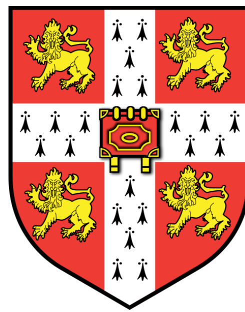
Andrew Wells

Deep Learning for Music Recommendation

Part II Computer Science Tripos



Queens' College

University of Cambridge

May 18, 2018

Proforma

Name:	**Andrew Wells (aw684)**
College:	**Queens' College**
Project Title:	**Deep Learning For Music Recommendation**
Examination:	**Computer Science Tripos – Part II, May 2018**
Word Count:	**11900[1] (within the 12000 limit)**
Project Originator:	Cătălina Cangea (ccc53)
Supervisor:	Cătălina Cangea (ccc53)

Aims of the Project

The aim of this project was to implement a Convolutional Neural Network music-tagging system[2] capable (with better-than-chance performance) of tagging songs with descriptive key words taken from four categories: genres (e.g. "rock", "pop"), moods (e.g. "happy", "chillout"), eras (e.g. "60s", "00s") and instruments (e.g. "female vocalists", "guitar"). The project further aimed to explore the use of such a system in facilitating content-based music recommendation, a technique used by music-streaming services such as Spotify[3].

Work Completed

The project was a huge success, exceeding both core criteria, confirming the relative model performance found in [8] and improving the training speed by more than three-fold from 2400 seconds per epoch to 727 ± 8.38 seconds per epoch. Furthermore, two extensions have been carried out to improve system performance (up to an AUC-ROC score of 0.78) and to explore the use of the system in content-based music recommendation.

Special Difficulties

None.

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z nn' | wc -w`
[2] Using architectures taken from current research [8].
[3] http://benanne.github.io/2014/08/05/spotify-cnns.html

Declaration

I, Andrew Wells of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date: May 18, 2018

Contents

1 Introduction **1**
1.1 Project motivation . . . 1
1.2 Background . . . 2
1.2.1 Supervised learning . . . 2
1.2.2 Automatic music tagging . . . 3
1.2.3 Convolutional neural networks . . . 4
1.3 Related work . . . 4

2 Preparation **5**
2.1 Starting point . . . 5
2.2 Requirements analysis . . . 6
2.3 Choice of tools . . . 6
2.3.1 Programming languages, libraries and licenses . . . 8
2.3.2 Development practices . . . 8
2.3.3 Development environment . . . 9
2.3.4 Backup . . . 9
2.4 Neural networks . . . 9
2.4.1 A single neuron . . . 9
2.4.2 Deep feedforward networks . . . 9
2.4.3 Activation functions . . . 10
2.5 Maximum Likelihood Estimation (MLE) . . . 12
2.6 Multi-label classification . . . 14
2.6.1 Bernoulli distribution . . . 14
2.6.2 Two-class classification . . . 14
2.6.3 Multi-label classification . . . 15
2.6.4 Binary cross entropy . . . 15
2.6.5 Data representations for music tagging . . . 16
2.7 Convolutional neural networks . . . 16
2.7.1 Convolution . . . 16
2.7.2 Convolutional layers . . . 17

3 Implementation **23**
3.1 Setting up the data pipeline . . . 23
3.1.1 The Million Song Dataset . . . 23
3.1.2 Data preprocessing . . . 25

3.1.3 Data pipeline . . . 28
3.2 Implementing the CNN models . . . 31
3.2.1 TensorFlow . . . 31
3.2.2 Convolutional unit . . . 33
3.2.3 Convolutional architectures . . . 36
3.2.4 Architectures implemented . . . 37
3.3 Defining loss functions . . . 43
3.4 Training and evaluating the models . . . 43
3.4.1 Stochastic gradient descent (SGD) . . . 44
3.4.2 The ADAM optimiser . . . 45
3.4.3 Evaluating the model . . . 46

4 Evaluation **49**
4.1 Testing . . . 49
4.2 Performance evaluation . . . 50
4.2.1 Accuracy . . . 50
4.2.2 Precision and recall . . . 55
4.2.3 AUC-ROC . . . 56
4.3 Using deep learning for music recommendation . . . 58
4.3.1 An analysis of predictions . . . 58
4.3.2 Exploring the latent representation of songs . . . 60

5 Conclusion **65**
5.1 Achievements . . . 65
5.2 Lessons learned . . . 65
5.3 Further work . . . 66

Bibliography **66**

A Project Proposal **69**

List of Figures

1.1 Reproduced from [19]. A t-SNE plot showing the latent space representations of artists, which can be used to find similar artists for music recommendation. . . . 3

2.1 All training and testing was performed on the Computational Biology Group's GPUs. . . . 7
2.2 A feedforward network. (Left:) The computation performed by a single neuron within a layer of the network. (Right:) A feedforward neural network with input, hidden and output layers. . . . 10
2.3 The *sigmoid* activation function (in blue) and its derivative (in red). . . . 11
2.4 The ReLU activation function (in blue) and its derivative (in red). . . . 11
2.5 An example showing 2D convolution without kernel flipping. The output is restricted to positions where the kernel lies entirely within the original image. . . . 17
2.6 A simple example of a convolutional layer with 3 filters. . . . 19
2.7 A simple example of a convolutional layer with 3 filters. . . . 19
2.8 An example of parameter sharing. . . . 20
2.9 Input x_3 only interacts with three outputs when the kernel size is 3. . . . 20
2.10 Output z_3 only interacts with three inputs when the kernel size is 3. . . . 20

3.1 A UML class diagram showing the architecture of the music tagging system. 24
3.2 A mel-spectrogram of a song sample from the dataset. . . . 26
3.3 A diagram showing the purpose of implementing an efficient data pipeline. Training time is significantly reduced by optimising the pre-fetch of input data. . . . 28
3.4 The input pipeline . . . 30
3.5 An example of a TensorFlow graph. . . . 32
3.6 A pictorial representation of a convolutional unit, implemented in the `ConvUnit` class (see Listing 3.5). . . . 33
3.7 The pooling operation over a 1D region of width 3, using a stride of 1. Invariance is achieved as follows: the input to the detector stage in the left-most diagram has been shifted to the right by a single unit in the rightmost diagram, which is reflected in the outputs of the pooling stage. Despite all of the input values from the detector stage changing, only half of the values after the pooling stage have been affected. . . . 34
3.8 Downsampling over regions of width 3, using a stride of 2. . . . 35

3.9 The ELU activation in blue plotted against the ReLU activation. 35
3.10 **Same** mode: the $[2, 2]$ output is padded with zeros to produce a $[3, 3]$ output. 38
3.11 **Valid** mode: when a $[2 \times 2]$ kernel is used with a stride of $(1, 1)$ a $[2, 2]$ output is produced. 39
3.12 An example of a 2D convolutional layer, with 3 $[2 \times 2]$ filters, a stride of $(1, 1)$ and no zero-padding. Neurons are represented by circles. The output volume is 3×3 and consists of three channels: each channel is the activation map corresponding to the response of each filter in the convolutional layer. Each neuron in the output volume has a 2×2 receptive field. 39
3.13 A UML diagram showing the class design for the DataCollector and its interaction with the script. 47

4.1 Unit test results. 50
4.2 The training loss for the *k2c1* model. 51
4.4 Average precision scores of the three architectures, each having 1M parameters. The blue, orange and green denote the precision averages for *k2c1*, *k1c2* and *k2c2* respectively. 56
4.5 AUCs for the top-50 tags obtained from each of the three architectures. The red, yellow and blue lines are the AUC averages for *k2c2*, *k2c1* and *k1c2* respectively. The green, purple, and blue points are the per-tag AUC scores for *k2c2*, *k1c2* and *k2c1* respectively. 57
4.6 AUCs averaged across the top-50 tags obtained from each of the three architectures. The orange, red, and green lines are the AUC averages for *k2c2*, *k1c2* and *k2c1* respectively. 57
4.7 AUCs for each of the top-50 tags obtained from the *k2c1* model with 1M parameters. 59
4.8 AUCs for each of the top-50 tags obtained from the *k1c2* model with 1M parameters. 59
4.9 AUCs for each of the top-50 tags obtained from the *k2c2* model with 1M parameters. 59
4.10 A t-SNE plot of 10,000 songs based on their predicted top-50 tags. 61
4.11 The t-SNE plot in Figure 4.10 with artist names. 62
4.12 The t-SNE plot from Figure 4.11 showing a close-up view of artists. 63

Acknowledgements

Writing the dissertation and implementing the CNN system has been a challenging yet extremely rewarding experience, but without the support and guidance of my supervisor, family, and friends, the journey would have been all the more difficult. I would therefore like to dedicate my work to the following individuals:

- My supervisor, **Cătălina Cangea**, for the extensive feedback and comments on the written dissertation, and also the invaluable support she has offered in the implementation process. I could not have wished for a more dedicated supervisor.

- My parents **Susan Wells** and **Graham Wells**, and my sister **Julia Wells**, for their unyielding support over not only the past year, but the last three years spent at Cambridge.

Chapter 1

Introduction

This project aimed to explore the use of Convolutional Neural Networks (CNNs) in automated music tagging. I have achieved this by successfully completing the core of my project—implementing a fully functional CNN music tagging model from a recent research paper [8]—and two additional extensions: implementing two more models from the same paper that outperformed the first one and investigating the potential of these architectures in a music recommendation scenario.

1.1 Project motivation

With an estimated 1,268.6m[1] users, streaming services such as Spotify, Pandora, and Apple Music have fundamentally changed the way we listen to music. Users can now discover millions of songs and artists they would otherwise never have discovered. Features like Discover Weekly, Artist Radio, AI-curated playlists and other recommendation-based products are constantly evolving through the underlying machine learning techniques used to develop them.

For many years, collaborative filtering has been the quintessential approach to music recommendation. Using this technique, new suggestions are primarily built upon **historical usage data**, which makes the whole process content agnostic. This makes collaborative filtering prone to the following[2]:

- **It does not work when usage data is sparse**: collaborative filtering does not work for new users—they have no listening preferences or usage data.
- **Recommendations are not personal**: unlike listening preferences, which are often unique and personal. Collaborative filtering suggests songs based on the usage data of individuals with similar listening preferences, which is prone to false-positive recommendations, particularly when users share little common ground, or some of them have more eclectic musical taste.

[1] https://www.statista.com/outlook/209/100/music-streaming/worldwide#market-revenue
[2] http://benanne.github.io/2014/08/05/spotify-cnns.html

- **Popular songs are more likely to be recommended**: sometimes a drawback, as popular songs have already gained exposure and the goal is to automate a means of music discovery, whilst maintaining the user's listening preferences.

- **The cold-start problem:** new songs and up-and-coming artists suffer, as they are not recommended until enough people have listened to them. The quality of recommendations are dependent on how many users the service has and, more specifically, how many of them have listened to each song.

An alternative approach avoids the scenarios described above by basing recommendations directly on the **musical content**. Songs can be grouped into clusters by humans and hand-tailored playlists curated, but this is time-consuming and not always possible: Spotify and Apple Music have a catalogue of around 20m songs, whilst Sony Music have 25m[3].

Automated music tagging offers a possible solution: train a system to automatically label songs with descriptive keywords, that can later be used to find similar songs with respect to their audio content. Figure 1.1 gives a visualization of similar artists based on musical tags associated with their songs.

My project focusses on implementing CNN-based systems to automatically label songs with informative tags. As an extension, these labels have been used to build latent-space representations of the songs and an analysis of this latent space has been carried out to explore its use in content-based music recommendation. These labels have subsequently been used to build a 2D representation of the songs (mainly achieved through the t-SNE dimensionality reduction algorithm [20]). This space can be visualised in order to analyse the potential of these tagging systems for content-based recommendation.

1.2 Background

1.2.1 Supervised learning

Supervised learning is a form of machine learning which aims to build a model that predicts output variables given input variables, with the input or observations $\mathbf{X} = (\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_m}) \in \mathbb{R}^n$ and target values or output variables existing in $\mathbf{Y} = (\mathbf{y_1}, \mathbf{y_2}, ..., \mathbf{y_k})$. Supervised learning attempts to represent a hypothesis $h : \mathbb{R}^n \to Y$ which models the true relationship between input and output. This hypothesis can be *learned* by a neural network, for example, which translates to finding an optimal set of parameters $\boldsymbol{\theta}$ (in this case, weights and biases)[4]. When Y is finite, we are looking at a classification task. Multi-label classification aims to find $h : \mathbb{R}^n \to Y^k$; relating data to a *vector of labels*.

[3] https://ti.me/1drNspu

[4] http://www.cl.cam.ac.uk/teaching/1718/MLBayInfer/ml-bayes-18.pdf

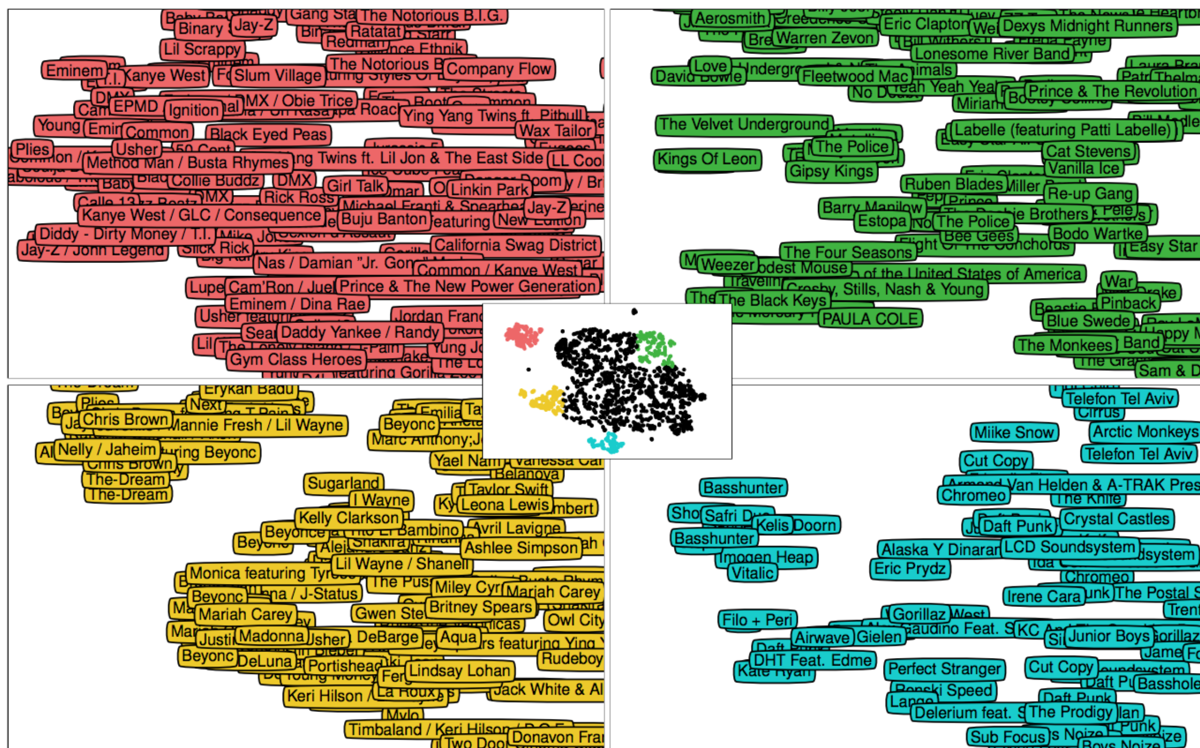


Figure 1.1: Reproduced from [19]. A t-SNE plot showing the latent space representations of artists, which can be used to find similar artists for music recommendation.

1.2.2 Automatic music tagging

Automatic tagging represents a form of *multi-label classification* (since a song can have more than one label to describe it) and is becoming increasingly relevant for producing better and more interesting music recommendations. Considerable research has already been carried out in this field, but many approaches consist of a feature extraction phase that precedes the learning and prediction steps: namely, producing a numerical representation that characterizes the audio content [18]. The problem with manual feature extraction is that it requires more computational effort and domain-specific knowledge, in order to devise suitable features that represent the raw data[5].

Designing descriptive features for raw audio files is one of the main challenges in automatic music tagging [7]. Extracting the features manually also gives rise to the question of *which* features to extract for a given application area. Having the neural network learn these features by itself, directly from audio data, requires no domain-specific knowledge and allows the network's final representation to encompass the most relevant features with respect to the tagging process.

[5] https://arxiv.org/pdf/1606.00298.pdf

1.2.3 Convolutional neural networks

Convolutional Neural Networks (CNNs) have many interesting applications, most notably image classification [2, 13], speech recognition [16] and even epileptic seizure detection [22]. More recently, they have been applied to automatic music tagging [10] in an attempt to build end-to-end systems which extract features from raw audio files.

CNNs have been shown to usefully exploit the inherent 2D structure of image data in machine learning tasks. Since an audio signal can be represented in 2D, CNNs have also been used in music tagging, genre classification, instrument tagging and content-based music recommendation [10] due to their ability to learn features relevant to the task. Labelling of moods and eras requires increasingly complex, high-level feature extraction—CNNs can learn by themselves which features are most relevant and more complex representations can be produced as the network depth is increased.

1.3 Related work

Automated music tagging is an area of active research (§1.1). Choi et al. [7] found the mel-spectrogram to be an effective 2D representation for music tagging using CNNs—I have therefore used this input representation throughout my project. The architectures I have implemented (§3.2.3) are inspired by a more detailed follow-up paper [8]—the evaluation in Chapter 4 includes a comparison between the results obtained from the different network architectures I implemented and against the results presented by the authors.

Chapter 2

Preparation

This chapter outlines the software engineering practices I have adopted throughout the project, providing an analysis of the required modules. It then proceeds to present the essential theoretical aspects needed to implement the music tagging system.

2.1 Starting point

Prior to undertaking work on the project, I had:

- theoretical knowledge of the basic concepts from the *Artificial Intelligence I* course,
- a basic knowledge of Python.

During the project and especially during Michaelmas term, before starting to implement any components of the automatic tagging system, I had to teach myself new concepts and explore new frameworks; many of these are outside the CST Part II syllabus. I gained:

- theoretical knowledge of supervised learning, classification and evaluation methods from the *Machine Learning and Bayesian Inference* course ahead of Lent term, when the course is taught,
- theoretical knowledge of Convolutional Neural Networks, optimization algorithms and t-SNE,
- theoretical knowledge of of audio processing techniques needed for input representation (in the form of spectrograms) from the *Digital Signal Processing* course,
- an understanding of the `TensorFlow` API,
- familiarity with the `Python` language and commonly used libraries (`Numpy`, `Pickle`),
- knowledge of the `Librosa`, an audio-processing library used to generate the spectrograms.

2.2 Requirements analysis

Before beginning the implementation, building a full specification of the deliverables required for a successful project was of great importance. As mentioned in the *Project Proposal*, the core aims of the project were to have:

- A working implementation of a Convolutional Neural Network.
- The ability of the network to perform better-than-chance prediction (accuracy above 50% for each tag and the lower bound of its confidence interval above 50%) of the top-50 tags.

These correspond to the first item (labelled with high priority and difficulty) in Table 2.1.

Table 2.1: Requirements analysis for the project.

Requirement	Priority	Difficulty
CNN *k2c1* (5 ConvLayers + 2 FC)	high	high
CNN *k1c2* (4 ConvLayers + 2 FC)	medium	medium
FCNN *k2c2* (5 ConvLayers)	medium	medium
t-SNE Visualisation Module	medium	low
Model Restoring Capabilities	medium	high
Training Suite	high	medium
Data Pipeline Module	high	medium
Data Pipeline optimisations	medium	high
Data Collection Suite	high	low
Data Analysis Suite	high	low

Deliverables of lower priority formed part of the two extension goals that were also achieved within the project. The use of a number of libraries—especially `TensorFlow`—required extensive research into the API and learning about how to implement neural networks at a high level. Because of this, the first model to be implemented (*k2c1*) was perceived as the most challenging; as this represented a core criterion, it was of the highest priority. As my project required working with big data (the training set alone has a size of over 200GB), optimising the data pipeline was a significant challenge.

2.3 Choice of tools

Throughout the project, it has been necessary to learn new technologies and tools (see Table 2.2). It was not possible to store the entire 600GB dataset on my laptop, so I had to request more disk space on the CL machines, where I trained and tested my models (see Figure 2.1). This meant the I had to gain familiarity with UNIX, `emacs` and `ssh`.

