmonics and highlight the format of the original speech. In addition, the number of calculations would be reduced.

This was implemented using the *melspectrogram()* function from the Librosa module with default parameter values to obtain a 2D matrix with shape $(128, 256)$. Finally, we convert the spectrogram from power to log scale using the *power_to_db()* function to produce the desired input feature map.

Fixed-Length Windows

Our design uses a 10ms fixed-length window to extract the Mel-spectrogram features. With the selected window, we assume that the analysis under window function is stationary (pseudo-smooth) in a short time interval. The window function is then shifted such that the output is a stationary signal representing a different finite time within the width. Hence, the power spectrum at various times can be calculated.

Frequency is a linear quantity with unit Hertz. However, the pitch is perceived in a logarithmic manner (introduced in chapter 2.1). The semitone notation maps frequency to pitch logarithmically. While FFT would define a consistent resolution for the whole frequency domain. This would result in partials with higher frequency would be processed with greater resolution than the partials with lower frequency (this would also be shown in the Mel filters distribution mentioned in chapter 2.2.3).

In our project, the notes played by all instruments would have a wide range of frequency distribution. Hence, the signal should have adequate length to guarantee even for low-frequency part the resolution is suitable for FFT analysis. Following this rule, there is a trade-off for FFT about the time length of the input signal and resolution of the output frequency. For example, if the input signal has a short time length, then the resolution of the low-frequency part of the output would be low, which could have a significant influence on the analysis. Notably, the low-frequency part is the vital range for musical audios.

To avoid the inconsistent resolution among the whole frequency spectrum, in this project, we would choose a constant and suitable time window rather than numerous windows sliding with time, which could overlap with each other.

5.1.2 Output Activations

To select our output activation function, we look at the output layer requirements as defined by the problem statement. This would consequently determine the desired properties of the output activation function.

From the inference phase in figure 5.1, we can see that the final stage of the model maps a C-dimensional output vector produced by the network to an integer in the set of possible labels. Intuitively, this can be implemented using the argmax function which returns the maximizing point in the domain of the input set.

$$\tilde{y} = \underset{i \in \{1,...,C\}}{\operatorname{argmax}} [\tilde{y}_1, ..., \tilde{y}_C]$$

The above implies that the output values at each output neuron are only important relative to each other. This relative importance can therefore be modelled as a probability distribution across all output neurons in the network. Therefore, we will use the Softmax activation function for the output layer of our model which emulates this property.

5.1.3 Loss Function

As a consequence of our Softmax output activation function selection and for reasons as described in chapter 2.5.4, we have chosen the categorical cross-entropy loss function A.4 to be optimized by the network.

In doing so, we will also need to convert the labels in the dataset into one-hot encoded vectors. The conversion was implemented using the *to_categorical* function on the Keras Ap-

plication Programming Interface (API).

5.2 Network Training

5.2.1 Learning Algorithm

The *fit* method on the Keras API provides us with a quick and efficient method to update network parameter weights by internally implement the learning algorithm derived in appendix B.1.0.1. We are also able to implement variants of the SGD algorithm as described in chapter 2.6.2 using the functions defined in the *keras.optimizers* module.

5.2.2 Regularization

To prevent our model from over-fitting, the *fit* method allows us to either select a validation split or to manually input a validation set. We have chosen the latter since we will perform cross validation analysis to ensure reproducibility of our results.

The early stopping regularization technique is then implemented using the *ModelCheckpoint* constructor. Using this, we will only save model weights into a *hdf5* file after epochs which minimize the validation loss. Note that the model continues training even after the stopping condition has been met. This allows us to analyze the model's long term training behaviour.

The use of dropout layers have also been implemented using the *Dropout* constructor on *Keras*. We will analyze the results of using different dropout rates and combinations in chapter 5.4.3.

5.2.3 Performance Evaluation

Metrics

For a single-label classifier, our model will be evaluated using the standard accuracy metric. This is implemented using the *categorical_accuracy* method on *Keras* which applies an Argmax function on our network output and returns the ratio of correct predictions to total examples.

Validation

To implement cross-validation analysis, we will use the *StratifiedKFold* constructor from the sklearn API. This is a variation of K-folds where folds are made to preserve percentages of

samples between each class. We have chosen to use $K = 10$ folds to create different splits in the dataset which will be used to train our model.

At the start of each iteration through the different dataset splits, our model weights will be reinitialized before calling *fit* on the chosen Keras model. The quantitative performance of the model will be saved after each iteration and then averaged out at the end of the entire k-fold training process.

Visualization

Our model will also be analyzed qualitatively through layer visualization techniques. We will mainly visualize intermediate layer CNN activations to identify the type of features being propagated through the network and to make inferences about the network performance and efficiency.

5.2.4 Hyper-parameter Tuning

We will use the grid-search technique to find the best combination of hyper-parameters and observe emerging trends to support our configuration decisions.

This will be implemented using *sklearn*'s *GridSearchCV* constructor which evaluates a given model on hyper-parameters specified in a predefined dictionary using a default 3-fold validation on each combination of parameters. Although *GridSearchCV* optimizes the accuracy of the model, other metrics may be specified in the *scoring* argument. This will be further elaborated upon in chapter 6.4.1.

To optimize our model using *GridSearchCV*, we will first create a builder function which take the parameters we wish to tune as arguments and returns our desired *Keras* model. The builder function will then be passed to a wrapper *KerasClassifier* class which we will use as input to *GridSearchCV*. Note that for reproducibility, we have used *numpy*'s *seed* method with $seed = 7$ to ensure that the folds being generated by *GridSearchCV* remain constant.

To improve time efficiency, we will perform grid-search on a subset of the entire data-set. The selection of the subset will be done using the training indices after calling the aforementioned *StratifiedKFold* which ensures that each subset has an equal number of label examples in the subset. Although this may not be the most accurate representation of the network performance, it will give us a general idea of how our chosen configurations will perform.

5.3 Training Configuration Selection

5.3.1 Batch-size and Epoch

Epoch/Batch	*1*	*64*	*128*	*256*
5	0.915 (0.022) [21]	0.854 (0.052) [7]	0.702 (0.080) [5]	0.619 (0.017) [4]
15	0.912 (0.026) [21]	0.912 (0.052) [7]	0.892 (0.038) [5]	0.842 (0.045) [4]
20	0.912 (0.033) [21]	0.929 (0.030) [7]	0.911 (0.027) [5]	0.899 (0.017) [4]

Table 5.1: Single-Label Network Accuracy, (std. dev) and [average training time/epoch(sec)] comparison using different epochs and batch sizes

To configure the batch size and number of epochs, we have trained a single layered spectral CNN model shown in figure 5.6 with one fully connected layer using *GridSearchCV*. Table 5.1 shows a comparison of the network accuracy, variance and training time per epoch for each combination of batch size and number of epochs. Note that these experiments were run using the standard Adam optimizer.

As discussed in chapter 2.6.1, our experiment yielded expected results. As we increase the batch size the variance of the results decrease due to the weights being updated using gradients averaged over more samples leading to better estimates. The computational time per epoch also decreases since weights are updated less frequently. The latter also explains why the accuracy decreases as we increase the batch size.

We will use a batch size of 256 and 20 epochs for the experiments going forward. This was done not only to reduce the variance in our experimentation results but also due to the training time being batch size dependent. Using the same training data-set size, we can consequently estimate each grid-search experiment time using equation 5.3. Note that although training times are heavily dependent on the processing unit, we expect their relative differences across different processors to remain approximately constant. Further analysis has been conducted in chapter 6.5.3.

$$\begin{aligned} Time[min] &= \#combinations \times \#folds \times \#epochs \times time/epoch[min] \\ &= \#combinations \times 4[min] \end{aligned} \tag{5.3}$$

5.3.2 Optimization Algorithm

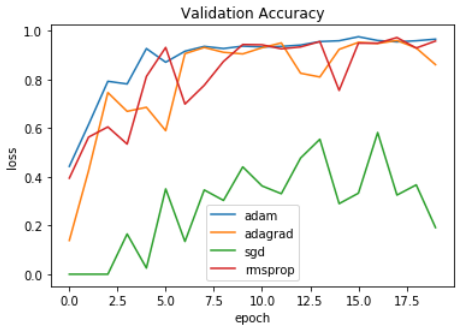


Figure 5.2: Validation Accuracy due to different optimization algorithms
(Figure created using Python)

Using the previously selected batch and epoch combination, we now evaluate the performance and behaviours of different optimization algorithms. To do this we will train the network and analyze the training and validation accuracy over 20 epochs. The results seen in figure 5.2 supports our research in chapter 2.6.2, where the optimizer can be seen to produce superior results. Due to weight updates being performed based on the first and second moments of the gradients, we can observe a significantly less noisy path and faster overall convergence. The noisy path typical of the SGD optimizer can also be clearly seen. We will therefore select the ADAM optimizer to train our network going forward.

It is interesting to note that since RMSProp was developed to counter the diminishing learning rates of Adagrad we initially expected to see RMSprop outperform Adagrad. However, since the network did not take many epochs to converge, we can see that two optimizers had similar performances.

5.4 Classifier Design Development

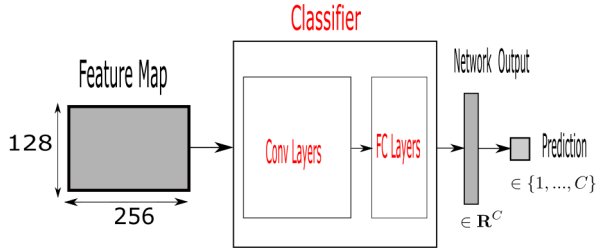


Figure 5.3: Single-Label Classifier Overview
(Figure created using Inkscape)

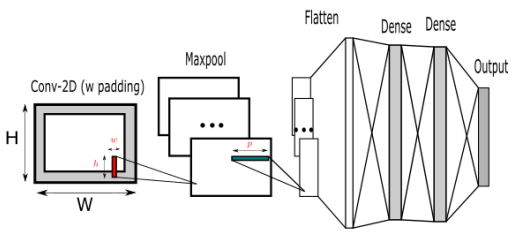


Figure 5.4: Single-Label CNN Architecture
(Figure created using Inkscape)

After defining our input transformation and output requirements as described in chapter 5.1 we can now formalize a problem statement for our classifier design. As depicted in figure 5.3, our classifier will be designed to map (128×256) dimensional features onto a C-dimensional

output space where $C = 7$ is the total number of possible output labels as described in chapter 4.3.1. The internals of the classifier design will be chosen and updated using the aforementioned learning algorithm to minimize the Softmax loss function.

For reasons as described in chapter 2.4.3, we have chosen a CNN architecture as our primary single-label classifier. The overall network will be implemented using the *Sequential* constructor on the *Keras*API and built using the *layers* constructor.

The training configurations as selected in chapter 5.3 will be defined in the arguments of the *compile* method of the *Sequential* constructor. Also note that while we have followed the standard practice of using hidden layer ReLU activation functions for the following experiments, its performance will be tested against other activation functions in chapter 5.4.3.

5.4.1 Convolutional Layer

Filter Dimensions

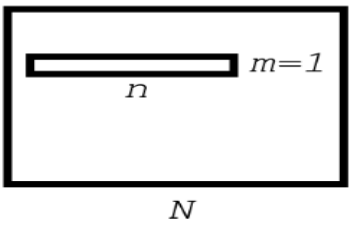


Figure 5.5: Temporal Filter Design

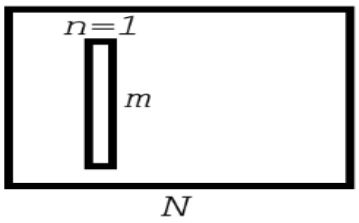


Figure 5.6: Spectral Filter Design

Inspired by the musically motivated filter dimensions discussed in chapter 2.4.3 and building off the Time-Frequency Architecture proposed by [36], we will decompose and experiment with filter dimensions based on their contextual meaning for audio processing. A general CNN architecture for the following experiments is depicted in figure 5.4.

Temporal Filters

Temporal filter designs as depicted in figure 5.5 specialize in learning rhythmic pattern. First, we set $m = 1$ and train our model with $n \in \{65, 129, 256\}$ which roughly corresponds to a quarter, half and the entire temporal range. For completeness, we will also use the standard $n = 5$ and conduct tests on the same set of n values with input padding as described in chapter 2.4.3. A pooling layer which chooses the maximum value across the spectral range at each temporal unit is then applied to the output of the convolutional layer for dimension reduction. This produces an output vector which is flattened and passed through a fully-connected network with one hidden layer and 128 neurons.

Filter width	*5*	*65*	*129*	*256*
No Padding	0.213 (0.098)	0.275 (0.125)	0.249 (0.109)	0.130 (0.121)
Padded	0.242 (0.079)	0.314 (0.132)	0.305 (0.099)	0.283 (0.052)

Table 5.2: Single-Label Network Accuracy and (std. dev) comparison using different filter widths and padding

From table 5.2, we can see that the temporal filters performed relatively poorly across all filter widths. This result supports our back-ground research which suggests that the timbral properties spread across the spectral range are the primary means of instrument sound recognition. Since we have set our filter height across the spectral range to one, this would imply that our model should not be able to learn any timbral information.

Conversely, the model still performs better than a random classifier which would either suggest that there are some useful temporal information for instrument classification or that the model was able to learn a portion of the timbral information in the fully connected layers.

Furthermore, we can also observe that the use of input padding improved the network performance across the entire set of filter heights.

Spectral Filters

Spectral filter designs in figure 5.6 specialize in learning spectral information such as timbre. Here, we set $n = 1$ and using the same ratios as the temporal filter dimension selection process we obtain the set of m values given by $m \in \{5, 33, 65, 128\}$. Max-pooling will now be applied to the filter outputs which chooses the maximum value across the temporal range at each spectral unit followed by the same process as before. Similarly, we will test the set of n values with and without padding.

Filter height	*5*	*33*	*65*	*128*
No Padding	0.894 (0.021)	0.893 (0.046)	0.845 (0.042)	0.599 (0.137)
Padded	0.896 (0.040)	0.903 (0.021)	0.890 (0.032)	0.859 (0.049)

Table 5.3: Single-Label Network Accuracy and (std. dev) comparison using different filter heights

Table 5.3 shows the results of the spectral filter model and as expected, it significantly out performs it's temporal counterpart. Similarly to the previous results, the network performs better for a padded input and we also obtain the best results when the filter height is set to approximately a quarter of the spectral range. We will therefore select a $[1 \times 33]$ filter dimension configuration with input padding.

Hyper-parameter Tuning

Layers/Filters	*16*	*32*	*64*
1	0.807 (0.113)	0.885 (0.033)	0.886 (0.022)
2	0.847 (0.029)	0.878 (0.045)	-
3	0.752 (0.125)	-	-

Table 5.4: Single-Label Network Accuracy and (std. dev) comparison using number of conv. layers and filters

Using our chosen filter dimension, we will now perform grid search to select the optimal number of filters and convolutional layers. Due to hardware constraints, we will not consider the cases as marked in table 5.4. Note that the the hidden convolutional layer filters will have the same dimensions as the first layer but they will not be padded.

Comparing our results, we can conclude that using 32 and 64 filters significantly outperforms the 16 filter network regardless of the number of hidden layers used. However, it is not immediately obvious from the table which of the remaining three configurations we should select.

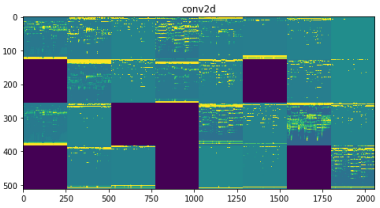


Figure 5.7: Network 1 (32 filter, single layer CNN): Layer one Output activations

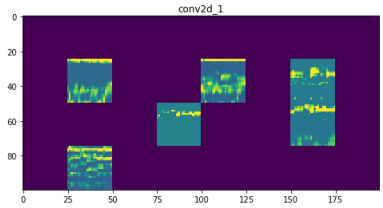


Figure 5.8: Network 2 (32 filter, double layer CNN): Layer two Output activations

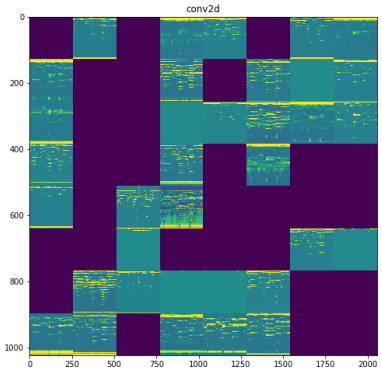


Figure 5.9: Network 3 (64 filter, single layer CNN): Layer one Output activations

As described in chapter 5.2.3 the intermediate activations of all the filters on a layer are gridded and displayed. From figures 5.8 and 5.9, we can see that the activations contain many dead filters which may be a symptom of overly complex networks for a task. Therefore, we will use a single convolutional layer with 32 filters for our design.

5.4.2 Fully-Connected Layer

Layers/Neurons	*64*	*128*	*256*	*512*
1	0.904 (0.037)	0.923 (0.028)	0.946 (0.025)	0.944 (0.024)
2	0.932 (0.029)	0.933 (0.045)	0.952 (0.023)	0.949 (0.034)
3	0.934 (0.022)	0.932 (0.044)	0.950 (0.030)	0.941 (0.025)

Table 5.5: Single-Label Network Accuracy and (std. dev) comparison using number of fully connected hidden layers and neurons

The standard grid-search technique was performed here and our chosen configuration will be a double layered fully connected architecture with 256 neurons in each layer.

