Music Genre Detection

with Neural Networks

Laurens C. F. Van Mieghem

Technische Universiteit Delft

TU Delft

Music Genre Detection

with Neural Networks

by

Laurens C. F. Van Mieghem

Student number: 4680669
Project duration: April 12, 2012 – June 9, 2020
Thesis committee: Dr. Ir. R. van der Toorn, TU Delft, supervisor
Drs. E. M. van Elderen, TU Delft
Dr. C. Kraaikamp, TU Delft

TU Delft

Abstract

In this thesis we classify samples of music according to the genre that the music belongs to using neural networks. We divide this task into four parts.
In the first part, we prepare the audio files to be used as input to a neural network. Specifically, we examine ways to create spectrograms. We then optimise the spectrograms by reducing and normalising them.
The second part consists of theoretical information regarding neural networks. We initially look at the perceptron, the building blocks of any neural network, and then extend this notion to various networks, such as the multilayer perceptron, the recurrent network and the convolutional network.
In the third part, we apply the theoretical knowledge that we gained in part two and implement a standard neural network, a recurrent neural network, a convolutional neural network, and a combination of recurrent and convolutional neural networks. We examine various network structures and we evaluate them based on what the networks can learn, how fast they can learn it and how accurate its classifications are. Simultaneously, we focus on creating an efficient network, using the fewest amounts of computational resources possible. We train each of the networks with data that we created in part one, and compare the performance of the networks with each other. In terms of accuracy and loss measures, we find that the best performing network is the combination of the recurrent and convolutional neural network. This network is able to determine which of six considered genres a 3 second sound sample belongs to with an accuracy of 90%. However, in terms of computational resources required to train the models, the convolutional neural network with many kernels during training converges using least computational cost. We then experiment with the amount of kernels in the convolutional layers, and find that a layer with many kernels learns faster, but does not necessarily yield better results. This is because networks with fewer kernels eventually learn the same kernels that are significant. Finally, we consider the impact of varying sound sample lengths on the performance of the networks. For a 1 second sound sample, we see that the recurrent network outperforms the other networks in terms of accuracy of the predictions. However, the larger we make the sound samples between 1 and 3 seconds, the better each network performs. Part four consists of an explanation of each component in the system, and presents a complete system built with Google's TensorFlow [15], in which all the components work together to create an end-to-end classification system.

Contents

1

Artificial Intelligence & Mathematics

Artificial intelligence, with neural networks as a sub-discipline, is a relatively new field that gained feasibility with the rise of computational power and the massive scale of data collection solutions. The artificial neural network is inspired by the biological neural networks that constitute the brain. The systems 'learn' to perform tasks by considering examples, generally without being programmed with task-specific rules. Such rules and definitions are a cornerstone of traditional mathematics such as analysis, and their absence is challenging, if not provoking, from a traditional mathematical point of view.

However, the task that we assign neural networks to complete is essentially a mapping from an input to an output. In this report, this input is audio data, and the output is a probability distribution over a finite set of music genres. In traditional mathematics, it would be difficult to create such an abstract mapping, as defining mathematical rules for music genres is already unattainable.

We could therefore see neural networks as a new field in mathematics, which, as we will see, has a strong connection to traditional mathematics. We will see elements of analysis during the training of the networks, links to thermodynamics as we let the networks calculate Boltzmann distributions, and functional analysis such as convolution when extracting patterns from input data.

We hope that readers from a mathematical background are as excited to get acquainted with this field as we are, and recognise its capability to create abstract functions using rather elementary mathematics.

2

Processing Audio Signals

Humans hear sound through vibrations that are caused by sound waves. Therefore, music is saved as sound waves, which on a computer is a discretised array of amplitude values representing the sound waves. In this chapter, we will explore a method of representing audio signals in a different form, called a spectrogram. We furthermore prepare the transformed audio data for the neural network, which we create in chapter 3.

2.1. Sample Rate

Sound are waves propagating through some medium, creating various densities at different times. Humans are able to perceive the frequencies of the waves and their amplitudes. The human ear is able to hear frequencies between approximately 20Hz - 20 kHz.
When recording music, the sound waves must be discretised in order to be stored on a computer. To this end, a tool such as a microphone measures the air pressure at every time step t and store the computer stores it into an array. The amount of samples that are recorded per second is called the *sample rate*. A higher sample rate allows us to preserve more information about the sound waves, thus being able to reconstruct the waves in more detail. It is desirable to have a sample rate which is as low as possible in order to save disk space, but high enough to lose very little information about the waves. The Nyquist theorem [26] states that if we were to sample a signal we would need samples with a frequency larger than twice the maximum frequency contained in the signal, that is

$$f_{sample} \geq 2f_{max}. \tag{2.1}$$

Since the human ear is unable to perceive frequencies larger than 20 kHz, recorded sound often has a maximum frequency slightly above this threshold. In accordance with the Nyquist theorem, many signals are sampled with a sample rate of 44.1 kHz. That is, 44100 times per second the air density is measured and stored in an array. In this thesis we exclusively work with signals of this sample rate. Changing the sample rate is possible, but requires reconstructing the sound waves and consequently resampling from this sound wave in different sample rates. We typically avoid this process as it is computationally intensive, since we must recreate a continuous function with discrete values, and consequently take new discrete samples from it.

2.2. Sound Representation

In the previous section, we argued that the sample rate of recorded sound waves should be 44.1 kHz, in which the maximum frequency present in the recording must be less than 22 kHz. In a computer, we can allocate 16 bits for an integer, which can store values in the range of -32,768 to 32,767. Suppose we want to store a 5 minute song as an array of *int16* values. We would need to allocate a total of

$$5 \cdot 60 \cdot 44100 = 13230000 \tag{2.2}$$

samples. When these samples are stored in *int16*'s, we need

$$13230000 \cdot 16 = 211680000 \text{ bits} = 26.46 \text{ MB}, \tag{2.3}$$

where MB are megabytes, a common measure of size in computer systems. Storing signals like this is called the *wave* format (.wav). When analysing the songs and preparing them for the neural network, we use the wave format.

2.3. Frequency Domain

When we have read the wave files, we can transform the sound waves to their *frequency domain*. Figure 2.1 shows a visualisation of a song in the wave format.

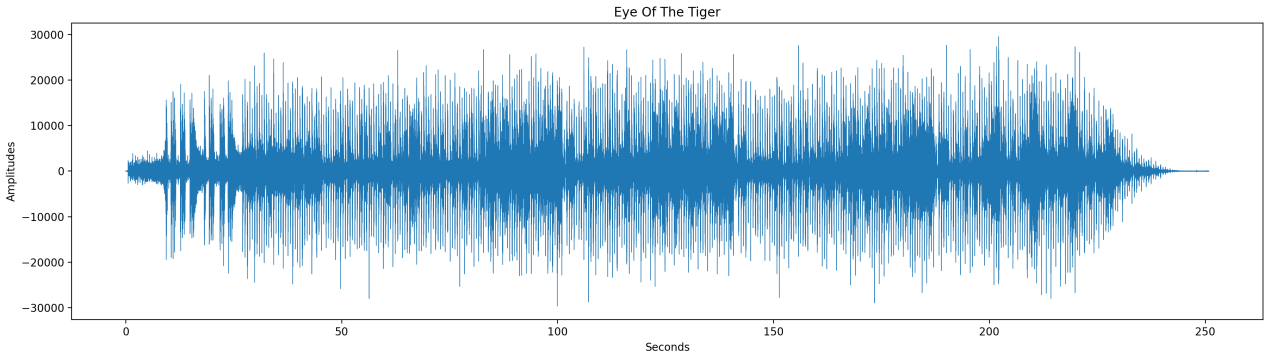

Figure 2.1: A plot of a wave file. The audio data visualised is from the song 'Eye Of The Tiger' [12]. On the horizontal scale is the time in seconds, and on the vertical scale the amplitude values.

The wave in figure 2.1 looks very complex and we are unable to see a clear structure in the sound waves. However, this wave is just a superposition of many pure sine waves, with different phases, amplitudes and frequencies (figure 2.2). We would like to decompose this complex wave into its individual sine waves, so that we can identify the frequencies present in the sound. For this we use *Fourier transforms*.

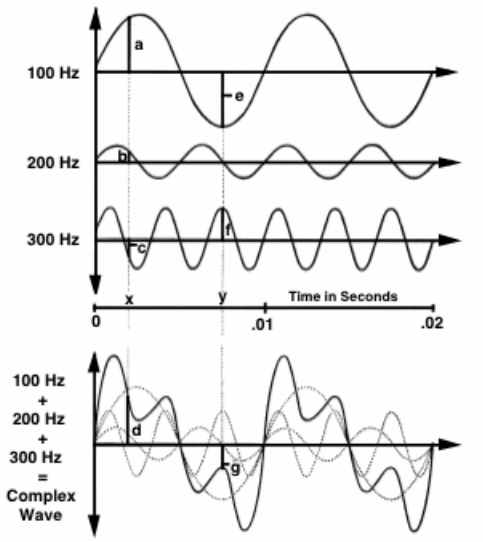

Figure 2.2: The superposition of a signal (bottom) from various sine waves with different frequency and amplitude. Adopted from [18].

2.3.1. Fourier Transform

Definition 1. *[7] In mathematics, a Fourier transform decomposes a function, such as a signal, into its*

constituent frequencies. The Fourier transform of $f(x)$ is defined as

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-i\xi x} dx. \tag{2.4}$$

Using Euler's formula

$$e^{ix} = \cos(x) + i\sin(x) \tag{2.5}$$

we see that the Fourier transform is essentially a composition of simple sine and cosine waves, weighted by the function f. Figure 2.3 illustrates the Fourier transform of a time signal into its frequency domain.

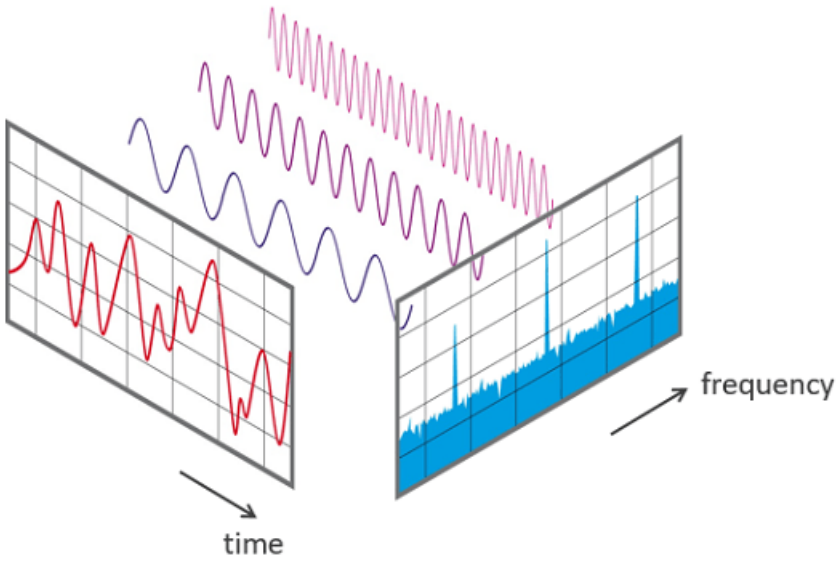

Figure 2.3: A visual representation of the Fourier transform. Adopted from [32].

In our scenario, we are not interested in knowing the Fourier transform of a continuous function, but rather want to transform an array of values. We have a discrete set of samples which we want to represent in the frequency domain. For this, we use the *discrete Fourier transform*, which is a discretisation of definition 1:

$$D_k(z) = \sum_{j=0}^{N-1} z_j e^{\frac{-2\pi i j k}{N}} \qquad k = 0 \ldots N-1$$

where $z := z_0, z_1, \ldots, z_{N-1}$ is our original sequence of samples, and $D := D_0, D_1, \ldots, D_{N-1}$ are the transformed samples. Evaluation of equation $k = 0 \ldots N-1$ for each sample of the song is computationally intensive, even when exponential factors are precomputed. Even though computational cost can be reduced by employing a fast Fourier transform (FFT) algorithm [5, 6, 30], computing the Fourier transform of an entire song (≈ 15 million samples) is expensive and thus something we want to avoid.

2.3.2. Spectrograms

In the previous section we have seen how to transform a time signal into its frequency domain. We are interested in the frequencies, because they reveal a lot about the composition of the song, and which musical instruments are likely to be present. Similar to how we use neural networks to replicate the human's biological neural network in the brain, we use a *spectrogram* in an attempt to replicate the sensory perception of humans. A spectrogram is a visual representation of the frequency spectrum of a signal as it changes through time.
Composition of a spectrogram in done in the following way, illustrated in figure 2.4:

1. We divide the time signal into subsets $A_1, \ldots A_n$ with equal length Δt.

2. For each subset A_i, we calculate the Fourier transform to obtain its frequency spectrum D_i, represented as a vector with the different frequencies as dimension and the values as intensities per frequency.

3. We then concatenate all the frequency vectors D_i in time order to create a matrix S, the spectrogram, with dimension $f_{max} \times n$, where f_{max} is the maximum frequency in the spectrum and n the amount of subsets.

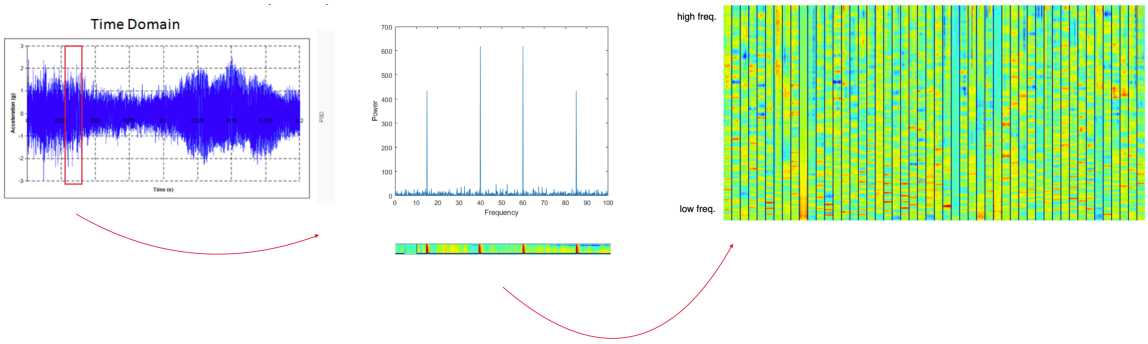

Figure 2.4: The composition of a spectrogram. We start on the left figure, where we take a subset of the time signal. We compute the Fourier transform and visualise the frequency spectrum as a heat map bar. We do this for all the consecutive time signal subsets and order them in time on the spectrogram.

