Applications of Machine Learning to Audio Synthesis and Production

Aoife McDonagh

College of Engineering and Informatics

National University of Ireland, Galway

This dissertation is submitted for the degree of

Master of Engineering Science

Supervisor: Prof. Peter Corcoran

December 2019

Abstract

In recent years, the field of machine learning has made large contributions to advances in consumer device technology. Computer vision and speech recognition account for a large part of these contributions. Examples include facial recognition in smartphone applications and virtual assistant software.

This thesis focuses on machine learning methods for audio synthesis and production. High quality, realistic audio is essential for creating an immersive experience, especially in gaming devices. Virtual and mixed reality devices are unique in that they must not only react to user input but also to ambient environment conditions. The methods described in this thesis contribute to knowledge of audio synthesis in relation to both the user and to their environment.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 45,000 words including appendices, bibliography, footnotes, tables and equations.

Aoife McDonagh

December 2019

Acknowledgements

Thank you Mammy, Dad and Roisin for your encouraging words.
Thank you to all my friends and housemates for your listening ears.
Thank you Peter, Chris and Claudia for your wise advice.
Thank you to all my friends at Xperi for the runs, foosball, chats and knowledge shared over the last two years.

I would also like to acknowledge the Irish Research Council and Xperi Ireland for their funding and support throughout my Masters degree. This work was supported by Irish Research Council Employment Based Programme Award EBP/2017/476.

Contents

List of Figures

List of Tables

Acronyms

ALU Arithmetic Logic Unit. 28

ANN Artificial Neural Network. 22, 23

CNN Convolutional Neural Network. 14, 21, 22, 25, 32–35, 47, 56, 57, 63, 64, 89–93, 108, 127

CPU Central Processing Unit. 27, 63

CRF Conditional Random Field. 56, 57, 89

CSV Comma Separated Variable. 70, 71

D Discriminator. 79, 155

DL Deep Learning. 14, 21, 22, 26, 127

DNN Deep Neural Network. 15, 16, 26, 30, 36, 58, 64, 114

DRAM Dynamic Random-Access Memory. 58

FFT Fast Fourier Transform. 48

FPGA Field Programmable Gate Array. 61

G Generator. 79, 155

GAN Generative Adversarial Network. 15, 19–22, 38–42, 46, 48, 64, 66, 67, 87, 127, 129, 130

GPIO General Purpose Input Output. 110, 112

GPU Graphics Processing Unit. 27, 28, 47, 63

GUI Graphical User Interface. 116, 122, 123

HSV Hue Saturation Value. 103, 123

IE Inference Engine. 115, 116

IoT Internet of Things. 59

IR Intermediate Representation. 116, 117

LSTM Long Short-Term Memory. 129

MINC Materials in Context Database. 89, 96, 100, 106, 108, 127, 130

ML Machine Learning. 22, 26, 128

MO Model Optimizer. 115–117

MR Mixed Reality. 15–18, 55, 56, 88, 89, 108, 126, 128

NCS Neural Compute Stick. 8, 9, 109, 112, 114–118, 121, 122, 126

Pi Raspberry Pi. 110, 112, 115

ReLU Rectified Linear Unit. 36

RGB Red Green Blue. 123

RIR Room Impulse Response. 88

SRAM Static Random-Access Memory. 58

TPU Tensor Processing Unit. 27

VPU Vision Processing Unit. 63, 64, 114

Chapter 1

Introduction

1.1 Background and Objectives

This thesis is an investigation into the use of Deep Learning (DL) methods for audio applications. The large majority of progress in the field of Deep Learning has been initiated by works in imaging applications. A classic example is the ImageNet challenge [1], started in 2010 with the goal of allowing researchers to compare object detection algorithms. A 2012 entrant achieved breakthrough accuracy in the competition with a deep neural network. A Convolutional Neural Network (CNN) model named "AlexNet" achieved a top-5 error of 15.3% in the large scale object recognition task [2]. This error rate was more than 10 percentage points lower than that achieved by the second place winner.

Since then, Deep Learning based approaches to image classification problems have achieved state-of-the-art results across many disciplines. Deep Learning based approaches to audio classification and synthesis tasks have been far fewer in number and received much less fanfare in comparison.

Chapter 2 gives a literature review of topics relevant to the work in this thesis. This includes an overview of the foundations of Deep Learning research, various

Deep Neural Network (DNN) architectures, the training of neural networks, and a review of audio synthesis techniques.

The three projects in this thesis relate to audio production or synthesis. The motivation for carrying out these works stemmed both from personal interest and the strategic goals of my industry partner, Xperi. Specifically, all three works share the following two goals of;

1. Improving of *realism* of artificial sounds.

2. Contributing to listener *immersion* in multi-media experiences.

The project in Chapter 3 is an attempt to directly synthesise audio waveforms which exhibit characteristics of multiple target classes. A synthesis model known as a Generative Adversarial Network (GAN) is used. This technique has potential to be utilised in the production of realistic fictional object sounds.

This project also contributes to listener immersion as a method of generating unique sound effects. Sound samples are generated by a Generative Adversarial Network trained on samples from multiple classes. This has the effect of "mixing" audio classes. In some applications, targeted mixing of samples is beneficial in creating a unique result. Fantasy movies or video games which contain fictional objects may need to be sonified in a way which is convincing to the listener. Using sound samples from real-world objects may break the immersive experience if the user is not convinced of their origin from the fictional object.

Chapter 4 describes a method for estimating materials present in a space based on a still image. Material is an important factor in room acoustics, and one which is usually known to sound engineers working in the production of audio. However in the case of Mixed Reality (MR), the characteristics of a user's environment are not always known beforehand. An image-based method of estimating room materials could allow Mixed Reality device designers to better produce spatial audio.

In this case, audio which matches the acoustics of the surrounding environment. Therefore, the work in this chapter aims to improve audio production by increasing its spatial realism.

The technique of spatial audio is used in gaming and films to immerse a viewer/listener into a scene. Spatial audio aims to produce audio consistent with the physical scene as well as the locations of any sound sources within the scene. It would be unappealing to watch a movie where all sounds were heard to be coming from the same place. It is far more captivating to hear sound sources moving around a scene. Imagine the roar of a plane crossing low in the sky above you, or the hollow clap of a book falling in a cathedral to get a sense of spatial audio's importance.

The method in Chapter 4 is designed to assist in the production of audio which is consistent with a physical environment, thereby increasing listener immersion. Based on images of a scene, material information is estimated by a Deep Neural Network based method of image segmentation. This information could be used to estimate the acoustic properties of the scene. Such a technique is particularly useful in a Mixed Reality use case. When audio from a virtual source is played in a real space, it needs to sound as if it is really present in that space, rather than "inside" the listeners head (i.e. internalised).

In Chapter 5, the final work of this thesis, an embedded systems based implementation of the work in Chapter 4 is described. Most Mixed Reality devices are in the form of headsets which severely limits power supply, computing resources and physical dimensions. Therefore, it would be important to Mixed Reality device designers using such a material estimation technique that it runs as smoothly as possible with limited hardware resources.

1.2 Commercial applications

There are various commercial applications for the work put forward in this thesis. Firstly, the topic of neural audio synthesis is relatively little explored in a commercial context. There are examples of neural generative models used for virtual assistants, such as a 2017 WaveNet model [3] used to synthesise Google Assistant's Japanese and English voices.

Neural audio synthesis would particularly benefit the problem of human-computer interaction. Improved audio synthesis methods would provide a more seamless interaction between computers and humans. To facilitate communication in both directions, audio synthesis advancements would need to coincide with similar improvements to natural language processing and human speech recognition.

Mixed reality device design is another area for commercial applications of work in this thesis. Many constraints exist for the development of Mixed Reality devices. The works described in this thesis have the potential to optimise audio production for Mixed Reality devices in the context of these constraints.

The first such constraint is cost, as Mixed Reality devices are generally aimed at the consumer device market. Materials used to build the devices are cheap and there is not much scope for redundant components. This keeps end-device price low to encourage consumers to purchase them. Image-based methods of estimating room acoustic properties, such as those described in Chapters 4 and 5 could eliminate the need for dedicated audio processing hardware for this task.

Device form is another constraint preventing inclusion of an unlimited number of components. Many Mixed Reality devices are designed to be worn on the head. Therefore they need to be light enough to comfortably wear. The dimensions are also constrained since it should fit around the shape of a human head. Inclusion of cameras are usually prioritised in such devices since much of their applications are image-based. Using image sensors for audio scene analysis as described in Chapters

4 and 5 may help designers keep to specifications regarding device physical form.

1.3 Industry Placement

Throughout the entire Master's program, I was based in the Galway office of Xperi (formerly FotoNation Ltd.). From the outset, my research projects were intended to be aligned with the commercial interests of Xperi. This company has a commercial focus on a number of areas including image processing technologies, home audio systems and intellectual property licensing. My work aimed to utilise the combined expertise of the imaging and audio divisions within the company. The projects I carried out (reverberation estimation and audio synthesis for gaming) achieved this goal, albeit in different ways and to varying degrees.

The work in Chapter 4 arose following a research question posed within the company: can room acoustic properties be evaluated using a minimally invasive technique? In addition, could such a technique be incorporated into a Mixed Reality device? Discussions with other researchers in the company lead to the proposition of an imaging-based technique for estimating room acoustic properties. For this work I collaborated with an engineering team based in Los Gatos, California. This project addressed a need within the company for research on audio production for Mixed Reality applications. The development of a proof-of-concept device for this work is described in Chapter 5.

The process of designing and implementing a proof-of-concept was a valuable experience, especially industry-based research setting. The device was constructed in a short time frame. It involved significant cooperation between separate teams from different company branches, based in different countries.

Following the assembly of a prototype device, I travelled to Los Gatos, California to work with the team I had collaborated with in the preceding six months.

Here I focused on testing and debugging the device. I took input from team members on producing a demonstration format which would be most appealing to other audio engineers at Xperi. The demonstration process is described in Chapter 5. It proved crucial to have direct input from audio engineers as the solution was intended to be used for audio production. In addition, the cross-domain nature of this project made clear communication a priority.

1.4 Disseminated Results

Portions of work presented in this thesis have been published in conference papers and patent disclosures.

"Synthesizing Game Audio Using Deep Neural Networks" [4] was accepted as a full paper at the 2018 "IEEE Games, Entertainment, Media Conference". This paper was based on the audio synthesis techniques described in Chapter 3. The full paper is included in Appendix A.

The work in this Chapter 3 also formed the basis of a patent filing application. Usually the output of the generative model used in this work (a Generative Adversarial Network (GAN)) is of fixed size. The patent extends on the system described in [4] to include a method of extending the generative model's output. Our proposal includes the training of a separate model to generate the latent space variables inputted to the generator model. This artificially lengthens the output signal since the next latent space vector will be based on previous signals synthesised by the generator model.

A patent disclosure was created which incorporates some of the work described in Chapter 4. The patent disclosure describes a method of adapting audio synthesis based on environmental information. Still images of an environment are passed through a neural network based pipeline to estimate acoustic properties. These

results will modulate how audio is produced such that it is consistent with the present physical environment. This is intended for use in a mixed reality scenario where audio playback needs to be modified in real-time based on previously unseen environmental conditions.

1.5 Acknowledgement of Contributors

The works presented in this thesis were carried out in collaboration with my industry partner, Xperi. The original project which produced content in Chapter 3 was supported by fellow researchers Joseph Lemley and Ryan Cassidy. Joseph assisted with any problems encountered during my first foray into GAN training. Ryan gave his expertise in audio engineering, which helped to shape the project.

The work in chapters 4 and 5 originated from the same project. This project was supported by fellow engineers at Xperi including Martin Walsh, Mike Goodwin and Edward Stein, Cosmin Rotariu and Peter Corcoran. Martin, Mike and Edward provided invaluable direction to the project with their knowledge of audio engineering. They also gave advice on developing the demonstration device described in Chapter 5. Cosmin and Peter designed and manufactured the 3D-printed case for the demonstration device. Their input resulted in a polished and professional final demonstration.

Chapter 2

Literature Review

This chapter serves as a literature review of Deep Learning (DL) techniques relevant to work described in this thesis. The objective of the thesis is to investigate Deep Learning based methods of improving audio production. Two audio production focused methods are described in Chapters 3 and 4. An embedded systems based demonstration of the work in Chapter 4 is documented in Chapter 5. Important foundational knowledge relating to the three aforementioned experimental chapters is presented in this literature review chapter.

Firstly, the origins of Deep Learning are described in Section 2.1. A brief review of neural network research is given in Section 2.2. The training of neural networks is described in Section 2.3. Then two neural network architectures which are of particular relevance to this thesis are reviewed, namely the Convolutional Neural Network (CNN) and the Generative Adversarial Network (GAN) (sections 2.4 and 2.5 respectively). Both of these techniques were originally developed to tackle problems in the field of computer vision research. After initial success on complex tasks such as object recognition [2] and image synthesis [5], their applications to audio processing began to be investigated.

The information in this chapter gives context to the techniques used in later

chapters of this thesis. The techniques are reviewed here in terms of their origins and general function. Details of their implementations and use cases are given in Chapter 3 for GANs and Chapter 4 for CNNs.

Audio synthesis techniques are reviewed in Section 2.6, including both traditional and neural network based methods. This information relates to the experiments on audio synthesis in Chapter 3.

The properties of materials which influence room acoustics are discussed in Section 2.7. Neural network based image segmentation techniques are summarised in Section 2.8. These sections give context to the image-based method of room acoustic property estimation described in Chapter 4.

Finally, the fundamental concepts relating to Edge-AI are presented in Section 2.9. Edge-AI is a new computing paradigm for neural networks where inference is carried out locally rather than by a remote server. A demonstration of the experiments in Chapter 4 is presented in Chapter 5 which utilises Edge-AI technologies on an embedded system.

2.1 Origins of Deep Learning

It is debated exactly when this field emerged as a distinct subset of Machine Learning (ML). The term Deep Learning was first used in the context of Machine Learning in 1986 by Rina Dechter [6]. However, the Artificial Neural Network (ANN) existed as a theoretical concept long before it was implemented in software. Computational models of the human nervous system published in the 1940's are the earliest works of neural network research [7]. The study of Artificial Neural Networks quickly became the subject of research in its own right, separate from the field of biophysics.

Interest in ANN research peaked three times since they were initially proposed. The first notable wave of interest was initiated by the *Perceptron* algorithm [8] in 1958. This algorithm described a computational model for artificial neurons like those in the nervous system. It takes a number of inputs, performs a weighted summation of their values and returns either 1 or 0 if the sum is more than some threshold value. A Perceptron is illustrated in Figure 2.1.

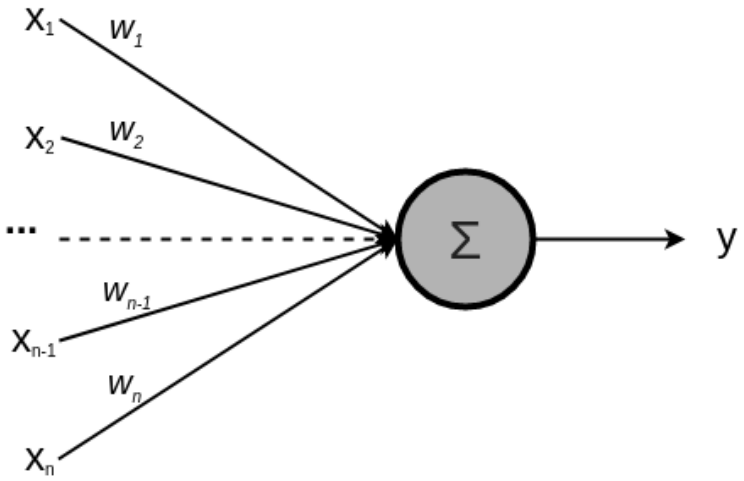


Figure 2.1: **A single *Perceptron***. The output y is 1 if the weighted sum of inputs is greater than a given threshold value.

A single *perceptron* neuron can only represent linearly separable functions. However the weights and threshold values can be learned. Hierarchical constructions of such neurons allow far more complex functions to be represented. A multi-layer Perceptron is shown in Figure 2.2. This is similar to the stacking of layers of neurons in modern neural network implementations to facilitate the learning of complex patterns.

The second wave of interest arrived with the introduction of back-propagation [9] for training neural networks. Back-propagation is a learning procedure for 'learning' the weights of connections between neurons in a neural network. Back-

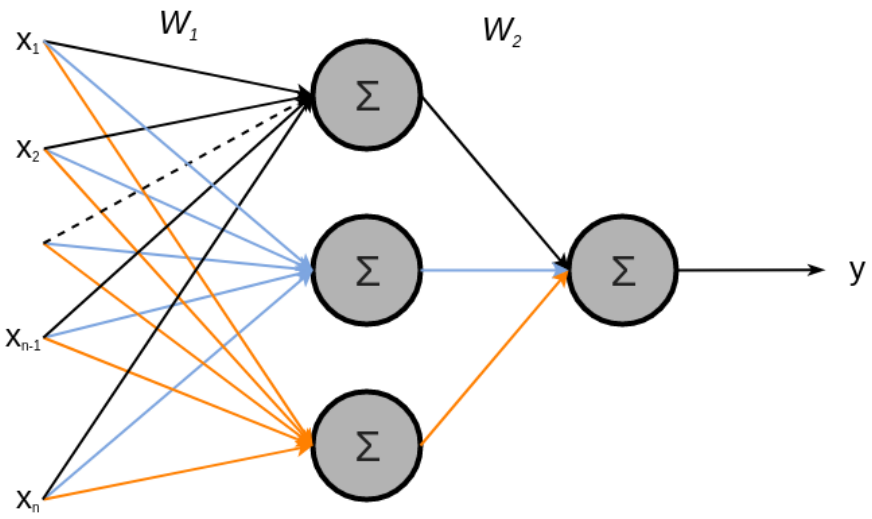


Figure 2.2: **A multi-layer *Perceptron.*** Weight matrices for each layer are denoted by a capital W with a subscript indicating the layer number. The same properties apply for individual neurons as for single *Perceptrons*. The layering of such neurons allows more complex functions to be represented.

propagation adjusts weight values so as to minimise a measure of difference between the network's output and ground truth data given a corresponding input value. This operation is repeated using a large pool of training data, continuously making small changes to network weights. Network training converges once the measure of difference is deemed small enough for a long enough period during training.

Derivatives of the difference measure propagate through the network, from the output layer back to the input layer. See Figure 2.3 for an illustration of a simple back-propagation operation using a loss function as a distance measure. These derivatives influence the degree to which connection weights are changed. Small errors in the output naturally warrant minimal changes to network weights. If there is a large margin of error then bigger adjustments to weights should be made to converge towards better results.

Crucially, back-propagation allows weights to be 'tuned' based on training data rather than 'hand-crafted' by a human expert. Weight values chosen by humans

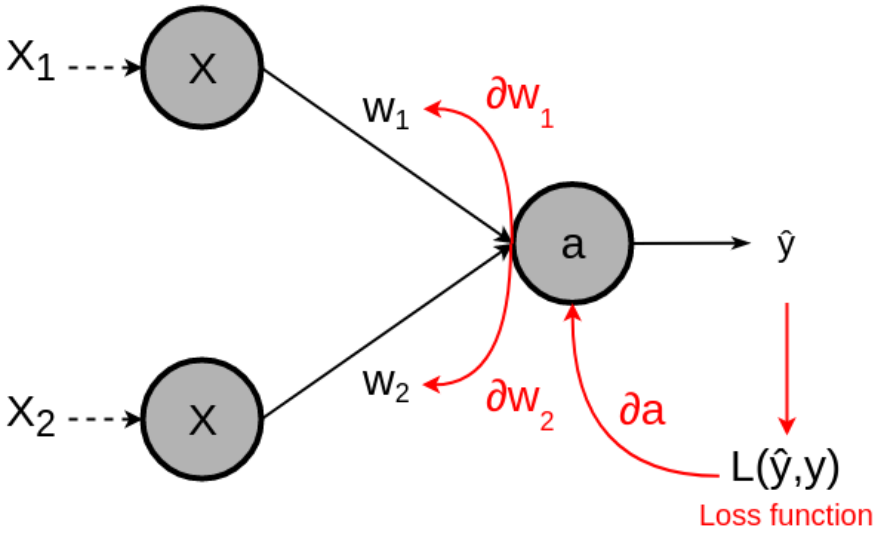


Figure 2.3: **Back-propagation of difference measure.** A measure of the difference between predicted and ground truth output is calculated known as a loss function. Derivatives of this measure are passed back through the network and used to update weights.

are highly unlikely to be anywhere near optimal. Nowadays, it is impossible to hand define neural network weights since most contain millions of connections. In the sense that it needs a ground truth value for every input, it is a learning procedure for supervised learning problems.

The third and most recent spike in neural network research was brought about by their success in image classification competitions such as ImageNet [1]. A deep Convolutional Neural Network achieved results significantly better than previous state-of-the-art methods in the ImageNet LSVRC-2012 contest [2]. This was proof that large, deep CNNs were capable of record breaking performance on a challenging, diverse dataset. Convolutional Neural Networks are discussed in detail in Section 2.4.

2.2 Factors Impacting Neural Network Research

The field of neural network research has rapidly grown in popularity since 2012. This is partly due to positive circumstances which did not exist in previous neural network research eras. The first factor to credit is the availability of digital data in volumes never before seen in history. Machine Learning, of which Deep Learning is a subset, deals with the problem of learning from data. A modern Deep Neural Network (DNN) can contain millions of learn-able weights. Figure 2.4 shows the large increases in DNN size since 1998. An enormous amount of training data is required for them to generalise, therefore it is vital to use datasets of a similar magnitude (+millions of images) to train these networks robustly.

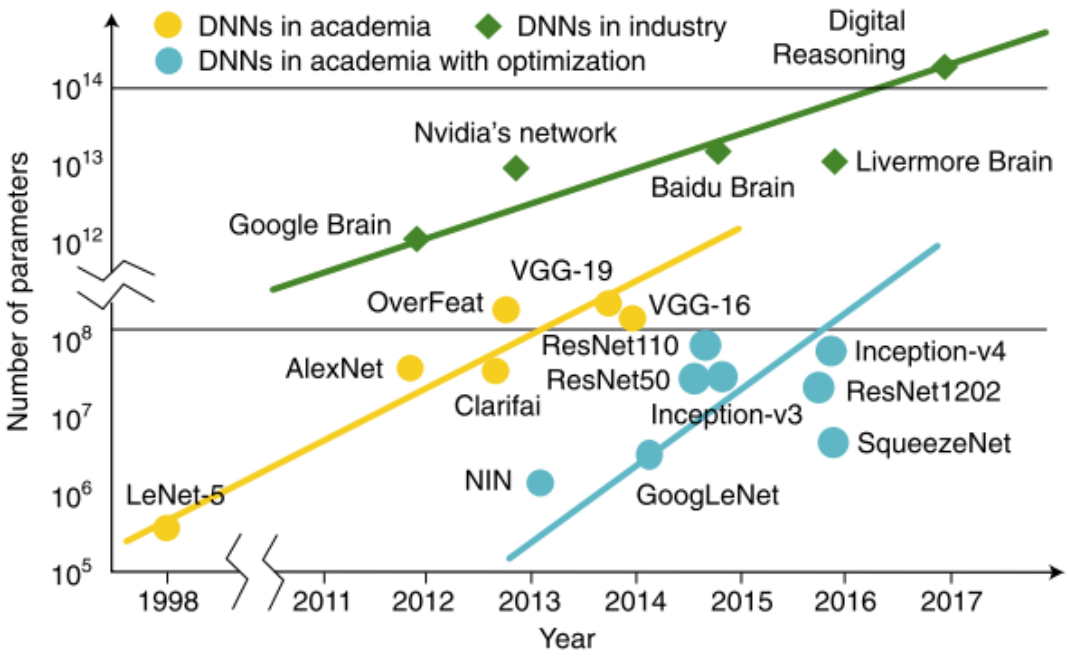


Figure 2.4: **Size of popular neural networks over time.** This figure illustrates the dramatic increase in the size of popular neural network models over time. The y-axis is in log scale. Diagram reproduced from [10] with permission from publisher [11].

With the increasing digitisation of the modern world, vast amounts of information are generated and stored. Valuable data comes from a wide variety of industries including healthcare, social media, military, and transport. Research institutions have curated enormous datasets and made them publicly available with the goal of furthering neural network research [1, 12].

While the amount of data available for training neural models is far larger now than during the last 'AI Summer', the size of the largest datasets hasn't greatly increased since the introduction of the ImageNet dataset [1] circa 2011. This is perhaps explained by the logarithmic relationship between training data volume and network performance [13].

The second factor to credit is the enormous growth in computing power facilitating network training. Neural network training is a computationally expensive process. It requires the repeated calculation of millions of matrix multiplications and derivatives, proportional to the overall network size. Performing network training on a sequential processor such as a CPU results in unacceptably slow training times.

Modern parallelised hardware architectures such as the Graphics Processing Unit (GPU) or Tensor Processing Unit (TPU) perform the required computations faster and more efficiently than CPUs [14, 15]. Figure 2.5 illustrates the difference between GPU and CPU architectures. The GPU in particular is the workhorse of modern deep learning research. GPUs contain hundreds of simple cores designed to perform many matrix multiplication operations in parallel. This is in contrast to CPUs which are made up of fewer, far larger cores designed to perform complex operations in sequence. CPUs have a large cache for storing volumes of data to be processed. GPUs have multiple cache structures to store smaller volumes of data.

While CPUs can perform individual operations faster than GPUs, the parallel structure of GPUs means that their total throughput is greater. In other words,

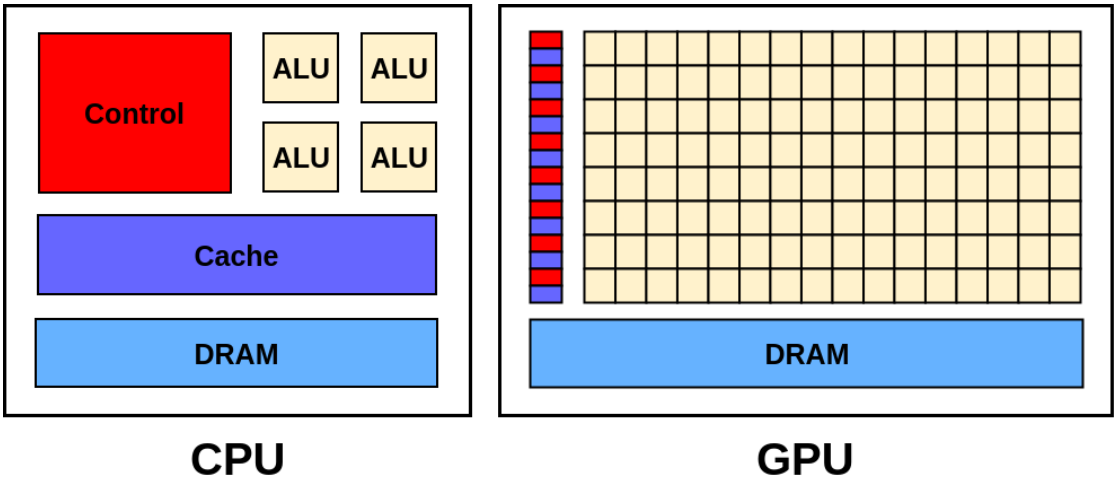


Figure 2.5: **CPU vs GPU architecture.** CPUs are designed to perform complex operations in sequential order. GPUs are designed to perform a large number of simpler operations in parallel. Their architectures are designed to optimise their respective functions. Note the difference in Arithmetic Logic Unit (ALU) count for sequential vs parallelised architectures.

CPUs are latency optimised, whereas GPUs are bandwidth optimised. Bandwidth optimisation is preferable in the context of neural network training since the amount of data which needs to be accessed and processed is so large.

Crucially, dedicating these architectures to neural network training enables deeper models to be trained. Since each layer takes as input the outputs from the previous layer, increasing the number of layers creates a more discriminative decision function. Therefore deeper networks, i.e. those with many layers, are able to learn more complex patterns from a dataset. Deeper networks generally achieve better performance which has improved state-of-the-art in many tasks [16, 17, 18].

2.3 Training Neural Networks

The training of a network on a single sample is evaluated by a *loss function*. It computes the error for a single training example. Loss functions are used in back-

propagation to define the derivative used to update parameter weights. Several types of loss functions exist. A specific loss function is usually chosen based on network architecture and problem being modelled. Equation 2.1 shows the calculation of a loss function suitable for regression, i.e. the prediction of a real valued quantity.

$$L(\hat{y}, y) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \tag{2.1}$$

The summation of loss functions over an entire training set is known as a *cost function*. The cost function is used by many training optimisation techniques to speed up the training process (see Section 2.3.2). Equation 2.2 shows the calculation of a cost function J based on loss function values for a set of m training samples. Varibles w and b refer to the weight and bias matrices respectively of the network for which the cost function J is computed.

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) \tag{2.2}$$

Training a neural network can be a delicate task. Using very deep architectures runs the risk of 'overfitting' a dataset. This is akin to memorisation. Overfitting is characterised by decision boundaries which closely fit training data values. See Figure 2.6 for an illustration of overfitting two-dimensional data. Overfitting data is risked when the number of network parameters greatly exceeds the number of samples used for training [2, 19]. A model which has overfit a dataset can be identified by comparing its accuracy on training data versus validation data [1]. High accuracy on training data combined with low accuracy on validation data is a reliable sign that the model has overfit the training dataset.

[1] For a classification task.

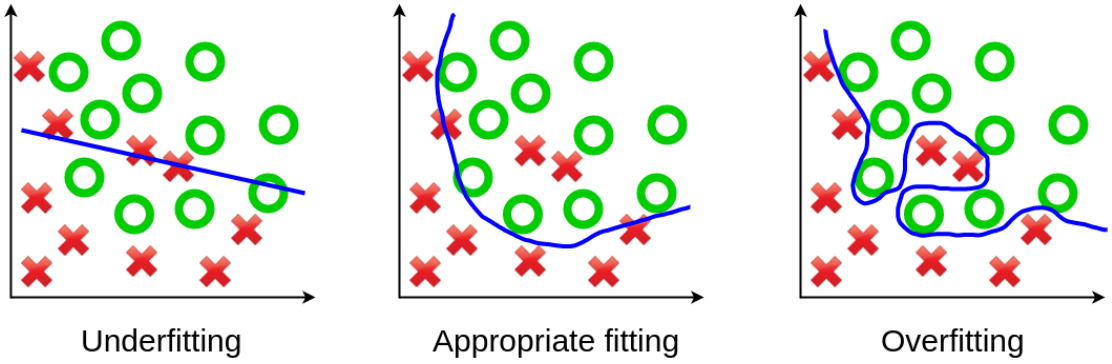


Figure 2.6: **Overfitting** This diagram shows a classic example of a model overfitting two-dimensional data. Neural networks are particularly prone to overfitting since the data they model is extremely high-dimensional.

2.3.1 Regularisation Techniques

A wide array of techniques have been developed for avoiding overfitting. Many of these techniques operate by attempting to simplify the network during training. This has the effect of *regularising* the network. An example of a *regularisation* technique is *Dropout*. Dropout is a commonly used method where for each training iteration only a random portion of nodes in each layer are trained [2] [19].

Many other *regularisation* techniques exist for the training of neural networks. Their primary objective is to reduce overfitting by simplifying the model. Too much regularisation can lead to the model *underfitting* the training set, so a degree of caution is reasonable when implementing these techniques in network training.

Data augmentation techniques can artificially increase the size of a training set. This helps to reduce the likelihood of overfitting and improve performance of DNNs [20, 21]. Preventing dataset overfitting allows models to learn a more robust decision function, ideally one which will perform well on data from the true distribution for the given task.

[2] i.e. undergo back-propagation

2.3.2 Neural Network Training as an Optimisation Problem

Training a neural network can be a long process given the volume of parameters in modern architectures. It is common practise to configure network training as an optimisation problem. Many optimisation algorithms have been developed to speed up the process of network training. Back-propagation is a crucial element in neural network training, allowing for network parameters to be *learned* rather than hand-crafted. It is difficult to control the rate and most importantly, *direction* of parameter learning without a suitable optimisation algorithm.

The most common optimisation algorithm is known as *gradient descent*. Its primary objective is to guide the back-propagation process to a global minimum cost function value. With gradient descent, parameter values are updated such that the cost function moves in the direction of the steepest negative gradient. Visualising the cost function as a parameter space, Figure 2.7 illustrates the process of gradient descent.

Using gradient descent to update model parameters will enforce that the cost function always moves in the direction of the steepest negative gradient. This is only possible if the entire training set is used to calculate the cost function. Other implementations of gradient descent use smaller *mini-batches* of samples to calculate a kind of intermediate cost function. The cost function value many fluctuate up and down, but will tend to slope downwards if implemented correctly. This is a more efficient solution if the volume of training data is large enough to make calculation of the cost function unfeasible.

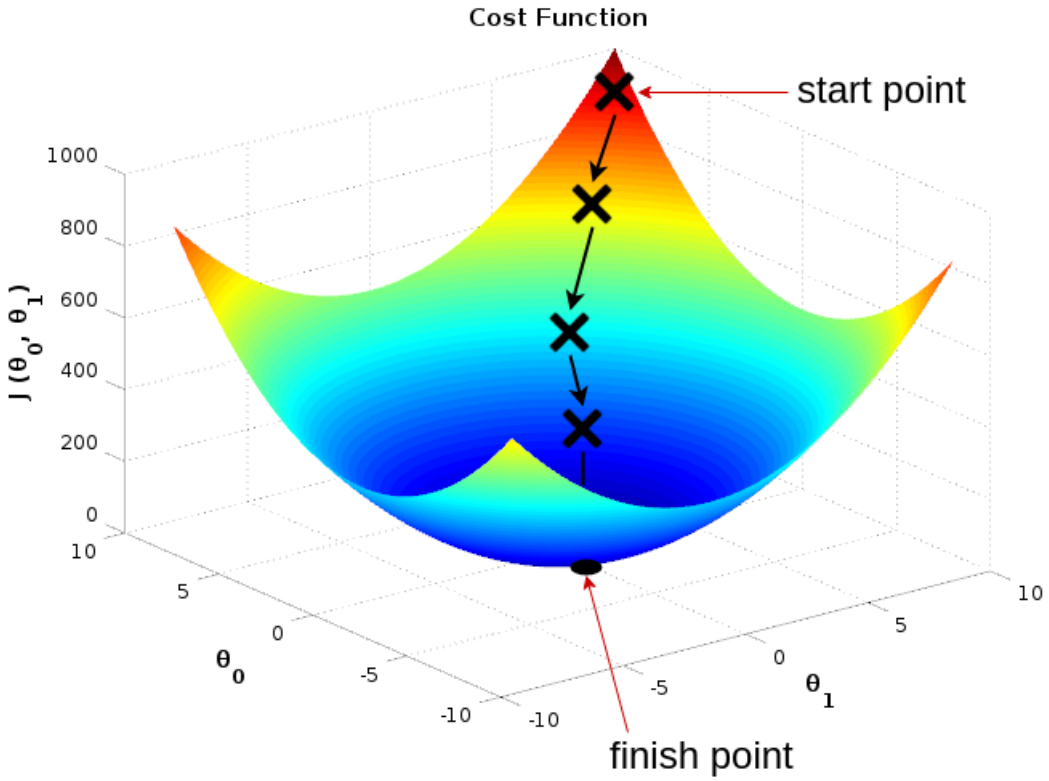


Figure 2.7: **Gradient descent.** This figure illustrates moving the cost function in the direction of the steepest negative gradient.

2.4 Convolutional Neural Networks

The Convolutional Neural Network (CNN) is a type of feed-forward neural network specifically designed to learn the variability of two-dimensional shapes. First published in 1998, it was originally applied to handwritten character recognition [22]. In 2012 CNNs revolutionised image classification when they were used in the ImageNet classification challenge and achieved error rates considerably better than state-of-the-art at the time [2]. The ImageNet dataset [1] was larger in scale and diversity, and more accurately labelled than other image datasets at the time of its publication. CNNs have a large learning capacity which allowed them to cope with the immense complexity of object classification using such a large dataset.

Since 2012, CNNs have surpassed human level accuracy on the ImageNet challenge [23].

The primary mathematical operation underpinning CNNs is the *convolution* operation. A convolution is a linear operation involving the multiplication of a set of weights, or a filter, with an input vector. A filter will produce a high activation when it closely matches the input. In the case of CNNs the input data is two-dimensional, therefore so are the convolutional filters. The filter is always smaller in size than the input so that sliding the filter across the entire input produces another vector. Applying a filter across an entire image allows it to identify the pattern anywhere in the image. This is called *translational invariance* and is an important feature of CNNs. Figure 2.8 illustrates a two-dimensional kernel convolving on an input matrix.

This process is expanded to three dimensions in the case of three-dimensional inputs, e.g. an image with three colour channels. In this case, the kernel will also be three dimensional to match the dimensionality of the input.