5.4.3 Network Hyper-parameters

Dropout Regularization

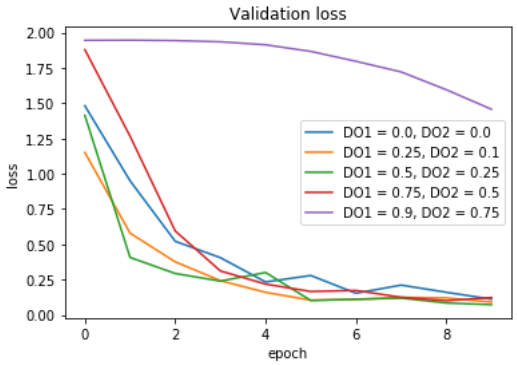


Figure 5.10: Validation Loss where $DO1 > DO2$

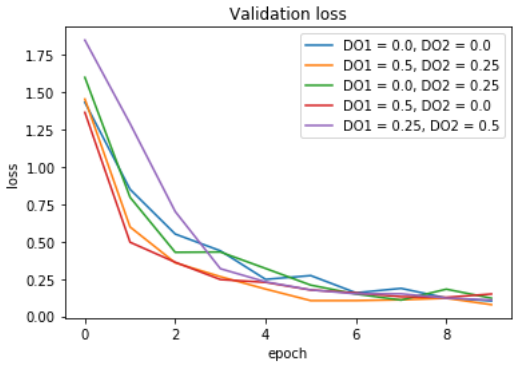


Figure 5.11: Validation Loss for different combinations of DO1 and DO2

We will now implement a dropout layer after flattening the filter output (Dropout 1) and another before the output layer (Dropout 2). To observe their effects on the network, we will compare the network validation loss for different dropout values.

Since the input to dropout 1 contains 3072 features compared to 128 neurons connected to dropout 2, we will set dropout 2 to a higher value compared to dropout 1. From figure 5.10, we see that the loss is minimized at a faster rate for configurations $(DO1, DO2) \in \{(0.25, 0.1), (0.5, 0.25)\}$. Then as we continue to increase the dropout rate, the training process performs worse due to insufficient information propagating through the network.

Now we select $DO1 = 0.5$ and $DO2 = 0.25$ and plot the loss for different configurations. The plot in figure 5.11 justifies our initial decision for $DO1 > DO2$ since when we switch the dropout rates and set $DO1 = 0.25$ and $DO2 = 0.5$, the training process to performs worse than had we not implemented any dropout layers.

Lastly, when we hold the dropout rate of one layer constant and remove the other, we can see that the first dropout layer is more significant compared to the second. This again justifies our decision to set $DO1 > DO2$.

Activation Functions

Activation	***Linear***	***Tanh***	***Sigmoid***	***ReLU***
Performance	0.939 (0.020)	0.901 (0.023)	0.455 (0.123)	0.942 (0.021)

Table 5.6: Single-Label Network Accuracy and (std. dev) comparison using different types of activation functions

Our model was then trained with different types of activation functions. Our results in table 5.6 displays some surprising results. Although it was expected that the ReLU activation performed the best, it is interesting to note that the performance of the network using linear activations produced almost identical results.

As described in chapter 2.5.3, activation functions are primarily used to introduce non-linearity into the network. Our results would therefore indicate that a large number of labels in our data-set are linearly separable. Furthermore, since it can be shown that an N layered network with linear activation functions can be represented by a single layer, this would also explain why a relatively shallow network design was sufficient for our given problem. However, due to the slight performance improvement when ReLU was used, we will use it for our final network architecture.

5.5 Results and Reflection

Using the results and analysis in chapter 5.4, we have selected a feed-forward CNN architecture shown in figure D.2. To evaluate it's performance, we trained our model on 80% of the data-set and tested it on the remaining 20%. Furthermore, to validate our results, we conducted a 10-fold cross validation analysis on the entire data-set which resulted in a 98.5% accuracy.

	cla	dru	flu	gui	pia	vio	voi
cla	250	0	10	0	0	0	0
dru	0	256	0	0	0	0	0
flu	5	0	255	0	0	0	0
gui	0	0	1	257	2	0	0
pia	0	0	0	0	260	0	0
vio	0	0	1	1	0	257	0
voi	2	0	1	2	0	1	256

Table 5.7: Single-Label Network Confusion Matrix

The network performance on each class label of the testing data-set can be seen by the confusion matrix in table 5.7, where the vertical axis represents the true label and the hoizontal axis represents the predicted labels.

Our results show that the network performs well on the majority of instruments. We can also notice that the majority of incorrect predictions occur between the flute and clarinet. These two instruments belong to the same woodwind family of musical instruments and could indicate that this family of instrument have underlying properties too complex for our model to identify.

Conversely, the string (violin and guitar) and percussion (drum and piano) family of instruments have significantly better predictions. This makes sense since instruments are grouped into families depending on a variety of factors, resulting in sounds being produced in comparable ways within each family. Our results would suggest that we could use our classification technique as a measure of similarity between different types of instruments.

Additionally, we notice that the voice had the widest spread of incorrect predictions. This is expected since the voice is the most complex type of sound being produced which means that it could have the ability to mimic the dominant properties used to identify the other instruments.

5.5.1 Comparisons with Past Works

Model	Acc	Num Inst	Dataset
	0.96	4	RWC
Diment et al.(2013)	0.84	9	RWC
	0.70	22	RWC
yu et al.(2014)	0.96	10	ParisTech
Yip et al.(2017)	0.96	18	MedleyDB multi-track
Final model (2019)	0.99	7	MedleyDB multi-track

Table 5.8: Single-Label comparison with past works

5.5.2 Multi-Label Extension

Prefacing chapter 6, we had initially planned an experimental process to analyze the performance of the model architecture described in this chapter trained on single-labeled data and tested on a multi-labeled data-set. However, we were unable to find an appropriate multi-labeled data-set which had the same combination of instruments as our single-labeled data-set. This will therefore be left for future works.

Chapter 6: Problem B - Multi-Label Classification

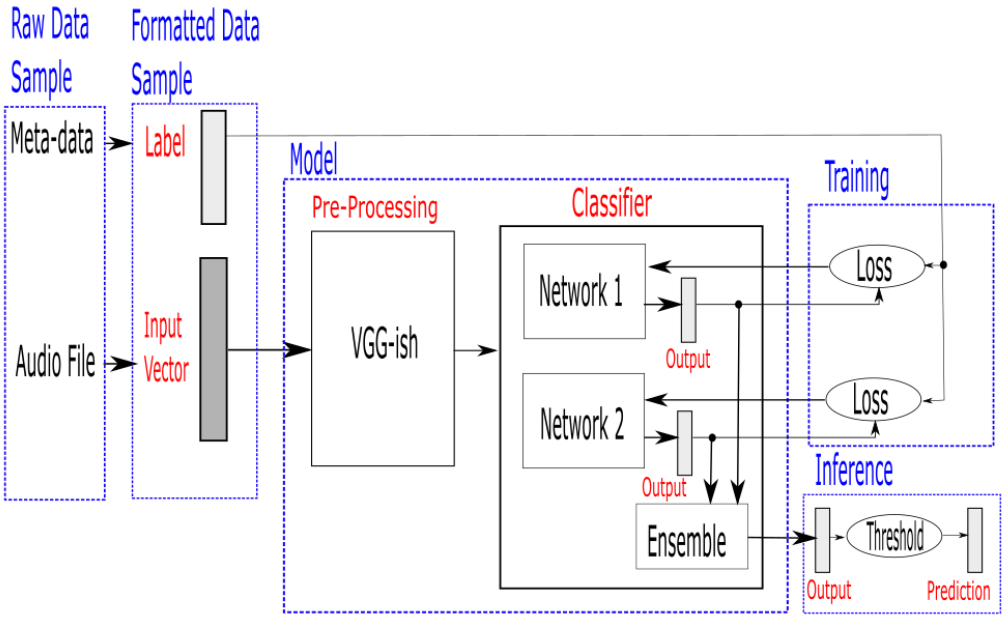


Figure 6.1: Problem B Process flow for a single data sample (Figure created using Inkscape)

6.1 Network Peripherals

6.1.1 Input Features

In this problem, input to our classifier is the 128-dimension embedding VGG features. To obtain the VGG features which is also the output of VGGish model, we will first pre-process the audio files of OpenMIC-2018 dataset. Then, apply VGGish model to get the embedding features.

Pre-processing

To pre-process the audio files: First, the audio should be resampled to 16kHz mono. Then, the process is similar with calculating Mel-spectrogram explained in chapter 6.1. Compute the spectrogram using STFT with a window size of 25ms, a window hop of 10ms, and a periodic Hann window. Then the spectrogram would go through 64 Mel-filter banks covering the frequency range $125 - 7500$ Hz to create a Mel-spectrum. To have a stabilized figure, function "$\log(\text{melspectrum} - 0.01)$" would be applied to have a log scale Mel-spectrogram. The offset value 0.01 can avoid computing "log(0)". These features can then framed into 0.96s long non-overlapping signals. Each 0.96s signal includes 64 Mel-bands and 96 frames of 10ms. The output for a 10 seconds long input after these processes are a three dimension array with shape [number of examples(10), number of frames(96), number of Mel-bands(64)]. This output indicates a sequence of segments; each segment includes a cluster of log Mel-spectrogram values. For each sequence, the size is [number of frames(96), number of Mel-bands(64)].

Obtaining VGG features

The array with size [96x64] could be inputted into the VGG model to have an embedding 128-Dimension value. Each training data from Openmic-2018 is 10 seconds long. Hence, the VGG features of an Openmic-2018 training data would have the size of [10x128]. However, the ten in size means the number of frames; each frame is 0.96s. Hence, there would be 0.4s audio in total being discarded during the VGG feature extraction. In addition, the post-process to the features are PCA and quantization. To apply PCA, the embedding matrix would be transposed and subtract the PCA mean column vector from each column. Then the matrix would be premultiplied by PCA matrix in size [output dimension(128), input dimension(128)]. Next, we would transpose the matrix again. In the end, quantizing is required. Taking the maximum and minimum value among the matrix, then zoom all the numbers into the scale [0.0, 255.0].

Generally, for each Openmic-2018 training data in 10 seconds. The output VGG features would be in size of [10x128], and every value in the matrix is in the range [0.0 , 255.0]. Using the transfer learning, we could use these features resulting from a neural network, which is the learned knowledge from a VGG network. Then we could connect the VGGish feature to the input of our neural network classifier.

6.2 Network Loss Design Development

6.2.1 Trivial Example

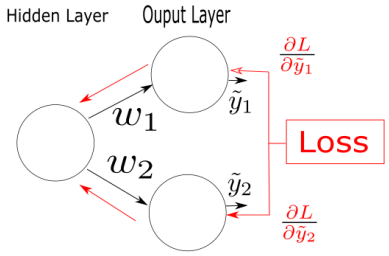


Figure 6.2: NN with two output neurons and one hidden layer neuron(Figure created using Inkscape)

To motivate the use of our proposed loss function, we consider the multi-label classification problem shown in figure 6.2 with two possible output classes $\{y_1, y_2\}$, and a single neuron with a Sigmoid activation in the hidden layer. Then W_1 and W_2 denote the weights connecting the hidden unit to the first and second output nodes respectively. The output predictions for each output neuron is therefore $\tilde{y}_1 = \sigma(W_1 a)$ and $\tilde{y}_2 = \sigma(W_2 a)$.

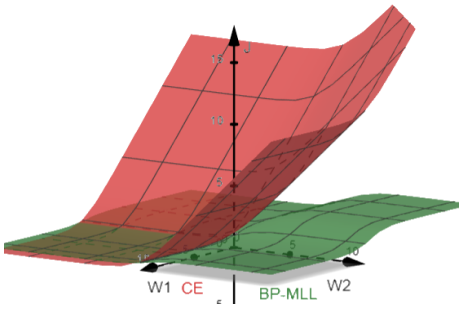


Figure 6.3: Loss landscape of BP-MLL and Cross-entropy (Figure created using geogebra)

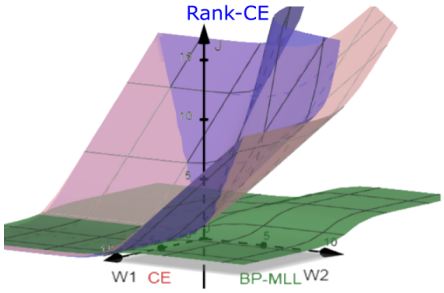


Figure 6.4: Loss landscape with Rank Cross-entropy (Figure created using geogebra)

Existing Multi-Label Loss Functions

Using binary cross entropy, the overall loss given by equation A.6 has partial derivatives (6.1) which depend only on the loss due to the binary problem in its on output neuron. Since output layer weight updates given by equation B.4 are performed using the upstream partial derivatives flowing back from the output neuron they connect to, output layer weight updates due to CE loss only depends on the prediction of the output neuron it connects to. This implies that the output layer can therefore be decomposed into separate parallel sub-networks which do not account for label dependencies.

$$\begin{aligned} \frac{\partial L_{CE}}{\partial \tilde{y}_1} &= -\frac{1}{\tilde{y}_1} \\ \frac{\partial L_{CE}}{\partial \tilde{y}_2} &= \frac{1}{1-\tilde{y}_2} \end{aligned} \tag{6.1}$$

Using BP-MLL and fixing the output labels, the loss function (A.5) reduces to $L_{bp} = e^{-(\tilde{y}_1 - \tilde{y}_2)}$ where the set with true value value 1 and its compliment contain one neuron each. Unlike the previous case, the partial derivatives (6.2) with respect to one output also depends on the predictions of the other output neuron.

$$\begin{aligned} \frac{\partial L_{bp}}{\partial \tilde{y}_1} &= -e^{-(\tilde{y}_1 - \tilde{y}_2)} \\ \frac{\partial L_{bp}}{\partial \tilde{y}_2} &= e^{-(\tilde{y}_1 - \tilde{y}_2)} \end{aligned} \tag{6.2}$$

Now, we input an example into the network with true labels $y_1 = 1$ and $y_2 = 0$. Then we can reproduce the loss landscape with respect to its weights as described by [20]. From figure 6.3, we can clearly see the steep gradients of the CE loss and the plateaus of the BP-MLL loss which causes networks trained to minimize it to perform poorly compared to those trained to minimize the CE loss.

Custom Loss Function: Rank-CE

To capture dependencies in the output layer and avoid the plateau effect mentioned previously, we took the product of both losses which uses the BP-MLL function to stretch or shrink the CE loss based on the overall network performance. Equation 6.3 shows the rank-CE loss which results in an adaptive gradient being propagated back from each output neuron based on the overall performance of the network. Furthermore, figure 6.4 shows that the loss is still able to retain steep gradients for weight updates.

$$L = -0.5(\log \tilde{y}_1 + \log 1 - \tilde{y}_2)e^{-(\tilde{y}_1 - \tilde{y}_2)} \tag{6.3}$$

Adaptive Gradients

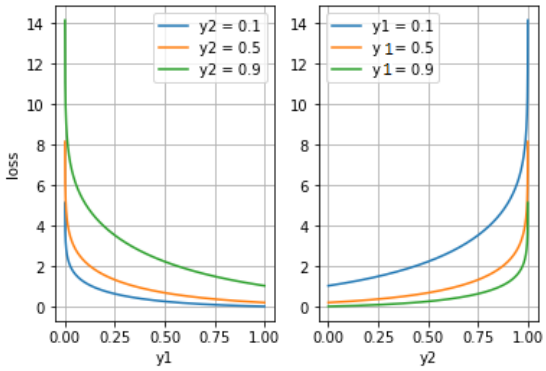


Figure 6.5: Network loss (Figure created using python and matpoltlib)

The adaptive gradients effect is illustrated in figure 6.5 where the Rank-CE loss function 6.3 for the above example is plotted for different output predictions. Note that since we have fixed the true labels at each output neuron to $y_1 = 1$ and $y_2 = 0$ this means that network output predictions would ideally be $\tilde{y}_1 = 1$ and $\tilde{y}_2 = 0$ In the first case, we fix the output predictions of the second neuron to $\tilde{y}_2 \in \{0.1, 0.5, 0.9\}$ and observe the loss across the range of output predictions for the first neuron ($\tilde{y}_1$). We can see that as the performance of the second neuron gets worse (i.e $\tilde{y}_2$ increases), the loss function gets stretched vertically, increasing the gradients for all possible output predictions at the first neuron ($\tilde{y}_1$). Looking at the case when $\tilde{y}_2 = 0.9$, we can see that even if the first neuron were to make a perfect prediction of $\tilde{y}_1 = 1$, there would still be a loss induced due to the other neuron performing poorly.The inverse effect can be observed in the second case.