Following this procedure we obtain a matrix with on the horizontal axis the time steps of Δt, and on the vertical axis the frequencies.

2.4. Establishing Trainability

Training neural networks can be a computationally expensive process. Larger dimensions of our input data in the network results in the need for a larger network to process this data. Scaling the network involves adding more parameters that have to be trained. Although it has been demonstrated that neural networks are capable of processing large input data [2], our aim is to develop a system in which performance is optimal, but reduces the computational cost as much as possible.
To this end, we decide to extract subsets of songs with a time period of 3 seconds. Heuristically, humans need not hear the entire song to determine the genre. Usually, a few seconds is enough. Therefore, we design the system in a similar way to human perception. In section 4.8, we will experiment with this time period and discover how the length of the subsets affect the performance of the system. The remaining part of the song, which is not included in a subset, is discarded. Using this method, we are able to extract multiple subsets from one song, enlarging our training set significantly. We take about $10-15$ random subsets of each song, where we do not include the first and last 30 seconds of the song, as these sections of the song might not be representative for the genre itself. When classifying new songs, we can follow the same procedure as in the training stage, where we extract multiple subsets of 3 seconds. When classifying each of these subsets and taking the maximum of the classifications, we increase the robustness of the classification.

2.4.1. Reducing the spectrogram matrix

For each of the 3 second sound samples we extract from the songs, we compute the spectrogram as described in section 2.3.2. Before doing this, we must determine an appropriate value of Δt so that different elements of the sound sample are located in at least one unique subset of the sample. We choose Δt to be 200ms, of which 100ms overlaps with the consecutive sample. According to an article written by an oscilloscope manufacturer [31], overlapping enables much greater visibility of frequency changes with time. A similar approach can be found in Lee et al. [22] on speech data. A Δt of 200ms with 100ms overlap is reasonable when considering that a 16th note, one of the shortest notes in music

theory, is about 150ms at 100bpm (beats per minute). Smaller values of Δt make the spectrogram matrix larger which increases computation cost. Using a 200ms Δt on a 3 second sound sample results in 29 time steps, and hence the amount of columns in the spectrogram matrix.

We can also reduce the amount of frequencies by creating *frequency bands*. To this end, we group frequencies and calculate a statistical measure over this group. Previous research [4, 22] suggests to use a technique called *principal component analysis*, which aims to reduce dimensionality by creating orthogonal variables. However, in this report we consider *max pooling*, as it is computationally less expensive than principal component analysis. When max pooling, we essentially replace each frequency band with the maximum of all frequencies present in the band. This procedure increases *translation invariance*, which is further discussed in section 3.4.6. We can greatly reduce our spectrogram size by subdividing the frequencies into 12 frequency bands, creating 4 bands for low, mid and high frequencies.

Performing these reduction measures we obtain a spectrogram which has dimension 12×29.

2.4.2. Normalisation

A study by Salimans and Kingma [21] has revealed that normalisation in neural networks accelerates their training speed, as the convergence of stochastic gradient descent (section 3.1.2) is sped up. Therefore, the final part of processing the spectrogram is to normalise it using

$$Z_{ij} = \frac{S_{ij} - \min S}{\max S - \min S}, \tag{2.6}$$

where $\min S$ is the minimum element in the spectrograms S, and $\max S$ the maximum element. For all elements $z_{ij} \in Z$ we have $z_{ij} \in [0, 1]$. Performing normalisation on all the samples results in uniform data. For instance, quieter and louder songs both get scaled to the same level of frequency intensities between 0 and 1. This way, the system cannot classify songs based on their loudness, which increases its robustness.

3

Neural Networks

In this chapter we examine what neural networks essentially are, and how they can be used to classify real-world observations. In our case we are interested specifically in how neural networks can aid us in determining the genre of music using only the audio data. To this end, we must first understand the foundation of any neural network, *the perceptron*. We then extend this notion to a multilayer perceptron, in which we create a network using multiple perceptrons. Consequently, we construct two networks using perceptrons: *The recurrent network* and the *convolutional neural network*. Once we got familiar with these networks, we compare them with each other in chapter 4 using sound sample representations created in chapter 2.

3.1. The Perceptron

Nature has inspired humans countless inventions. Just as dolphins inspired the invention of SONAR, neural networks are inspired by the brain's architecture [33]. The perceptron can be seen as a neuron in the brain. A neuron is a specialised cell designed to transmit information to other cells. Similarly, the perceptron takes an input, performs an arbitrary calculation and returns an output. In mathematics, a perceptron can be seen as a function taking an input $\vec{x}$ and performing an operation g on it, outputting $g(\vec{x})$.

A perceptron can have multiple inputs. Each input is multiplied by its corresponding weight. Then, all inputs are summed and passed into the *activation function*, a function which determines what the output of the perceptron should be. Figure 3.1 gives a visual representation of this process. Each of the weights give a certain relevance to their corresponding input variable. For instance, let x_i be a very relevant input variable. Its weight should be higher than a more irrelevant variable x_j. In an equation,

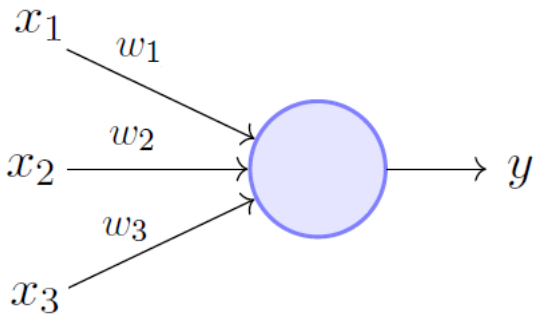

Figure 3.1: A visual representation of a perceptron. Adopted from [29].

the operations of a single perceptron can be denoted as

$$\hat{y} = f\left(\sum_{i=0}^{n} w_i x_i\right) \tag{3.1}$$

where $\hat{y}$ is the output, $\vec{x}$ the input, $\vec{w}$ the weights and f the activation function.

3.1.1. Activation Functions

An activation function in a perceptron can be seen as the rate of action potential firing in a neuron. In theory, any function can serve as an activation function. For instance, we could take f to be the Heaviside step function

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}. \tag{3.2}$$

In this case, the neuron will output 0 when $\sum_{i=0}^{n} w_i x_i < 0$ and 1 else. However, there are a two requirements that an activation must satisfy in order to be useful in perceptrons.

Most importantly, the activation function should be *non-linear*. This means that the output cannot be reproduced from a linear combination of inputs, which in turn allows the output to depend non-linearly on its inputs. Many problems that we attempt to solve with neural networks do not have a linear solution, and therefore using non-linear activation functions it is in theory possible to estimate any function.

Secondly, the activation function should be differentiable on preferably its entire domain. This allows us to define a loss function $J(\vec{w})$ that we can minimise in order to update our weights. By minimising the loss function, we can update the weights $\vec{w}$ such that we minimise the error between the expected output y and the actual output $\hat{y}$. Many present-day training utilities for neural networks use *gradient descent* for this process.

3.1.2. Gradient Descent

In section 3.1.1 we required that the activation function be differentiable. This raises the question what we need this derivative for. At the start of section 3.1 we defined a vector of weights, $\vec{w}$. These weights are initialised arbitrary, but should eventually represent the relevance of their corresponding input variables. Hence, the weights have to be updated based on the output that the perceptron generates.

In order to update the weights, we need a measure that determines how far away the output of the perceptron is from the desired output. Using this error measure, we can then update the weights in order to minimise this error. We define the error to be the sum of squared errors. We call this error the loss of the model, and refer to this function as the loss function.

$$J(\vec{w}) = \frac{1}{2}\left(y - \hat{y}\right)^2, \tag{3.3}$$

where y is the target or desired output, and $\hat{y}$ the actual output of the perceptron. In theory, we calculate the loss for each sample that we pass into the network. We then alter parameters in the neural network in order to decrease the loss for the consecutive samples. In practise, we calculate the loss per batch. A batch is a collection of multiple samples, often 32. Calculating the loss and the gradient on batches improves training speed. Note that the $\frac{1}{2}$ term is used for convenience to derive the gradient, as we will see shortly.

Consider the following convex function $J(\theta)$ illustrated in figure 3.2, where at the start of a step in the gradient descent process, our location is the light blue dot. Note that we are looking for the minimum of this function, since J is the loss function which we want to minimise. At each step, we calculate the gradient of the loss function (equation 3.3). We then move a factor η in the opposite direction of the gradient, ending up at the dark blue dot. The factor η is called the *learning rate*. Depending on this factor, we can descent faster or slower, depending on the landscape of the loss function.

We can now derive this process mathematically. We must find the updated weights $\vec{w}_{new} = \vec{w} + \Delta\vec{w}$,

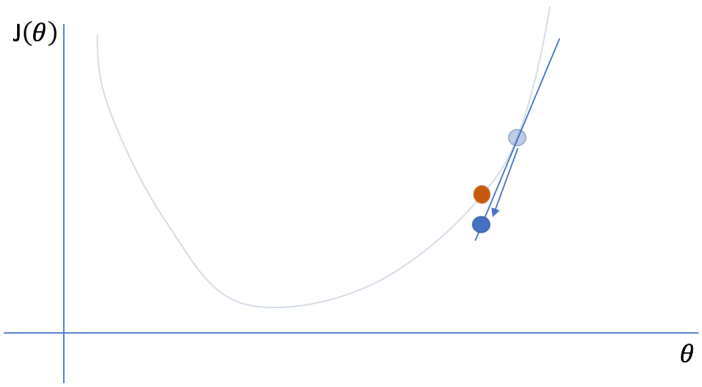

Figure 3.2: A visual representation of one step in the gradient descent process. Adopted from [19].

where $\Delta\vec{w} = -\eta\nabla J(\vec{w})$. The partial derivatives of $J(\vec{w})$ are then calculated as follows

$$\begin{aligned}\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j}\frac{1}{2}(y-\hat{y})^2 \\ &= \frac{1}{2}\frac{\partial}{\partial w_j}(y-\hat{y})^2 \\ &= (y-\hat{y})\frac{\partial}{\partial w_j}\left[y - f\left(\sum_i w_i x_i\right)\right] \\ &= (\hat{y}-y)x_j\frac{\partial}{\partial w_j} f\left(\sum_i w_i x_i\right)\end{aligned}$$

And for each weight w_j we have

$$w_j := w_j + \Delta w_j = w_j - \eta\frac{\partial J}{\partial w_j} = \eta(\hat{y}-y)x_j\frac{\partial}{\partial w_j} f\left(\sum_i w_i x_i\right). \tag{3.4}$$

3.1.3. The Bias

The final characteristic of the perceptron which we have not discussed yet is the bias. Each perceptron has a bias, which is a constant term b used to adjust the output along with the weighted sum of the inputs. Including the bias equation 3.1 becomes

$$\hat{y} = f\left(\sum_{i=0}^{n} w_i x_i + b\right). \tag{3.5}$$

Indeed, the input now represents an affine function

$$h(\vec{x}) = \vec{w}^T\vec{x} + b \tag{3.6}$$

where the bias acts as the intercept. The perceptron now has an additional parameter which allows the input to shift the activation function, resulting in a better ability to fit the target function. Note that the bias is updated using gradient descent in the same way all the weights are updated, using an additional weight w_{n+1} for the bias.

3.2. Multilayer Perceptron

Now that we have seen in section 3.1 the characteristics of a single perceptron, we can extend this notion to *multilayer perceptrons*, which is essentially a network of multiple perceptrons where outputs of certain perceptrons could serve as inputs for other perceptrons. Figure 3.3 gives a visual representation of a multilayer perceptron, where each node is a perceptron.

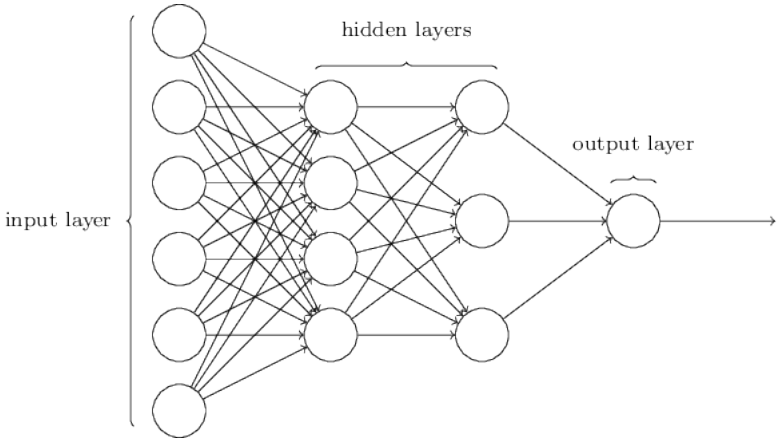

Figure 3.3: A visual representation of a multilayer perceptron. Adopted from [8].

A multilayer perceptron consists of layers, sets of perceptrons which are usually not connected with each other. The input set of perceptrons is called the *input layer*, and the output set of perceptrons is called the *output layer*. All other layers are denoted as *hidden layers*.

3.2.1. Dense Layers

When using the notion of layers, we can have many perceptrons in one layer be connected to many perceptrons in the consecutive layer. An example is illustrated in figure 3.3, where the first two layers are densely connected. These types of layers are called *dense layers*. Say that the first layer has m perceptrons, and the second layer has n. We then denote the connection between perceptron i and j as W_{ij}, where W is an $m \times n$ matrix. If such a connection exist, the value W_{ij} is the weight that perceptron i contributes to the input of perceptron j. If there is no connection between i and j, $W_{ij} = 0$. Analogously to the case in which we have a single perceptron, the task of the neural network is to learn the matrix W using gradient descent. Using the matrix W, we can write equation 3.5 as

$$y_i = f_i\left(\sum_j W_{ij}x_j + b_j\right), \tag{3.7}$$

where f_i is the activation function of perceptron i, and y_i the output of perceptron i. We generally use the same activation function for each perceptron in a layer, in which case $f_j = f_i$ for all $i, j \in \{1, \dots, n\}$

3.2.2. Activation Functions

In section 3.1.1 we discussed the relevance of an activation function, namely it's non-linearity. In this section we present a few popular activation functions which we use throughout the development of various neural networks. Each of the activation functions has its benefits in certain scenario's and drawbacks in others.

