Alternative Methods to Determine the Role of Intermediate Hidden Layer Neurons in the Generative Model

William Braithwaite

St Hugh's College

University of Oxford

A thesis submitted for the degree of

M.Sc. in Mathematical Modelling and Scientific Computing

Trinity Term 2022

Acknowledgements

First and foremost, I would like to thank my thesis supervisors Professor Jared Tanner and Doctor Trevor Wood. Their enthusiasm and expertise made this a thoroughly enjoyable experience.

I would also like to thank our course director Doctor Kathryn Gillow, whose help and guidance throughout the year has been invaluable.

Abstract

Deep learning allows us to solve complex tasks over large data sets. To understand these models, Bau et al. proposed the method of network dissection [3]. This works as an analytical framework to identify the semantics of individual units in image generation and classification models. In this report, we will investigate the feasibility of applying their method to text-to-speech (TTS) models. We hope that this will allow us to gain a better understand of them.

To do this, we develop and modify the code and ideas that were used to apply network dissection to image classification and generation networks. We test our method by examining which speech properties we can identify from our generated audio data, how these properties manifest themselves within our network and how well we can transfer these properties to other speech we generate.

Contents

Chapter 1

Introduction

With the rise of Amazon Alexa, Google Home and Apple's Siri, human interaction with machines is increasingly taking place via speech. Machines translate our speech to text, process the text to create a response then reply by translating text back into speech through their associated text-to-speech (TTS) system. Current state-of-the-art TTS systems are designed around variational autoencoders (VAEs); for example, see [14] where a VAE is used with learned conditional priors. This allows the VAE to be conditioned on speakers that it has been trained on. The authors of the paper then go on to hypothesise that such a prior could be learned on any condition that influences speech, such as emotion or language. However, in order to do this they would have to have labelled audio data for each of these characteristics to train their network on. We want to do something slightly different.

People are growing tired of their devices only communicating with them through a monotone voice with only a few different accents to choose from, none of which represent them or someone from their community [29]. However, companies see the voice of their TTS systems as a brand and therefore do not want to dispose of them completely [29]. As such, an ideal situation would be to be able to transfer some properties of speech to the speaker we are conditioning on while keeping all other properties constant.

A similar feat has been achieved in the generation of images using generative adversarial networks (GANs). In the paper [3], Bau et al. develop an analytical method by the name of network dissection for identifying and quantifying the roles of individual units of a network by comparing the activations of each unit with human-interpretable pattern-matching tasks, such as the detection of textures or objects. Once they have identified important units for a property they can manually edit the units' activations to change how much of that property that appears in the image,

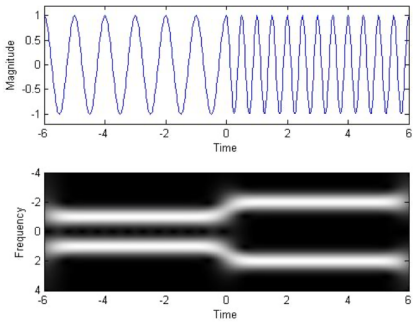
as shown in Figure 1.4. For example, if the GAN was trained to generate images of churches, we might be able to identify units of the network specific to trees using the network dissection method. We could then generate an image using our GAN generator and generate another image using the same input but with the activations of the tree-specific units increased. This should create two images, the latter with more trees but with the rest of the image looking fairly similar.

In this dissertation, we will apply the techniques of Bau et al. to audio generation rather than image generation, with the aim of attaining analogous results. That is, we want to be able to disentangle speech properties which might not have been given or might not be easy to define from the speakers we are conditioning on. We want to be able to alter certain properties of speech, such as accent, while keeping all the other properties the same. We want to be able to continuously modify properties of our generated speech such that we can make properties more subtle or more obvious. In this report we outline development of, what is to the best of our knowledge, the first GAN-based text-to-speech system with speaker characteristics modelled and directly adaptable.

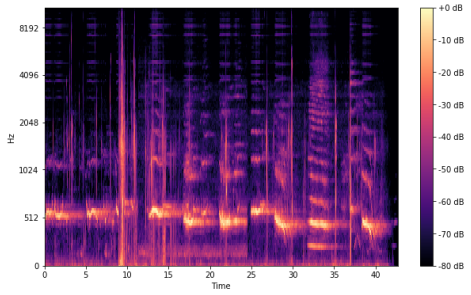
1.1 Modelling Audio

Sound is variations in air pressure propagated through an acoustic wave, hence audio signals can be given as the change in air pressure caused by the wave against time, as shown in Figure 1.1a. The change in pressure is represented by the amplitude of the wave, we can find the time period as the time taken for one full wave cycle and we can find the frequency as one over the time period. This representation can however seem more complex when dealing with sounds such as the human voice or music. Here, it is normal to have multiple frequencies being added together and changing with time, therefore often giving a non-periodic wave.

Rather than giving a one dimensional representation of the data, spectrograms give a two dimensional representation of the data by having both the frequency of the wave and time as independent variables and wave amplitude as the dependent variable. The data is therefore represented like an image, as shown in Figure 1.1b. The advantage of this is that it allows us to use convolutional neural network-based architectures on the data. These have shown great ability in generating synthetic data, for instance synthetic images which resemble a training set [24, 25]. The problem with this representation is that the human ear does not perceive frequencies linearly: it is more sensitive to differences in lower frequencies than higher frequencies [20]. For



(a) An audio waveform and its corresponding spectrogram produced by a Gabor transform [28].



(b) A Mel spectrogram of another signal [5].

Figure 1.1: Possible ways of representing audio.

example, two sound waves with the same amplitude at 100 Hz and 200 Hz sound much further apart than two sound waves with the same amplitude at 1000 Hz and 1100 Hz. To account for this we can scale the frequency logarithmically. A spectrogram with the frequency scaled logarithmically is called a Mel spectrogram.

1.2 Neural Networks

Another thing we need to understand before we can begin our research is what neural networks are and how they can be used as generative models. In general, neural networks are used to approximate (or learn) functions. For example, in classification tasks we assume there is one or more perfect functions which map each input to its class and we are trying to approximate one of these functions the best we can [6]. In this section, we combine the information from the references [6, 12].

1.2.1 Convolutional Layers

There are many different types of layer that can be used to make up a neural network, each typically developed with a purpose in mind. As we saw in the previous subsection, audio signals can be represented in a two dimensional, image-like format using Mel spectrograms. This lends itself to the use of convolutional layers and convolution-based layers, such as dilated convolutions. These have frequently shown a great ability to generate synthetic data as part of a larger network and have been subject to a huge amount of research in both TTS and image generation, amongst

other things [17, 24, 25, 32].

The input to convolutional layers is a two or three dimensional tensor, such as an image. These have the shape (input height)×(input width)×(number of input channels). To understand a convolutional layer, we will need to define the kernel. This is a one, two or three dimensional tensor, that is used to extract features from the input data. We select entries from the input of the layer by "sliding" the kernel along the input tensor, as shown in Figure 1.2. The dimension of a convolution refers to the number of dimensions we slide the kernel in. For a one dimensional convolution, we only slide the kernel in one direction, for example along the width of the input to the layer. For a two dimensional convolution, we slide the kernel along the height and width of the input to the layer, like in Figure 1.2. For each subtensor of selected units we calculate the Frobenius inner product. We then add a bias term to this which is either a scalar for each kernel or a scalar for every channel in each kernel and this gives the output tensor for the layer. This is shown in Figure 1.2 for an input tensor with only one channel and bias of 0. There are also slight generalisations of this to reduce the number of channels of the output, but the premise of multiplying and summing entries of our kernel and input subtensor is still usually what it comes down to. In addition, it is not necessary to have the kernel act on every eligible subtensor of our input data. Instead, we say the stride of the layer is n if we slide the kernel along by n entries between applications. For instance, we would say the convolutional layer in Figure 1.2 has stride 2. This allows us to still gain lots of information on the input layer without spending as much time computing the output [6].

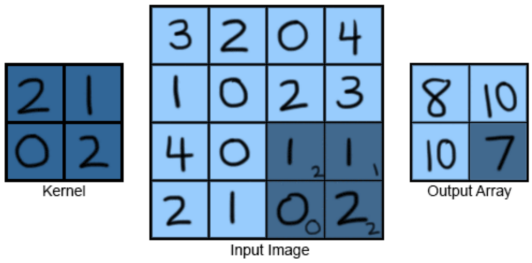


Figure 1.2: This shows a convolutional layer with stride 2 [13].

You might now be wondering how we choose which features we want the kernel to extract. In fact, we do not choose this: the entries of the kernel are weights and the weights and biases are altered by training our network.

From each convolutional layer we will have an output tensor. To this, we normally apply an activation function componentwise to produce another tensor of the same dimension called the feature map and with the entries called the activations of the

convolutional layer. Sometimes we have a residual connection before the activation layer which adds the output of one of the previous layers to the current output before we apply the activation function. Activation functions allow us apply nonlinear functions to each layer's output; there are many different ones each with their own benefit. One of the most common is the sigmoid function, which represents a continuously differentiable approximation to the identity function. We will see why it being continuously differentiable is beneficial when we discuss training. The sigmoid function is given by $\sigma(x) = 1/(1 + e^{-x})$ for all $x \in \mathbb{R}$.

Suppose we have a network of several pairs of convolutional and activation layers. The receptive field of a unit is the units in the previous layer whose outputs are used to calculate the output of that unit. Since we have layers of convolutions, units in the later layers will receive more information from the input of the network than units in the layers before them. In this way, convolutional networks work hierarchically. This may manifest itself in image classification networks as having the early layers identifying corners and edges while later layers will start to combine these to form objects [35].

Variations of convolutions exist, such as dilated convolutions. Rather than selecting adjacent elements of the input tensor to be acted on by the kernel, dilated convolutions work by selecting elements with gaps in between them. This makes us gather information from a larger area of the input for calculating each entry of the output. Ultimately, in our research the type of convolutional layer does not matter: these layers all act similarly.

1.2.2 Training Neural Networks

Neural networks become better at approximating their target functions through a process known as training. In general, what this involves is giving the network inputs which you know the correct output for, quantifying how inaccurate the predictions are and modifying the parameters of the network such that future predictions on the same inputs become more accurate. The hope is that future predictions on all inputs become more accurate as a result of this.

Once we have a large corpus of data we can split it into a training set and a test set. The training set is used to train the model, whereas the test set contains data the network has not been trained on so we can test how accurately our network predicts the output of "unseen" inputs. But how can we quantify the accuracy of a network? Actually, we tend to quantify the inaccuracy of it in that we quantify the

mismatch between our network's predictions and the correct outputs. We use a loss function to do this over a batch of inputs. The lower the loss function, the better our network is at approximating the target function. Now, we can reframe the training as an optimisation problem, hence we can use continuous optimisation algorithms, such as gradient descent or stochastic gradient provided the loss function is continuously differentiable with respect to the parameters of the model. This is why it is beneficial for the sigmoid function to be continuously differentiable: it enables the use of these techniques. These involve calculating the loss function on some collection of training points, finding the gradient of the loss function with respect to the weights and biases of our model and using this information to change the weights and biases such that the loss function on the same collection of points decreases. We then repeat this over all the data in our training set a specified number of times or until our loss function applied to our test set becomes small enough.

Approximating a function or defining a loss function is a clear notion when talking about something like classification tasks since the output of the network will likely be a vector giving the probability of the input belonging to each class. We can use this to quantify the mismatch between our prediction and the actual solution and use the method from the previous paragraph to optimise the parameters to improve future predictions. The notion of approximating a function becomes less clear when we talk about generative tasks. For example, if we want to generate images of churches that look realistic it becomes less clear how we would quantify the mismatch between an image we generated and how realistic of an image of a church it is. We need this mismatch to be given in a quantifiable way such that the generative network can learn from this and future images it generates will be more realistic. Generative adversarial networks (GANs) were developed to deal with this problem.

1.3 Generative Adversarial Networks (GANs)

1.3.1 Introduction to GANs

In the Theories of Deep Learning course, GANs were discussed for one lecture, but never in the tutorials or problem sheets. The fundamental idea behind GANs is to have two networks competing against each other: a generative network and a discriminative network. The job of the discriminator is to predict whether data input to it is data from the training set or data synthesised by the generator. The job of the generator is to fool the discriminator into thinking the data generated by it is

training data. Through training, each of these networks get better at their respective jobs. As such, the generator is indirectly trained to synthesise data that mimics the distribution of data in the training set [7]. This is an interesting idea but how would we describe it mathematically?

Suppose we have a data space $\mathcal{X}$ with a probability distribution ρ_X and a latent space $\mathcal{Z}$ with probability measure ρ_Z. If we define the generator as a measurable map $G : \mathcal{Z} \to \mathcal{X}$, then we can denote the push-forward measure of ρ_Z on $\mathcal{X}$ by ρ_G. This is the density of the random variable $G(Z)$, where Z is distributed according to ρ_Z. However, it is normally different from ρ_X. Lastly, we can define a discriminator by a map $D : \mathcal{X} \to [0, 1]$ and the GAN by a generator-discriminator pair (G, D), with G and D both neural networks trained according to the objectives from the last paragraph [7].

