KING'S College LONDON

University of London

Progressive adversarial music generation

MSci Computer Science - 7CCS4PRJ Final Project Report

(published version)

Supervisor: Professor Mischa Dohler

April 5, 2019

Abstract

This paper investigates different ways of generating pulse-code modulated (PCM)/raw musical audio with deep learning methods and presents an approach based on progressive generative adversarial networks that can produce results with state-of-the-art receptive fields. To facilitate this, the report gives a brief overview of the history of machine/deep learning, how sounds work in the real and the digital world, explores other approaches to neural audio generation and gives both a higher level and a detailed, in-depth view of deep learning techniques. The implementation chapter explains the base architecture of the program and elaborates on the improvements introduced to adversarial music generation, inspired by other deep learning areas. A few experimental models are also briefly presented, among them the first ever progressive variational autoencoder - generative adversarial network hybrid (VAE-GAN). Evaluating music generating networks is a non-trivial task but the report presents a few different ways, along with a large amount of generated audio samples included with the submission. The shortcomings of the current project are also explained and directions for future work are laid out. Appendix A presents a few of the more interesting generated samples along with a short analysis for each and appendix B explains how the project environment can be set up and how the program can be run.

Acknowledgements

As always, I would like to thank my family for their unending support and my deepest thanks go to my supervisor, Professor Mischa Dohler, for his encouragement and assistance during the project whenever I needed it.

Contents

Chapter 1

Introduction

This project aims to research different methods of generating raw (also known as pulse-code modulation - PCM) musical audio with deep learning methods. As opposed to MIDI (Musical Instrument Digital Interface) generation this particular sub-field is comparatively under-researched [1]. One of the main reasons for this is that MIDI audio has far fewer possible parameters than a PCM audio file and thus training PCM machine learning models is much more difficult. This project presents ways of improving on the current state-of-the-art and in order to be easily understandable it also touches on machine/deep learning and audio programming concepts.

1.1 Report structure

After this introductory chapter, the final report is structured into 3 major sections.

Chapters 2, 3 and 4 deal with the background: the history of machine and deep learning, the domain of the problem and the motivation for the project. Sounds are explored - both in the real world and in the digital. Finally, a high-level overview of general deep learning concepts and hardware is given.

Chapters 5, 6, 7 are the implementation: both the software and the neural network design are described as well as the requirements of the software and the libraries used. Afterwards, the different parts of neural networks and the way they work is explored in-depth. Here, some basic concepts are presented along with the more advanced ones used in this project. The

Implementation section deals with setting up the environment and the development of the supporting features but mostly it describes how the state-of-the-art can be reached and how it can be improved. Ideas from similar fields are also evaluated on the problem domain.

Chapters 8, 9 and 10 serve as the conclusion: the results of the research are presented, along with the shortcomings and the direction for future work. Since this research deals with artificial intelligence (AI) it is also crucial to observe what effects AI research projects might have on humans - in this case mostly musicians. Finally, a short concluding chapter summarises the report.

1.2 Contributions

To finish the introduction, here is a short overview of the contributions of this project:

- A novel, progressively grown generative adversarial network (GAN) based on WaveGAN [2] and PGGAN [3] is developed that is capable of generating samples with state-of-the-art receptive field for this network type. A way of smoothing the number of feature maps for later layers in the network is also presented.
- An approach inspired by transfer learning is used for transitioning between progressive stages. This approach uses layer freezing during transitional stages, which is simpler to implement, grants faster training during transitional stages and makes the model more flexible as well as more suited to audio data compared to the ”fade in” approach used in PGGAN.
- Mixed precision training [4] is introduced for audio generation. It provides a roughly 5% speed-up compared to single-precision training on cutting-edge hardware, with no noticeable downsides.
- Stereo training and generation is demonstrated for the first time.
- A form of data augmentation is described for audio GANs: by slightly offsetting the starting position of the data when slicing it up for batching it is possible to create additional unique slices from the training data. This approach has been implemented independently before (v2 of [2] most likely supports it) but as far as can be told it has not been described in a research paper previously.

- The first progressive variational autoencoder (VAE) and GAN hybrid is developed and briefly discussed. Early experimental results showed better musicality and slightly better quality than the main model, achieved in fewer iterations. Extended, experimental versions of the main model are also presented.

Chapter 2

Background

2.1 Glossary

- Artificial intelligence (AI): a sub-field of computer science that deals with decision making based on a limited amount of information available. The goal is to model the ways an intelligent agent would deal with the given problems, for example playing a board game.
- Neural networks: algorithms that contain parts that aim to mimic how neurons work in the human brain. These artificial neurons can be stacked in order to create a deep neural network.
- Machine learning (ML): A sub-field of AI and statistics. By feeding training data to an algorithm it is possible to train said algorithm to be able to classify new and unseen data or to predict future values based on past inputs. This term is sometimes used interchangeably with deep learning.
- Deep learning (DL): A special field of machine learning that uses neural networks for accomplishing complicated machine learning tasks. There are three main use cases for deep learning: classification, regression and generation. These shall be elaborated on in further sections. The literature is not consistent with the usage of the terms ”deep learning” and ”machine learning” so for clarity in this report ”deep learning” shall be used when referring to machine learning models that use neural networks and ”machine learning” is used as an umbrella term for both machine and deep learning.
- Pulse-code modulation (PCM): a method of turning analog signals into a digital repre-

sentation. This is, for example, the data found on an audio CD and if it is uncompressed it can also be referred to as a ”raw” digital signal. In order to be straightforward, in the rest of the report PCM audio is mostly going to be referred to as ”raw” audio.

- Musical Instrument Digital Interface (MIDI): A way of representing musical notation digitally and generating music from said notation. It can lead to good results for some instruments (piano) but it lacks the details that recorded physical instruments provide and it is not suitable for other types of audio (for example, human speech).

2.2 A brief history of machine and deep learning

The history of machine learning began in 1943, when McCulloch and Pitts published their paper [5] in which they explored how a neuron can be implemented in computer hardware of that era. One downside of the neurons presented in the paper is that - unlike modern neural networks - they were not able to learn. This problem was solved in a paper published in 1958, by Frank Rosenblatt [6]. This paper formed the basis of modern neural networks by introducing the perceptron - a model of a neuron that is capable of learning by containing weights and biases. These can be optimised by training the model on some data - just like in modern machine learning. The general idea of perceptrons is still in use, however, nowadays they are commonly called ”neurons” or ”units”. As revolutionary as this idea was, not everyone agreed that machine learning is going to be a relevant technology in the future. In 1969 Marvin Minsky and Seymour Papert published a paper called ”Perceptrons” [7] in which they showed that perceptrons are only capable of learning simple functions.

This book tempered the enthusiasm about machine learning for most of the 70s and the early parts of the 80s. It has to be mentioned that one key idea called backpropagation was proposed for neural networks in 1974 [8] and implemented in 1982 [9]. This idea allowed neural networks to ”backpropagate” the error to earlier layers, not just the final ones. It is still in use today, as it allows every layer of a neural network to learn from the training set.

In the 80s and 90s many of the now-popular neural network types were invented and saw limited use, for example convolutional neural networks [10], recurrent neural networks [11] and autoencoders [12]. The problem at this time was that these models were slow to train and suffered from a so-called ”vanishing gradient problem”. This meant that gradients of the

network (the values that tell the network how it should change its weights) ended up being so small that the network made no changes at all. This prevented the network from learning anything from that point.

The solution to both of the problems came in the 2000s. Slow training was solved by better utilising computational resources, by using the graphics processing unit (GPU) of the computer [13][14]. The latter problem was diminished in part by using a new activation function called "rectified linear unit" or ReLU [15]. The purpose of this function is to determine when a neuron "fires" and what value it should have. The ReLU function is rather simple: $f(x) = max(0, x)$ but still managed to improve on the problem. For the generative deep learning domain one of the most significant innovations happened in the 2010s: the introduction of generative adversarial networks (GANs) [16] in 2014. As the name implies, they are specifically tailored for generative tasks, which is also the domain of this project.

2.3 Problem domain and motivation

This project deals with generating music from raw musical audio training data. Due to the high amount of data that has to be "remembered" by the neural network this is a highly computationally intensive task. By using specialised neural networks this process is viable for pieces that exhibit stylistic consistency for a few seconds. However, due to the nature of neural networks this process eventually breaks down due to computational costs. Recently there have been new techniques introduced in the generative domain. For example, Dieleman et al. significantly improved on the previous state-of-the-art in their 2018 paper [1]. They used autoencoders, a special type of neural network for the first time in audio generation. They had been used in generative modelling for other fields (for example Nvidia used them to reconstruct a noisy digital image [17]). Due to their results it is worth investigating other, cutting-edge deep learning innovations as well, for example deep generative adversarial neural networks. Before breaking new ground, however, it is also important to investigate the state-of-the-art. Due to the different methods with different strengths and weaknesses it is not easy to decide what the state-of-the-art is. A later section in the report covers this.

Other ways of generating music have been explored in the literature, for example by using a MIDI representation of musical pieces [18]. The advantage of this is that sheet music can be generated so instead of sound samples it is necessary only to generate the individual notes.

This means that both the number of possible values and the frequency at which these have to be generated are significantly lower. Due to that fact this way of generating music is much more efficient and it is a viable alternative if the aim is to generate sheet music it has received more attention than raw generation. Even in the best case, however, MIDI cannot accurately represent the nuances of real music, thus if the aim is to generate music that sounds "real" it is necessary to use raw audio. Furthermore, another advantage is that, unlike MIDI, this method could potentially be easily adopted for other kinds of audio generation. As the conclusion of the introduction, let us examine what sounds are - in the real world and the digital.

Chapter 3

Sounds and audio programming

3.1 Sounds in the real world

Sounds in the real world are nothing more than waves in the air, cast by objects that emit the sound [19]. These objects disturb and compress air around them and thus create sound waves. When these waves reach an observer the observer can interpret these waves. The main attributes of the waves are sound pressure and frequency. Sound pressure is usually measured on a logarithmic scale called decibel (dB) and it signifies how much energy a sound wave carries, i.e. how loud the perceived sound is. Frequency signifies how quickly a sound wave repeats. For the observer this means that a sound with lower frequency is going to be interpreted as a low, bass-heavy sound while higher frequency sound waves are high, treble-like. Frequency is measured in hertz (Hz) and 1 Hz means that the given wave repeats once per second - this would sound like a low, rumbling sound, however, humans can generally only hear sounds between 20 Hz and 20000 Hz [20] so a sound with a frequency of 1Hz would not be audible.

3.2 Digital audio representation

From the previous description it can be seen that sounds are a continuous phenomenon in the real word. This can be captured easily on analog media by using a continuous/analog signal. This signal does not have a specified range of values it can take, thus translating it to a format that a digital computer can understand requires a few modifications.

In order to turn an analog signal into digital the signal is sampled at given points in time and the retrieved value is turned into a fixed-range representation. This represents one sample point of the audio signal. In order to approximate a continuous signal a large amount of measurements have to be taken each second. One popular sampling rate is 44.1 kHz, which is also generally used on audio CDs [21]. There are 44100 samples taken every second, then each sample is "classified" into one of the available amplitude values. These available values are given by the bit depth. This simply means how many distinct values the samples can take. In the case of 8 bits it is $2^8 = 256$, which is generally not enough for good quality audio. Audio CDs usually use 16 bits, which results in $2^{16} = 65536$ possible values.

Aside from these changes digital audio follows the rules of audio in real life. Sounds that carry more energy, that is, have a higher sound pressure are louder in real life and in the case of digital representation this corresponds to samples that are farther from 0. Waves that repeat more often are perceived as higher, treble-y in both cases. From this it can be seen that a higher bit depth makes the audio representation more detailed and a higher sampling rate makes it possible to accurately represent high frequencies.

3.3 The challenge of music generation

Why is music from raw audio challenging to generate? It boils down to a few different factors. The major one is that it exhibits structure on multiple different time scales. Raw audio files capture the amplitude of the recorded sound wave many thousand times per second, thus multiple samples are needed to represent even a millisecond of an audio file, which, in itself, is almost imperceptible to humans. On a larger scale, melodies, notes, chords or riffs play out over several seconds. Songs are made up of these over minutes, where a riff that appeared in the beginning of the song might also appear near the end, hundreds of thousands (or even millions) of samples later than it appeared for the first time. Teaching a neural network both the short- and long-range dependencies is still an open problem and due to how challenging it is, it is still rather under-researched. Models and research papers dealing with raw audio only started appearing in 2016 (with WaveNet [22]) so, as shown in later chapters, there is still plenty of room for improvement.

Chapter 4

Neural networks and deep learning

4.1 What is deep learning?

As stated in the glossary, deep learning is a branch of artificial intelligence and machine learning. It relies on feeding massive amounts of training data to a specially designed network that aims to mimic how the human brain works. The goal of the network is to make predictions that are close to the training data. Based on the error of these predictions the network can adjust (optimise) the different variables it contains, in order to minimise the error. For example, in the case of an image classification neural network, it could be asked to make predictions for every training image. A well-written network is able to minimise the amount of pictures it incorrectly classifies by modifying the variables it contains, based on the knowledge that it guessed right or wrong for every picture. This is an example of supervised learning. If the neural network is not told if it guessed right or wrong after a prediction it has to figure out the connections within the data to make guesses. This is unsupervised learning. If the neural network is not explicitly told if the guess is right or wrong, but it is given a reward based on the correctness of the guess it aims to maximise the rewards it gets. This is reinforcement learning.

It is also worth taking a look at the different types of neural networks, which is the topic of the next section.

4.2 Neural network types

Depending on the purpose of the neural network there have been multiple different types developed. Let us look at three of the major ones. Do note that these types are not mutually exclusive. For example, generative adversarial networks are usually based on convolutional networks.

4.2.1 Convolutional neural networks

Convolutional neural networks (CNNs) [10] contain ”convolutional” layers, the purpose of which is to extract specific types of information from inputs. These neural networks are usually used in the computer vision domain, so their convolutional layers can be trained to recognise different features of human faces, for example. They have also seen usage in generative models. One relevant example is an audio generating CNN called WaveNet [22], which is covered in the state-of-the-art section.

4.2.2 Recurrent neural networks

Recurrent neural networks (RNNs) [11] work by repeatedly returning to a specific layer and training that layer when new information is available. This makes them appropriate for time-series data, digital signals or when global data awareness is needed. An example: in an audio file the value of a specific sample depends on every sample that came before the current sample.

The problem is that these RNNs generally need to be as deep as the ”memory” of the network, i.e. there has to be a significant amount of layers in an (unrolled) RNN otherwise the network is only going to remember a limited amount of information. Making a neural network deeper, however, has a significant performance hit. That is the reason why the receptive field (the length of information an RNN can remember) is an important metric. Another problem is that RNNs are less optimal for GPU acceleration - since they process inputs as sequences it is harder to parallelise training, which means that they are likely to be slower than CNNs. This is because CNNs can process a chunk of data in parallel and thus use the GPU optimally. Recurrent neural networks can be used for regressive and generative tasks.

4.2.3 Generative adversarial networks

As their name implies, generative adversarial networks (GANs) [16] are mostly used for generative tasks. These neural networks usually have two main parts: a generator and a discriminator. The generator is responsible for creating new data based on a random input and a discriminator, whose purpose is to tell if the generator produced valid data. For this, the discriminator is trained on the training set. The idea behind this network type is that the generator and discriminator "play a game" as their training and as they play more both of them become better over time. The discriminator learns from the training set so it is going to become better at recognising the "fake" data created by the generator. Based on the output of the discriminator, the generator learns what kind of generated data can fool the discriminator. After the training is finished the generator can create new data that is similar to the training data. GANs are revisited in later chapters.

4.3 Use cases of deep learning

4.3.1 Classification

Classification is one of the traditional deep learning use cases. Classification is usually used in the computer vision domain, where a deep learning model is tasked with identifying objects in an image and categorising those images into a finite set of classes. For example, a model might be trained to recognise cars and ships. In this case the model could tell if pictures contain either ships or cars and appropriately put said pictures into those classes. Classification generally belongs to supervised learning since training pictures often come pre-labelled, which means that the training data would be fed into the network with an appropriate label. For example, a cat picture would be sent to the network with the label "cat". CNNs are usually suitable for this task.