- **Sigmoid function**. The sigmoid function is a differentiable function bounded on $(0, 1)$.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.8}$$

A sigmoid essentially 'squashes' its input to a value between 0 and 1, making it a useful function to output a Bernoulli probability. However, when back propagating through previous layers of the

neural network the sigmoid can cause the gradient to converge to zero way faster than desirable, especially when the network has many layers. This is due to the fact that the derivative of the sigmoid, defined as $\sigma(x)(1-\sigma(x))$, is always smaller than 1. Repeated multiplication of this derivative when performing gradient descent causes the gradient to get very small. When the gradient is very small, the layers will not be updated properly, causing the network to be unable to learn. This issue is called the *vanishing gradient problem*.

- **Hyperbolic Tangent (tanh)**. The hyperbolic tangent is very similar to the sigmoid, except it's range is $(-1, 1)$. It is therefore able to output negative values, indicating inverse relationships. Similarly to the sigmoid, it is prone to the vanishing gradient issue.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.9}$$

- **Rectified Linear Unit (ReLU)**. The rectified linear unit is a simple non-linear unbounded function. Which is very easy to compute. Its derivative for $x > 0$ is always 1, which makes it immune to the vanishing or *exploding gradient problem*, in which the latter is where the gradient blows up through repetitive multiplication with a value larger than 1. This rarely happens.

$$ReLU(x) = \max\{0, x\} \tag{3.10}$$

  Theoretically, the downside about this activation function is that it does not have a derivative at 0. In practise, however, this does not seem to be an issue.

- **Softmax**. The softmax function is a function that takes a vector as input and normalises it into a probability distribution. The softmax function is nearly identical to the *Boltzmann distribution* which is used in statistical thermodynamics to give the probability that a system will be in a certain state given the state energy and the temperature of the system. The Boltzmann distribution is expressed in the form:

$$p_i \propto e^{-\frac{\epsilon_i}{k_B T}}, \tag{3.11}$$

  where p_i is the probability that the system is in state i, ϵ_i the energy of state i, k_B the Boltzmann constant and T the thermodynamic temperature. The thermodynamic temperature is omitted in the softmax function, but the temperature still has an intuitive meaning. For instance, high temperature in thermodynamics causes particles to have more accessible energy states, which reduces the probability of the atom being in some state i. During training, we essentially attempt to lower the temperature, comparable to a condensation process, in which we reduce the energy in the system and therefore raise the probability of a particle to be in a certain state. Hence, in the course of a successful training process, the probability distribution associated with any given sample becomes more and more articulate.
  The softmax function is given below.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{3.12}$$

  The additional benefit of using the softmax function over other normalisation functions is that the softmax function can calculate a probability distribution over input vectors with negative components. We typically use this activation function in the final layer of the model, in order to output a probability distribution of classes. In our system, the classes are the different genres that we are attempting to predict.

3.3. Recurrent Networks

A recurrent neural network (*RNN*), is a type of neural network in which sequential data can be processed. In the following sections, we examine the structure of a recurrent neural network, and more specifically, what allows it to detect temporal behaviour. Recurrent networks are frequently used in natural language processing purposes, where the current word is likely to depend on the previous words in a sentence. In our case, we are dealing with a discrete time sequence of frequency bands. Using recurrent networks, we attempt to identify rhythmic elements in the sequence of frequency bands.

3.3.1. Properties of a Recurrent Neural Network

A recurrent neural network consists of one or more *recurrent layers*. A recurrent layer has a state $\vec{s}$ which depends on the previous time step $t-1$. Using this state, the input of the recurrent layer at time t depends on its current input as well as the previous state $\vec{s}(t-1)$ [23]. Let $\vec{w}(t)$ be the input of the recurrent layer at time t, and $\vec{s}$ be the state function. Then

$$x_j(t) = w_j(t) + s_j(t-1). \tag{3.13}$$

Consequently, the state gets updated using x, the sum of the current input and the previous state, using some activation function f and weights u_{xy}.

$$s_j(t) = f\left(\sum_i u_{ji} x_i(t)\right) \tag{3.14}$$

Using the updated state we compute the output $\vec{o}$ of the recurrent layer using again an arbitrary activation function g and weights v_{xy}.

$$o_k(t) = g\left(\sum_j v_{jk} s_j(t)\right) \tag{3.15}$$

Even though $\vec{x}(t)$ only depends on the input at time t and the state at time $t-1$, we see that the recurrent relationship makes $\vec{x}(t)$ depends on all previous times. For instance:

$$x_j(t) = w_j(t) + f\left(\sum_i u_{ji} x_i(t-1)\right) \tag{3.16}$$

$$= w_j(t) + f\left(\sum_i u_{ji}\left[w_i(t-1) + s_i(t-2)\right]\right). \tag{3.17}$$

We see from equation 3.17 that $x_j(t)$ depends on $s(t-2)$.

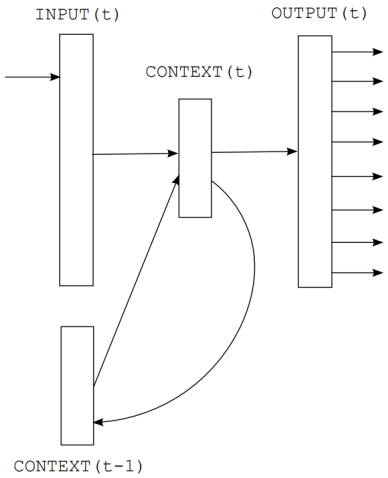

Figure 3.4: A visual representation of a recurrent layer. The output depends on a context layer, which depends in turn on both the previous context as well as the input data. Adopted from [23].

Figure 3.4 gives a visual representation of a possible recurrent layer implementation.

The recurrent neural networks can use their state s to process arbitrary sequence lengths. This makes them popular for connected tasks such as speech recognition. In our scenario, their ability

to process arbitrary sequence lengths means that we could give arbitrary lengths of sound samples without having to adjust the model. We do not explore this possibility in this report, as we only take small portions of a single song that do not have variable length.

3.3.2. Training Recurrent Neural Networks

The state layer in recurrent networks can make it unclear how to train this network using back propagation. During training, however, this recurrent relationship where the input depends on the current input as well as the past state is unrolled, creating essentially an ordinary multilayer perceptron (figure 3.5).

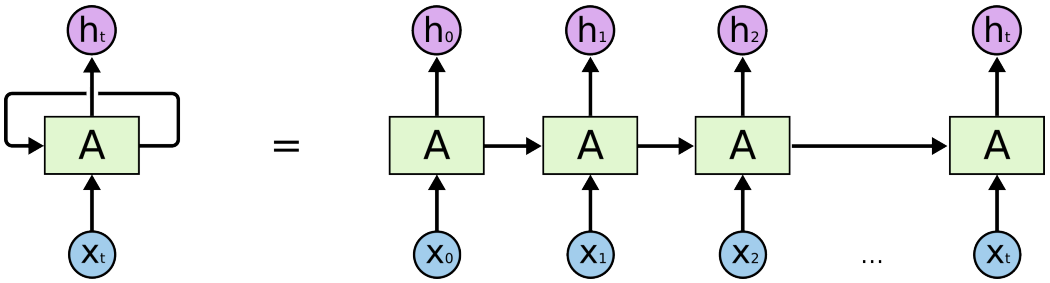

Figure 3.5: On the left a visual recurrent relation is illustrated. On the right its unrolled version. Adopted from [25].

In section 3.2.2 we discussed the vanishing gradient problem, which can occur when we back propagate gradient descent through many layers using activation functions which decrease the gradient, making it converge to 0. With no gradient, we are unable to update weights in the layers, and hence unable to minimise the loss function. The way unrolling in recurrent networks is implemented during training (section 3.3.2 makes them very prone to the vanishing gradient problem (figure 3.5). The longer the sequence is that we use as input to the recurrent neural network, the larger the unrolled network will be. Therefore, longer sequences make recurrent neural networks more difficult to train because of the vanishing gradient problem.

3.4. Convolutional Neural Networks

Another type of neural network is a convolutional neural network (*CNN*). The strength of convolutional neural networks comes from their ability to learn convolutional filters or *kernels* which can extract features from the data. Multiple connected convolutional layers allow the neural network to construct and 'recognise' objects or patterns from abstract shapes, such as curves. This property makes convolutional neural networks extremely popular for pattern recognition in images or videos. In order to understand these two dimensional convolution layers used for pattern recognition, we must understand what a convolution is.

3.4.1. Convolution

Definition 2. *A convolution is a mathematical operation on two functions that produces a third function expressing how the shape of one function is modified by the other. A convolution of f and g is written as $f * g$, and defined as*

$$(f * g)(t) = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau. \tag{3.18}$$

Intuitively, suppose you are standing in a concert hall, and you produce a sound $f(t)$. Because of the reverberation of the concert hall, at time t, you hear your sound $f(t)$, but also a linear combination of sounds at previous times: $h(0)f(t) + h(1)f(t-1) + h(2)f(t-2) + \ldots$. A continuous version of this is $f * h$, where the function $h(\tau)$ is how much you hear from τ seconds before current time t. From this example, it might become clear that $h(\tau)$ can serve as a filter in many signal processing applications. Say, for instance, that $h(\tau)$ decays slowly. The result would be that the sound $f(t)$ that you produce at time t is muffled by reverb. However, if we take $h(\tau) = \delta(\tau)$, the Dirac delta function, we hear only the sound produced at $\tau = 0$, which is only the sound $f(t)$.

3.4.2. Convolution in Neural Networks

We define a convolution in two dimensions of a function $f(x, y)$ and a kernel $h(x, y)$ as

$$(f * h)(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\tau_u, \tau_v) h(x - \tau_v, y - \tau_v) d\tau_u d\tau_v. \tag{3.19}$$

In machine learning applications, we have a finite multidimensional array as input, as well as a finite multidimensional array of values representing the kernel. Therefore, in applications we discretise equation 3.19 to

$$(f * h)(x, y) = \sum_m \sum_n f(m, n) h(x - m, y - n), \tag{3.20}$$

where m, n are the sizes of the multidimensional kernel array. We represent kernels as matrices instead of two dimensional functions.

Notice that by definition, h is only a function of the distances $d(x, u)$ and $d(y, v)$. This means that the kernel function h is *translation equivariant*. A function is equivariant when the output changes the same way as the input. More specifically, $f(x)$ is equivarant to h if $f(h(x)) = h(f(x))$. This property is extremely useful in image recognition, as we are able to detect shapes in an input image regardless of its location in the image. Thus, using convolution we are able to create a two dimensional map of where certain shapes appear in the input. If we then move the shape in the input, the representation of the shape will move an equal amount as in the input.

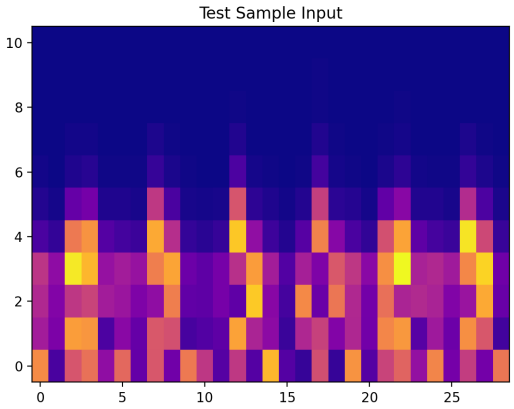

Figure 3.6: A three second spectrogram sample of a techno song, used in the test set. On the x-axis are the 29 timesteps, and on the y-axis the 11 frequency bands. Yellow pixels denote higher activity.

If we consider figure 3.6, in which a 3 second spectrogram sample of a techno song is visualised, we can see repetitive beats as vertical lines or activations. Suppose we have a 3×3 kernel K

$$K = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}. \tag{3.21}$$

If we now 'slide' K across figure 3.6, convolving them, the output map (figure 3.7) will have high activations where the beats occur, while simultaneously maintaining the distance between consecutive beats.

This means that if the beats were slower, and therefore would have more space between them, the output of the convolution would maintain this space. These features allow the network to learn non-linear relationships in the deeper layers. The convolutional neural network is therefore able to recognise differences in tempo. Analogously, using convolution across the frequency axis, we can detect identical melodies played at different frequencies or octaves. Due to the equivariance property of convolution, the output will have the same representation of the octaves, only shifted in height.

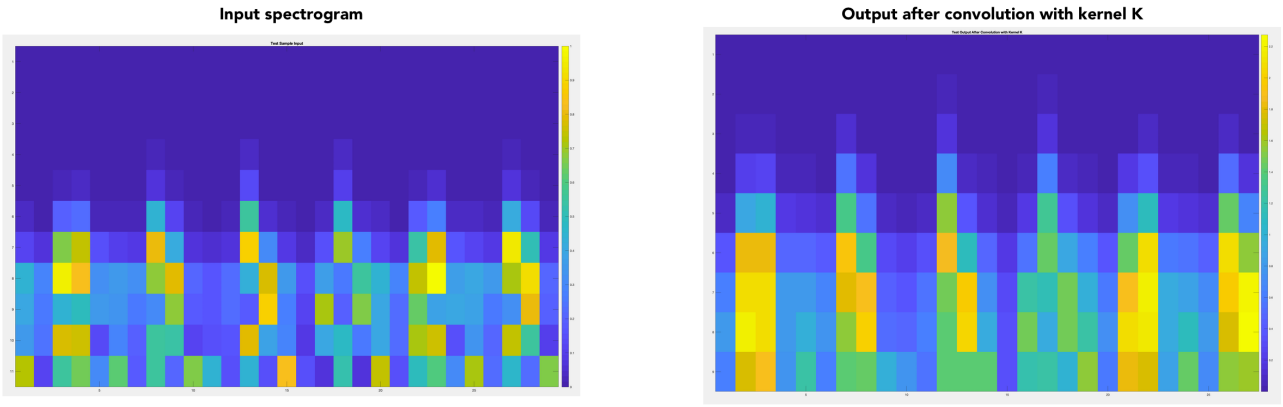

Figure 3.7: On the left we see the input spectrogram. This input is identical to figure 3.6, but has a different color mapping. On the right, we see the output after applying convolution with kernel K (equation 3.21).

