Catalina Cangea

Music Genre Classification using Convolutional Neural Networks

Computer Science Tripos, Part II

Murray Edwards College

11th May 2016

Proforma

Name:	**Catalina Cangea**
College:	**Murray Edwards College**
Project Title:	**Music Genre Classification using Convolutional Neural Networks**
Examination:	**Computer Science Tripos, Part II, June 2016**
Word Count:	11206
Project Originator:	Catalina Cangea
Supervisor:	Dr Sean Holden

Original aims of the project

The original aim of the project was to implement a system which uses a convolutional neural network to classify musical genres. The system should have CUDA implementations for the operations of neural network layers and be able to run them on GPUs. The input to the neural network should consist of the spectrograms obtained from audio files. In addition, the system should achieve a better-than-chance classification rate on two genres.

Work completed

The core objectives described above have been successfully completed. In addition, several extensions have been achieved, such as an improved accuracy for two genres, a better-than-chance prediction rate on four and six genres and an analysis of the filters from convolutional layers of the network.

Special difficulties

None.

Declaration

I, Catalina Cangea of Murray Edwards 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

Contents

List of Figures

Acknowledgements

This project was outlined, implemented and finalised with the helpful guidance received from Dr Sean Holden and the support offered by Dr David Chisnall. The evaluation of the project also contains results obtained with the help of my colleague, Mr Profir-Petru Partachi, who kindly provided an additional NVIDIA GPU.

Chapter 1

Introduction

In this chapter, I present the motivation for using machine learning algorithms in music information retrieval tasks, related work that has been carried out in this field and the reason for choosing to employ a convolutional neural network for music genre classification.

1.1 Motivation

Online music streaming and recommendation systems have seen a substantial user growth during the past few years. One of the most popular such services, Spotify, announced in 2015 that they had reached more than 75 million active users [11]. Given such a significant shift from traditional media—radio, television—to online streaming, the question of whether it is possible to learn from the available data and provide users with better recommendations naturally arises.

One of the strategies Spotify uses for suggesting music is collaborative filtering. The songs are encoded into a vector space represented by factors which influence listening preferences. By learning patterns from other users' listening history, the vector-space similarity between music tracks can be computed and used to provide suggestions. However, this approach makes it impossible for recent songs with no usage data to be recommended. More crucially, no analysis is performed on the audio content itself, which represents a strong indicator of users' musical preferences.

In 2014, Sander Dieleman brought a valuable improvement to Spotify's recommendation system [1]. He used the audio content of music tracks to train a deep neural network which learned the vector-space factors. The network could then predict vector values for new songs, allowing the similarity to be computed between any two tracks, regardless of how much user data was available. Dieleman's work addressed both issues: accounting for new songs and using the audio content to directly influence the modelling of the vector space.

This deep learning approach serves as a real-world example of the powerful enrichments

machine learning can bring to music information retrieval applications. My project focuses on genre classification, a closely-related problem in the context of music recommendation systems. Users are highly likely to base their listening activity on genre preferences, so classification plays a vital role in providing suggestions.

1.2 Supervised learning

A classification task in machine learning might be approached in a *supervised* manner. This requires a dataset which contains examples of the form $(\mathbf{x}, y)$, where $\mathbf{x}$ is the input to the classifier and y represents the class that $\mathbf{x}$ belongs to. A part of the dataset is typically used for *training* the classifier to predict the correct class of the input. Providing the classifier with the examples $(\mathbf{x}, y)$ allows it to adjust its parameters such that it can learn an increasingly accurate representation of the training data. However, the goal of supervised learning is a good *generalisation* ability (prediction of the correct class for inputs which were not seen during training). This may not be obtained if the classifier *overfits* (learns the training data too well), so its classification performance on unseen data needs to be monitored during the training phase.

1.3 Related work on music genre classification

During the early 2000s, online music collections were becoming large enough such that the idea of automatically structuring the content according to genre seemed increasingly attractive. In 2002, Tzanetakis et al. [12] published the work done and results obtained from one of the earliest explorations involving automatic genre classification. They proposed three sets of features to be extracted from the audio signal, concerning timbral texture, rhythmic content and pitch content. Extracting them assumes knowledge of digital signal processing techniques; the following are examples of features from the timbral texture category:

1. *Spectral Centroid*: the center of gravity of the magnitude spectrum of the Short-time Fourier Transform:

$$C_t = \frac{\sum_{n=1}^{N} M_t[n] * n}{\sum_{n=1}^{N} M_t[n]}$$

   where $M_t[n]$ is the magnitude of the Fourier transform at frame t and frequency bin n.

2. *Spectral Rolloff*: the frequency R_t below which 85% of the magnitude distribution is concentrated:

$$\sum_{n=1}^{R_t} M_t[n] = 0.85 * \sum_{n=1}^{N} M_t[n].$$

3. *Spectral Flux*: the squared difference between normalised magnitudes of successive spectral distributions:

$$F_t = \sum_{n=1}^{N} (N_t[n] - N_{t-1}[n])^2$$

where $N_t[n]$ and $N_{t-1}[n]$ are the normalised magnitudes of the Fourier transform at frames t and $t-1$, respectively.

Features concerning rhythmic content are used to detect beats in the sound signals and require more involved processing. First, the signal is decomposed into octave frequency bands by applying the discrete wavelet transform (DWT). Additional steps follow: computing the temporal envelope of the signal, low-pass filtering, downsampling, mean removal and enhanced autocorrelation. Finally, a beat histogram is created from the autocorrelation sequence and analysed to extract beat information.

The proposed features were extracted from the GTZAN dataset which consists of 1000 tracks equally split into 10 genres. Afterwards, GTZAN was made publicly available—it is the most popular dataset [10] used in music genre recognition experiments, including the project described in this dissertation.

Tzanetakis et al. evaluated the performance of several classification methods on their feature set: Gaussian classifier, Gaussian mixture model and k-Nearest neighbours. They obtained a maximum accuracy of 61% on 10 genres, which was presented as comparable to the human classification ability, given the blurred and subjective division between different genres.

In addition to the methods used by Tzanetakis et al., other supervised machine learning algorithms have been used to perform music genre classification. They include support vector machines, artificial neural networks and hidden Markov models [9]. However, all of these methods use sets of features obtained from the data. The results are therefore dependent on the manner in which the initial data is summarised by the feature extraction phase.

More recently, deep learning methods have been used to learn features directly from the audio signal. Van den Oord et al. [13] used audio clips to train a convolutional neural network which learned the underlying representation consisting of factors which influence listening preferences.

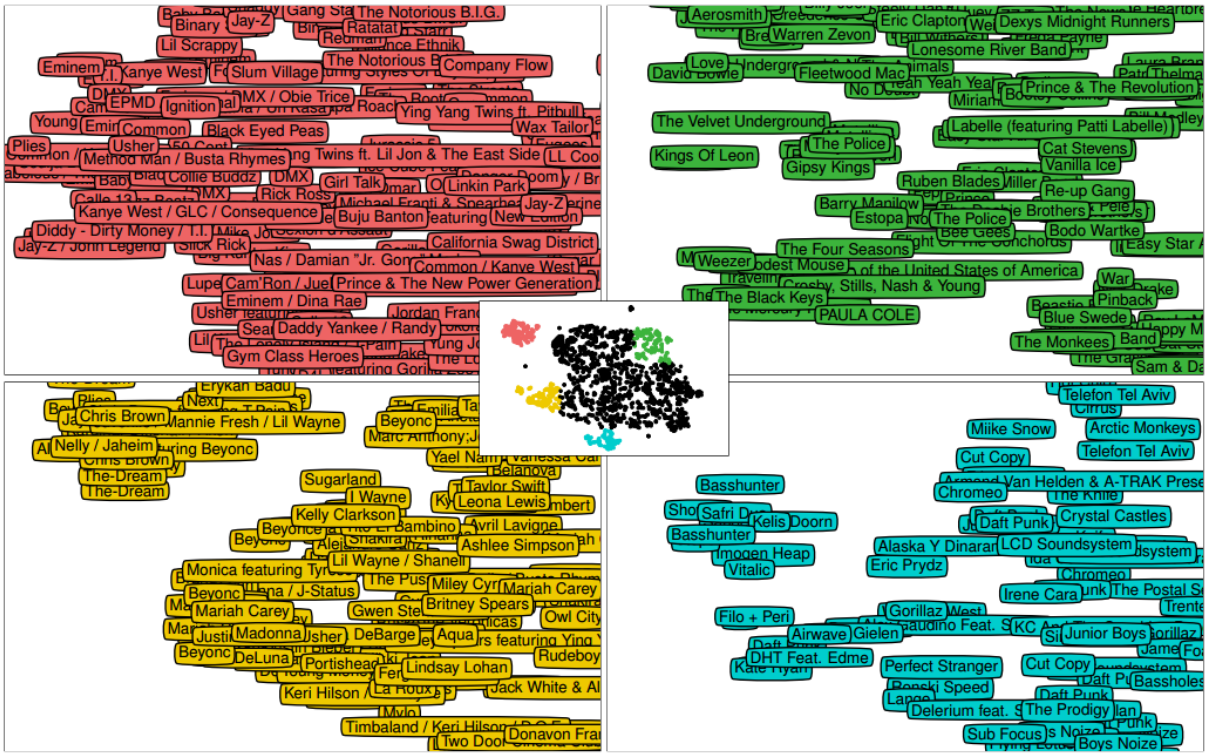


Figure 1.1: Clusters of artists preferred by the same audiences obtained from the latent factors learned. Reproduced from [13].

1.4 Convolutional neural networks (ConvNets)

As presented in the previous section, a feature-based classification approach requires extensive knowledge of digital signal processing techniques, sound characteristics and their relevance for distinguishing music genres. The major advantage of a convolutional neural network over other classification methods—*and my reason for choosing to implement one for this project*—is that the feature design and extraction phase is no longer needed. A ConvNet is a supervised learning method which has the ability to learn features by itself from the input data and use them for classification. Furthermore, the ConvNet might discover relevant features which humans may not consider, or be able to model in a suitable way, for extraction.

Convolutional neural networks [2] are based on neuroscientific principles, drawing inspiration from the visual cortex, and represent a type of artificial neural network (ANN). To perform classification, a regular ANN receives an input, repeatedly transforms it through one or more hidden layers and finally provides the probabilities of the input belonging to each class through the output layer. Hidden layers contain sets of neurons, each of them being fully connected to all neurons in the previous layer.

A ConvNet follows a similar strategy and additionally exploits the assumption that the input is an image. This motivates designing the connectivity of a special kind of layer—the *convolutional* layer—in a manner which allows learning the spatially-local correlations in

the image. In order to discover a particular local pattern, a neuron is no longer connected to all neurons in the previous layer, but only to a subset of adjacent ones. Moreover, its connectivity parameters are shared by all other neurons in the same layer, so a feature will be found regardless of its position in the image.

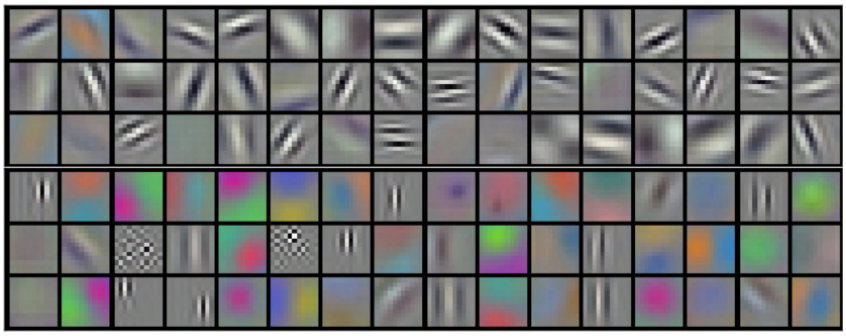


Figure 1.2: Image features learned by a convolutional layer. Reproduced from [4].

This type of neural network can also be used to classify audio content, if the sound file is first converted to a suitable image representation. Spectrograms constitute a good choice, as they plot the frequency spectrum information of the sound signal against time, thus allowing the ConvNet to learn features directly from input data.

The following chapters will describe the theoretical background and implementation of convolutional neural networks. I include discussions about the software engineering approach adopted, the architecture of the classification system and NVIDIA's CUDA programming model used for parallelising ConvNet layer computations on graphics processing units (GPUs).

Chapter 2

Preparation

In this chapter I present the project requirements, methodology used and possible risks. Following this, I describe the theory learned before starting the implementation and provide an overview of the CUDA programming model used to speed up some parts of the ConvNet computations.

2.1 Starting point

Before beginning to work on the project, I only had:

- theoretical knowledge of artificial neural networks from the *Artificial Intelligence I* Part IB course;
- basic theoretical knowledge of the CUDA programming model from the *Computer Design* Part IB course;
- programming experience with C acquired during high-school and the *Programming in C and C++* Part IB course;
- no experience with the Python programming language.

The following additional skills and knowledge had to be acquired throughout the course of the project:

- familiarity with the Python language and libraries used for implementation;
- practical experience with CUDA required to implement parallelisation of some ConvNet operations;
- the theory of convolutional neural networks;
- advanced machine learning concepts from the *Artificial Intelligence II* Part II course, including the relevant evaluation methods;
- audio processing concepts from the *Digital Signal Processing* Part II course.

2.2 Requirements analysis

This project had the following requirements, according to the objectives I initially proposed (Appendix A):

1. A classification system based on a convolutional neural network, containing the following components:
   (a) Implementations for all types of network layers required by a ConvNet (convolutional, max pooling, global pooling, fully-connected, activation, softmax).
   (b) Implementation of a ConvNet classifier which can be initialised with a given architecture (ordered set of layers), perform training, testing and prediction on input data.
2. Implementation of a data provider which can load a dataset consisting of spectrograms and provide them as training and test data to the classification system.
3. Implementations of ConvNet layer operations to run on GPUs by using the CUDA parallel computing platform.

2.3 Workflow

2.3.1 Software development methodology

I used the *Iterative Development Model* throughout the course of the project, the choice being motivated by the modular nature of the classification system architecture. This allowed flexible development and testing of smaller components first (for example, classes corresponding to different types of network layers), in order to ensure that they worked as required before integrating them with the system.

In addition, I started the implementation by creating interfaces for all classes, which meant that any component could be developed in isolation by making use of its dependencies, even if some of them were not yet fully functional. Moreover, I implemented the CUDA optimisations for layer computations only after writing the same operations in Python. The latter took less time and allowed me to verify that I had understood and applied the theory correctly.

2.3.2 Version control system and backup strategy

Version control was achieved by maintaining a `git` repository on my laptop and pushing every commit to the online repository on my *GitHub* account. This facilitated rolling back to a previous version of the project or comparing two versions of the same component, in case a bug would be discovered in the latest version. Regular backups of the project code, dissertation LaTeX source files and other resources were also made to my *Dropbox* account, MCS machine and an external hard-disk.

2.3.3 Programming language and development environment

I found the Python programming language the most suitable for this project for several reasons:

- Fully-documented libraries written in Python could be nicely integrated with different parts of the project (more details given in the Libraries sub-section).
- Python has support for object-oriented programming, so I could design components of the system while applying principles such as encapsulation and inheritance.
- Operations on multi-dimensional arrays can be written in a natural, mathematical manner and result in shorter code, while allowing greater focus on the higher-level functionality being implemented.

The development environment used for this project is PyCharm for Linux, Community Edition 5.0.1. This IDE offers useful development tools such as Python debugging, a testing framework, an integrated Python console and assistance for coding, analysis and refactoring.

2.3.4 Libraries

The following libraries were used in the project:

- `librosa`, which contains methods for extracting spectrograms from audio files. This meant that the Short-time Fourier Transform and other signal processing techniques did not require implementation from scratch and allowed me to focus on the classification system;
- `PyCUDA`, which allowed me to write functions for processing data on the GPU through NVIDIA's CUDA API while providing abstractions for transferring data to and from the GPU and useful features such as automatic object clean-up and CUDA error checking;
- `NumPy`, which facilitates working with multi-dimensional arrays and offers implementations of high-level functions to operate on them;
- `matplotlib`, which helped me plot graphs for the Evaluation chapter.

2.4 Risk analysis

The GTZAN dataset that I am using is the largest publicly-available music dataset, but it only consists of 1000 audio files equally distributed across 10 genres. The deep learning method adopted in this project involves training a model with many parameters—if not enough training data is available, the classifier might be unable to generalise for new examples. Therefore, the use of this dataset represents a significant risk which could affect the classification performance of the system.

2.5 Theoretical background

2.5.1 Deep feedforward networks

A *deep feedforward network* [2] represents a machine learning method which aims to learn high-level abstractions in data by modelling them as functions. These functions become increasingly complex as more layers are added and as the number of units in a layer is increased. In the context of a classification problem, the function f learned by the network places the input $\mathbf{x}$ in a category y. Hence, the mapping is defined as $y = f(\mathbf{x}; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ represents the parameters which the network learns during training.

Also called *forward-propagation*, the computation of the network is *feedforward* because it starts at the input $\mathbf{x}$, flows through the layers of the network and reaches the final layer which outputs the result $f(\mathbf{x}; \boldsymbol{\theta})$. The *depth* of the network is given by the number of layers, n. Each layer might be thought of as computing a function f_i, such that $f = f_n \circ f_{n-1} \circ ... \circ f_2 \circ f_1$.

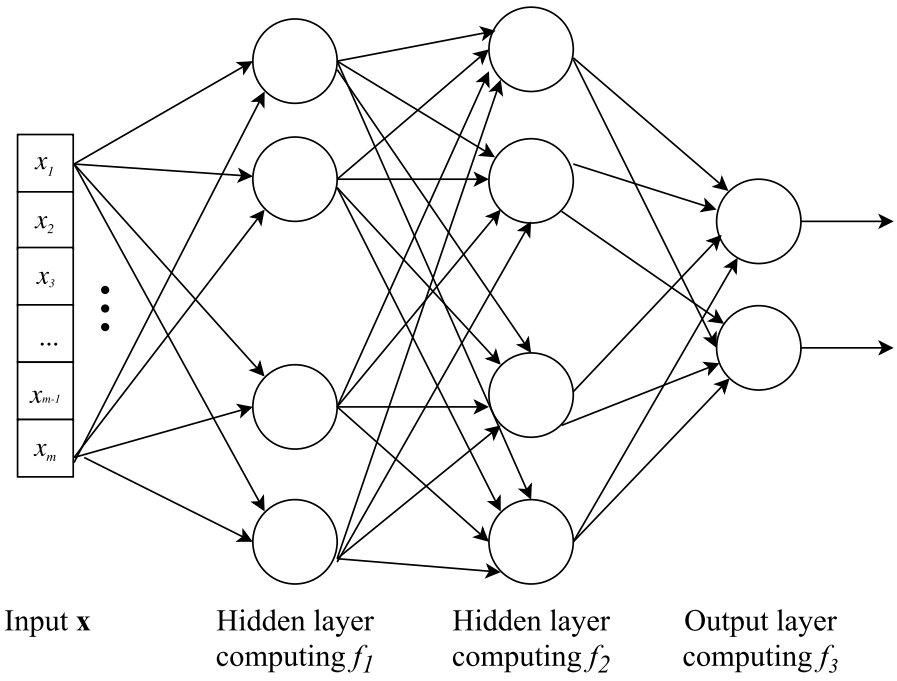


Figure 2.1: An example of a feedforward network with 2 hidden layers.

In a supervised learning approach, training the feedforward network in order to learn a function is achieved through *gradient descent*. This is an optimisation algorithm which takes small steps in the direction opposite to the function gradient at the current point. Gradient descent attempts to minimise the *cost function* which provides a measure of the distance between the computed output and the true output.