We are now able to write the training objectives mathematically. The discriminator's aim is to distinguish data sampled from ρ_X on $\mathcal{X}$ from data sampled from ρ_G on $\mathcal{X}$. The generator's aim is to make it as hard as possible for the discriminator to distinguish between the sampled data. We can capture this with our loss function; there are a few ways to do this. If we are given input data $x_1, \ldots, x_n$ and indicators $y_1, \ldots, y_n$, where $y_i = 1$ if x_i is sampled from ρ_X and $y_i = 0$ if x_i is sampled from ρ_G. We can then define the loss function by

$$-\frac{1}{n}\sum_{i=1}^{n} y_i \log(D(x_i)) + (1 - y_i)\log(1 - D(x_i)). \tag{1.1}$$

In doing this, we have reframed the problem of training our network as an optimisation problem so that we can use optimisation algorithms to train it. We will train both the networks, however we only change the parameters of one network at a time. If we change both networks simultaneously we are prone to causing the loss to increase as a result of us trying to hit a moving target [7].

As a concept training GANs seems relatively simple, however in practice it can be tricky to implement well. There are quite a few things that can go wrong, for example if the discriminator learns much faster than the generator then the discriminator may be able to tell between the samples it is fed with near-perfect accuracy. This leads to the well-known vanishing gradient problem in machine learning. The derivatives of the loss function with respect to the parameters of the generator will all be close to zero. In effect, the discriminator will not provide enough information for the generator to make progress [7].

A common method people use to get around the challenges of training a GAN from scratch is using something called transfer learning. This is the process of using a

pre-trained network as the start point of a new task. We can then train this network again such that it works for our purposes. Usually, this reduces the amount of data we need to train the network as the network should have less to learn. For example, if we were using a network to identify animals in images then repurposing it to identify objects in kitchens it should make for less training than if we trained a network from scratch. This is because the early layers will have already learnt how to extract features, such as edges. These layers will therefore also be useful in identifying objects and do not have to be learnt from scratch. Transfer learning can also be useful when training GANs for TTS.

1.3.2 Using GANs for Text-to-Speech (TTS)

To use a GAN for TTS we have to be able to condition on the text that we want to be "said" in the synthesised speech. Thus, to train a GAN for TTS we need audio recordings of speech and accompanying transcript for each recording. Audio files and their corresponding transcripts will be the inputs to the discriminator such that the discriminator judges not only if the speech sounds realistic, but also if it sounds like the correct thing is being said. In a similar way, we can train each GAN generator to synthesise multiple different voices if we have a training set containing multiple speakers. We simply condition on the speakers using our generator and also tell the discriminator which speaker we are conditioning on during training.

GANs which take text as an input and return raw audio waveforms exist, however it is much more common to use a text-to-Mel model and a vocoder. Using these two networks tends to allow us to produce more realistic speech. The text-to-Mel model is not normally a GAN generator, rather it is a sequence-to-sequence model. However, the vocoder, which converts the Mel spectrogram to raw audio waveforms, is normally a GAN generator.

The last of the background information we need to familiarise ourselves with is the theory and results from the paper which introduced the network dissection method [3].

1.4 Controlling the GAN's Output's Properties via Network Dissection

1.4.1 Introduction to Network Dissection

Deep learning allows us to create models to solve all kinds of complex problems. Yet, in many cases, we do not understand how these models work. Bau et al. [3] shed some light on this by examining how individual units of a neural network can coincide with human-interpretable concepts that were not explicitly taught to the network. This differs from most other attempts to understand deep neural networks [16, 18, 22] in that we are not creating an auxiliary model, we are trying to understand the internal computations of our own model.

As the learnt concepts are not explicitly taught to the network, a big question is what concepts can we identify? Bau et al.'s work is with the classification and generation of images and the units they identify are specific to objects, parts of objects, materials and colours, although more concepts may be discoverable. But what does it mean for a unit to be specific to one of these concepts and how do we identify that one is? The main contribution of the paper [3] is the analytic framework for identifying and quantifying the role of property-specific units in a neural network: network dissection. We will go into much more detail about this later, but the main idea is to collect statistics on the activations of units of the network over many different runs and use these statistics to identify which units activating strongly coincide with which semantic concepts being in the input or output data. For a GAN generator, once we know which units are specific to a certain concept we find that images generated with certain concepts, such as large windows, are likely to have certain units activating highly. Not only this but Bau et al. then show that property-specific units can actually play a causal role. They do this by artificially activating and removing some of the units which are most strongly tied to a semantic concept and examining the effect on the image by observing the change in the number of pixels corresponding to that concept. Not only can concepts be added to images, but they are generally adapted to the context of the image too. For example, the style of a door added to a building will tend to match the style of the building.

As we mentioned in Subsection 1.2.1, convolutional neural networks are often hierarchical in that lower level features may be identified in earlier layers with later layers combining these to identify more complex features [35]. This finding was backed up for the classifier in [3] by most object-specific units being found in the final layer

and most object part-specific units being found a layer or two earlier. More so, GANs learn in an unsupervised setting: the training data we use is unlabelled. It is therefore interesting to see that they learn the objects in the image that make it up. For the generative model the largest number of emergent concept units do not appear at the edge of the network as we saw in the classifier, but in the middle. We will investigate how this distribution of concepts compares to the distribution for audio generation.

Further points of interest will be which concepts we can identify in the audio domain and whether editing the activations will allow us to change the speech properties. More so, if we edit the activations relating to a property will we still have the adaptability of the edited property that we had for images? That is, will the edited property fit with the context of the original speaker and will only the specified property noticeably change? Lastly, we want to know if we can produce realistic speech while editing the activations.

For image data, the technique allows us to identify units specific to semantic concepts, such as chairs, which tend to look similar to each other but are rarely the same. We therefore hope this lends itself to identifying properties in speech since, in a similar way, speech properties manifest themselves in different ways for each person. Nonetheless, it will be an interesting problem to investigate.

Being able to identify and edit the activations of units specific to accent to get a change in accent would be a huge step forward for TTS. The benefit of this method over, say, training networks with lots of speakers and conditioning on them is that this allows us to only change one property of speech while keeping all other properties the same. This would massively reduce the amount of training data we would need. We would not need training data for speakers with every different combination of properties, since as long as we had a speaker with some of the properties we could just edit the others.

To the best of the author's knowledge, we will be the first people to apply Bau et al.'s method to generating audio. Before we can apply this method we need to thoroughly understand it; this is what we will do next.

1.4.2 How to Dissect a Network

Bau et al.'s paper [3] examines both classifiers and generators. Since we are focusing on audio generation in our research we will only discuss their findings on generators from here on out. They use a progressive GAN architecture [15] in their paper to mimic the distribution of images in a training set.

The crux of network dissection is the intersection over union ratio (IoU). This is the model they use to quantify the agreement between semantic concepts and units. The IoU is a function of a unit and a concept. To define it for the image generation GAN we need to be able to identify visual concepts in the generated images. This is done via a computer vision segmentation model $s_c : (x, p) \to \{0, 1\}$ which is trained to predict the presence of the visual concept c within an image x at position p. This can identify the object classes, parts of objects, materials, and colours that it has been trained to [30]. For each unit u we can compute the activation $a_u(x, p)$ for every generated image x and at each image position p. We can analyse the low resolution convolutional layers at high-resolution positions p by using bilinear upsampling. We denote the top 1% quantile level of a_u by t_u. That is, writing $\mathbb{P}_{x,p}[\cdot]$ to indicate the probability that an event is true when sampled over all positions and generated images, we define the threshold $t_u \equiv \max_t \mathbb{P}_{x,p}[a_u(x, p) > t] \geq 0.01$. Now, we can quantify the agreement between a concept c and a unit u using the IoU, where

$$\text{IoU}_{u,c} = \frac{\mathbb{P}_{x,p}[s_c(x, p) \wedge (a_u(x, p) > t_u)]}{\mathbb{P}_{x,p}[s_c(x, p) \vee (a_u(x, p) > t_u)]}. \tag{1.2}$$

This IoU is computed on a set of generated images. Each unit is scored against the 1,825 segmented concepts identifiable by the segmentation model, then each unit is labelled with the highest-scoring matching concept.

Bau et al. go on to test their algorithm using a progressive GAN [15] containing 15 convolutional layers (as shown in Figure 1.3a) trained on the LSUN kitchen dataset [34]. Their GAN takes a 512 dimensional vector as an input, which they sample from a multivariate Gaussian distribution. The network then produces a 256×256 image. Frequently, they visualise regions of images where a unit u activates highly $\{p \mid a_u(x, p) > t_u\}$, as shown in Figure 1.3b.

First, they apply this network dissection method to every unit in every layer to get an idea of the distribution of high-IoU units in the network. Figure 1.3c shows that the largest number of emergent concepts appear in the middle of the network, in `layer5`. Most of the concept classes then die off towards the later layers, except for the classes that represent colours.

Figure 1.3d shows a bar chart with each of the concept classes and the number of units they match in `layer5` with IoU $> 4\%$. We see that even within one layer concept classes can be matched with multiple units. Not only this, but the concept a unit is matching with can vary quite a bit from image to image. We see this in Figure 1.3e where the 'chair' and 'oven'-specific units activate highly on a range of different styles of these. This shows that the units do not simply match a rigid pattern of

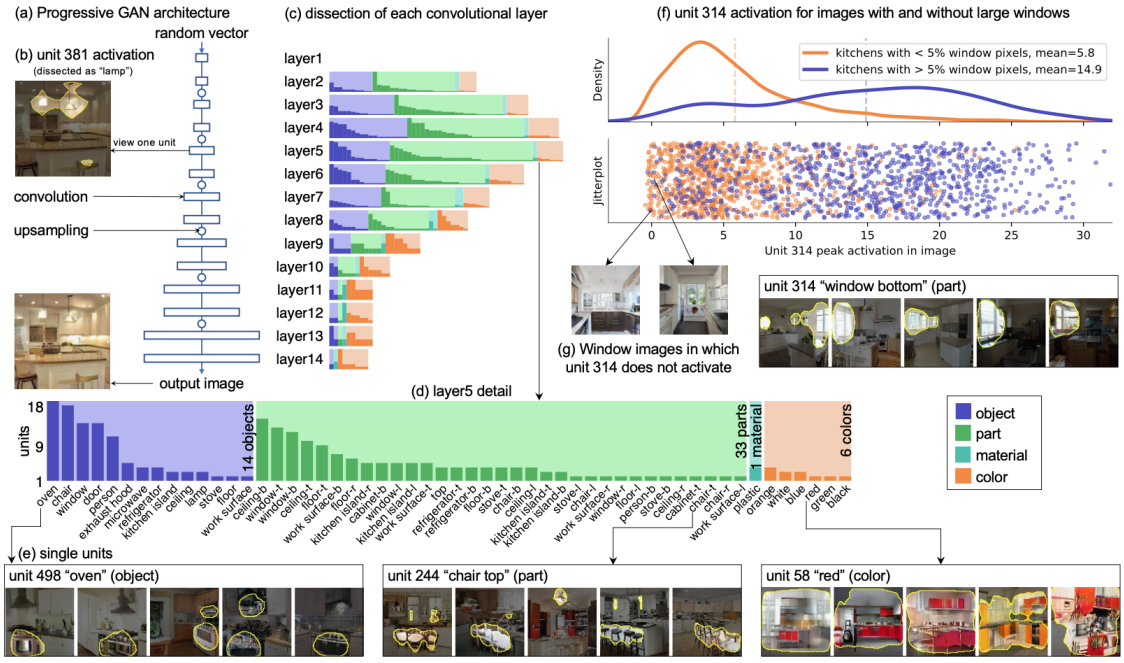


Figure 1.3: The emergence of concepts within a progressive GAN generator [3].

pixels.

As a way to show the IoU model works in finding concept-specific units, in Figure 1.3f they use the window-specific unit 314 as an image classifier. By setting a simple threshold on the peak activation of the unit, they find that they can achieve an accuracy of 78.2% in predicting whether or not a generated image will have a large window. Although, Figure 1.3f shows that we can actually generate images of kitchens with large windows without unit 314 activating highly. Two of the images are then shown in Figure 1.3g.

Thus, we have seen that there is a correlation between the activation of certain units and generated concepts, but we cannot say whether this relationship is causal. That is, we cannot say whether having these units activating highly actually causes the concept to be rendered in the output image. To test this Bau et al. remove and artificially activate units of the GAN generator and examine the difference in the generated images.

Bau et al. only investigate the activations of convolutional layers. As we discussed in Subsection 1.2.1, the activations of convolutional layers are the outputs of the activation function applied componentwise to the output of the convolutional layer. Therefore, when they say increasing the activation of these units they mean increasing the output of the activation function on these units, so that the edited outputs are

passed through the rest of the network. When we say removing units we mean setting their output to zero.