In section 3.2 we have seen that the layers in multilayer perceptrons are dense. That is, the layers use matrix multiplication by a matrix W of weights where each weight w_{ij} describes the interaction between an input and an output perceptron. If we used this approach for a two dimensional input, we would need to have a four dimensional weight matrix W_{ijkl}, where i, j are the input coordinates, and k, l the mapped input coordinates. This means that mapping an input image of 10×10 pixels to an output image of 10×10 requires 10^4 parameters, since $i, j, k, l \in \{0, 1, \cdots, 9, 10\}$.
This approach is infeasible for larger images, and using the same kernel to map each input pixel x, y to some output pixel k, l drastically reduces the amount of parameters. If we have a kernel of 10×10, we only need to learn the parameters of the kernel, which are 10^2 parameters.

In addition to using only one kernel to map the entire image with, we typically have *sparse weights* (figure 3.8). We accomplish this by using small dimensions for the kernel matrices. This results in fewer trainable weights, but also reduces computation power, as we require fewer operations since m, n in equation 3.20 are smaller. Ideally, we want the size of the kernel to be as small as possible, but large enough to learn certain shapes. For instance, if we want some kernel K to recognise a complete face, we need its dimension to be larger than a 3×3 matrix. If, however, we require the kernel to recognise only a simple vertical line, we do not need much more than a 3×3 matrix.
Additionally, we use the same kernel matrix to convolve with every input value, essentially 'sliding' the kernel matrix over the multidimensional input array. Hence, rather than learning a separate set of weights for each location in the input, we learn one set, the kernel. This again drastically reduces the amount of trainable weights per convolution.

3.4.3. Multiple Kernels

When using only a single kernel in a convolution, we are unable to recognise multiple patterns in the input. For instance, using the kernel in 3.21 we are only able to detect vertical lines. Any other patterns in the input will not be noticed by the network. A solution to this problem is to define multiple kernels $K_1, K_2, \ldots, K_n$. Each of the kernels we then convolve with the entire input image to retrieve multiple different features from the input. All kernels get randomly initialised similar to the weights in a multilayer perceptron, and get trained using gradient descent.

3.4.4. Chaining Convolution Layers

In the previous sections, we discussed how a convolution layer works for a two dimensional input. Suppose we have a network with multiple convolution layers, in which the output of the first convolution layer is an input image convolved with k kernels, generating k output images of size $n \times m$. These k output images are then the input of the next convolution layer.

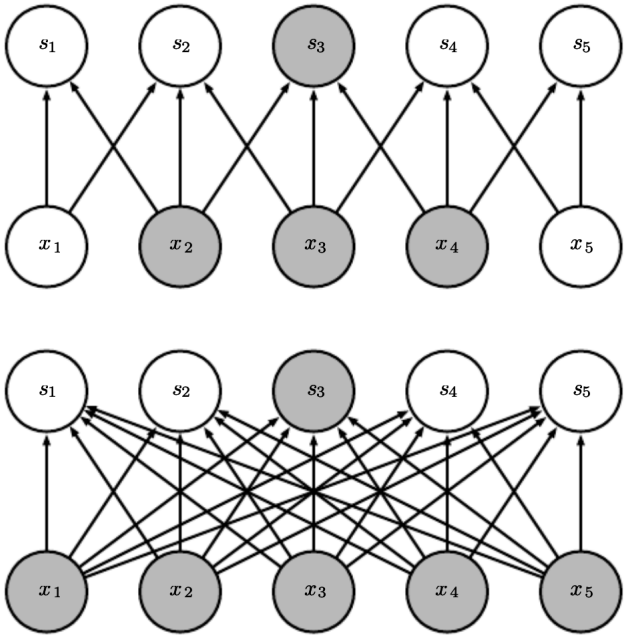

Figure 3.8: Two figures where we both highlight an output perceptron, s_3, and highlight the perceptrons in the input layer that s_3 is dependent on. The top figure is when $\vec{s}$ is formed by a convolution with a kernel of size 3 (three input perceptrons are connected to s_3). In the bottom figure we see a traditional dense layer, in which s_3 depends on all the perceptrons from the input layer. We can see many more relationships and a much more (computationally) complex network. Adopted from [14].

Instead of a two dimensional image as input, the second convolution layer now has k two dimensional input images. Let the second convolution layer have l kernels. We distinguish two fundamentally different implementations of a convolution layer:

1. The first implementation is intuitively more straightforward, but often not used in practise. In this implementation of convolution, we treat all of the k input images as separate images. This means that we convolve each of the k images with each of the l kernels in the second convolution layer. We then obtain $k \cdot l$ output images. Using this approach for all the consecutive convolution layers we obtain an exponential growth of output images. In a scenario where the convolutional neural network must learn complex features and requires many convolution layers, this approach can quickly lead to large sizes of output images. When we flatten all of the output images to a single one dimensional vector and pass them in a dense layer, this requires many parameters which can lead to extremely complex and infeasible neural networks. Therefore, in practise, we often use the second implementation.

2. A second, and more feasible implementation of convolution, is to stack the k output images from the first convolution layer together in a three dimensional matrix, with dimensions $n \times m \times k$. We must now convolve this three dimensional matrix with a three dimensional kernel where one dimension must be of size k. When we convolve the three dimensional kernel with the stacked k images from the first convolution layer, we essentially use all of the k feature maps to generate a new feature map. Hence, we prevent exponential growth of the amount of output images

Figure 3.9 provides a visualisation of a convolution layer of implementation 2 where $k = 3$. On the left we have 3 input images from the previous convolution layer, which must have had 3 kernels. In this convolution layer, we have 2 kernels each of size $4 \times 4 \times 3$, where the last dimension must equal 3 to perform valid convolution with the 3 input images. In this visualisation, the three dimensional kernels are called filters. Upon convolving the kernels with the three dimensional input, we obtain two output images of two dimensional size. We stack the two output images obtained from the convolution and pass the result through the non-linear $ReLU$ function. The output is then again a three dimensional

matrix. This means that every output image in the i-th convolution layer is a non-linear combination of every kernel in all of the previous convolution layers.

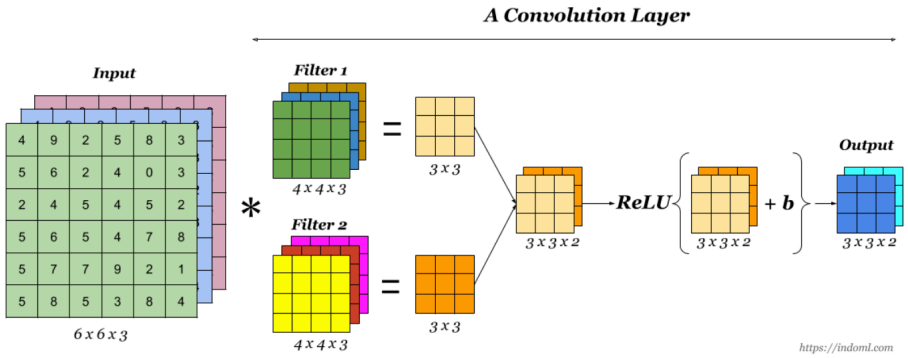

Figure 3.9: An example overview of a convolution layer described in scenario 2. The input on the left gets convolved with two separate three dimensional kernels, which are called filters in this instance. The result is then passed through a non-linear activation function, *ReLU* in this case. Adopted from [27].

3.4.5. Derivatives using Convolution

In the previous section we have seen a few examples of possible kernels, such as K in equation 3.21. In this section we examine the derivative filter. To this end, we use the following theorem

Theorem 1. *Assume that $f(\vec{x}), g(\vec{x})$ are integrable, then*

$$\frac{\partial}{\partial x_i}(f * g) = \frac{\partial}{\partial x_i} f * g = f * \frac{\partial}{\partial x_i} g \tag{3.22}$$

for all $x_i \in \vec{x}$.

Proof.

$$\begin{aligned}\frac{\partial}{\partial x_i}(f * g) &= \frac{\partial}{\partial x_i}\int_{-\infty}^{\infty} f(\tau) g(\vec{x} - \tau) d\tau \\ &= \int_{-\infty}^{\infty} f(\tau) \frac{\partial}{\partial x_i} g(\vec{x} - \tau) d\tau \\ &= f * \frac{\partial}{\partial x_i} g\end{aligned}$$

Analogously, by the commutativity property of convolution, it follows that $\frac{\partial}{\partial x_i}(f * g) = \frac{\partial}{\partial x_i} f * g$. □

From theorem 1 we see that the derivative of the convolution of an input with a kernel is equal to the convolution of an input with the derivative of a kernel. In practise, we can use this property to extract very useful features from the input data. Figure 3.10 illustrates the convolution of the top image of a tiger with a derivative kernel in both axes. The result shows where changes in pixel intensities are present. Using the derivative kernel we can find features through their edges. J. Zhu [34] shows more examples of derivative kernels.

One example of such a derivative kernel is the *Sobel derivative kernel*.

$$K_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}. \tag{3.23}$$

K_x denotes the Sobel derivative kernel in the x, or horizontal direction.

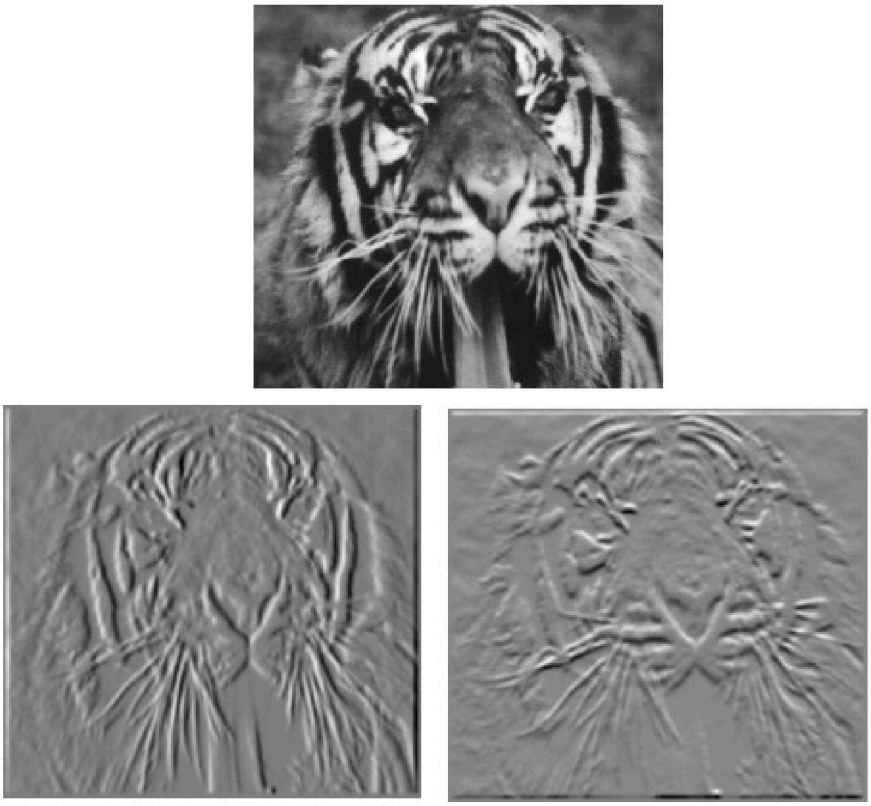

Figure 3.10: The top image representing the input image. The left bottom image is the result of the input image convolved with a derivative kernel in the x domain, and the right bottom image is the result of the input image convolved with a derivative kernel in the y domain. Adopted from [34].

3.4.6. Pooling

After a convolution layer we typically have a pooling layer. A pooling layer replaces the output of the preceding convolution layer in a certain location with a summary statistic of the nearby output values [14]. A particularly and widely used pooling method is *max pooling*, in which we take the maximum of a subset of the input and replace this subset with the maximum value in the subset. Figure 3.11 illustrates this procedure using a 4×4 matrix with a 2×2 max-pooling kernel. Other popular pooling functions include average pooling, taking the L^2 norm of a subset or a weighted average based on the distance of two points in the subset.

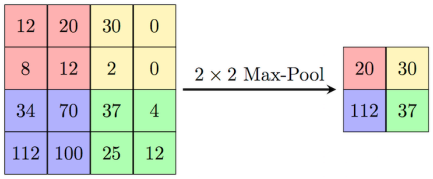

Figure 3.11: Max pooling on a 4×4 matrix with a 2×2 kernel. Adopted from [1].

Every pooling function contributes to making the input invariant to small translations. Thus, if we translate the input slightly, most of the max pooled outputs do not change value, since the maximum is still taken in a subset. This property is useful in scenario's where we care more about whether a shape is present than where the feature is located. In the context of spectrograms, translation invariance is not necessarily desirable, as the location of the feature in the input is likely more valuable than the knowledge of its presence. For instance, knowing that there is a specific drum element in the audio itself might not give us much information. Instead, knowing the location of the drum elements in the

spectrogram allows us to derive a possible tempo.

Another reason why we use pooling is to reduce the representation size. Consider the example in figure 3.11: Using a 2×2 pooling kernel we reduce the size of the representation by a factor of $2 \cdot 2 = 4$. This reduces the amount of trainable weights for the next layer, hence reducing the amount of complexity and statistical burden. Therefore, a pooling layer is helpful especially in cases where computation power is scarce.

3.4.7. Composition of a Convolutional Neural Network

In the previous sections we gained familiarity with the notion of convolutional layers and pooling layers. Using a combination of these two layers with the dense layers described in the multilayer perceptron (section 3.2), we are able to construct a full convolutional neural network which has the ability to classify shapes or objects in the input data. Typically, the first few layers of a convolutional network consist of an alternation between a convolution layer and a pooling layer, as illustrated in figure 3.12. In these layers, we attempt to extract features from the input using the convolution kernels, while simultaneously creating translation equivariance and small invariance to translation using pooling. In each layer the input reduces. This requires fewer parameters in the consecutive dense layers. After the convolution and pooling, we flatten the input into a one dimensional vector which is then fed into multiple dense layers. The non-linearity and strong interdependence in the dense layers allows the network to learn complex relationships between the extracted features in the convolutional layers.