Computing values within the output of a two-dimensional convolution operation is carried out using Equation 2.3, where;

- a,b are dimensions of the output space
- n, n is the size of the kernel filter k

$$Output_{a,b} = \sum_{i=0}^{n} \sum_{j=0}^{n} k_{i,j} Input_{a+i,b+j} \tag{2.3}$$

As an aside, what is referred to as a convolution operation in the context of deep learning is in fact a *cross-correlation* operation. The flipping of the kernel (i.e. filter) which is technically required for the operation to be a convolution is unnecessary since filter weights are learned. The learning algorithm employed

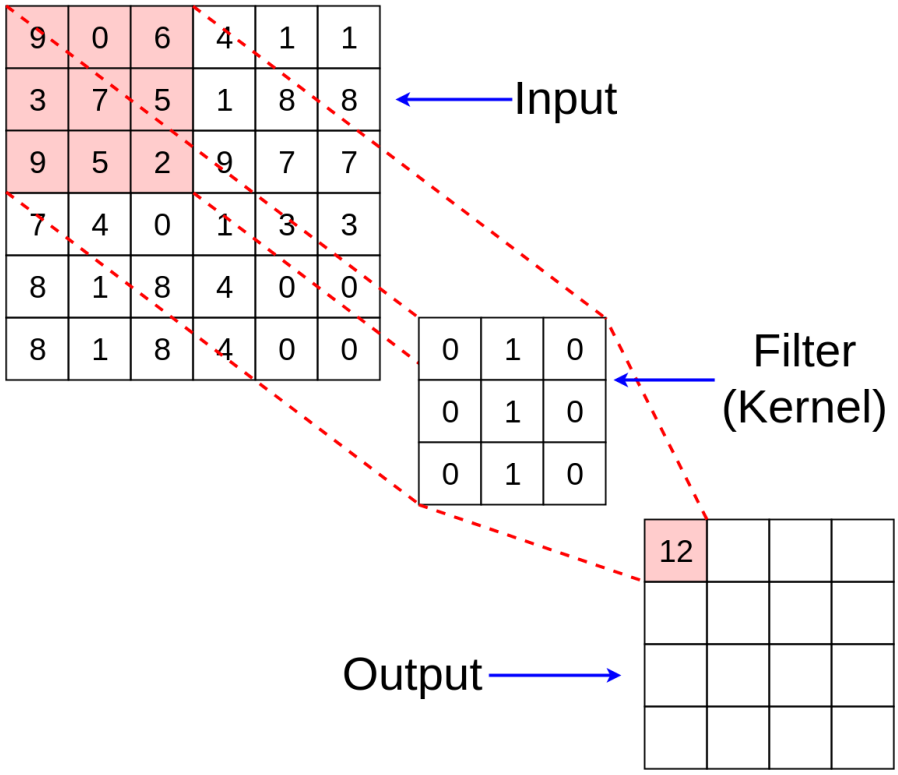


Figure 2.8: **Two-dimensional convolution operation.** The convolution operation is central to how CNNs function. Two-dimensional convolutions allow CNNs to identify objects anywhere in an image.

during network training will encourage the weights to be in the appropriate place and no deliberate kernel flipping is required [24].

Applying convolution operations to image data is not a new technique in computer vision. In the past hand-crafted filters were used for various tasks such as edge detection [25]. The distinguishing factor in CNNs is that the filters are learned from data rather than hard coded.

Convolution filters make up *convolutional layers* in CNNs. Each layer contains multiple filters which pass over an input. The number of filters in each layer tends to increase with network depth. Filters in the first few layers generally learn very simple features such as edges, and blobs [26]. Deeper layers build

upon the outputs of earlier layers, identifying more complex features in the input. Visualising patterns which highly activate intermediate layers greatly helps our understanding of how CNNs work [26, 27]. Figure 2.9 shows an example of the types of filters learned in CNN layers. Notice how feature complexity increases in deeper layers.

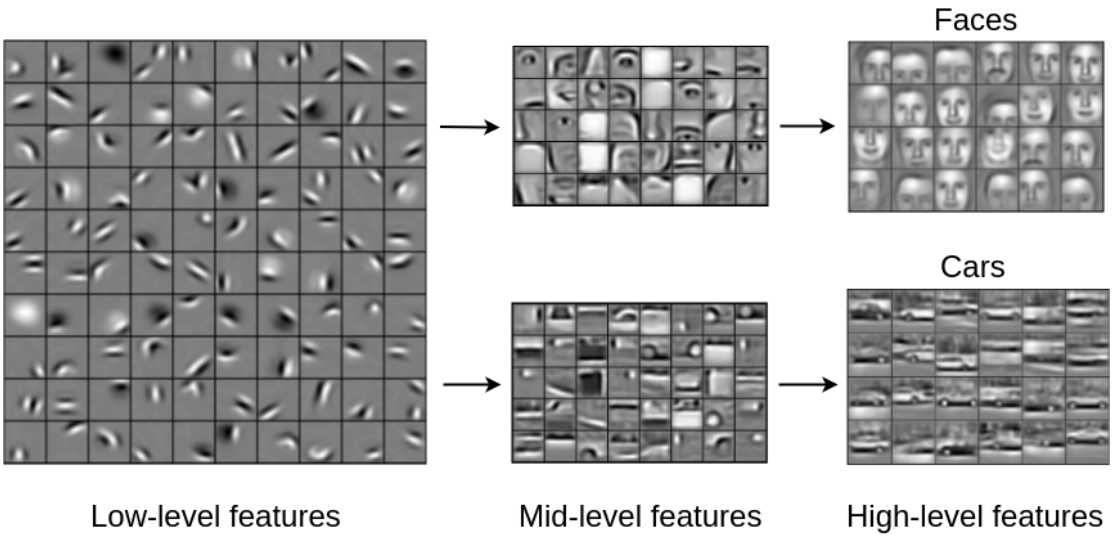


Figure 2.9: **Hierarchical features in CNN layers.** A *feature* refers to a learned filter in a convolutional layer of a Convolutional Neural Network. Low-level features are transferable across many target classes due to their simplicity. As each layer builds upon preceding layer outputs, features learned by a CNN increase in complexity towards deeper layers. Figure adapted from material in [28].

A convolution operation is usually paired with a non-linear operation which is applied to its output. These are known as activation functions. Several types exist which have different properties suited to certain applications. If the output is a binary classification, using a sigmoid activation function is a natural choice. The formula for this activation function is shown in Equation 2.4. An illustration of the sigmoid activation function is shown in Figure 2.10.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

However, the derivative of the sigmoid function tends towards zero as its input gets very large or very small. This slows down gradient descent, increasing training time. The most prevalent activation function in DNNs is the Rectified Linear Unit (ReLU), credited as a key to their recent success [2]. Using the ReLU activation function speeds up convergence during network training and leads to better solutions [2, 23, 29]. Calulcating the ReLU activation function is a simple maximum operation as shown in Equation 2.5. The ReLU activation function is also illustrated in Figure 2.10.

$$R(x) = max(0, x) \tag{2.5}$$

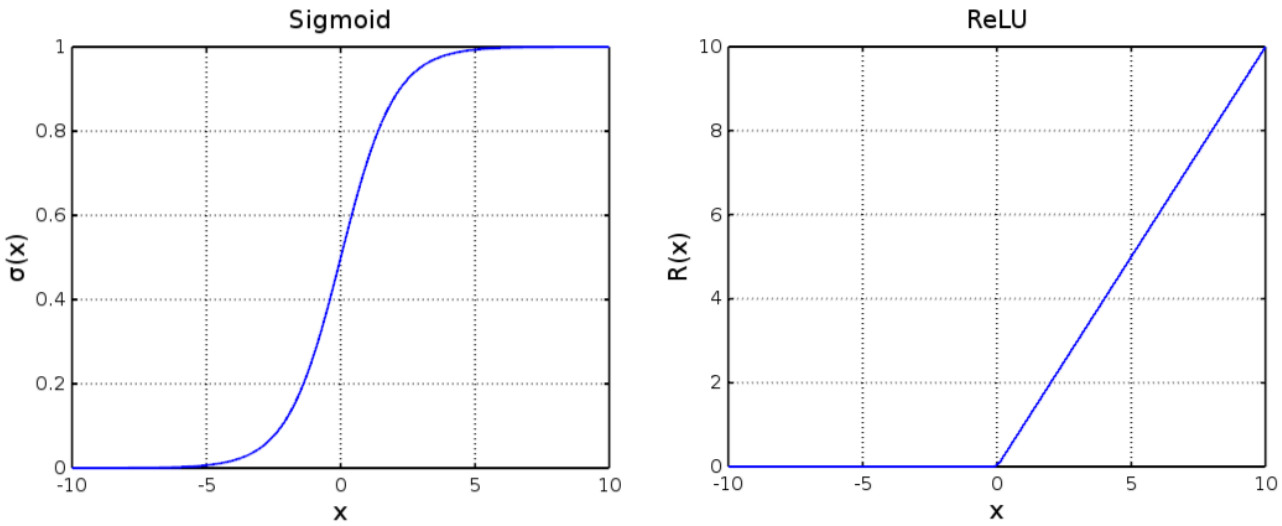


Figure 2.10: **Sigmoid and ReLU activation functions.**

2.5 Generative Adversarial Networks

The Generative Adversarial Network (GAN) is a class of generative model originally proposed in 2014 [5]. It consists of two separate networks; one for generating samples from a learned distribution, and one for discriminating real samples from generated ones. The networks are trained in an adversarial fashion, where each tries to 'fool' the other. A generator model G takes a random noise vector, or *latent space vector*, as input and learns to map it to an output which can be regarded as drawn from the training data distribution. The discriminator model D is trained to output, as a probability, whether a sample is generated by G or taken from the training data distribution. This architecture is illustrated in Figure 2.11.

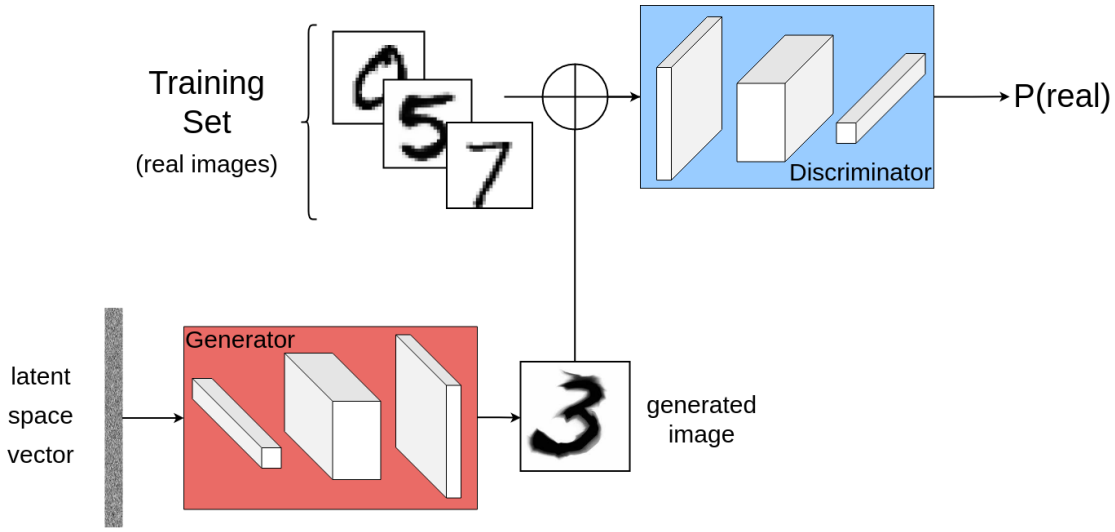


Figure 2.11: **Generative Adversarial Network**. A basic GAN architecture, including generator and discriminator networks. The generator is trained to differentiate between test set images and generated images. During testing, either test set or generated images are given to the discriminator.

Training requires a value function to be optimised such that both networks achieve a high level of performance. The value function to be optimised is shown in Equation 2.6, where $p_{data}(x)$ denotes the true data distribution and $p_z(z)$ denotes

the latent space distribution.

$$\begin{aligned} \min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] \\ + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \end{aligned} \tag{2.6}$$

D is trained to maximise the probability of assigning the correct labels to training and generated data, i.e. maximising $\log D(x)$. In theory, G is trained to minimise $\log(1-D(G(x)))$, but in practice it is trained to maximise $\log(D(G(x)))$. This is to avoid the saturation of $\log(1-D(G(x)))$ early in training when samples from G are poor and D can easily distinguish between them.

This training framework is an example of the 'minimax' algorithm, a commonly used decision making algorithm. It aims to minimise the loss of any decisions made assuming both 'players' are performing optimally. This training procedure in theory produces two robust models which can be used independently for inference after training.

Since a GAN is a learned generative model it synthesises new samples from the data distribution it has learned from a training set. Generation of realistic high-resolution face images shows the power of a generator model trained in this fashion [30, 31]. These generators are able to capture and reliably reproduce fine details in skin, hair and teeth. In addition, the structure of the face is kept consistent. This proves that GAN generator models can be constructed and trained to synthesise data with local and global structures learned from training data.

While GANs were originally developed for image synthesis tasks, they have been re-purposed in recent works to generate audio signals [32, 33]. A naive approach to audio synthesis using GANs would be to train a network designed for image synthesis to generate two-dimensional spectrograms. Spectrograms are image-like time-frequency representations of audio. This type of solution is suitable for discriminative tasks, however not for synthesis tasks. The reasons for this

are discussed fully in Section 2.6.3.1. A preferable approach is to directly synthesise raw audio waveforms, avoiding the use of feature representations. Other neural network types have been designed to operate on raw audio without the need for engineered acoustic features [34, 35]. The architectures of GAN generator and discriminator networks can be modified such that they operate on one-dimensional signals.

Reproduction of global and local structure is especially important in the synthesis of audio. Audio signals have very localised patterns which can repeat hundreds of times per second. Voice pitch is an example of such a pattern. Other structures exhibit periodicity over far longer ranges of time, such as melody in music. This reveals the wide range of temporal scales present in audio signals which complicate the training of generative models to produce high-fidelity audio with comprehensible global structure.

2.5.1 GANs for Audio Synthesis

Inspired by GAN's success on imaging tasks, several works have been carried out applying it to audio synthesis [36, 32]. Modifications can be made to existing GAN architectures which allow them to model one-dimensional signals. Flattening a GAN to operate on 1-D signals is one such modification. This involves turning two-dimensional filters into long one-dimensional filters, as shown in Figure 2.12. Widening the receptive field is also necessary to capture temporal patterns. This concept applies to all methods of neural audio synthesis. For example, autoregressive models such as WaveNet [34] incorporate dilated convolutions to widen receptive field exponentially with depth.

These modifications are necessary as audio and image data have vastly different traits. In particular, audio signals exhibit far more periodicity than image signals. One-dimensional GANs with wide receptive fields capture the temporal patterns

which characterise audio. For example, a 440Hz sinusoid sampled at 16kHz will repeat every 36 samples. A receptive field of at least this length would be required to capture this signal.

2.5.2 WaveGAN

The WaveGAN [32] model was chosen for this work on audio synthesis. A model which synthesises audio quickly is desirable as the postulated use-case is real-time audio synthesis in gaming. A fully parallel generator model is therefore necessary to achieve this goal.

The creators of WaveGAN state that while GANs have been successful in image synthesis tasks, they have not been applied much to audio synthesis. In a discriminative setting, the approach of 'boot-strapping' image processing algorithms for audio classification tasks is common. However, a barrier to implementing GANs naively to audio feature representations (e.g. spectrograms) is that audio features are non-invertible. Therefore, it is not possible to produce an audio waveform from a synthesised (2-D) feature representation without significant quality loss. This concept is discussed in more detail in Section 2.6.3.1.

WaveGAN is a one-dimensional version of the DCGAN architecture [37]. WaveGAN synthesises raw audio by operating on 1-D signals instead of higher dimensional representations of them. All filters in the WaveGAN architecture are one-dimensional of length 25, instead of two-dimensional 5x5 filters.

Audio signals (1-D) have different characteristics to image signals (2-D) so should be analysed and synthesised in matter reflecting this. For example, audio signals are more likely to exhibit periodicity than image signals. This results in patterns across long temporal ranges which need to be accounted for when analysing or synthesising audio signals.

WaveGAN's generator and discriminator models are shown in Figures 2.13 and

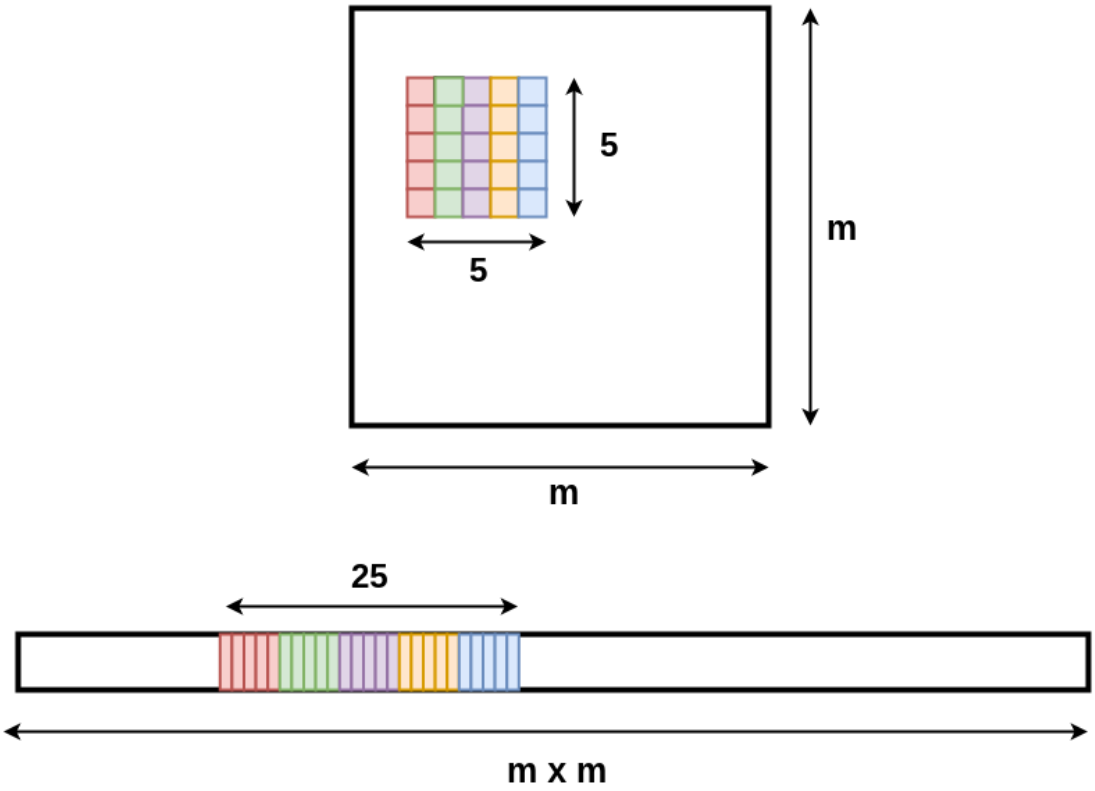


Figure 2.12: A 5x5 filter is "flattened" forming a 1-D filter of length 25. The input shape in the flattened one-dimensional architecture is equal to the product of the input dimensions in the two-dimensional GAN (not to scale).

2.14 respectively.

In this work, WaveGAN is retrained using data from AudioSet [12]. The data used for training is described in Section 3.2.2 We trained our version in a similar fashion to the authors of WaveGAN including using a similar amount of training data, keeping the data in the same format, and running training for the same number of iterations. The hyperparameters (i.e. settings) used for network training are found in Appendices B and D.

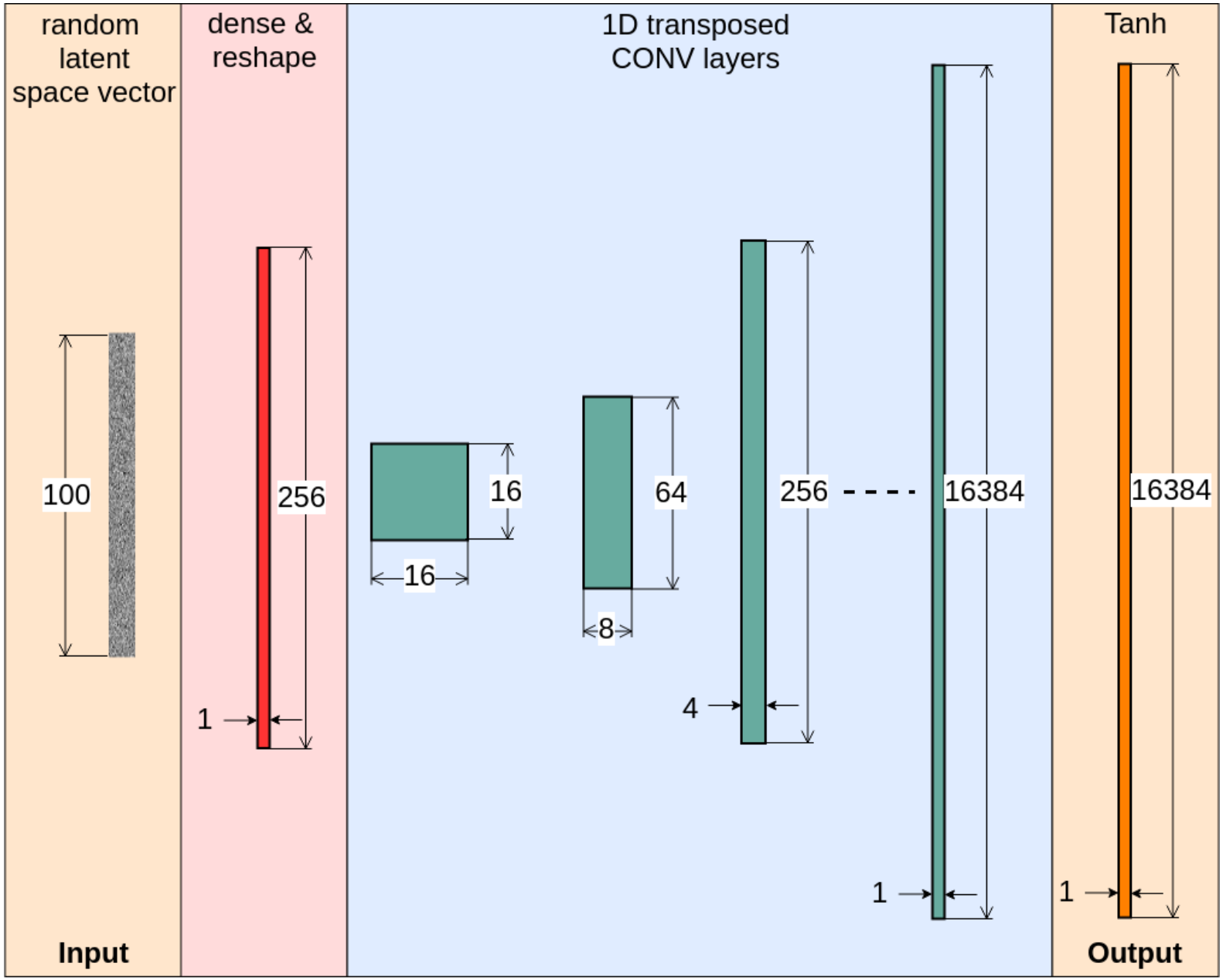


Figure 2.13: **WaveGAN generator.** This diagram shows the generator model of the WaveGAN architecture. The output corresponds to just over one second of audio sampled at 16kHz. A number of the convolution layers are skipped in this illustration for simplicity.

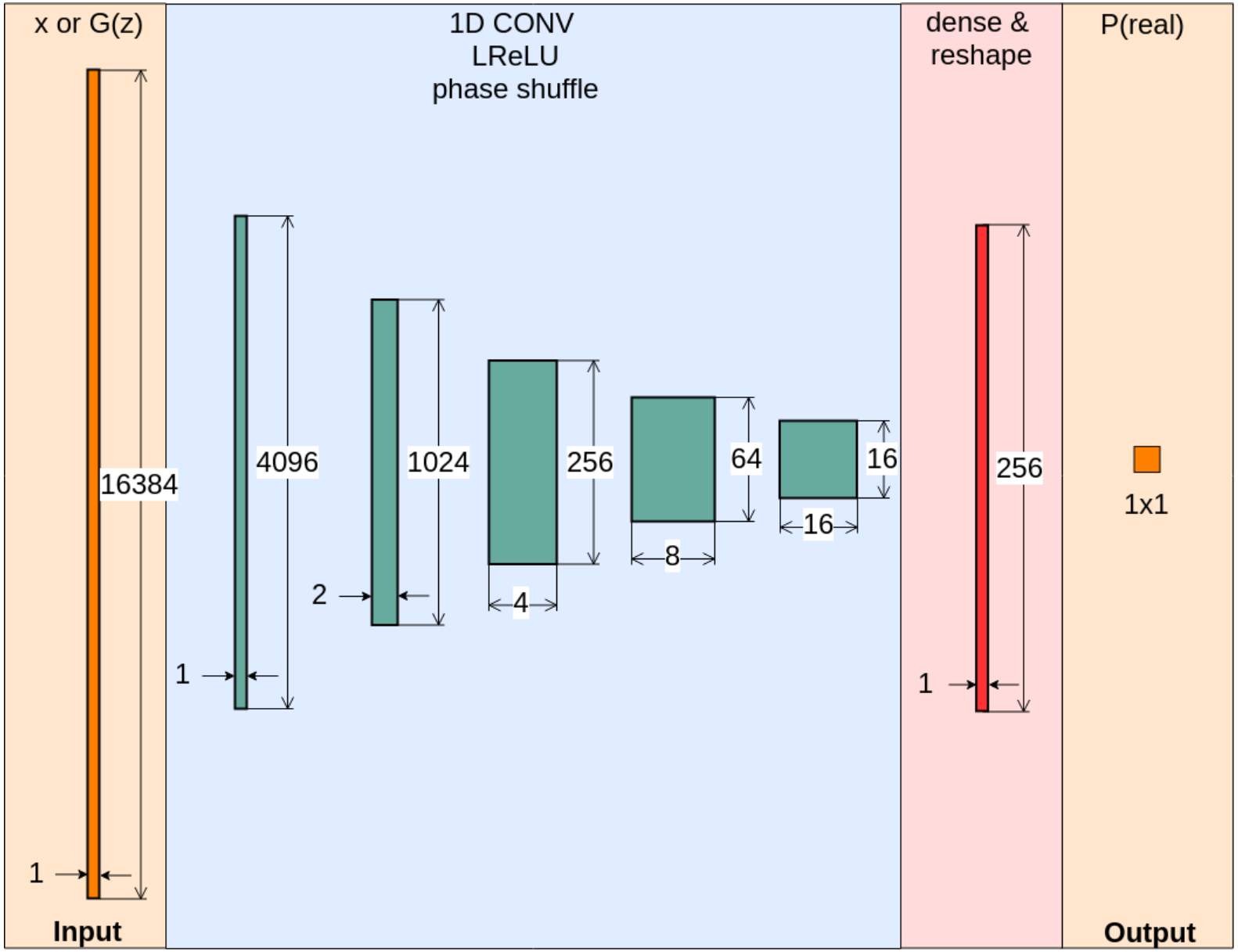


Figure 2.14: **WaveGAN discrimnator.** This diagram shows the discriminator model of the WaveGAN architecture. The output is a probability that the input sample is belonging to the distribution of real data (i.e. it is a real sample not from the generator).

2.6 Review of Audio Synthesis Techniques

In this section both traditional and machine learning based methods are discussed. Those most relevant to this project will be included in this review as many such techniques exist.

2.6.1 Traditional Methods

Audio synthesis is a complex task which in the past was usually left to experienced audio engineers. Trained professionals produced sound effects required in films and games with the help of dedicated audio production tools. In recent years, audio synthesis techniques have been developed which rely primarily on learning data to synthesise new audio. These techniques can be implemented by anyone with a 'software background' or even packaged into simple applications accessible to anyone.

Traditional methods of audio synthesis require an understanding of variables such as digital signal processing and specialised toolkits. To create a new sound effect one would need to apply this specialist knowledge to a number of production steps including; recording or sourcing suitable samples to start with, augmenting them with various mixing techniques, and layering other signals on top of the original signal. A trained operator would use their own judgement when deciding how to produce a new piece of audio. This may be beneficial if they have a personal style they wish to incorporate into the result. However, this is subjective and not important for every aspect of audio production.

The production of even simple sounds can be a laborious and monotonous task. Delivering custom audio scenes and effects for an entire application, movie or game requires a significant investment of time and money. To save on these costs, pre-mixed 'stock' audio is often used in audio production for low budget

projects.

In order to overcome the limitations of traditional audio synthesis methods, various neural network based methods have been developed in recent years. These new methods are generally more flexible than their traditional counterparts since they are learned from data rather than relying on a set of hand-crafted rules. They have shown considerable promise despite a lack of basis in fundamental audio processing techniques.

2.6.2 Neural Generative Models

Many types of generative models exist for synthesis of audio. Regardless of architecture, all generative models learn from large datasets the most important attributes needed to produce new signals. Once trained they can be given an input (i.e. sampled) which results in a signal which is most likely based on training data.

Generative models can be categorised in a number of ways including based on their method of sample computation (i.e. autoregressive vs parallel) and based on the type of data they generate (i.e. audio feature representations vs raw audio waveforms). The project described in this chapter is an example of a model which produces an audio waveform in a single parallelised-computation inference pass. A generative model known as a Generative Adversarial Network (as described in Section 2.5) is used for audio synthesis in Chapter 3.

Alternative methods exist for audio synthesis which pre-date GANs. Most notable are a class of models known as autoregressive models. Autoregressive models are a common statistical technique that generate output as a function of previous output values. Deep neural autoregressive models however, have only become prominent in the last few years, notably with the introduction of the PixelCNN model [38] in 2016. This model sequentially predicts image pixels along both

spatial dimensions. PixelCNN served as the basis for WaveNet [34], a sequential generative model for audio waveforms.

Autoregressive models have been used to synthesise images [38, 39, 40, 41], text [42, 43], speech [3, 44], music, and in sequence-to-sequence tasks such as machine translation [45, 46].

Autoregressive models are a logical choice for audio waveform synthesis as they produce sequential data to capture temporal patterns. Audio signals are fundamentally periodic in nature. At a basic level, audio waveforms are the summation of many sinusoidal signals (fundamental frequencies) at varying amplitudes. Audio signals can contain temporal dependencies at multiple time scales. For example, a speaker's pitch will produce an identifiable pattern approximately every 5ms at 180Hz. The intonation of their voice, due to accent or language, may fluctuate at larger timesteps, e.g. every few seconds.

Unfortunately the ability to model long range temporal dependencies as is necessary for speech synthesis comes at a large computational cost. With widespread adoption of GPUs and multi-core CPUs, modern computers are massively parallel. Convolutional Neural Network architectures take advantage of this to achieve fast discriminative and synthesis results. Traditional autoregressive models are unable to utilise parallel computing hardware for synthesis because computations must occur in sequential order. This is a large disadvantage in the case of audio signals. A high number of samples must be generated even for a short duration of audio waveform output.

Attempts have been made to parallelise the computation of samples from autoregressive models. Parallel WaveNet [3] introduces a technique called 'Probability Density Distillation'. This method trains a new parallel feed-forward network from an existing WaveNet model. The result is faster than real-time speech synthesis without significant decrease in quality. A trade-off exists between choosing

models with fast training time and slow inference and the inverse. Spending the extra time training can result in an autoregressive synthesis model achieving faster than real time performance.

2.6.3 Data Formats for Audio Synthesis

When training a neural network to synthesise audio, there are two ways in which audio data can be presented to the model. The first of which are two-dimensional feature representations of audio waveforms. The second format is to present a model with audio waveforms directly.

2.6.3.1 Feature representations for audio synthesis

Generative Adversarial Networks designed for image synthesis can be applied to the task of audio synthesis without modifying their architectures. This approach involves the use of audio feature representations which are similar to images.

Audio feature representations are image-like ways of representing one-dimensional signals. To produce a feature representation, first the signal is passed through some type of filter or transform, such as the Fast Fourier Transform (FFT), in a sliding window fashion. The transform outputs values at a number of frequency bands. The results of each window passing through the transform are stacked side by side to produce a two dimensional image-like result. Most commonly used representations group frequencies into logarithmically spaced bins and discard phase information [47]. This prevents the signal from being inverted. For this reason, using image-like feature representations is not advisable for audio synthesis tasks.

See Figure 2.15 for an example of a *spectrogram*. This is a popular audio feature representation sometimes applied to discriminative audio classification tasks [48].

The inadequacy of synthesis techniques based on audio feature representations is shown by Donahue *et al.* [32]. Authors of this paper implemented a GAN for syn-

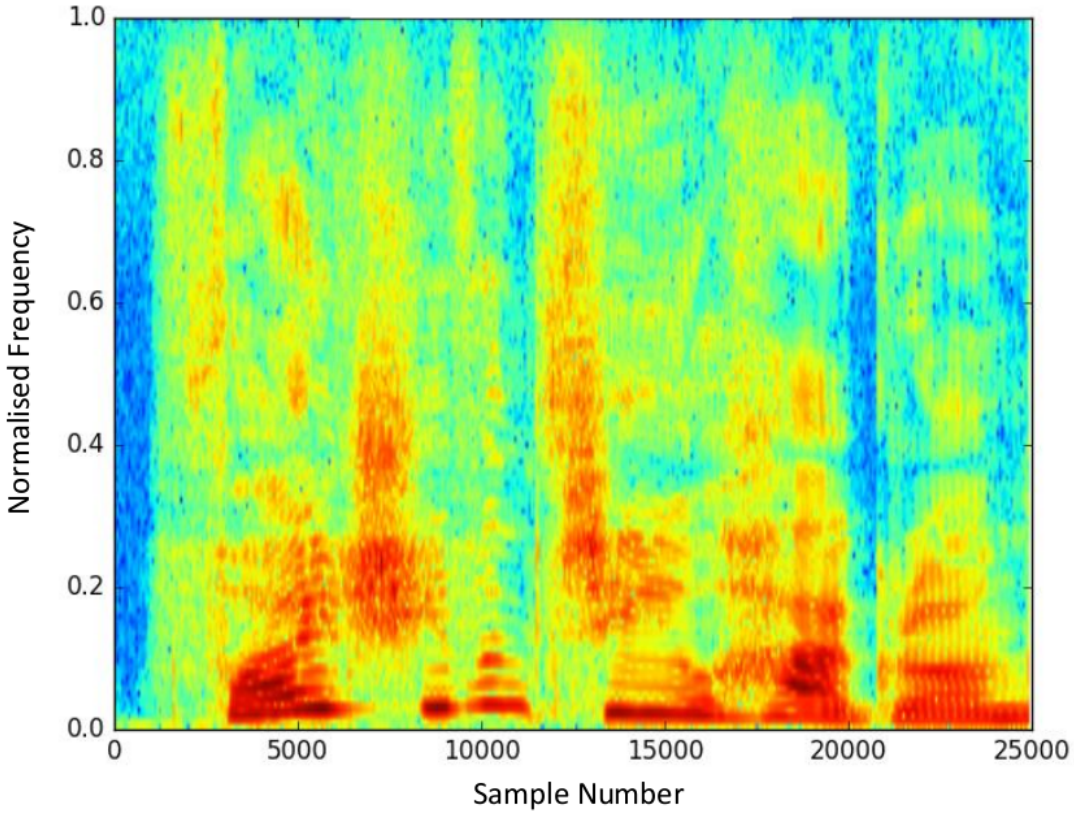


Figure 2.15: **Audio Feature Representation.** This is an example of a spectrogram. Frequency is normalised against the sampling frequency, in this case 44.1 kHz. The short-time Fourier transform (STFT) is taken of 128-sample windows of the input signal. The output of STFT is an estimation of the frequency content of input signal. STFT operation outputs are stacked side-by-side and plotted. Frequency amplitude is denoted by colour, with red being highest and blue lowest. Figure taken from work to classify regional language dialects [49].

thesis of spectral representations of audio called SpecGAN. The two-dimensional deep convolutional GAN (DCGAN) method [37] was applied to spectrogram representations of audio signals. The representations used were designed to allow for approximate inversion using the Griffin-Lim algorithm [47]. However, subjective listening tests showed that the quality of reconstructed signals from spectrograms

was poorer than for directly generated signals. The result can be attributed to the inevitable data loss in signal reconstruction as previously described.

2.6.3.2 Direct synthesis of audio waveforms

The second approach to audio synthesis using GANs is generating waveforms directly, without the use of intermediate representations. As previously explained, feature representations are suitable for discrimination tasks but not optimal for learning to generate new audio. The modifications described in Section 2.5.1 must be made to any two-dimensional GAN chosen to directly model audio waveforms.

This work involves using GANs to model audio waveforms. The WaveGAN [32] architecture is chosen as it is already modified to operate on one-dimensional signals. Section 2.5.2 describes this model in more detail and it's application to this work.

2.7 Material and Room Acoustics

Room acoustics is a combination of room geometry and the absorptive properties of materials. Room geometry dictates the distance sound waves travel before meeting a surface. Materials dictate how sound waves behave when they hit a surface. When analysing spaces acoustically, room geometry has the greatest effect on *early reflections*. These are the first sound reflections that are heard after an incident sound wave. The delay between the incident sound wave and reflected sound wave is a factor of the distance between the sound source and the surface it reflects off. Therefore, the greater the distance between the listener and a surface, the longer it will take for the reflected sound wave to reach the listener.

On the other hand, material has a greater effect on *late reflections*. Late reflections are those which have propagated across a room multiple times before

reaching the listener. They arrive much later than early reflections since they have bounced around the room several times before being observed. The amplitude of a reflected ray is dictated by the absorptive properties of the materials it reflected off. The absorptive effects are compounded with each reflection resulting in late reflected rays having a much lower amplitude than early reflected rays.

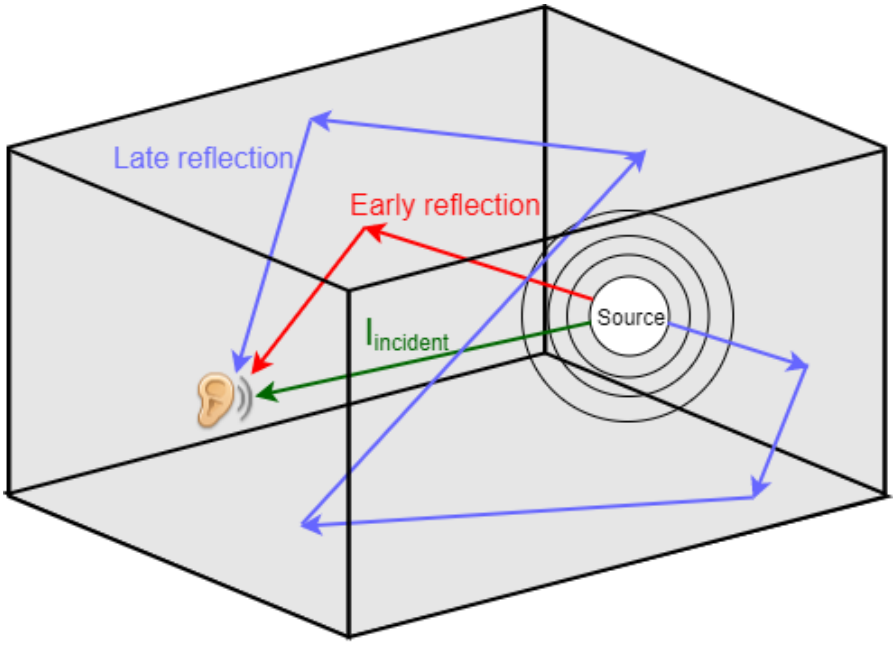


Figure 2.16: **Early vs late reflections.** Early reflections reach the listener soon after the incident wave. The late reflections reach the listener after reflecting off multiple surfaces in the space. The amplitude of the late reflections decreases rapidly as a result.