4.3.2 Regression

Regressive tasks in machine learning involve predicting a new value from previous training values. This could, for example, be predicting real estate prices over time or demand for a product based on specific parameters. RNNs are mostly used for this since they can be trained

on time-series data.

4.3.3 Generation

A rather broad field of deep learning and also the area of this project. The aim is to generate new data based on the training data. Among others, these can be a song where sound samples are generated, a noisy image where the aim is to remove noise by generating appropriate pixels at appropriate locations or maybe style transfer, where two works of art are merged into one final piece that exhibits features from both. Generative models can be based on all three of the previously mentioned neural network types, however, RNNs and GANs are the widely used ones. RNNs are useful since the model can learn temporal/spatial connections and GANs were invented for this specific use case - the generator can be trained to produce data that highly resembles the training data but does not exactly copy it.

4.4 Audio and deep learning

It is not trivial to find the absolute state-of-the-art for generative models, because different models can exhibit different strengths and weaknesses. Evaluating these models is also not an easy task. Ratings by human observers and the so called ”Inception Score” can be used. The latter means that the generated object is used as an input for a pre-trained classifier. If the object is correctly classified it means that the object can be accepted as a realistic example. This is useful for images, however, its applicability for general-purpose audio generation might be limited.

Because of this, this chapter is going to present the advanced models from each of the different areas (CNN, RNN, GAN).

4.4.1 The state-of-the-art

CNN

- WaveNet [22]: The first published model of the WaveNet family of models. It is an autoregressive convolutional neural network that operates on general raw audio waveforms.

In this case autoregressive means that the value of a generated sample depends on every previous sample but it does not depend on samples that come after it. This is implemented with causal convolutions. For performance improvements the causal convolutions are also dilated: the filter does not consider every input value, thus it operates on a rougher scale. Different layers have different dilation values. With these techniques it is possible to reach exponential receptive field growth with depth. The model can also be conditioned on other types of data (the identity of a speaker for text-to-speech, etc.). The training speed of this network is fast since it can be done in parallel, however, since it is an autoregressive model, at generation-time the newly generated sample has to be fed back into the model, which can only be done in a slower, linear fashion.

- Parallel WaveNet [23]: This model is explicitly focused on text-to-speech (TTS). It is, however, based on and improves on WaveNet so it is worth investigating. This model uses a pre-trained WaveNet model as a "teacher" and another model as a student. The teacher scores the output of the student network. The aim of the student network is to minimise the difference between its probability distribution and the probability distribution of the teacher network. The paper also mentions that this idea is rather close to generative adversarial networks, the two main differences being that the training is not adversarial (since the student tries to match the teacher) and the teacher is also not trained, unlike in a GAN where both the generator and discriminator are trained.

- WaveNet with autoencoder [1]: The final WaveNet paper described in this report and one of the very few research papers that explicitly deals with music generation from raw audio. The model described is based on the original WaveNet architecture. More precisely, the WaveNet architecture is used as a sub-model for parts of an autoencoder. An autoencoder is a type of neural network that tries to reproduce a given input sequence. Traditionally, an autoencoder has two main parts: an encoder and a decoder. The encoder constructs a coarse (lower resolution) representation of the input data and the decoder learns to recreate the original input from the lower resolution data as best as possible. This model is more complicated than that, however. It consists of three sub-networks and as previously stated, each of these are based on the WaveNet architecture. The encoder fulfils its traditional function and generates a coarse (in the paper "higher-level") representation of the input. In this case its output is continuous, so a "quantisation module" turns this into a discrete representation. The decoder itself consists of a modulator and a local model. The local model fulfils the role of a decoder in a traditional network, that is, it

is responsible for recreating the original input as close as possible. The purpose of the modulator is to help the local model produce optimal outputs.

RNN

- SampleRNN [24]: This model is a recurrent neural network. The main idea presented in this paper is the idea of a network with multiple tiers that operate on different timescales. The full network has three tiers. The lowest one operates on sample-level data while the higher ones operate on "non-overlapping frames" (i.e. a certain amount of samples grouped together). The output from higher levels is used as conditioning data for the lower levels. This model also uses "output quantisation" with 256 possible values, which corresponds to a bit depth of 8 bits in audio files. Despite this and the possibly relatively short receptive field (it was not discussed exactly) the paper mentioned promising results.

GAN

- WaveGAN [2]: One of the first successful applications of GANs to raw audio generation. The paper presents a CNN-based GAN that shares similarities with WaveNet [22] and DCGAN [25]. Traditional, image generation GANs frequently suffer from a "checkerboard artifact" problem. The paper finds that a similar issue is present for audio generation as well, which might lead to the discriminator rejecting samples way too easily. The proposed "phase shuffle" solves this problem. The downside of this model is the length of the generated audio samples, since they are limited to about one second[1]. The paper deals with general raw audio generation but it is mostly evaluated on a TTS domain. The paper also details another model, SpecGAN. This model aims to generate the spectrogram (an image) of a sound file, however, the quality trails behind WaveGAN.
- GANSynth [26]: Similarly to SpecGAN, this network generates spectrograms, since it is possible to apply knowledge from image generation GANs directly. Another advantage is the generation speed, since images can be generated in parallel, so it is applicable to spectrograms as well.

[1]This is true for the WaveGAN version available at the start of the project, the later released WaveGAN v2 supports audio generation up to 4 seconds.

4.4.2 Shortcomings of the state-of-the-art

Different approaches have different strengths and weaknesses. Some of them have long receptive fields but are marred by long training times and some audio quality issues (WaveNet Autoencoder [1] - CNN), have autoregressive training and generation (linear execution - rather slow) but audio quality is good (SampleRNN [24] - RNN). Finally, quick generation but limited receptive and generative field, along with some potential quality issues (WaveGAN [2] - GAN).

4.4.3 Similarity with other fields

Even though audio generation is a rather unique field, it does show some similarities with two other fields of deep learning: image and text generation. This is important to acknowledge, since techniques successfully used in those fields but never applied to audio generation can be adapted. Let us examine the similarities and differences with those two fields:

- Image generation: just as with audio, it is important for images to exhibit a kind of "global coherence". In other words, it is important that the different parts of the image make a logical, cohesive whole. The difference compared to audio is that the cohesion is spatial and images can be represented as 2D tensors - images have a height and a width, usually given in pixels. Each pixel can contain multiple different values, commonly one value each for red, blue and green channels. Audio is generally 1D time-series data and it has to be coherent over time. In other words, it has to exhibit temporal coherence.
- Text generation: it is a field of natural language processing that shares many similarities with audio generation. There are only a limited number of possible outputs, it is 1D time-series data and has to exhibit temporal coherence. The main difference is that text generation requires far fewer characters (samples) in order to learn an acceptable representation of the training data and generate a coherent output. For audio generation even for one second of audio at least 16000 samples are needed, more for better quality.

4.5 Hardware for deep learning

Deep learning tasks are well suited to parallelisation. For example, time-series data can be split into multiple parts then fed into the network at the same time whilst keeping most of

the temporal structure. This is done by feeding multiple streams into the network, starting from the point where the data was split. For classification tasks multiple images can be shown to the network at the same time. Deep learning is mostly matrix operations thus a processor well-suited for that task would be beneficial.

Conventional CPUs (Central Processing Units) are mostly optimised for linear, general-purpose computation. It is possible to define multiple threads but even a top-of-the-line consumer CPU can only run 12-16 parallel threads at the same time [27]. They are not optimised for matrix-matrix operations nor for massively parallel computations. Fortunately, there is a piece of hardware that suits those requirements.

The GPU (Graphics Processing Unit), as its name implies, originates from computer graphics. Generating a computer image, roughly speaking, can be made parallel by calculating the end value for every single pixel approximately at the same time. The way these pixel values are derived is by multiple matrix-matrix and matrix-vector operations. GPUs were designed to satisfy these requirements by utilising many, limited functionality cores ("shading units") that excel at linear algebra operations.

Nowadays GPUs are used to massively speed up deep learning training. Cutting-edge GPUs especially excel at 16-bit floating point calculations [28], however, that level of precision might be insufficient for some deep learning tasks. Because of this, there have been some techniques developed to use 16 bits for most of the training process, yet achieve accuracy close to 32 bits [4]. This shall be explored later.

4.5.1 Nvidia Turing for deep learning

Of the two major GPU manufacturers (AMD and Nvidia) Nvidia embraced deep learning much earlier than AMD, a trend which they continued with their new GPU architecture announced on 20th August 2018, called "Turing". Instead of only containing the aforementioned parallel shading units, these GPUs also contain "Tensor cores" [28]. These cores are specifically designed for deep learning acceleration, by speeding up matrix operations using 16-bit (half precision) floating point values. Traditionally, computer graphics mostly use 32-bit (single precision) floating point values so GPUs were much slower for FP16 operations compared to 32-bit ones: the previous flagship consumer card of Nvidia, the GeForce GTX 1080 Ti was capable of 177.2 giga floating point operations per second (GFLOPS) in FP16 [29]. The current flagship, the

GeForce RTX 2080 Ti is capable of 26895 GFLOPS in FP16 mode [30], not including the Tensor cores. However, deep learning does not necessarily need FP32 operations, especially since the invention of mixed-precision training. Additionally, even the shading units support FP16 operations, at double the speed of FP32 operations. AMD's latest architecture (Vega) also supports this [31], however, with the use of the Tensor cores a theoretical computational speed of above 100 TFLOPS (tera FLOPS, 1 TFLOPS = 1000 GFLOPS) can be achieved. In real-world usage it was shown to translate to about 83 TFLOPS on an earlier generation high-end datacentre card that also has Tensor cores [32]. Additionally, now even consumer-oriented GPUs are fitted with both the double-speed FP16 shading units and the Tensor cores. This means that this new GPU architecture is much more deep learning oriented than GPUs have traditionally been and should also be significantly faster in deep learning tasks.

Chapter 5

Design & specification

5.1 APIs and libraries

The technology stack for this project is supposed to be fully platform-independent but most of the development work is done on Microsoft Windows (7/10) and Linux (Ubuntu 18.04) operating systems. Training deep learning models can be a rather resource-intensive task so GPU acceleration is also utilised, by Nvidia GPUs. The technology stack is as follows:

- Python 3.6: Python is a general-purpose programming language. Many deep learning libraries are based on or have the most mature APIs for Python so it should be the chosen programming language. Two main versions of Python are in use: 2.7.x and 3.6.x. 2.7.x is slowly getting deprecated and the author is more familiar with 3.6.x so that is the version used.
- Tensorflow r1.12: it is a tensor processing, machine learning and deep learning library that provides both high and low level APIs. It also provides GPU acceleration by default for models that are suitable for it. Version r1.12 is used. The main competitor of Tensorflow is PyTorch, however, Tensorflow is more widely used in the industry and has a larger community, which leads to more widespread support. Tensorflow is also backed by Google. Google DeepMind, one of the most prominent organisations researching the domain of the current project is also using it [33].
- Numpy 1.15: adds extended array/matrix support as well as extended mathematical

functionality to Python.

- SciPy 1.1.0: a scientific library for Python. The whole package also encompasses the previous library, however, it also provides more advanced audio (.wav) file and audio sample handling than the built-in Python "wave" library.

5.2 Design for deep learning

Tensorflow works with a programming style called "dataflow programming". The computational graph (model) has to be completely defined before any operations are run. Data also has to be loaded, processed and passed to the graph as input. This separates the main structure of a Tensorflow program into 3 parts. This section explores the concepts behind a general deep learning program, the next subsection is going to explain the software design of the current project.

The first part of the program is data preparation. Neural networks are good at handling different sizes of tensors containing floating-point numbers so the first task of the program is to transform the input data into a suitable representation. Additionally, depending on the type of the network and the input data, it might be also necessary to divide up the data into a number of parts, called "batches". The idea behind batches is that they can be processed in parallel, thus speeding up training.

The second part is the graph building. The first stage is defining the neural network part of the graph. This is a bigger step and thus it shall be explored in its own subsection. The second stage is defining a loss function. This function tells the network how "correct" it is with its guesses. It is important to choose a loss function that is suitable for the data, since the result of this function tells the network how far it is from a correct result. The last stage is the optimiser. This function usually takes a learning rate and an objective to minimise through applying gradients to the network. A typical optimiser aims to minimise the output of the loss function.

The third part is running the graph, or in other words, training. This is done through running the training loop inside a Tensorflow session. The optimiser gradually adjusts the weights and biases within the network to reach a more-and-more accurate solution.

For generative models it is also important to let the user observe some of the output. This can be done by taking the output of the neural network and turning it back into a human-observable format.

5.2.1 Software design

A well-defined software architecture/design is crucial for general software projects, however, that is not necessarily the case for a deep learning research project. The reason why software design is useful in most cases is because there is a need to navigate and understand a sizeable code base. Deep learning programs are generally shorter, more concentrated thus the advantage of traditional software design techniques (for example, UML diagrams) is reduced. Having said that, in general it is possible to split the program into a few distinct parts. Do note that the programming language of the project allows functions outside classes. This is utilised, so the UML diagram is grouped by separate files. Within these files the contained classes are represented. If the file has no classes the purpose of the contained functions is shown:

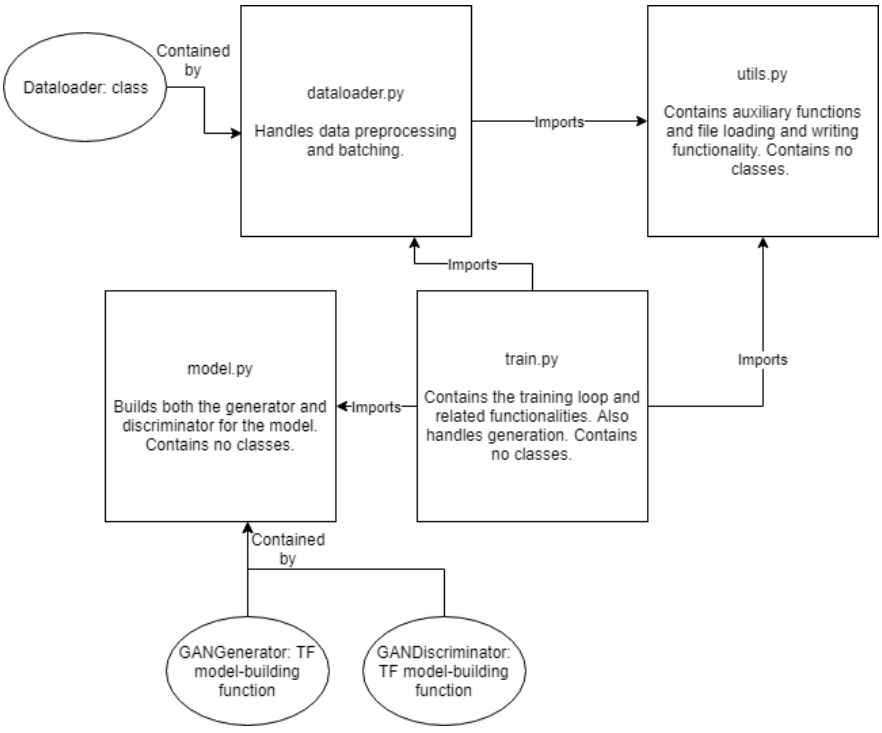

Figure 5.1: The design of the system.

As can be seen, the program can be divided into 3 major parts. The model.py file contains functions that build different parts of the neural network. This includes the layers of neurons that make up the network - including the trainable variables. For a regular neural network

there would be one function that builds the Tensorflow graph. It can be called with some input data and returns the output of the network. For a GAN, this file has to contain two such functions. One of them is the generator network: it takes a certain amount of random data points, called the ”latent vector”, passes it through a series of upsampling 1D convolutional layers and returns the final result. The discriminator is also located here: it takes a certain amount of input values and outputs a prediction that describes how certain the network is that the input was fake or real. The main training loop is defined in its own file (train.py) so it is possible to call that from the command line. This part of the program conducts a continuous training loop and also handles model saving and loading. The latter functionality is also useful for staggered training: conduct a few training loops, stop the program then return to training at the desired time. The data loader/preprocessor is in a separate file, within a class of its own. This class is also responsible for ”batching” the data. The user can define a few options as command line arguments when running the advanced train.py file variants, for example ”train_progressive.py”. Since generation relies on information from the rest of the program it was decided to have the generation functionality as a specifiable command line option for the advanced train.py file variants. There are multiple model.py and train.py file variants, for the different networks developed and two dataloader.py variants, one for the advanced models, one for the initial model.