Tool	Purpose
`Atom`	Writing the core code base
`git`	Version control
`ssh`	Remotely connecting to the GPU machines for running code
`Emacs`	Remotely writing code and data analysis
Unix	The OS installed on the CL machines

Table 2.2: Tools used in the project

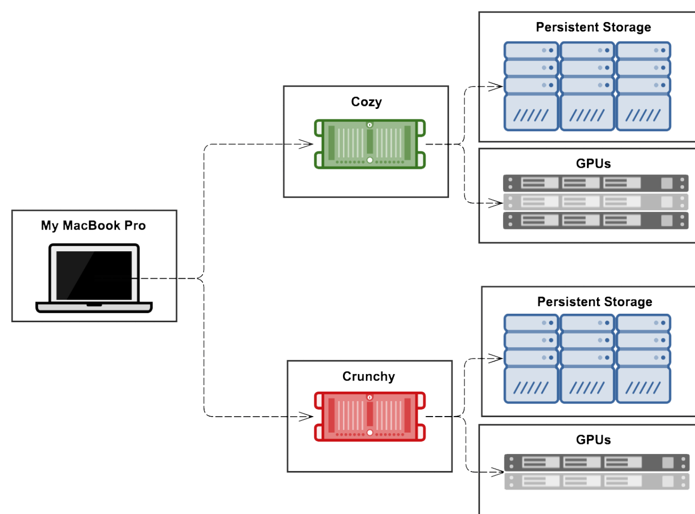


Figure 2.1: All training and testing was performed on the Computational Biology Group's GPUs.

Table 2.3: Libraries used in the project

Library	Version	Purpose	License
TensorFlow-gpu	1.4.0	Implementing and training the CNNs	Apache 2.0
Matplotlib	2.1.1	Plotting evaluation results	BSD license
Numpy	1.14.2	Data analysis	BSD license
Scikit learn	0.19.1	Data analysis and t-SNE training	BSD license

2.3.1 Programming languages, libraries and licenses

The project was implemented entirely in `Python`, using the `TensorFlow` Python API for implementing the CNNs and training functionality. I chose `Python` because it is the only language that fully supports the `TensorFlow` API, while also providing extensive online documentation.

Table 2.3 enumerates the libraries I have used in the project. The `TensorFlow-gpu` [3] library was of key importance in implementing the CNN models. This library is frequently used in machine learning projects, due to its extensive availability of neural network modules and ability to train and test models on GPUs, without additional manual configuration.

Under both the BSD[1] and Apache 2.0[2] licenses, I am free to use the libraries for any purpose: to distribute them, to modify them, and to distribute modified versions of the libraries under the terms specified by the licenses. The `tikz` diagrams included in my dissertation were inspired by the work of Petar Veličković[3] and as such are viable to the terms of the MIT[4] license. My dissertation repository will also be released under the MIT license to enable the free distribution and modification of my code.

2.3.2 Development practices

I used an Agile development methodology, with fixed constraints in terms of *what* was to be implemented, but with flexibility in *how* those components were to be implemented. This consisted of setting two week timeframes for sprints where I would implement a specific component or carry out research in a particular area, with a review period at the end of those sprints to track progress. This turned out to be particularly effective, especially when improving the data pipelining implementation.

[1] https://opensource.org/licenses/BSD-3-Clause
[2] https://www.apache.org/licenses/LICENSE-2.0
[3] https://github.com/PetarV-/TikZ
[4] https://opensource.org/licenses/MIT

2.3.3 Development environment

The majority of the development process was carried out on my personal laptop, using the `Atom` text editor with `git` integration for version control. The development cycle consisted of implementing a change locally, committing and pushing to `git`, and then pulling new changes to the server. While remotely connected to the GPU machines through `ssh`, I edited the code using `Emacs`.

2.3.4 Backup

The project implementation and write-up were both stored locally, backed up periodically onto an external hard disk, and also synced with iCloud Drive. They were also given separate `git` repositories.

2.4 Neural networks

My project implements a convolutional neural network, which first requires understanding of a neural network. In this section, I describe the components of a neural network: neurons and activations.

2.4.1 A single neuron

Recall from §1.2.1 that our model estimates a hypothesis function $h(\theta; \mathbf{x})$. The simplest case of hypothesis function, also known as *linear discriminant* or *perceptron*, applies an *activation function* σ to a function that is linear in its parameters $\mathbf{w} = (w_0, w_1, ..., w_{D-1})$:

$$h(\mathbf{w}; \mathbf{x}) = \sigma\left(w_0 + \sum_{j=1}^{D-1} w_j \phi_j(\mathbf{x})\right) \tag{2.1}$$

2.4.2 Deep feedforward networks

Feedforward networks (an example is shown in Figure 2.2) are built up of layers of individual neurons: each of them takes the dot product between their inputs and weights, adds a bias vector and applies an activation function to provide non-linearity to the model. The output of one layer is fed to the input of the next layer—hence the term "feedforward", and the number of layers represent the network depth.

A neural network represents the aforementioned hypothesis function $f = h(\theta, \mathbf{x})$, where θ is the set of optimal parameters that are learned using gradient descent—an optimisation algorithm that I describe in the Implementation chapter. This method updates the model parameters according to a **loss function**, which describes the error between the model

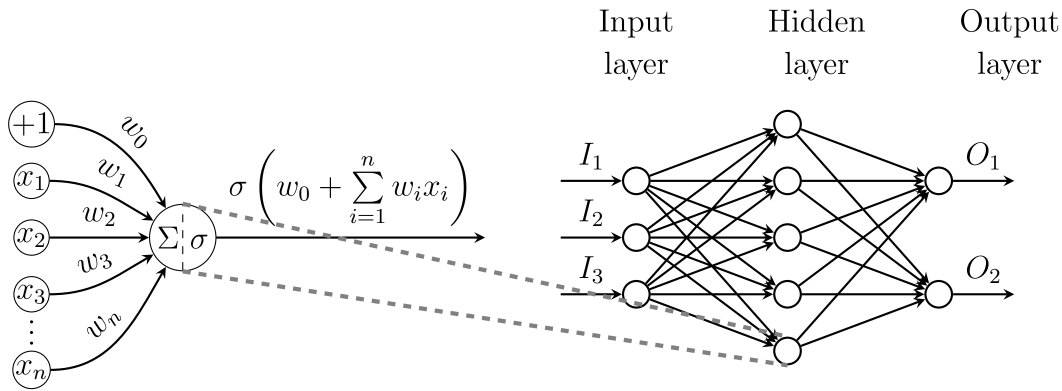


Figure 2.2: A feedforward network. (Left:) The computation performed by a single neuron within a layer of the network. (Right:) A feedforward neural network with input, hidden and output layers.

output and the target output. To obtain the required loss function to train our model, maximum likelihood estimation can be used (§2.5).

2.4.3 Activation functions

Activation functions add non-linearity to our model, allowing it to learn more complex representations, and can also be used to limit the range of the output. This is particularly useful in classification, where we aim to output *probabilities* of our input belonging to a specific class: my system outputs, for each tag, the probability of that tag being associated with the input song. The output of a layer can be written as:

$$\mathbf{z} = \mathbf{W}^T\mathbf{x} + \mathbf{b}, \tag{2.2}$$

where $\mathbf{x}$ is the input to the neuron, $\mathbf{W}$ represents the layer's weights (each row corresponds to a single neuron) and $\mathbf{b}$ is the vector of bias terms for each of the neurons.

To introduce non-linearity, an activation function can be applied to $\mathbf{z}$. The activation functions used in my project are the sigmoid (§2.4.3) and ELU (§3.2.2) activation functions.

Sigmoid

The *sigmoid* activation function is used for probabilistic classification which will be described in Section 2.6.3. It takes the form:

$$\sigma(z) = \frac{1}{1 + exp(-z)} \tag{2.3}$$

and its derivative can be written as:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)) \tag{2.4}$$

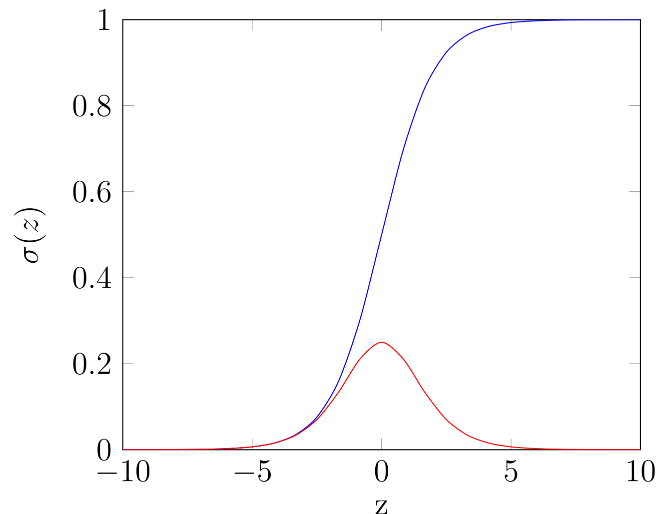


Figure 2.3: The *sigmoid* activation function (in blue) and its derivative (in red).

ReLU

The ReLU activation function in Equation 2.5 is commonly used due to the stability in its gradient. ReLU diminishes the *vanishing gradient* problem that arises with other activations, since its gradient is 1 for non-negative values. The *vanishing gradient* problem arises due to the way in which gradients are computed in back-propagation: the chain-rule involves multiplying intermediate gradients, and for small values these gradients vanish [5]. ReLU, however, introduces a bias: it has a mean activation greater than zero. I used the ELU activation to overcome this issue (§3.2.2), which is discussed in the Implementation chapter.

$$ReLU(z) = max(0, z) \tag{2.5}$$

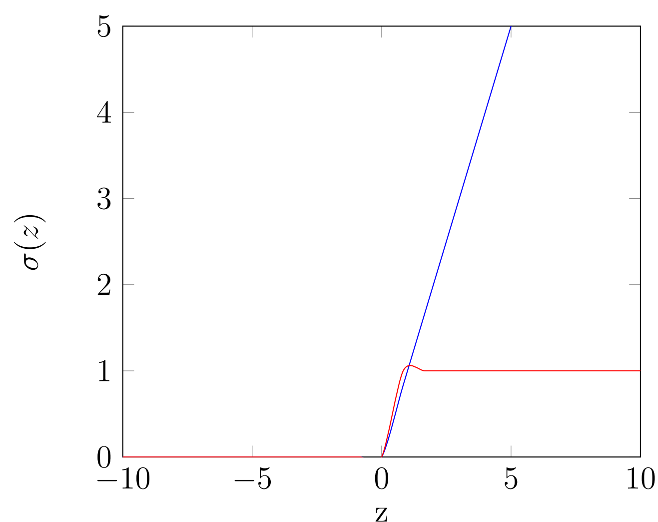


Figure 2.4: The ReLU activation function (in blue) and its derivative (in red).

[5] http://neuralnetworksanddeeplearning.com/chap5.html#discussion_why

2.5 Maximum Likelihood Estimation (MLE)

As mentioned in §2.4.2, we want our model to represent a hypothesis function that *estimates* the relationship between our input and output data. Maximum Likelihood Estimation (MLE) [11] can be used to derive a suitable loss function, which we employ when adjusting our model parameters to reach an optimal representation of the hypothesis function. The MLE principle enables us to find the loss functions required to train our model. Take a set

$$\mathbb{X} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(m)}\}$$

of m independent, identically distributed (IID) examples, taken from an unknown distribution p_{data}. We aim to train a model $p_{model}(\mathbf{x}; \theta)$ of this distribution by updating the parameters in θ. Conceptually, $p_{model}(\mathbf{x}; \theta)$ represents the parametric family of probability distributions over the same space as $p_{data}(\mathbf{x})$, indexed by the choice of parameters θ. The model aims to map our input vectors to a real number:

$$p_{model}(\mathbf{x}; \theta) : \mathbf{x} \to \mathbb{R}$$

providing an estimate of the true probability $p_{data}(\mathbf{x})$. When building a model using neural networks, our parameters are the respective weights and biases of the network that are updated during training.

MLE principle enables us to find the optimal parameters[6] [12]:

$$\begin{aligned} \theta_{ML} &= \arg\max_{\theta} \left[p_{model}(\mathbb{X}; \theta) \right] \\ &= \arg\max_{\theta} \left[\prod_{i=1}^{m} p_{model}(\mathbf{x}^{(i)}; \theta) \right] \end{aligned} \tag{2.6}$$

where the second line follows from the IID assumption about the training examples in $\mathbb{X}$. Unfortunately, in training a model based on Equation 2.6, the computation is prone to underflow due to the constituents of the product being small. An equivalent optimisation can be achieved by taking the log of the expression in Equation 2.6, which allows us to transform the product into a sum:

$$\theta_{ML} = \arg\max_{\theta} \left[\sum_{i=1}^{m} \log p_{model}(\mathbf{x}^{(i)}; \theta) \right] \tag{2.7}$$

Dividing Equation 2.7 by the number of samples m does not affect the quantity we have to maximise for finding an optimal set of parameters θ, so we obtain the following equivalent expression:

$$\theta_{ML} = \arg\max_{\theta} \left[\mathbb{E}_{\mathbf{x} \sim p_{train}} \left(\log p_{model}(\mathbf{x}; \theta) \right) \right] \tag{2.8}$$

[6] Optimal in the sense that they are chosen from the parametric family of probability distributions ($p_{model}(\mathbf{x}; \theta)$) indexed by the choice of parameters θ, such that the probability of generating the input $\mathbf{x}$ with these parameters is maximal.

which corresponds to maximising the expectation of the log-likelihood given that $\mathbf{x}$ follows the distribution p_{train}—the *empirical distribution* defined by the training set with which the model is trained.

The maximum likelihood estimator above can be thought of as reducing the dissimilarity between two probability distributions. Since the exact distribution is unknown (we only have access to a limited amount of data sampled from it), we take our empirical distribution represented by the training set (p_{train}) and attempt to reduce the dissimilarity between it and p_{model}. A useful measure for the dissimilarity between two distributions is the *Kullback Leibler divergence*, which gives a measure of the dissimilarity between the two distributions.

Given some unknown distribution $p(\mathbf{x})$ that is modelled by an estimator $q(\mathbf{x})$, the *Kullback-Leibler divergence* (KL) or *relative entropy* between the two distributions is defined as:

$$D_{KL}(p||q) = \Big(-\int p(\mathbf{x})\log q(\mathbf{x})d\mathbf{x}\Big) - \Big(-\int p(\mathbf{x})\log p(\mathbf{x})d\mathbf{x}\Big) \tag{2.9}$$

If $KL(p||q) = 0$, then the two distributions are equal. The two terms are the entropy when approximating the distribution p with q and the actual entropy of the unknown distribution p, respectively.

The application of KL to the maximum likelihood estimator problem is useful because it tells us that the estimator is trying to minimize the distance between the model distribution and the empirical distribution. Applying Equation 2.9 to the MLE and taking the data to be discrete, we arrive at:

$$\begin{aligned} D_{KL}(p_{train}||p_{model}) &= -\sum_{i=1}^{m} p_{train}(\mathbf{x})\log p_{model}(\mathbf{x}) - \left(\sum_{i=1}^{m} p_{train}(\mathbf{x})\log p_{train}(\mathbf{x})\right) \\ &= \sum_{i=1}^{m} p_{train}(\mathbf{x})\Big(\log p_{train}(\mathbf{x}) - \log p_{model}(\mathbf{x})\Big) \\ &= \mathbb{E}_{\mathbf{x}\sim p_{train}}\Big(\log p_{train}(\mathbf{x}) - \log p_{model}(\mathbf{x})\Big) \end{aligned} \tag{2.10}$$

Since the empirical distribution p_{train} is defined by the training set and is not a function of the model itself, training the model requires us to minimize the following quantity by updating our model parameters θ, which is an equivalent expression to Equation 2.8:

$$\begin{aligned} \theta_{ML} &= -\arg\min_{\theta}\Big[\mathbb{E}_{\mathbf{x}\sim p_{train}}\Big(\log p_{model}(\mathbf{x};\theta)\Big)\Big] \\ &= \arg\max_{\theta}\Big[\mathbb{E}_{\mathbf{x}\sim p_{train}}\Big(\log p_{model}(\mathbf{x};\theta)\Big)\Big] \end{aligned} \tag{2.11}$$

Training a model by minimizing KL divergence or equivalently maximising the log-likelihood both yield the same optimal parameters θ, but our loss functions for each scenario differ. A loss function (as mentioned in Section 2.4.2) is a term frequently coined to refer to the function used in the optimization of our model, which we aim to minimize during training—we "minimize the cost function".

When maximising the log-likelihood, we therefore minimise the negative log likelihood. In doing so, it is helpful to relate to minimizing the cross-entropy as this has a minimum at zero, as described above in Equation 2.9, whereas the negative log likelihood can become negative for real-valued $\mathbf{x}$. I will discuss loss functions in more detail in Chapter 3.

2.6 Multi-label classification

The probabilistic approach to classification is based on *probabilistic generative models* and allows the derivation of a multi-label classification model. Taking the probability of a tag being associated with a song (§2.6.1) allows us to derive a model for two-class classification (§2.6.2), which can then be applied to the multi-label classification problem at hand (§2.6.3).

2.6.1 Bernoulli distribution

We can use the Bernoulli distribution [6] to model the probability of a label being assigned to an example. This distribution is over a single *binary* random variable X, where any instance of X takes one of two values: $x \in \{0, 1\}$. The Bernoulli distribution is defined by a single parameter ϕ, for which the following properties hold:

$$\begin{aligned} \mathbb{P}(X = 1) &= \phi \\ \mathbb{P}(X = 0) &= 1 - \phi \\ \mathbb{P}(X = x) &= \phi^x (1 - \phi)^{1-x} \end{aligned} \tag{2.12}$$

2.6.2 Two-class classification

Given an input vector $\mathbf{x}$, we wish to classify $\mathbf{x}$ into one of two (mutually-exclusive) classes C_1 and C_2. Taking the probabilistic approach [6], this is equivalent to modelling the posterior probability $\mathbb{P}(C_i|\mathbf{x})$. This is a **posterior** probability since it is *generated* when training the model, using our **prior** probabilities $\mathbb{P}(\mathbf{x}|C_i)$ and $\mathbb{P}(C_i)$. The posterior probability of C_1 given x then becomes:

$$\begin{aligned} \mathbb{P}(C_1|\mathbf{x}) &= \frac{\mathbb{P}(C_1, \mathbf{x})}{\mathbb{P}(\mathbf{x})} \\ &= \frac{\mathbb{P}(\mathbf{x}|C_1)\mathbb{P}(C_1)}{\mathbb{P}(\mathbf{x})} \\ &= \frac{\mathbb{P}(\mathbf{x}|C_1)\mathbb{P}(C_1)}{\mathbb{P}(\mathbf{x}|C_1)\mathbb{P}(C_1) + \mathbb{P}(\mathbf{x}|C_2)\mathbb{P}(C_2)} \end{aligned} \tag{2.13}$$

Neural networks can be trained to model this probability distribution with the help of the **sigmoid activation function** (§2.4.3). The sigmoid activation $\sigma : (-\infty, +\infty) \to [0, 1]$ has the property of mapping the real numbers to a finite interval. By applying it to the

output unit of a neural network, Equation 2.13 (the posterior distribution of the model) becomes the new output:

$$\begin{aligned}\mathbb{P}(C_1|\mathbf{x}) &= \frac{1}{1+\exp\left[\ln\frac{\mathbb{P}(\mathbf{x}|C_2)\mathbb{P}(\mathbf{C_2})}{\mathbb{P}(\mathbf{x}|C_1)\mathbb{P}(\mathbf{C_1})}\right]} \\ &= \frac{1}{1+\exp(-z)} \\ &= \sigma(z)\end{aligned} \tag{2.14}$$

where z is defined to be:

$$z = \ln\frac{\mathbb{P}(\mathbf{x}|C_1)\mathbb{P}(\mathbf{x})}{\mathbb{P}(\mathbf{x}|C_2)\mathbb{P}(\mathbf{x})} \tag{2.15}$$

Through gradient-based training methods, the network learns how to generate the posterior distribution. The learning algorithm specifies the **loss function** which is minimized during training to update the model parameters. In the case of two-class classification, we use the **binary cross-entropy** loss function (§2.6.4).

2.6.3 Multi-label classification

Multi-label classification takes some input vector $\mathbf{x} \in \mathbb{R}^n$ and predicts a K-*dimensional* vector $\mathbf{l} \in \mathbb{Z}^K$ of the label(s) associated with it. When training our network, we fix the set of possible labels to some size K. In the context of music tagging, the input vector is the song and the predicted vector contains labels associated with the song.

Multi-label classification is a natural extension of binary classification, which is, in turn, an instance of two-class classification. Given K labels we use K output units, each acting as a binary classifier, that predict the probability of an example belonging to class C_k. Each output unit uses the sigmoid activation function to turn the output into a valid Bernoulli probability distribution.

