School of Electronic Engineering and Computer Science

MSc in Digital Signal Processing
Project Report 2016

Automatic Musical Instrument Recognition Using Convolutional Neural Networks

Nevo Segal

Queen Mary University of London

August 2016

Abstract

This paper presents a novel approach to Automatic Musical Instrument Recognition (AMIR) of solo recordings based on Convolutional Neural Networks (ConvNets). Previous AMIR methods are based on designing and extracting hand-crafted features from the audio signal in order to describe its characteristics and classify it. In contrast, ConvNets learn the features directly from the input data. This method has proven successful in solving various complex Multimedia Information Retrieval (MIR) problems, such as image classification or voice recognition. This project seeks to explore whether this success can be imported to AMIR as well. The results from this study show a 28.3% top-1 error rate and a 63.9% average F-measure score when combining the results of all instrument classes. These are encouraging results, especially considering the small amount of available data used for training the system.

Table of Contents

Table of Figures

Table of Equations

Table of Tables

1. Introduction

Automatic instrument recognition has been at the heart of audio research for many years. Solving this problem would allow advancements in related areas such as automatic transcription, source separation and many more. Although it seems like a straightforward problem, computers struggle in this task even for basic audio recordings that consist only of a single instrument.

Many algorithms and techniques have been applied to tackle this problem, such as the ones discussed in Eronen and Klapuri (2000), Heittola, Klapuri and Virtanen (2009) and Essid, Richard and David (2004). As with many other Multimedia Information Retrieval (MIR) tasks, the approaches used in this area often consist of extracting hand-crafted features from the audio signal, which are subsequently fed to a classifier in order to eventually determine the correct instrument. Designing these features is a difficult task: it requires vast knowledge in signal processing and data analysis, and results in feature extractors that perform well under very specific conditions, but often fail to generalize.

1.1. Motivation

The advancements achieved by Convolutional Neural Networks (ConvNets), most notably the results achieved by Krizhevsky, Sutskever and Hinton (2012) in image classification of the ImageNet database, is what inspired me to try and apply ConvNets to instrument recognition.

In many ways, instrument classification is very similar to object recognition in images: both contain one or more objects/instruments, both could come in different qualities and in both the object/instrument can appear in different positions of the image/song. There are also some differences between the two tasks. Perhaps the biggest one is that an image doesn't evolve over time while sound does. In order to tackle this difference, the spectrogram, a time-frequency domain representation of the audio signal is used in this project.

As mentioned, Deep neural networks, and ConvNets in particular, have shown incredible performance when faced with complex problems. Despite that, there has been little research seeking to apply these techniques to Automatic Musical Instrument Recognition (AMIR). This project investigates whether ConvNets can improve the current state-of-the-art in this field.

1.2. The project

This project uses ConvNets to tackle the problem of AMIR, focusing on recordings consisting of single instruments. It is implemented in Python, using *TensorFlow*: Google's machine learning library. The aim of this study is to be able to discriminate between 12 different pitched and percussive instruments. A limited number of solo recordings are used in order to train, validate and test the system.

1.3. About the report

The report consists of the following sections: background (chapter 2), design (chapter 3), implementation (chapter 4), evaluation (chapter 5) and conclusion (chapter 6). Chapter 2 lists and investigates previous research done in the field of AMIR and ConvNets. The next chapter presents the proposed system architecture, along with the data collection techniques. In chapter 4, the pre-processing, training and testing decisions are discussed. Next, the results are presented and compared to other AMIR algorithms. Lastly in chapter 6, the results of the project will be discussed as well as possibilities of future work.

2. Background

2.1. Instrument recognition

Being able to automatically extract and analyse meaningful content from audio waveforms has many practical applications. To name a few, automatic speech recognition (ASR) allows us to control computers using solely our voice, automatic transcription allows musicians to easily transcribe audio recordings, and so on.

This project looks into another such application, AMIR. As mentioned earlier, we have witnessed great advancements in this area. This is mainly thanks to work carried out in artificial deep neural networks, which will be discussed later in this chapter. Despite this progress, AMIR algorithms have yet to achieve human-comparable performance, and hence it remains an unsolved problem. Although AMIR may seem elementary at first, especially compared to other complex real-world problems, it is not. The main reason human beings find it simple to distinguish between musical instruments is because of the competence of the human auditory system (Fuhrmann, 2012). Computers do not possess this competence and therefore find this task extremely complex. The following section will introduce recent research carried out in the field of AMIR.

2.1.1. Previous work

Many algorithms have been developed in the context of AMIR. After looking into the most common techniques discussed in the literature, we can see that although a lot of them use different paradigms and algorithms, the vast majority are based on extracting hand-crafted features.

Marques and Moreno (1999) discuss their research of instrument recognition, in which instruments are being classified using a support vector machine (SVM) classifier. In their paper, they discuss the features used for classification: Linear Prediction Coefficients (LPC), cepstral coefficients and Mel-Frequency Cepstral Coefficients (MFCC). In this project, they chose to use only spectral features. The evaluation result shows that approximately 70% of the audio waveforms were classified correctly using this technique. In another paper, Eronen and Klapuri (2000) develop another approach to AMIR, in which they design and extract over 20 different features. Among them, they use both temporal and spectral features, in the hope that such combination would create a "more robust instrument recognition system than described in experiments so far" (Eronen and Klapuri, 2000: p.1). These features include several variations of MFCC, spectral centroid, attack and decay time, and many more. Even with the use of a large set of temporal and spectral hand-crafted feature extractors, the classification results obtained in this study are still far from those a human achieves, with individual instruments successfully classified only approximately 80% of the time.

Feature extraction is key in all areas of MIR algorithms. As evidence, many feature extraction libraries were developed (for example see Rawlinson, Segal and Fiala, 2015 and Mathieu et al., 2010), with the sole purpose of providing users with access to a variety of spectral, temporal and perceptual features. Furthermore, in his paper, Eronen (2001) lists, describes and compares different hand-crafted features specifically designated to be used in AMIR. Other papers (El Ayadi, Kamel and Karray, 2011; Haralick, Shanmugam and Dinstein, 1973) compose similar "feature cocktails" for other MIR applications.

Eggink and Brown (2004), Essid, Richard and David (2004a, 2004b), Livshin and Rodet (2004), Heittola, Klapuri and Virtanen (2009) and many more, all use feature extraction as a key step in their classification paradigm. I observe that most often these attempts differ from one another mainly in the combination of features used, along with the type of classifier. It is clear however, that these implementations, that use hand-crafted features at their core, obtain accuracy results that are significantly lower than human performance.

There has been little research into systems that do not use hand-crafted features in the area of AMIR. Deep neural networks are such systems, as they learn their own features directly from the input data. According to Choi, Fazekas and Sandler (2016), deep learning algorithms became the de facto standard in the area of computer vision in recent years. They have been applied to auditory data as well, showing better results over the previous state-of-the-art in areas such as automatic chord recognition. This project explores whether the recent success deep learning had in other fields can be imported to AMIR. The following sections will describe what artificial deep neural networks are, also known as deep learning. We will focus on a specific type of deep learning architecture, namely ConvNets, which is the one used in this project.

2.2. Deep neural networks

Artificial Neural networks (ANNs) are a group of computational models. ANNs were originally inspired by biological structures, especially the structure of the brain, because of brains' ability to solve complex problems (Basheer and Hajmeer, 2000). ANNs got its name from its structure: many processing units, also known as neurons are organised in layers. Each layer is connected to the preceding and following layer, by that creating a network of connected neurons. Researchers use ANNs to solve difficult tasks, often problems that relate to our perception, that cannot be solved using simpler techniques.

As ANNs, and more specifically deep neural networks (ANNs with multiple hidden layers), became popular in the past few years, people tend to believe that it is a new technology. The truth is that the ideas and techniques behind deep neural networks can be tracked back to the 1940s (McCulloch and Pitts, 1943). The renewed attention around neural networks is largely due to two reasons. Firstly, we live in a world in which data is generated at a colossal frequency. According to Mary Meeker's annual Internet trends report, a staggering 1.8 billion photos are being uploaded to the Internet every day (Meeker, 2014). This explosion of data allows researchers to train their neural networks on large datasets, which was impossible before. Having a large quantity of versatile data significantly improves the accuracy of a neural network (Nielsen, 2016). Large datasets require powerful computers, which leads to the second reason: highly efficient, affordable Graphics Processing Units (GPUs). Thanks to the gaming industry, GPUs have become highly optimised and relatively low-priced. Along with some low-level libraries such as *Tensorflow* (Abadi et al., 2015) or *Theano* (Bergstra et al., 2010), these GPUs can be used to compute complex neural networks. To summarise: we can now train our neural network models on a large amount of data, using affordable, highly-optimised computers.