6.2.2 Rank-CE: General Case

This was extended to the case where there are C output classes. Consider a training example with $|Y|$ output neurons with true label 1 and $|\tilde{Y}|$ output neurons with true label 0, where $|.|$ is the carnality of that set. Then set $Y = \{y_1, ..., y_{|Y|}\}$ and set $\bar{Y} = \{\bar{y}_1, ..., \bar{y}_{|\bar{Y}|}\}$. Then the overall network loss is given by:

$$L = \sum_{(p,n)\in(Y\times\bar{Y})} \frac{\exp^{-(y_p-\bar{y}_n)}}{C|Y||\bar{Y}|} \left(\sum_{p\in Y} -log(y_p) + \sum_{n\in\bar{Y}} -log(1-\bar{y}_p) \right) \tag{6.4}$$

The loss associated with an output neuron y_i in set Y is given by:

$$L_p = \frac{-R_{\bar{y}}}{C|Y||\bar{Y}|} \left(R_y log(y_i) + e^{-y_i} \log(y_i) + e^{-y_i} D \right) \tag{6.5}$$

$$R_{\bar{y}} = \sum_{\bar{y}_j \in \bar{Y}} e^{\bar{y}_j} \tag{6.6}$$

$$R_y = \sum_{y_k \in Y, y_k \neq y_i} e^{-y_k} \tag{6.7}$$

$$D = \sum_{y_k \in Y, y_k \neq y_i} \log(y_j) + \sum_{\bar{y}_j \in \bar{Y}} \log(1 - \bar{y}_j) \tag{6.8}$$

We can then calculate the gradient for that neuron:

$$\frac{\partial L}{\partial y_i} = \frac{-R_{\bar{y}}}{C|Y||\bar{Y}|} \left(\frac{R_y}{y_i} + \frac{e^{-y_i}(y_i log(y_i) - 1)}{y_i} + e^{-y_i} D \right) \tag{6.9}$$

Similar steps can be taken to obtain the gradients for the neurons in set $\bar{Y}$ given by:

$$\frac{\partial L}{\partial \bar{y}_i} = \frac{R_y}{C|Y||\bar{Y}|} \left(\frac{R_{\bar{y}}}{1 - \bar{y}_i} + e^{\bar{y}_i} \left(\frac{1}{1 - \bar{y}_i} - log(1 - \bar{y}_1) - e^{\bar{y}_i} D \right) \right) \tag{6.10}$$

These gradients are then propagated back through their respective neurons and multiplied by the local gradient for parameter updates in the preceding layers. Since *Keras* does not currently have any primitive functions implementing BP-MLL, we will implement it using the *Keras* backend based on [80] and the primitive *cross_entropy* loss function. Further analysis on networks trained to minimize Rank-CE loss will done is chapter 6.4.4.

6.3 Classifier Design Development

The classifier for this problem would be an ensemble model, which was explained in chapter 2.7.3. We will train a CNN-based model and a RNN-based model separately as the basic classifier. To implement the final classifier, we will using stacking method mentioned in chapter 2.7.3 to combine the CNN and RNN models.

6.4 Network 1: CNN-based

6.4.1 Training Process

The network was trained in a similar way to the process as described in chapter 5.2 using *fit*. To give ourselves more freedom in implementing the network architecture, we will build our model using the *Model* constructor on *Keras* instead of the *Sequential* constructor from the previous section. This allows us to create multiple input sources, parallel channels through the network and perform concatenation on different layer outputs.

Metrics

As described in chapter **??**, we cannot rely purely on the prediction accuracy to evaluate our network and will therefore use the f1-score going forward. Since the f1-score is threshold dependent, we will also implement a thresholding metric function using the *Keras* back-end which calculates the f1-score of a model for different threshold levels.

Furthermore, we have also discovered that the *sklearn* module does not contain many primitive functions to evaluate multi-label performance metrics. We have therefore written a *Multi_Label_Metrics* class which plots the Receiver-Operator Curves, Precision-Recall Curves and class probability distributions for multi-label predictions.

Parameter Tuning

Since *GridsearchCV* is unable to score multi-labeled datasets, we will define a custom f1-scoring function and use a wrapper *make_scorer* constructor to pass it to *GridsearchCV*. Furthermore, since *StratifiedKFold* is currently unable to handle multi-labeled data we will use the entire data-set during the grid-search process.

6.4.2 Architecture

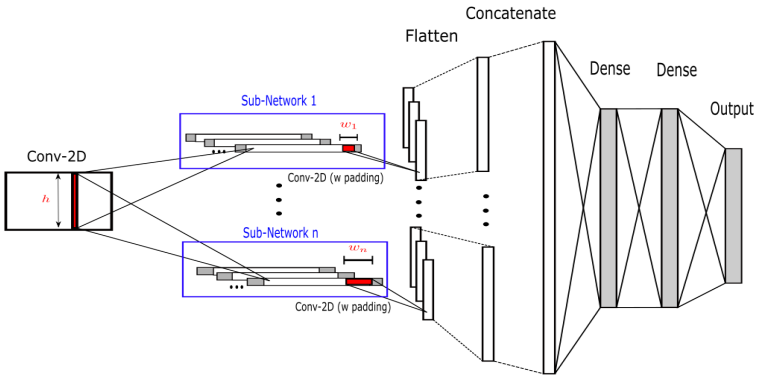


Figure 6.6: Multi-Label Network 1 Architecture (Figure created using Inkscape)

Inspired by the use of CNN for NLP and sentence classification [Kim et al.(2014)], we will apply the same principles on the input feature map where instead of fixed length embeddings for each word in a sentence, we have fixed length feature embeddings at each time step of our signal.

As depicted in figure 6.6, the input layer filters will connect to all features at each time step and convolve across the temporal domain at discrete time steps. The input to the second layer will therefore be 1D feature arrays of fixed lengths. Since the convolutions in the first layer were time-invariant, we will use the filters in the second layer to capture time dependencies.

Filter Dimensions

We will first fix the dimensions of all F_1 filters in the first layer to $[128 \times 1]$, where F_1 is the number of filters used in the first layer. We can then calculate the dimensions of the first layer output activations as described in 2.4.3. This results in $[1 \times 10 \times F_1]$ dimensional outputs corresponding to F_1 depth-wise features per time step. The second layer filters will therefore be of dimension $[1 \times w]$, where the filter width (w) corresponds to the range on the temporal axis which the filter connects to at each convolution step.

We then use the grid-search process as described in 6.4.1 to evaluate the performance of using different filter widths. Furthermore, we will retrain the network with a padded second layer to see if the results in chapter 5.4.1 can be reproduced.

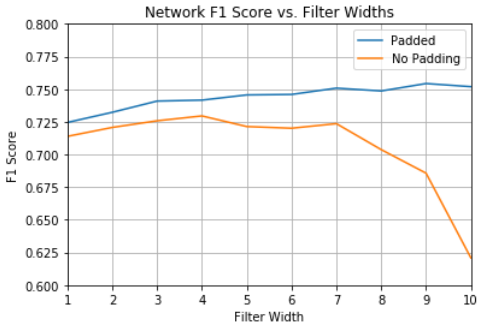


Figure 6.7: Plot of Network F1 scored for different filter widths (Figure created using Python)

As depicted in figure 6.7, our experimental results produced several interesting trends. Consistent with the results in the previous chapter, padding the input to a layer improves the network performance, which can be seen across all filter widths.

The results for the input without padding shows similarity to that in 5.4.1 where the performance displayed an inverse parabolic behaviour. However, the effect of padding in this model results in an exponential improvement in network performance with respect to the filter widths, resulting in near linear performance.

Furthermore when we analyze the behaviour of the network with padding, we observe that apart from $w = 1$, even numbered filter widths displayed a subtle downward trend compared to the odd numbered filter widths.

Sub-Network Configurations

Using $w = 9$ with input padding, we conduct grid search to configure the number of filters in the first and second convolutional layer.

Next, we evaluate the network using different combinations of parallel sub-networks as seen in figure 6.6. Each sub-network will be set to a different filter and the outputs of each sub-

network will be flattened and concatenated. As per our results in the previous section, we will conduct our experiments using odd numbered filter widths with padded inputs.

Width1/Width2	***3***	***5***	***7***	***9***
3	0.789	0.787	0.789	0.787
5	-	0.788	0.787	0.789
7	-	-	0.780	0.794
9	-	-	-	0.787

Table 6.1: CNN double input channel: F1 Score comparison using different filter widths

First, we conduct grid search using two parallel channels to find the best filter width combination. Since the channels are parallel, we can expect the results in table 6.1 to be approximately symmetric. Although our results show that using two channels improves the network performance across all filter widths, there does not seem to be significant differences in the relative improvement which may suggest that the combination of filter widths is not that significant. However, the combination of $w_1 = 9$ and $w_2 = 7$ does slightly better than the rest and we will use that combination for the following experiment.

Channels/Width	***3***	***5***
3	0.812	0.799
4	-	0.785

Table 6.2: CNN multiple input channel: F1 Score comparison using different number of channels

Next we will evaluate the results of adding additional channels. Shown in table 6.2, we will choose $w_3 = 3$ for the third channel filter width. Since the score decreases when the fourth channel is introduced, we will use three sub-networks with filter widths $w_i \in \{3, 7, 9\}, i = 1, 2, 3$ for our network design.

Parameter Tuning

L1 Neurons/L2 Neurons	***64***	***128***	***256***
64	0.820 (0.011)	0.818 (0.009)	0.816 (0.013)
128	0.820 (0.011)	0.821 (0.012)	0.817 (0.014)
256	0.822 (0.014)	0.818 (0.015)	0.816 (0.009)

Table 6.3: Multi-Label two layer CNN F1-Score and (std. dev) comparison using different combinations of neurons

Using the convolutional layer configuration as described above, we ran grid-search on a a network with two fully connected hidden layers to analyze the effect different combinations of

the number of neurons had on it's performance. Table 6.3 shows the results of our simulation where the F1 score due to different combinations of number of neurons is the first layer (L1 Neurons) and second layer (L2 Neurons) is shown.

Our results indicates that at almost every case, the network's performance decreases when the number of neurons in the second hidden layer was greater than the first layer.

6.4.3 Final Architecture and Results

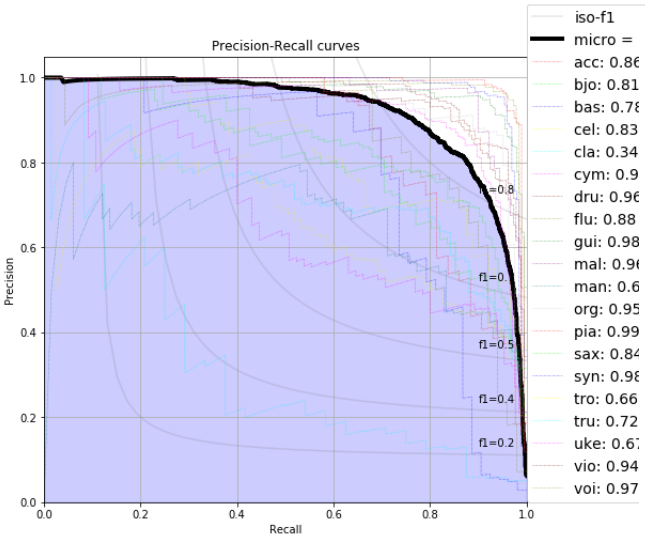


Figure 6.8: PR Curves for Instruments (Figure created using Keras)

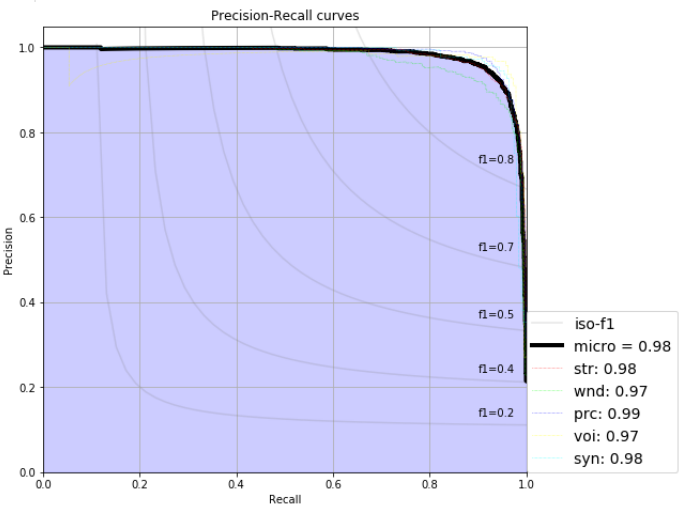


Figure 6.9: PR Curves for Instrument families (Figure created using Keras)

Using the analysis done in the previous section, our final CNN architecture for network 1 contains 713, 925 tunable parameters and will be as depicted in figure D.3. The model was trained on the data-set as described in 4.3 and the Precision-Recall (PR) curves of each instrument and the micro averaged curve (given by the bold black curve) was plotted using our custom built python class. Note that the iso-F1 curves were also plotted for better visualizaion of the model's performance.

Figure 6.8 shows our network's performance on 20 instrument labels. Although the micro-averaged PR curve shows that the model performs well in general, when we look at the individual curves we see that several labels performed very poorly. This was be seen by the Area Under the Curve (AUC) values printed on the side of the PR plot. An inspection of the data-set revealed that the labels which performed poorly coincided with those with fewer training examples.

We therefore, restructured the given data-set to classify musical instrument families instead of instruments. This was also done to justify our claim from the first problem regarding

instruments from the same family being harder to classify due to sounds being produced in more comparable ways. The classification performance for musical instrument families can be seen in figure 6.9 where we can see the performance improve drastically for all families of instruments. The voice and synthetic data were not assigned to a family and hence had a smaller training set relative to the string, woodwind and percussion families. However, we notice that their performances did not decrease which would indicate that the relative size of each label in a data-set does not have as large an impact on the overall performance as the actual size of the label in the data-set.

6.4.4 Loss Comparison

Our CNN network design as depicted in figure D.3 will now be used to compare the effect of minimizing different types of loss functions on network performance. Our proposed Rank-CE loss will be evaluated against the existing BP-MLL and CE options.

Loss	***BP-MLL***	***CE***	***Rank-CE***
Threshold Bias	0.731	0.831	0.831
No Threshold Bias	0.730	0.822	0.831

Table 6.4: F1 scores comparison using different loss functions, with and without threshold bias

Table 6.4 shows the average network f1-scores after after a 10-fold cross validation analysis. The first row shows the network performance where a range of F1 scores are computed based on different threshold values and the maximum F1 score is selected. The second row shows the network performance where there is no added bias during the inference phase which corresponds to a threshold of 0.5.

Our results supported the finding by [35], where the network trained using BP-MLL performed considerably worse compared to the one trained using CE loss. When a threshold bias was implemented, there was no difference in the F1 scores between the network trained using CE and Rank-CE. However when the bias was removed, the F1 score of the network trained using CE dropped while the other two network performances did not change significantly.

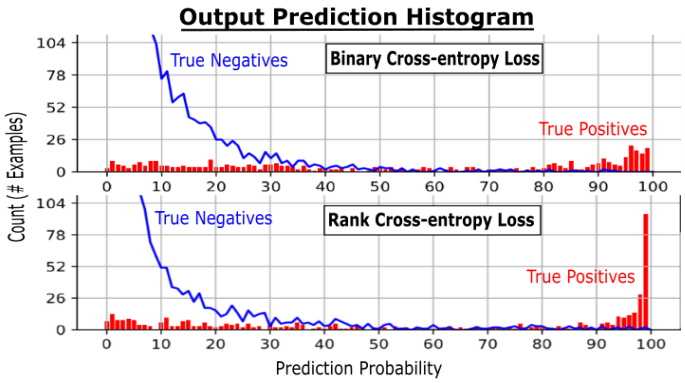


Figure 6.10: Plot of Network F1 scored for different filter widths (Figure created using Python)

To further explain our findings, we plotted a histogram of network output predictions on the testing data-set as shown in figure 6.10. Note that the true negative histograms have been plotted using a line for better visibility. From the figure, we can see that the true positive prediction probabilities of the network trained using CE loss appears to have a larger spread across the range of probabilities. Conversely, the true positive predictions of the network trained using Rank-CE is more skewed to either ends of the probability range. Similarly, we can see a skew in the true negative plot for the network using Rank-CE. However, this means that there are more true negative examples which are given higher probabilities. This can be seen when we compare both plots in the range $90 - 100$.

This explains why the network trained using the CE has a decrease in F1 score when no threshold bias is applied. Furthermore, the implicit biasing of the output probabilities displayed by the network trained using Rank-CE could be an effect of the the output layer not decomposing to separate output networks as discussed in chapter 6.2.

6.5 Network 2: RNN-based

Our second network architecture was designed by evaluating three different RNN-based models. The first is Bidirectional RNN, which was invented in 1997 by Schuster and Paliwal[67]. The second model is a self-designed Mean RNN model. The last is an Attention based bidi-RNN, inspired by Zhou's paper focusing in natural language processing in 2016[59].

6.5.1 Network Architecture

Basic LSTM Cell

All our RNN models are based on the LSTM structure to overcome the limitations of traditional RNN models as explained in chapter 2.4.4. Furthermore, the LSTM cell is the same for all mod-

els. To build an LSTM cell in Tensorflow, we need to use "tf.nn.rnn_cell.BasicLSTMCell(rnn_size,forget_bias
In our design, we choose the the number of units to be 200. We set forget bias to 2.0 which is the bias value for forget gate to decline the scale of forgetting in the fist stage of training. The structure of the implemented LSTM cell structure is shown in figure 6.11.