2.6.4 Binary cross entropy

We can generalise the MLE (§2.5) to predict output labels $\mathbf{y}$ for input vectors $\mathbf{x}$. This corresponds to maximising $\mathbb{P}(\mathbf{Y}|\mathbf{X};\theta)$. From Equation 2.6 in Section 2.5, we seek optimal parameters satisfying:

$$\theta_{ML} = \arg\max_{\theta}\left[\sum_{i=1}^{m}\log\mathbb{P}(y^{(i)}|x^{(i)};\theta)\right] \tag{2.16}$$

Over all input-output pairs, this is equivalent to:

$$\theta_{ML} = \arg\max_{\theta}\left[\log\mathbb{P}(\mathbf{Y}|\mathbf{X};\theta)\right] \tag{2.17}$$

and so the loss function for maximum likelihood learning of a Bernoulli random variable is given by:

$$\mathbf{J}(\theta) = -\log \mathbb{P}(\mathbf{Y}|\mathbf{X};\theta) \tag{2.18}$$

which corresponds to maximising $\log \mathbb{P}(\mathbf{Y}|\mathbf{X};\theta)$. To implement this in the context of a neural network, we define our output unit to use the sigmoid activation function (§2.6.2) giving:

$$\mathbf{J}(\theta) = -\log\left(\frac{1}{1+\exp(-z)}\right) \tag{2.19}$$

2.6.5 Data representations for music tagging

Let us define the set of labels $L = \{l_1, ..., l_K\}$ where a given l_i is a descriptive label of a song (for example, "rock" or "happy"). Automated music tagging is a form of *multi-label classification*: tags are not mutually exclusive and multiple tags may be used to describe a song. *Genre classification*—in which the set L is exclusively genres—is commonly a multi-class problem and seeks to assign a single genre to a song, giving 1 of K possible solutions. However, in music tagging, the set L can be populated by any combination of appropriate keywords. The key difference between genre classification and multi-label classification is that there may be 2^K different outputs in music tagging, as opposed to only K possible class assignments in genre classification. Consequently, music tagging tasks require much larger datasets, such that a suitable proportion of tag combinations can be represented and learned.

2.7 Convolutional neural networks

2.7.1 Convolution

Convolution is a mathematical operation taking as input two functions $f : \mathbb{R} \to \mathbb{C}$ and $g : \mathbb{R} \to \mathbb{C}$ and producing a third function denoted $f * g$ that is a combination of f and g. We define the **convolution** [11] of f and g as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau \tag{2.20}$$

In the context of convolutional networks, we define the first input f as our **convolutional layer input** and the second, g, as the **kernel**. The resulting output is often referred to as a **feature map**. Below we define convolution between a 2D image **I** and a 2D kernel **K**:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.21}$$

From the commutativity of convolution, we can equivalently write Equation 2.21 as:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.22}$$

In most machine learning libraries, convolution is implemented as defined in Equation 2.22, since there is reduced variation in values of m and n for images, whereas kernels fairly frequently vary in dimensionality. The reason the commutative property applies is due to the flipping of the kernel relative to the input (we first flip the kernel in both horizontal and vertical directions; this is because during convolution, as m increases, the image index increases, but the kernel one decreases).

Figure 2.5 depicts how convolution works intuitively: the operation is the integral over point-wise multiplications of the two functions, where one of the functions is shifted with respect the the other.

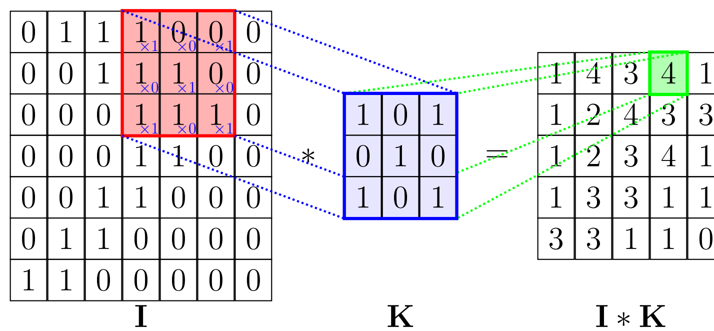


Figure 2.5: An example showing 2D convolution without kernel flipping. The output is restricted to positions where the kernel lies entirely within the original image.

A commonly used variant of convolution is **cross-correlation**, which is convolution without flipping the kernel relative to the image:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.23}$$

For the purposes of convolutional neural networks, it is common practice to treat cross correlation and convolution synonymously, by implementing the operation in Equation 2.23, visually represented in Figure 2.5, except in certain cases that will be explicitly stated otherwise.

2.7.2 Convolutional layers

Convolutional networks typically consist of convolutional layers, max-pooling layers and batch-normalisation units. Max-pooling and batch normalisation will be described in Chapter 3.

Convolutional networks are built upon three key ideas: **sparse interactions**, **parameter sharing** and **equivariant representations** [11]. In a convolutional neural network, the interactions between output and input units differ from traditional neural networks: rather

than being densely connected, the network exhibits sparse interactions between its layers. The convolutional layers consist of a set of learnable parameters that define a set of filters in that layer.

I will now define several terms that represent essential concepts for convolutional layers:

- **Kernel size**: corresponds to the *dimensions* of a given filter and dictates the dimensionality of a neuron's receptive field in the layer. For example, a convolutional layer may have filters with dimensions $[5 \times 5]$.
- **Filter size**: denotes the *number* of filters in the convolutional layer. For example, a convolutional layer might have 64 $[3 \times 3]$ filters, each of which produce a feature map in the output volume.
- **Feature map**: shows the response a filter has over a single channel of the input volume.
- **Input volume**: the input to the convolutional layer, of dimensions $W_{in} \times H_{in} \times C_{in}$.
- **Output volume**: of dimensions $W_{out} \times H_{out} \times C_{out}$, where C_{out} is determined by the number of filters in the layer; for K filters, the layer will output K activation maps, each of dimensions $W_{out} \times H_{out}$.

Receptive field

Consider the convolutional layer with three $[2 \times 2]$ filters illustrated in Figure 2.6. Let the yellow circle represent the output of a neuron in the convolutional layer. This neuron has a **receptive field** of 4—it "sees" four input values in each channel of the input volume. When each of the filters are convolved with the input volume, the dot product between each filter parameter and the input values is computed and the results are summed to give the output value of the neuron. In this case, the output volume is made up of 3 channels, each representing a **feature map** containing the response of a filter to the input. The green highlighted filter in Figure 2.6 is responsible for producing the green highlighted activation map in the output volume.

Parameter sharing

Parameter sharing refers to the weight parameters in Figure 2.7 being shared by each output neuron. Each filter produces a separate channel in the output volume. The green highlighted filter is responsible for producing the green highlighted channel in the output volume—this is the activation map of the filter. The red highlighted circles represent inputs that use the same filter parameter (highlighted in red), as the filter is moved across the input. Figure 2.8 further illustrates the sharing of parameters between neurons, where the red circles denote inputs that are multiplied by the **same** weights (corresponding to the top left cell of the kernel) when the filter is moved across the input.

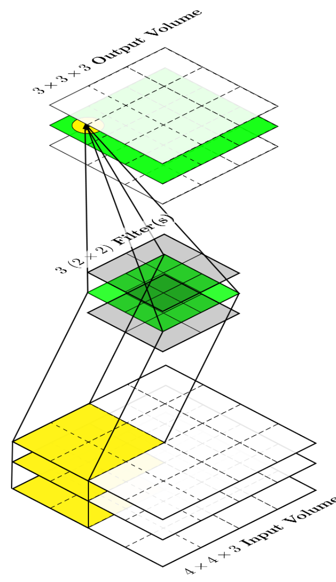


Figure 2.6: A simple example of a convolutional layer with 3 filters.

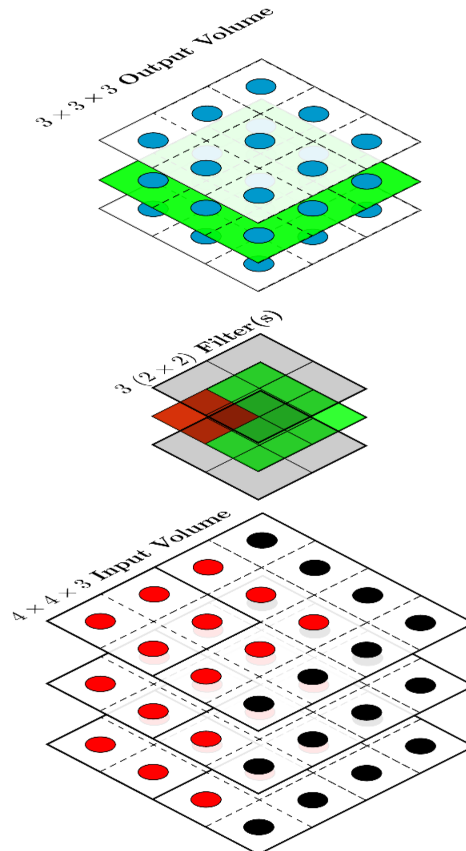


Figure 2.7: A simple example of a convolutional layer with 3 filters.

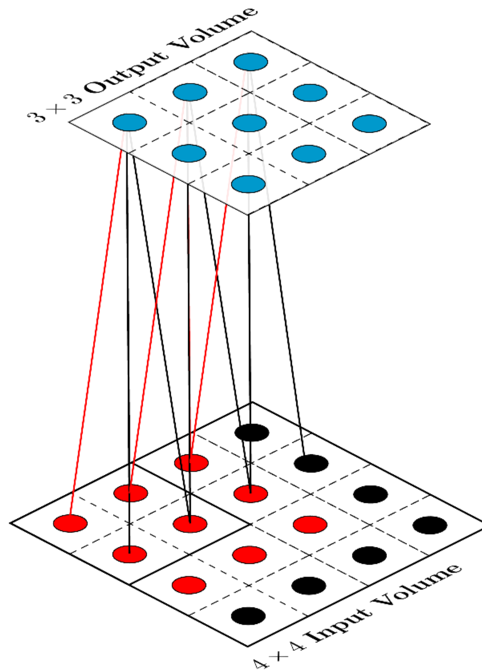


Figure 2.8: An example of parameter sharing.

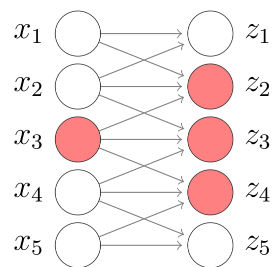


Figure 2.9: Input x_3 only interacts with three outputs when the kernel size is 3.

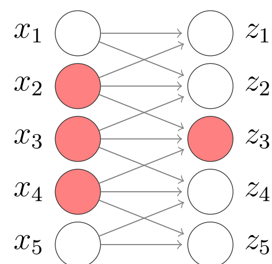


Figure 2.10: Output z_3 only interacts with three inputs when the kernel size is 3.

When the input volume is given to the convolutional layer, each filter is convolved over the input to produce a **feature map**. These feature maps are then stacked to produce the output volume, resulting in the number of filters F in a convolutional layer determining the number of channels in the output volume.

Fully-connected neural network layers use matrix multiplication to multiply a set of parameters—namely weights and biases—with the input to that layer. Each parameter has a single purpose: control the interaction between a single output unit and a single input unit, which results in a high number of parameters.

Sparse interactions

Convolutional layers tend to have **sparse interactions** (visualised in Figure 2.9 & Figure 2.10), meaning not every output unit interacts with every input unit. This is beneficial, because the number of required parameters is vastly reduced, particularly if the kernel size is much smaller than the input dimensions (for example, a $[3 \times 3]$ kernel convolved over a 28×28 pixel image).

Chapter 3

Implementation

In this chapter, I will detail the implementation of the music tagging system, (shown as a UML diagram in Figure 3.1) carried out towards the successful completion of my project core and extensions. This process consists of four stages:

1. Setting up the data pipeline: this stage posed the most problems, as I had to design an optimised multi-threaded pipeline that would speed up the loading of the 200GB training set in batches from disk, to reduce the infeasible initial epoch[1] time (3300s).
2. Implementing the three CNN architectures from the paper [8]: *k2c1*, *k2c2*, *k1c2*.
3. Choosing suitable loss functions for the music tagging task.
4. Training and evaluating the models: the size of the dataset affected this stage as well in terms of training time, so I had to ensure that model checkpointing and restoration could be easily performed. Evaluation results are presented in Chapter 4.

3.1 Setting up the data pipeline

3.1.1 The Million Song Dataset

The Million Song Dataset [5] was used for the project, which is described as being a "freely-available collection of audio features and metadata for a million contemporary popular music tracks"[2]. The tags the CNNs predict are the *last.fm*[3] tags, and are the top-50 such tags from the dataset. The CNNs were implemented and trained using the tags in Table 3.1.

[1] An epoch is a training iteration over the entire dataset.

[2] `https://labrosa.ee.columbia.edu/millionsong/`

[3] `https://labrosa.ee.columbia.edu/millionsong/lastfm`

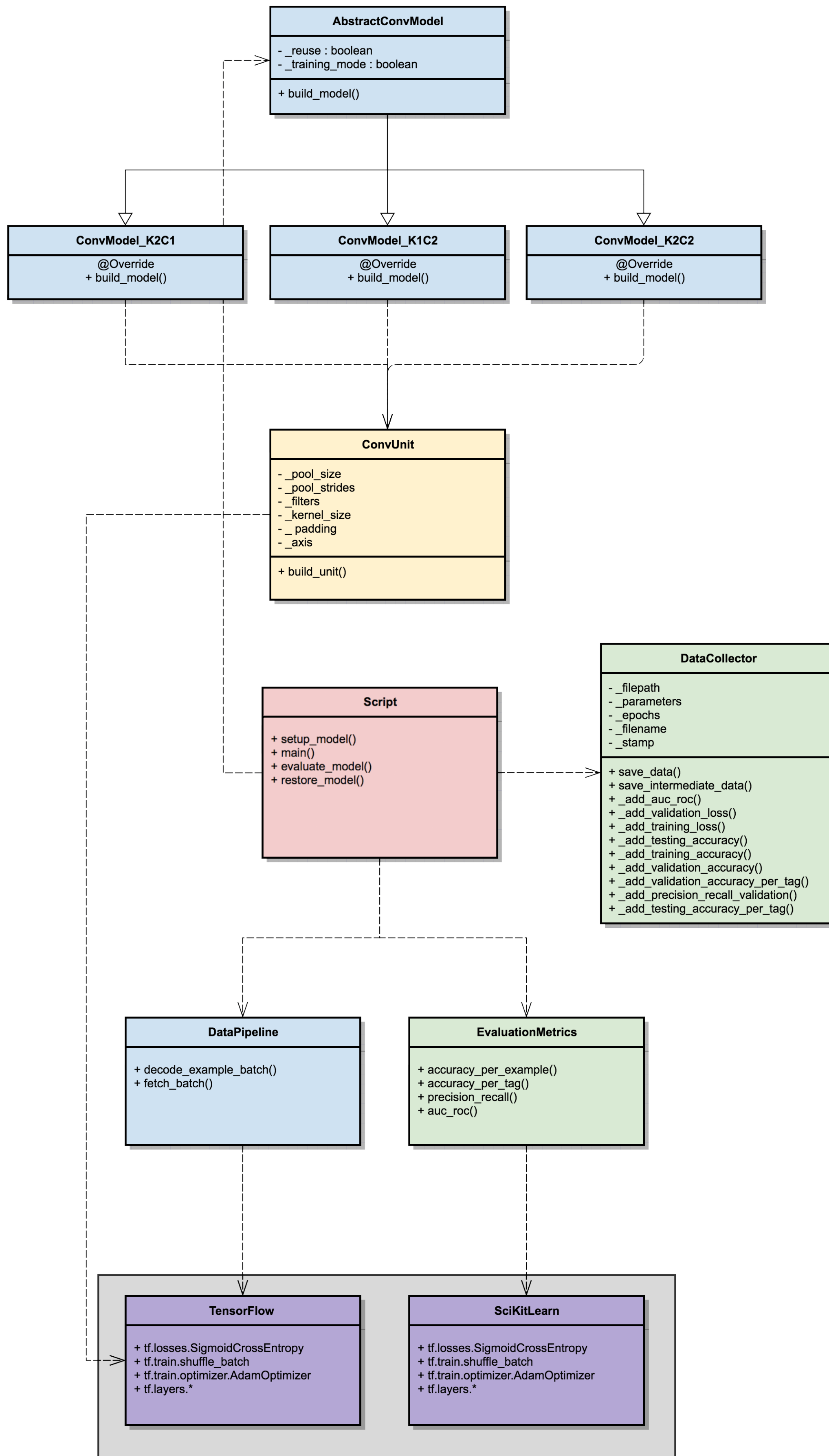


Figure 3.1: A UML class diagram showing the architecture of the music tagging system.

Table 3.1: The 50 last.fm tags the CNNs have been trained on.

Genres	Moods	Instruments	Eras
rock (1)	beautiful (11)	female vocalists (6)	00s (8)
pop (2)	chillout (13)	male vocalists (14)	80s (20)
alternative (3)	mellow (18)	instrumental (24)	90s (22)
indie (4)	chill (23)	female vocalist (32)	70s (35)
electronic (5)	oldies (26)	guitar (33)	60s (45)
dance (7)	ambient (29)		
alternative rock (9)	party (36)		
jazz (10)	easy listening (38)		
metal (12)	sexy (39)		
classic rock (15)	catchy (40)		
soul (16)	sad (48)		
indie rock (17)	happy (50)		
electronica (19)			
folk (21)			
punk (25)			
blues (27)			
hard rock (28)			
acoustic (30)			
experimental (31)			
hip-hop (34)			
country (37)			
funk (41)			
electro (42)			
heavy metal (43)			
progressive rock (44)			
R&B (46)			
indie pop (47)			
house (49)			

3.1.2 Data preprocessing

Obtaining and cleaning up the dataset

One of the first challenges encountered during the preparation phase was acquiring the dataset and ensuring that it only contained valid samples. The Million Song Dataset doesn't actually consist of raw audio files—instead, it maps song IDs to a set of attributes. The dataset itself consists of 201,680 songs for training, 12,605 songs for validation and 28,540 song entries for testing. At the time of writing the project proposal, the website

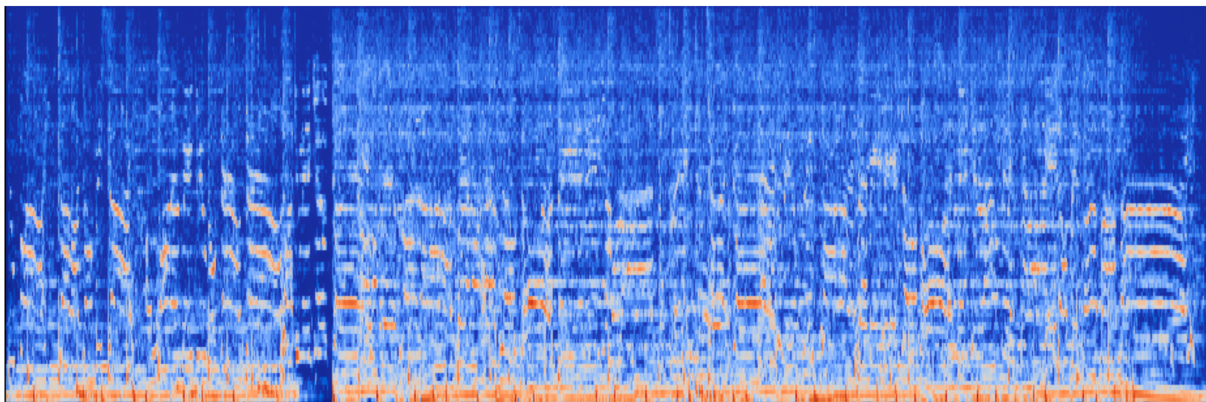


Figure 3.2: A mel-spectrogram of a song sample from the dataset.

7Digital[4] offered an API for accessing previews of the songs in the dataset using the IDs provided by the MSD—however, when I attempted to download the dataset during the preparation phase, the website no longer offered API keys to new users. I am therefore very grateful to Dr Keunwoo Choi, the author of the paper [8], for kindly syncing the 600GB raw audio dataset to the Computer Laboratory's servers. Dr Keunwoo Choi kindly provides a Github repository with files containing the train/validation/test song IDs and another file that stores the tags associated with each song.

Several song files were corrupted (zeroed files or files that are not encoded correctly), which meant the corresponding song IDs from the train/validation/test files had to be removed, along with the corresponding arrays of tags. After cleaning the dataset, the training, validation and test splits were left with 39, 5 and 6 fewer songs, respectively. Given the size of the dataset, these songs constituted a relatively small number of omissions, which I did not expect to have drastic consequences on the final results.

Mel-frequency spectrograms

A consideration that had to be made during the project was the whether to use the raw audio wave as input or to seek an alternative representation of the songs. Convolutional neural networks are primarily built for processing data with a grid-like topology and work particularly well with 2D images [11], which makes spectrograms—a 2D frequency-time representation of the audio—a good choice. Spectrograms were also used by Choi et al. [8], so a fair comparison between their results and mine could only have been achieved by using this input representation.

In particular, the log-amplitude mel-frequency spectrogram was chosen, since research in music classification and genre labelling [7] has shown that using log-amplitude mel-frequency spectrograms yields better results than those obtained from other representations (for example, short time Fourier transform, mel-frequency cepstral coefficients (MFCCs), linear-amplitude mel-spectrograms).