As mentioned, the neurons in the neural network are organised in layers. There are three types of layers: input, hidden and output. The input layer is the input data. In greyscale images for example, each pixel in the input image will be represented by a neuron in the input layer. The values of these pixels are passed on to the hidden layers. The hidden layers are in charge of transforming the input into something that would be usable by the output layer. The more we go up in the hierarchical chain of hidden layers, the more sophisticated the information it can represent. For example, let's say we are trying to detect cars in images. The first hidden layer might extract lines and edges, while the second one corners. Deeper hidden layers may be able to recognise more abstract shapes such as wheels and the shape of windows, passing this information to the output layer and allowing it to make an educated decision.

2.2.1. Traditional neural networks

Traditional neural networks are fully connected: each neuron is connected to all of the other layers' neurons (either directly or indirectly). When we say that a neuron is connected to another neuron, we mean that its output is being fed as the input to the neuron in the following layer. But how do compute this output? Mathematically, it simply takes a few multiplications and additions per neuron. However, it is often easier to think of this layer-wise, rather than neuron-wise by working with matrix multiplication (Nielsen, 2016). Consider the following equation:

$$y^j = \sigma(w^j y^{j-1} + b^j)$$

Equation 2.1: The output of layer j in a traditional, fully connected neural network

Here we can clearly see the mathematical operations needed in order to compute the output of a layer j. We take the output vector from the previous layer y^{j-1}, multiply it by a weight matrix w, add a bias vector b, and apply a non-linear activation function σ. The activation function will be discussed later in the report.

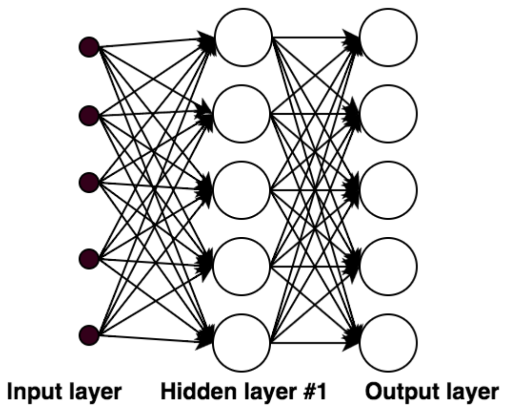


Figure 2.1: A traditional, fully-connected neural network

This has several implications, among which is the large computational complexity that this entails. ConvNets work slightly differently to traditional neural networks, as will be explained in the following section.

2.2.2. Convolutional neural networks

Simply put, ConvNets are neural networks that use two-dimensional convolution instead of matrix multiplication to compute weights. Although it may seem like this is not a major difference, this small change is actually significant. As mentioned, traditional neural networks are fully connected; all of the output neurons are connected to all of the input neurons (through hidden layers or otherwise). Applying matrix multiplication on large inputs results in inefficient computational complexity, along with a large memory consumption. By using convolution instead of matrix multiplication, ConvNets improves this.

A thorough guide to the convolution operation is outside the scope of this report. However, in order to understand how ConvNets work and what is the motivation for using convolution, a brief introduction is necessary.

2.2.2.1. Convolution

Convolution is a key concept in signal processing, used in several domains such as audio analysis, image processing and as will be shown here – neural networks. The basic idea behind convolution is taking some input signal x(t) sliding a filter1 h(t) along its samples on one or more axes. At each input sample we point-wise multiply the filter with the input signal and sum, resulting in an output sample. These operations can be concisely written using the following equation for convolution:

$$(x * h)[n] = \sum_{m=-\infty}^{\infty} x[m]\, h[n-m]$$

Equation 2.2: 1-Dimensional convolution

[1] The terms filter and kernel will be used interchangeably.

Where *x* is the input signal and *h* is the filter.

Convolution is often used to apply filters and effects on signals, such as smoothing or sharpening. This exact operation can be applied over multiple axes as well. For example, when processing digital images, we can apply a 2D filter and slide it over the two axes of the image (assuming grayscale).

2.2.2.2. Motivation

Convolution is a mathematical operation with many real-world applications. But how can it assist neural networks? Well, the use of convolution in the context of neural networks utilises two beneficial concepts, which will be discussed below.

Sparse Connectivity

Meaningful features are small and do not usually cover the whole of the image. For example, an image can have millions of pixels, but an edge or a corner in that image would be represented using only a handful of them. Convolution allows us to detect and extract these features, by convolving the image with a small kernel. Therefore, kernels used in Convnets are often smaller than the image size by orders of magnitude, such as 3x3 or 5x5 pixels. In terms of computational complexity and memory usage, this improves things drastically. *figure 2.2* is a graphical illustration of this concept.

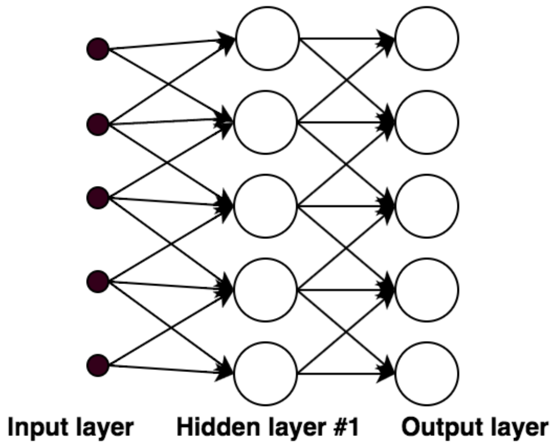


Figure 2.2: Basic system architecture of a convolutional neural network

As can be seen in the image, there are much fewer connections in our neural network compared to *figure 2.1*. Using matrix multiplication, a system with M inputs and N outputs will require MxN parameters and hence, its computational complexity will be O(MxN) per training example. Limiting the number of connections to our kernel size K, where K is significantly smaller than M, our system will contain much fewer connections – approximately KxN, which leads to less memory consumption. Furthermore, the number of operations needed per example data also decreases significantly to O(KxN) (Goodfellow, Bengio and Courville, 2016), which results in a much quicker runtime.

Parameter Sharing

Another important concept, directly resulting from the use of convolution instead of matrix multiplication is called Parameter Sharing. In convolution, we slide a kernel over a 2D input matrix in order to compute the output. Let us call each element in the kernel a parameter. Hence, a 3x3 sized kernel will have 9 parameters, a 5x5 will have 25 parameters, and so on. Every time we slide the kernel and use it to compute an output, we reuse these parameters. *Figure 2.3* provides a good graphical illustration of this. All the parameters with the same colour are shared; they use the same weight and bias value. This not only improves memory consumption when implemented efficiently, it also allows our features to be detected regardless to where they are located in the image – making them invariant to translation (Deep Learning, 2016). Making the features translation-invariant is one of the major assets convolutional neural networks have compared to other types of neural networks.

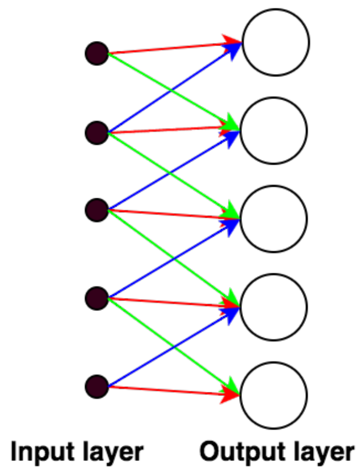


Figure 2.3: Shared weights on a simple neural network, using a 1x3 sized kernel

On the other hand, in traditional neural networks each parameter is used only once when computing the output, and is not shared across the inputs. Therefore, we need to learn a distinct set of parameters for every location on the input matrix (Goodfellow, Bengio and Courville, 2016).

2.2.2.3. Hyper parameters

A convolutional layer has several tuneable parameters. Perhaps the most important one is the kernel size. Choosing a good kernel size is important, as it dictates the size of the features extracted by the system. If the kernel is too small, complex patterns are likely to be missed. Too large and the features might become too broad. Another tuneable parameter is the number of features. Too many features would add a lot of redundancy to the network, while too few features would not have enough data to represent the various patterns. All of the above scenarios might result in sub-optimal training and testing accuracy.

3. Design

3.1. TensorFlow

TensorFlow is an open-source library originally developed by Google. It allows users to construct and run computational models easily and efficiently. It is primarily used in machine learning and deep neural networks research. There are a few reasons for choosing *TensorFlow* over other equivalent libraries. Firstly, the same code can run both on a CPU and on a GPU. All of the heavy-lifting required to write and optimise code for a GPU is done behind the scenes, and is completely transparent to the user. As I don't have experience coding for GPUs, this feature was a great advantage. The fact that the largest multi-national technology corporate is behind this library is also a major advantage. A very large community has already been formed, features are added and bugs fixed by the minute. Google's Brain team, the team that is responsible for most of Google's machine learning abilities, are the ones who developed this library for their own purposes. It is being actively used in many of Google's technologies and hence proved to be useful solving real-world problems.