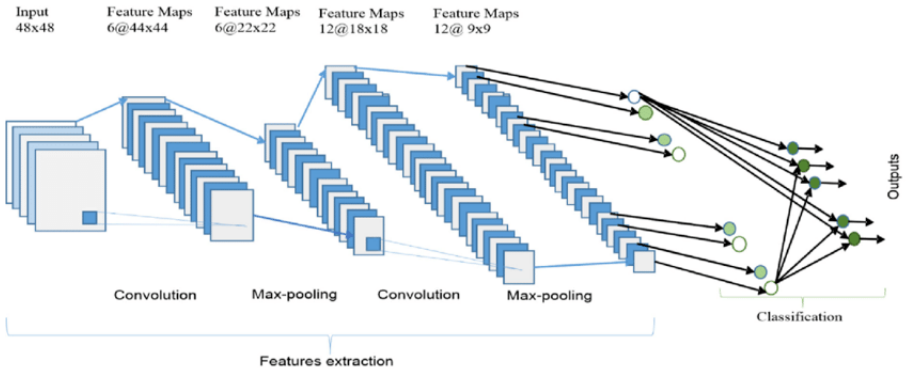

Figure 3.12: A convolutional neural network from left (input) to right (output). The first layers alternate between convolution layers and pooling layers. Consequently, the input is flattened and fed into dense layers, which are denoted as the 'classification' section. This part of the network attempts to construct relationships between the features through non-linearity. Adopted from [3].

4

Comparing Neural Networks

In this chapter we create multiple neural networks derived from building blocks that we discussed in chapter 3. The aim of this chapter is to compare the performance of these networks and gain insights as to why networks behave a certain way under specific circumstances. We experiment with various input data, layers, activation functions, optimisers and discuss more practical features during training.

4.1. Input and Output of the Neural Network

We concluded chapter 2 with a normalised dense matrix, which represents a spectrogram with on the x-axis the time values and on the y-axis the frequency bands. This matrix will be the input for all of the following networks.
Similarly, the output of every model will be a probability distribution in which for each genre class a probability is given. We let the networks output a probability distribution rather than a standard unit vector is because we want to see the uncertainty of the output prediction. Additionally, if the genre of a song is not present in our classes, a probability distribution allows the network to output a combination of available classes, approximating this genre. Finally, genres of various songs can be ambiguous, and a probability distribution enables us to quantify this ambiguity.
All of the networks will attempt to predict the following six genres:

- Jazz
- Classical
- Rock
- House
- Hip-Hop
- Techno

Since we want all networks to output a probability distribution, and we have the amount of classes over which it should output this distribution, we can construct the final layer which will be identical for every network.
The amount of classes we have is six. Hence, we need a final layer of six perceptrons, each outputting the probability that the input sample belongs to 'their' class. The activation function for the final layer should be the softmax function, as this function outputs a probability distribution (section 3.2.2).

4.1.1. Loss function

Aside from the final layer, we can make another architecture decision based on the final layer, namely the loss function. In section 3.1.2 we discussed the relevance of a loss function. We need the loss function to quantify the distance of the prediction from the actual value. We used the mean-squared-error (equation 3.3) as a loss function. Theoretically, we could use this function for our purposes. However,

decision boundaries in a classification task such as classifying genres are large, and the mean squared error is unable to penalise the network enough when a wrong class is predicted. Additionally, for classification tasks, networks using mean squared error as loss do not converge to a good optimum [13].

The alternative to the mean squared error loss function (equation 3.3) is *cross-entropy* loss. For discrete probabilities such as the genre classes we use

$$H(p,q) = -\sum_{x \in C} p(x) \log q(x) \tag{4.1}$$

where p is the actual discrete probability distribution, q the predicted probability distribution, and C the set of classes. During training, we let p be the standard basis vector for a C dimensional space. For instance, when the sample belongs to class i, $p = e_i$. When we want to classify songs that have a combination of multiple genre classes, we can adapt this discrete probability distribution vector.
We use cross-entropy for all of the following models.

4.1.2. Overfitting & Dataset Split

During the training of the various networks it is desirable to have a statistical measure of how the network is performing. We could base the performance solely on the training data. However, it is possible that the network performs very well on the training set, as all the parameters are fitted to the training set, but actually cannot generalise to unseen observations. The network is *overfitting* the training data. In order to avoid this problem, we create a validation set, which contains data that the network does not train on. After each training cycle, we adjust the weights using gradient descent. Consequently, we test the new weights on the validation set. We do not perform gradient descent on the validation set, as this set must remain independent from the training set. Based on the validation set, we can adapt other parameters such as the learning rate. These kinds of parameters are called *hyperparameters*. Hyperparameters are not derived through gradient descent, but can control the learning process. Finally, we have a testing set. We need a testing set alongside a validation set to test whether the hyperparameters are properly configured. The test set should be identical for each network we train. This way, we have a measure to compare the various networks against. We choose to divide the total amount of observations we have into a 70%-training set, a 15%-validation set and a 15%-testing set.

4.2. Hidden Layers

As we have seen in section 3.2, every layer which is not an input or output layer is defined as a hidden layer. When composing multiple layers to create a network, an elementary question comes to mind: *'How many layers with how many perceptrons do we need?'*. Unfortunately, no exact answer exists to this question [9, 20]. At this point in time, apart from several guidelines and trail-and-error we are unable to proof optimality of layers and neurons. Therefore, using guidelines such as the *universal approximation theorem*, stating that a network with a single hidden layer containing a finite number of perceptrons can approximate continuous functions on compact subsets of $\mathbb{R}^n$, and simply experimenting with different configurations is how we explore the architecture of the hidden layers.

Another motivation when creating the networks is to minimise computational cost. Minimising computational cost includes limiting the amount of parameters in the networks (recall that parameters are weight connections between perceptrons), but also limiting the amount of hidden layers, as back propagating errors gets more computationally expensive as more hidden layers are introduced.

4.3. Parameters during training

During the process of training various networks, we apply two measures to increase the efficiency of training.

1. We train every network for an undefined amount of training cycles. The system keeps track of various statistics [16, 17], including the accuracy on the validation set (section 4.1.2). When the system detects that the validation accuracy has not increased by a certain value over a period of n training cycles, it terminates the training process. We use the same value of n to train every

network, so that we can compare them as equally as possible.
Using this method of training, we can see that the system sometimes trains networks for many more training cycles than other networks.

2. Based on another statistic that the system keeps track of, the training loss, we decrease the learning rate by 80% when after k training cycles the training loss has not decreased by some value. By decreasing the learning rate when the network does not descent anymore into a minimum, we try to ensure that the network has found the minimum by taking consecutively smaller gradient steps.

For all the networks, we decide to terminate training if after 50 training cycles the validation accuracy has not increased by 1%.
Similarly, we reduce the learning rate for each network when after 5 training cycles the training loss has not decreased by 0.0001. These values have proven to work well in practise.

4.4. Standard Neural Network

The first network we consider is the most vanilla type of network as described in section 3.2. Essentially, we use only dense layers (section 3.2.1). In the first layer, we flatten the two dimensional input spectrogram into a one dimensional vector, for instance, by their row order. We then pass every element of the vector to a perceptron in the input layer. We create two hidden layers of 256 and 128 perceptrons respectively. For each of the dense layers we use the $ReLU$ activation function (section 3.2.2). This activation function is computationally efficient and research suggests that $ReLU$ leads to faster convergence [10]. The final layer is the softmax layer with 6 perceptrons, as discussed in section 4.1.

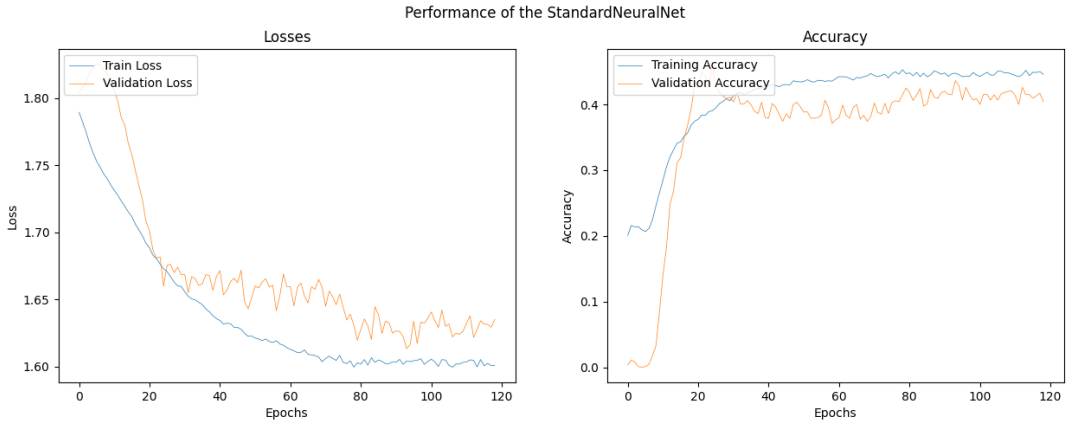

Figure 4.1: The loss and accuracy curve over consecutive epochs of training. The blue lines denote the training loss and accuracy, and the orange lines denote the validation loss and accuracy.

Figure 4.1 illustrates the loss and accuracy with respect to consecutive training cycles. One training cycle in which all of the training data is passed through the model is denoted as an *epoch*. After 120 epochs, the validation accuracy has not increased enough to continue training, and the process terminates. We see that the loss function hits a plateau, which indicates that a minimum is reached.

The standard neural network is able to get an accuracy of 42%. Notice that we have 6 classes, and a balanced dataset, meaning approximately equal amounts of training data for each class. Therefore, randomly picking genres would result in an accuracy of roughly $\frac{100}{6} = 16.7\%$. This means that even a neural network with only dense layers is able to extract features from the spectrogram, and predict half of the genres correctly. In theory, a network that only consists of dense layers could approximate any function due to its non-linearity. We can see in this example that the network is learning certain relationships, but seems to get stuck in a local minimum. The network could require more complexity in order to find more relationships. In the following sections, however, we will see more efficient ways to extract features for the dense non-linear layers, as they require fewer parameters.

In the following models, we build on top of this model, as the few dense layers in the standard network have shown to be able to learn relationships in the data. The aim of recurrence and convolution is to extract the features from which the dense layers can learn useful relationships. In the following sections, we attempt to find more features by examining recurrent relationships in the data using recurrent neural networks, and by treating the spectrogram as an image, using convolution to detect features. Consequently, we use a similar architecture as the standard neural network. We need the dense layers in the end of the network to apply non-linearity to the obtained features.

4.5. Recurrent Neural Network

In section 3.3 we got familiar with the network architecture of recurrent neural networks. Its recurrent behaviour allows us to detect rhythmic elements in the song. Heuristically, these rhythmic elements could be able to detect the energy peaks when drums occur in house, hip-hop and techno samples.

The first recurrent model we build is a model with one recurrent layer as described in section 3.3.1. This is the simplest form of a recurrent layer. Other recurrent layers are the *gated recurrent layer* and the *long short-term memory layer*, often referred to as *GRU layer* and *LSTM layer* respectively. We also test the gated recurrent network, but there are no noticeable performance gains over the simple recurrent layer. Since the simple recurrent layer has fewer computations, we use this layer in the networks. The LSTM layer requires significantly more computations than both the GRU as well as the simple recurrent layer and is therefore undesirable when limiting computational cost.
In the first configuration we use the $tanh$ as activation function for the recurrent layer with 128 perceptrons. The motivation behind the $tanh$ function in the recurrent layer is that the output of the $tanh$ can be both positive as well as negative, allowing for increases and decreases in the state. The ability for the network to decrease the state allows it to describe states that are inversely proportional to previous states. We are not specifically interested in inverse proportional relationships between time steps. Instead, we would like to have relationships between high energy states, as these are likely to tell us more about the tempo and the type of song.
Followed by the recurrent layer, we use an exact copy of the standard neural network, where we have two dense layers of 256 and 128 perceptrons respectively. Using the identical network allows us to compare performance of the two networks, and determine if the recurrence layer is capable of extracting any useful features, hence increasing performance.

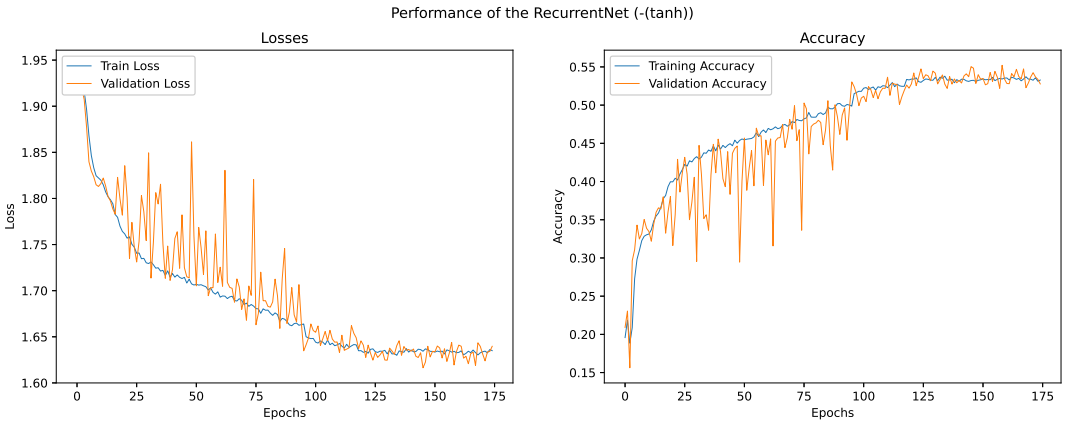

Figure 4.2: The loss and accuracy during training of the recurrent neural network with the $tanh$ activation function.

In figure 4.2 we see the result of training the recurrent network using the $tanh$ activation layer. The peaks in the validation loss and training are likely caused by the loss function being slightly different for the validation set than for the training set. This could be a sign of a learning rate that is too high, but in this case the validation loss and accuracy converge after 100 epochs.