There is also the utils.py file which is not considered to be a major part, as it only contains a few smaller utility functions.

This is a fairly standard architecture for a generative model. Where deep learning research projects can truly distinguish themselves lies within the contained neural network. Let us examine this in the next section.

5.2.2 Neural network design

There are multiple factors that have to be considered when designing a neural network. The first is the problem domain itself. Well-researched problem domains usually have their recommended network types, however, this is not always true for other domains. In that case it is worth investigating how the different network types are applicable to the problem domain and if previous research exists, how promising the results are and what shortcomings are present.

Another important consideration is training/inference speed and how good the results of

the network are. By stacking more layers training time increases but generally the network is able to hold more information, learn more complicated functions that describe the training data better and thus perform better at its task. It is important that these two aspects are balanced well. A network that trains too slow might be unusable, a network that trains fast but does not produce good results is rather pointless.

A network usually only contains either convolutional or recurrent cells, in many, stacked layers. It shall be mentioned, however, that convolutional layers can be made recurrent. For example, this technique appears in [17].

For normal neural networks overfitting is also a concern, however, GANs eliminate that problem. Since the generator of the GAN never receives real input it has no way of overfitting [34], since it never observed any real input. It only knows what a real image/piece of audio should look/sound like based on the guidance from the discriminator.

5.3 Requirements

The requirements of this program are fairly simple:

- The user must be able to initiate training by supplying a directory with one or more 16-bit .wav files to the program, with a sampling rate of 16 kHz.
- The user must be able to see the progress of training.
- The user must be able to interrupt training without losing training progress (up until the previous checkpoint).
- The program must be able to load 16-bit .wav files and transform these into a representation suitable for the neural network.
- The program must be able to train on well-formed training data.
- After the training is finished, the program must be able to generate audio files that are similar, yet different to the input files.
- The user must be able to specify a few options by specifying command line parameters, for example, whether to use data augmentation or mixed precision training. The full list is presented in the User guide.

Chapter 6

Neural networks in depth

In order to easily understand the Implementation section it is useful to know how neural networks work at a low level, which this chapter shall present. Of course, presenting every deep learning technique is outside the scope of this report, so it shall focus on the techniques relevant for the chosen network type - a progressive generative adversarial network, built with convolutional and transposed convolutional layers. The way this is done is covered in the Implementation section. GANs make use of a few rather advanced deep learning techniques so as an introduction to these a few of the more common techniques are presented as well. First, let us examine what lies at the depth of these networks: matrices and tensors.

6.1 Tensors

A matrix represents a 2D array of elements - a ”list of lists”. A tensor is simply a generalised matrix, where its dimensionality is called its ”rank”. Tensors can be 0 dimensional (a scalar), 1 dimensional (a vector), 2 dimensional (a matrix), 3 dimensional or even more. Why is this useful? A deep learning model learns to represent data by adjusting millions of variables. These variables can be stored in a structured manner and sometimes it is convenient to use a representation with a higher dimensionality than a simple matrix could provide. For example: 64 instances of a given training data, with 2 channels per instance and 16000 samples per channel are given to a deep learning model at the same time. There is no easy way to represent this data with a matrix but with a tensor it is simple: just use a tensor with shape (64, 16000, 2). If a single instance is taken from the outer dimension a rank 2 tensor is received with

shape (16000, 2), which is basically a matrix storing all of the samples for both channels for a single instance. If a value is taken from the outer dimension a sample pair is received, which can correspond to, for example, the amplitude of an audio file at a given point in time for both the left and right channels. Neural networks generally expect data to adhere to a given dimensionality and range of values said data can take, which might not correspond to the way the input data is stored, so there is a need to correctly load and preprocess data. Let us examine that in the next section.

6.2 Data loading and preprocessing

Neural networks can commonly operate on different sections of the input data, completely independently from other sections and the resulting outputs can be combined in a loss function. This is called ”batching”. It can lead to better hardware utilisation, which in turn leads to faster training speed. As shown in the previous section, it is also possible that the layers of the neural network have multiple channels. Some types of neural network layers use a large amount of channels where each channel corresponds to a feature that the neural network learned. For data that was originally 1D a format that can be used is (batch_size, sample_count, channels). In this case, each element of the batch stores the number of samples equivalent to sample_count and each of those samples stores a value for every channel. Batching can be done by shaping the data to consist of several tensors with the shape of (sample_count, channels) and simply taking the desired amount of these to make up a batch.

In order to aid how neural networks learn to represent data, it is useful to scale the data to zero mean with a standard deviation of 1, which corresponds to the floating point range of [-1, 1]. [0, 1] can also be used, but for audio data the former is a more convenient representation, since a 16-bit PCM audio file has sample values in the range of a signed 16-bit integer, so roughly from [-32767, 32767]. This is easily scaled to [-1, 1], by dividing the input value by 32767.

6.3 Activation functions

The purpose of neural networks is to learn a function that best represents the training data. Aside from trivial cases (e.g. linear data) this function is going to be non-linear. Each layer

of the neural network applies a transformation to the training data, but even a series of linear transformations is going to end up as linear. If a non-linear function is applied to the output of most or all of the layers, however, it becomes possible to model non-linear data. Let us look at a few significant activation functions:

6.3.1 ReLU

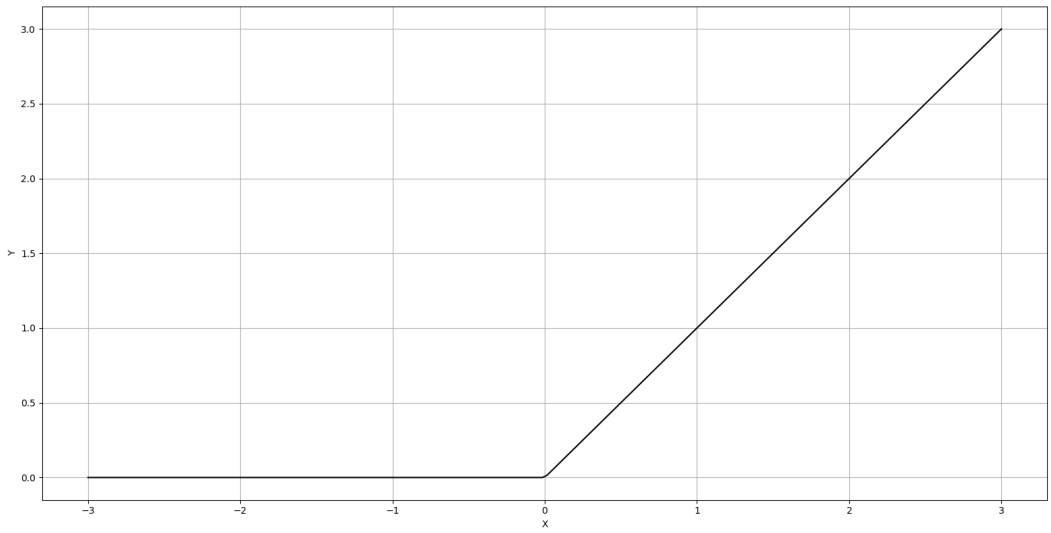

Figure 6.1: Rectified linear unit

The ReLU (or Rectified Linear Unit) [15] is a simple activation function:

$$f(x) = max(0, x) \tag{6.1}$$

If the input is below 0 it clamps the input to 0, otherwise it returns the input itself. Despite its simplicity, it is widely used since it managed to solve the "vanisihing gradient" problem that was prevalent in early deep convolutional networks.

6.3.2 Sigmoid

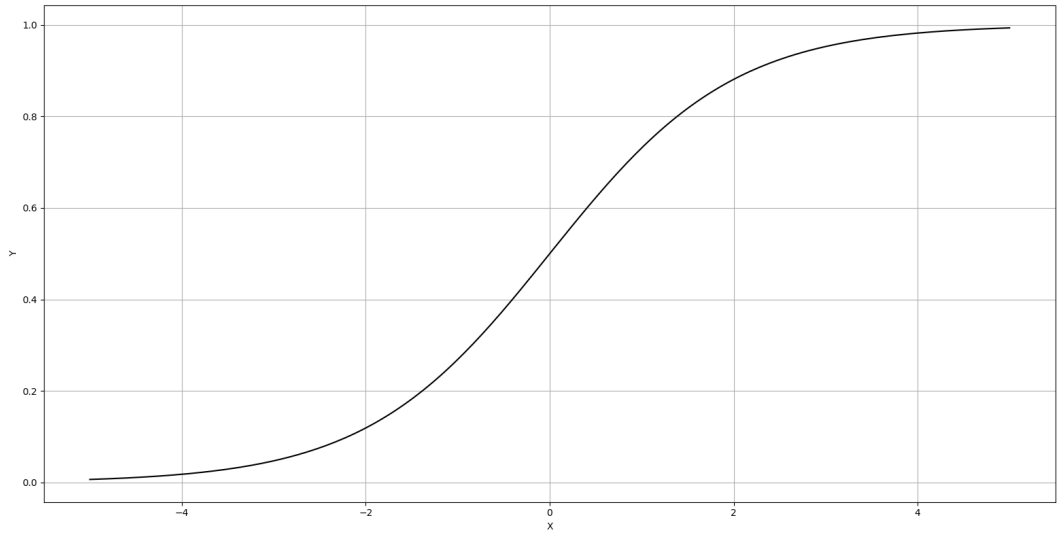

Figure 6.2: Sigmoid - the graph of the function is a smooth curve between 0 and 1.

The Sigmoid activation function is useful as the last activation function for networks where the range of possible values is in the floating point range of [0, 1], since the function maps an input value to the aforementioned range:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{6.2}$$

6.3.3 Tanh

Similarly to the Sigmoid function, the hyperbolic tangent (tanh) is useful as the last activation function for a network, but instead of the [0, 1] range this function maps values to the [-1, 1] range:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{6.3}$$

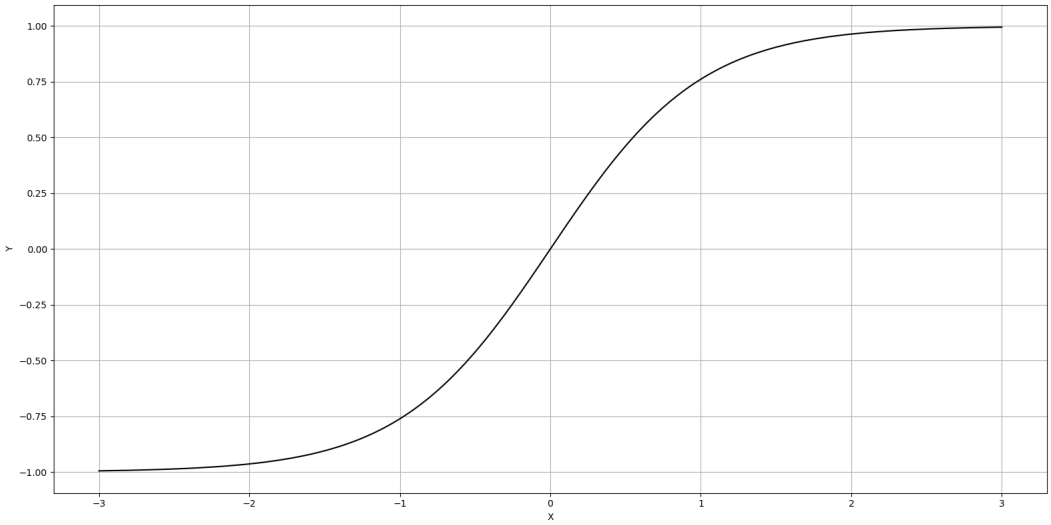

Figure 6.3: Tanh - the graph of the function is a smooth curve between -1 and 1.

6.3.4 Leaky ReLU

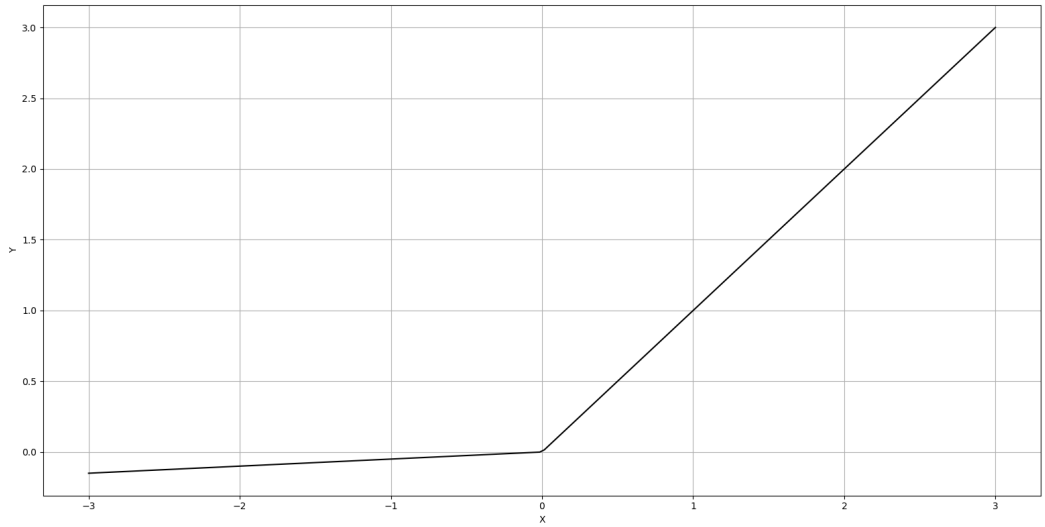

Figure 6.4: Leaky ReLU

The leaky ReLU [35] is a newer variant of the ReLU activation function that aims to solve the ”dying ReLU” problem: since the normal ReLU function sets every value below 0 to 0 it might end up outputting values that are locked to 0, effectively preventing the neuron to participate in the network in a useful capacity. The Leaky ReLU introduces a way to keep inputs that are below zero, but multiplied by a small term (α) so the function stays non-linear:

$$f(x) = \begin{cases} \alpha \times x \; for \; x < 0 \\ x \; for \; x \geq 0 \end{cases} \tag{6.4}$$

As important as the activation functions are, the layers that make up a neural network are equally important. Let us take a look at them in the next section.

6.4 Neural network layer types

In deep learning a layer simply means a number of neurons that take a common input tensor, update their stored values according to a specified operation and pass the resulting values to a new layer. This is a rather generic description since all of the layer archetypes operate rather differently. Recurrent layers are not going to be utilised for this project, so only fully connected and convolutional layers are described here. First, let us look at the simplest one, the fully-connected layer.

6.4.1 Fully-connected layer

Also called a "dense layer". The idea behind the fully-connected layer is that every input neuron is connected to every output neuron, with each neuron contributing an amount that is proportional to the weight assigned to it. This layer type combined with an activation function can be used to build a full neural network and in convolutional networks it can be used as the output layer since it is possible to connect a layer with many neurons to one or more output neurons. The range of the output values depends on the activation function used. Alternatively, a linear activation can be used so the network would return "logits". Logits are the raw output of the network. They are unbounded and first an activation/normalisation function is commonly applied to them so they can be used for calculating the loss of the output. Returning logits from the network allows greater flexibility when applying the loss function, since Tensorflow has variants of loss functions that take logits and first apply the chosen activation function then calculate the loss. For example: "tf.nn.sigmoid_cross_entropy_with_logits".

6.4.2 Convolutional layer

The purpose of convolutional layers is to detect features within the input. The idea behind convolutional layers with different dimensionalities is largely the same, however, 2D convolutions are rather intuitive so let us use them as the example here.

Just like a fully-connected layer, the purpose of a convolutional layer is to take a given amount of inputs, apply some operations on it and return an output. The applied operations are completely different, however. A convolutional layer operates on data with multiple channels. It uses each channel to identify/extract a different feature of the input. Because of this reason, channels are also called "feature maps". For example, in the case of an image of a car one channel could identify the parts of the image that represent the wheels of the car. Many of these channels stacked on top of each other are able to represent most of the major features that characterise the input well.