3.2. Model architecture

The model used in this project is based on research in the areas of computer vision and music, specifically the ImageNet image classification task (Krizhevsky, Sutskever and Hinton, 2012), along with trial and error experimentation. The model consists of four convolutional layers, followed by one fully connected layer and one softmax layer. It has been shown that the deeper the convolutional layer is, the more abstract the features it learns (Bengio, Goodfellow and Courville, 2016). I have experienced this during my experiments as well. Very shallow networks performed significantly poorer than deeper ones, as they only consist of the analysis of very basic features.

There are several decisions to be made when designing a neural network. These include but are not limited to:

- The kernel size and number of features in each convolutional layer.
- The type of activation function.
- The method of pooling if any.
- The loss function.
- The loss function optimiser and the learning rate.

In the following few paragraphs, I explain these concepts and discuss the choices made in constructing the neural network model for this project. Please refer to *figure 3.3* at the end of this section for an illustration of the full system architecture.

3.2.1. Convolution

The kernel size used in all convolutional layers of this model is 3x3 samples. Furthermore, esearch suggests (Han, Kim and Lee, 2016) that increasing the number of output features according to the depth of the layer can achieve better performance in some cases. Others use a constant value throughout all of the hidden layers. Using the former approach didn't prove useful in the case of this project. Hence, the decision was made to extract a fixed number of 64 features from each layer.

3.2.2. Activation function

Most real-world applications, or at least the interesting ones, are nonlinear. As seen in section 2, the output of the convolution computation is a linear combination of weights, inputs and biases. Hence, in order to be able to resolve nonlinear problems, a nonlinear transformation is required. The activation function does exactly that: apply nonlinearity to our data in order to enable our network to solve complex, real-world problems.

The choice of the activation function heavily impacts the accuracy of the system (Glorot and Bengio, 2010). Up to the emergence of deep learning a few years ago, the tangent (tanh) and sigmoid function have been the most commonly used one (Han, Kim and Lee, 2016). Lately it has been shown that, in many cases, Rectified Linear Units (ReLU) achieve superior results. A ReLU is equivalent to half-wave rectification, in which all negative values in the input become zero in the output. There also exist some variations of this function. Leaky ReLU is a commonly used one, in which negative values are multiplied by a very small number (e.g. 0.001), instead of being completely zeroed out. In some contexts (Han, Kim and Lee, 2016; Bengio, Goodfellow and Courville, 2016), this technique performs better than the standard ReLU. After some experimentation I found that in this project the leaky ReLU actually decreases the performance of the system, hence the standard ReLU seemed like a better choice. Each output from the ReLU is passed through a pooling step. Pooling is a way to "summarise" the data: reducing its dimensionality while keeping most of the important variations in it intact.

3.2.3. Pooling

Applying a pooling step is advantageous for two main reasons. Firstly, each pooling step reduces the size of the data. The amount of reduction depends on the size of the analysis window, as well as the distance between adjacent analysis windows (stride). Secondly and perhaps more importantly, applying a pooling step helps making our data invariant to translation. In fact, according to Bengio, Goodfellow and Courville, the pooling step is so powerful that by using it "the features can learn which transformations to become invariant to" (2016: p.343). *Figure 3.1* well illustrates this concept. In this example we use a stride of one pixel and a window size of three pixels. Although the input values are translated one pixel to the right, effectively changing all of the input, half of our output remains identical (Bengio, Goodfellow and Courville, 2016).

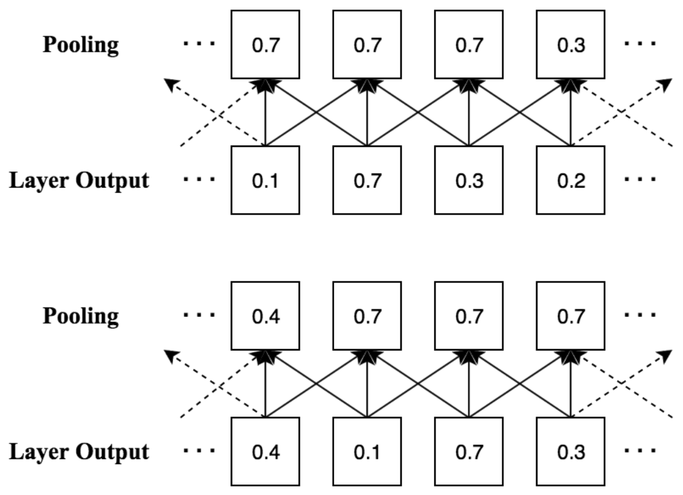


Figure 3.1: Translation-invariance in the pooling step

There are many types of pooling functions to choose from. In the context of deep learning, the most common one is max-pooling, and this is the one used in this project. In max-pooling, only the largest sample value inside a window is kept, discarding the rest. *Figure 3.2* is a visualisation of max-pooling algorithm. It is important to mention that other pooling functions exist, such as average pooling, which is analogue to smoothing the data.

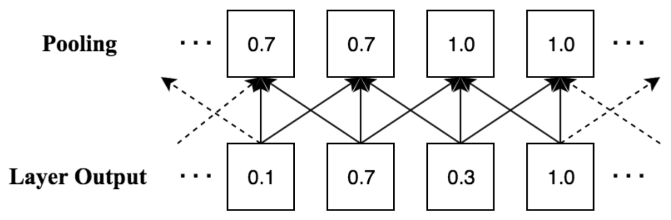


Figure 3.2: Max-pooling

3.2.4. Dropout

ConvNets can be very robust when trained with a large dataset, and when using the right parameters and hyper-parameters. A common issue with neural networks in general and ConvNets in particular is overfitting. Overfitting is defined as when a neural network obtains very high accuracy when evaluating the training set, but fails to achieve so when presented with new, previously unseen data. In other words, the network model does not generalize well. In 2014, Srivastava et al. discussed a new technique that helps preventing a neural network from overfitting - dropout. In the two years since the paper was published, dropout has become the preferred method to avoid overfitting by many researchers.

The idea behind it is simple yet effective. On every batch of training data, random neurons are removed from the network along with their connections to and from other neurons. Training a neural network with dropout is equivalent to training using a subset of slimmer networks, while each slimmed network is used very rarely (Srivastava et al., 2014). By doing so the learned weights do not assume a fixed network structure, improving robustness and reducing co-adaptation. Dropout is set by probability. In this project, a dropout of 0.5 is used after the second, third and fourth convolutional layers, as well as in the fully-connected layer. This means that there is a 50% chance that some neuron will be kept.

3.2.5. Loss function

The purpose of every neural network is to try to minimise some loss function. There are many functions to choose from such as mean-squared error (MSE) and Euclidean distance. In state-of-the-art implementations, MSE and Cross-Entropy (CE) have proved to be the most useful and hence became very popular. There has been some research comparing how these two loss functions perform in different scenarios (see for example Golik, Doetsch and Ney, 2013), which suggests that most often CE performs better. Having this in mind, I chose CE to be the loss function of the model.

3.2.6. Optimiser

In order to minimise the loss function, we use an optimiser. Gradient descent (GD) has been the de facto standard. Many variants, such as Stochastic Gradient Descent (SGD), have been developed. These often run faster than the classic GD, however this is at the cost of a reduced accuracy, especially when using a fixed learning rate (Ruder, 2016). These GD variants might not find the global minima: they get stuck on some local minima instead. Therefore, some optimisations of these GD variants were introduced in recent years, especially by the deep learning community. These try and overcome the downsides of the GD variants while still keeping the fast computational time offered by them. One such optimiser is called Adaptive Moment Estimation (ADAM). The abovementioned study of ConvNets-based ImageNet classification (Krizhevsky, Sutskever and Hinton, 2012) used a SGD optimizer. A few years later, ADAM was introduced and proved to operate better than SGD in most scenarios, including ConvNets (Kingma and Lei Ba, 2014; Ruder, 2016). In light of these recent developments, the ADAM optimizer was used in this project.

3.2.7. Softmax

The last layer is a softmax layer. The softmax function is a normalized exponential function (see *equation 3.1*). The output of this layer is a list of probabilities that sum to one. The size of the list is equivalent to the number of classes in the data, with each list element representing a class. The element with the highest probability represents the predicted instrument.

$$s[y_i] = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Equation 3.1: The softmax function

In this model we are trying to discriminate between 12 classes, each representing a different musical instrument. Softmax classification is by far the most popular type of classification layer, used in various classification contexts (Krizhevsky, Sutskever and Hinton, 2012; Glorot and Bengio, 2010; Lostanlen and Cella, 2016; Park and Lee, 2015b). Hence why I have decided to use it in this project.

