Audio Sample Selection

with

Generative Adversarial Networks

A dissertation submitted in partial fulfilment of the requirements for the MSc in Intelligent Technologies

by James Hill

Department of Computer Science and Information Systems

Birkbeck College, University of London

September 2018

This report is substantially the result of my own work except where explicitly indicated in the text. I give my permission for it to be submitted to the TURNITIN Plagiarism Detection Service. I have read and understood the sections on plagiarism in the Programme Handbook and the College website.

Abstract

Generative Adversarial Networks (GAN)s have displayed impressive capabilities in the process of generating images and research has turned to applying this framework to digital audio generation. WaveGAN is a convolutional GANs capable of learning from 1-second long audio samples of words and then generating similar samples. The following document outlines an attempt to introduce a conditioning element into a GAN with a WaveGAN-like architecture. There are difficulties in adjusting the size of the WaveGAN architecture and the values of useful hyperparameters vary widely depending on the number of convolution layers. The introduction of a conditional element also extends the learning time and reduces variation in generated samples. Further experimentation with different WaveGAN architectures and hyperparameter values is likely necessary before a stable model with a conditioning element may be developed.

Contents

List of Figures

List of Tables

Acronyms

ACGAN	Auxilliary Classifier Generative Adversarial Network
CIFAR-10	Candadian Institute for Advanced Research-10
D	Discriminator
DCGAN	Deep Convolutional Generative Adversarial Network
FVBN	Fully-Visible Bayes Network
G	Generator
GAN	Generative Adversarial Networks
GPU	Graphical Processing Unit
GSN	Generative Stochastic Network
MNIST	Modified National Institute of Standards and Technology
ReLU	rectified linear unit
SC09	Speech Commands Zero through Nine
SEGAN	Speech Enhancement Generative Adversarial Network
VAE	Variational Autoencoder
WGAN	WaveGAN

1 Introduction

Generative Adversarial Networks (GANs) are a powerful form of generative neural network framework (Goodfellow et al., 2014). Variations of this framework have practical applications such as image denoising, inpainting, super resolution, and even the creation of convincing new images; in the United States concerns have even been raised about the impact of such technology (popularly known as 'deepfakes') on national security (Hawkins, 2018).

1.1 The Problem Area

Much initial research on generative neural network frameworks focused on image datasets such as the Modified National Institute of Standards and Technology (MNIST) collection of photographs of written numerals, the Candadian Institute for Advanced Research-10 (CIFAR-10) collection of 60 thousand small colour images, or the ImageNet database of millions of labelled images.

More recently research has started to explore audio datasets and the applications resulting from successful generation of audio; WaveNet by DeepMind is a notable application of a deep neural network models to raw audio (van den Oord et al., 2016).

There are interesting practical applications that could result from realistic generation of audio: natural sounding text-to-speech will enable conversational interaction with computers and new possibilities for effective translation tools; artistic use of generated audio may also result in new forms of synthesizer sounds, new forms of 'sampling', and more realistic imitations of true instruments.

WaveGAN (WGAN) was the first attempt at applying GANs to the synthesis of raw audio waveforms and was shown through a number of experiments to be capable of synthesising audio from a variety of domains, including human speech, bird sounds and also musical instruments (Donahue et al., 2018). The authors reported success in generating audio that human judges preferred over other methods of generating audio.

Many practical applications of audio generation would require control of the output: a text-to-speech application would require that the word to be generated is communicated to the GAN; a musical synthesizer of a real instrument such as a piano would require that the note generated corresponded to a key played on a keyboard. The control of the output of a GAN is called 'conditioning' and a number of methods have been demonstrated that enable this with image generating GANs.

1.2 Aims and Objectives

This project is an attempt at designing and training a simple conditional GAN capable of generating audio samples controlled by an input. The framework of this audio-generating conditional GAN will build upon a modified application of the WGAN framework. As there are a number of different methods of including a conditional element to a GAN some experimentation will be required to determine whether any method is capable of producing successful results.

The original WGAN experiment included training on large audio datasets for long periods with a state-of-the-art and expensive Graphical Processing Unit (GPU). Given the limitations of a student budget, modifications will be made to ensure that the project is feasible: firstly, the size

of the audio samples in the training dataset will be reduced by a process known as downsampling; secondly, the total number of samples in the dataset will be reduced; and thirdly, the variety of the samples in the dataset will be reduced.

As these limitations require adjustments to the original WGAN model, the experiment will include the training of a modified baseline WGAN to allow for better comparisons with the conditional model. The modified baseline WGAN model also allows comparison to the original experiment and of the differences between the models when trained on datasets of different sample sizes.

The project has been divided into a series of stages that will allow for a methodical approach to the problem; the stages are listed below.

Table 1: Table of Objectives

Stage	Description
1	Identify methods for managing the computational cost of the project
2	Ensure the chosen machine learning framework can reproduce the architecture
3	Select an appropriate dataset for training
4	Confirm the final design of the WaveGAN architecture
5	Implement and train the WaveGAN model
6	Confirm the design of the conditioned WaveGAN architecture
7	Implement and train the conditioned WaveGAN model
8	Presentation and discussion of results

1.3 Structure of the Report

This report begins with a review of the theory of GANs with a focus on their application to audio. Following is an outline of the different models created for use in this project with an explanation of their architecture and components. There is then an examination of the dataset selected for training the models with an explanation of modifications made so that the experiment would be feasible. Finally there is an outline of the experiments and results followed by a conclusion that reflects on the project.

2 Theory

There has been much research in the last decade on generative neural network frameworks and a variety of architectures have been designed such as the Fully-Visible Bayes Network (FVBN), the Boltzmann machine, the Generative Stochastic Network (GSN), and the Variational Autoencoder (VAE). GANs were first invented by Ian Goodfellow in 2014 as an alternative to these frameworks and have since become the focus of a large amount of research.

This popularity results from advantages GANs have over other frameworks: firstly, they generate samples in parallel with reduced computation time; secondly, the design of the generator function has few restrictions; thirdly, they do not require Markov Chains that impose additional computational costs and fail to scale to higher dimensions; fourthly, they are usable with neural network models known to be universal approximators; and lastly, they are reported as generating subjectively better samples (Goodfellow, 2017).

Despite the range of advantages over the other generative frameworks, GANs are still notoriously difficult to train; chief amongst the difficulties is the problem of obtaining stability between the two adversarial networks. In the following sections we will review the framework of GANs and the various difficulties involved in training them successfully, and finish with a review of attempts at synthesising audio.

2.1 GAN Framework

The GAN framework is designed around an adversarial conflict between two functions, a Generator (G) which is trained to produce artificial samples and a Discriminator (D) that is trained to distinguish between true samples from a dataset and the artificial samples produced by the generator. This conflict results in a zero-sum minimax game that drives each of the functions to mutual improvement (Goodfellow et al., 2014).

The discriminator is thus trained to maximise the probability of assigning the correct label to samples from the training dataset and samples created by the generator; the generator is trained to minimise this probability:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim P_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{1}$$

An analogy used by Goodfellow et al. compares the generative model to a team of counterfeiters producing fake currency, and the discriminative model as police trying to detect those counterfeits. Counterfeiters begin with the production of unsophisticated fakes but the detection of the fakes by the police causes the counterfeiters to increase the sophistication of their methods; this results in the police needing to improve their ability to distinguish real and fake currency, which again cause the counterfeiters to improve, and so on.

Within the GAN framework, the generator begins with the production of very simple fake samples of data, generally noise, and the discriminator soon learns to distinguish this noise from the true data. The generator therefore must produce different samples which may fool the discriminator for a short time, and once the discriminator is also capable of distinguishing these slightly more sophisticated samples from the true data the cycle begins again.

Over a large number of iterations the generator may begin to generate samples that are of a quality that to a human judge are indistinguishable from real data.

2.2 GAN Training Difficulties

The success of a GAN however is dependent on the generator and discriminator entering into a stable relationship within which neither dominates the other until the point is reached where the generator is capable of producing realistic samples.

2.2.1 Nash Equilibrium

Successful training of GANs requires the generator and discriminator enter a Nash equilibrium, a state where neither player in the minimax game is capable of winning and defeating the other. The majority of GANs however are not being trained with a function that is designed to find the Nash equilibrium of a game; instead they are trained to find a low value of a cost function via gradient descent (Salimans et al., 2016).

Finding the Nash equilibria is actually a very difficult problem and Salimans et al. note that they are not aware of any algorithms that are feasible to apply to the continuous high-dimensional parameter space of the GAN game.

The substitution of gradient descent techniques that seek a minimum cost for the discriminator and minimum cost for the generator is therefore an intuitively sensible move; the goal being that the cost for both generator and discriminator reach zero, the equilibrium point. Unfortunately, adjustments to discriminator that decrease the cost may result in an increase of the cost for the generator, and vice versa, resulting in the two networks constantly adjusting each other without ever reaching the equilibrium point.

Failure of gradient descent to find the Nash equilibrium is a serious difficulty for GANs and results in the models often failing to converge. Much research has focused on finding methods that improve the general ability of models to converge but there is still no guaranteed methodology; the difficulty in finding a workable network architecture and hyperparameters values results in GANs being a difficult problem.

2.2.2 Disjoint Supports

It has been noted that updates to the generator of a GANs tend to worsen as the ability of the discriminator to distinguish between real and generated samples improves and that this is ultimately the result of the probability distributions of the real and generated data being continuous (Arjovsky and Bottou, 2017). A reason for the probability distributions not being continuous is that their supports, the functions that define the subset of the domain containing the elements which are in the dataset, are concentrated on low dimensional manifolds.

For an intuitive understanding behind this consider recordings of bass drums (also known as kick drums), large drums commonly used for Jazz, Rock and other popular music forms. It might appear that such recordings exist in a high-dimensional space defined by the sample rate of each recording multiplied by the audio bit depth but there are limitations on the representation of a bass drum sound within that space.

A bass drum recording begins with a relatively high-pitched burst of noise which is then followed by a loud peak of sound before dropping slowly to a low-pitched wave; these limitations on the representation of a bass drum force the recordings into a lower dimensional manifold than that describable by the higher dimensional space defined by the sample rate and bit depth.

When the probability distributions for the discriminator and generator both inhabit very low dimensional manifolds they are very probably disjoint with no overlap between them. The result is that there will be a 'perfect discriminator' that the gradient descent algorithm is capable of converging to, and which is capable of always distinguishing correctly between real and fake samples. Since the probably distributions are commonly inhabiting very low dimensional manifolds, GANs are afflicted by 'perfect' discriminators which ultimately improve to the point of overpowering the generator.

2.2.3 Vanishing Gradients

A consequence of the existence of 'perfect' discriminators is that the loss function will eventually fall to zero and there will no longer be a gradient to update the loss with during learning iterations. The discriminator therefore needs to be held within a state where it is neither too powerful or too weak relative to the generator.

In the case where the discriminator is too weak relative to the generator, the discriminator will fail to give useful feedback to the generator, and so the generator will fail to learn the probability distribution of the dataset. In the other case where the discriminator overpowers the generator, the discriminator will eventually cause the loss function to drop to zero and cause effective training to end.

2.2.4 Mode Collapse

Mode collapse is a failure of GANs to produce varied generated samples during training; instead the generator begins to output nearly identical samples which are consistently capable of fooling the discriminator. There are two different forms of mode collapse; discrete mode collapse and manifold collapse (Metz et al., 2016).

Discrete mode collapse is the result of modes of data in the dataset being 'dropped' by the GAN; for example, if we trained a GAN on recordings of twelve notes played on a piano, the GAN may ultimately only be able to generate six of the twelve notes; the other notes having been 'dropped'.

Manifold collapse is the result of the inability of the GAN to reproduce the entire manifold of a continuous dataset; for example, if a series of samples of various finger positions on a violin string were recorded, with the aim of generating the entire set of frequencies including and between those positions, the manifold would represent the entire range of frequencies between the highest and lowest recorded. Manifold collapse in this example would be the failure of the generator to produce samples within ranges of the frequencies recorded.

2.3 Conditioned GANs

The first GAN was unsupervised and generated data samples randomly from the dataset it was trained from; but Goodfellow et al. already noted at the end of their article describing the original GAN framework that the z input could be conditioned with the concurrent input of labels (Goodfellow et al., 2014).

The first reported implementation of conditional GANs was demonstrated with both image and textual-tag generation via simple concatenation of an extra y input label onto the z input into the generator and onto the real sample data input into the discriminator (Mirza and Osindero, 2014).

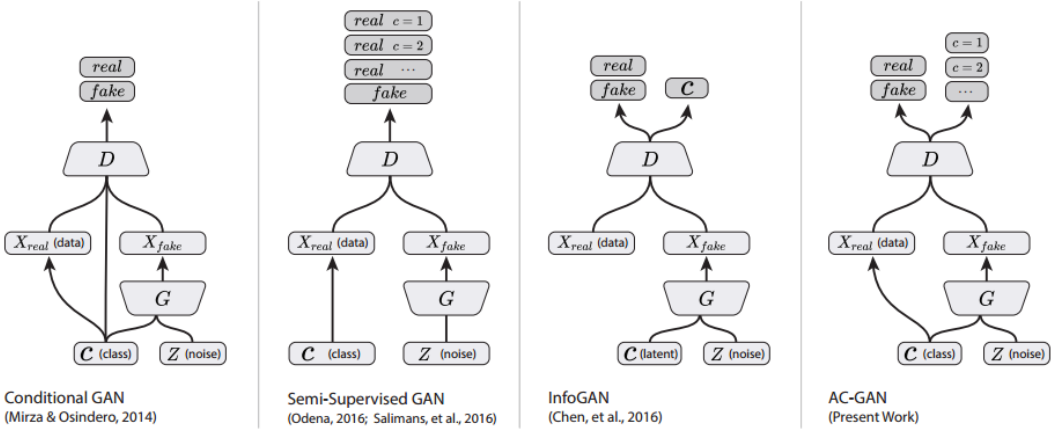
$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim P_{\text{data}}(x)}[\log D(x|y)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z|y)))] \tag{2}$$

Since then the conditional GAN has been successfully applied to other image generation and modification tasks: one variation has been used to age human faces in images (Antipov et al., 2017).

Other methods of conditioning the GAN framework beside simple concatenation have also been devised: the semi-supervised GAN, which utilises an auxiliary decoder network that outputs class labels for the training data; InfoGAN, which utilises an auxiliary decoder network to output a subset of the latent variables from which the samples are generated; and the Auxilliary Classifier Generative Adversarial Network (ACGAN) which conditions the z input with labels but also uses an auxiliary network to classify the output samples.

Figure 1 portrays a selection of the most prominent conditional architectures (Odena et al., 2016):

Figure 1: Conditional GAN Architectures from Odena et al.



2.4 Generated Audio

It was noted in the introduction that the application of generative neural network frameworks to the generation of raw audio waves has not yet received a similar intensity of research as the generation of images; despite speech synthesis being a common application that has great potential for improvement through new techniques. The most common method of speech synthesis has been a concatentative text-to-speech method which strings together pre-recorded words or syllables in a manner that seems unnatural when compared to true speech.

A lack of suitable audio datasets for training undoubtedly retarded efforts to train effective deep neural networks for audio generation, but this is perhaps more a symptom rather than cause of the deficit of research in audio generation, given that there are two disadvantages that samples of audio have in comparison to images for deep learning.

Firstly, recordings that are meaningful to a listener need to be somewhat larger in data size than images which can equivalently be understood meaningfully. Recordings less than a second long and with a sample rate lower than 4096 samples per second resemble little more than noise whereas the images of the MNIST dataset are saved as only 784 pixels.

Secondly, images are quickly understood and evaluated by humans due to their instantaneous nature; the quality of an image is almost immediately discernible to an evaluator. Images may also be printed for evaluation by readers and placed side-by-side for comparison. In contrast, a recording needs to be listened to through a period of time to be evaluated; it is not possible to sensibly listen to multiple recordings in parallel and they cannot be printed for evaluation by readers of a journal.

The first notable application of a deep neural network to the generation of raw audio waves was DeepMind's WaveNet, this was not a GAN but an autoregressive model (van den Oord et al., 2016). The original WaveNet could not be efficiently applied to real-time production due to the architecture requiring sequential generation of individual samples (a difficulty that GANs were designed to overcome) but the authors have since reported the design of an efficient Parallel WaveNet which is deployed in production for Google Assistant (van den Oord et al., 2017).

There have been a small number of applications of GANs to raw audio since the development of WaveNet, including the Speech Enhancement Generative Adversarial Network (SEGAN) which was created to demonstrate the application of GANs to the problem of denoising speech (Pascual et al., 2017); and VoiceGAN which demonstrated the application of the GAN framework to 'style transfer' for human speech (Gao et al., 2018).

The WaveGAN architecture which this project will explore extensions of is another recent application of the GAN framework to raw audio; it was developed as a competitor to the WaveNet architecture and was initially reported as generating superior quality audio and having the advantage of parallel sample generation Donahue et al..

The second advantage has however been rendered void by the introduction of the more efficient Parallel WaveNet and WaveGAN has a disadvantage in comparison to WaveNet; it is currently only capable of generating audio samples of a preset length whereas WaveNet may generate audio of arbitrary length. Donahue et al. have however reported continuing work on an extension to WaveGAN that will allow for the generation of arbitrary length samples; so there is healthy competition currently in the development of generative audio.

3 Model Design

Generative Adversarial Networks consist of two neural network models, a generator and discriminator, competing in a minimax game with the goal that the generator will be trained to produce new samples of data at a high standard.

3.1 WaveGAN Model

WaveGAN is a modified Deep Convolutional Generative Adversarial Network (DCGAN) architecture adjused to train with 1-dimensional audio datasets instead of 2-dimensional image datasets (Radford et al., 2015). It follows the architectural recommendations of Radford et al.: replacing pooling layers with convolution layers; using rectified linear unit (ReLU) activation function in all generator layers other than the output; and using LeakyReLU activation function for all layers in the discriminator.