How does the network know what to look out for? Convolutional layers contain "kernels", also called "filters". These are usually small matrices that contain trainable variables - weights and biases. The number of kernels depends on the desired number of channels for the output and the exact dimensionality of a kernel depends on the input - if the size of the kernel is 3×3 and the input has 5 channels, the dimensionality of each kernel is going to be $3 \times 3 \times 5$. If the output is set to have only 2 channels, there are going to be 2 such kernels in that layer. The first output value (which can also be called an activation) is obtained by first "placing" the kernel matrix on the first 3×3 square of the input matrix and multiplying the two matrices together element-wise, then this has to be done for each channel and the final value is going to be the sum of the values obtained from each channel. To obtain the next value, the kernel slides over to the next 3×3 area of the input and the process continues until the whole input is processed. This creates one output channel. If there is more than one kernel in the layer the process is the same for producing other output channels. What is different, however, the variables in the kernels. The network is responsible for optimising these variables so it can get the desired results.

This is the main idea behind convolutional layers. Just as with fully-connected layers, an activation function is usually applied to the outputs since the operations within the layer are linear. Stacking many of these layers allows the network to first learn coarse representations of the training data followed by finer, more accurate details. As mentioned earlier, the logic

behind 1D convolutions - which are suitable for time-series and audio data - is the same, the difference is that 1D kernels are used and the input/output data is a multi-channel vector, instead of a multi-channel matrix.

Depending on the parameters of the given convolutional layer, it might downsample input data. For example, it might output a 32×32 image from a 64×64 image input. This can be the desired behaviour of a layer in a classifier, which corresponds to one layer in the discriminator in a GAN. The generator network, however, has to upsample the input data. Fortunately, there is a variant of the normal convolutional layer that can do just that, called "transposed convolution".

6.4.3 Transposed convolution

Also commonly and incorrectly called a "deconvolution" - a deconvolution implies that it reverses a convolution, which is not the case here. A transposed convolution does a similar operation compared to a normal convolution but in order to create a higher resolution output some padding has to be introduced. In addition to this, convolutional layers support "striding": for a normal convolutional layer this means that if a stride higher than 1 is specified the filter is not simply moved by one column to the right (and at the end of the columns) one row down. When it moves one-one, it can be also understood as a stride of 1. If a higher number is specified the amount the kernel slides by is going to correspond to that number. This is useful for summarising the data, since it can significantly reduce the output size.

Compared to this, transposed strided convolutions create an expanded representation of the input data, on which the convolution operation is applied. This way it can learn to create a higher-quality upsampled version of the input. This is especially important for GANs, since the generator has to create a good quality output purely from the random latent vector and the weights and biases it learns during training.

6.5 Loss functions

A neural network has to have an objective it aims to achieve as best as possible. The loss/objective function serves this purpose. In a simple case, this function can take the output of the network and some real training data as its inputs and return a value that tells the network how

close its output/prediction was to the real data. Let us look at a few simple loss functions, followed by a more complicated one.

6.5.1 Mean squared error

First let us look at one of the simplest loss functions. This loss functions takes the predictions from the neural network and compares these to the real training values. It does this by subtracting the predicted value from the real value, squaring the result and averaging it. Because of the squared value it can be seen that this loss function can never be negative and that the lower the output of the loss function is, the better the predictions of the network are. This loss function is useful for regressive tasks even though it can be problematic since the squared term heavily penalises even a single heavily incorrect guess, which might lead to the network to overcompensate and end up with even worse outputs as a result.

6.5.2 Log loss

Logarithmic loss (log loss, cross-entropy) is a loss function most commonly used for classification tasks. This loss function is based on another loss function called the likelihood function. Let us take a look at that first.

The likelihood function takes a series of predictions and a series of real values. It calculates how likely a given event is to occur based on the predictions. For example, a model predicts that a given picture is 60% likely to be a cat - the real data can either be 1 (which means cat) or 0 (which means it is a dog). In this case the likelihood function returns 0.6. If we have another picture (a dog) and the model says it is 20% sure that it is a cat - so 80% sure it is a dog. The likelihood of independent probabilities is the product of said probabilities, which would be $0.6 \times 0.8 = 0.48$. If many, possibly millions of probabilities are multiplied together the resulting likelihood is going to be so small that floating-point accuracy issues would arise. The solution to this is to take the log of the likelihood and multiply it by -1: the ”-1” is only needed so smaller output values signify a higher accuracy. With these modifications the likelihood function becomes the log loss function.

6.5.3 Wasserstein loss

Let us look at one of the more complicated loss functions used in special network types: the Wasserstein loss [36]. Since the training of GANs relies on two separate networks it is not possible to use a loss function that only depends on the output of one network. One of the loss functions capable of handling two networks is the Wasserstein loss, also known as the Earth-Mover (EM) metric.

The Wasserstein distance means the smallest cost to transport one data distribution to another data distribution. The neural network can be made to learn the function that provides the "least upper bound" or the most expensive way to move one data distribution into another. This was needed since the normal Wassersetin distance is intractable. The end result gives a function that serves as the loss function for the discriminator, which is commonly called "critic" in a WGAN. The learned function also has to be 1-Lipschitz continuous, which means:

$$|f(x_1) - f(x_2)| \leq |x_1 - x_2| \tag{6.5}$$

This can be enforced by clipping the weights of the network to a small range, however, this was regarded as a sub-optimal method. Instead, a new version of the WGAN loss function was invented, the WGAN-GP [37].

A function is 1-Lipschitz continuous if all of the gradients have a norm of at most 1. In order to make the neural network learn a function that adheres to this, gradients above 1 are heavily penalised. With this loss function the discriminator is not trying to discern between fake and real images but it is returning a value that signifies the Wasserstein distance between real and fake data. This information is also passed to the generator and as this value goes down, the data distribution of the generator is going to resemble the real distribution more and more.

The way how networks learn has not been covered yet, so let us take a look at the parts that are responsible for it.

6.6 Forward pass, backpropagation and backward pass

6.6.1 The forward pass

As discussed earlier, neural networks take a certain kind of input and produce an output of a specified format. In this regard they can approximately be considered a more complicated variant of a function. As opposed to functions, neural networks also contain a number of variables that can be trainable. These variables - called weights and biases - influence the output of the operations defined by the computational graph of a neural network and thus also influence the final output.

The forward pass simply refers to passing an input value through the neural network and receiving an output, which is influenced by the aforementioned operations and learned variables. Afterwards, with this output and a specified loss function the loss itself is calculated. The goal of the network is to minimise this loss, which is done through a two-part process: backpropagation and the backward pass.

6.6.2 Backpropagation

In order for a network to learn, it has update all of its trainable variables in a manner that reduces the output of the loss function. For a large network this is a non-trivial problem. It is solved by finding the amount that the output of each neuron (including the specified activation function) contributes to the final value. For this to be possible, the derivative of the error with respect to the weight in question has to be calculated. The derivative of a function gives the rate the output of a function changes with respect to an input value. In the above case, the loss changes based on the value of a weight. This is called a ”gradient”.

During the process of backpropagation the neural network has to pass through the numerous functions the network might have, backwards from the output. Of course, these functions change how a weight contributes to the output value. Because of this, in order to calculate each gradient, the network has to utilise the ”chain rule”: this rule gives the derivative of a composite function. This composite function contains the functions the weight goes through until it reaches the output value, so to calculate the gradient (the derivative of the error with respect to the weight) the chain rule has to be applied to the aforementioned series of functions,

which are composited in the reverse order. The process is the same for biases as well. By the end of this process all of the gradients are calculated, so the only part of the learning process that is left is the backward pass.

6.6.3 Backward pass

In the simplest case, the variables are recalculated by multiplying the old value of the variable by the negative gradient. It is necessary to multiply by the negative gradient since the objective is to minimise the loss and the gradient gives the direction in which the variable increases the output of the function the most. This is not desirable for a minimisation problem but multiplying by the negative gradient solves it.

Additionally, the old variable value is not necessarily multiplied by the whole gradient, just a fraction of it. This fractional multiplier is called the ”learning rate”. This learning rate signifies how quickly the network learns to look out for input data that has specific features. A high value makes the network learn quickly but its ability to generalise is going to be limited and vice versa for a low value. This learning rate is a ”hyperparameter”: it is set by the programmer of the network. The challenge is to set a learning rate that lets the network converge but does not increase training time to unreasonable levels.

This leads to the next section - optimisers. The previous paragraph was, in fact, the description of how one of the simplest optimisers works, called gradient descent (GD). Note that all the data has to be evaluated before a single variable update can be done with this optimiser, unlike with the ones in the next section.

6.7 Optimisers

6.7.1 Stochastic gradient descent

Stochastic gradient descent (SGD) [38] is a variant of the standard gradient descent optimiser. The most significant change is that the variable update happens after every training iteration. Stochastic means the same for a process as ”random” does for a variable. SGD stochastically approximates the true/final gradient after every training sample seen and thus it might converge faster than standard GD. SGD has another variant as well, called minibatch gradient descent

(MB-GD). It is an intermediate solution between GD and SGD: it processes a small subset of the training data (a minibatch), then updates the variables. MB-GD is also commonly called simply SGD, since the minibatch training allows much more efficient use of computational resources than simply training on a single example at a time, so it has mostly taken over the SGD name.

6.7.2 Adam

Adam [39] stands for "Adaptive moment estimation". It is an improvement on SGD where the learning rate is not constant for the whole network, but it changes during training. Moreover, the learning rate is different for every trainable variable within the network. To do this it uses the "moment" of every variable. The first moment of a random variable is the expected value of the variable, to the power of one, the second moment is the expected value to the power of two and so on. The expected value of a random variable can be obtained by, for example, running a simulation multiple times. In a neural network the gradient can be considered a random variable and Adam estimates the expected value by keeping track of the exponential moving averages of the variables. To get the actual amount by which the variables should be updated Adam uses the first and second moment of the estimates and does the update round after a minibatch is processed. The advantage of Adam over SGD is that it is able to train models significantly faster, however, some models faced problems with worse convergence rates.

This concludes the part of a chapter that presents how neural networks work at a low level. However, it is a good place to dive deeper into two of the more unusual neural network types, which is the topic of the next section.

6.8 Special network types

6.8.1 Autoencoders and variational autoencoders

The idea behind the autoencoder [12] is simple: construct a model that takes an input of a certain size and downsample it to a bottleneck: a lossy compressed, latent vector representation of the input. This part of the network is called the "encoder". The other part of the network is responsible for recreating the original input from this latent vector by a series of upsampling

operations. This is the ”decoder”. The challenge for the network is two-fold: first, it has to learn a way to compress the input into a representation that the decoder can easily utilise; second, it has to learn how to reconstruct the original image from the compressed representation. The loss function of the network evaluates how well the network managed to reconstruct the original image. This network type can be used for multiple tasks: creating a lossy deep learning compression algorithm; improving images by super-resolution or removing noise, etc. As a real-world application, Nvidia's Deep-learning super-sampling (DLSS) is also based on an autoencoder network [40].

This is not a strictly generative network by itself. It only outputs data that is roughly identical to its input and a generative model should be able to output something similar to the data it was trained on, but not necessarily identical. That is the purpose of variational autoencoders (VAEs) [41]. The idea behind VAEs is the same as normal autoencoders, however, as mentioned earlier, their goal is different: they learn to model the data so the decoder can create previously unseen samples.

6.8.2 Generative adversarial networks

As explained in an earlier chapter, a generative adversarial network (GAN) [16] consists of two networks, a generator and a discriminator. Traditionally, the goal of the generator is to create output that makes the discriminator think that the fake output came from the real training data and the goal of the discriminator is to tell apart the output of the generator from the real data.

In terms of game theory, this kind of game is called a ”zero-sum” game: one side wins if the other side loses. Because of this, both sub-networks are trying to optimise their values to ”win”. As is common in deep learning, this is done through an iterative process: the discriminator learns to identify real data better and better and the generator learns to mimic this data more and more, with information passed on from the discriminator through the loss function. Convergence in these networks means that both sub-networks reach the Nash equilibrium, which means that neither sub-network can unilaterally improve its score.

The interesting part of GANs is that the input of the generator is a random hidden/latent vector, which means that the generator has no knowledge of what the real data is like, it only knows how well it can make the discriminator think that its (the generator's) output

is real. This way one of the problems with generative CNN and RNNs can be completely avoided: overfitting to the point where the network generates output memorised straight from the training data.

GANs are a rather new invention. This, unfortunately also means that there are some unresolved problems with this network type. The most significant problem is that these networks can be rather unstable during training. This means that even after a significant amount of training a given network might not learn to represent the training data properly. It is, however, entirely possible that a different training run of the same network is going to end up with acceptable results. Another problem is "mode collapse": the generator learns to produce a few samples that the generator always classifies as real. This way the variance of the generated output would be significantly below the variance of the training data. In other words, the generator would learn to repeatedly output only a small number of identical samples.

The output of the loss function of a GAN is also not necessarily indicative of the quality of the output of the generator. This is especially problematic when the output sample is not properly human-evaluable, for example, in the early stages of progressive training. Some GAN types also require long training times in order to produce optimal results: the aforementioned progressive GAN required 2 weeks of training on a cutting-edge GPU [42].

These shortcomings can be mostly mitigated by utilising training hardware that is fast enough to generate promising early results in a short amount of time. Additionally, the good quality results for image generation and because of the fact that there has only been one major GAN developed for raw audio generation it is worth considering this network type as the main focus of the project.

Chapter 7

Implementation

7.1 Tensorflow setup

As mentioned earlier, the Python API is used for Tensorflow. However, Tensorflow is also based on native (C++) code for execution speed. By default Tensorflow only supports execution on the CPU but as previously established, deep learning greatly benefits from using the GPU. For this, Tensorflow is using Nvidia's general-purpose GPU (GPGPU) computation library/platform called CUDA. It also uses Nvidia's deep neural network library called cuDNN. Due to the fast evolution speed of both Tensorflow and the Nvidia libraries it is not always clear which library versions are compatible with the provided (compiled) Tensorflow packages. Tensorflow, CUDA and cuDNN have to be set up separately so it is rather likely that there are going to be compatibility issues. Instead of the manual setup, the current setup is done in a simpler way with Anaconda.

Anaconda is a Python distribution for scientific Python applications. It features a virtual environment manager, a package manager and it supports multiple Python versions. Anaconda also provides a fully functional GPU-accelerated Tensorflow package with CUDA 9. Unlike the official Tensorflow package, this one requires no additional setup aside from installing the package through the package manager of Anaconda so this is the preferred setup method.

7.2 Supporting features

The project is mainly centred around deep learning and neural network model construction, however, there are a few necessary supporting features. The main one is audio file loading.

The project only deals with uncompressed, .wav format audio files. Explicitly only 16-bit, 16 kHz data is supported. This is done in order to reduce the amount of data the neural network has to process, without compromising the quality of the audio file too much. File loading is done with SciPy which is then fed into a custom data preprocessor (the data loader). File writing is also handled by SciPy.

For training progress logging Tensorflow's Tensorboard can be utilised: it is a graphical user interface for Tensorflow models, but it is possible to keep track of program-specified variables, for example the output of loss functions, generated images/audio at given time steps or the current training time step, among others.

7.3 Plans for improving on the state-of-the-art

As discussed earlier, identifying the state-of-the-art is at best not a trivial task. Out of the three approaches GANs are by far the newest and they also have not seen widespread application to raw audio/music generation thus exploring that approach is likely to be the most beneficial. The initial model is based on WaveGAN [2]. There are a high amount of improvements that can be attempted, mostly based on similar research done in the image generation domain. Additionally, the rather impressive image generation results demonstrated, for example, in [3] make this a reasonable course to follow.

The plans for improvement are:

- Progressive growing of GANs [3]: a technique from the image generation domain. The idea is to start training a small network on small/low-resolution inputs then as the training goes on add further layers to both the generator and the discriminator that allows higher resolution images to be processed and generated. This technique is also close to another deep learning technique called "transfer learning". In transfer learning a pre-trained neural network is used as the base network, with optional, new layers on top of that network.

For progressive growing the "pre-trained" part would be layers 0 to $x - 1$ if the neural network at a given training step has x layers.