We see that the network is reaching a plateau around 50% accuracy. This suggests that the loss function has reached a minimum. Notice, however, that we have almost a 10% increase in performance using a single recurrent layer. This indicates that the recurrent layer is in fact able to recognise features

from which the dense layers can learn more effectively.
Another issue that arises in this network could be that the gradient converges during training as a result of the vanishing gradient problem (section 3.2.2). A way to mitigate this issue is by using another activation function. The $ReLU(x) = \max 0, x$ as activation function does not output negative values. From equation 3.14 we can see that using $ReLU$ as the activation function for the recurrent layer does not allow negative states, but is immune to vanishing or exploding gradients due to its derivative. Negative states could be used to imply inverse relationships between previous states.
Finally, since the network shows slight increased performance, a possible solution to gain a better performance could be to add more complexity to the recurrent layers. We test this hypothesis by adding another recurrent layer with 256 units. The stacking of recurrent layers allows the recurrent layers to create a more complex attribute representation of the current input.

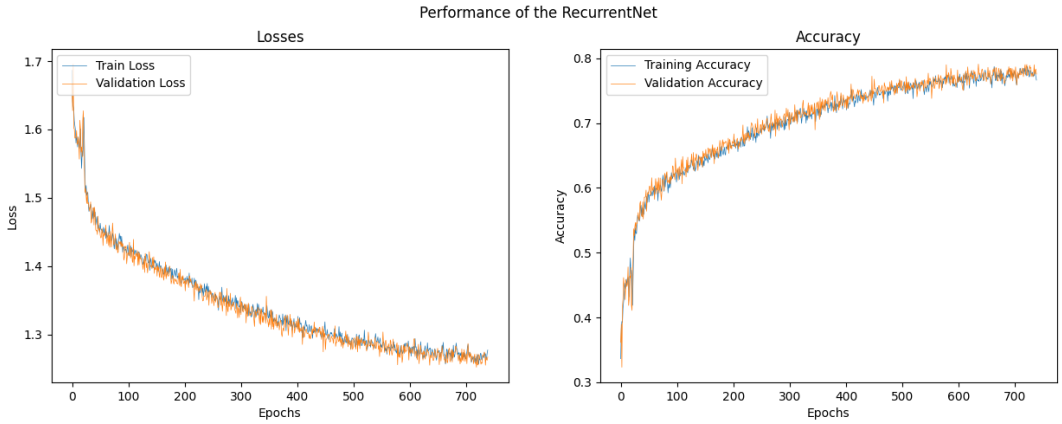

Figure 4.3: The loss and accuracy during training of the recurrent neural network with two recurrent layers and a $ReLU$ activation function.

We can see that the network went through many more training cycles than the previous network. This is a result of the measures taken in section 4.3. We notice a significant increase in performance, with the accuracy reaching nearly 80% in its final training cycles.

This result is very informative. We can conclude that more complexity in the recurrent layers leads to more useful features that are extracted. Additionally, it indicates that the standard neural network, consisting of two dense layers, which serves also as the final part of the recurrent network, is enough to learn about many relationships in the sound samples.

Finally, we let the model predict 100 test samples, and visualise the predictions against the actual genre (figure 4.4). Figure 4.4 also shows a genre called *ballad*, which we discuss in section 4.9. The recurrent model performs surprisingly well on jazz and classical sound samples, even though fewer rhythmic elements can generally be found in these genres than in the others. Overall, we are unable to identify a clear cause of misclassification.

4.6. Convolutional Neural Network

The final type of network we construct is a convolutional neural network. Convolutional neural networks have proven to be very useful in recognising patterns in images and videos. In the recurrent network the approach was to detect rhythmic elements in the sound samples. In the convolutional network we are treating the spectrogram of the sound samples essentially as grayscale images, where higher energy in the sound is represented by brighter pixel values.

The first three layers of the network consist of alternating convolution and max pooling layers. The motivation of this architecture is described in section 3.4.7. We choose a 3×3 kernel in the convolution layers, and a 2×2 max pooling matrix, minimising translation invariance and loss of information. After this process, the convolved and max pooled matrix is flattened and passed into a simplified version of the standard neural net, with only one dense layer of 128 perceptrons. For the convolution layers

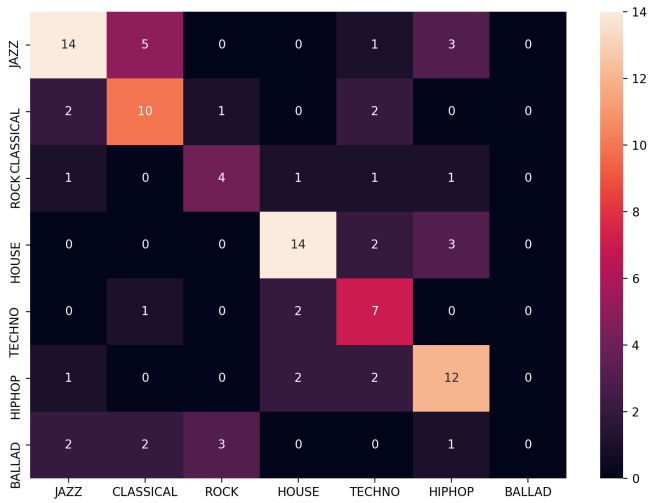

Figure 4.4: A classification of 100 test samples from the recurrent network. On the y-axis are the samples known to be this class, and on the x-axis are the predicted classes by the network.

we use 32 kernels for both layers. This amount is chosen arbitrary at first, but we will later see what results changing the amount of kernels in the convolution layers gives. Figure 4.5 gives a schematic representation of the discussed architecture.

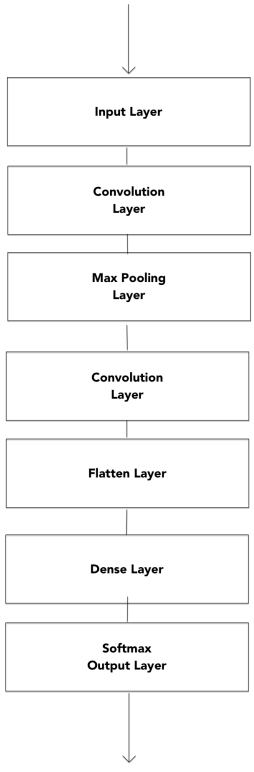

Figure 4.5: A schematic representation of the convolutional neural network architecture.

In figure 4.6, we see that within 100 training cycles, the accuracy of the convolutional network is already higher than the accuracy of the recurrent network.

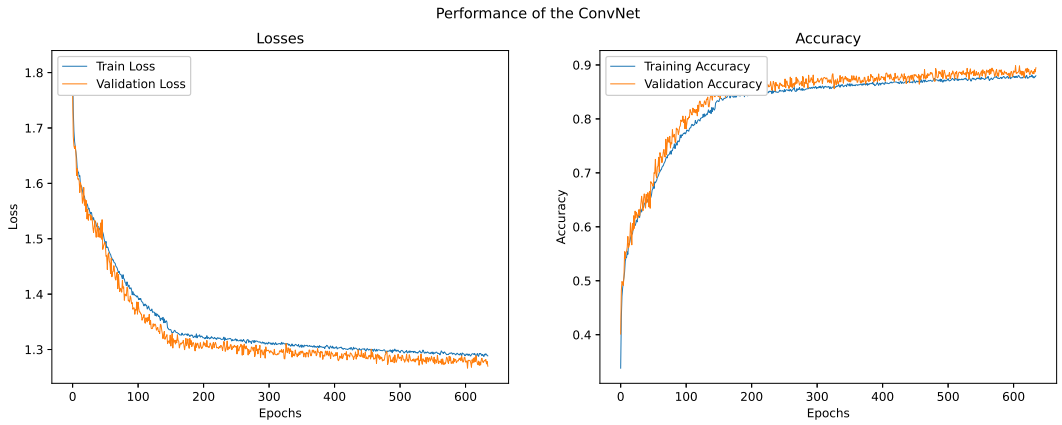

Figure 4.6: The loss and accuracy during training of the convolutional neural network with two convolution layers, max pooling and one dense layer.

Also, when visualising the confusion matrix of the convolutional net using the same 100 test samples as the recurrent network, we see significantly more accurate classifications (figure 4.7). Just like in figure 4.4, figure 4.7 also shows a genre called *ballad*, which we discuss in section 4.9.

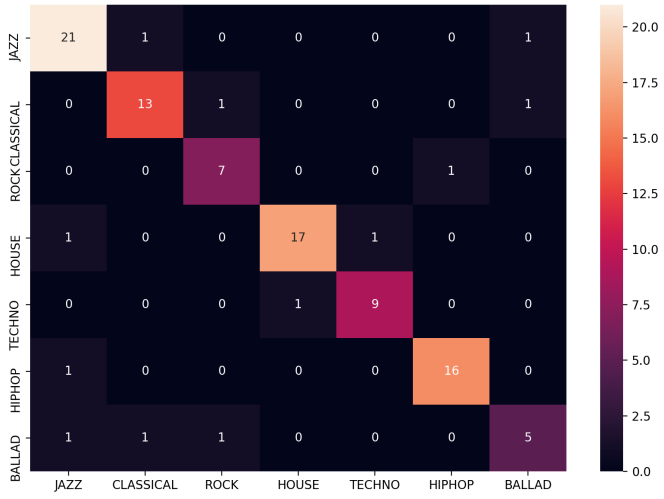

Figure 4.7: A classification of 100 test samples from the convolutional neural network. On the y-axis are the samples known to be this class, and on the x-axis are the predicted classes by the network.

4.6.1. Kernel Visualisation

In an attempt to gain more insight into the kernels, and which convolutions they output, we look at a specific test sample, visualised in figure 4.8. This test sample is a sound sample of a jazz song. We do not see any clear rhythmic elements in this sound sample, unlike the techno sound sample we saw earlier (figure 3.6).

After the first convolution layer we apply 32 kernels on this image, and record the output of every image convolved with each kernel in figure 4.9.

In this visualisation we can see 32 output images. The green, white, black and blue boxes around

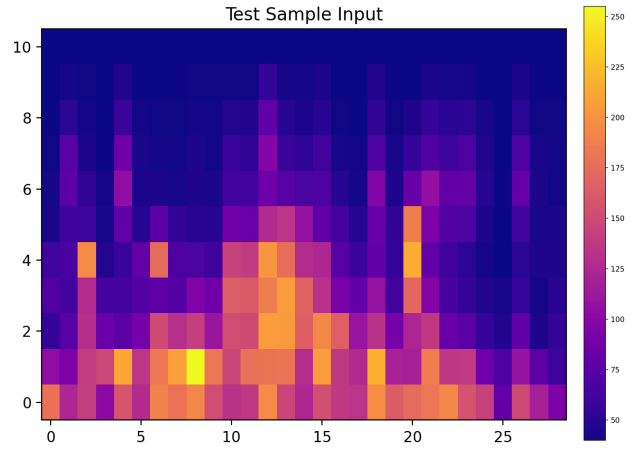

Figure 4.8: An input spectrogram of a jazz sound sample. Yellow values indicate higher energy in its corresponding frequency band. On the y-axis are the frequency bands, and on the x-axis are the different time steps.

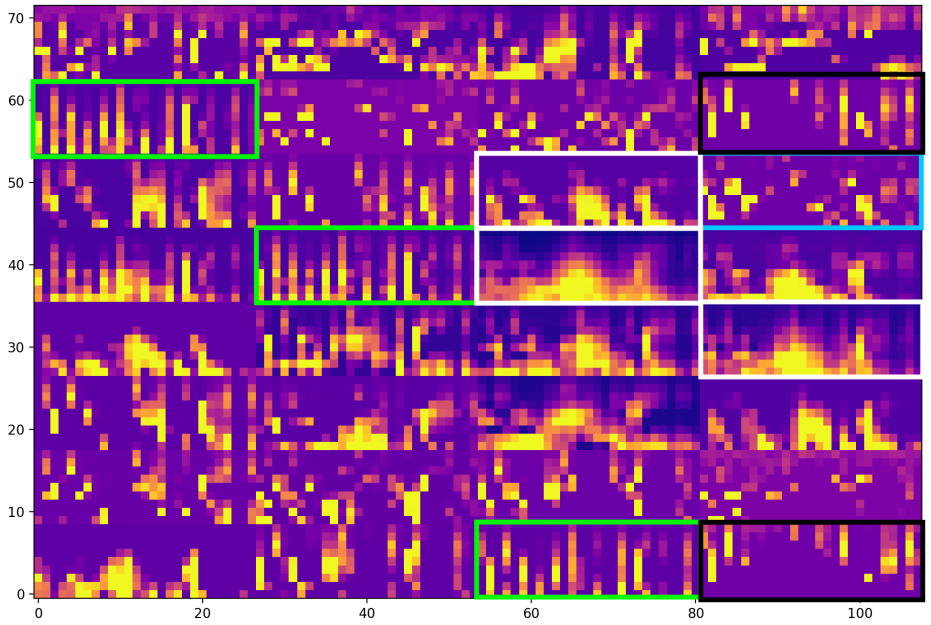

Figure 4.9: 32 different outputs of the first convolution layer. The green, white, black and blue boxes around certain output images imply certain behaviour which the kernels are trained to detect.

certain output images imply certain behaviour which the kernels are trained to detect. The green boxes have high activation in rhythmic patterns, and hence are likely looking for drum elements. The blue element seems to be a gradient, as discussed in 3.4.5. Finally, the white boxes seem to attenuate the sections in the song with high activity. This behaviour seems similar to sharpening the picture, by emphasis of gradients.

Possibly the most interesting output images are the images surrounded by the black box. The kernels that produced this input give us a clear intuition as to how the network is reasoning about the various input images. To understand this, carefully compare the images in the black boxes with the input image (figure 4.8). Notice that the energy is inverted. That is, the top left and right corners have barely any

energy in the input image (since they are dark blue), whereas the output images of the convolution layer surrounded by the black boxes do have high energy in these locations. This kernel is responsible for giving the information that certain elements are not present in that location. Using this information, the dense layers can create logical decisions using the information of absent energy together with other convolved output images. The ability of convolution to generate information on which elementary logical operations can be performed, combined with the non-linearity in the consecutive dense layers, results in the ability to classify genres with a high precision.