3.1.1 Architecture

The architecture for the WaveGAN model designed for this project was a modification of the original WaveGAN, the major difference being the reduction of the number of layers in both the generator and discriminator.

The generator had the lowest convolution layer removed as the original WaveGAN was generating samples of size 16,384; the modified version outputs layers of size 4,096. Conversely, the discriminator had the uppermost convolution layer removed; in that case the layer was accepting input samples of size 16,384 but the modified WaveGAN accepts samples of size 4,096.

The reduction of the count of layers required an adjustment of the number of filters applied to the convolution layers at each level so that the generator and discriminator have the correct tensor dimension size at the input and output convolution layers.

3.1.2 Generator

The z input to the generator is a length 100, 1-dimensional array of real numbers between -1 and 1, sampled from a uniform distribution. This array passes through a dense layer that transforms it into a tensor with dimensions suitable for the first convolution layer; a layer with length of 16 and 256 filters.

The data then passes through a series of transposed convolution layers each with a larger length but a reduced count of filters; these layers use a kernel of size 25 and a stride distance of 4. Each of the convolution layers is followed by a ReLU layer that retains only the positive aspect of the real numbered data.

The output layer of the generator is a tanh function which represents a hyperbolic tangent; the effect is to ensure that the data is mapped to the range -1 to 1. The output of the tanh function is a generated raw audio waveform (Table 3).

3.1.3 Discriminator

An audio sample is passed into the discriminator as a parameter and enters the first convolution layer of length 4,096. The data is then passed through a series of alternating layers: 1-dimensional

convolution (with kernel size 25 and stride length 4), LeakyReLU, and phase shuffle. Finally the data is reshaped then passed through a dense layer that outputs single digits (Table 2).

3.1.4 Phase Shuffle

The original WaveGAN experiment showed that audio generated by transposed convolution layers tends to produce artefacts at linear multiple frequencies of the sample rates of each convolution layer (Donahue et al., 2018). Phase corresponds to the alignment of audio samples along the time dimension; in audio processing the phase is used in reference to the exact alignment of stereo tracks in relation to each other.

Shuffling the phase of the audio between convolution layers was shown by Donahue et al. to reduce the occurrence of these artefacts in the sound. This shuffling involves the shift of the 1-dimensional audio layer forwards or backwards by a random number of sample positions at each layer; for this project the a phase shuffle layer was placed between the convolution layers of the discriminator and the number of sample positions moved was a random number between -2 and 2, as recommended on the basis of the experimental results of the WaveGAN experiment.

This shifting of the entire data layer forwards or backwards results in some samples being moved out of the layer, and data being removed from other samples. This problem is resolved through 'reflection' where the samples that are moved out of the layer are placed back in at the opposite end, where there would now be missing samples.

3.1.5 Adam Optimiser

The Adam optimiser is an algorithm that is used in place of stochastic gradient descent to update network weights in a neural network. It combines the advantages of two other variants of stochastic gradient descent, AdaGrad and RMSProp, to allow for the adaptation of parameter learning rates depending on both the mean and uncentered variance of the gradients.

Adam is generally considered the default optimisation algorithm for GAN (perhaps influenced by the recommendations of Radford et al. in their paper introducing DCGAN) and was chosen for the original WaveGAN model. It is used to update both the generator and discriminator with variable learning rates depending on the selection of hyperparameters for each training run.

3.1.6 Wasserstein Loss with Gradient Penalty

The Wasserstein metric is a measure of the distance between two probability distributions that may be used to calculate the minimum cost of transforming a first probability distribution into a second. This metric is colloquially known as the 'earth mover's distance' (EMD) due to the common conceit that the probability distributions are sets of piles of earth; one set being the piles of earth as they actually are, and the other set as we would like them to be, with the distance being the number of steps needed to effect the transformation from one to the other.

Loss functions based on the Wasserstein metric transform the discriminator in a GAN from being a critic of the generator to an assistant that helps the generator estimate the distance between the real and generated distributions. Unfortunately, these functions require a method called 'clipping' to enforce a Lipschitz constraint on the output, with the consequence that the capacity of the models and their ability to model complex functions is reduced.

The original WaveGAN used a loss function that is a modification of the Wasserstein GAN loss function but which also avoids the necessity for clipping through the introduction of a gradient penalty that ensures the Lipschitz constraint remains close to the target norm value of 1. It has been demonstrated that the Wasserstein loss function with gradient penalty for GANs increases the stability of training (Gulrajani et al., 2017).

3.2 Conditional WaveGAN Model

There are a variety of methods for conditioning the inputs to GANs; this report has previously noted that major methods include simple concatenation, a semi-supervised architecture, the InfoGAN architecture, and the auxiliary classifier architecture. The current project will attempt two different methods of conditioning; the first is the simple concatenation of Mirza and Osindero and the second is the auxiliary classifier method devised by Odena et al..

The concatenation method is chosen as it is the first and simplest of the methods to apply and requires very little change to the original WaveGAN architecture. The auxiliary classifier architecture has been chosen as it was deliberately devised to include the best features of both the semi-supervised and InfoGAN architectures.

3.2.1 Concatenation

The concatenation method requires the introduction of labels specifying the class of each sample into both the generator and the discriminator. The labels are passed as parameters to the neural networks and then concatenated onto the z within the generator and the true data sample in the discriminator.

In order to concatenate this extra labelling information onto the tensors within the generator and discriminator, the labels (which are originally created as one-hot labels) are transformed into embedding layers. An embedding layer is similar in structure to a one-hot label but is transformed so that the 'zeroes' and 'ones' do not only fill a single point within a tensor, they instead fill an entire dimension.

The embedding layer for the generator matches on multiple dimensions the size of the z random input after it has been transformed by the initial dense layer; except that the dimension on which it will be concatenated has a length matching the number of modes of data within the dataset (Table 7). These layers that match the modes of the data are ordered to represent those modes; the first layer will match the first mode, the second layer will match the second mode, and so forth. Each of these layers then completely fills the entire dimension with 'zeroes' or 'ones'; the layer with 'ones' representing the mode of data on which the generator should be conditioned.

The discriminator is similarly conditioned with an embedding layer which matches the dimensions of the input samples; the length of the embedding layers is 4,096 but the embedding layers are placed in separate channels and concatenated onto the true samples (Table 4).

The introduction of the embedding layers also required changes to the Wasserstein-GP loss function; the discriminator call within the function was adapted to also accept the embedding layers for the true data samples.

A second simpler loss function was also tested with the concatenation method of conditioning; this second loss function calculated the loss via the mean of the sigmoid cross entropy of raw

logits being provided by the discriminator (in this case the final dense layer from the baseline WaveGAN was bypassed and an extra convolutional layer was introduced to output the logits.)

A novel method of conditioning the discriminator was also tested; in this case the embedding layers were not concatenated onto the true data sample but were instead multiplied with the true data sample. The result of this operation is to produce a new tensor that has the same dimensionality as the original embedded layer tensor but with the dimension that would be filled with 'ones' instead containing the data samples that represent the true data. In this case the data sample is conditioned by the channel that it has been multiplied into rather than an extra layer of 'ones'.

3.2.2 Auxiliary Classifier

The second method to be used to introduce conditioning to WaveGAN was the ACGAN framework developed by researchers at Google Brain. This method was designed to take advantage of two different strategies that had been introduced in previous conditional GAN architectures: class conditioning via concatenation and the introduction of an auxiliary decoder tasked with reconstructing class labels (Odena et al., 2016).

The class conditioning via concatenation occurs only within the generator of the ACGAN framework (Table 5). There is no difference in methodology here in comparison to the simpler concatenation model of conditioning; an embedding layer is passed into the generator along with each random z array and is concatenated onto the output of the dense layer that transforms the noise into a tensor of the correct dimensionality.

The discriminator however is unlike the conditional GANs discriminator as it doesn't accept an embedding layer for concatenation conditioning and it also outputs two different probabilities (Table 6). The first probability to be output is identical to that produced by the baseline WaveGAN discriminator; a probability that the audio sample tested is either from the real dataset or is instead a sample output by the generator. The second probability however is the probability that the sample is sampled from any of the modes of data existing within the dataset:

$$P(S|X), P(C|X) = D(X) \tag{3}$$

The loss function for the ACGAN has two components; firstly a calculation of the loss via the mean of the sigmoid cross entropy of the logits, identical to the loss function that has previously been described as having been tested with the concatenation method of conditioning; and secondly, the calculation of the mean of the softmax cross entropy of the categorical logits and one-hot arrays representing the classes defined for the real dataset and generated samples. The sum of both of these component losses for the generator and discriminator are then output as the final loss.

3.3 Hyperparameter Search

A key difficulty in training GANs is finding a set of hyperparameter values that allow the networks to converge and successfully train the generator to produce interesting samples. Much research into the properties of GANs is driven by the desire to understand how the hyperparameters influence the training and how they may be selected via automatic methods.

Initial testing of the baseline WaveGAN framework developed for this project showed that the hyperparameters used for the original WaveGAN model were not transferable to the downsized model, despite being trained on the same data and differing only in the number of layers in the generator and discriminator. These tests showed early divergence within the loss function with the discriminator loss falling rapidly to negative infinity and the generator loss increasing rapidly to positive infinity; suggestive of the discriminator overpowering the generator.

A series of tests of the various hyperparameters were therefore run so as to discover the effect each had on the likelihood of the networks diverging and no longer being able to learn. The results of a number of these tests are produced as tables in the appendices.

Testing proceeded on three different versions of the models; the baseline WGAN and two versions of the conditional WGAN model each with slightly different implementations of the conditional Wasserstein-GP loss. These models were also reduced versions of those that would eventually be used for the final experiments; they tested and generated audio samples of size 1,024, a convolution layer smaller than the final networks.

The parameters initially tested were the batch size, the Adam optimiser learning rate, the Wasserstein-GP loss lambda variable, and the relative proportion of discriminator to generator updates per iteration of the model. The parameters were tested over a series of values for five thousand iterations; early stopping was introduced for the models when either the absolute discriminator or generator loss value passed 100 (chosen based on previous tests showing this value was only ever passed when the networks started to diverge).

Each model was tested with every chosen value for each parameter ten times and the number of iterations reached before divergence (rounded down to the nearest thousand) were recorded, unless the number of iterations reached 5,000, at which point the test was complete. During the testing of values for each hyperparameter, all other hyperparameter values were maintained at the values which were reported as successful for the original WaveGAN model.

The tests were adapted throughout the process as some parameter values were quickly revealed to be unusable and then dropped from further tests. Early results also showed that one of the implementations of the conditional Wasserstein-GP function was superior to the other and from that point onwards the inferior implementation was dropped from testing. A small number of tests were also run on the larger network models that dealt with audio sample rates of 4,096 samples per second; these confirmed that the trends in the values were relatively consistent across the models of different sizes.

Results showed that the values chosen for each of the tested hyperparameters had varying effects on whether and how early the generator and discriminator losses were likely to diverge; although it was also clear that there was a large degree of randomness determining how quickly any particular test would result in divergence.

Batch size had a clear effect on the likelihood of a model diverging but it was not initially clear whether this was evidence that smaller batch sizes resulted in more stable training, or whether this was the result of a slow down in the overall training of the models, with the training of each epoch simply taking more iterations.

The Adam optimiser learning rate had a clear effect on whether a model was likely to diverge

early; the initial value used for the original WaveGAN model (1e-4) was found to be too high for the tested models as they all diverged very early. The tests showed that lower values (from 1e-5) were far more likely to avoid early stopping.

The lambda variable within the Wasserstein-GP loss function also displayed a noticeable effect on the number of iterations passed before the divergence of the loss function. Higher lambda values tended to prolong the training and avoid early stopping whereas any lambda value less than 10 with the original WaveGAN model hyperparameters was very likely to diverge early.

The proportion of discriminator to generator updates was tested with the number of discriminator updates either equal to or larger than the number of generator updates; this choice was made with consideration that the original WaveGAN model used a rate of five discriminator updates to each generator update. The tests showed that with even an equal number of iterations the models only avoided diverging before 5,000 iterations approximately half of the time; suggesting that the discriminator was relatively more powerful than the generator.

Tests of larger proportions of generator to discriminator updates were not completed at this stage as consideration was first given to other methods that might decrease the power of the discriminator, such as the addition of Gaussian noise to the discriminator input layer and the use of soft and noisy labels in the loss functions.

The hyperparameter search was extremely useful as it revealed the tendencies of each parameter to effect the likelihood of divergence. Unfortunately it did not however reveal any parameter values as having a dominant and stabilising effect on the training and further testing was required during the experimentation stage of the project.

4 Data

The architecture and functions of a neural network model are to some degree determined by the dataset chosen for training; in this case a dataset has already been created for experiments with digital audio, the *Speech Commands Dataset* (Warden, 2017) but we will see how it has been adapted for GAN experiments.

4.1 Characteristics of Digital Audio

The major characteristic of audio data is that it is 1-dimensional and is understood only through passage in the time dimension. Audio data may be mono or stereo, or may have even more channels than stereo (such as audio used with surround sound systems) but this experiment focuses on mono audio only with 1 channel only.

Digital audio separates the single time dimension into points of measurement that are referred to as samples; these samples each describe a point on a complex sound wave which is the result of multiple intermixed frequencies. For digital audio to be heard as sound, the digital stream of these measurements must be converted into analogue at the point where sound is introduced into the environment, such as via speakers, headphones or audio monitors.

4.1.1 Comparison with Images

This report has already noted that audio has disadvantages compared to images when being considered as a dataset for GAN training but these disadvantages are worth further consideration as they have some consequences for the training process.

Images are static and can be evaluated almost instantaneously; they have an immediacy that audio recordings do not. A human evaluator only needs to look at an image and can almost immediately judge whether it is a realistic representation of a figure or not; but to evaluate the quality of an audio recording the human evaluator must listen for a set period of time.

A consequence of audio requiring a set period of listening time is that it is more difficult to compare audio recordings to each other than it is to compare images. When comparing images the evaluator may have both images available in their vision and may quickly move their focus between them; when comparing audio however they must first fully listen to one sample before they may then consider the other, and must retain in memory the segments they wish to compare.

This time taken to switch between audio samples and the need to hold the evaluation of one sample in mind while listening to another makes comparison difficult. Long samples increase the difficulty of retaining in memory the section of the sample the evaluator wishes to compare; slicing the samples into small sections may help but introduces the difficulty of trying to understand slices of audio outside of the context of the recording.

There is also a difference in the size (in terms of data) of an image that can be comprehended by a human evaluator and the size of an audio sample that can be comprehended; very small images may be easily understood, especially if they contain symbolic representations such as letters or numerals. Small audio files are however less easily understood by an evaluator; they cannot be resized as images can without fundamentally changing the nature of the sound being represented.

For example, the shortening of a sound may have two consequences: if done naively the pitch

of the audio recording will increase until it bears little resemblance to the original recording; at moderate levels this produces a 'chipmunk' effect but beyond that the sound becomes unrecognisable; however, if the recording is shortened with an algorithm that maintains the original pitch, then the sound still becomes unrecognisable because it passes at a speed beyond comprehension.

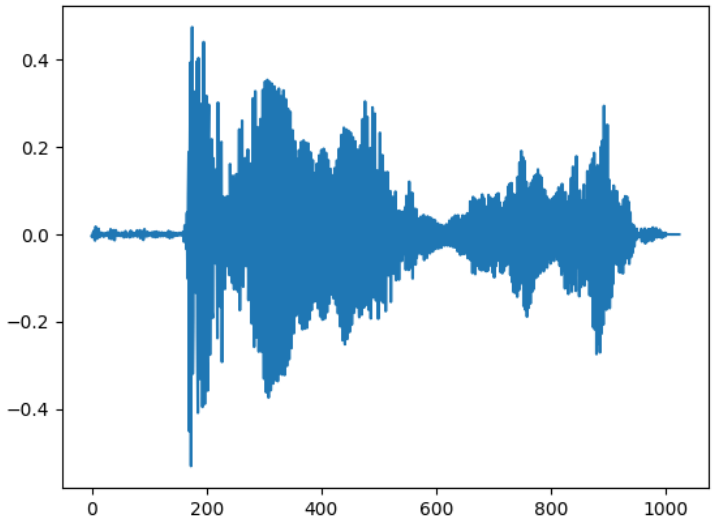
Another method of reducing the size of an audio sample is to lower the sample rate; this may be effective to a certain degree as audio often contains frequencies which are not essential for human comprehension of the sounds. However, this method may reduce the size of the recording in terms of the amount of data used for storage but it doesn't reduce the time required for a human listener to evaluate it.

4.2 Graphical Representation of Audio

The difficulty in evaluating audio samples during the training process required the introduction of visual feedback to facilitate monitoring, and especially to assist in deciding whether training should continue or be stopped and restarted. Graphs showing the wave and spectogram of the real and generated audio samples were therefore output to the tensorboard monitoring dashboard.

The first plot type shows the wave of the audio files with the time dimension displayed on the x axis and amplitude of the audio shown on the y axis. The peaks and troughs in the graph correspond to the changes in the amplitude of the frequencies over the time period of the audio recording; these plots can show how the shape of the generated samples changes from random noise to something similar to a true recording.