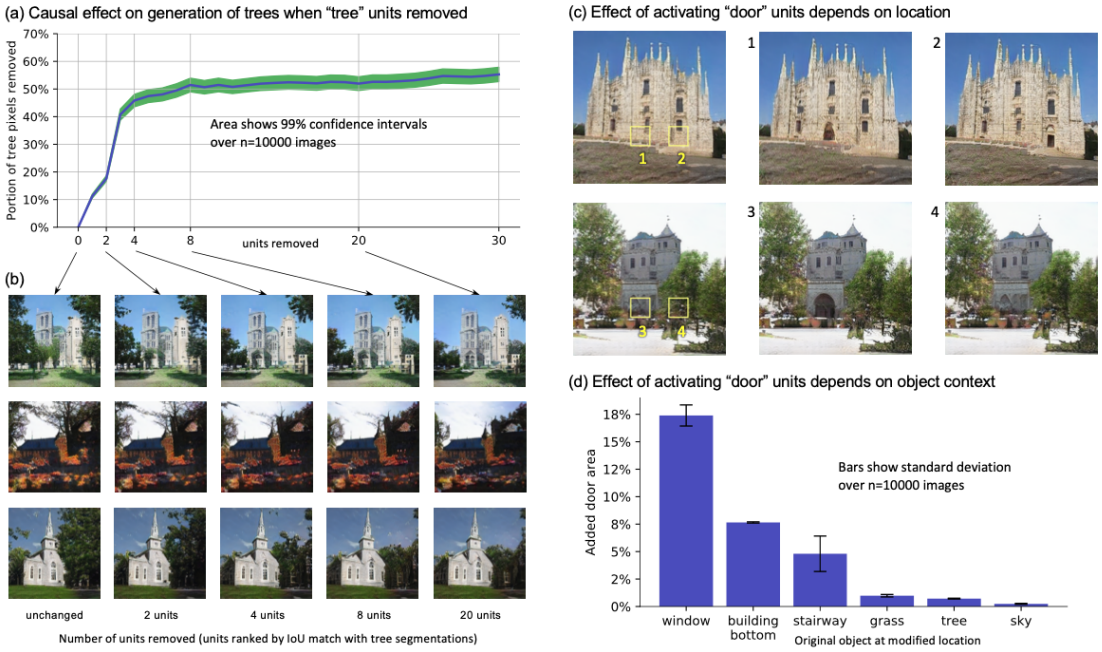


Figure 1.4: This shows the causal effect of altering units activations in a GAN [3].

The nature of these next experiments is as follows: they find the IoU for all the units in `layer4` corresponding to the concept c, they generate an image using an input y, they then generate more images using the same input y but editing the activations and, finally, they compare the differences in the generated images. First, they do this for a progressive GAN [15] trained on LSUN church scenes [34] with the concept c being 'trees'. They remove the 20 most tree-specific units, as chose by the IoU, and record the change in the number of tree pixels in the generated output using the segmentation model [30]. Over $10,000$ randomly generated images they find that removing these units causes a 53.3% reduction in the number of tree pixels in the edited outputs. To see the trend more clearly they repeat this experiment varying the number of units that they remove and plot the proportion of tree pixels removed against the number of units they remove. This is shown in Figure 1.4a; it clearly indicates a causal relationship. Not only does the number of tree pixels decrease, but in place of them are pixels which make the image look equally realistic. For example, in Figure 1.4b, we see how parts of the building that were previously occluded by trees are hallucinated in the same style. This strongly suggests the GAN is learning a structured statistical model of the scene rather than simply a summarisation of

visible pixel patterns. How does this compare to when they activate units?

Following the same experimental procedure as before, only this time with the concept c being 'doors', Bau et al. activate the 20 most door-specific units at different locations in an image and examine the difference. The units' activations are set to their 1% quantile level t_u. Figure 1.4c shows the effect of doing this for two locations in two generated images. Despite the same procedure having been applied to edit these networks with the position being the only difference, we see that the model adapts the door to fit the context of the image in terms of style, size and location. More so, we see in Image 4 that the model has not added a door where it simply would not make sense to, over a tree.

They then show how this seeming "intelligence" of the model manifests itself over many more images. They do this by generating 10,000 images and activating the same 20 door-specific units at every feature map location then recording the number of newly synthesised door pixels using a segmentation algorithm. Figure 1.4d shows how doors can be more easily added to the bottom of a building or in place of a window than, say, the sky. As such, we can conclude activating units suggests the GAN is learning a statistical model of the scene in a similar way to how removing units did.

Bau et al. released quite a bit of code to showcase the network dissection method for images; we want to develop this such that it works for TTS too. Now that we understand all the background material, it is time to start working on our problem. The first thing we will need to do is find a working TTS model.

Chapter 2

Overview of Mathematical Models for Speech

2.1 Exploring TTS Systems

In this project, our aim is to apply the techniques of Bau et al. to audio generation and get analogous results. As such, we came up with a basic test to check how well our method performs. More so, with this being a large project to undertake this test gives us an objective we can work towards. The test requires us to have a GAN generator which we can condition on two or more speakers, each with speech containing two or more properties, such as accent or gender. We then want to see if we can change one property of a speaker's speech to be the same as another speaker's while keeping all the other properties constant. For example, if we had data for an American male and an English female we could try to make the American male sound like an English male.

To go about this, we will first look for a pre-trained TTS GAN. Alexa's voice is somewhat of a brand to Amazon, therefore we are not allowed to access their models. As we mentioned in Subsection 1.3.1, training GANs from scratch is a fragile process and therefore time-consuming. Finding a pre-trained model will allow us to skip the cumbersome task of finding data sets and fiddling with hyperparameters until the model is usable. Furthermore, if our pre-trained model does not precisely satisfy the requirements we have, we can employ transfer learning at much less of a computational cost than training from scratch, as we described in Subsection 1.3.1.

Initially, it seems appealing to find any high-quality TTS GAN, find audio data and transcripts for two speakers that have two properties different from one another

and use transfer learning to train the network to condition on these two new speakers. The problem with this is, if we only have two speakers and we want to find units specific to a property that only one of the speakers has, then in trying to find units specific to that property we will just be finding units specific to that speaker's speech. This is because, of the speakers we have, only that speaker's speech exhibits that property and it always exhibits that property, so in learning units specific to that property we will just be learning units specific to that speaker's speech. Editing the units would likely then cause the new speech to sound like that specific speaker rather than only changing the selected property while the others are held constant.

As such, to find units specific to a property it is beneficial to be able to condition on lots of different speakers whose speech has that property, but with their other properties of speech different from one another. This allows us to disentangle properties. For example, if we were trying to find units specific to an Australian accent and all the training data we have for Australian speakers is from males then we would naturally be finding units specific to Australian males. If editing the activations then gave us an Australian accent it would likely also give us a male voice. However, if the other properties of the Australian speakers' speech are all different from one another then we should have a better chance of disentangling the Australian accent from the other properties. As such, we want a trained GAN which is able to condition on lots of speakers with some of these speakers sharing properties of speech and some not. The other criterion we have for our GAN is that the language of the speech it generates needs to be English, so that we can check if it is generating speech corresponding to the text and if the changes in accent work well.

Finding pre-trained GANs for TTS can be difficult as lots of papers release the method they use to train the GAN but don't actually release the GAN they trained. Thankfully, there exists a package called ESPnet [11, 26] which contains and facilitates the use of pre-trained vocoder GANs with pre-trained text-to-Mel models. The vocoders are implemented using PyTorch as is the code Bau et al. released to aide with the implementation of network dissection. This will therefore be helpful when we try to integrate these later. The package also supports the end-to-end training of TTS systems, which is ideal if we need to train on more speakers. All the network architectures have versions trained on the multi-speaker VCTK and LibriTTS data sets [31, 36]. The package gives a choice of several text-to-Mel models and vocoder GANs. These are all unofficial implementations in that they have not been trained by the authors of the original papers on these networks, but by the contributors to ESPnet. Details of the training and implementation can be found in [10, 11].

Also, note here that we only condition the text-to-Mel model on the speaker, the GAN does not change. This is not ideal as we may miss the emergence of some concepts if they occur in the text-to-Mel model. Nonetheless, we discussed this with our supervisors and decided it would be an interesting research direction in any case.

There is a notebook in the ESPnet GitHub repository explaining how to use the TTS system [9]. If we are to integrate the GANs from this package with Bau et al.'s code it is important that we understand both bits of code. Distilling it down into its main components: we instantiate an object of the `Text2Speech` class, we randomly select a speaker vector to condition on, then we use the object like a function to synthesise the speech.

It seems counter-intuitive that something can be both a class and a function. The `Text2Speech` class is imported from a file in ESPnet. Looking at the file this class is imported from, we see within the class the built-in Python method `__call__` is defined. This means that once we have instantiated an object of the class we can call this method simply by using the name of our object and the required inputs to `__call__`. The one necessary input to this method is a string, the rest are optional ones mainly relating to choosing the speaker we want to condition on. The method then synthesises speech saying what was written in the input string in the voice of the speaker you are conditioning on.

The code for the `__call__` method is large and complex, however we can identify where and how each of the models is downloaded and used. This is not necessary right now but it gives us confidence we will be able to integrate it with Bau et al.'s code later.

Another thing we need to be confident in if we are to use the ESPnet package is that we can synthesise high-quality speech. This will be essential if network dissection is to identify properties of the speech and if we are to discern any differences in generated speech. As such, we investigate the different combinations of text-to-Mel and vocoder models to ensure we can synthesise speech that is clear enough. We believe there are a few usable combinations, although the speech is not always great for any combination. We believe, the combination which provides the clearest speech most consistently is the Conformer-FastSpeech 2 [8, 11, 21] text-to-Mel model with the HiFi-GAN [17] vocoder.

Now that we have models that we can use to synthesise speech, we need to identify properties of the speakers' speech that we might be able to reproduce in other speakers.

2.2 Identifying Properties of Speech

There are a number of properties that affect speech. Some of the most obvious ones are gender, accent, age and emotion. We need to be able to identify properties within the speech we generate such that we can apply network dissection to locate the relevant units. That is, we want something similar to the segmentation model Bau et al. use so we can define a function analogous to their IoU. The difference is, they had a concept of position in their generate data, but we do not in ours since when we condition on a speaker they will be speaking with the same properties throughout the generated audio. More so, the speakers the networks are currently trained on all speak in a monotone voice, devoid of any emotion. As such, the properties of any generated speech are based solely on the speaker's identity, therefore we may sometimes refer to a speaker's properties rather than a speaker's speech properties as it is less cumbersome.

We want to find something similar to the segmentation model for images, that is, some kind of algorithm to identify properties of speech. Searching for papers related to their segmentation model, we can only find algorithms which identify emotion and accent. Unfortunately, all the speech we can currently generate is monotone and, for networks trained on the VCTK data set, the speakers' accents are already labelled. This prompts us to check what properties the LibriTTS and VCTK data set have labelled. The LibriTTS data set only has the speakers' genders labelled, whereas the VCTK data set has age, gender and accent labelled for each speaker. Furthermore, the VCTK data set has 108 speakers, some of which share properties, but there is not too much sharing so this should be useful for disentangling them. This means that it is unlikely we will have to train the network with any more speakers, as such we will use the networks trained on the VCTK data set. Since we now have trained networks that we can use for TTS, we will move on to developing the code to dissect these networks.

Chapter 3

Developing the Network Dissection Code

3.1 Getting started

As we mentioned in Section 1.4.2, Bau et al. released code with this paper to aide with the implementation of their network dissection method [2]. Looking through Bau's GitHub profile, we see that he has released a number of repositories with this aim in mind [2]. Consequently, we collate these repositories so we do not miss any important information or notebooks that could simplify our task. As far as we can tell, he has not released any documentation explaining how the functions work, only tutorial notebooks. We will therefore learn from these and from examining the code and comments he has written in the GitHub repositories. It is important to note that these repositories contain thousands of lines of code and hundreds of interdependent functions and, as far as the author is aware, they have only been applied to image classification and generation tasks. Therefore, there is a lot of complex work to do to make the code applicable to audio generation.

When developing code, our process is to understand their code, decide how we will adapt it to work for our purposes, implement our ideas and, lastly, test the code. As such, this is how we will display the process in this dissertation, except we normally will not show much of the implementation as it is quite space-consuming and can be found in our GitHub repository [4]. The first step to understanding the network dissection code is to run the notebooks we have collected.

When attempting to run any of the notebooks on my computer or any of my office's desktop computers lots of different errors occur, mainly a result of not being

able to find files and functions. Yet, when we run these same notebooks on Google Colaboratory (Colab) they run much further, although still not to the end. Speaking to my industry supervisor, we agreed it looked to be the result of problems with the dependencies. He said that about a third of his software engineers' time is spent worrying about dependencies, so we would be better off using Colab to save time.

Comments within the notebook 'gandissect_solutions.ipynb' [2] describe how to implement the network dissection method. After inspecting the notebook, we distill the code to dissect a network down into three main parts: reading the activations, editing the activations and calculating activation statistics. For this notebook, we can run the parts of the code which read the activations and which calculate activation statistics, but we eventually get an error when we try to import a file relating to the segmentation algorithm. Fortunately for us, we do not need the segmentation algorithm as we have a different way to identify properties in generated data, as described in the Section 2.2. The problem is, the part of the code in which they edit activations is after this error. The notebook contains a way around using the segmentation algorithm for if you do not have access to a GPU. Following this, we still have one error, however we realise it is a result of some variable we had to comment out not being defined. Thus, by defining the variable and running the code, we can now edit activations. Therefore, we have all the parts of the code we need working for image generation and we can move on to developing this code to work for our TTS system.

3.2 Reading the Activations

As we said in the last Section, to edit their code we will first need to understand it. After, importing packages and downloading their GAN generator they "wrap" their GAN generator with an `InstrumentedModel`. This gives them the functionality to be able to "hook" layers. If they have a wrapped GAN generator called `model`, hooking the layer `layername` using `model.retain_layer('layername')` allows them to store the outputs of that layer after a computation has taken place. To return a tensor containing the outputs of that layer from the last computation they can then use `model.retained_layer('layername')`. It is important to remember that in PyTorch we apply convolutional layers then apply activations as separate layers, so we will be hooking the activation layers.

To apply this to our TTS code, we need to wrap the vocoder and put the hooks in after we configure the vocoder but before we apply the vocoder to our Mel spec-

trogram. The vocoder is configured when instantiating an object of the `Text2Speech` class and the vocoder is applied in the method `__call__`. This means that we do not have to edit the code in the ESPnet package: we can just wrap our model and add hooks in between instantiating the object and calling the method. Finally, we can return our activations using `model.retained_layer` after we have synthesised some speech.