Developed as a *subjective* pitch scale, the mel-scale [17] was introduced to investigate how

[4]https://www.7digital.com/

Algorithm 1 Spectrogram generation

```
 1: procedure GenerateMelFrequencySpectrogram(filepath)
 2:     for IDs in batch do
 3:         audio, sampleRate ← load(filepath)
 4:         downsampleRate ← 12000
 5:         duration ← 29.12                    ▷ Take a 29.12s snippet from the audio file
 6:         audioDS ← resample(audio, sampleRate, downsampleRate)
 7:         sampleLength ← audioDS.getLength()
 8:         length ← duration * downSampleRate
 9:         if sampleLength < length then                        ▷ add zero padding
10:             audioDS ← audioDS.addPadding(length − sampleLength)
11:         else                                                 ▷ take the middle section
12:             upper ← (sampleLength − length)/2
13:             lower ← (sampleLength + length)/2
14:             audioDS ← audioDS[lower : upper]
15:         end if
16:         params ← {n_fft : 512 nmels : 96 hopLength : 256 sr : downsampleRate}
17:         spectrogram ← melspectrogram(audioDS, params)
18:         powerSpectrogram ← square(spectrogram)
19:         logSpectrogram ← logamplitude(powerSpectrogram, refPower = 1.0)
20:         return logSpectrogram
21:     end for
22: end procedure
```

humans judge changes in pitch. When presenting an audio representation that is as close as possible to the auditory experience of humans, we are allowing CNNs to learn some of the features that humans recognise when listening to music. The scale [21] is defined as follows:

$$\mathrm{f_{mel}} = 2595\log_{10}\big(1 + \frac{f}{700}\big) \tag{3.1}$$

where f is the initial frequency and f_{mel} is the resulting frequency on the mel scale.

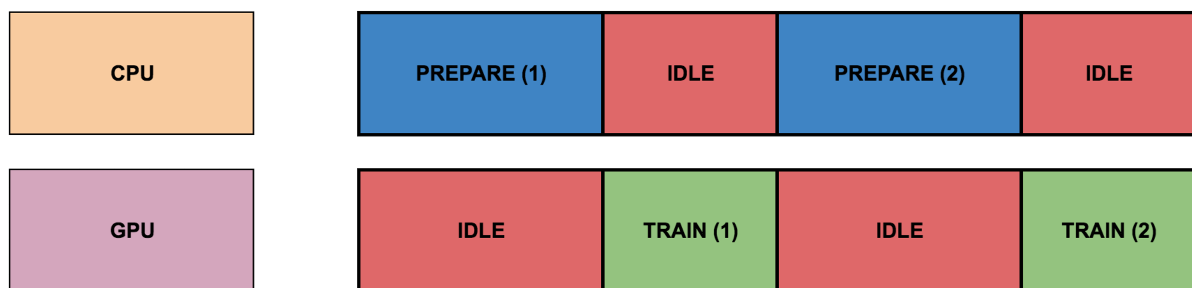
I used the `librosa` library to extract the spectrograms from the raw audio files—the dataset was processed in batches to reduce computational time. Algorithm 1 illustrates the data pre-processing steps taken. For compatibility with the architectures implemented in [8], the audio was re-sampled and a specific-length snippet taken to ensure the correct dimensionality for input. The CNNs have a 96×1366 input (mel-frequency $\times$ time-frame), meaning the sample rate and duration were chosen to satisfy: $1366 = (\text{duration} \times \text{sample rate})/\text{hop length}$. The hop-length used was 256, with the duration taken to be 29.12s (as in the paper), leaving the required sample rate as 12kHz. Zero padding was also used to ensure the spectrogram matched the input dimensions.

3.1.3 Data pipeline

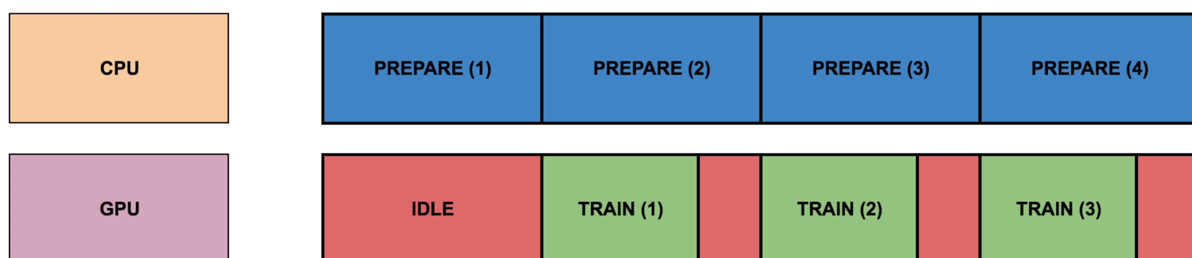
Given the size of the dataset, I spent a large proportion of the implementation phase optimising the data storage and retrieval processes. The mel-spectrograms were stored as `numpy` arrays—each of size 1.05MB. For the training set alone, this required approximately 211GB of storage, the entire dataset reaching 255GB in size. The GPU machine that was used throughout the project shares 62GB of RAM across 3 GPU kernels. This made loading the entire dataset into memory (the usual approach for many supervised learning tasks) infeasible, which required me to seek an alternative solution (see Figure 3.3).

Pre-fetching the data

Initially, a number of methods were implemented for loading the dataset from disk. This required maintaining pointers to an array of IDs for training and evaluation purposes. On initialisation, a block of data was loaded from disk into RAM: this involved taking a `block_size` number of IDs, loading each spectrogram and storing it in main memory. When the next batch was required, a method `get_chunk()` would simply take `batch_size` number of elements from the current block.



(a) A non-pipelined approach: both the CPU and GPU are idle for most of the time.



(b) A pipelined approach: idle time is considerably reduced. Whether the GPU still remains idle for a short time depends on the training time: for more complex models, training took longer than the time to fetch the next epoch's worth of data.

Figure 3.3: A diagram showing the purpose of implementing an efficient data pipeline. Training time is significantly reduced by optimising the pre-fetch of input data.

I quickly realised that this approach had a considerable drawback: the fetching of the next block is a synchronous, blocking operation. Spectrogram arrays are loaded *sequentially*,

Algorithm 2 Setting up the data pipeline for training (analogous for testing)

1: $example_training = fetch_batch(train_set_filenames, TRAINING_BATCH_SIZE)$
2: $params \leftarrow \{batch_size, queue_capacity, min_dequeue, num_threads\}$
3: $example_batch_training \leftarrow$ **tf.train.shuffle_batch**$([example_training], params)$

which is highly inefficient and fails to exploit exploit the potential parallelism. Training and data fetching can occur simultaneously, by training a model on the GPU and fetching the required data using the CPU. I therefore decided to implement a pipelined multi-threaded approach using TensorFlow's queue data structures, which vastly reduced idle time and maximised the GPU throughput. This resulted in a significant acceleration (4.5× faster[5]) per *epoch*—a single iteration over the entire training set.

Switching to a more flexible and optimised version of the data pipeline brought additional delays to the overall project timeline. I had to familiarise myself with TensorFlow `QueueRunners`, `RandomShuffleQueues` and `InputProducers`, in order to incorporate them in the implementation and use the API to build a flexible, optimised input pipeline. The pipeline consists of multiple producer threads, and a single consumer: multiple threads *enqueueing* examples onto the examples buffer—otherwise known as "producing"—and the training thread dequeues mini-batches using the `dequeue_many()` operation, otherwise known as "consuming".

The approach I adopted initialises the data pipeline as shown in Algorithm 2 and uses TensorFlow's `shuffle_batch()` method, which utilises a `RandomShuffleQueue`, and ensures the dataset is sufficiently shuffled for training.

The `fetch_batch()` function call is added as an operation to the TensorFlow graph, and `example_training` is the output tensor of the operation. When `example_training` is passed to the method `tf.train.shuffle_batch()`[6], TensorFlow adds a `Queue` (managed by the `QueueRunner`) to the graph that holds data from `example_training`.

Rather than requiring the developer to handle multi-threading, TensorFlow provides `QueueRunners` and a thread `Coordinator`, the former starts the threads for all the queue runners in the current graph, and the latter deals with clean-up of threads.

Figure 3.4 shows a schematic of the input pipeline containing the following components:

- **Filename queue:** holds the filename(s) of the dataset: for the training set, this queue holds the names of the files, each being a shard of the training set. The `string_input_producer` owns an `enqueue_many()` operation in the graph that builds and randomly shuffles a list of filenames, which is subsequently added to the filename queue.

- **Example queue:** added to the graph by the `shuffle_batch()` method. The example queue holds the batches of examples, which are dequeued and decoded when the training operation is executed.

[5]Comparison based on training the *k2c1* model using the sequential and multi-threaded approaches.
[6]https://www.tensorflow.org/api_docs/python/tf/train/shuffle_batch

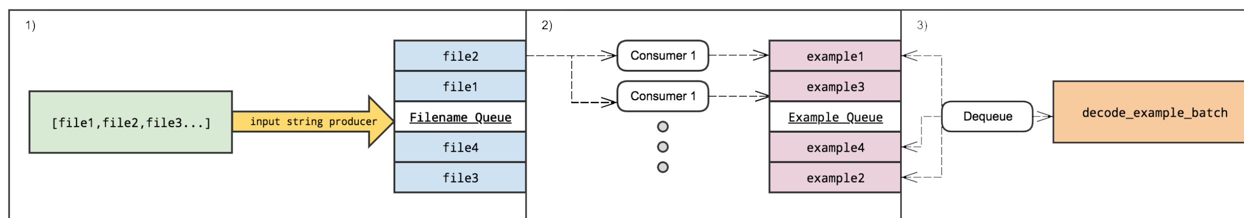


Figure 3.4: The input pipeline

When the training operation is called, `batch_size` examples are randomly sampled from the examples queue, which holds at least `min_to_dequeue` items. The batch is decoded and then fed forward through the CNN model. The pipeline adds two additional operations to the graph: `fetch_batch()` (shown in Listing 3.1) and `decode_example_batch()` (shown in Listing 3.2).

Listing 3.1: Code for fetching a batch of examples.

```
def fetch_batch(filename,batch_size,epochs):
    # build queue of filenames
    queue = tf.train.string_input_producer(filename,num_epochs=epochs,
        shuffle=True)
    # setup record reader
    reader = tf.TFRecordReader()
    batch = []
    # fetch the next batch
    for i in range(batch_size):
        _, example_serialised = reader.read(queue)
        batch.append(example_serialised)

    return [batch]
```

Listing 3.2: Code for decoding an example batch.

```
def decode_example_batch(batch_to_decode,batch_size):
    # use TF's parse_example to decode a batch of examples
    features = tf.parse_example(batch_to_decode, features={
        'song_id': tf.FixedLenFeature([], tf.int64),
        'data': tf.FixedLenFeature([131136], tf.float32),
        'target': tf.FixedLenFeature([50], tf.int64)
    })

    song_id = tf.cast(features['song_id'], tf.int32)
    data = tf.reshape(features['data'],[batch_size,96,1366,1])
    target = tf.reshape(features['target'],[batch_size,50])

    return data, target, song_id
```

TensorFlow records

Another important consideration regarding the overall feasibility of the training and evaluation process, both restricted by the project timeline, was how the spectrograms were to be stored. I aimed to improve the data pipeline efficiency by finding a way to minimise disk access latency during fetching of new batches, at the same time creating a much simpler interface for this operation.

The solution was to use TensorFlow Records: a binary file format made up of a sequence of strings. The records do not allow for random access, only sequential access, which required the sharding of the dataset to allow for shuffling. The `fetch_batch()` method in Listing 3.1 shows the `TFRecordReader()` method used to read the record file, and the `decode_example_batch()` method in Listing 3.2 shows the code used to decode an example.

Shuffling the dataset

During training, the dataset requires random shuffling to prevent the network from learning features from the *order* of the data, which would lead to it overfitting on the training set and not being able to generalise to unseen examples. Shuffling of the input data has also been shown to increase convergence speed [4] and the random sampling improves gradient estimates.

To maximise shuffling, the dataset was sharded into 13 files, each containing 16,384 examples. The `string_input_producer` in Listing 3.1 randomly shuffles the shard file names at the beginning of each epoch and adds the filenames to the filename queue as in (Stage 1 shown in Figure 3.4). These filenames are then dequeued sequentially by the consumer threads (Stage 2 shown in Figure 3.4), which then add batches of example to the example queue. The training thread then dequeues `training_batch_size` examples selected at random from the example queue, which results in a sufficiently random shuffling of the entire dataset.

3.2 Implementing the CNN models

3.2.1 TensorFlow

TensorFlow [3] is a framework that allows us to define machine learning algorithms using already implemented smaller units (e.g. network layers, activation functions). To describe a computation in TensorFlow, we must first build a graph: a set of nodes linked by directed edges. Here, nodes represent an instantiation of some operation—for example, applying an activation function, performing a matrix multiplication or convolving a kernel with the input. *Nodes* can take zero or more inputs and produce zero or more outputs and *tensors* are values that flow along edges in the graph—*tensors* are arbitrary dimensionality arrays defined during graph construction.

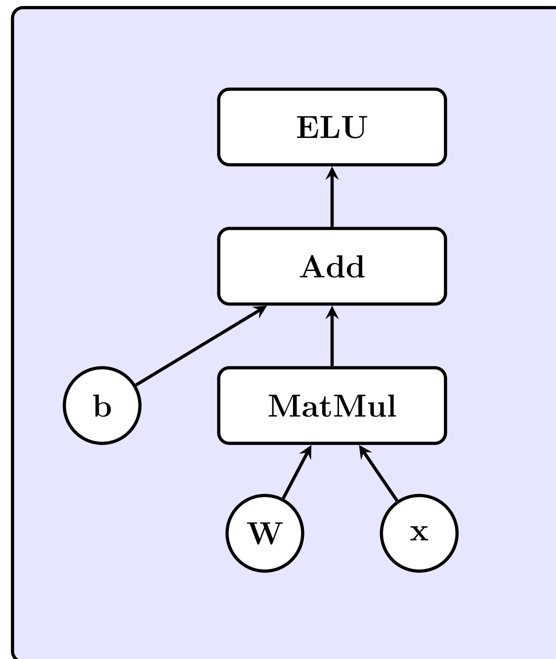


Figure 3.5: An example of a TensorFlow graph.

Setting up the graph

TensorFlow requires the initialisation of the tensors and variables being used in the computation. Variables are stored in the graph whilst the session remains active, holding the weights and biases for the created network.

The diagram in Figure 3.5 and code fragment in Listing 3.3 give an example of setting up a simple graph: the nodes represent variables and placeholders and rectangles denote `operations` that take tensors as arguments. When training the network, TensorFlow's `session` interface introduces a `run` operation which takes a set of outputs that need to be computed, and computes the order in which to evaluate each of them. Inputs can also be fed into the `run` operation to replace values in the graph: x is a placeholder that can be updated with the next input batch, for example.

Listing 3.3: An example of a TensorFlow code fragment corresponding to the TensorFlow graph in Figure 3.5.

```
# 100-d vector, init to zeroes
b = tf.Variable(tf.zeros([100]))
# 784x100 matrix of rand vals
W = tf.Variable(tf.random_uniform([784,100],-1,1))
# Placeholder for input
x = tf.placeholder(name=``x'')
# ELU(Wx+b)
elu = tf.nn.elu(tf.matmul(W,x)+b)
# get output of elu
sess.run([elu],feed_dict={x:input})
```

3.2.2 Convolutional unit

The CNNs implemented all follow a typical network with the layout (convolutional, activation, pooling layers), to which regularisation has been added in the form of batch normalisation, which I explain later in this subsection. Consequently, a convolutional unit consists of four stages:

1. **Convolution stage:** this is where the convolution is applied. The layer's filters are convolved with the input to produce an intermediary output volume which is fed into the second stage.
2. **Batch normalisation:** outputs from the convolution are fed into batch normalisation units.
3. **Detector stage:** non-linearities are applied at this point. The ELU activation function (discussed later on) was used in the CNN implementations.
4. **Pooling stage:** provides a summary statistic of several adjacent outputs and is used for down-sampling, at the same time making the network invariant to shifts.

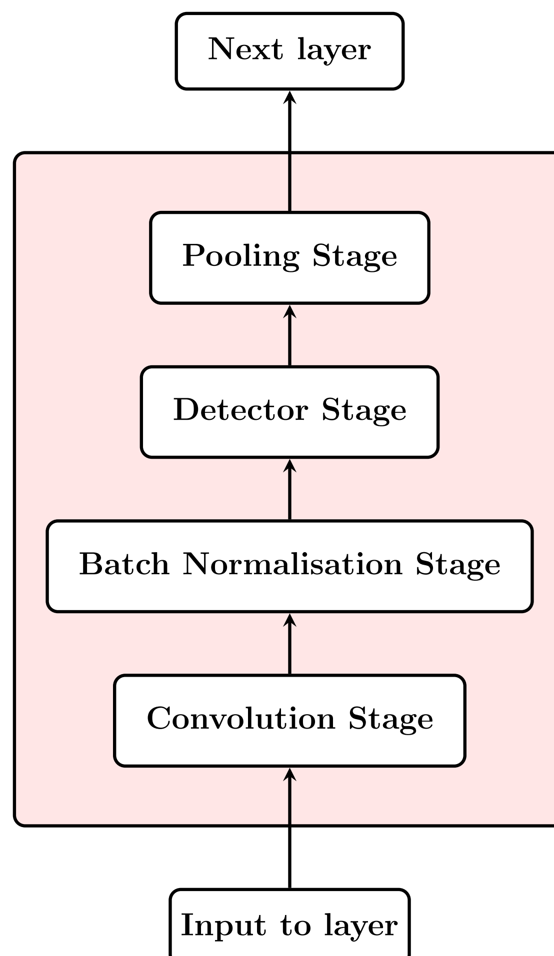


Figure 3.6: A pictorial representation of a convolutional unit, implemented in the `ConvUnit` class (see Listing 3.5).

Pooling

Pooling [11] is used in the implementation to modify the output of a convolutional unit before it is fed to the next layer of the network, by replacing nearby values with a summary statistic of that region. Pooling functions allow down-sampling to be introduced in the network, reducing the dimensionality of the tensor and thereby the subsequent computational effort. This operation also creates translation invariance (see Figure 3.7), which is useful for detecting features regardless of their position in the input.

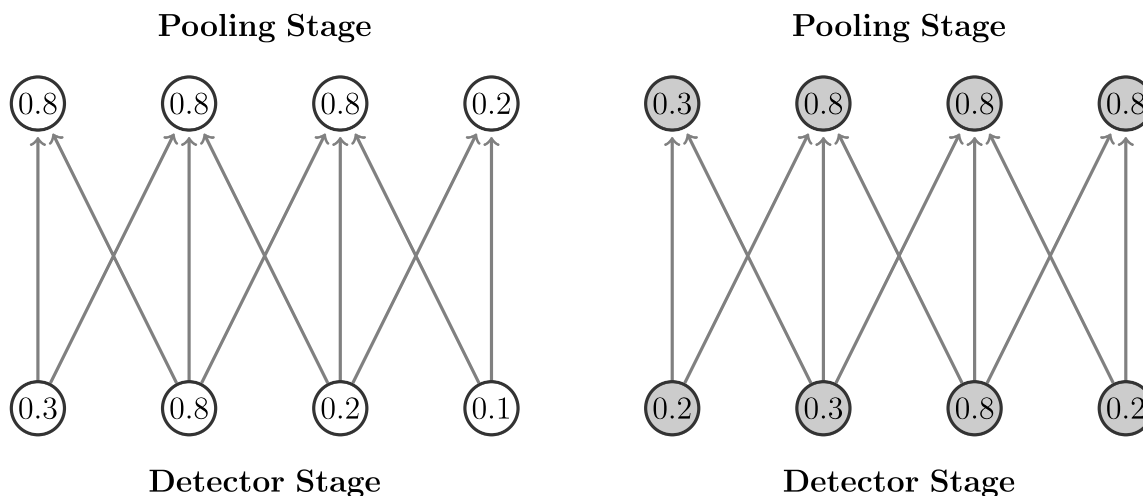


Figure 3.7: The pooling operation over a 1D region of width 3, using a stride of 1. Invariance is achieved as follows: the input to the detector stage in the leftmost diagram has been shifted to the right by a single unit in the rightmost diagram, which is reflected in the outputs of the pooling stage. Despite all of the input values from the detector stage changing, only half of the values after the pooling stage have been affected.

As the deeper layers learn increasingly higher-level features, this invariance is important, because the precise locations of lower-level features such as bass drum frequencies, or the frequency patterns of a guitar solo are not important. Rather, it is the existence of these features that matters to the network when predicting that the audio file can be tagged with “rock”, for example. The stride of the pooling can determine the down-sampling factor. For example, if the detector stage produces 6 outputs, and the pooling width is 3 with a stride of 2 units, then the pooling stage would reduce the representation size by a factor of two as shown in Figure 3.8.