Maximum likelihood and cost function

Consider a training set of m independent and identically distributed examples $\mathbb{X} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(m)}\}$ drawn from a real distribution $p_{\text{real\_data}}$. Let $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$ be a family of distributions indexed by the parameters $\boldsymbol{\theta}$, where each distribution $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$ provides an estimation of the probability $p_{\text{real\_data}}$. The *maximum likelihood estimator* for $\boldsymbol{\theta}$ is then defined as:

$$\begin{aligned} \boldsymbol{\theta}_{ML} &= \arg\max_{\boldsymbol{\theta}} p_{\text{model}}(\mathbb{X}; \boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{\text{model}}(\mathbf{x}^{(i)}; \boldsymbol{\theta}). \end{aligned} \tag{2.1}$$

Taking the logarithm of the product in 2.1 achieves an equivalent result:

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\text{model}}(\mathbf{x}^{(i)}; \boldsymbol{\theta}). \tag{2.2}$$

Dividing by m, we obtain an estimate of the expectation of maximum likelihood with respect to the distribution of the training data:

$$\mathbb{E}_{\mathbf{x} \sim p_{\text{train\_data}}}[\log p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})], \tag{2.3}$$

where the notation $\mathbb{E}_{x \sim P(X)}[f(X)]$ indicates that the distribution of X is P. Consequently, the estimation method for parameters $\boldsymbol{\theta}$ based on maximum likelihood attempts to minimise the difference between the distribution of the training set, $p_{\text{train\_data}}$, and the model distribution, p_{model}.

A neural network estimates the *conditional probability* $P(\mathbf{y}|\mathbf{x}; \boldsymbol{\theta})$ in order to predict $\mathbf{y}$, given $\mathbf{x}$. This naturally extends the maximum likelihood approach—if $\mathbb{X}$ is the set of training examples and $\mathbb{Y}$ consists of the observed labels, then the conditional maximum likelihood estimator is:

$$\begin{aligned} \boldsymbol{\theta}_{ML} &= \arg\max_{\boldsymbol{\theta}} P(\mathbb{Y}|\mathbb{X}; \boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log P(\mathbf{y}^{(i)}|\mathbf{x}^{(i)}; \boldsymbol{\theta}). \end{aligned} \tag{2.4}$$

In the context of *regression*, if we assume the network model to be $p_{\text{model}}(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; f(\mathbf{x}; \boldsymbol{\theta}), \sigma^2 \mathbf{I})$, such that the function $f(\mathbf{x}; \boldsymbol{\theta})$ approximates the labels $\mathbf{y}$, the conditional log-likelihood is given by:

$$\sum_{i=1}^{m} \log p_{\text{model}}(\mathbf{y}^{(i)}|\mathbf{x}^{(i)}; \boldsymbol{\theta}) = -m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_{i=1}^{m} \frac{\|\mathbf{y}^{(i)} - f(\mathbf{x}^{(i)}; \boldsymbol{\theta})\|^2}{2\sigma^2}. \tag{2.5}$$

In 2.5, the first two terms represent constants, so we are only interested in maximising the quantity:

$$J(\boldsymbol{\theta}) = -\sum_{i=1}^{m} \frac{\|\mathbf{y}^{(i)} - f(\mathbf{x}^{(i)}; \boldsymbol{\theta})\|^2}{2\sigma^2}, \tag{2.6}$$

which represents the cost function that the neural network minimises. Ideally, we would be minimising the quantity in 2.7, but since we do not have access to *all* the values from the distribution of the training data, we approximate the quantity in 2.6 instead.

$$\frac{1}{2}\mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{train\_data}}}\|\mathbf{y} - f(\mathbf{x};\boldsymbol{\theta})\|^2 \tag{2.7}$$

Classification tasks minimise a different cost function called the *cross-entropy* loss function, which will be presented in the Implementation chapter.

During training, the quantity $f(\mathbf{x};\boldsymbol{\theta})$ is computed by forward-propagation and represents the output of the network for a given input $\mathbf{x}$. *Back-propagation* then enables the information from $J(\boldsymbol{\theta})$ to flow backwards through the network, such that the gradient is computed and used to minimise the cost $J(\boldsymbol{\theta})$ by gradient descent. I discuss the back-propagation algorithm in more detail in the next chapter, providing examples from my implementation.

Output layer for multiclass classification

A feedforward network layer computes unnormalised log probabilities as a function of its input $\mathbf{x}$:

$$\mathbf{z} = \mathbf{W}^T\mathbf{x} + \mathbf{b}, \tag{2.8}$$

where $z_i = \log \tilde{P}(y = i|\mathbf{x})$, $\mathbf{W}$ are the layer weights and $\mathbf{b}$ are the bias values.

However, a multiclass classification problem requires a valid probability distribution over n classes. We wish to represent this through a vector $\mathbf{y}$ for which $y_i = P(y = i|\mathbf{x})$, $y_i \in [0, 1]$ and $\sum_{i=1}^{n} y_i = 1$. These properties are achieved by the *softmax* function, which exponentiates $\mathbf{z}$ and then normalises the result to obtain a valid distribution $\mathbf{y}$ at the output of the neural network:

$$y_i = \text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{n}\exp(z_j)}. \tag{2.9}$$

2.5.2 Convolutional networks

Previously discussed in the Introduction chapter, convolutional neural networks are a type of neural network specialised for processing data with a grid structure (for example, image data, which may be seen as a 2D grid of pixels). ConvNets are based on the operation of *convolution* and replace the general matrix multiplication presented in 2.8 with convolution in at least one layer.

Convolution operation

Convolution is a commutative mathematical operation on two functions of a real-valued argument:

$$\begin{aligned}(f * g)(x) &= \int_{-\infty}^{\infty} f(x-y)g(y)dy \\ &= \int_{-\infty}^{\infty} f(x)g(x-y)dy.\end{aligned} \tag{2.10}$$

The equivalent operation in the discrete case is:

$$\begin{aligned}(f * g)(x) &= \sum_{y=-\infty}^{\infty} f(x-y)g(y) \\ &= \sum_{y=-\infty}^{\infty} f(x)g(x-y).\end{aligned} \tag{2.11}$$

The implementation of machine learning algorithms, including convolutional neural networks, uses discrete convolution. The first function is represented by the *input* and the second argument, by the *kernel*. In the context of processing images, the input is a 2D array of data and the kernel is a multidimensional array of parameters (*tensor*) which the algorithm learns in order to detect relevant features in the image. Consequently, the summation is made over a finite number of array elements. Moreover, convolutions can be performed over several axes. For example, a two-dimensional kernel $\mathbf{K}$ might be convolved with an input image $\mathbf{I}$ as follows:

$$(\mathbf{I} * \mathbf{K})_{i,j} = \sum_m \sum_n \mathbf{I}_{m,n}\mathbf{K}_{i-m,j-n}. \tag{2.12}$$

By the commutativity of convolution, this is equal to:

$$(\mathbf{I} * \mathbf{K})_{i,j} = \sum_m \sum_n \mathbf{I}_{i-m,j-n}\mathbf{K}_{m,n}. \tag{2.13}$$

In 2.13, the kernel indices increase while the image indices decrease, so the kernel is *flipped*. A related operation called *cross-correlation* achieves the same result as convolution, without flipping the kernel:

$$(\mathbf{I} * \mathbf{K})_{i,j} = \sum_m \sum_n \mathbf{I}_{i+m,j+n}\mathbf{K}_{m,n}. \tag{2.14}$$

A learning algorithm implements the version in 2.14 more naturally, as shown in Figure 2.2. The algorithm will learn the appropriate values for the kernel with respect to their position in the array, so the only difference in the final result is that the learned kernel will be flipped, while still detecting the same features.

Motivation for using convolution

Convolution allows the concepts of *sparse interactions*, *parameter sharing* and *equivariant representations* to be applied in a neural network to improve the performance of an

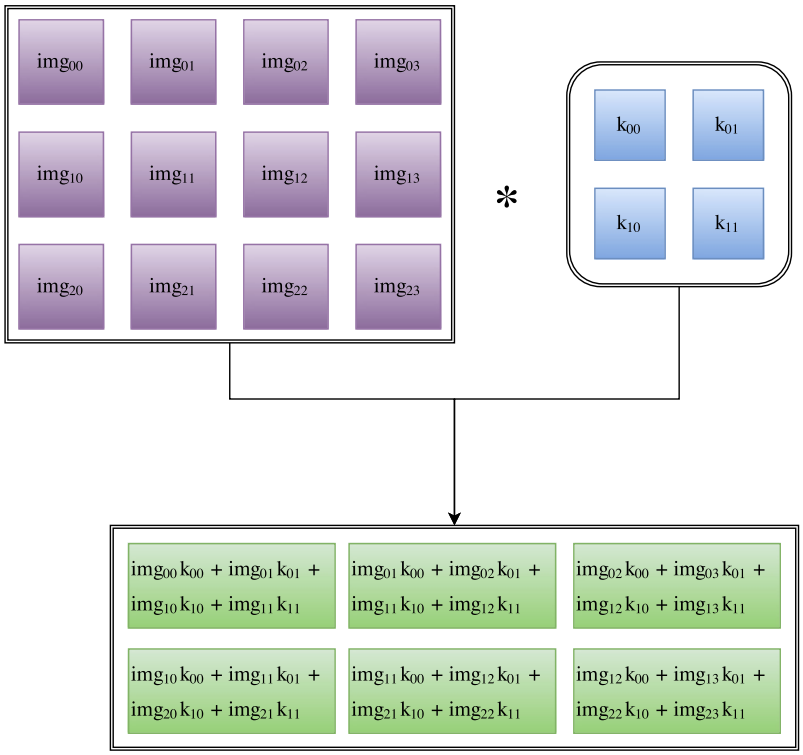


Figure 2.2: An example of two-dimensional convolution with a 2x2 kernel.

image classification algorithm.

Feedforward neural networks use matrix multiplication to represent the interaction between the input and output of a layer, so every output unit depends on each input unit. Convolutional layers have *sparse interactions* instead, by making the kernel smaller than the input. It is possible to detect relevant features such as edges by using a kernel whose area is only a small fraction of the image size.

Moreover, a kernel is moved across the entire input during convolution, so each of its elements interacts with the majority of the input pixels (or all of them, depending on whether the input is padded). Therefore, the parameters stored in kernels are shared by many input locations, as opposed to the parameters of a typical feedforward network layer. This manner of *parameter sharing* used by the convolution operation means that we are not required to learn a set of parameters for every input location anymore. Hence, fewer parameters can be stored, reducing the memory requirements and possibly improving the complexity of the model by learning more kernels.

In addition, the property of parameter sharing in a convolutional layer causes it to have *equivariance* to translation. A function f is equivariant if a change in x produces the same change in $f(x)$. Considering the case of convolution, if a function f shifts the initial input image and produces a new one, then the convolution function is equivariant to f. This means that the convolutional layer will still detect the same features in the shifted image and in the original one, regardless of their position.

Pooling

A typical section of a ConvNet consists of three layers:

1. a convolutional layer containing several kernels which produce linear outputs;
2. an *activation* layer which runs the result of the convolutional layer through a non-linear activation function (for example, the *rectified linear* function I use in my implementation and discuss in the next chapter);
3. a *pooling* layer which replaces the information in a certain region with a summarised representation (for example, max pooling outputs the largest value in that region).

Pooling achieves the effect of making the representation learned by the convolutional layer invariant to small translations in the input. Therefore, if certain features are slightly rotated or translated in an image, we will still obtain the same values for most of the pooling output.

2.6 CUDA programming model

The Compute Unified Device Architecture (CUDA) computing platform and API [7] were developed by NVIDIA such that programmers could use GPUs for parallelising general purpose computations. The framework allows a problem to be partitioned into smaller sub-problems which are then solved independently or cooperatively, depending on the level of parallelism being exploited.

2.6.1 CUDA kernels

A *kernel* represents a C function which runs on the GPU. The programmer can define a kernel and run it in parallel as many times as required, each time on a separate CUDA *thread*. Kernels are defined with the `__global__` specifier, which signifies that functions are called from *host* (CPU) code and run on the *device* (GPU). The following code represents an example of a CUDA kernel which can be used to compute the sum of two vectors:

```
__global__ void vector_sum(int* in1, int* in2, int* out) {
  out[threadIdx.x] = in1[threadIdx.x] + in2[threadIdx.x];
}
```

Listing 2.1: A CUDA kernel.

The built-in variable `threadIdx` provides a unique identifier for each thread. Upon calling the function, the programmer specifies the number of threads allocated for the computation through an *execution configuration* `<<<...>>>`. In the example below, the `vector_sum` kernel from Listing 2.1 is executed on `N` threads, performing pair-wise addition of vectors `a` and `b` (both of length `N`) and storing the result in `c`:

```
int main() {
  int *a, *b, *c;
  ...
  // Call kernel to run on N threads
  vector_sum<<<1, N>>>(a, b, c);
  ...
}
```

Listing 2.2: A kernel invocation.

2.6.2 Processing flow

A CUDA computation involves communication between the *host* and the *device*, according to the following model:

1. allocate the GPU memory required for the computation;
2. copy input data from CPU memory to GPU memory;
3. load and execute the GPU code written by the developer;
4. copy computation results back to CPU memory;
5. perform any required clean-up.

Therefore, the complete code for the kernel invocation is:

```
#define N 1024

int main() {
  int array_size = sizeof(int) * N;
  int *cpu_a, *cpu_b, *cpu_c; // CPU arrays
  int *a,     *b,     *c;     // GPU arrays

  // Allocate memory on CPU
  cpu_a = (int *)malloc(size);
  cpu_b = (int *)malloc(size);
  cpu_c = (int *)malloc(size);
  // Initialise cpu_a and cpu_b with desired input data
  ...

  // Step 1
  cudaMalloc((void **)&a, array_size);
  cudaMalloc((void **)&b, array_size);
  cudaMalloc((void **)&c, array_size);
  // Step 2
  cudaMemcpy(a, cpu_a, array_size, cudaMemcpyHostToDevice);
  cudaMemcpy(b, cpu_b, array_size, cudaMemcpyHostToDevice);
  // Step 3 - kernel call
  vector_sum<<<1, N>>>(a, b, c);
```

```
  // Step 4
  cudaMemcpy(cpu_c, c, array_size, cudaMemcpyDeviceToHost);
  // Use the results in cpu_c
  ...

  // Step 5
  free(cpu_a); free(cpu_b); free(cpu_c);
  cudaFree(a); cudaFree(b); cudaFree(c);

  return 0;
}
```

Listing 2.3: Data allocation and transfer associated with the kernel computation.

We notice that each time a kernel invocation is performed, there is some additional code required for data transfer and memory deallocation. The `PyCUDA` library handles this elegantly, as I will show in the Implementation chapter.

2.6.3 Parallelism: threads and blocks

In the example code from Listing 2.3, the execution configuration `<<<...>>>` takes two parameters. This is because a CUDA computation is usually divided into *blocks* consisting of *threads* (the example uses one block with `N` threads).

Threads may form a 1-, 2- or 3-dimensional *block* and are uniquely identified by `threadIdx`. The same applies to blocks, which can form a 1-, 2- or 3D *grid* and are identified by `blockIdx`. The built-in variables `threadIdx` and `blockIdx` are 3-component vectors which allow indexing into the arrays which the kernel accesses.

This organisation allows operations on vectors, matrices and volumes to be easily parallelised. Figure 2.3 shows a possible grid layout, where `Thread(x,y)` has `threadIdx` equal to `(x,y,1)` and `Block(x,y)` has `blockIdx` equal to `(x,y,1)`.

We can now index an element in an array using the two variables and a third one, `blockDim`, which provides the number of threads in a block, as in Listing 2.4:

```
__global__ void vector_sum(int* in1, int* in2, int* out) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  out[index] = in1[index] + in2[index];
}
```

Listing 2.4: Indexing in a 1D array.

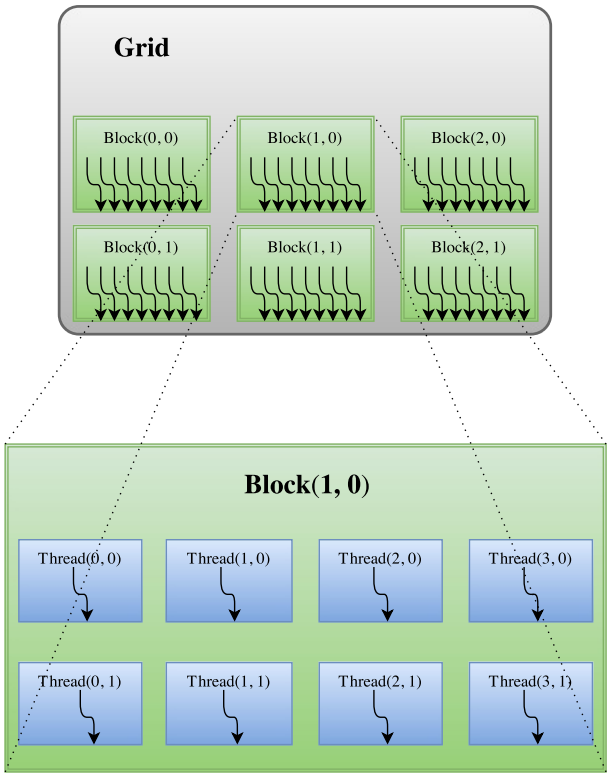


Figure 2.3: An example of a 2D grid consisting of 2D blocks.

2.6.4 Shared memory

We might sometimes like to perform a computation which involves all the elements inside a block; for example, replace each element of a `float` array with the average value of its block. In this case, operating within a single thread is not enough—we need to know the values of all elements inside the block first, in order to modify the current one.

CUDA solves this problem by allowing the programmer to allocate shared memory inside each block, such that all threads may access it:

```
__global__ void avg_inside_block(float* a) {
  // Allocate shared memory for each block
  __shared__ float temp[BLOCK_SIZE];

  int index = blockIdx.x * blockDim.x + threadIdx.x;
  // Store all values of the block in shared memory
  temp[threadIdx.x] = a[index];

  __syncthreads();

  float avg = 0.0;
  // Compute average of values inside block
  for (int i = 0; i < BLOCK_SIZE; ++i) {
    avg += temp[i];
  }
```

```
  // Replace current element with average
  a[index] = avg / (float)BLOCK_SIZE;
}
```

Listing 2.5: Working with shared memory.

The synchronisation function `__syncthreads()` prevents data hazards: it acts like a barrier which allows threads to resume execution only when all of them reach that point in the kernel. We need this to ensure that the shared memory contains all values in the block before computing their average.

2.7 Summary

In this chapter, I have presented the work completed before starting the implementation. This involved establishing the software engineering approach and tools to be used in the project, learning the underlying theory and familiarising myself with the CUDA programming model.

The Implementation chapter provides details about how I built a fully functional classification system by applying all of these concepts.

Chapter 3

Implementation

This chapter describes the implementation of the classes forming the classification system and the manner in which they interact while the classifier is learning the data model. I also explain some of the CUDA kernels which were written to run the forward- and back-propagation operations of neural network layers on GPUs.

3.1 Software UML

Figure 3.1 presents the overall architecture of the classification system derived from the project requirements. The `ConvNet` class contains high-level operations which could be implemented by only using the interfaces defined for the `DataProvider` class and the `Layer` abstract class. This allowed the different types of layers sub-classing `Layer` and the `DataProvider` class to be developed and tested independently.

The implementations of the abstract `Layer` class shown in the diagram represent computations which take place exclusively on the CPU. After these classes were completed and tested, the `forward_prop()` and `back_prop()` methods for each type of layer were overridden with CUDA versions in the corresponding subclasses `ActivationLayerCUDA`, `ConvLayerCUDA`, `FullyConnectedLayerCUDA`, `GlobalPoolingLayerCUDA` and `MaxPoolingLayerCUDA`.

The `SoftmaxLayer` class representing the output layer was not extended by a GPU version. The operations are performed on an array of length equal to the number of classes, which never exceeds ten, so the data transfer to and from the GPU alone would have taken orders of magnitude longer than the actual computation time.