We need to find where the GAN is saved in our `Text2Speech` class in order to wrap it. Searching through the method `__init__` of the `Text2Speech` class, we see it is saved as `self.vocoder`. Now we just need to find the names of the layers of the vocoder such that we hook them. Assuming we now have an object of the `Text2Speech` class called `model`, we can print `model.vocoder` to see all the layer names and what types of layer our vocoder GAN contains. We can then take the layer names we want and write them into our functions.

Running this code, we find that we return large tensors when we print `model.retained_layer`, however we need to test if this corresponds to the activations. This is hard to test as we do not know what the activations are meant to be for most layers. One thing we can do is print the activations of the final layer of the model as this should be identical to the output of the model. Comparing these we find they are identical for all the different text and speaker combinations we run, hence this indicates that we are reading the activations correctly. We will test this function more when we can edit our activations.

3.3 Editing the Activations

The next task we want to accomplish is to be able to edit our vocoders' activations. Let us start by understanding Bau et al.'s code. In the same way as last time, we have to wrap the GAN generator and hook the layers we want to apply our function to. Suppose their GAN generator is called `model`, then the method they use to edit their layers is called using `model.edit_layer`, with the inputs being a layer name and the `rule` function. Now, if they input something to the network, the output of the specified layer is fed into the `rule` function which directly edits this data before it is fed back into the next layer of the network.

In the 'gandissect_solutions.ipynb' notebook, the function which is input as `rule` has two inputs: `data` and `model`. This function defines how to act on the tensor `data`, here they select entries of it to change to 0 using the code `data[:, tree_units, :, :] = 0.0`, where `tree_units = [365, 157, 119, 374, 336, 195, 278, 76,`

`408, 125]`. The notebook then shows images generated using the same inputs with and without editing side-by-side, so we can see the decrease in the number of tree pixels. Finally, they use `model.remove_edits()` to remove any changes they have made to the units: next time they run the model the edits will not be applied.

To implement this with our TTS system, we can use the same code as before to wrap and hook the network. Initially, we attempt to use the same `rule` function as Bau et al.'s notebook. We add this and the `edit_layer` method to our TTS code just before we use the `__call__` method on our `Text2Speech` class object. Initially, we get an error saying that we have too many indices for a three dimensional tensor. As such, we write code to read the layer outputs of the layer that we are editing and print the shape of the tensor we return. We then change the code to only edit entries which lie in this three dimensional tensor; this lets our code run. To test that we can only select indices within this three dimensional tensor, we try selecting many different indices, particularly at the borders. This shows that we have to select indices from within this tensor.

Since we can also read activations, a good test would be to edit activations then read them back to make sure both of these functions are working correctly. Doing this for a number of different layers and making a number of different edits to the units, we find that we always read the edited units correctly.

Although we can see that the activations change, we want to do one more test to ensure that the editing function is leading to audible changes in the audio we produce. To test this, we produce some speech, edit all the outputs of one layer of the GAN and then produce more speech to see if it differs from the original. It does, therefore we are happy that our editing function is working correctly.

3.4 Vocoder Architecture Difficulties

While editing the activations, we realise that sometimes when we run our code we get an error and other times, when we run exactly the same code, we do not. The error always relates to an index of an output we want to edit being out of bounds. Given that this happens sometimes but not others we hypothesise that the shape of the outputs of our GANs' layers are changing and this is due to randomness in our text-to-Mel models or GANs (or both). We will refer to the shape or width of the output of a layer as the layer's shape or width, respectively.

This is a problem for us: we need the layers' widths, weights and biases of our chosen GAN to stay the same each time we run the code or we cannot use network

dissection. If the weights and biases change between each run then we are dealing with several different networks and each network's units interact with each other differently. It therefore does not make sense to run TTS and find property-specific units as the units will be different for each network. Similarly, if the layers' widths change then on some runs of our GAN certain layer outputs will exist and on other runs they will not, so we would be working with several different networks rather than one. Again, we would have the same problem.

To test if it is the case that the layers' widths are changing, we run our TTS function multiple times with the same inputs, each time printing the layers' widths then we compare them. Although it does not always happen, the layers' widths and even final output size sometimes vary from run-to-run. These tests were done with the Tacotron2 [23] text-to-Mel model and Parallel WaveGAN vocoder [32] both trained on the VCTK data set. We want to see if this still happens when we use different combinations of models in order to identify which models cause the randomness. As we know, we are only using networks trained on the VCTK data set. Thus, by testing every relevant combination by running the TTS function multiple times and comparing the GANs' layers' widths, we find that the layers' widths do not change for any GAN if we use the Conformer-Fastspeech 2 [8, 11, 21] text-to-Mel model and condition on the same text and speaker for every run. However, changing either the speaker or the text normally causes a change in layers' widths for all the layers of every GAN we have. Similarly, using either of the other two text-to-Mel models will normally cause the GANs' widths to change each time we run the code, even if we are conditioning on the same text and speaker.

Since, we see that the layers' widths do not change for some network combinations when conditioned on the same text and speaker, we want to see if the activations change for these combinations. As such, we repeat the experiment we last did but rather than comparing layer sizes we compare layer outputs. We do this by converting the three dimensional layer output tensors to one dimensional tensors using `torch.flatten`, converting these to NumPy arrays, subtracting one NumPy array from the other and using `numpy.any` to check if any of the entries are non-zero. We find that if we combine Conformer-FastSpeech 2 with either HiFi-GAN [17] or Multi-band MelGAN [33], then the activations stay the same for each run. However, if we combine it with Parallel WaveGAN or StyleMelGAN [19, 32] despite the layers being the same width, the activations change with each run. The evidence so far suggests two important things to us: Conformer-FastSpeech 2, HiFi-GAN and Multi-band MelGAN are deterministic whereas the other models exhibit randomness, and

the change in the layers' widths is being caused by the change in the length of the input Mel spectrogram. Ultimately, we need to ensure that we have a vocoder GAN that acts on the input Mel spectrogram in the same way each time we apply it, or we cannot apply the network dissection method.

To test if Conformer-FastSpeech 2 always produces the same output if we input the same text and speaker ID, we decided to investigate the input to the vocoder. The vocoder is applied in the method `__call__` using `wav = self.vocoder(input_feat)` where either `input_feat = output_dict["feat_gen_denorm"]` or `input_feat = output_dict["feat_gen"]`. The method ultimately returns the dictionary `output_dict`. Hence, if we save the output of the method then we can access `input_feat` by accessing the values stored with the keys `feat_gen` and `feat_gen_denorm`. Although, we will not know which value `input_feat` is being set to, we can run the `__call__` method twice with identical inputs and compare if either of the values changes from one run to the next. We can check this using the same method as the previous paragraph.

Doing this for Conformer-FastSpeech 2 we find that all of the values are $r \times 80$ shaped tensors, where $r \in \mathbb{N}$, and none of the values change from one run to the next. Although, the values are different for each key. We find this to be true for all different sentences and speakers we use, thus we conclude that Conformer-FastSpeech 2 always acts on the same inputs the same way. If we repeat this experiment for either of the other two text-to-Mel models, we find that often the size of the generated tensors are different from one run to the next, and if not then their entries are different anyway. The lengths of `feat_gen` and `feat_gen_denorm` are always the same. There seems to be a correlation between these lengths and the widths of the layers of our GAN generator. To investigate this more thoroughly we will need to be able to change the length of `input_feat`.

The first thing we want to check is that if we give the same shape input to each vocoder architecture we will always get the same shape layers within each vocoder architecture. To do this we are going to edit the `__call__` method such that it always extends or cuts `input_feat` to the same length. For each GAN architecture, we are then going to run the `__call__` method multiple times changing the text and speaker we are conditioning on and checking that the shape of the layers does not change. A simple way to edit the `__call__` method is to install the ESPnet package but not import the `Text2Speech` class straight away. Instead, find the code for this method in the installed packages and edit it before we import it.

We extend `input_feat` by concatenating rows of zeros and cut it by removing

the bottom rows such that the Mel spectrogram input to the vocoder is always the same length. This can be implemented using an if statement; the way we did this can be seen in the notebooks in our GitHub repository [4]. If the length of the original `input_feat` is cut then we find that the speech will sound like normal, but be cut off part way through. On the other hand, if the length of the original `input_feat` is extended we have a loud, fuzzy noise after the speech.

Running these experiments for all the different combinations of networks, we find that within each GAN architecture the layer size does not change. This is still the case when we change the length of `input_feat` and repeat these experiments. In fact, the width of each layer of our GANs is always just a constant multiplied by the length of `input_feat`. Hence, we are now sure that provided we keep the input Mel spectrogram's length the same, the vocoders' layer sizes should stay constant.

Now, we just need to ensure that we have a GAN that acts on every input of the same length in the same way so that applying network dissection to it makes sense. By inputting the same Mel spectrogram to the HiFi-GAN vocoder multiple times and comparing the outputs, we see that the vocoder always gives the same output. Reviewing the architecture of HiFi-GAN in the paper [17], we can confirm that this is the case. Given that Conformer-FastSpeech 2 and HiFi-GAN is the network combination that provided us the best quality speech and HiFi-GAN is a deterministic vocoder we do not need to investigate whether any other vocoders are deterministic. We will use this combination from here onward.

We know we will have to either cut or extend `input_feat` such that its length stays constant whenever we run network dissection experiments. Rather than having to edit the `__call__` method every time we open the code in Colab, we clone the ESPnet package's GitHub repository [26] and rename it 'billiespnet' [4]. The class `Text2Speech` is located in 'billiespnet/espnet2/bin/tts_inference.py'. We edit `__call__` method in this file of the repository and install billiespnet instead of installing ESPnet in our notebooks. Now when we run our notebooks the TTS function will automatically cut `input_feat` to the length we set in our GitHub repository.

Thus, we have fixed the problems with our vocoders and we can go back to integrating network dissection code with our TTS model.

3.5 Collecting Activation Statistics

We want to get two of the functions that calculate statistics on the activations of networks working for our GANs: the `tally_quantile` and `tally_topk` functions.

The first of these runs a network on some given inputs and calculates the quantile statistics for specified units. This function will be used when we want to increase the output of a unit to its 1% quantile level. The second function runs a network for some given inputs and returns two tensors both of which are of shape (number of chosen units)×(number of runs of the network). The first tensor contains the outputs of each of our chosen units for every run of our network ranking them from largest output to smallest output for each unit. The second tensor contains the indices corresponding to which run of the network caused each of these activations. This function will be used in calculating the IoU by allowing us to find which runs are causing which units to activate in their top q% quantile, for some $q \in \mathbb{R}$. To clarify, when we say the top q% quantile level of a unit's activations, we mean the largest activation such that the top q% of our activation data is greater than or equal to this value.

Understanding how to use these functions will take some time as, like we said earlier, there is no documentation released with the code to describe how it works. We will start by attempting to understand how the `tally_quantile` function works; first, we look at the 'gandissect_solutions.ipynb' notebook. In this notebook, the function takes two inputs: a function `compute` and a TensorDataset. The function `compute` takes `zbatch` as an input which we assume is a batch of data. It applies the GAN to it once, changes the shape of the tensor containing the stored activations and then returns the tensor. The TensorDataset is defined earlier in the code and contains 50 inputs for the image generation GAN being used in this notebook.

To get a better idea of what is going on in the `tally_quantile` function, we want to run it with HiFi-GAN from our TTS code, rather than their image generation network. To do this, we add their `compute` function to our TTS code, replacing the model they run with the `__call__` method we use to generate speech and replacing the name of the layer they were retaining with a layer name from our HiFi-GAN. Adding the `tally_quantile` function, we realise that we need to redefine their TensorDataset input. Their TensorDataset input contained 50 inputs for their model, so we need to change this to contain the input to our model that we want to vary. For now, this will only be the speaker we are conditioning on. There are 108 speakers we can condition on, hence we use `zds = TensorDataset(torch.linspace(0, 107, 108))` to create the data set of speaker indices. We then add the lists we use to get the speakers' IDs from this index to the `compute` function. Lastly, we set the text to be inputted to the TTS system, before running the code.

We get an error as a result of trying to input multiple speaker IDs into our TTS function at once when it only accepts one-at-a-time. Adding print functions to the

original code, we hypothesise that `tally_quantile` works by running the `compute` function on batches of 10 inputs, with the entirety of each batch being input to the model at once. Then, once the network has run, the `compute` function flattens the retained activations into a 10×(number of chosen units) tensor and returns it. The `tally_quantile` function repeats this process for all the inputs in our data set in batches of 10 while keeping track of the quantiles for each column of the tensors returned from `compute`. In this case, this is each of the units in the retained layer. The question now is, how can we adapt our TTS code so it will work with the `tally_quantile` function?

The problem is, we cannot input multiple speaker IDs into our TTS function at once. If the `tally_quantile` function works like we think it does and the `compute` function is not used elsewhere, we think we may have a solution to the problem. Within our `compute` function, we could add a for-loop which loops over the entries of `zbatch`, runs TTS for each one and then flattens and stacks the retained activations from each run such that they are of the correct form to be returned by the `compute` function. Before we implement this, we inspect the code for `tally_quantile` in 'global-model-repr/netdissect/tally.py' in the GitHub repository [1] to check the `compute` function is only used where we think it is. Although the code is quite difficult to understand due to it using functions and classes defined elsewhere in the repository which then also use functions from elsewhere in the repository, we do not believe it uses the `compute` function in another way. Thus, we decide to implement our idea.