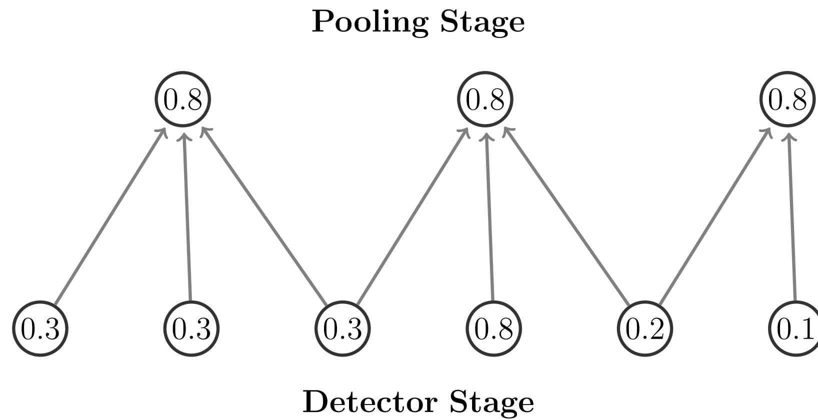


Figure 3.8: Downsampling over regions of width 3, using a stride of 2.

The ELU activation function

The ELU activation, introduced by Clevert, Unterthiner, and Hochreiter [9], alleviates the vanishing gradient effect in a similar manner to that of the ReLU activation—for non-negative values, both of them are identity functions with a constant gradient of 1:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(exp(x) - 1), & \text{if } x \leq 0 \end{cases} \tag{3.2}$$

and the derivative is given by:

$$f(x) = \begin{cases} 1, & \text{if } x > 0 \\ f(x) + \alpha, & \text{if } x \leq 0 \end{cases} \tag{3.3}$$

For non-negative inputs, ELU is not contractive, unlike other activation functions (tanh and sigmoid). For negative inputs, ELU saturates to a negative value determined by α. Figure 3.9 depicts the ELU and ReLU activations, with $\alpha = 1$.

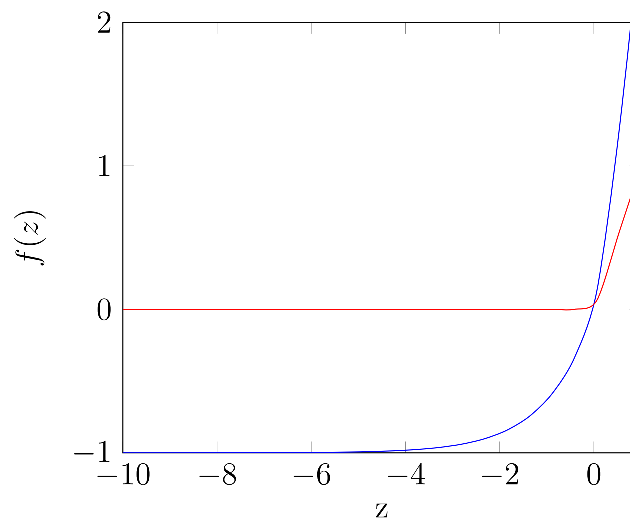


Figure 3.9: The ELU activation in blue plotted against the ReLU activation.

ELUs aim to improve upon the performance of ReLUs by avoiding the bias introduced by having an expected value of the activation greater than zero. Since ReLUs are non-negative everywhere, ELUs have negative values, which brings the mean activation closer to zero. This has been shown to result in faster learning, as ELUs bring the gradient closer to the *unit natural gradient* [9].

Batch normalisation

Batch normalisation [14] was used in the implementation as it generally speeds up training. It is a regularization technique that seeks to reduce a phenomenon referred to as **internal covariate shift**. Due to the hierarchical structure of the network, the input distribution to a given layer depends on the input distribution of the preceeding layer. Ioffe and Szegedy [14] explained that **internal covariate shift** occurs when updating the parameters of a layer in the network. This means that layers must repeatedly re-adapt to the change in input distribution.

The solution Ioffe and Szegedy devised is based on normalising the input distribution for a given layer by the mean and variance of the corresponding mini-batch (denoted by k):

$$\hat{\mathbf{x}}^{(k)} = \frac{\mathbf{x}^{(k)} - \mathbb{E}[\mathbf{x}^{(k)}]}{\sqrt{\mathrm{Var}[\mathbf{x}^{(k)}]}} \tag{3.4}$$

For each activation $x^{(k)}$ a pair of parameters $\gamma^{(k)}$ and $\beta^{(k)}$ are used to scale and shift the normalised value:

$$y^{(k)} = \gamma^{(k)}\hat{\mathbf{x}}^{(k)} + \beta^{(k)} \tag{3.5}$$

During inference, the mean and variance statistics are fixed, since they are not using moving averages as in the case of training. These moving averages facilitate the updates of γ and β by measuring the model accuracy.

The batch-normalised network then performs inference using fixed statistics, gathered during training over the entire training set.

3.2.3 Convolutional architectures

I have implemented three different CNN models, each being formed of several ConvUnit objects that encompass a standard CNN building block. Each model inherits from the `AbstractConvModel` class (see Listing 3.4) and incorporates the creation of TensorFlow layer objects, thus allowing for a cleaner, modular interface.

Listing 3.4: The AbstractConvModel is used to define all three models: *k2c1*, *k1c2*, and *k2c1*. The build model method is overridden by each concrete instance of the abstract class, and uses the ConvUnit class defined in Listing 3.5.

```
class AbstractConvModel:
        def __init__(self, reuse, training_mode):
            self._reuse = reuse
            self._training_mode = training_mode
            super().__init__()

        @abstractmethod
        def build_model(self):
            pass
```

The three convolutional networks I have implemented are named to define both kernel shape (e.g. *k1* for 1D kernels) and convolution dimension (e.g. *c2* for 2D convolutions). Inputs dimensions to the network are $B \times 96 \times 1366 \times 1$ (batch size $\times$ number of mel-frequency bands $\times$ number of samples across time $\times$ channels).

Listing 3.5: The ConvUnit class which is used to define a single convolutional unit (see Figure 3.6).

```
class ConvUnit:
        def __init__(self, pool_strides, pool_size, filters,
                         kernel_size, axis, padding):
                self._pool_size = pool_size
                self._pool_strides = pool_strides
                self._filters = _filters
                self._kernel_size = kernel_size
                self._padding = padding
                self._axis = axis

        def buildUnit(self, input_layer, training_mode=True):
                # build the unit and return it
```

3.2.4 Architectures implemented

An in-depth description of network structures and calculations for the number of parameters follows, to aid the evaluation of each model in Chapter 4. Each of the architectures implemented learns different features due to the differing kernel sizes and their complexity is mainly given by the widths of the network layers. The **layer width** refers to the number of feature maps (within convolutional layers) or the number of hidden units (in fully-connected layers) [8].

Trainable parameters

All the implemented models have been trained with approximately 1M parameters. The number of *trainable* parameters can be computed by considering the number of parameters in each layer $L^{(n)}$. In the convolutional case, the number of parameters is a function of kernel size $[K_1^{(n)}, K_2^{(n)}]$, the number of channels in the input volume C_{n-1} and the number of channels in the output volume C_n. For the fully-connected units, the number of input units I and output units O are considered.

For the convolutional layers, each filter has its own set of parameters *per input channel*. Each filter thus requires $C_{n-1} \times K_1^{(n)} \times K_2^{(n)}$ parameters. Each filter produces an activation map in the output volume, corresponding to the number of output channels C_n. A **convolutional layer** therefore requires $(C_{n-1} \times K_1^{(n)} \times K_2^{(n)}) \times C_n + C_n$ parameters, where the addition of C_n originates from the bias term shared by every neuron in a given output channel.

Recall that in a **fully-connected** layer, every input unit has an edge to every output unit. Defining $|l^n|$ as the number of inputs or outputs in layer n: a layer $l^{(n)}$ with $|l^{(n-1)}|$ inputs and $|l^{(n)}|$ outputs therefore requires $|l^{(n-1)}| \times |l^{(n)}|$ weights and $|l^{(n)}|$ biases, one for each output. The **fully-connected** layer thus requires a total of $|l^{(n-1)}| \times |l^{(n)}| + |l^{(n)}|$ trainable parameters.

Strides and padding

The implementation details of the CNNs require the definition of padded and non-padded convolutional layers. A non-padded convolutional layer simply takes the input, performs the convolution operation and outputs the feature map. A padded convolutional layer, on the other hand, appends zeros to the input to ensure that all of the input locations are "seen" by the convolution operator an equal number of times, such that the output dimensions remain unchanged. In `TensorFlow`, a "valid" convolution adds no padding, whereas the "same" convolution ensures that the input and output have the same dimensions. Figures 3.10 and 3.11 show examples of the two padding modes.

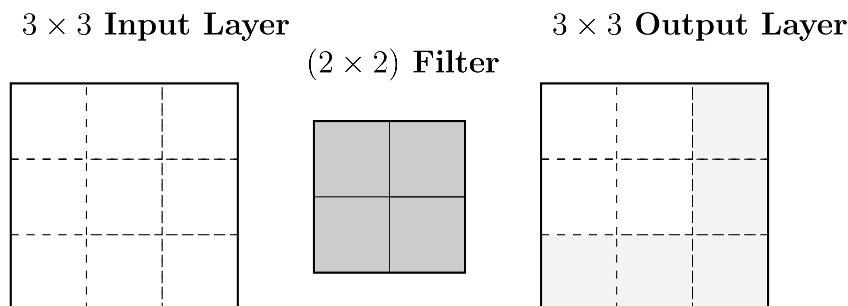


Figure 3.10: **Same** mode: the $[2, 2]$ output is padded with zeros to produce a $[3, 3]$ output.

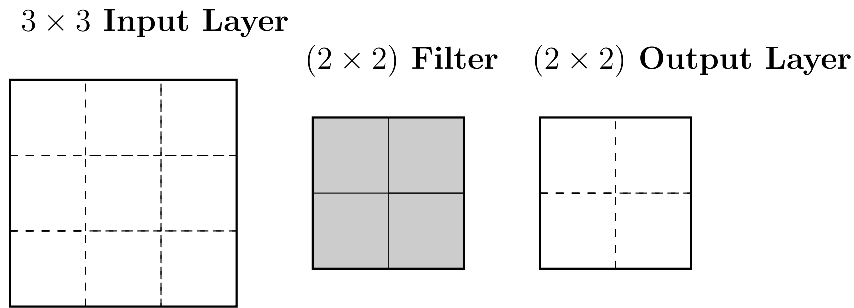


Figure 3.11: **Valid** mode: when a $[2 \times 2]$ kernel is used with a stride of $(1, 1)$ a $[2, 2]$ output is produced.

In general, the output dimensions of a convolutional layer [1] can be written as a function of the input volume W, the padding P, the filter's receptive field size F, and the stride length S: $(W - F + 2P)/S + 1$. For example, in Figure 3.12 a stride of $(1, 1)$ is used which results in the filter being applied 4 times: it is applied at two positions in the x-direction for each of the two positions in the y-direction.

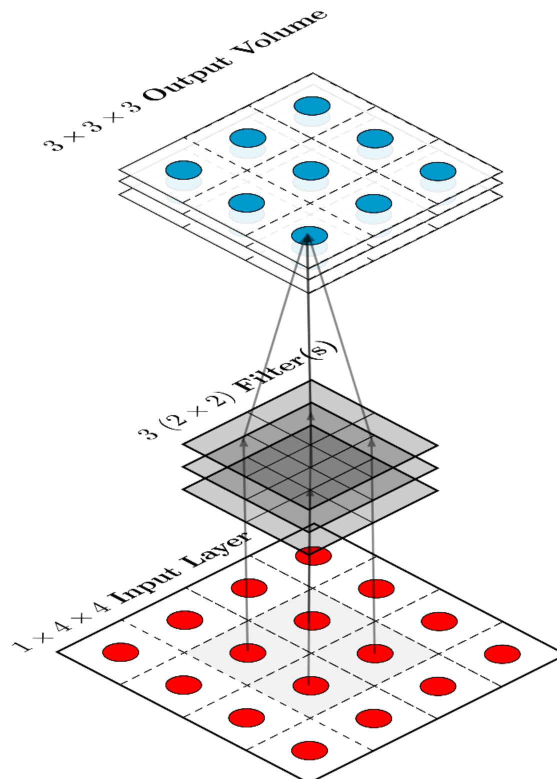


Figure 3.12: An example of a 2D convolutional layer, with 3 $[2 \times 2]$ filters, a stride of $(1, 1)$ and no zero-padding. Neurons are represented by circles. The output volume is 3×3 and consists of three channels: each channel is the activation map corresponding to the response of each filter in the convolutional layer. Each neuron in the output volume has a 2×2 receptive field.

Unit Name	Layer type	Input Size	Output Size	Width	Kernels
ConvUnit1	ConvLayer	$b \times 96 \times 1366 \times 1$	$b \times 1 \times 1363 \times 152$	152	$[96 \times 4]$
	ELU	$b \times 1 \times 1363 \times 152$	$b \times 1 \times 1363 \times 152$		
ConvUnit2	ConvLayer	$b \times 1 \times 1363 \times 152$	$b \times 1 \times 1360 \times 152$	152	$[1 \times 4]$
	BatchNorm	$b \times 1 \times 1360 \times 152$	$b \times 1 \times 1360 \times 152$		
	ELU	$b \times 1 \times 1360 \times 152$	$b \times 1 \times 1360 \times 152$		
	MaxPooling	$b \times 1 \times 1360 \times 152$	$b \times 1 \times 340 \times 152$		
ConvUnit3	ConvLayer	$b \times 1 \times 340 \times 152$	$b \times 1 \times 337 \times 152$	152	$[1 \times 4]$
	BatchNorm	$b \times 1 \times 337 \times 152$	$b \times 1 \times 337 \times 152$		
	ELU	$b \times 1 \times 337 \times 152$	$b \times 1 \times 337 \times 152$		
	MaxPooling	$b \times 1 \times 337 \times 152$	$b \times 1 \times 84 \times 152$		
ConvUnit4	ConvLayer	$b \times 1 \times 84 \times 304$	$b \times 1 \times 81 \times 304$	304	$[1 \times 4]$
	BatchNorm	$b \times 1 \times 81 \times 304$	$b \times 1 \times 81 \times 304$		
	ELU	$b \times 1 \times 81 \times 304$	$b \times 1 \times 81 \times 304$		
	MaxPooling	$b \times 1 \times 81 \times 304$	$b \times 1 \times 20 \times 304$		
ConvUnit5	ConvLayer	$b \times 1 \times 20 \times 304$	$b \times 1 \times 17 \times 304$	304	$[1 \times 4]$
	BatchNorm	$b \times 1 \times 17 \times 304$	$b \times 1 \times 17 \times 304$		
	ELU	$b \times 1 \times 17 \times 304$	$b \times 1 \times 17 \times 304$		
	MaxPooling	$b \times 1 \times 17 \times 304$	$b \times 1 \times 4 \times 304$		
FlatteningLayer		$b \times 1 \times 4 \times 304$	$b \times 1216$		
DenseLayer1		$b \times 1216$	$b \times 304$	304	
DenseLayer2		$b \times 304$	$b \times 304$	304	
OutputLayer		$b \times 304$	$b \times 50$	50	

Table 3.2: The *k2c1* network architecture. **Width** refers to the number of hidden units in fully-connected layers or the number of feature maps in convolutional layers. The convolutional layers for this model use *valid* padding and pooling sizes are all (1×4).

k2c1

The first architecture I implemented was the *k2c1* model [8]: a 2D-kernel, 1D-convolutional architecture which consists of 5 ConvUnits and two fully-connected layers at the tail of the network. The first convolutional unit learns 2D kernels that are convolved along the time axis for the entire frequency band, compressing the information of the whole frequency range into a single band. The following convolutional units learn 1D kernels for convolution along the time axis. The network structure is described in Table 3.2, and the number of trainable parameters for this architecture is calculated in Table 3.3.

Kernel width	**Kernel height**	**Number of filters**	**Learnable parameters**
96	4	152	58520
1	4	152	92568
1	4	152	92568
1	4	304	185136
1	4	304	369968
	Input units	**Output units**	**Learnable parameters**
	1216	304	369968
	304	304	92720
	304	50	15250
			$= 1.2767M \approx 1M$

Table 3.3: Calculations for the number of trainable parameters for *k2c1*. The first and second halves show the calculations for the convolutional layers and the fully-connected layers respectively.

k1c2

For comparison against *k2c1*, the second architecture I have implemented (detailed in Table 3.4) is built from 4 ConvUnits that contain 1D convolutional layers with $[1 \times 4]$ kernels. The max-pooling sizes are $(1 \times 4) - (1 \times 5) - (1 \times 8) - (1 \times 8)$. The convolutional layers have padding, such that the output dimensions of the convolution match the input dimensions. Max pooling then produces summary statistics to reduce the dimensionality. The output of the last ConvUnit is the same height as the input (96), resulting in the feature map encoding a feature for each frequency band. Having only 4 ConvUnits reduces the number of trainable parameters required, detailed in Table 3.5.

Unit Name	**Input**	**Output**	**Layer Width**	**Kernel**	**Pooling**
ConvUnit1	$b \times 96 \times 1366 \times 1$	$b \times 96 \times 341 \times 47$	47	$[1 \times 4]$	$(1, 4)$
ConvUnit2	$b \times 96 \times 341 \times 47$	$b \times 96 \times 68 \times 47$	47	$[1 \times 4]$	$(1, 5)$
ConvUnit3	$b \times 96 \times 68 \times 47$	$b \times 96 \times 8 \times 95$	95	$[1 \times 4]$	$(1, 8)$
ConvUnit4	$b \times 96 \times 8 \times 95$	$b \times 96 \times 1 \times 95$	95	$[1 \times 4]$	$(1, 8)$
FlatteningLayer	$b \times 96 \times 1 \times 95$	$b \times 9120$			
DenseLayer1	$b \times 9120$	$b \times 95$	95		
DenseLayer2	$b \times 95$	$b \times 95$	95		
OutputLayer	$b \times 95$	$b \times 50$	50		

Table 3.4: The *k1c2* network architecture.

Kernel width	Kernel height	Number of filters	Learnable parameters
1	4	47	235
1	4	47	8883
1	4	95	17955
1	4	95	36195
	Input units	**Output units**	**Learnable parameters**
	9120	95	866495
	95	95	9120
	95	50	4800
			$= 0.9437M \approx 1M$

Table 3.5: Calculations for the number of trainable parameters for *k1c2*.

k2c2

The third and final architecture I have implemented is *k2c2* (detailed in Tables 3.6 and 3.7): a fully-convolutional network consisting of consists of 5 ConvUnits with $[3 \times 3]$ kernels, and pooling sizes $(2 \times 4) - (2 \times 4) - (2 \times 4) - (3 \times 5) - (4 \times 4)$. The feature maps in the final layer are 1-dimensional and cover the entire input rather than each frequency band [8], as in *k2c1* and *k1c2*, since the kernels convolve across both axes.

Unit Name	Input	Output	Layer Width	Kernel	Pooling
ConvUnit1	$b \times 96 \times 1366 \times 1$	$b \times 48 \times 341 \times 67$	67	$[3 \times 3]$	$(2, 4)$
ConvUnit2	$b \times 48 \times 341 \times 67$	$b \times 24 \times 85 \times 135$	135	$[3 \times 3]$	$(2, 4)$
ConvUnit3	$b \times 24 \times 85 \times 135$	$b \times 12 \times 21 \times 135$	135	$[3 \times 3]$	$(2, 4)$
ConvUnit4	$b \times 12 \times 21 \times 135$	$b \times 4 \times 4 \times 203$	203	$[3 \times 3]$	$(3, 5)$
ConvUnit5	$b \times 4 \times 4 \times 203$	$b \times 1 \times 1 \times 271$	271	$[3 \times 3]$	$(4, 4)$
FlatteningLayer	$b \times 1 \times 1 \times 271$	$b \times 271$			
OutputLayer	$b \times 271$	$b \times 50$	50		

Table 3.6: The k2c2 network architecture.

Kernel width	Kernel height	Number of filters	Learnable parameters
3	3	67	670
3	3	135	81540
3	3	135	164160
3	3	203	246848
3	3	271	495388
	Input units	**Output units**	**Learnable parameters**
	271	50	13550
			$= 1.0022M \approx 1M$

Table 3.7: Calculations for the number of trainable parameters for *k2c2*.