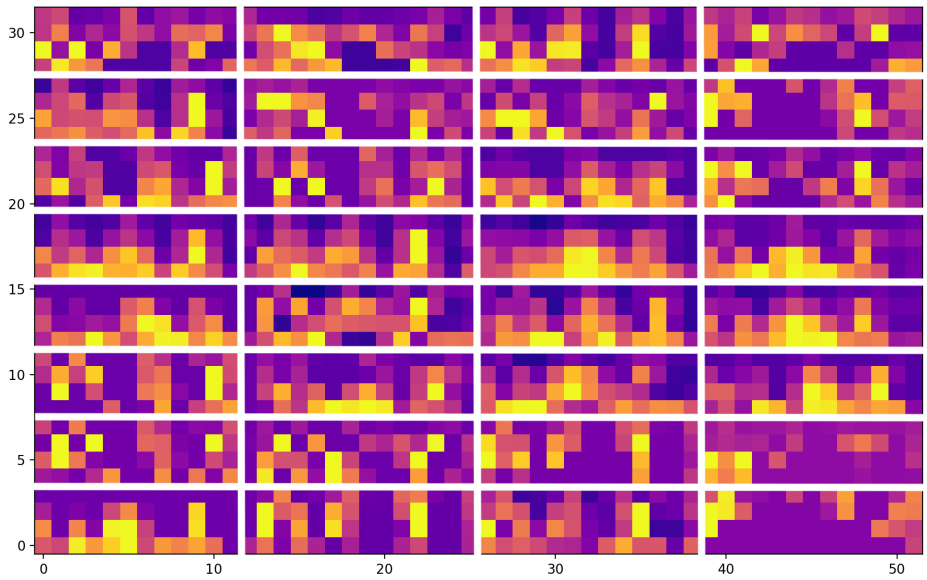

Figure 4.10: 32 different outputs of the max pooling layer after the first convolution layer.

After the first convolution layer, all of the images are passed into the max pooling layer, in which we shift a 2×2 matrix over the images and take the maximum value. The result of the max pooled layer is visualised in figure 4.10. We can still see certain kernel attributes, such as the bottom right kernel responsible for inverting the energy. However, visually it becomes quite difficult to see relationships in the output images.

In section 3.4.4 we discussed two implementations of the convolution layers. In this neural network, we use implementation 2, in which the convolution layers have three dimensional kernels, and hence each output depends on every input image in the convolution layer. When we look at the output images of the second convolution layer that comes after the max pooling layer (figure 4.11), we see that certain kernels remove all the activity. This could mean that the kernel is a zero matrix, and therefore not contributing to the network. Another more likely possibility could be that the kernels are specifically looking for certain elements in the input. Also in this scenario, the absence of energy in these output images might be likely to also give information about the input image.

Visualising the process of the convolution of input images gives us an intuition of how the non-linear dense layers might be functioning.

4.6.2. Varying the Amount of Kernels

In the previous section, we gained insight in the convolution and max pooling layers by visualising them. There is, however, something else we can see in the output images of the various layers. Namely, there is quite a few kernels that output almost identical images. For instance, the output images surrounded

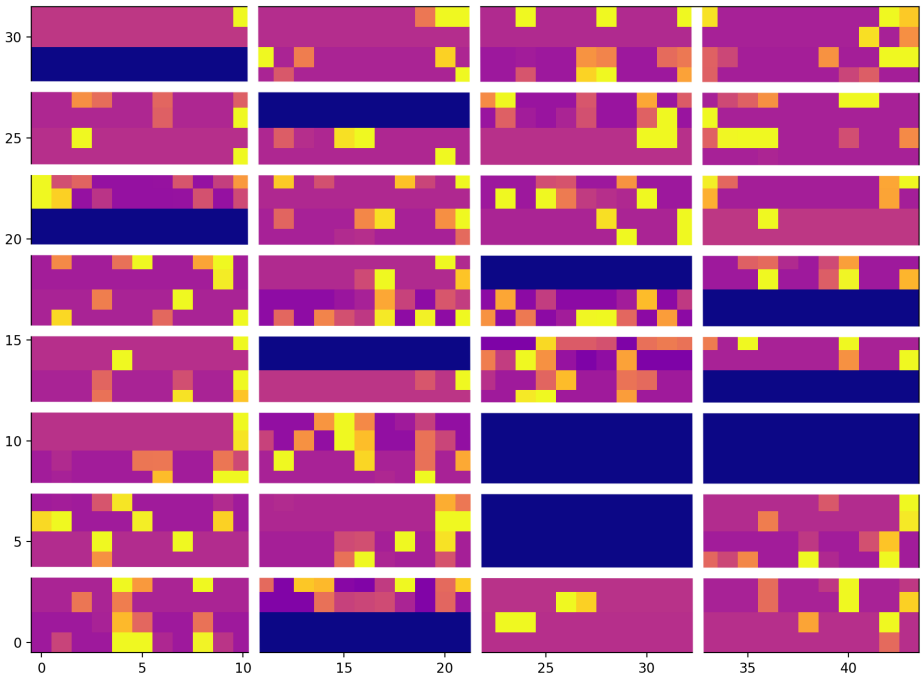

Figure 4.11: 32 different outputs of the second convolution layer, which comes after the max pooling layer.

by the green boxes in figure 4.9 look very similar. This leads us to question the necessity of the amount of kernels, which was arbitrarily chosen. In an attempt to answer this question, we created identical convolutional networks with different amounts of kernels in the convolution layers, and trained them for 100 epochs. The result of their performance is illustrated in figure 4.12.

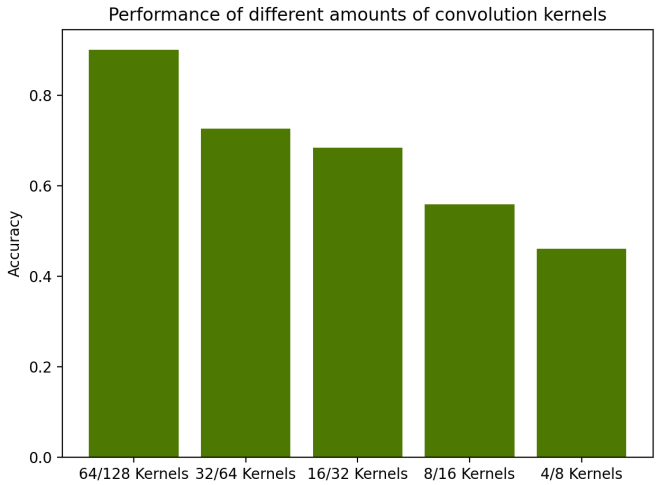

Figure 4.12: The performance of various convolutional neural networks with different amount of kernels in their convolution layers.

Notice that after 100 epochs, the network with many kernels has already reached an accuracy of over 80%. As we reduce the amount of kernels in the convolution layer, the performance decreases. Figure 4.12 indicate that the network converges significantly faster with more kernels. A possible explanation for this fact could be that useful kernels are faster found, since the kernels get initialised randomly. Larger amounts of randomly initialised kernels increase the probability that some kernels

are already close to an optimum.

It is likely that only a few kernels are used to extract information from the input images. When we continue training on all of the convolutional neural networks, eventually the 3 networks with the highest amount of kernels all reach a similar accuracy and loss of about 87% and 1.3 respectively. This result implies that, for this specific problem, one requires at least a combination of 16 and 32 kernels in the two convolution layers.

Using this knowledge, we can construct appropriate convolutional networks based on our requirements. If we want to train a network in the shortest amount of time, or with as little computational resources as possible, it is advisable to train many kernels, as the probability of finding useful kernels is increased. On the other hand, if we want to deploy our model after training and ensure it minimises computational costs, we want to have only useful kernels which contribute to the neural network.
Ultimately, we can create an architecture in which we train with many kernels, reaching higher performance faster. Consequently, we remove kernels and test if the removal of kernels impacts the performance of the model. If not, we remove another one. If it does, we put the kernel back in the model and we can conclude that this kernel is an essential component of the convolution layer.

4.7. Combination of Recurrence and Convolution

In the last sections we have constructed a recurrent network and a convolutional network, which both consisted of the standard neural network we defined in section 4.4. The convolutional neural network was able to get slightly higher accuracy, and make accurate predictions on the 100 test samples.
In order to test whether the recurrent and convolutional network extract different features from the data, we create a network which consists of both convolution layers as well as recurrence layers. We then combine the output of the two and pass this in to non-linear dense layers. If the recurrence and the convolution are able to recognise different useful features in the input data, a combination of both techniques might result in better performance than a single network.

In figure 4.13 a diagram of the architecture of the combined network is given. We seperately evaluate the two networks, and then merge the output of the convolution and recurrence together into dense layers that can apply non-linearity to the output of both the convolution and the recurrence.

Figure 4.14 shows the loss and accuracy information during training. If we compare this to our previous best performing model, the convolutional neural network (figure 4.6), we see that the combined network reaches an accuracy of 85% within 100 epochs. The convolutional network, on the other hand, needs over 250 epochs to reach the same loss and accuracy as the combined network. It seems like the combined neural net is the better performer, eventually reaching an accuracy of 89% on the training set, and 90% on the test set, which is 3% higher than the convolutional neural net. We must, however, keep in mind that the combined network requires the training of roughly 250000 parameters, which is $2.5\times$ more than the 100000 parameters of the convolutional neural network.
When choosing between the convolutional neural network and the combined network, we must again make the trade-off between training and evaluation speed versus the extra accuracy and more robust predictions.

Since the combined neural network was not able to achieve significantly higher accuracy than the convolutional network, it is plausible that the convolution filters extract similar information from the input data as the recurrent layers. Therefore, combining the techniques does not result in significantly higher performance. Depending on the use case, a model with 3% more accuracy and $2.5\times$ as many parameters can be desirable. In our case, however, we choose to go with the convolutional neural network instead of a combination of recurrence and convolution, as the performance gain does not weigh up to the additional computational cost.

4.8. Adjusting the Sample Lengths

When we transformed the songs into spectrograms, we decided to take rather arbitrary periods of 3 seconds from the songs. Now that we have constructed the architecture for various networks, we can

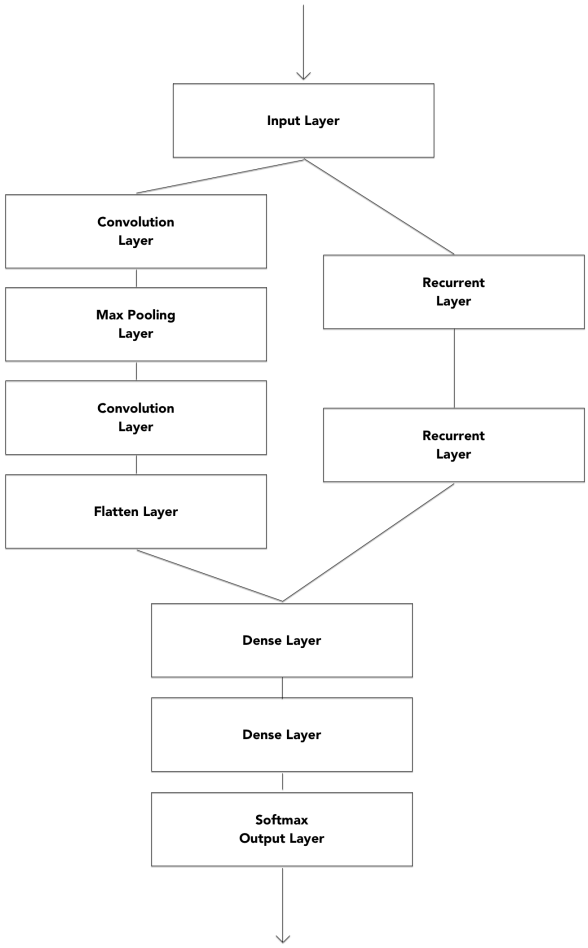

Figure 4.13: A diagram for the architecture of the combined network.

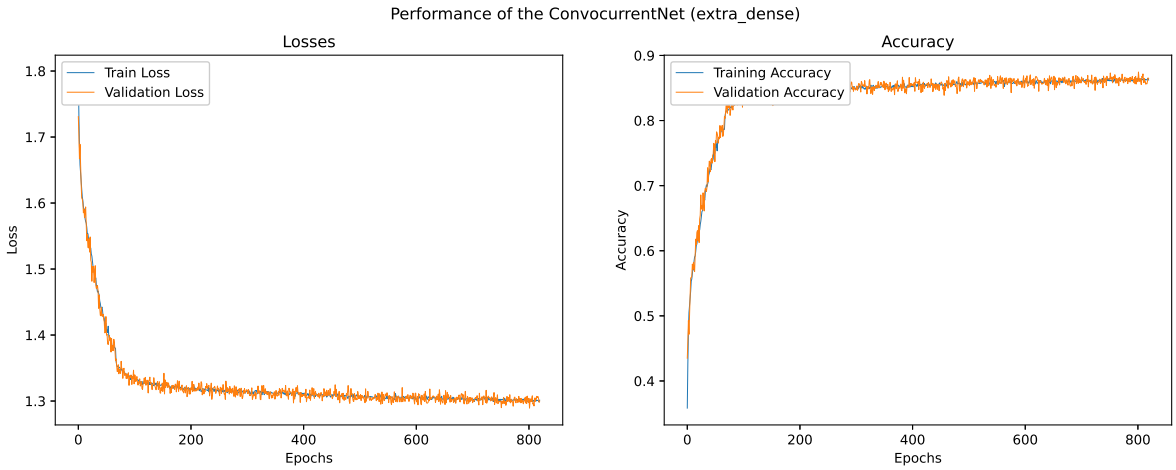

Figure 4.14: The training and validation loss and accuracy of the combined convolution and recurrent network.

generate datasets with different sound sample lengths. We train the networks on these various sample lengths to gain information regarding the impact of the sample length on the performance of various models.