- Adding a variational autoencoder to the generator of the GAN [43]: the point of autoencoders is to learn a compressed representation of data in an unsupervised manner. Variational autoencoders are simply a generative version of regular autoencoders. The idea behind a VAE-GAN is to have two separate networks: the VAE and the GAN where the two networks are connected through one part of their networks: the merged encoder and generator (which serves as the decoder). This approach has seen limited usage in image generation [43][44] but it is not known how applicable it is to audio generation since it is unlikely to have been implemented before. The improvement over a normal GAN is that the encoder part of the generator receives a real input as well, so the decoder (which roughly corresponds to the original generator) receives a latent vector that is not random, but is a compressed, lossy representation of the input. In theory, this would provide further guidance for the generator.

- Mixed precision training [4]: modern GPUs are capable of 16-bit floating point (FP16) calculations significantly faster than FP32 calculation. Mixed precision training takes advantage of this by performing most of the calculations in FP16 but with only marginally less accuracy than what is possible with full FP32 training. Hardware is utilised better this way, which should lead to a speed-up in both training and inference.

- Stereo generation: instead of single channel audio, generate stereo audio.

7.4 Initial implementation

The first implementation is heavily based on v1 of WaveGAN [2] so after a brief overview of the implementation a short discussion about WaveGAN is going to follow, in a manner not covered by the original paper. The later iterations of the program also retain some parts of the base architecture so it is important to understand how the basic program works.

The first major part in a deep learning program is file reading and data preparation. For reading and writing .wav files a scientific package of Python called "SciPy" provides such functionality. In the implementation two utility wrapper functions are used for this purpose.

The data read by the aforementioned function is used in a custom data loader. The input

data is an array of 16-bit PCM wave audio - this means that the values range between -32768 and 32767. Neural networks generally prefer the data to have a mean of 0 with a standard deviation of 1. In other words, the data should fit into the [-1, 1] floating point range. One easy way to enforce this constraint is to divide the input data point by the maximum positive value in the range. Since the positive and negative limits are asymmetric it is not possible to exactly map the values to inclusive [-1, 1] - either the lower range slightly extends past -1 or the upper range is going to be excluding 1. In practice both options are equally viable so the decision is to divide the input by 32767.

Another preprocessing step that has to be done is padding the input data. Each slice of audio sent to the neural network has to be consistent through a training stage and the only way to enforce this is to pad the audio so the amount of samples in the final accumulated training data is divisible by the window length - in other words, $number\_of\_all\_samples \bmod slice\_length = 0$. This can be calculated by taking the modulo of the number of overall samples and the slice length and subtracting this amount from the slice length. This gives us how many zero-valued samples have to be appended to the end.

The program loads all of the audio files from a directory, one by one and appends all of the samples to a list, on which the previously mentioned preprocessing steps are carried out. In order to support minibatch training the list is divided up into slice-length sections, which results in a list of 2D tensors, i.e. a 3D tensor.

The final part is passing these values to Tensorflow, which is done through the Dataset API. With it, it is possible to easily create batched tensors for training out of arbitrary data. In order to retrieve a minibatch for training a Tensorflow iterator has to be used. The initial implementation uses the simplest one, a one-shot iterator. The most significant downside of this iterator is that it embeds the training data into the Tensorflow graph, which leads to checkpoint files being larger than optimal and also slower to create. This problem linearly gets worse with the amount of available training data. The upside of this iterator is that it is easy to use and implement, so it was chosen for the initial implementation. The second implementation uses an initialisable iterator, which avoids the issues with the one-shot iterator but it is more difficult to implement. It is covered in the section of the second implementation.

The model building happens in two main functions, the ”GANGenerator” and the ”GAN-Discriminator”. These are responsible for building up the Tensorflow computation graph for

each of the networks. These graphs are run during each training iteration. One thing to pay attention to is that this is not well-reflected in the Python code itself - it is necessary to keep in mind the underlying computational graph when developing a Tensorflow application. As mentioned previously, both of these are convolutional networks, using strided convolutions for upsampling and downsampling operations. This implementation follows the first WaveGAN model.

The generator network assumes a random (uniformly distributed) latent vector as its input, which it reshapes into the first convolutional layer, which has 16 samples with 1024 channels as its input and 64 samples with 512 channels as its output. The sample amount is quadrupled with strided transposed convolutions, with a stride of 4. The network consists of 5 layers similar to the previously discussed one and outputs a generated slice of audio that is 16384 samples (slightly above 1 second) long. The network uses ReLU after most of the layers, except the last one which uses tanh, in order to map the final output values to the [-1, 1] range.

The discriminator network takes an input that contains as many samples as the output of the generator, passes this input through 5 layers of strided convolutional layers (with a stride of 4 so the amount of samples is always one fourth of the previous layer) then uses a fully-connected layer to output a single prediction value. Unlike the output of the generator this value is not bounded, so the network does not use an activation function after the last fully-connected layer. Aside from this the whole network uses leaky ReLUs, following the guidelines set out in DCGAN [25].

Variable scoping in Tensorflow is also worth mentioning here. Tensorflow creates a default variable scope for layers if an explicit one is not defined. This means that variables in one layer might be shared with a later layer, which is not the desirable outcome in this case. To prevent this, in this model every layer is given its own variable scope. This variable scoping works differently than variable scopes in traditional programming languages since the variable scopes in Tensorflow are tied to the computational graph itself rather than a specific block of code.

This model was also tested with phase shuffle, a technique proposed in the WaveGAN paper. The authors argued that the upsampling convolutions create artifacts similar to checkerboarding artifacts found in images produced by the same convolution type. These artifacts make it easy for the discriminator to detect fake input. The idea behind it is to slightly shuffle the input to the different layers of the discriminator so it could not reject generated samples based on

artifacts. Experiments showed that this technique slightly increased the quality and variance of generated outputs but it made training more prone to collapse, where the generator did not learn to produce anything useful. It is not used in the progressive version of the model due to its instability.

Most of the functionality is contained within train.py. The three main functionalities are training, setting up for inference/generation and audio generation itself. Let us look at them in the order they are used. The inference setup is the first.

The way audio is generated is through running another instance of the program, but with it set to generation mode. For this to be possible it is necessary to save a version of the computational graph to disk. This graph is called a "metagraph" and it does not contain any trainable variable data, just the graph itself. Compared to the graph built for normal training this one only contains the generator and a few extra operations to convert the data from the [-1, 1] floating point range into the range and type .wav files expect.

The most important functionality is the training part. It is conducted by creating a Dataloader object, which supplies the real training data to the discriminator through a Tensorflow iterator. The input to the generator is a uniformly distributed random float in the [-1, 1] range. After this, the previously discussed model building functions are called, in order to build the Tensorflow graphs for both the generator and the discriminator. Note that the discriminator function is called multiple times. This way it is possible to easily pass it the real training data and the fake generated data as well. By default, in this case a discriminator network would be trained on the real data and a separate copy of the network (with its own trainable variables) would be trained on the generated data. This is not how the network should function, so when the function of the discriminator is called its variable scope is always set to the same scope, with variable reuse set to true. This way only a single set of trainable variables is kept (and trained) for the discriminator.

The loss function is the Wasserstein loss with gradient penalty and the optimiser is Adam, with the WaveGAN recommended default hyperparameters.

The final part is the Tensorflow session and the training loop. A Tensorflow session makes it possible to run the previously built computational graph. Depending on the graph and the specified operations within the session, this might be inference/generation, training or even a simple mathematical operation, such as multiplying two tensors together. In this case the

session is used for training.

There are multiple ways to create a session in Tensorflow. The simplest one only handles the session itself, nothing else. In this case variable initialisation, model saving and restoration (checkpointing) and Tensorboard visualisations (in order to monitor the progress of training) all have to be done manually. One of the session types takes care of all this: the monitored training session. It is used by both this and the advanced models. One downside of this session type is that it is less flexible when functionalities are needed that it was not explicitly designed for, for example progressive model growing. Overall, however, the downsides are less significant than the benefits it provides.

Generation is done in a rather simple way, in two different places. First, the output of the generator is sent to Tensorboard at specified time intervals so these can be listened to in the Tensorboard dashboard. The second one is a generator function that writes a specified amount of generated examples to disk. This function uses the inference metagraph described earlier and the newest training checkpoint. The function executes a loop periodically that checks for a new checkpoint so it can be run alongside training as a separate process. It can either be used to generate audio from a fixed latent vector, in which case it is useful for monitoring how training changes the sound of a fixed vector or generating a random latent vector every time, in which case it functions as a normal audio generating functionality. The latter is implemented in code, but changing it to the other one is a small task.

7.5 Initial implementation: discussion

This implementation does not greatly differ from the first version of WaveGAN, if anything, it is a tailored implementation upon which the more advanced models can be built. Because of this the receptive field is identical and the quality of the output is similar as well. The WaveGAN paper did not cover music generation in great depth so let this part cover that topic. The music dataset in this case was an hour-long slice from the MAESTRO piano dataset [45] released by Google. This dataset contains both MIDI and raw, recorded audio versions of the performances - this project uses the recorded versions. Some manual preprocessing was carried out to resample the audio to 16 kHz, make it mono and remove some sections of the recordings that were silent.

The model was tested with both shuffled and unshuffled data slices. In this case shuffling only means that the training batches contain different slices of audio. Training without phase shuffle was noticed to be stable over almost every training run, however, without shuffling the dataset the generated audio samples seemed to be worse both in terms of quality and musicality. Phase shuffle added to unshuffled data always ended up with unstable training, with the samples not being recognisable as any kind of music. A possible explanation for this is that the discriminator learned the batches themselves instead of the individual slices and could easily tell apart slices that were not part of those batches, i.e. the ones generated by the generator. Shuffling the dataset solved this problem and made it possible to use phase shuffle. Generated samples with phase shuffle were observed to be marginally better quality than without. The main drawback of this model is that the generated audio samples are limited to one second. It is a suitable amount of time for some audio generation domains but it is hardly ideal for music. One common metric with generative models is ”receptive field”: the amount of data the model can coherently generate. For audio this would be the length for which a generated sample would stay consistent. GANs aim to produce output that could have come from the training set, so the size/length of the output of a GAN can function as a ”global receptive field”. This is advantageous since for some models it is not trivial to calculate the receptive field, however, since this is not an autoregressive model the length of its output is strictly fixed to its receptive field. At the moment of this project the best way of progression is lengthening this receptive field, which is the purpose of the next model.

7.6 The progressive model

To start, here is the complete network architecture for the full progressive model with feature map smoothing. First the generator, then the discriminator:

Table 7.1: The architecture of the full generator.

Block number	Layer type	Activation	Input length	Output length	Input channels	Output channels
Reshape/projection	Fully connected	LReLU	100	16	1	1024
1	Transposed strided conv	LReLU	16	64	1024	512
2	Transposed strided conv	LReLU	64	256	512	256
3	Transposed strided conv	LReLU	256	1024	256	128
4	Transposed strided conv	LReLU	1024	4096	128	64
5	Transposed strided conv	LReLU	4096	16384	64	48
6	Transposed strided conv	LReLU	16384	65536	48	32
7	Transposed strided conv	LReLU	65536	262144	32	24
8	Transposed strided conv	LReLU	262144	1048576	24	16
Output	Conv	Tanh	1048576	1048576	16	1 (mono)/2 (stereo)

Table 7.2: The architecture of the full discriminator.

Block number	Layer type	Activation	Input length	Output length	Input channels	Output channels
Input	Conv	LReLU	1048576	1048576	1 (mono)/2 (stereo)	16
8	Strided conv	LReLU	1048576	262144	16	24
7	Strided conv	LReLU	262144	65536	24	32
6	Strided conv	LReLU	65536	16384	32	48
5	Strided conv	LReLU	16384	4096	48	64
4	Strided conv	LReLU	4096	1024	64	128
3	Strided conv	LReLU	1024	256	128	256
2	Strided conv	LReLU	256	64	256	512
1	Strided conv	LReLU	64	16	512	1024
Reshape/prediction	Fully connected	-	16	1	1024	1

The kernel size is 25 for every convolutional layer, aside from the output/input convolutional layers, those have a kernel size of 1. The stride is 4 for the transposed strided and normal strided layers.

This model is based on WaveGAN [2] and PGGAN [3], with heavy modifications. The latter is a progressively grown GAN for image generation, capable of generating images up to a resolution of 1024×1024, which corresponds to an audio slice length of about 65 seconds. This model stays stable even for high resolutions through progressive training: the network is divided up into blocks (which contain one or more layers) and more and more of these blocks are stacked on top of each other as training progresses. This means that only the new layers have to be trained from a freshly initialised stage, while the other layers should have values that are close to being optimal. Thus the model only has to significantly adjust a much lower amount of variables compared to a non-progressive model.

The original PGGAN for image generation works by loading full resolution images (which is equivalent to the final target resolution of the model) and these get downsampled to the suitable resolution for the current block. The equivalent operation for audio would be dividing up the training data into 65 second slices and downsampling them from the 16 kHz sampling rate in a way that the final amount of samples would be equivalent to what the block expects as its input. Each block would output a 65 second slice with a non-standard sampling rate, below that of the input data. Due to some inherent differences between audio and images, e.g. the sampling rate of audio is rather well standardised across applications so it is usually desirable to follow those, which this approach could not do for any of the blocks before the final one, a different approach was taken. The original PGGAN model also explicitly uses upsampling and downsampling operations between the blocks which can cause aliasing problems with audio. The alternative approach presented here avoids the need for these operations.

The version of progressive growing used for this project instead takes a variable length slice of audio of a fixed sampling rate. Each newer block extends the length of the training and generated audio slices. For example, 5 blocks output slightly longer than 1 second slices (16384 samples, precisely). Transitioning to 6 blocks quadruples this, to 65536 samples, so just above 4 seconds.

Another difference between PGGAN and the network presented here is the way transition is done when a new layer is introduced. PGGAN gradually fades in the new layer during a fixed-length training stage where the output of the new block (block number x) is mixed with the upsampled/downsampled output of block x-1. The reason for this is not to ”shock” the already trained layers when a new block is introduced. In this context ”shock” means that the network would likely react negatively to new, randomly initialised variables. As previously mentioned the project network aims to avoids explicit upsampling and downsampling operations but it is still desirable to avoid shocking/destabilising the rest of the network. The technique used here for that purpose is called layer freezing, adapted from transfer learning [46].

Transfer learning is a field of deep learning where a pre-trained network is repurposed for a similar-but-different area compared to its original purpose. A network trained to recognise dogs and cats could be applied to recognise wolves and lions. This is done by loading the variables from most of the pre-trained network, but replacing the last few layers with new layers. First, the loaded part of the network is ”frozen”: the variables are set to non-trainable so they do not change during training and the new layers are trained to more optimal values than random

initialisation. As a second round, the loaded layers are ”unfrozen” and the network is trained end-to-end until optimal values are reached. There are parallels between transfer learning and progressive growing, just in the case of the latter instead of the domain of the problem, it is the ”size” of the problem that changes. Originating from that idea, this project adapts layer freezing as the transition technique. Other advantages of this technique are that it is reasonably easy to implement: there is no need to keep track of the progress of the training aside from the training stage; transitional training is faster since a large part of the network is not trained; it is easy to train networks starting from intermediate stages. For quick prototyping it might be desirable to start network training from more than just a single block. It is possible with this network.

These kinds of deep networks take a significant time to train even on cutting-edge hardware, so in order to fit into the time constraints of this project the software was developed with a focus on the ability for reasonably quick prototyping. It does, however, mean that there are some adverse effects on the quality of the output. In a later section it is explained how this was mitigated. Now let us examine in detail how the program works.

The data loader is overhauled from the first implementation, which used a one-shot iterator. One-shot iterators have a few problems but an initialisable iterator can be used to solve these. Initially the initialisable iterator takes a ”placeholder” as input, which signifies that some kind of data has to be supplied to the operation when it is run as part of a session. Afterwards the iterator is used in the same way. A significant difference, however, comes from the fact that this iterator has to be initialised before the computational graph is run. The monitored training session expects every variable to be initialised by the time the session is available for use. One convenient way would be to use a custom Tensorflow scaffold, since monitored training sessions use scaffolds in order to reload or initialise variables for a model. The problem is that a scaffold only runs its initialisers if a new training run commences, i.e. when it creates checkpoints in a new folder but they do not get run when reloading from the same folder but the iterator has to be initialised for every run. The solution is a hook - a special object that can be passed to the monitored training session, The functions of this object get run at specific places during session creation so it is a convenient place to run the initialiser of the iterator, which gets data passed to it from the data loader.