While the work in this chapter focuses on material in the context of room acoustics, both are equally important in the context of room acoustic analysis as a whole.

2.7.1 What physical properties dictate a material's absorptivity?

The physical properties of materials dictate how sound waves interact with surfaces. When a sound wave hits a surface it is either reflected, absorbed or transmitted.

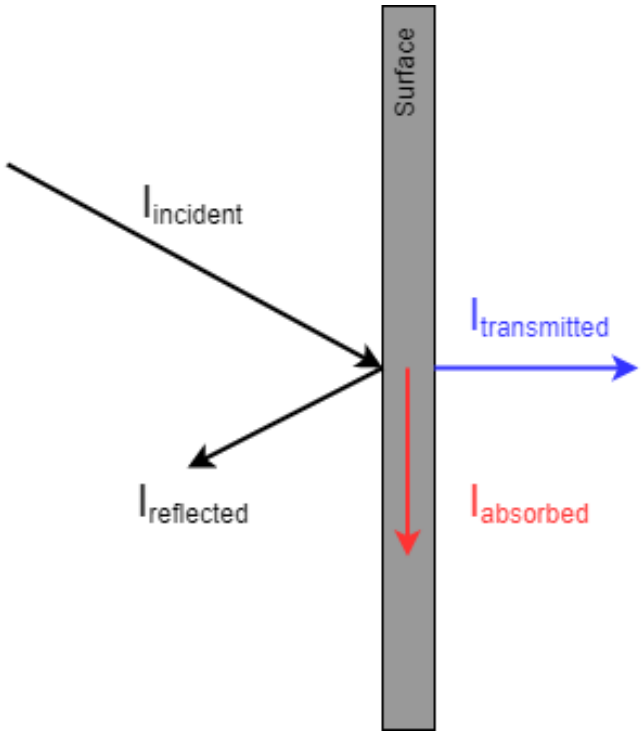


Figure 2.17: **Sound wave at a surface.** The incident wave $I_{incident}$ hits a surface. A portion of the sound energy is absorbed by the material. Another portion of the incident energy is transmitted on the other side of the surface. The rest of the energy is reflected as $I_{reflected}$.

Materials are given an absorption coefficient α based on the ratio of incident to reflected sound wave amplitudes. The higher the absorption coefficient the more sound is absorbed by the material.

$$\alpha = \frac{I_{incident} - I_{reflected}}{I_{incident}} \tag{2.7}$$

The more porous a material is the more sound it will absorb. A sound wave hitting a porous surface will pump air in and out of the material. Friction between moving air (i.e. the sound wave) and the surface dissipates sound energy. Denser materials reflect more of the incident sound wave since there is less friction between the air and the material's surface.

The less sound energy absorbed by a room's surfaces the more late reflections will propagate around the room. The summation of all late reflections is known as *reverberation*, or reverb for short. Reverberation is a key aspect of acoustic design of buildings. It can be desirable depending on an environment's purpose. Materials have a large impact on the reverb produced by a space.

Reverb time is the length of time taken, in seconds, for an initial sound source to decay by 60 dB from an initial source amplitude. See Figure 2.18 for a graph illustrating reverb time.

Reverberation is one of the most important factors shaping room acoustics. Every environment produces reverb uniquely, so audio engineers usually account for its effects during recording and playback. Increasingly, virtual and mixed reality environments need to be accounted for. These are spaces for which no reverberation measurements have been recorded or can ever be recorded. This poses a problem for audio engineers seeking to produce high quality audio which matches a given virtual environment.

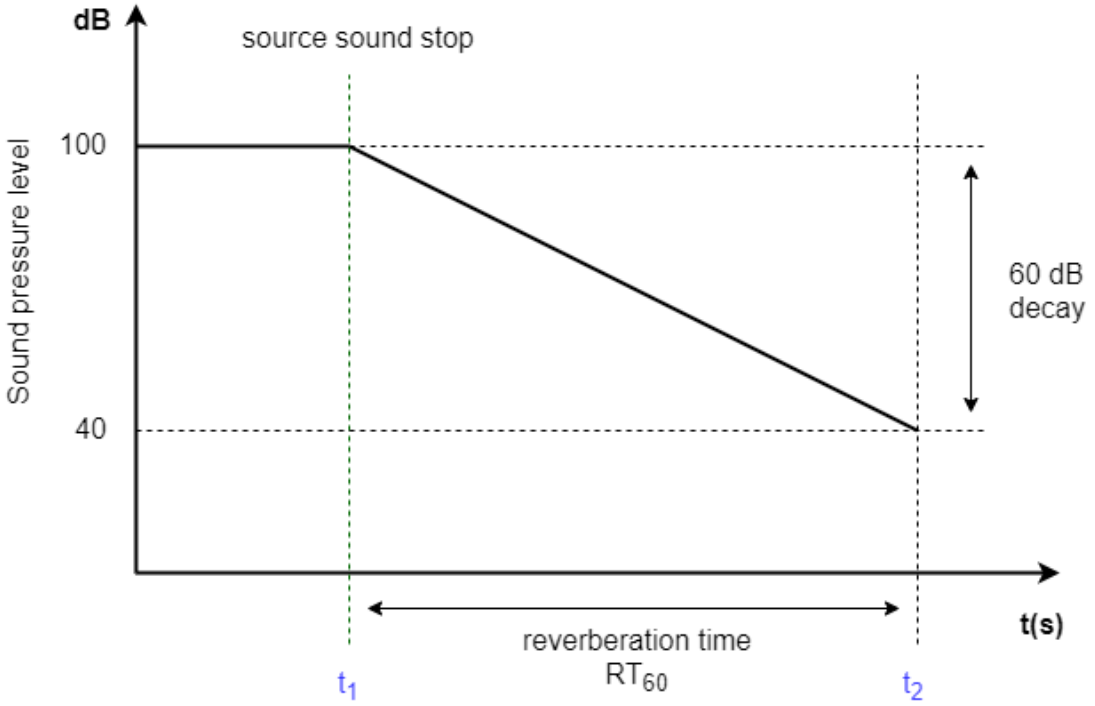


Figure 2.18: **Measuring reverberation time.**

2.7.2 How is material used advantageously in sound design?

Choice of building material is an important factor in designing spaces with *"good"* acoustics. The definition of "good" varies depending on the application. In an office, a high level of sound absorption may be required to keep noise levels tolerably low. When designing motorway barriers, material which does not transmit sound it absorbs is desirable since it keeps traffic noise low in surrounding areas. In a concert hall, a level of sound reflectivity which complements music is designed for.

Acoustic engineers will always factor in the materials present in a space whenever they are recording or playing audio. In movies, sound effects are added which match the acoustics of the recorded environment. This is easy to do since it is in

post-production. A more complex problem exists for the generation of audio in MR environments. How can a virtual sound be in sync with a real room if the room parameters are not known to sound designers beforehand?

2.7.3 Creating Immersion with Mixed Reality Audio

Immersion is an important aspect of creating a realistic and satisfying multi-media experience [50]. The term immersion refers to the feeling of being cut off from reality, as if spatially located in the environment with which one is interacting [51]. Both visual and auditory aspects of interactive media are carefully designed to increase immersion. The extent to which someone feels immersed in an experience correlates with higher use of visual, auditory and mental facilities [51]. It is a delicate state of mind which is easily broken if the user or listener receives cues from outside of the digital experience.

In the case of Mixed Reality, one way in which immersion could be broken is if the audio cues do not match the surrounding real environment. For example, if a user was located in a carpeted bedroom but the audio was recorded in a large wooden hall, the user would not perceive the audio source as being located in the room with them.

Intelligent audio design is a useful tool for creating immersion. In the context of entertainment media, the use of spatial audio techniques create a more interactive, realistic experience. Spatial audio can be used to emphasise objects in motion, by modulating their audio in relation with the degree of movement. Spatial audio can also be used to give cues about an environment's geometry. Large rooms can be made to sound as such by increasing the reverberation time of sounds created in that space. It is the spatial aspect of Mixed Reality audio which this project seeks to improve upon.

The system described in Chapter 4 is designed to estimate room materials.

This information could be used to infer room acoustic properties by MR devices. Chapter 5 details an embedded systems based demonstration of this material segmentation work, proving its potential to be deployed on low powered hardware such as an MR headset.

2.8 Image Segmentation using Deep Learning

Image segmentation involves the partitioning of an image into sub-regions based on some measure of information in the image. Semantic image segmentation in particular describes the process of assigning a class label to each pixel in an image. Semantic segmentation is a computer vision problem loosely based on classification. It can be thought of as a progression from coarse classification to fine grained inference. Many popular deep learning based classification networks form the basis of image segmentation pipelines. In this work, a modified GoogLeNet [18] architecture is used to infer coarse segmentations of input images.

CNNs trained in an end-to-end fashion learn hierarchical abstractions of data [26]. This makes them ideally suited to high-level computer vision tasks such as object detection/recognition. Low level tasks such as semantic segmentation are hampered by this aspect of CNNs since they require fine grained localisation rather than abstraction. Repeated combinations of max-pooling and down-sampling operations give CNNs an advantage in in-variance, but reduce localisation accuracy [52].

Most image segmentation techniques employ a Conditional Random Field (CRF) model to compensate for the limited spatial accuracy of CNNs. CRFs refine coarse segmentation results from fully convolutional CNNs. They combine class scores from multi-class classifiers with low-level information such as local edge interactions. In [53], a fully connected CRF model is used to output per-pixel classifica-

tions based on results from a fully convolutional CNN. This CRF model outputs a pixel class based on the value of every other pixel in the image. Usually several iterations of CRF calculations are performed before an acceptable solution is produced.

For the work described in this thesis, the CRF model was not implemented. The reason for the omission being that spatial accuracy is not critical for the task of estimating absorption coefficients. Realistically, a far larger margin of error results from the material classifications rather than their spatial accuracy. In theory, a more accurate material segmentation could produce a more accurate whole-room absorption coefficient result. However, the trade-off between model complexity and accuracy resulted in the CRF model not being deemed worthwhile to implement.

For the use-case of this work, inference speed is more critical to optimise than spatial accuracy. This model is designed to be deployed on an embedded device with limited computational resources for performing inference. A model with fewer parameters will execute faster, which provided another reason for discarding the CRF model. A CNN with a reduced number of parameters could be investigated as an avenue for future work on this task.

2.9 Edge-AI

Edge-AI refers to artificial intelligence technology which is located on a network edge or end device. In most cases, this refers to low-powered embedded devices like phones, smart-sensors or other consumer devices. In an Edge-AI system machine learning algorithms are run on local hardware using data generated by the same device. No internet connection is required to send data to and from a cloud server for analysis. Edge-AI devices are designed to act independently of any remote hosts to carry out tasks. Devices supporting such autonomously acting AI will

become more prevalent in coming years. Growth will be driven by the increasing number of applications requiring data processing by machine learning algorithms.

Edge-AI technology is already widespread in the consumer device market. Devices such as smartphones, gaming consoles, and smart assistants have expanded their capabilities using machine learning algorithms. Many applications requiring real-time data processing cannot afford the latency introduced by sending data to a cloud server. On device data processing cuts out this delay. Other factors which have instigated the development of Edge-AI are explained in the following section 2.9.1.

Many other markets are experiencing an influx of Edge-AI technology. Processing speed is of upmost importance in hard real-time systems such as self-driving cars where delayed decisions can result in serious harm or death for humans. Designing models to achieve fast performance and high accuracy on environment perception for autonomous driving is an active area of research. Deep Neural Network based object detection algorithms have surpassed human levels of accuracy [54] and are being applied to autonomous driving scenarios [55]. Data processing will need to be performed locally to meet hard real-time performance requirements of autonomous driving systems. Models will have strict power consumption and size limitations to be feasibly run on the embedded devices targeting the automotive market. SqueezeDet [56] is an object detection model designed to meet the performance requirements of autonomous driving and be deployed on an embedded system. It is an extremely small model (4.72MB) designed to fit entirely on an embedded device SRAM. This avoids performing DRAM operations which require over 100 times more energy [57].

Medical industries will benefit from machine learning algorithms which can be deployed on local devices. Significant data privacy concerns regarding medical data can be dealt with if information never has to leave the device it's collected

on. Some disease diagnoses can be performed by machine learning algorithms with accuracy surpassing that of trained experts [58]. Using these algorithms as a first-pass check on diagnoses could potentially free up time spent on these mundane tasks. This is especially promising given that these diagnosis tasks are generally carried out by expert clinicians who could spend the saved time giving direct care to patients.

It is important to differentiate Edge-AI devices as distinct from Internet of Things (IoT) devices. IoT devices generally rely on an internet connection to send data for processing by a remote service. Edge-AI devices explicitly do not require an internet connection to carry out data processing. However, many use cases of Edge-AI and IoT are similar if not the same. As such, Edge-AI can be considered a sub-set of IoT. See Figure 2.19 for an illustration of these concepts.

2.9.1 Trends in Edge-AI

A combination of recent technology and consumer trends has created a unique opportunity for development of Edge-AI technologies. The first major trend driving growth of Edge-AI is data security. The number of cyber-attacks has risen dramatically in recent years [59]. Remote consumer devices are especially vulnerable with a 600% increase in attacks against Internet of Things (IoT) devices between 2016 and 2017 [60]. Even seemingly innocuous information like touchscreen tapping patterns can be used to identify users with high accuracy [61].

Transmitted information is open to being intercepted, even if encryption methods are used [62]. Once that information reaches its destination it can still be regarded as unsafe. Data centres are extremely vulnerable to hacking attempts, particularly 'Cloud Computing' facilities [63], or those storing financial or identifiable information. A successful breach on a centralised enterprise data centre can have an extremely lucrative payoff compared to a computer belonging to one

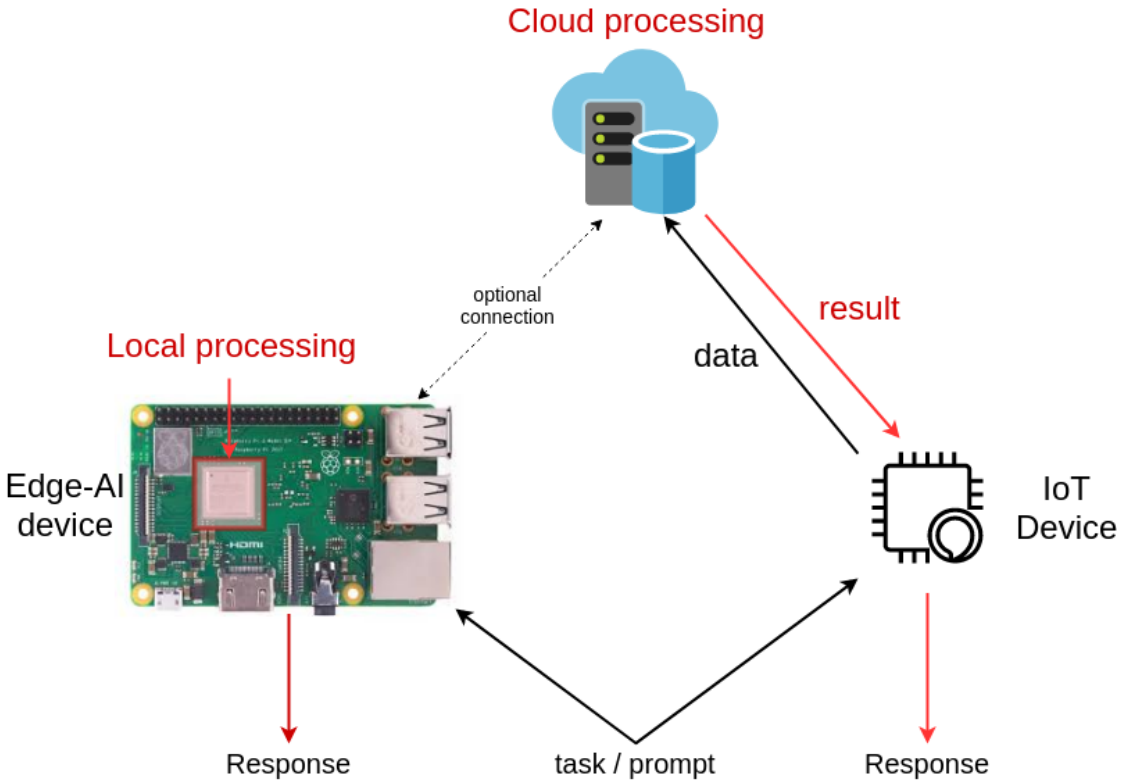


Figure 2.19: **IoT vs Edge-AI.** High level differences between IoT and Edge-AI devices. To perform tasks IoT devices rely on a remote service for data processing. Edge-AI devices carry out data processing locally. Edge-AI devices can optionally be connected to a remote service for occasional, non-critical communication.

individual. These risks have prompted a shift in how data is handled, processed and stored. In a traditional IoT device, information is sent to a remote server over an internet connection posing a potential security threat. In comparison, Edge-AI devices are advantageous as all collected data are processed and stored locally.

Strong consumer demand for 'smart' devices contributes to the growth of Edge-AI. The first generations of smart devices sent data to remote servers to be processed. As the number of such devices (and their rate of utilisation) increases, the bandwidth required to carry all the associated information will dramatically increase [64]. This will lead to network congestion and latency issues unless a large portion of data processing can be moved offline [65, 66].

Deploying AI technology at the network edge will lessen the amount of data that needs to be sent to a remote server. Novel methods have been proposed for conducting inference over distributed computing hierarchies [67]. Shallow portions of a network located at edge devices can perform localised inference with the aim of minimising communication bandwidth required. Ideally an edge device can act independently of any centralised services unless explicitly requested by a user. Methods which perform the entirety of inference computations should be preferred to avoid the network congestion issues outlined above. Hardware accelerated methods make use of specialised hardware setups to perform inference in an optimised fashion. In work by Tsung *et al.* [68], a dedicated parallel AI processor is used for power efficient computation. Memory access operations are modified to reuse data locally and avoid DRAM access operations. This has the desirable effect of reducing power consumption and heat generation. Field Programmable Gate Array (FPGA)s have the potential to gain traction as a machine learning processing platform as a GPU alternative [69]. They provide higher performance per Watt power than GPUs [70] and are inherently re-configurable. More hardware optimisation methods for Edge-AI applications are discussed in Section 2.9.2.

Software accelerated methods use existing hardware present on an edge device for machine learning model inference. Model compression is often used to achieve power savings and performance improvements [71]. Other approaches aim to replace parts of network architectures with more energy efficient accelerators [72]. Such examples may have an advantage where hardware longevity is concerned. Distributing improvements to software accelerated methods may postpone the need to replace the hardware performing inference computations.

Gathering of user data is also an common trend which Edge-AI devices can play a role in. The collected data can be used to fine tune a neural network model to personalise it to a user. Or data could be sent to a central location to train algorithms from scratch. While this may conflict with the theme of data security, the importance of data collection for developing neural network models must be acknowledged. Without the enormous quantities of data generated by social media and consumer devices many recent advances in machine learning may never have been implemented. Ground-breaking models can exist as theoretical concepts but most need massive quantities of training data to be implemented in practice.

Improvements to low-power processor performance have enabled the development of Edge-AI devices. Low power consumption is highly desirable for most embedded devices, especially those which are battery powered. Edge devices may be designed to run for many months or years on a single battery. While it helps if wireless communications aren't required for data transfer, a large amount of power consumption is usually from moving data from memory to a processor [73]. Optimisations to processors targeted at embedded devices have made it possible for AI to be placed at network edge devices. These improvements are discussed in detail in the following Section 2.9.2.

2.9.2 Edge Hardware

Performing neural network inference on a CPU is a slow and inefficient process as they are optimised to run complicated processes in sequential order. Running inference on a CPU means that every computation step in a network is executed sequentially. Even large multi-core CPUs cannot achieve desirable inference speeds as the number of calculations in most neural networks is so enormous. On the other hand, parallelised processor architectures such as a Graphics Processing Unit (GPU) are optimised for high computation bandwidth. GPUs contain a large number of simple cores which execute thousands of computations in parallel. Convolutional Neural Networks are especially suited to execution on GPUs due to their parallel structure.

Edge-AI devices can't leverage the computational bandwidth of GPUs as they demand too much power for the typical embedded system. CPUs on low power devices also have severe performance limitations to conserve power and avoid overheating. Low-power consumption and real-time performance requirements necessitate a unique type of processor which can operate under these conditions. The Vision Processing Unit (VPU) is an emerging class of microprocessor designed to perform computer vision tasks on 'edge' devices [74]. They can perform inference at a fraction of the cost and size of a full GPU. It has been demonstrated that multiple VPUs can deliver excellent power to throughput ratios compared to plain GPU or CPU implementations [75].

VPUs are not usually intended to be the main processor in a system, rather an accelerator for vision processing tasks. However, VPUs are increasingly being deployed as 'inference engines' where CNN inference is required under hardware and power constraints. They have parallel architectures, similar to GPUs, as their intended purpose is to perform computations on large arrays such as images. Ultra-low power processors such as Intel's Myriad 2 consume as little as 1

Watt while performing up to 1TOPS [3]; enough performance to run neural network inference [76, 77]. This kind of performance to power consumption ratio is particularly attractive to industries designing processors to optimise power and thermal dissipation [77].

Intel's 'Myriad X' VPU is the successor to the Myriad 2. While many VPUs are re-purposed to perform DNN inference the Myriad X is the first VPU in production to contain a dedicated DNN hardware accelerator. This contributes to a large increase in performance as entire networks can be loaded onto this dedicated hardware.

2.10 Discussion and Conclusion

The topics outlined in this Chapter prepare the reader for the following experimental chapters in this thesis. Firstly, the history and fundamentals of neural network research were presented. An outline of neural network training followed, which covered the procedures involved in training the vast majority of neural networks. In depth descriptions of Generative Adversarial Network and Convolutional Neural Network architectures were given which relate to the work in chapters 3 and 4 respectively.

Several audio synthesis methods were summarised in this literature review relating to the work in Chapter 3. This included both traditional and neural network based synthesis approaches.

The work in Chapter 4 is prefaced by two literature review sections: one on material's effect on room acoustics, and another on image segmentation techniques. These two sections, combined with Section 2.4 on CNNs deliver sufficient context for describing an image-based method of room acoustic analysis.

[3] 1 Trillion Operations Per Second

Lastly, the *Edge-AI* computing paradigm is described in this literature review. An embedded systems based demonstration of work in Chapter 4 is presented in Chapter 5 which utilises Edge-AI technologies.

In conclusion, this chapter summarises the knowledge base the reader should be familiar with before reading the rest of the thesis.

Chapter 3

Deep Neural Networks for Audio Synthesis

This chapter documents work carried out to examine a method of generating audio exhibiting qualities of multiple classes using a Generative Adversarial Network (GAN). This work was applied to a gaming context and presented at the 2018 IEEE Gaming Entertainment & Media conference.

The term *audio synthesis* refers to the creation of audio waveforms. This chapter describes a method of creating audio waveforms using a neural network trained on samples of audio from YouTube. The particular goal of this synthesis method is to create audio which exhibits qualities of multiple targets. To achieve this goal, the neural network based synthesis model is trained on multiple target classes of audio.

The chapter starts with a description of the proposed use-case in a gaming and multi-media context. The dataset used in this work is described as well as a software toolkit which was developed to process it. Finally the experimental methodology and results associated with this work are presented.

3.1 Proposed Use-Case

The hypothesised use-case of the system described in this chapter is real-time synthesis of audio in virtual environments. Dynamic audio in multi-media experiences is a new area of research with a large scope for new contributions. Motivating this work is the idea to synthesise audio for objects which don't match the identity of any object previously seen in the virtual or real environment. How would an object which was previously completely foreign to the mixed reality environment sound to a listener? The human imagination might approximate a sound based on similarities to objects it has seen and heard before. This projects investigates whether deep learning can act with such imagination-like characteristics.

While only the methods directly involved in audio synthesis are described in this chapter, it was hypothesised what kind of system could be built around it. An object recognition model trained on a very broad dataset of objects could be used to estimate what object classes a 'foreign' object looks most similar to. For example, the top two highest scoring categories could be used as an input to the audio synthesis model. These results would inform the model which two classes to 'blend' in order to create a new and unique sound. In such a setup an audio synthesis GAN would need to be trained on audio from the same classes as the object recognition model was trained on.

In our experiments, the GAN was trained on two audio classes at a time for simplicity and to shorten training time. It was hypothesised mid-way through the project that a GAN trained on more than two classes could be used in combination with targeted latent space sampling to blend more than two classes of audio. This avenue for future work is discussed in Section 6.2.1.

3.2 Methodology

This section describes the methodology used for experiments on WaveGAN [32] using the AudioSet dataset [12]. Sub-sets of AudioSet are used to train a WaveGAN model. Training sets include examples of two audio classes with the aim of synthesising samples with attributes of both classes.

The steps involved in processing the AudioSet dataset are described in Section 3.2.3.1. Training a WaveGAN model is described in Section 3.2.5.

3.2.1 Experimental Set-up

The experiments on training WaveGAN models were carried out on a server running Ubuntu 16.04 OS. Two GPUs were used simultaneously for training (Nvidia 1080Ti). The Tensorflow machine learning framework [78] is used to train WaveGAN models. Other software dependencies include;

- Python 2.7
- CUDA, cuDNN (versions matching GPU hardware)
- Python libraries (numpy, scipy)

3.2.2 AudioSet Dataset

AudioSet [12] is a large scale collection of 10-second sound clips from YouTube videos. AudioSet is analogous to the ImageNet dataset [1] in computer vision research. The goals of AudioSet are to improve the state-of-the-art in audio event recognition much like ImageNet drove developments in the computer vision field. Before its creation, no audio dataset of its size existed or represented as diverse a range of audio events. Containing over 5,000 hours of audio in total, this dataset

was chosen to train WaveGAN from scratch on new data. Figure 3.1 compares several large audio datasets in terms of total hours of audio.

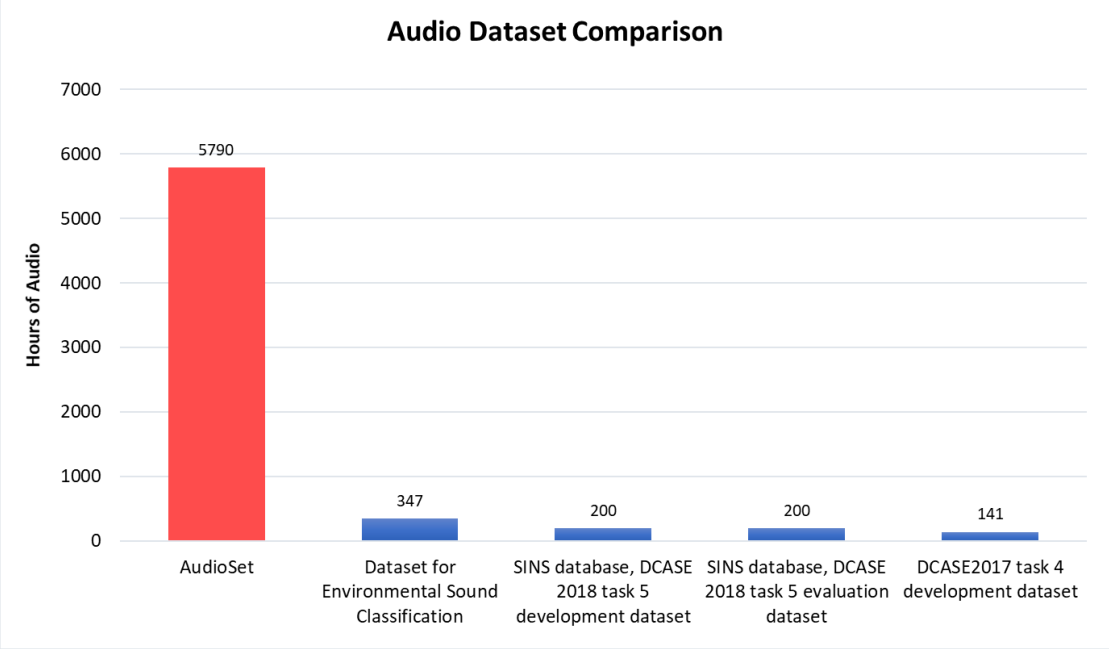


Figure 3.1: **Hours of audio in large audio datasets.** Data gathered from `http://www.cs.tut.fi/~heittolt/datasets`.

AudioSet contains a diverse range of audio event classes. Figure 3.2 compares the number of audio event classes contained in a number of audio datasets. Some audio event classes in AudioSet contian over 100 hours of audio alone.

The data in AudioSet is gathered directly from YouTube videos. Human annotators were presented with 10-second long video clips and asked to confirm the presence of one or more audio classes. A single 10-second clip could have multiple audio events occur. Therefore it is possible for a single audio sample in the dataset to have multiple class labels. An audio event does not have to span the entire clip for it to be labelled.

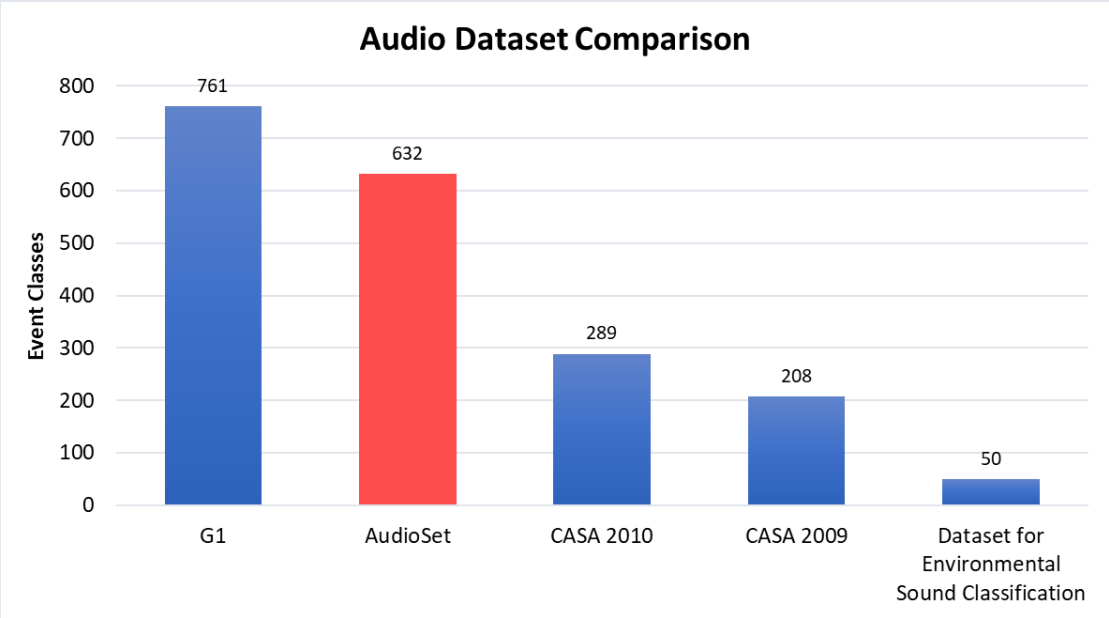


Figure 3.2: **Number of audio event classes in large audio datasets.** Data gathered from `http://www.cs.tut.fi/~heittolt/datasets`.

3.2.3 Data Pre-processing

3.2.3.1 Gathering Data from AudioSet (AudioSet Toolkit)

The first step in the training process is to gather training data. AudioSet sub-classes contain anywhere between 120 and 1 million examples, depending on the sub-class chosen. The amount of audio data used for experiments in this chapter was chosen to be in the same order of magnitude as that of the original WaveGAN experiments in [32]. Two hours of audio per class was chosen as a reasonable amount of training data to train a WaveGAN model.

AudioSet is also available as a list of YouTube-IDs structured as CSV files. The problem with using YouTube-IDs is that they are only references to where the audio can be found online, not the samples themselves. However, using these identifiers is the only way to obtain raw audio to train a WaveGAN model for this

project. Timestamps are also supplied with the YouTube-IDs so that 10 second samples can be identified.

Manually gathering all samples for an entire class from a CSV file would take significant time and be prone to human error. It would involve a number of lengthy steps which would have to be repeated every time a new data needed to be downloaded;

1. Parsing the CSV dataset for samples labelled with corresponding class identifier
2. Storing YouTube-IDs labelled with class identifier.
3. Putting all IDs into a separate URL addresses.
4. Downloading YouTube video from which a sample originated
5. Extracting audio, discarding video stream.
6. Using timestamp information in CSV file to retrieve sample.
7. Storing sample on local machine.

Since these steps are repeatable for downloading any target class in AudioSet, it made sense to automate this process. A toolkit for downloading the raw audio samples in AudioSet was developed to solve this problem. The toolkit comprises of a set of Python scripts for taking user input, parsing through the dataset, and downloading the relevant audio clips. Figure 3.3 illustrates at a high-level the download function of the toolkit.

To download a sub-set of AudioSet, the user must specify one or more target classes. Then the CSV files distributed for the dataset are parsed for all YouTube-IDs which have a label associated with the given class (or classes). Using a number of Python packages, URLs are formed with the YouTube-IDs. Ten second audio

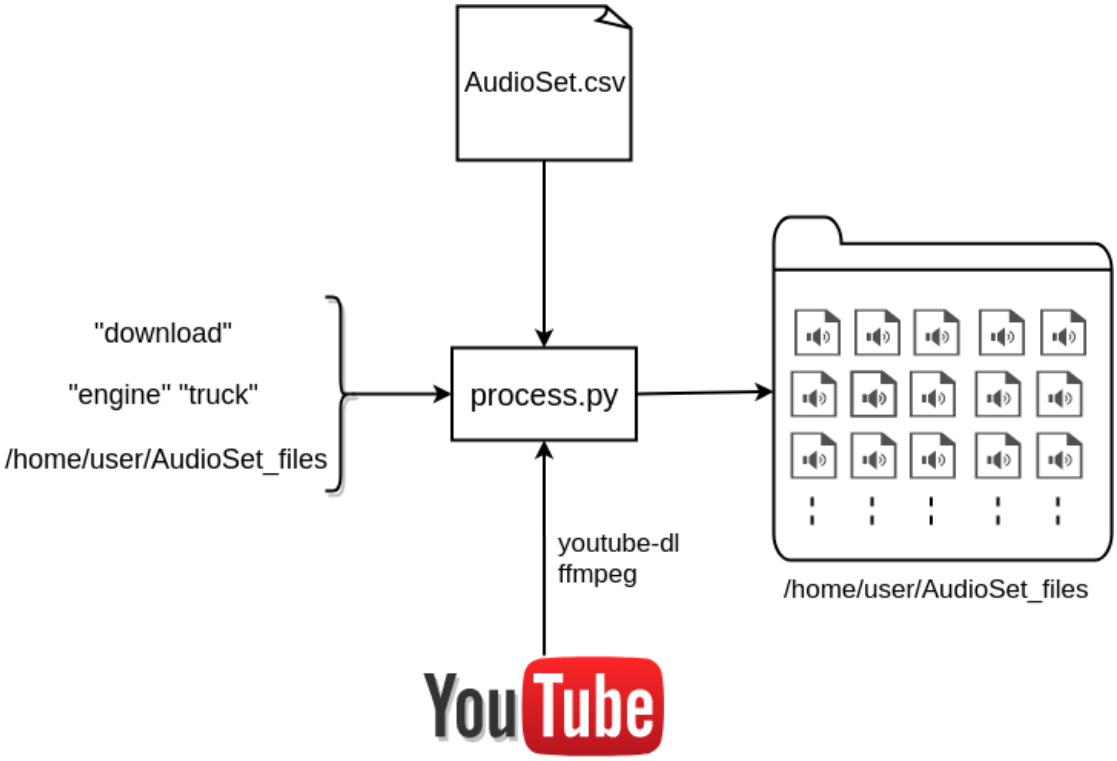


Figure 3.3: Example user inputs to the toolkit are shown on the right. The *process.py* script parses a CSV file for entries matching the input classes. URL addresses are generated with matching entries, and used to download audio from YouTube. Command line tools *youtube-dl* and *ffmpeg* are used to download audio. The samples are stored locally in a path specified by the user.

clips are downloaded using the generated URLs and corresponding timestamps for each video. Clips are stored locally on the user's machine for subsequent use.

Figure 3.4 illustrates the downloading functionality of the toolkit as a flowchart. A file called *process.py* controls the downloading process. It takes as input the desired class label, a destination path to store files in and the string 'download' to specify this operation.