In LSTM cells:

$$\text{Forget gate}: \ f_t = sigmoid\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{6.11}$$

Forget gate (6.11) reads the previous output and current input to decide how much information can be stored or forgotten. If f_t is 1, the information is completely preserved. If f_t is 0, the information is totally forgot.

$$\begin{aligned} \text{Input gate}: \ i_t &= sigmoid\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \\ \text{Candidate cell state}: \ \tilde{C}_t &= \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \end{aligned} \tag{6.12}$$

Input gate (6.12) decides which value to be updated. $\tilde{C}_t$ has the range within [-1,1] indicates that cells state value need to strengthen or weaken in certain dimensions.

$$\text{Updated cell state}: \ C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{6.13}$$

The previous cell state times the output of forget gate to discard information. Input gate value has range from [0,1], with $i_t * \tilde{C}_t$, the updated cell state can be scaled down.

$$\begin{aligned} \text{Output gate}: \ o_t &= sigmoid\left(W_o\left[h_{t-1}, x_t\right] + b_o\right) \\ \text{Output}: \ h_t &= o_t * \tanh\left(C_t\right) \end{aligned} \tag{6.14}$$

The sigmoid in (6.14) decides which dimensions can be the output. The C_t after a tanh function can produce a Candidate output value.

The weighted matrices U, V and W are shared by all time steps, this is explained in chapter 2.4.4.

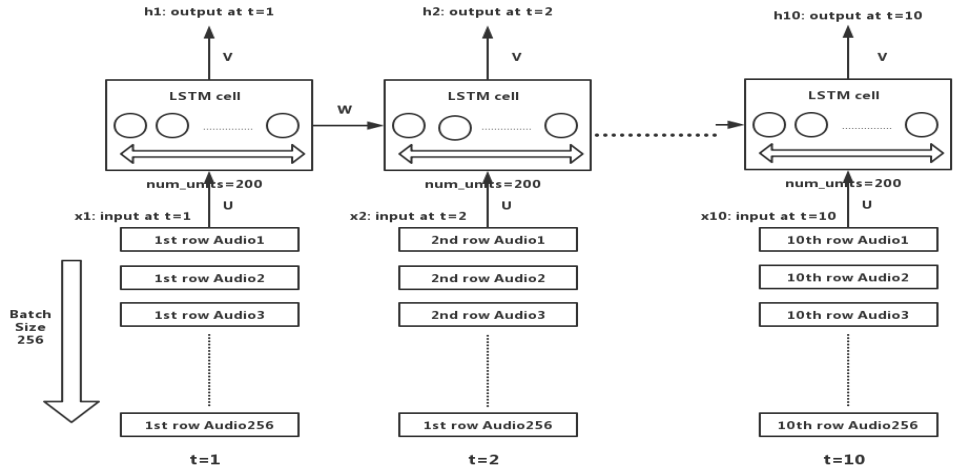


Figure 6.11: Implemented LSTM cell structure

Our input feature is VGG feature has the shape of [10×128], which means it has 10 time steps. At each time step, we will input 256(batch size) rows of 128-dimension data to LSTM cell. With number of units equal to 200, the output of a LSTM cell at a certain time step would be 200-dimension. After 10 time steps, we will input the whole features for a audio file into the neural network. The final output would be the output at last time step, in this case, the final output of LSTM cell would be output at "t=10".

Bidirectional RNN Model

Bidirectional RNN based on LSTM is our first model. Figure 6.12 introduces the model structure of BiRNN mdoel.

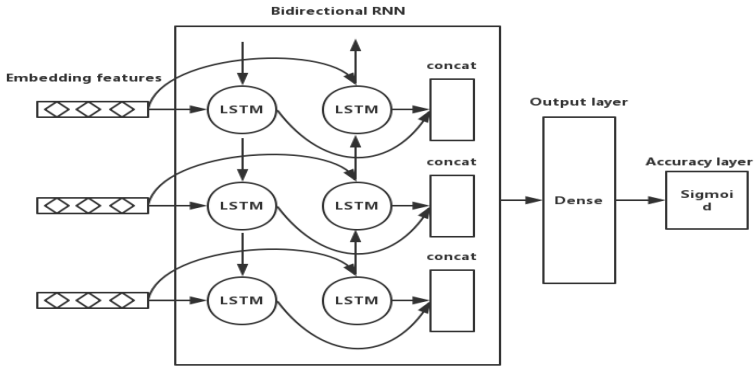


Figure 6.12: Bidirectional RNN Model Structure

- Bidirection RNN Layer:

we construct a forward cell and a backward cell to capture the information from both directions. After the every LSTM cell, we add a dropout layer with a probability of 0.8 to regularize our model. To combine two LSTM cells, we use "$tf.nn.bidirectional\_dynamic\_rnn$" in Tensorflow.

The forward LSTM would read the sequence in the order form x_1 to $x_{maxlength}$ (i.e. the maximum length of the sequence in our input training data is 10). Then, the hidden state sequence for forward LSTM can be computed as $\left(\vec{h}_1, \cdots, \vec{h}_{maxlength}\right)$. For the other LSTM, it would read the sequence in the opposite direction and calculate the hidden state sequence $\left(\overleftarrow{h}_1, \cdots, \overleftarrow{h}_{maxlength}\right)$. Therefore, for the specific row in our embedding features, for example, x_j the final annotation by integrating the information from the following and preceding vector row is in (6.15):

$$h_j = \left[\vec{h}_j^\top; \overleftarrow{h}_j^\top\right]^\top \tag{6.15}$$

To obtain the output of BiRNN layer, "tf.concat" from Tensorflow is used to concatenate the forward output and backward "state.h". The returned value from the function "$bidirectional\_dynamic\_rnn$" is (outputs, last_states), where the "last_states" is h_{10}, the hidden layer state value of final time sequence, the "outputs" is a large scale of matrix which contains the outputs values of all time steps.

look back to "$BasicLSTMCell$", the 'outputs' of this structure is the same as the 'outputs' of 'bidirectional_dynamic_rnn' which is the outputs of hidden layer state value for all time sequence. The difference is the 'last_states' of LSTM would return a tuple value, which contains two elements: '.c file' and '.h file', where the c file is the cell inner state of last moment, and the h file is the hidden layer state value of last moment. Hence, in our bidirectional RNN model, we would concatenate two '.h' file as the output. Following the concatenated output, we add a dropout layer to regularize the model.

- Output Layer:

we use "tf.layers.dense" function from Tensorflow to implement a dense layer with dense size equals to 400. the number 400 is the same as the dimension of BiRNN layer output. For each LSTM cell, the output dimension would be 200, with concatenated two LSTM cells, the output dimension would be 400. The activation function in dense layer is relu.

After the dense layer, we will use a softmax to convert 400-dimension data to probability for each time step in all labels. In this step, the softmax acts as a fully connected layer to scale down the dimension of data form 400 dimension to 20 dimension (the total number of instruments of OpenMIC-2018 dataset). Also, the softmax process is important to the calculation of cross-entropy loss function(Explained in following loss layer section).

- Accuracy Layer:

In the accuracy layer, we will do a sigmoid to the results of each classification calculation to determine whether the sample belongs to the same hypothesis of a certain category. This idea follows the Binary-Relevance method mentioned in chapter 3.3. The result of sigmoid function can be considered as the probability of the appearance of a specific instrument. Initially, we set the threshold probability value to be 50%, which means all instruments with probability greater than 50% would be taken as recognition result.

- Loss Layer:

In loss layer, we will use "tf.cast" to get the true label value Y of data and use 'sigmoid cross entropy with logits' loss function from tensorflow[60]. This loss function is specially designed for multi-label classification. The input to this function is the logits of network outputs and targets(correct label value). Logits is the $W \times X$ matrix in the neural network model. Note that inputs do not need to go through sigmoid, and the shape of targets are the same with the shape of logits. The way to calculate logit is (6.16):

$$L(p) = \ln \frac{p}{1-p} \tag{6.16}$$

Logit is a function that maps the probability of the value range [0,1] to the real field [-inf, inf]. With x refers to logits and y indicating the targets, the loss function is computed as in (B.6).

This calculation is the standard implementation of Cross-Entropy algorithm, which performs sigmoid activation on the value obtained by $W \times X$, guarantees that the value is between 0 and 1, and then calculates the loss of cross-entropy.

Referring to the softmax process mentioned in output layer, the target labels are one-hot encoding, if we do not apply softmax, the claculated cross entropy must be a large value, for example, when the logits are [1,2,3,4], computing with one-hot label, the cross entropy would be large. Hence the softmax would improve the performance of loss layer.

Why is this loss function special to multi-label classification?

The reason is the cross_entropy loss calculated from network output logits and label is a measurement to error of independent non-repulsive discrete classification tasks, and these tasks mean that all labels are independent but not mutually exclusive among all tasks.

Take our project as an example, an audio could have multiple instruments labels, for a network with five nodes in output layer and a data with encoding true label [0,1,1,0,1]. The

presence of each label in this music is an independent event, but multiple labels can exist in a music, which means that the events are not mutually exclusive. So we can directly use the output of the network as the logits input of the this loss function to perform the cross-entropy loss of the output and the label. But for the multiclass classifications, for example, the logits value range is [0,10], and the target value is also [0,10]. In this loss function, after passing through sigmoid, the predicted value is limited to 0 to 1, and the $1 - y$ in the formula would have negative numbers as a result. Also, there is no linear relationship between 0 and 10. If directly bringing the label value into the calculation, there must be a very large error. Therefore, for multiclass problems, this loss function cannot be directly substituted.

Inside the function, since the shapes of labels and logits are [batch_size, num_classes], how to calculate their cross entropy since they are not valid probability distributions (the output of a batch after sigmoid would not add to 1) . The calculation of loss is element-wise. The shape of the loss returned by the method is the same as that of labels. It is also [batch_size, num_classes], and then calls the reduce_mean method to calculate the average loss in the batch. So the cross-entropy here is a class-wise cross entropy. Whether a label exists is an event. For each event, cross entropy loss is obtained, and final loss would be the average loss among the batch.

Mean RNN

Our second model Mean RNN is based on BiRNN structure. The difference is we add a mean layer after the BiRNN layer to post-process the hidden layer state value.

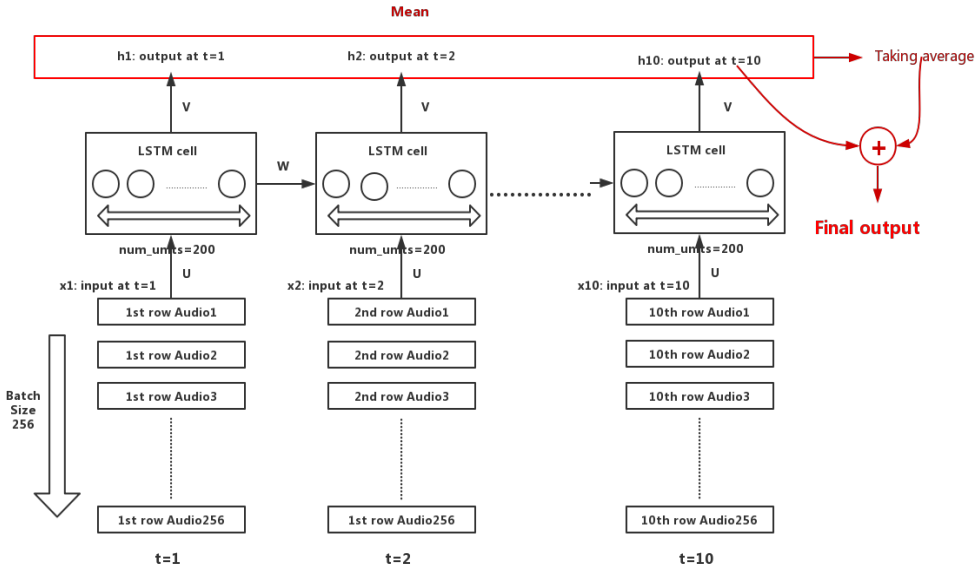


Figure 6.13: Mean layer

Figure 6.13 shows the algorithm in mean layer.

$$\text{Calculate average value}: \tilde{h}_{it} = h_{it} + \frac{\sum_{k=1}^{T_x} h_{ik}}{\overline{T}_x} \tag{6.17}$$

We add all corresponding dimension of all LSTM hidden layer state value together, and divide by total time steps to get the average value. As shown in (6.17).

$$\text{Updated output value } = h_{imax-t} + \tilde{h}_{it} \tag{6.18}$$

In (6.18), we add the average value to the hidden state value of last time step to get the new output. By adding the mean layer, information shared by all 10 time steps can be added to the output. For our project, features of all time steps have the same label. Hence, adding the mean layer can improve the final recognition performance.

To implement Mean RNN model, we use tensor mathematical calculation functions from Tensorflow. For example, "$tf.reduce\_sum$" is used to compute the sum value, "tf.cast(tf.expand dims(inputx, -1)" is used to obtain the number of time steps and divide to get the average value, "tf.add" is used to plus mean value to output. Detailed code will be attached in appendix.

Attention based BiRNN

Our last model structe is attention based BiRNN, the difference between last model and BiRNN model is that we add an attention layer between the BiRNN layer and dense layer. Figure 6.14 describes the structure of attention model.

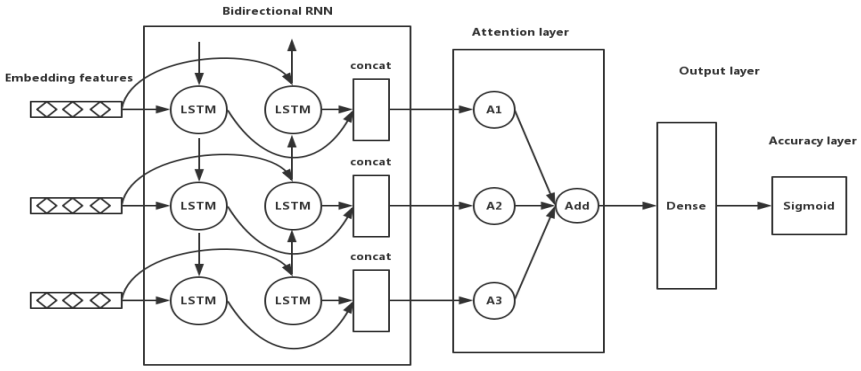


Figure 6.14: Attention Model Structure

Algorithm of this model is shown below:

$$u_{it} = \tanh\left(W_w h_{it} + b_w\right) \tag{6.19}$$

(6.19) applies a tanh function to the hidden layer state value to get a representation value for each time step.

$$\alpha_{it} = \frac{\exp\left(u_{it}^{\top} u_w\right)}{\sum_t \exp\left(u_{it}^{\top} u_w\right)} \tag{6.20}$$

In (6.20), weight of every hidden state value is computed by softmax.

$$s_i = \sum_t \alpha_{it} h_{it} \tag{6.21}$$

At last, the final output would be a weighted sum among all time steps, shown in (6.21). After applying attention mechanism, reffering to chapter 2.4.4, our model can focus on most representative features of particular time steps. Hence, the performance of recognition purpose would be increased.

To implement the attention model in Tensorflow, we use "tf.layers.dense" build a dense layer with tanh as activation function to apply the tanh transforming. "tf.exp", "tf.reduce_sum" and element-wise multiplication are used to calculate the weights. To get the weighted sum as final output, we use element-wise multiplication. Detailed code is shown in appendix.

6.5.2 Other Network Design Considerations

•Data spilt: 90% training, 10% training and validation

- Batch size : 256 (reason would be shown in following result part)
- Optimizer : Adam
- Learning rate : decaying learning rate starting from 0.0005
- Epoch : 500
- Stop training : validation loss starts to increase
- Regularization : Dropout
- Gradient clipping : global gradient with threshold 5
- Initialization: Xavier (to break the symmetry of network)

Decaying Learning Rate

The idea of having decaying learning rate is: in the early stage of model training, the model is optimized using a large learning rate to accelerate learning, making the model more accessible to local or global optimal solutions. As the number of iterations increases, the learning rate will gradually decrease, ensuring that the model will not fluctuate too much in the later stage of training, and thus the model could be trained closer to the optimal solution.

To implement this algorithm, we apply a linear relationship (learning rate / 5) when the validation accuracy would have no the new highest value for the following 8 iterations

Gradient Clipping algorithm

Gradient clipping is used to solve the gradient exploding[63]. It can effectively control the weight within a specific range. When we update the parameters using the gradient descent method, the value of the loss function decreases in the direction of the gradient. However, if the gradient (partial derivative) is large, the function value would jump, and the convergence cannot reach the optimal situation. By adding the sum of the squares of the partial derivatives of all parameters. Set the cropping threshold to "c" and set (6.22).

$$g_1 = \frac{\partial J(\mathbf{w})}{\partial w_1}, g_2 = \frac{\partial J(\mathbf{w})}{\partial w_2}, \|\mathbf{g}\|_2 = \sqrt{g_1^2 + g_2^2} \tag{6.22}$$

When $\|\mathbf{g}\|_2$ is greater than c , the gradient is calculated as (6.23)

$$\mathbf{g} = \frac{c}{\|\mathbf{g}\|_2} \cdot \mathbf{g} \tag{6.23}$$

To implement this algorithm, when $\|\mathbf{g}\|_2$ is greater than c , we keep the $\mathbf{g}$ value unchanged and use 'tf.clip_by_global_norm' function in tensorflow. This would compute the global norm and would clip after all the gradients have been calculated. The gradient value could be calculated by 'optimizer.compute_gradients()'. The rule of our chosen gradient clipping function is in (6.24).

$$t_i = \begin{cases} t_i * \frac{\text{clip norm}}{\text{avg.norm}}, & \text{avg norm} \geq \text{clip norm} \\ t_i, & \text{otherwise} \end{cases} \tag{6.24}$$

where avg_norm $= \sqrt{\sum_i \|t_i\|_2^2}$ and t_i is the tensor for each gradient in the list to be clipped. $\|t\|_2$ is the two-norm of the tensor t, that is, the sum of the squared values of all the elements. Clip_norm is clipping threshold; in our design we set it to be 5.