3.2 Operation of the classification system

Performing classification for a given dataset starts with initialising a `ConvNet` object with an ordered list of `Layer` objects corresponding to the chosen network structure and a `DataProvider` instance. Listing 3.1 provides an example of the initialisation with the network architecture I used in the project; later on in this chapter, I give more

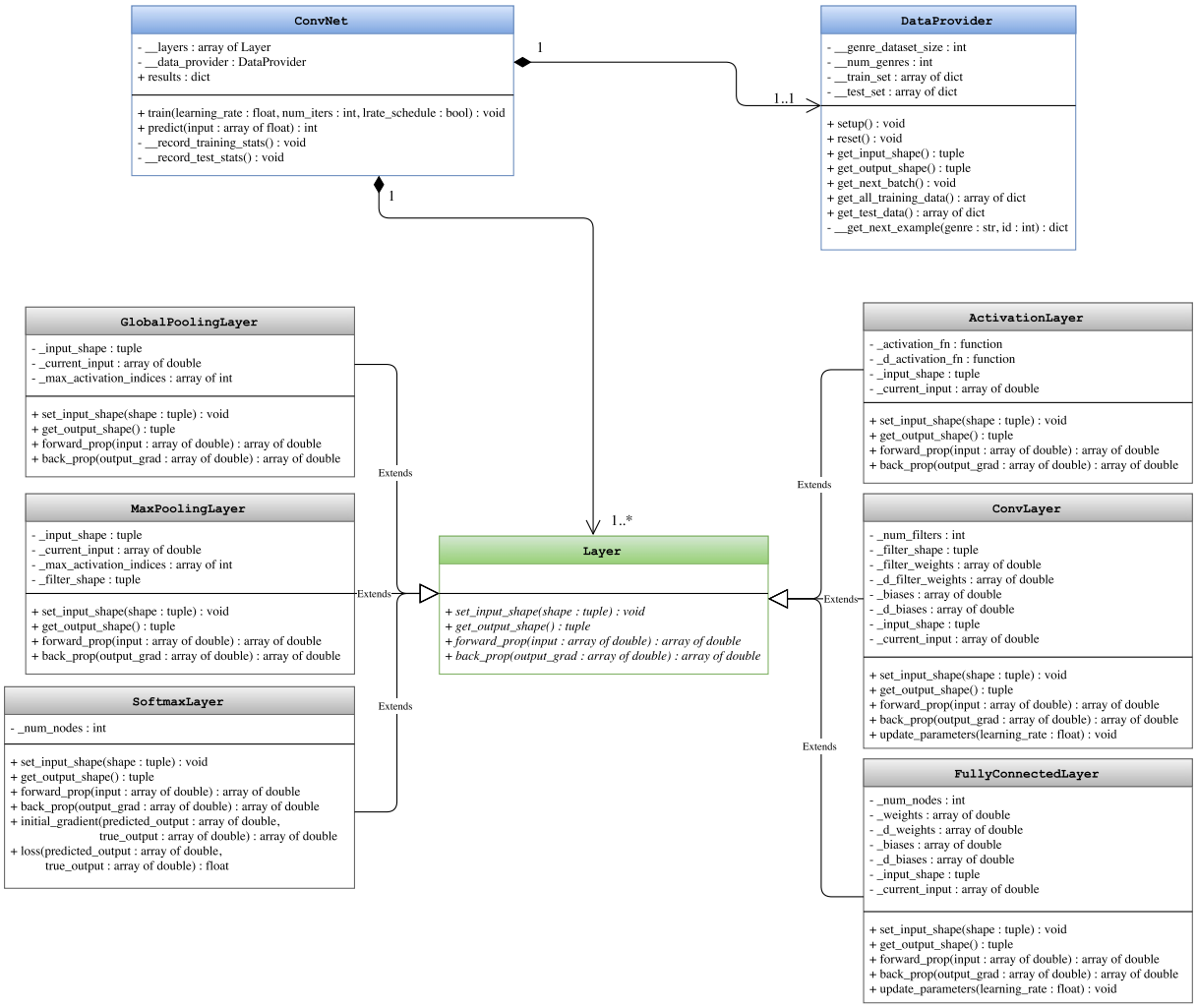


Figure 3.1: UML Class Diagram showing the architecture of the project.

details about the parameters required by each type of layer.

```
neural_net = ConvNet([ConvLayer(32, (128, 4), 0.044),
                      ActivationLayer('leakyReLU'),
                      MaxPoolingLayerCUDA((1, 4)),

                      ConvLayer(32, (32, 4), 0.088),
                      ActivationLayer('leakyReLU'),
                      MaxPoolingLayerCUDA((1, 2)),

                      ConvLayer(32, (32, 4), 0.088),
                      ActivationLayer('leakyReLU'),
                      GlobalPoolingLayer(),

                      FullyConnectedLayer(32, 0.125),
                      ActivationLayer('leakyReLU'),
                      FullyConnectedLayer(32, 0.125),
                      ActivationLayer('leakyReLU'),
                      FullyConnectedLayer(4, 0.17),
                      SoftmaxLayer()],
                     DataProvider(num_genres=4))
```

Listing 3.1: Initialising the convolutional neural network.

3.2.1 Convolutional neural network

In order for the neural network computations to take place correctly, the input size of each layer (except for the first one) must be equal to the output size of the previous layer. The method `setup_layers()` is called after the initialisation and checks that the given input and output shapes are consistent with the neural network architecture.

Algorithm 1 Setting up layers in the neural network.

procedure SETUP_LAYERS(*cnn_input_shape*, *cnn_output_shape*)
    current_shape ← *cnn_input_shape*
    for each *layer* in *layers* **do** ▷ Process the ordered list of layers
        layer.set_input_shape(current_shape)
        current_shape ← *layer*.get_output_shape()
    assert(current_shape = *cnn_output_shape*)

Following the setup phase, the neural network learns to classify inputs by undergoing a specified number of *training iterations*, according to Algorithm 2. Each iteration consists of performing forward- and back-propagation for every example in the training set. After each batch of examples has been processed, the network updates the parameters in the convolutional and fully-connected layers, using the gradients computed through back-propagation.

Algorithm 2 Neural network training.

```
procedure TRAIN(learning_rate, num_iters, lrate_schedule)
    for iter_id = 1 to num_iters do                          ▷ New training iteration
        while training set not fully processed do
            batch ← data_provider.get_next_batch()
            for each example in batch do
                input ← example.spectrogram                  ▷ Forward-propagation
                for each layer in layers do
                    input ← layer.forward_prop(input)
                gradient ← initial gradient from output layer ▷ Back-propagation
                for each layer in non-output layers from last to first do
                    gradient ← layer.back_prop(gradient)
            for each layer in layers do                      ▷ Update parameters
                if layer is convolutional or fully-connected then
                    layer.update_parameters(learning_rate)
```

After the training phase has finished, `ConvNet` evaluates the accuracy of the neural network on the test set which consists of *unseen data*, to observe how well the model has learned to generalise for new examples. The estimate of the classifier accuracy [3] is given by:

$$\hat{\text{err}}_{\mathbb{X}_{\text{test}}}(f) = \frac{1}{m}\sum_{i=1}^{m}\mathbb{I}(f(\mathbf{x}^{(i)}) \neq y_i), \tag{3.1}$$

where f represents the model learned by the network, $\mathbb{X}_{\text{test}}$ is the test set, the pair $(\mathbf{x}^{(i)}, y_i)$ represents a spectrogram input with its corresponding genre and

$$\mathbb{I}(x) = \begin{cases} 1 & \text{if } x = \text{true} \\ 0 & \text{if } x = \text{false.} \end{cases} \tag{3.2}$$

3.2.2 Optimisation of the model learned by the network

As discussed in the Preparation chapter, the neural network models a function $f(\mathbf{x}; \boldsymbol{\theta})$. We would like this function to approximate the real representation of the data as well as possible. Therefore, the optimisation process minimises the associated cost function $J(\boldsymbol{\theta})$, also known as the *objective function*. In this section, I discuss *stochastic gradient descent* and *back-propagation*, the two methods forming the optimisation process used by the classification system.

Stochastic gradient descent

The general gradient descent algorithm computes the *gradient* of $J(\boldsymbol{\theta})$ with respect to the network parameters, which gives the direction of the steepest increase of the objective function. In order to reach the minimum value of $J(\boldsymbol{\theta})$, we need to take steps in the opposite direction, so gradient descent updates the parameters as follows:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \epsilon\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}), \tag{3.3}$$

where ϵ represents the *learning rate*—a small positive value determining the size of the step in the direction of the negative gradient.

If the objective function is given by the negative conditional log-likelihood of the training data:

$$\begin{aligned} J(\boldsymbol{\theta}) &= \mathbb{E}_{\mathbb{X},\mathbf{y}\sim p_{\text{train\_data}}} L(\mathbb{X}, \mathbf{y}, \boldsymbol{\theta}) \\ &= -\mathbb{E}_{\mathbb{X},\mathbf{y}\sim p_{\text{train\_data}}} \log p(\mathbf{y}|\mathbb{X}; \boldsymbol{\theta}), \end{aligned} \tag{3.4}$$

then it may be written as the sum of errors per training example:

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\frac{1}{m} \log \prod_{i=1}^{m} p(y^{(i)}|\mathbf{x}^{(i)}; \boldsymbol{\theta}) \\ &= \frac{1}{m} \sum_{i=1}^{m} - \log p(y^{(i)}|\mathbf{x}^{(i)}; \boldsymbol{\theta}). \end{aligned} \tag{3.5}$$

Gradient descent computes the following quantity:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}), \tag{3.6}$$

which requires $O(m)$ computational time. Hence, a single update step involves processing the entire training set. A deep neural network (with many layers) takes a relatively large amount of time to cover all training examples, so I chose to speed up the optimisation process by using stochastic gradient descent.

Stochastic gradient descent (SGD) is a variant of the general algorithm which is based on the observation that the gradient computed over the whole training set represents an expectation. By sampling this set to obtain a small subset of training examples $\mathbb{B} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(m_b)}\}$, we can approximate the value of the gradient with the following unbiased estimate:

$$\mathbf{g} = \frac{1}{m_b} \sum_{i=1}^{m_b} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}), \tag{3.7}$$

which leads to updating the network parameters in the same manner as in gradient descent:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \epsilon \mathbf{g}. \tag{3.8}$$

Algorithm 3 SGD update for training iteration k.

procedure SGD(learning rate ϵ_k)
  $\mathbb{B} \leftarrow$ new batch sampled from the training set
  $\mathbf{g} \leftarrow \frac{1}{m_b} \sum_{i=1}^{m_b} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \mathbf{g}$

It can be noticed that the network training procedure described by Algorithm 2 incorporates SGD, including the steps in Algorithm 3 as follows:

1. The training set is sampled through: *batch* ← data_provider.get_next_batch().
2. The gradient is computed by back-propagation, which enables the information about the objective function to flow backwards through all layers.
3. The update is performed for layers which contain parameters.

The gradient computed at each step of SGD represents only an estimate of the true gradient. Even when the algorithm finds a local minimum, the gradient may still be non-zero. This might happen as the gradient is computed by randomly sampling the training set, which introduces a source of noise. The problem can be solved by decreasing the learning rate used by SGD over time. My implementation uses the following *learning rate schedule*:

$$\epsilon_k = (1 - \frac{k-1}{N})\ \epsilon_0, \tag{3.9}$$

where ϵ_0 represents the initial learning rate and N is equal to the total number of training iterations.

The optimisation process could also include the *hyperparameters* (for example, the number of filters in a convolutional layer or the number of neurons in a fully-connected layer). However, the network architecture that I am using contains many hyperparameters, so searching for the best combination would increase the duration of the training process too much, given the computational and time constraints. Therefore, I have manually set these values in the convolutional and fully-connected layers.

Back-propagation

The gradient estimate $\mathbf{g}$ is computed with respect to $\boldsymbol{\theta}$, which contains parameters usually found in hidden layers. Therefore, SGD requires the network to propagate the gradient information backwards through the layers, in order to be able to update the parameters as in 3.8. This is achieved by the *back-propagation* algorithm.

Algorithm 4 Back-propagation in a neural network.

procedure BACKPROP
  // Start at the output layer, computing the loss for the output of the network
  // and the real output given by the training example
  $\mathbf{g} \leftarrow \nabla_{\mathbf{h}_{\text{final}}} J(\boldsymbol{\theta}) = \nabla_{\mathbf{h}_{\text{final}}} L(y_{\text{pred}}, y_{\text{real}})$
  for each *layer* in *non-output layers* from last to first **do**
    // Compute gradients for parameters
    if *layer* has parameters **b**, **W then**
      $\nabla_{\mathbf{b}} J(\boldsymbol{\theta}) \leftarrow \mathbf{g}$
      $\nabla_{\mathbf{W}} J(\boldsymbol{\theta}) \leftarrow \mathbf{g}\ \mathbf{h}_{\text{in}}^T$
    // Compute new gradient for next layer to be processed
    $\mathbf{g} \leftarrow \nabla_{\mathbf{h}_{\text{in}}} J(\boldsymbol{\theta}) = \mathbf{g} \cdot f'_{\text{layer}}(\mathbf{h}_{\text{in}}^T)$

Consider a layer which receives an input $\mathbf{h}_{\text{in}}$ and computes the output $\mathbf{h}_{\text{out}}$. As discussed in Chapter 2, a layer can be thought of as computing a function f_{layer} of its input, hence:

$$\mathbf{h}_{\text{out}} = f_{\text{layer}}(\mathbf{h}_{\text{in}}). \tag{3.10}$$

During back-propagation, the incoming gradient is used to compute the gradients for the parameters (if the current layer uses any), namely the weights and biases. The layer then derives the new gradient which indicates how its output should change to reduce the objective function. This gradient is obtained by multiplying the incoming gradient with the derivative of the output with respect to the input. This derivative is expressed as:

$$\begin{aligned} \frac{d}{d\mathbf{h}_{\text{in}}}\mathbf{h}_{\text{out}} &= \frac{d}{d\mathbf{h}_{\text{in}}} f_{\text{layer}}(\mathbf{h}_{\text{in}}) \\ &= f'_{\text{layer}}(\mathbf{h}_{\text{in}}). \end{aligned} \tag{3.11}$$

Finally, the layer sends the new gradient to the previous layer. Algorithm 4 summarises back-propagation, where the notation $\nabla_x f$ is equivalent to $\frac{\partial f}{\partial x}$ and $\mathbf{h}_{\text{final}}$ denotes the input to the output layer.

3.2.3 Data provider

To obtain training examples from the dataset, the neural network training procedure uses the `_data_provider` instance variable which was previously passed to the `ConvNet` constructor. The training examples are provided in *batches* through the `DataProvider` method `get_next_batch()`. Each training example consists of:

- *the spectrogram for the current audio file*: a matrix of 32-bit floating point values corresponding to the greyscale intensities of the image, which in turn represent the frequency magnitudes computed from the audio signal; the matrix is normalised by dividing the brightness of each pixel by 255 to obtain a value in the range $[0, 1]$, such that the input to the neural network does not produce exploding results during forward- and back-propagation;

- *the corresponding genre*: an array of C values, where C is the number of genres currently being used; the k^{th} value is 1 if the training example belongs to the k^{th} genre and 0 otherwise.

As previously explained, I have used *stochastic gradient descent* as the optimisation method for training the network. Therefore, shuffling the examples in the dataset played an important role in computing an unbiased estimate of the expected gradient for each batch. The `DataProvider` class method `reset()` implements the shuffling of the indices for the training examples and is called at the start of each training iteration.

In addition, `DataProvider` allows the `ConvNet` object to retrieve the entire training set for computing training statistics through the method `get_all_training_data()`. The test set may be obtained similarly through `get_test_data()`.

3.3 Spectrogram generation from audio files

Convolutional neural networks learn features from the input data through the operations described in the Preparation chapter, being particularly well-suited for processing images. Therefore, the audio files in the GTZAN dataset were converted to images before being provided as input to the network.

An informative way to visualise a sound signal is through its *spectrogram* [5], which represents a plot of the frequency spectrum of the signal against time. In order to obtain the spectrogram, the audio file is first sampled at a specified frequency. Then, the Short-Time Fourier transform of overlapping chunks from the array of samples is performed. This results in a vertical band for each moment in the duration of the signal, where the magnitude of each frequency is given by the intensity of the corresponding point. Finally, the resulting bands are appended to form the complete representation of frequency magnitudes against time.

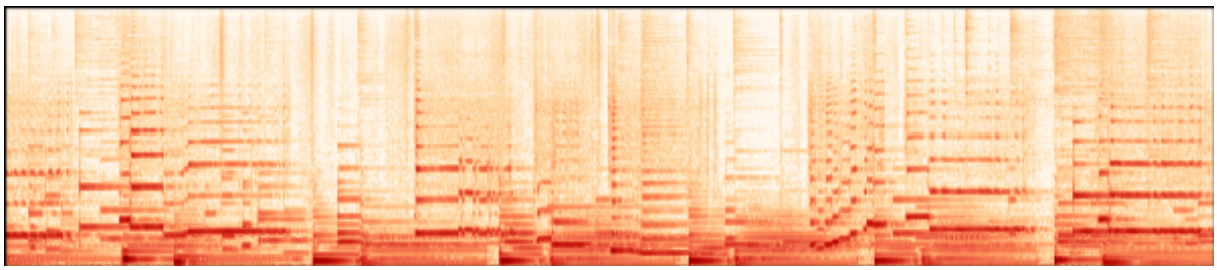


Figure 3.2: Spectrogram of the 97^{th} example in the jazz genre dataset.

Humans determine the genre of a song by listening to it, but the auditory system does not interpret frequencies in a linear manner, so a spectrogram would need to use a different scale in order to resemble the human perception of sounds. The *mel scale* was developed especially for this purpose, providing a scale of pitches which are perceived by humans as having the same distance between them. The relationship between f hertz and m mel [8] may be expressed as:

$$m = 2595 \log_{10}(1 + \frac{f}{700}). \tag{3.12}$$

All of these sound processing techniques are implemented in the `librosa` library for audio analysis which I used to extract the spectrograms from the dataset, as shown in Listing 3.2. Since its properties are relevant for the music genre classification task, the mel scale was used to plot frequency magnitudes against time. Figure 3.2 shows a spectrogram computed from an audio file in the dataset.

```
audio_time_series, sampling_rate = librosa.load(filepath)
spectrogram = librosa.feature.melspectrogram(
        y=audio_time_series,
        sr=sampling_rate,
        n_mels=128, # Number of mel scale values to use in the plot
        n_fft=1024, # Number of samples in the Fourier transform window
        fmax=10000)
```

Listing 3.2: Spectrogram extraction with `librosa` methods.

3.4 Implementation of neural network layers

In this section, I describe the types of layers which are present in the architecture from Listing 3.1. This structure was used to train the network to recognise musical genres.

3.4.1 Convolutional layer

As discussed in the Preparation chapter, convolutional layers have the ability to learn features in the input image. Each feature corresponds to a *kernel* or *filter*. The convolution operation of the image with a filter may be performed along one or two axes. My implementation of the `ConvLayer` class uses convolution along the time axis, because I require the network to model a time-invariant representation of the features. This means that a filter will recognise the feature it has learned at any moment within the song.

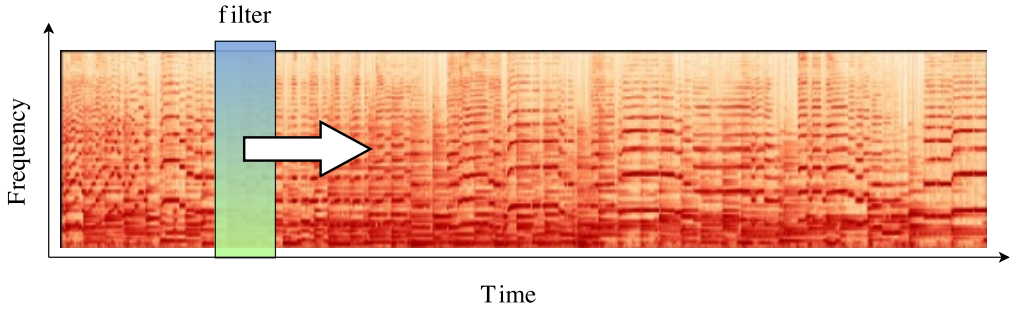


Figure 3.3: Convolution along the time axis.

A `ConvLayer` object is created by passing the following parameters to its constructor:

- `num_filters`: the number of filters corresponding to features which will be learned by the layer;
- `filter_shape`: a tuple `(h,w)` representing the dimensions of each filter; as convolution is made along the time (horizontal) axis, the height of the filter equals the height of the input;
- `weight_scale`: the variance σ^2 of the distribution $\mathcal{N}(0, \sigma^2)$ from which the filter weights are initialised; I used a heuristic [2] for setting the initial weight scale as

$$\sigma^2 = \frac{1}{\sqrt{\text{input\_size}}}, \tag{3.13}$$

  in order to ensure that the output of the layer is large enough to represent meaningful input data to the next layer, but not too large, in order to avoid exploding values.