Figure 2: Example of an audio waveform plot

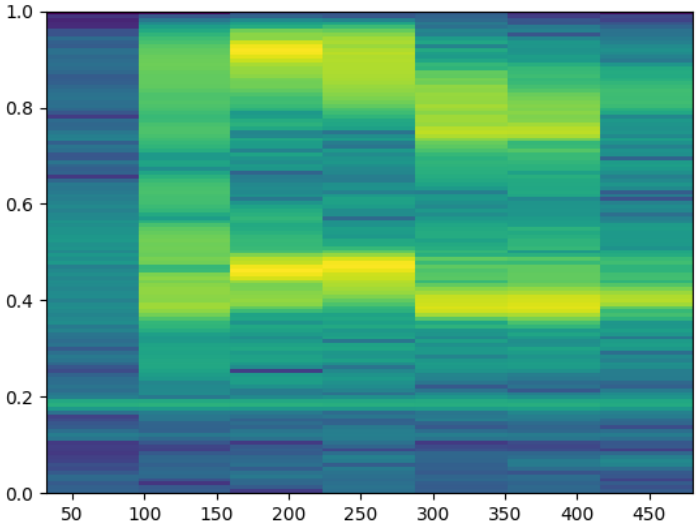


The second plot type is a spectogram which shows how the frequencies of the audio samples vary over the time dimension: the x axis displays the time dimension separated into bins that group

frequencies over a period of time together; the y axis represents the pitch of the frequencies, with lower frequencies near the bottom and higher frequencies near the top. The spectogram also has a third dimension; the colour represents the amplitude and intensity of the frequencies.

The spectogram was useful to monitor how the generated samples were transforming over the epochs of training; the vertical distribution of the frequencies was particularly revealing as it showed whether or not actual frequencies within the audio rather than amplitude only were being learned by the generator.

Figure 3: Example of a spectogram



4.3 Sample Rates

The sample rate of a recording defines the count of samples that are used per second to represent sounds; higher sample rates require more data but allow higher frequencies to be captured; lower sample rates however result in smaller file sizes and lower training times.

Recordings within the *Speech Commands Dataset* are approximately one second long with a sample rate of 16,000 hertz (Hz), or cycles per second. The most recognisable features of human speech lie in the range from 30 Hz to 5,000 Hz so there is an element of redundancy in the *Speech Commands Dataset*; the higher sample rate recordings sound more natural but lower sample rate audio is still recognisable as speech.

Reduction of the sample rate of audio is called 'downsampling'; this operation allows the higher sample rate recording to be reduced in size with a consequent reduction in the number of frequencies represented. Downsampling was used for this project to reduce the computational requirements for training; the subjective quality of the downsampled audio however was still

suitable for evaluation of spoken words.

4.4 Dataset Selection

The Speech Commands Zero through Nine (SC09) dataset is a subset of the *Speech Commands Dataset* which includes only the ten spoken words for the numbers zero through nine; each word being represented by approximately 2,370 recordings. This dataset contains recordings from a wide variety of alignments, spoken by a wide range of speakers, and recorded under a wide range of conditions (Donahue et al., 2018).

This dataset was used for the training of the original WaveGAN model but results in long training times; given that a minimum of two different words are required for the demonstration of conditioning, it was decided that a subset of SC09 would be used for the experiments within this project.

4.4.1 Speech Commands Binary Dataset

A new *Speech Commands Binary Dataset* containing only the spoken words for numbers 'zero' and 'one' was therefore selected for the project. This dataset has words of different syllable counts, with no shared syllables, and the words are easily distinguishable; the dataset also has consistency with the subject matter by representing the numbers used for digital computation.

The training set is derived from the training sets recommended for the *Speech Commands Dataset* and there are 1,850 utterances of each word. A single epoch passing through the sets of recordings for both spoken words therefore requires iteration through 3,700 samples.

4.5 Data Manipulation

A tool named 'Downsample.py' was written in Python to assist in the downsampling of the audio files to be used in these experiments. It is capable of downsampling to various different sample rates and allows a listed collection of the folders (hence words) to be resampled at any time; this allows for the saving of space as it ensures only the necessary files are saved.

A function was added to check that the each resampled file originally had the expected sample rate of 16,384 samples per second so that resampling would be as clean as possible; if the sample rate is higher than expected the excess samples are trimmed off the end; if the sample rate is lower than expected then zero-valued samples are added to the end to ensure the final length of the resampled recording is one-second.

5 Experiments

Experiments reported in the WaveGAN paper showed that the model converges within 700 epochs on a wide variety of datasets including spoken words, bird vocalizations, and drum and piano recordings (Donahue et al., 2018). Given that these datasets were more complex than the *Speech Commands Binary Dataset*, particularly with more modes of data in the SC09 dataset, the experiments for this project were run for approximately 700 epochs then stopped for comparison.

5.1 WaveGAN

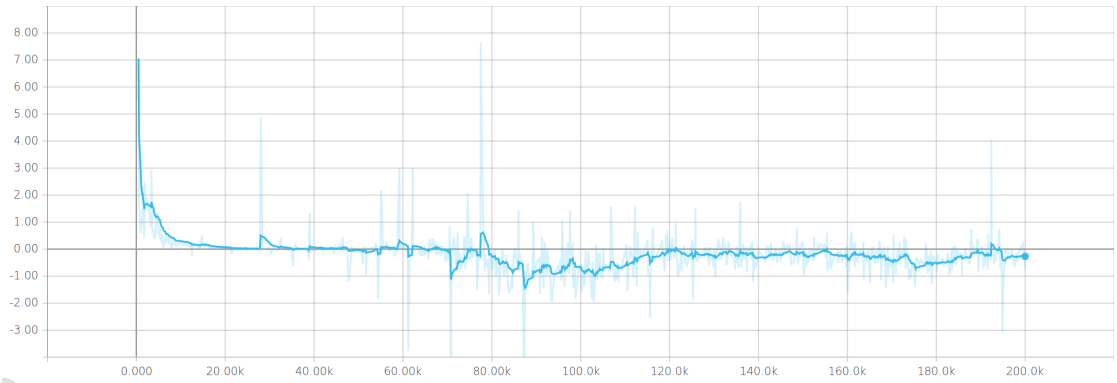
Early testing and the hyperparameter search already showed that the baseline WaveGAN would fail to converge when trained using the hyperparamters reported as successfully used in the original WaveGAN experiment. Training was therefore preceded by further tests on variations in the values of the hyperparameters to find a set that would allow for convergence.

Variations in the values for the batch size, learning rate and the ratio of discriminator to generator updates were all found to stabilize training, but only at relatively extreme values. For example, the original WaveGAN model was capable of converging with a batch size of 64, a learning rate of 1e-4 and a discriminator to generator update ratio of 5 to 1; one of the few hyperparameter settings capable of converging with the baseline WaveGAN used for these experiments was a batch size of 8, a learning rate of 1e-5 and a discriminator to generator update ratio of 1 to 1.

Further experimentation showed that the ratio of discriminator to generator updates was key to the stabilization of the training for the baseline WaveGAN; despite recommendations that Wasserstein-GP loss be used with a discriminator to generator update ratio of 5 to 1, only the reverse ratio of 5 generator updates to each discriminator update was capable of avoiding early divergence with the recommended batch size of 64 and learning rate of 1e-4.

The discrepancy between the values that result in convergence for the original WaveGAN and the baseline WaveGAN used in this experiment needs further investigation; the discriminator is far more powerful in the downsized model compared to the original; ten times more powerful given that the ratio of updates changes from 5 to 1 with the original model to 1 to 5 with this modified model.

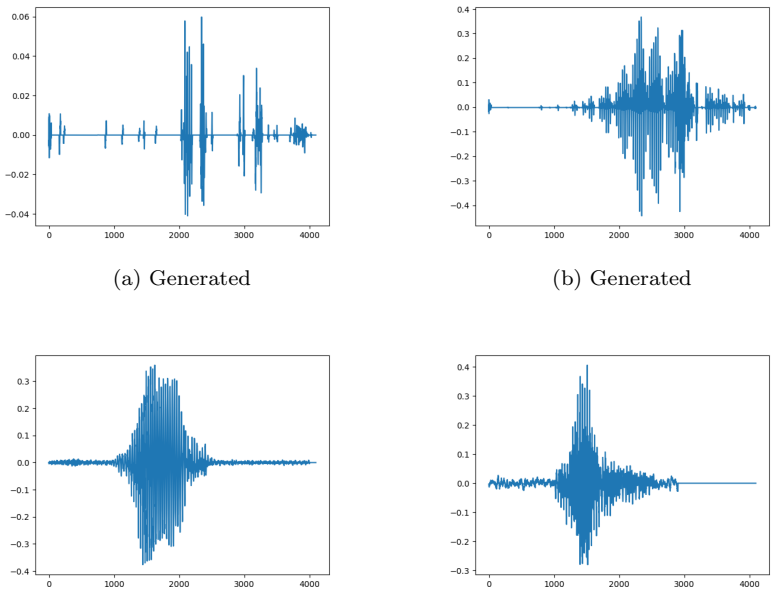
Figure 4: The discriminator loss for WGAN



The architecture was checked multiple times for errors that might be causing the discriminator to be overpowered but the implementation appears sound. Since the model was ultimately capable of generating samples that were distinguishable as words, further investigation on the effect of the number of layers *and the number of filters available for the transposed convolution* is a possible direction of research that might reveal the cause of this imbalance of power between the discriminator and generator.

Within the limitations of the experiment, training of approximately 700 epochs, the baseline WaveGAN was not generating high quality audio samples although they were distinguishable as spoken words. The samples were very noisy with buzzing sounds that suggest the generator had not been able to capture the more complex interactions between the frequencies that constitute a sound.

Visual examination of the plots of the audio waves show that the generator was able to capture the shape of the amplitude of the sounds, relating to the change of the volume of the audio over the period of the sound. These changes in amplitude were shown to be generated very early in the training and seem straightforward for the generator to learn. The wave plots also show that mode collapse has not occurred and that the generator was capable of creating a variety of different sounds (Figure 5).



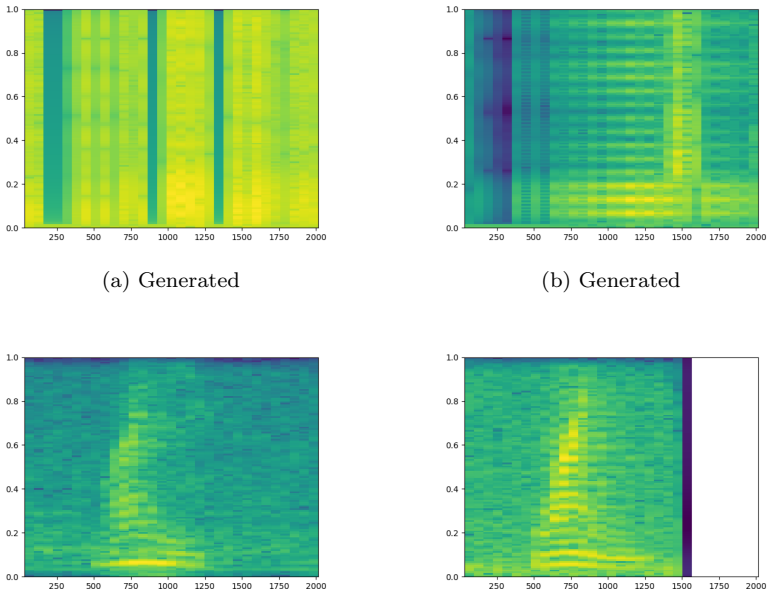
(a) Generated (b) Generated

(c) Real (d) Real

Figure 5: Wave plots of WGAN generated and real samples

The spectogram plots also show that the baseline WaveGAN is capable of generating a varied set of samples and has avoided mode collapse. Horizontal lines of high intensity also divide the plots vertically; this shows that the model has started to capture the frequencies of the sound, but unlike the true audio samples, the lines are straight rather than curved. Again, this suggests

that high level interactions between the frequencies are not being learned (Figure 6).



(a) Generated (b) Generated

(c) Real (d) Real

Figure 6: Spectograms of WGAN generated and real samples

A review of the spectogram plots suggest that the discriminator and generator were repeating the same learning process in a cycle and the generator was unable to ultimately capture the more complex movement of frequencies over time. Still, the baseline WaveGAN shows that it is capable of converging slowly; further training or adjusted hyperparameters that favour the generator should result in a model that generates acceptable samples.

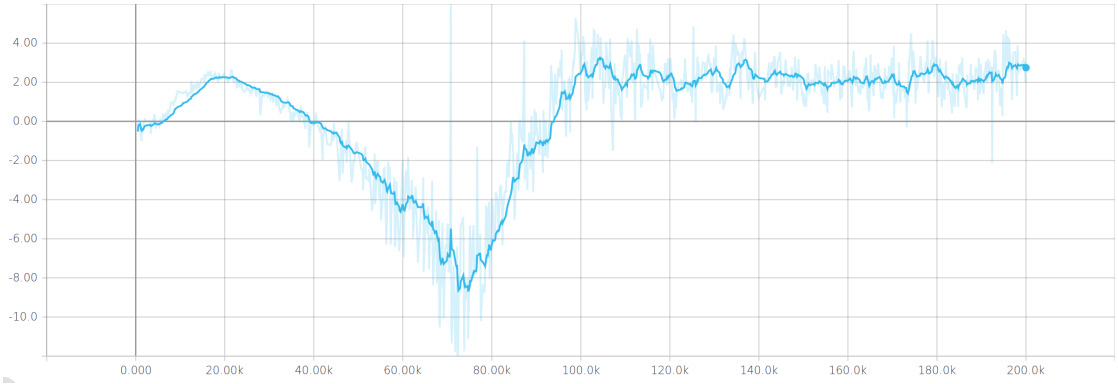
5.2 Conditional WaveGAN with Concatenation

This methodology was unable to train the WaveGAN model successfully; the discriminator loss consistently dropped to zero within a very few epochs. Even when the ratio of generator to discriminator updates was increased to relatively extreme values such as 8 or even 16 the generator was still unable to counteract the early drop of the discriminator loss to zero.

Using the tensor channels as signifiers of the label for each sample had a similar result as the concatenation of labels; the discriminator would still consistently drop to zero within very few epochs. However, the removal of the embedding layer of 'ones' did seem to enable the training to take place more quickly even though it would also ultimately fail; monitoring the wave plots during training showed that the samples would arrive at a sensible shape more quickly than with the simple concatenation method.

Two different loss functions were tested with this concatenation method; the first was the con-

Figure 7: The generator loss for WGAN



ditional Wasserstein-GP loss and the second was a simpler loss calculating from the mean of the sigmoid cross entropy of the logits; neither resulted in an advantage. The introduction of noise to samples entering the discriminator before they were passed to the first convolution layer also had no effect on the tendency of the discriminator to rapidly overpower the generator.

The concatenation method of conditioning the output of the WaveGAN model developed for this project was ultimately a total failure. Even testing with high ratios of generator to discriminator updates, and with the ratio of generator updates increasing each time the discriminator loss dropped below set values, even to extreme values such as 60, the discriminator loss would still drop to zero within a very epochs.

5.3 Conditional WaveGAN with Auxiliary Classifier

The ACGAN method was more stable than the concatenation method of conditioning but was also ultimately unable to produce high quality samples within the constraints of the experiment. It does show some promise however and unlike the concatenation method there may be some advantage gained from training further epochs or further refining the hyperparameter choices.

The best results with the auxiliary classifier were achieved with a learning rate of 1e-3, faster than was possible with the baseline WGAN, and a generator to discriminator update ratio of 2 to 1. This trial run also had label smoothing applied to the labels being compared in the discriminator loss function.

The results from this experiment showed that two modes of data were being generated but there was very little variability within those modes. This would seem to be a form of manifold mode collapse as the discrete modes are still being generated but only a very narrow range of each manifold is generated. The two generated samples in Figure 10 are representative of nearly all the samples being generated.

While the amplitude of the generated waveforms seems superficially similar to those of the real data samples, the spectogram plots in Figure 11 show that the relationships between the frequencies over time have not been captured in a sophisticated manner.