3.3 Defining loss functions

Recall from earlier discussion (§2.6.1) the notion of a Bernoulli random variable (BRV). Given that the presence of a tag is a BRV, let $\mathbf{y}$ denote the probability that a given song is associated with a tag, as present in the empirical distribution $p_{train}(\mathbf{x})$, and $\hat{\mathbf{y}}$ denote the model's estimate of $\mathbf{y}$, as given by the model distribution $p_{model}(\mathbf{x})$. The binary-cross entropy in Equation 2.9 (§2.6.4) can then be written as:

$$\sum_{i=1}^{m} p_{train}(\mathbf{x})(\log p_{train}(\mathbf{x}) - \log p_{model}(\mathbf{x})) = \mathbf{y}(\log\mathbf{y} - \log\hat{\mathbf{y}}) + (1-\mathbf{y})(\log(1-\mathbf{y}) - \log(1-\hat{\mathbf{y}})) \tag{3.6}$$

The empirical distribution $\mathbf{y}$ is fixed for any given training set; only the model distribution $\hat{\mathbf{y}}$ changes during training, so the following quantity—the **binary cross-entropy loss function**—will be minimised:

$$\mathcal{L} = -\mathbf{y}\log\hat{\mathbf{y}} - (1-\mathbf{y})\log(1-\hat{\mathbf{y}}) \tag{3.7}$$

3.4 Training and evaluating the models

This section follows the use of the ADAM optimiser, which is a variant of Stochastic Gradient Descent, for training the CNN models. Algorithm 3 shows the process of training for a number of epochs, and the use of mini-batching to perform updates.

Algorithm 3 Training the CNN

for epoch in epochs **do**
  while there are still more examples to train on **do**
    run training operation and obtain loss value ▷ compute loss using ADAM
    training loss ← training loss + loss value
    if at the end of the epoch **then**
      add the average training loss to the data collector
      initialise buffers for validation set predictions and corresponding targets
      while there are still more examples to retrieve **do**
        get the next batch of validation predictions
        compute the loss
        validation loss ← validation loss + new loss
        for i in range(evaluation batch size) **do**
          add each prediction, target pair from the batch to the buffers
        end for
      end while
      add the average loss to the data collector
    end if
  end while
end for

3.4.1 Stochastic gradient descent (SGD)

Although traditional gradient descent [12] is effective, it is generally infeasible to update the model after calculating the gradient over the *entire* dataset. Instead, SGD calculates the gradient over a mini-batch of examples and uses this to update the model parameters. The mini-batch gradient is an approximation of the gradient over the entire dataset.

To obtain an unbiased estimate of the gradient, mini-batching randomly selects m examples $\{\mathbf{x}^{\mathbf{1}}, ..., \mathbf{x}^m\}$ from the training data set [11] and takes the average.

Algorithm 4 Stochastic Gradient Descent (SGD)

1: **Require:** Learning rate α_k
2: **Require:** Initial parameter θ
3: **while** there are still examples to see **do**
4:   Obtain a mini-batch of m examples $\{\mathbf{x_1}, ..., \mathbf{x}_m\}$ with corresponding targets $\mathbf{y}^{(i)}$
5:   Compute the unbiased estimate of the gradient: $\hat{\mathbf{g}} \leftarrow +\frac{1}{m}\nabla_\theta \mathcal{L}(f(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}))$
6:   Apply update $\theta \leftarrow \theta - \alpha_k \hat{\mathbf{g}}$
7: **end while**

3.4.2 The ADAM optimiser

ADAM [15] is an **adaptive learning rate** optimisation algorithm [11] that addresses one of the most time-consuming challenges related to the training of neural networks: tuning hyper-parameters. Adjusting the learning rate can have profound effects on training times and the optimality of the learned parameters. The algorithm, which accumulates the gradients with respect to each parameter by taking exponentially weighted moving averages, computes estimates of the first and second moments of the gradient and uses these to perform parameter updates. The TensorFlow implementation of the algorithm was used and has four parameters:

1. α: the learning rate (default value 0.001),
2. β_1: the decay rate of the first moment of the gradient (default value 0.9),
3. β_2: the decay rate of the second moment of the gradient (default value 0.999),
4. ϵ: for numerical stability (default value $1e$-08).

ADAM stores estimates of the first and second moments of the gradient, which are used to alter the learning rates for each parameter update. By computing the gradient of the loss with respect to each of the parameters and using adaptive learning rates per-parameter, they are being updated such that a better set of parameters is reached. This is achieved by scaling the rates inversely proportional to the square root of the second moment of the gradient, which is computed element-wise as the following:

$$\mathbf{g_t} = \frac{1}{m}\nabla_\Theta \sum_i \mathcal{L}(f(\mathbf{x}^{(i)};\Theta_{t-1}),\mathbf{y}^{(i)}) \tag{3.8}$$

Algorithm 5 The ADAM algorithm

Require: α ▷ step size
Require: β_1, β_2 in $[0,1)$ ▷ exponential decay rates for moment estimates
Require: ϵ ▷ small constant for numerical stabilization
Require: Θ_0 ▷ initial parameter vector
  $\mathbf{m_0} \leftarrow \mathbf{0}, \mathbf{v_0} \leftarrow \mathbf{0}$ ▷ initialize 1st and 2nd moment variables
  $t \leftarrow 0$ ▷ initialize time step
while stopping criterion not met **do**
  Sample mini-batch of m examples from training set $\{\mathbf{x}^{(1)},...,\mathbf{x}^{(m)}\}$ with targets $\mathbf{y}^{(i)}$
  $t \leftarrow t+1$
  $\mathbf{g_t} \leftarrow \frac{1}{m}\nabla_\Theta \sum_i \mathcal{L}(f(\mathbf{x}^{(i)};\Theta_{t-1}),\mathbf{y}^{(i)})$ ▷ compute gradient wrt objective
  $\mathbf{m}_t \leftarrow \beta_1\mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t$ ▷ update biased first moment estimate
  $\mathbf{v}_t \leftarrow \beta_2\mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2$ ▷ update biased second moment estimate
  $\hat{\mathbf{m}}_\mathbf{t} \leftarrow \mathbf{m_t}/(\sqrt{1-\beta_1^t})$ ▷ correct bias in first moment
  $\hat{\mathbf{v}}_\mathbf{t} \leftarrow \mathbf{v}_t/(\sqrt{1-\beta_2^t})$ ▷ correct bias in second moment
  $\mathbf{\Theta}_t \leftarrow \mathbf{\Theta}_{t-1} - \alpha\hat{\mathbf{m}}_\mathbf{t}/(\sqrt{\hat{\mathbf{v}}_\mathbf{t}}+\epsilon)$ ▷ apply the update
end while
  return $\mathbf{\Theta}_t$

I used the TensorFlow implementation (see Listing 3.6) of the ADAM optimization algorithm for computing the gradient, and (as mentioned in Section 3.3) the **binary cross-entropy** loss function (§2.6.4), which computes the loss element-wise for each tag. Substituting Equation 3.7 (§3.3) into Equation 3.8 gives a closed expression for second moment of the gradient used by ADAM:

$$\mathbf{g_t} = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i -\mathbf{y}^{(i)}\log f(\mathbf{x}^{(i)};\boldsymbol{\theta}_{t-1}) - (1-\mathbf{y}^{(i)})\log(1-f(\mathbf{x}^{(i)};\boldsymbol{\theta}_{t-1})) \tag{3.9}$$

where $f(\mathbf{x}^{(i)};\boldsymbol{\theta}_{t-1})$ denotes the output of the network for batch i, using parameters $\boldsymbol{\theta}_{t-1}$ from the last epoch.

Listing 3.6: Setting up the loss function and optimisation algorithm in TensorFlow.

```
# Use the TensorFlow loss function implementation, taking the model
    output and target output as arguments
loss_training = tf.losses.sigmoid_cross_entropy(multi_class_labels=
    target_batch_training,logits=training_model_output)
# Define the optimizer
adam_optimizer = tf.train.AdamOptimizer()
# Construct a training operation
training_operation = adam_optimizer.minimize(loss=loss_training)
```

3.4.3 Evaluating the model

Data collection

The models were evaluated using the testing set, and a number of metrics were stored: precision-recall, AUC-ROC and accuracy. At the end of every epoch, the `DataCollector` module saves a file for later analysis, which required me to implement the aforementioned metrics using functionality provided by the `sci-kit learn` package [7]—a machine learning and data mining library. I will define the metrics and discuss the results in the Evaluation chapter.

Restoring a model

Checkpoints were saved at the end of every epoch to enable the restoration of trained models for use in the tagging system and for further evaluation. The system loads the relevant checkpoint file, reconstructs the graph, and then loads the saved model weights and variables.

[7] http://scikit-learn.org/stable/

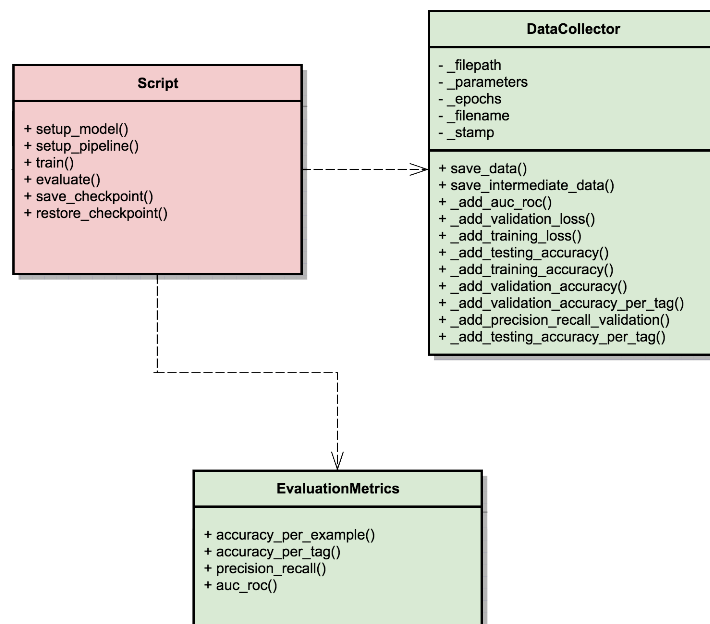


Figure 3.13: A UML diagram showing the class design for the DataCollector and its interaction with the script.

Chapter 4

Evaluation

This chapter discusses the testing process and suitable evaluation metrics for assessing the performance of the CNN architectures, with results showing that the relative performances of the models are in accordance with the paper [8]. I also analyse the potential of the best performing model (*k2c2*) for music recommendation through visualisations of the learned latent space and by investigating the tags given to several songs that were unseen during training.

4.1 Testing

The music tagging system I have presented in the previous chapter requires interaction between several modules. The latter thus required isolated testing before being integrated, which was carried out using Python's `UnitTest` module. Tests for `ConvUnit`, instances of `AbstractConvModel` and `EvaluationMetrics` were created and run (see Figure 4.1). This ensured that the `ConvUnit` and `AbstractConvModel` return correct tensors with the corresponding output shapes. Additionally, the `EvaluationMetrics` method results were asserted against pre-calculated values to establish that correct statistics can be provided.

Unit tests ensure modular correctness, but are insufficient in ensuring that the entire system works as required. Since the project utilised the `TensorFlow` API, *integration tests* were performed to test the main tagging system module, which utilises all the modules that previously underwent unit testing and the `TensorFlow` API. For all the CNN models implemented, the *training binary cross-entropy loss* was monitored—if its value is decreasing during training, then the parameters of the model are indeed being updated to reach an optimal state. A graph of the training loss for *k2c1* is shown in Figure 4.2.

As previously discussed in the Implementation chapter, the timeline of the project imposed a great restriction in terms of the training time allowed, even when using a GPU. I have therefore optimised the data pipeline for loading the dataset in batches from disk, which has led to a drastic reduction of training times per epoch, as shown in Figure 4.3.

Perhaps most notably, the optimised pipeline has brought a significant improvement over

```
(tensorFlow) dhcp-10-249-89-95:src Andy$ python3 tests.py
testBuildUnit (test_convunit.ConvUnitTest) ... 2018-04-26 20:27:28.6472
mpiled to use: AVX2 FMA
ok
test_session (test_convunit.ConvUnitTest)
Returns a TensorFlow Session for use in executing tests. ... ok
testAccuracyPerTag (test_metrics.MetricsTest) ... ok
testComputeF1Score (test_metrics.MetricsTest) ... ok
testComputeROC (test_metrics.MetricsTest) ... ok
test_session (test_metrics.MetricsTest)
Returns a TensorFlow Session for use in executing tests. ... ok
testScope (test_convmodel.ConvModelTest) ... ok
test_session (test_convmodel.ConvModelTest)
Returns a TensorFlow Session for use in executing tests. ... ok

----------------------------------------------------------------------
Ran 8 tests in 3.451s

OK
```

Figure 4.1: Unit test results.

the training times reported by Choi et al. [8]. For example, they report a runtime of 2400 seconds per epoch for the *k2c1* model, whereas my pipeline allows this to decrease more than threefold to 727±8.38 seconds.

4.2 Performance evaluation

The lengthy pipeline optimisation process only allowed me to train the three architectures for the following number of epochs—150 for *k2c1* and *k2c2*, 89 for *k1c2*. During training, the validation loss continued to generally decrease and so further performance improvement can be expected in subsequent epochs. However, the results I have obtained, which are described in the remained of this section, accomplish and significantly surpass the **core criterion** of my project proposal for *k2c1*, furthermore achieving several **extensions**.

4.2.1 Accuracy

One method of evaluating the performance of the CNN models is to measure the per-tag *accuracy*. The classifier was tested on the test set s of m (28,534) examples, giving the output probabilities of the classifier $h_\theta(\mathbf{x})$. The output probabilities were rounded up (if ≥ 0.5) or down (if < 0.5), giving a binary outcome for each tag, and then compared against the *ground truth labels*. If a given tag is associated with a song, an output of 1 from the classifier is deemed as a success; if, instead, the tag is not associated with the song, an output of 0 is considered a success.

The *per-tag accuracy score* [12] (averaged over m samples) can be defined as:

$$\text{accuracy}_{tag} = \frac{1}{m} \sum_{i=1}^{m} \mathbb{I}[h_\theta(x^{(i)})_{tag} = y_{tag}^{(i)}] \tag{4.1}$$

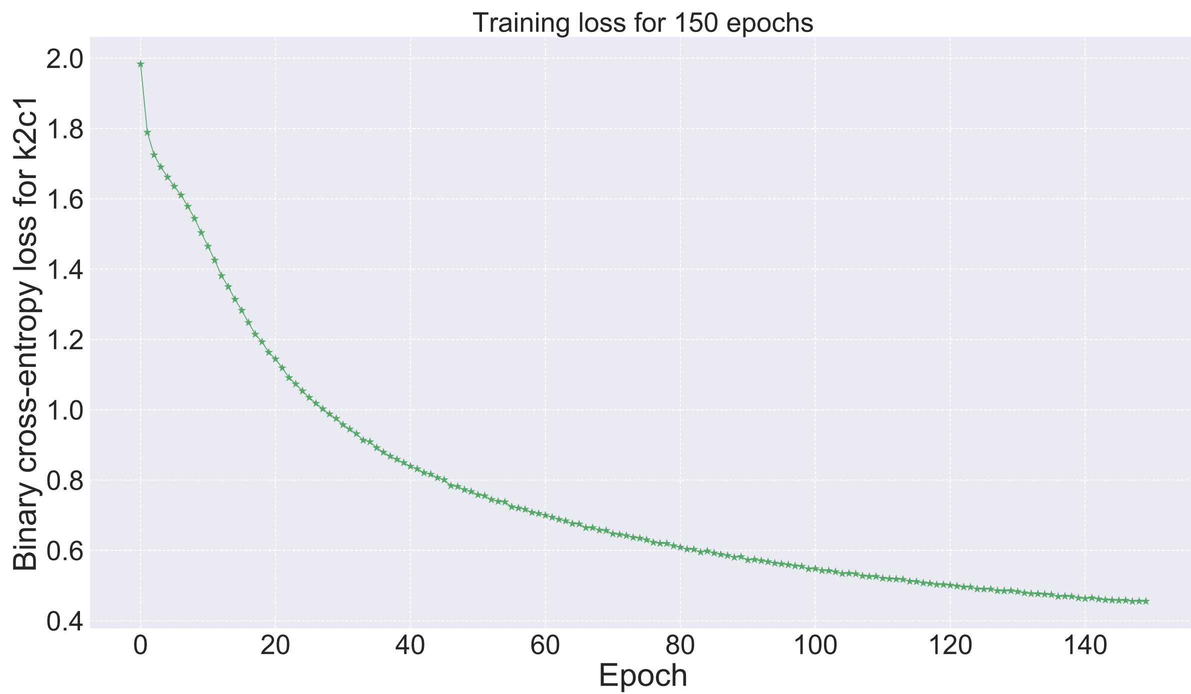


Figure 4.2: The training loss for the *k2c1* model.

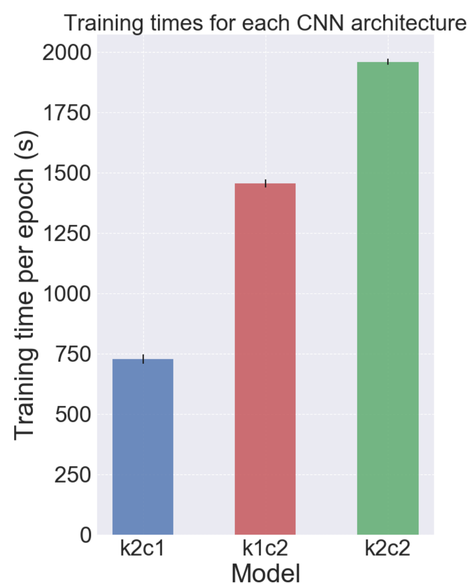


Figure 4.3: The training times per epoch for each architecture. Each model has approximately 1M trainable parameters. Error bars are shown (in black) and were constructed using the Student's t-Distribution at a 95% confidence level.

For the accuracy scores, confidence intervals have been constructed over $N = 3$ training sessions. Each training session was run for 30 epochs, and accuracy scores computed after this period. It was a *core criterion* to achieve above 50% accuracy on each tag and this has been **achieved**.

Since the sample size N is less than 30, the Student's t-distribution[1] was used to construct a two-tailed 95% confidence interval for the mean accuracy score of each tag:

$$CI_{tag} = \text{accuracy}_{tag} \pm t\frac{s}{\sqrt{N}}$$

where s is the standard deviation.

Accuracy results for the ***k2c1*** model

A core success criterion in the proposal was to obtain better-than-chance prediction (accuracy above 50% for each tag and the lower bound of its confidence interval above 50%) of the top-50 tags. This goal has been **achieved**: accuracy scores range from 81%–99%, as presented in Tables 4.1 and 4.2.

[1] http://mathworld.wolfram.com/Studentst-Distribution.html

Table 4.1: *k2c1* accuracy results for genres.

Tag	**Accuracy score**
heavy metal	98.0% ± 0.2%
indie pop	97.9% ± 0.6%
rnb	97.74% ± 0.02%
house	97.7% ± 0.2%
hard rock	97.6% ± 0.4%
acoustic	97.3% ± 2.7%
hip-hop	97.3% ± 1.8%
electro	97.2% ± 3.2%
funk	96.9% ± 3.3%
punk	96.9% ± 0.3%
metal	96.7% ± 0.6%
electronica	96.6% ± 0.6%
experimental	96.4% ± 5.3%
classic rock	96.3% ± 4.5%
alternative rock	96.2% ± 1.2%
progressive rock	96.1% ± 0.1%
country	96.0% ± 0.6%
indie rock	95.8% ± 3.3%
blues	95.7% ± 4.5%
soul	94.5% ± 4.9%
dance	94.4% ± 0.9%
folk	94.0% ± 4.5%
jazz	93.0% ± 3.7%
alternative	92.2% ± 1.0%
electronic	89.7% ± 5.1%
indie	88.9% ± 0.6%
pop	87.5% ± 0.5%
rock	81.8% ± 2.8%

Table 4.2: *k2c1* accuracy results for the moods, instruments and eras categories.

Tag	Accuracy score
Moods:	
happy	99.6% ± 0.1%
party	99.4% ± 0.1%
catchy	99.4% ± 0.1%
sad	99.4% ± 0.5%
sexy	99.2% ± 1.1%
easy listening	98.9% ± 1.1%
chill	98.8% ± 0.3%
beautiful	98.7% ± 0.6%
mellow	98.3% ± 0.5%
chillout	96.5% ± 1.8%
ambient	96.3% ± 0.5%
oldies	96.0% ± 0.6%
Instruments:	
female vocalist	98.9% ± 1.4%
male vocalists	98.4% ± 0.4%
guitar	98.2% ± 1.6%
instrumental	94.6% ± 2.5%
female vocalists	91.2% ± 2.0%
Eras:	
00s	98.4% ± 0.3%
70s	97.54% ± 0.04%
60s	97.3% ± 1.8%
90s	97.3% ± 0.7%
80s	94.9% ± 6.3%

Upon further inspection of the dataset during the preparation phase, I realised that accuracy is not the most suitable metric, as the MSD dataset is imbalanced. This can lead to rare tags acheving high accuracies, which can provide a misleading indicator of the system performance. Therefore, the following alternative metrics were used for the rest of the evaluation: AUC-ROC, precision and recall—the former being used in [7], [8], and [10] for assessing music-tagging systems.