Forward-propagation in the convolutional layer is illustrated in Figure 3.4. Each of the layer filters is convolved with the input, computing the dot product between the weight matrix of the filter and the current sub-matrix of the input which the filter overlaps, according to 3.14:

$$\text{dotProd}_{i,j} = \sum_{h=0}^{\text{filter\_height}} \sum_{w=0}^{\text{filter\_width}} (\text{filter}_i)_{h,w} \times \text{input}_{h,j+w}. \tag{3.14}$$

An additional value associated with the filter, called the *bias value*, is added to the dot product as shown in 3.15. The bias can be thought of as an additional weight stored by the layer which enables the associated filter to represent a larger class of features, by allowing the *threshold* value for the activation to be learned.

$$\text{output}_{i,j} = \text{dotProd}_{i,j} + \text{bias}_i \tag{3.15}$$

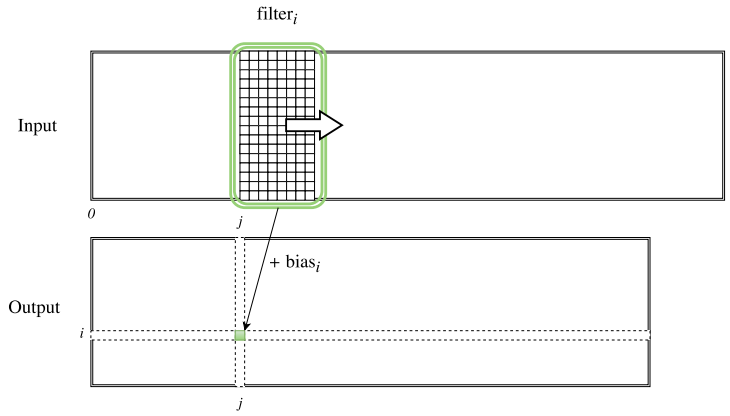


Figure 3.4: Forward-propagation in the convolutional layer.

3.4.2 Pooling layers

Also previously discussed in the Preparation chapter, a pooling layer in a convolutional neural network introduces a *prior assumption* that the representation learned by the layer must be invariant to translations. This is useful if we do not require the features to be detected at a specific location in the input, but just to know *whether they are present or not*. I have used two different types of pooling in the neural network architecture.

Max pooling

A *max pooling* layer summarises the input by reporting the maximum value in all (possibly overlapping) rectangular regions of a certain shape.

The second variant of max pooling in Figure 3.5 also performs *downsampling* by introducing a *stride* between the regions being pooled. Downsampling reduces the size of the representation by a factor less than or equal to the stride, which means that the input to the next layer will be smaller. The filters in the next convolutional layer will therefore have fewer parameters to be learned, so the probability of overfitting the training data and the computational time are reduced.

The `MaxPoolingLayer` object is created by passing the `filter_shape` parameter to the constructor. The parameter is a tuple representing the dimensions of the rectangular regions over which pooling is made. Since I had a very limited amount of training data to use in the project, the representations had to be reduced as much as possible in the

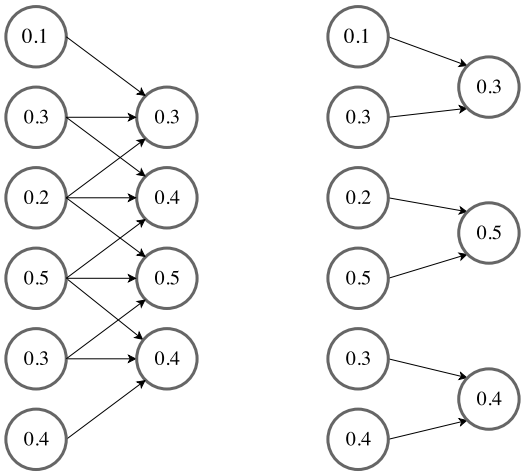


Figure 3.5: *(Left)* Max pooling with a pool width of 3. *(Right)* Max pooling with a stride of 2 and pool width of 2.

max pooling layers. This led to the decision of using a maximum stride, hence working with non-overlapping pooling regions.

As the max pooling operation is performed on the output from a convolutional-activation sequence of layers, each filter from the convolutional layer produces a row in the input to the max pooling layer. Therefore, I summarised the information for each row by using a pooling filter of shape `(1,stride)`, as shown in Figure 3.6.

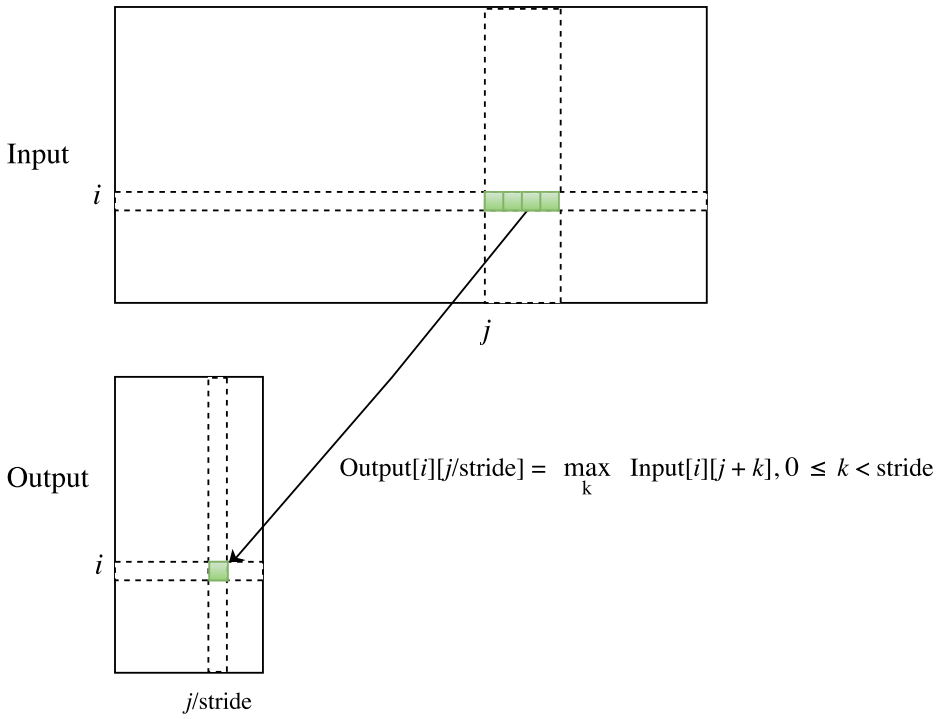


Figure 3.6: Forward-propagation in the max pooling layer.

Forward-propagation also stores the indices of the maximum values reported by the pooling operation. This allows the gradient from the next layer to flow back to the correct location, in the gradient computed by the max pooling layer. Back-propagation in the max pooling layer is presented in the CUDA layer operations section.

Global pooling

The purpose of convolutional layers is to learn features from the input data. In the architecture used, the max pooling layer in the last convolutional–activation–max-pooling sequence is replaced with a *global pooling* layer. This type of layer extracts a summarised representation of the features learned by the filters in the convolutional section of the network. Three varieties of pooling are performed on each row from the input (which corresponds to a filter output from the final convolutional layer):

- *max* pooling—reports the maximum value found in a row:

$$\max_{\text{row}} = \max_{k} \text{row}_k; \tag{3.16}$$

- *average* pooling—reports the mean of the row values:

$$\text{average}_{\text{row}} = \frac{1}{\text{input\_length}} \sum_{k=1}^{\text{input\_length}} \text{row}_k; \tag{3.17}$$

- L^2*-norm* pooling—reports the L^2-norm (Euclidean length) of the vector equal to the row:

$$L^2_{\text{row}} = \sqrt{\sum_{k=1}^{\text{input\_length}} {\text{row}_k}^2}. \tag{3.18}$$

The global pooling layer then forwards these features to the fully-connected region, which uses them for learning to classify the input.

```
def forward_prop(self, layer_input):
        assert self._input_shape == layer_input.shape,
               "Input does not have correct shape"
        self._current_input = layer_input

        output = np.empty(self.get_output_shape())
        for f in range(self._input_shape[0]):
            # Average pooling
            output[f] = np.mean(layer_input[f])
            # Max pooling
            output[self._input_shape[0] + f] = np.max(layer_input[f])
            self._max_activation_indices[f] = np.argmax(layer_input[f])
            # L2-norm pooling
            output[2 * self._input_shape[0] + f] = np.sqrt(np.sum(
                layer_input[f] ** 2))
        return output
```

Listing 3.3: Implementation of forward-propagation in the global pooling layer.

3.4.3 Activation layer

This type of layer applies an *activation function* to each element of the input. Popular activation functions used in neural networks include the *sigmoid*, *tanh* and *Rectified Linear Unit (ReLU)*, all illustrated in Figure 3.7.

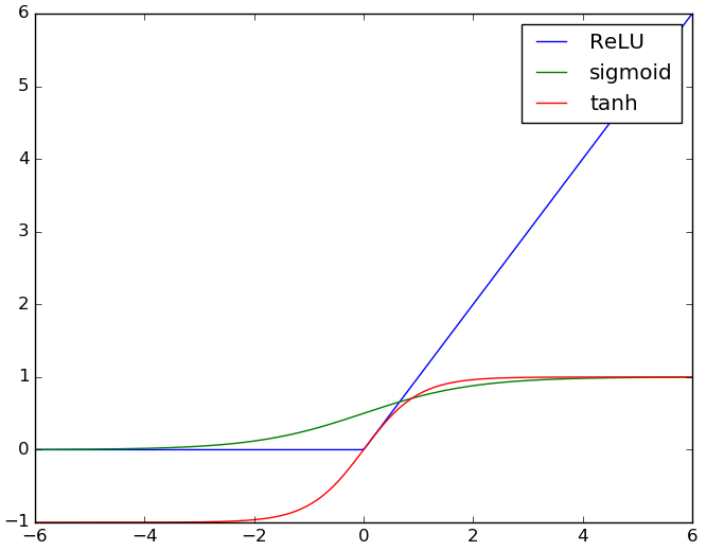


Figure 3.7: Activation functions.

The ReLU function has the following definition:

$$\text{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0. \end{cases} \tag{3.19}$$

and has recently become the most used activation function for deep neural networks, due to a number of desirable properties:

- it accelerates the optimisation through stochastic gradient descent (as reported, for example, in [4]);
- it is easier to implement and less computationally expensive;
- it does not saturate, whereas the sigmoid and tanh functions do.

However, the ReLU function might have the effect of updating the weights in the previous (convolutional or fully-connected) layer in such a manner that the output of the latter will never activate the ReLU neuron again. Therefore, a variant of this function—the *leaky ReLU*—has been applied by all activation layers which are part of the network architecture used in this project. The leaky ReLU function has the following form:

$$\text{leakyReLU}(x) = \begin{cases} 0.01x & \text{if } x \leq 0 \\ x & \text{if } x > 0, \end{cases} \tag{3.20}$$

and differs from the original function by still allowing a small gradient to flow back during back-propagation, even though the input was negative. The two derivatives in 3.21 illustrate this difference:

$$\frac{d}{dx}\text{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0. \end{cases}$$

$$\frac{d}{dx}\text{leakyReLU}(x) = \begin{cases} 0.01 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0. \end{cases} \tag{3.21}$$

3.4.4 Fully-connected layer

The second part of the network used for genre classification contains two pairs of fully-connected and activation layers, followed by a fully-connected layer and a final, output layer. *Fully-connected* layers implement the affine transformation:

$$\mathbf{z} = \mathbf{W}^T\mathbf{x} + \mathbf{b}, \tag{3.22}$$

where $\mathbf{W}$ is the matrix of weights such that W_{ij} = weight of the connection between the i^{th} input value and the j^{th} neuron, $\mathbf{b}$ is a vector containing the bias values for all neurons and $\mathbf{x}$ is the input to the layer.

A `FullyConnectedLayer` object is created by providing the following parameters:

- `num_nodes`: the number of neurons (hence the output size of the layer);
- `weight_scale`: has an identical purpose to the same parameter for the `ConvLayer` constructor, representing the variance σ^2 of the normal distribution from which the weights are initialised, using the discussed heuristic.

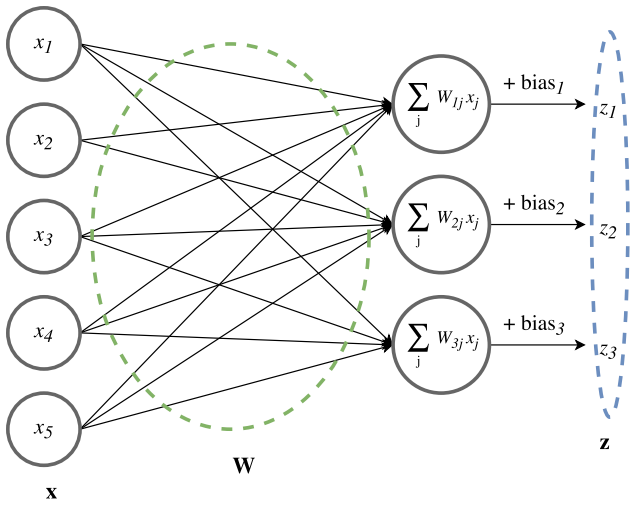


Figure 3.8: An example of a fully-connected layer with 5 inputs and 3 neurons.

Algorithm 5 Back-propagation in the fully-connected layer.

function BACK_PROP(**out_grad**)
  dW $\leftarrow$ (**input**) (**out_grad**)^T
  db $\leftarrow$ **out_grad**
  return W $\cdot$ **out_grad**

The back-propagation operation in a fully-connected layer takes the incoming gradient and modifies it as shown in Algorithm 5, before passing it to the previous layer.
Back-propagation also computes the values required for the parameter update phase of stochastic gradient descent, given by Algorithm 6.

Algorithm 6 Parameter update for the fully-connected layer.

procedure UPDATE_PARAMETERS(*learning_rate*)
  W $\leftarrow$ **W** $-$ *learning_rate* $\cdot$ **dW**
  b $\leftarrow$ **b** $-$ *learning_rate* $\cdot$ **db**

3.4.5 Softmax classification layer

The Preparation chapter introduced the *softmax* function in 3.23 as a way to convert the output of a fully-connected layer into a valid probability distribution. The neural network predicts the musical genre of the input, so we need the output layer to compute a probability distribution over C genres through *softmax classification*.

$$y_i = \text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} \tag{3.23}$$

The output of the softmax function can saturate when the difference between input values is extremely large. However, it can be noticed that adding the same value to all inputs results in the same output:

$$\begin{aligned} \text{softmax}(\mathbf{z} + \alpha)_i &= \frac{\exp(z_i + \alpha)}{\sum_{j=1}^{n} \exp(z_j + \alpha)} \\ &= \frac{\exp(z_i)\ \exp(\alpha)}{\sum_{j=1}^{n} \exp(z_j)\ \exp(\alpha)} \\ &= \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} \\ &= \text{softmax}(\mathbf{z})_i. \end{aligned} \tag{3.24}$$

Therefore, a more numerically stable version of softmax has been implemented for the `SoftmaxLayer` method `forward_prop()`. This variant only accounts for the distances to the maximum value and makes all inputs at most equal to 0, giving negligible numerical errors when the initial input contains extremely large or negative values:

$$\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} - \max_k z_k). \tag{3.25}$$

The softmax layer uses a different cost function than the one derived in the Preparation chapter, namely the *cross-entropy* loss for multiclass classification. This function is derived from the negative log-likelihood:

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\log \prod_{i=1}^{C} y_i^{t_i} \\ &= -\sum_{i=1}^{C} t_i \log y_i, \end{aligned} \tag{3.26}$$

where $t_i = 1$ if the current input belongs to class i and $t_i = 0$, otherwise. Since t_i does not depend on the predicted values y_i, minimising $J(\boldsymbol{\theta})$ is equivalent to minimising the following quantity:

$$-\sum_{i=1}^{C} t_i \log y_i + \sum_{i=1}^{C} t_i \log t_i = -\sum_{i=1}^{C} t_i \log \frac{y_i}{t_i}, \tag{3.27}$$

which results in an initial back-propagation gradient which is easy to implement and is provided by the function `initial_gradient(predicted_output, true_output)`:

$$\frac{\partial J(\boldsymbol{\theta})}{\partial z_j} = y_j - t_j. \tag{3.28}$$

3.5 CUDA layer operations

Apart from `SoftmaxLayer`, all `Layer` sub-classes have been extended with CUDA versions. These replace the `forward_prop()` and `back_prop()` methods with implementations which run on NVIDIA GPUs. The CUDA execution was easily integrated with the rest of the project by using the `PyCUDA` Python library. In this section, I present some of the kernels which were written to run on GPUs.

3.5.1 Convolutional layer

Forward-propagation

Listing 3.4 shows how a CUDA kernel call is executed with the help of `PyCUDA`. First, the kernel is defined by creating a `SourceModule`, which is then compiled by the NVCC (NVIDIA CUDA Compiler). Then, the function is loaded into the Python namespace and the kernel call is made, passing the following arguments:

- the parameters for the kernel; in this case, the input to the layer, the filter weights, the biases of the filters and a matrix which will contain the output, after the call returns;
- `block`, which specifies the dimensions of the blocks of threads;
- `grid`, which specified the dimensions of the grid containing blocks.

```
mod = SourceModule("""
    #define INPUT_WIDTH """ + str(self._input_shape[1]) + """
    #define FILTER_AREA """ + str(self._filter_shape[0] *
                                  self._filter_shape[1]) + """

    __global__ void conv_fwd_prop(double *in, double *f_weights,
                                  double *biases, double *out) {
    ...
    }
""")
conv_fwd_prop = mod.get_function('conv_fwd_prop')
conv_fwd_prop(driver.In(padded_input), driver.In(self._filter_weights),
              driver.In(self._biases), driver.InOut(output),
              block=(self._filter_shape[1], self._filter_shape[0], 1),
              grid=(self.get_output_shape()[1], self._num_filters, 1))
```

Listing 3.4: Forward-propagation kernel call in `ConvLayerCUDA`.

`PyCUDA` can abstract data transfers and device memory (de-)allocation, thus eliminating the need for boilerplate code encountered otherwise in CUDA, as seen in the Preparation chapter. The `pycuda.driver` module provides this functionality, allowing the programmer to specify the type of the argument passed to the call:

1. `driver.In(data)`—the data is copied to the device before the kernel call.
2. `driver.Out(data)`—the data is copied from the device after the kernel call.
3. `driver.InOut(data)`—performs both operations.

I have parallelised the forward-propagation as follows:

- each block corresponds to a location `(blockIdx.x, blockIdx.y)` in the output of the layer. Consequently, a block computes a pointwise multiplication between a filter weight matrix and the overlapping sub-matrix in the input;
- inside a block, each thread `(threadIdx.x, threadIdx.y)` multiplies the corresponding elements in the two matrices;
- `blockIdx.y` represents the index into the array of filters, so the i^{th} filter will correspond to a `blockIdx.y` value equal to i; the width of the filter is `blockDim.x` and its height is `blockDim.y`.