In addition to that, instead of relying on the built-in Python lists, the advanced data loader uses NumPy arrays throughout nearly the whole process. The reason for this is mostly efficiency:

with the former method loading and preprocessing about 15 hours of training data could take above 70 minutes on a machine with 32GB RAM, whereas the latter method accomplishes this task in about a minute.

The next part is the progressive model. Due to the fact that the dimensionality of each already trained layer in the progressive model has to stay the same between stages (in order to support variable reloading) there is a need for an extra layer for both the discriminator and the generator. These layers also appear in [3], in a slightly different form. For the generator its purpose is to convert the output of the current last layer into a 1/2 channel output, for the discriminator it is exactly the opposite - to convert the input from 1/2 channels into as many channels as the current top layer expects. These layers use convolutions with a kernel size of 1 (which is equivalent to 1×1 convolutions for 2D), the purpose of which is similar to a fully-connected layer, just operating on multiple channels. These layers are only kept for the frozen and stable parts of a specific stage. Since the input/output channels and sample counts expected from them change between different block amounts they are not reusable.

The rest of the model building is done in loops. If the early layers are not frozen the permanent layers of both networks are constructed this way. If the early layers are frozen only the frozen layers are constructed in the loop (with their trainability set to false) and the non-frozen ones are constructed separately.

Another important piece is the way the amount of feature maps is calculated. Ideally, the large number of feature maps from the PGGAN network would be optimal, however, its highly adverse effect on training speed prevents that calculation method from being usable for the project. WaveGAN uses a simple way of calculating feature map counts: take a power of two number (for example 1024) and divide that count by 2^{block_id}. This grants good training speeds but it only leaves a low amount of feature maps for later blocks - as low as 4 for the 8th block. In practice, samples with recognisable melodies can be produced with this method and it is used for some of the 16 second examples presented. The other method used is a ”smoothed” version of the WaveGAN method: for blocks up until and including block 4 follow the WaveGAN method. For 5 and above smooth out the count so it is either 25% or 50% less than the next highest power of 2 number. This is done in the following way: take the feature map count of block 4 and use it as a ”base count”. Afterwards calculate an exponent from the current block ID minus the block ID of the block that gives the base count by ceiling dividing the working block ID (which is current block ID - 4 in this case) by 2. The next highest power of two

number can be obtained by dividing the base filter count by $2^{exponent-1}$. The final feature map count is obtained by multiplying the result of the previous computation by 0.75 if the current block ID is odd and by 0.5 if it is even.

Due to the different stages of progressive training the program heavily relies on model saving and reloading. Each stage is saved into a directory of its own, including separate directories for frozen stages. Transitioning to the next non-frozen stage x (aside from stage 1) is done in the following way: the program first checks if checkpoints for a frozen stage x exist - if the answer is yes, it reloads the variables from there. Otherwise, if a non-frozen version of stage x-1 exists the variables are reloaded from there, since even without the transitional stage it is more optimal to start with a partly trained network than a completely new one. If none of these exist the network starts training from stage x, with newly initialised variables. Variable reloading is done similarly for frozen stages.

From an implementation perspective this is done with a Tensorflow scaffold. One issue with the default scaffold is that it tries to restore every trainable variable for a given model from a checkpoint. During progressive growing this can cause problems, since the checkpoint for stage x-1 is not going to contain restorable variables for the new block. Variables for the optimiser can also be problematic: intuitively, the parts of the model that are frozen need no optimiser variables. When moving from a frozen to a "stable" stage, the default scaffold would attempt to restore non-existent variables for the optimiser. The custom scaffold solves this by specifying the correct list of variables to restore for every stage.

Ideally, training is done from block 1. If every stage (both frozen and stable) is given only a day of training it would take 15 days to train that network. In order to find a compromise between quality and training time the networks presented here are initially trained from block 5.

This concludes the main part of the implementation. However, as laid out earlier there are two additional noteworthy contributions to raw audio generating GANs: the implementation of mixed precision training and stereo generation. In addition to those a simple form of audio augmentation was also implemented. Let us look at them in order.

Mixed precision training [4] is a relatively recent technique aimed at making use of the significant half-precision computing power and specialised half-precision cores of cutting-edge GPUs. Normally neural networks use single-precision (32-bit - FP32) floating point numbers

for calculations and for storing variables. This means that the half-precision (16-bit - FP16) cores are not in use, only when calculations are in FP16. Of course, if FP16 is used instead of FP32 accuracy is lost, which might end up hurting the speed of convergence for the network. Mixed precision training aims to reduce the negative effects while using FP16 for most parts of the training run. Precisely, the trainable variables are stored in FP32 but the forward pass of the network is in FP16 - depending on the input data type of the network Tensorflow takes care of using the correct type for the forward pass. In order to avoid losing data for the backward pass the loss is multiplied by a number to make the average value of gradients higher, thus avoiding underflow that could happen because of the lower precision. The number that the loss needs to be multiplied by (the "loss scale") changes from model-to-model. Nvidia recommended 128 in one of their talks [47], however, for the model used in this project 32 was found to be a suitable, stable number both for the generator and discriminator losses.

Stereo generation is done in a rather simple way, thanks to the way how convolutional networks work. With a recurrent neural network there would have been need for an embedding matrix with size equal to the square of the number of possible output sample values, in order to account for all of the possible value pairs in the left and right channels. With a convolutional network the output layer's channel count (of the generator) simply has to be changed from 1 to 2. Implementation-wise, the data loader detects the channel count of the training files and the suitable network is constructed accordingly. For now, only mono and stereo audio is supported but making it compatible with arbitrary channel counts would be a trivial task. The new version of WaveGAN released in early February also explicitly supports multi-channel audio files, however, this was developed alongside the current project so there are only a few similarities.

The form of data augmentation here is not novel, however, it has seemingly never been mentioned in literature, just in implementations. The purpose of data augmentation is to create a larger training set from a limited amount of data. Images, for example, can be rotated, mirrored, etc. so there would be more "unique" images, even if to a human they are basically the same. In the case of audio GANs, a similar action can be done: offset the start of the training data by a few samples (10% of the slice length was chosen for this project) and append the offset data to the end of the "normal" data. This way, when the loaded audio block is divided up into slices, the slices from the augmented data are going to be slightly offset so they are different from the slices created from the normal data. This is useful since this makes it

harder for the discriminator to memorise the specific training slices since there is going to be a lot more unique ones. Of course, it is preferable to have more real training data but it is useful if the network is trained on a short album, since if the augmentation mode is set to maximum then 30 minutes of training audio can be augmented to roughly 300 minutes.

That concludes the advanced implementation. For the sake of thoroughness there have been many experimental models implemented and briefly tested, with changes from smaller scale to larger scale. These are strictly experimental so only a short description of them is presented, along with a short evaluation. The main evaluation part - the evaluation of the main advanced model - is done after this section.

7.7 Experimental models

These models were only given a relatively short amount of training time and mostly only trained to produce one second audio slices. This was necessary to fit into the time and hardware constraints of the project. Some of the models are also significantly larger than the main model of the project and thus training them would take longer. Because of this it is not completely possible to tell how well they would work with longer training times and longer slice lengths. With that disclaimer, let us look at some of the models that differ significantly from the main model variants.

- **Main model architecture with all of the improvements from the PGGAN paper implemented**: aside from progressive growing itself, the PGGAN paper [3] presented 3 other significant techniques: pixel normalisation, concatenating a channel that contains the mean of the standard deviation of the minibatch to the last convolution and equalising learning rates through custom variable initialisation and weight scaling. The implementation of these techniques was based on the PGGAN implementation by Tensorflow Models, however, the techniques combined with the model used for the project could not produce any usable results. The only technique that worked was a slightly modified version of pixel normalisation, renamed ”sample normalisation”. This is used for some of the results presented in the evaluation section.
- **Asymmetric and extended discriminator and generator networks**: this point presents two networks: one where the discriminator has an extra non-strided convolutional

layer in every block and one where both the generator and discriminator have the extra layers. These networks are not much different compared to the main network, they only have more trainable variables. This made them slow to train and rather unstable in the early stages so they were only trained for a brief time period. The loss curve stayed stable so they would likely produce better results with sufficient training time. Because of the size of these networks it is also likely that it would be highly beneficial to train them from stage 1 instead of stage 5.

- **Progressive VAE-GAN**: while VAE-GANs have been used before [43], as far as can be told this is the first ever implementation of a progressive VAE-GAN. This network is a mix of a variational autoencoder and the main network - a progressive Wasserstein GAN. The encoder part of the VAE is responsible for generating a lossy compressed latent vector for the generator from some real input training data. This model was trained on 1 and 4 second audio slices and could generate audio comparable to the main model, however, it did so in fewer iterations and with less noise, especially for the 1 second variant. The training loop is set up in the following way: first the encoder is trained, followed by 5 iterations of discriminator training, followed by 1 iteration of generator training. A few of the generated samples are included with the report. This network could also be explored further as a part of the future work.

- **Above 1 minute audio generation**: this is not exactly a different network, but the full main network. It is capable of generating about 65 seconds of coherent audio, which is state-of-the-art for any kind of audio generating network in terms of purely the receptive field[1], however, the output is rather noisy - melodies can be heard under the noise but they are not easy to discern. Because of this, the main focus of the evaluation chapter is the 16 second model variants. Despite that, a few of the produced audio samples are included with the report. This model would likely benefit from much longer training times and maybe the extra convolutional layers described earlier.

Having looked at the experimental models, let us evaluate the main networks of the project. There are a few variations, but the main covered one is a mono audio generating network with feature map smoothing enabled. It is explicitly mentioned if the network is different, i.e. generates stereo audio or has sample normalisation enabled.

[1] As far as can be told the previous state-of-the-art was 25 seconds in [1], however, not every paper discusses the receptive field. [24], for example, does not mention it at all so it is hard to evaluate. Having listened to some samples from [24], it seems to often get stuck in loops where it only outputs the same 1-2 second long audio section, so it is likely its receptive field does not reach 25 seconds.

Chapter 8

Evaluation, discussion and future work

8.1 General deep learning testing methodology

Non-generative deep learning models can be easily tested on how well they perform on a subset of the training data that is not used for training. This way the accuracy of the model can be tested without having to worry about overfitting. The output of the loss function is also a good indicator.

Specifically for GANs it can also be said that they do not overfit [34] since the generator never observes the real training data. For the models presented in this project the output values of the loss functions did not necessarily correlate to better or worse quality. The only observable metrics were that if the loss jumped to extreme values between training iterations the samples turned out unremarkable. The same was true if the loss stayed roughly constant during thousands of iterations.

For image generation models, the Inception Score [48] or the Fréchet Inception Distance [49] can be used. The point of both of these methods is to take a pre-trained classifier network (which is commonly the Inception-v3 network) or train a new network in a supervised manner. If the generator is good it is going to be able to make the classifier classify the output of the generator correctly.

8.2 Audio testing

As far as can be told, no such methods exist for raw audio generation, however, it can be made to work if only a small, controlled subset of raw audio is tested by creating a spectrogram classifying model. [2] used this methodology for creating a classification network for the SC09 dataset, which consists of humans saying the numbers from 0 to 9. In this case, the classifier decided which number the input was.

Unfortunately, as found by [1] for music generation a network like that does not exist, most likely due to the possible diversity exhibited even by single-instrument music. Their conclusion was that the most important metric is listening to the generated samples. Because of this there is a significant amount of samples attached to this report[1], with a few of the noteworthy ones mentioned and analysed in the appendix. Aside from that, however, there are a few rough ways to estimate how good the output of the model is. One of them is comparing the average absolute loudness of the generated audio and the real audio. This can indicate that the model is capable of reasonably modelling the training data by matching its loudness. Another metric is visually comparing the waveforms of generated and real audio. This is done with the help of a tool called Audacity.

Improvements not strictly related to audio quality are easier to test. The efficiency of mixed precision training is tested by running the same model with FP32 and mixed precision training for 500 iterations and seeing how much time it took the model to reach step 500 step from step 101. The first 100 is not counted in order to discount any possible inconsistencies that might arise from the GPU clocking up during the first iterations. Stereo generation can be evaluated by looking at the waveforms for the 2 channels. If they are mostly similar (with minor differences) it can be seen that the AI learned to generate ”cross-coherent” (i.e. the two channels are consistent) audio. Keep in mind that in this chapter the term ”sample” refers to a whole, generated audio file, not the individual sample points contained within. This is only different in the case of ”sample normalisation”, which refers to the technique described in the previous chapter.

[1]Found either in the ”generated_samples” sub-folder in the source code submission or at this Google Drive address: `https://drive.google.com/open?id=1WIWQnQNQeto5Z_yewTlj3dADoxpsPoPw`

8.3 Discussion and observations

All of the results presented here were trained on a subset of the Google MAESTRO piano dataset [45]. It is an unedited, stereo-recorded dataset with a bit depth of 16 bits and a sampling rate of 44.1 kHz. There was a minimal amount of manual preprocessing carried out: the data was resampled to 16 kHz, most of the longer breaks were removed and for mono generation only the left track was kept.

Most of the training was carried out either on an Nvidia GTX 1080 or on an Nvidia RTX 2080 Ti. The GPU used for training only affects the training speed, not the final quality. The only part where it makes a difference is when evaluating mixed precision training, since the 2080 Ti is newer and is much better optimised for FP16 operations. Let us look at that first.

8.3.1 Mixed precision training

The test model was the stage 5 progressive model, with smoothed feature maps. As stated previously, the evaluation measures how quickly iteration 500 is reached from iteration 101. The time it took was logged by Tensorboard and the Nvidia RTX 2080 Ti was used as the GPU.

With pure FP32 training it took the network 12 minutes and 59 seconds to reach iteration 500. With mixed precision training it took only 12 minutes and 26 seconds. This is about a 4.4% speed-up. It does not come close to reaching the speed-up that Nvidia claims for normal convolutional networks (about 2x) [47] but since mixed precision training should have no effect on convergence speed it still provides a ”free” speed-up. Additionally, the FP16 Tensor cores are a rather new technology and current CUDA versions are supposedly non-optimal with strided convolutional layers on the Tensor cores [50]. Nearly every layer in the current network is strided so once the CUDA backend is improved it is likely to lead to further speed-ups, without the need to change the application itself.

Additionally, it can be confirmed that the Tensor cores are used by using the Nvidia profiler tool called ”nvprof” and looking for calls to kernels with ”884” in their name [47]. This screenshot was taken after profiling the program with nvprof:

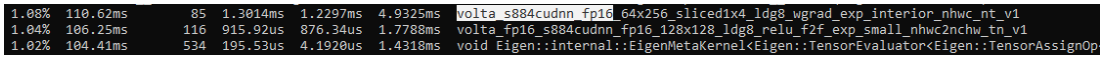

Figure 8.1: The ”884” call signifies that the FP16 Tensor cores are in use.

8.3.2 Stereo generation

The training here used the stereo version of the MAESTRO dataset. The two channels in this dataset are rather similar, since the recordings appear to be of a single piano playing recorded by multiple microphones, to give the music a greater sense of depth during playback. This is also expected of the generated audio - the two channels should be similar with small differences. Aside from listening to them, this can also be checked by inspecting the waveform of a generated audio file:

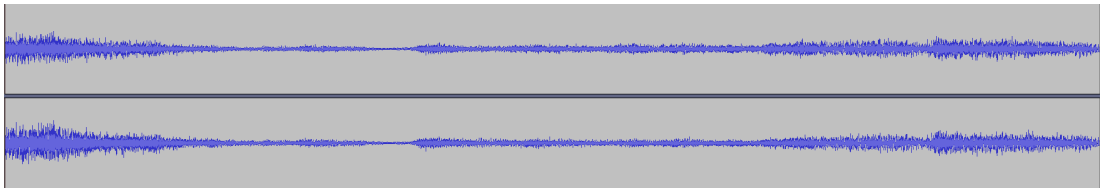

Figure 8.2: Waveform of the generated audio file ”4_sec_60967.wav”. The top waveform is the left channel, the bottom is the right.