We decide to first implement the idea using Bau et al.'s 'gandissect_solutions.ipynb' notebook with his image generation network. Then, we can check if we get the same quantiles that they were getting before we edited the code. This would be a good indication that our idea works correctly and is transferable to the TTS version.

Implementing this, we see it is slightly more elaborate than the code we are replacing. Nonetheless, once we get it working, we find all the quantiles we test are identical for both sets of code, thus we will now move on to implementing this idea with our TTS code. To test the quantiles are calculated correctly for our vocoder GAN, we add print statements to our `compute` function which print the activations of the units we are calculating the quantiles for. Then, we test our code using only 20 runs of our model while also calculating the quantiles for a few units by hand from the printed activations. Running this test, we find the quantiles calculated by `tally_quantile` and by hand are identical, therefore we will assume our code to calculate the quantiles is working correctly. As usual, it can be found in our GitHub

[4].

We now want to integrate the `tally_topk` function with our TTS code. Again, this function takes two inputs: a function `compute` and a TensorDataset. Initially, looking at the 'gandissect_solutions.ipynb' notebook, we thought this may involve editing our `compute` function, as they change theirs from when they were using `tally_quantile`. However, running `tally_topk` with our TTS GAN and the same inputs as `tally_quantile`, examining the indices of the speakers that cause high activations and the corresponding activations then running TTS with those same speakers and reading back their activations, we see the activations agree when using Conformer-FastSpeech 2 and HiFi-GAN. Since `tally_topk` puts the activations in the correct order, this implies that `tally_topk` is working as we would want it to. Testing this further by picking random units and speaker IDs and finding their activations from `tally_topk`, then checking this by running TTS for that speaker assures us the function is working correctly.

Now that we have integrated all the functions we need from Bau et al.'s code with our TTS code, we only need to implement one more function before we can try to find property-specific units for speech. We need to implement a function analogous to their IoU, but for our TTS system. To do this, we will first need to design a function to capture the desired properties.

Chapter 4

The IoU

4.1 Creating our IoU

As we discussed in Subsection 1.4.2, the IoU is the model used to quantify the agreement between semantic concepts and units. We will not recap it any further here, but it is important that we understand the idea behind it so that we can adapt it to create a new model applicable to our vocoder GAN. Therefore, the reader may wish to refresh themselves on the information at the start of Subsection 1.4.2.

We also want our IoU to quantify the agreement between semantic concepts and units. The difference is, we no longer have a notion of position in our generated data since the speech from each audio file we generate's properties only depend on the speaker we condition on. The properties we have are the ones labelled in the VCTK data set: accent, gender and age. Rather than having a segmentation algorithm to identify whether the generated data x has property c at position p, we can instead check which speaker we are conditioning on in the generated data x and use the information from the VCTK data set to see if the speech exhibits property c. We can then define the function $s_c : (x) \to \{0, 1\}$, where an output of 1 indicates that the generated data exhibits property c and an output of 0 indicates otherwise. Lastly, in a similar way to the image generation model, denote the activation of the unit u for generated audio data x by $a_u(x)$ and the top 1% quantile level of a_u by t_u. That is, writing $\mathbb{P}_x[\cdot]$ to indicate the probability that an event is true when sampled over all generated audio data, we define the threshold $t_u \equiv \max_t \mathbb{P}_x[a_u(x) > t] \geq 0.01$.

This allows us to define the IoU for our problem as

$$\text{IoU}_{u,c} = \frac{\mathbb{P}_x[s_c(x) \wedge (a_u(x) > t_u)]}{\mathbb{P}_x[s_c(x) \vee (a_u(x) > t_u)]}. \tag{4.1}$$

Hence, we see that a larger IoU would mean we have a larger numerator relative to

the denominator. The numerator represents how probable it is for our speech to have the property c and the unit u to activate above its top 1% quantile level. Whereas, the denominator represents how probable it is for our speech to have the property c or the unit u to activate above its top 1% quantile level. Thus, we can see how our IoU models the notion of agreement between a unit and a concept for our TTS networks.

For example, in the data we use to calculate the IoU, if a unit u only activates above its top 1% quantile level when generating speech with the property c and it always activates above its top 1% quantile level when generating speech with that property, then $\text{IoU}_{u,c} = 1$. On the contrary, in the data we use to calculate the IoU, if a unit u never activates above its top 1% quantile level for a property c despite there being speech with that property then we will have $\text{IoU}_{u,c} = 0$. If we have some agreement between a concept and a unit's activation being above its top 1% quantile level, but not complete agreement like in the first example, then we will have $\text{IoU}_{u,c} \in (0, 1)$.

Now that we have defined and gained some intuition on the IoU for our problem, we can move on to implementing it.

4.2 Implementing our IoU

To avoid unneeded complexity, we will first implement the calculating of our IoU for one unit, before generalising the code to work with whole layers then, lastly, to work with the whole network. Bau et al. use the `tally_conditional_quantile` function to implement their IoU, but this involves using a segmentation algorithm to identify which pixels align with which visual concepts. We do not have a notion of position and, as such, we have to do this in a different way. Ultimately, we need to find a way to calculate the probabilities in Equation (4.1).

Suppose use our TTS code to generate n pieces of audio data which we store in a set Z. Let $X = \{z \in Z \ : s_c(z) \wedge (a_u(z) > t_u)\}$ and $Y = \{z \in Z \ : s_c(z) \vee (a_u(z) > t_u)\}$. Then we have

$$\mathbb{P}_x[s_c(x) \wedge (a_u(x) \geq t_u)] = |X|/n, \tag{4.2}$$

$$\mathbb{P}_x[s_c(x) \vee (a_u(x) \geq t_u)] = |Y|/n \tag{4.3}$$

and we can write

$$\begin{aligned} \text{IoU}_{u,c} &= \frac{\mathbb{P}_x[s_c(x) \wedge (a_u(x) \geq t_u)]}{\mathbb{P}_x[s_c(x) \vee (a_u(x) \geq t_u)]} \\ &= |X|/|Y|. \end{aligned} \tag{4.4}$$

Now the question is, if we run our TTS code n times how can we find $|X|$ and $|Y|$? This can be done by splitting the problem into three parts: finding which pieces of audio data caused the unit u to activate above its top 1% quantile level, finding which pieces of audio data contain speech with property c and comparing these files. We will begin by implementing a way to find which pieces of audio data caused the unit u to activate above its top 1% quantile level.

First, suppose that we want to generate n pieces of audio data and find which ones made the unit u activate above its top 1% quantile level. We can find these using the `tally_topk` function if we set the `compute` function to return the activations of the unit stacked columnwise. We will then be able to find the indices of the pieces of audio data that caused the unit u to activate above its top 1% quantile level by selecting the first $\lceil n/100 \rceil$ columns of the first row of the second tensor output by `tally_topk.result()`.

This can easily be extended to work with different quantile levels by selecting the indices from a different number of columns. If we want the indices of the audio data that cause the unit to activate above its top q% quantile level and n audio files are generated, then we select the first $\lceil q \times (n/100) \rceil$ columns.

If we want to increase the number of units we are working with so we are working with a whole layer we need to edit our `compute` function. We want it to retain the activations of the chosen layer and then for each run concatenate them eventually returning a 10×(number of units in layer) shaped tensor. Finally, we can select which units we want the speaker indices for by selecting specific rows of the second output tensor.

The second part of our problem was to find which pieces of audio data we generate contain speech with property c. As we saw, `tally_topk.result()` returns indices corresponding to pieces of audio data. We need to find the properties of the speech contained in each of these pieces of audio data from their indices. This is fairly simple when we are only conditioning on each of our speakers once. We can set the TensorDataset input of `tally_topk` to `TensorDataset(torch.linspace(0, 107, 108))` and we can create a set containing the indices of the speakers with property c by using the information from the VCTK data set which tells us which properties each speaker has. In this case, the pieces of audio data with property c are just the

audio data indexed in that set. Creating these sets for different properties, we see that the accents and gender tend to be easily identifiable in the audio data we produce, however age much less so. As such, we decide not to use age as a concept any more since if we cannot tell a speaker is sounding older or younger, we cannot tell if our code is working when we edit the activations.

Finding which of the returned indices corresponds to which speaker becomes trickier when we start running TTS with each speaker more than once. Nonetheless, it can be done by inputting the data in an ordered way. For example, if each of our 108 speakers are conditioned on k times we could input the data to `tally_topk` such that we condition on our first speaker k times, then condition on our second speaker k times, and so on. This would mean that if the mth speaker has a property, the $mk + j$th pieces of audio data will have this property, for all $j \in \{0, \ldots, k-1\}$. To implement this, we work with our set of indices of speakers with property c. We convert it to a NumPy array, multiply it by k and create $k-1$ more NumPy arrays by adding the integers from 1 to $k-1$ to this array. Lastly, we convert each of these back into a set and find the union of them, which will be the indices of all the audio files with property c.

The last part of our problem is to compare the sets of indices we obtain. We saw that for any unit u we can create a set of indices for pieces of audio data that cause the GAN to activate above the chosen quantile level. Similarly, we can create a set containing the indices of the audio files with property c. We can then compute $|X|$ or $|Y|$ for a unit u by finding the intersection or union of these sets, respectively, and using the built-in `len` function to find the number of elements in each of these sets. Finally, dividing $|X|$ by $|Y|$ gives the IoU for the unit u and concept c. Our code to do this can be seen in the GitHub repository [4].

Initially, we test our code with the 108 speakers we have, conditioning on each of them once then trying to calculate the IoU of one unit. In this case, using a quantile level of 1% in our IoU we learn very little as we only get a non-zero IoU if the speech has the property c and it is the one of the two pieces of audio data that will cause the unit u to activate above its top 1% quantile level. This is very rare so the IoU is 0 for most concepts and units, in addition, for a given property, the IoU can only take two other values. Thus, we realise that it may be worth changing this to a 5% quantile level if we only want to run the network 108 times so that we can get more information from our IoUs.

To calculate the IoU for each of the units in a layer with concept c, we need to loop over our previous code for all the units in a layer. To tell us which of the

generated audio files have property c, we use the same set of indices we would if we were working with a single unit. Thus, we need to find the indices of the pieces of audio data that cause the highest activations for each unit. As mentioned earlier, this is done by selecting the first few columns of the `tally_topk.result()` output. We can then apply set operations to calculate the IoU for each unit individually.

Running the code to do this, we find that calculating the IoUs for one layer of our model takes around 30 minutes when using Colab Pro. Therefore, this is not feasible to run for all 73 activation layers of HiFi-GAN multiple times. It may be possible to run our code on the departmental machines, however we will likely run into the same problems as when we initially started this project with the incorrect dependencies causing errors. This could take days to fix. Instead, we will attempt to optimise the code, shown in Listing 4.1.

Listing 4.1: Initial code for calculating the IoUs of a layer.

```
IoU_list = []
for i in range(0, topk.result()[1].shape[0]):
  top = topk.result()[1][i][0:5]
  top = set(top.tolist())
  num = top.intersection(southern_units)
  den = top.union(southern_units)
  IoU_list.append(len(num) / len(den))
```

The first thing we need to deduce is which part of the loop is taking the longest to run. This loop iterates over every unit in the layer, thus the body is usually run tens or hundreds of thousands of times. This means a small decrease in running time of the body will cause a much larger difference overall. Changing the quantile level we use from 5% to 1% means we only select the first two elements of each row rather than the first six. If it is the set operations that are taking a long time to run, we believe this may decrease that. This however does not cause a significant decrease in running time. Instead, we decide to cast the whole tensor `topk.result()[1]` to a list before the for-loop rather than doing it for the selected elements in each row. This massively decreases the running time for each layer: previously it tended to be around 30 minutes, now it tends to be around 3 seconds.

Lastly, we test if the IoUs we generate are correct by picking units, using `tally_topk` to find the indices of the pieces of audio data that cause these units to activate in their top 5% quantile and calculating their IoU by hand. Comparing this with the results from our code shows us that everything is working correctly. Now we just need

to scale this up one last time so we can work with all the activations in the network.

Initially, we attempt to use our `tally_topk` function once to do this by having the `compute` function retain all the activations for each run and arrange them into rows. This would allow us to find the indices of the audio data that cause the highest activations for every unit in our vocoder GAN. Unfortunately, working with this much data at once causes our Colab session to crash due to a lack of RAM. As an alternative, since we know that we can run `tally_topk` for each of the individual layers, we create a loop that calculates and returns the IoUs of each layer one-by-one. Inside the loop we define the `compute` function, run `tally_topk` and apply the set operations for all the units in that layer. This allows us to calculate the IoUs for an entire network, but with our current experimental set-up of Conformer-FastSpeech 2, HiFi-GAN and an `input_feat` length of 100 it takes around 2 hours to run. This is not ideal if we want to dissect the network for many different properties and using many different parameters. Nonetheless, it is a huge step that we can now apply the network dissection method to a whole network. We test this by comparing some of the IoUs this calculates with some of the IoUs calculated by the code which only runs one layer, since we know these IoUs are correct. This indicates that our IoUs for the whole network are being calculated correctly.

The last thing we do before we move on is create a way to store our IoU data so we can use it later. Bau et al. normally only edit 20 units at a time, thus we save the indices and IoUs of the 20 units with the highest IoUs for each activation layer. We do this by creating a Pandas DataFrame, and save this data for future use in a CSV file.