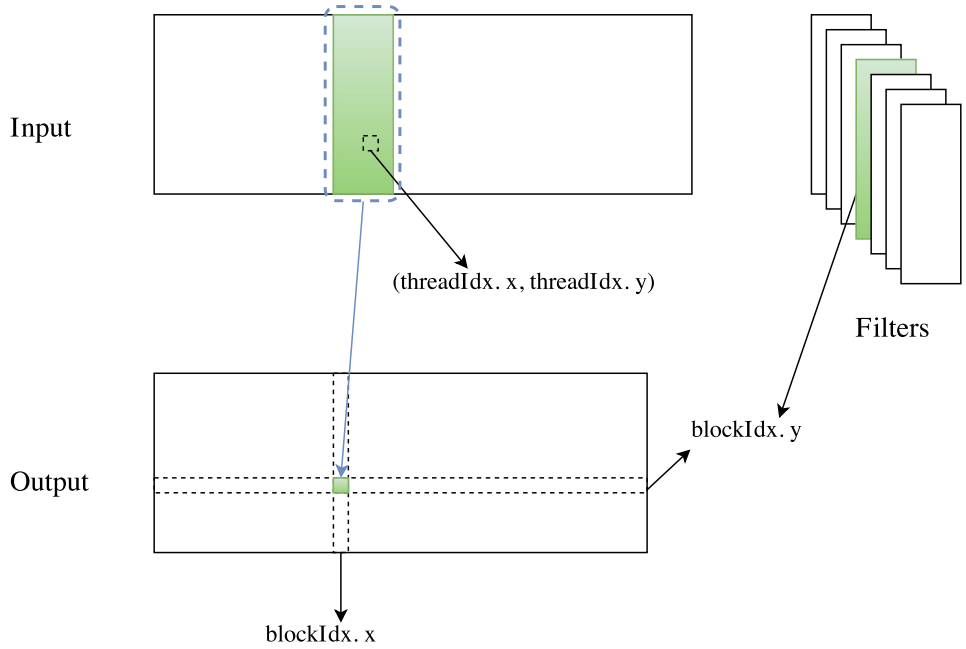


Figure 3.9: Parallelisation for forward-propagation in `ConvLayerCUDA`.

The kernel in Listing 3.5 implements the forward-propagation operation for convolutional layers.

```
__global__ void conv_fwd_prop(double *in, double *f_weights,
                              double *biases, double *out) {
    // Allocate shared memory for storing the point-wise multiplication
    // results for the current overlap of the filter with the input.
    __shared__ double temp[FILTER_AREA];

    // For the current location in the filter area,
    // compute the product between the filter weight and input value.
    temp[blockDim.x * threadIdx.y + threadIdx.x] =
        in[threadIdx.y * INPUT_WIDTH + blockIdx.x + threadIdx.x] *
           f_weights[blockIdx.y * blockDim.x * blockDim.y +
                     threadIdx.y * blockDim.x + threadIdx.x];

    // Ensure all results are available before computing the value
    // which is stored in the output matrix.
    __syncthreads();

    double sum = 0.0;
    for (int i = 0; i < FILTER_AREA; ++i) {
        sum += temp[i];
    }
    // Compute the final output by adding the bias value for the
    // current filter.
    out[blockIdx.y * gridDim.x + blockIdx.x] = sum + biases[blockIdx.y];
}
```

Listing 3.5: Kernel for forward-propagation in `ConvLayerCUDA`.

Back-propagation

The CUDA version for back-propagation in convolutional layers consists of two kernels:

- `compute_in_grad`, which takes the incoming gradient matrix and produces the new gradient which is passed on to the previous layer;
- `compute_derivatives`, which computes the necessary values for updating the filter weights and bias values.

I have written two separate kernels because the operations described above require different block dimensions and grid sizes in order to operate efficiently on the data, without wasting threads on the GPU.

The first kernel is invoked with a block size of `(filter_width, num_filters, 1)` and a grid dimension of `(input_width, input_height, 1)`. Therefore, the computation inside every block of threads corresponds to a location in the gradient being computed by the layer. Moreover, a block computes the sum of all the values in the incoming gradient which contribute to its corresponding location in the new gradient. Due to filter convolutions, the contributing values which are on the same row in `out_grad` are adjacent and each row corresponds to a filter, so the block sums over a vertical sub-matrix of `out_grad`.

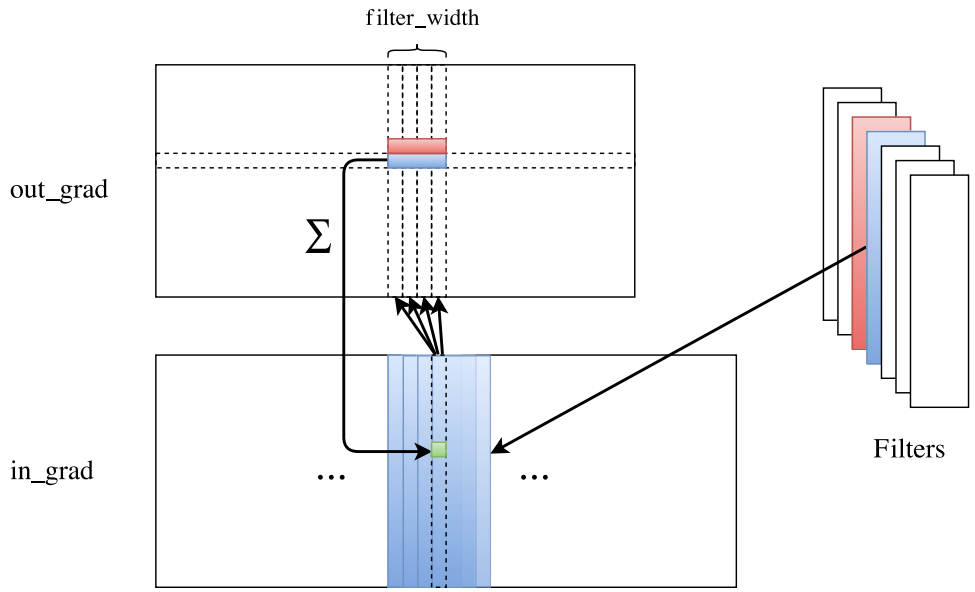


Figure 3.10: Illustration of the gradient computation during back-propagation in `ConvLayerCUDA`.

It can be noticed in Figure 3.10 that a given filter (shown in blue) overlaps a location in the input a number of times equal to `filter_width` (except near the input margins). Therefore, the gradient at that location needs to contain the sum of the corresponding elements from the incoming gradient, which are emphasised by the blue rectangle in `out_grad`. In addition, this summation is repeated for all filters to produce the final

value for the gradient at the given location (shown by the green square).

```
__global__ void compute_in_grad(double *out_grad, double *in,
                                double *f_weights, double *in_grad) {
    __shared__ double temp[FILTER_WIDTH * NUM_FILTERS];

    // Check that the filter completely lies inside the input grid.
    if (blockIdx.x - threadIdx.x < 0 ||
        blockIdx.x - threadIdx.x >= OUT_WIDTH) {
        return;
    }

    // Compute the locations of the current thread in the incoming
    // gradient and in the filter weight matrix.
    int out_idx = OUT_WIDTH * threadIdx.y + blockIdx.x - threadIdx.x;
    int f_weights_idx = threadIdx.y * FILTER_WIDTH * FILTER_HEIGHT +
                        threadIdx.x;
    // Compute the current contribution to the gradient.
    int idx = threadIdx.y * blockDim.x + threadIdx.x;
    temp[idx] = out_grad[out_idx] * f_weights[f_weights_idx];

    __syncthreads();

    double sum = 0.0;
    int i_max = FILTER_WIDTH * NUM_FILTERS;
    // Sum over the shared memory in the block to compute the value
    // at the corresponding location in the new gradient.
    for (int i = 0; i < i_max; ++i) {
        sum += temp[i];
    }
    in_grad[blockIdx.y * gridDim.x + blockIdx.x] = sum;
}
```

Listing 3.6: Kernel computing the gradient during back-propagation in `ConvLayerCUDA`.

The kernel which computes the values used in the parameter update phase is invoked with a block size of `(filter_width, filter_height, 1)` and a grid dimension of `(num_filters, 1, 1)`. Therefore, each block computes the update for the corresponding filter matrix and the associated bias value, as shown in Listing 3.7. I have split the computation according to the following mapping:

- the i^{th} filter corresponds to a value of `blockIdx.x` equal to i;
- `threadIdx.x` and `threadIdx.y` indicate the filter weight being updated on the current thread.

```
__global__ void compute_derivatives(double *out_grad, double *in,
                                    double *d_f_weights,
                                    double *d_biases) {
    // Compute the location in the current filter.
    int idx = FILTER_WIDTH * FILTER_HEIGHT * blockIdx.x +
              FILTER_WIDTH * threadIdx.y + threadIdx.x;
    // Compute the start index of the row in the input matrix.
    int in_idx = IN_WIDTH * threadIdx.y + threadIdx.x;
    // Compute the index for the start of the current row in out_grad.
    int out_idx = OUT_WIDTH * blockIdx.x;
    double d_bias = 0.0;

    for (int i = 0; i < OUT_WIDTH; ++i) {
        double val = out_grad[out_idx + i];
        // Add the product between the gradient value at the current
        // location and the corresponding input value which this filter
        // weight visited.
        d_f_weights[idx] += val * in[in_idx + i];
        // Sum all the gradient values being covered by the current
        // filter weight, in order to compute the bias value update.
        if (!threadIdx.x && !threadIdx.y) {
            d_bias += val;
        }
    }
    if (!threadIdx.x && !threadIdx.y) {
        d_biases[blockIdx.x] += d_bias;
    }
}
```

Listing 3.7: Kernel computing the parameter update values in `ConvLayerCUDA`.

3.5.2 Max pooling layer

As a final example, I explain the back-propagation implementation for the max pooling layer. The gradient `out_grad` is received from the next layer, allowing a new gradient `in_grad` to be computed and passed on to the previous layer. The `max_activation_indices` matrix computed in forward-propagation stores, for each output location, the corresponding input location from where pooling extracted the maximum value. Figure 3.11 illustrates the parallelisation of this operation.

```
__global__ void max_pool_back(double *out_grad, double *in_grad,
                              int *max_activation_indices) {
    // Compute the location in the gradient matrix for the current thread.
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    // Ensure we are at a valid location in the matrix.
    if (col >= gridDim.x * blockDim.x || row >= gridDim.y * blockDim.y) {
        return;
    }
```

```
    int out_idx = col + row * gridDim.x * blockDim.x;
    // Send the incoming gradient value to the location corresponding
    // to the maximum value found during forward-propagation.
    in_grad[max_activation_indices[out_idx] +
           row * INGRAD_WIDTH] = out_grad[out_idx];
}
```

Listing 3.8: `MaxPoolingLayerCUDA` back-propagation kernel.

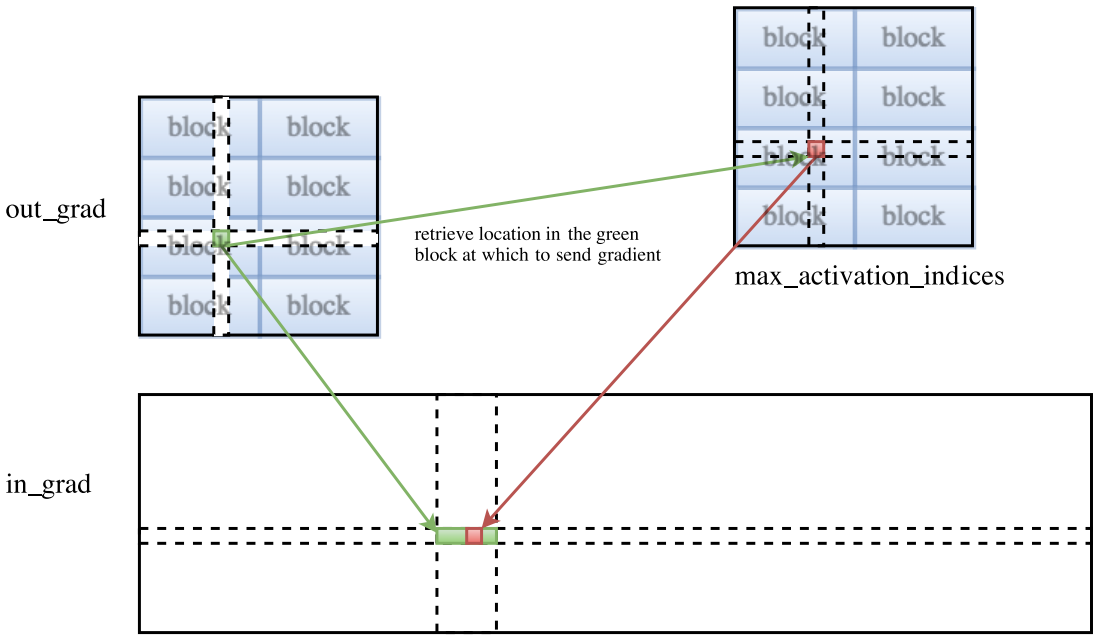


Figure 3.11: Back-propagation in `MaxPoolingLayerCUDA`.

3.6 Summary

In this chapter, I have discussed how the classification system performs training on a dataset, the methods it uses for optimising the learned model and the classes which it consists of. Following this, I have presented several CUDA kernels which perform forward- and back-propagation in two types of layers. The next chapter will show the results I obtained from evaluating the accuracy and performance of the classification system.

Chapter 4

Evaluation

In this chapter, I show how the implementation was tested, discuss the accuracy evaluation of the classifier and provide a comparison of the runtimes of layer operations on a CPU and two different NVIDIA GPUs. Finally, I illustrate the operation of a convolutional layer filter on spectrograms.

4.1 Unit testing

Unit testing is performed on individual software units to verify that they are correct and show the expected behaviour. I implemented the classification system by following object-oriented programming principles, so the units which were tested correspond to individual class methods. I used the integrated testing framework of the PyCharm IDE to develop and run automated unit tests.

White-box testing was carried out to ensure that special cases were handled correctly (such as particular values in layer inputs). In addition, black-box testing verified the functionality of the units before they were integrated with the rest of the system.

Figures 4.2 – 4.7 show snapshots of the unit testing results for the two types of `Layer` subclasses implementing CPU and GPU versions of layer operations. The `DataProvider` methods have also been tested for correctness, as shown in Figure 4.1.

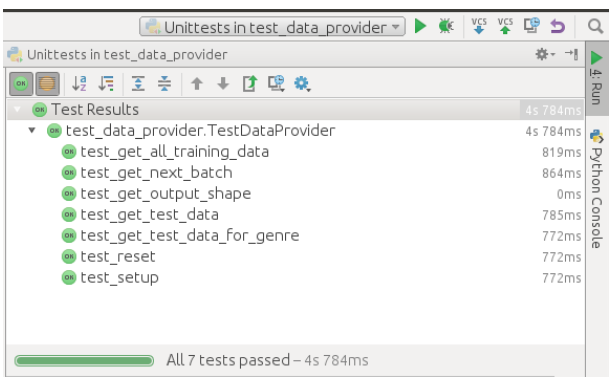
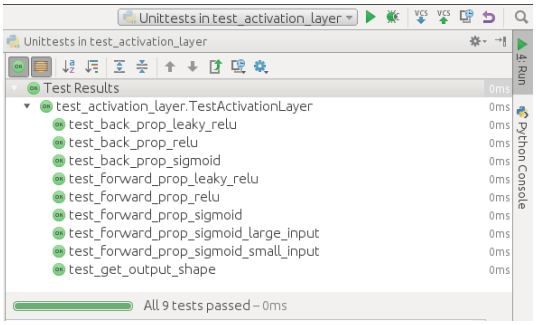
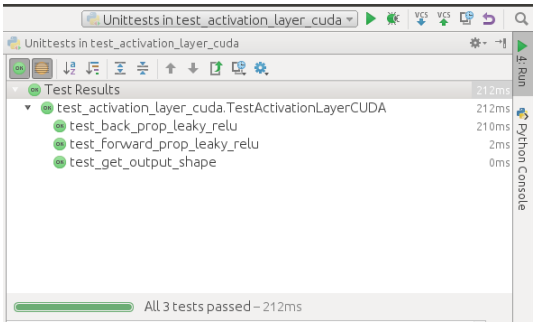


Figure 4.1: Snapshot showing results of unit tests for the `DataProvider` class.

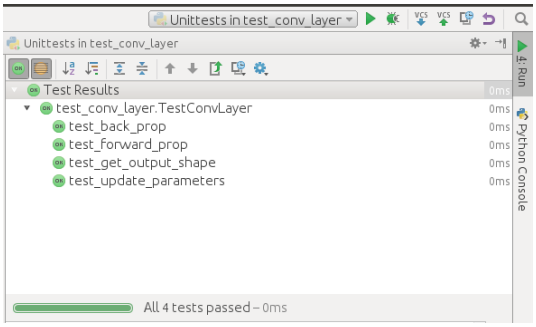


(a) `ActivationLayer` testing results.

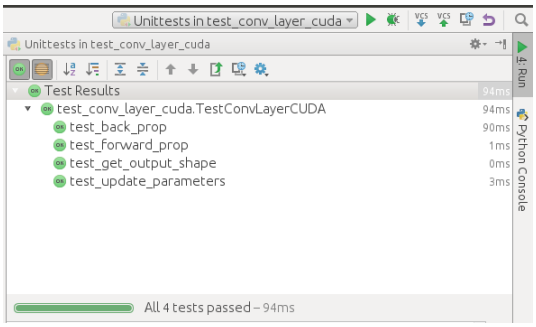


(b) `ActivationLayerCUDA` testing results.

Figure 4.2: Unit testing for classes implementing the activation layer.

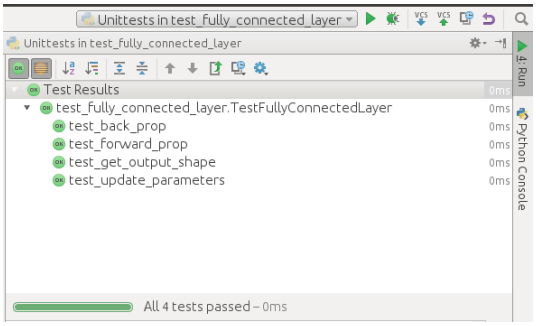


(a) `ConvLayer` testing results.

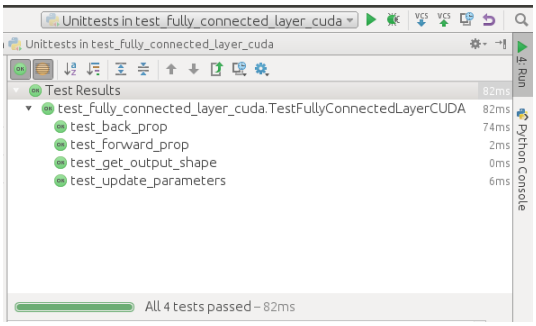


(b) `ConvLayerCUDA` testing results.

Figure 4.3: Unit testing for classes implementing the convolutional layer.

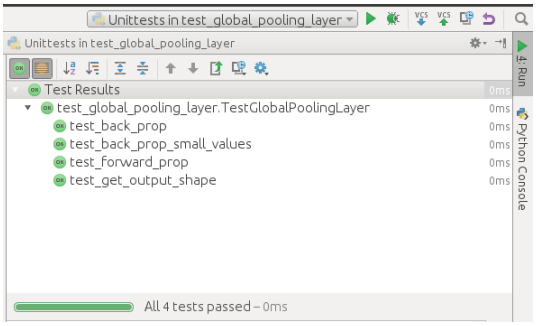


(a) `FullyConnectedLayer` testing results.

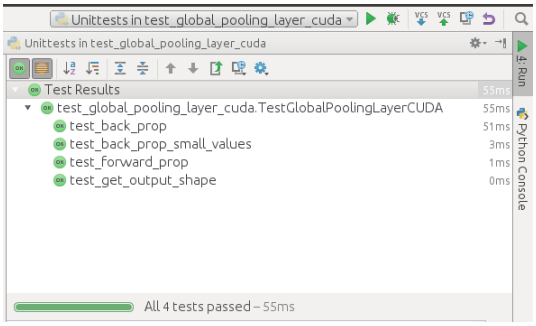


(b) `FullyConnectedLayerCUDA` testing results.

Figure 4.4: Unit testing for classes implementing the fully-connected layer.

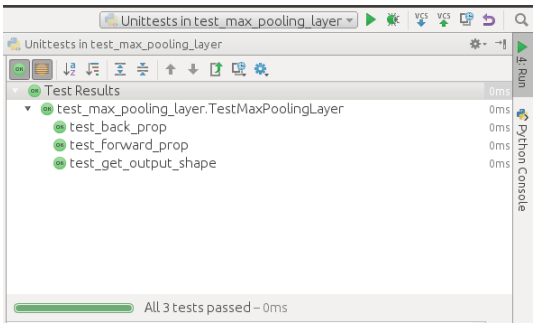


(a) `GlobalPoolingLayer` testing results.