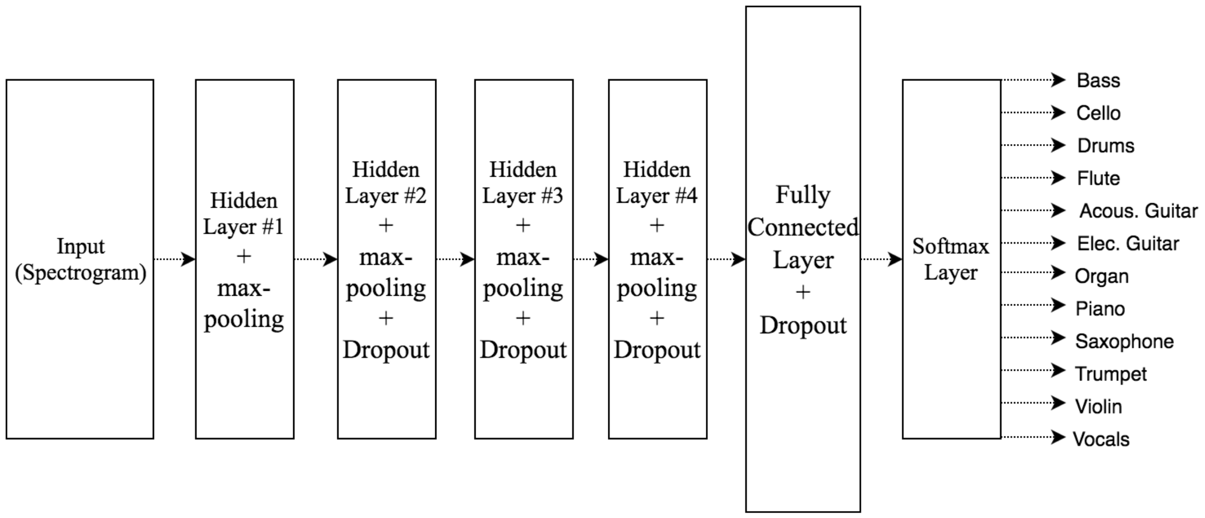

Input (Spectrogram)
Hidden Layer #1 + max-pooling
Hidden Layer #2 + max-pooling + Dropout
Hidden Layer #3 + max-pooling + Dropout
Hidden Layer #4 + max-pooling + Dropout
Fully Connected Layer + Dropout
Softmax Layer
Bass
Cello
Drums
Flute
Acous. Guitar
Elec. Guitar
Organ
Piano
Saxophone
Trumpet
Violin
Vocals

Figure 3.3: System architecture

3.3. Data collection

As mentioned, ConvNets shine best when training them with large datasets. The huge amount of data produced in the Internet era is the main reason for them being so popular again. Krizhevsky, Sutsteker and Hinton (2012) ImageNet classification algorithm used a large dataset of 1,000,000 different images for training and achieved incredible results. This shows that with sufficient data ConvNets can exceed the state-of-the-art even for difficult tasks.

Having this in mind, I started this project knowing that data collection would be key for its success.

3.3.1. Datasets

3.3.1.1. IRMAS [2]

A dataset curated by the Music Technology Group (MTG) at Pompeu Fabra University (UPF) in Barcelona. This dataset was created for instrument recognition, and it is very well structured for this purpose. Overall, it consists of 6,705 audio 3-second excerpts from 2,000 distinct recordings. The data covers 11 pitched instruments. The instruments included are: violin, cello, electric guitar, acoustic guitar, trumpet, saxophone, flute, clarinet, organ and piano and lastly, human voice. The database includes audio snippets from various recording scenarios and in different qualities.

The biggest drawback of this dataset is that it was highly unbalanced. For example, there are 778 excerpts of human voice but only 388 excerpts of cello. Research carried out on the implications of training a model on unbalanced datasets suggests that this could significantly decrease the performance of the system (Masko and Hensman, 2015). Hence, some audio snippets were discarded in order to make the database balanced across classes. Finally, 388 excerpts from each of the 11 instrument classes were left, making it an overall of 4,268 excerpts for training and validating the model. At that point it was clear that more data had to be gathered: not only the dataset was too small to achieve any significant results, it was also missing core musical instruments such as drums.

[2] For more information, please visit http://www.mtg.upf.edu/download/datasets/irmas

3.3.1.2. MedleyDB

The next step was to find more datasets and combine them with the existing set. Although not curated specifically for AMIR, MedleyDB (Bittner et al., 2014) was used. Created by NYU's Music and Audio Research Lab, MedleyDB consists of 122 songs with an overall of 52 different instruments. Each song directory contains the mixed song along with its stems: a separate audio file for each of the instruments in the song. A metadata file detailing what instruments each song contains is also available inside the song directory. This data was processed and cropped to 3-second snippet, each then added to its relevant directory according to the instrument class. As mentioned, this dataset consists of many types of instruments. Therefore, two more important instruments were added to the dataset: drums and electric bass guitar.

After adding MedleyDB, the merged dataset consisted of 11,609 audio waveforms, covering 13 different instruments. Clarinet, flute, trumpet and saxophone still suffered from insufficient amount of data, which will be discussed in the next section.

The use of song stems provides a lot of data. This technique however comes at a cost which is the repeatable nature of instruments in songs. When we examine individual instruments in songs, we often find them playing the same riff or beat throughout the whole song. In the context of neural networks and machine learning, this behaviour is not optimal, as it increases the chances of overfitting our model to the data. Nevertheless, with the lack of available data I decided to keep this data, using less of it than initially planned.

3.3.1.3. Other data

Some instruments still consisted of too few training examples. As no more relevant datasets were found, a manual search of individual files was performed. All the data that was found and used is licensed under a Creative Commons license[3]. Most of the files were downloaded from the *Jungle Vibe*[4] online database and the *QMUL Open Multitrack Testbed*[5] (OMT). Despite my efforts, not enough clarinet data was found and it was removed from the dataset. Same as earlier, the files were then cropped into 3-second snippets and placed in their appropriate directories. *Table 3.1* provides a summary of the number of examples used per dataset.

3.3.1.4. Testing data

All the abovementioned data were gathered exclusively for training and validating the model. Data for testing had to be assembled separately, in order to ensure the datasets used for training and testing do not consist of overlapping examples. *Good Sounds* (Bandiera, Romani Picas and Serra, 2016) by UPF contains recordings of musicians playing scales and single notes on several instruments. Recordings of flute, cello, saxophone, trumpet and violin from this dataset were used in the testing set. The missing instruments were manually collected from *Jungle Vibe*.

Purpose	**IRMAS**	**MedleyDB**	**OMT**	**Good Sounds**	**Jungle Vibe**	**Overall**
Training	3,660	7,496	911	0	1,492	13,559
Validation	332	520	48	0	660	1,560
Testing	0	0	0	1,149	2,076	3,225
					Total	18,344
					Per instrument	1,528

Table 3.1: A summary of the number of examples used per dataset

[3] For more information, please visit https://creativecommons.org/licenses/

[4] For more information please visit http://junglevibe2.net/

[5] For more information, please visit http://multitrack.eecs.qmul.ac.uk/

3.4. Pre-processing

Many song stems and improvisation recordings contain mostly silence. Simply using the cropped audio snippets would have added many empty audio files to the dataset, which is undesirable. Therefore, the RMS of each 3-second snippet was computed prior to cropping in order to get a rough indication of its loudness. If the RMS was above a fixed threshold, the snippet was added to the database. Otherwise, it was skipped. The threshold was selected after performing some trial and error.

4. Implementation

4.1. Overview

The project was implemented in Python using the *Tensorflow* library. Since the appearance of deep learning, Python has been a popular choice among many researchers and developers. *Keras*[6], *Lasagne*[7], *Theano* and *Tensorflow* are only a few examples of Python libraries that implement deep learning related algorithms.

Due to the high computational complexity, this project was computed on a GPU. An Amazon Web Services (AWS) g2.2xlarge instance was created. This instance type consists of a single physical NVIDIA GPU with 1,536 CUDA cores, 15 GiB of memory and 8 virtual CPUs. This configuration allowed me to run the model approximately 10 times faster than on simple CPU, which translates to hours instead of days.

As with most other machine learning implementations, this project consists of three major steps. Firstly, pre-processing the data. Pre-processing always has a crucial impact on the robustness of an algorithm. In this project, we transform the signal into a different domain and filter some of its content in order to obtain the most relevant representation prior to training the network. Once we have the pre-processed data, we begin training. In many supervised learning algorithms, such as the one in this project, training the network consists of iteratively feeding it with labelled training examples. Therefore, we train the network during several epochs, until it produces satisfactory accuracy on the training data. Lastly, we test the network, presenting it with previously unseen data and letting it classify it based on what it learned during training.