The toolkit has been made publicly available by the author of this thesis. The full source code is accessible on GitHub [1]. Detailed documentation on using the

[1] `https://github.com/aoifemcdonagh/audioset-processing`

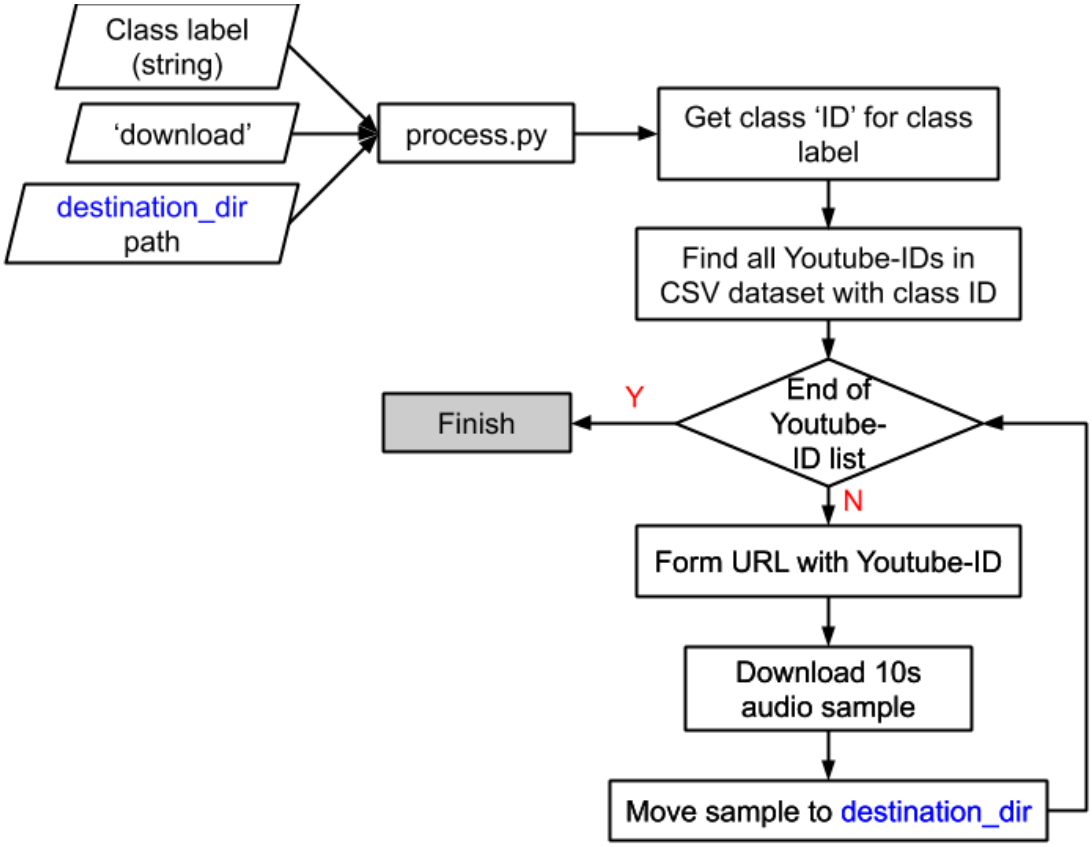


Figure 3.4: Flowchart of downloading samples from AudioSet using toolkit developed in this work.

toolkit for downloading AudioSet and its other functionalities outside the scope of this work are outlined there.

3.2.3.2 Creating TFRecord Files

Once training data has been downloaded as *.wav* files using the downloading toolkit described in Section 3.2.3.1, it must be converted to a serialised format known as a *TFRecord* file. TFRecords are a binary storage format designed for the Tensorflow framework. Using a binary data storage format has advantages in speeding up the DNN training process. Binary data takes up less space in memory and is faster

to read and copy. This advantage is amplified when data volumes are of the scale required for training deep neural networks.

The conversion process is carried out using the Tensorflow framework. Firstly paths to training data files are found and shuffled. *Slices* are created by breaking individual training samples down into smaller pieces of data. A placeholder for sample slices is instantiated. This placeholder acts as a variable that data will be assigned to later in the execution process. This is common practise in the Tensorflow framework. The choice of slice length is important as it corresponds to the length of each WaveGAN training sample. Samples are iterated through in blocks and saved in binary format TFRecord files. Each block of samples represents the contents of one TFRecord file. Python code used to create TFRecord files for the experiments in this chapter is included in Appendix C.

Figure 3.5 shows the process of creating a dataset of TFRecord files as a flowchart. Appendix C includes a similar diagram with code snippets included for steps in the TFRecord file creation process.

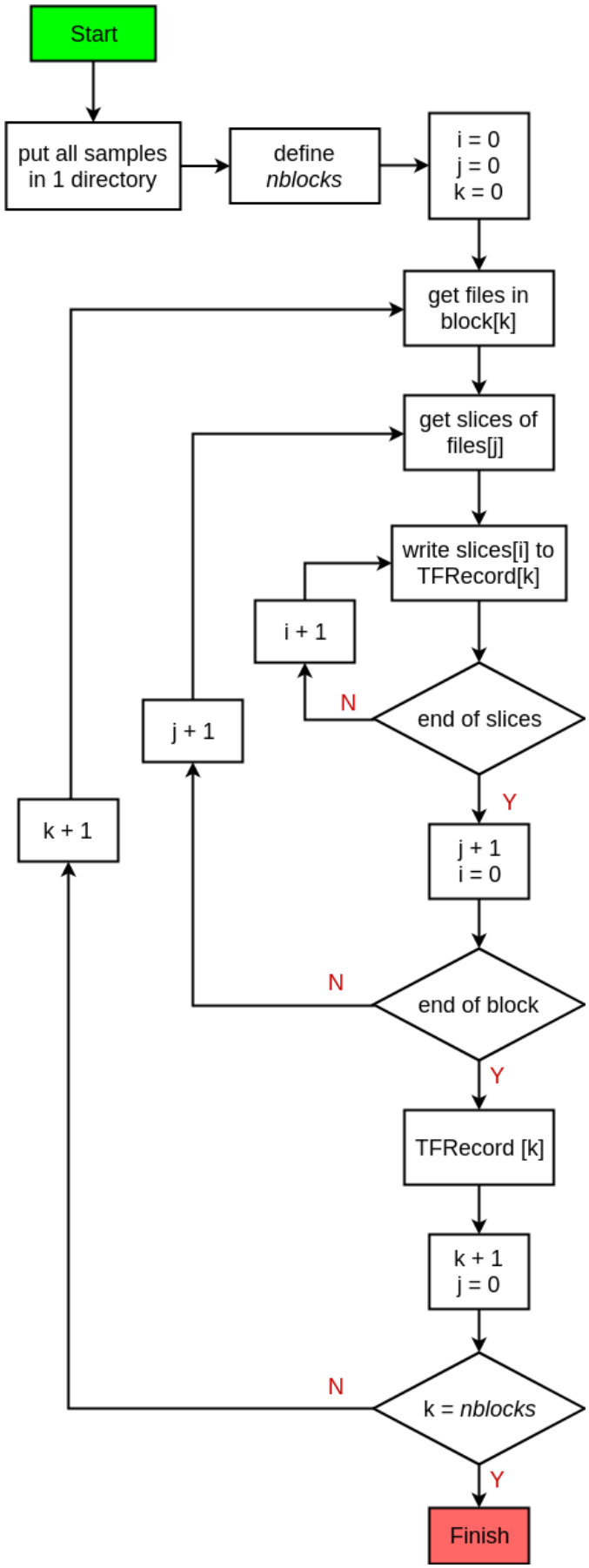


Figure 3.5: **Creation of TFRecord files.**

3.2.4 Duplication of WaveGAN Experiments

The experimental steps outlined in [12] are followed to verify the capabilities of the model. No hyperparameters are modified, and training is run for the same number of iterations (200000). For this experiment, training of a WaveGAN model is carried out on a subset of the Speech Commands Dataset [79].

3.2.5 Training WaveGAN

In this experiment, a WaveGAN model is trained from scratch using sub-classes of the AudioSet dataset. Two audio classes are included in the training data with the aim of training the GAN to synthesise audio exhibiting qualities of both classes. After network training the generator model is sampled using random latent space vectors.

3.2.5.1 Network initialisation

Training *'from scratch'* means that the network weights are initialised randomly, drawn from a particular distribution, or are all zeros, rather than the results of prior training. In this experiment weights are initialised using the Xavier initialisation technique [80]. This method of weight initialisation draws weight values from a distribution with zero mean and a specific variance. The variance is chosen based on the number of units in the input and output weight tensors of each layer. Equation 3.1 is the general equation for calculating the variance, were n_{in} is the number of weights in the input weight tensor and n_{out} the number of weights in the output weight tensor for layer i.

$$Var(w_i) = \frac{1}{n_{in} + n_{out}} \tag{3.1}$$

The Tensorflow implementation of Xavier initialisation draws weights from a

uniform distribution between $[-limit, limit]$ where *limit* is defined as per Equation 3.2.

$$limit = \sqrt{\frac{6}{n_{in} + n_{out}}} \tag{3.2}$$

Choosing weights using Xavier initialisation is far superior to initialising weight values randomly. Naively initialising weight values can slow network learning, and lead to sub-optimal solutions. This is undesirable in the case of training GANs which are notoriously unstable and difficult to train.

3.2.5.2 Running training

Training a model using the Tensorflow framework requires the creation of a *graph* which represents the model's parameters and connections. Once placeholder variables for the graph are instantiated, the graph's computations are run with training data. Executing previously created variables is known as running a Tensorflow *Session*.

In the case of WaveGAN we first initialise empty variables for ground truth and latent space input data.

```
# ground truth data. input window length = 16384
x = loader.get_batch(tfrecord_files, batch_size, 16384)
# latent space vector (length = 100, range [-1,1])
z = tf.random_uniform([batch_size, 100], -1., 1., dtype=tf.float32)
```

Then the generator and discriminator models are created as variables which will be executed later. Both are custom Python objects written by the authors of WaveGAN [2]. All model layers and connections are defined as a graph which is

[2] https://github.com/chrisdonahue/wavegan

executed in a Tensorflow session.

```
# Make generator graph
with tf.variable_scope('G'):
    G_z = WaveGANGenerator(z, train=True, **args.wavegan_g_kwargs)
G_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope='G')
```

Two discriminator models are created during training. One discriminator classifies whether samples from ground truth data are real or fake. The second discriminator classifies samples from the generator as real or fake. Since the goal of training WaveGAN is to achieve good synthesis results, it doesn't much affect the end result if there are actually two discriminators used during training.

```
# discriminator for 'real' samples
with tf.name_scope('D_x'), tf.variable_scope('D'):
    D_x = WaveGANDiscriminator(x, **args.wavegan_d_kwargs)
D_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope='D')
# discriminator for generated samples.
with tf.name_scope('D_G_z'), tf.variable_scope('D', reuse=True):
    D_G_z = WaveGANDiscriminator(G_z, **args.wavegan_d_kwargs)
```

Lastly, the loss function and the optimiser are instantiated. In the experiments carried out in this work WGAN-GP training method [81] is used. This method imposes a *gradient penalty* in loss calculations to improve the stability of GAN training which is notoriously prone to collapse. The code for generating the loss, gradient penalties and optimisers is included in Appendix D. The loss variables for

the Generator and Discriminator are instantiated as *G_loss* and *D_loss* respectively.

The final operations are instantiated which combine all previous elements. They are run in a Tensorflow 'Monitored Training Session'. Starting this session executes all the previously un-executed operation placeholders. The monitored training session takes as arguments:

- *checkpoint_dir* directory path to save model checkpoints
- *save_checkpoint_secs* interval for saving model checkpoints in seconds
- *save_summaries_secs* interval for saving summaries in seconds

For each Generator model update the Discriminator updates five times. This is managed by a for loop and an argument to specify the update ratio.

```
# Operation for training Generator
G_train_op = G_opt.minimize(G_loss, var_list=G_vars,
        global_step=tf.train.get_or_create_global_step())
# Operation for training Discriminator
D_train_op = D_opt.minimize(D_loss, var_list=D_vars)

# Start training
with tf.train.MonitoredTrainingSession(
        checkpoint_dir=args.train_dir,
        save_checkpoint_secs=args.train_save_secs,
        save_summaries_secs=args.train_summary_secs) as sess:
    while True:
        # Train D 'nupdates' times per 1 G update
        for i in xrange(args.nupdates):
            sess.run(D_train_op)
```

```
sess.run(G_train_op) # 1 G update per 'nupdates' D updates
```

The steps involved in training a WaveGAN model from scratch are illustrated in Figure 3.6.

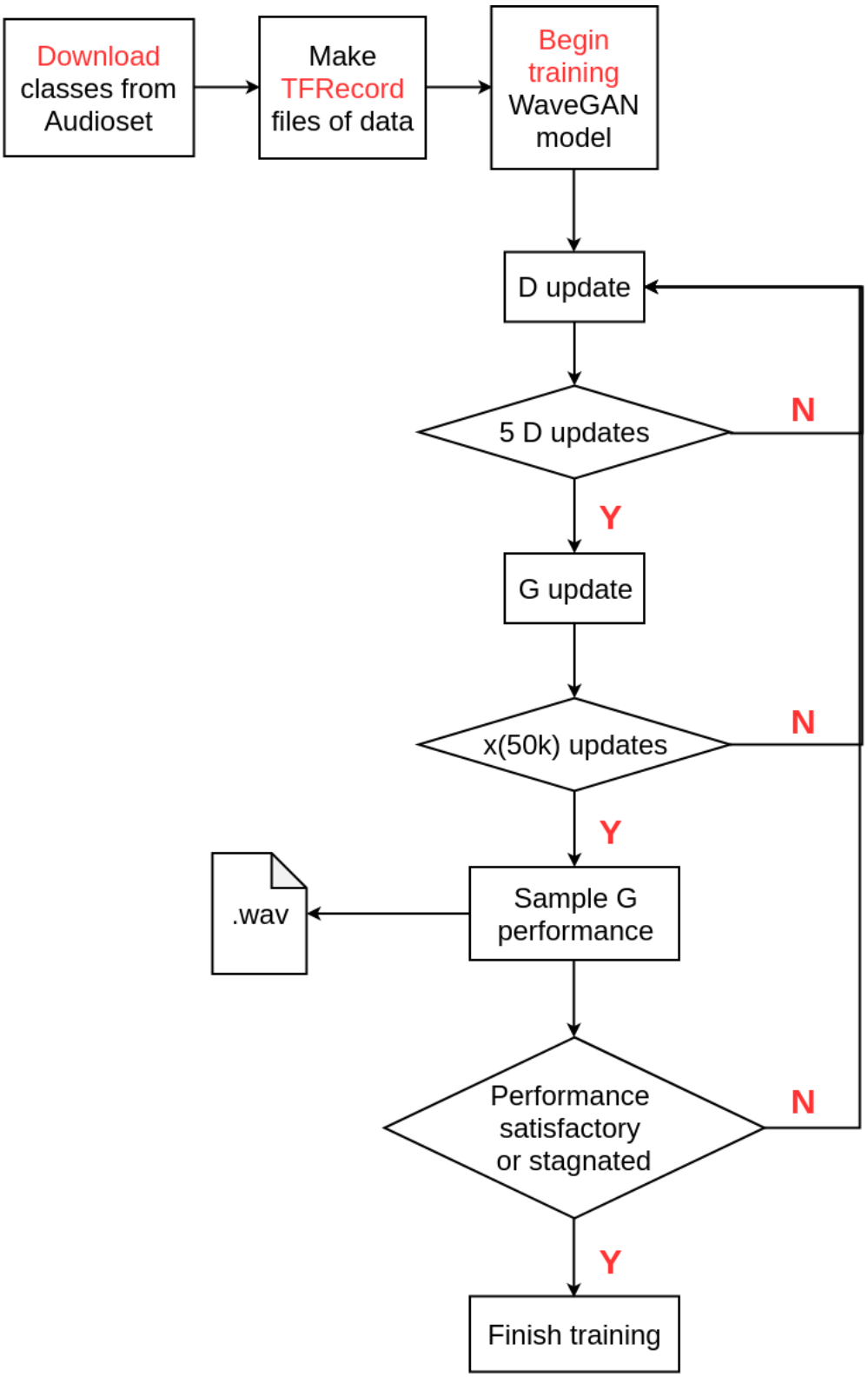


Figure 3.6: **Training WaveGAN.** Flowchart illustrating steps in training a WaveGAN model on AudioSet sub-classes. The toolkit described in Section 3.2.3.1 is used to download training data. *'D'* refers to the discriminator model and *'G'* to the generator model. An *'update'* refers to one training iteration.

3.2.6 Sampling WaveGAN

After a WaveGAN model has been trained, the generator portion can be executed to produce audio waveforms. Each generator output is 16384 samples long, or approximately 1 second at 16kHz.

The first step in this process is to load a trained model file. Usually this is a checkpoint which has been generated during training. A checkpoint is a copy of the model weights captured during the training process. Checkpoints are used to evaluate model performance during training or taken as the final model if performance is satisfactory.

- *meta_file_path* is a path to a *metagraph* created during WaveGAN training.
- *checkpoint_path* is a path to a model checkpoint created during WaveGAN training.

```
tf.reset_default_graph() # Load the graph
saver = tf.train.import_meta_graph(meta_file_path)
graph = tf.get_default_graph()
sess = tf.InteractiveSession()
saver.restore(sess, checkpoint_path) # Refer to model checkpoint
```

The generator is given a random latent space vector as input. Multiple latent space vectors can be created and run through the model at the same time to produce multiple samples.

```
# Create 50 random latent vectors z
_z = (np.random.rand(50, 100) * 2.) - 1
```

Next, placeholders are initialised for model graph *Tensor* objects. These tensors are evaluated by a Tensorflow *Session*. The Session object encapsulates the environment in which *Tensor* objects are evaluated. When the Session is run, the placeholder is set to the random latent space vector which was generated previously.

```
z = graph.get_tensor_by_name('z:0') # placeholder for input tensor
G_z = graph.get_tensor_by_name('G_z:0') # placeholder for output tensor
_G_z = sess.run(G_z, {z: _z}) # Synthesise G(z)
```

The Session evaluates the output tensor, in this case resulting in multiple audio samples. Samples are written to files for further use.

- *audio_file_path* path to a *.wav* file to write sample to
- *16000* sample rate of output sample
- *_G_z[0]* the first item in the evaluated Generator model output Tensor.

```
scipy.io.wavfile.write(audio_file_path, 16000, _G_z[0])
```

This method is used to produce sample results for evaluation.

3.3 Results and Discussion

The quality of generated audio varies for each set of data used to train the model. Generally, output from a model trained on data from a single class is superior to output from a model trained on data from multiple classes. It is unsurprising that there would be some difficulty with reproducing samples from a data set where

there are multiple classes present but no explicit labels associated with them. Exact reproduction is also complicated by the random input vectors, which means the same dataset could result in a slightly different set of sounds each time a network is trained on them.

Another factor that contributes to the output of the model is the quality of the training data. The numerical digits subset of the Speech Commands Dataset [79] (SC09) is a set of high quality data with little background noise. This was one dataset for which WaveGAN produced good results when used as training data. Each example in this set contains only one digit. On the other hand, AudioSet [12] examples can contain sound from multiple sources (classes). Problems arise when, for example, a ten-second clip contains only 1 second of the desired audio class, or when a significant amount of mis-labeled data makes it into the set used for training. These factors could contribute to the noisy, somewhat chaotic sounding output from models trained on data from AudioSet.

Due to the shortcomings in qualitative evaluation metrics, the best method of evaluating generated samples is currently to listen to them. A more rigorous approach that does not contradict human judgement has yet to be developed and remains an important aspect of future work.

A subjective scoring of each models output was performed by a human listener. A models score at a given training iteration is given by the percentage of output samples which resemble the class of data it is trained on, as judged by a human listener. The listener agreeing that an output sample could have originated from the target object is considered resemblance in this case. The scores for each model are given in Table 3.1. For the model trained on two audio classes, a pass is given when the output sample resembles either of the original classes or is considered a credible mix of the two.

Model	100k	150k	200k	250k
SC09	63%	53%	75%	-
Dog	50%	41%	28%	-
Violin	28%	28%	47%	-
Dog & Violin mix	34%	34%	44%	25%

Table 3.1: Pass rates of models at various stages of training (given in number of iterations).

An interesting contradiction occurs when considering what constitutes as a pass for the model trained on the digits subset of the Speech Commands Dataset [79]. Some examples produced by the model resemble mixtures of multiple digits. In this case, mixed words are undesirable outcomes, because a listener would expect to hear one of the ten digits in the training data. However, in the case of our mixed model, the goal is to produce samples which exhibit characteristics of all classes included in the training data. This anomaly supports the idea that GANs can be used to synthesise audio with multi-class characteristics as a desirable trait. For the purposes of comparing with the samples provided by the creators of WaveGAN [32] in the case of the SC09 dataset, a pass is considered to be an output sample which resembles a single digit appearing in this set.

Our methods, which employed a single GAN model, performed well on multiple datasets. This is an encouraging step as it is common for GANs to perform poorly when a new dataset is used without hyperparameter tuning.

3.3.1 Interpreting Unclear Synthesis Results

Generative models are challenging to evaluate due to the subjective nature of their output. The realness of a generated image or piece of audio can be judged instinc-

tively by a human observer, yet remains an almost impossible task for a machine. There are no robust techniques to objectively measure generative model output quality apart from some ad-hoc methods like Inception Scoring [82]. This method aims to measure the diversity and semantic discriminability of generated samples (images) by employing a pre-trained Inception classifier [83]. The effectiveness of this method in encouraging models to produce more realistic samples is debatable [84]. More importantly, the Inception Scoring method has fundamental flaws which are outlined in recent work [84] and by the creators of WaveGAN itself.

No concrete methods exist to evaluate how successfully a model generates realistic samples in part because it cannot be assessed how well generated samples fit the true data distribution. A generated sample may fit well with the training data distribution but appear to be of poor quality to human observers or listeners. This is likely because the true data distributions are so complex due to extremely high dimensionality. A sub-set of the training data's dimensions may be learned by a model well, and others poorly. This could practically result in synthesised samples with high realism in certain features (e.g. wrinkles on the face) but poor realism in others (e.g. hair texture or tooth shape).

Currently, the best way to analyse GAN performance is to subject generated samples to human subjective evaluation. By reinforcing a model on outputs that humans deem acceptable, hopefully it learns the features which are most important in reproducing realistic samples. For example, in the case of speech synthesis, the occasional breath or throat clearing may aid in creating a more natural sounding voice.

3.4 Conclusion

This chapter presents a method of synthesising audio using a Generative Adversarial Network. The GAN is trained on multiple sub-classes of the AudioSet dataset [12]. This results in outputs which sound "blended", or possessing attributes from multiple audio classes. Cross-synthesis is the most comparable traditional[3] method of blending audio samples.

The audio produced by this method has the potential to sound "other-worldly" or unlike anything occurring naturally, depending on the target classes used for training. Sounds which are genuinely unlike anything heard on earth may help with listener immersion where fantasy objects and characters need to be made audible. This method could be further enhanced for use in audio production, particularly for video games and film.

[3] i.e. not neural network based

Chapter 4

Material Segmentation for Estimation of Room Acoustics

In this chapter, a method of image segmentation based on material classification is described. Based on these segmentations, material absorption coefficients can be estimated. This work began with the following research question: could room acoustic properties be evaluated using a minimally invasive technique? In addition, could this technique be incorporated into a Mixed Reality device?

An image-based method satisfies the condition of being *minimally invasive* if no Room Impulse Response (RIR) measurements need to be taken. An RIR is a measurement taken to asses the acoustic properties of a room. It involves playing a loud impulse which excites all frequencies in the human hearing range (or more depending on the application). A recording is made of the impulse and for a short period afterwards as it propagates and reflects around the room.

Room Impulse Responses specifically measure reverberation, which is the summation of sound reflections in a space following an initial sound. Section 2.7 discusses how materials affect sound reflections, and thus reverberation.

The inspiration for developing an image-based method is the observation that

audio engineers can accurately estimate room acoustic properties from images alone. This ability comes from their experience in solving these kinds of problems over their careers. This chapter investigates whether machine learning can be used to solve this problem in a similar fashion, but where knowledge is garnered from data rather than years of human experience.

This work is developed with the goal of assisting audio engineers and designers working on Mixed Reality audio. Mixed Reality is defined as the combination of the real and virtual worlds [85]. This implies several scenarios; virtual objects added to reality, real objects added to a virtual environment, or simply virtual objects in a virtual environment. In the case where the real world persists, and virtual objects are added, one might question how the virtual objects should sound. As in any spatial audio setup, the sound coming from an object should match its positioning in its environment. MR has the added problem of incorporating acoustic properties of environments which are not always known or measurable. A robust method of estimating a rooms reverberation, suitable for incorporation into an MR device is needed to solve this problem.

4.1 Methodology

The methodology used in this chapter is largely based on experiments by Bell *et al.* [53] for full scene material classification. A large scale dataset of images labelled with material information, known as the Materials in Context Database (MINC), is introduced alongside the experiments by Bell *et al.* The creators of MINC train several CNN models on their dataset for the prediction of material class and semantic segmentation of large images based on material. The main difference between the experiments in [53] and this chapter is that a Conditional Random Field model was not implemented due to factors mentioned in Section

2.8. The models trained by the authors of [53] are publicly available in the form of patch classifiers [1]. As patch classifiers, the models are trained to classify material type at the central pixel of an image patch. These CNNs contain fully connected layers which limit the input image to be of a fixed size.

In this chapter, the patch classification model is modified to operate on images of arbitrary dimensions. The justification and implementation of this approach is outlined in Section 4.1.2. A GoogLeNet [18] model trained on the MINC dataset was chosen as it achieved the highest accuracy on image patch classification in the experiments carried out by Bell *et al.* [53]. Image patch classification forms the basis of the method described in this chapter, therefore the choice of the best performing patch classification model was most appropriate.

The GoogLeNet model's fully connected layers were replaced by convolutional layers using a popular technique known as *'network surgery'*. Using the Caffe [86] deep learning framework, layer type can be changed by editing the text file which describes the network's topology. The network parameters, stored as binary weights, remain unchanged since the mathematical operation performed by the new layer is the same as before.

The remainder of this section describes the methods used to segment images based on material and absorption coefficient information.

4.1.1 Experimental Setup

The experiments in this chapter were carried out on a desktop running a standard distribution of Ubuntu 16.04 operating system. A single Nvidia GPU (GeForce GTX 1080) was used to perform inference. A single GPU was sufficient since a pre-trained model was used and no further model training was required.

The experiments on full scene segmentation required the configuration of an

[1] `http://opensurfaces.cs.cornell.edu/publications/minc/`

appropriate software environment. Firstly, several software dependencies were installed to facilitate the experiments.

- Python 3 was required to run the sample code published by the authors of [53].
- The Caffe deep learning framework [86] was installed since the original models published by the authors of [53] used this framework. This allowed quick prototyping with the published pre-trained models.
- The following Python packages were required; numpy, pillow, scikit-image, argparse, OpenCV, python3-tk, matplotlib.

4.1.2 Full Scene Segmentation

In order to perform full scene material segmentation, a material classification result needs to be generated for every pixel in the image. One way to do this is to repeatedly perform inference on image patches in a sliding window fashion. Each image pixel will be the central pixel to one image patch fed through the classifier. Figure 4.1 shows an output using this technique with a stride equal to the receptive field of the CNN.

For image segmentation, a classification is required for every pixel, such that the output grid is the same dimension as the input image. Decreasing the stride of the CNN over the input increases the output dimensions. Equation 4.1 is used to calculate output dimensions where:

- a , b are output dimensions
- m, n are input dimensions
- l is length of CNN receptive field

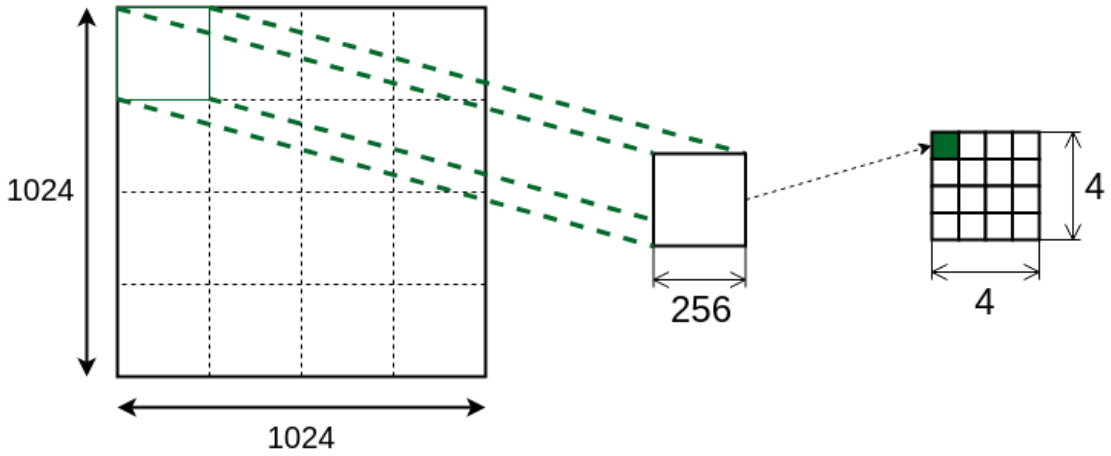


Figure 4.1: **Inefficient sliding window classification.** This diagram shows an inefficient method of performing sliding window classification across an image. Segments of the input image are fed forward through a CNN sequentially. Each forward pass generates a single classification output. The result is a grid of output values.

- p is the number of pixels of padding on the input
- s is the stride length

$$a, b = [1 + \frac{m - l + 2p}{s}, 1 + \frac{n - l + 2p}{s}] \tag{4.1}$$

For the example in Figure 4.2, the output dimensions are calculated as in Equation 4.2:

$$\text{output size} = 1 + \frac{1024 - 256 + 2(0))}{16} = 49 \tag{4.2}$$

As simple as this method sounds, it is extremely inefficient. A dramatically more efficient method is to modify a CNN's architecture such that it can take in an input of any size. The structure of CNNs allows for computations to be shared across overlapping regions. Figure 4.3 illustrates an efficient method of sliding

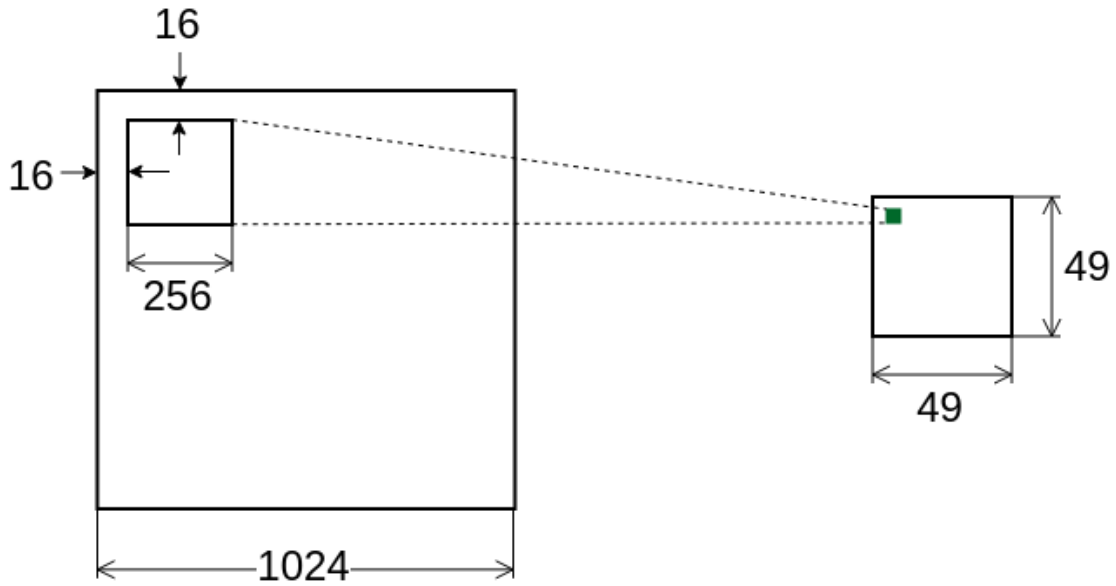


Figure 4.2: **Sliding window classification with small stride.** This diagram shows a CNN sliding over an input with stride of 16. A larger output grid is produced compared to that in Figure 4.1 since the stride is decreased.

window classification. The CNN has a receptive field of size (14,14). Any input outside this area (in green) requires extra computation.

This principle is explored in [87] where larger images are fed forward through a CNN at test time, producing a spatial output map rather than a single spatial output. This avoids computing the entire CNN pipeline for each window in an input image.

To facilitate larger input images, any fully connected CNN layers must be replaced by convolutional layers at test time. Replacement convolutional layers perform the equivalent mathematical operation as the fully connected layer. The new convolutional layer's filters are of the same dimensions as the previous layer's output during training. The number of filters is equal to the length of the replaced fully connected layer. Figure 4.4 illustrates the conversion process of a fully connected layer to a mathematically equivalent convolutional layer. At test time, the

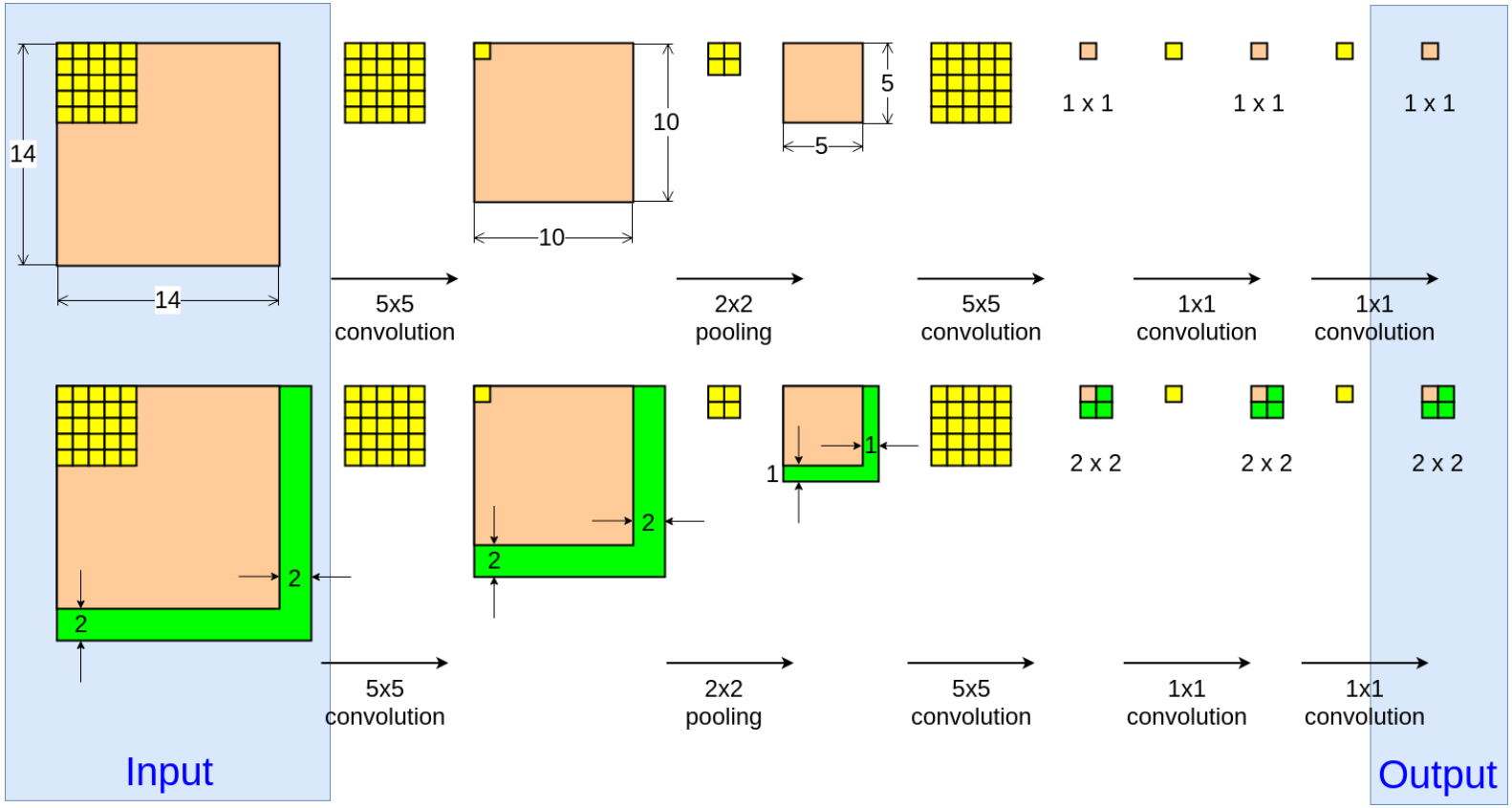


Figure 4.3: **Fully convolutional CNN.** This diagram shows how a *fully convolutional* CNN can efficiently compute a dense grid of outputs for a large input image. Convolution computations can be reused in areas where input segments overlap. This saves a significant amount of computational cost when the stride is small relative to the original CNN input dimensions. Area in green is where extra computation is required. This diagram is based on examples in [87]

network is a series of convolution and pooling layers exclusively.