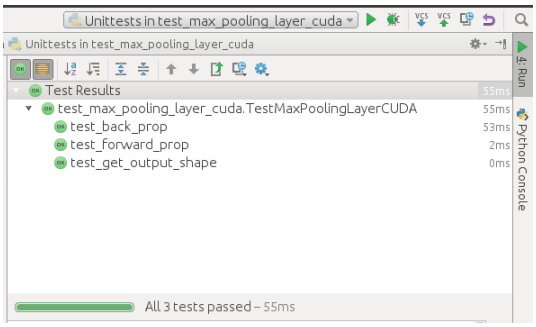


(b) `GlobalPoolingLayerCUDA` testing results.

Figure 4.5: Unit testing for classes implementing the global pooling layer.



(a) `MaxPoolingLayer` testing results.



(b) `MaxPoolingLayerCUDA` testing results.

Figure 4.6: Unit testing for classes implementing the max pooling layer.

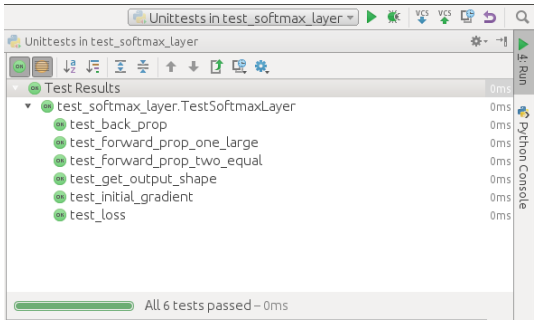


Figure 4.7: Unit testing for the `SoftmaxLayer` class.

4.2 Integration testing

The *first core objective* of the project involved implementing a working classification system which uses a convolutional neural network. The functionality of the system lies in its ability to minimise the objective function discussed in Chapter 3 by gradient descent, during the training phase.

I performed integration testing on the system by running the classifier for a specified number of training iterations. After each iteration, the system recorded the accuracy on both training and test sets, along with the average value of the cross entropy loss function. After the training finished, I plotted the obtained values for each iteration and checked that the classifier was able to minimise the objective function on both training and test sets, as expected. Figure 4.8 shows an example of how the value of the function changes as the system undergoes more training iterations. The corresponding error rates, which are also decreasing as expected, are shown in Figure 4.9.

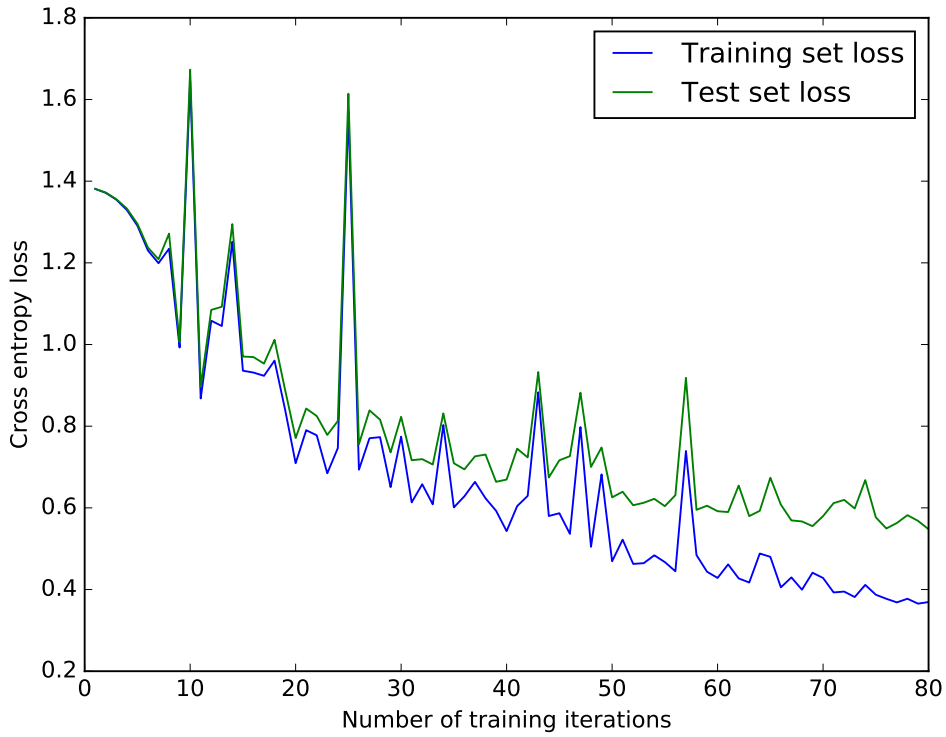


Figure 4.8: Minimisation of the loss function in the neural network.

4.3 Accuracy evaluation

4.3.1 Evaluation data

The GTZAN dataset was used to evaluate the performance of the classification system. GTZAN consists of 1000 audio clips with a duration of 30 seconds each and a sampling

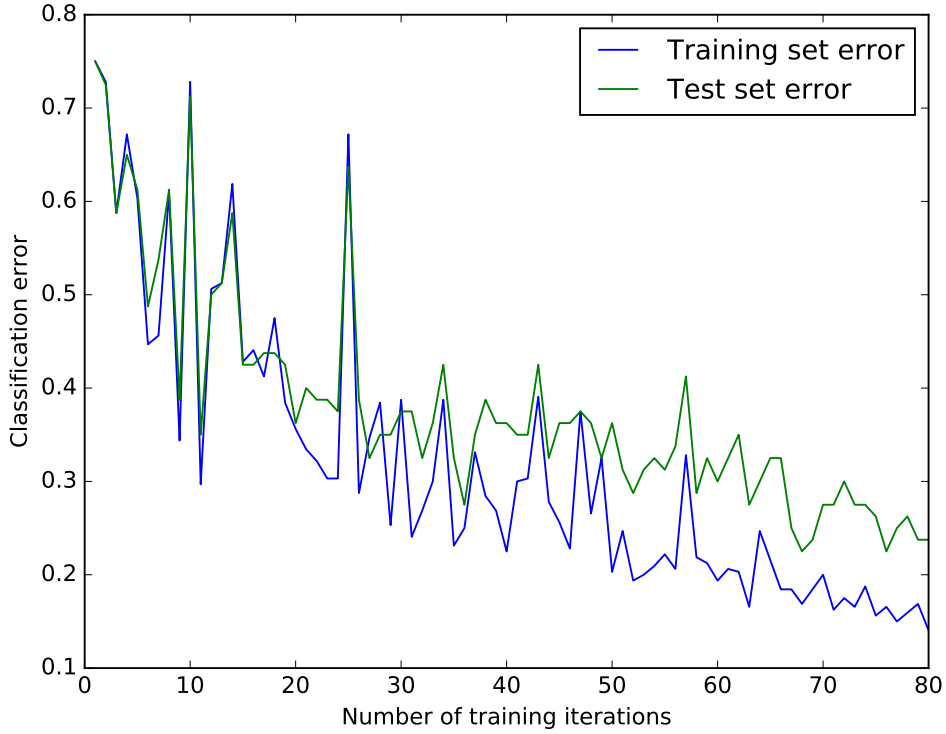


Figure 4.9: Classification error of the neural network.

rate of 22050 Hz. Each of them is labelled with one of ten musical genres: classical, metal, disco, blues, pop, country, jazz, reggae, rock and hip-hop. GTZAN is a balanced dataset, having 100 audio files belonging to each genre.

4.3.2 Performance metrics

Accuracy

The performance of the classifier was evaluated with balanced data. In this case, the *accuracy* metric is most suitable for indicating the *generalisation* ability of the classifier (how well it does on assigning the correct class to previously unseen data). The accuracy of the classifier [3] is equivalent to the *probability of error* on a test set $\mathbb{X}_{\text{test}}$ containing m examples, given by:

$$\hat{\text{err}}_{\mathbb{X}_{\text{test}}}(f) = \frac{1}{m}\sum_{i=1}^{m}\mathbb{I}(f(\mathbf{x}^{(i)}) \neq y_i). \tag{4.1}$$

In order to derive a confidence interval for the error estimate, the classifier was trained on $N = 50$ random partitions of the dataset when performing binary classification and on $N = 30$ random partitions for classifying four and six genres. The errors from training the classifier on each partition $(\mathbb{X}_{\text{test}})_i$ were recorded and then used to compute the error estimate:

$$\hat{\text{err}}(f) = \frac{1}{N}\sum_{i=1}^{N}\hat{\text{err}}_{(\mathbb{X}_{\text{test}})_i}(f). \tag{4.2}$$

Assuming the true error is $\text{err}(f)$, such that $\mathbb{E}[\text{err}(f)] = \mu$ and $\text{Var}(\text{err}(f)) = \sigma^2$, we also have $\mathbb{E}[\hat{\text{err}}(f)] = \mu$. However, the variance changes:

$$\begin{aligned}\text{Var}(\hat{\text{err}}(f)) &= \text{Var}(\frac{1}{N}\sum_{i=1}^{N}\hat{\text{err}}_{(\mathbb{X}_{\text{test}})_i}(f)) \\ &= \sum_{i=1}^{N}\frac{1}{N^2}\text{Var}(\hat{\text{err}}_{(\mathbb{X}_{\text{test}})_i}(f))) \\ &= \sum_{i=1}^{N}\frac{1}{N^2}\sigma^2 \\ &= \frac{\sigma^2}{N}.\end{aligned} \tag{4.3}$$

The Binomial distribution gives the probability of seeing a number of errors for a test set containing m examples. If the mean of the distribution is p, then its variance is $\frac{p(1-p)}{m}$. If $m \geq 30$ [6], we can approximate the Binomial distribution with the Normal distribution in order to derive a 95% confidence interval (with $z_{.95} = 1.96$) [3] for the probability of error of the classifier:

$$\text{err}(f) = \hat{\text{err}}(f) \pm 1.96 \times \sqrt{\frac{\hat{\text{err}}(f)(1-\hat{\text{err}}(f))}{N \times m}}. \tag{4.4}$$

Confusion matrix

When evaluating the classifier on more than two genres, it is useful to visualise the *confusion matrix*. This provides the proportion of examples correctly classified for each genre, along with how the error rate for that genre is spread across the other classes.

		Predicted		
		C_1	C_2	C_3
Actual	C_1	**50%**	20%	30%
	C_2	20%	**60%**	20%
	C_3	0%	20%	**80%**

Table 4.1: An example of a confusion matrix for three classes.

4.3.3 Results

Two genres

The *second core objective* of the project was to achieve better-than-chance prediction on a test set containing *two genres*. I evaluated the performance of the classifier on $N = 50$ random partitions of the dataset containing 200 examples from the *classical* and *metal* genres. The training set had 80% of the examples, while the test set was formed by the remaining 20% ($m = 40$); the two genres were equally represented in the partitions. Using the values for N and m, the 95% confidence interval for the classification error is:

$$\text{err}(f) = \hat{\text{err}}(f) \pm 1.96 \times \sqrt{\frac{\hat{\text{err}}(f)(1-\hat{\text{err}}(f))}{2000}}. \tag{4.5}$$

The predictions of the classifier on the random test sets have led to the following confidence interval for the probability of error:

$$0.8\% \pm 0.4\%.$$

As the value above corresponds to an almost perfect genre prediction, this result also achieves an *extension* of the project—obtaining a better two-class accuracy by improving the network architecture.

Four genres

Another *extension* involved achieving better-than-chance prediction on a test set having *more than two genres*. I evaluated the performance of the classifier on the *classical*, *metal*, *blues* and *disco* genres, training the network on 30 random partitions of the dataset consisting of 400 examples. As the amount of data per genre was limited, I increased the proportion of training examples when classifying more than two genres. Hence, the training set consisted of 90% of the examples, while the remaining 10% formed the test set. The 95% confidence interval for the probability of error is given by:

$$\mathrm{err}(f) = \hat{\mathrm{err}}(f) \pm 1.96 \times \sqrt{\frac{\hat{\mathrm{err}}(f)(1 - \hat{\mathrm{err}}(f))}{1200}}. \tag{4.6}$$

Results showed that the error rate of the classifier lies in the following confidence interval:

$$20\% \pm 2\%.$$

This proves that increasing the number of classes while training the network with almost the same amount of data per genre lowers the performance of the classifier. However, the obtained error still corresponds to a prediction ability which is much better than chance, or than an error rate of 75%.

The confusion matrix in Table 4.2 illustrates the average prediction behaviour of the classifier over the 30 random partitions. Classical and metal remain the genres with the lowest misclassification rates. Blues has the highest error, being mispredicted almost one in four times as disco; the latter is also mistaken for blues 18% of the time.

		Predicted			
		Classical	*Metal*	*Blues*	*Disco*
Actual	*Classical*	**92%**	1%	3.3%	3.6%
	Metal	0%	**86.6%**	7%	6.3%
	Blues	2.6%	8.3%	**65.3%**	23.6%
	Disco	2.6%	3.3%	18%	**76%**

Table 4.2: Confusion matrix for four genres.

Six genres

In order to evaluate the performance of the classifier on six genres, the network was trained on 30 random partitions of the dataset containing 600 examples from the *classical*, *metal*, *blues*, *disco*, *hip-hop* and *reggae* genres. The same 90:10 proportion was used for the training and test sets. The 95% confidence interval for the error rate is computed according to:

$$\mathrm{err}(f) = \hat{\mathrm{err}}(f) \pm 1.96 \times \sqrt{\frac{\hat{\mathrm{err}}(f)(1 - \hat{\mathrm{err}}(f))}{1800}}. \tag{4.7}$$

I have obtained the following probability of classification error from the test results:

$$31\% \pm 2\%.$$

Once again, the resulting error corresponds to a prediction ability which is much better than chance, in this case equivalent to an error rate of 83.3%. Moreover, the decrease in the error rate from 4 to 6 classes (11%) is smaller than the one from 2 to 4 classes (19.2%), for the subset of genres used to evaluate the classifier.

The confusion matrix in Table 4.3 illustrates the average prediction behaviour of the classifier over the 30 random partitions. Classical and metal remain the least misclassified genres, with error rates comparable to the ones obtained in Table 4.2 (classical even improves by 4.6%). Disco becomes the most mispredicted genre, with reggae having a close error rate. Blues and hip-hop achieve slightly better prediction rates, but the proportion of correctly classified examples for the latter 4 genres does not reach 60%. Interestingly, blues and disco are still mostly mistaken for one another, as in previous results, and so are hip-hop and reggae (almost one in five times).

		Predicted					
		Classical	*Metal*	*Blues*	*Disco*	*Hip-Hop*	*Reggae*
Actual	*Classical*	**96.6%**	0%	1.6%	1.6%	0%	0%
	Metal	0%	**86.3%**	7.0%	2.6%	4%	0%
	Blues	4.3%	10%	**59.3%**	15.3%	4%	7%
	Disco	3.3%	3.3%	19.3%	**56.0%**	10.6%	7.3%
	Hip-Hop	0.3%	2%	4.3%	14%	**59.6%**	19.6%
	Reggae	2%	0%	12.3%	9.3%	19%	**57.3%**

Table 4.3: Confusion matrix for six genres.

More genres

Since I used a deep learning method for this project, the training of the neural network required learning many parameters: the weights for each filter in each convolutional layer and one matrix of weights per fully-connected layer, along with the corresponding bias values. The GTZAN dataset contains a very limited amount of data—100 examples per genre—which decreased the performance of the classifier when trained to recognise eight and ten genres. The errors obtained were above 40% in each case, still corresponding to a better-than-chance prediction ability, but resulting in higher confusion between genres.

4.4 Performance evaluation

In this section, I compare the running times of the CPU and GPU versions of forward- and back-propagation for each type of layer discussed in the Implementation chapter. The CPU methods were run on a quad-core hyperthreaded Intel Core i7-3630QM 2.40GHz processor. In addition, the CUDA layer operations were evaluated on two NVIDIA GPUs: a GeForce GT640M with 384 cores, running at a maximum frequency of 625 MHz, and a GeForce GTX980M with 1536 cores and a base clock of 1038 MHz.

4.4.1 Activation layer

As shown in Figure 4.10 and Figure 4.11, the CPU version is faster for an input size of up to 2^{17} for forward-propagation and 2^{18} for back-propagation. This suggests that the Python operation of applying a pointwise activation function to an array of up to 2^{15} elements is much faster than the sole transfer of data required for GPU computations (almost 2 orders of magnitude faster, for inputs of up to 1024 elements).

The GPU versions are not considerably faster, even for input sizes above the specified thresholds. Forward-propagation runtimes are slightly lower on both GPUs, whereas back-propagation only achieves better performance on the GT640M.

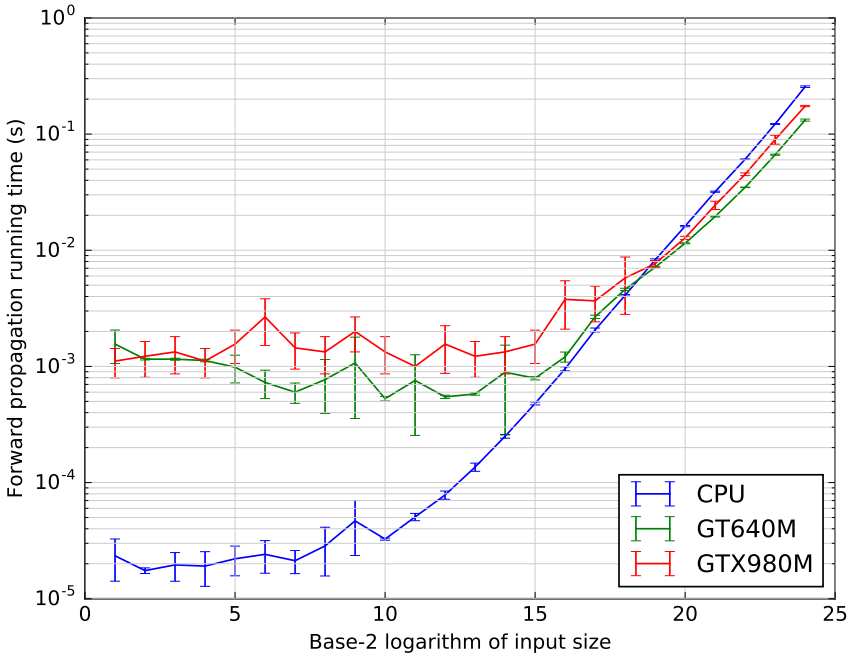


Figure 4.10: Forward-propagation average runtimes in the activation layer. In the current and following plots, the size represents the product of the width and the height of the matrix and the error bars represent one standard deviation obtained from ten runs of the layer operation for the corresponding matrix size.

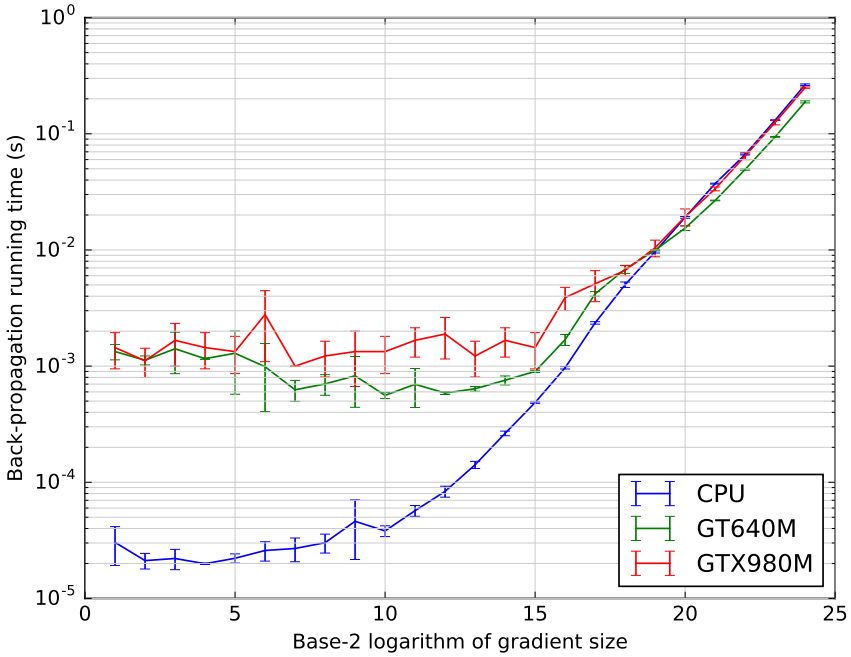


Figure 4.11: Back-propagation average runtimes in the activation layer.