---

[6] For more information, please visit https://keras.io/

[7] For more information, please visit https://github.com/Lasagne/Lasagne

4.2. Pre-processing

Often, the time-domain representation of the signal doesn't provide us with a good indication about the contents of the audio signal. Hence, when analysing audio, it is useful to use the frequency-domain representation of the signal. However, this operation discards all temporal information, which is crucial in sound and music. To overcome this, we slide an analysis window over the signal, and compute the Fourier Transform (FT) on each. This operation is called the Short-Time Fourier Transform (STFT). Its output is a large matrix containing the frequency spectrum of the signal over time. This matrix is called the spectrogram, and it is the input data to the network.

4.2.1. Spectrogram

Using a spectrogram in ConvNets is simple and achieves good results in many use-cases. Many researchers (Choi, Fazekas and Sandler, 2016; Jang, Kim and Oh, 2014; Lostanlen and Cella, 2016) use this technique. As mentioned, the inspiration to this project was the ImageNet classification challenge. The winner of this challenge used ConvNets to beat the state-of-the-art. In order to apply a similar model to the one used in this challenge, it was necessary to transform the audio waveform to a 2D representation; the spectrogram was the natural choice. In order to compute the spectrogram, we use *Librosa*[8]: A Python library for audio and music analysis. A standard spectrogram uses linear frequency. As this project tries to simulate human perception, this is not appropriate. Hence, we use the the Mel-scale based spectrogram in this project. The Mel scale is logarithmic, and was created using comprehensive listening tests.

[8] For more information, please visit https://github.com/librosa/librosa

Differences between steps on this scale are equal in terms of human perception, and therefore are ideal for this project.

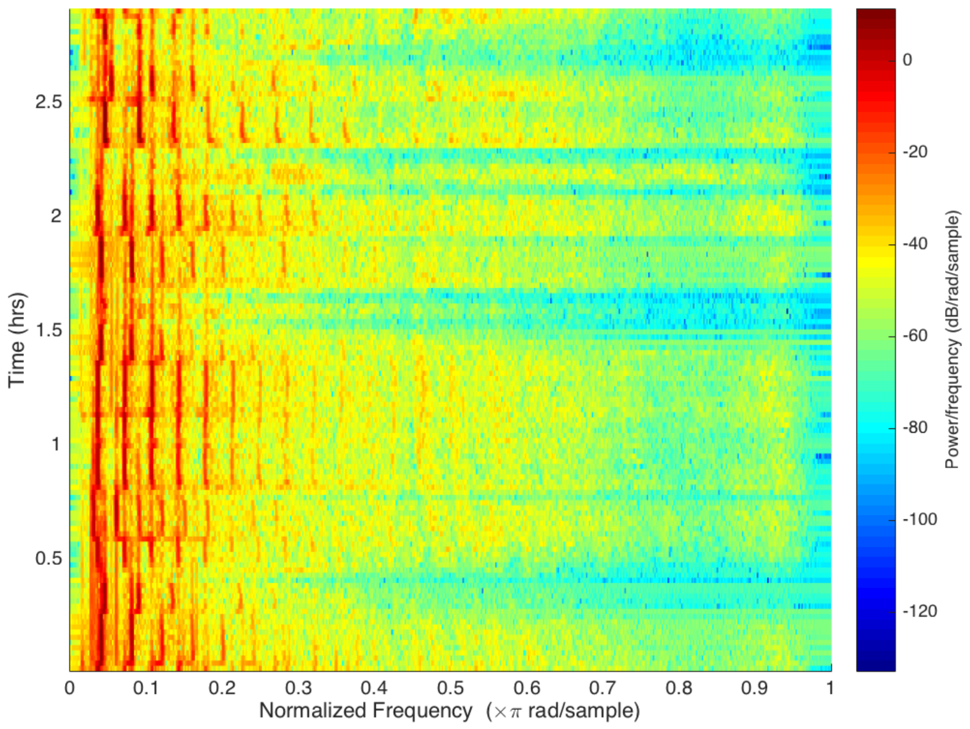


Figure 4.1: An example spectrogram

In music, the most important frequency information is often located on the lower part of the spectrum. The higher frequencies are often more susceptible to noise. Therefore, prior to computing the Mel-spectrogram, we apply a low-pass filter and resample the data from 44100Hz to 22,050Hz. By doing so, we only keep frequencies up to 11,025Hz (Nyquist frequency), and filter the rest. This removes noise and keeps the important information intact. As a by-product, this also significantly reduces the amount of memory required during run-time, which is often equally as important.

4.2.2. Data serialization

After computing the spectrograms, they are serialized into a single file using the *HDF5* format. Implemented via the *h5py*[9] library, this efficient file serialization format allowed me to easily store the spectrograms and their corresponding labels as dictionaries in one binary file. The *h5py* library also supports various compression algorithms, which helped me save time when transferring this large file to the server. By storing everything in one file, the data could easily be loaded, used and manipulated.

4.3. Training

Once we decided on our model architecture and have all our data serialized, the network could start the training stage. In neural networks, training is synonymous to learning the weights that "connect" the neurons. In each layer we learn a matrix of weights. These are convolved with the input tensors in each layer to produce an output. After every batch of training examples, the weights are updated. When updating the weights, we take into account the incorrect classifications. The system attempts to change the weights so that the loss is reduced and hence, the correct number of classifications is increased. As mentioned earlier, the loss function used in the project is CE. Later, during testing, the weights learned are used to classify unknown data. The following section will describe the implementation steps, and rationalize the decisions that were made along the way.

[9] For more information, please visit http://www.h5py.org/

4.3.1. Loading data

As the project consists of a large amount of data, a *DataLoader* class was implemented. This class implements a read pointer that iterates through the data, along with some utility methods to help manage the data more efficiently and neatly. The key methods of this class are:

- `load_next_batch()` – the data should be fed into the network in batches of equal size. This method returns the next available batch of data according to our read pointer. It is important to split the data into batches, as loading all examples at once requires an enormous amount of memory.
- `randomize()` – randomizes the order of the data. We shuffle the order on the start of every epoch in order improve the robustness of our network. I found that presenting the data at the same order on every epoch, increases predictability and was more prone to generate overfitting.
- `reset_read_pointer()` – as mentioned, read pointers are used to manage our batches queue. This method is called at the end of each epoch, after presenting all of the training examples to the network.

Creating a separate class for the I/O allowed me to create a more modular codebase which adheres to the DRY[10] (Do not Repeat Yourself) coding convention.

[10] For more information please read https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

4.3.2. Normalizing batches

Normalizing the data significantly reduces the time it takes the network to converge (Ioffe and Szegedy, 2015). We normalize each batch separately. By normalizing, we transform the data to have a zero mean, and a unit variance.

If we don't normalize the data, the optimizer has to search in many locations in order to find the global minima of our loss function. This could even result in the network not converging at all. A `BatchNormalizer` class was written in which we define a `normalize()` method. This calls *TensorFlow*'s `batch_norm_with_global_normalization()` method that performs the normalization per batch of examples. This method accepts a mean and a variance as the input parameters, to which I pass zero and one respectively.

4.3.3. Minimising the loss

As mentioned, we optimize the loss function using an ADAM optimizer. The role of the optimiser is to minimise the loss, which in our case is the CE between the ground-truth labels and the estimated ones. *TensorFlow* makes it easy to implement this: All it takes is a single line of code. One thing we need to decide about before running the optimizer is the learning rate. We can think of the learning rate as the step size the optimizer takes when searching for the global minima of the loss function. *Figure 4.2* (Wikimedia Commons, 2006) shows us an illustration of this. When this value is too large, the optimiser might miss the minima altogether. If too small, the system would be highly inefficient, and would take too long to find a solution. After some trial and error, the choice was made to use a learning rate of 1e-4. This value seems to give the best overall performance both in terms of testing accuracy, and in terms of computational efficiency.