Figure 8: The generator loss for ACGAN

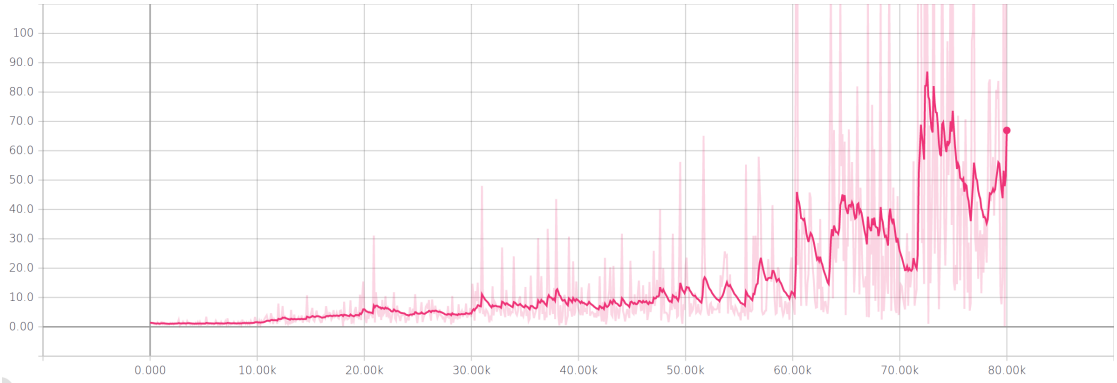
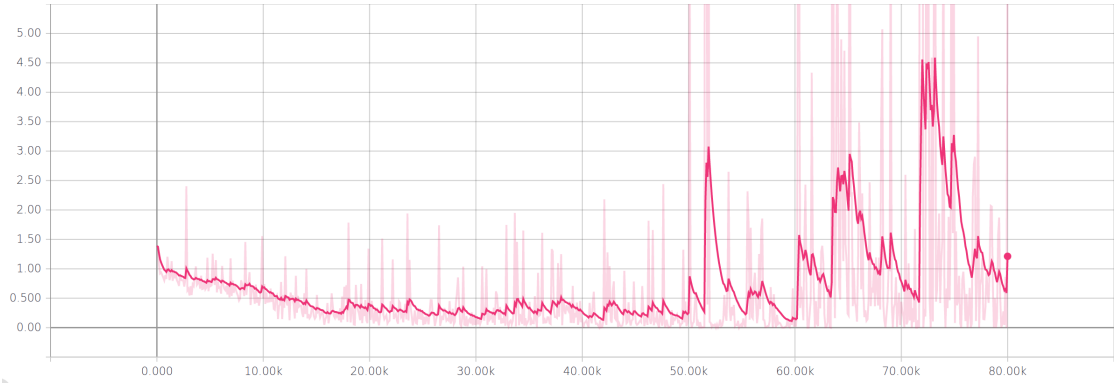
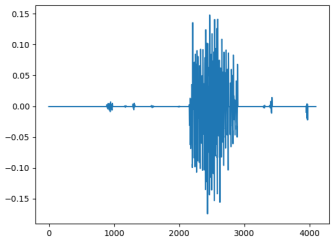
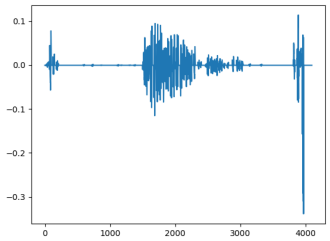


Figure 9: The discriminator loss for ACGAN

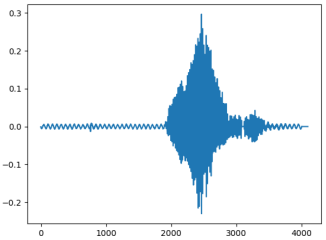




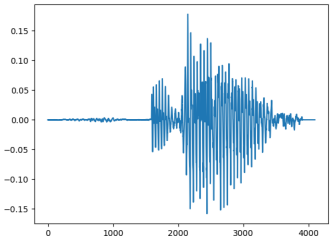
(a) Generated



(b) Generated

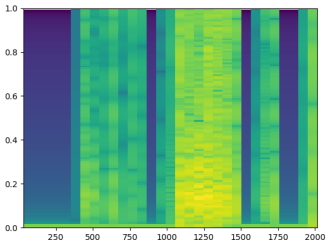


(c) Real

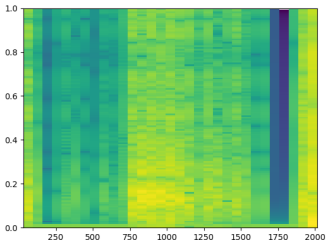


(d) Real

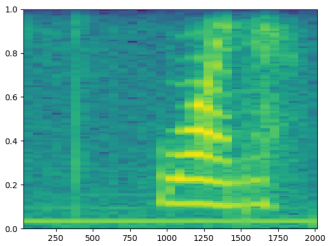
Figure 10: Wave plots of ACGAN generated and real samples



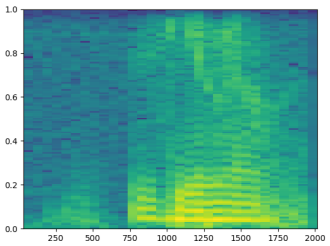
(a) Generated



(b) Generated



(c) Real



(d) Real

Figure 11: Spectograms of ACGAN generated and real samples

6 Conclusion

WaveGAN is a difficult model to replicate and train to convergence; the hyperparameters reported by Donahue et al. do not result in successful training with downsized models and the search for a working set of hyperparameters required a large amount of testing. A greater budget would have allowed the replication of the original WaveGAN experiment and may have freed time to focus on finding the correct architecture and hyperparameters for a conditional model.

The project did however reveal that there are difficulties modifying the number of layers in the WaveGAN model and this way have an impact on the feasibility of designing a WaveGAN with the capability of generating audio of arbitrary lengths. Further research is needed into the relationship between the number of layers in the model, the number of filters that are used in the convolution layers, and the values of hyperparameters that result in convergence.

Designing and training a conditional WaveGAN model upon the already unstable downsized WaveGAN model required significant testing of variations in the architecture and hyperparameters. Two different methods of conditioning were attempted with variations on the cost functions used and with the addition of noise to the discriminator input and label smoothing in the cost functions. Further attempts at producing a stable conditional WaveGAN are not recommended without first researching further the problems with the implementation of the baseline WaveGAN.

Nevertheless, we can at least state that the concatenation method of conditioning is unlikely to be the simplest to develop into a stable functioning model; even extreme values for the parameters had little impact on the time taken for the loss to diverge. The auxiliary classifier method of conditioning showed some promise, even though it was not entirely successful within the limitations of the experiment; a better understanding of the relationship between the original and downsized WaveGAN models will hopefully give an insight into how this method may be improved so as to avoid mode collapse and train more quickly.

During the testing of the concatenation method of conditioning there was an attempt at bypassing the one-hot style embedding layer and instead placing the audio wave data directly into a channel to label the class of data being passed in as a parameter. This is a minor change to the simple concatenation of the embedding layers onto the original data but monitoring showed that this seemed to result in faster training with the generated waveforms taking a form similar to that of the real data more quickly. A minor program of research to review previous conditional GAN experiments and compare results generated with this new method may be worthwhile.

7 Bibliography

Antipov, G., Baccouche, M. and Dugelay, J.-L. (2017), 'Face Aging With Conditional Generative Adversarial Networks', *ArXiv e-prints* .

Arjovsky, M. and Bottou, L. (2017), 'Towards Principled Methods for Training Generative Adversarial Networks', *ArXiv e-prints* .

Donahue, C., McAuley, J. and Puckette, M. (2018), 'Synthesizing Audio with Generative Adversarial Networks', *ArXiv e-prints* .

Gao, Y., Singh, R. and Raj, B. (2018), 'Voice Impersonation using Generative Adversarial Networks', *ArXiv e-prints* .

Goodfellow, I. (2017), 'NIPS 2016 Tutorial: Generative Adversarial Networks', *ArXiv e-prints* .

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y. (2014), 'Generative Adversarial Networks', *ArXiv e-prints* .

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V. and Courville, A. (2017), 'Improved Training of Wasserstein GANs', *ArXiv e-prints* .

Hawkins, D. (2018), 'The cybersecurity 202: Lawmakers want intelligence chiefs to help counter threat from doctored videos', *Washington Post* .

Metz, L., Poole, B., Pfau, D. and Sohl-Dickstein, J. (2016), 'Unrolled generative adversarial networks', *CoRR* **abs/1611.02163**.
URL: *http://arxiv.org/abs/1611.02163*

Mirza, M. and Osindero, S. (2014), 'Conditional Generative Adversarial Nets', *ArXiv e-prints* .

Odena, A., Olah, C. and Shlens, J. (2016), 'Conditional Image Synthesis With Auxiliary Classifier GANs', *ArXiv e-prints* .

Pascual, S., Bonafonte, A. and Serrà, J. (2017), 'SEGAN: Speech Enhancement Generative Adversarial Network', *ArXiv e-prints* .

Radford, A., Metz, L. and Chintala, S. (2015), 'Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks', *ArXiv e-prints* .

Salimans, T., Goodfellow, I. J., Zaremba, W., Cheung, V., Radford, A. and Chen, X. (2016), 'Improved techniques for training gans', *CoRR* **abs/1606.03498**.
URL: *http://arxiv.org/abs/1606.03498*

van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. W. and Kavukcuoglu, K. (2016), 'Wavenet: A generative model for raw audio', *CoRR* **abs/1609.03499**.
URL: *http://arxiv.org/abs/1609.03499*

van den Oord, A., Li, Y., Babuschkin, I., Simonyan, K., Vinyals, O., Kavukcuoglu, K., van den Driessche, G., Lockhart, E., Cobo, L. C., Stimberg, F., Casagrande, N., Grewe, D., Noury, S., Dieleman, S., Elsen, E., Kalchbrenner, N., Zen, H., Graves, A., King, H., Walters, T., Belov, D. and Hassabis, D. (2017), 'Parallel WaveNet: Fast High-Fidelity Speech Synthesis', *ArXiv e-prints* .

Warden, P. (2017), 'Speech commands: A public dataset for single-word speech recognition.', *Dataset available from* *http://download.tensorflow.org/data/speech_commands_v0.01.tar.gz* .

Appendices

A Development Environment

A.1 Software

A.1.1 Tensorflow

Tensorflow (version 1.9.0) is an open source machine learning framework created by Google with an emphasis on the design and training of graph models, particularly neural networks. The discriminator and generator models for this project were both written and run with TensorFlow.

A.1.2 Tensorboard

Tensorboard (version version 1.9.0) is a visualization dashboard tool built to accompany Tensorflow that allows for live monitoring of experiments. A number of features of Tensorboard was greatly important in the training of the models used in this project; particularly the ability to monitor the discriminator and generator loss, visualize the plotted output of the audio samples; and also listen to the audio samples as they are being generated.

A.1.3 Python

All code was written in Python (version 3.6.5.) while utilising a number of libraries, such as: NumPy (version 1.13.3), a library allowing more efficient writing of code for processing array; (version Librosa 0.6.0), a library for the processing and plotting of digital audio; (version Soundfile 0.10.1), another library for audio processing; and matplotlib, a visualization library.

Another library tf-matplotlib was imported to allow integration between Tensorboard and matplotlib, a mathematical plotting library for Python. This library allows for diverse and richer plotting functions than is currently available within TensorBoard to be visualized during training. It is itself dependent on the library matplotlib (version 2.2.0).

A.2 Cloud Infrastructure

The original WaveGAN model implemented by Donahue et al. required four days of training with a top-of-the-range NVIDIA Tesla P100 GPU. At the time of writing, this GPU is available from Amazon for US $5,500, beyond an acceptable budget for a student project. Without physical access to a powerful GPU the experiments were instead run on cloud service GPUs.

A.2.1 Amazon EC2

Amazon Elastic Compute Cloud 2 is a web service that providing computing power in the cloud. Users have a choice over the development environment that they want to use in the cloud; these are called Amazon Machine Images (AMIs) AMIs exist for common tasks; the Ubuntu Deep Learning AMI used for these experiments provides a Ubuntu server interface, Tensorflow, Tensorboard, CUDA and Python.

AWS also includes the choice of 'instances' which are physical configurations of GPUs that the user can compute with. The 'instance' type used for these experiments was the p2.xlarge instance which gives access to only a single GPU, an NVIDIA K80 GPU.

B Instructions for Using the Software

There are two programs; the first prepares data for training; and the second is for training the model, testing hyperparameters and generating data.

B.1 Data Preparation

Preparation of the data takes place through the command line by calling the file 'Downsampler.py'.

The first command is 'divider' which can accept values of '1', '4', and '16'; this commands the program to divide the sample rate of approximately 16k with either of these numbers to obtain a downsampled dataset.

The second and third commands 'inPath' and 'outPath' accept the path where your data is held and the path where you would like to save your downsampled data.

The third command is 'words' which takes a list of strings representing the words from the SC09 dataset that will be downsampled; the default is 'zero' and 'one'.

Following is an example of how to use this command:

```
python3 Downsampler.py 4 '/path/to/my/data/' '/path//I/am/downsampling/to/'
```

B.2 Training and Generating

Training and generation with the models is through the command line by calling the file 'Manager.py'.

The 'mode' command tells the program whether it should 'train' or 'gen' (generate data).

The 'model' command tells the program whether to train or generate from a 'WGAN' (WaveGAN), 'CWGAN' (concatenated WaveGAN), or 'ACGAN' (auxiliary classifier GAN).

The 'wave' command tells the program which wave length (sample size) to use, default is 4096.

The 'runName' argument gives an extra textual name by which the trained model can be identified.

The 'checkpointNum' argument tells the program which checkpoint to use when generating new samples.

The 'genMode' argument tells the program which mode of the data to generate samples from when generation is through a conditional model.

The 'genLength' argument tell the generator how many samples to generate; the default is 100.

The 'lamb' argument pass the gradient penalty lambda number to the Wasserstein loss; the

default is 10.

The 'batch' argument tells the program which batch size to use during model training; the default is 64.

The 'iterations' argument tells the program how many iterations through batches need to be completed before training is complete.

The 'D_updates' argument tells the program how many updates to the discriminator should take place per iteration; the default is 1.

The 'G_updates' argument tells the program how many updates to the generator should take place per iteration; the default is 1.

The 'learnRate' argument passes a learning rate into the Adam optimizer; the default is 0.0001.

The 'words' argument accepts a list of words to include in the training; the default is 'zero' and 'one'.

An example of a command that would train the model:

```
python3 Manager.py -mode=train -model=WGAN -wave=4096 -iterations=2 -lamb=10 -
    D_updates=1 -G_updates=1 -runName=TestA -batch=64 -learnRate=0.0001 -words
    zero one
```

An example of a command that generate from the model created with the previous command:

```
python3 Manager.py -mode=gen -model=WGAN -wave=4096 -checkpointNum=0 -genLength
    =100 -runName=TestA
```

C Model Architecture Tables

C.1 WaveGAN

Table 2: Layers of the WGAN Discriminator

Operation	Kernel Size	Output Shape
Input x or $G(z)$		(n, 4096, 1)
Conv1D (Stride=4)	(25, d, $2d$)	(n, 1024, 32)
LReLU ($\alpha = 0.2$)		(n, 1024, 32)
Phase Shuffle ($n = 2$)		(n, 1024, 32)
Conv1D (Stride=4)	(25, $2d$, $4d$)	(n, 256, 64)
LReLU ($\alpha = 0.2$)		(n, 256, 64)
Phase Shuffle ($n = 2$)		(n, 256, 64)
Conv1D (Stride=4)	(25, $4d$, $8d$)	(n, 64, 128)
LReLU ($\alpha = 0.2$)		(n, 64, 128)
Phase Shuffle ($n = 2$)		(n, 64, 128)
Conv1D (Stride=4)	(25, $8d$, $16d$)	(n, 16, 256)
LReLU ($\alpha = 0.2$)		(n, 16, 256)
Reshape		(n, 4096)
Dense	(4096, 1)	(n, 1)

Table 3: Layers of the WGAN Generator

Operation	Kernel Size	Output Shape
Input $z \sim$ Uniform(-1, 1)		(n, 100)
Dense 1	(100, 4096)	(n, 4096)
Reshape		(n, 16, 256)
ReLU		(n, 16, 256)
Trans Conv1D (Stride=4)	(25, $8d$, $4d$)	(n, 64, 128)
ReLU		(n, 64, 128)
Trans Conv1D (Stride=4)	(25, $4d$, $2d$)	(n, 256, 64)
ReLU		(n, 256, 64)
Trans Conv1D (Stride=4)	(25, $2d$, d)	(n, 1024, 32)
ReLU		(n, 1024, 32)
Trans Conv1D (Stride=4)	(25, d, c)	(n, 4096, 1)
Tanh		(n, 4096, 1)

C.2 Conditional WaveGAN with Concatenation

Table 4: Layers of the Concatenated Conditional WaveGAN Discriminator

Operation	Kernel Size	Output Shape
Concat x and y		(n, 4096, 3)
Conv1D (Stride=4)	(25, d, $2d$)	(n, 1024, 32)
LReLU ($\alpha = 0.2$)		(n, 1024, 32)
Phase Shuffle ($n = 2$)		(n, 1024, 32)
Conv1D (Stride=4)	(25, $2d$, $4d$)	(n, 256, 64)
LReLU ($\alpha = 0.2$)		(n, 256, 64)
Phase Shuffle ($n = 2$)		(n, 256, 64)
Conv1D (Stride=4)	(25, $4d$, $8d$)	(n, 64, 128)
LReLU ($\alpha = 0.2$)		(n, 64, 128)
Phase Shuffle ($n = 2$)		(n, 64, 128)
Conv1D (Stride=4)	(25, $8d$, $16d$)	(n, 16, 256)
LReLU ($\alpha = 0.2$)		(n, 16, 256)
Conv1D (Stride=1)	(16, 16, 256)	(n, 1, 1)

Table 5: Layers of the Concatenated Conditional WaveGAN Generator

Operation	Kernel Size	Output Shape
Dense	(100, 4096)	(n, 4094)
Reshape *as* z		(n, 16, 254)
Concat z and y		(n, 1, 4096)
ReLU		(n, 1, 4096)
Trans Conv1D (Stride=1)	(16, $8d$, $4d$)	(n, 16, 256)
ReLU		(n, 16, 256)
Trans Conv1D (Stride=4)	(25, $8d$, $4d$)	(n, 64, 128)
ReLU		(n, 64, 128)
Trans Conv1D (Stride=4)	(25, $4d$, $2d$)	(n, 256, 64)
ReLU		(n, 256, 64)
Trans Conv1D (Stride=4)	(25, $2d$, d)	(n, 1024, 32)
ReLU		(n, 1024, 32)
Trans Conv1D (Stride=4)	(25, d, c)	(n, 4096, 1)
Tanh		(n, 4096, 1)

C.3 Conditional WaveGAN with Auxiliary Classifier

Table 6: Layers of the Auxiliary Classifier Conditional WaveGAN Discriminator

Operation	Kernel Size	Output Shape
Conv1D (Stride=4)	(25, d, $2d$)	(n, 1024, 32)
LReLU ($\alpha = 0.2$)		(n, 1024, 32)
Phase Shuffle ($n = 2$)		(n, 1024, 32)
Conv1D (Stride=4)	(25, $2d$, $4d$)	(n, 256, 64)
LReLU ($\alpha = 0.2$)		(n, 256, 64)
Phase Shuffle ($n = 2$)		(n, 256, 64)
Conv1D (Stride=4)	(25, $4d$, $8d$)	(n, 64, 128)
LReLU ($\alpha = 0.2$)		(n, 64, 128)
Phase Shuffle ($n = 2$)		(n, 64, 128)
Conv1D (Stride=4)	(25, $8d$, $16d$)	(n, 16, 256)
LReLU ($\alpha = 0.2$)		(n, 16, 256)
Conv1D (Stride=1)	(16, 16, 256)	(n, 1, 1)
Flatten	(64, 16, 256)	(n, 4096)
Dense	(4096)	(n, 2)

Table 7: Layers of the Auxiliary Classifier Conditional WaveGAN Generator

Operation	Kernel Size	Output Shape
Dense	(100, 4096)	(n, 4094)
Reshape *as* z		(n, 16, 254)
Concat z and y		(n, 1, 4096)
ReLU		(n, 1, 4096)
Trans Conv1D (Stride=1)	(16, $8d$, $4d$)	(n, 16, 256)
ReLU		(n, 16, 256)
Trans Conv1D (Stride=4)	(25, $8d$, $4d$)	(n, 64, 128)
ReLU		(n, 64, 128)
Trans Conv1D (Stride=4)	(25, $4d$, $2d$)	(n, 256, 64)
ReLU		(n, 256, 64)
Trans Conv1D (Stride=4)	(25, $2d$, d)	(n, 1024, 32)
ReLU		(n, 1024, 32)
Trans Conv1D (Stride=4)	(25, d, c)	(n, 4096, 1)
Tanh		(n, 4096, 1)

D Model Graphs

The following pages contain the graph structures for the baseline and conditional WaveGAN models as displayed through Tensorboard. The 'Main Graph' segments show the high level architecture of the models; the 'Auxiliary Nodes' segments show the names and structures of various other components.