4.2.2 Precision and recall

When evaluating my music-tagging system, I found it more useful to consider alternative performance measures for imbalanced data. If we take the test set $s = \{(\mathbf{x_1}, b_1), ..., (\mathbf{x_n}, b_n)\}$ of tuples $(\mathbf{x_i}, b_i)$ denoting the i-th test sample and the ground truth (binary label $b \in \{0, 1\}$) for the i-th test sample respectively, then we can define the following quantities:

- **True positives (TP)**: the number of instances of labelling as positive a sample that is positive:
$$TP = |\{(\mathbf{x}, 1) \in s | h_\theta(\mathbf{x}) = 1\}| \tag{4.2}$$
- **False positives (FP)**: the number of instances of labelling as positive a sample that is negative:
$$FP = |\{(\mathbf{x}, 0) \in s | h_\theta(\mathbf{x}) = 1\}| \tag{4.3}$$
- **True negatives (TN)**: the number of instances of labelling as negative a sample that is negative:
$$TN = |\{(\mathbf{x}, 0) \in s | h_\theta(\mathbf{x}) = 0\}| \tag{4.4}$$
- **False negatives (FN)**: the number of instances of labelling as negative a sample that is positive:
$$FN = |\{(\mathbf{x}, 1) \in s | h_\theta(\mathbf{x}) = 0\}| \tag{4.5}$$

where $|\bullet|$ denotes the cardinality of its argument.

Precision and *recall* are two metrics based on the number of true positives, false positives, true negatives, and false negatives. Both of these properties are valuable metrics, since it is often the case that we are required to trade off precision in the favour of recall, or vice-versa. For example, since I am recommending music based upon the tags, it is perhaps better to have a high precision, since erroneously tagging a song may lead to poor recommendations. The two metrics, along with a third one that summarises them, are defined as [2]:

- **Precision** (PPV): the positive predictive value, defined as the ratio of true positives to the sum of the true positives and false positives:
$$\frac{TP}{TP + FP} \tag{4.6}$$
summarises the ability of the classifier not to label as positive a sample that is negative.
- **Recall** (TPR): the true positive rate, defined as the ratio true positives to the sum of the true positives and false negatives:
$$\frac{TP}{TP + FN} \tag{4.7}$$
summarises the ability of the classifier to find all the positive samples.

[2] http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html

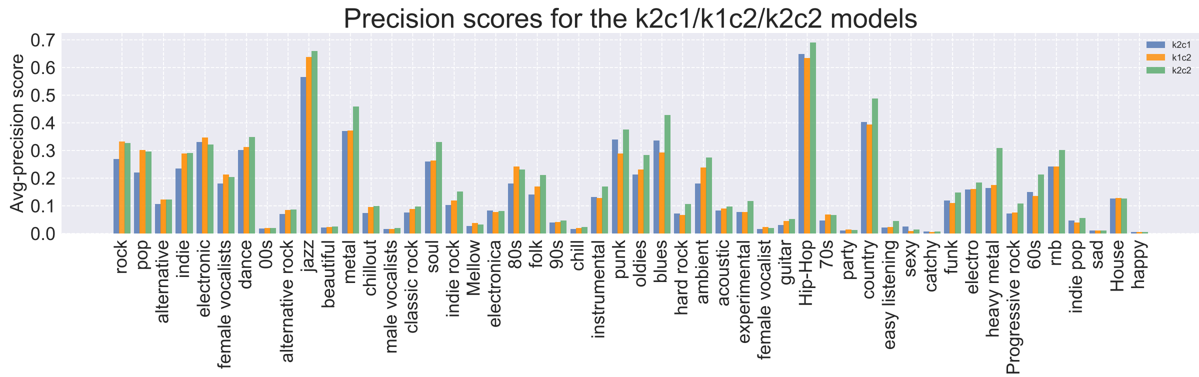


Figure 4.4: Average precision scores of the three architectures, each having 1M parameters. The blue, orange and green denote the precision averages for *k2c1*, *k1c2* and *k2c2* respectively.

- **Average precision**[3] (AP): a summary of the precision-recall curve as a weighted mean of precisions obtained at each recall level, with the previous threshold used as the weight:

$$AP = \sum_n (R_n - R_{n-1}) P_n \tag{4.8}$$

Figure 4.4 displays average precision scores for all models: *k2c2* outperforms the other two models for 32 of 50 tags.

4.2.3 AUC-ROC

The accuracy score metric has two caveats when dealing with class imbalances: fewer instances of a tag occurrence increase the likelihood of a higher score, even if all the predictions are incorrect. Secondly, the acceptability threshold is set at 0.5: this does not consider other thresholds and fails to explain the certainty in the decision of the classifier. AUC-ROC overcomes this by observing the TP- vs FP-rate for all possible thresholds. The true positive rate can be plotted against the false positive rate for varying acceptability thresholds to produce the **receiver operator characteristic** (ROC curve). The area under the ROC curve (AUC) is referred to as the AUC-ROC metric.

Average AUC-ROCs for each structure are plotted in Figures 4.5 and 4.6, and are in agreement (performance of *k2c2* > *k1c2* > *k2c1*) with the results reported in the paper [8] (per-tag AUC-ROCs for each model are shown in Figures 4.7, 4.8, 4.9). *k2c2* outperforms *k2c1* for *all tags*, and *k1c2* also outperforms *k2c1* when averaged across all tags (see Figure 4.6), which means I have achieved an **extension goal:** “implement more architectures to improve performance”.

[3] http://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html

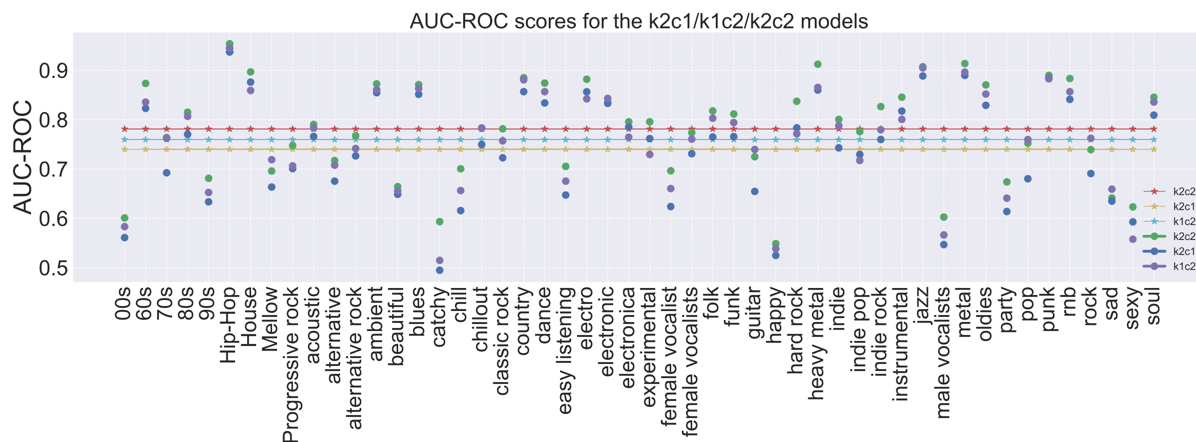


Figure 4.5: AUCs for the top-50 tags obtained from each of the three architectures. The red, yellow and blue lines are the AUC averages for *k2c2*, *k2c1* and *k1c2* respectively. The green, purple, and blue points are the per-tag AUC scores for *k2c2*, *k1c2* and *k2c1* respectively.

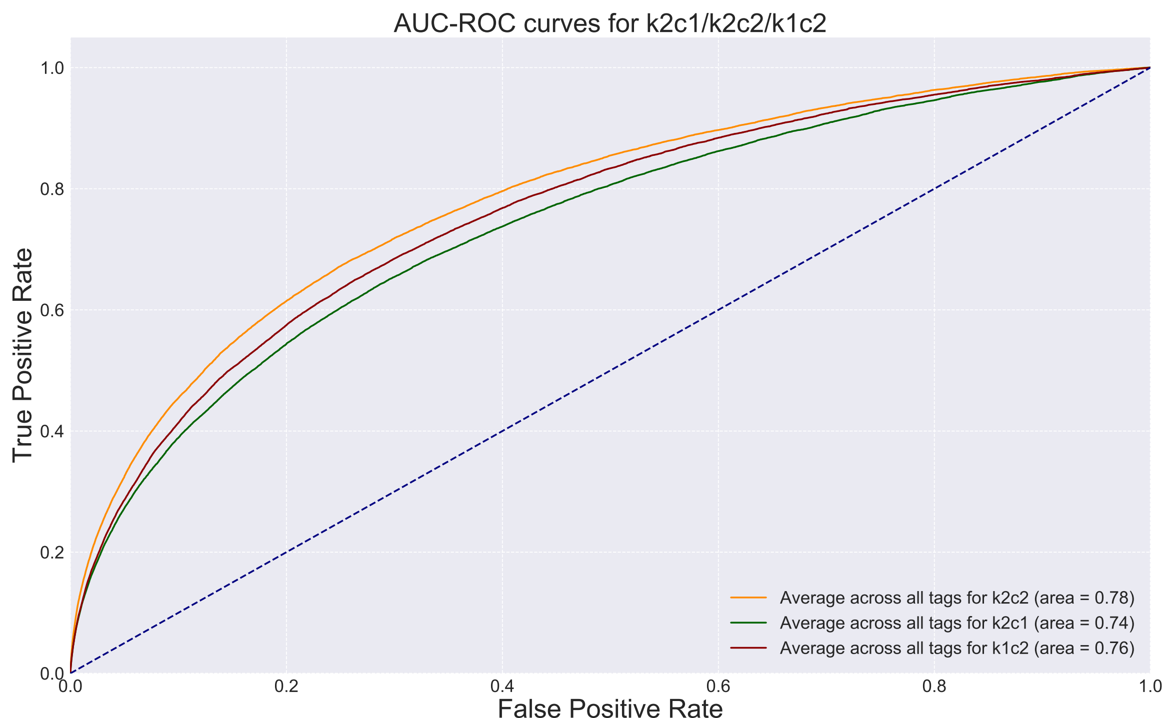


Figure 4.6: AUCs averaged across the top-50 tags obtained from each of the three architectures. The orange, red, and green lines are the AUC averages for *k2c2*, *k1c2* and *k2c1* respectively.

Rather than output learned features into two fully connected layers as in *k2c1* and *k1c2*, *k2c2* is instead **fully-convolutional**, and the performance gain can be attributed to the first layer: each convolution is across both time frames and mel-frequency bands, increasing the model's ability to recognise higher-level features.

4.3 Using deep learning for music recommendation

4.3.1 An analysis of predictions

After the CNN model training was completed, the resulting tagging system could be easily restored, using the checkpoint functionality described in the Implementation chapter. This allowed songs from the test set to be fed to the system and generate tag predictions in the form of 50-dimensional vectors. A selection of songs tagged with a variety of genres, moods, instruments and eras are listed in Table 4.3. When comparing these predictions with AllMusic[4] and Spotify artist biographies[5], genre predictions are consistent, including the additional tags from moods, genres and instruments.

For some songs, such as "The Carol Of The Bells" by Frankie Valli & The Four Seasons, the system predicts some tags—60s and instrumental, for example—with low confidence, yielding probabilities less than 0.5. However, the existence of these tags is important because it allows further differentiation between this song and other "oldies" songs. This advanced level of insight into complex musical features is of key importance in using the tags to drive music recommendation (§4.3.2) and consequently illustrates the potential of these music tagging models to be part of a larger recommender system.

Artist	Song title	Tags	Scores
Lou Reed / The Velvet Underground	Femme Fatale	indie	0.98
		classic rock	0.87
		guitar	0.41
Duke Ellington and His Famous Orchestra	Ko-Ko	jazz	1.0
LCD Soundsystem	Sound of Silver	electronic	1.0
		electronica	0.81
Frankie Valli & The Four Seasons	The Carol Of The Bells	oldies	1.0
		male vocalist	0.90
		instrumental	0.15
		60s	0.11

Table 4.3: Tag predictions for various songs from the test set.

[4] https://www.allmusic.com/
[5] https://support.spotify.com/us/using_spotify/features/artist-profile/

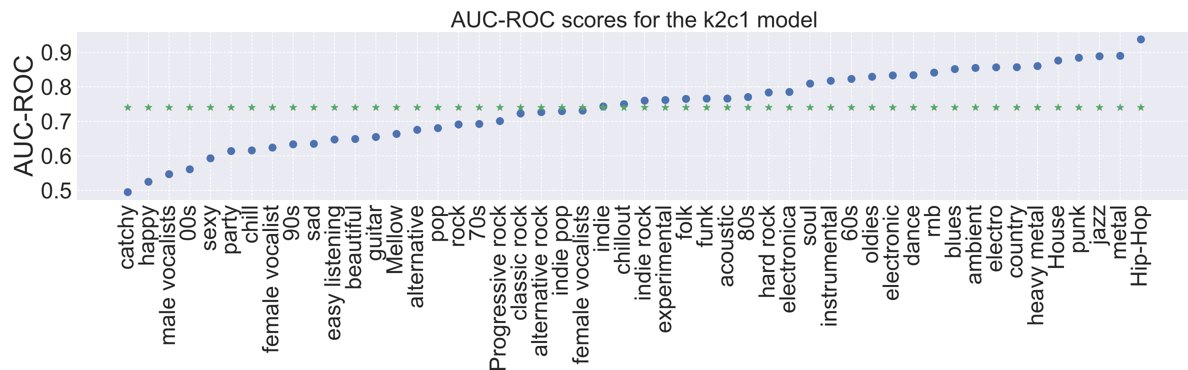


Figure 4.7: AUCs for each of the top-50 tags obtained from the *k2c1* model with 1M parameters.

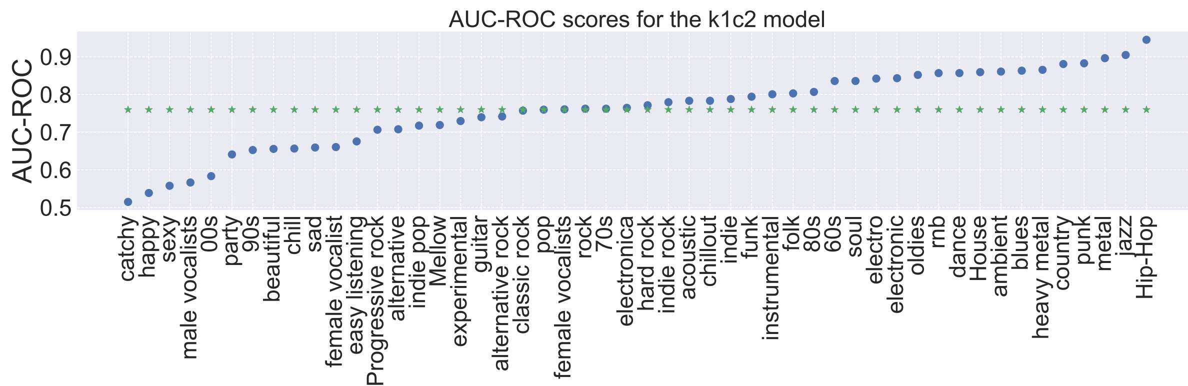


Figure 4.8: AUCs for each of the top-50 tags obtained from the *k1c2* model with 1M parameters.

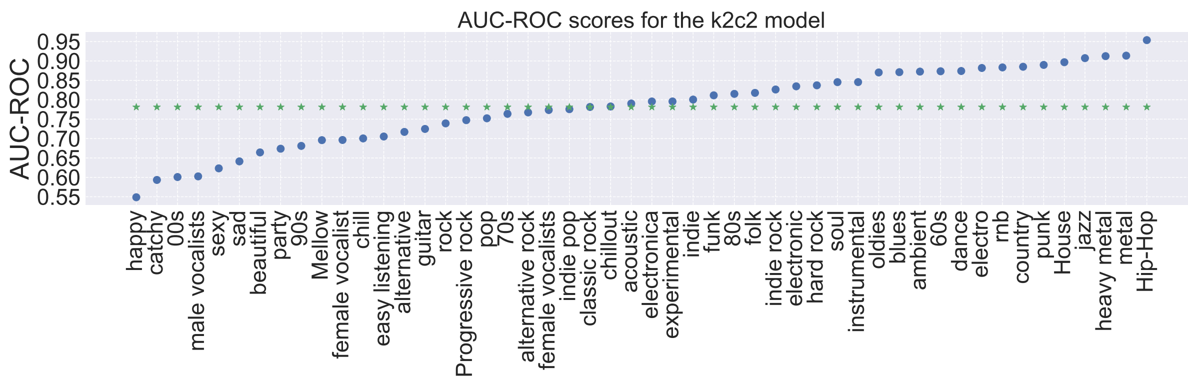


Figure 4.9: AUCs for each of the top-50 tags obtained from the *k2c2* model with 1M parameters.

4.3.2 Exploring the latent representation of songs

Another extension involves a visual exploration of the predictions from the tagging system, in order to evaluate the potential of suggesting similar songs to users.

Each 50-tag vector can be seen as a representation of the corresponding song, lying in 50-dimensional space. To visualise songs that were considered similar by the music tagger, I took the predicted tags of songs in the test set (unseen during training) and processed the 50-dimensional vectors using t-Distributed Stochastic Neighbour Embedding (t-SNE) [20]. This is a technique for dimensionality reduction that aims to preserve the distances between points x_i and x_j in high-dimensional space and the mapped-to points y_i and y_j in the low-dimensional space.

t-SNE describes the *similarity* of two data points x_i and x_j based on the conditional probability $p_{j|i}$ that x_i would chose x_j as its neighbour, if chosen in proportion to its probability density centred at x_i. t-SNE uses the Student t-distribution to compute the similarities between two points in low-dimensional space. The variance σ^2 is computed using the user-specified *perplexity*—a smooth measure of the effective number of neighbours.

The technique minimizes the KL divergence between the joint probability distribution P, in the high-dimensional space, and the joint probability distribution Q in the low-dimensional space:

$$C = KL(P||Q) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}, \tag{4.9}$$

where $p_{ii}, q_{ii} = 0$, the pairwise similarities in the low-dimensional space are given by:

$$q_{j|i} = \frac{exp(-||y_i - y_j||^2)}{\sum_{k \neq l} exp(-||y_k - y_l||^2)} \tag{4.10}$$

and the pairwise similarities in high-dimensional space are given by:

$$p_{j|i} = \frac{exp(-||x_i - x_j||^2)/2\sigma^2}{\sum_{k \neq l} exp(-||x_k - x_l||^2)/2\sigma^2} \tag{4.11}$$

The results that follow were obtained using tags predicted by the system and the t-SNE functionality available in the `sci-kit learn` library. After the dimensionality reduction to the 2D latent space, Figure 4.10 shows the clustering[6] of 10,000 songs based on the tags predicted by the *k2c2* model, and the corresponding artists are shown in Figure 4.11.

The t-SNE plot in Figure 4.12 shows a cluster of blues and jazz artists. Table 4.4 shows a selection of artists and song tags appearing in the cluster. Songs from each of these artists share labels "blues" and "jazz", demonstrating that the clusters formed by t-SNE make use of common tags between artists, which further improves the music recommendation abilities of the system.

[6] Artists are omitted here to allow for clearer visualisation of the clustering.

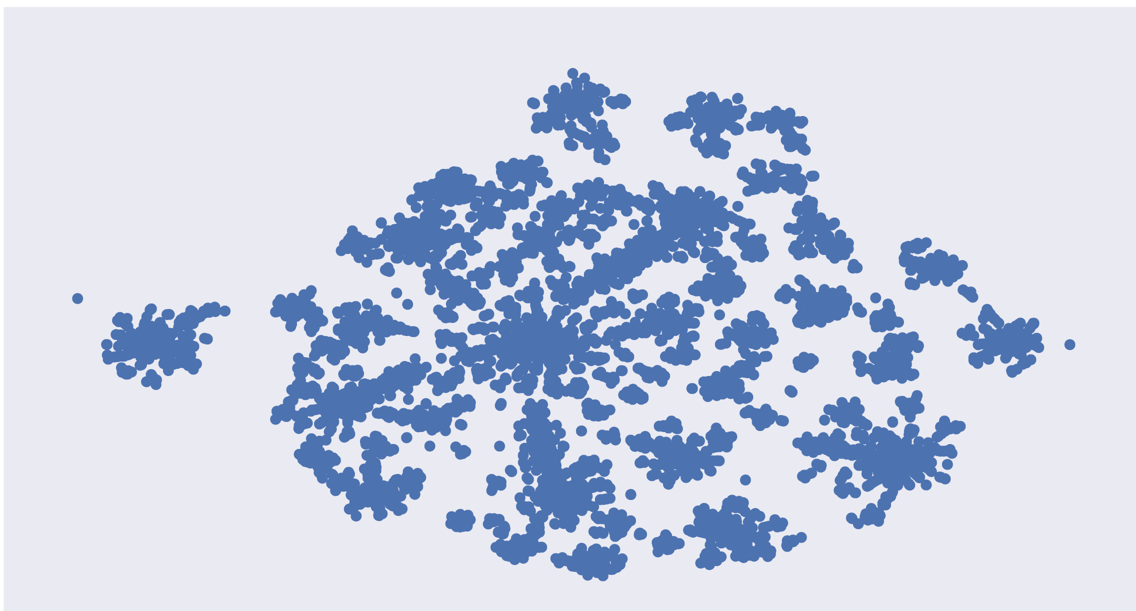


Figure 4.10: A t-SNE plot of 10,000 songs based on their predicted top-50 tags.