As can be seen, the waveforms resemble each other closely. Let us take a look at a zoomed in part so a few differences can be seen, which means that the AI did not just output 2 identical tracks:

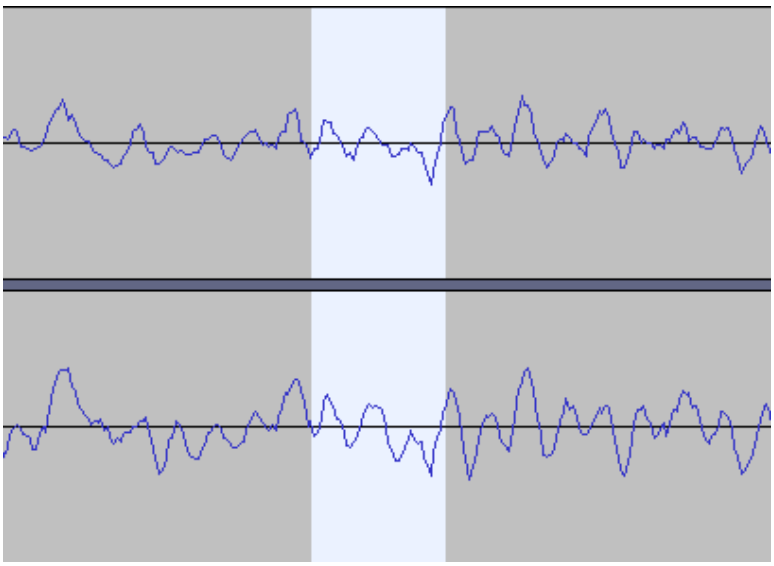

Figure 8.3: Zoomed in view of ”4_sec_60967.wav”.

There are noticeable differences throughout the waveform in the screenshot, however, one of the more visible parts is highlighted. Of course, listening to the audio files is very much encouraged: the stereo audio files are found within ”generated_samples/stereo”.

8.3.3 Long receptive field audio generation and quality

As mentioned previously, there is no good way of quantitatively testing generated audio so this section also functions as a discussion, but through waveform comparisons between real and training data it is attempted to provide another way of evaluation aside from just listening to the samples. It is very much encouraged to listen to them, however, to properly experience the results.

It is possible to split the evaluation roughly into two categories: musicality and quality. A sample might be noisy but if melodies are audible under the noise the musicality of the sample can be considered good. The time a sample stays consistent (the receptive field) should also contribute towards musicality. The evaluation mostly focuses on samples up until 16 seconds and analyses one generated audio file from each stage, in order. Unless explicitly mentioned otherwise, the results here used the model with smoothed feature maps and without sample normalisation. There were no manual post-processing steps carried out on the generated samples. The generated waveform is the top one, the one taken from real data is the bottom one. There are only two tracks, since these are mono files.

The first one is a 1 second slice:

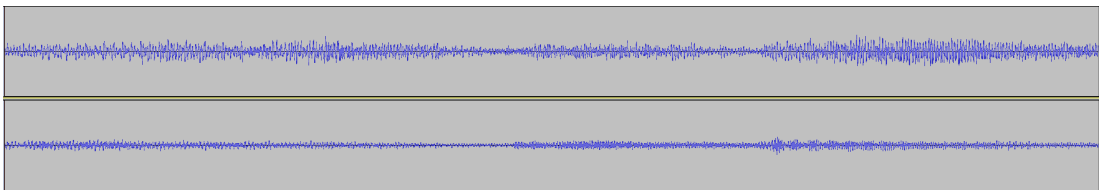

Figure 8.4: Waveform of the file ”stage_5_30910.wav” compared to a slice from the training data. The generated data is slightly louder but the general shape is similar to the real data.

A short melody can be heard in the generated sample, with a minimal amount of static/noise. This is about on par with expectations, since this also matches the generative field of the WaveGAN v1 model. In the zoomed in version of the same two waveforms, it can be seen that the generated data is not as smooth as the real data. This is likely the cause of the noise:

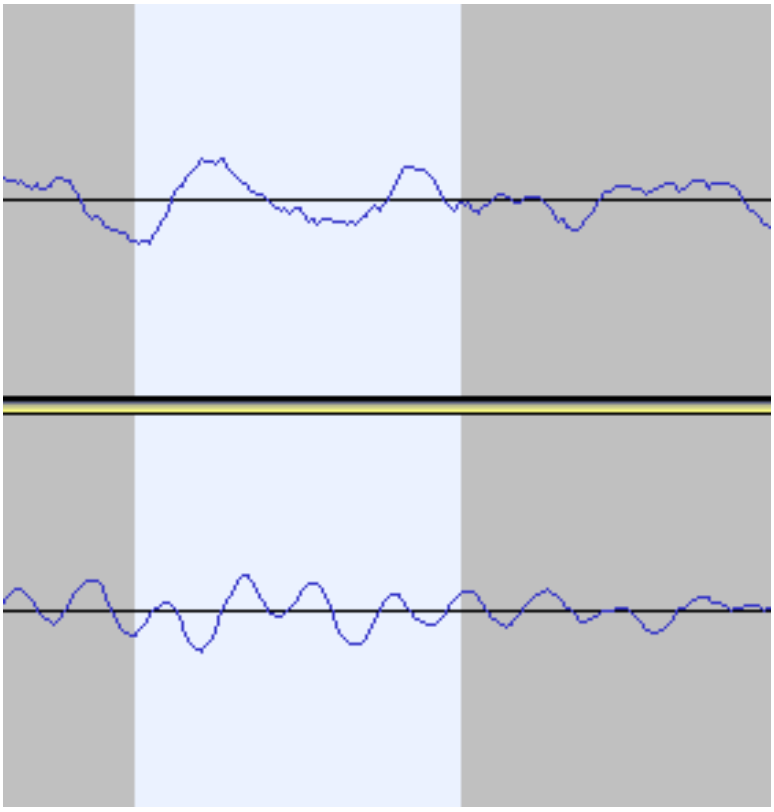

Figure 8.5: Zoomed in view of "stage_5_30910.wav". It can be seen that the real waveform, while not perfectly smooth, is somewhat smoother than the generated waveform. This phenomenon can be observed with other generated waveforms as well.

The second one is a 4 second slice, which is equivalent to the maximum output length of WaveGAN v2. It has to be mentioned here that as the model gets bigger the batch size of the model has to be reduced to stay within GPU RAM (VRAM) constraints. This means that the networks are trained less per iteration. For example, for stage 5 (1 second) the batch size is 64, for stage 6 (4 seconds) it was only 24 for these results (it was later raised to 32 since it proved stable). This means that 1 iteration for stage 5 is roughly equivalent to 3 iterations of stage 6.

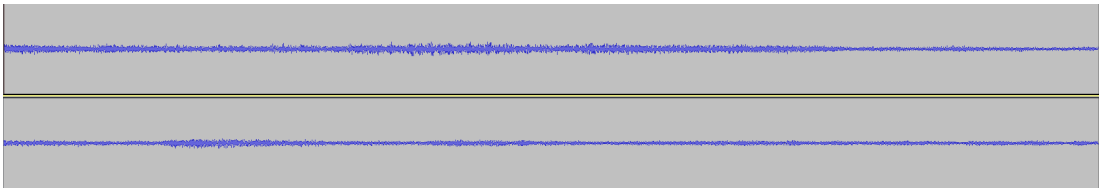

Figure 8.6: Waveform of the file "stage_6_42050.wav". As previously, the two waveforms are similar.

Musicality is about on par with the previous stage, but with 4 seconds of consistency. A melody is clearly audible but the levels of noise are slightly higher.

The 16 second slice is third. The batch size was 8 for this model. This is state-of-the-art for GANs in terms of receptive field (ahead of the 4 seconds produced by v2 of [2]) and as far as can be told only second to the 25 seconds of [1] overall.

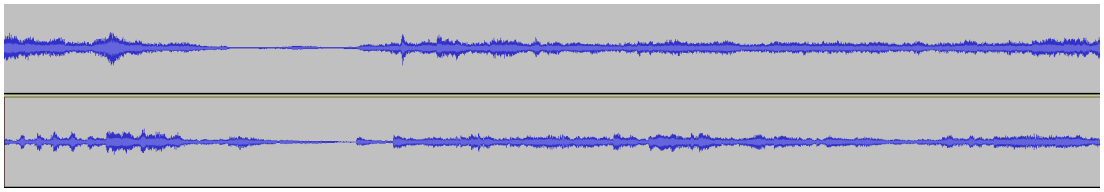

Figure 8.7: Waveform of the file "stage_7_77368.wav". There are some differences between the two in this case but both feature quieter and louder sections as well as small "peaks" which correspond to more intense piano key presses.

Musicality is about on par with the previous two stages but aside from noise some artifacts are also exhibited. They are not serious enough to make the melodies inaudible, however. This is also a reasonable place to take a look at effectiveness of progressive growing. The directory for the demonstration samples contains a subfolder called "progressive_demonstration". Both models were trained on stage 7 (16 seconds) for about 11000 iterations but one of the models was progressively trained from stage 5, the other started training from stage 7. The sample generated by the former model exhibits full melodic consistency with some noise while the non-progressively trained one failed to output any coherent melodies.

The average loudness of the generated samples always roughly matched the average loudness of the training data. Here is a comparison, visualised by Tensorboard:

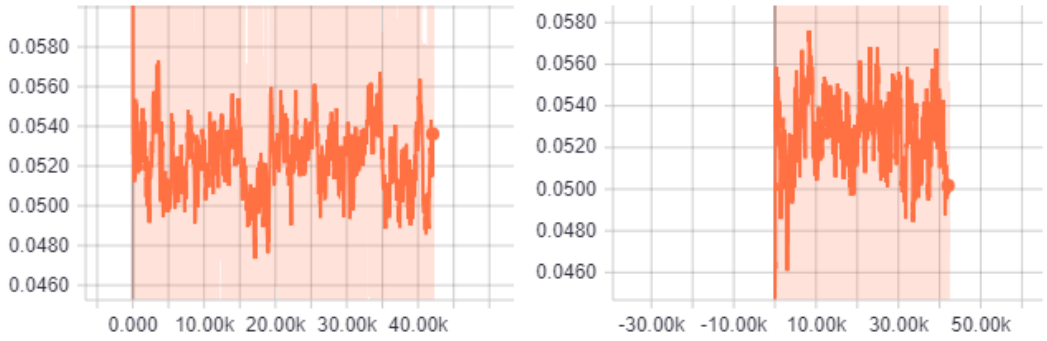

Figure 8.8: This figure shows the average deviation from 0 per sample point on a 0-1 float scale, i.e. how far away from 0 a sample point is on average. This can be interpreted as average loudness. The left graph shows the average loudness for the generated data over about 40000 training iterations for stage 6 and the right graph shows it for the training data. It can be seen that they are approximately equal.

With sample normalisation enabled the quality of the samples was about on par with it not being enabled. An oddity during training was observed, however: if sample normalisation is not enabled, the generated samples start off very quiet and the network learns to generate samples that are as loud as the training data. With sample normalisation it proceeds in the opposite way: the samples start out loud and get quieter until they reach the level of the real data.

Feature map smoothing did not seem to make much of a difference in early stages (since even without it the number of feature maps is relatively high), however, in later stages it seemed to increase both musicality and quality, even though the training time was predictably longer.

Next, let us briefly look at two of the more experimental models. The one that generates audio with a 65 second, state-of-the-art receptive field and the progressive VAE-GAN.

The 65 second model was trained with a batch size of 2, for about 160000 iterations. This would correspond to only about 5000 updates with a batch size of 64 (which is the size of stage 5). Despite this, the model managed to generate samples with recognisable melodies, even though the amount of noise made it hard to easily discern those melodies. The generated samples also have quieter passages, which does reflect the training data, since it had some breaks and sections with only quiet piano notes. A few samples are included with the submission.

The VAE-GAN exhibited results similar to the base model but it required fewer iterations for discernible melodies to appear and the generated samples are somewhat less noisy. To reiterate, this is a novel experimental model and thorough hyperparameter tuning is likely to help it significantly. Nevertheless, the generated audio samples seem promising.

To conclude, the samples included with the submission and the samples presented here were trained on piano data since such open-licence data is easy to find but the model is by no means limited to piano. The 1 second stereo model was also tested on rock/heavy metal and the model coherently captured the sound of all instruments - guitars, bass, drums and vocals. In the next section, let us look at the shortcomings of the project and the future work to mitigate them.

8.4 Shortcomings and future work

Audio quality improvements: The biggest shortcoming of the project is the quality of audio generated with extreme (16-65 seconds) receptive fields. The extended models proposed earlier seem to be a reasonable mitigation to this problem, given more training time and more hardware resources. For these models an extended hyperparameter search would also likely be beneficial for optimal training. It might also be beneficial to experiment with an added "denoising autoencoder" [51]. It is an autoencoder that is trained to clean up noise from the input data but otherwise does not change its input. Given the biggest problem with the generated audio is that they are noisy, this kind of autoencoder might be able to fix that problem.

Continuous generation: one significant issue with GAN-based models is that their output is limited in size/length. Once a receptive field with a significant size is achieved while also retaining good quality, research into "continuous audio GANs" would constitute a large step in audio generation. Due to the generator of VAE-GANs being aware of the real data through the encoder they could be a reasonable research area for this goal too.

Automatic evaluation for music generation: as mentioned earlier, music generation has no metric similar to the Inception Distance, which makes it harder to evaluate models that focus on music generation. A research project into this area could determine if there is a way to evaluate these models similarly to image generation models.

Further research on the Progressive VAE-GAN: this novel model proposed earlier was a late-stage experimental addition and thus it only received relatively minor attention. Despite the promising early samples this model has additional hyperparameters that have to be tuned so finding those optimal parameters would likely take a significant amount of research effort.

Long-term plans: even though this model demonstrated results with state-of-the-art receptive field (16 seconds and 65 seconds, the latter in a limited form) this is still relatively short, even for a single song. Deep learning is a fast moving field where it is not uncommon to have several significant breakthroughs in a year. Even though music generation is rather neglected this project showed that ideas from other domains work for music generation as well, so it is important to keep this domain up-to-date and experiment with new ideas, since ideas

that work here might be applicable to other fields too.

AI is a powerful tool but because of its power and the fact that most AI systems are hard to understand it is also important to be careful with them. The next chapter examines this matter, as well as how the project minimised the possible legal issues arising from the training data.

Chapter 9

Legal, social, ethical and professional issues

9.1 Audio training data and fair use

Music is frequently copyrighted, the point of which is to make it illegal to share among listeners for free since this cuts off the music label and the artist from revenue they should earn by selling copies of the music. Using copyrighted material for generative modelling is a grey area - research and study is covered under Sections 29 [52] and 30 [53] of the Copyright, Designs and Patents Act 1988 so it is allowed by law, but it is not known what would happen if, in theory, in addition to other generated samples the model could recreate the training data either in part or whole. Pure GANs, at least, do not overfit [34] but their goal is to fit the training data as close as possible, so an optimal model might recreate parts of it regardless. In order to avoid any possible problems the project decided to use a piano dataset [45] made available by Google under a Creative Commons licence.

9.2 Human and AI musicians

Aside from just the legal side it is also important to consider the matter from an ethical standpoint. Music generating AI is not sufficiently advanced to replace human musicians but it might become in the future, towards which this project also contributed. Due to it not being

a ”limited” industry, unlike, for example, the jobs that were replaced by mechanisation during the industrial revolution it is not likely AI would have the same effect on musicians. It is, however, a valid concern that record labels could release albums solely generated by AI so they could cut down on costs, since the AI does not require any payment. The impact of this is also mitigated by the emergence of platforms artists can self-release albums on, for example Bandcamp.

In the future, this issue, along with the effects of automation and the possible emergence of general-purpose AI, i.e. an AI that can completely mimic a human or other intelligent agent, unlike the specialised AI nowadays (image recognition, music generation, etc.), this issue might have to be revisited from another angle, taking into account factors such as if the AI is conscious. One solution to this would be granting AI human rights as well.

9.3 The dangers of AI research

Since the learning and decision making process going on in machine learning is not easy to understand it is necessary to exercise caution when working with systems that lives depend on. For example, a crash avoidance system used in cars could lead to serious injuries or even loss of life if the decision of the machine is faulty. This is a problem even without getting into the ethical side, i.e. whom should the machine save if it can either save the driver while hitting a pedestrian or swerve in the last second and injure the driver? This is exponentially more problematic in military applications. As a slightly exaggerated example, if an AI is tasked with maintaining world peace, it might try to achieve this goal by wiping out humanity, since if there is no humanity there is unlikely to be a war again. This is meant to illustrate that AI can bring significant changes in human lives, but if said lives also depend on the AI it is important to be careful and carry out as many controlled experiments as possible.