4.3 Changing Activations Using the IoU

We now have code which allows us to dissect a network, but it is important that we understand a few things about the architecture of network we are dissecting. As we mentioned earlier, we are using the HiFi-GAN vocoder. The main part of the architecture of this is shown in Figure 4.1 with their also being a convolutional layer before what is shown and two activation layers and a convolutional layer after what is shown. The specific parameters of the model are given in the paper [17] as model V1. Each residual block contains a sequence of activation and convolutional layers, with some residual connections between layers. The residual connections work by taking the output of a previous layer and adding it to the output of the current layer to give the input to the next layer. The important point for us here is that the residual

blocks are stacked in 4 groups of 3. That is, the same tensor is input to 3 residual blocks at a time. The outputs of these blocks are then added together, a transposed convolution is applied to this and the output is fed into another group of 3 blocks. This process happens for once for each group of blocks, until the output of blocks 9, 10 and 11 are added and input to the final activation and convolutional layers. Given that the blocks are stacked in groups of 3, blocks 3, 4 and 5 contain layers later in the network than blocks 0, 1 and 2, and so on. This will be important for understanding distribution of property-specific units throughout the network.

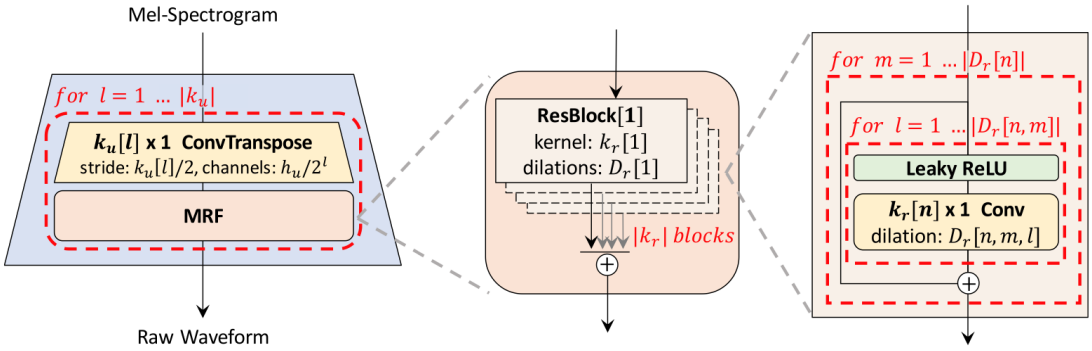


Figure 4.1: Part of the architecture of HiFi-GAN [17].

In the paper [3], they usually activate 20 units at a time to produce a change in their generated output. Given that the HiFi-GAN we use tends to have over a million units, we decide to test whether we can only activate the 20 units with the highest IoU in a layer and get an audible difference. We cannot no matter what layer we choose. Instead, we get an audible difference by modifying the activations of a similar proportion of the units in the layer to Bau et al., around 5% of the units. Here, we did this by setting the activations to 0 or 1, but to follow Bau et al.'s method we need to set each of the activations to their 1% quantile level. We do this by writing code which calculates the IoUs for all units in a given layer for a given concept, finds the indices of the units with the highest IoUs then sets each of their activations to their 1% quantile level using the `tally_quantile` function. The code to do this can be seen in 'billiespnet/billys-notebooks/layer+edit.ipynb' in our GitHub repository [4].

Now, we have code such that we can apply network dissection to our vocoder GAN, find the important layers and edit the activations within them. In order to utilise the code effectively, we will design some experiments and make hypotheses about the outcomes.

Chapter 5

Results

5.1 Experiments and Hypotheses

The goal of our experiments is to see which of the aims set out in Section 1 we have achieved. When we have dissected a network the first thing we tend to do is create a plot of the proportion of units that are property-specific in each block against the blocks, in a similar way to Figure 1.3c. In the paper [3], they can use the number of units in the layer rather than the proportion as there is the same number of units in every layer, but for us it makes more sense to do it this way. We use a threshold on the IoU to decide whether units are property specific. Picking the threshold such that we have a few hundred property-specific units lets us see their distribution clearly. The plot then allows us to see where the in the network has the greatest proportion of property-specific units for each property and therefore where to act if we want to alter a property of speech. More so, seeing the distribution may also be a good indication that the properties are transferable, as we would expect the units to be distributed throughout the layers in a similar way to that in Figure 1.3c. Therefore, we expect a peak in the first three blocks for this plot. For a given property, if we see no pattern at all or a very uniform distribution it suggests that we have found some units with high IoUs but that are not actually important to this property. This can happen when we have not got enough activation data for speakers with this property as the denominator in Equation (4.1) will be small. A unit will only need to activate highly when generating speech with this property a few times for the numerator to be relatively large, resulting in a large IoU and therefore indicating this unit is specific to this property. We generate millions of activations for each run of our GAN generator, thus there exist high-IoU units which are not really specific to that property but

were just activating highly when generating speech with that property by chance. Calculating the IoUs using more activations, and therefore more runs of the GAN generator can prevent this happening as much.

To test the IoU is locating units that activate highly when certain properties are present, we can find high-IoU units then create jitter plots, in a similar way to Figure 1.3f. Plotting the activation of a unit for runs of the network when the property is present and when it is not shows if there is any correlation between certain units and certain properties. However, a correlation here does not show that these units activating highly causes this property to be present, only that units tend to activate highly when this property is present.

We also want to see if we achieved the objective we set in Section 2.1: to be able to change a property of one speaker's speech to be the same as another speaker's while keeping all the other properties constant. Furthermore, we want to see if we can continuously modify the prevalence of properties in the speech. Since the reader cannot hear the audio we generate, our aim was to find a measure to quantify the difference in people's voices. We could then change properties of our speakers' voices and use the measure to determine if they were getting more or less similar. Unfortunately, we could not find a measure for this. Instead, if we are altering the activations of the n most property-specific units in a layer, we will also run another test altering n random units in the same layer. We can then compare the audio side-by-side and to speech conditioned on the same speaker with no modifications to see whether our method alters the speech as we hope. We will also inform the readers of relevant notebooks as we go through such that they too can alter and listen to the generated audio.

Ultimately, we want to facilitate the best property transfer and we have multiple variables we can change to achieve this. We have lots of properties labelled, such as accents and genders. Some of these may transfer better than others. For us, we hypothesise that the properties that will transfer the best will be the ones which we have the most speakers for, as we explained in Section 2.1. Other choices include the length of the Mel spectrogram we input to our vocoder and the text we condition on. Ideally, we would be able to generate lots of speech by using long sentences and synthesising many different sentences so properties would be able to reveal themselves in the generated speech. Unfortunately, we are limited by the amount of RAM we have so we found some sentences which are meant to reveal accents and will condition on these [27]. When cutting the length of our input Mel spectrogram such that we only synthesise around 6 words and applying our network dissection method, we can

still only generate speech conditioning on each speaker twice before running out of RAM from storing the activations. As such, we will not have a large amount of activation data to calculate the IoUs with. To account for this we will first try to learn a property of our generated speech which is simpler than accent or gender as this should require less activation data. Once we know our method works we can then apply it to more challenging problems.

5.2 Changing Artificial Properties

To the best of the author's knowledge, applying network dissection to a TTS system and using its output to modify the activations of units has never been done before. Therefore, to ensure it is possible, we will start by trying to change a simple property in our generated audio. The only properties we have labelled are age, accent and gender, but we want something even easier to identify when listening to the synthesised speech. To create this, we alter our Mel spectrogram. Initially, we thought that increasing the volume or frequency of the synthesised speech would be relatively easy to do by acting on the Mel spectrogram which is input to our vocoder GAN. For example, we multiplied all the entries in the Mel spectrogram by a scalar attempting change the volume of the synthesised speech. However, as the vocoder GAN is not trained on Mel spectrograms representing speech of that volume, we did not get the desired effect. We instead decide that an easier property to identify would be if we added an electrical fuzzy sound to the end of our synthesised audio. This can be achieved by adding a few rows of zeros at the end of our Mel spectrograms.

We set up our experiment by setting the length of `input_feat` to 100 and the last 20 rows of it to be all zeros for around a third of our speakers. We then create a set of the indices of the pieces of audio data that will have this property. Each of our speakers will be conditioned on once and will say 'Please call Stella. Ask her too.'. We will refer to this as sentence 1. We also define sentence 2: 'Bring these things with her from the store.', so that we can see if we can transfer the property to sentences which we have not collected activation data on.

Running network dissection and plotting the proportion of units that are property-specific in each block against the blocks we see a clear pattern, but not the one we were expecting. This is shown in Figure 5.1. We expected the peak to be in the earlier layers of the network but instead it is in the later layers. This may be due to the hierarchical nature of CNNs in that the later layers' outputs can only effect local regions of the network output, whereas earlier layers outputs effect much more of the

network output. This is due to the way receptive fields of units combine, as discussed in Subsection 1.2.1. This feature only occurs at the end of the generated audio, hence to produce it we would only want to act locally. Thus, it makes sense to have a lot more property-specific units in the later layers. This trend may be different when we work with the more complex speech properties which effect all of the generated audio.

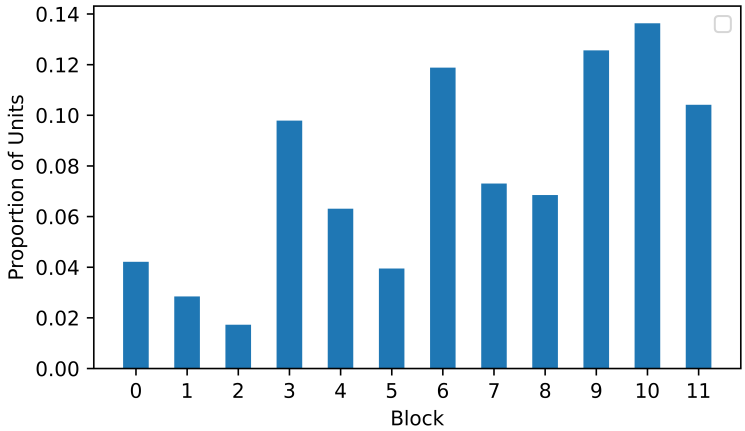


Figure 5.1: The distribution of the property-specific units across the blocks for our artificial property. We used a threshold IoU of 0.16 to give the property-specific units.

Randomly selecting one of the units with the joint-highest IoU in the network (unit $[0, 0, 23970]$ in layer 'blocks.9.convs1.1.0'), we recorded its activation for a number of different runs of our TTS system and plotted them in Figure 5.2. As we said earlier, we only ran the network dissection code with activations collected by conditioning on each speaker once and we altered the Mel spectrogram to add the feature to around a third of these speakers' speech. We call the speakers we added the feature to then the 'chosen speakers', and we call the rest of the speakers the 'other speakers'. However, when collecting the activations for Figure 5.2 we do not always add the feature to these same speakers' speech: whether or not we add the feature is labelled on the x-axis. Lastly, sentences 1 and 2 are still as we defined them at the start of this section.

Figure 5.2 shows us that our IoU works well as a model to quantify how important units are to given properties. We can clearly see that the unit we identified from its high-IoU tends to activate more highly when it is generating a sentence with the feature present. We see that the activations of this unit tend to be higher when generating sentence 2 with the feature present than when generating either sentence without the feature present. Nonetheless, these activations still tend to be less than when generating sentence 1 with the feature present. This suggests that although

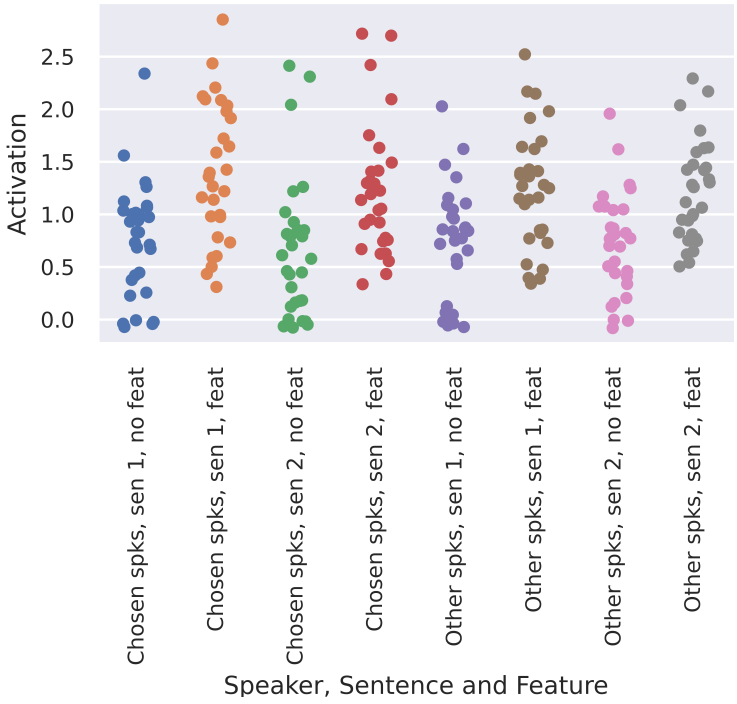


Figure 5.2: The activations of a unit with the joint-highest IoU for the artificial property.

there is some sharing of important units for different sentences with the same property, there also exist units which are important to specific sentences with the property but not to other sentences with the same property. Thus, to find units which are specific to a certain property we would have to collect activations over numerous sentences. Otherwise, we may be altering units which are important to a sentence which we are not generating. This would likely effect the quality of the property transfer. The last thing we note about Figure 5.2 is that the unit can still activate relatively lowly and audio be generated with our artificial feature, even when generating sentence 1. This suggests that other units are also important in generating audio with this feature, which agrees with us having thousands of units with the same IoU for this property.