6.5.3 Results and Discussion

Batch size and loss function plot oscillation

In the first few times of the implementation, we suffer oscillating loss plot where the horizontal axis is the iteration time, and the vertical axis is the value of loss function. The plot is shown in figure 6.16

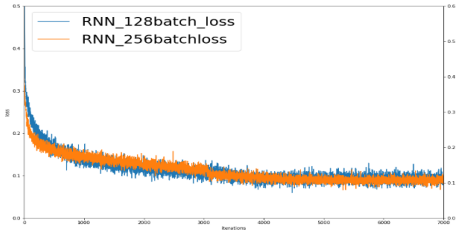


Figure 6.15: Oscillating loss function plot

However, the oscillation is not what we expect. To reduce or eliminate the oscillation, we would increase the batch size. The reasons are when the batch size is too small, and there are more categories of classification, the loss function may oscillate without converging, especially when the network structure is more complicated. While the batch size grows, the data in the same quantity would be processed faster also the time of epochs needed to train for an equal accuracy would raise. Combining the two contradictory phenomena mentioned above, when Batch_Size increases to a certain value, a relatively optimal condition would be obtained. Whereas, the result of having an overlarge batch size is that the network is easy to converge to some bad local optimal point. An undersize batch also has some problems, such as training speed is very slow and not easy to converge.

Compare the loss plot for 128 and 256 batch size, after increasing the batch size, the oscillation declining is relatively small. However, due to the limitation of Computer Memory, we could not continue to grow the batch size[64]. Hence, we choose 256 as batch_size.

Evaluation

We set three accuracies as the evaluations of our model:

(1). All-correct accuracy: this would only count the prediction, which exactly matches all the true label of the testing data as the positive(P).

(2). One accuracy: this would take one label with the highest probability as the prediction result which can be considered as single instrument recognition.

(3). Inter-accuracy: this would take the prediction as positive when all the predicted labels have an intersection with the true labels. Hence, considering the computational process, the inter-accuracy would have the highest numerical value and the all-correct accuracy would have the lowest numerical value since it is the strictest standard.

Experiment: Standard Training on three RNN Models

Table 6.5 shows the performance of three RNN models.

Model	*All-correct acc*	*One-acc*	*Inter-acc*	*F1-score*
Bidi-RNN	0.7314	0.8338	0.8705	0.822287
Mean-RNN	0.7565	0.8589	0.8966	0.848760
Attention-RNN	0.7488	0.8560	0.8937	0.844831

Table 6.5: Performance of three differnent RNN models

For the accuracies, due to the rule is becoming loose from all-correct accuracy to intersectional acccuray. Hence, the numerical value would increase. And with the Mean RNN and attention RNN model, the performance of network would be increase.

Discussion for standard training

From the results from standard training on three RNN models. We could see the importance of choosing a proper batch_size value as well as the improvement of performance by updating the RNN architecture. The model with the weakest performance is the bidirectional RNN model, it can read features from both directions, which can double the data size. Attention RNN has a better performance than BiRNN, because by including attention mechanism, the output of our model would have the information concentrating on a certain state. In our project, not

all the input sequences would carry the information of the label they have, because the input represents a piece of 10-second long music tagged with all the instruments shown in this time interval. However, for multiple instruments, not all of them would play the whole time for the 10s time interval. They might start at some middle point. However, the time sequence without all instruments playing is tagged as having all labels for our dataset. With the attention, our model can focus on the most crucial sequence which has all labelled instruments playing in that time slot. Our self-designed Mean RNN has best results among three models. Because, by introducing the mean value of all dimensions of RNN output state value, some information shared with the whole sequence can be added to the output. Hence, the output would carry overall information for a certain sequence data. For our project, the ten sequence inputs have the same label. Thus, adding global information would be helpful to train our model.

Considering the interpretation of each RNN models, we expect the Attention RNN would have the best performance. However, the practical results shows that our self-designed Mean RNN model has slightly better performance than the attention model. However, the differences between these two models are really close. The reason for this could be that both models are trained to fully achieve the most optimal performance of the dataset(referring to chapter 2.3), then the slight performance differences are caused by random processes happening when models deal with the testing data.

By the results from three different accuracies , it can be shown that our model has the higher recognizing ability when the number of playing instruments in music is less:

(1) For 'One accuracy' with highest numerical value, our model behaves like a single instrument classifier. Our model in Problem B has worse performance when predicting single instrument comparing with model built in Problem A(chapter 5.5.1). This is due to the different training dataset. The Medley-Solos-DB dataset(chapter 4.3.1) used in Problem A contains only audio sound with single instrument playing, and the OpenMIC-2018 dataset(chapter 4.3.2) for Problem B has multiple instruments playing in every training example audio. Hence, our model produced in Problem B has inherent limitation when recognizing single instrument.

(2) For the 'all-correct' accuracy with least numerical value, after analyzing all the prediction result of experiments, this accuracy is mainly given credit to the testing examples with one label or two labels. For the other testing examples with three labels or even more labels, it is infrequent for our model to predict all the true labels correctly.

(3) The 'intersection' accuracy is used to test the ability of our model to recognize part of true labels. For intractable testing audio data with 3 or more instrument playing, our model has a high capacity to recognize part of the instruments in the music.

Experiment : Changing Threshold Value

In this experiment, we change the threshold value from 30% to 90% to observe the performance of each model. Among all the evaluation methods, the most concerned one is the accuracy of our project. And the all-correct accuracy is the strictest one; hence we would compare the variation of the all-correct accuracy while changing the threshold values.

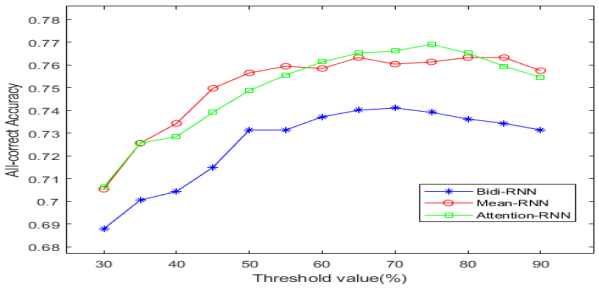


Figure 6.16: All-correct accuracy with threshold value

From the result, we can tell that by changing the threshold value, the performance would be improved and each model has its own optimal threshold value.

Discussion for changing threshold value

With the result of this experiment, the BiRNN model would have the best performance when the threshold value is set to 60%, for mean RNN the optimal threshold value is 65%, and for attention RNN the best threshold value is 70%. When we have a low threshold value, for example, 30%, the model might make many wrong predictions. And then setting 90% as the threshold, the model would make few predictions which may miss some true labels. Each model has its own optimal value, this is due to the different architecture of network.

Experiment: Changing the Split Ratio of Dataset

The dataset Openmic-2018 does not offer additional testing data, hence to make full use of the provided data, we would divide the data into two parts with ratio 9:1 at the beginning of our implementation. In other words, 90% of the data would be training data.

To have a control group; we use another split ratio of the data which is 8:1:1, that is 80% data is training data, 10% data is validation and remaining 10% would be testing data. This is a quite standard way to divide the data which would provide independent validation set and testing set. 80% is training data and 10% of data is validation and testing data.

Model	*All-acc 9:1*	*F1 9:1*	*All-acc 8:1:1*	*F1 8:1:1*
Bidi-RNN	0.7314	0.8222	0.7125	0.8045
Mean-RNN	0.7565	0.8488	0.7345	0.8236
Attention-RNN	0.7488	0.8448	0.7297	0.8195

Table 6.6: Performance of RNN models in different split ratio

From the result, we can tell that if our model would have more training data, the performance of our models would be increased. This is also a limitation of OpenMic-2018 dataset.

Discussion for changing split ratio

In this experiment, we use different split ratio. OpenMIC-2018 only have 20,000 examples, which is not an ideal size to fully train the neural network. One of the characteristics of the neutral network is the more the training data is, the better the model would be trained. Nevertheless, referring to chapter 4.3.2, finding an open dataset with the labels of multiple instruments for each audio is not easy. For now, the only friendly dataset we could find is openmic-2018.

6.6 Network Ensemble

After training the CNN-based model and the RNN-based model separately, we will use ensemble learning to combine the models to improve the performance referring to chapter 2.7.3. We will use jupyter notebook to implement ensemble learning. We assign the predicted probabilities to be scores of a model, the score can be used with threshold value to determine the predicted labels. To combine the models, scores of Mean-RNN model and CNN model would be combined together, then we calculate the average score as the final prediction results. We test the model with different threshold values to get best performance considering all-correct accuracy. The optimal threshold value would be used to evaluate the model. The performance comparison is shown in table 6.7.

Model	**all-acc**	**F1-score**
CNN	0.7507	0.840500
Mean-RNN	0.7565	0.848760
Ensemble-model	0.7739	0.853237

Table 6.7: Performance of Ensemble model

Compare final results of multiple instrument recognition with previous works:

Model	Precision	Recall	F1-score
Bosch et al.[16]	0.504	0.501	0.503
Han et al.[5]	0.655	0.557	0.602
Our model	0.880	0.846	0.853

Table 6.8: Recognition Performance Comparison with Previous Works

Table 6.8 compares our model against previous where our model performs better using all three evaluation metrics.

6.7 Reflection

In our design, the pre-trained VGGnet is used as a feature extractor, the generalization of VGG model has been confirmed by many ML tasks from different areas. Hence, the embedding feature can be considered as typical information with high representativeness extracted from the raw audio data. Also, the VGG model emphasizes that CNN must be deep enough that the hierarchical representation of visual and audio data would be useful. But for a Deep structure, the simplicity must be satisfied. For music instruments recognition purpose, VGGish model has not been widely used. By our transfer learning based model, the superiority of VGGish model acting as feature extractor for audio signal has been proved. Hence, for other music related ML tasks, VGGish model can be used to improve the performance.

A NLP based CNN model was then implemented where we showed that techniques developed for sentence classification were successfully transferred for our use case when combined with the feature embeddings from the VGGish model. Here, we also saw similarities in the results with experiments conducted in Problem A such as the effect kernel size and input padding has on the overall performance of the model. Although we have discussed this relationship in chapter 6.4.2, we were unable to find any any conclusive literature detailing this relationship. We have also introduced a novel method of selecting sub-network configurations for our custom architecture.

The bad performance for predicting data (chapter 6.5.3) with number of labels greater than three could occur due to the dataset. The label of OpenMIC-2018 is tagged automatically by machine, also there is correlation index of each label for every data. In our project, we only use the label with relevance value greater than 0.9. Though the label with other relevance value could also exist in audio file. Hence, the performance would be affected by the inaccurate tagging of dataset. Also, for multiple instruments playing together, we do not consider noise exists in recording audio. For example, there would be crosstalks between the microphone and

the instruments when capturing sounds, and the microphone would capture the sound from the environment [68].

By using ensemble learning method, the overall performance of our classifier is improved, which can give us inspiration that for complicated classification tasks (some may not provide abundant training data or some may not implement complex classifier architecture due to limitation of resources), ensemble learning can combine several "weak" classifier to a "strong" classifier which has enhanced performance. Actually, ensemble learning is advantageous for students who only have PC as the training tool. The training time of ML models depends on the hardwares like CPU and GPU. If people do not want their PC run for a total week or even weeks to train a model, they can build simple models and ensemble them together to save the engineering efforts.

However, even using the model created for Problem A can get the multiple prediction results. In Problem A, we use softmax as the output activation function which would make probabilities of all labels sum together to 1. For recognizing only one instrument, we can pick up the label with the highest probability. If we set a threshold value to each softmax result, we can get the multiple prediction results. However, using softmax to multiple label classification is not common and has worse performance comparing with sigmoid results.

Chapter 7: Conclusion and Future Works

Our project approached the task of musical instrument recognition as a two phase classification problem, each with specific aims and parameters as defined in 1.3. In the first phase, we evaluated state of the art neural network tools and techniques and used them to build a single instrument recognizer. We also used the most current research concepts in the field of musical signal processing to develop a CNN model with context specific filter dimensions. Through our research, we found that spectral filters as described in 5.4.1 with widths spanning approximately a quarter of the spectral range produced the best performance.

The importance of padding input features into CNN filter was also experimented with and the positive results motivated its implementation in the second phase model. By experimenting with different types of activation functions, we also inferred that our output labels appeared to be linearly separable. However it is unclear if this is due to the data-set, choice of musical instrument or an inherent musical signal property. This phase concluded with a 98.5% classification accuracy, which was a slight improvement from past works in the field. However, it should also be noted that we experimented using a considerable smaller set of labels which may explain our superior results.

Phase two was then used to approach the problem of classifying multiple instruments playing simultaneously in an audio recording. Here we looked to incorporate machine learning concepts developed for applications outside the scope of musical signal processing into our final design.

First we took advantage of the newly released VGGish model to to perform feature extraction on our input data. Historically used to produce embedding features for visual data, the VGGish model allowed us implement transfer learning in our final design. Comparing our results with past works in the field, We found that using the VGGish feature embeddings instead of the traditional spectrogram resulted in superior performance.

The implementation of VGGish opened new channels for us to incorporate contextually new methods and architectures in our design. To optimize our model's performance, we utilized the proven ensemble technique capable of maximizing the overall performance of several models.

Our CNN architecture drew from the works for NLP and we developed a model specifically for multiple instrument recognition using feature embeddings. Here we also discovered result similarities to the first phase such as the effect of padding. Our self-designed Mean RNN model was inspired by architectures for medical purposes has been proven to have a more efficient recognition capacity than the normal BiRNN and attention-based BiRNN models. Our overall model performed better than the past works in the field and can be directly transferable and retrained to perform recognition on other application such as sentence classification.

Finally we identified an area of improvement in the field of multi-label classification through our analysis and evaluation of our proposed Rank-CE loss function. We found that using this function, the final inference phase did not require any threshold biasing which we theorize could be use to label dependencies being propagated trough the output layer. Other areas of improvements we have identified include the current lack of multi-label analysis tools on the python API and we have built and implemented classes and functions capable of doing so. Furthermore, we have identified several issues with blind usage of the OpenMIC-2018 data-set since the relevance factor determines the quality of the examples. However, only using the examples with a high relevant factor results in a very uneven data-set with respect to the number of examples within each class.

Future works

- Implementing the experiment as described in 5.5.2
- Further investigation into the linearly separable labels found in the first phase.
- Testing the model developed in the first phase on a larger and more comprehensive data-set.
- Taking the cross-talk of audio data into account and design a method to reduce or eliminate the cross-talk. Cross-talk is happening everywhere when recording actual music. The cross-talk between microphones and cross-talk with environment can both affect the quality of recording, hence, the performance of classifier would also be affected.
- Further experiments on our proposed Rank-CE loss function and observe its performance on data-sets with highly correlated output labels. An example of this would be for sentence classification applications.
- Considering few audio datasets with labels tagged with all instruments, in the future

work, it is significant to create a new dataset which could meet our requirements. The way to construct the dataset could be hand-tagging or processing existing dataset with a single label using mixing technology.

- Limited by the resources, in this project, we cannot build model with very deep structure. The acknowledged practical experience is that the deeper neural network is, the better the final performance would be. To further develop our project, we can try deeper network structure.

- Our project can be the base of many MIR tasks, for example music source separation, musical content similarity and music recommendation system.

Bibliography

[1] Toghiani, B., & Windmark, M.(2016). 'Musical Instrument Recognition Using their Distinctive Characteristics in Artificial Neural Networks'

[2] Mazarakis, G., & Tzevelekos, P., & Kouroupetroglou, G. 'Musical Instrument Recognition and Classification Using Time Encoded Signal Processing and Fast Artificial Neural Networks'. SETN 2006: Advances in Artificial Intelligence pp 246-255

[3] R.A.King, T. C. Phipps: Shannon, TESPAR and Approximation Strategies. ICSPAT 98, Vol. 2, pp 1204-1212

[4] Guignard, L., & Kehoe, G.(Autumn 2015). 'Learning Instrument Identification'. Stanford University

[5] Han, Y., & Kim, J., & Lee, K.(Jan 2017). 'Deep convolutional neural networks for predominant instrument recognition in polyphonic music'. In IEEE/ACM Transactions on Audio, Speech, and Language Processing, vol.25, no.1, pp.208-221

[6] Loughran, R., & Walker, J., & O'Neill, M., & O'Farrell, M. 'The Use of Mel-frequency Cepstral Coefficients in Musical Instrument Identification'. In ICMC 2008

[7] Banchhor, S, K., & Khan, A. (2012). 'Musical Instrument Recognition using Spectrogram and Autocorrelation'. International Journal of Soft Computing and Engineering (IJSCE) ISSN: 2231-2307, vol 2, issue 1

[8] Copeland, C., & Mehrotra, S. (2013). 'Musical Instrument Modeling and Classification'. Stanford University

[9] Singh, P., & Bachhav, D., & Joshi, O., & Patil, N. (Mar 2019). 'Musical Instrument Recognition using CNN and SVM'. International Research Journal of Engineering and Technology (IRJET), vol 06, issue 3

[10] Yun, M., & Bi, J. (2017). 'Deep Learning for Musical Instrument Recognition'. Department of Electrical and Computer Engineering University of Rocheester

[11] Spyromitros, E., & Tsoumakas, G., & Vlahavas, I. 'Multi-Label Learning Approaches for Music Instrument Recognition'. ISMIS 2011: Foundations of Intelligent Systems pp 734-743

[12] Donnelly, P. (Aug 2015). Doctor Thesis, 'Learning Spectral Filters for Single and Multi-label Classfication of Musical Instruments', pp 20-23, pp 140-174

[13] Yip, H., & Bittner, R. (2017). 'An Accurate Open-source Solo Musical Instrument Classifier', The Spence School New York & Music and Audio Research Lab New York University

[14] Yu, L., & Su, L., & Yang, Y. (2014). 'Sparse Cepstral Codes and Power Scale for Instrument Identification', Research Center for Information Technology Innovation, Academia Sinica, Taiwan

[15] Pons, J., & Slizovskaia, O., & Gomez, E., & Serra, X. (Jun 2017). 'Timbre Analysis of Music Audio Signals with Convolutional Neural Networks', Music Technology Group, University Pompeu Fabra, Barcelona

[16] Bosch, J., & Janer, J., & Fuhrmann, F., & Herrera, P. 'A Comparison of Sound Segregation Techniques for Predominant Instrument Recognition in Musical Audio Signals', ISMIR 2012.