Figure 12: WaveGAN Model Main Graph

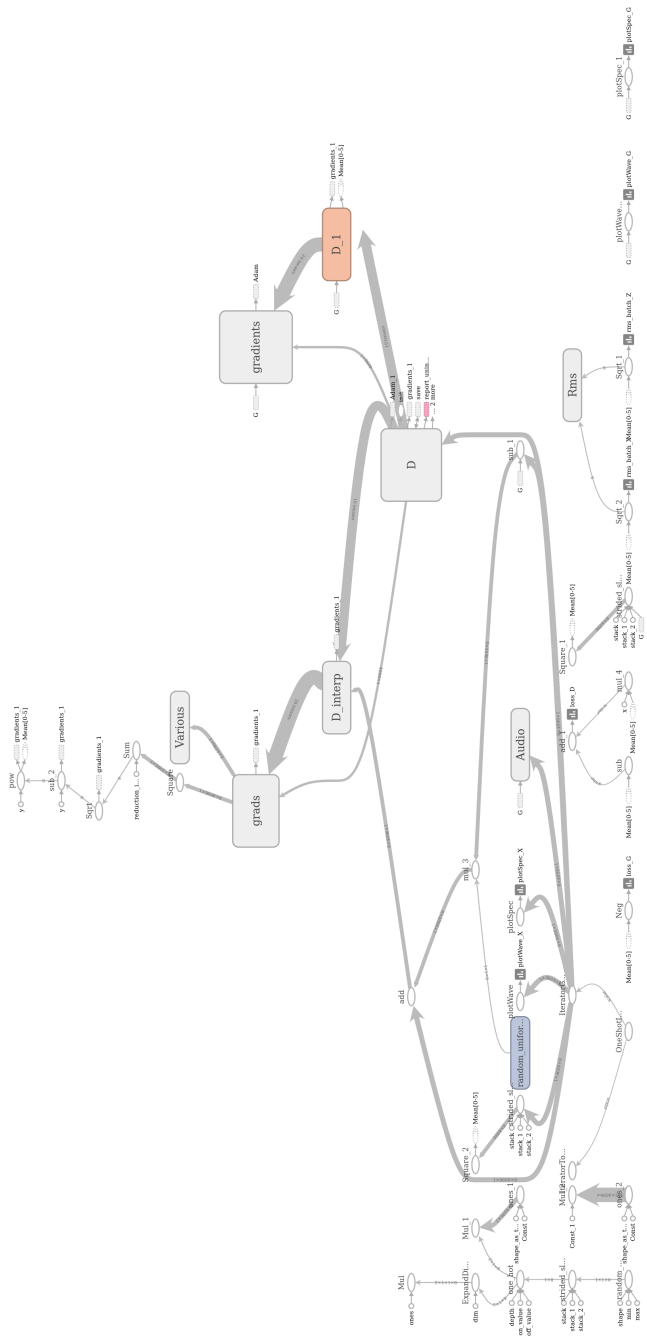


Figure 13: WaveGAN Model Auxiliary Nodes

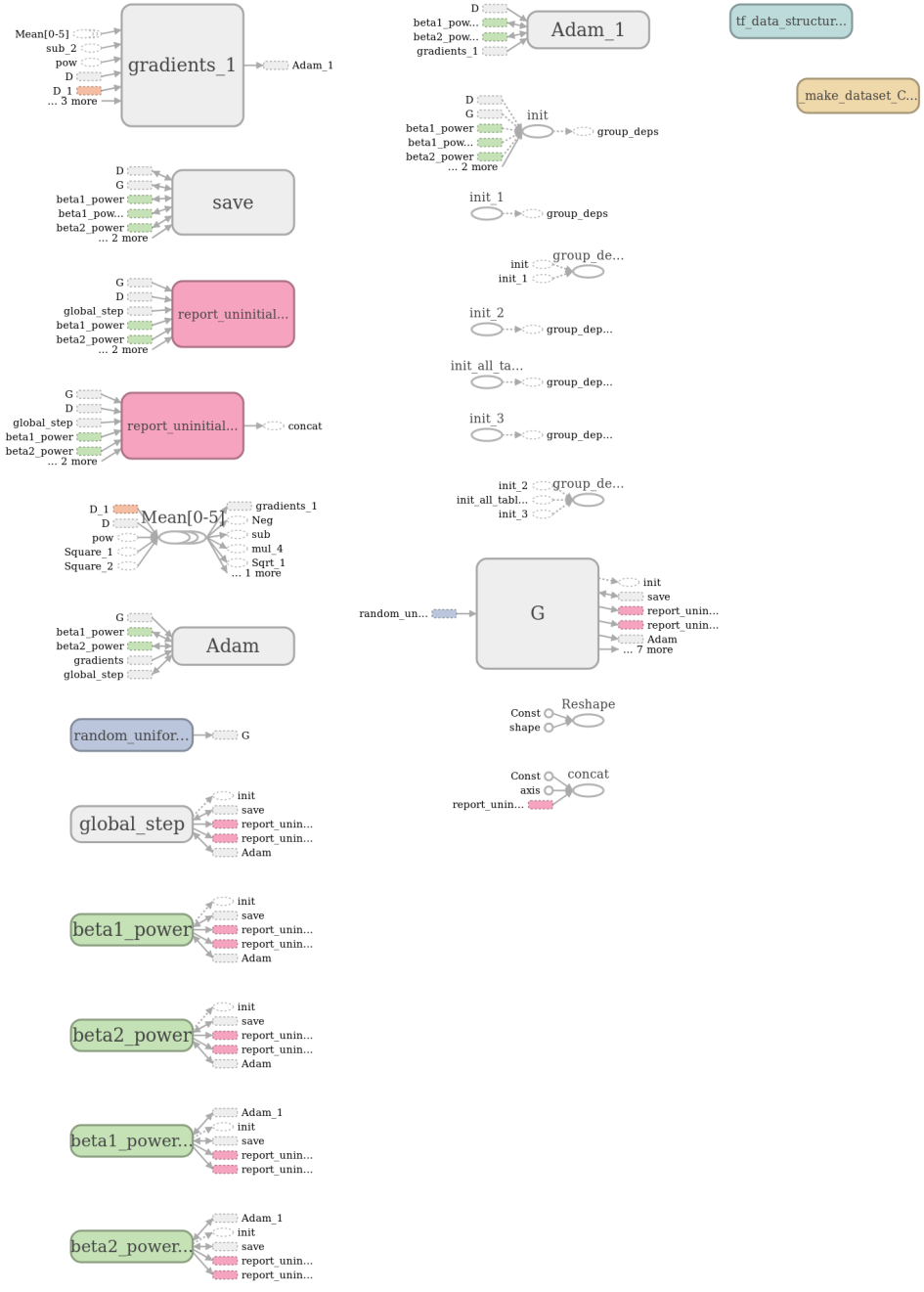


Figure 14: Concatenated Conditional WaveGAN Model Main Graph

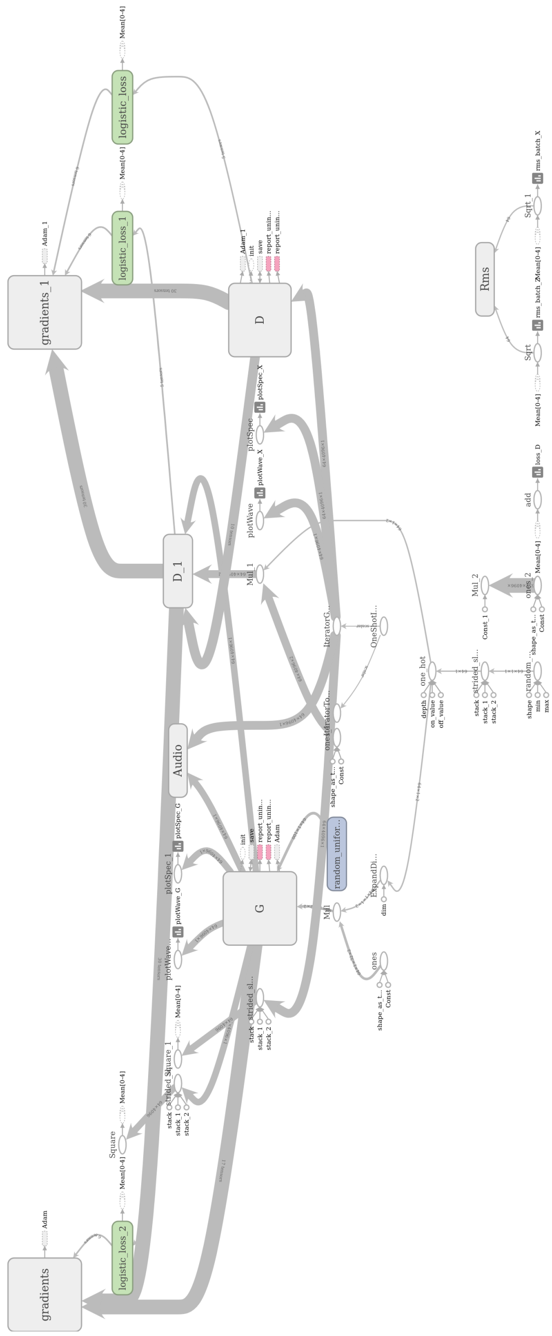


Figure 15: Concatenated Conditional WaveGAN Model Auxiliary Nodes

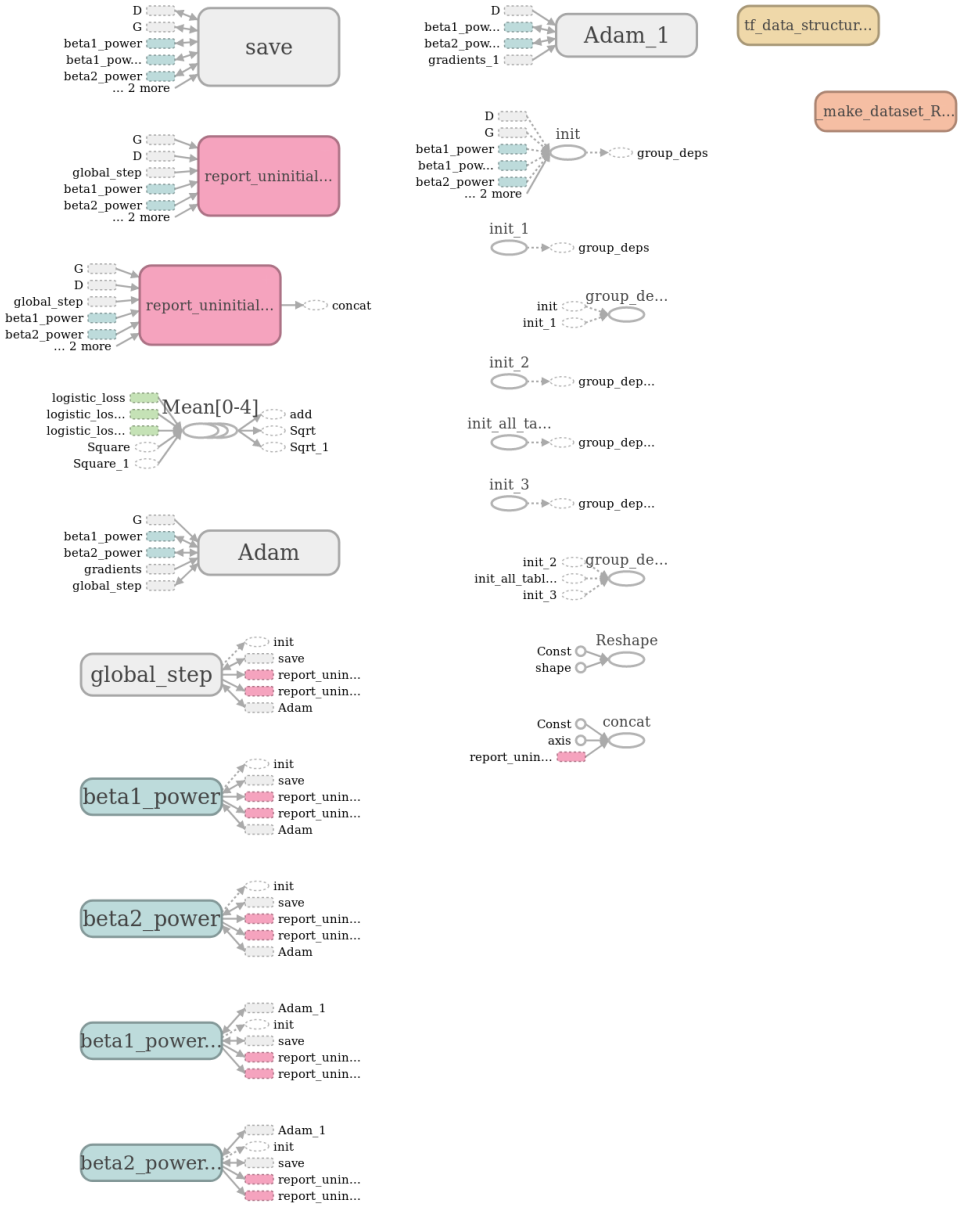


Figure 16: Auxiliary Classifier Conditional WaveGAN Model Main Graph

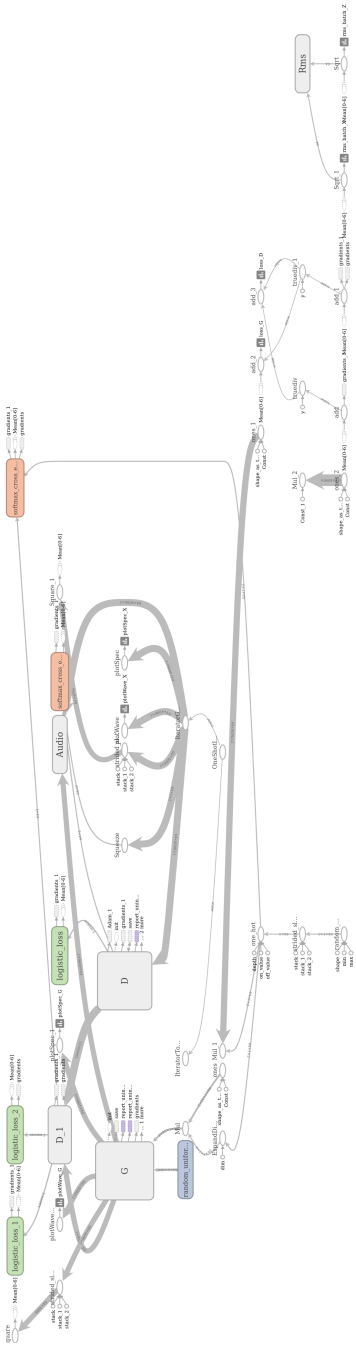
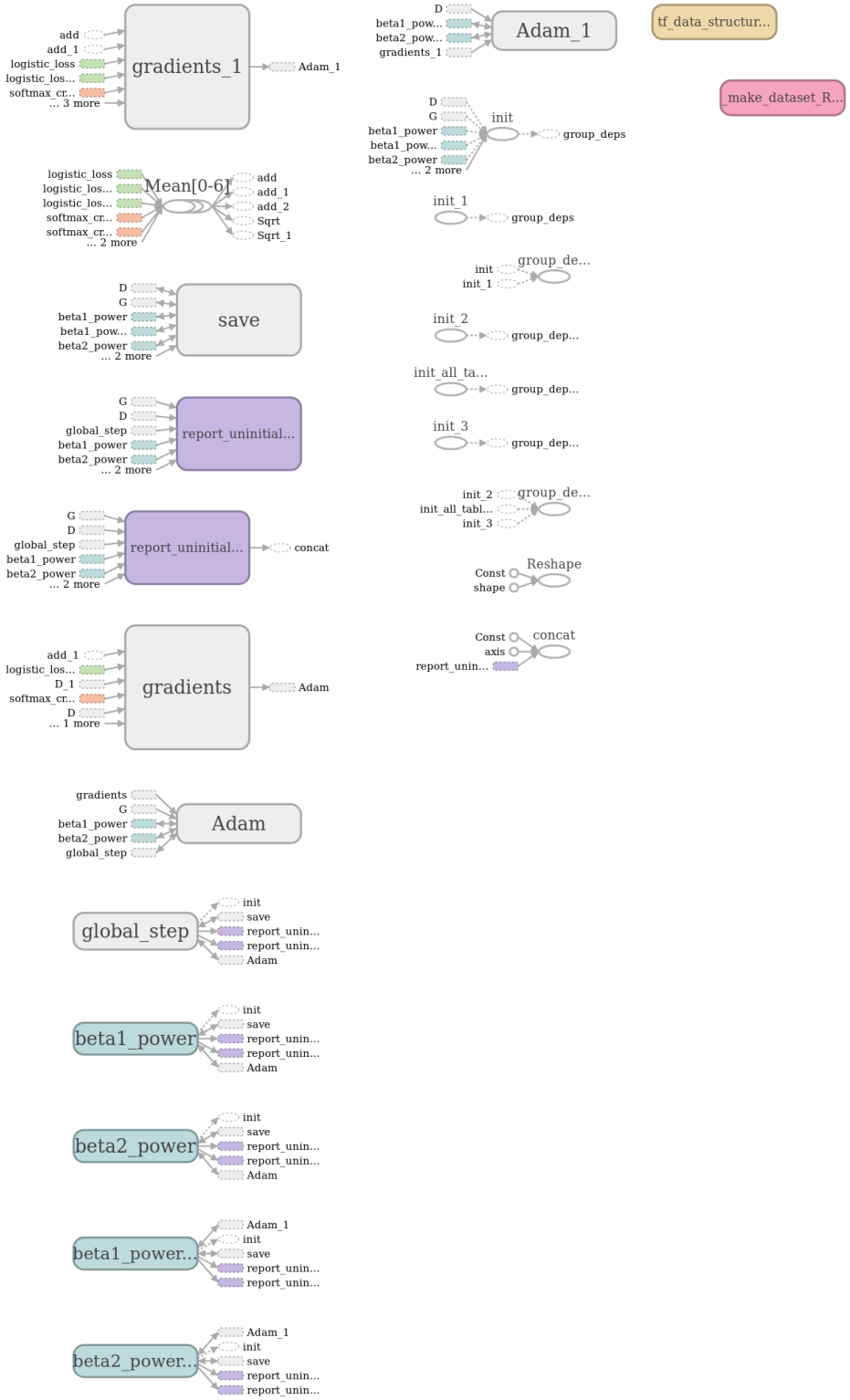


Figure 17: Auxiliary Classifier Conditional WaveGAN Model Auxiliary Nodes



E Hyperparameter Search Test Results

The following tables show the results of various tests that were used to reveal the potential best hyperparameters for the model.