9.4 Professional issues

The project was carried out adhering to the relevant parts of the Code of Conduct [54] by the British Computer Society. Code or ideas are attributed whenever they are adapted from another source and the training dataset uses a Creative Commons licence. In order to share knowledge gained during the course of this project, after its conclusion, the source code is

planned to be open-sourced.

The project used GitHub for version control and for project organisation a limited form of Agile was used. Code reviews for an individual project are not practically feasible but feature branches and pull requests were utilised for cataloguing progress.

Chapter 10

Conclusions

Generating raw audio music is a challenging task even for advanced deep learning systems - there are many different approaches but all of them have non-negligible downsides. These approaches were investigated and the most viable way was decided to be a relatively new network type: generative adversarial networks. The project described general deep learning concepts and looked at this network type in depth for audio generation and applied ideas from different deep learning fields in novel ways to reach state-of-the-art results in terms of the receptive field of the generated samples. This was done by combining the state-of-the-art audio generating GAN and progressive growing used in image generation, with significant modifications. This approach is not without its downsides either, mostly manifesting in long training times and somewhat noisy samples but with good musicality. The project also introduced/demonstrated different types of improvements for audio generation: mixed precision training, stereo generation and a form of data augmentation. Other, experimental network variants were developed and briefly evaluated as well, including a novel progressive variational autoencoder - generative adversarial network hybrid, which showed promising early results and would be a good candidate for future research endeavours.

References

[1] Sander Dieleman, Aäron van den Oord, and Karen Simonyan. The challenge of realistic music generation: modelling raw audio at scale. *arXiv preprint arXiv:1806.10474*, 2018.

[2] Chris Donahue, Julian McAuley, and Miller Puckette. Adversarial audio synthesis, 2018.

[3] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.

[4] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2017.

[5] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[6] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[7] Marvin Minsky and Seymour Papert. *Perceptrons: an introduction to computational geometry*. MIT Press, 1969.

[8] Paul Werbos. Beyond regression:" new tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974.

[9] Paul J Werbos. Applications of advances in nonlinear sensitivity analysis. In *System modeling and optimization*, pages 762–770. Springer, 1982.

[10] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[11] Jürgen Schmidhuber. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242, 1992.

[12] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[13] Abdel-rahman Mohamed, George E Dahl, and Geoffrey Hinton. Acoustic modeling using deep belief networks. 2010.

[14] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880. ACM, 2009.

[15] Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Mahowald, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947, 2000.

[16] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[17] Chakravarty R Alla Chaitanya, Anton S Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (TOG)*, 36(4):98, 2017.

[18] Allen Huang and Raymond Wu. Deep learning for music. *arXiv preprint arXiv:1606.04930*, 2016.

[19] The method behind the music. Physics of sound. `https://method-behind-the-music.com/mechanics/physics/`. Accessed: 2018-12-12.

[20] Harry Ferdinand Olson. *Music, physics and engineering*, volume 1769. Courier Corporation, 1967.

[21] Stas Bekman. Why 44.1khz? why not 48khz? `https://stason.org/TULARC/pc/cd-recordable/2-35-Why-44-1KHz-Why-not-48KHz.html`. Accessed: 2018-12-12.

[22] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio, 2016.

[23] Aaron van den Oord, Yazhe Li, Igor Babuschkin, Karen Simonyan, Oriol Vinyals, Koray Kavukcuoglu, George van den Driessche, Edward Lockhart, Luis C. Cobo, Florian Stimberg, Norman Casagrande, Dominik Grewe, Seb Noury, Sander Dieleman, Erich Elsen, Nal Kalchbrenner, Heiga Zen, Alex Graves, Helen King, Tom Walters, Dan Belov, and Demis Hassabis. Parallel wavenet: Fast high-fidelity speech synthesis, 2017.

[24] Soroush Mehri, Kundan Kumar, Ishaan Gulrajani, Rithesh Kumar, Shubham Jain, Jose Sotelo, Aaron Courville, and Yoshua Bengio. Samplernn: An unconditional end-to-end neural audio generation model. *arXiv preprint arXiv:1612.07837*, 2016.

[25] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[26] Jesse Engel, Kumar Krishna Agrawal, Shuo Chen, Ishaan Gulrajani, Chris Donahue, and Adam Roberts. Gansynth: Adversarial neural audio synthesis, 2019.

[27] Intel. Intel® core™ i9-9900k processor. `https://ark.intel.com/products/186605/Intel-Core-i9-9900K-Processor-16M-Cache-up-to-5-00-GHz-`. Accessed: 2018-12-12.

[28] NVIDIA. Nvidia turing gpu architecture. `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`. Accessed: 2018-12-12.

[29] Techpowerup. Nvidia gtx 1080 ti specifications. `https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877`. Accessed: 2018-12-12.

[30] Techpowerup. Nvidia rtx 2080 ti specifications. `https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-ti.c3305`. Accessed: 2018-12-12.

[31] Techpowerup. Amd radeon rx vega 64 specifications. `https://www.techpowerup.com/gpu-specs/radeon-rx-vega-64.c2871`. Accessed: 2018-12-12.

[32] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. *arXiv preprint arXiv:1803.04014*, 2018.

[33] Koray Kavukcuoglu. Deepmind moves to tensorflow. `https://ai.googleblog.com/2016/04/deepmind-moves-to-tensorflow.html`. Accessed: 2019-03-31.

[34] Ryan Webster, Julien Rabin, Loic Simon, and Frederic Jurie. Detecting overfitting of deep generative networks via latent recovery, 2019.

[35] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, 2013.

[36] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[37] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans, 2017.

[38] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[39] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[40] Nvidia. Dlss: What does it mean for game developers? `https://news.developer.nvidia.com/dlss-what-does-it-mean-for-game-developers/`. Accessed: 2019-03-31.

[41] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[42] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive gan github repository. `https://github.com/tkarras/progressive_growing_of_gans`. Accessed: 2019-03-31.

[43] Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, Hugo Larochelle, and Ole Winther. Autoencoding beyond pixels using a learned similarity metric. *arXiv preprint arXiv:1512.09300*, 2015.

[44] Ming-Yu Liu, Thomas Breuel, and Jan Kautz. Unsupervised image-to-image translation networks. In *Advances in Neural Information Processing Systems*, pages 700–708, 2017.

[45] Curtis Hawthorne, Andriy Stasyuk, Adam Roberts, Ian Simon, Cheng-Zhi Anna Huang, Sander Dieleman, Erich Elsen, Jesse Engel, and Douglas Eck. Enabling factorized piano music modeling and generation with the maestro dataset, 2018.

[46] Julius Kunze, Louis Kirsch, Ilia Kurenkov, Andreas Krug, Jens Johannsmeier, and Sebastian Stober. Transfer learning for speech recognition on a budget. *Proceedings of the 2nd Workshop on Representation Learning for NLP*, 2017.

[47] Michael Carilli, Christian Sarofeen, Michael Ruberry, and Ben Barsdell. Training neural networks with mixed precision. `http://on-demand.gputechconf.com/gtc-taiwan/2018/pdf/5-1_Internal%20Speaker_Michael%20Carilli_PDF%20For%20Sharing.pdf`. Accessed: 2018-04-02.

[48] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans, 2016.

[49] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium, 2017.

[50] Github Issue. Mixed precision training is no faster than using float32 for deepspeech2. `https://github.com/NVIDIA/OpenSeq2Seq/issues/270`. Accessed: 2019-04-04.

[51] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[52] UK Government. Copyright, designs and patents act 1988, section 29. `https://www.legislation.gov.uk/ukpga/1988/48/section/29`. Accessed: 2019-04-01.

[53] UK Government. Copyright, designs and patents act 1988, section 30. `https://www.legislation.gov.uk/ukpga/1988/48/section/30`. Accessed: 2019-04-01.

[54] British Computer Society. Bcs code of conduct. `https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/`. Accessed: 2019-04-06.

Appendix A

A few thoughts about some of the more interesting generated files

A.1 Feature map smoothing enabled but no sample normalisation

As a foreword, these are hand-curated samples, since it is not guaranteed that the neural network is trained enough to reconstruct something meaningful from all of the possible random latent vectors. Because the training data also contained some breaks a second of silence might also be a possible sample that meaningfully represents a (small) part of the training data. To balance this out, some worse quality samples are also included.

1 second samples:

There are three samples presented from iteration 30910. All of them are rather different, with 3 distinct melodies. 1 and 2 are somewhat louder whereas number 3 resembles the quieter moments from the training data.

4 second samples:

There are 5 samples presented, from 3 different stages, let us look at one from each:

- stage_6_9771.wav: a rather dissonant sample but with a clearly audible melody. There is

some static background noise but not an excessive amount.

- stage_6_35224.wav: the interesting section is a trill/mordent just after 0.985s in. It shows that the model has no trouble learning more advanced techniques, since they are just another part of the final waveform. Unfortunately there are a few artifacts in this piece, for example 0.515s onwards.
- stage_6_42050.wav: a quieter piece, with a minimal amount of static.

16 second samples. From this stage the breaks in the training data also become more significant, since longer slices are more likely to have sections that contain a (short) break. There are 6 samples presented with the submission, let us analyse a few here:

- stage_7_20727.wav: a rather fast, loud and chaotic sample with lots of notes in the beginning, leading to a more melodic ending.
- stage_7_40960_2.wav: this is the complete opposite of the previous one - slow and quiet, with the melody starting around 5 seconds in.
- stage_7_42025.wav: consistent melody throughout, with a prominent section with repeated but lowering notes in the beginning. A similar motif returns between 10 to 12 seconds in, but more in the background.
- stage_7_77368.wav: the most impressive sample: a 16 second consistent melody even though there are a few artifacts and some noise throughout.

65 second samples. Only a handful of them are presented, a few from relatively early in the training and one from much later. As mentioned in the main report, keep in mind that due to the low batch size even 160000 iterations would only be the equivalent of about 5000 updates for stage 5 (which has a batch size of 64). The early samples are really noisy and many of the samples (both early and late) also unsurprisingly have quieter sections. Let us analyse the best one:

- stage_8_166170.wav: the sample stays rather consistent throughout its running time, with a short break around 20 seconds in. It is rather noisy but it is reasonable to assume that this is the first ”intelligible” example of generated audio that has a receptive field of about 65 seconds, which is about 16 times that of WaveGAN v2 (4 seconds) and about 2.5 times that of the WaveNet Autoencoder (25 seconds).

A.2 Miscellaneous samples

Here samples generated by other configurations are briefly analysed. It focuses more on noteworthy samples as opposed to the previous section, which discussed both good and bad. The samples included with the project, of course, have both.

Sample normalisation enabled but no feature map smoothing:

- 16_sec_2478_2.wav: a very good early result, with repeated notes and an intro that gently fades into the main melody.
- 16_sec_118533.wav: after a significant amount of training the generator produced its own take on a motif from the training data. It wrote the motif a new intro and in the original one the pianist stops for a brief moment and his breathing can be heard, which is replaced by a few notes (around 9.5 seconds in). This is a fascinating sample, especially because it cannot really be said that the model overfit - this motif appears right in the beginning (without the intro) of the first file loaded by the program, so the training slice would have little to do with the generated slice on a sample-point-by-sample-point basis. The program simply learned a melody that the discriminator reacted well to and that melody seemed to be so optimal that it literally could have come from the training set, because in a way that is exactly where it came from. The model that produced this was only trained on a 3h-3.5h subset of the MAESTRO dataset and as fascinating as this sample is, in order to avoid this and encourage sample diversity the training set was expanded to about 15 hours for other models after this one.

Stereo samples

- 4_sec_60967.wav: it can be heard that this sample is more ”spacious” compared to mono samples, since there are two slightly different tracks, one for the left channel, one for the right.

VAE-GAN samples:

- 1_sec_5613.wav: an early sample that has very little noise nonetheless.

- 4_sec_33785.wav: 4 second samples are a bit more noisy but they still retain a large amount of coherence and musicality.

Appendix B

User guide

B.1 Setup guide

The program was tested on Window 7/10 and Ubuntu 18.04. The easiest way to set up the environment the program needs is by using Anaconda: `https://www.anaconda.com/`

Depending on the hardware of the computer (i.e. has an Nvidia GPU that was released approximately in the last 5 years or not) it has to be set up differently. CPU-only operation is slow but easy to set up. Simply install Anaconda, make it sure it is working by using the command:

```
1 conda env list
```

It should list the available Anaconda environments. If it is working, create a new Anaconda environment with v1.12.0 of Tensorflow and Python 3.6:

```
1 conda create -n tf-cpu python=3.6 tensorflow=1.12.0
```

Once it is installed, activate the environment:

```
1 conda activate tf-cpu
```

...and done. It is a bit more complicated if a GPU is present. Make it sure that the installed Nvidia driver version is at least v410, on Ubuntu this can be checked by inspecting the output of *nvidia-smi*. If that is done, the rest of the procedure is similar. After Anaconda is installed

and working, install and activate the GPU variant of Tensorflow:

```
conda create -n tf-gpu python=3.6 tensorflow-gpu=1.12.0
conda activate tf-gpu
```

That is it for the setup. In terms of system requirements, for training it is advised to have the strongest hardware possible, especially on the GPU side, with at least 32GB main system RAM for larger datasets. For generation (if a trained model is present) more modest hardware is enough. In terms of OS compatibility, Windows 7/10 or Ubuntu 18.04 is recommended. It might work on Mac OS X but it is not guaranteed.

After this is done, make it sure a ”checkpoints” and a ”data” folder is present in the folder where the .py files are.

B.2 Instructions

For training, the program explicitly expects that all files are mono or stereo (only one type should be present in the training set at the same time), 16-bit, 16 kHz audio files in .wav format. If your training data is different, edit them with an audio editor, for example Audacity: `https://www.audacityteam.org/` Put these files in the ”data” folder. The program that shall be run for normal usage is ”train_progressive.py”. There are a few command line options:

- –preview: instead of training, generate 5 audio slices from the latest checkpoint. It writes a .wav file into the checkpoint folder of the chosen training stage. It also waits for new checkpoints continuously, so exit it with CTRL+C if only one file is desired to be generated.
- –num_blocks followed by a number between 1 and 8, inclusive: Defines how many convolutional blocks the model should have. It defaults to 5. Also known as ”stage number”.
- –use_mixed_precision_training: If specified, it uses mixed precision training, which is faster on cutting-edge GPUs (Nvidia Volta or Turing) but might be slower on old GPUs.
- –augmentation_level followed by a number between 1 and 9, inclusive: It switches on different amounts of data augmentation. Only recommended if the training set is small (below 1 hour).

- –use_sample_norm: switches on sample normalisation. Useful for research purposes, otherwise not recommended.
- –freeze_early_layers: freezes the early layers of the model so only the new layers are trained. Useful for progressive growing. The ideal training workflow is if training is started at stage 5: train stage 5 - train FROZEN stage 6 - train stage 6 and so on.
- –batch_size: the amount of slices the model trains on at the same time. Lower it if the program crashes with an out-of-memory error, increase it if the GPU is under-utilised, otherwise defaults to the following values. The first number represents the number of blocks (stage number), the second one is the batch size for that stage:
  - 1: 128
  - 2: 112
  - 3: 96
  - 4: 80,
  - 5: 64,
  - 6: 32,
  - 7: 8,
  - 8: 4

Monitoring the training can be done through Tensorboard. Launch the training script, open a new terminal/command line window, activate the Tensorflow environment, navigate to the "checkpoints" directory then execute:

```
tensorboard --logdir <directory for the checkpoints of the current training
    ↪ stage>
```

As said previously, "train_progressive.py" runs the stable model that has full, completely stable functionality for both mono and stereo generation, whereas "train.py" is stable but it is just a modified implementation of WaveGAN, so its limitations apply. With "train_progressive_vaegan.py", it is advised to use Tensorboard for generating samples. The experimental models should be stable as well but they have not been run as much as "train_progressive.py" so do exercise caution.