[17] Rosenblatt, F. 1957. The Perceptron—a perceiving and recognizing automaton. Report 85–460–1, Cornell Aeronautical Laboratory.

[18] McCulloch, W. S. and Pitts, W. 1943. A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics 5:115–133.

[19] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. Neural Networks : The Official Journal of the International Neural Network Society, 12(1), 145–151. http://doi.org/10.1016/S0893-6080(98)00116-6

[20] Fukushima, K. (1980). 'A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position', Volume 36, Issue 4, pp 193–202

[21] Krizhevsky, A.,& Sutskever, I., & Hinton, G. (2012). 'ImageNet Classification with Deep Convolutional Neural Networks', University of Toronto

[22] Krizhevsky, Alex; Sutskever, Ilya; Hinton, Geoffrey E. (2017-05-24). "ImageNet classification with deep convolutional neural networks" (PDF). Communications of the ACM. 60 (6): 84–90. doi:10.1145/3065386. ISSN 0001-0782.

[23] Springenberg, J., & Dosovitskiy, A., & Brox, T., & Riedmiller, M. (2015). 'STRIVING FOR SIMPLICITY: THE ALL CONVOLUTIONAL NET', Department of Computer Science, University of Freiburg

[24] Scherer, D., & Muller, A., & Behnke, S. (2010). 'Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition', 20th International Conference on Artificial Neural Networks (ICANN), Thessaloniki, Greece

[25] Kong, C., & Lucey, S. (2017). 'Take it in your stride: Do we need striding in CNNs?', Carnegie Mellon University

[26] Nair, V., & Hinton, G. (2010). 'Rectified Linear Units Improve Restricted Boltzmann Machines', 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel

[27] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for Activation Functions," ArXiv, 2017. [Online]. Available: 1710.05941;http://arxiv.org/abs/1710.05941

[28] Glorot, X., & Bordes, A., & Bengio, Y. (2011). 'Deep Sparse Rectifier Neural Networks', 14th International Conference on Artificial Intelligence and Statistics (AISTATS)

[29] Web Page: http://cs231n.github.io/neural-networks-1/actfun

[30] Song, Y., & Schwing, A., & Zemel, R., & Urtasun, R. (2016). 'Training Deep Neural Networks via Direct Loss Minimization', ICML2016

[31] Hsieh, C., & Lin, Y., & Lin, H. (2018). 'A Deep Model with Local Surrogate Loss for General Cost-sensitive Multi-label Learning', Department of Computer Science and Information Engineering, National Taiwan University

[32] Machine Learning: Theory and Applications. From Dr. Dirk Kroese, The Cross-Entropy Method for Estimation. In: Venu Govindaraju, C. R. Rao, editors, Handbook of Statistics, Vol 31. Chennai: Elsevier B.V., 2013, p. 19-34.

[33] Grodzicki, R., & Mandziuk, J., & Wang, L. (2008). 'Improved Multilabel Classification with Neural Networks', DOI: 10.1007/978-3-540-87700-4_41

[34] Zhang, M., & Zhou, Z. 'Multi-Label Neural Networks with Applications to Functional Genomics and Text Categorization', IEEE Transactions on Knowledge and Data Engineering 18 (2006) 1338-1351

[35] Nam, J., & Kim, J., & Mencia, E., & Gurevych, I., & Furnkranz, J. (2014). 'Large-scale Multi-label Text Classification — Revisiting Neural Networks', ECML PKDD 2014: Machine Learning and Knowledge Discovery in Databases pp 437-452

[36] Pons, J., & Lidy, T., & Serra, X. (2016). "Experimenting with Musically Motivated Convolutional Neural Networks". In 14th International Workshop on Content-Based Multimedia Indexing (CBMI). Publisher: IEEE.

[37] Pons, J., & Serra, X. (2017). 'DESIGNING EFFICIENT ARCHITECTURES FOR MODELING TEMPORAL FEATURES WITH CONVOLUTIONAL NEURAL NETWORKS', 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)

[38] Minsky, & Papert, & Seymour. (1969). 'Perceptrons : An Introduction to Computational Geometry', Cambridge, Mass: MIT Press, 1972. 2nd. ed.

[39] Springenberg, J., & Dosovitskiy, A., & Brox, T., & Riedmiller, M. (2015). 'STRIVING FOR SIMPLICITY: THE ALL CONVOLUTIONAL NET', arXiv:1412.6806

[40] Stephen McAdams. Musical timbre perception. The Psychology of Music, pages35–68, 2012.

[41] Thorsten Wuest, Daniel Weimer, Christopher Irgens & Klaus-Dieter Thoben (2016) Machine learning in manufacturing: advantages, challenges, and applications, Production & Manufacturing Research, 4:1, 23-45, DOI: 10.1080/21693277.2016.1192517

[42] John M Grey. Multidimensional perceptual scaling of musical timbres. The Journal of the Acoustical Society of America, 61(5):1270–1277, 1977

[43] Douglas O'shaughnessy. Speech communication: human and machine. Universities press,1987.

[44] Jan Schluter. Deep Learning for Event Detection, Sequence Labelling and Similarity Estimation in Music Signals. PhD Thesis, Johannes Kepler University Linz, Austria,July 2017.

[45] Shetty, B. (2019). Supervised Machine Learning: Classification: An exhaustive understanding of classification algorithms in machine learning. Retrieved from https://towardsdatascience.com/supervised-machine-learning-classification-5e685fe18a6d

[46] Turner, C. R., Fuggetta, A., Lavazza, L., & Wolf, A. L. (1999). A conceptual basis for feature engineering. Journal of Systems and Software, 49(1), 3-15.

[47] Zou, H., & Xue, L. (2018). A Selective Overview of Sparse Principal Component Analysis. Proceedings Of The IEEE, 106(8), 1311-1320. doi: 10.1109/jproc.2018.2846588

[48] Schmidhuber, Jürgen (January 2015). "Deep Learning in Neural Networks: An Overview". Neural Networks. 61: 85–117. arXiv:1404.7828. doi:10.1016/j.neunet.2014.09.003. PMID 25462637

[49] Elman, Jeffrey L. (1990). "Finding Structure in Time". Cognitive Science. 14 (2): 179–211. doi:10.1016/0364-0213(90)90002-E.

[50] Hochreiter, Sepp; & Schmidhuber, Jürgen (1997-11-01). "Long Short-Term Memory". Neural Computation. 9 (8): 1735–1780. doi:10.1162/neco.1997.9.8.1735.

[51] Li, Xiangang; & Wu, Xihong (2014-10-15). "Constructing Long Short-Term Memory based Deep Recurrent Neural Networks for Large Vocabulary Speech Recognition". arXiv:1410.4281

[52] Graves, Alex; & Schmidhuber, Jürgen (2005-07-01). "Framewise phoneme classification with bidirectional LSTM and other neural network architectures". Neural Networks. IJCNN 2005. 18 (5): 602–610. CiteSeerX 10.1.1.331.5800. doi:10.1016/j.neunet.2005.06.042. PMID 16112549.

[53] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. NIPS.

[54] S. J. Pan & Q. Yang, "A Survey on Transfer Learning," in IEEE Transactions on Knowledge and Data Engineering, vol. 22, no. 10, pp. 1345-1359, Oct. 2010. doi: 10.1109/TKDE.2009.191

[55] S. Dieleman & B. Schrauwen, "End-to-end learning for music audio," 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Florence, 2014, pp. 6964-6968. doi: 10.1109/ICASSP.2014.6854950

[56] Humphrey, Eric J., Durand, Simon, & McFee, Brian. "OpenMIC-2018: An Open Dataset for Multiple Instrument Recognition." in Proceedings of the 19th International Society for Music Information Retrieval Conference (ISMIR), 2018.

[57] R. Bittner, J. Salamon, M. Tierney, M. Mauch, C. Cannam & J. P. Bello, "MedleyDB: A Multitrack Dataset for Annotation-Intensive MIR Research", in 15th International Society for Music Information Retrieval Conference, Taipei, Taiwan, Oct. 2014.

[58] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

[59] Zhou, P., Shi, W., Tian, J., Qi, Z., Li, B., Hao, H., & Xu, B. (2016). Attention-based bidirectional long short-term memory networks for relation classification. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers) (Vol. 2, pp. 207-212).

[60] tf.nn.sigmoid_cross_entropy_with_logits — TensorFlow Core 1.13 — TensorFlow. (2019). Retrieved from https://www.tensorflow.org/api_docs/python/tf/nn/sigmoid_cross_entropy_with_logits

[61] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[62] Williams, R. J., & Zipser, D. (1995). Gradient-based learning algorithms for recurrent. Backpropagation: Theory, architectures, and applications, 433.

[63] Pascanu, R., Mikolov, T., & Bengio, Y. (2012). Understanding the exploding gradient problem. CoRR, abs/1211.5063, 2.

[64] SHARMA, S. (2017). Epoch vs Batch Size vs Iterations: Know your code. Retrieved from https://towardsdatascience.com/epochvsiterations-vs-batchsize4dfb9c7ce9c9

[65] Hershey, S., Chaudhuri, S., Ellis, D. P., Gemmeke, J. F., Jansen, A., Moore, R. C., ... & Slaney, M. (2017, March). CNN architectures for large-scale audio classification. In 2017 ieee international conference on acoustics, speech and signal processing (icassp) (pp. 131-135). IEEE.

[66] Gemmeke, J. F., Ellis, D. P., Freedman, D., Jansen, A., Lawrence, W., Moore, R. C., ... & Ritter, M. (2017, March). Audio set: An ontology and human-labeled dataset for audio events. In 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 776-780). IEEE.

[67] Schuster, M., Paliwal, K. K. (1997). Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 45(11), 2673-2681.

[68] Lee, H., & Rumsey, F. (2005). Investigation into the effect of interchannel crosstalk in multichannel microphone technique.

[69] Sander Dieleman. Learning feature hierarchies for musical audio signals. PhD Thesis, Ghent University, 2015.

[70] Eric J Humphrey, Juan P Bello, and Yann LeCun. Feature learning and deep architectures: New directions for music informatics. Journal of Intelligent Information Systems, 41(3):461-481, 2013.

[71] Sander Dieleman and Benjamin Schrauwen. End-to-end learning for music audio. In Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on, pages 6964-6968. IEEE, 2014.

[72] Rokach, L. (2010). Ensemble-based classifiers. Artificial Intelligence Review, 33(1-2), 1-39.

[73] Britz, D. (2015). Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs. Retrieved from http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

[74] Olah, C. (2015). Understanding LSTM Networks – colah's blog. Retrieved from https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[75] Mohsen Kaboli. A Review of Transfer Learning Algorithms. [Research Report] Technische Universität München. 2017.

[76] Bahnsen, A. (2017). Building AI Applications Using Deep Learning. Retrieved from https://blog.easysol.net/building-ai-applications/

[77] Breiman, Leo. "Stacked regressions." Machine learning 24.1 (1996): 49-64.

[78] Jesse Read, Bernhard Pfahringer, Geo Holmes, Eibe Frank. Classier chains for multi-label classication. Machine Learning, 85(3):333-359, 2011.

[79] Alom, Md. Zahangir & M. Taha, Tarek Yakopcic, Christopher Westberg, Stefan Hasan, Mahmudul C Van Esesn, Brian Awwal, Abdul Asari, Vijayan. (2018). The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches.

[80] Havel, V. (2019). vanHavel/bpmlltensorflow. Retrieved from https://github.com/vanHavel/bpmlltensorflow

Appendix A: Equations

A.0.1 Activation Functions

Sigmoid

$$g(z) = \frac{1}{1 + e^{-z}} \tag{A.1}$$

Softmax

$$g(z)_i = \frac{e^{z_i}}{\sum_j^k e^{z_j}} \tag{A.2}$$

ReLU

$$g(z) = max(0, z) = \begin{cases} z_i & \text{if } z_i > 0 \\ 0 & \text{if } z_i < 0 \end{cases} \tag{A.3}$$

A.0.2 Loss Functions

Categorical Cross-entropy

$$\begin{aligned} L(\tilde{y}, y) &= -\log \tilde{y}_p \\ &= -\log \frac{e^{z_p}}{\sum_j^k e^{z_j}} \end{aligned} \tag{A.4}$$

BP-MLL

$$L = \frac{1}{|Y||\bar{Y}|} \sum_{(p,n)\in(Y\times\bar{Y})} \exp -(y_p - \bar{y}_n) \tag{A.5}$$

Binary CE

$$L(\tilde{y}, y^t) = -\sum_i^C y_i^t \log \tilde{y}_i + (1 - y_i^t) \log 1 - \tilde{y}_i \tag{A.6}$$

A.0.3 Batches

Batch Gradient Descent

$$L(\tilde{y}, y^{true}) = \frac{1}{m} \sum_{i=1}^{m} L(\tilde{y}^{(i)}, y^{true(i)}) \tag{A.7}$$

SGD

$$L(\tilde{y}, y^{true}) = L(\tilde{y}^{(i)}, y^{true(i)}) \tag{A.8}$$

Appendix B: Mathematical Derivations

B.1 Chapter 6

B.1.0.1 Back-Propagation Learning Algorithm

To derive the necessary equations to implement the back-propagation learning algorithm 2, we will consider a MLP with overall network loss $L(\tilde{y}, y^{true})$ which has been averaged across all training examples in a batch. The predicted values at the output layer (L+1) are given by equation B.1 where the output activations $\boldsymbol{z}^{[\boldsymbol{L+1}]}$ are a linear combination of the previous layer's outputs multiplied by $W^{[L+1]}$.

$$\begin{aligned} \tilde{\boldsymbol{y}} &= \boldsymbol{a}^{[\boldsymbol{L+1}]} \\ &= f\left(\boldsymbol{z}^{[\boldsymbol{L+1}]}\right) \\ &= f\left((\boldsymbol{W}^{[\boldsymbol{L+1}]})^T \boldsymbol{a}^{[\boldsymbol{L}]} + \boldsymbol{b}^{[\boldsymbol{L+1}]}\right) \end{aligned} \tag{B.1}$$

To minimize the loss function, the gradient descent update rule given by equation 2.17 is performed on $W^{[L+1]}$ and $b^{[L+1]}$. This requires the partial derivatives $\frac{\partial L}{\partial W^{[L+1]}}$ and $\frac{\partial L}{\partial b^{[L+1]}}$ to be computed for each parameter update.