Test were initially run with two different conditional WaveGAN models but I've only included results from the better of the two as it consistently produced better results. The number of iterations are rounded down to the nearest thousand.

Table 8: Table of WaveGAN Learning Rate Experiments

Runs	0.0001	0.00001
1	2k	5k
2	5k	5k
3	4k	5k
4	4k	5k
5	4k	5k
6	3k	5k
7	3k	5k
8	5k	5k
9	4k	5k
10	3k	5k
Avg.	3.7k	5k

Table 9: Table of Conditional WaveGAN Learning Rate Experiments

Runs	0.0001	0.00001
1	1k	5k
2	0	5k
3	0	4k
4	0	5k
5	1k	5k
6	0	5k
7	0	5k
8	0	3k
9	0	5k
10	0	4k
Avg.	0.2k	4.5k

Table 10: Table of WaveGAN Batch Size Experiments

Runs	8	16	32	64
1	5k	5k	3k	3k
2	5k	5k	5k	4k
3	4k	4k	4k	3k
4	5k	4k	4k	3k
5	5k	5k	5k	3k
6	4k	5k	5k	3k
7	4k	3k	5k	5k
8	5k	5k	5k	3k
9	5k	5k	4k	4k
10	5k	3k	5k	5k
Avg.	4.7k	4.4k	4.6k	3.6k

Table 11: Table of Conditional WaveGAN Batch Size Experiments

Runs	8	16	32	64
1	2k	0	1k	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	2k
5	0	0	0	1k
6	0	0	3k	0
7	0	0	0	0
8	0	5k	0	1k
9	0	0	0	0
10	1k	0	0	0k
Avg.	0.3k	0.5k	0.4k	0.4k

Table 12: Table of WaveGAN Lambda Experiments

Runs	10	100
1	3k	5k
2	3k	4k
3	5k	3k
4	4k	5k
5	4k	5k
6	3k	5k
7	5k	5k
8	3k	5k
9	4k	5k
10	4k	5k
Avg.	3.7k	4.5k

Table 13: Table of Conditional WaveGAN Lambda Experiments

Runs	10	100
1	0	0
2	0	1k
3	0	1k
4	0	1k
5	0	3k
6	0	0
7	0	0
8	0	1k
9	0	3k
10	0	0
Avg.	0	1k

Table 14: Table of WaveGAN Number of Discriminator Updates Experiments

Runs	1	2	3	4	5
1	3k	1k	5k	5k	0
2	5k	1k	0	5k	0
3	3k	1k	1k	0	0
4	5k	5k	0	0	0
5	5k	1k	0	0	0
6	3k	2k	1k	0	0
7	4k	1k	1k	0	0
8	5k	5k	0	1k	0
9	4k	1k	5k	0	0
10	3k	1k	0	0	0
Avg.	4k	1.9k	1.3k	1.1k	0

Table 15: Table of Conditional WaveGAN Number of Discriminator Updates Experiments

Runs	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	0	0
3	0	2k	0	0	0
4	0	0	0	0	0
5	2k	0	0	0	0
6	0	0	0	0	0
7	0	0	0	0	0
8	0	0	0	0	0
9	0	0	0	0	0
10	1k	0	0	0	0
Avg.	0.3k	0.2k	0	0	0

F Code

The following pages include the code for the GAN high-level architecture and training functions (Manager.py), code for the neural networks (Networks-WGAN-4096.py, Networks-CWGAN-4096.py, and Networks-ACGAN-4096.py), and other code that was necessary to complete the project (audioDataLoader.py and Downsampler.py).