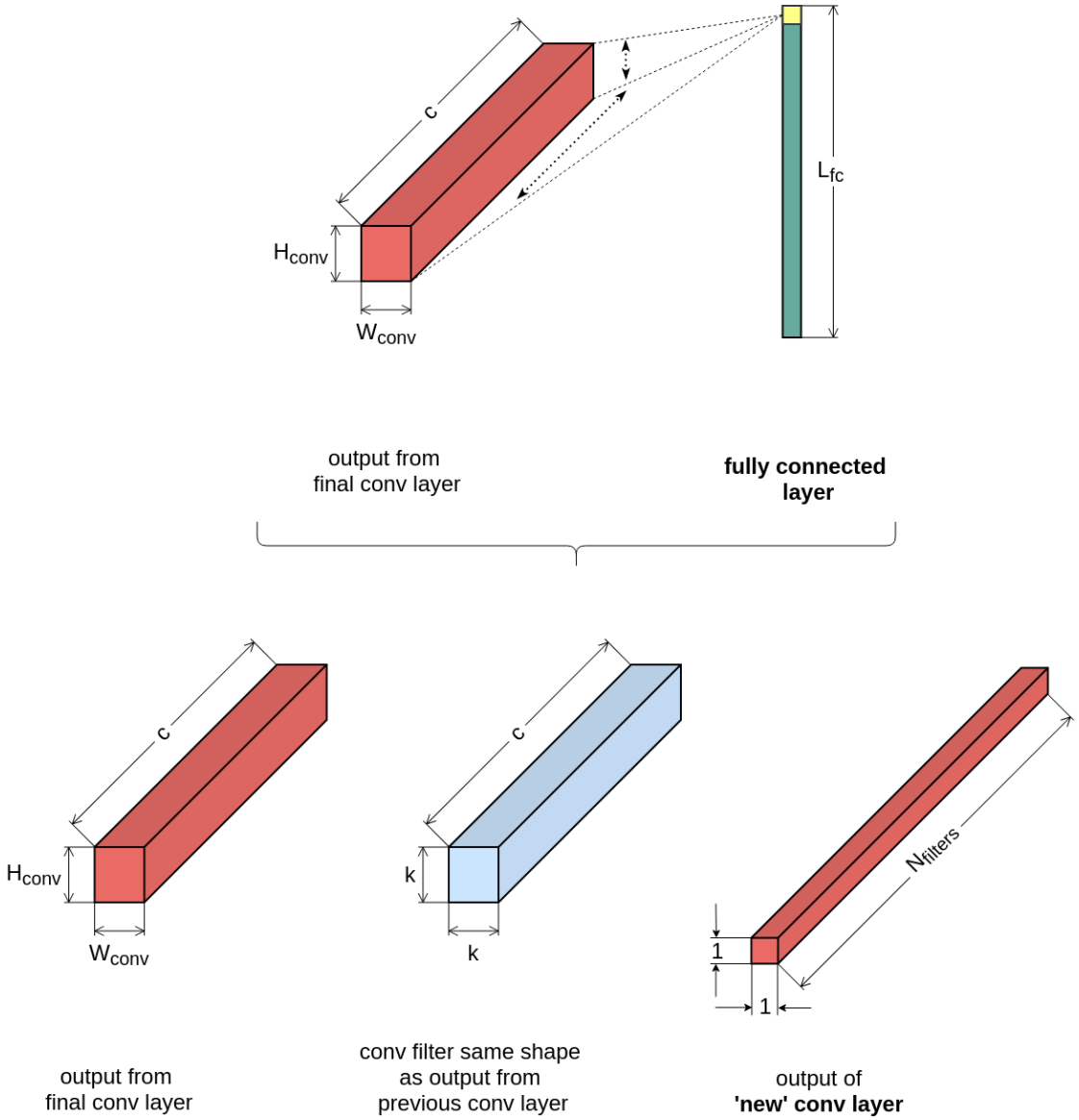


Figure 4.4: **Converting FC layer to CONV layer.** This diagram shows the process of converting a fully connected CNN layer into a mathematically equivalent convolutional layer. These modifications are performed at test time after the network (including fully connected layers) has been trained. Note that $k = H_{conv} = W_{conv}$ of the previous layer during training time. During test time H_{conv} and W_{conv} can increase according to the input image size.

To verify the process illustrated in Figure 4.4, the number of connections can be calculated to be the same using both layer types using Equations 4.3 and 4.4. This proves that the number of weights stays the same. Keeping the same weights allows this process to be carried out at test time without any retraining.

$$\text{connections}_{fc} = H_{conv} * W_{conv} * c * L_{fc} \tag{4.3}$$

$$\text{connections}_{conv} = N_{filters} * c * k * k \tag{4.4}$$

For the experiments outlined in this chapter the CNN used is made fully convolutional using the process described in this section. The base model is a GoogLeNet model [18] trained on the Materials in Context Database. This model contains a single fully connected layer to be converted to a convolutional layer.

4.1.3 Segmentation Experiments

4.1.3.1 Coarse Classification

The first experiment carried out on material segmentation was to produce coarse material class maps from input images. This experiment verified the function of the fully convolutional network described in 4.1.2. Input images considerably larger than those used to train the network were fed forward through the fully convolutional network. The results of inference on these images were processed to extract a grid of classification values. Figure 4.5 illustrates the dimensions of a hypothetical input image and output matrix. Results produced by this process on example images are shown in Section 4.2.

The dimensions of the output matrix depend on the input image size (m, n), number of pixels of padding (p) and the stride (s) used for dense classification. The channel dimension is constant for this experiment since it is the number of

classes classified by the model (23). Equation 4.5 shows the calculation of the output dimensions a, b which are illustrated in Figure 4.5.

$$a, b = [1 + \frac{m - 224 + 2p}{s}, 1 + \frac{n - 224 + 2p}{s}] \tag{4.5}$$

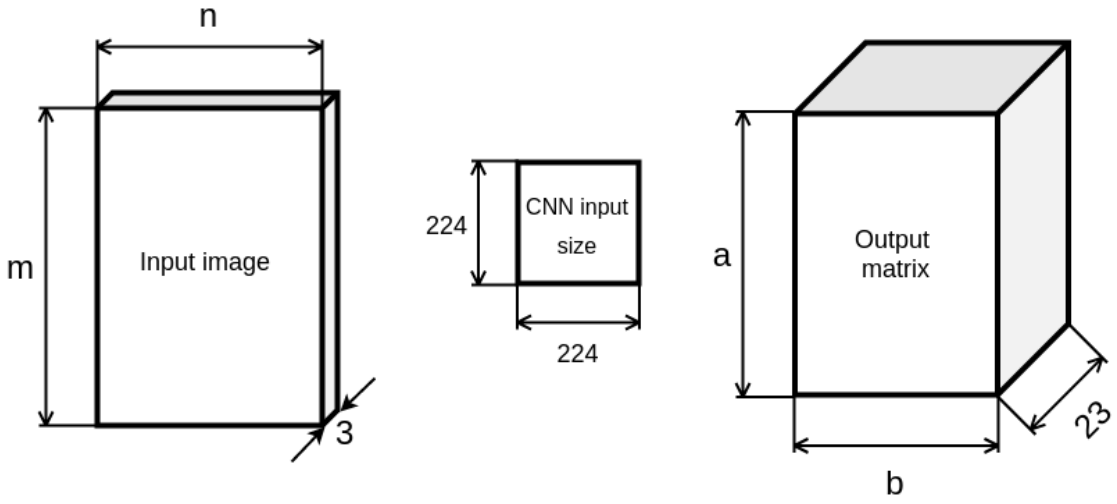


Figure 4.5: **CNN input and output matrices.** Illustration of input image dimensions, CNN receptive field dimensions and output matrix dimensions. If the input image is larger than a fully convolutional CNN's receptive field, the output of inference will be a three-dimensional array.

Each element in the output grid corresponds to an inference pass on a 224x224 window of the input image, as illustrated in Figure 4.6.

4.1.3.2 Fine-Grained Segmentation

The second part of the experiments on material segmentation involved producing fine-grained image segmentations based on material classification. The fully convolutional network described in Section 4.1.2 was used for this experiment.

The experimental process is based on a pyramidal upsampling technique. There are multiple examples of using multi-scale inputs to fully-convolutional CNNs for

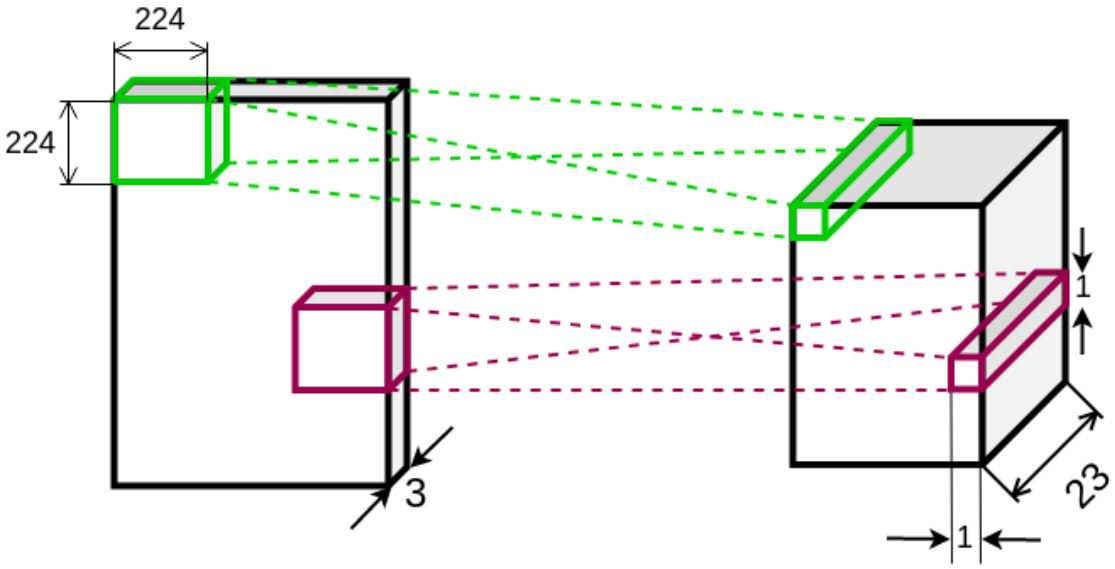


Figure 4.6: **Sliding window CNN output.** A CNN applied in a sliding window fashion over an input results in a three-dimensional output space. The output matrix comprises the outputs of CNN inference on 224 x 224 sections of the input.

image segmentation tasks [53, 88, 89]. First, the input image is resized at three scales [2]. Each resized image is passed through a fully convolutional network. The output of inference is a set of probability maps, one for each class. All probability maps are upsampled to the dimensions of the original image. The average value across the three upsampled probability maps for each class is taken to produce a final set of probability maps. The averaged probability maps are used to produce a material segmentation of the original input image. The highest probability class at each pixel is taken as the material classification for that pixel. The steps involved in image segmentation based on material classification are illustrated in Figure 4.7. Appendix F expands on the workflow to illustrate the outputs of each step in the segmentation process.

The probability maps are the key components to creating material segmen-

[2]The scales used by the authors of [53] are used in this experiment.

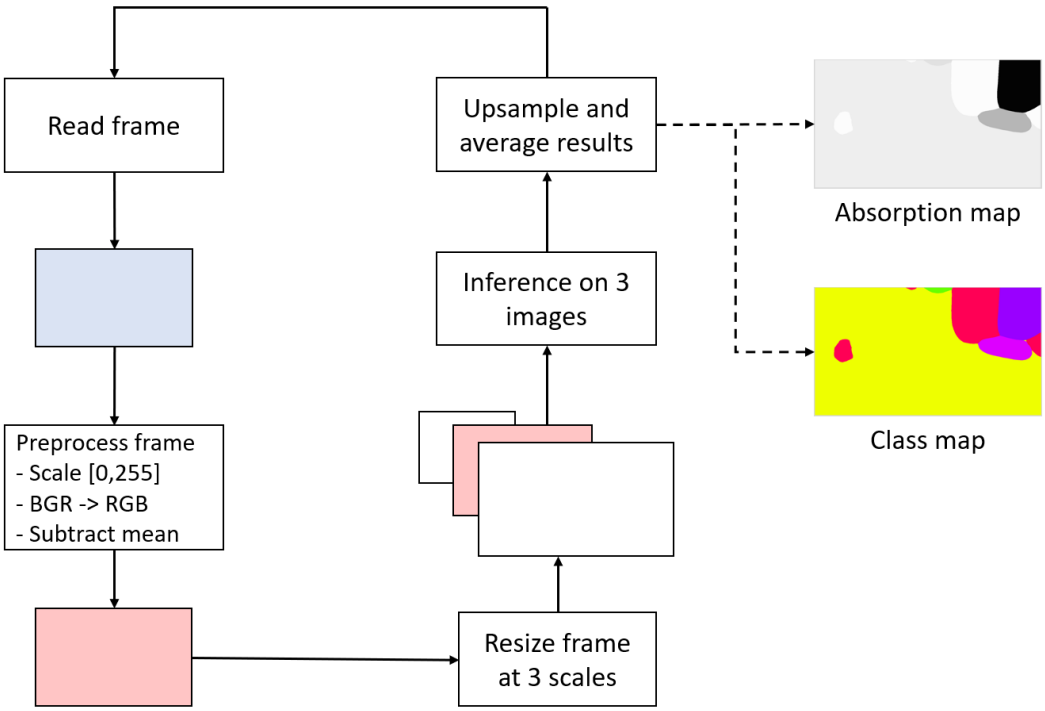


Figure 4.7: **Flowchart of image segmentation process.** This high-level flowchart shows the key steps involved in performing full scene material segmentation using the method described in this chapter. Images are pre-processed so that they are suitable for inference by a GoogLeNet [18] model using the Caffe [86] machine learning framework.

tations in this experiment. There is one probability map for each material class which can be classified. Figures 4.8 and 4.9 show the plotted probability maps used in the segmentation process. Values in the probability map always lie in the range [0,1]. Lighter pixel values correspond to a higher classification probability for a material class.

4.1.4 Modified Plotting Method

The output of the material segmentation process is a two-dimensional grid of integers between 0 and 22, representing each of the 23 material classes in the

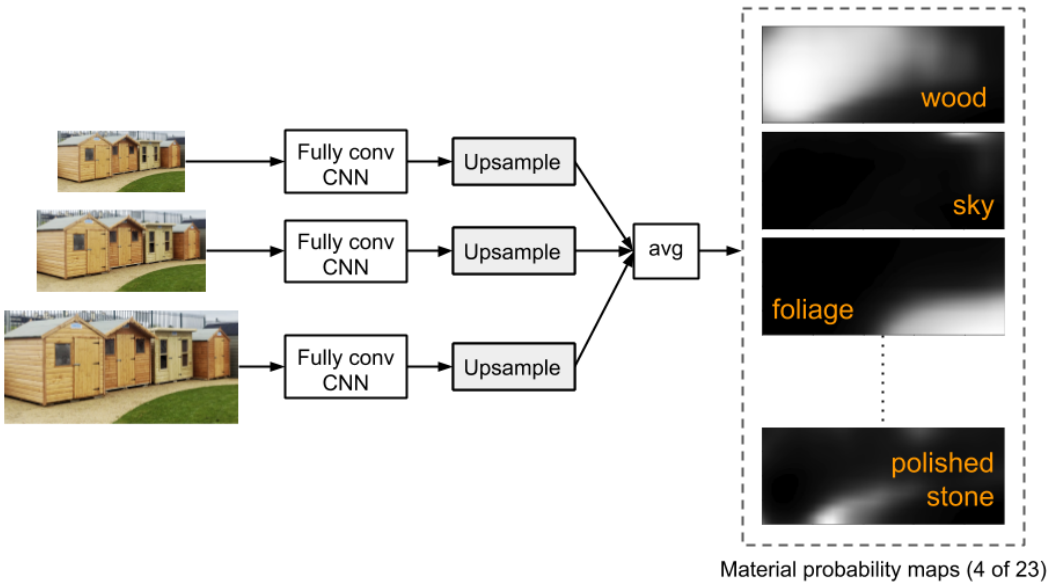


Figure 4.8: **Image segmentation process part I.** Detailed illustration of steps in the image segmentation process. The input image is resized at three scales and passed forward through a fully convolutional CNN. The resulting output maps are upsampled to the original image size and averaged. This produces classification probability maps for each material class.

MINC dataset. This grid is the same size as the original input image. Plotting this result naively with a Python plotting package will not produce meaningful segmentations. A method of plotting results was formulated which produces easily discernible material segmentations and absorption coefficient maps.

A *colormap*[3] is a scale for translating scalar values into three-dimensional colour space. Matplotlib [90], a Python plotting library used in this work, contains many built-in colormaps but none which met the unique needs of this work. In the case of material class segmentations, the colours used for materials need to be spaced as far apart on the colour spectrum as possible. Up to 23 individual could be required for any one image segmentation.

[3]US-English spelling used here since it aligns with terminology used in most plotting software packages such as Matplotlib [90] which is used in this work.

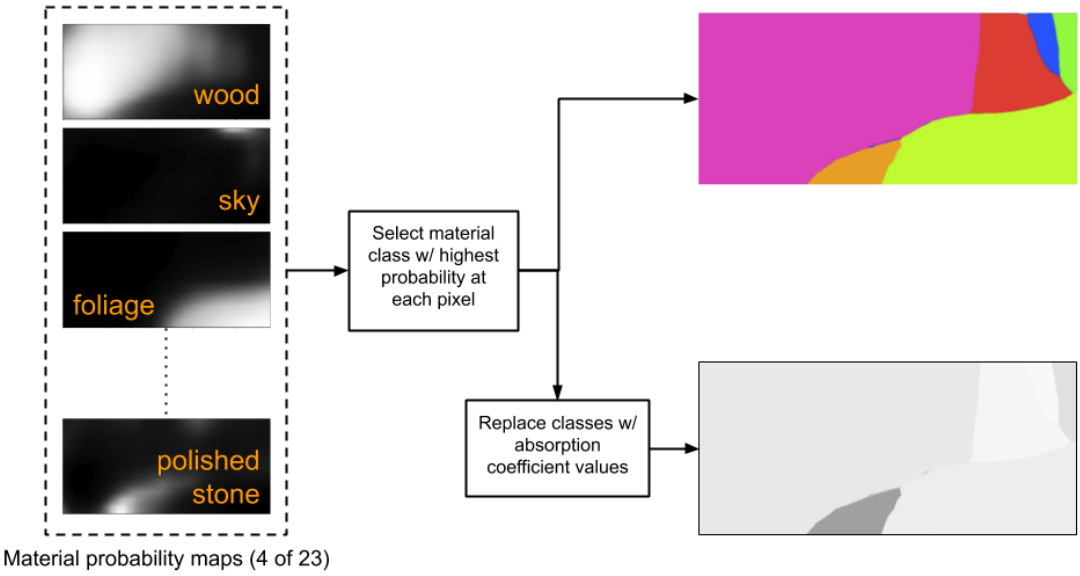


Figure 4.9: **Image segmentation process part II.** Detailed illustration of steps in the image segmentation process. The classification probability maps are processed to generate class maps and absorption maps. To produce a class map, the highest probability material at each pixel is taken as the result for that location. Swapping materials in the class map for their respective absorption coefficients produces the absorption map.

To create a colormap suitable for this work, a built-in Matplotlib is modified such that it comprises of evenly spaced colours. The number of colours is equal to the number of material types present in the image segmentation. The following code excerpt shows the process of generating a material class plot like in Figure 4.10.

```
fig, ax = plt.subplots(figsize=aspect)
# class_map is image segmentation result
# i.e. 2D grid of integer values
```

```
hb = ax.imshow(class_map, cmap=plt.get_cmap("gist_rainbow",
    len(unique_classes)))

# Create a colorbar alongside plot
# Define the step length between ticks for colorbar.
step_length = float(len(unique_classes) - 1) /
    float(len(unique_classes))
# Shift each tick location so that the label is in the middle
loc = np.arange(step_length / 2, len(unique_classes),
        step_length) if len(unique_classes) > 1 else [0.0]
cb = fig.colorbar(hb, ax=ax, ticks=loc)
cb.set_ticklabels(get_tick_labels(unique_classes))
cb.set_label('Classes')
```

4.2 Results and Discussion

The results of the experiments in this chapter are image segmentations based on material classification. Absorption maps are produced from the material segmentations which can be used to inform room acoustic analysis. An example of image segmentation is shown in Figure 4.11.

A *confidence map* can be created using the results of the material segmentation process. It is a plot of the probability of the highest confidence class at each pixel. Figure 4.12 shows an example of a confidence map. Distinct drops in classification confidence can be seen at material borders, or where lighting conditions may confuse the model. In this particular example, the model is highly confident of the class 'fabric' where the bed is located. On the other hand the floor is entirely of wood, but segmentation results are mixed with *'wood', 'wallpaper' and 'plastic'*

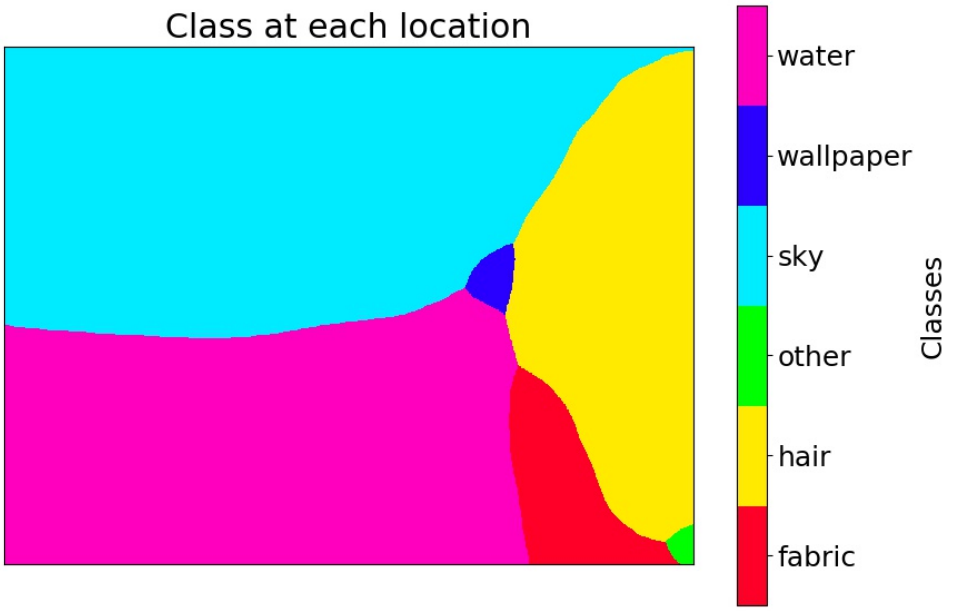


Figure 4.10: **Material segmentation with evenly spaced colours.** This plot shows the results of material segmentation. The colours chosen for materials are evenly spaced *hues* in the Hue Saturation Value (HSV) colour space.

all appearing. Looking at the confidence map, the low classification probabilities reflect the model's uncertainty about the floor's material type where lighting conditions vary.

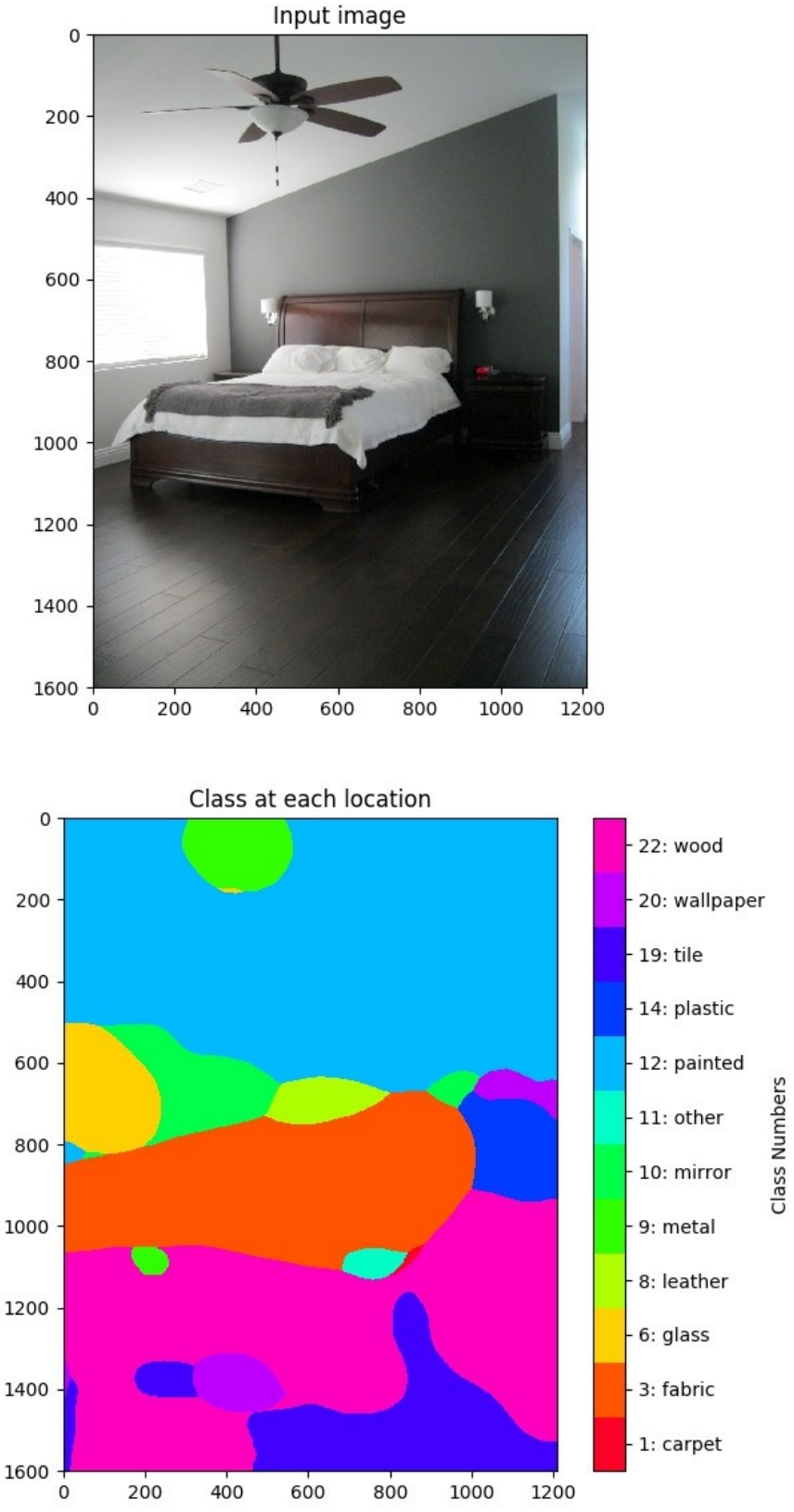


Figure 4.11: **Image segmentation plot.** Segmentation results for input image shown are plotted using technique described in Section 4.1.4. The input image dimensions are (1208, 1600).

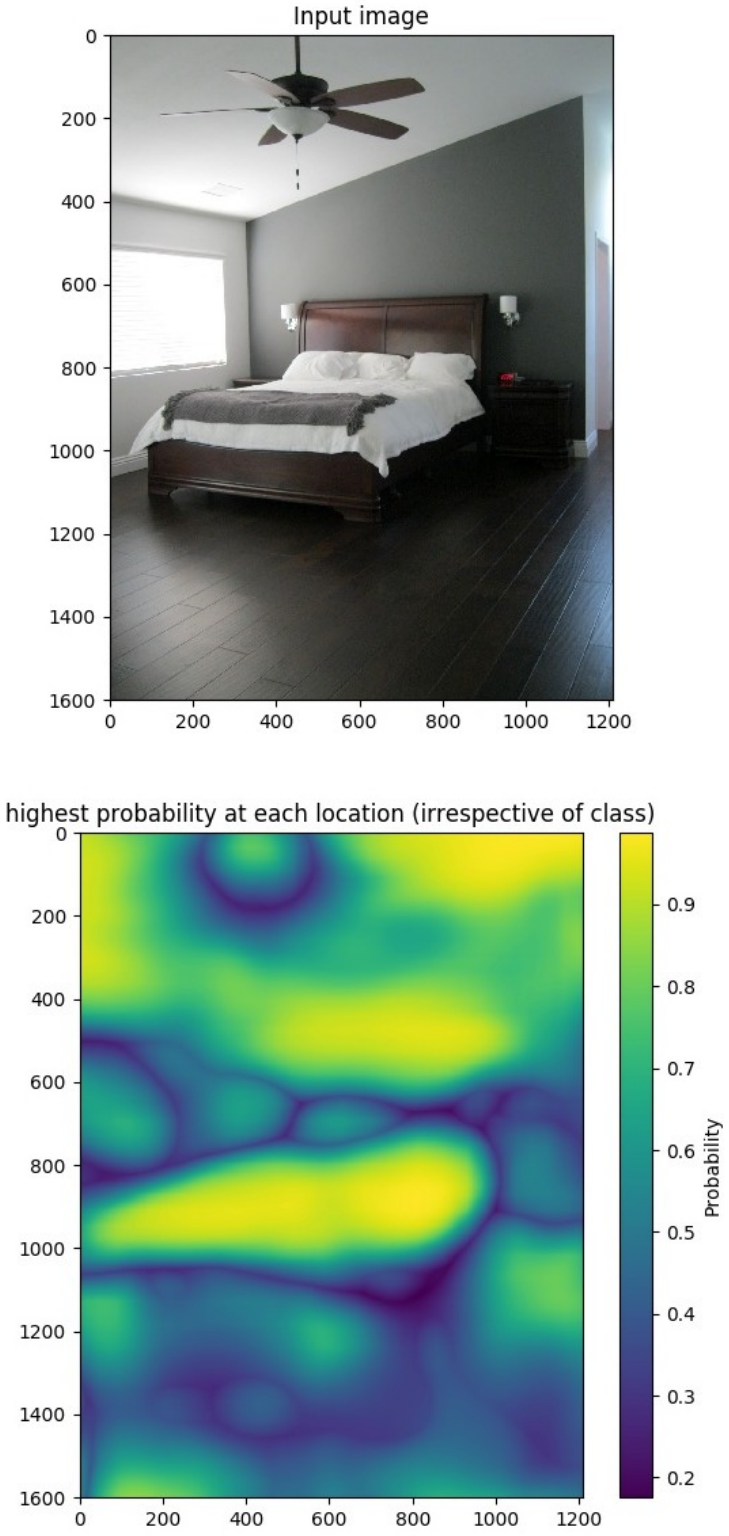


Figure 4.12: **Image segmentation confidence map.** Plot of the highest probability value at each pixel location across all 23 material classes. This *confidence map* highlights areas in the input where the classifications are highly confident. Lower confidence predictions are seen especially at material borders.

Absorption maps are created from material segmentation results. An absorption map is a material segmentation plot where the colour of the material is determined by its absorption coefficient. Highly absorptive material classes such as *carpet* and *sky* are shown in dark greyscale colours. Reflective materials which have low absorption coefficients are shown in light greyscale colours.

Segmentation results are a grid of numbers between 0 and 22 representing material classification at each pixel. Instead of converting the numbers to RGB pixel vectors to produce a material segmentation, they are converted to greyscale pixel scalars. See Section 4.1.4 for a description of how pixel values are chosen for material segmentation and absorption maps.

Appendix G contains examples of segmentation results for various images. Material segmentations, confidence maps and absorption maps are included.

4.2.1 Limitations of MINC Dataset

A fully convolutional CNN trained on the Materials in Context Database was used for the task of image segmentation based on material classification. This dataset included 23 material types, not all of which were relevant to the problem of material segmentation for estimation of acoustic properties. Classes such as 'hair' and 'food' were rarely seen in the environments we used for testing or the live demo. Other classes like 'wallpaper' tell us nothing about the surfaces underneath which actually affect the reverberation of sound. This dataset was chosen in spite of its limitations as it was the largest material dataset that could be found before starting the experiments described in this chapter. MINC was believed to contain a sufficient range of materials to serve as a proof-of-concept for the method described in this chapter.

Overall, it was difficult to quantify actual absorption information from the materials represented by this dataset. This was partly due to errors in mate-

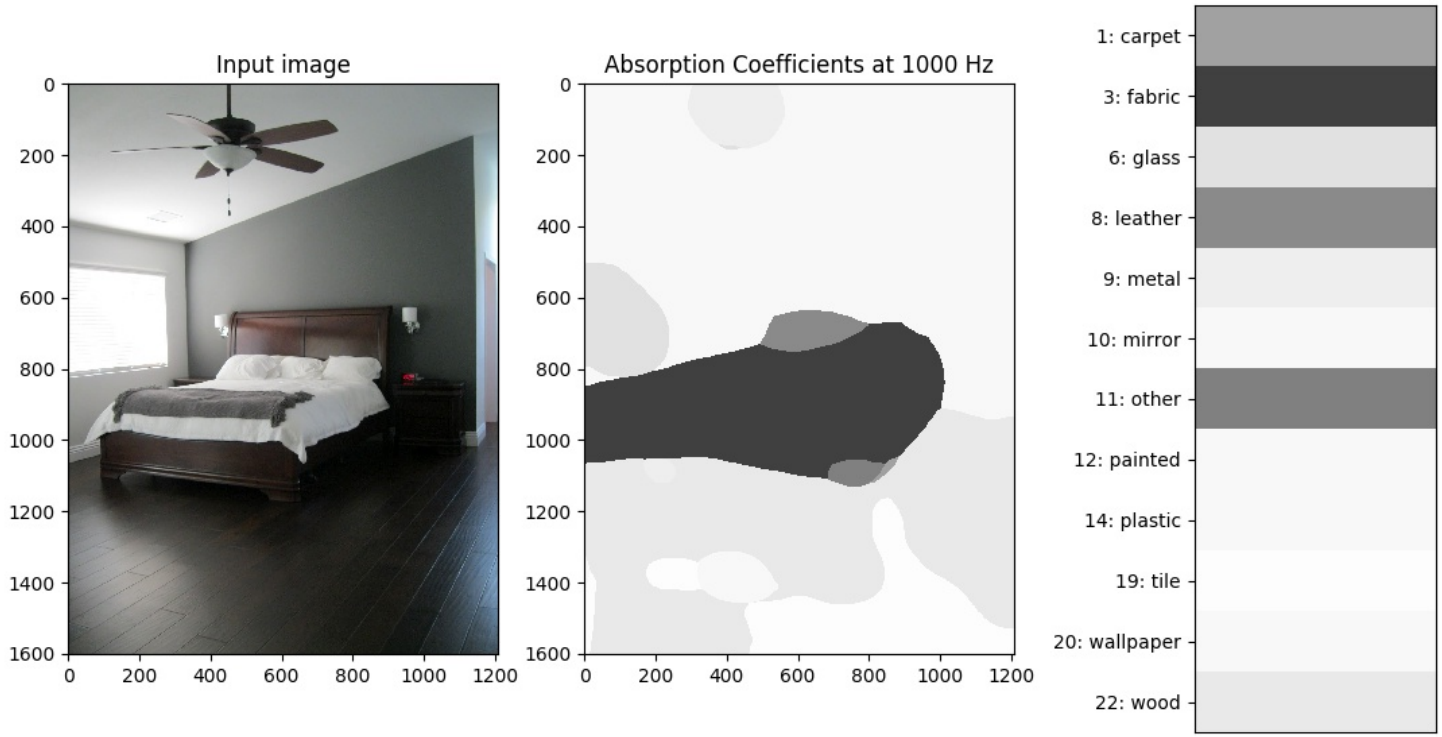


Figure 4.13: **Image segmentation absorption map.** Image segmentation based on material absorption coefficients generates an 'absorption map'. It is a visualisation of surfaces in an environment which will reflect and absorb sound waves. Dark pixels correspond to areas which will absorb more sound. Light areas represent materials which reflect more acoustic waves.

rial segmentation arising from the limited scope of material classes represented in MINC. However, the project proved that such a system could still be implemented to quickly generate material information of a scene. If a material dataset were used which contained a wider range of material information, then it is likely that the absorption maps produced by this system would be more informative.

4.3 Conclusion

This chapter presents an image-based method of analysing room acoustics. This method has the potential to be a minimally invasive technique for audio scene understanding. It is minimally invasive since no acoustic measurements are required, which can be damaging to human hearing. Material information is estimated by a Convolutional Neural Network performing image segmentation. Image segmentations are produced based on pixel-level material classification. Corresponding absorption maps are also generated. They are produced based on the absorption coefficients of materials classified at each pixel location in an image.

This technique shows promise for use in Mixed Reality audio production. Acoustic scene analysis will be an important aspect of producing audio for virtual objects in a mixed reality scenario. Audio which sounds as if it is really present in a scene will enhance user immersion, a critical aspect of multi-media experiences. This method produces information which can inform audio synthesis methods of the surrounding environment. It can also be deployed on minimal hardware as is shown in Chapter 5.

Chapter 5

Development of an Edge-AI Demo

This chapter describes the design and construction of an embedded system based demonstration kit for neural network models. It was built to demonstrate the material segmentation algorithm presented in Chapter 4 for the estimation of room acoustic properties. The work in this chapter was carried out in collaboration with employees at Xperi Ireland, the industrial partner for this project. The demo served to communicate my academic work to audio engineers and management who were interested in machine learning and image processing. This work is particularly relevant to spatial audio in mixed reality, an area which has a large scope for research and development.

5.1 Edge-AI Demonstration Setup

This section describes the setup used for the two Edge-AI demonstrations in this chapter. All hardware components used are listed and described. Software requirements and dependencies are also outlined. An overview of the toolkit used to interact with the Intel Neural Compute Stick (NCS) is given in the 'Software' section.

5.1.1 Hardware

Choosing suitable hardware components is necessary to accurately replicate Edge-AI device conditions. A primary component which influences all others is the micro-computer. This device interfaces with all peripheral components, orchestrates information processing and needs to be compatible with all software dependencies which may arise. The Raspberry Pi (Pi) micro-computer 3B+ model was chosen for this project as it fulfils all requirements of the demonstration. Its CPU is a quad-core 64-bit ARM processor running at 1.4 GHz. Combining the processor with 1GB of SDRAM, the Raspberry Pi 3B+ provides sufficient computing performance to run the demonstrations. In terms of connectivity, it has four USB ports allowing all peripheral components required by the demo to be connected. The operating system is loaded on a micro SD card for which there is a slot on the Pi. Gigabit Ethernet (over a USB 2.0 bus) and WiFi provide internet connection. While internet connectivity is not required to run the demonstrations, it's used to load source code onto the device from a remote git repository. Figure 5.1 illustrates the Pi and connections for power and data which are used in this experiment.

The Raspberry Pi is cheap and widely available. Future edge devices will most likely be produced in mass quantities and therefore will also need to be relatively low-cost. A large suite of supplemental components are available designed specifically for use with the Pi. This includes several pieces of equipment used in this project such as a camera module and a touch screen display. Power consumption by the Pi is reasonably low even when accounting for all peripheral components being in use. All peripheral components received power directly from the Pi, either via GPIO pins, USB or ribbon cable (flat flex cable connector). See Figure 5.2 for an illustration of these connections.