To do this we will first calculate the upstream gradient into the current layer's activations $z^{[L+1]}$ is obtained using the chain rule as shown by equation B.2, where $\frac{\partial L}{\partial a^{[L+1]}}$ is the partial derivative of the network loss with respect to the output predictions and $\frac{\partial a^{[L+1]}}{\partial z^{[L+1]}}$ is the partial derivative of the activation function $f(.)$ with respect to the output layer's activations.

$$\begin{aligned} \delta^{[L+1]} &= \frac{\partial L}{\partial z^{[L+1]}} \\ &= \frac{\partial L}{\partial a^{[L+1]}} \frac{\partial a^{[L+1]}}{\partial z^{[L+1]}} \end{aligned} \tag{B.2}$$

We can again use the chain rule to calculate the required partial derivatives by multiplying the local gradients by the upstream gradient shown in equations B.3.

$$\begin{aligned}\frac{\partial L}{\partial W^{[L+1]}} &= \frac{\partial L}{\partial z^{[L+1]}}\frac{\partial z^{[L+1]}}{\partial W^{[L+1]}} \\ &= \delta^{[L+1]}a^{[L]} \\ \frac{\partial L}{\partial b^{[L+1]}} &= \frac{\partial L}{\partial z^{[L+1]}}\frac{\partial z^{[L+1]}}{\partial b^{[L+1]}} \\ &= \delta^{[L+1]}\end{aligned} \tag{B.3}$$

The layer parameters can therefore be updated using equations B.4.

$$\begin{aligned}W^{[L+1]} &= W^{[L+1]} - \alpha\frac{\partial L}{\partial W^{[L+1]}} \\ &= W^{[L+1]} - \alpha\delta^{[L+1]}a^{[L]} \\ b^{[L+1]} &= b^{[L+1]} - \alpha\frac{\partial L}{\partial b^{[L+1]}} \\ &= W^{[L+1]} - \alpha\delta^{[L+1]}\end{aligned} \tag{B.4}$$

To make parameter updates in each previous consecutive layer, we will use a similar method to obtain the gradient being propagated back given by equation B.5.

$$\begin{aligned}\frac{\partial L}{\partial a^{[L]}} &= \frac{\partial L}{\partial a^{[L+1]}}\frac{\partial z^{[L+1]}}{\partial a^{[L]}} \\ &= \delta^{[L+1]}W^{[L+1]}\end{aligned} \tag{B.5}$$

B.2 Chapter 7

B.2.0.1 Bidirectional RNN Loss Layer

$$\begin{aligned}&y * (-\log(\text{sigmoid}(x))) + (1-y) * (-\log(1-\text{sigmoid}(x))) \\ =&y * \left(-\log\left(1/1+e^{-x}\right)\right) + (1-y) * \left(-\log\left(e^{-x}\right)/\left(1+e^{-x}\right)\right) \\ =&y * \log\left(1+e^{-x}\right) + (1-y) * \left(-\log\left(e^{-x}\right) + \log\left(1+e^{-x}\right)\right) \\ =&y * \log\left(1+e^{-x}\right) + (1-y) * \left(x + \log\left(1+e^{-x}\right)\right) \\ =&x - x * y + \log\left(1+e^{-x}\right) \quad use \ * \ for \ element \ wise \ multiplication\end{aligned} \tag{B.6}$$

Appendix C: Algorithms

Algorithm 1 Perceptron Algorithm

0: **procedure** PERCEPTRON(α, b, y^{true}, x)
0: $w_0 \leftarrow \mathbf{0}$
0: **for** $t = 0, 1, ..$ **do**
0: $\tilde{y_t} \leftarrow sgn(w_t x + b)$
0: $w_{t+1} \leftarrow w_t + \alpha(y^{true} - \tilde{y_t})x$
0: **end for**
0: **return** w
0: **end procedure**=0

Algorithm 2 Backpropagation learning algorithm

for d in data **do**
FORWARDS PASS
Starting from the input layer, propagate trough the network, computing the activations of the neurons at each layer
Compute and store the derivatives of the error function with respect to the output layer activities.
BACKWARDS PASS
for layer in layers **do**
Compute loss partial derivatives with respect to inputs of the upper layer neurons
Compute loss partial derivatives with respect to the weights between the outer layer and the layer below
Compute loss partial derivatives with respect to the activities of the layer below
end for
Perform weight updates
end for=0

Appendix D: Images

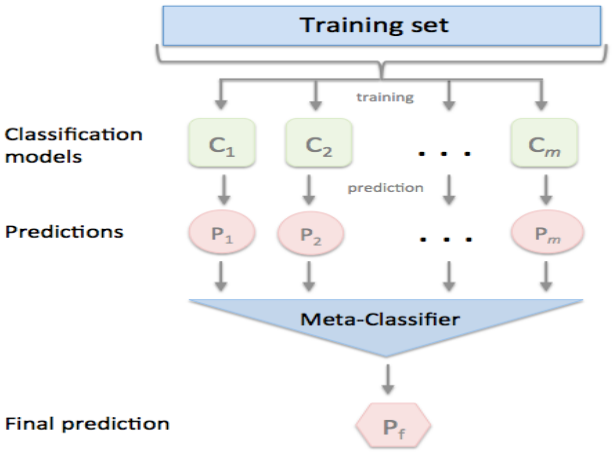


Figure D.1: Structure of ensemble learning [77]

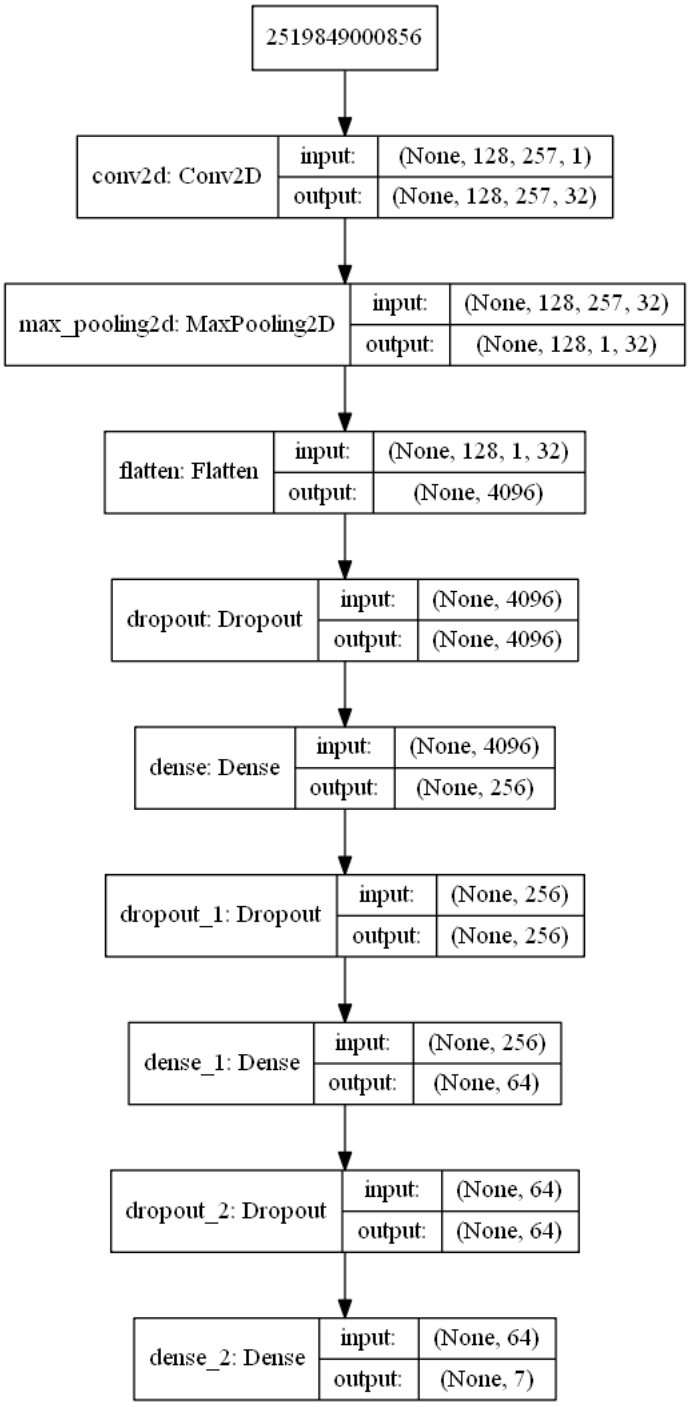


Figure D.2: Single-Label Final Network Architecture (Figure created using Keras)

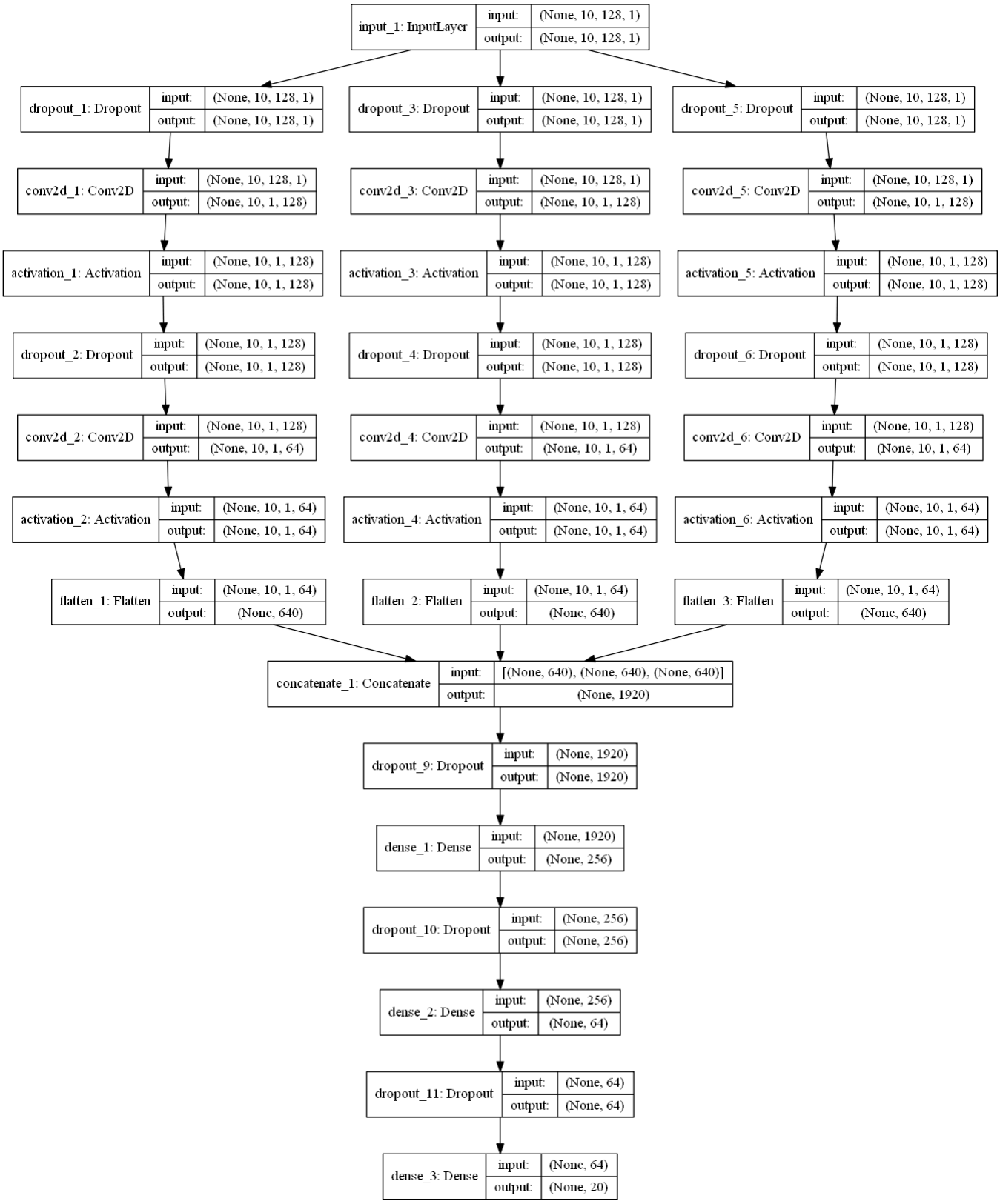


Figure D.3: Multi-Label CNN Final Architecture
(Figure created using Keras)

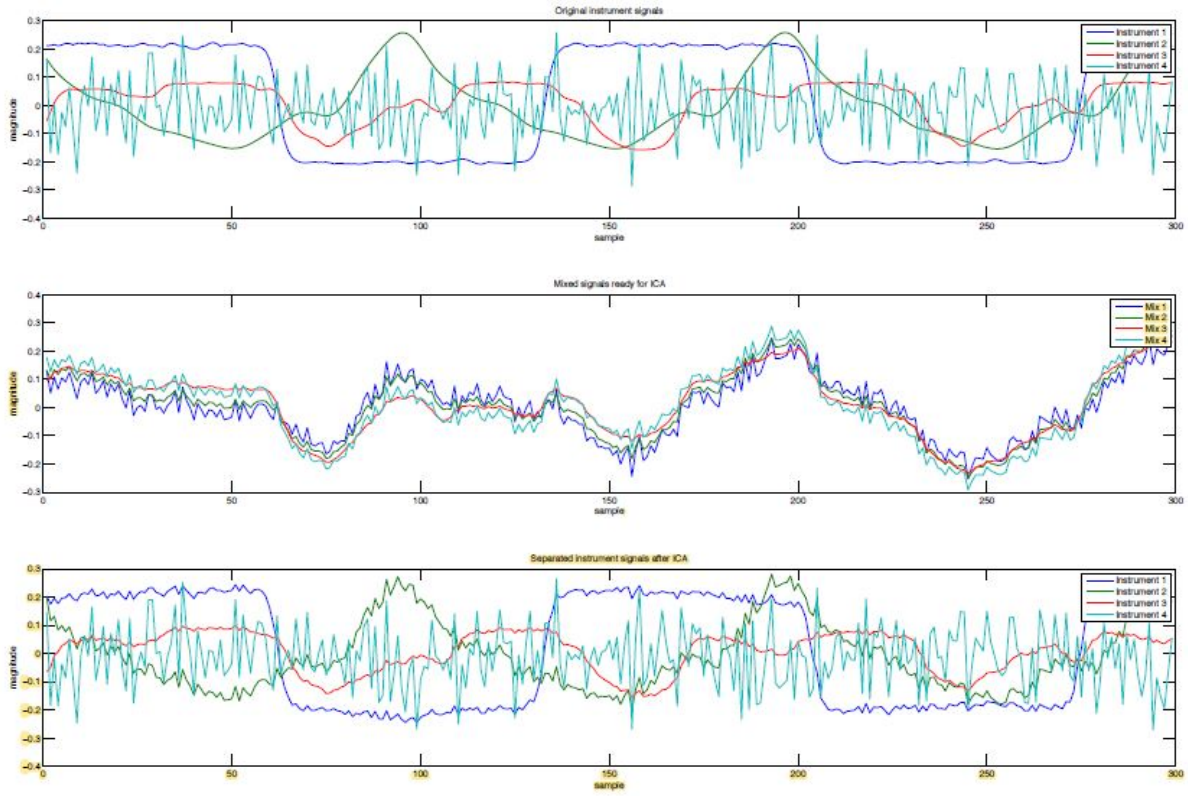


Figure D.4: Original 4 instrument samples, mixed recordings, and separated results from ICA[8]

Dropout 1/Dropout 2	***0.00***	***0.10***	***0.25***	***0.50***
0.00	0.902 (0.024)	0.892 (0.023)	0.878 (0.028)	0.861 (0.022)
0.10	0.878 (0.049)	0.883 (0.029)	0.894 (0.008)	0.880 (0.029)
0.25	0.909 (0.028)	0.876 (0.045)	0.883 (0.031)	0.849 (0.054)
0.50	0.897 (0.040)	0.876 (0.056)	0.870 (0.051)	0.884 (0.018)

Table D.1: Single-Label Network Accuracy and (std. dev) comparison using number of fully connected hidden layers and neurons