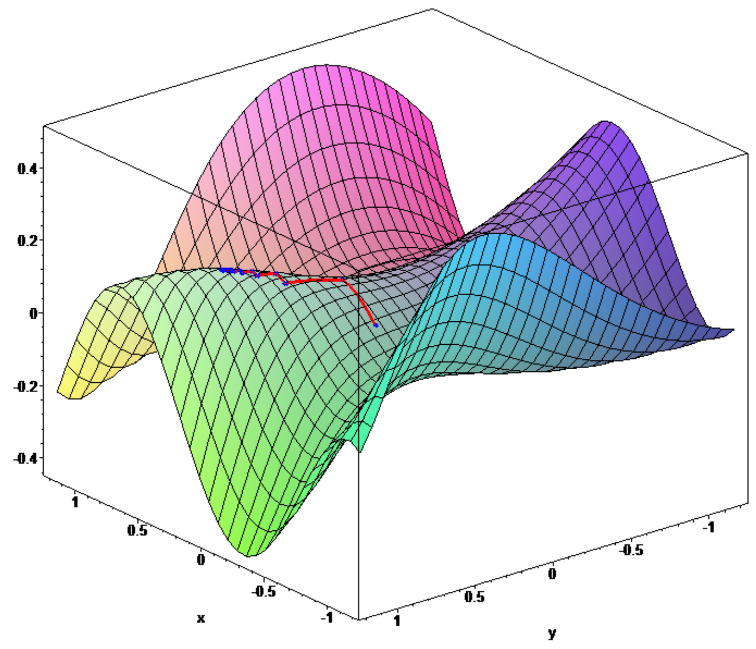


Figure 4.2: A 3-D illustration of the loss function and the optimizer

4.4. Testing

Once we have our trained model, we begin testing it using new audio examples. This section intends to explain how the testing is done from an implementation perspective. The evaluation results will be provided in the next chapter.

In many ways the testing is very similar to the training. The only difference in implementation is that we do not update the weights after feeding in a batch of audio examples. Instead, we evaluate their accuracy. The way we do this is based on one-hot encoding. In a machine learning context, one-hot encoding is a data representation technique that allows us to easily determine whether the classification was correct. In this method, each testing example is assigned with a label vector. This vector has the same number of elements as the number of musical instruments in our dataset. Also, each instrument is assigned with an index, from zero to N-1 where N is the total number of instruments. All the elements in the label vector of each audio snippet is set to zero, apart from the one that has the same index as the the instrument it contains. This method is used because the softmax layer outputs a vector of probabilities. These can then be easily compared with the one-hot encoding vector. We can also think of the one-hot encoding as a probability vector as well: we set the ground-truth instrument to a 100% probability, and all of the incorrect ones to 0%.

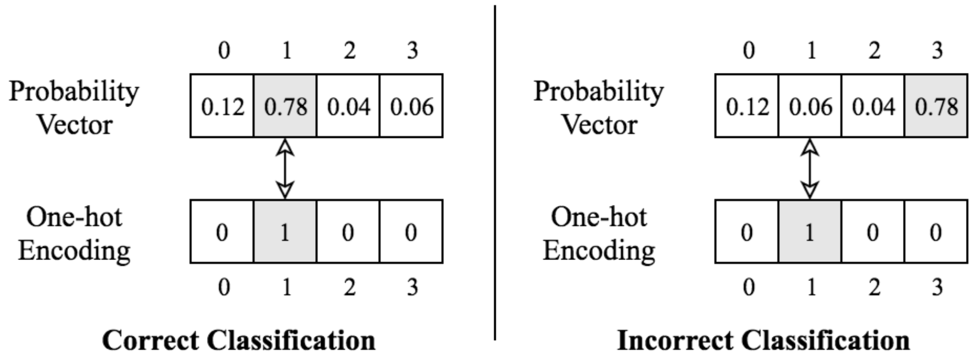


Figure 4.3: A comparison of a probability vector and a one-hot encoding vector

Once we have the ground-truth one-hot vector of the audio snippet and the probability vector predicted from our network, we compare them. If the highest probability on the probability vector is at the same index as the instrument in the one-hot vector, the classification was correct. Otherwise, it was not (see *figure 4.3* for an illustration of this).

5. Evaluation

This project is still in its infancy and there is still a lot of research and experimentation yet to be conducted. Despite this, we can already see the potential of using ConvNets in the context of AMIR. In order to evaluate the performance, two evaluation metrics were used.

- **Top-1 error rate** – the percentage of test data that was incorrectly classified. This gives us an idea of the overall performance of the system, regardless of the instrument type.
- **Confusion Matrix** – a per-class analysis of the system performance. From the confusion matrix we also derive the precision, recall and F-measure of each class. Formally, these are defined as:

$$P = \frac{TP}{TP + FP}$$

$$R = \frac{TP}{TP + FN}$$

$$F = 2 \cdot \frac{P \cdot R}{P + R}$$

Equation 4.1: Precision, recall and F-measure

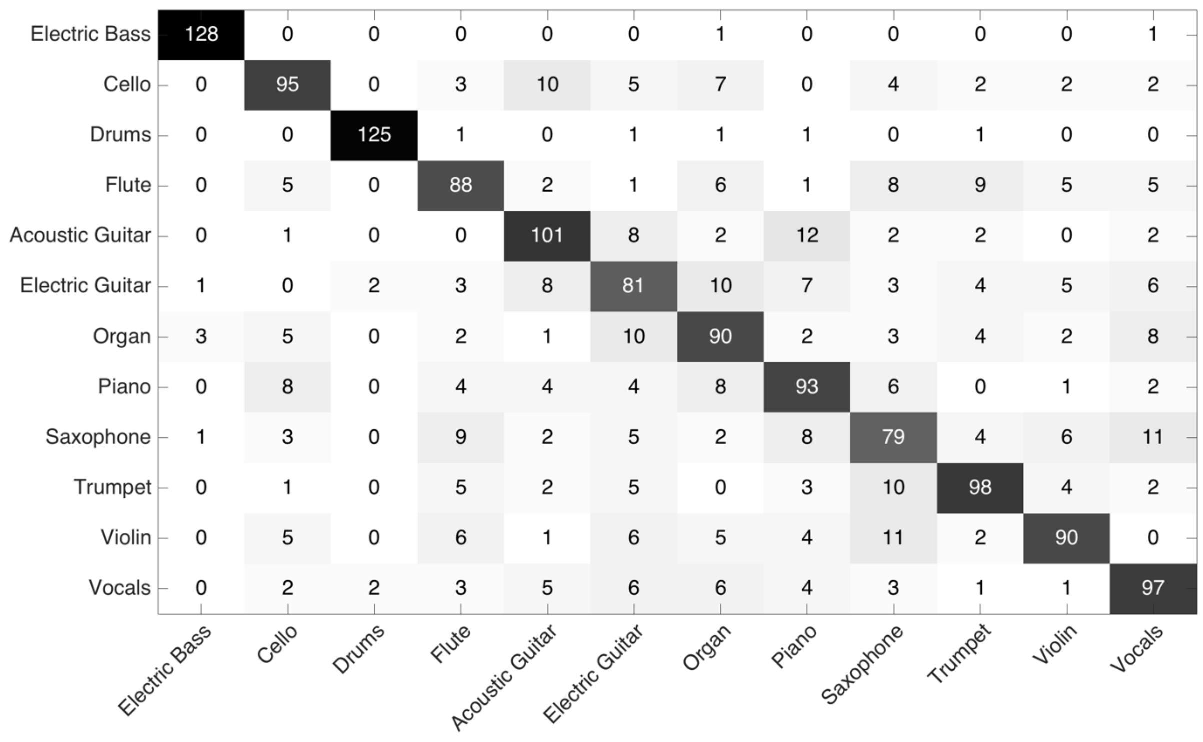
Where *TP* is True Positive, *TN* is True Negative and *FN* is False Negative.

Several AMIR research papers (see for example Eronen, 2000) assess their algorithms by computing the individual instrument performance and the instrument family performance. The latter checks how many instruments were classified to their correct instrument family (such as brass, strings etc.). As this wasn't one of the goals I set to myself when designing the system, this evaluation metric was not used.

5.1. Results

As mentioned, at the beginning of the project the dataset was divided into three categories: training, validation and testing. Due to the fact that the dataset is relatively small, the majority of it was used for training, and only a small portion for validation and testing. Concretely, approximately 70% of the examples were used for training, and 30% for validating and testing the network.

Using the validation set, the ConvNet achieves a Top-1 error rate of 25.6%. Consider the following confusion matrix:



	Electric Bass	Cello	Drums	Flute	Acoustic Guitar	Electric Guitar	Organ	Piano	Saxophone	Trumpet	Violin	Vocals
Electric Bass	128	0	0	0	0	0	1	0	0	0	0	1
Cello	0	95	0	3	10	5	7	0	4	2	2	2
Drums	0	0	125	1	0	1	1	1	0	1	0	0
Flute	0	5	0	88	2	1	6	1	8	9	5	5
Acoustic Guitar	0	1	0	0	101	8	2	12	2	2	0	2
Electric Guitar	1	0	2	3	8	81	10	7	3	4	5	6
Organ	3	5	0	2	1	10	90	2	3	4	2	8
Piano	0	8	0	4	4	4	8	93	6	0	1	2
Saxophone	1	3	0	9	2	5	2	8	79	4	6	11
Trumpet	0	1	0	5	2	5	0	3	10	98	4	2
Violin	0	5	0	6	1	6	5	4	11	2	90	0
Vocals	0	2	2	3	5	6	6	4	3	1	1	97

Figure 4.4: Confusion matrix of validation set. The set consisted of 130 audio clips for each instrument.