4.4.2 Convolutional layer

As in the previous case, the runtimes for convolutional layer operations were obtained from performing forward- and back-propagation on inputs of exponentially increasing size. Regardless of the input size, 32 filters were used, each having a width of 4.

The graph for back-propagation in Figure 4.13 shows that both GPUs outperform the CPU, starting from an input size of 256. In addition, the GTX980M is at least an order of magnitude faster than the CPU on inputs containing more than 2^{16} elements. However, the CPU performs forward-propagation faster than the GT640M, when the input size is at least 2^{14}. The GTX980M outruns the CPU very early, starting from inputs of 32 elements, while the GT640M is faster than the CPU only when processing inputs which contain between 32 and 8192 elements. The latter might happen because the GT640M only uses 384 CUDA low-performance cores (4 times fewer than the GTX980M), which may result in a slower computation than the one performed by the CPU.

4.4.3 Fully-connected layer

The results for the fully-connected layer operations show that the matrix multiplication method in the `NumPy` library is highly optimised, compared to the version implemented in the CUDA kernels. In addition, for inputs of up to 4096 elements, the overheads of transferring the data alone require an amount of time two orders of magnitude greater than the CPU time. The GT640M slightly outperforms the GTX980M in some cases, while the latter becomes faster on both operations by approximately 25% and 20%, respectively,

when processing the largest input. The CPU and GPU runtimes only become comparable at this point, with the CPU outperforming the GPUs by at most 0.003s, achieving 0.001s on forward-propagation and 0.002s on back-propagation.

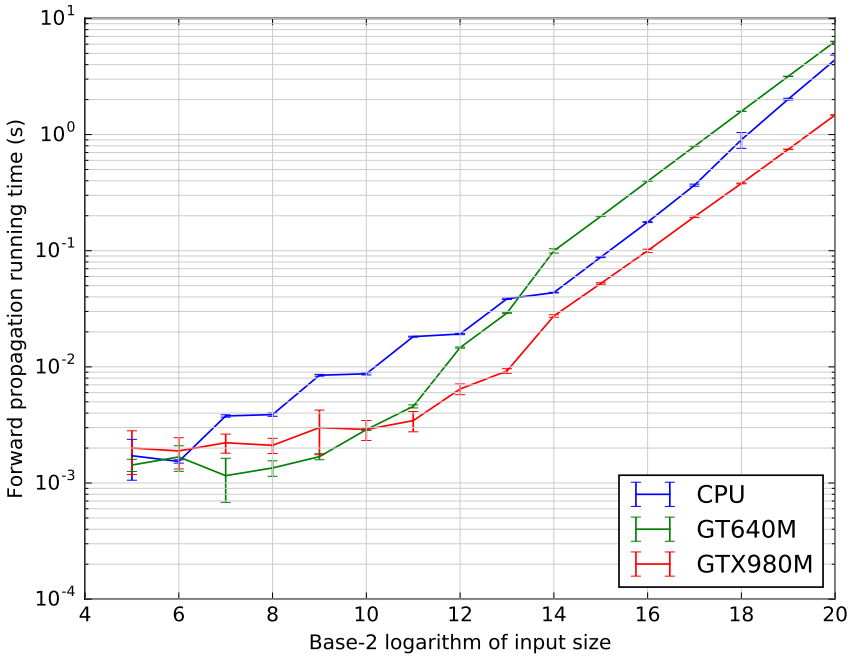


Figure 4.12: Forward-propagation average runtimes in the convolutional layer.

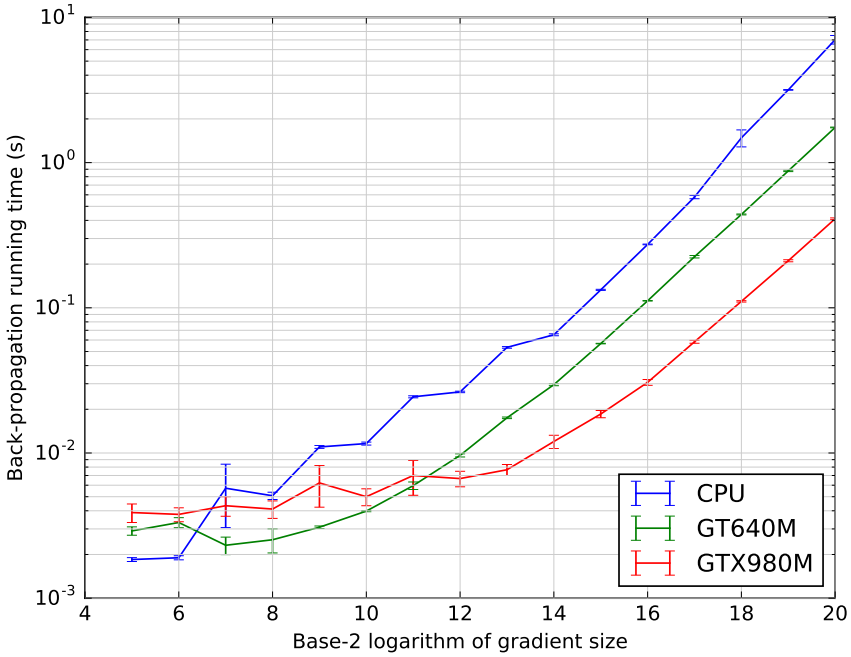


Figure 4.13: Back-propagation average runtimes in the convolutional layer.

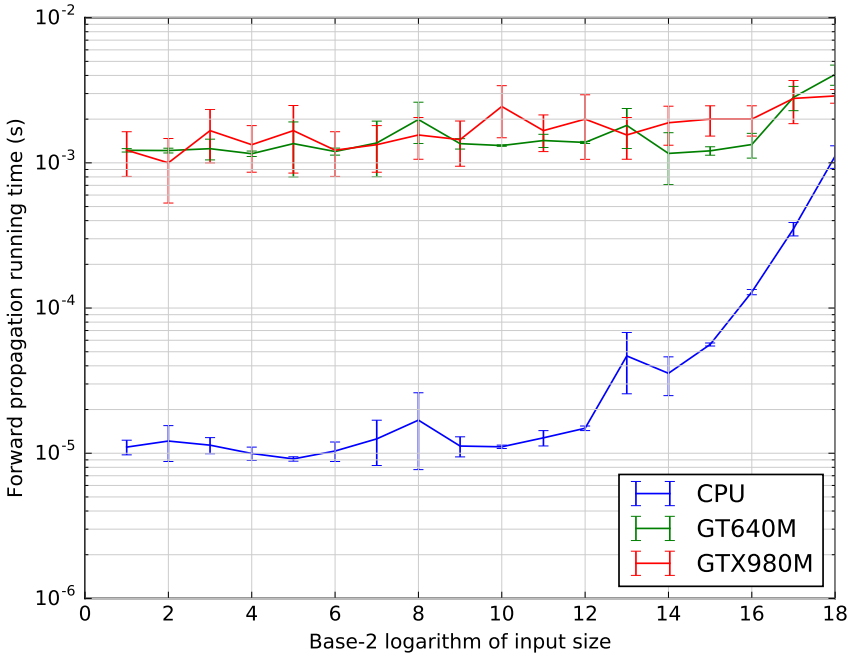


Figure 4.14: Forward-propagation average runtimes in the fully-connected layer.

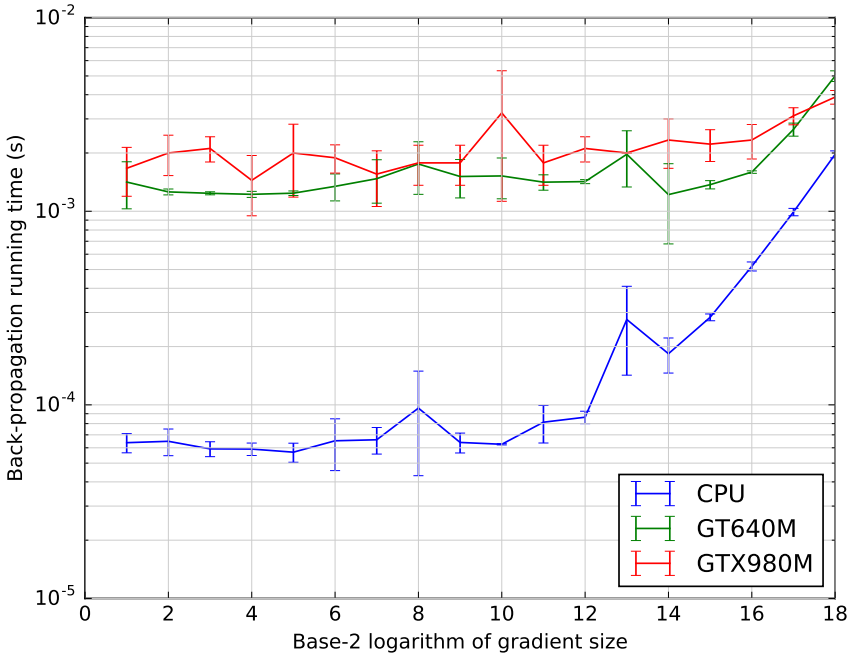


Figure 4.15: Back-propagation average runtimes in the fully-connected layer.

4.4.4 Global pooling layer

The GPU versions of forward-propagation in the global pooling layer become faster when the input size reaches 2^{14}. However, the GT640M is outperformed by the CPU for the two largest input sizes. Back-propagation is eventually faster on both GPUs, starting from inputs of size 2^{16}, achieving a relative improvement of at most 25%.

4.4.5 Max pooling layer

The max pooling runtimes illustrate that the two layer operations are highly parallelisable. This is because the computations in each block do not use shared memory or synchronisation function calls, having no dependencies on other data. For inputs of more than 8192 elements, forward- and back-propagation are at least one order of magnitude faster on both GPUs. The CUDA implementations eventually achieve a speedup of approximately 100x for the maximum input size of 2^{24}.

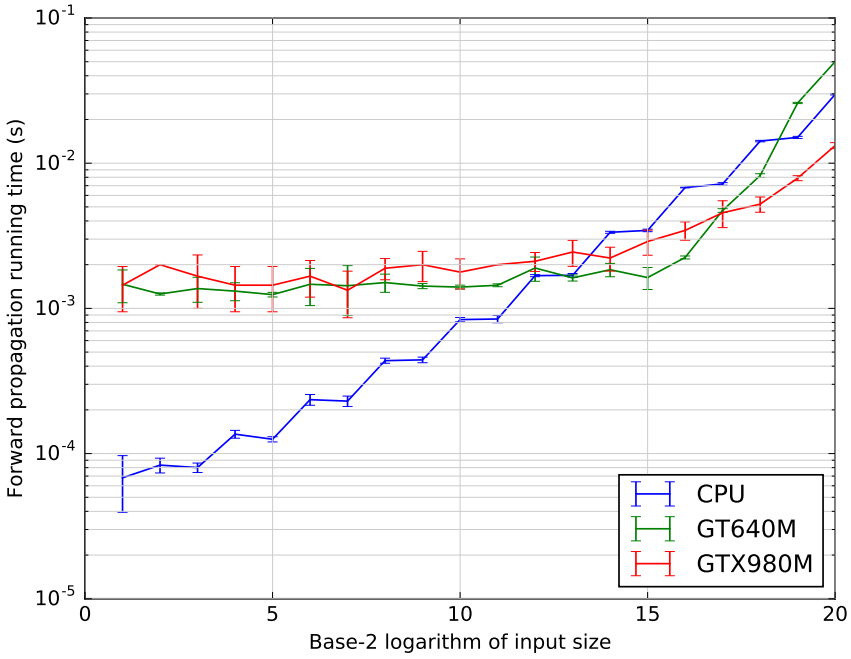


Figure 4.16: Forward-propagation average runtimes in the global pooling layer.

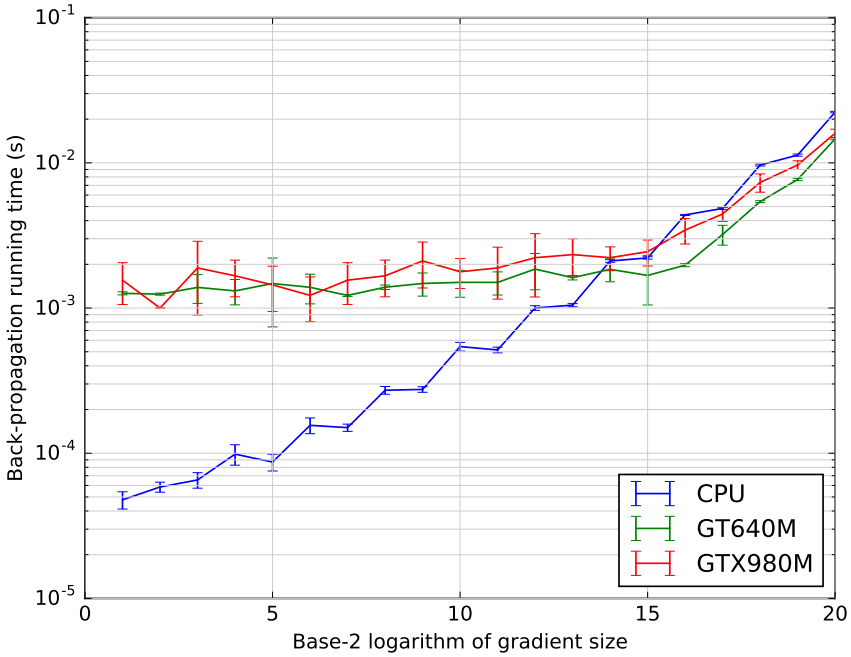


Figure 4.17: Back-propagation average runtimes in the global pooling layer.

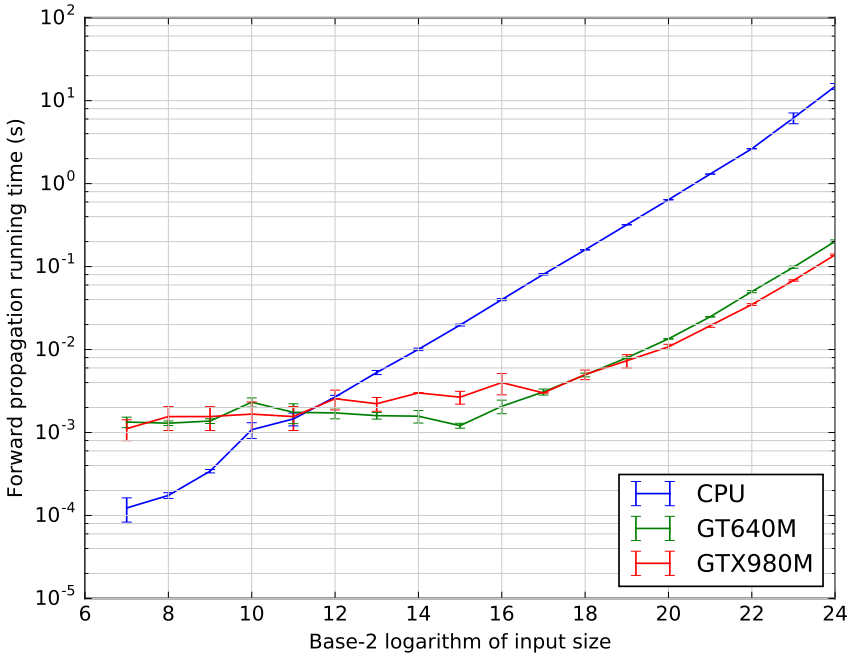


Figure 4.18: Forward-propagation average runtimes in the max pooling layer.

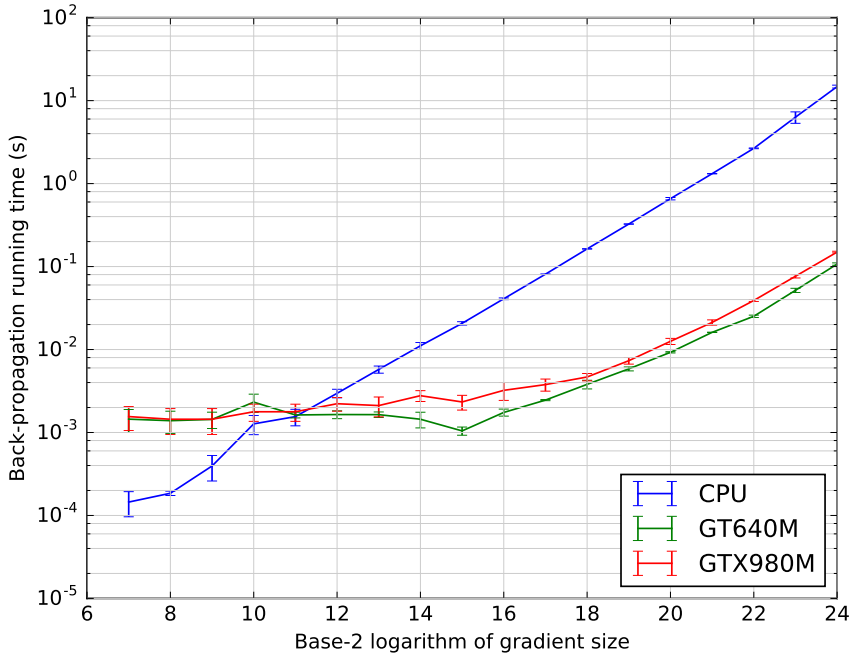


Figure 4.19: Back-propagation average runtimes in the max pooling layer.

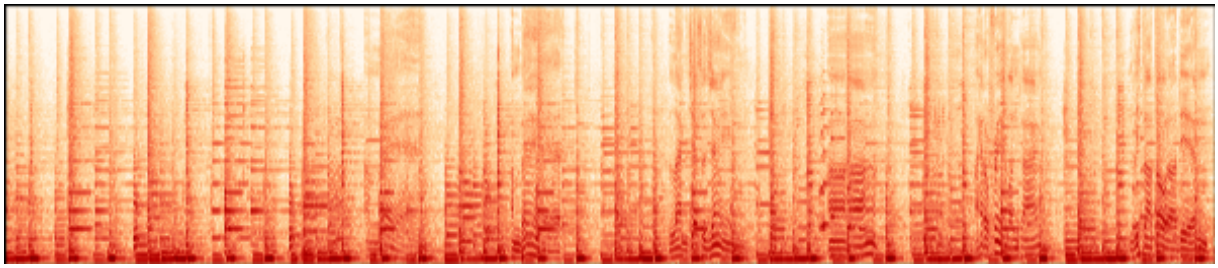
4.5 Analysis of a filter

Another *extension* of the project aimed to investigate what the filters in convolutional layers are learning about and therefore producing as output. I trained the neural network on a random partition of the dataset for 4 genres, obtaining an error of 17.5% on the test set. I then used the parameters learned by the convolutional layers to plot the outputs computed by each filter on all spectrograms.

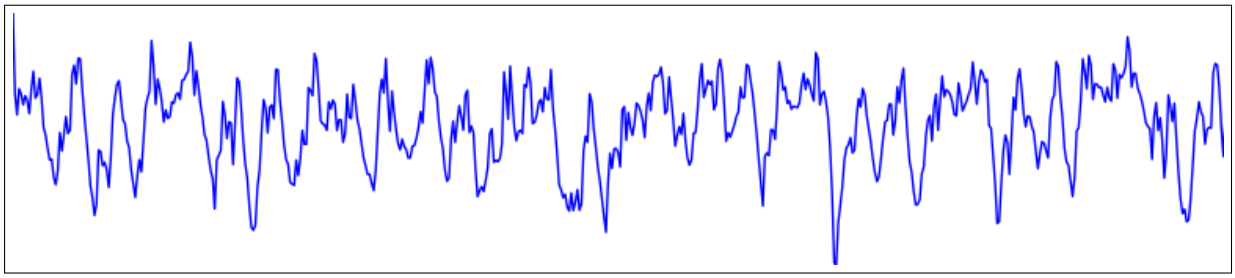
The following pairs of figures show four spectrograms, each one corresponding to a sample from each genre, along with the outputs of a filter in the first convolutional layer of the network which were obtained by processing these spectrograms. As the filter is convolved with the image along the time axis, a higher output value signifies that the filter is overlapping a region in the input which is more similar to the feature it has learned.