Artist	Song title	Tags	Scores
Cab Calloway	Doin' The Rumba	jazz blues	1.0 0.58
Otis Rush	Right Place Wrong Time	blues jazz	0.95 0.49
Muddy Waters / Son Simms Four	Ramblin' Kid Blues	blues jazz	1.0 0.50

Table 4.4: Tag predictions for songs by artists appearing in the t-SNE cluster shown in Figure 4.12.

Playlist curation, artist discovery and song recommendations

High-level representations of audio files are an extremely powerful and useful tool for recommendation. The t-SNE plots provide a way of visualising clusters present between songs (and implicitly artists), which can be easily used to drive artist or song discovery. More generally, to explore new music, whenever a user listens to a song, the title or artist can be found in the latent space and nearby points used to provide new suggestions. Another application is **playlist curation**, which is similar to music recommendation, but starts from a random song vector. The playlist can then be constructed based on the closest k neighbours of the song vector in the latent space.

Figure 4.11: The t-SNE plot in Figure 4.10 with artist names.

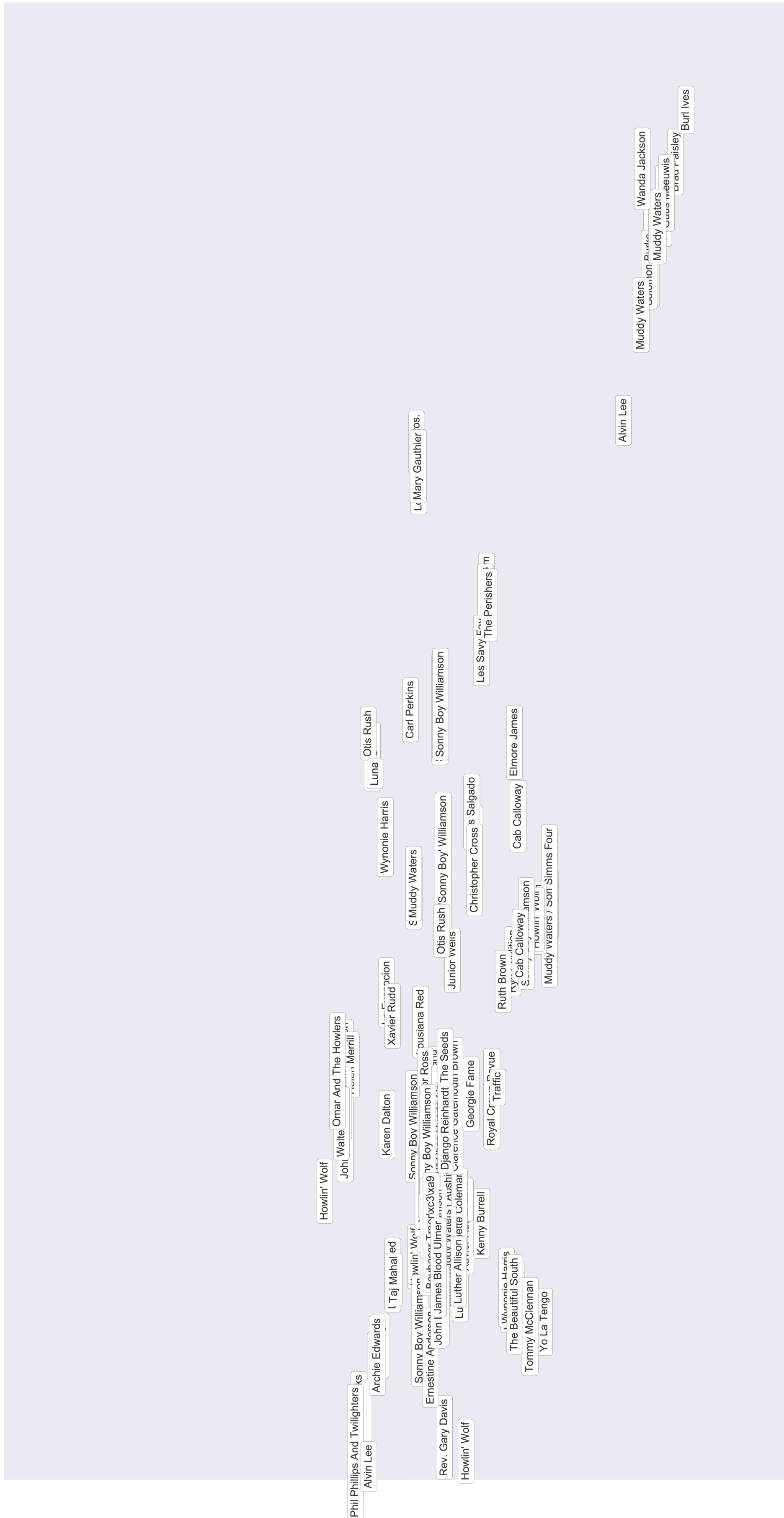


Figure 4.12: The t-SNE plot from Figure 4.11 showing a close-up view of artists.

Chapter 5

Conclusion

The project has been a success, exceeding all success criteria and demonstrating that deep learning architectures are powerful tools for content-based music recommendation.

5.1 Achievements

Deep Learning for Music Recommendation was a hugely successful project. All core objectives were completed: the tagging system is fully functional and the *k2c1* model achieves better-than-chance performance on all top-50 tags. Moreover, two of the three initially proposed extensions were achieved. The first one shows that the performance of the *k2c1* model was improved on each tag, by implementing two additional models: *k1c2* and *k2c2* (§3.2.4).

The goal of the project was to build a CNN system capable of predicting the top-50 last.fm (§3.1.1) tags associated with a given song, which has been successfully achieved (§4.3.1). Its ability to predict these tags has, in turn, allowed me to complete the second extension by exploring (§4.3.2) content-based music recommendation using dimensionality reduction techniques. This has the potential of enabling automatic artist discovery, song recommendations and playlist curation (§4.3.2)—features heavily invested in by the music industry to further improve the way users interact with and listen to music.

5.2 Lessons learned

Throughout the project, I have learned not only theoretical aspects of machine learning, but also about the typical workflows and challenges encountered during the practical implementation of these kinds of systems. The usage of commonly used libraries and frameworks has enabled me to experience development practices used in industry: developing with frameworks and libraries requires additional skills, often surpassing the ones acquired from developing a project from scratch—for example, researching extensive documentation and learning through experimentation.

Understanding how to build a complex system in `TensorFlow` had a steep learning curve. The implementation required more advanced insight into the computation model of `TensorFlow` than I had initially (and perhaps naïvely) thought. The project has also taught me how to work in a scenario that involves big data: waiting for files to download, scripts to run and CNNs to train[1] has been a true test of patience, but has only strengthened my abilities as a computer scientist. Testing through a combination of unit tests and integration tests has been of great importance, in the same way that planning has.

The training of deep convolutional neural networks requires careful planning; if I were to do the project again, I would allocate *much* more time to training and implementation. One difficult problem encountered during the past year has been training the CNNs for days and not obtaining the expected results: the process of implementation and training should not be underestimated. Finally, I feel that my academic writing skills have significantly improved and further developed my ability to describe theoretical and practical concepts concisely—a vital skill for computer scientists.

5.3 Further work

As concluded in the Evaluation chapter, the fully-convolutional *k2c2* model was the best-performing architecture, confirming the findings of Choi et al. [8]. Further research into fully-convolutional networks and even recurrent neural networks has been undertaken to further improve the music tagging capabilities of these systems. Therefore, other models (possibly incorporating recurrent mechanisms) can be implemented and their performance compared to the ones already described in the previous chapter. Moreover, training for the existing architectures could be resumed from the existing checkpoints to see whether the results I have reported can be improved.

Research in music recommendation is thriving as more and more users are seeking music recommendation services. As in other information retrieval tasks, adding a form of relevance feedback to this process would improve the system even more. Having the ability to monitor the "success" of recommendations, based on the duration of a song that users listen to or a rating given by the user, would further strengthen the quality of recommendations.

[1] Training was done for 150 epochs, which equates to roughly three days for *k2c2*.

Bibliography

[1] Cs231n: Convolutional Neural Networks for Visual Recognition, 2018. http://cs231n.stanford.edu/.

[2] I. Sutskever A. Krizhevsky and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks,. *Advances in Neural Information Processing Systems*, 25, 2012.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-Scale Machine Learning on Heterogeneous Distributed systems. *CoRR*, abs/1603.04467, 2016.

[4] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. *CoRR*, abs/1206.5533, 2012.

[5] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The Million Song Dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[7] Keunwoo Choi, George Fazekas, and Mark B. Sandler. Automatic tagging using deep convolutional neural networks. *CoRR*, abs/1606.00298, 2016.

[8] Keunwoo Choi, George Fazekas, Mark B. Sandler, and Kyunghyun Cho. Convolutional Recurrent Neural Networks for Music Classification. *CoRR*, abs/1609.04243, 2016.

[9] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (elus). *CoRR*, abs/1511.07289, 2015.

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

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[12] Sean Holden. Machine Learning and Bayesian Inference. *University of Cambridge Part II CST lecture notes*, 2017-2018.

[13] Rajesh Ranganath Honglak Lee, Roger Grosse and Andrew Y. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations,. *Proceedings of the 26th Annual International Conference on Machine Learning, New York, NY, USA, ICML '09*, page 609–616, 2012.

[14] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*, abs/1502.03167, 2015.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014.

[16] Tara N. Sainath, Abdel rahman Mohamed, Brian Kingsbury, and Bhuvana Ramabhadran. Deep convolutional neural networks for LVCSR. In *ICASSP*, pages 8614–8618. IEEE, 2013.

[17] S. S. Stevens. A Scale for the Measurement of the Psychological Magnitude Pitch. *Acoustical Society of America Journal*, 8:185, 1937.

[18] George Tzanetakis and Perry Cook. Musical genre classification of audio signals. ieee Trans Speech Audio Process. 10:293 – 302, 08 2002.

[19] Aaron van den Oord, Sander Dieleman, and Benjamin Schrauwen. Deep content-based music recommendation. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2643–2651. Curran Associates, Inc., 2013.

[20] Laurens van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[21] R. Vergin, D. O'Shaughnessy, and A. Farhat. Generalized mel frequency cepstral coefficients for large-vocabulary speaker-independent continuous-speech recognition. *IEEE Transactions on Speech and Audio Processing*, 7(5):525–532, Sep 1999.

[22] D. Madhavan W. Mirowski, Y. LeCun and R. Kuzniecky. Comparing svm and convolutional networks for epileptic seizure prediction from intracranial eeg,. *Machine Learning for Signal Processing*, page 244–249, 2008.

Appendix A

Project Proposal

University of Cambridge

Part II Project Proposal

Andrew Wells (aw684) - Queens' College

Deep Learning for Music Recommendation

Name:
Andrew Wells

Supervisor:
Catalina Cangea

Overseers:
Dr Hatice Gunes
Dr Robert Watson

Director of Studies:
Dr Alastair Beresford

October 19, 2017

Introduction and Description

With the influx of users moving towards music-streaming services such as Spotify, Apple Music, and Pandora, content providers are now, more than ever, pushing the boundaries of music classification and recommendation. Features like Discover Weekly on Spotify and Apple's automated song suggestions are just two examples of this fascinating area of Computer Science. A plethora of Machine Learning techniques have been and are used for classifying songs by genre, mood, instrument and era. Many content-rich, data-driven music-streaming applications have fundamentally changed how users listen to and discover new music.

Collaborative filtering [4] is one of the methods used for recommending new songs and artists and is based on historical usage data, clustering users together based on the songs they've listened to. At first sight, this seems like a perfectly suitable method of recommending songs, but it ignores new songs, new artists and bands that haven't yet been listened to frequently. Music discovery should be oriented towards giving users the best possible recommendations, a possible approach to this being to analyse the audio signal itself.

Many Music Information Retrieval Systems have a two-stage approach, starting with feature extraction from audio signals, and subsequently using these features as input to a classifier or regressor such as logistic regression or support vector machines. The problem with this approach, however, is that *any* audio signal must first be analysed using complex DSP techniques to find such features.

This project will explore the use of CNNs for music tagging, which requires only an audio spectrogram of the audio signal, and as such vastly reduces the time spent on pre-processing. I will be generating spectrograms of the audio files, and using these as input to a Convolutional Neural Network (CNN), a model which has been actively used in a number of music classification tasks. Spectrograms will be provided as input because CNNs are in general well-suited to extracting features from images, and they provide perhaps the best representation of the audio signal for CNN architectures.

The CNN architecture I have chosen to implement [3] consists of 5 convolutional layers followed by 2 fully-connected layers. The output of the convolutional layers are fed into the fully connected layers, which act as the classifier, outputting the probability that each of the top-50 tags can be associated with the song. Sigmoid functions are used as activation at output nodes to facilitate multi-label classification.

Starting Point

I will be starting my Part II project with the following prior experience and knowledge:

- basic machine learning concepts from the Part IB Artificial Intelligence course
- basic knowledge of Python language

The project will thus require that I research and explore the following:

- the theory and background to ANNs with a focus on CNNs
- Python programming for TensorFlow's API (the only version that has stability guarantees)
- content from the Digital Signal Processing course (Michaelmas 2017) on spectrograms of audio-signals
- audio signal processing frameworks

Project Structure and Substance

Key concepts

The project will involve substantial research and knowledge acquisition in the theory of neural networks, an understanding of CNNs for music classification, and the network architectures that can be used. In addition, a ground-level knowledge of Digital Signal Processing techniques will be needed to understand the implementation of pre-existing libraries for audio-signal processing, which will be used in generating the spectrograms.

Major work items

My project is to build a music classification system using CNNs. This will involve classifying songs according to the top-50 relevant tags, acquired from the *last.fm* subset of the Million Song Dataset [2]. Sample audio will be fetched from services such as *7Digital*. The tags include genre, mood, era and instruments. The classification system (CS) will be implemented through the use of Convolutional Neural Networks which will use the 2*D* kernel, 1*D* convolution model *k2c1* described in the paper [3]. This model has been chosen specifically as it is motivated by structures for music tagging and genre classification.

The CNN system I will be implementing takes as input a spectrogram of an audio clip and outputs the probability that each of the top-50 tags is associated with the song.

The implementation of the CNN will require extensive knowledge of the TensorFlow API. It was decided that the implementation language be Python,

as TensorFlow only offers API stability guarantees for Python. The training and testing of the CNN will require use of a GPU, which will be obtained from the Computational Biology Group at the Computer Laboratory.

It is worth noting that the substantial part of the project consists of the CNN implementation, the choice of network architecture, and the training of the network. The recommendation system itself relies purely on the output of the CNN, so it is essential that the CNN is tested and evaluated before a recommendation system is built.

Using the top-50 tags for songs in the Million Song Dataset, a t-SNE plot can be made using songs in the dataset, to evaluate the potential of the system for clustering similar songs and being part of a recommendation system.

t-distributed stochastic neighbour embedding (t-SNE) is a machine learning algorithm for dimensionality reduction, which will project the top-50 tags of a song into a 2D space. The more similar two songs are, the closer they are to one another in some n-dimensional space. t-SNE preserves distances between points in higher dimensional space, so those points that are close together (i.e. similar songs) in a higher dimension will map to points that are close together in 2D space. The clustering of songs in a 2D space would then allow recommendations to be made, based upon a simple distance metric between points.

Methods of Evaluation

The implementation can be evaluated using the last.fm tags included in the Million Song Dataset. The majority of the evaluation will be purely quantitative; I have chosen to use AUC-ROC, as it is widely used for classification problems and will allow a direct comparison between the implementation found in [3] and my own.

AUC-ROC (Area Under the Receiver Operating Characteristic curve) is a metric actively used for assessing the performance of classifiers. Counting false-positives and true-positives will indicate incorrectly generated tags and correctly generated tags respectively. To allow for a comparison of performance between my implementation and that of [3], I will be reproducing the results for AUC-ROC per tag, and then averaging the AUC-ROC over all tags to obtain a single measure for the classifier's performance. Using this metric will allow a comparison between my implementation and the paper's implementation of the CNN.

For all models, evaluation will be carried out using the holdout method, which splits the dataset into training and test partitions, training on the former and testing on the latter.

Success Criteria

The core project will be considered successful upon completing the following:

1. A working implementation of a Convolutional Neural Network
2. The ability of the network to perform better-than-chance prediction (accuracy above 50% for each tag and the lower bound of its confidence interval above 50%) of the top-50 tags for a given audio file.

Extensions

Should I complete the core objectives, I will aim to do the following, in order of preference:

1. Analyse the high-level representations learned by the model by producing t-SNE plots for various representative subsets of the songs in the dataset, and explore its use in music recommendation.
2. Implement other models in the paper [3] to improve the performance of the system and enable further opportunity for comparison between my implementation and that of the paper
3. Implement an attention mechanism [1] for the CRNN model above to improve results even further

Project Plan

Week 0 (21st October—27th October)
Begin researching the theory behind convolutional neural networks, and the theory behind DSP techniques for spectrograms.

Weeks 1 & 2 (28th October—10th November)
Continue researching the theoretical aspects, and look at libraries to be used for the audio-signal processing.
Milestone: The theory topics have been covered and can be applied to the project.

Weeks 3 & 4 (11th November—24th November)
Start writing code snippets and gain familiarity with the libraries to be used. Implement a simple ANN with TensorFlow, and learn about how CNNs are implemented with TensorFlow.
Milestone: Familiar with the libraries and implemented the audio-signal pre-processing module.

Weeks 5 & 6 (25th November—8th December)
Implement the CNN
Milestone: A working implementation of the CNN has been completed.

Weeks 7 & 8 (9th December—22nd December)
Slack time for delays and unexpected issues with the CNN implementation. Can begin training and testing the CNN using the Million Song Dataset.

Weeks 9 & 10 (23rd December–5th January)
Finish the training and testing of the network's performance.
Milestone: The network provides a better-than-chance prediction of the top-50 tags for a given song.

Weeks 11 & 12 (6th January—19th January)
Perform hyperparameter optimisation to improve the accuracy of the network. Start implementing other models.
Milestone: Completion of training and testing. The CNN now has improved accuracy.

Weeks 13 & 14 (20th January—2nd February)
Additional slack time for delays in training and testing. Begin to write progress report and evaluate the project.
Milestone: Progress report finished and submitted before 2nd February. Completed implementation of other models.

Weeks 15 & 16 (3rd February—16th February)
Revise the evaluation based on feedback, and write the presentation. Continue testing other models.

Weeks 17 & 18 (17th February—2nd March)
Analyse the potential of the system for recommendation, using t-SNE.
Milestone: t-SNE plots are finalised. Evaluation of other models completed.

Weeks 19 & 20 (3rd March—16th March)
Start writing up the dissertation, focussing on the Introduction chapter.
Milestone: Introduction chapter has been completed.

Weeks 21 & 22 (17th March—30th March)
Write up the Preparation and Implementation chapters of the dissertation.
Milestone: Preparation and Implementation chapters have been completed.

Weeks 23 & 24 (31st March—13th April)
Write up the Evaluation and Conclusion chapters of the dissertation. Amend any changes suggested by my supervisor and/or DoS.
Milestone: The Introduction, Preparation, Implementation, Evaluation and Conclusion chapters of the dissertation are completed and ready for submission to supervisor.

Weeks 25 & 26 (14th April—27th April)
Send dissertation to supervisor by 16th April and amend any changes from comments received.

Weeks 27 & 28 (28th April—11th May)
Slack time for changes to write-up.
Milestone: Dissertation has been completed and submitted by 11th May to my DoS for approval.

Week 29 (12th May—18th May)
Slack time for any final comments received from my DoS.
Milestone: Final Dissertation submitted before 16th May.

Resource Declaration

My Macbook Pro (2.5 GHz Intel Core i7, 16 GB 1600 MHz DDR3 RAM, AMD Radeon R9 M370X 2048 MB) running macOS 10.12 Beta (16A313a) will be used for the project implementation and initial testing. Should the training take too long, I will use a GPU acquired from the department. I accept full responsibility for the equipment used and will use MCS machines should failure arise.

Regular back-ups of the project data will be undertaken, with the project implementation being stored locally, on-disk with usage of BitBucket which provides free private repositories and git for version control. The write-up will be synced with iCloud Drive and also be stored in a repository on BitBucket, using git for version control.

References

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". In: *arXiv* (2014).

[2] Thierry Bertin-Mahieux et al. "The Million Song Dataset". In: *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*. 2011.

[3] K. Choi et al. "Convolutional Recurrent Neural Networks for Music Classification". In: (2016). URL: https://arxiv.org/pdf/1609.04243.pdf.

[4] Sander Dieleman. "Recommending music on Spotify with deep learning". In: (2014). URL: http://benanne.github.io/2014/08/05/spotify-cnns.html.