The confusion matrix shows us in a very straightforward manner if our classifier is behaving as expected. It also allows us to do some deeper analysis in order to be able to draw further conclusions about the classifier. For example, we can immediately see that electric bass and drums have the highest precision. One possible explanation is that they are the most distinctive instruments in the set. Drums are the only percussive instrument and electric bass has the lowest frequency range. We can also see that the instruments with the lowest precision are saxophone and electric guitar. This is perhaps a predictable behaviour, as these two instruments have similar characteristics (such as in timbre and range) to other instruments on the dataset. Indeed, the confusion matrix shows us that trumpet is often confused with saxophone, acoustic guitar with electric guitar and electric guitar with organ. These mistakes are likely to also occur with humans.

Vocals are confused with almost every instrument in the dataset, despite being very different from acoustic and electric instruments. One possible explanation is the complex structure of the human voice, which covers a large range of frequencies and various different timbres. Dividing the vocals category to sub-categories can help improving this.

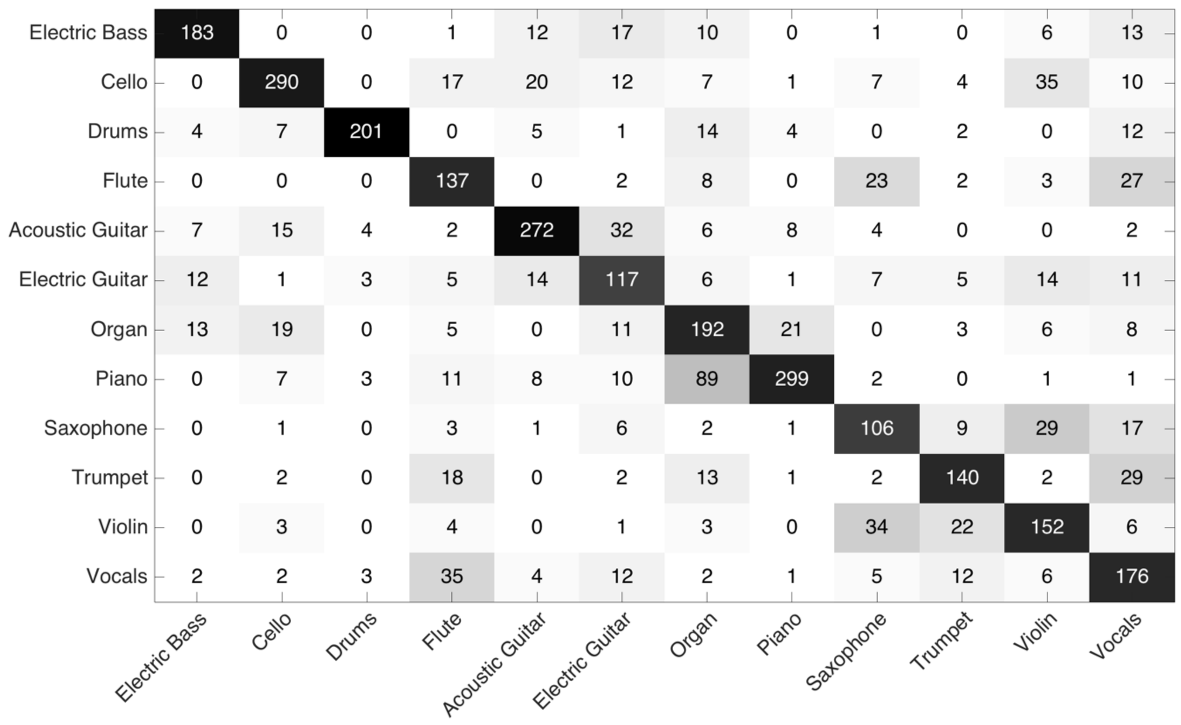
We can also conveniently compute the precision, recall and F-measure of each instrument class using the confusion matrix. To compute the precision, we sum each row, and to compute the recall we sum each column. The F-Measure is computed from the precision and recall. *Table 4.1* presents the results.

Instrument	**Precision**	**Recall**	**F-Measure**
Electric Bass	0.9846	0.9624	0.9734
Cello	0.7307	0.7600	0.7451
Drums	0.9615	0.9690	0.9652
Flute	0.6769	0.7097	0.6929
Acoustic Guitar	0.7769	0.7426	0.7594
Electric Guitar	0.6230	0.6136	0.6183
Organ	0.6923	0.6522	0.6717
Piano	0.7154	0.6889	0.7019
Saxophone	0.6076	0.6124	0.6100

Trumpet	0.7538	0.7717	0.7626
Violin	0.6923	0.7759	0.7317
Vocals	0.7462	0.7132	0.7293
Average	0.6893	0.6901	0.6893

Table 4.1: Precision, recall and F-measure of validation set

For testing, UPF's *Good Sounds* dataset was used, along with some randomly picked tracks of solo-instrument improvisations from the *Jungle Vibe* audio database. For the testing dataset, the system achieves a Top-1 error rate of 28.3%. The testing confusion matrix is described as following:



	Electric Bass	Cello	Drums	Flute	Acoustic Guitar	Electric Guitar	Organ	Piano	Saxophone	Trumpet	Violin	Vocals
Electric Bass	183	0	0	1	12	17	10	0	1	0	6	13
Cello	0	290	0	17	20	12	7	1	7	4	35	10
Drums	4	7	201	0	5	1	14	4	0	2	0	12
Flute	0	0	0	137	0	2	8	0	23	2	3	27
Acoustic Guitar	7	15	4	2	272	32	6	8	4	0	0	2
Electric Guitar	12	1	3	5	14	117	6	1	7	5	14	11
Organ	13	19	0	5	0	11	192	21	0	3	6	8
Piano	0	7	3	11	8	10	89	299	2	0	1	1
Saxophone	0	1	0	3	1	6	2	1	106	9	29	17
Trumpet	0	2	0	18	0	2	13	1	2	140	2	29
Violin	0	3	0	4	0	1	3	0	34	22	152	6
Vocals	2	2	3	35	4	12	2	1	5	12	6	176

Figure 4.5: Confusion matrix of test set

We can immediately observe that there are more misclassifications on the test set than on the validation set. Naturally, instruments that share a similar sound are the ones most likely to be confused. 299 out of 431 piano audio clips were classified correctly, while 89 were classified as organ. Another example would be acoustic and electric guitar, where 32 of 352 acoustic guitar samples were misclassified as electric guitar. Similar to the confusion matrix of the validation set, human voice is confused with all of the other categories quite often.
Although there are more misclassifications scattered around the confusion matrix, there still is dominant diagonal line representing the correct classifications.

The precision, recall and F-measure of each instrument from the test set are presented in the following table:

Instrument	**Precision**	**Recall**	**F-Measure**
Electric Bass	0.7531	0.8281	0.7888
Cello	0.7196	0.8357	0.7733
Drums	0.8040	0.9481	0.8701
Flute	0.6782	0.5756	0.6227
Acoustic Guitar	0.7727	0.8218	0.7965
Electric Guitar	0.5969	0.5247	0.5585
Organ	0.6906	0.5783	0.6295
Piano	0.6921	0.8192	0.7503
Saxophone	0.6057	0.5521	0.5777
Trumpet	0.6699	0.7254	0.6965
Violin	0.6756	0.5802	0.6243
Vocals	0.6769	0.5696	0.6186
Average	**0.6412**	**0.643**	**0.639**

Table 4.2: Precision, recall and F-measure of test set

The per-instrument precision obtained when examining the test set is generally lower than the validation set. However, apart from electric bass and drums, the decrease in precision is quite minor. This means that the network succeeded in generalizing and it has learned features that well represent the instrument. It seems that the high precision of drums and electric bass obtained on the validation set were the result of overfitting. Nevertheless, testing shows that both of these classes are still classified with high precision, recall and F-measure compared to the rest of the classes. It is important to note that both these classes did not exist in the original IRMAS dataset. Their data is mostly constructed from song stems and solo improvisations from very few sources. As can be seen, the lack of data considerably impacted the results of these two classes. The model has over-fitted to the training data: it classifies the training and validation data well, but it fails to generalize. Adding more data from different sources and recording qualities would improve these results.