Now, we will test how well the artificial property transfers. Whenever we modify the activation of a unit in this dissertation, we will set it to its 1% quantile level unless stated otherwise. We run the TTS model to generate speech for sentence 1 while modifying the activations of the 5000 units in the final activation layer of our GAN generator with the highest IoUs. This produces audio that contains a sound similar to the electrical fuzzy noise at the end and with a slight fuzzy hum over the speech throughout. Decreasing the number of chosen units to 1000 works even better,

eradicating the fuzzy noise over the speech such that we only have a fuzzy noise at the end of the audio. This suggests that picking too many activations to modify is also detrimental to property transfer. This makes sense as the more units you pick to change, the less specific they will be to the property so eventually they will not be important to the property whatsoever. We therefore want to pick enough units to get good speech transfer, but not too many. When we alter the activations of same number of units in the layer but select random ones we get a fuzzy hum throughout similar to the one we get from picking the 5000 units with the highest IoUs. This therefore backs up our point that the units' activations we are changing are less specific to this property when we pick too many. Picking to modify the units with the highest IoUs but less than 1000 of them, we see that the property still transfers just less so. Hence, we have achieved our aim of being able to continuously modify properties of audio by varying parameters. If the readers would like to test this out for themselves, the notebook to do this is given as 'billys-notebooks/artificial.ipynb' in our GitHub repository [4].

By repeating this experiment for other activations layers in the network, we find that the property does not transfer as well when we modify earlier layers, as we predicted from the trend in Figure 5.1. In general, the earlier the layer we pick, the more muffled the effect becomes until eventually it does not seem to be transferring at all. However, we still get good property transfer modifying layers in block 6 onward. We again find that the best property transfer tends to be when we alter the activations of around 1% to 5% of the units in the layer.

Modifying the activations and synthesising sentence 2 we find that we can still transfer the property. The transfer does not tend to be quite as good as for sentence 1: there tends to be a slight fuzzy hum over the whole sentence. This is precisely what we expected to happen after seeing Figure 5.4. Nonetheless, this shows that we can also transfer properties to sentences we did not collect activations for when dissecting the network. This suggests that we will not have to collect activations for every possible sentence we want our TTS system to be conditioned on, we just need enough activation data for it to learn the property.

Working with the artificial property has shown us numerous things. We have seen that the IoU we defined accurately quantifies the importance of units to a property and, in doing so, disentangles this property from others. We have shown that the network dissection method can also be applied to audio generation to produce analogous results to image generation [3]. Namely, we can transfer concepts identified in certain generated data to other generated data by editing the activations of units.

More so, we can vary how prevalent this concept will be in the output data by acting on more or fewer units. Now that we have established this is possible, we will attempt to apply the same techniques to learn the more complex properties of speech.

5.3 Changing Properties of Speech

In this section, we are going to attempt to alter the accent and gender properties of the speech we are synthesising. There are a large number of variables we can change when doing this, so we will investigate how changing these affects the transfer of properties and try to optimise these to best facilitate the transfer. For our initial experiment we cut or extend the length of the Mel spectrogram `input_feat` to 100. We calculate the IoUs using a 5% quantile level and we condition on each speaker once having them say the sentences 'Please call Stella. Ask her too.'. Now, we have our base set of variables we can start conducting experiments to optimise them.

Again, the plot of the proportion of units that are property-specific in each block against the blocks of HiFi-GAN should give a good indication of which properties to alter and which blocks to alter layers in. Unlike the paper by Bau et al. [3], we do not have the computing power to calculate the IoU for all units in the network with all the labelled properties we have. Instead, we will just select 5 properties: the Belfastian accent, the Southern accent, the South African accent, a male voice and a female voice. We chose these properties as they are easily identifiable audibly and each of them has at least 4 speakers. Figure 5.3 shows the plot of the proportion of units that are property-specific in each block against the blocks of HiFi-GAN. Here, we choose a threshold IoU of 0.5 as what classifies as a property-specific unit, such that we can see the distribution of the 100 or so units with the highest IoUs. The high-IoU units are fairly evenly distributed throughout the network, although it does seem like there are slightly more towards the earlier layers as we hypothesised in Section 5.1. Furthermore, note that neither the male nor the female property has any units with IoU greater than 0.5 for this experiment. This suggests that the IoUs of the high-IoU units seems to correlate somewhat with the number of speakers we have for the property. This is likely due to randomness having less of an effect, as discussed in Section 5.1. We would therefore need to collect more activations for each property to negate these probabilistic effects.

Before we do this, let us check if our IoU still captures the importance of units to a given concept. We will use the sentence 1 and sentence 2 that we defined in Section 5.2 to do this. Figure 5.4 shows the activations of a unit with the joint-highest IoU for the

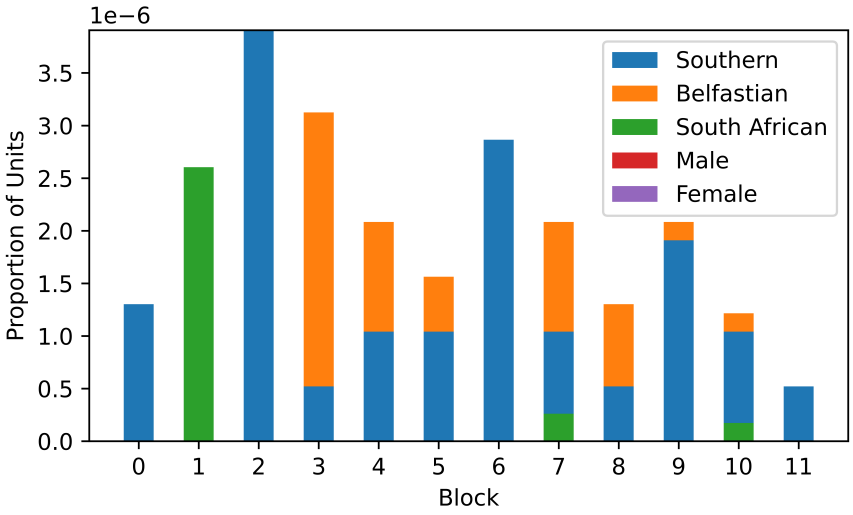


Figure 5.3: The distribution of the property-specific units across the blocks for when the length of our Mel spectrogram `input_feat` is 100. We used a threshold IoU of 0.51 to give the property-specific units.

Southern accent, unit $[0, 29, 6261]$ in layer 'blocks.6.convs1.1.0'. Clearly, we can see that the unit tends to activate more highly when conditioned on a Southern speaker and sentence 1. As opposed to when we were working with the artificial feature, the selected unit no longer tends to activate highly when conditioned on the chosen property and sentence 2. This is likely due to this property being more complex than the noise at the end of each audio clip we investigated in Section 5.2 and manifesting itself in different ways each time we generate audio. To account for this and have units which tend to activate highly when we synthesise general Southern speech, we will need to use a wider variety of sentences when collecting the activations we will use to dissect our network. Ideally, this would mean generating speech multiple times for each speaker.

Before we modify any variables, let us see how well we can alter speech properties with our current ones. Then, we can compare if other combinations of variables allows us to alter these properties any better. We see that there is the highest proportion of units specific to any property in block 2 with units specific to the Southern accent. This implies that this will be the best property and block to alter activations in to facilitate high-quality property transfer. Increasing the activations of any number of the high-IoU units does not give any speaker we condition on a Southern accent: it either has no effect or makes the voice more muffled and adds reverb. The readers can hear this for themselves using the notebooks in [4]. This is the same for any

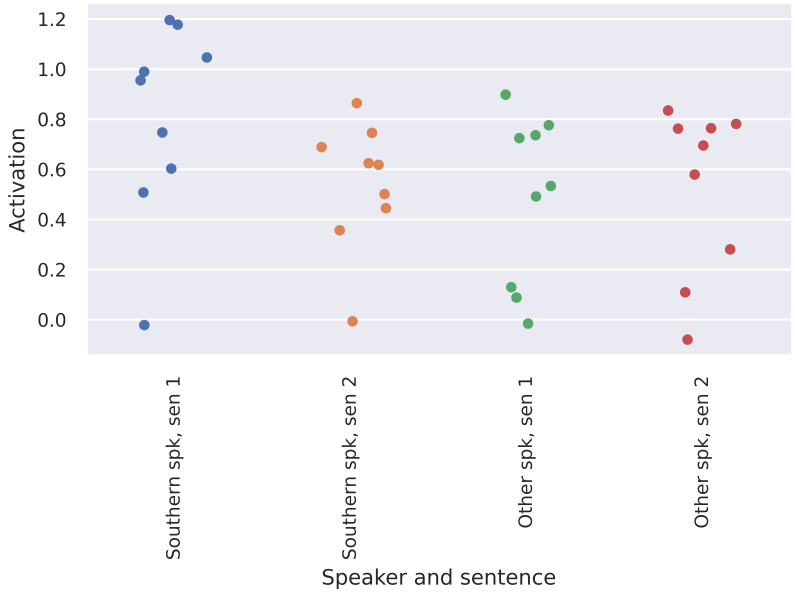


Figure 5.4: The activations of a unit with the joint-highest IoU for Southern speakers.

property and layer we use, even the gender properties which we have lots of speakers for. One thing we can say is that we can continuously modify the extent of the effect by altering the number of units we act on.

As opposed to Bau et al.'s paper in which they only either activate or deactivate layers each time they run their network, we hypothesise that we might have to do both at once. This is because we may be layering properties on top of one another, for example trying to add a Southern accent to a South African accent. We therefore add code which acts on a given layer by setting the activations of the units with the highest IoUs for a given property to zero. This can be seen in 'billiespnet/billys-notebooks/layer+edit.ipynb' in our GitHub repository [4]. Unfortunately, this did not help either with it instead combining two non-distinctive effects which depend on the properties and layers we choose to change. To get better results it seems necessary to use more activation data when dissecting the network.

To collect more activation data, we first choose to increase the length of our Mel spectrogram. We alter our code such that the length of `input_feat` is always 200 rather than 100 and we condition on the text 'Please call Stella. Ask her to bring these things with her from the store.'. Plotting the proportion of units that are property-specific in each block against the blocks, we find that the proportion of property-specific units increases towards the later layers of the network. This is not what we would have expected, however it is similar to when we had good property transfer for our artificial feature. Testing to see how well properties transfer, we find

it is very similar to when we used a shorter Mel spectrogram in that we tend to generate non-descriptive muffled and reverb sounds on top of our speech. There is a similar effect for any combination of the properties, layers or number of altered units we select.

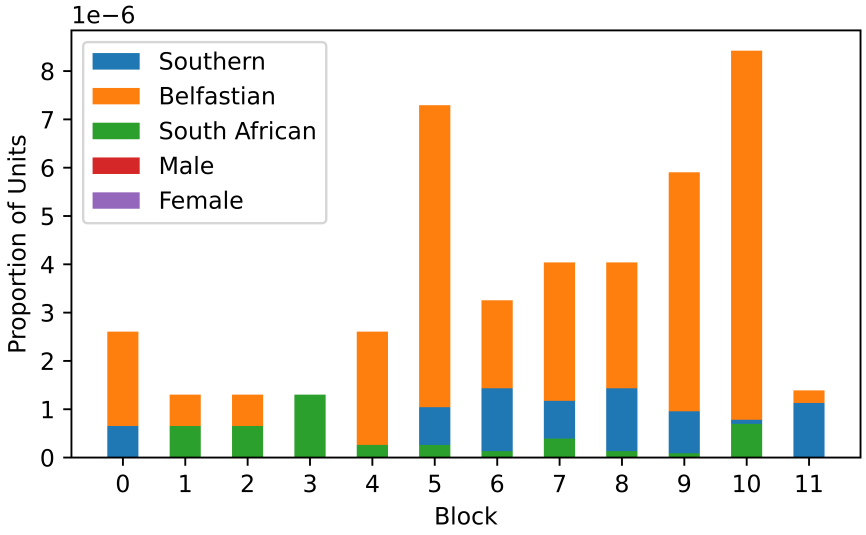


Figure 5.5: The distribution of the property-specific units across the blocks when the length of our Mel spectrogram `input_feat` is 200. We used a threshold IoU of 0.51 to give the property-specific units.

Instead, we hypothesise that having more activation data for each individual unit may reduce the probabilistic effects causing units which are not property-specific to activate highly. Furthermore, previously units may have activated highly due to being important to certain parts of the sentence and not important to the accent or gender. Collecting more activations for speakers saying multiple sentences should therefore make the identified units more specific to the accent rather than being specific to the accent and the sentence. As such, we revert the length of `input_feat` to 100 and condition on each of the speakers twice to generate our activation data. This requires two pieces of text to condition the speakers on. We choose 'Please call Stella. Ask her too.' and 'Bring these things with her from the store.'. We would have liked to condition on each of the speakers more than once as we suspect this is important to achieving good results, but we are limited by the amount of RAM we have. Nonetheless, we have added the number of times we condition on each of the speakers as a parameter to the code, such that with more RAM we can easily run more tests. This code can be found in 'billiespnet/billys-notebooks/IoU_multi.ipynb' in our GitHub repository [4].

Figure 5.6 gives the plot of the proportion of units that are property-specific in each block against the blocks when we condition on each of the speakers twice. This time we get the pattern we have been expecting: the proportion of units peaks in the earlier layers of the network and decreases as we get towards the later layers. We believe this is because we are now finding units specific only to the accent rather than to the accent and the sentence. Given the hierarchical nature of CNNs, the earlier layers cause the smaller sounds which make up the words whereas the later layers will make larger sound. This evidence backs up that idea and gives a similar pattern to the one Bau et al. gave for their image generation network in Figure 1.3. This indicates that our method is now locating property-specific units more accurately. Unfortunately, trying to change properties of our speakers' speech still gives similar sounding speech to that generated in the two previous examples, so it is unclear if we have actually made any improvements here.