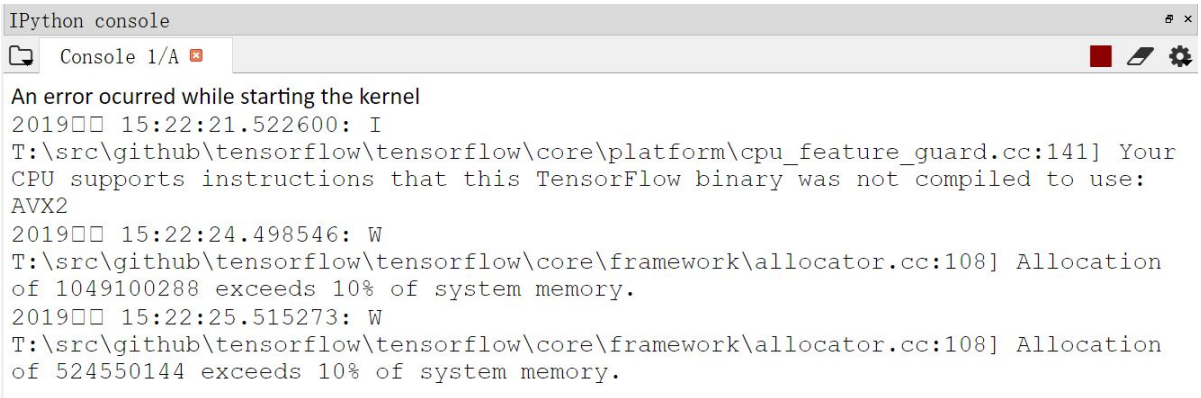


Figure D.5: Error Code

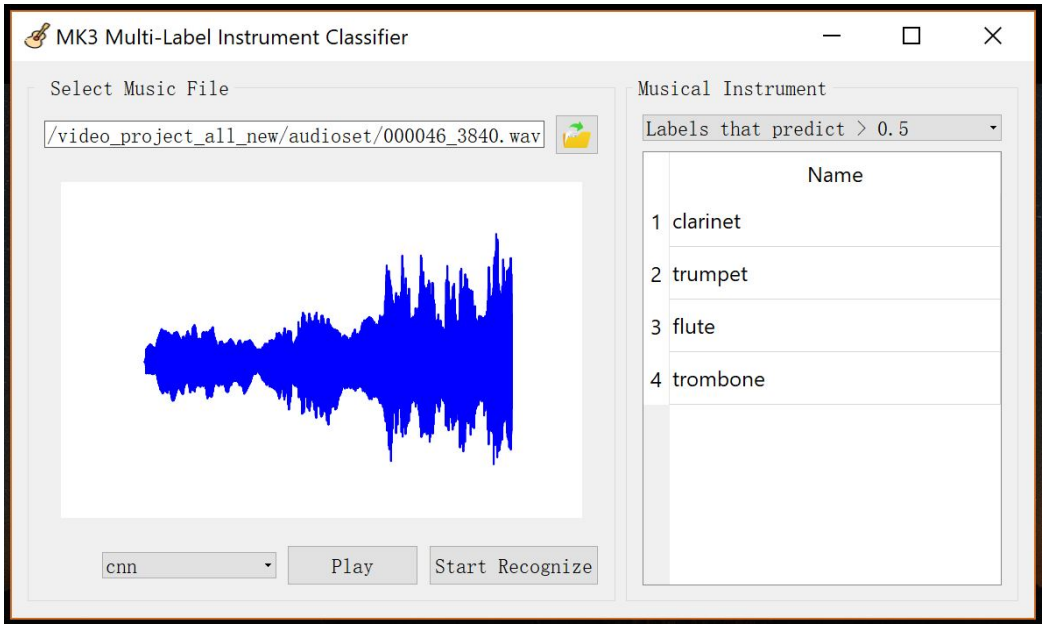


Figure D.6: Predominant Instruments in a 10s Test Music File

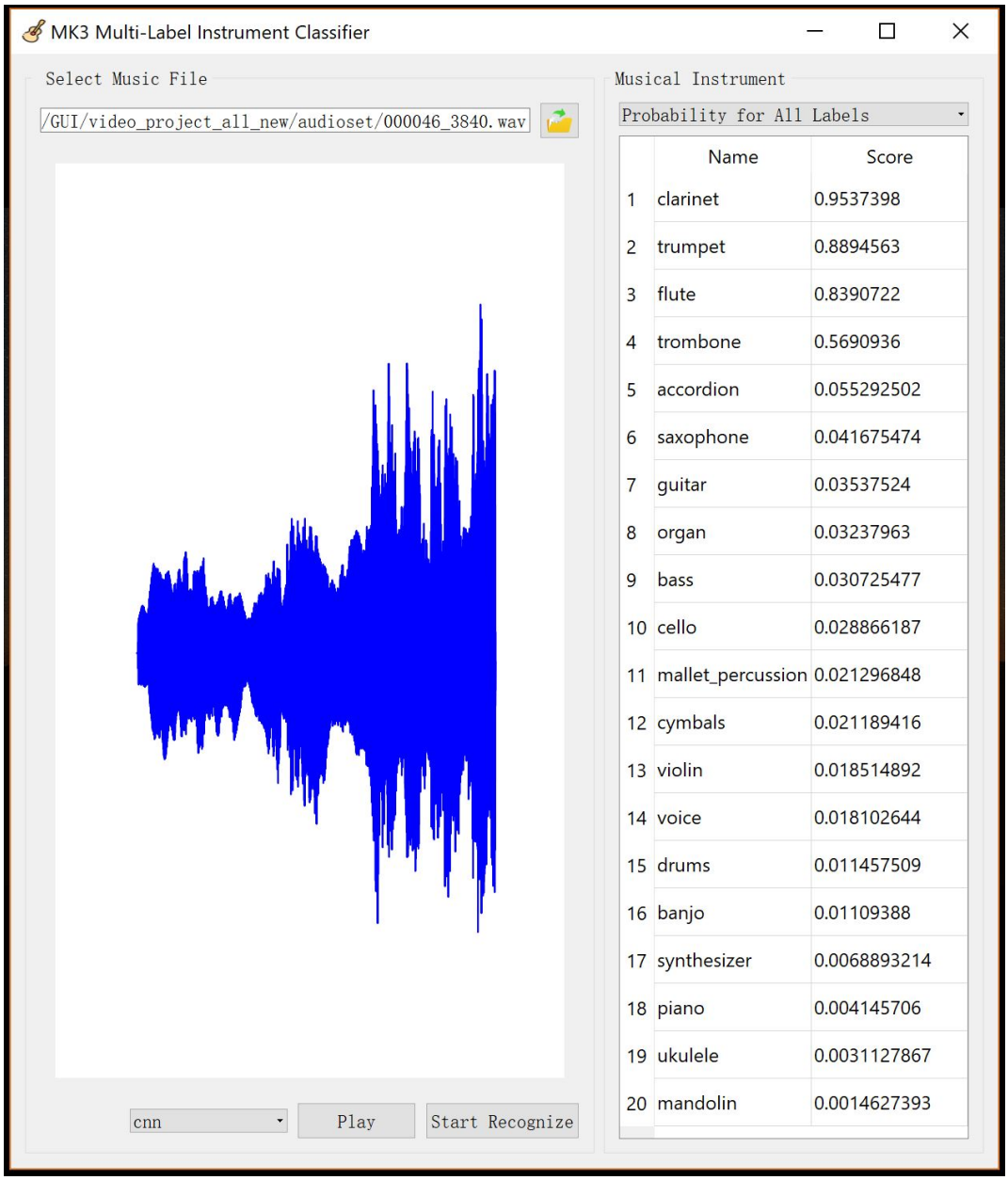


Figure D.7: Probability for All Instruments (10s file)

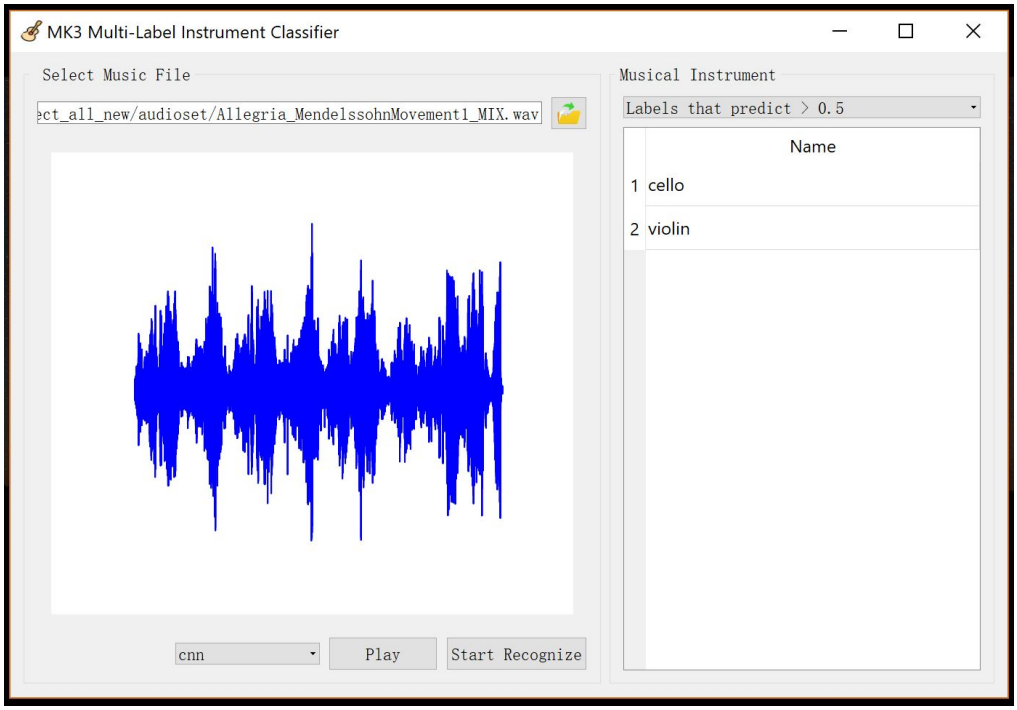


Figure D.8: Predominant Instruments in a 10 min 40 sec Test Music File

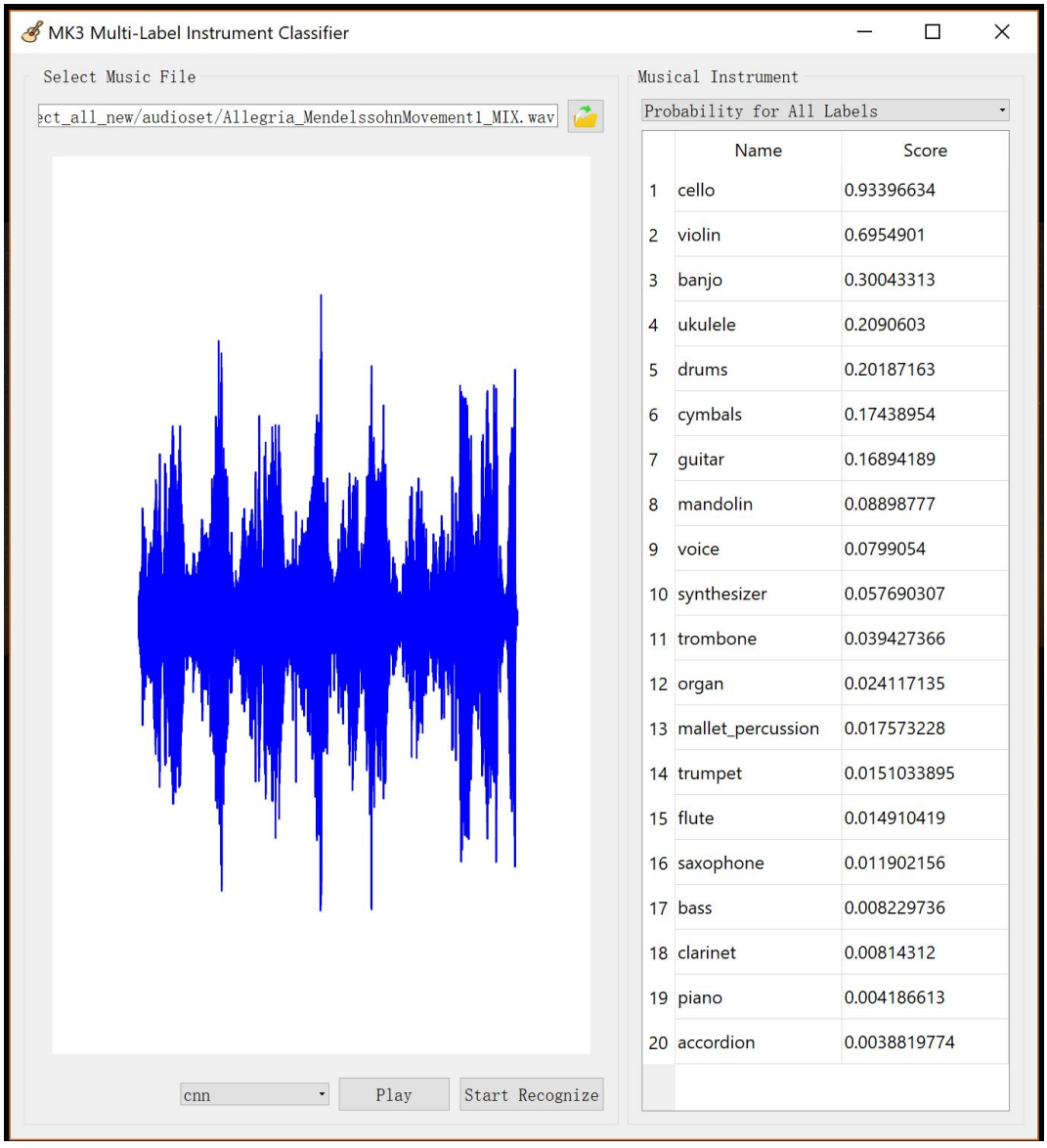


Figure D.9: Probability for All Instruments (10:40 file)

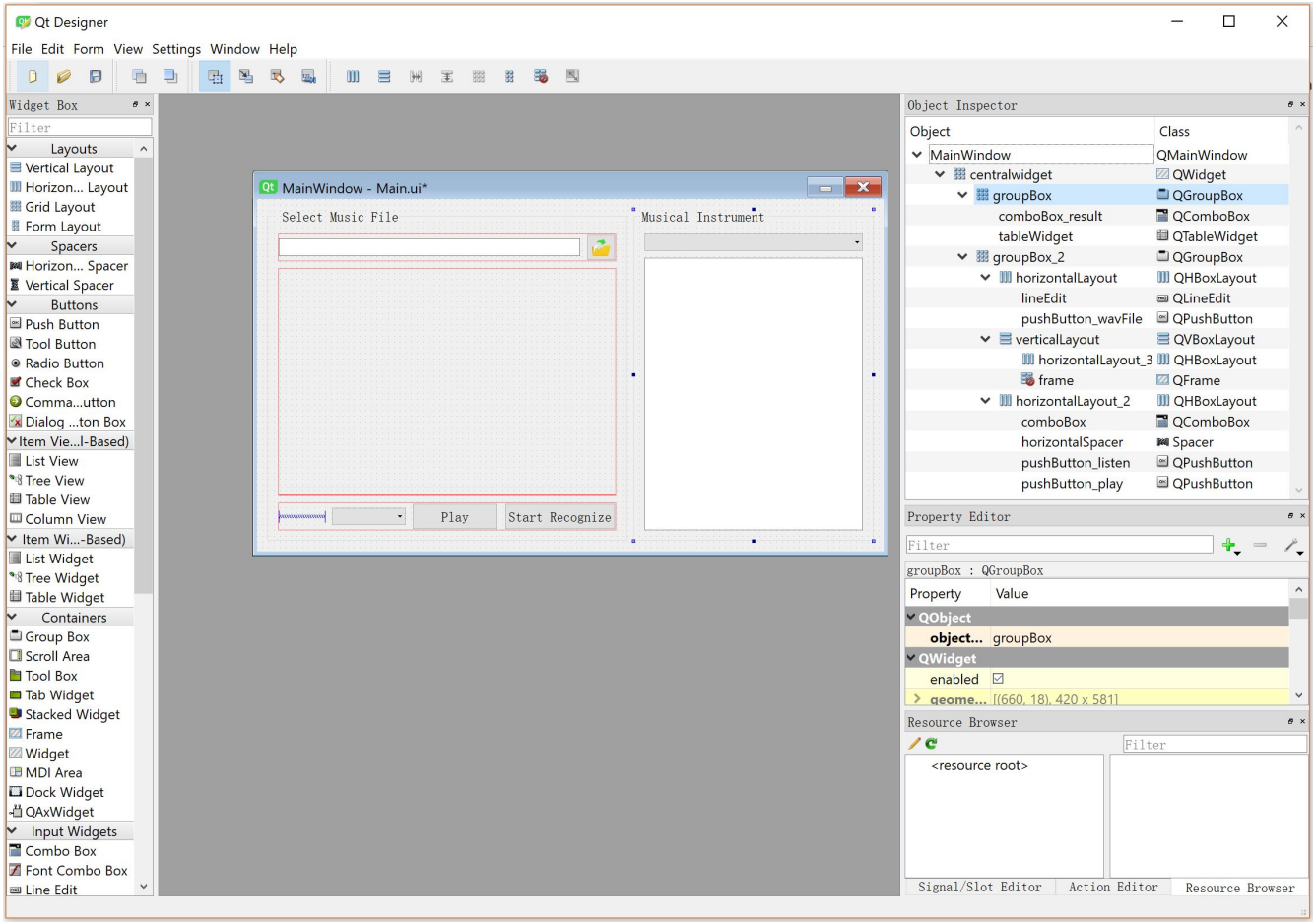


Figure D.10: Open 'ui' File in Qt Designer

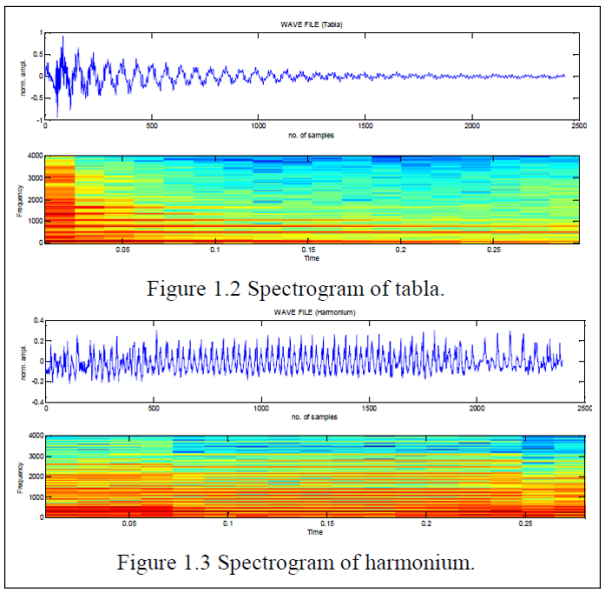


Figure 1.2 Spectrogram of tabla.

Figure 1.3 Spectrogram of harmonium.

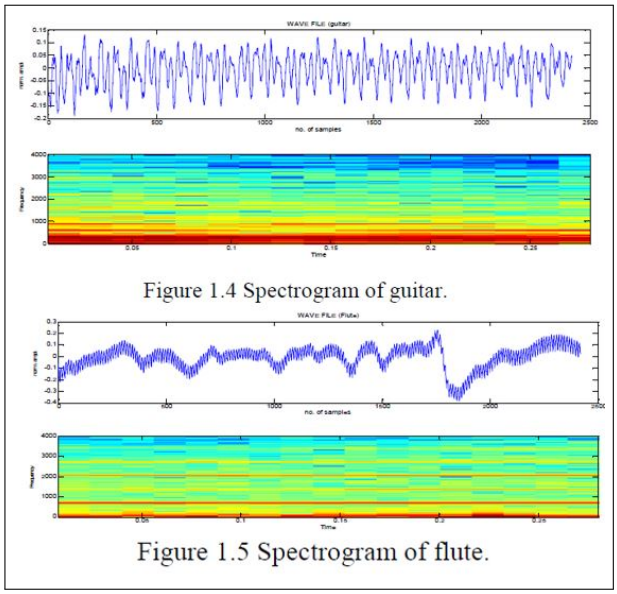


Figure 1.4 Spectrogram of guitar.

Figure 1.5 Spectrogram of flute.

Figure D.11: Banchhor and Khan[7] in 2012 four instruments recognition using Spectrogram