5.2. Comparison to previous work

Lately, researchers began investigating ConvNets in the context of AMIR. Lostanlen and Cella (2016) published a report in which they apply different models of ConvNets and compare their accuracies. These models were trained solely on annotated data from MedleyDB. The top-one error rate obtained by the best model was 26%. Details regarding the F-measure and the precision and recall are not specified in the report. Although this result is slightly better than the one obtained in this project, it is important to note that the dataset they used consisted of only eight instruments. Less instruments makes it slightly easier on the classifier to discriminate between them, especially with limited training data.

Earlier research such as Kitahara, Goto and Okuno (2003) achieved slightly better results with a top-1 error rate of 20.27%. Their approach consists of extracting 129 features from recordings of solo instruments. Their dataset consisted of 19 pitched instruments and the recordings are of single tones. Using single tone recordings is a simplification of the problem of AMIR, and hence the high accuracy obtained by this algorithm. A similar study was carried out by Eronen and Klapuri (2000), achieving 19.4% top-1 error rate on a dataset of 30 orchestral instruments. Although this current project achieves lower accuracies than these research projects, it does not make any simplifications nor assumptions about the data, and hence might be more suitable for real-world applications.

6. Conclusion

This project investigates the potential of using ConvNets for AMIR of solo recordings. Using *TensorFlow*, a deep ConvNet was constructed for this purpose. We then investigated the effect of its different components and parameters, trying to optimize the model in order for it to achieve the best possible results. It has been demonstrated that ConvNets are highly capable of obtaining impressive and competitive results in the context of AMIR, even with limited amount of data. With more data, this technique is likely to improve upon the current state-of-the-art, as seen in other areas of MIR.

6.1. Future work

There is still a lot of research yet to be carried out in this area. One of the important aspects yet to be researched is the input type. In this project, as in many other MIR applications, spectrograms were used as the input data. Different input types may improve the accuracy. It seems that in other MIR applications, researchers often experiment with different inputs, such as recurrence plots (Park and Lee, 2015a) and even the raw time-domain signal. It is certainly an area worth investigating.

Furthermore, there are many parameters that influence the performance of the system. Specifically, this project didn't experiment with many different values and combinations of dropout. Varying the dropout level across different layers has proved successful in some contexts (Sigtia, Benetos and Dixon, 2016).

One technical issue that should be resolved in the near future is the fact that the system only supports fixed length audio clips. Currently, the input to the network must be three seconds long, or more precisely, 66,500 samples. In order to allow this algorithm to work under real-world scenarios, it would be necessary to remove this limitation and support audio clips of any length. It would be worth investigating what would be the best way of doing so.

In order to improve the performance of the system, I would like to investigate the possibility of training and testing the model on several GPUs in parallel. *TensorFlow* already added support to this feature. This would be beneficial for real-world applications, especially in real-time scenarios.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M. and Ghemawat, S., 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems.

Bandiera G. Romani Picas O., Serra X., 2016. Good-sounds.org: a framework to explore goodness in instrumental sounds. In: *17th International Society of Music Information Retrieval (ISMIR)*.

Basheer, I.A. and Hajmeer, M., 2000. Artificial neural networks: fundamentals, computing, design, and application. *Journal of microbiological methods*, *43*(1), pp.3-31.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D. and Bengio, Y., 2010. Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf* (pp. 1-7).

Bittner, R.M., Salamon, J., Tierney, M., Mauch, M., Cannam, C. and Bello, J.P., 2014, October. MedleyDB: A Multitrack Dataset for Annotation-Intensive MIR Research. In *ISMIR* (pp. 155-160).

Choi, K., Fazekas, G. and Sandler, M., 2016. Explaining Deep Convolutional Neural Networks on Music Classification.

Deep Learning, 2016. *Convolutional Neural Networks (Lenet)*. [online] Available at: <http://deeplearning.net/tutorial/lenet.html>

Eggink, J. and Brown, G.J., 2004. Instrument recognition in accompanied sonatas and concertos. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on* (Vol. 4, pp. iv-217). IEEE.

El Ayadi, M., Kamel, M.S. and Karray, F., 2011. Survey on speech emotion recognition: Features, classification schemes, and databases. *Pattern Recognition*, *44*(3), pp.572-587.

Eronen, A. and Klapuri, A., 2000. Musical instrument recognition using cepstral coefficients and temporal features. In *ICASSP'00* (Vol. 2, pp. II753-II756). IEEE.

Eronen, A., 2001. Comparison of features for musical instrument recognition. In *Applications of Signal Processing to Audio and Acoustics* (pp. 19-22). IEEE.

Essid, S., Richard, G. and David, B., 2004a. Musical instrument recognition based on class pairwise feature selection. In *ISMIR*.

Essid, S., Richard, G. and David, B., 2004b. Musical instrument recognition on solo performances. In *12th European Signal Processing Conference* (pp. 1289-1292). IEEE.

Fuhrmann, F., 2012. *Automatic musical instrument recognition from polyphonic music audio signals* (Doctoral dissertation, PhD thesis, Universitat Pompeu Fabra).

Glorot, X. and Bengio, Y., 2010. Understanding the difficulty of training deep feedforward neural networks. In *Aistats* (Vol. 9, pp. 249-256).

Golik, P., Doetsch, P. and Ney, H., 2013, August. Cross-entropy vs. squared error training: a theoretical and experimental comparison. In *INTERSPEECH* (pp. 1756-1760).

Goodfellow, I., Bengio, Y., Courville, A., 2016. *Deep Learning*. MIT Press.

Han, Y., Kim, J. and Lee, K., 2016. Deep convolutional neural networks for predominant instrument recognition in polyphonic music.

Haralick, R.M., Shanmugam, K. and Dinstein I., 1973. Textural features for image classification. *IEEE Transactions on systems, man, and cybernetics*, (6), pp.610-621.

Heittola, T., Klapuri, A. and Virtanen, T., 2009. Musical Instrument Recognition in Polyphonic Audio Using Source-Filter Model for Sound Separation. In *ISMIR* (pp. 327-332).

Ioffe, S. and Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift.

Jang, G., Kim, H.G. and Oh, Y.H., 2014. Audio Source Separation Using a Deep Autoencoder. *arXiv preprint arXiv:1412.7193*.

Kingma, D. and Ba, J., 2014. Adam: A method for stochastic optimization.

Kitahara, T., Goto, M. and Okuno, H.G., 2003. Musical instrument identification based on F0-dependent multivariate normal distribution. In: *Acoustics, Speech, and Signal Processing* (Vol. 5, pp. V-421).

Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).

Livshin, A. and Rodet, X., 2004. Musical instrument identification in continuous recordings. In *Digital Audio Effects 2004* (pp. 1-1).

Lostanlen, V. and Cella, C.E., 2016. Deep convolutional networks on the pitch spiral for music instrument recognition.

Marques, J. and Moreno, P.J., 1999. A study of musical instrument classification using gaussian mixture models and support vector machines. *Cambridge Research Laboratory Technical Report Series CRL*, *4*.

Masko, D. and Hensman, P., 2015. The impact of imbalanced training data for convolutional neural networks.

Mathieu, B., Essid, S., Fillon, T., Prado, J. and Richard, G., 2010. YAAFE, an Easy to Use and Efficient Audio Feature Extraction Software. In *ISMIR* (pp. 441-446).

McCulloch, W.S. and Pitts, W., 1943. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4), pp.115-133.

Meeker, M., 2014. 2014 Internet Trends. [online] Available at: <http://www.kpcb.com/blog/2014-internet-trends> [Accessed 24 July 2016].

Nielsen, M., 2016. Neural networks and deep learning. [online] Available at: < http://neuralnetworksanddeeplearning.com> [Accessed 29 July 2016].

Park, T. and Lee, T., 2015a. Musical instrument sound classification with deep convolutional neural network using feature fusion approach.

Park, T. and Lee, T., 2015b. Music-Noise segmentation in spectrotemporal domain using convolutional neural networks.

Rawlinson, H., Segal, N. and Fiala, J., 2015. Meyda: an audio feature extraction library for the Web Audio API. In Proceedings of the 2015 Web Audio Conference.

Sigtia, S., Benetos, E. and Dixon, S., 2016. An End-to-End Neural Network for Polyphonic Piano Music Transcription. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, *24*(5), pp.927-939.

Srivastava, N., Hinton, G.E., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting.Journal of Machine Learning Research, 15(1), pp.1929-1958.

Ruder, S., 2016. An overview of gradient descent optimization algorithms. [online] Available at: <http://sebastianruder.com/optimizing-gradient-descent/index.html> [Accessed 30 July 2016].

Wikimedia Commons, 2016. Gradient ascent (surface). [online] Available at: < https://commons.wikimedia.org/wiki/File:Gradient_ascent_(surface).png> [Accessed 16 August 2016]