The two main peripheral components used for collecting sensory data are the

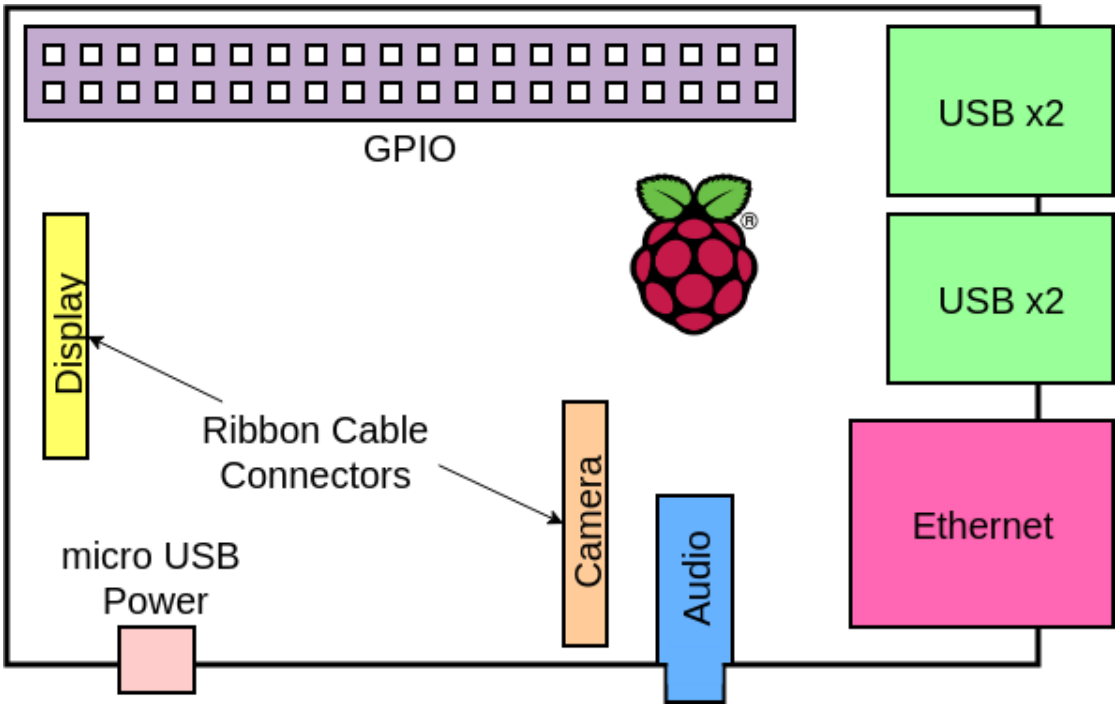


Figure 5.1: **Raspberry Pi IO.** This diagram shows the IO connections of the Raspberry Pi which are used in the demonstration described in this chapter.

Pi camera module and the touch screen display. Both are designed specifically for use with the Pi micro-computer. The camera module produced the video stream required for both demonstrations. It has a still resolution of 8 megapixels. A ribbon cable transfers data and power (250mA) between the camera and Pi.

The specifications of the camera chosen were not critical in the demo kit design. As long as the camera module delivers roughly (800x800) pixels it will satisfy the needs of the material segmentation demo. In the case of material segmentation, the input images need to be large enough so that an output map is produced rather than a single classification result. Two-dimensional output maps are used to estimate a material segmentation result which is the same dimension as the original image. Roughly, 800x800 pixel images are found to be large enough to produce good results.

The touch screen display is an important component in designing a hand-held demonstration device for Edge-AI technology. It allows a swift demo start up requiring no keyboard or mouse. Once the demo is running the results are displayed on-screen for the user to view. Visualising results of inference on images is more intuitive than console printouts. The screen quality is high enough for video stream images to be displayed with overlaying classification or segmentation results. The screen dimensions are 155mm x 86mm with a resolution of 800x400 pixels. It's connected by a ribbon cable to the serial (DSI) port on the Pi. It requires at least 500mA from the GPIO pins on the Pi. See Figure 5.2 for an illustration of how the screen is connected to the Pi.

Other components used to build the demonstration kit are a battery pack and a custom 3D-printed case. The entire system described in this section is powered by a single battery providing 2.1A at 5V. The battery capacity is 10400mAh which can support inference running continuously for over 2 hours. The custom case was designed to house all the components and be comfortable to hold while the demo runs. The Pi, camera module and battery are located directly behind the touch screen display, as shown in Figure 5.2. Handles either side of the screen allow the device to be held without obscuring any of the screen. They also hold the NCS, connected via USB cable to the Pi. See Figure 5.2 for the layout of all the components described in the demonstration device.

The setup used for this demonstration was designed specifically for this project. This included the custom 3D printed case for the hardware components. The case allowed the device to be fully portable, battery powered and hand-held. This accurately simulated requirements most Edge-AI devices would be designed to meet, i.e. low power consumption and to fit in a compact form-factor. The result of this setup was a prototype Edge-AI device capable of real time neural network inference on images from a live video stream. Figure 5.3 shows the inside of the

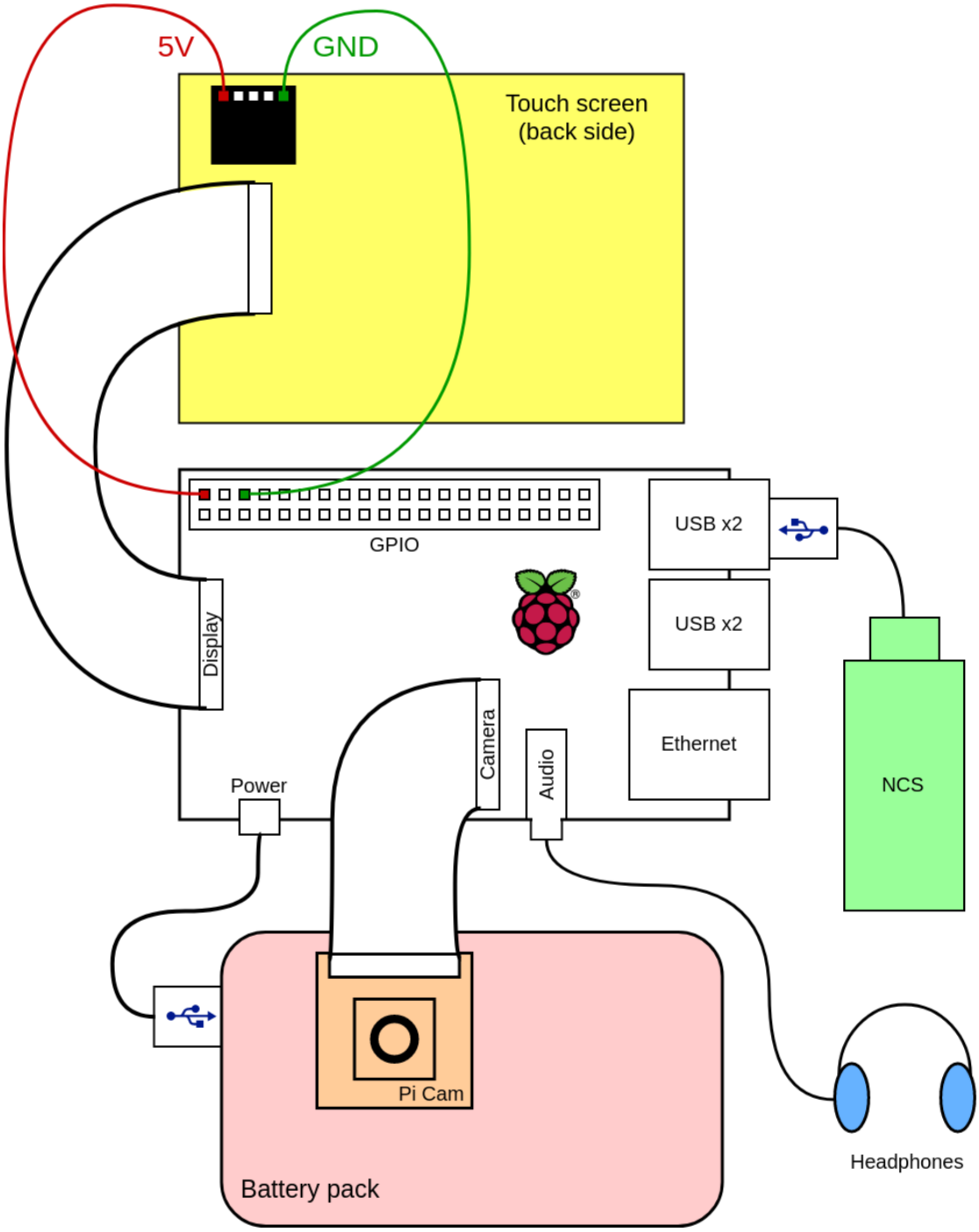


Figure 5.2: **Raspberry Pi and peripheral components.** This diagram shows the Pi and the peripheral components used in this work. Peripheral component connections to IO ports are also shown.

Figure 5.3: Hardware components inside the demonstration device

final prototype device, with all hardware components visible.

5.1.1.1 Neural Compute Stick

The Intel Neural Compute Stick (NCS) is a machine learning accelerator containing a VPU. It was designed as a deep neural network inference engine for low-power devices. The NCS is powered by Intel's 'Myriad X' VPU. While many VPUs are used to run DNN inference the 'Myriad X' is the first in production to contain a dedicated DNN hardware accelerator. This contributes to a massive increase in performance as entire networks can be loaded onto this dedicated hardware.

The Neural Compute Stick is used in the demonstration device to perform neural network inference for material segmentation. The network described in Section 4.1.2 is converted to a custom format designed for the NCS. This conversion

process is described later in Section 5.1.2 in relation to Edge-AI software. Inference is performed on images from the Pi camera module using the NCS device. The classification results are returned and processed to produce material segmentation maps.

5.1.2 Software Configuration

This section describes the software setup of the demonstration device. Software requirements arise from hardware components and the source code written to carry out the demonstration. It is important to faithfully simulate the conditions an embedded device imposes on an application. The performance limitations of embedded hardware will have strong influence on the design of commercial Edge-AI devices and what kinds of applications they can run. As such, the software configuration is designed such that it can be replicated on an embedded device.

5.1.2.1 Raspbian OS

Raspbian is the official operating system of the Raspberry Pi Foundation. It is based on Debian, an open source Unix-like operating system. Raspbian is highly optimised for low performance ARM CPUs such as those on all versions of the Pi.

5.1.2.2 OpenVINO Toolkit

OpenVINO is a software toolkit facilitating deploying of neural network models on Intel hardware platforms. It optimises models for deployment on various types of hardware, ranging from GPUs to VPUs on embedded devices. This helps to streamline the process of developing multi-platform applications which require neural network inference. The two main components of the toolkit relevant to this work are the Model Optimizer (MO) and the Inference Engine (IE). The MO performs optimisations on networks before they can be used by the IE.

The MO is a command line tool which converts models trained by supported frameworks into a common data format. This format called an Intermediate Representation (IR) is device agnostic. Two files are generated by the MO which make up the IR, one containing information on the network topology (.xml) and another containing the network's weights and biases (.bin). The conversion process can be adjusted by a set of parameters given to the MO tool. Some options are specific to the framework used to train the model and others are specific to the device the model will be deployed on.

The Inference Engine (IE) is an API included in the OpenVINO toolkit. Its function is to take an Intermediate Representation generated by the Model Optimizer and use it to infer data on a specific target device. There are classes in the Inference Engine (IE) for reading the IR, performing modifications to the network representation, and loading the network onto a target device. Modifications can be made to network topology such as layer resizing and reshaping. Other modifications are specific to the target device the model will be loaded onto.

The steps involved in preparing a neural network model for deployment on an NCS device using the OpenVINO toolkit are illustrated in Figure 5.4.

5.2 Demonstrations

This section describes the demonstration developed using the hardware setup described in this chapter. The demonstration is of a material segmentation algorithm for estimation of absorption coefficients.

5.2.1 Demonstration GUI

The demonstrations run on a Graphical User Interface (GUI) application developed in Python. The following software dependencies exist for the GUI application

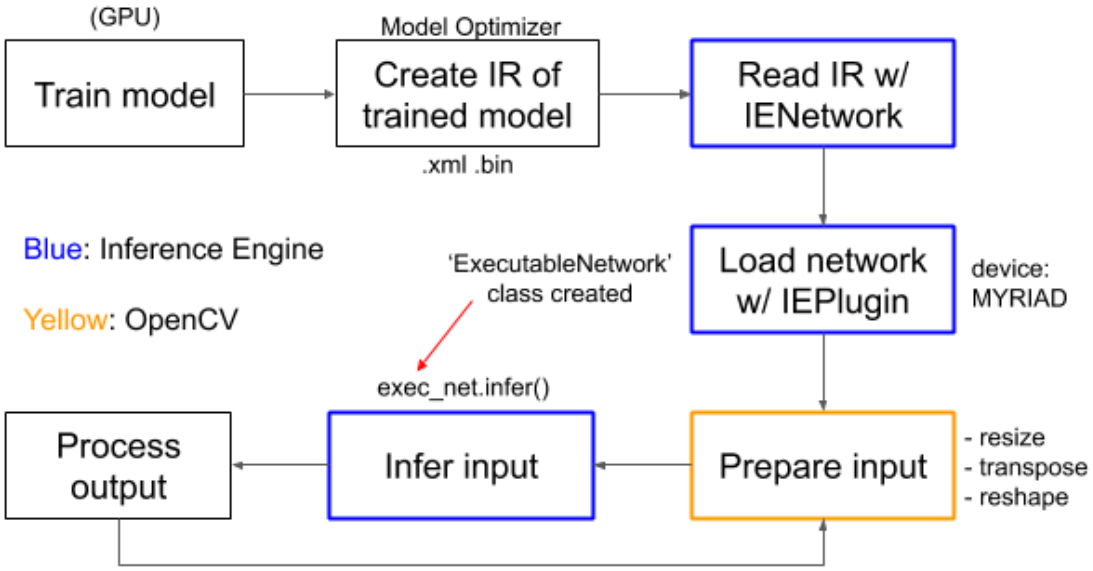


Figure 5.4: **NCS inference flowchart.** This flowchart illustrates the steps involved in deploying and using a deep learning model on the NCS. Firstly, a model is required which has been trained on a GPU. The *Model Optimizer (MO)* creates an *Intermediate Representation (IR)* of the model which can then be deployed for testing using the *OpenVINO* API. The IR is loaded for the target device, in this case the NCS. The modified input frame is then fed forward through the model to generate an output. The output is processed and displayed.

- *TkInter*: the default GUI package for Python
- *OpenCV* [91]: computer vision library
- *PIL*: plotting library
- *matplotlib*: plotting library
- *OpenVINO*: Neural Compute Stick (NCS) API
- *numpy*: Python scientific computing library
- *plotting_utils.py*: script containing plotting functions specific to this demonstration

A custom class was created which sets up and controls the GUI. The class is named *SegmentationApp*. Objects of this class must be given three inputs when instantiated;

1. A video stream (OpenCV 'VideoCapture' object)
2. A path to an NCS-compatible CNN model
3. An integer number of padding pixels

When an object of the *SegmentationApp* class is created a set-up process occurs. These steps must be carried out before any GUI operations or threads start.

1. Initialisation of GUI variables
2. GUI elements (buttons and windows) are created and arranged.
3. Set up NCS by loading the appropriate plugin, changing the input shape of the model, and loading the model onto the device.

Figure 5.5 shows a screenshot of the GUI immediately after being initialised.

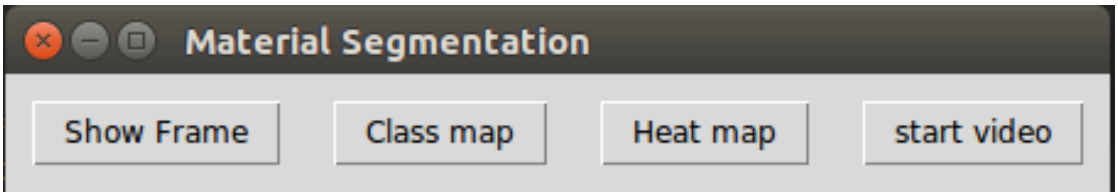


Figure 5.5: **Initialised GUI.** This screenshot shows the GUI immediately after being initialised. The user begins the video stream by pressing the *start video* button.

Once the GUI is set up, the user can press a button labelled *start video* to begin the video stream. This action starts a thread handling the reading of frames from an OpenCV *VideoCapture* object. Figure 5.6 shows the GUI with a video

stream frame being displayed. The code snippet below shows the setting up of a thread.

```
# Flag which when set will stop thread
self.stopVideo = threading.Event()
# Target of thread is function for plotting frames to GUI
self.video_thread = threading.Thread(target=self.video_loop, args=())
self.video_thread.start() # Start thread
```

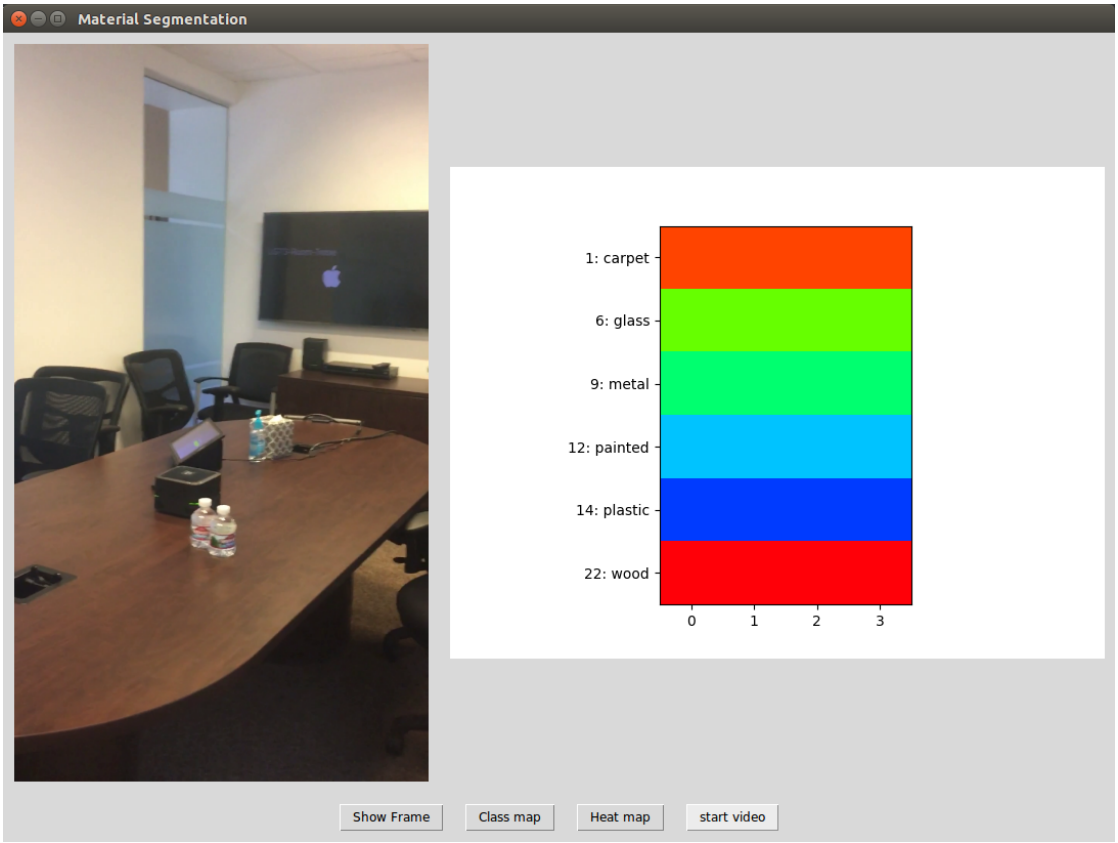


Figure 5.6: **GUI with video frame.** This screenshot shows the GUI with a frame from the video stream being displayed.

Once the video stream frames are being displayed, the user can segment a

frame by pressing buttons *Class map* or *Heat map*. Here, *Heat map* refers to a segmentation based on absorption coefficients. It was thought that this term might be more intuitive to some users. Figures 5.7 and 5.8 show examples of the GUI after both segmentations types have been performed.

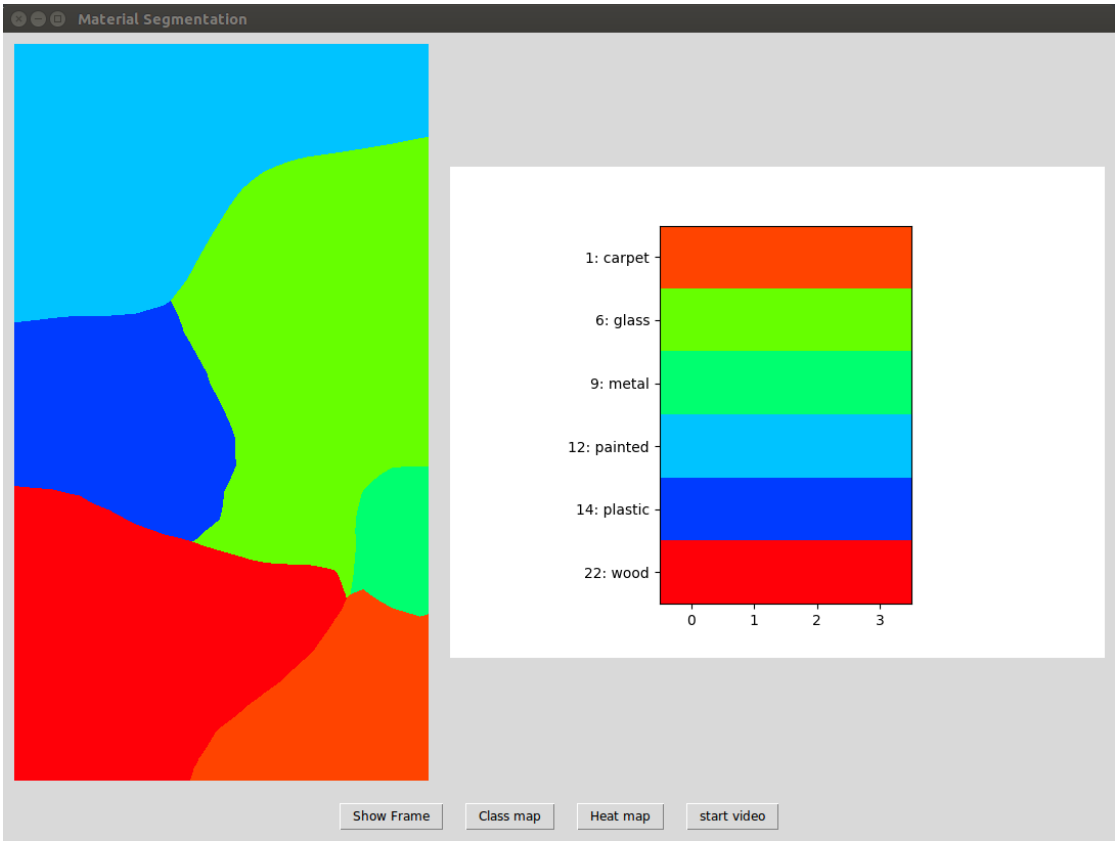


Figure 5.7: **GUI with material segmentation.**

Methods within the *SegmentationApp* class are responsible for handling tasks including reading video frames, segmenting frames, displaying frames in the GUI and handling button presses. The full Python code for the *SegmentationApp* class is included in Appendix H. Also in Appendix H is code for starting a GUI using this class.

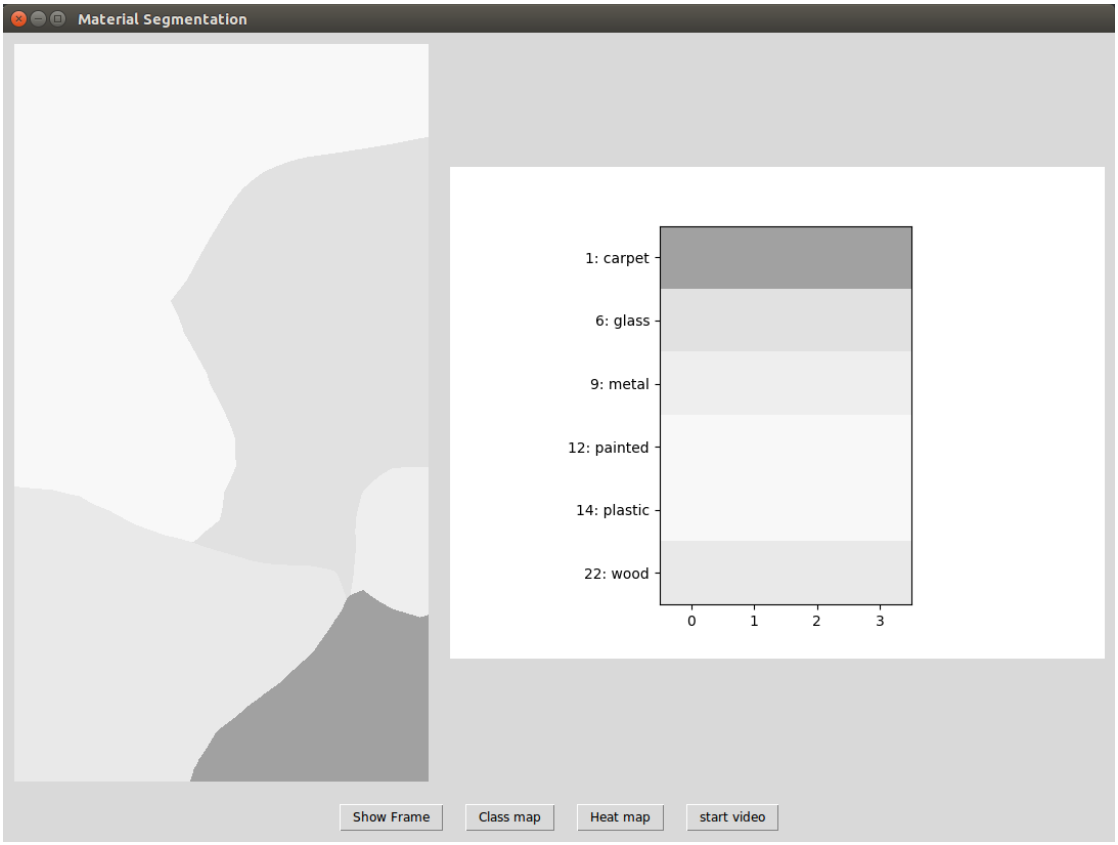


Figure 5.8: **GUI with absorption coefficient segmentation.**

5.2.2 Material Segmentation and Absorption Coefficients

This section describes an image segmentation pipeline customised for deployment on embedded hardware. It forms part of a demonstration of Edge-AI technology, with particular focus on using the Neural Compute Stick for performing neural network inference locally. The image segmentation process is based on material classification. A detailed technical background of the material segmentation process is given in Chapter 4.

Before executing material segmentation, a number of steps must be carried out to prepare a neural network model for deployment on the NCS device. These steps are illustrated in Figure 5.4. The output of the material segmentation process is

a two-dimensional vector. The workflow of image segmentation (once a model has been configured and loaded onto the NCS) is illustrated in Figure 5.9.

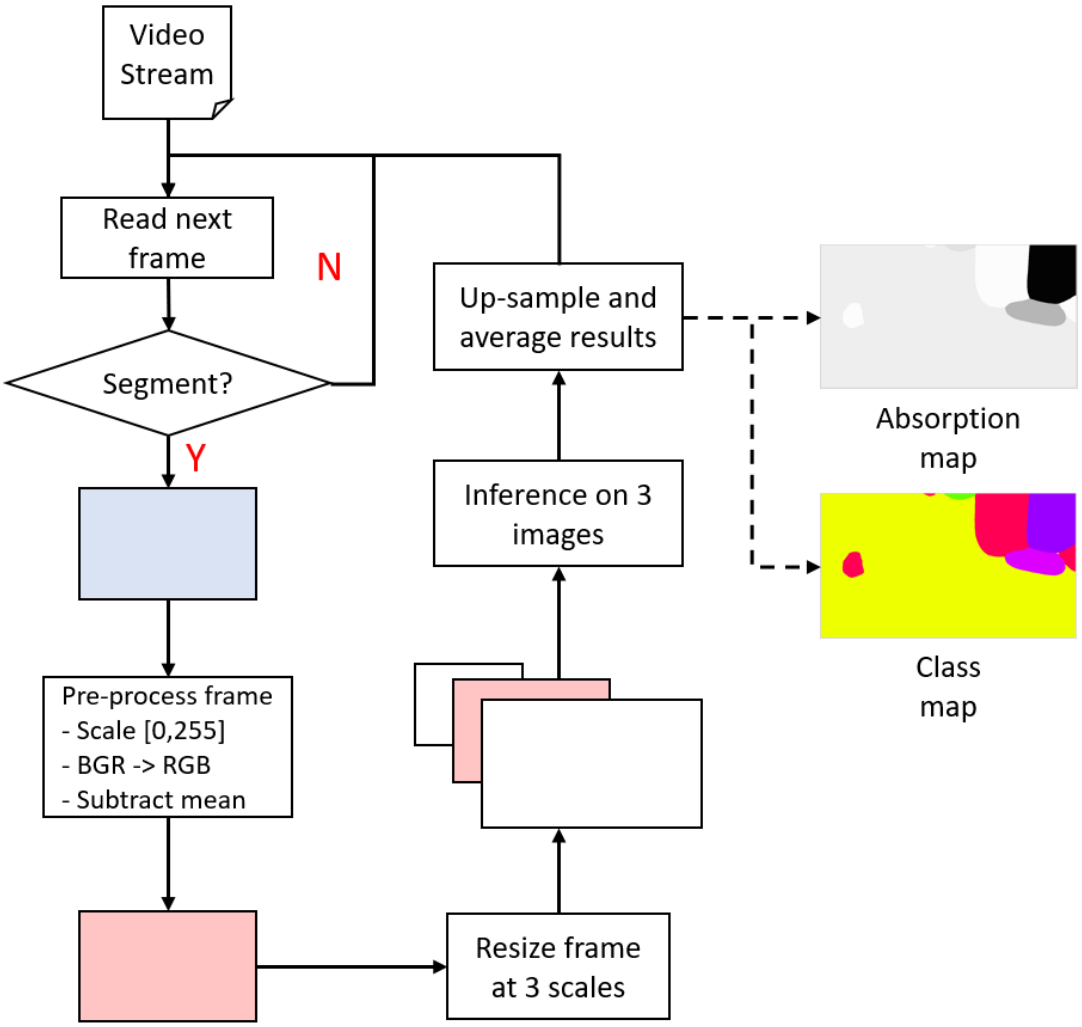


Figure 5.9: **Image segmentation on NCS flowchart.** This flowchart illustrates the image segmentation process using the Neural Compute Stick. Frames are read from a video stream, processed and resized at three scales. Inference is performed one each image. Results are upsampled to the original image size and then averaged. Material class and absorption coefficient maps are generated from the results.

A custom Graphical User Interface was designed to run the material segmenta-

tion demonstration. The GUI is controlled by a set of Python scripts. All source code associated with this demonstration can be found on my GitHub page [1].

5.2.3 Modified Plotting Method

The process of plotting material segmentation results in the Edge-AI demonstration differs from that described in Section 4.1.4 for still image segmentations. The same need for evenly spaced material label colours applies to this work.

The principle difference is that for a video stream (demonstration), it would be undesirable if the colour used to label a class changed between frames based on the total number of materials present. This is the result when the method in Section 4.1.4 is used. Therefore, the colormap stays constant for all frames, rather than changing every frame based on the number of material classes present.

The Hue Saturation Value (HSV) colour spectrum is used to create a colormap of 23 evenly spaced colours for plotting material segmentations. It is an alternative to the RGB colour model. The colour space is arranged in a cylindrical geometry as shown in Figure 5.10. Hues are radial slices ranging from 0°to 360°. The *saturation* dimension describes the intensity of a colour. The *value* dimension describes how much black is mixed with each colour (producing *shades* on the vertical dimension).

The OpenCV library [91] is used to convert HSV tensors to RGB tensors. In OpenCV hue values are in the range [0, 179], saturation and value dimensions have ranges [0, 255].

The first step in creating a colormap for material segmentation is to generate 23 evenly spaced hue values. Value and saturation are both set to 255 to represent *pure colours*. Tuples representing HSV colours are then converted to the RGB colour space using OpenCV's *cvtColor* method. The following code excerpt is the

[1] `https://github.com/aoifemcdonagh/material-segmentation`

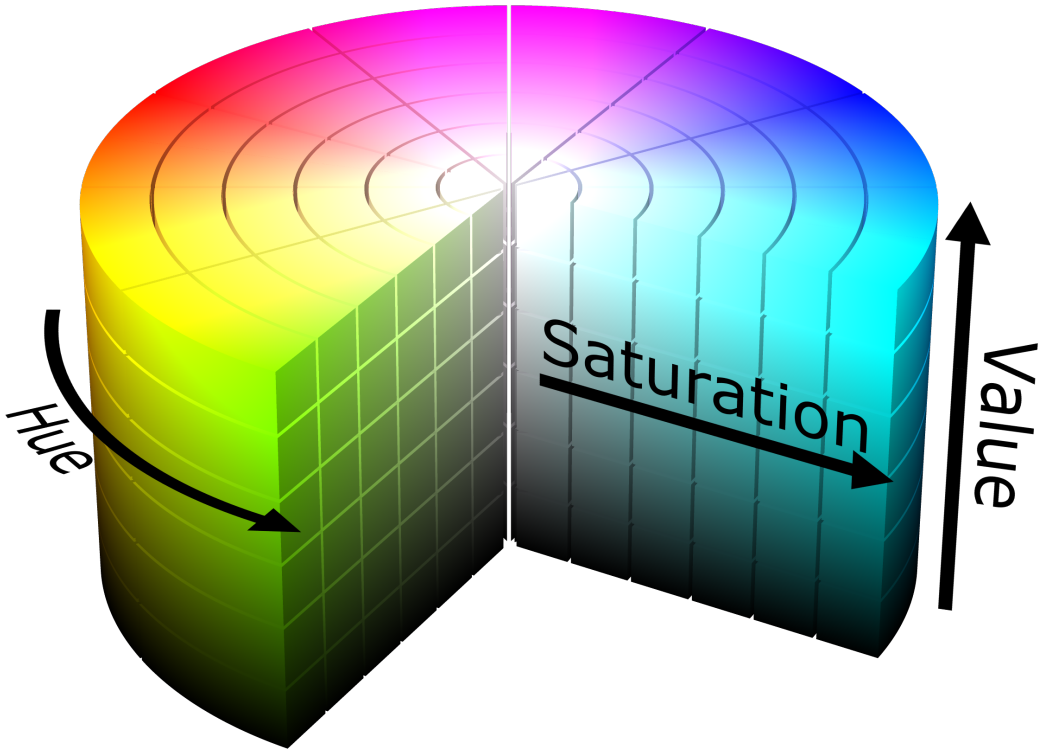


Figure 5.10: **HSV colour space.** Hues are arranged in a radial pattern around a vertical axis. Saturation defines the colour intensity. Value defines the amount of black (or shade) in a colour.

function used to generate a colormap as a list of RGB tuples.

```
def generate_color_map():
    # 23 evenly spaced Hue values
    hues = np.linspace(0, 179, num=len(CLASS_LIST), dtype=int)
    hsv_colors = []
    for hue in hues:
        hsv_colors.append((hue, 255, 255))
```

```
    rgb_colors = []
    for hsv_color in hsv_colors:
        rgb_colors.append(cv2.cvtColor(np.uint8([[hsv_color]]),
            cv2.COLOR_HSV2RGB)[0][0])
    return rgb_colors
```

The output of material segmentation is a two-dimensional array of integer values in the range [0, 22]. The value at each pixel represents the material classification at the equivalent location in the input image. The colormap is used to convert this two-dimensional array to an array of three-dimensional RGB tuples. The following function *get_pixel_map()* shows the process of generating a map of pixels for plotting.

```
def get_pixel_map(class_map):
    # Convert 2D array to array of RGB pixels
    pixel_map = np.array([[self.colormap[class_num]
            for class_num in row]
            for row in class_map], dtype=np.uint8)
    return pixel_map
```

The pixel map is plotted to a GUI using the following Python packages

- PIL
- TkInter

```
# PIL converts pixel_map to object plottable by TkInter
image = PIL.Image.fromarray(pixel_map)
image = PIL.ImageTk.PhotoImage(image)
```

```
# Update TkInter GUI panel with segmented image.
self.panel.configure(image=image)
self.panel.image = image
```

The full code for segmentation, generating pixel maps, and the GUI interface is available on my GitHub page[2].

5.3 Conclusion

This chapter describes a demonstration of room acoustic property estimation. It makes use of a new computing paradigm called *Edge-AI*. All neural network inference is carried out locally on the device instead of outsourced to a remote "cloud" service. The demonstration is based on embedded systems hardware. This is made possible by the neural network's relatively small size and low power consumption when performing inference on the Neural Compute Stick Edge-AI device.

This chapter serves as a proof-of-concept for the potential of the method in Chapter 4 in Mixed Reality device design. Audio which matches the acoustic properties of an environment can enhance user immersion. In this vein, the production of sound for virtual objects in a Mixed Reality environment could benefit from acoustic scene analysis. This demonstration shows how room acoustic properties can be estimated quickly and without exposing the user to loud test sounds.

[2] https://github.com/aoifemcdonagh/material-segmentation

Chapter 6

Conclusions and Future Work

6.1 Summary of Thesis

This thesis outlines examples of Deep Learning in audio engineering applications. Deep Learning can be utilised in many ways, including in both the analysis and synthesis of audio. While implementations of Deep Learning for audio applications tend to lag behind those in imaging, the field is making exciting gains over traditional methods. This thesis summarises a small portion of these advanced methods.

The first topic introduced in this thesis is that of audio synthesis using Generative Adversarial Networks. A WaveGAN [32] model was chosen and trained on sub-classes of the AudioSet dataset [12]. The result of this method was the blending of synthesised audio classes.

The second topic in this thesis is an investigation into the use of computer vision techniques for audio-scene understanding. A Convolutional Neural Network model (GoogLeNet [18]) trained on the Materials in Context Database [53] was used to perform full scene segmentation. Segmentation results were based on performing material classification densely across input images. The resulting material class

segmentations were used to generate absorption coefficient maps for input images. Absorption coefficient information is crucial for inferring acoustic characteristics of environments, such as reverberation.

The final topic in this thesis was the development of a demonstration device for real time inference on embedded systems hardware. Parts of the audio-scene understanding work in Chapter 4 were incorporated into this demonstration to deliver a solid proof-of-concept device. The concepts surrounding *Edge AI* were utilised in this work. Inference was performed on an embedded systems device without the need for connections to the "cloud" for data processing. The device was light and hand-held which allowed demonstrations to be conducted in any desired location.