F.1 Manager.py

```
import argparse as ag
import importlib.machinery as im
import os
import numpy as np
import shutil
import soundfile as sf
import tensorflow as tf
import tfmpl
import types

# Dimension
ABS_INT16 = 32767.
BATCH_SIZE = None
MODEL_SIZE = None
MODES = None
WAV_LENGTH = None
Z_LENGTH = 100

# Learning
BETA1 = 0.5
BETA2 = 0.9
LEARN_RATE = None

# Loss Constants
LAMBDA = None
LOSS_MAX = 400

# Messages
TRAIN_COMPLETE = False

# MinMax
D_UPDATES_PER_G_UPDATES = 1
G_UPDATES_PER_D_UPDATES = 1

# Objects
NETWORKS = None

# Tensor Management
CHECKPOINTS = 5000
ITERATIONS = None
PROGRAM_MODE = None
OUTPUT_DIR = None
SAMPLE_SAVE_RATE = 1000
STEPS = 100


def main(args):
    """ Runs the relevant command passed through arguments """

    global BATCH_SIZE
    BATCH_SIZE = args.batch
```

```
    global ITERATIONS
    ITERATIONS = args.iterations

    global LAMBDA
    LAMBDA = args.lamb

    global MODES
    MODES = len(args.words)

    global WAV_LENGTH
    WAV_LENGTH = args.wave

    global D_UPDATES_PER_G_UPDATES
    global G_UPDATES_PER_D_UPDATES
    D_UPDATES_PER_G_UPDATES = args.D_updates
    G_UPDATES_PER_D_UPDATES = args.G_updates

    global MODEL_SIZE
    if WAV_LENGTH == 1024:
        MODEL_SIZE = 16
    elif WAV_LENGTH == 4096:
        MODEL_SIZE = 32

    global LEARN_RATE
    LEARN_RATE = args.learnRate

    global PROGRAM_MODE
    PROGRAM_MODE = args.mode[0]

    # Train to complete
    if args.mode[0] == "complete":
        global TRAIN_COMPLETE
        TRAIN_COMPLETE = False
        while not TRAIN_COMPLETE:
            tf.reset_default_graph()
            model_dir = _setup(args.runName[0], args.model[0])
            _train(args.words, args.runName[0], model_dir, args.model[0])

    # Training mode
    elif args.mode[0] == "train":
        model_dir = _setup(args.runName[0], args.model[0])
        _train(args.words, args.runName[0], model_dir, args.model[0])

    # Generator mode
    elif args.mode[0] == "gen":
        _generate(
            args.runName[0],
            args.checkpointNum,
            args.genMode,
            args.model[0],
            args.genLength
        )

    return


def _setup(runName, model):
    model_dir = _modelDirectory(runName, model)
    os.makedirs(model_dir)
    _createGenGraph(model_dir, model)
    return model_dir
```

```
def _train(folders, runName, model_dir, model):
    """ Trains the WaveGAN model """

    # Prepare the data
    audio_loader = _loadAudioModule()
    training_data_path = "Data/"
    audio_loader.prepareData(training_data_path, folders)

    # Create generated data
    Z_x, Z_y, Z_yFill, Z_multi = _makeGenerated(True, None)

    # Prepare real data
    X, X_y = audio_loader.loadTrainData()
    epochSize = len(X)

    X = _makeIterators(
        tf.reshape(
            tf.convert_to_tensor(np.vstack(X), dtype=tf.float32),
            [len(X), WAV_LENGTH, 1]
        ),
        X_y,
        len(X),
        WAV_LENGTH
    )

    # Prepare link to the NNs
    global NETWORKS
    NETWORKS = _loadNetworksModule(
        'Networks-' + model + '-' + str(WAV_LENGTH) + '.py',
        'Networks-' + model + '-' + str(WAV_LENGTH) + '.py'
    )

    # Create networks
    if model == 'WGAN':
        with tf.variable_scope('G'):
            G = NETWORKS.generator(Z_x)
        with tf.variable_scope('D'):
            R = NETWORKS.discriminator(X["x"])
        with tf.variable_scope('D', reuse=True):
            F = NETWORKS.discriminator(G)
    elif model == 'CWGAN' or model == 'ACGAN':
        with tf.variable_scope('G'):
            G = NETWORKS.generator(Z_x, Z_y)
        with tf.variable_scope('D'):
            R_cat, R = NETWORKS.discriminator(X["x"], X["yFill"])
        with tf.variable_scope('D', reuse=True):
            F_cat, F = NETWORKS.discriminator(G, Z_yFill)

    # Create variables
    G_variables = tf.get_collection(
        tf.GraphKeys.TRAINABLE_VARIABLES,
        scope='G'
    )
    D_variables = tf.get_collection(
        tf.GraphKeys.TRAINABLE_VARIABLES,
        scope='D'
    )

    # Build loss
    if model == 'WGAN':
```

```
    G_loss, D_loss = _wasser_loss(G, R, F, X)
elif model == 'CWGAN':
    G_loss, D_loss = _alt_conditional_loss(R, F)
    # G_loss, D_loss = _conditioned_wasser_loss(G, R, F, X)
elif model == 'ACGAN':
    G_loss, D_loss = _categorical_loss(R, F, R_cat, F_cat, X["y"], Z_multi)

# Build optimizers
G_opt = tf.train.AdamOptimizer(
    learning_rate=LEARN_RATE,
    beta1=BETA1,
    beta2=BETA2
)
D_opt = tf.train.AdamOptimizer(
    learning_rate=LEARN_RATE,
    beta1=BETA1,
    beta2=BETA2
)

# Build training operations
G_train_op = G_opt.minimize(
    G_loss,
    var_list=G_variables,
    global_step=tf.train.get_or_create_global_step()
)
D_train_op = D_opt.minimize(
    D_loss,
    var_list=D_variables
)

# Root Mean Square
Z_rms = tf.sqrt(tf.reduce_mean(tf.square(G[:, :, 0]), axis=1))
X_rms = tf.sqrt(tf.reduce_mean(tf.square(X["x"][:, :, 0]), axis=1))

# Plot wave
X_plotWave = plotWave(X["x"])
G_plotWave = plotWave(G)

# Plot spectrogram
X_plotSpec = plotSpec(X["x"])
G_plotSpec = plotSpec(G)

# Summary
with tf.name_scope('Audio'):
    tf.summary.audio(
        name='X',
        tensor=X["x"],
        sample_rate=WAV_LENGTH,
        max_outputs=6
    )
    tf.summary.audio(
        name='G',
        tensor=G,
        sample_rate=WAV_LENGTH,
        max_outputs=6
    )

with tf.name_scope('WavePlot'):
    tf.summary.image(
        name='plotWave_X',
        tensor=X_plotWave,
        max_outputs=6
```

```
    )
    tf.summary.image(
        name='plotWave_G',
        tensor=G_plotWave,
        max_outputs=6
    )

with tf.name_scope('plotSpec'):
    tf.summary.image(
        name='plotSpec_X',
        tensor=X_plotSpec,
        max_outputs=6
    )
    tf.summary.image(
        name='plotSpec_G',
        tensor=G_plotSpec,
        max_outputs=6
    )

with tf.name_scope('Rms'):
    tf.summary.histogram('rms_batch_Z', Z_rms)
    tf.summary.histogram('rms_batch_X', X_rms)
    tf.summary.scalar('rms_X', tf.reduce_mean(X_rms))
    tf.summary.scalar('rms_Z', tf.reduce_mean(Z_rms))

with tf.name_scope('Loss'):
    tf.summary.scalar('loss_G', G_loss)
    tf.summary.scalar('loss_D', D_loss)

# Print hyperparameter summary
with open(model_dir + 'hyperparameters.txt', 'w') as f:
    f.write('Batch␣Size␣␣␣␣:␣' + str(BATCH_SIZE) + '\n')
    f.write('Checkpoints␣␣␣:␣' + str(CHECKPOINTS) + '\n')
    f.write('D␣Updates␣␣␣␣␣:␣' + str(D_UPDATES_PER_G_UPDATES) + '\n')
    f.write('Epoch␣Size␣␣␣␣:␣' + str(epochSize) + '\n')
    f.write('G␣Updates␣␣␣␣␣:␣' + str(G_UPDATES_PER_D_UPDATES) + '\n')
    f.write('Iterations␣␣␣␣:␣' + str(ITERATIONS) + '\n')
    f.write('Lambda␣␣␣␣␣␣␣␣:␣' + str(LAMBDA) + '\n')
    f.write('Learning␣Rate␣:␣' + str(LEARN_RATE) + '\n')
    f.write('Model␣Size␣␣␣␣:␣' + str(MODEL_SIZE) + '\n')
    f.write('Model␣Type␣␣␣␣:␣' + model + '\n')
    f.write('Modes␣␣␣␣␣␣␣␣␣:␣' + str(MODES) + '\n')
    f.write('Wave␣length␣␣␣:␣' + str(WAV_LENGTH) + '\n')
    f.close

# Create a session
sess = tf.train.MonitoredTrainingSession(
    checkpoint_dir=model_dir,
    config=tf.ConfigProto(log_device_placement=False),
    save_checkpoint_steps=CHECKPOINTS,
    save_summaries_steps=STEPS
)

# Run the session
_runSession(
    sess,
    D_train_op,
    D_loss,
    G_train_op,
    G_loss,
    G,
    model_dir
```

```
    )

    return


def _runSession(sess, D_train_op, D_loss, G_train_op, G_loss, G, model_dir):
    """ Runs a session """

    runawayLoss = False
    print("Starting␣experiment␣.␣.␣.")

    # Update model ITERATIONS number of times
    for iteration in range(1, ITERATIONS + 1):
        print(iteration)

        # Run Discriminator
        for D_update in range(D_UPDATES_PER_G_UPDATES):
            _, run_D_loss = sess.run([D_train_op, D_loss])
            if abs(run_D_loss) > LOSS_MAX:
                runawayLoss = True
            if runawayLoss:
                break

        # Stop if the Generator loss starts to accelerate
        if runawayLoss:
            print("Ending:␣D␣loss␣=␣" + str(run_D_loss))
            break

        # Run Generator
        for G_update in range(G_UPDATES_PER_D_UPDATES):
            _, run_G_loss, G_data = sess.run([G_train_op, G_loss, G])
            if abs(run_G_loss) > LOSS_MAX:
                runawayLoss = True
            if runawayLoss:
                break

        # Stop if the Generator loss starts to accelerate
        if runawayLoss:
            print("Ending:␣G␣loss␣=␣" + str(run_G_loss))
            break

        # Save samples every SAMPLE_SAVE_RATE steps
        if iteration % SAMPLE_SAVE_RATE == 0:
            print('Completed␣Iteration:␣' + str(iteration))

    sess.close()
    if runawayLoss and not PROGRAM_MODE == "train":
        shutil.rmtree(model_dir)
    elif not runawayLoss:
        global TRAIN_COMPLETE
        TRAIN_COMPLETE = True

    print("Completed␣experiment.")
    return


def _loadNetworksModule(modName, modPath):
    """ Loads the module containing the relevant networks """

    loader = im.SourceFileLoader(modName, modPath)
    mod = types.ModuleType(loader.name)
    loader.exec_module(mod)
```

```
    return mod


def _wasser_loss(G, R, F, X):
    """ Calculates the loss """

    # Cost functions
    G_loss = -tf.reduce_mean(F)
    D_loss = tf.reduce_mean(F) - tf.reduce_mean(R)

    with tf.name_scope('Loss'):
        tf.summary.scalar('loss_D_RAW', D_loss)

    alpha = tf.random_uniform(
        shape=[BATCH_SIZE, 1, 1],
        minval=0.,
        maxval=1.
    )
    differences = G - X["x"]
    interpolates = X["x"] + (alpha * differences)
    with tf.name_scope('D_interp'), tf.variable_scope('D', reuse=True):
        D_interp = NETWORKS.discriminator(interpolates)

    # Gradient penalty
    gradients = tf.gradients(D_interp, [interpolates], name='grads')[0]
    slopes = tf.sqrt(
        tf.reduce_sum(
            tf.square(gradients),
            reduction_indices=[1, 2]
        )
    )
    gradient_penalty = tf.reduce_mean((slopes - 1.) ** 2.)

    # Discriminator loss
    D_loss += LAMBDA * gradient_penalty

    # Summaries
    with tf.name_scope('Various'):
        tf.summary.scalar('norm', tf.norm(gradients))
        tf.summary.scalar('grad_penalty', gradient_penalty)

    return G_loss, D_loss


def _conditioned_wasser_loss(G, R, F, X):
    """ Calculates the loss """

    # Cost functions
    G_loss = tf.reduce_mean(F)
    D_loss = tf.reduce_mean(R) - tf.reduce_mean(F)

    with tf.name_scope('Loss'):
        tf.summary.scalar('loss_D_RAW', D_loss)

    alpha = tf.random_uniform(
        shape=[BATCH_SIZE, 1, 1],
        minval=0.,
        maxval=1.
    )
    x_hat = X["x"] * alpha + (1 - alpha) * G
    with tf.name_scope('D_interp'), tf.variable_scope('D', reuse=True):
```

```
        D_interp = NETWORKS.discriminator(x_hat, X["yFill"])

    # Gradient penalty
    gradients = tf.gradients(D_interp, x_hat, name='grads')[0]
    slopes = tf.sqrt(
        tf.reduce_sum(
            tf.square(gradients),
            axis=[1, 2]
        )
    )
    gradient_penalty = LAMBDA * tf.reduce_mean((slopes - 1.) ** 2.)

    # Discriminator loss
    D_loss += gradient_penalty

    # Summaries
    with tf.name_scope('Various'):
        tf.summary.scalar('norm', tf.norm(gradients))
        tf.summary.scalar('grad_penalty', gradient_penalty)

    return G_loss, D_loss


def _alt_conditional_loss(R, F):
    """ This is an alternative loss function to W-GP """

    D_loss_real = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(
            logits=R,
            labels=tf.ones([BATCH_SIZE, 1])
        )
    )
    D_loss_fake = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(
            logits=F,
            labels=tf.zeros([BATCH_SIZE, 1])
        )
    )
    D_loss = D_loss_real + D_loss_fake
    G_loss = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(
            logits=F,
            labels=tf.ones([BATCH_SIZE, 1])
        )
    )

    return G_loss, D_loss


def _categorical_loss(R, F, R_cat, F_cat, X_y, Z_multi):
    """ A loss for manaing categorical auxiliary values """

    # True / Fake loss
    D_loss_real = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(
            logits=R,
            labels=tf.multiply(
                tf.ones([BATCH_SIZE, 1]),
                tf.random_uniform(
                    shape=[BATCH_SIZE, 1],
                    minval=0.7,
                    maxval=1.2
```

```
                )
            )
        )
    )
    D_loss_fake = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(
            logits=F,
            labels=tf.zeros([BATCH_SIZE, 1])
        )
    )
    D_loss = (D_loss_real + D_loss_fake) / 2
    G_loss = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(
            logits=F,
            labels=tf.ones([BATCH_SIZE, 1])
        )
    )

    # Categorical loss
    D_catLoss_R = tf.reduce_mean(
        tf.nn.softmax_cross_entropy_with_logits_v2(
            logits=R_cat,
            labels=tf.squeeze(X_y)
        )
    )
    D_catLoss_F = tf.reduce_mean(
        tf.nn.softmax_cross_entropy_with_logits_v2(
            logits=F_cat,
            labels=Z_multi
        )
    )
    Cat_loss = (D_catLoss_R + D_catLoss_F) / 2

    G_loss = G_loss + Cat_loss
    D_loss = D_loss + Cat_loss

    return G_loss, D_loss


def _modelDirectory(runName, model):
    """ Creates / obtains the name of the model directory """
    directory = 'tmp/' + model + '_' + str(WAV_LENGTH) + '_' + runName + '/'
    return directory


def _makeGenerated(random, genMode):
    """ Makes the generator tensors """

    if random:
        one_hot = tf.random_uniform(
            [BATCH_SIZE, 1, 1],
            0,
            MODES,
            dtype=tf.int32
        )
    else:
        one_hot = tf.fill(
            [BATCH_SIZE, 1, 1],
            genMode
        )

    Z_x = tf.random_uniform(
```

```
            [BATCH_SIZE, 1, Z_LENGTH],
            -1.,
            1.,
            dtype=tf.float32
        )
        Z_one_hot = tf.one_hot(
            indices=one_hot[:, 0],
            depth=MODES
        )
        Z_y = tf.multiply(
            x=tf.expand_dims(Z_one_hot, axis=1),
            y=tf.ones(
                [BATCH_SIZE, 1, MODEL_SIZE, MODES],
                dtype=tf.float32
            )
        )
        Z_yFill = _makeYFill(BATCH_SIZE, Z_one_hot)

        return Z_x, Z_y, Z_yFill, Z_one_hot


    def _makeIterators(data, labels, data_size, data_length):
        """ Creates iterators for the data """

        oneHot = np.zeros((data_size, 1, MODES), dtype=np.float32)
        oneHot[np.arange(data_size), 0, labels] = 1.0
        oneHotFill = oneHot * \
            np.ones((data_size, WAV_LENGTH, MODES), dtype=np.float32)
        oneHot = tf.convert_to_tensor(oneHot, dtype=tf.float32)
        oneHotFill = _makeYFill(data_size, oneHot)

        dSet = tf.data.Dataset.from_tensor_slices(
            {
                "x": data,
                "y": oneHot,
                "yFill": oneHotFill
            }
        )
        dSet = dSet.shuffle(buffer_size=data_size)
        dSet = dSet.apply(
            tf.contrib.data.batch_and_drop_remainder(BATCH_SIZE)
        )
        dSet = dSet.repeat()

        iterator = dSet.make_one_shot_iterator()
        iterator = iterator.get_next()

        return iterator


    def _makeYFill(size, one_hot):
        """ Makes the Y-Fill tensor """

        yFill = tf.multiply(
            x=one_hot,
            y=tf.ones(
                [size, WAV_LENGTH, MODES],
                dtype=tf.float32
            )
        )

        return yFill
```

```
def _altMakeYFill(size, one_hot):
    """ Makes the Y-Fill tensor """

    divisor = 64

    # Result (example): [64, 256, 2]
    yFill = tf.multiply(
        x=one_hot,
        y=tf.ones(
            [size, WAV_LENGTH / divisor, MODES],
            dtype=tf.float32
        )
    )

    # Input: [64, 256, 2] > [64, 1024, 2]
    yFill = tf.layers.conv2d_transpose(
        inputs=tf.expand_dims(yFill, axis=1),
        filters=MODES,
        kernel_size=(1, 16),
        strides=(1, divisor),
        padding='same'
    )[:, 0]

    return yFill


def _loadAudioModule():
    """ Loads the audio module & returns it as objects """
    audio_loader = _loadNetworksModule(
        'audioDataLoader.py',
        'audioDataLoader.py'
    )
    return audio_loader


def _createConfigFile():
    """ Creates a config file for the training session """
    return


def _createGenGraph(model_dir, model):
    """ Creates a copy of the generator graph """

    # Prepare link to the NNs
    global NETWORKS
    NETWORKS = _loadNetworksModule(
        'Networks-' + model + '-' + str(WAV_LENGTH) + '.py',
        'Networks-' + model + '-' + str(WAV_LENGTH) + '.py'
    )

    # Create directory
    graphDir = os.path.join(model_dir + 'Generator/')
    if not os.path.isdir(graphDir):
        os.makedirs(graphDir)

    # Create graph
    Z_Input = tf.placeholder(tf.float32, [None, 1, Z_LENGTH], name='Z_Input')
    Z_Labels = tf.placeholder(
        tf.float32, [None, 1, MODEL_SIZE, MODES], name='Z_Labels')
```

```
    if model == 'WGAN':
        with tf.variable_scope('G'):
            G = NETWORKS.generator(Z_Input)
        G = tf.identity(G, name='Generator')
    elif model == 'CWGAN' or model == 'ACGAN':
        with tf.variable_scope('G'):
            G = NETWORKS.generator(Z_Input, Z_Labels)
        G = tf.identity(G, name='Generator')

    # Save graph
    G_variables = tf.get_collection(
        tf.GraphKeys.TRAINABLE_VARIABLES,
        scope='G'
    )
    global_step = tf.train.get_or_create_global_step()
    saver = tf.train.Saver(G_variables + [global_step])

    # Export All
    tf.train.write_graph(
        tf.get_default_graph(),
        graphDir,
        'generator.pbtxt'
    )
    tf.train.export_meta_graph(
        filename=os.path.join(graphDir, 'generator.meta'),
        clear_devices=True,
        saver_def=saver.as_saver_def()
    )
    tf.reset_default_graph()

    return


def _generate(runName, checkpointNum, genMode, model, genLength):
    """ Generates samples from the generator """

    # Load the graph
    model_dir = _modelDirectory(runName, model)
    tf.reset_default_graph()
    graph = tf.get_default_graph()
    sess = tf.InteractiveSession()
    tf.train.import_meta_graph(
        model_dir + 'Generator/generator.meta'
    ).restore(
        sess,
        model_dir + 'model.ckpt-' + str(checkpointNum)
    )

    # Generate sounds
    Z = np.random.uniform(-1., 1., [genLength, 1, Z_LENGTH])

    # Get tensors
    Z_input = graph.get_tensor_by_name('Z_Input:0')
    G = graph.get_tensor_by_name('Generator:0')

    # Enter into graph
    if model == 'WGAN':
        samples = sess.run(G, {Z_input: Z})
    elif model == 'CWGAN' or model == 'ACGAN':
        Z_y = np.zeros(
            shape=(genLength, 1, MODEL_SIZE, MODES),
            dtype=np.int16
```

```
        )
        Z_y[:, :, genMode] = 1
        Z_labels = graph.get_tensor_by_name('Z_Labels:0')
        samples = sess.run(G, {Z_input: Z, Z_labels: Z_y})

    # Create the output path
    path = os.path.abspath(
        os.path.join(
            os.path.dirname((__file__)),
            os.pardir,
            'Generated/',
            model + '_' + str(WAV_LENGTH) + '/',
            'ModelRun_' + str(runName)
        )
    )

    if model == 'WGAN':
        fileName = 'Random'
    elif model == 'CWGAN' or model == 'ACGAN':
        fileName = 'Mode_' + str(genMode)

    # Write samples to file
    _saveGenerated(path, samples, fileName)

    return


def _saveGenerated(path, samples, fileName):
    """ Saves the generated samples to folder as .wav """

    if not os.path.exists(path):
        os.makedirs(path)

    # Save the samples
    i = 0
    for sample in samples:
        i = i + 1
        sf.write(
            file=path + '/' + fileName + '_' + str(i) + '.wav',
            data=sample,
            samplerate=WAV_LENGTH,
            subtype='PCM_16'
        )

    return


@tfmpl.figure_tensor
def plotWave(audioTensor):
    """ Plots audio to a wave graph """
    figs = tfmpl.create_figures(BATCH_SIZE)
    for i, f in enumerate(figs):
        sp = f.add_subplot(111)
        sp.plot(audioTensor[i, :, :])

    return figs


@tfmpl.figure_tensor
def plotSpec(audioTensor):
    """ Plots audio to a wave graph """
    figs = tfmpl.create_figures(BATCH_SIZE)
```

```
    for i, f in enumerate(figs):
        sp = f.add_subplot(111)
        dim = audioTensor[i, :, :]
        dim = np.squeeze(dim)
        sp.specgram(
            x=dim,
            NFFT=256,
            Fs=2
        )

    return figs


if __name__ == "__main__":
    parser = ag.ArgumentParser()
    parser.add_argument(
        '-mode',
        nargs=1,
        type=str,
        default='train',
        help="How you wish to use the model."
    )
    parser.add_argument(
        '-model',
        nargs=1,
        type=str,
        help="Which generator type do you want to use?"
    )
    parser.add_argument(
        '-wave',
        type=int,
        default=4096,
        help="The wave length of files for this experiment"
    )
    parser.add_argument(
        '-runName',
        nargs=1,
        type=str,
        help="A name for this run of the experiment."
    )
    parser.add_argument(
        '-checkpointNum',
        type=int,
        help="The checkpoint number you wish to examine."
    )
    parser.add_argument(
        '-genMode',
        type=int,
        default=0,
        help="The number of the mode to be generated."
    )
    parser.add_argument(
        '-genLength',
        type=int,
        default=100,
        help="The count of samples you want generated."
    )
    parser.add_argument(
        '-lamb',
        type=float,
        default=10,
        help="The lambda to be applied to Wasserstein Loss."
```

```
)
parser.add_argument(
    '-batch',
    type=int,
    default=64,
    help="The batch size."
)
parser.add_argument(
    '-iterations',
    type=int,
    default=1000,
    help="The number of times you want the model to run."
)
parser.add_argument(
    '-D_updates',
    type=int,
    default=1,
    help="The number of discriminator updates to generator."
)
parser.add_argument(
    '-G_updates',
    type=int,
    default=1,
    help="The number of generator updates to discriminator."
)
parser.add_argument(
    '-learnRate',
    type=float,
    default=0.0001,
    help="The learning rate used."
)
parser.add_argument(
    '-words',
    nargs='*',
    type=str,
    default=['zero', 'one'],
    help="The words for sounds you want to train with."
)
main(parser.parse_args())
```

F.2 Networks-WGAN-4096.py

```
import random as rd
import tensorflow as tf


BATCH_SIZE = -1
CHANNELS = 1
KERNEL_SIZE = 25
MODEL_SIZE = 32
PHASE_SHUFFLE = 2
STRIDE = 4
WAV_LENGTH = 4096
Z_LENGTH = 100


def generator(z):
    """ A waveGAN generator """

    z = tf.cast(z, tf.float32)

    # Input: [64, 100] > [64, 4096]
    densify = tf.layers.dense(
        inputs=z,
        units=WAV_LENGTH,
        name="Z-Input"
    )

    # Input: [64, 4096] > [64, 16, 256]
    shape = tf.reshape(
        tensor=densify,
        shape=[BATCH_SIZE, 16, MODEL_SIZE * 8]
    )

    layer = tf.nn.relu(shape)

    # Input: [64, 16, 256] > [64, 64, 128]
    layer = tf.layers.conv2d_transpose(
        inputs=tf.expand_dims(layer, axis=1),
        filters=MODEL_SIZE * 4,
        kernel_size=(1, KERNEL_SIZE),
        strides=(1, STRIDE),
        padding='SAME',
        name="TransConvolution1"
    )[:, 0]

    layer = tf.nn.relu(layer)

    # Input: [64, 64, 128] > [64, 256, 64]
    layer = tf.layers.conv2d_transpose(
        inputs=tf.expand_dims(layer, axis=1),
        filters=MODEL_SIZE * 2,
        kernel_size=(1, KERNEL_SIZE),
        strides=(1, STRIDE),
        padding='SAME',
        name="TransConvolution2"
    )[:, 0]

    layer = tf.nn.relu(layer)

    # Input: [64, 256, 64] > [64, 1024, 32]
    layer = tf.layers.conv2d_transpose(
```

```
        inputs=tf.expand_dims(layer, axis=1),
        filters=MODEL_SIZE,
        kernel_size=(1, KERNEL_SIZE),
        strides=(1, STRIDE),
        padding='SAME',
        name="TransConvolution3"
    )[:, 0]

    layer = tf.nn.relu(layer)

    # Input: [64, 1024, 32] > [64, 4096, 1]
    layer = tf.layers.conv2d_transpose(
        inputs=tf.expand_dims(layer, axis=1),
        filters=CHANNELS,
        kernel_size=(1, KERNEL_SIZE),
        strides=(1, STRIDE),
        padding='SAME',
        name="TransConvolution4"
    )[:, 0]

    # Input: [64, 4096, 1]
    tanh = tf.tanh(
        x=layer,
        name="GeneratedSamples"
    )

    return tanh


def discriminator(features):
    """ A waveGAN discriminator """

    # Input: [64, 4096, 1] > [64, 1024, 32]
    layer = tf.layers.conv1d(
        inputs=features,
        filters=MODEL_SIZE,
        kernel_size=KERNEL_SIZE,
        strides=STRIDE,
        padding='same'
    )
    layer = _leakyRelu(layer)
    layer = _phaseShuffle(layer)

    # Input: [64, 1024, 32] > [64, 256, 64]
    layer = tf.layers.conv1d(
        inputs=layer,
        filters=MODEL_SIZE * 2,
        kernel_size=KERNEL_SIZE,
        strides=STRIDE,
        padding='same'
    )
    layer = _leakyRelu(layer)
    layer = _phaseShuffle(layer)

    # Input: [64, 256, 64] > [64, 64, 128]
    layer = tf.layers.conv1d(
        inputs=layer,
        filters=MODEL_SIZE * 4,
        kernel_size=KERNEL_SIZE,
        strides=STRIDE,
        padding='same'
    )
```

```
    layer = _leakyRelu(layer)
    layer = _phaseShuffle(layer)

    # Input: [64, 64, 128] > [64, 16, 256]
    layer = tf.layers.conv1d(
        inputs=layer,
        filters=MODEL_SIZE * 8,
        kernel_size=KERNEL_SIZE,
        strides=STRIDE,
        padding='same'
    )
    layer = _leakyRelu(layer)

    # Input: [64, 16, 256] > [64, 4096]
    flatten = tf.reshape(
        tensor=layer,
        shape=[BATCH_SIZE, WAV_LENGTH]
    )

    # Input: [64, 4096] > [64, 1]
    logits = tf.layers.dense(
        inputs=flatten,
        units=1
    )[:, 0]

    return logits


def _phaseShuffle(layer):
    """ Shuffles the phase of each layer """
    batch, length, channel = layer.get_shape().as_list()
    shuffle = _returnPhaseShuffleValue()
    lft = max(0, shuffle)
    rgt = max(0, -shuffle)
    layer = tf.pad(
        tensor=layer,
        paddings=[[0, 0], [lft, rgt], [0, 0]],
        mode='REFLECT'
    )
    layer = layer[:, rgt:rgt+length]
    layer.set_shape([batch, length, channel])
    return layer


def _leakyRelu(inputs, alpha=0.2):
    """ Creates a leaky relu layer """
    return tf.maximum(inputs * alpha, inputs)


def _returnPhaseShuffleValue():
    """ Returns a a ranom integer in the range decided for phase shuffle"""
    return rd.randint(-PHASE_SHUFFLE, PHASE_SHUFFLE)
```