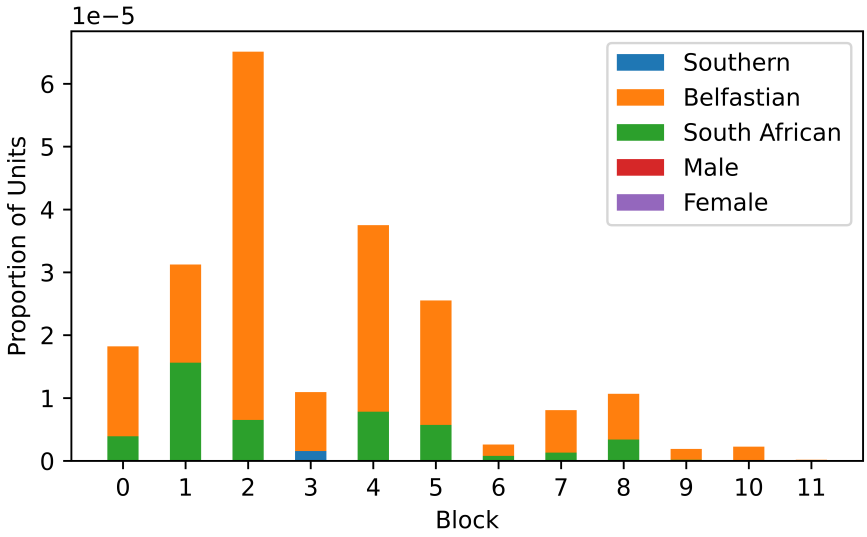


Figure 5.6: The distribution of the property-specific units across the blocks when the length of our Mel spectrogram `input_feat` is 100 and we condition on each speaker twice. We used a threshold IoU of 0.3 to give the property-specific units.

The last variable we want to adjust is the quantile level we use in the IoU. We continue with the current set up but change the quantile level in the IoU from 5% to 2%. This gives us a similar plot to Figure 5.6 when we locate the high-IoU units and similar effects when we try to transfer speech properties. Hence, despite trying many combinations of variables, we could not achieve the same quality of property transfer as we did for our artificial property.

Chapter 6

Conclusion

In this dissertation, we investigated the feasibility of applying network dissection to TTS systems to allow us to alter properties of the synthesised speech. We began by discussing the relevant background information. This included convolutional neural networks, GANs and different ways to represent audio. Once these were clear, we explored the network dissection method introduced by Bau et al., focusing on the theory behind it and the results on image generation tasks.

We identified some appropriate TTS models with pre-trained vocoder GANs as part of the ESPnet package [26]. Furthermore, the GANs were trained on large multi-speaker data sets which relinquished the need for more training. The properties of the speakers were labelled too, which enabled us to identify them easily.

We modified the network dissection software which was used for image generation to work with the TTS models given in the ESPnet package. This was made particularly challenging due to their being many interdependent functions within the code and a lack of documentation on the code. Numerous functions defined within the network dissection software were modified. In doing this, we realised that the pre-trained vocoder GANs we were using did not act on every input the same way. Through testing we identified that the widths of the layers of the GAN were dependent on the length of the input Mel spectrogram. Investigating further, we found that only two of the GANs were deterministic, one of these being HiFi-GAN.

In order to use the code we had adapted, we redefined the IoU such that it was applicable to our TTS system and implemented code to calculate the IoU for all the units in HiFi-GAN. Then, we implemented code to automatically increase the activations of the units with the highest IoUs to their 1% quantile level.

To test our method, we created an artificial property within some of our audio data. Applying our network dissection method to the activations produced, we saw

that our IoU modelled the agreement between semantic concepts and units by using jitter plots. High-IoU units even captured when properties were present for sentences we did not have activation data for. Most importantly, modifying the activations of these units allowed us to reproduce this property in other generated audio.

When working with real properties of speech, we found that our IoU still captured the agreement between semantic concepts and units, but only in certain cases. For example, if we collected activation data by conditioning on the same sentence for every speaker, the important units we identified would not tend to activate highly when we generated other sentences. When we collected activations conditioning on each speaker more than once, the distribution of property-specific units throughout the network was as expected.

Modifying the activations of the property-specific units for actual properties of speech did not produce the desired effects. This is likely because accents and gender can manifest themselves in different ways depending on what is being said, whereas the artificial property manifested itself similarly in every piece of generated audio making it easier to identify. The issue was the identification of the property-specific units, but due to time and resource constraints we were not able to fix this. Nonetheless, given that we could transfer the artificial property, there is reason to believe we could also transfer the accent and gender properties.

Returning to the aims we set out at the start, we saw that we were able to disentangle artificial properties from our audio data, but not accent or gender. We were able to continuously modify any of the properties disentangled by altering more or fewer activations. When we were modifying activations relating to the artificial property, we were able to keep the rest of the speech very similar, if not the same.

Hence, we have shown that network dissection and using its outputs to change properties of generated audio is possible, which had not been done before. The code we have written provides a great platform for further research into the subject and could even be adapted to work with other TTS models. With the resources my industrial partner, Amazon, have they should be able to get a clearer picture of whether using network dissection to transfer actual speech properties is possible. A method to do this could be to condition on each speaker several times, apply network dissection and use the results to modify the activations of the property-specific units. To do this they would need more RAM than we had.

Possible future research directions include applying network dissection to a network which takes text as an input and produces audio as an output. Similarly, it would be interesting to apply our method to a text-to-Mel network.

References

[1] D. Bau, *global-model-repr*. `https://github.com/SIDN-IAP/global-model-repr`, 2022. [Commit: 8b5901be7f581b6c3a63596b73658cfe9e532a73].

[2] D. Bau et al., *Gandissect: Pytorch-based tools for visualizing and understanding the neurons of a gan*. `https://github.com/CSAILVision/gandissect`, 2022. [Commit: f55fc5683bebe2bd6c598d703efa6f4e6fb488bf].

[3] D. Bau, J.-Y. Zhu, H. Strobelt, A. Lapedriza, B. Zhou, and A. Torralba, *Understanding the role of individual units in a deep neural network*, Proceedings of the National Academy of Sciences, 117 (2020), pp. 30071–30078.

[4] W. Braithwaite, *Billiespnet: End-to-end speech processing toolkit*. `https://github.com/billybraith17/billiespnet`, 2022. [Commit: ad6c7c8cafa43b66db6203977b821c50043f1092].

[5] D. Gartzman, *Getting to know the mel spectrogram*. `https://towardsdatascience.com/getting-to-know-the-mel-spectrogram-31bca3e2d9d0`, 2022. [Online; accessed 26-August-2022].

[6] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016.

[7] 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 (NIPS), vol. 27, 2014.

[8] A. Gulati, J. Qin, C.-C. Chiu, N. Parmar, Y. Zhang, J. Yu, W. Han, S. Wang, Z. Zhang, Y. Wu, and R. Pang, *Conformer: Convolution-augmented transformer for speech recognition*, in ISCA Interspeech Conference, 2020, pp. 5036–5040.

[9] T. Hayashi et al., *Espnet: End-to-end speech processing toolkit*. https://github.com/espnet/espnet, 2022. [Commit: e5d133c21800116b2fc5844859333ef83163d293].

[10] T. Hayashi et al., *Parallel wavegan*. https://github.com/kan-bayashi/ParallelWaveGAN, 2022. [Commit: 1f7949f593cc5600478cd4cc23bbf34bbcb0bcff].

[11] T. Hayashi, R. Yamamoto, K. Inoue, T. Yoshimura, S. Watanabe, T. Toda, K. Takeda, Y. Zhang, and X. Tan, *ESPnet-TTS: Unified, reproducible, and integratable open source end-to-end text-to-speech toolkit*, in IEEE International Conference on Acoustics, Speech and Signal processing (ICASSP), 2020, pp. 7654–7658.

[12] C. F. Higham and D. J. Higham, *Deep learning: An introduction for applied mathematicians*, SIAM review, 61 (2019), pp. 860–891.

[13] L. Holzbauer, *Convolutional neural networks explained... with american ninja warrior*. https://blog.insightdatascience.com/convolutional-neural-networks-explained-with-american-ninja-warrior-c6649875861c, 2022. [Online; accessed 26-August-2022].

[14] P. Karanasou, S. Karlapati, A. Moinet, A. Joly, A. Abbas, S. Slangen, J. Lorenzo-Trueba, and T. Drugman, *A learned conditional prior for the VAE acoustic space of a TTS system*, in ICSA Interspeech Conference, 2021, pp. 3620–3624.

[15] T. Karras, T. Aila, S. Laine, and J. Lehtinen, *Progressive growing of GANs for improved quality, stability, and variation*, in International Conference on Learning Representations (ICLR), 2018.

[16] B. Kim, J. Gilmer, M. Wattenberg, and F. Viégas, *TCAV: Relative concept importance testing with linear concept activation vectors*, arXiv preprint arXiv:1711.11279, (2017).

[17] J. Kong, J. Kim, and J. Bae, *HiFi-GAN: Generative adversarial networks for efficient and high fidelity speech synthesis*, in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 17022–17033.

[18] A. Koul, A. Fern, and S. Greydanus, *Learning finite state representations of recurrent policy networks*, in International Conference on Learning Representations (ICLR), 2019.

[19] A. Mustafa, N. Pia, and G. Fuchs, *StyleMelGAN: An efficient high-fidelity adversarial vocoder with temporal adaptive normalization*, in IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2021, pp. 6034–6038.

[20] A. J. Oxenham, *How we hear: The perception and neural coding of sound*, Annual Review of Psychology, 69 (2018), pp. 27–50.

[21] Y. Ren, C. Hu, X. Tan, T. Qin, S. Zhao, Z. Zhao, and T.-Y. Liu, *FastSpeech 2: Fast and high-quality end-to-end text to speech*, in International Conference on Learning Representations (ICLR), 2021.

[22] M. T. Ribeiro, S. Singh, and C. Guestrin, *"Why Should I Trust You?" Explaining the predictions of any classifier*, in ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 1135–1144.

[23] J. Shen, R. Pang, R. J. Weiss, M. Schuster, N. Jaitly, Z. Yang, Z. Chen, Y. Zhang, Y. Wang, R. Skerrv-Ryan, R. A. Saurous, Y. Agiomvrgiannakis, and Y. Wu, *Natural TTS synthesis by conditioning WaveNet on mel spectrogram predictions*, in IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2018, pp. 4779–4783.

[24] A. van den Oord, N. Kalchbrenner, and K. Kavukcuoglu, *Pixel recurrent neural networks*, CoRR, abs/1601.06759 (2016).

[25] A. van den Oord, N. Kalchbrenner, O. Vinyals, L. Espeholt, A. Graves, and K. Kavukcuoglu, *Conditional image generation with pixelcnn decoders*, CoRR, abs/1606.05328 (2016).

[26] S. Watanabe, T. Hori, S. Karita, T. Hayashi, J. Nishitoba, Y. Unno, N. Enrique Yalta Soplin, J. Heymann, M. Wiesner, N. Chen, A. Renduchintala, and T. Ochiai, *ESPnet: End-to-end speech processing toolkit*, in ISCA Interspeech conference, 2018, pp. 2207–2211.

[27] S. H. Weinberger et al., *The speech accent archive.* `http://accent.gmu.edu/`, 2022. [Online; accessed 26-August-2022].

[28] Wikipedia contributors, *Gabor transform*. https://en.wikipedia.org/wiki/Gabor_transform, 2022. [Online; accessed 26-August-2022].

[29] T. Wood. Personal communication (dissertation meeting), (June 27, 2022).

[30] T. Xiao, Y. Liu, B. Zhou, Y. Jiang, and J. Sun, *Unified perceptual parsing for scene understanding*, in European Conference on Computer Vision (ECCV), 2018, pp. 418–434.

[31] J. Yamagishi, C. Veaux, and K. MacDonald, *CSTR VCTK Corpus: English multi-speaker corpus for CSTR voice cloning toolkit (version 0.92)*, 2019.

[32] R. Yamamoto, E. Song, and J.-M. Kim, *Parallel WaveGAN: A fast waveform generation model based on generative adversarial networks with multi-resolution spectrogram*, in IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2020, pp. 6199–6203.

[33] G. Yang, S. Yang, K. Liu, P. Fang, W. Chen, and L. Xie, *Multi-band MelGAN: Faster waveform generation for high-quality text-to-speech*, in IEEE Spoken Language Technology Workshop (SLT), IEEE, 2021, pp. 492–498.

[34] F. Yu, A. Seff, Y. Zhang, S. Song, T. Funkhouser, and J. Xiao, *LSUN: Construction of a large-scale image dataset using deep learning with humans in the loop*, arXiv preprint arXiv:1506.03365, (2015).

[35] M. D. Zeiler and R. Fergus, *Visualizing and understanding convolutional networks*, in European Conference on Computer Vision (ECCV), Springer, 2014, pp. 818–833.

[36] H. Zen, V. Dang, R. Clark, Y. Zhang, R. J. Weiss, Y. Jia, Z. Chen, and Y. Wu, *LibriTTS: A corpus derived from librispeech for text-to-speech*, in ISCA Interspeech Conference, 2019, pp. 1526–1530.