The works presented in this thesis have the potential to be used in audio production environments. Their primary commercial applications include audio design for video games and films, and dynamic Mixed Reality audio scene understanding. Overall, they achieved the shared goal of improving audio production for multi-media experiences, utilising Machine Learning based methods.

6.2 Future Work

Future work in this field is poised to bring about many changes which will likely be beneficial in improving human-computer interactions. Other areas such as computer-environment understanding are effected by topics in this thesis with respect to audio-scene understanding.

6.2.1 Improved Audio Synthesis approaches

AudioSet [12] was created for the purpose of furthering the state-of-the-art in audio event recognition.

> "We would like to produce an audio event recognizer that can label hundreds or thousands of different sound events in real-world recordings with a time resolution better than one second" [12]

The samples in AudioSet were therefore collected with the goal of audio event recognition in mind. Samples are ten seconds long, with many samples containing labels for audio events less than a second in duration. This kind of data may not be suitable for training models to synthesise audio. For example, a ten second long sample may only contain two seconds of audio for class "violin". If such a sample is used to train a model on audio synthesis, the other eight seconds of audio will have a negative effect on the model's performance.

Future work on audio synthesis may require an application-specific version of this dataset. This would involve verifying that samples contain enough of each target class to be suitable for training a generative model.

Future work on audio synthesis using GANs may incorporate principles from time-synthesis models. One such idea would be to use a Long Short-Term Memory (LSTM) based model to generate the input vectors required for GAN synthesis. The LSTM would perform the task of 'remembering' previous output states. Future work on this problem may improve the quality of audio synthesis of GANs on output sequences longer than their fixed output size.

Since completing the work described in Chapter 3 many improved synthesis methods have been proposed in deep learning literature. Notable examples include auto-regressive models with significantly improved synthesis performance [92]. Auto-regressive models are particularly suitable for audio synthesis since their outputs can run continuously, with the result at each time-step dependent on outputs preceding it. This is advantageous compared to GANs whose outputs are of a fixed size, and have no dependence on previous output states.

All acoustic signals are periodic in nature, however some at different frequencies

than others. For example, speech signals exhibit periodicity at shorter timescales than music which can have temporal patterns at multiple timescales. In the context of audio synthesis, this means that models need to have varying receptive fields in order to synthesis audio signals exhibiting periodicity at varying time-scales. This is easier to achieve using auto-regressive models than GANs since the receptive field of a GAN is fixed, while auto-regressive models can increase their receptive field with model depth [34].

6.2.2 Analysis of Room Acoustics

The development of an application-specific dataset would benefit the work carried out in Chapter 4. Section 4.2.1 describes the limitations of the Materials in Context Database. Many of the material classes seen in the Materials in Context Database [53] are not relevant to the task of room acoustic estimation which confounded results. Ideally, a material dataset specific to the problem of room acoustics would contain more detail on materials frequently found indoors including carpets, fabrics, plaster, and brick.

The work described in Chapter 4 could also be extended by the addition of a depth estimation method. Room geometry is critically important to the measurement of room acoustics. Room geometry has a large effect on early sound reflections as described in Section 2.7. Early reflections give listeners a sense of room's size without the need to see it.

Techniques for depth estimation from monocular images have been developed in recent years [93, 94, 95]. Implementation of such a method would allow material segmentation and depth estimation systems to operate on the same hardware. The combination of material and depth measurements could provide a more robust reverberation estimation from still images.

Bibliography

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[3] A. v. d. Oord, Y. Li, I. Babuschkin, K. Simonyan, O. Vinyals, K. Kavukcuoglu, G. v. d. Driessche, E. Lockhart, L. C. Cobo, F. Stimberg, *et al.*, "Parallel wavenet: Fast high-fidelity speech synthesis," *arXiv preprint arXiv:1711.10433*, 2017.

[4] A. McDonagh, J. Lemley, R. Cassidy, and P. Corcoran, "Synthesizing game audio using deep neural networks," in *2018 IEEE Games, Entertainment, Media Conference (GEM)*, pp. 1–9, IEEE, 2018.

[5] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, pp. 2672–2680, 2014.

[6] R. Dechter, "Learning while searching in constraint-satisfaction-problems," in *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*, pp. 178–183, AAAI Press, 1986.

[7] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[8] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[9] D. E. Rumelhart, G. E. Hinton, R. J. Williams, *et al.*, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[10] X. Xu, Y. Ding, S. X. Hu, M. Niemier, J. Cong, Y. Hu, and Y. Shi, "Scaling for edge inference of deep neural networks," *Nature Electronics*, vol. 1, no. 4, p. 216, 2018.

[11] "Reprinted by permission from Springer Nature: Nature, Nature Electronics, Xu, X., Ding, Y., Hu, S.X. et al. Scaling for edge inference of deep neural networks., [COPYRIGHT] 2018, advance online publication, (doi: 10.1038/s41928-018-0059-3.),"

[12] J. F. Gemmeke, D. P. Ellis, D. Freedman, A. Jansen, W. Lawrence, R. C. Moore, M. Plakal, and M. Ritter, "Audio set: An ontology and human-labeled dataset for audio events," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 776–780, IEEE, 2017.

[13] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, "Revisiting unreasonable effectiveness of data in deep learning era," in *Proceedings of the IEEE international conference on computer vision*, pp. 843–852, 2017.

[14] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and scalability of gpu-based convolutional neural networks," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pp. 317–324, IEEE, 2010.

[15] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, IEEE, 2017.

[16] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[19] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[20] L. Perez and J. Wang, "The effectiveness of data augmentation in image classification using deep learning," *arXiv preprint arXiv:1712.04621*, 2017.

[21] J. Salamon and J. P. Bello, "Deep convolutional neural networks and data augmentation for environmental sound classification," *IEEE Signal Processing Letters*, vol. 24, no. 3, pp. 279–283, 2017.

[22] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

[24] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*, p. 333. MIT press, 2016.

[25] I. Sobel and G. Feldman, "A 3x3 isotropic gradient operator for image processing," *a talk at the Stanford Artificial Project in*, pp. 271–272, 1968.

[26] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*, pp. 818–833, Springer, 2014.

[27] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *arXiv preprint arXiv:1312.6034*, 2013.

[28] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," in *Proceedings of the 26th annual international conference on machine learning*, pp. 609–616, ACM, 2009.

[29] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, p. 3, 2013.

[30] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4401–4410, 2019.

[31] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive growing of gans for improved quality, stability, and variation," *arXiv preprint arXiv:1710.10196*, 2017.

[32] C. Donahue, J. McAuley, and M. Puckette, "Synthesizing audio with generative adversarial networks," *arXiv preprint arXiv:1802.04208*, vol. 1, 2018.

[33] J. Engel, K. K. Agrawal, S. Chen, I. Gulrajani, C. Donahue, and A. Roberts, "Gansynth: Adversarial neural audio synthesis," *arXiv preprint arXiv:1902.08710*, 2019.

[34] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[35] S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. Courville, and Y. Bengio, "Samplernn: An unconditional end-to-end neural audio generation model," *arXiv preprint arXiv:1612.07837*, 2016.

[36] S. Pascual, A. Bonafonte, and J. Serra, "Segan: Speech enhancement generative adversarial network," *arXiv preprint arXiv:1703.09452*, 2017.

[37] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[38] A. Van den Oord, N. Kalchbrenner, L. Espeholt, O. Vinyals, A. Graves, *et al.*, "Conditional image generation with pixelcnn decoders," in *Advances in neural information processing systems*, pp. 4790–4798, 2016.

[39] A. v. d. Oord, N. Kalchbrenner, and K. Kavukcuoglu, "Pixel recurrent neural networks," *arXiv preprint arXiv:1601.06759*, 2016.

[40] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra, "Draw: a recurrent neural network for image generation," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37*, pp. 1462–1471, JMLR. org, 2015.

[41] K. Gregor, I. Danihelka, A. Mnih, C. Blundell, and D. Wierstra, "Deep autoregressive networks," in *Proceedings of the 31st International Conference on International Conference on Machine Learning-Volume 32*, pp. II–1242, JMLR. org, 2014.

[42] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, "Exploring the limits of language modeling," *arXiv preprint arXiv:1602.02410*, 2016.

[43] I. Sutskever, J. Martens, and G. E. Hinton, "Generating text with recurrent neural networks," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 1017–1024, 2011.

[44] M. Shannon, H. Zen, and W. Byrne, "Autoregressive models for statistical parametric speech synthesis," *IEEE transactions on audio, speech, and language processing*, vol. 21, no. 3, pp. 587–597, 2012.

[45] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[46] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, pp. 3104–3112, 2014.

[47] D. Griffin and J. Lim, "Signal estimation from modified short-time fourier transform," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 32, no. 2, pp. 236–243, 1984.

[48] T. N. Sainath, R. J. Weiss, A. Senior, K. W. Wilson, and O. Vinyals, "Learning the speech front-end with raw waveform cldnns," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[49] A. McDonagh, "Dialect Detect Classification of Regional Accents in the Irish Language using Artificial Neural Networks," *BEng Thesis*, 2017.

[50] C. Jennett, A. L. Cox, P. Cairns, S. Dhoparee, A. Epps, T. Tijs, and A. Walton, "Measuring and defining the experience of immersion in games," *International journal of human-computer studies*, vol. 66, no. 9, pp. 641–661, 2008.

[51] E. Brown and P. Cairns, "A grounded investigation of game immersion," in *CHI'04 extended abstracts on Human factors in computing systems*, pp. 1297–1300, ACM, 2004.

[52] B. Hariharan, P. Arbeláez, R. Girshick, and J. Malik, "Hypercolumns for object segmentation and fine-grained localization," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 447–456, 2015.

[53] S. Bell, P. Upchurch, N. Snavely, and K. Bala, "Material recognition in the wild with the materials in context database," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3479–3487, 2015.

[54] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[55] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia, "Multi-view 3d object detection network for autonomous driving," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[56] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer, "Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 129–137, 2017.

[57] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 243–254, IEEE, 2016.

[58] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, p. 115, 2017.

[59] K. Richards, F. van den Dool, and K.-W. Joshua, "2017 Cost of Cyber Crime Study," 2017.

[60] Symantec, "Internet Security Threat Report," 2018.

[61] Y. Liang, Z. Cai, J. Yu, Q. Han, and Y. Li, "Deep learning based inference of private information using embedded sensors in smart devices," *IEEE Network*, vol. 32, no. 4, pp. 8–14, 2018.

[62] D. Poddebniak, C. Dresen, J. Müller, F. Ising, S. Schinzel, S. Friedberger, J. Somorovsky, and J. Schwenk, "Efail: Breaking s/mime and openpgp email encryption using exfiltration channels," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 549–566, USENIX Association, Aug. 2018.

[63] K. Hashizume, D. G. Rosado, E. Fernández-Medina, and E. B. Fernandez, "An analysis of security issues for cloud computing," *Journal of internet services and applications*, vol. 4, no. 1, p. 5, 2013.

[64] Z. Zou, Y. Jin, P. Nevalainen, Y. Huan, J. Heikkonen, and T. Westerlund, "Edge and fog computing enabled ai for iot-an overview," in *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 51–56, IEEE, 2019.

[65] Y. Ai, M. Peng, and K. Zhang, "Edge computing technologies for internet of things: a primer," *Digital Communications and Networks*, vol. 4, no. 2, pp. 77–86, 2018.

[66] J. Liu and Q. Zhang, "Offloading schemes in mobile edge computing for ultra-reliable low latency communications," *Ieee Access*, vol. 6, pp. 12825–12837, 2018.

[67] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 328–339, IEEE, 2017.

[68] P.-K. Tsung, T.-C. Chen, C.-H. Lin, C.-Y. Chang, and J.-M. Hsu, "Heterogeneous computing for edge ai," in *2019 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pp. 1–2, IEEE, 2019.

[69] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, ACM, 2015.

[70] G. Lacey, G. W. Taylor, and S. Areibi, "Deep learning on fpgas: Past, present, and future," *CoRR*, vol. abs/1602.04283, 2016.

[71] K. Guo, S. Han, S. Yao, Y. Wang, Y. Xie, and H. Yang, "Software-hardware codesign for efficient neural network acceleration," *IEEE Micro*, vol. 37, no. 2, pp. 18–25, 2017.

[72] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 449–460, IEEE Computer Society, 2012.

[73] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 134–144, IEEE, 2011.

[74] D. Moloney, "1TOPS/W software programmable media processor," in *2011 IEEE Hot Chips 23 Symposium (HCS)*, pp. 1–24, IEEE, 2011.

[75] S. Rivas-Gomez, A. J. Pena, D. Moloney, E. Laure, and S. Markidis, "Exploring the vision processing unit as co-processor for inference," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 589–598, IEEE, 2018.

[76] D. Moloney, B. Barry, R. Richmond, F. Connor, C. Brick, and D. Donohoe, "Myriad 2: Eye of the computational vision storm," in *2014 IEEE Hot Chips 26 Symposium (HCS)*, pp. 1–18, IEEE, 2014.

[77] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'Riordan, and V. Toma, "Always-on vision processing unit for mobile applications," *IEEE Micro*, vol. 35, no. 2, pp. 56–66, 2015.

[78] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

[79] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[80] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

[81] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved training of wasserstein gans," in *Advances in neural information processing systems*, pp. 5767–5777, 2017.

[82] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," in *Advances in neural information processing systems*, pp. 2234–2242, 2016.

[83] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[84] S. Barratt and R. Sharma, "A note on the inception score," *arXiv preprint arXiv:1801.01973*, 2018.

[85] P. Milgram and F. Kishino, "A taxonomy of mixed reality visual displays," *IEICE TRANSACTIONS on Information and Systems*, vol. 77, no. 12, pp. 1321–1329, 1994.

[86] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[87] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *arXiv preprint arXiv:1312.6229*, 2013.

[88] G. Lin, C. Shen, A. Van Den Hengel, and I. Reid, "Efficient piecewise training of deep structured models for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3194–3203, 2016.

[89] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1915–1929, 2012.

[90] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[91] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[92] N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimberg, A. v. d. Oord, S. Dieleman, and K. Kavukcuoglu, "Efficient neural audio synthesis," *arXiv preprint arXiv:1802.08435*, 2018.

[93] A. Saxena, S. H. Chung, and A. Y. Ng, "Learning depth from single monocular images," in *Advances in neural information processing systems*, pp. 1161–1168, 2006.

[94] D. Eigen, C. Puhrsch, and R. Fergus, "Depth map prediction from a single image using a multi-scale deep network," in *Advances in neural information processing systems*, pp. 2366–2374, 2014.

[95] E. Ricci, W. Ouyang, X. Wang, N. Sebe, *et al.*, "Monocular depth estimation using multi-scale continuous crfs as sequential deep networks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, no. 6, pp. 1426–1440, 2018.

Appendices

Appendix A

Synthesizing Game Audio Using Deep Neural Networks

Synthesizing Game Audio Using Deep Neural Networks

Aoife McDonagh * †, Joseph Lemley * †, Ryan Cassidy†, and Peter Corcoran*
* College of Engineering and Informatics, NUI Galway, Galway, Ireland
† Xperi Corporation, Poway, CA, United States

***Abstract*—High quality audio plays an important role in gaming, contributing to player immersion during gameplay. Creating audio content which matches overall theme and aesthetic is essential, such that players can become fully engrossed in a game environment. Sound effects must also fit well with visual elements of a game so as not to break player immersion. Producing suitable, unique sound effects requires the use of a wide range of audio processing techniques. In this paper, we examine a method of generating in-game audio using Generative Adversarial Networks, and compare this to traditional methods of synthesizing and augmenting audio.**

I. Introduction

Immersion is an important aspect of gaming for creating a realistic and satisfying player experience. In the gaming community, immersion refers to the feeling of being cut off from reality, as if spatially located in the game environment with which one is interacting [1]. To provide an immersive experience, a game must capture the full attention of the player, allowing him/her to lose sense of time and self. The extent to which a player feels immersed appears correlated with the degree to which their visual, auditory and mental facilities are drawn into the game environment [2]. Auditory techniques for creating immersion have largely been underdeveloped compared to the level of investment in visual aspects of gaming. The quality of computer graphics is the largest factor influencing consumers when buying video games, according to a report by the Entertainment Software Association [3]. However, negative effects on player experience have been demonstrated when audio is removed entirely from a game [4]. System usability decreases because players receive fewer or no responses from the system in response to their actions and commands. Sense of presence is also degraded when a game is reduced to "nothing but animated graphics on a screen" [4]. While the importance of having audio in computer games is widely accepted, it is perhaps the effect of audio quality and depth that is less widely known and discussed. In this paper, we examine a method for the creation of unique sounds based on deep learning methods. We employ this technique to generate samples of in-game audio which could be used to improve in-game immersive experiences.

This research is funded under the SFI Strategic Partnership Program by Science Foundation Ireland (SFI) and FotoNation Ltd. Project ID: 13/SPP/I2868 on Next Generation Imaging for Smartphone and Embedded Platforms. This work is also supported Irish Research Council Employment Based Programme Awards EBP/2017/476 and EBPPG/2016/280

A. Deep Learning

Deep Learning has experienced significant interest in recent years, resulting in top performance on a number of important tasks. The concept of deep learning refers to a loose assortment of techniques, utilizing artificial neural networks (ANNs), that make use of two or more layers. The most popular of these methods include Convolutional Neural networks (CNN) [5] and Recurrent Neural Networks (RNNs) such as Long Short Term Memory (LSTM) [6] models.

The strength of ANNS, like other machine learning techniques, is that, unlike conventional computer programs, they can learn through example rather than by direct instruction. This allows them to learn things that humans understand intuitively but have a difficult time putting into concrete sets of procedures and rules [7]. Convolutional Neural Networks are heavily used in computer vision tasks, where they have recently surpassed human level accuracy on some benchmarks. A CNN is a type of artificial neural network that can learn features of datasets they are trained on in a way that provides translation invariance. This involves moving layers of small convolutional kernels over a matrix (such as an image) at a specific stride [8].

B. Generative Advesarial Networks

Generative Adversarial Networks [9] (GANs) are unsupervised machine learning models that learn the distribution of a dataset. They utilize two neural networks which are often, but not always, both implemented as CNNs. One of the networks, called the generator, takes random noise as input and learns to generate data that will fool a second network, called a discriminator. At the same time, the discriminator learns the binary classification task that classifies its input as "real", or "forgery" categories. In the case of images, the goal is for the generator to create images that seem to belong in the same distribution as the original images in the dataset [4]. For example, if the dataset was trained on pictures of cats, the generator would try to generate convincing pictures of cats. Although this method was initially designed for images, it can be used for any type of data, including audio, where it has recently generated considerable interest for speech and audio synthesis [10].

This paper explores the idea of using such a generative model, trained on raw audio data, to automatically generate

convincing in-game sounds that are unique to the game experience.

II. Related Work

A. Traditional Methods of Audio Synthesis

Audio synthesis has a rich history preceding the deep learning era. Common methods of augmenting audio signals range in complexity from simple pitch shifting to more sophisticated warping, equalization, and distortion techniques. Many of the traditional techniques have origins in analogue signal processing, where physical circuits were used to distort signals. Software has taken the place of much of the hardware previously used, having proved more flexible and cost effective. Audio engineers and composers are responsible for creating audio scenes and scores which meet the needs of a game. Considerable experience is required to utilize signal processing tools effectively, commonly amounting to years of specialized training. In large budget games, it is common for composers to record live audio for use in-game [11]. This is a large, laborious task requiring expensive equipment, well trained operators and sometimes the use of voice actors. In smaller budget indie games, composition of scores using electronic means is widespread [11]. Open source sound libraries provide another means of acquiring audio, somewhat inexpensively. Usually customization of pre-recorded audio is required to meet the acoustic needs of a game. This requires substantial post-processing work to be carried out by audio engineers, even though time is saved that would have otherwise been spent recording.

B. Deep Learning Methods of Audio Synthesis

Deep learning approaches to audio synthesis do not yet produce results as sophisticated as those seen in imaging tasks, but there have been some notable advancements in recent years. Inspired by generative models for images, WaveNet [12] is a deep neural network designed to generate raw audio waveforms. It has been applied successfully to speech generation tasks and music generation tasks as well as phoneme recognition when employed in a discriminative form. Baidu Research implements a variant of WaveNet in the audio synthesis block of their Deep Voice text-to-speech (TTS) system [13]. Deep Voice uses the same processing structure as traditional TTS systems but with all components replaced by neural networks trained on simple audio features like phonemes and fundamental frequency. These networks are easily adapted to new datasets, unlike traditional pipelines that require complicated tuning of hand-engineered features. Kalchbrenner et al. achieve faster than real-time audio synthesis using a sequential model, WaveRNN [13]. They also demonstrate that high fidelity audio generation is feasible on low-power mobile CPUs. This is particularly relevant to the gaming industry, where mobile gaming represents half of the global market [14].

III. Methodology

The goal of this research is to investigate methods of audio synthesis employing deep neural networks. Furthermore, we compare these methods with conventional methods of audio production. This comparison gives perspective on the applicability of deep learning to audio production in gaming.

A. Dataset

We use the Audio Set dataset [15] in the experiments outlined in this paper. It contains 1.7 million 10 second audio clips labelled on 632 audio event classes. The audio is taken from YouTube videos and has been labelled manually by humans. The dataset is available as CSV files identifying start and end times of each 10 second segment in a YouTube video or as 128-dimensional audio features stored as TensorFlow Record files.

B. Synthesis of audio using traditional methods

The most comparable method of audio synthesis to a generative adversarial network, in terms of output characteristics, is cross-synthesis. Taking two signals, a carrier and a modulator, a magnitude spectrum transform is performed to impress spectral characteristics of the modulator onto the carrier [16]. Some potential drawbacks of this approach when compared to the method proposed in this work are:

1) it is typically best when the carrier is broadband in quality, so that its energy can fully occupy the time-varying spectral envelope of the modulator, and
2) it is not clear how to easily extend this technique to more than two input signal types at a time.

Other methods of audio synthesis include augmenting an existing sound with transforms such as pitch shifting, time-stretching, or applying various filtering techniques. However, these are typically applied to a single sound of a single type (e.g., the sound of a dog barking), rather than combining multiple sounds of two or more signal types (e.g., combining dog sounds with speech sounds). We compare the performance of our GAN model against methods such as those mentioned here, in the context of synthesis of unique audio.

C. Synthesis of audio using GANs

Generative adversarial networks produce unique samples of audio by learning from a training data set. If this training data set contains multiple types of audio, it is possible that GAN outputs will exhibit characteristics of all these types morphed together in a hybrid fashion. This will achieve an effect similar to cross-synthesis, where information from two signals are combined into one. However, it is also readily noticed that this technique may be easily extended to generate hybrids of sounds from more than two types, by simply adding sounds of these types in equal or differing proportions in the training data set. Moreover, the relative proportion of a given type of sound in the set may be increased or decreased, in order to magnify or lessen the effect of that type of sound in the resulting hybrid output. In our experiments, we take a WaveGAN [10] model and train it on classes of audio from the Audio Set dataset.

The authors of the WaveGAN paper trained their network separately on multiple datasets with as little as 0.3 hours of data. A majority of Audio Sets classes contain at least as much audio, easily meeting our needs. We do not attempt to use the entire dataset, which contains over 5k hours of audio. For the experiments of this paper, without precluding our technique's ability to use sound types in unequal proportion, we aim, for simplicity of demonstration, to use as close to an equal proportion as possible between classes when multiple classes are included in the training data.

IV. Experiments

A. Duplication of WaveGAN experiments

The experimental steps outlined in the WaveGAN paper [10] are followed to verify the capabilities of the model. No hyperparameters are modified, and training is run for the same number of iterations (200000). For this experiment, training of a WaveGAN model is carried out on a subset of the Speech Commands Dataset [17].

B. Synthesis of a variety of data

An important question is whether WaveGAN can perform well synthesizing a variety of data. This includes data unlike what is seen in the original paper, as well as data which is an aggregation of multiple classes. The steps from the first experiment are applied to data from new sources and a mix of sources. Classes 'dog' and 'violin' are retrieved from AudioSet [15] and used individually and in combination to train WaveGAN models. The code used to retrieve and sort the AudioSet dataset is publicly available online .

Data from multiple classes, and without labels, is used to train a single model in this experiment. This examines the potential of a WaveGAN model to synthesize audio which sounds like a 'mixture' of multiple classes. Classes for this experiment were arbitrarily chosen by the authors.

C. Interpolation using latent space mapping

In the previous subsection, the problem of synthesizing unique audio by training on two or more classes to generate sounds that are hybrids of two or more classes was addressed. In that approach, random latent vectors are input to the generator to create a sound sample with the intent that some subset of these samples will have the desired audio properties (ie. a violin sound that has the rhythm of a dog bark).

Once a generator has been trained, it is also possible to learn the reverse mapping. This mapping allows one to find an input vector that, when given to a GAN, will produce a sound closest to a provided sound sample. By using this technique, it is possible to interpolate between two or more sounds in the latent space. This inverse latent space mapping has been used successfully in computer vision tasks [18].

In this experiment, interpolation in the latent space is used to generate a range of sound samples between two chosen sounds.

https://github.com/aoifemcdonagh/audioset_processing

V. Results and Evaluation

The quality of generated audio varies for each set of data used to train the model. Generally, output from a model trained on data from a single class is superior to output from a model trained on data from multiple classes. It is unsurprising that there would be some difficulty with reproducing samples from a data set where there are multiple classes present but no explicit labels associated with them. Exact reproduction is also complicated by the random input vectors, which means the same dataset could result in a different slightly different set of sounds each time a network is trained on them.

Another factor that contributes to the output of the model is the quality of the training data. The numerical digits subset of the Speech Commands Dataset [17] (SC09) is a set of high quality data with little background noise. Each example in this set contains only one digit. On the other hand, AudioSet [15] examples can contain sound from multiple sources (classes). Problems arise when, for example, a ten-second clip contains only 1 second of the desired audio class, or when a significant amount of mis-labeled data makes it into the set used for training. These factors could contribute to the noisy, somewhat chaotic sounding output from models trained on data from AudioSet.

Generative models are notoriously difficult to evaluate due to the subjective nature of their output. The 'realness' of a generated image or piece of audio can be judged instinctively by a human observer, yet remains an almost impossible task for a machine. There are no robust techniques to objectively measure generative model output quality apart from some ad-hoc methods like Inception Scoring [19]. This technique has fundamental flaws which are outlined in recent work [20] and by the creators of WaveGAN [10] itself. Due to the shortcomings in qualitative evaluation metrics, the best method of evaluating generated samples is currently to listen to them. A more rigorous approach that does not contradict human judgment has yet to be developed and remains an important aspect of future work.

A subjective scoring of each models output was performed by a human listener. A model's score at a given training iteration is given by the percentage of output samples which resemble the class of data it is trained on. The scores for each model are given below in Table I. For the model trained on two audio classes, a pass is given when the output sample resembles either of the original classes or is considered a credible mix of the two.

Model	100k	150k	200k	250k
SC09	63%	53%	75%	-
Dog	50%	41%	28%	-
Violin	28%	28%	47%	-
Dog & Violin mix	34%	34%	44%	25%

TABLE I: Pass rates of models at various stages of training (given in number of iterations)

An interesting contradiction occurs when considering what constitutes as a 'pass' for the model trained on the digits subset

of the Speech Commands Dataset[17] (SC09). Some examples produced by the model resemble mixtures of multiple digits. In this case, mixed words are undesirable outcomes, because a listener would expect to hear one of the ten digits in the training data. However, in the case of our 'mixed' model, the goal is to produce samples which exhibit characteristics of all classes included in the training data. This anomaly supports the idea that GANs can be used to synthesize audio with multi-class characteristics as a desirable trait. For the purposes of comparing with the samples provided by the creators of WaveGAN [10] in the case of the SC09 dataset, a pass is considered to be an output sample which resembles a single digit appearing in this set.

Our methods, which employed a single GAN model, performed well on multiple datasets. This is an encouraging step as it is common for GANs to perform poorly when a new dataset is used without hyperparameter tuning.

VI. Conclusions

Through training a generative adversarial network on a set of raw audio containing data from multiple classes, audio samples are produced that exhibit combined features of these classes. Furthermore, using the latent space of the deep learning audio synthesis model, audio samples were produced that exhibit features of multiple audio classes. The resulting sound files will be made available to attendees at IEEE GEM Conference 2018.

It has been shown that GANs can be used to generate a wide range of audio without relying on spectrograms or predefined filters. The experiments in this paper demonstrate that they can also be used for synthesis of novel audio, although subjective sound quality could be improved and objective assessment remains an open problem.

The methods applied in this paper are applicable to the production of audio for games, particularly in the case where sound effects need to sound unlike anything existing in reality. Such a need can arise in games with fantasy or extra-terrestrial themes where it could break the immersive sensation if characters and objects sound overly earth-like.

References

[1] C. Jennett, A. L. Cox, P. Cairns, S. Dhoparee, A. Epps, T. Tijs, and A. Walton, "Measuring and defining the experience of immersion in games," *International Journal of Human-Computer Studies*, vol. 66, no. 9, pp. 641–661, September 2008.

[2] E. Brown and P. Cairns, "A grounded investigation of game immersion," in *Extended abstracts of the 2004 conference on Human factors and computing systems - CHI '04*. New York, New York, USA: ACM Press, 2004, p. 1297.

[3] ESA, "2007 Essential Facts about the Computer and Video Game Industry," The Entertainment Software Association, Tech. Rep., 2017. [Online]. Available: http://www.theesa.com/article/2017-essential-facts-computer-video-game-industry/

[4] K. Jørgensen, "Left in the dark: playing computer games with the sound turned off," in *From Pac-Man to Pop Music: Interactive Audio in Games and New Media*. Ashgate, 2008, ch. 11, pp. 163–176.

[5] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[6] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[7] J. Lemley, S. Bazrafkan, and P. Corcoran, "Deep Learning for Consumer Devices and Services: Pushing the limits for machine learning, artificial intelligence, and computer vision." *IEEE Consumer Electronics Magazine*, vol. 6, no. 2, pp. 48–56, 2017.

[8] A. Goodfellow, Ian, Bengio, Yoshua, Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org/

[9] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[10] C. Donahue, J. McAuley, and M. Puckette, "Synthesizing audio with generative adversarial networks," *CoRR*, vol. abs/1802.04208, 2018. [Online]. Available: http://arxiv.org/abs/1802.04208

[11] B. Schmidt, "GameSoundCon Game Audio Industry Survey 2017," GameSoundCon, Tech. Rep., 2017. [Online]. Available: https://www.gamesoundcon.com/single-post/2017/10/02/GameSoundCon-Game-Audio-Industry-Survey-2017

[12] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *CoRR*, vol. abs/1609.03499, 2016. [Online]. Available: http://arxiv.org/abs/1609.03499

[13] N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimberg, A. van den Oord, S. Dieleman, and K. Kavukcuoglu, "Efficient Neural Audio Synthesis," feb 2018. [Online]. Available: http://arxiv.org/abs/1802.08435

[14] Newzoo, "Global Games Market Report," Newzoo, Tech. Rep., 2018. [Online]. Available: https://newzoo.com/insights/articles/global-games-market-reaches-137-9-billion-in-2018-mobile-games-take-half/

[15] J. F. Gemmeke, D. P. W. Ellis, D. Freedman, A. Jansen, W. Lawrence, R. C. Moore, M. Plakal, and M. Ritter, "Audio Set: An ontology and human-labeled dataset for audio events," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, mar 2017, pp. 776–780.

[16] C. Roads and J. Strawn, "Linear Predictive Coding," in *The Computer Music Tutorial*. MIT Press, 1996, ch. 2, pp. 200–212.

[17] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *CoRR*, vol. abs/1804.03209, 2018. [Online]. Available: http://arxiv.org/abs/1804.03209

[18] S. Bazrafkan, H. Javidnia, and P. Corcoran, "Face synthesis with landmark points from generative adversarial networks and inverse latent space mapping," *arXiv preprint arXiv:1802.00390*, 2018.

[19] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," in *Advances in Neural Information Processing Systems*, 2016, pp. 2234–2242.

[20] S. Barratt and R. Sharma, "A note on the inception score," *arXiv preprint arXiv:1801.01973*, 2018.

Appendix B

WaveGAN Training Hyperparameters

The set of hyperparameters used in experiments in Chapter 3 for training WaveGAN are as follows;

- *sample rate (F_s)*: 16000
- *window length*: 16384
- *latent space vector length*: 100
- *kernel length*: 25
- *training batch size*: 64
- *Number of discriminator updates per generator update*: 5

Appendix C

Creation of TFRecord files

```
if __name__ == '__main__':
    import argparse
    import glob
    import os
    import random
    import numpy as np
    import tensorflow as tf

    parser = argparse.ArgumentParser()
    parser.add_argument('in_dir', type=str) # Path to audio files to make TFRecords
    parser.add_argument('out_dir', type=str) # Path for storing TFRecords
    parser.add_argument('--name', type=str)
    parser.add_argument('--ext', type=str)
    parser.add_argument('--fs', type=int)
    parser.add_argument('--nshards', type=int)
    parser.add_argument('--slice_len', type=float)

    parser.set_defaults(
        name='examples',
        ext='wav',
        fs=16000,
        nshards=64,
        slice_len=1.5)
    args = parser.parse_args()

    # Get list of audio files in in_dir
    audio_fps = glob.glob(os.path.join(args.in_dir, '*.{}'.format(args.ext)))
    random.shuffle(audio_fps) # Shuffle items in list of files
    # num of input files to go into each TFRecord file
    npershard = int(len(audio_fps) // (args.nshards - 1))

    slice_len_samps = int(args.slice_len * args.fs) # Get # samples in each slice

    # The following 4 lines of code executed in the Tensorflow session
    # Placeholder object to be assigned value in Tensorflow session
    audio_fp = tf.placeholder(tf.string, [])
    audio_bin = tf.read_file(audio_fp) # Read a file
    # Extract samples from audio file
    samps = tf.contrib.ffmpeg.decode_audio(audio_bin, args.ext, args.fs, 1)[:, 0]
    # Get frames of audio of length 'slice_len_samps' made up of samples in 'samps'
    slices = tf.contrib.signal.frame(samps, slice_len_samps, slice_len_samps,
        axis=0, pad_end=False)
```

```
sess = tf.Session() # Start Tensorflow session

# range between 0, #files in in_dir, step every npershard
for i, start_idx in enumerate(range(0, len(audio_fps), npershard)):
    # Generate filename for current TFRecord file
    shard_name = '{}-{}-of-{}.tfrecord'.format(args.name,
        str(i).zfill(len(str(args.nshards))), args.nshards)
    # full file path to current TFRecord file
    shard_fp = os.path.join(args.out_dir, shard_name)

    # Open writer to file path of current TFR
    writer = tf.python_io.TFRecordWriter(shard_fp)

    # iterate through 'npershard' files to go into current TFRecord file
    for _audio_fp in audio_fps[start_idx:start_idx + npershard]:
        audio_id = os.path.splitext(os.path.split(_audio_fp)[1])[0]

        try:
            # run session to generate slices for current file
            _slices = sess.run(slices, {audio_fp: _audio_fp})
        except:
            continue

        if _slices.shape[0] == 0 or _slices.shape[1] == 0:
            continue

        # Write slices to Example object
        for j, _slice in enumerate(_slices):
            example = tf.train.Example(features=tf.train.Features(feature={
                'id':
                    tf.train.Feature(bytes_list=tf.train.BytesList(value=audio_id)),
                'slice':
                    tf.train.Feature(int64_list=tf.train.Int64List(value=[j])),
                'samples':
                    tf.train.Feature(float_list=tf.train.FloatList(value=_slice))
            }))

            # Write Example object to current TFRecord file
            writer.write(example.SerializeToString())
    # Close writer object to current TFRecord file
    writer.close()
sess.close()
```

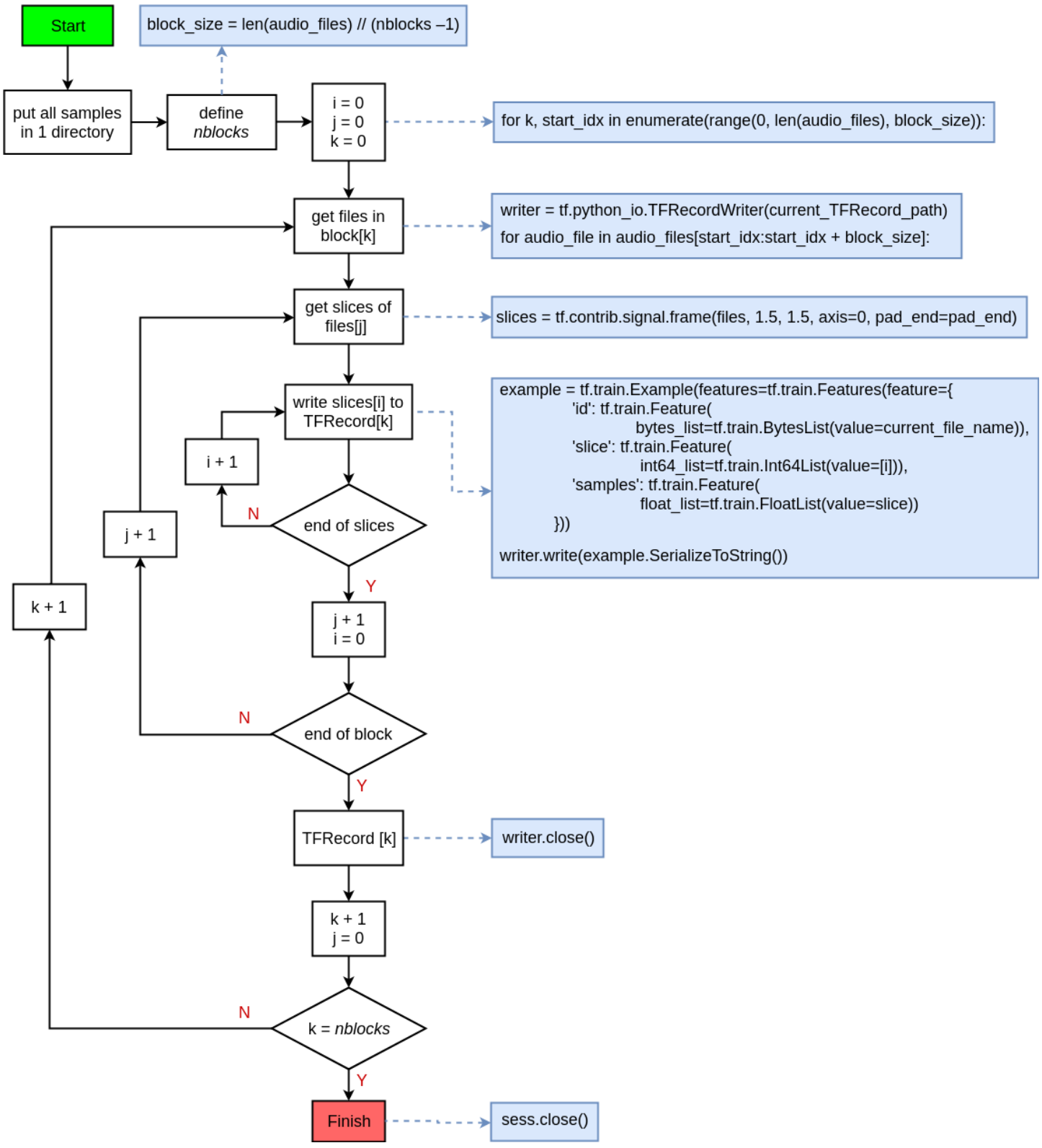


Figure C.1: **Creation of TFRecord files with code snippets.** This diagram shows where in the TFRecord creation process each code snippet belongs to.