F.3 Networks-CWGAN-4096.py

```
import random as rd
import tensorflow as tf


BATCH_SIZE = -1
CHANNELS = 1
CLASSES = 2
KERNEL_SIZE = 25
MODEL_SIZE = 32
PHASE_SHUFFLE = 2
STRIDE = 4
WAV_LENGTH = 4096
Z_LENGTH = 100


def generator(x, y):
    """ A waveGAN generator """

    x = tf.cast(x, tf.float32)
    y = tf.cast(y, tf.float32)

    # Input: [64, 100] > [64, 4094]
    densify = tf.layers.dense(
        inputs=x,
        units=WAV_LENGTH - CLASSES,
        name="Z-Input"
    )

    # Input: [64, 4032] > [64, 16, 254]
    shape = tf.reshape(
        tensor=densify,
        shape=[BATCH_SIZE, 1, 1, WAV_LENGTH - CLASSES]
    )

    y = y[:, :, 0:1, :]

    # Input: [64, 1, 1, 4094] > [64, 1, 1, 4096]
    concat = tf.concat(values=[shape, y], axis=3)

    layer = tf.nn.relu(concat)

    # Input: [64, 1, 1, 4096] > [64, 1, 16, 256]
    layer = tf.layers.conv2d_transpose(
        inputs=layer,
        filters=MODEL_SIZE * 8,
        kernel_size=(1, 16),
        strides=(1, 1),
        padding='valid',
        name="TransConvolution0"
    )

    # Input: [64, 1, 16, 256] > [64, 1, 64, 128]
    layer = tf.layers.conv2d_transpose(
        inputs=layer,
        filters=MODEL_SIZE * 4,
        kernel_size=(1, KERNEL_SIZE),
        strides=(1, STRIDE),
        padding='same',
        name="TransConvolution1"
    )
```

```
    layer = tf.nn.relu(layer)

    # Input: [64, 1, 64, 128] > [64, 1, 256, 64]
    layer = tf.layers.conv2d_transpose(
        inputs=layer,
        filters=MODEL_SIZE * 2,
        kernel_size=(1, KERNEL_SIZE),
        strides=(1, STRIDE),
        padding='same',
        name="TransConvolution2"
    )

    layer = tf.nn.relu(layer)

    # Input: [64, 1, 256, 64] > [64, 1024, 32]
    layer = tf.layers.conv2d_transpose(
        inputs=layer,
        filters=MODEL_SIZE,
        kernel_size=(1, KERNEL_SIZE),
        strides=(1, STRIDE),
        padding='same',
        name="TransConvolution3"
    )

    layer = tf.nn.relu(layer)

    # Input: [64, 1, 1024, 32] > [64, 4096, 1]
    layer = tf.layers.conv2d_transpose(
        inputs=layer,
        filters=CHANNELS,
        kernel_size=(1, KERNEL_SIZE),
        strides=(1, STRIDE),
        padding='same',
        name="TransConvolution4"
    )[:, 0]

    # Input: [64, 4096, 1]
    tanh = tf.tanh(
        x=layer,
        name="GeneratedSamples"
    )

    return tanh


def discriminator(x, y):
    """ A waveGAN discriminator """

    x = tf.concat(values=[x, y], axis=2)

    # Input: [64, 4096, 1] > [64, 1024, 32]
    layer = tf.layers.conv1d(
        inputs=x,
        filters=MODEL_SIZE,
        kernel_size=KERNEL_SIZE,
        strides=STRIDE,
        padding='same'
    )
    layer = _leakyRelu(layer)
    layer = _phaseShuffle(layer)
```

```
    # Input: [64, 1024, 32] > [64, 256, 64]
    layer = tf.layers.conv1d(
        inputs=layer,
        filters=MODEL_SIZE * 2,
        kernel_size=KERNEL_SIZE,
        strides=STRIDE,
        padding='same'
    )
    layer = _leakyRelu(layer)
    layer = _phaseShuffle(layer)

    # Input: [64, 256, 64] > [64, 64, 128]
    layer = tf.layers.conv1d(
        inputs=layer,
        filters=MODEL_SIZE * 4,
        kernel_size=KERNEL_SIZE,
        strides=STRIDE,
        padding='same'
    )
    layer = _leakyRelu(layer)
    layer = _phaseShuffle(layer)

    # Input: [64, 64, 128] > [64, 16, 256]
    layer = tf.layers.conv1d(
        inputs=layer,
        filters=MODEL_SIZE * 8,
        kernel_size=KERNEL_SIZE,
        strides=STRIDE,
        padding='same'
    )

    # Input: [64, 16, 256] > [64, 1, 1]
    disc = tf.layers.conv1d(
        inputs=layer,
        filters=1,
        kernel_size=16,
        strides=1,
        padding='valid'
    )[:, 0]

    return disc, disc


def _phaseShuffle(layer):
    """ Shuffles the phase of each layer """
    batch, length, channel = layer.get_shape().as_list()
    shuffle = _returnPhaseShuffleValue()
    lft = max(0, shuffle)
    rgt = max(0, -shuffle)
    layer = tf.pad(
        tensor=layer,
        paddings=[[0, 0], [lft, rgt], [0, 0]],
        mode='REFLECT'
    )
    layer = layer[:, rgt:rgt+length]
    layer.set_shape([batch, length, channel])
    return layer


def _leakyRelu(inputs, alpha=0.2):
    """ Creates a leaky relu layer """
    return tf.maximum(inputs * alpha, inputs)
```

```
def _returnPhaseShuffleValue():
    """ Returns a a ranom integer in the range decided for phase shuffle"""
    return rd.randint(-PHASE_SHUFFLE, PHASE_SHUFFLE)
```

F.4 Networks-ACGAN-4096.py

```
import random as rd
import tensorflow as tf


BATCH_SIZE = -1
CHANNELS = 1
CLASSES = 2
KERNEL_SIZE = 25
MODEL_SIZE = 32
PHASE_SHUFFLE = 2
STRIDE = 4
WAV_LENGTH = 4096
Z_LENGTH = 100


def generator(x, y):
    """ A waveGAN generator """

    x = tf.cast(x, tf.float32)
    y = tf.cast(y, tf.float32)

    # Input: [64, 100] > [64, 4094]
    densify = tf.layers.dense(
        inputs=x,
        units=WAV_LENGTH - CLASSES,
        name="Z-Input"
    )

    # Input: [64, 4032] > [64, 16, 254]
    shape = tf.reshape(
        tensor=densify,
        shape=[BATCH_SIZE, 1, 1, WAV_LENGTH - CLASSES]
    )

    y = y[:, :, 0:1, :]

    # Input: [64, 1, 1, 4094] > [64, 1, 1, 4096]
    concat = tf.concat(values=[shape, y], axis=3)

    layer = tf.nn.relu(concat)

    # Input: [64, 1, 1, 4096] > [64, 1, 16, 256]
    layer = tf.layers.conv2d_transpose(
        inputs=layer,
        filters=MODEL_SIZE * 8,
        kernel_size=(1, 16),
        strides=(1, 1),
        padding='valid',
        name="TransConvolution0"
    )

    # Input: [64, 1, 16, 256] > [64, 1, 64, 128]
    layer = tf.layers.conv2d_transpose(
        inputs=layer,
        filters=MODEL_SIZE * 4,
        kernel_size=(1, KERNEL_SIZE),
        strides=(1, STRIDE),
        padding='same',
        name="TransConvolution1"
    )
```

```
    layer = tf.nn.relu(layer)

    # Input: [64, 1, 64, 128] > [64, 1, 256, 64]
    layer = tf.layers.conv2d_transpose(
        inputs=layer,
        filters=MODEL_SIZE * 2,
        kernel_size=(1, KERNEL_SIZE),
        strides=(1, STRIDE),
        padding='same',
        name="TransConvolution2"
    )

    layer = tf.nn.relu(layer)

    # Input: [64, 1, 256, 64] > [64, 1024, 32]
    layer = tf.layers.conv2d_transpose(
        inputs=layer,
        filters=MODEL_SIZE,
        kernel_size=(1, KERNEL_SIZE),
        strides=(1, STRIDE),
        padding='same',
        name="TransConvolution3"
    )

    layer = tf.nn.relu(layer)

    # Input: [64, 1, 1024, 32] > [64, 4096, 1]
    layer = tf.layers.conv2d_transpose(
        inputs=layer,
        filters=CHANNELS,
        kernel_size=(1, KERNEL_SIZE),
        strides=(1, STRIDE),
        padding='same',
        name="TransConvolution4"
    )[:, 0]

    # Input: [64, 4096, 1]
    tanh = tf.tanh(
        x=layer,
        name="GeneratedSamples"
    )

    return tanh


def discriminator(x, y):
    """ A waveGAN discriminator """

    x = tf.concat(values=[x, y], axis=2)

    # Input: [64, 4096, 1] > [64, 1024, 32]
    layer = tf.layers.conv1d(
        inputs=x,
        filters=MODEL_SIZE,
        kernel_size=KERNEL_SIZE,
        strides=STRIDE,
        padding='same'
    )
    layer = _leakyRelu(layer)
    layer = _phaseShuffle(layer)
```

```
    # Input: [64, 1024, 32] > [64, 256, 64]
    layer = tf.layers.conv1d(
        inputs=layer,
        filters=MODEL_SIZE * 2,
        kernel_size=KERNEL_SIZE,
        strides=STRIDE,
        padding='same'
    )
    layer = _leakyRelu(layer)
    layer = _phaseShuffle(layer)

    # Input: [64, 256, 64] > [64, 64, 128]
    layer = tf.layers.conv1d(
        inputs=layer,
        filters=MODEL_SIZE * 4,
        kernel_size=KERNEL_SIZE,
        strides=STRIDE,
        padding='same'
    )
    layer = _leakyRelu(layer)
    layer = _phaseShuffle(layer)

    # Input: [64, 64, 128] > [64, 16, 256]
    layer = tf.layers.conv1d(
        inputs=layer,
        filters=MODEL_SIZE * 8,
        kernel_size=KERNEL_SIZE,
        strides=STRIDE,
        padding='same'
    )

    # Input: [64, 16, 256] > [64, 1, 1]
    disc = tf.layers.conv1d(
        inputs=layer,
        filters=1,
        kernel_size=16,
        strides=1,
        padding='valid'
    )[:, 0]

    return disc, disc


def _phaseShuffle(layer):
    """ Shuffles the phase of each layer """
    batch, length, channel = layer.get_shape().as_list()
    shuffle = _returnPhaseShuffleValue()
    lft = max(0, shuffle)
    rgt = max(0, -shuffle)
    layer = tf.pad(
        tensor=layer,
        paddings=[[0, 0], [lft, rgt], [0, 0]],
        mode='REFLECT'
    )
    layer = layer[:, rgt:rgt+length]
    layer.set_shape([batch, length, channel])
    return layer


def _leakyRelu(inputs, alpha=0.2):
    """ Creates a leaky relu layer """
    return tf.maximum(inputs * alpha, inputs)
```

```
def _returnPhaseShuffleValue():
    """ Returns a a ranom integer in the range decided for phase shuffle"""
    return rd.randint(-PHASE_SHUFFLE, PHASE_SHUFFLE)
```

F.5 audioDataLoader.py

```
import hashlib as hl
import librosa as lb
import os
import re


MAX_NUM_WAVS_PER_CLASS = 2**27 - 1

LOOKUP = None

ALL_DATA = None
ALL_LABELS = None

TEST_DATA = None
TEST_LABELS = None

TRAIN_DATA = None
TRAIN_LABELS = None

VALID_DATA = None
VALID_LABELS = None

TEST_PER = 20
VALID_PER = 5


def prepareData(inFilePath, folderNames):
    """ Loads the audio data & labels into list form """
    _resetGlobalVariables()
    label = 0
    for folder in folderNames:
        files = lb.util.find_files(
            inFilePath + '/' + folder + '/',
            ext='wav'
        )
        _appendInfo(files, label)
        LOOKUP.append((label, folder))
        label = label + 1
    return


def loadAllData():
    """ Returns all of the data """
    ALL_DATA = TRAIN_DATA + TEST_DATA + VALID_DATA
    ALL_LABELS = TRAIN_LABELS + TEST_LABELS + VALID_LABELS
    return ALL_DATA, ALL_LABELS


def loadTrainData():
    """ Returns the training data only """
    return TRAIN_DATA, TRAIN_LABELS


def loadTestData():
    """ Returns the evaluation data only """
    return TEST_DATA, TEST_LABELS


def loadValidData():
    """ Returns the validation data only """
```

```
    return VALID_DATA, VALID_LABELS


def getLookup():
    """ Returns the lookup data for the categories """
    return LOOKUP


def _resetGlobalVariables():
    """ Resets the variables if the module has already been used """
    global ALL_DATA
    global ALL_LABELS
    global TEST_DATA
    global TEST_LABELS
    global TRAIN_DATA
    global TRAIN_LABELS
    global VALID_DATA
    global VALID_LABELS
    global LOOKUP
    ALL_DATA = []
    ALL_LABELS = []
    TEST_DATA = []
    TEST_LABELS = []
    TRAIN_DATA = []
    TRAIN_LABELS = []
    VALID_DATA = []
    VALID_LABELS = []
    LOOKUP = []
    return


def _appendInfo(files, label):
    """ Appends file data series & label to the lists """

    for eachFile in files:
        series, sampRate = lb.core.load(eachFile, sr=None)
        path, fileName = os.path.split(eachFile)
        hashPercent = _getPercHash(fileName)
        if hashPercent < VALID_PER:
            VALID_DATA.append(series)
            VALID_LABELS.append(label)
        elif hashPercent < (TEST_PER + VALID_PER):
            TEST_DATA.append(series)
            TEST_LABELS.append(label)
        else:
            TRAIN_DATA.append(series)
            TRAIN_LABELS.append(label)

    return


def _getPercHash(name):
    """ Calculates & returns the percentage from the hash """

    hash_name = re.sub('_nohash_.*$', '', name)
    hash_name_encode = hash_name.encode('utf-8')
    hash_name_hashed = hl.sha1(hash_name_encode).hexdigest()
    percent_hash = (
        (int(hash_name_hashed, 16) % (MAX_NUM_WAVS_PER_CLASS + 1))
        * (100.0 / MAX_NUM_WAVS_PER_CLASS))

    return percent_hash
```

F.6 Downsampler.py

```
import argparse as ag
import librosa as lb
import numpy as np
import os
import soundfile as sf


# Constants
IN_PATH = None
OUT_PATH = None
WAV_LENGTH = 16384
SAMP_RATE = None


def main(args):
    """ Runs the code """
    global SAMP_RATE
    SAMP_RATE = int(WAV_LENGTH / args.divider[0])
    global IN_PATH
    IN_PATH = args.inPath[0]
    global OUT_PATH
    OUT_PATH = args.outPath[0]
    _loopFolders(args.words)
    print(IN_PATH)
    print(OUT_PATH)
    return


def _loopFolders(folders):
    """ Loops through all folders found at the IN_PATH """
    for folder in folders:
        path = IN_PATH + folder + '/'
        _loopFiles(path, folder)
    return


def _loopFiles(path, folder):
    """ Loop through all files found within the folder  """
    allFiles = lb.util.find_files(path, ext='wav')
    for eachFile in allFiles:
        filePath, fileName = os.path.split(eachFile)
        resampled = _resampleFile(eachFile)
        _saveFile(folder, resampled, fileName)
    return


def _resampleFile(wav):
    """ Resample the wav file passed to the function """
    series, sampRate = lb.core.load(wav, sr=None)
    newSeries = _standardizeLength(series)
    resampled = lb.core.resample(
        y=newSeries,
        orig_sr=WAV_LENGTH,
        target_sr=SAMP_RATE)
    return resampled


def _standardizeLength(series):
    """ Standardizes the length of the wav before resampling """
    if len(series) < WAV_LENGTH:
```

```
        series = np.append(series, np.zeros(WAV_LENGTH - len(series)))
    elif len(series) > WAV_LENGTH:
        series = series[:WAV_LENGTH]
    return series


def _saveFile(folder, resampled, fileName):
    """ Save the file """
    path = OUT_PATH + str(SAMP_RATE) + '/' + folder + '/'
    if not os.path.exists(path):
        os.makedirs(path)
    sf.write(
        file=path + fileName,
        data=resampled,
        samplerate=SAMP_RATE,
        subtype='PCM_16')
    return


if __name__ == "__main__":
    parser = ag.ArgumentParser(description="Pass␣a␣downsample␣divider:")
    parser.add_argument(
        dest='divider',
        nargs=1,
        type=int,
        choices=[1, 4, 16],
        help="A␣divider␣for␣downsampling."
    )
    parser.add_argument(
        dest='inPath',
        nargs=1,
        type=str,
        help="Path␣from␣which␣data␣will␣be␣downsampled."
    )
    parser.add_argument(
        dest='outPath',
        nargs=1,
        type=str,
        help="Path␣into␣which␣data␣will␣be␣downsampled."
    )
    parser.add_argument(
        dest='words',
        nargs='*',
        type=str,
        default=['zero', 'one'],
        help="Names␣of␣words␣(and␣folders)␣to␣be␣resampled."
    )
    main(parser.parse_args())
```