Figure 4.15 shows the accuracy of the four models with respect to different sample lengths. The combined convolution and recurrent net, named the *ConvocurrentNet*, is only trained on 3 second

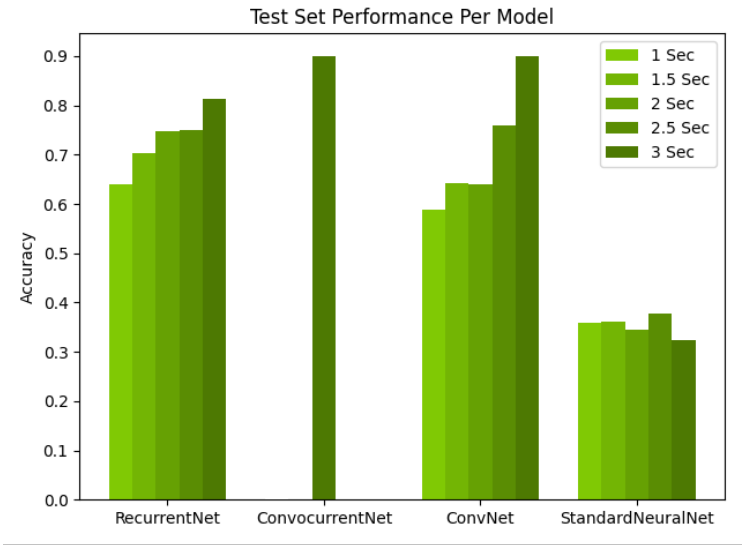

Figure 4.15: The accuracy of each model given the length of the input sound sample. The combined network is only trained on 3 second sound samples.

sound samples, as training this network multiple times required many computational resources.
From the accuracy increase of the recurrent neural network compared to the convolutional neural network, we can see that the recurrent network is more invariant to smaller sample sizes than the convolutional neural network. When taking 1 second sample sizes, the accuracy of the recurrent neural network is even higher than the convolutional neural network. This implies that the recurrent network obtains its strength from detecting an element that repeats in a second. It is likely that this element only occurs twice, given the short sample length.

4.9. Adding the Ballad Genre

Finally, we test the notion that the networks are able to detect the tempo of the sound samples by adding an additional genre. We decide to add the ballad genre, which is a popular genre that emerged mainly from the 1950s era. Ballad songs are very similar to rock songs, in that they use similar chord progressions and instruments. Their main difference is the tempo, which tends to be slower in ballad songs. We test the convolutional neural network on this extended range of classes to determine its ability to distinguish rock and ballad genres from each other.
We find, however, that the convolutional neural network does not change anything in performance. In fact, the training curve is identical to what we have seen in figure 4.6. Therefore, we can conclude that the convolutional neural network is capable of separating samples based on their tempo. Since this feature is mainly the strength of the recurrent network, we see why a combination of recurrence and convolution did not lead to significant results, as both networks have similar strengths.

5

Real Life Application: End-To-End Genre Prediction System

To give an idea of a possible real life application, we present in this chapter an end-to-end application which predicts genres from audio files. We look at each component of the system and determine their functions. The system is written in Python with the TensorFlow library [15], developed by Google. The source code is available here[1].

5.1. Project Structure

The application consists of seven different sections. Each section is encapsulated and should function on itself. The project structure discussed proved to be very useful to keep everything modular and organised. We recommend using a similar structure for small machine learning projects. We briefly discuss each section.

5.1.1. App

The first section is the *App*. This is the entry point of the application. We tell the application what we want to do (training, testing, downloading, processing) and it takes care of the full process based on our input parameters. The App can be seen as the general manager of the application, and holds a reference to all the other sections. Communication between sections is always done through the App.

5.1.2. I/O Handler (File System Manager)

This section takes care of everything related to storing and reading files. It stores locations of all the files and folders that we need during the entire process. When running the application for the first time, this section creates a directory to store the songs in and a directory to save trained models in. It is also responsible for reading $mp3$ files and converting them to wav.

5.1.3. Data Downloader

The data downloading section is responsible for downloading all the required songs and put them in the appropriate genre folder, with help of the file system manager. Here we also have a text file which contains id's of all the songs we have already downloaded, so that we do not download duplicate songs.

5.1.4. Pre-Processor

The third section is the *pre-processor*. In this section, everything related to what we discussed in chapter 2 is performed. The pre-processor selects multiple audio fragments from a song, creates spectrograms and reduces the spectrograms to a dense normalised matrix.

[1] https://github.com/laurensvm/getec

5.1.5. Dataset
The *dataset* section is responsible for serialising the genre and the corresponding normalised matrices created in the pre-processor into bytes, and saving them in a database file. Upon retrieval of the file, before training or testing, it deserialises the objects and splits them into training data, validation data and testing data. Consequently, all of the data is shuffled and ready to use for the neural networks.

5.1.6. Networks
The *networks* section contains the configuration and implementation for all the networks that we discussed in chapter 4. The networks can load themselves from the saved models directory, or train a new network.

5.1.7. Visualiser
The final section is what we call the *visualiser*. This section is responsible for the visualisations that are produced from the results. During training, the network keeps track of various statistics. After training, these statistics are passed into the visualiser, which creates plots from them.

5.2. Classifying a Single Example
Suppose all of the models we will use are trained, and we want to classify a single song. If we have to download the song from a cloud provider, we pass the URL to the App section, which then passes it to the downloader to download the data. Upon completing this process, the data is passed to the file system manager, which stores the data. It then converts it to wav format and reads it. We now have the audio data of the entire song, and we can process it.
The audio data is given to the pre-processor, which extracts 3 second sound samples from the song, and processes them into dense spectrograms. Then, without saving this to a database, we load the appropriate model from the saved models, and pass the extracted spectrograms to this model. It classifies all of the spectrograms and returns a probability distribution over the genres for all of them. Finally, we take the maximum of the probabilities and return this genre.

6

Conclusion & Discussion

6.1. Audio Processing; Representation of Audio Samples

In this thesis we focused primarily on developing a system which is able to classify the genre of a song based on its audio data. For the sake of computational feasibility, we decided to take small sections of the song and process only these sections.
When processing these audio sections, we split the section into even smaller chunks of 100ms, and overlapped them with each other to create a cover for the section. For each of these overlapping elements, we calculated their Fourier transform and merged the elements together to obtain a spectrogram. Again, for computational efficiency, we reduced the amount of frequencies to 11 frequency bands by taking the maximum values in each frequency band. We then store all of these dense matrices to be used as input to the neural networks

6.2. Networks

In chapter 3 we examined neural networks starting from its basic building block: the perceptron. We learned how the system uses a technique called gradient descent to update weights in order to reduce the error in the system and hence approximate a function producing desired output values.
Consequently, we learned to employ functions in the networks which have the ability to detect sequential structures in temporal data, by a technique referred to as recurrence. We used this technique to compose recurrent neural networks.
Finally, we got familiar with convolutions. We then applied this notion to neural networks, in which we detect shapes and patterns in the input data, which is usually set up as image data. It turns out that the alternation between convolution and pooling can be very beneficial in terms of translation invariance. This could be a useful property in image recognition.

6.3. Comparisons

After having built a solid foundation for the neural networks and the input data, we applied this knowledge to implement the networks and train them on the data.
The first network we created is what we referred to as the standard neural network, in which we put several dense layers together. We found that the $ReLU$ activation function worked appropriately, and the network was capable of reaching an accuracy significantly better than randomly picking from the genre classes.
We subsequently used the first network as the latter part of the following networks. The aim of this procedure is to use the standard dense layers as the non-linear layers designed to learn relationships between the features. The former layers we added had the task to extract useful features from the data.
To this end, we used recurrent layers to extract mainly periodic behaviour, such as drums and other rhythmic elements. We found that using two consecutive recurrence layers both with the $ReLU$ activation function resulted in a relatively well performing network which gained 78% accuracy on the test set. As we see in figure 4.4, various classes are still falsely predicted.

We then moved on to use convolution for the first part of the network. We found that the convolution layers are able to extract more useful features from the input data. We created a convolutional neural network which achieved an accuracy of 89% on the test data. We looked into the images that the consecutive convolutions produce, and noticed duplicate kernels in the layers. We then suggested that these kernels might be omitted without significant performance loss.
Finally, we created a combination of the recurrence features and the convolution features, and used both features as input to the dense non-linear layer. Our aim for this procedure was to create relationships based on a wider set of features, possibly independent from each other. It turns out that the convolutional layers can also extract temporal features, such as the recurrent network. Therefore, we found a minor increase in performance to 90% on the test set, while using 250.000 parameters, roughly $2.5\times$ as many as the convolutional neural network.

6.4. Further Research

During the research, the following questions and ideas emerged:

1. In this report, we looked at the input and output of the images as we passed them through the convolutional layers. Based on the mutations, we derived the structure of the kernel. However, in order to be able to compare these kernels across various convolutional networks with different amounts of kernels and layers, we should look at the kernel matrix itself. We can then determine which kernels are significant in the networks, by finding identical or similar kernels across the networks. To this end, we can create an algorithm which removes insignificant kernels by their similarity.

2. Extending on the first point, we concluded section 4.6.2 by suggesting that an appropriate training procedure of a convolutional neural network would be to train many kernels, and then remove insignificant ones. We have yet to test this idea, and determine whether this is useful in practise.

3. In view of the limited duration of this project, we have not been able to experiment with many different representations of the songs. For instance, figure 4.15 indicates that performance improves for both the recurrent as well as the convolutional network for longer sound samples, but we have not experimented with sound samples longer than 3 seconds. Hence, we have not found the optimum length of the sound samples. For performance reasons, knowing the optimum of these samples can be very beneficial, as we can get the best performance for the least amount of storage.

4. In addition to the previous point, other forms of input representation, other than spectrograms, should be considered. Examples of representations could be the envelope of the sound waves, or the sound waves themselves in the time domain instead of the frequency domain. It could be that the convolutional neural network is able to extract more features from these representations. Also, perhaps larger data representations, instead of the small dense spectrogram matrix, can increase the performance of the recurrent and convolutional neural network.

5. More extensive performance measures could be considered, other than the confusion matrix, accuracy and loss which we used as measures in this report. For instance, the $F1$-score could be a useful measure to add in order to classify the performance of the model. The $F1$-score measures both precision (how many instances of a class are correctly predicted given the total number of instances of the class) and recall (how many instances of a class are correctly predicted compared to the size of the class) and can therefore make an accuracy measure that gives more insight into where the model can improve.

Bibliography

[1] Max pooling. URL https://computersciencewiki.org/index.php/Max-pooling_/_Pooling.

[2] Szegedy C. Et Al. Going deeper with convolutions. *IEEE Xplore*. URL https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43022.pdf.

[3] Md. Zahangir Alom, Tarek Taha, Chris Yakopcic, Stefan Westberg, Paheding Sidike, Mst Nasrin, Mahmudul Hasan, Brian Essen, Abdul Awwal, and Vijayan Asari. A state-of-the-art survey on deep learning theory and architectures. *Electronics*, 8:292, 03 2019. doi: 10.3390/electronics8030292.

[4] Pinkowski B. Principal component analysis of speech spectrogram images. *Elsevier*, 1997. URL https://www.sciencedirect.com/science/article/pii/S0031320396001033.

[5] D. H. Bailey. *A High-Performance FFT Algorithm for Vector Supercomputers*, volume 2. 1988.

[6] D. H. Bailey. *FFTs in External or Hierarchical Memory*, volume 4. Journal of Supercomputing, 1990.

[7] K. Bochner, S.; Chandrasekharan. *Fourier Transforms*. Princeton University Press, 1949.

[8] R. Cassani. Multilayer perceptron example, 2017. URL https://github.com/rcassani/mlp-example.

[9] Stathakis D. How many hidden layers and nodes? *Taylor & Francis*.

[10] Krizhevsky A. et al. Imagenet classification with deep convolutional neural networks. *NIPS*, 2012.

[11] McLaughlin N. et al. Recurrent convolutional network for video-based person re-identification. *IEEE*, 2016.

[12] Survivor (Frankie Sullivan & Jim Peterik), 1982. Album: Rocky III. Label: EMI.

[13] Doetsch P. Ney H. Golik, P. Cross-entropy vs. squared error training: a theoretical and experimental comparison. 2013.

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

[15] Google Inc. Tensorflow library. URL https://www.tensorflow.org/.

[16] Google Inc. Tensorflow callback: Early stopping, 2020. URL https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping.

[17] Google Inc. Tensorflow callback: Reduce learning rate on plateau, 2020. URL https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ReduceLROnPlateau.

[18] Andrew Jahn. Introduction to independent components analysis, 2014. URL http://andysbrainblog.blogspot.com/2014/06/introduction-to-independent-components.html.

[19] Jeremy Jordan. Gradient descent., 2017. URL https://www.jeremyjordan.me/gradient-descent/.

[20] Hornik K. Approximation capabilities of multilayer feedforward networks. *Elsevier*, 1990.

[21] Salimans T. Kingma, D. P. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *NIPS*, 2016.

[22] Pham P. Y. Ng A. Lee H., Largman Y. Unsupervised feature learning for audio classification using convolutional deep belief networks. *NIPS*, 2009.

[23] T. Mikolov et al. Recurrent neural network based language model. *INTERSPEECH 2010*, 2010. URL https://www.isca-speech.org/archive/archive_papers/interspeech_2010/i10_1045.pdf.

[24] University of Oslo. Audio compression in practise. URL https://www.uio.no/studier/emner/matnat/math/MAT-INF1100/h07/undervisningsmateriale/kap6.pdf.

[25] C. Olah. Understanding lstm networks, 2015. URL https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[26] van Kooten M. Sarkovic V. Por, E. Nyquist–shannon sampling theorem. 2019. URL https://home.strw.leidenuniv.nl/~por/AOT2019/docs/AOT_2019_Ex13_NyquistTheorem.pdf.

[27] Benny Prijono. Student notes: Convolutional neural networks (cnn) introduction, 2018. URL https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/.

[28] R. Rojas. *The Backprogation Algorithm*. Springer-Verlag, Berlin, 1996.

[29] Gerry Saporito. What is a perceptron?, 2019. URL https://towardsdatascience.com/what-is-a-perceptron-210a50190c3b.

[30] P. N. Swarztrauber. *Multiprocessor FFTs*, volume 5. 1987.

[31] Tektronix. Understanding fft overlap processing fundamentals. URL http://tw.tek.com/dl/37W_18839_1.pdf.

[32] O. Trekhleb. Playing with discrete fourier transform algorithm in javascript, 2018. URL https://dev.to/trekhleb/playing-with-discrete-fourier-transform-algorithm-in-javascript-53n5.

[33] M. Van Gerven. Modeling human brain function with artificial neural networks. URL https://www.natmeg.se/onewebmedia/Marcel%20van%20Gerven%20-%20Machine%20learning%20-%20Karolinska.pdf.

[34] J. Zhu. Cs194-26: Image manipulation and computational photography. URL https://inst.eecs.berkeley.edu/~cs194-26/fa17/Lectures/ConvEdgesTemplate.pdf.