Appendix D

Calculation of GAN model loss using WGAN-GP

Careful calculation of the loss of Generator and Discriminator models in GANs helps to prevent training collapse. The WGAN-GP technique is one method which priorities training stability. It is used in the work in Chapter 3 for training WaveGAN models.

In the experiments in this work the following values are used:

- *WGAN-GP* λ: 10
- D updates per G update (*nupdates*): 5
- *Optimizer*: Adam $\alpha = 1e-4, \beta_1 = 0.5, \beta_2 = 0.9$

Python code for calculating loss, setting up an optimizer, and training models using the WGAN-GP method used in experiments in Chapter 3 is included below. The code is made available in an expanded form by the authors of WaveGAN [32] on GitHub [1].

[1] `https://github.com/chrisdonahue/wavegan`

```
# Set up loss variables
G_loss = -tf.reduce_mean(D_G_z)
D_loss = tf.reduce_mean(D_G_z) - tf.reduce_mean(D_x)
alpha = tf.random_uniform(shape=[train_batch_size, 1, 1], minval=0.,
    maxval=1.)
differences = G_z - x
interpolates = x + (alpha * differences)
with tf.name_scope('D_interp'), tf.variable_scope('D', reuse=True):
    D_interp = WaveGANDiscriminator(interpolates,
        **args.wavegan_d_kwargs)

LAMBDA = 10
gradients = tf.gradients(D_interp, [interpolates])[0]
slopes = tf.sqrt(tf.reduce_sum(tf.square(gradients),
    reduction_indices=[1, 2]))
gradient_penalty = tf.reduce_mean((slopes - 1.) ** 2.)
D_loss += LAMBDA * gradient_penalty

# Set up Adam Optimizer
G_opt = tf.train.AdamOptimizer(
           learning_rate=1e-4,
           beta1=0.5,
           beta2=0.9)
D_opt = tf.train.AdamOptimizer(
           learning_rate=1e-4,
           beta1=0.5,
           beta2=0.9)

# Create training operations (used in monitored training session)
# G_vars, D_vars are lists of trainable variables in G, D which will be
# trained by the optimiser.
G_train_op = G_opt.minimize(G_loss, var_list=G_vars,
    global_step=tf.train.get_or_create_global_step())
D_train_op = D_opt.minimize(D_loss, var_list=D_vars)

# Begin training G and D with WGAN-GP parameters
with tf.train.MonitoredTrainingSession(
       checkpoint_dir=args.train_dir,
       save_checkpoint_secs=args.train_save_secs,
       save_summaries_secs=args.train_summary_secs) as sess:
    while True:
        # Train D 'nupdates' times per 1 G update
```

```
    for i in xrange(args.nupdates):
        sess.run(D_train_op)
    sess.run(G_train_op) # 1 G update per 'nupdates' D updates
```

Appendix E

Material Absorption Coefficients

material	**125 Hz**	**250 Hz**	**500 Hz**	**1000 Hz**	**2000 Hz**	**4000 Hz**
brick	0.03	0.03	0.03	0.04	0.05	0.07
carpet	0.02	0.06	0.14	0.37	0.6	0.65
ceramic	0.01	0.01	0.01	0.01	0.02	0.02
fabric	0.07	0.31	0.49	0.75	0.7	0.6
foliage	0.15	0.11	0.1	0.07	0.06	0.07
food	0.15	0.11	0.1	0.07	0.06	0.07
glass	0.35	0.25	0.18	0.12	0.07	0.04
hair	0.25	0.35	0.42	0.46	0.5	0.5
leather	0.25	0.35	0.42	0.46	0.5	0.5
metal	0.1	0.05	0.06	0.07	0.09	0.08
mirror	0.18	0.06	0.04	0.03	0.02	0.02
other	0.5	0.5	0.5	0.5	0.5	0.5
painted	0.013	0.015	0.02	0.03	0.04	0.05
paper	0.013	0.015	0.02	0.03	0.04	0.05
plastic	0.02	0.03	0.03	0.03	0.03	0.02
polishedstone	0.1	0.05	0.06	0.07	0.09	0.08
skin	0.25	0.35	0.42	0.46	0.5	0.5
sky	0.99	0.99	0.99	0.99	0.99	0.99
stone	0.36	0.44	0.31	0.29	0.39	0.25
tile	0.01	0.01	0.01	0.01	0.02	0.02
wallpaper	0.013	0.015	0.02	0.03	0.04	0.05
water	0.008	0.008	0.013	0.015	0.02	0.025
wood	0.28	0.22	0.17	0.09	0.1	0.11

Table E.1: Material absorption coefficients used in experiments in Chapter 4.

Appendix F

Image Segmentation Workflow

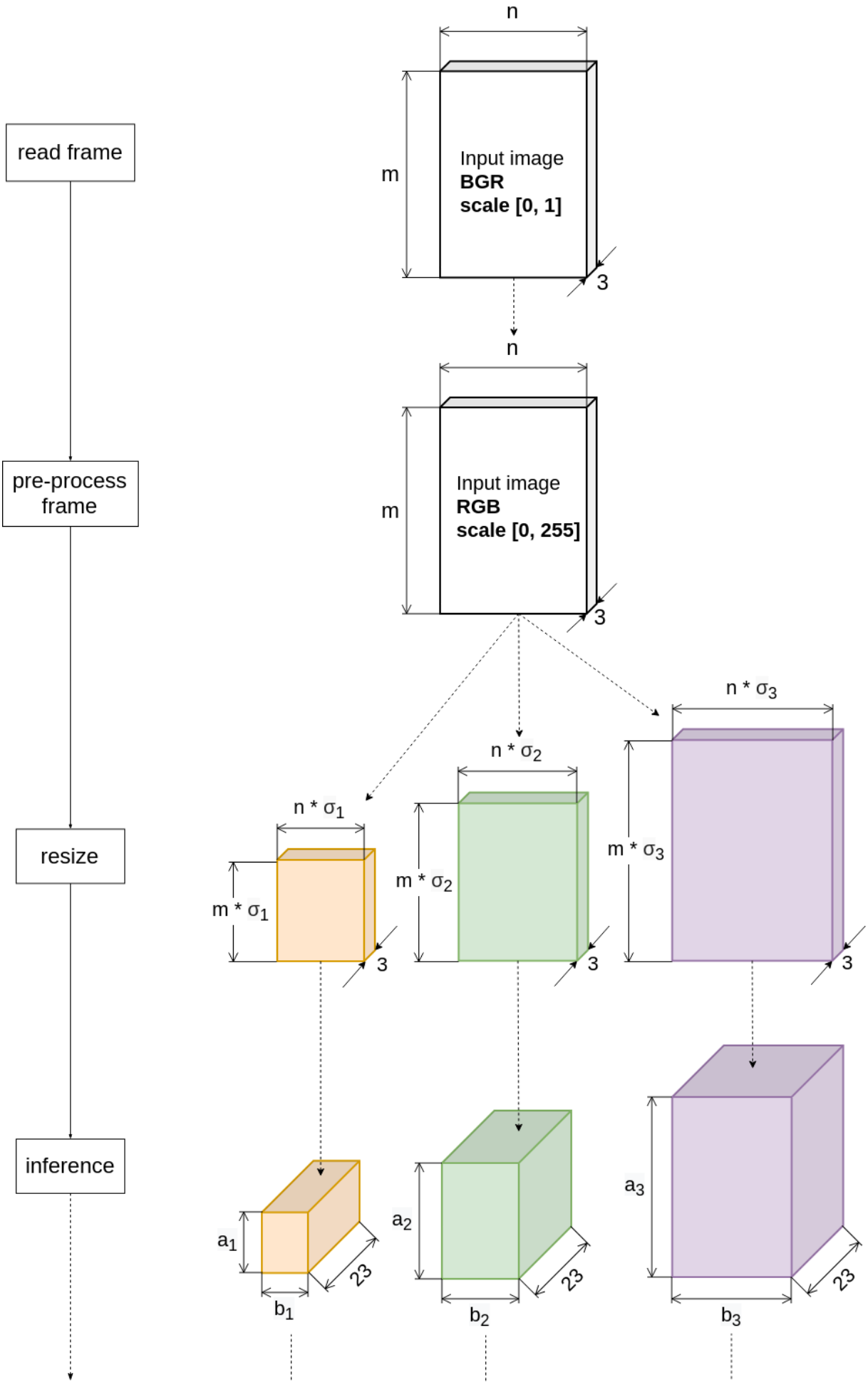
This diagram expands on Figure 4.7 to illustrate the outputs of each step in the segmentation process. The calculation of lengths a and b is shown in Equation F.1 where;

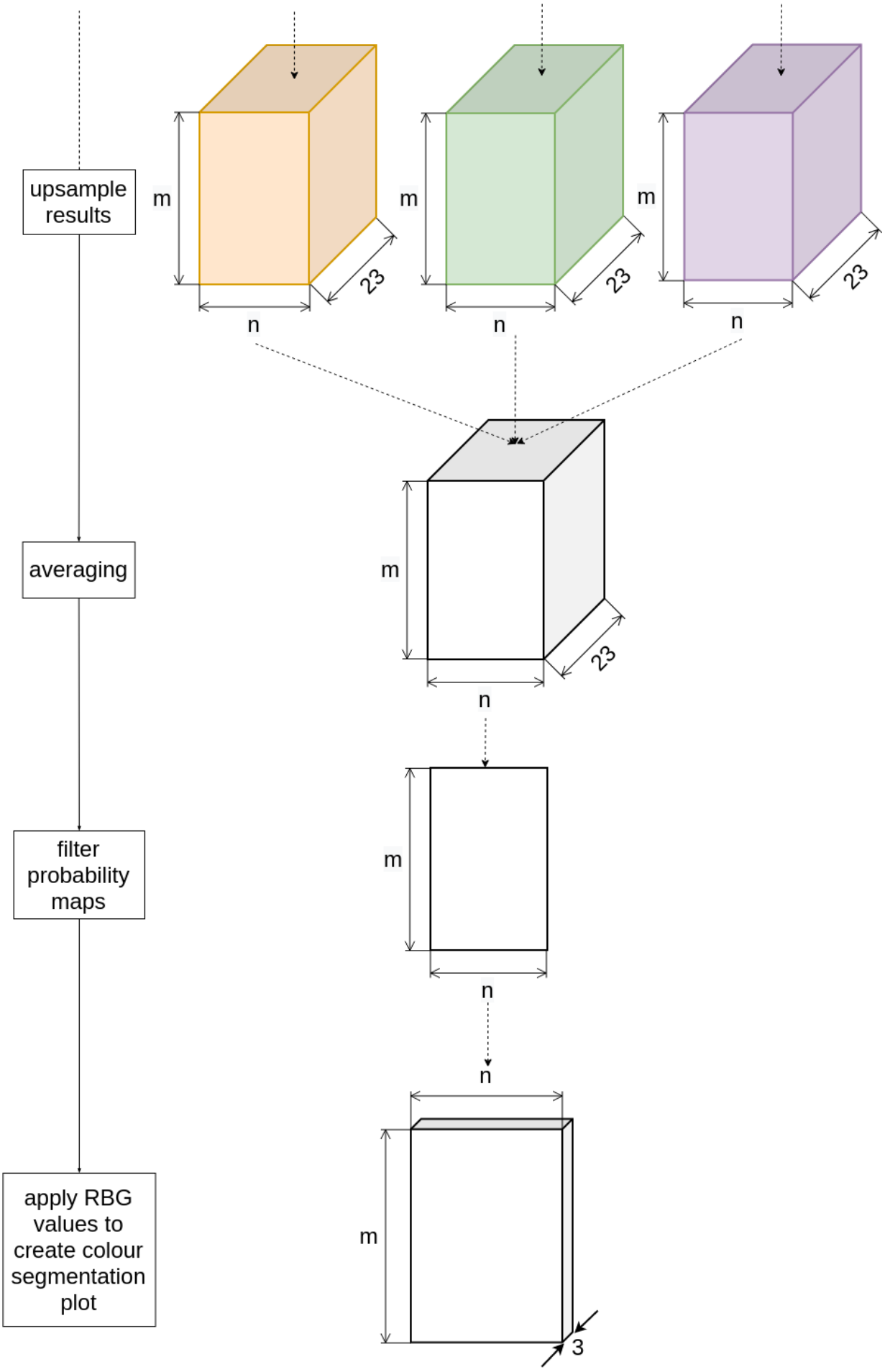
- m, n are input image dimensions
- σ is a vector containing scales for image resizing.
- p is the number of pixels of padding
- s is the stride length
- *224* is the receptive field length of the GoogLeNet model [18] used.

$$a, b = [1 + \frac{(m * \sigma) - 224 + 2p}{s}, 1 + \frac{(n * \sigma) - 224 + 2p}{s}] \tag{F.1}$$

In the experiments in Section 4.1.3.2 the following scale values are used

$$\sigma_1, \sigma_2, \sigma_3 = [\frac{1}{\sqrt{2}}, 1, \sqrt{2}] \tag{F.2}$$

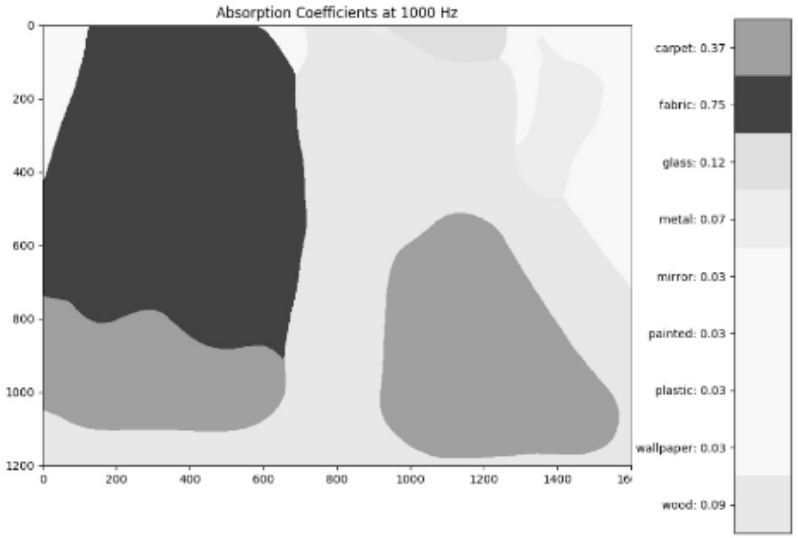

read frame
pre-process frame
resize
inference
n
m
Input image
BGR
scale [0, 1]
3
n
m
Input image
RGB
scale [0, 255]
3
n * σ1
m * σ1
3
n * σ2
m * σ2
3
n * σ3
m * σ3
3
a1
23
b1
a2
23
b2
a3
23
b3

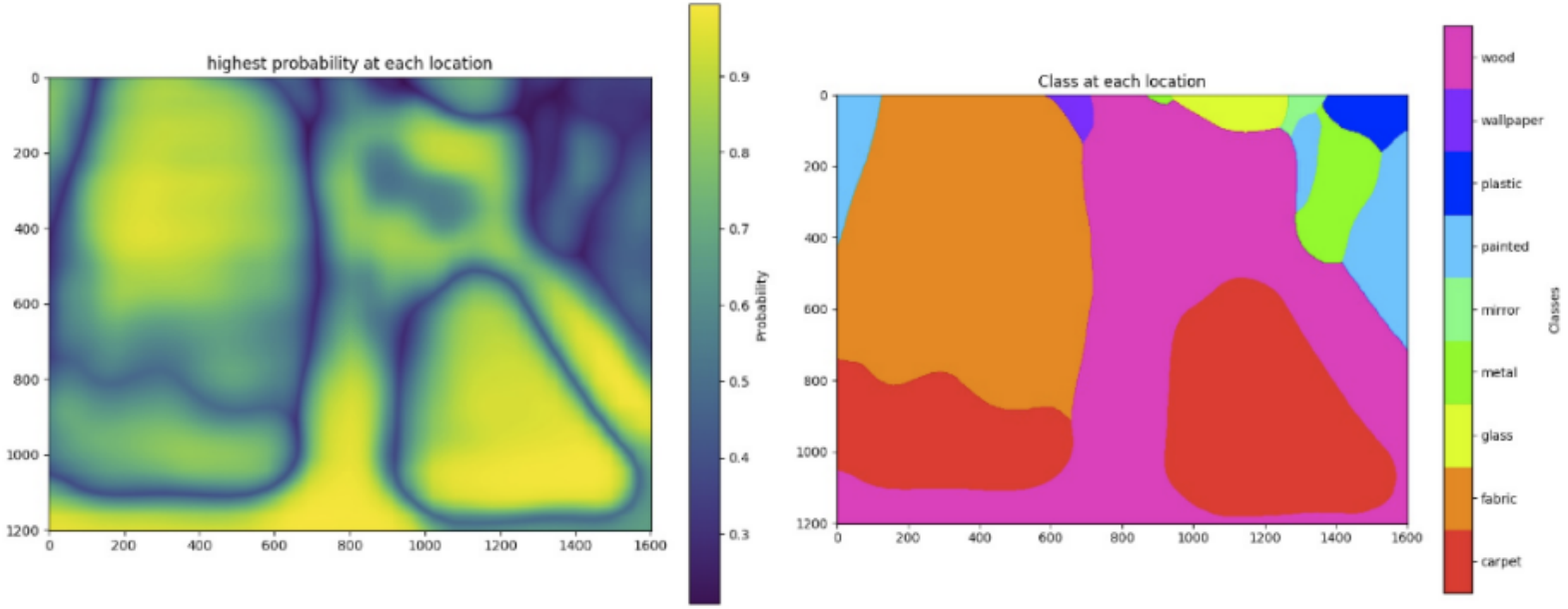

upsample results
averaging
filter probability maps
apply RBG values to create colour segmentation plot
m
n
23
3

Appendix G

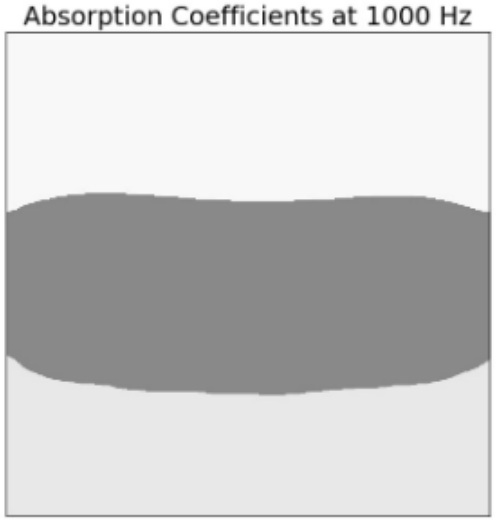
Material Segmentation Results

This section contains examples of segmentation results for various images.

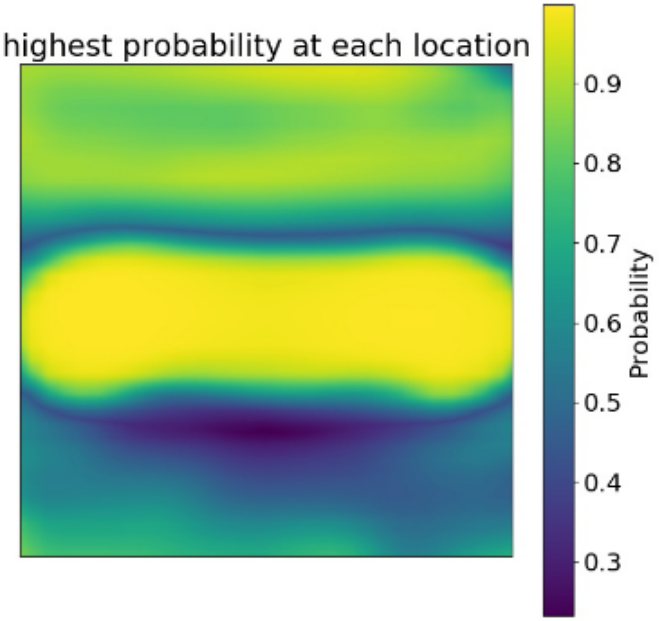

Absorption Coefficients at 1000 Hz
carpet: 0.37
fabric: 0.75
glass: 0.12
metal: 0.07
mirror: 0.03
painted: 0.03
plastic: 0.03
wallpaper: 0.03
wood: 0.09

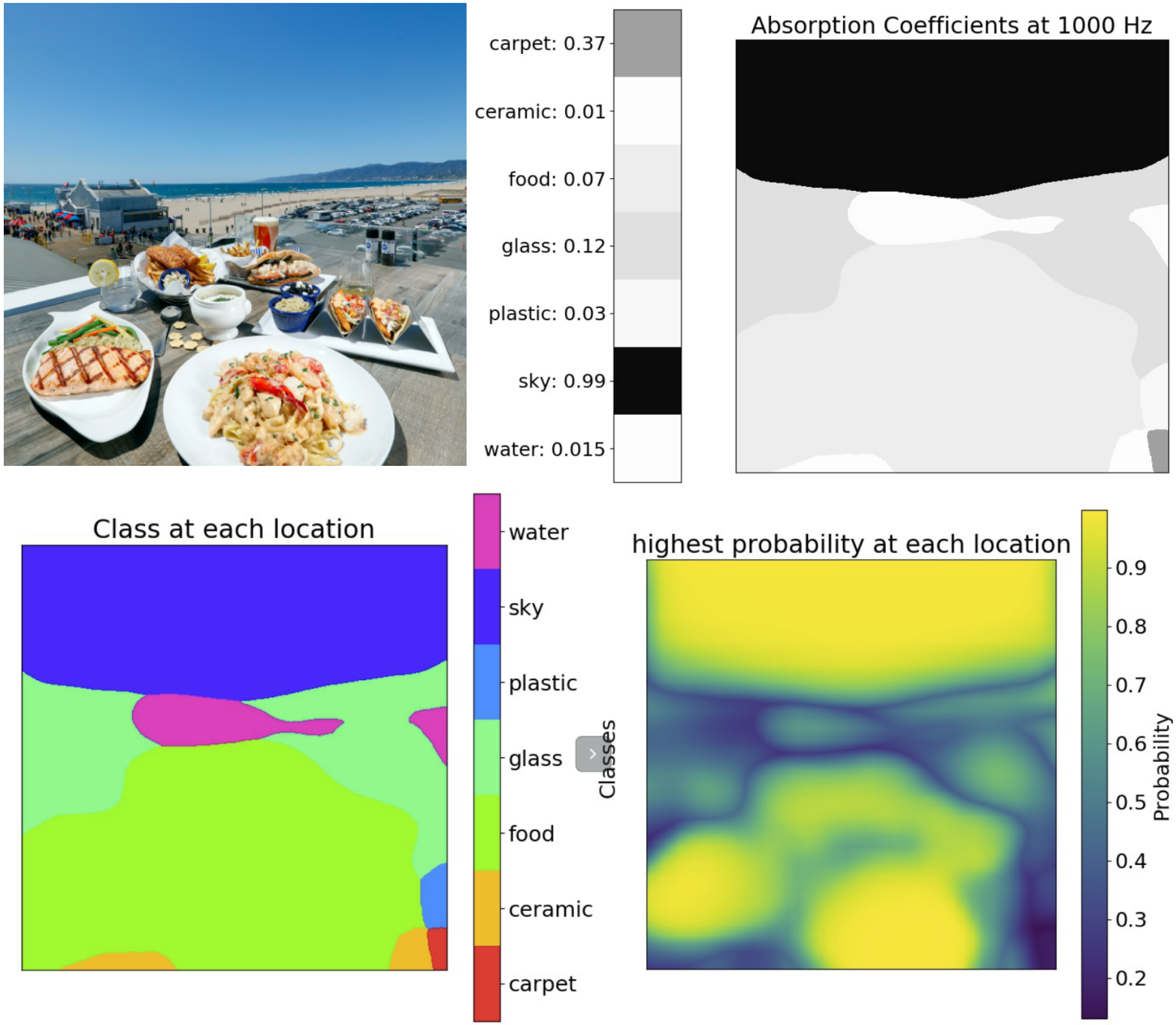

highest probability at each location
Probability
Class at each location
wood
wallpaper
plastic
painted
mirror
metal
glass
fabric
carpet
Classes

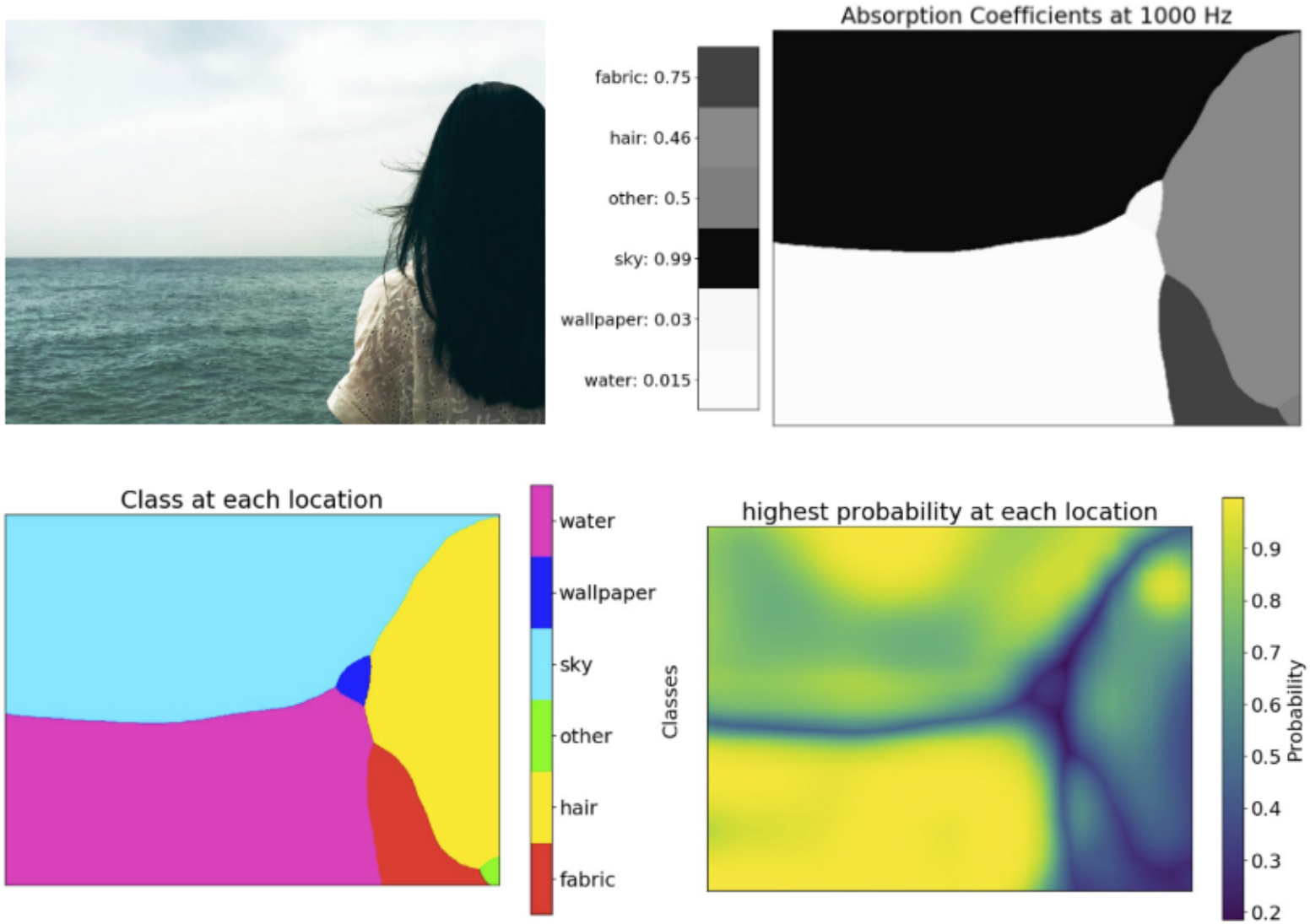

leather: 0.46
painted: 0.03
tile: 0.01
wood: 0.09


Absorption Coefficients at 1000 Hz


Class at each location
wood
tile
painted
leather
Classes


highest probability at each location
0.9
0.8
0.7
0.6
0.5
0.4
0.3
Probability


carpet: 0.37
ceramic: 0.01
food: 0.07
glass: 0.12
plastic: 0.03
sky: 0.99
water: 0.015
Absorption Coefficients at 1000 Hz
Class at each location
water
sky
plastic
glass
food
ceramic
carpet
Classes
highest probability at each location
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
Probability


Absorption Coefficients at 1000 Hz
fabric: 0.75
hair: 0.46
other: 0.5
sky: 0.99
wallpaper: 0.03
water: 0.015
Class at each location
water
wallpaper
sky
other
hair
fabric
highest probability at each location
Classes
Probability
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2

Appendix H

Demonstration GUI

H.1 NCS_SegmentationApp.py

```
from PIL import Image
from PIL import ImageTk
import tkinter as tk
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import threading
from material_segmentation import plotting_utils
import cv2
import os
from openvino.inference_engine import IENetwork, IEPlugin

class SegmentationApp:
    def __init__(self, vc, model_path, pad):
        self.vc = vc # video capture
        self.padding = pad # number of pixels padding
        self.model = model_path # path to NCS compatible model
        self.frame = None
        self.stopVideo = None # Flag for stopping video loop
        self.colorbar_frame = None

        self.root = tk.Tk() # root of GUI
        self.panel = None
        self.colorbar_panel = None
        self.resolution = [500, 700]

        button_frame = tk.Frame()
        button_frame.pack(side="bottom")

        # create a button that will show current frame
        btn_frame = tk.Button(button_frame, text="Show Frame",
            command=self.show_frame)
        btn_frame.pack(side="left", padx=10, pady=10)

        # button for segmenting and showing class map
        btn_segment = tk.Button(button_frame, text="Class map",
            command=self.segment_classes)
        btn_segment.pack(side="left", padx=10, pady=10)

        # button for segmenting and showing absorption coefficient map
        btn_abs = tk.Button(button_frame, text="Heat map",
            command=self.segment_heatmap)
        btn_abs.pack(side="left", padx=10, pady=10)

        # Button to start video stream display
        btn_start_display = tk.Button(button_frame, text="start video",
            command=self.start_video)
        btn_start_display.pack(side="right", padx=10, pady=10)

        # set a callback to handle when the window is closed
        self.root.wm_title("Material Segmentation")
        self.root.wm_protocol("WM_DELETE_WINDOW", self.on_close)

        # Set up NCS
        # Set up model once so it doesn't have to be loaded multiple times
```

```
        model_xml = model_path
        model_bin = os.path.splitext(model_xml)[0] + ".bin"

        # Plugin initialization for Movidius stick
        plugin = IEPlugin(device="MYRIAD")

        # Read IR
        net = IENetwork(model=model_xml, weights=model_bin)

        self.input_blob = next(iter(net.inputs))

        n = 1
        c = 3
        h = self.resolution[0]
        w = self.resolution[1]
        # Reshape network input layer
        net.reshape({self.input_blob: (n, c, h, w)})

        # Loading model to the plugin
        self.exec_net = plugin.load(network=net)

    def video_loop(self):
        """
        Function which displays frames from video stream
        """
        # keep looping over frames until we are instructed to stop
        while not self.stopVideo.is_set():
            # grab the frame from the video stream
            _, self.frame = self.vc.read()

            # If the frame is too large to plot, make it smaller.
            if self.frame.shape[0] > 500 or self.frame.shape[1] > 800:
                self.frame = cv2.resize(self.frame, (1280, 720),
                    interpolation=cv2.INTER_AREA)

            frame = cv2.cvtColor(self.frame, cv2.COLOR_BGR2RGB) # Convert to RGB
                before plotting

            frame = Image.fromarray(frame)
            frame = ImageTk.PhotoImage(frame)

            # if the panel is not None, we need to initialize it
            if self.panel is None:
                self.panel = tk.Label(image=frame)
                self.panel.image = frame
                self.panel.pack(side="left", padx=10, pady=10)

            # otherwise, simply update the panel
            else:
                self.panel.configure(image=frame)
                self.panel.image = frame

    def segment(self, map_type):
```

```
        """
        Function for performing and plotting segmentation
        """
        # Check if there are any Canvas objects in GUI children and destroy them
        for child in list(self.root.children.values()):
            if child.widgetName == 'canvas':
                child.destroy()

        try:
            # convert current frame to RGB for processing
            frame = cv2.cvtColor(self.frame, cv2.COLOR_BGR2RGB)

            in_frame = cv2.resize(frame, (self.resolution[1], self.resolution[0])) #
                resize image dimensions
            in_frame = in_frame.transpose((2, 0, 1)) # Change data layout from HWC
                to CHW
            in_frame = in_frame.reshape((1, 3, self.resolution[0],
                self.resolution[1])) # adding n dimension
            self.exec_net.infer(inputs={self.input_blob: in_frame}) # perform
                inference

            # Parse results of inference request
            results = self.exec_net.requests[0].outputs
            # Extract probability maps from results
            results = plotting_utils.get_average_prob_maps_single_image(results,
                [self.resolution[0], self.resolution[1]])

            engine = plotting_utils.PlottingEngine() # Instantiate PlottingEngine
                object
            engine.set_colormap(map_type=map_type, freq=1000) # set colormap type
                for plotting

            # Generate plot from results using PlottingEngine
            pixels, colorbar, _ = engine.process(results)

            # Get pixels in format suitable for TkInter GUI
            image = Image.fromarray(pixels)
            image = ImageTk.PhotoImage(image)

            # Put colorbar in GUI
            colorbar = FigureCanvasTkAgg(colorbar, master=self.root)
            colorbar.get_tk_widget().pack(side="right", padx=10, pady=10)

            # Update panel with segmented image.
            self.panel.configure(image=image)
            self.panel.image = image

            self.root.update_idletasks()

        except RuntimeError:
            print("[INFO] caught a RuntimeError")

    def show_frame(self):
```

```
        """
        Function which will show the current frame in GUI
        """
        try:
            frame = cv2.cvtColor(self.frame, cv2.COLOR_BGR2RGB)
            image = Image.fromarray(frame)
            image = ImageTk.PhotoImage(image)
            # If no panel set up in GUI (i.e. just initialised)
            if self.panel is None:
                self.panel = tk.Label(image=image)
                self.panel.image = image
                self.panel.pack(side="left", padx=10, pady=10)
            # otherwise, simply update the panel
            else:
                self.panel.configure(image=image)
                self.panel.image = image
        except RuntimeError:
            print("[INFO] caught a RuntimeError")

    def start_video(self):
        # Check if any Canvas objects in GUI children and destroy them
        for child in list(self.root.children.values()):
            if child.widgetName == 'canvas':
                child.destroy()

        # Start thread to display frames from video stream
        self.stopVideo = threading.Event()
        self.video_thread = threading.Thread(target=self.video_loop, args=())
        self.video_thread.start()

    def segment_classes(self):
        # Stop reading from video stream and segment current frame
        self.stopVideo.set()
        self.segment(map_type="classes")

    def segment_heatmap(self):
        # Stop reading from video stream and segment current frame
        self.stopVideo.set()
        self.segment(map_type="absorption")

    def on_close(self):
        # set the stop event, cleanup the camera, and allow the rest of
        # the quit process to continue
        print("[INFO] closing...")
        self.stopVideo.set()
        self.vc.release()
        self.root.destroy()
```

H.2 run_GUI.py

```
import logging as log
import caffe
import sys
import argparse
import os
import cv2
# Use segmentation app designed for NCS
from ncs_demos.NCS_SegmentationApp import SegmentationApp

def build_argparser():
    parser = argparse.ArgumentParser()
    parser.add_argument("-m", "--model", help="Path to an .xml file with
        a trained model.", type=str)
    parser.add_argument("-i", "--input", help="Input, 'cam' or path to
        image", required=True, type=str)
    parser.add_argument("-p", "--padding", help="Number of pixels of
        padding to add", type=int, default=0)
    return parser

if __name__ == '__main__':
    args = build_argparser().parse_args()
    # If video stream is camera
    if args.input == 'cam':
        input_stream = 0
    else: # if video stream is from file
        input_stream = args.input
        assert os.path.isfile(args.input), "Specified input file doesn't
            exist"
    # initialise VideoCapture object
    video = cv2.VideoCapture(input_stream)
    # start the app
    sa = SegmentationApp(video, args.model, args.padding)
    sa.root.mainloop()
```