Consequently, this filter seems to be modelling the information contained in low frequencies (present at the bottom of the spectrogram). This can be seen most clearly by looking at the example from the metal genre and the corresponding output in Figure 4.23. A darker shade at a location in the spectrogram corresponds to a higher magnitude of the frequency being represented.

Musical genres exhibit different characteristics for low frequencies (for example, the bass line in a disco song, compared with the one in a classical track). Therefore, this feature, which has been learned by the convolutional layer, is useful for distinguishing genres.

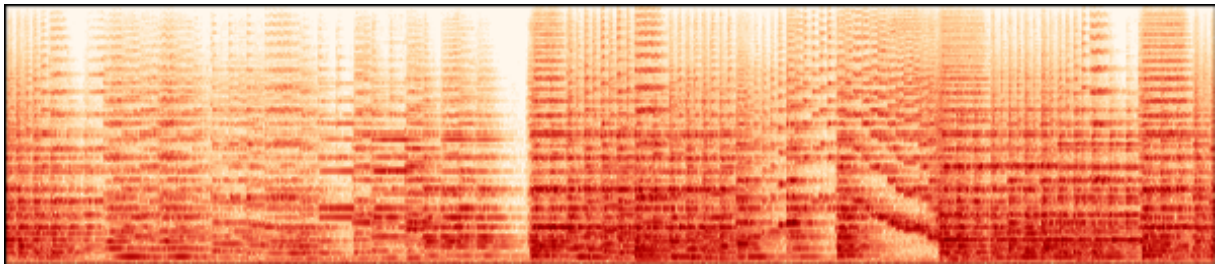


(a) Spectrogram for the 3^{rd} example in the blues dataset.

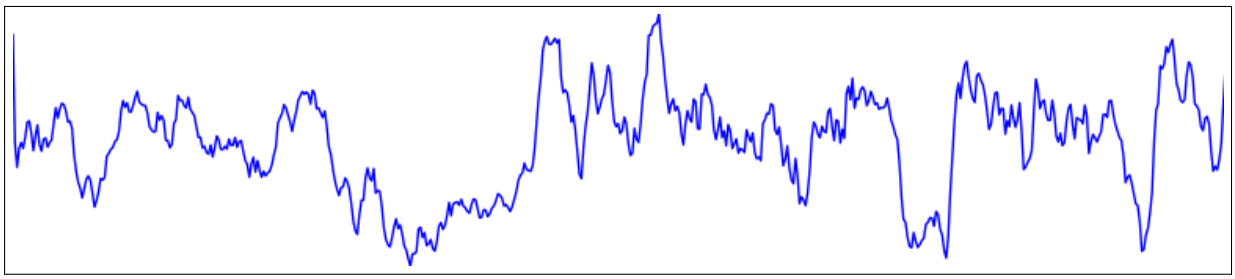


(b) Output of the filter.

Figure 4.20: Blues example.

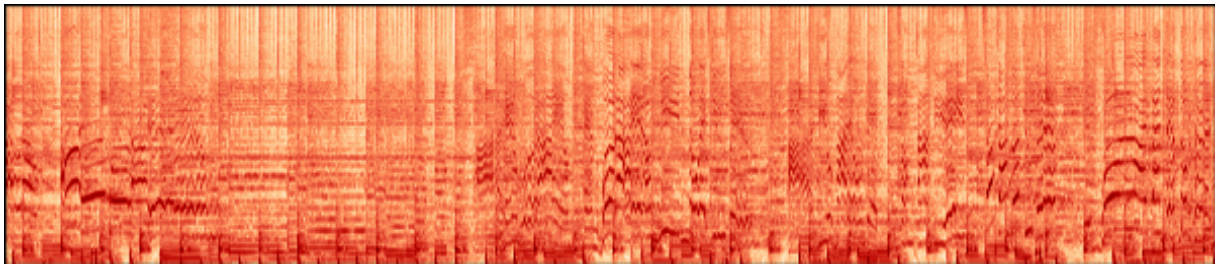


(a) Spectrogram for the 20^{th} example in the classical dataset.

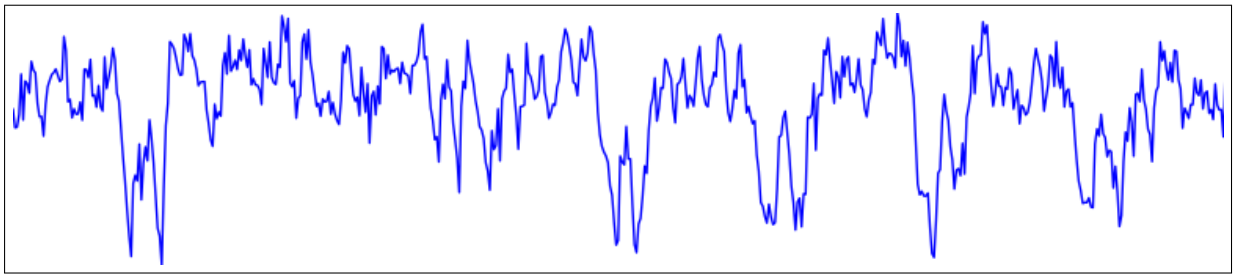


(b) Output of the filter.

Figure 4.21: Classical example.

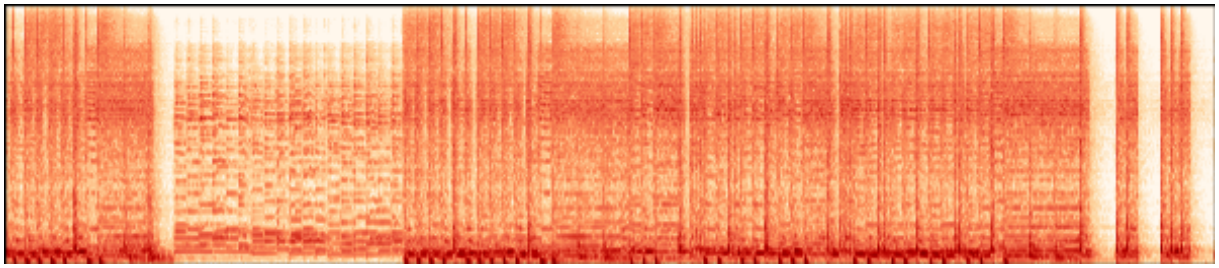


(a) Spectrogram for the 38^{th} example in the disco dataset.

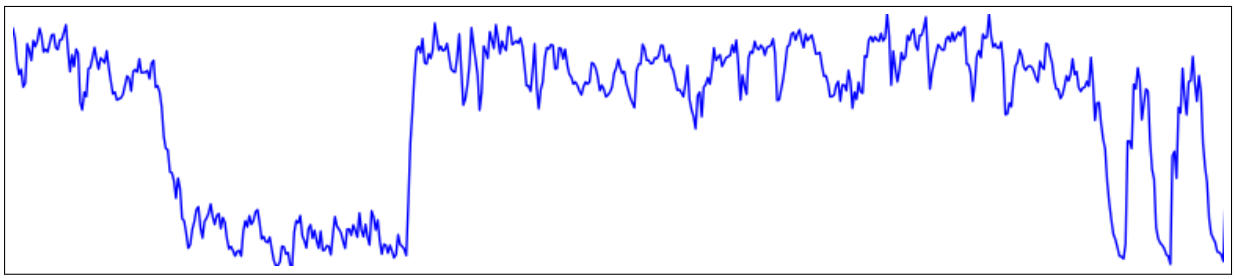


(b) Output of the filter.

Figure 4.22: Disco example.



(a) Spectrogram for the 38^{th} example in the metal dataset.



(b) Output of the filter.

Figure 4.23: Metal example.

4.6 Summary

This chapter has presented the results obtained from the evaluation of the classification system. The accuracy values for two, four and six genres illustrated that a neural network with a large number of parameters requires more training data as the number of classes increases, otherwise gradually losing its generalisation ability. Nevertheless, the error rates obtained showed that the classifier generalised well, despite having a very limited amount of training data available.

Moreover, the performance evaluation illustrated the amount of parallelism which can be extracted from different types of layer operations. The best improvements were noticed, as expected, in the convolutional and max pooling layer operations. In some cases, the GTX980M produced some surprising results, not proving to be faster than the GT640M. This is possibly due to the other laptop having different characteristics, such as the width of the bus that transfers data to and from the GPU.

Finally, the analysis of a filter from a convolutional layer revealed that the neural network is able to learn about the content of audio signals from their spectrograms and use this information to distinguish musical genres.

Chapter 5

Conclusions

5.1 Summary of achievements

The core objectives of this project involved developing a classification system based on a convolutional neural network with CUDA parallelisation of layer operations and achieving better-than-chance prediction on two musical genres.

These goals have been successfully accomplished—the implemented system can run forward- and back-propagation on a NVIDIA GPU as expected and is able to minimise the objective function when training the classifier. In addition, the mean probability of error for two-genre classification was equal to 0.8% and corresponds to an excellent prediction ability.

The performance evaluation showed which implementation is optimal for each type of layer, depending on the matrix size. As a result, the overall neural network computation can always be optimised in terms of runtime, combining the modular CPU and GPU implementations. The largest speedups were obtained for the convolutional and max pooling layer operations—up to approximately 10x and 100x, respectively.

Following the achievement of the core objectives, I have also completed three extensions which were initially proposed:

1. obtaining a better accuracy for two genres (implicitly achieved by the result above);
2. performing better-than-chance classification on more than two genres (the accuracy values were equal to 80% for 4 genres and 69% for 6 genres, respectively);
3. analysing what filters learn in the convolutional layers (shown in section 4.5).

5.2 Future work

As previously discussed in section 2.4, the GTZAN dataset I used for training the classifier represented the main risk of this project. The size of the training set limited the

generalisation ability of the classification system when increasing the number of genres to be recognised. The accuracy values for 4 and 6 classes were good, although it could be seen that only the classical and metal genres maintained similar error rates while the others were increasing. Using a larger dataset would therefore help the classifier learn a better, more complex model of the data and lower the confusion rates between genres.

Obtaining more data would also allow other optimisation algorithms to be implemented. For example, instead of splitting the dataset into training and test sets, an additional *validation* partition could be used. This would allow the training algorithm to be run on different values for a given hyperparameter (for example, the number of neurons in a fully-connected layer or the number of filters in a convolutional layer—values which were manually set in this project). Then, the best values could be chosen by comparing the performances of the variant classifiers.

5.3 Final remarks

In addition to the successful completion of this project, I have realised that many of my technical skills have progressed substantially throughout its course. As well as having gained practical experience with machine learning, I have used a complex algorithm to tackle a problem related to music information retrieval, a vital and exciting field nowadays. My software engineering discipline has also developed while designing and implementing the classification system. Finally, working on this dissertation has brought significant improvement to my technical writing abilities and to the manner in which I present my work.

Bibliography

[1] Sander Dieleman. Recommending music on Spotify with deep learning. http://benanne.github.io/2014/08/05/spotify-cnns, 2014.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. Book in preparation for MIT Press, 2016.

[3] Sean Holden. Artificial Intelligence II, 2002-16. University of Cambridge Part II CST lecture notes.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[5] Markus Kuhn. Digital Signal Processing, 2015. University of Cambridge Part II CST lecture notes.

[6] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[7] Nvidia. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2015.

[8] Douglas O'Shaughnessy. *Speech communication: human and machine*. Universities press, 1987.

[9] Nicolas Scaringella, Giorgio Zoia, and Daniel Mlynek. Automatic genre classification of music content: a survey. *Signal Processing Magazine, IEEE*, 23(2):133–141, 2006.

[10] Bob L Sturm. A survey of evaluation in music genre recognition. In *Adaptive Multimedia Retrieval: Semantics, Context, and Adaptation*, pages 29–66. Springer, 2012.

[11] The Spotify Team. 20 Million Reasons to Say Thanks. https://news.spotify.com/uk/2015/06/10/20-million-reasons-to-say-thanks/, 2015.

[12] George Tzanetakis and Perry Cook. Musical genre classification of audio signals. *Speech and Audio Processing, IEEE transactions on*, 10(5):293–302, 2002.

[13] Aaron Van den Oord, Sander Dieleman, and Benjamin Schrauwen. Deep content-based music recommendation. In *Advances in Neural Information Processing Systems*, pages 2643–2651, 2013.

Appendix A

Project Proposal

Catalina Cangea
Murray Edwards College
ccc53

Computer Science Part II Project Proposal

Music Genre Classification using Convolutional Neural Networks

14th October 2015

Project Originator: *Catalina Cangea*

Resources Required: *None*

Project Supervisor: *Dr Sean Holden*

Director of Studies: *Dr David Chisnall*

Overseers: *Prof Jean Bacon* and *Prof Ross Anderson*

Introduction and Description of the Work

Online music streaming services such as Spotify are widely used nowadays — their ability to recommend music and create playlists based on genre or mood similarities has a significant effect on the users' degree of contentment. This functionality can be achieved through collaborative filtering, where consumption patterns of users are examined to determine possible recommendations. However, this strategy might result in a bias towards popular songs as opposed to less popular or newer ones. Since listening to a song remains the best way to decide whether one enjoys it, a good idea is to analyse the audio signal and base predictions on its content.
The main challenge of performing genre classification lies in the feature extraction phase, as the final results are crucially influenced by the features which have been retrieved from examining the audio signal. Convolutional neural networks (CNNs) do not present this problem — they only require a representation of the raw input (e.g. spectrogram of the audio file) in order to learn increasingly complex musical features and predict genre. CNNs have been widely used in image information retrieval tasks but are also highly suitable for discovering the features which might otherwise not be straightforward to design.
This project aims to develop a classification system which predicts the music genre of an audio file, based on its contents. In order to achieve faster training time, the CNN will run on a GPU, so the implementation is most likely to involve the CUDA® programming model and GPU-accelerated libraries. For training and testing the network, I will be using the GTZAN Genre Collection dataset. A possible extension of the project will investigate some of the features which are learnt by different layers of the network.

Starting Point

I will be approaching this project with the following prior experience and knowledge:

- programming experience with C and C++ obtained from high-school and the Part IB *Programming in C and C++* course
- basic machine learning concepts from the Part IB *Artificial Intelligence I* course (but no practical experience)

In order to complete the project, I will need to acquire the following further knowledge:

- the underlying theory of convolutional neural networks
- using the chosen audio signal processing framework(s)
- working with CUDA® and any additional GPU-accelerated libraries
- the material from the Part II *Digital Signal Processing* course relevant for input processing

The GTZAN dataset will be used to train and test the convolutional neural network — it consists of 1000 tracks in *.wav* format which provide a sufficiently comprehensive sample of songs belonging to 10 musical genres. This dataset is the most widely used in genre recognition research, while also being a public and easily accessible resource.

Substance and Structure of the Project

The first step towards performing genre classification is converting the audio data to a suitable representation for the network input. Then, the central part of the project consists of implementing the convolutional neural network with CUDA® optimisations for running on an NVIDIA® GPU, followed by training and testing it with the chosen dataset.
I propose the following project structure:

1. Research convolutional neural networks and decide on a representation of the audio files for the network input.
2. Gain decent familiarity with suitable libraries for audio signal processing, CUDA® implementation and GPU acceleration.
3. Extract input representation of audio files.
4. Implement the convolutional neural network with CUDA® optimisations.
5. Train and test the network using the GTZAN dataset, expecting a better-than-chance prediction.

In the event that the classification performed by the convolutional neural network is not effective, data mining software such as Weka may be used to achieve this instead.

Success Criteria

The core part of the project will be considered successful upon completion of the following items:

1. A working CUDA® implementation of a convolutional neural network.
2. The ability of the network to perform better-than-chance prediction on a dataset containing two genres.

Evaluation Strategy

1. The implementation of the convolutional neural network can be tested using the GTZAN Genre Collection dataset.

2. *k-fold cross-validation* may serve as a model validation technique. The dataset is partitioned into k equal subsets, each of them being used as testing data for one round of cross-validation. The results are then averaged over all k rounds.

3. The running times on GPU and CPU can be used to determine the achieved speed-up.

4. An analysis may be performed on how the running time scales with the number of GPU processes assigned.

Extensions

Further goals can be achieved if time allows:

1. Achieving similar confidence when extending the dataset to contain more genres.

2. Modifying the architecture of the network with a view to improving the classification accuracy, in case the initial genre prediction does not turn out to be significantly better than chance.

3. Investigating what types of musical features are being learnt by neurons in different layers of the network; recommending songs similar to one which produces a specific activation in a neuron — the similarity will be based on the learnt feature.

4. Comparing the performance of the convolutional neural network against the one obtained by another classification method which requires extracting features from the audio signal.

Timetable and Milestones

Weeks 1 & 2 (Oct 24 — Nov 6)

Research convolutional neural networks. Determine which libraries to use for audio input processing, CUDA® implementation and GPU optimisation.
Milestone: Theory behind CNNs is understood and ready to be applied in the project.

Weeks 3 & 4 (Nov 7 — Nov 20)

Gain familiarity with libraries. Decide on a suitable format of the audio files for the network input and implement its extraction from the raw signal.
Milestone: Knowledge of the needed library functionalities is acquired. Signal processing implementation is complete.

Weeks 5 & 6 (Nov 21 — Dec 4)

Implement the convolutional neural network.
Milestone: A working implementation of the CNN is completed.

Weeks 7 & 8 (Dec 5 — Dec 18)

Slack time in case of unplanned delays. Start training the network and testing its performance.

Weeks 9 & 10 (Dec 19 — Jan 1)

Finish testing the network's performance.
Milestone: The CNN provides a better-than-chance prediction of music genre.

Weeks 11 & 12 (Jan 2 — Jan 15)

Improve the architecture of the network to achieve more accurate genre prediction. Test and train for as many adjustments as time allows.
Milestone: The CNN has an improved classification accuracy. Training and testing are completed.

Weeks 13 & 14 (Jan 16 — Jan 29)

Slack time in case of unplanned delays. Write the progress report and the presentation.
Milestone: Progress report is completed and submitted before January 29^{th}. Presentation is ready.

Weeks 15 & 16 (Jan 30 — Feb 12)

Recommend songs based on similar activation values for a few chosen neurons in different layers of the network.
Milestone: Have recommendations for songs exhibiting musical features of different complexities.

Weeks 17 & 18 (Feb 13 — Feb 26)

Write the Introduction section of the dissertation.
Milestone: Introduction section is completed.

Weeks 19 & 20 (Feb 27 — Mar 11)

Write the Preparation section of the dissertation.
Milestone: Preparation section is completed.

Weeks 21 & 22 (Mar 12 — Mar 25)

Write the Implementation section. Present the current draft to the project supervisor and Director of Studies.
Milestone: Implementation section is completed.

Weeks 23 & 24 (Mar 26 — Apr 8)

Slack time in case of unplanned delays. Resolve any comments received.
Milestone: Introduction, Preparation and Implementation sections are completed and corrected.

Weeks 25 & 26 (Apr 9 — Apr 22)

Write the Evaluation section. Start writing the Conclusions section.
Milestone: Evaluation section is completed.

Weeks 27 & 28 (Apr 23 — May 6)

Finish writing the Conclusions section. Send the completed dissertation to the supervisor and Director of Studies. Resolve all received comments.
Milestone: Dissertation is completed and corrected.

Week 29 (May 7 — May 13)

Slack time for any final changes to the written sections. Obtain approval for the dissertation from the supervisor and Director of Studies.
Milestone: Final version of dissertation is approved and submitted before May 13$^{\text{th}}$.

Resources Declaration

I will be using my laptop (Acer Aspire V3-571G, Intel i7, 4GB RAM, NVIDIA® GeForce® GT 640M) which is running a dual boot of Ubuntu 14.04 and Windows 7. Should it fail, I will switch to the MCS machines. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.
In case training the convolutional neural networks on my laptop's GPU requires too much computational time, Amazon Elastic Compute Cloud services can be used instead. I will choose the EC2 G2 instance, as it offers NVIDIA® GPUs with 1536 cores and 4GB memory.
Daily backups of project files will be made to an external hard drive and Dropbox. I will keep the project code in a repository on my laptop, using `git` for version control; the code will also be pushed periodically to MCS and a GitHub repository.